Arrays

Array Patterns: Here are some common patterns that use arrays. In all the examples, assume that the array is called *blah*; that it has length given by the variable *length*; and that the array holds numbers, except where indicated.

Follow the link to go straight to that pattern.

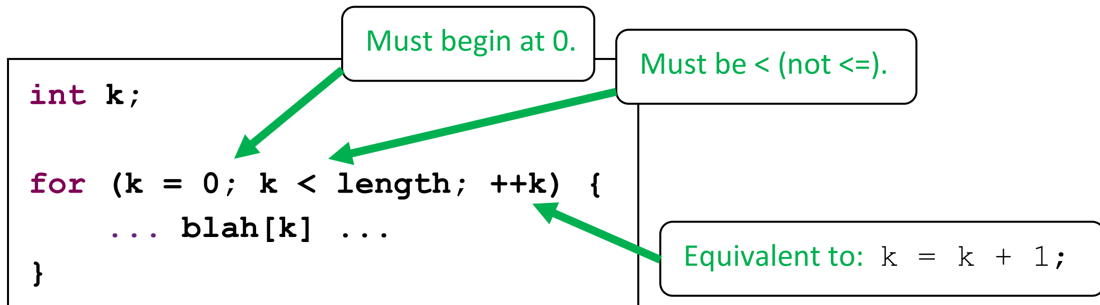
1. Accessing the kth element of the array, for a given *k*.
2. Printing (or otherwise processing) all the values of an array, from beginning to end.
3. Printing (or otherwise processing) all the values of an array, backwards.
4. Counting the number of elements in the array that meet a given condition.
5. Summing the elements in an array of numbers.
6. Finding the largest (or smallest) number in an array of numbers.
7. Finding a given element in an array, or an element that meets a given condition.
8. Initializing the elements of an array:
    a. Initializing the elements of an array to random values.
    b. Initializing the elements of an array to values that are a function of the index.
    c. Initializing the elements of an array to values input by the user (or from a file), asking the user how big the array should be.
    d. Initializing the elements of an array to values input by the user (or from a file), with a sentinel loop.
9. Patterns that refer to another element while processing the current element, e.g.:
    a. Determining whether an array is sorted from smallest to largest.
    b. Determining whether an array is a palindrome.
10. Loop-within-a-loop array patterns, e.g.:
    a. Printing (or otherwise processing) all the values of a two-dimensional array.
    b. Selection sort.
11. The sort-first pattern, e.g.:
    a. Finding the median value of an array.
    b. Binary search.
12. The "histogram pattern" for counting the occurrences of "events", e.g. to find the *mode* (most common number) of an array.
13. Using a pointer to simulate an array.

1. **Accessing the kth element of the array, for a given *k*.** Just use:

```
blah[k]
```

No loop is necessary to access this single element.

2. **Printing (or otherwise processing) all the values of an array, from beginning to end.**

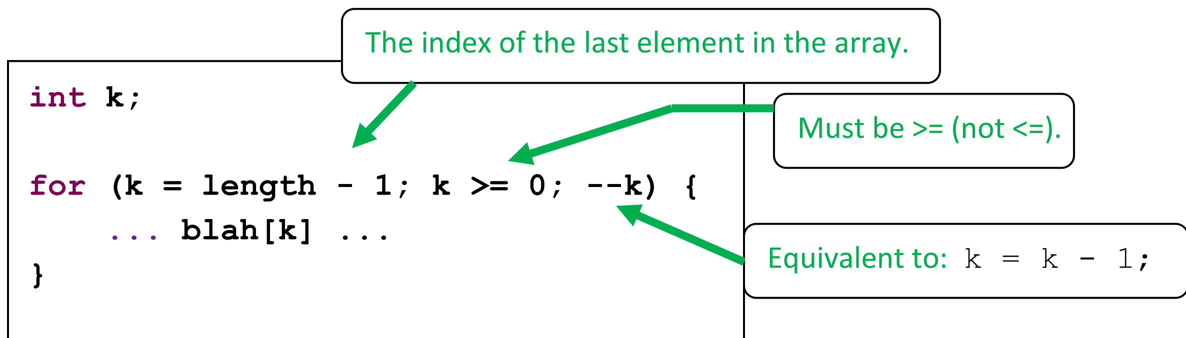



```
int k;

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

Many of the following patterns are special cases or variations of this general “processing” pattern.

3. **Printing (or otherwise processing) all the values of an array, backwards.**




```
int k;

for (k = length - 1; k >= 0; --k) {
    ... blah[k] ...
}
```

4. **Counting the number of elements in the array that meet a given condition.**

```
int k;
int count;

count = 0;

for (k = 0; k < length; ++k) {
    if (blah[k] >= 5 && blah[k] <= 8) {
        ++ count;
    }
}
... count ...
```

Don't forget to start the *count* variable at 0.

The condition (what is inside the if clause) will vary from problem to problem. This example counts the number of elements in the array that are between 5 and 8, inclusive.

After the loop concludes, the *count* variable contains the number of elements in the array that met the condition.

5. **Summing the elements in an array of numbers.**

```
int k;
double sum;

sum = 0;

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

If you are summing integers, the type of *sum* should be *int*, not *double*.

