第 16 章　高级继承

在此之前，你使用单继承和多重继承来创建 is-a（是一个）关系。

本章介绍如下内容：

- 什么是聚合（aggregation）？如何模拟它（has-a 关系）？
- 什么是代理（delegation）？如何模拟它？
- 如何以一个类的方式来实现另一个类？
- 如何使用私有继承？

16.1 聚　　合

正如你在以前的范例中见到的，类的成员数据可以是其他类的对象。这通常被称为聚合或 has-a（有一个）关系。

下面是 Name 类和 Address 类：

```
Class Name
{
     // Class information for Name
};
Class Address
{
     // Class information for Address
};
```

作为一个聚合的例子，可以将这两个类的对象作为 Employee 类的组成部分：

```
Class Employee
{
     Name    EmpName;
     Address EmpAddress;
     // Any other employee class stuff...
}
```

因此，Employee 类包含表示姓名和地址的成员变量（Employee 有一个 Name 以及 Employee 有一个 Address）。

程序清单 16.1 是一个更复杂的例子，不同于第 13 章创建的 String 类，这是一个不完整但很有用的 String 类。该程序清单没有任何输出，它只是提供了后面的程序清单使用的代码。

程序清单 16.1　String 类

```
0:  // Listing 16.1 The String Class
1:
```

```
#include <iostream>
#include <string.h>
using namespace std;

class String
{
  public:
    // constructors
    String();
    String(const char *const);
    String(const String &);
    ~String();

    // overloaded operators
    char & operator[](int offset);
    char operator[](int offset) const;
    String operator+(const String&);
    void operator+=(const String&);
    String & operator= (const String &);

    // General accessors
    int GetLen()const { return itsLen; }
    const char * GetString() const { return itsString; }
    // static int ConstructorCount;

  private:
    String (int);        // private constructor
    char * itsString;
    unsigned short itsLen;

};

// default constructor creates string of 0 bytes
String::String()
{
   itsString = new char[1];
   itsString[0] = '\0';
   itsLen=0;
   // cout << "\tDefault string constructor\n";
   // ConstructorCount++;
}

// private (helper) constructor, used only by
// class methods for creating a new string of
// required size.  Null filled.
String::String(int len)
{
   itsString = new char[len+1];
   for (int i = 0; i<=len; i++)
     itsString[i] = '\0';
   itsLen=len;
   // cout << "\tString(int) constructor\n";
```

```
54:    // ConstructorCount++;
55: }
56:
57: // Converts a character array to a String
58: String::String(const char * const cString)
59: {
60:    itsLen = strlen(cString);
61:    itsString = new char[itsLen+1];
62:    for (int i = 0; i<itsLen; i++)
63:       itsString[i] = cString[i];
64:    itsString[itsLen] = '\0';
65:    // cout << "\tString(char*) constructor\n";
66:    // ConstructorCount++;
67: }
68:
69: // copy constructor
70: String::String (const String & rhs)
71: {
72:    itsLen=rhs.GetLen();
73:    itsString = new char[itsLen+1];
74:    for (int i = 0; i<itsLen;i++)
75:       itsString[i] = rhs[i];
76:    itsString[itsLen] = '\0';
77:    // cout << "\tString(String&) constructor\n";
78:    // ConstructorCount++;
79: }
80:
81: // destructor, frees allocated memory
82: String::~String ()
83: {
84:    delete [] itsString;
85:    itsLen = 0;
86:    // cout << "\tString destructor\n";
87: }
88:
89: // operator equals, frees existing memory
90: // then copies string and size
91: String& String::operator=(const String & rhs)
92: {
93:    if (this == &rhs)
94:       return *this;
95:    delete [] itsString;
96:    itsLen=rhs.GetLen();
97:    itsString = new char[itsLen+1];
98:    for (int i = 0; i<itsLen;i++)
99:       itsString[i] = rhs[i];
100:   itsString[itsLen] = '\0';
101:   return *this;
102:   // cout << "\tString operator=\n";
103: }
104:
105: //non constant offset operator, returns
```

```
106: // reference to character so it can be
107: // changed!
108: char & String::operator[](int offset)
109: {
110:    if (offset > itsLen)
111:       return itsString[itsLen-1];
112:    else
113:       return itsString[offset];
114: }
115:
116: // constant offset operator for use
117: // on const objects (see copy constructor!)
118: char String::operator[](int offset) const
119: {
120:    if (offset > itsLen)
121:       return itsString[itsLen-1];
122:    else
123:       return itsString[offset];
124: }
125:
126: // creates a new string by adding current
127: // string to rhs
128: String String::operator+(const String& rhs)
129: {
130:    int  totalLen = itsLen + rhs.GetLen();
131:    String temp(totalLen);
132:    int i, j;
133:    for (i = 0; i<itsLen; i++)
134:       temp[i] = itsString[i];
135:    for (j = 0; j<rhs.GetLen(); j++, i++)
136:       temp[i] = rhs[j];
137:    temp[totalLen] = '\0';
138:    return temp;
139: }
140:
141: // changes current string, returns nothing
142: void String::operator+=(const String& rhs)
143: {
144:    unsigned short rhsLen = rhs.GetLen();
145:    unsigned short totalLen = itsLen + rhsLen;
146:    String  temp(totalLen);
147:    int i, j;
148:    for (i = 0; i<itsLen; i++)
149:       temp[i] = itsString[i];
150:    for (j = 0; j<rhs.GetLen(); j++, i++)
151:       temp[i] = rhs[i-itsLen];
152:    temp[totalLen] = '\0';
153:    *this = temp;
154: }
155:
156:  // int String::ConstructorCount = 0;
```

注意：将程序清单 16.1 中的代码保存到文件 String.hpp 中。以后要用到 String 类时，都可以使用

#include"String.hpp"将程序清单 16.1 包含进来。读者可能注意到了，该程序清单中有很多注释行，贯穿本章都将解释这些代码行。

分析：

程序清单 16.1 提供了一个 String 类，它与程序清单 13.12 使用的 String 类极其相似。最著的差别在于，在程序清单 13.12 中，构造函数和其他一些函数使用打印语句来显示其用途，而在程序清单 16.1 中，这些打印语句被注释掉了。这些函数将在以后的范例中用到。

第 25 行声明了静态成员变量 ConstructorCount，第 156 行对其进行了初始化。该变量在每次调用 String 构造函数时都将加 1。当前，这些代码都被注释掉，将在后面的程序清单中使用它们。

出于方便考虑，将实现与类声明放在一起。在实际的程序中，将把类声明保存在文件 String.hpp 中，而将实现保存在 String.cpp 中。然后将 String.cpp 添加到程序中（通过添加文件或使用 make 文件），并在 String.cpp 中包含文件 String.cpp。

当然，在实际程序中，将使用 C++标准库中的 String 类，而不是这里的 String 类。

程序清单 16.2 描述了 Employee 类，它包含 3 个 String 对象。这些对象用于存储雇员的姓、名和地址。

程序清单 16.2　Employee 类及使用它的程序

```
 0: // Listing 16.2 The Employee Class and Driver Program
 1: #include "MyString.hpp"
 2:
 3: class Employee
 4: {
 5:   public:
 6:     Employee();
 7:     Employee(char *, char *, char *, long);
 8:     ~Employee();
 9:     Employee(const Employee&);
10:     Employee & operator= (const Employee &);
11:
12:     const String & GetFirstName() const
13:       { return itsFirstName; }
14:     const String & GetLastName() const { return itsLastName; }
15:     const String & GetAddress() const { return itsAddress; }
16:     long GetSalary() const { return itsSalary; }
17:
18:     void SetFirstName(const String & fName)
19:       { itsFirstName = fName; }
20:     void SetLastName(const String & lName)
21:       { itsLastName = lName; }
22:     void SetAddress(const String & address)
23:       { itsAddress = address; }
24:     void SetSalary(long salary) { itsSalary = salary; }
25:   private:
26:     String   itsFirstName;
27:     String   itsLastName;
28:     String   itsAddress;
29:     long     itsSalary;
30: };
31:
32: Employee::Employee():
33:   itsFirstName(""),
```

```
34:    itsLastName(""),
35:    itsAddress(""),
36:    itsSalary(0)
37: {}
38:
39: Employee::Employee(char * firstName, char * lastName,
40:    char * address, long salary):
41:    itsFirstName(firstName),
42:    itsLastName(lastName),
43:    itsAddress(address),
44:    itsSalary(salary)
45: {}
46:
47: Employee::Employee(const Employee & rhs):
48:    itsFirstName(rhs.GetFirstName()),
49:    itsLastName(rhs.GetLastName()),
50:    itsAddress(rhs.GetAddress()),
51:    itsSalary(rhs.GetSalary())
52: {}
53:
54: Employee::~Employee() {}
55:
56: Employee & Employee::operator= (const Employee & rhs)
57: {
58:    if (this == &rhs)
59:       return *this;
60:
61:    itsFirstName = rhs.GetFirstName();
62:    itsLastName = rhs.GetLastName();
63:    itsAddress = rhs.GetAddress();
64:    itsSalary = rhs.GetSalary();
65:
66:    return *this;
67: }
68:
69: int main()
70: {
71:    Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
72:    Edie.SetSalary(50000);
73:    String LastName("Levine");
74:    Edie.SetLastName(LastName);
75:    Edie.SetFirstName("Edythe");
76:
77:    cout << "Name: ";
78:    cout << Edie.GetFirstName().GetString();
79:    cout << " " << Edie.GetLastName().GetString();
80:    cout << ".\nAddress: ";
81:    cout << Edie.GetAddress().GetString();
82:    cout << ".\nSalary: " ;
83:    cout << Edie.GetSalary();
84:    return 0;
85: }
```

输出：

```
Name: Edythe Levine.
Address: 1461 Shore Parkway.
Salary: 50000
```

分析：

程序清单 16.2 中的 Employee 类包含 3 个 String 对象（第 26～28 行）：itsFirstName、itsSecondName 和 itsAddress。

第 71 行中创建了一个名为 Edie 的 Employee 对象，并传入了 4 个值。第 72 行以字面量 50000 作为参数调用了 Employee 的存取器函数 SetSalary( )。在实际程序中，这要么是动态值（在运行阶段设置），要么是常量。

