*第五章 得力助手—— 函数*

第一节：函数与结构化程序设计

第二节：函数的声明和定义

第三节：函数的参数传递

第四节：变量的作用域

第五节：函数的嵌套和递归调用

第六节：多个文件的程序

第七节：函数的返回值

我们不是第一次接触到“函数（**function**）”这个词了，前面几章的内容里，我们已经接触过了很多的函数，但是我们只是会用，并不知道函数到底是什么东西。在我们看来，函数就像一个助手一样，当我们需要完成一件事情的时候，只需要找来对应的“助手“，然后叫它们去做这件事，而不是我们亲自动手去做。比如，在输入数据的时候，我们用scanf函数，输出的时候则用printf函数，我们给scanf或printf下达指令，让它们做输入或者输出的工作，而我们则不必关心它到底是怎么做到的。

5.1 函数与结构化程序设计

操作系统是一个很庞大的软件，要编写或维护一个操作系统是很难的，但是我们很容易看出，再大的操作系统也包含很多相对小得多的程序或模块，就Windows而言，查看文件的时候是explorer.exe在起作用，而关机的时候则需要用到shutdown.exe。Windows这样庞大的操作系统，任何人都不可能单独完成，所以它就被分成了很多个不同的模块，将这些模块交给不同的人来分别编写，最后统一调试运行。

C语言程序中也用到了类似的思想，为了提高开发效率，往往需要将程序拆分成不同功能的模块，每个模块再交给不同的人来编写，这样的模块就是函数。函数的优点之一就是降低了程序的复杂程度，增强了程序的可读性和可维护性。此外，函数还有一个好处就是省略重复的操作的代码，printf的输出其实是很复杂的，试想，如果我们每次输出都要写一大堆代码来实现printf的功能（如果我们有这个能力的话），那么我们的程序将会长得可怕。理解函数的使用方式对以后的学习有着不可或缺的作用。

函数是多种多样的，按照不同的角度有不同的分类方法，关于函数的分类，没必要死记硬背，只要对函数本身理解到位，对它的分类的理解也就水到渠成了。到目前为止，我们知道了第一种分类：系统与标准库提供的函数、自定义的函数。

5.2 函数的声明和定义

前面几章出现过很多的由系统或C语言标准所提供的函数，我们在调用这些函数的时候，都需要在它们的名字后面跟上了一对括号，而括号里面可能会写一些有特定意义的东西，也可能什么都没有，括号里的这些东西被称为参数（**parameter**）。那么，函数就又有了一种分类方式——有参数的函数、无参数的函数。

第1章我们曾提到过“返回值”这个词。一个助手除了可以帮我们完成一件事情以外，还可以在完成这件事之后给我们带回一点东西来，比如我们叫助手去买机票，那么助手就会带一张机票回来给我们。同样地，一个函数除了可以完成一定的任务，还可以反馈一点东西给我们，这就是它的返回值。当然，助手可以什么也不带回来，只需要完成他的工作就行了。函数也是如此，可以没有返回值。这样，函数又有了第三种分类方式——有返回值的函数和无返回值的参数。

接下来我们讲一讲C语言程序是怎样从一堆计算机看不懂的代码编程二进制的可执行文件的。

第一步，预编译，还记得第1章提到的这个概念吗？我们知道#include和#define都是预编译指令，那么它们的作用到底是什么呢？如果编译器在预编译阶段遇到了#include，那么它就会尝试去打开对应的文件，我们现在见到的stdio.h等头文件都是由一对尖括号括起来的，其实用双引号也是可以的，并且对于我们自己编写的头文件，只能用双引号来包含。C语言中有个约定俗成的规定：凡是标准库中的头文件一律用尖括号来包含，因为这样能够加快预编译的速度，编译器见到尖括号就直接去标准库的目录下寻找对应的头文件，而不是从当前文件夹开始寻找。如果编译器遇到了#define，编译器则会在源文件中寻找所有利用该指令定义的常量，并把它替换掉，也就是说，如果有这样一段代码：

#define N 10

int x=N;

那么在预编译阶段之后，实际上程序会变成这样：int x=10;。

第二步，编译，这一阶段就会直接把预编译之后的源文件“翻译”成一堆二进制的文件，但是这时候还是不可以执行的。

第三步，连接。上一阶段产生的二进制文件之所以不能运行，是因为它们还没有组合成一个整体。连接的作用就是把各个模块（每个模块含有不同的函数）以及使用到的标准库中的函数（如printf函数等）组合起来，形成一个可执行文件。

铺垫很长，而且C语言的生成过程似乎与我们要讲的函数的声明和定义毫无关系。其实不然，了解C语言程序生成过程有助于我们理解函数的声明和定义。

函数调用的原则是：先声明，再调用。我们必须在使用一个函数之前声明它。

函数声明的格式如下：

返回值类型 函数名(参数列表);

别忘了在声明的时候需要加上分号。

1. #include <stdio.h>
2. int function();
3. int main()
4. {
5. function();//调用函数的时候，一定要在函数名后面加一对括号
6. return 0;
7. }

程序5-2-1

如果你发现这个程序不能运行，别担心，这是正常的。程序5-2-1中，声明了一个返回值为int类型，名叫function的函数。然后第5行调用了这个函数，由于先声明，后调用的原则是满足的，所以能够通过编译。但是当到了连接阶段的时候，连接器试图寻找到function函数所对应的二进制代码，但是却发现怎么找都找不到（因为我们根本没写），于是最终的二进制文件就不能成功生成，程序也就运行不起来了。

光有声明是不够的，我们还要给它具体的定义，不然程序就不能成功运行。在定义程序的时候，要保证与声明函数的时候的返回值类型、参数列表一致。

1. #include <stdio.h>
2. int function();
3. int main()
4. {
5. int num;
6. num=function();
7. printf("%d",num);
8. getchar();
9. return 0;
10. }
11. int function()
12. {
13. printf("这是function函数的定义\n");
14. return 1;
15. }

程序5-2-2

和程序5-2-1一样，程序5-2-2在第2行声明了一个函数。此外，程序5-2-2还在第11行对这个函数进行了定义，我们可以看出，这个函数的作用是输出一句话，然后用1作为返回值。主函数中，用了一个int型变量num来接受这个返回值，然后输出它（当然，也可以不用接受它）。

一个C语言程序中，可以有很多个函数，但是不管怎样，程序执行的顺序一定是从主函数开始的，即使主函数被放到了最后面。下面这个程序将向我们揭示C语言程序的执行顺序。

1. #include <stdio.h>
2. int function()
3. {
4. printf("函数调用中\n");
5. return 1;
6. }
7. int main()
8. {
9. printf("函数调用前\n");
10. function();
11. printf("函数调用后\n");
12. getchar();
13. return 0;
14. }

程序5-2-3

程序5-2-3看起来不长，但是分量确实比较重的。这次我们不再是“先声明，后调用”了，而是直接声明，这样的写法是完全合理的，因为定义的同时也相当于声明了。第10行我们直接调用了function函数，而没有用任何变量来接受它的返回值，这种做法依然是合理的。程序最终运行结果如下：

函数调用前

函数调用中

函数调用后

可以看出，程序运行的顺序是第9行到第10行，再到第4行，再到第11行。这证明了不管主函数是不是放在最前面，总是从它开始执行，这也就是为什么，一个程序有且只能有一个主函数。

此外，程序执行到第10行之后，没有继续执行第11行，而是转而去执行function函数中的相关内容，这说明，当主函数中调用了一个函数时，程序会立刻跳转到这个函数，直到执行完毕，再返回到主函数中继续执行（遇到return语句就意味着一个函数将要结束执行并返回）。其实不止是在主函数中，在任何一个函数中，如果这个函数又调用了一个函数，那么程序就会跳转到目标函数，直到执行完毕，再返回并继续执行。

我们接下来将用函数来改写程序4-2-4，并进一步讲解有关函数的一些细节问题。

1. #include <stdio.h>
2. #include <conio.h>
3. #include <windows.h>
4. #include <tchar.h>
5. void webOpener(char target);
6. int main()
7. {
8. char choice;
9. printf("==========================\n");
10. printf("请选择打开哪个网站\n");
11. printf("1.百度(www.baidu.com)\n");
12. printf("2.微博(www.weibo.com)\n");
13. printf("3.优酷(www.youku.com)\n");
14. printf("==========================\n");
15. do
16. {
17. choice = \_getch();
18. } while ((choice<'1') || (choice>'3'));
19. webOpener(choice);
20. return 0;
21. }
22. void webOpener(char target)
23. {
24. switch (target)
25. {
26. case '1':
27. {
28. ShellExecute(NULL, \_T("open"),
29. \_T("www.baidu.com"), NULL, NULL, SW\_SHOWNORMAL);
30. break;
31. }
32. case '2':
33. {
34. ShellExecute(NULL, \_T("open"),
35. \_T("www.weibo.com"), NULL, NULL, SW\_SHOWNORMAL);
36. break;
37. }
38. case '3':
39. {
40. ShellExecute(NULL, \_T("open"),
41. \_T("www.youku.com"), NULL, NULL, SW\_SHOWNORMAL);
42. break;
43. }
44. }
45. return;
46. }

