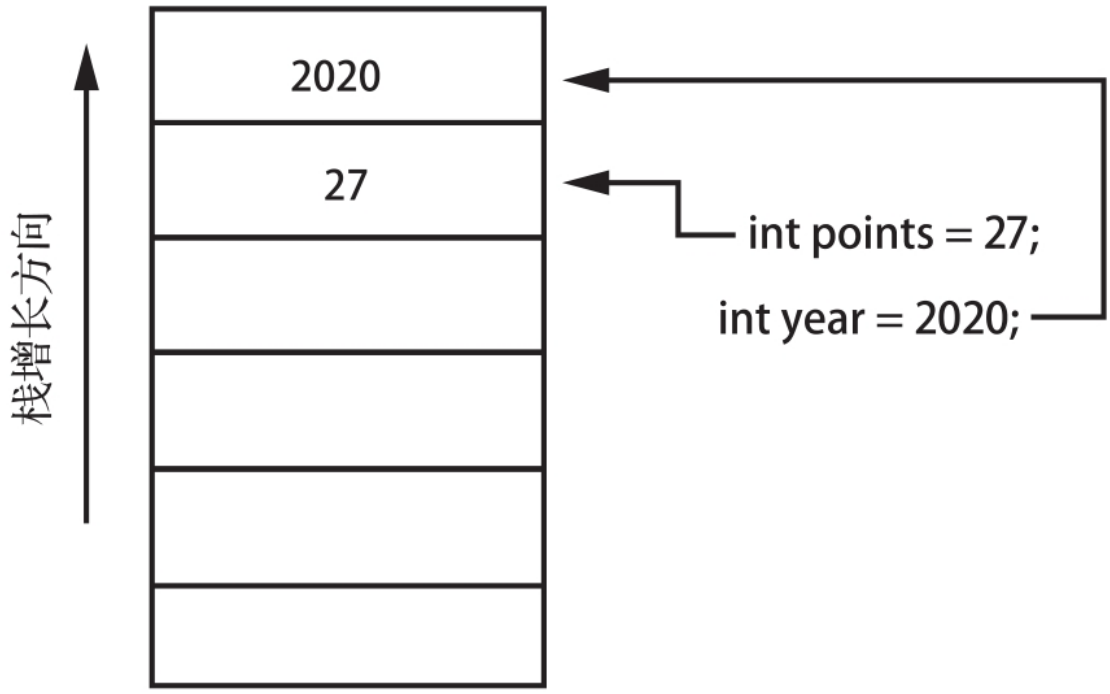

图9-2 栈用于保存C语言程序中的变量值

栈的访问速度很快，非常适合用于有限范围的小容量内存分配。独立的栈可供程序的每个执行线程使用。我们将在第10章详细地讨论线程，但现在你可以把线程看作程序中的并行任务。栈是有限资源，分配给栈的内存是有限的。在栈中存放过多的值会导致被称为“栈溢出”的故障。

9.5.2 堆

栈的作用是存放只需临时使用的小值。对于较大的或需要持续较长时间的内存分配，更适合使用堆（heap）。堆是程序可用的内存池。和栈不同，堆不采用LIFO模型，如何分配堆内存并没有标准模型。栈是特定于线程的，而程序的任何线程都可以访问堆。

程序从堆中分配内存，并且这个内存的使用会一直持续，直到它被程序释放或者程序结束。释放内存分配的意思就是把它释放回可用内存池。当分配的内存不再被引用时，有些编程语言会自动释放堆内存，执行这个操作的一种常见方法被称为“垃圾回收”。其他编程语言（比如C语言）则需

要程序员编写代码来释放堆内存。当释放未使用的内存时会发生内存泄漏。

在C编程语言中，用一种被称为指针的特殊变量来跟踪内存分配。指针就是一种保存内存地址的变量。指针值（内存地址）可以存储在栈内的局部变量中，这个值可以指向堆中的位置，如图9-3所示。

