```
1  // Illustrate the smart pointer approach using Templates
2  // George F. Riley, Georgia Tech, Spring 2012
3  // This is nearly identical to the earlier handout on smart pointers
4  // but uses a different syntax to "Create" the objects.  Note
5  // the Ptr constructor is templated to allow up to three arguments
6  // which are passed to the "T" constructor.  The "Create" methods
7  // of the prior approach are not needed in this design.
8
9  #include <iostream>
10 #include <vector>
11
12 using namespace std;
13
14 // The Ptr class contains two members, one a pointer to a generic
15 // template type "T", and the int pointer for the reference count
16
17 template <typename T>
18 class Ptr {
19 public:
20   Ptr();
21   template <typename T1> Ptr(T1);
22   template <typename T1, typename T2> Ptr(T1, T2);
23   template <typename T1, typename T2, typename T3> Ptr(T1, T2, T3);
24
25   // We need a copy constructor, assignment operator, and destructor
26   Ptr(const Ptr&);               // Copy constructor
27   Ptr& operator=(const Ptr& rhs); // Assignment operator
28   ~Ptr();                        // Destructor
29
30   // And we need a de-referencing operator, which should return
31   // the dereferenced "t" pointer;
32   const T& operator*() const;     // De-referencing operator
33   T& operator*();                 // Non-const version
34   // And the "arrow" operator.  Need both const and non-const
35   T* operator->();
36   const T* operator->() const;
37 private:
38   // Members are private
39   T* t;
40   int* refCount;
41 };
42
43 // Ptr Implementations
44 #ifdef OLD
45 template <typename T>
46 Ptr<T>::Ptr(T* t0)
47 : t(t0)
48 { // Constructor
49   refCount = new int; // Create the refcount variable
50   *refCount = 1;      // Initially only one reference (this one).
51 };
52 #endif
53
54 template <typename T>
55 Ptr<T>::Ptr()
56 {
```

Program template-smart-pointers-again.cc

```
57    t = new T;
58    refCount = new int; // Create the refcount variable
59    *refCount = 1;      // Initially only one reference (this one).
60  }
61
62  template <typename T>
63  template <typename T1>
64  Ptr<T>::Ptr(T1 t1)
65  {
66    t = new T(t1);
67    refCount = new int; // Create the refcount variable
68    *refCount = 1;      // Initially only one reference (this one).
69  }
70
71
72  template <typename T>
73  template <typename T1, typename T2>
74  Ptr<T>::Ptr(T1 t1, T2 t2)
75  {
76    t = new T(t1,t2);
77    refCount = new int; // Create the refcount variable
78    *refCount = 1;      // Initially only one reference (this one).
79  }
80
81  template <typename T>
82  template <typename T1, typename T2, typename T3>
83  Ptr<T>::Ptr(T1 t1, T2 t2, T3 t3)
84  {
85    t = new T(t1,t2,t3);
86    refCount = new int; // Create the refcount variable
87    *refCount = 1;      // Initially only one reference (this one).
88  }
89
90
91  // Copy Constructor
92  template<typename T>
93  Ptr<T>::Ptr(const Ptr<T>& rhs)
94  : refCount(rhs.refCount), t(rhs.t)
95  {
96    // Increment the refCount
97    (*refCount)++;
98  }
99
100 // Assignment operator
101 template <typename T>
102 Ptr<T>& Ptr<T>::operator=(const Ptr<T>& rhs)
103 {
104   // Protect against self assignment
105   if (&rhs == this) return *this;
106   // First reduce refcount in lhs and free if needed
107   (*refCount)--;
108   if (*refCount == 0)
109     { // No remaining references
110       delete t;
111       delete refCount;
112     }
```

Program template-smart-pointers-again.cc (continued)

```
113     refCount = rhs.refCount;
114     t = rhs.t;
115     (*refCount)++; // Another reference
116     return *this;
117   }
118
119   // Destructor
120   template<typename T>
121   Ptr<T>::~Ptr<T>()
122   {
123     (*refCount)--;
124     if (*refCount == 0)
125       { // No remaining references
126         delete t;
127         delete refCount;
128       }
129   }
130
131   // Dereferencing operators
132   template <typename T>
133   const T& Ptr<T>::operator*() const
134   {
135     return *t;
136   }
137
138   template <typename T>
139   T& Ptr<T>::operator*()
140   {
141     return *t;
142   }
143
144   // Arrow operators
145   template <typename T>
146   const T* Ptr<T>::operator->() const
147   {
148     return t;
149   }
150
151   template <typename T>
152   T* Ptr<T>::operator->()
153   {
154     return t;
155   }
156
157
158
159   // Now create classes A and B that we will use to illustrate the smart
160   // pointers.
161
162   class A
163   {
164   public:
165     // constructors and destructors
166     A();
167     A(int);
168     A(const A&);
```

