```
(gdb) p &n
$4 = (<data variable, no debug info> *) 0x200ac
(gdb) p &result
$5 = (<data variable, no debug info> *) 0x200b0
```

这告诉你*n*的值存放在地址0x200ac中，result的值存放在地址0x200b0中。注意，这些值在内存中是连续的，因为*n*和result都是4个字节。你可以用x命令检查内存：

```
(gdb) x/2xw 0x200ac
0x200ac:        0x00000005      0x00000000
```

x/2xw命令的意思是检查两个连续的值，用十六进制显示，每个都是“字”大小（4个字节），起始地址为0x200ac。因此，在这里你会再次看到*n*为5，result为0。这是查看内存的另一种方法，这次没有使用命名标签。

回到程序——你已经确认了初始内存值是按照预期设置的。继续执行到下一个断点，验证r0是否被设置为*n*的初始值。

```
(gdb) continue
Continuing.

Breakpoint 2, 0x0001007c in _start ()

(gdb) disas
Dump of assembler code for function _start:
   0x00010074 <+0>:     ldr     r1, [pc, #40]   ; 0x100a4 <end+20>
   0x00010078 <+4>:     ldr     r0, [r1]
=> 0x0001007c <+8>:     subs    r3, r0, #1
   0x00010080 <+12>:    ble     0x10090 <end>
End of assembler dump.

(gdb) info registers r0
r0             0x5      5
```

从前面的输出可以看到，程序已经按预期推进到指令0x1007c，且r0的值也是预期的5（*n*的值）。到目前为止，都很好。现在，前进到下一个断点，r0应该是计算5的阶乘所得值，即120。你可以把continue命令缩写为c，info registers命令缩写为i r。

```
(gdb) c
Continuing.

Breakpoint 3, 0x00010090 in end ()

(gdb) disas
Dump of assembler code for function end:
=> 0x00010090 <+0>:     ldr     r1, [pc, #16]   ; 0x100a8 <end+24>
   0x00010094 <+4>:     str     r0, [r1]
   0x00010098 <+8>:     mov     r0, #0
   0x0001009c <+12>:    mov     r7, #1
   0x000100a0 <+16>:    svc     0x00000000
   0x000100a4 <+20>:    andeq   r0, r2, r12, lsr #1
   0x000100a8 <+24>:    strheq  r0, [r2], -r0   ; <UNPREDICTABLE>
End of assembler dump.

(gdb) i r r0
r0             0x78     120
```

一切看上去都很好。回想一下，此时阶乘输出还未保存到result内存地址。现在，验证result还没有改变：

```
(gdb) p (int)result
$6 = 0
```

虽然阶乘输出暂时存放在r0中，但它还未写入内存。继续前进到程序结尾（最后一个断点），看看result内存位置是否已经更新：

```
(gdb) c
Continuing.

Breakpoint 4, 0x000100a0 in end ()

(gdb) p (int)result
$7 = 120
```

你应该看到result的值为120。如果是，那么很好，程序是按预期工作的。

破解程序以计算另一个阶乘

这个程序是硬编码计算5的阶乘的。如果想让它计算其他数的阶乘，该怎么办呢？你可以修改fac.s源代码中的硬编码值，重新编译代码并再次运行它。你也可以编写一些代码，使用户能在运行时输入期望的*n*值。但是，

想象一下，如果你不再拥有访问源代码的权限，并且你只想用一种快捷方式来改变程序运行时的行为，即用除了5之外的其他值来代替硬编码的*n*值。

首先，用run命令重启程序，并用y和n来回答问题：

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y


Starting program: /home/pi/fac

Breakpoint 1, 0x00010074 in _start ()
```

现在，回到程序的起点，即第一个断点处。你可以编辑*n*的内存值，把它设置为7而不是5。首先，获取*n*的内存地址，把这个地址处的值设为7。然后，输出*n*以确保已修改成功。

```
(gdb) p &n
$8 = (<data variable, no debug info> *) 0x200ac

(gdb) set {int}0x200ac = 7

(gdb) p (int)n
$9 = 7
```

现在转到程序的结尾，查看result是否已更新为预期的7的阶乘值，即5040。你可以去掉中间的两个断点（编号2和3），这样可以直接到达结尾：

```
(gdb) disable 2
(gdb) disable 3

(gdb) c
Continuing.

