第4章　初始化与清除

第2章利用了一些分散的典型C语言库的构件，并把它们封装在一个struct中，从而在库的应用方面做了有意义的改进。（从现在起，这个抽象数据类型称为类）。

这样不仅为库构件提供了单一一致的入口指针，也用类名隐藏了类内部的函数名。在第3章中，我们介绍了存取控制（隐藏实现），这就为类的设计者提供了一种设立界线的途径，通过界线的设立来决定哪些是用户可以处理的，哪些是禁止的。这意味着数据类型的内部机制对设计者来说是可控的和能自行处理的。这样让用户也清楚哪些成员是他们能够使用并加以注意的。

封装和实现的隐藏大大地改善了库的使用。它们提供的新的数据类型的概念在某些方面比从C中继承的嵌入式数据类型要好。现在C++编译器可以为这种新的数据类型提供类型检查，这样在使用这种数据类型时就确保了一定的安全性。

当然，说到安全性，C++的编译器能比C编译器提供更多的功能。在本章及以后的章节中，我们将看到许多C++的另外一些性能。它们可以让我们程序中的错误暴露无遗，有时甚至在我们编译这个程序之前，帮我们查出错误，但通常是编译器的警告和出错信息。所以我们不久就会习惯：在第一次编译时总听不到编译器那意味着正确的提示音。

安全性包括初始化和清除两个方面。在C语言中，如果程序员忘记了初始化或清除一个变量，就会导致一大段程序错误。这在一个库中尤其如此，特别是当用户不知如何对一个struct初始化，甚至不知道必须要初始化时。（库中通常不包含初始化函数，所以用户不得不手工初始化struct）。清除是一个特殊问题，因为C程序员一旦用过了一个变量后就把它忘记了，所以对一个库的struct来说，必要的清除工作往往被遗忘了。

在C++中，初始化和清除的概念是简化类库使用的关键所在，并可以减少那些由于用户忘记这些操作而引起的许多细微错误。本章就来讨论C++的这些特征。

4.1　用构造函数确保初始化

在stash和stack类中都曾调用initialize()函数，这暗示无论用什么方法使用这些类的对象，在使用之前都应当调用这一函数。很不幸的是，这要求用户必须正确地初始化。而用户在专注于用那令人惊奇的库来解决他们的问题的时候，往往忽视了这些细节。在C++中，初始化实在太重要了，所以不能留给用户来完成。类的设计者可以通过提供一个叫做构造函数的特殊函数来保证每个对象都正确的初始化。如果一个类有构造函数，编译器在创建对象时就自动调用这一函数，这一切在用户使用他们的对象之前就已经完成了。对用户来说，是否调用构造函数并不是可选的，它是由编译器在对象定义时完成的。

接下来的问题是这个函数叫什么名字。这必须考虑两点，首先这个名字不能与类的其他成员函数冲突，其次，因为该函数是由编译器调用的，所以编译器必须总能知道调用哪个函数。Stroustrup的方法似乎是最容易也是最符合逻辑的：构造函数的名字与类的名字一样。这使得这样的函数在初始化时自动被调用。

下面是一个带构造函数的类的简单例子：

```
class X {
  int i;
public:
  X();  // constructor
};
```

现在当一个对象被定义时：

```
void f() {
  X a;
  // ...
}
```

这时就好像a是一个整数一样：为这个对象分配内存。但是当程序执行到 a的定义点时，构造函数自动被调用，因为编译器已悄悄地在 a的定义点处插入了一个 X::X()的调用。就像其他成员函数被调用一样。传递到构造函数的第一个参数（隐含）是调用这一函数对象的地址。

像其他函数一样，我们也可以通过构造函数传递参数，指定对象该如何创建，设定对象初始值等等。构造函数的参数保证对象的所有部分都被初始化成合适的值。举例来说：如果类tree有一个带整型参数的构造函数，用以指定树的高度，那么我们就必须这样来创建一个对象：

```
tree  t(12);  // 12英尺高的树
```

