

PROGRAMAÇÃO EM OBJECTIVE-C

GUIA DA BIG NERD RANCH

AARON HILLEGASS & MIKEY WARD



Programação em Objective-C: Guia da Big Nerd Ranch

by Aaron Hillegass and Mikey Ward

Copyright © 2013 Big Nerd Ranch, LLC.

Todos os direitos reservados. Impresso nos Estados Unidos da América. Esta publicação é protegida por direitos autorais, e deve-se obter autorização da editora antes de qualquer reprodução, armazenamento em sistema de busca ou transmissão não autorizada, em qualquer formato ou por qualquer meio, seja eletrônico, mecânico, fotocópia, gravação ou similares. Para informações sobre autorizações, entre em contato com:

Big Nerd Ranch, LLC.
1989 College Ave NE
Atlanta, GA 30317
(404) 478-9005
<http://www.bignerdranch.com/>
book-comments@bignerdranch.com

O logotipo de chapéu com hélice é uma marca registrada da Big Nerd Ranch, LLC.

A distribuição mundial da edição em inglês deste livro é exclusiva da:

Pearson Technology Group 800 East 96th Street Indianapolis, IN 46240 USA <http://www.informit.com>

Os autores e editores declaram ter tomado todas as precauções ao escrever e imprimir este livro, mas ainda assim, não fazem garantias expressas ou implícitas e não se responsabilizam por erros e omissões. Nenhuma responsabilidade será assumida por danos incidentais ou consequenciais, relacionados ou decorrentes do uso das informações ou programas contidos neste livro.

App Store, Apple, Cocoa, Cocoa Touch, Instruments, Interface Builder, iMac, iOS, iPad, iPhone, iTunes, Mac, OS X, Objective-C, PowerBook e Xcode são marcas comerciais da Apple, Inc., registradas nos Estados Unidos e em outros países.

Muitas das designações usadas pelos fabricantes e vendedores para distinguir seus produtos são consideradas marcas comerciais. Quando aplicáveis neste livro, e sempre que o editor estivesse ciente da existência de registro de marca comercial, as designações foram impressas com letras maiúsculas ou em caixa alta.

ISBN-10 032194206X
ISBN-13 978-0321942067

Segunda edição. Novembro de 2013

Acknowledgments

É uma grande honra poder trabalhar com pessoas tão extraordinárias. Muitas delas dedicaram muito tempo e muita energia para fazer com que este livro ficasse excelente. Gostaríamos de aproveitar este momento para agradecê-las.

- Os outros instrutores que ministram aulas sobre o Objective-C nos forneceram inúmeras sugestões e correções. São eles Scott Ritchie, Bolot Kerimbaev, Christian Keur, Jay Campbell, Juan Pablo Claude, Owen Mathews, Step Christopher, TJ Usiyan e Alex Silverman.
- Sarah Brown, Sowmya Hariharan, Nate Chandler e James Majors gentilmente nos ajudaram a encontrar e corrigir falhas.
- Nossa brilhante editora, Susan Loper, transformou meu monólogo reflexivo, que reuniu tudo o que um programador precisa saber, em um manual acessível ao leitor.
- Ellie Volckhausen fez o projeto da capa.
- Chris Loper, da IntelligentEnglish.com, fez o projeto e produziu a versão impressa e as versões para EPUB e Kindle.
- A incrível equipe da Pearson Technology Group nos orientou com muita paciência por todo o processo de publicação do livro.

Table of Contents

I. Introdução	1
1. Você e este livro	3
C e Objective-C	3
Como este livro funciona	4
Como é a vida de um programador	4
2. Seu primeiro programa	7
Instalação das ferramentas da Apple para desenvolvedores	7
Introdução ao Xcode	7
Onde começo a escrever os códigos?	9
Como executo meu programa?	12
Então, o que é um programa?	14
Não pare	14
II. Como funciona a programação	17
3. Variáveis e tipos	19
Tipos	19
Um programa com variáveis	20
Desafio	22
4. if/else	23
Variáveis booleanas	24
Quando as chaves são opcionais	24
else if	25
Para os mais curiosos: operadores condicionais	25
Desafio	26
5. Funções	27
Quando devo usar uma função?	27
Como escrever e usar uma função?	27
Como as funções trabalham em conjunto	29
Bibliotecas padrão	30
Variáveis locais, frames e a pilha	30
Escopo	32
Recursão	33
Analizando frames no depurador	35
Retorno	36
Variáveis globais e estáticas	37
Desafio	38
6. Strings de formatação	41
Utilização de tokens	41
Sequências escape	42
Desafio	42
7. Números	43
Inteiros	43
Tokens para a exibição de inteiros	44
Operações com inteiros	45
Números de ponto flutuante	47
Tokens para exibição de números de ponto flutuante	47
A biblioteca matemática	47
Desafio	48
Uma observação sobre os comentários	48
8. Loops	49
O loop while	50
O loop for	51
break	51
continue	52
O loop do-while	53
Desafio: contagem regressiva	54

Desafio: entrada do usuário	54
9. Endereços e ponteiros	57
Obtenção de endereços	57
Armazenamento de endereços em ponteiros	58
Obtenção de dados em um endereço	58
Quantos bytes?	59
NULL	59
Declarações de ponteiro avançadas	60
Desafio: quanto de memória?	61
Desafio: quanto de faixa?	61
10. Passagem por referência	63
Escrevendo funções de passagem por referência	64
Evitando a desreferenciação de NULL	65
Desafio	65
11. Structs	67
Desafio	69
12. O heap	71
III. Objective-C e Foundation	75
13. Objetos	77
Objetos	77
Classes	77
Criação de seu primeiro objeto	78
Métodos e mensagens	79
Envios de mensagem	79
Outra mensagem	80
Métodos de classe vs. métodos de instância	81
Envio de mensagens ruins	82
Uma observação sobre terminologia	84
Desafio	84
14. Mais mensagens	85
Uma mensagem com um argumento	85
Vários argumentos	86
Aninhamento de envios de mensagens	87
alloc e init	87
Envio de mensagens para nil	88
id	89
Desafio	89
15. Objetos e memória	91
Sobre ponteiros e seus valores	91
Gerenciamento de memória	93
ARC	93
16. NSString	97
Criação de instâncias de NSString	97
Métodos NSString	97
Referências de classe	98
Outras partes da documentação	103
Desafio: encontrar mais métodos NSString	104
Desafio: usar readline()	104
17. NSArray	105
Criação de arrays	105
Acesso aos arrays	106
Iteração em arrays	109
NSMutableArray	110
Métodos antigos de array	111
Desafio: uma lista de itens de mercearia	112
Desafio: nomes interessantes	112
18. Sua primeira classe	115
Métodos acessores	118

Convenções de nomenclatura de acessores	119
self	119
Múltiplos arquivos	119
Prefixos de classe	120
Desafio	120
19. Propriedades	121
Declaração de propriedades	121
Atributos de propriedade	122
Notação de ponto	123
20. Herança	125
Sobrescrevendo métodos	128
super	129
Hierarquia de herança	130
description e %@	131
Desafio	132
21. Propriedades e variáveis de instância de objeto	133
Propriedade de objetos e ARC	134
Criação da classe BNRAAsset	135
Adição de um relacionamento to-many (para vários) à classe BNREmployee	136
Desafio: carteira de ações	139
Desafio: remoção de ativos	139
22. Extensões de classe	141
Ocultamento de mutabilidade	142
Cabeçalhos e heranças	143
Cabeçalhos e variáveis de instância geradas	143
Desafio	143
23. Impedindo vazamentos de memória	145
Ciclos de referências fortes	147
Referências fracas	149
Zerando referências fracas	149
Para os mais curiosos: história da contagem de referência manual e da ARC	151
Regras de contagem de retenção	152
24. Classes de coleção	155
NSSet/NSMutableSet	155
NSDictionary/NSMutableDictionary	157
Objetos imutáveis	160
Classificação de arrays	160
Filtragem	161
Coleções e propriedades	162
Tipos primitivos em C	163
Coleções e nil	163
Desafio: leitura	163
Desafio: principais holdings	163
Desafio: holdings classificados	163
25. Constantes	165
Diretivas de pré-processador	166
#include e #import	166
#define	167
Variáveis globais	167
enum	168
#define vs. variáveis globais	170
26. Gravação de arquivos com NSString e NSData	171
Gravação de uma NSString em um arquivo	171
NSError	172
Leitura de arquivos com NSString	174
Gravação de um objeto NSData em um arquivo	174
Leitura de um objeto NSData em um arquivo	175
Localização de diretórios especiais	176

27. Callbacks	179
O loop de execução	179
Destino-ação	180
Objetos auxiliares	183
Notificações	186
Qual usar?	187
Callbacks e propriedade de objetos	188
Para os mais curiosos: como os seletores funcionam	188
28. Blocos	191
Utilização de blocos	192
Declaração de uma variável de bloco	192
Composição de um bloco	192
Passagem em um bloco	193
typedef	195
Blocos vs. outros callbacks	196
Mais sobre os blocos	196
Valores de retorno	196
Blocos anônimos	196
Variáveis externas	197
Desafio: um bloco anônimo	198
Desafio: usando um bloco com o NSNotificationCenter	199
29. Protocolos	201
Chamada de métodos opcionais	203
30. Listas de propriedades	205
Desafio	207
IV. Aplicativos orientados a eventos	209
31. Seu primeiro aplicativo para iOS	211
Aplicativos baseados em GUI	211
Introdução ao iTahDoodle	211
BNRAppDelegate	213
Modelo-Visão-Controlador (MVC)	214
O delegate do aplicativo	216
Configuração de visões	217
Execução do simulador de iOS	219
Conexão do botão	220
Conexão da visão de tabela	222
Salvamento e carregamento de dados	225
Adição de uma função auxiliar em C	226
Salvamento de dados de tarefa	226
Carregamento de dados de tarefas	227
Para os mais curiosos: o que dizer sobre main()?	227
Para os mais curiosos: execução do iTahDoodle em um dispositivo	227
32. Seu primeiro aplicativo em Cocoa	229
Introdução ao TahDoodle	230
Configuração de visões no Interface Builder	231
Configuração do botão	232
Configuração da visão de tabela	234
Adição de restrições de autolayout	236
Criação de conexões	239
File's Owner	239
Definição do par destino-ação do botão	239
Conexão da visão de tabela	241
Implementação de NSTableViewDataSource	243
Salvamento e carregamento de dados	244
Desafio	245
V. Objective-C avançado	247
33. init	249
Escrevendo métodos init	249

Um método init básico	250
instancetype	250
Utilização e verificação do inicializador da superclasse	250
Métodos init que utilizam argumentos	251
Utilização de acessores	252
Vários inicializadores	253
Métodos init fatais	256
34. Mais sobre propriedades	257
Mais sobre os atributos de propriedade	257
Mutabilidade	257
Especificadores de ciclo de vida	257
Recomendações sobre atômico vs. não atômico	259
Implementação de métodos acessores	259
35. Codificação de chave-valor	261
Tipos não-objetos	262
Percursos de chaves	263
36. Observação de chave-valor	265
Utilização do contexto na KVO	266
Disparo explícito das notificações	266
Propriedades dependentes	267
37. Categorias	269
Desafio	270
VI. C avançado	271
38. Lógicas binárias	273
Operador binário OR	273
Operador binário AND	275
Outros operadores binários	275
Exclusive-OR	276
Complemento	276
Deslocamento à esquerda	277
Deslocamento à direita	277
Utilização de enum para definir máscaras de bits	278
Mais bytes	278
Desafio	278
39. Strings em C	279
char	279
char *	280
Strings literais	282
Conversão de e para NSString	283
Desafio	284
40. Arrays em C	285
Desafio	287
41. Execução a partir da linha de comando	291
Argumentos de linha de comando	292
Execução mais conveniente a partir da linha de comando	294
42. Instruções switch	297
Appendix: O tempo de execução do Objective-C	299
Instrospecção	299
Pesquisa dinâmica e execução de método	299
Gerenciamento de classes e herança de hierarquias	300
Como KVO funciona	303
Observações finais	306
Desafio: variáveis de instância	306
Próximos passos	307
Index	309

Part I

Introdução

1

Você e este livro

Vamos falar sobre você durante alguns instantes. Você deseja escrever aplicativos para iOS ou OS X, mas, ultimamente, não tem praticado muito (ou nada) a programação. Seus amigos têm falado com muito entusiasmo sobre outros livros da Big Nerd Ranch (como *iOS Programming: The Big Nerd Ranch Guide* [Programação para iOS: Guia da Big Nerd Ranch] e *Cocoa Programming for Mac OS X* [Programação em Cocoa para Mac OS X]), mas eles foram escritos para programadores experientes. O que você deve fazer?

Ler este livro.

Há livros similares, mas este é um dos que você deve ler. Por quê? Há muito tempo temos ensinado as pessoas a escrever aplicativos para iOS e Mac, e identificamos o que você precisa saber nesta fase de sua jornada. Trabalhamos muito para captar esse conhecimento e descartar o resto. Há muito conhecimento neste livro.

Nossa abordagem é um pouco incomum. Em vez de simplesmente tentar fazer com que você entenda a sintaxe do Objective-C, mostraremos como funciona a programação e o que os programadores experientes pensam a respeito dela.

Devido a essa abordagem, falaremos sobre algumas ideias complexas no início do livro. Não espere que esta leitura seja fácil. Além disso, quase todas as ideias são acompanhadas por um exercício de programação. Essa combinação de adquirir conceitos e imediatamente colocá-los em prática é a melhor maneira de aprender a programar.

C e Objective-C

Quando você executa um programa, um arquivo é copiado do sistema de arquivos para a memória (RAM), e as instruções desse arquivo são executadas por seu computador. Essas instruções são incompreensíveis aos humanos. Desse modo, os humanos escrevem programas de computador em uma linguagem de programação. A linguagem de programação de nível mais baixo é chamada de *código assembly*. No código assembly, você descreve cada etapa que a CPU (o cérebro do computador) deve realizar. Esse código é então transformado em *código de máquina* (a língua nativa do computador) por um *assembler* (montador).

A linguagem Assembly é cansativamente prolixo e dependente da CPU (porque o cérebro de seu novo iMac pode ser muito diferente do cérebro de seu amado e tão usado PowerBook). Em outras palavras, se você quiser executar o programa em um tipo diferente de computador, será necessário reescrever o código assembly.

Para fazer com que o código possa ser facilmente movido de um tipo de computador para outro, desenvolvemos “linguagens de alto nível”. Com as linguagens de alto nível, em vez de pensar em uma CPU específica, você poderá expressar as instruções de uma forma geral, e um programa (chamado *compilador*) transformará esse código em código de máquina altamente otimizado e específico à CPU. Uma dessas linguagens de alto nível é a C. Os programadores de C escrevem o código na linguagem C, e um compilador de C converte esse código em código de máquina.

A linguagem C foi criada no início dos anos 70 na AT&T. O sistema operacional Unix, que é a base do OS X e do Linux, foi escrito em C com um pouco de código assembly para operações de níveis muito baixos. O sistema operacional Windows também é escrito, sobretudo, em C.

A linguagem de programação Objective-C baseia-se em C, mas contém suporte para a programação orientada a objetos. O Objective-C é a linguagem de programação usada para escrever aplicativos para os sistemas operacionais iOS e OS X da Apple.

Como este livro funciona

Neste livro, você verá o suficiente sobre as linguagens de programação C e Objective-C para aprender a desenvolver aplicativos para Mac ou dispositivos com iOS.

Por que vamos lhe ensinar primeiro sobre a linguagem C? Todo programador de Objective-C eficiente precisa ter um conhecimento bastante aprofundado sobre C. Além disso, muitas ideias que parecem complicadas em Objective-C têm raízes muito simples em C. Apresentaremos sempre uma ideia usando C e depois faremos com que você entenda a mesma ideia em Objective-C.

Este livro foi criado para ser lido em frente a um Mac. Você lerá as explicações das ideias e fará exercícios práticos que ilustrarão essas ideias. Esses exercícios não são opcionais. Você não entenderá realmente o conteúdo do livro se não os fizer. A melhor maneira de aprender a programar é digitar códigos, cometer erros de digitação, corrigir esses erros e tornar-se fisicamente familiarizado com os padrões da linguagem. Somente ler os códigos e entender as ideias na teoria não serão ações muito úteis para você e suas habilidades.

Para praticar ainda mais, no final de cada capítulo, você encontrará exercícios chamados *Desafios*. Esses exercícios fornecem prática adicional e farão com que você fique mais confiante com relação ao que acabou de aprender. Sugerimos com veemência que você faça o máximo dos *Desafios* que conseguir.

No final de alguns capítulos, você também verá seções chamadas *Para os mais curiosos*. Essas seções contêm explicações mais aprofundadas sobre os tópicos abordados no capítulo. Elas não são extremamente essenciais para que você chegue aonde deseja, mas esperamos que você ache interessante e útil.

A Big Nerd Ranch conta com um fórum em que os leitores discutem sobre este livro e os exercícios contidos nele. Você o encontrará em: <http://forums.bignerdranch.com/>.

Você considerará este livro e a programação em geral muito mais agradáveis se souber digitar. A digitação, além de ser mais rápida, permite que você olhe para a tela e para o livro em vez de concentrar-se olhando para o teclado. Desse modo, fica muito mais fácil detectar os erros assim que estes ocorrerem. É uma habilidade que será útil para você em toda sua carreira. Há inúmeros programas de prática de digitação disponíveis para o Mac.

Como é a vida de um programador

Ao começar este livro, você decidiu tornar-se um programador. Você deverá saber com o que está se comprometendo.

A vida de um programador é, sobretudo, uma luta sem fim. Resolver problemas em um cenário técnico que está sempre passando por mudanças significa que os programadores estão sempre aprendendo coisas novas. Nesse caso, “aprender coisas novas” é um eufemismo para “lutar contra sua própria ignorância”. Mesmo que um programador esteja apenas corrigindo um bug em um código que utiliza uma tecnologia familiar, muitas vezes, o software que criarmos será tão complexo que simplesmente entender o que está errado pode ser uma tarefa que dure um dia inteiro.

Se você escrever códigos, você lutará. A maioria dos programadores aprende a lutar por horas e horas, dia após dia, sem ficar (muito) frustrada. Essa é outra habilidade que será muito útil para você. Se quiser saber mais sobre a vida dos programadores e sobre projetos modernos de software, recomendamos a leitura do livro *Dreaming in Code* (Sonhando em código) de Scott Rosenberg.

Agora, chegou a hora de se aventurar e escrever seu primeiro programa.

2

Seu primeiro programa

Agora que você já sabe como este livro está organizado, chegou a hora de ver como funciona a programação para Mac e dispositivos iOS. Para isso, você deve:

- instalar as ferramentas da Apple para desenvolvedores
- criar um projeto simples usando essas ferramentas
- explorar como essas ferramentas são usadas para garantir o funcionamento de seu projeto

No final deste capítulo, você terá escrito, com sucesso, seu primeiro programa para Mac.

Instalação das ferramentas da Apple para desenvolvedores

Para escrever aplicativos para OS X (Mac) ou iOS (iPhone e famílias de dispositivos), você usará as ferramentas da Apple para desenvolvedores. O principal aplicativo que você precisará chama-se **Xcode**.

O Xcode só está disponível no Mac (não está disponível no Windows ou Linux), portanto, você precisará de um Mac para trabalhar com este livro. Além disso, este livro é baseado no Xcode 5, que é compatível com o OS X 10.8 (Mountain Lion) e posterior.

Você pode fazer o download gratuitamente da versão mais recente do Xcode no Mac App Store. Talvez você queira arrastar o ícone do Xcode para seu dock, pois ele será muito utilizado.

Introdução ao Xcode

Xcode é o *ambiente integrado de desenvolvimento* da Apple. Tudo o que você precisa para escrever, compilar e executar novos aplicativos está no Xcode.

Uma observação quanto à terminologia: tudo que é executável em um computador é chamado de *programa*. Alguns programas têm interfaces gráficas do usuário; chamamos estes de *aplicativos*.

Alguns programas não têm interface gráfica do usuário e são executados por muitos dias em segundo plano; chamamos estes de *daemons*. A palavra daemons (demônio, em inglês) pode parecer assustadora, mas neste contexto não é. Você provavelmente tem cerca de 60 daemons sendo executados em seu Mac neste momento. Eles ficam aguardando, esperando serem úteis. Por exemplo, um dos daemons em execução em seu sistema é chamado de pboard. Quando você copia e cola algo, o daemon pboard retém os dados que você está copiando.

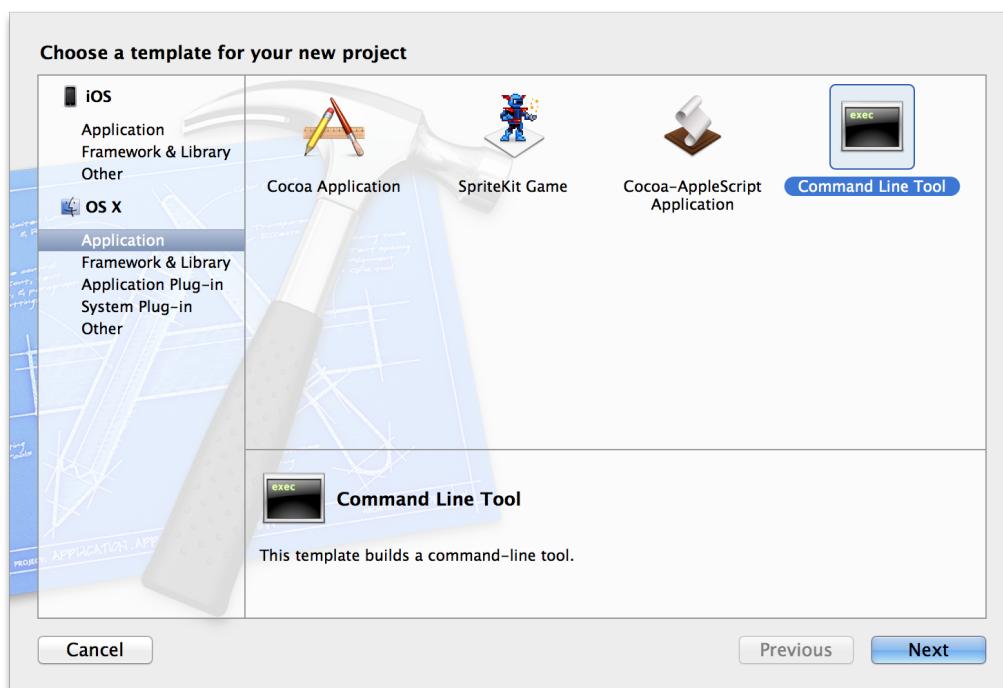
Alguns programas não têm interface gráfica do usuário e são executados por um breve período de tempo no terminal; chamamos estes programas de *ferramentas de linha de comando*. Neste livro, você escreverá principalmente ferramentas de linha de comando para manter o foco nos fundamentos de programação e evitar a distração de criar e gerenciar uma interface de usuário.

Agora, criaremos uma ferramenta de linha de comando simples usando o Xcode, de modo a permitir que você veja como tudo funciona.

Ao escrever um programa, você cria e edita um conjunto de arquivos. O Xcode mantém o controle desses arquivos em um *projeto*. Inicie o Xcode. No menu File, selecione New e, em seguida, Project....

Para ajudar a começar, o Xcode sugere vários templates de projetos. Você escolhe um template dependendo do tipo de programa que deseja escrever. Na coluna à esquerda, selecione Application na seção OS X. Depois, selecione Command Line Tool entre as opções exibidas à direita.

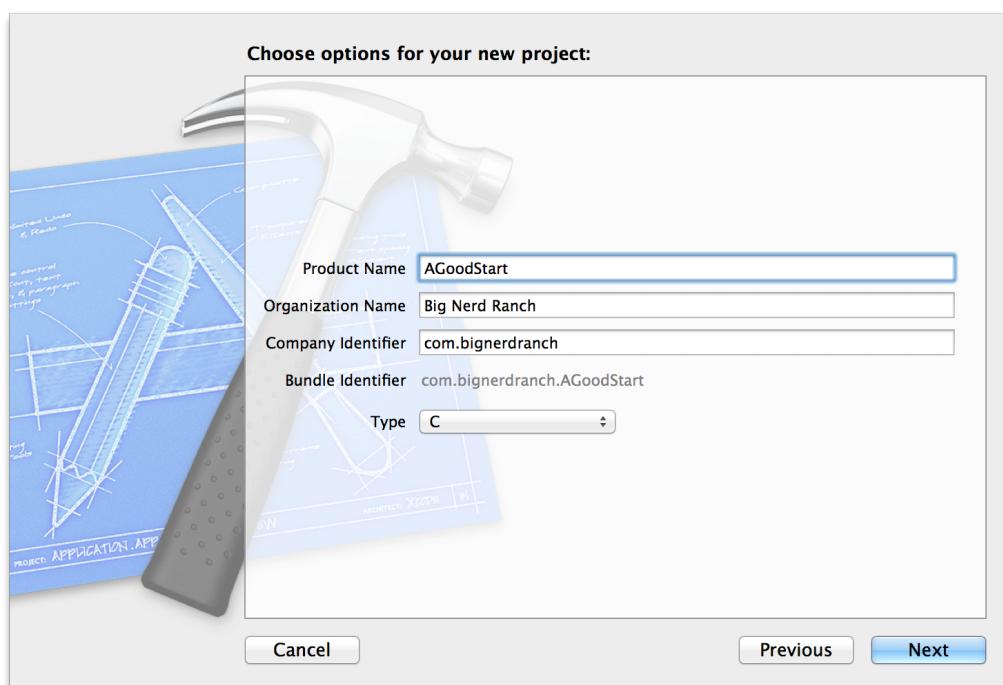
Figure 2.1 Escolha de um template



Clique no botão Next.

Nomeie seu novo projeto como AGoodStart. O nome da organização e o identificador da empresa não serão relevantes para os exercícios neste livro, mas serão necessários para continuar. Use Big Nerd Ranch e com.bignerdranch. No menu pop-up Type, selecione C.

Figure 2.2 Escolha das opções de projeto



Clique no botão Next.

Na próxima janela, escolha a pasta na qual deseja que seu diretório de projeto seja criado. (Se não estiver certo, aceite a localização padrão que o Xcode sugere.) Você não precisará de um repositório para o controle de versão, portanto, desmarque a caixa Create git repository. Por fim, clique no botão Create.

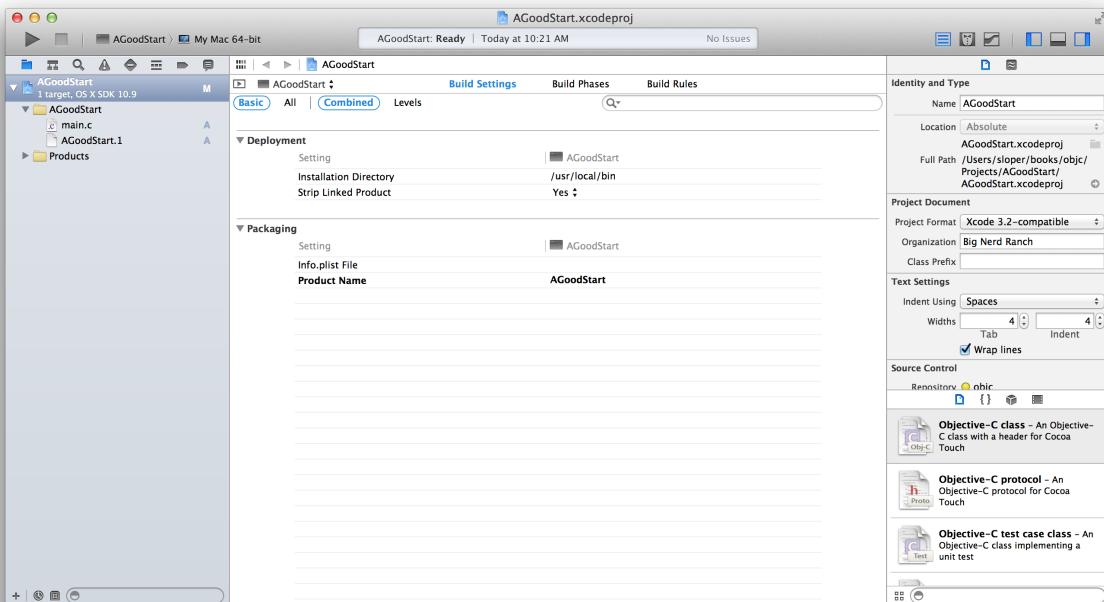
Você criará esse mesmo tipo de projeto em muitos dos próximos capítulos. No futuro, apenas diremos: “Crie uma nova C Command Line Tool chamada *nome-do-programa-aqui*” para que você siga essa mesma sequência.

Por que você está criando projetos em C? O Objective-C baseia-se na linguagem de programação C. Você precisará entender um pouco de C antes de obter detalhes sobre o Objective-C.

Onde começo a escrever os códigos?

Após criar o seu projeto, será exibida uma janela que mostra várias informações sobre o AGoodStart.

Figure 2.3 Primeira visão do projeto AGoodStart



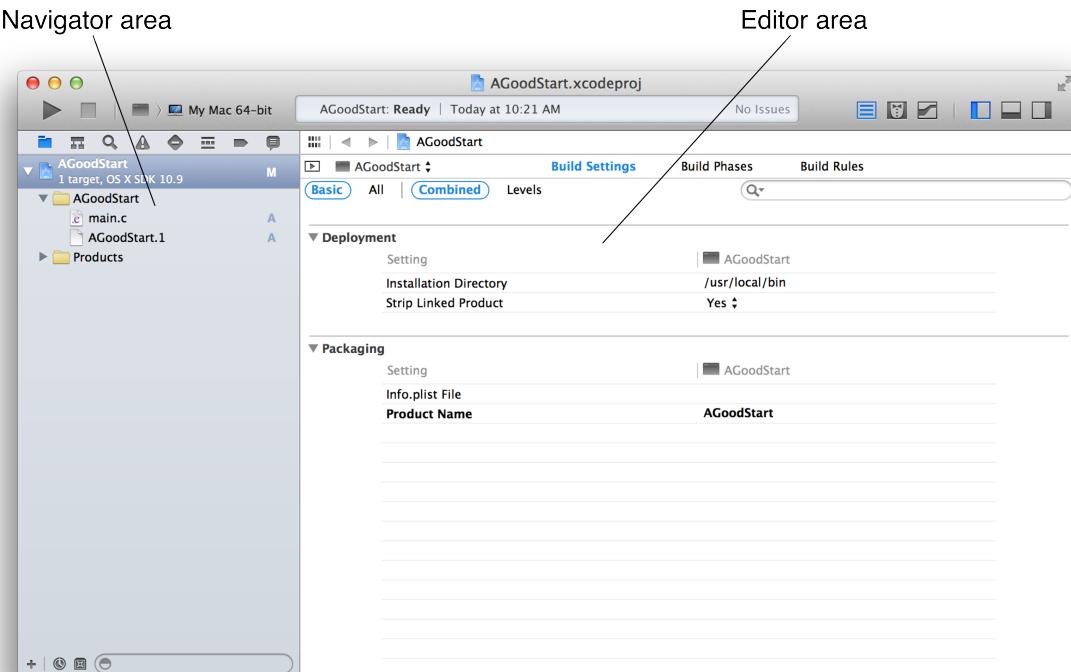
Essa janela possui mais detalhes do que você precisa, então, vamos simplificá-la.

Em primeiro lugar, no canto superior direito da janela, procure três botões que se assemelhem a estes:

Esses botões ocultam e mostram áreas diferentes da janela. Você só precisará da área à direita posteriormente, portanto, clique no botão à direita para ocultá-la.

Você agora tem duas áreas à sua disposição: a *área de navegação* à esquerda e a *área de edição* à direita.

Figure 2.4 Áreas de navegação e edição no Xcode

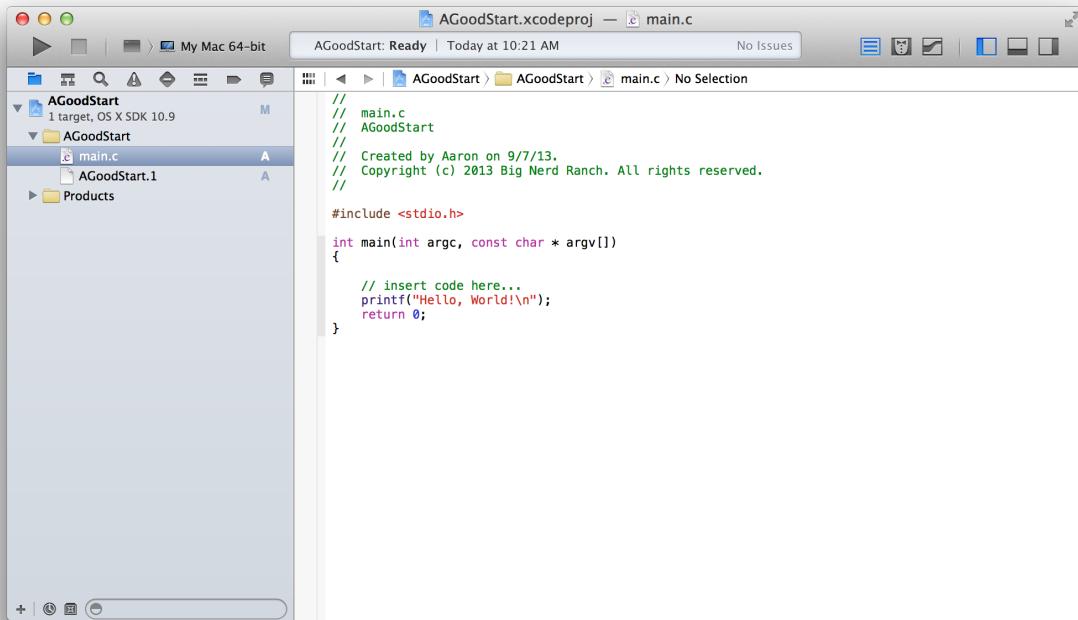


A área de navegação mostra o navegador atual. Existem vários navegadores, e cada um fornece uma maneira diferente de examinar o conteúdo do seu projeto. Você está olhando para o *navegador de projetos*. Esse navegador lista os arquivos que compõem seu projeto.

No navegador de projetos, procure um arquivo chamado `main.c` e clique nele. (Se você não vir o `main.c`, clique no triângulo, próximo da pasta `AGoodStart` para exibir seu conteúdo.)

Ao selecionar o `main.c` no navegador de projetos, a área de edição muda para exibir o conteúdo desse arquivo (Figure 2.5).

Figure 2.5 Seleção do main.c no navegador de projetos



O arquivo `main.c` contém uma *função* chamada `main`. Uma função é uma lista de instruções para o computador executar, e cada função tem um nome. Em um programa em C ou Objective-C, `main` é o nome da função que é chamada quando um programa é iniciado pela primeira vez.

```
#include <stdio.h>

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

    // insert code here...
    printf("Hello, World!\n");
    return 0;
}
```

Essa função contém os dois tipos de informações que você escreve em um programa: código e comentários.

- Código é o conjunto de instruções que dizem ao computador para fazer algo.
- Os comentários são ignorados pelo computador, mas utilizados pelos programadores para documentar o código que escreveram. Quanto mais difícil for o problema de programação que você está tentando resolver, maior será a utilidade dos comentários em ajudar a documentar o modo como você resolveu o problema. A importância desse documento se torna aparente quando você retorna ao trabalho alguns meses depois, olha para o código que esqueceu de comentar e pensa: “Sei que esta solução foi muito boa, mas esqueci completamente como ela funciona.”

Em C e Objective-C, há duas maneiras de distinguir os comentários do código:

- Se você inserir // em uma linha de código, tudo o que vier depois dessas barras até o final da linha será considerado um comentário. Você pode ver isso sendo usado no comentário “insert code here...” (insira o código aqui) da Apple.
- Caso você tenha observações mais extensas, poderá usar /* */ para marcar o início e o final dos comentários que ocupam mais de uma linha.

Essas regras para indicar os comentários fazem parte da *sintaxe* do C. Sintaxe é o conjunto de regras que controla como o código deve ser escrito em uma determinada linguagem de programação. Essas regras são extremamente específicas e, se você não as seguir, seu programa não funcionará.

Embora a sintaxe no que se refere a comentários seja muito simples, a sintaxe de código pode variar muito dependendo da atuação do código. Mas, há um recurso que permanece consistente: toda *instrução* é finalizada com ponto e vírgula. (Você verá exemplos de instruções de código daqui a alguns instantes.) Se você esquecer um ponto e vírgula, ocorrerá um erro de sintaxe, e seu programa não funcionará.

Felizmente, o Xcode conta com maneiras de avisar você sobre esses tipos de erros. Na verdade, um dos primeiros desafios que você enfrentará como programador será interpretar o que o Xcode informa a você quando algo está errado, e depois corrigir seus erros. Você verá algumas das respostas do Xcode aos erros de sintaxe comuns conforme avançarmos no conteúdo deste livro.

Vamos fazer algumas alterações no main.c. Primeiro, você precisa inserir um pouco de espaço. Localize as chaves ({ e }) que marcam o início e o fim da função **main**. Depois, exclua tudo o que estiver entre elas.

Agora substitua o conteúdo da função **main** com o conteúdo exibido abaixo. Você adicionará um comentário, duas instruções de código e outro comentário. Não se preocupe se não entender o que está digitando. A ideia é começar. Você tem um livro inteiro pela frente para aprender o que tudo isso significa.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Print the beginning of the novel
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");
    /* Is that actually any good?
       Maybe it needs a rewrite. */

    return 0;
}
```

Observe que o novo código que você precisa digitar é mostrado com fonte em negrito. O código que não está em negrito é o código que já está lá e lhe mostrará onde adicionar o novo código. Esta é uma convenção que usaremos no restante deste livro.

Ao digitar, talvez você note que o Xcode tenta fazer sugestões úteis. Esse recurso é chamado *preenchimento de código* e é bastante útil. Talvez você queira ignorá-lo por enquanto e concentrar-se em digitar os códigos você mesmo. Entretanto, conforme for avançando neste livro, comece a usar o preenchimento de código para ver como ele pode ajudá-lo a escrever códigos de maneira mais conveniente e exata.

(Você pode ver e configurar as diferentes opções de preenchimento de código nas preferências do Xcode. Selecione o Xcode → Preferences e, em seguida, abra as preferências do Text Editing.)

Além disso, o Xcode usa diferentes cores de fonte para facilitar a identificação de comentários e partes diferentes de seu código. Por exemplo, os comentários são sempre verdes. Depois de um certo tempo trabalhando com o Xcode, você começará a notar instintivamente quando algo estiver errado com as cores. Normalmente, isso é uma dica de que você cometeu um erro de sintaxe. E, quanto mais rápido souber que cometeu um erro, mais fácil será localizá-lo e corrigi-lo.

Como executo meu programa?

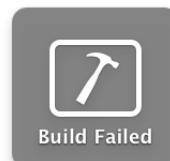
Chegou a hora de executar seu programa e ver o que ele faz. Esse é um processo de duas etapas. O Xcode *compila* seu programa e depois o *executa*. Ao compilar seu programa, o Xcode prepara seu código para ser executado. Isso inclui verificar a sintaxe e outros tipos de erros.

No canto superior esquerdo da janela do projeto, localize o botão que se parece com o botão Play no iTunes ou em um reprodutor de DVD. Se você mantiver o cursor sobre esse botão, verá uma dica de ferramenta que diz: **Build and then run the current scheme.** (Compile e execute o esquema atual). Clique nesse botão.

Se tudo correr bem, você verá:



Caso contrário, você verá:



O que fazer se isso ocorrer? Compare atentamente seu código com o código do livro. Procure erros de digitação e verifique se há ponto e vírgula faltando. O Xcode destacará as linhas que considerar problemáticas. Depois que você encontrar e corrigir o programa, clique novamente no botão Run. Repita esse procedimento até obter uma compilação bem-sucedida.

(Não desanime quando ocorrer falhas na compilação deste código ou de qualquer código que você escrever futuramente. Cometer erros e corrigi-los ajuda a entender o que se está fazendo. Na verdade, é bem melhor do que ter sorte e acertar logo na primeira vez.)

Após sua compilação ser bem-sucedida, uma nova área será exibida na parte inferior da janela (Figure 2.6). A metade à direita desta área é o *console*. O console mostra o resultado do seu código sendo executado:

Figure 2.6 Resultado no console na parte inferior direita

The screenshot shows the Xcode interface with the following details:

- Project Navigator:** Shows the project "AGoodStart" with one target "AGoodStart" using "OS X SDK 10.8".
- Editor Area:** Displays the code for "main.c" with the following content:

```

// main.c
// AGoodStart
//
// Created by Aaron on 10/12/13.
// Copyright (c) 2013 Big Nerd Ranch. All rights reserved.

#include <stdio.h>

int main(int argc, const char * argv[])
{
    // Print the beginning of the novel
    printf("It was the best of times.\n");
    printf("It was the worst of times.\n");
    /* Is that actually any good?
     * Maybe it needs a rewrite. */

    return 0;
}

```
- Console Area:** Shows the output of the program execution:

```

It was the best of times.
It was the worst of times.
Program ended with exit code: 0

```
- Status Bar:** Shows "Finished running AGoodStart : AGoodStart" and "No Issues".

Então, o que é um programa?

Agora que você já compilou e executou seu primeiro programa, vamos dar uma olhada em como ele funciona.

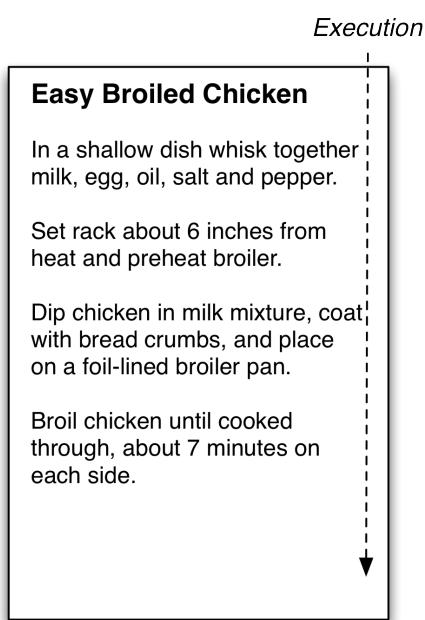
Um programa é uma coleção de funções. Uma função é uma lista de operações que o processador deve executar. Toda função tem um nome, e a função que você acabou de escrever chama-se `main`.

Quando os programadores falam sobre funções, nós normalmente incluímos um par de parênteses vazios. Assim, a função `main` é referenciada como `main()`.

Existia outra função em seu programa – `printf()`. Você não escreveu essa função, mas utilizou-a.

Para um programador, escrever uma função é muito parecido com o processo de escrever uma receita. Assim como uma função, um encarte de receita tem um nome e um conjunto de instruções. A diferença é que você executa uma receita, e o computador executa uma função.

Figure 2.7 Um encarte de receita chamado Easy Broiled Chicken (Frango assado simples)



Essas instruções de cozimento estão em inglês. Na primeira parte deste livro, suas funções serão escritas na linguagem de programação C. No entanto, o processador de um computador espera suas instruções em código de máquina. Como você faz isso?

Quando você escreve um programa em C (que é relativamente agradável para você), o *compilador* converte as funções de seu programa em código de máquina (que é agradável e eficiente para o processador). O próprio compilador é um programa executado pelo Xcode quando você clica no botão Run. Compilar um programa é o mesmo que construir um programa, e usaremos esses termos indistintamente.

Ao executar um programa, as funções compiladas são copiadas do disco rígido para a memória, e a função chamada `main` é executada pelo processador. Geralmente, a função `main` chama outras funções. Por exemplo, sua função `main` chamou a função `printf`. Você aprenderá mais sobre como as funções funcionam no Chapter 5.

Não pare

A essa altura, provavelmente você já lidou com diversas frustrações: problemas de instalação, erros de digitação, e um grande e novo vocabulário. E, talvez, nada do que você fez até agora pareça ter sentido. Isso é completamente normal.

Otto, o filho de Aaron, tem seis anos. Otto fica confuso muitas vezes ao longo do dia. Constantemente, ele tenta absorver conhecimentos que não se ajustam à estrutura mental dele. A confusão ocorre com tanta frequência,

que isso já não o aborrece mais. Ele não para de se questionar: “Por que é tão confuso? Devo jogar este livro fora?”

Conforme vamos ficando mais velhos, ficamos confusos com menos frequência – não porque sabemos tudo, mas porque temos a tendência de desviar das coisas que nos deixam confusos. Por exemplo, ler um livro de história pode ser muito agradável porque adquirimos muito conhecimento que nos faz desprender de nossa estrutura mental. Esse é um aprendizado fácil.

Aprender um novo idioma é um exemplo de aprendizado difícil. Você sabe que há milhões de pessoas que falam facilmente esse idioma, mas ele parece incrivelmente estranho e complicado para você. E, quando as pessoas falam com você usando esse idioma, você sempre se confunde.

Aprender a programar um computador também é um aprendizado difícil. De vez em quando, você se confunde – principalmente no começo. Sem problemas. Na verdade, é um pouco engraçado. É como se voltássemos a ter seis anos novamente.

Aproveite este livro ao máximo; prometemos que toda confusão desaparecerá antes de chegar à última página.

Part II

Como funciona a programação

Nestes próximos capítulos, você criará muitos programas que demonstram conceitos úteis. Esses programas de linha de comando não servirão para impressionar seus amigos, mas você experimentará uma certa sensação de domínio ao executá-los. Você está deixando de ser um usuário de computador para se tornar um programador de computador.

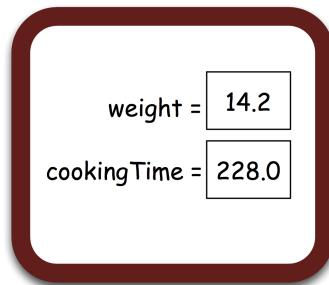
Seus programas nestes capítulos serão escritos em C. Observe que este livro não tem a intenção de abordar a linguagem C em detalhes. Mas sim o contrário: aprimorado através de anos de ensino, este é o subconjunto essencial de informações sobre programação e programação em C que pessoas iniciantes nesse campo precisam saber antes de aprender sobre programação em Objective-C.

3

Variáveis e tipos

Continuando com a metáfora da receita do último capítulo, muitas vezes, um chef terá um pequeno quadro na cozinha para armazenar dados. Por exemplo, ao desembalar um peru, ele vê uma etiqueta que diz “14,2 lb.”. Antes de jogar a embalagem fora, ele escreve “peso = 14,2” no quadro. Depois, antes de colocar o peru no forno, ele calculará o tempo de cozimento (15 minutos + 15 minutos por libra), baseando-se no peso anotado no quadro.

Figure 3.1 Controle dos dados usando um quadro



Durante a execução, um programa sempre precisa de locais para armazenar dados a serem usados posteriormente. Um local em que uma parte dos dados pode ser colocada é conhecida como *variável*. Cada variável tem um nome (como `cookingTime`) e um *tipo* (como um número). Além disso, quando o programa é executado, a variável terá um valor (como 228,0).

Tipos

Em um programa, você cria uma nova variável ao *declarar* o seu tipo e nome. Este é um exemplo de uma declaração de variável:

```
float weight;
```

O tipo dessa variável é `float` (que definiremos daqui a pouco), e o nome dela é `weight`. Neste ponto, a variável não tem um valor.

Em C, você deve declarar o tipo de cada variável por dois motivos:

- O tipo permite que o compilador verifique seu trabalho por você e alerte-o quanto a possíveis erros ou problemas. Por exemplo, digamos que você tenha uma variável de um tipo que contém texto. Se solicitar seu logaritmo, o compilador dirá a você algo do tipo: “não faz sentido solicitar o logaritmo dessa variável”.
- O tipo informa ao compilador a quantidade de espaço na memória (quantos bytes) que deve ser reservada para essa variável.

Aqui temos uma visão geral dos tipos mais comumente usados. Retornaremos a cada um dos tipos em mais detalhes nos próximos capítulos.

<code>short, int, long</code>	Esses três tipos são números inteiros; eles não exigem uma casa decimal. Um <code>short</code> geralmente tem menos bytes de armazenamento que um <code>long</code> , e um <code>int</code> está entre os dois tipos. Portanto, você pode armazenar um número muito maior em um <code>long</code> do que em um <code>short</code> .
<code>float, double</code>	Um <code>float</code> é um número de ponto flutuante: um número que pode ter uma casa decimal. Na memória, um <code>float</code> é armazenado como uma mantissa e um expoente. Por exemplo, 346,2 é representado como $3,462 \times 10^2$. Um <code>double</code> é um número de precisão dupla, que geralmente tem mais bits para conter uma mantissa maior, bem como expoentes maiores.
<code>char</code>	Um <code>char</code> é um inteiro de um byte que nós geralmente tratamos como um caractere, como a letra 'a'.
<code>ponteiro</code>	Um ponteiro contém um endereço de memória. Ele é declarado com o uso de um asterisco. Por exemplo, uma variável declarada como <code>int *</code> pode conter um endereço de memória em que um <code>int</code> é armazenado. Ela não contém o valor real do número, mas se você souber o endereço do <code>int</code> , então será possível obter facilmente seu valor. Os ponteiros são muito úteis, e falaremos mais sobre eles posteriormente. Muito mais.
<code>struct</code>	Um <code>struct</code> (ou <i>estrutura</i>) é um tipo composto por outros tipos. Você também pode criar novas definições de <code>struct</code> . Por exemplo, imagine que você quisesse um tipo <code>GeoLocation</code> que contivesse dois membros <code>float</code> : <code>latitude</code> e <code>longitude</code> . Nesse caso, você definiria um tipo <code>struct</code> .

Esses são os tipos que um programador de C usa diariamente. É muito surpreendente quais ideias complexas podem ser capturadas nessas cinco ideias simples.

Um programa com variáveis

De volta ao Xcode, você criará outro projeto. Primeiro, feche o projeto `AGoodStart` para que você não digite acidentalmente novos códigos no antigo projeto.

Agora, crie um novo projeto (`File → New → Project...`). Esse projeto será uma C Command Line Tool chamada `Turkey`.

No navegador de projetos, localize o arquivo `main.c` desse projeto, abrindo-o em seguida. Edite o `main.c` para que ele corresponda ao código a seguir.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    // Declare the variable called 'weight' of type float
    float weight;

    // Store a number in that variable
    weight = 14.2;

    // Log it to the user
    printf("The turkey weighs %f.\n", weight);

    // Declare another variable of type float
    float cookingTime;

    // Calculate the cooking time and store it in the variable
    // In this case, '*' means 'multiplied by'
    cookingTime = 15.0 + 15.0 * weight;

    // Log that to the user
    printf("Cook it for %f minutes.\n", cookingTime);

    // End this function and indicate success
    return 0;
}
```

(Está se perguntando sobre o `\n` que não para de aparecer em seu código? Você saberá o que ele faz no Chapter 6.)

Compile e execute o programa. Você pode tanto clicar no botão Run no canto superior esquerdo da janela do Xcode quanto utilizar o atalho de teclado Command-R. Seu resultado no console deve ser semelhante ao seguinte:

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

De volta ao seu código, vamos revisar o que você fez. Na linha de código semelhante a:

```
float weight;
```

dizemos que você está “declarando a variável `weight` para que seja do tipo `float`”.

Na próxima linha, suas variáveis obtêm um valor:

```
weight = 14.2;
```

Você está copiando os dados para essa variável. Dizemos que você está “atribuindo um valor de 14,2 a essa variável”.

No C moderno, você pode declarar uma variável e atribuir a ela um valor inicial em uma linha, como a seguir:

```
float weight = 14.2;
```

Veja outra atribuição:

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

O que está do lado direito do `=` é uma *expressão*. Uma expressão é algo que é avaliado e resulta em algum valor. Na verdade, toda atribuição tem uma expressão à direita do `=`.

Por exemplo, nesta linha:

```
weight = 14.2;
```

a expressão é simplesmente 14,2.

Uma expressão pode ter várias etapas. Por exemplo, ao avaliar a expressão `15,0 + 15,0 * weight`, o computador primeiro multiplica `weight` por 15,0 e então adiciona esse resultado a 15,0. Por que a multiplicação vem primeiro? Dizemos que a multiplicação tem *precedência* sobre a adição.

Para mudar a ordem em que as operações geralmente são executadas, você utiliza parênteses:

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

Agora a expressão entre parênteses é avaliada primeiro, de forma que o computador primeiro faz a adição e depois multiplica `weight` por 30,0.

Desafio

Bem-vindo ao seu primeiro desafio!

A maior parte dos capítulos deste livro terminará com um exercício de desafio para você fazer sozinho. Alguns desafios (como o que você está prestes a realizar) são fáceis e proporcionam a chance de praticar a mesma coisa que você fez no capítulo. Outros são mais difíceis e exigem maior resolução de problemas. Fazer esses exercícios ajuda a fixar o que você aprendeu e aumenta sua confiança em suas habilidades. Encorajamos você a encará-los.

(Se você ficar preso ao trabalhar em um desafio, dê uma parada e depois volte e tente de novo descansado. Se não funcionar, acesse o fórum desse livro em forums.bignerdranch.com para obter ajuda.)

Crie uma nova C Command Line Tool chamada `TwoFloats`. Em sua função `main()`, declare duas variáveis do tipo `float` e atribua a cada uma delas um número com uma casa decimal, como 3,14 ou 42,0. Declare outra variável do tipo `double` e atribua a ela a soma dos dois `floats`. Exiba o resultado usando a função `printf()`. Consulte o código neste capítulo caso precise verificar sua sintaxe.

4

if/else

Uma ideia importante na área de programação é realizar diferentes ações dependendo das circunstâncias.

- Todos os campos de cobrança do formulário de pedidos foram preenchidos? Se sim, habilitar o botão Submit.
- O jogador ainda tem vidas restantes? Se sim, retomar o jogo. Se não, exibir a imagem de um túmulo e tocar uma música triste.

Esse tipo de comportamento é implementado usando `if` e `else`, cuja sintaxe é:

```
if (conditional) {
    // Execute this code if the conditional evaluates to true
} else {
    // Execute this code if the conditional evaluates to false
}
```

Você não vai criar um projeto neste capítulo. Em vez disso, analise atentamente os exemplos de código com base no que aprendeu nos últimos dois capítulos.

Veja um exemplo de código usando `if` e `else`:

```
float truckWeight = 34563.8;

// Is it under the limit?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
} else {
    printf("It is a heavy truck\n");
}
```

Se você não tiver uma cláusula `else`, poderá simplesmente deixar esta parte de fora:

```
float truckWeight = 34563.8;

// Is it under the limit?
if (truckWeight < 40000.0) {
    printf("It is a light truck\n");
}
```

A expressão condicional é sempre verdadeira ou falsa. Em C, decidiu-se que 0 indicaria falso, e tudo que fosse diferente de zero seria considerado verdadeiro.

Na expressão condicional do exemplo acima, há um número em cada lado do operador `<`. Se o número da esquerda for menor que o número da direita, a expressão determinará o valor 1 (um modo comum de expressar a condição verdadeira). Se o número da esquerda for maior ou igual ao número da direita, a expressão determinará o valor 0 (o único modo de expressar a condição falsa).

Os operadores geralmente aparecem em expressões condicionais. A Table 4.1 mostra os operadores comuns usados ao comparar números (e outros tipos que o computador determina como sendo números):

Table 4.1 Operadores de comparação

<	O número da esquerda é menor que o número da direita?
---	---

>	O número da esquerda é maior que o número da direita?
<=	O número da esquerda é menor ou igual ao número da direita?
>=	O número da esquerda é maior ou igual ao número da direita?
==	Eles são iguais?
!=	Eles <i>não</i> são iguais?

O operador `==` merece uma observação adicional: em programação, o operador `==` é usado para *verificar a igualdade*. Usamos um único `=` para *atribuir* um valor. Inúmeros bugs provêm de programadores que usam `=` quando pretendiam usar `==`. Então, pare de pensar que `=` é o “sinal de igual”. A partir de agora, ele é “o operador de atribuição”.

Algumas expressões condicionais exigem operadores lógicos. E se você quiser saber se um número está dentro de um determinado intervalo, como maior que zero e menor que 40.000? Para especificar um intervalo, você pode usar o operador lógico AND (`&&`):

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

A Table 4.2 mostra os três operadores lógicos:

Table 4.2 Operados lógicos

<code>&&</code>	AND lógico -- verdadeiro se, e apenas se, ambos forem verdadeiros
<code> </code>	OR lógico -- falso se, e apenas se, ambos forem falsos
<code>!</code>	NOT lógico -- verdadeiro torna-se falso, falso torna-se verdadeiro

(Se você tem conhecimento de outra linguagem, observe que não há OR lógico exclusivo em Objective-C, então, não falaremos sobre isso aqui.)

O operador lógico NOT (`!`) nega a expressão à sua direita.

```
// Is it lighter than air?
if (!(truckWeight > 0.0)) {
    printf("The truck has zero or negative weight. Hauling helium?\n");
}
```

Variáveis booleanas

Como você pode ver, as expressões podem tornar-se muito longas e complexas. Muitas vezes, é útil colocar o valor da expressão em uma variável conveniente e bem nomeada.

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

Uma variável que pode ser verdadeira ou falsa é uma variável *booleana*. Historicamente, os programadores de C têm usado sempre um `int` para conter um valor booleano. Geralmente, os programadores de Objective-C usam o tipo `BOOL` para variáveis booleanas, então, é o que usamos aqui. (`BOOL` é um alias para um tipo inteiro.) Para usar `BOOL` em uma função C, como `main()`, você vai precisar incluir em seu programa o arquivo onde esse tipo é definido:

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

Você vai aprender mais sobre a inclusão de arquivos no próximo capítulo.

Quando as chaves são opcionais

Uma nota de sintaxe: se o código que segue a expressão condicional consistir em apenas uma declaração, então as chaves serão opcionais. Assim, o código a seguir equivale ao exemplo anterior.

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

No entanto, as chaves serão necessárias se o código consistir em mais de uma declaração.

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

Por quê? Imagine se você removesse as chaves.

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

Este código faria com que você não fosse bem-visto pelos motoristas de caminhão. Nesse caso, todo caminhão é apreendido, independentemente do peso. Quando o compilador não encontrar uma chave após a condicional, apenas a próxima declaração será considerada parte da construção `if`. Portanto, a segunda declaração será executada sempre. (E quanto ao recuo da segunda declaração? Embora o recuo seja muito útil para leitores humanos, ele não significa nada para o compilador.)

Neste livro, sempre vamos incluir as chaves.

else if

E se você tiver mais de duas possibilidades? Você poderá testá-las, uma a uma, usando `else if`. Por exemplo, imagine que um caminhão pertença a uma das três categorias de peso: flutuante, leve e pesado.

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

Você pode ter quantas cláusulas `else if` que desejar. Todas serão testadas na ordem em que aparecem até que seja determinado que uma delas é verdadeira. O trecho “na ordem em que aparecem” é importante. Certifique-se de ordenar suas condições para não obter um falso positivo. Por exemplo, se você trocar os dois primeiros testes no exemplo acima, nunca encontrará um caminhão flutuante, pois caminhões flutuantes também são caminhões leves. A cláusula `else final` é opcional, mas é útil quando você deseja obter tudo o que não atendeu às condições anteriores.

Para os mais curiosos: operadores condicionais

É comum usar `if` e `else` para definir o valor de uma variável de instância. Por exemplo, talvez você tenha o seguinte código:

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

Sempre que tiver um cenário onde um valor é atribuído a uma variável com base em uma condicional, você terá um candidato para o *operador condicional*, que é `?`. (Ocasionalmente, você o verá sendo chamado de *operador ternário* porque ele contém três operandos).

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

Essa linha equivale ao exemplo anterior. Em vez de escrever `if` e `else`, você escreve uma atribuição. A parte que vem antes de `?` é a condicional. Os valores após `?` são as alternativas para o caso de a condicional ser considerada verdadeira ou falsa.

Desafio

Considere o seguinte trecho de código:

```
int i = 20;
int j = 25;
int k = ( i > j ) ? 10 : 5;

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

O que será exibido no console?

5

Funções

No Chapter 3, você aprendeu que uma variável é um nome associado com um pedaço (chunk) de dados. Uma função é um nome associado a um pedaço (chunk) de código. Você pode passar informações a uma função. Você pode fazer com que a função execute códigos. Você pode fazer com que uma função retorne informações a você.

As funções são fundamentais para a programação, então, temos muito sobre o que falar neste capítulo – três novos projetos, uma nova ferramenta e muitas novas ideias. Vamos começar com um exercício que demonstrará para que servem as funções.

Quando devo usar uma função?

Suponha que você esteja escrevendo um programa para parabenizar os alunos pela conclusão de um curso da Big Nerd Ranch. Antes de preocupar-se em recuperar a lista de alunos de um banco de dados ou em imprimir os certificados em papel sofisticado da Big Nerd Ranch, você deseja testar a mensagem que será impressa nos certificados.

Crie uma nova C Command Line Tool chamada ClassCertificates. (Selecione File → New → Project... ou use o atalho do teclado Command-Shift-N para começar.)

Sua primeira consideração ao escrever este programa pode ser:

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

    return 0;
}
```

A ideia de digitar tudo isso aborrece você? É desagradavelmente repetitivo? Em caso afirmativo, você tem os atributos de um excelente programador. Quando você se vê repetindo trabalhos que são similares por natureza (neste caso, as palavras na instrução `printf()`), você deseja começar a pensar em uma função que seja a melhor forma de realizar a mesma tarefa.

Como escrever e usar uma função?

Agora que você já percebeu que precisa de uma função, será necessário escrever uma. Abra o `main.c` em seu projeto ClassCertificates e escreva uma nova função chamada `congratulateStudent`. Essa função deve vir antes de `main()` no arquivo.

```
#include <stdio.h>

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

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

(Quer saber o que %s e %d significam? Está confuso com o tipo `char *`? Tenha um pouco de paciência, logo chegaremos lá.)

Agora, edite `main()` para usar sua nova função:

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Kate", "Cocoa", 5);
    congratulateStudent("Bo", "Objective-C", 2);
    congratulateStudent("Mike", "Python", 5);
    congratulateStudent("Liz", "iOS", 5);

    return 0;
}
```

Compile e execute o programa. Localize seu resultado no console. Você pode precisar redimensionar a área da parte inferior. Você pode fazer isso clicando no cabeçalho cinza da área e, em seguida, arrastando-o para ajustar seu tamanho. (Você pode redimensionar qualquer uma das áreas no Xcode da mesma maneira.)

O resultado precisa ser idêntico ao que você veria se tivesse digitado tudo sozinho.

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

Pense sobre o que você fez aqui. Você notou um padrão repetitivo. Você obteve todas as características compartilhadas do problema (o texto repetitivo) e moveu essas características para uma função separada. Restaram as diferenças (nome do aluno, nome do curso, número de dias). Você lidou com essas diferenças ao adicionar três *parâmetros* à função. Observe novamente a linha onde você nomeou a função.

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

Cada parâmetro tem duas partes: o tipo dos dados que o argumento representa e o nome do parâmetro. Os parâmetros são separados por vírgulas e colocados entre parênteses à direita do nome da função.

E a palavra `void` à esquerda do nome da sua função? Este é o tipo de informação que retornou da função. Quando você não tem nenhuma informação para retornar, utiliza a palavra-chave `void`. Falaremos mais sobre *retornos* posteriormente neste capítulo.

Você também usou, ou *chamou*, sua nova função em `main()`. Ao chamar `congratulateStudent()`, você passou valores a ela. Os valores passados a uma função são conhecidos como *argumentos*. O valor do argumento é então atribuído ao nome do parâmetro correspondente. O nome desse parâmetro pode ser usado dentro da função como uma variável que contém o valor passado.

Na sua primeira chamada para **congratulateStudent()**, você passou três argumentos: "Kate", "Cocoa", 5.

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

Por enquanto, concentre-se no terceiro argumento. Quando 5 é passado à **congratulateStudent()**, ele é atribuído ao terceiro parâmetro, `numDays`. Os argumentos e os parâmetros são associados na ordem em que aparecem. Eles também devem ser do mesmo tipo (ou muito próximos ao mesmo tipo). Aqui, 5 é um valor inteiro, e o tipo de `numDays` é `int`.

Agora, quando **congratulateStudent()** usar a variável `numDays` dentro da função, seu valor será 5. Por fim, você pode comprovar que tudo isso funcionou observando a primeira linha do resultado, que exibe corretamente o número de dias.

Volte à primeira versão proposta de `ClassCertificates` com toda a repetitiva digitação. Qual é o propósito de usar uma função? Menos trabalho na digitação? Bem, digamos que sim, mas isso não é tudo. Dividir seu código em funções facilita a realização de alterações e a localização e correção de bugs. Você pode realizar uma mudança ou correção de um erro de digitação em um local, e isso terá efeitos em todos os locais que você chamar essa função.

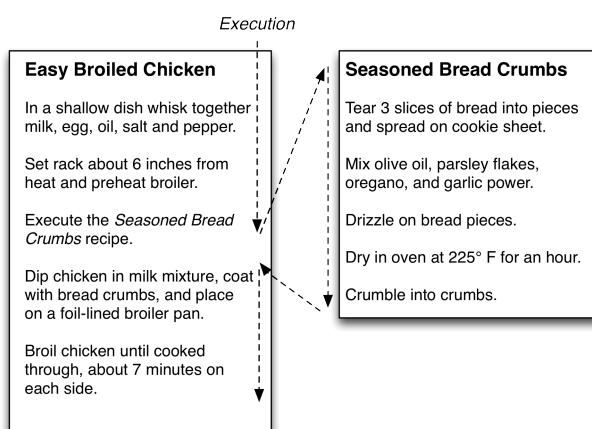
Outro benefício de escrever funções é a questão da capacidade de reutilização. Agora que você já escreveu essa prática função, poderá usá-la em outro programa.

Como as funções trabalham em conjunto

Um programa é uma coleção de funções. Ao executar um programa, essas funções são copiadas do disco rígido para a memória, e o processador localiza e executa a função chamada “main”.

Lembre-se de que uma função é como um encarte de receita. Se você começou a fazer a receita “Easy Broiled Chicken” (Frango assado simples), terá descoberto que a terceira instrução diz: “Execute the Seasoned Bread Crumbs recipe” (Prepare a farinha de pão temperada), que é explicada em outro encarte. Um programador diria: “A função The Easy Broiled Chicken (Frango assado simples) *chama* a função Seasoned Bread Crumbs (Farinha de pão temperada).”

Figure 5.1 Encartes de receita



De modo similar, `main()` pode chamar outras funções. Por exemplo, `main()` no `ClassCertificates` chamou `congratulateStudent()`, que, por sua vez, chamou `printf()`.

Enquanto você estava preparando a farinha de pão, parou de executar as etapas do encarte “Easy Broiled Chicken” (Frango assado simples). Quando a farinha ficar pronta, você retomará o trabalho com o encarte “Easy Broiled Chicken” (Frango assado simples).

De modo semelhante, a função `main` interrompe sua execução e fica “bloqueada” até que seja finalizada a execução da função que ela chamou. Para ver isso acontecer, você chamará uma função `sleep` que não faz

nada além de esperar por alguns segundos. Essa função é declarada no arquivo `unistd.h`. Na parte superior do `main.c`, inclua este arquivo:

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

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

Em sua função `main`, chame a função `sleep` após as chamadas para `congratulateStudent()`.

```
int main (int argc, const char * argv[])
{
    congratulateStudent("Kate", "Cocoa", 5);
    sleep(2);
    congratulateStudent("Bo", "Objective-C", 2);
    sleep(2);
    congratulateStudent("Mike", "Python", 5);
    sleep(2);
    congratulateStudent("Liz", "iOS", 5);

    return 0;
}
```

Compile e execute o programa. Você verá uma pausa de dois segundos entre cada mensagem de congratulações. Isso ocorre porque a função `main` parou de ser executada até que a função `sleep` fosse suspensa.

Bibliotecas padrão

Seu computador vem com muitas funções integradas. Realmente, isso é um pouco ilusório – a verdade é esta: antes de o OS X ter sido instalado em seu computador, não havia nada além de um caro aquecedor. Entre as coisas que foram instaladas como parte do OS X estavam os arquivos que contêm um conjunto de funções pré-compiladas. Essas coleções são chamadas de *bibliotecas padrão*.

Dois dos arquivos que compõem as bibliotecas padrão são `stdio.h` e `unistd.h`. Quando você inclui esses arquivos em seu programa, você pode então usar as funções que eles contêm. `printf()` está no `stdio.h`; `sleep()` está no `unistd.h`.

As bibliotecas padrão têm dois propósitos:

- Elas representam grandes pedaços de código, os quais você não precisa escrever nem manter. Portanto, elas permitem que você crie programas muito maiores e melhores do que conseguiria fazer de outra forma.
- Elas garantem que a maioria dos programas seja muito semelhante.

Os programadores gastam muito tempo estudando as bibliotecas padrão dos sistemas operacionais com que trabalham. Toda empresa que cria um sistema operacional também tem a documentação das bibliotecas padrão que o acompanha. Você verá como navegar pela documentação do iOS e do OS X no Chapter 16.

Variáveis locais, frames e a pilha

Toda função pode ter *variáveis locais*. As variáveis locais são variáveis declaradas dentro de uma função. Elas existem somente durante a execução dessa função e podem ser acessadas apenas dentro dessa função. Por exemplo, considere uma função que calcula o tempo de cozimento de um peru. Ela seria semelhante ao seguinte:

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

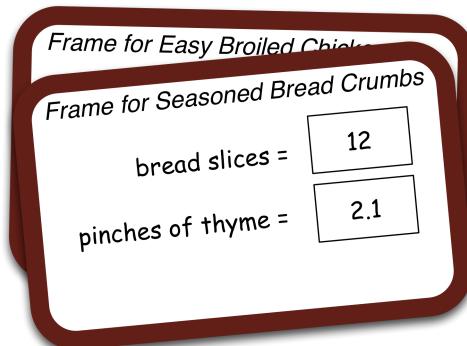
`necessaryMinutes` é uma variável local. Ela será criada quando `showCookTimeForTurkey()` começar a ser executada e deixará de existir assim que essa função finalizar sua execução. O parâmetro da função, `pounds`,

também é uma variável local. Um parâmetro é uma variável local que foi inicializada para o valor do argumento correspondente.

Uma função pode ter várias variáveis locais, e todas elas estão armazenadas no *frame* dessa função. Pense em um frame como se fosse um quadro-negro (lousa) em que você pode rabiscar durante a execução da função. Quando é finalizada a execução da função, a lousa é descartada.

Imagine por um momento que você está fazendo a receita Easy Broiled Chicken (Frango assado simples). Em sua cozinha, cada receita em progresso tem sua própria lousa, portanto, você já tem uma lousa para a receita Easy Broiled Chicken. Agora, quando você chamar a receita Seasoned Bread Crumbs (Farinha de pão temperada), precisará de uma nova lousa. Onde você a colocará? Na parte superior da lousa da receita Easy Broiled Chicken. Afinal de contas, você interrompeu a execução da receita Easy Broiled Chicken para fazer a receita Seasoned Bread Crumbs. Você não precisará do frame de Easy Broiled Chicken até que a receita Seasoned Bread Crumbs seja concluída e que seu frame seja descartado. O que você tem agora é uma pilha de frames.

Figure 5.2 Duas lousas em uma pilha



Os programadores usam a palavra *pilha* para descrever onde os frames são armazenados na memória. Quando uma função é chamada, um frame é enviado para o topo da pilha. Quando é finalizada a execução de uma função, dizemos que ela *retorna*. Isto é, ela desativa seu frame da pilha e permite que a função que o chamou retome a execução.

Vejamos detalhadamente como a pilha funciona, colocando **showCookTimeForTurkey()** em um programa. Crie uma nova C Command Line Tool chamada TurkeyTimer. Edite main.c para que se assemelhe ao seguinte:

```
#include <stdio.h>

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

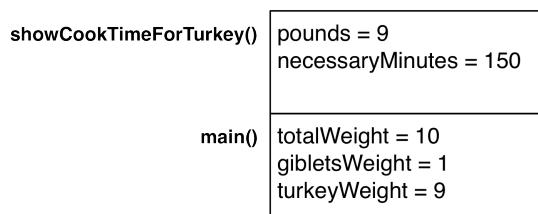
int main(int argc, const char * argv[])
{
    int totalWeight = 10;
    int gibletsWeight = 1;
    int turkeyWeight = totalWeight - gibletsWeight;
    showCookTimeForTurkey(turkeyWeight);
    return 0;
}
```

Compile e execute o programa. Você deve ver o seguinte resultado:

Cook for 150 minutes.

Lembre-se que **main()** é sempre executada primeiro. **main()** chama **showCookTimeForTurkey()**, que começa a ser executada. Como ficará, então, a pilha desse programa logo depois que **necessaryMinutes** for calculado?

Figure 5.3 Dois frames na pilha



A pilha é a última a entrar e a primeira a sair. Ou seja, o frame de `showCookTimeForTurkey()` é desativado da pilha antes do frame de `main()` ser desativado da pilha.

Observe que `pounds`, o único parâmetro de `showCookTimeForTurkey()`, faz parte do frame. Lembre-se de que um parâmetro é uma variável local à qual foi atribuído o valor do argumento correspondente. Para este exemplo, a variável `turkeyWeight` com um valor de 9 é passada como um argumento para `showCookTimeForTurkey()`. Em seguida, esse valor é atribuído ao parâmetro `pounds`, isto é, ele é copiado no frame da função.

Escopo

Em uma definição de função, qualquer par de chaves { ... } define o *escopo* do código que está entre elas. Uma variável não pode ser acessada fora do escopo em que foi declarada. Na verdade, ela não existe fora do escopo em que foi declarada.

Todos os pares de chaves, sejam elas parte de uma definição de função, uma instrução `if` ou um loop, definem seu próprio escopo que restringe a disponibilidade de quaisquer variáveis declaradas neles.

Adicione o seguinte código à sua função `showCookTimeForTurkey`:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
    if (necessaryMinutes > 120) {
        int halfway = necessaryMinutes / 2;
        printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
    }
}
```

Compile e execute o programa.

A instrução `printf` neste exemplo pode acessar variáveis que estão no escopo definido pelas chaves da instrução `if`, como `halfway`. Ela também pode ter acesso a variáveis no escopo externo definido pela própria função `showCookTimeForTurkey`, como `necessaryMinutes`.

Agora mova a chamada `printf` para fora do escopo da instrução `if`:

```
void showCookTimeForTurkey(int pounds)
{
    int necessaryMinutes = 15 + 15 * pounds;
    printf("Cook for %d minutes.\n", necessaryMinutes);
    if (necessaryMinutes > 120) {
        int halfway = necessaryMinutes / 2;
        printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
    }
    printf("Rotate after %d of the %d minutes.\n", halfway, necessaryMinutes);
}
```

Compile e execute o programa novamente. O programa não será executado, e um erro de compilação será exibido: Use of undeclared identifier 'halfway'. Fora do escopo da instrução `if`, a variável `halfway` não existe. Programadores avançados diriam que: “Quando a chamada `printf()` é feita, a variável `halfway` se desprende do escopo.”

Recursão

Uma função pode chamar ela mesma? Com certeza! Chamamos isso de *recursão*. Há uma música extremamente repetitiva chamada “99 Bottles of Beer” (99 garrafas de cerveja). Crie uma nova C Command Line Tool chamada `BeerSong`. Abra o `main.c` e adicione uma função para escrever a letra dessa música e depois inicie-a em `main()`:

```
#include <stdio.h>

void singSongFor(int numberOfBottles)
{
    if (numberOfBottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n\n");
    } else {
        printf("%d bottles of beer on the wall. %d bottles of beer.\n",
               numberOfBottles, numberOfBottles);
        int oneFewer = numberOfBottles - 1;
        printf("Take one down, pass it around, %d bottles of beer on the wall.\n\n",
               oneFewer);
        singSongFor(oneFewer); // This function calls itself!

        // Print a message just before the function ends
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               numberOfBottles);
    }
}

int main(int argc, const char * argv[])
{
    // We could sing 99 verses, but 4 is easier to think about
    singSongFor(4);
    return 0;
}
```

Compile e execute o programa. O resultado será semelhante ao seguinte:

```
4 bottles of beer on the wall. 4 bottles of beer.  
Take one down, pass it around, 3 bottles of beer on the wall.
```

```
3 bottles of beer on the wall. 3 bottles of beer.  
Take one down, pass it around, 2 bottles of beer on the wall.
```

```
2 bottles of beer on the wall. 2 bottles of beer.  
Take one down, pass it around, 1 bottles of beer on the wall.
```

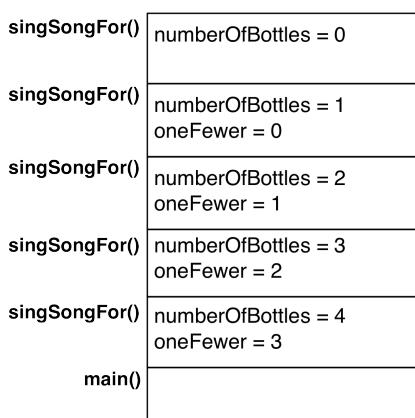
```
1 bottles of beer on the wall. 1 bottles of beer.  
Take one down, pass it around, 0 bottles of beer on the wall.
```

There are simply no more bottles of beer on the wall.

```
Put a bottle in the recycling, 1 empty bottles in the bin.  
Put a bottle in the recycling, 2 empty bottles in the bin.  
Put a bottle in the recycling, 3 empty bottles in the bin.  
Put a bottle in the recycling, 4 empty bottles in the bin.
```

Como ficará a pilha quando a última garrafa for retirada da prateleira, mas nenhuma for colocada para reciclagem?

Figure 5.4 Frames na pilha de uma função recursiva



Confuso? Veja o que aconteceu:

- **main()** chamou **singSongFor(4)**.
- **singSongFor(4)** exibiu um verso e chamou **singSongFor(3)**.
- **singSongFor(3)** exibiu um verso e chamou **singSongFor(2)**.
- **singSongFor(2)** exibiu um verso e chamou **singSongFor(1)**.
- **singSongFor(1)** exibiu um verso e chamou **singSongFor(0)**.
- **singSongFor(0)** exibiu “não há mais garrafas de cerveja na prateleira” e retornou.
- **singSongFor(1)** retomou a execução, exibiu a mensagem de reciclagem e retornou.
- **singSongFor(2)** retomou a execução, exibiu a mensagem de reciclagem e retornou.
- **singSongFor(3)** retomou a execução, exibiu a mensagem de reciclagem e retornou.
- **singSongFor(4)** retomou a execução, exibiu a mensagem de reciclagem e retornou.
- **main()** retomou, retornou e o programa foi finalizado.

Geralmente, o tópico sobre frame e pilha não é abordado em um curso de programação para iniciantes, mas achei que as ideias seriam extremamente úteis para os novos programadores. Primeiro, esses conceitos lhe dão uma compreensão mais concreta das respostas às perguntas, tais como “O que acontece com minhas variáveis locais quando é finalizada a execução da função?” Segundo, eles ajudam você a entender o *depurador*. O depurador é um programa que ajuda a entender o que seu programa realmente está fazendo, que, por sua vez, ajuda você a localizar e corrigir “bugs” (problemas em seu código). Ao compilar e executar um programa no Xcode, o depurador é *anexado* ao programa, de modo que você possa usá-lo.

Analizando frames no depurador

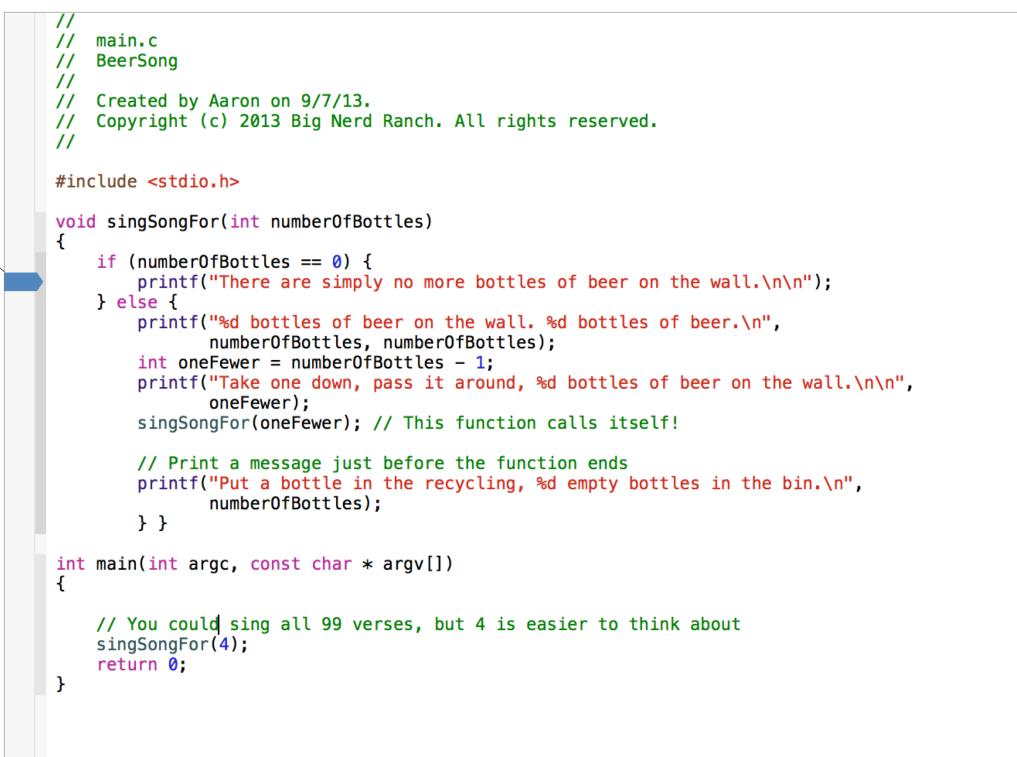
Você pode usar o depurador para navegar pelos frames na pilha. No entanto, para fazer isso, é preciso parar seu programa no meio da execução. Caso contrário, `main()` finalizará a execução, e não haverá frames para analisar. Para ver o máximo de frames possível em nosso programa BeerSong, interrompa a execução na linha que diz: “There are simply no more bottles of beer on the wall.” (Não há mais garrafas de cerveja na prateleira).

Como você faz isso? No `main.c`, localize a linha

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

Há duas colunas cinzas à esquerda do seu código. Clique na coluna mais larga, cinza claro, próxima a essa linha de código para definir um *ponto de interrupção*.

Figure 5.5 Definição de um ponto de interrupção



```

// main.c
// BeerSong
//
// Created by Aaron on 9/7/13.
// Copyright (c) 2013 Big Nerd Ranch. All rights reserved.
//

#include <stdio.h>

void singSongFor(int number_of_bottles)
{
    if (number_of_bottles == 0) {
        printf("There are simply no more bottles of beer on the wall.\n\n");
    } else {
        printf("%d bottles of beer on the wall. %d bottles of beer.\n",
               number_of_bottles, number_of_bottles);
        int oneFewer = number_of_bottles - 1;
        printf("Take one down, pass it around, %d bottles of beer on the wall.\n",
               oneFewer);
        singSongFor(oneFewer); // This function calls itself!

        // Print a message just before the function ends
        printf("Put a bottle in the recycling, %d empty bottles in the bin.\n",
               number_of_bottles);
    }
}

int main(int argc, const char * argv[])
{
    // You could sing all 99 verses, but 4 is easier to think about
    singSongFor(4);
    return 0;
}

```

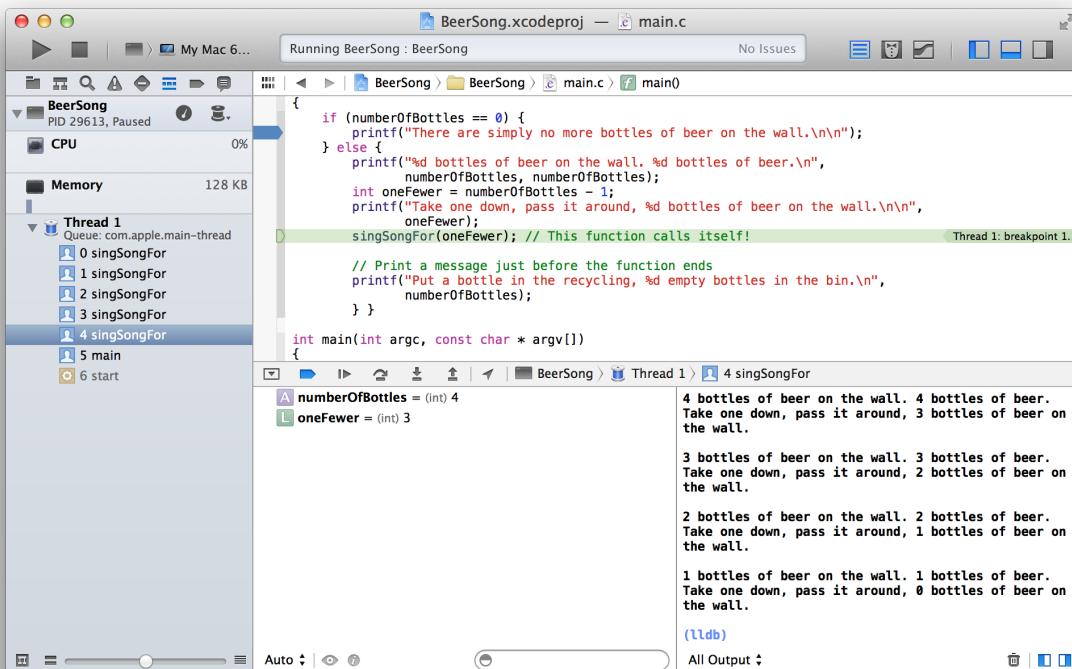
Um ponto de interrupção é um local no código em que você deseja que o depurador pause a execução de seu programa. Execute o programa novamente. Você pode ver a partir do resultado no console que seu programa foi interrompido (ou “falhou”) bem antes de executar a linha onde definiu o ponto de interrupção.

O programa está temporariamente congelado, e você pode examiná-lo mais detalhadamente. A área de navegação foi alternada para exibir o *navegador de depuração*, que mostra todos os frames atualmente na pilha, também chamado de *rastreamento de pilha*.

No rastreamento de pilha, os frames são identificados pelo nome de sua função. Como seu programa consiste quase que totalmente em uma função recursiva, esses frames têm o mesmo nome, e você deve distingui-los pelo valor de `oneFewer` que é passado a eles. Na parte inferior da pilha está o frame de `main()`.

Você pode selecionar um frame da pilha para ver as variáveis nesse frame e o código-fonte da linha de código que atualmente está sendo executada. Selecione o frame para a primeira vez em que `singSongFor()` é chamada.

Figure 5.6 Seleção do frame de `singSongFor(4)`

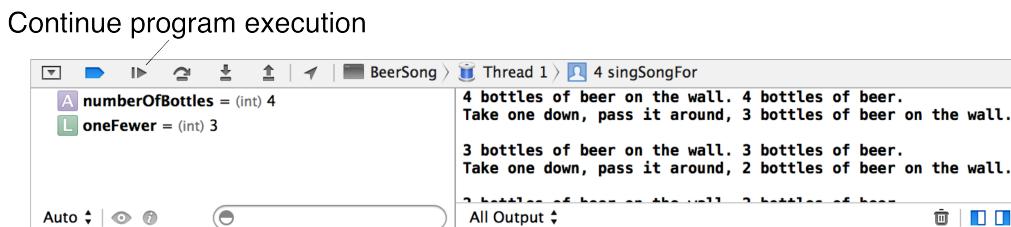


Você pode ver as variáveis desse frame e seus valores na área inferior esquerda do console. Essa área é chamada de *visão de variáveis*.

Agora, você precisa remover o ponto de interrupção para que o programa seja executado normalmente. Clique com o botão direito no indicador azul e selecione Delete Breakpoint.

Para retomar a execução de seu programa, clique no botão `>>` na barra cinza acima da visão de variáveis.

Figure 5.7 Reiniciando BeerSong



Nós demos apenas uma rápida olhada no depurador aqui para demonstrar como funcionam os frames. No entanto, usar o depurador para configurar pontos de interrupção e navegar pelos frames na pilha de um programa será útil quando seu programa não estiver fazendo o que você espera e quando você precisar analisar o que realmente está acontecendo.

Retorno

Muitas funções retornam um valor quando sua execução é concluída. Você sabe o tipo de dados que uma função retornará pelo tipo que precede o nome dela. (Se uma função não retornar nada, seu tipo de retorno é `void`.)

Crie uma nova C Command Line Tool chamada Degrees. No `main.c`, adicione uma função antes de `main()` que converta a temperatura de Celsius para Fahrenheit. Depois, atualize `main()` para chamar a nova função.

