第2章　数 据 抽 象

C++是一个能提高效率的工具。为什么我们还要努力（这是努力，不管我们试图做的转变多么容易）使我们从已经熟悉且效率高的语言（在这里是 C语言）转到另一种新的语言上？而且使用这种新语言，我们会在确实掌握它之前的一段时间内降低效率。这归因于我们确信通过使用新工具将会得到更大的好处。

用程序设计术语，多产意味着用较少的人在较少的时间内完成更复杂和更重要的程序。然而，选择语言时确实还有其他问题，例如运行效率（该语言的性质引起代码臃肿吗？）、安全性（该语言能有助于我们的程序做我们计划的事情并具有很强的纠错能力吗？）、可维护性（该语言能帮助我们创建易理解、易修改和易扩展的代码吗？）。这些都是本书要考察的重要因素。

简单地讲，提高生产效率，意味着本应花费三个人一星期的程序，现在只需要花费一个人一两天的时间。这会涉及到经济学的多层次问题。生产效率提高了，我们很高兴，因为我们正在建造的东西功能将会更强；我们的客户（或老板）很高兴，因为产品生产又快，用人又少；我们的顾客很高兴，因为他们得到的产品更便宜。而极大提高效率的唯一办法是使用其他人的代码，即使用库。

库，简单地说就是一些人已经写的代码，按某种方式包装在一起。通常，最小的包是带有扩展名如LIB的文件和向编译器声明库中有什么的一个或多个头文件。连接器知道如何在 LIB文件中搜索和提取相应的已编译的代码。但是，这只是提供库的一种方法。在跨越多种体系结构的平台上，例如UNIX，通常，提供库的最明智的方法是用源代码，这样在新的目标机上它能被重新编译。而在微软Windows上，动态连接库是最明智的方法，这使得我们能够利用新发布的DDL经常修改我们的程序，我们的库函数销售商可能已经将新 DDL发送给我们了。

所以，库大概是改进效率的最重要的方法。 C++的主要设计目标之一是使库容易使用。这意味着，在C中使用库有困难。懂得这一点就对 C++设计有了初步的了解，从而对如何使用它有了更深入的认识。

2.1　声明与定义

首先，必须知道“ 声明 ”和“ 定义 ”之间的区别，因为这两个术语在全书中会被确切地使用。“ 声明 ”向计算机介绍名字，它说，“ 这个名字是什么意思 ”。而“ 定义 ”为这个名字分配存储空间。无论涉及到变量时还是函数时含义都一样。无论在哪种情况下，编译器都在“ 定义 ”处分配存储空间。对于变量，编译器确定这个变量占多少存储单元，并在内存中产生存放它们的空间。对于函数，编译器产生代码，并为之分配存储空间。函数的存储空间中有一个由使用不带参数表或带地址操作符的函数名产生的指针。

定义也可以是声明。如果该编译器还没有看到过名字 A，程序员定义int A，则编译器马上为这个名字分配存储地址。

声明常常使用于extern关键字。如果我们只是声明变量而不是定义它，则要求使用 extern。对于函数声明， extern是可选的，不带函数体的函数名连同参数表或返回值，自动地作为一个声明。

函数原型包括关于参数类型和返回值的全部信息。 int f(float,char);是一个函数原型，因为它不仅介绍f这个函数的名字，而且告诉编译器这个函数有什么样的参数和返回值，使得编译器能对参数和返回值做适当的处理。 C++要求必须写出函数原型，因为它增加了一个重要的安全层。下面是一些声明的例子。

```
/*: DECLARE.C -- Declaration/definition examples */
extern int i; /* Declaration without definition */
extern float f(float); /* Function declaration */

float b;  /* Declaration & definition */
float f(float a) {  /* Definition */
  return a + 1.0;
}

int i; /* Definition */
int h(int x) { /* Declaration & definition */
  return x + 1;
}

main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
}
```

在函数声明时，参数名可给出也可不给出。而在定义时，它们是必需的。这在 C语言中确实如此，但在C++中并不一定。

全书中，我们会注意到，每个文件的第一行是一个注释，它以注释符开始，后面跟冒号。这是我用的技术，可以利用诸如“ grep ”和“ awk ”这样的文本处理工具从代码文件中提取信息。在第一行中还包含有文件名，因此能在文本和其他文件中查阅这个文件，本书的代码磁盘中也很容易定义这个文件。

2.2 一个袖珍C库

一个小型库通常以一组函数开始，但是，已经用过别的 C 库的程序员知道，这里通常有更多的东西，有比行为、动作和函数更多的东西。还有一些特性（颜色、重量、纹理、亮度），它们都由数据表示。在C语言中，当我们处理一组特性时，可以方便地把它们放在一起，形成一个struct。特别是，如果我们想表示我们的问题空间中的多个类似的事情，则可以对每件事情创建这个struct的一个变量。

这样，在大多数C库中都有一组struct和一组活动在这些struct上的函数。现在看一个这样的例子。假设有一个程序设计工具，当创建时它的表现像一个数组，但它的长度能在运行时建立。我称它为stash。

```
/*: LIB.H -- Header file: example C library */
/* Array-like entity created at run-time */

typedef struct STASHtag {
  int size;  /* Size of each space */
  int quantity; /* Number of storage spaces */
  int next; /* Next empty space */
  /* Dynamically allocated array of bytes: */
  unsigned char* storage;
} Stash;

void initialize(Stash* S, int Size);
void cleanup(Stash* S);
int add(Stash* S, void* element);
void* fetch(Stash* S, int index);
int count(Stash* S);
void inflate(Stash* S, int increase);
```

在结构内部需要引用这个结构时可以使用这个 struct的别名，例如，创建一个链表，需要指向下一个struct的指针。在C库中，几乎可以在整个库的每个结构上看到如上所示的 typedef。这样做使得我们能把struct作为一个新类型处理，并且可以定义这个struct的变量，例如：

```
stash A, B, C;
```