如果tree(int)是唯一的构造函数，编译器将不会用其他方法来创建一个对象（在下一章我们将看到多个构造函数以及调用它们的不同方法）。

关于构造函数，我们就全部介绍完了。构造函数是一个有着特殊名字，由编译器自动为每个对象调用的函数，然而它解决了类的很多问题，并使得代码更容易阅读。例如在上一个代码段中，对有些initialize()函数我们并没有看到显式的调用，这些函数从概念上说是与定义分开的。在C++中，定义和初始化是同一概念，不能只取其中之一。

构造函数和析构函数是两个非常特殊的函数：它们没有返回值。这与返回值为 void的函数显然不同。后者虽然也不返回任何值，但我们还可以让它做点别的。而构造函数和析构函数则不允许。在程序中创建和消除一个对象的行为非常特殊，就像出生和死亡，而且总是由编译器来调用这些函数以确保它们被执行。如果它们有返回值，要么编译器必须知道如何处理返回值，要么就只能由用户自己来显式地调用构造函数与析构函数，这样一来，安全性就被破坏了。

4.2 用析构函数确保清除

作为一个C程序员，我们可能经常想到初始化的重要性，但很少想到清除的重要性。毕竟，清除一个整型变量时需要作什么 ?只需要忘记它。然而，在一个库中，对于一个曾经用过的对象，仅仅“忘记它”是不安全的。如果它修改了某些硬件参数，或者在屏幕上显示了一些字符，或在堆中分配了一些内存，那么将会发生什么呢？如果我们只是“忘记它”，我们的对象就永远不会消失。在C++中，清除就像初始化一样重要。通过析构函数来保证清除的执行。

析构函数的语法与构造函数一样，用类的名字作函数名。然而析构函数前面加上一个 ~，以和构造函数区别。另外，析构函数不带任何参数，因为析构不需任何选项。下面是一个析构函数的声明：

```
class Y {
public:
  ~Y();
};
```

当对象超出它的定义范围时，编译器自动调用析构函数。我们可以看到，在对象的定义点处构造函数被调用，但析构函数调用的唯一根据是包含该对象的右括号，即使用 goto语句跳出这一程序块（为了与C 语言向后兼容，goto在C++中仍然存在，当然也是为了方便）。我们应该注意一些非本地的goto语句，它们用标准C语言库中的setjmp() 和longjmp()函数，这些函数将不会引发析构函数的调用。（这里作一点说明：有的编译器可能并不用这种方法来实现。依赖那些不在说明书中的特征意味着这样的代码是不可移植的）。

下例说明了构造函数与析构函数的上述特征：

```
//: CONSTR1.CPP -- Constructors & destructors
#include <stdio.h>

class tree {
  int height;
public:
  tree(int initialHeight);  // Constructor
  ~tree();  // Destructor
  void grow(int years);
  void printsize();
};

tree::tree(int initialHeight) {
  height = initialHeight;
}

tree::~tree() {
  puts("inside tree destructor");
  printsize();
}

void tree::grow(int years) {
  height += years;
}

void tree::printsize() {
  printf("tree height is %d\n", height);
}

main() {
  puts("before opening brace");
  {
    tree t(12);
    puts("after tree creation");
    t.printsize();
    t.grow(4);
    puts("before closing brace");
```

```
  }
  puts("after closing brace");
}
```

下面是上面程序的输出结果：

```
before opening brace
after tree creation
tree height is 12
before closing brace
inside tree destructor
tree height is 16
after closing brace
```

我们可以看到析构函数在包括它的右括号处被调用。

4.3 清除定义块

在C中，我们总要在一个程序块的左括号一开始就定义好所有的变量，这在程序设计语言中不算少见（Pascal中例外），其理由无非是因为“这是一个好的编程风格”。在这点上，我有自己的看法。我认为它总是给我带来不便。作为一个程序员，每当我需要增加一个变量时我都得跳到块的开始，我发现如果变量定义紧靠着变量的使用处时，程序的可读性更强。

也许这些争论具有一定的普遍性。在C++中，是否一定要在块的开头就定义所有变量成了一个很突出的问题。如果存在构造函数，那么当对象产生时它必须首先被调用，如果构造函数带有一个或者更多个初始化参数，我们怎么知道在块的开头定义这些初始化信息呢？在一般的编程情况下，我们做不到这点，因为C中没有私有成员的概念。这样很容易将定义与初始化部分分开，然而C++要保证在一个对象产生时，它同时被初始化。这可以保证我们的系统中没有未初始化的对象。C并不关心这些。事实上，C要求我们在块的开头就定义所有变量，在我们还不知道一些必要的初始化信息时，就要求我们这样做是鼓励我们不初始化变量。