```
#include <stdio.h>

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

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit.\n", freezeInF);
    return 0;
}
```

Viu como você obtém o valor de retorno de `fahrenheitFromCelsius()` e atribui esse valor à variável `freezeInF` do tipo `float`? Compile e execute o programa.

A execução de uma função é interrompida quando há retorno. Por exemplo, observe esta função:

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

Se você chamasse essa função, a chamada de `printf()` nunca seria executada.

Uma pergunta natural seria: “Por que sempre retornamos 0 de `main()`?” Quando 0 retorna para o sistema, você está dizendo: “Está tudo certo.” Se finalizar o programa porque nada saiu errado, você retornará 1.

Isso pode ser contraditório no que se refere a como 0 e 1 funcionam nas instruções `if`; porque 1 é verdadeiro e 0 é falso; é natural pensar em 1 como um êxito e em 0 como uma falha. Então, pense em `main()` como retornando um relatório de erros. Nesse caso, 0 significa boas notícias! Sucesso é a ausência de erros.

Para deixar isso mais claro, alguns programadores usam as constantes `EXIT_SUCCESS` e `EXIT_FAILURE`, que são apenas um alias de 0 e 1, respectivamente. Essas constantes são definidas no arquivo de cabeçalho `stdlib.h`:

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

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

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit.\n", freezeInF);
    return EXIT_SUCCESS;
}
```

Neste livro, geralmente usaremos 0 em vez de `EXIT_SUCCESS`.

Variáveis globais e estáticas

Neste capítulo, falamos sobre as variáveis locais que existem apenas enquanto uma função está sendo executada. Também há as variáveis que podem ser acessadas a partir de qualquer função, a qualquer momento. Chamamos essas variáveis de *variáveis globais*. Para criar uma variável global, você a declara fora de

uma função específica. Por exemplo, você pode adicionar uma variável `lastTemperature` que mantenha a temperatura que foi convertida de Celsius. Adicione uma variável global para `Degrees`:

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

// Declare a global variable
float lastTemperature;

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

int main(int argc, const char * argv[])
{
    float freezeInC = 0;
    float freezeInF = fahrenheitFromCelsius(freezeInC);
    printf("Water freezes at %f degrees Fahrenheit.\n", freezeInF);
    printf("The last temperature converted was %f.\n", lastTemperature);
    return EXIT_SUCCESS;
}
```

Qualquer programa complexo envolverá dezenas de arquivos que contêm funções diferentes. As variáveis globais estão disponíveis para o código em todos esses arquivos. Muitas vezes, o que você deseja é compartilhar uma variável entre arquivos diferentes. No entanto, como você pode imaginar, ter uma variável que pode ser acessada por várias funções também pode gerar muita confusão. Para lidar com isso, temos as *variáveis estáticas*. Uma variável estática é como uma variável global em que ela é declarada fora de qualquer função. Entretanto, uma variável estática somente pode ser acessada a partir do código no arquivo em que ela foi declarada. Desse modo, você obtém o benefício não local do tipo “existe fora de qualquer função” e evita o problema “alguém mexeu em minha variável!”.

Você pode alterar sua variável global para que seja uma variável estática, mas, como você tem apenas um arquivo, `main.c`, isso não produzirá nenhum efeito.

```
// Declare a static variable
static float lastTemperature;
```

É possível atribuir um valor inicial às variáveis estáticas e às variáveis globais quando elas são criadas:

```
// Initialize lastTemperature to 50 degrees
static float lastTemperature = 50.0;
```

Se você não atribuir um valor inicial, automaticamente, elas serão inicializadas com zero.

Neste capítulo, você aprendeu sobre as funções. Quando você chegar na Parte III do Objective-C, ouvirá a palavra *método* – um método é muito similar a uma função.

Desafio

A soma dos ângulos internos de um triângulo dever ser igual a 180 graus. Crie uma nova C Command Line Tool chamada `Triangle`. No `main.c`, escreva uma função que obtenha os dois primeiros ângulos e retorne o terceiro. Esta será a aparência de quando você chamar essa função:

```
#include <stdio.h>

// Add your new function here

int main(int argc, const char * argv[])
{
    float angleA = 30.0;
    float angleB = 60.0;
    float angleC = remainingAngle(angleA, angleB);
    printf("The third angle is %.2f\n", angleC);
    return 0;
}
```

O resultado será:

The third angle is 90.00

6

Strings de formatação

Agora que você já sabe como as funções operam, vamos observar a função `printf` que você tem usado para escrever para o registro.

A função `printf` aceita uma *string* como um argumento e a exibe no registro. Uma string é uma “string” de caracteres enfileirados como contas em um colar. Normalmente, uma string é um texto.

Uma *string literal* é um texto dentro de aspas duplas. No projeto `AGoodStart` do Chapter 2, você chamou o `printf()` com argumentos de string literal:

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

Seu resultado se assemelhou ao seguinte:

```
It was the best of times.
It was the worst of times.
```

Você pode armazenar uma string literal em uma variável do tipo `char *`:

```
char *myString = "Here is a string";
```

Esta é uma *string em C*. Você pode ter criado strings em C no `AGoodStart` e as passado para `printf()`:

```
// Write the beginning of the novel
char *firstLine = "It was the best of times.\n";
char *secondLine = "It was the worst of times.\n";

// Print the beginning of the novel
printf(firstLine);
printf(secondLine);
```

Seu resultado teria a mesma aparência.

Utilização de tokens

A função `printf` pode fazer mais do que apenas exibir strings literais. Você também pode usar `printf()` para criar strings personalizadas no tempo de execução utilizando tokens e variáveis.

Reabra seu projeto ClassCertificates. No `main.c`, localize `congratulateStudent()`. Nesta função, você chama `printf()` e passa uma string com três tokens e três variáveis como argumentos.

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

Quando você passa uma string contendo um ou mais tokens para `printf()`, a string que você passa é chamada de *string de formatação*. Neste exemplo, a string de formatação inclui três tokens: `%s`, `%s` e `%d`.

Quando o programa é executado, os tokens são substituídos pelos valores dos argumentos das variáveis correspondentes. Nesse caso, essas variáveis são `student`, `course` e `numDays`. Seu resultado se assemelhou ao seguinte:

```
Liz has done as much iOS Programming as I could fit into 5 days.
```

Os tokens são substituídos em ordem no resultado: a primeira variável substitui o primeiro token, e assim por diante. Assim, se você alternar `student` e `course` na lista de variáveis, você verá:

```
iOS has done as much Liz Programming as I could fit into 5 days.
```

Por outro lado, nem todos os tokens e variáveis são alternáveis. O token que você escolhe informa ao `printf()` como o valor da variável precisa ser formatado. O token `%s` informa ao `printf()` para formatar o valor como uma string. O `%d` informa ao `printf()` para formatar o valor como um inteiro. (O `d` substitui o “decimal”.)

Se você usar o token errado, tal como usar `%d` quando a substituição é a string “`Ted`”, `printf()` você tentará representar “`Ted`” com um valor inteiro, que lhe dará resultados estranhos.

Existem outros tokens para outros tipos. Você aprenderá e usará muitos outros à medida que continuar a fazer os exercícios deste livro.

Sequências escape

O `\n` que você inseriu no final de suas strings é uma *sequência escape*. Uma sequência escape começa com `\`, que é o *caractere escape*. Esse caractere informa ao compilador que o caractere que vem imediatamente depois não tem seu significado comum.

O `\n` representa o caractere de nova linha. Nas instruções `printf()`, você inclui um caractere de nova linha quando deseja que o resultado continue em uma nova linha. Tente remover uma dessas novas linhas e veja o que acontece com seu resultado.

Outra sequência escape é `\\"`. Você a utiliza quando precisa incluir aspas em uma string literal. O caractere escape informa ao compilador para tratar o `"` como parte da string literal e não como uma instrução para finalizar a string. Veja aqui um exemplo:

```
printf("\\"It doesn't happen all at once,\\" said the Skin Horse.\n");
```

E aqui temos o resultado:

```
"It doesn't happen all at once," said the Skin Horse.
```

Desafio

Crie um novo projeto (C Command Line Tool) chamado `Square`. Escreva um programa que calcula e exibe o quadrado do inteiro. Coloque os números entre aspas. Seu resultado precisa se assemelhar ao seguinte:

```
"5" squared is "25".
```

7

Números

Você usou números para medir e exibir a temperatura, o peso e por quanto tempo deve-se cozinhar um peru. Agora, vamos examinar atentamente como os números funcionam na programação em C. Em um computador, os números podem ser de dois tipos: números inteiros e números de ponto flutuante. Você já usou os dois tipos.

Inteiros

Um inteiro é um número sem uma casa decimal – um número inteiro. Os inteiros são úteis para tarefas, tais como contagens. Algumas tarefas, tais como contar cada pessoa no planeta, realmente exigem números grandes. Outras tarefas, como contar o número de crianças em uma sala de aula, exigem números que não sejam tão grandes assim.

Para lidar com essas diferentes tarefas, as variáveis de inteiros vêm em diferentes tamanhos. Uma variável de inteiro contém um determinado número de bits, em que ela pode codificar um número; quanto mais bits a variável tiver, maior será o número que ela poderá conter. Os tamanhos típicos são 8 bits, 16 bits, 32 bits e 64 bits.

De modo similar, algumas tarefas exigem números negativos, outras não. Desse modo, os tipos de inteiros podem ser com sinal e sem sinal.

Um número sem sinal de 8 bits pode conter um inteiro de 0 a 255. Por quê? $2^8 = 256$ números possíveis. E optamos por começar com 0.

Um número com sinal de 64 bits pode conter qualquer inteiro de -9,223,372,036,854,775,808 a 9,223,372,036,854,775,807. Um bit para o sinal deixa $2^{63} = 9,223,372,036,854,775,808$. Há apenas um zero.

Ao declarar um inteiro, você pode ser muito específico:

```
UInt32 x; // An unsigned 32-bit integer  
SInt16 y; // A signed 16-bit integer
```

Entretanto, é mais comum que os programadores usem apenas os tipos descritivos que você aprendeu no Capítulo 3.

```
char a;      // 8 bits  
short b;    // Usually 16 bits (depending on the platform)  
int c;       // Usually 32 bits (depending on the platform)  
long d;     // 32 or 64 bits  (depending on the platform)  
long long e; // 64 bits
```

Por que o char é um inteiro de 8 bits? Quando C foi projetado, quase todo mundo utilizou um ASCII para representar os caracteres. ASCII deu um número a cada caractere frequentemente usado. Por exemplo, ‘B’ foi representado pelo número 66. Os números subiram até 127, de modo que pudemos adequar facilmente qualquer caractere ASCII em 8 bits. Para lidar com outros sistemas de caracteres (como Cyrillic ou Kanji), precisamos muito mais do que 8 bits. Por enquanto, fique com os caracteres ASCII. Falaremos sobre como lidar com outras codificações (como o Unicode) no Chapter 26.

E quanto ao sinal? char, short, int, long e long long são signed por padrão, mas você pode prefixá-los com o tipo unsigned para criar o equivalente sem sinal.

Além disso, os tamanhos dos inteiros dependem da plataforma. (Uma *plataforma* é uma combinação de um sistema operacional e um computador ou dispositivo móvel específico.) Algumas plataformas são de 32 bits, e outras são de 64 bits. A diferença está no tamanho do endereço de memória; falaremos mais sobre isso no Chapter 9.

Tokens para a exibição de inteiros

Crie um novo projeto: uma C Command Line Tool chamada Numbers. No `main.c`, crie um inteiro e exiba-o na base 10 (isto é, como um número decimal) usando `printf()`:

```
#include <stdio.h>

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

Você verá algo semelhante a:

```
x is 255.
```

Como você viu, `%d` exibe um inteiro como um número decimal. Quais outros tokens funcionam? Você pode exibir o inteiro na base 8 (octal) ou base 16 (hexadecimal). Adicione algumas linhas ao programa:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int x = 255;
    printf("x is %d.\n", x);
    printf("In octal, x is %o.\n", x);
    printf("In hexadecimal, x is %x.\n", x);

    return 0;
}
```

Ao executá-lo, você verá o seguinte:

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

(Voltaremos ao assunto de números hexadecimais no Chapter 38.)

E se o inteiro tiver muitos bits? Você coloca um `l` (de `long`) ou um `ll` (de `long long`) entre o `%` e o caractere de formatação. Altere seu programa para usar um `long` em vez de um `int`:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    long x = 255;
    printf("x is %ld.\n", x);
    printf("In octal, x is %lo.\n", x);
    printf("In hexadecimal, x is %lx.\n", x);

    return 0;
}
```

Se você estiver exibindo um número decimal sem sinal, você precisa usar `%u`:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    unsigned long x = 255;
    printf("x is %lu.\n", x);

    // Octal and hex already assume the number was unsigned
    printf("In octal, x is %lo.\n", x);
    printf("In hexadecimal, x is %lx.\n", x);

    return 0;
}
```

Operações com inteiros

Os operadores aritméticos +, - e * funcionam como você esperaria. Eles também têm as regras de precedência que você esperaria: * é verificado antes de + ou -. No `main.c`, substitua o código anterior por um cálculo:

```
#include <stdio.h>

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

Você verá:

`3 * 3 + 5 * 2 = 19`

Divisão de inteiros

A maioria dos programadores em C iniciantes é surpreendida pelo modo como a divisão de inteiros funciona. Faça o teste:

```
#include <stdio.h>

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

Você obterá `11 / 3 = 3,666667`, certo? Não. Você obterá `11 / 3 = 3`. Ao dividir um inteiro por outro, você sempre obterá um terceiro inteiro. O sistema arredonda para zero. (Desse modo, `-11 / 3` é `-3`.)

Isso realmente fará sentido se você pensar: “11 dividido por 3 é 3, com resto igual a 2”. E isso mostra que o resto é sempre muito valioso. O operador módulo (%) é como a /, mas retorna o resto em vez do quociente. Adicione o operador módulo para obter uma instrução que inclua o resto:

```
#include <stdio.h>

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

E se você *quisesse* obter `3,666667`? Você converteria `int` em `float` usando o *operador cast*. O operador cast é o tipo que você deseja, colocado entre parênteses à esquerda da variável a ser convertida. Converta seu denominador em um `float` antes de fazer a divisão:

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

Agora, será feita a divisão de ponto flutuante em vez da divisão de inteiro, e você obterá `3,666667`. Esta é a regra para a divisão de inteiro vs. ponto flutuante: / é a divisão de inteiro apenas se tanto o numerador quanto o

denominador forem do tipo inteiro. Se um deles for um número de ponto flutuante, será realizada a divisão de ponto flutuante.

NSInteger e NSUInteger

Nesse ponto, o Xcode suporta o desenvolvimento tanto de aplicativos de 32 bits quanto de 64 bits. Em uma tentativa de facilitar para que você escreva um código que funcionará com perfeição em qualquer sistema, a Apple apresentou o `NSInteger` e o `NSUInteger`. O primeiro abrange inteiros de 32 bits em sistemas de 32 bits; o segundo abrange inteiros de 64 bits em sistemas de 64 bits. `NSInteger` é signed (com sinal). `NSUInteger` é unsigned (sem sinal).

`NSInteger` e `NSUInteger` são usados extensamente nas bibliotecas da Apple, assim, quando você começar a trabalhar no Objective-C, os usará bastante.

A maneira recomendada de produzi-los com `printf()` é um pouco surpreendente. Já que a Apple não deseja fazer muitas suposições sobre o que está realmente por trás deles, é recomendado que você os converta para o `long` apropriado antes de tentar exibi-los:

```
NSInteger x = -5;
NSUInteger y = 6;
printf("Here they are: %ld, %lu", (long)x, (unsigned long)y);
```

Atalho de operadores

Todos os operadores que você viu até agora retornam um novo resultado. Então, por exemplo, para aumentar `x` em 1, você usaria o operador `+` e depois atribuiria o resultado de volta à `x`:

```
int x = 5;
x = x + 1; // x is now 6
```

Os programadores de C realizam estes tipos de operações de modo tão frequente que foram criados operadores que alteram o valor da variável sem uma atribuição. Por exemplo, você pode aumentar o valor contido no `x` em 1 com o *operador increment* (`++`):

```
int x = 5;
x++; // x is now 6
```

Há também um *operador decrement* (`--`) que diminui o valor em 1:

```
int x = 5;
x--; // x is now 4
```

E se você quiser aumentar `x` em 5 em vez de apenas 1? Você poderá usar adição e atribuição:

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

Mas há também um atalho para isso:

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

Você pode pensar na segunda linha como “atribuir a `x` o valor de `x + 5`.” Além do `+=`, existem também `-=`, `*=`, `/=` e `%=`.

Para obter o valor absoluto de um `int`, você usa uma função em vez de um operador. A função é `abs()`. Se você quiser o valor absoluto de um `long`, use `labs()`. Ambas as funções são declaradas no `stdlib.h`:

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

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

    return 0;
}
```

Números de ponto flutuante

Se você precisa de um número com uma casa decimal (como 3,2), é necessário usar um número de ponto flutuante. A maioria dos programadores pensa no número de ponto flutuante como uma mantissa multiplicada por 10 para um expoente inteiro. Por exemplo, 345,32 é considerado como $3,4532 \times 10^2$. E, basicamente, esta é a forma como ele é armazenado: um número de ponto flutuante de 32 bits tem 8 bits dedicados a conter o expoente (um inteiro com sinal) e 23 bits dedicados a conter a mantissa, com o 1 bit restante usado para conter o sinal.

Assim como os inteiros, os números de ponto flutuante têm vários tamanhos. Diferente dos inteiros, o sinal está *sempre* presente nos números de ponto flutuante:

```
float g;      // 32 bits
double h;     // 64 bits
long double i; // 128 bits
```

Tokens para exibição de números de ponto flutuante

`printf()` também pode exibir os números de ponto flutuante, mas comumente usando os tokens `%f` e `%e`. No `main.c`, substitua o código relacionado ao inteiro:

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

    return 0;
}
```

Ao compilar e executá-lo você verá:

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

Então, `%f` usa a notação decimal normal, e `%e` usa a notação científica.

Observe que `%f` no momento está mostrando 6 dígitos depois da vírgula decimal. Isso é um pouco de exagero. Limite para dois dígitos, modificando o token:

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

Ao executá-lo, você verá o seguinte:

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

A biblioteca matemática

Se você fará muitos cálculos matemáticos, você precisará da biblioteca matemática. Para ver o que há nessa biblioteca, abra o aplicativo Terminal em seu Mac e digite `man math`. Você obterá um excelente resumo de tudo o que há na biblioteca matemática: trigonometria, arredondamento, potenciação, raiz quadrada e cúbica etc.

Se você usar qualquer uma dessas funções matemáticas em seu código, não se esqueça de incluir o cabeçalho da biblioteca matemática no topo desse arquivo:

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

Um aviso: todas as funções relacionadas à trigonometria são resolvidas em radianos, e não em graus!

Desafio

Use a biblioteca matemática! Adicione código ao `main.c` que exibe o seno do radiano 1. Exiba o número arredondado com três casas decimais. O número deverá ser 0,841. A função seno é declarada da seguinte maneira:

```
double sin(double x);
```

Uma observação sobre os comentários

À medida que você realiza os exercícios, não se sinta constrangido de acrescentar comentários que o ajudem a recordar o que o código está fazendo. Adquira o hábito de comentar os códigos. Escreva comentários úteis e específicos que possam ser compreendidos por qualquer pessoa que esteja lendo o código ou por você mesmo no futuro ao reconsultar esse código.

Os comentários podem ser úteis quando você resolve desafios. Por exemplo, digamos que você encontre um desafio, mas não tem certeza de como resolvê-lo de maneira eficiente. Escreva uma observação sobre o que está lhe incomodando. À medida que for lendo o livro, reveja os desafios antigos e veja se já consegue aprimorar suas soluções. Isso também testará sua habilidade de escrever comentários úteis. Algo como:

```
// Not sure if this is right  
...
```

não será útil para o seu futuro.

8

Loops

No Xcode, crie um novo projeto: uma C Command Line Tool chamada Coolness.

O primeiro programa que escrevi exibia a frase: "Aaron is Cool" (Aaron é legal). (Na época, eu tinha 10 anos.) Escreva um programa para fazer isso agora:

```
#include <stdio.h>

int main(int argc, const char * argv[])
{
    printf("Aaron is Cool\n");
    return 0;
}
```

Compile e execute o programa.

Vamos supor por um instante que você pudesse fazer com que meu ego de 10 anos de idade se sentisse mais confiante se o programa exibisse a afirmação dezenas de vezes. Como faríamos isso?

Temos aqui uma forma não muito inteligente:

O modo inteligente de fazer isso é criar um loop.

O loop while

O primeiro loop que você vai usar é um loop `while`. A construção `while` funciona como a construção `if` discutida no Chapter 4. Você fornece a ele uma expressão e um bloco de código contidos entre chaves. Na construção `if`, se a expressão for verdadeira, o bloco de código será executado uma vez. Na construção `while`, o bloco é executado diversas vezes até que a expressão se torne falsa.

Reescreva a função `main()` de modo que se assemelhe ao seguinte:

```
#include <stdio.h>

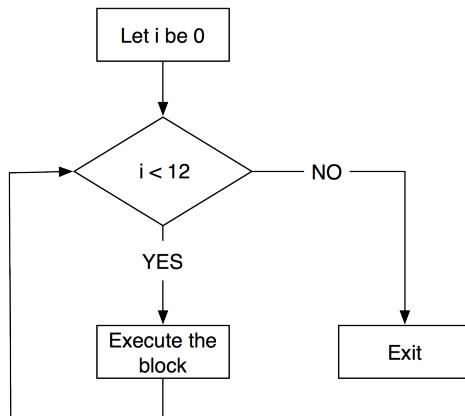
int main(int argc, const char * argv[])
{
    int i = 0;
    while (i < 12) {
        printf("%d. Aaron is Cool\n", i);
        i++;
    }
    return 0;
}
```

Compile e execute o programa.

A condicional (`i < 12`) está sendo verificada antes de cada execução do bloco. Na primeira vez em que ela for considerada falsa, a execução pulará para o código que está depois do bloco.

Observe que a segunda linha do bloco incrementa `i`. Isso é importante. Se `i` não fosse incrementada, esse loop, conforme está escrito, continuaria para sempre, pois a expressão sempre seria verdadeira.

Veja um fluxograma desse loop `while`:



O loop for

O loop while é uma estrutura de looping geral, mas os programadores de C usam muito o mesmo padrão básico:

```
some initialization
while (some check) {
    some code
    some last step
}
```

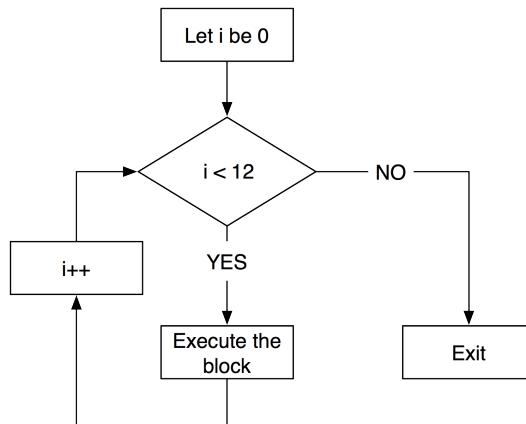
Desse modo, a linguagem C conta com um atalho: o loop for. No loop for, o padrão mostrado acima torna-se:

```
for (some initialization; some check; some last step) {
    some code;
}
```

Altere o programa para usar um loop for:

```
#include <stdio.h>

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



Observe que, nesse simples exemplo de loop, você usou o loop para indicar o número de vezes em que algo deve ocorrer. No entanto, os loops são mais comumente utilizados para *iterar* por uma coleção de itens, tais como uma lista de nomes. Por exemplo, eu poderia modificar esse programa para usar um loop em conjunto com uma lista de nomes de amigos. Por meio do loop, sempre um amigo diferente seria considerado legal. Você aprenderá mais sobre coleções e loops no Chapter 17.

break

Às vezes, é necessário interromper a execução do loop dentro do próprio loop. Por exemplo, digamos que você queira analisar os inteiros positivos em busca do número x , em que $x + 90 = x^2$. Seu plano é analisar os inteiros de 0 a 11 e interromper o loop quando a solução for encontrada. Altere o código:

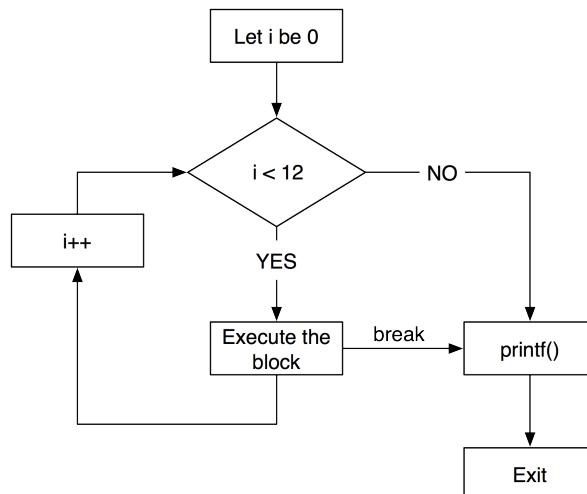
```
#include <stdio.h>

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

Compile e execute o programa. Você verá:

```
Checking i = 0
Checking i = 1
Checking i = 2
Checking i = 3
Checking i = 4
Checking i = 5
Checking i = 6
Checking i = 7
Checking i = 8
Checking i = 9
Checking i = 10
The answer is 10.
```

Observe que quando `break` é chamado, a execução pula diretamente para o fim do bloco de código.



continue

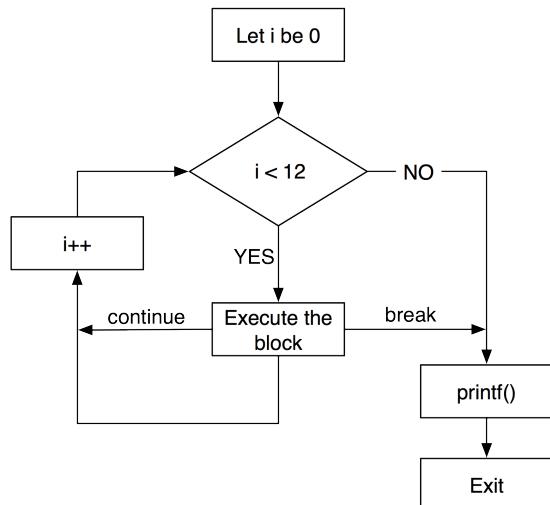
Às vezes, você se encontrará no meio de um bloco de código e precisará dizer: “Esqueça o restante desta execução do bloco de código e inicie a próxima execução do bloco de código.”. Isso é feito com o comando `continue`. Por exemplo, e se você tivesse plena certeza de que nenhum múltiplo de 3 atenderia à equação? Como seria possível evitar a perda de tempo precioso fazendo essa verificação?

```
#include <stdio.h>

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

Compile e execute:

```
Checking i = 1
Checking i = 2
Checking i = 4
Checking i = 5
Checking i = 7
Checking i = 8
Checking i = 10
The answer is 10.
```



O loop do-while

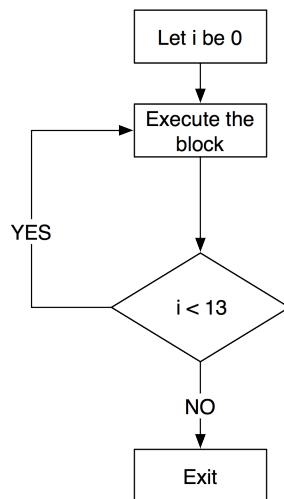
O loop do-while raramente é usado, mas, para que tudo seja mostrado a você, falaremos sobre ele. O loop do-while não verifica a expressão até que ele tenha executado o bloco. Portanto, ele garante que o bloco sempre seja executado pelo menos uma vez. Se você fosse reescrever o exercício original para usar um loop do-while, ele ficaria semelhante ao seguinte:

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

Observe o ponto e vírgula. Isso ocorre porque diferente dos outros loops, um loop do-while é, na verdade, uma instrução longa:

```
do { something } while ( something else stays true );
```

Veja um fluxograma desse loop do-while:



Desafio: contagem regressiva

Crie um novo projeto (C Command Line Tool) chamado CountDown e escreva um programa que faça a contagem decrescente de 99 a 0, de três em três, exibindo cada número.

Se o número for divisível por 5, também deverá ser exibida a frase: “Encontrei um!”. Portanto, o resultado será semelhante ao seguinte:

```
99
96
93
90
Found one!
87
...
0
Found one!
```

Desafio: entrada do usuário

Até agora, os programas que você escreveu realizam algum trabalho e, em seguida, produzem texto para o console. Neste desafio, você modificará sua solução CountDown para solicitar entrada do usuário.

Especificamente, você pedirá ao usuário para informar a partir de qual número a contagem regressiva deve começar.

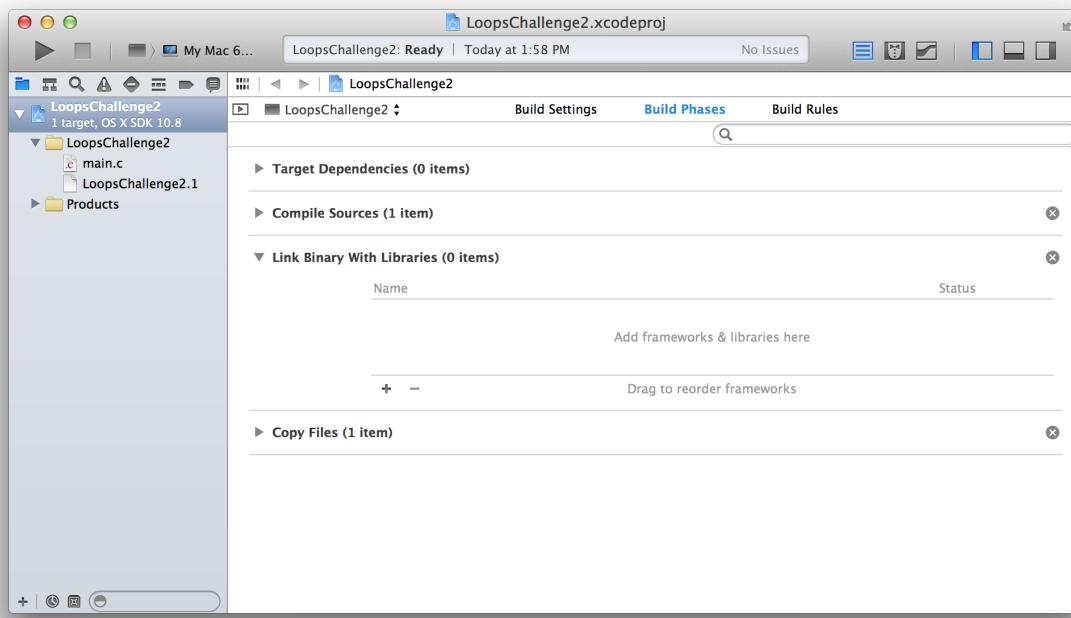
Para que isso aconteça, você precisa conhecer duas novas funções: **readline()** e **atoi()** (pronuncia-se “A a I”).

A função **readline()** é o oposto de **printf()**. Em vez de exibir texto na tela, ela obtém o texto que o usuário inseriu.

Antes de você poder usar **readline()**, você deve primeiramente adicionar a biblioteca que o contém em seu programa.

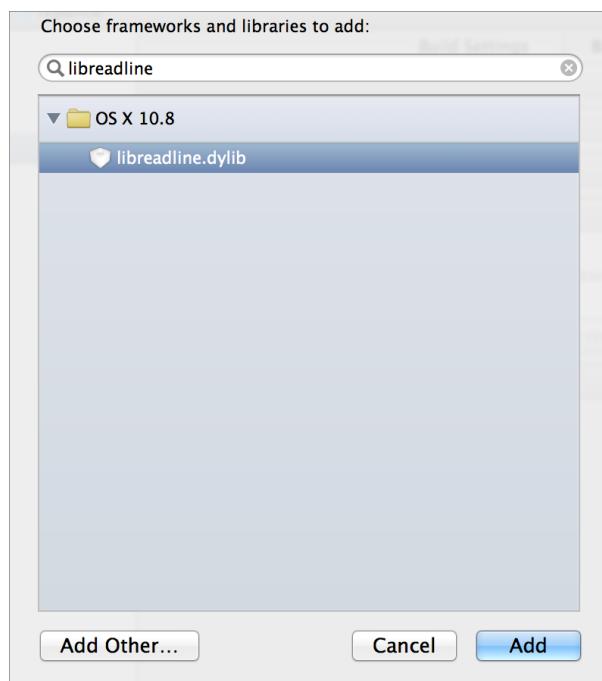
No navegador de projetos, clique no item Coolness de nível mais alto. Na área de edição, clique em Build Phases e, em seguida, divulgue o triângulo próximo a linha que diz: Link Binary With Libraries.

Figure 8.1 Vinculação de binários a bibliotecas



Clique no botão +. Uma página será exibida com uma lista de bibliotecas de códigos disponíveis. Use a caixa de pesquisa para pesquisar por `libreadline`. Quando ele for exibido na lista, selecione-o e clique em **Add**.

Figure 8.2 Bibliotecas



Selecione `main.c` no navegador de projetos para voltar a seu código.

Para que foram essas etapas? Às vezes, você deseja usar uma função que ainda não lhe foi fornecida. Então, você precisa informar ao Xcode qual *biblioteca de códigos* contém a função que deseja usar.

Vejamos um exemplo que usa a função **readline()**. Você iniciou este capítulo com um código que exibia a frase: Aaron is Cool (Aaron é legal). E se o usuário pudesse inserir o nome da pessoal que é legal? Veja como seria a aparência do programa quando executado. (A entrada do usuário é exibida em negrito)

```
Who is cool? Mikey
Mikey is cool!
```

O código ficaria assim:

```
#import <readline/readline.h>
#import <stdio.h>
int main(int argc, const char * argv[])
{
    printf("Who is cool? ");
    const char *name = readline(NULL);
    printf("%s is cool!\n\n",name);
    return 0;
}
```

(Digite este código em seu projeto Coolness e execute-o, caso queria vê-lo em ação.)

A primeira linha dessa função **main** é uma declaração de variável:

```
const char *name;
```

Lembre-se de que o **char *** é um tipo que você pode usar para strings.

Na terceira linha, você chama a função **readline** e passa **NULL** como seu argumento. Essa linha obtém o que o usuário digitou e armazena o conteúdo na variável **name**.

Agora, vamos nos focar na função **atoi**. Essa função utiliza uma string e a converte em um inteiro. (O ‘i’ substitui o inteiro, e o ‘a’ substitui o ASCII.)

Qual a utilidade de **atoi()**? O código de exemplo a seguir causaria um erro porque ele tenta armazenar uma string em uma variável de tipo **int**.

```
int num = "23";
```

Você pode usar **atoi()** para converter essa string em um inteiro com um valor de 23, que você felizmente pode armazenar em uma variável de tipo **int**:

```
int num = atoi("23");
```

(Se a string passada em **atoi()** não puder ser convertida em um inteiro, então **atoi()** retorna 0.)

Com essas duas funções em mente, modifique seu código para solicitar ao usuário a entrada e, em seguida, inicie a contagem regressiva a partir do ponto desejado. Seu resultado precisa se assemelhar ao seguinte:

```
Where should I start counting? 42
42
39
36
33
30
Found one!
27
...
```

Observe que o Xcode apresenta um comportamento interessante ao utilizar a função **readline**. Ele duplicará a entrada de texto como saída:

Figure 8.3 **readline()** saída



Esse é o comportamento esperado no Xcode.

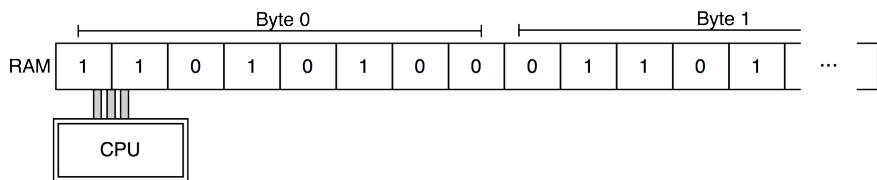
9

Endereços e ponteiros

Seu computador é, em seu cerne, um processador (a unidade central de processamento, ou CPU) e uma vasta área de switches (a memória de acesso aleatório ou RAM) que pode ser ligada ou desligada pelo processador. Dizemos que um switch contém um *bit* de informação. Com frequência, você verá 1 usado para representar “on” e 0 usado para representar “off”.

Oito desses switches compõem um *byte* de informação. O processador pode obter o estado desses switches, executar operações nos bits e armazenar o resultado em outro conjunto de switches. Por exemplo, o processador pode obter um byte daqui e outro dali, agrupá-los e armazenar o resultado em um byte em outro local.

Figure 9.1 Memória e CPU



A memória é numerada, e, geralmente, nos referimos ao *endereço* de um byte de dados específico. Quando alguém fala sobre uma CPU de 32 bits ou de 64 bits, normalmente está se referindo ao tamanho do endereço. Uma CPU de 64 bits pode lidar com imensamente mais memória do que uma CPU de 32 bits.

Obtenção de endereços

No Xcode, crie um novo projeto: uma C Command Line Tool chamada Addresses.

O endereço de uma variável é o local na memória em que o valor dessa variável é armazenado. Para obter o endereço da variável, você usa o operador &:

```
#include <stdio.h>

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

Observe o token %p. Este é o token que você pode substituir por um endereço de memória. Compile e execute o programa.

Seu resultado será semelhante ao seguinte:

```
i stores its value at 0xbffff738
```

embora seu computador possa colocar i em um endereço diferente. Os endereços de memória quase sempre são exibidos no formato hexadecimal.

Em um computador, tudo é armazenado na memória e, portanto, tudo tem um endereço. Por exemplo, uma função começa em algum endereço específico. Para obter esse endereço, basta usar o nome da função:

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

Compile e execute o programa.

Armazenamento de endereços em ponteiros

E se você quisesse armazenar um endereço em uma variável? Você poderia colocá-la em um inteiro sem sinal que tivesse o tamanho apropriado, mas o compilador poderá ajudá-lo a detectar seus erros se você for mais específico ao fornecer o tipo dessa variável. Por exemplo, se você quisesse uma variável chamada `ptr` que contivesse o endereço em que um `float` pudesse ser encontrado, você a declararia desta forma:

```
float *ptr;
```

Dizemos que `ptr` é uma variável que é um *ponteiro* para um `float`. Ela não armazena o valor de um `float`; ela pode conter um endereço no qual um `float` pode estar armazenado.

Declare uma nova variável chamada `addressOfI` que seja um ponteiro para um `int`. Atribua a ela o endereço de `i`.

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

Compile e execute o programa. Você não verá mudanças no comportamento.

Você está usando números inteiros no momento para simplificar. Mas, se quiser saber qual a finalidade dos ponteiros, eu direi. Passar o valor do inteiro atribuído a essa variável seria tão fácil quanto passar seu endereço. Entretanto, em breve, seus dados estarão bem maiores e muito mais complexos do que simples inteiros. É por isso que passamos endereços. Nem sempre é possível passar uma cópia dos dados com que você deseja trabalhar, mas você pode sempre passar o *endereço* de onde esses dados têm início. E é fácil acessar os dados uma vez que você tenha o endereço deles.

Obtenção de dados em um endereço

Se você tiver um endereço, pode obter os dados armazenados nele usando o operador `*`. Faça com que o log exiba o valor do inteiro armazenado em `addressOfI`.

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

Observe que o asterisco é usado de duas maneiras diferentes neste exemplo:

- Quando você declarou `addressOfI` para ser um `int *`. Ou seja, você disse ao compilador: “Ele conterá um endereço no qual um `int` possa ser armazenado.”
- Quando você ler, o valor de `int` está armazenado no endereço armazenado em `addressOfI`. (Os ponteiros também são chamados de referências. Portanto, o uso do ponteiro para ler os dados em um endereço é, muitas vezes, chamado de *desreferenciar* o ponteiro.)

Você também pode usar o operador * no lado esquerdo de uma atribuição para armazenar dados em um endereço específico:

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

Compile e execute o seu programa.

Não se preocupe se você ainda não tiver a questão dos ponteiros totalmente compreendida na sua mente. Neste livro, você passará muito tempo trabalhando com ponteiros, então, irá adquirir muita prática.

Quantos bytes?

Considerando que tudo fica na memória e que agora você sabe como localizar o endereço em que os dados têm início, a próxima pergunta é: “Quantos bytes esse tipo de dados consome?”

Usando **sizeof()**, você pode encontrar o tamanho de um tipo de dados. Por exemplo,

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

Aqui, há mais outro token novo nas chamadas para **printf()**: %zu. A função **sizeof()** retorna um valor do tipo **size_t**, para o qual %zu é o token de placeholder correto.

Compile e execute o programa. Caso seu ponteiro tenha 4 bytes de comprimento, seu programa executará no modo de 32 bits. Caso seu ponteiro tenha 8 bytes de comprimento, seu programa executará no modo de 64 bits.

sizeof() usará uma variável como argumento, então, você pode ter escrito o programa anterior desta forma:

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

NULL

Às vezes, você precisa de um ponteiro para nada. Ou seja, você tem uma variável que pode conter um endereço e deseja armazenar algo nela que torne explícito que a variável não está definida para nada. Usamos **NULL** para isso:

```
float *myPointer;
// Set myPointer to NULL for now, I'll store an address there
// later in the program
myPointer = NULL;
```

O que é `NULL`? Lembre-se de que um endereço é apenas um número. `NULL` é zero. Isso é muito útil em instruções `if`:

```
float *myPointer;  
...  
// Has myPointer been set?  
if (myPointer) {  
    // myPointer is not NULL  
    ...do something with the data at myPointer...  
} else {  
    // myPointer is NULL  
}
```

Às vezes, `NULL` indica que não há valor; por isso, você pode encontrar algo semelhante ao seguinte:

```
float *measuredGravityPtr = NULL;  
  
// Some code that might set measuredGravityPtr to be non-NULL  
...  
  
float actualGravity;  
  
// Did we measure the gravity?  
if (measuredGravityPtr) {  
    actualGravity = *measuredGravityPtr;  
} else {  
    actualGravity = estimatedGravity(planetRadius);  
}
```

Ou, pode usar o operador ternário para fazer a mesma coisa mais concisamente:

```
// If measuredGravityPtr is NULL, estimate the gravity  
float actualGravity =  
    measuredGravityPtr ? *measuredGravityPtr : estimatedGravity(planetRadius);
```

Depois, quando você estiver aprendendo sobre ponteiros para objetos, usará `nil` em vez de `NULL`. Eles são equivalentes, mas os programadores de Objective-C usam `nil` para indicar o endereço em que não há nenhum objeto.

Declarações de ponteiro avançadas

Ao declarar um ponteiro para um `float`, ele ficará assim:

```
float *powerPtr;
```

Como o tipo é um ponteiro para um `float`, talvez você se sinta inclinado a escrever desta forma:

```
float* powerPtr;
```

Sem problemas; o compilador permitirá que você faça isso. No entanto, os programadores avançados, não.

Por quê? Você pode declarar diversas variáveis em uma única linha. Por exemplo, se você quisesse declarar variáveis `x`, `y` e `z`, poderia fazer assim:

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

Cada uma delas é um `float`.

O que você acha que estas variáveis são?

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

Surpresa! `b` é um ponteiro para um `float`, mas `c` é apenas um `float`. Se você quiser que ambas sejam ponteiros, será necessário inserir um `*` na frente de cada uma:

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

Inserir o `*` diretamente ao lado do nome da variável torna isso mais claro.

Uma observação final: pode ser difícil entender os ponteiros no início. Não se preocupe se você ainda não tiver dominado essas ideias. Você trabalhará com elas durante o restante do livro, e elas farão mais sentido a cada vez que você usá-las.

Desafio: quanto de memória?

Escreva um programa que exiba quanto de memória um `float` consome.

Desafio: quanto de faixa?

Em um Mac, um `short` é um inteiro de 2 bytes, e um bit é usado para conter o sinal (positivo ou negativo). Qual é o menor número que um `short` pode armazenar? E o maior?

Um `short unsigned` (sem sinal) contém apenas números não negativos. Qual é o maior número que um `short unsigned` pode armazenar?

10

Passagem por referência

Há uma função em C padrão chamada `modf()`. Você fornece à `modf()` uma `double`, e ela calcula a parte inteira e a parte fracionária do número. Por exemplo, você fornece 3,14; a parte inteira será 3, e 0,14 será a parte fracionária.

Você, como o chamador de `modf()`, deseja as duas partes. No entanto, uma função em C pode retornar apenas um valor. Como `modf()` fornece a você as duas partes de informação?

Ao chamar `modf()`, você fornecerá um endereço em que ela poderá ocultar um dos números. Em particular, ela retornará a parte fracionária e copiará a parte inteira no endereço que você fornecer. Crie um novo projeto: uma C Command Line Tool chamada PBR.

Edite o `main.c`:

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

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

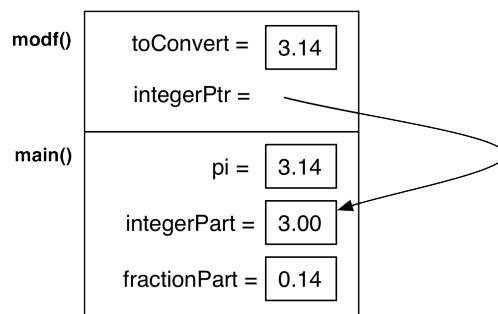
    // Pass the address of integerPart as an argument
    fractionPart = modf(pi, &integerPart);

    // Find the value stored in integerPart
    printf("integerPart = %.0f, fractionPart = %.2f\n", integerPart, fractionPart);

    return 0;
}
```

Este processo é conhecido como *passagem por referência*. Ou seja, você fornece um endereço (também conhecido como “referência”), e a função coloca os dados nesse local.

Figure 10.1 A pilha retornada por `modf()`



Esta é outra maneira de pensar na passagem por referência. Imagine que você dê ordens a espiões. Você pode dar a seguinte instrução: “Preciso de fotos do ministro da fazenda com sua namorada. Deixe um pequeno tubo de aço nos pés da estátua de anjo do parque. Quando você conseguir as fotos, enrole-as e coloque-as dentro do tubo. Irei pegá-las na terça-feira depois do almoço.” Em espionagem, esse método é conhecido como “dead drop”.

A `modf()` funciona como um “dead drop”. Você está solicitando que ela seja executada e informando a ela um local em que o resultado deve ser colocado, de modo que possa ter acesso a ele posteriormente. A única diferença é que, em vez de um tubo de aço, você está fornecendo a ela um local na memória em que o resultado poderá ser colocado.

Escrevendo funções de passagem por referência

O mundo é incrível. A variedade de culturas e pessoas ao redor do mundo inspira um resultado excelente das artes e ciências.

Uma complicação dessa diversidade é o fato de diferentes pessoas usarem diferentes unidades para medir o mundo ao redor delas. As comunidades científicas e de engenharia tendem a ter uma preferência por unidades métricas (como metros) em detrimento das unidades imperiais (como pés e polegadas), devido à sua facilidade de utilização em cálculos matemáticos.

Se você fosse construir um aplicativo para ser consumido por usuários em determinadas partes do mundo, porém, talvez fosse melhor poder imprimir os resultados dos seus cálculos baseados em metros usando pés e polegadas.

Como você escreveria uma função que converteria uma distância em metros para a distância equivalente em pés e polegadas? Ela precisaria ler um número de ponto flutuante e retornar dois outros. A declaração de uma função seria, então, semelhante à seguinte:

```
void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr);
```

Quando a função for chamada, um valor para `meters` será passado para ela. Também serão fornecidos a ela os locais nos quais os valores de `feet` e `inches` podem ser armazenados.

Agora, escreva a função próximo ao topo de seu arquivo `main.c` e chame-a a partir de `main()`:

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

void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr)
{
    // This function assumes meters is non-negative.

    // Convert the number of meters into a floating-point number of feet
    double rawFeet = meters * 3.281; // e.g. 2.4536

    // How many complete feet as an unsigned int?
    unsigned int feet = (unsigned int)floor(rawFeet);

    // Store the number of feet at the supplied address
    printf("Storing %u to the address %p\n", feet, ftPtr);
    *ftPtr = feet;

    // Calculate inches
    double fractionalFoot = rawFeet - feet;
    double inches = fractionalFoot * 12.0;

    // Store the number of inches at the supplied address
    printf("Storing %.2f to the address %p\n", inches, inPtr);
    *inPtr = inches;
}

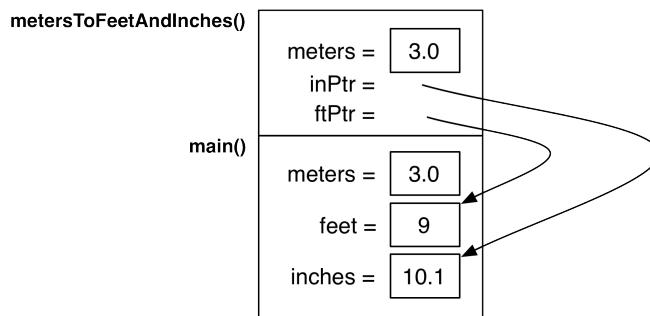
int main(int argc, const char * argv[])
{
    double meters = 3.0;
    unsigned int feet;
    double inches;

    metersToFeetAndInches(meters, &feet, &inches);
    printf("%.1f meters is equal to %d feet and %.1f inches.", meters, feet, inches);

    return 0;
}
```

Compile e execute o programa.

Figure 10.2 A pilha retornada por `metersToFeetAndInches()`



Evitando a desreferenciação de NULL

Às vezes, uma função pode fornecer muitos valores por referência, mas você pode se importar apenas com alguns deles. Como evitar a declaração dessas variáveis e a passagem de seus endereços quando você não pretender usá-las de forma alguma? Geralmente, você passa NULL como um endereço para informar à função: “Eu não preciso deste valor específico.”.

Isso significa que você sempre deve verificar se os ponteiros não são NULL antes de desreferenciá-los. Adicione essas verificações a `metersToFeetAndInches()`:

```

void metersToFeetAndInches(double meters, unsigned int *ftPtr, double *inPtr)
{
    double rawFeet = meters * 3.281;
    unsigned int feet = (unsigned int)floor(rawFeet);

    // Store the number of feet at the supplied address
    if (ftPtr) {
        printf("Storing %u to the address %p\n", feet, ftPtr);
        *ftPtr = feet;
    }

    double fractionalFoot = rawFeet - feet;
    double inches = fractionalFoot * 12.0;

    if (inPtr) {
        printf("Storing %.2f to the address %p\n", inches, inPtr);
        *inPtr = inches;
    }
}

```

Desafio

Em `metersToFeedAndInches()`, você usou `floor()` e subtração para desmontar `rawFeet` em suas partes inteiras e fracionárias. Altere `metersToFeedAndInches()` para, em vez disso, usar `modf()`.

11

Structs

Às vezes, você precisa que uma variável contenha vários blocos de dados relacionados. Em C, você pode fazer isso com uma *estrutura*, comumente chamada de *struct*. Cada bloco de dados é conhecido como um *membro* da struct.

Por exemplo, considere um programa que compute o Índice de Massa Corporal, ou IMC, de uma pessoa. IMC é o peso de uma pessoa em quilogramas dividido pelo quadrado da altura da pessoa em metros. (O IMC é uma ferramenta muito imprecisa para avaliar a condição física de uma pessoa, mas dá um ótimo exemplo de programação.)

Crie um novo projeto: uma C Command Line Tool chamada BMICalc. Edite `main.c` para declarar uma struct chamada Person que tenha dois membros: um float chamado `heightInMeters` e um int chamado `weightInKilos`. Em seguida, crie duas structs de Person:

```
#include <stdio.h>

// Here is the declaration of the struct
struct Person {
    float heightInMeters;
    int weightInKilos;
};

int main(int argc, const char * argv[])
{
    struct Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

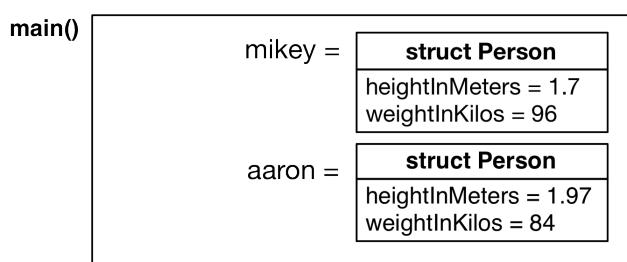
    struct Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);
    return 0;
}
```

Observe que você acessa os membros de uma struct usando um ponto final (programadores estilosos gostam de chamar apenas de “ponto”). Compile e execute o programa e confirme seu resultado.

Este é o frame de `main()` depois da atribuição de valores aos membros da struct.

Figure 11.1 Frame após designações de membro



Na maior parte do tempo, você usa uma declaração de struct por várias vezes. Sendo assim, é comum criar um `typedef` para o tipo de struct. Um `typedef` define um alias para uma declaração de tipo e permite usá-la de forma mais parecida com os tipos de dados comuns. Altere `main.c` para criar e usar um `typedef` para a struct `Person`. Observe que o código a ser substituído é mostrado riscado.

```
#include <stdio.h>
// Here is the declaration of the struct
struct Person {
    float heightInMeters;
    int weightInKilos;
};

// Here is the declaration of the type Person
typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;

int main(int argc, const char * argv[])
{
    struct Person mikey;
    Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

    struct Person aaron;
    Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);
    return 0;
}
```

Você pode passar um `Person` para outra função. Adicione uma função chamada `bodyMassIndex()` que aceite `Person` como um parâmetro e calcule o IMC. Depois, atualize `main()` para chamar essa função:

```
#include <stdio.h>

// Here is the declaration of the type Person
typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;

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

int main(int argc, const char * argv[])
{
    Person mikey;
    mikey.heightInMeters = 1.7;
    mikey.weightInKilos = 96;

    Person aaron;
    aaron.heightInMeters = 1.97;
    aaron.weightInKilos = 84;

    printf("mikey is %.2f meters tall\n", mikey.heightInMeters);
    printf("mikey weighs %d kilograms\n", mikey.weightInKilos);
    printf("aaron is %.2f meters tall\n", aaron.heightInMeters);
    printf("aaron weighs %d kilograms\n", aaron.weightInKilos);

    float bmi;
    bmi = bodyMassIndex(mikey);
    printf("mikey has a BMI of %.2f\n", bmi);

    bmi = bodyMassIndex(aaron);
    printf("aaron has a BMI of %.2f\n", bmi);

    return 0;
}
```

Aqui, você cria uma variável local `bmi` para conter o valor de retorno de `bodyMassIndex()`. Você obtém e imprime o IMC de Mikey. Depois, reutiliza a variável para obter e imprimir o IMC de Aaron.

Desafio

A primeira struct com que tive de lidar como programador foi `struct tm`, usada pela biblioteca padrão de C para conter a hora dividida em seus componentes. Esta é a definição da struct:

```
struct tm {
    int tm_sec;      /* seconds after the minute [0-60] */
    int tm_min;      /* minutes after the hour [0-59] */
    int tm_hour;     /* hours since midnight [0-23] */
    int tm_mday;     /* day of the month [1-31] */
    int tm_mon;      /* months since January [0-11] */
    int tm_year;     /* years since 1900 */
    int tm_wday;     /* days since Sunday [0-6] */
    int tm_yday;     /* days since January 1 [0-365] */
    int tm_isdst;    /* Daylight Savings Time flag */
    long tm_gmtoff; /* offset from CUT in seconds */
    char *tm_zone;   /* timezone abbreviation */
};
```

A função `time()` retorna o número de segundos desde o primeiro momento de 1970 em Greenwich, na Inglaterra. `localtime_r()` pode ler essa duração e agrupar uma struct `tm` com os valores apropriados. (Na verdade, ela utiliza o *endereço* do número de segundos desde 1970 e o *endereço* de uma struct `tm`.) Portanto, a obtenção do horário atual como uma struct `tm` seria semelhante ao seguinte:

```
long secondsSince1970 = time(NULL);
printf("It has been %ld seconds since 1970\n", secondsSince1970);

struct tm now;
localtime_r(&secondsSince1970, &now);
printf("The time is %d:%d:%d\n", now.tm_hour, now.tm_min, now.tm_sec);
```

O desafio é escrever um programa que informará a você a data (o formato 4-30-2015 está bom) daqui a 4 milhões de segundos.

(Uma dica: tm_mon = 0 indica janeiro, então, não se esqueça de adicionar 1. Além disso, inclua o cabeçalho <time.h> no início do seu programa.)

12

O heap

Até agora, seus programas usaram um tipo de memória – frames na pilha. Lembre-se de que toda função tem um frame onde suas variáveis locais são armazenadas. Essa memória é automaticamente alocada quando uma função é iniciada e é automaticamente desalocada quando a função é finalizada. Na verdade, variáveis locais são, às vezes, chamadas de *variáveis automáticas* devido a esse comportamento conveniente.

Às vezes, no entanto, você mesmo precisa solicitar um pedaço (chunk) contínuo de memória – um *buffer*. Geralmente, os programadores usam a palavra buffer para indicar uma longa linha de bytes de memória. O buffer vem de uma região da memória conhecida como *heap*, que é separada da pilha.

No heap, o buffer é independente de qualquer frame da função. Portanto, ele pode ser usado em várias funções. Por exemplo, você poderia solicitar um buffer de memória destinado a conter textos. Você poderia, então, chamar uma função que lerá um arquivo de texto no buffer, chamar uma segunda função que contaria todas as vogais no texto e, por fim, chamar uma terceira função para fazer a correção ortográfica. Quando tivesse terminado de usar o texto, você retornaria a memória que estava no buffer para o heap.

Você solicita um buffer de memória usando a função **malloc()**. Quando você já tiver usado o buffer, chamará a função **free()** para liberar sua solicitação dessa memória e retorná-la ao heap.

Digamos, por exemplo, que você precisou de um pedaço (chunk) de memória que fosse grande o suficiente para conter 1.000 floats. Observe o uso essencial de **sizeof()** para obter o número correto de bytes para o seu buffer.

```
#include <stdio.h>
#include <stdlib.h> // malloc() and free() are in stdlib.h

int main(int argc, const char * argv[])
{
    // Declare a pointer
    float *start0fBuffer;

    // Ask to use some bytes from the heap
    start0fBuffer = malloc(1000 * sizeof(float));

    // ...use the buffer here...

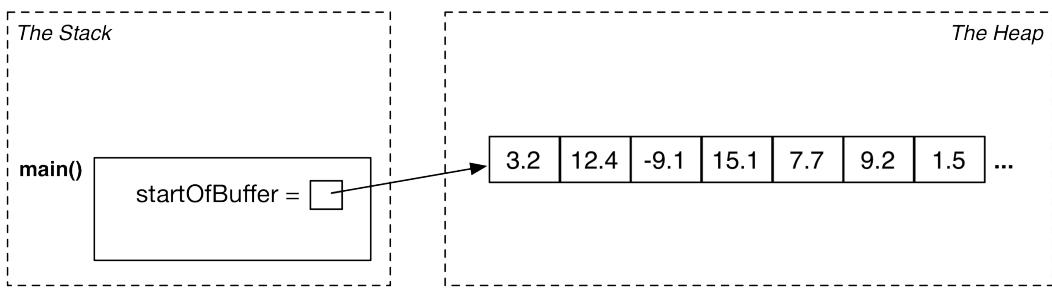
    // Relinquish your claim on the memory so it can be reused
    free(start0fBuffer);

    // Forget where that memory is
    start0fBuffer = NULL;

    return 0;
}
```

`start0fBuffer` é um ponteiro para o endereço do primeiro número de ponto flutuante no buffer.

Figure 12.1 Um ponteiro na pilha para um buffer no heap



Nesse ponto, a maioria dos livros sobre C levaria muito tempo falando sobre como usar `startOfBuffer` para ler e gravar dados em diferentes locais no buffer de números de ponteiros flutuantes. No entanto, este livro tenta mostrar os objetos a você o mais rápido possível. Portanto, ainda falaremos desses conceitos.

No Chapter 11, você criou uma struct como uma variável local no frame de `main()` na pilha. Você também pode alocar um buffer no heap para uma struct. Para criar uma struct `Person` no heap, você poderia escrever um programa semelhante ao seguinte:

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

typedef struct {
    float heightInMeters;
    int weightInKilos;
} Person;

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

int main(int argc, const char * argv[])
{
    // Allocate memory for one Person struct
    Person *mikey = (Person *)malloc(sizeof(Person));

    // Fill in two members of the struct
    mikey->weightInKilos = 96;
    mikey->heightInMeters = 1.7;

    // Print out the BMI of the original Person
    float mikeyBMI = bodyMassIndex(mikey);
    printf("mikey has a BMI of %f\n", mikeyBMI);

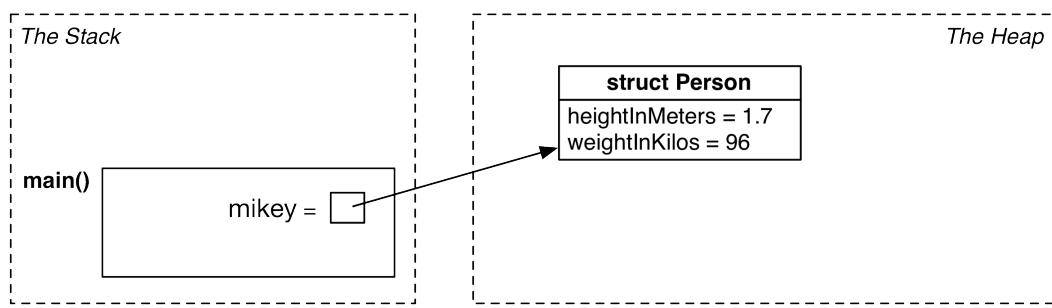
    // Let the memory be recycled
    free(mikey);

    // Forget where it was
    mikey = NULL;

    return 0;
}
```

Observe o operador `->`. O código `p->weightInKilos` diz: “Desreferencie o ponteiro `p` para a struct e obtenha-me o membro chamado `weightInKilos`.”

Figure 12.2 Um ponteiro na pilha para uma struct no heap



Essa ideia de structs no heap é muito eficiente. Ele forma a base dos objetos em Objective-C, sobre a qual falaremos a seguir.

Part III

Objective-C e Foundation

Agora que você já comprehende os princípios básicos de programas, funções, variáveis e tipos de dados, você está pronto para aprender sobre o Objective-C. Usaremos os programas de linha de comando, por enquanto, para manter o foco nos princípios básicos da programação.

Toda programação em Objective-C é feita através do framework Foundation. Um *framework* é uma biblioteca de classes que você utiliza para escrever programas. O que é uma classe? É sobre isso que falaremos primeiro...

13

Objetos

Neste capítulo, você escreverá seu primeiro programa em Objective-C. Esse programa será uma ferramenta de linha de comando como as que você já escreveu, mas será escrito em Objective-C.

No início da década de 1980, Brad Cox e Tom Love criaram a linguagem Objective-C. Quanto a objetos, eles se basearam na noção de structs alocadas no heap e adicionaram uma sintaxe de envio de mensagens.

Ao ir da programação em C para a programação em Objective-C, você estará entrando em um mundo de objetos e de programação orientada a objetos. Esteja preparado para encontrar novos conceitos e para ser paciente. Você usará esses padrões repetidamente, e eles se tornarão claros com o tempo e a prática.

Objetos

Um *objeto* é similar a uma struct (como a `struct Person` que você criou no Chapter 11). Assim como uma struct, um objeto pode conter diversas partes de dados relacionados. Em uma struct, as chamamos de *membros*. Em um objeto, as chamamos de *variáveis de instância* (ou você pode ouvir “*ivars*”).

A diferença de um objeto para uma struct é que um objeto também pode ter suas próprias funções que atuam nos dados que contém. Essas funções são chamadas de *métodos*.

Classes

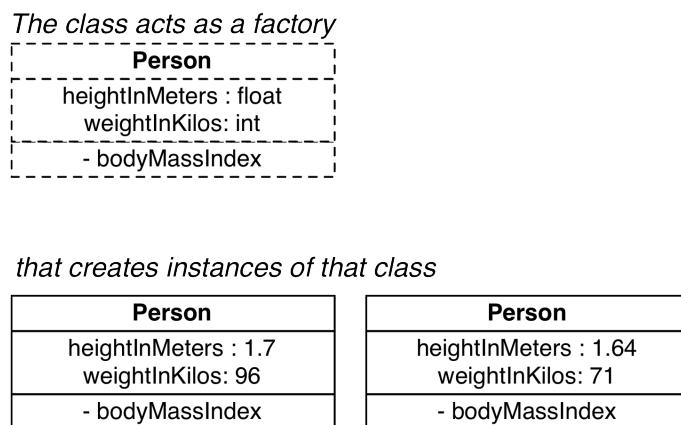
Uma *classe* descreve um determinado tipo de objeto listando as variáveis de instância e os métodos que o objeto terá. Uma classe pode descrever um objeto que representa:

- um conceito, como uma data, uma string, ou
- uma determinada coisa do mundo real, como uma pessoa, um local, ou uma conta corrente

Uma classe define um tipo de objeto. Além disso, produz objetos de tal tipo. Você pode pensar em uma classe como sendo ao mesmo tempo o plano de projeto e a fábrica.

No Chapter 18, você reescreverá o programa de cálculo de IMC usando objetos em vez de structs. Você vai criar uma classe chamada **Person**. Os objetos produzidos por ela serão *instâncias* da classe **Person**. Essas instâncias terão variáveis de instância para altura e peso, além de um método para calcular o IMC.

Figure 13.1 Uma classe **Person** e duas instâncias **Person**



Uma observação sobre nossos diagramas de objeto: Classes, como a classe **Person**, são diagramadas com uma borda tracejada. Instâncias são desenhadas com bordas sólidas. Essa é uma convenção de diagramação comum para distinguir entre classes e instâncias de uma classe.

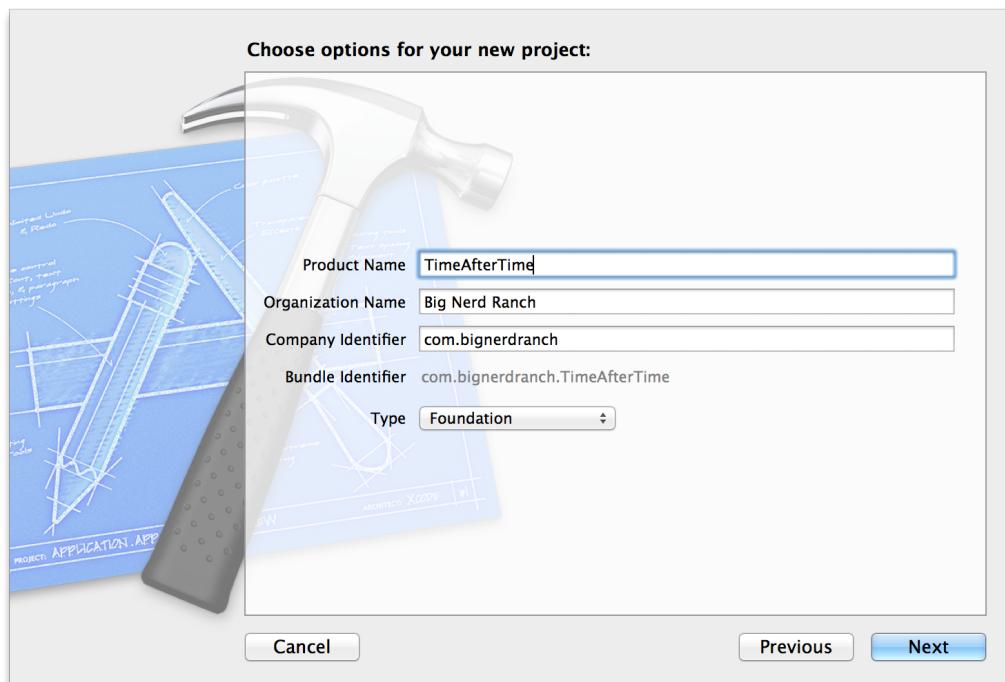
Nesse ponto do capítulo, vamos passar da teoria para a prática. Não se preocupe se objetos, classes, instâncias e métodos ainda não fazem tanto sentido. A prática ajudará.

Em vez de iniciar escrevendo novas classes personalizadas, você vai criar instâncias de uma classe fornecida pela Apple. Essa classe se chama **NSDate**. Uma instância de **NSDate** representa um ponto no tempo. Você pode pensar nela como um timestamp (registro de tempo). Você também usará métodos da classe **NSDate**.

Criação de seu primeiro objeto

Crie um novo projeto Command Line Tool chamado **TimeAfterTime**. Torne seu tipo Foundation – não C como em seus projetos anteriores (Figure 13.2).

Figure 13.2 Criação de uma ferramenta de linha de comando Foundation



Os arquivos que contêm código em Objective-C recebem o sufixo **.m**. Encontre e abra **main.m**.

No topo desse arquivo, encontre a linha que diz

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

Quando Xcode criou seu projeto, ele importou o framework Foundation para você. Um *framework* é um conjunto de classes, funções, constantes e tipos relacionados. O framework Foundation contém classes fundamentais usadas em todos os aplicativos iOS e OS X. A classe **NSDate** encontra-se no framework Foundation.

Qual a diferença entre `#import` e `#include`? `#import` é mais rápido e mais eficiente. Quando o compilador vê a diretriz `#include`, ele simplesmente copia e cola o conteúdo do arquivo a ser incluído. Quando o compilador vê a diretriz `#import`, ele primeiro verifica se outro arquivo já importou ou incluiu o arquivo.

No **main.m**, adicione a seguinte linha de código:

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

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

        NSDate *now = [NSDate date];

    }
    return 0;
}
```

Do lado esquerdo do operador de atribuição (=), há uma variável chamada `now`. O * indica que essa variável é um ponteiro. Esse ponteiro mantém o endereço na memória onde a instância de **NSDate** reside.

O código do lado direito retorna o endereço de uma instância de **NSDate**. Esse código é conhecido como um *envio de mensagem*, e você aprenderá mais sobre mensagens na próxima sessão. Primeiro, adicione a seguinte linha que escreve o endereço da instância **NSDate** usando a função **NSLog()**.

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
    }
    return 0;
}
```

NSLog() é uma função do framework Foundation bastante parecida com **printf()**. Ela aceita uma string de formatação e pode ter tokens substituíveis.

Compile e execute o programa. Você verá algo semelhante a:

```
2013-08-05 11:53:54.366 TimeAfterTime[4862:707] This NSDate object lives at 0x100116240
```

Diferente de **printf()**, **NSLog()** introduz seu resultado com a data, a hora, o nome do programa e a ID do processo. A partir de agora, quando mostrarmos o resultado de **NSLog()**, pularemos esses dados – a página é muito pequena.

```
This NSDate object lives at 0x100116240
```

Você criou uma instância de **NSDate** e ela reside no endereço armazenado em `now`. Para compreender como isso aconteceu, você precisa de conhecimento sobre métodos e mensagens.

Métodos e mensagens

Métodos são como funções. Eles contém código a ser executado por comando. Em Objective-C, para executar o código em um método, envia-se uma *mensagem* para o objeto ou a classe que possui tal método.

A classe **NSDate** possui um método **date**. No código que você acaba de escrever, você enviou a mensagem **date** para a classe **NSDate** para executar o método **date**.

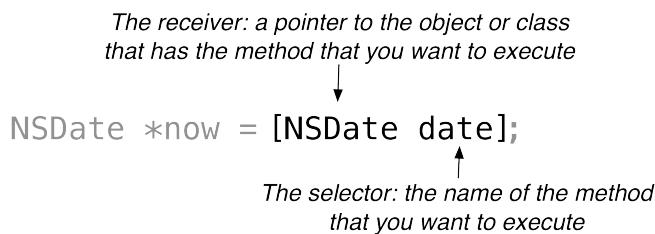
```
NSDate *now = [NSDate date];
```

Esse foi seu primeiro envio de mensagem.

Envios de mensagem

Um envio de mensagem fica entre colchetes e contém duas partes: o receptor e o seletor.

Figure 13.3 Um envio de mensagem



O que enviar a mensagem **date** faz? Quando o método **date** é executado, a classe **NSDate** reivindica parte da memória no heap para uma instância de **NSDate**, inicializa a instância para a data/hora atual e retorna o endereço do novo objeto.

Adicione outra chamada **NSLog()** a seu programa.

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"%@", now);

    }
    return 0;
}
```

Use aqui um novo token, **%@**. Esse token de Objective-C pede ao objeto uma descrição de si mesmo. (Você aprenderá mais sobre **%@** no Chapter 20.)

Compile e execute o programa. Você verá algo semelhante ao seguinte:

```
This NSDate object lives at 0x100116240
The date is 2013-08-05 16:09:14 +0000
```

Outra mensagem

Agora que você tem uma instância de **NSDate**, você pode enviar mensagens para esse novo objeto. Você enviará para ele a mensagem **timeIntervalSince1970**.

Ao enviar essa mensagem a uma instância de **NSDate**, você receberá de volta a diferença em segundos entre a data/hora que a instância **NSDate** representa e 00h00 de 01/01/1970 em Greenwich, Inglaterra. (Por que 1970? OS X e iOS baseiam-se em Unix, e 1970 é o início da “época Unix”.)

Envie a mensagem **timeIntervalSince1970** para a instância **NSDate** indicada por **now**. O método **timeIntervalSince1970** retorna uma **double**. (Lembre-se que uma **double** é um número de ponto flutuante que possui maior precisão que um **float**.) Coloque o resultado em uma variável chamada **seconds**.

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"%@", now);

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

    }
    return 0;
}
```

Compile e execute o programa para ver os resultados.

Métodos de classe vs. métodos de instância

Considere duas mensagens que você enviou:

Figure 13.4 Dois envios de mensagem

```
NSDate *now = [NSDate date]
               ↑       ↑
             receiver   selector

double seconds = [now timeIntervalSince1970]
               ↑       ↑
             receiver   selector
```

Você enviou a mensagem **date** para a classe **NSDate**. **date** é um *método de classe*. Normalmente, os métodos de classe criam uma instância da classe e inicializam suas variáveis de instância.

No segundo envio de mensagem, você enviou a mensagem **timeIntervalSince1970** para a instância **NSDate** indicada por **now**. **timeIntervalSince1970** é um *método de instância*. Normalmente, os métodos de instância fornecem informações sobre as variáveis de instância de uma instância ou executam uma operação nelas.

Métodos de instância tendem a ser mais comuns nos programas em Objective-C. Você envia uma mensagem para uma classe para criar uma instância. Essa mensagem causa a execução de um método de classe. Mas, logo que você tiver tal instância, a instância tenderá a receber muitas mensagens durante a execução do programa. Essas mensagens causarão a execução de métodos de instância.

Envio de mensagens ruins

O que aconteceria se você enviasse o método de classe **date** para uma instância **NSDate** ou o método de instância **timeIntervalSince1970** para a classe **NSDate**? Faça o teste:

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

        // Sending bogus messages to see errors...
        double testSeconds = [NSDate timeIntervalSince1970];
        NSDate *testNow = [now date];

    }
    return 0;
}
```

Compile seu programa (Command-B) e o Xcode relatará erros de compilação. Na primeira linha nova, encontre um erro que diz: No known class method for selector 'timeIntervalSince1970'.

(Há um outro erro nessa linha sobre a inicialização de uma **double**. Ignore-o por enquanto.)

O erro é claro: o receptor nesse envio de mensagem é uma classe **NSDate**, então o seletor deveria ser o nome de um método de classe **NSDate**. Esse seletor não é.

No seu próximo envio de mensagem com erro, encontre um erro que diz: No visible @interface for 'NSDate' declares the selector 'date'.

Esse erro não é tão claro: ele está dizendo que **NSDate** não possui nenhum método de instância cujo nome corresponda ao seletor **date**.

É importante que todo iniciante reconheça esses erros. Eles aparecem quando você digita incorretamente o nome da mensagem. Faça o teste:

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

        // Sending bogus messages to see errors...
        double testSeconds = [NSDate timeIntervalSince1970];
        NSDate *testNow = [now date];

        // Mistyped selector name
        testSeconds = [now fooIntervalSince1970];

    }
    return 0;
}
```

Compile seu programa e será dito a você que **NSDate** não possui um método de instância chamado **fooIntervalSince1970**.

Maiúsculas e minúsculas são consideradas!

O código em Objective-C diferencia maiúsculas e minúsculas. Portanto, `timeIntervalSince1970` e `timeintervalsince1970` são duas mensagens distintas. Apenas uma dessas mensagens equivale ao nome de um método `NSDate`. Faça o teste:

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

        // Sending bogus messages to see errors...
        NSDate *testNow = [now date];
        double testSeconds = [NSDate timeIntervalSince1970];

        // Mistyped selector name
        testSeconds = [now fooIntervalSince1970];

        // Typo! Lowercase 'i' and 's'
        testSeconds = [now timeintervalsince1970];

    }
    return 0;
}
```

Lembre-se da diferenciação entre maiúsculas e minúsculas nos nomes de métodos. Essa é uma fonte de muitos erros para os iniciantes. Remova seus envios de mensagem falsos antes de prosseguir:

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

        // Sending bogus messages to see errors...
        // NSDate *testNow = [now date];
        // double testSeconds = [NSDate timeIntervalSince1970];

        // Mistyped selector name
        // testSeconds = [now fooIntervalSince1970];

        // Typo! Lowercase 'i' and 's'
        // testSeconds = [now timeintervalsince1970];

    }
    return 0;
}
```

Convenções de nomes em Objective-C

- Nomes de variáveis que indicam instâncias são escritos em “camel case (minúsculas concatenadas)”. Eles começam com uma letra em minúsculo, com a primeira letra de cada palavra subsequente em maiúsculo. `now, weightLifter, myCurrentLocation`
- Nomes de métodos também usam camel case: `date, bodyMassIndex, timeIntervalSince1970`

- Nomes de classes utilizam maiúsculas, isto é, iniciam com letras maiúsculas mas depois usam o camel case: **NSDate**, **Person**, **CLLocation**, **NSMutableArray**

Normalmente, nomes de classe são iniciados com prefixos para evitar confusão entre classes de nomes parecidos. Os prefixos também informam a que framework algo pertence. O prefixo **NS** é usado para o framework Foundation: **NSDate**, **NSLog()**. **NS** é uma abreviação de NeXTSTEP, a plataforma para a qual o Foundation foi originalmente concebido.

- Muitos tipos e constantes criados pela Apple também utilizam maiúsculas. Por exemplo, **NSInteger** não é uma classe; trata-se simplesmente de um tipo de inteiro. **NSOKButton** é uma constante igual a 1.

Uma observação sobre terminologia

Ao falar sobre código, os desenvolvedores normalmente dizem “uma **NSDate**” para se referir a uma instância de **NSDate**. Também é comum se referir a uma instância pelo que ela representa. Você pode se referir a uma instância de **NSDate** como “um objeto de data” ou até mesmo apenas como “uma data”.

Para se referir a uma classe, os desenvolvedores costumam empregar apenas o nome da classe. Por exemplo, “**NSDate**” foi incluída no OS X 10.0”.

No início, pode ser difícil para você entender as noções de classes, objetos, mensagens e métodos. Se você ainda tiver dúvidas sobre objetos, não se preocupe. É apenas o começo. Você usará esses conceitos muitas e muitas vezes, e eles farão mais sentido a cada vez que você usá-los.

Desafio

Neste desafio, você escreverá uma Foundation Command Line Tool que imprima o nome de seu computador. Esse programa usará duas classes do framework Foundation: **NSHost** e **NSString**.

Primeiro, você chamará uma instância de **NSHost** contendo as informações de seu computador. Em seguida, você perguntará o nome de seu computador ao objeto **NSHost**. Por fim, você usará **NSLog()** para imprimir esse nome.

Você precisará das seguintes informações:

- Para chamar uma instância de **NSHost**, envie a mensagem **currentHost** para a classe **NSHost**.
- Quando você tiver uma instância de **NSHost**, envie a mensagem **localizedNome** para ela. O método **localizedNome** retorna um ponteiro para uma instância de **NSString**. Assim, você pode armazenar o resultado do envio dessa mensagem em uma variável de tipo **NSString ***.
- Use **NSLog()** e o token **%@** para imprimir o nome de seu computador.

Esse desafio é muito parecido com o que você fez neste capítulo: chamar um novo objeto, enviar uma mensagem para ele e enviar o resultado da mensagem em uma variável. Não deixe as novas classes e métodos te derrubarem. Além disso, a execução do programa pode levar um tempo surpreendentemente longo.

14

Mais mensagens

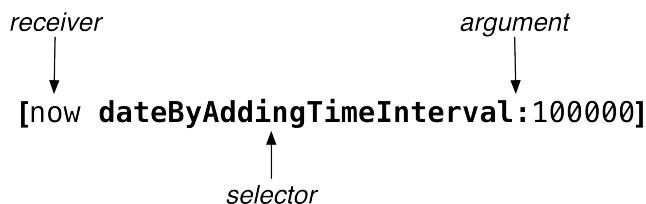
Os objetos são muito comunicativos por natureza. Eles enviam e recebem muitas mensagens sobre o trabalho que estão realizando. Neste capítulo, você aprenderá sobre mensagens com argumentos, envios de mensagens aninhadas, entre outros.

Uma mensagem com um argumento

O programa TimeAfterTime possui uma **NSDate** inicializada para a data e a hora em que é criada. E se você quiser representar uma data no futuro – por exemplo, 100.000 segundos depois da primeira data? Você pode criar uma data assim enviando a mensagem **dateByAddingTimeInterval:** para a instância original de **NSDate**.

Observe os dois pontos no final do nome do método **dateByAddingTimeInterval::**. Isso indica que **dateByAddingTimeInterval::** aceita um argumento. Os métodos, assim como as funções, podem ter zero, um, ou mais argumentos.

Figure 14.1 Envio de mensagem com um argumento



O método **dateByAddingTimeInterval::** aceita o número de segundos pelo qual a nova **NSDate** deve divergir da original. (Um número negativo daria a você uma **NSDate** no passado.)

No TimeAfterTime, use **dateByAddingTimeInterval::** para criar uma segunda data que seja 100.000 segundos (pouco mais de um dia) depois da data apontada por now:

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

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

    }
    return 0;
}
```

Quando um método tem um argumento, o dois pontos é uma parte essencial do nome do método. Não existe nenhum método chamado **dateByAddingTimeInterval**. Existe apenas **dateByAddingTimeInterval::**.

Vários argumentos

E se você quiser saber o dia do mês (p. ex. 1º de junho) de um objeto **NSDate**? Uma **NSDate** não sabe essa informação. Em vez disso, você deve perguntar a uma instância de **NSCalendar**.

NSCalendar é outra classe Foundation. Você pode criar uma instância de **NSCalendar** enviando para a classe **NSCalendar** a mensagem **currentCalendar**.

O método de classe **currentCalendar** retornará o endereço de uma instância de **NSCalendar** que corresponda às configurações do usuário. (Na maioria dos países ocidentais, o calendário gregoriano é o padrão, mas existem vários outros calendários, como o calendário hebreu e o calendário islâmico.) Peça à classe **NSCalendar** uma instância de **NSCalendar**.

