第5章 函数机制（Function Mechanism）

早在 C 语言中，就确立了函数在程序设计中的地位。函数是 C 也是 C++程序结构中的基本单位，只不过 C++函数在内涵上对 C 进行了诸多扩充。从结构上或者从本质上说，设计程序就是设计函数。从最初的 main 函数启动，到程序运行结束，无不是函数在起作用。函数的不同组织方式就形成了不同的程序设计方法。在数据还没有太复杂、太混乱、太庞大之前，玩弄函数的设计，“大了分，小了合”，就是基于函数的程序设计了。当然，在 C++中，函数与过程的相似性在概念上大可以玩一些文字游戏，但到了 5.3 节看透了函数之后，对 C++函数的本质应不会有什么悬念了。

也就是说，是函数构建了 C 或 C++程序。C++的函数其实是一个过程体，有些函数具有输入参数和返回类型，表现得像数学函数那样，有自变量和函数值，甚至还可以对其进行递归描述；另外一些没有返回类型，没有调用的数值结果，“什么也没有得到”，表现得像一个纯粹的过程。在 C++中，函数的作用并不只是为了对于给定的数据，计算后返回一个函数值那样完成一个预定的目标。C++语言的设计，其函数参数的意义并不仅仅是一个类型的变量值，很多是指向数据流的一个窗口，或者数据集合以及一组相关操作，即对象。完成了一个过程，也就完成了数据集合的处理。所以 C++中的函数可以涵盖一切数据操作过程，从这个意义上说，C++的函数是涵盖更广的过程，基于函数的程序设计就是基于过程的程序设计。

正因为函数在 C++中是如此重要，所以函数的使用也必须十分规范。函数使用的规范性体现在函数参数的性质上，体现在对待函数参数的传递规则和返回类型与返回值的审查上，体现在函数名字的识别原则上，体现在函数体编译的效率选择上，还体现在函数体对数据的访问权限上。

5.1 函数性质（Function Character）

❑ 5.1.1 函数的形态（The Function Forms）

数学函数描述自变量与因变量之间的多对一的数量关系，还研究其各种性质，包括可逆性。在实际应用中，强调求值趋向，且没有任何副作用，结果亦可重复计算获得。而C++函数描述操作序列，虽有求值表现，但更强调过程性，因而就强调构建过程的结构框架，即模块结构。在语言设计的时候，套用了函数的原始意义——求值，但却与函数的范畴相去甚远。它可以表现为数学意义上的函数，也可以表现为纯粹的过程操作，更可以表现为结果的多元化描述。由于它可以通过模块结构对过程进行分解，所以实际上具备了描述数学模型的条件，它甚至还可以有副作用，有意无意地让结果不可重复，以使函数的设计带有高度的技巧性和灵活性。那么，它在形式上是如何表现的呢？

有些函数具有求值表现，即有返回值。这些函数可以没有输入参数，也可以有一个或者多个。例如，数学函数为下列形式：

$$f(x)=\begin{cases} -x & x<0 \\ 2 & x=0 \\ x^2 & x>0 \end{cases} \qquad -\infty<x<\infty$$

其等价的C++函数描述为：

```
long double f(long double x){
  if(x < 0) return -x;
  if(x > 0) return x*x;
  return 2;
}
```

不过要注意的是，数学函数的自变量定义域为±∞，而C++函数中的long double型参数表示范围为$\pm 1.2\times 10^{4932}$。事实上，因为值域也为long double型，所以参数的表示范围在大于0时为函数值的平方根：

$$-1.2\times 10^{4932}\leqslant x\leqslant 1.1\times 10^{2466}$$

当然还要注意，实数与long double型数在精度上存在着差异。这些都是编程语言在描述能力上的局限性造成的。

函数还可以进行递归描述（☞CH5.6）。

函数可以没有参数或者有多于一个的参数。例如，求随机数的函数“int rand();”没有参数，但可以获得非负的整数范围内的随机数。该函数属于C库函数，它取系统中的当前时间与内部自身的一个素数加工成一个伪随机数，作为函数值。而求矩形面积的函数“double area(double width, double length);”显然有长度和宽度两个double型参数，并返回double型值，其为典型的多对一f(a,b)形式的数学函数。

另外一些函数没有“求值”表现，没有返回值，即返回类型用 void 描述。调用的结果“什么也没有得到”，表现得像个纯粹的过程。例如，输出一些字串的函数为：

```
void print(){
  cout<<"unsigned int f(unsigned int n){\n"
      <<"  if(n==1 || n==2) return 1;\n"
      <<"  return f(n-1)+f(n-2);\n"
      <<"}\n";
}
```

该过程不需要参数，也没有返回值。也有一些表现为过程的函数，具有参数。例如，延迟 n 秒钟的函数：

```
void delay(int n){
  for(int i=0; i<n; ++i)
    for(int j=0; j<100000000; ++j);  //该循环约耗时 1 秒
}
```

C++的函数形态，无论像数学函数，还是像纯粹的过程，都可以选择有参数或者无参数。所以 C++函数形态分为四类：

```
返回类型 func(参数列表);
返回类型 func();
void func(参数列表);
void func();
```

有返回类型的函数可以参加表达式运算，或者直接赋值给对应类型的变量，构成表达式语句。例如：

```
double s = sin(b)+1;     //s 定义语句中的函数调用
s = cos(c)/2;            //s 赋值语句
```

无返回类型的函数不能以值的形式赋给其他变量或者参加运算，其调用只能独立构成一条语句。例如：

```
print();             //ok
int t = print();     //错
```

需要时，有返回类型的函数也可以像无返回类型的函数一样单独构成语句。例如：

```
char s1[30];
char s2[] = "hello";
strcpy(s1, s2);         //单独构成语句，完成复制工作
cout<<strcpy(s1, s2);  //复制，并将结果输出
```

❑ 5.1.2 函数黑盒（Function Blackbox）

函数是独立的。正如其名，它反映了一个过程的功能（function）性。从理论上说，函

数只对输入（参数）、输出（返回值）负责，计算被封闭在黑盒中进行。函数调用完后，一切现场恢复如旧，因此也意味着函数调用操作的可重复性，其结果不随调用的时间、地点而转移，如图 5-1 所示。

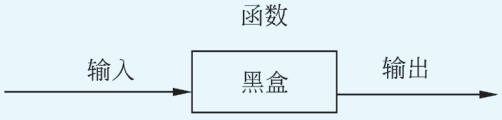

图 5-1　函数黑盒

函数的黑盒性还指只知道其外在的功能而不知道其内部的实现细节。例如，妈妈叫儿子去买酱油，可是，妈妈却唠唠叨叨地交代，从门口出去，往右，第二条街向左拐，斜对面的门口有棵桂花树的那家杂货店，不要到别的店去买。这就限制了儿子的行事自由和选择权。本来他完全可以到更近的一家小超市去买，虽然要穿过一条马路，但却近好多，或者价格更便宜，妈妈也许孤陋寡闻或者不想让儿子冒险穿马路。总之，儿子已经缺失了“黑盒性”。缺乏黑盒性带来的后果是儿子只能为妈妈买酱油，而不能为其他人买酱油，因为买酱油的功能被妈妈所捆绑，不适合其他地点的其他人的买酱油要求。受妈妈“设计思想”的束缚，并非最佳而耗时良久。而函数黑盒性带来的是实惠性，使得函数开发各负其责，程序员无须关心别人开发的函数细节，而无障碍地使用其函数。一人开发，人人适用。所以程序员总是关注拿来使用的函数的外在功能。而且，程序设计阶段对函数的描述是以函数的黑盒性为出发点的。例如，函数描述包括下列内容：

- 函数名及函数访问修饰；
- 输入参数列表；
- 输出参数列表以及返回类型；
- 函数功能描述。

例 5-1　有两种重量的盒子，A 种是 n 斤，B 种是 m 斤。运输这两种盒子的单位成本假设都是 10 元钱。每次给定 A、B 的斤数 n 和 m，返回运输 nm 斤的总成本（元）。

一方面，对于调用该盒子的函数（假设为 cost），无须关心其具体的实现，需要用时，直接给出 n 和 m 两个数，调用 cost(n,m)，就得到其总成本了。

另一方面，函数 cost 在实现时，如果运输 A 种盒子，则总成本为 m*10，如果运输 B 种盒子，则总成本为 n*10，所以实现者可以用两种方法来实现：

```
unsigned int cost(unsigned int n, unsigned int m){
  return n*10;   //运输 n 次 m 斤
}
unsigned int cost(unsigned int n, unsigned int m){
  return m*10;   //运输 m 次 n 斤
}
```

这两种方法的好坏取决于 n 和 m 的值，比较随机，更好的方法是：

```
unsigned int cost(unsigned int n, unsigned int m){
  return (n>m? m:n)*10;   //保证运输次数最少
}
```

函数总有各种各样的实现，就好像人有各种各样的差异一样。所调用的函数性能越好，程序的总体质量也越高。C++基本库中的函数都是高度精练的，所以能调用C++库中的函数就尽量调用，库中没有的函数才自己手写，这样可以减少程序员自己的编程量，而且总体质量也高。

5.1.3 传值参数（Value-Passed Parameters）

函数通过参数传递输入数据，参数通过传值机制实现。所谓传值，即在函数被调用之时，用克隆实参的办法创建形参。例如：

```
void f(Type a);   //Type 为任意数据类型，a 为形参
void g(){
  Type x;
  f(x);           //此处 x 为实参，相当于定义"Type a = x;"
}
```

克隆实参就是用实参创建形参而实参本身没有任何改变。

当上面的x的类型与a的类型不一致时，就要看两个类型是否相容，如果相容，那么，在调用时，会将类型作隐性转换而完成传值；如果不相容，则作为编译错误而绝不妥协。这种类型匹配的检查，给程序员带来的是福音，可以让程序员早早地发现潜在的错误。例如：

```
struct S{int a,b;};
void f1(S a);
void f2(int b);
void f3(int* c);
void f4(int& d);
//---------------------------
void h(){
  S x = {1,2};
  f1(x);     //ok: S a=x;
  f1(2);     //错: S a=2;类型不匹配
  f2(23);    //ok: int b=23;
  f2(2.5);   //ok: int b=int(2.5); //类型转换，精度受影响
  f2(x);     //错: int b=x;类型不匹配
  int e = 3;
  f3(&e);    //ok: int *c=&e;
  f3(e);     //错: int* c=e;类型不匹配
  f4(e);     //ok: int& d=e;
  f4(&e);    //错: int& d=&e;类型不匹配
}
```

函数是一个计算单位，它可以返回也可以不返回值完成一个计算。有许多任务光靠复制几个参数数据做输入是不够的，而且数据还要返回。返回总是只有一个数据项。例如，

输入数据为 100 个整数，让其排序输出。总不至于用 100 个参数吧，不敢想象下面这样的函数声明：

```
void sort(int a001,int a002,int a003,···,int a100);
```

