第 3 周复习

下面的程序（程序清单 R3.1）使用了读者在过去 3 周学到的很多高级技术，它提供了一个基于模板的链表，其中包含异常处理功能。请仔细阅读该程序，如果读者能够完全看懂，就是一名 C++程序员了。

警告：如果你的编译器不支持模板或者不支持 try 和 catch，将无法编译或运行该程序。

程序清单 R3.1　第 3 周复习程序清单

```
 0: // ************************************************
 1: //
 2: // Title:    Week 3 in Review
 3: //
 4: // File:     Week3
 5: //
 6: // Description:  Provide a template-based linked list
 7: //               demonstration program with exception handling
 8: //
 9: // Classes:   PART - holds part numbers and potentially other
10: //                   information about parts. This will be the
11: //                   example class for the list to hold.
12: //                   Note use of operator<< to print the
13: //                   information about a part based on its
14: //                   runtime type.
15: //
16: //            Node - acts as a node in a List
17: //
18: //            List - template-based list that provides the
19: //                   mechanisms for a linked list
20: //
21: //
22: // Author:   Jesse Liberty (jl)
23: //
24: // Developed:  Pentium 200 Pro, 128MB RAM MVC 5.0
25: //
26: // Target:   Platform independent
27: //
28: // Rev History:  9/94 - First release (jl)
29: //               4/97 - Updated (jl)
30: //               9/04 - Updated (pij)
31: // ************************************************

CH 21
32: #include <iostream>
```

```
CH 18
33: using namespace std;
34:
35: // exception classes

CH 20
36: class Exception {};
37: class OutOfMemory :  public Exception{};
38: class NullNode :   public Exception{};
39: class EmptyList :  public Exception {};
40: class BoundsError :  public Exception {};
41:
42:
43: // ********** Part ***********
44: // Abstract base class of parts
45: class Part
46: {
47:   public:
48:     Part():itsObjectNumber(1) {}
49:     Part(int ObjectNumber):itsObjectNumber(ObjectNumber){}
50:     virtual ~Part(){};
51:     int GetObjectNumber() const { return itsObjectNumber; }
52:     virtual void Display() const =0;  // must be overridden
53:
54:   private:
55:     int itsObjectNumber;
56: };
57:
58: // implementation of pure virtual function so that
59: // derived classes can chain up
60: void Part::Display() const
61: {
62:    cout << "\nPart Number: " << itsObjectNumber << endl;
63: }
64:
65: // this one operator<< will be called for all part objects.
66: // It need not be a friend as it does not access private data
67: // It calls Display(), which uses the required polymorphism
68: // We'd like to be able to override this based on the real type
69: // of thePart, but C++ does not support contravariance

CH 17
70: ostream& operator<<( ostream& theStream,Part& thePart)
71: {
72:    thePart.Display();  // virtual contravariance!

CH 20
73:    return theStream;
74: }
75:
76: // ********** Car Part ***********
```

```
77: class CarPart : public Part
78: {
79:   public:
80:     CarPart():itsModelYear(94){}
81:     CarPart(int year, int partNumber);
82:     int GetModelYear() const { return itsModelYear; }
83:     virtual void Display() const;
84:   private:
85:     int itsModelYear;
86: };
87:
88: CarPart::CarPart(int year, int partNumber):
89:    itsModelYear(year),
90:    Part(partNumber)
91: {}
92:
93: void CarPart::Display() const
94: {
95:   Part::Display();
96:   cout << "Model Year: " << itsModelYear << endl;
97: }
98:
99: // *************** AirPlane Part ************
100: class AirPlanePart : public Part
101: {
102:   public:
103:     AirPlanePart():itsEngineNumber(1){};
104:     AirPlanePart(int EngineNumber, int PartNumber);
105:     virtual void Display() const;
106:     int GetEngineNumber()const { return itsEngineNumber; }
107:   private:
108:     int itsEngineNumber;
109: };
110:
111: AirPlanePart::AirPlanePart(int EngineNumber, int PartNumber):
112:    itsEngineNumber(EngineNumber),
113:    Part(PartNumber)
114: {}
115:
116: void AirPlanePart::Display() const
117: {
118:    Part::Display();
119:    cout << "Engine No.: " << itsEngineNumber << endl;
120: }
121:
122: // forward declaration of class List
123: template <class T>
124: class List;
125:
126: // *************** Node ************
127: // Generic node, can be added to a list
128: // ************************************
```

```
129:

CH  19
130: template <class T>
131: class Node
132: {
133:   public:
CH 16
134:     friend class List<T>;
135:     Node (T*);
136:     ~Node();
137:     void SetNext(Node * node) { itsNext = node; }
138:     Node * GetNext() const;

CH 19
139:     T * GetObject() const;
140:   private:
141:     T* itsObject;
142:       Node * itsNext;
143: };
144:
145: // Node Implementations...
146:

CH 19
147: template <class T>
148: Node<T>::Node(T* pObject):
149:   itsObject(pObject),
150:   itsNext(0)
151: {}
152:
153: template <class T>
154: Node<T>::~Node()
155: {
156:   delete itsObject;
157:   itsObject = 0;
158:   delete itsNext;
159:   itsNext = 0;
160: }
161:
162: // Returns NULL if no next Node
163: template <class T>
164: Node<T> * Node<T>::GetNext() const
165: {
166:   return itsNext;
167: }
168:

CH 19
169: template <class T>
170: T * Node<T>::GetObject() const
171: {
```

```
172:    if (itsObject)
173:      return itsObject;
174:    else
175:      throw NullNode();
176: }
177:
178: // *************** List ************
179: // Generic list template
180: // Works with any numbered object
