第 5 章　组织成函数

虽然面向对象编程已经将注意力从函数转向对象，但函数仍然是任何程序的核心。全局函数位于对象和类之外，成员函数（也称成员方法）位于类内。

本章将学习：

- 什么是函数及其组成部分？
- 如何声明和定义函数？
- 如何向函数传递参数？
- 如何从函数返回一个值？

本章将介绍全局函数，下一章介绍函数如何在类和对象内部工作。

5.1　什么是函数

函数实际上是能够对数据进行处理并返回一个值的子程序。每个 C++程序都至少有一个函数：main( )。程序启动时，main( )函数被自动调用。main( )函数可能调用其他函数，而有些被调用的函数又调用了其他函数。

由于这些函数不是对象的一部分，它们被称为全局的，即可以在程序的任何地方访问它们。在本章中，除非特别声明，指的都是全局函数。

每个函数都有自己的名称，程序遇到函数名时，将执行函数体。这被称为调用函数。函数执行完（遇到 return 语句或函数末尾的大括号）后，程序回到函数调用的下一行继续执行。图 5.1 说明了这种流程。

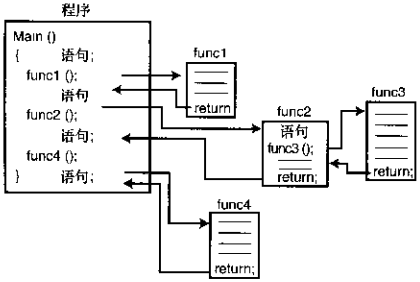

图 5.1　程序调用函数时将执行函数，然后返回到函数调用的下一行继续执行

设计良好的函数执行单个具体的、易于理解的任务，函数名指出了这种任务。复杂的任务应分成多个函数来完成，然后依次调用这些函数。

函数通常有两种类型：用户定义的函数和内置函数。内置函数是编译器软件包的一部分；由开发商提供

给用户使用。用户定义的函数是用户自己编写的函数。

5.2　返回值、参数和实参

正如第 2 章指出的，函数可以接受值，还能够返回一个值。

调用函数时，它可以完成工作，并返回一个值作为工作的结果。这个值被称为返回值，返回值的类型必须声明。下面的代码声明了一个名为 myFunction( )的函数，它返回一个 int 值：

```
int myFunction();
```

接下来看下面的声明：

```
int myFunction(int someValue, float someFloat);
```

该声明指出，myFunction 也返回一个 int 值，但它还接受两个值。

将值传递给函数时，这些值将作为变量，可以在函数中对其进行操纵。对传入值的描述称为参数列表。在前一个例子中，参数列表中包含 someValue 和 someFloat，它们分别是 int 类型和 float 类型的变量。

正如读者看到的，参数描述了函数被调用时传递给它的值的类型。传递给函数的实际值被称为实参。请看下面的语句：

```
int theValueReturned = myFunction(5,6.7);
```

上述代码将 int 变量 theValueReturned 初始化为函数 myFunction 的返回值，同时 5 和 6.7 被作为实参传递给该函数。实参的类型必须与声明的参数类型匹配。在这个例子中，将 5 和 6.7 分别传递给一个 int 变量和一个 float 变量，因此类型是匹配的。

5.3　声明和定义函数

要在程序中使用函数，必须先声明函数然后再定义它。声明将函数的名称、返回值类型和参数告诉编译器；定义将函数的工作原理告诉编译器。

如果不声明，任何函数都不能被其他函数调用。函数的声明又被称为原型。

有 3 种声明函数的方法：

- 将函数原型放在文件中，然后使用编译指令#include 将该文件包含到程序中。
- 将函数原型放在其中使用它的文件中。
- 在函数被其他函数调用前定义它。这样做时，定义将作为原型。

虽然可以在使用函数前定义它，从而避免创建函数原型，但这并不是好的编程习惯，原因有 3 个。

首先，要求函数在文件中以特定的顺序出现是不明智的。当需求发生变化时，这将使程序难以维护。

其次，在某些情况下，函数 A( )可能要调用函数 B( )，而函数 B( )也要调用函数 A( )。但不可能既在函数 A( )之前定义函数 B( )，又在函数 B( )之前定义函数 A( )。在这种情况下，至少需要声明其中一个函数。

第三，函数原型是一种优秀而强大的调试技术。如果原型指出函数将接受一组特定的参数或返回一个特定类型的值，当函数与原型不匹配时，编译器将指出错误，而不用等到运行程序时才显现出来。这像复式簿记。原型和定义相互检查，以减少输入错误导致程序错误出现问题的可能性。

虽然如此，但很多程序员还是选择第二种方式。这是因为这样可以减少代码行、简化维护工作（如果修改函数头，则必须修改原型），同时函数在文件中的顺序通常非常稳定。然而，在有些情况下，原型是必不可少的。

5.3.1 函数原型

你使用的很多内置函数的原型已经编写好，它们位于在程序中使用#include 包含进来的文件中。对于自己编写的函数，必须包含其原型。

函数原型是一条语句，这意味着它以分号结尾。它由函数的返回值类型和特征标（signature）组成。函数特征标包括函数名和参数列表。

参数列表是所有参数及其类型的列表，参数之间用逗号分开。图 5.2 说明了函数原型的组成部分。

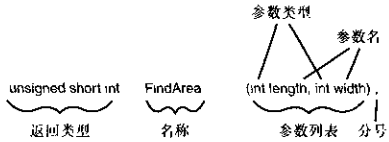

图 5.2 函数原型的组成部分

函数原型及其定义在返回类型和特征标方面必须完全相同；否则，将出现编译阶段错误。然而，函数原型可以不包含参数名，而只需包含参数类型。像下面这样的原型完全合法：

```
long Area(int, int);
```

该原型声明了一个名为 Area( )的函数，其返回类型为 long 且接受两个 int 参数。虽然这样做合法，但并非好主意。加入参数名可使原型更清晰。加入参数名后，该函数原型如下：

```
long Area(int length, int width);
```

现在，函数的功能和参数明显得多。

注意，所有函数都有一个返回值，如果没有明确指定，默认为 int。然而，明确地声明每个函数（包括 main( )）的返回类型，可使程序更清晰。

如果函数不返回任何值，将其返回类型声明为 void，如下所示：

```
void printNumber( int myNumber);
```

这声明了一个名为 printNumber 的函数，它接受一个 int 参数。由于返回类型为 void，因此不返回任何值。

5.3.2 定义函数

函数定义由函数头和函数体组成。函数头类似于函数原型，只是必须包含参数名，且不以分号结尾。

函数体是用一对大括号括起的一组语句。图 5.3 说明了函数头和函数体。

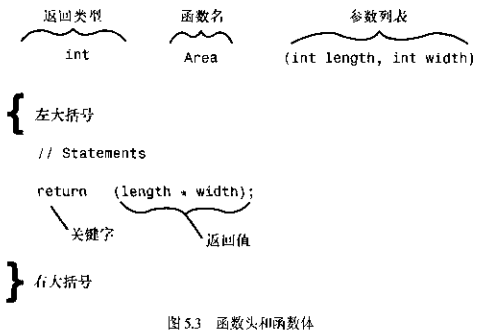

图 5.3 函数头和函数体

程序清单 5.1 中的程序包含了函数 Area( )的原型。

程序清单 5.1　函数的声明、定义和用法

```
1:  // Listing 5.1 - demonstrates the use of function prototypes
2:
3:  #include <iostream>
4:  int Area(int length, int width); //function prototype
5:
6:  int main()
7:  {
8:     using std::cout;
9:     using std::cin;
10:
11:    int lengthOfYard;
12:    int widthOfYard;
13:    int areaOfYard;
14:
15:    cout << "\nHow wide is your yard? ";
16:    cin >> widthOfYard;
17:    cout << "\nHow long is your yard? ";
18:    cin >> lengthOfYard;
19:
20:    areaOfYard= Area(lengthOfYard, widthOfYard);
21:
22:    cout << "\nYour yard is ";
23:    cout << areaOfYard;
24:    cout << " square feet\n\n";
25:    return 0;
26: }
27:
28: int Area(int len, int wid)
29: {
30:    return len * wid;
31: }
```

