第12章 类和对象

在C++中，类构成了实现C++面向对象程序设计的基础。我们使用类来定义对象的属性，类是C++封装的基本单元。本章将详细讨论类及对象。

12.1 类

我们使用关键字class创建类。一个类声明定义了一个链接代码和数据的新类型，这个新类型又可以用来声明该类的对象。因此，类是逻辑抽象的概念，而对象是物理存在的。也就是说，对象是类的实例。

类声明在语法上与结构相似。在第11章中，给出了一个简化的类声明的一般格式。这里我们给出一个完整的类声明的一般格式，它不继承任何别的类。

```
class class-name {
  private data and functions
access-specifier:
  data and functions
access-specifier:
  data and functions
// ...
access-specifier:
  data and functions
} object-list;
```

其中，object-list是任选项。如果存在，它声明类的对象。access-specifier为下面三个C++关键字之一：

```
public
private
protected
```

默认时，在类中声明的函数和数据属该类私有，只能被该类的成员访问。然而，如果用public访问限定符，函数或数据就可以被程序的其他部分访问了。protected访问限定符仅在涉及继承时才需要（参见第15章），访问限定符一经使用，其作用就保持到遇到别的访问限定符或到达类声明的结束处时。

在类声明内可以任意改变访问说明。例如，对某些声明，可以转换为public，然后再转换回private。下面这个例子中的类声明演示了这一特性：

```
#include <iostream>
#include <cstring>
using namespace std;

class employee {
  char name[80]; // private by default
```

```
public:
  void putname(char *n); // these are public
  void getname(char *n);
private:
  double wage; // now, private again
public:
  void putwage(double w); // back to public
  double getwage( );
};

void employee::putname(char *n)
{
  strcpy(name, n);
}

void employee::getname(char *n)
{
  strcpy(n, name);
}

void employee::putwage(double w)
{
  wage = w;
}

double employee::getwage( )
{
  return wage;
}

int main( )
{
  employee ted;
  char name[80];

  ted.putname("Ted Jones");
  ted.putwage(75000);
  ted.getname(name);
  cout << name << " makes $";
  cout << ted.getwage( ) << " per year.";

  return 0;
}
```

其中，employee是一个简单的类，用来存储雇员的姓名和薪水。注意，public访问限定符使用了两次。

尽管可以在类内频繁地使用访问限定符，这样做的惟一优点是：通过把一个类的各个部分可视地组在一起，可以使其他阅读程序的人更好地理解程序。然而，对编译器来说，使用多重访问限定符无异于使用单个访问限定符。实际上，大多数程序员会发现在每个类内仅有一个private, protected和public节会更容易些。例如，大多数C++程序员往往会像下面的示例一样给employee类编码，即把归类在一起的所有private元素和public元素放在一起：

```
class employee {
```

```
  char name[80];
  double wage;
public:
  void putname(char *n);
  void getname(char *n);
  void putwage(double w);
  double getwage( );
};
```

在类中声明的函数称为成员函数。成员函数可以访问类所属的任何元素，其中包括所有的私有元素。作为类的元素的变量称为成员变量或数据成员（也称为实例变量）。总之，类的任何元素都可被当作该类的成员。

对于类成员也存在一些限制。非静态成员变量没有初值，成员不能成为当前正被声明的类的对象（虽然一个成员可以是指向当前正被声明的类的指针）。成员不能声明为 auto, extern 或 register。

通常，可以使一个类的所有数据成员为该类私有，这是实现封装的方法的一部分。然而，有时也需要使一个或几个变量成为公有的（例如，为了获得较快的执行速度，频繁使用的变量就需要是全局可访问的）。一旦一个变量成为公有的，在用户程序的所有地方都可以访问这个变量。访问公有数据成员的语法与调用成员函数的语法一样：指定对象的名称、点运算符和变量名称。下面这个简单的程序演示了一个公有变量的使用方法。

```
#include <iostream>
using namespace std;

class myclass {
public:
  int i, j, k; // accessible to entire program
};

int main( )
{
  myclass a, b;

  a.i = 100; // access to i, j, and k is OK
  a.j = 4;
  a.k = a.i * a.j;

  b.k = 12; // remember, a.k and b.k are different
  cout << a.k << " " << b.k;

  return 0;
}
```