注意，这些函数声明用标准 C 风格的函数原型，标准 C 风格比“老”C 风格更安全和更清楚。我们不仅介绍了函数名，而且还告诉编译器参数表和返回值的形式。

storage指针是一个unsigned char*。这是 C 编译器支持的最小的存储片，尽管在某些机器上它可能与最大的一般大，这依赖于具体实现。人们可能认为，因为 stash被设计用于存放任何类型的变量，所以void*在这里应当更合适。然而，我们的目的并不是把它当作某个未知类型的块处理，而是作为连续的字节块。

这个执行文件的源代码（如果我们买了一个商品化的库，我们可能得到的只是编译好的OBJ或LIB或DDL等）如下：

```
/*: LIB.C -- Implementation of
    example C library */
/* Declare structure and functions: */
#include "..\1\lib.h"
/* Error testing macros: */
#include <assert.h>
/* Dynamic memory allocation functions: */
#include <stdlib.h>
#include <string.h> /* memcpy() */
#include <stdio.h>
void initialize(Stash* S, int Size) {
  S->size = Size;
  S->quantity = 0;
  S->storage = 0;
  S->next = 0;
```

```
}

void cleanup(Stash* S) {
  if(S->storage) {
       puts("freeing storage");
       free(S->storage);
  }
}

int add(Stash* S, void* element) {
  /* enough space left? */
  if(S->next >= S->quantity)
    inflate(S, 100);
  /* Copy element into storage,
  starting at next empty space: */
  memcpy(&(S->storage[S->next * S->size]),
         element, S->size);
  S->next++;
  return(S->next - 1); /* Index number */
}

void* fetch(Stash* S, int index) {
  if(index >= S->next || index < 0)
    return 0;  /* Not out of bounds? */
  /* Produce pointer to desired element: */
  return &(S->storage[index * S->size]);
}

int count(Stash* S) {
  /* Number of elements in stash */
  return S->next;
}

void inflate(Stash* S, int increase) {
  void* v =
    realloc(S->storage,
            (S->quantity + increase)
            * S->size);
  /* Was it successful? */
  assert(v);
  S->storage = v;
  S->quantity += increase;
}
```

注意本地的 #include 风格，尽管这个头文件在本地目录下，但仍然以相对于本书的根目录给出。这样做，可以创建不同于这本书根目录的另外的目录，很容易拷贝文件到这个新目录下去实验，而不必担心改变 #include 中的路径。

initialize( )完成对 struct stash 的必要的设置，即设置内部变量为适当的值。最初，设置storage指针为零，设置size 指示器也为零，表示初始存储未被分配。

add( )函数在stash的下一个可用位子上插入一个元素。首先，它检查是否有可用空间，如果没有，它就用后面介绍的 inflate( ) 函数扩展存储空间。

因为编译器并不知道被存放的特定变量的类型（函数返回的都是 void*），所以我们不能只做赋值，虽然这的确是很方便的事情。代之，我们必须用标准 C 库函数memcpy( )一个字节一个字节地拷贝这个变量，第一个参数是 memcpy( ) 开始拷贝字节的目的地址，由下面表达式产生：

```
&(S->storage[S->next * S->size])
```

它指示从存储块开始的第next个可用单元结束。这个数实际上就是已经用过的单元号加一的计数，它必须乘上每个单元拥有的字节数，产生按字节计算的偏移量。这不产生地址，而是产生处于这个地址的字节，为了产生地址，必须使用地址操作符 &。

memcpy( )的第二和第三个参数分别是被拷贝变量的开始地址和要拷贝的字节数。 next计数器加一，并返回被存值的索引。这样，程序员可以在后面调用 fetch( )时用它来取得这个元素。

fetch( )首先看索引是否越界，如果没有越界，返回所希望的变量地址，地址的计算采用与add( )中相同的方法。

对于有经验的C程序员count( )乍看上去可能有点奇怪，它好像是自找麻烦，做手工很容易做的事情。例如，如果我们有一个 struct stash，例假设称为intStash，那么通过用intStash.next找出它已经有多少个元素的方法似乎更直接，而不是去做 count(&intStash)函数调用（它有更多的花费）。但是，如果我们想改变 stash的内部表示和计数计算方法，那么这个函数调用接口就允许必要的灵活性。并且，很多程序员不会为找出库的“ 更好 ”的设计而操心。如果他们能着眼于struct和直接取next的值，那么可能不经允许就改变 next。是不是能有一些方法使得库设计者能更好地控制像这样的问题呢？（是的，这是可预见的）。

动态存储分配

我们不可能预先知道一个stash需要的最大存储量是多少，所以由 storage指向的内存从堆中分配。堆是很大的内存块，用以在运行时分一些小单元。在我们写程序时，如果我们还不知道所需内存的大小，就可以使用堆。这样，我们可以直到运行时才知道需要存放 200个airplane变量，而不仅是 20个。动态内存分配函数是标准 C 库的一部分，包括 malloc( )、 calloc( )、realloc( )和free( )。

inflate( )函数使用realloc( )为stash得到更大的空间块。realloc( )把已经分配而又希望重分配的存储单元首地址作为它的第一个参数（如果这个参数为零，例如 initialize( )刚刚被调用时，realloc( )分配一个新块）。第二个参数是这个块新的长度，如果这个长度比原来的小，这个块将不需要作拷贝，简单地告诉堆管理器剩下的空间是空闲的。如果这个长度比原来的大，在堆中没有足够的相临空间，所以要分配新块，并且要拷贝内存。 assert( )检查以确信这个操作成功。（如果这个堆用光了， malloc( )、calloc( )和realloc( )都返回零。）

注意，C 堆管理器相当重要，它给出内存块，对它们使用 free( )时就回收它们。没有对堆进行合并的工具，如果能合并就可以提供更大的空闲块。如果程序多次分配和释放堆存储，最终会导致这个堆有大量的空闲块，但没有足够大且连续的空间能满足我们对内存分配的需要。但是，如果用堆合并器移动内存块，又会使得指针保存的不是相应的值。一些操作环境，例如Microsoft Windows有内置的合并，但它们要求我们使用专门的内存句柄（它们能临时地翻转为