输出：

```
How wide is your yard? 100
How long is your yard? 200
Your yard is 20000 square feet
```

分析：

函数 Area( )的原型位于第 4 行。请将其与第 28 行的函数定义比较将发现，函数名、返回类型和参数类型都完全一致。如果它们不同，将出现编译错误。事实上，惟一的不同在于，函数原型以分号结尾且没有函数体。

另外，原型中的参数名为 length 和 width，而定义中的参数名为 len 和 wid。正如以前讨论的，原型中的参数名没有实际意义，只是为程序员提供一些信息。良好的编程习惯是，在原型和定义中使用相同的参数名；但正如程序清单 5.1 表明的，并非必须这样做。

实参按声明和定义中的参数排列顺序传递，但不进行名称匹配。如果传递 widthOfYard 和 lengthOfYard，函数 FindArea( )将把 WidthOfYard 的值用作长度，把 lengthOfYard 的值用作宽度。

注意：函数体总是由大括号括起来，即使像这个例子中那样只有一条语句。

5.4 函数的执行

调用函数时，将从左大括号后的第一条语句开始执行。使用 if 语句可以提供分支（if 及其他相关语句将在第 7 章介绍）。函数还可以调用其他函数甚至自己（参见本章后面的“递归”一节）。

函数执行完毕后，控制权将返回到调用函数。函数 main( )执行完毕后，控制权将返回到操作系统。

5.5 确定变量的作用域

变量都有作用域，作用域决定了变量在程序中的存活时间以及在什么地方可以访问它。在语句块中声明的变量的作用域为该语句块；只能在该语句块中访问它，离开该语句块后，它将不复存在。全局变量的作用域为全局，在程序的任何地方都可用。

5.5.1 局部变量

不仅可以将变量传递给函数，还可以在函数体中声明变量。在函数体内声明的变量被称为局部变量，因为它们只在函数中存在。当函数返回时，局部变量不再可用，编译器对其进行标记，以便销毁。

局部变量的定义方法与其他变量相同。传入函数的参数也可被视为局部变量，可以在函数体内使用它们，就像在函数体中定义了一样。程序清单 5.2 是一个使用参数和函数中定义的局部变量的例子。

程序清单 5.2 局部变量和参数的用法

```
#include <iostream>

float Convert(float);
int main()
{
   using namespace std;

   float TempFer;
   float TempCel;

   cout << "Please enter the temperature in Fahrenheit: ";
   cin >> TempFer;
   TempCel = Convert(TempFer);
   cout << "\nHere's the temperature in Celsius: ";
   cout << TempCel << endl;
   return 0;
}

float Convert(float TempFer)
{
   float TempCel;
   TempCel = ((TempFer - 32) * 5) / 9;
   return TempCel;
}
```

输出：

```
Please enter the temperature in Fahrenheit: 212
```

```
Here's the temperature in Celsius: 100

Please enter the temperature in Fahrenheit: 32

Here's the temperature in Celsius: 0

Please enter the temperature in Fahrenheit: 85

Here's the temperature in Celsius: 29.4444
```

分析：

在第 8 和 9 行，声明了两个 float 变量，一个用于存储华氏温度，另一个用于存储摄氏温度。第 11 行要求用户输入一个华氏温度，第 13 行将这个值传递给函数 Convert( )。

第 13 行调用 Convert( )后，跳到函数 Convert( )的第 1 行（即第 21 行）执行，这里声明了一个局部变量，其名称也为 TempCel。注意，这个局部变量与第 9 行的 TempCel 变量不是同一变量，它只在函数 Convert( )内存在。作为参数传入的值 TempFer 也只是 main( )传递的变量的一个局部拷贝。

在该函数中，可以给参数和局部变量以其他任何名称，程序的功能将不变。例如，将 TempFer 改为 FerTemp 或将 TempCel 改为 CelTemp 是完全合法的，函数的功能将不变。如果读者修改这些名称，并重新编译程序，将发现这是可行的。

在函数中，将参数 TempFer 的值减去 32，然后乘以 5 并除以 9，再将结果赋给局部变量 TempCel；然后将这个值作为函数的返回值返回。在第 13 行，这个返回值被赋给函数 main( )中的变量 TempCel，第 15 行打印它。

上述输出表明，该程序运行了 3 次。第一次的输入值为 212，这是水的沸点的华氏温度，转换后的摄氏温度为 100。第 2 次输入水的冰点温度进行测试。第 3 次输入了一个随机数，转换结果为小数。

5.5.2　作用域为语句块的局部变量

可以在函数的任何地方定义变量，而不仅限于函数开头。变量的作用域为定义它的语句所在的语句块。也就是说，如果在函数中的一对大括号内定义了一个变量，则该变量只在该语句块中可用。程序清单 5.3 说明了这一点。

程序清单 5.3　作用域为语句块的变量

```
 1: // Listing 5.3 - demonstrates variables
 2: // scoped within a block
 3:
 4: #include <iostream>
 5:
 6: void myFunc();
 7:
 8: int main()
 9: {
10:    int x = 5;
11:    std::cout << "\nIn main x is: " << x;
12:
13:    myFunc();
14:
15:    std::cout << "\nBack in main, x is: " << x;
16:    return 0;
17: }
```

```
18:
19: void myFunc()
20: {
21:    int x = 8;
22:    std::cout << "\nIn myFunc, local x: " << x << std::endl;
23:
24:    {
25:       std::cout << "\nIn block in myFunc, x is: " << x;
26:
27:       int x = 9;
28:
29:       std::cout << "\nVery local x: " << x;
30:    }
31:
32:    std::cout << "\nOut of block, in myFunc, x: " << x << std::endl;
33: }
```

输出：

```
In main x is: 5
In myFunc, local x: 8

In block in myFunc, x is: 8
Very local x: 9
Out of block, in myFunc, x: 8
Back in main, x is: 5
```

分析：

该程序首先在第 10 行对 main( )中的局部变量 x 进行初始化。第 11 行的打印结果证明 x 已被初始化为 5。在第 13 行，调用了函数 MyFunc( )。

在函数 MyFunc( )中，第 21 行将一个名称也为 x 的局部变量初始化为 8，第 22 行打印该变量的值。

第 24 行的左大括号是一个语句块的开始标记。第 25 行再次打印作用域为函数 myFunc( )的变量 x。在第 27 行，定义了一个名称也为 x 的新变量，但作用域为当前语句块；它被初始化为 9。第 29 行打印最新的变量 x 的值。

