Arrays

Gotcha's when using arrays: Avoid these hard-to-debug errors!

1. Treating `blah[1]` (instead of `blah[0]`) as the first (beginning) element of the array, e.g.:

```
for (k = 1; k < length; ++k) {
    ... temperatures[k] ...
}
```

Should be 0, for the loop to go through the entire array as intended.

2. Accessing an array with a subscript whose value is outside of the bounds of the array, for example:
   - Using `blah[k]` when `k` has not been assigned a value (hence is garbage), e.g.:

```
int k;
double blah[10];

blah[k] = 0;
```

The author of this code meant to loop through the 10 elements of the array, setting each to 0, but the author forgot to include the loop.

   - Using a loop that goes one past the end of the array, e.g.

```
for (k = 0; k <= length; ++k) {
    ... temperatures[k] ...
}
```

Should be < (not <=).

When you access an array with a subscript whose value is outside of the bounds of the array, ***any of the following are possible***:

- ***The program runs but with a garbage value*** (often 0) for the array element.
- ***The program crashes*** (because the space outside the array that was accessed is system memory).
- ***Another variable in the program changes its value mysteriously!***
- ***Behavior oscillates among the above, seemingly at random!***

For example, consider the function to the right. What do you think that the function should print?

```
void arrayOutOfBounds() {
    int a;
    int foo[1];

    a = 1;
    foo[1] = 99;

    printf("%i\n", a);
}
```

It looks like it should print `1`, since that appears to be the value to which `a` is set. But when I ran the above, it printed `99` for `a`, not `1`!

Here's why: The compiler happened to store the variable `a` immediately after the array `foo`. The declaration of the array allocates space for only one integer, to be accessed as `foo[0]`. So the assignment

```
foo[1] = 99;
```

accesses the place right after the place for `foo[0]`, which happens to be where variable `a` is stored. So the above statement changes the value of `a`!

Even worse, suppose that you modify the program, changing the

```
int a;
```

statement to

```
int a, b;
```

and making no other mention of `b` in the program. You would expect that this modification would have no effect on the program (since `b` is not used at all), but now the program will probably print the correct value for `a` (namely, `1`), because the out-of-bounds array access happens to access `b` now, not `a`.

Bottom line:

- Try hard to avoid these out-of-bounds array accesses, since they are so hard to debug.
- ***If your program behaves in ways that seem "impossible", suspect that somewhere in your program you have accessed an array out of bounds*** (or misused a pointer, which is a similar error).

3. Forgetting to store the length of the array, or storing it with a wrong value.

4. Forgetting to send the length of the array to a function that needs it, or putting an integer inside the brackets in an array that is sent to a function, e.g.:

```
void printArray(float myArray[15]) {
     int k;

     for (k = 0; k < 15; ++k) {
          printf("%f\n", myArray[k]);
     }
}
```

The author of this code appears to think that writing:

```
myArray[15]
```

here indicates that the array has length 15, but in fact the 15 is ignored.

Hence the loop might go beyond the end of the array or might go through only a portion of the array, depending on what the real length of the array is.

When you send an array to a function, you are really sending a *pointer* whose value is the address of the first element in the array. **Arrays do not know their length.**

5. When allocating space for an array using a variable for the array's length, putting the statement that allocates space *before* the *length* variable gets it correct value, e.g.:

```
int length, k;
double temperatures[length];

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

Space for the array is (incorrectly) allocated here, when *length* still has a garbage value. This statement should be placed after the *scanf* below that sets the value for *length*.

6. Not allocating enough space for the `'\0'` that terminates a string. For example:

```
int a = 999;
char blah[4];

strcpy(blah, "oops");

printf("%i\n", a);
```

Putting 4 "real" characters into a string requires a string array whose length is 5 (not 4).

The **`blah`** string has space for only 3 real characters, given that it needs a `'\0'` to indicate the end of the string. The **`strcpy`** copies 4 real characters (they fit fine), plus a `'\0'` that extends beyond the **`blah`** array. This is an out-of-bounds error and may result in any of the behaviors described in Gotcha #2 above. When I ran this code, it overwrote the **`a`** variable and printed **`0`** (not **`999`**).

7. Using *strcpy* (or any of its cousins) when you don't know the lengths of its arguments. For example, you must be very careful when using statements like this:

```
strcpy(blah, whatever);
```

- This is dangerous if you don't know the length of the *blah* string array – it might not have space enough for the copy of the *whatever* string, as in the previous example.
- It is also dangerous if you don't have control over the length of the *whatever* string – its copy might not fit in the space allocated for the *blah* string. An extreme example of this is when the *whatever* string doesn't have a terminating `'\0'` (so *strcpy* thinks that its length is infinite).

In either case, if the copy of *whatever* doesn't fit in the space allocated for *blah*, the *strcpy* function will blithely continue beyond the bounds of the *blah* array. This is an out-of-bounds error and may result in any of the behaviors described in Gotcha #2 above. Many viruses take advantage of this "buffer overrun" phenomenon, especially when the *whatever* string is an input to the program.

The *strncpy* function (and its cousins) is sometimes helpful in such situations.

8. Explicitly specifying the length of the array when you specify the initial contents of the array in the same statement, e.g.:

```
char x[] = "one";

char y[3] = "one";
```

This is correct – let the compiler determine the length of the array (which is 4, not 3, since the compiler inserts the terminating `'\0'` character.)

This is WRONG – it sets the length of the array to 3, which means that the terminating `'\0'` is not placed into the array. The result is that the standard string functions will not operate correctly on *y*.

9. Returning a statically-allocated array from a function, e.g.:

```
char* wrongWayToReturnAnArray() {
    char foo[100];
        ...
    return foo;
}
```

If you use this function, e.g. with the statement:

```
char* name = wrongWayToReturnAnArray();
```

the *name* array may have its contents corrupted at any point in the subsequent execution.

If you want to return storage for an array (more precisely, for a pointer that is simulating an array), you must use *malloc* or one of its cousins. See how to simulate an array using pointers in Using Pointers to Save Time and Space for more details.

10. “Off by 1” errors.

For example, consider the following code, which attempts to determine whether an array is a palindrome (i.e., whether or not it reads the same backwards as forwards). It has two “off by 1” errors, shown in *red*.

```
int k;
int isPalindrome = 1;

for (k = 0; k < (length – 1) / 2; ++k) {
    if (blah[k] != blah[length - k]) {
        isPalindrome = 0;
        break;
    }
}

if (isPalindrome == 1) {
    ...
} else {
    ...
}
```

Should be `k <  length / 2`

Should be `blah[length – 1 – k]`

The best way to avoid such “off by 1” errors is to trace the code on one or more small, concrete examples. For example, here you should try arrays of length 4 and 5. (You should do both odd and even lengths because of the division by 2 in the `for` statement.) The examples would both expose the errors and show you how to correct them.

11. Abusing two-dimensional arrays, or arrays of pointers, or pointers to an array, or pointers used to simulate an array.

There are lots of ways to make these sorts of errors. I’ll point out just two:

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

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

WRONG: You MUST specify the size of the second dimension, e.g. `float blah[][100]`.

WRONG: Using a variable to specify the second dimension of a two-dimensional array is allowed in the most modern version of C (called C99). However, if you do so, the declaration of that variable (`nCols` in this example) must precede its use as a dimension, like this:
`void printArray2(int nRows, int nCols, float blah[nRows][nCols]) (`