

Objective-C

КАРМАННЫЙ СПРАВОЧНИК

Objective-C

PHRASEBOOK

SECOND EDITION

David Chisnall

**DEVELOPER'S
LIBRARY**



Addison
Wesley

www.informit.com/developers

Objective-C

КАРМАННЫЙ СПРАВОЧНИК

ВТОРОЕ ИЗДАНИЕ

Дэвид Чиснолл



Москва • Санкт-Петербург • Киев
2012

ББК 32.973.26-018.2.75

Ч-67

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом “Вильямс”
по адресу:

info@williamspublishing.com, <http://www.williamspublishing.com>

Чиснолл, Дэвид.

Ч-67 Objective-C. Карманный справочник, 2-е изд. : Пер. с
англ. — М. : ООО “И.Д. Вильямс”, 2012. — 352 с. : ил. —
Парал. тит. англ.

ISBN 978-5-8459-1777-5 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc., Copyright © 2012

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2012

Научно-популярное издание

Дэвид Чиснолл

Objective-C. Карманный справочник

2-е издание

Литературный редактор *И.А. Попова*

Верстка *О.В. Мишутина*

Художественный редактор *Е.П. Дынный*

Корректор *Л.А. Гордиенко*

Подписано в печать 17.05.2012. Формат 70х100/32.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 11,0. Уч.-изд. л. 12,9.

Тираж 1500 экз. Заказ № 0000.

Первая Академическая типография “Наука”

199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1777-5 (рус.)

ISBN 978-0-321-81375-6 (англ.)

© Издательский дом “Вильямс”, 2012

© Pearson Education, Inc., 2012

Оглавление

Введение	13
Глава 1. Основные принципы Objective-C	19
Глава 2. Азбука Objective-C	35
Глава 3. Управление памятью	75
Глава 4. Шаблоны, распространенные в Objective-C	105
Глава 5. Числа	127
Глава 6. Манипулирование символьными строками	141
Глава 7. Работа с коллекциями	163
Глава 8. Даты и моменты времени	183
Глава 9. Работа со списками свойств	193
Глава 10. Взаимодействие с окружением	209
Глава 11. Доступ к значениям по ключам	219
Глава 12. Обработка ошибок	231
Глава 13. Доступ к файлам и каталогам	247
Глава 14. Потоки.....	263
Глава 15. Блоки и Grand Central Dispatch	277
Глава 16. Уведомления	289
Глава 17. Доступ к сети	299
Глава 18. Отладка программ на Objective-C	313
Глава 19. Динамическая библиотека Objective-C	325
Предметный указатель	341

Содержание

Об авторе.....	13
Благодарности.....	14
Введение.....	15
О версиях книги.....	16
От издательства	17
Глава 1. Основные принципы Objective-C	19
Понятие объектной модели	20
О двух системах типов	22
Родство C и Objective-C.....	23
Язык и библиотека	24
История развития Objective-C	26
Межплатформенная поддержка.....	30
Компилирование программ на Objective-C.....	32
Глава 2. Азбука Objective-C	35
Объявление типов Objective-C	36
Отправка сообщений	39
Представление о селекторах	43
Объявление классов	45
Применение протоколов	50
Добавление методов в класс.....	51
Применение неформальных протоколов	54

Синтезирование методов с помощью объявляемых свойств	55
Представление об аргументах self, _cmd и super.....	61
Представление об указателе isa.....	64
Инициализация классов.....	67
Чтение кодировок типов.....	69
Применение блоков.....	71

Глава 3. Управление памятью

Сохранение и освобождение объектов из памяти	76
Присваивание переменным экземпляра	77
Автоматический подсчет ссылок.....	79
Возврат объектов по аргументам-указателям	81
Исключение циклов сохранения.....	83
Переход к механизму ARC	85
Автоматически освобождаемые пулы.....	88
Применение автоматически освобождаемых конструкторов	90
Автоматическое освобождение объектов в методах доступа	91
Поддержка автоматической “сборки мусора”	92
Взаимодействие с кодом C.....	95
Представление о разрушении объектов.....	97
Применение слабых ссылок.....	99
Выделение просмотренной памяти	102

Глава 4. Шаблоны, распространенные в Objective-C

Поддержка шаблона создания объектов в два этапа	105
Копирование объектов	107

Архивирование объектов	109
Создание выделенных инициализаторов	112
Реализация шаблона-одиночки	114
Делегирование.....	117
Предоставление фасадов.....	118
Создание кластеров классов.....	120
Применение циклов исполнения.....	124
Глава 5. Числа.....	127
Сохранение чисел в коллекциях	129
Выполнение десятичных арифметических операций.....	132
Преобразование символьных строк в числа.....	135
Чтение чисел из символьных строк.....	137
Глава 6. Манипулирование символьными строками	141
Создание строковых констант	142
Сравнение символьных строк.....	143
Посимвольная обработка строк	146
Преобразование кодировок символьных строк.....	149
Обрезка символьных строк.....	151
Разделение символьных строк	153
Копирование символьных строк	154
Построение символьных строк по шаблонам	156
Сопоставление с шаблоном в символьных строках....	159
Сохранение форматированного текста	161
Глава 7. Работа с коллекциями	163
Применение массивов.....	164
Манипулирование индексами	166

Сохранение неупорядоченных групп объектов	168
Создание словаря	170
Циклическое обращение к коллекции	171
Поиск объекта в коллекции	175
Подклассификация коллекций	177
Сохранение объектов Objective-C в коллекциях C++	180
Глава 8. Даты и моменты времени	183
Обнаружение текущей даты	184
Преобразование дат для отображения	185
Расчет истекшего времени	188
Извлечение дат из символьных строк	189
Получение событий от таймера	190
Глава 9. Работа со списками свойств	193
Сохранение коллекций в списках свойств	194
Чтение данных из списков свойств	196
Преобразование форматов списков свойств	199
Применение формата JSON	200
Сохранение пользовательских настроек по умолчанию	202
Сохранение произвольных объектов в пользовательских настройках по умолчанию	206
Глава 10. Взаимодействие с окружением	209
Получение переменных окружения	209
Извлечение аргументов из командной строки	211
Доступ к региональным параметрам пользователя	213
Поддержка внезапного завершения процесса	214

Глава 11. Доступ к значениям по ключам.....	219
Доступ к значениям по ключу	220
Обеспечение совместимости с механизмом KVC	221
Представление о путях к ключам	224
Наблюдение за ключами.....	226
Обеспечение совместимости с механизмом KVO	228
Глава 12. Обработка ошибок	231
Отличия в обработке исключений во время выполнения.....	232
Генерирование и перехват исключений	235
Применение объектов исключений.....	237
Применение единой модели исключений	239
Управление памятью при обработке исключений ..	240
Передача делегатов ошибок.....	243
Возврат значений ошибок	244
Применение класса NSError	245
Глава 13. Доступ к файлам и каталогам	247
Чтение файла	248
Перемещение и копирование файлов	250
Получение атрибутов файлов.....	252
Манипулирование путями к файлам.....	253
Выяснение факта существования файла или каталога	254
Работа с пакетами.....	256
Поиск файлов в системе	259
Глава 14. Поток.....	263
Создание потоков	263
Управление приоритетностью потоков	265

Содержание	11
Синхронизация потоков	267
Сохранение данных, характерных для потоков.....	269
Ожидание по условию.....	272
Глава 15. Блоки и Grand Central Dispatch.....	277
Привязка переменных к блокам	278
Управление памятью при использовании блоков	282
Выполнение действий в фоновом режиме	285
Организация специальных очередей работ.....	287
Глава 16. Уведомления	289
Запрашивание уведомлений	290
Отправка уведомлений.....	291
Постановка уведомлений в очередь.....	292
Обмен уведомлениями между приложениями	294
Глава 17. Доступ к сети	299
Заключение сокетов C в оболочку.....	299
Установление связи с серверами	302
Общий доступ к объектам по сети.....	304
Поиск одноранговых узлов в сети.....	306
Загрузка данных по веб-адресу.....	309
Глава 18. Отладка программ на Objective-C	313
Инспектирование объектов	313
Выявление ошибок управления памятью	315
Наблюдение за исключениями	318
Утверждение исключений	320
Вывод отладочных сообщений.....	322

Глава 19. Динамическая библиотека Objective-C....	325
Отправка сообщений по имени	326
Поиск классов по имени	327
Проверка реагирования объекта на сообщение.....	329
Пересылка сообщений	330
Поиск классов	332
Инспектирование классов.....	334
Создание новых классов	336
Добавление переменных экземпляра.....	338
Предметный указатель	341

Об авторе

Дэвид Чиснолл является независимым писателем и консультантом. Повышая свое образование с целью получить ученую степень доктора философских наук, он принял участие в основании проекта *Étoile*, предназначавшегося для разработки открытой настольной среды поверх GNUStep — свободно доступной реализации прикладных программных интерфейсов OpenStep и Cocoa. Он является активным участником проекта GNUStep, первоначальным автором и сопровождающим динамической библиотеки GNUStep Objective-C 2, а также связанной с этим поддержки компилятора Clang.

Получив ученую степень доктора философских наук, Дэвид занимался некоторое время академической наукой, изучая историю языков программирования. Но в конечном итоге отошел от академической науки, осознав, что есть и другие прекрасные места для приложения его знаний, где не требуется много заниматься бумажной работой. Поэтому он периодически принимает участие в различных проектах по моделированию семантики динамических языков программирования.

Дэвид хорошо знаком со всеми особенностями Objective-C благодаря работе над проектами, в которых используется этот язык программирования, а также над реализацией самого языка. Ему пришлось работать над реализацией и других языков программирования, включая диалекты Smalltalk и JavaScript в качестве надстройки над динамической библиотекой Objective-C, что дает возможность совместно использовать код всех этих языков без промежуточного преобразования.

В свободное от написания книг и программ время Дэвид любит танцевать аргентинское танго и кубинскую сальсу, играть в бадминтон, метать летающие тарелки и вкусно готовить.

Благодарности

Что касается работы над этой книгой, то мне, как ее автору, хотелось бы прежде всего поблагодарить Николаса Роуда (Nicolas Road). Я приобрел свой первый Мак почти одновременно с началом работы над диссертацией на соискание ученой степени доктора философских наук и собирався воспользоваться им для программирования на Java, не желая изучать оригинальный язык данной платформы. В начале работы над диссертацией мне пришлось сотрудничать с Николасом Роудом, активным участником проекта GNUStep. Именно он убедил меня в том, что язык Objective-C и среда Сосоа пригодны не только для Маков, и поэтому заслуживают изучения. И он оказался совершенно прав: сопутствующие интегрированные среды в значительной степени упрощают разработку прикладных программ на Objective-C.

Далее мне хотелось бы выразить благодарность Фреду Кайферу (Fred Keifer). Фред сопровождает реализацию GNUStep в интегрированной среде разработки AppKit. Он выполнил невероятно скрупулезное (до педантизма) научное рецензирование этой книги, обнаружив в ней ряд неразъясненных мест. Если книгу будет приятно читать, то в этом огромная заслуга Фреда.

И наконец, хотелось бы поблагодарить всех, кто воплотил рукопись этой книги в настоящее издание, и в особенности Марка Тэйбера (Mark Taber), предложившего мне первым написать книгу по Objective-C.

Введение

Блез Паскаль однажды написал: “У меня не было времени на короткое письмо, и поэтому я написал длинное”. Этот карманный справочник — самая маленькая книга, которую мне довелось написать, и не скажу, что я с легкостью уместил в ее лаконичном формате все, что мне хотелось бы сказать на данную тему.

Когда Марк Тэйбер предложил мне написать эту книгу, я поначалу не был уверен в том, что это именно то, чем я хотел бы заняться. Ведь фразеологический словарь для родного языка — это перечень кратких идиом для тех, кто хотел бы одним-двумя меткими предложениями выразить важную мысль. Фразеологический словарь для языка программирования должен выполнять аналогичную роль.

Эту книгу не следует рассматривать в качестве исчерпывающего справочного пособия по языку Objective-C. Компания Apple подготовила такое пособие, доступное по адресу <http://developer.apple.com>. Ее не следует рассматривать и в качестве подробного учебного пособия по языку Objective-C, и, в отличие от другой моей книги на данную тему, *Cocoa Programming Developer's Handbook*, вы не найдете в ней примеров исходного кода законченных программ, но множество очень коротких примеров идиом, которыми, надо надеяться, вы сможете успешно пользоваться в самых разных ситуациях программирования на Objective-C.

Если вам доводилось в какой-нибудь книге по программированию встречать примеры кода, который на самом деле оказывался неработоспособным, то вы согласитесь

с тем, что в таких случаях чувствуешь себя попросту обманутым. В этой книге приведены листинги с двумя видами кода. Код на белом фоне предназначен для иллюстрации какой-нибудь простой идеи. Этот код может зависеть от некоторого предполагаемого контекста и не должен рассматриваться в качестве работоспособных и практически полезных примеров.

Большинство примеров кода в этой книге приведены в виде листингов на сером фоне. В нижней части каждого из таких примеров указано имя исходного файла, из которого взят данный листинг. Исходные файлы можно загрузить с веб-страницы этой книги на веб-сайте издательства InformIT по адресу <http://www.informit.com/title/0321743628>.

Во время работы над первым изданием этой книги я написал и протестировал все примеры на платформе Mac OS X. Отослав черновой вариант текста на редактирование, я протестировал их в среде GNUstep и был приятно удивлен тем, что практически все они работали. Ко времени выхода книги в свет эти примеры стали работоспособными полностью. Во втором издании книги рассматривается ряд средств, которые поддерживаются только на платформе Mac OS X 10.7 или iOS 5. Все приведенные здесь примеры также работают в среде GNUstep. Как и прежде, я подготовил все примеры в Mac OS X, не делая никаких уступок для GNUstep (если не считать уступкой тестирование примеров в данной среде).

О версиях книги

Настоящая книга написана в Vim (свободном режимном текстовом редакторе, созданном на основе более старого редактора vi) с использованием семантической разметки. В нем были сгенерированы три разные версии книги. Две из них создаются с помощью программы pdflatex. Если

вы читаете напечатанный текст книги или его электронный вариант в формате PDF, значит, имеете дело с одной из этих двух версий. Единственное различие между ними состоит в том, что печатная версия содержит метки обрезки страниц на печатной машине.

Третья версия представляет собой XHTML-документ, предназначенный для электронного издания в формате ePub. Она создается в среде EtoileText, которая сначала анализирует разметку в стиле LaTeX для построения древовидной структуры, а затем выполняет некоторые преобразования для обработки перекрестных ссылок и индексирования и, наконец, генерирует XHTML-документ. Весь необходимый для этого код написан на языке Objective-C.

Если у вас есть доступ как к текстовому, так и к электронному варианту книги, то вы можете заметить, что листинги примеров программ в формате издания ePub выглядят немного лучше. Дело в том, что в среде EtoileText для выделения синтаксиса используется пакет сервисных программ SourceCodeKit, еще одна среда типа Йтоиль, в которой для разметки листингов исходного кода служит часть Clang — современного компилятора Objective-C. Это означает, что места, занимаемые листингами в тексте книги, аннотированы точно такими же семантическими типами, которые воспринимает компилятор. Он, например, может отличать вызов функции от реализации макрокоманды.

Весь код, необходимый для выполнения описанного выше форматирования текста книги, вы найдете в хранилище подчиненных версий Etoile по адресу <http://svn.gna.org/viewcvs/etoile/trunk/Etoile/>.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать

лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: info@williamspublishing.com

Веб: <http://www.williamspublishing.com>

Адреса для писем из:

России: 127055, Москва, ул. Лесная, д. 45, стр.1

Украины: 03150, Киев, а/я 152

Основные принципы Objective-C

Для того чтобы овладеть языком Objective-C, необходимо усвоить основные принципы его построения. В отличие от языков программирования C++, D или Java, которые разработаны как новые C-подобные языки, язык Objective-C является гибридным языком. Он представляет собой чистое надмножество языка C, а это означает, что каждая действительная программа на C является также действительной программой на Objective-C. А кроме того, в нем допускается использовать некоторые Smalltalk-подобные синтаксические конструкции и семантику.

Один из разработчиков языка Objective-C, Том Лав (Tom Love), описал синтаксис квадратных скобок как “указательный столб”, напоминающий о том, что вы покинули область языка C и вступили на “землю объектов”. Первоначальная идея создания Objective-C зародилась в результате поиска подходящего способа для компоновки библиотек C, который способствовал бы слабой связности компонентов.

Одним из фундаментальных проектных решений в Objective-C послужило в отрицание какого бы то ни было “волшебства”. Все детали реализации должны быть открыты для программирующего. В отличие от C++, где подробности таблицы виртуальных функций скрыты от программирующего, Objective-C позволяет просматривать и модифицировать все, что касается объектов и классов.

В старых динамических библиотеках классы Objective-C были представлены структурами C, объявленными открытыми (`public`). Их можно было модифицировать непосредственно или даже создавать новые классы и регистрировать их средствами исполняющей системы. А в новых библиотеках эти структуры объявлены закрытыми (`private`), и для манипулирования ими как непрозрачными типами предусмотрен набор открытых функций.

Понятие объектной модели

В Objective-C используется Smalltalk-подобная объектная модель. Если у вас имеется некоторый опыт программирования на Java, то вам будет нетрудно усвоить Objective-C. Если же вы писали программы на таком языке семейства Simula, как C++, то перейти на Objective-C вам, возможно, будет труднее.

Алан Кей (Alan Kay) описывал идею объектов как простое упражнение в упрощении. Решая некоторую задачу, имеет смысл разделить ее на составные (более простые) части. Простейшее устройство, способное выполнять часть программы, может в то же время выполнять всю программу в целом, и таким устройством является компьютер. Объекты в представлении Алана Кея являются простыми моделями компьютеров, которые связываются друг с другом, обмениваясь сообщениями.

Точно так же ведут себя и объекты в Objective-C и его родителе — языке Smalltalk. Они представляют собой изолированные части программы, которые передают друг другу сообщения. Обычно эти сообщения доставляются синхронно и поэтому действуют подобно вызовам функций, но важно понимать, чем они отличаются.

Сообщения относятся к более высокому уровню абстракции, чем вызовы функций. Вызов функции организуется очень просто. Во многих архитектурах для этого доста-

точно единственной инструкции. Во многих других RISC-подобных архитектурах (с сокращенным набором команд) для вызова функции адрес возврата следует поместить в стек, а затем передать управление в начало кода функции. Но в любом случае адрес назначения фиксирован.

Когда сообщение отправляется, способ его обработки полностью определяется его получателем. Чаще всего он состоит в вызове метода, имя которого совпадает с именем сообщения, но существуют и другие, альтернативные способы. Промежуточные объекты способны переадресовать сообщение другому объекту, но перед этим они могут дополнить некоторую подстановку аргументов в сообщение.

В языке Simula не употреблялся термин “объектно-ориентированный подход”, но во многих Simula-подобных языках, появившихся после Smalltalk, этот термин был принят, что вносит некоторую путаницу. Для реализации внешне подобных функций в языках, относящихся к этому семейству, например в C++, используются *таблицы виртуальных функций (vtables)*. Объект в стиле Simula содержит указатель на свою таблицу виртуальных функций или аналогичные таблицы, если у него имеются суперклассы. При вызове виртуальной функции компилятор формирует сначала смещение в виртуальной таблице, а затем вызов функции с этим смещением.

Один из побочных эффектов этого отличия проявляется в том, что приведение типов указателей в Objective-C и C++ имеет совершенно разную семантику. Для преобразования указателя на объект одного типа в указатель на объект другого типа код в Objective-C вообще не генерируется. Приведение типов в этом случае рассматривается просто как указание для блока контроля типов в компиляторе. Но в C++ приведение типов указателей заключается в выполнении некоторых арифметических операций над указателями, чтобы передаваемый при этом код мог найти таблицу виртуальных функций в нужном месте. Это означает, что в Objective-C можно выпол-

нять преобразование любого объектного типа в любой другой объектный тип, тогда как в C++ для этого придется произвести специальный вид приведения типов.

Это отличие очень важно иметь в виду. В Objective-C на поиск метода оказывает влияние только тип объекта, а в C++ и других языках семейства Simula — также представление компилятора о том, с объектом какого именно типа ему приходится иметь дело.

О двух системах типов

Тех, кто решил программировать на Objective-C, может сначала сбить с толку наличие в этом языке двух систем типов. Такое положение имеет определенный смысл, если рассматривать первоначальную реализацию языка в качестве препроцессора. Препроцессор языка Objective-C должен сначала выполнить Strongtalk-подобный контроль типов, а затем компилятор C — контроль типов C.

В языке C применяется структурная система типов. Эквивалентность простых типов зависит от того, имеют ли два типа одно и то же представление. Сложные типы вообще не считаются эквивалентными. А в Objective-C добавлена алгебраическая система типов. Тип объекта определяется его сигнатурой, т.е. сообщениями, на которые он должен реагировать. Любой объект Objective-C можно неявным образом привести к его суперклассу, поскольку он гарантированноотреагирует на все сообщения, которые распознаются этим суперклассом.

¹ StrongTalk — это диалект языка Smalltalk, в который добавлен необязательный статический контроль типов. Разработчики этого языка продолжили в дальнейшем работу над Java. Язык StrongTalk считался самой быстросействующей реализацией Smalltalk, но для оптимизации в его компиляторе не использовались сведения о статических типах.

Преобразования типов объектов можно выполнять и явным образом. В частности, объект словаря можно было бы преобразовать в объект массива. И такое преобразование типов окажется вполне работоспособным, поскольку данному объекту будут в дальнейшем посылаться лишь те сообщения, которые распознаются как словарями, так и массивами.

Родство C и Objective-C

Одна из важнейших особенностей Objective-C состоит в том, что он представляет собой чистое надмножество языка C. В частности, C можно рассматривать как зависящий от конкретной предметной области язык, встроенный в Objective-C, для решения низкоуровневых задач, а подмножество языка Smalltalk — как зависящий от конкретной предметной области язык, встроенный в Objective-C, для решения задач высокого уровня.

Нет ничего ошибочного в том, что конкретную задачу можно попытаться решить только средствами языка C. Но одна из типичных ошибок, совершаемых многими при изучении Objective-C, заключается в следующем: они считают, что должны перестать пользоваться языком C. А ведь во многих случаях нужное решение задачи находится именно на языке C.

Это можно наблюдать в средах, интегрированных в Сосоа, где далеко не все считается объектом. Так, если требуется задать диапазон символов (подстроку) в строке, для этой цели можно воспользоваться структурой `NSRange`. Но ведь это структура C, содержащая местоположение и длину подстроки, а не объект Objective-C. Аналогичным образом задаются точки, прямоугольники и прочие элементы. Если бы они были объектами, для манипулирования ими потребовалось бы больше оперативной памяти, что существенно замедлило бы выполнение операций над ними, не говоря уже о гибкости такого подхода.

На заметку

У Objective-C имеется также родственный язык Objective-C++, который отличается такими же отношениями с языком C++, как у Objective-C с языком C. Язык Objective-C++ представляет собой чистое надмножество C++ и позволяет вызывать код на C++ из объектов Objective-C, и наоборот. В этих языках используются совершенно разные объектные модели, поэтому создать подкласс из класса Objective-C средствами C++ не удастся, но можно воспользоваться указателями на объекты Objective-C в качестве полей в объектах C++, а также указателями на объекты C++ в качестве переменных экземпляра в объектах Objective-C.

Тест Инголса² на объектно-ориентированный подход предполагает возможность создавать новый целочисленный тип и использовать его для расположения окна на экране. Язык Objective-C не проходит этот тест, но он представляет собой далекий от практики пример, поскольку большая часть времени, затрачиваемого на разработку новых целочисленных типов данных, уходит на нарушение многих правил, предполагающих, что целочисленные типы будут вести себя привычным, вполне ожидаемым образом.

При программировании на Objective-C совсем не обязательно использовать такие понятия динамического поведения, как позднее связывание и полиморфизм, если это не требуется для решения конкретной задачи.

Язык и библиотека

В динамических языках очень трудно провести границу между самим языком и библиотекой. Так, в языке Smalltalk отсутствует эквивалент оператора `if`. Стандартная библиотека определяет одиночные классы `True` и `False`, реагиру-

² Назван в честь Дэна Инголса (Dan Ingalls), одного из разработчиков объектно-ориентированного языка программирования Smalltalk. Дэн Инголс и предложил использовать этот тест.

ющие на сообщения `-ifTrue:`, принимающие замыкание в качестве аргумента. Подобным образом реализованы все сложные управляющие конструкции, в том числе циклы и перечисления.

Objective-C наследует управление выполнением программы от языка C, но все его динамическое поведение обеспечивается динамической библиотекой. Именно этим Objective-C отличается от языка C или C++. Так, можно скомпилировать автономный двоичный код C или C++, в котором ни одна из функций библиотеки `libc` или `libstdc++` не вызывается. Но всякая работоспособная программа на Objective-C, не написанная только на C, должна связываться с библиотекой `libobjc`.

Эта библиотека традиционно предоставляет три класса: простой корневой класс, класс для строковых констант и класс для протоколов. Первый из них обеспечивает необходимые функциональные возможности, включая размещение объектов в оперативной памяти, что является частью функций C++, Java и подобных им языков. А два других класса могут быть сгенерированы компилятором.

Пользоваться этими классам вам вряд ли придется. Практически во всем коде Objective-C, написанном за два последних десятилетия, используется реализация интегрированной среды Foundation в OpenStep — спецификации прикладного интерфейса, совместно разработанной компаниями NeXT и Sun и предназначенной для обеспечения современной объектно-ориентированной среды разработки межплатформенных приложений. Компания NeXT реализовала ее для собственной операционной системы OPENSTEP и Windows NT, тогда как компания Sun — для системы Solaris. Некоторое время спустя в рамках проекта GNU была осуществлена третья реализация под названием GNUstep.

В прикладном интерфейсе OpenStep были определены две интегрированные среды: Foundation Kit и Application Kit, обычно именуемые для краткости Foundation и AppKit.

Среда Foundation Kit обеспечивает основные функциональные возможности, необходимые разработчикам для создания всех видов приложений, включая классы коллекций, циклы исполнения, доставку уведомлений, абстракцию ОС и т.п. А среда Application Kit служит в качестве настройки над Foundation Kit, обеспечивая дополнительную поддержку в разработке графических приложений.

Реализация компании Sun не слишком продвинулась в своем развитии после первого выпуска. Несколькими годами позже компания Apple приобрела компанию NeXT и переименовала прикладной интерфейс OpenStep в среду Yellow Box, предложив ее как среду разработки в операционных системах Rhapsody и Windows. После выпуска операционной системы Mac OS X эта среда снова была переименована — на этот раз в Cocoa.

Проект GNUstep до сих пор активно развивается, но в настоящее время, помимо базовой спецификации, усовершенствования, внесенные компанией Apple, постепенно внедряются в спецификацию OpenStep, а также в базовую спецификацию. Это означает, что среда Foundation находит применение практически везде, хотя в реализации GNUstep опущены некоторые новые классы и методы.

Среду Foundation можно рассматривать как стандартную библиотеку Objective-C. Она содержит немало инструментальных средств, требующихся для создания нетривиальных программ. Некоторые языковые средства Objective-C 2 специально предназначены для работы со средой Foundation. Ими можно пользоваться и без Foundation, но только в том случае, если реализуется нечто равноценное.

История развития Objective-C

Термин “объектно-ориентированный” был придуман Аланом Кеем в 1960-х годах, а язык Smalltalk был создан им (вместе с коллегами) для демонстрации этого стиля програм-

мирования. Язык Smalltalk был разработан в 1970-е годы, но самая распространенная его версия появилась лишь в 1980-м году.

Тем не менее язык Smalltalk оказался недостаточно быстродействующим. Для нормальной работы программ на Smalltalk требовался такой мощный компьютер, как Xerox Alto с оперативной памятью объемом 512 Кбайт и (как минимум) процессором с тактовой частотой 2 МГц.

Брэду Коксу (Brad Cox) понравились принципы организации Smalltalk, но ему все-таки хотелось иметь язык, пригодный для применения на компьютере многими пользователями. Поэтому он решил объединить высокоуровневый язык Smalltalk, поддерживающий инкапсуляцию, слабую связность и повторное использование существующего кода, с языком C, организованным по совершенно другим принципам и отличающимся небольшим уровнем инкапсуляции и отсутствием динамической диспетчеризации, но обладавшим одним существенным преимуществом: высоким быстродействием.

Язык C опирался в значительной степени на систему команд мини-ЭВМ PDP-11. Очень простой компилятор C мог генерировать довольно быстро исполнявшийся код. И это положение остается справедливым до сих пор. В таких компиляторах, как LLVM, XLC и ICC, много внимания уделяется оптимизации кода, чего нельзя сказать о компиляторе PCC, но, несмотря на это, он генерирует достаточно быстрый код.

В 1986 году Брэд Кокс и Том Лав основали компанию StepStone и выпустили на рынок программный продукт, основанный на ранее созданном Брэдом проекте Object Oriented Precompiler. Этот продукт назывался Objective-C и представлял собой препроцессор и небольшую *динамическую библиотеку Objective-C*. Препроцессор генерировал код C, который можно было скомпилировать с помощью компилятора C, реализованного на конкретной платформе. В 2010 году мне довелось обнаружить компанию, где

компилятор StepStone до сих пор используется в коммерческом программном продукте.

В 1988 году компания NeXT приобрела лицензию на исходный код StepStone и выкупила торговую марку Objective-C. Затем разработчики из компании NeXT переписали препроцессор как внешний интерфейс для *набора компиляторов в рамках проекта GNU* (GCC — GNU Compiler Collection). После некоторых юридических споров Фонд свободного программного обеспечения (Free Software Foundation) заставил компанию NeXT открыть этот код, но динамическая библиотека так и осталась закрытой.

Тогда в рамках проекта GNU была написана замена для динамической библиотеки, но с некоторыми изменениями. Одно из них заключалось в том, что в версии GNU этой библиотеки селекторы имели связанный с ними тип, тогда как на платформах компаний NeXT и Apple селекторы представлены символьными строками. Это означает, что в версии GNU динамическая библиотека способна перехватывать во время выполнения разрушение стека, вызванное программными ошибками, которые не обнаруживаются аналогичной библиотекой в версии от компании NeXT.

Еще одно изменение было связано с тем, как работает отправка сообщений. При отправке сообщения средствами динамической библиотеки в версии компании NeXT компилятор преобразует отправляемое сообщение в вызов функции `objc_msgSend()`, которая находит нужный метод и вызывает его. К сожалению, эту функцию невозможно реализовать на C, поскольку пришлось бы написать отдельную сборку для каждой платформы и для каждого соглашения о вызовах, действующего на этой платформе. Поэтому в версии GNU упомянутую выше функцию было решено заменить функцией `objc_msg_lookup()`, возвращающей указатель на функцию. И хотя это немного замедляло исполнение кода, но в то же время означало, что один и тот же код мог выполняться на всех платформах.

С тех пор язык Objective-C не претерпел существенных изменений. В 1997 году компанию NeXT приобрела компания Apple, а Objective-C был принят в качестве основного языка для среды разработки Yellow Box в новой тогда операционной системе Rhapsody этой компании. В дальнейшем среда разработки Yellow Box была переименована в Socoa, а операционная система Rhapsody — в Mac OS X. В 2003 году компания Apple добавила в этот язык некоторые Java-подобные примитивы для обработки исключений.

Следующий ряд усовершенствований увидел свет в 2007 году, когда компания Apple представила язык Objective-C 2. Новая версия языка ввела в некоторое заблуждение тех, у кого имелся опыт программирования на Objective-C, поскольку предыдущая его версия, выпущенная компанией NeXT, называлась Objective-C 4. К счастью, подобная непоследовательность в нумерации версий языка никак не отразилась на коде сортировки в среде Socoa.

Версия Objective-C 2 обогатилась несколькими синтаксическими изюминками. В ней были предложены новые способы реализации перечислений, синтеза методов доступа и *“сборки мусора”*. Objective-C 2 — это скорее модное словечко для маркетинга, чем настоящий язык программирования, поскольку вы не найдете такого параметра компиляции, который позволил бы выбрать его. Одни возможности этого языка (например, неуязвимые переменные экземпляра) доступны только в некоторых динамических библиотеках, а другие (например, *“сборка мусора”*) — недоступны в мобильных телефонах iPhone. Объявляемые свойства и быстрое перечисление — единственные особенности Objective-C 2, которые доступны везде, включая и платформы других производителей, помимо компании Apple.

Начиная с версий операционных систем iOS 5 и Mac OS X 10.7, компания Apple перестала указывать номера версий в обозначении новых программных продуктов и начала применять вместо них номера версий компиляторов.

Еще до выхода в свет ряд новых средств языка был известен под названием Objective-C 2.1, но в отделе маркетинга компании Apple решили не пользоваться этим названием публично. К числу этих новых средств относилась улучшенная поддержка сокрытия данных, упрощающая вынос подробностей реализации из открытых заголовков, а также автоматический подсчет ссылок, сочетающий в себе почти все преимущества “сборки мусора” и подсчета ссылок вручную, но исключаящий большинство недостатков каждого из этих механизмов в отдельности.

Межплатформенная поддержка

Принуждение компании NeXT открыть внешний интерфейс Objective-C для проекта GCC в качестве свободно распространяемого программного продукта послужило чем-то вроде удачного тактического хода для Фонда свободного программного обеспечения. К сожалению, это было не очень хорошо с точки зрения межплатформенной поддержки в долгосрочной перспективе. Код, открытый по требованию компанией NeXT, оказался ужасным и в течение десяти лет практически игнорировался участниками проекта GCC. Положение немного улучшилось с выходом версии GCC 4.6, но появление современной версии Objective-C все-таки задерживается на пять лет.

В GCC имеется довольно приличная поддержка Objective-C после эпохи компании NeXT, а также новейших ключевых слов, связанных с обработкой исключений. Проект GNUstep предоставляет реализации для сред Foundation и AppKit, которые составляют основу разработок для платформ NeXT и Mac OS.

В 2007 году компания Apple начала работу над новым компилятором *Clang*, который представляет собой внешний интерфейс для *низкоуровневой виртуальной машины* (LLVM — Low Level Virtual Machine), а также для компиля-

ции программ, написанных на языках семейства C, а именно: C, Objective-C и C++. Компилятор Clang гораздо более модульный, чем GCC, и предназначен для автоматического завершения кода, выделения синтаксиса, статического анализа и просто компиляции. Этот компилятор четко разделяет код, используемый в различных слоях, и заключает в отдельные классы код, характерный для времени выполнения.

Этот новый компилятор реализует Objective-C лишь наполовину. Другую половину составляет динамическая библиотека, которая была разработана с использованием GCC, но без реализации функций, необходимых для Objective-C 2. С целью исправить это положение я написал интегрированную среду как часть проекта Étoile, в которой реализованы многие недостающие функции и открытые прикладные интерфейсы API исполняющей среды, внедренные компанией Apple в операционной системе Mac OS X 10.5.

В настоящее время эта среда является частью проекта GNUstep, предназначенной для использования со старыми версиями динамической библиотеки GCC. Я также написал новую динамическую библиотеку, которая поддерживает в GCC старый двоичный прикладной интерфейс ABI и новый неуязвимый двоичный интерфейс ABI, что сравнимо в настоящее время по своим возможностям с Mac OS X. Эта разработка является частью проекта GNUstep и называется *динамической библиотекой GNUstep*, или *libobjc2*.

Вместе с внешним компилятором Clang и динамической библиотекой, предоставляемыми в рамках проекта GNUstep, вы получаете в свое распоряжение полную поддержку всех языковых средств Objective-C 2, включая неуязвимые переменные экземпляров. Это означает, что вы можете пользоваться языком Objective-C 2 в Windows, Linux, *BSD, Solaris и т.д. По меньшей мере один человек тестировал его на платформе QNX, и он может оказаться вполне работоспособным на платформе Symbian.

Если вы пользуетесь версией GCC динамической библиотеки вместе со средой Objective-C 2 в проекте GNUstep, то получаете доступ к таким языковым средствам, как быстрое перечисление и объявляемые свойства, но при этом не извлекаете никакой выгоды от использования нового двоичного прикладного интерфейса ABI. Кроме того, GCC 4.6 и более поздние версии поставляются с обновленной динамической библиотекой, которая обеспечивает аналогичный уровень совместимости, но без обязательной совместимости со средой Objective-C 2.

Компилирование программ на Objective-C

```
0 $ gcc scanner.m
1 Undefined symbols:
2   "_OBJC_CLASS_$_NSAutoreleasePool", referenced
   from:
3   __objc_classrefs__DATA@0 in ccorpaJq.o
4   "_objc_msgSend", referenced from:
5   _main in ccorpaJq.o
6   ...
7 $ # On OS X
8 $ gcc -framework Foundation scanner.m
9 $ # On other platforms
10 $ gcc 'gnustep-config --objc-flags --base-libs'
    scanner.m
```

Компилятор Clang предназначен для использования вместо компилятора GCC, поэтому везде, где в инструкциях встречается команда gcc, можете подставить вместо нее команду clang. В Mac OS X компилятор GCC устанавливается в папке /usr/bin/gcc, поэтому он включается в путь к вашим ресурсам. В более старых версиях инструментальных средств Xcode компилятор Clang устанавливался в папке /Developer/usr/bin/clang, поэтому вам, возможно, придется указать полный путь или добавить эту папку

в путь к вашим ресурсам. На других платформах вы, как правило, найдете обе папки в пути к вашим ресурсам, если эти компиляторы были установлены.

Если у вас имеется некоторый опыт компиляции программ на C или C++ с помощью любого из этих компиляторов, то компилирование программ на Objective-C покажется вам очень знакомым делом. Так, если просто указать исходный файл в командной строке, компилятор попытается скомпилировать и скомпоновать его в исполняемый файл.

На заметку

На конференции разработчиков в 2011 году представители компании Apple заявили: "Мы прощаемся с GCC". Компания Apple больше не занимается активной разработкой компилятора GCC, поэтому он будет испытывать недостаток в поддержке недавно добавленных языковых средств и, скорее всего, не будет поставляться в будущей версии инструментальных средств разработки. В отделе маркетинга компании Apple немного путаются в компиляторах. ОС Mac OS X 10.7 поставляется вместе с компилятором, который обозначен как LLVM Compiler 3.0. Но в действительности это "слепок" компилятора Clang, сделанный на полпути между выпусками 2.9 и 3.0. Если вы выполните команду `clang -version`, то увидите на экране нечто вроде "based on LLVM 3.0svn", т.е. основывается на версии LLVM 3.0svn.

Зачастую такая компиляция не дает положительного результата, как следует из иллюстрации приведенной в начале этого раздела. Для программ, написанных на Objective-C, требуется ряд других библиотек. В драйверах компилятора на платформе Mac OS X поддерживается дополнительный параметр командной строки `-framework`, который задает пакет интегрированной среды, содержащий заголовки и совместно используемую библиотеку. Этот параметр служит одновременно для указания компоновщика и прелоадера.

Во всех примерах, приведенных в этой книге, используется среда Foundation. На других платформах она обычно реализована в библиотеке GNUstep Base. На этих платформах параметр командной строки `-framework`, как правило, не поддерживается. Альтернативный вариант предоставляется инструментальным средством `gnustep-config`, которое позволяет распечатать параметры командной строки, требующиеся для компиляции программ на Objective-C и компоновки с базовой (Foundation) или графической (AppKit) библиотекой.

Как и при вызове компилятора C и C++, можете задать параметр командной строки `-c`, чтобы просто скомпилировать файл, но без компоновки, а также параметр `-o`, чтобы задать имя выходного файла. Если компилируете несколько файлов, то, вероятно, имеет смысл пользоваться более современными средствами по сравнению с обыкновенным запуском компилятора из командной строки.

На платформе Mac OS X для этой цели можно пользоваться интегрированной средой разработки XCode IDE. Она позволяет создавать проекты с использованием различных типов шаблонов. Если вам понадобится скомпилировать их на других платформах, отыщите в хранилище GNUstep инструментальное средство под названием `pbxbuild`.

Если вы работаете на другой платформе, то можете пользоваться пакетом GNUstep Make для построения своих проектов. Он также работает на платформе Mac OS X, поэтому при выборе любого варианта у вас не должно возникнуть затруднений с переносимостью ваших программ.

Азбука Objective-C

Objective-C содержит совсем небольшой набор дополнений к языку C. Первая версия Smalltalk была создана на спор, что можно определить целостный язык общего назначения на одном листке бумаги. Расширения Objective-C до языка C несколько сложнее, чем у Smalltalk, но неамного.

Если вы уже знаете язык C, то изучение Objective-C займет у вас полдня — или даже меньше, при условии, что вы хорошо усвоили принципы объектно-ориентированного программирования в стиле Smalltalk. Это, конечно, небольшое преувеличение. Если бы освоение Objective-C действительно было таким простым, то читатель почувствовал бы себя слегка ограбленным, купив эту книгу. Подобно Smalltalk, основная причина “простоты” языка Objective-C заключается в том, что он делегирует многие свои “сложности” библиотеке.

Стандартная библиотека для оригинальной версии StepStone языка Objective-C была очень маленькой. Она предназначалась для применения вместе с библиотеками C, а не как самостоятельная языковая единица. В настоящее время в большей части кода, написанного на Objective-C, в качестве стандартной библиотеки используются программные продукты, опирающиеся на среду Foundation, разработанную в компании NeXT, а ныне доступную как среда Foundation от компании Apple или библиотека GNUstep Base. Любой из этих вариантов библиотеки поддерживает такие функциональные возможности, как управление памятью с использованием ме-

ханизма подсчета числа ссылок, стандартные структуры данных, механизмы пересылки и т.д.

Многое из того, что на первый взгляд кажется частью Objective-C, в действительности обеспечивается средой Foundation. Языком Objective-C можно, конечно, пользоваться и без среды Foundation, но это сродни желанию доказать свою способность добиться результата любой ценой, а не изящным способом.

Объявление типов Objective-C

```
6 NSMutableArray *mutableArray = [NSMutableArray  
    new];  
7 NSArray *array = mutableArray;  
8 NSObject *object = array;  
9 id obj = mutableArray;  
10 mutableArray = (NSMutableArray*)object;  
11 mutableArray = obj;
```

Пример кода из файла cast.m

Язык Objective-C, как следует из его названия, дополняет язык C объектами. Но это не простые объекты, а такие, которые соответствуют модели Smalltalk, т.е. являются экземплярами классов. Оперативная память для объектов всегда выделяется в “куче”, и поэтому доступ к ним всегда осуществляется по указателю.

В первых реализациях Objective-C, которые генерировали только код C из кода Objective-C, классы превращались в определения структур, описывавшие расположение объектов в памяти. Свидетельства этого можно до сих пор наблюдать в сообщениях компилятора GCC об ошибках, где объекты иногда называются “структурами”.

В языке C приходится непременно выполнять явное приведение типов, чтобы преобразовать указатель на структуру одного типа в указатель на структуру другого типа. А в Objective-C это правило немного ослаблено.

Расположение любого объекта определяется сначала *переменными экземпляра* в корневом классе, а затем переменными экземпляра в каждом последующем классе вниз по иерархии. Подклассы могут только добавлять новые переменные экземпляра, но не удалять их из суперклассов и не делать их перестановки. Это означает, что указатель на экземпляр одного класса можно безопасно (с точки зрения распределения памяти) привести к указателю на экземпляр суперкласса. Это безопасно и с точки зрения объектно-ориентированной системы типов, поскольку любой подкласс будет всегда реагировать на все сообщения, на которые должен реагировать суперкласс.

Указатели на объекты в Objective-C определяются именем класса, за которым следует “звездочка” (практически так же, как в C определяются указатели на структуры, но без ключевого слова `struct`). Указатель на экземпляр одного класса можно всегда привести неявным образом к указателю на экземпляр его суперкласса. Это правило приведения типов распространяется даже на косвенные суперклассы. Например, `NSMutableArray` — это подкласс, производный от класса `NSArray`, который, в свою очередь, является подклассом, производным от класса `NSObject`. Поэтому тип `NSMutableArray` можно привести к типу `NSArray` или `NSObject`. Но для приведения типов в обратном направлении потребуется явно указать приведение типов.

Кроме того, в Objective-C введен новый тип указателя: `id`. Это приблизительный аналог типа `void*`, что касается правил приведения типов. Любой указатель на объект можно неявно привести к типу `id`, а тип `id` — к любому типу указателя на объект.

Если вы привыкли программировать на таких языках без контроля типов, как Smalltalk, Ruby или Python, то у вас может возникнуть искушение пользоваться исключительно типом `id`. Как правило, это безопасно, но из этого правила имеется одно исключение. Для поиска в Objective-C исполь-

зуется только имя сообщения без дополнительных сведений о типе. Это означает, что вы можете объявить два отдельных метода в разных классах с одним и тем же именем, но разного типа. Когда компилятор генерирует отправку сообщения, для определения типов аргументов он использует тип получателя сообщения. Это, в свою очередь, определяет расположение фрейма вызова. Если существует только один метод, объявленный повсюду под этим именем, сведения о типе не требуются. А если требуется получить доступ к переменным экземпляра за пределами объекта, то без статического контроля типов не обойтись, хотя такой стиль программирования обычно не приветствуется.

Несмотря на то что указывать сведения о статических типах необязательно, зачастую это делать весьма полезно. Если не принимать во внимание упомянутый выше случай, эти сведения никак не повлияют на генерирование кода, но будут использованы компилятором на этапе семантического анализа, что позволит выявить некоторые ошибки во время компиляции. Так, если вы попытаетесь добавить объект в экземпляр класса `NSArray`, компилятор предупредит вас о том, что получатель не реагирует на соответствующее сообщение. Благодаря этому можете проверить, действительно ли вы имели в виду класс `NSMutableArray`, или вам нужно было создать изменяемую копию массива.

На заметку

В C++ для описания неизменяемых объектов принято использовать ключевое слово `const`. А в Objective-C ключевое слово `const` совершенно бесполезно. Оно указывает на то, что переменные экземпляра некоторого объекта нельзя модифицировать непосредственно, но это делается крайне редко. Данное ключевое слово не изменяет сообщения, которые могут быть отправлены объекту, поэтому объект, объявленный как `const`, остается потенциально изменяемым.

Если требуется подчеркнуть отличие изменяемых экземпляров от неизменяемых, программирующие на Objective-C, как правило,

пользуются *шаблоном изменяемого подкласса*, в котором неизменяемый класс содержит изменяемый подкласс. Изменяемую версию можно неявным образом привести к неизменяемой, но неизменяемый экземпляр нельзя сделать изменяемым.

Тип **Class** подобен типу `id`, но он может ссылаться на классы. В Objective-C класс представляет собой объект, поэтому вместо типа `Class` можно всегда использовать тип `id` таким же образом, как и вместо любого конкретного типа объекта. В стандартных заголовках Objective-C определены еще два типа¹. Тип **SEL** используется для селекторов, которые будут рассмотрены далее в этой главе.

Еще один тип **IMP** обозначает Instance Method Pointer, т.е. указатель на метод экземпляра. Этот тип предусмотрен для указателя на метод Objective-C. Потребность в использовании этого типа обычно не возникает. Он применяется только для некоторых динамических средств Objective-C и для оптимизации. Некоторые примеры уместного применения этого типа приведены в главе 19.

Отправка сообщений

```
9 [anObject autorelease];
10 [anObject addObject: anotherObject];
11 [anObject setObject: anotherObject
12     forKey: aThirdObject];
```

Пример кода из файла `exampleMessages.m`

Самым значительным дополнением, внесенным Objective-C в язык C, является отправка сообщений. Как упоминалось в предыдущей главе, *сообщение* является высокоуровневой управляющей конструкцией.

¹ Если вы пользуетесь новым компилятором Clang, то большинство этих типов встроено в определения, и поэтому заголовки необязательны.

Именно эта особенность Objective-C способна привести в замешательство программистов, переходящих на Objective-C из таких языков, как C++ и Java, поскольку синтаксис отправки сообщений взят непосредственно из Smalltalk и кажется чужеродным тем, кто больше знаком с синтаксисом языка C. Эта чужеродность намеренная. В языке C не предусмотрено поведение, аналогичное отправке сообщений в Objective-C, поэтому наличие нового синтаксиса явно подчеркивает факт наличия в этом языке новой семантики. Напомним, что Том Лав охарактеризовал Objective-C как гибридный язык с четким синтаксическим разделением на части, относящиеся только к языку C, и “землю объектов”, которая имеет Smalltalk-подобный синтаксис, сопутствующий Smalltalk-подобной семантике.

В простейшем виде отправка сообщений осуществляется без принимаемых аргументов. Примером тому может служить отправка сообщения `-count` массиву. Обратите внимание на знак `-` перед именем сообщения. Это делается по соглашению, принятому в документации на Objective-C, для обозначения сообщения, отправляемого экземпляру. А знаком `+` обозначается сообщение, отправляемое классу, например `+new` или `+arrayWithObject:`. Рассмотрим следующую строку кода:

```
size = [anArray count];
```

В этой строке кода определяется количество элементов массива и полученное значение сохраняется в переменной `size`. В таких языках, как C++ и Java, эквивалентный код выглядел бы следующим образом:

```
size = anArray.size();
```

Аналогичный синтаксис применялся бы и в Objective-C, если бы массив `anArray` был определен как структура C, а поле `size` — как указатель на функцию. Но поскольку это не имеет ничего общего с отправкой сообщения объек-

ту, разработчики Objective-C решили не применять подобный синтаксис во избежание недоразумений.

Сообщения, принимающие один аргумент, внешне очень похожи на сообщения без аргументов. Так, если в массив Objective-C требуется добавить объект, то для этой цели можно воспользоваться синтаксисом, аналогичным следующему:

```
[anArray addObject: anObject];
```

Несколько иначе обстоит дело, если сообщения принимают несколько аргументов. Как и в Smalltalk, у каждого аргумента имеется свое имя, поэтому для ввода объекта в массив по заданному индексу можно воспользоваться следующим синтаксисом:

```
[anArray insertObject: anObject  
                  atIndex: anIndex];
```

Благодаря этому код Objective-C очень легко читается, если, конечно, понятен его основной синтаксис. Ведь каждое сообщение раскрывает назначение каждого своего аргумента. В документации аналогичное сообщение было бы обозначено следующим образом: `-insertObject:atIndex:.` Два знака двоеточия в такой записи указывают на то, что сообщение принимает два аргумента, а текст перед каждым двоеточием определяет, каким должен быть соответствующий аргумент. Как упоминалось выше, знак `-` означает, что данное сообщение отправляется экземпляру.

Если сообщение отправляется объекту, для реализации которого отсутствует соответствующий метод, в таком случае используется ряд резервных методов. В стандартной реализации последний из резервных методов сгенерирует исключение, которое можно перехватить во время выполнения и соответственно обработать. Перед тем как отправлять сообщение объекту, можно также опросить его и выяснить, известно ли ему, как обработать такое сообщение. О том, как это делается, речь пойдет в главе 19.

На заметку

Если отправить сообщение константе `nil`, определенной как пустой указатель (`id`) 0, то в ответ будет получено нулевое значение (0) вместо ошибки. И это очень полезная особенность языка, поскольку она дает возможно послать последовательность сообщений результату предыдущих отправок сообщений, и если одно из возвращаемых значений (где-то в середине последовательности) окажется равным `nil`, то остальная часть сообщений все равно будет действовать. Это избавляет от необходимости выполнять большое количество проверок на равенство пустому значению `NULL` в коде Objective-C.

Поведение, когда в результате отправки сообщения константе `nil` возвращается структура, не определено. Если реализовать такое поведение в коде, скомпилированном с помощью GCC на платформе SPARC, то программа аварийно завершится сообщением об использовании запрещенной команды. В Mac OS X структура в подобной ситуации будет заполнена случайными значениями, аналогично неинициализированной структуре в стеке. Поэтому перед отправкой объекту сообщения, возвращающего структуру, следует непременно убедиться, что этот объект не является пустым, т.е. константой `nil`.

В Objective-C (как, впрочем, и в Smalltalk) классы также являются объектами. Сообщения можно отправлять классам, как и любому объекту, используя имя класса в качестве получателя сообщения. Наиболее распространенной причиной для отправки сообщения классу служит намерение создать экземпляр этого класса. В Objective-C классы сами несут ответственность за создание собственных экземпляров. В этом языке отсутствует эквивалент ключевого слова `new`, определенного в C++ или Java. Ближайшим ему эквивалентом является сообщение `+new`, которое можно послать классу. Например, новый изменяемый массив можно было бы создать следующим образом:

```
id anArray = [NSMutableArray new];
```

Несмотря на то что эта запись выглядит вполне привычно, язык не дает никаких гарантий в отношении действий, кото-

рые выполнит данное сообщение. Одни классы могут создавать экземпляры подкласса, а другие — возвращать одиночный экземпляр. И только специальное соглашение позволяет создавать новый экземпляр получателя сообщения.

В последней версии Objective-C внедрено понятие *селектов методов*. Они представляют собой молчаливые договоры, которые должны соблюдать некоторые методы. Например, предполагается, что метод `+new` должен возвращать *владеющую ссылку* на экземпляр получателя. В приведенной выше строке кода компилятор будет ожидать, что этот метод возвратит изменяемый массив. Если же попытаться присвоить его переменной типа `NSMutableDictionary`, компилятор выдаст предупреждение.

Представление о селекторах

```
5 SEL new = @selector(new);  
6 SEL set =  
7 @selector(setObject:forKey:);
```

Пример кода из файла `selector.m`

В языке C обращение к функциям происходит по имени. Имя функции может быть также использовано как указатель на эту функцию. При компилировании кода C функция становится меткой, доступ к которой можно получить по одной команде вызова или их последовательности.

Управление выполнением программы в Objective-C происходит более динамично. Для этого можно выбрать указатель на метод, но, как правило, лучше отправить объекту конкретное сообщение, чем вызывать отдельный метод. Если же вы вызываете конкретный метод, то должны быть совершенно уверены в том, что получатель действительно является предполагаемым классом.

В языке C указатели на функции часто используются в качестве аргументов других функций. В итоге вызываемая функция вызовет ту функцию, указатель на которую был

ей передан. Характерным тому примером может служить функция `qsort()` из стандартной библиотеки C. Она сортирует массив, используя в качестве аргумента функцию для определения порядка следования сортируемых элементов.

Для организации аналогичной сортировки в Objective-C объекты придется сравнивать, отправляя им сообщение `-compare:`, а не пользуясь указателем на функцию. В идеальном случае достаточно передать имя отправляемого сообщения методу, реализующему сортировку.

Имя сообщения называется *селектором*. Это абстрактное представление имени сообщения, которое имеет тип `SEL`, являющийся непрозрачным типом. Создать селектор можно с помощью директивы `@selector()`. Она принимает представление имени сообщения в виде строковой константы и преобразует его в селектор для имени этого сообщения.

Селекторы можно использовать для вызова метода по имени, как поясняется в главе 19. Целый ряд методов из среды Foundation селекторы принимают в качестве своих аргументов. Они используются аналогично функциям C, которые принимают в качестве своих аргументов указатели на функции для отложенного выполнения, обратных вызовов, сортировки коллекций и т.п.

На заметку

Существенное отличие реализации языка Objective-C в версиях компании Apple и проекта GNU состоит, в частности, в том, что в версии GNU используются *типизированные селекторы*. Селекторы — это имена, используемые для поиска методов при отправке сообщения. В версии GNU языка Objective-C они также допускают кодировку типов аргументов в сообщении. Это дает функциям возможность удостовериться в том, что они были вызваны с правильными типами аргументов, что весьма полезно для манипулирования распределенными объектами и в ряде других случаев.

В настоящее время *типовозависимая диспетчеризация* используется в динамической библиотеке GNUstep по умолчанию. Это означает,

что поиск метода зависит от кодировки типа в селекторе. Отправка сообщения с сигнатурой типа, которая не соответствует получателю, означает неопределенное поведение, поэтому в равной степени допустимыми являются подходы, реализованные в динамической библиотеке GNUstep (регистрация предупреждения или генерирование исключения) и в динамической библиотеке от компании Apple (разрушение стека).

Объявление классов

```
3 @interface Integer : NSObject
4 {
5     int integer;
6 }
7 - (int)intValue;
8 - (void)addInteger: (Integer*)other;
9 @end
10 @implementation Integer
11 - (int)intValue { return integer; }
12 - (void)addInteger: (Integer*)other
13 {
14     integer += [other intValue];
15 }
16 @end
```

Пример кода из файла `integer.m`

Для создания нового класса достаточно сделать следующее: описать его открытый интерфейс и написать его закрытую реализацию. А для этого нужно знать три ключевых слова Objective-C: **@interface**, **@implementation** и **@end**.

Обратите внимание на то, что все новые ключевые слова в Objective-C начинаются со знака @. Дело в том, что этот знак не разрешается использовать в идентификаторах C. Сравните эту ситуацию с использованием в C++ такого идентификатора, как `class`: любой код C окажется недопустимым, если в нем в качестве имени переменной используется идентификатор `class`.

В Objective-C наследуется модель компиляции от языка C, где компилятор ожидает получить один файл, а разделение осуществляется препроцессором, объединяющим разные файлы. По принятому соглашению интерфейсы для классов Objective-C помещаются в заголовочные файлы (с расширением `.h`), а их реализации — в файлы реализации (с расширением `.m`). Никто не может вспомнить, почему для файлов реализации было выбрано именно расширение `.m`. Вероятно, разработчики языка Objective-C предполагали, что буква “m” может обозначать “module” (модуль), “methods” (методы) или “messages” (сообщения).

Заголовочные файлы вставляются в единицу компиляции препроцессором еще до того, как их обнаружит компилятор Objective-C, и поэтому их разделение не является обязательным. Это же справедливо для C++ и непрозрачных типов данных в C.

Не совсем обычным в Objective-C является то обстоятельство, что переменные экземпляра (называемые в Java и C++ “полями”) объявляются как часть интерфейса, а не реализации. Это последствия того, как были реализованы ранние версии Objective-C. Каждый объект преобразовывался в структуру C, первое поле которой содержало указатель на соответствующий класс. Для того чтобы сформировать такую структуру, компилятор должен был в состоянии обнаружить все переменные экземпляра всех суперклассов. Один из неприятных побочных эффектов такого подхода заключался в том, что видоизменение расположения класса означало необходимость повторной компиляции всех его подклассов.

В настоящее время благодаря применению неуязвимого двоичного прикладного интерфейса ABI появилась возможность скрывать закрытые переменные экземпляра от подклассов. Каждая ссылка на них проходит через определенный уровень косвенности. Смещение теперь является переменной, а не константой времени компиляции. Современные компиляторы позволяют опре-

делять переменные экземпляра в контекстах директив `@implementation` и `@interface`.

По умолчанию переменные экземпляра Objective-C защищены, т.е. они доступны только для класса или его подкласса. Если объявить их с помощью ключевого слова **@private**, они будут доступны только для класса. А если объявить их с помощью ключевого слова **@public**, они будут доступны везде, что не совсем обычно. В связи с недостаточным контролем доступа и наличием механизмов его обхода все переменные экземпляра обычно оставляются защищенными, и лишь в закрытых классах они делаются открытыми.

На заметку

Созданный компанией Apple неузвимый двоичный прикладной интерфейс ABI ограничивает доступность переменных экземпляра, обозначающих смещение, выдавая систематическую ошибку при попытке доступа к закрытой переменной экземпляра из другой единицы компиляции. Тем не менее доступ к закрытым переменным экземпляра возможен посредством динамической интроспекции, но это средство защиты недоступно при использовании других динамических библиотек.

Базовая структура интерфейса класса делится на три части. В первой части определяется место, которое должен занять новый класс в иерархии, как показано ниже.

```
@interface имя_класса : имя_суперкласса <протоколы>
```

Имя класса указывается непосредственно после ключевого слова `@interface`. Далее следует двоеточие, а за ним — имя суперкласса. Конечно, можно разработать новые корневые классы, но сделать это правильно очень трудно, поэтому в качестве суперкласса обычно указывается `NSObject` или `NSProxy`.

После имени суперкласса можно (но необязательно) указать (через запятую) список протоколов, заключив его в угловые скобки. Протоколы в Objective-C подобны интерфейсам в Java. Они определяют списки методов, которые объект должен реализовать.