通常，在C++中，在还不拥有构造函数的初始化信息时不能创建一个对象，所以不必在块的开头定义所有变量。事实上，这种语言风格似乎鼓励我们把对象的定义放得离使用点尽可能近一点。在C++中，对一个对象适用的所有规则，对预定义类型也同样适用。这意味着任何类的对象或者预定义类型都可以在块的任何地点定义。这也意味着我们可以等到我们已经知道一个变量的必要信息时再去定义它，所以我们总是可以同时定义和初始化一个变量。

```
//: DEFINIT.CPP -- Defining variables anywhere
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>

class G {
  int i;
public:
  G(int I);
};

G::G(int I) { i = I; }
```

```
main() {
  #define SZ 100
  char buf[SZ];
  printf("initialization value? ");
  int retval = (int)gets(buf);
  assert(retval);
  int x = atoi(buf);
  int y = x + 3;
  G g(y);
}
```

我们可以看到首先是buf被定义，然后是一些语句，然后x被定义并用一个函数调用对它初始化，然后y和g被定义。在C中这些变量都只能在块的一开始定义。一般说来，应该在尽可能靠近变量的使用点定义变量，并在定义时就初始化（这是对预定义类型的一种建议，但在那里可以不做初始化）。这是出于安全性的考虑，减少变量误用的可能性。另外，程序的可读性也增强了，因为读者不需要跳到程序头去确定变量的类型。

4.3.1 for循环

在C++中，我们将经常看到for循环的计数器直接在for表达式中定义：

```
for(int j = 0; j < 100; j++) {
    printf("j = %d\n", j);
}
for(int i = 0; i < 100; i++)
    printf("i = %d\n", i);
```

上述声明是一种重要的特殊情况，这可能使那些刚接触C++的程序员感到迷惑不解。

变量i和j都是在for表达式中直接定义的（在C中我们不能这样做），然后他们就作为一个变量在for循环中使用。这给程序员带来很大的方便，因为从上下文中我们可以清楚地知道变量 i、j的作用，所以不必再用诸如i_loop_counter之类的名字来定义一个变量，以表示这一变量的作用。

这里有一个变量生存期的问题，在以前这是由程序块的右大括号来确定的。从编译器的角度来看这样是合理的，因为作为程序员，我们显然想让 i只在循环内部有效。然而很不幸的是，如果我们用这种方法声明：

```
for(int i = 0; i < 100; i++)
    printf("i = %d\n", i);
// ....
for(int i = 0; i < 100; i++){
    printf("i = %d\n", i);
}
```

（无论有没有大括号，）在同一程序块内，编译器将给出一个重复定义的错误，而新的标准C++说明书上说，一个在for循环的控制表达式中定义的循环计数器只在该循环内才有效，所以上面的声明是可行的。（当然，并不是所有的编译器都支持这一点，我们可能会遇到一些老式风格的编译器。）如果这种转变引起一些错误的话，编译器会指出，解决起来也很容易。注意，

那些局部变量会屏蔽这个封闭范围中的变量。

我发现一个在小范围内设计良好的指示器：如果我们的一个函数有好几页，也许我们正在试图让这个函数完成太多的工作。用更多的细化的函数不仅有用，而且更容易发现错误。

4.3.2 空间分配

现在，一个变量可以在某个程序范围内的任何地方定义，所以在这个变量的定义之前是无法对它分配内存空间的。通常，编译器更可能像C编译器一样，在一个程序块的开头就分配所有的内存。这些对我们来说是无关紧要的，因为作为一个程序员，我们在变量定义之前总是无法得到存储空间。即使存储空间在块的一开始就被分配，构造函数也仍然要到对象的定义时才会被调用，因为标识符只有到此时才有效。编译器甚至会检查我们有没有把一个对象的定义放到一个条件块中，比如在switch块中声明，或可能被goto跳过的地方。

