第12章　动态对象创建

有时我们能知道程序中对象的确切数量、类型和生命期。但情况并不总是这样。

空中交通系统必须处理多少架飞机？一个CAD系统需要多少个形状？在一个网络中有多少个节点？

为了解决这个普通的编程问题，在运行时能创建和销毁对象是基本的要求。当然， C已提供了动态内存分配函数 malloc( )和free( )（以及malloc( )的变种），这些函数在运行时从堆中（也称自由内存）分配存储单元。

然而，在C++中这些函数不能很好地运行。构造函数不允许通过向对象传递内存地址来初始化它。如果那么做了，我们可能

1) 忘记了。则对象初始化在C++中难以保证。

2) 期望某事发生，但结果是在给对象初始化之前意外地对对象作了某种改变。

3) 把错误规模的对象传递给了它。

当然，即使我们把每件事都做得很正确，修改我们的程序的人也容易犯同样的错误。不正确的初始化是编程出错的主要原因，所以在堆上创建对象时，确保构造函数调用是特别重要的。

C++是如何保证正确的初始化和清理并允许我们在堆上动态创建对象的呢？

答案是使动态对象创建成为语言的核心。 malloc( )和free( )是库函数，因此不在编译器控制范围之内。如果我们有一个能完成动态内存分配及初始化工作的运算符和另一个能完成清理及释放内存工作的运算符，编译器就可以保证所有对象的构造函数和析构函数都会被调用。

在本章中，我们将明白C++的new和delete是如何通过在堆上安全创建对象来出色地解决这个问题的。

12.1　对象创建

当一个C++对象被创建时，有两件事会发生。

1) 为对象分配内存。

2) 调用构造函数来初始化那个内存。

到目前为止，我们应该确保步骤 2）一定发生。C++强迫这样做是因为未初始化的对象是程序出错的主要原因。不用关心对象在哪里创建和如何创建的 —— 构造函数总是被调用。

然而，步骤1）可以以几种方式或在可选择的时间内发生：

1) 静态存储区域，存储空间在程序开始之前就可以分配。这个存储空间在程序的整个运行期间都存在。

2) 无论何时到达一个特殊的执行点（左花括号）时，存储单元都可以在栈上被创建。出了执行点（右花括号），这个存储单元自动被释放。这些栈分配运算内置在处理器的指令集中，非常有效。然而，在写程序的时候，必须知道需要多少个存储单元，以使编译器生成正确的指令。

3) 存储单元也可以从一块称为堆（也可称为自由存储单元）的地方分配。这称为动态内存分配，在运行时调用程序分配这些内存。这意味着可以在任何时候分配内存和决定需要多少内

存。我们也负责决定何时释放内存。这块内存的生存期由我们选择决定——而不受范围限制。

这三个区域经常被放在一块连续的物理存储单元里：静态内存、堆栈和堆（由编译器的作者决定它们的顺序），但没有一定的规则。堆栈可以在某一特定的地方，堆的实现可以通过调用由运算系统分配的一块存储单元来完成。对于一个程序设计者，这三件事无须我们来完成，所以我们所要思考的是什么时候申请内存。

12.1.1 C从堆中获取存储单元的方法

为了在运行时动态分配内存，C在它的标准库函数中提供了一些函数：从堆中申请内存的函数malloc( )以及它的变种calloc( )和realloc( )；释放内存返回给堆的函数free( )。这些函数是有效的但较原始，需要编程人员理解和小心使用。对使用C的动态内存分配函数创建一个类的实例，必须做：

```
//: MALCLASS.CPP -- Malloc with class objects
// What you'd have to do if not for "new"
#include <stdlib.h> // Malloc() & free()
#include <string.h> // Memset()
#include "..\allege.h"
#include <iostream.h>

class obj {
  int i, j, k;
  enum { sz = 100 };
  char buf[sz];
public:
  void initialize() { // Can't use constructor
    cout << "initializing obj" << endl;
    i = j = k = 0;
    memset(buf, 0, sz);
  }
  void destroy() { // Can't use destructor
    cout << "destroying obj" << endl;
  }
};

main() {
  obj* Obj = (obj*)malloc(sizeof(obj));
  allegemem(Obj);
  Obj->initialize();
  // ... sometime later:
  Obj->destroy();
  free(Obj);
}
```

在下面这行代码中，我们可以看到使用malloc( )为对象分配内存：

```
obj* Obj = (obj*)malloc(sizeof(obj)) ;
```

这里用户必须决定对象的长度（这也是程序出错原因之一）。因为它是一块内存而不是一个对象，所以malloc( )返回一个void*。C++不允许将一个void* 赋予任何指针，所以必须映射。

因为malloc( )可能找不到可分配的内存（在这种情况下它返回 0），所以必须检查返回的指针以确信内存分配成功。

但最坏的是：

```
Obj->initialize( ) ;
```

用户在使用对象之前必须记住对它初始化。注意构造函数没有被使用，因为构造函数不能被显式地调用 —— 而是当对象创建时由编译器调用。这里的问题是现在用户可能在使用对象时忘记执行初始化，因此这也是引入程序缺陷的主要来源。

许多程序设计者发现C的动态内存分配函数太复杂，令人混淆。所以， C程序设计者常常在静态内存区域使用虚拟内存机制分配很大的变量数组以避免使用动态内存分配。因为 C++能让一般的程序员安全使用库函数而不费力，所以应当避免使用 C的动态内存方法。

