```
1  // Demonstrate the use of the Standard Template Library "vector" class.
2  // George F. Riley, ECE3090 Georgia Tech, Fall 2009
3
4  // A vector is a variable length array.  It starts out as "zero" length
5  // and grows or shrinks as needed.  Further, the vector is a array
6  // of any arbitrary type, using the C++ "templates" feature.
7
8  #include <iostream>
9  #include <vector>
10 //#include "GFRVec.h" // Uncomment to use the "buggy" one
11 #include "GFRVec1.h"  // Comment out this one if using the above one
12 class A {
13 public:
14   A();         // Default constructor
15   A(int);      // Non-Default Constructor
16   A(const A&); // A copy constructor is used by the compile whenever
17   A& operator=(const A&); // Assignment operator
18   ~A();        // Destructor
19 public:
20   int x;       // Single data member
21 };
22
23
24 typedef std::vector<A>  AVec_t; // Define a type that is vector of A's
25 typedef std::vector<A*> APVec_t;// Define a type that is vector of A pointers
26 // Uncomment to use our own implementation (and comment out above)
27 //typedef GFRVec<A>  AVec_t; // Define a type that is vector of A's
28 //typedef GFRVec<A*> APVec_t;// Define a type that is vector of A pointers
29
30 A::A()
31 {
32   std::cout << "Hello from A::A() Default constructor" << std::endl;
33 }
34
35 A::A(int i)
36     : x(i)
37 {
38   std::cout << "Hello from A::A(int) constructor" << std::endl;
39 }
40
41 A::A(const A& a)
42     : x(a.x)
43 {
44   std::cout << "Hello from A Copy constructor" << std::endl;
45 }
46
47 A& A::operator=(const A& rhs)
48 {
49   std::cout << "Hello from A Assignment operator" << std::endl;
50   x = rhs.x;
51 }
52
53 A::~A()
54 {
55   std::cout << "Hello from A Destructor" << std::endl;
56 }
```

Program vector.cc

```
57
58  int main()
59  {
60    std::cout << "Creating A Vector"; getchar();
61    AVec_t av0;
62
63    std::cout << "Adding an three elements to av0"; getchar();
64    av0.push_back(A(2));    // Elements are appended using "push_back"
65    std::cout << "After first push_back"; getchar();
66    av0.push_back(A(10));   // Elements are appended using "push_back"
67    std::cout << "After second push_back"; getchar();
68    av0.push_back(A(100));  // Elements are appended using "push_back"
69    // Number of elements in a vector can be queried with "size()"
70    std::cout << "After third push_back, size av0 is "<<av0.size()<<std::endl;
71
72    // Now reserve space for  up to 10 elements, allowing for more
73    // efficient push_back.
74    std::cout << "Reserving 10 elements"; getchar();
75    av0.reserve(10);
76    // Push a few more to show better efficiency
77    std::cout << "Pushing three more elements"; getchar();
78    av0.push_back(A(101));
79    av0.push_back(A(102));
80    av0.push_back(A(103));
81
82    // Individual elements can be accessed with the [] operator
83    std::cout << "Accessing elements with the [] operator"; getchar();
84    std::cout << "av0[0].x is " << av0[0].x << std::endl;
85    std::cout << "av0[1].x is " << av0[1].x << std::endl;
86    std::cout << "av0[2].x is " << av0[2].x << std::endl;
87
88    // Front and back of list have special accessors
89    std::cout << "Accessing elements with the front and back"; getchar();
90    std::cout << "av0.front().x is " << av0.front().x << std::endl;
91    std::cout << "av0.back().x is "  << av0.back().x << std::endl;
92
93    // Vectors can be copied with copy constructor or assignment operator
94    std::cout << "Making a copy of av0"; getchar();
95    AVec_t av1(av0);
96    std::cout << "Size of av1 is " << av1.size() << std::endl;
97    std::cout << "av1[0].x is "    << av1[0].x << std::endl;
98
99    // Vectors can be shrunk with "pop_back".  Notice that pop_back
100   // does NOT return the element being popped
101   std::cout << "Shrinking av0 with pop_back"; getchar();
102   av0.pop_back(); // Remove last element
103   std::cout << "Size of av0 is " << av0.size() << std::endl;
104   av0.pop_back(); // Remove another element
105   std::cout << "Size of av0 is " << av0.size() << std::endl;
106
107   // Vectors can be initialized to "n" copies of a specified object
108   std::cout << "Constructing  AVec_t with 10 elements"; getchar();
109   AVec_t av2(10, A(1)); // Makes 10 elements of A(1)
110   std::cout << "Size of av2 is " << av2.size() << std::endl;
111   std::cout << "av2[0].x is " << av2[0].x << std::endl;
112
```