12.2 结构和类是相互关联的

结构是C子集的一部分，是从C语言中继承来的。如你所见，类在语法上类似于结构，但是类和结构之间的关系比你最初想的要更接近。在C++中，结构的作用被扩展了，使它成为指定一个类的替代方法。实际上类和结构的惟一区别就在于：默认时，结构的所有成员是公有的，而类的所有成员是私有的。除此之外，结构和类是等价的，也就是说，一个结构定义了一个类的类型。例如，考虑下面这个程序，它用一个结构来声明一个控制访问一个字符串的类。

```
// Using a structure to define a class.
#include <iostream>
#include <cstring>
using namespace std;

struct mystr {
  void buildstr(char *s); // public
  void showstr( );
private: // now go private
  char str[255];
} ;

void mystr::buildstr(char *s)
{
  if(!*s) *str = '\0'; // initialize string
  else strcat(str, s);
}

void mystr::showstr( )
{
  cout << str << "\n";
}

int main( )
{
  mystr s;

  s.buildstr(""); // init
  s.buildstr("Hello ");
  s.buildstr("there!");

  s.showstr( );

  return 0;
}
```

该程序显示字符串“Hello there!”。类 mystr 可以用下面的类改写：

```
class mystr {
  char str[255];
public:
  void buildstr(char *s); // public
  void showstr( );
} ;
```

读者也许还不清楚为什么C++包含两个实际上等价的关键字 struct 和 class。下面几点说明这个冗余是合理的。第一，最基本的原因就是增强结构的能力。在C语言中，结构提供了数据分组的方法，因此，使结构包含成员函数是一个小小的进步。第二，由于结构和类是相互关联的，所以把现成的C程序移植到C++程序就很容易了。第三，虽然结构和类现在实际上是等价的，提供两个不同的关键字可使类定义自由的演化。为了保持C++对C的兼容性，结构定义必须总是受限于它的C语言定义。

即使在某些地方可以用结构（struct）来代替类（class），但一般不要这样做。为了清楚起见，要用类的地方就用 class，要用类C结构的地方就用 struct。本书从现在开始就保持这种风

格。有时我们使用缩略语POD来描述C风格的结构——不包含成员函数、构造函数或析构函数的结构。POD代表Plain Old Data，即普通老数据。

记住：在C++中，结构声明定义了一个类类型。

12.3 联合和类是相互关联的

和结构一样，联合（union）也可以用来定义类。在C++中，联合既包含成员函数，也包含变量，还可以包含构造函数和析构函数。C++的联合保留了类C联合的所有特性，其中最重要的就是所有数据元素共享内存的相同地址。与结构相似，默认时，联合成员都是公有的，并且完全与C兼容。下面这个例子使用联合来交换组成一个无符号短整数的字节（本例假设短整数为2字节长）。

```
#include <iostream>
using namespace std;

union swap_byte {
  void swap( );
  void set_byte(unsigned short i);
  void show_word( );

  unsigned short u;
  unsigned char c[ 2] ;
};

void swap_byte::swap( )
{
  unsigned char t;

  t = c[ 0] ;
  c[ 0]  = c[ 1] ;
  c[ 1]  = t;
}

void swap_byte::show_word( )
{
  cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
  u = i;
}

int main( )
{
  swap_byte b;

  b.set_byte(49034);
  b.swap( );
  b.show_word( );

  return 0;
}
```

与结构一样，C++中的联合声明定义了一种特殊的类。这意味着保持了封装原则。

使用C++的联合时，必须遵守几个限制条件。第一，联合不能继承任何其他类型的类。第二，联合不能是基类，不能含有虚成员函数（虚函数在第17章中讨论）。静态变量不能是联合的成员；不能使用引用成员。联合不能有任何作为成员的重载“=”运算符的对象。最后，如果一个对象有明确的构造函数或析构函数，那么它就不能成为联合的成员。

与结构一样，术语POD一般也适用于不包含成员函数、构造函数或析构函数的联合。

12.3.1 匿名联合

C++有一种特殊的联合，称为匿名联合（anonymous union）。匿名联合没有类型名称，也不声明这类联合的任何变量。但是，匿名联合告诉编译器它的成员变量共享同一内存地址。然而，变量本身可被直接引用，无需常规的点运算符语法。例如，考虑下面的程序：

```
#include <iostream>
#include <cstring>
using namespace std;

int main( )
{
  // define anonymous union
  union {
    long l;
    double d;
    char s[ 4] ;
  } ;

  // now, reference union elements directly
  l = 100000;
  cout << l << " ";
  d = 123.2342;
  cout << d << " ";
  strcpy(s, "hi");
  cout << s;

  return 0;
}
```

可以看到，上例中对联合元素的引用和一般的局部变量没什么两样。事实上，和程序有关，这完全取决于如何使用它们。此外，即使它们是在联合声明中定义的，其作用域与其他任何位于同一块的局部变量是相同的。这意味着，匿名联合成员的名称不应与该联合域中的其他标识符发生冲突。

