Objective-C 2.0 Mac 和 iOS 开发实践指南

第 1 章 C，Objective-C 的基础

第 2 章 C 变量

1.1 C 程序的结构

第一部分 Objective-C 简介

本书第一部分是 Objective-C 的简介。Objective-C 是 C 语言的扩展，因此，本部分的前两章对 C 进行了回顾。回顾了 C 之后，介绍了面向对象编程的概念，以及这些概念在 Objective-C 中是如何实现的。第 4 章将带你一行一行地分析一个简单的 Objective-C 程序。

第 1 章　　C，Objective-C 的基础

第 2 章　　C 变量

第 3 章　　面向对象编程简介

第 4 章　　第一个 Objective-C 程序

第 1 章　　C，Objective-C 的基础

Objective-C 是 C 的一个扩展。本书大部分的内容都关注 Objective-C 对 C 添加了什么。但是，要使用 Objective-C 编程，必须知道 C 的基础知识。在 Objective-C 中进行两数相加、在代码中加入一条注释，或者使用一条 if 语句等之类的常规操作时，其方式都是与 C 相同的。Objective-C 非对象的部分不是与 C 相似，或者与 C 类似，它简直就是 C。目前的 Objective-C 2.0 基于 C 的 C99 标准。

本章是回顾 C 的两章中的第 1 章。注意，回顾并不是对 C 的完整介绍，而只是介绍了该语言的基本部分。像位运算符、类型转换的细节、Unicode 字符、带有参数的宏，以及其他

神奇之处并没有提及。回顾部分专门用来帮助那些曾经学过 C 语言的人们进行回忆，或者作为那些擅长从环境中捡起一门新语言的人的快速参考。第 2 章继续回顾 C，并且介绍了声明变量、变量作用域及 C 将变量放置在内存中何处等话题。如果你是一位资深 C/C++程序员，可以跳过这一章（然而，回顾一下也没什么坏处。在编写本章的过程中也学到了一些东西）。如果你从另一种类似 C 的语言（例如 Java 或 C#）转向 Objective-C，你至少应该快速浏览一下这些内容。如果你只有一种脚本语言的编程经验，或者你完全是一名初学者，那么，你会发现在阅读本书的同时阅读一本关于 C 语言的书会很有帮助。

注意　　建议每个人都阅读本书的第 2 章。以我的经验，很多人应该熟悉这些内容，但实际上他们并不是很熟悉。

关于 C 语言的书有很多。最早的 Kernighan 和 Ritchie 所著的《The C Programming Language》一书，仍然是其中最好的一本。大多数人学习 C 语言都是使用的这本书。关于 C 语法，或者说如果你要探究 C 语言的某些边边角角，那么，可以参考 Harbison 和 Steele 所著的《C: A Reference Manual》一书。

考虑一下，你是如何开始学习一门新的自然语言的。要做的第一件事情是不是看看这门语言是如何写的：它使用哪些字母（或象形文字）？它是从左向右读、从右向左读，还是从上向下读？然后，开始学习一些单词。你至少需要从一个较小的词汇表起步。建立起自己的词汇表后，就可以开始用单词组建短语，开始把短语组合为完整的句子。最终，可以把句子组合为完整的段落。

对 C 语言的回顾大致按照同样的过程。1.1 节看看一个 C 程序的结构，C 代码是什么格式，以及命名各种实体的规则和惯例。接下来的小节，介绍变量和运算符，它们大致类似于自然语言中的名词和动词；然后，看看它们如何组合为较长的表达式和句子。随后的小节，介绍控制语句。最后一个小节介绍 C 预处理器，它允许我们在将源文件发送给编译器之前对其进行一些编程式的编辑，该小节还介绍了 printf 函数，它用于字符输出。

1.1　C 程序的结构

本章首先介绍 C 程序结构的基本知识，例如 main 函数、格式化问题、注释、名称和命名惯例，以及文件类型等。

1.1.1 main 函数

1.1.1　main 函数

所有的C程序都有一个main函数。在操作系统载入一个C程序之后，程序从main函数的第一行代码开始执行。main函数的标准形式如下：

```
1.  int main(int argc, const char * argv[])
2.  {
3.    // The code that does the work goes here
4.    return 0;
5.  }
```

main函数的主要特征是：

第一行中的int表示main向操作系统以返回代码的形式返回一个整数值。

名称main是必需的。

第一行剩下的部分是从操作系统传递给程序的命令行参数。main接受argc那么多个参数，并且作为字符串存储于argv数组中。这部分目前不是很重要，可以忽略。

所有的可执行代码放在一对花括号中。

return 0;行表明了0作为返回代码传递回操作系统。在UNIX系统中（包括Mac OS X和iOS），为0的返回代码表示“没有出错”，而其他的任何值则意味着一个某种类型的错误。

如果你对于处理命令行参数或向OS返回一个错误代码不感兴趣（例如，在做后面几章的练习时），可以使用main的一种简化形式：

```
1.  int main( void )
2.  {
3.
4.  }
```

void表明main函数的这个版本不接受参数。在没有一条显式的return语句时，意味着返回值为0。

1.1.2 格式化

1.1.2 格式化

C语句由一个分号结束。空白字符（空格、制表符和换行符）是分隔名称和关键字所必需的。C忽略任何额外的空白：缩进和任何额外的空白对于编译的可执行代码都没有影响；可以自由地使用它们以使代码更具可读性。一条语句可以扩展到多行，例如，如下的3条语句是等价的：

```
distance = rate*time;

    distance     =      rate    *    time;

distance =
    rate *
        time;
```

1.1.3 注释

1.1.3 注释

注释是程序员的启示性表示。编译器会忽略注释。C 支持两种形式的注释：

跟在两个斜杠(//)后面，直到该行结束之前的所有内容，都是一条注释。例如：

```
// This is a comment.
```

在/*和*/之间的任何内容，也是一条注释：

```
/* This is the other style of comment */
```

这种类型的注释可以跨越多行。例如：

```
/* This is
      a longer
          comment. */
```

在编程的过程中，它也可以用来临时性地“注释掉”代码段。这种样式的注释不能像如下这样嵌套：

```
/* /* WRONG - won't compile */ */
```

然而，下面的形式是合法的：

```
/*
    // OK - can nest end of line comment
*/
```

1.1.4 变量和函数名

1.1.4 变量和函数名

C 中的变量和函数名，由字母、数字和下划线(_)字符组成：

第一个字符必须是一个下划线或一个字母。

C 名称是区分大小写的，例如，bandersnatch 和 Bandersnatch 是不同的名称。

在一个名称的中间不能有任何空白。

下面是一些合法的名称：

```
1.  j
2.  taxesForYear2010
3.  bananas_per_bunch
4.  bananasPerBunch
```

如下的名称是不合法的：

```
1.  2010YearTaxes
2.  rock&roll
3.  bananas per bunch
```

1.1.5 命名惯例

1.1.5 命名惯例

为了方便自己及必须阅读代码的任何人，应该为变量和函数使用具有描述性的名称。bpb 很容易录入，但是，当你一年后返回来看时，它会让你感到疑惑；bananas_per_bunch 则是一目了然的。很多普通的 C 程序在较长的变量和函数名中使用下划线来分隔开单词：

```
1.  apples_per_basket
```

Objective-C 程序员通常为变量使用 CamelCase（骆驼命名法）名称。CamelCase 名称使用首字母大写来表示名称中后续单词的开始：

```
1.  applesPerBasket
```

以一个下划线开始的名称，通常用作私有的变量和函数，或者是供内部使用的名称：

```
1.  _privateVariable
2.  _leaveMeAlone
```

然而，这是一个惯例；C 没有强制性的机制来保证变量或函数是私有的。

1.1.6 文件

1.1.6　　文件

普通 C 程序的代码放入到一个或多个以.c 为扩展名的文件中：

```
1.  ACProgram.c
```

注意　　Mac OS X 文件名不区分大小写。文件系统将会记住你用来命名一个文件的大小写，但是，它将 myfile.c、MYFILE.c 和 MyFile.c 当做相同的文件名。

使用 Objective-C 对象（本书从第 3 章开始介绍对象）的代码，放置在以.m 为扩展名的一个或多个文件中：

```
1.  AnObjectiveCProgram.m
```

注意　　由于 C 是 Objective-C 的一个子集，因此将一个普通 C 程序放入到一个.m 文件中也没有问题。

针对定义和实现 Objective-C 类（将在第 3 章介绍）的文件，还有一些命名惯例。但是，C 对于扩展名之前的名称部分，没有任何正式的规则。将包含一个记账程序的代码的文件命名如下，会很傻，但是合法：

```
1.  MyFlightToRio.m
```

C 程序还使用头文件。头文件通常包含了由很多.c 和.m 文件共同使用的各种定义。通过使用一个#include 或#import 预处理器指令，可以将这些文件合并到其他文件中（参见本章稍后的 1.6 节）。头文件有一个.h 扩展名，如下所示：

```
1.  AHeaderFile.h
```

注意　　这一话题超出了本书的讨论范围，但是，在同一个程序中混合使用 Objective-C 和 C++代码是可能的。结果叫做 Objective-C++。Objective-C++代码必须放置在一个扩展名为.mm 的文件中：

```
1.  AnObjectiveCPlusPlusProgram.mm
```

1.2.1 整数类型

1.2 变量

变量是程序中用于某些内存字节的名称。当为一个变量赋一个值时，实际所做的事情是，将该值存储到这些字节中。计算机语言中的变量，就像自然语言中的名词。它们表示程序的问题空间中的项或量。

C 要求通过声明变量来告诉编译器你将要使用的任何变量。变量声明的形式如下：

```
1.  variabletype name;
```

C 允许一次声明多个变量：

```
1.  variabletype name1, name2, name3;
```

变量声明导致编译器为这些变量保留存储空间（内存）。变量的值就是其内存位置的内容。第 2 章将更详细地介绍变量声明。将介绍变量声明放置于何处，变量创建于内存中的何处，以及不同类型的变量的生命期。

1.2.1 整数类型

C 提供了如下类型来保存整数：char、short、int、long 和 long long。表 1-1 给出了 32 位和 64 位 Mac OS X 可执行程序中的整数类型的字节大小（附录 C 将介绍 32 位和 64 位的可执行程序）。

表 1-1 整数类型的大小

类 型	32位	64位
char	1字节	1字节
short	2字节	2字节
int	4字节	4字节
long	4字节	8字节
long long	8字节	8字节

char 类型之所以叫做 char，是因为它最初是用来保存字符的；但是，它经常用作一个 8 位的整数类型。

整数类型可以声明为 unsigned 的：

```
unsigned char a;
unsigned short b;
unsigned int c;
unsigned long d;
unsigned long long e;
```

当 unsigned 单独使用时，表示 unsigned int：

```
unsigned a; // a is an unsigned int
```

无符号变量的位模式总是解释为一个正的数字。如果你给一个无符号变量赋一个负的值，结果是很大的正数。这几乎总是一个错误。

1.2.2 浮点类型

1.2.2 浮点类型

C 的浮点类型有：float、double 和 long double。浮点类型的字节大小在 32 位和 64 位可执行文件中都是相同的：

```
float aFloat;   // floats are 4 bytes
double aDouble; // doubles are 8 bytes
long double aLongDouble;  // long doubles are 16 bytes
```

浮点值总是有符号的。

1.2.3 真值

1.2.3 真值

普通的表达式经常用到真值。计算为零值的表达式认为是假，而计算为非零值的表达式认为是真。

_Bool、bool 和 BOOL

C 的早期版本没有定义布尔类型。普通表达式使用（并且现在仍然使用）布尔值（真值）。正如上文所述，计算为零值的表达式认为是假，而计算为非零值的表达式认为是真。大多数的 C 代码仍然这样编写。

当前的 C 标准 C99，引入了一个_Bool 类型。_Bool 是一个整数类型，它只有两个允许的值，即 0 和 1。赋给_Bool 任何非零值，将得到 1：

```
_Bool b = 35;    // b is now 1
```

如果在源代码文件中包含了文件 stdbool.h，可以使用 bool 作为_Bool 的一个别名，而且可以使用布尔常量 true 和 false（true 和 false 只不过分别定义为 1 和 0）。

```
#include <stdbool.h>
bool b = true;
```

我们很少会在 Objective-C 代码中看到_Bool 或 bool，这是因为 Objective-C 定义了自己的 Boolean 类型 BOOL。第 3 章将介绍 BOOL。

1.2.4 初始化

1.2.4 初始化

变量可以在声明的时候初始化：

```
int a = 9;

int b = 2*4;

float c = 3.14159;

char d = 'a';
```

包含在单引号中的单个字符是一个字符常量。它在数字上等于该字符的编码值。这里，变量 d 的数字值为 97，它是字符 a 的 ASCII 值。

1.2.5 指针

1.2.5 指针

指针是其值等于一个内存地址的一个变量。它“指向”内存中的一个位置。可以通过在声明中，在变量名前面放置一个*，将一个变量声明为一个指针变量。如下的代码把 pointerVar 声明为指向内存中保存一个整数的位置的一个变量：

```
int *pointerVar;
```

一元运算符&（“取址”运算符）用来获取一个变量的地址，以便将其存储到一个指针变量中。如下的代码将指针变量 b 的值设置为整数变量 a 的地址：

```
1  int a = 9;
2
3  int *b;
4
5  b = &a;
```

现在，我们一行一行地解释这个例子：

第 1 行将 a 声明为一个 int 变量。编译器拿出 4 个字节来存储 a，并且将其初始化为值 9。

第 3 行将 b 声明为一个指向 int 的指针。

第 5 行使用&运算符来获取 a 的地址，然后，将 a 的地址赋值给 b。

图 1-1 说明了这个过程（假设编译器分配的起始地址为 1048880）。图中的箭头表示了指向的概念。

图1-1 指针变量

一元运算符*（叫做“求值”或“解引用”运算符），通过使用指向一个内存位置的指针变量，来设置或获取该内存位置的内容。可以这样来看，把表达式*pointerVar 看做是一个别名，即针对存储在 pointerVar 的内容中的任何内存位置的另一个名称。表达式 *pointerVar 可以用来设置或获取该内存位置的内容。

在如下的代码中，将 b 设置为 a 的地址，因此，*b 变成了 a 的别名：

```
1  int a;
```

```
int c;
int *b;
a = 9;
b = &a;
c = *b;  // c is now 9
*b = 10; // a is now 10
```

指针在 C 中用来引用动态分配的内存（参见第 2 章）。指针也用来避免将大块的内存，例如数组和结构（将在本章稍后介绍），从一个程序的一部分复制到另一部分。例如，可以把指向结构的一个指针传递给函数，而不是把一个较大的结构传递给函数。函数随后使用指针来访问结构。随后我们还会看到，Objective-C 对象也总是通过指针来引用。

通用指针

声明为 void 的指针变量，是一个通用指针。

```
void *genericPointer;
```

可以将通用指针设置为任何变量类型的地址：

```
int a = 9;
void *genericPointer;
genericPointer = &a;
```

然而，试图从一个通用指针获取一个值，将导致一个错误发生，因为编译器没办法知道如何解释通用指针所表示的地址的字节：

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *genericPointer;  // WRONG - won't compile
```

要通过 void*指针获取一个值，必须将其转换为指向一个已知类型的指针：

