Template, Exceptions and Error Handling

Pertemuan 14

Outline

- Template, Exceptions and Error Handling
  - Instances of the Template
  - Using Template Items
  - How Exceptions Are Used

1. Instances of the Template

ParamList.cpp

```
1: // Demonstrates an object-oriented approach to parameterized
2: // linked lists. The list delegates to the node.
3: // The node is an abstract Object type. Three types of
4: // nodes are used, head nodes, tail nodes and internal
5: // nodes. Only the internal nodes hold Object.
6: //
7: // The Object class is created to serve as an object to
8: // hold in the linked list.
9: //
10: //*********************************************
11: #include <iostream>
12:
13: enum { kIsSmaller, kIsLarger, kIsSame};
14:
15: // Object class to put into the linked list
16: // Any class in this linked list must support two member
17: // functions: show (displays the value) and
18: // compare (returns relative position)
19: class Data
20: {
21:        public:
22:                   Data(int newVal):value(newVal) {}
23:                   ~Data()
24:                   {
25:                             std::cout << "Deleting Data object with value: ";
26:                             std::cout << value << "\n";
27:                   }
28:                   int compare(const Data&);
29:                   void show() { std::cout << value << "\n"; }
30:        private:
31:                   int value;
32: };
33:
```

...

```
34: // compare is used to decide where in the list
35: // a particular object belongs.
36: int Data::compare(const Data& otherObject)
37: {
38:         if (value < otherObject.value)
39:                     return kIsSmaller;
40:         if (value > otherObject.value)
41:                     return kIsLarger;
42:         else
43:                     return kIsSame;
44: }
45:
46: // Another class to put into the linked list
47: // Again, every class in this linked
48: // list must support two member functions:
49: // Show (displays the value) and
50: // Compare (returns relative position)
51: class Cat
52: {
53:         public:
54:                     Cat(int newAge): age(newAge) {}
55:                     ~Cat()
56:                     {
57:                                 std::cout << "Deleting ";
58:                                 std::cout << age << " year old Cat.\n";
59:                     }
60:                     int compare(const Cat&);
61:                     void show()
62:                     {
63:                                 std::cout << "This cat is ";
64:                                 std::cout << age << " years old\n";
65:                     }
66:         private:
67:                     int age;
68: };
69:
```

...

```
70: // compare is used to decide where in the list
71: // a particular object belongs.
72: int Cat::compare(const Cat& otherCat)
73: {
74:         if (age < otherCat.age)
75:                   return kIsSmaller;
76:         if (age > otherCat.age)
77:                   return kIsLarger;
78:         else
79:                   return kIsSame;
80: }
81:
82: // ADT representing the node object in the list
83: // Every derived class must override insert and show
84: template <class T>
85: class Node
86: {
87:         public:
88:                   Node(){}
89:                   virtual ~Node() {}
90:                   virtual Node* insert(T* object) = 0;
91:                   virtual void show() = 0;
92:         private:
93: };
94:
```

...

```
95: template <class T>
96: class InternalNode: public Node<T>
97: {
98:         public:
99:                     InternalNode(T* object, Node<T>* next);
100:                    ~InternalNode(){ delete next; delete object; }
101:                    virtual Node<T> * insert(T * object);
102:                    virtual void show()
103:                    {
104:                              object->show();
105:                              next->show();
106:                    } // delegate!
107:        private:
108:                    T* object; // the Object itself
109:                    Node<T>* next; // points to next node in the linked list
110: };
111:
112: // All the constructor does is initialize
113: template <class T>
114: InternalNode<T>::InternalNode(T* newObject, Node<T>* newNext):
115: object(newObject),next(newNext)
116: {
117: }
118:
```

...

```
119: // the meat of the list
120: // When you put a new object into the list
121: // it is passed to the node which figures out
122: // where it goes and inserts it into the list
123: template <class T>
124: Node<T>* InternalNode<T>::insert(T* newObject)
125: {
126:       // is the new guy bigger or smaller than me?
127:       int result = object->compare(*newObject);
128:
129:       switch(result)
130:       {
131:                 // by convention if it is the same as me it comes first
132:                 case kIsSame: // fall through
133:                 case kIsLarger: // new object comes before me
134:                 {
135:                           InternalNode<T>* objectNode =
136:                           new InternalNode<T>(newObject, this);
137:                           return objectNode;
138:                 }
139:                 // it is bigger than I am so pass it on to the next
140:                 // node and let HIM handle it.
141:                 case kIsSmaller:
142:                 next = next->insert(newObject);
143:                 return this;
144:       }
145:       return this; // appease MSC
146: }
147:
```