语句块在第 30 行结束。此时，离开了第 27 行定义的变量 x 的作用域，它不再可见。

第 32 行打印了变量 x 的值，这是函数 myFunc( )中第 21 行定义的 x 变量。该变量的值不受位于语句块中的第 27 行定义的 x 的影响，仍为 8。

在第 33 行，函数 myFunc( )结束，其局部变量 x 不可用。程序返回到第 14 行。在第 15 行打印第 10 行定义的局部变量 x，该变量不受 myFunc( )中定义的任何变量的影响。

显然，如果给这 3 个变量提供不同的名称，该程序将清晰得多。

5.6 参数是局部变量

传入函数的参数为函数中的局部变量。修改这些参数不会影响调用函数中的值。这被称为按值传递，这意味着将在函数中创建每个参数的局部拷贝。这些局部拷贝与其他局部变量一样，程序清单 5.4 说明了这一点。

程序清单 5.4 按值传递

```
1: // Listing 5.4 - demonstrates passing by value
```

```
 2: #include <iostream>
 3:
 4: using namespace std;
 5: void swap(int x, int y);
 6:
 7: int main()
 8: {
 9:    int x = 5, y = 10;
10:
11:    cout << "Main. Before swap, x: " << x << " y: " << y << endl;
12:    swap(x,y);
13:    cout << "Main. After swap, x: " << x << " y: " << y << endl;
14:    return 0;
15: }
16:
17: void swap (int x, int y)
18: {
19:    int temp;
20:
21:    cout << "Swap. Before swap, x: " << x << " y: " << y << endl;
22:
23:    temp = x;
24:    x = y;
25:    y = temp;
26:
27:    cout << "Swap. After swap, x: " << x << " y: " << y << endl;
28: }
```

输出：

```
Main. Before swap, x: 5 y: 10
Swap. Before swap, x: 5 y: 10
Swap. After swap, x: 10 y: 5
Main. After swap, x: 5 y: 10
```

分析：

该程序在函数 main( )中初始化两个变量，然后将它们传送给函数 swap( )，后者交换两个变量的值。然而，返回函数 main( )后再查看它们时，它们并未改变！

第 9 行对变量进行初始化，第 11 行打印它们的值。第 12 行调用函数 swap( )并传递变量。

程序跳到 swap( )函数处执行。在该函数中，第 21 行再次打印两个变量的值，结果与 main( )中相同。在第 23～25 行，交换变量的值，第 27 行的打印对此进行验证。在 swap( )函数中，两变量的值确实被互换。

然后返回到 main( )函数中的第 13 行继续执行，但在这里，变量的值并没有交换。

读者可能明白了，传递给函数 swap( )的参数是按值传递的，这意味着将在 swap( )中创建参数的局部拷贝。在第 23～25 行，交换的是局部变量的值，但函数 main( )中的变量不受影响。

在第 8 章和第 10 章，将介绍另一种传递参数的方式，这种方式让你能够修改 main( )中的变量。

5.6.1　全局变量

在函数外面定义的变量的作用域为全局，在程序的任何函数（包括 main( )）中都可用。

与全局变量同名的局部变量不会修改全局变量，但会隐藏它。如果函数中有一个与全局变量同名的局部变量，则在函数中使用该名称时，指的是局部变量而不是全局变量。程序清单 5.5 说明了这些概念。

程序清单 5.5　全局变量和局部变量

```
 1: #include <iostream>
 2: void myFunction();          // prototype
 3:
 4: int x = 5, y = 7;          // global variables
 5: int main()
 6: {
 7:    using namespacestd;
 8:
 9:    cout << "x from main: " << x << endl;
10:    cout << "y from main: " << y << endl << endl;
11:    myFunction();
12:    cout << "Back from myFunction!" << endl << endl;
13:    cout << "x from main: " << x << endl;
14:    cout << "y from main: " << y << endl;
15:    return 0;
16: }
17:
18: void myFunction()
19: {
20:    using std::cout;
21:
22:    int y = 10;
23:
24:    cout << "x from myFunction: " << x << endl;
25:    cout << "y from myFunction: " << y << endl << endl;
26: }
```

输出：

```
x from main: 5
y from main: 7

x from myFunction: 5
y from myFunction: 10

Back from myFunction!

x from main: 5
y from main: 7
```

分析：

这个简单程序说明了几个有关局部变量和全局变量的容易混淆的要点。在第 4 行，声明了两个全局变量 x 和 y。全局变量 x 被初始化为 5，y 被初始化为 7。

在 main( )函数中的第 9 和 10 行，将这些变量的值打印到控制台。在函数 main( )中，没有定义名称为 x 或 y 的局部变量；由于 x 和 y 为全局变量，因此它们在函数 main( )中可用。

第 11 行调用函数 myFunction( )，程序跳到第 18 行执行。第 22 行定义了一个局部变量 y，并将其初始化为 10。在第 24 行，myFunction( )打印变量 x 的值。这使用的是全局变量 x，就像在 main( )中一样。然而，在第 25 行使用变量名 y 时，使用的是局部变量 y，它隐藏了同名的全局变量。

函数执行完毕后返回到 main( )，再次打印全局变量的值。注意，给 my Function( )的局部变量 y 赋值，完全没有影响到全局变量 y 的值。

5.6.2　有关全局变量的注意事项

在 C++中，全局变量是合法的，但人们几乎不使用。C++源于 C，在 C 中，全局变量是一个危险但必不可少的工具。之所以必不可少，是因为程序员经常需要让数据对很多函数来说可用，而将数据作为参数在函数之间传递非常麻烦，尤其是当调用序列中的函数接受参数只是为了将其传递给其他函数时。

全局变量之所以危险，是因为它们是共享的数据，一个函数可能以另一个函数看不到的方式修改全局变量；这会导致难以发现的错误。

第 15 章将介绍一个功能强大的全局变量替代品：静态成员变量。

5.7　创建函数语句时的考虑因素

对于函数体可包含的语句数量和类型几乎没有任何限制。虽然在函数中不能定义其他函数，但可以调用其他函数，几乎在所有的 C++程序中，main( )所做的就是调用其他函数。函数甚至可以调用自身，这将在稍后有关递归的一节讨论。

虽然在 C++中，对函数的长度没有限制，但设计良好的函数通常较短。很多程序员都建议函数长度不超过一屏，这样可以同时看到整个函数。虽然这种经验规则经常被每位优秀程序员打破，但函数越短越容易理解和维护。

每个函数都应执行单个易于理解的任务。如果函数很大，应考虑能否将其分解为几个小任务。

5.8　再谈函数实参

任何合法的 C++表达式都可用作函数实参，包括常量、数学和逻辑表达式以及返回一个值的函数。重要的是，表达式的结果必须与函数期望的实参类型匹配。

甚至将函数作为参数进行传递；毕竟函数的结果为其返回类型。然而，将函数作为参数可能使代码难以理解和调试。

例如，假设有函数 myDouble( )、triple( )、square( )和 cube( )，其中每个函数都返回一个值，则可以编写这样的代码：

```
Answer = (myDouble(triple(square(cube(myValue)))));
```

可以以两种方式来看待上述语句。首先，可以认为函数 myDouble( )将函数 triple( )作为参数。而 triple( )将函数 square( )作为参数，后者又将函数 cube( )作为参数。函数 cube( )将变量 myValue 作为参数。

