Lecture 12 — More About Arrays

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 6, 2016

Acknowledgments: W.D. Bishop

Part I

The String Revisited

The `string` Revisited

We have already learned about a string type, but we haven't examined it in much detail.

The string was just a bunch of text, but there is much more to it than we might think at first glance...

```
string svar1; // Creates uninitialized string
string svar2 = "Literal"; // Initialized.
string svar3 = ""; // Initialized to the empty string.
```

The `string` is a complex type (there's a reason it wasn't introduced in the simple types alongside `int` and `double`).

String Concatenation

Strings can appear in expressions using the + operator. It does not add up the values, but instead performs concatenation.

```
string verb = "fore" + "see";
```

This means the variable `verb` contains `"foresee"`.

The use of += can also be used for concatenation:
`verb += "n";` → `verb` is now `foreseen`.

Remember that for concatenation, like an arithmetic expression, it is necessary to assign the value somewhere.

Strings are Complex

It turns out that the string has a member variable `Length` that tells you the number of characters in the `string`.

This information, combined with the fact that a `string` is a bunch of text characters should lead you to the conclusion that...

The `string` is really an array of `chars`.

An Array of `char`

This means we can access individual characters within a `string` using their index values (just as if they were entries of an array).

If the string is `string ex1 = "example"`, the `char` at `ex1[3]` is `'m'`.

We could use a `for` loop to iterate over all the characters of the `string` if we are looking for something specific.

Iterating Over the `string`

```
string s = "Hello World!";

for ( int i = 0; i < s.Length; i++ )
{
    cout << s[i] << endl;
}

for ( int j = s.Length -1; j >= 0; j-- )
{
    cout << s[j] << endl;
}
```

Mutating `string` Elements

Characters within a `string` can be changed using index values.

This is no different from an array of `int` where we could assign `array[7] = -98;`

The string type is *not* immutable. `string` variables can be changed.

In other languages, the string is immutable; every time a string must be "changed", a new `string` must be created.

Null Termination

So far, when we have used a string, it is either a literal, or taken from console input.

We can also build our own strings programmatically.

It is extremely important that when we do so, we remember to terminate the string correctly!

A string does not have a fixed length in C++ and so we know that a string terminates only by finding the termination character.

The termination character is null, or zero; escape sequence, `'\0'`.

Null Termination

This was a design decision from the days of C where the creators thought it best not to limit the length of the string.

Common error: forgetting to terminate the string with null!

Unfortunately it has led to massive problems including crashes, data corruption, and security breaches.

Thus, the correct way to write a string as a character array is:

```
char hello[6] = 'H', 'e', 'l', 'l', 'o', '\0';
```

Always ensure that a string you create is correctly null terminated.

Part II

Multi-Dimensional Arrays

Arrays of Arrays

You may have wondered if we can have an array of any type, can we have an array of arrays? Yes!

A multi-dimensional array is an array of array types.

The following statement declares and instantiates a multi-dimensional array of `int` named `days`: `int[12][31] days;`

Is this syntax confusing? Perhaps imagine it like this: `(int[])[]`

The type is `int[]` (in brackets) and when we declare an array of a type, write `[]` after the type.

Hence, we declare an array of type `int[]` (integer array).

Arrays of Arrays

`day[0]` is a reference to the first integer array
`day[1]` is a reference to the second integer array
`day[n-1]` is a reference to the nth integer array

The length of `day[0]` and `day[1]` may be different.

Setting up A Multi-Dimensional Array

```
int[] daysInMonth = { 31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31 };

int[12][31] year;

for ( int month = 0; month < 12; ++month )
{
    year[month] = new int[ daysInMonth[month] ];

    for ( int day = 0; d < daysInMonth[month]; ++d )
    {
        year[month][day] = 1;
    }

}
```

Setting up A Multi-Dimensional Array

Now let's print out this calendar.

```
for ( int month = 0; month < 12; ++month )
{
    for ( int day = 0; d < daysInMonth[month]; ++d )
    {
        cout << year[month][day] << " ";
    }
    cout << endl;
}
```

Further Initialization of Multi-Dimensional Arrays

It is possible to initialize a multi-dimensional array when it is declared.

For example, the following code defines a multi-dimensional array of characters named `myChars`:

```
char[][] myChars = {
          { 'B', 'i', 'l', 'l' },
          { 'D', 'a', 'v', 'e' },
          { 'G', 'e', 'o', 'r', 'g', 'e' }
          };
```

Arrays of Arrays of Arrays...

The multi-dimensional array examples we have shown so far are all "two dimensional".

We could have more, such as `int[][][] coordinates;` to describe x, y, and z co-ordinates.

Multidimensional arrays are also called "jagged arrays".

Storage in Memory

Remember that memory is linear, so a two dimensional array has to be stored in a linear fashion.

Therefore, a two dimensional array is just an abstraction.

Thus, the following are equivalent:

```
int[3][3];
int[9];
```

In C++ (and C) arrays are stored in row major order.

Thus, all elements in the first row appear first, then all elements in the second row, etc.