```
1  // Illustrate the smart pointer approach using Templates
2  // George F. Riley, Georgia Tech, Spring 2012
3
4  #include <iostream>
5  #include <vector>
6
7  using namespace std;
8
9  // The Ptr class contains two members, one a pointer to a generic
10 // template type "T", and the int pointer for the reference count
11
12 template <typename T>
13 class Ptr {
14 public:
15   Ptr(T* t0);
16   // We need a copy constructor, assignment operator, and destructor
17   Ptr(const Ptr&);                // Copy constructor
18   Ptr& operator=(const Ptr& rhs); // Assignment operator
19   ~Ptr();                         // Destructor
20
21   // And we need a de-referencing operator, which should return
22   // the dereferenced "t" pointer;
23   const T& operator*() const;     // De-referencing operator
24   T& operator*();                 // Non-const version
25   // And the "arrow" operator.  Need both const and non-const
26   T* operator->();
27   const T* operator->() const;
28 private:
29   // Members are private
30   T* t;
31   int* refCount;
32 };
33
34 // Ptr Implementations
35 template <typename T>
36 Ptr<T>::Ptr(T* t0)
37 : t(t0)
38 { // Constructor
39   refCount = new int; // Create the refcount variable
40   *refCount = 1;      // Initially only one reference (this one).
41 };
42
43 // Copy Constructor
44 template<typename T>
45 Ptr<T>::Ptr(const Ptr<T>& rhs)
46 : refCount(rhs.refCount), t(rhs.t)
47 {
48   // Increment the refCount
49   (*refCount)++;
50 }
51
52 // Assignment operator
53 template <typename T>
54 Ptr<T>& Ptr<T>::operator=(const Ptr<T>& rhs)
55 {
56   // Protect against self assignment
```

Program template-smart-pointers.cc

```
57    if (&rhs == this) return *this;
58    // First reduce refcount in lhs and free if needed
59    (*refCount)--;
60    if (*refCount == 0)
61      { // No remaining references
62        delete t;
63        delete refCount;
64      }
65    refCount = rhs.refCount;
66    t = rhs.t;
67    (*refCount)++; // Another reference
68    return *this;
69  }
70
71  // Destructor
72  template<typename T>
73  Ptr<T>::~Ptr<T>()
74  {
75    (*refCount)--;
76    if (*refCount == 0)
77      { // No remaining references
78        delete t;
79        delete refCount;
80      }
81  }
82
83  // Dereferencing operators
84  template <typename T>
85  const T& Ptr<T>::operator*() const
86  {
87    return *t;
88  }
89
90  template <typename T>
91  T& Ptr<T>::operator*()
92  {
93    return *t;
94  }
95
96  // Arrow operators
97  template <typename T>
98  const T* Ptr<T>::operator->() const
99  {
100   return t;
101 }
102
103 template <typename T>
104 T* Ptr<T>::operator->()
105 {
106   return t;
107 }
```

Program template-smart-pointers.cc (continued)

```
108  // Implement the create functions
109
110  template <typename T>
111  Ptr<T> Create(void); // Constructor with no arguments
112  template <typename T, typename T1>
113  Ptr<T> Create(T1); // With one argument
114  template <typename T, typename T1, typename T2>
115  Ptr<T> Create(T1, T2); // With two arguments
116  template <typename T, typename T1, typename T2, typename T3>
117  Ptr<T> Create(T1, T2, T3); // With three arguments
118    // Add additional "Create" with more arguments as needed
119
120
121  template<typename T>
122  Ptr<T> Create(void)
123  {
124    T* t0 = new T; // Constructor with no arguments
125    return Ptr<T>(t0);
126  }
127
128  template<typename T, typename T1>
129  Ptr<T> Create(T1 a1)
130  {
131    T* t0 = new T(a1); // Constructor with one argument
132    return Ptr<T>(t0);
133  }
134
135  template<typename T, typename T1, typename T2>
136  Ptr<T> Create(T1 a1, T2 a2)
137  {
138    T* t0 = new T(a1, a2); // Constructor with two arguments
139    return Ptr<T>(t0);
140  }
141
142  template<typename T, typename T1, typename T2, typename T3>
143  Ptr<T> Create(T1 a1, T2 a2, T3 a3)
144  {
145    T* t0 = new T(a1, a2, a3); // Constructor with three arguments
146    return Ptr<T>(t0);
147  }
148
149  // Now create classes A and B that we will use to illustrate the smart
150  // pointers.
151
152  class A
153  {
154  public:
155    // constructors and destructors
156    A();
157    A(int);
158    A(const A&);
159    ~A();
160    void Hello() const;
161  public:
162    int a;
163  public:
```

Program template-smart-pointers.cc (continued)

```
164   static int constructorCount; // Counts total number of constructors
165   static int destructorCount;  // Counts total number of destructors
166 };
167
168 class B
169 {
170 public:
171   B();
172   B(int, int, int); // three args
173   B(const B&);
174   ~B();
175   void Hello() const;
176 public:
177   int b0;
178   int b1;
179   int b2;
180 public:
181   static int constructorCount; // Counts total number of constructors
182   static int destructorCount;  // Counts total number of destructors
183 };
184
185 // Implement A and B
186 A::A()
187   : a(0)
188 {
189   constructorCount++;
190 };
191
192 A::A(int a0)
193   : a(a0)
194 {
195   constructorCount++;
196 };
197
198 A::A(const A& rhs)
199   : a(rhs.a)
200 {
201   constructorCount++;
202 };
203
204 A::~A()
205 { // destructor
206   destructorCount++;
207 };
208
209 void A::Hello() const
210 {
211   cout << "Hello from A, a is " << a << endl;
212 }
213
214
215 B::B()
216   : b0(0), b1(0), b2(0)
217 {
218   constructorCount++;
219 };
```