按相反的方向看时，该语句将变量 myValue 作为参数传给函数 cube( )，cube( )的返回值被作为参数传递给函数 square( )，square( )的返回值又被传递给 triple( )，而 triple( )的返回值又被传递给 myDouble( )。最后的结果被赋给变量 Answer。

很难确定上述的功能（是在平方之前还是之后乘以 3？），同时，如果答案不正确，也难以确定哪个函数有问题。

一种替代方式是，将每步的结果都赋给一个中间变量：

```
unsigned long myValue    = 2;
unsigned long cubed      = cube(myValue);       // cubed = 8
unsigned long squared    = square(cubed);       // squared = 64
unsigned long tripled    = triple(squared);     // tripled = 192
unsigned long Answer     = myDouble(tripled);   // Answer = 384
```

这样，可以检查每个中间结果，且执行顺序很明确。

警告：C++使得编写紧凑代码（如前述范例中将函数 cube()、square()、triple()和 myDouble()组合起来的代码）非常容易，但这并不意味着应该这样做。与使代码应尽可能紧凑相比，让代码易于阅读，进而易于维护是更好的选择。

5.9 再谈返回值

函数要么返回一个值要么返回类型为 void。void 告诉编译器，函数不返回任何值。

要从函数返回一个值，可在关键字 return 后面加上要返回的值。这可以是返回一个值的表达式。例如：

```
return 5;
return (x > 5);
return (MyFunction());
```

如果 myFunction()函数返回一个值，则上述 return 语句都是合法的。如果 x 不大于 5，第 2 条语句将返回 false，否则返回 true。返回的是表达式的值：false 或 true，而不是 x 的值。

遇到关键字 return 时，其后的表达式将作为函数的返回值，并立即返回到调用函数，return 后的所有语句都不会被执行。

在同一个函数中可以有多条 return 语句，程序清单 5.6 演示了这一点。

程序清单 5.6 可以包含多条返回语句

```
1:  // Listing 5.6 - demonstrates multiple return
2:  // statements
3:  #include <iostream>
4:
5:  int Doubler(int AmountToDouble);
6:
7:  int main()
8:  {
9:     using std::cout;
10:
11:    int result = 0;
12:    int input;
13:
14:    cout << "Enter a number between 0 and 10,000 to double: ";
15:    std::cin >> input;
16:
17:    cout << "\nBefore doubler is called... ";
18:    cout << "\ninput: " << input << " doubled: " << result << "\n";
19:
20:    result = Doubler(input);
21:
22:    cout << "\nBack from Doubler...\n";
23:    cout << "\ninput: " << input << "  doubled: " << result << "\n";
24:
25:    return 0;
26: }
27:
28: int Doubler(int original)
29: {
```

```
30:    if (original <= 10000)
31:       return original * 2;
32:    else
33:       return -1;
34:    std::cout << "You can't get here!\n";
35: }
```

输出：

```
Enter a number between 0 and 10,000 to double: 9000

Before doubler is called...
input: 9000 doubled: 0

Back from doubler...

input: 9000  doubled: 18000

Enter a number between 0 and 10,000 to double: 11000

Before doubler is called...
input: 11000 doubled: 0

Back from doubler...
input: 11000  doubled: -1
```

分析：

第 14 和 15 行要求用户输入一个数，第 17 和 18 行打印这个数和局部变量 result。第 20 行调用函数 Doubler( )，并将 input 作为参数传递给它。结果被赋给局部变量 result，第 23 行打印这个值。

在函数 Doubler( )中的第 30 行，检测参数是否大于 10 000。如果不是，函数返回 original 的 2 倍；否则返回–1，作为错误标记。

第 34 行的语句永远不会被执行。因为无论输入值是否大于 10 000，函数都将在第 31 或 33 行（到达第 34 行之前）返回。优秀的编译器将发出警告，指出该语句不可能被执行；优秀的程序员应删除它。

FAQ

int main( )与 void main( )的区别何在？应使用哪一个？这两种方式我都用过，都正常，那么，为什么要使用 int main( ){return 0;}呢？

答：对于大多数编译器来说，两者都可行，但只有 int main( )符合 ANSI 标准。因此，只能保证 int main( )在将来仍可行。

两者的区别如下：int main( )向操作系统返回一个值。当程序结束时，返回值可被批处理程序捕获。

在本书的程序中，不会使用这个返回值（例外的情况很少），但 ANSI 标准要求使用 int main( )。

5.10　默认参数

对每个在函数原型或定义中声明的变量，调用函数都必须为其传递一个值。传递的值必须是声明的类型。因此，如果像下面这样声明了一个函数：

```
long myFunction(int);
```

该函数必须接受一个 int 变量。如果函数定义与此不符，或者你传递的不是整数，将出现编译错误。

对于这种规则一种例外是，函数原型声明了参数默认值。默认值是在没有提供参数值时使用的一个值。

上述声明可以重写成这样：

```
long myFunction (int x = 50);
```

该原型指出：函数 myFunction( )接受一个 int 参数，并返回一个 long 值；如果没有提供参数，则使用默认值 50。由于函数原型中可以不包含参数名，因此上面的声明也可写成这样：

```
long myFunction (int = 50);
```

声明默认参数对函数定义没有影响。在上述函数的定义中，函数头如下：

```
long myFunction (int x)
```

如果调用该函数时没有提供参数，编译器将把 x 设置为默认值 50。在原型和函数头中，默认参数的名称可以不同，默认值是根据位置而不是参数名指定的。

可以给任何函数参数指定默认值，但有一条限制：如果某个参数没有默认值，它前面的所有参数都不得有默认值。

如果函数原型如下：

```
long myFunction (int Param1, int Param2, int Param3);
```

则仅当给 Param3 指定默认值后，才能给 Param2 指定默认值；仅当给 Param2 和 Param3 都指定了默认值后，才能给 Param1 指定默认值。程序清单 5.7 演示了默认值的用法。

程序清单 5.7　默认参数值

```
 1: // Listing 5.7 - demonstrates use
 2: // of default parameter values
 3: #include <iostream>
 4:
 5: int AreaCube(int length, int width = 25, int height = 1);
 6:
 7: int main()
 8: {
 9:    int length = 100;
10:    int width = 50;
11:    int height = 2;
12:    int area;
13:
14:    area = AreaCube(length, width, height);
15:    std::cout << "First area equals: " << area << "\n";
16:
17:    area = AreaCube(length, width);
18:    std::cout << "Second time area equals: " << area << "\n";
19:
20:    area = AreaCube(length);
21:    std::cout << "Third time area equals: " << area << "\n";
22:    return 0;
23: }
24:
25: AreaCube(int length, int width, int height)
26: {
27:
28:    return (length * width * height);
29: }
```

输出：

```
First area equals: 10000
Second time area equals: 5000
Third time area equals: 2500
```

分析：

在第5行，函数AreaCube( )的原型指定该函数接受3个int参数，其中最后两个有默认值。

该函数计算立方体的面积，立方体的尺寸是通过参数传入的。如果没有传递宽度，则宽度为25，高度为1。如果传入宽度但没有传入高度，则高度为1。如果没有传递宽度，则不能传递高度。

在第9~11行，对length、width和height进行初始化。第14行将它们传递给函数AreaCube( )。第15行打印使用这些数值计算得到的结果。

然后继续执行第17行，这里再次调用了函数AreaCube( )，但没有指定高度。因此使用默认值，再次计算面积并打印结果。

然后继续执行第20行，这次既没有指定宽度，也没有指定高度。该函数调用导致第二次跳到第25行执行。函数使用默认值计算面积，然后返回到main( )函数，在这里打印最后的值。