12.1.2 运算符new

C++中的解决方案是把创建一个对象所需的所有动作都结合在一个称为 new的运算符里。当用new（new的表达式）创建一个对象时，它就在堆里为对象分配内存并为这块内存调用构造函数。因此，如果我们写出下面的表达式

```
foo *fp = new foo(1,2) ;
```

在运行时等价于调用 malloc(sizeof(foo))，并使用（1，2）作为参数表来为 foo调用构造函数，返回值作为this指针的结果地址。在该指针被赋给 fp之前，它是不定的、未初始化的对象 —— 在这之前我们甚至不能触及它。它自动地被赋予正确的 foo类型，所以不必进行映射。

缺省的new还检查以确信在传递地址给构造函数之前内存分配是成功的，所以我们不必显式地确定调用是否成功。在本章后面，我们将会发现，如果没有可供分配的内存会发生什么事情。

我们可以为类使用任何可用的构造函数而写一个 new表达式。如果构造函数没有参数，可以写没有构造函数参数表的new表达式：

```
foo *fp = new foo ;
```

我们已经注意到了，在堆里创建对象的过程变得简单了 —— 只是一个简单的表达式,它带有内置的长度计算、类型转换和安全检查。这样在堆里创建一个对象和在栈里创建一个对象一样容易。

12.1.3 运算符delete

new表达式的反面是delete表达式。delete表达式首先调用析构函数，然后释放内存（经常是调用free( )）。正如new表达式返回一个指向对象的指针一样， delete表达式需要一个对象的地址。

```
delete fp ;
```

上面的表达式清除了早先创建的动态分配的对象 foo。

delete只用于删除由new创建的对象。如果用malloc( )（或calloc( )或realloc( )）创建一个对象，然后用delete删除它，这个行为是未定义的。因为大多数缺省的 new和delete实现机制都使

用了malloc( )和free( )，所以我们很可能会没有调用析构函数就释放了内存。

如果正在删除的对象指针是0，将不发生任何事情。为此，建议在删除指针后立即把指针赋值为0以免对它删除两次。对一个对象删除两次一定不是一件好事，这会引起问题。

12.1.4 一个简单的例子

这个例子显示了初始化发生的情况：

```
//: NEWDEL.CPP -- Simple demo of new & delete
#include <iostream.h>

class tree {
  int height;
public:
  tree(int Height) {
    height = Height;
  }
  ~tree() { cout << "*"; }
  friend ostream&
  operator<<(ostream& os, const tree* t) {
    return os << "tree height is: "
              << t->height << endl;
  }
};
main() {
  tree* T = new tree(40);
  cout << T;
  delete T;
}
```

我们通过打印tree的值以证明构造函数被调用了。这里是通过重载运算符 <<和一个ostream一起使用来实现这个运算的。注意，虽然这个函数被声明为一个友元（ friend）函数，但它还是被定义为一个内联函数。这仅仅是为了方便——定义一个友元函数为内联函数不会改变友元状态而且它仍是全局函数而不是一个类的成员函数。同时也要注意返回值是整个输出表达式的结果，它本身是一个ostream&（为了满足函数返回值类型，它必须是ostream&）。

12.1.5 内存管理的开销

当我们在堆里动态创建对象时，对象的大小和它们的生存期被正确地内置在生成的代码里，这是因为编译器知道确切的数量和范围。在堆里创建对象还包括另外的时间和空间的开销。这儿提供了一个典型的方案。（我们可以用calloc( )或realloc( )代替malloc( )）

1) 调用malloc( )，这个函数从堆里申请一块内存。

2) 从堆里搜索一块足够满足请求的内存。可以通过检查显示内存使用情况的某种图或目录来实现搜索。这个过程很快但可能要试探几次，所以它可能是不确定的——即不必指望malloc( )花费完全相同的时间。

3) 在指向这块内存的指针返回之前，这块内存大小和地址必须记录下来，这样以后调用malloc( )时就不会再使用它了，而且当我们调用free( )时，系统就会知道需要释放多大的内存。

实现这些运算的方法可能变化很大。例如，没有办法阻止在处理器里执行原始的内存分配。如果好奇的话，你可以写一个测试程序来猜测 malloc( )实现的方法；也可以读一读库函数的源代码（如果有的话）。

12.2 重新设计前面的例子

现在已经介绍了new和delete（以及其他许多主题）。对于本书前面的stash和stack例子，我们可以使用到目前为止讨论的所有的技术来重写。检查这个新代码将有助于对这些主题的复习。

12.2.1 仅从堆中创建string类

此处，类stash和stack自己都将不“拥有”它们指向的对象。即当 stash或stack出了范围，它也不会为它指向的对象调用 delete。试图使它们成为普通的类是不可能的，原因是它们是void指针。如果delete一个void指针，唯一发生的事是释放了内存，这是因为既没有类型信息也没有办法使得编译器知道要调用哪个析构函数。当一个指针从 stash或stack对象返回时，在使用它之前必须将它做类型映射。这个问题将在13章和15章讨论。