程序5-2-4

程序5-2-4和程序4-2-4的功能是完全一致的，这个程序中，有几个地方需要加以注意。第一就是第5行的void，void的意思是空，用void作为返回值类型意味着该函数不会有也不能有任何返回值。webOpener有一个参数，这个参数是char类型的，这代表着我们在调用这个函数的时候，需要在括号里给一个char型的数据，这个数据可以像第19行那样是一个变量，也可以是一个常量，比如我们完全可以把程序第19行改成:webOpener('1');这样，target的值就恒为'1'（注意'1'和1是有区别的），打开的网站也就一直都是百度了。注意到第45行的return语句，return后面没有任何内容，这是因为函数的返回值为void，这个return实际上是可以省略的。当且仅当返回值为void类型的时候，return可以省去（实际上，主函数中的return也是可以省去的，即使主函数的返回值是int，但是主函数只是一个特例）。

**小提示**

关于函数的参数，有这么两个概念：形参和实参，所谓形参，就是指函数在声明或定义时给出的参数，比如程序5-2-4中的webOpener中的target，形参用于标明参数的类型，形参看起来是一个变量。而程序5-2-4的第19行中，choice则是该函数的实参，实参不一定是变量，实参可以是任何与形参类型相同的数据，常量、变量、甚至其它函数的返回值。

通常情况下，函数的参数个数是固定的，并且在调用函数的时候，形参的类型及数量甚至顺序都必须与形参相对应。有一种比较特殊的函数，叫做可变长参数函数，这种函数的参数个数是不固定的，典型的，scanf函数和printf函数。由于可变长参数函数牵扯到更深的知识，为降低学习难度，本书对其不做具体介绍。

5.3 函数的参数传递

我们说，参数传递，必须保证形参与实参的的类型、数量、顺序都相同，后面两个条件是非常严格的，而参数的类型的限制则相对宽松许多，因为类型之间可以相互转化。

1. #include <stdio.h>
2. void fun(int,double);
3. int main()
4. {
5. fun(10.1,10);
6. getchar();
7. return 0;
8. }
9. void fun(int x,double y)
10. {
11. printf("%d\t%.1f",x,y);
12. }

程序5-3-1

程序第2行，我们在声明函数fun的时候，只是给出了参数的类型，而没有给出参数的名字，在函数声明的时候，这样写是合理的，但是，在函数定义的时候就不能这样写，因为如果在定义的时候参数没有名字，参数就无法被访问，设置参数就没有了意义。程序5-2-4中，webOpener函数在声明和定义的时候，参数的名字一致，实际上是不必要这么做的，只需要保证类型和顺序相同即可。

程序第5行，我们调用了fun函数，并且用10.1和10作为实参，容易看出，我们给出的参数类型是double和int，但是fun函数要求的却是int和double。我们知道，将int型数据赋值给double以及将double型数据赋值给int都是合理的，所以这样的写法也就是合理的了。

第11行的printf语句中，\t表示一个转义字符Tab，也就是制表符，制表符的长度与显示它的软件有关，一般是4到6个空格的长度。%.1f表示输出的数据保留1位小数，这是第2章有所提及的。

关于函数的形参和实参，还有一个容易犯错的地方。

1. #include <stdio.h>
2. void fun(int);
3. int main()
4. {
5. int x = 10;
6. fun(x);
7. printf("%d",x);
8. getchar();
9. return 0;
10. }
11. void fun(int x)
12. {
13. x++;
14. }

程序5-3-2

很多初学者会认为上面这段程序的运行结果是11，其实正确的结果是10。看到这里，读者可能不解，为什么会是10呢，第13行不是有x++吗？这是因为，fun函数中的x和主函数中的x不是同一个变量，它们只是数值上相等而已，所以在fun中对x的操作自然就不会影响到主函数中的x了。

变量是保存在内存中的，而内存里面有个概念就是地址，一个变量从创建到消亡这段时间，它的地址是不会发生变化的，因此只要我们能证明程序5-3-2中的两个x在内存中有不同的地址，就能证明它们是两个不同的变量。

1. #include <stdio.h>
2. void fun(int);
3. int main()
4. {
5. int x = 10;
6. printf("%p\n",&x);
7. fun(x);
8. getchar();
9. return 0;
10. }
11. void fun(int x)
12. {
13. printf("%p\n", &x);
14. }

程序5-3-3

%p是printf语句中的格式控制字符的一种，表示将一个数据按照地址的格式来输出，而&则是取地址操作符，&x表示取到x的地址。因此，第6行和第13行的意思就是，取到x的地址，并把这个值按地址的格式输出。虽然程序5-3-3的最终运行结果是无法预计的，但是至少有一点可以肯定，那就是第6行和第13行输出的数据肯定不同，读者可亲自上机感受。

形参不一定全是变量，也可以出现常量，如果出现了常量的参数，那么函数就不能对这个参数做修改。

1. include <stdio.h>
2. void fun(const int);
3. int main()
4. {
5. int x = 10;
6. fun(x);
7. getchar();
8. return 0;
9. }
10. void fun(const int x)
11. {
12. printf("%d",x);
13. }

程序5-3-4

程序5-3-4展示了常量作为函数参数的做法，如果我们试图在第11行和第12行之间插入一行对x赋值的语句，程序将会不能成功运行。此外，函数的返回值也是可以用const限定符来修饰的，只不过在绝大多数情况下，这样做是毫无意义的，只有在函数返回值是一个指针（指针是后面章节的内容）的时候，才有可能会这样做。

函数的调用是有一定代价的，程序需要在内存的一段代码中跳转到另外一段代码，并且，任何一个函数在执行的时候都会被分配一定的内存空间。对于一些功能简单的函数，我们可以通过一些手段来节约这一点函数调用的开销，而这个手段就是#define预编译指令。

对于#define指令，我们早已不再陌生，不过之前我们眼里的#define都是用来定义常量的，如果读者以为它的功能仅限于此的话，那么就太小看这个功能强大的工具了。5.2节我们说过，#define指令在“定义”常量的时候并不是像定义用const修饰的常量那样开辟一段内存空间来储存数据，而是在预编译阶段进行文本的替换，虽然我们眼里看起来和常量没什么区别，实际上在编译器看来是有本质区别的。那么，既然#define能够用于将一些符号替换成数字，它也能实现将符号替换成代码。我们说预编译阶段是发生在编译之前的，所以#define实现的函数实际上就只是一些简单的语句罢了，并没有被编译器单独划成一个模块，因此，在执行用#define实现的“函数”时就不会有函数调用的开销。

1. #include <stdio.h>
2. #define PRODUCT(a,b) a\*b
3. int main()
4. {
5. printf("%d", PRODUCT (2,3));
6. getchar();
7. return 0;
8. }

程序5-3-5

程序5-3-5中#define指令创建了一个“函数”，名叫SUM，不难看出，在预编译阶段之后，第5行的PRODUCT(2,3)会被替换成2\*3，看起来就像是一个求乘积函数一样。

不过，程序5-3-5还有一个致命的漏洞，这个漏洞的产生和#define本身有关。还是因为#define只是进行简单的文本替换，这会导致一些很微妙的问题。读者可尝试将第5行改成这样：printf("%d",PRODUCT(2+2,3+3));。看起来程序的运行结果应该是24，实际上是11，其实这也不难理解，因为PRODUCT并不是一个真正的函数，它只是简单的文本替换，所以刚才那段代码实际上会变成这样：printf("%d",2+2\*3+3);。由于\*的优先级高于+，所以最终的运行结果会变成11。解决方法是把第2行改写成这样：

#define PRODUCT(a,b) (a)\*(b)

再次分析不看看出，PRODUCT(2+2,3+3)会被替换成(2+2)\*(3+3)，程序运行结果也就是24了。

此外，#define语句可能会占用多行，这时候就要用一个符号来表示#define需要扩展到下一行，这个符号是\。

1. #include <stdio.h>
2. #define MAX(a,b) if ((a) > (b))\
3. {\
4. printf("%d", a);\
5. }\
6. else\
7. {\
8. printf("%d", b);\
9. }
10. int main()
11. {
12. MAX(10, 9);
13. getchar();
14. return 0;
15. }

程序5-3-6

上面这段程序看起来怪怪的，不过如果我们把所有的\都忽视掉，它就是一个简单的if-else语句。注意到第9行的花括号后面没有\，这是因为它是最后一行，不再需要扩展到下一行，所以也就不能再加\。第12行的MAX(10, 9);理所当然地会被替换一个条件语句。细心的读者注意到了第12行的分号，实际上，这个分号是完全可以省去的，只是为了代码的美观，我们依然加上了分号，它并不会造成什么影响。

#define指令的功能是十分强大的，凡是C语言和C++的开发，必然会有它的身影，一方面它给我们带来巨大的便利，另一方面它又会带来一些潜在的威胁，所以Java、C#等继承自C语言和C++的高级语言干脆就删除了这个功能。对于初学者来说，一定不要因为它容易导致错误而拒绝使用它，只有多犯错并且多思考多总结，才能真正地掌握好一门语言。

5.4 变量的作用域

在第4章的时候，我就简单提到过关于变量的作用域的概念，由于当时我们没有足够的知识储备，因此并没有做更深入的介绍。我们将在这一节系统地介绍有关于变量作用域的知识。

首先我们要介绍的就是局部变量。局部变量又被称为自动变量，指得是那些存在于语句块中并且没有任何修饰的变量。当一个局部变量被声明时，系统自动为它分配内存，当它的作用域结束时，系统又会自动回收它的内存空间。