第 73 行创建了一个名为 LastName 的 String 对象，并使用一个 C++字符串常量对其进行初始化。然后在第 74 行将该 String 对象作为参数来调用 SetLastName( )函数。

第 75 行使用另一个字符串常量作为参数来调用 Employee 的 SetFirstName( )函数。然而，如果仔细查看，你将发现 Employee 并没有接受字符串作为参数的 SetFirstName( )函数，SetFirstName( )函数接受一个常量 String 引用作为参数（见第 18 行）。

编译器知道如何根据常量字符串创建 String 对象，因此能够解决这种问题。编译器之所以知道如何完成这项工作，是因为在程序清单 16.1 的第 11 行指出了一点。

第 78、79 和 81 行有一些不同寻常的地方。读者可能会问，为何将 GetString( )附接在 Employee 类的不同方法的后面：

```
78:    cout<<Edie.GetFirstName().GetString();
```

Edie 对象的 GetFirstName( )方法返回一个 String。不幸的是，程序清单 16.1 中声明的 String 类不支持运算符 cout<<。为满足 cout 的要求，需要返回一个 C-风格字符串。对 String 对象调用 GetString( )方法将返回一个 C-风格字符串。稍后将解决这个问题。

16.1.1　访问被聚合类的成员

聚合了其他对象的类在访问这些对象的成员数据和成员函数方面并没有特权，而只有与其他常规类一样的访问权限。例如，Employee 对象在访问 String 类的成员变量方面并没有特权。如果 Employee 对象 Edie 试图访问其成员变量 itsFirstName 的私有成员变量 itsLen，将导致编译错误。然而，这并不是什么缺点。存取器函数为 String 类提供了接口，Employee 类不必关系 String 类的实现细节，只需关心 int 变量 itsSalary 是如何存储其信息的。

注意：被聚合的成员在访问聚合它的类的成员方面没有任何特权，它们唯一的特别之处是，在创建时或创建后，将传给它们一个指向其所属对象的 this 指针，除此之外，访问权限与其他函数完全相同。

16.1.2　控制对被聚合成员的访问

String 类提供了重载的 operator+。通过将所有存取 String 成员的函数（如 GetFirstName( )）声明为返回常量引用，Employee 类的设计者阻止了通过 Employee 对象调用 operator+。由于 operator+不是（也不能是）const 函数（它修改调用它时针对的对象），因此试图编写下面的语句将导致编译错误：

```
String buffer = Edie.GetFirstName() + Edie.GetLastName();
```

由于 GetFirstName( )返回一个常量 String，而你不能对常量对象调用 operator+。

为解决这个问题，可将 GetFirstName( )重载为非常量的：

```
const String & GetFirstName() const { return itsFirstName; }
String & GetFirstName()  { return itsFirstName; }
```

返回值不再是一个 const，成员函数本身也不再是 const。要重载函数，仅修改返回值是不够的，还必须修改函数本身的常量性（constness）。

16.1.3 聚合的代价

采用聚合时，可能在性能方面付出代价。每次创建或复制 Employee 对象时，都需要创建其包含的 String 对象。

为说明构造函数被调用的频繁程度，不要将程序清单 16.1 中的 cout 语句作为注释。程序清单 16.3 重写了 main()函数，添加了打印语句来指出在程序的什么地方创建了对象。请取消对程序清单 16.1 中 cout 语句的注释，然后编译程序清单 16.3。

注意：编译该程序清单前，请取消对程序清单 16.1 中第 40、53、65、77、86 和第 102 行的注释。

程序清单 16.3　被聚合的类的构造函数的调用情况

```
 0: //Listing 16.3 Aggregated Class Constructors
 1: #include "MyString.hpp"
 2:
 3: class Employee
 4: {
 5:   public:
 6:     Employee();
 7:     Employee(char *, char *, char *, long);
 8:     ~Employee();
 9:     Employee(const Employee&);
10:     Employee & operator= (const Employee &);
11:
12:     const String & GetFirstName() const
13:       { return itsFirstName; }
14:     const String & GetLastName() const { return itsLastName; }
15:     const String & GetAddress() const { return itsAddress; }
16:     long GetSalary() const { return itsSalary; }
17:
18:     void SetFirstName(const String & fName)
19:       { itsFirstName = fName; }
20:     void SetLastName(const String & lName)
21:       { itsLastName = lName; }
22:     void SetAddress(const String & address)
23:       { itsAddress = address; }
24:     void SetSalary(long salary) { itsSalary = salary; }
25:   private:
26:     String   itsFirstName;
27:     String   itsLastName;
28:     String   itsAddress;
29:     long     itsSalary;
30: };
31:
32: Employee::Employee():
33:   itsFirstName(""),
34:   itsLastName(""),
35:   itsAddress(""),
36:   itsSalary(0)
37: {}
```

```
38:
39: Employee::Employee(char * firstName, char * lastName,
40:    char * address, long salary):
41:    itsFirstName(firstName),
42:    itsLastName(lastName),
43:    itsAddress(address),
44:    itsSalary(salary)
45: {}
46:
47: Employee::Employee(const Employee & rhs):
48:    itsFirstName(rhs.GetFirstName()),
49:    itsLastName(rhs.GetLastName()),
50:    itsAddress(rhs.GetAddress()),
51:    itsSalary(rhs.GetSalary())
52: {}
53:
54: Employee::~Employee() {}
55:
56: Employee & Employee::operator= (const Employee & rhs)
57: {
58:    if (this == &rhs)
59:       return *this;
60:
61:    itsFirstName = rhs.GetFirstName();
62:    itsLastName = rhs.GetLastName();
63:    itsAddress = rhs.GetAddress();
64:    itsSalary = rhs.GetSalary();
65:
66:    return *this;
67: }
68:
69: int main()
70: {
71:    cout << "Creating Edie...\n";
72:    Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
73:    Edie.SetSalary(20000);
74:    cout << "Calling SetFirstName with char *...\n";
75:    Edie.SetFirstName("Edythe");
76:    cout << "Creating temporary string LastName...\n";
77:    String LastName("Levine");
78:    Edie.SetLastName(LastName);
79:
80:    cout << "Name: ";
81:    cout << Edie.GetFirstName().GetString();
82:    cout << " " << Edie.GetLastName().GetString();
83:    cout << "\nAddress: ";
84:    cout << Edie.GetAddress().GetString();
85:    cout << "\nSalary: " ;
86:    cout << Edie.GetSalary();
87:    cout << endl;
88:    return 0;
89: }
```

输出：

```
 1: Creating Edie...
 2:         String(char*) constructor
 3:         String(char*) constructor
 4:         String(char*) constructor
 5: Calling SetFirstName with char *...
 6:         String(char*) constructor
 7:         String destructor
 8: Creating temporary string LastName...
 9:         String(char*) constructor
10: Name: Edythe Levine
11: Address: 1461 Shore Parkway
12: Salary: 20000
13:         String destructor
14:         String destructor
15:         String destructor
16:         String destructor
```

分析：

程序清单 16.3 和 16.2 一样，使用了程序清单 16.1 中的类声明，但没有被注释掉其中的 cout 语句。为便于分析，对程序清单 16.3 的输出编了号。

在程序清单 16.3 的第 71 行，打印“Creating Edie...”，这对应于输出的第 1 行。第 72 行使用 4 个参数创建了 Employee 对象 Edie，其中前 3 个参数为字符串。如预期的那样，输出表明 String 的构造函数被调用了 3 次。

第 74 行打印出一条信息，第 75 行为语句 Edie.SetFirstName(“Edythe”)。该语句导致使用字符串“Edythe”来创建一个临时 String 对象，输出的第 5 行和第 6 行反映了这一点。注意，赋值语句使用完该临时 String 后，它被立即销毁。

第 77 行创建一个 String 对象。在这里，程序员显式地完成了编译器隐式地为前一条语句完成的工作。输出的第 8 行表明，构造函数被调用，但没有调用析构函数。执行到函数末尾，超出作用域后，该对象才被销毁。

到达 main()函数末尾后，由于 Employee 对象超出了作用域，因此其包含的 String 对象都被销毁，包括第 77 行创建的 String 对象 LastName。

16.1.4 按值传递导致复制

程序清单 16.3 表明，创建 Employee 对象时将导致 String 构造函数被调用 3 次，然后给两个 String 成员赋值时导致 String 构造函数被调用两次。程序清单 16.4 重写了 main()函数，这次没有使用打印语句，而是取消对 String 静态成员变量 ConstructorCount 的注释，使用它来确定 String 构造函数被调用的次数。

从程序清单 16.1 可知，每次调用 String 构造函数时，变量 ConstructorCount 都将加 1。在程序清单 16.4 的 main()函数中调用了两个打印函数；调用第一个打印函数时按引用传递一个 Employee 对象，调用第二个打印函数时按值传递一个 Employee 对象。将 Employee 对象作为参数传递时，ConstructorCount 变量记录创建了多少个 String 对象。

注意：编译该程序清单之前，保留编译程序清单 16.3 时撤销的对程序清单 16.1 中代码行的注释，同时撤销对程序清单 16.1 中第 25、41、54、66、78 和 155 行的注释。