指针，锁住内存后，堆压紧器不能移动它），而不是使用指针。

assert( )是在ASSERT.H中的预处理宏。assert( )取单个参数，它可以是能求得真或假值的任何表达式。这个宏表示：“我断言这是真的，如果不是，这个程序将打印出错信息，然后退出。”不再调试时，我们可以用一个标志使得这个断言被忽略。在调试期间，这是非常清楚和简便的测试错误的方法。不过，在出错处理时，它有点生硬：“对不起，请进行控制。我们的 C 程序对一个断言失败，并且跳出去。”在第17章中，我们将会看到， C++是如何用出错处理来处理重要错误的。

编译时，如果在栈上创建一个变量，那么这个变量的存储单元由编译器自动开辟和释放。编译器准确地知道需要多少存储容量，根据这个变量的活动范围知道这个变量的生命期。而对动态内存分配，编译器不知道需要多少存储单元，不知道它们的生命期，不能自动清除。因此，程序员应负责用 free( )释放这块存储， free( )告诉堆管理器，这个存储可以被下一次调用的malloc( )、calloc( )或realloc( )重用。合理的方法是使用库中cleanup( )函数，因为在这里，该函数做所有类似的事情。

为了测试这个库，让我们创建两个stash。第一个存放int，第二个存放80个字符的数组（我们可以把它看作新数据类型）。

```
/*: LIBTESTC.C -- Test demonstration library */
#include "..\1\lib.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  Stash intStash, stringStash;
  int i;
  FILE* file;
  char buf[BUFSIZE];
  char* cp;
/* .... */
initialize(&intStash, sizeof(int));
for(i = 0; i < 100; i++)
  add(&intStash, &i);
/* Holds 80-character strings: */
initialize(&stringStash,
           sizeof(char) * BUFSIZE);
file = fopen("LIBTESTC.C", "r");
assert(file);
while(fgets(buf, BUFSIZE, file))
  add(&stringStash, buf);
fclose(file);

for(i = 0; i < count(&intStash); i++)
  printf("fetch(&intStash, %d) = %d\n", i,
         *(int*)fetch(&intStash, i));
```

```
  i = 0;
  while((cp = fetch(&stringStash, i++)) != 0)
    printf("fetch(&stringStash, %d) = %s",
           i - 1, cp);
  putchar('\n');
  cleanup(&intStash);
  cleanup(&stringStash);
}
```

在main( )的开头定义了一些变量，其中包括两个stash结构变量，当然。稍后我们必须在这个程序块的对它们初始化。库的问题之一是我们必须向用户认真地说明初始化和清除函数的重要性，如果这些函数未被调用，就会出现许多问题。遗憾的是，用户不总是记得初始化和清除是必须的。他们只知道他们想完成什么，并不关心我们反复说的："喂，等一等，您必须首先做这件事。"一些用户甚至认为初始化这些元素是自动完成的。的确没有机制能防止这种情况的发生（只有多预示）。

intStash适合于整型，stringStash适合于字符串。这些字符串是通过打开源代码文件LIBTEST.C 和把这些行读到 stringStash 而产生的。注意一些有趣的地方：标准 C 库函数打开和读文件所使用的技术与在stash中使用的技术类似。fopen( )返回一个指向 FILE struct的指针，这个 FILE struct是在堆上创建的，并且能将这个指针传给涉及到这个文件的任何函数。（在这里是fgets( )）。fclose( )所做的事情之一是向堆释放这个FILE struct。一旦我们开始注意到这种模式的，包含着struct和有关函数的 C 库后，我们就能到处看到它。

装载了这两个stash之后，可以打印出它们。intStach的打印用一个for循环，用count( )确定它的限度。stringStash的打印用一个while语句，如果fetch( )返回零则表示打印越界,这时跳出循环。

在我们考虑有关 C 库创建的问题之前，应当了解另外一些事情（我们可能已经知道这些，因为我们是 C 程序员）。第一，虽然这里用头文件，而且实际上用得很好，但它们不是必须的。在C中可能会调用还未声明的函数。好的编译器会告诫我们应当首先声明函数，但不强迫这样做。这是很危险的，因为编译器能假设以 int参数调用的函数有包含 int的参数表，并据此处理它，这是很难发现的错误。

注意，头文件 LIB.H 必须包含在涉及 stash 的所有文件中，因为编译器不可能猜出这个结构是什么样子的。它能猜出函数，即便它可能不应当这样，但这是 C 的一部分。

每个独立的C文件就是一个处理单元。就是说，编译器在每个处理单元上单独运行，而编译器在运行时只知道这个单元。这样，用包含头文件提供信息是相当重要的，因为它为编译器提供了对程序其他部分的理解。在头文件中的声明特别重要，因为无论是在哪里包含这个头文件，编译器都会知道要做什么。例如，若在一个头文件中声明void foo(float)，编译器就会知道，如果我们用整型参数调用它，它会自动把int转变为float。如果没有声明，这个编译器就会简单地猜测，有一个函数存在，而不会做这个转变。

对于每个处理单元，编译器创建一个目标文件，带有扩展名 .o 或 .obj 或类似的名字。必须再用连接器将这些目标文件连同必要的启动代码连接成可执行程序。在连接期间，所有的外部引用都必须确定。例如在 LIBTEST.C 中，声明并使用函数 initialize( ) 和fetch( )，(也就是，编译器被告知它们像什么，)但未定义。它们是在 LIB.C 中定义的,这样，在 LIBTEST.C 中的这些调用都是外部引用。当连接器将目标文件连接在一起时，它找出未确定的引用并寻找这些引

用对应的实际地址，用这些地址替换这些外部引用。

重要的是认识到，在 C 中，引用就是函数名，通常在它们前面加上下划线。所以，连接器所要做的就是让被调用的函数名与在目标文件中的函数体匹配起来。如果我们偶然做了一个调用，编译器解释为 foo(int)，而在其他目标文件中有 foo(float)的函数体，连接器将认为一个_foo在一处而另一个_foo在另一处，它会认为这都是对的。在调用 foo( )处将一个int放进栈中，而foo( )函数体期望在这个栈中的是一个 float。如果这个函数只读这个值而不对它写，尚不会破坏这个栈。但从这个栈中读出的 float值可能会有另外的某种理解。这是最坏的情况，因为很难发现这个错误。

