// Example illustrating dynamic memory management

// ECE2036

// George F. Riley, Georgia Tech, Fall 2012

#include <iostream>

using namespace std; // We will discuss namespaces later

class A {
public:
  A(); // Default Constructor
  A(int i); // Int Constructor
  A(const A& a0); // Copy constructor
  ~A(); // Destructor

private:
  int a; // Member variable
};

// Implement the constructors
A::A() : a(0) { // Default Constructor
  cout << "Hello from A Default Constructor, a is " << a << endl;
}

A::A(int a0) : a(a0) { // Default Constructor
  cout << "Hello from A int Constructor, a is " << a << endl;
}

A::A(const A& a0) : a(a0.a) { // Default Constructor
  cout << "Hello from A Copy Constructor, a is " << a << endl;
}

A::~A() { // Destructor
  cout << "Hello from A destructor, a is " << a << endl;
}

// Define a global array of A objects.
// We already know that the compiler is responsible for finding
// memory, calling the constructor(s) prior to entering "main"
// and calling the destructors after exiting "main"
A globalArrayA[10]; // How many constructors?

// A subroutine that allocates an array dynamically and does
// not delete it.
void Sub1(int numberObjects)
{
  // Allocate an array of A objects using new
  cout << "In sub1, allocating " << numberObjects << " A objects with new" << endl;
  A* pA = new A[numberObjects];
  // Since we used the default constructor, all a’s should be
  // initialized to zero.
  cout << "In sub1, printing object values" << endl;
}

Program dynamic-memory.cc
for (int i = 0; i < numberObjects; ++i)
{
    cout << "pA[" << i << "] is " << pA[i].a << endl;
}
// Now we exit without doing anything with pA (no delete).
// What happens here?
}
A* Sub2(int numberObjects)
{
// Allocate an array of A objects using new
cout << "In sub2, allocating " << numberObjects << " A objects with new" << endl;
A* pA = new A[numberObjects];
// Initialize the pA.a members to non-default values
for (int i = 0; i < numberObjects; ++i)
{
    pA[i].a = i * 100;
}
// Now we exit by returning the pA pointer to the caller.
// but not "deleting" it. Is this a memory leak?
return pA;
}
int main()
{
    // Illustrate the use of dynamic memory
    // First some simple local variables. In all cases, the
    // memory to hold the variable is automatically allocated on
    // the stack, and the constructor is called. When the function
    // exits (in this case "main" exiting, the destructor is called
    // then the memory is "deleted"
    cout << "Entering main" << endl;
    A a0(1); // Single A object on stack with "int" constructor
    cout << "Creating local aArray[20]" << endl;
    A aArray[20]; // Array of 20 A's, using default constructor
    // Unfortunately, there is no easy syntax for creating an array of
    // "k" A object with non-default constructor, although it can be
    // done. We will discuss array initialization later.
    //A aArray2[10](10); // Won't compile
    // So far we create a global array of A objects "globalArrayA"
    // and a local array "aArray". The problem is that in both cases
    // we have to know AT COMPILe TIME the size of the array. What
    // we really want is a way to decide AT RUN TIME how big an array
    // should be. This can by done by using the HEAP.
    // For this simple example, we just use the constant 8, but let's
    // assume that "arraySize" is somehow determined at run time and
    // can be any reasonable value.
    int arraySize = 8;
    cout << "Allocating pointerToArray" << endl;
    A* pointerToArray = new A[arraySize];
    // Note the use of the "new" operator. This does three separate and
    // distinct things:
    // 1) Find enough contiguous memory for "arraySize" objects of class A
    // 2) Call the default constructor on each of the new A objects

Program dynamic-memory.cc (continued)
3) Return a POINTER to the allocated memory
// Let's initialize the new array to some value
for (int i = 0; i < arraySize; ++i)
{
    pointerToArray[i].a = i;
}

// And print them out
for (int i = 0; i < arraySize; ++i)
{
    cout << "element " << i << " is " << pointerToArray[i].a << endl;
}

// Since we explicitly allocated memory for "pointerToArray" with new,
// we must explicitly destroy the memory with "delete". And since
// we allocated an array of objects, we need a special form of
// delete as shown.
cout << "Deleting pointerToArray" << endl;
delete [] pointerToArray;

// We can also use "new" to allocate a single object. In this case
// we can specify a non-default constructor
cout << "Allocating single A object with int constructor" << endl;
A* pA = new A(200); // Int constructor
cout << "pA.a is " << pA->a << endl;
// And we should return the memory with "delete".
cout << "Deleting pA" << endl;
delete pA;

// Call a subroutine to illustrate a "memory leak".
cout << "Calling sub1" << endl;
Sub1(5);

// Call a subroutine illustrating a function returning a pointer
// to a dynamically allocated array
cout << "Calling sub2 to allocate an array " << endl;
A* pA2 = Sub2(10);
// Print out the returned pA2 array
for (int i = 0; i < 10; ++i)
{
    cout << "pA2[" << i << "] = " << pA2[i].a << endl;
}
// We need to return the memory for pA2 when we are done with it.
cout << "Deleting pA2" << endl;
delete [] pA2;
cout << "Exiting Main" << endl;