Don't forget to start the *sum* variable at 0.

Equivalent to: `sum = sum + blah[k];`

After the loop concludes, the *sum* variable contains the sum of the numbers in the array.

To sum just the elements that meet a given condition, use an *if* as in the *counting* example above.

6. **Finding the largest (or smallest) number in an array of numbers.**

```
int k;
int indexOfLargest = 0;

for (k = 0; k < length; ++k) {
    if (blah[k] > blah[indexOfLargest]) {
        indexOfLargest = k;
}
... blah[indexOfLargest] ...
```

Start by assuming that the index of the largest element in the array is element 0.

If the element at index *k* is bigger than the largest number found so far, then change the index of the largest number found so far to *k*.

After the loop concludes, the *indexOfLargest* variable contains the index of the largest number in the array.

Change the > to < to find the *smallest* number in the array.

If it is possible that the array has no elements, you'll need to make a special check for that. Also, modifications are necessary if you want the indices of *all* the largest elements (including ties).

7. **Finding a given element in an array, or an element that meets a given condition.**

```
int k;
int indexOfElement = -1;

for (k = 0; k < length; ++k) {
    if (blah[k] == whatYouAreLookingFor) {
        indexOfElement = k;
        break;
    }
}

if (indexOfElement == -1) {
    ...

} else {
    ... indexOfElement ...
}
```

Start by setting *indexOfElement* to an impossible value.

Break out of the loop if you find what you want.

The condition (what is inside the if clause) will vary from problem to problem. This example finds the first element that equals *whatYouAreLookingFor*.

After the loop concludes, if *indexOfElement* is still -1, then the array does not contain an element that meets the given condition. Do whatever the problem calls for in that case.

Otherwise, *indexOfElement* is the index of the first element in the array that meets the given condition. Do whatever the problem calls for in that case.

Modifications are necessary if you want the indices of *all* the elements that meet the given condition. Binary search is a much faster method for finding a given element in an array if the array is already sorted.

8. **Initializing the elements of an array**

a. **Initializing the elements of an array to random values.**

For example,
you can set 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;
}
```

b. **Initializing the elements of an array to values that are a function of the index.**

For example, you can set the kth entry of a *squares* array to k^2, as in the example to the right:

```
int k;
int length = 400;
int squares[length];

for (k = 0; k < length; ++k) {
    squares[k] = k * k;
}
```

c. **Initializing the elements of an array to values input by the user (or from a file), asking the user how big the array should be.**

```
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.

d. **Initializing the elements of an array to values input by the user (or from a file), with a sentinel loop.**

```
int k, arrayLength, actualLength;

arrayLength = 100;
double numbers[arrayLength];
double number;

actualLength = 0;
while (1) {
    printf("Enter a floating-point number: ");
    fflush(stdout);
    scanf("%lf", &number);

    if (number < -10000) {  // Sentinel reached, stop input
        break;
    }

    if (actualLength == arrayLength) {  // Array length reached, stop input
        printf("Sorry, I cannot hold more than %i numbers.\n", arrayLength);
        break;
    }

    numbers[actualLength] = number;
    ++ actualLength;
}
```

In this pattern, you don't know in advance how many numbers the user will input. So, your only choice is to pick an array size that you think will be big enough – 100 in this example.

In this example, the sentinel input (that is, the special input by which the user indicates that she is finished inputting numbers) is any number less than 10,000.

After breaking out of the loop, the array contains the numbers that the user inputted. The variable *actualLength* holds the number of inputs, hence is treated as the length of the array in the subsequent code.

This version of this pattern chooses an array length that the programmer hopes will be "big enough" in practice. This is not a very robust solution! A better (but more complicated) solution would be to use pointers to simulate the array and dynamically (i.e., at run-time) "grow" the array as needed using *malloc* or *realloc* – see How to Use Pointers to Save Time and Space.

9. **Patterns that refer to another element while processing the current element.**

The pattern above for "finding the largest (or smallest) number in an array of numbers" is an example of this pattern – the pattern loops through the array from beginning to end, and at each iteration compares the current element (`blah[k]`) to the largest element found so far (`blah[indexOfLargest]`).