应该：

请切记，函数参数在函数中的局部变量。

请切记，在一个函数中修改全局变量，将影响所有使用该变量的函数。

不应该：

如果第二个参数没有默认值时，不要为第一个参数指定默认值。

别忘了，按值传递参数时不会影响调用函数中变量的值。

5.11 重载函数

C++允许创建多个名称相同的函数，这被称为函数重载。在这些同名函数的参数列表中，必须有不同的参数类型、参数个数或兼而有之。下面是一个例子：

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

函数myFunction( )被重载了三次。前两个版本之间的区别在于参量类型不同，第三个版本与前两个版本之间的区别在于参量个数不同。

重载函数的返回类型可以相同，也可以不同。

注意：如果两个函数的函数名和参数列表都相同，但返回类型不同，将导致编译错误。要修改返回类型，必须同时修改特征标（名称和/或参数列表）。

函数重载也叫做函数多态（polymorphism）。多态指的是多种形态。

函数多态指的是可以对函数进行重载，使之有多种含义的能力。通过修改参数的个数或类型，可以让多个函数使用相同的名称，进而根据指定的参数，调用与之匹配的函数。这让你能够只使用同一个函数来计算int值、double值或其他类型值的平均值，而不必为每个函数使用不同的函数名，如AverageInts( )、AverageDoubles( )等。

假想编写了一个将输入值翻倍的函数，可能希望能够将int、long、float或double参数传递给它。如果不使用函数重载，必须创建4个函数名：

```
int DoubleInt(int);
long DoubleLong(long);
```

```
float DoubleFloat(float);
double DoubleDouble(double);
```

使用函数重载，可以做如下声明：

```
int Double(int);
long Double(long);
float Double(float);
double Double(double);
```

这更容易理解和使用。你不必考虑应该调用哪个函数，而只需传递一个变量，将自动调用正确的函数。程序清单 5.8 演示了函数重载的用途。

程序清单 5.8　函数多态

```
// Listing 5.8 - demonstrates
// function polymorphism
#include <iostream>

int Double(int);
long Double(long);
float Double(float);
double Double(double);

using namespace std;

int main()
{
   int     myInt = 6500;
   long    myLong = 65000;
   float   myFloat = 6.5F;
   double  myDouble = 6.5e20;

   int     doubledInt;
   long    doubledLong;
   float   doubledFloat;
   double  doubledDouble;

   cout << "myInt: " << myInt << "\n";
   cout << "myLong: " << myLong << "\n";
   cout << "myFloat: " << myFloat << "\n";
   cout << "myDouble: " << myDouble << "\n";

   doubledInt = Double(myInt);
   doubledLong = Double(myLong);
   doubledFloat = Double(myFloat);
   doubledDouble = Double(myDouble);

   cout << "doubledInt: " << doubledInt << "\n";
   cout << "doubledLong: " << doubledLong << "\n";
   cout << "doubledFloat: " << doubledFloat << "\n";
   cout << "doubledDouble: " << doubledDouble << "\n";

   return 0;
```

```
40: }
41:
42: int Double(int original)
43: {
44:    cout << "In Double(int)\n";
45:    return 2 * original;
46: }
47:
48: long Double(long original)
49: {
50:    cout << "In Double(long)\n";
51:    return 2 * original;
52: }
53:
54: float Double(float original)
55: {
56:    cout << "In Double(float)\n";
57:    return 2 * original;
58: }
59:
60: double Double(double original)
61: {
62:    cout << "In Double(double)\n";
63:    return 2 * original;
64: }
```

输出：

```
myInt: 6500
myLong: 65000
myFloat: 6.5
myDouble: 6.5e+20
In Double(int)
In Double(long)
In Double(float)
In Double(double)
DoubledInt: 13000
DoubledLong: 130000
DoubledFloat: 13
DoubledDouble: 1.3e+21
```

分析：

函数 Double( )被重载成能够接受 int、long、float 或 double 作为参数。函数原型位于第 5～8 行，定义位于第 42～64 行。

在这个例子中，第 10 行使用了语句 using namespace std;，它不在任何函数中，这使得该声明在整个文件中有效，因此该名称空间在该文件中声明的所有函数中都可用。

在主程序中，声明了 8 个局部变量。在第 14～17 行，对其中的 4 个变量进行了初始化；在第 29～32 行，将这 4 个变量分别传递给 Double( )函数，并将结果分别赋给余下的 4 个变量。调用 Double( )函数时，并没有指定要调用哪个版本；只需传递一个参数，将调用正确的版本。

编译器对参数进行检查，已决定选择 4 个 Double( )函数中的哪一个。输出表明，4 个函数被依次调用，与预期相同。

5.12 函数特有的主题

函数对编程非常重要，一些函数特有的主题对解决不同寻常的问题可能会有所帮助。如果使用得当，内联函数有助于提高性能；函数递归是奇妙、高深的编程技巧之一，它常常能使其他方法无法解决的问题迎刃而解。

5.12.1 内联函数

当你定义函数时，编译器通常会在内存中创建一组指令。当你调用函数时，程序将跳到这些指令处执行，当函数返回时，程序跳到调用函数的下一行继续执行。如果调用同一个函数 10 次，则每次程序都将跳到同一组指令处。这意味着内存中只有一个函数拷贝，而不是 10 个。

跳入和跳出函数都存在一些影响性能的开销。有些函数非常小，只有一两行代码，如果可避免程序仅为执行一两条指令而进行跳转，可能提高程序的效率。程序员说到效率时，通常指的是速度。如果能够避免调用函数，程序的运行将更快。

如果声明函数时使用了关键字 inline，编译器将不会创建函数，而直接将内联函数的代码复制到调用函数中。这样，便不需要进行跳转，就像函数的语句被写到调用函数中一样。

注意，内联函数可能使成本大大增加。如果函数被调用 10 次，内联代码将被复制到这 10 个函数调用处。这样，在速度方面的改善可能微不足道，而可执行程序变大了，因此程序的速度可能更慢。

事实上，现在优化编译器在做出这种决策方面比你更出色，因此除非函数只有一两条语句，否则不要将函数声明为内联的。如果没有把握，则不应使用关键字 inline。

注意：性能优化是一个复杂的问题，大多数程序员在没有帮助的情况下，并不擅强于找出程序中存在性能问题的地方。在这里，帮助指的诸如调试器和剖析器等专用程序。

另外，与对那种情况下运行速度会更快或更慢进行猜测，进而编写出难以理解的代码相比，编写清晰易懂的代码总是一种更好的选择。这是因为提高易于理解的代码的速度更容易。

程序清单 5.9 演示了一个内联函数。

程序清单 5.9 内联函数

```
1:  // Listing 5.9 - demonstrates inline functions
2:  #include <iostream>
3:
4:  inline int Double(int);
5:
6:  int main()
7:  {
8:     int target;
9:     using std::cout;
10:    using std::cin;
11:    using std::endl;
12:
13:    cout << "Enter a number to work with: ";
14:    cin >> target;
15:    cout << "\n";
16:
17:    target = Double(target);
18:    cout << "Target: " << target << endl;
```

```
19:
20:    target = Double(target);
21:    cout << "Target: " << target << endl;
22:
23:    target = Double(target);
24:    cout << "Target: " << target << endl;
25:    return 0;
26: }
27:
28: int Double(int target)
29: {
30:    return 2*target;
31: }
```