除了以下限制外，涉及联合的所有限制也适用于匿名联合。第一，匿名联合包含的元素只能是数据，不允许包含成员函数，同时也不能包含私有或保护元素。第二，全局匿名联合必须被声明为静态的。

12.4 友元函数

准许非成员函数通过使用友元（friend）访问类的私有成员是可能的。友元函数可以访问类的所有私有成员和保护成员。要在类中声明一个友元函数，在类中包含它的原型，只需在其之

前放上关键字 friend。考虑下面的程序：

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  friend int sum(myclass x);
  void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
  a = i;
  b = j;
}

// Note: sum( ) is not a member function of any class.
int sum(myclass x)
{
  /* Because sum( ) is a friend of myclass, it can
     directly access a and b. */

  return x.a + x.b;
}

int main( )
{
  myclass n;

  n.set_ab(3, 4);

  cout << sum(n);

  return 0;
}
```

在这个例子中，函数 sum( )不是 myclass 的成员，但它仍然可以访问 myclass 的私有成员。还有，注意调用 sum( )时没有使用点运算符。由于 sum( )不是成员函数，所以它不必用对象名来限制（实际上也不可能）。

虽然暂时看不出把 sum( )定义为 myclass 的一个友元而非成员函数有什么好处，但在某些情况下，友元函数具有很大的价值。第一，友元对重载某些运算符很有好处（参见第 14 章）。第二，友元函数使得构造某些类型的 I/O 函数更加容易（参见第 18 章）。第三，在某些情况下，当两个或两个以上的类包含与程序其他部分有关的相互关联的成员时，友元函数即为所求之物。下面来看看第三种用法。

让我们假设有两个不同的类，当出现错误时，每个类要在屏幕上显示弹出式信息。为了使信息不被意外地重写，程序的其他部分希望了解：在写到屏幕之前，弹出式信息当前是否正在显示。虽然可以在每个类中创建成员函数，而这些成员函数返回标识信息是否处于激活状态的值，但这意味着在检测该条件时会有额外的开销（即两次函数调用，而不是一次）。如果需要频繁地检测这个条件，这个附加的开销可能是不可接受的。然而，使用一个是每个类的友元的函数，就可以通过仅调用这一个函数来检测每个对象的状态。因此，在这种情况下，使用友元

函数能产生更有效的代码。下面的程序演示了这个概念：

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2; // forward declaration

class C1 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

class C2 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int idle(C1 a, C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

int idle(C1 a, C2 b)
{
  if(a.status || b.status) return 0;
  else return 1;
}

int main( )
{
  C1 x;
  C2 y;

  x.set_status(IDLE);
  y.set_status(IDLE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  x.set_status(INUSE);

  if(idle(x, y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
```

```
  return 0;
}
```

注意，对于类C2，该程序使用了一个超前声明（也称为超前引用），这是必要的。因为在声明C2之前，在C1中声明idle( )时引用了C2。要创建一个类的超前声明，可使用本程序所示的方法。

一个类的友元可以是另一个类的成员，例如，改写前面的程序，使idle( )成为C1的一个成员：

```
#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;

class C2; // forward declaration

class C1 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  int idle(C2 b); // now a member of C1
};

class C2 {
  int status; // IDLE if off, INUSE if on screen
  // ...
public:
  void set_status(int state);
  friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
  status = state;
}

void C2::set_status(int state)
{
  status = state;
}

// idle( ) is member of C1, but friend of C2
int C1::idle(C2 b)
{
  if(status || b.status) return 0;
  else return 1;
}

int main( )
{
  C1 x;
  C2 y;
```

```
  x.set_status(IDLE);
  y.set_status(IDLE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";
  x.set_status(INUSE);

  if(x.idle(y)) cout << "Screen can be used.\n";
  else cout << "In use.\n";

  return 0;
}
```

由于idle( )是C1的成员，所以它可以直接访问类型C1的对象的状态变量。因此，只有类型C2的对象要传给idel( )。

使用友元函数有两个重要的限制：第一，派生类不继承友元函数。第二，友元函数不能有存储类标识符，即，它们不能被定义为静态的（static）或外部的（extern）。

12.5 友元类

一个类可以是另一个类的友元。此时，友元类和它的所有成员函数可以访问其他类中定义的私有成员。例如：

```
// Using a friend class.
#include <iostream>
using namespace std;

class TwoValues {
  int a;
  int b;
public:
  TwoValues(int i, int j) { a = i; b = j; }
  friend class Min;
};

class Min {
public:
  int min(TwoValues x);
};

int Min::min(TwoValues x)
{
  return x.a < x.b ? x.a : x.b;
}

int main( )
{
  TwoValues ob(10, 20);
  Min m;

  cout << m.min(ob);

  return 0;
}
```