181: // ***********************************
182: template <class T>
183: class List
184: {
185:   public:
186:     List();
187:     ~List();
188:

CH 19
189:     T*       Find(int & position, int ObjectNumber)  const;
190:     T*     GetFirst() const;
191:     void     Insert(T *);
192:     T*     operator[](int) const;
193:     int     GetCount() const { return itsCount; }
194:   private:

CH 19
195:     Node<T> * pHead;
196:     int     itsCount;
197: };
198:
199: // Implementations for Lists...
200: template <class T>
201: List<T>::List():
202:    pHead(0),
203:    itsCount(0)
204: {}
205:

CH 19
206: template <class T>
207: List<T>::~List()
208: {
209:    delete pHead;
210: }
211:
212: template <class T>
213: T*  List<T>::GetFirst() const
214: {
215:    if (pHead)
216:      return pHead->itsObject;
217:    else
```

```
218:       throw EmptyList();
219:  }
220:

CH 19
221:  template <class T>
222:  T *  List<T>::operator[](int offSet) const
223:  {
224:    Node<T>* pNode = pHead;
225:
226:    if (!pHead)
227:      throw EmptyList();
228:
229:    if (offSet > itsCount)
230:      throw BoundsError();
231:
232:    for (int i=0;i<offSet; i++)
233:      pNode = pNode->itsNext;
234:
235:    return  pNode->itsObject;
236:  }
237:
238:  // find a given object in list based on its unique number (id)

CH 19
239:  template <class T>
240:  T*  List<T>::Find(int & position, int ObjectNumber)  const
241:  {
242:    Node<T> * pNode = 0;
243:    for (pNode = pHead, position = 0;
244:        pNode!=NULL;
245:        pNode = pNode->itsNext, position++)
246:    {
247:      if (pNode->itsObject->GetObjectNumber() == ObjectNumber)
248:        break;
249:    }
250:    if (pNode == NULL)
251:      return NULL;
252:    else
253:      return pNode->itsObject;
254:  }
255:
256:  // insert if the number of the object is unique

CH 19
257:  template <class T>
258:  void List<T>::Insert(T* pObject)
259:  {
260:    Node<T> * pNode = new Node<T>(pObject);
261:    Node<T> * pCurrent = pHead;
262:    Node<T> * pNext = 0;
263:
```

```
264:    int New = pObject->GetObjectNumber();
265:    int Next = 0;
266:    itsCount++;
267:
268:    if (!pHead)
269:    {
270:      pHead = pNode;
271:      return;
272:    }
273:
274:    // if this one is smaller than head
275:    // this one is the new head
276:    if (pHead->itsObject->GetObjectNumber() > New)
277:    {
278:      pNode->itsNext = pHead;
279:      pHead = pNode;
280:      return;
281:    }
282:
283:    for (;;)
284:    {
285:      // if there is no next, append this new one
286:      if (!pCurrent->itsNext)
287:      {
288:        pCurrent->itsNext = pNode;
289:        return;
290:      }
291:
292:      // if this goes after this one and before the next
293:      // then insert it here, otherwise get the next
294:      pNext = pCurrent->itsNext;
295:      Next = pNext->itsObject->GetObjectNumber();
296:      if (Next > New)
297:      {
298:        pCurrent->itsNext = pNode;
299:        pNode->itsNext = pNext;
300:        return;
301:      }
302:      pCurrent = pNext;
303:    }
304: }
305:
306:
307: int main()
308: {
CH 19
309:    List<Part> theList;
310:    int choice = 99;
311:    int ObjectNumber;
312:    int value;
313:    Part * pPart;
314:    while (choice != 0)
```

```
315:    {
316:      cout << "(0)Quit (1)Car (2)Plane: ";
317:      cin >> choice;
318:
319:      if (choice != 0)
320:      {
321:
322:        cout << "New PartNumber?: ";
323:        cin >>  ObjectNumber;
324:
325:        if (choice == 1)
326:        {
327:          cout << "Model Year?: ";
328:          cin >> value;

CH 20
329:          try
330:          {
331:            pPart = new CarPart(value,ObjectNumber);
332:          }

CH 20
333:          catch (OutOfMemory)
334:          {
335:            cout << "Not enough memory; Exiting..." << endl;
336:            return 1;
337:          }
338:        }
339:        else
340:        {
341:          cout << "Engine Number?: ";
342:          cin >> value;

CH 20
343:          try
344:          {
345:            pPart = new AirPlanePart(value,ObjectNumber);
346:          }

CH 20
347:          catch (OutOfMemory)
348:          {
349:            cout << "Not enough memory; Exiting..." << endl;
350:            return 1;
351:          }
352:        }

CH 20
353:        try
354:        {
355:          theList.Insert(pPart);
356:        }
```

```
CH 20
357:          catch (NullNode)
358:          {
359:            cout << "The list is broken, and the node is null!" << endl;
360:            return 1;
361:          }

CH 20
362:          catch (EmptyList)
363:          {
364:            cout << "The list is empty!" << endl;
365:            return 1;
366:          }
367:       }
368:    }

CH 20
369:    try
370:    {
371:      for (int i = 0; i < theList.GetCount(); i++ )
372:        cout << *(theList[i]);
373:    }

CH 20
374:       catch (NullNode)
375:       {
376:         cout << "The list is broken, and the node is null!" << endl;
377:         return 1;
378:       }

CH 20
379:       catch (EmptyList)
380:       {
381:         cout << "The list is empty!" << endl;
382:         return 1;
383:       }

CH  20
384:       catch (BoundsError)
385:       {
386:         cout << "Tried to read beyond the end of the list!" << endl;
387:         return 1;
388:       }
389:    return 0;
390: }
```