局部变量只在它所在的语句块中有效，并且只在声明它之后的语句有效。

1. #include <stdio.h>
2. int main()
3. {
4. int num1;
5. printf("请输入一个整数：\n");
6. scanf("%d", &num1);
7. if (num1 != 0)
8. {
9. int num2 = num1;
10. printf("你输入的数字是%d\n",num2);
11. }
12. {
13. int num1 = 0;
14. }
15. printf("%d",num1);
16. rewind(stdin);
17. getchar();
18. return 0;
19. }

程序5-4-1

上面这段程序没有什么实用价值，但是它能向我们揭示局部变量的一些特点。第4行声明了一个int型变量num1，它的作用域是第4行到主函数结束。第9行又声明了一个int型变量num2，它的作用域就只有第9行和第10行了。

第13行到第15行看起来有点怪，只是有一对花括号，也没有if-else或者while什么的。其实，只要是由一对花括号括起来的区域都算是一个语句块，而第14行声明的num1变量的作用域也就只有这一行了。注意，这里的num1和前面的num1重名了，我们说C语言的标识符是不能重名的，但是为什么这里可以呢，原因是，当两个变量的作用域在同一语句块时，系统将会不知道我们到底要访问哪一个变量。而像程序5-4-7里面这样的，一个num1作用域是整个主函数，一个则只有一行，程序将会很容易地根据num1使用的地点判断到底我们使用的是哪一个num1。这就好比，有一个名人叫张三，我们圈子里也有一个叫张三的人，当圈子外的人谈论张三时，它们谈论的那位名人，而我们圈子里的人谈起张三时，一般认为是在讨论我们的朋友。

局部变量的作用域弄清楚了，接下来我们将要探索局部变量的生命周期。

1. #include <stdio.h>
2. void fun(int x);
3. int main()
4. {
5. fun(2);
6. fun(1);
7. getchar();
8. return 0;
9. }
10. void fun(int x)
11. {
12. int i;
13. if (x % 2 == 0)
14. {
15. i = 10;
16. printf("%d\n", i);
17. return;
18. }
19. if (i == 10)
20. {
21. printf("i=10\n");
22. }
23. }

程序5-4-2

如果你在运行程序5-4-2的时候发现它崩溃了，别担心，这是正常的。在主函数中，我们调用了两次fun函数，给的参数一次是偶数，一次是奇数，从fun函数中我们容易看出，当x为偶数的时候，i会被赋值为10，然后输出i，之后返回。（在第17写一个return语句能够让程序运行到这里的时候提前返回，而不再执行之后的代码，这是函数返回值的使用技巧之一）当x为奇数的时候，程序将会判断i是否为10，如果是则输出提示信息，但是运行到这个地方的时候程序明显崩溃了，并提示说我们使用了未初始化的变量i。也就是说，在第一次调用fun的时候，虽然i被赋值为10，但是这并不代表它的值能保留到下一次调用，由于第一次调用结束之后，i就变得不再可用，它所占的内存也就被回收了，等到第2次调用的时候，系统又会再次为i分配内存空间。当然，理论上讲，两次函数调用i可能被分配到同一片内存，但是不可否认，第一次调用i和第二次调用i之间，i经历了一个“销毁”的过程。

说完了局部变量，我们再讲解一下全局变量。所谓全局变量，一定是在函数外定义的变量，它的作用域是从它声明开始之后的代码，它随程序开始运行而产生，随程序结束而消亡。当全局变量与局部变量发生命名冲突时，优先被访问的是局部变量。

1. #include <stdio.h>
2. int num = 10;
3. void fun1();
4. void fun2();
5. int main()
6. {
7. fun1();
8. fun2();
9. getchar();
10. return 0;
11. }
12. void fun1()
13. {
14. printf("函数1，num的值为%d\n",num);
15. num++;
16. }
17. void fun2()
18. {
19. printf("函数2，num的值为%d\n", num);
20. }

程序5-4-3

程序5-4-3演示了全局变量的用法，我们在第2行声明了一个全局变量num，并且将它初始化为10,。主函数中先后调用了fun1和fun2函数，我们可以看出，fun1和fun2都可以直接访问num，即使这两个函数中没有声明相关的变量，同时，fun1中对num进行了自增操作，我们可以从运行结果中看出，fun1函数的操作确实会影响到num的值，从而影响到fun2中产生的结果。

**试一试**

如果把第2行改成;int num;，num的值会是多少，请亲自编写程序上机测试。

在介绍全局变量之前，我们一直没有办法解决函数之间的数据交流的问题，如程序5-3-2中那样。在C语言中，解决这个问题的最佳途径是通过指针，但是我们目前还没有相关的知识，所以我们可以考虑用全局变量。我们可以把需要被多个函数共同处理的数据声明成全局变量，这样同一个数据就能被不同的函数访问，并且产生的结果会影响到其它的函数。

接下来，我们将使用全局变量来编写一个猜拳的游戏。对这个游戏做需求分析，它大概分成这样几个模块：菜单显示模块，控制整个程序的流程，并显示菜单；随机数产生模块，用于让计算机产生猜测的结果；用户输入模块，用于接受用户猜测的结果并判断是否合理；判断模块，用于判定双方的胜负并结算。

第一个菜单显示模块可以由主函数来负责，其它三个模块则分别需要编写一个对应的函数。

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <conio.h>
4. #include <time.h>
5. char guessPlayer, guessAI;
6. int money=500, ante;
7. void playerResult();
8. void AIResult();
9. void gameBalance();
10. int main()
11. {
12. char choice;
13. while (1)
14. {
15. printf("请选择操作\n");
16. printf("==============\n");
17. printf("1.进行游戏\n");
18. printf("2.退出游戏\n");
19. printf("==============\n");
20. do
21. {
22. choice = \_getch();
23. } while ( (choice<'1') || (choice>'2') );//输入结果不对则一直等待输入
24. switch (choice)
25. {
26. case '1': //进行游戏的分支，分别调用3个相关的函数
27. {
28. playerResult();
29. AIResult();
30. gameBalance();
31. break;
32. }
33. case '2':
34. {
35. system("cls");
36. printf("感谢使用\n");
37. \_getch();
38. exit(0);//这个函数可以直接退出程序
39. }
40. }
41. system("cls");//清空屏幕，为了更美观
42. }
43. }
44. void playerResult()
45. {
46. printf("请出招（1.石头 2.剪刀 3.布）\n");
47. do
48. {
49. guessPlayer = \_getch();
50. } while ((guessPlayer<'1') || (guessPlayer>'3'));
51. ante = -1;//将赌注设为-1，防止上一局游戏的干扰
52. while(1)
53. {
54. rewind(stdin);
55. printf("请下注，（您当前的财产为%d）\n", money);
56. scanf("%d", &ante);
57. if (ante >= 0 && ante <= money)
58. {
59. break;
60. }
61. }
62. }
63. void AIResult()
64. {
65. srand((unsigned)time(NULL));
66. int temp = rand() % 3 + 1;//产生随机数，结果在1~3之间
67. guessAI = temp + '0'; //将随机数转换为字符
68. }
69. void gameBalance()
70. {
71. int result = guessPlayer - guessAI;
72. switch (result)
73. {
74. case 0:
75. {
76. printf("平局，您当前的资产为：%d", money);
77. \_getch();
78. break;
79. }
80. case 1://石头克剪刀，剪刀克布
81. case -2: //布克石头
82. {
83. money += ante;
84. printf("您获胜了，您当前的资产为：%d", money);
85. \_getch();
86. break;
87. }
88. default:
89. {
90. money -= ante;
91. printf("您输了，您当前的资产为：%d\n", money);
92. if (money <= 0)
93. {
94. printf("您破产了，请下次再来");
95. \_getch();
96. exit(0);
97. }
98. \_getch();
99. break;
100. }
101. }
102. }

程序5-4-4

上面这个程序真是太长了，不过正是因为它长，所以才有讲解的价值，因为长才能体现分模块编程对C语言编程的重要性。其实如果我们一个模块一个模块地讲解，上面这个程序是非常简单的。

首先，前4行是头文件的包含，stdio.h自然不用说，conio.h提供\_getch函数，而stdlib.h和time.h则分别在其它的地方提供支持，至于到底是什么，后面的分析会有所提及。

第5行声明了两个全局变量，分别用于保存玩家和AI（**Artificial Intelligence**，人工智能）玩家的“招式”，所谓招式，必须被限定在石头、剪刀、布的范围内，我们用字符的'1'、'2'、'3'来分别表示这三种招式。第5行声明了两个全局变量，分别表示玩家的资产和他在一局游戏中所下的赌注。第7到9行则是3个分模块函数的声明。

第12行的局部变量choice用来接受用户选择操作时的输入结果。第13到16行是一个大的循环，我们称之为“游戏循环”。游戏循环中，首先是显示菜单，然后接受用户的输入，如果用户输入的结果不在'1'到'2'之间，则程序不会有任何响应，直到用户输入正确的选项为止。接下来的switch语句对用户输入的结果进行判断，如果是'1'则调用相关函数进行游戏，如果是'2'则在输出提示信息并等待用户确认之后，退出程序。exit是一个函数，它的作用是直接终止整个程序，括号内的参数表示程序的返回值，我们在括号里写0表示正常退出，相当于就是return 0了，注意到主函数的最后是没有return语句的，这个函数由stdlib.h提供。system(“cls”)的作用是清空当前屏幕，system函数是一个功能很强大函数，它允许我们直接调用一些系统命令，这些命令需要用双引号包含起来，cls是ClearScreen（清屏）的缩写。我们说C语言是大小写敏感的，但是稍微有经验的人都知道，Windows系统是大小写不敏感的，因此，在调用system命令的时候，我们不必太关心命令的大小写问题。