Program vector.cc (continued)

```
113     // All elements of a vector can be removed with "clear()"
114     std::cout << "Clearing av2"; getchar();
115     av2.clear();
116     std::cout << "Size of av2 is " << av2.size() << std::endl;
117
118     // Push another element to demonstrate that "clear" did not
119     // free the memory.
120     std::cout << "push another on av2"; getchar();
121     av2.push_back(A(100));
122
123     std::cout << "push another on av2"; getchar();
124     av2.push_back(A(200));
125
126     // Create and populate a vector of A poiners
127     std::cout << "Creating A Pointer Vector"; getchar();
128     APVec_t apv0;
129
130     std::cout << "Adding an three elements to apv0"; getchar();
131     apv0.push_back(new A(2));
132     apv0.push_back(new A(10));
133     apv0.push_back(new A(100));
134     // Number of elements in a vector can be queried with "size()"
135     std::cout << "Size of apv0 is " << apv0.size() << std::endl;
136
137     // Clear the apv0 vector.  Note: ~A() NOT called.  Why not?
138     std::cout << "Clearing apv0"; getchar();
139     apv0.clear();
140     std::cout << "Size of apv0 is " << apv0.size() << std::endl;
141
142     std::cout << "Main program exiting"; getchar();
143     return 0;
144   }
145
146   // Output from this program is:
147   //
148   // Creating A Vector
149   // Adding an three elements to av0
150   // Hello from A::A(int) constructor
151   // Hello from A Copy constructor
152   // Hello from A Destructor
153   // After first push_back
154   // Hello from A::A(int) constructor
155   // Hello from A Copy constructor
156   // Hello from A Copy constructor
157   // Hello from A Destructor
158   // Hello from A Destructor
159   // After second push_back
160   // Hello from A::A(int) constructor
161   // Hello from A Copy constructor
162   // Hello from A Copy constructor
163   // Hello from A Copy constructor
164   // Hello from A Destructor
165   // Hello from A Destructor
166   // Hello from A Destructor
167   // After third push_back, size av0 is 3
168   // Reserving 10 elements
```

Program vector.cc (continued)

```
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Destructor
// Hello from A Destructor
// Hello from A Destructor
// Pushing three more elements
// Hello from A::A(int) constructor
// Hello from A Copy constructor
// Hello from A Destructor
// Hello from A::A(int) constructor
// Hello from A Copy constructor
// Hello from A Destructor
// Hello from A::A(int) constructor
// Hello from A Copy constructor
// Hello from A Destructor
// Accessing elements with the [] operator
// av0[0].x is 2
// av0[1].x is 10
// av0[2].x is 100
// Accessing elements with the front and back
// av0.front().x is 2
// av0.back().x is 103
// Making a copy of av0
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Size of av1 is 6
// av1[0].x is 2
// Shrinking av0 with pop_back
// Hello from A Destructor
// Size of av0 is 5
// Hello from A Destructor
// Size of av0 is 4
// Constructing  AVec_t with 10 elements
// Hello from A::A(int) constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Copy constructor
// Hello from A Destructor
// Size of av2 is 10
// av2[0].x is 1
// Clearing av2
// Hello from A Destructor
// Hello from A Destructor
// Hello from A Destructor
```