Breakpoint 4, 0x000100a0 in end ()

(gdb) p (int)result
$10 = 5040
```

此时，你应该看到result的值为5040。如果是，那么你已经成功地破解了一个程序，使其执行你的命令——而且完全没有涉及源代码！

现在，你可能想尝试把*n*设置为其他值，看看是否能得到预期的结果。为此，用run命令重启程序，编辑*n*的内存值，继续前进到最后一个断点，然后检查result的值。但是，如果使用的*n*值大于12，则会得到错误的结果。参阅附录A中练习8-1的答案来了解其原因。

如果你允许程序继续执行到结尾，那么进程退出，你会得到一条类似于"Inferior 1（process 946）exited normally"的消息。这不是对你代码的侮辱，"inferior"只是gdb对正在调试的目标的称呼！你可以随时通过在gdb中输入quit来退出调试器。

设计13：检查机器码

前提条件：设计12。

假设给你的是fac可执行文件，而不是原始的汇编语言源文件。你想知道程序是做什么的，但你又没有源代码。如同你在设计12中看到的，你可以用gdb调试器来检查fac可执行文件。本设计将展示一组用于检查机器码的不同工具。

在Raspberry Pi上打开一个终端。默认情况下，终端应打开主文件夹，用~符号表示。这个文件夹中应该有三个与阶乘运算相关的文件，它们来自设计12。用如下命令来检查它们：

```
$ ls fac*
```

你会看到：

❑fac可执行文件。

❑fac.o汇编时生成的目标文件。

❑fac.s汇编语言源代码文件。

在这个虚构的场景中，你只有可执行文件fac，并且你想知道从这个文件的内容可以了解到程序的哪些信息。首先，用hexdump工具查看文件所含字节的十六进制值：

```
$ hexdump -C fac
```

hexdump输出的开头应该类似于图8-4（没有注释），显示fac可执行文件中的字节。

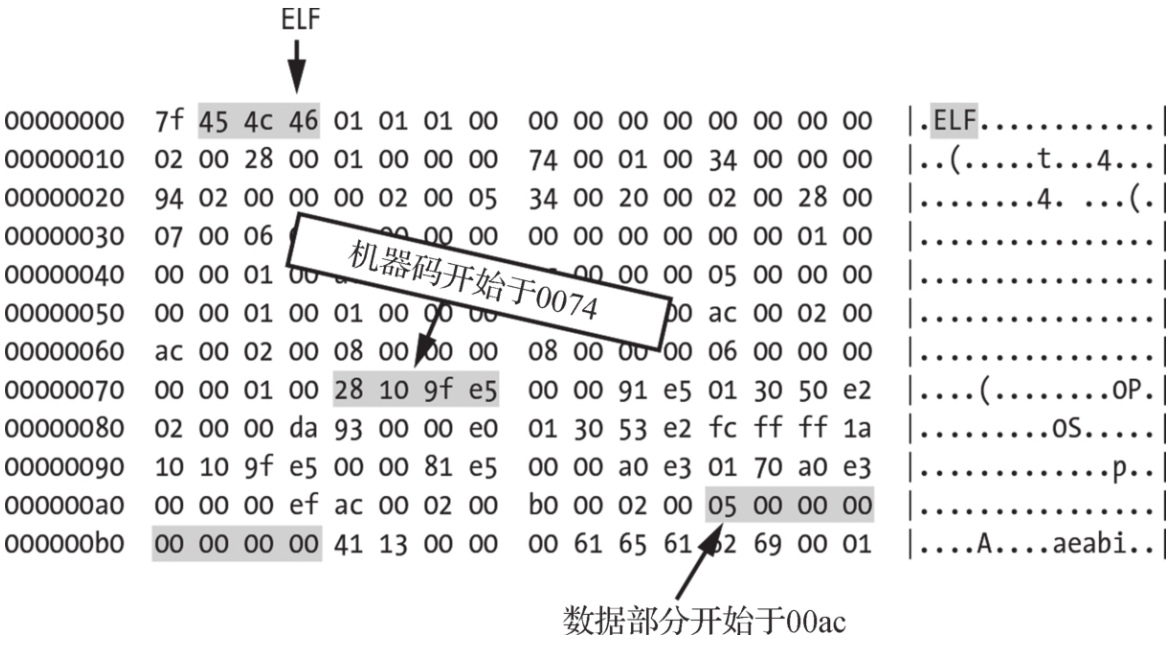

图8-4 Linux可执行文件的机器码

你看到的只是文件中字节的顺序列表，每个字节显示为两个字符的十六进制值。如果该命令的输出太大，以至于终端窗口装不下，就向上滚动来查看开头的字节。左列的八字符十六进制数表示对应行上第一个字节在文件中的偏移量。每行有16个字节，这意味着每行（沿左侧）的偏移量增幅为0x10。输出的右侧是解释为ASCII的相同的字节。不能对应可输出ASCII字符的字节代码用句点表示。

偏移量为00000000的地方是文件的开头，在这里你会看到7f，后面跟着45 4c 46（ASCII的ELF）。它表示这个文件是可执行或可链接的格式（Executable and Linkable Format，ELF）。ELF文件是可执行程序的标准Linux格式。这4个字节标识ELF头，是描述文件内容的一组属性。ELF头的后面是程序头，它提供操作系统运行程序所需的详细信息。

看过头信息后，便可以找到包含程序机器指令的文本部分。在我的系统中，偏移量00000074处是文本部分的开头，文本以字节28 10 9f e5开头。如果把这些字节从后往前重新排列，就会得到e59f1028，它是ldr r1, [pc, #40]的机器码指令。在这个部分中，每一组4个字节都是一条机器指令。用这种方式来查看程序可以很好地提醒，fac程序的代码仅仅表示为字节序列。参阅图8-1，了解如何用二进制表示机器码。

在后面的输出中——在我的系统中是偏移量为000000ac的地方，你会看到文件的数据部分，其中包含了程序定义的两个初始4字节的值。这里看不到n和result标签，但能看到05 00 00 00和00 00 00 00。在你的系统中，这些字节的偏移量可能和我的不一样。

附带说明一下，计算机保存较大数值数据的字节顺序称为"端序"（endianness）。当计算机首先保存最低有效字节（在最低地址）时，称为"小端序"。首先保存最高有效字节时，称为"大端序"。在hexdump输出中，你看到的是小端序存放，因为32位机器指令e59f1028的保存顺序是：28 10 9f e5。首先保存的是最低有效字节。*n*和result的值也是如此。*n*的值保存顺序为05 00 00 00，当你把它看成32位整数时，它表示的是00000005。

如果你想查看这个十六进制数据的某些部分，但又要把其分成多个部分时，可以使用objdump工具：

