17. 1D Arrays

Objectives

- Declare an array
- Declare and initialize an array with or without a size
- Access an element of an array using the `[]` operator
- Loop over the elements of an array

Subscripts

Suppose you have the following code:

```
int x = 0, y = 0, z = 0;
std::cin >> x;
std::cin >> y;
std::cin >> z;
```

Note that you're doing the *same thing* to three variables. There's a C++ concept that allows you (more or less) to say

loop through variables x, y, z and perform input for each

This sure looks like a loop.

To implement this idea, C++ uses the concept of a subscript. You have already seen this in your math classes: Suppose one of your math instructors has variables x_0, x_1, x_2, and x_3. He/she might write the following on the board: “For each x_i with i = 0, 1, 2, 3, we want to let x_i be 42.”

It's the same idea. Your instructor wanted to do something to all the variables x_0, x_1, x_2, and x_3. But instead of saying “Set x_0 to 42”, “Set x_1 to 42”, “Set x_2 to 42”, “Set x_3 to 42”, your math instructor was smart enough to say:“For each x_i with i = 0, 1, 2, 3, we want to set x_i be 42”. He/she is basically using subscripts.

C++ supports the same way of applying operations through a bunch of variables, and not just one.

That's the concept of an ***array*** of values.

Declaring arrays

We have seen many types: `int`, `double` (and `float`), `bool`, and `char`. These are called basic (or primitive) types. There are others.

Now we want to talk about creating a variable which is made up of a **collection** of **values** of the **same type**. In other words such a variable contains more than one value.

Try this:

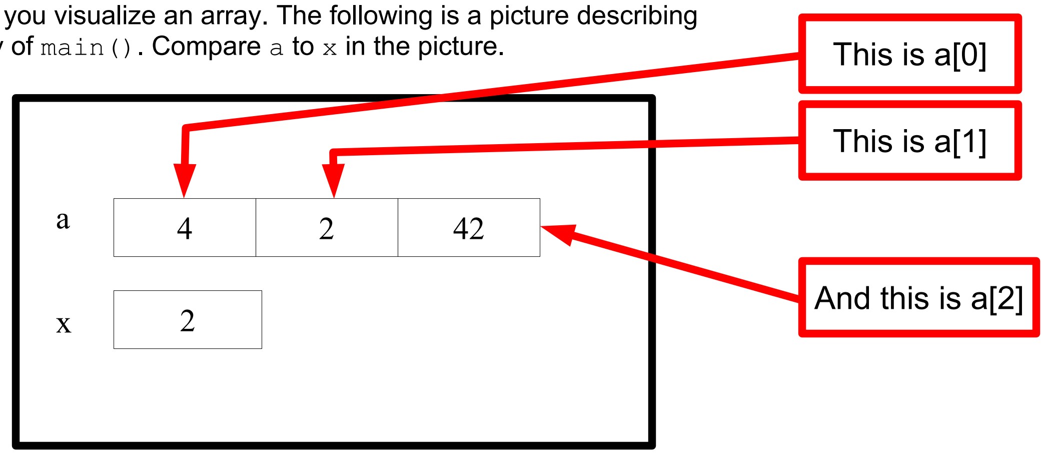
```
int a[3];
a[0] = 4;
a[1] = 2;
a[2] = 42;

std::cout << a[0] << std::endl;
int x = a[1];
std::cout << x << std::endl;
```

`a` is an **array** of 3 **integers**

The **size** of `a` is 3

The **index** of this value in array a is 0. If the size of a is 3, the possible index values are 0, 1, and 2.

This is how you visualize an array. The following is a picture describing the memory of `main()`. Compare `a` to `x` in the picture.



So this is how you declare an array:

```
[type] [array name][ [size] ];
```

As a programmer, you can think of `a[0]`, `a[1]`, `a[2]` as integer variables inside the array variable `a`. But really `a` itself is the variable.

WATCHOUT! There are **two uses of `[]`**. During the declaration of the array it is used to specify the **size** of the array. When accessing a value of the array, `[]` is used to specify the **index** of the value. Make sure you see the difference!

Although I'm calling `a[0]`, `a[1]`, and `a[2]` the values into array `a`, you can **think of them as integer variables**. Therefore you ***do*** know how to use them.

You might say: “Well what's the point? Can't we just use three variables such as `x`, `y`, and `z` instead of `a[0]`, `a[1]`, and `a[2]`???” Don't worry we'll come to that very soon.

Exercise. Add two lines of code to the above code segment. First assign the sum of `a[0]` and `a[1]` to `a[2]`. Second print `a[2]`.