其中，类Min访问了TwoValues类中定义的私有变量a和b。

理解这一点很重要，即当一个类是另一个类的友元时，它只能访问在另一个类中定义的名字，但不能继承该类，特别地，第一个类的成员不能成为友元类的成员。

友元类较少使用。支持友元类是为了允许处理一些特殊的情况。

12.6 内联函数

内联函数是C++中的一个重要特性，通常用在类中。由于本章经常用到它（甚至本书后面的部分也经常用到），因此，这里讨论一下内联函数。

在C++中，用户可以创建实际上不调用的短函数，它们的代码在每次调用的程序行里得到扩展。这个过程类似于使用类函数的宏。为了使一个函数在程序行里进行代码扩展而不被调用，只要在函数前面加上关键字inline即可。例如，在下面的程序里，函数max()在行内扩展而不被调用：

```
#include <iostream>
using namespace std;
inline int max(int a, int b)
{
  return a>b ? a : b;
}

int main( )
{
  cout << max(10, 20);
  cout << " " << max(99, 88);

  return 0;
}
```

就编译器所关心的，上面的程序等价于下面的程序：

```
#include <iostream>
using namespace std;

int main( )
{

  cout << (10>20 ? 10 : 20);
  cout << " " << (99>88 ? 99 : 88);

  return 0;
}
```

内联函数是C++的一个重要补充的原因是，它们能使程序员写出非常有效的代码。因为类一般要求几个经常被执行的接口函数（提供对私有数据的访问），因此，这些函数的效率在C++中是非常重要的。我们知道，每次调用函数时，调用和返回机制会产生数量可观的开销。典型的情况是，当调用一个函数时，变元要进栈，各种寄存器内容要保存；函数返回时，又要恢复它们的内容。问题是这些指令要占用时间。但是，如果函数在行内扩展，上述那些操作就不存在了。当然，虽然函数行内扩展能产生较快的执行速度，但由于重复编码会产生较长的代码，因此，最好只内联那些能明显影响程序性能的函数。

像register限定符一样，inline对编译器来说是一种请求，而不是命令。编译器可以选择忽略它。还有，一些编译器不能内联所有类型的函数。例如，通常编译器不能内联递归函数。必须查阅自己的编译器用户手册以了解对内联的限制。记住，如果一个函数不能内联，它就被当作一个正常函数调用。

内联函数可以是类的成员函数。例如，下面是一个完全有效的C++程序：

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  void init(int i, int j);
  void show( );
};

// Create an inline function.
inline void myclass::init(int i, int j)
{
  a = i;
  b = j;
}

// Create another inline function.
inline void myclass::show( )
{
  cout << a << " " << b << "\n";
}

int main( )
{
  myclass x;

  x.init(10, 20);
  x.show( );

  return 0;
}
```

注意： inline关键字不是C++的C子集的一部分，因此，C89没有定义它，然而，C99中增加了它。

12.7 在类中定义内联函数

在类声明内定义短函数是可能的。如果一个函数是在类声明内定义的，它将被自动地转换成内联函数（如果可能的话）。没有必要（但不是错误）在函数声明的前面加上关键字inline。例如，改写上面的程序，使init( )和show( )的定义包含在myclass的声明里：

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
```

```
public:
  // automatic inline
  void init(int i, int j) { a=i; b=j; }
  void show( ) { cout << a << " " << b << "\n"; }
};

int main( )
{
  myclass x;

  x.init(10, 20);
  x.show( );

  return 0;
}
```

注意myclass中函数代码的格式。由于内联函数非常短，这种编码风格具有代表性。但是，用户可以随意采用喜欢的格式。例如，下面是一个改写myclass声明的非常有效的方法：

```
#include <iostream>
using namespace std;

class myclass {
  int a, b;
public:
  // automatic inline
  void init(int i, int j)
  {
    a = i;
    b = j;
  }

  void show( )
  {
    cout << a << " " << b << "\n";
  }
};
```

从技术上来讲，由于I/O语句一般比函数调用的开销要大得多，所以，这里内联函数show()的意义不大。但是，在C++程序中，我们经常看到所有短成员函数都在它们的类里定义（实际上，在专业水平的C++代码中，在类声明之外定义短成员函数非常罕见）。

