Arrays

Discussion: 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***).

This document explains how to:

1. Allocate space for an array
2. Initialize the elements of an array
3. Access the elements of an array
4. Send an array to a function, as a parameter
5. Use two-dimensional arrays.
6. Use a pointer to simulate an array.

Also see these other documents:

- Some Gotcha's (common errors when using arrays)
- Some common Array Patterns
- Some Practice Problems on arrays

1. Allocating space for an array: Declare and allocate space for an array with the same *type/name* pattern that you use for declaring other variables, but:

- Indicate that it is an array by appending square brackets to the name, and
- Specify the length of the array by putting the length inside the square brackets.

For example:

```
double numbers[100];

char name[500];

Cat cats[10];

Dog* dalmations[101];

int ages[length];
```

100 numbers, each of type *double*

500 characters (hence room for a *string* of up to 499 characters plus the '\0' that terminates a string

10 instances of the Cat structure

101 pointers, each capable of pointing to an instance of the Dog structure

As many numbers, each of type *int*, as the value of variable *length*. For this to work correctly, the *length* variable must have a non-garbage value when this statement is encountered.

Note the last 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.

2. **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;
}
```

Or, you could ask the user for the length of the array, then for the initial values of the elements of the array, as in the example below:

```
int length, k;

printf("How many temperatures will you enter? ");
fflush(stdout);
scanf("%i", &length);

double temperatures[length];

for (k = 0; k < length; ++k) {
    printf("Enter a temperature: ");
    fflush(stdout);
    scanf("%lf", &(temperatures[k]));
}
```

This statement cannot be placed earlier in the code – it must appear only after the variable *length* has been assigned a meaningful value.

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

3. 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, as in both of the examples in the previous section. Here is another example, assuming that *length* is indeed the length of the array.

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

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

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.

4. Sending an array to a function, as a parameter: When you send an array to a function, the compiler actually sends a ***pointer*** whose value is the address of the first element in the array. This has three important consequences:

1. The function call is fast – even with an array of many thousands of elements, only a *single* thing (the address of the first element in the array) is copied and sent to the function.
2. The function can modify the elements of the array, and those modifications will still be in effect after control returns to the caller.
3. Usually, you also pass the *length* of the array to the function, since arrays do not know their length.

You don't need to know anything else about pointers to pass arrays to functions, since the compiler allows you to continue to use array notation (square brackets), as in this example (on the next page):

```
void printArray(float myArray[], int length);
void sortArray(float myArray[], int length);

int main() {
      int lengthOfWeightsArray = 400;
      float weights[lengthOfWeightsArray];

      int lengthOfVolumesArray = 150;
      float volumes[lengthOfVolumesArray];

        // Code here would set the values of the
        // above arrays, from a file or whatever.

      sortArray(weights, lengthOfWeightsArray);
      printArray(weights, lengthOfWeightsArray);

      sortArray(volumes, lengthOfVolumesArray);
      printArray(volumes, lengthOfVolumesArray);

      return EXIT_SUCCESS;
}

// Sorts the given array from smallest to largest.
void sortArray(float myArray[], int length) {
      int j, k, indexOfMin;
      float temp;

      for (j = 0; j < length - 1; ++j) {
            indexOfMin = j;
            for (k = j + 1; k < length; ++k) {
                  if (myArray[k] < myArray[indexOfMin]) {
                        indexOfMin = k;
                  }
            }
            temp = myArray[j];
            myArray[j] = myArray[indexOfMin];
            myArray[indexOfMin] = temp;
      }
}

// Prints the given array.
void printArray(float myArray[], int length) {
      int k;

      for (k = 0; k < length; ++k) {
            printf("%f\n", myArray[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.

Pass the array by putting its name as the argument, WITHOUT brackets.

As in the prototypes above, indicate that the parameter is an array by putting square brackets after the array name, WITHOUT anything inside the brackets.

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

Use ordinary array notation (square brackets with the index inside them) inside the function.

5. **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 to show:

- How to declare a two-dimensional array:

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

- How to access elements in a two-dimensional array:

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]`

- How to send a two-dimensional array to a function, as a parameter:

```
void printArray1(float blah[][50], int nRows) (
    int j, k;

    for (j = 0; j < nRows; ++j) {
        for (k = 0; k < 50; ++k) {
            printf("%f\n", blah[j][k]);
    }
}

void printArray2(int nRows, int nCols, float blah[nRows][nCols]) (
    int j, k;

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

// Function calls (e.g. in main):
    printArray1(blah, 100);
    printArray2(100, 50, blah);
```

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. If you are curious why this is so, see http://c-faq.com/aryptr/pass2dary.html

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.

6. Using a pointer to simulate an array:

We saw in the above section on Allocating space for an array that the most modern version of C (called C99) allows "Variable Length Arrays" (VLA's). In a VLA, the length of the array can be determined at *run-time*, that is, when the space-allocation statement executes.

However, if:

- you have an older compiler (pre-C99), or
- you are in one of the (infrequent) circumstances in which VLA's are not allowed in C99, or
- you need an array-like object that can change its length as execution progresses,

then the VLA solution is not available. Instead, the solution is to ***simulate an array using a pointer***.

See Using a Pointer to Simulate an Array in Using Pointers to Save Time and Space for how to do this. The discussion there also explains two applications of using pointers to simulate arrays:

- 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.

For further discussion about arrays, see:

- Gotcha's (common errors when using arrays).
- Some common Array Patterns.
- A Summary of this document.
- Some Practice Problems on arrays.

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