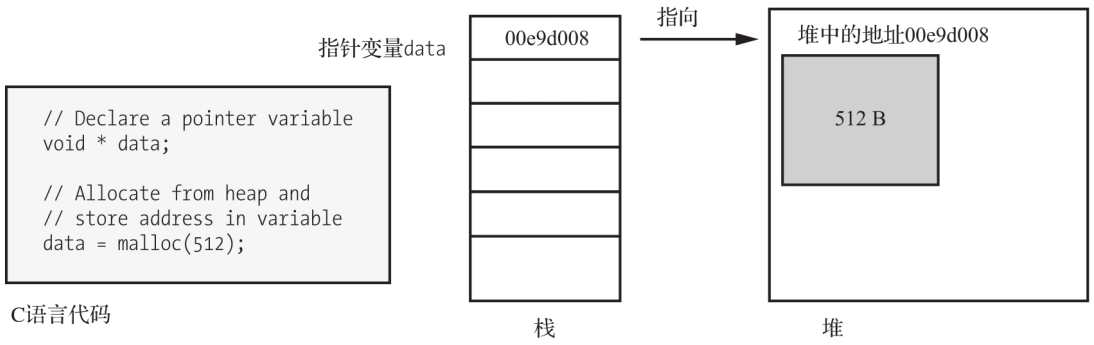

图9-3 名为data的指针变量在栈内，它指向堆中的一个地址

图9-3中的代码片段声明了一个名为data的变量。这个变量的类型是void *，意思是它是一个指向内存地址的指针（用 * 表示），这个内存地址可以保存任何类型的数据（void的含义是未指定类型）。由于data是局部变量，因此它在栈上分配。下一行代码调用了malloc，它是C语言中的一个函数，用于从堆中分配内存。程序正在申请512B的内存，malloc函数返回新分配内存中第一个字节的地址。该地址存储在栈内的data变量中。我们最终得到了一个局部变量，它在栈内的某个地址处，这个变量保存的是从堆中分配的地址。

注意

请参阅设计16亲眼看看变量是如何在正在运行的程序中分配内存的。

9.6 算术运算

由于处理器提供了执行算术运算的指令，因此高级语言也提供。汇编语言需要特定命名的指令（例如ARM处理器上用于减法运算的subs指令）

进行算术运算，不同于汇编语言，高级语言通常包含表示常见算术运算的符号，因此在代码中执行算术运算就变得简单了。包括C和Python在内的大量编程语言使用相同的运算符进行加减乘除运算，如表9-2所示。

表9-2 常用算术运算符

运算	运算符
加	+
减	-
乘	*
除	/

跨多种编程语言的另一个常见约定是用等于符号（=）表示赋值而不是表示相等。也就是说，形如x=5的语句表示把x的值设置为5。因此，可以用自然的方式为算术运算的结果变量赋值，就如下列语句一样：

```
// Addition is easy in C.
cost = price + tax;


# Addition is easy in Python too.
cost = price + tax
```

到目前为止，我们的重点在于整数运算，这在计算中很常见。但是，计算机和高级语言还支持所谓的浮点运算。与整数不同，浮点值可以表示小数。有些编程语言隐藏了这方面的细节，但是在内部，CPU对浮点运算和整数运算采用了不同的指令。在C语言中，浮点变量用浮点类型（比如float和double）声明，如下所示：

```
// Declaring a floating-point variable in C
double price = 1.99;
```

Python可以推断变量的类型，所以整数和浮点值都以同样的方式进行声明：

```
# Declaring integer and floating-point varibles in Python
year  = 2020 # year is an int
price = 1.99 # price is a float
```

整数与浮点数之间的差异有时会导致意想不到的结果。举个例子，假设有如下C代码：

```
// Dividing integers in C
int x = 5;
int y = 2;
int z = x / y;
```

你预期的z值是多少？事实证明，由于涉及的所有数都是整数，因此z的最终值是2，不是2.5。作为整数，z不能保存小数数值。

现在，我们稍微修改一下代码，如下所示：

```
// Dividing integers in C, result stored in a float
int x = 5;
int y = 2;
float z = x / y;
```