程序清单 16.4 按值传递导致复制

```
 0: // Listing 16.4 Passing by Value
 1: #include "MyString.hpp"
```

```
 2:
 3: class Employee
 4: {
 5:   public:
 6:     Employee();
 7:     Employee(char *, char *, char *, long);
 8:     ~Employee();
 9:     Employee(const Employee&);
10:     Employee & operator= (const Employee &);
11:
12:     const String & GetFirstName() const
13:       { return itsFirstName; }
14:     const String & GetLastName() const { return itsLastName; }
15:     const String & GetAddress() const { return itsAddress; }
16:     long GetSalary() const { return itsSalary; }
17:
18:     void SetFirstName(const String & fName)
19:       { itsFirstName = fName; }
20:     void SetLastName(const String & lName)
21:       { itsLastName = lName; }
22:     void SetAddress(const String & address)
23:       { itsAddress = address; }
24:     void SetSalary(long salary) { itsSalary = salary; }
25:   private:
26:     String   itsFirstName;
27:     String   itsLastName;
28:     String   itsAddress;
29:     long     itsSalary;
30: };
31:
32: Employee::Employee():
33:   itsFirstName(""),
34:   itsLastName(""),
35:   itsAddress(""),
36:   itsSalary(0)
37: {}
38:
39: Employee::Employee(char * firstName, char * lastName,
40:   char * address, long salary):
41:   itsFirstName(firstName),
42:   itsLastName(lastName),
43:   itsAddress(address),
44:   itsSalary(salary)
45: {}
46:
47: Employee::Employee(const Employee & rhs):
48:   itsFirstName(rhs.GetFirstName()),
49:   itsLastName(rhs.GetLastName()),
50:   itsAddress(rhs.GetAddress()),
51:   itsSalary(rhs.GetSalary())
52: {}
53:
```

```
54: Employee::~Employee() {}
55:
56: Employee & Employee::operator= (const Employee & rhs)
57: {
58:    if (this == &rhs)
59:       return *this;
60:
61:    itsFirstName = rhs.GetFirstName();
62:    itsLastName = rhs.GetLastName();
63:    itsAddress = rhs.GetAddress();
64:    itsSalary = rhs.GetSalary();
65:
66:    return *this;
67: }
68:
69: void PrintFunc(Employee);
70: void rPrintFunc(const Employee&);
71:
72: int main()
73: {
74:    Employee Edie("Jane","Doe","1461 Shore Parkway", 20000);
75:    Edie.SetSalary(20000);
76:    Edie.SetFirstName("Edythe");
77:    String LastName("Levine");
78:    Edie.SetLastName(LastName);
79:
80:    cout << "Constructor count: " ;
81:    cout << String::ConstructorCount << endl;
82:    rPrintFunc(Edie);
83:    cout << "Constructor count: ";
84:    cout << String::ConstructorCount << endl;
85:    PrintFunc(Edie);
86:    cout << "Constructor count: ";
87:    cout << String::ConstructorCount << endl;
88:    return 0;
89: }
90: void PrintFunc (Employee Edie)
91: {
92:    cout << "Name: ";
93:    cout << Edie.GetFirstName().GetString();
94:    cout << " " << Edie.GetLastName().GetString();
95:    cout << ".\nAddress: ";
96:    cout << Edie.GetAddress().GetString();
97:    cout << ".\nSalary: " ;
98:    cout << Edie.GetSalary();
99:    cout << endl;
100: }
101:
102: void rPrintFunc (const Employee& Edie)
103: {
104:    cout << "Name: ";
105:    cout << Edie.GetFirstName().GetString();
```

```
106:    cout << " " << Edie.GetLastName().GetString();
107:    cout << "\nAddress: ";
108:    cout << Edie.GetAddress().GetString();
109:    cout << "\nSalary: " ;
110:    cout << Edie.GetSalary();
111:    cout << endl;
112: }
```

输出：

```
String(char*) constructor
String(char*) constructor
String(char*) constructor
String(char*) constructor
String destructor
String(char*) constructor
Constructor count: 5
Name: Edythe Levine
Address: 1461 Shore Parkway
Salary: 20000
Constructor count: 5
   String(String&) constructor
   String(String&) constructor
   String(String&) constructor
Name: Edythe Levine.
Address: 1461 Shore Parkway.
Salary: 20000
   String destructor
   String destructor
   String destructor
Constructor count: 8
   String destructor
   String destructor
   String destructor
   String destructor
```

分析：

上述输出表明，第 74 行创建 Employee 对象时创建了 3 个 String 对象，接下来重新给两个 String 成员赋值时（第 76 和 78 行），共调用了 String 构造函数两次。第 82 行按引用将该 Employee 对象传递给 rPrintFunc( ) 时，没有创建 Employee 对象，因此没有创建 String 对象（它们也是按引用传递）。从输出可知，调用函数 rPrintFunc( )后，计数器 ConstructorCount 的值仍为 5，这表明没有再调用 String 构造函数。

第 85 行将 Employee 对象按值传递给 PrintFunc( )时，创建了该对象的一个拷贝，因此（通过调用复制构造函数）创建了 3 个 String 对象。

16.2　以继承方式实现和聚合/代理

有时候，想让一个类使用另一个类的一些功能。例如，假设要创建一个 PartsCatalog 类，已有的规范将 PartsCatalog 定义为零件集合，其中每个零件都有独一无二的零件号。PartsCatalog 不允许其中的任何两项相同，但允许根据零件号来访问其中的零件。

第 2 周复习中的程序清单提供了一个 PartsList 类。这个 PartsList 经过了仔细测试，读者也对其有深入理

解，因此你可能想以这个 PartsList 类为基础创建 PartsCatalog 类，而不是从头开始创建。

创建 PartsCatalog 类时，可以让它包含一个 PartsList 对象。PartsCatalog 可以将管理链表的工作交给它聚合的 PartsList 对象。

另一种解决方案是，从 PartsList 类派生出 PartCatalog 类，以继承 PartsList 类的属性。然而，公有继承提供的是 is-a 关系，你应自问：PartsCatalog 是否确实是一种 PartsList。

要回答问题“PartsCatalog 是否是一个 PartsList”，方法之一是假设 PartsList 是基类，PartsCatalog 是派生类，然后提出以下几个问题：

（1）基类中有什么东西不应出现在派生类中吗？例如，基类 PartsList 中有不适合于 PartsCatalog 类的函数吗？如果是这样，可能不应使用公有继承。

（2）要创建的类是否可能包含多个基类对象？例如，PartsCatalog 是否需要两个 PartsList 才能完成其工作？如果是这样，几乎可以肯定应使用聚合。

（3）需要从基类继承，以便可以利用虚函数或访问受保护的成员吗？如果是这样，必须使用公有或私有继承。

根据这些问题的答案，必须在公有继承（is-a 关系）和私有继承（稍后介绍）或聚合（has-a 关系）之间做出选择。

术语

这里使用了多个术语，下面对它们做一下总结：

聚合：将一个类的对象声明为另一个类的成员，这也被称为 has-a 关系。

代理：使用被聚合类的成员执行包含类的功能。

以...方式实现：在一个类的基础上创建另一个类，而不使用公有继承（例如，使用保护继承或私有继承）。

使用代理

为什么不从 PartsList 派生出 PartsCatalog 呢？PartsCatalog 不是一种 PartsList，因为 PartsList 是有序集合，集合中的成员可以重复。PartsCatalog 中的条目没有排序，且必须是独一无二的。PartsCatalog 中第 5 个成员的零件编号并不是 5。

当然，也可以从 PartsList 公开继承，然后覆盖函数 Insert()和下标运算符（[]），使它们做正确的工作，但这改变了 PartsList 类的本质。相反，应创建一个没有下标运算符、不允许条目重复、将 operator+定义为合并两个集合的 PartsCatalog 类。