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

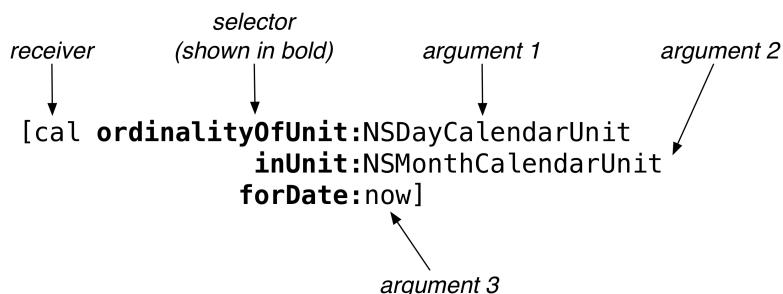
        NSDate *later = [now dateByAddingTimeInterval:100000];
        NSLog(@"In 100,000 seconds it will be %@", later);

        NSCalendar *cal = [NSCalendar currentCalendar];
        NSLog(@"My calendar is %@", [cal calendarIdentifier]);
    }
    return 0;
}
```

A **NSCalendar** tem um método **ordinalityOfUnit:inUnit:forDate:** que é capaz de fornecer mais informações sobre uma **NSDate**. Esse método usa três argumentos. Você pode perceber pelo número de dois pontos no nome do método.

Vamos começar com o terceiro argumento. Trata-se do objeto **NSDate** sobre o qual você deseja mais informações. O primeiro e o segundo argumentos são constantes da classe **NSCalendar** que descrevem o tipo de informação que você deseja. Para obter o dia do mês, você passa **NSDayCalendarUnit** para o primeiro argumento e **NSMonthCalendarUnit** para o segundo argumento.

Figure 14.2 Um envio de mensagem com três argumentos



Esse método leva três argumentos, por isso seu nome possui três partes, mas trata-se de *um* envio de mensagem que dispara *um* método.

Existem constantes de **NSCalendar** que você pode usar para encontrar informações sobre horas, dias, semanas, meses, trimestres etc. Por exemplo, para descobrir em que semana do mês uma **NSDate** cai, você enviará a mesma mensagem e passaria **NSWeekCalendarUnit** e **NSMonthCalendarUnit** como o primeiro e o segundo argumentos.

No TimeAfterTime, peça à instância de **NSCalendar** que encontre o dia do mês para a **NSDate** apontada por **now**.

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

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

        NSDate *now = [NSDate date];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

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

        NSCalendar *cal = [NSCalendar currentCalendar];
        NSLog(@"My calendar is %@", [cal calendarIdentifier]);
        unsigned long day = [cal ordinalityOfUnit:NSDayCalendarUnit
                                         inUnit:NSSMonthCalendarUnit
                                         forDate:now];
        NSLog(@"This is day %lu of the month", day);

    }
    return 0;
}
```

Observe que você dividiu o envio da mensagem **ordinalityOfUnit:inUnit:forDate:** em três linhas. Geralmente, os programadores de Objective-C alinharam os dois pontos para que fique fácil informar as partes do nome do método a partir dos argumentos. (O Xcode fará isso por você: sempre que você iniciar uma nova linha, será dado o recuo apropriado na linha anterior. Se isso não estiver acontecendo, verifique suas preferências no Xcode quanto aos recuos.)

Aninhamento de envios de mensagens

Envios de mensagens podem ser *aninhados*. Por exemplo, para descobrir o número de segundos desde o início de 1970, você pode escrever seu código desta maneira:

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

Ou você pode aninhar os dois envios de mensagem da seguinte forma:

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

Quando envios de mensagem são aninhados, o sistema primeiro trata do envio de mensagem de dentro e depois da mensagem que o contém. Desse modo, o método **date** é enviado à classe **NSDate**, e o resultado disso (um ponteiro para a instância recém-criada) é então enviado a **timeIntervalSince1970**.

Frequentemente, você verá envios de mensagem aninhados no código, e você precisa saber lê-los. Entretanto, ao escrever seu próprio código, você pode perceber que aninhar mensagens é contraproducente. O aninhamento deixa seu código mais difícil de ler e de depurar, porque mais de uma coisa está acontecendo em uma linha.

alloc e init

Há um caso em que sempre é correto e apropriado aninhar dois envios de mensagem. Você sempre aninha as mensagens **alloc** e **init**.

O método **alloc** é um método de classe que toda classe possui. Ele retorna um ponteiro para uma nova instância que precisa ser inicializada. Uma instância não inicializada pode existir na memória, mas não está pronta para receber mensagens. O método **init** é um método de instância que toda classe possui. Ele *inicializa* uma instância para que fique pronta para funcionar.

Pratique o uso de mensagens aninhadas em seu programa. Crie um objeto **NSDate** enviando mensagens **alloc** e **init** em vez da mensagem **date**.

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

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

        NSDate *now = [NSDate date];
        NSDate *now = [[NSDate alloc] init];
        NSLog(@"This NSDate object lives at %p", now);
        NSLog(@"The date is %@", now);

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

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

        NSCalendar *cal = [NSCalendar currentCalendar];
        NSLog(@"My calendar is %@", [cal calendarIdentifier]);

        unsigned long day = [cal ordinalityOfUnit:NSDayCalendarUnit
                                         inUnit:NSSMonthCalendarUnit
                                         forDate:now];
        NSLog(@"This is day %lu of the month", day);

    }
    return 0;
}
```

Não existe diferença entre as duas maneiras de se criar uma instância de **NSDate**. O método **init** de **NSDate** inicializa o objeto **NSDate** para a data e hora atuais – assim como o método **date** faz. O método **date** é uma maneira conveniente de obter uma instância **NSDate** com o mínimo de código. Inclusive, chamamos esse tipo de método de *método de conveniência*.

Envio de mensagens para nil

Quase todas as linguagens orientadas a objetos possuem a ideia de **nil**, o ponteiro para nenhum objeto. Em Objective-C, usamos **nil** em vez de **NULL**, o que foi discutido no Chapter 9. Na verdade, ambos são a mesma coisa: o ponteiro nulo. Por convenção, entretanto, usamos **nil** ao nos referirmos ao valor de um ponteiro vazio declarado como apontando para um tipo de objeto Objective-C, e **NULL** ao nos referirmos a qualquer outro ponteiro, tal como para uma struct.

Na maioria das linguagens orientadas a objetos, o envio de uma mensagem para **nil** não é permitido. Como resultado, você deve verificar se não há mensagens sendo enviadas para **nil** antes de acessar um objeto. Assim, você verá muito o seguinte:

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

Quando o Objective-C foi projetado, decidiu-se que não haveria problemas em enviar uma mensagem para **nil**; isso simplesmente não faria nada. Portanto, este código é totalmente válido:

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

Item importante 1: se você estiver enviando mensagens e nada estiver acontecendo, verifique se não está enviando mensagens a um ponteiro que foi definido como **nil**.

Item importante 2: se você enviar uma mensagem para **nil**, o valor de retorno não significará nada e deverá ser desconsiderado.

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

Nesse caso, **daily** será zero. (Em geral, se você esperar um número ou um ponteiro como resultado, enviar uma mensagem para **nil** retornará zero. No entanto, para outros tipos, como **structs**, você obterá valores de retorno estranhos e inesperados.)

id

Ao declarar um ponteiro para conter o endereço de um objeto, na maioria das vezes, você especifica a classe do objeto à qual o ponteiro irá se referir:

```
NSDate *expiration;
```

No entanto, frequentemente você precisará de um modo de criar um ponteiro sem saber exatamente a que tipo de objeto ele irá se referir. Para essa situação, você deve usar o tipo **id** para indicar “um ponteiro para um tipo de objeto do Objective-C”. Veja a aparência disso ao usá-lo:

```
id delegate;
```

Observe que não há asterisco nesta declaração. O asterisco está implícito em **id**.

Desafio

Use duas instâncias de **NSDate** para calcular quantos segundos você já viveu.

Primeiro, **NSDate** possui um método de instância **timeIntervalSinceDate:**. Esse método usa um argumento – outra instância de **NSDate**. Ele retorna o número de segundos entre a **NSDate** que recebeu a mensagem e a **NSDate** que foi passada como o argumento.

Isso será semelhante ao seguinte:

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

Depois, você precisará criar um novo objeto de data definido como um determinado ano, mês etc. Você fará isso com a ajuda de um objeto **NSDateComponents** e de um objeto **NSCalendar**. Veja aqui um exemplo:

```
NSDateComponents *comps = [[NSDateComponents alloc] init];
[comps setYear:1969];
[comps setMonth:4];
[comps setDay:30];
[comps setHour:13];
[comps setMinute:10];
[comps setSecond:0];

NSCalendar *g = [[NSCalendar alloc] initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *dateOfBirth = [g dateFromComponents:comps];
```

Boa sorte!

15

Objetos e memória

Neste capítulo, você aprenderá sobre a existência de objetos no heap e como a memória heap é gerenciada.

Sobre ponteiros e seus valores

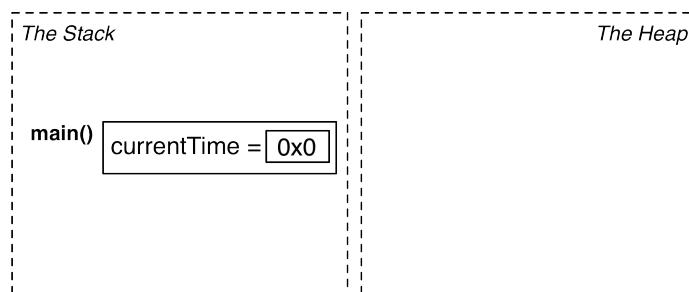
Objetos só podem ser acessados através de um ponteiro, e é prático, embora impreciso, se referir a um objeto por seu ponteiro, como em “now é uma **NSDate**”. Entretanto, é importante se lembrar de que o ponteiro e o objeto ao qual ele aponta não são a mesma coisa. Veja uma instrução mais precisa: “now é um ponteiro que pode reter um endereço de um local na memória onde uma instância de **NSDate** reside”.

Crie uma nova Command-line Tool chamada **TimesTwo**. Faça seu tipo ser Foundation. No **main.m**, declare uma variável que aponte para uma instância de **NSDate**.

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSDate *currentTime = nil;
        NSLog(@"%@", @"currentTime's value is %p", currentTime);
    }
    return 0;
}
```

Aqui você inicializou a variável de ponteiro para **nil**. Execute o programa, e você verá que **currentTime** aponta para **0x0**, que é o valor de **nil**.

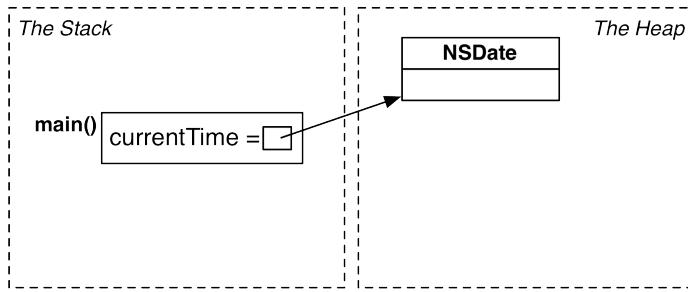


Esse diagrama mostra a variável local **currentTime** que faz parte do frame para **main()**. Seu valor atual é **nil**, e ainda não há objetos criados no heap.

Em seguida, crie uma **NSDate** para **currentTime** apontar para ela em vez de apontar para **nil**.

```
...
@autoreleasepool {
    NSDate *currentTime = [NSDate date];
    NSLog(@"%@", @"currentTime's value is %p", currentTime);
}
return 0;
}
```

Compile e execute o programa. O resultado informará o endereço do objeto apontado por **currentTime**. Um objeto **NSDate** existe agora no heap.



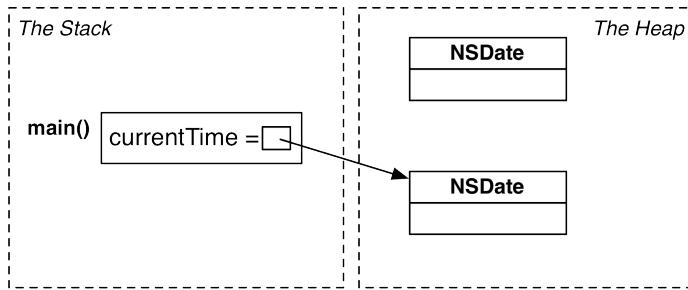
`currentTime` é uma variável, portanto, você pode mudá-la para que aponte para uma outra `NSDate`. Faça o programa ser suspenso por dois segundos depois da primeira instrução de registro e então apontar `currentTime` para a segunda instância de `NSDate`.

```
@autoreleasepool {
    NSDate *currentTime = [NSDate date];
    NSLog(@"currentTime's value is %p", currentTime);

    sleep(2);

    currentTime = [NSDate date];
    NSLog(@"currentTime's value is now %p", currentTime);
}
return 0;
}
```

Compile e execute o programa. Dois segundos depois da primeira linha de resultado, você verá uma segunda linha informando um endereço diferente para `currentTime`. `currentTime` agora aponta para um `NSDate` diferente:



E quanto ao objeto de data original? Da perspectiva de seu código, esse objeto e a informação que ele continha estão perdidos. Se você perder o seu único ponteiro para um objeto, você não poderá mais acessá-lo – ainda que ele continue existindo no heap.

Se você quisesse mudar o valor de `currentTime` e ainda ser capaz de acessar a data original, você poderia declarar outro ponteiro para armazenar o endereço da data original.

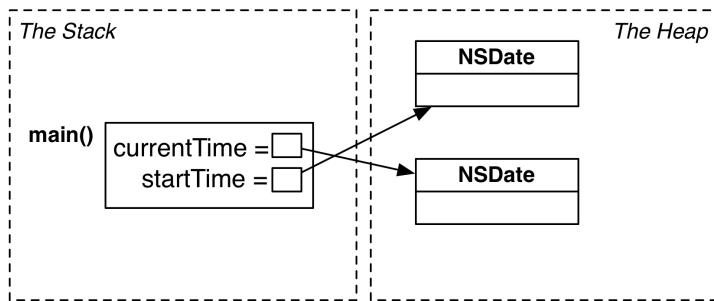
```
@autoreleasepool {
    NSDate *currentTime = [NSDate date];
    NSLog(@"currentTime's value is %p", currentTime);

    NSDate *startTime = currentTime;

    sleep(2);

    currentTime = [NSDate date];
    NSLog(@"currentTime's value is now %p", currentTime);
    NSLog(@"The address of the original object is %p", startTime);
}
return 0;
}
```

Compile e execute o programa.



Agora, vamos ver essa progressão de código da perspectiva do gerenciamento de memória.

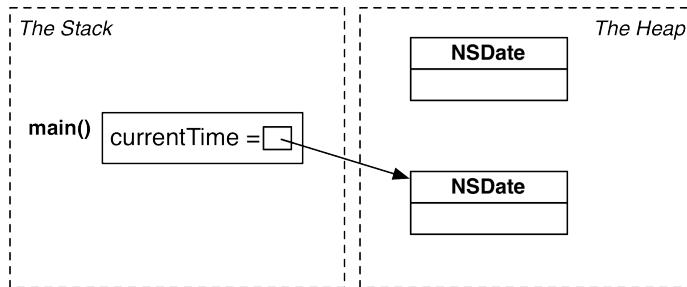
Gerenciamento de memória

Quando falamos de gerenciamento de memória, estamos falando sobre o gerenciamento de memória heap. Considere a diferença entre a pilha e o heap. Lembre-se do Chapter 5, em que a pilha consiste em frames empilhados de maneira ordenada. Cada frame é automaticamente desalocado quando a função que o utiliza termina. O heap, por outro lado, é um amontoado de memória, e é lá que seus objetos residem.

Gerenciar o heap é importante porque os objetos podem ser grandes e porque seu programa recebe uma memória heap limitada para seu próprio uso. Cada objeto criado consome uma parte dessa memória.

A execução com pouca memória é um problema. Isso causará o mau desempenho de um aplicativo Mac e fará um aplicativo iOS falhar. Portanto, é essencial que todo objeto que não seja mais necessário seja destruído para que sua memória seja recuperada e reutilizada.

Observe o programa novamente depois que você mudar `currentTime` pela primeira vez para um valor novo e antes que você tenha `startTime`.

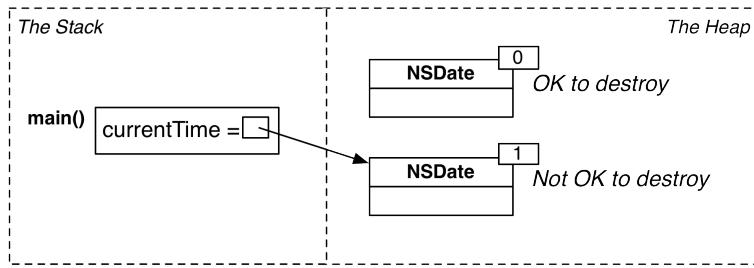


Nada aponta para o objeto de data original. Do ponto de vista do gerenciamento de memória, esse objeto é inútil e está ocupando memória heap valiosa. Ele precisa ser destruído.

ARC

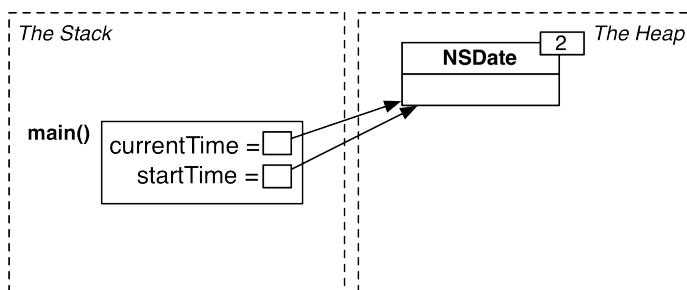
A configuração que instrui o compilador a garantir a destruição de objetos sem referência se chama ARC. ARC é uma sigla do inglês Automatic Reference Counting, que significa contagem de referência automática. Se você lembrar de que “referência” é outra palavra para ponteiro, torna-se fácil entender para que serve a ARC: Cada objeto mantém uma contagem de quantas referências a si mesmo existem. Quando essa contagem de referências chega a zero, o objeto sabe que não é mais necessário e se destrói. Quando o seu projeto tem a ARC habilitada, o compilador acrescenta código ao seu projeto para avisar a cada objeto toda vez que ganha ou perde uma referência. Em outra época, os desenvolvedores precisavam escrever código para manter a contagem de referências de um objeto atualizada – por isso a palavra “automática” em contagem de referência automática.

Quando você altera `currentTime` para que aponte para um novo objeto, o objeto original perde uma referência, e a ARC diminui sua contagem de referências. A contagem de referência do novo objeto de data é aumentada.

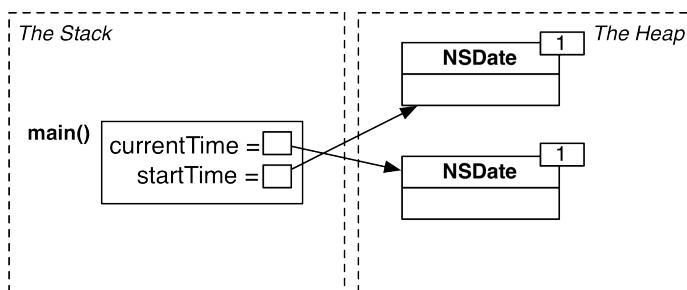


Já que `currentTime` era a única referência para a `NSDate` original, o objeto será destruído para que sua memória possa ser utilizada para outra coisa.

Quando você cria o ponteiro `startTime` e dá a ele o mesmo valor de `currentTime`, o objeto de data ganha outra referência.



Quando você altera `currentTime` para que aponte para uma nova data, a nova data ganha uma referência e a data original perde uma referência.



Dessa vez, porém, a data original ainda tem outra referência. Portanto, você ainda tem acesso a esse objeto e ele ainda existe.

Assim, enquanto você tiver um ponteiro para um objeto, pode ter certeza de que ele continuará existindo. Você nunca destruirá explicitamente um objeto, como você destruiria um buffer com `free()` (como você aprendeu no Chapter 12). Você pode apenas adicionar ou remover uma referência ao objeto. O objeto se destruirá quando sua contagem de referências chegar a zero.

E se você não precisar mais de um objeto? Você define o ponteiro para `nil`, ou deixa que o ponteiro seja destruído ao sair do escopo. Para ilustrar o que acontece, vá em frente e invalide `currentTime` manualmente em seu programa:

```

@autoreleasepool {
    NSDate *currentTime = [NSDate date];
    NSLog(@"currentTime's value is %p", currentTime);
    NSDate *startTime = currentTime;

    sleep(2);

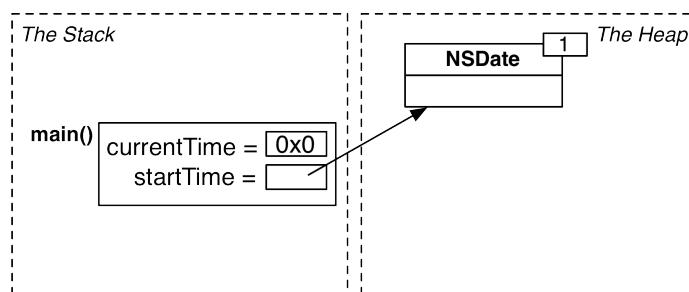
    currentTime = [NSDate date];
    NSLog(@"currentTime's value is now %p", currentTime);
    NSLog(@"The original object lives at %p", startTime);

    currentTime = nil;
    NSLog(@"currentTime's value is %p", currentTime);
}
return 0;
}

```

Compile e execute o programa. Ao final dos resultados, você verá que o valor de `currentTime` voltou para `0x0`.

Definir `currentTime` como `nil` fará com que o objeto `NSDate` perca uma referência e, nesse caso, seja destruído.



Um objeto também perde uma referência quando a variável ponteiro em si é destruída. As coisas ficam um pouco mais complicadas quando você tem objetos com variáveis de instância apontando para outros objetos. Você começará a aprender a lidar com esses casos no Chapter 21.

16

NSString

NSString é outra classe como **NSDate**. Instâncias de **NSString** contêm strings de caracteres. Desenvolvedores de Objective-C usam instâncias de **NSString** para reter e manipular texto em seus programas.

Criação de instâncias de NSString

No código, você pode criar uma instância de **NSString** da seguinte forma:

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

Observe que não há nenhuma mensagem explícita enviada para que a classe **NSString** crie a instância. O @"..." é um atalho do Objective-C para a criação de um objeto **NSString** com a string de caracteres em questão. Esse atalho é conhecido como sintaxe literal. Quando você a usa, dizemos que está criando uma instância *literal* de **NSString**, ou, mais comumente, uma **NSString** literal.

Instâncias de **NSString** podem conter qualquer caractere Unicode. Para inserir caracteres não ASCII, use \u seguido pelo número Unicode do caractere em questão em hexadecimal. Por exemplo, o símbolo para o naipe de copas branco no baralho é 0x2661:

```
NSString *slogan = @"I \u2661 New York!" ;
```

Como objetos **NSString** podem conter caracteres Unicode, eles facilitam a criação de aplicativos que lidem com strings de muitas linguagens.

Frequentemente, você precisará criar strings *dinamicamente*. Ou seja, você precisará criar uma string cujo conteúdo não será conhecido até que o programa esteja sendo executado. Para criar uma instância de **NSString** dinamicamente, você pode usar o método de classe **stringWithFormat:**.

```
NSString *dateString = [NSString stringWithFormat:@"The date is %@", now];
```

Na mensagem **stringWithFormat:**, você envia como argumento uma string de formatação com um ou mais tokens e a(s) variável/variáveis cujos valores serão usados no lugar do(s) token(s). Funciona da mesma maneira que a string de formatação que você vem passando para a função **NSLog**.

Métodos NSString

NSString é uma classe que desenvolvedores usam muito. Como todas as classes Objective-C, ela vem com métodos úteis. Se você quiser fazer algo com uma string, é provável que haja um método **NSString** que possa ajudar.

A seguir são apresentados alguns exemplos de métodos **NSString**. Para introduzir esses métodos, vamos mostrar a você a declaração do método e, em seguida, um exemplo dele sendo usado. A declaração lhe informa o que você precisa saber sobre um método: se ele é um método de instância ou de classe, o que ele retorna, seu nome e os tipos de seus argumentos, se houver.

Para obter o número de caracteres em uma string, você usa o método **length**:

```
- (NSUInteger)length;
```

Esse método é um método de instância. Você pode comprovar isso pelo "-" no início da declaração. (Um método de classe teria um "+" em vez disso.) Esse método retorna um **NSUInteger** e não tem nenhum argumento. **NSUInteger** é um tipo no framework Foundation. Ele é equivalente ao tipo **unsigned long** sobre o qual você aprendeu no Chapter 7.

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

Para ver se uma string é igual a outra, você pode usar o método **isEqualToString::**

```
- (BOOL)isEqualToString:(NSString *)other;
```

Esse método de instância irá percorrer as duas strings comparando-as caractere por caractere para ver se são iguais. Seu único argumento é a string que você quer comparar com a string que receberá a mensagem **isEqualToString::**. O método retorna um BOOL que informa se as duas strings são iguais de fato.

```
if ([slogan isEqualToString:lament]) {  
    NSLog(@"%@", slogan, lament);  
}
```

Para obter uma versão em maiúsculas de uma string, você usa o método **uppercaseString**.

```
- (NSString *)uppercaseString;
```

Esse método de instância retorna uma instância de **NSString** que é equivalente ao destinatário, só que totalmente em maiúsculas:

```
NSString *angryText = @"That makes me so mad!";  
NSString *reallyAngryText = [angryText uppercaseString];
```

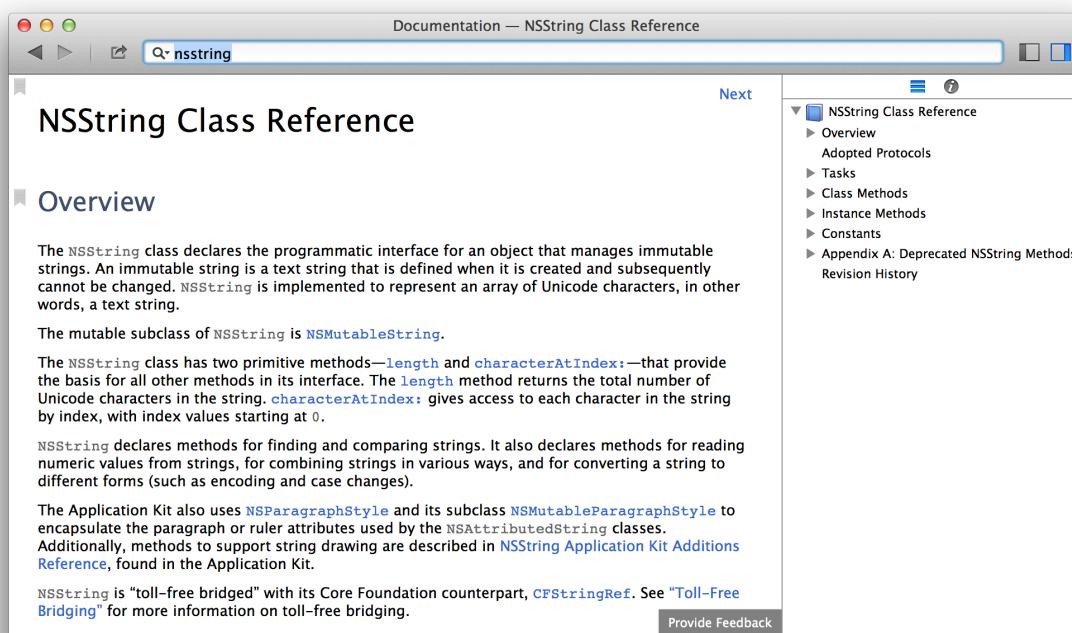
Referências de classe

Então, onde você encontra os métodos de que precisa? A Apple mantém uma *referência de classe* para cada classe em seus APIs. A referência de classe lista todos os métodos de uma classe e informações básicas sobre como usá-los.

No Xcode, selecione Help → Documentation and API Reference. Isso abrirá o navegador de documentação do Xcode.

No campo de pesquisa no topo da janela, digite **NSString**.

Figure 16.1 Referência de classe **NSString**

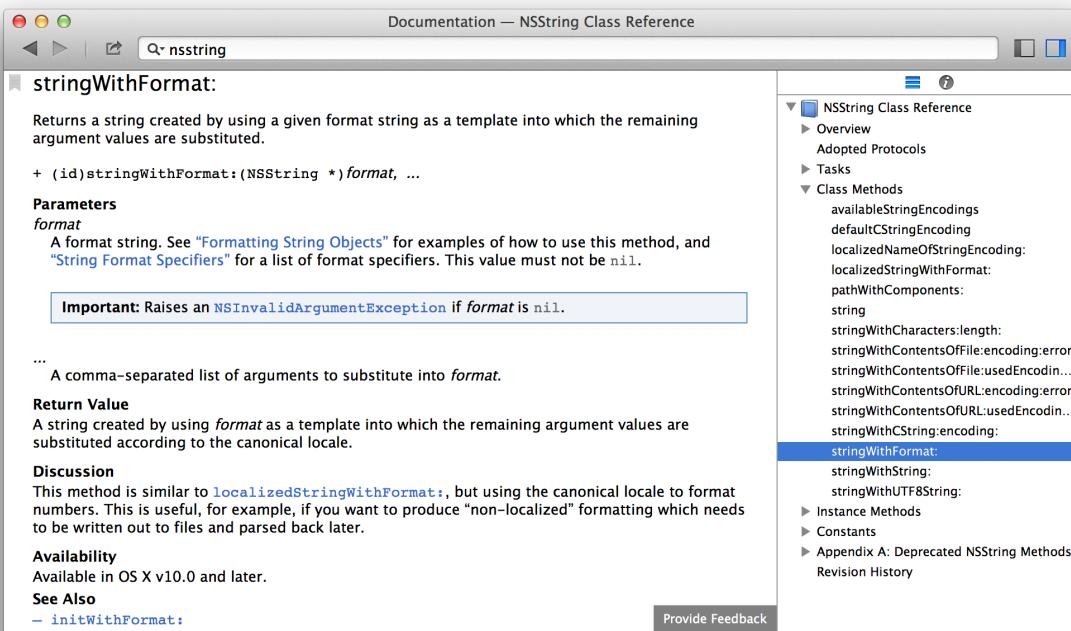


(Você também pode acessar a documentação por meio do website do desenvolvedor da Apple. Para chegar à referência de classe **NSString**, simplesmente faça uma busca por “referência de classe **NSString**”. O primeiro resultado retornado costuma ser a página de referência de **NSString** em developer.apple.com.)

No painel do lado direito, há o índice da referência de classe **NSString**. A Overview descreve a classe **NSString** no geral. Há também cabeçalhos que listam os métodos de classe e métodos de instância. Se você souber o nome do método que está procurando, pode encontrá-lo pelo nome sob um desses cabeçalhos e ler tudo a respeito dos detalhes dele.

Revele o conteúdo da categoria Class Methods. Encontre e selecione **stringWithFormat:** da lista para ver informações úteis sobre esse método, tal como descrições de seus parâmetros e valor de retorno.

Figure 16.2 Documentação para **stringWithFormat:**

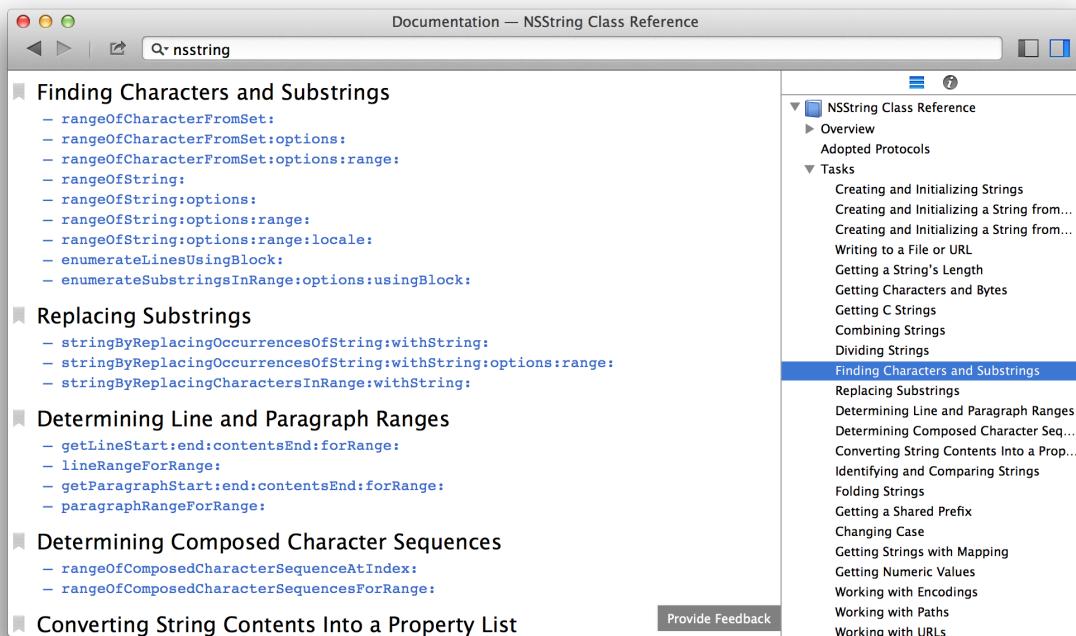


Se você precisar fazer algo com um objeto, mas não conhecer um método específico, o melhor lugar por onde começar é o título **Tasks**. Uma tarefa que desenvolvedores costumam precisar realizar com **NSString** é pesquisar uma string para ver se ela contém uma determinada *substring*. Uma substring é uma string que pode compor parte ou toda a outra string.

Por exemplo, digamos que você leia uma lista de nomes delimitada por vírgulas como um objeto **NSString**. Agora, você precisa verificar se um nome específico está na lista. Esse nome específico seria uma substring da string maior.

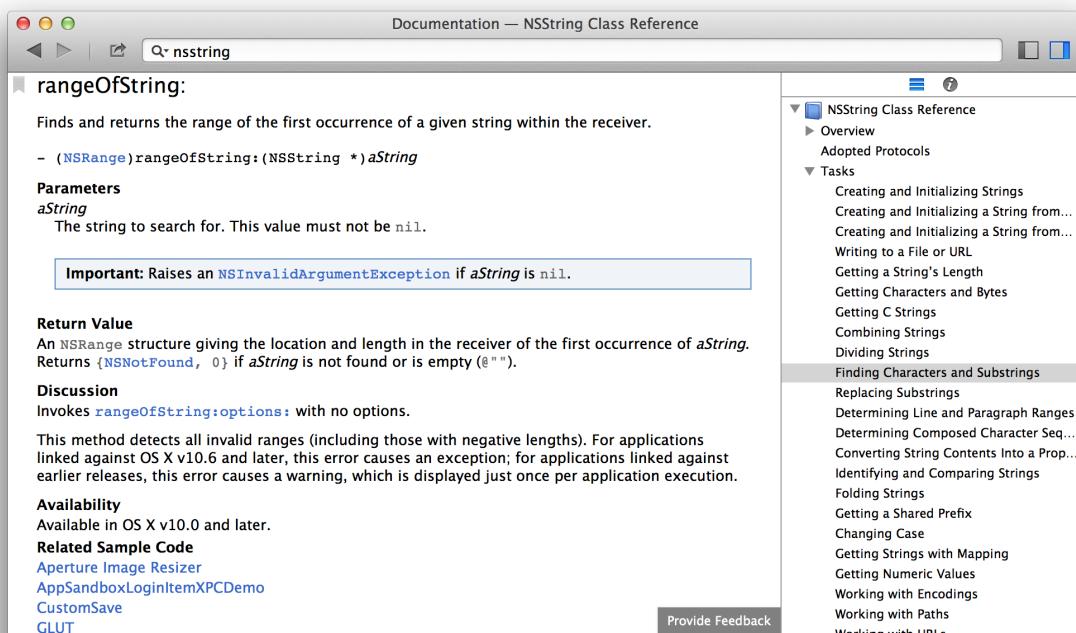
Revele o conteúdo sob o título Tasks. Encontre e selecione Finding Characters and Substrings. Isso revelará vários métodos potencialmente úteis.

Figure 16.3 Métodos para localizar caracteres e substrings



No mundo real, você navegaria pelos detalhes de métodos possíveis até encontrar um que funcionasse. Para este exemplo, daremos uma ajudinha a você: clique em **rangeOfString:** na lista de métodos para ver seus detalhes (Figure 16.4).

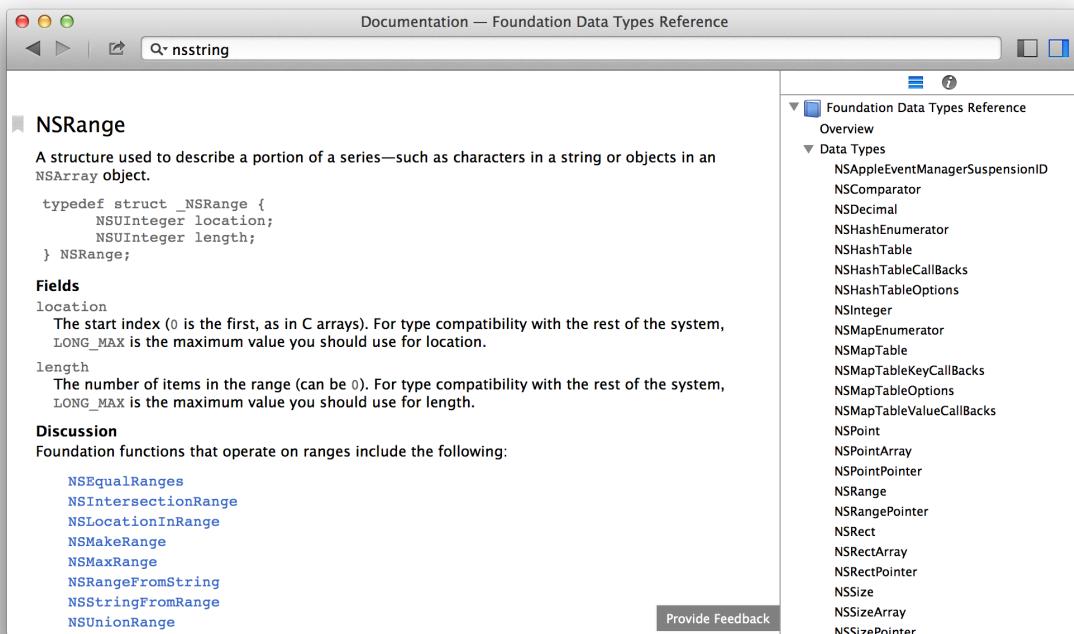
Figure 16.4 Documentação para **rangeOfString:**



Você pode ver que **rangeOfString:** tem um parâmetro que é uma instância de **NSString**. Essa é a “substring” pela qual você quer buscar – o único nome a ser encontrado na lista de nomes.

Você também pode ver que esse método retorna uma **NSRange**. O que é **NSRange**? Clique em **NSRange** para visualizar sua definição (Figure 16.5).

Figure 16.5 Documentação para **NSRange**



NSRange é um **typedef** para uma **struct**, como você usou no Chapter 11. Ela tem dois membros, **location** e **length**, que você pode usar para localizar uma substring dentro de uma string.

Contudo, no problema atual, você só quer ver se o nome ocorre na lista ou não. Para saber como fazer isso, pressione o botão de voltar no canto superior esquerdo do navegador da documentação para retornar à página anterior. Em seguida, encontre a seção **Return Value** na documentação **rangeOfString:**. Essa seção afirma que, quando a substring passada não ocorre, **rangeOfString:** retorna um **NSRange** cuja **location** é a constante **NSNotFound**.

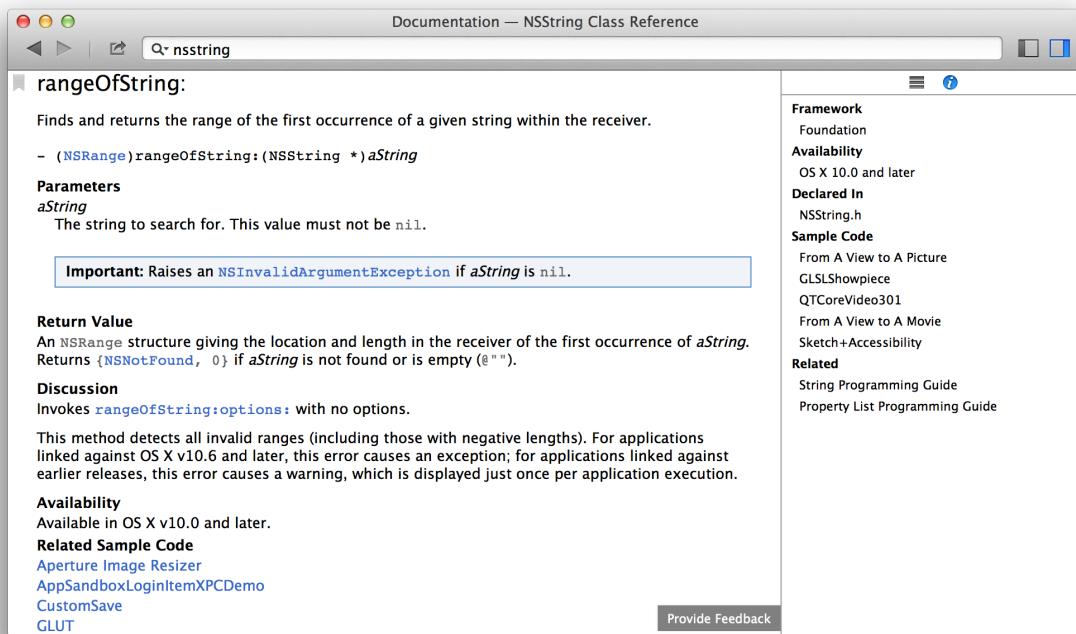
Assim, para determinar se o nome está na lista de nomes, você pode simplesmente verificar o membro **location** do valor de retorno. O código teria a seguinte aparência:

```
NSString listOfNames = @"..."; // a long list of names
NSString name = @"Ward";
NSRange match = [listOfNames rangeOfString:name];
if (match.location == NSNotFound) {
    NSLog(@"No match found!");
    // Other actions to be taken
} else {
    NSLog(@"Match found!");
    // Other actions to be taken
}
```

Outras partes da documentação

Antes de você fechar a documentação, vamos dar uma olhada em mais alguns itens que podem ser especialmente úteis para novos desenvolvedores de Objective-C. Retorne ao topo do painel direito e clique no botão ⓘ para revelar alguns detalhes básicos sobre a classe **NSString**.

Figure 16.6 Detalhes da **NSString**



Abaixo desses detalhes, há links para outras partes da documentação da Apple relacionada à **NSString**.

Sob o título **Sample Code**, há projetos pequenos e completos que demonstram como a Apple espera que a classe em questão seja usada. Muitas classes (especialmente as comumente usadas, como a **NSString**) têm links de código de amostra em suas páginas de referência.

Sob o título **Related**, há dois *guias do desenvolvedor* da Apple. Esses guias são organizados por tópico em vez de por classe ou método, e, por isso, são excelentes para se aprender a respeito de tópicos específicos em Objective-C, desenvolvimento para iOS e desenvolvimento para OS X.

Você pode navegar por todos os guias do desenvolvedor em <https://developer.apple.com/library>. Selecione iOS ou OS X para ir à biblioteca do desenvolvedor da plataforma em questão. As duas plataformas compartilham o framework Foundation; assim, tudo o que você estiver aprendendo estará em alguma das bibliotecas.

Selecione Guides do índice à esquerda para ver uma lista de guias com um útil campo de pesquisa no topo. Ou você pode selecionar Getting Started para ver um grupo menor de guias tutoriais iniciais.

É difícil exagerar ao falar da importância que a documentação da Apple terá para você e da importância dela para programadores de todos os níveis. À medida que você lê este livro, reserve um tempo para pesquisar novas classes e métodos conforme os encontra e veja o que mais eles podem fazer. Além disso, leia os guias do desenvolvedor e faça download de projetos de código de amostra que despertem seu interesse. Quanto mais à vontade você se sentir com o uso da documentação, mais rápido será seu desenvolvimento.

Desafio: encontrar mais métodos NSString

O método **rangeOfString:** faz distinção entre maiúsculas e minúsculas. Retorne à referência de classe **NSString** e encontre o método que você usaria se precisasse fazer uma busca sem distinção de maiúsculas e minúsculas.

Depois, encontre o método **NSString** que retornará a parte exata da string que foi encontrada.

Desafio: usar readline()

O valor de retorno da função **readline** do Chapter 8 é do tipo `const char *`, ou uma string em C. É possível obter uma instância de **NSString** com os mesmos caracteres de qualquer string em C específica enviando a mensagem de classe **stringWithUTF8String:** à classe **NSString** e passando a string em C como seu argumento.

Reescreva o desafio **readline()** do Chapter 8 para que use uma **NSString** e **NSLog()**, em vez de uma string em C e **printf()**. Será mais interessante que você crie uma nova Foundation Command Line Tool.

17

NSArray

NSArray é outra classe de Objective-C comumente usada. Uma instância de **NSArray** mantém uma lista de ponteiros para outros objetos.

Crie um novo projeto: uma Foundation Command Line Tool chamada DateList. Este programa vai criar um array que contém uma lista de ponteiros para objetos **NSDate**.

Criação de arrays

Abra `main.m` e altere `main()`:

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

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

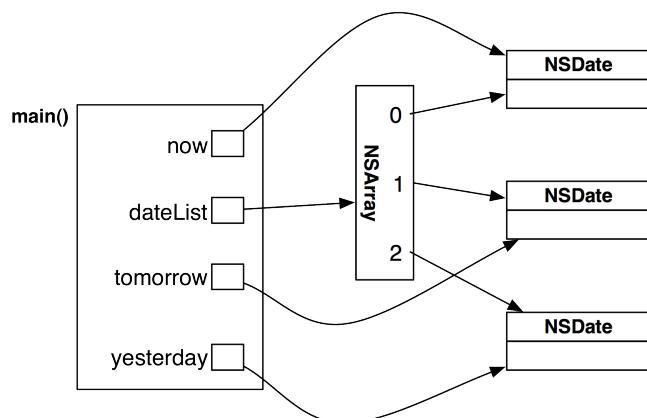
        // Create three NSDate objects
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // Create an array containing all three
        NSArray *dateList = @[now, tomorrow, yesterday];
    }
    return 0;
}
```

Como o **NSString**, o **NSArray** tem uma sintaxe literal para criação de instâncias. O conteúdo do array está em uma lista separada por vírgulas, circundado por colchetes e precedido de `@`. Não é necessário o envio de mensagem explícita.

Figure 17.1 é um diagrama de objetos de seu programa. Observe que a instância de **NSArray** contém ponteiros para os objetos **NSDate**.

Figure 17.1 Diagrama de objetos para DateList



Uma instância de **NSArray** é *imutável*. Assim que uma **NSArray** tiver sido criada, você nunca poderá adicionar ou remover um ponteiro desse array. Nem poderá alterar a ordem dos ponteiros naquele array.

Acesso aos arrays

Arrays são listas ordenadas; você pode acessar um item em um array por meio de seu *índice*. Os arrays têm base zero: o primeiro item é armazenado no índice 0, o segundo item é armazenado no índice 1 e assim por diante.

Você pode acessar um item individual no array utilizando o array seguido pelo índice do item entre colchetes. Adicione o código a seguir em seu programa para acessar e imprimir dois itens no array:

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

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

        // Create three NSDate objects
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // Create an array containing all three
        NSArray *dateList = @[now, tomorrow, yesterday];

        // Print a couple of dates
        NSLog(@"The first date is %@", dateList[0]);
        NSLog(@"The third date is %@", dateList[2]);

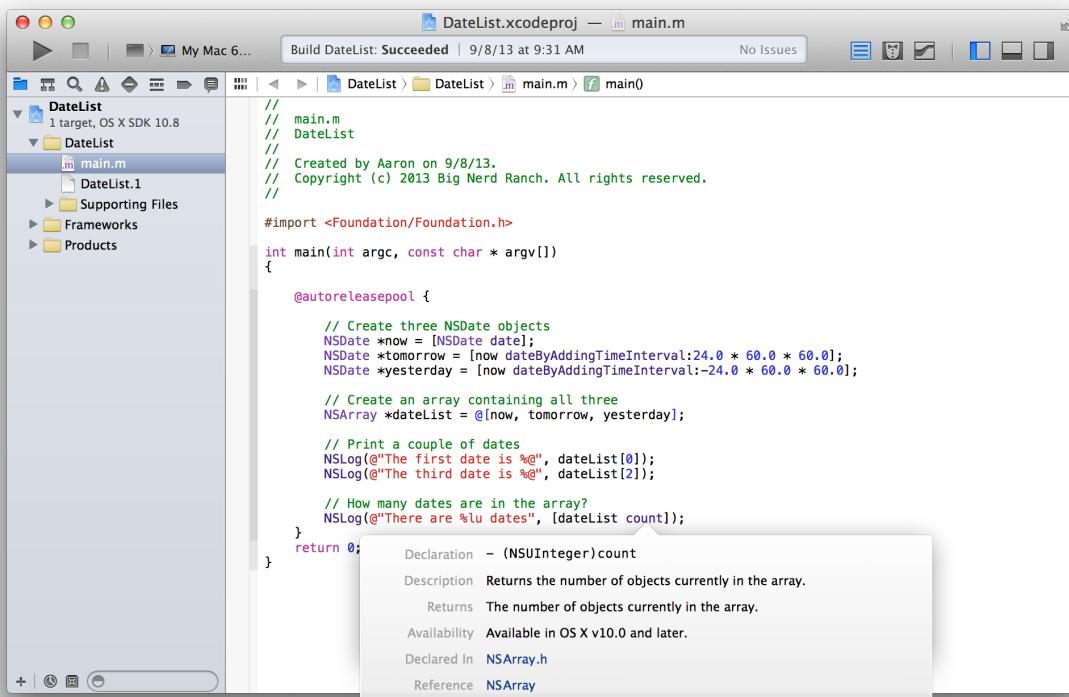
        // How many dates are in the array?
        NSLog(@"There are %lu dates", [dateList count]);
    }
    return 0;
}
```

Compile e execute o programa e verifique seu resultado.

Você enviou a `dateList` o `count` da mensagem. Para descobrir o que o método `count` faz, você pode ir à página de referência de classe da **NSArray**. Mas há uma maneira de obter um resumo rápido e imediato no Xcode.

Pressionando a tecla Option (Opção), clique em `count`. A janela Quick Help (Ajuda rápida) será exibida, trazendo informações sobre esse método:

Figure 17.2 Janela pop-up Quick Help



Observe que há links na janela Quick Help. Se você clicar em um link, ele abrirá a documentação apropriada no navegador da documentação do Xcode.

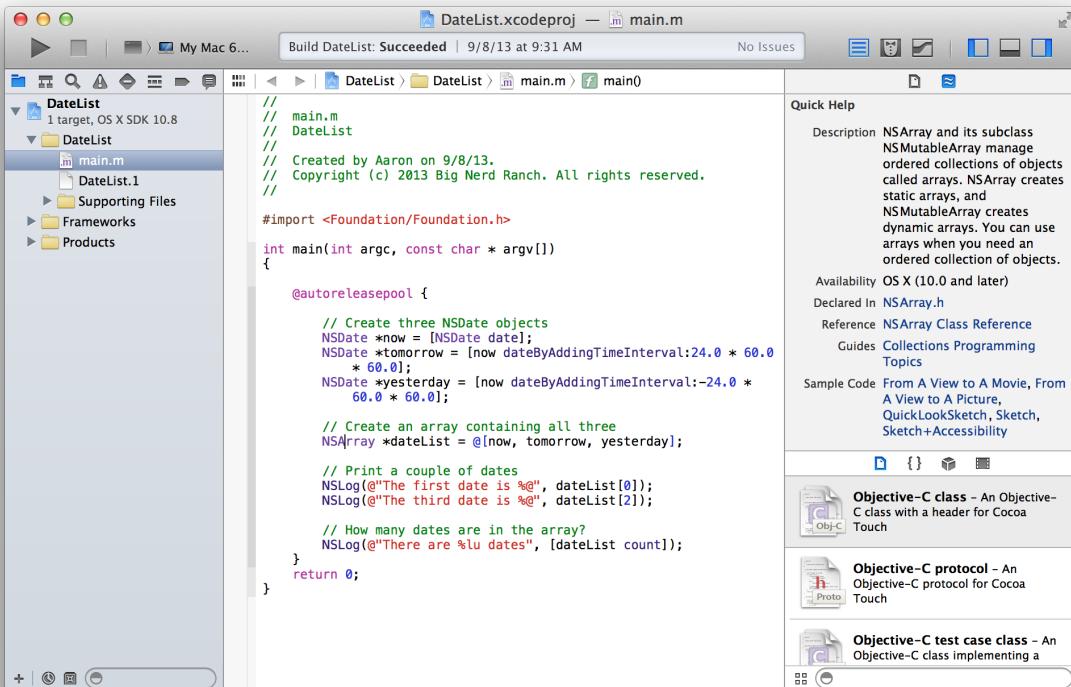
Também é possível abrir a Quick Help na área de utilitários do Xcode para ver suas informações a todo o momento.

No canto superior direito da janela do Xcode, clique no botão mais à direita desse grupo: .

Isso revelará a área de utilitários. Na parte superior da área de utilitários, clique no botão para revelar a Quick Help.

No seu código, clique em **NSArray** para ver sua documentação:

Figure 17.3 Quick Help na área de utilitários



Ao mover o cursor para algum outro lugar, a Quick Help é atualizada imediatamente caso haja uma documentação disponível.

Vamos retornar ao propósito do método **count**. Você viu que esse método recupera o número de itens no array. Saber a contagem de itens em um array é mais importante do que você pensa. Se o método **count** reportar que há 100 itens no array, então você pode solicitar por itens nos índices 0 a 99. Se você solicitar por um item com um índice acima de 99, será exibido um *erro de fora da faixa* que causará uma falha no seu programa.

Para ver um exemplo de um erro de fora da faixa, adicione a seguinte linha de código fatal:

```

#import <Foundation/Foundation.h>

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

        ...

        // How many dates are in the array?
        NSLog(@"There are %lu dates", [dateList count]);

        NSLog(@"The fourth date is %@", dateList[3]); // Crash!
    }
    return 0;
}

```

Você está solicitando o objeto no índice 3 (o quarto objeto) na `dateList` quando a `dateList` tem apenas três objetos. Compile e execute o seu programa. Quando houver uma falha nessa linha, primeiro interrompa o programa usando o botão Stop (Parar) no canto superior esquerdo do Xcode. Em seguida, exclua a linha problemática.

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

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        ...
        // How many dates are in the array?
        NSLog(@"There are %lu dates", [dateList count]);
        NSLog(@"The fourth date is %@", dateList[3]); // Crash!
    }
    return 0;
}
```

Compile e execute novamente para confirmar se o problema foi corrigido. Erros de fora da faixa são comuns com programadores iniciantes. Use o método `count` como uma verificação e lembre-se de que os arrays são sempre baseados em zero.

Iteração em arrays

Os programadores normalmente precisam realizar um loop e executar operações em cada item em um array (ou “fazer iteração em um array”). Você pode fazer isso com um `for`-loop. Edite o `main.m`:

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

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        ...
        // Create three NSDate objects
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // Create an array containing all three
        NSArray *dateList = @[now, tomorrow, yesterday];

        ...
        // Print a couple of dates
        NSLog(@"The first date is %@", dateList[0]);
        NSLog(@"The third date is %@", dateList[2]);

        ...
        // How many dates are in the array?
        NSLog(@"There are %lu dates", [dateList count]);

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

No `for`-loop, observe que você utiliza a contagem de itens do array para limitar o número de vezes que o loop vai executar, a fim de evitar erros de fora da faixa.

Os programadores fazem iteração em arrays com tal frequência que realizam uma adição especial ao `for`-loop chamado *enumeração rápida*. Esse tipo de loop é uma maneira extremamente eficiente para percorrer pelos itens em um array. Ao utilizar a enumeração rápida, a verificação da contagem de itens do array é feita por você. Edite seu código para usar a enumeração rápida:

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

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

        // Create three NSDate objects
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // Create an array containing all three
        NSArray *dateList = @[now, tomorrow, yesterday];

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

Compile e execute seu programa. O resultado será o mesmo de antes, mas o seu código é mais simples e eficiente.

NSMutableArray

Uma instância de **NSMutableArray** é parecida a uma instância de **NSArray**, mas você pode adicionar, remover e reordenar os ponteiros. (A **NSMutableArray** é uma *subclasse* de **NSArray**. Você saberá mais sobre subclasses no Chapter 20.)

Altere seu programa para usar uma **NSMutableArray** e métodos da classe **NSMutableArray**:

```

#import <Foundation/Foundation.h>

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

        // Create three NSDate objects
        NSDate *now = [NSDate date];
        NSDate *tomorrow = [now dateByAddingTimeInterval:24.0 * 60.0 * 60.0];
        NSDate *yesterday = [now dateByAddingTimeInterval:-24.0 * 60.0 * 60.0];

        // Create an array containing all three
        NSArray *dateList = @[now, tomorrow, yesterday];

        // Create an empty mutable array
        NSMutableArray *dateList = [NSMutableArray array];

        // Add two dates to the array
        [dateList addObject:now];
        [dateList addObject:tomorrow];

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

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

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

```

Você utilizou o método de classe **array** para criar **NSMutableArray**. Esse método recupera um array vazio, para o qual você pode adicionar objetos. Você também pode usar o **alloc** e o **init** para obter o mesmo resultado:

```
NSMutableArray *dateList = [[NSMutableArray alloc] init];
```

Você utilizou o método **addObject:** para preencher a **NSMutableArray**. Esse método adiciona o objeto no final da lista. Para adicionar um objeto em um índice específico, você pode usar o **insertObject:atIndex:**. Conforme os objetos vão sendo adicionados, o array aumentará de acordo com a necessidade de contê-los.

Você removeu um objeto do array usando o **removeObjectAtIndex:**. Uma contagem de itens do array será alterada à medida que os objetos forem removidos. Por exemplo, se você solicitasse o objeto no índice 2 na **dateList** após remover o ponteiro **yesterday**, o programa apresentaria uma falha.

Para referência futura, ao utilizar a enumeração rápida com uma **NSMutableArray**, você não tem permissão de adicionar ou remover itens enquanto faz iteração em array. Se precisar adicionar ou remover itens enquanto faz iteração, use um for-loop padrão.

Métodos antigos de array

Antes de uma sintaxe literal ser apresentada para criar instâncias de **NSArray**, os desenvolvedores usavam método de classe **arrayWithObjects:**.

```
// Create an array containing three pointers (nil terminates the list)
NSArray *dateList = [NSArray arrayWithObjects:now, tomorrow, yesterday, nil];
```

O **nil** no final pede ao método que pare. Assim, o array de data tem três objetos. (Se você esquecer o **nil**, ele provavelmente trará falhas ao seu programa, mas você receberá, pelo menos, um aviso do compilador.)

A sintaxe que você utilizou para acessar os itens no array **dateList** é conhecida como *subíndice*. Antes de um subíndice ser apresentado, os desenvolvedores usavam o método **objectAtIndex:** para acessar um item em um array:

```
// Print a couple of dates
NSLog(@"The first date is %@", [dateList objectAtIndex:0]);
NSLog(@"The third date is %@", [dateList objectAtIndex:2]);
```

Os métodos **arrayWithObjects:** e **objectAtIndex:** ainda existem e são aprovados. Sinta-se à vontade para usar a sintaxe literal e o subíndice ou os métodos de estilo antigo ao trabalhar com arrays.

Um problema que pode ocorrer com a sintaxe literal e o subíndice é que as diferentes utilizações dos colchetes podem dificultar a leitura do seu código. Considere que você agora tem três usos distintos para os colchetes:

- | | |
|---|--|
| 1. enviar mensagens | NSUInteger dateCount = [dateList count]; |
| 2. criar uma NSArray | NSArray *dateList = @[now, tomorrow, yesterday]; |
| 3. solicitar o item em um índice particular de um array | NSDate *firstDate = dateList[0]; |

Às vezes em seu código, essas utilizações diferentes serão combinadas. Quando isso acontecer, reverter para os métodos de array de estilo antigo pode facilitar a leitura do seu código. Por exemplo, isto pode ser confuso:

```
id selectedDog = dogs[[tableView selectedRow]];
```

Em tais situações, pode ser útil usar o acesso ao array de estilo antigo. Aqui temos a mesma linha de código reescrita:

```
id selectedDog = [dogs objectAtIndex:[tableView selectedRow]];
```

Desafio: uma lista de itens de mercearia

Crie uma nova Foundation Command Line Tool chamada Groceries. Comece criando um objeto **NSMutableArray** vazio. Depois, adicione diversas strings relacionadas a uma mercearia em um array. (Você também precisará criá-las.) Por fim, use a enumeração rápida para exibir sua lista de itens de mercearia.

My grocery list is:
Loaf of bread
Container of milk
Stick of butter

Desafio: nomes interessantes

Esse desafio é mais difícil. Leia o programa a seguir, que localiza nomes próprios comuns que contêm duas letras A adjacentes.

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

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

        // Read in a file as a huge string (ignoring the possibility of an error)
        NSString *nameString =
            [NSString stringWithContentsOfFile:@"/usr/share/dict/propnames"
                encoding:NSUTF8StringEncoding
                error:NULL];

        // Break it into an array of strings
        NSArray *names = [nameString componentsSeparatedByString:@"\n"];

        // Go through the array one string at a time
        for (NSString *n in names) {

            // Look for the string "aa" in a case-insensitive manner
            NSRange r = [n rangeOfString:@"AA" options:NSCaseInsensitiveSearch];

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

O arquivo `/usr/share/dict/propnames` vem pré-instalado no seu Mac. Ele contém nomes próprios comuns. O arquivo `/usr/share/dict/words` contém palavras regulares e nomes próprios. Nos arquivos de palavras, os nomes próprios são grafados com letra maiúscula.

Com base no programa acima, escreva um programa que localize nomes próprios comuns que também sejam palavras regulares; as palavras são apresentadas na lista de nomes próprios que também são apresentadas (em letra minúscula) na lista de palavras regulares.

Por exemplo, se você tivesse apenas estas listas:

(words)	(names)
woldy	Wilson
Wolf	Win
wolf	Winnie
wolfachite	Winston
wolfberry	Wolf
wolfdom	Wolfgang
wolfen	Woody
wolfer	Yvonne
Wolffia	
Wolffian	
Wolffianism	
Wolfgang	
wolfhood	

então, a solução teria apenas um par que corresponde a nossos critérios: “wolf” (lobo, em inglês) na lista de palavras corresponderia a “Wolf” (um sobrenome comum) na lista de nomes.

18

Sua primeira classe

Até agora, você só usou classes criadas pela Apple. Agora, você escreverá sua própria classe. Lembre-se de que uma classe descreve objetos de duas maneiras: variáveis de instância dentro de cada instância da classe e métodos implementados pela classe.

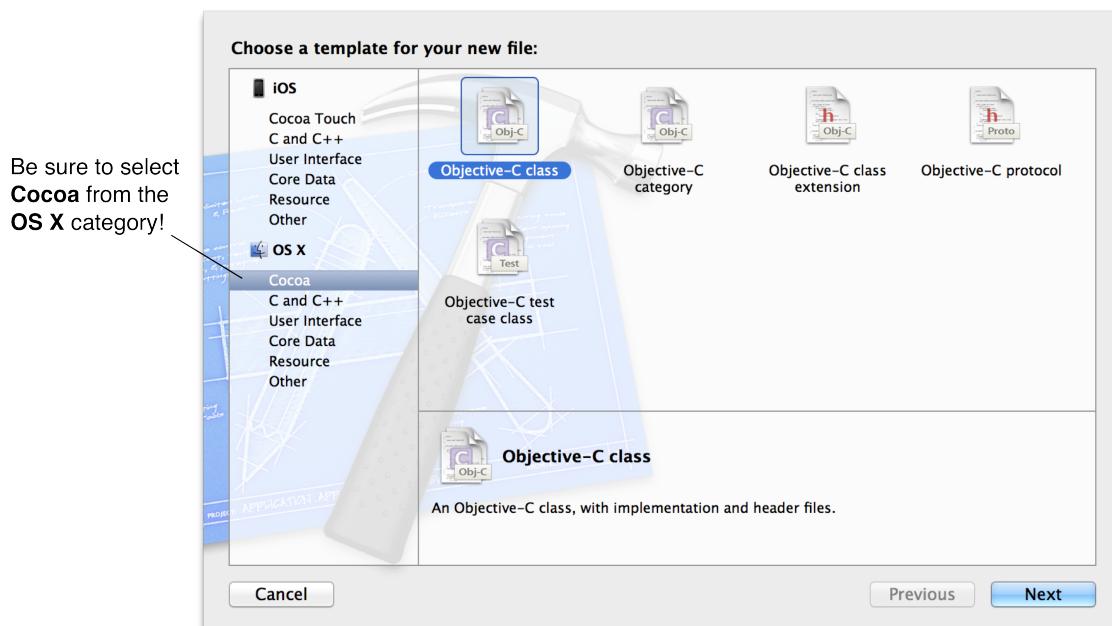
Você vai escrever uma classe **BNRPerson**, que será similar à `struct Person` que você escreveu em Chapter 11. Essa classe, como todas as classes Objective-C, será definida em dois arquivos:

- **BNRPerson.h** é o *cabeçalho* das classes e contém declarações de métodos e variáveis de instância.
- **BNRPerson.m** é o *arquivo de implementação*. É neste local que você escreverá o código, ou *implementará*, cada método.

Primeiramente, crie um novo projeto: uma Foundation Command Line Tool chamada **BMITime**.

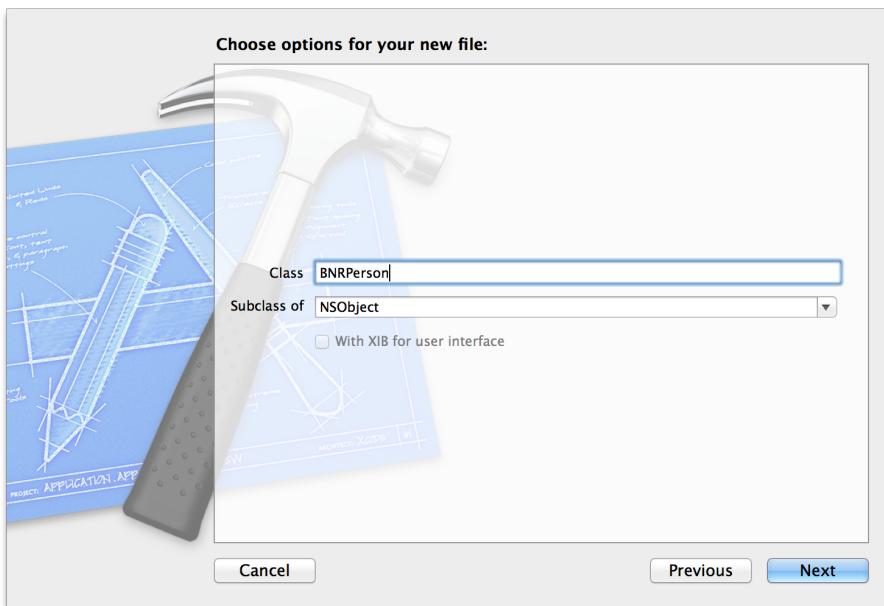
Para criar uma nova classe, selecione **File → New → File...**. A partir da seção OS X à esquerda, selecione **Cocoa**. Escolha o template **Objective-C class** e clique em **Next**.

Figure 18.1 Criação de uma nova classe



Nomeie sua classe como **BNRPerson** e torne-a uma subclasse de **NSObject**. (Você saberá mais sobre subclasses e **NSObject** no Chapter 20.)

Figure 18.2 Nomeando sua nova classe



Clique em Next. Por fim, verifique se o destino `BMITime` está marcado e, então, clique em Create.

Encontre o `BNRPerson.h` e o `BNRPerson.m` no navegador de projetos. Abra o `BNRPerson.h` e declare duas variáveis de instância:

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

@interface BNRPerson : NSObject

{
    // BNRPerson has two instance variables
    float _heightInMeters;
    int _weightInKilos;
}

@end
```

Um arquivo de cabeçalho começa com `@interface` e termina com `@end`. Observe que você declarou as variáveis de instância primeiro e dentro das chaves.

Por convenção, os nomes de variáveis de instância começam com um sublinhado (“`_`”). Usar o prefixo sublinhado permite que você facilmente diferencie variáveis de instâncias de variáveis locais ao ler um código. O sublinhado não significa nada de especial para o compilador; ele é simplesmente o primeiro caractere no nome da variável de instância.

Em seguida, declare cinco métodos de instância após as variáveis de instância e fora das chaves:

```

#import <Foundation/Foundation.h>

@interface BNRPerson : NSObject

{
    // BNRPerson has two instance variables
    float _heightInMeters;
    int _weightInKilos;
}

// BNRPerson has methods to read and set its instance variables
- (float)heightInMeters;
- (void)setHeightInMeters:(float)h;
- (int)weightInKilos;
- (void)setWeightInKilos:(int)w;

// BNRPerson has a method that calculates the Body Mass Index
- (float)bodyMassIndex;

@end

```

Para retornar ao `BNRPerson.m`, use o atalho do teclado Control-Command-Seta para cima. Esse atalho move você para trás e para frente entre o cabeçalho e os arquivos de implementação de uma classe.

Implemente os métodos que você declarou no `BNRPerson.h`. Os métodos que você implementar devem corresponder exatamente àqueles que você declarou no cabeçalho. No Xcode, isso é fácil; quando você começar a digitar um método no arquivo de implementação, o Xcode sugerirá nomes de métodos que você já declarou.

```

#import "BNRPerson.h"

@implementation BNRPerson

- (float)heightInMeters
{
    return _heightInMeters;
}

- (void)setHeightInMeters:(float)h
{
    _heightInMeters = h;
}

- (int)weightInKilos
{
    return _weightInKilos;
}

- (void)setWeightInKilos:(int)w
{
    _weightInKilos = w;
}

- (float)bodyMassIndex
{
    return _weightInKilos / (_heightInMeters * _heightInMeters);
}

@end

```

Agora que você implementou os métodos que declarou no `BNRPerson.h`, a classe `BNRPerson` está completa e você pode usá-la em um programa.

Abra o `main.m` e importe o `BNRPerson.h` de modo que `main()` possa ver as declarações no cabeçalho da classe. `BNRPerson`:

```

#import <Foundation/Foundation.h>
#import "BNRPerson.h"

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

```

Por que o Foundation.h está entre colchetes angulares e o BNRPerson.h entre aspas? Os colchetes angulares mostram ao compilador que o Foundation/Foundation.h é um cabeçalho pré-compilado encontrado nas bibliotecas da Apple. As aspas mostram ao compilador que ele deve procurar por BNRPerson.h no projeto atual.

Em seguida, adicione o código à `main()` que usa a classe `BNRPerson`:

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

        // Create an instance of BNRPerson
        BNRPerson *mikey = [[BNRPerson alloc] init];

        // Give the instance variables interesting values using setters
        [mikey setWeightInKilos:96];
        [mikey setHeightInMeters:1.8];

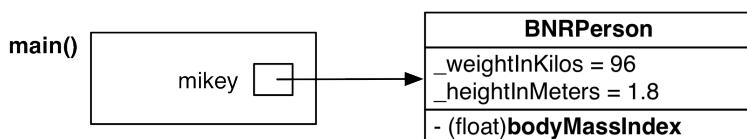
        // Log the instance variables using the getters
        float height = [mikey heightInMeters];
        int weight = [mikey weightInKilos];
        NSLog(@"mikey is %.2f meters tall and weighs %d kilograms", height, weight);

        // Log some values using custom methods
        float bmi = [mikey bodyMassIndex];
        NSLog(@"mikey has a BMI of %f", bmi);

    }
    return 0;
}
```

Compile e execute o programa.

Figure 18.3 Diagrama de objetos de BMITime



Métodos acessores

Quando você criou a struct Person lá no Chapter 11, você acessou os membros de dados da estrutura diretamente em `main()`:

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

No pensamento orientado por objeto, no entanto, o código que está fora de uma classe não deve ser lido ou escrito diretamente em variáveis de instância de uma instância daquela classe. Apenas o código dentro da classe pode fazer isso.

Em vez disso, uma classe vai fornecer métodos que permitem que o código externo (como em `main()`) accessem as variáveis de instância de uma instância. Foi isso que você fez em `BNRPerson`. Em `main()`, você envia mensagens para a instância de `BNRPerson` ler os valores de suas variáveis de instância:

```
int weight = [mikey weightInKilos];
float height = [mikey heightInMeters];
```

Os métodos `heightInMeters` e `weightInKilos` são *métodos getter*. Um método getter, ou *getter*, permite que o código fora de uma classe leia, ou obtenha, o valor de uma variável de instância.

`BNRPerson` também tem métodos `setHeightInMeters:` e `setWeightInKilos:`. Esses são *métodos setter*. Um método setter, ou *setter*, permite que o código fora de uma classe altere, ou defina, o valor de uma variável de instância.

Os métodos setter e getter são coletivamente conhecidos como *métodos acessores*, ou simplesmente *acessores*.

Convenções de nomenclatura de acessores

Na declaração de métodos acessores para **BNRPerson**, você seguiu importantes convenções de nomenclatura do Objective-C. Os métodos getter fornecem o nome da variável de instância menos o sublinhado.

```
// Instance variable declarations
{
    float _heightInMeters;
    int _weightInKilos;
}

// Getter method declarations
- (float)heightInMeters;
- (int)weightInKilos;
```

Os métodos setter começam com **set** seguido pelo nome da variável de instância menos o sublinhado. Observe que a caixa de tipos do nome do setter se ajusta a fim de preservar o camel-casing. Portanto, a primeira letra do nome da variável de instância está em letra maiúscula no nome do setter.

```
// Setter method declarations - notice difference in casing!
- (void)setHeightInMeters:(float)h;
- (void)setWeightInKilos:(int)w;
```

Aprender essas convenções é importante. No próximo capítulo, você vai aprender um atalho onde o compilador cria os métodos acessores para você. O compilador vai nomear os acessores de acordo com essas convenções.

self

Dentro de qualquer método, você tem acesso à variável local implícita **self**. **self** é um ponteiro para o objeto que está executando o método. Ele é usado quando um objeto deseja enviar uma mensagem para si mesmo.

Por exemplo, muitos programadores de Objective-C jamais leem ou escrevem diretamente uma variável de instância. Eles até mesmo chamam acessores dentro de implementações de outros métodos na mesma classe.

Atualmente, sua implementação de **bodyMassIndex** acessa diretamente a variável de instância. No **BNRPerson.m**, atualize o **bodyMassIndex** para usar os métodos acessores em seu lugar:

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

Você também pode passar **self** como um argumento para permitir que outros objetos saibam onde está o objeto atual. Por exemplo, sua classe **BNRPerson** pode ter um método **addYourselfToArray**: que se assemelhe a este:

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

Aqui, você usa **self** para informar ao array onde reside a instância de **BNRPerson**. É literalmente o endereço da instância de **BNRPerson**.

Múltiplos arquivos

Observe que o seu projeto agora tem código executável em dois arquivos: **main.m** e **BNRPerson.m**. (**BNRPerson.h** é uma declaração de uma classe e não tem nenhum código executável nele.) Ao compilar o projeto, esses arquivos são compilados separadamente e depois vinculados. É comum que um projeto do mundo real consista em centenas de arquivos de código C e Objective-C.

Quando o Xcode compila seu projeto, ele compila cada um dos arquivos .m e .c no código da máquina. Em seguida, ele vincula todos os arquivos em qualquer biblioteca em um arquivo executável. Quais bibliotecas? Nesta seção do livro, todos os seus executáveis foram vinculados ao framework Foundation.

Há muitas bibliotecas de código reutilizável que podem ser baixadas da Internet. Muitas são gratuitas, outras não.

Prefixos de classe

O Objective-C não tem noção de namespace. Isso significa que se você escrever um programa com uma classe chamada **Person** nele, e vincular em uma biblioteca com o código de alguém que também declara uma classe **Person**, então, o compilador não será capaz de separar essas duas classes e ocorrerá um erro.

Para evitar colisões de nomes como essa, a Apple recomenda que você prefixe cada um dos seus nomes de classe com três ou mais letras, para tornar os seus nomes de classe mais exclusivos e menos propensos a colidir com o nome de classe de outra pessoa. Grande parte dos desenvolvedores usa as iniciais de suas empresas ou projetos. Neste livro, usamos o prefixo de classe **BNR**.

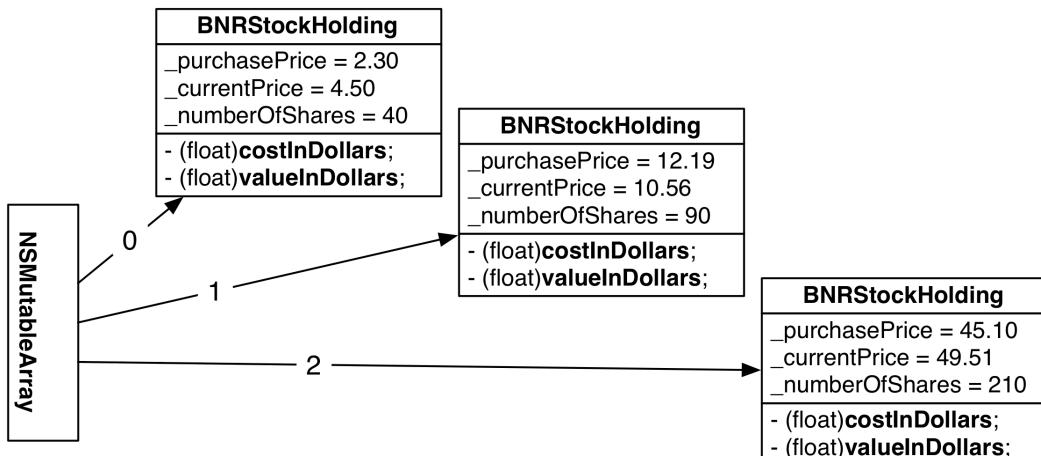
Desafio

Crie uma nova Foundation Command Line Tool chamada Stocks. Depois, crie uma classe chamada **BNRStockHolding** para representar as ações que você adquiriu. Ela será uma subclasse de **NSObject**. Para variáveis de instância, ela terá dois floats chamados `_purchasePrice` e `_currentSharePrice`, e um int chamado `_numberOfShares`. Use as propriedades para criar métodos acessores e variáveis de instância. Crie dois outros métodos de instância:

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

Em `main()`, preencha um array com três instâncias de **BNRStockHolding**. Depois, faça iteração pelo array, exibindo o valor de cada uma delas.

Figure 18.4 Um array de objetos **BNRStockHolding**



19

Propriedades

O Objective-C tem um atalho conveniente chamado *propriedades* que permite pular a declaração de variáveis de instância e a declaração e implementação de métodos acessores. O uso de propriedades simplifica o código da sua classe.

Declaração de propriedades

No BNRPerson.h, remova as declarações de variável de instância e método acessor e substitua-as por duas propriedades: `heightInMeters` e `weightInKilos`.

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

@interface BNRPerson : NSObject

// BNRPerson has two properties
@property (nonatomic) float heightInMeters;
@property (nonatomic) int weightInKilos;

{
    // BNRPerson has two instance variables
    float _heightInMeters;
    int _weightInKilos;
}

// BNRPerson has methods to read and set its instance variables
-(float)heightInMeters;
-(void)setHeightInMeters:(float)h;
-(int)weightInKilos;
-(void)setWeightInKilos:(int)w;

// BNRPerson has a method that calculates the Body Mass Index
- (float)bodyMassIndex;

@end
```

Uma declaração de propriedade começa com `@property` e inclui o tipo da propriedade e seu nome. Ignore o `(nonatomic)` por enquanto. Esse é um atributo de propriedade, o que discutiremos mais adiante neste capítulo.

Declarar propriedades deixa seu arquivo de cabeçalho curto e bom. No futuro, declarar propriedades fará você economizar algumas linhas de digitação. Mas espere, tem mais. Quando você declara uma propriedade, o compilador não apenas declara seus acessores para você, mas também os implementa com base na declaração da propriedade.

Isso significa que você não precisa mais das implementações de acessores que escreveu no `BNRPerson.m`. Abra o `BNRPerson.m` e exclua-as:

```
@implementation BNRPerson
- (float)heightInMeters
{
    return _heightInMeters;
}

- (void)setHeightInMeters:(float)h
{
    _heightInMeters = h;
}

- (int)weightInKilos
{
    return _weightInKilos;
}

- (void)setWeightInKilos:(int)w
{
    _weightInKilos = w;
}

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

@end
```

Compile e execute seu programa. Ele deve funcionar exatamente como antes. Com o uso de propriedades, você não alterou em nada essa classe. Você não fez nenhuma alteração no `main.m` porque `BNRPerson` ainda tem todos os mesmos métodos acessores com os mesmos nomes. Eles apenas estão escritos com uma sintaxe mais concisa (e estilisticamente melhor).

E quanto às variáveis de instância? O compilador criou variáveis de instância chamadas `_heightInMeters` e `_weightInKilos`. Entretanto, você não vê essas variáveis em seu código porque não há mais qualquer implementação de acessor explícita. Ao usar propriedades, você raramente precisa usar variáveis de instância de forma direta e pode confiar nos acessores criados pelo compilador.

Daqui para frente, você quase sempre utilizará propriedades ao criar uma classe. A Apple recomenda o uso de propriedades, e nós também.

Então por que você precisou aprender primeiro sobre variáveis de instância e métodos acessores? Existem algumas exceções, sobre as quais você aprenderá mais adiante, em que você mesmo precisa ajustar a declaração de propriedade ou implementar os métodos acessores. É muito mais fácil trabalhar com propriedades quando você entende o que elas fazem para você.

Atributos de propriedade

Uma declaração de propriedade tem um ou mais *atributos de propriedade*. Os atributos de propriedade fornecem ao compilador mais informações sobre como a propriedade deve se comportar. Os atributos de propriedade aparecem em uma lista delimitada por vírgulas, entre parênteses, após a anotação `@property`.

Na `BNRPerson`, as propriedades são declaradas como `nonatomic` (não atômicas).

```
@property (nonatomic) float heightInMeters;
@property (nonatomic) int weightInKilos;
```

As propriedades são `atomic` (atômicas) ou `nonatomic` (não atômicas). A diferença tem a ver com multithreading (multisegmentação), que é um tópico que está fora do escopo deste livro. Todas as propriedades que você vai declarar neste livro serão `nonatomic`.

Consideremos outro atributo de propriedade. Às vezes, uma classe precisa de uma propriedade “somente leitura” – uma propriedade cujo valor pode ser lido mas não alterado. Uma propriedade como essa deve ter um

método getter, mas nenhum método setter. Você pode instruir o compilador para que crie apenas um método getter através da inclusão de um valor `readonly` (somente leitura) na lista de atributos de propriedade.

```
@property (nonatomic, readonly) double circumferenceOfEarth;
```

Quando o compilador vir essa declaração, ele criará um método getter `circumferenceOfEarth`, mas não um método setter `setCircumferenceOfEarth:`.

As propriedades são `readonly` (somente leitura) ou `readwrite` (leitura e gravação).

```
@property (nonatomic, readwrite) double humanPopulation;
```

Com base nessa declaração, o compilador criará um método getter `humanPopulation` e um método setter `setHumanPopulation:`.

Ambas as propriedades `BNRPerson` são `readwrite`. Entretanto, você não precisou incluir o atributo `readwrite` porque `readwrite` é o valor padrão. Valores padrão são opcionais em declarações.

Infelizmente, `atomic` é o valor padrão do atributo atômico/não atômico, então você precisa incluir o valor `nonatomic` em todas as suas declarações de propriedade.

Outro atributo que você verá em breve é o `copy` (copiar). Em termos práticos, sempre que você declarar uma propriedade que aponte para uma `NSString` ou uma `NSArray`, você deve incluir o atributo `copy`. Você descobrirá por que no Chapter 34.

Existem outros atributos e valores de propriedade sobre os quais você aprenderá no decorrer do livro.

Notação de ponto

Os programadores de Objective-C chamam muito os métodos acessores. Quando as propriedades foram introduzidas, a Apple também introduziu uma *notação de ponto* como um atalho para chamar tais acessores.

Muitos programadores de Objective-C preferem a notação de ponto por ser mais fácil de digitar. No `main.m`, experimente a notação de ponto:

```
int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // Create an instance of BNRPerson
        BNRPerson *mikey = [[BNRPerson alloc] init];

        // Give the instance variables interesting values using setters
        [mikey setWeightInKilos:96];
        [mikey setHeightInMeters:1.8];
        mikey.weightInKilos = 96;
        mikey.heightInMeters = 1.8;

        // Log the instance variables using the getters
        float height = [mikey heightInMeters];
        int weight = [mikey weightInKilos];
        float height = mikey.heightInMeters;
        int weight = mikey.weightInKilos;
        NSLog(@"mikey is %.2f meters tall and weighs %d kilos", height, weight);

        // Log some values using custom methods
        float bmi = [mikey bodyMassIndex];
        NSLog(@"mikey has a BMI of %.2f", bmi);
    }
    return 0;
}
```

Essa notação se parece exatamente com a notação usada para acessar os membros de uma `struct`. É fundamental se lembrar, entretanto, que ao usar a notação de ponto com um objeto, uma mensagem está sendo enviada.

Estas duas linhas fazem exatamente a mesma coisa:

```
mikey.weightInKilos = 96;  
[mikey setWeightInKilos:96];
```

E estas duas linhas fazem exatamente a mesma coisa:

```
float w = mikey.weightInKilos;  
float w = [mikey weightInKilos];
```

Perceba que `mikey.weightInKilos` envia uma de duas mensagens possíveis, dependendo do contexto em que está sendo usado. Isto é, ele chama ou o método getter (`weightInKilos`) ou o método setter (`setWeightInKilos:`) dependendo se estiver sendo usado para obter ou definir o `_weightInKilos` de `mikey`.

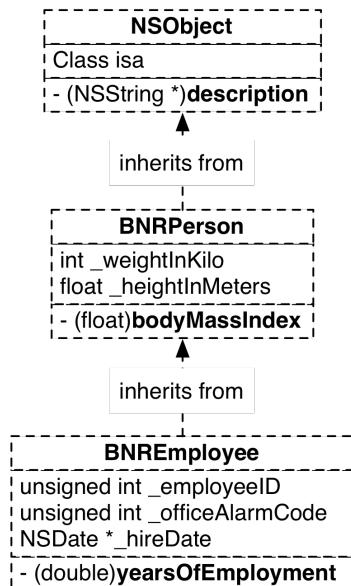
20

Herança

Quando você criou a classe **BNRPerson**, a declarou como subclasse de **NSObject**. Isso significa que toda instância de **BNRPerson** terá os métodos e as variáveis de instância definidos em **NSObject**, bem como os métodos e as variáveis de instância definidos em **BNRPerson**. Dizemos que **BNRPerson** *herda* as variáveis de instância e os métodos da **NSObject**.

Neste capítulo, você criará uma nova classe chamada **BNREmployee**. **BNREmployee** será uma subclasse de **BNRPerson**.

Figure 20.1 **BNREmployee** herda de **BNRPerson**



Faz sentido, não é mesmo? Os funcionários são pessoas. Eles têm alturas e pesos. Mas nem todas as pessoas são funcionários; os funcionários podem ter características específicas. Sua classe **BNREmployee** adiciona duas características específicas de **BNREmployee** – uma ID de funcionário e uma data de contratação.