```
int a = 9;
int b;
void *genericPointer;
genericPointer = &a;
b = *((int*) genericPointer) ;  // OK - b is now 9
```

强制转换运算符(int*)迫使编译器将 genericPointer 看做是指向一个整数的指针（参见本章后面的 1.3.10 节）。

C 不会检查一个指针变量是否指向内存的一个有效区域。在 C 编程中错误地使用指针，可能会比其他错误原因引发更多的程序崩溃。

1.2.6 数组

1.2.6 数组

C 数组的声明，首先是数组的类型和数组名，然后是一个方括号，方括号中是数组元素的数目：

```
1.  int a[100];
```

通过将元素的索引放到数组名称后面的[ ]中，可以访问数组的单个元素：

```
1.  a[6] = 9;
```

索引是基于 0 的。在前面的例子中，合法的索引在 0～99。访问 C 数组时系统不会在两端进行边界检查。C 允许做如下的事情：

```
1.  int a[100];
2.  a[200] = 25;
3.  a[-100] = 30;
```

使用超出数组边界的一个索引，将会导致访问属于其他变量的无用内存，这要么导致程序崩溃，要么导致数据毁坏。利用缺乏检查的缺陷，是恶意软件的惯用招数之一。

方括号表示法只是指针算术的一种不错的语法。不带有数组方括号的数组名称，是一个指向数组开始处的指针变量。如下两行代码是完全等价的：

```
1.  a[6] = 9;
2.  *(a + 6) = 9;
```

当要编译的表达式使用指针算术时，编译器会考虑指针所指向的类型的大小。如果 a 是一个 int 数组，表达式*(a+2)指的是距离数组 a 开始处 8 个字节（两个 int）地址处的一个 4 字节内存（这是一个 int 类型的大小）内容。然而，如果 a 是一个 char 数组，表达式*(a+2)指的是距离数组 a 开始处两个字节（两个 char）地址处的一个 1 字节内存（这是一个 char 类型的大小）内容。

多维数组

多维数组声明如下：

```
1.  int b[4][10];
```

多维数组按照行顺序地存储。这里，b[0][0]是第一个元素，b[0][1]是第二个元素，b[1][0]是第 11 个元素。

使用指针表示法为：

```
1.  b[i][j]
```

可以写作：

```
1.  *(b + i*10 + j)
```

1.2.7 字符串

1.2.7　　字符串

C 字符串是字节（char 类型）的一维数组，以一个 0 字节终结。C 中的常量字符串，通过把字符串的字符放置在双引号(″″)之间来编写：

```
1.  "A constant string"
```

当编译器在内存中创建一个常量字符串时，它自动在末尾添加 0 字节。但是，如果声明了一个用来保存字符串的 char 数组，在确定需要多少空间时，则必须记住包含 0 字节。如下的代码行将常量字符串“Hello”的 5 个字符及其最终的 0 字节复制到数组 aString 中：

```
1.  char aString[6] = "Hello";
```

与任何其他的数组一样，表示字符串的数组也不进行边界检查。溢出程序输入所使用的字符串缓冲区，是黑客惯用的招数。

char*类型的变量，可以用一个常量字符串来初始化。可以将这样的一个变量设置为指向一个不同的字符串，但是，不能够使用它来修改一个常量字符串：

```
1.  char *aString = "Hello";
2.
```

```
3.  aString = "World";
4.
5.  aString[4] = 'q';  // WRONG - causes a crash
```

第 1 行把 aString 指向常量字符串“Hello”。第 2 行将 aString 修改为指向常量字符串“World”。第 3 行引发程序崩溃，因为将常量字符串存储到了一段受保护的、只读的内存中。

1.2.8 结构

1.2.8 结构

结构将相关变量组成了一个集合，以便能够将这些变量作为一个单独的实体引用。如下是一个结构声明的例子：

```
1.  struct dailyTemperatures
2.    {
3.      float high;
4.      float low;
5.      int   year;
6.      int   dayOfYear;
7.  };
```

结构中的单个变量叫做成员变量，或者简单地称为变量。跟在关键字 struct 后面的名称，是结构的标签。结构标签标识该结构。它可以用来声明类型为该结构的变量：

```
1.  struct dailyTemperatures today;
2.
3.  struct dailyTemperatures *todayPtr;
```

在前面的例子中，today 是一个 dailyTemperatures 结构，而 todayPtr 是一个指向 dailyTemperatures 结构的指针。

点运算符(.)用来通过一个结构变量访问结构中的单个成员。指针运算符(->)用来从一个变量访问结构成员，而该变量是指向结构的一个指针：

```
1.  todayPtr = &today;
2.
```

```
today.high = 68.0;

todayPtr->high = 68.0;
```

后两条语句完成同样的事情。

结构可以用其他的结构作为成员。前面的例子可以写成如下所示的形式：

```
struct hiLow
{
    float high;
    float low;
};

struct dailyTemperatures
{
    struct hiLow tempExtremes;
    int    year;
    int    dayOfYear;
};
```

那么，设置今天的最高温度可以用如下形式表示：

```
struct dailyTemperatures today;
today.tempExtremes.high = 68.0;
```

注意　　编译器会很容易地在一个结构中插入补充，从而迫使结构成员按照内存中的一个特定边界来对齐。不要通过计算结构成员从结构开始处的偏移量来访问它们，或者做依赖于结构的二进制布局的任何其他事情。

1.2.9 typedef

1.2.9　　typedef

typedef 声明提供了一种方法来为变量类型创建别名：

```
typedef float Temperature;
```

现在，Temperature 可以用来声明变量，就好像它是一个内建的类型一样：

```
1.  Temperature high, low;
```

typedef 只是为变量类型提供了一个替代名称。这里，high 和 low 都仍然是浮点数。关键字 typedef 在 C 代码中往往用作一个动词，就像在“Temperature is typedef 誨 to float”中一样。

1.2.10 枚举常量

1.2.10 枚举常量

enum 语句允许定义一组整数常量：

```
1.  enum woodwind { oboe, flute, clarinet, bassoon };
```

该语句的结果是：oboe、flute、clarinet 和 bassoon，常量值分别为 0、1、2 和 3。

如果不想按照从 0 开始的顺序，可以自己给常量赋值。任何没有被指定值的常量，其值都比前一个常量大 1，如：

```
1.  enum woodwind { oboe=100, flute=150, clarinet, bassoon=200 };
```

上面的语句将 oboe、flute、clarinet 和 bassoon 的值分别指定为 100、150、151 和 200。

关键字 enum 后面的名称叫做枚举标签。枚举标签是可选的。枚举标签也可以用来声明变量：

```
1.  enum woodwind soloist;
2.  soloist = oboe;
```

枚举对于定义多个常量很有用，并且有助于增强代码的可读性，但是，它们不是明确的类型，并且没有得到编译器太多的支持。声明 enum woodwind soloist;表明你希望 soloist 应该限制为 oboe、flute、clarinet 或 bassoon 之一，但是，遗憾的是，编译器没有做任何事情来强化这一限制。编译器认为 soloist 是一个 int，并且，它允许将任何整数值赋给 soloist，而不会产生一条警告：

```
1.  enum woodwind { oboe, flute, clarinet, bassoon };
2.  enum woodwind soloist;
3.  soloist = 5280;  // No complaint from the compiler!
```

注意　　枚举常量占据与变量名称相同的名称空间。所以不能让一个变量和枚举常量具有相同的名称。

1.3.1 算术运算符

1.3　　运算符

运算符就像是动词。它们引发对变量的运算。

1.3.1　　算术运算符

C 中有+、-、*和/这些常用的二元运算符，它们分别用于加、减、乘和除运算。

注意　　如果除法运算符(/)的两个操作数都是整数类型，则 C 执行整数除法。整数除法会把除法的结果进行截取。例如，7 / 3 的值是 2。

1.3.2 余数运算符

1.3.2　　余数运算符

余数运算符或模运算符(%)计算一个整数除法的余数。例如，下面的表达式的结果是 1：

```
1.  int a = 7;
2.  int b = 3;
3.  int c = a%b;  // c is now 1
```

余数运算符的操作数必须都是整数类型。

1.3.3 自增和自减运算符

1.3.3　　自增和自减运算符

C 为自增和自减变量提供了运算符：

```
1.  a++;
2.
3.  ++a;
```

两行都给 a 的值增加 1。然而，当将两个表达式用作一个更大的表达式的一部分时，它们之间就有区别了。前缀版++a，会在任何计算发生之前增加 a 的值。在表达式中使用的是增加以后的 a 值。而后缀版 a++则是在其他计算进行之后才增加其值。在表达式中，使用的

是其最初的值。这可以通过如下的示例来说明：

```
1.  int a = 9;
2.  int b;
3.  b = a++; // postfix increment
4.
5.  int c = 9;
6.  int d;
7.  d = ++c; // prefix increment
```

自增运算符的后缀版，是在将变量的初始值用于表达式的计算之后，才将该变量增加 1。当示例中的代码执行完以后，b 的值是 9，而 a 的值是 10。自增运算符的前缀版，是在该变量值用于表达式计算之前就增加它。因此在这个示例中，c 和 d 的值都是 10。

自减运算符 a--和--a 以类似的方式工作。

使用该运算符的前缀版和后缀版之间的差别的代码，很可能令除编写者之外的人产生混淆。

1.3.4 优先级

1.3.4 优先级

如下的表达式是等于 18 还是 22：

```
1.  2 *7 + 4
```

答案似乎是含糊的，因为这取决于是先进行加法还是先进行乘法运算。C 通过指定一条规则，即在执行加法和减法之前，先执行乘法和除法，从而解决了二义性问题；因此，该表达式的值是 18。从技术性上来说，就是乘法和除法拥有比加法和减法更高的优先级。

如果需要先进行加法运算，可以使用圆括号指定：

```
1.  2 * (7 + 4)
```

编译器将会尊重你的请求，先执行加法，然后再执行乘法。

注意　C 为其所有的运算符定义了一个复杂的优先级表（参见 http://en.wikipedia. org/ wiki/Order_of_operations）。使用圆括号来指定

想要的运算顺序，比努力记住运算符优先级要容易得多。

1.3.5 取反

1.3.5 取反

一元减法符号(-)把一个算术值取反：

```
int a = 9;
int b;
b =  -a;  // b is now -9
```

1.3.6 比较

1.3.6 比较

C 提供了用于比较的运算符。比较的值是一个真值。如下的表达式，如果为真，则取值为 1；如果为假，则取值为 0。

```
a > b // true, if a is greater than b

a < b // true, if a is less than b

a >= b // true, if a is greater than or equal to b

a <= b // true, if a is less than or equal to b

a == b // true, if a is equal to b

a != b // true, if a is not equal to b
```

1.3.7 逻辑运算符

1.3.7 逻辑运算符

AND 和 OR 逻辑运算符的形式如下：

```
expression1 && expression2  // Logical AND operator

expression1 || expression2  // Logical OR operator
```

C 使用短路计算方式。表达式从左向右计算，并且，只要整个表达式可以得出真值，计算就停止。如果一个 AND 表达式中的 expression1 取值为假，那么，整个表达式的值都为假，从而 expression2 就不再计算了。同样，如果一个 OR 表达式中的 expression1 取值为真，那么，整个表达式都为真，从而 expression2 就不再计算了。如果第二个表达式有任何的副作用，短路计算就会得到有趣的结果。在下面的例子中，如果 b 大于或等于 a，就不会调用 CheckSomething()函数（本章后面将要介绍 if 语句）了：

```
1.  if (b < a && CheckSomething())
2.    {
3.      ...
4.    }
```

1.3.8 逻辑取反

1.3.8 逻辑取反

叹号(!)是 C 中的一元逻辑取反运算符。在如下的代码行执行之后，如果 expression 为真（非 0），则 a 的值为 0；并且，如果 expression 的值为假（0），则 a 的值为 1：

```
1.  a = ! expression;
```

1.3.9 赋值运算符

1.3.9 赋值运算符

C 提供了基本的赋值运算符：

```
1.  a = b;
```

上面的语句将 b 的值赋给了 a。当然，a 必须是能够对其赋值的变量。可以对其赋值的实体叫做左值（lvalue）（因为它们放在赋值运算符的左边）。如下是左值的一些例子：

```
1.   /* set up */
2.  float a;
3.  float b[100]
4.  float *c;
5.  struct dailyTemperatures today;
6.  struct dailyTemperatures *todayPtr;
7.  c = &a;
8.  todayPtr = &today;
9.
```

```
/* legal lvalues */
a = 76;
b[0] = 76;
*c = 76;
today.high = 76;
todayPtr->high = 76;
```

有些内容不是左值。不能对一个数组名称、一个函数的返回值或者任何没有引用一个内存位置的表达式赋值：

```
float a[100];
int x;

a = 76; // WRONG
x*x = 76; // WRONG
GetTodaysHigh() = 76; // WRONG
```

1.3.10 转换和强制类型转换

1.3.10 转换和强制类型转换

如果赋值的两端具有不同的变量类型，则右边的类型会转换为左边的类型。从较短的类型转换为较长的类型，或者从整数类型转换为浮点类型，这不会引发问题。但是，反过来，从一个较长的类型转换为一个较短的类型，就可能会导致有意义的数字丢失、截断或者完全无意义。例如：

```
int a = 14;
float b;
b = a;  // OK, b is now 14.0

float c = 12.5;
int d;
d = c;  // Truncation, d is now 12

char e = 128;
int f;
f = e;  // OK, f is now 128

int g = 333;
char h;
h = g;  // Nonsense, h is now 77
```

可以使用强制类型转换，来迫使编译器把一个变量的值转换为不同的类型。在下面的示例中的最后一行，(float)强制转换强迫编译器把 a 和 b 转换为浮点数，并且进行浮点数除法运算：

```
int a = 6;
int b = 4;
float c, d;

c = a / b;  // c is equal to 1.0 because integer division truncates

d = (float)a / (float)b; // Floating-point division, d is equal to 1.5
```

可以强制转换指针，从而将一种类型的指针转换为另一种类型的指针。强制转换指针可能是有风险的操作，因为这可能会毁坏内存，但是，对于以 void*类型传递给你的一个指针来说，这是将其解引用的唯一方式。成功地强制转换一个指针，需要理解指针“实际”所指向的实体的类型。

1.3.11 其他赋值运算符

1.3.11 其他赋值运算符

C 还有其他的快捷运算符，它们把计算和赋值组合了起来：

```
a += b;
a -= b;
a *= b;
a /= b;
```

其等价的形式分别如下：

```
aa = a + b;

aa = a - b;

aa = a * b;

aa = a / b;
```

1.4.1 表达式

1.4　　表达式和语句

C 中的表达式和语句相当于自然语言中的短语和句子。

1.4.1　　表达式

最简单的表达式只是单个的常量或变量：

```
1.  14
2.
3.  bananasPerBunch
```

每个表达式都有一个值。常量表达式的值，就是常量自身，例如，14 的值就是 14。变量表达式的值就是变量所保存的内容，例如，bananasPerBunch 的值，就是在初始化或赋值的时候为其最终设置的值。

表达式可以组合以形成其他的表达式。如下的形式也是表达式：