构造函数和析构函数也可以是内联的。

12.8 带参数的构造函数

向构造函数传递变元是可能的。典型情况下，这些变元用于在创建对象时初始化该对象。为了创建带参数的构造函数，只要像对任何别的函数一样简单地给构造函数加上参数即可。在定义构造函数时，用这些参数来初始化对象。例如，下面是一个包含带参数的构造函数的类：

```
#include <iostream>
using namespace std;

class myclass {
```

```
  int a, b;
public:
  myclass(int i, int j) { a=i; b=j;}
  void show( ) { cout << a << " " << b;}
};

int main( )
{
  myclass ob(3, 5);

  ob.show( );

  return 0;
}
```

注意，在定义myclass( )时，参数i和j用来给a和b赋初值。

该程序演示了在声明使用带参数的构造函数的对象时指定变元的常用方法。下面的语句：

```
myclass ob(3, 4);
```

创建了一个对象ob并分别将变元3和4传给myclass( )的参数i和j。还可以用下面的声明语句传递变元：

```
myclass ob = myclass(3, 4);
```

然而，第一种方法是常用的方法，也是本书大多数范例使用的方法。实际上，在与拷贝构造函数相关的这两种类型的声明方法之间技术上有一些细微的差别(拷贝构造函数将在第14章讨论)。

下面是使用带参数的构造函数的另一个范例，它创建了一个存储关于图书馆书籍信息的类。

```
#include <iostream>
#include <cstring>
using namespace std;

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
  char author[ 40] ;
  char title[ 40] ;
  int status;
public:
  book(char *n, char *t, int s);
  int get_status( ) { return status;}
  void set_status(int s) { status = s;}
  void show( );
};

book::book(char *n, char *t, int s)
{
  strcpy(author, n);
  strcpy(title, t);
  status = s;
}
```

```
void book::show( )
{
  cout << title << " by " << author;
  cout << " is ";
  if(status==IN) cout << "in.\n";
  else cout << "out.\n";
}

int main( )
{
  book b1("Twain", "Tom Sawyer", IN);
  book b2("Melville", "Moby Dick", CHECKED_OUT);

  b1.show( );
  b2.show( );

  return 0;
}
```

带参数的构造函数非常有用，因为它们使用户不必为了初始化一个对象中的一个或多个变量而额外地进行函数调用。少一次函数调用，程序就提高一些效率。注意，短函数 get_status( ) 和 set_status( )定义在 book 类中。这是编写 C++ 程序时常见的用法。

12.9 带一个参数的构造函数：特例

如果构造函数只有一个参数，那么，存在第三种方法将初值传递给构造函数。例如，考虑下面的程序：

```
#include <iostream>
using namespace std;

class X {
  int a;
  public:
  X(int j) { a = j; }
  int geta( ) { return a; }
};

int main( )
{
  X ob = 99; // passes 99 to j

  cout << ob.geta( ); // outputs 99

  return 0;
}
```

这里，X 的构造函数带有一个参数。请特别注意在 main( )中如何声明 ob。在这种形式的初始化中，99 被自动传给 X( )构造函数中的 j 参数。即，这个声明语句被编译器处理，仿佛它是像下面这样编写的：

```
X ob = X(99);
```

一般来讲，在构造函数只要求一个变元的时候，可以使用 ob(i)或 ob = i 来初始化对象。理

由是: 无论何时创建一个带一个变元的构造函数，也可以隐式地创建一个从那个变元的类型到类的类型的转换。

记住，这里所示的另一种替代形式只适用于带一个参数的构造函数。

12.10 静态类成员

类的函数和数据成员都可以成为静态的。本节将解释静态数据成员和静态成员函数。

12.10.1 静态数据成员

当把 static 置于一个成员变量的声明之前时，编译器被告知该变量只存在一个副本，且该类的所有对象将共享这个变量。与规则的数据成员不同，不能为每个对象都构造一个静态成员变量的副本。不管创建的该类对象有多少个，静态数据成员的副本只有一个。所以，该类的所有对象使用的都是同一变量。在创建第一个对象前，所有静态变量都被初始化为0。

当在一个类内声明一个静态数据成员时，你不是在定义它（即，没有为其分配存储空间）。相反，必须在类外对静态数据成员提供全局定义。用作用域分辨符重新定义静态变量以指明它所归属的类，这样可以为变量分配内存（记住，类声明只是一种逻辑结构，而不是物理存在）。

考虑下面的程序，就会理解静态数据成员的作用和效果：

```
#include <iostream>
using namespace std;

class shared {
  static int a;
  int b;
public:
  void set(int i, int j) {a=i; b=j;}
  void show( );
} ;

int shared::a; // define a

void shared::show( )
{
  cout << "This is static a: " << a;
  cout << "\nThis is non-static b: " << b;
  cout << "\n";
}

int main( )
{
  shared x, y;

  x.set(1, 1); // set a to 1
  x.show( );

  y.set(2, 2); // change a to 2
  y.show( );

  x.show( ); /* Here, a has been changed for both x and y
                because a is shared by both objects. */
```

```
  return 0;
}
```

