Dynamic Storage Allocation

Allocate an object or array from the heap / free store at runtime. Unlike automatic storage, this dynamic storage remains allocated until explicitly deallocated or the program ends.

Allocating an Array

- Declare a pointer to an element of the desired array type.

```
int* scores;     // Pointer to a dynamic array of integers
```

- Use the `new[]` operator to allocate the desired number of elements for the array. This number can be a variable.

```
scores = new int[numElements];
```

<Diagram of dynamic array goes here>

(Note that the two steps above can easily be combined into a single statement.)

- Each element of an array of built-in types will be initialized to “zero” (0 for numeric types, `false` for `bool`, null character for `char`).

- Each element of an array of objects will be initialized by calling the default constructor for the class.

Using the Array

- Use the pointer name as if it were an array name.

```
scores[2] = 27;

cout << scores[0] << endl;
```

Deallocating the Array

- Use `delete[]` to deallocate the array before the end of the program.

```
delete[] scores;
```

- This only works if the pointer actually points to a valid chunk of dynamic storage or if it contains the special value `nullptr` (address `0x0`). If not, you will get a runtime error.

- When you delete a dynamic array of objects, the class destructor will be called for each element of the array.

Allocating an Object

- Declare a pointer to an object of the desired type:

```
Student* s;     // Pointer to a Student object
```

- Use the `new` operator to allocate the object, passing arguments to a constructor for the class as desired.

```
s = new Student();                          // Calls default
                                            // constructor

s = new Student("Joe Murphy", 3.75);        // Calls alternate
                                            // constructor
```

<Diagram of dynamic object goes here>

- The object is initialized by the constructor that is called.

Using the Object

- You must access the object using the pointer to it.

```
s                // Address of the object pointed to by s

*s               // Value of the object pointed to by s

s->name          // Access a public data member of the object
                 // pointed to by s; could be a private member if
                 // you are in a method of the Student class or
                 // if this code is used in a friend

s->getName()     // Call a public method for the object pointed
                 // to by s
```

Deallocating the Object

- Use `delete` to deallocate the object before the end of the program.

```
delete s;
```

- This only works if the pointer actually points to a valid chunk of dynamic storage or if it contains the special value `nullptr` (address `0x0`). If not, you will get a runtime error.

- When you delete an object, the class destructor will be called for that object.

Pointers to Dynamic Storage as Class Data Members

- A class may have a pointer to dynamic storage as a data member. The storage would typically be allocated in the class constructor.

- Example: A class called `Vector` that serves as a "wrapper" around a dynamic array

```
class Vector
   {
   private:

      int* vArray = nullptr;      // Pointer to dynamic array of
                                  // integers
      size_t vCapacity = 0,       // Number of elements in the
                                  // dynamic array
             vSize = 0;           // Number of items currently
                                  // stored in the dynamic array

      ...

   };
```

- In this example, the dynamic array is initially empty (capacity 0, size 0). As items are added to the vector, we can allocate storage for the dynamic array to accommodate them, increasing the capacity.

<Diagram of Vector object goes here>

- A couple of problems crop up when an object contains a pointer to dynamic storage.

  1. Memory leaks
  2. Shallow copies

Memory Leaks

- When an object is deallocated, the dynamic storage that it "owns" must also be deallocated. This will not happen by default; we need to explicitly deallocate dynamic storage using `delete` or `delete[]`.

- An object typically "owns" any dynamic storage that it allocated. A Vector object will "own" its dynamic array, for example. In more complex linked data structures, an object may point to dynamic storage that it does not own.

- Failure to delete the dynamic storage an object owns leaves it allocated and also inaccessible, because the pointer we could use to access it was part of an object that no longer exists. Once the object is gone, we can't delete the dynamic storage anymore because we don't have a pointer to it. If this happens repeatedly, we will eventually run out of available storage on the heap. This problem is called a "memory leak", because it's like a bucket of water with a hole in the bottom of the bucket. Over time, all of the water will run out of the bucket.

<Diagram of memory leak goes here>

- The solution to this problem is to make sure that when an object is deallocated, the dynamic storage that it “owns” is also deallocated. We can do this by writing a special method for the class called a destructor.

Destructor

- A destructor is a special method that is called just before an object is deallocated. Its usual job is to deallocate any dynamic storage owned by the object.

- A class may have only one destructor. It has no return type, takes no arguments, can't be `const`, and is always named `~ClassName()`.

- Example: The `Vector` class destructor

```
Vector::~Vector()
   {
   delete[] vArray;     // Delete the dynamic array
   }
```

Shallow Copies

- The default process used to copy an object (when it’s passed to or returned from a function / method, when it's assigned, etc.) simply copies the bytes of the object. This is called a “shallow copy”. As long as all of your object’s data is inside the object, a shallow copy is fine.

- However, if an object has a pointer to dynamic storage as a data member, not all of its data is inside the object. The dynamic storage is separate from the object itself, and resides on the heap. A shallow copy will simply copy the pointer to this dynamic storage without actually making a copy of the storage contents. The result is that you end up with two different objects pointing to the same chunk of dynamic storage.