为此，第一种方法是使用聚合。PartsCatalog 将链表管理工作交给被聚合的 PartsList 去完成。程序清单 16.5 演示了这种方法。

程序清单 16.5　将链表管理工作交给被聚合的 PartsList

```
0:  // Listing 16.5 Delegating to an Aggregated PartsList
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  // ********** Part ***********
6:
7:  // Abstract base class of parts
8:  class Part
9:  {
10:   public:
11:     Part():itsPartNumber(1) {}
12:     Part(int PartNumber):
13:     itsPartNumber(PartNumber){}
```

```
14:     virtual ~Part(){}
15:     int GetPartNumber() const
16:       { return itsPartNumber; }
17:     virtual void Display() const =0;
18:   private:
19:     int itsPartNumber;
20: };
21:
22: // implementation of pure virtual function so that
23: // derived classes can chain up
24: void Part::Display() const
25: {
26:   cout << "\nPart Number: " << itsPartNumber << endl;
27: }
28:
29: // *************** Car Part ************
30:
31: class CarPart : public Part
32: {
33:   public:
34:     CarPart():itsModelYear(94){}
35:     CarPart(int year, int partNumber);
36:     virtual void Display() const
37:     {
38:       Part::Display();
39:       cout << "Model Year: ";
40:       cout << itsModelYear << endl;
41:     }
42:   private:
43:     int itsModelYear;
44: };
45:
46: CarPart::CarPart(int year, int partNumber):
47:   itsModelYear(year),
48:   Part(partNumber)
49: {}
50:
51:
52: // *************** AirPlane Part ************
53:
54: class AirPlanePart : public Part
55: {
56:   public:
57:     AirPlanePart():itsEngineNumber(1){};
58:     AirPlanePart
59:     (int EngineNumber, int PartNumber);
60:     virtual void Display() const
61:     {
62:       Part::Display();
63:       cout << "Engine No.: ";
64:       cout << itsEngineNumber << endl;
65:     }
```

```
66:   private:
67:     int itsEngineNumber;
68: };
69:
70: AirPlanePart::AirPlanePart
71: (int EngineNumber, int PartNumber):
72:   itsEngineNumber(EngineNumber),
73:   Part(PartNumber)
74: {}
75:
76: // ********** Part Node ***********
77: class PartNode
78: {
79:   public:
80:     PartNode (Part*);
81:     ~PartNode();
82:     void SetNext(PartNode * node)
83:       { itsNext = node; }
84:     PartNode * GetNext() const;
85:     Part * GetPart() const;
86:   private:
87:     Part *itsPart;
88:     PartNode * itsNext;
89: };
90: // PartNode Implementations...
91:
92: PartNode::PartNode(Part* pPart):
93:   itsPart(pPart),
94:   itsNext(0)
95: {}
96:
97: PartNode::~PartNode()
98: {
99:    delete itsPart;
100:   itsPart = 0;
101:   delete itsNext;
102:   itsNext = 0;
103: }
104:
105: // Returns NULL if no next PartNode
106: PartNode * PartNode::GetNext() const
107: {
108:   return itsNext;
109: }
110:
111: Part * PartNode::GetPart() const
112: {
113:   if (itsPart)
114:     return itsPart;
115:   else
116:     return NULL; //error
117: }
```

```
118:
119:
120:
121: // ***************** Part List ************
122: class PartsList
123: {
124:  public:
125:    PartsList();
126:    ~PartsList();
127:    // needs copy constructor and operator equals!
128:    void    Iterate(void (Part::*f)()const) const;
129:    Part*   Find(int & position, int PartNumber)  const;
130:    Part*   GetFirst() const;
131:    void    Insert(Part *);
132:    Part*   operator[](int) const;
133:    int     GetCount() const { return itsCount; }
134:    static  PartsList& GetGlobalPartsList()
135:    {
136:       return  GlobalPartsList;
137:    }
138:  private:
139:    PartNode * pHead;
140:    int itsCount;
141:    static PartsList GlobalPartsList;
142: };
143:
144: PartsList PartsList::GlobalPartsList;
145:
146:
147: PartsList::PartsList():
148:   pHead(0),
149:   itsCount(0)
150: {}
151:
152: PartsList::~PartsList()
153: {
154:   delete pHead;
155: }
156:
157: Part*  PartsList::GetFirst() const
158: {
159:   if (pHead)
160:      return pHead->GetPart();
161:   else
162:      return NULL;  // error catch here
163: }
164:
165: Part * PartsList::operator[](int offSet) const
166: {
167:    PartNode* pNode = pHead;
168:
169:    if (!pHead)
```

```
170:       return NULL; // error catch here
171:
172:    if (offSet > itsCount)
173:       return NULL; // error
174:
175:    for (int i=0;i<offSet; i++)
176:       pNode = pNode->GetNext();
177:
178:    return   pNode->GetPart();
179: }
180:
181: Part*  PartsList::Find(
182:    int & position,
183:    int PartNumber)  const
184: {
185:    PartNode * pNode = 0;
186:    for (pNode = pHead, position = 0;
187:         pNode!=NULL;
188:         pNode = pNode->GetNext(), position++)
189:    {
190:       if (pNode->GetPart()->GetPartNumber() == PartNumber)
191:          break;
192:    }
193:    if (pNode == NULL)
194:       return NULL;
195:    else
196:       return pNode->GetPart();
197: }
198:
199: void PartsList::Iterate(void (Part::*func)()const) const
200: {
201:    if (!pHead)
202:       return;
203:    PartNode* pNode = pHead;
204:    do
205:       (pNode->GetPart()->*func)();
206:    while ((pNode = pNode->GetNext()) != 0);
207: }
208:
209: void PartsList::Insert(Part* pPart)
210: {
211:    PartNode * pNode = new PartNode(pPart);
212:    PartNode * pCurrent = pHead;
213:    PartNode * pNext = 0;
214:
215:    int New = pPart->GetPartNumber();
216:    int Next = 0;
217:    itsCount++;
218:
219:    if (!pHead)
220:    {
221:       pHead = pNode;
```

```
222:       return;
223:     }
224:
225:     // if this one is smaller than head
226:     // this one is the new head
227:     if (pHead->GetPart()->GetPartNumber() > New)
228:     {
229:       pNode->SetNext(pHead);
230:       pHead = pNode;
231:       return;
232:     }
233:
234:     for (;;)
235:     {
236:       // if there is no next, append this new one
237:       if (!pCurrent->GetNext())
238:       {
239:         pCurrent->SetNext(pNode);
240:         return;
241:       }
242:
243:       // if this goes after this one and before the next
244:       // then insert it here, otherwise get the next
245:       pNext = pCurrent->GetNext();
246:       Next = pNext->GetPart()->GetPartNumber();
247:       if (Next > New)
248:       {
249:         pCurrent->SetNext(pNode);
250:         pNode->SetNext(pNext);
251:         return;
252:       }
253:       pCurrent = pNext;
254:     }
255: }
256:
257: class PartsCatalog
258: {
259:   public:
260:     void Insert(Part *);
261:     int Exists(int PartNumber);
262:     Part * Get(int PartNumber);
263:     operator+(const PartsCatalog &);
264:     void ShowAll() { thePartsList.Iterate(Part::Display); }
265:   private:
266:     PartsList thePartsList;
267: };
268:
269: void PartsCatalog::Insert(Part * newPart)
270: {
271:    int partNumber = newPart->GetPartNumber();
272:    int offset;
273:
```

```
274:    if (!thePartsList.Find(offset, partNumber))
275:    {
276:      thePartsList.Insert(newPart);
277:    }
278:    else
279:    {
280:      cout << partNumber << " was the ";
281:      switch (offset)
282:      {
283:        case 0:  cout << "first "; break;
284:        case 1:  cout << "second "; break;
285:        case 2:  cout << "third "; break;
286:        default: cout << offset+1 << "th ";
287:      }
288:      cout << "entry. Rejected!" << endl;
289:    }
290: }
291:
292: int PartsCatalog::Exists(int PartNumber)
293: {
294:    int offset;
295:    thePartsList.Find(offset,PartNumber);
296:    return offset;
297: }
298:
299: Part * PartsCatalog::Get(int PartNumber)
300: {
301:    int offset;
302:    Part * thePart = thePartsList.Find(offset, PartNumber);
303:    return thePart;
304: }
305:
306:
307: int main()
308: {
309:    PartsCatalog pc;
310:    Part * pPart = 0;
311:    int PartNumber;
312:    int value;
313:    int choice = 99;
314:
315:    while (choice != 0)
316:    {
317:      cout << "(0)Quit (1)Car (2)Plane: ";
318:      cin >> choice;
319:
320:      if (choice != 0)
321:      {
322:        cout << "New PartNumber?: ";
323:        cin >>  PartNumber;
324:
325:        if (choice == 1)
```

```
326:          {
327:             cout << "Model Year?: ";
328:             cin >> value;
329:             pPart = new CarPart(value,PartNumber);
330:          }
331:          else
332:          {
333:             cout << "Engine Number?: ";
334:             cin >> value;
335:             pPart = new AirPlanePart(value,PartNumber);
336:          }
337:          pc.Insert(pPart);
338:       }
339:    }
340:    pc.ShowAll();
341:    return 0;
342: }
```

输出：

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

注意：在有些编译器中，第 264 行不能通过编译，虽然它是合法的 C++代码。如果你的编译器是这样的，请将其修改为：

```
264:       void ShowAll() { thePartsList.Iterate(&Part::Display); }
```

即在 Part::Display 前面加上&。如果这样可以编译，立即致电编译器厂商，向它们提意见。

分析：

程序清单 16.5 使用了第 2 周复习中的 Part、PartNode 和 PartsList 类。

第 257～267 行声明了一个新类：PartsCatalog。PartsCatalog 将一个 PartsList 对象作为其数据成员（第 265

行），将链表管理工作交给它。换句话说，PartsCatalog 是以 PartsList 的方式实现的。

注意，PartsCatalog 的客户不能直接访问 PartsList，因为 PartsList 被声明为私有成员。只有通过 PartsCatalog 才能访问 PartsList，因此 PartsList 的行为发生了巨大变化。例如，PartsCatalog::Insert( )方法不允许将重复的条目插入到 PartsList 中。

PartsCatalog::Insert( )的实现从第 269 行开始。提取作为参数传入的 Part 的 itsPartNumber 成员变量的值。

第 274 行将这个值传递给 PartsList 的 Find( )方法，如果没有发现匹配的值，将这个值插入链表中（第 276 行），否则打印一条错误消息（从都 280 行开始）。

注意，PartsCatalog 通过对其成员变量 pl（一个 PartsList 对象）调用 Insert( )函数来完成实际的插入工作。插入和链表维护以及搜索和检索链表的工作由 PartsCatalog 的成员 PartsList 完成。PartsCatalog 没有理由重新包含这些代码，它可以充分利用已定义好的接口。

这就是 C++中可重用性的本质：PartsCatalog 可重用 PartsList 的代码，而 PartsCatalog 的设计者无需关心 PartsList 的实现细节。PartsList 的接口（即类声明）提供了 PartsCatalog 的设计者所需的全部信息。

注意：有关 PartsList 的更详细的信息，请参阅第 2 周复习中的程序清单和分析。

16.3 私有继承

如果 PartsCatalog 需要访问 PartsList 的保护成员（在这个例子中没有这样的情况）或需要覆盖 PartsList 的任何一个方法，则 PartsCatalog 将必须从 PartsList 继承而来。

由于 PartsCatalog 不是一个 PartsList 对象，且你不想将 PartsList 的全部功能暴露给 PartsCatalog 客户，因此需要使用私有继承。私有继承让你能够从一个类派生出另一个类，同时使基类的成员对派生类来说是私有的。

有关私有继承最重要的一点是，所有基类成员变量和函数都被视为私有的，而不管它们在基类中的实际访问限制级别如何。这样，对于任何不是 PartsCatalog 成员函数的函数来说，从 PartsList 继承而来的每个函数都是不可访问的。下面这一点很重要：私有继承不涉及继承接口，只涉及继承实现。

对于 PartsCatalog 类的客户而言，PartsList 类是不可见的。它们不能直接使用 PartsList 类的接口；不能调用 PartsList 的任何方法；但可以调用 PartsCatalog 的方法，而 PartsCatalog 的方法可以访问 PartsList 的任何方法，因为 PartsCatalog 是从 PartsList 派生而来。这里重要的一点是：不像公有继承意味的那样，PartsCatalog 并不是一个 PartsList；而像聚合那样，它是以 PartsList 的方式实现的。私有继承只是提供了方便。