该程序运行时显示如下的输出：

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

注意，整数a在shared内和shared外均有声明。如前所述，因为在shared中声明的a没有分配内存，所以这是必要的。

注意：为方便起见，老版本的C++不要求第二次声明静态成员变量。但这种方便引起了严重的不一致性，几年前就被取消了。然而，仍然可以发现没有再次声明静态成员变量的老版本的C++代码。这种情况下，需要增加所要求的定义。

静态成员变量在它的类的任何对象创建之前就存在了。例如，在下面的这段程序里，a即是公有的，又是静态的，所以在main( )里能直接访问它。还有，由于a在类shared的任何对象创建之前就已存在，所以可以在任何时候给a赋值。正如这个程序所表明的，a的值不随对象x的创建而改变。因此，两个输出语句都显示相同的值：99。

```
#include <iostream>
using namespace std;

class shared {
public:
  static int a;
} ;

int shared::a; // define a

int main( )
{
  // initialize a before creating any objects
  shared::a = 99;

  cout << "This is initial value of a: " << shared::a;

  cout << "\n";

  shared x;

  cout << "This is x.a: " << x.a;

  return 0;
}
```

注意程序是如何用类名和作用域分辨符来引用a的。通常，当程序独立地引用一个对象的静态成员时，必须用包含这个成员的类名来限定它。

静态成员变量的一个用途就是对一些被类的所有对象使用的共享资源提供访问控制。例如，可能创建几个对象，每个对象要对某个磁盘文件进行写操作，但显然一次只允许一个对象对该文件进行写操作。在这种情况下，你可能希望声明一个静态变量，该变量指出该文件何时处于

使用状态，何时处于空闲状态。然后，每个对象在写到文件之前询问这个变量。下面的程序演示了如何使用这种类型的静态变量来控制访问稀有资源：

```
#include <iostream>
using namespace std;

class c1 {
  static int resource;
public:
  int get_resource( );
  void free_resource( ) {resource = 0;}
};

int c1::resource; // define resource

int c1::get_resource( )
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1; // resource allocated to this object
  }
}

int main( )
{
  c1 ob1, ob2;

  if(ob1.get_resource( )) cout << "ob1 has resource\n";

  if(!ob2.get_resource( )) cout << "ob2 denied resource\n";

  ob1.free_resource( ); // let someone else use it

  if(ob2.get_resource( ))
    cout << "ob2 can now use resource\n";

  return 0;
}
```

静态成员变量的另一个有趣的应用是记录现有的具体类的对象数，例如，

```
#include <iostream>
using namespace std;

class Counter {
public:
  static int count;
  Counter( ) { count++; }
  ~Counter( ) { count--; }
};
int Counter::count;

void f( );

int main(void)
{
```

```
  Counter o1;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  Counter o2;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  f( );
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";

  return 0;
}

void f( )
{
  Counter temp;
  cout << "Objects in existence: ";
  cout << Counter::count << "\n";
  // temp is destroyed when f( ) returns
}
```

这个程序生成下面的输出。

```
Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2
```

可以看到，静态成员变量count在创建一个对象时增1，在销毁一个对象时减1。用这种方式，记录现在存在多少个Counter类型的对象。

通过使用静态成员变量，实际上可以销毁任何全局变量。全局变量给OOP带来的问题就是它们几乎总是违背封装原则。

12.10.2 静态成员函数

成员函数也可以声明为静态的。对静态成员函数有一些限制。第一，静态成员函数只能引用这个类的其他静态成员（当然可以访问全局函数和数据）。第二，静态成员函数没有this指针（参见第13章有关this的介绍）。第三，同一个函数不能有静态和非静态两种版本，静态成员函数不可以是虚函数。最后，它们不能被声明为const或volatile。

下面是对前面的程序稍加修改后的版本。注意，get_resource( )现在声明为静态的。正如程序所演示的，get_resource( )既可以通过使用类名和作用域分辨符被其本身调用（独立于对象），也可以和对象联系起来调用。

```
#include <iostream>
using namespace std;

class cl {
  static int resource;
public:
  static int get_resource( );
  void free_resource( ) { resource = 0; }
```

```
};

int cl::resource; // define resource

int cl::get_resource( )
{
  if(resource) return 0; // resource already in use
  else {
    resource = 1;
    return 1; // resource allocated to this object
  }
}

int main( )
{
  cl ob1, ob2;

  /* get_resource( ) is static so may be called independent
     of any object. */
  if(cl::get_resource( )) cout << "ob1 has resource\n";

  if(!cl::get_resource( )) cout << "ob2 denied resource\n";

  ob1.free_resource( );

  if(ob2.get_resource( )) // can still call using object syntax
    cout << "ob2 can now use resource\n";

  return 0;
}
```