На заметку

Пользуясь самыми последними версиями компилятора, можете предварить объявление интерфейса отдельного класса префиксом `__attribute__((weak_import))`. Благодаря этому все ссылки на него становятся слабыми. Если вы отправите сообщение такому классу в отсутствие связи с ним, это будет равнозначно отправке сообщения константе `nil`. Подклассы отдельного класса, импортированные подобным способом, не будут разрешены до тех пор, пока не загрузится суперкласс. Это дает возможность определить класс, наследующий от класса в дополнительно загружаемом пакете.

Во второй части интерфейса класса указывается список переменных экземпляра. Здесь используется такой же синтаксис, как и в определении структуры в C. Определение простого класса, включающее только переменные экземпляра, но исключающее какие-либо методы, может иметь следующий вид:

```
@interface NoMethods : NSObject
{
    int integer;
    NSObject *object;
    float floatingPoint;
}
@end
```

Все, что допустимо в определении структуры C, допустимо и здесь. В языке Objective-C++ можно также добавить указатели на объекты C++ и те из них, для которых имеется тривиальный конструктор.

И наконец, в третьей части необходимо перечислить открытые методы, которые реализуются в объявляемом классе. Перейдем, однако, от теории к практике и рассмотрим ряд следующих примеров объявления методов:

```
+ (id)new;
- (void)appendString: (NSString*)astring;
```

Знак `+`, поставленный перед объявлением первого метода, означает, что в данном случае мы имеем дело с методом,

присоединенным к классу. Это означает, что классу можно отправить сообщение `+new` и он возвратит указатель на свой объект. В рассматриваемом здесь примере `id` обозначает новый тип, введенный в Objective-C и приблизительно подобный типу `void*` в C, но ограниченный указанием на объекты Objective-C.

Обратите внимание на то, что для указания возвращаемого типа и типов параметров метода во втором примере объявления метода используется такой же синтаксис, как и при приведении типов. Это лишний раз подчеркивает еще одно важное отличие отправок сообщений от вызовов функций C. Метод компилируется в функцию и вызывается по указателю, который обнаруживается во время выполнения. При отправке сообщения компилятор неявно приводит указатель на метод к функции, описываемой этими типами. Такой синтаксис призван также подчеркнуть, что аргумент не обязан иметь тип `NSString*`, но только тип, который можно привести к типу `NSString*`.

Часть определения класса, относящаяся к его реализации, выглядит намного проще. Она начинается с директивы `@implementation` и содержит определения всех методов класса, как показано ниже.

```
@implementation NewClass
+ (int)classversion
{
    return 2;
}
- (void)log
{
    fprintf(stderr, "log message received" );
}
@end
```

Стоит отметить, что объявлять все методы, реализуемые классом, совсем не обязательно. Методы, не объявленные в интерфейсе класса, считаются закрытыми. При их вызове будет сформировано предупреждение во время ком-

пиляции, а будущая версия данного класса может вообще перестать реагировать на них.

Применение протоколов

Подобно классам, протоколы в Objective-C являются объектами. Поведение протоколов изменилось с появлением динамической библиотеки, модернизированной компанией Apple. Раньше протоколы идентифицировались по имени, и поэтому два протокола считались равнозначными, если у них были одинаковые имена. Это означало, что если бы в двух разных библиотеках были объявлены протоколы под одним и тем же именем, то нельзя было бы сказать, какой из них реализован объектом. Но еще важнее то, что протокол нельзя было бы обнаружить и узнать, какие методы ему требовались.

С появлением версии Objective-C 2 протоколы стали уникальными. Теперь протокол можно получить по имени и тщательно инспектировать его. Иногда для этой цели использовалась старая модель, но все зависело от того, сформировал ли уже компилятор полное определение протокола.

Ссылку на протокол можно получить с помощью ключевого слова `@protocol()`. Основной причиной для этого служит необходимость проверить протокол на соответствие, как показано ниже.

```
if (![delegate conformsToProtocol: @protocol(
    DelegateProtocol)])
{
    testIndividualMethods = NO;
}
```

Код, аналогичный приведенному выше, можно было бы использовать в методе, задающем делегат. Если делегат соответствует протоколу, он гарантированно реализует все методы этого протокола, что избавляет от необходимости проверять каждый из них в отдельности. В противном случае каждое сообщение, скорее всего, придется проверить перед его отправкой.

Протоколы обеспечивают контроль типов как во время компиляции, так и во время выполнения кода. Заклучив имя объекта в угловые скобки после имени типа, можно потребовать, чтобы этот объект соответствовал протоколу. Ниже приведены характерные тому примеры.

```
id <NSCopying> a;  
id <NSObject, NSCopying> b;  
NSObject<NSCopying> *c;
```

В первом примере любой объект, реализующий протокол `NSCopying`, присваивается переменной `a`. А объекты, присваиваемые переменной `b`, должны реализовывать и протокол `NSObject`, что будет автоматически делать любой объект, наследующий от класса `NSObject` или `NSObject`. И наконец, переменная `c`, представляя подкласс, производный от класса `NSObject`, должна реализовывать протокол `NSCopying`.

Если присвоить объект переменной, которая требует, чтобы он соответствовал протоколу, не указанному в списке, объявленном для подобного соответствия, то компилятор сформирует предупреждение, а не ошибку. Это не систематическая ошибка, поскольку у объекта по-прежнему существует возможность реализовать все методы, требуемые протоколом, даже если этот объект не извещает о соответствии протоколу.

Добавление методов в класс

```
3 @interface NSObject (Logging)  
4 - (void)log;  
5 @end  
6 @implementation NSObject (Logging)  
7 - (void)log  
8 {  
9     NSLog(@"%@", self);  
10 }  
11 @end
```

Пример кода из файла `Logging.m`

К числу самых существенных отличий Objective-C от таких языков, как C++ или Java, относится понятие *категорий*. Категории появились еще в Smalltalk, но как способ группирования методов для составления документации. А в Objective-C они позволяют добавлять методы в класс, даже в отсутствие доступа к исходному коду.

По своей структуре объявления категорий подобны объявлениям классов. Категория не может изменить переменные экземпляра класса или его суперкласса, поэтому эти части директивы `@interface` опущены в ее объявлении. Имя категории указывается в скобках после имени класса. Простой пример интерфейса и реализации категории показан в начале этого раздела, где метод `-log` добавляется в класс `NSObject`. Сообщение `-log` может быть затем отправлено любому объекту и выведено на консоль.

Обратите внимание на то, что интерфейс и реализация категории необязательны. В следующем разделе будет разъяснено, почему определение интерфейса категории без соответствующей реализации может оказаться полезным. А указание реализации без интерфейса категории оказывается полезным в случае, если требуется просто *заменить методы* в объекте.

Методы, объявленные в категории, заменяют методы, объявленные в классе под тем же самым именем. Такой порядок гарантируется, поэтому при возникновении конфликта имен преимущество всегда будет иметь категория. Если имеются две категории, определяющие один и тот же метод, возникает неопределенность.

К числу малоизвестных и отсутствующих в документации свойств категорий относится возможность добавлять в класс соответствие протоколу. В среде Foundation имеются различные коллекции без общего суперкласса, и поэтому они не реализуют общий протокол. А в среде EtoileFoundation все они дополняются категориями, прини-

мая протокол `ETCollection`. Это дает возможность использовать данный протокол там, где требуется коллекция, независимо от ее типа.

```
3 @protocol Collection
4 - (BOOL)isEmpty;
5 @end
6 @interface NSArray (Collection) <Collection> @end
7 @interface NSSet (Collection) <Collection> @end
8 @implementation NSArray (Collection)
9 - (BOOL)isEmpty
10 {
11     return [self count] == 0;
12 }
13 @end
14 @implementation NSSet (Collection)
15 - (BOOL)isEmpty
16 {
17     return [self count] == 0;
18 }
19 @end
```

Пример кода из файла `collection.m`

В приведенном выше примере из файла `collection.m` демонстрируется сильно упрощенная разновидность описанной выше ситуации. Протокол `Collection` добавляется в классы `NSSet` и `NSArray`. Если отправить сообщение `-conformstoProtocol:` экземпляру любого из этих классов, передав `@protocol(Collection)` в качестве аргумента, он возвратит логическое значение `YES`.

Для категорий возможен особый случай, когда интерфейс категории можно объявить без имени. Это так называемое *расширение класса*. В отличие от других категорий, которые должны сопровождаться отдельной директивой `@implementation`, задающей имя категории, в данном случае предполагается, что методы, объявленные в расширениях класса, должны быть указаны в основной дирек-

тиве `@implementation` реализации в этом классе. Этим особым случаем можно воспользоваться для предваряющего объявления закрытых методов.

Используя самые последние версии компиляторов Objective-C, переменные экземпляра можно объявить как в расширениях класса, так и в контекстах `@implementation` (точно так же, как это делается в контекстах `@interface`). Оба эти механизма позволяют выносить подробности реализации из открытых заголовков.

Если объявляются закрытые переменные экземпляра, то их следует затем объявить в контексте `@implementation`. Это гарантирует от случайного доступа к ним из кода, которому такой доступ запрещен. Если же объявляются открытые или защищенные переменные экземпляра, которые должны быть доступны только в одной и той же библиотеке, то их нужно объявить в расширении класса, а именно в закрытом заголовке. В новом коде следует объявлять только те переменные экземпляра, которые, как предполагается, должны быть непосредственно доступны для пользователей этого кода в открытых заголовках.

Применение неформальных протоколов

```
3 @interface NSObject (InformalProtocol)
4 - (id)doSomething;
5 @end
6 // только для версии Objective-C 2:
7 @protocol InformalProtocol
8 @optional
9 - (id)doSomething;
10 @end
```

Пример кода из файла `informalProtocol.m`

Довольно часто приходится определять ряд методов, которые желательно (но необязательно) реализовать в некотором классе. Это особенно характерно для делегатов, когда

перед отправкой сообщения делегату требуется проверить, реализует ли он данный метод.

В версии Objective-C 2 протоколам разрешается содержать методы, объявленные как необязательные, а до ее появления практиковалась реализация неформальных протоколов с помощью категорий. Так, если создать интерфейс для некоторой категории в классе `NSObject`, но не предоставить для него определение интерфейса, компилятор предположит, что каждый подкласс, производный от класса `NSObject`, будет реагировать на сообщения, объявленные в интерфейсе, а следовательно, компилятор не выдаст предупреждение при отправке объекту любого из этих сообщений.

Упомянутые выше методы на самом деле реализуются далеко не каждым объектом. Поэтому перед отправкой сообщения следует непременно выяснить факт реализации подобных методов, используя сообщение `-respondToSelector:`.

Синтезирование методов с помощью объявляемых свойств

```
3 @interface Integer : NSObject
4 @property (nonatomic,assign) int integer;
5 @end
6 @implementation Integer
7 @synthesize integer;
8 @end
```

Пример кода из файла `synth.m`

К числу самых крупных дополнений Objective-C 2 относятся *объявляемые свойства*. Они позволяют определить некоторые аспекты семантики средств доступа, а затем синтезировать подходящие методы, если не требуется специальная обработка.

На заметку

Точечный синтаксис относится к числу самых противоречивых изменений, которые компания Apple внесла в Objective-C. Ведь он нарушает один из основных принципов Objective-C (в новой семантике должен всегда использоваться новый синтаксис), повторно используя синтаксис доступа к полю структуры для отправки сообщений. Точечным синтаксисом удобно пользоваться и для сокрытия неудачных проектных решений, поскольку он поощряет косвенные ссылки, которые нарушают закон Деметры.

Свойства также вводят новый синтаксис. Они реализуются с точки зрения отправки сообщений. Так, если создать методы `-integer` и `-setInteger:`, то для их вызова, помимо обычного способа, можно также воспользоваться точечной записью, как показано ниже.

```
obj.integer = 12;  
int i = obj.integer;  
obj.integer += 42;
```

Последний пример оказывается самым полезным. Если распространить его на традиционный синтаксис сообщений в Objective-C, то можно получить примерно следующее:

```
[obj setInteger: [obj integer] + 42];
```

² Закон Деметры (*Law of Demeter — LoD*) — правило проектирования программного обеспечения, в том числе объектно-ориентированного. В общем, Закон Деметры является частным случаем слабой связности. Сформулирован в Северо-восточном Университете (Бостон, Массачусетс, США) в конце 1987 года, а основной его смысл состоит в том, что объект должен иметь минимальное представление о структуре и свойствах чего угодно (включая собственные подкомпоненты). Название происходит от проекта “Деметра”, в котором использованы принципы аспектно-ориентированного и адаптивного программирования. Сам проект назван в честь Деметры, греческой богини земледелия, чтобы подчеркнуть преимущества методики программирования “снизу-вверх”. — Примеч. ред.

Не следует забывать, что оба варианта записи дадут совершенно одинаковый код. Первая запись кажется более эффективной, но на самом деле это не так. В обоих случаях выполняются две отправки сообщений и одно сложение. А это значительно более дорогостоящая (с точки зрения потребляемых вычислительных ресурсов) последовательность действий, чем в том случае, если бы объект `obj` был структурой `C`.

На заметку

Термин *свойство* применяется в документации на Smalltalk для обозначения переменной экземпляра, которой можно манипулировать, используя средства доступа. Этот термин применяется в таком же значении и в более ранней документации на Objective-C. Поэтому определение понятия *объявляемых свойств*, а не просто свойств порождает свести к минимуму возникающую неоднозначность.

Объявление свойства делится на две части: интерфейс и реализацию. Интерфейс определяет имя и тип свойства (здесь нет ничего неожиданного), но он также определяет семантику высокого уровня.

Свойства атомарны по умолчанию. Это влечет за собой некоторые издержки, поэтому большинство создаваемых свойств целесообразно определять как *неатомарные* (*nonatomic*). Атомарность методов доступа не делает объект *потокбезопасным*, и поэтому для совместного использования объектов в потоке лучше всего прибегать к высокоуровневой блокировке.

Рассмотрим приведенный ниже пример из файла `nonatomic.m`, где сначала объявляется атомарное свойство, а затем используется одна операция инкремента. На первый взгляд эта операция кажется атомарной, но на самом деле это не так. Единственный оператор инкремента в действительности разворачивается в две операции отправки сообщений и одну операцию инкремента в промежутке между ними.

```
3 @interface NotReallyAtomic : NSObject
4 @property int a;
5 @end
6
7 int add(NotReallyAtomic *obj)
8 {
9     obj.a += 1;
10 }
```

Пример кода из файла `nonatomic.m`

Вся эта атомарность в контексте свойств означает, что в отсутствие “сборки мусора” методы установки и получения попытаются запросить одну и ту же блокировку, поэтому объект будет освобожден из памяти не случайным образом, если попытаться установить его в одном потоке и получить в другом. Для “сборки мусора” и необъектных свойств неатомарность не имеет никакого смысла.

Свойства могут быть доступными для чтения и записи (`readwrite`) или только для чтения (`readonly`). Их отличие заключается лишь в поддержке операции присваивания. Так, если определить свойство с атрибутом `readwrite` и не пользоваться “сборкой мусора”, то придется также определить семантику метода установки значения этого свойства.

На заметку

Иногда требуется иметь свойство, доступное для чтения и записи, но допускающее только чтение в открытом интерфейсе. Подходящий для этой цели механизм предоставляют расширения классов. Так, если объявить свойство доступным только для чтения в главном интерфейсе, а затем еще раз объявить его доступным для чтения и записи в расширении класса, то с помощью директивы `@synthesize` в основной директиве `@implementation` будут созданы соответствующие методы установки и получения. Но всякие попытки использовать метод установки там, где расширение класса недоступно, приведут к ошибке во время компиляции.

Существует несколько вариантов поведения метода установки. Самый простой (и единственно возможный для неobjектных типов) вариант определяется атрибутом `assign`. В этом случае значение устанавливается обыкновенным присваиванием. Для неobjектных типов выполняется простое копирование значения, а для objектов дело обстоит немного сложнее.

Конкретная семантика различных средств доступа к свойствам зависит от режима управления памятью. При подсчете ссылок (автоматическом или вручную) атрибут свойства `assign` обеспечивает выполнение простого присваивания. Далее при доступе к значению возвращается указатель для неатомарных свойств или же он сохраняется, а затем автоматически освобождается для атомарных свойств.

Следующий вариант определяется атрибутом `retain` и имеет смысл только в режимах подсчета ссылок. При установке свойства с атрибутом `retain` новое значение сохраняется, а старое освобождается. А для “сборки мусора” свойства с атрибутами `assign` и `retain` равнозначны.

Свойство с атрибутом `copy` подобно свойству с атрибутом `retain`. При его установке для нового значения отправляется сообщение `-copy`, а не `-retain`. Таким вариантом доступа следует пользоваться для свойств, имеющих неизменяемые типы. Так, если объявить свойство типа `NSString*`, то использование атрибута `copy` гарантирует, что в итоге будет получена неизменяемая символьная строка, даже если кто-нибудь попытается установить ее значение с помощью экземпляра типа `NSMutableString*`. В результате отправки сообщения `-copy` неизменяемые объекты возвращают самих себя, а следовательно, будет создана только копия, если потребуется гарантировать, что данное свойство не будет случайно изменено.

Если используется автоматический подсчет ссылок или “сборка мусора” (подробнее об этом — в главе 3), то свойст-

во можно объявить с атрибутом `weak`. Такое свойство не защищает свое значение от удаления и устанавливает его равным `nil`, если все остальные ссылки удалены.

В современных компиляторах поддерживается также тип свойства `unsafe_unretained`, эквивалентный атрибуту `assign`. Но он более явно указывает на отсутствии безопасности.

Типы свойств, имеющие смысл только для объектов, можно также использовать для типов указателей C, объявляемых с помощью атрибута `__attribute__((NSObject))`. Этот атрибут сообщает компилятору, что тип C на самом деле является объектом Objective-C. Характерными тому примерами могут служить объекты и блоки в среде Core Foundation (см. раздел “Применение блоков” в конце этой главы). Они открыто доступны в заголовках, которые могут быть включены в программы на C, и поэтому не раскрывают типы Objective-C явно, но могут быть использованы в качестве объектов Objective-C в программах на Objective-C.

Объявив интерфейс свойства, необходимо обеспечить его реализацию. Ведь свойства — это лишь способ определить методы доступа. Свойство `a` в последнем рассмотренном примере реализуется в методах `-a` и `-setA:`. Первый из них возвращает значение свойства, а второй устанавливает в нем новое значение.

Эти методы вы можете реализовать самостоятельно, а также наследовать их от суперкласса. В последнем случае компилятор может оказаться не в состоянии обнаружить их, и тогда он сформирует предупреждение. Запретить выдачу такого предупреждения можно с помощью директивы `@dynamic` с указанием нужного свойства. Эта директива сообщит компилятору, что методы доступа на самом деле существуют и будут разрешены во время выполнения.

В качестве альтернативного варианта можно воспользоваться директивой `@synthesize`, которая вынудит компилятор сформировать методы доступа автоматически.

Синтезированные методы доступа к свойству обращаются к переменной экземпляра по тому же имени, что и у свойства. Эти методы вызывают функцию из динамической библиотеки, используя смещение в переменной экземпляра и ряд признаков, указывающих на разновидность свойства.

Как правило, рекомендуется пользоваться директивой `@synthesize`. В этом случае компилятор сформирует метод с правильной семантикой для текущего режима компиляции. Это позволит легко переключаться между режимами подсчета ссылок вручную или автоматически и “сборки мусора”, вообще не видоизменяя методы доступа, поскольку компилятор сделает это автоматически.

В более старых двоичных прикладных интерфейсах ABI языка Objective-C все переменные экземпляров нужно было объявлять в описании `@interface`, чтобы компилятор мог преобразовать их в структуры C. Но с появлением неизменяемого двоичного прикладного интерфейса ABI такая необходимость отпала. Директива `@synthesize` позволяет создавать новые переменные экземпляра, не являющиеся частью открытого интерфейса класса.

Представление об аргументах `self`, `_cmd` и `super`

```
8 id method(id self, SEL _cmd, ...);
```

Пример кода из файла `object.m`

Каждый метод Objective-C преобразуется во время компиляции в вызов функции, которая имеет два скрытых аргумента: `self` и `_cmd`. Назначение первого вполне очевидно: он является получателем сообщения. Если сообщение адресовано экземпляру, то аргумент `self` будет экземпляром класса, а если — классу, то он будет самим классом.

Ключевое слово `this` в C++ практически равнозначно `self` в Objective-C, но с одним существенным отличии-

ем: `self` является не ключевым словом, а именем аргумента. Присваивание значения элементу `this` в C++ приводит к ошибке, присваивание значения элементу `self` в Objective-C вполне допустимо.

Второй скрытый аргумент является *селектором* для передаваемого сообщения. Он требуется для нормального функционирования таких механизмов, как пересылка. При отправке сообщения объекту, которому неизвестно, как его обработать, будет сначала построен объект типа `NSInvocation`, инкапсулирующий данное сообщение, а затем передан методу `-forwardInvocation:`.

Эти скрытые аргументы не являются ключевыми словами, а следовательно, их можно применять вне методов в качестве идентификаторов, хотя в спецификации C для последующей реализации зарезервированы идентификаторы, начинающиеся с символа подчеркивания, и поэтому аргументом `_cmd` лучше не пользоваться. В данном случае соблюдаются обычные правила области действия, а следовательно, в любом методе можно также объявить переменную под именем `self`, скрывая другой вариант, передаваемый в качестве аргумента. Если поступить подобным образом, компилятор сформирует предупреждение, поскольку это не очень разумно, но с точки зрения семантики вполне допустимо.

С псевдопеременной `super` дело обстоит немного сложнее. Она допустима только для использования в качестве получателя сообщения. В этом случае она сформирует отправку сообщения, и аргумент `self` станет его получателем. Но поиск вызываемого метода будет осуществляться в суперклассе, а не в текущем классе. Именно таким образом и вызываются переопределяемые методы.

При отправке сообщений псевдопеременной `super` используется другая функция поиска методов, которая принимает в качестве аргумента указатель на структуру `objc_`

`super`. Эта структура содержит как целевой класс, так и целевой объект.

Суперкласс фиксируется во время компиляции. При компиляции класса Objective-C поиск всех сообщений, предназначенных для псевдопеременной `super`, будет происходить в классе, определенном в его интерфейсе как суперкласс.

Для большинства программ это не имеет особого значения, но об этом следует помнить, если для добавления методов в класс используются функции из динамической библиотеки. Если метод, присоединяемый к новому классу, отправляет сообщения псевдопеременной `super`, это сообщение будет отослано не тому классу.

Следует также иметь в виду, что отправки сообщений суперклассу реализуются в категориях иначе, если используется динамическая библиотека GNU. Контекст реализации класса определяется такой же единицей компиляции, как и структура класса, поскольку структура класса формируется компилятором из кода, содержащегося в контексте `@implementation`. Указатель на суперкласс можно найти в результате простого просмотра этого контекста и получения поля суперкласса.

Структура класса недоступна в категории. Ее следует искать во время выполнения кода. Это замедляет отправку сообщений суперкласса из методов, определенных в категориях. В качестве выхода из положения динамические библиотеки от компании Apple предоставляют компоновщику самому разрешать внешние ссылки на классы.

Причина для динамического поиска состоит в поддержке замещения класса — способа замены одним классом определения другого класса во время выполнения. В современной (Modern) библиотеке от компании Apple способ замещения класса больше не поддерживается. Каждый класс экспортирует свою структуру как открытый символ, поэто-

му компоновщик разрешает указатели на классы во время загрузки программы.

Взаимодействие между замещением класса и категориями зависит от отличий в динамических библиотеках в компании Apple и GNU, но вы можете избежать лишних хлопот, просто отказавшись пользоваться замещением класса. Это все равно неплохой выход из положения. Ведь существуют крайние случаи, когда замещение класса может продемонстрировать неожиданное поведение, и очень легко написать код, который будет прекрасно работать на одном компьютере, но окажется совершенно неработоспособным на другом компьютере почти такой же конфигурации.

Представление об указателе isa

```
1 typedef struct objc_object *id;  
2  
3 struct objc_object  
4 {  
5     id isa;  
6 };
```

Пример кода из файла `object.m`

В языке C++ классы представляют собой структуры. Из структуры C (подобно классу C++) можно создать подкласс. При вызове метода для класса C++ в действительности вызывается функция C, имя которой немного откорректировано, а в качестве ее первого (скрытого) аргумента служит указатель на структуру.

В Objective-C структуры и классы сильно отличаются друг от друга. До появления неуязвимого двоичного прикладного интерфейса ABI в определении структуры можно было использовать ключевое слово `@defs`, чтобы создать структуру, формат которой совпадал бы с форматом класса. Но классы и структуры все равно нельзя было использовать как взаимозаменяемые языковые конструкции.

Объект отличается от структуры тем, что ему как экземпляру класса можно послать сообщение. Дело в том, что первая переменная экземпляра в структуре является указателем на структуру класса. Эта переменная экземпляра по соглашению называется `isa` (от *англ.* “this object is a...”, т.е. “этот объект является...”). Определяя новый корневой класс, не забудьте добавить в него эту переменную экземпляра.

Подобно скрытым аргументам `self` и `_cmd`, указатель `isa` является обычной переменной экземпляра, а не ключевым словом. Рассмотрим для сравнения вызов виртуального метода в C++. Любой объект C++, у которого имеются виртуальные методы, содержит (скрытый) указатель на таблицу виртуальных функций. Эта таблица представляет собой простой массив указателей на функции. При вызове виртуального метода компилятор разыменовывает указатель в этой таблице и вызывает функцию с правильным смещением.

Таблицу виртуальных функций нельзя использовать с целью выяснить что-нибудь об объекте, а его расположение является частью определения двоичного прикладного интерфейса ABI для целевой платформы, а не частью самого языка. Совсем иначе дело обстоит с указателем `isa`. Ведь этот указатель представляет собой определенную явным образом переменную экземпляра, а следовательно, является доступным подобно любой другой переменной экземпляра. Он указывает на тип `Class`, который является типом `typedef`, предоставляемым в заголовках динамической библиотеки Objective-C³ и определяемым (в зависимости от разновидности библиотеки) как непрозрачный тип или указатель на открытую структуру `C`.

В тех динамических библиотеках, где точное расположение структуры класса закрыто, можно все равно найти функции, которые позволят инспектировать ее. С их по-

³ В отсутствие определения компилятор Clang интерпретирует его как встроенный тип.

мощью можно выяснить смещения переменных экземпляра, имена и сигнатуры типов для методов и свойств, а также имя и тип суперкласса.

Указатель `isa` имеется и в других объектах. Тип `id` определен как структура `C` (без всякого контроля доступа), содержащая только указатель на класс, поэтому его можно читать подобно любым другим полям в структуре.

В самой последней версии Objective-C непосредственный доступ к указателю `isa` практически исключен. Поэтому для доступа к нему придется воспользоваться функциями `object_getClass()` и `object_setClass()`.

В самых последних версиях динамических библиотек заимствован прием из Smalltalk, с помощью которого мелкие объекты скрываются в указателе. Объекты всегда выравниваются по словам, поэтому младшие биты указателя всегда равны 0. Это позволяет сохранять 31-разрядное целочисленное значение в 32-разрядном указателе, причем его самый младший разряд устанавливается в 1, чтобы отличать указатель от настоящего объекта. Если интерпретировать такое значение как указатель и попытаться получить непосредственный доступ к указателю `isa`, последствия могут оказаться весьма плачевными.

Объекты Objective-C способны пересылать сообщения, а имеющийся указатель на объект может оказаться промежуточным объектом. Если же требуется получить класс другого объекта, сообщение `-class` следует отправить именно ему, а не промежуточному объекту. Это, конечно, более медленный способ, но он не нарушает присутствие промежуточных объектов.

Для того чтобы выяснить, является ли объект промежуточным, достаточно отправить ему сообщение `-isProxy`. Но отправляя одно сообщение, вы можете точно так же отправить ему еще одно сообщение, чтобы получить класс.

Инициализация классов

```
4  @implementation Init
5  + (void)load
6  {
7      NSLog(@"+load called");
8  }
9  + (void)initialize
10 {
11     NSLog(@"+initialize called");
12 }
13 - (id)init
14 {
15     NSLog(@"-init called");
16     return self;
17 }
18 @end
19
20 int main(void)
21 {
22     [NSAutoreleasePool new];
23     NSLog(@"main() entered");
24     id init = [[Init alloc] init];
25     init = [[Init alloc] init];
26     return 0;
27 }
```

Пример кода из файла classInit.m

В C++ можно выполнить статическую инициализацию, присвоив результат вызова функции глобальной или статической переменной. Для того чтобы добиться аналогичного результата в C, можно воспользоваться такими расширениями GCC, как `__attribute__((constructor))`. Но в обоих случаях используется специальная функция, которая должна быть вызвана до функции `main()`.

Очень трудно в подобных случаях избежать ошибок, поскольку порядок вызова не определен. В Objective-C имеется эквивалентный механизм. Метод `+load` вызывается для клас-

са в том случае, если этот класс регистрируется средствами динамической библиотеки, т.е. до вызова функции `main()`.

Из этого метода совсем небезопасно отправлять произвольные сообщения. С помощью метода `+load` будут гарантированно загружены сам класс, все его суперклассы и строковый константный класс. Возможно, загрузка пройдет успешно и для других классов, но гарантировать ничего нельзя, и даже если у вас такой код работает нормально, он может не работать при незначительном изменении порядка загрузки.

Более безопасной альтернативой является метод `+initialize`. Он вызывается автоматически во время выполнения, когда классу отправляется первое сообщение. Обычно первым отправляется сообщение `+alloc`, чтобы создать экземпляр класса, но возможны и другие варианты, состоящие в запросе имени класса или одиночного экземпляра.

Метод `+initialize` позволяет выполнить инициализацию элементов, необходимых классу, по требованию. Его вызов не произойдет до выполнения функции `main()`, поэтому все компоненты будут уже загружены, и процесс окажется во вполне определенном состоянии.

Реализовав оба эти метода, можно выяснить порядок выполнения. Первый метод `+load` будет вызван, как только загрузится класс. По завершении работы динамического загрузчика будет вызвана функция `main()`. При отправке сообщения классу настанет черед для вызова метода `+initialize`, а затем — метода, который был фактически вызван, и только после этого можно будет отправить сообщение `-init` возвращаемому экземпляру.

```
1 a.out[2690:903] +load called
2 a.out[2690:903] main() entered
3 a.out[2690:903] +initialize called
4 a.out[2690:903] -init called
5 a.out[2690:903] -init called
```

Результат выполнения кода из файла `classInit`

Обратите внимание на то, что в приведенном выше результате выполнения кода из файла `classInit` метод `+initialize` был вызван только один раз. При второй отправке сообщение доставляется классу непосредственно. Вызов метода `+initialize` является также потокобезопасным. Если отправить одновременно два сообщения одному и тому же классу из разных потоков, один из них будет заблокирован до тех пор, пока другой не завершит выполнение метода `+initialize`, после чего оба потока продолжают свою работу.

Чтение кодировок типов

```
3  typedef struct _s
4  {
5      int i;
6      float f;
7      NSArray *a;
8      char *str;
9      unsigned short shorts[5];
10 } s;
11
12 int main(void)
13 {
14     const char *encoding = @encode(s);
15     printf("Encoding: %s\n", encoding);
16     return 0;
17 }
```

Пример кода из файла `type.m`

В Objective-C многое зависит от интроспекции (т.е. самоанализа логики поведения) во время выполнения кода. Объекты Objective-C можно проанализировать, отправив им сообщения, но этого нельзя сделать в отношении типов C. В качестве выхода из этого положения в Objective-C определен способ кодировки типов C в виде символьных

строк. Каждый примитивный тип кодируется одним символом. Например, тип `int` кодируется символом `"i"`, а тип `unsigned long long` — символом `"Q"`. Для всех целочисленных типов применяется соглашение о прописных и строчных буквах, представляющих варианты типов без знака и со знаком соответственно.

Как показывает приведенный ниже результат выполнения кода из файла `type.m`, в эти кодировки типов могут быть включены любые составные типы `C`.

```
1 Encoding: {_s=if@[5S]}
```

Результат выполнения кода из файла `type`

В данном примере демонстрируется кодировка типов для структуры, содержащей примитивные типы `C`, объект Objective-C и массив `C`. Кодировка типов генерируется компилятором в ответ на директиву `@encode()`, которая используется подобно оператору `sizeof()`, принимая в качестве аргумента тип и возвращая в качестве результата кодировку данного типа. Сама структура обозначается парой фигурных скобок, в которых указывается имя структуры, знак равенства и кодировки типов для каждого из ее элементов.

Следующие (после знака равенства) четыре символа обозначают кодировки типов в первых четырех полях структуры. Здесь необходимо отметить два важных момента. Во-первых, все объекты Objective-C кодируются знаком `"@"` независимо от сведений об их статических типах. Во-вторых, тип `char*` кодируется знаком `"*"`. А все остальные указатели кодируются знаком `"^"`, после которого указывается тип указателя, но к символьным строкам `C` применяется особый подход.

Остальная часть рассматриваемой здесь кодировки относится к массиву. В квадратных скобках сначала указы-

вается количество элементов, а затем тип одного элемента массива.

В Objective-C кодировки типов применяются везде, где выполняется интроспекция на уровне типов C. Запросив сведения о типах метода или переменной экземпляра, вы получите кодировку типа. Поэтому очень важно знать, как с ней обращаться. В среде EtoileSerialize я лично пользуюсь этими сведениями для циклического обращения к кодировкам типов переменных экземпляра и автоматического их упорядочения в последовательном виде (так называемой сериализации).

Применение блоков

```
3  int(^getCounter(void))(void)
4  {
5      __block int counter;
6      int(^block)(void) =
7      ^{ return counter++; };
8      return Block_copy(block);
9  }
10
11 int main(void)
12 {
13     [NSAutoreleasePool new];
14     int(^block)(void) = getCounter();
15     block(); block();
16     NSCAssert(block() == 2,
17         @"Block counted incorrectly");
18     int(^block2)(void) = getCounter();
19     block2(); block2();
20     NSCAssert(block2() == 2,
21         @"Block didn't start from 0");
22     return 0;
```

Пример кода из файла block.m

С выходом версии Mac OS X 10.6 разработчики компании Apple ввели блоки в C. Несмотря на их наличие в расширении языка C, чаще всего они используются в сочетании с Objective-C.

Блоки представляют собой *замыкания*, иногда еще называемые *лямбда-выражениями*. Это понятие заимствовано из языка Smalltalk, где замыкания инкапсулирует класс BlockClosure. Синтаксис для создания блоков выглядит довольно неприглядно. Он основан на синтаксисе C для указателей на функции, но вместо знака звездочки в нем используется знак вставки.

Блоки можно использовать почти так же, как и указатели на функции. Они вызываются подобно функциям, но их нельзя привести к указателям на функции, поскольку в них используется другое соглашение о вызовах. Подобно методам, у блоков имеются скрытые аргументы.

В примере, приведенном в начале этого раздела, функция `getCounter()` возвращает новый блок, который реализует счетчик. При каждом последующем вызове этот блок будет возвращать значение, на единицу большее, чем значение, возвращаемое при предыдущем вызове. Эта конструкция оказывается вполне работоспособной, поскольку переменная `counter` объявлена в той области действия, где был создан этот блок, с помощью спецификатора `__block`.

Любые переменные в стеке, на которые ссылаются блоки, будут скопированы в область “кучи”, а ссылки на эти переменные сохраняются в блоке и подсчитываются. При вызове функции `Block_copy()` сам блок (если он находится в стеке) копируется в “кучу”, а иначе его счетчик ссылок инкрементируется. Функция `Block_copy()` в действительности является макрокомандой, которая заключает в оболочку функцию `_Block_copy()` и преобразует возвращаемый тип `void*` в тип аргумента.

У каждого блока, возвращаемого функцией `getCounter()` в рассматриваемом здесь примере, имеется копия переменной `counter`, поскольку она является автоматической (стековой) переменной, и поэтому при последующих обращениях к этой функции всякий раз создается новая версия переменной `counter`.

В самой последней версии среды Foundation можно обнаружить немало методов, принимающих блоки в качестве аргументов. Например, используя блок в качестве аргумента, можно организовать циклическое обращение к массиву.

Блоки не вносят в C никаких выразительных средств. Блок — это просто указатель на функцию (а указатель на данные скрыт внутри единственной структуры) с некоторой дополнительной логикой в компиляторе, обеспечивающей вставку указателя на данные в качестве скрытого аргумента, передаваемого функции при ее вызове.

Это совсем не означает, что блоки не приносят никакой пользы. Прежде всего, блок не позволяет сделать ничего такого, что нельзя сделать в языке C. Он просто вносит определенную синтаксическую изюминку, которая упрощает некоторые действия. И это относится ко всему языку Objective-C в целом.

Как и у объектов Objective-C, у блоков имеется указатель `isa`, и им можно пользоваться для отправки сообщений. Они являются экземплярами закрытого абстрактного класса `NSBlock`, который реализует протокол `NSCopying`.

Большая часть “объектного” поведения блоков не описана в документации. Блоки можно хранить в коллекциях Objective-C независимо от их способности воспринимать специальные сообщения. После выхода последних версий компилятора Clang блоки стали поддерживать интроспекцию, раскрывая свои типы в виде кодировок в формате Objective-C. Более подробно блоки рассматриваются в главе 15.

Управление памятью

Если у вас имеется некоторый опыт программирования на C или C++, то вам, возможно, приходилось отслеживать принадлежность объектов и вручную выделять память для них, а затем освобождать эту память, разрушая их. А если вы программировали раньше на таком языке, как Java, то, скорее всего, привыкли к тому, что обо всем этом вместо вас заботился “сборщик мусора”.

В Objective-C на уровне языка не предусмотрено никаких мер для выделения и освобождения оперативной памяти. Эта обязанность возлагается на код C. Как правило, для размещения объектов в памяти их классу отправляется сообщение `+alloc`, что в дальнейшем приводит к вызову функции вроде `malloc()` с целью выделить область оперативной памяти для конкретного экземпляра класса. А при последующей отправке этому экземпляру сообщения `-dealloc` память, занимаемая его переменными, будет освобождена, и таким образом данный экземпляр будет удален из нее.

Среда Foundation добавляет механизм подсчета ссылок в этот простой алгоритм управления памятью вручную. Если вы ясно понимаете принцип его действия, то вам будет намного легче программировать на Objective-C. Новейшие версии компиляторов данного языка даже предоставляют в этом некоторую помощь, избавляя программирующего от необходимости самому писать код для подсчета ссылок.

Сохранение и освобождение объектов из памяти

```
6 NSArray *anArray = [NSArray array];  
7 anArray = [[NSArray alloc] init];  
8 [anArray release];
```

Пример кода из файла retainRelease.m

Каждый объект, наследующий от класса NSObject, имеет связанное с ним количество ссылок. Когда это количество ссылок достигает нулевого значения, объект разрушается. Объект, созданный с помощью сообщения `+alloc` или любого из связанных с ним методов (например, `+new` или `+allocWithZone:`), начинает свое существование с количества ссылок, равного единице.

Для управления количеством ссылок на объект ему отправляются сообщения `-retain` и `-release`. Как следует из названий этих сообщений, они используются для сохранения (`-retain`) ссылки на объект или освобождения (`-release`) существующей ссылки на объект. В частности, сообщение `-retain` инкрементирует количество ссылок на объект, а сообщение `-release` декрементирует это количество.

Объекту можно также отправить сообщение `-retainCount`, чтобы определить текущее количество ссылок на него. Возникает сильное искушение воспользоваться этим сообщением в целях оптимизации, а также в тех особых случаях, когда есть полная уверенность в существовании только одной ссылки на объект. Но делать это все же не рекомендуется. Судя по названию данного сообщения, вызываемый при его отправке метод сообщит количество *сохраняемых* ссылок на объект, а не общее их количество. При создании указателя на объекты в стеке отправлять им сообщение `-retain`, как правило, нецелесообразно. Это означает, что ссылки на объект могут существовать в двух или более местах, хотя подсчет сохра-

няемых ссылок на него свидетельствует о наличии только одной ссылки.

Указатели в Objective-C делятся на две категории: *владеющие* и *не владеющие ссылки*. Владеющей называется такая ссылка, которая учитывается при подсчете сохраняемых ссылок на объект. Вызывая метод `+new` или `-retain`, вы получаете в итоге владеющую ссылку на новый объект. А в результате вызова большинства других методов возвращается не владеющая ссылка. Переменные экземпляра и глобальные переменные, как правило, являются владеющими указателями, поэтому им следует присваивать владеющие ссылки. При выполнении операции присваивания необходимо также удостовериться в том, что удаляется существующая владеющая ссылка на объект, отправляя ему сообщение `-release`.

Временные переменные обычно являются не владеющими ссылками. В таких механизмах, как автоматический подсчет ссылок и “сборка мусора” (подробнее о них речь пойдет далее в главе), применяются специальные разновидности не владеющих ссылок.

Присваивание переменным экземпляра

```
26 - (void)setStringValue: (NSString*)aString
27 {
28     id tmp = [aString retain];
29     [string release];
30     string = tmp;
31 }
```

Пример кода из файла `ivar.m`

Используя механизм подсчета ссылок, необходимо действовать аккуратно и внимательно. Рассмотрим следующий простой пример метода установки:

```
14 - (void)setStringValue: (NSString*)aString
15 {
16     [string release];
17     string = [aString retain];
18 }
```

Пример кода из файла `ivar.m`

На первый взгляд, исходный код этого метода написан вполне благоразумно. Сначала освобождается ссылка на старое значение, а затем новое значение сохраняется и присваивается переменной. Такой код оказывается зачастую вполне работоспособным, но не всегда, и найти ошибку в нем не так-то просто.

Что произойдет, если, например, значения переменных экземпляра `aString` и `string` окажутся одинаковыми? В этом случае одному и тому же объекту отправляется сначала сообщение `-release`, а затем сообщение `-retain`. Если в каком-нибудь другом коде сохраняются ссылки на этот объект, он будет по-прежнему работать, а иначе первое сообщение заставит данный объект разрушиться, а второе будет отправлено висячему указателю на несуществующий объект.

В более корректной реализации этого метода следовало бы сначала сохранить новый объект, как показано в примере кода, приведенном в начале этого раздела. Обратите внимание на то, что в операторе присваивания используется результат отправки сообщения `-retain`, поскольку при сохранении некоторые объекты могут возвращать другие объекты. Это случается довольно редко, но все же бывает.

И наконец, рассматриваемый здесь метод не является потокобезопасным. Если требуется написать потокобезопасный метод установки, сначала придется сохранить новое значение, выполнить атомарную операцию обмена между результатом и переменной экземпляра, а затем освободить старое значение. Но, как правило, практиче-

ски невозможно найти разумное обоснование для написания кода, который поддерживал бы столь мелкоструктурный параллелизм. Ведь связанные с этим издержки на кеширование сведут на нет любой выигрыш в производительности, достигнутый благодаря параллелизму, а следовательно, овчинка выделки не стоит. Но если это действительно нужно, то лучше воспользоваться объявляемыми свойствами, чтобы синтезировать метод доступа автоматически, чем пытаться написать его вручную.

Автоматический подсчет ссылок

С выходом версий операционных систем iOS 5 и Mac OS X 10.7 компания Apple внедрила механизм *автоматического подсчета ссылок* (Automatic Reference Counting — ARC). Концептуально этот механизм вынуждает компилятор автоматически выяснять, когда должны быть вызваны методы `-retain` и `-release`, чтобы вызывать их автоматически. Но реализуется механизм ARC на практике несколько сложнее.

На заметку

В большинстве примеров, приведенных в этой книге, используется подсчет ссылок вручную. И это не случайно. Несмотря на то что механизм ARC рекомендуется применять для разработки новых программ, существует немало унаследованного кода, написанного еще до внедрения ARC. Но даже применяя механизм ARC, следует хорошо понимать семантику операций сохранения и освобождения объектов из памяти, которая неявным образом автоматически добавляется в создаваемый код. Программисту, хорошо освоившему подсчет ссылок вручную, будет легче перейти к механизму ARC. Ведь всякому программисту полезно иметь хотя бы самое общее представление о наборе инструкций центрального процессора, даже если он никогда не писал код на языке ассемблера. Это же относится и к механизму ARC.

Вместо того чтобы вставлять отправки сообщений непосредственно в код, препроцессор компилятора вставляет в него вызовы таких функций, как `objc_retain()` и `objc_release()`. А затем оптимизатор пытается объединить или исключить эти вызовы. В простых случаях эти функции действуют аналогично отправке сообщений, но зачастую эти функции выполняются гораздо эффективнее.

Если ваш код не особенно сложный, то, применяя механизм ARC, вы можете забыть об управлении памятью. В последней версии интегрированной среды разработки XCode механизм ARC применяется по умолчанию. Если вы выполняете компиляцию своих программ из командной строки или в другой встроенной системе, добавьте в вызов компилятора параметр `-fobjc-arc`, и можете просто забыть о сохранении и освобождении объектов.

Но если бы все было так просто, то жить на свете было бы намного приятнее. К сожалению, у механизма ARC имеются некоторые ограничения. В частности, он формализует размытые границы между памятью C и объектами Objective-C. Механизм ARC делит указатели на три категории. *Строгие указатели* соблюдают ту же семантику сохранения и освобождения, которая была рассмотрена выше. *Нестрогие указатели* (о них речь пойдет чуть позже) соответствуют не владеющим ссылкам, которые автоматически обнуляются при разрушении объекта. *А небезопасные несохраняемые указатели* просто игнорируются механизмом ARC, и это означает, что вам придется брать на себя ответственность за отслеживание срока действия объектов, сохраняемых по этим указателям.

По умолчанию все указатели на объекты в переменных экземпляра или в стеке являются строгими. А указатели на объекты в структурах по умолчанию не определены. Их необходимо отметить явным образом с помощью описателя принадлежности `__unsafe_unretained`. Несмотря на то

что это единственный разрешенный описатель принадлежности для указателей данной категории, его нужно указывать явно, оставляя тем самым напоминание для тех, кто будет изучать код, что механизм ARC просто проигнорирует подобные указатели.

Возврат объектов по аргументам-указателям

```
3  __weak id weak;
4  int writeBack(id *aValue)
5  {
6      *aValue = [NSObject new];
7      weak = *aValue;
8      return 0;
9  }
10
11 int main(void)
12 {
13     @autoreleasepool
14     {
15         id object;
16         writeBack(&object);
17         NSLog(@"Object: %@", object);
18         object = nil;
19         NSLog(@"Object: %@", weak);
20     }
21     NSLog(@"Object: %@", weak);
22     return 0;
23 }
```

Пример кода из файла writeback.m

В режиме ARC работать с аргументами типа “указатель на указатель” не так-то просто. Такие аргументы обычно используются в двух целях: для передачи массивов или возврата объектов. Если массив передается вниз по стеку, он должен быть объявлен как `const`. Этим компилятор уведомляется о том, что массиву ничего не будет присваивать-

ся в вызываемом коде, и поэтому механизм ARC сможет передать массив без сложного взаимодействия.

Но если возвращается объект, то механизм ARC генерирует довольно сложный код. В примере, приведенном в начале этого раздела, при обращении к функции `writeBack()` генерируется приблизительно следующий код:

```
id tmp = [object retain];
writeBack(&tmp);
[tmp retain];
[object release];
object = tmp;
```

В функции `writeBack()` новый объект будет автоматически освобожден до сохранения его во временной переменной. Это означает, что в конце данного кода переменная `object` будет содержать владеющую ссылку на этот новый объект.

Если бы переменная `object` была объявлена как автоматически освобождаемая (`__autoreleasing id`), то сгенерированный код оказался бы намного более простым. В этом случае будет автоматически освобождаться значение, первоначально сохраненное в переменной `object`, но это не вызовет никакого действия, поскольку начальное значение для всех указателей на объекты (даже при автоматическом распределении памяти) равно `nil`. В итоге указатель будет передан непосредственно вызываемому коду, чтобы сохранить в этой переменной владеющий (автоматически освобождаемый) указатель, если он видоизменяется.

При выполнении кода из этого примера обнаружится, что слабая ссылка обнуляется только в том случае, если разрушается автоматически освобождаемый пул. Функция `writeBack()` в любом случае сохраняет автоматически освобождаемый объект в переданном ей в качестве аргумента указателе и вообще не освобождает переданное значение. Вызывающий код всегда несет ответственность за то, чтобы переданное значение было не владеющей ссылкой. А для

этого переданное значение должно быть указателем на автоматически освобождаемый объект или копией указателя на сохраняемый объект.

Если же обозначить параметр как `out`, что разрешается только для параметров методов, но не для параметров функций C, то в вызываемом коде значение этого параметра гарантированно не будет прочитано. Поэтому компилятор пропустит стадию, на которой создается копия указателя в переменной `object`, еще до вызова метода.

Если требуется передать несколько объектов вверх по стеку, то следует вернуть экземпляр класса `NSArray`. Возможны и другие варианты, но они настолько сложны, что игра просто не стоит свеч: при их реализации легко сделать ошибку и затем потерять массу времени на ее обнаружение при отладке. И даже если удастся сделать все правильно, то в конечном итоге может оказаться, что такой сложный вариант будет работать медленнее, чем тот, в котором используется экземпляр класса `NSArray`.

Если же передается несколько значений вниз по стеку, то в качестве параметра следует объявить тип массива с явно заданным описателем принадлежности вместо типа указателя. Например, следует выбрать `__unsafe_unretained id[]` вместо типа `id*`. Благодаря этому отпадает необходимость в применении механизма обратной записи.

Исключение циклов сохранения

```
19 -(void)setDelegate: (id)aDelegate
20 {
21     delegate = aDelegate;
22 }
```

Пример кода из файла `ivar.m`

Недостаток механизма подсчета ссылок в чистом виде заключается в том, что он не может выявлять циклы. Так,

если имеются два объекта, сохраняющие ссылки друг на друга, то ни один из них вообще не будет освобожден.

Зачастую это не вызывает особых осложнений. Ведь структуры данных Objective C, по существу, нециклически, но иногда циклы возникают неизбежно. Чаще всего это происходит в *шаблоне делегирования*. В этом шаблоне объект обычно реализует некоторые механизмы и делегирует правила их применения другому объекту. Подобным образом действует большинство классов из среды AppKit.

Делегату требуется ссылка на объект, а объекту — ссылка на свой делегат, что сразу же приводит к возникновению цикла. Во избежание подобной ситуации обычно рекомендуется программировать так, чтобы объекты не сохраняли свои делегаты. Если объект передается в качестве аргумента методу `setDelegate:`, необходимо, чтобы ссылка на него сохранялась в некотором другом объекте, иначе он будет преждевременно удален.

Для этой цели в режиме ARC имеются два варианта выбора: пометить переменную экземпляра как `__weak` или как `__unsafe_unretained`. Первый вариант считается более безопасным, поскольку он гарантирует невозможность создания висящего указателя на несуществующий объект. При разрушении делегата переменная экземпляра принимает значение `nil`.

Применению нестрогого указателя присущи два недостатка. Первым из них является переносимость. Нестрогие указатели действуют в среде iOS 5, Mac OS X 10.7 и GNUstep, но они не функционируют в более старых версиях iOS или Mac OS X. Вторым недостатком является быстродействие. Каждый доступ к нестрогому указателю реализуется через вспомогательную функцию, которая проверяет, находился ли объект в процессе освобождения из памяти, и сохраняет его, если он все еще действителен, а иначе возвращает значение `nil`.

С другой стороны, небезопасные несохраняемые указатели представляют собой обычные указатели. Они легко доступны и действуют в любой целевой среде развертывания прикладных программ. Но их главный недостаток заключается в том, что ответственность за исключение попыток доступа к ним после того, как указатель будет освобожден из памяти, возлагается на самого программирующего.

Нестрогими указателями целесообразно пользоваться для отладки, а небезопасными несохраняемыми указателями — для развертывания прикладных программ. Прежде чем отправлять указателю какие-либо сообщения, следует ввести отладочные операторы утверждения, чтобы проверить, что он не равен значению `nil`, и тогда при появлении программных ошибок их можно будет обработать на основании полученных данных, исключив аварийное завершение программы.

Переход к механизму ARC

Если вы начинаете работать над новым проектом в среде XCode, механизм ARC выбирается по умолчанию, и нет особых оснований, чтобы не пользоваться им. Если же вы работаете с уже существующей кодовой базой и в вашем проекте подсчет ссылок выполняется вручную, у вас есть возможность сэкономить трудозатраты на разработку в долгосрочной перспективе, перейдя к механизму автоматического подсчета ссылок (ARC). Но для такого перехода недостаточно одного лишь переключения компилятора в другой режим.

Для подобного перехода в компиляторе Clang предоставляется инструментальное средство, которое попытается переписать код Objective-C, чтобы применить механизм ARC. Его можно вызвать из командной строки с помощью аргументов `-ccc-arcmt-check` и `-ccc-arcmtmodify`. Первый из них задает режим уведомления о тех элементах кода, которые нель-

зя преобразовать автоматически, а второй — режим перезаписи (видоизменения) исходного файла в отсутствие ошибок.

Это инструментальное средство перехода оказывается вполне пригодным для простых программ на Objective-C. Очевидно, что оно, прежде всего, удаляет все отправки сообщений `-retain`, `-release` и `autorelease`. Кроме того, из метода `dealloc` удаляется явный вызов `[super dealloc]`, который вставляется компилятором автоматически. Механизм ARC автоматически освобождает все переменные экземпляра, поэтому вам придется лишь реализовать метод `-dealloc` полностью для освобождения памяти, выделяемой функцией `malloc()` или аналогичным образом.

На заметку

На самом деле для освобождения переменных экземпляра в механизме ARC создается метод `-cxx_destruct`. Этот метод изначально создавался для автоматического вызова деструкторов C++ при разрушении объекта. Действие механизма ARC заметно отличается тем, что переменные экземпляра в Objective-C теперь освобождаются не до, а после завершения работы метода `-dealloc` в корневом классе. Но зачастую это не должно иметь никакого значения.

Если вы реализуете собственные методы подсчета ссылок, вам придется удалить их. Одной из наиболее распространенных причин для этого служит предотвращение случайного освобождения одиночных экземпляров. Это не так важно, если применяется механизм ARC, поскольку маловероятно, чтобы программная ошибка могла привести к преждевременному удалению объекта.

Самые большие трудности могут возникнуть при попытке сохранить указатели Objective-C в структурах C. Во избежание этого проще всего пользоваться объектами Objective-C с открытыми переменными экземпляра. Это позволит компилятору корректно управлять памятью для размещения в ней объекта и его полей.

Использование структур, ссылающихся на объекты Objective-C, считается безопасным лишь при их передаче вниз по стеку. В этом случае указатели на объекты могут быть без особых затруднений описаны как `__unsafe_unretained`, при условии, что они остаются действительными в вызывающем коде.

После использования инструментального средства перехода к механизму ARC никаких свойств `assign` вообще не останется. Они будут переписаны как `unsafe_unretained` или `weak`, в зависимости от того, поддерживаются ли слабые ссылки в целевой среде, выбранной для развертывания прикладных программ. Возможно, некоторые из них придется заменить ссылками типа `unsafe_unretained`, если они служат для прерывания простых циклов, тогда как слабые ссылки снижают, как оказывается, производительность.

Инструментальное средство перехода попытается по мере необходимости вставить операции приведения типов `__bridge`, но это стоит проверить. Такие операции приведения типов служат для введения и вывода объектов из кода, управляемого механизмом ARC. В коде Objective-C, не управляемом механизмом ARC, можно свободно выполнять такие операции, как `(void*)someObject`, поскольку указатели на объекты в данном случае являются обычными указателями C, которым можно отправлять сообщения. В режиме ARC эта операция содержала бы неоднозначность, и поэтому компилятор отказался бы ее выполнять, поскольку он не знал бы, какую именно семантику принадлежности здесь предполагает выражение `void*`.

Инструментальное средство перехода переписывает оператор `(void*)someObject` в виде оператора `(__bridge void*)someObject`, но результат такого преобразования может отличаться от предполагаемого. Более подробно подобные операции приведения типов рассматриваются ниже, в разделе “Взаимодействие с кодом C”.

Автоматически освобождаемые пулы

```
3 id returnObject(void)
4 {
5     return [[NSObject new] autorelease];
6 }
7
8 int main(void)
9 {
10     @autoreleasepool {
11         id object = returnObject();
12         [object retain];
13     }
14     // здесь объект становится недействительным
15     [object release];
16     return 0;
17 }
```

Пример кода из файла autorelease.m

Помимо циклов, наибольшее затруднение при подсчете ссылок возникает в короткие промежутки времени, когда конкретной ссылкой на самом деле не владеет ни один из объектов. В языке C, действительно, непросто решить, какой именно код отвечает за выделение памяти: вызывающий или вызываемый.

В таких функциях, как `sprintf()`, ответственность за выделение памяти возлагается на вызывающий код. Но, к сожалению, вызывающему коду на самом деле ничего не известно, какой именно объем памяти требуется выделить, и поэтому данная функция была дополнена версией `snprintf()`, которая сообщает вызываемому коду объем доступной памяти. Но и это не разрешает рассматриваемое здесь затруднение, в связи с чем данная функция была дополнена еще одной версией `asprintf()`, которая позволяет именно вызываемому коду выделять нужное пространство в памяти.

Если выделением памяти занимается вызываемый код, то какой именно код отвечает за ее освобождение? По-видимому,

вызывающий код. Но поскольку он не принимает в этом никакого участия, то любой код, проверяющий соразмерность вызовов `malloc()` и `free()`, не сможет выявить утечку памяти.

В Objective-C подобное затруднение усугубляется в еще большей степени. Ведь многие методы могут возвращать временные объекты. Так, если возвращается временный объект, его необходимо освободить. Но если возвращается указатель на переменную экземпляра, этого делать не нужно. Можно было бы, конечно, сохранить такой указатель, но тогда пришлось бы помнить о необходимости освободить в отдельности каждый объект, который возвращается из метода, что превращается в довольно утомительное занятие.

В качестве выхода из рассмотренного выше затруднения может служить *автоматически освобождаемый пул*. При отправке объекту сообщения `-autorelease` он добавляется в активный на данный момент экземпляр класса `NSAutoreleasePool`. А при разрушении этого экземпляра каждому добавленному к нему объекту отправляется сообщение `-release`.

По существу, сообщение `-autorelease` является отложенным сообщением `-release`. Оно посылается объекту, когда ссылка на него больше не требуется, но кое-что еще может понадобиться.

Если используется класс `NSRunLoop`, то автоматически освобождаемый пул будет создаваться в начале каждого шага цикла, а разрушаться — в конце. Это означает, что ни один временный объект не будет разрушен до окончания текущего шага цикла. Если же в коде создается много временных объектов, то, возможно, имеет смысл организовать новый автоматически освобождаемый пул, как показано ниже.

```
id pool = [NSAutoreleasePool new];  
[anObject doSomethingThatCreatesObjects];  
[pool drain];
```

Обратите внимание на то, что в приведенном выше фрагменте кода для разрушения автоматически освобождаемого

пула ему отправляется сообщение `-drain`, а не `-release`. Дело в том, что в режиме “сборки мусора” сообщения `-release` будут игнорироваться динамической библиотекой Objective-C. А сообщение `-drain` в этом режиме дает “сборщику мусора” соответствующее указание, но не разрушает пул.

В Mac OS X 10.7 компания Apple сделала автоматически освобождаемые пулы частью языка. Программы, явно ссылающиеся на класс `NSAutoreleasePool`, считаются недопустимыми в режиме ARC, и компилятор откажется их компилировать. В качестве альтернативы можно воспользоваться конструкцией `@autoreleasepool`. Она определяет область памяти, в которой автоматически освобождаемый пул будет действителен. В режиме без применения механизма ARC с помощью этой конструкции будет вставлен точно такой же код, как и в приведенном выше фрагменте. А в режиме с применением механизма ARC эта конструкция позволит вставить вызовы функций `objc_autoreleasePoolPush()` и `objc_autoreleasePoolPop()`, выполняющие аналогичные действия.

Применение автоматически освобождаемых конструкторов

```
4 + (id)object
5 {
6     return [[[self alloc] init] autorelease];
7 }
```

Пример кода из файла `namedConstructor.m`

Как упоминалось выше, для объектов, созданных с помощью сообщения `+alloc`, подсчет сохраняемых ссылок равен единице. В самом деле, все объекты создаются с исходным числом ссылок, равным единице, но объекты, создаваемые с помощью именованного конструктора, например `+stringwithFormat:` или `+array`, являются также автоматически освобождаемыми.