就算可以，那 1000 个、10 000 个整数的排序呢？何况，初级的排序操作需要线性存储结构的下标操作，如向量。那么，如果传递给函数一个数据整体的复制品呢？例如：

```
vector<int> mySort(vector<int> v);

void f(){
  int a[]={3,5,7,1,8,4,9,2,6,0};
  vectot<int> va(a,a+10);
  vector<int> vb = mySort(va);   //vector<int> v=va;
...
}
```

那也不行！函数确实可以执行 mySort 排序任务，但那是施加在数据复制品上的操作。要完成任务，还必须将这些数据返回，这大批数据随着函数调用一来一往，函数工作的效率马上降低，见图 5-2，那就不是 C++的套路了。

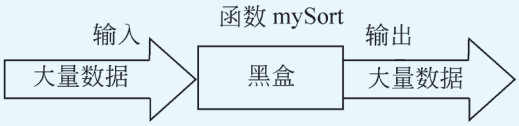

图 5-2　大数据流量的函数

至于数组，还不能实现这种传输呢，因为它没有数据整体复制能力，只能通过指针传递（☞CH5.2.1）。

诚然，该方法可以实现数据的排序。但翻遍 C++编程方法的经典书，你不会找到用这种方法编程的，除非是想测试用这种方法所带来的负面效应是多少。

5.2　指针参数（Pointer Parameters）

❑ 5.2.1　指针和引用参数（Pointer & Reference Parameters）

例 5-1 中有效的方法是传递给 mySort 排序函数以一个数组指针或者容器引用。这样，传递的是指针和引用值，也就无须大块数据的挪动了。然后，通过指针和引用的间接访问实现数据的操作。这时候，其实是赋予了函数去操作“异地”数据的权力。异地指非本函数数据区或者称非栈顶函数区（☞CH5.3）。为了完成计算任务，这个权利是必要的。因为这个原因，函数可以大言不惭地说，可以高效地完成数据操作任务了。而且正因为函数的这种能力，才使得 C++程序奠定了以函数为基石的结构。进而，函数作为过程的拓展，基于函数的程序设计就称为基于过程的程序设计了。正因为这个权利，函数调用才这么灵

活、高效，无须承受大块数据传递的沉重负担。例如，传递数组的方法：

```
void mySort(int* b, int size);
void f(){
  int a[] = {3, 5, 7, 1, 8, 4, 9};
  mySort(a, sizeof(a)/sizeof(a[0]));   //int* b=a;
  //...
}
```

数组是不能整体直接复制的，即

```
int a[10];
int b[10] = a;        //错
b = a;                //错
```

数组只能通过传递数组起始地址，达到使用该数组的目的。正因为如此，数组作为函数参数传递，实际上只是把数组的起始地址传给了函数，因此为了能有效地使用数组，还必须给数组传递一个元素个数。在 f 函数中，创建了数组 a，将其传递给 mySort 函数。由于知道传递的数组名只是一个数组的起始地址，因此，mySort 的形式参数有两个。

函数的数组传递的实质，是数组地址的克隆。当初，C 语言在设计的时候，看到了数据传递实在没有传递整个数组空间的必要，因而弄了一下巧，结果其效率享誉了全世界。就好像我要让你用车，何必把车扛过来呢，太重啊☹，只需将钥匙交给你就得了。

然而，这种编程方法，还是请不要学，至少是初学时不要学，因为它不是理想的编程方法，一维数组的传递有一些褒贬不一的说法，但二维数组的传递绝对是一边倒的共识。你可能会看到一些 C 或 C++语言的书对多维数组的描述（二维以上的数组，称为多维数组），若真的胆敢传递多维数组给函数，那么，一切美感将会消失殆尽。所以，等到琢磨透了编程，成为了高手，再研究这样的编程手法吧。

然而，传递指针和引用的特性作为一把双刃剑还是被专业程序员所看好。本质上，初级程序员与专业程序员的差别就可以从使用指针和引用参数中看出。由于参数值可以通过传递指针和引用获得，所以它意味着函数可以访问不属于自己管辖的数据，如图 5-3 所示。

很显然，只传递数组的地址，明显减轻了参数传递过程中的数据复制量。但是要看清，排序是在数组 a 上操作的，数据的改变不是在 mySort 函数数据区内，等到 mySort 返回之际，数组 a 的内容发生了改变，而参数 b 本身的值却没有发生任何改变（还是指向数组 a）。这便是函数运行的结果！

这意味着函数运行的结果并不一定要用返回类型规定的返回值反映，函数参数也能起到返回结果的作用。因为最终，函数结果要返回到调用函数中，而函数的指针和引用参数，指示着调用函数的数据内容已经发生变化，这不就是运行所要达到的目的吗!?

所以，函数可以传递多个输入参数，同时也可以让多个数据处理的对象发生变化。输出因而也可以呈多元化而非返回值一个形式。

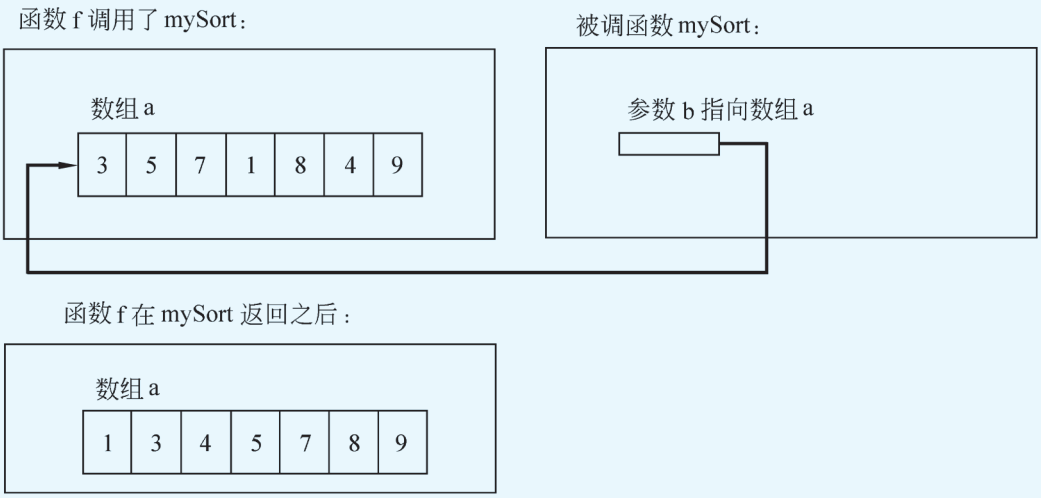

图 5-3　数组传递

例 5-2　在一个矩阵中找含有–1 元素的向量，若找到，则找到的结果存放在参数 v 中，返回 true；否则，没有结果，且返回 false。程序所用的数据放在文件 abc.in 中。文件中第 1 行放向量的个数 n，以便读入操作。接下来 n 行即 n 个长短不一的向量。程序代码如下：

```
(1)  //=====================================
(2)  //f0501.cpp
(3)  //向量参数传递
(4)  //=====================================
(5)  #include<vector>
(6)  #include<iostream>
(7)  #include<fstream>
(8)  #include<sstream>
(9)  using namespace std;
(10) //-----------------------------------
(11) typedef vector<int> VI;
(12) typedef vector<VI> VVI;
(13) void print(const VI&);
(14) void input(VVI&);
(15) bool findVec(const VVI&, VI&);
(16) //-----------------------------------
(17) int main(){
(18)   VVI matrix;
(19)   input(matrix);
(20)   VI vec;
(21)   if(findVec(matrix, vec))
(22)     print(vec);
(23) }//-----------------------------------
(24) void print(const VI& v){
(25)   for(int i=0; i<v.size(); ++i)
(26)     cout<<v[i]<<" ";
```

```
8
12 35 77 45
1 2 3 4 5 67 9
12 11 8 9 0
1 -11
2 3 4 5 6 7 8 9 0 -4 5 -1
5 4 3 2 1 6 7
13 34 56 78 99 98
11 111 11 11 11
```

abc.in

```
(27)   cout<<"\n";
(28) }//-----------------------------------
(29) void input(VVI& m){
(30)   ifstream in("abc.in");
(31)   int n, t;  in>>n;
(32)   m.resize(n);
(33)   for(string s; n-- && getline(in, s); )
(34)     for(istringstream sin(s); sin>>t; m[m.size()-n-1].push_back(t));
(35) }//-----------------------------------
(36) bool findVec(const VVI& matrix, VI& v){
(37)   for(int i=0; i<matrix.size(); ++i)
(38)   for(int j=0; j<matrix[i].size(); ++j)
(39)     if(matrix[i][j]==-1){
(40)       v = matrix[i];
(41)       return true;
(42)     }
(43)   return false;
(44) }//===================================
```

```
E:\ch05>f0501↙
2 3 4 5 6 7 8 9 0 -4 5 -1
```

程序中，由于向量的参数传递，才使得 main 函数能通过函数调用而显得抽象。细节部分都可以隐藏到所调用的函数中，甚至文件操作都可以免做。input 函数负责矩阵的输入，由于文件的向量个数在开始就标明了，所以根据向量个数先用 resize 初始化矩阵中的向量个数，能避免就尽量避免使用 push_back 操作。但每个向量中，其元素个数不一，故对每个向量的初始化还要采用 push_back 操作，而且从文件中逐行读入 string 对象中，再行分解读入。

findVec 是具有两个参数的查找函数，如果找到有关向量，就将该向量复制到参数 v 中，并返回 true，否则返回 false。该函数没有办法返回所找到的向量，因为没有办法确认找不到的情况。引用参数在这里起了很重要的作用，使得在函数 findVec 中给该参数赋值能够在主函数中获得修改信息。程序在必要的地方出现了 const 限定参数的操作，更多的是让其他程序员（在这里是读者）看清该函数的功能。

该程序演示了不同于数组传递的另一种指针参数传递。函数的指针或引用传递得益最大的还是大对象（像向量和矩阵这么大）的传递。虽然函数机制能够办到将一个大对象传递给调用的函数，但多数情况下这不是明智之举。在面向对象的编程中，几乎都是将对象以指针或引用的方式传递的，只有这样才有效率可言。所传递的向量，自身含有元素个数的信息，所以就免去了传递第二个参数的必要了。

C++不得不包容函数机制的指针传递方式，而且还享受了其高效，但同时，也要付出函数已经缺失了黑盒性的代价。如果还要坚持传递数组，那么还要承受缺乏类型检查严密性和操作不便的代价。

❑ 5.2.2 函数的副作用（Function's Side Effect）

我们再讨论函数的超限权力问题。函数参数传递指针和引用是一把双刃剑，它的负面作用是破坏非本地数据，破坏模块性。

函数的黑盒性不与外界发生额外沟通，只依赖输入参数，只送出返回结果，因此它是结果可重复的计算过程。但是指针和引用的参数传递向我们展示了函数可以访问非本地数据的途径，从而破坏了函数的黑盒性。