**小提示**

我们知道，Windows操作系统下有一个cmd.exe的程序，通过它我们可以在命令行下执行一些操作，C语言程序中，通过system函数所发出的命令其实都是调用这个程序来完成的，所以一切合法的cmd命令都能通过system函数来被C语言程序使用。

至于为什么Windows系统是大小写不敏感的，这是因为它继承了MS-DOS操作系统。MS-DOS是比尔·盖茨早期的作品，而了解比尔·盖茨的人都知道，他的创业历程中，有一门语言相当重要，那就是BASIC语言，而BASIC语言就是不区分大小写的，所以一般认为微软公司的操作系统大小写不敏感是因为比尔·盖茨对这门语言有较深的感情。

BASIC的全称是**Beginners' All-purpose Symbolic Instruction Code**，也就是“初学者通用代码”。当今流行的语言中，大多数受C语言语法影响，但是BASIC却不同，它拥有完全不同的语法风格。从任何一个角度来讲，BASIC都不是一门很好的语言，但是它却为PC机的推广和普及做出了巨大的贡献，影响之大，丝毫不低于C语言。

进行游戏的分支中，先后调用了3个函数。首先我们来看接受用户输入的playerResult函数，第51行提示用户应该怎样输入合法的数据，第52到55行接受输入。第56到66行则用于接受用户的下注，用户下注必须在一定范围之内，最低为0，最高为用户的当前资产，我们在循环接受用户的输入之前，先把ante设定为-1，防止上一局游戏中合法的数据造成干扰。另外，这里还有一个细节，是关于scanf函数的比较高的技巧了，如果我们在第61行的scanf函数中输入一个字符，那么scanf函数根本不会接受这个数据，因为类型不匹配，但是这个数据会一直残留在缓冲区中，直到下一次循环，scanf发现缓冲区中有数据，于是不再等待用户输入，接着依然不接受该数据，这样就会进入一个死循环，为了避免这种情况，我们需要在59行加上重置缓冲区的语句。如果用户输入了合法的数据，循环就会结束，函数也会相应地结束。

AIResult函数中，前两行代码我们从未见过，其实这两行的作用就是产生一个随机数。第一行的srand函数用于初始化随机数序列，srand函数会因为它的参数的不同而产生不同的随机数序列，随机数序列影响到随机数的产生，如果没有初始化它，程序将会因为随机数序列相同而产生同样顺序的随机数，由于每次运行程序的时间都不同，所以我们可以通过时间来设定不同的随机数序列，获取时间的函数是time，它由time.h提供。rand函数的作用才是产生随机数，它由stdlib.h提供。rand()%3表示用产生的随机数来对3取余，这就能保证结果在0~2之间，rand()%3+1则会使结果的范围变成1~3，正好和我们需要的范围一致。还有一个问题，产生的随机数是int型数据，而我们需要的是char型数据，这就需要一点转换了，根据ASCII码表，我们很容易看出一个0~9之间的整数加上'0'之后就能转换成对应的字符。

**小提示**

计算机没有任何思维，它怎么能像人一样实现猜测呢。要让计算机能够猜测，就需要使用随机数，然而问题又来了，随机数本身不也是一种猜测吗。其实，产生随机数的函数rand用到了一种很复杂的方法来实现产生随机数，严格来讲，它并不是真正意义上的随机数，它的产生和初始化它的“种子”有关，而每时每刻的时间都是不同的，所以用时间作为种子来初始化随机序列，就能达到产生随机数的效果。为了能使用随机数的功能，千万别忘了初始化随机序列，实际上，初始化的工作一次程序做一次就够了，每次都做反而很耗时。读者可能会问，一天只有24小时的时间，想要完全不重复恐怕不可能吧，其实，计算机储存的时间并不是我们人类所理解的时间，计算机的储存方式有点类似于数轴，每时每刻就对应数轴上的一个点，所以每时每刻都是不同的。

最后是结算的函数，我们可以使用多个if-else或者多个switch的嵌套来判断，不过那样会使代码显得很长很杂，仔细分析这个游戏的机制，我们能得到一种更好的结算方式。我们知道，'1'、'2'、'3'分别表示石头、剪刀、布，而石头克剪刀、剪刀克布、布克石头。如果用做差的方式来比较的话，很明显当差为1或者-2时，玩家获胜，当差为0时，游戏平局，其它情况为玩家失败。如果获胜，则增加玩家的资产，增加的多少取决于玩家所下的赌注，如果游戏失败，则提示失败，并计算玩家是否已经破产，如果玩家已破产，意味着游戏不能继续进行，所以需要直接结束程序。

虽然我们能够通过全局变量来实现这个游戏，但是事实上C语言能提供更好的解决方案，全局变量虽然看起来很好用，但是它存在一些缺点，一方面，全局变量会一直占用内存直到程序运行结束，另一方面，全局变量可以被多个函数修改，这不利于数据的维护。因此，全局变量不应该被滥用。

一个C语言程序所访问的内存分为很多个区段：栈区（**Stack**）、堆区（**Heap**）、静态区（**Static**）、常量区、代码区。代码区的作用自然是保存程序运行所需要的代码，常量区用于保存一些常量，比如const常量，以及一些文字常量。而栈区则是局部变量存在的区域，栈区的特点是空间小，速度快。堆区与动态内存有关，我们将在后面的章节中介绍。而静态区则是全局变量所在的区域，不过在静态区除了由全局变量以外，还有另外一种变量：静态变量，静态变量同时拥有局部变量和全局变量的一些特点，它的作用域和局部变量一样，只有一个语句块的大小，但是它也会一直占用内存，并且会被自动初始化。

声明静态变量的方式是在变量类型前加上一个static。

1. #include <stdio.h>
2. void count();
3. int main()
4. {
5. for (int i = 0; i < 5; i++)
6. {
7. count();
8. }
9. getchar();
10. return 0;
11. }
12. void count()
13. {
14. static int num = 0;
15. num++;
16. printf("函数被调用了%d次\n",num);
17. }

程序5-4-5

尝试着分析一下程序5-4-5的运行结果并上机测试。很多读者会认为运行结果会是输出5次“函数被调用了1次”，但是结果却是num不断增加，实际运行结果是从“函数被调用了1次”一直输出到“函数被调用了5次”。也就是说，第14行的num仅仅在第一次调用count函数的时候被初始化为0，在以后的调用中第14行的代码都不能影响到num的值，也就是说，静态变量只会被初始化一次，并且虽然它的作用域只有整个函数，但它的生命周期却不局限于函数调用的这段时间。

如果给在全局变量前面加上static会怎样呢？对于当前的源文件来讲，不会有任何变化，因为全局变量本质上讲也是静态变量的一种。

5.5 函数的嵌套和递归调用

本节将介绍函数的嵌套调用和递归调用。嵌套调用其实并不复杂，我们前面实际上也已经见过了嵌套调用，而递归调用则是函数调用中非常精彩的一部分。递归的技巧性很强，而难度又不是特别高，本节将会通过大量的例子来展示递归的精髓。

所谓嵌套调用，指的是一个函数1调用了函数2，函数2中又调用了函数3。实际上，前面出现的程序中，绝大多数都涉及嵌套调用，因为scanf和printf也算是函数。关于函数的嵌套调用，只需要对程序执行的顺序稍加注意即可，本身并没有太大的难度。

1. #include <stdio.h>
2. void fun1();
3. void fun2();
4. int main()
5. {
6. fun1();
7. getchar();
8. return 0;
9. }
10. void fun1()
11. {
12. printf("fun1开始\n");
13. fun2();
14. printf("fun1结束\n");
15. }
16. void fun2()
17. {
18. printf("fun2执行\n");
19. }

程序5-5-1

程序运行结果如下：

fun1开始

fun2执行

fun1结束

很明显，当运行到第13行的时候，程序暂时将fun1“晾在一边”，然后开始执行fun2，直到fun2执行完毕再返回fun1接着执行，这和前面说到的函数调用时程序执行顺序很相似，当调用层数更高时，分析方法也是类似的。

接下来我们将花很长的篇幅来介绍递归（**recursion**）。所谓递归调用，就是指一个函数在执行的过程中直接或者间接地调用自己。这听起来相当的不可思议，但是我们说，任何一个函数体内可以调用任何函数，包括这个函数本身，于是递归就有了存在的可能。递归也分为直接递归和间接递归，所谓直接递归，就是一个函数体内直接调用它自身，而间接递归则是指一个函数在多层的嵌套调用之后，某一层嵌套中又调用了这个函数，如果我们在程序5-5-1的第18行和第19行之间加上一句fun1()，这就属于间接递归。

