

```

1 // Example illustrating dynamic memory management
2 // ECE3090
3 // George F. Riley, Georgia Tech, Spring 2012
4
5 #include <iostream>
6
7 using namespace std; // We will discuss namespaces later
8
9 class A {
10 public:
11     A(); // Default Constructor
12     A(int i); // Int Constructor
13     A(const A& a0); // Copy constructor
14     ~A(); // Destructor
15 public:
16     int a; // Member variable
17 };
18
19
20 // Implement the constructors
21 A::A() : a(0)
22 { // Default Constructor
23     cout << "Hello from A Default Constructor, a is " << a << endl;
24 }
25
26 A::A(int a0) : a(a0)
27 { // Default Constructor
28     cout << "Hello from A int Constructor, a is " << a << endl;
29 }
30
31 A::A(const A& a0) : a(a0.a)
32 { // Default Constructor
33     cout << "Hello from A Copy Constructor, a is " << a << endl;
34 }
35
36 A::~A()
37 { // Destructor
38     cout << "Hello from A destructor, a is " << a << endl;
39 }
40
41 // Define a global array of A objects.
42 // We already know that the compiler is responsible for finding
43 // memory, calling the constructor(s) prior to entering "main"
44 // and calling the destructors after exiting "main"
45 A globalArrayA[10]; // How many constructors?
46
47 // A subroutine that allocates an array dynamically and does
48 // not delete it.
49 void Sub1(int numberObjects)
50 {
51     // Allocate an array of A objects using new
52     cout << "In sub1, allocating \"numberObjects\" A objects with new" << endl;
53     A* pA = new A[numberObjects];
54     // Since we used the default constructor, all a's should be
55     // initialized to zero.
56     cout << "In sub1, printing object values" << endl;

```

Program dynamic-memory.cc

```

57     for (int i = 0; i < numberObjects; ++i)
58     {
59         cout << "pA[" << i << "] is " << pA[i].a << endl;
60     }
61     // Now we exit without doing anything with pA (no delete).
62     // What happens here?
63 }
64
65 A* Sub2(int numberObjects)
66 {
67     // Allocate an array of A objects using new
68     cout << "In sub2, allocating \"numberObjects\" A objects with new" << endl;
69     A* pA = new A[numberObjects];
70     // Initialize the pA.a members to non-default values
71     for (int i = 0; i < numberObjects; ++i)
72     {
73         pA[i].a = i * 100;
74     }
75     // Now we exit by returning the pA pointer to the caller.
76     // but not "deleting" it. Is this a memory leak?
77     return pA;
78 }
79
80 int main()
81 { // Illustrate the use of dynamic memory
82     // First some simple local variables. In all cases, the
83     // memory to hold the variable is automatically allocated on
84     // the stack, and the constructor is called. When the function
85     // exits (in this case "main" exiting, the destructor is called
86     // then the memory is "deleted"
87     cout << "Entering main" << endl;
88     A a0(1); // Single A object on stack with "int" constructor
89     cout << "Creating local aArray[20]" << endl;
90     A aArray[20]; // Array of 20 A's, using default constructor
91     // Unfortunately, there is no easy syntax for creating an array of
92     // "k" A object with non-default constructor, although it can be
93     // done. We will discuss array initialization later.
94     //A aArray2[10](10); // Won't compile
95
96     // So far we create a global array of A objects "globalArrayA"
97     // and a local array "aArray". The problem is that in both cases
98     // we have to know AT COMPILE TIME the size of the array. What
99     // we really want is a way to decide AT RUN TIME how big an array
100    // should be. This can be done by using the HEAP.
101
102    // For this simple example, we just use the constant 8, but let's
103    // assume that "arraySize" is somehow determined at run time and
104    // can be any reasonable value.
105    int arraySize = 8;
106
107    cout << "Allocating pointerToArray" << endl;
108    A* pointerToArray = new A[arraySize];
109    // Note the use of the "new" operator. This does three separate and
110    // distinct things:
111    // 1) Find enough contiguous memory for "arraySize" objects of class A
112    // 2) Call the default constructor on each of the new A objects

```

Program dynamic-memory.cc (continued)

```

113 // 3) Return a POINTER to the allocated memory
114 // Let's initialize the new array to some value
115 for (int i = 0; i < arraySize; ++i)
116 {
117     pointerToArray[i].a = i;
118 }
119 // And print them out
120 for (int i = 0; i < arraySize; ++i)
121 {
122     cout << "element " << i << " is " << pointerToArray[i].a << endl;
123 }
124 // Since we explicitly allocated memory for "pointerToArray" with new,
125 // we must explicitly destroy the memory with "delete". And since
126 // we allocated an array of objects, we need a special form of
127 // delete as shown.
128 cout << "Deleting pointerToA" << endl;
129 delete [] pointerToArray;
130
131 // We can also use "new" to allocate a single object. In this case
132 // we can specify a non-default constructor
133 cout << "Allocating single A object with int constructor" << endl;
134 A* pA = new A(200); // Int constructor
135 cout << "pA.a is " << pA->a << endl;
136 // And we should return the memory with "delete".
137 cout << "Deleting pA" << endl;
138 delete pA;
139
140 // Call a subroutine to illustrate a "memory leak".
141 cout << "Calling sub1" << endl;
142 Sub1(5);
143
144 // Call a subroutine illustrating a function returning a pointer
145 // to a dynamically allocated array
146 cout << "Calling sub2 to allocate an array " << endl;
147 A* pA2 = Sub2(10);
148 // Print out the returned pA2 array
149 cout << "Printing pA2 (return from Sub2)" << endl;
150 for (int i = 0; i < 10; ++i)
151 {
152     cout << "pA2[" << i << "] = " << pA2[i].a << endl;
153 }
154 // We need to return the memory for pA2 when we are done with it.
155 cout << "Deleting pA2" << endl;
156 delete [] pA2;
157 cout << "Exiting Main" << endl;
158 }

```

Program dynamic-memory.cc (continued)