Exercise. Declare an array `b` of 4 **doubles**. Set `b[0]` to `3.1`, `b[1]` to `3.14`, `b[2]` to `3.141`, and `b[3]` to `3.1415`. Compute and display the average of all the values in `b`.

Exercise. Declare an array `prime` of 5 booleans. Set `prime[0]` to `true` if 0 is a prime, otherwise set `prime[0]` to `false`. Set `prime[1]` to `true` if 1 is a prime, otherwise set `prime[1]` to `false`. Etc. Print all the values in `prime` starting with the value at index 0 with consecutive values separated by a space. (You should see two 1's.)

Declaring an array with initialization

Try this:

`{4, 2, 42}` is called an initializer

```
int a[3] = {4, 2, 42};

std::cout << a[0] << std::endl;
int x = a[1];
std::cout << x << std::endl;
```

This declares `a` as an array of 3 integers and initialize `a[0]` to 4, `a[1]` to 2, and `a[2]` to 42.

Exercise. What if you declare an integer array without initialization and immediately print all the values in the array. What do you think you will get? Test it out with your compiler.

You can partially initialize an integer array. Uninitialized values will be automatically initialized to zero.

```
int a[3] = {4};

std::cout << a[1] << std::endl;
```

If on the other hand `a` is an array of `doubles`, then the uninitialized values are set to 0.0.

You can also declare an array **without specifying the size** if you have an initializer. In that case the size will be the number of values in the initializer:

```
double a[] = {1.1, 3.3, 5.5, 7.7};
```

In the above example a is an array of four doubles. In other words the above statement is the same as this:

```
double a[4] = {1.1, 3.3, 5.5, 7.7};
```

Gotchas

Here are some gotchas.

Exercise. Correct this statement which declares an array `x` of four doubles.

```
double[4] x;
```

Exercise. Does this work? Correct it!

```
double a[4];
a = {1.1, 3.3, 5.5, 7.7};
```

Exercise. Does this work?

```
int size;
std::cout << "how many integers in the array? ";
std::cin >> size;

int a[size] = {0};
```

POINT: ____________________

Exercise. (Assignment operator) Does this work?

```
bool a[4] = {true, true, false};
bool b[4];
b = a; // trying to copy all the values in a into b
```

Correct it.

Exercise. (Output of an array) Does this work?

```
int a[4] = {0, 1, 2, 42};
std::cout << a << std::endl; // trying to print all
                             // the values in a
```

The [ ] operator

Here's an example from the previous section.

```
int a[3] = {4, 2, 42};

std::cout << a[0] << std::endl;
int x = a[1];
std::cout << x << std::endl;
```

Actually you can specify an index value using any **integer expression**. Try this:

```
int a[3] = {4, 2, 42};

int i = 0
std::cout << a[i] << std::endl;

int x = a[i + 1];
std::cout << x << std::endl;
```

In fact this is the reason why we have arrays ... Try this:

```
int a[3] = {4, 2, 42};

for (int i = 0; i < 3; ++i)
{
    std::cout << a[i] << std::endl;
}
```

The following code declares an array and populates it with integers entered by the user:

```
int y[5] = {0};

for (int i = 0; i < 5; ++i)
{
    std::cin >> y[i];
}
```

Compare this to this code:

```
int y0 = 0, y1 = 0, y2 = 0, y3 = 0, y4 = 0;

std::cin >> y0 >> y1 >> y2 >> y3 >> y4;
```

They more or less achieve the same thing, right? Now suppose instead of 5 variables, there are 1000 variables. The first version becomes:

```
int y[1000] = {0};

for (int i = 0; i < 1000; ++i)
```

```
{
    std::cin >> y[i];
}
```

Now let's see how much time you'd need to modify the `std::cin` statement if you have not heard of arrays:

```
int y0 = 0, y1 = 0, y2 = 0, y3 = 0, y4 = 0, ...;

std::cin >> y0 >> y1 >> y2 >> y3 >> y4 >> y5 >> ...;
```

YIKES!!!

Get the point of arrays now?

In other words previously our for-loop allows us to iteratively execute a block of statements. With the array, the for-loop allows us to execute a block of statements on one variable from a huge list of variables one at a time (if you think of an element of an array as a variable). Previously our for-loop executes a bunch of statements on a single variable (or maybe two or three). But here we apply the same computation across a bunch of variables. POWERFUL IDEA!

In the next few sections, I'll show you how to apply this principle in different scenarios.