Создавая объект с помощью одного из этих механизмов, ему необходимо отправить сообщение `-retain`, если требуется его сохранить, а иначе он будет убран впоследствии “борщиком мусора” при разрушении автоматически освобождаемого пула.

Это соглашение важно соблюдать в собственных классах. Ведь если кто-нибудь другой создаст экземпляр одного из ваших классов с помощью именованного конструктора, он вряд ли посчитает себя обязанным собственноручно освобождать его. Типичный именованный конструктор должен выглядеть подобно конструктору, код которого приведен в начале этого раздела.

Обратите внимание на то, что объект `self` в таком конструкторе будет классом, поскольку в нем используется метод класса. Отправляя сообщение `+alloc` объекту `self`, а не на имя самого класса, этот метод автоматически может работать с подклассами.

При использовании механизма ARC эти соглашения формализованы в *семействах методов*. Методы, имена которых начинаются на `"alloc"`, `"new"`, `"copy"` или `"mutableCopy"`, возвращают *владеющую ссылку*, т.е. ссылку, которая должна быть освобождена, если ее не нужно сохранять. Другие методы возвращают *не владеющую ссылку*, которая автоматически освобождается или же сохраняется в другом месте с гарантией, что она не будет освобождена.

Автоматическое освобождение объектов в методах доступа

```
34 - (NSString*)stringValue
35 {
36     return [[string retain] autorelease];
37 }
```

Пример кода из файла `ivar.m`

Еще одно распространенное затруднение, возникающее при подсчете ссылок и связанное с его реализацией в среде Foundation, состоит в том, что объекты, на которые делаются ссылки только в стеке, обычно не сохраняются. Допустим, имеется следующий код:

```
NSString *oldString = [anObject stringValue];  
[anObject setStringValue: newString];
```

Если метод `-setStringValue:` реализован так, как предполагалось ранее, этот код приведет к аварийному завершению, поскольку объект, на который указывает ссылка `oldString`, будет удален после установки нового строкового значения. В этом, собственно, и состоит рассматриваемое здесь затруднение. Его можно разрешить двумя способами, причем оба подразумевают применение автоматически освобождаемых пулов. Первый из них состоит в автоматическом освобождении старого значения при установке нового, а второй — в определении метода `-stringValue`, приведенного в примере кода в начале этого раздела.

Но в любом случае гарантируется, что символьная строка не будет случайно разрушена в результате каких-нибудь действий, выполняемых объектом. Еще один распространенный способ состоит в замене сообщения `-copy` сообщением `-retain`. Это полезно делать в том случае, если переменная экземпляра может быть изменяемой. Если она неизменяемая, то сообщение `-copy` будет равнозначно сообщению `-retain`. А если она все же изменяемая, то вызывающий код получит объект, который не будет изменяться в результате отправки других сообщений данному объекту.

Поддержка автоматической “сборки мусора”

```
0 $ gcc -c -framework Cocoa -fobjc-gc-only  
   collected.m  
1 $ gcc -c -framework Cocoa -fobjc-gc collected.m
```


В версии Mac OS X 10.5 компания Apple внедрила в Objective-C поддержку автоматической “сборки мусора”. Это нововведение призвано облегчить жизнь программирующим на Objective-C, но зачастую оно приводит к снижению производительности. В “сборщике мусора”, предложенном компанией Apple, используется немалый объем оперативной памяти для отслеживания активных ссылок, а следовательно, этот механизм удаления объектов из памяти не доступен для применения в мобильных телефонах iPhone. Кроме того, автоматическая “сборка мусора” не поддерживается в прежних версиях Mac OS X, а в GNUstep имеет лишь ограниченную поддержку, и поэтому при написании переносимого кода ее следует избегать.

Если скомпилировать код в режиме “сборки мусора”, все сообщения `-retain`, `-release` и `-autorelease` будут проигнорированы. Для каждой операции присваивания в области памяти, отводимой под “кучу”, компилятор автоматически вставит обращения к соответствующим функциям во время выполнения кода.

Для использования “сборщика мусора” код должен быть скомпилирован со средствами поддержки данного механизма. Благодаря этому обеспечивается вставка обращений к набору функций, которые в исполняющей среде Mac OS X объявляются в заголовочном файле `objc-auto.h` для любой операции присваивания в области памяти, отводимой под “кучу”.

Эти функции гарантируют осведомленность “сборщика мусора” о записи в память. Без этого нельзя обойтись, поскольку “сборщик мусора”, параллельно выполняемый в фоновом потоке, будет удалять объекты, если не обнаружит больше никаких ссылок на них. “Сборщик мусора” должен быть уведомлен о модифицированных указателях, в противном случае он может случайно удалить объект, на который была только что создана ссылка.

Скомпилировать код в режиме “сборки мусора” можно двумя способами. Если код компилируется с параметром командной строки `-fobjc-gc-only`, в нем будет поддерживаться только “сборка мусора”. Если же компиляция кода выполняется с параметром командной строки `-fobjc-gc`, в нем будет поддерживаться как подсчет ссылок, так и автоматическая “сборка мусора”, что удобно для компиляции интегрированной среды. Не следует, однако, забывать о добавлении вызовов `-retain` и `-release` в нужных местах такой среды, хотя ее пользователи смогут затем применять ее как со “сборкой мусора”, так и без нее.

```
110 OBJC_EXPORT BOOL objc_atomicCompareAndSwapGlobal(  
    id predicate, id replacement, volatile id *  
    objectLocation)  
111     __OSX_AVAILABLE_STARTING(__MAC_10_6,  
    __IPHONE_NA) OBJC_ARC_UNAVAILABLE;  
112 OBJC_EXPORT BOOL  
    objc_atomicCompareAndSwapGlobalBarrier(id  
    predicate, id replacement, volatile id *  
    objectLocation)  
113     __OSX_AVAILABLE_STARTING(__MAC_10_6,  
    __IPHONE_NA) OBJC_ARC_UNAVAILABLE;  
114 // атомарное обновление переменной экземпляра  
115 OBJC_EXPORT BOOL  
    objc_atomicCompareAndSwapInstanceVariable(id  
    predicate, id replacement, volatile id *  
    objectLocation)  
116     __OSX_AVAILABLE_STARTING(__MAC_10_6,  
    __IPHONE_NA) OBJC_ARC_UNAVAILABLE;  
117 OBJC_EXPORT BOOL  
    objc_atomicCompareAndSwapInstanceVariableBarrier  
    (id predicate, id replacement, volatile id *  
    objectLocation)
```

Пример кода из файла `objc-auto.h`

Взаимодействие с кодом C

В режиме “сборки мусора” можно просмотреть не всю память. Вся память, выделяемая функцией `malloc()`, для “сборщика мусора” недоступна. Если указатель на объект передается в качестве параметра типа `void*` функции C, которая затем сохраняет его в области памяти, выделяемой функцией `malloc()`, эта память становится недоступной для “сборщика мусора” и может быть освобождена даже в том случае, если на нее есть ссылки.

В режиме ARC компилятор будет автоматически обрабатывать указатели, хранящиеся в стеке или в переменных экземпляра, но он не станет отслеживать указатели в структурах или любых других конструкциях, объявленных явным образом как `__unsafe_unretained`.

Как правило, до сохранения объекта в “куче” ему посылается сообщение `-retain`, но в режиме “сборки мусора” это ничего не дает, а в режиме ARC вообще запрещено. Вместо этого следует воспользоваться функцией `CFRetain()`, в которой количество ссылок на объект инкрементируется независимо от того, активен ли “сборщик мусора”. Этот сборщик освобождает объекты из памяти только в том случае, если количество ссылок на них равно нулю, но он не способен находить ссылки на них в отслеживаемой области памяти. Завершив операции со ссылкой, которая находится вне области действия “сборщика мусора”, следует вызвать функцию `CFRelease()`.

Механизм ARC предоставляет более развитую модель памяти для подобного рода операций. Явные операции приведения типов указателей на объекты к другим типам указателей больше не допускаются. Их необходимо заменить операциями *стыковочного приведения типов*. Рассмотрим следующую строку кода в режиме без применения механизма ARC:

```
void *aPointer = (void*)someObject;
```

При выполнении этой строки кода в режиме ARC из отслеживаемого указателя был бы сформирован неотслеживаемый указатель, что вряд ли было бы желаемой целью данной операции без особых на то причин. Из этого положения имеются три основных выхода. Первый из них чаще всего применяется для стековых переменных, т.е. переменных, указывающих на те объекты, на которые точно имеются ссылки в других местах кода.

```
void *aPointer = (__bridge void*)someObject;
```

В этой строке кода выполняется операция приведения типов без передачи прав владения. Если все остальные ссылки на объект `someObject` будут удалены, то `aPointer` станет висячим указателем. Если же указатель типа `void*` обозначает какое-то другое место в “куче”, где должна храниться владеющая ссылка на объект, в таком случае следует выполнить стыковочное приведение типов с сохранением.

```
void *aPointer = (__bridge_retained void*)
    someObject;
```

При выполнении этой строки кода одиночная владеющая ссылка будет выведена из-под контроля со стороны механизма ARC. Это приблизительно равнозначно opravке сообщения `-retain` объекту `someObject` перед его сохранением. Если же написать просто `(__bridge_retained void*) someObject` без оператора присваивания, то компилятор получит указание сохранить этот объект, чего рекомендуется избегать в практике программирования на Objective-C. Выполняя обратное приведение к типу указателя на объект, следует воспользоваться противоположной операцией, как показано ниже.

```
id anotherObjectPointer = (__bridge_transfer
    id)aPointer;
aPointer = NULL;
```

В этом случае владеющая ссылка передается под контроль механизму ARC. И теперь он будет отвечать за осво-

бождение объекта, о чем важно помнить при обнулении указателя `C`. Если же владение указателем не предполагается, следует воспользоваться простым стыковочным приведением типов `__bridge`.

Представление о разрушении объектов

```
3 @interface Example : NSObject
4 {
5     void *cPointer;
6     id objectPointer;
7 }
8 @end
9 @implementation Example
10 -(void)finalize
11 {
12     if (NULL != cPointer) { free(cPointer); }
13     [super finalize];
14 }
15 (void)dealloc
16 {
17     if (NULL != cPointer) { free(cPointer); }
18     #if !__has_feature(objc_arc)
19         [objectPointer release];
20         [super dealloc];
21     #endif
22 }
23 @end
```

Пример кода из файла `dealloc.m`

Имеются три метода, которые вызываются в процессе разрушения объекта (в зависимости от конкретного режима работы). Один из них, метод `-cxx_destruct`, всегда вызывается динамической библиотекой Objective-C для освобождения памяти, занимаемой полями, за очистку которых отвечает компилятор. В этом методе используются

объекты C++ в режиме Objective-C++ и указатели на объекты Objective-C в режиме ARC.

Двумя другими являются методы `-finalize` и `-dealloc`. В режиме “сборки мусора” для освобождения ссылок на объекты Objective-C делать ничего не нужно, но по-прежнему необходимо освобождать любые ресурсы, которые не находятся под управлением “сборщика мусора”. К числу подобных операций относятся закрытие дескрипторов файлов, освобождение памяти, выделяемой функцией `malloc()`, и пр. Если класс содержит переменные экземпляра, которым требуется подобного рода освобождение ресурсов вручную, в таком случае следует объявить метод `-finalize`.

На заметку

“Сборщик мусора” обычно вызывает методы `-finalize` в специальном потоке очистки. Это означает, что методы `-finalize` должны быть потокобезопасными. Если же они ссылаются на глобальные ресурсы, то следует принять меры против возникновения состояния гонок.

Если вы не пользуетесь услугами “сборщика мусора”, вам следует производить очистку в методе `-dealloc`. Содержимое этого метода зависит от того, пользуетесь ли вы механизмом ARC. По традиции методы `-dealloc` содержат отправки сообщений `-release` каждой переменной экземпляра, объявленной в классе. Если же применяется механизм ARC, они не обязательны, поскольку этот механизм освободит любые владеющие ссылки на объекты в методе `-.cxx_destruct`, а вам останется лишь очистить необъектные переменные экземпляра в методе `-dealloc`.

Как в режиме “сборки мусора”, так и в ручном режиме сохранения/освобождения ссылок сообщение `retain/release` следует переслать суперклассу, вызвав метод `[super dealloc]` или `[super finalize]`. В режиме ARC вызов `-dealloc` в явном виде не разрешен. Вместо него меха-

низм ARC вставит вызов `[super dealloc]` в конце метода `-dealloc` в любом некорневом классе.

Следует иметь в виду, что пример кода, приведенный в начале данного раздела, несколько усложнен. В реальном коде вряд ли понадобится поддерживать оба режима: ARC и ручного сохранения/освобождения ссылок. Но оба режима могут взаимодействовать в одной и той же программе. Единственной причиной для реализации режима без применения механизма ARC может быть желание поддерживать устаревшие компиляторы.

Применение слабых ссылок

```
4  __weak id weak;
5
6  int main(void)
7  {
8      id obj = [NSObject new];
9      weak = obj;
10     obj = nil;
11     objc_collect(OBJC_FULL_COLLECTION);
12     fprintf(stderr, "weak reference: %p\n", weak);
13     return 0;
14 }
```

Пример кода из файла `weak.m`

Одной из самых замечательных особенностей реализации компанией Apple процесса “сборки мусора” является *обнуление слабых ссылок*. Указатели, не сохраняющиеся принудительно, в документации на Objective-C, составленной еще до внедрения “сборки мусора”, часто называются *нестрогими*, а соответствующие ссылки — *слабыми*. Они представляют собой ссылки, которым разрешается продолжать существование по завершении срока действия объекта. К сожалению, не существует автоматического способа,

уведомляющего о том, являются ли они все еще действительными.

Слабые ссылки оказались настолько полезными для “сборки мусора”, что они до сих пор поддерживаются в механизме ARC, хотя и с помощью немного другой семантики. А в реализации механизма ARC, предоставляемой для обратной совместимости, слабые ссылки не поддерживаются, и поэтому на их ограниченное применение в механизме ARC можно рассчитывать только в его реализации на платформах компании Apple, начиная с версии Mac OS 10.7 или iOS 5.

На заметку

Слабые ссылки в среде с подсчетом ссылок, например при использовании делегатов, обычно применяются для удаления циклов сохранения. Этого не нужно делать в среде с отслеживанием ссылок, поэтому для указателей на делегаты и везде, где может образоваться цикл сохранения, допускается пользоваться строгими ссылками.

Объявив указатель на объект как `__weak`, вы получаете обнуляемую слабую ссылку. Она не учитывается “сборщиком мусора”, когда нужно определить, является ли объект по-прежнему активным, а счетчик ссылок на этот объект не инкрементируется при выполнении присваивания в режиме ARC. Если все ссылки на объект являются слабыми, он может быть разрушен. При последующем считывании слабых ссылок будет возвращено значение `nil`.

Слабые ссылки чаще всего используются в связи с уведомлениями. Обычно слабая ссылка на объект и отправляемые ему сообщения сохраняются до тех пор, пока где-то в другом месте кода существует другая ссылка на этот объект, а в дальнейшем можно реализовать его автоматическое освобождение.

В настоящее время в состав среды Сосоа входит несколько коллекций, которые позволяют сохранять слабые

ссылки. В более старых версиях среды Foundation предоставляются такие непрозрачные типы C, как `NSMutableDictionary` и `NSMutableTable`, а также набор функций C для их применения. Эти интерфейсы все еще доступны, но в версии 10.5 разработчики из компании Apple превратили оба указанных типа C в классы.

Тип `NSMutableDictionary` представляет собой обобщенную форму типа `NSDictionary`, который можно использовать для хранения любых типов длиной с указатель в качестве значений или ключей. В режиме “сборки мусора” класс `NSMutableDictionary` можно использовать для хранения результатов преобразования типов в строгие или нестрогие указатели на объекты, и обратно. Это полезно для выполнения таких действий, как передача уведомлений из центра уведомлений типа `NSNotificationCenter`, чтобы объекты можно было собрать в то время, пока они еще зарегистрированы для получения уведомлений, а затем автоматически удалить из этого центра после получения уведомлений.

В примере, приведенном в начале этого раздела, демонстрируется существенное отличие режимов ARC и “сборки мусора”. Если скомпилировать и выполнить этот код в режиме “сборки мусора”, то на экран будет выведен адрес объекта. Дело в том, что, просматривая стек, “сборщик мусора” обнаруживает устаревшие временные данные.

А в режиме ARC упомянутый выше код будет всегда выводить на экран нулевое значение. Механизм ARC, в отличие от “сборки мусора”, полностью детерминирован. В результате присваивания значения `nil` строгому указателю уменьшается на единицу количество ссылок на объект и иницируется процесс освобождения ресурсов. Нестрогий указатель обнуляется до начала процесса освобождения, что гарантирует его нулевое состояние на момент выполнения функции `fprintf()`.

Выделение просмотренной памяти

```
15 id *buffer =  
16     NSAllocateCollectable(  
17         10 * sizeof(id),  
18         NSScannedOption);
```

Пример кода из файла gc.m

Если память распределяется с помощью функции `malloc()`, она становится недоступной для “сборщика мусора”. Это может вызвать известное затруднение, если потребуется, например, воспользоваться контейнером, подобным массиву C, содержащему объекты. Мы уже рассматривали один способ выхода из подобного затруднения. В частности, можно вызвать функцию `CFRetain()` для объекта, который требуется сохранить, а функцию `CFRelease()` — для старого значения, после чего поменять их местами.

Это не идеальный, но вполне работоспособный способ. Другой способ состоит в выделении области памяти из резерва “сборщика мусора”. Для этой цели служит функция `NSAllocateCollectable()`, аналогичная функции `malloc()`, но у нее имеются два важных отличия. Первое из них состоит в том, что область памяти, которую она возвращает, выделяется в результате “сборки мусора”. А кроме того, не существует парной с ней функции `NSFreeCollectable()`. Когда исчезает последний указатель на буфер, выделяемая под него память освобождается “сборщиком мусора”.

На заметку

В “сборщике мусора” от компании Apple не поддерживаются внутренние ссылки, поэтому следует непременно сохранить указатель на начало выделенной области памяти. Указатели на середину буфера не защитят его от освобождения.

Другое отличие состоит в том, что второй параметр функции `NSAllocateCollectable()` определяет нужный вид памяти. Так, если буфер предполагается использовать для хранения типов данных C, в качестве второго параметра данной функции можно просто передать нулевое значение. Если же в качестве второго ее параметра передать объект типа `NSScannedOption`, возвращаемый буфер будет рассмотрен как потенциальное место для расположения указателей на объекты, а также указателей на другие области памяти, возвращаемые функцией `NSAllocateCollectable()`.

4

Шаблоны, распространенные в Objective-C

В каждом языке программирования поощряется применение различных шаблонов проектирования. Ясное представление о распространенных шаблонах проектирования, применяемых в среде Foundation, позволяет лучше понять особенности этой среды в целом и упрощает написание программ на Objective-C.

В этой главе дается краткий обзор некоторых наиболее употребительных шаблонов в Objective-C. Для более подробного изучения этих шаблонов отсылаем вас к книге *Cocoa Design Patterns* Эрика М. Бака (Erik M. Buck) и Дональда А. Яктмана (Donald A. Yacktman), издательство Addison-Wesley, 2009 г.

Поддержка шаблона создания объектов в два этапа

```
6 NSMutableString *buffer =  
7   [[NSMutableString alloc] init];  
8 NSMutableArray *array =  
9   [[NSMutableArray alloc]  
10    initWithObject: buffer];
```

Пример кода из файла allocInit.m

К числу тех шаблонов, которые вам, вероятно, уже встречались, относится *шаблон создания объектов в два этапа*. В традиционной версии Objective-C, разработанной до спецификации компании NeXT, объект создавался путем отправки сообщения `+new` соответствующему классу. При реализации нового класса метод, связанный с сообщением `+new`, сначала переопределялся для вызова реализации суперкласса, а затем класс инициализировался.

В спецификации компании NeXT это положение изменилось. Обязанности по созданию и инициализации объекта были распределены между двумя разными методами. Метод `+new` по-прежнему действует, но теперь посылает сообщение `+alloc` классу, а сообщение `-init` — возвращаемому объекту.

Сам же метод `+alloc` по умолчанию вызывает метод `+allocWithZone:`, принимая объект типа `NSZone` в качестве аргумента. Это объект непрозрачного типа, инкапсулирующий зону выделяемой памяти. Такой подход интенсивно применялся в системах NeXT для эффективного управления очень малым объемом оперативной памяти, которым обладали эти системы. Но для других систем, в том числе Mac OS X, он малоприспособлен и поэтому недостаточно поддерживается во всех классах, превратившись в нечто рудиментарное в большинстве современных программ на Objective-C.

Создавая новый класс, вы, по существу, переопределяете метод `-init` или предоставляете новый специально выделяемый для этой цели инициализатор.

На заметку

Некоторые отголоски старого подхода к созданию объектов можно обнаружить в GCC. Когда метод `+new` применялся для создания объектов, было принято получать сначала новый экземпляр, а затем присваивать его псевдопеременной `self` в методе данного класса. В GCC имеется код, поддерживающий такой подход и разрешающий доступ к переменной экземпляра из методов класса. Впрочем, такая функциональная возможность уже давно отменена, и при всякой попытке воспользоваться ему выдается соответствующее предупреждение.

Метод `+alloc` приходится переопределять очень редко. Иногда это делается для реализации пулов по отдельным классам или одиночных классов в специальных целях, но это совершенно не характерно для программирования на Objective-C. Если реализуется одиночный класс, то метод `+alloc` может быть переопределен для возврата экземпляра этого класса, при условии, что он существует. Более подробно данный вопрос рассматривается далее в главе.

Копирование объектов

```
3 @interface Pair : NSObject <NSCopying>
4 @property (nonatomic, retain) id first, second;
5 @end
6 @implementation Pair
7 @synthesize first, second;
8 - (id)copyWithZone: (NSZone*)aZone
9 {
10     Pair *new = [self-> isa allocWithZone: aZone];
11     new.first = first;
12     new.second = second;
13     return new;
14 }
15 @end
```

Пример кода из файла `copy.m`

В состав среды Foundation входит функция `NSCopyObject()`. Вполне логично предположить, что эта функция копирует объект, но это не совсем так. Она создает сначала новый объект того же самого класса, что и исходный, а затем использует нечто вроде функции `memcpy()` для копирования переменных экземпляра в памяти.

Такой способ вполне пригоден для объектов, в переменных экземпляров которых хранятся только данные примитивных типов C, но не подходит для остальных объектов. Для копирования указателей на объекты им приходится

отправлять сообщение `-retain`. Над указателями `C`, возможно, потребуется произвести ряд других операций, особенно если они относятся к непрозрачным типам, возвращаемым из какой-нибудь другой библиотеки.

Без достаточных сведений о взаимосвязи между объектами функция копирования не сможет автоматически создать подлинную копию объекта. А некоторые объекты вообще не должны поддерживать копирование. Какой, например, должна быть семантика копирования объекта, инкапсулирующего сокет? Будут ли получены в результате копирования два объекта, инкапсулирующие один и тот же сокет?

О том, как копировать объект, на самом деле известно только самому объекту. Так, если объект реализует протокол `NSCopying`, ему можно отправить сообщение `-copyWithZone:`, чтобы получить его копию. Вызываемый при этом метод принимает объект типа `NSZone` в качестве аргумента.

В большинстве современных программ на Objective-C объект типа `NSZone` вообще игнорируется. Ведь в классе `NSObject` предоставляется метод `-copy`, который просто вызывает метод `-copyWithZone:` с выделяемой по умолчанию зоной памяти в качестве аргумента. Но, несмотря на такую возможность вызвать метод `-copy`, следует также реализовать метод `-copyWithZone:`.

В начале этого метода, как правило, отправляется сообщение `+allocWithZone:` указателю `isa` получателя, по которому создается новый экземпляр. А затем в этом методе может быть вызван отдельный инициализатор или же заполнены переменные экземпляра в нем самом.

На заметку

При наследовании от суперкласса, реализующего протокол `NSCopying`, сообщение `-copyWithZone:` следует отправить псевдопеременной `super` в конкретной реализации метода `-copyWithZone:` и только затем скопировать переменные экземпляра, добавляемые подклассом.

Объект, возвращаемый методом `-copy` (или `-copyWithZone:`), следует интерпретировать таким же образом, как и объект, возвращаемый методом `+alloc` (или `+allocWithZone:`). Первоначальный подсчет сохраняемых ссылок на этот объект будет равен нулю, поэтому не забудьте отправить ему сообщение `-release` или `-autorelease` в определенный момент, чтобы исключить утечки памяти.

Архивирование объектов

```
55 void roundTripWithArchiver(id object, Class coder,
    Class decoder)
56 {
57     NSData *data = [coder
        archivedDataWithRootObject: object];
58     id copy = [decoder unarchiveObjectWithData:
        data];
59     NSCAssert(data, @"Archiving failed");
60     NSCAssert(copy, @"Unarchiving failed");
61     NSCAssert([object integer] == [copy integer],
62         @"Integer wasn't copied correctly");
63     NSCAssert([object string] isEqual:
        [copy string]),
64         @"String wasn't copied correctly");
65 }
```

Пример кода из файла `archive.m`

Нередко требуется сохранить отдельные части программы на более продолжительный период времени, чем одиночный вызов. Некоторые типы данных только объектов можно сохранить в *списках свойств*, как будет показано в главе 7. Но как быть с сохранением произвольных объектов?

Данные можно сохранить в архиве специального (или открытого) формата, а затем восстановить из него объекты. Но определять форматы файлов только для того, чтобы сохранить несколько объектов, не совсем рационально, поскольку это требует дополнительных затрат труда.

Правда, выход из этого затруднительного положения можно найти в протоколе `NSCoding` из среды `Foundation`. В этом протоколе определены два следующих метода: `-initWithCoder:` и `-encodeWithCoder:`. Большинство стандартных классов реализуют этот протокол, но его нетрудно реализовать самостоятельно.

Объект, передаваемый этим методам в качестве аргумента, выполняет функции кодера. Он относится к классу, инкапсулирующему некоторое представление данных. В старых системах эти функции обычно выполнял класс `NSArchiver`, а в новых — класс `NSKeyedArchiver`. В старом классе `NSArchiver` предоставляется интерфейс, позволяющий лишь записывать поток типизированных значений. Но в архивах с доступом по ключу поддерживается сохранение не только в виде словаря. Они позволяют сначала сохранить пары “ключ–значение”, а затем восстановить их в другом порядке.

Поддержка архивирования объектов разделяется на две части. Объект должен быть в состоянии архивироваться и восстанавливаться из архива. Так, в методе `-encodeWithCoder:` необходимо сохранить все состояние объекта в кодере.

Но это совсем не означает, что должны быть сохранены все переменные экземпляра объекта. Ведь у некоторых объектов имеются переменные экземпляра, используемые только для кэширования данных, которые могут быть затем восстановлены из какого-нибудь другого места. Поэтому эти переменные зачастую лучше воссоздавать после восстановления объекта, чем сохранять вместе с ним.

```
34 (void)encodeWithCoder: (NSCoder*)aCoder
35 {
36     if ([aCoder allowsKeyedCoding])
37     {
38         [aCoder encodeInt: integer
39             forKey: @"integer"];
40         [aCoder encodeObject: string
```

```
41         forKey: @"string"];
42     }
43     else
44     {
45         [aCoder encodeValueOfObjCType: @encode(int)
46                                     at: &integer];
47         [aCoder encodeObject: string];
48     }
49
50 }
```

Пример кода из файла `archive.m`

Имеет смысл поддерживать архиваторы с ключом и без ключа. Зачастую архиваторы без ключа применяются для временного кодирования (например, при передаче данных по сети), тогда как архиваторы с ключом — для долговременного хранения данных.

У архиваторов с ключом имеются более удобные методы для архивирования примитивных типов данных C. Им можно отправлять такие сообщения, как, например, `-encodeInt:forKey:`. А у старых архиваторов для сохранения примитивных типов данных C имеется лишь один метод, принимающий в качестве параметров кодировку типа данных Objective-C и указатель на переменную.

Соответствующий метод для загрузки архивов может оказаться несколько более сложным. Архиватор сохраняет *версию класса* для каждого архивируемого объекта. С этой целью следует отправить сообщение `+setVersion:` псевдопеременной `self` в методе `-initialize`, когда изменяется расположение переменной экземпляра объекта.

```
12 -(id)initwithCoder: (NSCoder*)aCoder
13 {
14     if ([aCoder versionForClassName: [self
15                                     className]] != 0)
16     {
17         [self release];
18     }
19 }
```

```

17     return nil;
18 }
19 if ([aCoder allowsKeyedCoding])
20 {
21     integer =
22     [aCoder decodeIntForKey: @"integer"];
23     string =
24     [aCoder decodeObjectForKey: @"string"];
25 }
26 else
27 {
28     [aCoder decodeValueOfObjCType: @encode(int)
29     at: &integer];
30     string = [aCoder decodeObject];
31 }
32 string = [string retain];
33 return self;
34 }

```

Пример кода из файла `archive.m`

В методе `-initWithCoder:`, возможно, потребуется загрузить архивы из старых версий программы, поэтому проверка версии и соответствующая обработка архива должны быть выполнены в нем иначе, если он относится к старой версии.

Не следует также забывать, что реализации обоих упомянутых выше методов в суперклассе следует вызывать в том и только в том случае, если суперкласс соответствует протоколу `NSCoding`. Если же он не соответствует этому протоколу, а вам требуется сохранить какое-нибудь состояние, вы должны взять на себя ответственность за архивирование этого состояния, а также переменных экземпляра используемого вами подкласса.

Создание выделенных инициализаторов

```

16 -(id)initWithSelectorName: (NSString*)aSel
17     arguments: (NSArray*)args
18 {

```

```
19  if (nil == (self = [super init]))
20  { return nil; }
21  selector = [aSel copy];
22  arguments = [args mutableCopy];
23  return self;
24  }
25  -(id)initWithSelectorName:(NSString*)aSel
26  {
27      return
28          [self initWithSelectorName: aSel
29              arguments: nil];
30  }
31  -(id)init
32  {
33      return [self initWithSelectorName: nil];
34  }
```

Пример кода из файла `designatedInit.m`

Большинство рассмотренных ранее классов инициализировались с помощью метода `-init`. Этот метод представляет собой *выделенный инициализатор* в классе `NSObject`. Он, как правило, переопределяется при создании нового подкласса, производного от класса `NSObject`.

При создании нового подкласса, производного от класса `NSObject`, все переменные его экземпляра обнуляются. Поэтому в своей версии инициализатора можете затем установить в этих переменных какие-нибудь другие значения, а также запросить любые другие ресурсы, требующиеся для объекта.

На заметку

Метод `-init` из класса `NSObject` ничего особенного не делает, возвращая лишь псевдопеременную `self`. Это означает, что в подклассах, прямых потомках класса `NSObject`, следует всячески избегать искушения отправлять сообщение `-init` псевдопеременной `self`, поскольку данный метод может быть заменен в категории класса `NSObject` методом, делающим нечто конкретное.

Не во всех классах метод `-init` сохраняется в качестве выделенного для них инициализатора. По соглашению инициализатор, принимающий аргументы, считается выделенным. А все остальные методы инициализации будут вызывать именно этот инициализатор.

Благодаря этому упрощается создание новых подклассов. В большинстве классов предоставляется целый ряд служебных методов инициализации. Если бы все эти методы пришлось переопределять в каждом подклассе, то в конечном итоге появилось бы немало кода, не имеющего особой ценности.

Если же в суперклассе применяется шаблон выделенного инициализатора, то в нем достаточно переопределить лишь один инициализатор. Пример кода, приведенный в начале этого раздела, взят из класса `LKMessageSend`, входящего в состав среды `LanguageKit`. Он демонстрирует построение узла абстрактного синтаксического дерева, представляющего операцию отправки сообщения. Выделенным инициализатором суперкласса является метод `-init`, поэтому он вызывается в данном классе из его собственного, выделенного для него инициализатора. В этом классе метод `-init` сначала переопределяется, а затем из его переопределенного варианта вызывается инициализатор, выделенный для данного класса.

Если из этого класса получить новый подкласс, то в нем достаточно будет переопределить метод `-initWithSelectorName:arguments:`, да и то лишь в том случае, если вводятся переменные экземпляра, которые требуется инициализировать. Код, отправляющий сообщение `-init` данному подклассу, получит обратно инициализированную переменную экземпляра.

Реализация шаблона-одиночки

```
5 @implementation Singleton
6 static Singleton *sharedInstance;
```

```
7 + (void)initialize
8 {
9     if ([Singleton class] == self)
10    {
11        sharedInstance = [self new];
12    }
13 }
14 + (Singleton*)sharedInstance
15 {
16     return sharedInstance;
17 }
18 + (id)allocWithZone: (NSZone*)aZone
19 {
20     if (sharedInstance &&
21         [Singleton class] == self)
22     {
23         [NSException raise: NSGenericException
24                      format: @"May not create more than
25                             one instance of singleton."];
26     }
27     return [super allocWithZone: aZone];
28 }
29 @end
```

Пример кода из файла singleton.m

Одиночки представляют собой объектно-ориентированный вариант глобальных переменных. У класса-одиночки, или, иначе, одиночного класса, имеется единственный экземпляр. Таких классов немало в Objective-C, но все они следуют одному и тому же общему шаблону.

В примере кода, приведенном в начале этого раздела, демонстрируется скелетное представление класса, допускающего создание единственного его экземпляра. В нем соблюдается соглашение о возврате одиночного экземпляра в ответ на сообщение `+sharedInstance`. Во многих одиночных классах имя класса подставляется в название этого сообщения вместо `Instance`.

Так, если требуется получить объект общего приложения в среде AppKit, классу `NSApplication` посылается сообщение `+sharedApplication`. Этот класс представляет собой особый случай одиночки, поскольку свой общий экземпляр он хранит в глобальной переменной `NSApp`. Если приложение уже запущено на выполнение, этой переменной можно воспользоваться непосредственно, не прибегая к отправке сообщения с целью получить объект приложения.

Наиболее распространенная ошибка в обращении с одиночками заключается во внесении состояния гонок в процесс их создания. В рассматриваемом здесь примере одиночный класс сохраняет свой единственный экземпляр в закрытой переменной. Невольно возникает искушение создать его в методе `+sharedInstance`. Это вполне допустимо для однопоточных приложений, но как быть с многопоточными приложениями?

Что, например, произойдет, если отправить сообщение `+sharedInstance` одновременно из двух потоков? В обоих потоках может быть проверено наличие переменной, и если обнаружится, что она не создана, то будет предпринята попытка создать ее. Во избежание подобной ситуации можно было бы воспользоваться блокировкой.

Но более простое решение состоит в создании одиночного экземпляра в методе `+initialize`. Этот метод вызывается из динамической библиотеки, когда упомянутое выше сообщение отправляется одиночному классу в первый раз. Именно динамическая библиотека отвечает за полное завершение данного метода перед тем, как одиночному классу будут доставлены сообщения из других потоков.

¹ *Правильному выполнению этой последовательности действий препятствовала давно укоренившаяся программная ошибка в динамической библиотеке GCC. Но в динамической библиотеке GNUStep она была исправлена.*

На заметку

В документации компании Apple рекомендуется использовать директиву `@synchronized(self)` в методе `+sharedInstance` при обращении с одиночками. Но такой подход оказывается значительно более медленным и требует большего объема кода, чем способ, предлагаемый в этом разделе, и поэтому данная рекомендация вряд ли может принести большую пользу.

Переопределять метод `+allocWithZone:` в одиночке не нужно, но если это сделать, то он обеспечит некоторые дополнительные средства проверки во время выполнения. В частности, код из рассматриваемого здесь примера сгенерирует исключение, если попытаться выделить память для нового экземпляра одиночного класса после создания единственного его экземпляра.

Обратите внимание на проверки в обоих методах, `+initialize` и `+allocWithZone:`. Они требуются для предотвращения подклассификации. Но вам совсем не обязательно генерировать исключение в своем коде, если будет предпринята попытка создать подкласс, производный от создаваемого вами одиночного класса. Ведь получить подкласс от одиночки не так-то просто, а следовательно, проверку подобного действия можно опустить, если ее поддержка нецелесообразна.

Делегирование

Множественное наследование в Objective-C не поддерживается. Впрочем, это не такое уж и сильное ограничение, поскольку воспользоваться множественным наследованием как следует не так-то просто. Но, несмотря на очевидные трудности, обращение к множественному наследованию оказывается порой вполне естественным решением.

Принцип, положенный в основу *шаблона делегирования*, аналогичен принципам реализации множественного наследования и прототипов: в одном объекте разрешается

определить некоторое подмножество видов поведения другого объекта.

В среде Foundation делегирование применяется мало, но очень широко распространено в средах AppKit и UIKit. В каждом объекте пользовательского интерфейса делегирование служит для определения того, что происходит в ответ на события, наступающие в пользовательском интерфейсе.

С делегированием связано несколько шаблонов, подчиняющихся следующему общему правилу: композиции объектов следует отдавать предпочтение над наследованием. Это очень важно для обеспечения слабой связности, поскольку на много упрощается повторное использование кода.

Так, если имеется класс C++ с тремя подклассами, обеспечивающими некоторые виды поведения, то видоизменить его на много труднее, чем класс Objective-C, делегирующий виды своего поведения и функции трем другим классам.

Предоставление фасадов

```
11 @interface Control: View
12 {
13     id cell;
14 }
15 @end
16 @implementation Control
17 -(id)selectedCell
18 {
19     return cell;
20 }
21 -(BOOL)isEnabled
22 {
23     return [[self selectedCell] isEnabled];
24 }
25 -(void)setEnabled: (BOOL)flag
26 {
27     [[self selectedCell] setEnabled: flag];
28     [self setNeedsDisplay: YES];
```

```
29 }  
30 @end
```

Пример кода из файла facade.m

К числу наиболее распространенных примеров применения делегирования относится *шаблон фасада*. Он включает в открытом интерфейсе один или несколько полузакрытых объектов. Этот шаблон широко применяется в Objective-C с целью обеспечить нечто похожее на множественное наследование, где в одном объекте сочетается поведение нескольких различных объектов, делегирующих друг другу часть своих функций.

Характерным тому примером служит иерархия классов `NSControl` в `AppKit`. Классы в этой иерархии наследуют поведение от класса `NSView` и делегируют свое поведение подклассу `NSCell`. А в подклассе `NSCell` предоставляются такие средства, как графический контекст, обработка событий и взаимодействие с иерархией представлений. На уровне отдельной ячейки данного класса предоставляются такие средства, как рисование. В коде примера, приведенного в начале этого раздела, демонстрируется упрощенный (в значительной степени) вариант класса `NSControl`.

Когда вы щелкаете на кнопке в Mac OS X, на самом деле выбираете не кнопку, а экземпляр подкласса `NSButton`, производного от класса `NSControl` и использующего для своей реализации класс `NSButtonCell`. В целях программирования можете применять подобные элементы управления пользовательского интерфейса практически попеременно, а также повторно использовать ячейки, которые они содержат. Обнаружив, например, кнопку в табличном или схематическом представлении, имейте в виду, что она нарисована средствами класса `NSButtonCell`, а не класса `NSButton`.

В главе 19 мы рассмотрим применение механизмов пересылки в Objective-C для быстрой и простой реализации подобного рода фасада. Их можно также (и это распростра-

нено на практике) реализовать, вызвав методы непосредственно в объектах, заключенных в оболочку.

Такой подход считается намного более гибким, чем множественное наследование, поскольку он позволяет использовать один и тот же класс с самыми разными делегатами. В языке C++ для достижения аналогичной цели пришлось бы прибегнуть к шаблонным классам. У механизмов пересылки имеются разные преимущества в обоих языках. В Objective-C генерируется намного более компактный код, что, в свою очередь, способствует более рациональному использованию кеша команд. А в C++ некоторые категории оптимизации (главным образом, встраивание) упрощаются во время компиляции.

Создание кластеров классов

```
26 static Pair *placeholder;
27 + (void)initialize
28 {
29     if ([Pair class] == self)
30     {
31         placeholder = [self alloc];
32     }
33 }
34 + (id)allocWithZone: (NSZone*)aZone
35 {
36     if ([Pair class] == self)
37     {
38         if (nil == placeholder)
39         {
40             placeholder =
41                 [super allocWithZone: aZone];
42         }
43         return placeholder;
44     }
45     return [super allocWithZone: aZone];
46 }
```

```
47 - (Pair*)initWithFloat: (float)a float: (float)b
48 {
49     return [[FloatPair alloc] initWithFloat: a
50             float: b];
51 }
52 - (Pair*)initWithInt: (int)a int: (int)b
53 {
54     return [[IntPair alloc] initWithInt: a int: b];
55 }
```

Пример кода из файла `classCluster.m`

Кластеры классов весьма распространены в Objective-C. Многие классы из среды Foundation на самом деле являются кластерами классов. И в этой связи полезно реализовать дополнительные кластеры классов самостоятельно.

Кластер классов является *абстрактным суперклассом*, скрывающим конкретные подклассы. Сделать это намного проще в Objective-C, чем в других языках программирования, поскольку в нем отсутствует ключевое слово для создания объектов. Когда классу посылается сообщение с запросом на новый экземпляр, класс может вернуть не только собственный экземпляр, но и некоторого подкласса.

Если, например, создать экземпляр класса `NSArray` с помощью метода `+arrayWithObjects:`, то можно получить подкласс, заключающий в оболочку простой массив `C`. Если же сделать это с помощью метода `+arrayWithArray:`, то, скорее всего, будет получен подкласс, только ссылающийся на другой массив.

Еще одним характерным примером служит класс `NSNumber`. Он заключает в оболочку единственное значение простого типа `C`. Реализовать то же самое можно было бы и с помощью двух переменных экземпляра: одной — для объединения всех возможных значений простых типов, а другой — для указания на конкретный тип каждого из них. Но добиться этого крайне сложно, не затратив немало памяти. Более

простое решение состоит в реализации отдельного класса для каждого поддерживаемого типа значений.

Большинство кластеров классов предоставляют именованные конструкторы, например `+numberwithFloat:`, для создания нового экземпляра подходящего класса. Если же попытаться создать их с помощью метода `+alloc`, то, как правило, возвращается класс-заполнитель, а затем получается реальный объект в ответ на инициализирующее сообщение.

Примером тому служит класс `Pair`. В этом классе не объявляется никаких переменных экземпляра, да и вообще он не применяется непосредственно. Если отправить этому классу сообщение `+alloc`, то в ответ будет получен *одиночный* экземпляр. Когда же этому классу отправляется инициализирующее сообщение, в ответ получается экземпляр одного из двух закрытых подклассов, как показано в приведенном ниже примере.

```
61 @implementation IntPair
62 - (Pair*)initWithInt: (int)a int: (int)b
63 {
64     first = a;
65     second = b;
66     return self;
67 }
68 - (NSString*)description
69 {
70     return [NSString stringWithFormat: @"(%d, %d)",
71         first, second];
72 }
73 - (float)firstFloat { return (float)first; }
74 - (float)secondFloat { return (float)second; }
75 - (int)firstInt { return first; }
76 - (int)secondInt { return second; }
77 @end
```

Пример кода из файла `classCluster.m`

Каждый из этих конкретных подклассов довольно прост. В обоих объявляются две переменные экземпляра: в одном — типа `int`, в другом — типа `float`. И каждый из них предоставляет единственный инициализатор, который вызывается только из суперкласса, а в самих этих подклассах переопределяются соответствующие методы.

В рассматриваемом здесь примере никаких полезных методов в суперклассе не предоставляется. Но в практической его реализации, скорее всего, появятся методы для выполнения арифметических операций или сравнения вместо четырех методов, представленных в данном примере. С другой стороны, они могут быть предоставлены в отдельной категории.

В кластере классов скрываются все подробности реализации. Любой аспект закрытых классов можно изменить, не видоизменяя или даже не перекомпилируя код, в котором они применяются. Это одна из причин, по которой повторное использование кода намного более распространено в Objective-C, чем в таких языках, как C++. В Objective-C очень просто получить код с открытым интерфейсом и закрытой реализацией, но без какого-либо кода, в котором этот интерфейс используется в зависимости от любых аспектов его реализации.

Для подклассификации кластера классов обычно требуется реализовать небольшое количество *примитивных методов*. А все остальные методы реализуются в суперклассе с точки зрения этих основных функций. Более подробно этот вопрос будет рассматриваться в разделе, посвященном подклассификации коллекций, главы 7.

Если, например, требуется создать новый подкласс, производный от класса `NSString`, то с этой целью следует также реализовать методы `-length` и `-characterAtIndex:`. А все остальные методы в данном классе должны лишь вызывать оба эти метода. Более эффективный способ состоит в том, чтобы реализовать и метод `-getCharacters:range:`.

Применение циклов исполнения

```
18 [[NSRunLoop currentRunLoop] run];
```

Пример кода из файла timer.m

Шаблон проектирования, в котором определяется структура большинства нетривиальных программ на Objective-C, называется *циклом исполнения*. В общем случае в цикле исполнения ожидаются данные из одного или более источника событий, а затем исполняется код в качестве реакции на эти события в соответствии с моделью *событийно-ориентированного программирования*.

В среде Foundation цикл исполнения реализуется в классе NSRunLoop. В процессе разработки приложения экземпляр этого класса зачастую получается автоматически. Но его можно также создать явным образом в инструментальных средствах, работающих в режиме командной строки.

В основном цикле исполнения разрабатываемого приложения от оконного сервера принимаются события, соответствующие нажатиям клавиш, перемещениям мыши, касаниям сенсорного экрана и т.д. В ряде других классов источники событий вводятся в цикл исполнения неявным образом. Так, в классе NSTimer, подробнее рассматриваемом в главе 8, в цикл исполнения вводится источник события от таймера, что дает возможность выполнить некоторое действие в заданный момент времени.

Механизм уведомлений, более подробно рассматриваемый в главе 16, позволяет доставлять уведомления о произвольных событиях, наступающих в программе, посредством цикла исполнения, а регистрировать эти уведомления можно в дескрипторах файлов средствами класса NSFileHandle.

Сам же цикл исполнения тесно связан с управлением памятью. У каждого шага цикла исполнения имеется свой автоматически освобождаемый пул. Это означает, что

объекты, которым отправляется сообщение `-autorelease` на шаге цикла исполнения, будут разрушаться по завершении шага цикла, если только они не сохраняются в каком-нибудь другом месте.

В силу этого, а также того факта, что цикл исполнения применяется для доставки уведомлений о событиях, наступающих при вводе пользователем данных, имеет смысл сохранять как можно более малым объем операций, выполняемых на одном шаге цикла.

В большинстве программ на Objective-C потребность в непосредственном взаимодействии с циклом исполнения отсутствует. Он функционирует в фоновом режиме, в то время как остальные классы предоставляют более удобный интерфейс.

5

Числа

Одно из главных отличий Objective-C от Smalltalk заключается в том, что в Objective-C все примитивные (необъектные) типы данных наследуются от языка C. К их числу (в порядке возрастания длины) относятся следующие типы данных: `char`, `short`, `int`, `long` и `long long`, как со знаком (*signed*), так и без знака (*unsigned*); а также два типа данных с плавающей точкой: `float` и `double`.

Все перечисленные выше типы данных ведут себя точно так же, как и в C, полностью подчиняясь правилам продвижения типов. А в компиляторах Objective-C можно также обнаружить поддержку типа `long double`, который не зависит от конкретной архитектуры.

Аналогичная ситуация, хотя и с некоторыми существенными отличиями, наблюдается и в языке Java, где имеется небольшой выбор необъектных типов. В Java *встроенные типы данных* определены с фиксированной длиной. А в языке C они определены с минимальной точностью. Так, по спецификации на C тип `int` “имеет естественную длину, определяемую архитектурой исполняющей среды”, тогда как в Java он определен явным образом как “32-рядное целое число со знаком в дополнительном коде”.

В дополнение к примитивным типам в C допускается также определять новые имена для уже существующих типов с помощью ключевого слова `typedef`. Наиболее распро-

страненной причиной для этого служит то обстоятельство, что по спецификации на С не требуется указывать конкретную длину ни для одного из стандартных типов данных, но достаточно, чтобы длина каждого последующего типа данных была не меньше длины предыдущего. В частности, на некоторых современных платформах тип `int` имеет длину 16, 32 и 64 бита, а следовательно, полагаться на конкретную его длину на этих платформах нельзя.

В Mac OS X поддерживаются режимы ILP32 и LP64. Оба сокращения служат для обозначения отличий в длине примитивных типов С, поддерживаемых на этой платформе. В частности, сокращение ILP32 означает, что типы `int`, `long` и указатели имеют длину 32 бита. А сокращение LP64 означает, что типы `long` и указатели имеют длину 64 бита, тогда как значения остальных типов неявно считаются более короткими. С другой стороны, на платформе Microsoft Windows поддерживается режим LLP64 для 64-разрядных архитектур, причем оба типа, `int` и `long`, остаются 32-разрядными и только указатели типа `long` имеют длину 64 бита. Эти отличия вызывают определенные затруднения, если тип указателя требуется благополучно привести к типу `long`. И если на большинстве платформ, включая и Win32, такое приведение осуществимо, то на платформе Win64 оно невозможно.

Приведение типа указателя к целочисленному типу вызывает серьезные трудности. Так, тип `long long` имеет длину не меньше 64 битов, поэтому на любой современной платформе гарантируется достаточная длина этого типа данных для хранения любого указателя, тогда как на 16- или 32-разрядных платформах он может оказаться слишком длинным. В версии C99 внедрено ключевое слово `typedef` с префиксом `intptr_t` с целью точно определить длину указателя. Аналогичное языковое средство внедрено в Objective-C компанией Apple в виде класса `NSInteger`. Этот класс широко применяется в среде

Сосоа и обеспечивает такую же длину, как и у конкретного указателя. Имеется также класс `NSNumber` для поддержки целочисленных значений без знака.

В коде графического пользовательского интерфейса нередко встречаются эквивалентные классы `CGFloat` и `CGFloat`. Оба класса обеспечивают такую же длину указателя, как и у типа `float` на 32-разрядных платформах или у типа `double` на 64-разрядных платформах.

Сохранение чисел в коллекциях

```
6 NSMutableArray *array = [NSMutableArray array];
7 [array addObject: [NSNumber numberWithInt: 12]];
```

Пример кода из файла `numberInArray.m`

Во всех стандартных коллекциях классов Objective-C допускается сохранение объектов, но, помимо них, в коллекциях зачастую требуется сохранять и данные примитивных типов. В качестве выхода из этого положения выбирается *упаковка* — заключение примитивного типа данных в оболочку объекта.

Для этой цели служит иерархия класса `NSNumber`. Класс `NSNumber` предназначается для заключения в оболочку одиночного значения примитивного типа. Этот довольно обобщенный класс служит характерным примером *кластера классов*. При создании экземпляра класса `NSNumber` в итоге получается подкласс, специально предназначенный для хранения разного рода данных. Так, если в экземпляре класса `NSNumber` требуется сохранить указатель, этот экземпляр не должен занимать столько же места, сколько отводится экземпляру, содержащему `NSRect` — структуру C, в которой хранятся четыре объекта типа `CGFloat`.

Особое значение имеет конкретный подкласс `NSNumber`, производный от класса `NSNumber`. Этот класс предназначен для заключения в оболочку одиночных числовых значений

и может быть инициализирован данными любых стандартных примитивных типов C.

Для обоих этих классов выделяется конструктор `+valueWithBytes:objCType`. В качестве первого его аргумента служит указатель на некоторое значение, а в качестве второго — *кодировка типа* этого значения в Objective-C. Кодировки типов являются символьными строками, представляющими конкретные типы данных. Они интенсивно используются для интроспекции в Objective-C. В частности, типы любых методов или переменных экземпляра класса можно выбрать сначала из символьной строки их кодировки, а затем произвести их синтаксический анализ, чтобы получить соответствующие типы во время компиляции.

Кодировку любого типа можно получить с помощью директивы `@encode()`. Эта директива аналогична функции `sizeof()` в C, но вместо длины искомого типа в виде целочисленного значения она возвращает кодировку этого типа в виде символьной строки C. В качестве очень удобного приема обращения с кодировками типов служит применение расширения функции `typeof()` в GCC. Оно возвращает тип выражения, и его можно сочетать с директивой `@encode()` следующим образом:

```
NSValue *value =  
    [NSValue valueWithBytes: &aPrimitive  
        objCType: @encode(typeof(aPrimitive))];
```

В этом фрагменте кода возвращается объект типа `NSValue`, заключающий в оболочку тип `aPrimitive`. Этот код будет работать независимо от конкретного примитивного типа. Его можно заключить в оболочку макрокоманды, но следует иметь в виду, что выражение, передаваемое в качестве параметра, не должно иметь побочных эффектов.

Следует также иметь в виду, что в рассматриваемом здесь примере кода в качестве параметра методу должен передаваться указатель на значение примитивного типа.

Кодировка примитивного типа будет использована в данном методе для выяснения длины этого типа, после чего он будет скопирован.

Одни конструкторы используются чаще других. Так, если требуется создать экземпляр класса `NSNumber` из целочисленного значения, это можно сделать следующим образом:

```
NSNumber *twelve = [NSNumber numberWithInt: 12];
```

Получающийся в итоге объект может быть затем добавлен в коллекцию. В отличие от экземпляров класса `NSValue`, экземпляры класса `NSNumber` упорядочены. Это позволяет отсортировать коллекции, содержащие экземпляры класса `NSNumber`.

В приведенном ниже примере кода из файла `numberArray.m` целая группа экземпляров класса `NSNumber` сохраняется в массиве с помощью селектора `-compare:`. Как следует из приведенного далее результата выполнения этого кода, упорядочение осуществляется независимо от количества созданных экземпляров.

```
6 NSArray *a = [NSArray arrayWithObjects:
7   [NSNumber numberWithInt: 2],
8   [NSNumber numberWithInt: 300.057],
9   [NSNumber numberWithInt: 1],
10  [NSNumber numberWithDouble: 200.0123],
11  [NSNumber numberWithLongLong: LLONG_MIN],
12  nil];
13 NSArray *sorted =
14   [a sortedArrayUsingSelector: @selector(compare
15   :)];
16 NSLog(@"%@", sorted);
```

Пример кода из файла `numberArray.m`

```
1 20100315 14 :50:48.166 a.out[51465:903] (
2   "9223372036854775808",
```

```

3      "2",
4      1,
5      "200.0123",
6      "300.057",
7      18446744073709551615
8  )

```

Результат выполнения кода из файла `numberArray.m`

Выполнение десятичных арифметических операций

```

6  NSDecimalNumber *one =
7      [NSDecimalNumber one];
8  NSDecimalNumber *fortyTwo =
9      [NSDecimalNumber decimalNumberWithString: @"42"];
10 NSDecimalNumber *sum =
11     [one decimalNumberByAdding: fortyTwo];
12 NSDecimal accumulator = [sum decimalValue];
13 NSDecimal temp = [fortyTwo decimalValue];
14 NSDecimalMultiply(&accumulator, &accumulator, &temp,
15     NSRoundPlain);
15 temp = [one decimalValue];
16 NSDecimalAdd(&accumulator, &accumulator, &temp,
17     NSRoundPlain);
17 NSDecimalNumber *result =
18     [NSDecimalNumber decimalNumberWithDecimal:
19     accumulator];

```

Пример кода из файла `decimal.m`

Для операций с числами в C предоставляются два типа данных: целочисленный и с плавающей точкой. Значения с плавающей точкой образуются из двух составляющих: мантиссы и порядка. В двоичной форме они представлены в виде мантиссы, умноженной на два в степени, обозначающей порядок числа.

Трудности выполнения операции со значениями с плавающей точкой объясняются двоичной формой их записи. Это

означает, что их точность определяется количеством двоичных разрядов, что не всегда удобно. Так, в финансовых расчетах может возникнуть потребность в сохранении сумм с точностью до четырех цифр. Но этого не позволяют добиться значения с плавающей точкой. Например, число 0,1 нельзя представить каким-нибудь конечным двоичным значением с плавающей точкой, подобно тому, как число 0,1 по основанию три (одну треть) нельзя представить какой-нибудь конечной последовательностью десятичных цифр.

Двоичное число представляет собой сумму показателей степени двух, подобно тому, как десятичное число — сумму показателей степени десяти. В дробных значениях цифры после запятой, отделяющей целую часть числа от дробной, обозначают половинные, четвертные, восьмые и прочие доли числа. Если попытаться сформировать значение 0,1, складывая показатели степени двух, то добиться желаемого так и не удастся, хотя результат и будет постепенно приближаться к желаемому. То же самое происходит при попытке сформировать одну треть, складывая показатели степени десяти (три десятых плюс три сотых плюс три тысячных и т.д.).

В качестве выхода из столь затруднительного положения можно, в частности, воспользоваться арифметическими операциями с плавающей точкой. Так, вместо долларов денежные суммы можно хранить в сотых долях цента. Но при этом нужно не забыть нормализовать значения в ограниченном диапазоне представления чисел целочисленного типа. В Objective-C предоставляется другой выход из положения: десятичные типы с плавающей точкой.

Тип `NSDecimal` является структурой C, представляющей десятичное значение. Как ни странно, но в C отсутствует прикладной интерфейс API для создания подобных структур. Поэтому и приходится сначала создавать экземпляр класса `NSDecimalNumber`, а затем отправлять ему сообщение `-decimalValue`.

Далее, имеются две возможности для выполнения арифметических операций. Во-первых, экземпляры класса `NSDecimalNumber` являются неизменяемыми. Поэтому новые экземпляры данного класса можно создать в результате арифметической операции, отправив, например, сообщение `-decimalNumberByAdding:` одному из них. И во-вторых, можно воспользоваться доступным в C прикладным интерфейсом API, чтобы видоизменить нужное значение непосредственно в структуре.

Первая возможность оказывается более удобной в том случае, если выполняется лишь одна арифметическая операция, а ее результат сохраняется в объекте. Если же арифметические операции выполняются в несколько этапов, то удобнее воспользоваться доступным в C прикладным интерфейсом API. В этом случае видоизменяется сама структура, и поэтому создавать новый объект на каждом промежуточном этапе не нужно.

На заметку

В спецификацию C1X включены типы десятичных чисел, которые поддерживаются некоторыми компиляторами в виде расширения. Но тип `NSDecimal` не совместим с ними. Впрочем, для большинства платформ это не имеет особого значения. Если же прикладная программа предназначена для такой платформы, как POWER6 компании IBM, где имеется аппаратная поддержка арифметических операций, то десятичными типами лучше пользоваться непосредственно.

Но ни одна из этих возможностей не обеспечивает достаточного быстродействия. Десятичные числа в данном случае представлены массивом цифр, а обрабатываются они попарно после нормализации двух предыдущих чисел. Аналогичной производительности можно ожидать и от программной реализации операций с плавающей точкой. Возможно, она окажется еще ниже, поскольку тип

`NSDecimal` не нашел широкого применения, а следовательно, не подвергался существенной оптимизации.

Класс `NSDecimalNumber` является подклассом, производным от класса `NSNumber`, и поэтому от него следует ожидать поддержки всех видов преобразования объектов типа `NSNumber` в символьные строки, как будет показано в следующем разделе. Их можно также преобразовать в примитивные типы C, используя стандартные методы для доступа к этим числовым объектам, но подобные методы могут усесть или округлить десятичные значения.

Преобразование символьных строк в числа

```
6  int answer = [@"42" intValue];
7  NSString *answerString =
8      [NSString stringWithFormat:@"%d", answer];
9  NSNumber *boxedAnswer =
10     [NSNumber numberWithInt: answer];
11  NSCAssert([answerString isEqualToString:
12     [boxedAnswer stringValue]],
13     @"Both strings should be the same");
```

Пример кода из файла `strtonum.m`

Символьные строки можно преобразовать в числа несколькими способами. У многих объектов, представляющих простые типы данных, имеются методы вроде `-intValue` для возврата целочисленного представления получателю.

В классе `NSString` предоставляется несколько методов из этого семейства. Так, если имеется символьная строка, содержащая числовое значение, ей можно отправить сообщение `-doubleValue`, `-floatvalue`, `-intValue` или `-longLongValue`, чтобы преобразовать ее в числовое значение любого из этих типов. В безопасных 64-разрядных версиях среды Foundation символьной строке можно также отправить сообщение `-integerValue`, чтобы получить в ответ объект типа `NSInteger`.

