

```

1  #include "treasurechest.h"
2
3  void TreasureChest::addItem (const Item& newItem)
4  {
5      this->chest.push_back( newItem );
6  }
7
8  /*
9   * Insert an item at the specified zero-indexed position in the chest.
10  * If position is not valid for the chest, add the item to
11  * the end of the chest.
12  * @param newItem the item to be inserted into the chest
13  * @param position the zero-indexed position where the insertion
14  *       is to take place
15  */
16  void TreasureChest::insertItem (const Item& newItem, unsigned int position)
17  {
18      if( position < chest.size() )
19      {
20          vector<Item>::iterator it = this->chest.begin();
21          this->chest.insert( (it + position), newItem );
22      }
23      else
24      {
25          this->addItem( newItem );
26      }
27  }
28
29  /*
30  * Get a pointer to an item at a specified zero-indexed position in the chest.
31  * @param position the zero-indexed position of the item
32  * @return a pointer to the item if position is valid, else NULL
33  */
34  const Item* TreasureChest::getItem (unsigned int position)
35  {
36      if( position < chest.size() )
37      {
38          return &this->chest.at( position );
39      }
40      else
41      {
42          return NULL;
43      }
44  }
45
46  /*
47  * Remove an item from the chest at a specified zero-indexed position.
48  * @param position the zero-indexed position of the item
49  * @return a copy of the Item removed from the chest
50  * @throw string("ERROR: attempting remove at invalid position") if

```

```

51     *           position is not valid
52     */
53 Item TreasureChest::removeItem (unsigned int position) throw (string)
54 {
55     if( position < chest.size() )
56     {
57         vector<Item>::iterator it    = this->chest.begin();
58         Item objRemovedItem         = chest.at( position );
59         this->chest.erase( (it + position) );
60
61         return objRemovedItem;
62     }
63     else
64     {
65         throw string( "ERROR: attempting remove at invalid position" );
66     }
67 }
68
69 /*
70  * Clear the chest of all items.
71  */
72 void TreasureChest::clear ()
73 {
74     this->chest.clear();
75 }
76
77 /*
78  * Check to see if the chest is empty.
79  * @return true if the chest is empty, else false
80  */
81 bool TreasureChest::empty () const
82 {
83     if( this->chest.empty() )
84     {
85         return true;
86     }
87
88     return false;
89 }
90
91 /*
92  * Get the size/number of items currently in the chest.
93  * @return an unsigned integer containing the current size of the chest
94  */
95 unsigned int TreasureChest::getSize () const
96 {
97     return this->chest.size();
98 }
99
100 /*

```

```

101  * Sort the items in the chest by name, using the sort function
102  * from the C++ standard algorithm library.
103  */
104  void TreasureChest::sortByName ()
105  {
106      sort( this->chest.begin(), this->chest.end(), compareItemsByName );
107  }
108
109  /*
110  * Sort the items in the chest by value, using the sort function
111  * from the C++ standard algorithm library.
112  */
113  void TreasureChest::sortByValue ()
114  {
115      sort( this->chest.begin(), this->chest.end(), compareItemsByValue );
116  }
117
118  /*
119  * Sort the items in the chest by quantity, using the sort function
120  * from the C++ standard algorithm library.
121  */
122  void TreasureChest::sortByQuantity ()
123  {
124      sort( this->chest.begin(), this->chest.end(), compareItemsByQuantity );
125  }
126
127  /*
128  * Place the names of the items in the chest on the specified stream,
129  * formatted as ITEM_NAME,ITEM_NAME,...ITEM_NAME
130  */
131  ostream& operator<< (ostream& outs, const TreasureChest& src)
132  {
133      for( vector<Item>::const_iterator p = src.chest.begin(); p < src.chest.end(); p++ )
134      {
135          outs << *p;
136          if( (p + 1) != src.chest.end() )
137          {
138              outs << ",";
139          }
140      }
141
142      return outs;
143  }
144
145  /*
146  * Compare two items by name.
147  * @return true if lsrc.name < rsrc.name, else false
148  */
149  bool compareItemsByName (const Item& lsrc, const Item& rsrc)
150  {

```

```
151     return ( lsrc.name < rsrc.name );
152 }
153
154 /*
155  * Compare two items by value.
156  * @return true if lsrc.value < rsrc.value, else false
157  */
158 bool compareItemsByValue (const Item& lsrc, const Item& rsrc)
159 {
160     return ( lsrc.value < rsrc.value );
161 }
162
163 /*
164  * Compare two items by quantity.
165  * @return true if lsrc.quantity < rsrc.quantity, else false
166  */
167 bool compareItemsByQuantity (const Item& lsrc, const Item& rsrc)
168 {
169     return ( lsrc.quantity < rsrc.quantity );
170 }
171
```