...

```
148: // Tail node is just a sentinel
149: template <class T>
150: class TailNode : public Node<T>
151: {
152:       public:
153:                   TailNode() {}
154:                   virtual ~TailNode() {}
155:                   virtual Node<T>* insert(T * object);
156:                   virtual void show() { }
157:       private:
158: };
159:
160: // If object comes to me, it must be inserted before me
161: // as I am the tail and NOTHING comes after me
162: template <class T>
163: Node<T>* TailNode<T>::insert(T * object)
164: {
165:       InternalNode<T>* objectNode =
166:       new InternalNode<T>(object, this);
167:       return objectNode;
168: }
169:
170: // Head node has no Object, it just points
171: // to the very beginning of the list
172: template <class T>
173: class HeadNode : public Node<T>
174: {
175: public:
176: HeadNode();
177:       virtual ~HeadNode() { delete next; }
178:       virtual Node<T>* insert(T * object);
179:       virtual void show() { next->show(); }
180: private:
181:       Node<T> * next;
182: };
183:
```

...

```
184: // As soon as the head is created
185: // it creates the tail
186: template <class T>
187: HeadNode<T>::HeadNode()
188: {
189:       next = new TailNode<T>;
190: }
191:
192: // Nothing comes before the head so just
193: // pass the Object on to the next node
194: template <class T>
195: Node<T> * HeadNode<T>::insert(T* object)
196: {
197:       next = next->insert(object);
198:       return this;
199: }
200:
201: // I get all the credit and do none of the work
202: template <class T>
203: class LinkedList
204: {
205:       public:
206:                  LinkedList();
207:                  ~LinkedList() { delete head; }
208:                  void insert(T* object);
209:                  void showAll() { head->show(); }
210:       private:
211:                  HeadNode<T> * head;
212: };
213:
```

...

```
214: // At birth, I create the head node
215: // It creates the tail node
216: // So an empty list points to the head which
217: // points to the tail and has nothing between
218: template <class T>
219: LinkedList<T>::LinkedList()
220: {
221:       head = new HeadNode<T>;
222: }
223:
224: // Delegate, delegate, delegate
225: template <class T>
226: void LinkedList<T>::insert(T* pObject)
227: {
228:       head->insert(pObject);
229: }
230:
```

...

```
231: // test driver program
232: int main()
233: {
234:       Cat* pCat;
235:       Data* pData;
236:       int val;
237:       LinkedList<Cat> listOfCats;
238:       LinkedList<Data> listOfData;
239:
240:       // ask the user to produce some values
241:       // put them in the list
242:       while (true)
243:       {
244:                 std::cout << "What value (0 to stop)? ";
245:                 std::cin >> val;
246:                 if (!val)
247:                 break;
248:                 pCat = new Cat(val);
249:                 pData = new Data(val);
250:                 listOfCats.insert(pCat);
251:                 listOfData.insert(pData);
252:       }
253:
254:       // now walk the list and show the Object
255:       std::cout << "\n";
256:       listOfCats.showAll();
257:       std::cout << "\n";
258:       listOfData.showAll();
259:       std::cout << "\n************\n\n";
260:       return 0; // The lists fall out of scope and are destroyed!
261: }
```

2. Using Template Items

…

```
1: #include <iostream>
2:
3: enum { kIsSmaller, kIsLarger, kIsSame};
4:
5: class Data
6: {
7:         public:
8:                     Data(int newVal):value(newVal) {}
9:                     ~Data()
10:                    {
11:                              std::cout << "Deleting Data object with value: ";
12:                              std::cout << value << "\n";
13:                    }
14:                    int compare(const Data&);
15:                    void show() { std::cout << value << "\n"; }
16:                    private:
17:                    int value;
18: };
19:
20: int Data::compare(const Data& otherObject)
21: {
22:         if (value < otherObject.value)
23:                    return kIsSmaller;
24:         if (value > otherObject.value)
25:                    return kIsLarger;
26:         else
27:                    return kIsSame;
28: }
29:
```