程序清单 16.6 将 PartsCatalog 类声明为以私有方式从 PartsList 类派生而来，演示了私有继承的用法。

程序清单 16.6 私有继承

```
 0: //Listing 16.6 demonstrates private inheritance
 1: #include <iostream>
 2: using namespace std;
 3:
 4: // *************** Part ************
 5:
 6: // Abstract base class of parts
 7: class Part
 8: {
 9:   public:
10:     Part():itsPartNumber(1) {}
11:     Part(int PartNumber):
12:     itsPartNumber(PartNumber){}
13:     virtual ~Part(){}
```

```
14:     int GetPartNumber() const
15:       { return itsPartNumber; }
16:     virtual void Display() const =0;
17:   private:
18:     int itsPartNumber;
19: };
20:
21: // implementation of pure virtual function so that
22: // derived classes can chain up
23: void Part::Display() const
24: {
25:    cout << "\nPart Number: " << itsPartNumber << endl;
26: }
27:
28: // ********** Car Part ************
29:
30: class CarPart : public Part
31: {
32:   public:
33:     CarPart():itsModelYear(94){}
34:     CarPart(int year, int partNumber);
35:     virtual void Display() const
36:     {
37:       Part::Display();
38:       cout << "Model Year: ";
39:       cout << itsModelYear << endl;
40:     }
41:   private:
42:     int itsModelYear;
43: };
44:
45: CarPart::CarPart(int year, int partNumber):
46:    itsModelYear(year),
47:    Part(partNumber)
48: {}
49:
50:
51: // ********** AirPlane Part ************
52:
53: class AirPlanePart : public Part
54: {
55:   public:
56:     AirPlanePart():itsEngineNumber(1){};
57:     AirPlanePart(int EngineNumber, int PartNumber);
58:     virtual void Display() const
59:     {
60:       Part::Display();
61:       cout << "Engine No.: ";
62:       cout << itsEngineNumber << endl;
63:     }
64:   private:
65:     int itsEngineNumber;
```

```
66: };
67:
68: AirPlanePart::AirPlanePart
69: (int EngineNumber, int PartNumber):
70:    itsEngineNumber(EngineNumber),
71:    Part(PartNumber)
72: {}
73:
74: // ********** Part Node ***********
75: class PartNode
76: {
77:   public:
78:     PartNode (Part*);
79:     ~PartNode();
80:     void SetNext(PartNode * node)
81:        { itsNext = node; }
82:     PartNode * GetNext() const;
83:     Part * GetPart() const;
84:   private:
85:     Part *itsPart;
86:     PartNode * itsNext;
87: };
88: // PartNode Implementations...
89:
90: PartNode::PartNode(Part* pPart):
91:    itsPart(pPart),
92:    itsNext(0)
93: {}
94:
95: PartNode::~PartNode()
96: {
97:    delete itsPart;
98:    itsPart = 0;
99:    delete itsNext;
100:   itsNext = 0;
101: }
102:
103: // Returns NULL if no next PartNode
104: PartNode * PartNode::GetNext() const
105: {
106:    return itsNext;
107: }
108:
109: Part * PartNode::GetPart() const
110: {
111:    if (itsPart)
112:       return itsPart;
113:    else
114:       return NULL; //error
115: }
116:
117:
```

```
118:
119: // ********** Part List ***********
120: class PartsList
121: {
122:   public:
123:     PartsList();
124:     ~PartsList();
125:     // needs copy constructor and operator equals!
126:     void    Iterate(void (Part::*f)()const) const;
127:     Part*   Find(int & position, int PartNumber)  const;
128:     Part*   GetFirst() const;
129:     void    Insert(Part *);
130:     Part*   operator[](int) const;
131:     int   GetCount() const { return itsCount; }
132:     static   PartsList& GetGlobalPartsList()
133:     {
134:       return  GlobalPartsList;
135:     }
136:   private:
137:     PartNode * pHead;
138:     int itsCount;
139:     static PartsList GlobalPartsList;
140: };
141:
142: PartsList PartsList::GlobalPartsList;
143:
144:
145: PartsList::PartsList():
146:   pHead(0),
147:   itsCount(0)
148: {}
149:
150: PartsList::~PartsList()
151: {
152:   delete pHead;
153: }
154:
155: Part*  PartsList::GetFirst() const
156: {
157:   if (pHead)
158:     return pHead->GetPart();
159:   else
160:     return NULL;  // error catch here
161: }
162:
163: Part * PartsList::operator[](int offSet) const
164: {
165:   PartNode* pNode = pHead;
166:
167:   if (!pHead)
168:     return NULL; // error catch here
169:
```

```
170:    if (offSet > itsCount)
171:       return NULL; // error
172:
173:    for (int i=0;i<offSet; i++)
174:       pNode = pNode->GetNext();
175:
176:    return  pNode->GetPart();
177: }
178:
179: Part*  PartsList::Find(int & position, int PartNumber)  const
180: {
181:    PartNode * pNode = 0;
182:    for (pNode = pHead, position = 0;
183:       pNode!=NULL;
184:    pNode = pNode->GetNext(), position++)
185:    {
186:       if (pNode->GetPart()->GetPartNumber() == PartNumber)
187:       break;
188:    }
189:    if (pNode == NULL)
190:       return NULL;
191:    else
192:       return pNode->GetPart();
193: }
194:
195: void PartsList::Iterate(void (Part::*func)()const) const
196: {
197:    if (!pHead)
198:       return;
199:    PartNode* pNode = pHead;
200:    do
201:       (pNode->GetPart()->*func)();
202:    while ((pNode = pNode->GetNext()) != 0);
203: }
204:
205: void PartsList::Insert(Part* pPart)
206: {
207:    PartNode * pNode = new PartNode(pPart);
208:    PartNode * pCurrent = pHead;
209:    PartNode * pNext = 0;
210:
211:    int New =  pPart->GetPartNumber();
212:    int Next = 0;
213:    itsCount++;
214:
215:    if (!pHead)
216:    {
217:       pHead = pNode;
218:       return;
219:    }
220:
221:    // if this one is smaller than head
```

```
222:    // this one is the new head
223:    if (pHead->GetPart()->GetPartNumber() > New)
224:    {
225:      pNode->SetNext(pHead);
226:      pHead = pNode;
227:      return;
228:    }
229:
230:    for (;;)
231:    {
232:      // if there is no next, append this new one
233:      if (!pCurrent->GetNext())
234:      {
235:        pCurrent->SetNext(pNode);
236:        return;
237:      }
238:
239:      // if this goes after this one and before the next
240:      // then insert it here, otherwise get the next
241:      pNext = pCurrent->GetNext();
242:      Next = pNext->GetPart()->GetPartNumber();
243:      if (Next > New)
244:      {
245:        pCurrent->SetNext(pNode);
246:        pNode->SetNext(pNext);
247:        return;
248:      }
249:      pCurrent = pNext;
250:    }
251:  }
252:
253:  class PartsCatalog : private PartsList
254:  {
255:    public:
256:      void Insert(Part *);
257:      int Exists(int PartNumber);
258:      Part * Get(int PartNumber);
259:      operator+(const PartsCatalog &);
260:      void ShowAll() { Iterate(Part::Display); }
261:    private:
262:  };
263:
264:  void PartsCatalog::Insert(Part * newPart)
265:  {
266:    int partNumber =  newPart->GetPartNumber();
267:    int offset;
268:
269:    if (!Find(offset, partNumber))
270:    {
271:      PartsList::Insert(newPart);
272:    }
273:    else
```

```
274:    {
275:       cout << partNumber << " was the ";
276:       switch (offset)
277:       {
278:         case 0:  cout << "first "; break;
279:         case 1:  cout << "second "; break;
280:         case 2:  cout << "third "; break;
281:         default: cout << offset+1 << "th ";
282:       }
283:       cout << "entry. Rejected!" << endl;
284:    }
285: }
286:
287: int PartsCatalog::Exists(int PartNumber)
288: {
289:    int offset;
290:    Find(offset,PartNumber);
291:    return offset;
292: }
293:
294: Part * PartsCatalog::Get(int PartNumber)
295: {
296:    int offset;
297:    return (Find(offset, PartNumber));
298:
299: }
300:
301: int main()
302: {
303:    PartsCatalog pc;
304:    Part * pPart = 0;
305:    int PartNumber;
306:    int value;
307:    int choice = 99;
308:
309:    while (choice != 0)
310:    {
311:       cout << "(0)Quit (1)Car (2)Plane: ";
312:       cin >> choice;
313:
314:       if (choice != 0)
315:       {
316:          cout << "New PartNumber?: ";
317:          cin >>  PartNumber;
318:
319:          if (choice == 1)
320:          {
321:             cout << "Model Year?: ";
322:             cin >> value;
323:             pPart = new CarPart(value,PartNumber);
324:          }
325:          else
```

```
326:          {
327:             cout << "Engine Number?: ";
328:             cin >> value;
329:             pPart = new AirPlanePart(value,PartNumber);
330:          }
331:          pc.Insert(pPart);
332:       }
333:    }
334:    pc.ShowAll();
335:    return 0;
336: }
```

输出：

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

分析：

程序清单 16.6 修改了 PartsCatalog 的接口和 main( )函数。其他类的接口与程序清单 16.5 中相同。

在程序清单 16.6 的第 253 行，将 PartsCatalog 声明为以私有方式从 PartsList 派生而来。虽然 PartsCatalog 的接口与程序清单 16.5 中相同，但不再需要将一个 PartsList 对象作为其成员数据。

在第 260 行，PartsCatalog 的 ShowAll( )函数调用 PartsList 的 Iterate( )函数，并将合适 Part 类成员函数作为参数。ShowAll( )函数充当了函数 Iterate( )的公有接口，它提供正确的信息，但禁止客户类直接调用 Iterate( )函数。虽然 PartsList 可能允许将其他函数作为参数传递给 Iterate( )函数，但 PartsCatalog 不允许。

对 Insert( )函数（第 164～284 行）也做了修改。注意第 269 行直接调用了 Find( )函数，因为它现在是从基类继承而来的。第 271 行对 Insert( )函数的调用必须完全限定，否则它将无休止地自我递归。

总之，PartsCatalog 的方法可以直接调用 PartsList 的方法。惟一的例外是，如果 PartsCatalog 覆盖了某个方法，而又需要调用该方法的 PartsList 版本时必须对方法名进行完全限定。

私有继承让 PartsCatalog 继承了它可以使用的东西，同时对于 Insert( )函数以及客户类不应直接访问的其他方法，提供了间接访问途径。

应该：

当派生类是一种基类时应采用公有继承。

要将功能交给另一个类去实现但无需访问其保护成员时应使用聚合。

需要以一个类的方式来实现另一个类且需要访问基类的保护成员时，应使用私有继承。

不应该：

需要使用多个基类实例时，不要使用私有继承，而必须使用聚合。例如，如果 PartsCatalog 需要两个 PartsList 对象，则不能使用私有继承。

不想让基类的成员对派生类的客户可用时，不应使用公有继承。

16.4 添加友元类

有时候，你会同时创建多个类，将它们作为一组。例如，PartNode 和 PartsList 是紧密耦合的，如果 PartsList 能够直接访问 PartNode 的指针 itsPart，将非常方便。