Abra o projeto **BMITime** e crie um novo arquivo: uma Objective-C class. Nomeie a classe **BNREmployee** e deixe sua superclasse como **NSObject** por enquanto.

Abra o BNREmployee.h. Importe o BNRPerson.h e altere a superclasse de **NSObject** para **BNRPerson**:

```
#import "BNRPerson.h"

@interface BNREmployee : BNRPerson

@end
```

BNREmployee é agora uma subclasse de **BNRPerson**.

A função **main** do BMITime precisará acessar a ID de funcionário e a data de contratação. Todo funcionário também tem um código de alarme para entrar no escritório. Portanto, você precisa de três novas propriedades e deve adicioná-las no BNREmployee.h. Também declare um método que calculará os anos de empresa com base na data de contratação do funcionário.

```
#import "BNRPerson.h"

@interface BNREmployee : BNRPerson

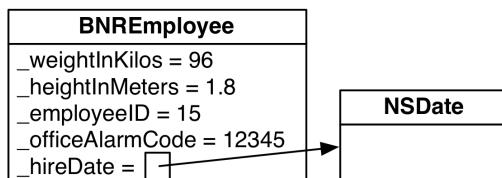
@property (nonatomic) unsigned int employeeID;
@property (nonatomic) unsigned int officeAlarmCode;
@property (nonatomic) NSDate *hireDate;
- (double)yearsOfEmployment;

@end
```

A propriedade **hireDate** é a primeira propriedade que você declarou que aponta para outro objeto. Quando uma propriedade aponta para um objeto, há implicações de gerenciamento de memória sobre as quais aprenderá nos capítulos posteriores.

Por enquanto, reconheça que você declarou uma propriedade chamada **hireDate** que é um ponteiro para uma **NSDate**. Você informa ao compilador que esta propriedade deve ser não atômica, como todas as outras propriedades.

Figure 20.2 **BNREmployee** tem um ponteiro para uma **NSDate**



E, como suas propriedades primitivas de tipo, por padrão você obtém uma variável de instância. Essa variável é chamada `_hireDate` e é um ponteiro para uma **NSDate**. O compilador também sintetiza dois métodos acessores:

```
- (void)setHireDate:(NSDate *)d;
- (NSDate *)hireDate;
```

No BNREmployee.m, implemente o método **yearsOfEmployment**:

```
@implementation BNREmployee
- (double)yearsOfEmployment
{
    // Do I have a non-nil hireDate?
    if (self.hireDate) {
        // NSTimeInterval is the same as double
        NSDate *now = [NSDate date];
        NSTimeInterval secs = [now timeIntervalSinceDate:self.hireDate];
        return secs / 31557600.0; // Seconds per year
    } else {
        return 0;
    }
}
@end
```

Para experimentar a classe **BNREmployee**, abra o `main.m` e realize duas alterações: importe o `BNREmployee.h` e crie uma instância de **BNREmployee** em vez de uma **BNRPerson**. Deixe a variável `person` declarada como um ponteiro para uma **BNRPerson** por enquanto:

```
#import <Foundation/Foundation.h>
#import "BNRPerson.h"
#import "BNREmployee.h"

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

        // Create an instance of BNREmployee
        BNRPerson *mikey = [[BNREmployee alloc] init];

        // Give properties interesting values using setter methods
        mikey.weightInKilos = 96;
        mikey.heightInMeters = 1.8;

        // Log some properties using getter methods
        NSLog(@"mikey has a weight of %d", mikey.weightInKilos);
        NSLog(@"mikey has a height of %f", mikey.heightInMeters);

        // Log the body mass index
        float bmi = [mikey bodyMassIndex];
        NSLog(@"mikey has a BMI of %f", bmi);

    }
    return 0;
}
```

Você acha que isso causará um problema? Compile e execute o programa para ver.

O programa funciona bem e nada no resultado foi alterado. *Um funcionário é uma pessoa* – ele pode fazer tudo o que uma pessoa faz. Uma instância de **BNREmployee** pode responder a métodos de **BNRPerson** (como `setWeightInKilos:`). Você pode usar uma instância de **BNREmployee** em qualquer lugar em que o programa espera uma instância de **BNRPerson**. Uma instância de uma subclasse pode substituir uma instância da superclasse sem problemas, já que ela herda tudo na superclasse.

Observe também que você não precisa importar uma `BNRPerson.h`. O compilador localizará a instrução `#import "BNRPerson.h"` no arquivo `BNREmployee.h`, portanto, inclui-la aqui seria redundante.

Agora realize as seguintes alterações no `main.m` para fazer uso total da classe **BNREmployee**. Dê à variável `mikey` uma ID de funcionário e defina a data de contratação para a data atual.

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

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

        // Create an instance of BNREmployee
        BNREmployee *mikey = [[BNREmployee alloc] init];

        // Give the instance variables interesting values using setter methods
        mikey.weightInKilos = 96;
        mikey.heightInMeters = 1.8;
        mikey.employeeID = 12;
        mikey.hireDate = [NSDate dateWithNaturalLanguageString:@"Aug 2nd, 2010"];

        // Log the instance variables using the getters
        float height = mikey.heightInMeters;
        int weight = mikey.weightInKilos;
        NSLog(@"%@", mikey is %.2f meters tall and weighs %d kilos", height, weight);
        NSLog(@"%@", Employee %u hired on %@", mikey.employeeID, mikey.hireDate);

        // Log the body mass index using the bodyMassIndex method
        float bmi = [mikey bodyMassIndex];
        double years = [mikey yearsOfEmployment];
        NSLog(@"%@", BMI of %.2f, has worked with us for %.2f years", bmi, years);
    }
    return 0;
}
```

Compile e execute o programa e verifique seu novo resultado.

Sobrescrevendo métodos

Normalmente, uma subclasse precisa fazer algo diferente de sua superclasse. Digamos, por exemplo, que, diferentemente das pessoas em geral, os funcionários sempre têm um IMC de 19. Neste caso, você *sobrescreveria* o método **bodyMassIndex** na **BNREmployee**.

Você sobrescreve um método herdado ao escrever uma nova implementação.

No BNREmployee.m, sobrescreva **bodyMassIndex**:

```
#import "BNREmployee.h"
@implementation BNREmployee

- (double)yearsOfEmployment
{
    // Do I have a non-nil hireDate?
    if (self.hireDate) {
        // NSTimeInterval is the same as double
        NSTimeInterval secs = [self.hireDate timeIntervalSinceNow];
        return secs / 31557600.0; // Seconds per year
    } else {
        return 0;
    }
}

- (float)bodyMassIndex
{
    return 19.0;
}

@end
```

Já que **BNREmployee** herda de **BNRPerson**, todos já sabem que as instâncias de **BNREmployee** responderão a uma mensagem **bodyMassIndex**. Não há necessidade de anunciar isso novamente, portanto, você não a declara no BNREmployee.h.

Isso também significa que, quando você sobrescreve um método, você só pode alterar sua implementação. Você não pode alterar como ele é declarado; o nome do método, o tipo de retorno e os tipos de argumento devem permanecer iguais.

Compile e execute o programa. Confirme que a implementação de **BNREmployee** do **bodyMassIndex** é aquela que é executada – e não a implementação de **BNRPerson**.

super

Ao sobrescrever um método, uma subclasse pode compilar na implementação de sua superclasse em vez de substitui-la indiscriminadamente. E se você decidiu que os funcionários tiveram uma redução de 10% de seu IMC conforme calculado na implementação da **BNRPerson**? Seria conveniente chamar a versão da **BNRPerson** do **bodyMassIndex** e, então, multiplicar o resultado por 0,9. Para fazer isso, você usa a diretiva **super**. Tente fazer isso no BNREmployee.m:

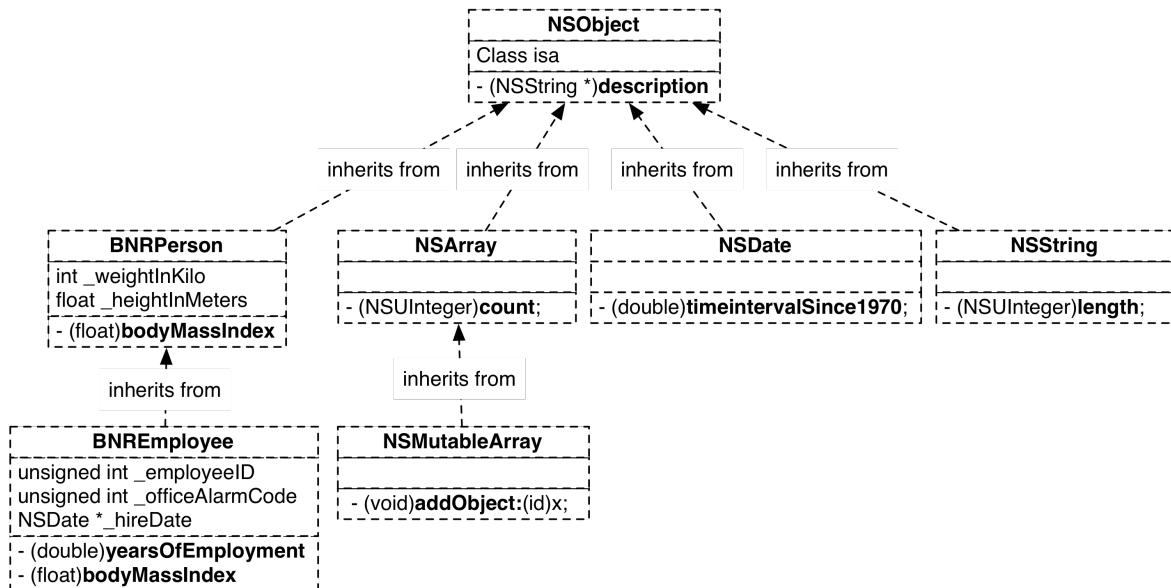
```
- (float)bodyMassIndex
{
    return 19.0;
    float normalBMI = [super bodyMassIndex];
    return normalBMI * 0.9;
}
```

Compile e execute o programa.

Hierarquia de herança

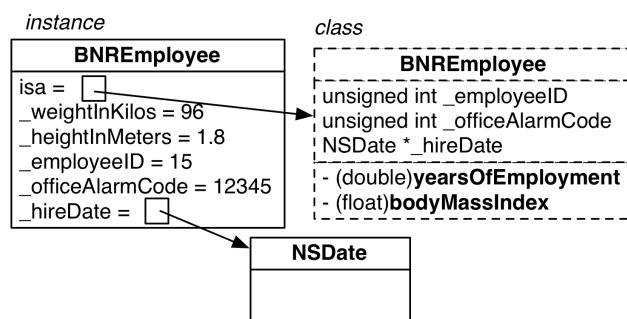
Todos os objetos herdam (tanto direta como indiretamente) de **NSObject**:

Figure 20.3 Diagrama de herança de algumas classes que você conhece



NSObject tem muitos métodos, mas apenas uma variável de instância: o ponteiro *isa*. Todo ponteiro *isa* do objeto aponta para a classe que o criou. (Entendeu? Quando você tem uma instância de **BNRPerson**, esse objeto “é uma” **BNRPerson**. Quando você tem uma instância de **NSString**, esse objeto “é uma” **NSString**.)

Figure 20.4 Todo objeto sabe qual classe o criou



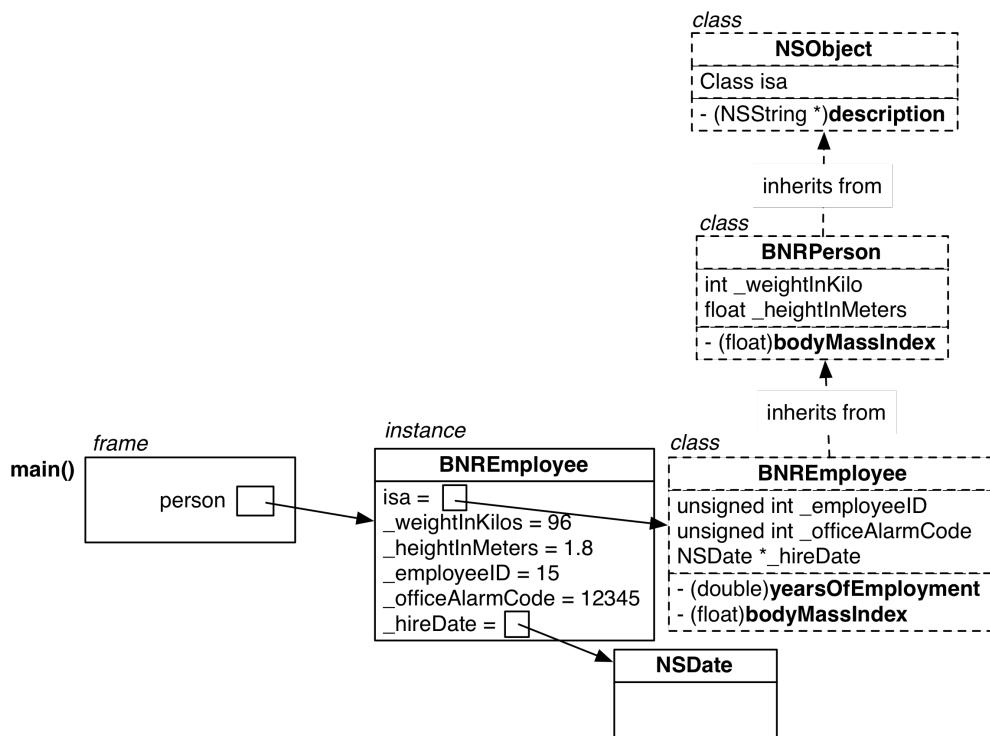
Ao enviar uma mensagem para um objeto, você inicia uma pesquisa por um método para esse nome. A pesquisa segue o ponteiro *isa* do objeto para começar a procurar pelo método na classe do objeto. Se não houver um método com esse nome lá, então ele está na superclasse. A busca é interrompida quando o método é encontrado ou quando o topo da hierarquia (**NSObject**) é alcançado.

Digamos que você envia a mensagem **fido** a um objeto. Para responder a essa mensagem, o objeto usa o ponteiro *isa* para encontrar sua classe e pergunta: “Você declara um método de instância chamado **fido**?”.

Se a classe tiver um método chamado **fido**, ele será executado. Se a classe não tiver um método **fido**, ela perguntará à sua superclasse: “Você declara um método de instância chamado **fido**?”.

E, até a hierarquia, ela continua a busca pela implementação de um método chamado **fido**. A busca é interrompida quando o método é encontrado ou quando o topo da hierarquia é alcançado.

Figure 20.5 Diagrama de objetos do BMITime



No topo da hierarquia, **NSObject** diz: “Não, não tenho nenhum método **fido**.”. Neste ponto, você obtém uma mensagem de erro que diz algo como: `- [BNREmployee fido]: unrecognized selector sent to instance 0x100106e102` (seletor não reconhecido enviado à instância `0x100106e102`). Isso pode ser traduzido como: “Não há método de instância desse nome definido em algum lugar nessa hierarquia de herança do objeto.”.

A primeira implementação encontrada é aquela que é executada. **BNREmployee** e **BNRPerson** têm implementações de **bodyMassIndex**, mas se uma **BNREmployee** enviar a mensagem **bodyMassIndex**, a implementação na **BNREmployee** será encontrada primeiro e executada. Essa busca termina antes de alcançar a classe **BNRPerson**.

Quando você usa a diretiva `super`, você está enviando uma mensagem para o objeto atual dizendo: “execute um método com esse nome, mas inicie a pesquisa por sua implementação na sua superclasse”.

description e %@

Em chamadas para `NSLog()`, você tem usado o token `%@` para fazer um objeto se descrever. O token `%@` envia uma mensagem **description** para o objeto apontado pela variável correspondente.

O método **description** retorna uma string que é uma descrição útil de uma instância da classe. Trata-se de um método **NSObject**, portanto, todo objeto o implementa. A implementação padrão de **NSObject** retorna o endereço do objeto na memória como uma string.

No entanto, um endereço de memória não é geralmente a maneira mais útil de descrever uma instância. Uma classe pode sobrescrever o **description**. Por exemplo, **NSDate** sobrescreve **description** para retornar a data/hora que a instância armazena. **NSString** sobrescreve **description** para retornar a string a si própria.

No `BNREmployee.m`, sobrescreva **description** para retornar uma string que descreva uma instância de **BNREmployee**:

```
@implementation BNREmployee
...
- (NSString *)description
{
    return [NSString stringWithFormat:@"<Employee %d>", self.employeeID];
}
@end
```

Modifique o `main.m` para usar a implementação de `description` da `BNREmployee`.

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

        // Create an instance of BNREmployee
        BNREmployee *mikey = [[BNREmployee alloc] init];

        ...

        NSLog(@"%@", mikey);
        NSDate *date = mikey.hireDate;
        NSLog(@"%@", hired on %@", mikey, date);

        // Log some values using custom methods
        float bmi = [mikey bodyMassIndex];
        double years = [mikey yearsOfEmployment];
        NSLog(@"BMI of %.2f, has worked with us for %.2f years", bmi, years);
    }
    return 0;
}
```

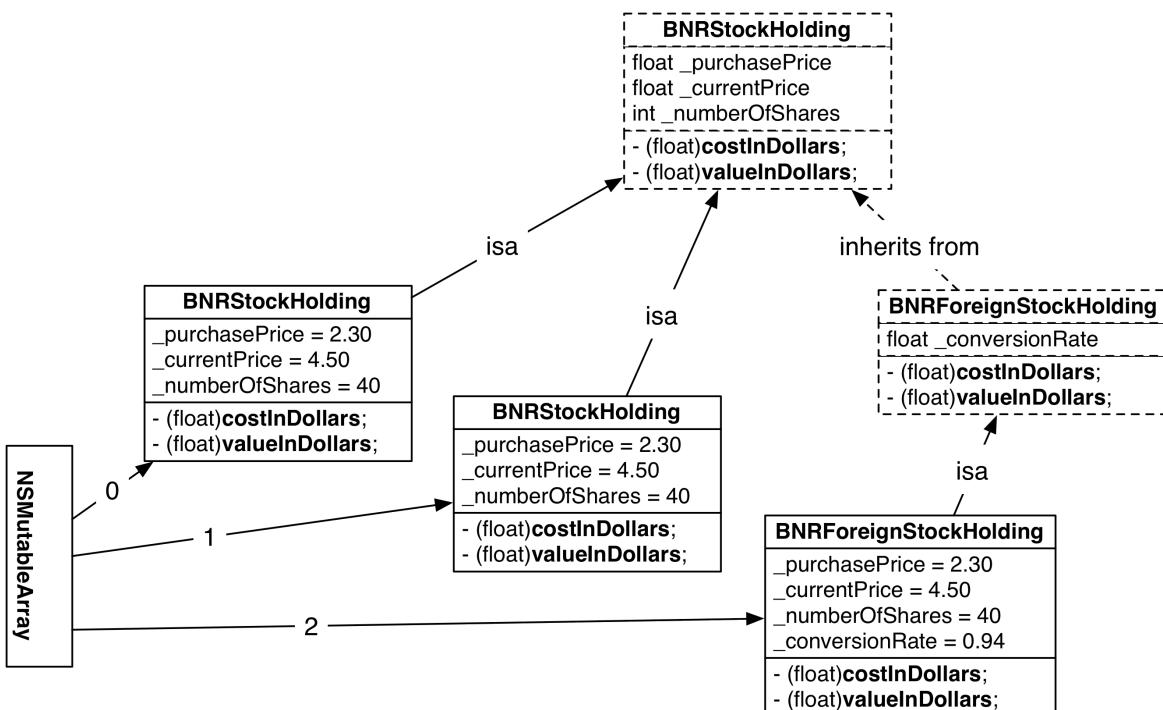
Desafio

Este desafio baseia-se no desafio do capítulo anterior.

Crie uma nova subclasse de `BNRStockHolding` chamada `BNRForeignStockHolding`. Dê à `BNRForeignStockHolding` uma propriedade adicional: `conversionRate`, que será do tipo `float`. (O preço local será multiplicado pela taxa de conversão a fim de obter um preço em US\$. Considera que `purchasePrice` e `currentPrice` estão na moeda local.) Sobrescreva `costInDollars` e `valueInDollars` para fazer o cálculo certo.

Em `main()`, adicione algumas instâncias de `BNRForeignStockHolding` em seu array.

Figure 20.6 Objetos `BNRStockHolding` e `BNRForeignStockHolding`

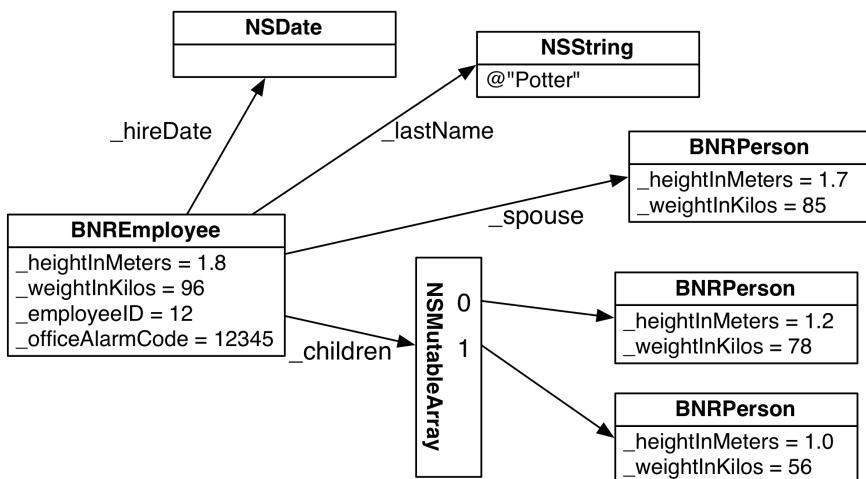


Propriedades e variáveis de instância de objeto

Até agora, as variáveis de instância de seus objetos foram em sua maioria tipos simples em C, como `int` ou `float`. É muito mais comum que as variáveis de instância sejam ponteiros para outros objetos, como `_hireDate`. Uma variável de instância de objeto aponta para outro objeto e descreve um relacionamento entre os dois objetos. Geralmente, as variáveis de instância de objeto classificam-se em três categorias:

- *Atributos objeto-tipo*: um ponteiro para um objeto simples e que se pareça com um valor, como uma `NSString` ou uma `NSDate`. Por exemplo, o sobrenome de um funcionário seria armazenado em uma `NSString`. Portanto, uma instância de `BNREmployee` teria uma variável de instância que seria um ponteiro para uma instância de `NSString`. Recomendamos que você sempre os declare como uma propriedade com uma variável de instância implícita; normalmente, você não precisará criar acessores de forma explícita.
- *Relacionamentos to-one (para um)*: um ponteiro para um único objeto complexo. Por exemplo, um funcionário poderia ter uma esposa. Portanto, uma instância de `BNREmployee` teria uma variável de instância que seria um ponteiro para uma instância de `BNRPerson`. Novamente, recomendaremos que você sempre os declare como uma propriedade com uma variável de instância implícita; normalmente, você não precisará criar acessores de forma explícita.
- *Relacionamentos to-many (para vários)*: um ponteiro para uma instância de uma classe de coleção, como uma `NSMutableArray`. (Você verá outros exemplos de coleções no Chapter 24.) Por exemplo, um funcionário poderia ter filhos. Nesse caso, a instância de `BNREmployee` teria uma variável de instância que seria um ponteiro para uma instância de `NSMutableArray`. Uma `NSMutableArray` conteria uma lista de ponteiros para um ou mais objetos de `BNRPerson`. Relacionamentos to-many (para vários) são mais complicados que atributos ou relacionamentos to-one (para um). Você frequentemente acabará criando, de forma explícita, variáveis de instância, acessores e métodos para a adição ou remoção de objetos do relacionamento.

Figure 21.1 Uma `BNREmployee` com variáveis de instância de objeto



Observe que, assim como em outros diagramas, os ponteiros são representados por setas. Além disso, esses ponteiros são nomeados. Assim, uma classe **BNREmployee** teria três novas variáveis de instância: `_lastName`, `_spouse` e `_children`. A declaração das propriedades de **BNREmployee** poderá ser semelhante à seguinte:

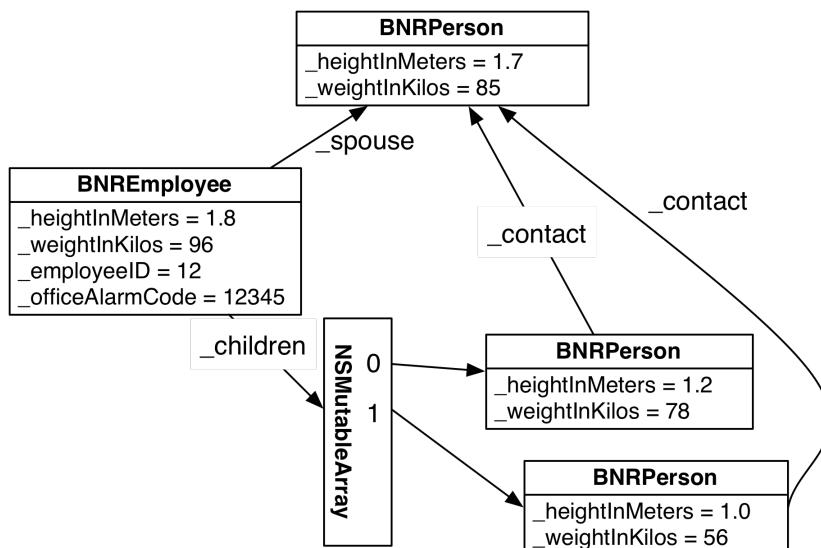
```
@interface BNREmployee : BNRPerson
@property (nonatomic) unsigned int employeeID;
@property (nonatomic) unsigned int officeAlarmCode;
@property (nonatomic) NSDate *hireDate;
@property (nonatomic) NSString *lastName;
@property (nonatomic) BNRPerson *spouse;
@property (nonatomic) NSMutableArray *children;
```

Com exceção de `employeeID` e `officeAlarmCode`, todas essas propriedades são ponteiros. Por exemplo, a variável `spouse` (esposa) é um ponteiro para outro objeto que reside no heap. O ponteiro chamado `spouse` está dentro do objeto de **BNREmployee**, mas o objeto de **BNRPerson** para o qual `spouse` aponta não está. Objetos não residem dentro de outros objetos. O objeto “employee” contém sua “employee ID” (a variável e o próprio valor), mas ele sabe apenas onde “spouse” reside na memória.

Há dois efeitos importantes causados aos objetos que apontam para outros objetos – em vez de contê-los:

- Um objeto pode assumir diversas funções. Por exemplo, é comum que a esposa do funcionário também seja listada como o contato dos filhos em casos de emergência:

Figure 21.2 Um objeto, diversas funções



- Você acaba ficando com muitos objetos diferentes usando a memória de seu programa. Você precisa que os objetos que estão sendo usados permaneçam, mas deseja que os desnecessários sejam desalocados (que a memória deles retorne ao heap), de modo que a memória possa ser reutilizada.

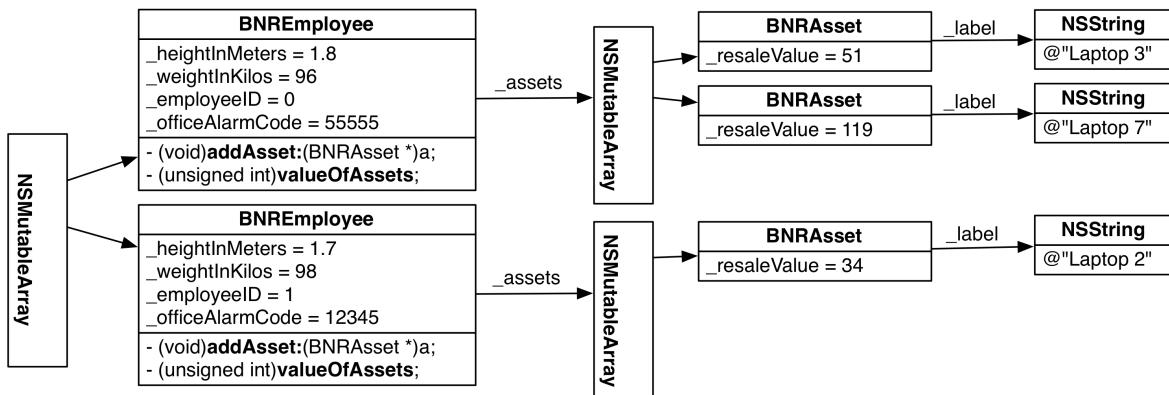
Propriedade de objetos e ARC

Para lidar com esses problemas, há a *propriedade de objeto*. Quando um objeto tem uma variável de instância de objeto, considera-se que o objeto com o ponteiro *possui* o objeto que está sendo apontado.

Por outro lado, por causa da ARC (discutida brevemente no Chapter 15), um objeto sabe quantos *proprietários* ele tem no momento. Por exemplo, no diagrama acima, a instância de **BNRPerson** tem três proprietários: o objeto de **BNREmployee** e dois objetos de **Child**. Quando um objeto não tem nenhum proprietário, ele considera que ninguém mais precisa dele e se desaloca sozinho. Antes da introdução da ARC no Xcode 4.2, lidávamos manualmente com a propriedade e gastávamos muito tempo e esforço fazendo isso. (Na seção final do Chapter 23, há mais informações sobre a contagem de referência manual e como ela funcionava. No entanto, todo código contido neste livro assume que você esteja usando a ARC.)

Vamos expandir o projeto do BMITime para ver como a propriedade funciona na prática. É comum que uma empresa controle quais ativos estão sendo distribuídos e a quais funcionários. Você vai criar uma classe **BNRAsset**, e cada **BNREmployee** (funcionário) terá um array que conterá seus ativos.

Figure 21.3 Funcionários e ativos



Esse relacionamento é frequentemente chamado de “pai-filho”: O pai (uma instância de **BNREmployee**) tem uma coleção de filhos (uma **NSMutableArray** de objetos de **BNRAsset**).

Criação da classe BNRAsset

Crie um novo arquivo: uma subclasse de **NSObject** em Objective-C. Nomeie-o como **BNRAsset**. Abra o **BNRAsset.h** e declare duas propriedades:

```

#import <Foundation/Foundation.h>

@interface BNRAsset : NSObject

@property (nonatomic, copy) NSString *label;
@property (nonatomic) unsigned int resaleValue;

@end

```

Lembre-se de que, quando um objeto não tem nenhum proprietário, ele é desalocado. Há um método **NSObject** chamado **dealloc**. Se uma classe sobreescrer **dealloc**, então esse método será executado quando uma instância da classe for desalocada. Você vai sobreescrer **dealloc** em **BNRAsset**, de modo que possa ver quando as instâncias de **BNRAsset** estiverem sendo desalocadas.

Para deixar claro qual instância específica de **BNRAsset** está sendo desalocada, você também vai sobreescrer **description** para retornar uma string que inclua as variáveis **label** e **resaleValue** da instância.

Abra o **BNRAsset.m**. Sobreescreve **description** e **dealloc**.

```

#import "BNRAsset.h"

@implementation BNRAsset

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

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

@end

```

Tente compilar o que você tem até agora para ver se cometeu algum erro na digitação. Você pode compilar seu programa sem executá-lo; basta usar o atalho de teclado Command-B. Isso é muito útil para testar seu código sem executar o programa ou quando você sabe que o programa ainda não está pronto para ser executado. Além disso, é sempre bom compilar após fazer alterações, pois, se você tiver inserido um erro de sintaxe, pode localizá-lo e corrigi-lo imediatamente. Se você esperar, não terá tanta certeza sobre quais alterações são responsáveis por seu “novo” bug.

Adição de um relacionamento to-many (para vários) à classe BNREmployee

Agora, você adicionará um relacionamento para vários à classe **BNREmployee**. Lembre-se que um relacionamento para vários inclui um objeto de coleção (como um array) e os objetos contidos na coleção. Há duas outras questões importantes a saber sobre coleções e propriedade:

- Quando um objeto é adicionado à coleção, esta estabelece um ponteiro para o objeto, e o objeto passa a ter um proprietário.
- Quando um objeto é removido de uma coleção, esta se desfaz de seu ponteiro para o objeto, e o objeto perde um proprietário.

Para configurar o relacionamento para vários na classe **BNREmployee**, você precisará de uma nova variável de instância para conter um ponteiro para um array de ativos mutável. Você também precisará de alguns métodos. Abra o **BNREmployee.h** e adicione-os:

```
#import "BNRPerson.h"
@class BNRAsset;

@interface BNREmployee : BNRPerson
{
    NSMutableArray *_assets;
}

@property (nonatomic) unsigned int employeeID;
@property (nonatomic) unsigned int officeAlarmCode;
@property (nonatomic) NSDate *hireDate;
@property (nonatomic, copy) NSArray *assets;
- (double)yearsOfEmployment;
- (void)addAsset:(BNRAsset *)a;
- (unsigned int)valueOfAssets;

@end
```

Observe a linha que contém: `@class BNRAsset;`. Como o compilador está lendo esse arquivo, ele encontrará o nome da classe **BNRAsset**. Se não souber nada sobre a classe, ela gerará um erro. A linha `@class BNRAsset;` informa ao compilador que: “Há uma classe chamada **BNRAsset**. Não se preocupe aovê-la neste arquivo. Isso é tudo o que você precisa saber por enquanto.”.

Usar `@class` em vez de `#import` fornece menos informações ao compilador, mas faz com que o processamento desse arquivo específico seja mais rápido. Você pode usar `@class` com o **BNREmployee.h** e com outros arquivos de cabeçalho, pois o compilador não precisa saber muito para processar um arquivo de declarações.

A propriedade tem o tipo **NSArray**, que diz às outras classes: “Se você pedir meus ativos, receberá algo que não é mutável.”. Entretanto, em segundo plano, o array de ativos é na verdade uma instância de **NSMutableArray** para que você possa adicionar e remover itens no **BNREmployee.m**. É por isso que você está declarando uma propriedade e uma variável de instância: Nesse caso, o tipo da propriedade e o tipo da variável de instância não são os mesmos.

Agora, voltemos a atenção para o **BNREmployee.m**. Com um relacionamento para vários, você precisa criar o objeto de coleção (nesse caso, um array) antes de colocar algo nele. Você pode fazer isso quando o objeto original (um funcionário) é criado inicialmente ou pode aguardar até que a coleção seja necessária. Vamos aguardar.

```
#import "BNREmployee.h"
#import "BNRAsset.h"
```

```

@implementation BNREmployee

// Accessors for assets properties
- (void)setAssets:(NSArray *)a
{
    _assets = [a mutableCopy];
}

- (NSArray *)assets
{
    return [_assets copy];
}

- (void)addAsset:(BNRAsset *)a
{
    // Is assets nil?
    if (!_assets) {

        // Create the array
        _assets = [[NSMutableArray alloc] init];
    }
    [_assets addObject:a];
}

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

- (double)yearsOfEmployment
{
    ...
}

```

Para processar o arquivo `BNREmployee.m`, o compilador precisa saber muita coisa sobre a classe `BNRAsset`. Portanto, você importou o `BNRAsset.h` em vez de usar `@class`.

Para controlar a desalocação das instâncias de `BNREmployee`, modifique a implementação de `description` e implemente `dealloc` no `BNREmployee.m`.

```

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

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

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

@end

```

Compile o projeto para ver se cometeu algum erro.

Agora, será necessário criar alguns ativos e atribuí-los aos funcionários. Substitua o conteúdo de `main.m`:

```
#import <Foundation/Foundation.h>
#import "BNREmployee.h"
#import "BNRAsset.h"

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

        // Create an array of BNREmployee objects
        NSMutableArray *employees = [[NSMutableArray alloc] init];

        for (int i = 0; i < 10; i++) {
            // Create an instance of BNREmployee
            BNREmployee *mikey = [[BNREmployee alloc] init];

            // Give the instance variables interesting values
            mikey.weightInKilos = 90 + i;
            mikey.heightInMeters = 1.8 - i/10.0;
            mikey.employeeID = i;

            // Put the employee in the employees array
            [employees addObject:mikey];
        }

        // Create 10 assets
        for (int i = 0; i < 10; i++) {
            // Create an asset
            BNRAsset *asset = [[BNRAsset alloc] init];

            // Give it an interesting label
            NSString *currentLabel = [NSString stringWithFormat:@"Laptop %d", i];
            asset.label = currentLabel;
            asset.resaleValue = 350 + i * 17;

            // Get a random number between 0 and 9 inclusive
            NSUInteger randomIndex = random() % [employees count];

            // Find that employee
            BNREmployee *randomEmployee = [employees objectAtIndexAtIndex:randomIndex];

            // Assign the asset to the employee
            [randomEmployee addAsset:asset];
        }

        NSLog(@"%@", employees);

        NSLog(@"Giving up ownership of one employee");

        [employees removeObjectAtIndex:5];

        NSLog(@"Giving up ownership of arrays");

        employees = nil;
    }
    return 0;
}
```

Compile e execute o programa. Você verá algo semelhante ao seguinte:

```
Employees: (
    "<Employee 0: $0 in assets>",
    "<Employee 1: $153 in assets>",
    "<Employee 2: $119 in assets>",
    "<Employee 3: $68 in assets>",
    "<Employee 4: $0 in assets>",
    "<Employee 5: $136 in assets>",
    "<Employee 6: $119 in assets>",
    "<Employee 7: $34 in assets>",
    "<Employee 8: $0 in assets>",
    "<Employee 9: $136 in assets>"
)
```

```

Giving up ownership of one employee
deallocating <Employee 5: $136 in assets>
deallocating <Laptop 3: $51 >
deallocating <Laptop 5: $85 >
Giving up ownership of arrays
deallocating <Employee 0: $0 in assets>
deallocating <Employee 1: $153 in assets>
deallocating <Laptop 9: $153 >
...
deallocating <Employee 9: $136 in assets>
deallocating <Laptop 8: $136 >

```

Quando o Funcionário 5 é removido do array, ele é desalocado porque não tem nenhum proprietário. Depois, seus ativos são desalocados porque não têm nenhum proprietário. (E você terá de confiar em nós sobre isso: os rótulos (instâncias de **NSString**) dos ativos desalocados também são desalocados, pois não têm nenhum proprietário.)

Ao configurar `employees` como `nil`, o array passa a não ter mais um proprietário. Então, ele é desalocado, o que estabelece uma reação em cadeia ainda maior de desalocação e limpeza de memória, quando, de repente, nenhum dos funcionários tem um proprietário.

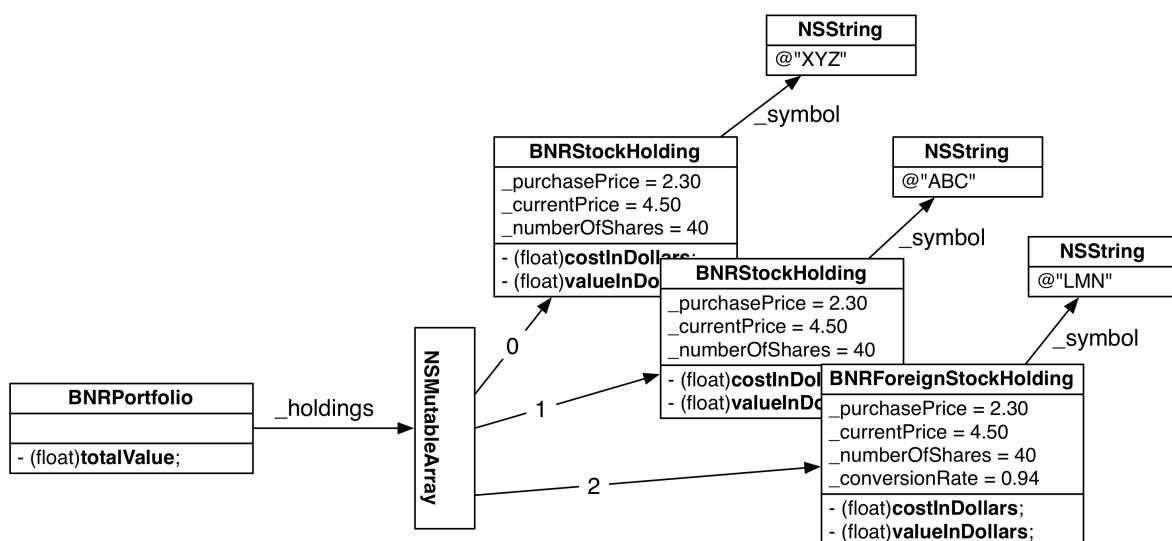
Organizado, não é mesmo? Assim que os objetos tornam-se desnecessários, eles vão sendo desalocados. Isso é excelente. Quando os objetos desnecessários não são desalocados, dizemos que há um *vazamento de memória*. Geralmente, um vazamento de memória faz com que cada vez mais objetos persistam desnecessariamente ao longo do tempo, o que fará com que seu aplicativo fique com pouca memória.

Desafio: carteira de ações

Usando a classe **BNRStockHolding** do desafio anterior, elabore uma ferramenta que crie uma instância de uma classe **BNRPortfolio** e inclua ações nela. Uma classe "portfolio" pode dizer a você qual é seu valor atual.

Além disso, adicione uma propriedade `symbol` a **BNRStockHolding** que retenha o símbolo ticker de ações como uma **NSString**.

Figure 21.4 Criação de uma classe **BNRPortfolio**



Desafio: remoção de ativos

Sua classe **BNREmployee** tem um método `addAsset:`. Dê a ela um método `removeAsset:` que funcione e teste-o em `main()`.

22

Extensões de classe

Até agora, você declarou todas as suas propriedades, variáveis de instância e métodos no arquivo de cabeçalho da classe. O cabeçalho é onde a classe anuncia suas propriedades e métodos de modo que outros objetos saberão como interagir com ela.

No entanto, nem toda propriedade ou método deve ser anunciado em um cabeçalho da classe. Algumas propriedades ou métodos podem apenas ser destinados ao uso pela classe ou instâncias da classe. Tais detalhes internos são mais bem declarados em uma *extensão de classe*. Uma extensão de classe é um conjunto de declarações que são privadas. Apenas a classe ou instâncias da classe são capazes de usar as propriedades, variáveis de instância ou métodos declarados em uma extensão de classe.

Por exemplo, a propriedade `officeAlarmCode` de `BNREmployee` deve ser privada. O objeto “employee” precisa conseguir acessar seu código de alarme, enquanto os objetos “non-employee” não precisam ter acesso ao código de alarme e não devem tê-lo. Você pode fazer isso acontecer ao mover a declaração de `officeAlarmCode` do cabeçalho (`BNREmployee.h`) para uma extensão de classe.

Normalmente, as extensões de classe são adicionadas ao arquivo de implementação de classe em cima do bloco `@implementation` onde os métodos são implementados. No `BNREmployee.m`, crie uma extensão de classe. Em seguida, declare a propriedade `officeAlarmCode` ali:

```
#import "BNREmployee.h"

// A class extension
@interface BNREmployee ()

@property (nonatomic) unsigned int officeAlarmCode;

@end

@implementation BNREmployee
...
```

Uma extensão de classe começa com `@interface` e termina com `@end`. Na verdade, uma extensão parece muito com um cabeçalho, e ambos são conhecidos como “interfaces”. No entanto, em uma extensão, em vez dos dois pontos e do nome da superclasse encontrado no cabeçalho, há um par de parênteses vazios.

No BNREmployee.h, remova a declaração officeAlarmCode:

```
#import "BNRPerson.h"
@class BNRAAsset;

@interface BNREmployee : BNRPerson
{
    NSMutableArray *_assets
}
@property (nonatomic) unsigned int employeeID;
@property (nonatomic) unsigned int officeAlarmCode;
@property (nonatomic) NSDate *hireDate;
@property (nonatomic, copy) NSArray *assets;
- (void)addAsset:(BNRAAsset *)a;
- (unsigned int)valueOfAssets;

@end
```

Compile e execute o programa. Seu comportamento não mudou. No entanto, mover a declaração officeAlarmCode para uma extensão de classe tem dois efeitos relacionados:

Primeiramente, os objetos que não são instâncias de **BNREmployee** não podem mais ver essa propriedade. Por exemplo, um objeto não-**BNREmployee** poderia tentar acessar o código do alarme de um funcionário como esse:

```
BNREmployee *mikey = [[BNREmployee alloc] init];
unsigned int mikeysCode = mikey.officeAlarmCode;
```

Essa tentativa resultaria em um erro de compilador que lê: “@interface não visível declara o método de instância officeAlarmCode”. A única interface visível para um objeto não-**BNREmployee** é o cabeçalho **BNREmployee**. E visto que a propriedade officeAlarmCode é declarada em uma extensão de classe em vez de no cabeçalho, ela não é visível (e, por isso, está indisponível) para objetos não-**BNREmployee**.

Em segundo lugar, o cabeçalho **BNREmployee** tem uma declaração a menos e, portanto, é um pouco mais simples. Isso é uma coisa boa. O cabeçalho destina-se a ser um quadro de avisos; sua função é anunciar o que os outros desenvolvedores precisam saber para fazer sua classe funcionar no código que eles escrevem. Uma quantidade demais de informações torna difícil a leitura e o uso de um cabeçalho por parte de outros desenvolvedores.

Ocultamento de mutabilidade

Agora vamos considerar um caso levemente diferente no que se refere à colocação de uma declaração em uma extensão de classe em vez de no cabeçalho da classe. No BNREmployee.h, você declarou uma propriedade assets que é uma **NSArray**, um método **addAsset:** e uma variável de instância de **_assets** que é uma **NSMutableArray**. Um desenvolvedor verá tanto a propriedade quanto a variável de instância anunciadas no cabeçalho e ficará inseguro quanto a qual delas você deseja que o usuário externo utilize.

Agora que você já sabe sobre as extensões de classe, a solução é simples: move a instância de **_assets** para a extensão de classe de **BNREmployee**. No BNREmployee.m, adicione esta declaração:

```
#import "BNREmployee.h"

// A class extension
@interface BNREmployee ()
{
    NSMutableArray *_assets;
}

@property (nonatomic) unsigned int officeAlarmCode;

@end

@implementation BNREmployee
...
```

No BNREmployee.h, remova a declaração de **_assets**:

```
#import "BNRPerson.h"
@class BNRAsset;

@interface BNREmployee : BNRPerson
{
    NSMutableArray *_assets
}
@property (nonatomic) unsigned int employeeID;
@property (nonatomic) NSDate *hireDate;
@property (nonatomic, copy) NSArray *assets;
- (void)addAsset:(BNRAsset *)a;
- (unsigned int)valueOfAssets;

@end
```

Agora, o array de ativos só é anunciado como um array imutável, assim os objetos não-**BNREmployee** precisarão usar o método **addAsset:** para manipular esse array. O fato de que há uma instância de **NSMutableArray** apoiando a propriedade **assets** é um detalhe de implementação privada da classe **BNREmployee**.

Cabeçalhos e heranças

Uma subclasse não tem acesso a sua extensão de classe da superclasse. **BNREmployee** é uma subclasse de **BNRPerson** e importa seu arquivo de cabeçalho da superclasse, **BNRPerson.h**. Portanto, **BNREmployee** sabe sobre o que foi declarado no cabeçalho de **BNRPerson**, mas não sabe nada sobre o que **BNRPerson** pode ter declarado em uma extensão de classe.

Por exemplo, se você implementou um método **hasDriversLicense** no **BNRPerson.m**, mas o declarou em uma extensão de classe em vez de em **BNRPerson.h**, a **BNREmployee** poderia não saber que esse método existia. Se tentou chamá-lo no **BNREmployee.m**:

```
BOOL canDriveCompanyVan = [self hasDriversLicense];
```

você obteve um erro do compilador: “@interface não visível declara o método de instância **hasDriversLicense**”.

Cabeçalhos e variáveis de instância geradas

Quando uma classe declara uma propriedade em seu cabeçalho, apenas os acessores dessa propriedade ficam visíveis para outros objetos. Os objetos não-**BNREmployee** (incluindo subclasses) não podem acessar diretamente as variáveis de instância geradas pelas declarações de propriedade.

Por exemplo, imagine que o **BNRPerson.h** declare esta propriedade:

```
@property (nonatomic) NSMutableArray *friends;
```

No **BNREmployee.m**, mesmo que **BNREmployee** seja uma subclasse de **BNRPerson**, você não pode acessar a variável de instância **_friends**:

```
[_friends addObject:@"Susan"]; // Error!
```

No entanto, você pode usar o acessor:

```
[self.friends addObject:@"Susan"];
```

Desafio

Reabra seu projeto a partir do desafio do Chapter 21.

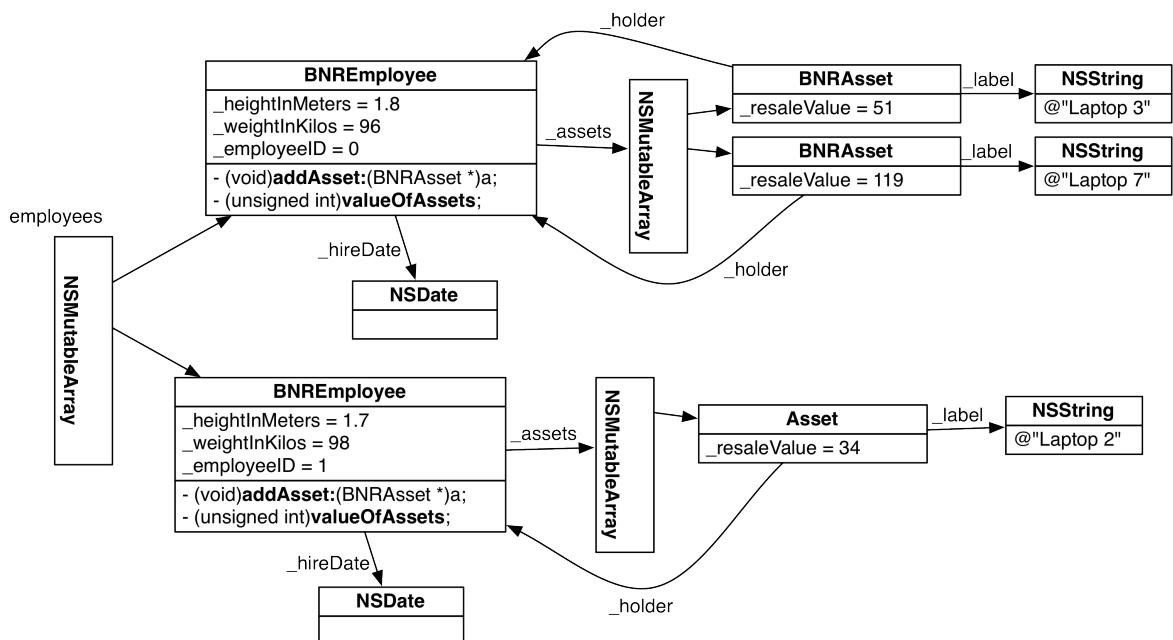
Apenas você (o criador da classe **BNRPortfolio**) precisa saber que está usando uma **NSMutableArray** para reter as instâncias de **BNRStockHolding**.

Mova a declaração de propriedade do array para uma extensão de classe no **BNRPortfolio.m** e adicione métodos para **BNRPortfolio**, a fim de permitir a adição e a remoção de ações.

Impedindo vazamentos de memória

É muito comum ter relacionamentos que seguem em duas direções. Por exemplo, talvez um ativo deva saber qual funcionário atualmente o retém. Vamos adicionar esse relacionamento. O novo diagrama de objetos será semelhante ao seguinte:

Figure 23.1 Adição do relacionamento de titular



Do ponto de vista de um projeto, você diria que está adicionando um ponteiro do filho (uma instância de **BNRAsset**) a seu pai (a instância de **BNREmployee** que o retém).

No **BNRAsset.h**, adicione uma variável de instância de ponteiro para conter o funcionário titular:

```

#import <Foundation/Foundation.h>
@class BNREmployee;

@interface BNRAsset : NSObject
@property (nonatomic, copy) NSString *label;
@property (nonatomic) BNREmployee *holder;
@property (nonatomic) unsigned int resaleValue;

@end
  
```

No **BNRAsset.m**, estenda o método **description** para exibir o **holder** (titular):

```
#import "BNRAsset.h"
#import "BNREmployee.h"

@implementation BNRAsset

- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@: $%d>", self.label, self.resaleValue];
    // Is holder non-nil?
    if (self.holder) {
        return [NSString stringWithFormat:@"<%@: $%d, assigned to %@>",
               self.label, self.resaleValue, self.holder];
    } else {
        return [NSString stringWithFormat:@"<%@: $%d unassigned>",
               self.label, self.resaleValue];
    }
}

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

@end
```

Isso nos leva a uma questão de estilo: quando as pessoas usam as classes **BNRAsset** e **BNREmployee** juntas, como você garante que esses dois relacionamentos sejam consistentes? Ou seja, um ativo deverá aparecer no array de ativos de um funcionário se, e somente se, o funcionário for o titular do ativo. Há três opções:

- Definir explicitamente ambos os relacionamentos:

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

- No método que define o ponteiro do filho, adicionar o filho à coleção do pai.

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

(Essa abordagem não é muito comum.)

- No método que adiciona o filho à coleção do pai, definir o ponteiro do filho.

Neste exercício, você usará esta última opção. No **BNREmployee.m**, estenda o método **addAsset:** para também definir **holder** (titular):

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

(Para um bug distrativo, tenha ambos os acessores automaticamente chamando um ao outro. Isso criará um loop infinito: **addAsset:** chama **setHolder:**, que chama **addAsset:**, que chama **setHolder:**, que chama....)

Compile e execute o programa. Você verá algo semelhante ao seguinte:

```
Employees: (
```

```

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

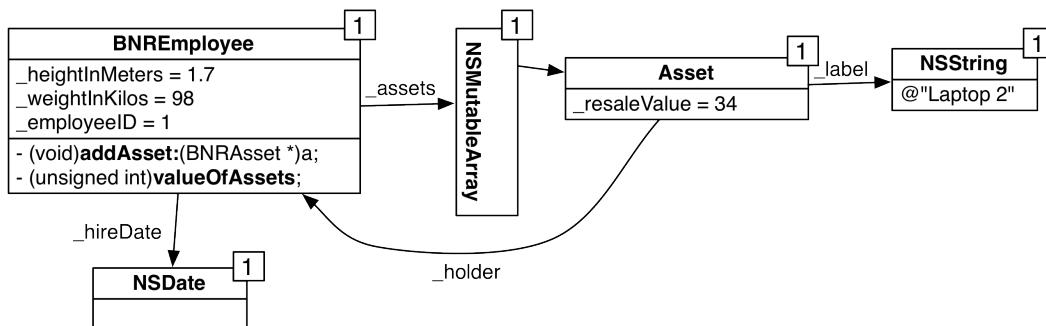
```

Observe que, agora, nenhum dos funcionários com ativos está sendo desalocado de modo apropriado. Além disso, nenhum dos ativos está sendo desalocado. Por quê?

Ciclos de referências fortes

O ativo possui o funcionário, o funcionário possui o array de ativos e o array de ativos possui o ativo. Esta é uma ilha de lixo criada por este círculo de propriedade. Esses objetos deveriam ser desalocados para liberar memória, mas não são. Isso é conhecido como um *ciclo de referência forte*. Ciclos de referências fortes são uma fonte muito comum de vazamentos de memória.

Figure 23.2 Todo objeto é propriedade de algum outro objeto



Para localizar ciclos de referências fortes em seu programa, você pode usar a ferramenta para criação de perfil da Apple, Instruments. Ao *criar o perfil* de um programa, você o monitora durante sua execução para saber o que está acontecendo em segundo plano com seu código e com o sistema. No entanto, a execução e o fechamento de seu programa ocorrem extremamente rápido. Para que você tenha tempo para criar o perfil, coloque cem segundos de `sleep()` no final de sua função `main()`:

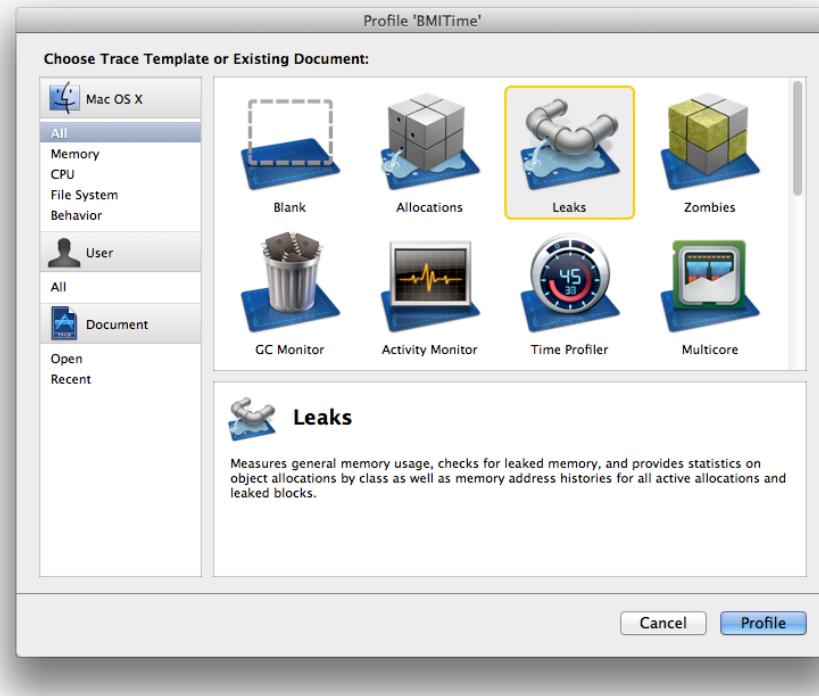
```

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

```

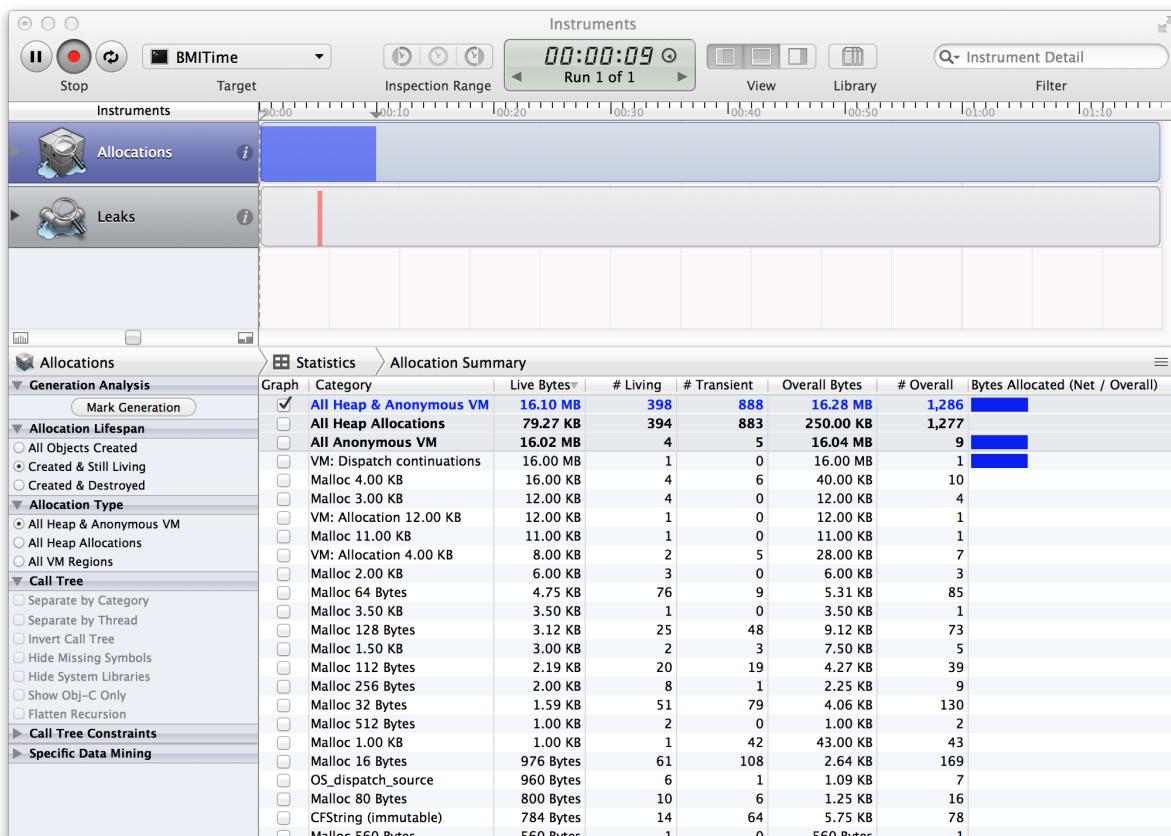
No Xcode, selecione Product → Profile no menu. O Instruments será iniciado. Quando a lista de possíveis instrumentos para criação de perfil for exibida, selecione Leaks:

Figure 23.3 Seleção de um criador de perfil



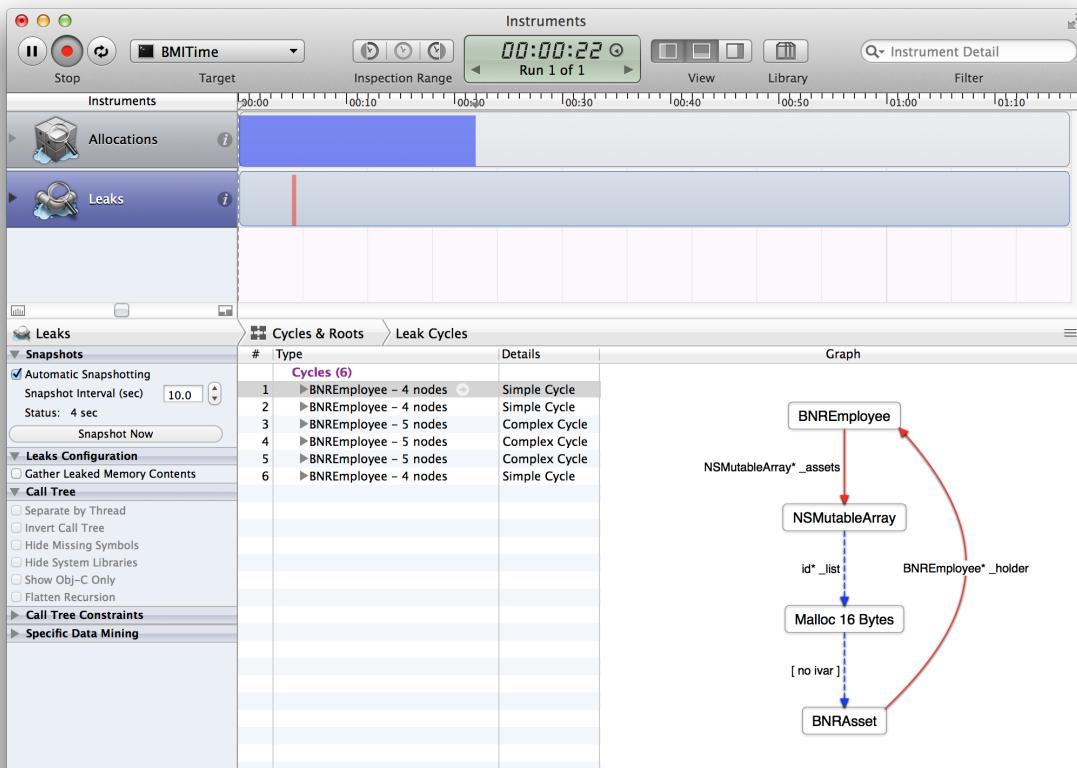
À medida que seu programa for executado, você poderá navegar pelo estado das coisas. Você pode escolher entre dois instrumentos no lado esquerdo da janela (Figure 23.4). Clicar no instrumento Allocations permitirá que você veja um gráfico de barras de tudo o que foi alocado em seu heap:

Figure 23.4 Instrumento Allocations



Para procurar por ciclos de referências fortes, clique em **Leaks** na barra de menus acima da tabela e escolha **Cycles & Roots** a partir do menu suspenso. Selecione um ciclo específico para visualizar um gráfico de objeto dele:

Figure 23.5 Vazamento de instrumento



Referências fracas

Como você corrige um ciclo de referência forte? Use uma referência fraca. Uma *referência fraca* é um ponteiro que *não* envolve propriedade. Para corrigir nosso ciclo de referência forte, um ativo não deve possuir seu titular. Edite o `BNRAsset.h` para fazer com que `holder` seja uma referência fraca:

```
#import <Foundation/Foundation.h>
@class BNREmployee;

@interface BNRAsset : NSObject

@property (nonatomic, copy) NSString *label;
@property (nonatomic, weak) BNREmployee *holder;
@property (nonatomic) unsigned int resaleValue;
@end
```

Compile e execute o programa. Observe que todos os objetos estão agora sendo desalocados corretamente.

Em um relacionamento pai-filho, a regra geral para impedir esse tipo de ciclo de referência forte é o pai possuir o filho, mas o filho não possuir o pai.

Zerando referências fracas

Para ver as referências fracas em ação, vamos adicionar outro array à combinação. E se você quisesse um array de todos os ativos – mesmo com aqueles que não foram atribuídos a um funcionário específico? Você poderia adicionar os ativos a um array à medida que eles fossem criados. Adicione algumas linhas de código ao `main.m`:

```
#import <Foundation/Foundation.h>
#import "BNREmployee.h"
#import "BNRAAsset.h"

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

        // Create an array of Employee objects
        NSMutableArray *employees = [[NSMutableArray alloc] init];

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

            // Create an instance of BNREmployee
            BNREmployee *mikey = [[BNREmployee alloc] init];

            // Give the instance variables interesting values
            [mikey setWeightInKilos:90 + i];
            [mikey setHeightInMeters:1.8 - i/10.0];
            [mikey setEmployeeID:i];

            // Put the employee in the employees array
            [employees addObject:mikey];
        }

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

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

            // Create an asset
            BNRAAsset *asset = [[BNRAAsset alloc] init];

            // Give it an interesting label
            NSString *currentLabel = [NSString stringWithFormat:@"Laptop %d", i];
            asset.label = currentLabel;
            asset.resaleValue = i * 17;

            // Get a random number between 0 and 9 inclusive
            NSUInteger randomIndex = random() % [employees count];

            // Find that employee
            BNREmployee *randomEmployee = [employees objectAtIndexAtIndex:randomIndex];

            // Assign the asset to the employee
            [randomEmployee addAsset:asset];

            [allAssets addObject:asset];
        }

        NSLog(@"%@", employees);

        NSLog(@"Giving up ownership of one employee");

        [employees removeObjectAtIndex:5];

        NSLog(@"%@", allAssets);

        NSLog(@"Giving up ownership of arrays");

        allAssets = nil;
        employees = nil;
    }
    sleep(100);
    return 0;
}
```

Antes de compilar e executar seu programa, pense em como você espera que seja o resultado. Você verá o conteúdo do array `allAssets` – depois que o Funcionário 5 tiver sido desalocado. Qual será o status dos ativos do Funcionário 5 a essa altura? Esses ativos perdem um proprietário (Funcionário 5), mas ainda são de propriedade de `allAssets`, de modo que eles não serão desalocados.

E com relação ao `holder` dos ativos que antes eram propriedade do Funcionário 5? Quando o objeto para o qual uma referência fraca aponta é desalocado, a variável de ponteiro é *zerada*, ou definida para `nil`. Desse modo, os ativos do Funcionário 5 não serão desalocados e suas variáveis `holder` serão automaticamente definidas para `nil`.

Agora, compile e execute o programa e verifique seu resultado:

```
Employees: (
    "<Employee 0: $0 in assets>",
    ...
    "<Employee 9: $136 in assets>"
)
Giving up ownership of one employee
deallocating <Employee 5: $136 in assets>
allAssets: (
    "<Laptop 0: $0, assigned to <Employee 3: $68 in assets>>",
    "<Laptop 1: $17, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 2: $34, assigned to <Employee 7: $34 in assets>>",
    "<Laptop 3: $51 unassigned>",
    "<Laptop 4: $68, assigned to <Employee 3: $68 in assets>>",
    "<Laptop 5: $85 unassigned>",
    "<Laptop 6: $102, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 7: $119, assigned to <Employee 2: $119 in assets>>",
    "<Laptop 8: $136, assigned to <Employee 9: $136 in assets>>",
    "<Laptop 9: $153, assigned to <Employee 1: $153 in assets>>"
)
Giving up ownership of arrays
deallocating <Laptop 3: $51 unassigned>
...
deallocating <Laptop 8: $136 unassigned>
```

Um resumo rápido: uma referência forte não deixará que o objeto ao qual ela aponta seja desalocado. Uma referência fraca não fará isso. Assim, as propriedades e variáveis de instância que são marcadas como fracas estão apontando para objetos que podem desaparecer. Se isso acontecer, a propriedade ou variável de instância será definida para `nil`, em vez de continuar a apontar para onde o objeto costumava existir.

Se você estiver declarando explicitamente uma variável de ponteiro que deve ser fraca, marque-a com `__weak` como a seguir:

```
__weak BNRPerson *parent;
```

Para os mais curiosos: história da contagem de referência manual e da ARC

Como mencionado no início do Chapter 21, antes de a ARC (Automatic Reference Counting, contagem de referência automática) ser adicionada ao Objective-C, você tinha a *contagem de referência manual*, que usava as *contagens de retenção*. Com a contagem de referência manual, as mudanças de propriedade somente aconteciam quando você enviava uma mensagem explícita a um objeto que diminuía ou aumentava a contagem de retenção.

```
[anObject release]; // anObject loses an owner
[anObject retain]; // anObject gains an owner
```

Você usaria esses tipos de chamadas principalmente em métodos acessores (em que o novo valor era retido e o valor antigo era liberado) e em métodos `dealloc` (em que todos os objetos retidos anteriormente eram liberados). O método `setHolder:` para `BNRAsset` assemelhava-se a este:

```
- (void)setHolder:(BNREmployee *)newEmp
{
    // Take ownership of the new holder
    [newEmp retain];

    // Give up ownership of the old holder
    [holder release];

    // Set the pointer to point to the new holder
    holder = newEmp;
}
```

O método **dealloc** ficaria parecido com este:

```
- (void)dealloc
{
    // You're dying, so give up ownership of all objects you used to own
    [label release];
    [holder release];
    [super dealloc];
}
```

E o método **description**? Ele cria e retorna uma string. A classe **BNRAsset** deve solicitar a propriedade da string? Isso não faria sentido; o ativo está se desfazendo da string que criou. Quando usa o método **autorelease** para liberar um objeto automaticamente, você o está marcando para ser enviado para o método **release** (liberação) futuramente. Antes da ARC, o método **description** de **BNRAsset** ficaria desta forma:

```
- (NSString *)description
{
    NSString *result = [[NSString alloc] initWithFormat:@"<%@: %d >",
                           [self label], [self resaleValue]];
    [result autorelease];
    return result;
}
```

Quando ele seria enviado para **release**? Quando o pool atual de liberação automática ficasse vazio:

```
// Create the autorelease pool
NSAutoreleasePool *arp = [[NSAutoreleasePool alloc] init];
BNRAsset *asset = [[BNRAsset alloc] init];

NSString *d = [asset description];
// The string that d points to is in the autorelease pool

 NSLog(@"%@", d);

[arp drain]; // The string is sent the message release
```

A ARC *usa* automaticamente o pool de liberação automática, mas você deve criar e esvaziar o pool. Quando a ARC foi criada, também obtivemos uma nova sintaxe para a criação de um pool de liberação automática. O código acima agora é parecido com o seguinte:

```
// Create the autorelease pool
@autoreleasepool {
    BNRAsset *asset = [[BNRAsset alloc] init];

    NSString *d = [asset description];
    // The string that d points to is in the autorelease pool
} // The pool is drained
```

Regras de contagem de retenção

Há um conjunto de convenções de gerenciamento de memória seguido por todos os programadores de Objective-C. Se você estiver usando a ARC, ela seguirá essas convenções em segundo plano.

Nessas regras, usamos a palavra “você” para indicar “uma instância de qualquer classe com que você esteja trabalhando atualmente”. É uma forma útil de empatia: você imagina que é o objeto que você está escrevendo. Então, por exemplo, “se você retiver a string, ela não será desalocada” realmente significa “se uma instância da classe com que você está trabalhando atualmente retiver a string, ela não será desalocada”.

Então, aqui temos as regras. (Os detalhes de implementação estão entre parênteses.)

- Se você criar um objeto usando um método cujo nome comece com **alloc** ou **new** ou contenha **copy**, então você terá assumido a propriedade dele. (Ou seja, assume que o novo objeto tem uma contagem de retenção de 1 e *não* está no pool de liberação automática.) Você tem a responsabilidade de liberar o objeto quando não precisar mais dele. Estes são alguns dos métodos comuns que exprimem propriedade: **alloc** (que é sempre seguido por um método **init**), **copy** e **mutableCopy**.

- Um objeto criado com *qualquer* outro meio *não* será de sua propriedade. (Ou seja, assuma que ele contém uma contagem de retenção de um e já está no pool de liberação automática e, portanto, está condenado, a menos que seja retido antes que o pool de liberação automática seja esvaziado.)
- Se você *não* possuir um objeto e deseja garantir sua existência contínua, assuma a propriedade enviando a ele a mensagem **retain**. (Isso aumenta a contagem de retenção.)
- Quando você possuir um objeto e não precisar mais dele, abra mão da propriedade enviando a ele a mensagem **release** ou **autorelease**. (**release** diminui imediatamente a contagem de retenção. **autorelease** faz com que a mensagem **release** seja enviada quando o pool de liberação automática for esvaziado.)
- Contanto que um objeto tenha, pelo menos, um proprietário, ele continuará existindo. (Quando sua contagem de retenção chegar a zero, a mensagem **dealloc** será enviada.)

Um dos artifícios para entender o gerenciamento de memória é pensar localmente. A classe **BNRAsset** não precisa saber nada sobre os outros objetos que também se interessam por sua variável `label`. Contanto que uma instância de **BNRAsset** retenha os objetos que deseja manter, você não terá nenhum problema. Os programadores novos na linguagem muitas vezes cometem o erro de tentar manter marcações nos objetos por todo o aplicativo. Não faça isso. Se você seguir essas regras e sempre pensar em uma classe localmente, nunca terá de preocupar-se com o que o restante de um aplicativo está fazendo com um objeto.

Seguindo a ideia de propriedade, agora fica claro porque você precisa liberar automaticamente a string em seu método **description**: O objeto “employee” criou a string, mas não quer ter a propriedade dela. Ele quer se desfazer dela.

24

Classes de coleção

Uma *classe de coleção* é aquela cujas instâncias armazenam ponteiros para outros objetos. Você já usou duas classes de coleção: **NSArray** e sua subclasse **NSMutableArray**. Neste capítulo, você analisará os arrays de modo mais aprofundado e aprenderá algumas outras classes de coleção: **NSSet/NSMutableSet** e **NSDictionary/NSMutableDictionary**.

NSSet/NSMutableSet

Um *conjunto* é uma coleção que não tem um senso de ordem, e cujo objeto em particular pode aparecer somente uma vez em um conjunto.

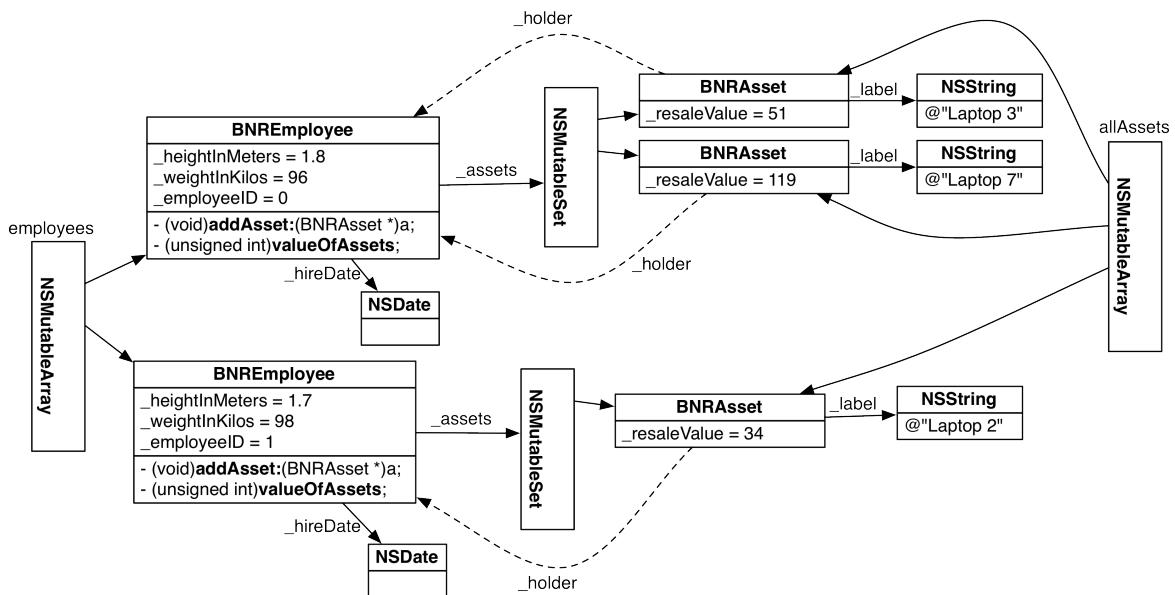
Os conjuntos são essencialmente úteis para fazer a pergunta: “Está aí?”. Por exemplo, você pode ter um conjunto de URLs que não sejam apropriados para crianças. Antes de exibir qualquer página da web a uma criança, você fará uma rápida verificação para confirmar se o URL está no conjunto. Os conjuntos são mais rápidos na hora de testar a associação de objetos do que os arrays.

Como os arrays, os conjuntos vêm como imutáveis e mutáveis: Uma **NSSet** é imutável — você não pode adicionar ou remover objetos após o conjunto ter sido criado. **NSMutableSet** é a subclasse que adiciona a capacidade de adicionar e remover objetos de um conjunto.

Nesta seção, você vai mudar seu programa de modo que o relacionamento funcionário-ativo use uma **NSMutableSet** em vez de uma **NSMutableArray**.

A **NSMutableSet** é uma boa opção para descrever o relacionamento funcionário-ativo: uma variável **assets** do funcionário não tem ordem inerente, e um ativo jamais deve aparecer duas vezes na mesma coleção de assets do funcionário.

Figure 24.1 Utilização de **NSMutableSet** para ativos



No BNREmployee.h, altere a declaração da propriedade:

```
#import "BNRPerson.h"
@class BNRAAsset;

@interface BNREmployee : BNRPerson

@property (nonatomic) unsigned int employeeID;
@property (nonatomic) NSDate *hireDate;
@property (nonatomic, copy) NSSet *assets;
- (void)addAsset:(BNRAAsset *)a;
- (unsigned int)valueOfAssets;

@end
```

No BNREmployee.m, altere a declaração de variável de instância e certifique-se de que você criou uma instância da classe correta:

```
// A class extension
@interface BNREmployee ()
{
    NSMutableSet *_assets;
}
@property (nonatomic) unsigned int officeAlarmCode;
@end

@implementation BNREmployee

...

- (void)addAsset:(BNRAAsset *)a
{
    if (!_assets) {
        _assets = [[NSMutableSet alloc] init];
    }
    [_assets addObject:a];
    a.holder = self;
}

...
```

Compile e execute o programa. Ele funcionará da mesma forma.

Você não pode acessar um objeto em um conjunto pelo índice, pois não há senso de ordem em um conjunto. Em vez disso, tudo o que você pode fazer é perguntar: “Há um destes lá?”. Você faz essa pergunta com o seguinte método **NSSet**:

```
- (BOOL)containsObject:(id)x;
```

Quando você envia essa mensagem para um conjunto, ela vai por meio de sua coleção de objetos que busca um objeto igual a x. Se encontrar um, ela retornará YES; caso contrário, retornará NO.

Isso nos leva a fazer uma pergunta mais aprofundada: o que significa *igual*? A classe **NSObject** define um método chamado **isEqual:**. Para verificar se dois objetos são iguais, você usa o método **isEqual:**:

```
if ([myDoctor isEqual:yourTennisPartner]) {
    NSLog(@"my doctor is equal to your tennis partner");
}
```

NSObject contém uma implementação simples de **isEqual:**. Isso será semelhante ao seguinte:

```
- (BOOL)isEqual:(id)other
{
    return (self == other);
```

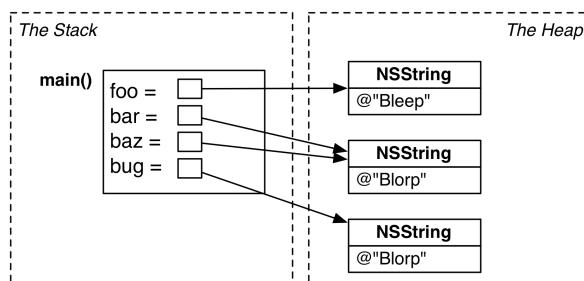
```
}
```

Portanto, se você não tiver substituído **isEqual:**, o trecho de código será equivalente a:

```
if (myDoctor == yourTennisPartner) {
    NSLog(@"my doctor is equal to your tennis partner");
}
```

Algumas classes sobrescrevem **isEqual:**. Por exemplo, em **NSString**, **isEqual:** é substituído para comparar os caracteres na string. Para essas classes, existe uma diferença entre *igual* e *idêntico*. Considere uma situação em que você pode ter quatro ponteiros **NSString**:

Figure 24.2 Igual vs. idêntico



Você poderia descrever os relacionamentos entre esses ponteiros de várias maneiras:

- **foo** não é igual ou idêntico a qualquer outro.
- **bar** e **baz** são *iguais* e *idênticos* porque os objetos para os quais eles apontam têm as mesmas letras na mesma ordem, e, na verdade, eles apontam para o mesmo objeto.
- **baz** e **bug** são *iguais*, mas *não idênticos*.

Portanto, objetos idênticos sempre são iguais. Objetos iguais nem sempre são idênticos.

Essa diferença é importante? Sim. Por exemplo, **NSMutableArray** tem dois métodos:

- **(NSUInteger)indexOfObject:(id)anObject;**
- **(NSUInteger)indexOfObjectIdenticalTo:(id)anObject;**

O primeiro analisa a coleção perguntando a cada objeto “**isEqual:anObject?**” O segundo analisa a coleção perguntando a cada objeto “**== anObject**”?

NSDictionary/NSMutableDictionary

Um *dicionário* é uma coleção de *pares chave-valo*r. A *chave* é normalmente uma string, e o *valor* pode ser qualquer tipo de objeto. Os dicionários são indexados por chave: você fornece uma chave e obtém o valor (um objeto) associado à chave específica. As chaves em um dicionário são exclusivas e os pares chave-valo do dicionário não são mantidos em nenhuma ordem particular.

Como arrays e conjuntos, dicionários podem ser mutáveis (**NSMutableDictionary**) ou imutáveis (**NSDictionary**). Como **NSArray**, **NSDictionary** tem um atalho que você pode usar ao criar um dicionário imutável. A sintaxe literal do dicionário é formada com o símbolo @ e chaves. Dentro das chaves, você fornece uma lista delimitada por vírgula dos pares chave-valo e separa cada chave de seu valor com um dois pontos.

Por exemplo, imagine que você quisesse um lugar para manter o número de luas de cada planeta no sistema solar. Veja como você criaria uma **NSDictionary** com os nomes dos planetas como chaves e o número de luas como valores.

```
NSDictionary *numberOfMoons = @{@"Mercury" : @0,
                               @"Venus" : @0,
                               @"Earth" : @1,
                               @"Mars" : @2,
                               @"Jupiter" : @67,
                               @"Saturn" : @62,
                               @"Uranus" : @27,
                               @"Neptune" : @13, };
```

As chaves são objetos **NSString**, e os valores são objetos **NSNumber**. Ambos são criados no local usando a sintaxe literal.

Veja como você acessaria um item a partir desse dicionário:

```
NSNumber *marsMoonCount = numberOfMoons[@"Mars"];
```

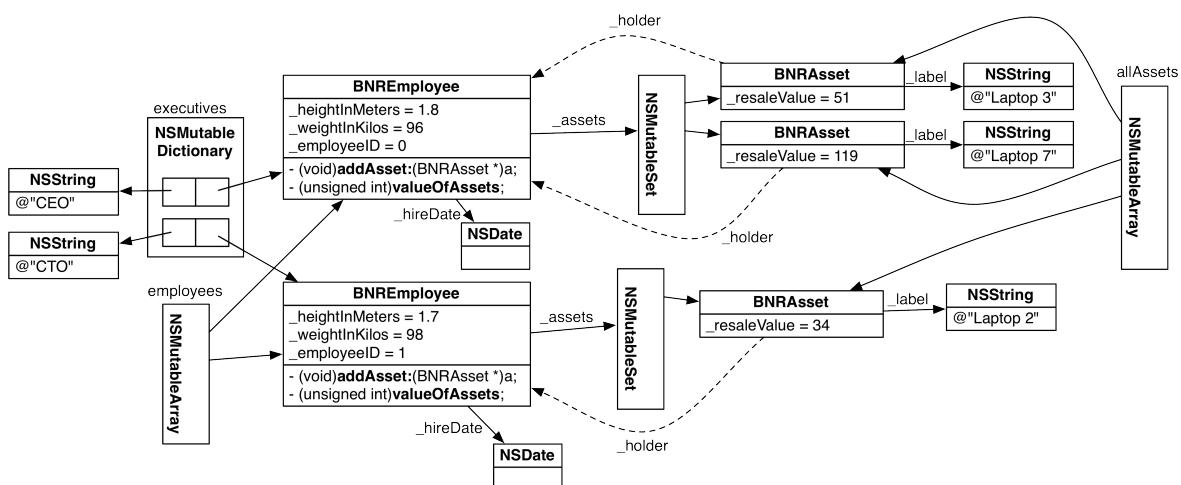
Isso é similar ao modo como você acessa um item em um array, exceto entre colchetes, onde você fornece a chave do item em vez de seu índice inteiro.

Às vezes é útil *aninhar* coleções. Por exemplo, aqui temos uma instância de **NSDictionary** cujos valores são instâncias de **NSArray**.

```
NSDictionary *innerPlanetsMoons = @{@"Mercury" : @[], // @[] is an empty array, equivalent to [NSArray array]
                                    @"Venus" : @[],
                                    @"Earth" : @[@[@"Luna"]],
                                    @"Mars" : @[@[@"Deimos", @"Phobos"]]
};
```

Agora, você vai adicionar um dicionário mutável de executivos ao projeto do BMITime. A chave será o cargo de um executivo, e o valor será uma instância de **BNREmployee**. O primeiro funcionário no array `employees` será colocado no dicionário em @"CEO"; o segundo em @"CTO".