注意，z现在的类型是float。你现在预期的z值是多少？有意思的是，z现在等于2.0，仍然不是2.5！这是因为除法运算发生在两个整数之间，所以其结果也是整数。除法运算结果是2，当它被赋值给浮点变量z时，其值为2.0。C语言是很确切的，它被编译成与程序员所述非常相似的指令。这对于那些想要精确控制任务处理的程序员来说非常棒，但对于那些希望从其编程语言中获得更直观行为的程序员来说就不总是那么好了。

Python试图变得更有用：自动分配类型以允许在这种情况下获得小数结果。如果我们用Python编写与之等效的代码，z中保存的结果将为2.5。

```
# Dividing integers in Python
# z will be 2.5 and its inferred type is float
x = 5
y = 2
z = x / y
```

有些语言提供算术运算符，用缩写方式表示运算。例如，C语言提供了递增（加1）和递减（减1）运算符，如下所示：

```
// In C, we can add one to a variable the long way,
x = x + 1;
// or we can use this shortcut to increment x.
x++;
// On the other hand, this will decrement x.
x--;
```

注意

有趣的是，编程语言C++的名字意在传递一种想法，即它是对C编程语言的改进。

Python还为算术运算提供一些快捷运算符。+=和-=运算符允许程序员对变量进行加减运算，例如：

```
# In Python, we can add 3 to a variable like this...
cats = cats + 3

# Or we can do the same thing with this shortcut...
cats += 3
```

+=和-=运算符在C语言中也可以使用。

9.7 逻辑运算

如前所述，处理器非常善于执行逻辑运算，因为逻辑是数字电路的基础。编程语言也提供了逻辑处理功能。绝大多数高级语言提供两种类型的运算符来处理逻辑运算：按位运算符和布尔运算符。前者处理整数的位，后者处理布尔（真/假）值。这里的术语可能会令人困惑，因为不同的编程语言使用不同的术语。Python用“按位”和“布尔”描述，C用“按位”和“逻辑”描述，其他语言还使用其他术语。这里我们继续使用“按位”和“布尔”等说法。

9.7.1 按位运算符

按位运算符作用于整数值的每个位上，其结果为整数值。按位运算符类似于算术运算符，但并不执行加法或减法运算，而是对整数的位执行AND、OR，以及其他逻辑运算。这些运算符按照第2章介绍的真值表工作，并行地对整数的所有位进行运算。

表9-3 编程语言中常用的按位运算符

按位运算	按位运算符
AND	&
OR	|
XOR	^
NOT（取反）	~

许多编程语言（包括C和Python）都使用表9-3所示的运算符进行按位运算。

让我们看看Python中的按位运算：

```
# Python does bitwise logic.
x = 5
y = 3
a = x & y
b = x | y
```

上面代码的结果是a为1，b为7。我们来看二进制中的这些运算（见图9-4），弄清楚为什么是这样的。

图9-4 对5和3进行AND和OR按位运算

首先来看图9-4中的AND按位运算，回忆一下第2章的内容，AND意味着当两个输入都是1时，结果为1。这里一次查看一列的位。如你所见，对于两个输入来说，只有最右边的位都为1，所以AND的结果是二进制的0001或十进制的1。因此，前面的代码为a赋值1。

对于OR运算，只要有一个输入（或两个输入都）为1，结果就为1。在本例中，两个输入右边的3列位中每一列都至少有一位为1，所以结果是二进制的0111或十进制的7。因此，前面的代码为b赋值7。

练习9-1：按位运算

考虑如下Python语句。该代码执行后，a、b和c的值是多少？

```
x = 11
y = 5
a = x & y
b = x | y
c = x ^ y
```

9.7.2 布尔运算符

高级编程语言中的另一种逻辑运算符是布尔运算符。布尔运算符处理布尔值，其结果也是布尔值。

我们先花点时间谈谈布尔值。布尔值指“真”和“假”。不同的编程语言用不同的方式表示“真”和“假”。布尔变量是一个确定的内存地址，其中保存了一个布尔值（“真”或“假”）。例如，在Python中我们可以用布尔变量来跟踪某商品是否在销售：item_on_sale=True。