```
$ objdump -s fac
```

这会转储出一些与之前相同的字节，但是分成了多个部分，如下所示：

```
Contents of section .text:
 10074 28109fe5 000091e5 013050e2 020000da  (........0P.....
 10084 930000e0 013053e2 fcffff1a 10109fe5  .....0S.........
 10094 000081e5 0000a0e3 0170a0e3 000000ef  .........p......
 100a4 ac000200 b0000200                    ........
Contents of section .data:
 200ac 05000000 00000000                    ........
Contents of section .ARM.attributes:
 0000 41130000 00616561 62690001 09000000  A....aeabi......
 0010 06010801                             ....
```

注意左侧的数字是如何变化的。.text部分（即代码）不是从0074开始的，而是从10074开始的。包含*n*和result值的.data部分不是从00ac开始的，而是从200ac开始的。hexdump工具只显示文件内的字节偏移量，而objdump输出指出的是程序运行时字节加载到内存中的地址。查看ELF可执行文件中各个部分地址的另一种方法是用readelf -e fac。这将显示文件中的头信息。

你现在可以尝试objdump的另一个功能——反汇编机器码，这样你就可以在机器码字节值旁边看到汇编语言指令了。

```
$ objdump -d fac

fac:     file format elf32-littlearm

Disassembly of section .text:

00010074 <_start>:
   10074:       e59f1028        ldr     r1, [pc, #40]   ; 100a4 <end+0x14>❶
   10078:       e5910000        ldr     r0, [r1]
   1007c:       e2503001        subs    r3, r0, #1
   10080:       da000002        ble     10090 <end>

00010084 <loop>:
   10084:       e0000093        mul     r0, r3, r0
   10088:       e2533001        subs    r3, r3, #1
   1008c:       1afffffc        bne     10084 <loop>