Имеются также способы преобразования в обратном порядке: чисел в символьные строки. В главе 6 будет рассмотрен один из способов преобразования целых чисел в символьные строки. Так, если вызвать метод `+stringwithFormat:` для объекта типа `NSString`, то можно построить символьную строку из любых примитивных типов C, подобно тому, как символьная строка формируется с помощью функции `sprintf()` в C.

Если число уже имеется в экземпляре класса `NSNumber`, то символьную строку можно получить из него двумя способами, один из которых является оболочкой для другого. В частности, метод `-descriptionwithLocale:` возвращает символьную строку, сформированную в результате форматирования числа в соответствии с конкретными региональными стандартами.

В действительности этот метод сам не выполняет никакого преобразования. Он лишь отправляет сообщение `-initWithFormat:locale:` новому экземпляру класса `NSString`. Содержимое формирующей строки зависит от типа числа. Например, числа с плавающей точкой двойной точности преобразуются с помощью формирующей строки `@"%0.16g"`. Эта строка форматирует строковое представление числа до 16 значащих цифр числа, включая его порядок, если требуется.

Вид десятичного разделителя (точки или запятой) определяется региональными стандартами. Так, если отправить объекту `NSNumber` сообщение `-stringValue`, это будет равнозначно отправке сообщения `-descriptionwithLocale:` со значением `nil` аргумента одноименного метода. В этом случае используются *канонические региональные стандарты*, а это означает отсутствие локализации, и поэтому полученный результат будет выглядеть одинаково на любой платформе.

Чтение чисел из символьных строк

```
6 NSScanner *parser =
7 [NSScanner scannerWithString: @"1 plus 2"];
8
9 int operands[2];
10 NSString *operation;
11
12 [parser setCharactersToBeSkipped:
13 [NSCharacterSet whitespaceCharacterSet]];
14
15 [parser scanInt: operands];
16 [parser scanCharactersFromSet:
17 [NSCharacterSet letterCharacterSet]
18 intoString: &operation];
19 [parser scanInt: operands+1];
```

Пример кода из файла `scanner.m`

Всякий начинающий программировать на C прежде всего учится пользоваться функциями `printf()` и `scanf()`. Эти функции очень похожи, хотя одна из них выполняет практически обратные действия другой, но обе позволяют строить форматированные символьные строки и производить их синтаксический анализ. Как было показано ранее, в классе `NSString` имеется приблизительный аналог функции `sprintf()`, что дает возможность строить символьные строки из формирующих строк и переменных, но что же на самом деле является аналогом функции `sprintf()` в Objective-C? Как произвести синтаксический анализ содержимого заданной символьной строки?

Ответы на эти вопросы можно найти в классе `NSScanner`. Этот класс содержит очень эффективные средства разбиения на лексемы. Сначала создается экземпляр класса `NSScanner`, присоединяемый к символьной строке, а затем ее содержимое анализируется лексически по частям.

Все сообщения, отправляемые этому лексическому анализатору, имеют одинаковую форму. Они принимают в качестве параметра указатель на переменную и возвращают логическое значение типа `BOOL`, обозначающее, был ли достигнут успешный результат. Лексический анализатор сохраняет текущий индекс лексического анализа в символьной строке, инкрементируя его только при успешном результате подобного анализа, что дает возможность проанализировать следующие символы в строке иначе. Средствами класса `NSScanner` нетрудно реализовать лексический анализ с упреждающим считыванием и возвратом. Так, если послать лексическому анализатору сообщение `-scanLocation:`, он возвратит текущий индекс в анализируемой символьной строке. А далее можно попытаться выполнить лексический анализ, дойти до ошибки и вернуться назад, послав сообщение `-setScanLocation:`, чтобы восстановить старый индекс.

Одним из самых эффективных в классе `NSScanner` считается метод `-scanCharactersFromSet:intoString:`. Этот метод считывает содержимое символьной строки из текущего места ее лексического анализа до тех пор, пока не встретится символ, отсутствующий в заданном наборе. Как будет показано в главе 6, подобным образом можно сформировать экземпляры класса `NSCharacterSet` с любым произвольным набором символов или же воспользоваться одним из стандартных наборов символов.

В примере кода, приведенном в начале этого раздела, из символьной строки считывается число, затем слово и далее еще одно число. Число считывается встроенным методом `-scanInt:`, а со словом дело обстоит немного сложнее. Для этой цели в данном случае используется набор всех символов в экземпляре класса `NSCharacterSet`.

Но в данном примере используется не только экземпляр класса `NSCharacterSet`. Лексический анализатор настроен

также на пропуск пробелов. Для этой цели ему посылается сообщение `setCharactersToBeSkipped:`, предписывающее игнорировать любые символы из набора, передаваемого в качестве аргумента. Передачей набора символов пробела лексическому анализатору предписывается пропускать любые пробелы, встречающиеся в промежутках между его вызовами. Если на позиции, с которой лексический анализатор начинает считывание из символьной строки, когда ему посылается сообщение для лексического анализа, находятся символы, присутствующие в данном наборе, они пропускаются. Но они не будут пропущены при анализе лексемы, поэтому символьная строка "1 2" будет прочитана как два отдельных числа, а не как число 12.

Манипулирование символьными строками

В Objective-C допускаются два вида символьных строк: строки C и строки Objective-C, называемые также *строковыми объектами*. В качестве расширения в GCC допускается также пользоваться символьными строками из языка Pascal, но они применяются крайне редко и поддерживаются лишь для совместимости с библиотеками Pascal.

Строка C представляет собой довольно примитивный тип данных в виде массива символов, завершающегося байтом пустого символа NULL. На нее не распространяется принцип кодировки символов, и в ней можно хранить любую последовательность байтов, не содержащую нулевой байт. Кодировка UTF-8 была специально предназначена для применения в символьных строках C. Это кодировка с переменной длиной символов, в которой отсутствуют нулевые байты — даже в многобайтовых последовательностях.

Символьная строка Objective-C представляет собой высокоуровневую абстракцию. Она доступна в виде последовательности символов в кодировке UTF-16, но ее внутреннее представление является закрытым. Если код Objective-C компилируется для применения вместе с динамической библиотекой GNU, в таком случае строковые константы будут представлены экземпляром класса со следующими тремя переменными экземпляра: указателем `isa`, указателем на символьную строку C и целочисленной переменной, содержащей длину этой строки.

Динамическая библиотека установит указатель `isa` на нужный класс при загрузке исполняемого модуля.

Для доступа к отдельным символам в строке Objective-C служит тип `unichar`, представляющий один символ в кодировке UTF-16. Следует иметь в виду, что UTF-16 также является кодировкой с переменной длиной символов и что один символ в уникоде (Unicode) может обозначать несколько символов в кодировке UTF-16. Более того, для обозначения одного специального знака с метками вроде ударений может потребоваться не один символ в уникоде.

Создание строковых констант

```
4 const char* cstring = "C string";  
5 NSString *objcstring = @"ObjectiveC string";  
6 CFStringRef cfstring = CFSTR("Core Foundation  
    string");
```

Пример кода из файла `constantStrings.m`

В Mac OS X символьные строки Objective-C *свободно со-
стыкованы* с символьными строками прикладного интер-
фейса Core Foundation (CF). Строковые константы CF ото-
бражаются аналогичным образом на символьные строки
Objective-C в динамической библиотеке GNU, хотя и с не-
которыми отличиями. В указателе `isa` на них устанавливается внешняя ссылка, присутствующая в среде Core
Foundation, и поэтому для манипулирования ими не тре-
буется привязывать динамическую библиотеку Objective-C
или среду Foundation. У них имеется также поле флагов,
позволяющее выбрать их внутреннее представление в ко-
дировке UTF-8 или UTF-16 в зависимости от того, какая из
них окажется более эффективной для представления содер-
жимого этих строковых констант.

Для создания строковых констант Objective-C строко-
вый литерал константы C предваряется знаком `@`, как, на-
пример, `"string"`. А для создания строковых констант

Core Foundation в коде Objective-C и C служит макрокоманда `CFSTR()`. Эта макрокоманда вызывает встроенную в компилятор функцию, формирующую объект строковой константы.

Сравнение символьных строк

```
6 NSString *twelve = @"12";
7 NSString *twelveFromInt =
8     [NSString stringWithFormat:@"%d", 12];
9 NSNumber *numberTwelve =
10    [NSNumber numberWithInt: 12];
11
12 if ([twelve isEqualToString: twelveFromInt])
13     NSLog(@"Both strings are equal");
14 if (twelve == twelveFromInt)
15     NSLog(@"Both strings are identical");
16 if (twelve == @"12")
17     NSLog(@"Constant strings are identical");
18 if ([twelve isEqual: numberTwelve])
19     NSLog(@"String is equal to number");
```

Пример кода из файла `CompareStrings.m`

Простейший и наиболее очевидный способ сравнения символьных строк на равенство состоит в использовании оператора сравнения `==` в C. К сожалению, такой способ оказывается работоспособным далеко не всегда, что может привести к едва заметным программным ошибкам. Ведь с помощью оператора сравнения проверяется тождественность, а не равенство символьных строк. Подобным способом, в частности, проверяется, являются ли они одним и тем же объектом, а не объектами с равными значениями.

Сравнение двух указателей выполняется намного быстрее, чем сравнение двух символьных строк, и даже быстрее, чем отправка сообщения. Поэтому иногда может быть быстрее попробовать сначала сравнить указатели. В результате сравнения указателей можно выяснить, являются ли символьные строки одинаковыми или только могут быть та-

ковыми, но нельзя узнать, что они совершенно разные. Это позволяет ускорить выполнение кода, если в нем часто сравниваются символьные строки, но зачастую служит характерным примером *преждевременной оптимизации*.

Если требуется проверить две символьные строки на равенство, то для этой цели следует воспользоваться методом `-isEqualToString:`. Этот метод принимает в качестве аргумента другую сравниваемую строку Objective-C. Для сравнения двух объектов Objective-C им отправляются сообщения `-isEqual:`, но именно их получатель должен решить, каким из других объектов он считает себя равным.

При компиляции кода из примера, приведенного в начале этого раздела, компилятор обычно объединяет строковые константы, и поэтому константы `twelve` и `@"12"`, по существу, будут обозначать один и тот же объект. У компилятора нет особых причин не сделать такое объединение, хотя это не гарантируется.

Более интересный случай представляет сравнение символьной строки и экземпляра объекта `NSNumber`. На первый взгляд, символьная строка в данном случае сравнивается с результатом отправки сообщения `-stringValue` числовому объекту, но на самом деле это не так.

Это обстоятельство важно иметь в виду, поскольку равенство должно быть коммутативным. Если `[a isEqual: b]`, то в коде чаще всего будет сделано предположение, что и `[b isEqual: a]`. И это особенно справедливо для классов из коллекций. Так, если ввести два объекта в коллекцию `NSMutableSet`, где равенство поддерживает лишь в одном направлении, но не в другом, то результат окажется неопределенным.

Зачастую при сравнении символьных строк (или соответствующих объектов) требуется выяснить не только их равенство, но и упорядочение. Именно здесь дело начинает принимать более серьезный оборот. Если для сравнения двух чисел имеется единое каноническое упорядочение, то для символьных строк — несколько вполне определенных

строгих упорядочений. В каком, например, порядке следует расположить строки "etoile", "йтоїлй" и "Etoile"?

Если сравнить эти символьные строки с учетом регистра, то первая и последняя строки будут иметь одинаковый ранг упорядочения. Следуют ли символы с ударением до или после их аналогов без ударения либо до или после всех символов без ударения, зависит от конкретных региональных стандартов. Так, если имеется список файлов, то пользователь программы может выказать сильное неудовольствие, если в результате их сортировки файл `file10` окажется в списке перед файлом `file2`, даже если это словарный порядок сортировки двух имен файлов.

Для простого сравнения строк на основании только символьных значений можно воспользоваться методом `-compare:`. В результате такого сравнения, как, впрочем, и всех других видов сравнения, возвращается объект `NSComparisonResult` перечислимого типа со следующими тремя возможными значениями: `NSOrderedAscending`, `NSOrderedSame` и `NSOrderedDescending`. Если же данные должны быть представлены пользователю программы, то для сравнения строк лучше воспользоваться методом `-localizedCompare:`. Этот метод возвращает две строки, упорядоченные в соответствии с правилами, определенными в выбранном пользователем языке. Оба упомянутых метода служат оболочкой для следующего обобщенного метода сравнения:

```
- (NSComparisonResult)
    compare: (NSString*)aString
    options: (NSStringCompareOptions)mask
    range: (NSRange)range
    locale: (id)locale;
```

Этот метод принимает в качестве второго параметра разные варианты сравнения, в том числе объект типа `NSNumericSearch` для сортировки чисел с расположением 10 после 9, а также объект типа `NSCaseInsensitiveSearch` для сравнения без учета регистра. Если же передать ему преде-

лы сравнения в качестве третьего параметра, то тем самым можно ограничить сравнение с подстрокой у получателя. А если передать в качестве четвертого параметра ссылку на региональные стандарты, то можно указать правила упорядочения для символов, не представленных в коде ASCII. Следует иметь в виду, что если в качестве четвертого параметра передается константа `nil`, то при сравнении символьных строк используются текущие или *канонические* региональные стандарты. В рассматриваемом здесь примере передача константы `nil` означает использование канонических региональных стандартов при сравнении символьных значений в числовых строках.

Посимвольная обработка строк

```
7 for (NSUInteger i=0 ; i<[aString length] ; i++)
8 {
9     unichar c =
10     [aString characterAtIndex: i];
11     // обработать символьную строку c
12 }
```

Пример кода из файла `stringIteratesSlow.m`

Класс `NSString` представляет собой кластер классов. Это абстрактный класс, и поэтому при создании его объекта получается экземпляр некоторого закрытого класса. В любом подклассе, производном от класса `NSString`, должны быть реализованы простые методы `-length` и `-characterAtIndex:`. Первый из этих методов должен возвращать длину символьной строки, а второй — символ по конкретному индексу в строке.

Если требуется выполнить какие-нибудь действия над каждым символом в строке, то обращение к ней можно организовать в цикле, вызывая на каждом его шаге метод `-characterAtIndex:`. Но это крайне неэффективный способ, поскольку он требует отправки сообщения для каж-

дого символа. С точки зрения используемых вычислительных ресурсов такой способ обходится намного дороже, чем доступ к символам из строки `C`. В данном случае для каждого символа посылается также сообщение `-length`, что требуется лишь в том случае, если длина строки может измениться в течение отдельного шага цикла.

В качестве альтернативного способа достаточно получить символьную строку `C`, чтобы затем обработать ее. Например, символьной строке можно было бы отправить сначала сообщение `-UTF8String`, а затем обработать возвращаемую символьную строку `C`. Но, несмотря на всю обоснованность такого подхода, не следует забывать, что символьная строка может оказаться довольно длинной, чтобы вписаться в формат `UTF8`. Поэтому внутреннее представление возвращаемого объекта, возможно, придется преобразовать в формат `UTF8`, выделить память для его сохранения, а впоследствии очистить эту память средствами автоматически освобождаемого пула или “сборки мусора”.

Более эффективный способ доступа к символам из строки предоставляет метод `-getCharacters:range:`. Методы Objective-C, имена которых начинаются с `get`, обычно принимают в качестве параметра указатель на некоторую область памяти, выделенную в вызывающей части кода, а возвращают они данные, хранящиеся в этой области памяти. В данном случае первым аргументом такого метода служит указатель на буфер со значениям типа `unichar`, который должен быть достаточно большим, чтобы хранить запрашиваемое количество символов.

В приведенном ниже примере кода упомянутый выше механизм используется для подсчета количества точек с запятой в исходном файле. Сначала исходный код загружается в символьную строку, а затем он извлекается по десяти символов за один прием. Обратите внимание на применение в данном примере двух вложенных друг в друга циклов. Во внешнем цикле символы получаются блоками,

а во внутреннем цикле — блок обрабатывается посимвольно. Аналогичная структура кода формируется компилятором, когда для циклического обращения к коллекции применяется *быстрое перечисление*.

В данном случае отправлять сообщение требуется уже десяти символам, а не каждому из них. Быстродействие можно попытаться повысить еще больше, если увеличить размер буфера. Следует, однако, иметь в виду, что этот буфер выделяется в стеке. Такой подход способствует повышению быстродействия, но в то же время требует особого внимания, чтобы правильно задать пределы. В противном случае в защите прикладной программы могут образоваться бреши.

```
11 int semicolons = 0;
12
13 NSUInteger length = [str length];
14 NSRange range = { 0, 10 };
15 while (range.location < length)
16 {
17     unichar buffer[10];
18     if (range.location + range.length > length)
19     {
20         range.length = length - range.location;
21     }
22     [str getCharacters: buffer range: range];
23     range.location += 10;
24     for (unsigned i=0 ; i<range.length ; i++)
25     {
26         unichar c = buffer[i];
27         if (c == ';')
28         {
29             semicolons++;
30         }
31     }
32 }
33 printf("Source file contained %d semicolons\n",
34        semicolons);
```

Пример кода из файла StringIterate.m

Напомним, что строковые объекты скрывают свою кодировку. Код из рассматриваемого здесь примера работоспособен независимо от внутреннего способа хранения символьной строки. Так, если это 7-разрядный код ASCII или UTF-8, большинство символов все равно будет преобразовано в формат UTF-16 очень быстро. А если это нечто вроде кодировки Mac Roman, то преобразование символов в нужный формат может отнять чуть больше времени, но все равно оно будет выполнено довольно быстро.

Преобразование кодировок символьных строк

```
5 NSString *str =
6   [NSString stringWithUTF8String:
7    "some text in a C string"];
8 const char *utf16 =
9   [str cStringUsingEncoding:
10    NSUTF16StringEncoding];
11 const char *utf32 =
12   [str cStringUsingEncoding:
13    NSUTF32StringEncoding];
14 const char *macRoman =
15   [str cStringUsingEncoding:
16    NSMacOSRomanStringEncoding];
```

Пример кода из файла `encodings.m`

К числу наиболее широко применявшихся в классе `NSString` из прикладного интерфейса `OpenStep` относились методы `-cString` и `+stringWithCString:`, которые преобразовывали строку C в строку Objective-C, и наоборот. Но дело в том, что в строках C не учитывается кодировка.

Символьная строка C представляет собой массив байтов, в котором можно хранить любой вид символьных данных. Все операции с символами определяются внешним образом, и поэтому осведомленность о кодировке символов

является внешним фактором по отношению к символьной строке. Совсем иначе дело обстоит с символьными строками в Objective-C.

Для преобразования строки C в строку Objective-C, и наоборот, нужно знать кодировку символов в строке C. Функции из стандартной библиотеки C проверяют переменную окружения, чтобы выяснить, какой должна быть эта кодировка. Но, к сожалению, это вызывает немало затруднений. Если с помощью этих функций символьные данные читаются из сокета или файла, а затем из них формируется строка Objective-C, то в конечном итоге получается текст в кодировке, выбранной пользователем программы, а не в кодировке, выбранной создателем файла.

По этим причинам упомянутые выше методы теперь исключены из употребления, а вместо них применяются методы, принимающие в качестве параметра задаваемую явным образом кодировку. Не следует, однако, забывать, что по умолчанию для любой многобайтовой кодировки принимается обратный порядок следования байтов (от старшего к младшему), что было оправданно для компьютеров Macintosh на базе процессора Motorola 68000 и рабочих станций RISC, где язык Objective-C нашел широкое применение, но менее оправданно для ПК на базе процессоров x86 с прямым порядком следования байтов (от младшего к старшему).

Наиболее распространенной для текста в тех странах, где применяется видоизмененный так или иначе вариант латинского алфавита, является кодировка UTF-8. Она позволяет представить одним байтом любые символы в коде ASCII, а большинство остальных символов — двумя байтами¹. В классе NSString имеются следующие служебные методы получения и установки символьных данных в кодировке UTF-8: -UTF8String и +stringWithUTF8String:. Ими

¹ Кодировка UTF-8 совершенно не годится для обозначения идеографических символов, в том числе иероглифов.

можно пользоваться преимущественно в коде для преобразования в символьные строки C, и наоборот.

Для формирования строкового объекта из символьной строки C в какой-нибудь другой кодировке служит метод `+stringWithUTF8String:`. Его первым аргументом служит символьная строка C, а вторым аргументом — член перечислимого типа, представляющий конкретную кодировку. Если символьные данные происходят из старой системы Mac OS, то в качестве их кодировки можно выбрать `NSMacOSRomanStringEncoding`. Эта кодировка применялась по умолчанию до появления поддержки уникода в Mac OS. Если же требуется организовать взаимодействие со старыми системами Windows, то в качестве кодировки можно выбрать `NSWindowsCP1252StringEncoding`. Эта кодировка широко применяется во всем остальном мире. А в новых системах Windows применяется кодировка UTF-16, но предпочтение отдается ее варианту `NSUTF16LittleEndianStringEncoding` с прямым порядком следования байтов.

Если строковые данные сохраняются в конкретной кодировке, то вместо указателей зачастую используются экземпляры класса `NSData`. Это очень простой класс, инкапсулирующий область памяти: указатель на нее и размер. А поскольку в данном случае приходится иметь дело с объектом, то его можно подвергнуть интроспекции, воспользоваться подсчетом ссылок для очистки из памяти и сохранить в коллекции. В целом ряде случаев такой способ оказывается более удобным, чем непосредственное применение указателей.

Обрезка символьных строк

```
6 NSString *str =  
7   @" a String with leading spaces ";  
8 str = [str  
9   stringByTrimmingCharactersInSet:  
10    [NSCharacterSet whitespaceCharacterSet]];  
11 str = [str substringFromIndex: 2];
```

```
12  NSAssert([str isEqualToString:  
13      @"String with leading spaces"],  
14      @"Trimming failed");
```

Пример кода из файла trim.m

Имеющуюся символьную строку нередко требуется обрезать с обоих концов, удалив из нее фиксированное количество символов или же отдельные символы из конкретного набора. Характерным примером второго случая служит удаление начальных и конечных пробелов из символьной строки.

Для этих целей в строковых объектах Objective-C поддерживается немало методов. Так, для удаления лишь начальных или конечных символов из строки наиболее удобным оказывается метод `-substringWithRange:`, вызываемый для объектов класса `NSString`. В итоге будет создан новый строковый объект, содержащий ряд символов в другой символьной строке. Подобная операция над неизменяемыми символьными строками обходится очень дешево с точки зрения используемых вычислительных ресурсов, поскольку в результате ее выполнения символьная строка не копируется, а только возвращается строковый объект, ссылающийся на данные в исходной символьной строке. Но при вызове данного метода для изменяемой символьной строки приходится копировать из нее символы.

Для удаления символов из конкретного набора требуется иметь ясное представление об объектах класса `NSCharacterSet`. Как подразумевает название этого класса, в нем инкапсулируется определенный набор символов, который может быть сформирован из символьной строки, хотя имеются и предопределенные наборы, представляющие конкретный, вполне определенный набор символов.

Этот класс применяется аналогично функциям из заголовочного файла `ctype.h` в языке C. Его объектам можно отправить сообщение `-characterIsMember:`, чтобы проверить, входит ли отдельный символ в заданный набор.

В частности, вызов функции `ispunct()` из библиотеки C равнозначен отправке сообщения `-characterIsMember:` объекту, получаемому в результате отправки сообщения `+punctuationCharacterSet` классу `NSCharacterSet`.

Вариант данной операции в Objective-C оказывается более многословным, но и более обобщенным. Объекты из набора символов без особого труда могут передаваться в коде прикладной программы и использоваться классом `NSString` в самых разных целях. В частности, можно запросить в строке определенный ряд символов из заданного набора или же разделить строку, определив разделители из конкретного набора символов.

Разделение символьных строк

```
6 NSString *str =  
7   @"A String with\twords and spaces";  
8 NSArray *wordswithTab = [str  
9   componentsSeparatedByString:@" "];  
10 NSArray *words = [str  
11   componentsSeparatedByCharactersInSet:  
12   [NSCharacterSet whitespaceCharacterSet]];
```

Пример кода из файла `split.m`

Символьные строки в Objective-C можно разделить тремя способами. Первый из них вполне очевиден: сначала организуется циклическое обращение к символам в строке до тех пор, пока в ней не будет найдено удобное место для ее разделения, а затем вызывается метод `-substringWithRange:` для создания новой лексемы. Этот способ пригоден в самом общем случае разделения символьных строк, хотя и оказывается довольно трудоемким для выполнения тривиальных операций над строками.

В языке C для разделения символьной строки в любом месте, где обнаруживается заданный символ из конкретного набора, можно воспользоваться функцией `strsep()`

или `strtok()`, хотя последняя уже вышла из употребления. Аналогичным образом действует метод `-componentsSeparatedByCharactersInSet:`, вызываемый для строковых объектов класса `NSString`, хотя он и принимает объект типа `NSCharacterSet` вместо символьной строки в качестве своего аргумента.

Для того чтобы этот метод действовал аналогично функции `strtok_r()`, достаточно сформировать набор символов с помощью метода `+characterSetWithCharactersInString:`. В результате вызова этого метода для объекта типа `NSString` возвращается массив символьных строк типа `NSArray`. Аналогичным образом действуют функции C: пустые строки получаются везде, где встречаются два следующих друг за другом разделяющих символа.

С другой стороны, символьную строку можно разделить по вхождениям в нее конкретной подстроки. Это очень удобно при наличии разделителей, состоящих из нескольких символов.

Копирование символьных строк

```
5 NSString *staticString = @"a string";
6 NSString *constantString =
7   [staticString copy];
8 NSString *stringAlias =
9   [constantString retain];
10 NSMutableString *mutableString =
11   [stringAlias mutableCopy];
12 NSMutableString *mutableStringAlias =
13   [mutableString retain];
```

Пример кода из файла `stringcopy.m`

Одним из слабых мест языка C++, где проявляется недостаточное его быстроедействие, является обработка символьных строк. В любой среде, где доступно только ручное управление памятью, наблюдается склонность к защитно-

му копированию объектов с целью соблюсти в чистоте правила владения объектами.

Для языка Objective-C это менее характерно, хотя код зачастую страдает противоположным недостатком: сохранением ссылок на изменяемые объекты, когда требуются их копии. Так, если объекту типа `NSString` отправляется сообщение `-copy`, результат получается таким же, как и при отправке сообщения `-retain`.

В связи с этим невольно возникает вопрос: зачем нужны два метода, делающие одно и то же? Ответ довольно прост: класс `NSMutableString` является подклассом, производным от класса `NSString`. Если отправить сообщение `-retain` объекту типа `NSMutableString`, то подсчет ссылок будет инкрементирован. А если отправить сообщение `-copy`, то будет получен новый объект. Но дело может осложниться еще и тем, что в конкретных подклассах, производных от класса `NSMutableString`, поддерживается режим копирования при записи, в результате чего получаются два объекта, ссылающихся на одни и те же данные, а настоящая копия получается лишь после видоизменения одного из них.

В связи с тем что класс `NSMutableString` является подклассом, производным от класса `NSString`, указатель на объект типа `NSMutableString` можно передать в любую часть кода, где ожидается указатель на объект типа `NSString`. Так, если ожидается неизменяемая символьная строка и она получается, то не имеет значения, копируется ли объект или только делается еще одна ссылка на него. Если же, с другой стороны, получается изменяемая символьная строка, то придется не только сохранить ссылку на нее, но и, возможно, видоизменить ее впоследствии.

Не следует, однако, забывать, что в результате отправки сообщения `-copy` любому подклассу, производному от класса `NSString`, всегда возвращается неизменяемая символьная строка. Если же требуется получить изменяемую символьную строку, то следует отправить другое сообщение, а именно: `-mutableCopy`.

На заметку

Если вы перешли на Objective-C из Java, не забывайте, что класс `NSString` является приблизительным эквивалентом класса `String` в Java, а класс `NSMutableString` — приблизительным эквивалентом класса `StringBuffer` в Java. В Objective-C отсутствует эквивалент ключевого слова `final` в Java, и поэтому у класса `NSString` могут и должны быть подклассы. А изменяемые символьные строки могут использоваться вместо изменяемых строк везде, где это уместно.

Копирование символьных строк по такому образцу можно обнаружить в нескольких местах в среде Foundation. В большинстве классов, имеющих неизменяемый суперкласс и изменяемый подкласс, будет реализован протокол `NSMutableCopying`. В результате отправки сообщения `-mutableCopying` изменяемым или неизменяемым экземплярам этих классов будет получен экземпляр изменяемого подкласса.

Построение символьных строк по шаблонам

```
6 int a = 12;
7 float b = 42.0;
8 const char *cString = "words";
9 id object = [NSObject new];
10 NSString *string =
11     [NSString stringWithFormat:
12         @"%d, %f, %s, %@",
13         a, b, cString, object];
```

Пример кода из файла `formatstring1.m`

Если у вас имеется опыт программирования на C, то одной из первых освоенных вами, вероятно, была функция `printf()`. Это *функция с переменным числом аргументов*, принимающая форматирующую строку в качестве своего первого аргумента. В таком случае содержимое форматирующей строки определяет типы остальных аргументов данной функции.

Язык Objective-C унаследовал среди прочего от C и понятие форматирующей строки. В Objective-C форматирующие строки почти полностью идентичны форматирующим строкам C, но с одним существенным дополнением. Спецификатор формата %@ обозначает в них, что совпадающий с ним аргумент является объектом.

Объектам, как предполагается, должно быть известно, как себя описывать. Каждый объект должен реагировать на сообщение `-description`, в результате чего возвращается объект типа `NSString`, описывающий данный объект. Многие классы наследуют эту функцию от класса `NSObject`, возвращая имя класса и адрес его объекта в памяти. А некоторые классы предоставляют более подходящие сведения. В частности, классы коллекций выводят из памяти свое содержимое подобно *спискам свойств* в стиле NeXT.

На заметку

В отладчик *GNU (gdb)* встроена поддержка методов типа `-description`. При выборе команды печати объекта, сокращенно называемой `ro`, этот объект вызывает свой метод `-description`. Но пользоваться этой командой следует аккуратно и внимательно, поскольку она будет выполнять в процессе отладки некоторый код, который может также содержать программные ошибки. Тем не менее эта команда приносит пользу, если она действует правильно. Подробнее об отладке речь пойдет в главе 18.

Строковые объекты в Objective-C могут быть созданы из форматирующих строк с помощью метода с *переменным числом аргументов*, который действует подобно функции `asprintf()` в C. В качестве первого аргумента этого метода указывается символьная строка Objective-C, содержащая ряд спецификаторов формата, а в качестве остальных аргументов — переменные, используемые вместо спецификаторов формата. В примере кода из файла `formatstring1.m`, приведенном в начале этого раздела, показано, каким обра-

зом форматирующие строки используются вместе с объектами типа `NSString`.

А в приведенном ниже примере кода символьная строка сначала формируется из аргументов, предоставляемых для исполняемого кода, а затем выводится в стандартный поток ошибок. Следует, однако, иметь в виду, что в начале программы из файла `formatstring.m` данного примера необходимо создать объект типа `NSAutoreleasePool`. С помощью форматирующих строк в данном случае создаются временные объекты, которые должны быть автоматически освобождены из памяти. В отсутствие автоматически освобождаемого пула будет выдано предупреждение об утечке этих объектов из памяти. В столь простой программе их утечка произойдет в любом случае, поскольку нет никакой возможности указать автоматически освобождаемому пулу собрать их в “мусор”, а операционная система все равно затребует память по завершении процесса.

```
3  int main(int argc, char **argv)
4  {
5      [NSAutoreleasePool new];
6      NSMutableString *str =
7      [NSMutableString stringWithFormat:
8       @"%d arguments: ", argc];
9      for (int i=0 ; i<argc ; i++)
10     {
11         [str appendFormat: @"%s ", argv[i]];
12     }
13     NSLog(@"%@", str);
14     return 0;
15 }
```

Пример кода из файла `formatstring.m`

В рассматриваемом здесь примере сначала создается изменяемая символьная строка, содержащая количество аргументов, передаваемых программе. А затем организуется циклическое обращение к остальным аргументам, причем каждый из них присоединяется (в виде символьной строки C) к изме-

няемой символьной строке. И наконец, полученный результат передается функции `NSLog()`.

В функции `NSLog()` также используются форматирующие строки. Возникает искушение передать ей только объект `str` в качестве первого аргумента, но это допустимо лишь до тех пор, пока в данной программе не будет указана символьная строка со знаком процента в качестве первого ее аргумента. В этом случае функция `NSLog()` попытается обнаружить второй аргумент и интерпретировать его. Но, к сожалению, второй аргумент для нее не указан, и поэтому она выберет произвольное значение из регистра. Если она интерпретирует его как числовое — это еще полбеды. А если интерпретирует его как указатель (на символьную строку C или объект Objective-C), то это, скорее всего, приведет к аварийному завершению программы. Передавая же функции `NSLog()` аргумент `@:@"%@"`, мы тем самым указываем ей, что у нее имеется единственный настоящий аргумент, которым является объект.

Сопоставление с шаблоном в символьных строках

```
8  NSRegularExpression *regex = [NSRegularExpression
9    regularExpressionWithPattern: @"abcd*"
10     options: 0
11     error: NULL];
12  NSMutableString *str = [@"abcddd fish, wibble
13    abcd, abc, foo" mutableCopy];
14  NSRange all = NSMakeRange(0, [str length]);
15  [regex enumerateMatchesInString: str
16    options: 0
17    range: all
18    usingBlock:
19    ^(NSTextCheckingResult *result, NSMatchingFlags
20      flags, BOOL *stop)
```

```
20  NSLog(@"Matched range: %@", NSStringFromRange
21      ([result range]));
22
23  NSLog(@"Copy with replacement: %@",
24      [regex stringByReplacingMatchesInString: str
25          options: 0
26          range: all
27          withTemplate: @"foo"]);
```

Пример кода из файла `regex.m`

Если вам приходилось раньше программировать на таком языке, как Perl или JavaScript, то вы, вероятно, привыкли пользоваться регулярными выражениями для сопоставления с шаблоном в символьных строках. В последние версии среды Foundation включен класс `NSRegularExpression`, являющийся тонкой оболочкой, в которую заключается код регулярного выражения из библиотеки ICU (International Components for Unicode — Международные компоненты для уникода).

У этого класса имеется единственный примитивный метод, принимающий блок в качестве своего конечного аргумента и вызывающий его при каждом сопоставлении с образцом. А все остальные методы этого класса реализуются на основе данного метода.

В примере кода, приведенном в начале этого раздела, демонстрируются два наиболее распространенных способа применения класса `NSRegularExpression`. В обоих случаях все начинается с создания экземпляра данного класса, инкапсулирующего простое регулярное выражение. Ради эффективности следует попытаться использовать объекты регулярных выражений повторно, вместо того чтобы создавать их всякий раз из символьных строк. После создания экземпляра входная символьная строка подвергается синтаксическому анализу и преобразуется в конечный автомат, быстро пробегающий символьные строки. А поскольку

ку объект регулярного выражения является неизменяемым и потокобезопасным, то часто используемые шаблоны можно хранить в глобальных переменных и повторно использовать их в нескольких потоках.

В первом способе применения регулярных выражений демонстрируется низкоуровневый прикладной интерфейс API. Ему передается блок, вызываемый при каждом сопоставлении с шаблоном. В данном простом примере на экран выводится местоположение каждого совпадения с шаблоном в символьной строке.

В прикладном интерфейсе API более высокого уровня замена обнаруженных совпадений с шаблоном происходит автоматически, и делается это двумя путями. В первом случае берется объект типа `NSString` и возвращается новая его копия с результатом замены, а во втором — берется объект типа `NSMutableString` и в нем производится замена.

Следует, однако, иметь в виду, что шаблон, используемый для замены, не является простой символьной строкой. Он может содержать ссылки на отдельные части исходного регулярного выражения, используя синтаксис `$n`, обычно применяемый в регулярных выражениях в других целях.

Сохранение форматированного текста

```
6 NSDictionary *keyword = [[NSDictionary
7   dictionaryWithObject: @"keyword"
8   forKey: @"type"] retain];
9 NSMutableAttributedString *program =
10   [NSMutableAttributedString new];
11
12 NSAttributedString *fragment =
13   [[NSAttributedString alloc]
14    initWithString: @"int"
15    attributes: keyword];
16 [program appendAttributedString: fragment];
```

Пример кода из файла `richText.m`

Строка — это всего лишь совокупность символов. Прежде чем отобразить ее, обычно требуется указать ряд других атрибутов. Эти атрибуты описывают стили начертания и размера шрифта, выравнивания, подчеркивания и прочие особенности форматирования текста.

Если приходится создавать прикладные программы для обработки структурированного текста, например, в формате HTML или в другой форме семантической разметки, включая DocBook, то упомянутые выше атрибуты могут быть использованы для форматирования заголовков первого уровня или упорядоченного списка.

В среде Foundation предоставляется класс `NSAttributedString` для присваивания символьным строкам произвольных атрибутов. Этот класс позволяет присоединять словари к отдельным рядам символов в строках с приписываемыми атрибутами. Подобные словари могут содержать любые произвольные сведения, в том числе и семантическую разметку.

Этот класс применяется и в среде `AppKit` для определения нескольких ключей к словарям и представления разметки. С его помощью можно присоединить экземпляр класса `NSFont` к фрагменту текста, чтобы отобразить этот фрагмент текста отформатированным указанным шрифтом, используя имеющуюся в `AppKit` систему форматирования текста.

В примере кода, приведенном в начале этого раздела, демонстрируется применение строки с приписываемыми атрибутами для определения ключевых слов. При последующем преобразовании с помощью этой строки можно сформировать атрибуты для отображения данных в текстовом представлении, а также для составления HTML-документа из полученной разметки.

Не следует, однако, забывать, что класс `NSAttributedString` является обобщенным. С его помощью можно присоединить любые атрибуты, которые требуется определить, к отдельным рядам символов в строке.

Работа с коллекциями

Подобно другим рассмотренным ранее языковым средствам, в Objective-C имеются две разновидности коллекций: объектно-ориентированные и примитивные, характерные для C. Первые из них, как обычно, построены на основании последних.

В языке C имеются две разновидности составных типов данных: массивы и структуры. Массивы представляют собой блоки памяти, содержащие однотипные данные. А структуры имеют фиксированное расположение и могут содержать разнотипные данные в качестве своих элементов.

Составные типы данных образуют в C тонкий слой синтаксической надстройки, накладываемый на арифметические операции с указателями. При доступе к элементу массива компилятор умножает индекс массива на длину его элемента и складывает полученный результат с указателем на начало массива. А при доступе к элементу структуры компилятор складывает фиксированное смещение с указателем на начало структуры.

Коллекции в Objective-C представляют собой высокоуровневые структуры. В то же время каждая такая коллекция является объектом подобно большинству других дополнений Objective-C. Связь с коллекциями поддерживается благодаря отправке сообщений.

В среду Foundation входит целый ряд классов коллекций, предназначенных для хранения упорядоченных и неупорядоченных данных, а также отображений и множеств

индексов. В большинстве классов коллекций Objective-C хранятся объекты. В отличие от коллекций STL в C++, которые требуется уточнять на предмет хранящихся в них типов данных, коллекции Objective-C являются разнотипными. Это означает, что в них можно хранить объекты любых доступных в Objective-C типов.

Как и в символьных строках, в большинстве коллекций Objective-C применяется *шаблон изменяемого подкласса*, где суперкласс реализует поведение неизменяемой коллекции, а подкласс предоставляет ее изменяемый вариант.

Все классы, придерживающиеся этого шаблона, реализуют протокол `NSMutableCopying`, а это означает, что коллекции можно отправить сообщение `-mutableCopy` и получить ее изменяемую копию, даже если исходная коллекция не является изменяемой.

Большинство коллекций являются также *кластерами классов*. При создании такого кластера получается некоторый закрытый подкласс. А открытые классы, как правило, являются абстрактными. Все удобные для пользователей методы реализуются на основе небольшого количества примитивных методов. Если же создать собственный подкласс, эти методы придется реализовать самостоятельно.

Применение массивов

```
6 NSArray *array = [NSArray arrayWithObjects:
7   @"array", @"containing", @"string", @"objects",
8   nil];
9 NSMutableArray *mutable = [array mutableCopy];
10 [mutable sortUsingSelector: @selector(
11   localizedCompare:)];
12 [mutable addObject: [NSNumber numberWithInt:
13   12]];
14 [mutable removeAllObjects];
```

Пример кода из файла `nsarray.m`

Помимо массивов C, в среде Foundation предоставляется класс `NSArray` и его изменяемый подкласс `NSMutableArray`. Эти классы реализуют абстрактный тип данных для преобразования в объекты целочисленных значений, находящихся в непрерывном диапазоне, начиная с нуля. Этот тип данных может быть реализован внутренним образом с помощью массивов C, списков с пропусками или некоторых других структур данных в зависимости от того, как он был создан.

Так, если создать массив, соединив два неизменяемых массива, то в итоге можно получить составной массив, в котором ничего не хранится и который служит лишь для доступа к двум другим массивам. С другой стороны, можно получить совершенно новый массив, в зависимости от того, что окажется более эффективным в данном конкретном случае для тех, кто реализует соответствующий подкласс, производный от класса `NSArray`.

Массив Objective-C проще всего создать с помощью метода `+arrayWithObjects:`, реализуемого в обоих классах, `NSArray` и `NSMutableArray`. Это еще один пример метода с *переменным числом аргументов*, принимающего список объектов в качестве своих аргументов и использующего константу `nil` для обозначения конца списка. Но пользоваться переменными в качестве аргументов данного метода следует очень аккуратно, чтобы ни одна из этих переменных не содержала константу `nil`, иначе она будет интерпретирована как конец списка аргументов, а остальные переменные будут просто проигнорированы.

На заметку

В большинстве коллекций Objective-C не допускается хранить константу `nil` в качестве значения. Если же требуется обозначить пустое значение, а не отсутствие значения, то для этой цели можно воспользоваться классом `NSNull`. Этот класс предоставляет одиночный объект, доступный с помощью синтаксической конструкции `[NSNull null]`.

Массивы являются *кластерами классов*. В классе NSArray доступны методы `-objectAtIndex` и `-count`. А все остальные методы реализованы на их основе. Разумеется, в некоторых конкретных классах могут быть реализованы более эффективные варианты этих методов, но если требуется создать новый подкласс, производный от класса NSArray, то все равно придется реализовать оба эти метода, чтобы остальные методы нормально действовали в суперклассе.

Выполнить подклассификацию класса NSMutableArray немного сложнее, поскольку в нем реализуются еще шесть примитивных методов. Но на практике чаще всего приходится пользоваться массивами, а не прибегать к подклассификации. Объект можно ввести в изменяемый массив с помощью метода `-addObject:`, добавляющего его в конец массива, или же метода `-insertObjectAtIndex:`, вводящего объект в массив по указанному индексу. Последний метод смещает все последующие объекты на единицу, чтобы освободить в массиве место для нового элемента. Для замены объекта в массиве служит метод `-replaceObjectAtIndex:withObject:`.

Манипулирование индексами

```
13 NSMutableIndexSet *indexSet =
14     [NSMutableIndexSet indexSetWithIndex: 1];
15 [indexSet addIndexesInRange:
16     [NSMakeRange(5, 20)]];
17 [array removeObjectAtIndexes:
18     indexSet];
```

Пример кода из файла `indexset.m`

Массивы индексируются целочисленными значениями. Зачастую над совокупностью значений в массиве требуется произвести некоторую операцию. Для хранения совокупностей индексов служит специальный класс `NSIndexSet`. Он сохраняет внутренним образом индексные множества в заданных диапазонах, располагая их довольно плотно.

Чаще всего индексные множества служат для собирания совокупности индексов при перечислении массива и удаления их по завершении этой операции. Удалять объекты из коллекции при ее перечислении нежелательно, а зачастую это приводит к генерированию исключения. Если же добавить каждый индекс, который требуется удалить, в индексное множество, то все индексы можно будет удалить сразу по завершении перечисления.

Операции с индексными множествами обычно выполняются более эффективно, чем соответствующие операции с отдельными индексами. Это справедливо даже для относительно простой реализации, поскольку требует отправки меньшего количества сообщений. Кроме того, исключается проверка диапазона. В каждой операции с массивом Objective-C проверяются его границы, чтобы обращение по индексу не происходило за пределами массива. Имея в своем распоряжении индексное множество, получателю остается лишь сравнить результат последнего вызова метода `-lastIndex` с последним индексом в массиве, а затем оперировать каждым элементом массива в индексном множестве по отдельности, не прибегая к проверке диапазона.

Подобно другим коллекциям, индексные множества бывают изменяемыми и неизменяемыми. На практике чаще всего приходится иметь дело с изменяемыми индексными множествами. Манипулирование таким множеством состоит в добавлении и удалении отдельных индексов (типа `NSNumber`) или целых диапазонов индексов. Индексные множества можно использовать для хранения индексов на все что угодно, но наибольшую пользу они приносят в сочетании с объектами типа `NSArray`, имеющими несколько методов, принимающих индексные множества в качестве своих аргументов. Ими можно воспользоваться в качестве альтернативы созданию подмассива, если требуется обработать лишь часть некоторого произвольного подмножества элементов массива.

Сохранение неупорядоченных групп объектов

```
6 NSArray *array = [NSArray arrayWithObjects:
7   @"set", @"object", @"containing",
8   @"seven", @"objects", @"not",
9   @"eight", @"objects", nil];
10 NSMutableSet *set = [NSMutableSet setWithArray: array];
11 NSMutableSet *mSet = [set mutableCopy];
12 NSMutableSet *cSet =
13   [NSMutableSet setWithArray: array];
```

Пример кода из файла nsset.m

Если требуется сохранить коллекцию объектов, не определяя их порядок следования, то для этой цели можно воспользоваться классом `NSSet` и его изменяемым подклассом. В этом классе моделируется математическое множество, и поэтому если ввести в него один и тот же объект дважды, то в нем сохранится лишь одна копия объекта.

Равенство объектов определяется в классе `NSSet` довольно сложным образом. Ведь равенство — это не одно и то же, что и тождество. Оно не означает сохранение двух копий одного и того же объекта, но и не двух объектов с одинаковым значением. Для определения равенства двух объектов одному из них посылается сообщение `-isEqual:`, а другой указывается в качестве аргумента вызываемого при этом метода.

Самый простой способ реализовать класс `NSSet` — сравнить каждый новый объект с каждым существующим объектом. Но это было бы слишком медленно. Правда, класс `NSSet` не действует подобным способом. Вместо этого в нем используется метод `-hash`. Предполагается, что этот метод должен быть реализован каждым объектом, возвращая целочисленное хеш-значение.

Если два объекта равны, у них должно быть одно и то же хеш-значение. Коллекции типа `NSSet` помещают объекты в сегменты памяти в зависимости от их хеш-значений. Для

проверки наличия объекта в коллекции достаточно проверить объекты с одинаковым хеш-значением.

Следовательно, правильное хеширование имеет большое значение для достижения приемлемой производительности классов коллекций. Правда, объекты класса `NSString`, чаще всего сохраняемые в подобных коллекциях, уже содержат эффективную реализацию хеширования.

Единственный мелкий недостаток такого подхода заключается в том, что объект не должен изменять свое хеш-значение, находясь в коллекции. Но добиться этого невозможно для изменяемых объектов, не нарушив ряд других правил. И поэтому очень важно не видоизменять объекты до тех пор, пока они хранятся в коллекции.

Еще одной разновидностью коллекции является множество с подсчетом, реализуемое подклассом `NSCountedSet`, производным от класса `NSMutableSet` и поддерживающим добавление одного и того же объекта в коллекцию несколько раз. Для этой цели в класс `NSMutableSet` вводится метод `-countForObject:`, возвращающий значение, обозначающее, сколько раз конкретный объект был добавлен в коллекцию.

На заметку

В Java и ряде других языков программирования множество с подсчетом называется *множеством с повторяющимися элементами*.

Следует, однако, иметь в виду, что подкласс `NSCountedSet` является производным от класса `NSSet`, и поэтому в некоторых его методах разрешается использовать экземпляры класса `NSCountedSet` вместо экземпляров класса `NSSet`. Так, если отправить сообщение `-count` любому из множеств, созданных в примере, приведенном в начале этого раздела, то в конечном итоге будет получено одно и то же значение. Если же организовать циклическое обращение к любому из этих множеств, то будет получено семь объектов. А для того чтобы выяснить, сколько раз объект повторяется во множестве с подсчетом, достаточно отправить этому множеству сообщение `-countForObject:`.

Создание словаря

```
6 NSMutableDictionary *dict = [NSMutableDictionary  
7   dictionaryWithObjectsAndKeys: @"One", @"1",  
8   @"two", @"2", nil];  
9 [dict setObject: @"three" forKey: @"3"];
```

Пример кода из файла dictionary.m

Словарь, нередко называемый *отображением* или *ассоциативным массивом*, предоставляет коллекцию одних объектов, индексируемых по другим объектам. Чаще всего словари индексируются по символьным строкам, но это далеко не единственная возможность. Впрочем, на классы, которые можно использовать в качестве ключей в словарях, накладывается ряд ограничений.

Словари подобны множествам, и поэтому на них накладываются аналогичные ограничения. Объекты, используемые в качестве ключей, должны реализовывать методы `-hash` и `-isEqual:`, возвращающие логическое значение YES при сравнении объектов и то же самое хеш-значение, если они равны. Это хеш-значение не должно изменяться до тех пор, пока объекты находятся в коллекции.

Кроме того, ключи должны реализовывать протокол `NSCopying`. Когда в словарь вводится пара “ключ–значение”, в нем сохраняется значение, а ключ копируется. Это делается для того, чтобы предотвратить случайное видоизменение ключа. Если передается неизменяемый объект, например, типа `NSString`, то это все равно, что сохранить его. А если передается изменяемый объект, например, типа `NSMutableDictionary`, то в словаре будет использоваться его неизменяемая копия.

И наконец, стоит упомянуть о том, что только словари, в которых используются символьные строки, способны поддерживать механизм *доступа к значениям по ключам*, как поясняется в главе 11. Кроме того, пользуясь различными классами в качестве ключей в одном и том же сло-

варе, следует проявлять особое внимание и аккуратность, чтобы их методы сравнения возвращали правильные значения, когда им передаются экземпляры других классов.

Для обращения к словарям чаще всего применяется метод `-objectForKey:`, возвращающий объект, связанный с заданным ключом, или константу `nil`, если для ключа не задан объект. А для обращения к изменяемым словарям зачастую применяется метод `-setObject:forKey:`, задающий пары “ключ–значение” в словаре.

Циклическое обращение к коллекции

```
19 NSLog(@"The ObjectiveC 1 way:");
20 NSEnumerator *e=[a objectEnumerator];
21 for (id obj=[e nextObject] ;
22      nil!=obj ;
23      obj=[e nextObject])
24 {
25     [obj print];
26 }
27 NSLog(@"Fast enumeration:");
28 for (id obj in a)
29 {
30     [obj print];
31 }
32 NSLog(@"Using blocks:");
33 [a enumerateObjectsUsingBlock:
34     ^(id obj, NSUInteger idx, BOOL *stop)
35     {
36         [obj print];
37     }];
38 NSLog(@"Avoiding enumeration:");
39 [a makeObjectsPerformSelector: @selector(print)];
```

Пример кода из файла `enum.m`

К числу наиболее распространенных операций с коллекциями относится циклическое обращение к ней по каж-

дому ее элементу для выполнения какой-нибудь обработки. Это можно сделать в Objective-C самыми разными способами. А поскольку данная операция с коллекциями довольно распространена, то постоянно появляются все новые способы ее выполнения.

Для циклического обращения к массивам можно пройти по его индексам, начиная с 0 и до [array count] и отправляя сообщение -objectAtIndex:. Но такой способ не очень эффективен. Традиционный способ обращения к элементам коллекции состоит в использовании перечисления и, в частности, экземпляра подкласса NSEnumerator.

На заметку

Если вы переходите на Objective-C из Java, то, заменив мысленно понятие *итератор* на понятие *перечислитель*, вы тотчас обнаружите, что сможете по-прежнему пользоваться большинством уже известных вам шаблонов.

Для того чтобы получить перечислитель коллекции, достаточно отправить ей сообщение -objectEnumerator. Это довольно простой объект. Он реагирует только на сообщения, но на практике чаще всего используется лишь одно из них: -nextObject. В итоге возвращается объект, следующий по порядку в коллекции, или же константа nil, если уже перечислены все объекты, содержащиеся в коллекции.

В версии Objective-C 2 компания Apple внедрила так называемое *быстрое перечисление*. Его можно применять для обращения к коллекциям, реализующим протокол NSFastEnumeration. В этом протоколе определен метод получения нескольких объектов в результате отправки единственного сообщения. Конкретное количество объектов определяется вызывающим кодом и получателем сообщения. Компилятор выделяет в памяти буфер и передает указатель на него получателю сообщения. После этого получатель может скопировать ряд объектов в свой буфер или воз-

вернуть указатель на его внутреннее хранилище, если он сохраняет данные внутренним образом в виде одного массива `C` или группы массивов.

Метод из этого протокола применяется в новой управляющей конструкции: цикле `for...in`. В этом случае компилятор организует два вложенных друг в друга цикла. Во внешнем цикле вызывается метод быстрого перечисления, получающий несколько объектов, следующих по порядку в коллекции. А во внутреннем цикле происходит поочередное обращение к элементам данного массива `C`.

На заметку

Если вы пользуетесь автоматическим подсчетом ссылок (ARC), то соглашение о быстром перечислении оказывается более строгим. В частности, от коллекции требуется, чтобы хранящиеся в ней объекты были действительными в течение перечисления, а иначе она должна уведомить о происшедших в ней изменениях. Вследствие этого присваивание переменной перечисления не разрешается без явного указания описателя. Ведь в противном случае механизму ARC не будет известно, как следует интерпретировать такое присваивание. Как правило, для хранения новых временных значений новую переменную лучше объявить в цикле.

Быстрым перечислением можно пользоваться и без поддержки со стороны компилятора. В заголовочном файле `GSFastEnumeration.h` реализации библиотеки `GNUstep` предоставляются макрокоманды `FOR_IN` и `END_FOR_IN` в качестве расширения, генерирующего такой же самый код для циклов `for...in`, как и компилятор. Но эти команды довольно сложны, поэтому не стоит и пытаться вызывать методы быстрого перечисления самостоятельно.

В `Mac OS X 10.6` компания `Apple` внедрила еще один новый способ циклического обращения к коллекциям с помощью *блоков*. Блоки являются новым расширением языка `C`, но предназначены для тесного взаимодействия с `Objective-C`.

Как и у объектов Objective-C, у блоков имеется указатель `isa`. Он указывает на закрытый класс, реализующий протокол `NSObject`, что дает возможность хранить блоки в коллекциях. Блоки поддерживают также интроспекцию, используя кодировки типов Objective-C, хотя эта поддержка пока еще не стала частью их открытого интерфейса на платформе Mac OS X.

Большинство классов коллекций теперь реализуют методы перечисления с помощью блоков. Безусловно, пользоваться блоками удобно, хотя это и ограничивает переносимость кода. С помощью блоков можно также выполнять параллельные перечисления, передавая объект типа `NSEnumerationConcurrent` в качестве первого аргумента такому методу, как, например, `-enumerateObjectsWithOptions:usingBlock:`. Но, как правило, одного этого оказывается явно недостаточно для повышения производительности, если блоку требуется много времени на выполнение при обращении к каждому элементу коллекции.

Для перечисления с помощью блоков у каждой коллекции имеются свои методы. Так, если коллекция реализована в виде массива, то соответствующий метод принимает индекс объекта в качестве своего аргумента. А если она реализована в виде словаря, то ее метод принимает в качестве своих аргументов ключ и значение. Эти методы будут действовать медленнее, чем другие их аналоги, поскольку в них требуется вызывать блок для каждого объекта, что по издержкам вычислительных ресурсов равнозначно вызову функции, хотя они и могут оказаться более удобными.

В некоторых случаях можно вообще обойтись без перечисления. Например, в классе `NSArray` для этой цели реализуется метод `-makeObjectsPerformSelector`, а другой его вариант принимает объект в качестве второго аргумента. Если во время перечисления только отправляется сообщение каждому объекту, то этими методами можно воспользоваться вместо организации цикла.

Поиск объекта в коллекции

```
6 NSArray *array = [NSArray arrayWithObjects:
7   @"a", @"group", @"of",
8   @"string", @"objects", nil];
9 NSSet *set = [NSSet setWithArray: array];
10 NSUInteger i =
11   [array indexOfObjectIdenticalTo: @"group"];
12 if (NSNotFound == i)
13   i = [array indexOfObject: @"group"];
14 NSString *original = [set member: @"string"];
```

Пример кода из файла getObject.m

В обоих классах, NSArray и NSSet, реализуется метод `-containsObject:`, возвращающий логическое значение YES, если в коллекции содержится объект, равный аргументу данного метода. В данном случае используется то же самое понятие равенства, что и в рассмотренных ранее классах коллекций: два объекта считаются равными, если при отправке одному из них сообщения `-isEqual:` с другим объектом в качестве аргумента соответствующего вызываемого метода возвращается логическое значение YES.

В массиве, скорее всего, потребуется выяснить точное местоположение объекта, а не только его присутствие. Для этой цели служат следующие два метода: `-indexOfObject:` и `-indexOfObjectIdenticalTo:`. Во втором из них для выявления факта равенства выполняется сравнение указателей.

Но все это замедляет поиск объектов в массивах, поскольку требует линейного поиска, а зачастую — отправки сообщения каждому объекту. Как правило, упомянутыми выше методами следует пользоваться для поиска объектов только в мелких массивах или нечасто вызываемых ветвях кода. Если же подобное обратное преобразование требуется выполнять часто, то лучше воспользоваться словарем или чем-то аналогичным для хранения обратного отношения.

При поиске объектов во множествах производительность несколько отличается. А поскольку объекты из множеств хранятся в сегментах памяти по их хеш-значениям, то во множестве можно быстро учесть немало объектов как отличающихся от остальных. Очевидно, что множества неупорядочены, и поэтому у них отсутствует метод для поиска индекса объекта. Но у них имеется метод для поиска хранящихся объектов, равных искомому объекту.

Метод `-member:`, вызываемый для объекта типа `NSSet`, возвращает хранящийся во множестве объект, равный аргументу этого метода. Это очень удобно для обеспечения однозначности в объектах. В приведенном ниже примере кода из файла `unique.m` демонстрируется метод, в котором однозначно определяемые объекты строковых констант строятся из символьных строк `C`. Если передать одну и ту же символьную строку `C` этому методу дважды, то оба раза будет возвращен один и тот же экземпляр класса `NSString`.

Следует, однако, иметь в виду, что рассматриваемый здесь метод не является потокобезопасным. Существует вероятность возникновения гонок между проверкой наличия символьной строки во множестве, и добавлением новой строки, если такая строка отсутствует. В потокобезопасном варианте данного метода потребовалось бы запросить блокировку после вызова метода `-member:`, отправить вызывающее его сообщение еще раз и, наконец, снять блокировку после добавления нового объекта.

```
3 static NSMutableSet *unique_strings;
4
5 NSString *uniqueString(const char *str)
6 {
7     NSString *new = [[NSString alloc]
8         initWithUTF8String: str];
9
10    NSString *old =
11        [unique_strings member: new];
```

```
12
13 if (nil != old)
14 {
15     [new release];
16     return old;
17 }
18 [unique_strings addObject: new];
19 return new;
20 }
```

Пример кода из файла `unique.m`

Если вы создаете новый неизменяемый класс, вам, возможно, потребуется реализовать в его конструкторе нечто подобное описанному выше. В таком случае при построении объектов придется принести быстроедействие в небольшую жертву эффективному использованию памяти, но это будет означать, что сравнение указателей можно всегда выполнить с целью определить равенство объектов. И в некоторых случаях это может в целом дать существенный выигрыш в быстроедействии.

Подклассификация коллекций

```
25 - (NSUInteger)count
26 {
27     return [realArray count];
28 }
29 - (id)objectAtIndex: (NSUInteger)anIndex
30 {
31     return
32         [realArray objectAtIndex: anIndex];
33 }
```

Пример кода из файла `checkedArray.m`

Все стандартные коллекции являются *кластерами классов*. Это означает, что создание нового их подкласса — дело совсем непростое. Суперкласс коллекции является аб-

страктным, а конкретные его подклассы — закрытыми, и поэтому они не подлежат подклассификации.