2.3 放在一起：项目创建工具

分别编译时（把代码分成多个处理单元），我们需要一些方法去编译所有的代码，并告诉连接器把它们与相应的库和启动代码放在一起，形成一个可执行文件。大部分编译器允许用一条命令行语句。例如编译器命名为cpp，可写:

```
cpp libtest.c lib.c
```

这个方法带来的问题是，编译器必须首先编译每个处理单元，而不管这个单元是否需要重建。虽然我们只改变了一个文件，但却需要耗费时间来对项目中的每一个文件进行重新编译。

对这个问题的第一种解决办法，已由 UNIX（C的诞生地）提出，是一个被称为 make 的程序。make比较源代码文件的日期和目标文件的日期，如果目标文件的日期比源代码文件的早，make 就调用这个编译器对这个单元进行处理。我们可以从编译器文档 [1] 中学到更多的关于 make 的知识。

make 是有用的，但学习和配置 makefile 有点乏味。makefile 是描述项目中所有文件之间关系的文本文件。因此，编译器销售商发行它们自己的项目创建工具。这些工具向我们询问项目中有哪些处理单元，并确定它们的关系。这些关系有些类似于 makefile文件，通常称为项目文件。程序设计环境维护这个文件，所以我们不必为它担心。项目文件的配置和使用随系统而异，假设我们正在使用我们选择的项目创建工具来创建程序，我们会发现如何使用它们的相应文档（虽然由编译器销售商提供的项目文件工具通常是非常简单的，可以不费劲地学会它们）。

文件名

应当注意的另一个问题是文件命名。在 C中，惯例是以扩展名 .h命名头文件（包含声明），以 .c 命名实现文件（它引起内存分配和代码生成）。C++ 继续演化。它首先是在 Unix 上开发的，这个操作系统能识别文件名的大小写。原来的文件名简单地变为大写，形成 .H 和 .C 版本。这样对于不区分大小写的操作系统，例如 MS-DOS，就行不通了。DOS C++ 厂商对于头文件和实现文件分别使用扩展名.hxx 和 .cxx。后来，有人分析出，需要不同扩展名的唯一原因是使得编译器能确定编译C还是C++文件。因为编译器不直接编译头文件，所以只有实现文件的扩展名需要改变。现在人们已经习惯于在各种系统上对于实现文件都使用 .cpp而对于头文件使用 .h。

2.4 什么是非正常

我们通常有特别的适应能力，即使是对本不应该适应的事情。 stash 库的风格对于 C 程序员已经是常用的了，但是如果观察它一会儿，就会发现它是相当笨拙的。因为在使用它时，必

[1] 参看由作者编写的C++ Inside & Out，(Osborne/McGraw-Hill,1993)。

须向这个库中的每一个函数传递这个结构的地址。而当读这些代码时，这种库机制会和函数调用的含义相混淆，试图理解这些代码时也会引起混乱。

然而在 C 中，使用库的最大的障碍是名字冲突问题。C 对于函数使用单个名字空间，所以当连接器找一个函数名时，它在一个单独的主表中查找，而当编译器在单个处理单元上工作时，它只能对带有某些特定名字的函数进行处理工作。

现在假设要支持从不同的厂商购买的两个库，并且每个库都有一个必须被初始化和清除的结构。两个厂商都认为initialize( )和cleanup( )是好名字。如果在某个处理单元中同时包含了这两个库文件，C 编译器怎么办呢？幸好，标准 C 给出一个出错，告诉在声明函数的两个不同的参数表中类型不匹配。即便不把它们包含在同一个处理单元中，连接器也会有问题。好的连接器会发现这里有名字冲突，但有些编译器仅仅通过查找目标文件表，按照在连接表中给出的次序，取第一个找到的函数名（实际上，这可以看作是一种功能，因为可以用自己的版本替换一个库函数）。

无论哪种情况，都不允许使用包含具有同名函数的两个库。为了解决这个问题， C 库厂商常常会在它们的所有函数名前加上一个独特字符串。所以， initialize( )和cleanup( )可能变为stash_initialize( )和stash_cleanup( )。这是合乎逻辑的，因为它“分解了”这个struct 的名字，而该函数以这样的函数名对这个struct操作。

现在，迈向 C++第一步的时候到了。我们知道， struct 内部的变量名不会与全局变量名冲突。而当一些函数在特定 struct上运算时，为什么不把这一优点扩展到这些函数名上呢？也就是，为什么不让函数是 struct 的成员呢？

2.5 基本对象

C++的第一步正是这样。函数可以放在结构内部，作为“成员函数”。在这里 stash 是：

```
//: LIBCPP.H -- C library converted to C++

struct stash {
  int size;  // Size of each space
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Dynamically allocated array of bytes:
  unsigned char* storage;
  // Functions!
  void initialize(int Size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
};
```

首先注意到的可能是新的注释文法//。这是对 C 风格注释的补充， C 原来的注释文法仍然能用。C++注释直到该行的结尾，它有时非常方便。另外，我们会在这本书中，在文件的第一行的//之后加一个冒号，后面跟的是这个文件名和简要描述。这就可以了解代码所在的文件。并且还可以很容易地用本书列出的名字从电子源代码中识别出这个文件。

其次，注意到这里没有typedef。在C++中，编译器不要求我们创建typedef，而是直接把结构名转变为这个程序的新类型名（就像int、char、float、double一样）。stash 的用法仍然相同。

所有的数据成员与以前完全相同，但现在这些函数在 struct的内部了。另外，注意到，对应于这个库的 C 版本中第一个参数已经去掉了。在C++中，不是硬性传递这个结构的地址作为在这个结构上运算的所有函数的一个参数，而是编译器背地里做了这件事。现在，这些函数仅有的参数与这些函数所做的事情有关，而不与这些函数运算的机制有关。

认识到这些函数代码与在 C 库中的那些同样有效，是很重要的。参数的个数是相同的（即便我们还没有看到这个结构地址被传进来，实际上它在这里），并且每个函数只有一个函数体。正因为如此，写：

```
stash A, B, C;
```