Figure 24.3 Duas instâncias de **BNREmployee** em uma **NSMutableDictionary**



Esse dicionário será uma **NSMutableDictionary**, então você vai usar `alloc` e `init` para criá-lo e não a sintaxe literal mostrada acima.

Altere `main.m` para criar e preencher uma **NSMutableDictionary**. Exiba algumas informações executivas e, em seguida, defina o ponteiro para o dicionário para `nil` de modo que você possa ver a desalocação do dicionário.

```

// Create an array of BNREmployee objects
NSMutableArray *employees = [[NSMutableArray alloc] init];

// Create a dictionary of executives
NSMutableDictionary *executives = [[NSMutableDictionary alloc] init];

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

    // Create an instance of BNREmployee
    BNREmployee *mikey = [[BNREmployee alloc] init];

    // Give the instance variables interesting values
    [mikey setWeightInKilos:90 + i];
    [mikey setHeightInMeters:1.8 - i/10.0];
    [mikey setEmployeeID:i];

    // Put the employee in the employees array
    [employees addObject:mikey];

    // Is this the first employee?
    if (i == 0) {
        [executives setObject:mikey forKey:@"CEO"];
    }

    // Is this the second employee?
    if (i == 1) {
        [executives setObject:mikey forKey:@"CTO"];
    }

}

...

NSLog(@"%@", allAssets);

// Print out the entire dictionary
NSLog(@"%@", executives);

// Print out the CEO's information
NSLog(@"%@", executives[@"CEO"]);
executives = nil;

 NSLog(@"Giving up ownership of arrays");

allAssets = nil;
employees = nil;
}
return 0;
}

```

Compile e execute o programa. O dicionário de executivos se desconectará sozinho:

```

executives = {
    CEO = "<Employee 0: $0 in assets>";
    CTO = "<Employee 1: $153 in assets>";
}

CEO: "<Employee 0: $0 in assets>"

```

Antes de o Objective-C ter um subíndice, utilizamos métodos em vez de colchetes:

```

NSMutableDictionary *dict = [[NSMutableDictionary alloc] init];
[dict setObject:@"Hola" forKey:@"Hello"];
NSString *greeting = [dict objectForKey:@"Hello"];

```

As chaves em um dicionário são exclusivas. Se você tentar adicionar um segundo objeto em uma chave existente, o primeiro par chave-valor será substituído.

```
// Create a dictionary
NSMutableDictionary *friends = [NSMutableDictionary dictionaryWithObjectsAndKeys:[NSMutableArray dictionary], @"bestFriend", betty, @"bestFriend", jane, @"bestFriend"];
```

Agora, friends tem apenas um par chave-valor (bestFriend => jane).

Objetos imutáveis

A maioria dos programadores iniciantes é surpreendida pela imutabilidade de **NSArray**, **NSSet** e **NSDictionary**. Por que é que alguém desejaría uma lista que não pode ser alterada? As razões são desempenho e segurança:

Você não confia nas pessoas com quem trabalha. Ou seja, você deseja permitir que elas vejam um array, mas não quer que sejam capazes de alterá-lo. Uma abordagem mais branda é fornecer uma **NSMutableArray** a eles, mas dizer que é uma **NSArray**. Por exemplo, suponha que temos o seguinte método:

```
- (NSArray *)odds
{
    NSMutableArray *oddsArray = [[NSMutableArray alloc] init];
    int i = 1;
    while ([oddsArray count] < 30) {
        [oddsArray addObject:[NSNumber numberWithInt:i]];
        i += 2;
    }
    return oddsArray;
}
```

Qualquer pessoa que chamar esse método assumirá a partir de sua declaração

```
- (NSArray *)odds;
```

que ele está retornando uma **NSArray** imutável. Se o chamador tentar adicionar ou remover itens do array retornado, o compilador emitirá um aviso – mesmo que ele, de fato, seja uma instância de **NSMutableArray**.

Usar uma coleção imutável economiza memória e melhora o desempenho, visto que essa coleção nunca precisa ser copiada. Com um objeto mutável, existe a possibilidade de algum outro código alterar o objeto sem você saber enquanto você o estiver usando. Para evitar essa situação, você precisaria fazer uma cópia privada da coleção. E, assim, para todo mundo, o que levaria a várias cópias de um objeto potencialmente grande.

Com objetos imutáveis, fazer uma cópia é desnecessário. Na verdade, onde o método **copy** de **NSMutableArray** fizer uma cópia de si mesmo e retornar um ponteiro para o novo array, o método **copy** de **NSArray** não faz nada – ele apenas retorna silenciosamente um ponteiro para si mesmo.

Objetos imutáveis são bastante comuns na programação em Objective-C. Na Foundation, há muitas classes que criam instâncias imutáveis: **NSArray**, **NSString**, **NSAttributedString**, **NSData**, **NSCharacterSet**, **NSDictionary**, **NSSet**, **NSIndexSet** e **NSURLRequest**.

Todas elas têm subclasses mutáveis: **NSMutableArray**, **NSMutableString**, **NSMutableAttributedString** etc.

NSDate e **NSNumber** são imutáveis, mas não têm subclasses mutáveis. Se você precisar de uma nova data ou número, então, precisa criar um novo objeto.

Classificação de arrays

Os arrays normalmente requerem classificação. Os arrays imutáveis não podem ser classificados, mas os mutáveis podem. Existem várias maneiras de classificar uma **NSMutableArray**. A mais comum é usando o método **NSMutableArray**:

```
- (void)sortUsingDescriptors:(NSArray *)sortDescriptors;
```

O argumento é um array de objetos **NSSortDescriptor**. Um *descritor de classificação* apresenta o nome de uma propriedade dos objetos contidos no array e informa se essa propriedade deve ser classificada em ordem

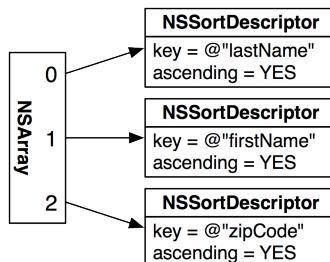
crescente ou decrescente. Imagine que você tenha uma lista de médicos. Se você quisesse classificar a lista pelo último nome em ordem crescente (A-Z), você criaria o seguinte descritor de classificação:

```
NSSortDescriptor *lastAsc = [NSSortDescriptor sortDescriptorWithKey:@"lastName" ascending:YES];
```

A propriedade que você classifica pode ser qualquer variável de instância ou o resultado de qualquer método do objeto.

Por que você passa um array de descritores de classificação? E se dois médicos tiverem o mesmo sobrenome? Você pode especificar assim: “Classificar por sobrenome em ordem crescente e se os sobrenomes forem os mesmos, classificar por nome em ordem crescente e, se os nomes forem os mesmos, classificar por CEP.”

Figure 24.4 Classificar por lastName, em seguida por firstName, finalmente por zipCode



Vamos retornar ao projeto do BMITime para ver a classificação em prática. Em `main()`, antes de registrar o array `employees`, classifique-o por `valueOfAssets`. Se dois funcionários tiverem ativos do mesmo valor, classifique-os por `employeeID`. Edite o `main.m`:

```
...
    [allAssets addObject:asset];
}

NSSortDescriptor *voa = [NSSortDescriptor sortDescriptorWithKey:@"valueOfAssets"
                                                       ascending:YES];
NSSortDescriptor *eid = [NSSortDescriptor sortDescriptorWithKey:@"employeeID"
                                                       ascending:YES];
[employees sortUsingDescriptors: @[voa, eid]];
NSLog(@"%@", employees);
...
```

Compile e execute o programa. Você verá a lista de funcionários ordenada corretamente:

```
Employees: (
    "<Employee 0: $0 in assets>",
    "<Employee 4: $0 in assets>",
    "<Employee 8: $0 in assets>",
    "<Employee 7: $34 in assets>",
    "<Employee 3: $68 in assets>",
    "<Employee 2: $119 in assets>",
    "<Employee 6: $119 in assets>",
    "<Employee 5: $136 in assets>",
    "<Employee 9: $136 in assets>",
    "<Employee 1: $153 in assets>"
```

Conjuntos e dicionários são desordenados por natureza, por isso eles não são normalmente classificados.

Filtragem

Quando você filtra uma coleção, você compara seus objetos a uma declaração lógica para obter uma coleção resultante que contém apenas objetos para os quais a declaração é verdadeira.

Um *predicado* contém uma declaração que pode ser verdadeira, como: “O `employeeID` é maior que 75.” Existe uma classe chamada **NSPredicate**. **NSMutableArray** tem um método útil para descartar todos os objetos que não satisfazem o predicado:

```
- (void)filterUsingPredicate:(NSPredicate *)predicate;
```

Com **NSArray**, você não pode remover objetos que não correspondam ao predicado. Em vez disso, **NSArray** tem um método que cria um novo array que contém todos os objetos que satisfazem o predicado:

```
- (NSArray *)filteredArrayUsingPredicate:(NSPredicate *)predicate;
```

Imagine que você tenha que resgatar todos os ativos concedidos aos funcionários que atualmente têm ativos de mais de US\$ 70 no total. Adicione o código próximo do final de `main.m`:

```
...
// Print out the CEO's information
NSLog(@"CEO: %@", executives[@"CEO"]);
executives = nil;

NSPredicate *predicate = [NSPredicate predicateWithFormat:
    @"holder.valueOfAssets > 70"];
NSArray *toBeReclaimed = [allAssets filteredArrayUsingPredicate:predicate];
NSLog(@"toBeReclaimed: %@", toBeReclaimed);
toBeReclaimed = nil;

NSLog(@"Giving up ownership of arrays");

allAssets = nil;
employees = nil;
}
return 0;
}
```

Compile e execute o programa. Você verá uma lista de ativos:

```
toBeReclaimed: (
    "<Laptop 1: $17, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 3: $51, assigned to <Employee 5: $136 in assets>>",
    "<Laptop 5: $85, assigned to <Employee 5: $136 in assets>>",
    "<Laptop 6: $102, assigned to <Employee 6: $119 in assets>>",
    "<Laptop 8: $136, assigned to <Employee 9: $136 in assets>>",
    "<Laptop 9: $153, assigned to <Employee 1: $153 in assets>>"
)
```

A string de formatação usada para criar o predicado pode ser muito complexa. Se você realiza muita filtragem de coleções, certifique-se de ler o *Predicate Programming Guide* (Guia de programação de predicados) da Apple.

A filtragem pode ser feita com conjuntos e arrays. **NSSet** tem o método:

```
- (NSSet *)filteredSetUsingPredicate:(NSPredicate *)predicate;
```

, e **NSMutableSet** tem o método:

```
- (void)filterUsingPredicate:(NSPredicate *)predicate;
```

Coleções e propriedades

Ao adicionar um objeto a uma coleção, a coleção reivindica a propriedade dele. Ao remover o objeto da coleção, esta renuncia a propriedade. Isso é verdadeiro para **NSMutableArray**, **NSMutableSet** e **NSMutableDictionary**. (Uma coleção imutável também reivindica a propriedade de seus objetos, mas a imutabilidade da coleção significa que todos os objetos na coleção são reconhecidos como próprios quando a coleção é criada e renegados como próprios quando a coleção é desalocada.)

Tipos primitivos em C

As coleções abordadas neste capítulo armazenam apenas objetos. E se você quiser uma coleção de floats ou ints? Você pode incluir tipos de número em C comuns usando **NSNumber**.

Você pode criar uma instância literal de **NSNumber** usando o símbolo @ – de modo semelhante a como você cria instâncias literais de **NSString**. Por exemplo, se quisesse colocar os números 4 e 5.6 em um array, você criaria a instância de **NSNumber** e, em seguida, adicionaria o objeto **NSNumber** ao array:

```
NSMutableArray *list = [[NSMutableArray alloc] init];
[list addObject:@4];
[list addObject:@5.6];
```

Observe que você não pode fazer uma correspondência diretamente com uma **NSNumber**, apenas com primitivos. Você deve primeiramente extrair o valor primitivo usando um dos vários métodos **NSNumber**, fazer o cálculo, em seguida, reincluir o valor em uma **NSNumber**. Você pode encontrar os métodos para extrair e reincluir valores primitivos na referência da classe **NSNumber**.

E quanto a structs? Você pode incluir um ponteiro para uma struct em uma instância de outra classe empacotadora – **NSValue** (a superclasse de **NSNumber**). Structs comumente usadas, tal como **NSPoint** (que contém os valores x e y de uma coordenada), podem ser encaixotados usando instâncias de **NSValue**:

```
NSPoint somePoint = NSMakePoint(100, 100);
NSValue *pointValue = [NSValue valueWithPoint:somePoint];
[list addObject:pointValue];
```

As instâncias de **NSValue** podem ser usadas para manter qualquer valor escalar. Leia a referência da classe **NSValue** para saber mais.

Coleções e nil

Você não pode adicionar **nil** em nenhuma das classes de coleção que abordamos. E se for necessário inserir essa ideia de inexistência, um “buraco”, em uma coleção? Há uma classe chamada **NSNull**. Há exatamente uma instância de **NSNull**, que é um objeto que representa a inexistência. Veja aqui um exemplo:

```
NSMutableArray *hotel = [[NSMutableArray alloc] init];
// Lobby on the ground floor
[hotel addObject:lobby];
// Pool on the second
[hotel addObject:pool];
// The third floor has not been built out
[hotel addObject:[NSNull null]];
// Bedrooms on fourth floor
[hotel addObject:bedrooms];
```

Desafio: leitura

Explore as referências de classe de **NSArray**, **NSMutableArray**, **NSDictionary** e **NSSMutableDictionary**. Você usará essas classes todos os dias.

Desafio: principais holdings

Esse desafio e o próximo baseiam-se no desafio do Chapter 22. Adicione um método à classe **BNRPortfolio** que retorna uma **NSArray** apenas dos três principais holdings mais valiosos, classificados pelo valor atual em dólares. Teste isso em **main()**.

Desafio: holdings classificados

Adicione outro método à classe **BNRPortfolio** que retorna uma **NSArray** de todas as suas ações, classificadas alfabeticamente por símbolo. Teste este método em **main()** também.

25

Constantes

Passamos muito tempo falando sobre variáveis, as quais, como o próprio nome indica, mudam seus valores conforme o programa é executado. No entanto, há partes de informação que *não* mudam de valor. Por exemplo, a constante matemática π nunca muda. Chamamos isso de *constantes*; os programadores em Objective-C definem as constantes de duas maneiras comuns: `#define` e variáveis globais.

No Xcode, crie uma nova Foundation Command Line Tool chamada Constants.

Nas bibliotecas padrão do C, as constantes são definidas com o uso da diretiva de pré-processador `#define`. A parte matemática da biblioteca padrão do C é declarada no arquivo `math.h`. Uma das constantes definidas lá é `M_PI`. Use-a no `main.m`:

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSLog(@"\u03c0 is %f", M_PI);
    }
    return 0;
}
```

Ao compilar e executá-lo você verá:

```
π is 3.141593
```

Para a definição da constante `M_PI`, pressione a tecla Command e, em seguida, clique em `M_PI` em seu código.

Figure 25.1 Definição para M_PI

```

Constants.xcodeproj — math.h
Constants: Ready | Today at 2:19 PM
No Issues
File Edit View Project Navigator Editor Run Scheme Help
OS X 10.8 /usr/include/math.h No Selection
Constants
  Constants
    main.m
    Constants.1
    Supporting Files
    Frameworks
    Products
extern long double lgammal_r(long double, int *) __OSX_AVAILABLE_STARTING(__MAC_10_6, __IPHONE_3_1);
#endif /* _REENTRANT */

/****************************************************************************
 * POSIX/UNIX extensions to C99
 ****/
#if __DARWIN_C_LEVEL >= 199506L
extern double j0(double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double j1(double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double jn(int, double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double y0(double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double y1(double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double yn(int, double) __OSX_AVAILABLE_STARTING(__MAC_10_0, __IPHONE_3_2);
extern double scalb(double, double);
extern int signgam;

/* Even though these might be more useful as long doubles, POSIX requires
   that they be double-precision literals.
#define M_E 2.71828182845904523536028747135266250 /* e */
#define M_LOG2E 1.44269504088906340735902468100189214 /* log2(e)
#define M_LOG10E 0.434294481903251827651128918916605082 /* log10(e)
#define M_LN2 0.693147180559945389417232212458176568 /* loge(2)
#define M_LN10 2.30258509299404568401799145468436421 /* loge(10)
#define M_PI 3.14159265358979323846264338327950288 /* pi */
#define M_PI_2 1.57079632679489661923132169163975144 /* pi/2 */
#define M_PI_4 0.78539816339744830961566084581987521 /* pi/4
#define M_1_PI 0.318309886183790671537767526745028724 /* 1/pi
#define M_2_PI 0.636619772367581343075535053490057448 /* 2/pi
#define M_2_SQRTPI 1.12837916709551257389615890312154517 /* 2/sqrt(pi)
#define M_SQRT2 1.41421356237309504886168872420969808 /* sqrt(2)
#define M_SQRT1_2 0.707106781186547524400844362104849039 /* 1/sqrt(2)

#define MAXFLOAT 0x1,fffffepp+127f
#endif /* __DARWIN_C_LEVEL >= 199506L */

/****************************************************************************
 * Legacy BSD extensions to C99
 ****/
#if __DARWIN_C_LEVEL >= __DARWIN_C_FULL
#define FP_SNAN FP_NAN
#endif

```

Onde está você? Se olhar na barra de navegação na parte superior da área do editor, você verá que está agora no `math.h`.

Clicar em Command é útil sempre que você desejar ver como alguma coisa é definida. Você pode usar esse recurso com constantes, funções, classes, métodos, tipos e muito mais.

Para voltar para o `main.m`, clique no botão ▶ à esquerda da barra de navegação, na parte superior da área do editor. Ou selecione o `main.m` no navegador de projetos.

Você pode estar pensando por que não teve que incluir explicitamente o `math.h` no `main.m` para usar `M_PI`. Quando você criou uma nova ferramenta de linha de comando Foundation, o template importou `Foundation/Foundation.h` para você. `Foundation/Foundation.h` inclui `CoreFoundation/CoreFoundation.h`, que inclui `math.h`.

Diretivas de pré-processador

Compilar um arquivo de código C, C++ ou Objective-C é feito em dois passos. Primeiro, o *pré-processador* executa o arquivo. O resultado do pré-processador vai para o compilador real. As diretivas de pré-processador começam com `#`, e as três mais conhecidas são `#include`, `#import` e `#define`.

#include e #import

`#include` e `#import` fazem essencialmente a mesma coisa: solicitam que o pré-processador leia um arquivo e adicione-o em seu resultado. Geralmente, você inclui um arquivo de declarações (um arquivo `.h`), e essas declarações são usadas pelo compilador para interpretar o código que ele está compilando.

Qual a diferença entre `#include` e `#import`? `#import` garante que o pré-processador inclua um arquivo somente uma vez. `#include` permitirá incluir o mesmo arquivo diversas vezes. Os programadores em C tendem a usar `#include`. Os programadores em Objective-C tendem a usar `#import`.

Ao especificar o nome do arquivo a ser importado, você pode incluir o nome do arquivo entre aspas ou colchetes angulares. As aspas indicam que o cabeçalho está no diretório do seu projeto. Os colchetes angulares

indicam que o cabeçalho está em um dos locais padrão que o pré-processador conhece. (<math.h>, por exemplo, está em /Applications/Xcode46-DP3.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.8.sdk/usr/include/math.h.) Aqui, temos dois exemplos de diretivas #import:

```
// Include the headers I wrote for Pet Store operations
#import "PetStore.h"

// Include the headers for the OpenLDAP libraries
#import <ldap.h>
```

Em um projeto, é comum incluir uma coleção de cabeçalhos em *todos* os arquivos do código. Isso deixa o início de seu arquivo muito desorganizado e torna as compilações mais demoradas. Para facilitar a vida e acelerar a compilação, a maioria dos projetos do Xcode tem um arquivo que lista os cabeçalhos a serem pré-compilados e inclusos em cada arquivo. Em seu projeto Constants, esse arquivo é chamado de Constants-Prefix.pch.

Então, como uma constante de math.h foi inclusa na compilação de main.m? Seu arquivo main.m apresenta a seguinte linha:

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

O arquivo Foundation.h tem essa linha:

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

E o arquivo CoreFoundation.h tem essa linha:

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

#define

#define informa ao pré-processador que “sempre que encontrar A, substitua-o por B antes que o compilador veja.” Observe novamente a linha em math.h:

```
#define M_PI      3.14159265358979323846264338327950288
```

Na diretiva #define, você simplesmente separa as duas partes (o token e seu substituto) com um espaço em branco.

#define pode ser utilizada para fazer algo, assim como uma função. No main.m, exiba o maior de dois números:

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

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

        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%@", MAX(10, 12));

    }
    return 0;
}
```

MAX não é uma função; é uma diretiva #define. A versão C mais básica de MAX é:

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
```

Então, quando o compilador viu a linha que você acabou de adicionar, ela estava assim:

```
NSLog(@"%@", MAX(10, 12));
```

Ao usar #define para trabalhar como uma função em vez de apenas substituir um valor, você está criando uma *macro*.

Variáveis globais

Em vez de usar #define, os programadores em Objective-C geralmente usam variáveis globais para conter valores de constantes.

Vamos adicionar em seu programa para explicar. Primeiro, há uma classe chamada **NSLocale** que armazena informações sobre diferentes localizações geográficas. Você pode obter uma instância da localidade atual do usuário e depois fazer algumas perguntas. Por exemplo, se quisesse saber qual é a moeda na localidade do usuário, poderia perguntar assim:

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

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

        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%@", MAX(10, 12));

        NSLocale *here = [NSLocale currentLocale];
        NSString *currency = [here objectForKey:@"currency"];
        NSLog(@"%@", currency);

    }
    return 0;
}
```

Compile e execute. Dependendo de onde você está, verá algo semelhante a:

```
Money is USD
```

Entretanto, se digitar algo errado, como @"Kuruncy", não obterá nenhum retorno. Para evitar esse problema, o framework Foundation define uma variável global chamada **NSLocaleCurrencyCode**. Ela não é mais fácil de ser digitada, mas se você digitar algo errado, o compilador avisará. Além disso, o preenchimento de código no Xcode funciona adequadamente para uma variável global, mas não para a string @"currency". Altere seu código para usar a constante:

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

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

        NSLog(@"\u03c0 is %f", M_PI);
        NSLog(@"%@", MAX(10, 12));

        NSLocale *here = [NSLocale currentLocale];
        NSString *currency = [here objectForKey:NSLocaleCurrencyCode];
        NSLog(@"%@", currency);

    }
    return 0;
}
```

Quando a classe **NSLocale** foi escrita, essa variável global apareceu em dois lugares. No **NSLocale.h**, a variável foi declarada de modo similar a este:

```
extern NSString * const NSLocaleCurrencyCode;
```

O **const** significa que este ponteiro não mudará por todo o ciclo de vida do programa. O **extern** significa que “eu garanto que isso existe, mas será definido em algum outro arquivo”. No arquivo **NSLocale.m** (que existe em uma câmara na Apple), há certamente uma linha como esta:

```
NSString * const NSLocaleCurrencyCode = @"currency";
```

enum

Geralmente, você precisará definir um conjunto de constantes. Por exemplo, suponha que você estivesse criando um liquidificador com cinco velocidades: misturar, cortar, liquidificar, pulsar e moer gelo. Sua classe **Blender** teria um método chamado **setSpeed:**. Seria melhor se o tipo indicasse que apenas uma das cinco velocidades fosse permitida. Para fazer isso, você teria de definir uma enumeração:

```

enum BlenderSpeed {
    BlenderSpeedStir = 1,
    BlenderSpeedChop = 2,
    BlenderSpeedLiquify = 5,
    BlenderSpeedPulse = 9,
    BlenderSpeedIceCrush = 15
};

@interface Blender : NSObject
{
    // speed must be one of the five speeds
    enum BlenderSpeed speed;
}

// setSpeed: expects one of the five speeds
- (void)setSpeed:(enum BlenderSpeed)x;
@end

```

Os desenvolvedores se cansam de digitar `enum BlenderSpeed`, então, eles geralmente usam `typedef` para criar uma forma abreviada para ele:

```

typedef enum {
    BlenderSpeedStir = 1,
    BlenderSpeedChop = 2,
    BlenderSpeedLiquify = 5,
    BlenderSpeedPulse = 9,
    BlenderSpeedIceCrush = 15
} BlenderSpeed;

@interface Blender : NSObject
{
    // speed must be one of the five speeds
    BlenderSpeed speed;
}

// setSpeed: expects one of the five speeds
- (void)setSpeed:(BlenderSpeed)x;
@end

```

Normalmente, você não se importará com quais números as cinco velocidades representam – somente com o fato de elas serem diferentes umas das outras. Você pode omitir os valores; o compilador irá compor os valores por você:

```

typedef enum {
    BlenderSpeedStir,
    BlenderSpeedChop,
    BlenderSpeedLiquify,
    BlenderSpeedPulse,
    BlenderSpeedIceCrush
} BlenderSpeed;

```

Começando com OS X 10.8 e iOS 6, a Apple apresentou uma nova sintaxe de declaração `enum`: `NS_ENUM()`. Esta é a aparência do seu `enum` ao usar essa sintaxe:

```

typedef NS_ENUM(int, BlenderSpeed) {
    BlenderSpeedStir,
    BlenderSpeedChop,
    BlenderSpeedLiquify,
    BlenderSpeedPulse,
    BlenderSpeedIceCrush
};

```

`NS_ENUM()` é, na verdade, uma macro de pré-processador que adota dois argumentos: um tipo de dados e um nome.

A Apple adotou `NS_ENUM()` para declarações `enum`. A vantagem mais importante de `NS_ENUM()` em relação a outra sintaxe é a capacidade de declarar o tipo integral de dados que o `enum` representará (`short`, `unsigned long` etc.).

Com a sintaxe antiga, o compilador poderá escolher um tipo de dados apropriado para o enum, normalmente `int`. Se seu enum terá apenas quatro opções cujos valores não importam, você não precisa de quatro bytes para armazená-lo; um byte representará tranquilamente os números inteiros até 255. Recordando o Chapter 3 que informava que um `char` é um byte inteiro, você pode declarar um enum que economiza espaço:

```
typedef NS_ENUM(char, BlenderSpeed) {
    BlenderSpeedStir,
    BlenderSpeedChop,
    BlenderSpeedLiquify,
    BlenderSpeedPulse,
    BlenderSpeedIceCrush
};
```

#define vs. variáveis globais

Visto que você pode definir uma constante usando `#define` ou uma variável global (que inclui o uso de `enum`), por que os programadores em Objective-C tendem a usar variáveis globais? Em alguns casos, o uso de variáveis globais proporciona vantagens de desempenho. Por exemplo, você poderá usar `==` em vez de `isEqual:` para comparar strings caso use consistentemente a variável global (uma comparação de ponteiro é mais rápida que o envio de uma mensagem e a varredura de duas strings caractere por caractere). Além disso, é mais fácil trabalhar com variáveis globais quando você está no depurador.

Em geral, você deve usar variáveis globais e `enum` para constantes, e não `#define`.

26

Gravação de arquivos com NSString e NSData

O framework Foundation fornece ao desenvolvedor algumas maneiras fáceis de ler arquivos e gravar neles. Neste capítulo, você tentará usar algumas delas.

Gravação de uma NSString em um arquivo

Primeiramente, vejamos como você obteria o conteúdo de uma **NSString** e o colocaria em um arquivo. Ao gravar uma string em um arquivo, você precisa especificar qual *codificação de string* está usando. Uma codificação de string descreve como cada caractere é armazenado como um array de bytes. ASCII é uma codificação de string que define a letra ‘A’ como sendo armazenada como 01000001. Em UTF-16, a letra ‘A’ é armazenada como 0000000010000001.

O framework Foundation suporta cerca de 20 codificações de string diferentes, mas acabamos por usar o UTF muito mais, já que ele pode lidar com uma enorme variedade de sistemas de gravação. Ele está disponível em dois tipos: UTF-16, que usa dois ou mais bytes para cada caractere, e UTF-8, que usa um byte para os 128 primeiros caracteres ASCII e dois ou mais para os outros caracteres. Para a maioria das finalidades, o UTF-8 é a melhor opção.

Crie um novo projeto: uma Foundation Command Line Tool chamada Stringz. Em **main()**, use os métodos da classe **NSString** para criar uma string e gravá-la no sistema de arquivos:

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

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

        NSMutableString *str = [[NSMutableString alloc] init];
        for (int i = 0; i < 10; i++) {
            [str appendString:@"Aaron is cool!\n"];
        }
        [str writeToFile:@"/tmp/cool.txt"
            atomically:YES
            encoding:NSUTF8StringEncoding
            error:NULL];
        NSLog(@"done writing /tmp/cool.txt");

    }
    return 0;
}
```

Este programa criará um arquivo de texto que pode ser lido e editado em qualquer editor de textos. A string /tmp/cool.txt é conhecida como o caminho de arquivo.

Os caminhos de arquivo podem ser absolutos ou relativos: os caminhos absolutos começam com uma /, que representa o topo do sistema de arquivos, sendo que os caminhos relativos começam no diretório de trabalho do programa. Os caminhos relativos não começam com uma /. Na programação em Objective-C, você verá que quase sempre usamos caminhos absolutos, pois, geralmente, não sabemos qual é o diretório de trabalho do programa.

Compile e execute o programa. (Para localizar o diretório /tmp no Finder, use o item de menu Go → Go to Folder.)

NSError

Como você pode imaginar, todos os tipos de coisas podem dar errado quando você tenta gravar uma string em um arquivo. Por exemplo, o usuário pode não ter acesso de gravação ao diretório em que o arquivo deverá ser gravado. Ou o diretório pode simplesmente não existir. Ou, ainda, o sistema de arquivos pode estar cheio. Para situações como essas, em que pode ser impossível concluir uma operação, o método precisa de um modo para retornar uma descrição do que ocorreu de errado, além do valor booleano de falha ou êxito.

Vimos no Chapter 10 que, quando você precisa de uma função para retornar algo além de seu valor de retorno, você pode usar uma passagem por referência. Você passa à função (ou ao método) uma referência para uma variável em que ela pode armazenar ou manipular diretamente um valor. A referência é o endereço de memória dessa variável.

Para o tratamento de erros, muitos métodos adotam um ponteiro **NSError** por referência. Adicione o tratamento de erros ao Stringz:

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

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

        NSMutableString *str = [[NSMutableString alloc] init];
        for (int i = 0; i < 10; i++) {
            [str appendString:@"Aaron is cool!\n"];
        }

        // Declare a pointer to an NSError object, but do not instantiate it.
        // The NSError instance will only be created if there is, in fact, an error.
        NSError *error;

        // Pass the NSError pointer by reference to the NSString method
        BOOL success = [str writeToFile:@"/tmp/cool.txt"
                               atomically:YES
                               encoding:NSUTF8StringEncoding
                               error:&error];

        // Test the returned BOOL, and query the NSError if the write failed
        if (success) {
            NSLog(@"done writing /tmp/cool.txt");
        } else {
            NSLog(@"writing /tmp/cool.txt failed: %@", [error localizedDescription]);
        }
    }
    return 0;
}
```

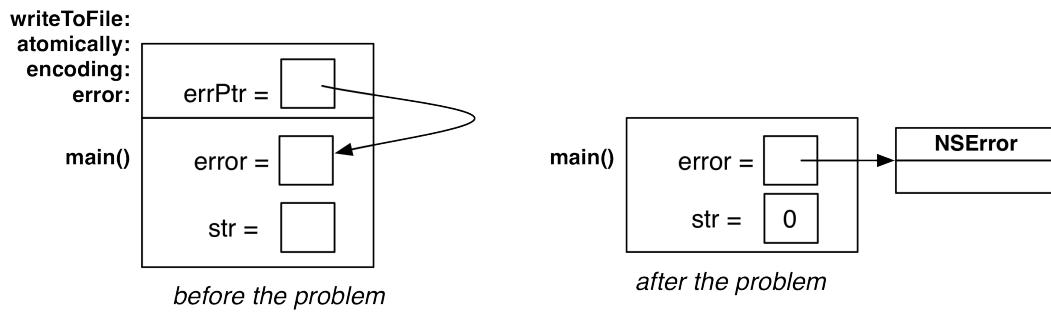
Compile e execute. Agora, altere o código para passar ao método de gravação um caminho de arquivo que não existe, como @"~/too/darned/bad.txt". Você verá uma simpática mensagem de erro.

Observe que você declara um ponteiro para uma instância de **NSError** neste código, mas não cria, ou *instanciaria*, um objeto **NSError** para atribuir a esse ponteiro.

Por que não? Você quer evitar a criação de um objeto de erro desnecessário caso não haja erros. Se houver um erro, **writeToFile:atomically:encoding:error:** será responsável pela criação de uma nova instância de **NSError** e depois pela modificação do ponteiro **error** que você declarou para apontar para o novo objeto de erro. Depois, você pode perguntar a esse objeto o que ocorreu de errado por meio de seu ponteiro **error**.

Essa criação condicional de **NSError** requer que você passe uma referência para **error** (**&error**) visto que não há nenhum objeto ainda para passar. No entanto, diferente da passagem por referência que realizou no Chapter 10, onde você passou a referência de uma variável em C primitiva, aqui você está passando o endereço de uma variável de ponteiro. Na verdade, você está passando o endereço de outro endereço (que pode passar a ser o endereço de um objeto **NSError**).

Figure 26.1 Os erros são passados por referência



Para examinar novamente nossa analogia de espionagem internacional do Chapter 10, você pode dizer a sua espiã: “Se algo sair errado, faça um relatório completo (não muito grande para colocar em um tubo de aço) e esconda-o em um livro na biblioteca. Preciso saber onde você o escondeu, então, coloque o número de referência do livro no tubo.” Ou seja, você está fornecendo à espiã um local onde ela possa colocar o endereço de um relatório de erro que ela criou.

Aqui temos a classe **NSString** em que **writeToFile:atomically:encoding:error:** está declarado:

```
- (BOOL)writeToFile:(NSString *)path
  atomically:(BOOL)useAuxiliaryFile
  encoding:(NSStringEncoding)enc
  error:(NSError **)error
```

Observe o asterisco duplo. Muitos programadores diriam: “O método espera um ponteiro para um ponteiro para uma **NSError**.” No entanto, isso parece mais confuso do que precisa ser. Na nossa opinião, isto é mais descriptivo: “O método espera um endereço onde ele possa colocar um ponteiro para uma instância de **NSError**.”

Os métodos que passam uma **NSError** por referência sempre retornam um valor que indica se ocorreu um erro ou não. Esse método, por exemplo, retorna NO se houver um erro. Não tente acessar **NSError** a menos que o valor de retorno indique que ocorreu um erro; se o objeto **NSError** realmente não existir, a tentativa de acesso causará falhas em seu programa.

Leitura de arquivos com NSString

Ler um arquivo em uma string é muito semelhante a:

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

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

        NSError *error;
        NSString *str = [[NSString alloc] initWithContentsOfFile:@"/etc/resolv.conf"
                                                       encoding:NSUTFStringEncoding
                                                       error:&error];
        if (!str) {
            NSLog(@"read failed: %@", [error localizedDescription]);
        } else {
            NSLog(@"resolv.conf looks like this: %@", str);
        }
    }
    return 0;
}
```

Aqui você está criando uma nova string ao ler o conteúdo de um arquivo como um texto ASCII. Se a leitura falhar (por exemplo, se você não teve permissão para ler o arquivo), então o método retornará `nil`. Nesse caso, você exibe a descrição de localização do erro.

Gravação de um objeto NSData em um arquivo

Um objeto `NSData` representa um buffer de bytes. Por exemplo, se você obtiver alguns dados de um URL, terá uma instância de `NSData`. Então, você pode solicitar que `NSData` grave a si mesma em um arquivo. Crie uma nova Foundation Command Line Tool, chamada `ImageFetch`, que obtenha uma imagem do site do Google para uma instância de `NSData`. Depois, solicite que `NSData` grave seu buffer de bytes em um arquivo:

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

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

        NSURL *url = [NSURL URLWithString:
                      @"http://www.google.com/images/logos/ps_logo2.png"];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        NSError *error = nil;
        NSData *data = [NSURLConnection sendSynchronousRequest:request
                                                returningResponse:NULL
                                                error:&error];

        if (!data) {
            NSLog(@"fetch failed: %@", [error localizedDescription]);
            return 1;
        }

        NSLog(@"The file is %lu bytes", (unsigned long)[data length]);

        BOOL written = [data writeToFile:@"/tmp/google.png"
                               options:0
                               error:&error];

        if (!written) {
            NSLog(@"write failed: %@", [error localizedDescription]);
            return 1;
        }

        NSLog(@"Success!");
    }
    return 0;
}
```

Compile e execute o programa. Abra o arquivo de imagem resultante no Preview.

Observe que o método **writeToFile:options:error:** usa um número que indica as opções a serem usadas no processo de gravação. A opção mais comum é **NSDataWritingAtomic**. Digamos que você já tenha obtido uma imagem e esteja apenas obtendo-a novamente e substituindo-a por uma versão mais recente. Durante a gravação da nova imagem, a energia acaba. Um arquivo gravado pela metade não tem valor algum. Nos casos em que é pior ter um arquivo gravado pela metade do que não ter nenhum arquivo, você poderá fazer com que a gravação seja atômica. Adicione esta opção:

```
NSLog(@"The file is %lu bytes", (unsigned long)[data length]);  
  
BOOL written = [data writeToFile:@"/tmp/google.png"  
                  options:NSDataWritingAtomic  
                  error:&error];  
  
if (!written) {  
    NSLog(@"write failed: %@", [error localizedDescription]);  
    return 1;  
}
```

Agora, os dados serão gravados em um arquivo temporário e, quando a gravação for concluída, o arquivo será renomeado com o nome correto. Dessa maneira, ou você obtém o arquivo inteiro, ou não obtém nada. (Observe que isso não tem nada a ver com as propriedades atomic/nonatomic.)

Leitura de um objeto NSData em um arquivo

Você também pode criar uma instância de **NSData** a partir do conteúdo de um arquivo. Adicione duas linhas em seu programa:

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

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

        NSURL *url = [NSURL URLWithString:
                        @"http://www.google.com/images/logos/ps_logo2.png"];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];
        NSError *error;

        // This method will block until all the data has been fetched
        NSData *data = [NSURLConnection sendSynchronousRequest:request
                                                returningResponse:NULL
                                                error:&error];

        if (!data) {
            NSLog(@"fetch failed: %@", [error localizedDescription]);
            return 1;
        }

        NSLog(@"The file is %lu bytes", (unsigned long)[data length]);

        BOOL written = [data writeToFile:@"/tmp/google.png"
                                options:NSDataWritingAtomic
                                error:&error];

        if (!written) {
            NSLog(@"write failed: %@", [error localizedDescription]);
            return 1;
        }

        NSLog(@"Success!");

        NSData *readData = [NSData dataWithContentsOfFile:@"/tmp/google.png"];
        NSLog(@"The file read from the disk has %lu bytes",
              (unsigned long)[readData length]);
    }

    return 0;
}
```

Compile e execute o programa.

Localização de diretórios especiais

Os usuários esperam que os arquivos sejam salvos em diretórios específicos. Por exemplo, meu navegador, por padrão, baixará os arquivos para /Users/aaron/Downloads/. Para facilitar que o programador faça a coisa certa, a Apple criou uma função que lhe informará os diretórios corretos para o propósito apropriado. Por exemplo, esse pedaço de código vai obter para você o caminho para o diretório da área de trabalho (Desktop) do usuário:

```
// The function returns an array of paths
NSArray *desktops =
    NSSearchPathForDirectoriesInDomains(NSDesktopDirectory, NSUserDomainMask, YES);

// But I know the user has exactly one desktop directory
NSString *desktopPath = desktops[0];
```

Quais outros diretórios especiais se encontram lá? Veja as constantes mais comumente usadas:

- NSApplicationDirectory
- NSLibraryDirectory
- NSUserDirectory
- NSDocumentDirectory

- `NSDesktopDirectory`
- `NSCachesDirectory`
- `NSApplicationSupportDirectory`
- `NSDownloadsDirectory`
- `NSMoviesDirectory`
- `NSMusicDirectory`
- `NSPicturesDirectory`
- `NSTrashDirectory`

27

Callbacks

Até agora, seu código tem sido o chefe. Ele tem enviado mensagens aos objetos padrão do Foundation, como instâncias de **NSString** e **NSArray**, e dizendo a eles o que fazer. Quando seu código finaliza a execução, o programa é finalizado.

Neste capítulo, você vai criar um programa que não apenas inicia, executa e finaliza. Em vez disso, esse programa é *orientado a eventos*. Ele vai iniciar e esperar por um evento. Quando esse evento ocorrer, o programa executará um código em resposta. Esse programa não será finalizado sozinho; ele continuará em operação e esperando pelo próximo evento até que você diga a ele para parar.

Os eventos que podem ocorrer em um dispositivo Mac ou iOS são muitos e variados. Veja alguns exemplos: o usuário clica no mouse ou toca em um botão, passa-se algum período de tempo, o dispositivo fica com pouca memória, o sistema se conecta à rede, o usuário fecha uma janela.

Um *callback* permite que você escreva um pedaço de código e, em seguida, associe esse código a um evento particular. Quando o evento acontece, seu código é executado.

No Objective-C, um callback pode adotar quatro formas:

- *Destino-ação*: Antes de a espera começar, você diz: “Quando esse evento começar, envie essa mensagem a esse objeto.” O objeto que recebe a mensagem é o *destino*. O seletor da mensagem é a *ação*.
- *Objetos auxiliares*: Antes de a espera começar, você diz: “Aqui temos um objeto que adotará uma função que ajuda outro objeto a fazer seu trabalho. Quando um dos eventos relacionados a essa função ocorrer, envie uma mensagem ao objeto auxiliar.” Os objetos auxiliares geralmente são conhecidos como *delegates* ou *fontes de dados*.
- *Notificações*: Há um objeto chamado central de notificações. Quando um evento ocorre, uma notificação associada a esse evento será postada na central de notificações. Antes de a espera começar, você informa à central de notificações “Esse objeto está interessado nesse tipo de notificação. Quando uma notificação for postada, envie essa mensagem ao objeto.”
- *Blocos*: Um bloco é um pedaço (chunk) de código a ser executado. Antes de a espera começar, você diz: “Aqui temos um bloco. Quando esse evento ocorrer, execute esse bloco.”

Neste capítulo, você implementará os três primeiros tipos de callbacks e aprenderá quais deles usar e em quais circunstâncias. Os blocos serão tratados no Chapter 28.

O loop de execução

Em um programa orientado a eventos, deve haver um objeto que realiza a operação e a espera de eventos. No OS X e iOS, esse objeto é uma instância de **NSRunLoop**. Dizemos que quando um evento acontece, o loop de execução faz com que ocorra um callback.

Crie um novo projeto: uma Foundation Command Line Tool chamada **Callbacks**. Primeiramente, você simplesmente obterá um loop de execução e iniciará a execução. Edite o `main.m`:

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        [[NSRunLoop currentRunLoop] run];
    }
    return 0;
}
```

Compile e execute o programa. Observe que o método **run** nunca retorna. O console não reporta o familiar **Program ended with exit code: 0** (Programa finalizado com o código de saída: 0). O loop de execução está esperando que algo aconteça. Escolha **Product → Stop** para parar o programa.

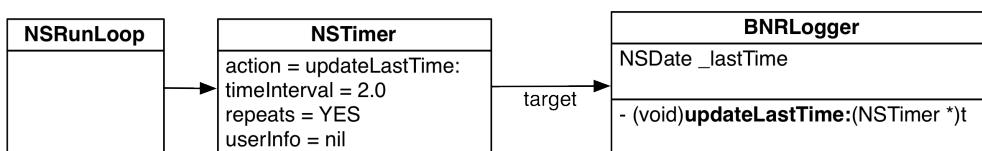
Agora que você tem um loop de execução, pode começar a implementar callbacks.

Destino-ação

Os temporizadores usam um mecanismo destino-ação. Você cria um temporizador com um intervalo de tempo, um destino e uma ação. Após o intervalo ter decorrido, o temporizador envia a mensagem de ação a seu destino.

Você vai adicionar uma instância de **NSTimer** no seu programa. A cada dois segundos, o temporizador enviará a mensagem de ação a seu destino. Você também vai criar uma classe chamada **BNRLogger**. Uma instância dessa classe será o destino do temporizador.

Figure 27.1 **BNRLogger** é o destino do **NSTimer**



Crie um novo arquivo: uma classe do Objective-C chamada **BNRLogger** que é uma subclasse de **NSObject**.

No **BNRLogger.h**, declare uma propriedade que retém uma data, um método que retorna a data como uma string e o método de ação a ser acionado pelo temporizador:

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

@interface BNRLogger : NSObject
@property (nonatomic) NSDate *lastTime;
- (NSString *)lastTimeString;
- (void)updateLastTime:(NSTimer *)t;
@end
```

Os métodos de ação sempre adotam um argumento – o objeto que está enviando a mensagem de ação. Nesse caso, é o objeto temporizador.

No **BNRLogger.m**, implemente os métodos:

```
#import "BNRLogger.h"

@implementation BNRLogger

- (NSString *)lastTimeString
{
    static NSDateFormatter *dateFormatter = nil;
    if (!dateFormatter)
    {
        dateFormatter = [[NSDateFormatter alloc] init];
        [dateFormatter setTimeStyle:NSDateFormatterMediumStyle];
        [dateFormatter setDateStyle:NSDateFormatterMediumStyle];
        NSLog(@"created dateFormatter");
    }
    return [dateFormatter stringFromDate:self.lastTime];
}

- (void)updateLastTime:(NSTimer *)t
{
    NSDate *now = [NSDate date];
    [self setLastTime:now];
    NSLog(@"Just set time to %@", self.lastTimeString);
}

@end
```

Essa pode ser a primeira vez que você viu o modificador `estático` usado desta maneira. Se você tiver milhares de instâncias de `BNRLogger` e todas elas formatarem suas strings da mesma maneira, você vai querer que todas as instâncias de `BNRLogger` compartilhem uma única instância de `NSDateFormatter`. Muitas linguagens orientadas a objetos têm variáveis de classe (em vez de variáveis de instância) para esse tipo de coisa. O Objective-C apenas utiliza variáveis estáticas, que foram discutidas no Chapter 5.

No `main.m`, crie uma instância de `BNRLogger` e torne-a o destino de uma instância de `NSTimer`. Configure a ação para ser `updateLastTime:`:

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

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

        BNRLogger *logger = [[BNRLogger alloc] init];

        NSTimer *timer =
            [NSTimer scheduledTimerWithTimeInterval:2.0
                target:logger
                selector:@selector(updateLastTime:)
                userInfo:nil
                repeats:YES];

        [[NSRunLoop currentRunLoop] run];

    }
    return 0;
}
```

Observe a sintaxe `@selector` que você usou para passar o nome da mensagem de ação para esse método. Isso é exigido para esse argumento; você não pode simplesmente passar o nome do método. Há mais sobre o `@selector` e seletores de passagem no final deste capítulo.

Compile e execute o programa. (Você verá o aviso de uma variável não utilizada. Ignore-o por enquanto.) A instrução de registro com a data e a hora atuais será exibida no console a cada 2 segundos.

Agora, veja o aviso de variável não utilizada do compilador. Ele está dizendo: “Ei, você criou essa variável de `timer`, mas nunca a usou. Isso pode ser um problema.” Em algumas configurações, como esta, isso não é um problema, e você pode sinalizar uma variável como não utilizada intencionalmente para silenciar esses avisos. Isso é feito com o modificador `_unused`.

No `main.m`, sinalize a variável `timer` como não utilizada:

```
_unused NSTimer *timer =
    [NSTimer scheduledTimerWithTimeInterval:2.0
        target:logger
        selector:@selector(updateLastTime:)
        userInfo:nil
        repeats:YES];
```

Compile novamente e o aviso desaparecerá.

Os temporizadores são simples. Eles fazem apenas uma coisa: disparos. Portanto, destino-ação é a melhor opção. Muitos controles simples de interface do usuário, tais como botões e controles deslizantes, usam o mecanismo destino-ação. E algo mais complexo?

Objetos auxiliares

No Chapter 26, você usou uma **NSURLConnection** para obter dados a partir de um servidor web. A conexão foi síncrona – todos os dados foram fornecidos de uma só vez. Tudo funcionou bem, com exceção de dois problemas com uma conexão síncrona:

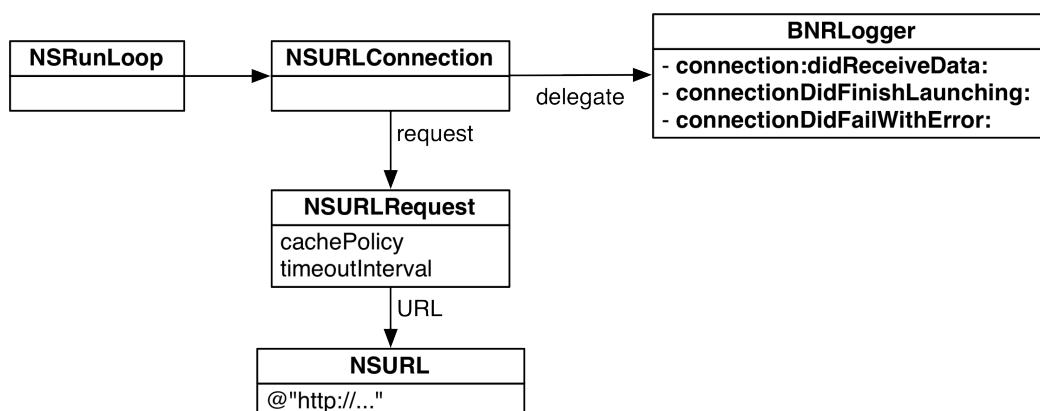
- Ela bloqueia a thread principal enquanto aguarda a chegada de todos os dados. Se você usar esse tipo de conexão em um aplicativo interativo, a interface do usuário não responderá enquanto os dados estiverem sendo obtidos.
- Não há maneira de retornar a chamada se, por exemplo, o servidor web solicitar um nome de usuário e senha.

Por essas razões, é mais comum usar uma **NSURLConnection** de maneira assíncrona. Em uma conexão assíncrona, os dados vêm em pedaços em vez de todos de uma vez. Isso significa que há eventos relacionados a conexões que você precisa estar pronto para responder. Alguns exemplos de eventos relacionados a conexões são pedaços de dados que chegam, o servidor web que requer credenciais e a conexão que falha.

Para gerenciar essa conexão mais complexa, você deve atribuir a ela um objeto auxiliar. No objeto auxiliar, você implementa os métodos a serem executados em resposta a diferentes eventos relacionados a conexões.

No seu programa Callbacks, você vai usar uma **NSURLConnection** assíncrona para obter dados de um website. A instância de **BNRLogger** servirá como o objeto auxiliar da **NSURLConnection**. Mais especificamente, a **BNRLogger** será o delegate da **NSURLConnection**.

Figure 27.2 **BNRLogger** é o delegate da **NSURLConnection**



No `main.m`, crie uma `NSURL` e uma `NSURLRequest` como você fez no Chapter 26. Depois, crie uma `NSURLConnection` e defina a instância de `BNRLogger` para que seja seu delegate:

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

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

        BNRLogger *logger = [[BNRLogger alloc] init];

        NSURL *url = [NSURL URLWithString:
                      @"http://www.gutenberg.org/cache/epub/205/pg205.txt"];

        NSURLRequest *request = [NSURLRequest requestWithURL:url];

        __unused NSURLConnection *fetchConn =
            [[NSURLConnection alloc] initWithRequest:request
                                         delegate:logger
                                         startImmediately:YES];

        __unused NSTimer *timer =
            [NSTimer scheduledTimerWithTimeInterval:2.0
                                         target:logger
                                         selector:@selector(updateLastTime:)
                                         userInfo:nil
                                         repeats:YES];

        [[NSRunLoop currentRunLoop] run];

    }
    return 0;
}
```

Agora, na `BNRLogger`, você precisa implementar os métodos de callback – os métodos a serem executados em resposta a eventos específicos.

Você não propõe ou declara esses métodos você mesmo. Eles já foram declarados em um *protocolo*. Um protocolo é uma lista de declarações de métodos. Você aprenderá mais sobre protocolos no Chapter 29, mas por agora, pense em um protocolo como um conjunto predeterminado de mensagens que um objeto pode enviar para seu objeto auxiliar.

No `BNRLogger.h`, declare que `BNRLogger` vai implementar métodos a partir dos protocolos `NSURLConnectionDelegate` e `NSURLConnectionDataDelegate`:

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

@interface BNRLogger : NSObject
    <NSURLConnectionDelegate, NSURLConnectionDataDelegate>

@property (nonatomic) NSDate *lastTime;
- (NSString *)lastTimeString;
- (void)updateLastTime:(NSTimer *)t;
@end
```

Há três mensagens às quais **BNRLogger** precisará responder como delegate da **NSURLConnection**. Duas são provenientes do protocolo **NSURLConnectionDataDelegate**:

- **(void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data;**
- **(void)connectionDidFinishLoading:(NSURLConnection *)connection;**

A outra é do protocolo **NSURLConnectionDelegate**:

- **(void)connection:(NSURLConnection *)connection didFailWithError:(NSError *)error;**

(Como você sabe quais métodos um protocolo tem e quais você precisa implementar? Consulte a documentação do desenvolvedor. Os protocolos têm referências, similares a referências de classes, com informações sobre seus métodos. Você também aprenderá mais sobre protocolos e seus métodos no Chapter 29.)

Antes de obter a implementação desses métodos, **BNRLogger** precisa de uma variável de instância. Quando você criou uma **NSURLConnection** síncrona no Chapter 26, você usou uma instância de **NSData** para reter os bytes provenientes do servidor. Em uma conexão assíncrona, você precisa de uma instância de **NSMutableData**. À medida que os pedaços de dados chegarem, você os adicionará a esse objeto.

No **BNRLogger.h**, adicione uma variável de instância de **NSMutableData**:

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

@interface BNRLogger : NSObject
    <NSURLConnectionDelegate, NSURLConnectionDataDelegate>
{
    NSMutableData *_incomingData;
}
@property (nonatomic) NSDate *lastTime;
- (NSString *)lastTimeString;
- (void)updateLastTime:(NSTimer *)t;
@end
```

No **BNRLogger.m**, implemente os três métodos de protocolo:

```
#import "BNRLogger.h"

@implementation BNRLogger

...

// Called each time a chunk of data arrives
- (void)connection:(NSURLConnection *)connection
    didReceiveData:(NSData *)data
{
    NSLog(@"received %lu bytes", [data length]);

    // Create a mutable data if it does not already exist
    if (!_incomingData) {
        _incomingData = [[NSMutableData alloc] init];
    }

    [_incomingData appendData:data];
}

// Called when the last chunk has been processed
- (void)connectionDidFinishLoading:(NSURLConnection *)connection
{
    NSLog(@"Got it all!");
    NSString *string = [[NSString alloc] initWithData:_incomingData
                                                encoding:NSUTF8StringEncoding];
    _incomingData = nil;
    NSLog(@"string has %lu characters", [string length]);

    // Uncomment the next line to see the entire fetched file
    // NSLog(@"The whole string is %@", string);
}

// Called if the fetch fails
- (void)connection:(NSURLConnection *)connection
    didFailWithError:(NSError *)error
{
    NSLog(@"connection failed: %@", [error localizedDescription]);
    _incomingData = nil;
}

@end
```

Compile e execute o programa. Você verá os dados chegando da web em pedaços (chunks). Por fim, a **BNRLogger** será informada de que a obtenção está concluída.

Estas são as regras, até agora, para os callbacks: Ao enviar um callback para um objeto, a Apple usa o mecanismo de destino-ação. Ao enviar uma série de callbacks para um objeto, a Apple usa um objeto auxiliar com um protocolo. Esses objetos auxiliares geralmente são chamados de delegates ou fontes de dados.

E se for preciso que o callback vá para diversos objetos?

Notificações

Imagine que o usuário altere o fuso horário em um Mac. Muitos objetos em seu programa podem querer saber se esse evento aconteceu. Cada um deles pode registrar-se na central de notificações como um observador. Quando o fuso horário for alterado, a notificação `NSSystemTimeZoneDidChangeNotification` será enviada à central, e esta encaminhará essa mensagem a todos os observadores relevantes.

No `main.m`, registre a instância de **BNRLogger** para receber uma notificação quando o fuso horário for alterado:

```

#import <Foundation/Foundation.h>
#import "BNRLogger.h"

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

        BNRLogger *logger = [[BNRLogger alloc] init];

        [[NSNotificationCenter defaultCenter]
            addObserver:logger
            selector:@selector(zoneChange:)
            name:NSSystemTimeZoneDidChangeNotification
            object:nil];

        NSURL *url = [NSURL URLWithString:
                      @"http://www.gutenberg.org/cache/epub/205/pg205.txt"];
        NSURLRequest *request = [NSURLRequest requestWithURL:url];

        __unused NSURLConnection *fetchConn =
            [[NSURLConnection alloc] initWithRequest:request
                                         delegate:logger
                                         startImmediately:YES];

        __unused NSTimer *timer =
            [NSTimer scheduledTimerWithTimeInterval:2.0
                                         target:logger
                                         selector:@selector(updateLastTime:)
                                         userInfo:nil
                                         repeats:YES];

        [[NSRunLoop currentRunLoop] run];
    }
    return 0;
}

```

Agora, implemente o método que será chamado no `BNRLogger.m`:

```

- (void)zoneChange:(NSNotification *)note
{
    NSLog(@"The system time zone has changed!");
}

```

Compile e execute o programa. Durante a execução, abra **System Preferences** e altere o fuso horário de seu sistema. Você verá que seu método `zoneChange:` será chamado. (Em alguns sistemas, parece que ele é chamado duas vezes. Isso não é motivo de preocupação.)

Muitas das classes que a Apple escreveu postam notificações quando coisas interessantes acontecem. Você pode descobrir quais notificações uma classe posta em sua referência na documentação do desenvolvedor.

Quando você se registra como um observador na central de notificações, pode especificar o nome da notificação (por exemplo, `NSWindowDidResizeNotification`) e sobre quais postagens dessa notificação você mais se interessa (“Quero apenas ser informado sobre a notificação de redimensionamento desta janela específica.”). Para cada um desses parâmetros, você pode fornecer um `nil` que funciona como o curinga. Se você fornecer um `nil` para ambos, receberá cada notificação postada por cada objeto em seu programa. Em um aplicativo para desktop, isso significa *muitas* notificações.

Qual usar?

Neste capítulo, você viu três tipos de callbacks. Como a Apple decidiu qual deles usar em cada situação específica?

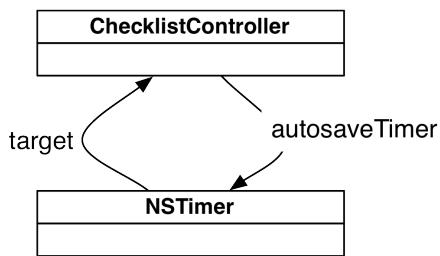
- Os objetos que fazem apenas uma coisa (como `NSTimer`) usam o callback de destino-ação.
- Os objetos que têm uma existência mais complicada (como uma `NSURLConnection`) usam objetos auxiliares; o tipo mais comum de objeto auxiliar é o delegate.

- Os objetos que talvez precisem disparar callbacks em diversos outros objetos (como uma **NSTimeZone**) usam notificações.

Callbacks e propriedade de objetos

O risco de ciclos de referências fortes é inerente em todos esses esquemas de callback. Frequentemente, o objeto que você cria tem um ponteiro para o objeto que vai retornar a chamada. E ele tem um ponteiro para o objeto que você criou. Se cada um deles tiver referências fortes entre si, você terminará com um ciclo de referência forte – nenhum deles jamais será desalocado.

Figure 27.3 Ciclo de referência forte



Portanto, decidiu-se que:

- As centrais de notificações não são proprietárias de seus observadores. Se um objeto for um observador, tipicamente ele se removerá da central de notificações em seu método **dealloc**:

```

- (void)dealloc
{
    [[NSNotificationCenter defaultCenter] removeObserver:self];
}
  
```

- Os objetos não são proprietários de seus delegates ou suas fontes de dados. Se criar um objeto que seja um delegate ou uma fonte de dados, seu objeto deverá “liberar-se” em seu método **dealloc**:

```

- (void)dealloc
{
    [windowThatBossesMeAround setDelegate:nil];
    [tableViewThatBegsForData setDataSource:nil];
}
  
```

- Os objetos não são proprietários de seus destinos. Se você criar um objeto que seja um destino, ele deverá zerar o ponteiro do destino em seu método **dealloc**:

```

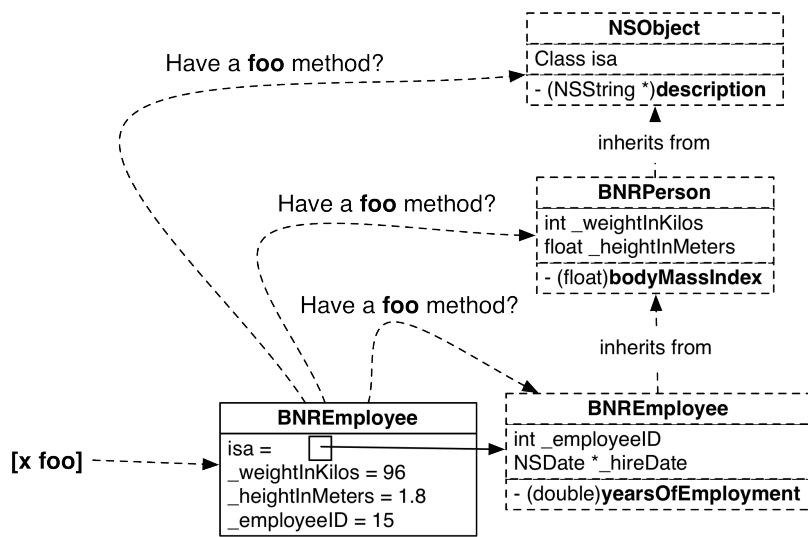
- (void)dealloc
{
    [buttonThatKeepsSendingMeMessages setTarget:nil];
}
  
```

Nenhum desses problemas existe neste programa, pois seu objeto **BNRLLogger** não será desalocado antes do encerramento do programa. (Além disso, por um feliz acaso, neste exercício você acabou usando duas exceções bem documentadas para as regras: uma **NSURLConnection** realmente possui seu delegate enquanto a conexão está sendo executada, e uma **NSTimer** realmente possui seu destino enquanto o temporizador é válido.)

Para os mais curiosos: como os seletores funcionam

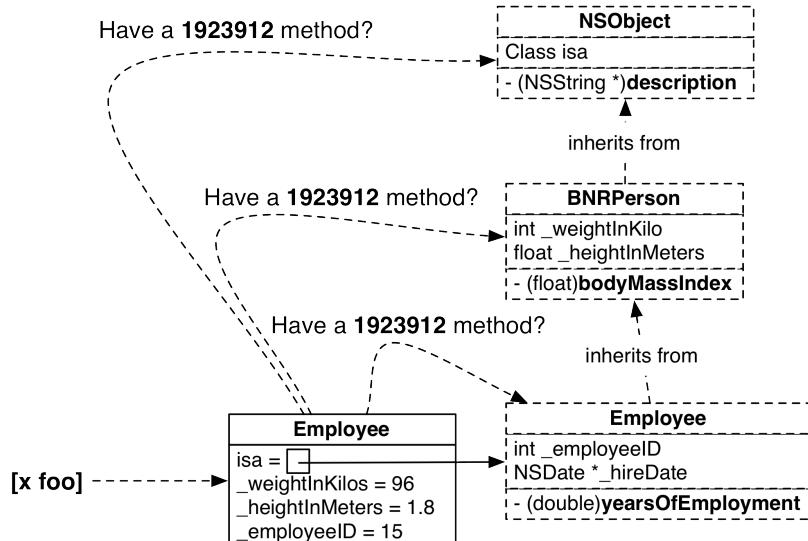
Você aprendeu no Chapter 20 que quando envia uma mensagem a um objeto, é perguntado à classe do objeto se ela contém um método com esse nome. A pesquisa processa a hierarquia de herança até que uma classe responda: “Sim, tenho um método com esse nome.”.

Figure 27.4 Pesquisa de um método com o nome correto



Como você pode imaginar, esta pesquisa precisa ocorrer extremamente rápido. Se o compilador usasse o nome real do método (que poderia ser muito longo), a pesquisa do método seria realmente lenta. Para agilizar, o compilador atribui um número exclusivo ao nome de cada método que ele encontra. No tempo de execução, ele usa esse número em vez do nome do método.

Figure 27.5 Como ele realmente funciona



Assim, um seletor é um número exclusivo que representa um nome do método específico. Quando um método espera um seletor como um argumento (como `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` faz), ele está esperando esse número. Você usa a diretiva do compilador `@selector` para informar ao compilador para procurar o seletor com o nome do método determinado.

28

Blocos

Um *bloco* é um pedaço (chunk) de código. Aqui temos um bloco:

```
^{
    NSLog(@"This is an instruction within a block.");
}
```

Parece uma função C; é um conjunto de instruções dentro de chaves. Não tem, no entanto, um nome. Em vez disso, o circunflexo (^) identifica este bit de código como um bloco.

Como uma função, um bloco pode usar argumentos e valores de retorno. Aqui temos outro bloco:

```
^(double dividend, double divisor) {
    double quotient = dividend / divisor;
    return quotient;
}
```

Este bloco usa duas `doubles` como argumentos e retorna uma `double`.

Você pode transformar um bloco como um argumento em um método que aceita um bloco. Muitas das classes da Apple têm métodos que aceitam blocos como argumentos.

Por exemplo, `NSArray`, `NSDictionary` e `NSSet` permitem enumeração baseada em blocos: cada classe tem no mínimo um método que aceita um bloco. Quando um desses métodos é chamado, ele executará o código dentro do bloco passado uma vez para cada objeto na coleção. Neste capítulo, você vai utilizar o método `enumerateObjectsUsingBlock:` de `NSArray`.

(Se você tem conhecimento de outra linguagem de programação, talvez conheça os blocos como funções anônimas, fechamentos ou lambdas. Se estiver familiarizado com os ponteiros de funções, os blocos podem parecer similares, mas os blocos permitem a criação de códigos apropriados que podem ser escritos com ponteiros de funções.)

Crie uma nova Foundation Command Line Tool e nomeie como `VowelMovement`. Esse programa vai fazer iteração pelo array de strings, remover as vogais de cada string e armazenar as strings “devowelized” (sem as vogais) em um novo array.

No `main.m`, configure três arrays: um para as strings originais, outro para as strings sem vogais e um terceiro para uma lista de vogais.

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

        // Create array of strings and a container for devowelized ones
        NSArray *originalStrings = @[@"Sauerkraut", @"Raygun",
                                      @"Big Nerd Ranch", @"Mississippi"];

        NSLog(@"original strings: %@", originalStrings);

        NSMutableArray *devowelizedStrings = [NSMutableArray array];

        // Create a list of characters to be removed from the string
        NSArray *vowels = @[@"a", @"e", @"i", @"o", @"u"];

    }
    return 0;
}
```

Nada novo até aqui; você está apenas configurando arrays. Por enquanto, compile seu programa e ignore os avisos sobre as variáveis não utilizadas.

Utilização de blocos

Logo você vai criar seu primeiro bloco. Esse bloco vai fazer uma cópia de uma determinada string, remover as vogais da string copiada e, em seguida, adicionar essa string no array `devowelizedStrings`.

Você vai enviar o array `originalStrings` com a mensagem `enumerateObjectsUsingBlock:` com seu bloco sem vogais como seu argumento. Mas antes, há mais um pouco de sintaxe de blocos a ser aprendido.

Declaração de uma variável de bloco

Um bloco pode ser armazenado em uma variável. No `main.m`, insira a seguinte declaração de variável de bloco.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        // Create array of strings and a container for devowelized ones
        NSArray *originalStrings = @[@"Sauerkraut", @"Raygun",
                                      @"Big Nerd Ranch", @"Mississippi"];

        NSLog(@"original strings: %@", originalStrings);

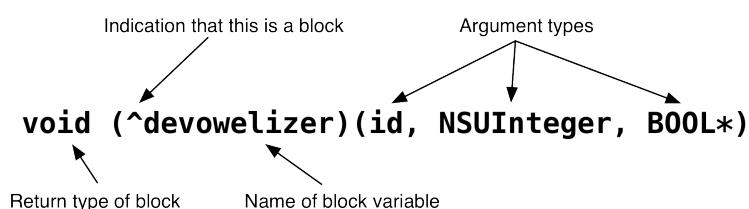
        NSMutableArray *devowelizedStrings = [NSMutableArray array];

        // Create a list of characters to be removed from the string
        NSArray *vowels = @[@"a", @"e", @"i", @"o", @"u"];

        // Declare the block variable
        void (^devowelizer)(id, NSUInteger, BOOL *);
    }
    return 0;
}
```

Vamos dividir esta declaração. O nome da variável de bloco (`devowelizer`) está em um conjunto de parênteses logo após o circunflexo. A declaração inclui o tipo de retorno do bloco (`void`) e os tipos de seus argumentos (`id`, `NSUInteger`, `BOOL *`), assim como em uma declaração de função.

Figure 28.1 Declaração da variável de bloco



Qual é o tipo dessa variável de bloco? Ela não é simplesmente um “bloco”. Seu tipo é “um bloco que usa um objeto, um inteiro e um ponteiro `BOOL` e que não retorna nada”. Esse é o tipo de bloco que `enumerateObjectsUsingBlock:` espera. Você logo vai saber para que cada um desses argumentos é usado.

Composição de um bloco

Agora você precisa compor um bloco do tipo declarado e atribui-lo à nova variável. No `main.m`, componha um bloco que realize uma cópia mutável da string original, remova suas vogais, e, em seguida, adicione a nova string ao array de strings sem vogais e o atribua à `devowelizer`:

```

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

        // Declare the block variable
        void (^devowelizer)(id, NSUInteger, BOOL *);

        // Compose a block and assign it to the variable
        devowelizer = ^(id string, NSUInteger i, BOOL *stop) {

            NSMutableString *newString = [NSMutableString stringWithString:string];

            // Iterate over the array of vowels, replacing occurrences of each
            // with an empty string
            for (NSString *s in vowels) {
                NSRange fullRange = NSMakeRange(0, [newString length]);
                [newString replaceOccurrencesOfString:s
                                              withString:@"""
                                              options:NSCaseInsensitiveSearch
                                              range:fullRange];
            }

            [devowelizedStrings addObject:newString];
        }; // End of block assignment
    }
    return 0;
}

```

Observe que a atribuição do bloco termina com um ponto e vírgula, exatamente como ocorreria com qualquer atribuição de variável. Compile seu programa para verificar sua digitação. Os avisos sobre as variáveis não utilizadas desaparecerão.

Assim como com qualquer variável, você pode fazer a declaração e a atribuição de devowelizer em uma ou duas etapas. Veja como ficaria em uma etapa:

```

void (^devowelizer)(id, NSUInteger, BOOL *) = ^(id string, NSUInteger i, BOOL *stop) {

    NSMutableString *newString = [NSMutableString stringWithString:string];

    // Iterate over the array of vowels, replacing occurrences of each
    // with an empty string.
    for (NSString *s in vowels) {
        NSRange fullRange = NSMakeRange(0, [newString length]);
        [newString replaceOccurrencesOfString:s
                                      withString:@"""
                                      options:NSCaseInsensitiveSearch
                                      range:fullRange];
    }

    [devowelizedStrings addObject:newString];
};

```

Passagem em um bloco

No `main.m`, envie a mensagem `enumerateObjectsUsingBlock:` com a variável `devowelizer` ao array de strings originais e, em seguida, exiba as strings sem vogais.

```
int main (int argc, const char * argv[])
{
    @autoreleasepool {
        ...
        // Declare the block variable
        void (^devowelizer)(id, NSUInteger, BOOL *);

        // Assign a block to the variable
        devowelizer = ^(id string, NSUInteger i, BOOL *stop) {
            NSMutableString *newString = [NSMutableString stringWithString:string];
            // Iterate over the array of vowels, replacing occurrences of each
            // with an empty string.
            for (NSString *s in vowels) {
                NSRange fullRange = NSMakeRange(0, [newString length]);
                [newString replaceOccurrencesOfString:s
                                              withString:@""'
                                              options:NSCaseInsensitiveSearch
                                              range:fullRange];
            }
            [devowelizedStrings addObject:newString];
        }; // End of block assignment

        // Iterate over the array with your block
        [originalStrings enumerateObjectsUsingBlock:devowelizer];
        NSLog(@"devowelized strings: %@", devowelizedStrings);
    }
    return 0;
}
```

Compile e execute o seu programa. Você verá dois arrays registrados no console.

```
2011-09-03 10:27:02.617 VowelMovement[787:707] original strings: (
    Sauerkraut,
    Raygun,
    "Big Nerd Ranch",
    Mississippi
)
2011-09-03 10:27:02.618 VowelMovement[787:707] new strings: (
    Srkrt,
    Rygn,
    "Bg Nrd Rnch",
    Msssspp
)
```

Os três argumentos desse tipo de bloco são especificamente projetados para fazer iteração em um array. O primeiro é um ponteiro para o objeto atual. Observe que o tipo desse ponteiro é `id`, de modo que ele funcionará independentemente dos tipos de objetos contidos no array. O segundo argumento é um `NSUInteger`, que é o índice do objeto atual. O terceiro argumento é um ponteiro para `BOOL`, que, por padrão, é `NO`. Mudando-o para `YES`, o bloco cessará de ser executado após a iteração atual.

Modifique seu bloco para verificar se há um caractere ‘y’ em maiúsculo ou minúsculo. Se houver um, defina o ponteiro para `YES` (o que evitaria que o bloco realize mais iterações) e finalize a iteração atual.

```

devowelizer = ^(id string, NSUInteger i, BOOL *stop){

    NSRange yRange = [string rangeOfString:@"y"
                                    options:NSCaseInsensitiveSearch];

    // Did I find a y?
    if (yRange.location != NSNotFound) {
        *stop = YES; // Prevent further iterations
        return;      // End this iteration
    }

    NSMutableString *newString = [NSMutableString stringWithString:string];

    // Iterate over the array of vowels, replacing occurrences of each
    // with an empty string.
    for (NSString *s in vowels) {
        NSRange fullRange = NSMakeRange(0, [newString length]);
        [newString replaceOccurrencesOfString:s
                                       withString:@"""
                                       options:NSCaseInsensitiveSearch
                                       range:fullRange];
    }

    [devowelizedStrings addObject:newString];
};

// End of block assignment

```

Compile e execute o programa. Novamente, dois arrays são registrados no resultado do depurador, mas, dessa vez, a enumeração do array foi cancelada durante a segunda iteração quando o bloco encontrou uma palavra com a letra ‘y’ nela. Tudo o que você obtém é `Srkrt`.

typedef

A sintaxe de bloco pode ser confusa, mas você pode torná-la mais clara usando a palavra-chave `typedef` da qual você tomou conhecimento no Chapter 11. Lembre-se de que `typedefs` ficam na parte superior do arquivo ou em um cabeçalho, fora das implementações de quaisquer métodos. No `main.m`, adicione a seguinte linha de código:

```

#import <Foundation/Foundation.h>

typedef void (^ArrayEnumerationBlock)(id, NSUInteger, BOOL *);

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

```

Observe que é idêntica à declaração de uma variável de bloco. Entretanto, você está aqui definindo um tipo e não uma variável, então, coloque o nome de um tipo apropriado próximo ao circunflexo. Com isso, você pode simplificar as declarações de blocos similares.

Agora, você pode declarar `devowelizer` usando o seu novo tipo:

```

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

        ...

        // Declare the block variable
        void (^devowelizer)(id, NSUInteger, BOOL *);
        ArrayEnumerationBlock devowelizer;

        // Compose and assign a block to the variable
        devowelizer = ^(id string, NSUInteger i, BOOL *stop) {
            ...

```

Observe que, sozinho, o tipo de bloco define somente os tipos de retorno e os argumentos do bloco; ele não tem relação com o conjunto de instruções dentro de um bloco desse tipo.

Blocos vs. outros callbacks

No Chapter 27, você aprendeu sobre notificações e objetos auxiliares dos mecanismos de callback. Os callbacks permitem que outros objetos chamem métodos em seu objeto como resposta a eventos. Embora sejam perfeitamente funcionais, essas abordagens dividem seu código. Geralmente, partes de seu programa que você gostaria que ficassem próximas para fins de clareza não estão dessa forma.

O programa `Callbacks` inclui o seguinte código que retorna a chamada ao método `zoneChange::`:

```
[[NSNotificationCenter defaultCenter]
    addObserver:logger
        selector:@selector(zoneChange:)
            name:NSSystemTimeZoneDidChangeNotification
            object:nil];
```

É natural, portanto, que alguém lendo seu código pense, “O que esse método `zoneChange::` faz?” Para responder a essa pergunta, o programador deve se voltar à implementação do `zoneChange::`, que pode estar a muitas linhas de distância.

Os blocos, por outro lado, mantém o código de modo a ser disparado por um evento próximo. Por exemplo, `NSNotificationCenter` tem um método `addObserverForName:object:queue:usingBlock:`. Esse método é similar ao `addObserver:selector:name:object:`, mas aceita um bloco em vez de um seletor. Esse bloco pode ser definido próximo à chamada ao `addObserverForName:object:queue:usingBlock:`. Assim, seu amigo programador curioso pode ver o que o seu código faz do começo ao fim, a partir de um só lugar.

Você terá a chance de fazer essa mudança para o programa `Callbacks` no segundo desafio no final deste capítulo.

Mais sobre os blocos

Veja algumas outras coisas que você pode fazer com blocos.

Valores de retorno

O bloco que você criou para o `VowelMovement` não tem um valor de retorno, mas muitos blocos terão. Quando um bloco retorna um valor, você pode obter o valor de retorno ao chamar a variável de bloco como uma função.

Observe novamente um dos blocos de amostra que você viu no começo do capítulo.

```
^(double dividend, double divisor) {
    double quotient = dividend / divisor;
    return quotient;
}
```

Esse bloco usa duas doubles e retorna uma. Para armazenar esse bloco em uma variável, você teria que declarar uma variável desse tipo e atribuir o bloco a ela:

```
// Declare divBlock variable
double (^divBlock)(double,double);

// Assign block to variable
divBlock = ^(double dividend, double divisor) {
    double quotient = dividend / divisor;
    return quotient;
}
```

Você pode então chamar o `divBlock` como uma função para obter seu valor de retorno:

```
double myQuotient = divBlock(42.0, 12.5);
```

Blocos anônimos

Um *bloco anônimo* é um bloco que você passa diretamente para um método sem antes atribui-lo a uma variável de bloco.

Vamos primeiramente considerar o caso de um inteiro anônimo. Ao passar um inteiro para um método, você tem três opções:

```
// Option 1: Totally break it down
int i;
i = 5;
NSNumber *num = [NSNumber numberWithInt:i];

// Option 2: Declare and assign on one line
int i = 5;
NSNumber *num = [NSNumber numberWithInt:i];

// Option 3: Skip the variable entirely
NSNumber *num = [NSNumber numberWithInt:5];
```

Se escolher a terceira opção, você passará o inteiro de forma anônima. Ele fica anônimo porque não possui um nome (ou uma variável) associado a ele.

Você tem as mesmas opções quando deseja passar um bloco para um método. Atualmente, seu código coloca a declaração de bloco, sua atribuição e uso em três linhas separadas do código. Mas é mais comum passar blocos anonimamente. O primeiro desafio no final deste capítulo é modificar o programa `VowelMovement` para usar um bloco anônimo.

Variáveis externas

Geralmente, um bloco usa outras variáveis (primitivas e ponteiros para objetos) que foram criadas fora do bloco. Elas são chamadas de *variáveis externas*. Para garantir que elas estarão disponíveis durante o período que o bloco precisar delas, essas variáveis são *capturadas* pelo bloco.

Para variáveis primitivas, esses valores são copiados e armazenados como variáveis locais dentro do bloco. Para ponteiros, o próprio bloco manterá uma referência forte aos objetos que ele referencia. Isso significa que há a garantia de que quaisquer objetos referenciados pelo bloco estarão ativos, enquanto o bloco estiver. (Se você queria saber a diferença entre blocos e ponteiros de função, ela está bem aqui. Veremos um ponteiro de função fazer isso!)

Usar self em blocos

Se precisar escrever um bloco que usa `self`, você precisa realizar algumas etapas adicionais para evitar um ciclo de referência forte. Considere um exemplo onde uma instância de `BNREmployee` cria um bloco que vai registrar a instância de `BNREmployee` cada vez que ela executa:

```
myBlock = ^{
    NSLog(@"%@", self);
};
```

A instância de `BNREmployee` tem um ponteiro para um bloco (`myBlock`). O bloco captura a variável `self`, de modo a ter um ponteiro voltado para a instância de `BNREmployee`. Você tem um ciclo de referência forte.

Para quebrar o ciclo de referência forte, você declara um ponteiro `_weak` fora do bloco que aponta para `self`. Em seguida, você pode usar esse ponteiro dentro do bloco em vez de `self`:

```
_weak BNREmployee *weakSelf = self; // a weak reference
myBlock = ^{
    NSLog(@"%@", weakSelf);
};
```

A referência do bloco para a instância de `BNREmployee` agora é fraca, e o ciclo de referência forte está quebrado.

No entanto, visto que a referência é fraca, o objeto para o qual `self` aponta pode ser desalocado enquanto o bloco está executando.

Você pode eliminar esse risco ao criar uma referência local forte para `self` dentro do bloco:

```
_weak BNREmployee *weakSelf = self; // a weak reference
myBlock = ^{
    BNREmployee *innerSelf = weakSelf; // a block-local strong reference
    NSLog(@"%@", innerSelf);
};
```

Ao criar a referência `innerSelf` forte, você criou novamente um ciclo de referência forte entre o bloco e a instância de `BNREmployee`. Mas visto que a referência `innerSelf` é local ao escopo do bloco, o ciclo de

referência forte só existirá enquanto o bloco estiver executando e será quebrado automaticamente quando o bloco for finalizado.

Essa é uma boa prática de programação a ser empregada sempre que você escrever um bloco que precise fazer referência a `self`.

Utilização de `self` inesperadamente em blocos

Se você utilizar uma variável de instância diretamente dentro de um bloco, o bloco vai capturar `self` em vez da variável de instância. Isso se deve a uma nuança pouco conhecida de variáveis de instância. Considere esse código que acessa uma variável de instância diretamente:

```
_weak BNREmployee *weakSelf = self;
myBlock = ^{
    BNREmployee *innerSelf = weakSelf; // a block-local strong reference
    NSLog(@"%@", innerSelf);
    NSLog(@"Employee ID: %d", _employeeID);
};
```

O compilador interpreta o acesso da variável direta da seguinte forma:

```
_weak BNREmployee *weakSelf = self;
myBlock = ^{
    BNREmployee *innerSelf = weakSelf; // a block-local strong reference
    NSLog(@"%@", innerSelf);
    NSLog(@"Employee ID: %d", self->_employeeID);
};
```

A sintaxe `->` parece familiar? É a sintaxe para acessar o membro de uma struct no heap. Em seus núcleos mais profundos e escuros, os objetos são, na verdade, structs.

Já que o compilador lê o `_employeeID` como `self->_employeeID`, `self` é inesperadamente capturado pelo bloco. Isso causará o mesmo ciclo de referência forte que você evitou com o uso de `weakSelf` e `innerSelf`.

A solução? Não acesse as variáveis de instância diretamente. Use seus acessores!

```
_weak BNREmployee *weakSelf = self; myBlock = ^{
    BNREmployee *innerSelf = weakSelf; // a block-local strong reference
}
```

Agora não há uso direto de `self`, assim não há nenhum ciclo de referência forte desintencional. Problema resolvido.

Nessa situação, é importante entender o que o compilador está pensando para evitar o ciclo de referência forte ocultado. No entanto, você nunca deve usar a sintaxe `->` para acessar variáveis de instância de um objeto em seu código. Fazer isso é muito perigoso e vai além do âmbito deste livro. Acessores são seus amigos e você deve usá-los.

Modificação de variáveis externas

Por padrão, as variáveis capturadas por um bloco são constantes dentro do bloco e você não pode alterar seus valores. Se deseja modificar uma variável externa dentro de um bloco, você deve declarar a variável externa usando a palavra-chave `__block`.

Por exemplo, no código a seguir, você aumenta a variável externa `counter` dentro do bloco.

```
__block int counter = 0;
void (^counterBlock)() = ^{
    ...
    counter++; // Increments counter to 1
    counter++; // Increments counter to 2
}
```

Sem a palavra-chave `__block`, você obteria um erro do compilador.

Desafio: um bloco anônimo

Modifique o exercício deste capítulo para passar o bloco anonimamente como um argumento para `enumerateObjectsUsingBlock:`. Ou seja, mantenha o bloco, mas descarte a variável `devowelizer`.

Desafio: usando um bloco com o NSNotificationCenter

No Chapter 27, você usou o método `NSNotificationCenter` de `addObserver:selector:name:object:` para registrar o recebimento de callbacks por meio de seu método `zoneChange:`. Atualize esse exercício para usar o método `addObserverForName:object:queue:usingBlock:`, em vez disso.

Esse método utiliza um bloco como um argumento e depois executa o bloco em vez de realizar o callback para seu objeto quando a notificação especificada é enviada. Isso significa que seu método `zoneChange:` nunca será chamado. O código que estava dentro desse método estará, em vez disso, no bloco.

O bloco passado deverá usar um único argumento (uma `NSNotification *`) e não retornar nada, assim como faz o método `zoneChange:`.

Passe o `nil` como o argumento para `queue::`; esse argumento é usado para simultaneidade – um assunto que não abordaremos neste livro.

Para obter mais detalhes importantes sobre o `addObserverForName:object:queue:usingBlock:`, consulte a documentação do desenvolvedor.