实际上，静态成员函数的应用是有限的，使用它的好处是在实际创建任何对象之前可以“预初始化”私有的静态数据。例如，下面是一段十分有效的C++程序：

```
#include <iostream>
using namespace std;

class static_type {
  static int i;
public:
  static void init(int x) { i = x;}
  void show( ) {cout << i;}
};

int static_type::i; // define i

int main( )
{
  // init static data before object creation
  static_type::init(100);

  static_type x;
  x.show( ); // displays 100

  return 0;
}
```

12.11 何时执行构造函数和析构函数

一般的规则是，在声明对象时，对象的构造函数即被调用，对象销毁时，其析构函数即被调用。下面讨论其细节。

局部对象的构造函数在遇到对象声明语句时执行。局部对象的析构函数按照与其构造函数相反的顺序执行。

全局对象的构造函数在main( )开始执行之前执行。全局构造函数在同一文件里按照它们声明的顺序执行。用户无法知道分布在几个文件里的全局构造函数执行的顺序。全局析构函数在main( )结束之后按照相反的顺序执行。

下面的程序演示构造函数和析构函数执行的时间：

```
#include <iostream>
using namespace std;

class myclass {
public:
  int who;
  myclass(int id);
  ~myclass( );
} glob_ob1(1), glob_ob2(2);

myclass::myclass(int id)
{
  cout << "Initializing " << id << "\n";
  who = id;
}

myclass::~myclass( )
{
  cout << "Destructing " << who << "\n";
}

int main( )
{
  myclass local_ob1(3);

  cout << "This will not be first line displayed.\n";

  myclass local_ob2(4);

  return 0;
}
```

它显示如下的输出：

```
Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
```

```
Destructing 1
```

由于编译器和执行环境的差别，你可能看到、也可能看不到最后两行的输出。

12.12 作用域分辨符

我们知道，运算符::用来连接类名和成员名，以告知编译器这个成员属于哪个类。然而，作用域分辨符还有另一个相关的用途：它允许在一个封闭作用域里被同一名称的局部声明“隐藏”的名字。例如，考虑下面的代码：

```
int i; // global i

void f( )
{
  int i; // local i

  i = 10; // uses local i
  .
  .
  .
}
```

如注释所说的那样，赋值表达式i=10引用局部变量i。但是，如果函数f( )要访问全局变量i怎么办？只需要将运算符::置于i之前即可，如下所示：

```
int i; // global i

void f( )
{
  int i; // local i

  ::i = 10; // now refers to global i
  .
  .
  .
}
```

12.13 嵌套类

可以在一个类中定义另一个类，这样就创建了嵌套类。事实上，因为一个类声明定义了一个作用域，嵌套类只在封闭类的作用域中有效。坦率地讲，嵌套类很少使用。由于C++的灵活性和强大的继承机制，实际上不需要使用嵌套类。

12.14 局部类

类可以在函数内定义。例如，下面是一段有效的C++程序：

```
#include <iostream>
using namespace std;

void f( );

int main( )
```

```
{
  f( );
  // myclass not known here
  return 0;
}

void f( )
{
  class myclass {
    int i;
  public:
    void put_i(int n) { i=n; }
    int get_i( ) { return i; }
  } ob;

  ob.put_i(10);
  cout << ob.get_i( );
}
```

当在一个函数内声明一个类时，这个类只为该函数所知，在函数外它是不可知的。

局部类有一些限制条件。第一，所有成员函数必须在类声明内定义，局部类不能使用或访问它所属的函数的局部变量（除了局部类可以访问某些静态局部变量外，这些局部变量是在函数内声明或者是作为 extern 声明的）。然而，它可以访问由所包含的函数定义的类型名和枚举值。第二，不能在局部类里声明任何静态变量。由于这些限制，在C++程序里局部类并不常见。

12.15 向函数传递对象

对象能以和其他任何类型的变量相同的方法传递到函数。对象是通过使用标准的按值调用机制（call-by-value）传递函数的。尽管对象的传递很直接，但会出现一些与构造函数和析构函数有关的意外事件。要理解为什么，考虑下面这个程序。

```
// Passing an object to a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  myclass(int n);
  ~myclass( );
  void set_i(int n) { i=n; }
  int get_i( ) { return i; }
};

myclass::myclass(int n)
{
  i = n;
  cout << "Constructing " << i << "\n";
}

myclass::~myclass( )
{
```

```
  cout << "Destroying " << i << "\n";
}

void f(myclass ob);

int main( )
{
  myclass o(1);

  f(o);
  cout << "This is i in main: ";
  cout << o.get_i( ) << "\n";

  return 0;
}

void f(myclass ob)
{
  ob.set_i(2);
  cout << "This is local i: " << ob.get_i( );
  cout << "\n";
}
```