输出：

```
Enter a number to work with: 20

Target: 40
Target: 80
Target: 160
```

分析：

在第 4 行，函数 Double( )被声明为一个内联函数，它接受一个参数，返回一个 int 值。除在返回类型前加上关键字 inline 外，该声明与其他原型类似。

编译上述代码时，就像在输入下述语句的每个地方输入了 target = 2 * target;　样：

```
target = Double(target);
```

程序执行时，指令已经就位，并被编译为.obj 文件。这避免了在执行程序时跳转和返回的开销，但代价是可执行程序更大。

注意：关键字 inline 是一种提示，它告诉编译器，你希望函数是内联的。编译器可以忽略这种提示，创建真正的函数调用。

5.12.2　递归

函数可以调用自身，这被称为递归。递归可以是直接的或间接的。直接递归是指函数调用自身；而间接递归是指函数调用了另一个函数，而后者又调用了它。

使用递归能轻松地解决一些问题，在这些问题中，通常需要对数据进行某种处理，然后对结果进行相同的处理。两种递归（直接和间接）都有两种可能的结果，一种是最终结束递归并得出答案，另一种是无休止地递归下去，导致运行阶段错误。

需要注意的是，函数调用自身时，系统将在内存创建该函数的一个新拷贝。新拷贝和前一个拷贝中的局部变量彼此独立，它们之间不会彼此直接影响，就像程序清单 5.3 表明的，main( )中的局部变量不能影响被调用的函数中的局部变量一样。

为演示如何使用递归解决问题，请考虑下面 Fibonacci 数列：

```
1,1,2,3,5,8,13,21,34...
```

这个数列中，从第 3 个数开始，每个数都是它前面两个数之和。一个 Fibonacci 问题可能是，计算该数列中第 12 个数的值。

为解决这个问题，必须仔细分析这个数列。前两个数都为 1，以后的每个数均是其前两个数的和。因此，第 7 个数是第 5 个和第 6 个数的和。推而广之，如果 n 大于 2，则第 n 个数是第 n-1 个数和第 n-2 个数之和。

递归函数需要一个结束条件。必须有某件事情导致程序停止递归，否则递归将没完没了。在 Fibonacci

数列中，n<3 是结束条件（也就是说，当 n<3 时，程序可以结束对问题的处理）。

算法是一组解决问题时遵循的步骤。对于 Fibonacci 数列，一种算法如下：

（1）要求用户输入数列中的一个位置。

（2）调用 fib( )函数，并将用户输入的值传给它。

（3）函数 fib( )检查参数 n，如果 n<3，则返回 1；否则 fib( )递归调用自身，并将 n–2 传递给它；然后再次调用自身，并将 n–1 传递给它，最后，返回这两次调用的返回值之和。

如果调用 fib(1)，将返回 1；如果调用 fib(2)，也将返回 1；如果调用 fib(3)，将返回函数调用 fib(1)和 fib(2)的返回值之和。由于 fib(1)和 fib(2)均返回 1，因此 fib(3)返回 2（1+1）。

如果调用 fib(4)，将返回函数调用 fib(3)和 fib(2)的返回值之和。已知 fib(3)返回 2（通过调用 fib(2)和 fib(1)），fib(2)返回 1，因此 fib(4)返回 3，这是数列中的第 4 个数。

再进一步，如果调用 fib(5)，将返回函数调用 fib(4)和 fib(3)的返回值之和。已知 fib(4)返回 3，fib(3)返回 2，因此返回的和为 5。

这种方法并不是解决该问题最有效的方法（对于 fib(20)，将调用函数 fib( )13529 次），但确实可行。请注意，如果用户输入的数过大，内存将被耗尽。每次调用 fib( )时，都需要分配内存；返回时，内存被释放。使用递归时，将不断分配内存，系统内存将很快被耗尽。程序清单 5.10 实现了函数 fib( )。

警告：运行程序清单 5.10 时，请输入一个较小的数（小于 15）。该程序使用了递归，可能会消耗大量内存。

程序清单 5.10　使用 Fibonacci 数列演示递归

```
1:  // Fibonacci series using recursion
2:  #include <iostream>
3:  int fib (int n);
4:
5:  int main()
6:  {
7:
8:     int n, answer;
9:     std::cout << "Enter number to find: ";
10:    std::cin >> n;
11:
12:    std::cout << "\n\n";
13:
14:    answer = fib(n);
15:
16:    std::cout << answer << " is the " << n;
17:       std::cout << "th Fibonacci number\n";
18:    return 0;
19: }
20:
21: int fib (int n)
22: {
23:    std::cout << "Processing fib(" << n << ")... ";
24:
25:    if (n < 3 )
26:    {
27:       std::cout << "Return 1!\n";
28:       return (1);
29:    }
```

```
30:    else
31:    {
32:       std::cout << "Call fib(" << n-2 << ") ";
33:          std::cout << "and fib(" << n-1 << ").\n";
34:       return( fib(n-2) + fib(n-1));
35:    }
36: }
```

输出：

```
Enter number to find: 6

Processing fib(6)... Call fib(4) and fib(5).
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(5)... Call fib(3) and fib(4).
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
Processing fib(4)... Call fib(2) and fib(3).
Processing fib(2)... Return 1!
Processing fib(3)... Call fib(1) and fib(2).
Processing fib(1)... Return 1!
Processing fib(2)... Return 1!
8 is the 6th Fibonacci number
```

注意：有些编译器不能处理在 cout 语句中使用的运算符的情况。如果编译器发出警告，指出第 32 行有问题，请使用括号将减法运算括起，使第 32 和 33 行变成如下所示：

```
std::cout << "Call fib(" << (n-2) << ") ";
     std::cout << "and fib(" << (n-1) << ").\n";
```

分析：

这个程序在第 9 行要求用户指定要计算第几个数的值，并将其赋给 *n*。然后调用函数 fib( )，并将 *n* 作为参数传递给它。程序跳到 fib( )函数处执行，在该函数中，第 23 行打印参数。

第 25 行检测参数 *n* 是否小于 3：如果是，fib( )返回 1；否则，调用 fib(*n*–2)和 fib(*n*–1)，并返回这两个函数调用的返回值之和。

对 fib( )的递归调用结束前，不会返回这些值。为描绘该程序，可将 fib( )调用不断细分，直到 fib( )调用能够直接返回一个值为止。只有调用 fib(2)和 fib(1)可直接返回一个值。这些值向上传递给调用函数，后者将它们相加，得到自己的返回值，然后返回。图 5.4 和图 5.5 说明了对 fib( )的递归调用过程。

在这个例子中，*n* 为 6，因此 main( )调用 fib(6)。程序跳到 fib( )函数处执行。第 25 行检测 *n* 是否小于 3，结果为假，因此 fib(6)在第 34 行调用 fib(5)和 fib(4)，并返回它们的返回值之和。第 34 行的代码如下：

```
return (fib(n-2) + fib(n-1));
```

该返回语句调用 fib(4)（由于 *n*=6，因此 fib(*n*–2)就是 fib(4)）和 fib(5)（fib(*n*–1)）。然后，函数调用 fib(6)处于等待状态，直到这些调用都返回一个值。这些函数调用都返回一个值后，函数调用 fib(6)便可以返回这两个值的和。