并不意味着每个变量得到不同的add( )函数。

被产生的代码几乎和我们已经为 C 库写的一样。有趣的是，这同时就包括了为过程stash_initialize( )、stash_cleanup( )等所做的“名字分解”。当函数在 struct 内时，编译器有效地做了相同的事情。因此，在 stash 内部的 initialize( ) 将不会与任何其他结构中的 initialize( ) 相抵触。大部分时间都不必为函数名字分解而担心——即使使用未分解的函数名。但有时还必须能够指出这个initialize( )属于这个 struct stash 而不属于任何其他的struct。特别是，定义这个函数时，需要完全指定它是哪一个。为了完成这个指定任务， C++有一个新的运算符::，即范围分解运算符（这样命名是因为名字现在能在不同的范围：在全局范围或在这个 struct 的范围）。例如，如果希望指定initialize( )属于 stash，就写stash::initialize(int Size, int Quantity)。对于stash 的C++版本，我们可以看到，在函数定义中如何使用范围分解运算符：

```
//: LIBCPP.CPP -- C library converted to C++
// Declare structure and functions:
#include "..\1\libcpp.h"
#include <assert.h> // Error testing macros
#include <stdlib.h> // Dynamic memory
#include <string.h> // memcpy()
#include <stdio.h>

void stash::initialize(int Size) {
  size = Size;
  quantity = 0;
  storage = 0;
  next = 0;
}

void stash::cleanup() {
  if(storage) {
    puts("freeing storage");
    free(storage);
  }
}

int stash::add(void* element) {
```

```
  if(next >= quantity) // Enough space left?
    inflate(100);
  // Copy element into storage,
  // starting at next empty space:
  memcpy(&(storage[next * size]),
         element, size);
  next++;
  return(next - 1); // Index number
}

void* stash::fetch(int index) {
  if(index >= next || index < 0)
    return 0;  // Not out of bounds?
  // Produce pointer to desired element:
  return &(storage[index * size]);
}

int stash::count() {
  return next; // Number of elements in stash
}

void stash::inflate(int increase) {
  void* v =
    realloc(storage, (quantity+increase)*size);
  assert(v);  // Was it successful?
  storage = (unsigned char*)v;
  quantity += increase;
}
```

这个文件有几个其他的事项要注意。首先，在头文件中的声明是由编译器要求的。在 C++ 中，不能调用未事先声明的函数，否则编译器将报告一个出错信息。这是确保这些函数调用在被调用点和被定义点之间一致的重要方法。通过强迫在调用之前必须声明， C++ 编译器可以保证我们用包含这个头文件的方式完成这个声明。如果我们在这个函数被定义的地方还包含有同样的头文件，则编译器将作一些检查以保证在这个头文件中的声明和这个定义匹配。这意味着，这个头文件变成了函数声明的有效的仓库，并且保证这些函数在项目中的所有处理单元中使用一致。

当然，全局函数仍然可以在定义和使用它的每个地方用手工方式声明（这是很乏味的，以致于变得不太可能）。然而，结构的声明必须在它们定义和使用之前，而放置结构定义的最习惯的位置是在头文件中，除非我们有意把它藏在代码文件中。

可以看到，除了范围分解和来自这个库的C版本的第一个参数不再是显式的这一事实以外，所有这些成员函数实际上都是一样的。当然，这个参数仍然存在，因为这个函数必须工作在一个特定的struct 变量上。但是，在成员函数内部，成员选择照常使用。这样，不写s->size = size，而写size = size。这就去除了并不能在此增加信息的冗余 s->。当然， C++ 编译器必须为我们做这些事情。实际上，它取“ 秘密 ”的第一个参数，并且当提到类的数据成员的任何时候，必须

应用成员选择器。这意味着，无论何时，当在另一个类的成员函数中时，我们可以通过简单地给出它的名字，提及任何成员（包括成员函数）。编译器在找出这个名字的全局版本之前先在局部结构的名字中搜索。这样这个性能意味着不仅代码更容易写，而且更容易阅读。

但是，如果因为某种原因，我们希望能够处理这个结构的地址，情况会怎么样呢？在这个库的 C 版本中，这是很容易的，因为每个函数的第一个参数是称作 S 的一个 stash* 。在 C++ 中，事情是更一致的。这里有一个特殊的关键字，称为 this，它产生这个struct的地址。它等价于这个库的 C 版本的 S 。所以我们可以用下面语句恢复成C风格。

```
this->size = Size;
```

对这种书写形式进行编译所产生的代码是完全一样的。通常，不经常用 this，只是需要时才使用。在这些定义中还有最后一个变化。在 C 库中的 inflate( )中，可以将void*赋给其他任何指针，例如

```
S->storage = v;
```

而且编译器能够通过。但在 C++ 中，这个语句是不允许的，为什么呢？因为在 C 中，可以给任何指针赋一个 void*（它是 malloc( )、calloc( )和 realloc( )的返回），而不需计算。C 对于类型信息不挑剔，所以它允许这种事情。 C++不同，类型在C++中是严格的，当类型信息有任何违例时，编译器就不允许。这一点一直是很重要的，而对于 C++尤其重要，因为在struct中有成员函数。如果我们能够在 C++ 中向struct传递指针而不被阻止，那么我们就能最终调用对于struct逻辑上并不存在的成员函数。这是防止灾难的一个实际的办法。因此， C++允许将任何类型的指针赋给 void*（这是 void* 的最初的意图，它要求 void* 足够大，以存放任何类型的指针），但不允许将void* 指针赋给任何其他类型的指针。 一项基本的要求是告诉读者和编译器，我们知道正在用的类型。这样，我们可以看到 calloc( ) 和 realloc( ) 的返回值严格地指派为(unsigned char*)。

这就带来了一个有趣的问题，C++的最重要的目的之一是能编译尽可能多的已存在的 C 代码，以便容易地向这个新语言过渡。那么在上述例子中如何使用标准 C 库函数？另外，所有的 C 运算符和表达式在 C++ 中都可用。但是，这并不意味着 C 允许的所有代码在 C++ 中也允许。有一些C编译器允许的东西是危险的和易出错的（本书中还会看到它们）。C++ 编译器对于这些情况产生警告和出错信息，这样将更有好处。实际上， C中有许多我们知道的错误只是不能找出它的原因，但是一旦用C++重编译这个程序，编译器就能指出这些问题。在 C 中，我们常常发现能使程序通过编译，然后我们必须再花力气使它工作。在 C++中，常常是，程序编译正确了，它也就能工作了。这是因为该语言对类型要求更严格的缘故。