下例中解除注释的句子会导致一个警告或一个错误。

```
//: NOJUMP.CPP -- Can't jump past constructors

class X {
public:
  X() {}
};

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Error: goto bypasses init
  }
  X x1;  // Constructor called here
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Constructor called here
      break;
  //! case 2 : // Error: case bypasses init
      X x3;  // Constructor called here
      break;
  }
}

main() {}
```

在上面的代码中，goto和switch都可能跳过构造函数的调用点，然而这个对象会在后面的程序块中起作用，这样，构造函数就没有被调用，所以编译器给出了一条出错信息。这就确保了对象在产生的同时被初始化。

当然，这里讨论的内存分配都是在一个堆栈中。内存分配是通过编译器向下移动堆栈指针来实现的（这只是相对而言，实际指针值可能增加，也可能减少，这依赖于机器）。也可以在堆中分配对象的内存，这将在第12章中介绍。

4.4 含有构造函数和析构函数的stash

在前几章的例子中，都有一些很明显的函数对应为构造函数和析构函数： initialize()和cleanup()。下面是带有构造函数与析构函数的stash头文件。

```
//: STASH3.H -- With constructors & destructors
#ifndef STASH3_H_
#define STASH3_H_

class 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;
  void inflate(int increase);
public:
  stash(int Size);
  ~stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H_
```

下面是实现文件，这里只对initialize()和cleanup()的定义作了修改，它们分别用构造函数与析构函数代替。

```
//: STASH3.CPP -- Constructors & destructors
#include "..\3\stash3.h"
#include <assert.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

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

stash::~stash() {
  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;
}
```

注意，在下面的测试程序中，stash对象的定义放在紧靠对象调用的地方，对象的初始化通过构造函数的参数列表来实现，而对象的初始化似乎成了对象定义的一部分。

```
//: STSHTST3.CPP -- Constructors & destructors
#include "..\3\stash3.h"
#include <stdio.h>
#include <assert.h>
#define BUFSIZE 80

main() {
  stash intStash(sizeof(int));
  for(int j = 0; j < 100; j++)
    intStash.add(&j);

  FILE* file = fopen("STASHTST.CPP", "r");
  assert(file);
```

```
// Holds 80-character strings:
stash stringStash(sizeof(char) * BUFSIZE);
char buf[BUFSIZE];
while(fgets(buf, BUFSIZE, file))
  stringStash.add(buf);
fclose(file);
  for(int k = 0; k < intStash.count(); k++)
    printf("intStash.fetch(%d) = %d\n", k,
           *(int*)intStash.fetch(k));

  for(int i = 0; i < stringStash.count(); i++)
    printf("stringStash.fetch(%d) = %s",
           i, (char*)stringStash.fetch(i++));
  putchar('\n');
}
```

再看看cleanup()调用已被取消，但当intStash和stringStash越出程序块的范围时，析构函数被自动地调用了。

4.5 含有构造函数和析构函数的stack

重新实现含有构造函数和析构函数的链表（在stack内）。这是修改后的头文件：

```
//: STACK3.H -- With constructors/destructors
#ifndef STACK3_H_
#define STACK3_H_

class stack {
  struct link {
    void* data;
    link* next;
    void initialize(void* Data, link* Next);
  } * head;
public:
  stack();
  ~stack();
  void push(void* Data);
  void* peek();
  void* pop();
};
#endif // STACK3_H_
```

注意，虽然stack有构造函数与析构函数，但嵌套类link并没有，这并不是说它不需要。当它被使用时，问题就来了：

```
//: STACK3.CPP -- Constructors/destructors
#include <stdlib.h>
#include <assert.h>
```

```
#include "..\3\stack3.h"

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

stack::stack() { head = 0; }

void stack::push(void* Data) {
  // Can't use a constructor with malloc!
  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;
}

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

link是在stack::push内部产生的，但它是创建在堆栈上的，这儿就产生了一个疑难问题。

如果一个对象有构造函数，我们怎么创建它呢？到目前为止，我们一直这样说：“好吧，这是堆中的一块内存，我想您就假定它是给这个对象的吧。”但构造函数并不允许我们就这样把一个内存地址交给它来创建一个对象[1]。对象的创建很关键，C++的构造函数想控制整个过

[1] 实际上，确有允许这么做的语法。但它是在特殊情况下使用的，不能解决在此描述的一般问题。

程以保证安全。有个很容易的解决办法，那就是用new操作符，我们将在第12章讨论这个问题。现在，只要用C中的动态内存分配就行了。因为内存分配与清除都隐藏在stack中，它是实现部分，我们在测试程序中看不到它的影响。