Program template-smart-pointers-again.cc (continued)

```
169     ~A();
170     void Hello() const;
171   public:
172     int a;
173   public:
174     static int constructorCount; // Counts total number of constructors
175     static int destructorCount;  // Counts total number of destructors
176   };
177
178   class B
179   {
180   public:
181     B();
182     B(int, int, int); // three args
183     B(const B&);
184     ~B();
185     void Hello() const;
186   public:
187     int b0;
188     int b1;
189     int b2;
190   public:
191     static int constructorCount; // Counts total number of constructors
192     static int destructorCount;  // Counts total number of destructors
193   };
194
195   // Implement A and B
196   A::A()
197     : a(0)
198   {
199     constructorCount++;
200   };
201
202   A::A(int a0)
203     : a(a0)
204   {
205     constructorCount++;
206   };
207
208   A::A(const A& rhs)
209     : a(rhs.a)
210   {
211     constructorCount++;
212   };
213
214   A::~A()
215   { // destructor
216     destructorCount++;
217   };
218
219   void A::Hello() const
220   {
221     cout << "Hello from A, a is " << a << endl;
222   }
223
224
```

Program template-smart-pointers-again.cc (continued)

```
225  B::B()
226    : b0(0), b1(0), b2(0)
227  {
228    constructorCount++;
229  };
230
231  B::B(int b00, int b01, int b02)
232    : b0(b00), b1(b01), b2(b02)
233  {
234    constructorCount++;
235  };
236
237  B::B(const B& rhs)
238    : b0(rhs.b0), b1(rhs.b1), b2(rhs.b2)
239  {
240    constructorCount++;
241  };
242
243  B::~B()
244  { // destructor
245    destructorCount++;
246  };
247
248  void B::Hello() const
249  {
250    cout << "Hello from B, b0 is " << b0 << endl;
251  }
252
253  // Helper functions
254  void PrintCounts()
255  {
256    cout << "A constructor " << A::constructorCount
257         << " A destructor " << A::destructorCount << endl;
258    cout << "B constructor " << B::constructorCount
259         << " B destructor " << B::destructorCount << endl;
260  };
261
262  // Define the A and B static variables
263  int A::constructorCount = 0;
264  int A::destructorCount = 0;
265  int B::constructorCount = 0;
266  int B::destructorCount = 0;
267
268  void Test1()
269  {
270    // Just create a single A and B (with Create) and do not delete
271    Ptr<A> pA(10);
272    Ptr<B> pB(10,20,30);
273
274    // just exit; smart pointer semantics call destructors automatically
275  }
276
277  void Test2()
278  {
279    // Just create a single A and B (with Create), then use Ptr copy constructor
280    // to create two more; no deletes
```

Program template-smart-pointers-again.cc (continued)

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

Program template-smart-pointers-again.cc (continued)

```
337   pB->Hello();
338 }
339
340 void Test6()
341 {
342   // Create a vector of 100 Ptr<A> objects
343   vector<Ptr<A> > v;
344   for (size_t i = 0; i < 100; ++i)
345     {
346       v.push_back(Ptr<A>(i));
347     }
348   // and just exit, no delete or clean up
349 }
350
351 // Main program
352 int main(int argc, char** argv)
353 {
354   if (argc < 2)
355     {
356       cout << "Usage: ./template-smart-pointers (testNum)" << endl;
357       exit(1);
358     }
359   int testNum = atol(argv[1]);
360   switch (testNum)
361     {
362     case 1:
363       Test1();
364       PrintCounts();
365       break;
366     case 2:
367       Test2();
368       PrintCounts();
369       break;
370     case 3:
371       Test3();
372       PrintCounts();
373       break;
374     case 4:
375       Test4();
376       PrintCounts();
377       break;
378     case 5:
379       Test5();
380       PrintCounts();
381       break;
382     case 6:
383       Test6();
384       PrintCounts();
385       break;
386     }
387 }
```

Program template-smart-pointers-again.cc (continued)