因为包容器自己不拥有指针，所以用户必须对它负责。这意味着在一个包容器上增加一个指向在栈上创建的对象的指针或增加一个指向在同一个包容器堆上创建的对象的指针时将会发生严重的问题。因为delete表达式对于不在堆上分配的指针是不安全的。（从包容器取回一个指针时，如何知道它的对象已经在哪儿分配了内存呢？）为了在如下一个简单的 String类的版本中解决这个问题，下面采取了一些步骤以防止在堆以外的地方创建 String：

```
//: STRINGS.H -- Simple string class
// Can only be built on the heap
#ifndef STRINGS_H_
#define STRINGS_H_
#include <string.h>
#include <iostream.h>

class String {
  char* s;
  String(const char* S) {
    s = new char[strlen(S) + 1];
    strcpy(s, S);
  }
  // Prevent copying:
  String(const String&);
  void operator=(String&);
public:
  // Only make Strings on the heap:
  friend String* makeString(const char* S) {
    return new String(S);
  }
  // Alternate approach:
  static String* make(const char* S) {
```

```
    return new String(S);
  }
  ~String() { delete s; }
  operator char*() const { return s;}
   char* str() const { return s; }
   friend ostream&
     operator<<(ostream& os, const String& S) {
       return os << S.s;
   }
};
#endif // STRINGS_H_
```

为了限制用户使用这个类，主构造函数声明为 private，所以，除了我们以外，没有人可以使用它。另外，拷贝构造函数也声明为 private，但没有定义，以防止任何人使用它，运算符‘ = ’也是如此。用户创建对象的唯一方法是调用一个在堆上创建 String（所以可以知道所有的String都是在堆上创建的）并返回它的指针的特殊函数。

访问这个函数的方法有两种。为了使用简单，它可以是全局的friend函数（称为makeString( )）。但如果不想“ 污染 ”全局名字空间，可以使之为 static成员函数（称为 make( )）并通过String::make( )调用它。后一种形式对于更加明显地表示它属于这个类是有好处的。

在构造函数中，注意表达式：

```
s = new char[strlen(S) +1] ;
```

方括号意味着一个对象数组被创建（此处是一个 char数组），方括号里的数字表示将创建的对象的数量。这就是在程序运行时如何创建一个数组的方法。

对char*类型的自动转换意味着在任何需要 char*的地方都可以使用一个 String对象。另外，一个iostream输出运算符扩充了 iostream库，使得它能够处理String对象。

12.2.2 stash指针

在第5章看到的类stash版本现已被修改，它反映了第 5章以来所介绍的新技术。另外，新的pstash拥有指向在堆中本来就存在的对象的指针，但在第 5章和它前面章节中，旧的stash是拷贝对象到stash包容器里的。有了新介绍的 new和delete，控制指向在堆中创建的对象的指针就变得安全、容易了。

下面提供了“ pointer stash ”的头文件：

```
//: PSTASH.H -- Holds pointers instead of objects
#ifndef PSTASH_H_
#define PSTASH_H_

class pstash {
  int quantity; // Number of storage spaces
  int next; // Next empty space
   // Pointer storage:
  void** storage;
  void inflate(int increase);
public:
```

```
  pstash() {
    quantity = 0;
    storage = 0;
    next = 0;
  }
  // No ownership:
  ~pstash() { delete storage; }
  int add(void* element);
  void* operator[](int index) const; // Fetch
  // Number of elements in stash:
  int count() const { return next; }
};
#endif // PSTASH_H_
```

基本的数据成分是非常相似的，但现在 storage是一个void指针数组，并且用 new代替malloc( )为这个数组分配内存。在下面这个表达式中，对象的类型是void*，所以这个表达式表示分配了一个void指针的数组。

```
storage = new void*[quantity = Quantity] ;
```

析构函数删除void指针本身，而不是试图删除它们所指向的内容（正如前面指出的，释放它们的内存不调用析构函数，这是因为一个void指针没有类型信息）。

其他方面的变化是用运算符[]代替了函数fetch( )，这在语句构成上显得更有意义。因为返回一个void指针，所以用户必须记住在包容器内存储的是什么类型，在取回它们时要映射这些指针（这是在以后章节将要修改的问题）。

下面是成员函数的定义：

```
//: PSTASH.CPP -- Pointer stash definitions
#include "..\11\pstash.h"
#include <iostream.h>
#include <string.h> // Mem functions

int pstash::add(void* element) {
  const InflateSize = 10;
  if(next >= quantity)
    inflate(InflateSize);
  storage[next++] = element;
  return(next - 1); // Index number
}

// Operator overloading replacement for fetch
void* pstash::operator[](int index) const {
  if(index >= next || index < 0)
    return 0;  // Out of bounds
  // Produce pointer to desired element:
  return storage[index];
}
```

```
void pstash::inflate(int increase) {
  const psz = sizeof(void*);
  // realloc() is cleaner than this:
  void** st = new void*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete storage; // Old storage
  storage = st; // Point to new memory
}
```

除了用储存指针代替整个对象的拷贝外，函数 add( )和以前一样有效。拷贝整个对象对于普通对象来说，实际上只需要拷贝构造函数。

实际上，函数 inflate( )代码比先前的版本更复杂且更低效。这是因为先前使用的函数realloc( )可以调整现有内存块的大小，也可以自动地把旧的内存块内容拷贝到更大的内存块里。在任何一件事中我们都不用为它担心，如果不用移动内存，它将进行得更快。因为 new和realloc不等价，所以在这个例子中必须分配一个更大的内存块、执行拷贝和删除旧的内存块。在这种情况下，使用malloc( )、realloc( )和free( )比new和delete在实现方面可能更有意义。幸运的是，这些实现是隐藏的，所以用户可以不用知道这些变化。只要不对同一块内存混合调用，malloc( )家族函数在和new( )及delete( )并行使用时能够保证相互之间的安全。所以这些是完全可以做到的。