В каждой коллекции документируются методы, которые должны быть реализованы в подклассе. И сделать это проще всего посредством делегирования: нужно лишь получить в виде переменной экземпляра объект того же типа, что и коллекция, которая подлежит подклассификации. Именно такой подход применяется в приведенном выше примере кода из файла `checkedArray.m`.

Класс `TypedArray` является подклассом, производным от класса `NSMutableArray`. Переменная экземпляра `realArray` в этом классе относится к типу `NSMutableArray`. Она создается в специально выделенном для этого класса инициализаторе и, подобно любому другому коду, создающему объект класса `NSMutableArray`, получает экземпляр одного из конкретных подклассов, производных от этого класса. Другая переменная экземпляра относится к типу `Class` и служит для контроля типов объектов, вводимых в массив.

```
10 - (id)initWithType: (Class)aClass
11 {
12     if (nil == (self = [super init]))
13     {
14         return nil;
15     }
16     type = aClass;
17     realArray = [NSMutableArray new];
18     return self;
19 }
20 - (void)dealloc
21 {
22     [realArray release];
23     [super dealloc];
24 }
```

Пример кода из файла `checkedArray.m`

В отличие от стандартных классов массивов, которые являются разнотипными, в данном случае каждый вводимый в массив объект должен быть экземпляром (или подклассом) того класса, который передается инициализатору. Если попытаться ввести другие объекты в массив, рассматриваемый здесь код завершится аварийно без всяких предупреждений, но его нетрудно видоизменить, чтобы сгенерировать соответствующее исключение.

В классе `TypedArray` реализован целый ряд простых методов. Например, в методе `-removeObjectAtIndex:` просто передается то же самое сообщение заключенному в оболочку массиву. А в остальных методах, в том числе и в методе `-insertObject:atIndex:`, сначала производится контроль типов, а затем вызывается настоящий массив.

В данном примере реализованы все семь простых методов, объявленных в классе `NSMutableArray`. Это означает, что все остальные методы объявленные в классе `NSArray` или `NSMutableArray`, будут действовать корректно.

```
34 - (void)insertObject: (id)anObject
35         atIndex: (NSUInteger)anIndex
36 {
37     if (![anObject isKindOfClass: type])
38     {
39         return;
40     }
41     [realArray insertObject: anObject
42                 atIndex: anIndex];
43 }
44 - (void)removeObjectAtIndex: (NSUInteger)idx
45 {
46     [realArray removeObjectAtIndex: idx];
47 }
```

Пример кода из файла `checkedArray.m`

На заметку

В языке C++ простые методы обычно объявляются в суперклассе только как виртуальные (`virtual`), и компилятор препятствует непосредственному получению экземпляра суперкласса или любого подкласса, в котором эти методы не реализованы. А в Objective-C соответствующие языковые средства отсутствуют.

Более общим основанием для подклассификации коллекции служит потребность в реализации другой структуры данных для базового представления. Так, если требуется ввести объекты на одном конце массива, а затем удалить их на другом, то для реализации такого массива может потребоваться кольцевой буфер изменяемой длины.

В таком случае следует по-прежнему переопределить те же самые методы, но на этот раз заменить их реализацией на основе собственного механизма сохранения. Это даст возможность и далее использовать свой подкласс там, где и суперкласс, включая его передачу другим библиотекам, но применяя собственный механизм сохранения.

Сохранение объектов Objective-C в коллекциях C++

```
30 std::tr1::unordered_map<id, __weak id,  
    object_hash<id>, object_equal<id> > map;  
31 @autoreleasepool {  
32     NSString *str = [[NSString alloc]  
        initWithUTF8String: argv[0]];  
33     map[@"arg0"] = str;  
34     NSLog(@"%@", map[@"arg0"]);  
35     str = nil;  
36 }  
37 NSLog(@"%@", map[@"arg0"]);
```

Пример кода из файла `collection.mm`

Одним из исторически сложившихся недостатков Objective-C++ являются трудности, проявляющиеся в со-

вместимости типов коллекций. Коллекции Objective-C предназначены для хранения объектов Objective-C и поэтому непригодны для хранения структур C++. В этом отношении коллекции C++ проявляют большую гибкость. Они реализованы как шаблоны и поэтому могут теоретически хранить объекты любого типа.

На практике сохранение объектов Objective-C в коллекциях C++ усложняется применяемым механизмом управления памятью. Объекты Objective-C ожидают получить сообщение `-retain` отовсюду, где требуется сохранить владеющую ссылку на них. А коллекции C++, не написанные с учетом Objective-C, не делают этого, и поэтому пользоваться ими в Objective-C++ следует очень аккуратно и внимательно.

Дело значительно упрощается благодаря автоматическому подсчету ссылок. Когда для типа объекта Objective-C используется шаблон C++, он расширяется, и каждое присваивание в шаблоне автоматически снабжается вставляемыми вокруг него соответствующими вызовами механизма подсчета ссылок. Такое расширение распространяется даже на слабые ссылки в тех системах, где они поддерживаются. В примере кода, приведенном в начале этого раздела, показано, как воспользоваться типом отображения C++, чтобы реализовать нечто эквивалентное классу `NSMapTable` со строгими указателями на нестрогие указатели.

Коллекции `unordered_map` в C++ требуется поддержка со стороны двух классов: одного — для определения равенства объектов, а другого — для хеширования объекта. Для реализации этих классов достаточно заключить в оболочку соответствующие сообщения Objective-C.

Нечто подобное можно сделать и без поддержки нестрогих указателей, используя в C++ класс интеллектуального указателя, посылающий сообщение `-retain` при построении объекта и сообщение `-release` при его разрушении. Но такой способ, скорее всего, окажется медленным, поскольку в нем не используются преимущества оптимизации, обеспе-

чиваемые механизмом ARC. В типичной операции с коллекцией C++ указатель может присваиваться несколько раз, что, в свою очередь, вызовет целый ряд операций сохранения и освобождения объектов из памяти, и тогда оптимизатор ARC удалит те из них, которые окажутся избыточными. А при использовании класса интеллектуального указателя эти операции вызовут немало отправок сообщений.

```
8  template <typename X>
9  struct object_equal
10 {
11     bool operator()(const X s1, const X s2) const
12     {
13         return (s1==s2) || [(id)s1 isEqual:(id)s2];
14     }
15 };
16
17 template <typename X>
18 struct object_hash
19 {
20     size_t operator()(const X s1) const
21     {
22         return (size_t)[(id)s1 hash];
23     }
24 };
```

Пример кода из файла `collection.mm`

В терминологии C++ это означает, что указатель `__unsafe_unretained id` относится к типу *простых данных* (POD), тогда как указатели `__weak id` и `__strong id` — к типу *непростых данных*. Они неявным образом реализуют перегружаемые операторы присваивания и непростые копирующие конструкторы. Если вам не совсем понятно, что это означает, не отчаивайтесь. Просто имейте в виду, что вы можете воспользоваться механизмом ARC в C++, и все заработает как по мановению волшебной палочки.

Даты и моменты времени

К общей категории времени в Objective-C относятся два понятия: абсолютные моменты времени и временные интервалы. С временными интервалами работать легче. Они представляют собой простые скалярные величины, измеряемые, как правило, в секундах. В частности, тип `NSTimeInterval` служит для хранения временных интервалов. Он определяется, как правило, в виде числового значения с плавающей точкой двойной точности, чего зачастую оказывается более чем достаточно.

В различных средах можно обнаружить и другие способы представления временных интервалов. Характерным тому примером служит среда `UIKit` на платформе `Mac OS X`, где временные интервалы хранятся в виде рациональных чисел. Это дает возможность повторно вводить их без потери точности вследствие ошибок округления чисел с плавающей точкой, что очень важно для работы с носителями информации. Ведь на протяжении всей ленты носителя составные ошибки округления чисел с плавающей точкой могут привести к заметному сдвигу видео- и аудиодорожек относительно друг друга и потере их синхронизации.

Временные интервалы определяют отличие двух абсолютных моментов времени. Дать точное определение абсолютному времени очень трудно. Такие понятия, как часовые пояса и даже разные календари, затрудняют определение абсолютного времени. Большая часть кода, связанного

с моментами времени и датами в среде Foundation, призвана разрешить подобное затруднение.

Обнаружение текущей даты

```
6 NSDate *now = [NSDate date];
7 now =
8 [NSDate dateWithTimeIntervalSince1970: time(NULL)];
```

Пример кода из файла `date.m`

Класс `NSDate` инкапсулирует значение абсолютного времени, которое хранится в двух компонентах. Одним из них является вполне определенный фиксированный момент времени: дата начала эпохи, а другим — временной интервал от начала эпохи.

В классе `NSDate` поддерживаются две стандартные даты начала эпохи, и обе они определены на основе григорианского календаря. Одной из них является дата 1 января 1970 года, называемая `UNIX Epoch`. Эта дата начала эпохи была впервые определена в ОС `UNIX` и впоследствии внедрена в стандарт `ISO C`. Функция `time()` в `C` возвращает количество секунд, отсчитанных от этой даты.

Другая дата начала эпохи была определена в спецификации прикладного интерфейса `OpenStep` и называется *исходной датой*. Эта дата выбрана несколько менее произвольно и обозначает начало тысячелетия с первого дня 2001 года.

Назначение класса `NSDate` отчасти состоит в том, чтобы избавить программирующих на `Objective-C` от необходимости знать, какая именно исходная дата используется. Для сравнения двух дат можно воспользоваться методом `-compare`: или любыми связанными с этим методами безотносительно к той дате начала эпохи, к которой они обращаются внутренним образом.

Если новая дата формируется с помощью метода `+alloc/init` или `+date`, она устанавливается по текущему времени.

Даты в Objective-C представлены изменяемыми объектами, но их зачастую лучше интерпретировать так, как будто они не являются объектами. Для видоизменения дат имеется единственный метод `-addTimeInterval:`. Он выведен из употребления в Mac OS X 10.6 и может вызвать сложности в устаревшем коде, поскольку многие программирующие обычно забывают о возможности видоизменить даты и просто сохраняют их вместо того, чтобы копировать.

Преобразование дат для отображения

```
6 NSDate *now = [NSDate date];
7 NSCalendar *cal = [NSCalendar currentCalendar];
8
9 unsigned int components = NSYearCalendarUnit;
10 NSInteger year =
11     [[cal components: components
12      fromDate: now] year];
13
14 NSString *date = [NSDateFormatter
15     localizedStringFromDate: now
16     dateStyle:
17         NSDateFormatterLongStyle
18     timeStyle:
19         NSDateFormatterNoStyle];
```

Пример кода из файла `calendar.m`

Как упоминалось выше, в классе `NSDate` хранится абсолютное время, но пользователи вашей программы, если они, конечно, не являются роботами, могут выразить неудовольствие, если это время будет представлено им в секундах, начиная с 1 января 1970 года или 1 января 2001 года.

Большая часть западного мира живет по григорианскому календарю, на который она перешла в 1582 году. В первые годы его принятия многие, вероятно, ощутили на себе все неудобства и неточности этого календаря, особенно в таких

католических странах, как Италия, Франция, Португалия, Испания; в 1918 году — в России; а в 1929 году — в Китае.

Это обстоятельство вызывает ряд существенных трудностей в локализации программного обеспечения. Так, если имеется дата за 120 лет до начала эпохи UNIX, то как отобразить ее? Это должна быть дата 1 января 1850 года по григорианскому календарю, но что, если в стране пребывания пользователя прикладной программы григорианский календарь не был принят к этой дате?

Характерным тому примером служит Октябрьская революция в России, которая произошла 7 ноября 1917 году по новому стилю (т.е. григорианскому календарю). Если представить эту дату пользователям из России, она окажется некорректной, поскольку в тот момент в России пользовались юлианским календарем. Именно поэтому совершенный большевиками переворот называется Октябрьской революцией.

В других регионах мира григорианский календарь может вообще не применяться. У большинства пользователей компьютеров, вероятно, имеется некоторый опыт употребления григорианского календаря, но вынуждать остальных пользователей выполнять преобразование дат самостоятельно было бы неверно.

На заметку

Класс `NSDate` служил прежде для представления дат к отображению средствами `OpenStep`, но теперь он выведен из употребления. В отличие от класса `NSDate`, сохраняющего абсолютное время, в классе `NSDate` дата сохранялась по григорианскому календарю.

Для преобразования даты с целью ее отображения, возможно, придется воспользоваться одним из двух классов. Так, если требуется получить отдельные составляющие даты по конкретному календарю, то следует воспользо-

ваться классом `NSCalendar`, а если требуется сформировать строковое значение — классом `NSDateFormatter`.

Класс `NSCalendar` инкапсулирует календарь как способ приведения моментов абсолютного времени к периодам времени, разделенным на годы, месяцы, дни и т.д. С помощью этого класса создается объект типа `NSDateComponents` из объекта типа `NSDate`. Объект типа `NSDateComponents` инкапсулирует составляющие даты относительно конкретного календаря.

В этом объекте составляющих даты не хранится календарь, относительно которого она установлена, и поэтому для сравнения двух подобных объектов не определено никакого конкретного способа. И сделано это намеренно. Ведь если сравнивать даты, то придется сравнивать моменты времени, используя объекты типа `NSDate`, а затем локализуя их для отображения. Это существенное изменение по сравнению с прежним классом `NSCalendarDate`, который был подклассом, производным от класса `NSDate`, а следовательно, определял способ сравнения дат.

Если имеются составляющие даты, то для ее формирования с целью отображения можно было бы воспользоваться методом `+stringWithFormat:` из класса `NSString`, но это не самое удачное решение. Так, если сформировать дату 1/2/2012, то пользователи из США, привыкшие оперировать датами в формате со смешанным порядком следования байтов, интерпретируют эту дату как 2 января, тогда как в остальном мире, где даты представлены в формате с прямым порядком следования байтов, она будет воспринята как 1 февраля.

Подобное затруднение автоматически разрешается в классе `NSDateFormatter`, в котором имеется удобный метод `+localizedStringFromDate:dateStyle:timeStyle:`, возвращающий символьную строку, содержащую локализованные дату и время. Второй и третий аргументы этого

метода определяют тот уровень детализации, который требуется для представления составляющих даты и времени соответственно. Этот уровень детализации может быть указан в самых широких пределах: от полного запрета отображать дату к очень краткому числовому ее представлению и вплоть до самого подробного описания, включая полные названия месяца и дня.

Расчет истекшего времени

```
6 NSDate *start = [NSDate date];
7 sleep(1);
8 NSTimeInterval elapsed =
9     0 - [start timeIntervalSinceNow];
```

Пример кода из файла `elapsed.m`

Для обнаружения отличий в двух объектах даты достаточно отправить одному из них сообщение `-timeIntervalSinceDate:`, указав другой объект в качестве аргумента вызываемого метода. С другой стороны, давность даты можно определить, отправив ее объекту сообщение `-timeIntervalSinceNow`. В итоге будет получено отрицательное числовое значение, если дата относится к прошлому. Поэтому если требуется выяснить, сколько времени прошло с того момента, когда эта дата была сформирована с помощью метода `+date` или `+new`, полученное числовое значение следует вычесть из нуля.

Это не самый быстрый способ расчета истекшего времени. А в результате системного вызова `gettimeofday()` будет возвращено системное время в структуре. Это позволяет избежать затрат на создание объекта и отправку сообщения, но в то же время требует некоторых затрат на обработку самой структуры. В отдельных полях этой структуры хранится количество секунд и микросекунд системного време-

ни, поэтому операция вычитания требует больших затрат, чем сравнение двух скалярных величин.

Извлечение дат из символьных строк

```
6 NSDate *isoDate =
7     [NSDate dateWithString:
8         @"1982-06-15 06:10:00 +0000"];
9
10 NSLocale *gb = [[NSLocale alloc]
11     initWithLocaleIdentifier:@"en_GB"];
12 NSDateFormatter *formatter =
13     [NSDateFormatter new];
14 [formatter setLocale: gb];
15 [formatter setDateStyle:
16     NSDateFormatterShortStyle];
17 [formatter setTimeStyle:
18     NSDateFormatterNoStyle];
19 NSDate *britishDate =
20     [formatter dateFromString:
21         @"15/06/1982"];
```

Пример кода из файла `stringdate.m`

В классе `NSDate` имеется конструктор, позволяющий формировать даты из символьных строк, отформатированных по стандарту ISO 8601. Это удобно для синтаксического анализа и извлечения дат из файлов, используемых для обмена, но не совсем удобно для обработки вводимых пользователем данных.

Если требуется сформировать дату из локализованной символьной строки, то нужно обратиться к классу `NSDateFormatter`. Этот класс инкапсулирует описание форматов дат и способен выполнять преобразования объектов дат в строковые объекты, и наоборот.

При создании средства форматирования дат с помощью этого класса можно задать региональные параметры яв-

ным образом или предоставить пользователю возможность самому выбрать текущие региональные параметры. После этого можно воспользоваться одним из образцовых форматов даты и времени для заданных региональных параметров или указать такой формат явным образом при вызове метода `-setDateFormat:`. Этот метод принимает шаблонную строку в качестве своего аргумента, но пользоваться им все же не рекомендуется, поскольку в нем не учитываются региональные стандарты.

И наконец, имеется еще одна возможность для извлечения дат из символьных строк. Если требуется прочитать даты в каком-нибудь специальном формате, то с этой целью можно воспользоваться классом `NSScanner`, чтобы извлечь отдельные составляющие даты из символьной строки, а затем построить объект типа `NSDateComponents` и передать его объекту типа `NSCalendar` для формирования даты. Подробнее о классе `NSScanner` см. в главе 5.

Получение событий от таймера

```
6 - (void)periodic: (NSTimer*)theTimer
7 {
8     NSLog(@"Timer fired");
9 }
10 - (void)start
11 {
12     SEL sel = @selector(periodic:);
13     [NSTimer scheduledTimerWithTimeInterval: 1
14                          target: self
15                          selector: sel
16                          userInfo: nil
17                          repeats: YES];
18     [[NSRunLoop currentRunLoop] run];
19 }
```

Пример кода из файла `timer.m`

Класс `NSTimer` инкапсулирует источник событий, формирующий события в отдельные моменты времени. По принципу своего действия он аналогичен системному вызову `alarm()`, хотя и позволяет установить сразу несколько таймеров.

Как правило, таймеры создаются с помощью метода, представленного в примере кода в начале этого раздела. Этот метод создает и устанавливает новый таймер в течение одной операции.

По истечении установленного промежутка времени таймер запускается и посылает сообщение объекту. Определить это сообщение проще всего, передав селектор, адресат и словарь при создании таймера с помощью данного метода. Селектор должен быть предназначен для метода, принимающего единственный аргумент: экземпляр класса `NSTimer`. А для извлечения словаря можно отправить таймеру сообщение `-userInfo`.

У рассматриваемого здесь класса имеется еще один конструктор, принимающий объект типа `NSInvocation` в качестве своего аргумента. Но пользоваться им сложнее, поскольку он допускает отправку любых сообщений с произвольными аргументами.

Объект таймера будет автоматически освобожден из памяти, если создать его описанным выше способом, но он будет удержан объектом *цикла исполнения*. Таймерами можно пользоваться только вместе с объектом класса `NSRunLoop`. Объект цикла исполнения сначала ожидает события от ядра операционной системы, например, от таймеров или данных, становящихся доступными по дескрипторам файлов, а затем посылает сообщения, реагируя на эти события.

В среде `AppKit` или `UIKit` цикл исполнения используется автоматически. В обоих классах, `NSApplication` и `UIApplication`, цикл исполнения организуется внутрен-

ним образом. С помощью цикла исполнения эти классы регистрируют события, чтобы получать уведомления о событиях, наступающих в пользовательском интерфейсе, от сервера отображения и передавать их в окна и т.д.

Если вы разрабатываете приложение, запускаемое из командной строки или сервера, то можете по-прежнему пользоваться классом `NSRunLoop`. Он входит в состав среды Foundation, но его объект придется создать вручную и запустить на исполнение. Это делается очень просто, как показано в последней строке кода метода `-start` в упомянутом выше примере.

Работа со списками свойств

Списки свойств представляют собой способ сохранения структурированных данных. В списках свойств можно хранить стандартные классы коллекций (массивы и словари) том числе символьные строки, числа, даты и т.д.

Списки свойств подразумеваются как абстрактные. Они не связаны с конкретным языком программирования или представлением. Для упорядочения в последовательной форме (или так называемой сериализации) списков свойств в настоящее время определены три формата. Самым старым считается формат списков свойств в прикладном интерфейсе OpenStep. Это очень компактный, удобочитаемый формат.

К сожалению, в Mac OS X нельзя записывать данные в списки свойств старого формата. Их можно прочитать, но в спецификации не определено, как сохранять в них даты или другие элементы данных, и поэтому записать их в подобные списки не удастся. Формат OpenStep списков свойств был расширен в проекте GNUstep и позволяет сохранять в них все, что поддерживается в более новых форматах.

В Mac OS X компания Apple внедрила формат XML и двоичный формат списков свойств. Формат XML оказался невероятно многословным, но он обладает тем преимуществом, что может быть подвергнут синтаксическому анализу другими совместимыми с XML инструментальными средствами и встроен в другие XML-документы.

Двоичный формат списков свойств очень быстро подвергается синтаксическому анализу и довольно компактный. Но поскольку он двоичный, то неудобочитаем.

В некоторых библиотеках других языков программирования имеются средства для обработки списков свойств. Так, библиотека WINGS в диспетчере окон WindowMaker позволяет читать и записывать в них данные, как, впрочем, и библиотека, входящая в состав ОС NetBSD. В среде Core Foundation компании Apple и ее открытой библиотеке CFLite также можно пользоваться списками свойств, но из кода C.

Сохранение коллекций в списках свойств

```
6 NSArray *array = [NSArray arrayWithObjects:
7   @"array", @"containing", @"string",
8   @"objects", nil];
9 [array writeToFile: @"example.plist" atomically: NO];
10 NSMutableArray *cycle = [array mutableCopy];
11 [cycle addObject: cycle];
12 [cycle writeToFile: @"failure.plist" atomically: NO];
```

Пример кода из файла writeplist.m

Классы массивов и словарей реализуют метод `-writeToFile:atomically:`. Это позволяет очень просто вывести содержимое коллекции в файл, используя выбираемый по умолчанию формат списка свойств.

Подобные методы не делают ничего сверхъестественного. Они будут действовать только в том случае, если в коллекциях не содержатся данные ни одного из тех типов, которые нельзя сохранить в списке свойств.

Если передать упомянутому выше методу логическое значение YES в качестве второго параметра, он обеспечит постоянство и согласованность представления данных на диске. Этот метод сначала запишет список свойств во временный файл, а затем переименует этот файл по завершении записи.

При выполнении кода из примера, приведенного в начале этого раздела, будет сформирован файл `example.plist`, содержащий массив в качестве корневого элемента с четырьмя символьными строками в нем.

```
1 <?xml version="1.0" encoding="UTF8"?>
2 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
   EN" "http://www.apple.com/DTDs/PropertyList
   -1.0.dtd">
3 <plist version="1.0">
4 <array>
5   <string>array</string>
6   <string>containing</string>
7   <string>string</string>
8   <string>objects</string>
9 </array>
10 </plist>
```

Пример списка свойств из файла `example.plist`

В текущих версиях Mac OS X подобным способом списки свойств используются в формате XML. Как следует из приведенного выше примера, формат XML списков свойств оказывается слишком многословным. В старом формате OpenStep списков свойств массивы заключались в квадратные скобки, а символьные строки — в кавычки, и поэтому списки свойств получались компактными менее чем наполовину. Если же преобразовать список свойств из приведенного выше примера в двоичный формат, то размер его файла сократится с 294 до 82 байтов.

Для такого компактного списка свойств сокращение занимаемого им места на диске уже не имеет особого значения. Списки свойств в обоих упомянутых выше форматах, скорее всего, влезут в одной единице размещения (так называемом кластере) на диске, и поэтому они займут одинаковое пространство на диске. А для синтаксического анализа списков свойств в обоих форматах не потребуется много времени.

Но этого нельзя сказать о более крупных списках свойств. Например, в браузере Safari предыстория посещений веб-страниц сохраняется в двоичном списке свойств. На момент написания этой книги файл такого списка на моем компьютере занимал 15 Мбайт. Если преобразовать его в формат XML, размер его файла возрастет до 34 Мбайт. В связи с этим увеличиваются также издержки вычислительных ресурсов (времени ЦП и объема оперативной памяти) на синтаксический анализ крупных списков свойств.

К сожалению, в классах коллекций не поддерживается вывод коллекций в списки свойств ни в одном из форматов, кроме XML. Поэтому приходится обращаться к классу `NSPropertyListSerialization`, рассматриваемому далее в этой главе.

Все форматы списков свойств имеют иерархическую структуру. Они не поддерживают запись циклических структур данных. Второй массив в приведенном выше примере сохранения коллекции в списке свойств содержит указатель на самого себя. Поведение кода при записи такого массива в список свойств не определено. В Mac OS X, например, в файле списка свойств сохраняются только те элементы, которые находятся перед рекурсивным указателем, а все следующие за ним элементы просто игнорируются.

Чтение данных из списков свойств

```
6 NSArray *array =
7 [NSArray arrayWithContentsOfFile:
8 @"example.plist"];
9 NSData *data = [NSData dataWithContentsOfFile:
10 @"example.plist"];
11 NSMutableArray *mutable =
12 [NSPropertyListSerialization
13 propertyListFromData: data
14 mutabilityOption:
15 NSPropertyListMutableContainersAndLeaves
16 format: NULL
```



```
17         errorDescription: NULL];  
18     NSAssert([mutable isKindOfClass:  
19         [NSMutableArray class]],  
20         @"Should have read a mutable array");  
21     [[mutable objectAtIndex: 0]  
22     appendString: @"suffix"];
```

Пример кода из файла `readplist.m`

Данные из списка свойств могут быть прочитаны двумя способами. Прежде всего, можно воспользоваться обратным вариантом рассмотренного выше метода сохранения данных в списке свойств. В обоих классах коллекций, поддерживающих сериализацию списков свойств, имеется метод инициализации `-initWithContentsOfFile:`, а также соответствующий конструктор.

На первый взгляд, реализовать такой способ совсем не трудно, но по ходу дела возникает ряд осложнений. И наиболее существенное из них состоит в том, что в списках свойств не сохраняются никакие сведения об изменчивости объектов. Например, объекты типа `NSString` и `NSMutableString` сохраняются совершенно одинаково.

При чтении этих объектов обратно из списка свойств возникает вопрос: какой же из них будет получен на самом деле? Так, если прочитать из списка свойств массив, отправив сообщение `+arrayWithContentsOfFile:` объекту типа `NSArray`, то в конечном итоге будет получен неизменяемый массив. А если отправить то же самое сообщение объекту типа `NSMutableArray`, то будет получен изменяемый массив.

Такой способ пригоден для чтения внешнего элемента из списка свойств, но что, если требуется прочитать объекты, находящиеся в массиве? В обоих случаях будут получены массивы, заполненные неизменяемыми объектами. Но ведь это может оказаться совсем не тем, что нужно.

Если требуется прочитать изменяемые объекты обратно из списка свойств, для этой цели следует воспользоваться

классом `NSPropertyListSerialization`. В этом классе организуется чтение и запись объектов в списки свойств и обеспечивается более скрупулезный контроль над этими процессами, чем в различных классах коллекций.

На заметку

В самой свежей документации на версию Objective-C от компании Apple можно обнаружить, что метод, вызываемый для объекта типа `NSPropertyListSerialization` в примере кода, приведенном в начале этого раздела, отмечен как не рекомендуемый в будущем к употреблению. Дело в том, что он был внедрен в то время, когда в методах версии Objective-C от компании Apple начали применяться параметры `error:`, но прежде, чем был внедрен класс `NSError`. К сожалению, метод, рекомендуемый в качестве замены, пока еще не обеспечивает аналогичные функциональные возможности.

Организуя чтение объектов из списка свойств средствами этого класса, можно передать в качестве параметра режим изменчивости объектов. Для этого имеются три возможности. По умолчанию выбирается режим, делающий все объекты неизменяемыми. В примере кода, приведенном в начале этого раздела, выбран совершенно другой режим, делающий все объекты изменяемыми. На это указывает суффикс, присоединяемый к одной из символьных строк в массиве.

Третий режим представляет собой нечто среднее между двумя предыдущими. В этом режиме контейнерные объекты становятся изменяемыми, а остальные объекты (символьные строки, числа, даты и т.д.) — неизменяемыми. Этот режим оказывается удобным в том случае, если требуется видоизменить древовидную структуру списка свойств, но не элементы на ее вершинах, т.е. листьях. Если объекты читаются из списка свойств подобным способом, то класс сериализации этого списка сообщает его формат.

Преобразование форматов списков свойств

```
7 NSString *file =
8   [NSString stringWithUTF8String: argv[1]];
9 NSData *data = [NSData dataWithContentsOfFile:
10  file];
11 NSPropertyListFormat fmt;
12 id plist = [NSPropertyListSerialization
13  propertyListWithData: data
14                  options: 0
15                  format: &fmt
16                  error: NULL];
17 if (fmt == NSPropertyListBinaryFormat_v1_0)
18 {
19     return 0;
20 }
21 data = [NSPropertyListSerialization
22  dataWithPropertyList: plist
23                  format: NSPropertyListBinaryFormat_v1_0
24                  options: 0
25                  error: NULL];
26
27 [data writeToFile: file atomically: NO];
```

Пример кода из файла: `makebinaryplist.m`

В Mac OS X нельзя записывать данные в списки свойств формата OpenStep, но все-таки можно воспользоваться двумя другими форматами. Как правило, для сохранения закрытых данных следует пользоваться двоичным форматом, а для сохранения всех остальных данных, которые пользователю, возможно, потребуется отредактировать другими инструментальными средствами, — форматом XML.

Впрочем, эти правила не являются раз и навсегда установленными. Назначение формата списка свойств состоит в том, чтобы преобразовать его в другое представление без потерь. Поэтому и рекомендуется пользоваться преимущественно двоичным форматом. Ведь он более компактный, обеспечивает ускоренный синтаксический анализ, а также возможность

для пользователя преобразовать список свойств в более удобный формат, если его потребуется отредактировать.

Для выбора формата списка свойств при его записи в файл служит класс `NSPropertyListSerialization`. Этот класс любезно уведомляет также о текущем формате списка свойств при его чтении.

На заметку

Методы, применяемые в примере кода, приведенном в начале этого раздела, были внедрены в Mac OS X 10.6. Если вы пользуетесь более старой версией Mac OS X, то метод чтения из списка свойств можете заменить рассмотренным в предыдущем разделе, а метод записи в список свойств — соответствующим методом из раздела, предшествовавшего предыдущему.

Нередко списки свойств приходится сохранять в двоичном формате ради повышения эффективности. Если же пользователям прикладной программы потребуется загрузить файлы списков свойств в текстовый редактор для последующего редактирования, они могут воспользоваться утилитой `plutil`, чтобы преобразовать списки свойств в формат XML. Но они могут, конечно, забыть преобразовать их обратно в двоичный формат, и поэтому в следующий раз прикладная программа прочитает их медленно.

Если же видоизменение списка свойств не предполагается при каждом запуске прикладной программы на выполнение, то в этот момент целесообразно сначала проверить, является ли его формат по-прежнему двоичным, а затем перезаписать его в этом формате, если он окажется в другом формате.

Применение формата JSON

```
22 NSData *jsonData =  
23     [jsonDataUsingEncoding::NSUTF8StringEncoding];  
24 NSError *e;  
25 id object =
```

```
26 [NSJSONSerialization JSONObjectWithData: jsonData
27         options: 0
28         error: &e];
29 NSAssert(nil == e, @"Failed to parse JSON");
30 NSData *data =
31     [NSJSONSerialization dataWithJSONObject: object
32         options: NSJSONWritingPrettyPrinted
33         error: &e];
34 NSAssert(nil == e, @"Failed to export JSON");
```

Пример кода из файла json.m

За последние несколько лет широкое распространение получил еще один формат, очень похожий на старый формат OpenStep списков свойств. Это формат *JSON* (JavaScript Object Notation — Представление объектов JavaScript). Он является подмножеством упорядоченных в двоичной форме (сериализованных) объектов JavaScript, хранящих данные, но не код.

В последние версии среды Сосоа включен класс, смоделированный на основе класса `NSPropertyListSerialization` для загрузки и сохранения данных в формате JSON. Такому способу сохранения и загрузки данных присуще больше ограничений, чем кодировке списков свойств. В частности, числа или символьные строки можно хранить только в листьях древовидной структуры, а все коллекции должны быть массивами или словарями.

Как правило, пользоваться форматом JSON для приложений, функционирующих только в среде Objective-C, не приходится. Этот формат оказывается удобным, главным образом, для обеспечения стыкуемости программных средств. На многих платформах имеются библиотеки и языки программирования, способные читать и записывать данные в формате JSON, а для большинства веб-приложений он является собственным форматом. Если данные записываются в формате JSON, то в коде JavaScript веб-страницы на стороне клиента нетрудно преобразовать в нужную форму. Справедливо и обратное: можно сформировать данные

в формате JSON на веб-странице и затем воспользоваться ими в коде на Objective-C.

В примере кода, приведенном в начале этого раздела, показано, каким образом осуществляется преобразование данных в формат JSON, и наоборот. В соответствии со стандартом JSON данные должны быть представлены в некоторой кодировке уникода. Класс сериализации способен автоматически распознать эту кодировку, поскольку два первых символа в потоке данных формата JSON всегда представлены в коде ASCII. Это дает возможность обнаруживать многобайтовые кодировки, размещая на месте первых четырех байтов пустые байты NULL, если метка порядка следования байтов отсутствует.

В рассматриваемом здесь примере осуществляется структурная распечатка программы в формате JSON, что полезно, главным образом, для отладки. Выводимый на печать исходный код программы автоматически форматируется с отступами для удобства его чтения. Для этого требуется немного больше времени ЦП и объема оперативной памяти, и поэтому передавать данные в таком формате по сети или применять их в другой программе нецелесообразно, но имеет смысл для правки данных вручную.

Сохранение пользовательских настроек по умолчанию

```
6  NSUserDefaults *def =
7  [NSUserDefaults standardUserDefaults];
8  id persistentString =
9  [def stringWithKey:@"example"];
10 NSLog(@"Old value: %@", persistentString);
11 if (argc > 1)
12 {
13     NSString *new =
14     [NSString stringWithUTF8String:
15     argv[1]];
```

```
16 [def setObject: new
17     forKey: @"example"];
18 [def synchronize];
19 }
```

Пример кода из файла defaults.m

Система пользовательских настроек по умолчанию считается одним из самых крупных потребителей списков свойств. Если заглянуть в каталог `~/Library/Preferences`, то в нем можно обнаружить немало файлов со списками свойств. В этих файлах хранятся глобальные параметры настройки системы текущего пользователя. Соответствующий глобальный каталог располагается в каталоге `/Library/Preferences`.

Для каждого приложения выделяется своя *область настроек по умолчанию*, которая, как правило, обозначается строкой в обратном порядке по отношению к форме записи доменных имен DNS (например, `com.apple.TextEdit`). Доступ к данным из этой области может быть осуществлен средствами класса `NSUserDefaults`.

Система пользовательских настроек по умолчанию может читать данные из самых разных источников. К числу самых полезных источников относится командная строка. Пользовательские настройки можно указать парами “ключ–значение” в качестве аргументов командной строки любого приложения для Mac OS X, что очень удобно для целей отладки.

Система пользовательских настроек по умолчанию доступна в виде единого словаря с несколькими служебными методами. В общем, объект пользовательских настроек по умолчанию можно себе представить как постоянный словарь.

На заметку

Систему пользовательских настроек по умолчанию можно сравнить с реестром Windows, но с учетом нескольких важных отличий. В обоих поддерживается несколько уровней ключей (пользовательских, систем-

ных и сетевых), но все они скрыты от программиста в системе пользовательских настроек по умолчанию. И хотя оба имеют форму древовидной структуры, реестр представляет собой единую базу данных, тогда как система пользовательских настроек по умолчанию хранит данные в отдельных файлах для каждой области. Благодаря этому упрощается процесс внесения изменений в пользовательские настройки по умолчанию инструментальными средствами независимых производителей. А кроме того, система пользовательских настроек по умолчанию масштабируется лучше, поскольку она не требует загрузки больше двух файлов в оперативную память одновременно. К числу недостатков такой системы относится то обстоятельство, что операции с настройками по умолчанию не носят характер транзакций. Внесение изменений в одной и той же области одновременно из двух программ не определено.

В отличие от словаря, у объекта пользовательских настроек по умолчанию имеется несколько служебных методов для доступа к необъектным типам данных и контроля объектных типов. В примере кода, приведенном в начале этого раздела, этому объекту отправляется сообщение `-stringForKey:`. При этом в оболочку заключается метод `-objectForKey:`, но гарантируется, что возвращаемое значение окажется строковым.

Имеются и другие служебные методы, в том числе `-floatForKey:` и `-setFloat:ForKey:`, принимающие примитивные типы C и заключающие их в оболочку экземпляров класса `NSNumber` перед сохранением в системе пользовательских настроек по умолчанию.

Код из примера, приведенного в начале этого раздела, завершается отправкой сообщения `-synchronize` объекту настроек по умолчанию, но зачастую в этом нет особой необходимости. Ведь объект настроек по умолчанию периодически синхронизируется с хранилищем на диске. Но в данном примере цикл исполнения не применяется, и поэтому таймеры не действуют, а выполнение кода завершается сразу же после внесения изменений в настройки по умолчанию. Следовательно, внесенные изменения не будут вообще зафиксированы на диске без отправки этого сообщения.

Если выполнить код из рассматриваемого здесь примера, то можно обнаружить, что начальное значение равно `nil`, поскольку в настройках по умолчанию пока еще ничего не хранится. Значение, передаваемое короткой программе в качестве аргумента, сохраняется в настройках по умолчанию и выбирается автоматически при последующем ее запуске на выполнение.

Это положение можно продемонстрировать и средствами класса `NSArgumentDomain`. Для настроек по умолчанию имеется несколько источников, загружаемых по порядку и перезаписываемых предыдущие. Последними и наиболее высокоприоритетными настройками по умолчанию являются аргументы. Так, если указать аргумент `-example` в командной строке, то в настройках по умолчанию будет задан ключ `@example` для любого значения, передаваемого в качестве аргумента.

Но это не заменяет постоянное значение, а лишь переопределяет его при данном конкретном запуске программы на выполнение. В этом можно убедиться, если выполнить программу еще раз, но без указания упомянутого выше аргумента в командной строке. В итоге будет использовано старое значение, установленное два вызова программы назад. Оно было зафиксировано на диске при третьем запуске программы на выполнение, но не использовалось при четвертом ее запуске, поскольку аргумент был заменен. А при пятом запуске без аргумента `-example` это значение появится снова. Отдельные области настроек по умолчанию могут быть просмотрены средствами класса `NSUserDefaults`, хотя потребность в этом возникает редко.

```
1 $ ./a.out persistent
2 a.out[72216:903] old value: (nil)
3 $ ./a.out new
4 a.out[72216:903] old value: persistent
5 $ ./a.out example arg store
6 a.out[72362:903] old value: arg
```

```

7 $ ./a.out example arg
8 a.out[72363:903] old value: arg
9 $ ./a.out new
10 a.out[72363:903] old value: store
11 $ plutil convertxml1 \
12 ~/Library/Preferences/a.out.plist
13 $ cat ~/Library/Preferences/a.out.plist
14 <?xml version="1.0" encoding="UTF8"?>
15 <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//
    EN" "http://www.apple.com/DTDs/PropertyList
    -1.0.dtd">
16 <plist version="1.0">
17 <dict>
18   <key>example</key>
19   <string>new</string>
20 </dict>
21 </plist>

```

Результат выполнения кода из файла defaults.m

При разработке инструментального средства, работающего в режиме командной строки, настройки по умолчанию оказываются очень удобным средством для синтаксического анализа аргументов командной строки. Исходные значения аргументов командной строки можно хранить в системе пользовательских настроек по умолчанию, предоставив пользователю возможность переопределять их автоматически, не прибегая к синтаксическому анализу кода.

Сохранение произвольных объектов в пользовательских настройках по умолчанию

```

6 UserDefaults *def =
7   [NSUserDefaults standardUserDefaults];
8 NSData *serialized = [def objectForKey:@"color"];
9
10 id favoriteColor =
11 [NSKeyedUnarchiver unarchiveObjectWithData:

```

```
12     serialized];
13
14     if (nil == favoriteColor)
15     {
16         favoriteColor = [NSColor blackColor];
17         serialized = [NSKeyedArchiver
18             archivedDataWithRootObject: favoriteColor];
19         [def setObject: serialized
20             forKey: @"color"];
21         [def synchronize];
22     }
```

Пример кода из файла: `colordefault.m`

Очень часто в настройках по умолчанию требуется сохранить нечто, отличающееся от данных основных типов, поддерживаемых в среде Foundation. Характерным тому примером служат цвета. Если предоставить пользователям возможность настраивать каким-то образом интерфейс прикладной программы, то, как правило, придется сохранять выбранные ими цвета в промежутках между последовательными вызовами программы. И сделать это, на первый взгляд, совсем не трудно: достаточно сохранить объект типа `NSColor` в настройках по умолчанию.

В самом деле, это было бы нетрудно, если бы в списках свойств и расширяющих их пользовательских настройках по умолчанию поддерживались объекты типа `NSColor`. К сожалению, такая поддержка отсутствует. Но в то же время в пользовательских настройках по умолчанию можно хранить экземпляры класса `NSData`. И если иметь в своем распоряжении какой-нибудь механизм преобразования объекта в данные, то этот объект можно сохранить в пользовательских настройках по умолчанию.

Для этой цели, к счастью, имеется общий механизм, поддерживаемый большинством наиболее распространенных объектов. Любой объект, реализующий протокол

NSCoding, может быть подвергнут сериализации и десериализации средствами класса NSCoder.

В отличие от списков свойств, механизм, реализуемый в классе NSCoder, поддерживает произвольные объекты и циклы сохранения. К сожалению, для этого требуется реализовать в объекте два метода из протокола NSCodering, что подходит не для всех, но все же для большинства объектов. Но благодаря этому нетрудно организовать сохранение объектов в настройках по умолчанию.

Если операцию сохранения цветов приходится выполнять часто, то нетрудно создать *категорию*, вводящую в классNSUserDefaults методы `-colorForKey:` и `-setColor:forKey:`. Эти методы могут принимать экземпляр класса NSColor в качестве своего параметра, подвергать его сериализации с помощью кодера и затем восстанавливать в исходном виде.

В качестве более общего решения можно ввести метод `-setEncodedObject:forKey:`, принимающий в качестве своего аргумента указатель `id<NSCoding>` и сохраняющий его с помощью кода из примера, приведенного в начале этого раздела.

Для сохранения в настройках по умолчанию объекта, не поддерживающего протокол NSCodering, имеются две возможности. Во-первых, можно ввести в него поддержку протокола NSCodering. С этой целью придется реализовать методы `-encodeWithCoder:` и `-initWithCoder:` в отдельной категории.

И во-вторых, можно выполнить преобразование другого рода. У большинства объектов имеются методы `-stringValue` и `-initWithString:`. С их помощью можно сохранить объекты в настройках по умолчанию, предварительно преобразовав их в символьные строки, а при чтении из настроек по умолчанию — создав новые объекты из этих символьных строк.

Взаимодействие с окружением

Язык Objective-C обычно применяется на UNIX-подобных платформах, откуда он наследует понятие окружения. В это понятие входит ряд других понятий, в том числе рабочие каталоги, переменные окружения и т.д.

В традиционном окружении Objective-C значения этих переменных предполагалось получать точно таким же образом, как и в программе на C. Но в прикладном интерфейсе OpenStep компании Sun и NeXT внедрили класс `NSProcessInfo`, обеспечивавший более высокий уровень абстракции.

Побудительной причиной для этого отчасти послужило то обстоятельство, что программы с прикладным интерфейсом OpenStep предполагалось выполнять на платформах OPENSTEP, Solaris и Windows NT. Все эти платформы мало чем отличались друг от друга в воплощении понятия окружения, и поэтому для сокрытия их отличий потребовалась некоторая абстракция.

Получение переменных окружения

```
3 int main(int argc, char **argv, char **envp)
4 {
5     // Способ, применяемый в C
6     while (*envp)
```

```
7 {
8     printf("%s\n", *envp);
9     envp++;
10 }
11 printf("working Directory: %s\n", getenv("PWD"));
12
13 // Способ, применяемый в Objective-C
14 [NSAutoreleasePool new];
15 NSDictionary *env =
16     [[NSProcessInfo processInfo] environment];
17 for (NSString *key in env)
18 {
19     NSLog(@"%@ = %@", key, [env objectForKey: key]);
20 }
21 NSLog(@"working Directory: %@\n",
22     [env objectForKey: @"PWD"]);
23 return 0;
24 }
```

Пример кода из файла env.m

Доступ к переменным окружения в Mac OS X можно получить тремя способами. Два из них пригодны для любой платформы, а третий — только для некоторых платформ.

Как и в Windows и большинстве систем UNIX, в Mac OS X сведения об окружении передаются процессам в качестве третьего аргумента функции `main()`. Они представляют собой массив символьных строк C, завершающийся пустым значением `NULL`. Но по стандартам C и POSIX передача этого третьего аргумента не требуется, и поэтому полагаться на него в переносимом коде не стоит.

При запуске программы на выполнение загрузчик произведет загрузку всех требующихся общих библиотек и затем перейдет непосредственно к специальной функции, обычно называемой `_start`, из стандартной библиотеки C. Из этой функции, в свою очередь, вызывается функция `main()`, по прежнему сохраняется указатель на переменные окружения.

Для доступа к этим переменным в памяти можно вызвать стандартную в С функцию `getenv()`. С другой стороны, получить доступ к переменным окружения можно, отправив сообщение `-environment` одиночному объекту сведений о процессе. В итоге будет возвращен словарь, содержащий сведения об окружении.

На заметку

В коде Objective-C нельзя установить переменные окружения. Внешение изменений в них обычно производится в коде С перед вызовом одной из функций из семейства `exec()`. Но такой способ оказывается пригодным только для UNIX-подобных платформ, а не для таких систем, как Windows или Symbian. Впрочем, можно составить новый словарь, который будет использоваться в качестве окружения для вновь порожденного процесса, отправив сообщение `-setEnvironment:` объекту типа `NSTask`.

Извлечение аргументов из командной строки

```
3 int main(int argc, char **argv)
4 {
5     [NSAutoreleasePool new];
6     NSUserDefaults *d =
7         [NSUserDefaults standardUserDefaults];
8     NSString *file = [d stringWithKey:@"file"];
9     if (nil == file)
10    {
11        fprintf(stderr,
12            "USAGE: %s file {filename}", argv[0]);
13        return 1;
14    }
15    NSLog(@"Using file %@", file);
16    return 0;
17 }
```

Пример кода из файла `args.m`

В стандарте C не предусматривается порядок получения аргументов программой, помимо функции `main()`. В стандартной библиотеке C проекта GNU предоставляются некоторые расширения для доступа к аргументам после запуска программы на выполнение, а на большинстве UNIX-подобных платформ их можно прочитать из каталога `/proc` файловой системы, хотя получить их переносимым способом из кода C не так-то просто.

В Objective-C для получения аргументов можно, как всегда, отправить сообщение `-arguments` объекту типа `NSProcessInfo` и в конечном итоге получить объект типа `NSArray`, содержащий символьные строки Objective-C со всеми аргументами. Эти строки аргументов можно затем обработать как угодно.

Но такой способ доступа к аргументам не совсем обычен. Простейший способ извлечения и синтаксического анализа аргументов из командной строки состоит в применении объекта типа `NSArgumentDomain` в объекте типа `NSUserDefaults`, как было показано в предыдущей главе. При запуске на выполнение программы на Objective-C все аргументы читаются из командной строки и вводятся в словарь, используемый в системе пользовательских настроек по умолчанию.

Подобным способом очень просто предоставить ряд аргументов командной строки наряду с (настраиваемыми) значениями по умолчанию. Такой подход вообще исключает непосредственное обращение с аргументами. Нужно лишь запросить значение по соответствующему ключу в системе пользовательских настроек по умолчанию, чтобы получить в ответ данные из места их постоянного хранения или же из командной строки.

В среде EtoileFoundation предоставляется функция `ETGetOptionsDictionary()`, заключающая в оболочку стандартную функцию `getopt()`. Это дает возможность вос-

пользоваться в Objective-C аргументами UNIX-подобной командной строки, что удобно для разработки инструментальных средств в стиле UNIX на Objective-C. В противном случае для получения аргументов из командной строки лучше воспользоваться системой пользовательских настроек по умолчанию.

Доступ к региональным параметрам пользователя

```
6  NSLocale *l =  
7  [NSLocale autoupdatingCurrentLocale];  
8  NSString *language =  
9  [l objectForKey: NSLocaleIdentifier];  
10 NSString *humanReadableLanguage =  
11 [l displayNameForKey: NSLocaleIdentifier  
12    value: language];
```

Пример кода из файла locale.m

Если прикладная программа так или иначе взаимодействует с пользователем непосредственно, ее, вероятно, придется в какой-то момент локализовать. И для этой цели в Objective-C предоставляется немало богатых языковых средств.

Региональные параметры инкапсулируются в объекте класса `NSLocale`. Экземпляры этого класса можно получить несколькими способами. Так, если отправить этому классу сообщение `+systemLocale`, то в итоге будут получены системные региональные параметры. Но на практике так обычно не поступают, поскольку к системным региональным параметрам как к резервным прибегают лишь в том случае, если региональные параметры пользователя не действуют корректно.

Текущие региональные параметры пользователя можно получить двумя способами. С одной стороны, можно отпра-

вить классу `NSLocale` сообщение `+currentLocale` и получить в ответ объект, инкапсулирующий текущие региональные параметры. Если пользователь выберет новые региональные параметры, то будет отправлено уведомление **`NSCurrentLocaleDidChangeNotification`**, и для новых региональных параметров можно будет получить их новый объект.

С другой стороны, можно отправить классу `NSLocale` сообщение `+autoupdatingCurrentLocale` и получить в ответ объект региональных параметров, который будет всегда представлять текущие региональные параметры, даже если они изменяются. Но в этом случае может возникнуть небольшая путаница, поскольку в результате двух последовательных вызовов одного и того же метода с одинаковым аргументом для объекта данного класса могут быть возвращены разные значения.

Очень важно избежать кеширования данных, возвращаемых из объектов региональных параметров, если требуется отслеживать изменения в региональных параметрах по ходу выполнения программы. Так, если данные постоянно выводятся на экран с учетом региональных стандартов, то следует обеспечить контроль уведомлений об изменениях региональных параметров и соответственно обновлять выводимые на экран данные, даже если используется объект автоматически обновляющихся региональных параметров.

Поддержка внезапного завершения процесса

```
11 NSProcessInfo *p =  
12     [NSProcessInfo processInfo];  
13 [p enableSuddenTermination];  
14 // выполнить здесь инициализацию  
15 [p disableSuddenTermination];  
16 // выполнить здесь другие операции,
```

```
17 // которые не могут быть прерваны
18 modifyFiles();
19 [p enableSuddenTermination];
```

Пример кода из файла `suddenDeath.m`

По традиции процесс завершается двумя путями. С одной стороны, процесс может завершиться изящно, достигнув благополучно момента своего окончания, а затем вызвав функцию `exit()` или же достигнув оператора `return` в функции `main()`. А с другой стороны, процесс может завершиться внезапно, получив необрабатываемый сигнал вроде ошибки сегментации, приводящей к аварийному завершению программы.

В обоих случаях ядро операционной системы затребует все свои разделяемые ресурсы, но при внезапном завершении некоторые ресурсы могут остаться в неопределенном состоянии. Характерным тому примером служит файл в процессе его видоизменения.

В работе таких сложных приложений, как процессоры обработки текстов, это вызывало раньше серьезные осложнения. В них открывался файл документа для записи, начиналась его правка, а затем происходило аварийное завершение, тогда как файл оставался в таком состоянии, из которого очень трудно было восстановить его содержимое.

В качестве выхода из подобного затруднительного положения можно воспользоваться атомарными средствами записи для большинства объектов из среды Foundation. Этими средствами осуществляется запись во временный файл, который затем переименовывается. В итоге запись завершается успешно или неудачно, но полностью, а не наполовину.

В Mac OS X 10.6 компания Apple внедрила механизм *внезапного завершения*. В его основу положено следующее теоретическое положение: большинство программ затрачивают немало времени при своем завершении на выполнение деструкторов, уничтожающих объекты, что не дает ни-

какого эффекта, кроме освобождения памяти, которая все равно будет затребована операционной системой. При внезапном завершении процесс может уведомить, что у него отсутствуют не сохраненные данные. В таком случае операционной системе ничто не мешает внезапно уничтожить этот процесс, вместо того чтобы запрашивать его изящное завершение.

Это особенно удобно для выполнения длительных фоновых заданий, простаивающих немало времени. Они будут зачастую откачиваться из памяти ядром операционной системы. Если же запросить изящное их завершение, то большую часть кода и данных придется подкачать страницами в память для выполнения необходимых операций завершения. А при внезапном завершении страницы, находящиеся в файле подкачки, будут просто помечены как доступные для повторного использования.

Поддержка внезапного завершения реализуется в прикладной программе очень просто. С этой целью для объекта типа `NSProcessInfo` следует вызвать два метода: по одному с каждой стороны действия, которое нельзя прервать.

На заметку

Если вы пользуетесь средой `AppKit` с инфраструктурой `NSDocument`, то вам вообще ничего не нужно делать. Ваша прикладная программа будет помечена как неподдерживающая внезапное завершение, если в любых документах останутся несохраненные данные.

Эти методы называются `-enableSuddenTermination` и `-disableSuddenTermination`, но эти имена не совсем соответствуют их назначению. На самом деле они не активизируют и не прекращают внезапное завершение, а только видоизменяют счетчик. Когда же счетчик достигает нуля, прикладная программа помечается как уничтожаемая.

Это означает, что их можно без особого труда использовать в разных потоках или даже вкладывать друг в друга. Исходно в счетчике устанавливается единица, и поэтому объекту сведений о процессе следует явным образом опра- вить сообщение `-enableSuddenTermination`. Но вместо этого при разработке прикладной программы можно устано- вить соответствующий ключ в файле со списком свойств па- кета приложения.

В отличие от инструментальных средств, работающих в режиме командной строки, приложения Socoa, как прави- ло, представляют собой распределенные пакеты в виде ка- талогов, которые должны рассматриваться как файлы гра- фическими диспетчерами файлов. На платформе NeXT они очень эффективно использовались для хранения двоичных файлов одного и того же приложения, предназначавшего- ся для разных операционных систем. Благодаря этому на файловом сервере можно было хранить одно приложения для разных операционных систем, включая OPENSTEP, Solaris и даже Windows. Среди прочего в таком пакете со- держится список свойств (в файле `info.plist`), описываю- щий ряд атрибутов приложения, в том числе имя его глав- ного класса, ресурсы, загружаемые при запуске на выпол- нение, и т.д. К числу этих атрибутов относится исходное состояние внезапного завершения.

В Mac OS X 10.7 и iOS внезапному завершению отведена очень важная роль. В этих операционных системах прило- жения должны завершаться внезапно, когда используемая ими память почти исчерпана. При этом предполагается, что приложения должны быть способны перезапустить- ся, не теряя своего состояния, доступного для пользовате- ля. Так, в Mac OS X 10.7 предполагается, что каждому окну приложения удастся сохранить свое состояние и автомати- чески перезагрузиться.

Доступ к значениям по ключам

Немало кода в типичной программе связано с доступом и видоизменением различных аспектов состояния объекта.

Механизм *доступа к значениям по ключам* (KVC) предоставляет абстрактный интерфейс для установки и получения свойств объектов. Он применяется вместе с другим механизмом, называемым *наблюдением за значениями по ключам* (KVO) и предоставляющим абстрактный способ наблюдения за изменениями в конкретном свойстве другого объекта.

Механизм KVC позволяет взаимодействовать с каждым объектом как со словарем. Аналогичный механизм применяется для чтения и записи свойств безотносительно к способу их сохранения. Главное преимущество механизма KVC заключается в том, что он исключает в значительной степени потребность создавать специальные классы контроллеров для сопряжения объектов модели с представлениями.

Как правило, в классе представления требуется получить и устанавливать свойства объекта модели. С помощью механизма KVC это можно сделать на обобщенном, параметризованном уровне. Достаточно предоставить объекту представления ключ или путь к ключу, чтобы автоматически извлечь данные из модели, даже не зная, как она реализована и какой у нее интерфейс, кроме того факта, что она поддерживает механизм KVC.

Доступ к значениям по ключу

```
68 [object setValue: @"42"  
69         forKey: @"integer"];  
70 int value =  
71     [[object valueForKey: @"integer"]  
72         intValue];  
73 NSCAssert(value == 42,  
74     @"value set and retrieved");
```

Пример кода из файла `kvc.m`

Объекты, которыми манипулирует механизм KVC, очень похожи на словари. Сам класс `NSDictionary` совместим с этим механизмом, хотя методы KVC, поддерживаемые в этом классе, обычно вместе с ним не применяются.

Для взаимодействия с KVC-совместимыми объектами обычно применяются следующие два метода: `-valueForKey:` и `-setValue:forKey:`. Метод `-objectForKey:` действует аналогичным образом для извлечения значений из объекта типа `NSDictionary`. Оба метода установки значений по ключу в объекте типа `NSDictionary` действуют по-разному.

Прежде всего следует обратить внимание на то, что если применить метод `-setObject:forKey:` к неизменяемому словарию, то во время компиляции будет получено соответствующее предупреждение, чего не произойдет, если воспользоваться с той же целью методом `-setValue:forKey:`. Этот метод реализуется в классе `NSObject`. Если вызвать его для обращения к неизменяемому словарию во время выполнения, это приведет к сбою, хотя никаких намеков на нечто подобное во время компиляции не делается.

Еще одно существенное отличие состоит в использовании константы `nil`. Если использовать эту константу как значение в методе `-setValue:forKey:`, соответствующий ключ будет удален. А если сделать то же самое в методе `-setObject:forKey:`, то будет сгенерировано исключение.

Обеспечение совместимости с механизмом KVC

```
3 @interface KeyPublic : NSObject @end
4 @implementation KeyPublic @end
5
6 @interface KeyIvar : KeyPublic
7 {
8     int integer;
9 }
10 @end
11 @implementation KeyIvar @end
```

Пример кода из файла kvc.m

Для применения методов KVC практически всегда вызываются их реализации в классе NSObject. В этом случае они попытаются выработать конкретный способ доступа к свойству объекта.

Прежде всего они пытаются обнаружить методы доступа. По соглашению классы Objective-C предоставляют методы `-property` и `-setProperty:` для получения и установки свойств. Если подобные методы имеются в классе, они вызываются. Механизм KVC пытается обнаружить и ряд других методов, в том числе метод `-isProperty` для доступа к логическому свойству.

```
13 @interface KeyAccessor : KeyIvar @end
14 @implementation KeyAccessor
15 - (int)integer
16 {
17     return integer;
18 }
19 - (void)setInteger: (int)aValue
20 {
21     integer = aValue;
22 }
23 @end
```

```

24 @interface KeyProperty : KeyPublic
25 @property int integer;
26 @end
27 @implementation KeyProperty
28 @synthesize integer;
29 @end

```

Пример кода из файла kvc.m

Если эти попытки потерпят неудачу, классу буде отправлено сообщение `+accessInstanceVariablesDirectly`. И если в итоге будет возвращено логическое значение `YES`, то методы KVC попытаются найти переменную экземпляра, совпадающую по имени с искомым свойством, а затем воспользуются ею. Реализация метода, вызываемого при отправке упомянутого выше сообщения, в классе `NSObject` возвращает логическое значение `YES`, и поэтому остается лишь переопределить этот метод, если требуется предотвратить поиск механизмом KVC переменных экземпляра.