```
//: STKTST3.CPP -- Constructors/destructors
#include "..\3\stack3.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

main(int argc, char** argv) {
  assert(argc == 2); // File name is argument
  FILE* file = fopen(argv[1], "r");
  assert(file);
  #define BUFSIZE 100
  char buf[BUFSIZE];
  stack textlines;  // Constructor called here
  // 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 lines from the stack and print them:
  char* s;
  while((s = (char*)textlines.pop()) != 0) {
    printf("%s", s); free(s); }
}  // Destructor called here
```

textlines的构造函数与析构函数都是自动调用的，所以类的用户只要把精力集中于怎样使用这些对象上，而不需要担心它们是否已被正确地初始化和清除了。

4.6 集合初始化

集合，顾名思义，就是多个事物聚集在一起。这个定义包括各种类型的集合：像 struct和class等。数组就是单一类型的集合。

初始化集合往往既冗长又容易出错。而C++中集合的初始化却变得很方便而且很安全。当我们产生一个集合对像时，我们要做的只是指定初始值就行了，然后初始化工作就由编译器去承担了。这种指定可以用几种不同的风格，取决于我们正在处理的集合类型。但不管是哪种情况，指定的初值都要用大括号括起来。比如一个预定义类型的数组可以这样定义：

```
int a[5]={1,2,3,4,5};
```

如果给出的初始化值多于数组元素的个数，编译器就会给出一条出错信息。但如果给的初

始化值少于数组元素的个数，那将会怎么样呢？例如：

```
int b[6]={0};
```

这时，编译器会把第一个初始化值赋给数组的第一个元素，然后用 0赋给其余的元素。注意，如果我们定义了一个数组而没有给出一列初始值时，编译器并不会去做这些工作。所以上面的表达式是将一个数组初始化为零的简洁方法，它不需要用一个 for循环，也避免了“偏移1位”错误（它可能比for循环更有效，这依赖于编译器）。

数组还有一种叫自动计数的快速初始化方法，就是让编译器按初始化值的个数去决定数组的大小：

```
int c[] = {1,2,3,4};
```

现在，如果我们决定增加其他的元素到这个数组上，只要增加一个初始化值即可，如果以此建立我们的代码，只需在一处作出修改即可，这样，我们在修改时出错的机会就减少了。但怎样确定这个数组的大小呢？用表达式sizeof c/sizeof *c(整个数组的大小除以第一个元素的大小)即可算出，这样，当数组大小改变时它无需修改。

```
for(int i = 0; i< sizeof c / sizeof *c; i++)
  c[i]++;
```

struct也是一种集合类型，它们也可以用同样的方式初始化。因为 C风格的struct的所有成员都是公共型的，所以它们的值可以直接指定：

```
struct X {
  int i;
  float f;
  char c;
};
X x1 = {1,2.2,'c' };
```

如果我们有一个这种struct的数组，我们也可以用嵌套的大括号来初始化每一个对象。

```
X x2[3] = {{1,1.1, 'a'},{2,2.2, 'b'}};
```

这里，第三个对象被初始化为零。

如果struct中有私有成员，或即使所有成员都是公共成员，但有一个构造函数，情况就不一样了。在上例中，初始值被直接赋给了集合中的每个元素，但构造函数是通过外在的接口来强制初始化的。这里，构造函数必须被调用来完成初始化，因此，如果有一个下面的 struct类型：

```
struct Y {
  float f;
  int i;
  Y(int A); // presumably assigned to i
};
```

我们必须指示构造函数调用，最好的方法像下面这样：

```
Y y2[] = {Y(1),Y(2),Y(3)};
```

这样我们就得到了三个对象和进行了三次构造函数调用。只要有构造函数，无论是所有成员都是公共的struct还是一个带私有成员的class,所有的初始化工作都必须通过构造函数，即使我们正在对一个集合初始化。

下面是构造函数带多个参数的又一个例子：