29

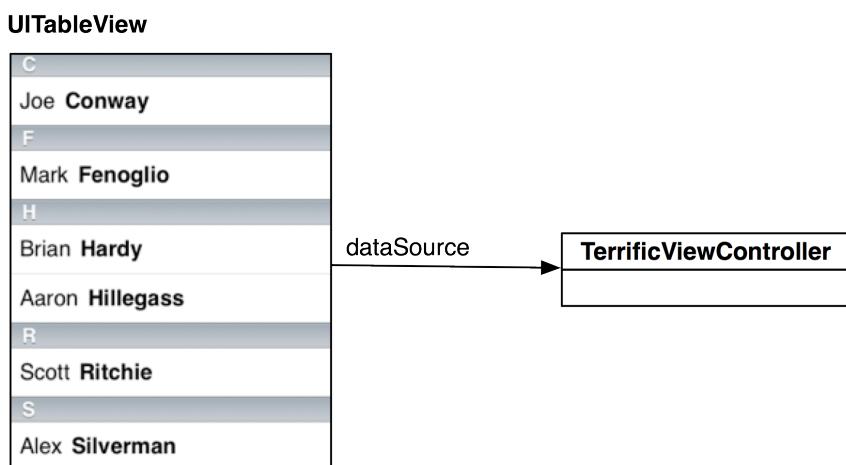
Protocolos

Agora, precisamos falar sobre um conceito um pouco abstrato. Certa vez, alguém disse: “Quem você é não é igual ao que você faz.” O mesmo é aplicável a objetos: a classe de um objeto é diferente de sua *função* em um sistema de trabalho. Por exemplo, um objeto pode ser uma instância de `NSMutableArray`, mas sua função em um aplicativo pode ser uma fila de trabalhos de impressão a ser executada.

Assim como o exemplo de array como fila de impressão, excelentes classes são mais gerais do que a função que elas podem desempenhar em qualquer aplicativo específico. Portanto, as instâncias dessas classes podem ser usadas de diferentes maneiras.

Por exemplo, em um aplicativo para iOS, frequentemente você exibe dados em uma instância de `UITableView`. Contudo, o objeto `UITableView` não contém os dados que exibe; ele precisa obter os dados de um objeto auxiliar. Você deve dizer a ele: “Aqui está o objeto que desempenhará a função de sua fonte de dados.”

Figure 29.1 Fonte de dados de `UITableView`



Um protocolo pode especificar um papel que um objeto pode desempenhar. (Se você está chegando ao Objective-C vindo de Java ou C#, protocolos são chamados de “interfaces” nessas comunidades.)

Como você aprendeu no Chapter 27, um protocolo é uma lista de declarações de método. Alguns desses métodos são obrigatórios, outros são opcionais. Se um objeto deve desempenhar um determinado papel, ele deve implementar os métodos obrigatórios. Ele pode escolher implementar um ou mais dos métodos opcionais.

O desenvolvedor que criou a classe `UITableView` especificou o papel da fonte de dados de `UITableView` criando o protocolo `UITableViewDataSource`. Aqui está ele:

```

// Just like classes, protocols can inherit from other protocols
// This protocol inherits from the NSObject protocol
@protocol UITableViewDataSource <NSObject>

// The following methods must be implemented by any table view data source
@required

// A table view has sections, each section can have several rows
- (NSInteger)tableView:(UITableView *)tv numberOfRowsInSection:(NSInteger)section;

// This index path is two integers (a section and a row)
// The table view cell is what the user sees in that section/row
- (UITableViewCell *)tableView:(UITableView *)tv
    cellForRowAtIndexPath:(NSIndexPath *)ip;

// These methods may be implemented by a table view data source
@optional

// If data source does not implement this method, table view has only one section
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tv;

// Rows can be deleted and moved
- (BOOL)tableView:(UITableView *)tv canEditRowAtIndexPath:(NSIndexPath *)ip;

- (BOOL)tableView:(UITableView *)tv canMoveRowAtIndexPath:(NSIndexPath *)ip;

- (void)tableView:(UITableView *)tv
commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
forRowAtIndexPath:(NSIndexPath *)ip;

- (void)tableView:(UITableView *)tv
moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath
    toIndexPath:(NSIndexPath *)destinationIndexPath;

// To save ink and paper, we are leaving out a few optional method declarations.

@end

```

Como classes, protocolos que a Apple fornece têm páginas de referência na documentação do desenvolvedor. Você pode navegar pela referência de um protocolo para ver os métodos que ela contém.

A classe **UITableView** tem a propriedade `dataSource`. Veja a declaração dela:

```
@property(nonatomic, assign) id<UITableViewDataSource> dataSource;
```

Sendo assim, a fonte de dados de uma visão de tabela pode ser um objeto de qualquer tipo (`id`), contanto que o objeto esteja em conformidade com o protocolo `UITableViewDataSource`. O compilador considera que um objeto está em conformidade com um protocolo se o objeto tiver implementado todos os métodos obrigatórios do protocolo.

Ao criar uma classe para desempenhar a função de fonte de dados da `UITableView`, você diz explicitamente: “essa classe está em conformidade com o protocolo `UITableViewDataSource`” no arquivo de cabeçalho. Isso será semelhante ao seguinte:

```
@interface TerrificViewController : UIViewController <UITableViewDataSource>
...
@end
```

Ou seja, “`TerrificViewController` é uma subclasse de `UIViewController` e está em conformidade com o protocolo `UITableViewDataSource`.”

Caso sua classe esteja em conformidade com diversos protocolos, liste-os dentro de colchetes angulares:

```
@interface TerrificViewController : UIViewController
    <UITableViewDataSource, UITableViewDelegate, UITextFieldDelegate>
```

Então, no arquivo `TerrificViewController.m`, você deve implementar os métodos obrigatórios. Se você se esquecer de implementar um dos métodos obrigatórios, verá um aviso importante do compilador.

Chamada de métodos opcionais

No Chapter 20, você aprendeu que, se enviar uma mensagem a um objeto e esse método não for implementado pela classe do objeto, o programa falhará. Então, como funcionam métodos opcionais em protocolos? Por exemplo, se uma classe que atua como fonte de dados de uma visão de tabela escolher não implementar o método de protocolo opcional **numberOfSectionsInTableView:**, você esperaria que o programa falhasse se a visão de tabela enviasse essa mensagem para sua fonte de dados.

Para evitar essa situação, primeiro, a visão de tabela pede para ver se sua fonte de dados implementa **numberOfSectionsInTableView:**.

Você pode perguntar a um objeto se ele implementa um método usando **respondsToSelector:**. Esse método é implementado em **NSObject**, de forma que você pode mandar a mensagem para qualquer objeto no seu programa. Você passa o seletor para o método sobre o qual está perguntando. O valor de retorno será YES se o objeto tiver esse método e NO se não tiver.

Será algo semelhante ao seguinte:

```
...
if ([_dataSource respondsToSelector:@selector(numberOfSectionsInTableView:)]) {
    _numberOfSections = [_dataSource numberOfSectionsInTableView:self];
} else {
    _numberOfSections = 1; // 1 is the default number of sections
}
...
```


30

Listas de propriedades

Algumas vezes, você precisará de um formato de arquivo que possa ser lido por computadores e por pessoas. Por exemplo, digamos que você queira manter uma descrição de sua carteira de ações em um arquivo. Conforme vai adicionando novas ações, seria bom que você conseguisse editar esse arquivo facilmente à mão. No entanto, também seria muito útil que um de seus programas pudesse ler esse arquivo. Ao enfrentar esse problema, a maioria dos programadores de Objective-C usa uma *lista de propriedades*.

Uma lista de propriedades é uma combinação de qualquer um dos seguintes itens:

- **NSArray**
- **NSDictionary**
- **NSString**
- **NSData**
- **NSDate**
- **NSNumber** (integer, float ou Boolean)

Por exemplo, um array de dicionários com chaves de string e objetos de dados é uma lista de propriedades (ou simplesmente uma “P-list”).

Ler e gravar uma lista de propriedades em um arquivo é muito fácil. No Xcode, crie um novo projeto: uma Foundation Command Line Tool chamada Stockz. No `main.m`, adicione o código a seguir:

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

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

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

        NSMutableDictionary *stock;

        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"AAPL"
                    forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:200]
                    forKey:@"shares"];
        [stocks addObject:stock];

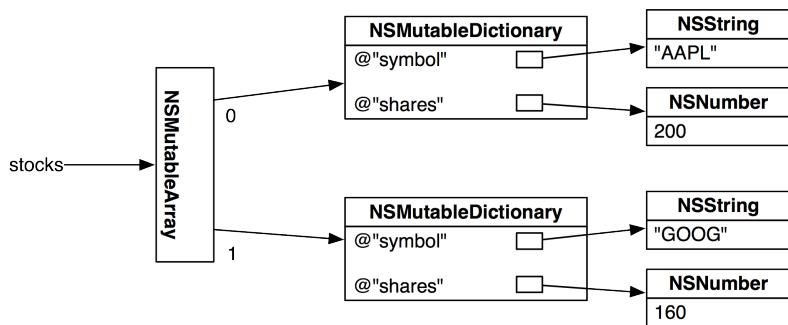
        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"GOOG"
                    forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:160]
                    forKey:@"shares"];
        [stocks addObject:stock];

        [stocks writeToFile:@"/tmp/stocks.plist"
                     atomically:YES];

    }
    return 0;
}
```

(Observe que você reutiliza o ponteiro `stock`. Você o utiliza para apontar para o primeiro dicionário e depois para o segundo.)

Figure 30.1 Um array de dicionários



Ao executar o programa, você obterá um arquivo: `stocks.plist`. Se abri-lo em um editor de textos, verá algo semelhante ao seguinte:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC
  "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<array>
  <dict>
    <key>shares</key>
    <integer>200</integer>
    <key>symbol</key>
    <string>AAPL</string>
  </dict>
  <dict>
    <key>shares</key>
    <integer>160</integer>
    <key>symbol</key>
    <string>GOOG</string>
  </dict>
</array>
</plist>
  
```

Interessante, não é mesmo? Legível a humanos. XML. Uma linha de código.

Se você estiver criando listas de propriedades manualmente, deverá saber que o Xcode conta com um editor integrado específico para listas de propriedades.

Agora, adicione o código que faça a leitura no arquivo:

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

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

        NSMutableDictionary *stock;

        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"AAPL"
                     forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:200]
                     forKey:@"shares"];
        [stocks addObject:stock];

        stock = [NSMutableDictionary dictionary];
        [stock setObject:@"GOOG"
                     forKey:@"symbol"];
        [stock setObject:[NSNumber numberWithInt:160]
                     forKey:@"shares"];
        [stocks addObject:stock];

        [stocks writeToFile:@"/tmp/stocks.plist"
                      atomically:YES];

        NSArray *stockList = [NSArray arrayWithContentsOfFile:@"/tmp/stocks.plist"];

        for (NSDictionary *d in stockList) {
            NSLog(@"I have %@ shares of %@", [d objectForKey:@"shares"], [d objectForKey:@"symbol"]);
        }
    }
    return 0;
}
```

Compile e execute o programa.

Desafio

Escreva uma ferramenta que crie uma lista de propriedades que contém todos os 8 tipos nela: array, dictionary, string, data, date, integer, float e boolean.

Part IV

Aplicativos orientados a eventos

Este é o ponto em que estamos nos concentrando e o motivo pelo qual você está lendo este livro: escrever aplicativos para iOS e Cocoa. Nos próximos dois capítulos, você terá uma noção do desenvolvimento de aplicativos. Seus aplicativos terão uma GUI (interface gráfica do usuário) e serão orientados a eventos.

Primeiramente, você escreverá um aplicativo para iOS e, em seguida, um aplicativo similar para Cocoa. Cocoa é a coleção de frameworks escrita pela Apple que você utiliza para escrever aplicativos no Mac. Você já tem familiaridade com um desses frameworks – o Foundation.

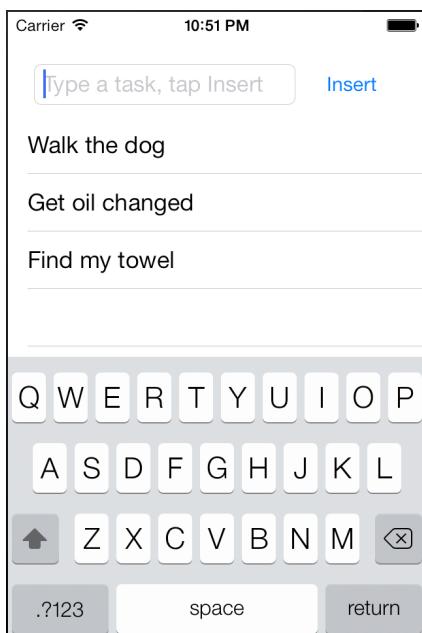
Para escrever aplicativos para iOS, você utiliza outro conjunto de frameworks chamado Cocoa Touch. Cocoa e Cocoa Touch têm alguns frameworks em comum, como o Foundation. Outros são específicos a uma plataforma ou a outra.

31

Seu primeiro aplicativo para iOS

Neste capítulo, você vai criar um aplicativo para iOS: um aplicativo simples de lista de tarefas, chamado iTahDoodle, que armazena seus dados como uma lista de propriedades. Veja abaixo como ficará a aparência quando você tiver terminado.

Figure 31.1 Aplicativo iTahDoodle completo



Mantivemos intencionalmente este capítulo e este aplicativo muito simples. Ele não se destina a prepará-lo para criar seus próprios aplicativos para iOS, mas lhe dará uma noção rápida sobre o desenvolvimento para iOS. Ele apresenta alguns conceitos e padrões importantes, e esperamos que ele lhe inspire a se aprofundar na programação para iOS após ter finalizado a leitura deste livro.

Aplicativos baseados em GUI

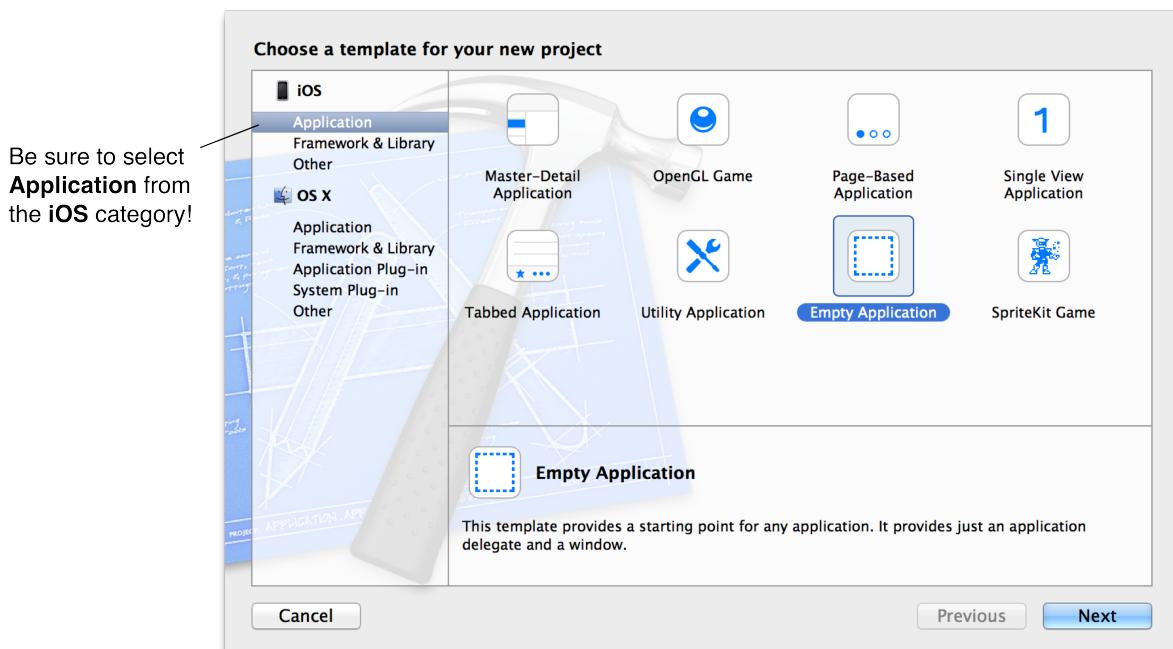
Nenhum dos programas que você criou até agora teve uma interface do usuário. Agora, você vai criar um aplicativo que tem uma *interface gráfica do usuário*, ou GUI.

Um aplicativo baseado em GUI é orientado a eventos. Quando o aplicativo é iniciado, ele comece um loop de execução que fica aguardando por eventos. Os eventos podem ser gerados pelo usuário (como o toque em um botão) ou pelo sistema (como um aviso de pouca memória). Quando um evento acontece, o aplicativo entra em ação para responder ao evento específico. Todos os aplicativos para iOS são orientados a eventos.

Introdução ao iTahDoodle

No Xcode, escolha File → New → Project... Na seção iOS (não na seção OS X), clique em Application. Entre as opções de template exibidas, selecione Empty Application.

Figure 31.2 Criação de um novo aplicativo para iOS

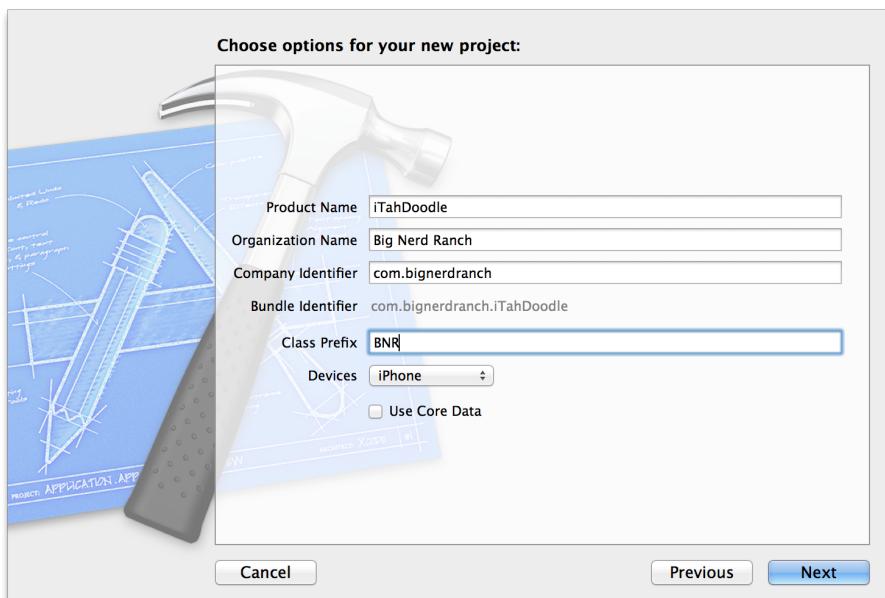


Os templates de projeto do Xcode contêm um código padronizado que pode acelerar o desenvolvimento. No entanto, você está usando o template Empty Application, que é o mais próximo de um template em branco que você pode obter. Permitir que o Xcode gere bastante código padronizado neste ponto pode atrapalhar seu aprendizado no que se refere ao modo como tudo funciona.

Os nomes dos templates geralmente mudam com as novas versões do Xcode, então não se preocupe caso não vir um template Empty Application. Procure o template que parecer mais simples e depois faça alterações para corresponder seu código com o código do livro. Se você tiver problemas ao harmonizar seu código ou seus templates de projeto, acesse o fórum da Big Nerd Ranch deste livro em forums.bignerdranch.com para obter ajuda.

Clique em Next, e na janela que é exibida, nomeie esse projeto iTahDoodle (Figure 31.3).

Figure 31.3 Configuração do projeto iTahDoodle



Para o Company Identifier, insira com.bignerdranch. O Company Identifier é usado para gerar o Bundle Identifier, que identifica de maneira exclusiva cada aplicativo na App Store.

O Class Prefix será prefixado ao nome da classe que o template cria para você e manterá suas classes separadas das classes que os outros criaram.

Por fim, torne o iTahDoodle um aplicativo de iPhone (em oposição a Universal ou iPad). O iTahDoodle não vai usar o Core Data. Clique em Next e finalize a criação do projeto.

BNRAppDelegate

Quando o Xcode criou esse projeto, ele criou uma classe chamada **BNRAppDelegate**. O “delegate de aplicativo” é o ponto inicial de um aplicativo e todo aplicativo para iOS tem um.

Abra o `BNRAppDelegate.h`. Você pode ver que o `UIKit.h` foi importado pelo template. O UIKit é o framework que contém a maioria das classes específicas de iOS, como `UITableView`, `UITextField` e `UIButton`. Observe também que **BNRAppDelegate** é uma subclasse de `UIResponder` e está em conformidade com o protocolo `UIApplicationDelegate`.

BNRAppDelegate tem uma propriedade que aponta para uma instância de `UIWindow`. Esse objeto preenche a tela de seu aplicativo de iOS. Você adiciona outros objetos (p. ex., uma instância de `UIButton`) à janela a fim de criar a interface do usuário do seu aplicativo.

No `BNRAppDelegate.h`, adicione quatro propriedades e um método de instância:

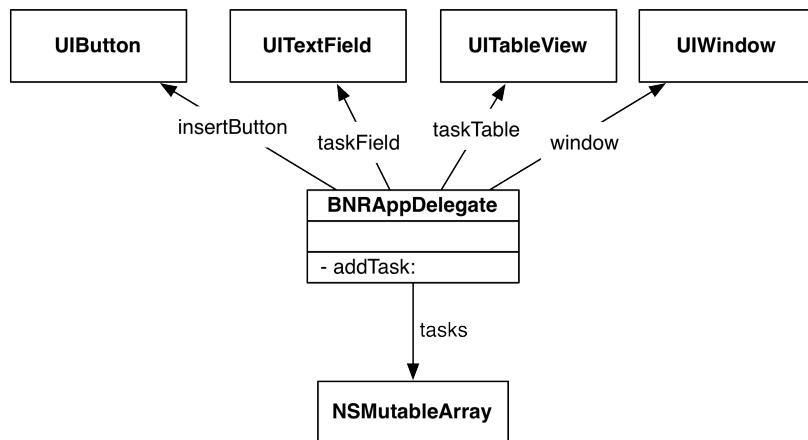
```
#import <UIKit/UIKit.h>

@interface BNRAppDelegate : UIResponder <UIApplicationDelegate>
@property (strong, nonatomic) UIWindow *window;
@property (nonatomic) UITableView *taskTable;
@property (nonatomic) UITextField *taskField;
@property (nonatomic) UIButton *insertButton;
@property (nonatomic) NSMutableArray *tasks;
- (void)addTask:(id)sender;
@end
```

As três primeiras propriedades são ponteiros para os objetos que o usuário pode ver e com os quais pode interagir – uma visão de tabela que exibirá todas as tarefas a serem realizadas, um campo de texto em que você pode inserir uma nova tarefa e um botão que adicionará a nova tarefa à tabela. O quarto objeto é um array mutável. Esse é o local em que você armazenará as tarefas como strings.

A Figure 31.4 é um diagrama dos seis objetos que vão compor o iTahDoodle. Há a instância de `BNRAppDelegate`; esse objeto tem ponteiros para outros cinco: instâncias de `UIWindow`, `UITableView`, `UITextField`, `UIButton` e `NSMutableArray`.

Figure 31.4 Diagrama de objetos para iTahDoodle



Xcode pode avisá-lo que o `addTask:` ainda não foi implementado. Isso é verdadeiro, mas você pode ignorar o aviso agora. Você implementará o `addTask:` mais tarde no capítulo.

Antes de você continuar a trabalhar nos objetos exibidos na Figure 31.4, vejamos um pouco de teoria sobre objetos e seus relacionamentos.

Modelo-Visão-Controlador (MVC)

Modelo-Visão-Controlador, ou MVC, é um padrão de projeto baseado na ideia de que qualquer classe que você cria deve se classificar em uma das três categorias de trabalho: modelo, visão ou controlador. Veja um detalhamento da divisão de trabalho:

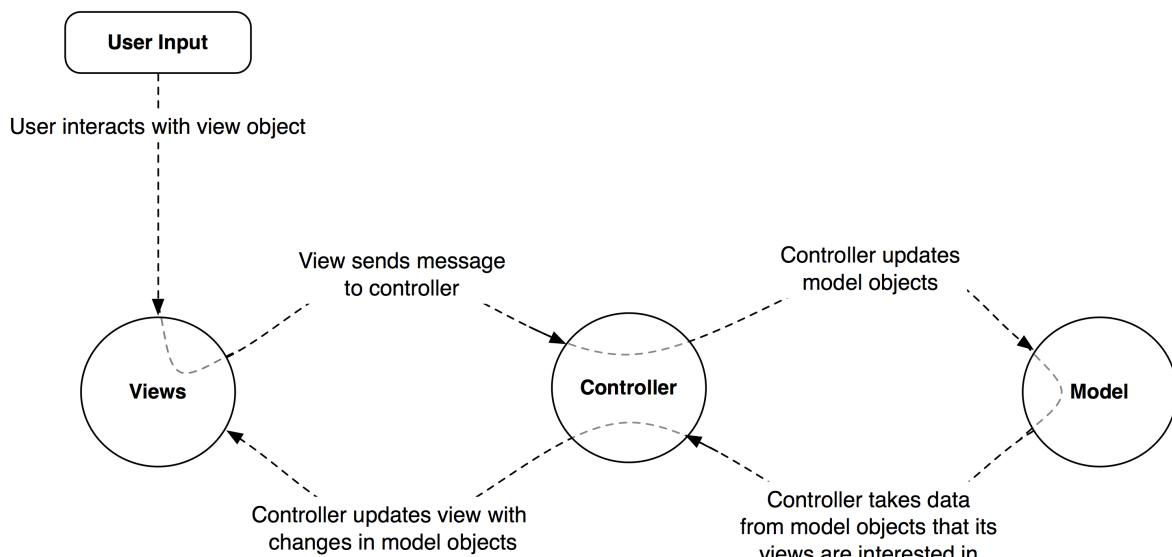
- Modelos são responsáveis por armazenar dados e torná-los disponíveis para outros objetos. Os objetos modelo não conhecem a interface de usuário nem sabem como se projetarem na tela; a única finalidade é

reter e gerenciar dados. **NSString**, **NSDate** e **NSArray** são objetos modelo tradicionais. No iTahDoodle, a **NSMutableArray** onde as tarefas são armazenadas é um objeto modelo. Mais tarde, cada tarefa individual será descrita em uma instância de **NSString**, e essas também serão objetos modelo.

- Visões são os elementos visuais de um aplicativo. As visões sabem como se projetarem na tela e como responder à entrada do usuário. As visões não conhecem os dados reais que exibem nem sabem como eles são estruturados e armazenados. **UIView** e suas diversas subclasses, incluindo **UIWindow**, são exemplos comuns de objetos visão. No iTahDoodle, seus objetos visão são as instâncias de **UITableView**, **UITextField** e **UIButton**. Uma regra geral simples é que: se você pode ver, é uma visão.
- Os controladores executam a lógica necessária para conectar e conduzir diferentes partes de seu aplicativo. Eles processam eventos e coordenam os outros objetos em seu aplicativo. Os controladores são a base real de qualquer aplicativo. Embora **BNRAppDelegate** seja o único controlador no iTahDoodle, um aplicativo complexo terá muitos controladores diferentes que coordenam os objetos modelo e visão, assim como outros controladores.

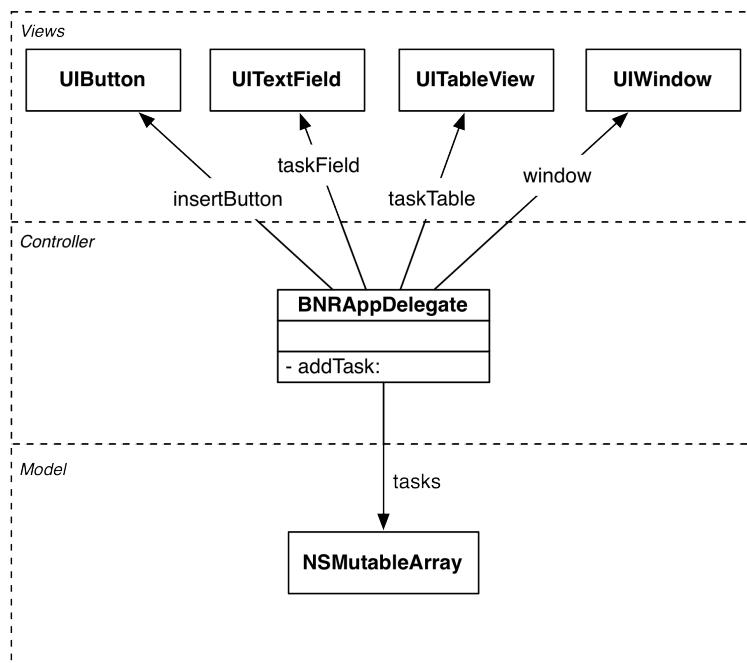
Figure 31.5 mostra o fluxo de controle entre os objetos em resposta a um evento de usuário, como o toque em um botão. Observe que os objetos modelo e visão não se comunicam diretamente uns com os outros; os controladores ficam no centro de tudo, recebendo as mensagens de alguns objetos e encaminhando instruções a outros.

Figure 31.5 Fluxo MVC com entrada do usuário



A maioria das APIs Cocoa e Cocoa Touch é escrita levando o MVC em consideração; desse modo, o mesmo também deve ocorrer com seu próprio código. A Figure 31.6 mostra essa divisão de trabalho no iTahDoodle.

Figure 31.6 Diagrama de objetos do iTahDoodle



Agora, vamos voltar a nosso controlador, a instância de **BNRAppDelegate**.

O delegate do aplicativo

Quando um aplicativo para iOS é iniciado pela primeira vez, existem muitas configurações ocorrendo em segundo plano. Uma instância de **UIApplication** é criada para controlar o estado do aplicativo e atuar como um contato com o sistema operacional. Uma instância de **BNRAppDelegate** também é criada e configurada como delegate da instância de **UIApplication** (que explica o nome “delegate de aplicativo”).

Isso torna **BNRAppDelegate** uma classe ocupada. Na verdade, todos os códigos que você vai escrever para este aplicativo estarão no `BNRAppDelegate.m`. Seria bom ter uma maneira de organizar a classe de modo que fosse fácil encontrar métodos rapidamente. Você pode usar `#pragma mark` para agrupar métodos em uma classe para navegação mais fácil.

No `BNRAppDelegate.m`, adicione uma `#pragma mark` que identifique os métodos existentes como callbacks de delegate de aplicativo:

```
#import "BNRAppDelegate.h"

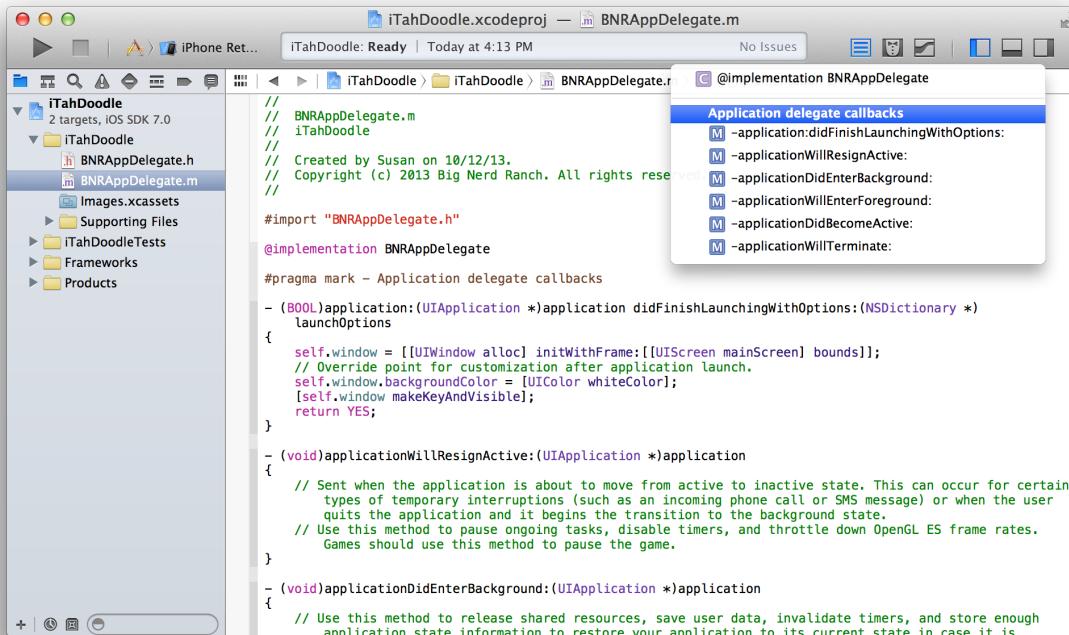
@implementation BNRAppDelegate

#pragma mark - Application delegate callbacks

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...
}
```

A seguir, na barra de navegação, na parte superior da área do editor, localize o item à direita de `BNRAppDelegate.m`. Clique nesse item, e o Xcode exibirá uma lista de locais nesse arquivo. Você pode clicar em qualquer um dos nomes de métodos para ser direcionado diretamente para a implementação do método em questão. Observe que sua `pragma mark` é exibida na parte superior.

Figure 31.7 Navegação usando pragma mark



Atualmente, a classe de **BNRAppDelegate** contém apenas callbacks de delegate de aplicativo. Mas logo você vai adicionar métodos com funções diferentes e usará `#pragma mark` para agrupá-los.

O primeiro callback de delegate de aplicativo na pragma mark é o **application:didFinishLaunchingWithOptions:**. Esse método é muito importante. Embora o aplicativo esteja sendo iniciado, ele ainda não está pronto para funcionar ou receber entradas. Quando o aplicativo estiver pronto, a instância de **UIApplication** enviará a seu delegate a mensagem **application:didFinishLaunchingWithOptions:**. É onde você coloca tudo o que precisa ser feito antes de o usuário interagir com o aplicativo.

Configuração de visões

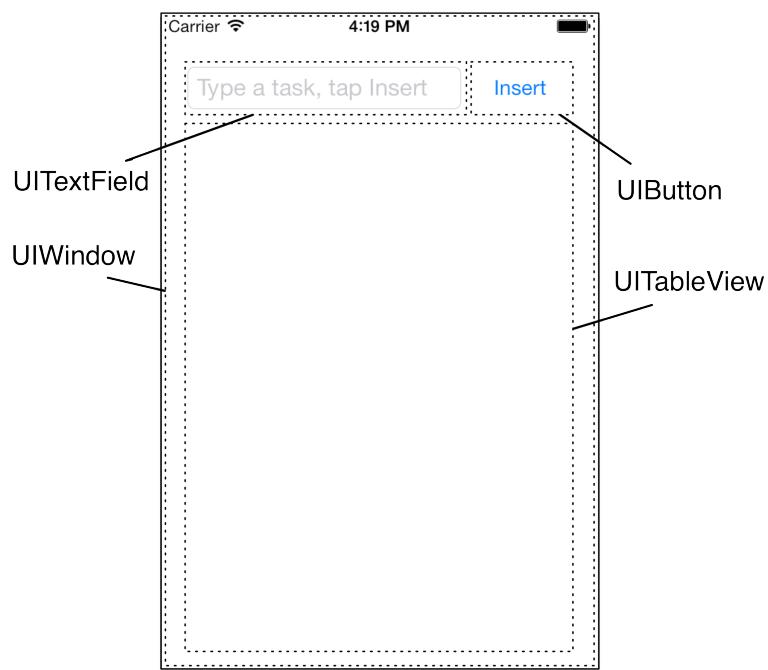
Uma coisa que você precisa fazer antes de o aplicativo ficar pronto para o usuário é configurar seus objetos visão: o campo de texto, o botão e a visão de tabela. Isso significa criá-los, configurá-los e colocá-los na tela.

No iTahDoodle, você vai configurar suas visões de maneira programática no **application:didFinishLaunchingWithOptions:**. O Xcode também tem uma ferramenta visual do tipo “arrastar e soltar” para configurar visões, que você usará no Chapter 32.

O código para criar e configurar visões é denso, e não vamos cobri-lo em detalhes. A sintaxe detalhada de criação e exibição de visões na tela é um tópico para um livro especificamente sobre programação de aplicativos para iOS.

Ainda assim, você pode acompanhar a essência do que está acontecendo. Primeiramente, você cria cada objeto e depois o configura definindo algumas de suas propriedades. Em seguida, os objetos de visão configurados são adicionados como *subvisões* do objeto de janela e, por fim, a janela é exibida na tela.

Figure 31.8 Objetos de visão no iTahDoodle



No `BNRAppDelegate.m`, remova qualquer código padronizado do
`application:didFinishLaunchingWithOptions:` e substitua-o com o seguinte código:

```

#pragma mark - Application delegate callbacks

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Create and configure the UIWindow instance
    // A CGRect is a struct with an origin (x,y) and a size (width,height)
    CGRect winFrame = [[UIScreen mainScreen] bounds];
    UIWindow *theWindow = [[UIWindow alloc] initWithFrame:winFrame];
    self.window = theWindow;

    // Define the frame rectangles of the three UI elements
    // CGRectGetMake() creates a CGRect from (x, y, width, height)
    CGRect tableFrame = CGRectMake(0, 80, winFrame.size.width,
                                   winFrame.size.height - 100);
    CGRect fieldFrame = CGRectMake(20, 40, 200, 31);
    CGRect buttonFrame = CGRectMake(228, 40, 72, 31);

    // Create and configure the UITableView instance
    self.taskTable = [[UITableView alloc] initWithFrame:tableFrame
                                              style:UITableViewStylePlain];
    self.taskTable.separatorStyle = UITableViewCellStyleNone;

    // Tell the table view which class to instantiate whenever it
    // needs to create a new cell
    [self.taskTable registerClass:[UITableViewCell class]
        forCellReuseIdentifier:@"Cell"];

    // Create and configure the UITextField instance where new tasks will be entered
    self.taskField = [[UITextField alloc] initWithFrame:fieldFrame];
    self.taskField.borderStyle = UITextBorderStyleRoundedRect;
    self.taskField.placeholder = @"Type a task, tap Insert";

    // Create and configure the UIButton instance
    self.insertButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    self.insertButton.frame = buttonFrame;

    // Give the button a title
    [self.insertButton setTitle:@"Insert"
                      forState:UIControlStateNormal];

    // Add our three UI elements to the window
    [self.window addSubview:self.taskTable];
    [self.window addSubview:self.taskField];
    [self.window addSubview:self.insertButton];

    // Finalize the window and put it on the screen
    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];

    return YES;
}

```

Agora que você configurou suas visões, pode executar o iTahDoodle no simulador de iOS para vê-las.

Execução do simulador de iOS

O Xcode vem com um simulador de iOS que lhe permite executar aplicativos de iOS em seu Mac. O simulador é uma forma fácil de ver como o seu aplicativo se comportará quando for executado em um dispositivo iOS. Em particular, você vai simular o iTahDoodle sendo executado em um iPhone com tela retina de 3,5 polegadas.

Primeiramente, olhe para a direita da execução e pare os botões na barra de ferramentas do Xcode. Você verá o iTahDoodle e, em seguida, uma descrição do dispositivo.

Figure 31.9 Simulação de um iPhone com uma tela retina de 3,5 polegadas

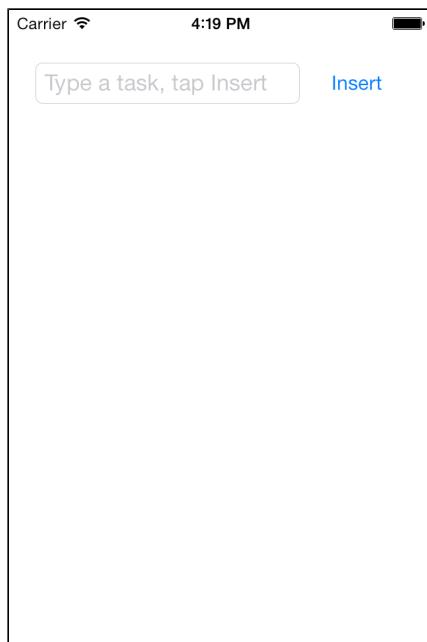


Se na descrição não estiver escrito iPhone Retina (3.5-inch), então, clique na descrição para ver um menu suspenso com as opções disponíveis. Escolha iPhone Retina (3.5-inch) no menu.

Compile e execute o aplicativo. O simulador vai ser iniciado imediatamente, mas pode levar um minuto para o iTahDoodle ser exibido na tela pela primeira vez.

Você pode ver o campo de texto e o botão. Por padrão, a visão de tabela e a janela não têm delimitações ou áreas sombreadas, o que as torna de difícil visualização. Mas elas estão lá!

Figure 31.10 Visualização de objetos na tela



Se você clicar no campo de texto, o teclado será exibido e você poderá digitar o texto no campo. Esse é o comportamento padrão para uma instância de **UITextField**. No entanto, o **UIButton** não faz nada quando é tocado, e **UITableView** não exibe nada.

A sua próxima etapa é gravar um código que informa a esses objetos como se comportarem. Os desenvolvedores geralmente se referem à implementação do comportamento de objetos visão como “rede” ou “conexão”.

Coneção do botão

Um botão funciona usando um destino-ação, como você aprendeu no Chapter 27. A ação do botão é a mensagem que você deseja enviar quando o botão é pressionado. O destino do botão é o objeto para o qual essa mensagem precisa ser enviada.

No `BNRAppDelegate.m`, dê ao botão `Insert` um par destino-ação:

```
#pragma mark - Application delegate callbacks

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...

    // Create and configure a rounded rect Insert button
    self.insertButton = [UIButton buttonWithType:UIButtonTypeRoundedRect];
    self.insertButton.frame = buttonFrame;

    // Give the button a title
    [self.insertButton setTitle:@"Insert"
                      forState:UIControlStateNormal];

    // Set the target and action for the Insert button
    [self.insertButton addTarget:self
                           action:@selector(addTask:)
                     forControlEvents:UIControlEventTouchUpInside];

    ...

    return YES;
}
```

O destino é `self`, e a ação é `addTask:`. Assim, quando o botão Insert for pressionado, ele enviará a mensagem `addTask:` para `BNRAppDelegate`. A próxima etapa, então, é implementar o método `addTask:` em `BNRAppDelegate`.

Por fim, o `addTask:` vai recuperar o texto inserido em `taskField` e adicioná-lo no array `tasks`. A tarefa será, então, exibida na visão de tabela. Mas, visto que você ainda não encadeou a visão de tabela, você vai implementar `addTask:` para recuperar o texto do `taskField` e simplesmente fazer o registro dele no console.

No `BNRAppDelegate.m`, adicione uma implementação de `addTask:` na parte inferior do arquivo junto com uma nova pragma mark:

```
@implementation BNRAppDelegate

#pragma mark - Application delegate callbacks

...

#pragma mark - Actions

- (void)addTask:(id)sender
{
    // Get the task
    NSString *text = [self.taskField text];

    // Quit here if taskField is empty
    if ([text length] == 0) {
        return;
    }

    // Log text to console
    NSLog(@"Task entered: %@", text);

    // Clear out the text field
    [self.taskField setText:@""];
    // Dismiss the keyboard
    [self.taskField resignFirstResponder];
}

@end
```

Qual a finalidade desse `resignFirstResponder`? Veja uma breve explicação:

Alguns objetos visão também são *controles*. Um controle é uma visão com a qual o usuário pode interagir: Botões, barras deslizantes e campos de texto são exemplos de controles. (Tenha em mente que o termo “controle” não tem nada a ver com “controladores” no MVC.)

Quando há controles na tela, um deles pode ser o *primeiro respondente*. Quando um controle tem o status de primeiro respondente, ele obtém a primeira chance de lidar com a entrada de texto feita a partir do teclado e eventos de agitação (como quando o usuário agita o dispositivo para desfazer a última ação).

Quando o usuário interage em um controle que pode aceitar o status de primeiro respondente, é enviada a mensagem `becomeFirstResponder` a esse controle. Quando um controle que aceita a entrada de texto (como um campo de texto) torna-se o primeiro respondente, é exibido um teclado na tela. No final de `addTask:`, você diz ao campo de texto que abra mão de seu status, o que faz com que o teclado deixe de ser exibido.

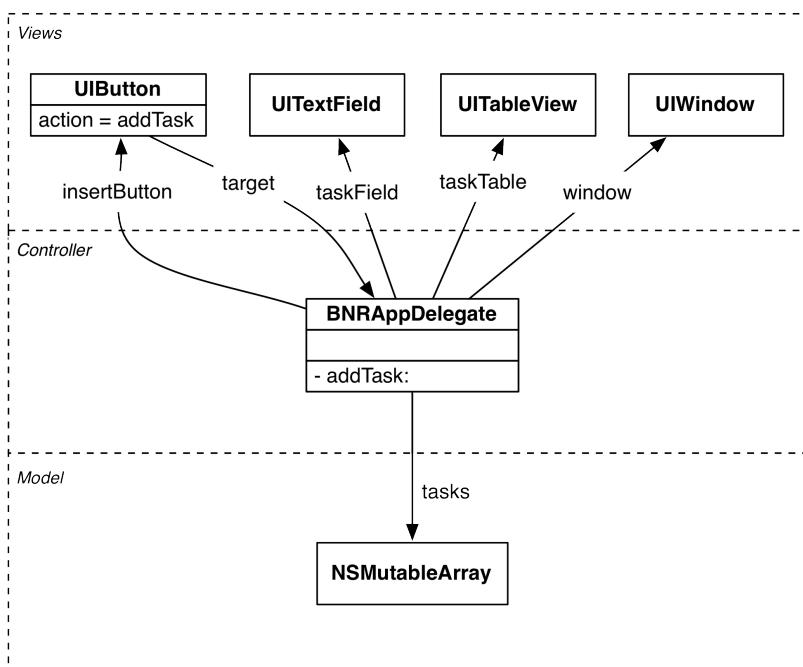
Compile e execute o aplicativo. (Você terá que interromper a instância de execução atual do iTahDoodle antes de o Xcode poder criar e ser executado novamente.)

Assim que o iTahDoodle estiver sendo executado novamente, insira alguma coisa no campo de texto, clique em Insert e confirme se o texto foi registrado no console.

Você também pode ver um aviso no console com a seguinte mensagem: `Application windows are expected to have a root view controller at the end of application launch.` (Espera-se que as janelas do aplicativo tenham um controlador de visão raiz no final da inicialização do aplicativo). Você pode ignorar isso. Para usar somente as configurações básicas, você não deve implementar um “controlador de visão raiz” para o iTahDoodle, e esse aplicativo simples vai operar bem sem o controlador. Você aprenderá tudo sobre os controladores de visão em *iOS Programming: The Big Nerd Ranch Guide* (Programação para iOS: Guia da Big Nerd Ranch) ou qualquer outro livro sobre desenvolvimento para iOS.

Veja um diagrama de objetos atualizado mostrando um par destino-ação.

Figure 31.11 Diagrama de objetos do iTahDoodle atualizado



Conexão da visão de tabela

Você tem uma visão de tabela na tela, mas ela não exibe nada. Como um objeto de visão, a visão de tabela não contém nada sobre os dados reais. Ela precisa de um objeto que atue como sua *fonte de dados*. Uma fonte de dados da visão de tabela informa à visão de tabela o que exibir.

No iTahDoodle, a fonte de dados da visão de tabela será a instância de `BNRAppDelegate`. Para que `BNRAppDelegate` faça esse trabalho, ela deve estar em conformidade com o protocolo `UITableViewDataSource`.

No `BNRAppDelegate.h`, declare que `BNRAppDelegate` está em conformidade com `UITableViewDataSource`:

```

@interface BNRAppDelegate : UIResponder
    <UIApplicationDelegate, UITableViewDataSource>

@property (strong, nonatomic) UIWindow *window;

@property (nonatomic) UITableView *taskTable;
@property (nonatomic) UITextField *taskField;
@property (nonatomic) UIButton *insertButton;

@property (nonatomic) NSMutableArray *tasks;

- (void)addTask:(id)sender;

```

No BNRAppDelegate.m, atualize **application:didFinishLaunchingWithOptions:** para enviar uma mensagem à visão de tabela que torne a instância de **BNRAppDelegate** sua fonte de dados:

```

...
// Create and configure the table view
self.taskTable = [[UITableView alloc] initWithFrame:CGRectMake(0, 0, 320, 460)
                                         style:UITableViewStylePlain];
self.taskTable.separatorStyle = UITableViewCellSeparatorStyleNone;

// Make the BNRAppDelegate the table view's dataSource
self.taskTable.dataSource = self;

// Tell the table view which class to instantiate whenever it
// needs to create a new cell
[self.taskTable registerClass:[UITableViewCell class]
    forCellReuseIdentifier:@"Cell"];
...

```

O protocolo UITableViewDataSource tem dois métodos necessários. A fonte de dados de uma visão de tabela deve estar preparada para informar à visão de tabela:

- quantas linhas há em uma determinada seção da tabela (**tableView:numberOfRowsInSection:**)
- qual deve ser a célula em uma determinada linha (**tableView:cellForRowAtIndexPath:**)

No BNRAppDelegate.m, implemente esses callbacks:

```

@implementation BNRAppDelegate
...

#pragma mark - Table view management

- (NSInteger)tableView:(UITableView *)tableView
 numberOfRowsInSection:(NSInteger)section
{
    // Because this table view only has one section,
    // the number of rows in it is equal to the number
    // of items in the tasks array
    return [self.tasks count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath
{
    // To improve performance, this method first checks
    // for an existing cell object that we can reuse
    // If there isn't one, then a new cell is created
    UITableViewCell *c = [self.taskTable dequeueReusableCellWithIdentifier:@"Cell"];

    // Then we (re)configure the cell based on the model object,
    // in this case the tasks array, ...
    NSString *item = [self.tasks objectAtIndex:indexPath.row];
    c.textLabel.text = item;

    // ... and hand the properly configured cell back to the table view
    return c;
}
@end

```

Esses métodos interagem com o array tasks. Você declarou essa propriedade, mas ainda não criou o objeto de array.

No BNRAppDelegate.m, na parte superior do **application:didFinishLaunchingWithOptions:**, crie um array vazio e mutável:

```
#pragma mark - Application delegate callbacks

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Create an empty array to get us started
    self.tasks = [NSMutableArray array];

    ...
}
```

Finalmente, modifique a implementação do **addTask:** para exibir a tarefa na visão de tabela.

```
#pragma mark - Actions

- (void)addTask:(id)sender
{
    // Get the task
    NSString *text = [self.taskField text];

    // Quit here if taskField is empty
    if ([text length] == 0) {
        return;
    }

    // Log task to console
    NSLog(@"%@", text);

    // Add it to the working array
    [self.tasks addObject:text];

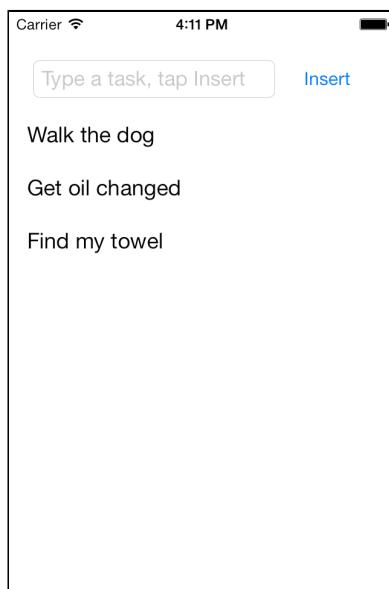
    // Refresh the table so that the new item shows up
    [self.taskTable reloadData];

    // Clear out the text field
    [self.taskField setText:@""];
}

// Dismiss the keyboard
[self.taskField resignFirstResponder];
}
```

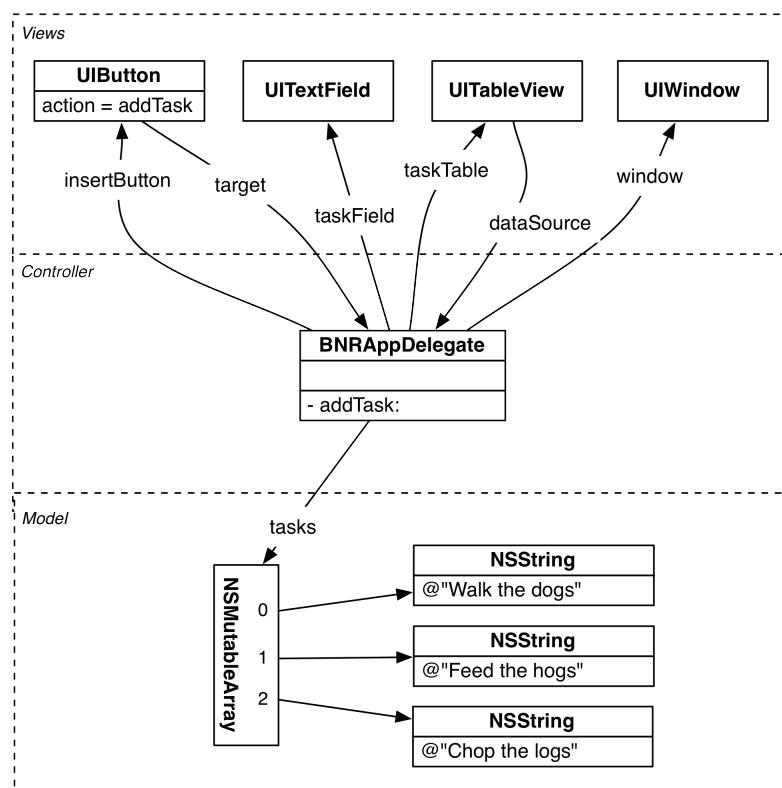
Compile e execute o aplicativo. Insira algumas tarefas para sua lista. Cada uma deve ser exibida na visão de tabela.

Figure 31.12 aplicativo iTahDoodle, concluído



A Figure 31.13 exibe o diagrama de objetos completo, incluindo o relacionamento da fonte de dados que você estabeleceu e as strings adicionadas ao array de tasks.

Figure 31.13 Diagrama de objetos completo do iTahDoodle



Salvamento e carregamento de dados

Atualmente, se o usuário forçar o encerramento do iTahDoodle, ou se o sistema fechar o aplicativo enquanto o usuário estiver fazendo alguma coisa, todas as alterações na lista de tarefas serão perdidas e o usuário terá que inseri-las todas novamente quando reiniciar o aplicativo. Essa não é a experiência de usuário que você deseja. Quando os usuários encerram o aplicativo, eles querem que suas listas de tarefas continuem lá para que

possam consultá-las novamente depois. Nesta seção, você habilitará o iTahDoodle para salvar as tarefas do usuário entre execuções do aplicativo.

Adição de uma função auxiliar em C

Para salvar as tarefas do usuário entre execuções, o iTahDoodle vai armazenar as tarefas no disco como uma lista de propriedade – um arquivo XML. Você precisará de uma maneira de obter o local do arquivo durante a execução de seu aplicativo. Assim, você vai criar uma função em C chamada **BNRDocPath** que retorne esse caminho de arquivo como uma **NSString**.

No Objective-C, nós normalmente concluímos as tarefas com métodos. Então, quando usamos uma função em C em um aplicativo em Objective-C, sempre nos referimos a ela como uma *função auxiliar*.

No `BNRAppDelegate.h`, declare **BNRDocPath()**.

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

// Declare a helper function that you will use to get a path
// to the location on disk where you can save the to-do list
NSString *BNRDocPath(void);

@interface BNRAppDelegate : UIResponder
    <UIApplicationDelegate, UITableViewDataSource>
...
```

Observe que você declara **BNRDocPath()** acima da declaração da classe. Embora **BNRDocPath()** seja declarada no arquivo `BNRAppDelegate.h`, ela não faz parte da classe **BNRAppDelegate**. Na verdade, essa função poderia ter seu próprio cabeçalho e arquivos de implementação no projeto iTahDoodle. No entanto, como há apenas uma função auxiliar no iTahDoodle, você a está colocando nos arquivos de classe do delegate de aplicativo para simplificar as coisas.

No `BNRAppDelegate.m`, implemente **BNRDocPath()**. Certifique-se de implementá-la após `#import`, mas antes da linha `@implementation` (que é onde a implementação da classe **BNRDelegate** começa).

```
#import "BNRAppDelegate.h"

// Helper function to fetch the path to our to-do data stored on disk
NSString *BNRDocPath()
{
    NSArray *pathList = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                                          NSUserDomainMask,
                                                          YES);
    return [pathList[0] stringByAppendingPathComponent:@"data.td"];
}

@implementation BNRAppDelegate
...
```

Observe que a função **BNRDocPath()** usa a função **NSSearchPathForDirectoriesInDomains()** que discutimos no Chapter 26.

Salvamento de dados de tarefa

Quando um aplicativo em Cocoa Touch é fechado ou enviado para segundo plano, ele envia a seu delegate uma mensagem do protocolo `UIApplicationDelegate` para que o delegate possa assumir e responder a esses eventos de maneira apropriada.

No `BNRAppDelegate.m`, localize o método **applicationDidEnterBackground:** e substitua seu conteúdo pelo código que usa **BNRDocPath()** para salvar a lista de tarefas quando o usuário sai do aplicativo:

```
- (void)applicationDidEnterBackground:(UIApplication *)application
{
    // Save our tasks array to disk
    [self.tasks writeToFile:BNRDocPath() atomically:YES];
}
```

Agora, quaisquer tarefas que o usuário inserir serão gravadas no disco por questões de segurança.

Carregamento de dados de tarefas

Para finalizar, você precisa carregar as tarefas salvas pelo usuário quando o aplicativo é inicializado.

No `BNRAppDelegate.m`, adicione o código a seguir no início do `application:didFinishLaunchingWithOptions:`:

```
#pragma mark - Application delegate callbacks

- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    // Create an empty array to get us started
    self.tasks = [NSMutableArray array];

    // Load an existing dataset or create a new one
    NSArray *plist = [NSArray arrayWithContentsOfFile:BNRDocPath()];
    if (plist) {
        // We have a dataset; copy it into tasks
        self.tasks = [plist mutableCopy];
    } else {
        // There is no dataset; create an empty array
        self.tasks = [NSMutableArray array];
    }

    // Create and configure the UIWindow instance
    // A CGRect is a struct with an origin (x,y) and a size (width,height)
    CGRect winFrame = [[UIScreen mainScreen] bounds];
    UIWindow *theWindow = [[UIWindow alloc] initWithFrame:winFrame];
    self.window = theWindow;
    ...
}
```

Compile e execute o aplicativo. Insira algumas tarefas e clique no botão Home. Em seguida, clique no ícone do iTahDoodle para reinicializar o aplicativo. Suas tarefas estarão lá esperando por você.

Observe que os dados só serão salvos e rearmazenados quando o aplicativo for enviado para segundo plano ou encerrado de modo apropriado. Se você interromper o aplicativo a partir do Xcode enquanto ele estiver ativo, os dados podem não ser salvos.

Parabéns! Você criou seu primeiro aplicativo para iOS. Ainda há muito, muito mais a fazer e aprender, e esperamos que você esteja disposto a aceitar o desafio.

Para os mais curiosos: o que dizer sobre `main()`?

Quando você começou a aprender sobre C e Objective-C, viu que o ponto inicial para o código de seu programa é a função `main()`. Isso também ocorre no desenvolvimento em Cocoa e Cocoa Touch, embora seja extremamente raro editar essa função em aplicativos em Cocoa e Cocoa Touch. Abra o `main.m` e você verá o porquê:

```
return UIApplicationMain(argc, argv, nil, NSStringFromClass([BNRAppDelegate class]));
```

Bom, isso foi decepcionante. Apenas uma linha de código real.

A função `UIApplicationMain` cria os objetos necessários para a execução de seu aplicativo. Primeiro, ela cria uma única instância da classe `UIApplication`. Depois, ela cria uma instância de qualquer classe que seja indicada pelo quarto e último argumento e a define como o delegate do aplicativo. Desse modo, ela poderá enviar mensagens a seu delegate quando houver pouca memória, quando o aplicativo for fechado ou enviado para segundo plano ou quando a inicialização for finalizada.

E este é o caminho percorrido desde `main()` até `application:didFinishLaunchingWithOptions:` e seu código personalizado.

Para os mais curiosos: execução do iTahDoodle em um dispositivo

No momento, seu aplicativo está sendo executado no simulador. Para executá-lo em um dispositivo (ou para publicar aplicativos na App Store), você vai precisar se associar ao programa de desenvolvedores para iOS (iOS

Developer Program) da Apple, que custa US\$ 99 por ano. Assim que for um membro, você pode se registrar e registrar seus dispositivos no portal do desenvolvedor. Essas informações serão usadas para criar um perfil de provisionamento para seus aplicativos que lhe permitirá executá-los em seus dispositivos.

O processo de provisionamento é complicado e trabalhoso. Para ajudá-lo no processo, a Apple criou um *App Distribution Guide* (Guia de distribuição de aplicativos) que descreve as etapas necessárias em detalhes. Procure por ele no site do desenvolvedor para iOS da Apple.

Na verdade, a menos que você não consiga viver sem ver o iTahDoodle no seu iPhone, sugerimos que você deixe de lado essa aventura até que esteja lendo um livro de desenvolvimento para iOS, como o *iOS Programming: The Big Nerd Ranch Guide* (Programação para iOS: Guia da Big Nerd Ranch).

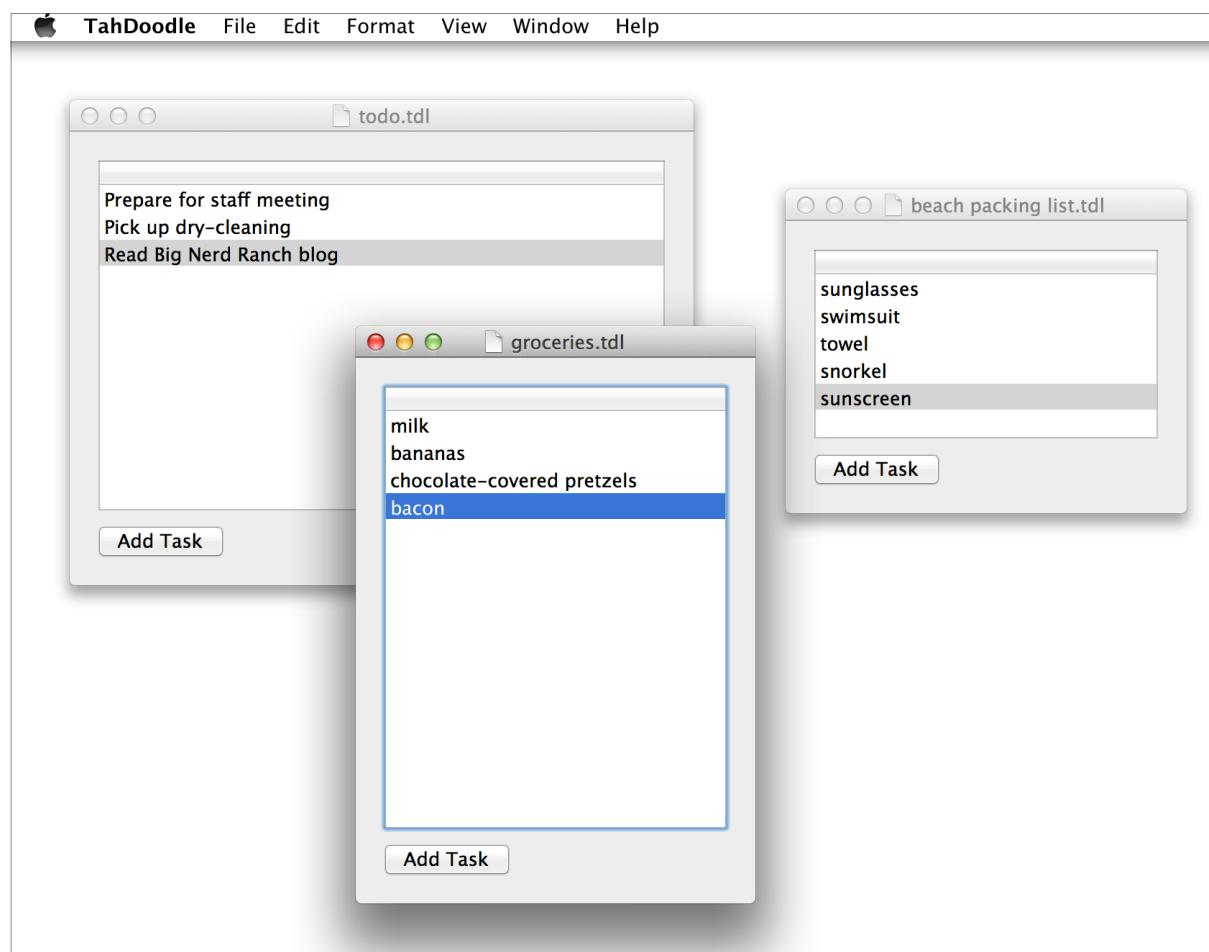
32

Seu primeiro aplicativo em Cocoa

Neste capítulo, você vai criar o TahDoodle, um aplicativo de desktop para Mac. Como o iTahDoodle, o TahDoodle é um aplicativo simples de lista de tarefas, que armazena seus dados como uma lista de propriedades. Como no último capítulo, esse aplicativo é muito simples; ele vai lhe dar uma noção rápida de como o desenvolvimento para Cocoa funciona.

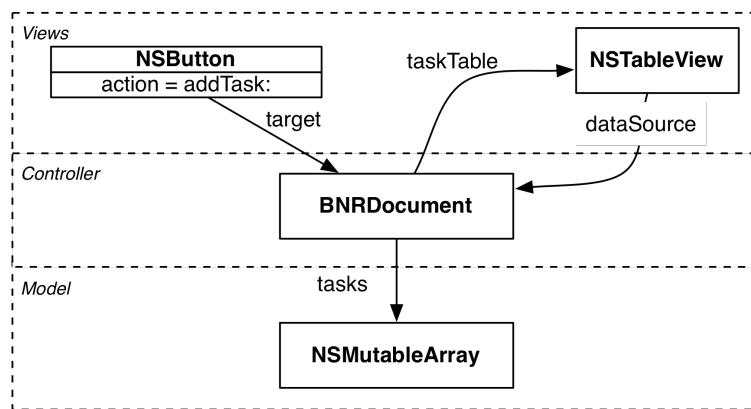
TahDoodle será um *aplicativo baseado em documento*. Isso permite que os usuários tenham várias janelas (cada uma representando um arquivo diferente) abertas ao mesmo tempo.

Figure 32.1 Aplicativo TahDoodle completo



Aqui temos um diagrama de objetos do aplicativo TahDoodle completo.

Figure 32.2 Diagrama de objetos para TahDoodle



Você vai apresentar a lista de tarefas em uma instância de **NSTableView**. Uma instância de **NSButton** permitirá aos usuários adicionar uma linha na visão de tabela, de modo que uma nova tarefa possa ser adicionada. Todas as tarefas podem ser editadas diretamente na **NSTableView**.

Uma instância de uma classe chamada **BNRDocument** é o controlador para o TahDoodle. Ela vai conectar o objeto de modelo (um array mutável das strings) com dois objetos de visão. Os objetos de visão são **NSButton** e **NSTableView**.

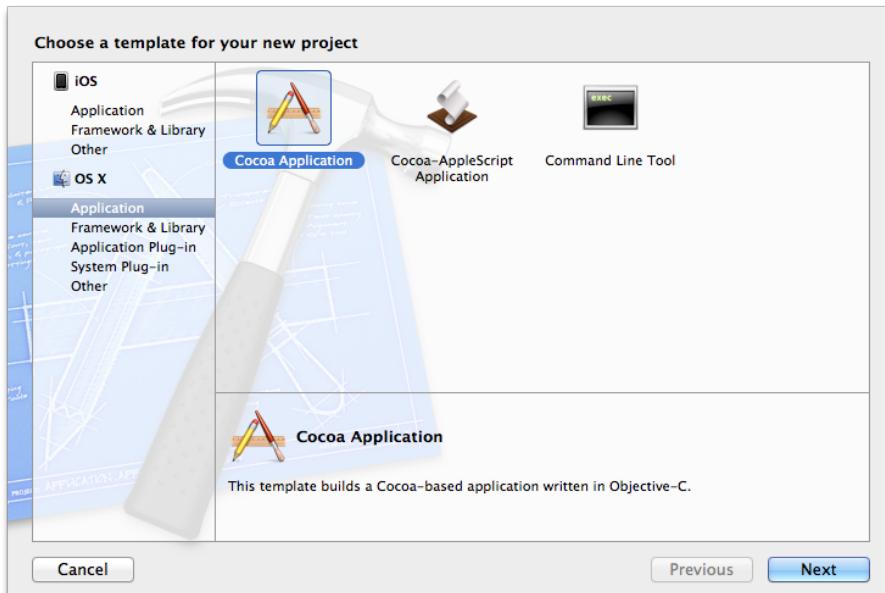
O **NSButton** tem um par destino-ação. Quando clicado, ele enviará a mensagem **addTask:** à **BNRDocument**. A classe **BNRDocument** também será a fonte de dados para a **NSTableView**.

No último capítulo, você criou seus objetos de visão e fez essas conexões de forma programática. Neste capítulo, você vai usar o Interface Builder com a ferramenta do tipo arrastar e soltar do Xcode para criar interfaces de usuário.

Introdução ao TahDoodle

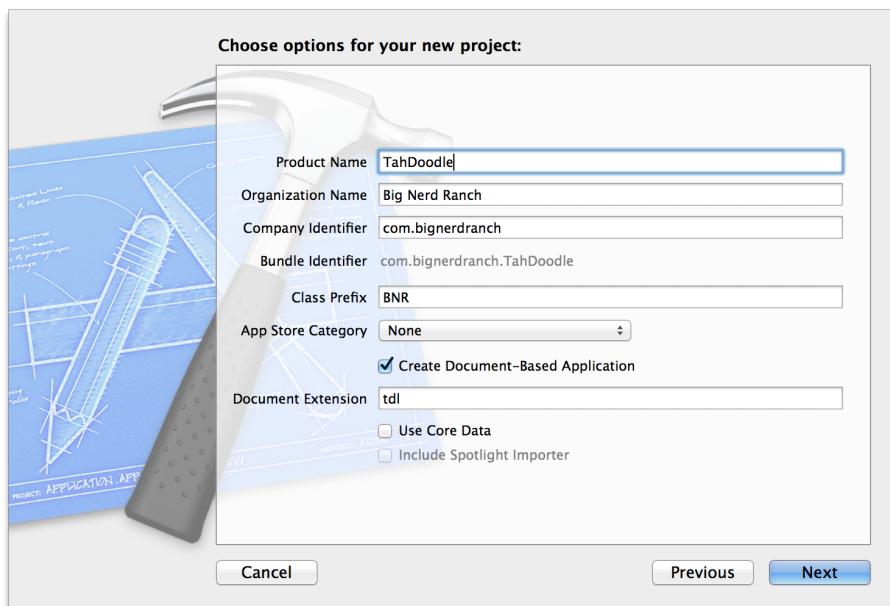
No Xcode, escolha File → New → Project... Na seção OS X (não iOS), clique em Application. Entre as opções de template exibidas, selecione Cocoa Application.

Figure 32.3 Escolha o template do aplicativo em Cocoa



Na próxima janela, nomeie o projeto TahDoodle (Figure 32.4). Marque a caixa para Create a Document-Based Application. Para o Document Extension, insira tdl. Essa será a extensão do nome de arquivo usada quando os documentos TahDoodle (listas de tarefas) forem salvos.

Figure 32.4 Configuração do TahDoodle



Clique em Next e finalize a criação do seu projeto.

O template criou a classe **BNRDocument** para você. Essa classe é o controlador para o TahDoodle. Para cada lista de tarefas que o usuário abrir, haverá um instância de **BNRDocument**, que atuará como o controlador para aquela janela.

Abra o `BNRDocument.h`. Adicione duas propriedades e um método de instância:

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

@interface BNRDocument : NSDocument

@property (nonatomic) NSMutableArray *tasks;
@property (nonatomic) IBOutlet NSTableView *taskTable;
- (IBAction)addTask:(id)sender;
@end
```

Primeiramente, observe que você não declarou uma propriedade para **NSButton**. Todo o trabalho de criar e conectar esse objeto de visão será feito no Interface Builder. Você não precisa de uma propriedade para ele já que você não irá interagir com o botão de forma programática.

Em segundo lugar, você usou duas novas palavras-chave: **IBOutlet** e **IBAction**. **IBOutlet** informa ao Xcode que o ponteiro `taskTable` será atribuído no Interface Builder e não nos arquivos de código das classes. **IBAction** informa ao Xcode que o `addTask:` é um método de ação e que o par destino-ação associado será configurado no Interface Builder e não nos arquivos de código das classes.

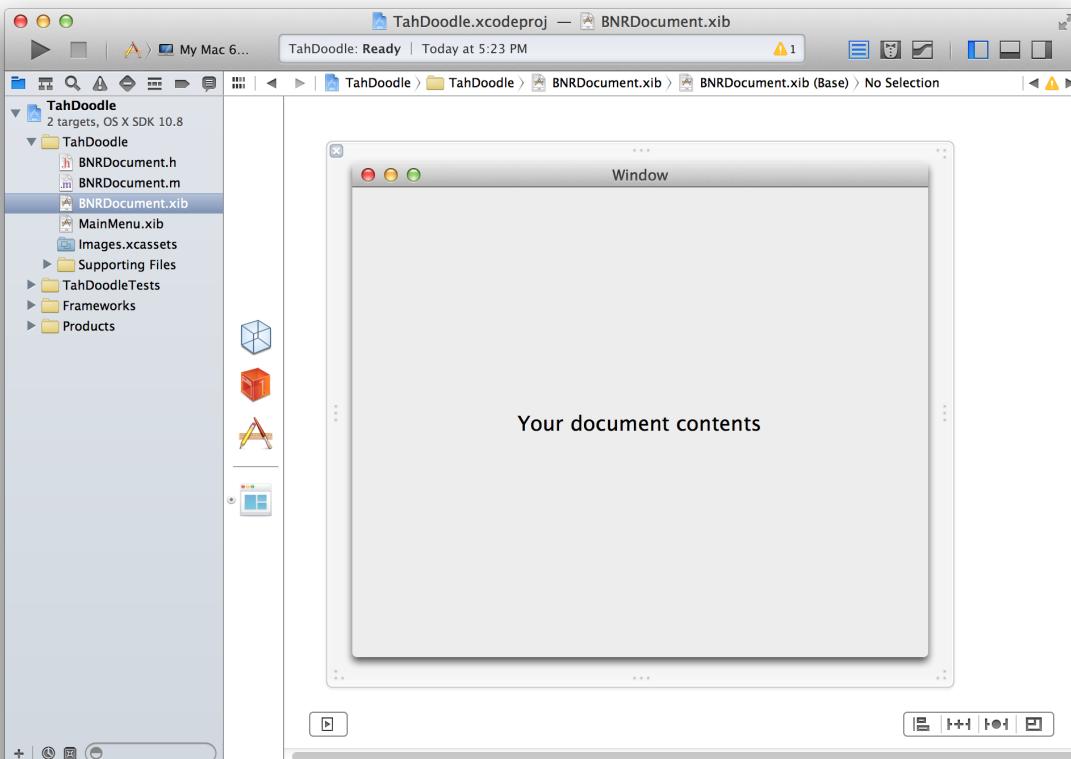
Configuração de visões no Interface Builder

No navegador do projeto, encontre e selecione o `BNRDocument.xib`. Quando você seleciona um arquivo no navegador de projetos que termina em `.xib`, o Interface Builder é aberto na área do editor. Em vez do código,

você verá uma grade de layout exibindo objetos de visão definidos naquele arquivo XIB (documento XML Interface Builder).

Agora mesmo, há dois objetos de visão definidos no `BNRDocument.xib`: uma janela e um campo de texto. Estes são instâncias de `NSWindow` e de `NSTextField`.

Figure 32.5 Conteúdo `BNRDocument.xib` atual



Se você fosse executar esse aplicativo agora, uma janela apareceria com o texto `Your document contents here` centralizado na janela.

O objeto da janela é uma instância de `NSWindow`. O template definiu esse objeto para você no `BNRDocument.xib`. Você simplesmente o utiliza como um canvas sobre o qual você cria sua interface. Você não vai interagir com ele em código no `TahDoodle`.

No `BNRDocument.xib`, clique no texto `Your document contents here` para selecionar o campo de texto. Em seguida, pressione a tecla `Delete` em seu teclado para remover esse objeto do layout.

No canto superior direito da janela Xcode, clique no botão para revelar a área de utilitários.

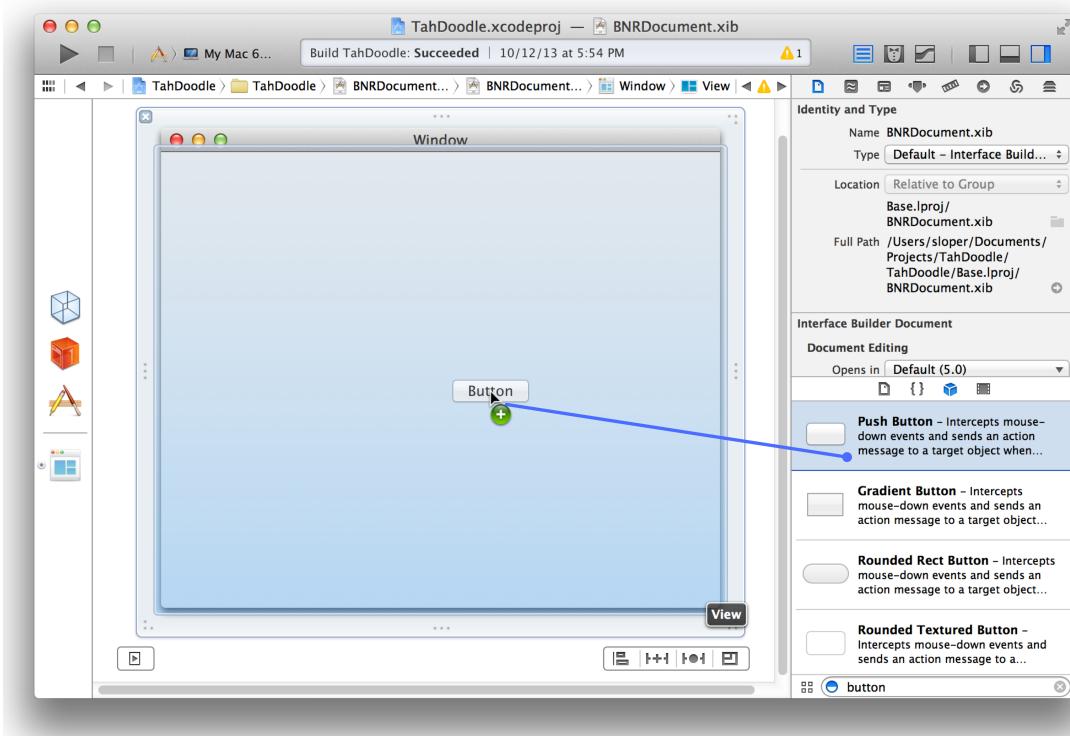
A metade inferior da área de utilitários é a *biblioteca*. A biblioteca é dividida em guias. Selecione a guia para revelar a *biblioteca de objetos*. A biblioteca de objetos apresenta os diferentes tipos de objeto que você pode arrastar e soltar na grade de layout para criar sua interface do usuário.

Configuração do botão

Na parte inferior da biblioteca, há um campo de pesquisa. Procure pelo “botão”. O primeiro item, `Push Button`, representa uma instância da classe `NSButton`.

Aqui está a parte divertida. Para criar uma instância de `NSButton` e adicioná-la ao seu layout, simplesmente arraste e solte a partir do item `Push Button` na biblioteca de objetos em qualquer lugar no objeto de janela na grade de layout.

Figure 32.6 Arrastando da biblioteca para a grade de layout



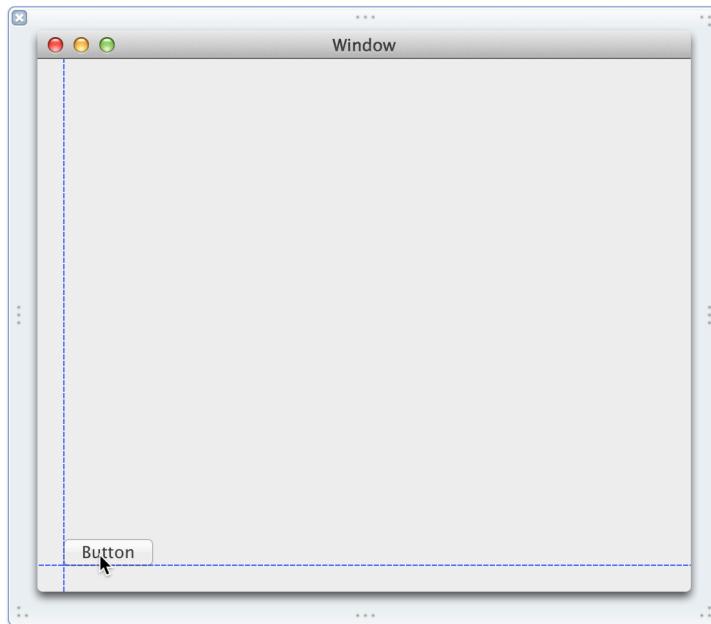
Agora, há uma instância de **NSButton** na interface do usuário do TahDoodle. Você não apenas criou um objeto de botão, mas ao arrastá-lo para o objeto de janela, você também o adicionou como uma subvisão da janela.

Ser uma subvisão da janela é importante; é o que apresenta um objeto de visão na tela quando o aplicativo é iniciado. No iTahDoodle, você adicionou o botão na janela no `BNRAppDelegate.m`:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...
    [self.window addSubview:self.insertButton];
    ...
}
```

Vamos colocar esse botão no lugar certo. Arraste o botão para o canto inferior esquerdo do objeto de janela. Ao se aproximar do canto, linhas azuis tracejadas serão exibidas. Posicione o botão dentro das linhas (Figure 32.7).

Figure 32.7 Alteração da posição do botão



As linhas tracejadas são das Diretrizes para Interface Humana da Apple, ou HIGs. As HIGs são um conjunto de regras que os desenvolvedores devem seguir ao projetar interfaces de usuário para o Mac. Também há HIGs para o iPhone e o iPad. Você pode navegar por todas essas HIGs na documentação do desenvolvedor.

Agora você precisa mudar o título do botão para Add Task. No iTahDoodle, você definiu o título do botão no `BNRAppDelegate.m`:

```
- (BOOL)application:(UIApplication *)application  
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions  
{  
    ...  
  
    [self.insertButton setTitle:@"Insert"  
                      forState:UIControlStateNormal];  
    ...  
}
```

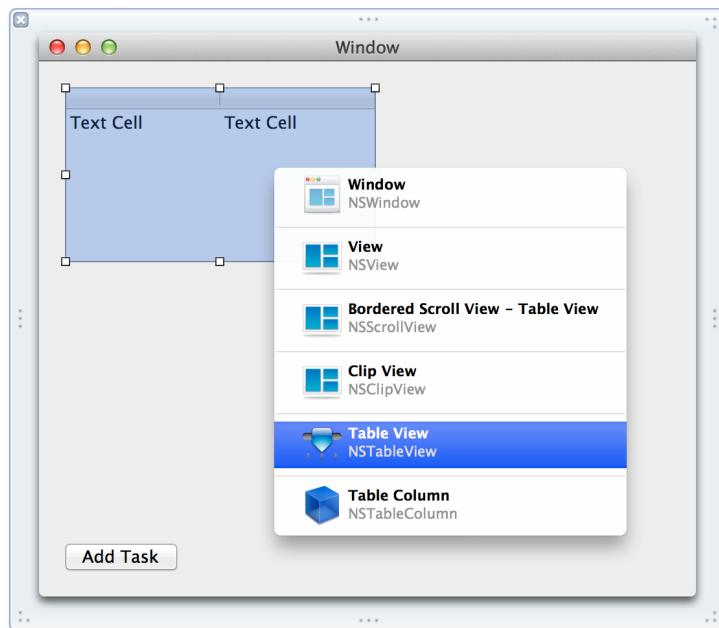
Configurar o título do botão no Interface Builder é ainda mais simples: clique duas vezes no objeto de botão, digite Add Task e pressione Return. O objeto vai se redimensionar para se ajustar ao título um pouco mais longo.

Configuração da visão de tabela

Retorne à biblioteca de objetos e procure pela “visão de tabela”. Selecione o item Table View e arraste-o para o canto superior esquerdo da janela. Mantenha-o dentro das diretrizes azuis tracejadas.

Se você vir um aviso sobre uma “visão colocada no lugar errado”, pode ignorá-lo; você vai corrigir o posicionamento das suas visões na próxima seção.

Esse objeto de visão de tabela é, na verdade, uma coleção de objetos aninhados: uma `NSScrollView`, que contém uma `NSTableView`, que, por padrão, contém duas instâncias de `NSTableColumn`. Para obter um objeto específico nessa coleção, pressione as teclas Control e Shift e, mantendo-as pressionadas, clique na visão de tabela. Você verá uma lista dos objetos e poderá selecionar o objeto no qual está realmente interessado. Selecione a `NSTableView`.

Figure 32.8 Seleção de **NSTableView**

Agora você definirá o número de visões de tabela das colunas usando outra ferramenta – o *inspetor*.

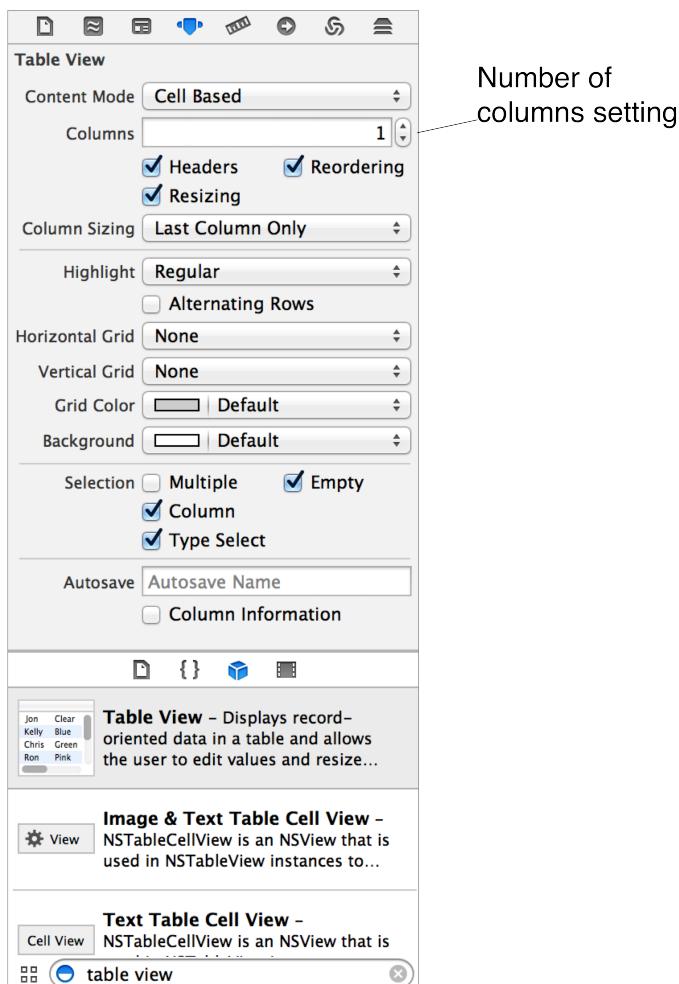
O inspetor está na área de utilitários acima da biblioteca de objetos. É a sua opção exclusiva para descobrir e configurar o objeto que está selecionado na grade de layout.