你想将 itsPart 声明为私有而不是公有或保护的，因为这是 PartNode 的实现细节。然而，你想将其暴露给 PartsList。

要将私有成员数据或函数暴露给另一个类，必须将后者声明为友元类。这样就将类的接口扩展为包括友元类。

一个类将另一个类声明为友元类后，前者所有的成员数据和函数对后者来说都是公有的。例如，如果 PartsNode 将 PartsList 声明为友元类，则对 PartsList 来说，PartsNode 的所有成员数据和函数都是公有的。

需要指出的是，友元关系不能传递。虽然你是我的朋友，而 Joe 是你的朋友，但 Joe 并不一定是我的朋友。友元关系也不能继承。虽然你是我的朋友，我愿意与你共享我的秘密，但这并不意味着我愿意与你的孩子共享我的秘密。

最后，友元关系不是互通的。将类 1 声明为类 2 的友元并不能使类 2 成为类 1 的友元。也许你愿意将你的秘密告诉我，但这并不意味着我愿意将我的秘密告诉你。

要将类声明为友元，可使用 C++关键字 friend：

```
class ClassOne
{
    public:
      friend class BefriendedClass;
          . . .
```

在这个例子中，ClassOne 将 BefriendClass 声明为其友元。这意味着 BefriendClass 能够访问 ClassOne 的任何成员。

程序清单 16.7 说明友元关系，它重写了程序清单 16.6 中的范例，将 PartsList 声明为 PartsNode 的友元。注意，这并不会使 PartsNode 成为 PartsList 的友元。

程序清单 16.7　演示友元类

```
0:  //Listing 16.7 Friend Class Illustrated
1:
2:  #include <iostream>
3:  using namespace std;
4:
5:  // ********** Part ***********
6:
7:  // Abstract base class of parts
8:  class Part
```

```
 9: {
10:   public:
11:     Part():itsPartNumber(1) {}
12:     Part(int PartNumber):
13:     itsPartNumber(PartNumber){}
14:     virtual ~Part(){}
15:     int GetPartNumber() const
16:       { return itsPartNumber; }
17:     virtual void Display() const =0;
18:   private:
19:     int itsPartNumber;
20: };
21:
22: // implementation of pure virtual function so that
23: // derived classes can chain up
24: void Part::Display() const
25: {
26:   cout << "\nPart Number: ";
27:   cout << itsPartNumber << endl;
28: }
29:
30: // *************** Car Part ************
31:
32: class CarPart : public Part
33: {
34:   public:
35:     CarPart():itsModelYear(94){}
36:     CarPart(int year, int partNumber);
37:     virtual void Display() const
38:     {
39:       Part::Display();
40:       cout << "Model Year: ";
41:       cout << itsModelYear << endl;
42:     }
43:   private:
44:     int itsModelYear;
45: };
46:
47: CarPart::CarPart(int year, int partNumber):
48:   itsModelYear(year),
49:   Part(partNumber)
50: {}
51:
52:
53: // *************** AirPlane Part ************
54:
55: class AirPlanePart : public Part
56: {
57:   public:
58:     AirPlanePart():itsEngineNumber(1){};
59:     AirPlanePart(int EngineNumber, int PartNumber);
60:     virtual void Display() const
```

```
61:     {
62:       Part::Display();
63:       cout << "Engine No.: ";
64:       cout << itsEngineNumber << endl;
65:     }
66:   private:
67:     int itsEngineNumber;
68: };
69:
70: AirPlanePart::AirPlanePart(int EngineNumber, int PartNumber):
71:    itsEngineNumber(EngineNumber),
72:    Part(PartNumber)
73: {}
74:
75: // ********** Part Node **********
76: class PartNode
77: {
78:   public:
79:     friend class PartsList;
80:     PartNode (Part*);
81:     ~PartNode();
82:     void SetNext(PartNode * node)
83:       { itsNext = node; }
84:     PartNode * GetNext() const;
85:     Part * GetPart() const;
86:   private:
87:     Part *itsPart;
88:     PartNode * itsNext;
89: };
90:
91:
92: PartNode::PartNode(Part* pPart):
93:    itsPart(pPart),
94:    itsNext(0)
95: {}
96:
97: PartNode::~PartNode()
98: {
99:    delete itsPart;
100:   itsPart = 0;
101:   delete itsNext;
102:   itsNext = 0;
103: }
104:
105: // Returns NULL if no next PartNode
106: PartNode * PartNode::GetNext() const
107: {
108:   return itsNext;
109: }
110:
111: Part * PartNode::GetPart() const
112: {
```

```
113:    if (itsPart)
114:      return itsPart;
115:    else
116:      return NULL; //error
117:  }
118:
119:
120:  // *************** Part List ************
121:  class PartsList
122:  {
123:   public:
124:     PartsList();
125:     ~PartsList();
126:     // needs copy constructor and operator equals!
127:     void     Iterate(void (Part::*f)()const) const;
128:     Part*    Find(int & position, int PartNumber) const;
129:     Part*    GetFirst() const;
130:     void     Insert(Part *);
131:     Part*    operator[](int) const;
132:     int      GetCount() const { return itsCount; }
133:     static   PartsList& GetGlobalPartsList()
134:     {
135:        return  GlobalPartsList;
136:     }
137:   private:
138:     PartNode * pHead;
139:     int itsCount;
140:     static PartsList GlobalPartsList;
141:  };
142:
143:  PartsList PartsList::GlobalPartsList;
144:
145:  // Implementations for Lists...
146:
147:  PartsList::PartsList():
148:    pHead(0),
149:    itsCount(0)
150:  {}
151:
152:  PartsList::~PartsList()
153:  {
154:    delete pHead;
155:  }
156:
157:  Part*   PartsList::GetFirst() const
158:  {
159:    if (pHead)
160:       return pHead->itsPart;
161:    else
162:       return NULL;  // error catch here
163:  }
164:
```

```
165: Part * PartsList::operator[](int offSet) const
166: {
167:    PartNode* pNode = pHead;
168:
169:    if (!pHead)
170:       return NULL; // error catch here
171:
172:    if (offSet > itsCount)
173:       return NULL; // error
174:
175:    for (int i=0;i<offSet; i++)
176:       pNode = pNode->itsNext;
177:
178:    return  pNode->itsPart;
179: }
180:
181: Part* PartsList::Find(int & position, int PartNumber) const
182: {
183:    PartNode * pNode = 0;
184:    for (pNode = pHead, position = 0;
185:       pNode!=NULL;
186:    pNode = pNode->itsNext, position++)
187:    {
188:       if (pNode->itsPart->GetPartNumber() == PartNumber)
189:          break;
190:    }
191:    if (pNode == NULL)
192:       return NULL;
193:    else
194:       return pNode->itsPart;
195: }
196:
197: void PartsList::Iterate(void (Part::*func)()const) const
198: {
199:    if (!pHead)
200:       return;
201:    PartNode* pNode = pHead;
202:    do
203:       (pNode->itsPart->*func)();
204:    while (pNode = pNode->itsNext);
205: }
206:
207: void PartsList::Insert(Part* pPart)
208: {
209:    PartNode * pNode = new PartNode(pPart);
210:    PartNode * pCurrent = pHead;
211:    PartNode * pNext = 0;
212:
213:    int New =  pPart->GetPartNumber();
214:    int Next = 0;
215:    itsCount++;
216:
```

```
217:    if (!pHead)
218:    {
219:       pHead = pNode;
220:       return;
221:    }
222:
223:    // if this one is smaller than head
224:    // this one is the new head
225:    if (pHead->itsPart->GetPartNumber() > New)
226:    {
227:       pNode->itsNext = pHead;
228:       pHead = pNode;
229:       return;
230:    }
231:
232:    for (;;)
233:    {
234:       // if there is no next, append this new one
235:       if (!pCurrent->itsNext)
236:       {
237:          pCurrent->itsNext = pNode;
238:          return;
239:       }
240:
241:       // if this goes after this one and before the next
242:       // then insert it here, otherwise get the next
243:       pNext = pCurrent->itsNext;
244:       Next = pNext->itsPart->GetPartNumber();
245:       if (Next > New)
246:       {
247:          pCurrent->itsNext = pNode;
248:          pNode->itsNext = pNext;
249:          return;
250:       }
251:       pCurrent = pNext;
252:    }
253: }
254:
255: class PartsCatalog : private PartsList
256: {
257:  public:
258:    void Insert(Part *);
259:    int Exists(int PartNumber);
260:    Part * Get(int PartNumber);
261:    operator+(const PartsCatalog &);
262:    void ShowAll() { Iterate(Part::Display); }
263:  private:
264: };
265:
266: void PartsCatalog::Insert(Part * newPart)
267: {
268:    int partNumber =  newPart->GetPartNumber();
```

```
269:    int offset;
270:
271:    if (!Find(offset, partNumber))
272:      PartsList::Insert(newPart);
273:    else
274:    {
275:      cout << partNumber << " was the ";
276:      switch (offset)
277:      {
278:        case 0:  cout << "first "; break;
279:        case 1:  cout << "second "; break;
280:        case 2:  cout << "third "; break;
281:        default: cout << offset+1 << "th ";
282:      }
283:      cout << "entry. Rejected!" << endl;
284:    }
285: }
286:
287: int PartsCatalog::Exists(int PartNumber)
288: {
289:    int offset;
290:    Find(offset,PartNumber);
291:    return offset;
292: }
293:
294: Part * PartsCatalog::Get(int PartNumber)
295: {
296:    int offset;
297:    return (Find(offset, PartNumber));
298: }
299:
300: int main()
301: {
302:    PartsCatalog pc;
303:    Part * pPart = 0;
304:    int PartNumber;
305:    int value;
306:    int choice = 99;
307:
308:    while (choice != 0)
309:    {
310:      cout << "(0)Quit (1)Car (2)Plane: ";
311:      cin >> choice;
312:
313:      if (choice != 0)
314:       {
315:        cout << "New PartNumber?: ";
316:        cin >>  PartNumber;
317:
318:        if (choice == 1)
319:        {
320:          cout << "Model Year?: ";
```

```
321:            cin >> value;
322:            pPart = new CarPart(value,PartNumber);
323:          }
324:          else
325:          {
326:            cout << "Engine Number?: ";
327:            cin >> value;
328:            pPart = new AirPlanePart(value,PartNumber);
329:          }
330:          pc.Insert(pPart);
331:        }
332:      }
333:      pc.ShowAll();
334:      return 0;
335:  }
```