• 一个测试程序

下面是为了测试pstash而将stash的测试程序重写后的程序：

```
//: PSTEST.CPP -- Test of pointer stash
#include "..\11\pstash.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  pstash intStash;
  // new works with built-in types, too:
  for(int i = 0; i < 25; i++)
    intStash.add(new int(i)); // Pseudo-constr.
  for(int u = 0; u < intStash.count(); u++)
    cout << "intStash[" << u << "] = "
         << *(int*)intStash[u] << endl;

  ifstream infile("pstest.cpp");
  allegefile(infile);
  const bufsize = 80;
  char buf[bufsize];
  pstash stringStash;
  // Use global function makeString:
  for(int j = 0; j < 10; j++)
```

```
    if(infile.getline(buf, bufsize))
      stringStash.add(makeString(buf));
  // Use static member make:
  while(infile.getline(buf, bufsize))
    stringStash.add(String::make(buf));
  // Print out the strings:
  for(int v = 0; stringStash[v]; v++) {
    char* p = *(String*)stringStash[v];
    cout << "stringStash[" << v << "] = "
         << p << endl;
  }
}
```

和前面一样，stash被创建并添加了信息。但这次的信息是由 new表达式产生的指针。注意下面这一行：

```
intStash.add( new int(i) ) ;
```

这个表达式new int(i) 使用了伪构造函数形式，所以一个新的 int对象的内存将在堆上创建并且这个int对象被初始化为值i。

注意在打印的时候，由 pstash::operator[]返回的值必须被映射成正确的类型，对于这个程序其他部分的pstash对象，也将重复这个动作。这是使用 void指针作为表达式而出现的不希望的出现效果，将在后面的章节中解决。

第2步测试打开源程序文件并把它读到另一个 psatsh里，把每一行转换为一个 String对象。我们可以看到，makeString( )和String::make( )都被使用以显示两者之间的差别。静态(static)成员函数可能是更好的方法，因为它更明显。

当取回指针时，我们可以看到如下表达式：

```
char* p = *(String*)stringStash[i] ;
```

由运算符[]返回产生的指针必须被映射为 String*以使之具有正确的类型。然后， String*被逆向引用，所以表达式可对一个对象求值。此时，当编译器想要一个 char*时，将看到一个String对象，所以它在String里调用自动类型转换运算符来产生一个char*。

在这个例子里，在堆上创建的对象永远不被销毁。这并没有害处，因为当程序结束时内存会被释放，但在实际情况下我们并不想这样，这个问题将在以后的章节里予以修正。

12.2.3 stack例子

stack例子用了许多自第4章以来介绍的技术。下面是新的头文件。

```
//: STACK11.H -- New version of stack
#ifndef STACK11_H_
#define STACK11_H_

class stack {
  struct link {
    void* data;
    link* next;
    link(void* Data, link* Next) {
```

```
      data = Data;
      next = Next;
    }
  } * head;
public:
  stack() { head = 0; }
  ~stack();
  void push(void* Data) {
    head = new link(Data,head);
  }
  void* peek() const { return head->data; }
  void* pop();
};
#endif // STACK11_H_
```

嵌套的struct link现在可以有自己的构造函数，因为在stack::push( )里， new的使用可以安全地调用那个构造函数。（注意语法很纯正，这将减小了出错的可能。）link::link构造函数简单地初始化了data和next指针，所以在stack::push( )里，下面这行不仅给新的link分配内存，而且巧妙地为link初始化：

```
head = new link(Data, head) ;
```

其余部分的逻辑实际上和第4章是一样的。下面是剩下的两个非内联函数的实现内容。

```
//: STACK11.CPP -- New version of stack
#include <stdlib.h>
#include "..\11\stack11.h"

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

stack::~stack() {
  link* cursor = head;
  while(head) {
    cursor = cursor->next;
    delete head;
    head = cursor;
  }
}
```

唯一的不同是在析构函数里使用delete代替free( )。

跟stash一样，使用void指针意味着在堆上创建的对象不能被stack销毁，所以，如果用户对

stack里的指针不加以管理的话，可能出现不希望的内存漏洞。可以在下面的测试程序里看到这些：

```
//: STKTST11.CPP -- Test new stack
#include "..\11\stack11.h"
#include "..\11\strings.h"
#include <fstream.h>
#include "..\allege.h"

main() {
  // Could also use command-line argument:
  ifstream file("stktst11.cpp");
  allegefile(file);
  const bufsize = 100;
  char buf[bufsize];
  stack textlines;
  // Read file and store lines in the stack:
  while(file.getline(buf,bufsize))
    textlines.push(String::make(buf));
  // Pop lines from the stack and print them:
  String* s;
  while((s = (String*)textlines.pop()) != 0)
    cout << *s << endl;
}
```

与stash例子一样，先打开一个文件，文件的每一行转换成为一个 String对象并存放在stack里，然后打印出来。这个程序没有删除 stack里的指针，stack本身也没有做，所以那块内存丢失了。