例 5-3 两个整数向量，元素个数相同，试做其加法，其代码如下：

```
(1)  //=====================================
(2)  //f0502.cpp
(3)  //函数设计的随意性
(4)  //=====================================
(5)  #include<iostream>
(6)  #include<vector>
(7)  using namespace std;
(8)  //-------------------------------------
(9)  void print(vector<int>& a){
(10)   for(int i=0; i<a.size(); ++i)
(11)     cout<<a[i]<<" ";
(12)   cout<<endl;
(13)  }//-------------------------------------
(14)  vector<int> add(vector<int>& a, vector<int>& b){  //用 a 存放结果
(15)   for(int i=0; i<a.size(); ++i)
(16)     a[i] += b[i];
(17)   return a;
(18)  }//-------------------------------------
(19)  int main(){
(20)   int aa[]={2,3,1,2,3,2,1}, bb[]={5,3,1,1,6,2,2};
(21)   vector<int> a(aa,aa+7), b(bb,bb+7);     //复制数组给向量
(22)   vector<int> c = add(a, b);
(23)   print(a);  print(b);  print(c);          //查验向量内容
(24)  }//=====================================
```

```
E:\ch05>f0502↙
7 6 2 3 9 4 3
5 3 1 1 6 2 2
7 6 2 3 9 4 3
```

程序是将两个初始化了的数组赋给两个向量 a 和 b，然后调用 add 相加，并输出这两个向量以及结果向量。设计函数是很机械的，只要输入数据，按照功能要求、性能要求，满足它就行，本例中的 add 函数就是一个典型的例子。说实在话，设计者还为自己第 16 行的“+=”操作符而得意呢，因为针对对象的操作的效率不差于“a=a+b”，而且，设计中简化了操作，不建临时向量，一切以效率为第一要素，只要完成计算就行的思想贯彻得很好。

不过，main 函数可傻眼了，它调用的 add 函数结果虽然对了，但是原始数据的向量 a 却被破坏了。这就是函数运行所带来的副作用！就好像心脏病患者，吃了药之后，心脏病控制住了，但药物所带来的副作用，使关节炎却意外地发作了☺。

独立性是 C++对函数的定位，可能设计函数者远在天边，通过 Internet 与软件工程师（程序设计的规划者）保持联系。软件工程师与程序员的沟通不可能在调用 add 函数的环境上。事实上，调用环境是千变万化的。软件工程师也不可能对 add 函数的设计指手画脚，就像前面的比喻：妈妈叫儿子买酱油般的唠叨。软件工程师只能通过函数声明和相关的性能要求描述，框定函数设计要求。而程序员自有一片任意发挥的天空，只要满足了软件工程师的要求就行。

问题出在规定函数参数传递的声明上，传递引用给了函数超限的权力，函数循着传递的引用名既读又写地访问了引用的空间，而那一片引用空间并不是函数所拥有的。因此，C++语言的限制手段就是在指针和引用参数上加 const 修饰，以此限制函数体中对参数的写操作。即

```
vector<int> add(const vector<int>& a, const vector<int>& b){
  for(int i=0; i<a.size(); ++i)
    a[i] += b[i];            //错: a[i]不能做左值
  return a;
}
```

声明中对参数加上 const 之后，负责 add 函数设计的程序员在编译报错之后明白了，既要完成加法，又不能在原始数据上“打草稿”。如果还是那样设计，编译这一关就休想过！那只能另辟向量空间的蹊径以存放中间结果了，于是 add 函数做这样的修改：

```
vector<int> add(const vector<int>& a, const vector<int>& b){
  vector<int> c(a);
  for(int i=0; i<a.size(); ++i)
    c[i] += b[i];            //ok
  return a;
}
```

当然，设计 add 的程序员还是有很多发挥的空间，与下面这种设计相比，上面代码的性能更好。程序员之间的编程功底还是看得出来的，想想为什么？

```
vector<int> add(const vector<int>& a, const vector<int>& b){
  vector<int> c(a.size());
  for(int i=0; i<a.size(); ++i)
    c[i] = a[i] + b[i];
  return c;
}
```

更差一点，下列代码虽可完成加法任务，但全然没有 C++的性能优势了：

```
vector<int> add(const vector<int>& a, const vector<int>& b){
  vector<int> c;
```

```
    for(int i=0; i<a.size(); ++i)
      c.push_back(a[i] + b[i]);
    return c;
}
```

参数的 const 声明框定了传递的参数只能以形参规定的原则操作，所以 a 向量再也不能在 add 函数中被修改了，即在表达式中不能作左值。当模块设计或结构设计与编程设计相分离时，所规定的函数参数传递的常量性就能规范编程行为，不致产生不良副作用，达到安全编程的目的。

C++在函数参数传递时，只进行实参与形参匹配的类型检查，并不规定指针的取值范围，而引用在参数一旦传递了之后，就捆绑在某个对象上了，所以相对安全些。因而可以说，指针是灵活性更大的实体。有许多专门论述指针的书和文章，认为指针是程序不可靠的真正原因，当然高级程序员会反驳说，妙就妙在指针！读者将会在下一节中看到 C++程序运行中的布局和函数数据区的功用，指针的踪迹也会更多地曝光，并将继续描述函数的副作用，以及对指针下一些定论。

5.3 栈机制（The Stack Mechanism）

❑ 5.3.1 运行时内存布局（Runtime Memory Layout）

一个程序要运行，就要先将可执行程序文件装载到计算机的内存中。装载是操作系统掌控的，一般而言，操作系统将程序装入内存后，将形成一个随时可以运行的进程空间，该进程空间分四个区域，如图 5-4 所示。

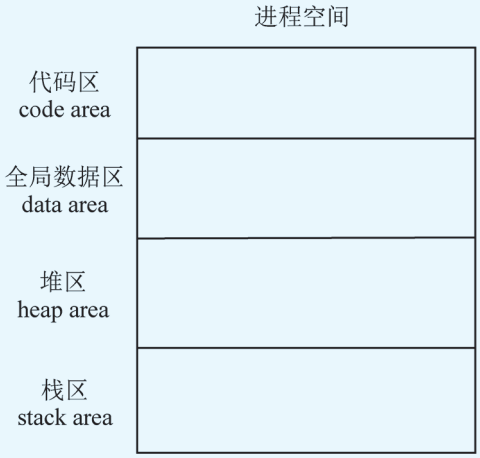

图 5-4 运行中的内存布局

一个运行的程序在内存中表示为这四个空间区域，其中代码区存放程序的执行代码。所谓执行代码，就是索引了的一个个函数块代码，它由函数定义块编译得到。

全局数据区存放全局数据、常量、字串字面量、静态全局量和静态局部量。

堆区存放动态内存，供程序随机申请使用。

栈区存放函数数据区（即局部数据区），它动态地反映了程序运行中的函数状态，其运动轨迹正好用来观察函数的调用与返回，可借此研究函数机制。

5.3.2 栈区（The Stack Area）

栈是一种数据结构，它的工作原理就像叠盘子一样，最先叠上去的盘子要等到随后叠上去的盘子都拿走了才能拿到，而最后叠上去的盘子则可以首先拿到。

C++的函数调用过程需要初始化和善后处理的环节。函数调用的整个过程就是栈空间操作的过程。函数调用时，C++做以下工作：

（1）建立被调函数的栈空间，栈空间的大小由函数定义体中的数据量大小决定；

（2）保护调用函数的运行状态和返回地址；

（3）传递参数；

（4）将控制权转交给被调函数；

（5）函数运行完成后，复制返回值到函数数据块底部；

（6）恢复调用函数的运行状态；

（7）返回调用函数。

调用一个函数，可以看作是一个栈中元素的压栈和退栈操作。然而，压栈之后的函数运行可能导致其他函数被调用。因此，程序运行表现为栈中一系列元素的压栈和退栈。最初，操作系统将 main 函数压入栈中，标志着程序运行的开始，等到最后 main 函数退栈，程序也就运行结束了。

例 5-4　下面的 f0503.cpp 程序在 main 函数中调用一个 funcA 函数，该函数又调用了一个 funcB 函数，得到如图 5-5 所示的栈结构布局。

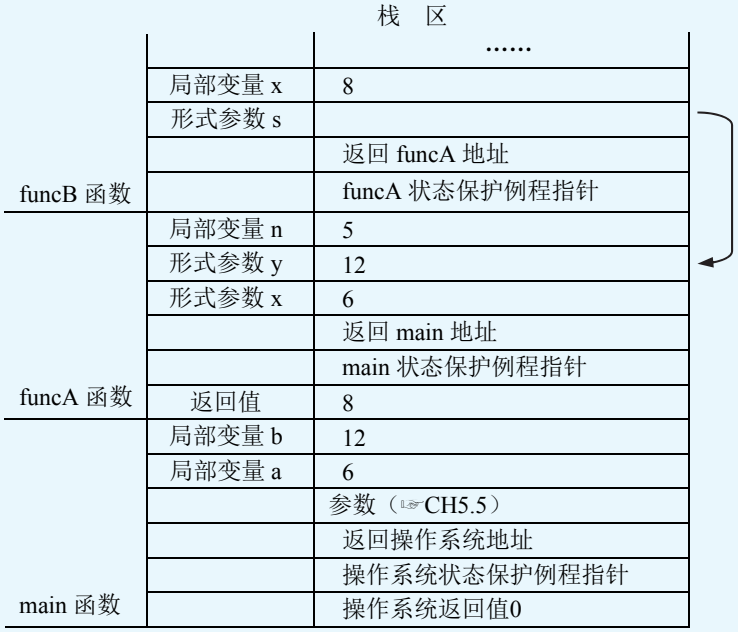

图 5-5　运行中的函数栈结构

```
//=====================================
//f0503.cpp
//栈区运动的演示程序
//=====================================
int funcA(int x, int y);
void funcB(int& s);
//-------------------------------------
int main(){
  int a=6, b=12;
  a = funcA(a,b);
}//-------------------------------------
int funcA(int x, int y){
  int n=5;
  funcB(n);
  return n;
}//-------------------------------------
void funcB(int& s){
  int x=8;
  s = x;
}//=====================================
```

注意，该程序没有输出语句，没有运行结果，仅用来演示栈区结构。

首先要说明的是，栈机制的实现是随编译器的不同而不同的，我们不是学习编译实现，而是学习函数的栈机制原理。

main 函数被操作系统调用时，在栈中安排了返回值位置，操作系统状态保护例程指针是指完成恢复工作的函数地址，该函数或为硬件设置，或为操作系统专用的函数。在 main 函数返回时，取其地址调用之；然后再取操作系统调用本 main 时的地址，返回调用点。main 函数也有参数（☞CH5.5）。

main 函数在进行变量 a 的赋值运算时，随之就调用了 funcA 函数，以期获得函数值。调用时，funcA 函数安排了返回值空间，设置了恢复 main 函数状态的函数指针（☞CH5.3）和返回 main 调用点地址。main 函数状态就是调用 funcA 的瞬间，CPU 中的各个寄存器状态的总和。main 随后便进行参数传递，即分别将实参 a、b 的值传给 funcA 的形参 x、y。funcA 又建立起自己的局部变量 n，然后调用了 funcB。

funcB 是一个没有返回类型的函数，因此，创建 funcB 栈元素时，没有给 funcB 预留返回值空间。funcB 设置了返回 funcA 状态的恢复函数指针和返回地址后，进行了实参到形参的拷贝，该拷贝实际上建立了形参 s 引用实参 n 的对应关系，以至于修改 s 为 8 时，相应的 n 值也跟着为 8。当函数 funcB 返回时，其栈区结构示意见图 5-6。