И наконец, если и эти попытки потерпят неудачу, то из кода KVC будет вызван один из следующих двух резервных методов, в зависимости от того, устанавливается или же получается свойство: `-valueForUndefinedKey:` и `-setValue:forUndefinedKey:`.

```

31 @interface KeyFallback : KeyPublic
32 {
33     NSMutableDictionary *dict;
34 }
35 @end
36 @implementation KeyFallback
37 - (id)valueForUndefinedKey: (NSString*)akey
38 {
39     return [dict valueForKey: akey];
40 }
41 - (void)setValue: (id)aValue
42     forUndefinedKey: (NSString*)akey
43 {

```

```
44     [dict setValue: aValue  
45           forKey: akey];  
46 }  
47 @end
```

Пример кода из файла `kvc.m`

Эти методы предоставляют последнюю возможность обеспечить поддержку механизма KVC. Если реализовать их, то следует вызвать их реализацию в суперклассе для всех ключей, которые не поддерживаются непосредственно. Если же вызвать их для корневого класса, то будет сгенерировано исключение, указывающее на то, что ключ обработан неправильно.

Если для синтеза методов доступа используются объявляемые свойства, то совместимость с механизмом KVC будет достигнута практически даром. В приведенных выше примерах кода из файла `kvc.m` демонстрируются самые разные способы реализации одного и того же свойства KVC.

В этих примерах используется один открытый класс и ряд закрытых подклассов, в каждом из которых поддерживается данное свойство. Экземплярами всех этих классов можно пользоваться попеременно — по крайней мере, когда требуется сохранить данный ключ. Но еще важнее, что для создания прототипа и последующего перехода к чему-то другому можно воспользоваться такими классами, как `KeyIvar` или `KeyFallback`.

Резервный метод особенно полезен для создания прототипа. Приступая к работе над прототипом, создайте закрытую переменную экземпляра словаря и получите доступ ко всем свойствам объекта с помощью механизма KVC. Определив свойства, которые действительно нужны создаваемому вами классу, можете заменить резервную реализацию рядом методов доступа, а в качестве дополнительного шага заменить вызовы механизма KVC прямыми отправками соответствующих сообщений.

Возможно, вы обратили внимание в рассматриваемом здесь примере на то обстоятельство, что механизм KVC оперирует только объектами, тогда как переменная экземпляра относится к типу `int`. Тем не менее код остается работоспособным, поскольку механизм KVC выполняет *автоупаковку*. Для установки данного свойства методу `-setValue:forKey:` в качестве аргумента передается значение `@12`. Как только обнаружится, что свойство относится к целочисленному типу, код KVC посылает сообщение `-intValue` строковому объекту.

С другой стороны, создается объект типа `NSNumber`, инкапсулирующий значение примитивного типа. Если в переменной экземпляра или методах доступа используются типы объектов, то этот код пропускается. Следует также иметь в виду, что механизм KVC получает доступ только к сведениям о типах во время выполнения, а все типы объектов закодированы в кодировках типов как `"@"`.

Это означает, что механизм KVC можно использовать для установки экземпляра класса `NSString` в качестве значения переменной экземпляра, объявленной как `NSArray`. Если попытаться сделать это, ни механизм KVC, ни компилятор не выдаст никаких ошибок. Но если получатель попытается затем отправить сообщение вроде `-objectAtIndex:` строковому объекту, он получит в ответ исключение. При разработке методов доступа к типам объектов со стороны механизма KVC, возможно, потребуются ввести операторы утверждения, чтобы проверить, относится ли объект к классу, совместимому с данным механизмом.

Представление о путях к ключам

Итак, мы рассмотрели порядок получения и установки значений по отдельному ключу. Это, конечно, очень полез-

но, но недостаточно для демонстрации всей гибкости механизма KVC.

Большую часть того, что нам удалось добиться с помощью механизма KVC, можно было бы легко сделать и без него. Так, если требуется организовать доступ одного объекта к конкретному свойству другого объекта, не прибегая к жесткому кодированию, то для этого достаточно сохранить селектор и затем вызвать его с помощью обычного для Objective-C механизма отправки сообщений.

Такой подход оказывается вполне пригодным в том случае, если свойство хранится в данном объекте, но что, если требуется доступ к свойству косвенным образом? С этой целью можно было бы сохранить массив селекторов и воспользоваться каждым из них, чтобы получить промежуточный объект, но такое решение оказалось бы довольно неуклюжим.

Механизм KVC дает возможность пользоваться *путями к ключам*. Они представляют собой простые символьные строки, содержащие имена ключей, разделенные точками. Если требуется доступ к свойству свойства объекта, это можно сделать в одной операции, используя пути к ключам.

На заметку

Точечное обозначение операции доступа к объявляемым свойствам очень похоже на обозначение путей к ключам в механизме KVC, и поэтому их, к сожалению, легко перепутать, хотя между ними нет никакой связи. Точечное обозначение порождает отправку сообщений непосредственно методам доступа. А методы KVC, принимающие пути к ключам, относятся к более высокому уровню абстракции. И это одна из причин, по которой точечное обозначение вызывает всеобщую неприязнь.

К сожалению, такое сходство сделано намеренно. До обнародования разработчики из компании Apple обозначали объявляемые свой-

ства, пользуясь терминологией KVC, вероятно, имея в виду более тесную связь между обоими упомянутыми выше обозначениями.

Огромное преимущество путей к ключам заключается, в частности, в том, что в обобщенном объекте представления достаточно хранить лишь один указатель на объект модели. Ему не требуются указатели на различные компоненты внутри модели, поскольку он может получить доступ к ним по путям к ключам.

Так, если в представлении требуется отобразить сведения о конкретном лице, то с этой целью можно соединить копии в родословной, используя пути к ключам @"father" и @"mother". В том же самом представлении можно было бы также отобразить связи с прародителями с помощью пути к ключу @"father.mother" и т.д.

Пути к ключам широко применяются в *привязках Cocoa*. Такие привязки представляют собой обобщенные классы контроллеров, связывающие модели с представлениями.

Наблюдение за ключами

```
5 + (void)observeValueForKeyPath:
6     (NSString*)keyPath
7     ofObject: (id)object
8     change: (NSDictionary*)change
9     context: (void*)context
10 {
11     NSLog(@"%@.%@ is now %@", object, keyPath,
12         [change objectForKey: NSKeyValueChangeNewKey]);
13 }
14 + (void)watchChange
15 {
16     NSMutableDictionary *dict =
17         [NSMutableDictionary new];
18     [dict addObserver: self
19         forKeyPath: @"akey"
20         options:
```

```
21     NSKeyValueObservingOptionNew
22         context: NULL];
23
24     [dict setObject: @"set as object"
25         forKey: @"aKey"];
26
27     [dict removeObserver: self
28         forKeyPath: @"aKey"];
```

Пример кода из файла kvo.m

Конечно, иметь в своем распоряжении единый механизм получения и установки ключей удобно, но не так уж и важно. Истинную силу механизму KVC придает механизм KVO, который позволяет контролировать изменения, происходящие в ключах.

В механизме привязок оба механизма, KVC и KVO, используются с целью обойтись в большинстве случаев без контроллеров. Когда представлению требуется видоизменить модель, оно обращается к механизму KVC. А когда изменяется модель, представление получает соответствующее уведомление от механизма KVO и обновляется с целью отразить произошедшие изменения. Сочетая эти тесно связанные друг с другом технологии, можно установить прочные связи между объектами, не прибегая к жесткому связыванию их конструкций.

Регистрация в качестве наблюдателя KVO во многом подобна регистрации для получения уведомлений, как поясняется в главе 16, хотя у нее имеются некоторые отличия. К числу наиболее существенных отличий относится то обстоятельство, что уведомления от механизма KVO всегда отправляются одному и тому же методу. В первом методе из примера кода, приведенного в начале этого раздела, демонстрируется, как это должно выглядеть на практике.

В данном примере используется метод класса, но зачастую для подобных целей применяются методы экземпляра.

Следует, однако, иметь в виду, что классы являются объектами в Objective-C, и поэтому их можно использовать практически везде, где обычно применяются объекты.

Рассматриваемая здесь реализация первого метода довольно проста. Он просто выводит новое значение ключа на экран. А во втором методе демонстрируется, каким образом регистрируется уведомление. В качестве аргумента `change:` в первом, уведомляющем методе указывается словарь, который в данном случае содержит новое значение ключа. Он содержит только новое значение, потому что ничего другого больше не было запрошено.

Параметр `options:` в методе добавления наблюдателя обозначает битовое поле. Это дает возможность запросить новое значение, старое значение или же и то и другое. Кроме того, можно запросить уведомления как до, так и после изменения.

Рассматриваемый здесь механизм наблюдения за значениями по ключам позволяет без особого труда решать такие задачи, как, например, регистрация изменений, сделанных в объекте, что удобно для реализации отмены внесенных изменений. Если объекты вашей модели совместимы с механизмом KVO, вы можете получать уведомления обо всех изменениях, сохранять старое значение и восстанавливать его впоследствии, используя механизм KVO. Подобная логика реализации отмены может быть совершенно не связана с подробностями реализации объектов вашей модели.

Обеспечение совместимости с механизмом KVO

```
34 [self setValue: @"42"  
35     forKey: @"aKey"];  
36 self.aKey = 12;  
37 [self willChangeValueForKey: @"aKey"];
```



```
38 akey = 47;  
39 [self didChangeValueForKey: @"akey"];
```

Пример кода из файла kvo2.m

Механизм KVO, как, впрочем, и все остальные языковые средства Objective-C, не делает ничего чудесного. Если установить ключ с помощью методов доступа KVC, то оправка уведомлений наблюдателям осуществляется довольно просто. Но что произойдет, если установить переменную экземпляра непосредственно? К сожалению, ответ на этот вопрос очень прост: ничего.

Непосредственная установка переменной экземпляра означает лишь сохранение присваиваемого значения в памяти. Определить автоматически момент, когда отдельное значение изменилось в памяти, не так-то просто. Это означает, что непосредственный доступ к переменным экземпляра следует заключить в квадратные скобки, как демонстрируют вызовы двух соответствующих методов во фрагменте кода, приведенном в начале этого раздела.

Подобным образом запускается отправка уведомлений от механизма KVO, если они требуются. Следует, однако, иметь в виду, что эти уведомления не обязательно отправлять, если вызывается метод доступа. В этом и состоит одно из преимуществ позднего связывания, обнаруживаемого в Objective-C.

Код, реализующий механизм KVO, довольно сложный, и в нем применяется ряд более развитых языковых средств Objective-C. Когда регистрируется наблюдатель за значением по ключу, метод установки значения этого ключа заменяется специальным его вариантом, предназначенным для обработки уведомлений.

Механизм KVO, вероятно, относится к тем компонентам среды Foundation, которые на первый взгляд кажутся загадочными. Но на самом деле он не совершает ничего загадочного из того, чего нельзя реализовать в своем коде. Он

не опирается на какие-то недокументированные интерфейсы, модификации компиляторов или аналогичные специальные приемы программирования. Нечто подобное вы можете по желанию реализовать в своем коде. С тем, как это делается, вы можете, например, ознакомиться, обратившись к реализации GNUstep.

В GNUstep и Mac OS X механизм KVO реализован с помощью специального приема, называемого *настройкой указателя isa по адресам*. Этот прием состоит в том, что указатель `isa` на объект настраивается таким образом, чтобы указывать на другой класс после создания объекта. Как правило, данный прием применяется вместе с созданием динамического класса, и поэтому новый класс обычно вводится в качестве листа в древовидную структуру иерархии классов, а указатель `isa` на объект настраивается, чтобы указывать на этот класс.

Правда, такие механизмы, как KVO, специально внедрены в среду Foundation, чтобы избавить вас от необходимости разбираться в том, как они устроены и работают, если, конечно, вы не проявите к этому особый интерес. Но у реализации механизма KVO (и аналогичных средств) с помощью настройки указателя `isa` по адресам имеется один существенный побочный эффект. Он заключается в следующем: не стоит полагаться только на то, что указатель `isa` на объект будет указывать на конкретный класс или останется неизменным.

Если требуется проверить, является ли объект экземпляром конкретного класса, следует воспользоваться методом `-isKindOfClass:` или `-isKindOfClass:.` Для того чтобы получить класс объекта, достаточно отправить ему сообщение `-class`, не ограничиваясь только проверкой указателя `isa`.

Обработка ошибок

Большая часть кода содержит программные ошибки. Это обстоятельство непременно учитывается в грамотно написанном коде, и поэтому в нем организована изящная обработка ошибок. А в идеально написанном коде применяются формальные методы, проверяющие отсутствие программных ошибок, хотя написать код идеально удастся немногим.

В большинстве примеров кода, приведенных в этой книге, предполагается, что программные ошибки вообще не возникнут. Но в связи с малым форматом книги продемонстрировать надлежащую обработку ошибок в каждом примере кода не представляется возможным. Ведь в этом случае код обработки ошибок, не относящийся непосредственно к рассматриваемым примерам, занял бы немало страниц книги, количество которых ограничено.

Большинство ошибок, которые могут быть обнаружены во время выполнения, возникают в функции или методе, вызывающих другие функции или методы с недействительными входными значениями. Наилучший способ обработки таких ошибок в значительной степени определяется возможностями языка программирования. Например, в языке Erlang не поощряется так называемое “безопасное” программирование на модульном уровне. Если модуль находится в неопределенном состоянии, его следует уничтожить и создать новый его вариант. В языке С поощряется проверка достоверности каждого входного значения и последующая проверка каждого возвращаемого значения.

В языке Java обработку ошибок можно отсрочить, используя исключения. А в языках Lisp и Smalltalk предоставляется возможность инспектировать стек при возникновении ошибок и динамически исправлять код, дающий неправильное входное значение.

В этом отношении язык Objective-C занимает промежуточное положение где-то между C и Java. Одни ошибки лучше обрабатывать как можно ближе к моменту их возникновения, тогда как другие могут быть обработаны выдачей сообщений об их причине с последующим выходом из программы.

Еще во времена NeXT в Objective-C применялись такие средства обработки ошибок, как *исключения*, но они действовали лишь как исключения в других современных языках программирования. А поскольку язык Objective-C является надмножеством C, то в нем поддерживаются все формы сообщения об ошибках, доступные в C.

Отличия в обработке исключений во время выполнения

Реализовать обработку исключений языковыми средствами не так-то просто. Генерируя исключение, приходится разворачивать каждый стековый фрейм в промежутке между моментами генерирования и перехвата исключения. Этого можно добиться несколькими способами. Если в языке вроде Smalltalk имеются средства для интроспекции стека, то сделать это будет нетрудно. Достаточно просмотреть в исключении каждый стековый фрейм и развернуть его.

В Java и подобных языках программирования такой способ обычно реализуется возвратом двух значений. В типичной виртуальной машине Java один регистр резервируется для возврата объекта исключения. После каждого вызова виртуальная машина Java проверяет, содержит ли этот регистр нуль, а иначе выполняет условный переход к коду обработки исключений.

Реализовать нечто подобное в Objective-C невозможно, поскольку приходится взаимодействовать с языковыми средствами C. Некоторые стековые фреймы в промежутке между моментами генерирования и перехвата исключения могут относиться к коду C, и поэтому вряд ли можно ожидать, что весь код C в системе будет видоизменен для возврата другого значения в том случае, если будет сгенерировано исключение.

Другой способ представляет собой нечто, подобное *структурной обработке исключений* (SEH), внедренной корпорацией Microsoft, где связный список адресов кода очистки помещается в стек и извлекается из него при входе и выходе из блоков обработки исключений соответственно. Когда генерируется исключение, этот стек используется для поиска и выполнения кода очистки и безусловного перехода к безопышечному месту в стеке. Такой способ обработки исключений можно, конечно, организовать и в Objective-C, но, к сожалению, слишком дорогой ценой. Ведь для этого потребовалось бы выполнять некоторый код при каждом входе и выходе из блока обработки исключений, а затем и дополнительный код при генерировании исключения.

В состав C входит пара функций, при вызове которых можно реализовать нечто, подобное разворачиванию стека. Так, при вызове функции `setjmp()` сохраняется содержимое текущих регистров ЦП, включая стек и указатели на его фреймы, а при вызове функции `longjmp()` они перезагружаются. В итоге стек устанавливается в то состояние, в котором он находился при вызове функции `setjmp()`. Именно таким образом исключения по традиции и реализовывались в Objective-C.

Нетрудно себе представить, что такой способ обработки ошибок был далек от идеала. В нем не поддерживалось выполнение действий по очистке в промежуточных стековых фреймах. Так, если код Objective-C вызывался из кода C++, откуда вызывался снова код Objective-C, то из кода C++

приходилось вызывать функцию `longjmp()`, чтобы предотвратить выполнение в нем деструкторов.

Совместимая с языком C++ обработка исключений была внедрена в динамическую библиотеку GNU еще в 2004 году. Каждая функция, скомпилированная в этом режиме независимо от исходного языка, экспортировала в формате DWARF отладочные данные, описывавшие расположение стекового фрейма и порядок его разворачивания. Такой способ зачастую называется *беззатратной обработкой исключений*, поскольку он не налагает во время выполнения никаких дополнительных затрат на код, в котором обработка исключений не применяется. Если скомпилировать код C в этом режиме, двоичная его форма окажется немного больше, поскольку она будет содержать дополнительные сведения о разворачивании стекового фрейма, но эти сведения не подкачиваются в память, если исключение не генерируется в коде C.

При такой поддержке появилась возможность выполнять код очистки в C++, когда исключение Objective-C генерировалось в стековых фреймах C++, и наоборот. Кроме того, появилась возможность регистрировать код очистки в C, используя расширение `__attribute__((cleanup))` компилятора GCC.

Упомянутые выше изменения были приняты в компании Apple при разработке 64-разрядного варианта динамической библиотеки. Это привело к значительным изменениям характеристик обработки исключений в отношении производительности. В частности, обращение к области памяти для обработки исключений с помощью функций `setjmp()` и `longjmp()` оказалось более затратным, поскольку требовало сохранения всех регистров ЦП, но само исключение генерировалось довольно быстро. Тем не менее подобная обработка исключений была небезопасной, и поэтому в документации рекомендовалось, как правило, преждевременное завершение программы после перехвата исключения. Таким образом, исключениями следовало пользоваться для изящного наполовину преждевременного

завершения программы, если только они не перехватывались очень близко к моменту их генерирования.

При беззатратной обработке исключений для входа в блок, обрабатывающий исключение, не требуется никаких затрат, тогда как генерирование исключения обходится очень дорого и к тому же небезопасно. Это означает, что блоки `@try` можно ввести в код прикладной программы, не замедляя ее выполнение, а генерировать исключения лишь при возникновении совершенно исключительных ситуаций.

Если же требуется соблюсти совместимость с 32-разрядным вариантом той же самой динамической библиотеки от компании Apple, то рекомендуется по-прежнему избегать исключений. Ведь они налагают немалые издержки, если в коде программы часто используются блоки `@try`.

На заметку

Обработывая исключения в старом стиле, нужно соблюдать особую аккуратность, объявляя как `volatile` любые переменные экземпляра, которые должны быть доступны после перехвата исключения. В противном случае они могут быть восстановлены с теми значениями, которые они содержали при вызове функции `setjmp()`.

Генерирование и перехват исключений

```
6 NSArray *array = [NSArray array];
7 @try
8 {
9     [array objectAtIndex: 0];
10 }
11 @catch (NSEException *e)
12 {
13     NSLog(@"Caught exception %@", e);
14 }
```

Пример кода из файла `exception.m`

В устаревшем коде Objective-C можно, скорее всего, обнаружить блоки, начинающиеся с метки `NS_DURING`. Это старые макрокоманды, определявшие порядок обработки исключений в стиле функции `setjmp()`. А в новых версиях среды Foundation они определены в заголовочном файле `NSException.h`, чтобы использовать в них новые ключевые слова для обработки исключений.

На эти старые макрокоманды накладываются два ограничения, хотя они и применяются в качестве оболочек, в которые заключаются новые ключевые слова. Во-первых, они позволяют перехватывать только исключения типа `NSException`. И во-вторых, у них отсутствует эквивалент блока `@finally`, возвращающего код очистки и продолжающего разворачивание стека, хотя этот процесс можно симитировать, сначала перехватив исключение и затем сгенерировав его еще раз.

Пользоваться этими макрокомандами было крайне неудобно. Так, для возврата из блока `NS_DURING` пришлось бы вызывать макрокоманду `NS_VALUEReturn` или `NS_VOIDReturn`, чтобы непременно удалить код очистки из стека обработки исключений.

```
67 #define NS_DURING @try {
68 #define NS_HANDLER } @catch (NSException *
    localException) {
69 #define NS_ENDHANDLER }
70 #define NS_VALUEReturn(v,t) return (v)
71 #define NS_VOIDReturn return
```

Пример кода из файла `NSException.h`

Ныне код обработки исключений в значительно большей степени похож на аналогичный код в Java или C++. Но и на него накладываются ряд ограничений. В частности, генерирование других объектов исключений, кроме `NSException`, допускается, но не поддерживается. В Mac OS X это объясняется тем, что объект исключения генерируется

в том случае, если он скомпилирован вместе с устаревшей динамической библиотекой, а перехватить его можно с помощью старой макрокоманды `NS_HANDLER`, ожидающей получения экземпляра типа `NSException`. Средствами динамической библиотеки GCC, но не `GNUstep`, вполне возможно сгенерировать экземпляр того класса, который не был надлежащим образом зарегистрирован в такой библиотеке, что, скорее всего, приведет к аварийному сбою разворачивающей библиотеки. А генерирование необъектных типов данных вообще не поддерживается.

Самый лучший способ сгенерировать исключение по-прежнему состоит в том, чтобы отправить сообщение `-raise` экземпляру класса `NSException`. Для генерирования произвольного объекта исключения можно также воспользоваться ключевым словом `@throw`, но не стоит особенно удивляться, если такой способ окажется неработоспособным. Если же вы пользуетесь динамической библиотекой GNU или современной версией динамической библиотеки от компании Apple, то данный способ может оказаться вполне работоспособным, хотя соответствующий код не подлежит полноценному тестированию.

Применение объектов исключений

```
5 [NSException raise: NSGenericException  
6   format: @"This is an example"];
```

Пример кода из файла `throwexception.m`

Класс `NSException` служит для реализации стека буферов с вызовами функции `longjmp()` и безусловного перехода к вершине стека, когда в этом возникает потребность. Теперь разворачивание стека полностью выполняется средствами динамических и разворачивающих библиотек, тем не менее класс `NSException` по-прежнему служит для инкапсуляции исключений.

Исключения можно разделить на три составляющие: имя, описание причины и словарь пользовательских данных. Имя представляет собой однозначный идентификатор типа исключения в виде символьной строки. Но чаще всего для этой цели служит символическая константа вроде `NSRangeException`, что дает возможность сравнивать указатели по имени исключения при его перехвате.

Описание причины исключения обычно представлено в удобочитаемом виде. Как правило, оно используется лишь в целях отладки, и поэтому его локализация необязательна. Ведь важнее понять, что именно произошло, а не какие действия пользователей к этому привели. И наконец, словарь пользовательских данных относится к типу `NSDictionary` и содержит ряд пар “ключ–значение”. Содержимое этого словаря полностью определяется кодом, генерирующим исключение, хотя в большинстве методов, генерирующих исключения, документируется то, что должно быть помещено в словарь пользовательских данных.

С помощью подобного словаря можно передавать произвольные сведения всякий раз, когда перехватывается исключение. А поскольку это словарь, то в него можно впоследствии добавить ряд дополнительных пар “ключ–значение”, не нарушая двоичную совместимость с существующим кодом.

Это один из тех случаев, когда принято пользоваться неизменяемым словарем типа `NSDictionary`, а не `NSMutableDictionary`, поскольку в коде, перехватывающем исключение, не предполагается внесение изменений в словарь. Для создания неизменяемого словаря однократного применения зачастую используется конструктор `+dictionaryWithValuesAndKeys:` класса `NSDictionary`, если словарь присоединяется к генерируемому исключению.

В последних версиях Mac OS X объект исключения содержит также дополнительные сведения, полезные для целей отладки. Так, если отправить ему сообщение `-callStackReturnAddresses`, то в ответ будет получен массив

типа `NSArray`, содержащий указатели типа `NSNumber` на адреса возврата из всех функций в стеке. Это не очень удобно, хотя в версии `Mac OS X 10.6` был внедрен метод `-callStackSymbols`. Он возвращает имена всех функций или методов, сохраненных в стеке при генерировании исключения.

Этот последний метод применяется в обработчике необрабатываемых исключений, который вызывается всякий раз, когда разворачивание исключения не перехватывается, чтобы вывести трассировку стека прежде чем выполнение программы будет преждевременно завершено. Вместо этого обработчика можете воспользоваться функцией `NSSetUncaughtExceptionHandler()`, чтобы реализовать свой алгоритм обработки необрабатываемых исключений. Если у вас имеется механизм сообщения об ошибках, то возможность обратной трассировки событий, начиная с необработанного исключения, может оказаться очень полезной.

Применение единой модели исключений

```
5 try {  
6     @throw @"An exception!";  
7 } catch (int a) {  
8     NSLog(@"Caught: %d", a);  
9 } catch (id exception) {  
10     NSLog(@"Caught: %@", exception);  
11 }
```

Пример кода из файла `tryCatch.mm`

В первых реализациях `Objective-C++` модели исключений `Objective-C` и `C++` были совершенно разными и несовместимыми. В частности, исключения `C++` можно было генерировать в блоке `throw` и перехватывать в блоке `catch`, а исключения `Objective-C++` — генерировать в операторе `@throw` и перехватывать в операторе `@catch`. Оба способа обработки исключений можно было попеременно использовать в коде прикладной программы, но не в той же самой функции и не для обработки одного и того же исключения.

В последующих реализациях Objective-C++ нашла применение единая модель исключений. Оператор `catch` с типом данных Objective-C позволяет перехватывать объекты, генерируемые в операторе `@throw`. Аналогично оператор `throw` с типом данных Objective-C позволяет генерировать исключение, которое может быть перехвачено в операторе `@catch`.

Простейший случай такой обработки исключений демонстрируется в примере кода, приведенном в начале этого раздела. Но старые компиляторы Objective-C++ откажутся компилировать такой код, поэтому нужно проверить, поддерживается ли единая модель исключений в целевой среде исполнения.

Управление памятью при обработке исключений

```
12 id pool = [NSAutoreleasePool new];
13 id array = [NSArray new];
14 @try
15 {
16     throw(array);
17 }
18 @catch (id e)
19 {
20     NSLog(@"Caught exception %@", e);
21     e = [e retain];
22     [pool drain];
23     [e autorelease];
24     pool = nil;
25     @throw;
26 }
27 @finally
28 {
29     [array release];
30     [pool drain];
31 }
```

Пример кода из файла `releaseexcept.m`

При обработке исключений управление памятью немного усложняется. Если в начале метода создается новый объект, а в конце объект освобождается из памяти, то во внимание необходимо принимать, что исключения представляют собой тонко замаскированные нелокальные операторы безусловного перехода и что возможны самые разные пути возврата из метода. Каждый вызов функции или каждая отправка сообщения становится потенциальным путем возврата.

В качестве простого выхода из этого положения достаточно ввести блок `@finally` в конце блока `@try`. Код в блоке `@finally` будет выполнен в любом случае: будь то по завершении блока `@try` или в результате генерирования исключения.

Еще одно затруднение, связанное с обработкой исключений, может возникнуть при повторном генерировании объекта исключения. Как правило, объекты исключений освобождаются из памяти автоматически. Это означает, что при управлении памятью вручную они будут удалены вместе с автоматически освобождаемым пулом.

В большей части кода это не должно вызвать никаких затруднений. Автоматически освобождаемый пул, чаще всего используемый к коду прикладных программ, сохраняется на время действия цикла исполнения. Но иногда требуется создать собственный автоматически освобождаемый пул.

Если перенести удаление автоматически освобождаемого пула в блок `@finally`, что кажется, на первый взгляд, очевидным решением, то возникнет следующее затруднение: объект исключения будет удален при разворачивании текущего стекового фрейма, а следующий вверх по стеку фрейм, скорее всего, разрушится. В итоге произойдет разыменование повисшего указателя при попытке выявить тип исключения. Во избежание этого следует непременно сохранить объект исключения, прежде чем удалять

автоматически освобождаемый пул, а в дальнейшем освободить его автоматически из памяти. Это одна из тех причин, по которым вообще не рекомендуется генерировать никаких исключений, кроме объекта типа `NSException`. Если же последовать рекомендациям, которые даются по этому поводу в документации на версию Objective-C от компании Apple, то в конечном итоге получится код, способный правильно обрабатывать только объекты исключений типа `NSException` и преждевременно разрушающий объекты любого другого типа.

На заметку

В Java исключениями можно пользоваться и как локальным оператором безусловного перехода, генерируя и перехватывая исключение в одном и том же методе, хотя это и довольно необычный способ сообщить ветвь кода виртуальной машине. Нечто подобное возможно и в Objective-C, хотя и совершенно неэффективно. В этом случае оператор `@throw()` вызывает функцию `objc_exception_throw()`, а та, в свою очередь, — функцию из разворачивающей библиотеки.

В итоге данные формата DWARF будут прочитаны для стекового фрейма и произойдет вызов идентифицирующей функции Objective-C. Эта функция далее прочитает дополнительные данные формата DWARF, сравнит класс исключения с классом в блоке перехвата исключения, а затем установит указатель команд на адрес оператора перехвата.

Полученный результат будет преобразован виртуальной машиной Java в безусловный переход к блоку перехвата исключения. Если последовать этому шаблону в Objective-C, то необходимо воспользоваться доступным в C оператором `goto`. Но делать этого, как правило, не рекомендуется, а лучше поискать иное решение, не нарушающее принципы структурного программирования.

Для того чтобы правильно реализовать обработку исключений подобным способом, потребуется универсальный оператор `@catch` с указателем типа `id`. Этот оператор будет перехватывать все исключения Objective-C. Напомним, что

исключения C++ не будут по-прежнему перехватываться в блоке этого оператора, поэтому для полной очистки памяти потребуется блок `@finally`, если разворачивание осуществляется с помощью чужеродного исключения. В блоке `@catch` можно установить значение указателя на автоматически освобождаемый пул равным константе `nil`, чтобы исключить освобождение этого пула дважды.

Передача делегатов ошибок

```
23 NSFileManager *fm = [NSFileManager defaultManager];
24 [fm copyPath: source
25    toPath: destination
26    handler: [ErrorHandler new]];
```

Пример кода из файла `errordelegate.m`

В языке Common Lisp имеется понятие *возобновляемых исключений*, в отличие от разворачивающих исключений в Objective-C, Java, C++ и прочих языках программирования. Возобновляемое исключение не разворачивает стек, а выполняет код обработки исключений и затем продолжает выполнение программы.

Прямого аналога возобновляемым исключениям в Objective-C нет, но добиться сходного результата можно, передав методу *делегат ошибки* в качестве аргумента. Подобным образом действовали прежние прикладные интерфейсы API в классе `NSFileHandle`. В качестве последнего аргумента в них использовался объект, которому посылались сообщения, когда выполнение операции наталкивалось на определенные трудности.

Метод, которому передается делегат ошибки, возвращает логическое значение типа `BOOL`, указывающее, следует ли продолжить выполнение операции. Такой способ обработки ошибок можно сочетать с разворачивающими исключениями в коде прикладной программы, генерируя исключение, если делегату ошибки не удалось соблюсти заданные условия.

Я, например, воспользовался этим шаблоном в среде `LanguageKit` для сообщения об ошибках компиляции и соответствующих предупреждениях. Если компилятор обнаружит ошибку, он вызовет метод в делегате ошибки. В стандартной реализации такого способа обработки ошибок на консоль просто выводилось сообщение, а в других реализациях выполнялось преобразование абстрактного синтаксического дерева для устранения ошибки. Если данный метод возвратит логическое значение `YES`, то компилятор вернется назад и попытается проверить код еще раз. А если возвратится логическое значение `NO`, то компилятор сгенерирует исключение, которое будет перехвачено в коде, вызвавшем компилятор.

Этим шаблоном следует, как правило, пользоваться везде, где в ходе выполнения операции может встретиться исправимая ошибка. В частности, им можно воспользоваться при загрузке одного файла, в котором делается ссылка на другой файл. Если один из файлов, на которые делается ссылка, отсутствует, то по умолчанию выполнение кода завершится преждевременно. Но делегат ошибки мог бы предоставить новую ветвь кода для его продолжения, используя какую-нибудь фиксированную трансляцию или выдав пользователю подсказку.

Возврат значений ошибок

```
4 - (BOOL)trySomething;  
5 (id)tryToDoSomethingWithObject: (id)anObject  
6                               error: (NSError**)e;
```

Пример кода из файла `errorReturn.m`

В языке C принято возвращать нулевое значение при успешном завершении кода, а иначе — ненулевое значение, обозначающее состояние ошибки. В Objective-C такой шаблон нашел менее широкое распространение, хотя в этом языке относительно часто возвращается логическое

значение типа `BOOL`, обозначающее, насколько успешным оказалось выполнение операции.

Зачастую возврат значений ошибок осуществляется методами, видоизменяющими тем или иным способом получатель сообщений. При вызове такого метода он должен внести какие-то изменения в объект, но не получить значение, и поэтому возвращаемое значение нельзя использовать в каких-нибудь других целях, кроме обработки ошибок.

В качестве примера реализации такого шаблона может служить метод `-setPropertyForKey:` из класса `NSStream`. Данный метод пытается установить свойство потока и возвращает логическое значение `YES`, если ему удастся это сделать. Следует, однако, иметь в виду, что в С этот шаблон реализуется иначе, поскольку истинное значение возвращается при неудачном выполнении операции.

Применение класса `NSError`

```
6 NSError *error = nil;
7 NSStringEncoding enc;
8 NSString *str = [NSString
9     stringWithContentsOfFile: @"DoesNotExist"
10        usedEncoding: &enc
11        error: &error];
12 if (nil != error)
13 {
14     NSLog(@"Error: %@",
15         [error localizedDescription]);
16 }
```

Пример кода из файла `nserror.m`

Ряд методов, внедренных компанией Apple в версии Mac OS X 10.2, в качестве аргумента принимают указатель `NSString**`, используемый для возврата второго значения, сообщающего об ошибке. Если переменная, на которую указывает этот указатель, содержит константу `nil` после

вызова метода, выполнение операции считается успешным. В противном случае эта переменная содержит символьную строку, описывающую характер ошибки.

Такому подходу был присущ ряд очевидных ограничений. Конечно, символьная строка вполне пригодна для сообщения пользователю об ошибке, но она не очень полезна для устранения ошибки. Так, если требуется обработать ошибку в коде прикладной программы, символьную строку придется каким-то образом подвергнуть синтаксическому анализу.

Когда компания Apple выпустила браузер Safari версии 10.2.7, то вместе с ним внедрила новый класс `NSError`. Объект этого класса практически всегда возвращается по указателю, передаваемому методу в качестве его последнего аргумента. Этот объект ошибки очень похож на объект исключения. У него имеется имя, называемое *областью*, а также связанный с ним словарь.

Но в отличие от объекта исключения словарь объекта ошибки содержит ряд предопределенных ключей. К их числу относится локализованное описание ошибки, которое можно предоставить пользователю без дополнительной обработки.

В объекты ошибок иногда включается определенная стратегия устранения ошибок. Так, объект, *пытающийся устранить ошибку*, может попробовать сделать это, хотя некоторые ошибки не подлежат устранению. А зачастую ошибка возникает при попытке сделать что-нибудь вполне возможное, но небезопасное. В таком случае об ошибке следует сначала сообщить пользователю, а затем обратиться к объекту, пытающемуся устранить ошибку.

Если у вас имеется некоторый опыт программирования на Common Lisp, то подобный шаблон покажется вам знакомым как особая форма перезапускаемого исключения.

Доступ к файлам и каталогам

Если вы пользуетесь средой AppKit, то у вас имеются две возможности для решения большинства задач, связанных с файловой системой. Во-первых, воспользоваться классом `NSFileManager` из среды Foundation, в котором предоставляется немало методов для манипулирования файлами и каталогами на низком уровне. А во-вторых, воспользоваться классом `NSWorkspace` из среды AppKit.

Целый ряд функций делегируются классом `NSWorkspace` *процессу рабочего пространства*. В старых системах NeXT это была единая программа, тогда как в Mac OS X отдельные функции данного процесса реализуются оконным сервером и сервисной программой Finder.

В силу этого не имеет никакого смысла говорить о применении каких-то вариантов высокого уровня, когда присутствие процесса рабочего пространства не гарантируется, например, в инструментальных средствах, работающих в режиме командной строки, а возможно, и в режиме удаленного соединения, или же серверном коде.

В среде UIKit отсутствует нечто, эквивалентное классу `NSWorkspace`. Если же в качестве целевой используется среда Cocoa Touch, то для манипулирования файлами и каталогами придется воспользоваться прикладными интерфейсами API низкого уровня.

Чтение файла

```
7 NSData *copy =  
8   [NSData dataWithContentsOfFile: file];  
9 NSData *mapped =  
10  [NSData dataWithContentsOfMappedFile: file];  
11 NSData *read = [[NSFileHandle  
12   fileHandleForReadingAtPath: file]  
13   readDataToEndOfFile];
```

Пример кода из файла `readFile.m`

В системах UNIX, как правило, предоставляются два способа доступа к файлу для чтения: с помощью функций `read()` и `mmap()`. А в большинстве других систем (по крайней мере, тех, что работают на аппаратных средствах с *блоком управления памятью*) имеются эквиваленты этих функций. Первая из этих функций копирует поток байтов из файла в буфер памяти, а вторая — отображает считанные из файла данные в области памяти.

В среде Foundation предоставляются два класса, выполняющие приблизительно те же самые функции. В частности, класс `NSData` инкапсулирует область памяти, содержащую нетипизированные данные. Экземпляры этого класса можно создать из файлов самыми разными способами.

Для чтения мелких файлов можно воспользоваться вариантом, заключающим в оболочку вызов метода `read()` и читающим копию данных из файла в память. А для чтения крупных файлов, возможно, придется воспользоваться методом `+dataWithContentsOfMappedFile:`.

Преимущество последнего способа заключается в том, что он обеспечивает очень хорошее взаимодействие с подсистемой виртуальной памяти операционной системы. Если физическая память окажется исчерпанной, то выделять место для подкачки замещающих страниц, содержащих данные из отображаемого файла, не потребуется.

Достаточно сначала отвергнуть их, а затем прочитать снова в следующий раз, когда они будут доступны.

Это очень удобно для работы на платформе с весьма ограниченным объемом оперативной памяти и полным отсутствием виртуальной памяти. Так, мобильные телефоны iPhone до появления модели 3GS имели оперативную память объемом 12 Мбайт, и ни у одного из них не было пространства подкачки. Загрузив в память файл размером 4 Мбайт, пользователь тем самым значительно урезал свободное пространство, доступное для приложения.

С другой стороны, если создать объект отображаемых данных, представляющий читаемый файл, то система прочитает данные фрагментами по 4 Кбайт по мере надобности, а затем вытеснит их из памяти, когда они уже не будут нужны. Если же при создании нового объекта память исчерпается, ядро операционной системы вытеснит отображаемый файл из памяти, чтобы освободить место. Но оно не сможет этого сделать, если файл прочитан, и поэтому приложение будет уничтожено вследствие исчерпания памяти.

Для чтения файла по частям следует воспользоваться классом `NSData`. Этот класс включает в оболочку дескриптор файла и реализует имеющиеся в Objective-C варианты тех же самых основных операций с файлами, которые поддерживаются в стандартной библиотеке C. Все данные могут быть прочитаны из дескриптора файла в течение одной операции, но чаще всего читается следующий доступный бит данных.

А для чтения фиксированного фрагмента данных достаточно отправить дескриптору файла сообщение `-readDataOfLength:`. Большинство других методов из упомянутого выше класса не представляют особого интереса для чтения данных из файла. Они предназначены в основном для взаимодействия с сетевыми сокетами. Мы еще вернемся к этому классу в главе 17.

Перемещение и копирование файлов

```
12 NSFileManager *fm = [NSFileManager new];
13 if (shouldMove)
14 {
15     [fm copyItemAtPath: source
16         toPath: destination
17         error: nil];
18 }
19 else
20 {
21     [fm moveItemAtPath: source
22         toPath: destination
23         error: nil];
24 }
```

Пример кода из файла fileCopy.m

Класс `NSFileManager` инкапсулирует файловую систему в целом и позволяет манипулировать ею. Этот класс делает доступными для программирующих на Objective-C те же самые средства, которые пользователи могут обнаружить в сервисной программе Finder в Mac OS X или Проводнике Windows. До появления версии Mac OS X 10.5 применялись методы, принимавшие объект в качестве своего последнего параметра. Этому объекту затем отправлялись сообщения по ходу выполнения операции копирования или перемещения файла. В этих сообщениях, в частности, запрашивалось, следует ли продолжить операцию после ошибки.

С версии Mac OS X 10.5 в компании Apple решили внести существенные изменения в интерфейс. Теперь в методах применяется новый шаблон сообщения об ошибках, рассматривавшийся в главе 12, и поэтому в качестве последнего аргумента они принимают указатель `NSError**`, по которому возвращается ошибка. А сообщения отправляются теперь делегату объекта.

Это означает, что теперь классом `NSFileManager` следует пользоваться только из одного потока. До появления вер-

сии Mac OS X 10.5 этот класс служил примером реализации *шаблона-одиночки*. Значение, возвращавшееся методом `+defaultManager`, а это был экземпляр одиночного объекта, зачастую сохранялось в кеше. Но теперь код, в котором используется кеширование и который может быть вызван из других потоков, считается небезопасным.

Если вы пишете новый код, непременно создавайте новый экземпляр класса `NSFileManager` всякий раз, когда требуется им воспользоваться, а впоследствии удаляйте его, иначе может возникнуть ситуация, когда сообщения будут отправляться не тому делегату. Если не сделать этого, то в двух потоках может быть параллельно установлен делегат для одиночного диспетчера файлов, который будет получать все уведомления, вместо того, чтобы каждый из них получал только те уведомления, которые его интересуют.

На заметку

Приведенный выше пример служит образчиком непродуманного подхода к проектированию программного обеспечения. Более взвешенное решение состоит в том, чтобы сохранить делегата в *словаре потока*, как поясняется в главе 14. Такое решение следует взять на вооружение в том случае, если требуется внести сходные изменения в одиночный класс. С другой стороны, можно просто сохранить дополнительный параметр для обратного вызова. Но для поддержки новых интерфейсов не следует требовать слишком радикальных изменений в уже существующем коде.

Если вы пользуетесь старыми, вышедшими из употребления методами для обращения к диспетчеру файлов, то потокобезопасным ваш код делает одиночка. А если не вызываете ни один из методов, отправляющих сообщения делегату, то и в этом случае безопаснее пользоваться одиночкой. Но, к сожалению, это означает, что вам, скорее всего, придется создать новый экземпляр класса `NSFileManager` при копировании файла, даже если у вас уже имеется указатель на экземпляр одиночного объекта.

Получение атрибутов файлов

```
6  NSFileManager *fm = [NSFileManager defaultManager];
7  NSDictionary *attrs =
8      [fm attributesOfItemAtPath: @"fileAttributes.m"
9          error: NULL];
10 NSString *fileType = [attrs fileType];
11 NSNumber *creator =
12     [attrs objectForKey: NSFileHFSCreatorCode];
```

Пример кода из файла `fileAttributes.m`

Класс `NSFileManager` заключает в оболочку большинство функций манипулирования файловой системой по стандарту POSIX. В языке C для поиска сведений о файле обычно используется функция `stat()`, которая принимает указатель на структуру в качестве своего аргумента и заполняет ее сведениями о файле.

Такому способу получения сведений о файле присуще следующее очевидное ограничение: в структуру нельзя добавить новые поля, не нарушив весь имеющийся код, в котором она используется. Поэтому в классе `NSFileManager` принят намного более гибкий подход. В частности, метод `-attributesOfItemAtPath:error:` возвращает объект типа `NSDictionary`, в который совсем не трудно добавить дополнительные словарные статьи, соблюдая при этом двоичную совместимость.

В примере кода, приведенном в начале этого раздела, обращает на себя внимание то обстоятельство, что для получения требуемого значения словарю можно отправить сообщение `-fileType`. Вызываемый в итоге метод входит в состав класса `NSDictionary` и возвращает словарную статью, соответствующую ключу типа `NSString`. Имеются и другие аналогичные методы обращения к словарю.

В Mac OS X предоставляются несколько более подробные метаданные файлов, чем на других платформах POSIX. Так, в файловой системе HFS+ хранится код созда-

теля и код типа каждого файла. И хотя эти сведения теперь используются нечасто, тем не менее в классической операционной системе Classic MacOS они служили вместо расширений файлов для определения подходящего приложения, в котором следовало открыть файл.

Манипулирование путями к файлам

```
6 NSString *home = @"~";
7 NSString *full =
8     [home stringByExpandingTildeInPath];
9 NSString *users =
10    [full stringByDeletingLastPathComponent];
11 NSString *file =
12    [users stringByAppendingPathComponent: @"users"];
13 NSString *fileWithExtension =
14    [file stringByAppendingPathExtension: @"db"];
```

Пример кода из файла: path.m

В большей части кода на платформе UNIX манипулирование путями к файлам осуществляется с помощью функций `sscanf()` и `sprintf()`. Это вызывает впоследствии немало затруднений, когда код приходится переносить на такую платформу, как, например, Windows или Symbian, где применяется другое расположение файловой системы и иной разделитель путей к файлам.

Прикладной интерфейс OpenStep был с самого начала разработан переносимым на разные платформы в семействе операционных систем. Так, в классе `NSString` предоставляется ряд методов, предназначенных для манипулирования путями к файлам.

Несмотря на то что эти методы манипулируют символическими строками, они раскрывают более абстрактный интерфейс. Это дает возможность добавлять и удалять составляющие пути как одиночные его элементы, обозначающие файлы или каталоги, а также видоизменять расширения файлов.

В Mac OS X, как и в любой другой системе UNIX, имеется единая корневая папка, или каталог, обозначаемый одной косой чертой. Подкаталоги отделяются от корневого каталога косой чертой, а расширения файлов — точкой от их имен. В то же время в Windows имеется несколько корневых каталогов, а составляющие пути отделяются от них обратной косой чертой, но расширения файлов — по-прежнему точкой. Если методы манипулирования путями к файлам вызываются для объектов типа `NSString`, то код прикладной программы будет выполняться корректно независимо от условных обозначений, принятых в конкретной файловой системе.

К сожалению, аналогичные операции нельзя выполнять над объектами типа `NSMutableString`. Во избежание дополнительных издержек, связанных с созданием большого количества временных объектов, можно воспользоваться методами `+pathwithComponents:` и `pathComponents`, формирующими символьную строку из массива составляющих путей к файлам и создающими такой массив соответственно.

Что касается кода, манипулирующего путями к файлам, то беспокоиться о его эффективности особенно не стоит. Подавляющее большинство операций с файловой системой все равно обойдется намного дороже с точки зрения используемых вычислительных ресурсов, чем создание нескольких временных объектов.

Выяснение факта существования файла или каталога

```
9  NSFileManager *fm =
10  [NSFileManager defaultManager];
11  BOOL isDir;
12  if ([fm fileExistsAtPath: path
13      isDirectory: &isDir])
14  {
15      if (isDir)
```

```
16     printf("Folder exists\n");
17     else
18         printf("File exists\n");
19 }
20 else
21     printf("File does not exist\n");
```

Пример кода из файла `fileExists.m`

Самый простой способ выяснить, существует ли файл, — попытаться сначала получить доступ к нему, а затем проверить наличие ошибок. И хотя это не самый изящный способ, он нередко оказывается вполне пригодным. Так, если требуется обеспечить обратную связь с пользователем в отношении предполагаемой операции с файлом, прежде чем пытаться ее выполнить, то для этой цели потребуется явная проверка.

Именно это делает метод `-fileExistsAtPath:isDirectory:`. Имя этого метода не совсем соответствует его истинному назначению. Он проверяет факт существования файла или каталога по указанному пути и сообщает, присутствует ли он в каталоге.

На заметку

Не пользуйтесь описанным выше механизмом для создания временных файлов. В противном случае возможно появление состояния гонка между проверкой существования файла и его созданием. Вместо этого воспользуйтесь функцией `mkstemp()` из стандартной библиотеки C. В среде EtoileFoundation для этой функции имеется отдельный класс-оболочка, которым вы можете воспользоваться, если не желаете покидать область объектно-ориентированного программирования.

В качестве второго параметра этого метода служит указатель на логическое значение типа `BOOL`. Если вы перешли на Objective-C из C или C++, то найдете такой способ вполне привычным для вас. А тем, кто перешел на Objective-C

из других языков программирования, в том числе из Java, он покажется не совсем обычным.

В языке C, а также в Objective-C как его расширении, поддерживается возврат только одного значения, но из данного метода требуется вернуть два значения, которые обозначают, существует ли искомый элемент (файл или каталог) и имеется ли он в каталоге. Такой возврат можно реализовать несколькими способами. В частности, можно определить перечислимый тип с элементами, обозначающими следующее: ничего не существует, файл существует и каталог существует.

Благодаря этому весь код, в котором используется данный метод, должен бы упроститься, и поэтому нет никаких прямых оснований не поступить именно таким образом. Но дело прояснится, если осознать, что в качестве второго аргумента данному методу можно передать пустое значение NULL. Если сделать это, то в методе не будет выполнена проверка, является ли каталогом объект, указанный по заданному пути. Это относительно небольшая экономия, и поэтому не совсем ясно, стоило ли на нее вообще идти.

Работа с пакетами

```
6 NSBundle *mainBundle =  
7   [NSBundle mainBundle];  
8 NSLog(@"%% links against: %@",  
9   [mainBundle executablePath],  
10  [NSBundle allFrameworks]);
```

Пример кода из файла `bundles.m`

В системе Apple System 7 и более ранних ее версиях на уровне файловой системы поддерживались две разновидности ветвления: ветвление кода и ветвление данных. В частности, ветвление данных использовалось для хранения произвольных ресурсов. Начиная с версии Mac OS X 8.1

компания Apple внедрила файловую систему HFS+, в которой поддерживается произвольное количество ветвлений, благодаря чему файлы, по существу, ничем не отличаются от каталогов в отношении способности содержать произвольное количество потомков. Аналогичные функциональные возможности имеются и в файловой системе NTFS.

В Mac OS X до сих пор поддерживаются ветвления ресурсов и хранение данных в скрытом файле даже в тех файловых системах, где такая поддержка отсутствует, например в UFS и FAT, но применять такие ветвления все же не рекомендуется. Ведь скопировать файл, содержащий ветвления к другим файловым системам, не так-то просто. Так, если попытаться скопировать файл сначала на диск FAT в Mac OS X, а затем с этого же диска в Windows, то скрытый файл не скопируется, а следовательно, данные в ветвлении будут потеряны.

Между тем разработчики из компании NeXT пошли по совсем другому пути. Они задались вопросом: если благодаря ветвлению файлы становятся подобными каталогам, то почему бы не пользоваться вместо них только каталогами? Функции, аналогичные ветвлениям, были реализованы в ОС NeXTSTEP поверх стандартной файловой системы UNIX, только вместо файлов в ней использовались каталоги.

Каталоги, которые должны интерпретироваться как файлы, называются *пакетами*. С точки зрения простейших, низкоуровневых операций с файловой системой они подобны каталогам. Так, если скопировать пакет приложения для Mac OS X на внешний накопитель, подключаемый через интерфейс USB, а затем просмотреть его в Windows, то он будет выглядеть как любой другой каталог. Но если просмотреть его в окне Finder, то он будет выглядеть как отдельный файл.

В большинстве программ на Objective-C чаще всего применяются две разновидности пакетов: интегрированных сред и приложений. Обе разновидности пакетов содержат

некоторый исполняемый код (общую библиотеку или прикладную программу) и ряд необходимых ресурсов.

Конкретное расположение пакета зависит от системы. Так, в GNUstep применяется старое расположение в стиле NeXT, тогда как в Mac OS X — несколько упрощенное расположение. Создать пакеты, имеющие расположение GNUstep и Cocoa, совсем не трудно, причем один и тот же пакет приложения с расширением .app можно переносить из одной системы в другую без перекомпиляции.

Для того чтобы получить главный пакет приложения, достаточно отправить сообщение `+mainBundle` классу `NSBundle`. В этом классе инкапсулируются пакеты, а также отслеживаются все динамически загружаемые пакеты.

Пакеты, естественно, локализируются. Ресурсы, хранящиеся в различных подкаталогах для каждого регионального стандарта, могут быть загружены по требованию. В частности, метод `-pathForResource:ofType:` возвращает локализованный вариант именованного ресурса с указанным расширением в каталоге ресурсов пакета.

На заметку

Инструментальные средства, работающие в режиме командной строки, обычно не хранятся в пакетах. Но у них все же имеется объект главного пакета. Этот объект содержит путь к каталогу, в котором находится главный исполняемый модуль.

Подобным образом в коде прикладной программы можно получить локализованную версию файла ресурсов, хранящегося в пакете приложения или же в любых загружаемых пакетах. Если же код прикладной программы находится в интегрированной среде, то необходимые ресурсы можно получить из пакета этой среды. Хотя сделать это немного сложнее, чем получить ресурсы из главного пакета.

Метод `+bundleForClass:` предоставляет пакет, содержащий код для заданного класса. Этим методом целесообразно пользоваться для загрузки ресурсов в том случае, если в дальнейшем предполагается перенести класс в интегрированную среду.

Поиск файлов в системе

```
11 NSArray *dirs =  
12     NSSearchPathForDirectoriesInDomains(  
13         NSLibraryDirectory,  
14         NSAllDomainsMask,  
15         YES);
```

Пример кода из файла `frameworkLoader.m`

В Mac OS X расположение файловой системы довольно хорошо определено. Можно даже жестко закодировать пути к файлам, и такой код окажется вполне работоспособным. Но если перенести этот код на другие платформы, то дело сразу же усложнится, и весьма существенно.

Именно с таким затруднением программирующие в среде OpenStep сталкивались довольно часто. Например, для перенесения кода между платформами Solaris, OPENSTEP и Windows NT требовалось изменить немало жестко закодированных путей для каждой из этих платформ.

После того как компания Apple приобрела компанию NeXT, она выпустила новую версию OPENSTEP на уровне совместимости с Mac OS X, а также новый пользовательский интерфейс под названием Rhapsody. В этой версии использовалась та же самая иерархия файловой системы, что и в OPENSTEP, но дополненная каталогом /NeXT в корневом каталоге для хранения путей к тем частям иерархии, которые отличались от UNIX-подобных.

В OC Rhapsody DR2 разработчики из компании Apple внесли ряд существенных изменений непосредственно

в расположение каталогов и внедрили новое расположение, которое ныне знакомо всем пользователям Mac OS X. Для упрощения разработки приложений и поддержки среды YellowBox (ныне Cocoa) для Windows, программного продукта, который вскоре после этого прекратил свое существование, была внедрена функция `NSSearchPathForDirectoriesInDomains()`.

Эта функция возвращает массив каталогов для особого применения. При его вызове в качестве первого аргумента указывается требуемая разновидность каталога. Это может быть каталог приложений, каталог библиотек и т.д. А в качестве второго параметра данной функции указывается *область файловой системы*.

Как и в OPENSTEP, иерархия файловой системы в Mac OS X разделяется на ряд областей с приблизительно одинаковым содержимым, но разным применением. Эти области упорядочены, и поэтому в высокоприоритетных областях следует отдавать предпочтение файлам. Ниже перечислены четыре такие области.

- **Пользовательская.** Находится в начальном каталоге пользователя. Расположенные в ней файлы являются закрытыми и находятся под полным контролем пользователя.
- **Локальная.** В этой области содержатся файлы, являющиеся локальными для конкретного компьютера. В Mac OS X к этой области относится, среди прочего, каталог `/Library`, который может быть видоизменен администратором локальной системы.
- **Сетевая.** В этой области можно очень редко что-нибудь обнаружить. Ее вообще невозможно обнаружить на локальном компьютере или тех компьютерах, которые подключены к неоднородной сети. Она служит для хранения каталогов, находящихся под контролем администратора сети.

- **Системная.** В этой области содержатся системные файлы. Как правило, записывать данные в любое место этой области вообще не следует, хотя читать данные из нее можно. Любые изменения, вносимые в этой области, могут быть отменены при последующем обновлении системы пользователем.

В каждой из перечисленных выше областей существуют каталоги далеко не всех типов. Например, в системной области отсутствует каталог документов, поскольку в этом нет никакого смысла.

В качестве последнего аргумента упомянутой выше функции указывается логическое значение, определяющее, следует ли дополнять пути к файлам знаками тильды. Зачастую указывается логическое значение YES.

В примере кода из файла `frameworkLoader.m`, приведенном в начале и в конце этого раздела, наглядно демонстрируется ряд положений, рассмотренных в настоящей главе. В данном примере загружается именованная среда. А поскольку интегрированные среды относятся к категории пакетов, то для их загрузки можно воспользоваться классом `NSBundle`.

В начале этого примера получается список всех каталогов `Library` в системе. Затем они просматриваются по порядку с целью проверить, существует ли в каждом из них каталог с именем среды и расширением `.framework`.

Если что-нибудь обнаруживается под заданным именем, то классу `NSBundle` предписывается загрузить его. С помощью этого кода можно загружать интегрированные среды по требованию во время выполнения, вместо того чтобы связывать их явным образом. С другой стороны, можно видоизменить этот код, чтобы искать требующиеся для приложения подключаемые модули в каталоге, вложенном в каталог библиотеки. И тогда их искать следует в каталоге типа `NSApplicationSupportDirectory`. Если же подключаемые модули komponуются как пакеты, то и в этом случае они загружаются аналогичным образом.

```
11 NSArray *dirs =
12     NSSearchPathForDirectoriesInDomains(
13         NSLibraryDirectory,
14         NSAllDomainsMask,
15         YES);
16 for (NSString *dir in dirs)
17 {
18     NSString *f =
19         [[[dir stringByAppendingPathComponent:
20             @"Frameworks"]
21             stringByAppendingPathComponent: framework]
22             stringByAppendingPathExtension:
23                 @"framework"];
24     // проверить, существует ли среда и каталог
25     BOOL isDir = NO;
26     if ([fm fileExistsAtPath: f
27         isDirectory: &isDir]
28         && isDir)
29     {
30         NSBundle *bundle =
31             [NSBundle bundleWithPath: f];
32         if ([bundle load])
33         {
34             NSLog(@"Loaded bundle %@", f);
35             return YES;
36         }
37     }
38 }
```

Пример кода из файла frameworkLoader.m

Потоки

Первоначально операционная система компании NeXT, как, впрочем, и Mac OS X, была построена на основе ядра Mach. В этом ядре операционной системы с самого начала поддерживались потоки, что было впервые осуществлено в UNIX-подобных системах. В дальнейшем в прикладных интерфейсах API, разработанных для многопоточной обработки по стандарту POSIX, была предоставлена возможность для создания потоков и манипулирования ими на межплатформенном уровне.

В Mac OS X потоки реализованы по стандарту POSIX на основе потоков из ядра Mach, а потоки в Objective-C — на основе потоков POSIX. Вам вряд ли придется пользоваться потоками из ядра Mach непосредственно, поскольку они не полностью согласуются с моделью процессов в UNIX, да и дело это непростое. Но вам, возможно, потребуется обратиться непосредственно к функциям многопоточной обработки из интерфейса POSIX.

Создание потоков

```
10 - (void)processInNewThread
11 {
12     SEL sel = @selector(process:);
13     [NSThread
14         detachNewThreadSelector: sel
15         toTarget: self
```

```
16         withObject: nil];  
17     }
```

Пример кода из файла `thread.m`

Потоки в Objective-C инкапсулированы в объекты типа `NSThread`. Эти объекты заключают в оболочку поток, создаваемый по стандарту POSIX, и предоставляют ряд служебных методов для многопоточной обработки.

Еще в версии Mac OS X 10.5 появились два способа создания новых потоков. Традиционный способ во многом напоминает типичный способ создания потоков по стандарту POSIX, тогда как второй способ рассчитан главным образом на тех, у кого имеется опыт программирования на Java.

При создании нового потока по стандарту POSIX вызывается функция с параметром типа `void*` в новом потоке. А при создании нового потока средствами класса `NSThread` аналогичная операция выполняется путем отправки сообщения объекту этого класса с другим объектом в качестве аргумента вызываемого метода.