12.3 用于数组的new和delete

在C++里，同样容易在栈或堆上创建一个对象数组。当然，应当为数组里的每一个对象调用构造函数。有一个约束：除了在栈上整体初始化（见第 4章）外必须有一个缺省的构造函数，因为不带参数的构造函数必须被每一个对象调用。

当使用new在堆上创建对象数组时，还必须做一些事情。下面有一个对象数组的例子：

```
foo* fp = new foo[100] ;
```

这样在堆上为100个foo对象分配了足够的内存并为每一个对象调用了构造函数。现在，我们拥有一个foo*。它和用如下表达式创建单个对象得到的结果是一样的：

```
foo* fp2 = new foo ;
```

因为这是我们自己写的代码，并且我们知道 fp实际上是一个数组的起始地址，所以用 fp[2]选择数组的元素是有意义的。销毁这个数组时发生了什么呢？下面的语句看起来是完全一样的：

```
delete fp2 ; //OK
delete fp ;  // Not the desired effect
```

并且效果也会一样：为所给地址指向的 foo对象调用析构函数，然后释放内存。对于 fp2，这样是正确的，但对于fp，另外99个析构函数调用没有进行。正确的存储单元数量还会被释放，因

为它被分配在一个大块的内存里，整个内存块的大小被分配程序藏在某处。

解决办法是需要给编译器一个数组起始地址的信息。这可以用下面的语法来实现：

```
delete []fp ;
```

空的方括号告诉编译器产生从数组创建时存放的地方取回数组中对象数量的代码，并为数组的所有对象调用析构函数。这实际上是早期语法的改良形式，偶尔仍可以在老的代码里看到如下的代码：

```
delete [100]fp ;
```

这个语法强迫程序设计者在数组里包含对象的数量，程序设计者有可能把对象数量弄错。而让编译器处理这件事的附加代价是很低的，所以只在一个地方指明对象数量要比在两个地方指明好些。

使指针更像数组

作为题外话，上面定义的fp可以被修改指向任何类型，但这对于一个数组的起始地址来讲没有什么意义。一般讲来，把它定义为常量更有意义些，因为这样任何修改指针的企图都会被指出出错。为了得到这个效果，我们可以试着用下面的表达式：

```
int const* q = new int[10] ;
```

或

```
const int* q = new int[10] ;
```

但在上面这两种情况里，const将和int 捆绑在一起，限定指针指向的内容而不是指针本身。必须用下面的表达式代替：

```
int* const q = new int[10] ;
```

现在在q里的数组元素可以被修改，但对q本身的修改（例如q++）是不合法的，因为它是一个普通数组标识符。

12.4 用完内存

当运算符new找不到足够大的连续内存块来安排对象时将会发生什么？一个称为 new-handler的函数被调用。或者，检查指向函数的指针，如果指针非0，那么它指向的函数被调用。

对于new-handler的缺省动作是抛出一个异常(throw an exception)，这个主题在第17章介绍。然而，如果我们在程序里用堆分配，至少要用“内存已用完”的信息代替new-handler，并异常中断程序。用这个办法，在调试程序时会得到程序出错的线索。对于最终的程序，我们总想使之具有很强的容错性。

通过包含NEW.H，然后以我们想装入的函数地址为参数调用set_new_handler( )函数，这样就替换了new-handler。

```
//: NEWHANDL.CPP -- Changing the new-handler
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

void out_of_memory() {
  cerr << "memory exhausted!" << endl;
  exit(1);
```

```
}

main() {
  set_new_handler(out_of_memory);
  while(1)
    new int[1000]; // Exhausts memory
}
```

new-handler 函数必须不带参数且具有void返回值。while循环将持续分配int对象（并丢掉它们的返回地址）直到全部内存被耗尽。在紧接下去对 new调用时，将没有内存可被分配，所以调用new-handler。

当然，可以写更圆满的new-handler，甚至它可以收回内存（通常叫做垃圾回收）。这不是编程新手的工作。

12.5 重载new和delete

当创建一个new表达式时有两件事发生。首先，使用运算符 new分配内存，然后调用构造函数。在delete表达式里，调用析构函数，然后使用运算符 delete释放内存。我们永远无法控制构造函数和析构函数的调用（否则我们可能意外地搅乱它们），但可以改变内存分配函数运算符new和delete。

被new和delete使用的内存分配系统是为通用目的而设计的。但在特殊的情形下，它不能满足我们的需要。改变分配系统的原因是考虑效率：我们也许要创建和销毁一个特定的类的非常多的对象以至于这个运算变成了速度的瓶颈。 C++允许重载new和delete来实现我们自己的存储分配方案，所以可以像这样处理问题。

另外一个问题是堆碎片：分配不同大小的内存可能造成在堆上产生很多碎片，以至于很快用完内存。也就是内存可能还有，但由于是碎片，找不到足够大的内存满足我们的需要。通过为特定类创建我们自己的内存分配器，可以确保这种情况不会发生。

在嵌入和实时系统里，程序可能必须在有限的资源情况下运行很长时间。这样的系统也可能要求分配内存花费相同的时间且不允许出现堆内存耗尽或出现很多碎片的情况。由客户定制的内存分配器是一种解决办法，否则程序设计者在这种情况下要避免使用 new和delete，从而失去了C++很有价值的优点。