表达式可以计算为“真”或“假”，且其结果不用存储到变量中。例如，根据item_cost变量的值，表达式item_cost > 5在运行时可以计算为“真”或“假”。

布尔运算符允许对布尔值执行逻辑运算，比如AND、OR和NOT。例如，可以用Python的AND布尔运算符来检查两个条件是否都为真：item_on_sale AND item_cost > 5。AND左右两边的表达式的计算结果都是布尔值，所以整个表达式的计算结果也是布尔值。这里，C和Python使用不同的运算符，如表9-4所示。

表9-4 C和Python编程语言中的布尔运算符

布尔运算	C 运算符	Python 运算符
AND	&&	`and`
OR	||	`or`
NOT	!	`not`

运算符左侧的数是否大于运算符右侧的数。——译者注

当我们讨论返回布尔值的运算符时，不能遗漏比较运算符。比较运算符用来比较两个值并把比较结果评估为“真”或“假”。例如，>运算符比较两个数并确定一个是否大于另一个[注]。表9-5展示了在C和Python中都使用的比较运算符。

表9-5 C和Python编程语言中的比较运算符

比较运算	比较运算符
等于	==
不等于	!=
大于	>
小于	<
大于或等于	>=
小于或等于	<=

在前面的例子item_cost > 5中，你已经看到了其中一个运算符的用法。注意，对于等于运算符，C和Python都使用双等号，因为单等号表示赋值。这就意味着x==5是一个比较运算，需返回“真”或“假”，而x=5是一个赋值操作，表示把x的值设置为5。

9.8 程序流

布尔运算符和比较运算符用于计算表达式是否为真，但这本身并不是很有用，因为我们需要的是一种回应方式！程序流或控制流语句可以改变程序的行为以响应某些条件。让我们来看一些跨编程语言的常见程序流结构。

9.8.1 if语句

if语句（通常与else语句一起使用）允许程序员在条件为真时执行一些操作。反过来，else语句允许程序在条件为假时执行另一些不同的操作。下面是一个Python示例：

```
  # Age check in Python
❶ if age < 18:
    ❷ print('You are a youngster!')
❸ else:
    ❹ print('You are an adult.')
```

本例中，if语句❶检查age变量引用的值是否小于18。如果是，它会输出一条消息，指示用户非常年轻❷。如果age为18或更大的值❸，那么else语句告诉系统输出另一条消息❹。

下面是同一“检查年纪”的逻辑，这次用C语言编写：

```
  // Age check in C
  if (age < 18)
❶ {
    printf("You are a youngster!");
❷ }
  else


  {
    printf("You are an adult.");
  }
```

在C语言例子中，请注意if语句后面使用的花括号❶❷。这些括号标记了响应if应执行的代码块。在C语言中，代码块可以由多行代码构成，不过当代码块由单行组成时可以忽略花括号。Python不使用花括号来分隔代码块，它使用缩进格式表示。在Python中，具有相同缩进级别（例如缩进4个空格）的连续行被视为同一个代码块的组成部分。

Python还包含elif语句，其含义为“else if”。仅当某elif语句前面的if或elif语句为假时，才处理该elif语句。

```
# A better age check in Python
if age < 13:
    print('You are a youngster!')
elif age < 20:
    print('You are a teenager.')
else:
    print('You are older than a teen.')
```

同样的功能在C语言中可用else加上if来实现：

```
// A better age check in C
if (age < 13)
  printf("You are a youngster!");
else if (age < 20)
  printf("You are a teenager!");
else
  printf("You are older than a teen.");
```

注意，由于这里的代码块都是单行的，所以省略了花括号。

9.8.2 循环

有些时候，程序需要反复执行某个操作。while循环允许代码重复运行，直到满足某个条件。在下面的Python例子中，while循环被用于输出从1到20的数字：

```
# Count to 20 in Python.
n = 1
while n <= 20:
    print(n)
    n = n + 1
```