```
1.  j + 14
2.  a < b
3.  distance = rate * time
```

算术或逻辑表达式的值，就是通过执行算术或逻辑运算所得的结果。赋值表达式的值，就是作为赋值运算的目标的变量所得到的值。

函数调用也是表达式：

```
1.  SomeFunction()
```

函数调用表达式的值就是函数的返回值。

1.4.2 计算表达式

1.4.2　　计算表达式

当编译器遇到一个表达式时，它创建二进制代码来执行表达式并得到其值。对于原型表达式，没有什么事情可做，其值就是它们自身。对于较为复杂的表达式，编译器会生成执行特定算术计算、逻辑运算、函数调用和赋值的二进制代码。

计算表达式可能引起副作用。最常见的副作用是，由于赋值而修改了一个变量的值，或者由于函数调用而执行了函数中的代码。

在各种控制结构中，表达式的值用来决定一个程序的流程（参见后面的 1.5 节）。在其他情况下，可能计算表达式，只是为了得到计算它们的副作用。通常，一个赋值表达式的位置，就是赋值所发生的地方。在极少数情况下，值和副作用二者都很重要。

1.4.3 语句

1.4.3 语句

当在表达式的末尾添加一个分号(;)时，它就变成了一条语句。这类似于在自然语言中，给一个短语添加一个句点来得到一个句子。代码中的一条语句等同于一个完整的想法。当通过编译一条语句而得到的所有机器语言指令都执行完毕，并且，该语句所影响到的所有内存位置的修改也都已经完成时，该语句的执行也就完成了。

1.4.4 复合语句

1.4.4 复合语句

在能够使用单条语句的任何地方，都可以使用一系列的语句，不过要用一对花括号将其括起来：

```
{
  timeDelta = time2 - time1;
  distanceDelta = distance2 - distance1;
  averageSpeed = distanceDelta / timeDelta;
}
```

在结束花括号的后面没有分号。像这样的一组语句，叫做复合语句或语句块。复合语句经常与控制语句一起使用。

注意　使用“块（block）”这个词来作为复合语句的同义词，在 C 的描述中很常见，这可以追溯到 C 语言的创始之初。遗憾的是，Apple 已经采用“块”来表示其对 C 添加的闭包（参见第 16 章）。为了避免混淆，本书后面的部分使用更加复杂一点的名称，即复合语句。

1.5 程序流程

1.5 程序流程

程序中的语句是顺序执行的，除非由一个 for、while、do-while、if、switch 或 goto 语句或一个函数调用将流程导向到其他地方去做其他的事情。

一条 if 语句根据一个表达式的真值来有条件地执行代码。

for、while 和 do-while 语句用于构建循环。在循环中，重复地执行相同的语句或一组语句，直到满足一个条件为止。

switch 语句根据一个整数表达式的算术值，来选择一组语句执行。

goto 语句无条件地跳转到一条标记的语句。

函数调用跳入到函数体中的代码。当该函数返回时，程序从函数调用之后的位置开始执行。

些控制语句都将在后面的小节中详细介绍。

注意　　在阅读后面的内容时，记住，凡是提到语句的每个地方，我们都可以使用复合语句。

1.5.1 if

1.5.1　　if

if 语句根据一个表达式的真值来有条件地执行代码。其形式如下：

```
1.  if ( expression )
2.  
3.    statement
```

如果 expression 计算为真（非零），将执行 statement；否则，从 if 语句之后的下一条语句开始继续执行。可以通过添加 else 部分来扩展一条 if 语句：

```
1.  if ( expression )
2.  
3.    statement1
4.  
5.  else
6.  
7.    statement2
```

if 表达式为真（非零），将执行 statement1；否则，执行 statement2。

也可以通过添加 else if 部分来扩展 if 语句，如下所示：

```
if ( expression1 )
  statement1
else if ( expression2 )
  statement2
else if ( expression3 )
  statement3
...
else
  statementN
```

这个表达式按照顺序执行。当表达式的值非零时，对应的语句会执行，并且从 if 语句之后的下一条语句开始继续执行。如果表达式都为假，则 else 子句后面的语句会执行（就像一条简单的 if 语句一样，else 子句是可选的，并且可以省略）。

1.5.2 条件表达式

1.5.2 条件表达式

条件表达式由 3 个子表达式组成，其形式如下：

```
expression1 ? expression2 : expression3
```

当计算条件表达式时，先计算 expression1 的值。如果它为真，将会执行 expression2，并且整个表达式的值就是 expression2 的值。expression3 不会执行。

如果 expression1 为假，则执行 expression3 并且条件表达式的值就是 expression3 的值。expression2 不会执行。

条件表达式往往用作一条简单的 if 语句的一种缩写形式。例如：

```
a = ( b > 0 ) ? c : d;
```

等价于：

```
if ( b > 0 )
{
  a = c;
}
else
{
  a = d;
```

1.5.3 while

1.5.3 while

while 语句用来构成循环，如下所示：

```
while ( expression ) statement
```

当 while 语句执行时，计算 expression 的值，如果为真，则执行 statement 并且再次计算条件。重复这一过程，直到表达式的值为假。此时，从 while 后面的下一条语句开始继续执行。偶尔会见到这种结构：

```
while ( 1 )
  {
    ...
  }
```

从 while 的角度来看，这是一个无限循环。假设循环体内的内容是检查一个条件，并且当该条件满足的时候，跳出循环。

1.5.4 do-while

1.5.4 do-while

do-while 语句类似于 while，区别在于，它的测试位于 statement 之后，而不是在其之前：

```
do statement while ( expression );
```

其结果是，不管 expression 的值是什么，statement 总是会执行一次。即便条件为假，循环体也至少要执行一次，然而，存在这种程序逻辑的情况并不常见。因此，do-while 语句在实际中很少用到。

1.5.5 for

1.5.5 for

for 语句是最常用的循环结构。其形式如下：

```
1.  for (expression1; expression2; expression3) statement
```

当执行一条 for 语句时，会按照如下顺序进行：

1）在循环开始前，计算一次 expression1。

2）计算 expression2 的值。

3）如果 expression2 为真，则执行 statement；否则，循环结束，继续从循环后的下一条语句开始执行。

4）计算 expression3。

5）重复步骤 2、3、4 直到 expression2 变为假。

计算 expression1 和 expression3 仅仅是为了其副作用。它们的值会丢弃。它们通常用来初始化和自增一个循环计数器变量：

```
1.  int j;
2.
3.  for ( j=0; j < 10; j++ )
4.    {
5.      // Something that needs doing 10 times
6.    }
```

在 for 结构中，任何表达式都可以省略（分号必须保留）。如果省略了 expression2，则循环就是一个无限循环，类似于 while(1)：

```
1.  for ( i=0; ; i++ )
2.    {
3.      ...
4.      // Check something and exit if the condition is met
5.    }
```

注意　　当使用循环来遍历一个数组的元素时，记住，数组的索引是从 0 开始的，直到数组中元素的数目减 1 为止：

```
int j;
int a[25];

for (j=0; j < 25; j++ )
  {
    // Do something with a[j]
  }
```

在前面的示例中，常见的错误是，把 for 循环编写为 for (j=1; j <= 25; j++)。

1.5.6 break

1.5.6　　break

break 语句用来跳出一个循环或一条 switch 语句。

```
int j;
for (j=0;  j < 100; j++ )
 {
    ...

    if ( someConditionMet ) break;
//Execution continues after the loop
 }
```

从 while、do、for 或 switch 语句末尾后面的下一条语句开始继续执行。当有嵌套循环时，break 只是从最内层的循环跳出。编写一条 break 语句，而没有一个循环或 switch 结构包围它，这将会导致一个编译器错误：

```
error: break statement not within loop or switch
```

1.5.7 continue

1.5.7　　continue

continue 用于 while、do 或 for 循环的内部，用来取消当前循环迭代的执行。例如：

```
1.        int j;
2.        for (j=0;  j < 100; j++ )
3.         {
4.            ...
5.
6.            if ( doneWithIteration ) continue; // Skip to
   the next iteration
7.            ...
8.         }
```

当执行 continue 语句时，控制传递给循环的下一次迭代。在 while 或 do 循环中，控制表达式针对下一次迭代而计算。在 for 循环中，计算迭代表达式（即第三个表达式），然后，计算控制表达式（即第二个表达式）。编写一条 continue 语句，而没有一个循环包围它，这将会导致一个编译器错误。

1.5.8 逗号表达式

1.5.8 逗号表达式

逗号表达式由逗号隔开的两个或多个表达式组成：

```
1.  expression1, expression2, ..., expressionN
```

该表达式按照从左到右的顺序计算，并且，整个表达式的值就是最右侧的子表达式的值。

逗号运算符的首要用法就是，在一个 for 循环中初始化和更新多个循环变量。在如下示例的循环迭代中，j 从 0 到 MAX-1，并且 k 从 MAX-1 到 0：

```
1.  for ( j=0, k=MAX-1; j < MAX; j++, k--)
```

```
2.   {
3.     // Do something
4.   }
```

在一个 for 循环中使用逗号表达式时，只有计算子表达式的副作用（在前面的例子中，就是初始化并自增或自减 j 和 k）是重要的。逗号表达式的值会被丢弃。

1.5.9 switch

1.5.9 switch

switch 根据一个整数表达式的值来分支到不同的语句。switch 语句的形式如下所示：

```
1.  switch ( integer_expression )
2.   {
3.     case value1:
4.       statement
5.       break;
6.
7.     case value2:
8.       statement
9.       break;
10.    ...
11.
12.    default:
13.      statement
14.      break;
15. }
```

这里与 C 的其他部分稍微有点不一致，每个 case 可以有多条语句，而不需要一条复合语句。

value1, value2, ...必须是整数、字符常量或者计算为一个整数的常量表达式（换句话说，它们在编译时必须得到一个整数）。不允许具有相同的整数值的重复的 case。

当执行一条 switch 语句时，计算 integer_expression，并且，switch 将结果与整数 case 标签相比较。如果找到一个匹配，执行将跳到匹配的 case 标签后面的语句。执行顺序进行，直到遇到一条 break 语句或到达了 switch 的末尾。break 语句会导致执行跳出到 switch 之后的第一条语句。

case 后面并不一定必须有一条 break 语句。如果省略了 break，则执行将跳入到后续的 case。如果你看到已有的代码中省略了 break，这可能是一个错误（这是很容易犯的错误），也可能是有意的(如果程序员想要一个 case 及其后续的 case 都执行相同的代码会这样做)。

integer_expression 没有和任何 case 标签匹配，如果有该标签的话，执行将跳到可选的 default:标签后面的语句。如果没有匹配也没有 default:，那么 switch 什么也不做，它将从 switch 后面的第一条语句开始继续执行。

1.5.10 goto

1.5.10 goto

C 提供了一条 goto 语句：

```
goto label;
```

执行 goto 语句时，控制将会无条件地跳转到 label 所标记的语句：

```
label: statement
```

标签不是可执行的语句，它们只是标记出代码中的一个位置。

命名标签的规则与命名变量和函数的规则相同。

标签总是以一个冒号结束。

滥用 goto 语句，可能会导致杂乱的、令人混淆的代码（通常称之为面条式代码）。通常的标准建议是，尽量不要使用 goto 语句。尽管如此，goto 语句在某些情况下还是有用的，例如，跳出嵌套的循环（break 语句只能够跳出最内层的循环）：

```
for ( i=0; i < MAX_I; i++ )
  for ( j=0; j < MAX_J; j++ )
     {
        ...
       if ( finished ) goto moreStuff;
     }

moreStuff: statement      // more statements
```

1.5.11 函数

1.5.11 函数

函数通常的形式如下：

```
returnType functionName( arg1Type arg1, ..., argNType argN )
```

```
{
 statements
}
```

一个简单函数示例如下：

```
float salesTax( float purchasePrice, float taxRate )
{
  float tax = purchasePrice * taxRate;
  return tax;
}
```

可以这样来调用一个函数：先写函数名，后面跟着一个括号括起来的表达式列表，其中每个表达式对应函数的一个参数。每个表达式类型必须与声明中的函数参数的类型一一对应。如下的示例展示了一个简单的函数调用：

```
float carPrice = 20000.00;
float stateTaxRate = 0.05;


float carSalesTax = salesTax( carPrice, stateTaxRate );
```

当执行函数调用的那行代码时，控制跳转到函数体中的第一条语句。执行继续进行，直到遇到一条 return 语句，或者到达函数的末尾。执行随后返回到调用环境中。在调用环境中，函数表达式的值就是 return 语句所设置的值。

注意　　函数并非必须有参数或返回一个值。没有返回值的函数，其类型为 void：

```
void FunctionThatReturnsNothing( int arg1 )
```

在不返回值的函数中，可以省略 return 语句。通过对参数列表使用空的圆括号，来表示不接受任何参数的函数：

```
int FunctionWithNoArguments()
```

函数有时候仅仅为了其副作用而执行。下面这个函数打印出了销售税，但是，它对程序的状态没有做出任何修改：

```
void printSalesTax ( float purchasePrice, float taxRate )
{
```

```
  float tax = purchasePrice * taxRate;
  printf( "The sales tax is: %f.2\n", tax );

}
```

C 函数根据值来调用。当调用一个函数时，计算调用语句的参数列表中的表达式，并且将它们的值传递给函数。函数不能直接修改调用环境中的任何变量的值。下面这个函数对于调用环境中的任何内容没有影响：

```
void salesTax( float purchasePrice, float taxRate, float carSalesTax
)
{
  // Changes the local variable calculateTax but not the value of
  // the variable in the calling context

     carSalesTax = purchasePrice * taxRate;
     return;
}
```

要改变调用环境中的变量的值，必须传入一个指向该变量的指针，并且，使用指针来操作该变量的值：

```
void salesTax( float purchasePrice, float taxRate,
float *carSalesTax)
{
  *carSalesTax = purchasePrice * taxRate; // this will work

  return;
}
```

注意　　前面的例子仍然是按照值来调用的。将调用环境中的一个变量的指针的值，传递给了该函数。然后，该函数使用这个指针（它没有修改）来设置它所指向的变量的值。

1.5.12 声明函数

1.5.12　　声明函数

当调用一个函数时，编译器需要知道函数的参数和返回值的类型。它使用这一信息来建立函数与其调用者之间的通信。如果函数的代码出现在函数调用之前（在源代码文件中），那么你不必做任何其他的事情。如果函数代码在函数调用的后面，或者位于另一个文件中，则在使用函数之前必须先声明它。

函数声明重复了函数的第一行，而且在末尾添加了一个分号：

```
1.  void printSalesTax ( float purchasePrice, float taxRate );
```

常用的做法是将函数声明放入到一个头文件中。然后，将头文件包含到（参见后面的小节）使用该函数的任何文件中。

注意　　忘记声明函数，可能会导致很隐蔽的错误。如果调用一个函数，而这个函数的代码在另一个文件中（或者在同一个文件中，但是位于函数调用之后），并且，你没有声明该函数，那么编译器和连接器都不会给出提示。但是，该函数将会针对任何浮点参数接收垃圾信息，并且，如果函数的返回类型也是浮点数的话，那么它会返回垃圾信息。

1.6.1 包含文件

1.6　　预处理器

当编译 C（和 Objective-C）代码文件时，在将它们发送给相应的编译器之前，首先将它们发送给一个初始化程序，叫做预处理器。以一个#字符开始的行，是给预处理器的指令。使用预处理器指令，我们可以：

将一个文件的文本导入到指定位置的一个或多个文件中。

创建定义的常量。

有条件地编译代码（根据条件编译或忽略语句块）。

1.6.1　　包含文件

如下的一行代码：

