Memory Management: Efficient Headers

Where are we? We are talking about memory, and in particular the approach of hiding dynamic memory via header classes. We are creating a vector class as an example. The vector class is like a header that holds the pointer into the heap and auxiliary information. The idea is to hide the fact that the underlying data is on the heap.

We previously started to design of a vector class:

```cpp
class int_vector {
public:
    int_vector(int size = 0);
    ~int_vector();

    int at(unsigned int index) const;
    int& at(unsigned int index);

    void push_back(int value);

private:
    int* data;
    int size;
    int allocated;
};
```

We implemented the constructor to allocate memory, setting the data and size fields. We should now expand it to set the allocated field as well. For starters, it can be the same as the size field.

We implemented the destructor to delete data, set data to NULL, and set size to zero. We should now expand it to set allocated to zero, too.

We ended talking about push_back(int value). The simple version of push_back (which doesn’t use the allocated field) looks like:

```cpp
void int_vector::push_back(int value)
{
    int* new_data = new int[size+1];
    for(int i = 0; i < size; i++) {
        data[i] = new_data[i];
    }
    new_data[size] = value;
    size++;
    delete [] data;
    data = new_data;
}
```

The problem with this is efficiency. If you push N items on the stack, the first item gets copied a total of N-1 times, the second gets copied N-2 times, and so on. This is O(N^2).

We can do better. The idea is that if you have to expand the underlying array, you double its size. This is where the allocated field comes in: the size field is the number of elements in the array, while the allocated field is the amount of memory allocated. Now push_back becomes
void int_vector::push_back(int value)
{
    if (allocated < (size + 1))
    {
        int* new_data = new int[2*allocated];
        allocated *= 2;
        for(int i = 0; i < size; i++) {
            new_data[i] = data[i];
        }
        delete [] data;
        data = new_data;
    }
    new_data[size] = value;
    size++;
}

So let’s summarize:

- In C++, the burden of memory management lies with you. Be systematic about it.
- Local variables are easy. They are deleted when they leave scope. Just don’t return a pointer or a reference to one.
- Local variable have two main limitations:
  - Fixed size
  - Restricted lifespan (need to be copied out)
- A memory management scheme for dynamically sized but short-term data is:
  - Create a header object that holds a pointer to the heap
  - Define a deep destructor
  - Define a deep copy constructor
  - Define an deep assignment operator
  - Use header as local variable
  - Your book calls these “the Big 3”
- Creates data that is dynamically sized and won’t leak memory

This scheme allows dynamic data to be treated like local (stack) data. Unfortunately, it doesn’t solve the persistent data problem.