

```
1 // Shaun Chemplavil U08713628
2 // shaun.chemplavil@gmail.com
3 // C/C++ Programming IV : Advanced Programming with Objects
4 // 152488 Raymond L. Mitchell III
5 // hw5.cpp
6 // Win10
7 // Visual C++ 19.0
8 //
9
10 #include <iostream>
11 #include <exception>
12
13 // To avoid auto_ptr ambiguity we need to avoid using std namespace
14 using std::cout;
15 using std::clog;
16
17 template <typename X>
18 class auto_ptr
19 {
20 public:
21     // Constructors
22     explicit auto_ptr(X * = 0) throw();
23     auto_ptr(auto_ptr &) throw();
24     template <typename Y> auto_ptr(auto_ptr<Y> &) throw();
25     // Destructor
26     ~auto_ptr() throw();
27     // Access
28     X *get() const throw();
29     X &operator*() const throw();
30     X *operator->() const throw();
31     // Assignment
32     auto_ptr &operator=(auto_ptr &) throw();
33     template <typename Y> auto_ptr &operator=(auto_ptr<Y> &) throw();
34     // Release & Reset
35     X *release() throw();
36     void reset(X * = 0) throw();
37
38 private:
39     X *aPtr;
40     template<typename Y>
41     friend class auto_ptr; // make all auto_ptr classes
42                           // friends of one another
43 };
44
45 // Explicit Constructor
46 template<typename X>
47 auto_ptr<X>::auto_ptr(X *ptr)
48     : aPtr(ptr) {}
49
50 // Copy Constructor
51 template<typename X>
52 auto_ptr<X>::auto_ptr(auto_ptr &rhs)
```

```
53     : aPtr(rhs.release()) {}
54
55 // Constructor Taking Ownership from auto_ptr
56 template<typename X>
57 template<typename Y>
58 auto_ptr<X>::auto_ptr(auto_ptr<Y>& rhs)
59     : aPtr(rhs.release()) {}
60
61 // Destructor
62 template<typename X>
63 auto_ptr<X>::~~auto_ptr()
64 {
65     delete aPtr;
66 }
67
68 template<typename X>
69 auto_ptr<X> &auto_ptr<X>::operator=(auto_ptr& rhs)
70 {
71     reset(rhs.release());
72     return *this;
73 }
74
75 // Copy Assignment
76 template<typename X>
77 template<typename Y>
78 auto_ptr<X>& auto_ptr<X>::operator=(auto_ptr<Y>& rhs)
79 {
80     // Check for self-assign
81     if (this != &rhs)
82         reset(rhs.release());
83     return *this;
84 }
85
86 // Dereference Operator
87 template<typename X>
88 X& auto_ptr<X>::operator*() const
89 {
90     return *aPtr;
91 }
92
93 // Pointer Access Operator
94 template<typename X>
95 X* auto_ptr<X>::operator->() const
96 {
97     return aPtr;
98 }
99
100 // Get function (Pointer Access Operator)
101 template<typename X>
102 X* auto_ptr<X>::get() const
103 {
104     return aPtr;
```

```
105 }
106
107 template<typename X>
108 X* auto_ptr<X>::release()
109 {
110     X *aPtrOld = aPtr;
111     // Set pointer to Null and output original address
112     aPtr = 0;
113     return aPtrOld;
114 }
115
116 // Reset Auto Pointer and Point to input address
117 template<typename X>
118 void auto_ptr<X>::reset(X *ptr)
119 {
120     if (aPtr != ptr) {
121         delete aPtr;
122         aPtr = ptr;
123     }
124 }
125
126 // Unit Tests:
127 void testAutoPtrExplicitConstructor()
128 {
129     try
130     {
131         const int testInput(9);
132         auto_ptr<int> testAutoPtr(new int(testInput));
133         clog << "testAutoPtrExplicitConstructor PASSED\n";
134     }
135     catch (...)
136     {
137         clog << "testAutoPtrExplicitConstructor FAILED\n";
138     }
139 }
140
141 void testAutoPtrCopyConstructor()
142 {
143     try
144     {
145         const int testSource(9);
146         auto_ptr<int> sourceAutoPtr(new int(testSource));
147         auto_ptr<int> sinkAutoPtr(sourceAutoPtr);
148
149         if ((testSource == *sinkAutoPtr) && (sourceAutoPtr.get() == 0))
150             clog << "testAutoPtrCopyConstructor PASSED\n";
151         else
152             clog << "testAutoPtrCopyConstructor FAILED : Expected output "
153             << testSource << " instead saw " << *sinkAutoPtr << "\n";
154     }
155     catch (...)