栈 区

		……
		8
		返回 funcA 地址
		funcA 状态保护例程指针
	局部变量 n	8
	形式参数 y	12
	形式参数 x	6
		返回 main 地址
		main 状态保护例程指针
funcA 函数	返回值	8
	局部变量 b	12
	局部变量 a	6 <= 8
		参数（☞CH5.5）
		返回操作系统地址
		操作系统状态保护例程指针
main 函数		操作系统返回值 0

图 5-6　funcB 返回时的栈结构

funcB 退栈后，数据仍残留在栈中，而函数结构已经消失，不再有 funcB 这回事了，有的只是 funcA 现场。待到 funcA 返回后，其栈区结构示意见图 5-7。

栈　区

		……
		8
		返回 funcA 地址
		funcA 状态保护例程指针
		8
		12
		6
		返回 main 地址
		main 状态保护例程指针
	返回值	8
	局部变量 b	12
	局部变量 a	6 <= 8
		参数（☞CH5.5）
		返回操作系统地址
main 函数		操作系统状态保护例程指针
		操作系统返回值 0

图 5-7　funcA 返回时的栈结构

funcA 的返回值是在 return 语句中计算出来而存放的位置，它在 main 函数中参加运算，所以事实上作为 main 函数的临时空间使用。系统将返回值 8（函数 funcA）最后赋给了局部变量 a。当 main 函数返回时，除了栈中一片狼藉，一切都已无意义。操作系统将这块区域收回，以便下次有新的程序运行请求时，可以分配之。

❑ 5.3.3 局部数据的不确定性（Uncertainty of Local Data）

我们已经知道栈中数据即为局部数据。这里我们无法猜测的是，操作系统启动 main 函数时，栈中是否已经打扫干净。main 返回的时候并没有将数据清除啊！因此我们假定局部变量只要未经初始化，其值总是不确定的。事实上，测试结果证实了这一点。例如：

```
//=====================================
//f0504.cpp
//局部数据不确定
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
void f(){
  int b;
  cout<<"B=>"<<b<<endl;
}//-------------------------------------
int main(){
  int a;
  cout<<"A=>"<<a<<endl;
  f();
}//=====================================
```

```
E:\ch05>f0504↙
A=>8804248
B=>2788048
```

为什么操作系统这么懒，连这点工作都不愿做，害得程序员一个个都要提心吊胆地给局部变量赋初值呢？那是因为任何软件设计都有它的合理性。分配给申请运行的程序的内存空间是属于用户（在操作系统眼里）的，用户的内存怎么用，栈空间的沉浮变化，操作系统管得到那么宽吗？既然管是毫无意义的，对于“清扫”返回的和分配出去的用户空间同样是“狗咬耗子，多管闲事”。

❑ 5.3.4 指针作祟（The Menacing Pointers）

我们再次看到了，指针通过间接访问改变了上层函数的数据。其实，它不但可以改变上层函数的数据，任何本程序的数据都可以改。

因为编译器检查的是类型匹配，它的目的是为杜绝编程错误，并有意无意地防止越

权，至于数据值是什么，只要在表达范围内，编译器是不管的。指针可以赋予任何值，只要操作系统不介意的话。但事实是，操作系统的内存管理把指令和访问数据的地址都管了起来，一个进程只能访问自己的程序空间，不能访问其他的进程，更不能访问操作系统区域的数据。

那么，通过数据指针（相对于函数指针☞CH5.4），能够给程序造成什么样的破坏，又是如何破坏的呢？

在程序内存布局中我们看到，程序运行有四块区域，除了代码区域，其他都是数据区域。对这些数据区域，理论上，指针可以通过强制类型转换的方法绕过编译的检查而访问。最原始的方法是，知道内存区域的直接地址，规定一个访问的类型，就可以通过指针间接访问读出或改变其值了。例如，已知全局数据区某个地址上的整数是我觊觎已久的数据，代码如下：

```
//=====================================
//f0505.cpp
//指针的强行访问
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int a=5;
int b=6;
//-------------------------------------
int main(){
  int* ap=(int*)4202660;
  *ap=8;
  cout<<a<<endl;
  cout<<int(&b)<<endl;
}//=====================================
```

```
E:\ch05>f0505↙
8
4202664
```

先声明，该程序不可移植，运行结果也不可重复。程序中，全局变量 a（☞CH7.3）的值通过非正规途径的访问而改变了。它意味着，只要有意想让程序运行不正常，真是太简单了。

只要没有指针和引用，尤其是指针，函数就不可能去修改其他地方的数据，也就不会引起副作用。因此，当引用和指针通过参数传递时，可能会给函数带来副作用，而通过 const 修饰，可以抑制其副作用。但是指针值的任意性使得虽然是局部指针也能突破函数的框框去访问非本地数据，而使函数蒙受副作用的冤屈。指针的无法无天，是函数副作用的根源。

当看到了指针丑陋的一面时，我们就理解了为什么 Java 中没有指针。编程本是要让程序做想要做的事，不可控制的指针怎么能被人类容忍呢？！虽然引用也可以间接访问，但引用一旦创建就与某个对象捆绑在一起了，这个性质使得在编程中不知可以少犯多少

错误而又不失程序的灵活性。然而在一些高级程序设计的场合，实现一些函数指针的时候，或者在一些低级程序设计的场合，不顾一切地以高效来对可靠性进行冒险的时候，玩指针还是饶有兴趣的。

从另一个角度讲，程序设计语言是帮助程序员设计出为人类服务的程序的，因此，它想方设法提高其编程方便性、容易查错性，讨好人类。但并没有想方设法不让人们钻空子，若存心要搞小动作的话，那是一定可以搞的，中国古代说的“防不胜防”就是这个道理。当你学习了函数指针这一节后，会更加迷上 C++的神通。

5.4 函数指针（Function Pointers）

一个运行的程序在内存中的布局，分为四个区域：Data Area、Heap Area 和 Stack Area 这三个区域称为数据区域，Code Area 称为代码区域。指向数据区域的指针称为数据指针（Pointer to Data）。指向代码区域的指针称为指向函数的指针，简称函数指针（Pointer to Function 或 Function Pointer）。但要注意，与返回指针类型的函数不同，该函数称为指针函数（Pointer Function）。例如：

```
int* f(int a);
char* copy(char* s1, char* s2);
```

f 函数返回 int 指针，即 int 指针函数，copy 函数为 char 指针函数。运行中的程序，其中的每个函数都存放在代码区，占据着一个区域。故每个函数都有起始地址，指向函数起始地址的指针完全不同于数据指针，函数指针与数据指针不能相互转换，通过函数指针可以调用所指向的函数。

❑ 5.4.1 指向函数的指针（Function Pointers）

1 函数类型

从指针操作使用的地址性质来说，数据指针和函数指针都是一样的。函数指针也是指针，指针作为实体当然要先定义或声明才能使用。函数指针也有不同类型，函数有多少种类型，函数指针就有多少种类型。例如：

```
void f();
int k();
int g(int);
int h(char);
int m(int, int);
```

都是不同类型的函数。

函数是以参数个数、参数类型、参数顺序甚至返回类型的不同区分不同类型的。函数的类型表示是函数声明去掉函数名，所以，上面 f 函数的类型为 void()，同样 m 函数的类型为 int(int,int)。声明一个 int(int)类型的函数 g，就是把函数名放在返回类型和括号之间：

int g(int)。

2 函数指针

声明一个 int(int)类型的函数指针 gp，就是把指针名放在返回类型和括号之间，即

```
int (*gp)(int);
```

注意，上面是定义一个函数指针，不是声明，而且容纳函数指针名的括号不能省，表示“*gp”是一个整体，它描述的是一个指针，有括号和无括号意义完全不一样：

```
int *gp(int);
```

表示声明一个含有一个整数参数的整数指针函数。即等价于：

```
int*  gp(int a);
```

3 函数指针初始化

定义函数指针还可以初始化。如果在定义中伴随着初始化，则应写成：

```
int g(int);
int (*gp)(int) = g;
```

其中 g 应该与指针 gp 所指向的函数类型相同。

当然，函数指针赋值也可以与函数指针定义分开，像下面这样：

```
int g(int);
int (*gp)(int);
gp = g;
```

定义了一个函数指针，就拥有了一个指针实体，一个指针实体的大小跟 int 型实体大小是一样的。

4 函数指针类型检查

函数指针的类型必须接受编译器的检查。gp 指针所指向的函数拥有一个 int 型参数，并返回 int 型值。因此，如果：

```
void f();
void (*fp)();
gp = f;   //错:可疑的指针变换
gp = fp;  //错:可疑的指针变换
```

由于函数指针本身也是一种数据类型，即

```
int (*)(int);
```

是 int(int)型函数的指针类型，其中的“(*)”的括号也是不能省略的，为什么？
因为去掉括号就变成一个整型参数且返回整型指针的函数了，于是可以：

```
void f(int* a, int* b, int(*)(int));
```

所声明的 f 函数，含有三个参数，前两个是整型指针，第三个是函数指针。

5 typedef 函数指针类型

函数指针的定义形式看起来比较复杂，所以通常采用 typedef 简化。例如：

```
typedef int (*Fun)(int a, int b);
```

表示声明了一个函数指针类型。注意，作者习惯用大写字母开头的名字来表示类型名，定义类型名不是定义函数指针实体。因此：

```
int m(int, int);
Fun funp = m;   //ok
Fun = m;        //错:不恰当地使用类型名'Fun'
```

❑ 5.4.2 函数指针参数（Function Pointer Parameters）

数据指针除了进行参数传递外，还承接申请的存储空间、释放空间等；而函数指针则主要是用来进行参数传递的，就像引用一样。

例如，我们看一下函数指针的参数传递工作。在标准 STL 排序算法 sort 中，对于所提供的整数容器 vector，无须提供其他操作就可以顺利完成排序任务。代码如下：

```
int a[] = {33, 61, 12, 19, 14, 71, 78, 59};
vector<int> aa(a, a+8);
sort(aa.begin(), aa.end());
```