Более привычный для программирования на Java способ создания нового потока состоит в подклассификации класса `NSThread` и переопределении метода `-main`. После этого можно создать новый экземпляр объекта потока и отправить ему сообщение `-start`, чтобы запустить поток на исполнение. При создании нового потока для него не выделяется автоматически освобождаемый пул, и поэтому необходимо прежде всего создать такой пул или же новый цикл исполнения нового потока.

Создание нового потока в Objective-C, как, впрочем, и в C, обходится относительно дорого с точки зрения потребляемых вычислительных ресурсов. Ведь процесс должен потребовать у ядра дополнительную память под стек нового потока, создать все необходимые структуры данных пользовательского пространства и ядра для планирования порядка исполнения потока, а также выделить до-

полнительное пространство под локальную память потока. Поэтому рекомендуется не оставлять потоки без внимания, как только они будут созданы.

Для этой цели лучше всего создать новый цикл исполнения в потоке. После этого можно зарегистрировать таймеры для запуска, проконтролировать дескрипторы файлов на предмет новых данных и, наконец, выбрать событийно-ориентированный стиль программирования в новом потоке.

Управление приоритетностью потоков

```
6 double oldPriority =  
7     [NSThread threadPriority];  
8 [NSThread setThreadPriority: 0];  
9 [[NSThread mainThread]  
10  setThreadPriority: 1];
```

Пример кода из файла `threadPriority.m`

В классе `NSThread` методы `threadPriority` и `setThreadPriority` реализуются одновременно как методы класса и экземпляра. При отправке классу `NSThread` сообщений для вызова этих методов они управляют приоритетностью текущего потока. Так, если сообщения отправляются экземпляру данного класса, упомянутые выше методы управляют приоритетностью потока, инкапсулированного в этом классе.

Приоритет потока определяет, сколько времени ЦП требуется потоку и как долго ему придется ждать в промежутке между последовательными моментами, когда ему предоставляется доступ к ЦП. С помощью низкоуровневых прикладных интерфейсов API, построенных по стандарту POSIX, можно также установить правила для планировщика исполнения потоков. Все необходимые для этого функции входят в расширения стандарта POSIX для реального времени, которые также предоставляют очереди сигналов и т.д. А в прикладных интерфейсах API языка Objective-C они пока еще не предоставляются.

Приоритеты, устанавливаемые средствами POSIX API, представляют собой значения с плавающей точкой двойной точности в пределах от 0 до 1. Они могут быть приведены к масштабу целых значений в определенном диапазоне, допустимом для планировщика. Поэтому не следует забывать, что небольшие отличия между ними могут исчезнуть в результате округления.

В спецификации стандарта POSIX для потоков определяются две *области действия конфликтов*. Приоритет потока определяет исходное значение, обозначающее, будет ли потоку выделено время ЦП. Это значение масштабируется, исходя из того, сколько времени уже было выделено потоку и как долго ему пришлось с тех пор ждать. Область действия конфликтов определяет те потоки, которым позволено состязаться за время ЦП. В системной области действия конфликтов потока состязается со всеми остальными потоками, действующими в системе. А в области действия конфликтов процесса он состязается только с теми потоками, которые принадлежат тому же самому процессу.

В Mac OS X реализована только системная область действия конфликтов, а это означает, что высокоприоритетный поток будет заимствовать время ЦП у потоков из других процессов с более низким приоритетом. Так, если установить приоритет потока больше 0,5, то следует иметь в виду, что это может неблагоприятно сказаться на работе остальной части системы.

Тем не менее это не означает полное зависание процесса. В версии Mac OS X 10.6 потоку с наивысшим приоритетом предоставляется чуть больше, чем удвоенный промежуток времени ЦП, по сравнению с тем, что выделяется потоку с наинизшим приоритетом, но только в счетных задачах. А в большинстве других задач приходится ожидать поступления данных с диска. Потоки, находящиеся в заблокированном состоянии, вообще не расходуют время ЦП независимо от их приоритета.

Правила для планирования исполнения потоков в Mac OS X противоречат традиционной модели UNIX, где повышать приоритет процесса разрешается только привилегированному пользователю. Любопытно, что приоритет потока в последних версиях Mac OS X фактически оказывает большее влияние на время ЦП, выделяемое потоку, чем приоритет процесса, которому он принадлежит. Это означает, что того же самого эффекта можно достигнуть, повысив приоритет всех потоков в процессе, хотя приоритет самого процесса увеличить нельзя.

Первоначальный замысел препятствовать обычным пользователям увеличивать приоритет процесса состоял в том, чтобы не дать одному из пользователей возможность несправедливо завладеть всем временем ЦП. Системы Mac OS X, как правило, предназначены для работы на компьютерах с единственным пользователем, и поэтому подобный вопрос в них не возникает. Тем не менее в них имеется все же возможность заимствовать время ЦП у более важных задач, принадлежащих тому же самому пользователю. Например, захват видеоизображения в реальном масштабе времени очень чувствителен к производительности, и поэтому пользователь будет весьма раздосадован, если ваша программа решит повысить свой приоритет настолько, что в приложении, захватывающем видеоизображение, начнут пропускаться кадры.

Устанавливая приоритеты потоков, следует проявлять благоразумие. Как правило, приоритеты устанавливаются для потоков в пределах от 0 до 0,5. А более высокие приоритеты допускается устанавливать лишь в исключительных случаях, когда просто нет иного выхода. И не следует этого делать только потому, что требуется каким-то образом ускорить решение счетной задачи.

Синхронизация потоков

```
3 static NSLock *lock;  
4 static NSMutableArray *messages;
```

```
5
6 void recordLogMessage(NSString *msg)
7 {
8     [lock lock];
9     @try
10    {
11        [messages addObject: msg];
12        NSLog(@"%@", msg);
13    }
14    @finally
15    {
16        [lock unlock];
17    }
18 }
```

Пример кода из файла `lock.m`

Два потока можно синхронизировать несколькими способами. Все классы, выполняющие такого рода синхронизацию, соответствуют протоколу `NSLocking`. В нем определены два метода: `-lock` и `-unlock`.

Простейший класс блокировки `NSLock` реализует *блокировку со взаимным исключением* (так называемый *мьютекс*). Это позволяет блокировать потоки по очереди. Если послать одному потоку сообщение `-lock`, то другой поток, отправляющий сообщение `-lock` тому же самому объекту, будет заблокирован до тех пор, пока не будет отправлено соответствующее сообщение `-unlock`.

Такой механизм блокировки потоков удобен для защиты критических, но коротких разделов кода. А для защиты более длинных разделов кода более пригодным зачастую оказывается класс `NSRecursiveLock`. Если отправить два сообщения `-lock` объекту типа `NSLock` в одном и том же потоке, то возникнет взаимоблокировка. В то же время объекту типа `NSRecursiveLock` можно отправить сколько угодно сообщений `-lock`, но при условии, что ему будет отправлено столько же сообщений `-unlock`. Такой механизм блоки-

ровки потоков удобно применять в тех методах, которые можно вызывать не только из кода, уже получившего блокировку, но из кода, пока еще не имеющего ее.

Традиционные трудности применения обоих упомянутых выше классов состоят в том, что они не достаточно хорошо сочетаются с обработкой исключений. Если в коде получена блокировка и затем вызывается блок, генерирующий исключение, это может привести к серьезным осложнениям, поскольку блокировка вообще не освободится.

Когда разработчики из компании Apple внедрили в свою версию Objective-C новые ключевые слова для обработки исключений, то вместе с ними они внедрили и ключевое слово `@synchronized`. Это ключевое слово позволяет заблокировать произвольный объект и убедиться в том, что блокировка освобождается, если генерируется исключение.

Вообще говоря, пользоваться этим ключевым словом не стоит. Несмотря на то что оно немного упрощает синхронизацию потоков и чем-то напоминает аналогичное языковое средство Java, его реализация неэффективна. Ведь оно требует связать блокировку с объектом и всегда применяет рекурсивную блокировку, даже если она далека от идеала. Аналогичного эффекта можно добиться, непременно организовав отправку сообщения `-unlock` в блоке `@finally`.

Но можно обойтись и без блоков `@try` и `@finally`, если точно знать, что ни один из вызываемых методов или функций не сгенерирует исключение. Хотя в современных динамических библиотеках от GNU и компании Apple эти блоки не накладывают никаких дополнительных затрат вычислительных ресурсов во время выполнения.

Сохранение данных, характерных для потоков

```
6 NSMutableDictionary *threadDict =  
7    [[NSThread currentThread] threadDictionary];
```

```
8 [threadDict setObject: @"default"  
9      forKey: @"NewKey"];
```

Пример кода из файла threadDict.m

Раз уж имеются потоки, то не мешает сохранить данные частным образом для каждого потока. Характерным тому примером может служить текущий графический контекст в среде Сосоа, который должен быть доступным для любого кода, выполняющего рисование, но не разделяемым среди потоков.

Как правило, подобная операция реализуется с помощью функции `pthread_set_specific()`, сохраняющей указатель, связанный с текущим потоком. С этой целью адрес области памяти сохраняется в регистре, а каждый бит характерных для потока данных — с фиксированным смещением относительно начала этой области. Это означает, что для каждого локального указателя на отдельный поток требуется место в памяти, даже если лишь в одном из них устанавливается значение, отличающееся от `NULL`.

Если бы графический контекст сохранялся в среде Сосоа подобным образом, для каждого потока в отдельности потребовалось бы пространство в одно слово. Хотя большинству программ графический контекст требуется лишь в одном или максимум в двух потоках.

В качестве выхода из этого положения может служить *словарь потока*. Это изменяемый словарь, связанный с потоком. Для сохранения указателя на каждый поток требуется одно слово, но в таком словаре можно хранить сколько угодно объектов. Пространство для сохранения значения, связанного с ключом, требуется только тем потокам, у которых имеется такое значение.

Для доступа к словарю потока отправляется сообщение `-threadDictionary` текущему потоку. После этого им можно манипулировать, как и любым другим словарем. По завершении потока его словарь разрушается. А поскольку

словари сохраняют свои аргументы, то это означает, что после запуска на выполнение соответствующих деструкторов из памяти будут освобождены лишь те объекты, на которые делается ссылка в словаре потока.

Данным обстоятельством можно было бы воспользоваться для выполнения кода по завершении потока, но делать этого все же не рекомендуется. Потоки будут посылать уведомление `NSThreadWillExitNotification` перед своим завершением, и поэтому достаточно организовать наблюдение за этим уведомлением, если требуется выполнить некоторый код очистки.

На заметку

На большинстве платформ можно пользоваться ключевым словом `__thread`, которое представляет собой расширение GCC для сохранения переменных в локальной памяти потока. Но загрузчик Darwin не поддерживает области локальной памяти потоков, и поэтому в Mac OS X такая возможность отсутствует. Впрочем, это положение может вскоре измениться с появлением аналогичных функциональных возможностей в последующих версиях обоих стандартов на C и C++.

Словарь потока предоставляет отличную возможность для передачи параметров между отдельными стековыми фреймами. Ведь иногда требуется установить полуглобальное состояние, которое должно сохраняться в течение вызова метода и быть доступно из всех методов в данной области действия.

Характерным тому примером может служить объект графического контекста в среде Cocoa, который устанавливается в качестве корня иерархии представлений, а затем видоизменяется и используется во всех вложенных вызовах. Аналогичным шаблоном я, например, воспользовался в среде `LanguageKit` для организации доступа к объекту компилятора из узлов абстрактного синтаксического дерева (AST). В обоих случаях отсутствие сильной связности означает, что

один и тот же метод может быть вызван для того же самого объекта из двух разных потоков с разными контекстами.

Ожидание по условию

```
19 - (void)addToQueue
20 {
21     [condition lock];
22     queueSize++;
23     [condition signal];
24     [condition unlock];
25 }
26 - (void)main
27 {
28     while (1)
29     {
30         id pool = [NSAutoreleasePool new];
31         [condition lock];
32         while (queueSize == 0)
33         {
34             [condition wait];
35         }
36         queueSize--;
37         [condition unlock];
38         NSLog(@"Processed data from queue");
39         [pool release];
40     }
41 }
```

Пример кода из файла condition.m

Очень часто требуется, чтобы один поток находился в состоянии ожидания и активизировался в тот момент, когда в другом потоке выполняется некоторое действие. В прикладном интерфейсе API, разработанном для многопоточной обработки по стандарту POSIX, для этой цели предоставляются *условные переменные*.

Условная переменная применяется вместе с мьютексом. Для ожидания условной переменной в потоке следует сна-

чала получить мьютекс, а затем автоматически освободить его и перейти в состояние ожидания по условной переменной. Когда условная переменная получает сигнал, поток активизируется и автоматически получает мьютекс снова. В этой связи условные переменные можно рассматривать как способ передачи мьютекса между потоками.

В спецификации на прикладной интерфейс OpenStep определен класс `NSConditionLock`, инкапсулирующий низкоуровневую условную блокировку примитивного типа. В отличие от версии этой условной блокировки по стандарту POSIX, которую требуется связывать с мьютексом при каждом вызове, в этом классе содержится собственный мьютекс и реализуется протокол `NSLocking`.

Таким образом, условную блокировку можно применять как обычную блокировку, хотя это и было бы расточительством. Как и с обычной блокировкой, с условной блокировкой связана целочисленная переменная, представляющая некоторое условие. При переходе в состояние ожидания по условной блокировке этой переменной можно присвоить определенное значение. В таком случае поток не активизируется до тех пор, пока условная блокировка не получит сигнал с этим значением.

Условная блокировка оказывается очень удобной в том случае, если ожидаемое условие представляет собой целочисленное значение, но менее удобной в остальных случаях. Нередко требуется, чтобы поток активизировался при переходе объекта в определенное состояние. Этого нетрудно добиться с помощью низкоуровневых функций из интерфейса POSIX, поскольку в них предполагается, что условие будет проверяться в коде вручную. Но добиться того же самого результата средствами класса `NSConditionLock` несколько труднее.

В качестве выхода из положения в версии Objective-C от компании Apple был внедрен класс `NSCondition`. Он представляет собой простую оболочку для мьютекса и условной

переменной из интерфейса POSIX. Условный объект этого класса можно заблокировать, перейти по нему в состояние ожидания, сняв блокировку, а затем сигнализировать ему из другого потока активизироваться.

При активизации после ожидания по условию типа `NSCondition` условный объект будет заблокирован. Поэтому заданное условие следует проверить вручную, а затем разблокировать условный объект или же вернуться в состояние ожидания.

По этому же шаблону условными переменными можно оперировать практически везде, где они применяются. Как правило, они позволяют установить отношения “поставщик–потребитель” между потоками. Поток-потребитель должен перейти в состояние ожидания до тех пор, пока не появятся некоторые ожидающие его данные. А поток-поставщик должен активизировать поток-потребитель, когда для него имеются некоторые данные.

В примере кода, приведенном в начале этого раздела, демонстрируются два метода из одного класса. Метод `-addToQueue` предполагается вызывать из потока-поставщика, тогда как метод `-main` реализует поток-потребитель.

В коде потока-потребителя демонстрируется ряд важных особенностей потокового объекта. Вместо цикла исполнения типа `NSRunLoop` в этом коде организуется собственный, бесконечный цикл исполнения, на каждом шаге которого создается и разрушается явным образом автоматически освобождаемый пул. Кроме того, на каждом шаге этого цикла сначала осуществляется блокировка по условию, а затем блокировка снимается и происходит переход в состояние ожидания, если ожидаемые данные отсутствуют.

Далее поток-поставщик может получить блокировку и сигнализировать условие. В этом случае поток-потребитель активизируется, как только блокировка становится доступной. Поток-поставщик разрешает это, снимая блокировку.

После этого поток-потребитель получает следующий фрагмент данных для обработки и снимает блокировку. В рассматриваемом здесь примере инкрементирование и декрементирование общей переменной служит для замещения операции ввода некоторых данных в разделяемую структуру.

На заметку

Последовательность операций, требующихся для активизации потока по условной переменной, обходится очень дорого с точки зрения потребляемых вычислительных ресурсов, и поэтому ее следует по мере возможности избегать. Для сообщения между двумя потоками, как правило, рекомендуется применять неблокируемый кольцевой буфер, который переходит в режим блокировки лишь после его опорожнения. Эта модель реализована в технологии Grand Central Dispatch, разработанной компанией Apple, и поэтому ею можно воспользоваться, не прибегая к самостоятельному написанию кода кольцевого буфера.

Блоки и Grand Central Dispatch

К числу самых последних дополнений Objective-C следует отнести поддержку *блоков*, называемых иначе *замыканиями*. Блоки применялись еще в языке Smalltalk, но были исключены из Objective-C по целому ряду причин. В Smalltalk они служили для управления потоками, тогда как в Objective-C примитивные управляющие конструкции унаследованы от C, и поэтому блоки не требуются. Дополнительное усложнение компилятора и динамической библиотеки наряду с потерей быстродействия не способствовали поначалу внедрению блоков в Objective-C как приоритетной задачи.

Замыкание представляет собой функцию, которая может быть создана в пределах какой-нибудь другой области действия, чтобы обращаться к переменным в этой области действия. Как правило, такая функция может обращаться к следующим элементам кода: глобальным переменным, аргументам и локальным переменным. Она может существовать только в глобальной области действия. С другой стороны, блок может быть объявлен в функции и даже в другом блоке. Блок может обращаться к любым переменным, доступным в той области действия, где он объявлен.

Привязка переменных к блокам

```
3 int (^add)(int, int) =  
4   ^(int a, int b)  
5   {  
6     return a + b;  
7   };
```

Пример кода из файла `blockScope.m`

Функции можно рассматривать как частный случай блоков. В принципе функция представляет собой блок, объявленный в глобальной области действия. Они реализуются по-разному, хотя эти отличия весьма незначительны.

Таким образом, блок в простейшем случае эквивалентен функции. Он принимает ряд аргументов, выдает некоторый результат и не обращается ни к одному из элементов кода за пределами своей области действия. Подобного рода блок демонстрируется в кратком примере кода, приведенном в начале этого раздела. Он объявляется в глобальной области действия и скорее похож на довольно многословную форму объявления функции.

Другая разновидность блока обращается к элементам кода в области действия той функции, где этот блок объявлен. Такого рода блок демонстрируется в остальной части кода из файла `blockScope.m`.

Этот блок обращается к двум переменным. Так, переменная `counter` является статической, а это означает, что она хранится в некоторой области памяти, выделяемой из образа программы, подобно глобальной переменной. Обращение к этой переменной в блоке осуществляется аналогично обращению к любой другой статической или глобальной переменной. Но у этой переменной имеется только одна копия.

На заметку

Несмотря на то что для программирования доступна лишь одна разновидность блоков, в реализации присутствуют обе их разновидности. Для того чтобы выяснить имеющуюся разновидность блока, достаточно проверить указатель `isa`, хотя это и не гарантируется открытым прикладным интерфейсом API и подлежит изменению без всяких уведомлений.

Если создать блок, который обращается только к своим параметрам или глобальным переменным, то память для такого блока распределяется статически. И всякий раз, когда он создается концептуально, получается указатель на один и тот же объект. А если создать блок, который обращается к элементам данных в стеке, то динамическая библиотека блоков будет порождать его новую копию в “куче” всякий раз, когда этот блок будет создаваться.

Другая переменная называется `c` и является локальной. Для нее выделяется место в стеке средствами функции `getCounter()`. При возврате из этой функции переменная `c` разрушается. А блок продолжает существовать, но теперь он обращается к копии этой переменной. Это означает, что переменная `c` в блоке всегда содержит значение, которое он имела на момент создания блока. Это видно из приведенного ниже результата выполнения рассматриваемого здесь примера кода. Каждый блок выводит на экран значение счетчика на момент своего создания, а также текущее значение счетчика.

```
10 void(^getCounter(void))(void)
11 {
12     static int counter;
13     int c = counter;
14     void(^block)(void) = ^(void)
15     {
16         printf("current: %d ", counter);
17         printf("old: %d\n", c);
18     };
```

```
19     counter++;
20     return Block_copy(block);
21 }
22
23 int main(void)
24 {
25     void(^block)(void) = getCounter();
26     block(); block();
27     void(^block2)(void) = getCounter();
28     block2(); block2();
29     block(); block();
30     return 0;
31 }
```

Пример кода из файла `blockScope.m`

```
1 current: 1 old: 0
2 current: 1 old: 0
3 current: 2 old: 1
4 current: 2 old: 1
5 current: 2 old: 0
6 current: 2 old: 0
```

Результат выполнения кода из файла `blockScope.m`

По традиции замыкание обращается к подлинным переменным в охватывающей области действия, а не к их копиям. Такого поведения можно добиться, объявив переменную с описателем `__block`. Но реализовать его очень сложно. Ведь эта переменная перемещается из стека во фрагмент кода с подсчетом ссылок в области “кучи”, выделяемой в памяти. А это означает, что каждая ссылка на данную переменную будет ссылкой на ту же самую переменную. Это обстоятельство демонстрируется в приведенном ниже примере кода из файла `blockCapture.m`, где представлена функция, создающая два связанных вместе счетчика.

```
3 typedef int(^counter_t)(void);
4
```

```
5 void linkedCounters(counter_t *a, counter_t *b)
6 {
7     __block int c = 0;
8     *a = ^(void) { return c++; };
9     *b = ^(void) { return ++c; };
10    *a = Block_copy(*a);
11    *b = Block_copy(*b);
12    c++;
13 }
14
15 int main(void)
16 {
17     counter_t pre, post;
18     linkedCounters(&post, &pre);
19     printf("%d %d, %d, %d\n",
20         pre(), post(), pre(), post());
21     return 0;
22 }
```

Пример кода из файла blockCapture.m

Оба счетчика обращаются к одной и той же переменной в памяти: один из них возвращает и затем инкрементирует ее, а другой — инкрементирует и затем возвращает ее. При выполнении рассматриваемого здесь примера кода можно обнаружить, что вызов одного блока оказывает влияние на возврат значения переменной при последующем вызове другого блока, как показано ниже.

```
1 2 2, 4, 4
```

Результат выполнения кода из файла blockCapture.m

При последующем вызове функции `linkedCounters()` будет получена новая пара блоков, обращающихся к новой переменной счетчика. Обратите внимание на отличие в описателях области памяти `static` и `__block`. Оба описателя дают одинаковый результат, перемещая переменную

из стека и давая возможность сохранить ее дольше, чем сама функция, но действуют они по-разному.

Если переменная объявляется как `static`, то каждая ссылка на нее будет обращаться к одной и той же ячейке памяти. А если переменная объявляется как `__block`, то каждая ссылка на нее из той же самой области действия будет по-прежнему обращаться к той же ячейке памяти, но при каждом переходе к той области действия, где она объявлена, будет получена ее новая копия.

Управление памятью при использовании блоков

```
5 __block int counter;
6 int(Ablock)(void) =
7   ^(void) { return counter++; };
8 block = [block copy];
9 block_release(block);
```

Пример кода из файла `blockMemory.m`

По традиции для управления памятью в среде Сосоа применяется подсчет ссылок. По этому же принципу действуют и блоки. Для управления подсчетом ссылок в блоке имеются две функции. В частности, функция `_block_copy()` увеличивает свой подсчет ссылок, а функция `_block_release()` уменьшает его.

Любопытно, что первая из этих функций не называется `_block_retain()`, чтобы было явно видно, что возвращаемый блок может и не оказаться блоком, переданным этой функции в качестве аргумента. В некоторых случаях блок может быть сначала размещен в стеке, а затем скопирован в “кучу”. Но чаще всего эти функции не вызываются непосредственно. Вместо этого используются макрокоманды с аналогичными именами, но без знака подчеркивания в качестве префикса. Эти макрокоманды заключают в обо-

лочку обе функции и гарантируют, что возвращаемый тип будет полностью соответствовать типу аргумента.

В Objective-C имеется также возможность посылать блокам сообщения `-retain`, `-release` и `-autorelease`, как и любому другому объекту. В действительности блокам можно отправить какие угодно сообщения, хотя блоки отреагируют лишь на некоторые из них. Компилятор не выдаст никаких предупреждений, если отправить блокам любые сообщения, но во время выполнения будет получено исключение.

Отправляя сообщение `-retain` блокам, следует проявить особую аккуратность и внимательность. Соглашение о вызываемом при этом методе требует, чтобы он возвращал получатель сообщения. Поэтому для гарантии копирования блоков из стека в “кучу” следует воспользоваться методом `-copy`. Это особенно важно сделать в режиме “сборки мусора”, где сообщения `-retain` негласно игнорируются. А в режиме ARC вызывать упомянутые выше функции или отправлять эти сообщения вообще не нужно, поскольку управление памятью, выделяемой под блоки, осуществляет механизмом ARC автоматически.

Управление памятью, выделяемой под переменные, к которым обращаются блоки, может оказаться более сложным для понимания. В приведенном ниже примере кода из файла `blockRetain.m` демонстрируются два блока, обращающихся к объектам. Первый блок получает копию указателя на объект, а второй — разделяемую ссылку, поскольку в нем используется описатель области памяти `__block`.

```
5 id a = [@"a" mutableCopy];
6 __block id b = [@"b" mutableCopy];
7 unsigned long(^refCountA)() = ^()
8 { return [a retainCount]; };
9 unsigned long(^refCountB)() = ^()
10 { return [b retainCount]; };
11 printf("%ld %ld\n", refCountA(), refCountB());
12 refCountA = Block_copy(refCountA);
```

```
13 refCountB = Block_copy(refCountB);  
14 printf("%ld %ld\n", refCountA(), refCountB());  
15 [refCountA release]; [refCountB release];  
16 printf("%ld %ld\n", [a retainCount],  
    [b retainCount]);
```

Пример кода из файла `blockRetain.m`

При сохранении блока с копией указателя на объект вместе с ним автоматически и неявным образом сохраняется и сам объект. Но это происходит лишь один раз. А при сохранении блока во второй раз этого не происходит. Объект освобождается из памяти при разрушении блока.

```
1 1 1  
2 2 1  
3 1 1152921504606846975
```

Результат выполнения кода из файла `blockRetain.m`

Совсем иначе дело обстоит со вторым блоком. Этот блок был освобожден из памяти, и поэтому оправка ему сообщение приводит к неопределенному поведению. В данном конкретном случае возвращаются некоторые данные из произвольной области памяти. А в другом случае это могло бы привести к аварийному сбою.

Ответственность за сохранение объекта действительным до тех пор, пока сохраняется блок, возлагается на программирующего. Но сделать это корректно не так-то просто, поскольку блоки, в отличие от объектов, не позволяют зарегистрировать код очистки. В качестве выхода из этого затруднительного положения можно скомпилировать код в режиме ARC. В этом случае блоки автоматически сохраняют любые объекты, хранящиеся в связанных с ними переменных, а затем освободят их из памяти, когда сами разрушатся.

Затруднение возникает и в том случае, если блок обращается к другим указателям. Даже если указатель и не снабжен описателем области памяти `__block`, область па-

мяти, на которую он ссылается, все равно не копируется. Это означает, что такую область памяти придется отслеживать как и любую другую область памяти в С. Но дело усложняется еще и тем, что блок, содержащий ссылку на эту область памяти, может храниться длительное время.

Выполнение действий в фоновом режиме

```
6 dispatch_queue_t q =
7   dispatch_get_global_queue(0,0);
8   __block int count;
9   dispatch_async(q, ^(void){ count++; });
10  sleep(1);
11  NSCAssert(count == 1,
12    @"Counter incremented in background");
```

Пример кода из файла libdispatch.m

Если вы пользуетесь средой Cocoa на платформе Mac OS X 0.6 или средой GNUstep на платформе FreeBSD, то для параллельной обработки вам доступна библиотека *libdispatch*, распространяемая компанией Apple под маркой Grand Central Dispatch. Эта библиотека была разработана в компании Apple и выпущена под лицензией веб-сервера Apache 2. Пользуясь библиотекой libdispatch, вам не придется управлять потоками вручную, а вместо этого управлять очередями работ.

Очереди работ выполняют единицы работы кода или данных. Имеются две разновидности очередей: параллельные и простые (действующие по принципу FIFO — “первым пришел — первым обслужен”). Параллельная очередь начинает выполнять единицы работы в том порядке, в каком они помещаются в очередь, но может делать это параллельно, выполняя одновременно не одну работу. А простая очередь ожидает завершения каждой очередной единицы работы, прежде чем приступить к следующей.

Очереди нельзя однозначно отождествлять с потоками. Параллельные очереди могут исполнять одновременно несколько потоков, а в одном потоке могут выполняться единицы работы из нескольких очередей. Оптимальное количество потоков, используемых в очереди, определяется ядром операционной системы, исходя из количества ядер процессора и загрузки системы.

Переключение контекста между потоками обходится относительно дорого с точки зрения потребляемых вычислительных ресурсов. Очевидно, что затраты оказываются не меньше, чем при вызове функции `setjmp()`, поскольку в этом случае приходится сохранять текущее состояние ЦП и затем восстанавливать его. Но имеются и другие, не вполне очевидные затраты. Доступ к различным фрагментам данных программы вряд ли будет осуществляться из разных потоков одновременно, и поэтому переключение потоков может также вызвать немало неудачных обращений к кеш-памяти и буферу быстрого преобразования адреса (TLB).

В соответствии с моделью очереди работ в отдельном потоке может выполняться сначала одна единица работы из одной очереди, затем другая единица работы из другой очереди без необходимости постоянно переключаться между двумя рабочими наборами. Если процессор состоит из нескольких ядер, с их помощью можно повысить степень распараллеливания выполняемых работ, не увеличивая количество переключений контекстов. Когда система занята, количество потоков, используемых в одном приложении, можно сократить, чтобы эффективно использовать одно ядро процессора, предоставив другое его ядро в распоряжение другого приложения. И все это делается автоматически средствами библиотеки `libdispatch`.

Функционирование библиотеки `Grand Central Dispatch` не зависит от блоков, но ею намного проще пользоваться в сочетании с блоками. В простейшем случае блоки просто

помещаются в очередь работ и затем выполняются в фоновом режиме. Именно это и демонстрируется в примере кода, приведенном в начале данного раздела. В этом коде сначала получается дескриптор используемой по умолчанию очереди работ с обычным приоритетом, а затем в нее помещается блок. Блок начнет выполняться в фоновом режиме работы очереди, как только появится свободное время ЦП.

Организация специальных очередей работ

```
5 dispatch_queue_t q =  
6 dispatch_queue_create("Example", 0);  
7 __block int count;  
8 dispatch_async(q, ^(void){ count++; });  
9 dispatch_async(q, ^(void){ count++; });  
10 dispatch_async(q, ^(void){ count++; });  
11 dispatch_async(q, ^(void){ printf("%d\n", count); });
```

Пример кода из файла libdispatchFIFO.m

В состав Grand Central Dispatch входят три параллельные очереди, которые организуются по умолчанию. Они регистрируются на трех уровнях приоритетности: низком, исходном и высоком. Количество параллельно выполняемых единиц работ в каждой из этих очередей зависит от загрузки системы и количества имеющихся ядер процессора. Дескриптор одной из этих очередей можно получить с помощью функции `dispatch_get_global_queue()`, употреблявшейся в примере кода из предыдущего раздела.

Организовывать новую параллельную очередь не имеет никакого смысла. Блоки, помещаемые в параллельную очередь, начинают выполняться в порядке их размещения в этой очереди, но они могут завершиться в произвольном порядке, а выполняться одновременно — в любом количестве.

Имеется также специальная очередь, предназначенная для исполнения основного потока. Ссылку на нее можно по-

лучить с помощью функции `dispatch_get_main_queue()`. Эта очередь используется в Mac OS X 10.6 для реализации цикла исполнения.

В связи с тем что эта очередь является простой, имеется возможность создать аналогичные очереди. Простая очередь сродни легковесному потоку. Она выполняется последовательно и по блокам. Взаимосвязь между потоками и очередями переменчива. Очереди могут в конечном итоге выполняться параллельно в отдельных потоках или же все вместе в одном потоке операционной системы, что зависит от различных факторов.

Для создания новой простой очереди достаточно вызвать функцию `dispatch_queue_create()`. Это полезно при наличии нескольких единиц работы, которые требуется выполнить в определенном порядке, но совершенно независимо от остальной части программы. А кроме того, сокращается потребность в явной синхронизации.

В примере кода, приведенном в начале этого раздела, в очередь помещаются три блока, инкрементирующие счетчик, а еще один блок, выводящий результирующее значение счетчика, — в новую очередь. Все эти блоки взаимосвязаны, поэтому выполнять их в неверном порядке нельзя. В данном примере все блоки гарантированно выполняются в правильном порядке относительно друг друга, но не относительно остальной части программы. Это означает, что на переменную `counter` не нужно накладывать никаких блокировок, если, конечно, не предполагается доступ за пределами очереди.

Уведомления

В Objective-C поощряется слабая связность, и лучше всего эту особенность данного языка программирования выражает поддерживаемый в среде Foundation механизм *уведомлений*. Уведомления представляют собой простой способ реализации обратных вызовов для произвольного числа приемников.

Уведомления можно принимать по конкретному имени, от конкретного объекта или и тем и другим способом. Само уведомление содержит адрес отправителя, имя уведомления, а также связанный с ним словарь. Это позволяет передавать из объектов произвольное количество информации приемникам уведомлений.

Благодаря уведомлениям можно очень просто организовать повторное использование классов. Вместо того чтобы рассчитывать на то, что в каждом классе, который собирается принимать события от создаваемых вами классов, должен быть реализован конкретный интерфейс, вы можете просто послать ему уведомление. Любое количество классов может принимать уведомления от одного и того же объекта, а в своем классе вам совсем не обязательно реализовывать какую-то логику для сохранения ссылок на них.

Запрашивание уведомлений

```
5 - (void)notify: (NSNotification*)note
6 {
7     NSLog(@"Received %@", note);
8 }
9 - (void)registerListener
10 {
11     NSNotificationCenter *nc =
12         [NSNotificationCenter defaultCenter];
13     [nc addObserver: self
14         selector: @selector(notify:)
15         name: @"Example"
16         object: nil];
17 }
18 - (void)dealloc
19 {
20     NSNotificationCenter *nc =
21         [NSNotificationCenter defaultCenter];
22     [nc removeObserver: self];
23     [super dealloc];
24 }
```

Пример кода из файла notify.m

Для организации приема уведомлений необходимо зарегистрировать свою заинтересованность в них с помощью объекта типа `NSNotificationCenter` для своего потока. Этот объект служит в качестве своего рода пункта маршрутизации уведомлений. Когда уведомление отправляется, оно направляется сначала в центр уведомлений, а затем пересылается всем заинтересованным сторонам.

Чаще всего используется стандартный центр уведомлений, доступный по умолчанию. Можно, конечно, создать и другие центры уведомлений для различного применения, но на практике обычно используется стандартный центр. Для его получения достаточно отправить сообщение `+defaultCenter` объекту типа `NSNotificationCenter`.

Далее для регистрации уведомления центру уведомлений посылается сообщение `-addObserver:selector:name:object:.` Обратите внимание на аргумент в этом сообщении. Один объект может принимать разные уведомления для различных методов. И это очень удобно, поскольку избавляет от необходимости проверять вид получаемого уведомления, но дает возможность отчетливо выделить различные обработчики событий в разные методы.

Объект или имя уведомления может быть обозначено константой `nil` при регистрации наблюдателя. Это дает возможность зарегистрировать все сообщения, отправленные конкретному объекту, или же все уведомления конкретного вида независимо от их отправителя.

Это удобно в большей степени для отладки, чем для каких-нибудь иных целей. Например, можно написать код регистрации, принимающий все уведомления определенного вида и регистрирующий отправителя.

Организуя прием уведомлений, следует иметь в виду, что принимающий уведомления объект не сохраняется в центре уведомлений. Если этот объект все еще зарегистрирован для приема уведомлений в тот момент, когда он разрушается, то при последующей отправке этих сообщений произойдет аварийный сбой программы.

Во избежание этого центру уведомлений следует непременно отправить сообщение `-removeObserver:` в методе `-dealloc`. Если забыть это сделать, аварийный сбой программы окажется не совсем очевидным. При отслеживании кода в памяти может быть выявлено немало странного, в том числе и центр уведомлений.

Отправка уведомлений

```
33 NSNotificationCenter *nc =  
34 [NSNotificationCenter defaultCenter];  
35 [nc postNotificationName: @"Example"
```

```
36         object: @"sender"  
37         userInfo: nil];
```

Пример кода из файла `notify.m`

Отправить уведомления еще проще, чем зарегистрироваться для их приема. Канонический способ отправки уведомления состоит в том, чтобы сначала создать объект уведомления типа `NSNotification`, а затем послать его.

Но на практике едва ли можно встретить код, в котором уведомления посылаются именно таким способом. Ведь у центра уведомлений имеется ряд служебных методов для формирования и отправки уведомлений в течение одной операции.

Чаще всего уведомления отправляются с помощью метода `-postNotificationName:object:userInfo:`. Имеется упрощенный вариант этого метода без параметра `userInfo:`, что равнозначно передаче данному методу константы `nil` в качестве последнего параметра.

Обратите внимание на то, что объект, выступающий в роли отправителя уведомления, указывается в качестве параметра данного метода. Это означает, что уведомления можно посылать по требованию от другого объекта. Например, в шаблонах фасадов или в коде C можно посылать уведомления от имени других объектов.

Постановка уведомлений в очередь

```
44 NSNotificationQueue *nq =  
45     [NSNotificationQueue defaultQueue];  
46 NSNotification *note = [NSNotification  
47     notificationWithName: @"Example"  
48         object: @"sender"];  
49 [nq enqueueNotification: note  
50     postingStyle: NSPostWhenIdle];  
51 [[NSRunLoop currentRunLoop] run];
```

Пример кода из файла `notify.m`

Иногда в коде прикладной программы может быть сформировано много уведомлений или же те уведомления, которые не требуют немедленной обработки. Обе эти задачи позволяет решить класс `NSNotificationQueue`.

Очереди уведомлений позволяют не только отложить их доставку, но и объединить их. В отличие от центра уведомлений типа `NSNotificationCenter`, у очереди нет служебных методов для формирования уведомлений, и поэтому вам придется сделать это самостоятельно. Сформированное таким образом уведомление затем посылается одним из трех способов.

В примере кода, приведенном в начале этого раздела, демонстрируется откладывание доставки уведомления до тех пор, пока не наступят другие события в цикле исполнения. Таймеры и данные, которые становятся доступными по дескрипторам файлов, получают приоритет. С другой стороны, уведомление можно послать на следующем шаге цикла исполнения или же безотлагательно.

Если приходится отправлять много однотипных уведомлений, то объект уведомления рекомендуется кешировать. Неоднократная передача одного и того же объекта уведомления очереди обойдется совсем дешево с точки зрения потребляемых вычислительных ресурсов, а само уведомление может быть сразу же отправлено из очереди автоматически.

Если же для постановки уведомлений в очередь применяется более сложный метод, то один из его параметров, `coalesceMask:`, определяет порядок объединения уведомлений. Уведомления можно объединять по одинаковому имени, отправителю или и тому и другому.

Объединять уведомления удобнее всего при способе отправки типа `NSPostWhenIdle`. Одно и то же уведомление можно отправить многократно, когда код прикладной программы занят обработкой других событий. А когда они будут обработаны, очередь уведомлений отправит лишь одну копию уведомления. Такой способ идеально подходит для

тех фрагментов кода, в которых запускается низкоприоритетная задача по учету использования системных ресурсов.

Обмен уведомлениями между приложениями

```
19 [c      addObserver: self
20         selector:
21         @selector(newCopyStarted:)
22         name: ProcessDidStart
23         object: nil
24     suspensionBehavior:
25     NSNotificationSuspensionBehaviorHold];
```

Пример кода из файла `distributedNotify.m`

Слабая связность, обеспечиваемая уведомлениями, оказалась настолько полезной, что была расширена в среде Foundation с целью предоставить простой механизм общения между приложениями. С этой целью был внедрен подкласс `NSDistributedNotificationCenter`, производный от класса `NSNotificationCenter` и пересылающий уведомления другим приложениям.

Такое расширение требует внесения некоторых изменений. Обычное уведомление, как правило, связывается с отправителем, который представляет собой указатель на некоторый объект. Но указателем на объект не очень удобно пользоваться в процессе с другим адресным пространством, и поэтому в распространяемых уведомлениях для указания отправителя используется символьная строка. Эта строка может содержать все что угодно, но зачастую в ней указывается имя приложения.

С уведомлениями могут быть также связаны словари. В самом приложении эти словари могут содержать любые объекты, но для пересылки их содержимое ограничивается объектами, которые могут храниться в *списках свойств* (см. главу 9).

На заметку

В Mac OS X имеется лишь одна разновидность центра распространяемых уведомлений, посылающего их приложениям, выполняющимся под тем же самым идентификатором пользователя, что и у отправителя. А в GNUstep предоставляются две другие разновидности этого центра: одна — для сообщения со всеми приложениями, работающими на компьютере, независимо от конкретного пользователя, другая — для пересылки уведомлений по локальной сети.

Семантика доставки распространяемых уведомлений также имеет некоторые отличия. Если отправляется обычное уведомление, то оно доставляется синхронно. Посылающий его объект уже имеет управление ЦП. А стековый фрейм для его метода находится на вершине стека и свободен для делегирования дальнейшего выполнения программы центру уведомлений, который затем делегирует его приемникам уведомлений.

Совсем иначе дело обстоит с распространяемыми уведомлениями. В этом случае нельзя просто прервать выполнение другого процесса посредине¹ и придется ожидать удобного момента. Распространяемые уведомления для этой цели сопрягаются с циклом исполнения. Для получения распространяемых уведомлений необходимо воспользоваться классом `NSRunLoop`. В примере кода из файла `distributedNotify.m` демонстрируется простая программа, выводящая зарегистрированное сообщение всякий раз, когда запускается новый ее экземпляр.

```
14 - (id)init
15 {
16     NSDistributedNotificationCenter *c =
17     [NSDistributedNotificationCenter
```

¹ На самом деле можно прервать выполнение процесса. Для этой цели служат сигналы, но операции, которые могут быть выполнены обработчиком сигналов, весьма ограничены.

```
        defaultCenter];  
18  
19 [c addObserver: self  
20      selector:  
21      @selector(newCopyStarted:)  
22      name: ProcessDidStart  
23      object: nil  
24      suspensionBehavior:  
25      NSNotificationSuspensionBehaviorHold];  
26  
27 NSProcessInfo *pinfo =  
28 [NSProcessInfo processInfo];  
29 NSNumber *pid = [NSNumber numberWithInt:  
30 [pinfo processIdentifier]];  
31 NSDictionary *userInfo = [NSDictionary  
32 dictionaryWithObject: pid  
33                      forKey: @"pid"];  
34  
35 [c postNotificationName: ProcessDidStart  
36      object: [pinfo processName]  
37      userInfo: userInfo  
38      options: 0];  
39 [[NSRunLoop currentRunLoop] run];  
40 }
```

Пример кода из файла `distributedNotify.m`

Все необходимые действия в этой программе выполняются в методе `-init`. Сначала в нем регистрируется наблюдатель распространяемого уведомления. В качестве имени объекта указывается константа `nil`, и поэтому он будет получать данное уведомление от любого получателя. Затем формируется само уведомление. Словарь со сведениями о пользователе содержит единственную пару “ключ–значение” для хранения идентификатора процесса.

После отправки уведомления начинается цикл исполнения. Далее программа переходит в состояние блокировки до тех пор, пока в цикле исполнения не наступит

какое-нибудь событие. В рассматриваемом здесь примере программы определен обработчик лишь одного события, который только получает сообщение `-newCopyStarted`: всякий раз, когда посылается уведомление `ProcessDidStart`, что происходит лишь в тот момент, когда запускается на выполнение копия данной программы.

Если скомпилировать и запустить на выполнение программу из данного примера, то можно обнаружить, что она будет получать собственное уведомление о том, что она запущена. Если же запустить на выполнение еще одну копию этой программы, то из каждого ее экземпляра будет получено зарегистрированное сообщение о запуске ее новой копии.

Данный пример довольно прост, но распространяемые уведомления можно также использовать для сообщения о более сложных событиях. Я, например, воспользовался ими в среде `XMPPKit` для уведомления всех выполняющихся приложений, когда пользователь изменяет свое подключенное состояние. Другие программы принимают это уведомление и могут разослать сообщение о новом состоянии на сайты микроблоггинга.

Доступ к сети

Применение прикладного интерфейса Berkeley Sockets API относится к стандартным межплатформенным способам написания приложений, поддерживающих работу в сети. Этот прикладной интерфейс был построен в рамках того же субсидировавшегося проекта, в котором был разработан протокол TCP/IP, но таким образом, чтобы стать независимым от сетевых протоколов.

В большинстве языков программирования имеются реализации этого прикладного интерфейса. Как и следовало ожидать, в Objective-C можно свободно пользоваться написанной на C версией этого прикладного интерфейса, обеспечивающего всю эффективность, на которую только способны низкоуровневые интерфейсы.

Но чаще всего это требует немалых усилий, приносящих неоправданно мало пользы. Поэтому в среде Foundation предоставляется ряд классов, упрощающих написание сетевых приложений. В силу подобных абстракций иногда теряется эффективность низкоуровневых прикладных интерфейсов, хотя и не намного.

Заключение сокетов C в оболочку

```
42     return
43     [[[NSFileHandle alloc]
44      initWithFileDescriptor: s
```

```
45         closeOnDealloc: YES]  
46         autorelease];
```

Пример кода из файла `NSFileHandle+Socket.m`

Класс `NSFileHandle` был вкратце рассмотрен в главе 13, где упоминалось, что он заключает в оболочку дескриптор файла. Прикладной интерфейс API для сокетов разработан на основе применяющейся в UNIX модели, действующей по следующему принципу: все относится к файлу, даже если в этом нет никакого смысла. Поэтому сокет также является дескриптором файлов. А это означает, что для записи и чтения из сокета могут быть использованы те же самые вызовы функций, что и для записи и чтения из любого другого дескриптора файла.

Класс `NSFileHandle` позволяет заключить в оболочку и сокет. Это дает возможность пользоваться низкоуровневыми прикладными интерфейсами API для создания сокета в точном соответствии с конкретными требованиями, чтобы затем организовать взаимодействие с созданным сокетом более абстрактным способом. Если требуется воспользоваться любыми частями низкоуровневого прикладного интерфейса API для сокетов, которые труднодоступны из прикладного интерфейса API более высокого уровня, то это можно сделать только, заключив сокет в оболочку класса `NSFileHandle`. Но и в других случаях такой способ зачастую оказывается наиболее удобным.

Самое главное преимущество заключения сокета в оболочку объекта дескриптора файла состоит в том, что его можно легко внедрить в цикл исполнения. Благодаря этому намного упрощается написание кода управляемых событиями сокетов. Еще одно существенное преимущество заключается в том, что для сокетов можно пользоваться подсчетом ссылок, доступным в Objective-C. Так, если передать логическое значение YES в качестве аргумента `closeOnDealloc`: при создании объекта типа `NSFileHandle`, то дескриптор файла будет закрыт при освобождении этого объекта из памяти.

На заметку

Напомним, что язык Objective-C разрабатывался с целью упростить жизнь программирующим на С. Поэтому пользоваться прикладными интерфейсами API языка Objective-C следует лишь в том случае, если это более оправданно для решения конкретной задачи, чем обращение к прикладным интерфейсам API языка С. Не следует забывать, что применение прикладного интерфейса API языка Objective-C лишь в качестве простой оболочки для прикладного интерфейса API языка С влечет за собой дополнительные издержки и сложности, но не упрощает сопровождение кода прикладной программы.

Если отправить объекту сообщение `-waitForDataInBackgroundAndNotify`, он будет добавлен в список дескрипторов файлов, опрашиваемых в цикле исполнения. Если же на следующем шаге цикла исполнения появятся данные, ожидающие обработки, то объект типа `NSFileHandle` pošлет соответствующее уведомление. И если заранее был зарегистрирован обработчик этого уведомления, то новые данные могут быть прочитаны.

Документация на упомянутый выше метод в какой-то степени вводит в заблуждение. В ней утверждается, что в этом методе будет создан очередной поток. На самом же деле в цикле исполнения на платформе Mac OS X используется функция `kqueue()` из прикладного интерфейса API¹, а при вызове данного метода лишь добавляется дескриптор файла в список, где предполагается его обнаружить. Начиная с версии Mac OS X 10.6 этот метод реализован средствами *Grand Central Dispatch* для отправки уведомления, но его базовые функциональные возможности остались прежними.

И еще одно, последнее преимущество заключения сокета в оболочку состоит в том, что оно избавляет от необходимости беспокоиться о буферах. Для чтения данных из сокета, как правило, сначала выделяется память под буфер,

¹ Вместо этой функций в GNUstep может быть использована функция `poll()` или `select()`.

а затем указатель на него передается вызываемой системной функции `read()` или `recv()`. А если используется объект дескриптора файла, то достаточно лишь прочитать экземпляры класса `NSData`.

Если приходится много обмениваться символьными строками по сети, то, возможно, имеет смысл ввести категорию в класс `NSFileHandle` с методами `-writeUTF8String:` и `-readUTF8String:` или эквивалентными им методами для кодировки, используемой в сетевом протоколе. Это даст возможность получать и посылать символьные строки по сети непосредственно.

Установление связи с серверами

```
6  NSInputStream *in;  
7  NSOutputStream *out;  
8  NSHost *host = [NSHost hostWithName:  
9      @"example.com"];  
10 [NSStream getStreamsToHost: host  
11     port: 80  
12     inputStream: &in  
13     outputStream: &out];  
14 [in open];
```

Пример кода из файла `nsstream.m`

В Objective-C предоставляются Java-подобные классы потоков ввода-вывода для обмена данными по сети, а также доступны низкоуровневые интерфейсы UNIX. К сожалению, эти варианты реализации потоков ввода-вывода не приносят особой пользы. Ведь они не предоставляют ясный способ ввода фильтров, например, для сжатия или согласования процедуры шифрования данных посредине соединения, чего требует большинство современных сетевых протоколов.

Но самое удручающее ограничение этих классов состоит в том, что они не упрощают поддержку служебных записей SRV. Это записи, извещающие службу DNS о необходи-

мости переадресовать порты, а также отобразить имя хоста на IP-адрес. Они применяются в сетевом протоколе DNS-SD, но из-за нехватки IP-адресов находят все более широкое распространение в иерархических службах DNS общедоступного Интернета.

Если найти адрес сервера в Mac OS X, используя функцию `getaddrinfo()` из интерфейса POSIX, то на его основании можно предоставить имена хоста и службы. А затем система отобразит их на сетевые адреса и номера портов в любых поддерживаемых в ней сетевых протоколах.

При построении объекта типа `NSStream` номер порта необходимо указать в виде целочисленного значения. Но ведь это означает, что созданный таким образом поток ввода-вывода не сможет нормально функционировать в любой службе, где используются записи DNS SRV. Если же сначала вызвать функцию `getaddrinfo()`, а затем заключить полученный в итоге сокет в оболочку объекта типа `NSFileHandle`, то он будет действовать корректно. В классе `NSFileHandle` из среды `EtoileFoundation` предоставляется категория, которая позволяет создавать дескрипторы файлов непосредственно из имени хоста и наименования сетевого протокола, заключая вызов упомянутой выше функции в оболочку.

```
16 hints.ai_family = PF_UNSPEC;
17 hints.ai_socktype = SOCK_STREAM;
18 // запросить адрес потока ввода-вывода
19 error = getaddrinfo(server, service, &hints, &res0);
20 if (error) { return nil; }
21
22 int s = 1;
23 for (struct addrinfo *res = res0;
24      res != NULL && s < 0 ;
25      res = res->ai_next)
26 {
27     s = socket(res->ai_family, res->ai_socktype,
28               res->ai_protocol);
29     // если не удалось создать сокет,
```

```
// попробовать следующий адрес
30 if (s < 0) { continue ; }
31
32 // если не удалось установить соединение,
// попробовать следующий адрес
33 if (connect(s, res->ai_addr, res->ai_addrlen) < 0)
34 {
35     close(s);
36     s = 1;
37     continue;
38 }
39 }
40 freeaddrinfo(res0);
```

Пример кода из файла `NSFileHandle+Socket.m`

Применение этой категории демонстрируется на примере кода из файла `NSFileHandle+Socket.m`. При вызове функции `getaddrinfo()` предоставляются имя сервера и тип службы в виде символьных строк C, а также ряд указаний на требуемую разновидность сокета. В итоге возвращается массив структур `addrinfo`, содержащих все необходимые сведения для создания подключаемого сокета.

Эти структуры располагаются в наиболее удобном для получателя порядке, поэтому их можно опробовать по очереди до тех пор, пока не будет найдена наиболее пригодная. Так, если соединение устанавливается по протоколам IPv6 и v4, то в рассматриваемом здесь примере кода будут приняты попытки соединиться по обоим этим протоколам, при условии, что имеются записи DNS, указывающие и на тот и на другой протокол.

Общий доступ к объектам по сети

```
6 NSMutableDictionary *object =
7 [NSMutableDictionary new];
8 NSConnection *conn =
9 [NSConnection new];
```

```
10 [conn setRootObject: object];
11 if ([conn registerName: @"sharedDict"])
12 {
13     [[NSRunLoop currentRunLoop] run];
14 }
```

Пример кода из файла doPublish.m

К числу самых эффективных и недостаточно используемых языковых средств Objective-C относится система *распределенных объектов*. Так, если отправить объекту сообщение, а он не сумеет его обработать, тогда он получит сообщение `-forwardInvocation:` с объектом, инкапсулирующим отправленное ранее сообщение, в качестве аргумента. Такой объект, как и все остальное в Objective-C, можно подвергнуть интроспекции, чтобы выяснить селектор, количество аргументов, а также их типы и значения.

В состав среды Foundation входит класс `NSDistantObject`, в котором используется описанный выше алгоритм. Отдаленный объект выполняет функции промежуточного объекта, пересылающего сообщения удаленному объекту. В качестве удаленных объектов могут служить различные процессы или отдельные компьютеры. Если объекты передаются в качестве аргументов сообщению, посылаемому отдаленному объекту, то они могут быть скопированы, а иначе удаленный промежуточный объект может быть создан на дальнем конце соединения, откуда сообщение пересылается обратно.

Пользоваться распределенными объектами очень просто. Любая программа, в которой они применяются, разделяется на две основные части: клиент и сервер. Сервер предоставляет объекты другим процессам, а клиент подключается к серверу для доступа к ним.

По умолчанию общий доступ к объектам организует-ся на локальном компьютере. А для организации общего доступа к ним по сети следует воспользоваться классом `NSSocketPortNameServer` при регистрации объекта.

```

6 NSMutableDictionary *object = (id)[NSConnection
7   rootProxyForConnectionWithRegisteredName:
8   @"sharedDict"
9                                     host: nil];
10 NSLog(@"Object: %@", object);
11 [object setObject: @"aValue"
12   forKey: @"akey"];
13 NSLog(@"Object: %@", object);

```

Пример кода из файла doAccess.m

В примере кода, приведенном в начале этого раздела, демонстрируется простая программа, организующая общий доступ к экземпляру класса `NSMutableDictionary` из разных процессов. Следует, однако, иметь в виду, что для обслуживания распределенных объектов потребуется организовать цикл исполнения типа `NSRunLoop`. Система распределенных объектов вводит канал межпроцессорной связи (IPC), т.е. порт Mach или сокет, для сообщения с циклом исполнения. Когда другой процесс отправляет сообщение этому порту, некоторые данные поступают по каналу IPC, о чем своевременно уведомляется система распределенных объектов. Далее эта система строит объект типа `NSInvocation` из полученных данных и вызывает его, а затем возвращает результат, как показано ниже.

```

1 doAccess[85613:903] Object: {
2 }
3 doAccess[85613:903] Object: {
4   akey = aValue;
5 }

```

Результат выполнения кода из файла doAccess.m

Поиск одноранговых узлов в сети

```

12 NSProcessInfo *pi = [NSProcessInfo processInfo];
13 NSString *name =
14   [NSString stringWithFormat: @"%@/%@",

```

```
15     [pi hostName],
16     NSFullUserName()];
17
18     NSNetService *service =
19     [[NSNetService alloc]
20      initWithDomain: @"
21                  type: type
22                  name: name
23                  port: 123];
24     [service publish];
25
26     NSNetServiceBrowser *sb =
27     [NSNetServiceBrowser new];
28     [sb setDelegate: self];
29     [sb searchForServicesOfType: type
30      inDomain: @""];
```

Пример кода из файла `netservice.m`

Многие годы для поиска служб в локальной сети применялось немало сетевых протоколов. К числу наиболее распространенных в локальных сетях до середины 1990-х годов относились протоколы NetBIOS и AppleTalk, после чего они постепенно начали заменяться набором протоколов TCP/IP.

Но в набор протоколов TCP/IP не было внедрено никаких средств для поиска служб в сети. Наиболее близким к этому был лишь широковещательный адрес. Так, если отправить пакет по широковещательному адресу, то он будет послан всем узлам в локальном сегменте сети. Но такой подход влечет за собой немало затруднений. Он пригоден только для локального физического сегмента сети, но не подходит для современных коммутируемых сетей. А кроме того, он является слишком низкоуровневым.

Рабочая группа ZeroConf из организации IETF (Проблемной группы проектирования Интернета) предложила в качестве выхода из положения *службу многоадресных доменных имен (mDNS)*. Служба доменных имен

(DNS) представляет собой расширяемый способ публикации отображений сетевых адресов. Чаще всего она используется для отображения имен хостов (т.е. узлов сети) на IP-адреса, но с таким же успехом ее можно применять для определения телефонных номеров, геологических данных и многих других целей.

С помощью службы mDNS отдельные компьютеры в сети могут публиковать записи DNS в домене `.local`. Это зачастую служит основанием для работы *протокола обнаружения служб средствами DNS (DNS-SD)*, в котором описывается способ извещения произвольных служб через DNS. Этот стандартный протокол можно использовать и вместе с традиционной иерархической службой DNS, но наибольшую пользу он приносит во взаимодействии со службой mDNS.

Система, сочетающая в себе оба упомянутых выше стандарта (службу mDNS и протокол DNS-SD), называется *Bonjour* и доступна на уровне Objective-C через класс `NSNetService`. С помощью этого класса приложения могут публиковать записи DNS-SD и находить все записи конкретной службы.

Такой возможностью пользуются многие приложения в Mac OS X. Так, в браузере Safari подобным образом организуется общий доступ к закладкам, в приложении iChat — поиск контактов для обмена сообщениями без участия сервера, в приложении iTunes — обмен музыкальными записями, а в самой операционной системе — поиск общих папок и принтеров. Если вам приходится разрабатывать любого рода сетевые приложения, класс `NSNetService` может оказаться полезным для извещения одноранговых узлов сети.

Для извещения службы следует указать ее уникальное имя. Если вы пользуетесь рассмотренными выше распределенными объектами, то этим именем можете обозначить обслуживаемый вами объект, проигнорировав внутренний механизм работы системы Bonjour на основе службы DNS. А в других случаях полезнее указать порт для целей публи-

кации, тогда как имя можно предоставить пользователям прикладной программы.

Поиск одноранговых узлов в сети осуществляется средствами класса `NSNetServiceBrowser`, хотя правильно воспользоваться им непросто. Публикация служб производится синхронно. Достаточно послать сообщение `-publish` экземпляру класса `NSNetService`, и служба будет опубликована. А поиск осуществляется асинхронно и требует активности *цикла исполнения*.

Если отправить поисковое сообщение браузеру, он передаст запрос службе DNS. Для получения ответа может потребоваться некоторое время. Но для того чтобы не отвлекать пользователя от других занятий на время поиска, происходит немедленный возврат из вызываемого при этом метода.

Получив ответ, класс `NSNetServiceBrowser` отправит сообщение своему делегату с новым экземпляром класса `NSNetService`, описывающим каждый найденный сервер. Далее он перейдет в режим приема сообщений, оставаясь в нем до тех пор, пока ему не будет послано сообщение `-stop`.

Во многих случаях это сообщение вообще не посылается. Вы можете оставить браузер работающим в фоновом режиме на весь срок действия своего процесса и получить уведомление, как только новый одноранговый узел появится в локальной сети. После этого можно запустить обновление пользовательского интерфейса прикладной программы, чтобы отобразить в нем вновь обнаруженный одноранговый узел. Именно таким образом действует приложение iTunes, добавляя новый общий список воспроизведения музыкальных композиций в фонотеку по мере их появления в сети.