初始时，变量n被设置为1。当n小于或等于20时才开始运行while循环。由于n初始为1，符合要求，因此while循环运行，输出n的值，然后将n加1。现在，n等于2，代码回到while循环的顶部。这个过程一直持续到n等于21，此时它不再满足while循环的要求，所以循环结束。

下面是用C语言实现的相同功能：

```
// Count to 20 in C.
int n = 1;
while(n <= 20)
{
  printf("%d\n", n);
  n++;
}
```

在这两个例子中，while循环的循环体使n的值递增。实际上，还有一种更简洁的方式可以实现这一功能。for循环允许对某个范围内的数字或一组值进行迭代，以便程序员对其中每个数字执行一些操作。这里给出一个C语言的例子：输出数字1到10。

```
// C uses a for loop to iterate over a numeric range.
// This will print 1 through 10.
for(❶int x = 1; ❷x <= 10; ❸x++)
{
❹ printf("%d\n", x);
}
```

for循环声明x并将其初始值设置为1❶，表明当x小于或等于10时循环将继续❷，最后声明x在循环体运行后递增❸。通过把这些信息全部放入单行的for语句中，我们可以轻松地查看循环运行的条件。for循环的循环体只输出x的值❹。

Python的for循环则采用不同的方法，它允许程序对一组值中的每一个重复执行操作。下面的Python示例输出列表中的动物名：

```
# Python uses a for loop to iterate over a collection.
# This will print each animal name in animal_list.
animal_list = ['cat', 'dog', 'mouse']
for animal in animal_list:
    print(animal)
```

首先，声明一个动物名称列表变量，并命名为animal_list。在Python中，列表是一个有序的值集合。接着，在for循环中，对animal_list中的每一项，代码块都要运行一次，且每次代码运行时，列表中的当前值都被赋给animal变量。因此，第一次运行循环体时，animal等于cat，程序输出cat。接下来依次输出dog和mouse。

9.9 函数

循环允许一组指令连续运行多次。但是，还有一种情况也很常见，即程序多次运行一组特定指令，且不一定是循环。这样的指令可能需要从程序的不同部分、在不同的时间、用不同的输入和输出来调用。当程序员意识到同一代码需要出现在多个地方时，他们可能会把这些代码编写成函数。函数是一组程序指令，它可以被其他代码调用。函数可选地接收输入（称为"参数"）并返回输出（称为"返回值"）。不同的高级语言使用不同术语称呼函数，例如子例程、过程或方法。在某些情况下，这些不同的名称实际上传递了略有差异的含义，但在这里，我们坚持用术语"函数"。

把字符串转换成小写、将文本输出到屏幕、从互联网下载文件都是可以以函数的形式使用可重用代码执行的例子。程序员希望避免多次输入相同的代码，因为多次输入相同的代码意味着维护了相同代码的多个副本，且增加了程序总的代码量。这违反了"不重复自己"（Don't Repeat Yourself DRY）的软件工程原则，该原则鼓励减少重复性代码。

函数是封装的另一个例子。我们在之前的硬件部分看到过封装，这里我们再次到它，这次是软件封装。函数封装了代码块的内部细节，并提供了使用该代码的接口。想要使用函数的开发人员只需了解其输入和输出，无须完全了解该函数的内部工作方式。

9.9.1 定义函数

函数在使用前必须先定义。一旦定义了函数，你就可以通过调用来使用它。"函数定义"包括了函数名、输入参数、函数的程序语句（称为函数体），在某些语言中还有返回值类型。这里有一个C语言函数的例子，即给定半径，计算圆的面积的函数：

```
  // C function to calculate the area of a circle
❶ double ❷areaOfCircle(❸double radius)
  {
    double area = 3.14 * radius * radius;
  ❹ return area;
  }
```

开始的double类型❶表示函数返回一个浮点数（double是C语言中的一种浮点类型）。函数名为areaOfCircle❷，旨在传递函数的功能——在本例中为计算圆的面积。函数接收一个名为radius的输入参数❸，其类型也是double。