输出：

```
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4434
Model Year?: 93
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 1234
Model Year?: 94
1234 was the first entry. Rejected!
(0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2345
Model Year?: 93
(0)Quit (1)Car (2)Plane: 0

Part Number: 1234
Model Year: 94

Part Number: 2345
Model Year: 93

Part Number: 4434
Model Year: 93
```

分析：

第 79 行将类 PartsList 声明为类 PartNode 的友元。

在这个程序清单中，友元声明放在公有部分，但并非必须这样做；可以将该语句放在类声明的任何地方而不会改变其含义。由于这条语句，对类 PartsList 的任何成员函数来说，PartNode 的所有私有成员数据和函数都是可用的。

在第 157 行，成员函数 GetFirst( )的实现反映了这种变化。该函数现在通过代码 pHead->itsPart 返回私有成员数据，而不是返回 pHead->GetPart。同样，现在 Insert( )函数可以使用代码 pNode->itsNext = pHead 而不是 pNode->SetNext(pHead)。

必须承认，这些都是微不足道的变化，没有充分的理由将 PartsList 声明为 PartNode 的友元，但它们确实说明了关键字“friend”的工作原理。

一定要小心使用友元类声明。如果两个类的联系非常紧密，其中一个类必须频繁访问另一个类的数据，则可能有充分的理由使用这种声明。然而，应尽可能少用这种声明，通常使用公有存取器函数更简单，这样当你修改一个类时，不用重新编译另一个类。

注意：你经常会听到 C++程序员新手抱怨友元声明破坏了封装，而后者对面向对象编程而言非常重要，这种说法未必正确。友元声明使友元类成为类接口的一部分，并不一定会破坏封装。使用友元声明意味着有义务同时维护两个类，这可能降低模块化程度。

友元类

要将一个类声明为另一个类的友元，可在后者的声明中使用关键字 friend。也就是说，我可以说你是我的朋友，但你不能说你是我的朋友。

范例：

```
class PartNode{
public:
friend class PartsList;  // declares PartsList to be a friend of PartNode
};
```

16.5 友元函数

将另一个类声明为友元给它提供了所有访问权限。有时候可能不想将这种级别的访问权限授予整个类，而只授予这个类的一两个函数。为此，可以将另一个类的成员函数声明为友元，而不是将整个类声明为友元。事实上，可以将任何函数声明为友元函数，而不管它是否是另一个类的成员函数。

16.6 友元函数和运算符重载

程序清单 16.1 中的 String 类重载了 operator+，还提供了一个接受常量字符指针作为参数的构造函数，以便可以使用 C-风格字符串来创建 String 对象。这让你能够创建一个 String 对象并将其与一个 C-风格字符串相加。

注意：C-风格字符串是以空字符结尾的字符数组，如 char myString[ ] = "Hello World"。

然而，你不能像下面这样创建一个 C-风格字符串并将其与一个 String 对象相加：

```
char cString[] = {"Hello"};
String sString(" World");
String sStringTwo = cString + sString;  //error!
```

C-风格字符串没有重载 operator+。正如第 10 章讨论的，代码 cString + sString 实际上表示的是 cString.operator+(sString)。由于不能对 C-风格字符串调用 operator+( )，因此这将导致编译错误。

为解决这种问题，可在 String 中声明一个这样的友元函数：它将 operator+重载为接受两个 String 对象作为参数。这样，将通过合适的构造函数将 C-风格字符串转换为 String 对象，然后使用两个 String 对象为参数来调用 operator+。为明白这一点，请看程序清单 16.8。

程序清单 16.8 友元函数 operator+

```
0:  //Listing 16.8 - friendly operators
1:
```

```
#include <iostream>
#include <string.h>
using namespace std;

// Rudimentary string class
class String
{
  public:
    // constructors
    String();
    String(const char *const);
    String(const String &);
    ~String();

    // overloaded operators
    char & operator[](int offset);
    char operator[](int offset) const;
    String operator+(const String&);
    friend String operator+(const String&, const String&);
    void operator+=(const String&);
    String & operator= (const String &);

    // General accessors
    int GetLen()const { return itsLen; }
    const char * GetString() const { return itsString; }

  private:
    String (int);        // private constructor
    char * itsString;
    unsigned short itsLen;
};

// default constructor creates string of 0 bytes
String::String()
{
   itsString = new char[1];
   itsString[0] = '\0';
   itsLen=0;
   // cout << "\tDefault string constructor" << endl;
   // ConstructorCount++;
}

// private (helper) constructor, used only by
// class methods for creating a new string of
// required size.  Null filled.
String::String(int len)
{
   itsString = new char[len+1];
   for (int i = 0; i<=len; i++)
     itsString[i] = '\0';
   itsLen=len;
   // cout << "\tString(int) constructor" << endl;
   // ConstructorCount++;
```

```
55: }
56:
57: // Converts a character array to a String
58: String::String(const char * const cString)
59: {
60:    itsLen = strlen(cString);
61:    itsString = new char[itsLen+1];
62:    for (int i = 0; i<itsLen; i++)
63:       itsString[i] = cString[i];
64:    itsString[itsLen] = '\0';
65:    // cout << "\tString(char*) constructor" << endl;
66:    // ConstructorCount++;
67: }
68:
69: // copy constructor
70: String::String (const String & rhs)
71: {
72:    itsLen=rhs.GetLen();
73:    itsString = new char[itsLen+1];
74:    for (int i = 0; i<itsLen;i++)
75:       itsString[i] = rhs[i];
76:    itsString[itsLen] = '\0';
77:    // cout << "\tString(String&) constructor" << endl;
78:    // ConstructorCount++;
79: }
80:
81: // destructor, frees allocated memory
82: String::~String ()
83: {
84:    delete [] itsString;
85:    itsLen = 0;
86:    // cout << "\tString destructor" << endl;
87: }
88:
89: // operator equals, frees existing memory
90: // then copies string and size
91: String& String::operator=(const String & rhs)
92: {
93:    if (this == &rhs)
94:       return *this;
95:    delete [] itsString;
96:    itsLen=rhs.GetLen();
97:    itsString = new char[itsLen+1];
98:    for (int i = 0; i<itsLen;i++)
99:       itsString[i] = rhs[i];
100:   itsString[itsLen] = '\0';
101:   return *this;
102:   // cout << "\tString operator=" << endl;
103: }
104:
105: //non constant offset operator, returns
106: // reference to character so it can be
107: // changed!
```

```
108: char & String::operator[](int offset)
109: {
110:    if (offset > itsLen)
111:       return itsString[itsLen-1];
112:    else
113:       return itsString[offset];
114: }
115:
116: // constant offset operator for use
117: // on const objects (see copy constructor!)
118: char String::operator[](int offset) const
119: {
120:    if (offset > itsLen)
121:       return itsString[itsLen-1];
122:    else
123:       return itsString[offset];
124: }
125:
126: // creates a new string by adding current
127: // string to rhs
128: String String::operator+(const String& rhs)
129: {
130:    int  totalLen = itsLen + rhs.GetLen();
131:    String temp(totalLen);
132:    int i, j;
133:    for (i = 0; i<itsLen; i++)
134:       temp[i] = itsString[i];
135:    for (j = 0, i = itsLen; j<rhs.GetLen(); j++, i++)
136:       temp[i] = rhs[j];
137:    temp[totalLen] = '\0';
138:    return temp;
139: }
140:
141: // creates a new string by adding
142: // one string to another
143: String operator+(const String& lhs, const String& rhs)
144: {
145:    int  totalLen = lhs.GetLen() + rhs.GetLen();
146:    String temp(totalLen);
147:    int i, j;
148:    for (i = 0; i<lhs.GetLen(); i++)
149:       temp[i] = lhs[i];
150:    for (j = 0, i = lhs.GetLen(); j<rhs.GetLen(); j++, i++)
151:       temp[i] = rhs[j];
152:    temp[totalLen] = '\0';
153:    return temp;
154: }
155:
156: int main()
157: {
158:    String s1("String One ");
159:    String s2("String Two ");
160:    char *c1 = { "C-String One " } ;
```

```
161:    String s3;
162:    String s4;
163:    String s5;
164:
165:    cout << "s1: " << s1.GetString() << endl;
166:    cout << "s2: " << s2.GetString() << endl;
167:    cout << "c1: " << c1 << endl;
168:    s3 = s1 + s2;
169:    cout << "s3: " << s3.GetString() << endl;
170:    s4 = s1 + c1;
171:    cout << "s4: " << s4.GetString() << endl;
172:    s5 = c1 + s2;
173:    cout << "s5: " << s5.GetString() << endl;
174:    return 0;
175: }
```

输出：

```
s1: String One
s2: String Two
c1: C-String One
s3: String One String Two
s4: String One C-String One
s5: C-String One String Two
```

分析：

除 operator+外，String 类的方法实现与程序清单 16.1 中相同。第 20 行再次重载了 operator+，使其接受两个常量 String 引用作为参数并返回一个 String，该函数被声明为友元。

注意，这个 operator+不是这个类或任何其他类的成员函数。在 String 类的声明中声明它只是为了使其成为友元，但由于它已被声明了，因此不需要额外的函数原型。

该 operator+的实现位于第 143～154 行。除接受两个 String 参数（通过公有的存取器方法来访问它们）外，该 operator+与以前的 operator+相似。

在 main()函数中，第 172 行演示了这个函数的用法，其中是对一个 C-风格字符串调用 operator+的。

友元函数

要将函数声明为友元，可使用关键字 friend 以及该函数的权限定名。将函数声明为友元并没有给它提供访问 this 指针的权限，但确实提供了访问所有私有和保护成员数据和函数的权限。

范例：

```
class PartNode
{  // ...
     // make another class's member function a_friend
     friend void PartsList::Insert(Part *);
     // make a global function a friend
     friend int SomeFunction();
     // ...
};
```

16.7 重载插入运算符

现在，可以给 String 类提供像其他类型一样使用 cout 的功能。到目前为止，要打印 String 对象时，必须

使用这样的代码：

```
cout<<theString.GetString();
```

你希望可以编写这样的代码：

```
cout<<theString;
```

为此，必须重载 operator <<()。第 17 章将介绍用于 iostream 的 cin 和 cout；就现在而言，程序清单 16.9 演示了如何使用友元函数来重载 operator<<。