Program vector.cc (continued)

```
225  // Hello from A Destructor
226  // Hello from A Destructor
227  // Hello from A Destructor
228  // Hello from A Destructor
229  // Hello from A Destructor
230  // Hello from A Destructor
231  // Hello from A Destructor
232  // Size of av2 is 0
233  // push another on av2
234  // Hello from A::A(int) constructor
235  // Hello from A Copy constructor
236  // Hello from A Destructor
237  // Creating A Pointer Vector
238  // Adding an three elements to apv0
239  // Hello from A::A(int) constructor
240  // Hello from A::A(int) constructor
241  // Hello from A::A(int) constructor
242  // Size of apv0 is 3
243  // Clearing apv0
244  // Size of apv0 is 0
245  // Main program exiting
246  // Hello from A Destructor
247  // Hello from A Destructor
248  // Hello from A Destructor
249  // Hello from A Destructor
250  // Hello from A Destructor
251  // Hello from A Destructor
252  // Hello from A Destructor
253  // Hello from A Destructor
254  // Hello from A Destructor
255  // Hello from A Destructor
256  // Hello from A Destructor
```

Program vector.cc (continued)

```
1  // An implementation of a simplified STL Vector
2  // George F. Riley, Georgia Tech, Fall 2009
3
4  template<class T> class GFRVec
5  {
6  public:
7    GFRVec() : first(0), last(0), end(0) {}
8    GFRVec(size_t n)
9      { // Create a GFRVec with "n" copies of T, with default constructor
10       first = new T[n];
11       last = first + n;
12       end = last;
13     }
14   GFRVec(size_t n, const T& t)
15     { // Create a GFRVec with "n" copies of t
16       first = new T[n];
17       for (size_t i = 0; i < n; ++i) first[i] = t; // Populate the vector
18       last = first + n;
19       end = last;
20     }
21   GFRVec(const GFRVec& c)
22     { // Copy Constructor
23       first = new T[c.size()]; // Allocate memory
24       for (size_t i = 0; i < c.size(); ++i) first[i] = c[i]; // Copy elements
25       last = first + c.size();
26       end = last;
27     }
28   ~GFRVec()
29     { // Destructor, remove all elements
30       clear();
31     }
32   GFRVec& operator=(const GFRVec& rhs)
33     { // Assignment operator
34       if (this == &rhs) return *this; // Self assignment
35       delete [] first;                // Free old memory
36       first = new T[rhs.size()]; // Allocate memory
37       for (size_t i = 0; i < rhs.size(); ++i)
38         first[i] = rhs[i]; // Copy the elements
39       last = first + rhs.size();
40       end = last;
41     }
42   T& operator[](size_t i) const
43     { // Indexing operator
44       return first[i];
45     }
46   T& back() const
47     { // Return last element
48       return first[size()-1];
49     }
50   T& front() const
51     { // Return last element
52       return first[0];
53     }
54   void pop_back()
55     { // Remove last element
56       last--;
```

Program GFRVec.h

```
57        first[size()].~T(); // Call destructor on just popped object
58      }
59    void push_back(const T& t)
60      { // Add new element
61        if (last != end)
62          { // Room for new object without re-allocating
63            first[size()] = t;
64            last++;
65          }
66        else
67          { // Need to re-allocate
68            T* tmp = new T[end-first+1];
69            for (size_t i = 0; i < size(); ++i) tmp[i] = first[i];
70            tmp[size()] = t; // Add new element
71            last = tmp + (last - first) + 1;
72            end = last;
73            delete [] first;   // Delete and destroy old objects
74            first = tmp;
75          }
76      }
77    size_t size() const
78      { // Number of elements in the vector
79        return last - first;
80      }
81    void reserve(size_t n)
82      { // Reserve space for "n" elements
83        if (n <= (end-first)) return; // Less than already reserved
84        T* tmp = new T[n];            // Allocate new memory
85        for (size_t i = 0; i < size(); ++i) tmp[i] = first[i];
86        last = tmp + last - first;
87        delete [] first;
88        first = tmp;
89        end = first + n;
90      }
91    void clear()
92    { // Erase all elements
93      while(size()) pop_back();
94    }
95
96  private:
97    T* first; // Initial element
98    T* last;  // Last element
99    T* end;   // End of allocated storage
100 };
101
102
103
104
```