这个程序生成下面的输出：

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

注意，如输出所示，构造函数只调用了一次，出现在main( )中o创建时，但析构函数调用了两次。让我们看看这是为什么。

当一个对象传递给函数时，那个对象生成了一个拷贝（这个拷贝变成了函数中的参数）。这意味着一个新对象得以存在。当函数终止时，变元（即参数）的拷贝被销毁。这就带来了两个问题。第一，生成拷贝时，对象的构造函数是否被调用了；第二，销毁拷贝时，析构函数是否被调用。最初，这两个问题的答案会让人感到吃惊。

当在函数调用过程中生成一个变元的拷贝时，没有调用构造函数。相反，调用了对象的拷贝构造函数。拷贝构造函数定义了对象的拷贝是如何生成的。正如第14章中解释的，可以为你所创建的类显式地定义一个拷贝构造函数。然而，如果一个类没有显式地定义一个拷贝构造函数，正如本例一样，那么，默认时C++会提供一个。这个默认的拷贝构造函数创建了这个对象的一个按位（即相同的）拷贝。如果想一想，就不难理解生成按位拷贝的原因了。因为我们使用一般的构造函数来初始化对象的某些方面，不必调用它来生成一个已存在对象的拷贝，这样一个调用会改变对象的内容。当把一个对象传给函数时，你会想要使用对象的当前状态，而不是它的初始状态。

然而，当函数终止并且用做变元的对象的拷贝销毁时，调用析构函数。这是必需的，因为对象不在作用域内。这就是前面的程序两次调用构造函数的原因。第一次是当f( )的参数位于作用域外时。第二次是当main( )中的o在程序结束时被销毁时。

总之，当对象的拷贝被创建以用做函数的变元时，不调用正常的构造函数相反，默认的拷

贝构造函数生成一个逐位相同的拷贝。然而，当拷贝销毁时（通常是由于在函数返回时超出了作用域），要调用析构函数。

因为默认的拷贝构造函数创建了原对象的一个精确的复制品，有时它也会带来一些麻烦。按值调用的参数传递机制，从理论上讲可以起到保护和隔离调用变元的作用，但即使按照这种方法将对象传递给函数，仍然可能产生副作用，甚至破坏作为变元的对象。例如，如果一个作为变元的对象分配内存并在其被销毁时释放这些内存，那么，函数中其局部拷贝在其析构函数被调用时将释放相同的内存，这样就破坏了原对象并使其变为无用的。要防止这种问题，需要通过为这个类创建一个拷贝构造函数来定义拷贝操作，正如第14章中解释的那样。

12.16 返回对象

函数可以向调用者返回对象。例如，下面是一段有效的程序：

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i( ) { return i; }
};

myclass f( ); // return object of type myclass

int main( )
{
  myclass o;

  o = f( );

  cout << o.get_i( ) << "\n";

  return 0;
}

myclass f( )
{
  myclass x;

  x.set_i(1);
  return x;
}
```

当对象被函数返回时，就自动生成一个临时对象来容纳返回值。函数返回的正是这个对象。值返回以后，这个对象即被销毁。这种临时对象的销毁在某些情况下可能产生意外的副作用。例如，如果由函数返回的对象有一析构函数释放动态分配的内存，那么，即使接收返回值的对象还在使用，这一内存也将被释放。本书第15章将讨论解决这些问题的方法，即引入重载赋值运算符和定义拷贝构造函数（参见第14章）。

12.17 对象赋值

假设有两个相同类型的对象，可以把一个对象赋给另一个对象。这导致右边对象的数据拷贝到左边对象的数据中。例如，下面的程序显示99：

```
// Assigning objects.
#include <iostream>
using namespace std;

class myclass {
  int i;
public:
  void set_i(int n) { i=n; }
  int get_i( ) { return i; }
};

int main( )
{
  myclass ob1, ob2;

  ob1.set_i(99);
  ob2 = ob1; // assign data from ob1 to ob2

  cout << "This is ob2's i: " << ob2.get_i( );

  return 0;
}
```

默认时，一个对象的所有数据都是通过使用按位拷贝赋给另一个对象的。然而，重载赋值运算符并定义一些其他的赋值过程也是可能的（参见第15章）。