在下面的测试程序中，可以看到 stash 的C++版本所使用的另一些东西。

```
//: LIBTEST.CPP -- Test of C++ library
#include "..\1\libcpp.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  stash intStash, stringStash;
  int i;
  FILE* file;
  char buf[BUFSIZE];
```

```
  char* cp;
  // ....
  intStash.initialize(sizeof(int));
  for(i = 0; i < 100; i++)
    intStash.add(&i);
  // Holds 80-character strings:
  stringStash.initialize(sizeof(char)*BUFSIZE);
  file = fopen("LIBTEST.CPP", "r");
  assert(file);
  while(fgets(buf, BUFSIZE, file))
    stringStash.add(buf);
  fclose(file);

  for(i = 0; i < intStash.count(); i++)
    printf("intStash.fetch(%d) = %d\n", i,
           *(int*)intStash.fetch(i));

  i = 0;
  while(
    (cp = (char*)stringStash.fetch(i++))!=0)
      printf("stringStash.fetch(%d) = %s",
             i - 1, cp);
  putchar('\n');
  intStash.cleanup();
  stringStash.cleanup();
}
```

这些代码与原来的代码相当类似，但在调用成员函数时，在函数名字之前使用成员运算符“.”。这是一个传统的文法，它模仿了结构数据成员的使用。所不同的是这里是函数成员，有一个参数表。

当然，该编译器实际产生的调用，看上去更像原来的库函数。如果我们考虑名字分解和this传递，C++函数调用 intStash.initialize(sizeof(int), 100) 就和 stash_initialize(&intStash, sizeof(int), 100)一样了。如果我们想知道在内部所进行的工作，可以回忆最早的 C++编译器cfront，它由AT&T开发，它输出的是C代码，然后再由C 编译器编译。这个方法意味着 cfront能使C++很快地转到有 C 编译器的机器上，有助于传播C++ 编译器技术。

在下面语句中我们还会注意到类型转化的情况。

```
while(cp = (char*)stringStash.fetch(i++))
```

这是由于在 C++ 中有严格类型检查而导致的结果。

2.6 什么是对象

我们已经看到了一个最初的例子，现在回过头来看一些术语。把函数放进结构是 C++ 中的根本改变，并且这引起我们将结构作为新概念去思考。在 C 中，结构是数据的凝聚，它将数据捆绑在一起，使得我们可以将它们看作一个包。但这除了能使程序设计方便之外，别无其他好处。这些结构上的运算可以用在别处。然而将函数也放在这个包内，结构就变成了新的创

造物，它既能描述属性（就像 C中的 struct 能做的一样），又能描述行为，这就形成了对象的概念。对象是一个独立的有约束的实体，有自己的记忆和活动。

“对象”和“面向对象的程序设计”（OOP）术语不是新的。第一个OOP语言是 Simula-67，于1967年由 Scandinavia发明，用于辅助解决建模问题。这些问题似乎总是包括一束相同的实体（诸如人、细菌、小汽车），它们为互相交互而忙碌。 Simula允许对实体创建一般的描述，描写它的属性和行为，然后取总的一束。在 Simula 中，这种“一般的描述”称为 class （类）（一个将在后面章节中看到的术语）。由类产生的大量的项称为对象。在 C++ 中，对象只是一个变量，最纯的定义是“存储的一个区域”。它是能存放数据的空间，并隐含着还有在这些数据上的运算。

不幸的是，对于各种语言，涉及这些术语时，并不完全一致，尽管它们是可以接受的。我们有时还会遇到面向对象语言是什么的争论，虽然到目前为止这已被认为是相当好的选择。还有一些语言是object-based（基于对象的），意味着它们有像 C++ 的结构加函数这样的对象，正如我们已经看到的。然而，这只是到达面向对象语言历程中的一部分，停留在把函数捆绑在结构内部的语言是基于对象的，而不是面向对象的。

2.7 抽象数据类型

将数据连同函数捆绑在一起，这一点就允许创建新的类型。这常常被称为封装 [1]。一个已存在的数据类型，例如 float，有几个数据块，一个指数，一个尾数和一个符号位。我们能够告诉它：与另一个 float 或 int 相加，等等。它有属性和行为。

stash 也是一个新数据类型，可以add( )、fetch( )和 inflate( )。由说明stash S创建一个 stash 就像由说明 float f 创建一个 float 一样。一个 stash 也有属性和行为，甚至它的活动就像一个实数——一个内建的数据类型。我们称stash为抽象数据类型（abstract dada type），也许这是因为它能允许我们从问题空间把概念抽象到解空间。另外，C++编译器把它看作一个新的数据类型，并且如果说一个函数需要一个 stash，编译器就确保传递了一个 stash 给这个函数。对抽象数据类型（有时称为用户定义类型）的类型检查就像对内建类型的类型检查一样严格。

然而，我们会看到在对象上完成运算的方法有所不同。 object.member_function(arglist)是对一个对象“调用一个成员函数”。而在面向对象的用法中，也称之为“向一个对象发送消息”。这样，对于 stash S，语句 S.add(&i)“发送消息给 S”，也就是说，“对自己 add( )”。事实上，面向对象程序设计可以总结为一句话，“向对象发送消息”。需要做的所有事情就是创建一束对象并且给它们发送消息。当然，问题是勾画出我们的对象和消息是什么，但如果完成了这些，C++ 的实现就直截了当了。

2.8 对象细节

这时我们大概和大多数 C 程序员一样会感到有点困惑，因为原有的 C 是非常低层和面向效率的语言。在研讨会上经常提出的一个问题是“对象应当多大和它应当像什么”。回答是“最好莫过于和我们希望来自 C的 struct一样”。事实上，C struct （不带有 C++ 装饰）在由 C 和 C++ 编译器产生的代码上完全相同，这可以使那些关心代码的安排和大小细节的 C程序员放心了。并且，由于某种原因，直接访问结构的字节，而不是使用标识符，不一定是效率更高的办法。

