Arrays

Summary: An ***array*** is a collection of data stored contiguously in memory. You refer to the elements of the array by using the array name and subscripts, e.g.:

```
blah[0]          blah[1]          blah[2]     ...   blah[n-1]
```

where ***blah*** is the name of this array and the array has ***n*** elements (numbered ***0*** through ***n-1***).

Allocating space for an array: Declare and allocate space for an array like these examples:

```
double numbers[100];
char name[maxNameLength];
```

The first statement above declares an array of 100 double's called *numbers*. The second statement declares an array of characters (hence a *string*) called *name* whose length is whatever value *maxNameLength* has when this statement executes.

Note the second example: in the most modern version of C (called C99), the length of the array can be determined at *run-time*, that is, when the space-allocation statement executes. Be aware that older compilers will NOT allow this and that there are some restrictions on its use.

Alternatively, you can simulate an array using pointers and use *malloc* or one of its cousins to allocate space for the simulated array. This works in all versions of C (not just C99).

Per the above examples, elements of an array must all be of the same type.

Initializing the elements of an array: When you allocate space for an array (as in the above statements), ***the initial values of the array elements are, in general, garbage***. So, you should initialize the array elements yourself.

One way to do so is to list the values when you allocate space for the array, e.g.:

```
float weights[] = {45.7, 900.4, 32.8, 74.3};
char name[] = "Nelson Mandela";
```

You don't need to specify the length of the array when you use this form; the compiler sets it for you to the number of items in the curly-brackets or quotes. For strings initialized this way, the array includes the `'\0'` that terminates the string; thus, the length of the *name* array in the above example is 15, not 14.

However, the usual way to initialize the array elements is to loop through all the elements of the array, setting each one in turn.

For example, you could initialize all the values of a *dieRolls* array to random values between 1 and 6, as in the example to the right.

```
int k;
int length = 1000;
int dieRolls[length];

for (k = 0; k < length; ++k) {
    dieRolls[k] = 1 + rand() % 6;
}
```

Note that ***arrays do not remember their length*** – you are responsible for maintaining that length in a variable and using it as needed.

Arrays

Accessing elements of an array: Access elements of an array using the "subscript" notation:

```
blah[0]          blah[1]          blah[2]     ...   blah[n-1]
```

where ***blah*** is the name of this array and the array has ***n*** elements (numbered ***0*** through ***n-1***).

Often, we use a loop and an ***index variable*** to go through all the elements of the array, like this:

```
for (k = 0; k < length; ++k) {
    printf("%f\n", temperatures[k]);
}
```

Loop starts at 0 and goes up to (but NOT including) *length*.

The index variable in the example is *k* – note that it starts at 0 and goes up to (but NOT including) *length*. Note the use of **`temperatures[k]`** inside the loop – very typical.

See the Gotcha's (common errors) for what can happen when you access an array element with a subscript whose value is outside of the bounds of the array.

See Array Patterns for more array patterns and examples.

Sending an array to a function, as a parameter:
Here is an example:

```
void printArray(float numbers[], int length) {
    int k;

    for (k = 0; k < length; ++k) {
        printf("%f\n", numbers[k]);
    }
}
```

Indicate that the parameter is an array by putting square brackets after the array name.

Don't put anything inside the square brackets – the compiler ignores anything there, so putting something there only misleads the reader.

If you will loop through the array in the function, you need to pass the array's length, as in this example.

You use the array using the usual bracket notation, even though this is really a pointer simulating an array.

```
int lengthOfWeightsArray = 400;
float weights[lengthOfWeightsArray];
   ...
printArray(weights, lengthOfWeightsArray);
```

Pass the array by putting its name as the argument, WITHOUT brackets. The array name is really a pointer.

Two-dimensional arrays: A ***two-dimensional array*** can be thought of as a grid of rows and columns and is sometimes called a ***matrix***. Here are examples:

```
float blah[100][50];
```

Note the loop-within-a-loop pattern for processing all the elements of the two-dimensional array.

This example assumes that `nRows` is the size of the first dimension (100 in the above example declaration) and `nColumns` is the size of the second dimension (50 in the above example declaration).

```
for (j = 0; j < nRows; ++j) {
    for (k = 0; k < nColumns; ++k) {
        printf("%f\n", blah[j][k]);
    }
}
```

Use `blah[j][k]`
NOT `blah[j, k]`

Pass two-dimensional arrays to functions as in this example:

```
void printArray1(float blah[][50], int nRows) (
    ...
}


void printArray2(int nRows, int nCols, float blah[nRows][nCols]) (
    ...
}
```

IMPORTANT: You MUST specify the size of the second dimension (50 in this example).
For multi-dimensional arrays, you must specify the size of all the dimensions except for the first.

In the most modern version of C (called C99), the sizes of the dimensions can be parameters, as shown here. But:

1. You must declare the size parameters BEFORE (i.e., to the left of) their use inside the brackets, as shown here.
2. Be aware that older compilers will NOT allow this.

Using a pointer to simulate an array: Applications of this include:

- Setting the length of the array at *run-time*, that is, when the space-allocation statement executes, even if you have an older compiler (pre-C99).
- Two-dimensional "ragged arrays" in which the lengths of the rows vary from row to row (this is especially handy for inputting and storing lines of text).
- Passing an array to a function.

See Using a Pointer to Simulate an Array in Using Pointers to Save Time and Space for details.

For further discussion about arrays, see:

- Gotcha's (common errors when using arrays).
- Some common Array Patterns.
- A Discussion that is a more complete version of this document.
- Some Practice Problems on arrays.

Also, if you are curious to learn more about arrays (and only if you are curious – the material in this tutorial covers all of the common issues with arrays), you can also see lots of details about arrays and their curious relationship to pointers in:

http://c-faq.com/aryptr/index.html