位于开始和结束花括号之间的是函数体，它准确定义了函数的功能。我们声明了一个名为area的局部变量，它的类型也是double。函数用$\pi \times radius^2$计算面积，并把结果赋给area变量。最后，函数返回area变量的值❹。注意，area变量的作用域是有限的，它不能在函数外被访问。当函数

返回时，局部变量area被丢弃（它可能存储在栈中），但它的值可能通过处理器寄存器被返回给调用者。

下面是一个类似的面积函数，只不过这次是用Python编写的。

```
# Python function to calculate the area of a circle
def area_of_circle(radius)
    area = 3.14 * radius * radius
    return area
```

让我们比较一下这两个函数。两个函数都用π×radius^{2}计算面积，然后返回结果值。两个函数也都接收了一个名为radius的输入参数。C语言版本显式定义返回类型为double，radius的类型也是double，而Python语言版本没有要求声明类型。Python中函数定义以def关键字开始。

9.9.2 调用函数

虽然在程序中定义了函数，但函数不一定会被运行。函数定义只是让代码在需要时可以供其他代码调用。这种调用被称为"函数调用"。调用代码传递所有需要的参数，并把控制权转给函数。然后，函数执行其代码并把控制权（以及所有输出）返还给调用者。下面演示一下如何在C语言中调用示例函数：

```
// Calling a function twice in C, each time with a different input
double area1 = areaOfCircle(2.0);
double area2 = areaOfCircle(38.6);
```

这是在Python中的调用：

```
# Calling a function twice in Python
area1 = area_of_circle(2.0)
area2 = area_of_circle(38.6)
```

当函数返回时，由调用代码把返回值存储在某个地方。两个例子都声明用变量area1和area2保存函数调用的返回值。在两种语言中，area1是12.56，area2是4678.4744。实际上，调用代码可以忽略返回值，不将它们

赋给变量，但考虑到本函数的目的，这不是十分有用。图9-5演示了调用函数时是如何把控制权暂时移交给被调用函数的。

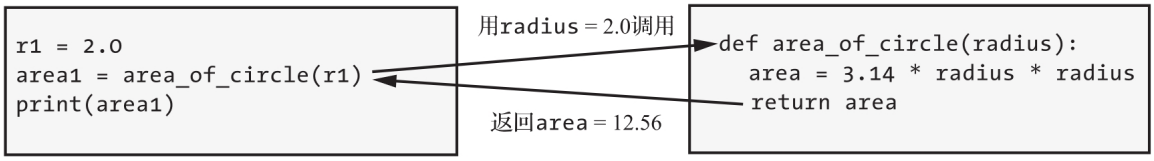

图9-5 调用一个函数

在图9-5中，左边的Python代码调用area_of_circle函数，向它传递输入参数radius的值2.0。然后，左边的代码进入等待状态，直到右边的函数完成其工作。一旦右边的函数返回，左边的代码就把返回值存储到变量area1中，然后继续执行。

9.9.3 使用库

尽管程序员确实可以自定义函数，但编程的一个重要部分是如何充分利用其他人已经编写好的函数。编程语言通常包括大量函数，它们称为该语言的"标准库"。这里，"库"是指给其他软件使用的代码集合。C和Python都有标准库，以提供函数完成诸如输出到控制台、处理文件和处理文本等功能。Python的标准库使用特别广且备受推崇。虽然并非总是如此，但一种语言的绝大多数实现都涵盖了该语言的标准库，所以程序员可以依赖这些函数。

注意

请参阅设计17用所学知识编写一个简单的Python版猜谜游戏。这包括使用Python标准库。

除了标准库，其他函数库也可以供编程语言使用。开发人员编写库给其他人使用，并以源代码或编译文件的形式进行共享。这些库有时候是非正式共享的，某些编程语言用众所周知且被接受的机制来发布库。一组共享的库称为包，共享这种包的系统称为包管理器。C语言有几种可用的包管理器，但没有一个是被C程序员广泛接受成为标准的。Python包含的包管理