Array Basics

Definition

An **array** is an indexed collection of data elements of the same type.
1) **Indexed** means that the array elements are numbered (starting at 0).
2) The restriction of the **same type** is an important one, because arrays are stored in consecutive memory cells. Every cell must be the same type (and therefore, the same size).

Declaring Arrays:

An array declaration is similar to the form of a normal declaration (typeName variableName), but we add on a size:

```
typeName variableName[size];
```

This declares an array with the specified *size*, named *variableName*, of type *typeName*. The array is indexed from 0 to size-1. The size (in brackets) must be an integer literal or a constant variable. The compiler uses the size to determine how much space to allocate (i.e. how many bytes).

Examples:

```
int list[30];            // an array of 30 integers
char name[20];           // an array of 20 characters
double nums[50];         // an array of 50 decimals
int table[5][10];        // a two dimensional array of integers
```

The last example illustrates a two dimensional array (which we often like to think about as a table). We usually think of the first size as rows, and the second as columns, but it really does not matter, as long as you are consistent! So, we could think of the last declaration as a table with 5 rows and 10 columns, for example.

Initializing Arrays:

With normal variables, we could declare on one line, then initialize on the next:

```
int x;
x = 0;
```

Or, we could simply initialize the variable in the declaration statement itself:

```
int x = 0;
```

Can we do the same for arrays? Yes, for the built-in types. Simply list the array values (literals) in set notation { } after the declaration. Here are some examples:

```
int list[4] = {2, 4, 6, 8};
char letters[5] = {'a', 'e', 'i', 'o', 'u'};
double numbers[3] = {3.45, 2.39, 9.1};
int table[3][2] = {{2, 5} , {3,1} , {4,9}};
```

C-style strings

Arrays of type **`char`** are special cases.

- We use **strings** frequently, but there is no built-in string type in the language
- A **C-style string** is implemented as an array of type `char` that ends with a special character, called the "null character".
  - The null character has ASCII value 0
  - The null character can be written as a literal in code this way: `'\0'`
- Every string literal (something in double-quotes) **implicitly** contains the null character at the end

Since character arrays are used to store C-style strings, you can initialize a character array with a string literal (i.e. a string in double quotes), as long as you leave room for the null character in the allocated space.

```
char name[7] = "Johnny";
```

Notice that this would be equivalent to:

```
char name[7] = {'J', 'o', 'h', 'n', 'n', 'y', '\0'};
```

Variations in initializing

Array declarations **must** contain the information about the size of the array. It is possible to leave the size out of the [ ] in the declaration **as long as** you initialize the array inline, in which case the array is made just large enough to capture the initialized data. Examples:

```
char name[] = "Johnny";         // size is 7
int list[] = {1, 3, 5, 7, 9};  // size is 5
```

Another shortcut with initializer sets is to use fewer elements than the size specifies. Remaining elements in an array of type int will default to 0. It is illegal to use a set containing **more** elements than the allocated size.

```
int list[5] = {1, 2};          // array is {1, 2, 0, 0, 0}
int nums[3] = {1, 2, 3, 4};    // illegal declaration.
```

Note: Using initializers on the declaration, as in the examples above, is probably not going to be as desirable with very large arrays.
Another common way to initialize an array -- with a **for** loop:
This example initializes the array numList to {0, 2, 4, 6, 8, 10, 12, 14, 16, 18}.

```
int numList[10];
int i;
for (i = 0; i < 10; i++)
   numList[i] = i * 2;
```

Using Arrays:

Once your arrays are declared, you access the elements in an array with the array name, and the index number inside brackets [ ]. If an array is declared as: typeName varName[size], then the element with index n is referred to as varName[n]. Examples:

```
int x, list[5];         // declaration
double nums[10];        // declaration

list[3] = 6;            // assign value 6 to array item with index 3
cout << nums[2];        // output array item with index 2
list[x] = list[x+1];
```

It would not be appropriate, however, to use an array index that is outside the bounds of the valid array indices:

```
list[5] = 10;         // bad statement, since your valid indices are 0 - 4.
```

The statement above is syntactically **legal**, however. It is the programmer's job to make sure that out of bounds indices are not used. Do not count on the compiler to check it for you -- it will not!

Copying arrays:

If we have these two arrays, how do we copy the contents of `list2` to `list1`?

```
int list1[5];
int list2[5] = {3, 5, 7, 9, 11};
```

With variables, we use the assignment statement, so this would be the natural tendency -- but it is wrong!

```
list1 = list2;  // this does NOT copy the array contents
```

We must copy between arrays element by element. A **for** loop makes this easy, however:

```
for (int i = 0; i < 5; i++)
  list1[i] = list2[i];
```

Simple I/O with strings:

In the special case of strings (null-terminated character arrays), they can be used like normal arrays. Accessing a single array element means accessing one character.

```
char greeting[] = "Hello";
char word1[20];
```

```
cout << greeting[1];  // prints the letter 'e'
cout << greeting[4];  // prints the letter 'o'
```

Strings can also be output and input in their entirety, with the standard input and output objects (cin and cout):
The following line outputs the word "Hello":

```
cout << greeting;
```

Be careful to **only** use this on char arrays that are being used as C-style strings. (This means, only if the null character is present as a terminator).

The following line allows the entry of a word (up to 19 characters and a terminating null character) from the keyboard, which is stored in the array `word1`:

```
cin >> word1;
```

Characters are read from the keyboard until the first "white space" (space, tab, newline, etc) character is encountered. The input is stored in the character array and the null character is automatically appended.

Examples

- arrayinit.cpp -- example of array declarations and initializer sets
- strings.cpp -- initializing and printing c-style strings
- arrayuse.cpp -- a few examples of array usage