<Diagram of shallow copy goes here>

- In C++, this is a recipe for disaster. Several problems may occur:

  1. Changing the dynamic storage for one object also changes it for the other object. That’s not the behavior we would expect from a copy.
  2. When one of the objects is deallocated, its destructor will deallocate the dynamic storage. The other object is left with a pointer to a chunk of dynamic storage that has been returned to the heap. Using that pointer may result in strange output, cause a segmentation fault, or even corrupt the heap, leading to a runtime error on a future use of new or new[].
  3. When the second object’s destructor tries to delete the (nonexistent) dynamic storage again, you will get a runtime error.

- A shallow copy of an object that points to dynamic storage will **always** result in some type of runtime error in C++.

- The solution to this problem is to make a “deep copy” of the object - a copy of both the object AND the dynamic storage that it points to.

<Diagram of deep copy goes here>

- We can ensure this happens by writing replacements for two methods of our class, the copy constructor and copy assignment operator. The compiler automatically supplies versions of these two methods that make shallow copies. We'll write new versions that make deep copies instead.

Copy Constructor

- Called when a new object is initialized with an existing object of the same class.

- Some examples of when the copy constructor *may* be called:

  1. When a new object is declared and initialized (explicitly or implicitly) with an existing object of the same class.

     ```
     Vector v2(v1);      // Explicit call to copy constructor

     Vector v2 = v1;     // Implicit
     ```

  2. When an object is passed to a function or method by value.

  3. When an object is returned by a function or method by value.

- Other cases are possible.

Logic

1. Copy all non-dynamic data members from other object to corresponding members of new object (the object pointed to by this).
2. If there is any dynamic storage for the other object, allocate the same amount for the new object.
3. Copy the contents of the other object's dynamic storage to the new object.

- Example: The copy constructor for the Vector class

```
Vector::Vector(const Vector& other)
   {
   // Step 1
   vCapacity = other.vCapacity;
   vSize = other.vSize;

   // Step 2
   if (vCapacity > 0)
      vArray = new int[vCapacity];
   else
      vArray = nullptr;

   // Step 3
   for (size_t i = 0; i < vSize; ++i)
      vArray[i] = other.vArray[i];
   }
```

Copy Assignment Operator

- Called when an existing object is assigned to another existing object of the same class, e.g.:

```
v2 = v1;
```

- Because the left-hand-side object already exists, it may already have a dynamic array (but it's probably the wrong capacity). So we will need to delete that existing array to avoid a memory leak.

- We need to make sure that assigning an object to itself does not wreck it! This should work without problems:

```
v2 = v2;
```

- We also need to make sure that we can cascade the assignment operator, e.g.:

```
v3 = v2 = v1;    // Assigns v1 to v2, then assigns v2 to v3
```

Logic

1. Check for self-assignment. If so, skip to last step.
2. Delete the dynamic array for the object that called the method (the one pointed to by `this`).
3. Copy all non-dynamic data members from the `other` object to corresponding members of the object pointed to by `this`.
4. If there is any dynamic storage for the `other` object, allocate the same amount for `this` object.
5. Copy the contents of the `other` object's dynamic storage to `this` object.
6. Return `this` object (`return *this;`)

- Example: The copy assignment operator for the Vector class

```
Vector& Vector::operator=(const Vector& other)
   {
   // Step 1
   if (this != &other)
      {
      // Step 2
      delete[] vArray;

      // Step 3
      vCapacity = other.vCapacity;
      vSize = other.vSize;

      // Step 4
      if (vCapacity > 0)
         vArray = new int[vCapacity];
      else
         vArray = nullptr;
```

```
        // Step 5
        for (size_t i = 0; i < vSize; ++i)
            vArray[i] = other.vArray[i];
        }

    // Step 6
    return *this;
    }
```

Move Semantics

- Move semantics are a C++11 feature designed to cut down on unnecessary allocation and copying of dynamic storage. Instead of creating new storage and copying in the contents of the existing array, we will simply “pilfer” the existing array from the object being used to initialize our new object, or the existing object being assigned to our object.

- That will invalidate the existing object, but if it’s about to be deallocated anyway, that won’t matter. All we need to do is make sure the object won’t cause a crash when its destructor runs.

- You can write a move constructor and move assignment operator to supplement your copy constructor and copy assignment operator. The compiler will call these whenever possible to avoid the costly allocation and copying. If you don’t write them, the copy constructor and copy assignment operator will be used.

Move Constructor

- Example: A move constructor for the `Vector` class

```
Vector::Vector(Vector&& other)    // rvalue reference to a Vector
    {
    // Step 1 - "pilfer" other's resources
    vCapacity = other.vCapacity;
    vSize = other.vSize;
    vArray = other.vArray;

    // Step 2 - set other object to default state
    other.vCapacity = 0;
    other.vSize = 0;
    other.vArray = nullptr;
    }
```

Move Assignment Operator

- Left as an exercise for the student.