当重载运算符new和delete时，记住只改变原有的内存分配方法是很重要的。编译器将用new代替缺省的版本去分配内存，然后为那个内存调用构造函数。所以，虽然编译器遇到 new时会分配内存并调用构造函数，但当我们重载 new时，可以改变的只是内存分配部分。（delete也有相似的限制。）

当重载运算符new时，也可以替换它用完内存时的行为，所以必须在运算符 new里决定做什么：返回0、写一个调用new-handler的循环、再试着分配或用一个 bad_alloc异常处理（在第17章中讨论）。

重载new和delete与重载任何其他运算符一样。但可以选择重载全局内存分配函数，或为特定的类使用特定的分配函数。

12.5.1 重载全局new和delete

当全局版本的new和delete不能满足整个系统时，对其重载是很极端的方法。如果重载全局

版本，那么缺省版本就完全不能被访问——甚至在这个重载定义里也不能调用它们。

重载的new必须有一个size_t参数。这个参数由编译器产生并传递给我们，它是要分配内存的对象的长度。必须返回一个指向等于这个长度（或大于这个长度，如果我们有这样做的原因）的对象的指针，或如果没有找到存储单元（在这种情况下，构造函数不被调用），返回一个0。然而如果找不到存储单元，不能仅仅返回0，还应该调用new-handler或进行异常处理，通知这里存在问题。

运算符new的返回值是一个void*，而不是指向任何特定类型的指针。它所做的是分配内存，而不是完成一个对象的建立——直到构造函数调用了才完成对象的创建，这是由编译器所确保的动作，不在我们的控制范围内。

运算符delete接受一个指向由运算符new分配的内存的void*。它是一个void*因为它是在调用析构函数后得到的指针。析构函数从存储单元里移去对象。运算符 delete的返回类型是void。

下面提供了一个如何重载全局new和delete的简单的例子：

```
//: GLOBLNEW.CPP -- Overload global new/delete
#include <stdio.h>
#include <stdlib.h>

void* operator new(size_t sz) {
  printf("operator new: %d bytes\n", sz);
  void* m = malloc(sz);
  if(!m) puts("out of memory");
  return m;
}

void operator delete(void* m) {
  puts("operator delete");
  free(m);
}

class s {
  int i[100];
public:
  s() { puts("s::s()"); }
  ~s() { puts("s::~s()"); }
};

main() {
  puts("creating & destroying an int");
  int* p = new int(47);
  delete p;
  puts("creating & destroying an s");
  s* S = new s;
  delete S;
  puts("creating & destroying s[3]");
  s* SA = new s[3];
  delete []SA;
}
```

这里可以看到重载new和delete的一般形式。为了实现内存分配器，使用了标准 C库函数malloc( )和free( )（可能缺省的new和delete也使用这些函数）。它们还打印出了它们正在做什么的信息。注意，这里使用printf( )和puts( )而不是iostreams。当创建了一个iostream对象时（像全局的cin、cout和cerr），它们调用new去分配内存。用printf( )不会进入死锁状态，因为它不调用new来初始化本身。

在main( )里，创建内部数据类型的对象以证明在这种情况下重载的 new和delete也被调用。然后创建一个类型s的单个对象，接着创建一个数组。对于数组，我们可以看到需要额外的内存用于存放数组对象数量的信息。在所有情况里，都是使用全局重载版本的 new和delete。

12.5.2 为一个类重载new和delete

为一个类重载new和delete时，不必明说是static，我们仍是在创建static成员函数。它的语法也和重载任何其他运算符一样。当编译器看到使用 new创建类对象时，它选择成员版本运算符new而不是全局版本的new。但全局版本的new和delete为所有其他类型对象使用（除非它们有自己的new和delete）。

在下面的例子里我们为类framis创建了一个非常简单的内存分配系统。程序开始时在静态数据区域内留出一块存储单元。这块内存是用于 framis类型对象分配的内存空间。为了决定哪块存储单元已被使用，这里使用了一个字节(bytes)数组，一个字节(byte)代表一块存储单元。

```
//: FRAMIS.CPP -- Local overloaded new & delete
#include <stddef.h> // Size_t
#include <fstream.h>
ofstream out("framis.out");

class framis {
  char c[10];
  static unsigned char pool[];
  static unsigned char alloc_map[]; // Alloc map
public:
  enum { psize = 100 };  // # of framis allowed
  framis() { out << "framis()\n"; }
  ~framis() { out << "~framis() ... "; }
  void* operator new(size_t);
  void operator delete(void*);
};
unsigned char framis::pool[psize * sizeof(framis)];
unsigned char framis::alloc_map[psize] = {0};

// Size is ignored -- assume a framis object
void* framis::operator new(size_t) {
  for(int i = 0; i < psize; i++)
    if(!alloc_map[i]) {
      out << "using block " << i << " ... ";
      alloc_map[i] = 1; // Mark it used
      return pool + (i * sizeof(framis));
```

```
    }
  out << "out of memory" << endl;
  return 0;
}

void framis::operator delete(void* m) {
  if(!m) return; // Check for null pointer
  // Assume it was created in the pool
  // Calculate which block number it is:
  unsigned long block = (unsigned long)m
    - (unsigned long)pool;
  block /= sizeof(framis);
  out << "freeing block " << block << endl;
  // Mark it free:
  alloc_map[block] = 0;
}

main() {
  framis* f[framis::psize];
  for(int i = 0; i < framis::psize; i++)
    f[i] = new framis;
  new framis; // Out of memory
  delete f[10];
  f[10] = 0;
  // Use released memory:
  framis* x = new framis;
  delete x;
  for(int j = 0; j < framis::psize; j++)
    delete f[j]; // Delete f[10] OK
}
```