**小提示**

任何函数，包括主函数，都能调用直接或间接地调用自身，但是处于对程序安全性以及代码可读性的考虑，我们不推荐主函数递归调用，也不推荐任何其它函数调用主函数。

递归可没有说起来这么简单，试想，如果一个函数体内只有1行代码，这行代码的作用就是调用它自身，那么程序运行到这个函数的时候将会“停不下来”。因此，在递归调用的时候，必须分情况讨论，在满足条件时不再递归。接下来我们用函数递归调用的思想来求斐波拉契数列的某一项，并以此为例来进一步讲解递归的注意事项。

1. #include <stdio.h>
2. int fibSequence(int);
3. int main()
4. {
5. int result = fibSequence(10);
6. printf("斐波那契数列的第10项是：%d",result);
7. return 0;
8. }
9. int fibSequence(int index)
10. {
11. if (index <= 0)
12. {
13. printf("数据有误\n");
14. return -1;
15. }
16. if (index ==1)
17. {
18. return 0;
19. }
20. else if (index == 2)
21. {
22. return 1;
23. }
24. return fibSequence(index - 1) + (index - 2);
25. }

程序5-5-2

斐波那契数列的第1项为0，第2项为1，之后的每一项都等于它前面两项的和，于是斐波那契数列就是这样的：0 1 1 2 3 5 8...。对于递归来说，规律很重要，现在我们找到了规律。

在程序5-5-2中，用result来保存斐波那契数列的第10项的值。主函数中，我们先假设fibSequence函数能够求出斐波那契数列的任意项的值，而不关心它到底是怎么求出来的。

在fibSequence函数中，首先对参数进行判断，如果参数不是正整数，那么就没办法求对应的数据，于是输出错误信息并结束调用。

如果输入的数据是合法的，则首先检查是不是1或者2这两个特殊的项（0项和1项不能用规律求解，它们的值是固定的），如果是1或者是2，则直接返回对应的数据（1对应0，2对应1）。

如果输入的数据合法，而且不是1也不是2，那么就轮到递归大显神威了。既然斐波那契数列的第n项（n>2）等于第n-1项和第n-2项的和，那么我们在用fibSequence函数来求第n项的时候，等价于用fibSequence函数来求第n-1项和第n-2项的和。于是这才有了第25行的调用。表面看来，在递归调用的过程中，这些值始终都是未知的，可是最后为什么成功求出了值呢，这是因为在不断递归的过程中，n-1和n-2以及它们再减1再减2的结果最终会等于1或者2，一旦它们等于这两个值，函数就能一步一步地倒推回去，最终求出第n项的值。

递归调用的过程分为两部分，递推阶段和回归阶段。所谓递推阶段，就是让问题不断拆分成相同或类似的更小的问题，更小的问题能使用与原问题相同的思路求解。而回归阶段，则是遇到了一个不能再拆分的最小的问题，到了这一层调用之后，函数递归不会继续进行，而是会依次地返回上一层递归调用并通过这一层的结果来解决上一层的问题，最终解决最大的那个问题。而这个最小的问题，就被称为递归出口，一个有意义的递归调用一定要有递归出口，否则只有递推阶段没有回归阶段，递归调用也就没有任何价值。

在程序5-5-2中，将求fibSequence(n)转换成求fibSequence(n-1)+ fibSequence(n-2)的过程就是递推阶段，当这个转换进行到某一层的时候，n就等于1或者2，这时候问题不能再拆分，于是直接利用已知的1和2的结果返回上一层调用，开始回归阶段，如此重复，最终得到第一层调用（fibSequence(10)）的结果。

下面我们用一个求一个自然数的阶乘，前面第4章我们用循环语句求过，通过这个例子，我们来感受递归与循环语句的差别。

1. #include <stdio.h>
2. int factorial(int);
3. int main()
4. {
5. int result = factorial(5);
6. printf("5的阶乘是%d",result);
7. getchar();
8. return 0;
9. }
10. int factorial(int index)
11. {
12. if (index < 0)
13. {
14. printf("数据有误\n");
15. return 0;
16. }
17. if ( (index == 0) || (index==1) )
18. {
19. return 1;
20. }
21. else
22. {
23. return index\*factorial(index - 1);
24. }
25. }

程序5-5-3

我们知道，0和1的阶乘是1，而对于一般情况来讲，n!=n\*(n-1)!（n>1）。上面这段程序中，当index等于0或者1的时候，就能遇到递归出口。而对于其它的自然数n，则分别用n\*factorial(n-1)来作为结果。这段代码并不比程序4-2-2中用循环语句来求阶乘看起来短，但是却显得优雅许多。

上一章的习题中，有一道题是关于“猴子吃桃”问题的，当时要求用循环语句来解决，现在我们尝试用递归的思想来分析它的递推公式和递归出口：每一天的桃子数都等于前一天的一半再减一，这是递推公式，而第10天的桃子数为1，这就是递归出口。有了这两个条件，用递归来解决猴子吃桃问题就显得相当简单了。

1. #include <stdio.h>
2. int peach(int);
3. int main()
4. {
5. int numOfPeach = peach(10);
6. printf("第一天的桃子数为：%d",numOfPeach);
7. getchar();
8. return 0;
9. }
10. int peach(int day)
11. {
12. if (day <= 0)
13. {
14. printf("数据有误\n");
15. return -1;
16. }
17. if (day == 1)
18. {
19. return 1;
20. }
21. return peach(day - 1) \* 2 + 2;
22. }

程序5-5-4

程序5-5-4中，将1设为递归出口，这一天的桃子数为1，而递推阶段的公式则和上一节中的循环语句非常相似。总体来说是很简单的一个程序。

前面说了这么多递归的优点，那么递归真的就是完美的吗？其实不然，递归在很多情况下都存在效率低下的缺点。

所谓效率低下，就是指通过其它手段能在更短的步骤中求解的问题，用递归需要花费更多的时间和步骤。我们将通过一个例子来证明这一点。

接下来我们将用递归的思想来解决“楼梯问题”，所谓楼梯问题，就是指1次能跨1步或者2步，求n层楼梯一共有多少种上楼方式。

首先我们来分析这个问题的递推公式，假设有n级（n>2）台阶，那么第1步我们可以走1级或者2级，很明显第1步走1级台阶之后并没有走完，第1步走1级台阶的之后的走法总数和直接走n-1级台阶的走法总数相同。这样问题就被转换成了求n-1级台阶的走法与n-2级台阶走法之和，这就是递推公式。容易看出，这也是斐波那契数列，只不过首项不为0。

同时，当n=1的时候，只有1种走法，而当n=2的时候，有2种走法（直接走2级台阶

或者一次走1级），这是不能再分割的最小的问题，也就是递归出口。

1. #include <stdio.h>
2. int floor(int);
3. int num1 = 0,num2=0;
4. int main()
5. {
6. int result =floor(20);
7. printf("20级台阶的走法有%d种\n",result);
8. printf("%d %d",num1,num2);
9. getchar();
10. return 0;
11. }
12. int floor(int num)
13. {
14. if (num <= 0)
15. {
16. printf("数据有误\n");
17. return -1;
18. }
19. if (num == 1)
20. {
21. num1++;
22. return 1;
23. }
24. else if (num == 2)
25. {
26. num2++;
27. return 2;
28. }
29. return floor(num - 1) + floor(num - 2);
30. }

程序5-5-5

现在看来程序5-5-5应该算是比较简单的了，毕竟求斐波那契数列在之前的程序中已经做过了，现在我们要分析为什么这样用递归来求斐波那契数列效率很低。就用程序5-5-5中的数据来分析，要求floor(20)，首先要求floor(19)和floor(18)，这里不能体现它效率低的一面，但是如果继续往下分析。求floor(19)等价于求floor(18)+ floor(17)，求floor(18)相当于求floor(17)+ floor(16)。如果把18全部消去的话，那么求floor(20)就一共要求3次floor(17)和2次floor(16)。而floor(17)和floor(16)还可以继续往下展开，每展开一次，当展开到floor(2)和floor(1)的时候，这个数字将会非常大，我们定义了两个全局变量num1和num2，分别用来统计floor(1)和floor(2)被计算的次数，得到的结果是2584和4181，效率低得恐怖。

而如果不用递归，我们可以通过floor(2)=2，floor(1)=1这两个已知条件来逐渐往上层求，最终每个floor(n)都只需要求一次，最终效率会提高无数倍。

我们看看用循环语句如何求解楼梯问题。

1. #include <stdio.h>
2. int main()
3. {
4. int numOfFloor,result;
5. do
6. {
7. rewind(stdin);
8. printf("请输入台阶数：\n");
9. scanf("%d", &numOfFloor);
10. } while (numOfFloor <= 0);
11. if (numOfFloor == 1)
12. {
13. result = 1;
14. }
15. else if(numOfFloor==2)
16. {
17. result = 2;
18. }
19. else
20. {
21. int i = 1, j = 2;
22. for (int k=0;k<numOfFloor-2;k++)
23. {
24. result = i + j;
25. i = j;
26. j = result;
27. }
28. }
29. printf("%d层台阶有%d种走法\n",numOfFloor,result);
30. rewind(stdin);
31. getchar();
32. return 0;
33. }

程序5-5-6

程序5-5-6为循环语句版的楼梯问题求解程序。针对于1和2的情况，程序直接的出结果。而对于n>2的一般性情况，则通过一个循环语句以及两个循环变量来不断累加，对于任意的n（n>2），只需要循环n-2次即可求出结果。即使是20级台阶，也只需要循环18次，执行18\*3=54条语句就能得到答案，n每增加1，需要执行的语句就只增加3。台阶数与耗时之间是一个一次函数的关系，而对于递归来说，台阶数与耗时之间的关系则是一个指数函数，当n=100的时候，程序5-5-6依然能很快求出答案，而程序5-5-5恐怕就得让你等上很长很长一段时间了。