Como a biblioteca de objetos, o inspetor é organizado em um conjunto de guias, e cada guia é um inspetor diferente.

Clique na guia para acessar o *inspetor de atributos*. O inspetor de atributos é onde você pode ver e modificar os atributos do objeto.

No inspetor de atributos, localize o atributo **Columns** da **NSTableView**. Modifique a **NSTableView** para ter apenas uma coluna.

Figure 32.9 Definição do número de colunas



Agora, você precisa ajustar o tamanho da visão de tabela. Isso é fácil de fazer na grade de layout. Pressione as teclas Control-Shift na visão de tabela e selecione **NSScrollView**, que contém a **NSTableView**. Arraste o canto inferior direito da visão com rolagem de forma a preencher toda a janela. Deixe espaço para o botão e mantenha-se dentro das diretrizes azuis.

Os objetos que você acabou de criar e configurar agora estão descritos em XML no `BNRDocument.xib`. Eles serão automaticamente alocados e inicializados no tempo de execução quando uma nova janela TahDoodle for aberta e uma nova instância de **BNRDocument** for criada.

(Se você quiser ver o XML, clique com o botão direito no `BNRDocument.xib` no navegador de projetos e selecione Open As → Source Code.)

Compile e execute o TahDoodle. Você pode precisar clicar no ícone do aplicativo em seu Dock para ver a janela. Selecione File → New na barra de menu do TahDoodle ou use o atalho do teclado Command-N para abrir uma segunda janela. Cada janela é uma instância separada de **BNRDocument** e um conjunto separado de objetos de visão.

Adição de restrições de autolayout

Para controlar como sua interface do usuário será exibida quando a janela redimensionar, você pode criar várias *restrições de autolayout*. Uma restrição de autolayout especifica um relacionamento individual entre as visões em seu aplicativo.

Em um aplicativo para desktop, as restrições de autolayout são normalmente usadas para controlar como sua interface do usuário é exibida quando a janela é redimensionada. Em um aplicativo para iOS, as restrições de autolayout são normalmente usadas para controlar como sua interface de usuário é exibida nos dispositivos com diferentes tamanhos de tela ou tamanhos de fontes.

Atualmente, a interface do usuário do TahDoodle não ajusta de maneira nenhuma quando o usuário redimensiona a janela. A visão de tabela e o botão mantêm sua posição e tamanho independentemente do que o usuário fizer com a janela. (Experimente você mesmo.)

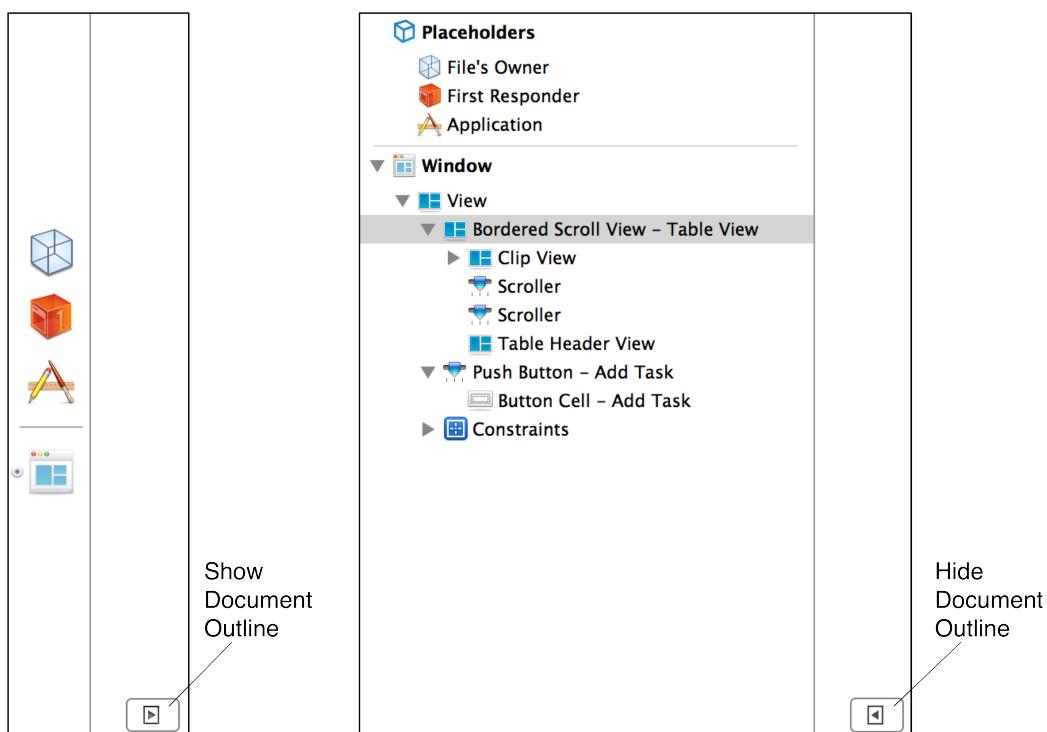
Seria muito melhor se a visão de tabela e o botão mantivessem suas posições e se a visão de tabela redimensionasse para ampliar e diminuir junto com a janela.

Vamos começar adicionando restrições à visão de tabela. Lembre-se de que a visão de tabela é uma coleção de objetos aninhados. O objeto mais distante é uma instância de **NSScrollView**. Você vai aplicar quatro restrições de autolayout para especificar o relacionamento do layout entre a visão com rolagem e a janela.

As restrições de autolayout são adicionadas individualmente, e pressionar as teclas Control-Shift para selecionar a visão com rolagem na grade de layout é cansativo. Felizmente, você também pode selecionar um objeto de visão a partir da *estrutura do documento*.

No **BNRDocument.xib**, procure a pequena seta preta apontando para a esquerda em um retângulo arredondado no canto inferior esquerdo da área de edição (Figure 32.10). Clique nessa seta para exibir a estrutura do documento. (Você pode clicar novamente para ocultar a estrutura do documento quando precisar de mais espaço para trabalhar na grade de layout.)

Figure 32.10 Exibição e ocultação da estrutura do documento



A estrutura do documento inclui uma lista hierárquica dos seus objetos de visão. Em Window, procure e selecione o objeto Bordered Scroll View – Table View. Esta é a instância de **NSScrollView** que contém o restante dos objetos que compõem a visão de tabela. Na barra de menus do Xcode, selecione Editor → Pin → Leading Space to Superview.

Leading Space to Superview é uma restrição de autolayout. A supervisão da visão com rolagem é o objeto de janela. Adicionar essa restrição assegura que a sua visão permaneça a uma distância fixada da borda esquerda da janela, independentemente do tamanho da janela.

(Na maioria dos casos, borda “leading” significará “à esquerda”. No entanto, se o seu sistema do usuário estiver configurado para uma linguagem que seja executada da direita para a esquerda, então, a borda “leading” será a borda à direita.

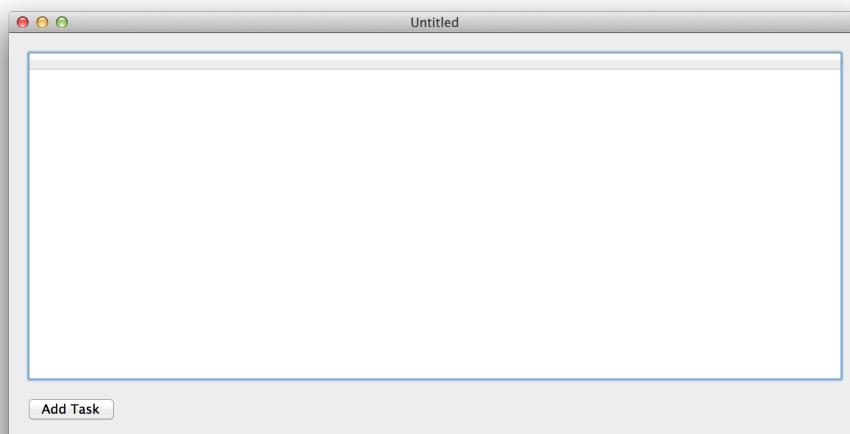
Você pode ver a nova restrição listada sob o título Constraints na estrutura do documento. Clique nessa restrição para vê-la exibida na grade de layout. Ela será exibida como um strut vermelho entre a visão com rolagem e a janela.

Selecione a visão com rolagem novamente na estrutura do documento e adicione uma segunda restrição de autolayout: Editor → Pin → Trailing Space to Superview.

Selecione a visão com rolagem novamente. Observe que os dois struts que identificam suas restrições de autolayout são vermelhos. A cor vermelha lhe informa que a visão com rolagem ainda não tem restrições suficientes para garantir sua posição quando o aplicativo estiver sendo executado. Vamos confirmar isso.

Compile e execute o TahDoodle. Arraste a borda direita da janela para ampliá-la. A visão com rolagem (e as visões que ela contém) serão expandidas para manter os relacionamentos entre a janela e suas bordas esquerda e direita.

Figure 32.11 A visão de tabela se redimensiona de acordo com as restrições horizontais

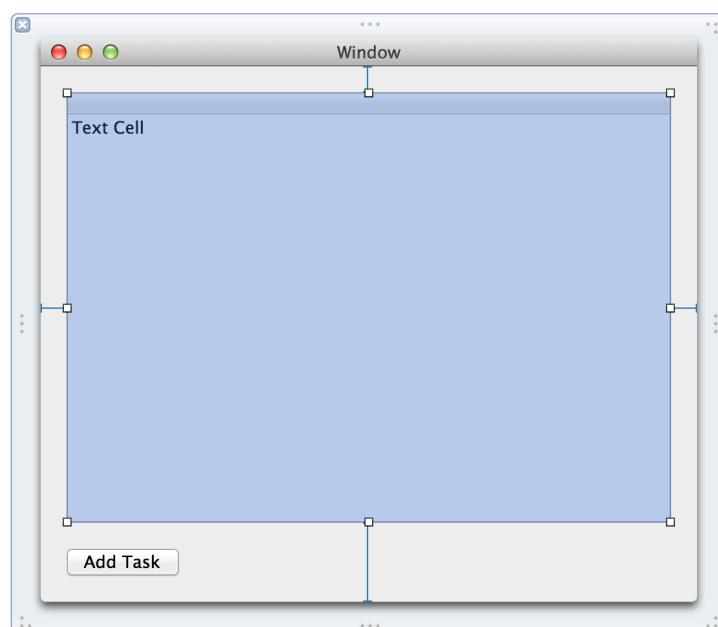


No entanto, se você alterar a altura da janela arrastando pela parte superior ou inferior, o tamanho da visão de tabela não será alterado. Você precisa de restrições para a parte superior e inferior da visão com rolagem.

Selecione a visão com rolagem e acrescente duas restrições adicionais: no menu Editor, fixe Top Space to Superview e Bottom Space to Superview.

Agora, há restrições suficientes para garantir a posição e o tamanho da visão com rolagem, independentemente do que a janela fizer. Selecione a visão com rolagem mais uma vez e você verá que os struts estão azuis agora.

Figure 32.12 Restrições satisfatórias da visão com rolagem



Compile e execute o aplicativo. A visão com rolagem vai se redimensionar à medida que a janela o fizer. No entanto, se você tornar a janela mais alta, a visão com rolagem cobrirá o botão. Seu botão precisa de suas próprias restrições de autolayout para demarcar sua posição.

Na estrutura do documento, selecione Push Button e adicione duas restrições de layout: fixe o espaço leading (à esquerda) para a supervisão e fixe o espaço inferior para a supervisão.

Compile e execute o aplicativo. A visão com rolagem vai redimensionar a janela, e o botão manterá sua posição na borda inferior à esquerda.

Criação de conexões

Criar e configurar visões não é tudo o que você pode fazer no Interface Builder. Você pode também conectar objetos de visão em seu arquivo XIB para o código do seu aplicativo. Em particular, você pode configurar pares destino-ação e atribuir ponteiros.

File's Owner

No `BNRDocument.xib`, localize o título `Placeholders` na estrutura do documento. Um `placeholder` substitui um objeto em particular que não pode ser especificado até o tempo de execução.

Um desses placeholders é o File's Owner. O File's Owner substitui o objeto que carregará o arquivo XIB como sua interface do usuário. Em seu caso, ele representa a instância de `BNRDocument`.

Definição do par destino-ação do botão

Recordando o diagrama de objetos no começo do capítulo (Figure 32.2) em que `NSButton` tem um par destino-ação: seu destino é a instância de `BNRDocument`, e sua ação é o `addTask:`.

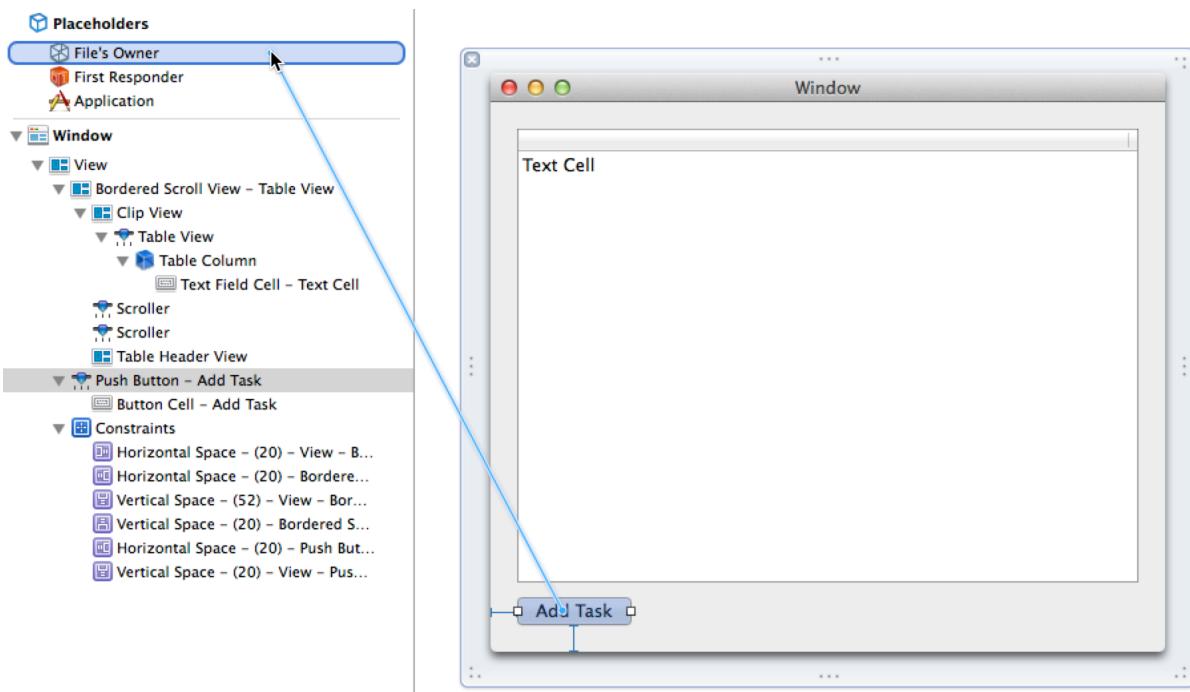
Para o `iTahDoodle`, você configurou o par destino-ação para o botão no `BNRAppDelegate.m`:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...
    [self.insertButton addTarget:self
                           action:@selector(addTask:)
                     forControlEvents:UIControlEventTouchUpInside];
    ...
}
```

Para o `TahDoodle`, você vai fazer essa conexão no Interface Builder.

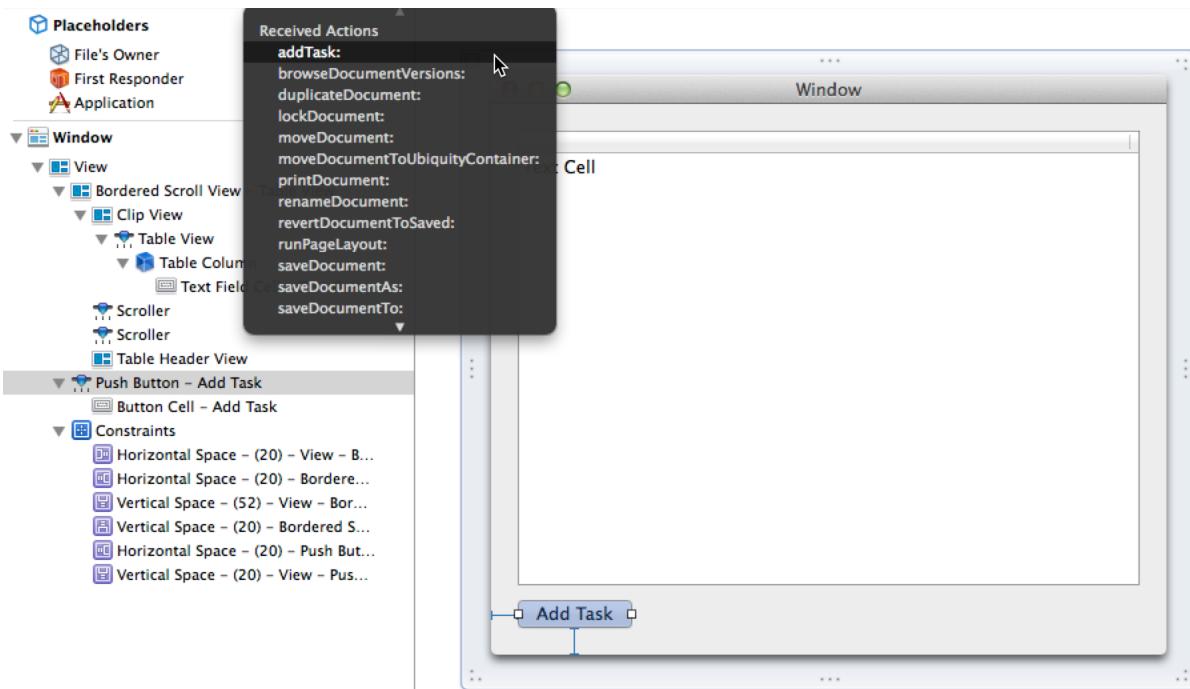
Na grade de layout, selecione o botão Add Task. Enquanto pressiona e segura a tecla Control, arraste a partir do botão para o File's Owner na estrutura do documento.

Figure 32.13 Realização de uma conexão



Solte o botão do mouse, e uma lista de métodos será exibida. Selecione **addTask:..**.

Figure 32.14 Seleção de uma ação



Pressionando a tecla Control e arrastando a partir do botão para File's Owner, você define **BNRDocument** como o destino do botão. Quando selecionou **addTask:** na lista, você definiu esse método como a ação do botão.

Observe que **addTask:** estava apenas disponível para escolha na lista de métodos porque você incluiu a palavra-chave **IBAction** em sua declaração no **BNRDocument.h**:

- (**IBAction**)**addTask:**(id)sender;

`IBAction` é um flag para o Interface Builder que diz: “Olá! Quando eu tentar conectar um par destino-ação no IB, certifique-se de incluir esse método na lista de possíveis ações.”

Veja a definição real do `IBAction`:

```
#define IBAction void
```

Lembra-se do que aprendeu sobre `#define` no Chapter 25? Essa declaração lhe informa que o `IBAction` é substituído por `void` antes de o compilador o vir. Todas as palavras-chave `IBAction` podem ser substituídas por `void` visto que não se espera que as ações invocadas pelos controles de interface do usuário tenham um valor de retorno.

Para executar e testar essa conexão, você precisa implementar `addTask:`. No navegador de projetos, selecione `BNRDocument.m`. Você acaba de sair do Interface Builder e está de volta a seu familiar editor de código.

No `BNRDocument.m`, primeiramente adicione uma pragma mark ao grupo de métodos existentes no `BNRDocument`:

```
#import "BNRDocument.h"

@implementation BNRDocument

#pragma mark - NSDocument Overrides

...
@end
```

Em seguida, implemente `addTask:` como um stub que carrega uma mensagem para o console:

```
#import "BNRDocument.h"

@implementation BNRDocument

#pragma mark - NSDocument Overrides

...
#pragma mark - Actions

- (void)addTask:(id)sender
{
    NSLog(@"Add Task button clicked!");
}

@end
```

Compile e execute o aplicativo. Clique no botão Add Task e confirme que o seu par destino-ação está funcionando conforme esperado.

Conexão da visão de tabela

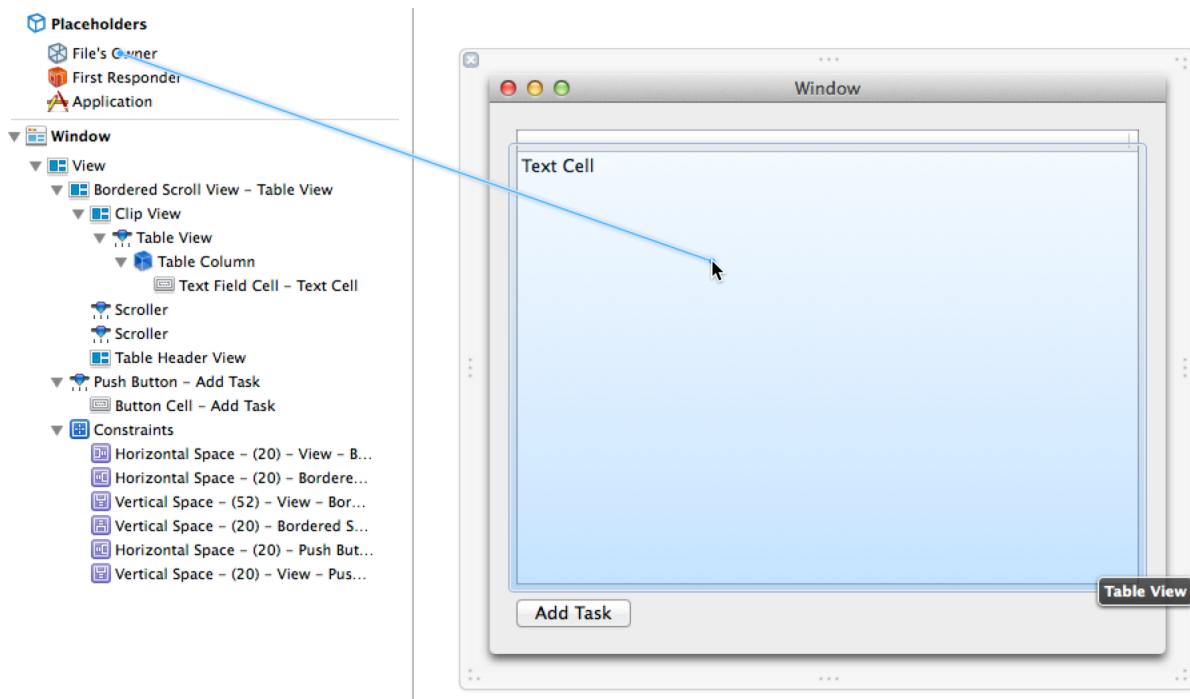
No início do capítulo, você declarou o ponteiro `taskTable` no `BNRDocument.h`:

```
@property (nonatomic) IBOutlet NSTableView *taskTable;
```

Você deseja atribuir o objeto `NSTableView` no `BNRDocument.xib` a esse ponteiro.

Abra novamente o `BNRDocument.xib`. Pressione a tecla Control e arraste do File's Owner (que está substituindo `BNRDocument`) para a visão de tabela na grade de layout. Quando soltar o botão do mouse, escolha `taskTable` na lista de conexões.

Figure 32.15 Realização de mais conexões



Agora o ponteiro `taskTable` que você declarou no `BNRDocument.h` aponta para a instância específica de `NSTableView` definida no `BNRDocument.xib`.

“Outlet” é apenas outra palavra para “ponteiro de objetos”. Lembre-se que você incluiu o `IBOutlet` ao declarar o ponteiro `taskTable`:

```
@property (nonatomic) IBOutlet NSTableView *taskTable;
```

Diferentemente do `IBAction`, o `IBOutlet` é definido de maneira a desaparecer completamente antes de o compilador o vir:

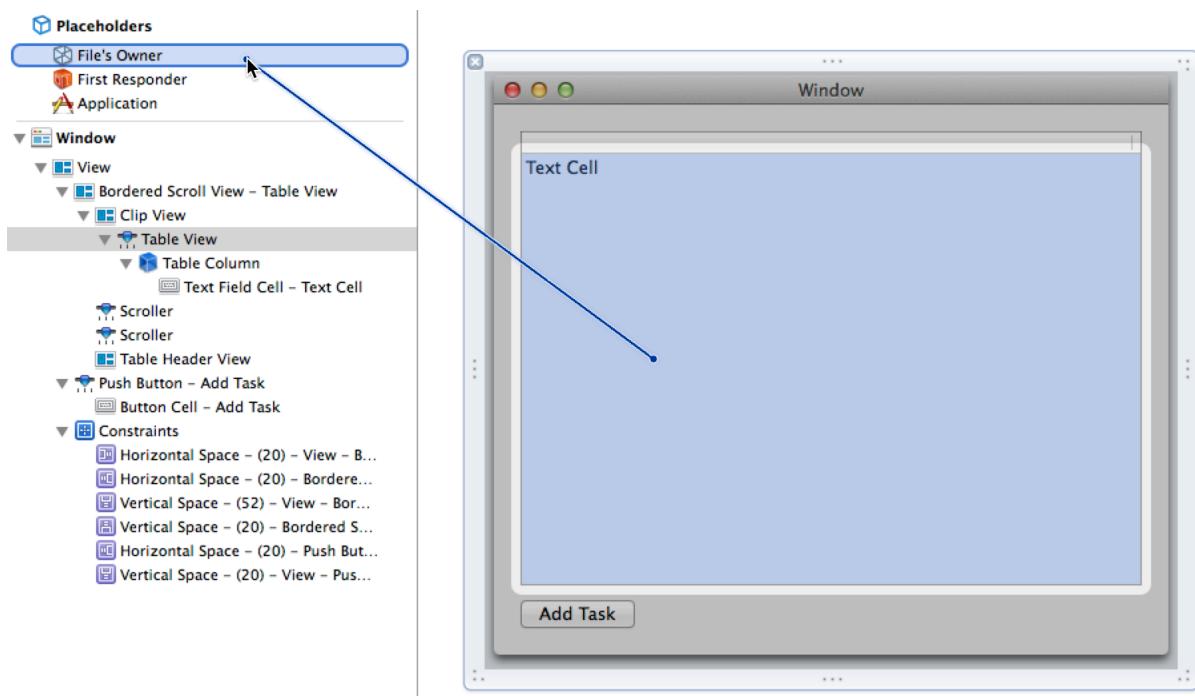
```
#define IBOutlet
```

Portanto, no tempo de compilação, todas as palavras-chave `IBOutlet` são removidas, deixando para trás suas saídas (ponteiros).

A `NSTableView` tem poucos ponteiros próprios, incluindo `dataSource`. Você deseja que a instância de `BNRDocument` seja atribuída ao ponteiro `dataSource` da visão de tabela.

Na grade de layout, pressione as teclas Control-Shift na visão de tabela para selecionar a `NSTableView`. Então, pressione a tecla Control e arraste da visão de tabela para o File's Owner. Quando soltar o botão do mouse, escolha `dataSource` na lista de conexões.

Figure 32.16 Conexão da fonte de dados da visão de tabela



Isso realiza a mesma coisa que você fez no iTahDoodle com o seguinte código:

```
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    ...
    self.taskTable.dataSource = self;
    ...
}
```

Implementação de NSTableViewDataSource

Agora que você já criou, configurou e conectou a interface do usuário do TahDoodle, a próxima etapa é fazer com que o relacionamento da fonte de dados-visão de tabela funcione. Você não pode fazer isso no Interface Builder, então chegou a hora de voltar à escrita de código.

Reabra o `BNRDocument.h` e declare que `BNRDocument` está em conformidade com o protocolo `NSTableViewDataSource`:

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

@interface BNRDocument : NSDocument
<NSTableViewDataSource>

...
@end
```

No `BNRDocument.m`, implemente os dois métodos `NSTableViewDataSource` necessários:

```
#pragma mark Data Source Methods

- (NSInteger)numberOfRowsInTableView:(NSTableView *)tv
{
    // This table view displays the tasks array,
    // so the number of entries in the table view will be the same
    // as the number of objects in the array
    return [self.tasks count];
}

- (id)tableView:(NSTableView *)tableView
objectValueForTableColumn:(NSTableColumn *)tableColumn
row:(NSInteger)row
{
    // Return the item from tasks that corresponds to the cell
    // that the table view wants to display
    return [self.tasks objectAtIndex:row];
}

- (void)tableView:(NSTableView *)tableView
setObjectValue:(id)object
forTableColumn:(NSTableColumn *)tableColumn
row:(NSInteger)row
{
    // When the user changes a task on the table view,
    // update the tasks array
    [self.tasks replaceObjectAtIndex:row withObject:object];

    // And then flag the document as having unsaved changes.
    [self updateChangeCount:NSChangeDone];
}
```

Finalmente, no `BNRDocument.m`, atualize a implementação do `addTask:` para realmente adicionar tarefas.

```
#import "BNRDocument.h"

@implementation BNRDocument

...

#pragma mark - Actions

- (IBAction)addTask:(id)sender
{
    NSLog(@"Add Task button clicked!");
    // If there is no array yet, create one
    if (!self.tasks) {
        self.tasks = [NSMutableArray array];
    }

    [self.tasks addObject:@"New Item"];

    // -reloadData tells the table view to refresh and ask its dataSource
    // (which happens to be this BNRDocument object in this case)
    // for new data to display
    [self.taskTable reloadData];

    // -updateChangeCount: tells the application whether or not the document
    // has unsaved changes, NSChangeDone flags the document as unsaved
    [self updateChangeCount:NSChangeDone];
}
```

Compile e execute o `TahDoodle`. Clique no botão para adicionar uma linha à visão de tabela. Clique duas vezes na linha para editar seu conteúdo.

Salvamento e carregamento de dados

Para adicionar a capacidade de salvar e reabrir a lista de tarefas, você precisa sobrescrever dois métodos herdados da superclasse de `BNRDocument`, `NSDocument`:

```

- (NSData *)dataOfType:(NSString *)typeName
                      error:(NSError **)outError
{
    // This method is called when our document is being saved
    // You are expected to hand the caller an NSData object wrapping our data
    // so that it can be written to disk
    // If there is no array, write out an empty array
    if (!self.tasks) {
        self.tasks = [NSMutableArray array];
    }

    // Pack the tasks array into an NSData object
    NSData *data = [NSPropertyListSerialization
                    dataWithPropertyList:self.tasks
                    format:NSPropertyListXMLFormat_v1_0
                    options:0
                    error:outError];

    // Return the newly-packed NSData object
    return data;
}

- (BOOL)readFromData:(NSData *)data
                 ofType:(NSString *)typeName
                  error:(NSError **)outError
{
    // This method is called when a document is being loaded
    // You are handed an NSData object and expected to pull our data out of it
    // Extract the tasks
    self.tasks = [NSPropertyListSerialization
                  propertyListWithData:data
                  options:NSPropertyListMutableContainers
                  format:NULL
                  error:outError];

    // return success or failure depending on success of the above call
    return (self.tasks != nil);
}

```

Observe que você está implementando um método que utiliza uma `NSError**`. Nesse caso, você está simplesmente retornando à classe `NSError` gerada por `propertyListWithData:options:format:error:`, mas você também poderia criar e retornar uma nova `NSError`, dependendo da natureza da falha.

Compile e execute o aplicativo novamente. Adicione algumas tarefas à lista. Salve e feche a lista (usando os familiares comandos do menu ou atalhos do teclado), e, em seguida, reabra-a. Parabéns! O TahDoodle está concluído.

Então, quando você deve usar o Interface Builder para criar sua interface do usuário e quando você deve definir visões de forma programática? Em circunstâncias simples, ambos funcionarão. A interface do iTahDoodle poderia ter sido criada usando o Interface Builder; as visões do TahDoodle poderiam ter sido criadas de forma programática.

No entanto, em geral, quanto mais complexa for sua interface do usuário, mais sentido fará o uso do Interface Builder.

Agora que você viu mais sobre o Xcode, consulte o encarte do verso deste livro. Esse cartão contém os atalhos do teclado para navegar pelo Xcode. Conforme for utilizando o Xcode, use esse encarte para encontrar os atalhos que permitirão que você economize tempo e cliques.

Desafio

Adicione um botão Delete Selected Item que exclui a tarefa atualmente selecionada.

Part V

Objective-C avançado

Agora você já sabe o suficiente sobre o Objective-C para iniciar com a programação para iOS ou Cocoa. Mas não tenha pressa. Estes próximos capítulos fornecem uma discussão clara das técnicas e conceitos que serão úteis no seu primeiro ano como programador de Objective-C.

33

init

Na classe **NSObject**, há um método chamado **init**. O uso de **init** será semelhante ao seguinte:

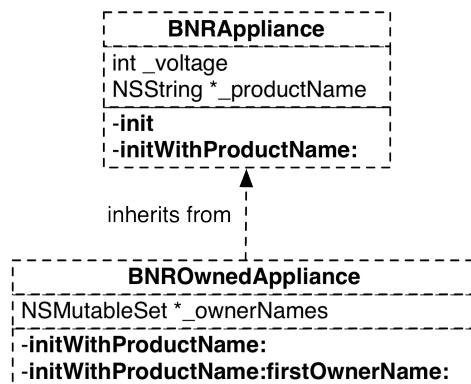
```
NSMutableArray *things = [[NSMutableArray alloc] init];
```

Você envia a mensagem **init** para a nova instância, de modo que ela possa inicializar suas variáveis de instância para valores utilizáveis; o **alloc** cria o espaço para um objeto, e o **init** faz com que o objeto fique pronto para trabalhar. Observe que **init** é um método de instância que retorna o endereço do objeto inicializado. Ele é o *inicializador* para **NSObject**. Este capítulo tratará sobre como escrever inicializadores.

Escrevendo métodos init

Crie um novo projeto: uma Foundation Command Line Tool chamada Appliances. Neste programa, você criará duas classes: **BNRAppliance** e **BNROwnedAppliance** (uma subclasse de **BNRAppliance**). Uma instância de **BNRAppliance** terá uma **productName** e uma **voltage**. Uma instância de **BNROwnedAppliance** também terá um conjunto que contém os nomes de seus proprietários.

Figure 33.1 **BNRAppliance** e sua subclasse, **BNROwnedAppliance**



Crie um novo arquivo: uma subclasse de **NSObject** chamada **BNRAppliance**. No **BNRAppliance.h**, crie declarações de propriedade para **productName** e **voltage**:

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

@interface BNRAppliance : NSObject
@property (nonatomic, copy) NSString *productName;
@property (nonatomic) int voltage;
@end
```

Você criará uma instância de **BNRAppliance** desta maneira:

```
BNRAppliance *a = [[BNRAppliance alloc] init];
```

Observe que como **BNRAppliance** não implementa um método **init**, ela executará o método **init** definido em **NSObject**. Quando isso ocorre, todas as variáveis de instância específicas à **BNRAppliance** são zeradas. Portanto, a **productName** de uma nova instância de **BNRAppliance** será **nil** e **voltage** será zero.

Um método **init** básico

Em alguns casos, um valor inicial igual a zero para suas variáveis de instância pode funcionar bem. Em outros casos, no entanto, você precisará que as instâncias de sua classe sejam criadas com suas variáveis de instância inicializadas com valores diferentes de zero.

Digamos que toda instância de **BNRAppliance** deve começar seu ciclo com uma tensão igual a 120. No **BNRAppliance.m**, sobrescreva o método **init** de **NSObject** adicionando uma nova implementação de **init**.

```
- (instancetype)init
{
    // Call the NSObject's init method
    self = [super init];

    // Did it return something non-nil?
    if (self) {

        // Give voltage a starting value
        _voltage = 120;
    }

    // Return a pointer to the new object
    return self;
}
```

Agora, quando você criar uma nova instância de **BNRAppliance**, por padrão ela terá a tensão igual a 120. (Observe que isso não muda nada com relação aos métodos acessores. Depois que a instância for inicializada, ela poderá ser alterada como antes, usando **setVoltage:**.)

instancetype

Esse método **init** retorna um **instancetype**. A palavra-chave **instancetype** informa ao compilador para esperar uma instância da classe a qual o método pertence. Todos os inicializadores que você escrever ou sobrescrever devem sempre retornar **instancetype**.

Por que não retornar **BNRAppliance *** do inicializador da **BNRAppliance**? Fazer isso poderia causar problemas caso **BNRAppliance** viesse ter subclasses. Imagine que houvesse uma subclasse de **BNRAppliance** chamada **BNROven**. **BNROven** herdaria um inicializador declarado como:

```
- (BNRAppliance *)init;
```

No entanto, se uma mensagem **init** fosse enviada para uma instância de **BNROven**, uma instância de **BNROven** seria retornada. Embora uma **BNROven** seja tecnicamente uma **BNRAppliance**, a discrepância causaria problemas mais tarde que seriam difíceis de resolver. Usar **instancetype** garante que os inicializadores possam ser herdados com segurança.

Às vezes, você verá inicializadores retornando **id**. Antes do Xcode 4.3 (quando **instancetype** foi apresentado), os desenvolvedores retornaram **id** dos inicializadores. Lembre-se de que **id** significa “qualquer objeto”, então **id** também fornece flexibilidade para a criação de subclasses ao escrever seus próprios inicializadores. No entanto, **instancetype** é a melhor opção. Ele fornece flexibilidade, mas ainda permite ao compilador verificar o tipo do que é retornado em oposição ao resto do seu código.

Utilização e verificação do inicializador da superclasse

Seu método **init** começa com duas verificações:

- Na primeira linha do **init**, você configura **self** para apontar para o objeto retornado a partir do método **init** da superclasse.

- Você verifica que o inicializador da superclasse retorna um objeto válido e não um `nil`.

O que essas verificações fazem? Algumas classes têm métodos `init` divergentes. Há duas formas possíveis de divergência:

- O método `init` detecta uma otimização mais apropriada ao que pode fazer, desaloca o objeto original, aloca um objeto diferente e retorna o novo objeto.

Para atender a essa possibilidade, a Apple *exige* que você defina `self` para apontar para o objeto retornado a partir do inicializador da superclasse.

- O método `init` falha, desaloca o objeto e retorna `nil`.

Para lidar com essa possibilidade, a Apple *recomenda* que você verifique se o inicializador da superclasse retorna um objeto válido e não um `nil`. No final das contas, não há razão de realizar uma configuração personalizada em um objeto que não existe.

Na verdade, esses tipos de verificações são necessários apenas em alguns casos muito específicos. Portanto, na prática, muitos programadores de Objective-C frequentemente não realizam essa segunda verificação. No entanto, neste livro, sempre faremos as duas verificações, pois é a maneira aprovada pela Apple de implementar métodos `init`.

Métodos init que utilizam argumentos

Às vezes, um objeto não pode ser inicializado apropriadamente sem alguma informação do método que o está chamando. Por exemplo, suponha que um appliance (dispositivo) não possa funcionar sem um nome. (`nil` não conta.) Nesse caso, você precisa ser capaz de passar ao inicializador um nome para ser usado.

Não se pode fazer isso com `init`, pois `init` nunca terá argumentos. Então, é preciso criar um novo inicializador. Depois, quando outro método criar uma instância de `BNRAppliance`, ela terá a seguinte aparência:

```
BNRAppliance *a = [[BNRAppliance alloc] initWithProductName:@"Toaster"];
```

O novo inicializador de `BNRAppliance` é `initWithProductName:` e ele aceita uma `NSString` como argumento. Declare esse novo método no `BNRAppliance.h`:

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

@interface BNRAppliance : NSObject

@property (nonatomic, copy) NSString *productName;
@property (nonatomic) int voltage;
- (instancetype)initWithProductName:(NSString *)pn;

@end
```

No `BNRAppliance.m`, localize a implementação de `init`. Altere o nome do método para `initWithProductName:` e defina `productName` usando o valor passado.

```
- (instancetype)initWithProductName:(NSString *)pn
{
    // Call NSObject's init method
    self = [super init];

    // Did it return something non-nil?
    if (self) {

        // Set the product name
        _productName = [pn copy];

        // Give voltage a starting value
        _voltage = 120;
    }
    return self;
}
```

Antes de continuar, compile o projeto para garantir que a sintaxe esteja correta.

Agora, você pode criar uma instância de **BNRAppliance** com um nome específico. No entanto, se você der o **BNRAppliance.h** e o **BNRAppliance.m** a outro programador, esse programador pode não saber chamar **initWithProductName:**. E se o programador criar uma instância de **BNRAppliance** da maneira mais comum?

```
BNRAppliance *a = [[BNRAppliance alloc] init];
```

Essa não seria uma ação insensata. Como uma subclasse de **NSObject**, espera-se que uma instância de **BNRAppliance** faça tudo o que uma instância de **NSObject** pode fazer. As instâncias de **NSObject** respondem às mensagens de **init**. Entretanto, isso causa um problema, pois a linha de código acima cria uma instância de **BNRAppliance** que tem um **nil** para o nome de um produto e zero para **voltage**. Anteriormente, decidimos que toda instância de **BNRAppliance** precisa de uma tensão de 120 e de um nome real para funcionar corretamente. Como você pode impedir que isso aconteça?

A solução é simples. No **BNRAppliance.m**, adicione um método **init** para chamar **initWithProductName:** com um valor padrão para o nome.

```
- (instancetype)init
{
    return [self initWithProductName:@"Unknown"];
}
```

Observe que este novo método **init** de substituição não funciona muito – ele apenas chama o método **initWithProductName:**, que faz o trabalho mais pesado.

Para testar seus dois inicializadores, você precisará de um método **description**. Implemente **description** no **BNRAppliance.m**:

```
- (NSString *)description
{
    return [NSString stringWithFormat:@"%@: %d volts",
                           self.productName, self.voltage];
}
```

Agora, no **main.m**, exerçite um pouco a classe:

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

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

        BNRAppliance *a = [[BNRAppliance alloc] init];
        NSLog(@"%@", a);
        [a setProductName:@"Washing Machine"];
        [a setVoltage:240];
        NSLog(@"%@", a);

    }
    return 0;
}
```

Compile e execute o programa.

Utilização de acessores

Você tem um inicializador perfeitamente bom para **BNRAppliance**, mas vamos reservar um momento para observar uma variação que você verá no código de outras pessoas. Geralmente, fazemos uma atribuição simples

em um inicializador, mas muitos programadores usarão os métodos acessores. Altere `initWithProductName:` para fazer isso:

```
- (instancetype)initWithProductName:(NSString *)pn
{
    // Call NSObject's init method
    self = [super init];

    // Did it return something non-nil?
    if (self) {

        // Set the product name
        [self setProductName:pn];

        // Give voltage a starting value
        [self setVoltage:120];
    }
    return self;
}
```

Na maioria dos casos, há poucas razões para fazer muitas atribuições, mas isso gera um excelente argumento. O argumento seria semelhante ao seguinte: o cara da atribuição diz: “Você não pode usar um método acessor em um método `init!` O acessor assume que o objeto esteja pronto para funcionar, mas ele não estará pronto até *após* a conclusão do método `init.`”. Depois, o cara do método acessor diz: “Ah, pare com isso! No mundo real, isso quase nunca é um problema. Meu método acessor pode cuidar de muita coisa para mim. Uso meu acessor sempre que configuro essa variável.”.

Ambas as abordagens funcionarão na grande maioria dos casos. Compile e execute o programa.

Na Big Nerd Ranch, costumamos definir as variáveis de instância diretamente, e normalmente realizamos a atribuição e verificamos o inicializador da superclasse em uma linha. Faça isso em seu método `initWithProductName::`

```
- (instancetype)initWithProductName:(NSString *)pn
{
    if (self = [super init]) {

        _productName = [pn copy];
        _voltage = 120;
    }
    return self;
}
```

Vários inicializadores

Crie um novo arquivo: uma subclasse de `BNRAppliance` chamada `BNROwnedAppliance`. No `BNROwnedAppliance.h`, adicione um conjunto mutável de nomes de proprietários e três métodos.

```
#import "BNRAppliance.h"

@interface BNROwnedAppliance : BNRAppliance
@property (readonly) NSSet *ownerNames;
- (instancetype)initWithProductName:(NSString *)pn
                           firstOwnerName:(NSString *)n;
- (void)addOwnerName:(NSString *)n;
- (void)removeOwnerName:(NSString *)n;

@end
```

Observe que um dos métodos que você declarou é um inicializador que usa dois argumentos.

Implemente os métodos no `BNROwnedAppliance.m`:

```

#import "BNROwnedAppliance.h"

@interface BNROwnedAppliance ()
{
    NSMutableSet *_ownerNames;
}
@end

@implementation BNROwnedAppliance

- (instancetype)initWithProductName:(NSString *)pn
                           firstOwnerName:(NSString *)n
{
    // Call the superclass's initializer
    if (self = [super initWithProductName:pn])

        // Create a set to hold owners names
        _ownerNames = [[NSMutableSet alloc] init];

        // Is the first owner name non-nil?
        if (n) {
            [_ownerNames addObject:n];
        }
    }
    // Return a pointer to the new object
    return self;
}

- (void)addOwnerName:(NSString *)n
{
    [_ownerNames addObject:n];
}

- (void)removeOwnerName:(NSString *)n
{
    [_ownerNames removeObject:n];
}

- (NSSet *)ownerNames
{
    return [_ownerNames copy];
}

@end

```

Observe que essa classe não inicializa `voltage` ou `productName`. É o método `initWithProductName:` em `BNRAppliance` que cuida dessas variáveis. Ao criar uma subclasse, geralmente é preciso apenas inicializar as variáveis de instância que a subclasse incluiu; deixe que a superclasse cuide das variáveis de instância que ela incluiu.

Agora, no entanto, você se encontra na mesma situação que ocorreu com `BNRAppliance` e o inicializador de sua superclasse, `init`. No momento, um de seus colegas de trabalho poderia criar um grande bug com esta linha de código:

```
OwnedAppliance *a = [[OwnedAppliance alloc] initWithProductName:@"Toaster"];
```

Esse código fará com que o método `initWithProductName:` em `BNRAppliance` seja executado. Esse método não sabe nada sobre o conjunto de `ownerNames`, o que significa que `ownerNames` não será inicializada corretamente para esta instância de `BNROwnedAppliance`.

A correção para isso é a mesma de antes. No `BNROwnedAppliance.m`, adicione uma implementação do inicializador da superclasse `initWithProductName:` que chama `initWithProductName:firstOwnerName:` e passa um valor padrão para `firstOwnerName`.

```

- (instancetype)initWithProductName:(NSString *)pn
{
    return [self initWithProductName:pn firstOwnerName:nil];
}

```

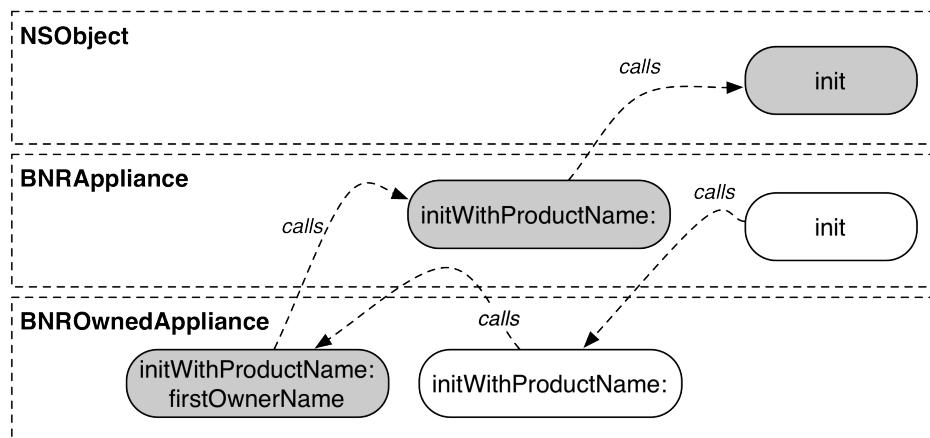
Momento de teste: você também precisa implementar `init` em `BNROwnedAppliance`? Não. Nesse ponto, o seguinte código funcionará bem:

```
OwnedAppliance *a = [[OwnedAppliance alloc] init];
```

Por quê? Não há implementação de `init` em `BNROwnedAppliance`, portanto, essa linha irá disparar o método `init` implementado em `BNRAppliance`, que chama `[self initWithProductName:@"Unknown"]`. `self` é uma instância de `BNROwnedAppliance`, então, ela chama `initWithProductName:` em `BNROwnedAppliance`, que chama `[self initWithProductName:pn firstOwnerName:nil]`.

O que você acabou de fazer é uma cadeia de inicializadores que chamam outros inicializadores.

Figure 33.2 Cadeia de inicializadores



Observe que a Figure 33.2 mostra um inicializador sombreado para cada classe. O inicializador é o *inicializador designado* para aquela classe. `init` é o inicializador designado para `NSObject`, `initWithProductName:` é o inicializador designado para `BNRAppliance`, e `initWithProductName: firstOwnerName:` é o inicializador designado para `BNROwnedAppliance`.

O inicializador designado atua como um ponto de afunilamento. Uma classe tem apenas um método inicializador designado. Se a classe tiver outros inicializadores, então a implementação desses inicializadores deverá chamar (direta ou indiretamente) o inicializador designado.

Ao criar uma classe cujo inicializador designado tenha um nome diferente do inicializador designado de sua superclasse (como você fez em `BNRAppliance` e `BNROwnedAppliance`), você terá a responsabilidade de documentar isso no arquivo de cabeçalho. Adicione o comentário apropriado no `BNRAppliance.h`:

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

@interface BNRAppliance : NSObject

@property (nonatomic, copy) NSString *productName;
@property (nonatomic) int voltage;

// The designated initializer
- (instancetype)initWithProductName:(NSString *)pn;

@end

e no BNROwnedAppliance.h:

#import "BNRAppliance.h"

@interface BNROwnedAppliance : BNRAppliance

@property (readonly) NSSet *ownerNames;

// The designated initializer
- (instancetype)initWithProductName:(NSString *)pn
    firstOwnerName:(NSString *)n;
- (void)addOwnerName:(NSString *)n;
- (void)removeOwnerName:(NSString *)n;

@end
```

Assim, chegamos às regras que todos os modernos programadores de Objective-C seguem ao escrever inicializadores:

- Se uma classe tiver vários inicializadores, apenas um fará o trabalho real. Esse método é conhecido como inicializador designado. Todos os outros inicializadores chamarão, direta ou indiretamente, o inicializador designado.
- O inicializador designado chamará o inicializador designado da superclasse antes de inicializar suas variáveis de instância.
- Se o inicializador designado de sua classe tiver um nome diferente do inicializador designado de sua superclasse, você deverá substituí-lo para que ele possa chamar o novo inicializador designado.
- Se você tiver vários inicializadores, documente claramente no cabeçalho qual é o inicializador designado.

Métodos init fatais

De vez em quando, no entanto, você não consegue sobrescrever, de modo seguro, o inicializador designado da superclasse. Digamos que você esteja criando uma subclasse de **NSObject** chamada **BNRWallSafe** e que seu inicializador designado seja **initWithSecretCode:**. Entretanto, ter um valor padrão para **secretCode** não é seguro o suficiente para seu aplicativo. Isso significa que o padrão que estamos usando – sobrescrever **init** para chamar o novo inicializador designado da classe com valores padrão – não é aceitável.

Então, o que fazer? Uma instância de **BNRWallSafe** ainda responderá a uma mensagem de **init**. Alguém pode facilmente fazer isto:

```
BNRWallSafe *ws = [[BNRWallSafe alloc] init];
```

A melhor coisa a fazer é sobrescrever o inicializador designado da superclasse de uma forma que ele permita que os desenvolvedores saibam que cometem um erro e que informe a eles como corrigi-lo:

```
- (instancetype)init
{
    [NSError exception raise:@"BNRWallSafeInitialization"
                      format:@"Use initWithSecretCode:, not init"];
}
```

Disparar uma exceção como essa causará uma falha no programa. No resultado do console, os desenvolvedores irão ver seus erros que levaram à falha.

Mais sobre propriedades

Por enquanto, você usou as propriedades em muitos programas. Neste capítulo, você aprenderá mais algumas coisas sobre as propriedades e o que você pode fazer elas fazerem.

Mais sobre os atributos de propriedade

Primeiramente, vamos analisar os diferentes atributos que você pode usar para controlar como os acessores serão criados.

Mutabilidade

Uma propriedade pode ser declarada `readonly` ou `readwrite`. O padrão é `readwrite` (leitura e gravação), que indica que tanto um método setter quanto um getter são criados. Se você não quiser que um método setter seja criado, será necessário marcar a propriedade como `readonly` (somente leitura):

```
@property (readonly) int voltage;
```

Especificadores de ciclo de vida

Uma propriedade também pode ser declarada como `unsafe_unretained` (insegura não retida), `assign` (atribuir), `strong` (forte), `weak` (fraca) ou `copy` (cópia). Essa opção determina como o setter lida com o gerenciamento de memória da propriedade.

`assign` é o padrão para tipos não-objeto, e o mais simples: ele simplesmente atribui à propriedade o valor passado. Imagine esta declaração e esta definição:

```
@property (assign) int averageScore;
// "@property int averageScore" would also work here
```

Isso resultaria em um método setter muito equivalente a:

```
- (void)setAverageScore:(int)d
{
    _averageScore = d;
}
```

Em `BNRAppliance`, `voltage` é uma propriedade atribuída. Você sempre usará `assign` para propriedades que contêm tipos não-objetos. Já que ele é o padrão para tipos não-objeto, você não precisa adicioná-lo a sua declaração de propriedade.

Conforme você aprendeu no Chapter 23, `strong` garantirá que uma referência forte seja mantida para o objeto passado. Ele também permitirá a propriedade do objeto antigo (que então se desalocará caso não tenha nenhum outro proprietário). Para propriedades de objeto, `strong` é o padrão para ponteiros de objeto e isso é normalmente o que você quer.

`weak` não implica na propriedade do objeto apontado. Se esse objeto for desalocado, a propriedade será definida como `nil`. Esse é um recurso perfeito que o mantém protegido de *ponteiros pendentes*. Um ponteiro pendente aponta para um objeto que não existe mais. Geralmente, enviar uma mensagem a um ponteiro pendente gera uma falha em seu programa.

Propriedades `unsafe_unretained`, tal como propriedades `fracas`, não implicam em propriedade. No entanto, uma propriedade `unsafe_unretained` não é automaticamente definida para `nil` quando o objeto para o qual ela aponta é desalocado.

copy forma uma referência forte para uma cópia do objeto passado. Porém, um detalhe aqui que grande parte das pessoas não entende ...

copy

A opção copy faz uma cópia de um objeto e, em seguida, altera o ponteiro para se referir a essa cópia. Imagine que você tivesse uma declaração e uma definição de propriedade como esta:

```
@property (copy) NSString *lastName;
```

O método setter gerado ficaria muito parecido com o seguinte:

```
- (void)setLastName:(NSString *)d
{
    _lastName = [d copy];
}
```

O uso do atributo copy é mais comum com tipos de objetos que têm subclasses mutáveis. Por exemplo, **NSString** contém uma subclasse chamada **NSMutableString**. Suponha que possa ser passada uma string mutável a seu método **setLastName::**:

```
// Create a mutable string
NSMutableString *x = [[NSMutableString alloc] initWithString:@"Ono"];

// Pass it to setLastName:
[myObj setLastName:x];

// 'copy' prevents this from changing the lastName
[x appendString:@" Lennon"];
```

E se o objeto passado *não* for mutável? É desnecessário fazer uma cópia de um objeto imutável. O método **copy** simplesmente chama **copyWithZone:** e passa nil como o argumento. Por exemplo, na **NSString**, o método **copyWithZone:** é substituído desta forma:

```
- (id)copyWithZone:(NSZone *)z
{
    return self;
}
```

Ou seja, não é feita uma cópia. (Observe que **NSZone** e o zoneamento de memória, em geral, não são aprovados – recursos que são traços da programação em Cocoa –, desse modo, não iremos nos aprofundar neles. No entanto, **copyWithZone:** ainda tem alguma utilização, e seu uso não foi totalmente descartado.)

Para objetos para os quais há versões mutáveis e imutáveis, o método **copy** retorna uma cópia imutável. Por exemplo, **NSMutableString** contém um método **copy** que retorna uma instância de **NSString**. Se você quiser que a cópia seja um objeto mutável, use o método **mutableCopy**.

Não há um especificador de ciclo de vida de propriedade chamado **mutableCopy**. Se quiser que seu setter defina a propriedade como uma cópia mutável de um objeto, você mesmo deverá implementar o método setter de modo que ele chame o método **mutableCopy** no objeto de entrada. Por exemplo, na **BNROwnedAppliance**, você pode criar um método **setOwnerNamesInternal::**:

```
- (void)setOwnerNamesInternal:(NSSet *)newNames
{
    _ownerNamesInternal = [newNames mutableCopy];
}
```

Mais informações sobre cópia

A maioria das classes no Objective-C que não vem da Apple não tem método **copyWithZone::**. Os programadores de Objective-C fazem muito menos cópias do que você pensa.

Curiosamente, os métodos **copy** e **mutableCopy** são definidos na **NSObject** desta maneira:

```

- (id)copy
{
    return [self copyWithZone:NULL];
}

- (id)mutableCopy
{
    return [self mutableCopyWithZone:NULL];
}

```

Portanto, se você tiver um código como este:

```
BNRAppliance *b = [[BNRAppliance alloc] init];
BNRAppliance *c = [b copy];
```

Obterá um erro como este:

```
- [BNRAppliance copyWithZone:]: unrecognized
selector sent to instance 0x100110130
```

Os métodos **copyWithZone:** e **mutableCopyWithZone:** estão declarados nos protocolos **NSCopying** e **NSMutableCopying**, respectivamente. Muitas das classes no framework Foundation estão em conformidade com um ou dois desses protocolos. Você pode descobrir quais protocolos uma classe está em conformidade em sua referência de classe na documentação do desenvolvedor.

Se você deseja que suas classes estejam em conformidade com o especificador de ciclo de vida de propriedade `copy`, então, você deve se certificar de que elas estão em conformidade com o protocolo **NSCopying**.

Recomendações sobre atômico vs. não atômico

Este é um livro de introdução sobre programação e a opção `atomic/nonatomic` está relacionada a um tópico relativamente avançado conhecido como multithreading (multisegmentação). Isto é o que você precisa saber: a opção `nonatomic` fará com que seu método setter seja executado um pouco mais rápido. Se você observar os cabeçalhos do UIKit da Apple, verá que toda propriedade é marcada como `nonatomic`. Faça sempre com que suas propriedades `readwrite` sejam `nonatomic` também.

(Faço essa recomendação a todo mundo. No entanto, em todo grupo, sempre há alguém que irá questionar, mesmo pelo pouco que conhece do assunto. Essa pessoa diria: “Mas, ao fazer com que meu aplicativo seja multithread, precisarei da proteção que os métodos setter atômicos me proporcionam.” E eu *diria*: “Não acho que você escreverá códigos multithread tão cedo. E, quando você o fizer, não acho que os métodos setter atômicos irão ajudá-lo.” Mas o que eu realmente direi será: “OK, então mantenha seus setters como `atomic`.” Você não pode dizer a alguém algo que essa pessoa não está pronta para ouvir.)

Infelizmente, o padrão para propriedades é `atomic`, então, você precisará marcar explicitamente cada uma de suas propriedades `nonatomic`.

Implementação de métodos acessores

Por padrão, o compilador sintetiza os métodos acessores para qualquer propriedade que você declarar. Normalmente, as implementações do método acessor são simples e, portanto, bem ajustadas para serem controladas pelo compilador.

No entanto, há momentos em que você precisa que o acessor faça coisas não muito usuais. Quando este for o caso, você pode implementar o acessor você mesmo no arquivo de implementação.

Existem dois casos razoáveis para implementar um acessor você mesmo:

- Você precisa atualizar a interface do usuário do aplicativo quando a alteração ocorrer.
- Você precisa atualizar algumas informações em cache quando a alteração ocorrer.

Por exemplo, digamos que você declarou uma propriedade em um arquivo de cabeçalho:

```
@property (nonatomic, copy) NSString* currentState;
```

Quando um objeto chama o método **setCurrentState:**, você deseja que esse método faça mais do que simplesmente alterar o valor da propriedade. Neste caso, você pode implementar explicitamente o setter.

```
- (void)setCurrentState:(NSString *)currentState
{
    _currentState = [currentState copy];
    // Some code that updates UI
    ...
}
```

O compilador verá sua implementação do **setCurrentState:** e não criará um setter para você. Ele ainda criará o método getter **currentState**.

Se você declarar uma propriedade e implementar os dois acessores você mesmo, o compilador *não* sintetizará uma variável de instância.

Se você ainda desejar uma variável de instância (e você normalmente vai querer), precisará criá-la por conta própria ao adicionar uma instrução **@synthesize** para a implementação da classe.

```
#import "Badger.h"

@interface Badger : NSObject
@property (nonatomic) Mushroom *mushroom;
@end

@implementation Badger;

@synthesize mushroom = _mushroom;

- (Mushroom *)mushroom
{
    return _mushroom;
}

- (void)setMushroom:(Mushroom *)mush
{
    _mushroom = mush;
}

...
```

A instrução **@synthesize** informa ao compilador que uma variável de instância chamada **_mushroom** é a variável de apoio para os métodos **mushroom** e **setMushroom:** e que a variável de instância precisa ser criada caso ela ainda não exista.

Se você deixar de lado a instrução **@synthesize** neste caso, o compilador poderá reclamar que **_mushroom** está indefinida.

Quando você declara uma propriedade **readonly**, o compilador automaticamente sintetiza apenas um método getter e uma variável de instância. Assim, se você mesmo implementar o método getter para uma propriedade **readonly**, o efeito é o mesmo que implementar os dois acessores para uma propriedade **readwrite**. O compilador não sintetizará uma variável de instância e você precisará sintetizá-la você mesmo.

Você pode estar pensando: “por que declarar uma propriedade nesses casos?”. Declarar a propriedade ainda é um atalho eficiente para as declarações de acessor e leva à consistência visual em seu código.

Codificação de chave-valor

Codificação de chave-valor é a capacidade de ler e definir uma propriedade usando seu nome. Os métodos de codificação de chave-valor são definidos na **NSObject**. Desse modo, todo objeto conta com essa capacidade.

Abra o `main.m` e localize a linha:

```
[a setProductName:@"Washing Machine"];
```

Reescreva a mesma linha para usar a codificação de chave-valor:

```
[a setValue:@"Washing Machine" forKey:@"productName"];
```

Nesse caso, o método **setValue:forKey:**, conforme definido na **NSObject**, procurará um método setter chamado **setProductName:**. Se o objeto não tiver um método **setProductName:**, ele acessará a variável de instância diretamente.

Também é possível ler o valor de uma variável usando a codificação de chave-valor. Adicione uma linha no `main.m` que exiba o nome do produto:

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

        BNRAppliance *a = [[BNRAppliance alloc] init];
        NSLog(@"%@", a);
        [a setValue:@"Washing Machine" forKey:@"productName"];
        [a setVoltage:240];
        NSLog(@"%@", a);

        NSLog(@"the product name is %@", [a valueForKey:@"productName"]);
    }
    return 0;
}
```

Nesse caso, o método **valueForKey:**, conforme definido na **NSObject**, procura um acessor chamado **productName**. Se não existir nenhum método **productName**, a variável de instância será acessada diretamente.

Esse uso da palavra “key” (chave) parece incomodar alguns leitores. Você pode imaginar o problema: o engenheiro que nomeou esses métodos precisou de uma palavra que pudesse significar o nome de uma variável de instância, ou de uma propriedade ou de um método. “Key” foi a palavra mais específica que ele pôde encontrar.

Se você digitar o nome da propriedade incorretamente, não verá um aviso do compilador, mas ocorrerá um erro de tempo de execução. Cometa esse erro no `main.m`:

```
NSLog(@"the product name is %@", [a valueForKey:@"productNammmme"]);
```

Ao compilar e executar, você verá em erro:

```
*** Terminating app due to uncaught exception 'NSUnknownKeyException',
reason: '[<BNRAppliance 0x100108dd0> valueForUndefinedKey:]:'
this class is not key value coding-compliant for the key productNammmme.'
```

Corrija o erro antes de prosseguir.

Por que a codificação de chave-valor é útil? Sempre que um framework padrão quiser enviar dados a seus objetos, ele usará o método **setValue:forKey:**. Sempre que um framework padrão quiser ler dados de seus

objetos, ele usará o **valueForKey:**. Por exemplo, Core Data é um framework que facilita o processo de salvar seus objetos em um banco de dados SQLite e depois fazer com que eles voltem a ficar ativos. Ele manipula seus objetos personalizados que contêm dados usando a codificação de chave-valor.

Para comprovar que a codificação de chave-valor manipulará suas variáveis mesmo que você não tenha acessores, declare explicitamente uma variável de instância para o `productName` e comente a declaração `@property` para `productName` no `BNRAppliance.h`:

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

@interface BNRAppliance : NSObject
{
    NSString *_productName;
}
// @property (nonatomic, copy) NSString *productName;
@property (nonatomic) int voltage;

// The designated initializer
- (instancetype)initWithProductName:(NSString *)pn;

@end
```

Além disso, remova todos os usos dos métodos `setProductName:` e `productName` do `BNRAppliance.m`:

```
@implementation BNRAppliance

- (instancetype)initWithProductName:(NSString *)pn
{
    if (self = [super init]) {
        _productName = [pn copy];
        _voltage = 120;
    }
    return self;
}

- (instancetype)init
{
    return [self initWithProductName:@"Unknown"];
}

- (NSString *)description
{
    return [NSString stringWithFormat:@"<%@: %d volts>", _productName, self.voltage];
}

@end
```

Compile e execute o programa. Observe que, mesmo que você não tenha métodos acessores para `productName`, a variável ainda pode ser definida e lida a partir de outros métodos. Esta é uma violação óbvia da ideia de *encapsulamento de objeto* – os métodos de um objeto são públicos, mas as variáveis de instância são frágeis e devem ser mantidas como privadas. Se a codificação de chave-valor não fosse extremamente útil, ninguém a aceitaria.

Tipos não-objetos

Os métodos de codificação de chave-valor foram criados para trabalhar com objetos. No entanto, algumas propriedades contêm um tipo não-objeto, como um `int` ou um `float`. Por exemplo, `voltage` é um `int`. Como você definiria `voltage` usando a codificação de chave-valor? É necessário usar uma `NSNumber`.

No `main.m`, altere a linha de definição de `voltage` de:

```
[a setVoltage:240];
```

para:

```
[a setValue:[NSNumber numberWithInt:240] forKey:@"voltage"];
```

Adicione um acessor explícito para `BNRAppliance.m` de modo que você possa vê-lo sendo chamado:

```

- (void)setVoltage:(int)x
{
    NSLog(@"setting voltage to %d", x);
    _voltage = x;
}

```

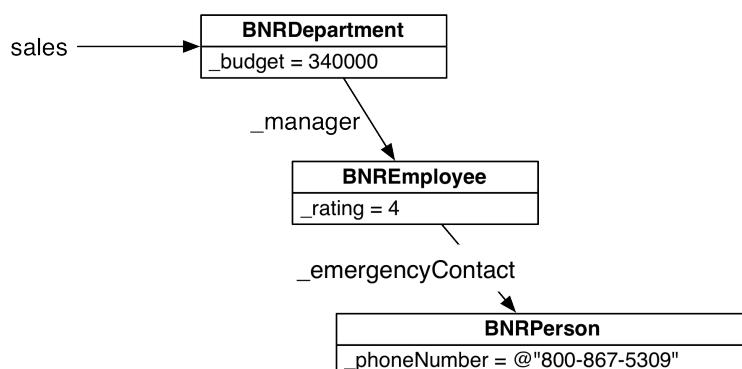
Compile e execute o programa.

De modo similar, se você solicitar o `valueForKey:@"voltage"`, obterá de volta uma `NSNumber` que contém o valor de `voltage`.

Percursos de chaves

A maioria dos aplicativos termina com um gráfico de objetos relativamente complexo. Por exemplo, você pode ter um objeto `BNRDepartment` que tem uma propriedade `manager` que é um ponteiro para um objeto `BNREmployee` que tem uma propriedade `emergencyContact` que é um ponteiro para um objeto `BNRPerson` que tem uma propriedade `phoneNumber`.

Figure 35.1 Gráfico de objetos complexo



Imagine que lhe foi perguntando o seguinte: “Qual é o número de telefone do contato de emergência do gerente do departamento de vendas?” Você poderia usar a codificação de chave-valor para percorrer esses relacionamentos de uma vez só:

```

BNRDepartment *sales = ...
BNREmployee *sickEmployee = [sales valueForKey:@"manager"];
BNRPerson *personToCall = [sickEmployee valueForKey:@"emergencyContact"];
NSString *numberToDial = [personToCall valueForKey:@"phoneNumber"];

```

No entanto, há uma maneira mais fácil. Usando um percurso de chaves, você pode fazer o sistema percorrer os relacionamentos para você. Coloque as chaves que você deseja que sejam seguidas em uma longa string separada por pontos. A ordem é importante; o primeiro relacionamento que você deseja que seja percorrido vem primeiro:

```

BNRDepartment *sales = ...
NSString *numberToDial =
    [sales valueForKeyPath:@"manager.emergencyContact.phoneNumber"];

```

Você também pode definir a propriedade no final de um percurso de chaves:

```

BNRDepartment *sales = ...
[sales setValue:@"555-606-0842" forKeyPath:@"manager.emergencyContact.phoneNumber"];

```

é equivalente a:

```

BNRDepartment *sales = ...
BNREmployee *sickEmployee = [sales valueForKey:@"manager"];
BNRPerson *personToCall = [sickEmployee valueForKey:@"emergencyContact"];
[personToCall setValue:@"555-606-0842" forKey:@"phoneNumber"];

```


36

Observação de chave-valor

A observação de chave-valor é uma técnica que lhe permite obter uma notificação quando uma propriedade específica de um objeto é alterada. Embora você não a usará todos os dias, a observação de chave-valor (ou KVO) é uma parte essencial do que torna ligações Cocoa e Core Data possíveis.

Essencialmente, você informa a um objeto: “Quero observar sua propriedade `fido`. Se ela for alterada, me informe.” Quando o método `setFido:` for chamado, você receberá uma mensagem do objeto que você está observando: “Ei, minha propriedade `fido` tem um novo valor.”

Quando você se adiciona como um observador do objeto, você especifica o nome da propriedade que está observando. Você também pode especificar algumas opções. Em específico, você pode informar ao objeto que lhe envie o valor antigo e/ou novo da propriedade quando você for notificado da alteração.

(Infelizmente, a linguagem usada ao discutir a observação de chave-valor e ao discutir `NSNotificationCenter` é muito parecida. Neste capítulo, não vamos falar sobre `NSNotification` ou `NSNotificationCenter`, mesmo se usarmos a palavra “notificar” ou “notificação”.)

Abra seu projeto `Callbacks`. Você criará um novo objeto que observará a propriedade `lastTime` da sua classe `BNRLogger`. Comece criando uma nova classe em Objective-C, uma subclasse de `NSObject` chamada `BNRObserver`.

No `main.m`, crie uma instância de `BNRObserver` e a torne um observador da propriedade `lastTime` do logger (registrar cronológico):

```
__unused NSTimer *timer =
    [NSTimer scheduledTimerWithTimeInterval:2.0
                                    target:logger
                                      selector:@selector(updateLastTime:)
                                        userInfo:nil
                                       repeats:YES];

__unused BNRObserver *observer = [[BNRObserver alloc] init];

// I want to know the new value and the old value whenever lastTime is changed
[logger addObserver:observer
    forKeyPath:@"lastTime"
    options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
    context:nil];

[[NSRunLoop currentRunLoop] run];
```

Não se esqueça de importar o `BNRObserver.h` na parte superior do `main.m`.

A seguir, implemente o método que será chamado quando `lastTime` for alterado. Abra o `BNRObserver.m` e adicione esse método:

```
#import "BNRObserver.h"

@implementation BNRObserver

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context
{
    NSString *oldValue = [change objectForKey:NSKeyValueChangeOldKey];
    NSString *newValue = [change objectForKey:NSKeyValueChangeNewKey];
    NSLog(@"Observed: %@ of %@ was changed from %@ to %@", keyPath, object, oldValue, newValue);
}

@end
```

Compile e execute o programa. A cada dois segundos, `lastTime` precisa obter um novo valor, e seu observador deve ser informado.

Utilização do contexto na KVO

Observe que quando você se registra como um observador, você pode passar um ponteiro para qualquer coisa como contexto. Quando você é notificado sobre a alteração, você receberá aquele mesmo ponteiro com a notificação. O uso mais comum disso é responder: “Essa é realmente a notificação que solicitei?” Por exemplo, a sua superclasse pode usar a KVO. Se você sobrescrever `observeValueForKeyPath:ofObject:change:context:`, como você sabe quais notificações precisam ser encaminhadas para a implementação da superclasse? O segredo é aparecer com um ponteiro exclusivo, use-o como contexto quando começar a observar e verifique-o em oposição ao contexto cada vez que você for notificado. O endereço de uma variável estática funciona bem. Portanto, se você está criando subclasses de uma classe que pode já ter sido registrada para receber notificações da KVO, a aparência será a seguinte:

```
static int contextForKVO;
...

[petOwner addObserver:self
    forKeyPath:@"fido"
    options:0
    context:&contextForKVO];
...

- (void)observeValueForKeyPath:(NSString *)keyPath
    ofObject:(id)object
    change:(NSDictionary *)change
    context:(void *)context
{
    // Is this not mine?
    if (context != &contextForKVO) {

        // Pass it on to the superclass
        [super observeValueForKeyPath:keyPath
            ofObject:object
            change:change
            context:context];
    } else {
        // Handle the change
    }
}
...
```

Disparo explícito das notificações

O sistema pode automaticamente informar ao observador se você usar o método acessor para definir a propriedade. E se você, por alguma razão, escolher não usar o acessor? Você pode explicitamente deixar que o sistema saiba que você está alterando uma propriedade. Altere seu `BNRLogger.m` de modo que ele não use o acessor para definir o `lastName`:

```
- (void)updateLastTime:(NSTimer *)t
{
    NSDate *now = [NSDate date];
    _lastTime = now;
    NSLog(@"Just set time to %@", self.lastTimeString);
}
```

Compile e execute o programa. Observe que o **Observer** nunca é informado de que `_lastTime` foi alterado.

Para corrigir isso, informe explicitamente o sistema antes e depois de alterar a propriedade:

```
- (void)updateLastTime:(NSTimer *)t
{
    NSDate *now = [NSDate date];
    [self willChangeValueForKey:@"lastTime"];
    _lastTime = now;
    [self didChangeValueForKey:@"lastTime"];
    NSLog(@"Just set time to %@", self.lastTimeString);
}
```

Compile e execute o programa. O observador precisa ser notificado corretamente agora.

Propriedades dependentes

E se você quiser observar `lastTimeString` em vez de `lastTime`? Tente fazer isso. No `main.m`, comece a observar `lastTimeString`:

```
__unused BNRObserver *observer = [[BNRObserver alloc] init];
[logger addObserver:observer
    forKeyPath:@"lastTimeString"
    options:NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld
    context:nil];
```

Se você compilar e executar o programa, verá que as notificações não estão sendo enviadas apropriadamente. O sistema não sabe que a `lastTimeString` é alterada sempre que `lastTime` sofre mudanças.

Para corrigir isso, você precisa informar ao sistema que `lastTime` afeta `lastTimeString`. Isso é feito através da implementação de um método de classe. Abra o `BNRLogger.m` e adicione este método de classe:

```
+ (NSSet *)keyPathsForValuesAffectingLastTimeString
{
    return [NSSet setWithObject:@"lastTime"];
}
```

Observe o nome desse método: ele é **keyPathsForValuesAffecting** mais o nome da chave, em letra maiúscula. Essa é a nomenclatura canônica para métodos como esse, assim como os setters de propriedade são chamados de **set** mais o nome da propriedade, em letra maiúscula.

Não há necessidade de declarar esse método no `BNRLogger.h`; ele será encontrado no tempo de execução.

37

Categorias

As categorias permitem que um programador adicione métodos a quaisquer classes existentes. Por exemplo, a Apple nos deu a classe **NSString**. A Apple não compartilha o código-fonte com essa classe, mas você pode usar uma categoria para adicionar novos métodos a ela.

Crie uma nova Foundation Command Line Tool chamada VowelCounter. Depois, crie um novo arquivo que seja uma Objective-C category. Nomeie a categoria **BNRVowelCounting** e torne-a uma categoria em **NSString**.

Agora, abra **NSString+BNRVowelCounting.h** e declare um método que você deseja adicionar à classe **NSString**:

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

@interface NSString (BNRVowelCounting)
- (int)bnr_vowelCount;
@end
```

Agora, implemente o método em **NSString+BNRVowelCount.m**:

```
#import "NSString+BNRVowelCounting.h"

@implementation NSString (BNRVowelCounting)

- (int)bnr_vowelCount
{
    NSCharacterSet *charSet =
        [NSCharacterSet characterSetWithCharactersInString:@"aeiouyAEIOUY"];

   NSUInteger count = [self length];
    int sum = 0;
    for (int i = 0; i < count; i++) {
        unichar c = [self characterAtIndex:i];
        if ([charSet characterIsMember:c]) {
            sum++;
        }
    }
    return sum;
}
@end
```

Agora, use o novo método em **main.m**:

```
#import <Foundation/Foundation.h>
#import "NSString+BNRVowelCounting.h"

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

        NSString *string = @"Hello, World!";
        NSLog(@"%@", string, [string bnr_vowelCount]);

    }
    return 0;
}
```

Compile e execute o programa. Legal, não é mesmo? As categorias demonstram ser muito úteis.

É importante observar que somente este programa contém a categoria. Se você quiser que o método esteja disponível em outro programa, será necessário adicionar os arquivos a seu projeto e compilar a categoria ao compilar esse programa.

Observe também que o método que você escreveu começa com **bnr_**. Ao implementar um método usando uma categoria, ele substitui qualquer método com o mesmo nome que já exista na classe. Assim, se no futuro, a Apple implementar um método chamado **vowelCount** em **NSString**, você não vai querer que seu método substitua o deles. Assim, é uma boa ideia adicionar um prefixo como esse aos nomes de quaisquer métodos que você adicionar nas classes da Apple usando uma categoria.

Você deve usar categorias para adicionar funcionalidade a classes existentes. Não as utilize para substituir funcionalidade em classes existentes; use subclasses em vez disso.

Desafio

Crie uma nova Foundation Command Line Tool chamada DateMonger. Adicione uma categoria **NSDate** chamada **BNRDateConvenience**.

Na categoria, adicione um método de classe à **NSDate** que tenha três inteiros (ano, mês e dia) e retorne uma nova instância de **NSDate** que seja inicializada à meia-noite do dia passado.

Teste isso no **main()**.

Dica: É recomendável consultar a classe **NSDateComponents** sobre a qual aprendeu no Chapter 14.

Part VI

C avançado

Para se tornar um programador de Objective-C competente, você também precisa ser um programador de C competente. Em nosso anseio de torná-lo familiarizado com os objetos, pulamos algumas coisas que você pode querer saber sobre o C. Estes tópicos não são ideias que você usará todos os dias, mas os encontrará ocasionalmente, portanto, queremos apresentá-los a você aqui.

38

Lógicas binárias

Na primeira parte deste livro, descrevemos a memória de um computador como uma grande área de switches (bilhões de switches) que podem ser ativados ou desativados. Cada switch representa um bit; geralmente, usamos 1 para indicar “ativado” e 0 para indicar “desativado.”

No entanto, nunca lidamos com um único bit. Em vez disso, lidamos com blocos de bits do tamanho de bytes. Se você considerar um byte como um inteiro sem sinal de 8 bits, cada bit representará outra potência de dois:

Figure 38.1 Um byte representando o número decimal 60

128	64	32	16	8	4	2	1
0	0	1	1	1	1	0	0
$32 + 16 + 8 + 4$						= 60	

Por ter 10 dedos, as pessoas gostam de trabalhar com números decimais (base 10). Os computadores, por envolverem o uso de switches que podem apenas estar ativados ou desativados, gostam de trabalhar com potência de 2. Os programadores geralmente utilizam um sistema numérico de base 16 ($16 = 2^4$) conhecido como *hexadecimal* ou simplesmente “hex”. Isso vale principalmente ao lidarmos com bits individuais de um inteiro.

Usamos as letras a, b, c, d, e e f para dígitos extras. Desse modo, a contagem em hexadecimal é esta: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, ...

Para deixar claro quando estamos escrevendo em hexadecimal, prefixamos o número com 0x. Aqui temos o mesmo número expresso em bytes usando o sistema hexadecimal:

Figure 38.2 Um byte representando o número hexadecimal 0x3c

0x80	0x40	0x20	0x10	0x8	0x4	0x2	0x1
0	0	1	1	1	1	0	0
$0x20 + 0x10 + 0x8 + 0x4$						= 0x3c	

Observe que um byte sempre pode ser descrito como um número hexadecimal de dois dígitos (como 3c). Isso faz com que o sistema hexadecimal seja uma maneira aceitável de analisar dados binários. Um programador muito experiente diria assim: “Fiz a engenharia reversa do formato do arquivo analisando os arquivos do documento em um editor hexadecimal.” Você quer visualizar um arquivo como uma lista de bytes codificados de modo hexadecimal? No Terminal, execute hexdump no arquivo:

```
$ hexdump myfile.txt 0000000 3c 3f 78 6d 6c 20 76 65 72 73 69 6f 6e 3d 22 31 0000010 2e 30 22 3f 3e 0a 3
```

A primeira coluna é o deslocamento (em hexadecimal) a partir do início do arquivo do byte listado na segunda coluna. Cada número de dois dígitos representa um byte.

Operador binário OR

Se você tiver dois bytes, poderá usar o operador binário OR para uni-los e criar um terceiro byte. Um bit no terceiro byte será 1 se, pelo menos, um dos bits correspondentes nos dois primeiros bytes for 1.

Figure 38.3 Dois bytes unidos com o operador binário OR

0	0	1	1	1	1	0	0	= 0x3c
1	0	1	0	1	0	0	1	= 0xa9
1	0	1	1	1	1	0	1	= 0xbd

Isso é feito com o operador `|`. Para testar sua habilidade na manipulação de bits, crie um novo projeto: uma C Command Line Tool (e não Foundation) chamada `bitwise`.

Edite o `main.c`:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    unsigned char a = 0x3c;
    unsigned char b = 0xa9;
    unsigned char c = a | b;

    printf("Hex: %x | %x = %x\n", a, b, c);
    printf("Decimal: %d | %d = %d\n", a, b, c);

    return 0;
}
```

Ao executar este programa, você verá os dois bytes unidos com o operador binário OR:

```
Hex: 3c | a9 = bd
Decimal: 60 | 169 = 189
```

Qual a utilidade disso? No Objective-C, geralmente usamos um inteiro para especificar uma determinada configuração. Um inteiro é sempre uma sequência de bits, e cada bit é usado para representar um aspecto da configuração que pode ser ativado ou desativado. Criamos este inteiro (também conhecido como *bit mask*, ou máscara de bits) fazendo a seleção a partir de um conjunto de constantes. Essas constantes também são inteiros; cada constante especifica um único aspecto da configuração com a ativação de apenas um de seus bits. Você pode unir as constantes que representam os aspectos particulares que deseja usando o operador binário OR. O resultado será a configuração exata que você está procurando.

Vejamos um exemplo. O iOS vem com uma classe chamada **NSDataDetector**. As instâncias de **NSDataDetector** analisam o texto e buscam padrões comuns, como datas ou URLs. Os padrões que uma instância procurará serão determinados pelo resultado do operador binário OR de um conjunto de constantes de inteiros.

`NSDataDetector.h` define estas constantes: `NSTextCheckingTypeDate`, `NSTextCheckingTypeAddress`, `NSTextCheckingTypeLink`, `NSTextCheckingTypePhoneNumber` e `NSTextCheckingTypeTransitInformation`. Ao criar uma instância de **NSDataDetector**, você informa o que procurar. Por exemplo, se quiser procurar por números de telefone e datas, você deve fazer o seguinte:

```
NSError *e;
NSDataDetector *d = [NSDataDetector dataDetectorWithTypes:
    NSTextCheckingTypePhoneNumber|NSTextCheckingTypeDate
    error:&e];
```

Observe o operador binário OR. Cada um dos números sendo unidos com o operador OR tem exatamente um bit, assim, a bit mask (máscara de bits) resultante teria dois bits. Você verá muito este padrão na programação no Cocoa e no iOS, e agora saberá o que ocorre em segundo plano.

Operador binário AND

Você também pode unir dois bytes usando o operador binário AND para criar um terceiro byte. Nesse caso, um bit no terceiro byte será 1 se ambos os bits correspondentes nos dois primeiros bytes forem 1.