由于函数 fib(5)的参数不小于 3，因此再次调用 fib( )，这次使用参数 4 和 3。然后，fib(4)将调用 fib(3)和 fib(2)。

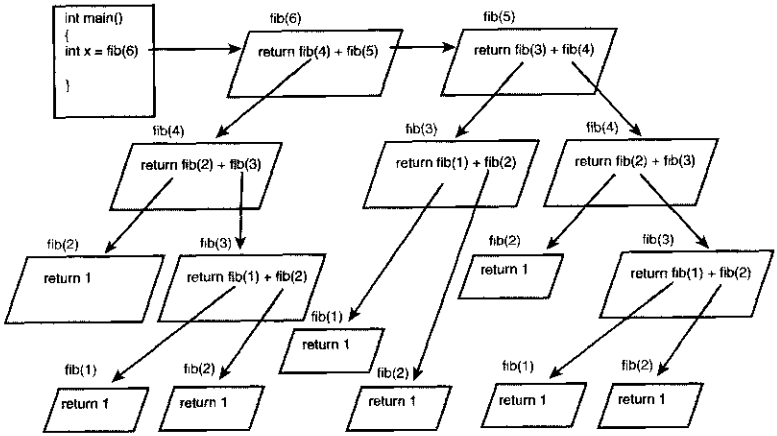

图 5.4　使用递归

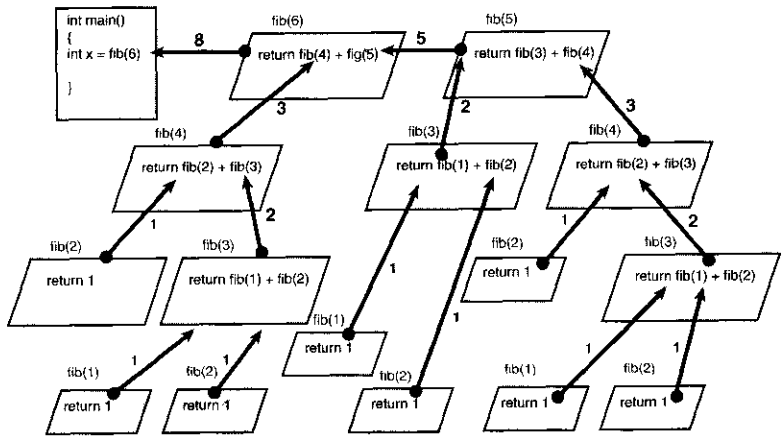

图 5.5　从递归调用返回

前面的输出跟踪了这些调用和返回值。请编译、链接和运行该程序，依次输入 1、2、3，最后输入 6，并仔细观察输出。

这是尝试使用调试器的绝好时机。在第 21 行设一个断点，然后跟踪每次 fib( )调用，每递归调用 fib( )时，观察 *n* 值的变化。

在 C++编程中，并不经常使用递归，但对于某些问题，它是一个方便而有力的工具。

注意：递归是高级编程中比较棘手的部分。之所以在这里介绍它，旨在让读者了解其基本原理。如果读者不能完全理解所有的细节，也不必担心。

5.13　函数的工作原理

调用函数时，跳转到调用函数处，接着传入参数并执行函数体。函数结束时，如果返回类型不为 void，将返回一个值，同时控制权返回到调用函数。

这种任务是如何完成的呢？代码是如何知道应跳转到何处呢？变量传入后被存储在哪里？在函数体中声明的变量又如何呢？返回值如何被传回？代码如何知道该返回哪里？

很多入门书都没有回答这些问题，但不搞清楚这些问题，将发现编程始终有一种神秘感。为回答这些问题，必须简要地介绍计算机内存。

5.13.1　抽象层次

新程序员遇到的一个主要难点是，需要理解众多抽象层次。当然，计算机只是一种电子机器。它们没有窗口和菜单的概念，也没有程序和指令的概念，甚至没有 0 和 1 的概念。实际发生的一切就是在集成电路的各个不同位置测量电压。即使这样描述仍然是一种抽象：电本身是表示亚原子粒子行为的一个抽象概念，而这些粒子本身也是抽象概念。

很少有程序员会不厌其烦地去了解比 RAM 中的值更低层的细节。毕竟，并不需要了解粒子物理学，就可以开汽车、烤面包或打棒球，同样不必了解计算机的电子构造也可以编程。

然而，确实需要了解内存是如何组织的。如果在创建变量时对于变量位于何处以及值如何在函数间传递没有清楚的了解，那么编程仍然是一件无法控制的神秘事情。

5.13.2　划分 RAM

程序启动时，操作系统（如 DOS、UNIX/Linux 或 Microsoft Windows）将依据编译器的需求设置各种内存区域。作为 C++程序员，经常需要关心的是全局名称空间、自由存储、寄存器、代码空间和堆栈。

全局变量都放在全局名称空间中。全局名称空间和自由存储将在接下来的几章详细介绍，这里将重点放在寄存器、代码空间和堆栈上。

寄存器是中央处理器（CPU）中的一个特殊存储区域。它们负责进行 CPU 内部事务处理。寄存器的具体工作原理超出了本书的范围，但读者应该知道的是在任意给定时刻指向下一行代码的寄存器组。这些寄存器被统称为指令指针。指令指针的任务跟踪接下来将执行哪行代码。

代码本身位于代码空间中，后者是分配用于存储程序中指令的二进制形式的那部分内存。每行源代码都被转换为一系列指令，每条指令都位于内存中的某个地址处。指令指针中存放了接下来要执行的指令的地址。图 5.6 说明了这种概念。

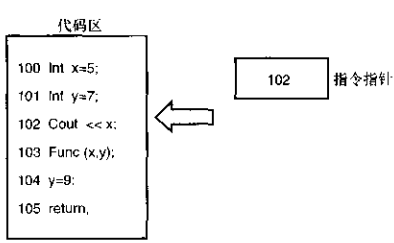

图 5.6　指令指针

堆栈是为程序分配的一块特殊内存区域，用来存储程序中每个函数所需的数据。之所以称为堆栈，是因为它是后进先出的，就像咖啡馆中堆放盘子的架子一样，如图 5.7 所示。

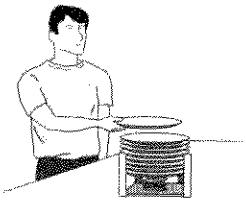

图 5.7 堆栈

后进先出意味着最后被加入到堆栈中的东西首先被取出。这不同于大多数队列，后者是先进先出的，就像剧院售票处前的长队，谁先来谁先走。堆栈像一堆硬币：如果在桌上堆放 10 枚硬币，然后再取走一些，则最后放进去的 3 枚硬币就是最先被取走的 3 枚硬币。

数据被压入堆栈后，堆栈将增大；数据被弹出堆栈后，堆栈将缩小。像放盘子的架子一样，上面的盘子没拿走之前，无法拿走下面的盘子。

盘子架是一种常见的类比。就当前情形而言，这种类比是正确的，但从基本原理上说，这是错误的。一种更准确的类比是，堆栈像一排上下对齐的方盒子。栈顶是堆栈指针（另一个寄存器）指向的方盒子。