Another example of this pattern is the code to the right, which determines **whether an array is sorted from smallest to largest**. It does so by looping through the array from beginning to end, comparing the current element to the next element in the array.

```
int k;
int isSorted = 1;

for (k = 0; k < length - 1; ++k) {
    if (blah[k] > blah[k+1]) {
        isSorted = 0;
        break;
    }
}

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

Start by setting *isSorted* to 1 (which means "true").

Here we have found a pair of elements that are NOT in the right order, so set *isSorted* to 0 ("false") and quit the loop.

After the loop concludes, if *isSorted* is still 1 (true), then every element of the array is no bigger than the element after it, so the array is sorted from smallest to largest. Do whatever the problem calls for in that case.

Otherwise, the loop found an element of the array that is smaller than the element after it, so the array is NOT perfectly sorted from smallest to largest. Do whatever the problem calls for in that case.

One more example of this pattern is the code to the right, which determines **whether an array is a palindrome** (i.e., whether the array reads the same backwards as forwards). Here, the current element is compared to its mate that is the same distance from the end that the current element is from the beginning.

```
int k;
int isPalindrome = 1;

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

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

Start by setting *isPalindrome* to 1 (which means "true").

Here we have found a pair of matching elements that are NOT the same, so set *isPalindrome* to 0 ("false") and quit the loop.

Being ***"off by 1"*** is a common error in problems like this, for example, using
    `blah[length - k]`
instead of
    `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 `length / 2` in the `for` statement.)

10. **Loop-within-a-loop array patterns.**

One classic example here is ***printing (or otherwise processing) all the values of a two-dimensional array***:

```
int nRows = 100;
int nColumns = 100;
float blah[nRows][nColumns];

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

Another classic example of the loop-within-a-loop pattern is ***selection sort***, an algorithm for sorting an array. Selection sort is simple to understand/code and a good example of a loop-within-a-loop pattern. However, be aware that there are much faster sorting algorithms (e.g. quicksort) and that the standard C library includes a sort function (qsort).

```
int j, k, indexOfMinimum;
float temp;

for (j = 0; j < length - 1; ++j) {

    indexOfMinimum = j;
    for (k = j + 1; k < length; ++k) {
        if (blah[k] < blah[indexOfMinimum]) {
            indexOfMinimum = k;
        }
    }

    temp = blah[j];
    blah[j] = blah[indexOfMinimum];
    blah[indexOfMinimum] = temp;
}
```

Repeatedly:

Find the smallest remaining element in the array.

Put that element just after the other elements that you have already placed into their sorted order.

So the first time through the outer loop, you put the smallest element into **blah[0]**. The second time through the outer loop, you put the second-smallest element into **blah[1]**. And so forth.

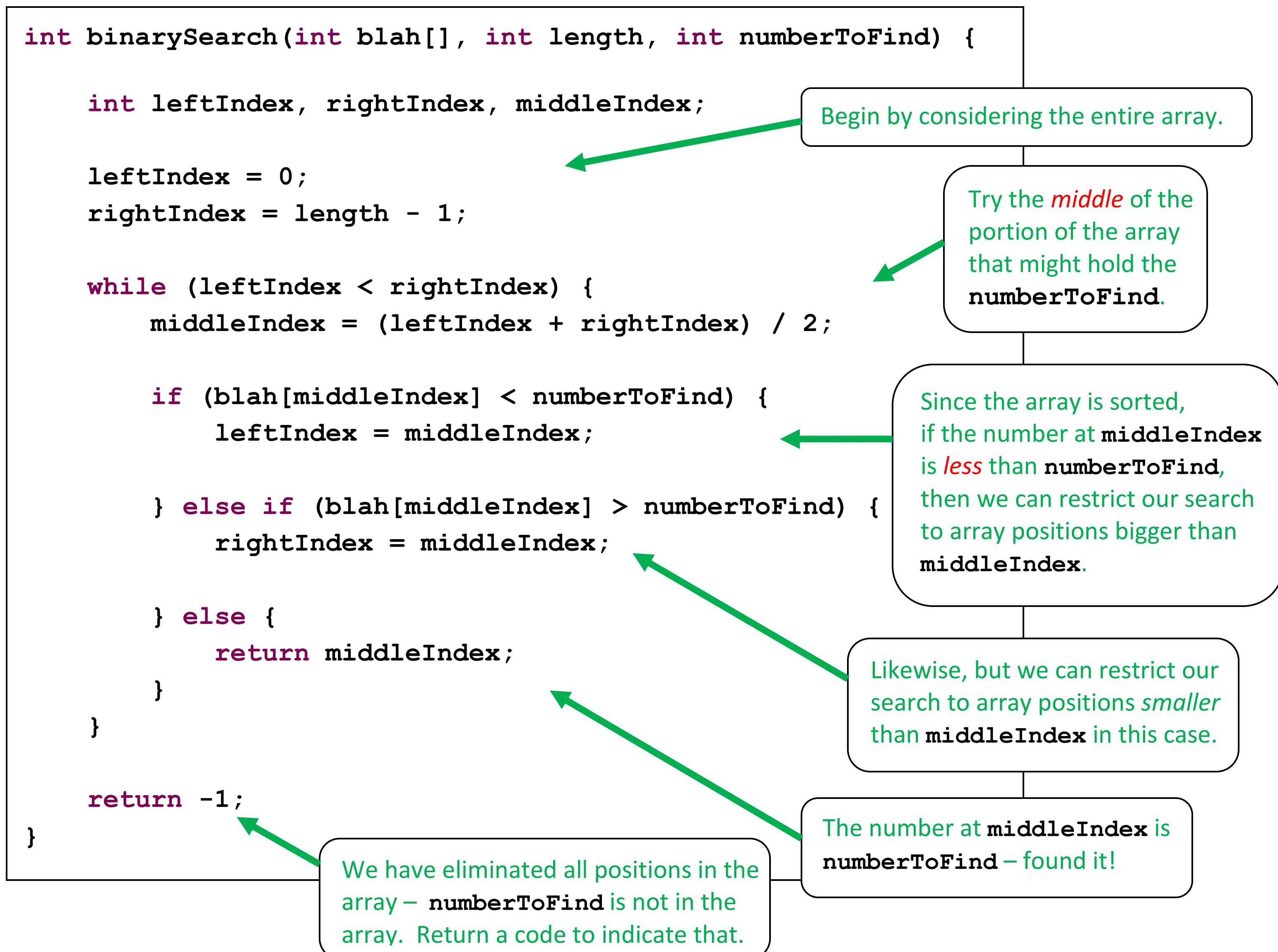
11. **The sort-first pattern.**

Some operations are easier to perform on a sorted array than an unsorted one. For example, one way to find the ***median*** of an array is:

- Step 1: Sort the array (using selection sort, per the previous pattern, or some faster sorting method).
- Step 2: The median is the middle element of the sorted array if the array length is odd, else it is the average of the two closest-to-middle elements in the sorted array.

Be aware that there are more sophisticated methods for finding the median of an array that are slightly faster than the sort-first approach of this pattern.

Another example of this sort-first pattern is searching for a given element: if the array is sorted, ***binary search*** will usually find the element (or determine that it is not in the array) much faster than the linear search pattern presented earlier. It is especially useful if you can sort once, then search many times. Here is the code for binary search:



```
int binarySearch(int blah[], int length, int numberToFind) {

    int leftIndex, rightIndex, middleIndex;

    leftIndex = 0;
    rightIndex = length - 1;

    while (leftIndex < rightIndex) {
        middleIndex = (leftIndex + rightIndex) / 2;

        if (blah[middleIndex] < numberToFind) {
            leftIndex = middleIndex;

        } else if (blah[middleIndex] > numberToFind) {
            rightIndex = middleIndex;

        } else {
            return middleIndex;
        }
    }

    return -1;
}
```

12. **The "histogram pattern" for counting the occurrences of "events"**, e.g. to find the **mode** (most common number) of an array.

    This pattern is useful whenever you want to know how often an "event" occurs, when you have a collection of events that can be numbered into a relatively small range. The general pattern goes like this:

    - Step 1: Declare an array that will hold values 0 .. *n*, where "events" are numbered and *n* is the biggest event that can occur.
    - Step 2: Initialize all elements of the array to 0.
    - Step 3: Repeatedly:
        - Get the next "event" in the collection of events from which the histogram is to be built. That event's number is, say, *m*.
        - Increment the array value at index *m*.

    After Step 3 completes, the array holds the desired histogram – the value at index *i* is the number of times that event *i* occurred (hence the array is a histogram).

    Here is a concrete example of this pattern: Suppose that you generate 10,000 random integers, each between 0 and 100, inclusive, and suppose that you want to know ***which random integer occurred most often***. The following code solves this problem.

Finding the mode of an array of 10,000 numbers, each of which is between 0 and 100, works just like this, except that you loop through the array here instead of generating the numbers.

```
int k, generatedNumber, indexOfMaximum;
int numbers[101];

for (k = 0; k < 101; ++k) {
    numbers[k] = 0;
}

for (k = 0; k < 10000; ++k) {
    generatedNumber = rand() % 101;
    ++ numbers[generatedNumber];
}

indexOfMaximum = 0;
for (k = 1; k < 101; ++k) {
    if (numbers[k] > numbers[indexOfMaximum]) {
        indexOfMaximum = k;
    }
}

printf("%i occurred most often. It occurred %i times.\n",
                indexOfMaximum, numbers[indexOfMaximum]);
```

There are 101 "events" – the randomly generated numbers can be any integer from 0 to 100, inclusive.

Initially, none of the events have occurred, so initialize all the entries in the histogram array to 0.

Generate the 10,000 random numbers. Each time, increment the histogram array at the index of the generated number (i.e., at the index of the "event".

Find the largest number in the histogram array – that is the event that occurred most often.

13. **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.