输出：

```
  (0)Quit (1)Car (2)Plane: 1
New PartNumber?: 2837
Model Year? 90
```

```
  (0)Quit (1)Car (2)Plane: 2
New PartNumber?: 378
Engine Number?: 4938

  (0)Quit (1)Car (2)Plane: 1
New PartNumber?: 4499
Model Year? 94

  (0)Quit (1)Car (2)Plane: 1
New PartNumber?: 3000
Model Year? 93

  (0)Quit (1)Car (2)Plane: 0

Part Number: 378
Engine No. 4938

Part Number: 2837
Model Year: 90

Part Number: 3000
Model Year: 93

Part Number 4499
Model Year: 94
```

分析：

该程序清单对第 2 周复习中的程序进行了修改，添加了模板、ostream 处理和异常处理。输出结果是相同的。

第 36～40 行声明了一些异常类。在该程序提供的有点原始的异常处理中，不需要这些异常类包含数据或方法；它们用作 catch 语句的标记，该语句打印出一条简单的警告消息，然后退出。更健壮的程序可能按引用传递这些异常，然后在试图恢复故障时从异常对象中提取上下文信息或其他数据。

第 45 行声明的抽像基类 Part 与第 2 周复习中完全相同。这里惟一值得注意的变化是非类成员函数 operator<<( )，它是在第 70～74 行声明的。注意，它既不是 Part 的成员函数，也不是 Part 的友元函数，而只是将 Part 引用作为其参数之一。