[1] 应当知道，这个术语似乎是有争议的题目。一些人就像这里的用法一样用它，而另一些人用它描述隐藏的实现，这将在第三章中讨论。

一个结构的大小是它的所有成员大小的和。有时，当一个 struct 被编译器处理时，会增加额外的字节以使得捆绑更整齐，这主要是为了提高执行效率。在第 14章和第16章中，将会看到如何在结构中增加“秘密”指针，但是现在不必关心这些。

用sizeof运算可以确定 struct 的长度。这里有一个小例子：

```
//: SIZEOF.CPP -- Sizes of structs
#include <stdio.h>
#include "..\1\lib.h"
#include "..\1\libcpp.h"
struct A {
  int I[100];
};

struct B {
  void f();
};

void B::f() {}

main() {
  printf("sizeof struct A = %d bytes\n",
         sizeof(A));
  printf("sizeof struct B = %d bytes\n",
         sizeof(B));
  printf("sizeof Stash in C = %d bytes\n",
         sizeof(Stash));
  printf("sizeof stash in C++ = %d bytes\n",
         sizeof(stash));
}
```

第一个打印语句产生的结果是200，因为每个 int 占二个字节。struct B 是奇异的，因为它是没有数据成员的 struct。在 C 中，这是不合法的，但在 C++ 中，以这种选择方式创建一个 struct，唯一的目的就是划定函数名的范围，所以这是允许的。尽管如此，由第二个 printf ()语句产生的结果是一个有点奇怪的非零值。在该语言较早的版本中，这个长度是零，但是，当创建这样的对象时出现了笨拙的情况：它们与紧跟着它们创建的对象有相同的地址，没有区别。这样，无数据成员的结构总应当有最小的非零长度。

最后两个 sizeof 语句表明在 C++ 中的结构长度与 C 中等价版本的长度相同。C++ 尽力不增加任何花费。

2.9 头文件形式

当我第一次学习用 C 编程时，头文件对我是神秘的。许多有关 C语言的书似乎不强调它，并且编译器也并不强调函数声明，所以它在大部分时间内似乎是可要可不要的，除非要声明结构时。在 C++ 中，头文件的使用变得非常清楚。它们对于每个程序开发是强制的，在它们中放入非常特殊的信息：声明。头文件告诉编译器在我们的库中哪些是可用的。因为对于 CPP 文件能够不要源代码而使用库（只需要对象文件或库文件），所以头文件是存放接口规范的唯

一地方。

头文件是库的开发者与它的用户之间的合同。它说：“这里描述的是库能做什么。”它不说如何做，因为如何做存放在CPP文件中，开发者不需要分发这些描述“如何做”的源代码给用户。

该合同描述数据结构，并说明函数调用的参数和返回值。用户需要这些信息来开发应用程序，编译器需要它们来产生相应的代码。

编译器强迫执行这一合同，也就是要求所有的结构和函数在它们使用之前被声明，当它们是成员函数时，在它们被定义之前被声明。这样，就强制把声明放在头文件中并把这个头文件包含在定义成员函数的文件和使用它们的文件中。因为描述库的单个头文件被包括在整个系统中，所以编译器能保证一致和避免错误。

为了恰当地组织代码和写有效的头文件，有一些问题必须知道。第一个问题是将什么放进头文件中。基本规则是“只声明”，也就是说，对于编译器只需要一些信息以产生代码或创建变量分配内存。这是因为，在一个项目中，头文件也许会包含在几个处理单元中，而如果内存分配不止一个地方，则连接器会产生多重定义错误。

这个规则不是非常严格的。如果在头文件中定义“静态文件”的一段数据（只在文件内可视），在这个项目中将有这个数据的多个实例，编译器不会报错。基本上，不要在头文件中做在连接时会引起混淆的任何事情。

关于头文件的第二个问题是重复声明。C 和 C++ 都允许对函数重复声明，只要这些重复声明匹配，但决不允许对结构重复声明。在 C++ 中，这个规则特别重要，因为，如果编译器允许对结构重复声明而且这两个重复声明又不一样，那么应当使用哪一个呢？

重复声明问题在 C++ 中很少出现，因为每个数据类型（带有函数的结构）一般有自己的头文件。但我们如果希望创建使用某个数据类型的另一个数据类型，必须在另一个头文件中包含它的头文件。在整个项目中，很可能有几个文件包含同一个头文件。在编译期间，编译器会几次看到同一个头文件。除非做适当的处理，否则编译器将认为是结构重复声明。

典型的防止方法是使用预处理器隔离这个头文件。如果有一个头文件名为 FOO.H，一般用“名字分解”产生预处理名，以防止多次包含这个头文件。FOO.H 的内部可以如下：

```
#ifndef FOO_H_
#define FOO_H_
// Rest of header here ...
#endif // FOO_H_
```

注意：不用前导下划线，因为标准 C 用前导下划线指明保留标识符。

在项目中使用头文件

用 C++ 建立项目时，我们通常要汇集大量不同的类型（带有相关函数的数据结构）。一般将每个类型或一组相关类型放在一个单独的头文件中，然后在一个处理单元中定义这个类型的函数。当使用这个类型时必须包含这个头文件，形成适当的声明。

有时这个模式会在本书中使用，但如果例子很小，结构声明、函数定义和 main( ) 函数可以出现在同一个文件中。应当记住，在实际上使用的是隔离的文件和头文件。

2.10 嵌套结构

在全局名字空间之外为数据和函数取名字是有好处的，可以将这种方式推广到对结构的处理

中。我们可以将一个结构嵌套在另一个中，这就可以将相关联的元素放在一起。声明文法在下面结构中可以看到，这个结构用非常简单的链接表方式实现了一个栈，所以它决不会运行越界。

```
//: NESTED.H -- Nested struct in linked list
#ifndef NESTED_H_
#define NESTED_H_

struct stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
  void initialize();
  void push(void* Data);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // NESTED_H_
```