当然，递归的效率不一定都很低，在很多情况下递归也能做到不浪费性能，下一章我们将介绍一个这样的例子。

5.6 多个文件的程序

函数的意义在于结构化编程，在用C语言开发项目的时候，我们总是会把一个项目拆分成多个函数。前面说过，这样做的目的之一是将一个项目分给不同的人来编写。而既然要分给多个人来写，那么就肯定应该把这些函数写在不同的源文件里，不然这样的分工就毫无意义了。这一节我们将讲解如何将一个C语言程序拆分成不同的函数并且在不同的文件里分别编写这些函数。

我们知道，生成一个C语言程序的最后一步是连接，这个连接指的是把程序的各个模块组合起来。如果在某个函数中调用另外一个函数，那么编译阶段只会检测这个函数是否已经声明，如果已经声明则能成功通过编译，而连接阶段才会检测这个函数是否已经定义，已经定义才能通过连接并生成可执行的程序。程序5-6-1则能证明这一点。

1. #include <stdio.h>
2. void fun();
3. int main()
4. {
5. fun();
6. getchar();
7. return 0;
8. }

程序5-6-1

我们在Visual Studio 2015下面输入这个程序，然后按Ctrl+F7的组合键（这个组合键的作用是只执行编译阶段，而不连接和执行），我们可以从编译的结果看出，这段程序是能够成功通过编译的。但是，如果我们直接按F5，则会被提示连接错误，错误原因是fun是一个无法解析的外部符号。主函数被编译成了一个模块，这个模块中调用了fun函数，而连接阶段，连接器试图找到一个包含fun函数的模块来和主函数的模块相组合，但是我们根本就没有为fun函数编写相关代码，因此编译阶段也不会产生包含它的模块，所以连接阶段也就不能顺利完成。

从上面这个例子中我们可以看出，只要在调用一个函数之前声明了它，程序就能成功通过编译。

而我们在分模块开发项目的时候，对于某个单独的模块来讲，能够通过编译即可，并不需要能够成功执行。通常情况下，我们是等所有的模块都成功通过编译之后，再来统一调试。

既然要分成多个源文件来编写不同的函数，并且一个源文件中需要调用另外一个源文件中的函数，这个时候应该怎样声明函数呢？最简单的想法是在一个源文件前面声明所有它需要调用到的函数，包括本文件内的和其它文件内的，这样做当然是合理的，但是这样做的缺点是一个源文件的开头出可能会出现大量的函数声明的语句，不利于程序的可读性和可维护性。最好的做法是在头文件中给出这些函数的声明，对于这种做法我们一点也不陌生，事实上，stdio.h头文件中就包含了printf等函数的声明，我们只需要包含stdio.h就能使用printf函数，而printf函数是编译器已经为我们定义好了的，所以我们无需自己编写代码来实现printf函数，程序也一样能通过连接。

下面我们将通过一个简单的例子来讲解如何使用头文件来实现函数的声明。

前面第1章讲解了如何添加一个源文件，添加头文件的思路也是相似的。这时候，我们需要右击解决方案视图中的头文件而不是源文件了，同样地，选择添加->新建项，这时候我们要选择的就是头文件而不是源文件了，给新建的头文件起名为functions.h，然后点击确定。

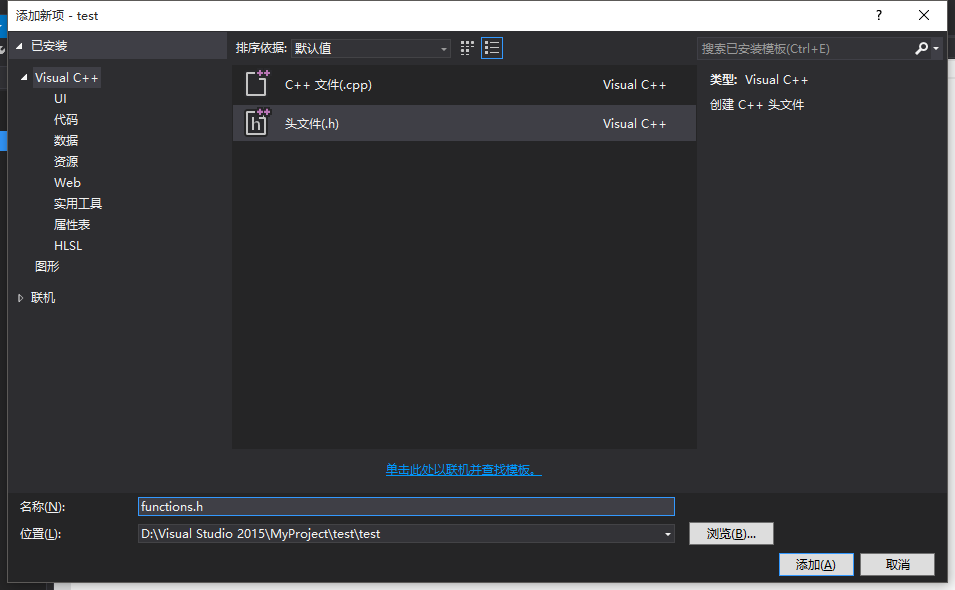


图5-6-1

在新建的头文件中，我们可以看到，有这样一段代码：#pragma once。这是Visual Studio 2015比较人性化的一面，这段代码的作用是防止头文件被重复包含，关于头文件重复包含所带来的危害，我们将在后面的章节中讲解，现在我们暂时删掉这一行代码或者干脆不理它。然后在头文件中添加这一行代码：void printHello();

接下来，我们新建两个源文件（具体方法请参考第2章），分别起名为main.c和printHello.c。在printHello.c中，我们添加下面这段代码：

1. #include <stdio.h>
2. #include "functions.h"
3. void printHello()
4. {
5. printf("Hello，这是一个函数\n");
6. }

程序5-6-2

然后在main.c中添加如下代码：

1. #include <stdio.h>
2. #include "functions.h"
3. int main()
4. {
5. printHello();
6. getchar();
7. return 0;
8. }

程序5-6-3

这时候，我们按F5来直接执行这个程序，发现编译和连接都是能通过的。首先看到printHello.c，在这个源文件里面，我们包含了两个头文件，一个是stdio.h，另外一个则是functions.h。注意到，包含stdio.h的时候，我们用的是尖括号，包含functions.h的时候则换成了双引号。两种方式有一点点细微的区别，如果我们用尖括号来包含一个头文件的话，编译器就会直接到库文件的目录下去寻找相关的头文件，如果用双引号来包含，那么编译器会优先在当前工程的目录下去寻找目标头文件，如果找不到再去库文件目录下寻找。也就是说，如果我们用双引号来包含stdio.h，程序依然是合法的，但是如果我们用尖括号来包含functions.h，程序就不能编译通过了，除非你把functions.h复制到库文件的目录下。既然双引号是万能的，而尖括号不是，那么我们有没有必要一律采用双引号而不使用尖括号呢？事实上，如果用双引号来包含stdio.h，会比尖括号花费更多的时间（首先在当前目录下找，然后再去库文件目录找）。所以通常情况下有这样的约定：包含库文件用尖括号，包含自定义头文件用双引号。printHello.c中还给出了printHello函数的定义。

我们再来看main.c中，在这里我们同样包含了两个头文件，由于functions.h中给出了printHello函数的声明，所以我们可以直接使用printHello函数，在连接阶段，连接器会把编译main.c产生的主函数模块和printHello.c产生的printHello函数模块连接起来，程序会成功执行。

有时候，一个文件里面不止有一个函数，而这些函数里面，并非所有的函数都是需要对外提供接口的，也就是说，有的函数是我们希望其它文件无法调用的，这时候，我们可以用static来修饰这样的函数。

假如我们在functions.h中添加这样一行代码：void fun();来声明fun函数，然后在printHello.c中添加这段代码：

1. static void fun()
2. {
3. printf("这是一个本地函数\n");
4. }

程序5-6-3

然后再在main.c的第5行和第6行之间增加这样一行代码：fun();。

接着按F5尝试运行。我们将会发现，程序不能通过连接。这说明functions.h中声明的fun函数的定义并不是printHello.c中的fun函数，这也说明了凡是用static修饰的函数都不能被其它文件中的函数所调用。

**小提示**

面向对象的程序设计语言中，有一个概念叫做封装（**encapsulation**），意思是只对外提供接口，而隐藏具体的实现细节。C语言中也有类似封装的思想。C语言中用函数来实现接口，这些作为接口的函数本身可能还需要调用其它的函数，而这些函数不需要作为对外的接口，这时候，我们就应该对外隐藏这些函数，具体做法就是用static来修饰这些函数。

前面5.4节在介绍全局变量的时候，我们曾说全局变量的生命周期为整个程序的运行时间，又说全局变量的作用域是从它声明之后的代码，那么这是否意味着全局变量只能在某一个文件中使用呢？其实不然，我们知道，函数可以分为声明和定义两部分，而我们总是说声明了一个变量，却从来没说过定义一个变量，事实上，通常情况下，变量的声明和定义都是放在一起的，但是在某些情况下，我们可以将声明和定义分离，从而实现将一个全局变量的作用域从一个文件扩张到另外一个文件。接下来我们将演示这种操作。