00010090 <end>:
   10090:       e59f1010        ldr     r1, [pc, #16]   ; 100a8 <end+0x18>
   10094:       e5810000        str     r0, [r1]
   10098:       e3a00000        mov     r0, #0
   1009c:       e3a07001        mov     r7, #1
   100a0:       ef000000        svc     0x00000000
   100a4:       000200ac        .word   0x000200ac
   100a8:       000200b0        .word   0x000200b0
```

你应该看到类似于上面的输出。请注意，地址10074上的指令❶与图8-4高亮显示的字节序列相同，即机器码的前4个字节。这个输出与设计12的gdb输出非常相似。考虑一下这意味着什么：使用诸如gdb或objdump之类的工具，你可以轻松地查看任何可执行文件的机器码和相应的汇编语言代码！

使用前面描述的方法，你可以用视图的方式查看ELF可执行文件的内容。这适用于Linux系统上的所有标准ELF文件，而不仅仅是你编写的代码。请随意探索计算机上任意ELF文件的机器码。例如，如果你想查看ls的机器码——ls是之前用于列出目录内容的工具，首先，你需要找到ls ELF文件的文件系统位置，如下所示：

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

这告诉我们ls的二进制可执行文件位于/bin/ls（你可以忽略返回的任何其他结果）。现在，你可以运行objdump（或任何其他已描述过的工具）来查看ls的机器码：

```
$ objdump -d /bin/ls > ls.txt
```

这个命令的输出相当长，所以它被重定向到名为ls.txt的文件。你在终端窗口看不到反汇编的代码，它已经被写入ls.txt文件，这个文件可以用文本编辑器来查看。当然，由于Linux是开源的，因此你可以在线查找ls工具的源代码。不过，并非所有工具都是开源的，这个设计应该能让你了解到如何查看任意Linux可执行程序的反汇编代码。

第9章

高级编程

第8章研究了软件的基础：在处理器上运行的机器码，以及汇编语言（一种人类可读的机器码表示）。尽管所有的软件最终都必须转换成机器码的形式，但绝大多数软件开发人员都用更高、更抽象的语言编写程序。本章将介绍高级编程。我们将概述高级编程，讨论各种编程语言中的常见元素，并研究几个示例程序。

9.1 高级编程概述

虽然可以用汇编语言（甚至可以用机器码）编写软件，但这既费时又容易出错，并且会使软件难以维护。此外，汇编语言是特定于CPU架构的，所以，如果汇编语言开发人员想要在其他类型的CPU上运行其程序，就必须重新编写代码。为了克服这些缺点，人们开发了高级编程语言，这就允许程序用独立于特定CPU的语言来编写，并且它们在语法上更接近人类语言。许多语言都需要编译器——一种把高级程序语句转换为特定处理器机器码的程序。使用高级编程语言，软件开发人员可以编写程序一次，然后针对不同类型的处理器进行编译，有时只需对源代码进行少量修改，甚至不用修改。

编译器的输出是一个目标文件，它包含了针对特定处理器的机器码。正如我们在设计12中讨论的一样，计算机执行的目标文件的格式不正确。另一种被称为链接器的程序用于把一个或多个目标文件转换成一个可执行文件，这样操作系统就可以运行这个可执行文件了。链接器还可以在需要的时候引入其他已编译代码库。编译和链接过程如图9-1所示。

编译和链接的过程被称为打包。但是，在常见的用法中，软件开发人员在说到编译代码时，其实际意思是指包含编译、链接和任何其他把代码

转换成最终形式所需步骤的完整过程。编译器通常会自动调用链接步骤，使其对软件开发人员的可见度降低。

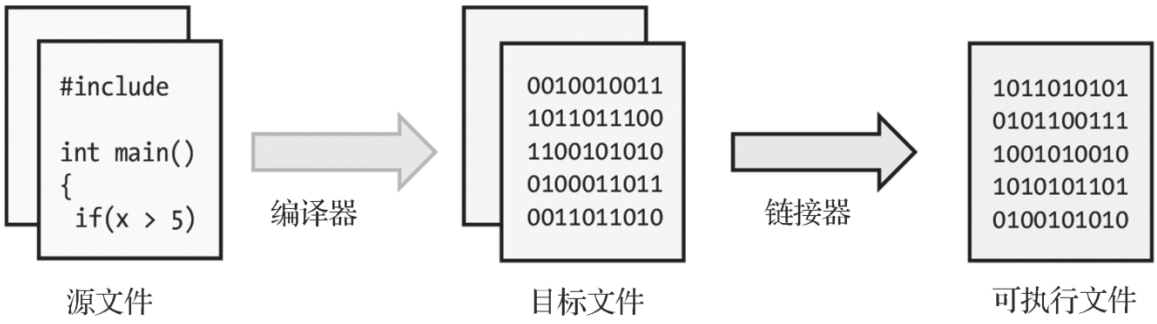

图9-1 从源代码构建可执行文件

9.2 C和Python简介

学习高级编程最好的方法是研究编程语言并用这些语言编写一些程序。本章选择两种高级语言：C和Python。这两种语言都很强大，也很有用，下面用它们来演示编程语言如何用不同的方式来提供类似的功能。我们先对它们进行简要介绍。

C编程语言可追溯到20世纪70年代早期，当时它被用于编写UNIX操作系统的一个版本。尽管C是一种高级语言，但它与底层机器码离得并不远，这使得它成为操作系统开发或其他直接与硬件交互的软件开发的绝佳选择。20世纪80年代出现了C的升级版，称为C++。C和C++是功能强大的语言，几乎可以用来完成所有事情。然而，这些语言很复杂且没有提供很多防止程序员错误的保护措施。对于需要与硬件交互的程序以及那些要求高性能的程序（比如游戏）来说，它们仍然是主流的选择。C对于教育目的也很有用，它提供了底层概念与高级概念之间的直接映射，这就是本章选择它的原因。

与C相比，Python编程语言离底层硬件更远。Python最早是在20世纪90年代发布的，历经多年越来越受欢迎。它以易读且对初学者友好而闻名，同时它还能提供支持复杂软件设计所需的一切。Python有“自带电池”的理念，意思是Python的标准发行版包含了有用的功能库，开发人员能轻松地

将其用于程序设计中。Python的直接属性使其成为教授编程概念的上佳选择。

现在，我们来看看大多数高级编程语言中都有的元素。我们的目标不是把你教成特定语言的程序员，而是让你熟悉编程语言中常见的思想。请记住，高级编程语言中的功能是CPU指令的抽象。如你所知，CPU提供访问内存、逻辑运算和控制程序流的指令。让我们来看看高级语言如何揭开这些底层功能。

9.3 注释

我们先介绍编程语言的一个特性，这个特性实际上并不会指示CPU做事情。几乎所有的编程语言都支持在代码中添加注释。注释是源代码中对代码解读的文本。注释是给其他开发人员看的，通常会被编译器忽略，它们对编译的软件没有任何影响。在C编程语言中添加注释的方式如下所示：