每个方盒子都有一个顺序排列的地址，其中有一个地址保存在堆栈指针寄存器中，这个地址称为堆顶，其下的所有数据都被认为是存放于堆栈中；其上的所有数据都被认为在堆栈外，因而无效。图 5.8 说明了这一点。

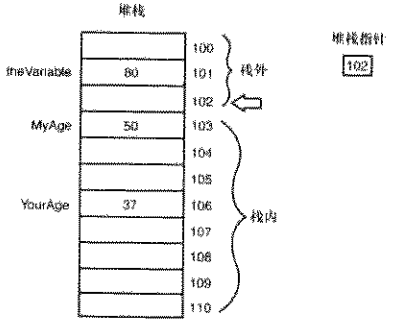

图 5.8 堆栈指针

数据被压入堆栈时，放在堆栈指针上面的方盒子中，然后堆栈指针上移，指向新数据。当数据弹出堆栈时，其效果只是堆栈指针沿堆栈下移。图 5.9 说明了这种规则。

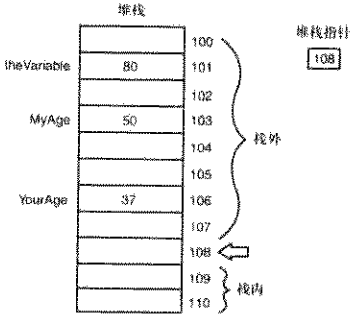

图 5.9 移动堆栈指针

位于栈指针以上（栈外）的数据在任何时候都可能变化，也可能不变，这些值被称为垃圾，因为它们的值是不可靠的。

5.13.3　堆栈和函数

以下是程序跳转到函数处执行时发生的大致情况（具体细节随操作系统和编译器而异）：

（1）指令指针中的地址值增加 1，指向函数调用后的下一条指令。这个地址随后被放入堆栈，它是函数返回时的返回地址。

（2）在堆栈中为声明的返回值类型分配空间。在 int 变量占两字节的系统上，如果返回类型被声明为 int，堆栈再增加两个字节，但在这个两字节中不存放任何值（这意味着这两个字节中的“垃圾”保持不变，直到本地变量被初始化）。

（3）被调函数的地址存储在为此而分配的一块特殊内存区域中，这个地址被加载到指令指针中，这样将执行的下一条指令为被调用的函数。

（4）当前的栈顶被记录下来，并存入一个称为栈帧（stack frame）的特殊指针中。从现在开始到函数返回前被加入到堆栈中的任何数据都将被视为函数的局部数据。

（5）函数的所有参数都被放入堆栈。

（6）现在执行指令指针中的指令，即执行函数的第一条指令。

（7）局部变量在被定义时被压入堆栈。

当函数准备好返回时，返回值被放入第 2 步预留的堆栈区域中。随后不断对堆栈执行弹出操作，直接遇到栈帧指针，这相当于丢弃函数的所有局部变量和参数。

返回值被弹出堆栈，将其作为函数调用本身的值。然后检索第 1 步存储的地址，将其放入指令指针。程序回到函数调用后执行，并检索函数调用的返回值。

上述过程中的一些细节可能随计算机操作系统、编译器和处理器而异，但基本思想是一致的。一般而言，调用函数时，返回地址和参数将被压入堆栈。在函数执行期间，局部变量也被压入堆栈。函数返回时，这些值都从堆栈中弹出，从而被删除。

接下来的几章将介绍内存的其他区域，这些区域用来存储那些生命周期比函数更长的数据。

5.14　小　　结

本章介绍了函数。函数实际上是可以转入参数和返回一个值的子程序。每个 C++程序都从 main( )函数开始执行，而 main( )函数可以调用其他函数。

函数是使用函数原型声明的，原型描述了返回值、函数名和参数类型。函数可被声明为内联的。在函数原型中，还可以为一个或多个参数声明默认值。

函数定义必须在返回类型、函数名和参数列表方面与原型一致。通过改变参数数目或类型，可以重载函数；编译器将根据参数列表找到正确的函数。

在函数中声明的变量以及传入函数的参数的作用域都是其声明所在的语句块。按值传递的参数只是拷贝，不会影响调用函数中变量的值。

5.15　问　与　答

问：为什么不将所有变量都声明为全局的？

答：在编程中，一度是这样做的。但随着程序越来越复杂，这样做就使得很难发现程序中的错误，因为任何一个函数都可能破坏数据——在程序中的任何地方都可以修改全局变量的值。多年的编程经验使程序员们相信，数据应该尽量局部化，可修改数据的地方也应尽可能小。

问：什么情况下应在函数原型中使用关键字 inline？

答：如果函数很小，只有一两行，且不会在程序中的很多地方调用它，则可以将其声明为内联的。

问：为什么函数参数值的变化不会反映到调用函数中？

答：传递给函数的参数是按值传递的。这意味着函数中的参数只是原始值的一个拷贝。这种概念在“函数的工作原理”一节中进行了详细介绍。

问：如果参数是按值传递的，要将参数值的变化反映到调用函数中该如何办？

答：第 8 章将讨论指针，而第 9 章将讨论引用。使用指针和引用可解决这种问题，同时还可突破函数只能返回一个值的限制。

问：如果有以下两个函数，将发生什么情况？

```
int Area (int width, int length = 1); int Area (int size);
```

它们是重载吗？这里虽然参数个数不同，但第一个函数指定了默认值。

答：这两个声明可以通过编译，但使用一个参数来调用函数 Area 时，将发生错误：无法判断应使用 Area (int, int) 还是 Area (int)。

5.16 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

5.16.1 测验

1. 函数原型和函数定义之间有何区别？
2. 参数的名称在函数原型、函数定义和函数调用中必须一致吗？
3. 如果函数不返回任何值，如何声明它？
4. 如果没有声明返回值，默认返回类型是什么？
5. 什么是局部变量？
6. 什么是作用域？
7. 什么是递归？
8. 什么时候应该使用全局变量？
9. 什么是函数重载？

5.16.2 练习

1. 编写一个名为 Perimeter( )的函数的原型，它接受两个 unsigned short int 参数，返回类型为 unsigned long int。

2. 编写练习 1 中函数 Perimeter( )的定义。两个参数表示矩形的长和宽，函数返回周长（长与宽之和两倍）。

3. 查错：下述代码中的函数有什么错误？

```
#include <iostream>
void myFunc(unsigned short int x);
int main()
{
      unsigned short int x, y;
      y = myFunc(int);
      std::cout << "x: " << x << " y: " << y << "\n";
      return 0;
```

```
}

void myFunc(unsigned short int x)
{
      return (4*x);
}
```

4. 查错：下述代码中的函数有什么错误？

```
#include <iostream>
int myFunc(unsigned short int x);
int main()
{
      unsigned short int x, y;
      x = 7;
      y = myFunc(x);
      std::cout << "x: " << x << " y: " << y << "\n";
      return 0;
}

int myFunc(unsigned short int x);
{
      return (4*x);
}
```

5. 编写一个函数，它接受两个 unsigned short 参数，返回第 1 个数与第 2 个数的商。如果第 2 个数为 0，则不执行除法，并返回 1。

6. 编写一个程序，请用户输入两个数，然后调用练习 5 的函数。打印结果；如果返回值为-1，打印错误消息。

7. 编写一个程序，要求用户输入一个数和一个指数。编写一个递归函数，接受这两个数作为参数，并计算幂运算结果。也就是说，如果数为 2，指数为 4，该函数返回 16。