```
// Example illusrating dynamic memory management for a class that
// uses dynamic memory as a member variable
// ECE3090
// George F. Riley, Georgia Tech, Spring 2009

#include <iostream>

using namespace std;  // We will discuss namespaces later

// For this example we will define a new C++ object called "Vector"
// A vector is essentially an array of "int", but the size of the array
// is not known at compile time; rather, at run-time in the Vector
// constructor we will specify how big (number of elements) the array should
// be.  We do this by using the "new" operator in the constructor,
// and allocating the specified amount of memory for the dynmaically
// sized array
class Vector
{
public:
  // Notice no "default" constructor. What would a default constructor do?
  Vector(int nElements);  // Specify the number of elements in the constructor
  Vector(const Vector&);  // Define the copy constructor
  void operator=(const Vector&);  // Define the "assignment operator"
  ~Vector();              // Define the destructor
  // Now define the various member functions that we need to manage
  // the vector.  For this simple example will will provide a method
  // to query the "length" (maximum number of elements) of the array.
  int Length() const;       // Return the size of the array
  // We also need a way to "index" the array.  There are two ways
  // to do this:
  int  GetElement(int whichElement) const;     // Return an existing element
  void SetElement(int whichElement, int newValue); // Set a value in the array
  // However, much simpler than the above would be to overload the
  // "indexing" operator, which is the "[]" operator.
  // Pay particular attention to the return type for this function.
  // It is not an "int" but an "int reference".  It will be clear
  // later why this is the case.
  int& operator[](int whichElement);
  // Finally provide a "Print" operator  for debugging
  void Print() const;
  // Here are the member variables needed
private:       // Note the use of "private" here.  Will discuss in class
  int length;  // Size of the array
  int* pArray; // Dynamic memory pointer to the actual array
};
// End of class declaration for Vector class

// Implementation of Vector class here
// Constructors
Vector::Vector(int nElements)
{
  cout << "Hello from Vector::Vector(int nElements)" << endl;
  length = nElements;             // Set array length
  pArray = new int[length];       // Allocate memory for "length" int variables
  // Should we "zero out" the array here?
}
```

Program dynamic-memory2.cc

```

// Copy Constructor
Vector::Vector(const Vector& v)
{
  cout << "Hello from Vector::Vector(const Vector& v)" << endl;
  // This is similar to the "int" constructor, but we get the
  // length from the vector object being copied
  length = v.Length();
  pArray = new int[length];
  // Copy the actual contents
  for (int i = 0; i < length; ++i)
    {
      pArray[i] = v.GetElement(i);
    }
};

// Destructor
// Since the constructors allocated memory with "new", it makes sense
// that the destructor "delete" (give back) the memory.
Vector::~Vector()
{ // Destructor
  cout << "Hello from Vector::~Vector() destructor" << endl;
  delete [] pArray;   // Free the memory previously allocated
}

// Assignment operator
void Vector::operator=(const Vector& rhs)
{
  cout << "Hello from Vector::operator=(const Vector& rhs) assignment" << endl;
  // Assign one vector to another (with the "=" operator)
  // FIRST..VERY IMPORANT, PROTECT AGAINST "Self-Assignment"
  if (&rhs == this) return;  // We will discuss this in class
  // Next delete any memory associated with the left hand side
  delete [] pArray;
  // Set new length
  length = rhs.Length();
  // And allocate the memory
  pArray = new int[length];
  // Copy the actual contents
  for (int i = 0; i < length; ++i)
    {
      pArray[i] = rhs.GetElement(i);
    }
}

// Member functions
int Vector::Length() const
{ // Return the length of the array
  return length;
}

int Vector::GetElement(int whichElement) const
{ // Return the specified element in the array
  // We could add an extra check here to make sure that the specified
  // "whichElement" is valid. This is extra overhead however, so
  // we decide not to do that.
```

Program dynamic-memory2.cc (continued)

```
  return pArray[whichElement];
}

void Vector::SetElement(int whichElement, int newValue)
{ // Set a new value in the array
  pArray[whichElement] = newValue;
}

// The indexing operator
int& Vector::operator[](int whichElement)
{
  // return a reference to the specified element. Since we are returning
  // REFERENCE to an element, we can use the indexing operator either
  // on the left side OR THE RIGHT side of an assignment.
  // See the code in main for an example.
  return pArray[whichElement];
}

// Print for debugging
void Vector::Print() const
{
  for (int i = 0; i < Length(); ++i)
    {
      cout << "Element " << i << " = " << GetElement(i) << endl;
    }
  cout << endl; // Extra end of line to space out the printouts
}

int main()
{
  Vector v1(5);  // Vector with 5 element
  // Set some initial values
  for (int i = 0; i < v1.Length(); ++i)
    {
      v1.SetElement(i, i);
    }
  Vector v2(v1); // A copy of v1
  cout << "Printing v1" << endl;
  v1.Print();
  cout << "Printing v2" << endl;
  v2.Print();
  Vector v3(10);  // Another vector with 10 elements
  // Set some initial values
  for (int i = 0; i < v3.Length(); ++i)
    {
      v3.SetElement(i, i * 10);
    }
  // Assigning v2 from v3
  v2 = v3;  // Assignment operator called
  cout << "Printing v3" << endl;
  v3.Print();
  cout << "Printing v2" << endl;
  v2.Print();
  // Illustrate the indexing operator, both left-hand-side and right-hand-side
  int val1 = v1[4];  // Get index 4 from v1
  cout << "v1[4] is " << val1 << endl;
```

Program dynamic-memory2.cc (continued)

```
  // Set a new value with indexint operator
  v1[4] = 50;  // Note indexing operator on LHS
  cout << "v1[4] is " << v1[4] << endl;
  // Illustrate "self-assignment".  We will discuss this in class
  v1 = v1;   // Clearly not very useful or meaningful, but we need to handle
  // Destructor automatically called for v1, v2 and v3.
}
```

Program dynamic-memory2.cc (continued)