```
1.  #include "HeaderFile.h"
```

导致预编译器把 HeaderFile.h 文件的文本插入到要编译的文件中#include 行所在的位置。其效果就等同于，使用一个文本编辑器把 HeaderFile.h 的文本复制并粘贴到要编译的文件中。

如果被包含的文件放在双引号中(""):

```
1.  #include "HeaderFile.h"
```

那么预处理器将首先在要编译的文件所在的同一目录下查找 HeaderFile.h，然后在可以作为参数提交给编译器的位置列表中查找，最后在一系列的系统位置中查找。

如果包含的文件放在尖括号中(<>):

```
1.  #include <HeaderFile.h>
```

预处理器将只在标准系统位置中查找包含的文件。

注意　　在 Objective-C 中，#include 被#import 所取代了，后者也产生同样的结果，只不过它不允许指定的文件导入多次。如果预处理器遇到了同一个头文件的多条#import 指令，这些指令将会被忽略。

1.6　　预处理器

当编译 C（和 Objective-C）代码文件时，在将它们发送给相应的编译器之前，首先将它们发送给一个初始化程序，叫做预处理器。以一个#字符开始的行，是给预处理器的指令。使用预处理器指令，我们可以：

将一个文件的文本导入到指定位置的一个或多个文件中。

创建定义的常量。

有条件地编译代码（根据条件编译或忽略语句块）。

1.6.1　　包含文件

如下的一行代码：

```
1.  #include "HeaderFile.h"
```

导致预编译器把 HeaderFile.h 文件的文本插入到要编译的文件中#include 行所在的位置。其效果就等同于，使用一个文本编辑器把 HeaderFile.h 的文本复制并粘贴到要编译的文件中。

如果被包含的文件放在双引号中(""):

```
1.  #include "HeaderFile.h"
```

那么预处理器将首先在要编译的文件所在的同一目录下查找 HeaderFile.h，然后在可以作为参数提交给编译器的位置列表中查找，最后在一系列的系统位置中查找。

如果包含的文件放在尖括号中(<>):

```
1.  #include <HeaderFile.h>
```

预处理器将只在标准系统位置中查找包含的文件。

注意　　在 Objective-C 中，#include 被#import 所取代了，后者也产生同样的结果，只不过它不允许指定的文件导入多次。如果预处理器遇到了同一个头文件的多条#import 指令，这些指令将会被忽略。

1.6.2 #define

1.6.2　　#define

#define 用于环境替换。#define 最常见的用法是定义常量，如下所示：

```
1.  #define MAX_VOLUME 11
```

预处理器会把将要编译的文件中的每个 MAX_VOLUME 都替换为 11。通过在定义中除最后一行以外的所有行的末尾放置一个反斜杠(\)，一个#define 可以延续到多行。

注意　　如果你这么做，那么\必须是该行的最后一项。如果在\的后面再跟着某些内容的话（例如，以“//”开始的一条注释），将会导致一个错误。

经常采用的一种做法是，把#define 放置到一个头文件中，然后，将其包含到各种源文件中。这样，就可以通过在头文件中更改单个的值，从而改变该常量在所有源文件中的值。

传统的 C 对于定义常量所使用的命名惯例是，使用全部大写字母。传统的 Apple 命名惯例是以一个 k 开始常量名称，剩下的名称使用 CamelCase 法命名：

```
1.  #define kMaximumVolume 11
```

两种形式你都会遇到，有时候它们会出现在同一段代码中。

1.6.3 条件编译

1.6.3 条件编译

预处理器允许条件编译：

```
1.  #if condition
2.
3.    statements
4.
5.  #else
6.
7.    otherStatements
8.
9.  #endif
```

这里，condition 必须是在编译时可以计算为真值的一个常量表达式。如果 condition 计算为真（非零），将会编译 statements，而不会编译 otherStatements。如果 condition 为假，将会略过 statements，而编译 otherStatements。

#endif 是必需的，但是#else 和替代的代码是可选的。条件编译语句块必须以一个 #ifdef 指令开始：

```
1.  #ifdef name
2.
3.    statements
4.
5.  #endif
```

这个示例的行为和前面的示例相似，只不过#ifdef 的真值由 name 是否已经被#define 过来决定。

#if 的用途之一是，在调试的过程中，可以很容易地删除或替换代码块：

```
1.  #if 1
```

```
  statements
#endif
```

通过把 1 更改为 0，可以临时忽略 statements 而进行一项测试。随后，再通过把 0 更改为 1 来取代该测试。

#if 和#ifdef 指令可以嵌套，如下所示：

```
#if 0
#if 1
statements
#endif
#endif
```

在前面的例子中，编译器忽略了位于#if 0 及其对应的#endif 之间的所有代码，包括其他的编译器指令。statements 不会被编译。

如果需要关闭或重新打开多个语句块，可以像下面这样编写每个语句块：

```
#if _DEBUG

  statements

#endif
```

可以将定义的常量_DEBUG 使用编译器的_D 标志添加到一个头文件中，或从头文件中删除。

1.7 printf

1.7　　printf

输入和输出(I/O)不是 C 语言的一部分。字符和二进制 I/O 都是通过 C 标准 I/O 库中的函数来处理的。

注意　　标准 I/O 库是与每个 C 环境一起提供的一组函数库。

要使用标准 I/O 库中的函数，必须在自己的程序中包含该库的头文件：

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

这里唯一介绍的函数是 printf，它把字符串打印到终端窗口（或者，如果你使用 Xcode 的话，是打印到 Xcode 控制台窗口）。printf 函数接受可变数目的参数。printf 的第一个参数是一个格式字符串。其他的任何参数，都是以格式字符串所指定的方式打印出来的内容：

```
1.  printf( formatString, argument1, argument2, ... argumentN );
```

格式字符串包含了普通字符和转换修饰符：

控制字符串中的普通字符（没有%）不经修改就可以发送给输出。

转换修饰符以一个百分号（%）开始。跟在%后面的字母表示了修饰符所期待的参数的类型。

每个转换修饰符按照顺序作用于格式字符串后面的参数之一。参数转换为表示参数的值的字符，并且，这些字符发送给输出。

本书使用的转换修饰符只有用于 char 和 int 的%d，用于 float 和 double 的%f，用于 C 字符串的%s。C 字符串表示为 char*。

下面是一个简单的例子：

```
1.  int myInt = 9;
2.  float myFloat = 3.145926;
3.  char* myString = "a C string";
4.
5.  printf( "This is an Integer: %d, a float:
    %f, and a string: %s.\n",
6.      myInt, myFloat, myString );
```

注意　　\n 是一个换行字符。它放在输出内容之前，以便任何后续的输出都出现在下一行。

前面的示例结果是：

```
1.  This is an Integer: 9, a float: 3.145926,
    and a string: a C string.
```

如果跟在格式字符串后面的参数的数目，与转换修饰符的数目不一致，那么 printf 会忽略掉多余的参数或者针对多余的修饰符打印出垃圾信息。

注意　　本书只是使用 printf 来记录和调试非对象变量，而不会将其用于一个正规程序的输出，因此，本节只是粗略地介绍一下格式字符串和转换修饰符。

printf 处理大量的类型，并且它提供了对输出外观的非常精细的控制。关于转换修饰符的可用类型及如何控制格式化细节的完整介绍，可通过 UNIX 的 man 命令来了解。要查阅它们，在终端窗口中输入如下内容：

```
1.  man 3 printf
```

注意　Foundation 框架提供了 NSLog，这是另一个日志函数。它类似于 printf，但是，它添加了打印对象变量的功能。它还添加了对程序名称、日期，以及用小时、分钟、秒和毫秒表示的时间的输出。如果只想知道一两个变量的值，那么，这些添加的信息可能在视觉上会分散你的注意力，因此，本书在不需要 NSLog 的额外功能的某些地方使用了 printf。NSLog 将在第 3 章中介绍。

1.8 使用 gcc 和 gdb

1.8　　使用 gcc 和 gdb

当为 Mac OS X 或 iOS 编写程序时，应该使用 Xcode 来编写和编译自己的程序，Xcode 是 Apple 的集成开发环境。本书第 4 章介绍了如何建立一个简单的 Xcode 项目。然而，对于本章和第 2 章中的练习所需的简单 C 程序，你会发现，在自己喜爱的文本编辑器中编写程序，然后使用 GNU 编译器 gcc 通过命令行编译和运行它们，会更容易。为此，需要：

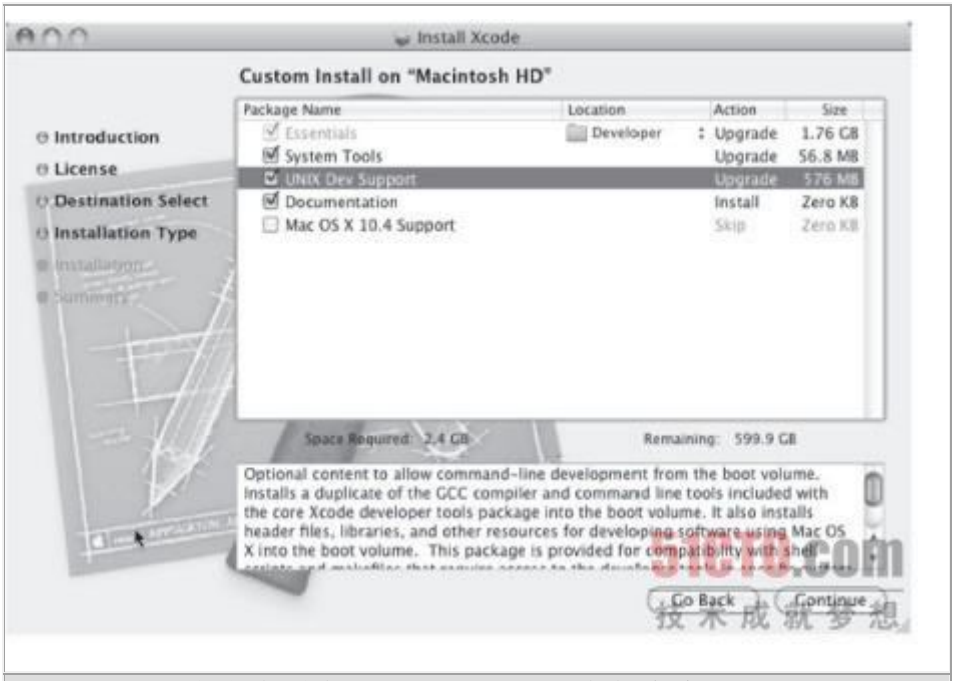
1）一个终端窗口。使用 Mac OS X 所带的 Terminal app（/Applications/Terminal）。如果你使用另一种 UNIX 环境，并且习惯了 xterms，则可能需要下载和使用 iTerm，这是 OS X 本地终端应用程序，它的作用和 xterm（http://iterm.sourceforge.net/）类似。

2）一个文本编辑器。Mac OS X 带有 vi 和 emacs，或者你可以使用自己所拥有的一款不同的编辑器。

3）命令行工具。这可能还没有在你的系统上安装。要检查一下，在命令提示行中输入 gcc。如果得到的回应是/usr/bin/gcc，则说明它们都安装好了。然而，如果没有回应，或者回应是 gcc: Command not found，则必须从安装盘来安装命令行工具，或者从下载的 Xcode 磁盘镜像来安装（可以在 Mac Dev Center Web 页面 http://developer.apple.com/mac/中找到开发者工具的当前版本的一个链接）。开始安装过程，然后，当你进入 Custom Install 阶段时，确保选中 UNIX Dev Support，如图 1-2 所示。继续安装过程。

现在准备好进行编译了。如果源代码文件名为 MyCProgram.c，可以通过在命令提示行中输入如下命令来编译它：

```
1.  gcc -o MyCProgram MyCProgram.c
```



（点击查看大图）图 1-2　　安装命令行工具

-o 标志允许给编译器指定一个名称，以用于最后的可执行文件。如果编译器指出你犯了一两个错误，那么返回去修正错误，然后再次尝试。当你的程序成功通过了编译，可以在命令提示行输入可执行文件名称来运行它：

```
1.  MyCProgram
```

如果想要使用 GNU 调试器 gdb 调试你的程序，则在编译时必须使用-g 标志：

```
1.  gcc -g -o MyCProgram MyCProgram.c
```

-g 标志使 gcc 将 gdb 的调试信息附加到最终的可执行文件。要使用 gdb 来调试一个程序，输入 gdb，后面跟着可执行文件的名称：

```
1.  gdb MyCProgram
```

可以在 GNU 的 Web 站点 www.gnu.org/software/gdb/或者 Apple 站点 http://developer.apple.com/mac/library/documentation/DeveloperTools/gdb/gdb/gdb_toc.html 找到关

于 gdb 的文档。此外，还有很多站点提供了关于使用 gdb 的说明。可通过搜索“gdb tutorial”找到相关文档。

1.9 小结

1.9 小结

本章回顾了 C 语言的基本部分。第 2 章将继续回顾 C 程序的内存布局、声明变量、变量作用域和生命周期，以及内存的动态分配等内容。第 3 章开始进入本书的主题：面向对象编程和 Objective-C 的对象部分。

1.10 练习

1.10 练习

1. 编写一个函数，返回两个浮点数的平均值。编写一个小程序，来测试你的函数，并记录输出。接下来，将函数放入到一个单独的源文件中，但是特意在拥有 main 函数的文件中不声明该函数，看看会发生什么情况。现在，在有 main 程序的文件中添加函数声明，并且验证该声明解决了出现的问题。

2. 编写另一个平均数函数，但是，这一次，尝试将结果通过函数的某个参数传递回来。你的函数应该按如下声明：

```
void average( float a, float b, float average )
```

编写一个小的测试程序，并且验证你的函数无法工作。这说明你不能通过设置一个函数的参数来影响调用环境中的一个变量。现在，修改该函数及其调用，以传递一个指向调用环境中的变量的指针。验证该函数可以使用指针来修改调用环境中的一个变量。

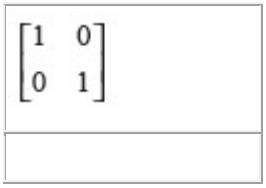
3. 假设你有一个函数，int FlipCoin()，它随机地返回一个 1 表示头部或一个 0 表示尾部。说明如下的代码段是如何工作的：

```
int flipResult;

if ( flipResult = FlipCoin() )
  printf("Heads is represented by %d\n", flipResult );
else
  printf("Tails is represented by %d\n", flipResult );
```

在第 6 章我们将看到，类似该例子中的 if 条件，会在 Objective-C 对象的初始化过程中用到。

4. 单位矩阵是一个方形的数字数组，其中，对角线上（行号等于列号的元素）都是 1，而其他的任何位置都是 0。2×2 的单位矩阵如下所示：



编写一个程序来计算并存储 4×4 的单位矩阵。并且程序在完成矩阵计算后，它应该以格式整齐的方形数组输出结果。

5. 斐波那契数(http://en.wikipedia.org/wiki/Fibonacci_number)是一个数字序列，它在自然界和数学中都经常出现。前两个斐波那契数定义为 0 和 1。第 n 个斐波那契数是前面的两个斐波那契数的加和：

编写一个程序来计算和存储前 20 个斐波那契数。在计算这些数字之后，程序应该每行一个数字，并还带上索引地输出这些数。输出行应该如下所示:

```
1.  Fibonacci Number 2 is: 1
```

使用#define 来控制程序所产生的斐波那契数的编号，以便可以很容易地修改它。

6. 重新编写练习 5 中的程序，使用一个 while 循环来代替使用一个 for 循环。

7. 如果要求计算前 75 个斐波那契数呢？如果使用 int 来存储数字，将会有一个问题。你会发现第 47 个斐波那契数太大了，而无法用 int 来存储。如何解决这个问题呢？

8. 从 iPhone App Store 中可用的小费计算程序的数目来看，真的有很多人都已经忘记了如何做乘法运算。帮助那些不会做乘法运算而又买不起 iPhone 的人编写一个程序，在所有$10 到$50 之间的账单上计算一个 15%的小费（为了简单起见，以$0.50 为增量）。显示出账单和小费。

9. 现在，让这个小费计算程序看上去更专业一些。添加一栏用来计算 20%的小费（Objective-C 程序员都是在档次较高的地方吃饭）。给每一栏添加一个相应的标题，并且使用一对嵌套的循环，使程序按照每$10 的增量来输出一个空白行。

使用转换修饰符%.2f 而不是%f，这将限定账单和小费都输出两位小数。在格式字符串中使用%%，将会使 printf 输出一个单个的%字符串。

10. 定义一个结构，用来存储一个矩形。定义的这个结构，应包含一个点的坐标，还包含另外一个结构，后者通过存储宽度和高度来表示大小。矩形结构中的点表示矩形的左上角，另一个结构表示矩形的大小。（Cocoa 框架定义了类似这样的结构，但是，现在你需要自行定义一个）。

11. 高效的计算机图形的一个基本原则是，“不到万不得已，不要自行绘制”。图形程序通常为每个图形对象保留一个边界矩形。当需要在屏幕上绘制图形时，程序将图形的边界矩形与表示窗口的矩形进行比较。如果两个矩形之间没有重叠，那么程序将会跳过绘制图形的尝试。总的来说，这通常是好办法，因为比较矩形比绘制图形要简单多了。

编写一个函数，它接受两个矩形结构作为参数（使用练习 10 定义的结构）。如果两个矩形之间有一个非零的重叠的话，那么函数应该返回 1；否则的话，返回 0。编写一个测试程序，来创建一些矩形，并用它测试你的函数是否有效。

第 2 章 C 变量

2.1 Objective-C 程序的内存布局

第 2 章　　C 变量

当用大多数常见的脚本编程语言编写一个程序时，几乎不必花时间来考虑变量。只是在使用变量时才创建它们，并且不必担心用完它们之后会发生些什么。语言的解释器会负责所有的细节。

当你在编译语言中编写代码时，事情就没那么简单了。必须告诉编译器每个变量的类型和名称，以声明任何将要在程序中使用的变量。编译器随后查看变量的声明类型，为其保留相应数目的字节，并且将变量名与这些字节关联起来。

本章我们将介绍 Objective-C 中变量声明的形式，以及编译器如何存储各种不同的变量。

2.1　　Objective-C 程序的内存布局

要理解本章的内容，就要知道 Objective-C 程序是如何管理内存的。图 2-1 给出了运行程序的虚拟地址空间的一个简图。

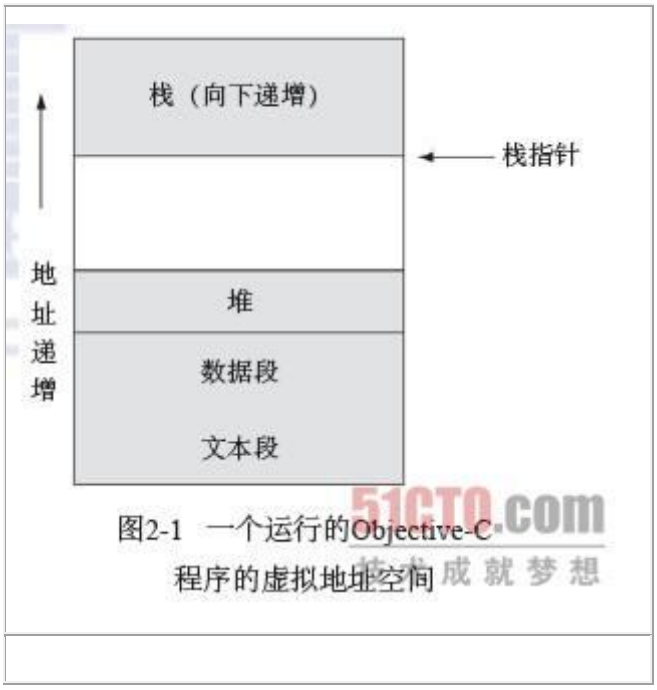


图2-1 一个运行的Objective-C程序的虚拟地址空间

注意　虚拟地址空间是程序“看到的”地址空间。虚拟地址空间和实际的物理地址空间之间的转换，由操作系统和计算机的内存管理单元（Memory Management Unit，MMU）隐式地进行。

按照虚拟地址从低向高的顺序：

文本段包含了程序的可执行代码和只读的数据。

数据段包含了可读写的数据，包括全局变量。

堆包含了根据请求分配给程序的内存块（参见本章稍后的 2.6 节）。当需要更多内存时，系统可能会向上扩展堆。

栈用于调用者函数。当调用一个函数时，系统为被调用的函数构建一个栈帧。栈帧是在栈的底部（最低的地址）构建的一个内存区域。栈指针（它指向栈的最低位置的地址）向下移动。栈帧包含了用于被调用函数的参数和局部变量的空间，以及用来保存在函数调用的过程中需要保存的任何寄存器的值的空间，还有用于一些控制信息的空间。当函数返回时，栈指针恢复到其最初的值（较高的地址），并且控制返回给调用者函数。关于这一过程，需要记住的重要的一点是，当函数返回之后，其栈帧的内容就不再有效了。

注意　一些人喜欢把栈看做是向上扩展的，类似于桌面上物理地堆放着的一摞纸。他们说栈的指针指向“栈的顶部”。尽管可以这么考虑，但是，栈在内存地址中是向下递增

的。调用一个函数会将栈指针向一个较低的地址移动，并且，从函数返回则会将栈指针移回到一个较高的地址。

2.2 自动变量

2.2 自动变量

在函数或子程序中声明的变量叫做自动变量或局部变量。考虑如下的简单函数：

```
1.  void logAverage( float a, float b)
2.  {
3.    float average;
4.    average = 0.5 * (a + b);
5.    printf( "The average is %f\n", average);
6.  }
```

在上面的例子中，average 是一个自动变量。

关于自动变量，有一些重要的事情需要记住：

自动变量是在栈上创建的。它们只是从声明它们的地方到函数的末尾有效（参见本章稍后的 2.5 节）。当函数返回时，系统将栈指针移动到前面的帧的底部。后续对其他的函数的调用会创建新的栈帧，并且，很可能会覆盖分配给前面的函数的自动变量的内存位置。如果使用参数 10.0 和 12.0 调用 logAverage()，当执行到 printf 语句时，变量 average 的值为 11.0。如果随后再次调用 logAverage()，在该程序顶部，average 的值不太可能是 11.0。

自动变量不是由系统初始化的。在再次给它们分配一个值之前，它们包含了上次使用它们时在内存位置中留下的随机垃圾信息。

自动变量与函数的单次调用有关。如果有函数调用自己的递归代码，则每次对函数的调用都有其自己的栈帧及其自动变量的副本。在一个调用中修改该变量的值，不会修改堆栈上的任何其他调用中该变量的值。

使用一个&运算符将一个自动变量的地址取出，并且分配给函数之外的一个指针变量来保存，这是很糟糕的做法，因为只要函数退出，指针变量就指向了垃圾信息。

注意　自动变量所谓自动的含义是，当调用使用了自动变量的函数时，它们的存储在栈上分配。然而，仍然必须用一个声明告诉编译器与自动变量相关的信息。

函数参数

函数参数实际上是在调用函数时，已经用提供的值初始化了的自动变量。它们在栈上创建，并且当函数返回时，它们变成无效的。正如第 1 章所介绍的，可以在函数体中重置它们的值，但修改它们的值对于调用者函数中相应的变量没有影响。

2.3 外部变量

2.3 外部变量

在 main 函数或任何子程序作用域之外的源文件中声明的变量，叫做外部变量。在如下的示例中，pageCount 就是一个外部变量：

```
1.  int pageCount;
2.
3.  main()
4.  {
5.  ...
6.    printf("The current page count is: %d\n", pageCount );
7.  }
8.
9.  void addpage()
10. {
11.   pageCount++;
12.   ...
13. }
14.
15. void deletePage
16. {
17.   pageCount--;
18.   ...
19. }
```

尽管这并不总是一种好的设计选择，外部变量有时候也用作全局变量，以便在不同的函数和不同的源文件之间共享信息。关于外部变量，有一些重要的事情需要记住：

编译器在虚拟地址空间的数据段中分配了外部变量的内存位置。外部变量在程序的生命周期中都持久存在，它们不会超出作用域或消失，并且，只有当给它们分配一个新值时，它们的值才会改变。

如果没有显式初始化它，那么编译器会将外部变量初始化为 0。

外部变量在单个函数的作用域之外是可见的（可用的）。实际上，外部变量是全局标记。除非将一个外部变量声明为 static 的（参见 2.4.3 小节），否则它可能对于任何源文件中的任何函数都是可见的。

2.4.1 auto

2.4 声明关键字

Objective-C 定义了几个关键字，它们可以用来修饰声明。

2.4.1 auto

auto 关键字只在一个函数内使用，用来告诉编译器，该变量是一个自动变量。由于这是默认的，所以很少使用它。一般不会看到它。

2.4.2 extern

2.4.2 extern

当想要引用一个在不同文件中声明的外部变量时，使用 extern 关键字。以 extern 开头的声明使得编译器知道变量的名称和类型，但是，该声明不会导致编译器为该变量保留任何的存储。

下面的语句告诉编译器，pageCount 是一个 int，并且，已经在程序中的其他某个地方使用一个声明为其保留了存储：

```
1.  extern int pageCount;
```

如果还没有在程序中的其他地方将 pageCount 声明为一个外部变量（没有使用 extern 关键字），而使用如下的一条语句：

```
1.  int pageCount;
```

则程序将会编译，但是，它不会连接。因为编译器通过 extern 声明得到满足，但是，连接器失败了，因为它查找不存在的 pageCount。

2.4.3 static

2.4.3 static

根据是用于函数中的变量还是用于外部变量，static 关键字具有不同的含义。当它用于函数中的变量时，static 关键字创建了类似于外部变量的一个变量：

```
1.  void countingFunction()
2.  {
3.    static int timesCalled;
4.    timesCalled++;
5.    ...
6.  }
```

在前面的例子中，timesCalled 保存了调用 countingFunction 函数次数的一个计数。

关于函数的 static 变量，需要记住如下的几个要点：

编译器在数据段中为一个函数 static 变量创建存储。

函数 static 变量初始化为 0，除非为其提供一个显式初始化值。

函数 static 变量的值在函数调用之间保持不变。

在对函数的多次调用中（即便是函数递归调用其自身），对一个函数 static 变量的引用，都是指向同一个内存位置。

函数 static 变量与外部变量的唯一区别在于,它只是在声明它的函数的作用域内可见。

当 static 关键字用于外部变量时，它将该变量的可见性限定在了声明它的文件之中，并且对其他的源文件隐藏。如果在一个函数体外部编写了：

```
1.  static int pageCount;
```

那么 pageCount 只能由同一个源文件中的函数引用。如果试图使用一条 extern 语句来声明 pageCount，从而要在一个不同的源文件中使用它：

```
1.  extern int pageCount;
```

则连接器将会失败。

2.4.4 register

2.4.4 register

register 关键字提示编译器，它所修饰的变量会频繁地被引用：

```
register int count;
```

结果，编译器可能选择将该变量存储在一个寄存器中（以便于快速访问），而不是存储在 RAM 中。然而，编译器并不一定遵从这一提示。

2.4.5 const

2.4.5 const

const 关键字告诉编译器应该将一个变量视做常量。对于任何试图修改该变量的行为，编译器会提示一个错误：

```
const int maxAttempts = 100;
maxAttempts = 200;  // This will cause a compiler error.
```

当 const 用于指针变量时，顺序很重要。声明应该从左向右读取。如下的示例声明了 ptr 是指向一个整数的常量指针：

```
int a = 10;
int b = 15;

int *const ptr = &a;

*ptr = 20;  // OK. a is now 20.

ptr = &b; // Error, ptr is a constant.
```

不能修改 ptr 的值以指向一个不同的变量，但是，可以使用 ptr 来修改它所指向的变量。

将 const 放在声明的其他内容的前面，则有着不同的含义。ptr 现在是指向一个 int 常量的指针：

```
int a = 10;
int b = 15;

const int *ptr = &a;

*ptr = 20; // Error, ptr's contents are constant.
a = 20;    // OK, a itself has not been declared as const.

ptr = &b;  // OK, ptr now points at b.
```

可以修改 ptr 以指向另一个变量，但是，不能通过 ptr 来修改它所指向的任何变量。即便在上面的例子中，变量自身并没有被声明为 const，也不能通过 ptr 来修改它（但是，在上面的例子中，由于 a 没有被声明为 const，因此如果直接访问它的话是可以修改它的）。

2.4.6 volatile

2.4.6 volatile

volatile 关键字声明了一个变量的内容可能会被程序的 main 线程以外的其他参数修改。然后，编译器将避免像没有该关键字时那样优化。例如，如果变量的存储实际上是属于外部设备的一个硬件寄存器，并且该设备已经被内存映射到了程序的地址空间中，就会发生这种情况。

考虑这样一种情况，其中 shouldContinue 被映射到一个硬件设备上的一条控制线：

```
1.  int shouldContinue = 1;  // initialization
2.
3.  // Check the control line before each pass through the loop.
4.
5.  while (shouldContinue ) doStuff;
```

如果 doStuff 所表示的代码没有修改 shouldContinue，优化的编译器将会得出 shouldContinue 没有被修改的结论，并且，它将代码优化为如下的等价形式：

```
1.  while (1) doStuff;
```

遗憾的是，这是一个无限循环。将 shouldContinue 声明为 volatile 则阻止了编译器进行这一优化：

```
1.  volatile int shouldContinue;
```

2.5.1 自动变量的作用域

2.5 作用域

变量的作用域就是可以“看见”该变量语句的范围。看见，在某种意义上，意味着一条语句可以引用该变量，而编译器不会给出一条错误信息来提示说它不知道关于该变量的任何信息。

2.5.1 自动变量的作用域

自动变量可以在函数体中的任何位置声明。自动变量的作用域为从声明它的位置直到函数的末尾。在声明该变量之前，不能使用它：