通过分配一个足够大的数组来保存 psize 个 framis对象的方法来为framis堆创建一块内存。这个内存分配图是psize个字节，所以每一块内存对应一个字节。使用设置首元素为 0的集合初始化技巧，编译器能够自动地初始化其余的元素，来使内存分配图里的所有字节都初始化为 0。

局部运算符new和全局运算符new具有相同的形式。它所做的是通过对内存分配图进行搜索来找到为0的字节，然后设置该字节为1，以此声明这块存储单元已经被分配并返回这个存储单元的地址。如果它找不到内存，将发送一个消息并返回 0（注意new-handler没有被调用，也没报告异常，这是因为当我们用完了内存时，行为在我们的控制之下）。因为没有涉及全局运算符new和delete，所以在这个例子里使用iostreams是可行的。

运算符delete假设framis的地址是在这个堆里创建的。这是一个公平的假设，因为无论何时在堆上创建单个framis对象 —— 不是一个数组，都将调用局部运算符new。全局版本的new在数组情况下使用。所以用户偶尔会在没有用空方括号语法来声明数组被消除的情况下调用运算符delete，这可能会引起问题。因为用户也可能删除在栈上创建的指向对象的指针。如果我们不想这样的事情发生，应该加入一行代码以确保地址是在这个堆内并是在正确的地址范围内。

运算符delete计算这个指针代表这个堆里的哪一块内存，然后将代表这块内存的分配图的标志设置为0来声明这块内存已经被释放。

在main( )里，动态地分配了很多framis对象以使内存用完，这样就能检查地址用完时的行为。然后释放一个对象，再创建一个对象来显示那个释放了的内存被重新使用了。

因为这个内存分配方案是针对framis对象的，所以可能比用缺省的new和delete针对一般目的内存分配方案效率要高一些。

12.5.3 为数组重载new和delete

如果为一个类重载了运算符new和delete，那么无论何时创建这个类的一个对象都将调用这些运算符。但如果为这些对象创建一个数组时，将调用全局运算符 new( )立即为这个数组分配足够的内存。全局运算符 delete( )被调用来释放这块内存。可以通过为那个类重载数组版本的运算符new[]和delete[]来控制对象数组的内存分配。这里提供了一个显示两个不同版本被调用的例子：

```
//: NEWARRY.CPP -- Operator new for arrays
#include <new.h> // Size_t definition
#include <fstream.h>
ofstream trace("newarry.out");

class widget {
  int i[10];
public:
  widget() { trace << "*"; }
  ~widget() { trace << "~"; }
  void* operator new(size_t sz) {
    trace << "widget::new: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete(void* p) {
    trace << "widget::delete" << endl;
    ::delete []p;
  }
  void* operator new[](size_t sz) {
    trace << "widget::new[]: "
          << sz << " bytes" << endl;
    return ::new char[sz];
  }
  void operator delete[](void* p) {
    trace << "widget::delete[]" << endl;
    ::delete []p;
  }
};

main() {
```

```
    trace << "new widget" << endl;
    widget* w = new widget;
    trace << "\ndelete widget" << endl;
    delete w;
    trace << "\nnew widget[25]" << endl;
    widget* wa = new widget[25];
    trace << "\ndelete []widget" << endl;
    delete []wa;
};
```

这里，全局版本的new和delete被调用，除了加入了跟踪信息以外，它们和未重载版本 new和delete的效果是一样的。当然，我们可以在重载的new和delete里使用想要的内存分配方案。

可以看到除了加了一个括号外，数组版本的new和delete与单个对象版本是一样的。在这两种情况下，要传递分配的对象内存大小。传递给数组版本的内存大小是整个数组的大小。应该记住重载运算符new唯一需要做的是返回指向一个足够大的内存的指针。虽然我们可以初始化那块内存，但通常这是构造函数的工作，构造函数将被编译器自动调用。

这里构造函数和析构函数只是打印出字符，所以我们可以看到它们已被调用。下面是这个跟踪文件的输出信息：

```
new widget
widget::new: 20 bytes
*
delete widget
~widget::delete

new widget[25]
widget::new[]: 504 bytes
*************************
delete []widget
~~~~~~~~~~~~~~~~~~~~~~~~~widget::delete[]
```

和预计的一样（这个机器为一个 int使用2个字节），创建一个对象需要 20个字节。运算符new被调用，然后是构造函数（由 *指出）。以一个相反的形式调用 delete使得析构函数被调用，然后是delete。

当创建一个widget对象数组时，可使用数组版本new。但注意，申请的长度比期望的大4个字节。这额外的4个字节是系统用来存放数组信息的，特别是数组中对象的数量。当用下面的表达式时，方括号就告诉编译器它是一个对象数组，所以编译器产生寻找数组中对象的数量的代码，然后多次调用析构函数。

```
delete []widget ;
```

我们可以看到，即使数组运算符new和delete只为整个数组调用一次，但对于数组中的每一个对象，缺省的构造函数和析构函数都被调用。

12.5.4 构造函数调用