...

```
30: class Cat
31: {
32:         public:
33:                     Cat(int newAge): age(newAge) {}
34:                     ~Cat()
35:                     {
36:                               std::cout << “Deleting “ << age
37:                               << “ year old Cat.\n”;
38:                     }
39:                     int compare(const Cat&);
40:                     void show()
41:                     {
42:                               std::cout << “This cat is “ << age
43:                               << “ years old\n”;
44:                     }
45:         private:
46:                     int age;
47: };
48:
49: int Cat::compare(const Cat& otherCat)
50: {
51:         if (age < otherCat.age)
52:                     return kIsSmaller;
53:         if (age > otherCat.age)
54:                     return kIsLarger;
55:         else
56:                     return kIsSame;
57: }
58:
```

...

```
59: template <class T>
60: class Node
61: {
62:         public:
63:                     Node() {}
64:                     virtual ~Node() {}
65:                     virtual Node* insert(T* object) = 0;
66:                     virtual void show() = 0;
67:         private:
68: };
69:
70: template <class T>
71: class InternalNode: public Node<T>
72: {
73:         public:
74:                     InternalNode(T* theObject, Node<T>* next);
75:                     virtual ~InternalNode(){ delete next; delete object; }
76:                     virtual Node<T>* insert(T* object);
77:                     virtual void show()
78:                     {
79:                                 object->show();
80:                                 next->show();
81:                     }
82:         private:
83:                     T* object;
84:                     Node<T>* next;
85: };
86:
87: template <class T>
88: InternalNode<T>::InternalNode(T* newObject, Node<T>* newNext):
89: object(newObject), next(newNext)
90: {
91: }
92:
```

...

```
93: template <class T>
94: Node<T>* InternalNode<T>::insert(T* newObject)
95: {
96:         int result = object->compare(*newObject);
97:
98:         switch(result)
99:         {
100:                 case kIsSame:
101:                 case kIsLarger:
102:                 {
103:                         InternalNode<T> * objectNode =
104:                         new InternalNode<T>(newObject, this);
105:                         return objectNode;
106:                 }
107:                 case kIsSmaller:
108:                 next = next->insert(newObject);
109:                 return this;
110:         }
111:         return this;
112: }
113:
114: template <class T>
115: class TailNode : public Node<T>
116: {
117:         public:
118:                 TailNode() {}
119:                 virtual ~TailNode() {}
120:                 virtual Node<T>* insert(T* object);
121:                 virtual void show() {}
122:         private:
123: };
123:
```

...

```
125: template <class T>
126: Node<T>* TailNode<T>::insert(T* object)
127: {
128:       InternalNode<T>* objectNode =
129:       new InternalNode<T>(object, this);
130:       return objectNode;
131: }
132:
133: template <class T>
134: class HeadNode : public Node<T>
135: {
136:       public:
137:                  HeadNode();
138:                  virtual ~HeadNode() { delete next; }
139:                  virtual Node<T>* insert(T* object);
140:                  virtual void show() { next->show(); }
141:       private:
142:                  Node<T>* next;
143: };
144:
145: template <class T>
146: HeadNode<T>::HeadNode()
147: {
148:       next = new TailNode<T>;
149: }
150:
151: template <class T>
152: Node<T>* HeadNode<T>::insert(T* object)
153: {
154:       next = next->insert(object);
155:       return this;
156: }
157:
```

...

```
158: template <class T>
159: class LinkedList
160: {
161:       public:
162:                  LinkedList();
163:                  ~LinkedList() { delete head; }
164:                  void insert(T* object);
165:                  void showAll() { head->show(); }
166:       private:
167:                  HeadNode<T>* head;
168: };
169:
170: template <class T>
171: LinkedList<T>::LinkedList()
172: {
173:       head = new HeadNode<T>;
174: }
175:
176: template <class T>
177: void LinkedList<T>::insert(T* pObject)
178: {
179:       head->insert(pObject);
180: }
181:
182: void myFunction(LinkedList<Cat>& listOfCats);
183: void myOtherFunction(LinkedList<Data>& listOfData);
184:
```