这个嵌套 struct 称为 link，它包括指向这个表中的下一个 link 的指针和指向存放在 link 中的数据的指针，如果 next 指针是零，意味着表尾。

注意，head 指针紧接在 struct link 声明之后定义，而不是单独定义 link* head。这是来自C的文法，但它强调在结构声明之后的分号的重要性，分号表明这个结构类型的定义表结束（通常这个定义表是空的）。

正如到目前为止所有描述的结构一样，嵌套结构有它自己的 initialize( ) 函数。为了确保正确地初始化，stack 既有 initialize( ) 又有 cleanup( ) 函数。此外还有push( ) 函数，它取一个指向希望存放数据的存储单元（假设已经分配在堆中）；还有 pop( ),它返回栈顶的data指针，并去除栈顶元素（注意，我们对破坏 data 指针负有责任）；peek( )函数也从栈顶返回 data 指针，但是它在栈中保留这个栈顶元素。

cleanup 去除每个栈元素，并释放data 指针。

下面是一些函数的定义。

```
//: NESTED.CPP -- Linked list with nesting
#include <stdlib.h>
#include <assert.h>
#include "nested.h"

void stack::link::initialize(
    void* Data, link* Next) {
  data = Data;
  next = Next;
}

void stack::initialize() { head = 0; }
```

```
void stack::push(void* Data) {
  link* newlink = (link*)malloc(sizeof(link));
  assert(newlink);
  newlink->initialize(Data, head);
  head = newlink;
}

void* stack::peek() { return head->data; }

void* stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  link* oldHead = head;
  head = head->next;
  free(oldHead);
  return result;
}

void stack::cleanup() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    free(head->data); // Assumes a malloc!
    free(head);
   head = cursor;
  }
}
```

第一个定义特别有趣，因为它表明如何定义嵌套结构的成员。简单地两次使用范围分解运算符，以指明这个嵌套 struct 的名字。stack::link::initialize( ) 函数取参数并把参数赋给它的成员们。虽然用手工做这些事情相当容易，但是，我们将来会看到这个函数的不同的形式，所以它更有意义。

stack::initialize( ) 函数置 head 为零，使得这个对象知道它有一个空表。

stack::push( ) 取参数，也就是一个指向希望用这个 stack 保存的一块数据的指针，并且把这个指针放在栈顶。首先，使用 malloc( ) 为 link 分配空间，link将插入栈顶。然后调用 initialize( ) 函数对这个 link 的成员赋适当的值。注意，next 指针赋为当前的 head，而 head 赋为新link 指针。这就有效地将 link 放在这个表的顶部了。

stack::pop( )取出当前在该栈顶部的 data 指针，然后向下移 head 指针，删除该栈老的栈顶元素。stack::cleanup( )创建cursor 在整个栈上移动，用free( )释放每个link的data和link 本身。

下面是测试这个栈的例子。

```
//: NESTEST.CPP -- Test of nested linked list
#include "..\1\nested.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```

```
main(int argc, char** argv) {
  stack textlines;
  FILE* file;
  char* s;
  #define BUFSIZE 100
  char buf[BUFSIZE];
  assert(argc == 2); // File name is argument
  textlines.initialize();
  file = fopen(argv[1], "r");
  assert(file);
  // Read file and store lines in the stack:
  while(fgets(buf, BUFSIZE, file)) {
    char* string =(char*)malloc(strlen(buf)+1);
    assert(string);
    strcpy(string, buf);
    textlines.push(string);
  }
  // Pop the lines from the stack and print them:
  while((s = (char*)textlines.pop()) != 0) {
    printf("%s", s); free(s); }
  textlines.cleanup();
}
```

这个例子与前面一个非常类似，这些行存放到这个栈中，然后弹出它们，这会使这个文件被反序打印出。另外，要打开文件的文件名是从命令行中取出的。

全局范围分解

编译器通过缺省选择的名字（“最近”的名字）可能不是我们所希望的，范围分解运算符可以避免这种情况。例如，假设有一个结构，它的局域标识符为 A，希望在成员函数内选全局标识符 A。编译器会缺省地选择局域的那一个，所以必须另外告诉编译器。希望用范围分解指定一个全局名字时，应当使用前面不带任何东西的运算符。这里有一个例子，表明对变量和函数的全局范围分解。

```
//: SCOPERES.CPP -- Global scope resolution
int A;
void f() {}

struct S {
  int A;
  void f();
};

void S::f() {
  ::f();  // Would be recursive otherwise!
  ::A++;  // Select the global A
  A--;    // The A at struct scope
```

```
}

main() {}
```

如果在 S::f( ) 中没有范围分解，编译器会缺省地选择 f( ) 和 A 的成员版本。

2.11 小结

在这一章中，我们已经学会了使用C++ 的基本方法，也就是在结构的内部放入函数。这种新类型被称为抽象数据类型，用这种结构创建的变量被称为这个类型的对象或实例。向对象调用成员函数被称为向这个对象发消息。面向对象的程序设计中的主要活动就是向对象发消息。

虽然将数据和函数捆绑在一起很有好处，并使得库更容易使用，因为这可以通过隐藏名字防止名字冲突，但是，还有大量的工作可以使 C++程序设计更安全。在下面一章中，将学习如何保护 struct 的一些成员，以使得只有我们能对它们操作。这就在什么是结构的用户可以改动的和什么只是程序员可以改动的之间形成了明确的界线。

2.12 练习

1) 创建一个 struct 声明，它有单个成员函数，然后对这个成员函数创建定义。创建这个新数据类型的对象，再调用这个成员函数。

2) 编写并且编译一段代码，这段代码完成数据成员选择和函数调用。

3) 写一个在另一个结构中的被声明结构的例子（嵌套结构）。并说明如何定义这个结构的成员。

4) 结构有多大？写一段能打印各个结构的长度的代码。创建一些结构，它们只有数据成员，另一些有数据成员和函数成员。然后创建一个结构，它根本没有成员。打印出所有这些结构的长度。对于根本没有成员的结构的结果作出解释。

5) C++对于枚举、联合和struct 自动创建 typedef 的等价物，正如在本章中看到的。写一个能说明这一点的小程序。