程序清单 16.9　重载 operator<<

```
// Listing 16.9 Overloading operator<<()

#include <iostream>
#include <string.h>
using namespace std;

class String
{
  public:
    // constructors
    String();
    String(const char *const);
    String(const String &);
    ~String();

    // overloaded operators
    char & operator[](int offset);
    char operator[](int offset) const;
    String operator+(const String&);
    void operator+=(const String&);
    String & operator= (const String &);
    friend ostream& operator<<
       ( ostream& theStream,String& theString);
    // General accessors
    int GetLen()const { return itsLen; }
    const char * GetString() const { return itsString; }

  private:
    String (int);        // private constructor
    char * itsString;
    unsigned short itsLen;
};


// default constructor creates string of 0 bytes
String::String()
{
   itsString = new char[1];
   itsString[0] = '\0';
   itsLen=0;
   // cout << "\tDefault string constructor" << endl;
   // ConstructorCount++;
```

```
42: }
43:
44: // private (helper) constructor, used only by
45: // class methods for creating a new string of
46: // required size.  Null filled.
47: String::String(int len)
48: {
49:    itsString = new char[len+1];
50:    for (int i = 0; i<=len; i++)
51:       itsString[i] = '\0';
52:    itsLen=len;
53:    // cout << "\tString(int) constructor" << endl;
54:    // ConstructorCount++;
55: }
56:
57: // Converts a character array to a String
58: String::String(const char * const cString)
59: {
60:    itsLen = strlen(cString);
61:    itsString = new char[itsLen+1];
62:    for (int i = 0; i<itsLen; i++)
63:       itsString[i] = cString[i];
64:    itsString[iteLen] = '\0';
65:    // cout << "\tString(char*) constructor" << endl;
66:    // ConstructorCount++;
67: }
68:
69: // copy constructor
70: String::String (const String & rhs)
71: {
72:    itsLen=rhs.GetLen();
73:    itsString = new char[itsLen+1];
74:    for (int i = 0; i<itsLen;i++)
75:       itsString[i] = rhs[i];
76:    itsString[itsLen] = '\0';
77:    // cout << "\tString(String&) constructor" << endl;
78:    // ConstructorCount++;
79: }
80:
81: // destructor, frees allocated memory
82: String::~String ()
83: {
84:    delete [] itsString;
85:    itsLen = 0;
86:    // cout << "\tString destructor" << endl;
87: }
88:
89: // operator equals, frees existing memory
90: // then copies string and size
91: String& String::operator=(const String & rhs)
92: {
93:    if (this == &rhs)
```

```
94:       return *this;
95:     delete [] itsString;
96:     itsLen=rhs.GetLen();
97:     itsString = new char[itsLen+1];
98:     for (int i = 0; i<itsLen;i++)
99:       itsString[i] = rhs[i];
100:    itsString[itsLen] = '\0';
101:    return *this;
102:    // cout << "\tString operator=" << endl;
103: }
104:
105: //non constant offset operator, returns
106: // reference to character so it can be
107: // changed!
108: char & String::operator[](int offset)
109: {
110:    if (offset > itsLen)
111:       return itsString[itsLen-1];
112:    else
113:       return itsString[offset];
114: }
115:
116: // constant offset operator for use
117: // on const objects (see copy constructor!)
118: char String::operator[](int offset) const
119: {
120:    if (offset > itsLen)
121:       return itsString[itsLen-1];
122:    else
123:       return itsString[offset];
124: }
125:
126: // creates a new string by adding current
127: // string to rhs
128: String String::operator+(const String& rhs)
129: {
130:    int  totalLen = itsLen + rhs.GetLen();
131:    String temp(totalLen);
132:    int i, j;
133:    for (i = 0; i<itsLen; i++)
134:       temp[i] = itsString[i];
135:    for (j = 0; j<rhs.GetLen(); j++, i++)
136:       temp[i] = rhs[j];
137:    temp[totalLen] = '\0';
138:    return temp;
139: }
140:
141: // changes current string, returns nothing
142: void String::operator+=(const String& rhs)
143: {
144:    unsigned short rhsLen = rhs.GetLen();
145:    unsigned short totalLen = itsLen + rhsLen;
```

```
146:    String temp(totalLen);
147:    int i, j;
148:    for (i = 0; i<itsLen; i++)
149:      temp[i] = itsString[i];
150:    for (j = 0, i = 0; j<rhs.GetLen(); j++, i++)
151:      temp[i] = rhs[j-itsLen];
152:    temp[totalLen] = '\0';
153:    *this = temp;
154: }
155:
156: // int String::ConstructorCount =
157: ostream& operator<< ( ostream& theStream,String& theString)
158: {
159:    theStream << theString.itsString;
160:    return theStream;
161: }
162:
163: int main()
164: {
165:    String theString("Hello world.");
166:    cout << theString;
167:    return 0;
168: }
```

输出：

```
Hello world.
```

分析：

在第 21～22 行，运算符 operator<<被声明为友元函数，它接受一个 ostream 引用和一个 String 引用作为参数并返回一个 ostream 引用。注意，它不是 String 的成员函数。它返回一个 ostream 引用，以便可以像下面这样拼接对 operator<<的调用：

```
cout << "myAge: " << itsAge << " years.";
```

该友元函数的实现位于第 157～161 行。其全部功能只是隐藏将字符串提供给 ostream 的实现细节（本应该如此）。第 17 章将更详细地介绍如何重载该运算符和 operator<<。

16.8 小　　结

本章介绍了如何将功能交给被聚合的对象；还介绍了如何通过聚合或私有继承，以一个类的方式实现另一个类。聚合的局限性在于，新类不能访问被聚合类的保护成员，也不能覆盖被聚合类的成员函数。聚合使用起来比私有继承更简单，因此应尽可能使用聚合。

读者学习了如何声明友元函数和友元类，还学习了如何使用友元函数来重载提取运算符，让新类能够像内置类那样使用 cout。

公有继承表示 is-a 关系，聚合表示 has-a 关系，私有继承表示“以...的方式实现”关系。代理关系可以用聚合或私有继承来表示，虽然聚合更常用。

16.9 问 与 答

问：为什么区分 is-a、has-a 和“以...的方式实现”非常重要？

答：C++的关键是实现精心设计、面向对象的程序。保留这些关系有助于确保设计符合要模拟的现实。另外，易于理解的设计被反映在精心设计的代码中的可能性更大。

问：什么是包含？

答：包含是聚合的同义词。

问：为什么聚合优于私有继承更好？

答：现代编程面临的挑战是处理复杂性。可作为黑箱使用的对象越多，需要考虑的细节就越少，可处理的复杂性越高。被聚合的类隐藏了它们的细节，而私有继承暴露了实现细节。从某种程度上说，这也适用于公有继承，有时候在聚合更佳的解决方案时却使用了公有继承。

问：为什么不将所有的类声明为它们使用的所有类的友元？

答：将类声明为另一个类的友元暴露了实现细节，降低封装程度。理想情形是，将尽可能多的类细节对其他所有类隐藏。

问：如果重载了一个函数，需要将该函数的每种形式都声明为友元吗？

答：是的。如果重载了一个函数并将其声明为另一个类的友元，必须将你希望具有这种访问权限的每种函数形式都声明为友元。

16.10　作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

16.10.1　测验

1. 如何建立 is-a 关系？
2. 如果建立 has-a 关系？
3. 聚合和代理之间有什么不同？
4. 代理和“以...的方式实现”之间有什么不同？
5. 什么是友元函数？
6. 什么是友元类？
7. 如果 Dog 是 Boy 的友元类，Boy 是否也是 Dog 的友元类？
8. 如果 Dog 是 Boy 的友元类，且 Terrier 从 Dog 派生而来，则 Terrier 是 Boy 的友元类吗？
9. 如果 Dog 是 Boy 的友元类，且 Boy 是 House 的友元类，则 Dog 是 House 的友元类吗？
10. 友元函数声明必须在什么地方？

16.10.2　练习

1. 声明 Animal 类，它将 String 对象作为数据成员。
2. 声明 BoundedArray 类，它是一个数组。
3. 声明以数组方式实现的 Set 类。
4. 修改程序清单 16.1，为 String 类提供提取运算符>>。
5. 查错：下面的程序有什么错误？

```
0:   Bug Busters
1:   #include <iostream>
2:   using namespace std;
3:   class Animal;
4:
5:   void setValue(Animal& , int);
6:
```

```
 7:  class Animal
 8:  {
 9:    public:
10:      int GetWeight()const { return itsWeight; }
11:      int GetAge() const { return itsAge; }
12:    private:
13:      int itsWeight;
14:      int itsAge;
15:  };
16:
17:  void setValue(Animal& theAnimal, int theWeight)
18:  {
19:    friend class Animal;
20:    theAnimal.itsWeight = theWeight;
21:  }
22:
23:  int main()
24:  {
25:    Animal peppy;
26:    setValue(peppy,5);
27:    return 0;
28:  }
```

6. 修改练习 5 中的代码，使其能够通过编译。
7. 查错：下面的代码有什么错误？

```
 0:  // Bug Busters
 1:  #include <iostream>
 2:  using namespace std;
 3:  class Animal;
 4:
 5:  void setValue(Animal& , int);
 6:  void setValue(Animal& ,int, int);
 7:
 8:  class Animal
 9:  {
10:     friend void setValue(Animal& ,int); // here's the change!
11:     private:
12:       int itsWeight;
13:       int itsAge;
14:  };
15:
16:  void setValue(Animal& theAnimal, int theWeight)
17:  {
18:     theAnimal.itsWeight = theWeight;
19:  }
20:
21:  void setValue(Animal& theAnimal, int theWeight, int theAge)
22:  {
23:    theAnimal.itsWeight = theWeight;
24:    theAnimal.itsAge = theAge;
25:  }
26:
27:  int main()
```

```
28:    {
29:       Animal peppy;
30:       setValue(peppy,5);
31:       setValue(peppy,7,9);
32:       return 0;
33:    }
```

8、修改练习 7 中的代码，使其能够通过编译。