Figure 38.4 Dois bytes unidos com o operador binário AND

<table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	0	0	1	1	1	1	0	0	= 0x3c
0	0	1	1	1	1	0	0		
<hr/>									
<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr> </table>	1	0	1	0	1	0	0	1	= 0xa9
1	0	1	0	1	0	0	1		
<hr/>									
<table border="1"> <tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr> </table>	0	0	1	0	1	0	0	0	= 0x28
0	0	1	0	1	0	0	0		

Isso é feito com o operador `&`. Adicione as seguintes linhas ao `main.c`:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    unsigned char a = 0x3c;
    unsigned char b = 0xa9;
    unsigned char c = a | b;

    printf("Hex: %x | %x = %x\n", a, b, c);
    printf("Decimal: %d | %d = %d\n", a, b, c);

    unsigned char d = a & b;

    printf("Hex: %x & %x = %x\n", a, b, d);
    printf("Decimal: %d & %d = %d\n", a, b, d);

    return 0;
}
```

Ao executar, você verá os dois bytes unidos com o operador binário AND:

```
Hex: 3c & a9 = 28
Decimal: 60 & 169 = 40
```

No Objective-C, usamos o operador binário AND para verificar se um determinado bit, ou *flag*, está ativado. Por exemplo, se você encontrar uma instância de **NSDataDetector**, poderá verificar se ela foi configurada para procurar por números de telefone desta maneira:

```
if ([currentDetector checkingTypes] & NSTextCheckingTypePhoneNumber) {
    NSLog(@"This one is looking for phone numbers");
}
```

O método **checkingTypes** retorna um inteiro que é o resultado do operador binário OR de todos os flags que esta instância de **NSDataDetector** ativou. Você usa o operador binário AND para este inteiro com uma constante **NSTextCheckingType** particular e verifica o resultado. Se o bit que estiver ativado em **NSTextCheckingTypePhoneNumber** não estiver na configuração do detector de dados, o resultado do uso do operador binário AND será zero. Caso contrário, você obterá um resultado diferente de zero e saberá que este detector de dados procura por números de telefone.

Observe que, quando usamos bits desse modo, não nos importamos, nesses casos, com os equivalentes numéricos desses inteiros.

Outros operadores binários

Vejamos quais são os outros operadores binários. É importante conhecê-los, embora sejam usados com menos frequência no Objective-C.

Exclusive-OR

Você pode usar o operador exclusive-or (XOR) para unir dois bytes e criar um terceiro. Um bit no terceiro byte será 1 se exatamente um dos bits correspondentes nos bytes resultantes for 1.

Figure 38.5 Dois bytes unidos com o operador binário XOR

0	0	1	1	1	1	0	0	= 0x3c
1	0	1	0	1	0	0	1	= 0xa9
1	0	0	1	0	1	0	1	= 0x95

Isso é feito com o operador `^`. Adicione ao `main.c`:

```
unsigned char e = a ^ b;  
  
printf("Hex: %x ^ %x = %x\n", a, b, e);  
printf("Decimal: %d ^ %d = %d\n", a, b, e);  
  
return 0;  
}
```

Ao executá-lo, você verá o seguinte:

```
Hex: 3c ^ a9 = 95  
Decimal: 60 ^ 169 = 149
```

Muitas vezes, este operador gera confusão para os iniciantes. Na maioria dos programas de planilha, o operador `^` representa exponenciação: 2^3 significa 2^3 . Em C, usamos a função `pow()` para a exponenciação:

```
double r = pow(2.0, 3.0); // Calculate 2 raised to the third power
```

Complemento

Se você tiver um byte, o complemento será o byte que for o oposto exato: cada 0 torna-se 1 e cada 1 torna-se 0.

Figure 38.6 O complemento

1	0	1	0	1	0	0	1	= 0xa9
0	1	0	1	0	1	1	0	= 0x56

Isso é feito com o operador `~`. Adicione algumas linhas ao `main.c`:

```
unsigned char f = ~b;  
printf("Hex: The complement of %x is %x\n", b, f);  
printf("Decimal: The complement of %d is %d\n", b, f);  
  
return 0;  
}
```

Você verá:

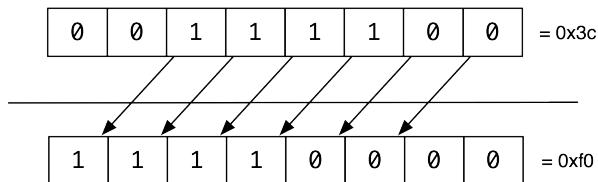
```
Hex: The complement of a9 is 56
```

Decimal: The complement of 169 is 86

Deslocamento à esquerda

Se você deslocar os bits à esquerda, cada bit será movido para o bit mais significativo. Aqueles que estiverem à esquerda do número serão desprezados, e os espaços criados à direita serão preenchidos com zeros.

Figure 38.7 Deslocamento de 2 casas à esquerda



O deslocamento à esquerda é feito com o operador <<. Adicione um deslocamento de duas casas ao main.c:

```
unsigned char g = a << 2;
printf("Hex: %x shifted left two places is %x\n", a, g);
printf("Decimal: %d shifted left two places is %d\n", a, g);

return 0;
}
```

Ao executar este código, você verá:

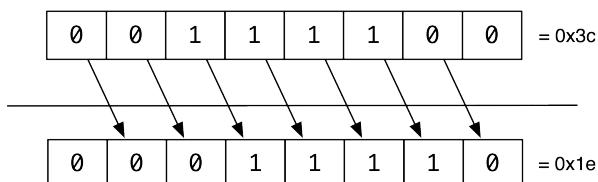
```
Hex: 3c shifted left two places is f0
Decimal: 60 shifted left two places is 240
```

Sempre que deslocar um número uma casa à esquerda, você dobrará o valor dele.

Deslocamento à direita

O operador para deslocamento à direita não será algo surpreendente.

Figure 38.8 Deslocamento de 1 casa à direita



Adicione código ao main.m:

```
unsigned char h = a >> 1;
printf("Hex: %x shifted right one place is %x\n", a, h);
printf("Decimal: %d shifted right one place is %d\n", a, h);

return 0;
}
```

Ao executá-lo:

```
Hex: 3c shifted right one place is 1e
Decimal: 60 shifted right one place is 30
```

Sempre que deslocar um número uma casa à direita, você dividirá seu valor pela metade. (Se ele for ímpar, arredonde-o para menos.)

Utilização de enum para definir máscaras de bits

Normalmente você irá querer definir uma lista de constantes, cada uma representando um inteiro com um bit ativado. Depois, será possível usar o operador binário OR para unir esses inteiros e testá-los usando o operador binário AND, conforme descrito acima.

O modo apropriado de fazer isso é definir um enum que utilize o operador de deslocamento à esquerda para definir os valores. Este é o modo como as constantes de **UIDataDetector** são definidas:

```
enum {
    UIDataDetectorTypePhoneNumber = 1 << 0,
    UIDataDetectorTypeLink      = 1 << 1,
    UIDataDetectorTypeAddress   = 1 << 2,
    UIDataDetectorTypeCalendarEvent = 1 << 3,
    UIDataDetectorTypeNone      = 0,
    UIDataDetectorTypeAll       = NSUIntegerMax
};
```

Mais bytes

Neste capítulo, você trabalhou com o `unsigned char`, que é um byte de 8 bits. Todo tipo de inteiro sem sinal funcionará da mesma maneira. Por exemplo, `NSTextCheckingTypePhoneNumber` é efetivamente declarado como `uint64_t`, um número sem sinal de 64 bits

Desafio

Escreva um programa que crie um inteiro sem sinal de 64 bits, de modo que todos os outros bits estejam ativados. (Efetivamente, existem dois números resultantes possíveis: um é par e o outro é ímpar. Crie o ímpar.) Exiba o número. Para verificar seu trabalho, a resposta é 6,148,914,691,236,517,205.

39

Strings em C

Em vista das opções, um programador em Objective-C sempre optará por trabalhar com **NSString** em vez de strings em C. No entanto, às vezes não temos opção. O motivo mais comum de terminarmos usando strings em C? Quando acessamos uma biblioteca do C a partir de nosso código em Objective-C. Por exemplo, há uma biblioteca de funções em C que permite que seu programa se comunique com um servidor de banco de dados do PostgreSQL. As funções nessa biblioteca usam strings em C, e não instâncias de **NSString**.

char

Na última seção, falamos sobre como um byte pode ser tratado como um número. Também podemos tratar um byte como um caractere. Como mencionado anteriormente, há muitas codificações diferentes de strings. A mais antiga e mais famosa é ASCII. O ASCII (American Standard Code for Information Interchange) define um caractere diferente para cada byte. Por exemplo, 0x4b é o caractere ‘K’.

Crie uma nova C Command Line Tool e nomeie como *yostring*. Neste programa, você vai listar alguns dos caracteres no padrão ASCII. Edite o *main.c*:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    char x = 0x21; // The character '!'
    while (x <= 0x7e) { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
    }
    return 0;
}
```

Compile e execute. Você deve estar se perguntando: “Um byte pode conter qualquer um dos 256 números? Você exibiu apenas 94 caracteres. O que aconteceu com o restante?”. Bem, o ASCII foi escrito para conduzir terminais抗igos de teletipo que imprimiam em papel em vez de usar uma tela, então os caracteres de 1 a 31 em ASCII são códigos de controle não imprimíveis. Por exemplo, o número 7 em ASCII faz com que soe uma campainha no terminal. O número 32 é o caractere de espaço. O número 127 é a exclusão – ele faz com que o caractere anterior desapareça. E os caracteres de 128 a 255? O ASCII usa apenas 7 bits. Não há caractere ASCII para o número 0.

Você pode usar caracteres ASCII como literais no código. Basta colocá-los dentro de aspas simples. Altere seu código para usar estes:

```
int main (int argc, const char * argv[])
{
    char x = '!'; // The character '!'
    while (x <= '~') { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
    }
    return 0;
}
```

Compile e execute.

Os caracteres não imprimíveis podem ser expressos com o uso de sequências de escape que começam com \. Você já usou \n para o caractere de nova linha. Veja alguns outros comuns:

Table 39.1 Sequências escapes comuns

\n	nova linha
\t	guia
\'	aspas simples
\"	aspas duplas
\0	byte nulo (0x00)
\\"	barra invertida

char *

Uma string em C é simplesmente um conjunto de caracteres um ao lado do outro na memória. A string termina quando o caractere 0x00 é encontrado.

Figure 39.1 A palavra “Love” como uma string em C

0x4c	0x6f	0x76	0x65	0x00
'L'	'o'	'v'	'e'	'\0'

As funções que utilizam strings em C esperam o endereço do primeiro caractere da string. `strlen()`, por exemplo, contará o número de caracteres em uma string. Tente compilar uma string e usar `strlen()` para contar as letras:

```

#include <stdio.h> // For printf
#include <stdlib.h> // For malloc/free
#include <string.h> // For strlen

int main (int argc, const char * argv[])
{
    char x = '!'; // The character '!'
    while (x <= '~') { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
    }

    // Get a pointer to 5 bytes of memory on the heap
    char *start = malloc(5);

    // Put 'L' in the first byte
    *start = 'L';

    // Put 'o' in the second byte
    *(start + 1) = 'o';

    // Put 'v' in the third byte
    *(start + 2) = 'v';

    // Put 'e' in the fourth byte
    *(start + 3) = 'e';

    // Put zero in the fifth byte
    *(start + 4) = '\0';

    // Print out the string and its length
    printf("%s has %zu characters\n", start, strlen(start));

    // Print out the third letter
    printf('The third letter is %c\n', *(start + 2));

    // Free the memory so that it can be reused
    free(start);
    start = NULL;

    return 0;
}

```

Compile e execute.

Observe os locais em que você adicionou um ponteiro e um número juntos. `start` é declarada como um `char *`. Um `char` é um byte. Então, `start + 1` é um ponteiro com um byte a mais na memória que `start`. `start + 2` é dois bytes a mais na memória que `start`.

Figure 39.2 O endereço de cada caractere

start	start+1	start+2	start+3	start+4
L	o	v	e	\0

Esta adição a um ponteiro e esta "desreferenciação" do resultado são tão comuns que há um atalho para elas: `start[2]` equivale a `*(start + 2)`. Altere seu código para usar isto:

```
char *start = malloc(5);
start[0] = 'L';
start[1] = 'o';
start[2] = 'v';
start[3] = 'e';
start[4] = '\0';

printf("%s has %zu characters\n", start, strlen(start));
printf("The third letter is %c\n", start[2]);

free(start);
start = NULL;

return 0;
}
```

Compile e execute.

É importante mencionar que isso funciona com qualquer tipo de dados. Aqui, por exemplo, posso listar meus 3 números de ponto flutuante favoritos e exibi-los:

```
int main (int argc, const char * argv[])
{
    // Claim a chunk of memory big enough to hold three floats
    float *favorites = malloc(3 * sizeof(float));

    // Push values into the locations in that buffer
    favorites[0] = 3.14158;
    favorites[1] = 2.71828;
    favorites[2] = 1.41421;

    // Print out each number on the list
    for (int i = 0; i < 3; i++) {
        printf("%.4f is favorite %d\n", favorites[i], i);
    }

    // Free the memory so that it can be reused
    free(favorites);
    favorites = NULL;

    return 0;
}
```

A única diferença interessante aqui é que `favorites` é escrito como um `float *`. Um `float` é 4 bytes. Portanto, `favorites + 1` é 4 bytes a mais na memória que `favorites`.

Figure 39.3 Um array de três floats

favorites	favorites + 1	favorites + 2
3.14158	2.71828	1.41421
4 bytes	4 bytes	4 bytes

Strings literais

Se você está lidando muito com strings em C, alocar a memória e incluir os caracteres, um a um, seria muito trabalhoso. Em vez disso, você pode criar um ponteiro para uma string de caracteres (terminada com o caractere zero) colocando a string entre aspas. Altere seu código para usar uma string literal:

```

int main (int argc, const char * argv[])
{
    char x = '!'; // The character '!'
    while (x <= '~') { // The character '~'
        printf("%x is %c\n", x, x);
        x++;
    }

    char *start = "Love";
    printf("%s has %zu characters\n", start, strlen(start));
    printf("The third letter is %c\n", start[2]);

    return 0;
}

```

Compile e execute.

Observe que não é preciso alocar e liberar memória para uma string literal. Ela é uma constante e aparece na memória apenas uma vez, então, o compilador cuida de seu uso de memória. Como uma consequência de ela ser uma constante, problemas poderão ocorrer se você tentar alterar os caracteres na string. Adicione uma linha que causará erro em seu programa:

```

char *start = "Love";
start[2] = 'z';
printf("%s has %zu characters\n", start, strlen(start));

```

Ao compilar e executar, você verá um sinal EXC_BAD_ACCESS. Você tentou gravar na memória para a qual não tem permissão para fazer isso.

Para permitir que o compilador avise sobre a gravação em partes constantes da memória, você pode usar o modificador `const` para especificar que um ponteiro está se referindo a dados que não devem ser alterados. Faça o teste:

```

const char *start = "Love";
start[2] = 'z';
printf("%s has %zu characters\n", start, strlen(start));

```

Agora, ao compilar, você verá um erro a partir do compilador.

Exclua a linha problemática (`start[2] = 'z'`) antes de continuar.

Você pode usar as sequências de escape, mencionadas acima, em suas strings literais. Use algumas:

```

const char *start = "A backslash after two newlines and a tab:\n\n\t\"";
printf("%s has %zu characters\n", start, strlen(start));
printf("The third letter is '\"%c\"\n", start[2]);

return 0;
}

```

Compile e execute.

Conversão de e para NSString

Se estiver usando strings em C em um programa em Objective-C, será necessário saber como criar uma **NSString** a partir de uma string em C. A classe **NSString** tem um método para isso:

```

char *greeting = "Hello!";
NSString *x = [NSString stringWithCString:greeting encoding:NSUTF8StringEncoding];

```

Você também pode obter uma string em C a partir de uma **NSString**. Isso é um pouco mais difícil, pois a **NSString** pode lidar com alguns caracteres que determinadas codificações não podem. É recomendável verificar se a conversão pode ocorrer:

```

NSString *greeting = "Hello!";
const char *x = NULL;
if ([greeting canBeConvertedToEncoding:NSUTF8StringEncoding]) {
    x = [greeting cStringUsingEncoding:NSUTF8StringEncoding];
}

```

Você não tem a string em C resultante; o sistema irá liberá-la para você posteriormente. Você tem a garantia de que ela continuará existindo, pelo menos, enquanto existir o pool atual de liberação automática; no entanto, se precisar que a string em C exista por um longo tempo, será necessário copiá-la em um buffer que você criou com `malloc()`.

Desafio

Escreva uma função chamada `spaceCount()` que conte os caracteres de espaço (ASCII 0x20) em uma string em C. Teste fazer assim:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    const char *sentence = "He was not in the cab at the time.";
    printf ("\n%s\n has %d spaces\n", sentence, spaceCount(sentence));

    return 0;
}
```

Lembre-se: ao executar em '\0', você terá alcançado o final da string!

40

Arrays em C

No último capítulo, você trabalhou com strings em C. Uma string em C demonstrou ser uma lista de caracteres dispostos um ao lado do outro na memória. Arrays em C são listas de outros tipos de dados dispostos um ao lado do outro na memória. Assim como ocorre com as strings, você lida com a lista retendo o endereço do primeiro.

Imagine que você quisesse escrever um programa que calculasse a média de 3 notas. Crie um novo projeto de C Command Line Tool e nomeie-o como gradeInTheShade.

Edito o main.c:

```
#include <stdio.h>
#include <stdlib.h> // malloc(), free()

float averageFloats(float *data, int dataCount)
{
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}

int main (int argc, const char * argv[])
{
    // Create an array of floats
    float *gradeBook = malloc(3 * sizeof(float));
    gradeBook[0] = 60.2;
    gradeBook[1] = 94.5;
    gradeBook[2] = 81.1;

    // Calculate the average
    float average = averageFloats(gradeBook, 3);

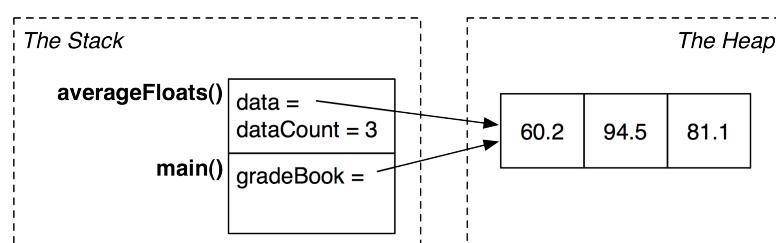
    // Free the array
    free(gradeBook);
    gradeBook = NULL;

    printf("Average = %.2f\n", average);

    return 0;
}
```

Compile e execute.

Figure 40.1 Ponteiros na pilha para um buffer de floats



`malloc()` aloca um buffer no heap, por isso, você precisa liberá-lo ao finalizar. Não seria ótimo se você pudesse declarar esse buffer como parte do frame (na pilha) de modo que ele fosse desalocado automaticamente quando a função finalizasse a execução? Você pode. Altere `main.c`:

```
import <stdio.h>

float averageFloats(float *data, int dataCount)
{
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}

int main (int argc, const char * argv[])
{
    // Declares the array as part of the frame
    float gradeBook[3];

    gradeBook[0] = 60.2;
    gradeBook[1] = 94.5;
    gradeBook[2] = 81.1;

    // Calculate the average
    float average = averageFloats(gradeBook, 3);

    // No need to free the array!
    // Cleanup happens automatically when the function returns

    printf("Average = %.2f\n", average);

    return 0;
}
```

Compile e execute.

A literal de string facilita o processo de dispor um array com caracteres. Também há muitas literais de array. Use uma para inicializar `gradeBook`:

```
int main (int argc, const char *argv[])
{
    float gradeBook[] = {60.2, 94.5, 81.1};

    float average = averageFloats(gradeBook, 3);

    printf("Average = %.2f", average);

    return 0;
}
```

Compile e execute o programa.

Observe que você não precisou especificar o comprimento de `gradeBook` como 3; o compilador calcula isso a partir da literal de array. Você pode usar esse tipo em muitos lugares em que deve usar *. Por exemplo, altere a declaração de `averageFloats()` para fazer isto:

```
float averageFloats(float data[], int dataCount)
{
    float sum = 0.0;
    for (int i = 0; i < dataCount; i++) {
        sum = sum + data[i];
    }
    return sum / dataCount;
}
```

Compile e execute o programa.

Desafio

Antes de ler este livro, você deve ter usado provavelmente notação hexadecimal: quase toda chave de licença de software ou código de cupom fornecido é escrito em hexadecimal. Você provavelmente já deve ter visto isso “Digite esse código no campo de código do cupom: 4af812e660ba8c123ee.” É uma maneira comum de fazer um usuário digitar um conjunto aparentemente aleatório de bytes.

No entanto, a notação hexadecimal está repleta de perigo quando esses valores precisam funcionar com reconhecimento de fala. “B”, “C”, “D” e “E” parecem muito com humanos e máquinas. A Big Nerd Ranch tinha um cliente com esse problema, que me solicitou um sistema de chave de licença que o permitisse recitar facilmente as chaves de licença pelo telefone.

Nossa solução era um sistema baseado em um conjunto de palavras internacionalmente reconhecidas. Quais são essas palavras? Em sua maioria, marcas internacionais, como “Honda,” “Google,” e “Nike.” A ideia de usar nomes de marcas veio de uma passagem do livro *Ruído Branco* de Don DeLillo. O narrador está ouvindo uma criança recitar marcas de carro enquanto dorme. Ele diz “Ela estava apenas repetindo o que ouviu na televisão. Toyota Corolla, Toyota Celica, Toyota Cressida. Nomes supranacionais, gerados por computador, mais ou menos universalmente pronunciáveis. Parte de cada ruído do cérebro de uma criança - regiões muito profundas para serem investigadas. Seja qual for a fonte, a elocução me surpreendeu com o impacto de um momento de transcendência esplêndida.”

Essa solução permitiu ao nosso cliente criar e entender strings feitas de números e nomes de marcas. Essas strings poderiam ser transferidas em um byte de dados. Seu desafio é criar um sistema similar. Aviso: Este é o desafio final deste livro, e é bastante difícil.

Os 3 bits mais à esquerda de cada byte serão codificados como um dígito entre 2 e 9, inclusive. Não se deve usar 0 e 1 porque na escrita esses números são facilmente confundidos com as letras O e I. Os cinco bits restantes serão representados por um nome de marca. 2^5 é 32. Portanto, você precisará de 32 nomes de marcas internacionais. Veja uma lista que você pode usar:

0. Camry
1. Nikon
2. Apple
3. Ford
4. Audi
5. Google
6. Nike
7. Amazon
8. Honda
9. Mazda
10. Buick
11. Fiat
12. Jeep
13. Lexus
14. Volvo
15. Fuji
16. Sony
17. Delta
18. Focus
19. Puma
20. Samsung
21. Tivo
22. Halo
23. Sting

24. Shrek
25. Avatar
26. Shell
27. Visa
28. Vogue
29. Twitter
30. Lego
31. Pepsi

Assim, por exemplo, quatro bytes de dados aleatórios podem ser codificados em hexadecimal como 53ec306f. Em nosso sistema, os mesmos dados se tornariam a string “4 Puma 9 Jeep 3 Sony 5 Fuji”. Ao fazer a análise sintática da string, nós ignoraríamos o espaço em branco e o uso de maiúsculas.

O desafio, então, é criar uma nova ferramenta de linha de comando que inclua dois métodos:

- Obter um buffer dos bytes e retornar uma string que represente esses bytes
- Obter uma string e retornar o buffer de bytes que ela representa

Você adicionará esses métodos à **NSData** como uma categoria. Eis a declaração da categoria:

```
@interface NSData (Speakable)
- (NSString *)encodeAsSpeakableString;
+ (NSData *)dataWithSpeakableString:(NSString *)s
                           error:(NSError **)e;
@end
```

O primeiro método é considerado mais fácil de escrever do que o segundo.

Eis um **main.m** que testará esses métodos:

```

#import <Foundation/Foundation.h>
#import "NSData+Speakable.h"

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

        // Generate 8 bytes of random data
        srand((unsigned int)time(NULL));
        int64_t randomBytes = (random() << 32) | random();

        // Pack it in an NSData
        NSData *inData = [NSData dataWithBytes:&randomBytes
                                         length:sizeof(int64_t)];
        NSLog(@"In Data = %@", inData);

        // Convert to a speakable string
        NSString *speakable = [inData encodeAsSpeakableString];
        NSLog(@"Got string \"%@\"", speakable);

        // Converting it back to an NSData
        NSError *err;
        NSData *outData = [NSData dataWithSpeakableString:speakable
                                                 error:&err];
        if (!outData) {
            NSLog(@"Unexpected error: %@", [err localizedDescription]);
            return -1;
        }
        NSLog(@"Out data: %@", outData);

        // outData better be the same as inData
        if (![outData isEqual:inData]) {
            NSLog(@"Data coming out not the same as what went in.");
            return -1;
        }

        // Test a misspelling ("Teevo" not "Tivo")
        speakable = @"2 Jeep 3 Halo 7 Teevo 2 Pepsi 2 Volvo";
        outData = [NSData dataWithSpeakableString:speakable
                                         error:&err];
        if (!outData) {
            NSLog(@"Expected error: %@", [err localizedDescription]);
        } else {
            NSLog(@"Missed bad string");
            return -1;
        }
    }
    return 0;
}

```

Esse programa deve produzir resultados como este:

```

In Data = <53ec306f 955c6668>
Got string "4 Puma 9 Jeep 3 Sony 5 Fuji 6 Tivo 4 Vogue 5 Nike 5 Honda"
Out data: <53ec306f 955c6668>
Expected error: Unable to parse
Program ended with exit code: 0

```

Em algum ponto, você vai se encontrar perguntando, “Onde está o próximo dígito?” Você pode usar **NSCharacterSet** para encontrá-lo.

```
NSString *string =  
  
// Get the digit character set  
NSCharacterSet *digits = [NSCharacterSet decimalDigitCharacterSet];  
NSRange searchRange;  
searchRange.location = 0;  
searchRange.length = [string length];  
  
// Find the location of the first digit in the string  
NSRange digitRange = [str rangeOfCharacterFromSet:digits  
                           options:NSLiteralSearch  
                           range:searchRange];  
  
// Are there no digits?  
if (digitRange.length == 0) {  
    NSLog(@"Searched whole string and found no digits");  
} else {  
    NSLog(@"Character %d is a digit", digitRange.location);  
}
```

Você precisa consultar a documentação de **NSString** e de **NSMutableString** para descobrir algumas das formas que você pode manipular substrings usando NSRange.

Em **dataWithSpeakableString:error:**, você precisará lidar com strings formatadas de maneira ruim ao criar uma **NSError** e retornar nil. Ficará mais ou menos assim:

```
// Did the parse fail?  
if (!success) {  
  
    // Did the caller give me a place to put the error?  
    if (e) {  
        NSDictionary *userInfo = @{@"NSLocalizedDescriptionKey" : @"Unable to parse"};  
        *e = [NSError errorWithDomain:@"SpeakableBytes"  
                           code:1  
                           userInfo:userInfo];  
        return nil;  
    }  
}
```

Você usará várias operações binárias.

Boa sorte!

41

Execução a partir da linha de comando

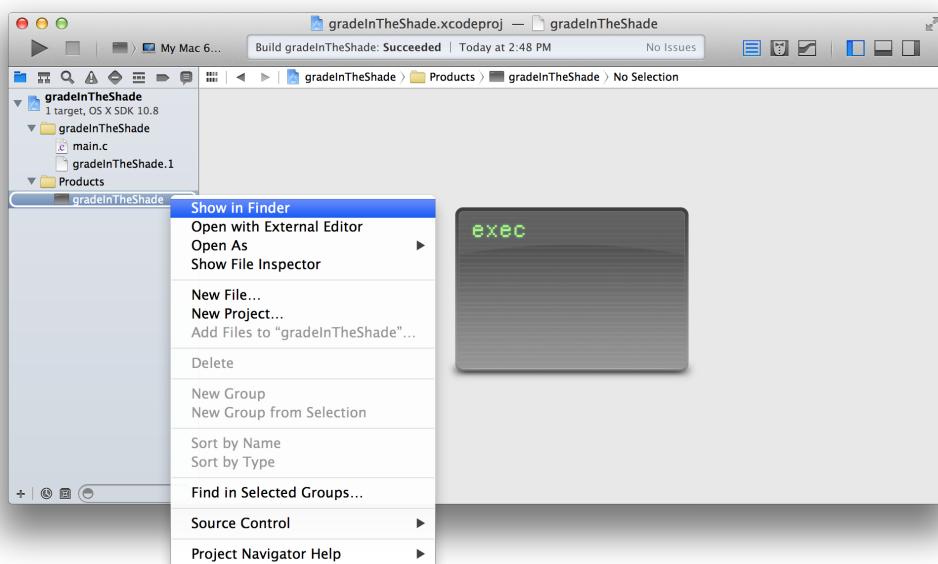
Neste livro, você compilou e executou muitas ferramentas de linha de comando no Xcode. A execução no Xcode funciona muito bem para testar programas e aprender programação. No entanto, se você criar uma ferramenta de linha de comando para usar na vida real, você pode querer executá-la a partir da linha de comando.

Em um Mac, você normalmente usa o Terminal para executar programas a partir da linha de comando. O aplicativo Terminal é apenas uma interface bonita para o que é chamado de *shell*. Existem alguns shells diferentes com nomes que chamam a atenção, como csh, sh, zsh e ksh, mas quase todos os usuários de Mac usam o bash.

Para executar um programa a partir da linha de comando, você insere o caminho do arquivo executável do programa no Terminal e pressiona Return.

No Xcode, retorne para seu projeto `gradeInTheShade` a partir do Chapter 40. No navegador do projeto, exiba o conteúdo da pasta Products e encontre um arquivo chamado `gradeInTheShade`. Esse é seu arquivo executável. Clique com o botão direito nesse arquivo e escolha Show In Finder.

Figure 41.1 Exibição de executável no Finder



Copie o arquivo `gradeInTheShade` a partir do Finder para o seu desktop.

Em seguida, abra o Terminal e digite o seguinte comando:

```
$ ~/Desktop/gradeInTheShade
```

Pressione Return; o programa será executado e o resultado será exibido na janela do Terminal. Sua ferramenta de linha de comando corresponde agora ao seu nome.

Argumentos de linha de comando

Uma ferramenta de linha de comando pode ter um ou mais *argumentos de linha de comando*. Um argumento de linha de comando fornece informações para a ferramenta sobre o que você deseja que ela faça.

Nenhuma das ferramentas que você criou até agora já teve argumentos de linha de comando, mas ferramentas de linha de comando úteis normalmente têm. Neste capítulo, você vai criar outra ferramenta de linha de comando chamada **Affirmation**. Não será particularmente útil, mas terá argumentos de linha de comando.

Veja a aparência do **Affirmation** em execução quando você tiver terminado:

```
$ Affirmation awesome 4
You are awesome!
You are awesome!
You are awesome!
You are awesome!
```

Essa ferramenta tem dois argumentos de linha de comando: um adjetivo inspirador e o número de vezes para exibir a afirmação. Os argumentos de linha de comando são digitados logo após o caminho do arquivo, separados por um espaço em branco. Esses argumentos serão lidos no programa **Affirmation** como strings, e, em seguida, o programa vai usar essas informações para finalizar seu trabalho.

No Xcode, crie um novo projeto C Command Line Tool chamado **Affirmation**. Abra o **main.c** e verifique esse código familiar:

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

A função **main** tem dois argumentos. O segundo argumento, **argv**, é um array de strings em C. É nele onde os argumentos de linha de comando são armazenados. Cada argumento de linha de comando se torna uma string em C e é compactado em **argv** antes de **main()** ser chamado. O primeiro argumento, **argc**, é o número de strings em **argv**.

No **main.c**, edite **main()** para exibir o conteúdo de **argv**:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    for (int i = 0; i < argc; i++) {
        printf("arg %d = %s\n", i, argv[i]);
    }

    return 0;
}
```

Use o Command-B para compilar (mas não para executar) esse programa. A partir do navegador de projetos, selecione o arquivo executável e exiba-o no Finder. Copie o arquivo para sua área de trabalho (desktop). Em seguida, no Terminal, insira o seguinte comando:

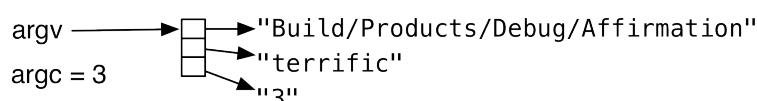
```
$ ~/Desktop/Affirmation terrific 3
```

Aqui está o seu resultado:

```
arg 0 = /Users/mward/Desktop/Affirmation
arg 1 = terrific
arg 2 = 3
```

Surpresa! O primeiro item em **argv** não é o primeiro argumento de linha de comando; ele é o caminho para o arquivo executável. O item está sempre em **argv** mesmo em programas que não aceitam argumentos de linha de comando.

Figure 41.2 **argv** e **argc** no **Affirmation**



No Xcode, modifique `main()` para exibir as afirmações. Use `atoi()` para converter a string do item `argv` para um `int` que você possa usar.

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

int main (int argc, const char * argv[])
{
    int count = atoi(argv[2]);

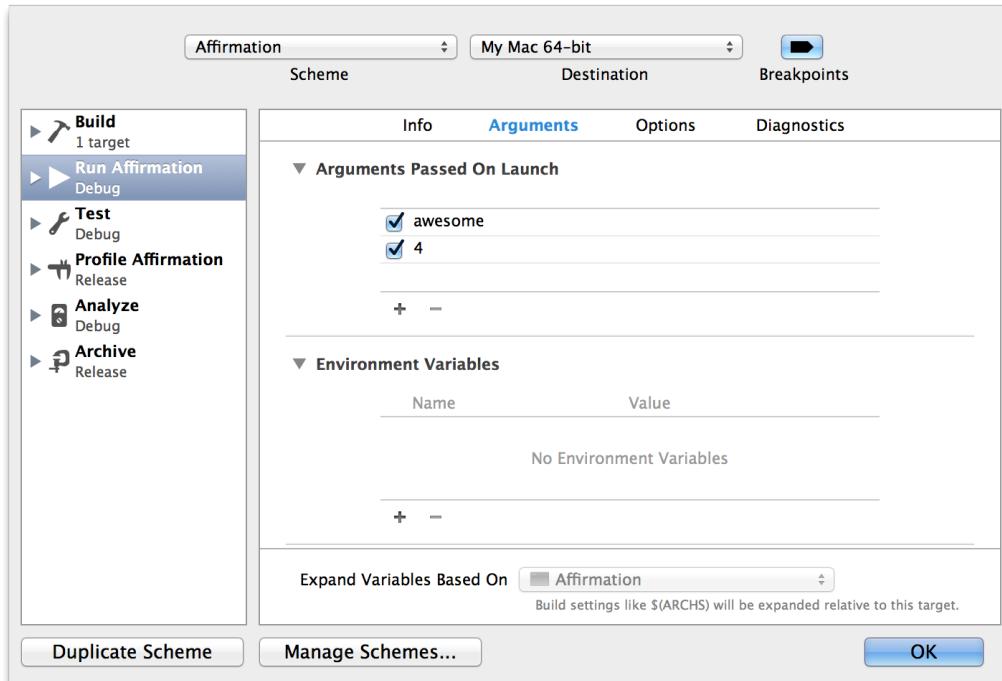
    for (int j = 0; j < count; j++) {
        printf("You are %s!\n", argv[1]);
    }

    return 0;
}
```

Para executar o código modificado, você pode seguir o mesmo processo que realizou antes: compile, exiba o Finder, copie para o Desktop, execute a partir do Terminal. Felizmente, no entanto, existe uma maneira de fornecer os argumentos de linha de comando para o Xcode de forma que você possa criar e executar sem ter que deixar o conforto da familiaridade.

A partir da barra de menus do Xcode, selecione Product → Scheme → Edit Scheme.... Na página que aparece, selecione Run Affirmation no lado esquerdo. Depois, selecione Arguments a partir das opções na parte superior da página. Encontre a lista chamada Arguments Passed On Launch e use o botão + para adicionar dois argumentos específicos:

Figure 41.3 Adição de argumentos



Clique em OK para fechar a página. Em seguida, compile e execute o programa. O console exibirá o resultado com base nos dois argumentos de linha de comando que você incluiu no esquema.

Observe que não há nada atualmente no código que verifica o número de argumentos de linha de comando inseridos. Normalmente, é importante verificar para assegurar que o programa terá as informações de que precisa para executar.

No Xcode, edite o `main.c` para verificar o número correto dos argumentos de linha de comando. Não se esqueça de `argv[0]` (o caminho do arquivo executável) ao testar o valor de `argc`.

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: Affirmation <adjective> <number>\n");
        return 1;
    }

    int count = atoi(argv[2]);

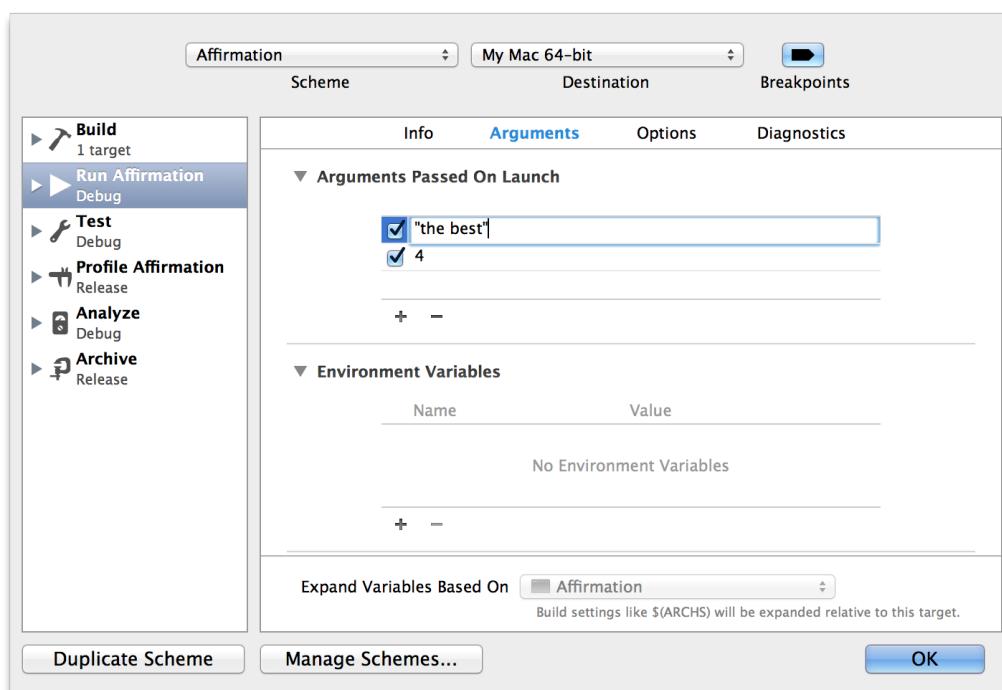
    for (int j = 0; j < count; j++) {
        printf("You are %s!\n", argv[1]);
    }

    return 0;
}
```

Edite o esquema novamente para testar se há erro na caixa de tipos (Product → Scheme → Edit Scheme...). Remova um dos argumentos e, em seguida, compile e execute novamente.

Os argumentos de linha de comando não precisam de palavras individuais. Você pode usar aspas para passar um argumento de várias palavras. Por exemplo, você poderia editar o esquema novamente e fazer do primeiro argumento "o melhor":

Figure 41.4 Adição de um argumento de várias palavras



Execução mais conveniente a partir da linha de comando

E se você quiser executar o Affirmation a partir da linha de comando regularmente? A sua melhor opção é mover o executável para um dos diretórios padrão para executáveis. Isso permitirá que você execute a partir da linha de comando usando apenas o nome do executável:

```
$ Affirmation "the best" 4
```

Os diretórios padrão para os executáveis são determinados pela variável de ambiente PATH.

No Terminal, encontre sua variável de ambiente PATH:

```
$ echo $PATH
```

```
/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
```

O diretório `/usr/local/bin` é uma boa casa para o `Affirmation`; por convenção, ele é o diretório para ferramentas instaladas pelo usuário no Unix e sistemas baseados em Unix, como o OS X.

Mesmo que o diretório `/usr/local/bin` esteja em sua variável de ambiente PATH, ele pode não existir em seu Mac. Se esse for o caso, então, você precisará criar o diretório `/usr/local/bin`. Digite o seguinte comando para criar o diretório, caso ele ainda não exista:

```
$ mkdir -p /usr/local/bin
```

Em seguida, digite o seguinte comando para abrir o diretório `/usr/local/bin` no Finder:

```
$ open /usr/local/bin
```

No Finder, copie `Affirmation` para `/usr/local/bin`.

Agora que o `Affirmation` está em `/usr/local/bin`, você não precisa mais fornecer o caminho completo para o executável no Terminal; você pode usar apenas seu nome:

```
$ Affirmation "good enough" 3
```

O sistema irá varrer os diretórios na variável de ambiente PATH, encontrar e executar o `Affirmation`.

42

Instruções switch

É comum verificar uma variável quanto a um conjunto de valores. Usando instruções `if-else`, isso seria semelhante ao seguinte:

```
int yeastType = ...;

if (yeastType == 1) {
    makeBread();
} else if (yeastType == 2) {
    makeBeer();
} else if (yeastType == 3) {
    makeWine();
} else {
    makeFuel();
}
```

Para facilitar esse processo, a linguagem C conta com a instrução `switch`. O código acima poderia ser alterado da seguinte maneira:

```
int yeastType = ...;

switch (yeastType) {
    case 1:
        makeBread();
        break;
    case 2:
        makeBeer();
        break;
    case 3:
        makeWine();
        break;
    default:
        makeFuel();
        break;
}
```

Observe as instruções `break`. Sem o `break`, após a execução da cláusula `case` apropriada, o sistema executaria todas as cláusulas `case` subsequentes. Por exemplo, se você tivesse isto:

```
int yeastType = 2;

switch (yeastType) {
    case 1:
        makeBread();
    case 2:
        makeBeer();
    case 3:
        makeWine();
    default:
        makeFuel();
}
```

o programa executaria `makeBeer()`, `makeWine()` e `makeFuel()`. Uma instrução `switch` funciona dessa forma para que você possa ter mais de um valor desencadeando o mesmo código:

```
int yeastType = ...;
```

```
switch (yeastType) {  
    case 1:  
    case 4:  
        makeBread();  
        break;  
    case 2:  
    case 5:  
        makeBeer();  
        break;  
    case 3:  
        makeWine();  
        break;  
    default:  
        makeFuel();  
        break;  
}
```

Como você pode imaginar, esquecer de inserir `break` no final de uma cláusula `case` é um erro comum dos programadores, que somente é detectado quando seu programa começa a agir de modo estranho.

Em C, as instruções `switch` destinam-se a uma situação muito específica: o valor de cada `case` deve ser uma constante de número inteiro. Assim, você não vê muitas instruções `switch` na maioria dos programas em Objective-C. Foi por isso que encaixamos isso aqui logo antes do final do livro.

Appendix

O tempo de execução do Objective-C

“Qualquer tecnologia suficientemente avançada é indistinguível da mágica.” — Arthur C. Clarke

“Mágica é idiota.” — Qualquer engenheiro, em qualquer época

Pessoas que se tornam programadoras costumam ser do tipo de gente que não se satisfaz com mágica e com a afirmação: “Simplesmente funciona.”. Queremos saber *como* e *por que* funciona.

Este capítulo revelará alguns dos mecanismos subjacentes que fazem programas em Objective-C “simplesmente funcionarem”. Esses mecanismos fazem parte do Tempo de execução do Objective-C.

O termo “tempo de execução” tem diversos significados. Até agora, nós o usamos para descrever o período de tempo durante o qual seu aplicativo está sendo executado no computador de um usuário. O tempo de execução contrasta com o “tempo de compilação”, que é o período antes de executar no qual você compila seu programa usando o Xcode.

Desenvolvedores de Objective-C também chamam de “Runtime” (em geral, com “R” maiúsculo). Essa é a parte do OS X e do iOS que executa o código Objective-C. O Objective-C Runtime (tempo de execução) é responsável por rastrear dinamicamente quais classes existem, que métodos elas têm definidos e ver que mensagens são passadas adequadamente entre objetos.

Introspecção

Uma característica do tempo de execução é a *introspecção*: a capacidade de um objeto de responder a perguntas sobre si mesmo enquanto o programa está sendo executado. Por exemplo, há um método **NSObject** chamado **respondsToSelector:**.

- (BOOL) **respondsToSelector:** (SEL) aSelector;

Seu único argumento é um selector (o nome de um método). O valor de retorno será YES se o objeto implementar o método nomeado e NO se não implementar. Usar **respondsToSelector:** é um exemplo de introspecção.

Pesquisa dinâmica e execução de método

Um aplicativo Objective-C em execução consiste, predominantemente, em objetos enviando mensagens uns aos outros. Quando um objeto envia uma mensagem, ele inicia uma pesquisa pelo método a ser executado. A pesquisa começa normalmente com a classe indicada pelo ponteiro `isa` do destinatário e, em seguida, prossegue subindo pela hierarquia de interface até encontrar um método com esse nome.

A pesquisa dinâmica e a execução do método encontrado compõem a base de todo envio de mensagem do Objective-C e são outra característica do tempo de execução.

Realizar essa busca e executar o método é o trabalho da função em C **objc_msgSend()**. Os argumentos dessa função são o destinatário da mensagem, o seletor do método a ser executado e quaisquer argumentos para o método.

Por exemplo, considere este curto programa que registra a versão em maiúsculas de uma string:

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSString *nameString = @"Mikey Ward";
        NSString *capsName = [nameString uppercaseString];
        NSLog(@"%@", nameString, capsName);
    }
    return 0;
}
```

Quando o compilador vê sua mensagem **uppercaseString**, ele substitui a mensagem por uma chamada para **objc_msgSend()**:

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

int main (int argc, const char * argv[])
{
    @autoreleasepool {
        NSString *nameString = @"Mikey Ward";
        NSString *capsName = objc_msgSend(nameString, @selector(uppercaseString));
        NSLog(@"%@", nameString, capsName);
    }
    return 0;
}
```

A função **objc_msgSend()** é uma de uma família de funções que fica no cerne de cada mensagem enviada em um programa em Objective-C. Essas funções são declaradas no **objc/message.h**. Para mais informações sobre essa e outras funções do tempo de execução, vá até a documentação do desenvolvedor e navegue para Objective-C Runtime Reference (Referência de Tempo de Execução do Objective-C).

Gerenciamento de classes e herança de hierarquias

O tempo de execução é responsável por rastrear quais classes você está usando, além daquelas que estão sendo usadas por bibliotecas e frameworks incluídos em seu aplicativo. Há diversas funções que existem para se manipular as classes carregadas pelo Runtime (tempo de execução).

Crie uma nova Foundation Command Line Tool chamada ClassAct.

No **main.m**, atualize a função **main** com o código a seguir. Não deixe de importar **objc/runtime.h** no topo.

```

#import <Foundation/Foundation.h>
#import <objc/runtime.h>

int main(int argc, const char * argv[])
{
    @autoreleasepool {
        // Declare a variable to hold the number of registered classes
        unsigned int classCount = 0;

        // Get a pointer to a list of all registered classes
        // currently loaded by your application.
        // The number of registered classes is returned by reference
        Class *classList = objc_copyClassList(&classCount);

        // For each class in the list...
        for (int i = 0; i < classCount; i++) {

            // Treat the classList as a C array to get a Class from it
            Class currentClass = classList[i];

            // Get the class's name as a string
            NSString *className = NSStringFromClass(currentClass);

            // Log the class's name
            NSLog(@"%@", className);
        }

        // We're done with the class list buffer, so free it
        free(classList);
    }
    return 0;
}

```

A função `objc_copyClassList` retorna um array em C de ponteiros para objetos `Class`. Lembre que, de acordo com o Chapter 12, a memória obtida ao se chamar `malloc()` deve ser liberada quando você concluir com ela. Por convenção, uma memória obtida ao se chamar uma função com “copy” ou “create” em seu nome, tal como `objc_copyClassList` deve ser tratada da mesma maneira. Isso se chama *criar regra*. De forma semelhante, uma memória obtida ao se chamar qualquer outra função, tal como as com “get” no nome, não é de sua propriedade e você não precisa liberá-la. Isso se chama *obter regra*. Observe que essas regras são semelhantes àquelas usadas para gerenciamento manual da memória (discutido no final do Chapter 23).

Compile e execute o seu programa.

Navegue por uma lista de classes que foram registradas pelo seu programa. É muito impressionante o fato de muitas classes diferentes terem sido escritas e fornecerem a base dos programas que escrevemos. Você deverá ver muitos nomes de classe que começam com sublinhados. Essas são classes internas apenas, que existem nas profundezas dos frameworks da Apple, fazendo nossos programas funcionarem.

Em seguida, você vai adicionar funções a seu programa para mostrar as hierarquias de classe de cada uma das classes listadas e também uma lista de todos os métodos implementados por cada classe.

No `main.m`, adicione uma função auxiliar para criar uma `NSArray` de `Classes` que representam a hierarquia de herança de uma `Class` passada:

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

NSArray *BNRHierarchyForClass(Class cls) {

    // Declare an array to hold the list of
    // this class and all its superclasses, building a hierarchy
    NSMutableArray *classHierarchy = [NSMutableArray array];

    // Keep climbing the class hierarchy until we get to a class with no superclass
    for (Class c = cls; c != Nil; c = class_getSuperclass(c)) {
        NSString *className = NSStringFromClass(c);
        [classHierarchy insertObject:className atIndex:0];
    }

    return classHierarchy;
}

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

Essa função pegará um objeto de **Class** e obterá sua superclasse. Em seguida, ela pegará a superclasse *dessa* classe e subirá pela hierarquia até alcançar uma classe sem nenhuma superclasse, que costuma ser **NSObject**.

Agora, construa uma função para obter uma lista de todos os métodos que estão implementados em uma determinada classe.

```
...
return classHierarchy;
}

NSArray *BNRMethodsForClass(Class cls) {

    unsigned int methodCount = 0;

    Method *methodList = class_copyMethodList(cls, &methodCount);

    NSMutableArray *methodArray = [NSMutableArray array];

    for (int m = 0; m < methodCount; m++) {
        // Get the current Method
        Method currentMethod = methodList[m];
        // Get the selector for the current method
        SEL methodSelector = method_getName(currentMethod);
        // Add its string representation to the array
        [methodArray addObject:NSStringFromSelector(methodSelector)];
    }

    return methodArray;
}

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

Esse código é parecido com o código que você escreveu para obter a lista de classes. Há um novo tipo aqui que você ainda não viu: Método. Neste contexto, Método é o nome de um tipo de struct cujos membros incluem o seletor de um método (variável do tipo SEL), além de um *ponteiro de função* – um ponteiro para o trecho de código a ser executado de fato no segmento de memória de dados do programa. Esse ponteiro de função é uma variável do tipo IMP.

Agora, edite **main()** para que use suas funções auxiliares:

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

```

// Create an array of dictionaries, where each dictionary
// will end up holding the class name, hierarchy, and method list
// for a given class
NSMutableArray *runtimeClassesInfo = [NSMutableArray array];

// Declare a variable to hold the number of registered classes
unsigned int classCount = 0;

// Get a pointer to a list of all registered classes
// currently loaded by your application
// The number of registered classes is returned by reference
Class *classList = objc_copyClassList(&classCount);

// For each class in the list...
for (int i = 0; i < classCount; i++) {

    // Treat the classList as a C array to get a Class from it
    Class currentClass = classList[i];

    // Get the class's name as a string
    NSString *className = NSStringFromClass(currentClass);

    // Log the class's name
    NSLog(@"%@", className);

    NSArray *hierarchy = BNRHierarchyForClass(currentClass);
    NSArray *methods = BNRMethodsForClass(currentClass);

    NSDictionary *classInfoDict = @{
        @"classname" : className,
        @"hierarchy" : hierarchy,
        @"methods"   : methods
    };

    [runtimeClassesInfo addObject:classInfoDict];
}

// You are done with the class list buffer, so free it
free(classList);

// Sort the classes info array alphabetically by name, and log it.
NSSortDescriptor *alphaAsc = [NSSortDescriptor sortDescriptorWithKey:@"name"
                                                               ascending:YES];
NSArray *sortedArray = [runtimeClassesInfo
                       sortedArrayUsingDescriptors:@[alphaAsc]];
NSLog(@"There are %ld classes registered with this program's Runtime.",
      sortedArray.count);
NSLog(@"%@", sortedArray);

}
return 0;
}

```

Compile e execute o seu programa.

Agora, você deve ter muitos resultados nos quais pode ver o nome da classe, hierarquia de herança e lista de métodos para todas as classes registradas com o Runtime pelo seu programa.

Como KVO funciona

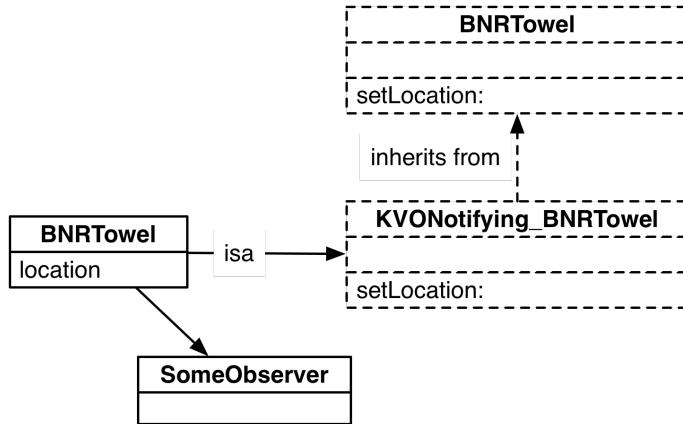
Um exemplo de um API da Apple que depende de funções do tempo de execução como os acima é o Key-Value Observing (Observação de Chave-Valor). Quando você aprendeu sobre KVO no Chapter 36, viu que um observador é automaticamente notificado a respeito de uma mudança em uma propriedade se os acessores do objeto afetado forem usados.

No tempo de execução, quando se envia a mensagem **addObserver:forKeyPath:options:context:** a um objeto, esse método:

- determina a classe do objeto observado e define uma nova subclasse daquela classe usando a função **objc_allocateClassPair**

- altera o ponteiro de `isa` do objeto para apontar para a nova subclasse (efetivamente mudando o tipo do objeto)
- sobrescreve os acessores do objeto observado para enviar mensagens KVO

Figure A.1 Subclasse dinâmica KVO



Por exemplo, considere um setter de classe para uma propriedade `location`:

```

- (void)setLocation:(NSPoint)location
{
    _location = location;
}
  
```

Na nova subclasse, esse acessor seria sobreescrito desta forma:

```

- (void)setLocation:(NSPoint)location
{
    [self willChangeValueForKey:@"location"];
    [super setLocation:location];
    [self didChangeValueForKey:@"location"];
}
  
```

A implementação da subclasse do acessor chama a implementação da classe original e a envolve com mensagens de notificação KVO explícitas. Essas novas classes e métodos são todos definidos no tempo de execução com o uso de funções do Objective-C Runtime. Não há nenhuma mágica aqui.

Para ver a criação de subclasses em ação, adicione uma nova classe ao seu programa `ClassAct` chamada `BNRTowel`. Dê uma única propriedade a ela:

```

@interface BNRTowel : NSObject
// Always know where your towel is!
@property (nonatomic, assign) NSPoint location;
@end
  
```

Compile e execute seu programa. Procure por “Towel” na saída do depurador e você verá que, simplesmente por ter definido a classe do seu programa, você tem uma nova entrada na sua saída – apesar de não ter instanciado um ou de não ter implementado nenhum método por conta própria:

```

{
    classname = BNRTowel;
    hierarchy = (
        NSObject,
        BNRTowel
    );
    methods = (
        location,
        "setLocation:"
    );
}
  
```

Agora, adicione um observador KVO nil a uma instância de **BNRTowel**:

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

        // You don't have an object to do the observing, but send
        // the addObserver: message anyway, to kick off the runtime updates
        BNRTowel *myTowel = [BNRTowel new];
        [myTowel addObserver:nil
            forKeyPath:@"location"
            options:NSKeyValueObservingOptionNew
            context:NULL];

        ...
    }
}
```

Compile e execute seu programa. Procure novamente na saída por “Towel”. Desta vez, além da classe **BNRTowel**, você verá sua subclasse novinha em folha na lista:

```
{  
    classname = "NSKVONotifying_BNRTowel";  
    hierarchy = (br/>        NSObject,  
        BNRTowel,  
        "NSKVONotifying_BNRTowel"  
    );  
    methods = (br/>        "setLocation:",  
        class,  
        dealloc,  
        "_isKVOA"  
    );  
}
```

Observações finais

Se você quiser aprender ainda mais sobre o Objective-C Runtime, vá para a documentação e pesquise no *Objective-C Runtime Programming Guide* (Guia de Programação do Tempo de Execução do Objective-C) e na *Objective-C Runtime Reference* (Referência de Tempo de Execução do Objective-C).

Agora que você aprendeu parte da mágica, deve estar querendo experimentar algumas dessas funções em seu código.

Não faça isso.

Saber que essas funções existem ajuda a abrir as cortinas para a compreensão do que está acontecendo nos bastidores dos seus programas. No entanto, as funções do tempo de execução são principalmente para uso por parte dos desenvolvedores da Apple para dar suporte aos APIs da Apple e podem ser muito difíceis de serem controladas por nós, mortais.

Desafio: variáveis de instância

Modifique seu programa para também registrar todas as variáveis de instância que cada classe tem.

Próximos passos

Bem, isso é tudo o que você precisará saber para escrever extraordinários aplicativos para iOS e Mac OS X.

Isso é o que gostaríamos de poder lhe contar. Sabemos que você trabalhou bastante para chegar até aqui.

A verdade é que você concluiu o primeiro trecho de uma jornada divertida e gratificante. Esta é uma jornada muito longa. Agora, chegou a hora de dedicar algum tempo estudando os frameworks padrão que a Apple disponibiliza para desenvolvedores de Objective-C como você.

Deixe-nos repetir a última frase para você saborear bem: “desenvolvedores de Objective-C como você”. Parabéns.

Se estiver aprendendo a desenvolver aplicativos para iOS, recomendo que você trabalhe com o *iOS Programming: The Big Nerd Ranch Guide* (Programação para iOS: Guia da Big Nerd Ranch). No entanto, há outros inúmeros livros sobre iOS, e você está pronto para qualquer um deles.

Se estiver aprendendo a desenvolver aplicativos para OS X, recomendamos que você trabalhe com o *Cocoa Programming for Mac OS X* (Programação em Cocoa para Mac OS X). Entretanto, há diversos outros livros sobre Cocoa, e você também está pronto para qualquer um deles.

Há muitos grupos de desenvolvedores que se encontram mensalmente para falar sobre seus trabalhos. Na maioria das grandes cidades, há encontros de desenvolvedores para iOS e de CocoaHeads. As palestras são sempre muito boas. Também há grupos de discussão on-line. Reserve algum tempo para encontrar e usar esses recursos.

A minha propaganda

Você pode encontrar nós dois no Twitter. O twitter do Aaron é @AaronHillegass e o do Mikey é @wookiee. Você pode também seguir a Big Nerd Ranch: @bignerdranch.

Fique atento aos futuros guias da Big Nerd Ranch. Oferecemos também cursos de uma semana para desenvolvedores. E, se você precisa apenas de algum código escrito, fazemos contratos de programação. Para obter mais informações, acesse nosso site em: www.bignerdranch.com.

É você, querido leitor, que possibilita nossas vidas de escrever, codificar e ensinar. Então, muito obrigado por adquirir nosso livro.

Index

Symbols

!= (diferente de) operador, 23
\ sequência escape, 280
#define, 165-166, 167-167, 170
#import, 166
#include, 166
% (tokens), 41-42, 42
%@, 131
%d, 42
%e, 47
%f, 47
%ld, 44
%lo, 44
%lu, 44
%o, 44
%p, 57
%s, 42
%u, 44
%x, 44
%zu, 59
()
 em nomes de funções, 14
 para parâmetros de funções, 28
 operadores cast, 45
 ordem das operações e, 22
(cabeçalho pré-compilado) .pch, 167
* (asterisco)
 operador aritmético, 45
 operador de ponteiro, 58
+ (sinal de mais), 45
++ (operador increment), 46
- (sinal de menos), 45
-- (operador decrement), 46
-> operador (desreferenciar), 72
.h, arquivos (see arquivos de cabeçalho)
.m (arquivos de implementação), 115
/ (operador de divisão), 45
/* ... */ (comentários), 12
// (comentários), 12
; (ponto e vírgula), 12
 loop do-while e, 54
< (menor que) operador, 23
< > (colchetes angulares)
 conformidade com protocolos, 202
 importar arquivos, 166
<< operador, 277
<= operador, 23
= operador, 24
== operador, 23, 24
> (maior que) operador, 23
>= operador, 23

token de string de formatação, 131
@interface
 arquivos de cabeçalho, 116
 extensões de classe, 141
 visibilidade de, 142
@property, 121
@selector(), 189
@synthesize, 260
\ (barra invertida), 280
 caractere escape, 42
\n, 42
\\" sequência escape, 280
^ (circunflexo)
 identificar blocos, 191
 operador exclusive-or, 276
{ }

A

abs(), 46
ações (métodos), 179
addObject:, 111
alloc, 87, 249
AND (binário), 275
aninhamento de mensagens, 87-87
aplicativos
 baseado em documento, 229
 baseados em documento, 231, 231
 baseados em GUI, 211
 Cocoa, 229-245
 Cocoa Touch, 211-227
 compilar, 119
 desktop, 229-245
 executar a partir da linha de comando, 291
 executar em um dispositivo, 227
 executar no Xcode, 12
 implementar, 227
 iOS, 211-227
 MVC e, 214
 orientados a eventos, 179, 211
 vs. programas, 7, 179
 projetar, 214
aplicativos baseados em documento, 229, 231, 231
aplicativos baseados em GUI, 211
aplicativos orientados a eventos, 179, 211
application:didFinishLaunchingWithOptions:, 217-223, 220, 227
ARC (Automatic Reference Counting), 134-139
ARC (contagem de referência automática), 93-95
área de edição (Xcode), 10
área de utilitários, 232, 235
argumentos
 em funções, 28
 de
 init, 251

- linha de comando, 292-294
 argumentos de linha de comando, 292-294
 arquivos, 171
 (see also listas de propriedade)
 classe, 115
 gravação em, 171-173, 205
 importar, 127, 166
 leitura em, 206
 ler em, 174
NSData e, 174-176
NSError e, 172-174
 projeto, 119
 arquivos .xib (XML Interface Builder), 231
 arquivos de cabeçalho
 vs. extensões de classe, 141-143
 definidos, 115
 importar, 117
 pré-compilado, 167
 arquivos de cabeçalho pré-compilado, 167
 arquivos de implementação (.m), 115
array, 111
 arrays, 105-110
 acessar por índice, 106
 acesso ao, 111
 aninhamento, 158
 em C, 285-287
 classificar, 160, 161
 criar, 105, 111
 erros de fora da faixa e, 108
 fazer iteração em, 109, 109, 111
 filtrar, 161-162
 imutáveis, 160
 imutável, 106
 mutável, 110-111
 tipos primitivos e, 163
arrayWithContentsOfFile:, 206
arrayWithObjects:, 111
assign (atributo de propriedade), 257
 asterisco (*)
 operador aritmético, 45
 operador de ponteiro, 58
 AT&T, 3
 atalhos de teclado, 245
 atalhos, teclado, 245
atoi(), 56
atomic (atributo de propriedade), 259
 atributos de propriedade
 assign, 257
 atomic, 259
 copiar, 123
 copy, 258-259
 definição, 122
 nonatomic, 259
 readonly, 257
 readwrite, 257
 strong, 257
 unsafe_unretained, 257
 valores padrão, 123
 weak, 257
 atributos objeto-tipo, 133
 autolayout, 236-239
 Automatic Reference Counting (ARC), 134-139
 aviso de visão colocada no lugar errado, 234
 aviso do controlador de visão raiz, 222
 avisos, 182
 (see also erros)
 controlador de visão raiz, 222
 “Definição de método para ... não encontrada”, 214
 variável não utilizada, 182
 visão colocada no lugar errado, 234
- B**
- banco de dados do PostgreSQL, 279
 barra invertida (\), 280
 caractere escape, 42
 base 16 (hexadecimal)
 inteiros, 44
 sistema numérico, 273
 biblioteca de objetos, 232, 232
 biblioteca matemática, 47
 (see also operações aritméticas, números)
 bibliotecas, 120
 matemáticas, 47
 padrão, 30, 30
 vincular binário a, 54, 55
 bibliotecas padrão, 30, 30
 binário OR, 273
 bits, 57, 273
 bits à esquerda, 277
_block palavra-chave, 198
 blocos, 191-198
 anônimos, 196
 capturar variáveis, 197
 self e, 197-198
 typedef e, 195
 valores de retorno de, 196
 variáveis de bloco e, 192
 variáveis externas e, 197-198
 _block palavra-chave, 198
 blocos anônimos, 196
 bloquear (funções), 29
 bloquear (funções, métodos), 29
 bloqueio (funções, métodos), 183
 BOOL (tipo), 24
 buffers, 71
 bugs, 35
 bytes, 57, 273
- C**
- callbacks
 blocos, 191-198
 decidindo entre, 196
 delegação, 183-186
 destino-ação, 180-182
 gerenciamento de memória e, 188, 188

-
- notificações, 186, 187
objetos auxiliares, 183-186
tipos de, 179, 187
caractere de nova linha (\n), 42, 280
caractere escape (\), 42
Caracteres ASCII, 171
caracteres ASCII, 279
categorias, 269-270
char (tipo), 20, 43, 279
char * (tipo), 41
chaves
 nas expressões condicionais, 24, 24
 escopo de, 32, 32, 33
 em funções, 12, 12
 nas declarações de variáveis de instância, 116, 116
ciclos de referências fortes, 147-149
 em blocos, 197
 callbacks e, 188
circunflexo (^)
 identificar blocos, 191
 operador exclusive-or, 276
classes, 115
 (see also objetos, *nomes de classes individuais*)
 definidas, 77
 documentação para, 98-103
 escrever, 115-120
 em frameworks, 78
 herança e, 125-131, 143
 nomenclatura, 120
 nomes, 84
 subclasses, 125-131, 270
classes de coleção, 155-163, 155
 (see also arrays, dicionários, conjuntos)
classificar arrays, 160, 161
Cocoa Touch, 209
codificação de chave-valor (KVC), 261-263
codificações (string), 171
codificações de strings, 171
código
 comentários no, 12, 48
 definido, 11, 12
código assembly, 3
código de amostra (na documentação), 103
código de máquina, 3
chetes angulares (< >)
 conformidade com protocolos, 202
 importar arquivos, 166
comentários (no código), 11, 12, 48
compilação de projetos, 136
compiladores, 3, 14
compilar (projetos Xcode), 12, 119
complementos, 276
conjuntos, 155-156
 filtrar, 162
console, 13
constanteM_PI, 165
constantes, 165-170
contagem de referência automática (ARC), 93-95
contagem de referência manual, 151
contagens de referência, 93
contagens de retenção, 151, 152
containsObject:, 156
controles, 221
copy (atributo de propriedade), 258-259, 258
copy (atributos de propriedade), 258
copy (método), 258, 258
copyWithZone:, 258
count (NSArray), 108
Cox, Brad, 77
CPU (unidade central de processamento), 57
criar regra, 301
currentCalendar, 86
- ## D
- daemons, 7
dataOfType:error:, 244
date (NSDate), 80
dateByAddingTimeInterval:, 85
dealloc, 135
declarações
 método, 116
 propriedade, 121
 variável, 19
 variável de instância, 116
#define, 165-166, 167-167, 170
delegação, 179, 183-186
delegates de aplicativo, 213, 216
DeLillo, Don, 287
depurador, 35, 35-36, 35-36
description (NSObject), 131
descritores de classificação, 160, 161
desenvolvimento para Cocoa, 229-245
 frameworks para, 209
Desenvolvimento para Cocoa
 aplicativos baseados em documento, 229, 231, 231
 usos de autolayout, 236
desenvolvimento para iOS, 211-227
 Cocoa Touch e, 209
 delegates de aplicativo no, 216
 frameworks para, 209
 gerenciamento de memória em, 139
 implementar para um dispositivo, 227
 Objective-C e, 3
 simulador, 219
Desenvolvimento para iOS
 delegates de aplicativo no, 213
 frameworks para, 213
 usos de autolayout, 236
desenvolvimento para Mac (see desenvolvimento para Cocoa)
deslocamento de bits à direta, 277
desreferenciar (->) operador, 72
desreferenciar ponteiros, 58
destino-ação, 179-182
dicionários, 157-160
diretiva super, 129

diretivas de pré-processador, 166-167
 diretórios especiais, 176
 diretórios, gravação de arquivos para, 172
 diretórios, gravar arquivos para, 176
 Diretrizes para Interface Humana (HIGs), 234
 divisão (inteiros), 45
 documentação
 acessar online, 99
 código de amostra, 103
 guias do desenvolvedor, 103
 Quick Help, 106-108
 referências de classe, 98-103
 referências de protocolo, 202
 usar, 98-104
double (tipo), 20
Dreaming in Code (Rosenberg), 5

E

else, 23, 23, 25
else if, 25
 encapsulamento de objeto, 262
 endereços, 57-60
enum, 168
 máscaras de bits, definir, 278
 enumeração rápida, 109, 111
 enumeração, rápida, 109, 111
enumerateObjectsUsingBlock:, 192, 193, 194
 erros, 12, 12, 12, 13
 (see also avisos)
 “@interface não visível...”, 82, 142
 fora da faixa, 108
 “Nenhum método conhecido para seletor...”, 82
NSError e, 172-174, 245
 “...seletor não reconhecido enviado a uma instância...”, 131
 “Uso de identificador não declarado...”, 33
 erros de sintaxe, 12, 12
 escopo, 32, 33
 estrutura do documento, 237
 estruturas (see structs)
 eventos, 211
 exclusive-or (XOR), 276
EXIT_FAILURE, 37
EXIT_SUCCESS, 37
 exponentes, 47
 expressões, 21, 22
 condicionais, 23-26, 25
 expressões condicionais, 23-26, 23, 25
 extensões de classe, 141-143

F

fechamentos, 191
 ferramentas de linha de comando
 em C, 7
 definidas, 7
 Foundation, 78
 File's Owner, 239

filteredArrayUsingPredicate:, 162
filteredSetUsingPredicate:, 162, 162
filterUsingPredicate:, 162
 filtrar, 161-162
 flags (operadores binários), 275
float (tipo), 20
 conversão e, 45
 fontes de dados, 179, 201, 222
for-loop, 109, 109, 111
 frames, 31-36
 framework Foundation, 78, 84, 209
 framework UIKit, 213
 frameworks, 75
 definidos, 78
 Foundation, 78, 84, 209
 UIKit, 213
free(), 71, 301
 funções, 11
 (see also métodos)
 argumentos e, 28
atoi(), 56
 auxiliares, 226
 benefícios de usar, 29
 bloquear, 29, 29
 chamar, 28, 29
 definidas, 11
 escopo, 32, 33
 escrever, 27-29
 frames para, 31-36
main(), 11, 14
 em `math.h`, 47
 metáfora de receita para, 14-14, 29-31
modf(), 63
 noções básicas de, 27-30
 nomes de, 14
 Objective-C, 300, 306
 parâmetros de, 28-29, 30
printf(), 41
 em programas, 29
readline(), 54
 recursivas, 33-36
 tempo de execução, 300, 306
 usos de, 27
 valores de retorno de, 36-37
 variáveis locais em, 30-32
 funções anônimas, 191
 funções auxiliares, 226

G

gerenciamento de memória
 atributos de propriedade e, 257-259
 em C, 71
 callbacks e, 188, 188
 ciclos de referências fortes, 147
 contagem de referência automática (ARC), 134-139
 contagem de referência manual, 151
 desalocação de objetos e, 134

referências fracas e, 149-151
variáveis locais e, 71
gravação em arquivos, 171-173, 205
GUI (interface gráfica do usuário)
criar de forma programática, 245
criar de maneira programática, 217-223
criar no Interface Builder, 231-243, 245
guias do desenvolvedor, 103

H

.h, arquivos (see arquivos de cabeçalho)
heap, 71-73, 93, 93
herança, 125-131, 143
hexadecimal (base 16)
inteiros, 44
sistema numérico, 273

I

IBAction, 240
IBOutlet, 242
id (tipo), 89
if construção, 23-26
#import, 166
importar arquivos, 127, 166
#include, 166
índice (arrays), 106
índices (arrays), 106
inicializadores, 249-256, 256
inicializadores designados, 255
init, 87, 249-256, 252
insertObject:atIndex:, 111
inspetor de atributos, 235
inspetores, 235
instancetype, 250
instanciar (objetos), 172
instrução break, 51
instrução continue, 52
instruções switch, 297-298
int (tipo), 20, 24, 43
conversão e, 45
inteiros, 43-46
inteiros na base 8 (octal), 44
inteiros octais (base 8), 44
interação (arrays), 109
@interface
arquivos de cabeçalho, 116
extensões de classe, 141
visibilidade de, 142
Interface Builder, 230, 231-243, 245
interfaces, 201, 201
(see also interfaces de usuário)
interfaces de usuário
criar de forma programática, 245
criar de maneira programática, 217-223
criar no Interface Builder, 231-243, 245
introspecção, 299
isEqual:, 156

isEqualToString:, 98
iteração (arrays), 109, 111

K

KVC (codificação de chave-valor), 261-263
KVO (observação de chave valor), 265-267
KVO (observar chave-valor), 303-306

L

labs(), 46
lambdas, 191
leitura em arquivos, 206
length (NSString), 97
linguagem de programação C, 4
arrays, 285-287
descrição, 3
strings, 279-284
linguagens de alto nível, 3
linguagens de programação, 3-3
linha de comando, executar a partir da, 291-295
listas de propriedades, 205-207
lógicas binárias, 273-278
long (tipo), 20, 43
long long (tipo), 43
loop de execução, 179, 211
loop do-while, 53
loop for, 51
loops, 49-54
for, 51
para, 109
while, 50
Love, Tom, 77

M

arquivos .m, 78, 115
macros, 167
main(), 11, 14, 27, 35, 227
malloc(), 71, 283, 286, 301
math.h, 47, 166
memória
ARC e, 93-95
endereços, 57-60
gerenciamento, 93-95
heap, 71-73, 93
objetos em, 91-95
pilha, 31-36, 31, 32, 93
mensagens
anatomia de, 79
aninhamento, 87-87
argumentos de, 85-87
definidas, 79
enviando, 79-81, 85-88
envio, 82-82, 88
métodos e, 79
metáfora de receita (funções), 14-14, 29-31
métodos, 11
(see also funções)

- acessores, 118-119
 argumentos de, 85-87
 classe, 81, 97
 declaração, 116
 declarar, 116
 definidos, 77
 documentação para, 99
 implementar, 117
 instância, 81, 97
 mensagens e, 79
 nomes, 83
 protocolo, 201-203
 sobrescrever, 128
- métodos acessores
 implementação, 118
 implementar, 259, 260
 notação de ponto para, 123
 propriedades e, 121
 sobre, 118-119
- métodos de
 conveniência, 153
- métodos de classe, 81, 97
- métodos de conveniência, 153
- métodos de instância, 81, 81, 97
 (see also métodos)
 métodos getter, 118, 119
 métodos setter, 118, 119
- Modelo-Visão-Controlador (MVC), 214, 215
- modf()**, 63
- mutableCopy**, 258, 258
- MVC (Modelo-Visão-Controlador), 214, 215
- ## N
- navegador de depuração, 35
 navegador de projetos, 10
 navegadores, 10
 NeXTSTEP, 84
nil, 88, 91, 94, 111, 163
nonatomic (atributo de propriedade), 259
 notação de ponto, 123
 notificações, 179, 186, 187
NS prefixo, 84
NSArray, 105-110
count, 108
filteredArrayUsingPredicate:, 162
 instâncias literal de , 105
 listas de propriedades e, 205
objectAtIndex:, 111
 objetos imutáveis e, 160
- NSButton**, 232
- NSCalendar**, 86
- NSData**
 gravar em arquivos, 174-175
 ler em arquivos, 175
 listas de propriedades, 205
- NSDate**, 78, 78, 160
 listas de propriedades e, 205
- NSDesktopDirectory**, 176
- NSDictionary**, 157
 listas de propriedades e, 205
- NSDocument**, 244
- NSError**, 172-174, 245
- NSInteger** (tipo), 46
- NSLocale**, 168
- NSLog()**, 79
- NSMutableArray**
 addObject:, 111
array, 111
arrayWithObjects:, 111
 descrito, 110-111
filterUsingPredicate:, 162
insertObject:atIndex:, 111
removeObject:atIndex:, 111
sortUsingDescriptors:, 160, 161
- NSMutableDictionary**, 157
- NSMutableSet**, 155-156
filterUsingPredicate:, 162
- NSNotificationCenter**, 186, 187, 188
- NSNull**, 163
- NSNumber**, 160, 163, 262
 listas de propriedades e, 205
- NSObject**
alloc, 87
 codificação de chave-valor e, 261
dealloc, 135
description, 131
init, 87, 249
 como superclasse, 130
- NSPredicate**, 161-162
- NSRange**, 102
- NSRunLoop**, 179
- NSScrollView**, 234
- NSSearchPathForDirectoriesInDomains()**, 226
- NSSet**, 155-156
containsObject:, 156
filteredSetUsingPredicate:, 162
- NSString**, 97-102, 97
 (see also strings)
 a partir de string em C, 283
 gravação em arquivos, 171
 instâncias literais de, 97
 ler arquivos em, 174
 propriedade, 205
- NSTableView**, 230, 234
- NSTableViewDataSource**, 243
- NSTimer**, 180
- NSUInteger** (tipo), 46
- NSURLConnection**, 174, 175, 183, 183, 185
- NSNumber**, 163
- NS_ENUM()**, 169
- NULL**, 59, 65
- números, 43
 (see also números de ponto flutuante, inteiros)
 com sinal vs. sem sinal, 43
 hexadecimal, 44, 273
 octal, 44

números com sinal, 43
números de ponto flutuante, 47-47
números sem sinal, 43

O

objc_msgSend(), 299, 300
objectAtIndex:, 111
objectForKey:, 159
Objective-C, 3, 77, 83
 convenções de nomes, 83-84
 tempo de execução, 299-306
objetos
 ARC e, 93-95
 criar, 78-80
 definidos, 77
 desalocação, 134
 herança e, 125-131, 143
 imutáveis, 160
 como variáveis de instância, 133-139
 em memória, 91-95
 ponteiros para, 91-95
 relações entre, 133
 vs. structs, 77
objetos auxiliares, 179, 183-186
objetos de controlador (MVC), 215
objetos de erro, 172
objetos idênticos, 157
objetos iguais, 157
objetos imutáveis, 160
objetos modelo (MVC), 214
objetos visão (MVC), 215
observação de chave-valor (KVO), 265-267
observar chave-valor (KVO), 303-306
obter regra, 301
opção **NSDataWritingAtomic** (**NSData**), 175
operações aritméticas, 45-46
operações, ordem das, 22, 45
operador ! (NOT lógico), 24
operador %, 45
operador %=, 46
operador && (AND lógico), 24
operador &, recuperar endereços, 57
operador *=, 46
operador +=, 46
operador -=, 46
operador /=, 46
operador binário AND, 275
operador cast, 45
operador decrement (--), 46
operador increment (++), 46
operador lógico AND (&&), 24
operador lógico NOT (!), 24
operador lógico OR (||), 24
operador módulo (%), 45
operador ternário (?), 25
operador || (OR lógico), 24
operadores (lógicos/de comparação), 23
operadores lógicos, 24, 24, 24

0R (binário), 273
OS X, 3, 3, 3
 (see also desenvolvimento para Cocoa)

P

p-lists, 205-207
palavra-chave **unsigned**, 43
parâmetros, 28-29, 30
parênteses (())
 em nomes de funções, 14
 para parâmetros de funções, 28
 operador cast, 45
 ordem das operações e, 22
pares chave-valor (objetos), 157, 159, 261, 261
passagem por referência, 63-65, 172-173
percursos de chaves, 263
pilha, 31-36, 31, 32
 vs. heap, 93
placeholders, 239
ponteiro **isa**, 130
ponteiro nulo
 nil, 88
ponteiros, 57-60
 armazenar endereços em, 58
 definição, 20
 função, 191
 NSMutableArray e, 110
 para objetos, 91-95
 pendentes, 257
ponteiros de função, 302
ponteiros de funções, 197
ponteiros pendentes, 257
ponto e vírgula (;), 12
 loops do-while e, 54
Pontos de interrupção, 35-36
#pragma mark, 216, 217
pré-processador, 166
precedência (das operações aritméticas), 45
precedência (de operações aritméticas), 22
predicados, 161-162
preenchimento automático, 12
preenchimento de código, 12
prefixos de classe, 213
primeiro respondente, 221
printf(), 41, 47
programas
 vs. aplicativos, 7, 179
 compilar, 12, 14, 119
 criação de perfil, 147
 executar (a partir da linha de comando), 291, 294
 executar (em Xcode), 293
 executar a partir da linha de comando, 291
 executar no Xcode, 12, 21
programas de criação de perfil, 147
projetos
 arquivos em, 119
 compilação, 136
 compilar, 119

criação, 7
 templates para, 212
@property, 121
 propriedade
 definição, 121
 propriedade de objeto
 com callbacks, 188
 com coleções, 162
 propriedade de objetos, 134-139
 propriedades, 265
 (see also métodos acessores, variáveis de instância, KVC, KVO)
@synthesize, 260
 atributos de, 257-259
 declaração, 121
 definidas, 121
 herança e, 143
 implementar acessores com, 260
 implementar métodos acessores com, 259
 métodos acessores e, 121
 notação de ponto e, 123
 variáveis de instância e, 122
 protocolos, 184, 201-203

Q

Quick Help, 106-108

R

RAM (memória de acesso aleatório), 57
rangeOfString:, 102, 102
readFromData:ofType:error:, 244
readline(), 54
readonly (atributo de propriedade), 257
readwrite (atributo de propriedade), 257
 receptores, 79
 recursão, 33-36
 redes, 174, 175, 183, 183, 185
 referências, 63
 (see also ARC, propriedade de objeto, passagem por referência, ciclos de referências fortes)
NSError e, 172
 “ponteiros”, 58
 referências de classe, 98-103
 referências fracas, 149-151
 relacionamento pai-filho (objetos)
 vazamentos de memória e, 146
 relacionamentos pai-filho (objetos), 135
 relacionamentos to-many (para vários), 133
 relacionamentos to-one (para um), 133, 145
removeObjectAtIndex:, 111
resignFirstResponder, 221
respondsToSelector:, 203, 299
 retorno, 36-37
 em blocos, 196
Ruído Branco (DeLillo), 287

S

@selector(), 189
 seletores, 79, 188, 189
self
 em blocos, 197-198
 definido, 119
 sequência escape com aspas duplas (\\"), 280
 sequências de escape, 280
 sequências escape, 42
 sequências escapes, 280
setObject:forKey:, 159
setValue:forKey:, 261, 261
 shell, 291
short (tipo), 20
 simulador (iOS), 219
 sinal de mais (+), 45
 sinal de menos (-), 45
 sinal EXC_BAD_ACCESS, 283
 sintaxe (código), 12
 sintaxe (do código), 12
sizeof(), 59
sleep(), 29, 147
 sobrescrever métodos, 128
sortUsingDescriptors:, 160, 161
stdio.h, 30
 strings, 97-102
 em C, 279-284
 converter strings em C, 283
 criar, 97
 definidas, 41
 encontrar substrings, 100-102
 literais, 97, 282
printf() e, 41
 strings de formatação, 41-42, 42, 44, 47, 79
 Strings UTF, 171
stringWithFormat:, 97
strlen(), 280
strong (atributos de propriedade), 257
 structs, 67-70
 em objetos de coleção, 163
 definição, 20
 vs. objetos, 77
 subclasses, 125, 143, 270
 substrings, 100
 subvisões, 217
 superclasses, 125, 143
@synthesize, 260

T

sequência escape \t, 280
 templates (Xcode), 212
 tempo de compilação, 299
 tempo de execução, 299
 Tempo de execução, Objective-C, 299-300
 temporizadores, 180
 Terminal, 291
 til (~), 276

timeIntervalSince1970, 80

tipos

definir, 68

introdução, 19-22

tipos de dados (see tipos)

tipos primitivos (C), 163

tokens

%@, 131

%d, 42

%e, 47

%f, 47

%ld, 44

%lo, 44

%lu, 44

%lx, 44

%o, 44

%s, 42

%u, 44

%x, 44

em string de formatação, 41-42

inteiro, 44-44, 44

NSLog() e, 79

número de ponto flutuante, 47

ponteiro, 57

printf() e, 41-42

tratamento de erros, 172-174

typedef

com blocos, 195

explicado, 68

U

UIButton, 220

UIKit.h arquivo de cabeçalho, 213

UITableView, 201

UITableViewDataSource, 201-202, 222, 223, 224

unidade central de processamento (CPU), 57

unistd.h, 30

Unix, 3, 80

unsafe_unretained (atributo de propriedade), 257

modificador **__unused**, 182

uppercaseString, 98

URLs, conectar a, 174, 175, 183, 183, 185

V

valor absoluto, 46

valueForKey:, 261

variáveis, 115

(see also variáveis de instância, tipos)

ambiente PATH, 294

atribuição, 21

automáticas, 71

bloco, 192

booleanas, 24

capturadas, 197

declarar, 19, 21

estáticas, 38, 181

externas (em blocos), 197-198

globais, 37-38, 167-170

instância, 133-139

introdução, 19-22

locais, 30-32

não utilizadas, 182

nomes, 83

variáveis automáticas, 71

variáveis booleanas, 24

variáveis capturadas, 197

variáveis de instância

declaração, 116

herança e, 125, 143

nomenclatura, 116

objeto, 133-139

propriedades e, 122

variáveis de instâncias

como privado, 118

variáveis estáticas, 38, 181

variáveis externas (em blocos), 197-198

variáveis globais, 37-38, 167-170, 170

variáveis locais, 30-32, 71

variável de ambiente PATH, 294

vazamentos de memória, 139, 145-151, 147

visão de variáveis, 36

visões

criar de forma programática, 245

criar de maneira programática, 217-223

criar no Interface Builder, 231-243, 245

visões de tabela, 222-225

visões de tabela., 235

void, 28, 36

W

weak (atributo de propriedade), 257

while loops, 50

writeToFile:atomically:, 205

writeToFile:atomically:encoding:error:, 172

X

Xcode

área de edição, 10

área de navegação, 10, 10

área de utilitários, 232, 235

argumentos de linha de comando e, 293

atalhos de teclado, 245

biblioteca de objetos, 232

codificação por cores no, 12

compilação de programas em, 136

compilar programas no, 12

console, 13

criação de classes no, 115

depurador, 35-36

documentação em, 98-104

executar programas em, 293

executar programas no, 12, 21

inspetor de atributos, 235

inspetores, 235

instalar, 7
Interface Builder, 230, 231-243
navegador de depuração, 35
navegador de projetos, 10
preenchimento automático, 12
preenchimento de código, 12
preferências, 12
Quick Help, 106-108
simulador de iOS, 219
sobre o, 7
templates, 7
templates de aplicativos, 212
visão de variáveis, 36
arquivos .xib (XML Interface Builder), 231, 236
XOR (exclusive-or), 276

Z

zero (falso)
NULL e, 60