但若整数的大小是以各位数字之和的大小决定的，则就不能直接使用 sort 函数排序。需要先定义一个比较函数，然后再对 sort 传递比较函数指针，以让 sort 知道大小关系不是默认的整数值比较，而是根据比较函数判定。可用函数指针调取比较函数进行排序工作。代码如下：

```
(1)  //=====================================
(2)  //f0506.cpp
(3)  //函数指针传递
(4)  //=====================================
(5)  #include<iostream>
(6)  #include<algorithm>
(7)  #include<vector>
(8)  using namespace std;
(9)  //-------------------------------------
(10) int bitSum(int a);
(11) bool lessThanBitSum(int a, int b){ return bitSum(a)<bitSum(b); }
(12) //-------------------------------------
(13) int main(){
(14)   int a[] = {33, 61, 12, 19, 14, 71, 78, 59};
(15)   vector<int> aa(a, a+8);
```

```
(16)    sort(aa.begin(), aa.end(), lessThanBitSum);   //函数名即函数指针
(17)    for(int i=0; i<aa.size(); ++i)
(18)     cout<<aa[i]<<" ";
(19)   cout<<"\n";
(20) }//-----------------------------------
(21) int bitSum(int a){
(22)   int sum=0;
(23)   for(int x=a; x; x/=10)  sum += x%10;
(24)   return sum;
(25) }//===================================
```

```
E:\ch05>f0506↙
12 14 33 61 71 19 59 78
```

第 16 行的 sort 调用，其第三个实参为比较函数名 lessThanBitSum。在参数传递时函数名即为函数指针，正像数组名即为指针一样。sort 的形参为一个对应的函数指针，正像数组传递中形参为对应的数组指针那样。

标准排序算法的使用依赖于容器中元素类型的小于“＜”操作，如果排序的容器中是整数元素，那么小于“＜”的比较判断函数就不需要。因为整数的大小比较操作在 C++中本就具备。否则，sort 函数的调用必须提供一个函数指针作为第三个参数，其参数类型为某个元素类型 T 的 bool(const T&, const T&)。

❑ 5.4.3 函数指针数组（Function Pointer Arrays）

函数指针作为一种数据类型，当然可以作为数组元素的类型。例如，要实现用菜单驱动函数调用的程序框架，则用函数指针数组实现就比较容易维护：

```
(1)  //====================================
(2)  //f0507.cpp
(3)  //函数指针数组
(4)  //====================================
(5)  #include<iostream>
(6)  using namespace std;
(7)  //------------------------------------
(8)  typedef void (*MenuFun)();          //函数指针类型名称
(9)  void f1(){ cout<<"good!\n"; }
(10) void f2(){ cout<<"better!\n"; }
(11) void f3(){ cout<<"best!\n"; }
(12) //------------------------------------
(13) int main(){
(14)   MenuFun fun[]={f1,f2,f3};         //函数指针数组
(15)   for(int choice=1; choice;){
(16)     cout<<"1-----display good\n"
(17)         <<"2-----display better\n"
(18)         <<"3-----display best\n"
(19)         <<"0-----exit\n"
```

```
(20)          <<"Enter your choice:";
(21)      cin>>choice;
(22)      switch(choice){
(23)        case 1: fun[0](); break;
(24)        case 2: fun[1](); break;
(25)        case 3: fun[2](); break;
(26)        case 0: return 0;
(27)        default: cout<<"you entered a wrong key.\n";
(28)      }
(29)    }
(30)  }//====================================
```

程序中（第 8 行）首先定义了一个函数指针类型 MenuFun。如果前面没有 typedef，则后面部分就是一个函数指针定义，所以，正因为有了 typedef，MenuFun 就是函数指针的类型名，可以依此创建函数指针，但其本身并不是函数指针。

在第 14 行，根据 MenuFun 创建了一个函数指针数组 fun，并予以初始化。初始化的每个值都是同类型的函数名，它们在第 9、10、11 行定义。接下来就是循环处理键盘输入，每当输入值为 1、2 或 3 时，显示 good、better 或 best 等字样。同一功能的程序也可以用向量来实现，以下是改用向量实现的代码：

```
(1)  //====================================
(2)  //f0508.cpp
(3)  //函数指针向量
(4)  //====================================
(5)  #include<iostream>
(6)  #include<vector>
(7)  using namespace std;
(8)  //------------------------------------
(9)  typedef void(*MenuFun)();
(10)  void f1(){ cout<<"good!\n";   }
(11)  void f2(){ cout<<"better!\n"; }
(12)  void f3(){ cout<<"best!\n";   }
(13)  //------------------------------------
(14)  int main(){
(15)    vector<MenuFun> fun(3);
(16)    fun[0]=f1, fun[1]=f2, fun[2]=f3;
(17)    for(int choice=1; choice;){
(18)      cout<<"1-----display good\n"
(19)          <<"2-----display better\n"
(20)          <<"3-----display best\n"
(21)          <<"0-----exit\n"
(22)          <<"Enter your choice:";
(23)      cin>>choice;
(24)      if(choice>0 && choice<4) fun[i-1]();
(25)      else if(choice==0) return 0;
(26)      else cout<<"you entered a wrong key.\n";
(27)    }
(28)  }//====================================
```

5.4.4 简略函数指针表示（The Outline of Function Pointers）

1 无名函数与无名函数指针

C++语句：

```
int ();
```

是一个无名函数声明，表示一个函数类型。

```
int func();
```

也是一个函数声明，表示又一个函数类型，其中 func 是函数的名字，凭此名字可以推断其函数类型。

```
int (*)();
```

是一个无名函数指针；

```
int (*pf)();
```

是一个函数指针定义，pf 是函数指针名，该指针将会指向类型是 int()的函数。

2 typedef 函数类型

```
typedef int Func();      //将该函数类型 int()定义为 Func
```

则上面的函数指针可等价定义为：

```
Func* pf;                //不能写为 int ()* pf;
```

3 函数指针的函数（函数指针函数）声明

```
int (*func(int))();
```

是一个指针函数声明，即指针函数*func(int)的返回类型是 int()。也就是：

```
Func* func(int);         //函数指针函数 func
```

由于在定义函数指针的时候，总是要“穿插”在函数类型中，所以编程中，更多的是先定义 typedef 函数类型，再定义函数指针。例如：

```
typedef int(*SIG)();          //指向 int 函数的指针类型 SIG
typedef void(*SIGARG)();      //指向 void()函数的指针类型 SIGARG
SIG signal(int, SIGARG);      //声明一个函数,其返回 SIG,其第二参数为 SIGARG
```

5.4.5 函数指针的意义（The Sense of Function Pointers）

（1）C 语言处理类型不严密，C++必须改变。就像数组参数只能传递以指针，函数参

数也只能传递以函数指针。如果参数中给出了一个函数类型，则自动转换为函数指针。这种现象称为蜕变(decay)。程序 f0507.cpp 中的语句：

```
MenuFun fun[] = {f1, f2, f3};
```

就是因为蜕变才有这样的形式，本应为：

```
MenuFun fun[] = {&f1, &f2, &f3};
```

也使得函数指针调用函数的形式成为：

```
fun[0]();                    //应理解为*fun[0]();
```

（2）可以同理使用函数引用：

```
void g();
typedef void Fun();
Fun& f = g;
f();
```

（3）函数指针使得 C++可以沟通其他语言编写的程序，通过函数指针挂接，方便地将其他语言写就的函数和过程引入 C++中来。

（4）函数指针使程序表现出更多的灵活性。一个函数的函数指针参数取不同值，就可以让该函数引用不同的函数，表现出不同的行为，如程序 f0506.cpp 中的 sort 可以用不同的比较函数规定元素的大小规则。

（5）函数指针也是 C++面向对象编程机制中的重要手段，多态实现的动态绑定技术就是基于函数指针（☞CH12.3.2）。

（6）事实上，因为函数代码是跨进程的，所以通过函数指针可以越过本地进程，通过动态链接库（☞CH13.5.1）的方式，访问共享性质的其他进程（服务器），执行其函数，甚至操作系统函数。这些内容涉及高级编程。

（7）函数指针使得函数的副作用更为恶劣，函数的意义更远离黑盒的初衷，使得函数设计更“变幻莫测”。因为函数通过函数指针参数调用其他的函数，可以在更深远的范围中不可逆地改变环境，使得运算无法严格地重复运行结果。

5.5 main 函数参数（The main's Arguments）

编好了程序，要投入应用，就要启动程序。程序运行是在操作系统环境下进行的。操作系统对所启动的程序也提供一些服务，接受程序启动时的临时输入/输出重定向请求，或者干脆把用户的请求通过参数传递给程序，让程序处理运行要求。

❑ 5.5.1 命令行重定向（Redirecting Command Line）

各种编程在许多情况下都归结为计算，也就是说，根据输入要求进行计算，最后获得

结果。这种强调计算的过程，虽然留下了输入/输出的口子，但往往对输入/输出设备并不明确规定，而是视运行的环境而定。因此，一些程序就灵活地用可重定向的标准输入/输出设备担当此任务。

例如，从标准输入设备循环读入两个整数（直到读空），输出其和。程序代码如下：

```
//=====================================
//f0509.cpp
//重定向输入输出
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(){
  for(int a,b; cin>>a>>b; cout<<a+b<<"\n");
}//=====================================
```

```
E:\ch05>f0509↙
8 9↙
17
```

现在改变程序在集成环境中运行的方式，从“命令提示符”（开始|所有程序|附件|命令提示符）窗口输入命令“f0509 <abc.txt”，即在运行命令上加上“<abc.txt”，便改从该文件中获得数据了：

```
E:\ch05>f0509 <abc.txt↙
17
21
357
```

```
8 9
9 12
12 345
```

abc.txt

显然，这个 abc.txt 文件事先已经在路径 E:\ch05 中存在了。

可重定向程序的一个好处是可以在运行的当场临时决定输入和输出的设备。

如果不专门指定设备，就是标准输入/输出，否则，“<”后面规定输入设备，“>”后面规定输出设备。例如，通过发出命令“f0509 <abc.txt >xyz.txt”，便可以从 abc.txt 文件中读入数据，而将输出送到 xyz.txt 中去：

```
8 9
9 12
12 345
```

abc.txt

```
17
21
357
```

xyz.txt

```
E:\ch05>f0509 <abc.txt >xyz.txt↙
```

运行后，屏幕中没有输出结果，而打开 xyz.txt 文件，则看到运行结果全在里面。还可以通过将“>”改为“>>”让输出结果追加到文件中去。读者可以自行验证。

❑ 5.5.2 使用 main 参数（Using Main Arguments）

重定向是基于标准输入/输出的，也就是说，命令若不重定向，则默认的是标准输入/输出。

在编程的时候，往往不是这种单纯的情况，要处理的可能是若干个非标准设备的资源文件，也可能是遵循某种语言的命令，这个命令加在程序名后面构成命令行，而且该命令行可能由其他程序生成。这时候，重定向就不行了，用简单的标准输入会话形式输入命令也不行了，需要依赖 main 函数的参数。

main 函数的参数结构为两项参数：

```
int main(int argc, char** argv){ ... }
```

最早的时候，还有第三项参数形式，那是为了设置运行环境的，现在因为操作系统的不断进化，甚至在 Java 中也很少用到这项参数。故我们看到的是两个参数的 main 形式。