```
/*
  This is a C-style comment.
  It can span multiple lines.
*/

// This is a single-line C comment, originally introduced in C++.
```

Python用#表示注释，如下所示：

```
# This is a comment in Python.
```

Python不支持多行注释，程序员只需一行接一行地使用多个单行注释即可。

9.4 变量

内存访问是处理器的基本功能，因此它也是高级语言的必备功能。编程语言访问内存的最基本方式是通过变量。变量是一个确定的内存地址。变量允许程序员给内存地址（或内存地址范围）指定一个“名字”，然后通过

这个名字访问对应地址的数据。在大多数编程语言中，变量都有“类型”，以说明它们所保存数据的种类。例如，一个变量可能是整数类型或文本字符串类型的。变量也有一个“值”，它是存储在内存中数据。变量还有一个“地址”，尽管这通常对程序员而言是隐藏的，这个地址是变量值在内存中的存储位置。最后，变量有“作用域”，意思是只有在程序的某些部分（即作用域）才能访问它们。

9.4.1 C中的变量

我们来看一个C编程语言变量的例子：

```
// Declare a variable and assign it a value in C.
int points = 27;
```

这段代码声明了一个变量，变量名为points，类型是int，在C语言中这表示该变量保存整数。该变量被分配了一个值，即27。当运行代码时，十进制数27被保存到某个内存地址，不过开发人员不用操心存储这个变量的具体地址。当前，大多数C编译器把int视为32位的数，因此在运行时（程序执行的时间）为这个变量分配4个字节（4B×8bit/B=32bit），而变量的内存地址指的就是第一个字节的地址。

现在声明第二个变量并为其赋值，然后看看这两个变量是如何分配内存的。