Program GFRVec.h (continued)

```
1  // An implementation of a simplified STL Vector
2  // This one uses uninitialized alloc and placement new operator
3  // George F. Riley, Georgia Tech, Fall 2009
4
5  template<class T> class GFRVec
6  {
7  public:
8    GFRVec() : first(0), last(0), end(0) {}
9    GFRVec(size_t n)
10     { // Create a GFRVec with "n" copies of T, with default constructor
11       first = (T*)malloc(n * sizeof(T));
12       for (size_t i = 0; i < n; ++i)
13         {
14           new (&first[i]) T();
15         }
16       last = first + n;
17       end = last;
18     }
19   GFRVec(size_t n, const T& t)
20     { // Create a GFRVec with "n" copies of t
21       first = (T*)malloc(n * sizeof(T));
22       for (size_t i = 0; i < n; ++i)
23         { // Use copy constructor to populate
24           new (&first[i]) T(t);
25         }
26       last = first + n;
27       end = last;
28     }
29   GFRVec(const GFRVec& c)
30     { // Copy Constructor
31       first = (T*)malloc(c.size() * sizeof(T));
32       for (size_t i = 0; i < c.size(); ++i)
33         {
34           new (&first[i]) T(c[i]); // Copy elements
35         }
36
37       last = first + c.size();
38       end = last;
39     }
40   ~GFRVec()
41     { // Destructor, remove all elements
42       clear();
43     }
44   GFRVec& operator=(const GFRVec& rhs)
45     { // Assignment operator
46       if (this == &rhs) return *this; // Self assignment
47       free(first);
48       first = (T*)malloc(rhs.size() * sizeof(T));
49       for (size_t i = 0; i < rhs.size(); ++i)
50         {
51           new (&first[i]) T(rhs[i]); // Copy the elements
52         }
53       last = first + rhs.size();
54       end = last;
55     }
56   T& operator[](size_t i) const
```

Program GFRVec1.h

```
57      { // Indexing operator
58        return first[i];
59      }
60    T& back() const
61      { // Return last element
62        return first[size()-1];
63      }
64    T& front() const
65      { // Return last element
66        return first[0];
67      }
68    void pop_back()
69      { // Remove last element
70        last--;
71        first[size()].~T(); // Call destructor on just popped object
72      }
73    void push_back(const T& t)
74      { // Add new element
75        if (last != end)
76          { // Room for new object without re-allocating
77            new (&first[size()]) T(t);
78            last++;
79          }
80        else
81          { // Need to re-allocate
82            T* tmp = (T*)malloc((size() + 1) * sizeof(T));
83            for (size_t i = 0; i < size(); ++i)
84              { // Copy old elements
85                new (&tmp[i]) T(first[i]);
86              }
87            new (&tmp[size()]) T(t);
88            for (size_t i = 0; i < size(); ++i)
89              { // Destroy old elements
90                first[i].~T();
91              }
92            last = tmp + (last - first) + 1;
93            end = last;
94            free(first);
95            first = tmp;
96          }
97      }
98    size_t size() const
99      { // Number of elements in the vector
100       return last - first;
101     }
102   void reserve(size_t n)
103     { // Reserve space for "n" elements
104       if (n <= (end-first)) return; // Less than already reserved
105       T* tmp = (T*)malloc(n * sizeof(T));
106       for (size_t i = 0; i < size(); ++i)
107         { // Copy elements to new space
108           new (&tmp[i]) T(first[i]);
109         }
110       last = tmp + last - first;
111       free(first);
112       first = tmp;
```

Program GFRVec1.h (continued)

```
113         end = first + n;
114       }
115     void clear()
116     { // Erase all elements
117       while(size()) pop_back();
118       free(first);
119       first = 0;
120       last = 0;
121       end = 0;
122     }
123
124   private:
125     T* first; // Initial element
126     T* last;  // Last element
127     T* end;   // End of allocated storage
128   };
129
130
131
132
```

Program GFRVec1.h (continued)