在 main 参数表达中，要么不声明参数，像以前一样；要么按这里的参数格式声明。只有在定义 main 函数时写出了参数项，在运行时才会接受操作系统的参数传递。

main 函数的参数由操作系统传递，所以比较特殊。两个形参名一般是采用习惯名称 argc 和 argv，表示 argument count 和 argument vector，即第一项是表示传递的 C-串有几个，第二项是表示具体的 C-串数组，该数组最后一项是空串，即指向 0 的串。正像在函数中传递数组那样，既要传递数组地址，也要传递数组的元素个数。要注意的是，C-串的类型为 char*，数组是以指向 C-串的指针为元素的，因而数组描述为 char**。其参数结构的示意图见图 5-8。

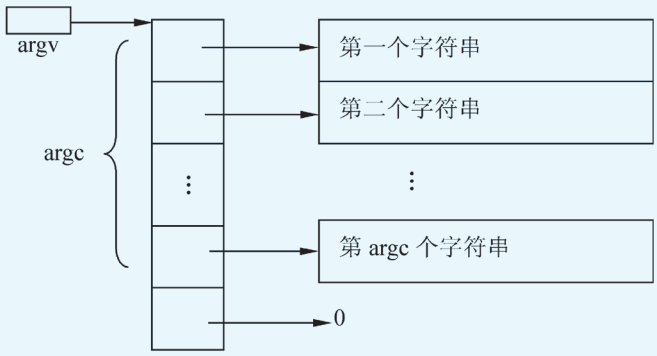

图 5-8 main 参数结构

对于以下程序，若发出命令行“f0510 abc1 abc2 abc3”，则可以根据 main 的形参读取命令行的相关信息：

```
//=====================================
//f0510.cpp
//读入main参数
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
int main(int argc, char** argv){
  for(int i=0; i<argc; ++i)
    cout<<argv[i]<<endl;
}//=====================================
```

```
E:\ch05>f0510 abc1 abc2 abc3↙
f0510
abc1
abc2
abc3
```

因此，如果命令行中为输入/输出文件名的信息，则程序可以直接读取和创建文件流对象，进行文件操作。

例如，写一个程序，简单地响应输入，将其全部输出。也即如果输入文件是 abc.txt，则运行后，输出文件 xyz.txt 的内容与 abc.txt 相同，则程序代码如下：

```
//=====================================
//f0511.cpp
//打开main参数为名字的文件
//=====================================
#include<iostream>
#include<fstream>
using namespace std;
//-------------------------------------
int main(int argc, char** argv){
  if(argc!=3)
    cout<<"Usage: f0511 infile outfile\n";
  else{
    ifstream in(argv[1]);
    ofstream out(argv[2]);
    if(in && out)
      out<<in.rdbuf();
  }
}//=====================================
```

```
asfdf
asfasdfsdfgfgfdgdg
gfdgdfg
fgfdgfgfg
```

abc.txt

```
asfdf
asfasdfsdfgfgfdgdg
gfdgdfg
fgfdgfgfg
```

xyz.txt

```
E:\ch05>f0511↙
Usage: f0511 infile outfile
```

再次运行：

```
E:\ch05>f0511 abc.txt xyz.txt↙
```

然后打开看一下文件 xyz.txt 的内容，正如上面所示。

要解释的是，输出语句“out<<in.rdbuf()”中的 in.rdbuf()直接关联到输入流缓冲区，该缓冲区承载一切输入数据和操作，因此，相当于将整个输入和盘托出给输出流。这是一种简捷设计（☞参考文献[9]之条款 1）。

如果命令行中要求计算的是某种待解释的命令，则也可以通过读取参数后再解析的办法加以处理：

```
//=====================================
//f0512.cpp
//命令解析
//=====================================
#include<iostream>
#include<sstream>
using namespace std;
//-------------------------------------
int main(int argc, char** argv){
  if(argc!=2){
    cout<<"usage: f0512 command\n";
    return 1;
  }
  string s(argv[1]);
  istringstream sin(s);              //用 string 串流来解析命令
  int a,b;  char c;
  sin>>a>>c>>b;
  switch(c){
    case'-': b=-b;
    case'+': cout<<a+b; break;
    case'*': cout<<a*b; break;
    case'/': cout<<a/b; break;
    default: cout<<"error";
  }
}//=====================================
```

```
E:\ch05>f0512 23+5↙
28
```

该程序虽然对命令做了解析，但还远谈不上完善。感兴趣的读者可以修改该程序，使其具有处理各种错误情况的能力。例如出现以下字句：

```
E:\ch05>f0512 23+
E:\ch05>f0512 23+*
E:\ch05>f0512 23+0
E:\ch05>f0512 *23+
```

等错误，如何辨认？

要做这种命令式处理，首先要定义语法规则及其操作，即一种语言。可以将其做成一个计算器（☞参考文献[1]，CH10.2）。

5.6 递归函数（Recursive Functions）

❑ 5.6.1 递归本质（Essence of Recursions）

递归函数即在函数体中出现调用自身的函数。例如，阶乘 n!的数学函数描述为：

$$f(n)=\begin{cases}1 & n=1\\ n\,f(n-1) & n>1\end{cases}$$

其对应的 C++函数描述为：

```
unsigned f(unsigned n){
  if(n==1) return 1;
  return n*f(n-1);
}
```

不过要注意的是，$f(13)>2^{32}$。所以，对于 unsigned int 型（无符号类型）的表示范围，其 n 的取值范围只有 $1\leqslant n\leqslant 12$ 了。

又如，Fibonacci 数列的数学函数描述为：

$$f(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ f(n-1)+f(n-2) & n>2\end{cases}$$

其等价的 C++函数为：

```
unsigned int f(unsigned int n){
  if(n==0 || n==1) return n;
  return f(n-1)+f(n-2);
}
```

这里要注意 n 的表示范围与数学函数的差异。由于 C++函数 $f(47)>2^{32}$，超过了无符号整型数的表示范围，所以该 C++函数参数的范围取值为 $1\leqslant n\leqslant 46$。

递归有直接递归和间接递归之分。所谓间接递归，是指函数体中没有直接调用自身函数，而是调用了另一个函数，在那个函数里出现了调用本函数的语句；或者，在那个函数里又调用了一个其他函数，反复出现调用其他函数，而最后有一个函数调用了本函数。例如：

```
int fn1(int a){
  int b;
  b = fn2(a+1);   //调用 fn2
}
int fn2(int s){
  int c;
  c = fn1(s-1);   //调用 fn1
}
```

由于函数的独立性，即一个函数从来不从属于另一个函数，所以任何函数内不能嵌套定义其他函数，也即不存在子函数的概念。

函数调用时，被调函数中保护了调用函数的状态，包括硬件运行状态、返回地址和数据环境，使得调用函数的状态可以在被调函数返回时全面恢复。恢复与被调函数的状态是无关的。函数的调用在形式上只与函数的参数和返回类型发生关系。

递归函数在运行时，被调函数的数据环境与调用函数数据环境在结构上是一致的。只是由于被调函数与调用函数传递的参数值不同，才使数据环境显出差异。数据环境的操作统一由栈机制实施。栈操作总是处理位于栈顶的函数数据区，函数调用的嵌套深度体现在“货栈”的高度，而栈中不同层次的数据彼此之间在理论上被看作是无关的。

递归函数在运行中，其调用与被调函数的指令代码是同一个函数副本，只不过各个不同运行中的调用点，作为状态的一部分，在栈中被分别保护了起来。因此，是 C++的函数机制决定了递归操作的可能性与形式。

例如，上述 n！的函数，当调用 f(5)时，其运行栈描述如图 5-9 所示。

f (1)	n = 1
	返回值 1
f (2)	n = 2
	返回值 2×f (1)
f (3)	n = 3
	返回值 3×f (2)
f (4)	n = 4
	返回值 4×f (3)
f (5)	n = 5
	返回值 5×f (4)

图 5-9　递归运行栈结构描述

❑ 5.6.2　递归条件（Condition of Recursions）

递归函数设计往往是从数学上的递推关系导出，而递推关系都是有数值规定的，这就给出了递归函数的条件执行性。

（1）递归不能无限制地调用下去，因为栈空间是有限的。所以递归函数是有条件地调用自身，该条件就是递归的停止条件。例如，阶乘函数中的“if(n==1) return 1;”当 n 为 1

时，函数就不再递归了。

（2）递归函数中必须有完成终极任务的语句序列，以使函数有意义。例如，“return 1;”就是阶乘函数完成终极计算的语句，而递归调用并非终极。

（3）递归函数当然有递归调用语句。然而，递归调用应有参数，而且参数值应该是逐渐逼近停止条件的。例如，阶乘函数中的递归调用 f(n–1)对于 f(n)来说，是逐渐逼近了停止条件。限于计算机的硬件资源和性能因素，递归调用的嵌套深度实在有限，所以逼近的速度应该比较现实。例如，若为了做一个 1～n 的求和计算，用递归就很傻，因为逼近的速度实在不敢恭维：

```
int sigma(int n){
  if(n==1) return 1;
  return sigma(n-1)+n;
}  //-------------------------
```

（4）递归条件应先测试，后递归调用。无条件递归的逻辑错误编译器是检查不出来的，要靠程序员自己把握。例如：

```
void count(int val){
  count(val-1);      //无尽递归
  if(val>1)          //无法到达此处
    cout<<"ok:"<<val<<endl;
}  //-------------------------
```

该函数虽然逐渐减小递归调用时的参数值，但由于无条件递归，会最终导致栈空间崩溃。

❑ 5.6.3 消去递归（Removing Recursions）

递归函数都能用非递归函数替代。理论上已经有了递归函数到非递归函数的形式转换方法，也就是可以让计算机自动将递归函数转化成非递归函数。当然，一些小递归函数用手工转换效率会更高。例如，下面有 3 种不同代码，求两个整数的最大公约数：

```
long gcd1(int a, int b){
  if(a%b==0)
    return b;
  return gcd(b, a%b);
}//-------------------------
long gcd2(int a, int b){
  for(int temp; b; a=b, b=temp)
    temp = a%b;
  return a;
}//-------------------------
int gcd3(int a, int b){
  stack<int> x;        //需要包含 stack 头文件
  x.push(a); x.push(b);
```

```
    while(1){
      int y=x.front();    //y<--b
      x.pop();
      int z=x.front();    //z<--a
      x.pop();
      if(z%y==0) return y;
      x.push(y); x.push(z%y);
    }
}//--------------------------
```

gcd1 为递归函数，gcd2 和 gcd3 为非递归函数，gcd2 为简捷版，gcd3 为用栈消去递归之一般方法，自动非递归转换就依赖于这种形式化的方法。

❑ 5.6.4 递归评说（Comment on Recursions）

递归的目的是简化程序设计，使程序易读。对于一个能够写出递归通项的数学函数，我们总是很容易将其程序化，然后考虑一些边界条件，就可以设计成递归函数。例如，前述的 Fibonacci 数列。但递归也会迅速递增系统开销，在时间上，执行函数的调用与返回的次数明显要大于非递归函数；在空间上，栈空间资源也会遭到空前的劫掠，随着每递归一次，栈内存就会多占用一截。递归函数在时空开销上的不利局面势必影响性能（☞CH6.3）。