Program template-smart-pointers.cc (continued)

```
220
221 B::B(int b00, int b01, int b02)
222   : b0(b00), b1(b01), b2(b02)
223 {
224   constructorCount++;
225 };
226
227 B::B(const B& rhs)
228   : b0(rhs.b0), b1(rhs.b1), b2(rhs.b2)
229 {
230   constructorCount++;
231 };
232
233 B::~B()
234 { // destructor
235   destructorCount++;
236 };
237
238 void B::Hello() const
239 {
240   cout << "Hello from B, b0 is " << b0 << endl;
241 }
242
243 // Helper functions
244 void PrintCounts()
245 {
246   cout << "A constructor " << A::constructorCount
247        << " A destructor " << A::destructorCount << endl;
248   cout << "B constructor " << B::constructorCount
249        << " B destructor " << B::destructorCount << endl;
250 };
251
252 // Define the A and B static variables
253 int A::constructorCount = 0;
254 int A::destructorCount = 0;
255 int B::constructorCount = 0;
256 int B::destructorCount = 0;
257
258 void Test1()
259 {
260   // Just create a single A and B (with Create) and do not delete
261   Ptr<A> pA = Create<A>(10);
262   Ptr<B> pB = Create<B>(10, 20, 30);
263   // just exit; smart pointer semantics call destructors automatically
264 }
265
266 void Test2()
267 {
268   // Just create a single A and B (with Create), then use Ptr copy constructor
269   // to create two more; no deletes
270   Ptr<A> pA = Create<A>(10);
271   Ptr<A> pACopy(pA);
272   Ptr<B> pB = Create<B>(10, 20, 30);
273   Ptr<B> pBCopy(pB);
274   // just exit; smart pointer semantics call destructors automatically
275 }
```

Program template-smart-pointers.cc (continued)

```
276
277 void Test3Helper(Ptr<A> aByValue,
278                  Ptr<B> bByValue)
279 {
280   // Also illustrates the dereferencing operator
281   cout << "Hello from Test3Helper a is " << (*aByValue).a
282        << " b0 is " << (*bByValue).b0 << endl;
283   // Again, using arrow operator
284   cout << "Hello agin from Test3Helper a is " << aByValue->a
285        << " b0 is " << bByValue->b0 << endl;
286
287 }
288
289 void Test3()
290 {
291   // Create an A and B, pass by value to a function
292   Ptr<A> pA = Create<A>(10);
293   Ptr<B> pB = Create<B>(10, 20, 30);
294   Test3Helper(pA, pB);
295 }
296
297 void Test4Helper(const Ptr<A>& a, // by const reference
298                  const Ptr<B>& b)
299 {
300   // Illustrate calling A and B member functions using
301   // arrow operator
302   a->Hello();
303   b->Hello();
304 }
305
306 void Test4()
307 {
308   // Create an A and B, pass by const reference to a function
309   Ptr<A> pA = Create<A>(10);
310   Ptr<B> pB = Create<B>(10, 20, 30);
311   Test4Helper(pA, pB);
312 }
313
314 void Test5()
315 {
316   // Create an two A and two B Ptr objects, then use assignment
317   // operator.
318   Ptr<A> pA = Create<A>(10);
319   Ptr<B> pB = Create<B>(10, 20, 30);
320   Ptr<A> pA1 = Create<A>(100);
321   Ptr<B> pB1 = Create<B>(100, 200, 300);
322   // Use assignment operator
323   pA = pA1;
324   pB = pB1;
325   pA->Hello();
326   pB->Hello();
327 }
328
329 void Test6()
330 {
331   // Create a vector of 100 Ptr<A> objects
```

Program template-smart-pointers.cc (continued)

```
332     vector<Ptr<A> > v;
333     for (size_t i = 0; i < 100; ++i)
334       {
335         v.push_back(Create<A>(i));
336       }
337     // and just exit, no delete or clean up
338   }
339
340   // Main program
341   int main(int argc, char** argv)
342   {
343     if (argc < 2)
344       {
345         cout << "Usage: ./template-smart-pointers (testNum)" << endl;
346         exit(1);
347       }
348     int testNum = atol(argv[1]);
349     switch (testNum)
350       {
351       case 1:
352         Test1();
353         PrintCounts();
354         break;
355       case 2:
356         Test2();
357         PrintCounts();
358         break;
359       case 3:
360         Test3();
361         PrintCounts();
362         break;
363       case 4:
364         Test4();
365         PrintCounts();
366         break;
367       case 5:
368         Test5();
369         PrintCounts();
370         break;
371       case 6:
372         Test6();
373         PrintCounts();
374         break;
375       }
376   }
```

Program template-smart-pointers.cc (continued)