```
// Two variables in C
int points = 27;
int year = 2020;
```

我们先后声明了两个变量：points和year。它们都是整数，所以每个变量都需要4个字节的存储空间。变量在内存中的存储方式如表9-1所示。

表9-1 变量在内存中的存储方式

地址	变量名	变量值
0x7efff1cc	?	?
0x7efff1d0	year	2020
0x7efff1d4	points	27
0x7efff1d8	?	?

表9-1中使用的内存地址只是例子，实际地址随硬件、操作系统、编译器等不同而不同。注意地址增幅为4，因为存储的是4字节整数。已知变量前后的地址都用问号表示变量名和变量值，因为根据前面的代码，我们并不知道存储在那里的是什么。

注意

请参阅设计14查看内存中的变量。

如同"变量"这个名字所暗示的，变量的值是可以变化的。在前面的C程序中，如果需要把points设置为其他值，我们只需简单地在后面的程序中这样写：

```
// Setting a new points value in C
points = 31;
```

请注意，与前面的C代码片段不同，这行代码没有在变量名的前面指定int类型或任何其他类型。我们只需要在最初声明变量时指定类型即可。在这个例子中，变量已经在前面声明过了，所以这里只需给它赋值即可。不过，C语言要求变量类型保持不变，因此，一旦points被声明为int，那么就只能为它赋整数值。如果试图赋其他类型的值，比如文本字符串，就会在代码编译时出错。

9.4.2 Python中的变量

并不是所有的语言都要求声明类型。例如，Python就可以像下面这样声明并赋值变量：

```
# Python allows new variables without specifying a type.
age = 22
```

在这个例子中，Python可识别出数据类型为整数，因此程序员不用指定类型。与C语言不同，变量类型可以随着时间变化，所以下列语句在Python中是有效的：

```
# Assiging a variable a value of a different type is valid in Python.
age = 22
age = 'twenty-two'
```

让我们仔细看看在这个例子中实际发生了什么。Python变量没有类型，但赋给它的值是有类型的。这是一个重要的区别：类型与值相关联，而不是与变量相关联。Python变量可以引用任何类型的值。当变量被分配新值时，并不是变量的类型真的发生了变化，而是变量绑定了不同类型的值。C语言则与此相反，C语言变量自身有类型，且只能保存该类型的值。这个差异解释了为什么Python中的变量可以被赋予不同类型的值，而C中的变量则不可以。

注意

请参阅设计15更改Python中变量引用的值的类型。

9.5 栈和堆

当程序员使用高级语言访问内存时，后台管理内存的细节是模糊的，具体取决于使用的编程语言。像Python这样的编程语言会让程序员几乎看不到内存分配的细节，而像C这样的语言则会暴露一些底层的内存管理机制。不管这些细节是否暴露给程序员，程序通常使用两种类型的内存：栈和堆。

9.5.1 栈

栈（stack）是内存的一个区域，其运行模型为后进先出（Last-In First-Out，LIFO）。也就是说，最后入栈的项是最先出栈的。你可以把栈想象成一叠盘子。当你往这一叠盘子上放新盘子时，新的盘子是放在顶部的。当要从这个栈上取盘子时，最先拿掉的是顶部的盘子。这并不是说栈中的内容只能按LIFO顺序访问（读或修改）。实际上，当前栈中的任何项在任何时候都可以被读取和修改。但是，当从栈中移除不需要的项时，它们会以从上到下的顺序被丢弃，也就是说，最后入栈的项会最先被删除。

保存栈顶值的内存地址存储在“栈指针”处理器寄存器中。当向栈顶添加一个值时，会调整栈指针的值以增加栈的大小，为新值腾出空间。当从栈顶移除一个值时，也会调整栈指针的值以减小栈的大小。

编译器生成的代码利用栈来跟踪程序执行的状态，以及保存局部变量。这个机制对高级语言程序员是透明的。图9-2展示了C程序是如何使用栈来保存之前在表9-1中介绍的两个局部变量的。

在图9-2中，首先声明points变量，并为其赋值27，这个值保存在栈中。接下来声明了year变量，并为其赋值2020。在栈中，第二个值位于前一个值“之上”。其他值将继续添加到栈顶，直到不再需要它们，此时会将它们从栈中移除。请记住，图中的每个长条只表示内存中的一个已经分配了内存地址的位置，虽然图中没有显示这些地址。你可能会惊讶地发现，在许多架构中，分配给栈的内存地址实际上会随着栈增长而减少。在本例中，这就意味着year变量的地址低于points变量的地址。