Dynamic Arrays

Written by Rachel J. Morris, last updated 2016-02-15

Arrays of any size!

Pointers and Arrays

As we've seen in the pointer lecture, we can utilize pointers to allocate memory for new variables...

```
int main()
{
    int* myInteger;
    myInteger = new int;

    (*myInteger) = 30;

    cout << "Address of new integer: " << myInteger << endl;
    cout << "Value of new integer:   " << (*myInteger) << endl;

    delete myInteger; // Free the memory
    myInteger = NULL; // Reset the pointer to no address

    return 0;
}
```

Pointers and Arrays

We can also allocate space to create an array.

Because we're allocating space at run-time,
we no longer need to know the exact size of the array at compile-time.

```
int main()
{
    int amount;
    cout << "How many items? ";
    cin >> amount;

    float* prices = new float[ amount ];

    for ( int i = 0; i < amount; i++ )
    {
        cout << "Enter price for item " << i << ": ";
        cin >> prices[i];
    }

    delete [] prices; // Free the memory
    prices = NULL; // Reset the pointer to no address

    return 0;
}
```

Pointers and Arrays

Static Array (old news)

```
int staticArray[ 10 ];
```

- Must know size of array at compile time.
- Cannot be resized.

Dynamic Array (cool new thing)

```
int* dynamicArray;
dynamicArray = new int[ 10 ];
delete [] dynamicArray;
```

- Can be set to any size!
- Can be deleted then re-created with a new size!

Even though it is more work to deal with dynamic arrays,

not being tied to a maximum value in your program can be pretty powerful!

Creating a dynamic array...

Step 1: Create a pointer variable

```
string* nameList;
```

Step 2: Make a **new** array with the pointer!

```
nameList = new string[ 10 ];
```

Step 3: Use the array like normal!

```
nameList[0] = "Stephanie Shirley";
nameList[1] = "Ada Lovelace";
nameList[2] = "Grace Hopper";
```

Step 4: Delete the array when you're done with it!

```
delete [] nameList;
```

Step 5: Be a happy programmer

Finally!
Resizing our arrays!

Resizing arrays

Resizing a dynamic array isn't as simple as adding on to the end of it, unfortunately.

(we will write data structures later on more like this!)

In order to resize an array, we essentially have to :

(1) make a new, bigger array,
(2) copy the contents of our array over,
(3) delete the old small array
(4) set the pointer to point to the new bigger array.

Resizing arrays

So, let's say we created an array...

```
string* names = new string[ 5 ];
```

...But it filled up quickly.

```
names[0] = "Snake";
names[1] = "Ocelot";
names[2] = "Octopus";
names[3] = "Fox";
names[4] = "Wolf";
```

Now we need more space!

Resizing arrays

We need to create a new array with more space!

```
string* newArray = new string[ 10 ];
```

And we need to copy the values from the smaller array into the larger array!

```
for ( int i = 0; i < 5; i++ )
{
    newArray[i] = names[i];
}
```

Resizing arrays

We don't need the small array anymore, so let's free that memory.

```
delete [] names;
```

But, we still want to keep using the variable name **names**. We don't want to use the variable name **newArray**!

No problem, let's set the **names** pointer to point to the bigger array!

```
names = newArray;
```

No more use for **newArray**, so let's reset it to NULL!

```
newArray = NULL;
```

Resizing arrays

Now we can keep using our **names** array!

```
names[5] = "Raven";
names[6] = "Mantis";
names[7] = "Fortune";
names[8] = "Vamp";
names[9] = "Fatman";

// Output the results! :)
for ( int i = 0; i < 10; i++ )
{
    cout << (i+1) << ": " << names[i] << endl;
}
```

...Any everybody lived happily ever after.

```
delete [] names;
```

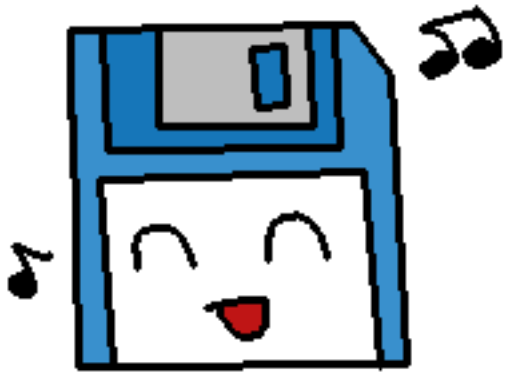
And don't forget to free any allocated memory at the end of the program!!

Too much to remember?
No problem!

Lightening the Load

Q: What is the point of programming?

A: To automate tasks so we don't have to waste our time on it!!



Lightening the Load

Don't want to keep track of **new** and **delete**?

Don't want to keep rewriting resizing logic any time you need a bigger array?

Write a class to “wrap” it all up for you!!

Lightening the Load

Through the magic of classes, you can put everything you need to deal with dynamic arrays in one class!

```
class DynamicString
{
    public:
    DynamicString( int size );
    ~DynamicString();

    void Set( int index, const string& val );
    string Get( int index );

    int Size();

    private:
    string* m_stringList;
    int m_maxVal;
};
```

```
int main()
{
    DynamicString muffins( 5 );

    muffins.Set( 0, "Chocolate" );
    muffins.Set( 1, "Blueberry" );
    muffins.Set( 2, "Lemon" );
    muffins.Set( 3, "Apple" );
    muffins.Set( 4, "Strawberry" );

    for ( int i = 0; i < muffins.Size(); i++ )
    {
        cout << (i+1)
            << "\t"
            << muffins.Get( i ) << endl;
    }

    return 0;
}
```

Sample Code

Let's work with some dynamic arrays to get a feel for 'em!

And we'll make this special wrapper class to deal with Memory Allocation for us!

Sample Programs...

A. Simple Dynamic Array

B. Resizing an Array

C. Array Wrapper Class

Sample Programs...

A. Simple Dynamic Array

B. Resizing an Array

C. Array Wrapper Class

Sample Programs...

A. Simple Dynamic Array

B. Resizing an Array

C. Array Wrapper Class

Sample Programs...

A. Simple Dynamic Array

B. Resizing an Array

C. Array Wrapper Class