...

```
185: int main()
186: {
187:       LinkedList<Cat> listOfCats;
188:       LinkedList<Data> listOfData;
189:
190:       myFunction(listOfCats);
191:       myOtherFunction(listOfData);
192:
193:       std::cout << "\n";
194:       listOfCats.showAll();
195:       std::cout << "\n";
196:       listOfData.showAll();
197:       std::cout << "\n************\n\n";
198:       return 0;
199: }
200:
201: void myFunction(LinkedList<Cat>& listOfCats)
202: {
203:       Cat* pCat;
204:       int val;
205:
206:       while (true)
207:       {
208:                 std::cout << "\nHow old is your cat (0 to stop)? ";
209:                 std::cin >> val;
210:                 if (!val)
211:                 break;
212:                 pCat = new Cat(val);
213:                 listOfCats.insert(pCat);
214:       }
215: }
216:
```

...

```
217: void myOtherFunction(LinkedList<Data>& listOfData)
218: {
219:       Data* pData;
220:       int val;
221:
222:       while (true)
223:       {
224:                 std::cout << "\nWhat value (0 to stop)? ";
225:                 std::cin >> val;
226:                 if (!val)
227:                           break;
228:                 pData = new Data(val);
229:                 listOfData.insert(pData);
230:       }
231: }
```

3. How Exceptions Are Used

Exception.cpp

```
1: #include <iostream>
2:
3: const int defaultSize = 10;
4:
5: // define the exception class
6: class XBoundary
7: {
8:          public:
9:                       XBoundary() {}
10:                      ~XBoundary() {}
11:                      private:
12: };
13:
14: class Array
15: {
16:         public:
17:                      // constructors
18:                      Array(int size = defaultSize);
19:                      Array(const Array &rhs);
20:                      ~Array() { delete [] pType; }
21:
22:                      // operators
23:                      Array& operator=(const Array&);
24:                      int& operator[](int offSet);
25:                      const int& operator[](int offSet) const;
26:
27:                      // accessors
28:                      int getSize() const { return size; }
29:
30:                      // friend function
31:                      friend std::ostream& operator<<(std::ostream&, const Array&);
32:
33:                      private:
34:                      int *pType;
35:                      int size;
36: };
37:
```

...

```
38: Array::Array(int newSize):
39: size(newSize)
40: {
41:         pType = new int[size];
42:         for (int i = 0; i < size; i++)
43:                     pType[i] = 0;
44: }
45:
46: Array& Array::operator=(const Array &rhs)
47: {
48:         if (this == &rhs)
49:                     return *this;
50:         delete [] pType;
51:         size = rhs.getSize();
52:         pType = new int[size];
53:         for (int i = 0; i < size; i++)
54:                     pType[i] = rhs[i];
55:         return *this;
56: }
57:
58: Array::Array(const Array &rhs)
59: {
60:         size = rhs.getSize();
61:         pType = new int[size];
62:         for (int i = 0; i < size; i++)
63:                     pType[i] = rhs[i];
64: }
65:
66: int& Array::operator[](int offSet)
67: {
68:         int size = getSize();
69:         if (offSet >= 0 && offSet < size)
70:                     return pType[offSet];
71:         throw XBoundary();
72:         return pType[offSet];
73: }
74:
```

...

```
75: const int& Array::operator[](int offSet) const
76: {
77:         int size = getSize();
78:         if (offSet >= 0 && offSet < size)
79:                     return pType[offSet];
80:         throw XBoundary();
81:         return pType[offSet];
82: }
83:
84: std::ostream& operator<<(std::ostream& output,
85: const Array& array)
86: {
87:         for (int i = 0; i < array.getSize(); i++)
88:                     output << "[" << i << "] " << array[i] << "\n";
89:         return output;
90: }
91:
92: int main()
93: {
94:         Array intArray(20);
95:         try
96:         {
97:                     for (int j = 0; j < 100; j++)
98:                     {
99:                                 intArray[j] = j;
100:                                std::cout << "intArray[" << j
101:                                << "] OK ..." << "\n";
102:                    }
103:        }
104:        catch (XBoundary)
105:        {
106:                    std::cout << "Unable to process your input\n";
107:        }
108:        std::cout << "Done\n";
109:        return 0;
110: }
```