我们新建一个工程，为这个工程新建两个源文件，分别起名叫main.c和showInfo.c（事实上，改成其它任何合法的名字也都是可行的），然后添加一个头文件，起名为function.h。接着向三个文件中分别添加一下的代码：

1. /\*main.c\*/
2. #include <stdio.h>
3. #include "function.h"
4. int x = 10;
5. int main()
6. {
7. showInfo();
8. getchar();
9. return 0;
10. }

程序5-6-4 main.c

1. /\*showInfo.c\*/
2. #include <stdio.h>
3. #include "function.h"
4. extern int x;
5. void showInfo()
6. {
7. printf("main.c中的变量的值：%d\n",x);
8. }

程序5-6-5 showInfo.c

至于function.h，我们只需要在里面加上这样一行即可：void showInfo();。然后我们按F5执行此程序，发现它能够成功通过编译和连接。注意到showInfo.c中的第4行，这里我们声明了一个全局变量x，只是x有一个前缀extern，extern的意思是外面的，外来的。所以这里声明的全局变量实际上是外部的一个变量，而这个变量当然就是main.c中的x了。也就是说，showInfo.c中的x只是一个声明，而它的定义在main.c中。如果我们删掉main.c中的第4行，那么程序依然是能够通过编译的，但是不能连接成功，就像一个被调用的函数只有声明没有定义一样。

另外，头文件中也可以包含其它的头文件，很多情况下，我们不得不这么做，不过就目前而言，我们还没有必要这样做。就上面这个例子中的function.h来说，我们完全可以在里面加上这样一句：#include <stdio.h>。这样我们在main.c和showInfo.c中就不用再包含stdio.h头文件了，不过这样显然会降低代码的可读性，所以一般我们在使用某个函数的时候，都要加上对应的头文件，即使之前包含的头文件已经把它们包含过了，不用担心这样会造成头文件重复包含，标准库头文件的作者已经在头文件中加入了特殊的代码来防止重复包含（比如前面说过的#pragma once）。

事实上，extern不止可以修饰变量，还可以修饰函数。我们重新建立一个工程，并向其中添加两个文件，分别起名为main.c和extern.c，分别添加如下的代码。

1. /\*main.c\*/
2. #include <stdio.h>
3. extern void externFun();
4. int main()
5. {
6. externFun();
7. getchar();
8. return 0;
9. }

程序5-6-6 main.c

1. /\*extern.c\*/
2. #include <stdio.h>
3. void externFun()
4. {
5. printf("这里是一个外部函数\n");
6. }

程序5-6-7 extern.c

程序能够成功运行，而如果我们删掉extern.c中的全部内容，那么程序同样能通过编译，只是不能连接成功。

5.7 函数的返回值

函数的返回值很好解释，返回值就相当于助手做完一件事之后给我们带回来的东西。不过由于本章的例子中返回值为void的函数占了绝大多数，所以对返回值这一部分的理解显得不是很深刻。

我们说，当程序在执行某个函数的时候，一旦发现了return语句，那么函数调用就会终止，而配合if-else语句，一个函数中可以出现多个return语句，不难理解，所有的这些return只有一个有可能会被执行，事实上我们在前面已经见过了这样的例子。

1. #include <stdio.h>
2. int max(int, int);
3. int main()
4. {
5. int num = max(5, 6);
6. printf("5和6中比价大的是%d\n",num);
7. num = max(8, 7);
8. printf("8和7中比价大的是%d\n", num);
9. getchar();
10. return 0;
11. }
12. int max(int a, int b)
13. {
14. if (a >= b)
15. {
16. return a;
17. }
18. return b;
19. }

程序5-7-1

在上面这段程序中，我们编写了一个max函数用来求两个整数中更大的那个值。在max函数中，我们用到了一个条件语句，如果a大于等于b，那么就把a作为返回值。在接下来的代码中我们没有写else，而是直接写了一个return b;，因为我们知道，一旦a>=b的条件满足，那么程序就会执行第16行的return语句，这样第18行的return语句就没有机会被执行，所以不会对程序运行结果造成影响。而如果a>=b的条件不满足，那就说明b更大，同时第16行的return不能被执行。所以两个return语句是互相不干扰的。

**试一试**

在程序5-6-8的第18、19行之间插入一句printf语句（随便输出什么都行），观察运行结果是否会有变化。

我们在提到void的时候，曾说过返回值为void的函数不能有任何返回值，但是我们也说过这并不意味着返回值为void的函数中不能出现return语句。事实上前面的程序5-2-4中就已经在void函数中使用过return，只不过那个程序中的return语句毫无存在感，它对函数的执行没有任何影响。

在某些情况下，合理地在返回值为void的函数中使用return语句能够使代码更为简洁。

假设我们要编写一个函数factorial，这个函数以一个int型变量作为参数，它的作用是求出并输出这个int型变量的阶乘。factorial函数中肯定要对参数的合法性加以判断，如果参数小于0，则只能输出错误信息；如果参数等于0，那么输出0的阶乘是1，如果参数大于0，那么就用循环语句来求阶乘。如果我们用if-else语句来实现这一功能的话，很可能会出现嵌套，这就不利于程序的可读性和可维护性了。

通过分析，我们不难看出，上面提到的这3种情况是彼此独立的，它们没有可能同时成立，因此我们可以用if+return的方式来避免if-else的嵌套。

1. #include <stdio.h>
2. void factorial(int);
3. int main()
4. {
5. int num;
6. printf("请输入一个自然数，程序将求出它的阶乘\n");
7. scanf("%d",&num);
8. factorial(num);
9. rewind(stdin);
10. getchar();
11. return 0;
12. }
13. void factorial(int num)
14. {
15. if (num < 0)
16. {
17. printf("无法求出负数的阶乘\n");
18. return;
19. }
20. if (num == 0)
21. {
22. printf("0的阶乘是1\n");
23. return;
24. }
25. int result=1;
26. for (int i = 1; i <= num; i++)
27. {
28. result \*= i;
29. }
30. printf("&d的阶乘是%d\n",num,result);
31. }

程序5-7-2

前面我们说，主函数的返回值是可以省去的，而其它返回值不为void的函数则必须有一个返回值。这个必须有一个返回值指的是函数中必须出现至少一次return语句，而在某

些情况下，我们可以让返回值不为void的函数也没有返回值。

1. #include <stdio.h>
2. int fun(int);
3. int main()
4. {
5. int num;
6. num = fun(10);
7. printf("%d", num);
8. getchar();
9. return 0;
10. }
11. int fun(int num)
12. {
13. if (num != 10)
14. {
15. return 10;
16. }
17. }

程序5-7-3

在Visual Studio 2015的环境下，程序5-7-3能够成功编译运行，观察运行结果，我们发现输出的是一个随机值。fun函数中出现了return语句，但是只有在参数不为10的时候这个return语句才会被执行，而我们正好把10作为参数传递给了fun函数，于是函数就没办法返回任何值，在这种情况下，第6行的赋值语句就不能成功执行，于是主函数中的num就保持了它原有的值。像程序5-7-3中显然存在一个巨大的隐患，有些编译器是不允许这样的代码通过编译的。我们也不推荐读者写出这样的程序。

**小提示**

其实在Visual Studio 2015的环境下，程序5-7-3运行之后会得到一个固定的结果而不是随机值，这是因为Visual Studio 2015会默认将栈空间中的变量初始化成某个值，而num作为一个局部变量，自然也就会被初始化。而在其它的编译器中，就不一定会出现这种情况了。

我们再说一说主函数的返回值，C语言的标准规定，返回值类型只能为int，但是很多C语言的编译器是允许用其它类型作为返回值的（包括Visual Studio 2015）。

1. #include <stdio.h>
2. char main()
3. {
4. printf("返回值为char类型的主函数\n");
5. getchar();
6. return '0';
7. }

程序5-7-4

在Visual Studio 2015下面，程序5-7-4能够成功运行。注意到第6行，这里写的是'0'而不是0，这两个数据是不同的。

因为编译器支持这么做，所以很多C/C++的基础教程甚至很多程序员都喜欢用void类型作为主函数的返回值，但是我们说这种行为是非常不好的，读者可以参考一些著名的C/C++入门书籍（《C Primer Plus》 、 《C++ Primer》等），在这些著作里面，我们绝对看不到void main这样的写法。C++之父**Bjarne Stroustrup**曾明确表示，void main的写法从来不存在于C/C++的标准规定中。即使有些编译能支持这样的写法，我们也应该养成良好的习惯，按照C语言的标准来写。另外，如果你还在看其它的一些发行时间较早的C语言著作，你可能还会看见main函数前面没有任何类型修饰而只有一个main的写法，这种写法就默认了返回值是int，这样做曾经是合理的，但是现在已经不再适用。C语言的早期版本允许这样写，但是C99及以后的C语言标准明确规定C语言不再支持这样的写法。

当然，主函数中也可以出现多个return语句。

我们说过，return后面的数据必须与函数的返回值类型保持一致，其实这个限定并没有那么严格。

1. #include <stdio.h>
2. int fun();
3. int main()
4. {
5. double douNum = fun();
6. int intNum = fun();
7. printf("%f\n", douNum);
8. printf("%d", intNum);
9. getchar();
10. return '0';
11. }
12. int fun()
13. {
14. return 10.1;
15. }