Загрузка данных по веб-адресу

```
44 [NSAutoreleasePool new];
45 NSArray *args = [[NSProcessInfo processInfo]
    arguments];
```

```
46 NSString *target = [args objectAtIndex: 1];
47 NSURL *targetURL = [NSURL URLWithString: target];
48 if (nil == targetURL) { return 1; }
49 NSURLRequest *req = [NSURLRequest requestWithURL:
    targetURL];
50 id delegate = [DownloadDelegate new];
51 NSURLDownload *dl =
52     [[NSURLDownload alloc] initWithRequest: req
53                                     delegate: delegate];
54 [dl setDestination: [targetURL lastPathComponent]
55     allowOverwrite: YES];
56 [[NSRunLoop currentRunLoop] run];
```

Пример кода из файла wget.m

Если вам приходится писать код для современной системы, подключаемой к Интернету, то у вас, скорее всего, возникнет потребность извлечь данные из удаленного места по веб-адресу, называемому иначе *унифицированным указателем информационного ресурса (URL)*. Веб-адрес служит глобальным идентификатором, составленным по определенной схеме с характерными для нее данными. В состав среды Foundation входит класс `NSURL`, инкапсулирующий веб-адреса. Этот класс производит синтаксический анализ веб-адреса из полученной символьной строки и возвращает его абстрактное представление.

В состав среды Foundation входят еще два класса для извлечения данных по заданным веб-адресам. Более простым из них является класс `NSURLDownload`, инкапсулирующий данные, загружаемые по удаленному веб-адресу. А более сложным является класс `NSURLConnection`, предоставляющий низкоуровневый способ обработки запросов и ответов.

Система загрузки данных по веб-адресу действует независимо от конкретного сетевого протокола, но предполагает определенный формат запросов и ответов. Запрос инкапсулируется в классе `NSURLRequest`, экземпляр которого затем передается одному из двух упомянутых выше классов. Далее

устанавливается соединение и посылается запрос, а в ответ возвращается один или больше объектов типа `NSURLRequest`.

Исходно в среде Foundation поддерживается загрузка данных по веб-адресам, начинающимся с префиксов сетевых протоколов `file://`, `http://`, `https://` и `ftp://`. Но помимо этих сетевых протоколов, можно добавить поддержку других протоколов, реализовав ее в подклассе, производном от класса `NSURLProtocol`. Хотя потребность в этом возникает крайне редко, вы все-таки можете, например, реализовать подобным способом поддержку SFTP или аналогичного ему сетевого протокола, а существующий код — использовать повторно для делегирования доступа к файлам системе загрузки данных по веб-адресу.

В примере кода, приведенном в начале этого раздела, демонстрируется порядок извлечения файла по указанному пользователем удаленному веб-адресу. В данном примере класс делегата используется для извещения пользователя о ходе выполнения процесса, хотя этого и не требуется.

Делегат сохраняет размер полученного в ответ файла, чтобы извещать пользователя о ходе его загрузки. Если во время передачи данных происходит сбой, на экран выводится локализованный вариант сообщения об ошибке. А при нормальном ходе загрузки на экран выводятся точки, каждая из которых обозначает 1% от общего объема загруженных данных.

Система загрузки данных по веб-адресу действует весьма эффективно, но чаще всего можно ограничиться ее подмножеством, представленным в рассматриваемом здесь примере. Это дает возможность внедрять простые веб-службы, затрачивая очень мало кода.

```
3 @interface DownloadDelegate : NSObject
4 {
5     int progress;
6     long long fetched;
7     long long size;
8 }
```

```

9  @end
10 @implementation DownloadDelegate
11 - (void)downloadDidBegin: (NSURLDownload*)download
12 {
13     NSLog(@"Started downloading");
14 }
15 - (void)download: (NSURLDownload*)download
16     didReceiveResponse: (NSURLResponse*)response
17 {
18     size = [response expectedContentLength];
19 }
20 - (void)download: (NSURLDownload*)download
21     didReceiveDataOfLength: (NSUInteger)length
22 {
23     fetched += length;
24     for (int newProgress = (fetched * 100) / size ;
25         newProgress > progress ; progress++)
26     {
27         putc('.', stderr);
28     }
29 - (void)download: (NSURLDownload*)download
30     didFailWithError: (NSError*)error
31 {
32     NSLog(@"Error: %@", [error localizedDescription]);
33     exit(1);
34 }
35 - (void)downloadDidFinish: (NSURLDownload*)download
36 {
37     NSLog(@"Finished downloading");
38     exit(0);
39 }
40 @end

```

Пример кода из файла wget.m

Отладка программ на Objective-C

Существуют две категории программистов: всегда пишущие идеальный код сразу и все остальные. Если вы причисляете себя к первой категории, то можете попустить эту главу. А если относите себя ко всем остальным, простым смертным, то и вам порой приходится проверять свой код и обнаруживать, что он работает не так, как вам хотелось бы.

Очевидно, что испытывать разочарование от написанного кода не очень приятно, но это чувство окажется не таким болезненным, если уметь быстро исправить программные ошибки. В этой главе будут рассмотрены некоторые способы выявления причин ошибок в программах на Objective-C.

Инспектирование объектов

```
3 @interface DebugObject : NSObject @end
4 @implementation DebugObject
5 - (NSString*)description
6 {
7     return @"Not a very helpful message";
8 }
9 @end
10 void debug(id anObject) {}
```

Пример кода из файла `description.m`

Чаще всего для отладки программ на Objective-C применяется *отладчик GNU (GDB)*, имеющий встроенную поддержку

ку Objective-C. Если вы работаете в интегрированной среде Xcode, то фактически пользуетесь отладчиком GDB через внешний графический интерфейс. А доступ к интерфейсу командной строки можете получить из меню Run (Выполнить). И зачастую это оказывается очень полезно, поскольку в среде Xcode предоставляется графический пользовательский интерфейс для большинства, но не для всех функций отладчика.

На заметку

В последних версиях Xcode используется отладчик *LLVM Debugger (LLDB)* вместо GDB. Графический интерфейс у обоих этих отладчиков одинаковый, а в LLDB предоставляются те же самые (в большей или меньшей степени) средства отладки, что и в GDB.

Отладчик GDB обладает двумя средствами, благодаря которым он отлично подходит для отладки программ на Objective-C. Во-первых, в его состав входит простой синтаксический анализатор и интерпретатор кода Objective-C, что дает возможность писать выражения на Objective-C непосредственно в отладчике, inspectируя подобным образом переменные. И во-вторых, в этом отладчике имеется команда `print-object`.

```
1 $ gdb ./a.out
2 (gdb) break debug
3 Breakpoint 1 at 0x100000e4d: file description.m,
   line 10.
4 (gdb) r
5 Starting program: a.out
6 Breakpoint 1, debug (anObject=0x100111bc0) at
   description.m:10
7 10 void debug(id anObject) {}
8 (gdb) print anObject
9 $1 = (id) 0x100111bc0
10 (gdb) printobject anObject
11 Not a very helpful message
12 (gdb) printobject [NSObject new]
```

```
13 <NSObject: 0x100111c30>
14 (gdb) printobject [anObject class]
15 DebugObject
16 (gdb) printobject [anObject className]
17 DebugObject
18 (gdb) printobject anObject->isa
19 DebugObject
```

Результат отладки кода из файла `description.m`

Эта команда распечатывает объект, посылая сообщение `-description` объекту, в результате чего возвращается строковый объект типа `NSString` с описанием данного объекта. Это означает, что данная команда будет выполняться только для действительных объектов. Если же попытаться выполнить ее для объектов, которые были преждевременно освобождены из памяти или область памяти которых разрушена, то отладчик аварийно завершится или же перехватит сигнал нарушения сегментации.

Исходно метод, выполняющий эту команду, реализуется в классе `NSObject`, распечатывая имя класса и адрес. В ряде других стандартных классов этот метод реализуется по-разному. Например, в классах коллекций содержимое коллекций распечатывается в формате списка свойств. В приведенном выше примере отладки кода распечатывается не очень полезное сообщение, но в своем коде вы должны попытаться вывести на экран нечто более полезное для его отладки.

Выявление ошибок управления памятью

```
5 id pool = [NSAutoreleasePool new];
6 NSMutableString *str = [NSMutableString
    stringWithString: @"example"];
7 [pool drain];
8 pool = [NSAutoreleasePool new];
9 NSMutableString *set = [NSMutableString setWithObject: str];
```

Пример кода из файла `prematureDealloc.m`

К числу самых распространенных ошибок в программах на Objective-C относится преждевременное освобождение объектов из памяти. Если пропустить в соответствующем месте программы отправку сообщения `-retain`, то указатель на объект зависнет, а подсчет ссылок на него не увеличится. Если на этот указатель делается ссылка из другого места программы или же если он не освобожден из памяти автоматически, то такой указатель еще долго будет оставаться действенным, прежде чем станет недействительным.

Когда указатель становится недействительным, то, как правило, возникает аварийный сбой в функции `objc_msgSend()` из динамической библиотеки на платформе Mac OS X или же функции `objc_msg_lookup()` из динамической библиотеки GNU. И это происходит потому, что в динамической библиотеке предпринимается попытка найти метод для обработки сообщения, но указатель на класс теперь указывает на нечто недействительное.

```
1 $ gcc -framework Cocoa -g prematureDealloc.m &&
   ./a.out
2 segmentation fault
3 $ gdb ./a.out
4 (gdb) r
5 starting program: a.out
6 Program received signal EXC_BAD_ACCESS, Could not
   access memory.
7 Reason: KERN_INVALID_ADDRESS at address: 0
   x00000007a0000010
8 0x00007ffff8296111c in objc_msgSend ()
9 (gdb) up 6
10 #6 0x00000001000000eef in main () at
   prematureDealloc.m:9
11 9   NSMutable *set = [NSMutable setWithObject: str];
12 (gdb) printobject str
13 Program received signal EXC_BAD_ACCESS, Could not
   access memory.
14 (gdb) print *str
```



```
15 $2 = {
16   <NSString> = {
17     <NSObject> = {
18       isa = 0x7a0000000
19     }, <No data fields>}, <No data fields>}
20 (gdb) printobject str->isa
21 Cannot access memory at address 0x7a0000000
22 (gdb) quit
23 $ NSZombieEnabled=YES ./a.out
24 a.out[39571:903] *** -[CFString hash]: message
    sent to deallocated instance 0x100112f00
```

Результат отладки кода из файла `prematureDealloc.m`

Обнаружить в отладчике объект, освобожденный из памяти, совсем не трудно. Для этого достаточно переместиться вверх по стеку до тех пор, пока не будет обнаружен получатель сообщения. С этой же целью можно воспользоваться классом `NSZombie`. Если установить переменную среды `NSZombieEnabled`, то объекты не будут освобождаться из памяти, а их классом станет класс `NSZombie`, который только выводит сообщение всякий раз, когда оно посылается ему.

Но найти объект — это лишь полдела. Необходимо также выяснить причину его преждевременного освобождения из памяти. Зачастую для этого достаточно просмотреть соответствующие исходные файлы и проверить в них все отправки сообщений `-retain` и `-release`. Правда, статический анализатор, входящий в состав компилятора Clang, делает это автоматически. А если и это не поможет, то можно попытаться переопределить методы `-retain` и `-release` в соответствующем классе, добавив точку прерывания на них в отладчике и затем выяснив, где именно объект действительно сохраняется, а где освобождается из памяти.

Если вы не пользуетесь автоматическим подсчетом ссылок (ARC), то переход к этому механизму может помочь вам в отладке программы. В этом случае обязанность отслеживать указатели возлагается на компилятор, который

обычно справляется с ней лучше, неоднократно применяя соответствующие шаблоны. А если вы уже пользуетесь механизмом ARC, то можете попробовать сделать небезопасные несохраняемые указатели нестрогими. Этим гарантируется, что они будут обнуляться при освобождении из памяти объектов, на которые указывают, а кроме того, помогает надежно проверить действительность указателей.

Наблюдение за исключениями

```
3  int throw(void)
4  {
5      [NSException raise: @"ExampleException"
6                      format: @"watch this"];
7  }
```

Пример кода из файла `throw.m`

В документации на класс `NSException` можно найти следующее описание метода `-raise`:

“Все остальные методы, возбуждающие исключение, вызывают этот метод, поэтому установите в данном месте точку прерывания, если отлаживаете исключения”.

Это было справедливо до появления версии Mac OS X 10.5 и остается в силе для GNUstep. Но, к сожалению, в последующих версиях Mac OS X это положение потеряло силу, и поэтому для отладки исключений требуется дополнительно создавать отдельную категорию, предупреждающую о поведении, связанном с генерированием исключений.

Ныне в разных частях среды Foundation на платформе Mac OS X, отвечающих за генерирование исключений, непосредственно используется оператор `@throw`. Это несколько облегчает дело, поскольку из данной синтаксической конструкции автоматически вызывается функция `objc_exception_throw()`. Данная функция, в свою очередь, вы-

зывает код разворачивающей библиотеки для выполнения всего кода в блоках `@finally` и поиска подходящего обработчика для исключения.

Если установить точку прерывания на вызове функции `objc_exception_throw()`, то можно перехватить все исключения, независимо от того, как они генерируются. Это позволяет также перехватывать такие синтаксические конструкции, как, например, `@throw @"string";`, которые оказываются работоспособными, хотя и не поддерживаются на платформе Mac OS X официально.

```
1 (gdb) break [NSException raise]
2 Breakpoint 1 at 0x51eb851eb14a14
3 (gdb) r
4 Starting program: a.out
5 2010031114:52:54.519 a.out[39782:a0f] ***
   Terminating app due to uncaught exception '
   ExampleException', reason: 'watch this'
6   terminate called after throwing an instance of '
   NSException'
7 Program received signal SIGABRT, Aborted.
8 0x00007fff81c3afe6 in __kill ()
9 (gdb) break objc_exception_throw
10 Breakpoint 2 at 0x7fff829660da
11 (gdb) r
12 Starting program: a.out
13 Breakpoint 2, 0x00007fff829660da in
   objc_exception_throw ()
14 (gdb) bt
15 #0 0x00007fff829660da in objc_exception_throw ()
16 #1 0x00007fff81fd0267 in +[NSException raise:
   format:arguments:] ()
17 #2 0x00007fff81fd01f4 in +[NSException raise:
   format:] ()
18 #3 0x0000000100000ea9 in throw () at throw.m:5
19 #4 0x0000000100000eca in main () at throw.m:12
```

Результат отладки кода из файла `throw.m`

Утверждение исключений

```
3 @interface IgnoreAsserts : NSAssertionHandler
4 @end
5 @implementation IgnoreAsserts
6 - (void)handleFailureInFunction: (NSString*)
    functionName
7     file: (NSString*)fileName
8     lineNumber: (NSInteger)line
9     description: (NSString *)format, ...
10 {
11     NSLog(@"Ignoring assert on line %d of %@() in %@",
12         line, functionName, fileName);
13 }
14 @end
15
16 int main(void)
17 {
18     [NSAutoreleasePool new];
19     [[[NSThread currentThread] threadDictionary]
20         setObject: [[IgnoreAsserts new] autorelease]
21         forKey: NSAssertionHandlerKey];
22     NSCAssert(1 == 2, @"This might not be true");
23     return 0;
24 }
```

Пример кода из файла `assert.m`

В стандартной библиотеке C предоставляется макрокоманда `assert()`, выводящая сообщение и вызывающая функцию `abort()`, если предикатный аргумент вычисляется как ложный. А в Objective-C применяются несколько более полезные варианты этой макрокоманды.

В среде Foundation для целей отладки и тестирования кода предоставляются две макрокоманды — `NSAssert()` и `NSCAssert()`. Если вы пользуетесь версией C99 (что настоятельно рекомендуется), то эти макрокоманды доступны с переменным числом аргументов. В противном случае

придется воспользоваться такими их вариантами, как, например, `NSAssert1()`, которые принимают фиксированное число аргументов. Макрокоманда `NSAssert()` обращается к объекту по ссылке `self` и может быть использована только в теле метода на Objective-C. А макрокоманда `NSCAssert()` может быть использована в функциях C. В отличие от своих вариантов в C, эти макрокоманды не прерывают выполнение кода, если утверждение оказывается ложным. Вместо этого они посылают сообщение объекту обработчика утверждений в отдельном потоке. По умолчанию этот объект является экземпляром класса `NSAssertionHandler` и генерирует исключение `NSInternalConsistencyException` по достижении утверждения. Это исключение можно перехватить и возобновить выполнение кода, но зачастую программа завершается аварийно из-за того, что достигнута вершина стека.

Если же вам требуется иное поведение, можете создать собственный подкласс, производный от класса `NSAssertionHandler`, и переопределить один из доступных в нем методов обработки утверждений. В примере кода, приведенном в начале этого раздела, утверждения игнорируются в функциях C, а когда они встречаются, то на экран просто выводится соответствующее сообщение, как показано ниже.

```
1 $ ./a.out
2 a.out[40031:903] Ignoring assert on line 21 of
   main in $@
```

Результат отладки кода из файла `assert.m`

Переопределение этих методов — дело не совсем обычное, но для него имеется ряд веских оснований. Одним из самых очевидных оснований служит выдача сообщений об ошибках. Если в коде встречается утверждение, то это означает, что выполнение программы пошло по неверному пути. В таком случае потребуется собрать сведения о тех

условиях, которые к этому привели, а затем выгрузить их в систему отслеживания программных ошибок вместо того, чтобы доводить до аварийного завершения программы.

Для запрета утверждений можно определить макрокоманду `NS_BLOCK_ASSERTS`. Но это стоит делать только в коде, критичном к производительности. Как правило, лучше затратить немного лишнего времени ЦП, чем оставлять код в неопределенном состоянии.

Вывод отладочных сообщений

```
7 - (void)log
8 {
9     NSLog(@"%s:%d:%s (%@) Log message", __FILE__,
           __LINE__, __PRETTY_FUNCTION__, self);
10 }
```

Пример кода из файла `nslog.m`

В приведенных ранее примерах функция `NSLog()` применялась очень часто. Если прикладная программа выполняется в режиме терминала, то сообщения, передаваемые функции `NSLog()`, направляются в стандартный поток сообщений об ошибках. В графических приложениях на платформе Mac OS X функция `NSLog()` посылает сообщения на системную консоль, тогда как на платформе Microsoft Windows с GNUstep они направляются монитору системных событий.

Функция `NSLog()` выводит формирующую строку, предваряемую датой, временем, наименованием и идентификатором процесса. Если указать в формирующей строке спецификатор формата `%@`, то для получения строкового представления объекта в ней будет использован тот же самый метод, что и в отладчике.

На заметку

В среде GNUstep Additions предоставляется семейство `NSDebugLog()` макрокоманд для вывода сообщений. Эти макрокоманды заключают в оболочку функцию `NSLog()`, но выводят сообщения только в том случае, если в пользовательских настройках по умолчанию установлено конкретное значение. Благодаря этому упрощается включение и выключение режима вывода отладочных сообщений по отдельным аспектам или средам.

Имеется также возможность пользоваться специальными макрокомандами, определяемыми в компиляторе при построении форматирующей строки. В частности, макрокоманда `__FILE__` представляет собой символьную строку C, содержащую имя текущего исходного файла; макрокоманда `__LINE__` — целочисленное значение, обозначающее номер текущей строки кода; а макрокоманда `__PRETTY_FUNCTION__` — расширение символьной строки C, содержащее имя текущего метода или функции. Ниже приведен результат применения этих макрокоманд при отладке кода из файла `nslog.m`.

```
1 ./a.out
2 2010-03-11 15:57:43.200 a.out[40861:903] nslog.m
   :9:[Log log] (<Log: 0x100112b70>) Log message
```

Результат отладки кода из файла `nslog.m`

Динамическая библиотека Objective-C

Как упоминалось в самом начале этой книги, реализация Objective-C состоит из двух частей: компилятора и динамической библиотеки. Компилятор преобразует исходный код программы и динамические его фрагменты в вызовы динамической библиотеки.

Динамическая библиотека написана на C, а это означает, что ее код составляет подмножество Objective-C. Все интерфейсы динамической библиотеки, в том числе и те, что вызываются из кода, генерируемого компилятором, доступны для кода Objective-C.

Все динамические языковые средства Objective-C реализуются динамической библиотекой. Наиболее очевидным тому примером служит отправка сообщений. При отправке сообщения динамическая библиотека отвечает за выбор средств для его обработки.

Но это не единственная обязанность динамической библиотеки. Она обеспечивает также все языковые средства интроспекции. А кроме того, отслеживает метаданные всех классов и протоколов, делая их доступными для программирующих на Objective-C.

Пользуясь механизмом доступа к значениям по ключам, вы косвенно обращаетесь к динамическим функциям Objective-C для обнаружения методов и переменных экземпляра у получателя сообщений.

Отправка сообщений по имени

```
5 id obj = [NSObject new];
6 SEL sel =
7     NSSelectorFromString(@"release");
8 IMP release = [obj methodForSelector: sel];
9 release(obj, sel);
```

Пример кода из файла `methodName.m`

Методы Objective-C, по существу, являются функциями C, вызываемыми через уровень косвенности. Конкретная реализация такого подхода несколько отличается в разных динамических библиотеках. Так, в динамических библиотеках от компаний NeXT и Apple простые отправки сообщений реализуются с помощью функции `objc_msgSend()`.

К сожалению, для отправки сообщений одной этой функции явно недостаточно. Эта функция объявляется как возвращающая указатель типа `id` и готовая взаимодействовать с любым методом, отвечающим аналогичным соглашениям о вызовах. Так, на платформе Darwin/x86 указатели на объекты возвращаются в регистре `eax` центрального процессора. Если вернуть значение с плавающей точкой, то оно будет возвращено в регистре с плавающей точкой. А если вернуть структуру, то для нее будет выделено место в вызывающей части программы, а указатель на него — передан в качестве скрытого аргумента.

Кроме того, нужно не забыть о вызове функции `objc_msgSend_sret()` или `objc_msgSend_fpret()`. Но это крайне неудобно, поскольку означает, что нужно принимать во внимание соглашения о вызовах, принятые на целевой платформе.

В динамических библиотеках GNU дело обстоит намного проще, поскольку в них отсутствует функция `objc_msgSend()`. Вместо этого сначала вызывается функция `objc_msg_lookup()`, а затем происходит обращение по возвращаемому указателю, но после того, как он будет приведен к нужному типу.

Простейший способ сделать отправку сообщений по имени переносимой между динамическими библиотеками — воспользоваться методом `-methodForSelector:` из класса `NSObject`. Этот метод возвращает указатель на функцию, используемую для реализации метода, который можно вызвать в дальнейшем. Он принимает два скрытых аргумента: получатель сообщения и селектор.

Этого должно быть почти достаточно для вызова метода по имени. Недостаёт лишь некоторого механизма для преобразования символьных строк в селекторы. Именно это и делает функция `NSSelectorFromString()`. Она принимает объект типа `NSString` в качестве своего аргумента и возвращает селектор, который можно затем передать методу `-methodForSelector:` в качестве первого аргумента, а возвращаемой функции — в качестве второго аргумента.

Поиск классов по имени

```
6 class nsview = NSClassFromString(@"NSView");
7 if (Nil == nsview)
8 {
9     NSLog(@"Not linked against AppKit");
10 }
11 else
12 {
13     id view = [nsview new];
14     NSLog(@"View: %@", view);
15 }
```

Пример кода из файла `weakClass.m`

К числу обязанностей динамической библиотеки, в частности, относится сохранение таблицы всех загруженных классов. Этой таблицей можно воспользоваться для просмотра каждого класса по перечислению или для поиска классов по имени.

Последнее оказывается очень удобным для реализации *слабых ссылок на классы*. Такие ссылки поддерживаются непосредственно в динамической библиотеке современной (Modern) версии Objective-C от компании Apple, хотя эта поддержка пока еще недоступна через языковые средства, а следовательно, не приносит особой пользы. Слабая ссылка на класс позволяет использовать класс дополнительно.

В частности, можно написать код в интегрированной среде, использующей классы из среды Foundation, а возможно, и классы из среды AppKit, если привязать и ее. Такая интегрированная среда будет действовать везде, где имеется реализация среды Foundation, в то же время выгодно используя средства из среды AppKit, если они доступны. Подобным образом можно также избавиться от зависимости от интегрированных сред сторонних производителей.

```
1 $ gcc -framework Foundation -g -std = c99 weakClass  
   .m && ./a.out  
2 a.out[53194:903] Not linked against Appkit  
3 $ gcc -framework Cocoa -g -std = c99 weakClass.m &&  
   ./a.out  
4 a.out[53184:903] view: <NSview: 0x1001163b0>
```

Результат выполнения кода из файла weakClass.m

Обратите внимание на отличия в результатах выполнения кода из примера, приведенного в начале этого раздела, если привязать динамическую библиотеку из среды Cocoa, а следовательно, из сред AppKit и Foundation, или же только из среды Foundation. Если в динамической библиотеке присутствует класс `NSview`, создается его экземпляр, в противном случае на экран выводится сообщение о том, что среда AppKit не привязана.

Код из данного примера можно также использовать для обратной совместимости со старыми версиями интегрированной среды. Если в результате поиска новых классов с помощью такого механизма получается значение `Nil`,

его можно использовать как признак отсутствия искомого класса в динамической библиотеке.

Проверка реагирования объекта на сообщение

```
12 SEL sel = @selector(delegateMethod:);  
13 if ([delegate respondsToSelector: sel])  
14 {  
15     [delegate delegateMethod: self];  
16 }
```

Пример кода из файла `respondToSelector.m`

В коде Objective-C обычно наблюдается тенденция к использованию *неформальных протоколов*. Они представляют собой группы сообщений, которые могут быть дополнительно реализованы делегатом. Если вы пользуетесь такими протоколами в своем коде, вам придется проверять, действительно ли вызываемый метод реализован в классе, прежде чем отправлять сообщение этому классу.

На заметку

Это не совсем так. Конечно, сообщение можно было бы отправить в блоке `@try` и перехватить исключение, генерируемое в том случае, если объект не отреагирует на вызываемый при этом метод. Но такой способ не рекомендуется, учитывая затраты вычислительных ресурсов на генерирование исключений.

Правда, сделать это совсем не трудно. Класс `NSObject` реализует метод `-respondToSelector:`, проверяющий, способен ли получатель обрабатывать сообщения с заданным селектором.

Этот метод не только реализуется, но и является составной частью класса `NSObject`. Он реализуется и другими корневыми классами. И это очень важно для пересылки сообщений.

В классе `NSObject` этот метод вызывает динамическую функцию `class_respondsToSelector()`. А в других классах его реализация оказывается иной. Если обращаться к данной функции непосредственно, то код программы окажется весьма уязвимым.

Чаще всего требуется выяснить, в состоянии ли объект, которому предполагается отправить сообщение, обработать это сообщение, а не наличие обрабатывающего его метода. Подобные средства можно обнаружить и в других языках программирования, например в C++, но в Objective-C они реализуются на совершенно других принципах.

Когда сообщение `-respondToSelector:` отправляется промежуточному объекту, у реального объекта будет проверено наличие метода для заданного селектора. Объект, заключенный в оболочку промежуточного объекта, может оказаться еще одним промежуточным объектом, направляющим далее запрос.

Соответствие протоколу можно также проверить, отправив классу сообщение `-conformsToProtocol:`. Но в этом случае проверяется лишь следующее: был ли класс объявлен как принимающий указанный протокол. Классы могут принимать протоколы, не реализуя их методы, хотя и не должны бы этого делать. Но чаще всего классы могут реализовать все методы из протокола без явного соответствия этому протоколу.

Пересылка сообщений

```
3 @interface Facade : NSObject
4 {
5     id delegate;
6 }
7 @end
8 @implementation Facade
9 - (id)forwardingTargetForSelector: (SEL)aSel
10 {
11     return delegate;
```

```
12
13 }
14 - (void)forwardInvocation: (NSInvocation*)anInv
15 {
16     [anInv invokeWithTarget: delegate];
17 }
18 @end
```

Пример кода из файла `forward.m`

Сообщения в Objective-C в большей степени подобны сообщениям в Smalltalk, чем функциям C. Им присуще позднее связывание и возможность интроспекции. На самом низком уровне сообщения реализуются с помощью функций C, но это лишь подробности их реализации. Сообщения можно представить себе как одни объекты, передаваемые другим объектам, подвергаемые затем интроспекции и после этого обрабатываемые методами.

Если сообщение посылается объекту, для которого отсутствует соответствующий метод, у такого объекта все же имеется ряд способов для обработки получаемого сообщения. Эти способы были выработаны в ходе эволюции языка Objective-C.

В старом коде Objective-C вызывался метод `-forward::` для объекта. Это был метод с переменным числом аргументов, требовавший, чтобы получателю сообщения была известна структура стекового фрейма. Компания NeXT заменила его на метод `-forwardInvocation::`. Этому методу передавался объект, инкапсулирующий оправку сообщений.

Механизм, реализуемый методом `-forwardInvocation::`, довольно гибок, но он не обеспечивает достаточную производительность. Создание объекта и разрушение стекового фрейма обходится в несколько сотен раз дороже с точки зрения потребляемых вычислительных ресурсов, чем обычная отправка сообщения. А это означает, что пользоваться данным механизмом не стоит — даже в тех случаях, когда производительность не особенно критична.

Если вы пользуетесь механизмом пересылки сообщений, вам придется реализовать метод `-methodSignatureForSelector:`. Этот метод возвращает объект, представляющий структуру фрейма вызова, связанного с сообщением, что и будет использовано в классе `NSInvocation` при создании объекта, инкапсулирующего сообщение.

В Mac OS X 10.5 компания Apple внедрила новый, более быстросействующий механизм пересылки сообщений. Если объект реализует метод `-forwardingTargetForSelector:`, то он будет вызван с селектором отсутствующего метода. В итоге может быть возвращен объект, которому и будет доставлено сообщение.

Подобно шаблону фасада, такой механизм намного упрощает дело. Любой метод, не воспринимаемый объектом, можно быстро передать его делегату. Если же требуется дополнительная обработка до или после вызова метода делегата, достаточно реализовать ее в передаваемом методе, а затем вызвать метод делегата явным образом.

На заметку

Этот новый механизм не поддерживается ни в двухэтапном механизме отправки сообщений, применяющемся в динамической библиотеке GNU, ни в старых версиях динамической библиотеки от компании Apple. Если вы все-таки пользуетесь им, то непременно снабдите его резервным кодом, используя метод `-forwardInvocation:`. А если пользуетесь последней версией динамической библиотеки от компании Apple или GNUStep с неузависимым двоичным прикладным интерфейсом ABI, то можете извлечь дополнительную выгоду, повысив быстросействие кода. В противном случае ваш код будет выполняться медленно, хотя и останется вполне работоспособным.

Поиск классов

```
9  int classCount =
10     objc_getClassList(NULL, 0);
11  class *classList =
```



```
12  calloc(classCount, sizeof(Class));
13  objc_getClassList(classList,
14   classCount);
15  for (int i=0 ; i<classCount ; i++)
16  {
17      [classes addObject:
18       NSStringFromClass(
19        classList[i])];
20  }
21  free(classList);
22  [classes sortUsingSelector:
23   : @selector(compare:)];
```

Пример кода из файла `ClassTree.m`

Найти класс Objective-C можно двумя способами. Если известно имя класса, то для его поиска достаточно вызвать функцию `NSClassFromString()`. Эта функция принимает строковый объект в качестве своего аргумента и возвращает искомый класс, в противном случае — значение `Nil`, если под указанным именем не зарегистрирован ни один из классов.

Если же имя класса неизвестно, то придется прибегнуть к другому способу его поиска. Имя требуемого класса оказывается неизвестным лишь в очень редких случаях. Так, если у вас имеется подключаемая модульная архитектура, можете предоставить пользователю возможность указать загружаемые пакеты, а затем собрать вместе все подклассы, производные от конкретного класса, или же все классы, соответствующие заданному протоколу.

На заметку

Для проверки классов на соответствие протоколу целесообразно пользоваться динамическими функциями вместо отправки сообщения `+conformsToProtocol:` этим классам. Во многих классах для инициализации по требованию служит сообщение `+initialize`. И если отправить им сообщение `+conformsToProtocol:`, то код проверки на соответствие протоколу все равно будет выполнен — даже если класс не используется.

При вызове функции `objc_getClassList()` возвращается массив классов. Эта функция принимает в качестве своих аргументов два указателя: один — на массив, в котором возвращаются искомые классы, а другой — на размер буфера. Если вызвать ее с пустым значением `NULL` в качестве первого аргумента, она не возвратит ни один из классов, а только известит об их количестве.

Получив этот массив, можно организовать циклическое обращение к нему и проинспектировать классы. В примере кода, приведенном в начале этого раздела, выполняется лишь сохранение имен классов в отдельном массиве, где они затем сортируются по порядку. А в следующем разделе будет показано, как извлечь имена классов и вывести на экран переменные экземпляра и методы каждого класса.

Для поиска классов, соответствующих конкретному протоколу, следует вызвать функцию `class_conformsToProtocol()`. Эта функция сообщает, соответствует ли класс заданному протоколу. Если класс соответствует протоколу, то его можно добавить в список загружаемых подключаемых модулей.

Инспектирование классов

```
26 const char *name = [className UTF8String];
27 printf("%s\n\tIvars:\n", name);
28 Class cls = NSClassFromString(className);
29 unsigned int ivarCount;
30 Ivar *ivars =
31     class_copyIvarList(cls, &ivarCount);
32 for (unsigned int i=0 ; i<ivarCount ; i++)
33 {
34     printf("\t\t%s\n", ivar_getName(ivars[i]));
35 }
36 free(ivars);
37 unsigned int methodCount;
38 Method *methods =
```

```
39 class_copyMethodList(cls, &methodCount);
40 printf("%s\n\tMethods:\n", name);
41 for (unsigned int i=0 ; i<methodCount ; i++)
42 {
43     printf("\t\t%s\n",
44         sel_getName(method_getName(methods[i])));
45 }
```

Пример кода из файла `classTree.m`

Получив указатель на класс в результате перечисления всех классов, отправки сообщения `-class` объекту или вызова функции `NSClassFromString()`, можно организовать его инспектирование. В структуре классов содержится немало сведений о них, и все эти сведения доступны для программирующих на Objective-C.

В прежних динамических библиотеках тип `Class` определялся в виде указателя на структуру с помощью выражения **`typedef struct objc_class *Class`** в заголовочном файле. А в более современных системах тип `Class` является непрозрачным, тогда как взаимодействие с ним предполагается с помощью целого ряда открытых функций. Это дает возможность для постепенного развития двоичного прикладного интерфейса ABI. Для того чтобы изменить расположение структуры класса, теперь достаточно внести соответствующие изменения в компилятор и динамическую библиотеку, не затрагивая весь остальной код.

В примере кода из файла `classTree.m`, приведенном в начале этого раздела, на экран выводятся переменные экземпляра и методы из всех классов, доступных в системе. Но, помимо этих сведений о классах, доступно немало других метаданных, в том числе тип переменных экземпляра и их смещение относительно начала расположения объекта в памяти.

Все эти сведения о классах используются в реализации механизма KVC. Как только будет установлено значение ключа, а механизм KVC получит разрешение на прямой доступ

к переменным экземпляра, он попытается сначала найти переменную экземпляра, совпадающую с заданным именем, а затем определить ее тип и требуется ли распаковать соответствующий объект. И наконец, этот механизм добавит смещение переменной экземпляра к указателю на объект и затем выполнит присваивание по полученному в итоге адресу.

Аналогичные действия выполняются по отношению к методам доступа. Эти методы хранят свои имена, типы и указатели. Особое значение имеют сведения о типе метода, поскольку они позволяют сформировать фрейм вызова метода из другого кода. Проверяя типы методов доступа, механизм KVC может сначала привести тип указателя на метод к нужному типу, а затем вызвать его, установив или получив отдельное свойство.

В более современных динамических библиотеках также доступны для сохранения метаданные объявляемых свойств. Практически все, что сообщается о классе, может быть проинспектировано в результате вызова динамических функций. Единственное, чего нельзя сделать, — это извлечь исходный код отдельных методов.

Создание новых классов

```
2 #include <objc/runtime.h>
3
4 id fakeDealloc(id self, SEL _cmd, ...)
5 {
6     NSLog(@"%@ sent a dealloc message", self);
7 }
8
9 void makeIndestructible(id obj)
10 {
11     Class new =
12         objc_allocateClassPair(obj->isa, "Fake", 0);
13     class_addMethod(new, @selector(dealloc),
14         fakeDealloc, "v@:");
```

```
15    objc_registerClassPair(new);  
16    obj->isa = new;  
17 }
```

Пример кода из файла `newClass.m`

Во время выполнения классы создаются очень редко. Тем не менее в динамических библиотеках предоставляется немало возможностей сделать это в динамическом или оптимизированном режиме. Ранее уже было отчасти показано, каким образом это осуществляется в среде Foundation. Механизм наблюдения за значениями по ключам создает новые классы с целью перехватывать вызовы методов доступа.

Но самое главное, что при создании классов во время выполнения демонстрируется один из основополагающих принципов Objective-C: компилятор не совершает ничего чудесного. В коде C можно сделать то же самое, что и средствами компилятора Objective-C. Разумеется, компилятор Objective-C сделает это намного лучше и проще, оптимизировав конечный результат. Если бы это было не так, то проку от Objective-C было бы немного.

Процесс создания нового класса разделяется на три основных этапа. Сначала выделяется память для пары классов, затем добавляются методы, переменные экземпляра, протоколы и, наконец, пара классов регистрируется средствами динамической библиотеки.

На заметку

Термин *пара классов* может показаться на первый взгляд не совсем понятным. Но следует иметь в виду, что каждый класс Objective-C из исходного кода программы компилируется в виде двух классов во время выполнения. Объект является экземпляром класса, а сам класс — экземпляром *метакласса*. Методы экземпляра присоединяются к классу, а методы класса — к метаклассу. Когда же классу отправляется сообщение, метаклас служит для поиска метода, подходящего для вызова.

Выделяя память для пары классов, необходимо предоставить суперкласс, имя нового класса и объем пространства, добавляемого в конце структуры класса. Имя класса должно быть однозначным. Пример кода, приведенный в начале этого раздела, окажется неработоспособным, если вызвать функцию дважды. Это препятствие проще всего обойти, сохраняя счетчик в статической переменной, инкрементируя его всякий раз, когда создается новый класс, и присоединяя его к символьной строке с именем класса.

Дополнительное пространство в конце класса может пригодиться в том случае, если создание класса сочетается с *настройкой указателя isa по адресам*. Изменяя указатель isa на существующий класс, можно добавлять методы, но не переменные экземпляра. Но ведь дополнительное пространство можно создать и в конце структуры класса, сохранив данные именно там. В результате вызова функции `object_getIndexedIvars()` с классом в качестве ее аргумента возвращается адрес начала расположения этих данных.

Описанными выше функциональными возможностями удобно воспользоваться для реализации других объектных моделей. Я, например, воспользовался ими для внедрения прототипных объектно-ориентированных языковых средств в Objective-C. Это позволило добавлять замыкания к единому объекту, применяя их в качестве методов.

Добавление переменных экземпляра

```
2 #include <objc/runtime.h>
3
4 @interface NSObject (Annotated)
5 - (NSString*)annotation;
6 - (void)setAnnotation: (NSString*)aString;
7 @end
8 static char key;
9 @implementation NSObject (Annotated)
10 - (NSString*)annotation
```

```
11 {  
12     return objc_getAssociatedObject(self, &key);  
13 }  
14 - (void)setAnnotation: (NSString*)aString  
15 {  
16     objc_setAssociatedObject(self, &key, aString,  
17                             OBJC_ASSOCIATION_RETAIN);  
18 }  
19 @end  
20 int main(void)  
21 {  
22     [NSAutoreleasePool new];  
23     id obj = [NSDictionary new];  
24     [obj setAnnotation: @"A note"];  
25     NSLog(@"Annotation: %@", [obj annotation]);  
26     return 0;  
27 }
```

Пример кода из файла `associate.m`

Начиная с версии Mac OS X 10.6 в динамической библиотеке Objective-C предоставляется механизм, позволяющий практически добавлять новые переменные экземпляра в класс во время выполнения. Для сохранения ассоциативных ссылок на объекты служат две функции. Эти ссылки являются указателями на объекты, ассоциируемые с объектами после их создания.

После создания объекта добавлять переменные экземпляра нельзя, поскольку его расположение в памяти уже зафиксировано, но в качестве выхода из этого положения можно воспользоваться ассоциативными ссылками. В примере кода, приведенном в начале этого раздела, в классе `NSObject` создается категория, демонстрирующая подобную возможность. В частности, она позволяет связать строковое значение с любым объектом с помощью пары методов.

В отношении функций установки и получения ассоциативных ссылок необходимо обратить внимание на две следующие особенности. Первая из них связана с ключом, который должен быть однозначным значением указателя, которое не используется в каком-нибудь ином коде для обозначения другой ассоциативной ссылки. Для этого проще всего объявить некоторую статическую переменную в заголовочном файле и затем воспользоваться ее адресом.

Вторая особенность связана с правилами ассоциирования. Эти правила отражают правила, установленные для объявляемых свойств. В данном случае динамической библиотеке предписывается сохранить объект в настоящий момент и освободить его из памяти в тот момент, когда его потребуется разрушить.

Ассоциативные ссылки оказываются полезными в нескольких случаях, и особенно при отладке. Если соблюдаются правила сохранения, то ассоциируемый объект будет освобожден из памяти при освобождении связанного с ним объекта. Это, в частности, позволяет воспользоваться блоком для вывода сообщения об освобождении объекта из памяти, не видоизменяя сам освобождаемый объект. Ассоциативные ссылки позволяют также провести без особого труда различие между двумя объектами, добавив примечание, указывающее, где именно они были созданы, с помощью категории, аналогичной той, что демонстрируется в рассматриваемом здесь примере. Следует, однако, иметь в виду, что ассоциативные ссылки обходятся довольно дорого с точки зрения потребляемых вычислительных ресурсов. Чрезмерное их употребление в выходном коде существенно замедляет его выполнение.

Предметный указатель

А

Автоматически освобождаемые
пулы
назначение, 89
создание и удаление, 89
Аргументы командной строки,
получение, 212
Архиваторы
без ключа, 111
с ключом, 111

Б

Библиотеки
Grand Central Dispatch,
285; 301
libdispatch, 285
динамические
GCC, 31; 237
GNU, 28; 234; 237; 326
GNUstep, 31; 44
Objective-C
назначение, 325
современные, от компании
Apple, 63; 237; 328
состав, 25
старые, 20
разворачивающие,
применение, 237
стандартные
C, 212; 249; 320
Foundation, 35
GNUstep Base, 35
Блоки
выполнение в фоновом
режиме, 287
назначение, 72
обращение
к коллекциям, 173
особенности, 277
применение, 72

разновидности, 278
управление памятью, 282
Быстрое перечисление,
применение, 172

Д

Даты
извлечение из символьных
строк, 189
исходные, 184
начала эпохи,
стандартные, 184
определение давности, 188
преобразование, 186
текущие, обнаружение, 184
Делегирование, применение, 118
Динамическая библиотека
Objective-C
добавление переменных
экземпляра, 339
инспектирование классов, 335
назначение, 25; 325
основные классы, 25
отправка сообщений по имени,
326
пересылка сообщений, 332
поиск классов, 327; 333
создание классов, 337
Доступ к значениям по ключам
главное преимущество, 219
методы, 221
механизм, 219
пути к ключам, 225
совместимость, 223

И

Индексные множества
изменяемые
и неизменяемые, 167

- применение, 167
- Интегрированные среды
 - Application Kit, 26
 - Cocoa, 26
 - Foundation, 25
 - Foundation Kit, 26
 - GNUstep, 25
 - GNUstep Additions, 323
 - XCode, 34
- Исключения
 - генерирование и перехват, 236
 - единая модель,
 - применение, 239
 - необрабатываемые,
 - обработка, 239
 - обработка
 - беззатратная, 234
 - во время выполнения, 232
 - структурная, 233
 - управление памятью, 241
 - разворачиваемые и
 - возобновляемые, 243
 - составляющие, 238
- Истекшее время, расчет, 188

К

- Категории
 - назначение, 52
 - объявление, 52
 - расширение класса, 53
- Классы
 - CGFloat, 129
 - LKMessageSend, 114
 - NSApplication, 191
 - NSArchiver, 110
 - NSArgumentDomain, 205
 - NSArray
 - назначение, 165
 - подкласс
 - NSMutableArray, 165
 - TypedArray, 178
 - NSAssertionHandler, 321
 - NSAttributedString, 162
 - NSAutoreleasePool, 89
 - NSBlock, 73
 - NSBundle, 261

- NSCalendar, 187
- NSCell, 119
- NSCharacterSet, 138
- NSCoder, 208
- NSCondition, 273
- NSConditionLock, 273
- NSControl, иерархия, 119
- NSData, 151; 248
- NSDate, 184
- NSDateFormatter, 187; 189
- NSDictionary, 220
- NSDistantObject, 305
- NSError, 246
- NSException, 237; 318
- NSFileHandle, 249; 300
- NSFileManager, 247; 250
- NSFloat, 129
- NSFont, 162
- NSIndexSet, 166
- NSInteger, 128
- NSInvocation, 332
- NSKeyedArchiver, 110
- NSLocale, 213
- NSLock, 268
- NSMutableSet
 - множество с подсчетом,
 - реализация, 169
 - подкласс NSCounted-Set, 169
- NSNetService, 308
- NSNetServiceBrowser, 309
- NSNotificationCenter
 - подкласс NSDistributedNo-
 - tificationCenter, 294
 - центр уведомлений,
 - реализация, 290
- NSNotificationQueue, 293
- NSObject, 47; 113; 221; 315;
 - 327; 329
- NSPropertyListSerialization,
 - 198; 200
- NSProxy, 47
- NSRecursiveLock, 268
- NSRegularExpression, 160
- NSRunLoop, 192; 295
- NSScanner, 137; 190

- NSSet, 168
- NSSocketPortNameServer, 305
- NSStream, 303
- NSString, 135; 146
- NSThread, 264
- NSTimer, 191
- NSUInteger, 129
- NSURL, 310
- NSURLConnection, 310
- NSURLDownload, 310
- NSURLRequest, 310
- NSUserDefaults, 203
- NSValue
 - иерархия, 129
 - подкласс
 - NSDecimalNumber, 135
 - NSNumber, 129
- NSView, 119
- NSWorkspace, 247
- NSZombie, 317
- UIApplication, 191
- базовая структура интерфейса, 47
- изменяемые и неизменяемые, 39
- инициализация, 68
- коллекций, 163
- объявление и создание, 45
- одиночки, 115
- отправка сообщений, 42
- Кластеры классов
 - именованные
 - конструкторы, 122
 - определение, 121
 - подклассификация, 123
- Кодировки
 - символов
 - UTF-8, 141
 - UTF-16, 141
 - преобразование, 150
 - типов
 - назначение, 69
 - особенности, 70
 - получение, 130
 - применение, 71
- Коллекции

- C++, сохранение объектов
 - Objective-C, 181
 - назначение, 163
 - подклассификация, 178
 - поиск объектов, 175
 - разновидности, 163
 - создание, 165
 - сохранение
 - в списках свойств, 194
 - объектов, 168
 - типов, 129
 - циклическое обращение, 171
- Компиляторы
 - Clang, назначение и установка, 32
 - GCC, назначение и установка, 32
- Константы строковые, создание, 142

M

- Массивы
 - изменяемые и неизменяемые, 165
 - индексирование, 166
 - поиск объектов, 175
 - создание, 165
 - циклическое обращение, 172
- Методы
 - KVC, 222
 - доступа, синтезирование, 60
 - класса, определение, 49
 - объявление, 48
 - объявляемые в категории, 52
 - переопределяемые, вызов, 62
 - протокола, 50
 - резервные, 223
 - семейства, 43; 91
 - с переменным числом аргументов, 157; 165

N

- Наблюдение за значениями по ключам
 - механизм, 219

настройка указателя isa по адресам, 230
 регистрация наблюдателя, 227
 Настройки по умолчанию
 система и область, 203
 сохранение
 особенности, 204
 произвольных
 объектов, 208

О

Области файловой системы, 260
 Объекты
 автоматическое
 освобождение, 91
 архивирование, 110
 интроспекция, 69
 исключений, применение, 238
 копирование, 107
 отличие от структур, 65
 ошибок, применение, 246
 промежуточные,
 назначение, 66
 разрушение, 97
 распределенные
 назначение, 305
 применение, 305
 сохранение и освобождение из
 памяти, 76
 Окружение
 внезапное завершение
 процесса, 215
 определение, 209
 переменные, способы
 доступа, 210
 региональные параметры,
 доступ, 213
 Оптимизация,
 преждевременная, 144
 Отладка программ
 вывод отладочных
 сообщений, 322
 инспектирование
 объектов, 315
 наблюдение за
 исключениями, 318

отладчики
 GDB, 313
 LLDB, 314
 ошибки управления памятью,
 выявление, 316
 утверждение исключений, 321
 Очереди работ
 порядок выполнения
 работ, 286
 разновидности, 285
 специальные,
 организация, 287
 Ошибки
 возврат значений, 245
 возникновение, 231
 способы обработки, 232
 управления памятью,
 выявление, 316

П

Пакеты
 локализация, 258
 определение, 257
 разновидности, 257
 Пара классов, определение, 337
 Переменные экземпляра
 закрытые, сокрытие, 46
 защита, 47
 объявление, 54
 Подключение к сети
 загрузка данных по веб-
 адресу, 310
 заключение сокета
 в оболочку, 300
 общий доступ к объектам
 по сети, 305
 поиск одноранговых узлов
 в сети, 307
 установление связи
 с серверами, 303
 Подсчет ссылок
 автоматический, механизм, 79
 ручную, механизм, 76
 переход к механизму ARC, 85
 Потоки
 блокировка

- рекурсивная, 269
 - со взаимным
 - исключением, 268
 - условная, 273
 - области действия
 - конфликтов, 266
 - ожидание по условию, 272
 - синхронизация, 268
 - создание, 264
 - сохранение характерных
 - данных, 270
 - управление
 - приоритетностью, 265
 - Привязки Cocoa,
 - назначение, 226
 - Протоколы
 - NSCoding, 110; 208
 - NSCopying, 108; 170
 - NSFastEnumeration, 172
 - NSLocking, 268
 - NSMutableCopying, 164
 - назначение, 47
 - неформальные, применение, 55; 329
 - применение, 50
 - Пути к ключам
 - назначение, 225
 - преимущество, 226
 - применение, 226
- С**
- Сборка мусора
 - автоматическая, 93
 - режим, 93
 - Свойства
 - атомарные и неатомарные, 57
 - доступные для
 - чтения записи, 58
 - объявляемые, 55
 - определение, 57
 - Селекторы
 - определение, 44
 - применение, 44
 - создание, 44
 - типизированные, 44
 - Символьные строки
 - копирование, 155
 - обрезка, 152
 - получение из чисел, 136
 - посимвольная обработка, 147
 - построение по шаблонам, 157
 - преобразование в числа, 135
 - разделение, 153
 - разновидности, 141
 - синтаксический анализ, 137
 - сопоставление с шаблоном, 160
 - с приписываемыми
 - атрибутами, 162
 - сравнение, 143
 - форматирование, 162
 - Словари
 - накладываемые
 - ограничения, 170
 - обращение, 171
 - определение, 170
 - потоков, 270
 - создание, 170
 - Сообщения
 - определение, 39
 - отправка, 40
 - пересылка, 332
 - реализация, 331
 - Списки свойств
 - определение, 193
 - преобразование
 - форматов, 199
 - синтаксический анализ, 196
 - форматы, 193
 - чтение данных, 197
 - Ссылки
 - ассоциативные,
 - применение, 339
 - владеющие и не владеющие, 77; 91
 - на протокол, получение, 50
 - слабые
 - на классы, реализация, 328
 - обнуление, 99
 - определение, 99
 - применение, 100
 - Стыковочное приведение
 - типов, 95

Т

- Таблицы виртуальных функций, назначение, 21
- Таймеры
 - запуск, 191
 - построение, 191
 - применение, 191
- Типы
 - NSDecimal, 133
 - NSDictionary, 101
 - NSMapTable, 101
 - объявление, 36
 - операций с числами, 132
 - приведение, 36
 - примитивные, 127
 - свойств, 60
 - указателей, приведение, 128
 - упаковка, 129

У

- Уведомления
 - запрашивание, 290
 - механизм, 289
 - обмен между приложениями, 294
 - обычные и распространяемые, доставка, 295
 - отправка, 292
 - постановка в очередь, 293
 - регистрация наблюдателя, 291
- Указатели
 - isa, назначение, 65
 - на методы, 43
 - на объекты
 - определение, 37
 - приведение типов, 37
 - небезопасные
 - несохраняемые, 80
 - строгие и нестрогие, 80
 - типа id, назначение, 37

Ф

- Файлы и каталоги
 - копирование или перемещение файлов, 250

- манипулирование путями к файлам, 253
- поиск файлов, 259
- получение атрибутов файлов, 252
- проверка существования файла или каталога, 256
- чтение файлов, 248
- Форматирующие строки
 - назначение, 157
 - применение, 158
- Форматы
 - DWARF
 - чтение данных, 242
 - экспорт данных, 234
 - ФорматыJSON
 - назначение, 201
 - JSONпреобразование данных, 202

Циклы

- исполнения
 - назначение, 124
 - применение, 124
 - реализация, 124
 - удержание объекта таймера в памяти, 191
 - управление памятью, 124
- Циклысохранения, исключение, 84

Ш**Шаблоны**

- возврата значений
 - ошибок, 245
- выделенного
 - инициализатора, 114
- делегирования, 84; 117
- изменяемого подкласса, 39; 164
- одиночки, 115; 251
- передачи делегата
 - ошибки, 243
- создания объектов в два этапа, 106
- фасада, 119
- цикла исполнения, 124

Я**Язык Objective-C**

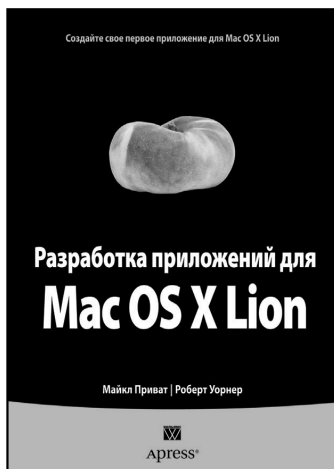
динамическая библиотека, 25
история развития, 27
как надмножество языка C, 23
компиляция программ, 33
межплатформенная
поддержка, 30

объектная модель

обмен сообщениями, 20
понятие, 20
приведение типов
указателей, 21
основные принципы, 19
системы типов, 22

Разработка приложения для MAC OS X LION

*Майкл Приват,
Роберт Уорнер*



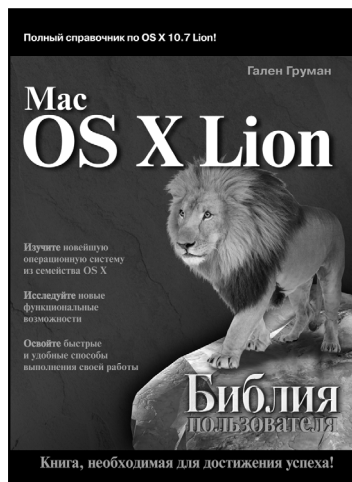
www.williamspublishing.com

Книга посвящена разработке приложений для Mac OS X Lion. Она отличается тем, что все стадии процесса разработки подобных приложений рассматриваются на конкретном, реальном примере создания графопостроительного калькулятора. Прорабатывая материал отдельных глав, читатель получает отличную возможность овладеть принципами разработки приложений для Mac OS X Lion на практике, начиная с замысла проекта, продолжая с построения пользовательского интерфейса, постепенного внедрения новых функциональных возможностей, тестирования, доработки и кончая отправкой приложения в интернет-магазин Mac App Store для выставления на продажу. Книга рассчитана на широкий круг читателей, интересующихся разработкой приложений на платформе Mac OS X Lion.

ISBN 978-5-8459-1789-8 **в продаже**

БИБЛИЯ ПОЛЬЗОВАТЕЛЯ

Гален Груман



www.dialektika.com

Новейшая операционная система Mac в полной мере использует современную технологию мультитач, новый магазин App Store для настольных компьютеров и множество обновлений, учитывающих все самое лучшее из опыта использования устройства iPad. Эта книга охватывает все новые возможности, а также содержит основные сведения о системе Mac для читателей всех уровней подготовки. Начинающие узнают все о системе Mac OS X Lion и способах ее использования, а более продвинутые пользователи смогут погрузиться в изучение советов, приемов работы и профессиональной информации более высокого уровня.

Начинающим библия Mac OS X Lion дает прочные знания о новой операционной системе Mac, а опытным пользователям предоставляет информацию, позволяющую поднять свое мастерство на новый уровень.

ISBN 978-5-8459-1764-5 в продаже

MAC OS X LION ДЛЯ ЧАЙНИКОВ

Боб Ле-Витус



www.dialektika.com

Собираетесь стать укротителем Mac OS X Lion? Новейшая версия операционной системы для компьютеров Mac готова покорно лечь у ваших ног независимо от того, являетесь ли вы новичком или просто обновили предыдущую систему. Благодаря замечательному руководству, написанному экспертом Бобом Ле-Витусом, вы научитесь повелевать компьютерным “зверем”, заставив его выполнять все ваши команды. Основные темы книги:

- настройка клавиатуры, мыши и других устройств;
- навигация с помощью жестов при работе с трекпадом;
- ускорение работы с помощью приложений Mission Control и Launchpad;
- запуск приложений Windows в среде Mac;
- планирование встреч с помощью программы iCal;
- подключение компьютера к сети;
- обслуживание системы и устранение неисправностей.

ISBN 978-5-8459-1767-6

в продаже

MAC OS X LION РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Д.Н. Колисниченко



www.dialektika.com

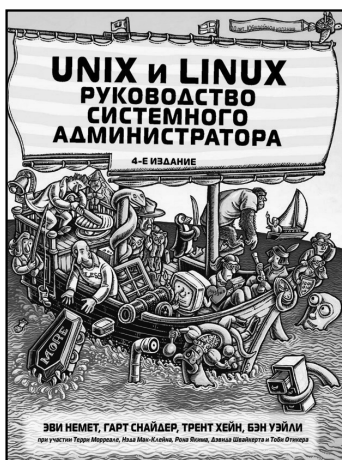
Эта книга — практическое руководство по использованию компьютеров компании Apple операционной системой Mac OS X Lion. В ней рассказывается, как работать в этой ОС, какие возможности она предоставляет пользователю, как ее правильно настроить. Рассмотрены все стандартные приложения Mac OS X, а также другие программы для Mac — пакеты iWork, OpenOffice.org, iLife, браузеры Safari и Opera, программы Apple Mail, iChat и Skype. Подробно обсуждается подключение компьютера к Интернету и развертывание домашней беспроводной сети Wi-Fi. Изложение ориентировано прежде всего на владельцев ноутбуков MacBook, но книга будет полезна и владельцам нетбуков MacBook Air или компьютеров iMac.

ISBN 978-5-8459-1763-8

в продаже

UNIX И LINUX РУКОВОДСТВО СИСТЕМНОГО АДМИНИСТРАТОРА 4-Е ИЗДАНИЕ

**Эви Немет
Гарт Снайдер
Трент Хейн
Бэн Уэйли**



www.williamspublishing.com

В новом издании всемирно известной книги описываются эффективные методы работы и все аспекты управления системами UNIX и Linux, включая управление памятью, проектирование и управление сетями, электронную почту, веб-хостинг, создание сценариев, управление конфигурациями программного обеспечения, анализ производительности, взаимодействие с системой Windows, виртуализацию, DNS, безопасность, управление провайдерами IT-услуг и многое другое. В справочнике описаны современные версии систем UNIX и Linux — Solaris, HP-UX, AIX, Ubuntu Linux, openSUSE и Red Hat Enterprise Linux. Книга будет чрезвычайно полезной всем системным администраторам, а также пользователям систем UNIX и Linux, студентам, преподавателям и специалистам по сетевым технологиям.

ISBN 978-5-8459-1740-9

в продаже