你可能想让 operator<<接受一个 CarPart 参数和一个 AirPlanePart 参数，进而根据传递的是汽车零件还是飞机零件来调用正确的 operator<<。然而，由于程序传递一个指向零件的指针，而不是指向汽车零件或飞机零件的指针，因此 C++必须根据函数参数的实际类型调用正确的函数。这就是所谓的反变性（contravariance），C++不支持。

在 C++中实现多态的方式只有两种：函数多态和虚函数。函数多态在这里不管用，因为在每种情形下匹配的都是相同的特征标：将 Part 引用作为参数。

虚函数在这里也不管用，因为 operator<<不是 Part 的成员函数。你不能将 operator<<作为 Part 的成员函数，因为你想编写这样的代码：

```
cout << thePart
```

这意味着实际代码为 cout.operator<<(Part&)，而 cout 并没有将 Part 引用作为参数的 operator<<版本！

为避开这种限制，该程序只使用了一个 operator<<，它将 Part 引用作为参数。然后调用 Display( )，它是一个虚成员函数，从而调用了正确的版本。

在第 130～143 行，Node 被定义为模板。其功能与第 2 周复习程序中相同，但没有捆绑到 Part 对象中。

事实上，它可以是任何类型的对象的节点。

注意，如果试图获取 Node 中的对象，而其中又没有对象，则被视为异常，因此第 175 行引发异常。

在第 182 和 183 行，定义了一个 List 类模板。这个 List 类可存储任何有惟一标识号的对象的节点，并将它们升序排列。List 的每个函数都检查异常情形，并在必要时引发相应的异常。

在第 307 和 308 行，main()函数创建了两种类型的 Part 对象链表，然后使用标准流机制打印链表中对象的值。

FAQ

第 70 行前面的注释指出，C++不支持反变性。什么是反变性呢？

答：反变性是将基类指针赋给派生类指针的能力。

如果 C++支持反变性，可以在运行阶段根据对象的实际类型覆盖函数。程序清单 R3.2 不能编译，如果 C++支持反变性将能够编译。

警告：程序清单 R3.2 不能编译！

程序清单 R3.2　反变性

```
 0: #include <iostream>
 1: using namespace std;
 2: class Animal
 3: {
 4:   public:
 5:     virtual void Speak()
 6:       { cout << "Animal Speaks" << endl; }
 7: };
 8:
 9: class Dog : public Animal
10: {
11:   public:
12:     void Speak() { cout << "Dog Speaks" << endl; }
13: };
14:
15:
16: class Cat : public Animal
17: {
18:   public:
19:     void Speak() { cout << "Cat Speaks" << endl; }
20: };
21:
22: void DoIt(Cat*);
23: void DoIt(Dog*);
24:
25: int main()
26: {
27:    Animal * pA = new Dog;
28:    DoIt(pA);
29:    return 0;
30: }
31:
32: void DoIt(Cat * c)
33: {
```

```
34:    cout << "They passed a cat!" << endl << endl;
35:    c->Speak();
36: }
37:
38: void DoIt(Dog * d)
39: {
40:    cout << "They passed a dog!" << endl << endl;
41:    d->Speak();
42: }
```

当然，可以使用虚函数（如程序清单 R3.3 所示），这可以部分解决问题。

程序清单 R3.3　使用虚函数

```
 0: #include<iostream>
 1: using namespace std;
 2:
 3: class Animal
 4: {
 5:   public:
 6:     virtual void Speak() { cout << "Animal Speaks" << endl; }
 7: };
 8:
 9: class Dog : public Animal
10: {
11:   public:
12:     void Speak() { cout << "Dog Speaks" << endl; }
13: };
14:
15:
16: class Cat : public Animal
17: {
18:   public:
19:     void Speak() { cout << "Cat Speaks" << endl; }
20: };
21:
22: void DoIt(Animal*);
23:
24: int main()
25: {
26:
27:    Animal * pA = new Dog;
28:    DoIt(pA);
29:    return 0;
30: }
31:
32: void DoIt(Animal * c)
33: {
34:    cout << "They passed some kind of  animal" << endl << endl;
35:    c->Speak();
36: }
```