```
1.  void someFunction()
2.  {
3.    int a;
4.    a = 7; // OK, a is in scope
5.    ...
6.    b = 7; // WRONG, b is not in scope at this point
7.    int b;
8.    ...
9.  }
```

当声明了自动变量的函数返回之后，自动变量的作用域就消失了。在该函数之外，它是不可见的。

2.5.2 复合语句和作用域

2.5.2 复合语句和作用域

一条复合语句的内部是一个独立的作用域（一条复合语句，是包含在花括号中的一个语句序列）。可以在复合语句中的任意位置声明一个变量，该变量从声明它的地方到复合语句结束的地方，都是可见的。复合语句存在于其中的作用域，叫做该复合语句的外围作用域。在外围作用域中声明的变量，在复合语句中是可见的；但是，在复合语句中声明的变量，在外围作用域中是不可见的：

```
1.  void someFunction()
2.  {
3.    int a = 7;
4.      {
5.        int b = 2;
6.        int c;
7.
8.        c = a * b;  // OK, a is visible
    inside the compound statement
```

```
    }

  int d = 2 * c;  // WRONG,  c is not visible at this point
}
```

编译包含上述代码段的程序，将会产生一个编译器错误：

```
someFunction.c:12: error: 'c' undeclared
(first use in this function)
someFunction.c:12: error: (Each undeclared
identifier is reported only
someFunction.c:12: error: once for each
function it appears in.)
```

如果在一条复合语句中声明了一个变量，而该变量与外围作用域中的一个变量具有相同的名称，那么，在内部变量的作用域中，内部的变量将会覆盖（隐藏）外围作用域中的变量：

```
void someFunction()
{
  int a = 7;
  int b = 2;
    {
      int c;
      c = a * b;  // c is 14, enclosing
scope a is still visible;

      int a = 10; // This hides the
enclosing scope's variable a
      c =  a * b; // c is now 20
    }
}
```

2.5.3 外部变量的作用域

2.5.3 外部变量的作用域

外部变量（在函数之外声明的变量）从文件中声明它的地方到文件的末尾可见。正如本章前面介绍的，可以通过使用一个 extern 关键字，将一个外部变量的可见性扩展到不同的文件。而为遵循减少可见性的指导方针，以一个 static 关键字开始一个外部变量的声明，

会将该变量的作用域限定在声明它的文件之中。static 关键字优先于对同一变量的任何 extern 声明。

2.6 动态分配

2.6 动态分配

本章到目前为止介绍的变量声明，都是静态分配的。在编译时，必须告诉编译器所需的变量的数目和类型。

Static 和 Static

单词 static 在这里重载了。不要将通用术语“静态分配（static allocation）”和前面小节所讨论的 static 关键字的含义搞混淆了。是的，这可能会引起混淆。

在很多情况下，在编译时指定所需的内存量是一个问题。假设想要读入一个灰度（每像素 8 位）图像的像素。你将需要一个字节数组来存储它们：

```
1.  #define MAX_WIDTH 1000
2.  #define MAX_HEIGHT 1000
3.  unsigned char pixels[MAX_WIDTH * MAX_HEIGHT];
```

但是，MAX_HEIGHT 和 MAX_WIDTH 实际上应该是多少呢？如果把它们指定得太小了，将会无法读取图像。可以将它们设置得很大，但这也会有问题：对于一般情况来说，你浪费了很多内存，并且程序的“内存占用”没必要很大，否则这会影响到性能。这也很容易导致墨菲定律的出现，只要你直接编码大小，某人、某地总会产生一个比直接编码的大小更大的图像文件，并且期望你的程序来读取它。

这个问题的解决方法是使用动态分配。动态分配允许在运行时，也就是当程序在执行的时候，再请求更多的内存。当需要更多的存储时，向系统请求一块内存。当使用完内存时，要释放它。用于动态分配的字节，位于图 2-1 所示的名为“堆”的区域中。

注意　　堆是描述这一区域的常用 Unix 术语。Mac OS X 实际上有一个更加复杂的系统。在 OS X 中，这一区域在技术上叫做一个 malloc 区域（参见下面“malloc 区域”中的内容）。在大多数情况下，本书将继续使用术语“堆”。

动态分配由两个函数处理：malloc 和 free。使用一个参数来调用 malloc 以请求内存，该参数指定了所需的字节数。malloc 返回指向请求的数目的字节的一个指针，随后可以将

这个指针强制转换为想要的数据类型，并且赋给一个变量，而这个变量的类型就是指向所请求的类型的一个指针。

注意　　曾经，当没有可用的内存可分配时，malloc 返回 NULL。然而，OS X 和 iOS 都使用延迟分配技术。使用延迟分配，malloc 返回指向请求的内存的一个指针，但是，在执行一段代码来访问该内存之前，系统不会为该请求分配内存资源。结果，如果你多次请求内存，并且没有使用该内存或没有释放该内存，那么 malloc 可能会返回一个非 NULL 的值，即便没有更多内存可用也是如此。在这种情况下，使用返回的指针可能会导致程序崩溃。假设给 RAM 和交换空间指定一些典型值，即便有可能，你也是极少会看到这种情况在 OS X 系统上发生。在 iOS 中，遇到这种情况之前，应用程序很可能会从系统接收到一条低内存警告。

要计算需要的字节数，可以使用 sizeof 函数来确定每一项所需的字节数，然后，用该结果乘以所需的项数。例如，一个 8 位灰度图像所需的字节数是（假设已经从文件头部的某些信息中读取了图像的高度和宽度）：

```
numBytesNeeded = imageHeight * imageWidth
* sizeof( unsigned char );
```

内存分配编写如下：

```
numBytesNeeded = imageHeight * imageWidth *
sizeof( unsigned char );
unsigned char* pixels =
    (unsigned char*) malloc( numBytesNeeded );
```

使用完 malloc 分配的内存后，调用 free 函数释放这些内存：

```
free( pixels );
pixels = NULL;
```

通过对 free 的调用，从而使程序可以重用该内存。当这些字节释放以后，可以通过后续对 malloc 的调用，将它们拿来用于不同的用途。动态分配的黄金法则是：如果你分配了某些内存，当你用完之后，要负责把它们收回来。否则，你的应用程序将会“泄漏”内存。因此你的进程的大小将会比所需的还要大，并且，你的程序及同时运行的其他程序可能会为此遭殃。

注意　　每个 free 调用必须匹配一个相应的 malloc。释放没有分配内存的指针，或者释放相同的指针两次，都会导致程序立即崩溃。

注意　　使用已经释放的一个指向字节的指针，可能会破坏程序的内存，并且导致随机的崩溃，而且这很难调试。将 pixels 设置为 NULL 提供了一种安全的方法。如果你在代码中犯了错误，并且试图使用 NULL 指针，那么你的程序将会立即崩溃。这样的问题及其位置将会很明显。

malloc 区域

OS X 有一个或多个 malloc 区域，而不是一个统一的堆。第一个 malloc 区域，叫做默认 malloc 区域，它在第一次调用 malloc 时创建。标准的 malloc 函数从默认的 malloc 区域获取其字节。也可以使用 malloc_create_zone 函数来创建额外的 malloc 区域。

使用其他的 malloc 区域是一项高级的技术，在特定的情况下，它可以用于优化性能。它允许成组地频繁访问内存位置相邻的变量，或者一次释放多个变量（通常释放一个完整的区域）。

尽管没有正式废弃，Apple 现在不鼓励在 Objective-C 中使用除默认 malloc 区域以外的其他区域（参见 http://developer.apple.com/mac/library/DOCUMENTATION/ Performance/ Conceptual/ ManagingMemory/Articles/MemoryAlloc.html）。

2.7 小结

2.7　　小结

本章介绍了如何声明 Objective-C 变量。要点如下：

必须声明要在 Objective-C 程序中使用的任何变量。声明告诉编译器，该变量的类型及其他信息。作为对声明的响应，编译器为变量安排存储（内存中的某些字节），并且将变量名与存储关联起来。

编译器可以将变量放置在 3 个地方：放在栈上，作为数据段的一部分，以及放在堆中。

在函数中声明的变量是自动变量。当调用函数时，自动变量在栈上创建。编译器不会初始化自动变量，并且，只有在声明它们的函数作用域之内，它们才是有效的。

在任何函数作用域之外声明的变量叫做外部变量。外部变量在编译时创建于数据段中，并且初始化为 0（除非提供显式的初始化）。在程序的生命周期内，它们都有效。

Objective-C 提供了如下的关键字来修饰变量声明：auto、static、extern、register、const 和 volatile。

在程序运行时，可以使用 malloc 函数来请求额外的内存。用 malloc 获取的内存来自于堆。

使用完用 malloc 获取的内存后，必须使用函数 free 将该内存释放回堆。

本章完成了对 Objective-C 的非对象部分的快速概览。在第 3 章中，本书将继续介绍面向对象编程及 Objective-C 对象的基础知识。

2.8 练习

2.8 练习

1. 编写一个小程序并声明一个外部变量，该程序要调用一个函数，该函数声明并使用一个自动变量（使用哪个函数来做这个练习并不重要。你可以只是声明并初始化一个本地变量，然后，使用一条 printf 语句来输出两个变量的值）。构建该程序，然后在 gdb 中运行它。在函数中设置一个断点。当达到断点时，查看自动变量和外部变量的地址。可以通过在 gdb 命令提示行中输入如下内容来显示一个变量的地址：

```
1.  p &variableName
```

结果与你在本章中学习的内容一致吗？

2. 修改练习 1 中的程序，添加第二个函数，它也声明一个自动变量。修改最初的函数，以使它调用新函数。验证栈确实是向着较低的地址扩展：在新函数内部设置一个断点并在此停止。显示新的函数的自动变量的地址。然后，在 gdb 的命令提示行中输入：

```
1.  up 1
```

这将会把 gdb 的注意力转移到前一个栈帧，从而可以显示调用者函数的自动变量的地址。

3．编写一个函数，显示调用它的次数。将次数存储在一个函数作用域的静态变量中。调用该函数数次以验证计数是正确的。有什么方法让计数从一个非零的值开始吗？

4．本练习使用两个源文件。在第一个文件中，声明一个外部变量并且将其初始化为某个值。在第二个源文件中，放置一个函数来记录外部变量的值（别忘了，在第二个文件中进行 extern 声明）。构建并运行程序，以测试其有效性。如果忽略了 extern 声明，将会发生什么？恢复第二个源文件中的 extern 声明，并且在第一个文件中给变量声明添加关键字 static。看看会发生什么事情？

5．编写一个程序来计算并存储前 10 个整数的平方。当它确实计算了所有的 10 个数时，程序应该使用 printf 来显示结果。使用 malloc 来获取保存计算的值的内存，而不是声明一个保存 10 个整数的数组。不要忘了，在完成之后释放内存。

第 3 章

3.1 面向对象编程

第 3 章　　面向对象编程简介

早期的高级计算机语言，例如 Fortran、COBOL 和 C 都是面向过程的。以面向过程的语言编写的程序的实质结构，是要按照顺序执行的一系列任务。过程式编程特别适合于某些类型的问题，例如，求解数学方程式。然而，有很多类型的问题，例如，用户界面编程，却是过程式编程所不适合的。在如今计算的很多领域中，占有主导地位的编程泛型是另一种叫做面向对象的编程泛型。Objective-C 是面向对象语言的一种。它添加了结构和语法来支持使用过程式的 C 语言进行面向对象编程。

本章从一个简介开始，介绍面向对象编程的基础知识。然后说明这些概念在 Objective-C 中是如何实现的。本章 3.3 节列出了 Objective-C 对于 C 语言添加的内容。

3.1　　面向对象编程

面向对象编程是编程的一种形式，它将程序组织为彼此交互的对象的集合。对象是一组相关的变量及一些过程，这些变量对问题空间中的某些内容建模；而这些过程叫做方法，它们表示对象知道如何去执行的动作。对象通过发送和接收消息来通信。发给对象的一条消息，就是对对象的一个请求，要求它执行其方法之一。

例如，一个绘图程序可能有一个 Shape 对象，它表示用户可以在屏幕上绘制的任意形状（见图 3-1）。一个 Shape 对象可能有多个变量，用于存储确定其边框的点、颜色及其在绘图中的位置。要在屏幕上绘制这个 Shape，需要给 Shape 发送一条 draw 消息。作为响应，Shape 将会执行其 draw 方法（该方法包含实际绘制的代码）。

3.1.1 类和实例

3.1.1　　类和实例

对象是根据其类来分类的。每个对象都属于某个类，它是该类的一个实例。用现实世界的事物来类比的话，一辆保时捷跑车的设计和规范中所包含的信息是一个类。这些信息定义了一辆保时捷跑车是什么，以及如何构造一辆保时捷跑车。你在赛道中实际驾驶的跑车是

Porsche 类的一个实例。回到计算机程序的世界，类是定义了一组变量和一组方法的一个模板或菜谱，这一组变量叫做实例变量，这一组方法则包含了实现这些方法的代码。对象（类的一个实例）是实际的一块内存，用来为类中定义的一组变量提供存储。一个给定的类可能有多个实例。每个实例是内存的一个单独区域，并且拥有类中定义的实例变量的一个副本。

3.1.2 方法

3.1.2 方法

方法类似于函数，但是，它们并不完全相同。方法代表定义它们的类的一个实例而执行。当一个对象执行一个方法时，该方法可以访问对象的数据。如果向一个 Shape 对象发送一条 draw 信息，draw 方法将使用 Shape 对象的边框、位置和颜色信息。如果向不同的 Shape 对象发送同样的 draw 消息，draw 方法将使用相应的 Shape 的边框、位置和颜色信息。

3.1.3 封装

3.1.3 封装

封装，有时候也叫做信息隐藏，指的是向类的用户隐藏一个类的内部工作。它是通过减少程序的不同部分之间的联系，从而减少复杂性的一种方法。对象只能由一个定义的接口来进行封装，也就是其类所实现的一组方法。

使用 Shape 类编写代码的程序员，不需要知道 Shape 的数据在内部是如何存储的或者绘制代码是如何工作的。他只需要知道，当一个 Shape 实例接收到一条 draw 消息时，它将绘制自身。

封装给了类的开发者修改其实现的自由，而不会影响到使用该类的代码。Shape 类的开发者可能会改变 Shape 用来存储描述其边框的点的坐标系统；或者在执行绘制代码之前，通过检查以验证 Shape 位于窗口边界之中，从而添加一些改进来提高 draw 方法的性能。只要新的坐标系统和新的改进的 draw 方法在所有情况下都产生与旧版本同样的结果，其他的代码就都不需要修改。使用 Shape 类的代码，在程序下一次编译和连接时，自动从改进的 draw 方法获益。

3.1.4 继承

3.1.4 继承

继承提供了一种方法，它通过扩展和修改一个已有类的行为来创建新的类。假设你想要给绘制程序添加一个 AnnotatedShape 类，用来绘制一个形状及形状下面的文本注释，如图 3-2 所示。

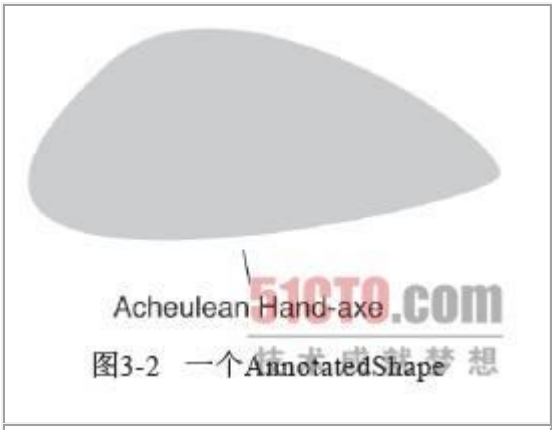


图3-2 一个AnnotatedShape

可以从头开始编写 AnnotatedShape 类，但是，这可能会有些浪费时间。AnnotatedShape 类中的大多数代码都是重复 Shape 类中的代码。如果你可以指出 AnnotatedShape 与 Shape 存在的以下区别，就太好了：

AnnotatedShape 多了一个变量，这是一个字符串变量，用来保存注释的文本。

AnnotatedShape 有两个额外的方法，它们设置和获取注释的文本。

AnnotatedShape 的 draw 方法与 Shape 的 draw 方法的实现不同。它必须同时绘制注释和形状。

如果如下这样定义 AnnotatedShape，那么，以面向对象的说法：

AnnotatedShape 继承自 Shape。

AnnotatedShape 的 draw 方法的实现覆盖了 Shape 的 draw 实现。

AnnotatedShape 是 Shape 的一个子类。

Shape 是 AnnotatedShape 的一个超类。

3.1.5 多态

3.1.5 多态

多态是不同类的对象响应同一消息的能力。例如，可以给绘制程序添加一个 Image 类，使它也可以绘制图像（位图）。就像 Shape 类一样，Image 类定义了一个 draw 方法。每个

类的 draw 方法的实现是不同的:Shape 的 draw 方法知道如何绘制一个 Shape,Image 的 draw 方法知道如何绘制图像。如果有一个要绘制的图形对象的列表，可以通过给列表中的每个对象发送一条 draw 消息，从而将它们绘制到屏幕上。即便每种情况下的消息都是相同的，也是列表中的 Shape 对象将执行 draw 的 Shape 版本,而 Image 对象将执行 draw 的 Image 版本。

3.1.6 面向对象语言的主要特点是什么

3.1.6 面向对象语言的主要特点是什么

使用一种过程式语言来实现面向对象的编程风格，这是可能的，但是，会很困难，并且有些不令人愉快。以 C 为例，可以把结构用作对象，并且将常规的 C 函数用作方法。要给对象添加方法，可以给对象的结构添加字段，来保存指向方法函数的指针。也可以给每个方法函数定义一个参数，该参数允许传入一个指针，该指针指向在其上执行该方法函数的对象。但是，你将很快遇到问题。在单个数组中存储对象的不同类并遍历它们，即便像这样简单的事情，也需要艰难地使用强制类型转换运算符。

还有其他的一些缺点。如果你没有想清楚自己设计的所有用意，构建自己的对象系统可能导致不稳定的代码，并且可能出现不希望的后果。一个更为严重的问题是，自建的对象系统很可能是一次性的设计。一次性的定制设计，使得很难将来自其他源的代码或库整合到你的程序中。你将必须从头构建一切，这要花费较长的开发时间，并且产生 bug 的机会要增加很多。

对于已经内建了面向对象编程系统的语言来说，使用其进行一个面向对象的设计，要容易很多。

3.2 Objective-C 简介

3.2 Objective-C 简介

Objective-C 是 C 的面向对象的扩展。正如你在本节及后面将看到的，Objective-C 对 C 的添加很少。主要添加是，定义类的方式、调用方法的方式，以及十几个关键字和针对编译器的指令。这些概念很精巧而且很强大，但是，需要记住的语法相对很少。

除了给 C 添加了一个对象系统，Objective-C 没有剔除掉 C 的过程式部分。这样你可以吸取两个世界的精华。可以使用对象来完成那些适合它们的任务，而使用 Objective-C（普通 C）的过程式部分来解决它们最为适用的问题。用普通 C 作为该语言的基础，也使得 Objective-C 程序可以很容易地使用那些用普通 C 编写的库和代码。

本节以概览的方式介绍了 Objective-C 如何实现面向对象编程的概念。Objective-C 的核心部分将在第 5 章、第 6 章和第 7 章中介绍。

3.2.1 定义类

3.2.1　　定义类

Objective-C 类定义有两个部分：接口部分和实现部分。接口部分声明了类的实例变量及其方法。实现部分则包含了实现类的方法的代码。接口部分通常放在一个头文件中，按照惯例，头文件按照类来命名。实现部分放在一个以.m 为扩展名的文件中，也以类来命名。.m 文件告诉编译器，该文件包含了 Objective-C 的源代码。

接口部分

类定义的接口部分如下所示：