156     {
```

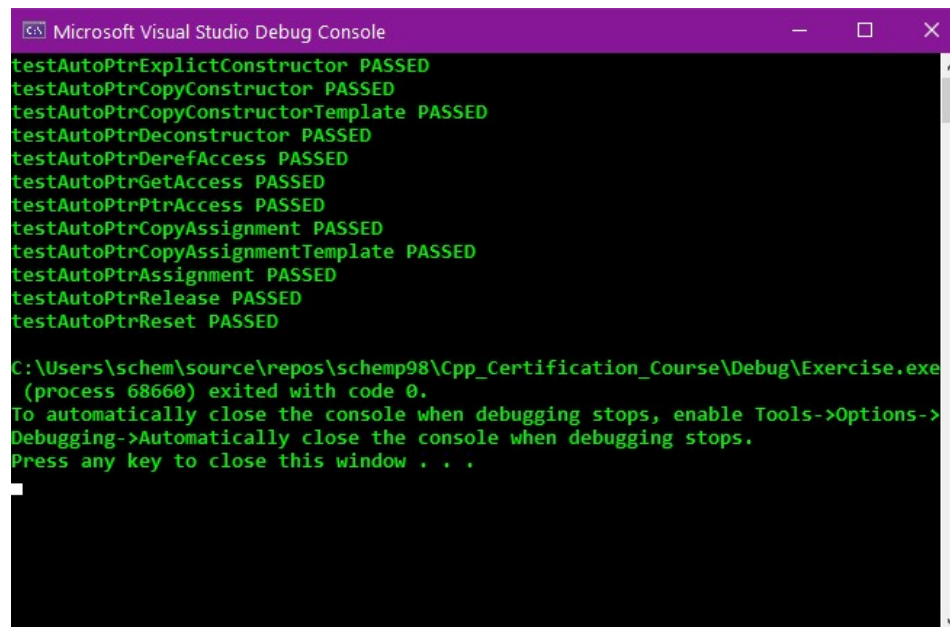
```
157     clog << "testAutoPtrCopyConstructor FAILED\n";
158 }
159 }
160
161 void testAutoPtrCopyConstructorTemplate()
162 {
163     try
164     {
165         const int testSource(9);
166         auto_ptr<int> sourceAutoPtr(new int(testSource));
167         auto_ptr<const int> sinkAutoPtr(sourceAutoPtr);
168
169         if ((testSource == *sinkAutoPtr) && (sourceAutoPtr.get() == 0))
170             clog << "testAutoPtrCopyConstructorTemplate PASSED\n";
171         else
172             clog << "testAutoPtrCopyConstructorTemplate FAILED : Expected output "
173                 << testSource << " instead saw " << *sinkAutoPtr << "\n";
174     }
175     catch (...)
176     {
177         clog << "testAutoPtrCopyConstructorNewType FAILED\n";
178     }
179 }
180
181 void testAutoPtrDestructor()
182 {
183     const int testInput(9);
184     auto_ptr<int> *testAutoPtr = new auto_ptr<int>(new int(testInput));
185
186     try
187     {
188         delete testAutoPtr;
189         clog << "testAutoPtrDestructor PASSED\n";
190     }
191     catch (...)
192     {
193         clog << "testAutoPtrDestructor FAILED\n";
194     }
195 }
196
197 void testAutoPtrDerefAccess()
198 {
199     try
200     {
201         const int testSource(9);
202         auto_ptr<int> sourceAutoPtr(new int(testSource));
203
204         if (testSource == *sourceAutoPtr)
205             clog << "testAutoPtrDerefAccess PASSED\n";
206         else
207             clog << "testAutoPtrDerefAccess FAILED : Expected output "
208                 << testSource << " instead saw " << *sourceAutoPtr << "\n";
```

```
209     }
210     catch (...)
211     {
212         clog << "testAutoPtrDerefAccess FAILED\n";
213     }
214 }
215
216 void testAutoPtrGetAccess()
217 {
218     try
219     {
220         const int testSource(9);
221         auto_ptr<int> sourceAutoPtr(new int(testSource));
222         int * sinkPtr = sourceAutoPtr.get();
223
224         if (sinkPtr == sourceAutoPtr.get())
225             clog << "testAutoPtrGetAccess PASSED\n";
226         else
227             clog << "testAutoPtrGetAccess FAILED : Expected output "
228                 << sinkPtr << " instead saw " << sourceAutoPtr.get() << "\n";
229     }
230     catch (...)
231     {
232         clog << "testAutoPtrGetAccess FAILED\n";
233     }
234 }
235
236 void testAutoPtrPtrAccess()
237 {
238     try
239     {
240         const int testSource(9);
241         auto_ptr<int> sourceAutoPtr;
242
243         int *sinkPtr = sourceAutoPtr.operator->();
244
245         if (sinkPtr == sourceAutoPtr.get())
246             clog << "testAutoPtrPtrAccess PASSED\n";
247         else
248             clog << "testAutoPtrPtrAccess FAILED : Expected output "
249                 << sinkPtr << " instead saw " << sourceAutoPtr.get() << "\n";
250     }
251     catch (...)
252     {
253         clog << "testAutoPtrPtrAccess FAILED\n";
254     }
255 }
256
257 void testAutoPtrCopyAssignment()
258 {
259     try
260     {
```

```
261     const int testSource(9);
262     auto_ptr<int> sourceAutoPtr(new int(testSource));
263
264     //auto_ptr<int> sinkAutoPtr(new int(0));
265     auto_ptr<int> sinkAutoPtr = sourceAutoPtr;
266     //sinkAutoPtr = sourceAutoPtr;
267
268     if ((testSource == *sinkAutoPtr) && (sourceAutoPtr.get() == 0))
269         clog << "testAutoPtrCopyAssignment PASSED\n";
270     else
271         clog << "testAutoPtrCopyAssignment FAILED : Expected output "
272             << testSource << " instead saw " << *sinkAutoPtr << "\n";
273 }
274 catch (...)