4. Catching by Reference and Polymorphism

PolyException.cpp

```
1: #include <iostream>
2:
3: const int defaultSize = 10;
4:
5: // define the exception classes
6: class XBoundary {};
7:
8: class XSize
9: {
10:         public:
11:                     XSize(int newSize):size(newSize) {}
12:                     ~XSize(){}
13:                     virtual int getSize() { return size; }
14:                     virtual void printError()
15:                     { std::cout << "Size error. Received: "
16:                     << size << "\n"; }
17:         protected:
18:                     int size;
19: };
20:
21: class XTooBig : public XSize
22: {
23:         public:
24:                     XTooBig(int size):XSize(size) {}
25:                     virtual void printError()
26:                     {
27:                                 std::cout << "Too big! Received: ";
28:                                 std::cout << XSize::size << "\n";
29:                     }
30: };
31:
```

...

```
32: class XTooSmall : public XSize
33: {
34:         public:
35:                     XTooSmall(int size):XSize(size) {}
36:                     virtual void printError()
37:                     {
38:                               std::cout << “Too small! Received: “;
39:                               std::cout << XSize::size << “\n”;
40:                     }
41: };
42:
43: class XZero : public XTooSmall
44: {
45:         public:
46:                     XZero(int newSize):XTooSmall(newSize){}
47:                     virtual void printError()
48:                     {
49:                               std::cout << “Zero Received: “;
50:                               std::cout << XSize::size << “\n”;
51:                     }
52: };
53:
54: class XNegative : public XSize
55: {
56:         public:
57:                     XNegative(int size):XSize(size){}
58:                     virtual void printError()
59:                     {
60:                               std::cout << “Negative! Received: “;
61:                               std::cout << XSize::size << “\n”;
62:                     }
63: };
64:
```

...

```
65: class Array
66: {
67:       public:
68:                 // constructors
69:                 Array(int size = defaultSize);
70:                 Array(const Array &rhs);
71:                 ~Array() { delete [] pType; }
72:
73:                 // operators
74:                 Array& operator=(const Array&);
75:                 int& operator[](int offSet);
76:                 const int& operator[](int offSet) const;
77:
78:                 // accessors
79:                 int getSize() const { return size; }
80:
81:                 // friend function
82:                 friend std::ostream& operator<< (std::ostream&, const Array&);
83:
84:
85:       private:
86:                 int *pType;
87:                 int size;
88: };
89:
```

...

```
90: Array::Array(int newSize):
91: size(newSize)
92: {
93:         if (newSize == 0)
94:                     throw XZero(size);
95:
96:         if (newSize < 0)
97:                     throw XNegative(size);
98:
99:         if (newSize < 10)
100:                    throw XTooSmall(size);
101:
102:        if (newSize > 30000)
103:                    throw XTooBig(size);
104:
105:        pType = new int[newSize];
106:        for (int i = 0; i < newSize; i++)
107:                    pType[i] = 0;
108: }
109:
110: int& Array::operator[] (int offset)
111: {
112:        int size = getSize();
113:        if (offset >= 0 && offset < size)
114:                    return pType[offset];
115:        throw XBoundary();
116:        return pType[offset];
117: }
118:
119: const int& Array::operator[] (int offset) const
120: {
121:        int size = getSize();
122:        if (offset >= 0 && offset < size)
123:                    return pType[offset];
124:        throw XBoundary();
125:        return pType[offset];
126: }
127:
```

...

```
128: int main()
129: {
130:       try
131:       {
132:                 int choice;
133:                 std::cout << "Enter the array size: ";
134:                 std::cin >> choice;
135:                 Array intArray(choice);
136:                 for (int j = 0; j < 100; j++)
137:                 {
138:                           intArray[j] = j;
139:                           std::cout << "intArray[" << j << "] OK ..."
140:                           << "\n";
141:                 }
142:       }
143:       catch (XBoundary)
144:       {
145:                 std::cout << "Unable to process your input\n";
146:       }
147:       catch (XSize& exception)
148:       {
149:                 exception.printError();
150:       }
151:       catch (...)
152:       {
153:                 std::cout << "Something went wrong,"
154:                 << "but I've no idea what!" << "\n";
155:       }
156:       std::cout << "Done\n";
157:       return 0;
158: }
```

Tugas

- Bandingkan program ParamList.cpp dengan program LinkedList.cpp dan tuliskan perbedaannya.
- Lakukan extend program ParamList.cpp dengan menambahkan class Dog. Simpan dalam sebuah linked list dan perlakukan seperti ha;nya object Cat pada fungsi main().
- Cobalah mengecek semua program yang telah Anda kerjakan, carilah program yang berpotensi terjadi error. Berikan fungsi try dan catch pada program tersebut.