```
1.  @interface className : superclassName
2.  {
3.     Instance variable declarations
4.  }
5.
6.  Method declarations
7.
8.  @end
```

@interface 和@end 用来表示接口部分的开始和结束。Objective-C 中以@字符开始的单词，是编译器指令，用来指示编译器，而不是可执行的代码。

@interface 后面跟着类名、一个冒号及类的超类的名字。

类的接口变量的声明，位于一对花括号之间，放在@interface 行的后面。

在实例变量声明之后，声明类的方法。

注意　　方法声明必须位于实例变量声明之后。如果将方法声明放置在花括号之前，那么这些代码将不会编译。

为了让这些内容更加具体，考虑一个类 Accumulator，它保存加和的总数。Accumulator 类有一个示例变量来保存总和，还有向总和添加，打印出总和将总和及重置为 0 的方法。这是一个很小的例子，但是，它让你对语法的了解更具体。如下是 Accumulator 的接口部分：

```
1.  1 @interface Accumulator : NSObject
```

```
2.  2 {
3.  3   int total;
4.  4 }
5.  5
6.  6 - (void) addToTotal:(int) amount;
7.  7 - (int) total;
8.  8 - (void) zeroTotal;
9.  9
10. 0 @end
```

按照命名惯例，将这段代码放入到一个名为 Accumulator.h 的文件中。

第 1 行：这个类的名称为 Accumulator。按照惯例，Objective-C 类名以一个大写字母开头。Accumulator 的超类是 NSObject。

第 3 行：Accumulator 类有一个实例变量 total。该变量在类的接口部分的声明，没有保留任何存储。当创建 Accumulator 的一个实例的时候，新的实例会得到它自己的 total 的副本，但是，没有与该类自身相关的存储。

第 6～8 行：这些是方法声明。方法声明的形式如下：

```
1.  -(return type) method_name:(argument_type) argument;
```

打头的连字符(-)表示这些是实例方法，也就是由类的一个实例来执行的方法。也有可能声明类方法，即由类自身执行的方法。类方法以一个加号打头(+)：

```
1.  + (return type) class_method_name:(argument_type) argument;
```

类方法将在本书第 7 章介绍。

按照惯例，Objective-C 方法名以一个小写字母开头。方法名的剩下部分则遵循 CamelCased 命名法，即名称中其余的每个单词，都以一个大写字母开头，并且，不会用一个下划线字符(_)将单词分隔开。

实现部分

实现部分包含了类的方法的实现。Accumulator 的实现部分如下所示：

```
1.  1 #import "Accumulator.h"
2.  2
3.  3 @implementation Accumulator
4.  4
5.  5 - (void) addToTotal:(int) amount
```

```
6.  6 {
7.  7   totaltotal = total + amount;
8.  8 }
9.  9
10. 10 - (int) total
11. 11 {
12. 12   return total;
13. 13 }
14. 14
15. 15 - (void) zeroTotal
16. 16 {
17. 17   total = 0;
18. 18 }
19. 19
20. 20 @end
```

将这些代码放入到一个名为 Accumulator.m 的文件中：

第 1 行：编译器在编译相应的实现部分时，需要类的接口部分的信息，因此 Accumulator.m 导入了 Accumulator.h 文件。

第 3 行和第 20 行：方法的实现放在一个@implementation 指令（后面跟着类名）和一个@end 指令之间。

第 5～8 行：一个方法实现包括一个重复的方法声明（后面没有结束的分号），其后跟着实现该方法的代码，这些代码放在一对花括号中。方法实现的主体，其编写的方式与编写一个普通 C 函数的方式相同。

第 7 行：注意，一个方法实现能够直接访问其类的实例变量，即便它没有声明这些变量。

继承

所有的 Objective-C 类，除了根类以外，都继承自某个其他的类。在某些特殊的条件下，例如分布式对象系统，需要关注的唯一根类就是 NSObject。几乎所有的 Objective-C 类，都直接或间接地是 NSObject 的子类。NSObject 定义了类工厂方法 alloc，它负责为那些需要与 Objective-C 的内存管理系统交互的对象实例和实例方法分配内存。其他的类，例如 Accumulator，通过继承 NSObject 而获取这些方法。本书第 6 章将详细介绍继承。

3.2.2 类名作为类型

3.2.2 类名作为类型

Objective-C 对象创建于堆上。这意味着，当创建一个对象实例时（在本章稍后，将会看到这一点），我们得到的是一个指向堆上的某些字节的指针。当人们说：“这个变量保存了一个 Accumulator”或者“这个变量保存了一个 Accumulator 对象”，他们实际的意思是“这个变量保存了一个指针，它指向 Accumulator 类的一个实例”。

使用类名作为类型，来声明保存指向对象实例的指针的变量。下面这行代码声明了变量 anAccumulator，它保存了指向 Accumulator 类的一个实例的一个指针：

```
1.  Accumulator *anAccumulator;
```

显式的星号(*)在表示 anAccumulator 是一个指针时是必需的。如果忽略了星号，例如：

```
1.  Accumulator anAccumulator;  //WRONG
```

将会导致一个编译器错误（编译器认为你试图在栈上创建该对象，而这在 Objective-C 中是不允许的）。

3.2.3 消息（调用方法）

3.2.3 消息（调用方法）

如果不能够让对象为你做某些事情，那么即便拥有对象，也没有太大的用处。我们需要使用某种方式来让对象执行其方法。在某些面向对象的语言中，例如 Java 和 C++中，方法调用就是与特定对象相关联的函数调用，它们通过在对象变量的后面用一个点号，然后再附加函数来完成调用。如果用 C++编写 Accumulator，应该编写如下的代码来给 anAccumulator 的总和添加一个数字：

```
1.  anAccumulator.addToTotal( 137 );
```

Objective-C 使用一种不同的方法，叫做消息，它借用自计算机语言 Smalltalk (http://en.wikipedia.org/wiki/Smalltalk)。使用消息机制，我们要给对象发送一条消息，而不是调用一个函数。如下的代码行显示了一条消息的表达形式：

```
1.  [receiver message]
```

receiver 是想要其执行一个方法的对象。之所以称其为接收者（receiver），是因为它接收该消息。message 就是想要执行的方法的名称，加上给该方法的任何参数。当计算

message 表达式时，receiver 执行与 message 中的名称对应的方法（第 5 章将详细介绍这是如何实现的）。

以 Accumulator 类的一个实例为例，如下的代码把 137 加到了 anAccumulator 的总和中：

```
[anAccumulator addToTotal: 137];
```

如下的代码获取了 anAccumulator 的总和的当前值，并且将其存储到名为 currentTotal 的变量中：

```
int currentTotal = [anAccumulator total];
```

首先，消息似乎只是编写函数调用的另一种方法。然而，它不是。差别很细微，但是，功能差异很大。这里接收者和消息在编译时没有绑定到一起。消息只是包含一个方法名。它不会将该方法名与任何特定的类或方法实现联系起来。不同的类可能提供同一方法名的不同实现。当计算消息表达式时，接收者对象执行在其类中定义的该方法的版本。

消息还考虑到了动态技术，而这一技术对于一种语言是不可能实现的，例如 C++，它要求在编译时将方法调用绑定（确定）。例如，一个 Objective-C 消息表达式的消息部分是一个变量，这是有可能的。在这样的一个消息表达式中，实际发送的消息是在执行时确定的，而不是在编译时。也可以通过编程来构建消息表达式，并且将其保存为对象以便稍后执行（参见第 16 章相关内容）。

多态

正如本章前面所述，向作为不同的类的成员的对象发送同一条消息，并且让它们以特定于类的方式执行该消息，这一功能叫做多态。

作为示例，考虑一个简单的绘图程序。该程序定义了 Shape 类和 Image 类，以表示程序可以绘制的不同的项。两个类都实现了一个名为 draw 的方法，但是，每个类的 draw 方法的实现是不同的。要在屏幕上绘制一个图形，程序遍历对象的一个列表，并且按照顺序绘制每个对象：

```
while( /* check for loop ending condition goes here */ )
  {
    id graphic = [graphicEnumerator nextObject];
    [graphic draw];
  }
```

graphicEnumerator 是负责将集合中的对象一个一个地传递出来的类的一个实例(第 10 章将介绍枚举器)。

id 是一个 Objective-C 类型，它被定义为一个通用的“指向对象的指针”。当一个变量将要在不同的时刻保存不同的类的对象时；或者在编译时，我们不知道在运行时将要在该变量中存储何种类型的对象；就使用该类型。可以将任何类型的对象赋值给一个 id 变量，而不会得到编译器的警告。

在这个例子中，graphic 可能在循环的一次迭代中包含一个 Shape 对象，而在另一次迭代中包含一个 Image 对象。当 graphic 包含一个 Shape 的时候，draw 消息引发执行 Shape 的 draw 实现。当它包含一个 Image 的时候，draw 消息引发执行 Image 的 draw 实现。

尽管 id 使得编写多态代码很方便，多态的真正源头还是 Objective-C 消息系统。相反，我们应该将 graphic 声明为指向一个完全无关的类的一个指针，这个类没有实现 draw 方法：

```
1.  while( /* check for loop ending condition goes here */ )
2.    {
3.      // Bad practice, but might work at run time
4.      Shmoo *graphic = [graphicEnumerator nextObject];
5.      [graphic draw];
6.    }
```

如果你这么做，编译器将会在一条警告中指出，Shmoo 类没有实现 draw 方法，你的同事会认为你疯了，而你的老板可能炒你的鱿鱼；但是，只要在执行时 graphic 包含一个 Shape、一个 Image，或者任何其他实现了 draw 方法的类的一个实例，这段代码就会工作。总之，重要的是运行时的对象类型，而不是变量类型。

注意　　将一个对象赋值给一个变量，而该变量声明为指向一个不同的类的对象的指针，这么做并不会改变最初的对象的类。对象的类是在创建对象的时候设置的，并且不会改变。

3.2.4 类对象和对象创建

3.2.4　　类对象和对象创建

Objective-C 中的每个类都由一个类对象来表示，该类对象可以代表类执行方法。一个 Objective-C 类对象实际上是一块内存，其中包含了它所代表的类的实例变量和方法的相关信息。类对象是一个名为 Class 的特殊类的实例。

可以在消息表达式中使用类名作为接收者，从而给一个类对象发送消息：

```
1.  [classname classmessage]
```

当给一个类对象发送消息时，消息中的方法名必须是一个类方法的名称。

类对象的主要用法是作为一个工厂，来创建类的实例。使用类方法 alloc 来创建对象实例，所有的类都从 NSObject 继承了 alloc 方法：

