C++ Arrays & Strings

An *array* in C++ (as opposed to the array class that we'll see later in the semester) is a fixed-length sequence of elements that are:

- identical in data type, size, shape and purpose;
- contiguous in memory;
- accessed using an integer *index* whose value lies between zero and the number of elements minus one, inclusive.

The *base type* of an array's elements can be any valid (primitive or user-defined) type or class.

For example, we might have an array named `a` of 50 `int` elements. Assuming that an individual `int` takes up 4 bytes in memory (which is typical these days), the array `a` will take up a total of 200 bytes. Its first element will be `a[0]`, and its last element will be `a[49]` (since its length is 50). If the address of `a[0]` is, say, 13250 (decimal), then the address of `a[20]` is

$$13250 + (20 \cdot 4) = 13330$$

C++ can allocate arrays both at compile time and at runtime. A declaration that allocates an array at compile time, also known as *static* allocation, might look like:

```
int a[50];
```

Note that the length of the array in a compile time allocation **must** be an integer constant, either:

- an integer literal constant, such as:
  ```
  double q[22];
  ```
- a declared integer constant, such as:
  ```
  const int stringLength = 97;
  char myString[stringLength];
  ```
- a preprocessor macro whose text substitution is one of the above, such as:
  ```
  #define MY_FLOAT_ARRAY_LENGTH 32
  #define MY_BYTE_ARRAY_LENGTH  stringLength
  ...
  float highTemperature[MY_FLOAT_ARRAY_LENGTH];
  unsigned char outBuffer[MY_BYTE_ARRAY_LENGTH];
  ```
- an integer-valued expression composed of any combination of the above, such as:
  ```
  float highTemperatureAndThenSome
            [(stringLength + MY_FLOAT_ARRAY_LENGTH) * 2];
  ```

An array can also be allocated at runtime, also known as *dynamic* allocation:

```
int* b;
...
b = new int[bLength];
```

In this case, the value inside the square brackets can be any integer-valued expression, which can include integer variables, function calls, method invocations and so on.

If an array is allocated dynamically, then it must also be deallocated (or *deleted*):

```
delete[] b;
```

Note the square brackets after the reserved word `delete`; these indicate that the variable to be deleted is a C++ array. So, the above statement can be read as

Delete the C++ array named `b`.

A *string* in C++ (as opposed to the string class that we'll see later in the semester) is simply an array whose base type is `char`. The contents of the string is the sequence of characters stored in its elements, up to but excluding the first occurrence of the null character (corresponding to integer value 0). So, the length of the string must be at least one more than the length of the character sequence that it stores. For example:

```
char myFirstName[6] = "Henry";
```

In this case, the elements of `myFirstName` are:

```
myFirstName[0] ≡ 'H';
myFirstName[1] ≡ 'e';
myFirstName[2] ≡ 'n';
myFirstName[3] ≡ 'r';
myFirstName[4] ≡ 'y';
myFirstName[5] ≡ '\0';
```

Note that an individual character literal is delimited by single quotes, while a string literal is delimited by double quotes.

A string can be allocated with more space than it needs for its contents, in which case the extra elements are ignored:

```
char myFirstNameLong[20] = "Henry";
```

In this case, elements 0 through 5 of `myFirstNameLong` are identical to the corresponding elements of `myFirstName`, and elements 6 through 19 have garbage in them and are unused.

Since C++ strings are C++ arrays, they can be allocated dynamically as well as statically, in the same way that arrays are allocated dynamically. For example:

```
char *myLastName;
...
myLastName = new char[7];
```

They should also be deleted in the same way as arrays:

```
delete[] myLastName;
```

Although statically allocated strings can be initialized to a string literal (as in the declaration of `myFirstName`, above), string variables **should not** have string literals assigned to them; for example, this statement should not be used:

```
myLastName = "Neeman";
```

Why? Well, `myLastName` is a pointer to a `char` array, and a string literal is a statically allocated `char` array. Assigning the string literal to `myLastName` means giving it the address of that string literal, which may exist only on the program stack, and which therefore may be wiped out by the program later on in the run. Also, there's nothing to prevent changing the value of a dynamically allocated string, and the compiler sees `myLastName` as a dynamically allocated string, which means that the programmer might end up messing around with parts of memory that shouldn't be disturbed; that could lead to garbage in the string, or crashing the program (or even the whole computer).

Instead, the best way to assign a string literal to a string is to use the standard C++ library function `strdup`:

```
...
#include <string.h>
...
myLastName = strdup("Neeman");
```

The `strdup` function allocates sufficient space for a string and copies the source string into it, then returns the address of the new string (i.e., a pointer to it). So, it's also necessary to delete a string that has been allocated by `strdup`.