```
foo* f = new foo ;
```

上面的表达式调用new分配一个foo长度大小的内存，然后在那个内存上调用 foo构造函数。

假设所有的安全措施都失败了并且运算符 new返回值是0，将会发生什么？在上述情况下，构造函数没有调用，所以虽然没有一个成功创建的对象，但至少我们也没有调用构造函数和传递给它一个0指针。下面是一个证明这一点的例子：

```
//: NOMEMORY.CPP -- Constructor isn't called
// If new returns 0
#include <iostream.h>
#include <new.h> // size_t definition
void my_new_handler() {
  cout << "new handler called" << endl;
}

class nomemory {
public:
  nomemory() {
    cout << "nomemory::nomemory()" << endl;
  }
  void* operator new(size_t sz) {
    cout << "nomemory::operator new" << endl;
    return 0; // "Out of memory"
  }
};

main() {
  set_new_handler(my_new_handler);
  nomemory* nm = new nomemory;
  cout << "nm = " << nm << endl;
}
```

当程序运行时，它仅打印来自运算符 new的信息。因为new返回0，所以构造函数没有被调用，当然它的信息不会打印出来。

12.5.5 对象放置

重载运算符new还有其他两个不常见的用途。

1) 可能想要在内存的指定位置上放置一个对象。这对于面向硬件的内嵌系统特别重要，在这个系统中，一个对象可能和一个特定的硬件是同义的。

2) 可能想在调用new时，可以从不同的内存分配器中选择。

这两种情形都用相同的机制解决：重载的运算符 new可以带多于一个的参数。像前面所看到的，第一个参数总是对象的长度，它是在内部计算出来的并由编译器传递。但其他参数可以由我们自己定义：一个放置对象的地址、一个对内存分配函数或对象的引用、或其他方便的任何设置。

起先在调用过程中传递额外的参数给运算符 new的方法看起来似乎有点古怪：在关键字new后是参数表（没有size_t参数，它由编译器处理），参数表后面是正在创建的对象的类名字。例如：

```
X* xp = new(a) X ;
```

将传递a作为第二个参数给运算符new。当然，这是在这个运算符new已经声明的情况下才有效的。

下面的例子显示了如何在一个特定的存储单元里放置一个对象。

```
//: PLACEMNT.CPP -- Placement with operator new
#include <stddef.h> // Size_t
#include <iostream.h>

class X {
  int i;
public:
  X(int I = 0) { i = I; }
  ~X() {
    cout << "X::~X()" << endl;
  }
  void* operator new(size_t, void* loc) {
    return loc;
  }
};
main() {
  int l[10];
  X* xp = new(l) X(47); // X at location l
  xp->X::~X(); // Explicit destructor call
  // ONLY use with placement!
}
```

注意运算符new仅返回传递给它的指针。因此，应该由调用者决定对象存放在哪里，决定作为new表达式的一部分的构造函数将在哪块内存上被调用。

销毁对象时将会出现两难选择的局面。因为仅有一个版本运算符 delete，所以没有办法说“对这个对象使用我的特殊内存释放器”。我们可以调用析构函数，但不能用动态内存机制释放内存，因为内存不是在堆上分配的。

答案是用非常特殊的语法：可以显式地调用析构函数。例如：

```
xp->X::~X( ) ; //explicit destructor call
```

这里会出现一个严厉的警告。一些人把这看作是在范围结束前的某一时刻销毁对象的一种方法，而不是调整范围，或想要这个对象的生命期在运行时确定时，更正确地使用动态对象创建的方法。如果用这种方法为在栈上创建的对象调用析构函数，将会出现严重的问题，因为析构函数在出范围时又会被调用一次。如果为在堆上创建的对象用这种方法调用析构函数，析构函数将被执行，但内存不释放，这可能是我们不希望的结果。用这种方法显式地调用析构函数，其实只有一个原因，即为运算符new 支持存放语法。

虽然这个例子仅显示一个附加的参数，但如果为了其他目的而增加更多的参数，也是可行的。

12.6 小结

在栈上创建自动对象既方便又理想，但为了解决一般程序问题，我们必须在程序执行的任

何时候，特别是需要对来自程序外部信息反应时，能够创建和销毁对象。虽然 C的动态内存分配可以从堆上得到内存，但它在C++上不易使用且不能够保证安全。使用new和delete进行动态对象创建，这已经成为C++语言的核心，所以可以像在栈上创建对象一样容易地在堆上创建对象，另外，还可以有很大的灵活性。如果new和delete不能满足要求，尤其是它们没有足够的效率时，程序员还可以改变它们的行为，也可以在内存用完时进行修改。（第17章讨论的异常处理也在这里使用了）

12.7 练习

1. 写一个有构造函数和析构函数的类。在构造函数和析构函数里通过 cout宣布自己。通过这个类自己证明new和delete总是调用构造函数和析构函数。用 new创建这个类的一个对象，用delete销毁它。在堆上创建和销毁这些对象的一个数组。

2. 创建一个pstash对象，并把练习1的对象用new创建填入。观察当这个对象出了范围和它的析构函数被调用时发生的情况。

3. 写一个有单个对象版本和数组版本的重载运算符 new和delete的类。演示这两个版本的工作情况。

4. 设计一个对FRAMIS.CPP进行测试的程序来显示定制的new和delete比全局的new和delete大约快多少。