非递归函数虽然效率高，但有时候却比较难编程，而且相对来说可读性差。

现代编程在层次上做了很多分工，低级编程强调性能（☞CH6.7），抽象编程（相对较高层次）强调可读（☞CH11.1）。就好像车间工人以产品质量为考核依据，而公司经理以经营战略论英雄成败。这些层次，随着计算机硬件性能的不断提高，随着软件方法的不断改善，还将进一步深化。所以，作为对特殊问题的一种处理方法，递归仍然占有编程的一席之地。许多高难算法的简捷描述，往往采用递归，况且递归在迅速退出栈结构的技术处理上还能受到异常机制的意外帮助（☞CH15.7.2）。

5.7 函数重载（Function Overloading）

❑ 5.7.1 重载概念（Concept of Function Overload）

最基本的编程就是给变量和函数命名，在函数中编写一组操作，再以一定方式组织函数。请设想一下在一个程序中，调用函数的数量像语句那么多的时候（面向对象程序中调用的函数真有那么多），其函数命名的难度就开始显现。因为函数多为全局的，彼此都能看到，所以不能重名。对于 C++这样的产业化编程工具来说，命名问题确实是个实际问题，然而，C++的名空间机制（☞CH7.6）也确实妥善地处理了这个问题。

另有一类问题，是对于一组概念相同、处理对象不同的函数（在 C 中只能归为不同的函数）。例如，求绝对值函数：

```
int abs(int);           //求整数的绝对值
double fabs(double);    //求浮点数的绝对值
```

希望通过函数重名达到简化编程的目的，这也是C++致力于简化编程之处。

生活中，人们习惯于区分不同场景下的同名操作。例如，指令“open the door”，在冰箱面前指的是开冰箱门，在轿车面前则指开车门。上面两个函数声明，同样是求绝对值，却要说明成“求整数绝对值（abs）”“求双精度数绝对值（fabs）”，对于这两个函数的参数的不同却置若罔闻，这实在是不必要的。事实上，从声明中提供的信息来说，参数类型已经足够描述所要求的操作性质。C++完全可以同时定义下列函数而不会引起函数意义上的冲突：

```
int abs(int);
double abs(double);
```

剩下的就是技术上的处理了。C++编译器事实上也能够根据函数参数的类型、数量和排列顺序的差异，区分同名函数，其技术称为重载技术，相应的同名函数称为重载函数。重载技术化解了一部分名称命名问题，然而更重要的是在编程逻辑上，亲近了人类的自然思维，从而方便了表达。

例如，针对上述函数声明，可以如下表述：

```
//=====================================
//f0513.cpp
//函数重载
//=====================================
#include<iostream>
//#include<cmath>
//-------------------------------------
int abs(int a){
  return (a>0)? a : -a;
}//-------------------------------------
double abs(double a){
  return (a>0)? a : -a;
}//-------------------------------------
int main(){
  std::cout<<abs(-10)<<endl;
  std::cout<<abs(-12.23)<<endl;
}//=====================================
```

```
E:\ch05>f0513↙
10
12.23
```

而在C中，调用“abs(–12.23)”只能得到近似值12，因为择名而调，转换–12.23为int型时，精度丢失。要想得到精确值12.23，则必须调用“fabs(–12.23)”。

C 头文件都是以.h 结尾的，如 stdio.h、math.h。C 关于 abs 函数的头文件为 math.h，C++对其头文件进行了改良，设置了重载函数，命名为 cmath。所以需要使用重载库函数的时候，一定要用 C++改良版头文件。

C 语言头文件经过 C++改良之后，头文件名字作了有规律的修改，即其头文件名前加 c，再去掉后缀.h。于是 math.h 就成了 cmath 了。常用 C 头文件与改良后的 C++头文件对比如下：

```
assert.h    cassert
math.h      cmath
stdio.h     cstdio
stdlib.h    cstdlib
string.h    cstring
time.h      ctime
types.h     ctypes
conio.h     cconio
```

C++程序既可以使用 C 头文件（C++兼容 C 的特征），又可以使用改良版 C++头文件。

因为 C++改良版头文件的函数功能全覆盖了 C 头文件，所以在 C++编程中，一般就毫无悬念地使用 C++改良版头文件。

早期的 C++（标准没有形成之前），其头文件也是带后缀.h 的，但标准化后，C++头文件去掉了后缀。例如，早期 C++的 IO 流头文件为 iostream.h，标准化后，头文件变成 iostream。出于 C++向下兼容，头文件 iostream.h 在标准 C++中仍然可用，但是所使用的流操作或者烦冗（例如，标准流操作“cout<<setw(5)<<right<<"abc";”在早期头文件下只能写成“cout<<setw(5)<<setioflags(ios::right)<<"abc";”），或者低效（非标准头文件中的操作常常被再次优化）。所以，既然是标准 C++编程，就毫无悬念地使用无后缀的 C++标准头文件。

重载的目的是通过自然描述函数名称简化编程和增强程序的可读性。它让程序员方便地将特定的操作序列与一个自然的名称相对应。因此，可以很容易地分辨什么是滥用重载的不良编程，例如：

```
int abs(int a);                 //返回 a 的绝对值
double abs(double a);           //返回 a 的平方根，与 abs 的意义背离

void func(int a, double d){
  int b = abs(a);               //理解!
  double c = 1.0+abs(d);        //将 abs 的功能误解为求绝对值
}
```

同名函数应该具有相同功能，定义一个 abs 函数而返回的却是一个数的平方根，则该程序的可读性遭到了破坏。

5.7.2 重载函数匹配（Overloaded Function Call Matches）

只要参数个数不同，参数类型不同，参数顺序不同，函数就可以重载。然而，返回类型不同则不允许函数重载。例如：

```
void func(int a);              //ok
void func(char a);             //ok
void func(char a, int b);      //ok
void func(int a, char b);      //ok
char func(int a);              //与第一个函数冲突
```

因为调用函数是只看参数匹配的，有的函数虽有返回类型，但不参与表达式运算而单独作为一条语句，因此函数的匹配不能根据上下文判断。例如：

```
func(23);  //错: void func(int) 还是 char func(int) ?
```

C++按下列三个步骤的先后顺序找到匹配并调用函数：

（1）寻找一个严格匹配，如果找到了，就用那个函数。

（2）通过相容类型的隐式转换寻求一个匹配，如果找到了，就用那个函数。

（3）通过用户定义的转换（☞CH9.7.1）寻求一个匹配，若能查出有唯一的一组转换，就用那个函数。

例如，重载函数 print 的匹配：

```
void print(double);
void print(int);

void func(){
  print(1);           //匹配 void print(int);          规则 1
  print(1.0);         //匹配 void print(double);       规则 1
  print('a');         //匹配 void print(int);          规则 2
  print(3.1415f);     //匹配 void print(double);       规则 2
}
```

C++允许 int 型到 long 型，int 型到 double 型的隐式转换。但若必须在两者之间抉择时，则会引起错误。例如，当实参是整数，而重载函数一为 long 型参数，一为 double 型参数时，下面的函数调用将引起错误：

```
void print(long a);
void print(double a);

void func(int a){
  Print(a);          //错: long 型与 double 型匹配哪一个呢?
}
```

为避免二义性，在调用时，应显式表明是 print(long(a))还是 print(double(a))。

❑ 5.7.3 重载技术（Function Overload Technology）

C++用名称压轧（name mangling）技术改变函数名，区分参数不同的同名函数。名称压轧是十分简单的过程，一系列代码被附加到函数名上以标记参数类型以及它们出现的次序。例如，用 v、c、i、f、l、d、r 分别表示 void、char、int、float、long、double 以及其引用，则重载函数：

```
int func(char a);
int func(char a, int b, double c);
```

在编译器内部分别被表示为 func_c 和 func_cid，但是，若恰好有一个函数也叫“int func_cid();”呢？没有关系！根据名称压轧规则，其编译器内部表示为“func_cid_v”，还是做到了与上述函数名相区别。

名称压轧对编程来说是看不到的，它在编译过程中悄悄进行，在链接过程中默默使用。名称压轧不是 C++标准，所以各个编译器的名称压轧方案可能不同。例如，函数“int func(char a,int b,double c)”可能被压轧成 func_c_i_d，或者 func$c$i$d，但这不影响程序的移植，因为编译和链接总是捆绑的。只是在不同语言之间穿梭时，如 C 函数在 C++中使用，或者其他语言的模块移植到 C++中时，因为它们都是直接的机器代码、直接的函数名称，而且编译已经完成，要参加 C++程序的链接，会带来衔接上的问题，但通过函数名前加上 extern C 修饰，阻止函数名压轧，便可解决这个问题。

❑ 5.7.4 默认参数（Default Parameters）

C++可以给函数声明中的参数使用默认参数值。这样在函数调用时，对应的实参就可以省略。

函数参数的作用是传递输入数据，有时候，函数可以通过默认参数的形式使用。假想某一确定的输入数据，省略其调用时的参数传递，完成函数调用。例如：

```
//=====================================
//f0514.cpp
//默认参数
//=====================================
#include<iostream>
using namespace std;
//-------------------------------------
void delay(int a = 2);          //函数声明时
//-------------------------------------
int main(){
  cout<<"delay 2 sec.....";
  delay();                      //等于调用 delay(2)
  cout<<"ended.\n";
  cout<<"delay 5 sec.....";
  delay(5);
```

```
  cout<<"ended.\n";
}//-----------------------------------
void delay(int a){                  //函数定义时
  int sum=0;
  for(int i=1; i<=a; ++i)
  for(int j=1; j<3500; ++j)
  for(int k=1; k<100000; ++k) sum++;
}//===================================
```

delay（延时）函数有一个参数，表示延迟的时间，如果没有函数声明中对参数的默认值描述，则调用的形式必须带一个参数以确定延迟时间。但真要这样就缺少了一些灵活性，因为对于 delay 函数，可能大部分情况都需要延迟 2 秒，在特殊情况时，才要延迟更多的秒数。于是，大多数情况下，delay 函数调用是固定的 2 秒形式，就像生活中经常碰到的“老时间、老地点、老方法”之类，而无须重新精确描述大家都清楚的具体时间、地点和方法。而且如果程序进行必要的维护时，如参数值的单位改成毫秒，则参数值将从 2 变成 2000，这时带有默认参数值的函数调用就可以免于调整，而其他没有默认参数的函数都需要进行修改。所以，从方便编程的角度看，默认参数形式带来了诸多便利。

况且在编程中，默认参数还可以避免调用中实参描述上的不统一。例如：

```
delay(2);                           //文字量实参
int a=2;    delay(a);               //变量实参
const int b=2;     delay(b);        //常量实参
```