275 {
276     clog << "testAutoPtrCopyAssignment FAILED\n";
277 }
278 }
279
280 void testAutoPtrCopyAssignmentTemplate()
281 {
282     try
283     {
284         const int testSource(9);
285         auto_ptr<int> sourceAutoPtr(new int(testSource));
286
287         auto_ptr<const int> sinkAutoPtr = sourceAutoPtr;
288
289         if ((testSource == *sinkAutoPtr) && (sourceAutoPtr.get() == 0))
290             clog << "testAutoPtrCopyAssignmentTemplate PASSED\n";
291         else
292             clog << "testAutoPtrCopyAssignmentTemplate FAILED : Expected output "
293                 << testSource << " instead saw " << *sinkAutoPtr << "\n";
294     }
295     catch (...)
296     {
297         clog << "testAutoPtrCopyAssignmentTemplate FAILED\n";
298     }
299 }
300 void testAutoPtrAssignment()
301 {
302     try
303     {
304         const int testSource(9);
305         auto_ptr<int> sourceAutoPtr(new int(testSource));
306
307         auto_ptr<int> sinkAutoPtr(new int(0));
308         sinkAutoPtr = sourceAutoPtr;
309
310         if ((testSource == *sinkAutoPtr) && (sourceAutoPtr.get() == 0))
311             clog << "testAutoPtrAssignment PASSED\n";
312         else
```

```
313     clog << "testAutoPtrAssignment FAILED : Expected output "  
314     << testSource << " instead saw " << *sinkAutoPtr << "\n";  
315 }  
316 catch (...)  
317 {  
318     clog << "testAutoPtrAssignment FAILED\n";  
319 }  
320 }  
321  
322 void testAutoPtrRelease()  
323 {  
324     try  
325     {  
326         const int testSource(9);  
327         auto_ptr<int> sourceAutoPtr(new int(testSource));  
328         int *sinkVal(sourceAutoPtr.release());  
329  
330         if ((testSource == *sinkVal) && (sourceAutoPtr.get() == 0))  
331             clog << "testAutoPtrRelease PASSED\n";  
332         else  
333             clog << "testAutoPtrRelease FAILED : Expected output "  
334             << testSource << " instead saw " << sinkVal << "\n";  
335     }  
336     catch (...)  
337     {  
338         clog << "testAutoPtrRelease FAILED\n";  
339     }  
340 }  
341  
342 void testAutoPtrReset()  
343 {  
344     try  
345     {  
346         const int testSource(9);  
347         int * sourcePtr = new int(testSource * testSource);  
348         auto_ptr<int> sinkAutoPtr(new int(testSource));  
349  
350         sinkAutoPtr.reset(sourcePtr);  
351  
352         if (sinkAutoPtr.get() == sourcePtr)  
353             clog << "testAutoPtrReset PASSED\n";  
354         else  
355             clog << "testAutoPtrReset FAILED : Expected output "  
356             << sourcePtr << " instead saw " << sinkAutoPtr.get() << "\n";  
357     }  
358     catch (...)  
359     {  
360         clog << "testAutoPtrReset FAILED\n";  
361     }  
362 }  
363  
364 int main(void)
```

```
365 {  
366     // 1)  
367     testAutoPtrExplicitConstructor();  
368     testAutoPtrCopyConstructor();  
369     testAutoPtrCopyConstructorTemplate();  
370  
371     // 2)  
372     testAutoPtrDestructor();  
373  
374     // 3)  
375     testAutoPtrDerefAccess();  
376     testAutoPtrGetAccess();  
377     testAutoPtrPtrAccess();  
378  
379     //4)  
380     testAutoPtrCopyAssignment();  
381     testAutoPtrCopyAssignmentTemplate();  
382     testAutoPtrAssignment();  
383  
384     //5)  
385     testAutoPtrRelease();  
386     testAutoPtrReset();  
387 }  
388
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar is purple and contains the text "Microsoft Visual Studio Debug Console". The console output is as follows:

```
testAutoPtrExplicitConstructor PASSED  
testAutoPtrCopyConstructor PASSED  
testAutoPtrCopyConstructorTemplate PASSED  
testAutoPtrDestructor PASSED  
testAutoPtrDerefAccess PASSED  
testAutoPtrGetAccess PASSED  
testAutoPtrPtrAccess PASSED  
testAutoPtrCopyAssignment PASSED  
testAutoPtrCopyAssignmentTemplate PASSED  
testAutoPtrAssignment PASSED  
testAutoPtrRelease PASSED  
testAutoPtrReset PASSED  
  
C:\Users\schem\source\repos\schemp98\Cpp_Certification_Course\Debug\Exercise.exe  
(process 68660) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->  
Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```