```
1.  Accumulator *anAccumulator = [Accumulator alloc];
```

alloc 为对象分配了内存；填充了名为 isa 的编译器所创建的一个特殊的实例变量，它是指向对象的类对象的一个指针；清空所有其他的实例变量，并且返回指向该对象的一个指针。

警告　　前面的代码行是一个说明，可以用一条初始化消息来嵌套 alloc 消息，如下一个例子所示。

所有的Objective-C对象都需要初始化。创建Accumulator的一个初始化实例的语法是：

```
1.  Accumulator *anAccumulator = [[Accumulator alloc] init];
```

执行前面的代码行：

1）向 Accumulator 类的类对象发送一条 alloc 消息。

2）类方法 alloc 为 Accumulator 对象分配内存，并且返回了指向未初始化的对象的一个指针。

3）未初始化的 Accumulator 对象变成了一条 init 消息的接收者。init 是 Accumulator 从 NSObject 继承的一个实例方法。

4）init 返回指向已经初始化后的 Accumulator 对象的一个指针，然后，将其赋值给变量 anAccumulator。

要了解关于类对象和类方法的更多内容，参见本书第 7 章。

注意　　你不必关心如何创建类对象。编译器会为你创建类对象。

3.2.5 内存管理

3.2.5　　内存管理

当在堆上分配内存时，当内存不再需要时，需要释放该内存。如果使用 malloc 分配了一块内存，当使用完以后，必须使用 free 释放它。Objective-C 使用一种略微不同的方法来管理对象内存，即所谓的引用计数（也叫做保留计数或管理内存）。每个 Objective-C 对象都有一个引用计数，它记录了使用该对象的地方的数目。当一个对象的引用计数变为 0 时，对象就销毁了，其字节也返还给堆。

方法 retain 和 release 会自增或自减一个对象的引用计数：

```
1.  [anObject retain];  //Increments anObject's retain count
2.  [anObject release];  //Decrements anObject's retain count
```

对象通过一个值为 1 的引用计数来创建。如果使用一个名称以 alloc 或 new 开头（或名称包含了 copy）的方法创建了一个对象，你就“拥有”了该对象。当使用完该对象后，必须向其发送一条 release 消息，以结束其创建并放弃你的所有权。如果通过名称不是以 alloc 或 new 开头（或不包含单词 copy）的方法来接收一个对象，则你并不拥有该对象。在大多数情况下，对象将在获取它的作用域中保持有效。但是，如果想要更长久地使用对象，例如，将其存储到一个实例变量中，那么，必须向其发送一条 retain 消息来“获得对象的所有权”。如果对象的创建者释放了对象，那么这么做会防止对象的引用计数变为 0。最终，当使用完对象后，必须给对象发送一条 release 消息来对应 retain 消息。

要更系统地了解本节所介绍的内存管理，参见本书第 14 章。要了解垃圾收集的内存管理替代方法，参见第 15 章。

3.3.1 运行时

3.3 Objective-C 添加

本节是 Objective-C 对 C 所做出的添加的一个简单概览。

3.3.1 运行时

Objective-C 需要运行时。运行时是 C 函数的一个动态链接库，在支持 Objective-C 的所有系统上都提供了运行时库。它负责建立和运行 Objective-C 的消息系统。运行时在幕后默默地坚守岗位。在一般情况下，不需要调用运行时的函数与其直接交互。

3.3.2 名称

3.3.2 名称

和普通 C 一样，Objective-C 的对象中的名称也是区分大小写的。Objective-C 有多个命名惯例，例如本章前面介绍的针对类名和方法名的命名惯例。这些惯例并不是语言强制要求的，但是，它们总是被广泛遵从的。如果你违反了它们，熟悉 Objective-C 的人将很难阅读你的代码。

其他的一些命名规则如下：

类可以声明一个实例变量，它可以与另一个不同的类中的实例变量具有相同的名称。

类可以声明一个方法，该方法可以与一个不同的类中的方法具有相同的名称，就像本章前面关于多态的小节中所展示的那样。

方法可以与实例变量具有相同的名称（对于返回一个实例变量的值的方法来说，这很常见。参见本书第 12 章）。

类的实例方法可以拥有与同一个类的类方法相同的名称。

Apple 将名称以下划线(_)字符开始的方法，视为保留供 Apple 内部使用的方法。

3.3.3 消息表达式

3.3.3 消息表达式

Objective-C 对于 C 所做出的最为重要的添加，就是消息表达式：

```
1.  [receiver message]
```

receiver 是一个对象，或者，严格地讲，是类的一个实例的指针。message 是 receiver 的类所定义的一个方法的名称，以及该方法所接受的任何参数。当执行包含了消息表达式的语句时，Objective-C 运行时确定接收者的类，并且在该类的方法列表中查找 message 中所包含的方法名，从而执行该方法相应的代码。第 5 章将介绍消息表达式。

注意　当消息中的方法是一个类方法时，接收者是一个类对象或者是类名（可以用来在消息表达式中代替类对象）。

3.3.4 编译器指令

3.3.4 编译器指令

以@字符开头的单词是编译器指令，而不是可执行代码。我们已经看到了@interface，它表示一个类定义的接口部分的开始；还有@implementation，它表示实现部分的开始；还有@end,用来表示这些部分的结束。附录 A 给出了 Objective-C 编译器指令的一个完整列表。

3.3.5 直接量字符串

3.3.5 直接量字符串

直接量字符串是保存字符串文本的常量。Objective-C 使用 NSString 的常量实例而不是普通的 C 字符串，而 NSString 是在 Foundation 框架中定义的一个类。可以用创建一个 C 字符串直接量相同的方式来创建一个 NSString 直接量，只不过在字符串开始处添加一个@，例如：

```
1.  "The Big Apple"   // Literal C string
2.  @"The Big Apple"  // Literal NSString
```

注意　严格地讲，@"The Big Apple"是一条编译器指令，它告诉编译器创建一个 NSString 直接量，其文本为"The Big Apple"。

3.3.6 Objective-C 关键字

3.3.6 Objective-C 关键字

id

id 是保存“指向对象的指针”的类型。声明为 id 的变量,可以保存指向任何 Objective-C 对象的一个指针，该对象独立于对象的类：

声明：

```
1.  id myObject;
```

它告诉你（和编译器）myObject 是指向一个对象的指针。位于该地址的字节，是某些类的一个实例的内存表示。

可以把一个更加具体的类型的变量，赋值给一个 id 类型的变量，或者反之亦然，不需要显式的强制类型转换，例如：

```
1.  NSString* string1 = @"The Big Apple";
2.  id something;
```

```
3.  NSString* string2;
4.  something = string1;
5.  string2 = something;
```

编译器将会静默地假设你知道自己在做什么。注意，实际上，id 保存的指针是内建到类型中的。这里不需要使用一个星号(*)：

```
1.  id *myObject;  // WRONG !
```

最后，不要把 id 与 void*搞混淆了。声明：

```
1.  void *someBytes;
```

没有说明 someBytes 是什么。它只是指向某些字节的一个指针。这些字节表示什么，对于编译器来说是未知的。

nil

nil 是一个定义的常量，它表示“指向没有对象的一个指针”。尽管 nil 定义为零，但它通常用于一个对象指针必须是空的情况，而不是直接使用零。

使用零不会导致任何真正的问题：

```
1.  [anObject setObjectInstanceVariable: 0];  // Bad form
```

然而，这被认为是一种糟糕的形式。使用 nil 来替代，这提醒你将要设置为空的是一个对象指针。考虑如下替代方式：

```
1.  [anObject setObjectInstanceVariable: nil];  // Correct form
```

在 Objective-C 中，可以向一个接收者发送一条消息，消息的值为 nil。nil 消息表示没有操作：它们不做任何事情，并且继续执行下一行代码。它们不会导致程序崩溃。

这一功能使你不必再编写如下所示的守护代码：

```
1.  // This check is unnecessary
2.  if ( anObject )
3.    {
4.      [anObject someMethod];
5.    }
```

BOOL

在其大多数历史阶段，C 都缺乏一个定义的布尔类型。它通过计算表达式来确定真值。如果一个表达式计算为 0，则被认为是假；相反，则为真。C99 标准添加了一个布尔类型，bool，并且还添加了真值 true 和假值 false。

Objective-C 有自己的布尔类型，BOOL，并且还有真值常量 YES 和假值常量 NO。BOOL 不是一个基本类型。它是无符号的 char 的一个 typedef（别名）。YES 和 NO 只是为 1 和 0 定义的常量。

由于 Objective-C 继承了所有的 C 类型，因此我们可以在自己的 Objective-C 程序中使用 bool 类型。然而，Cocoa 框架和大多数已有的 Objective-C 代码使用 BOOL。尽管可以在 bool 和 BOOL 之间相互转换，除非你的程序用到的库使用了 bool，否则干脆忘掉 bool 会更容易一些。

注意　　尽管 Objective-C 当前的版本是基于 C99 标准的，但 Objective-C 最初只是作为缺乏布尔类型的一个较早版本的 C 的扩展而开发的。尽管 C99 的 bool 类型也可以使用，但大多数 Objective-C 社群使用 Objective-C 的 BOOL。

SEL

SEL 是保存了一个 Objective-C 方法名表示的一种类型。SEL 是 selector 的缩写。方法名有时候叫做选择器，因为运行时使用它们来选择要执行的代码，以响应一条消息（见第 5 章）。

由于性能的原因，用于这一决定的是一个 SEL，而不是方法的实际字符串名称。字符串不便于操作，只是比较两个字符串的相等性，就是一个很慢的、多步骤的过程。比较两个 SEL 要快很多：

```
1.  SEL aSelector = @selector( aMethodName );
```

IMP

IMP 是一个 typedef，用于“一个指针，它指向接受参数 id、SEL 及可能的其他参数并且返回 id 的函数”。

编译器最终将 Objective-C 方法转换为常规的 C 函数。当执行一个方法时，运行时使用方法的选择器来找到执行该方法的实际函数，然后执行该函数。但是，偶尔为了性能的目的或者某些幕后的技巧，会有直接调用实现了一个方法函数的情况。可以通过获取该函数的一个指针，然后使用指针来调用该函数，从而做到直接调用。可以用来获取这样的指针的方法和运行时函数，以 IMP 类型返回该指针：

```
IMP methodImplementation =
    [anObject methodForSelector: @selector( aMethodName )];
```

注意　　在文档和图书中，IMP 也用作表示方法实现的一个术语，例如“使用 methodForSelector:获取 IMP”。

Class

Class 是保存了 Objective-C 的类的引用的一个类型。在 Objective-C 中，类自身也是对象。由于它们是对象，类必须实例化某些类。这个类，也就是类的类，称作（请注意）Class。就像 id 和 IMP 一样，而且与任何其他的类类型不同，Class 本身是一个指针类型，并且不需要星号(*);例如：

```
Class *myClass; //WRONG !
```

第 7 章将介绍 Class 对象。

3.3.7 Cocoa 数字类型

3.3.7　　Cocoa 数字类型

本节中的类型不是 Objective-C 语言的一部分。它们都定义于 Cocoa 框架中，但是，我们会看到它们经常在这里提及。从 Mac OS X 10.5 开始，Apple 已经使用定义的类型替代了 Cocoa 框架中 int 和 float 的大多数出现，而定义的类型的长度取决于代码是编译为 32 位可执行程序还是 64 位可执行程序（参见附录 C）。

NSInteger

NSInteger 替代了 Cocoa 框架中大多数 int 的出现。它在 32 位环境中定义为 int，在 64 位环境中定义为 long（64 位整数）。

NSUInteger

NSUInteger 是 NSInteger 的无符号形式。它替代了 Cocoa 框架中的大多数 unsigned 的出现。在 32 位环境中，它是一个无符号的 32 位整数；在 64 位环境中，它是一个无符号的 long（无符号的 64 位整数）。

CGFloat

CGFloat 替代了 float。当针对 32 位环境编译的时候，它是一个 float（32 位）；当针对 64 位环境编译的时候，它是一个 double（64 位）。Foundation 框架提供了一个定义的

常量，CGFLOAT_IS_DOUBLE，如果需要通过编程知道 CGFloat 在当前的环境中是一个 float 还是一个 double，使用下面的语句：

```
1.  if ( CGFLOAT_IS_DOUBLE )
2.    NSLog( @"Double !\n");
3.  else
4.    NSLog(@"Float !\n");
```

NSLog

NSLog 是 Foundation 框架中定义的一个用于字符输出的函数。NSLog 不是 Objective-C 语言自身的一部分，但是，这里介绍它，是因为它用于本书中的很多示例和练习中。

NSLog 类似于 printf，但是有如下一些区别：

NSLog 写入控制台日志，也写入一个终端窗口。控制台日志是操作系统负责维护的一个消息日志。在OS X上，可以使用Console应用程序(/Applications/Utilities/ Console.app)来查看控制台日志。

NSLog 的格式字符串是一个 NSString 直接量，而不是一个 C 字符串直接量。

NSLog 在打印后自动换到一个新行。不需要在格式字符串末尾添加一个额外的\n。

NSLog 使用一个额外的转换修饰符，%@，它接受一个 Objective-C 对象作为其参数。在转换中，NSLog 调用参数对象的 description 方法。该 description 方法返回一个 NSString，用以描述该对象。返回的 NSString 替代了输出中的%@，如下面的例子所示：

```
1.  NSString *aString =  @"Hello New York!";
2.
3.  NSLog( @"The object description is: %@", aString );
```

NSString 的描述只是字符串本身。执行以上代码，将会产生如下的输出：

```
1.  The object description is: Hello New York!
```

当创建自己的类时，可以覆盖 description 方法为自己的类提供定制的描述。

注意　　如果使用带有%@描述符的格式字符串，但是，忘记了提供一个对应的对象参数，那么NSLog将尝试向位于对象参数所应该放置的地址的字节发送一条description消息。这通常会导致程序崩溃。

在一个发布的程序中，不应该使用 NSLog 语句（这么做会在客户的控制台日志中产生杂乱信息），但是，对于在学习和调试过程的简单输出来说，它很有用。

NSLog 有一项功能可能很恼人，它在你要求其输出的内容前面加了一个长长的字符串信息，其中包括执行该语句的时间和日期（详细到毫秒）、可执行程序的名称，以及执行它的程序的进程 id。如下的 NSLog 语句：

```
1.  NSLog( @"Greetings from NSLog" );
```

将产生如下的输出：

```
1.  2010-02-01 11:41:26.556 a.out[33955:903] Greetings from NSLog
```

为了显示上的清晰，在本书后面的部分中，在显示 NSLog 的输出时，我们去除了额外的信息。

3.4 小结

3.4 小结

本章是对面向对象编程的一个一般性介绍。在阅读完本章后，你应该对于如下的概念有了一个基本的理解：类、对象、实例变量、方法、封装、继承、多态和消息。

本章的第二部分尝试说明这些概念是如何在 Objective-C 中实现的。第 4 章将在这部分内容的基础上，带领你一步一步地了解一个较小但完整的 Objective-C 程序。我们将学习如何使用 Apple 的集成开发环境 Xcode 来构建和运行程序。