```
//: MULTIARG.CPP -- Multiple constructor arguments
// with aggregate initialization

class X {
  int i, j;
public:
  X(int I, int J) {
    i = I;
    j = J;
  }
};

main() {
  X xx[] = { X(1,2), X(3,4), X(5,6), X(7,8) };
}
```

注意，它看上去就像数组中的每个对象都对一个没有命名的构造函数调用了一次一样。

4.7 缺省构造函数

缺省构造函数就是不带任何参数的构造函数。缺省的构造函数用来创建一个“香草（vanilla）对象”,当编译器需要创建一个对象而又不知任何细节时，缺省的构造函数就显得非常重要。比如，我们有一个类Y，并用它来定义对象：

```
Y y4[2] = {Y(1)}
```

编译器就会报告找不到缺省的构造函数，数组中的第二个对象想不带参数来创建，所以编译器就去找缺省的构造函数。实际上，如果我们只是简单地定义了一个Y对象的数组：

```
Y y5[7] ;
```

或一个单一的对象

```
Y y;
```

编译器会报告同样的错误，因为它必须用一个缺省的构造函数去初始化数组中的每个对象。（记住，一旦有了一个构造函数，编译器就会确保不管在什么情况下它总会被调用）。

缺省的构造函数是如此重要，所以在一个构造类型（ struct 或class）中没有构造函数时，编译器会自动创建一个。因此下面例子将会正常运行：

```
class Z {
  int i; // private
}; // no constructor

Z z,z2[10];
```

然而，一旦有构造函数而没有缺省构造函数，上面的对象定义就会产生一个编译错误。

我们可能会想，缺省构造函数应该可以做一些智能化的初始化工作，比如把对象的所有内存置零。但事实并非如此。因为这样会增加额外的负担，而且使程序员无法控制。比如，如果我们把在C中编译过的代码用在C++中，就会导致不同的结果。如果我们想把内存初始化为零，必须亲自去做。

对一个C++的新手来说，自动产生的缺省构造函数并不会使编程更容易。它实际上要求与已有的C代码保持向后兼容。这是C++中的一个关键问题。在C中，创建一个struct数组的情况

很常见，而在C++中，在没有缺省构造函数时，这会引起一个编译错误。

如果我们仅仅因为风格问题就去修改我们的 C代码，然后用C++重新编译，也许我们会很不乐意。当将C代码在C++中编译时，我们总会遇到这样、那样的编译错误，但这些错误都是C++编译器所发现的C的不良代码，因为C++的规则更严格。事实上，用C++编译器去编译C代码是一个发现潜在错误的很好的方法。

4.8 小结

由C++提供的细致精巧机制应给我们这样一个强烈的暗示：在这个语言中，初始化和清除是多么至关重要。在Stroustrup设计C++时，他所作的第一个有关C语言产品的观察就是，由于没有适当地初始化变量，从而导致了程序不可移植的问题。这种错误很难发现。同样的问题也出现在变量的清除上。因为构造函数与析构函数让我们保证正确地初始化和清除对象（编译器将不允许没有调用构造函数与析构函数就直接创建与销毁一个对象），使我们得到了完全的控制与安全。

集合的初始化同样如此——它防止我们犯那种初始化内部数据类型集合时常犯的错误，使我们的代码更简洁。

编码期间的安全性是C++中的一大问题，初始化和清除是这其中的一个重要部分，随着本书的进展，我们可以看到其他的安全性问题。

4.9 练习

1) 用构造函数与析构函数修改第3章结尾处的HANDLE.H,HANDLE.CPP 和USEHANDL.CPP文件。

2) 创建一个带非缺省构造函数和析构函数的类，这些函数都显示一些信息来表示它们的存在。写一段代码说明构造函数与析构函数何时被调用。

3) 用上题中的类创建一个数组来说明自动计数与集合初始化。在这个类中增加一个显示信息的成员函数。计算数组的大小并逐个访问它们，调用新成员函数。

4) 创建一个没有任何构造函数的类，显示我们可以用缺省的构造函数创建对象。现在为这个类创建一个非缺省的构造函数（带一个参数），试着再编译一次。解释发生的现象。