这些不统一会给严格类型对等匹配的模板带来一些麻烦（☞CH14.2.1）。

❑ 5.7.5 默认参数规则（Default Parameter Rules）

一般来说，默认参数值总是在函数声明时描述。因为实用的程序中，函数声明总是与函数定义分离的，而在又有声明又有定义时，默认参数值只能置身于声明中。例如：

```
void point(int =3, int =4);         //默认参数值，同时注意可以形参省略

void point(int x, int y){           //定义中不允许再给出默认值
  cout<<x<<endl;
  cout <<y<<endl;
}
```

函数参数默认值只能从后往前设置，例如：

```
void func(int a=1, int b, int c=3, int d=4);    //错：b 和 a 的位置违规
void func(int a, int b=2, int c=3, int d=4);    //ok
```

调用时的实参按位置解析，默认实参也只能从后往前逐个替换尾部的“缺漏”。例如：

```
func(10,15,20,30);          //ok
func();                     //错：a 没有默认值，所以必须给出实参
```

```
func(12,12);              //ok: c 和 d 默认
func(2,15,,20);           //错: d 不默认则 c 也无资格默认
```

默认参数值的设定不能出尔反尔，也就是说，不能一会儿声明这样，一会儿声明那样。这首先是破坏了编程的风格，其次是为编译所不容。例如：

```
void func(int a=1);
void func(int a=2);   //错
```

❑ 5.7.6 无名参数（Nameless Parameters）

函数的声明和定义中，都可以省略形参名，例如：

```
void point(int a, int);      //第二个形参名省略
```

如果在函数定义中也省略形参名，则函数体中就不能使用这个形参了。例如：

```
void point(int a, int){      //第二个参数形同虚设
  cout<<a<<endl;
}
void func(){
  point(12,15);              //因为声明规定要有两个参数，所以调用中要有第二个实参
}
```

一般来说，函数设计有这样的问题，一旦确定了函数的声明形式，就不能随便更改。因为函数声明给了调用者一个确定的信息，也给了函数定义者一个确定的信息，等于在调用和定义之间达成了一个约定。而调用者和定义者往往是不照面的，甚至可以在天南地北并处于不同的开发周期。因此，若开始用了一个函数参数，后来发现不需要用它，就可以将它无名化，而且不需要改动那些调用该函数以前版本的代码。

特别地，在 C++的异常机制中，捕获异常是通过类型匹配的方式，知道了类型也就知道了处理的方式，所以常常利用参数省略名称实现其免于传递的简捷性（☞CH15.2.4）。

❑ 5.7.7 重载或参数默认（Overload or Parameter Default）

设计为重载函数，还是函数带参数默认，有个策略问题。

（1）如果两个重载函数做基本相同的事，只不过参数个数不同而已，则还不如用默认参数值的方法做。例如，延迟函数的两个版本：

```
void delay();        //延迟 2 秒
void delay(int a);   //延迟 a 秒
```

最笨的方法是仿照 f0513.cpp 的办法，将这两个函数重载定义。

第二种方法是在 void delay(int a)中定义时间延迟操作，然后重载 void delay()，直接调用前一个函数：

```
void delay(){
  delay(2);
}
```

更好的方法就是 f0514.cpp 默认参数值的方法，因为两个过程内容几乎没有差别。

（2）如果一个参数的值用来确定不同的操作，则用重载函数较好。例如：

```
//=====================================
//f0515.cpp
//别扭的参数默认
//=====================================
#include<iostream>
#include<vector>
using namespace std;
//-------------------------------------
vector<int> b(10, 0);
void print(const vector<int>& a = b);
bool process(vector<int>& a);
//-------------------------------------
int main(){
  vector<int> a(10, 5);
  //...
  if(process(a))  print(a);
  else            print();
}//-------------------------------------
void print(const vector<int>& a){
  if(a==vector<int>(10, 0)){
    cout<<"failed.\n";
    return;
  }
  for(int i=0; i<a.size(); ++i)
    cout<<a[i]<<" ";
  cout<<"\n";
}//-------------------------------------
bool process(vector<int>& a){
  int sum = 0;
  for(int i=0; i<a.size(); ++i) sum += a[i];
  if(sum>100) return true;
  else return false;
}//=====================================
```

该程序做的工作是通过调用 process 函数，判断是调用 print(a)还是调用 print()。两种调用虽然都是输出工作，但内容截然不同。所以还不如用两个函数描述两种操作为妥。

程序 f0515.cpp 中的 print 函数用了默认参数值，该参数值决定了函数中的不同操作，是输出 failed 还是输出其整个向量，这比起将两部分代码分开成独立的重载函数效率要低一些。更重要的是，从程序的逻辑性和可维护性来说，重载函数也更优些。重载函数版本见下面的 f0516.cpp 程序代码：

```
//=====================================
//f0516.cpp
//重载函数版本
//=====================================
#include<iostream>
#include<vector>
using namespace std;
//-------------------------------------
void print(const vector<int>& a);
void print();
bool process(vector<int>& a);
//-------------------------------------
int main(){
  vector<int> a(10, 5);
  //...
  if(process(a)) print(a);
  else           print();
}//-------------------------------------
void print(){
  cout<<"failed.\n";
}//-------------------------------------
void print(const vector<int>& a){
  for(int i=0; i<a.size(); ++i)
    cout<<a[i]<<" ";
  cout<<"\n";
}//-------------------------------------
bool process(vector<int>& a){
  int sum = 0;
  for(int i=0; i<a.size(); ++i) sum += a[i];
  if(sum>100) return true;
  else return false;
}//=====================================
```

5.8 目的归纳（Conclusion）

函数是程序的基本组成。它不仅是数学意义上定义域到值域的映射，而且是灵活多样的过程。本质上基于过程的程序设计就是以函数作为基本过程单位的。

然而，仅以函数组成程序是远远不够的，没有数据结构架构，在大型程序设计中，面对数量巨大的函数，人们只能束手无策。

函数的灵活多样性反映在其副作用上，副作用又是反映在参数的指针和引用传递上。

人们利用指针和引用的参数传递，大大地提高了数据传递的效率。但是，人们又在百般提防使用函数的指针参数，因为任何有意无意的越权数据和代码访问都是指针造成的。数据和代码的越权访问是程序缺陷所在。

由于指针和引用，函数的输出数据可以在参数中表示，甚至可以将全局数据用于函数的输入/输出（☞CH7.3.2）。

通过规范的函数设计，可以避免许多函数的缺陷。函数的黑盒性是程序设计规范化的基点，如何使用函数黑盒概念设计模块和过程结构，如何利用函数的副作用提高程序的性能，又如何认识到函数副作用的隐蔽性而丰富程序调试的经验，是高级程序员与初级程序员的分水岭。

C++中函数机制是靠栈结构支撑的，不同的编译器处理函数的手法大同小异，栈机制使得函数的执行代码与函数的数据区域相分离，使得函数递归调用成为可能。然而，递归调用全套复制了函数的局部数据，因而对于深度的递归可能在时间和空间上陷入困境。

函数指针的使用有点别扭，使用它需要经验，初学者少有涉及。然而函数指针开拓了程序运行所覆盖的计算机系统空间，它可以让程序优美灵活，也可以让程序充满邪恶。在设计上它是一柄双刃剑。

任何一种程序设计方法，最重要的问题都是程序组织的问题。基于过程的程序设计，也必然涉及程序的组织。推出高级语言，在于非专业程序员要求介入的呼声，在于软件大生产，在于程序员的分工。所以，除了用函数去堆积一个程序外，还需要程序员互相之间的协调，当不在同一台计算机上的多个程序员为同一个工程而忙碌的时候，名称冲突，共享名称使用便成了需要解决的问题。对于包含多个函数的文件的组织问题也需要一并解决。在第 7 章将介绍函数堆积所构成的结构，即程序的组织问题。

main 函数是操作系统调用的函数，因而它是唯一不能由其他函数调用的函数，它的返回类型一定是 int 型，否则就不是标准的。但是 main 函数中的返回语句可以网开一面地免去，这也是它的特权。main 函数所带的参数规定了操作系统对它数据传递的形式，借此参数，程序可以处理运行命令中的附加信息。重定向是操作系统启动命令的功能之一。

函数可以重载，即重名，但必须有不同的参数类型、个数和顺序。重载函数带来程序设计概念上的亲和性，使得函数名不必处处不同。函数参数可以默认，其效果类似于函数重载，但本质上是两回事。函数重载和默认参数在使用中存在一些差别。

练习 5（Exercises 5）

1. 已知函数 poly 是用递归方法计算 x 的 n 阶勒让德多项式的值。数学函数如下：

$$\begin{cases} 1 & n=0 \\ poly_n(x)=x & n=1 \\ ((2n-1)*x*poly_{n-1}(x)-(n-1)*poly_{n-2}(x))/n & n>1 \end{cases}$$

请写出 C++函数，其参数为 x 和 n。并求当 x 为 1.2，n 为 8 时，其 poly 函数的值。

2. 使用函数声明、调用和定义的三部曲，将下列程序用三个函数拆开，并求出运行结果。

```
//====================================
//e0502.cpp
//calcu students grade
//====================================
#include<iostream>
using namespace std;
//----------------------------------------
const int n = 5;
const int n = 4;
int a[n][m];
//----------------------------------------
int main(){
  //input
  for(int i=0; i<n; i++)
  for(int j=0; j<m; j++)
    cin >>a[i][j];
  //total
for(int i=0; i<n; j++){
    int sum=0
    for(int j=0; j<m; j++)
      sun += a[i][j];
    cout<<(I+1)<<":"<<sum<<"\n"
}
//average
for (int i=0; i<m; i++){
  int sum=0
  for (int j=0; j<n; j++)
    sum +=a[j][i];
  cout<<"NO"<<i<<"average is "<<double(sum)/n<<"\n";
}
}//====================================
```

3. 有一个文件 abc.in，其中含有一些整数对，求出这些整数对的最大公约数，并对这些最大公约数按从小到大的顺序排序输出。其样板文件内容和结果如下：

```
E:\ch05e>e0503↙
1 2 3 4 7 8 16
```

```
12 35
77 91
123 789
24 28
64 112
1024 888
98 54
```

abc.in

4．用递归方法求解 2.9 节第 10 题。

5．编程实现：

（1）从文件 abc.txt 中读入全部数据到一个整数向量中。

（2）文件 abc.txt 的内容为 12 567 91 33 657 812 2221 3 77。

（3）以该向量为参数，将该整数向量中的数据按各位数字之平方和的大小排序。

（4）设计一个函数，以该向量为参数，以每个元素空一格的格式在屏幕中输出。

（5）编程实现读入向量，若无元素，则仅输出元素个数；否则，排序并输出该向量。

6．在文件 abc.txt 中，存有一些各不相同的 C-串，每串占一行。请使用适当的数据类型，读入这些 C-串并使用 STL 的 sort 算法进行字典序排序，然后输出。如果需要，也可以自定义比较函数，作为函数指针参数。文件样本如下：

```
asfdf rtr
Asfasdf sdfgfgfdgdg
gfdgdfg
fgf dgf    gfg
efdf b34 4345 tr6
efdsf th 776
```

abc.txt