程序5-7-5

程序5-7-5能够成功运行。但明明fun函数的返回值就是int，但是我们却把10.1这个浮点数作为了返回值。在主函数中，分别用一个double型数据和一个int型数据来接受这个数据，结果发现double型数据的值为10.0，int型数据也是。这说明第14行的10.1被强制转换成int型数据，然后才被当成返回值返回。

我们还说过，char型数据和int型数据是相互关联的，char型数据本身也是以整数来储存的，char和int相互转换的依据是ASCII码。用char类型的数据来作为一个返回值为int型数据的函数的返回值是合理的，我们可以编写一个函数来求一个字符的ASCII码。

1. #include <stdio.h>
2. int toAscii(char);
3. int main()
4. {
5. char ch;
6. printf("请输入一个字符，程序将求出它的ASCII码\n");
7. scanf("%c",&ch);
8. int num = toAscii(ch);
9. printf("%c的ASCII码是%d",ch,num);
10. rewind(stdin);
11. getchar();
12. return '0';
13. }
14. int toAscii(char ch)
15. {
16. return ch;
17. }

程序5-7-6

程序5-7-6中，toAscii函数接受一个char类型的数据作为参数，然后将这个参数按照int型的方式返回，得到的当然就是这个char型数据的ASCII码了。

有的读者可能会说，直接用int num=(int)ch的方式来求ASCII码不好吗？这样做当然是合理的，并且也是最佳的选择，不过程序5-7-6的意义并不在于这个转换的过程，而在于说明返回值类型的转换的原理。

**试一试**

改写程序5-7-6，提示用户输入一个整数，然后求出这个整数对应的字符。尝试输入超出char型数据表示范围的数据，看会得到什么样的结果。

如果读者亲自做了上面的“试一试”，读者将会发现一个有趣的现象，输入97和输入353所得到的结果是一样的。这种现象叫做绕回处理，我们知道char型数据的表示范围是-127~128，一共256个数。如果我们尝试把129赋值给一个char型数据，那么这个char数据实际的值将会是-127，也就是说，一旦超过最大值，便又从最小值开始循环。而353正好等于97+256，所以它们对应的结果也就一样了。

scanf函数和printf函数是我们使用最多的函数，正如我们前面所说，这两个函数的使用技巧是非常多的，而事实上它们也有返回值。Visual Studio 2015是一款非常智能的工具，当我们把鼠标悬停在scanf或printf函数上的时候，它会自动显示提示信息，从提示信息里我们可以看出这两个函数的返回值都是int型的，如下图所示。



图5-7-1 scanf的返回值



图5-7-2 printf的返回值

我们知道scanf可以一次性输入多个数据，printf也可以一次性输出多个数据。在它们的函数声明中，const char \* const表示的是双引号内的字符串（按原样输出的字符或格式控制字符），读者对此只需了解即可，第二个参数...则表示不定长的参数，我们可以向它们传递任意数量的参数（正是因为可以传递任意数量的参数，一次性输入输出多个数据才有了可能），读者同样只需了解即可。我们先说printf，printf的返回值就是成功输出的字符的个数，这样说读者可能不会有什么概念，我们可以通过程序来深入理解。

1. #include <stdio.h>
2. int main()
3. {
4. int count,i=0,j=0;
5. count=printf("i=%d,j=%d\n",i,j);
6. printf("上面的printf函数成功输出的字符个数：%d\n", count);
7. count = printf("i=%d,j=%d\n", i, j,i);
8. printf("上面的printf函数成功输出的字符个数：%d\n", count);
9. count = printf("i=%d,j=%d%d\n", i, j);
10. printf("上面的printf函数成功输出的字符个数：%d\n", count);
11. getchar();
12. return 0;
13. }

程序5-7-7

程序的某次运行结果如下：

i=0,j=0

上面的printf函数成功输出的字符个数：8

i=0,j=0

上面的printf函数成功输出的字符个数：8

i=0,j=02297931

上面的printf函数成功输出的字符个数：15

我们一共调用了3次printf函数，第一次，格式控制字符与参数完全匹配，2个%d分别对应于i和j。因此成功输出的字符个数是8（("i=0,j=0"一共是7个字符，而\n也算一个字符，因此一共8个字符）。第二次调用的时候，依然是有2个格式控制字符，但是我们故意多给了一个参数，输出后printf的返回值依然是8，这说明在没有对应的格式控制字符的情况下，多出的参数不会被输出。第三次调用的时候，printf函数中有3个格式控制字符，但是参数只给出了2个，因此最后一个格式控制字符没能找到匹配的参数，然后输出了一个随机值，成功输出的字符一共有15个。

我们可以通过printf的返回值来判断它的输出情况，虽然这在实际项目开发中的应用是比较少的。

而对于scanf来说，它的返回值是成功匹配的格式控制字符的个数（而不是成功输入的字符的个数）。我们同样通过程序实例来深入理解。

1. #include <stdio.h>
2. int main()
3. {
4. int count,i,j;
5. count = scanf("%d",&i);
6. printf("上面的scanf函数成功匹配的格式控制字符数是：%d\n",count);
7. count = scanf("%d%d", &i,&j);
8. printf("上面的scanf函数成功匹配的格式控制字符数是：%d\n", count);
9. count = scanf("%d", &i);
10. printf("上面的scanf函数成功匹配的格式控制字符数是：%d\n", count);
11. rewind(stdin);
12. getchar();
13. return 0;
14. }

程序5-7-8

程序的某次运行结果如下：

**12**

上面的scanf函数成功匹配的格式控制字符数是：1

**1 2**

上面的scanf函数成功匹配的格式控制字符数是：2

**c**

上面的scanf函数成功匹配的格式控制字符数是：0

本书约定，程序运行结果中加粗的部分表示用户输入的内容。第一次我们输入的是整数12，相当于是2个字符，但是函数的返回值依然1，这是因为scanf只返回匹配成功的格式控制字符的个数，而不管我们到底输入了多长的数据。第二个scanf语句我们输入了两个数据1和2（空格是分隔符，关于分隔符的概念请参考第2章），这时候scanf的返回值是2，因为两个格式控制字符都匹配成功了。第三次，我们试图用scanf来输入i的值，但是实际输入的时候却是输入的字符c，由于字符c不是一个整数，因此scanf会拒绝接受这个值，这样就会导致格式控制字符匹配失败，所以这里的scanf函数的返回值是0。

在第1章的讲解如何在Visual Studio下创建工程的时候，我们就专门提到过要取消安全开发生命周期检查的复选框。如果没有取消勾选，那么在写程序的时候就不能使用scanf，而必须使用scanf\_s。通过上面这个例子，我们可以看出scanf是一种不安全的函数，并且我们还说过，在用scanf输入数据之后，输入缓冲区里会数据残留。基于安全开发的考虑，微软公司提供了scanf\_s函数，这是一个更为优秀更为安全的输入函数，而C11标准把这个函数引入了标准库。事实上，我们也可以自己编写代码来打造安全的输入函数，我们会在后面的章节展示scanf\_s的用法和自己实现安全输入函数的方法。

至于\_getch、getchar等函数，它们的返回值自然就是一个字符了，这个我们也已经很熟悉了。

在第2章我们介绍过的输出函数中，还有一个putchar函数，我们再来看看它的返回值是什么。

1. #include <stdio.h>
2. int main()
3. {
4. int count;
5. count = putchar('A');
6. printf("\n上面的putchar函数的返回值是：%d\n", count);
7. count = putchar('a');
8. printf("\n上面的putchar函数的返回值是：%d\n", count);
9. getchar();
10. return 0;
11. }

程序5-7-9

程序的运行结果是65和97，我们通过查阅ASCII码表（附录1）可以发现，字符'A'和字符'a'的ASCII码正好就是65和97，因此我们可以得出结论，putchar的返回值是它输出的字符的ASCII码。

练习题

1.参考5.6节的例子，改写程序5-4-4，把里面的各个函数分别写到不同的文件中。要求使用头文件。

2.已知在包含math.h之后，可以使用这个函数：double sqrt(double);。在传入一个实数作为参数之后，该函数将会返回参数的平方根。编写程序，提示用户输入三个数a、b、c分别作为一元二次方程ax2+bx+c=0的三个系数，然后求出该方程的解。提示：求根公式是：[-b±√(b²-4ac)]/2a。当b²-4ac<0时方程无解，当b²-4ac=0时方程有唯一解，当b²-4ac>0时方程有一对解。

3.已知某数列an，an=2\*a(n-1)+1，并且a1=1。编写程序，用函数递归的方法来求出a10的值。

4.在包含stdlib.h之后，使用exit(0)可以直接以返回值0退出程序，使用system("cls")可以用来清屏，system("shutdown –s –t 1000")可以在1000秒后关闭计算机（1000可以改成其它数字），system("shutdown –r –t 1000")可以在1000秒后重启计算机，system("shutdown –a")可以取消之前下达的关机或重启命令，而system("shutdown –l ")则可以立即注销计算机。编写程序，显示菜单，分别为：关闭计算机、重启计算机、注销计算机、取消操作、退出程序，然后接受用户输入并执行相应的命令。要求用一个函数接受用户输入，另一个函数执行相关操作。