One final thing before I show you different uses of the array. It's always a good idea not to hardcode constants, remember? The size of an array is not an exception. Therefore

```
int y[1000] = {0};

for (int i = 0; i < 1000; ++i)
{
    std::cin >> y[i];
}
```

should be

```
const int SIZE = 1000;
int y[SIZE] = {0};

for (int i = 0; i < SIZE; ++i)
{
    std::cin >> y[i];
}
```

For our example in this set of notes I will not use constants for array sizes since the code is very short. But when the program is longer (such as your assignments) you should use constants.

There's another way to avoid hardcoding the size of an array. The function `sizeof()` will tell you the mount of memory used by the computer to store the value of a variable. Try this:

```
int x = 0;
double y = 0.0;
char z = 'a';
int a[5] = {0};

std::cout << sizeof(x) << '\n'
          << sizeof(y) << '\n'
          << sizeof(z) << '\n'
          << sizeof(a) << '\n';

std::cout << sizeof(int) << '\n'
          << sizeof(double) << '\n'
          << sizeof(char) << '\n';
```

So here's how you can determine the number of elements in an array. Try this:

```
int a[5] = {0};
double b[10] = {0};
char c[100];

std::cout << sizeof(a) / sizeof(int) << '\n';
std::cout << sizeof(b) / sizeof(double) << '\n';
std::cout << sizeof(c) / sizeof(char) << '\n';
```

Get it?

One final thing. You should NEVER use an index value outside the bound of your array. In other words if the size of your array a is 5, you MUST not access `a[5]` or `a[1000]`; you should only access `a[0]`, `a[1]`, `a[2]`, `a[3]`, and `a[4]`.

As mentioned earlier in the section on gotchas, you cannot assign arrays:

```
int a[3] = {1, 2, 3};
int b[3];
b = a;            // BAD BAD BAD!!!
```

This is what you have to do:

```
int a[3] = {1, 2, 3};
int b[3];
for (int i = 0; i < 3; ++i)
{
    b[i] = a[i];
}
```

Exercise. Declare an array of 10 integers; call this array `dice`. Using a for-loop, put random integers 1 to 6 into the array. Using another for-loop, print all the values in the array.

Max and min of an array of values

Exercise. The following program computes the maximum value in variables a, b, c, d, e:

```
double a = 5.4;
double b = 1.2;
double c = 5.9;
double d = 3.14;
double e = 2.718;

double max = a;
if (max < b) max = b;
if (max < c) max = c;
if (max < d) max = d;
if (max < e) max = e;
```

We can use arrays instead:

```
double a[] = {5.4, 1.2, 5.9, 3.14, 2.718};

double max = a[0];
for (int i = 1; i < 5; ++i)
{
    if (max < a[i]) max = a[i];
    std::cout << max << std::endl;
}
```

If you don't see it, just unroll the for-loop.

Neat isn't it?

Of course you can also compute the minimum:

```
double a[] = {5.4, 1.2, 5.9, 3.14, 2.718};

double min = a[0];
for (int i = 1; i < 5; ++i)
{
    if (min > a[i]) min = a[i];
    std::cout << min << std::endl;
}
```

Sum of an array of values

You can easily compute the sum of the values in an array:

```
double a[] = {5.4, 1.2, 5.9, 3.14, 2.718};

double sum = 0.0;
for (int i = 0; i < 5; ++i)
{
    sum += a[i];
    std::cout << sum << std::endl;
}
```

Exercise. Modify the above to so that instead of computing the sum of the values in the array a, your program computes the product of the values in the array.

Exercise. Declare an array of 1,000,000 random doubles with values between 0.0 and 1.0 using the `rand()` function. If the `rand()` is a good random number generator, you would expect the average of the 1,000,000 to be close to 0.5. Compute the average of the values in the array and print it.

Exercise. Declare an array of 1,000,000 random doubles with values between 0.0 and 1.0 using the `rand()` function. Compute the standard deviation of the numbers. (The formula for the standard deviation is in an earlier assignment.)

Linear search for a value in an array

Suppose now we want to look for a value in an array and set a boolean variable to true exactly where the value is found. This is how you do it:

```
char x[] = {'h', 'e', 'l', 'l', 'o'};
char target =  ' ';
std::cin >> target;

bool found = false;
for (int i = 0; i < 5; ++i)
{
    if (x[i] == target)
    {
        found = true;
        break;
    }
}

std::cout << found << std::endl;
```

Exercise. Rewrite the above so that it does not have the break statement.

You might be interested in the index of the array when the value first occurred. (And if it cannot be found, the index value is -1.) This is an example code:

```
char x[] = {'h', 'e', 'l', 'l', 'o'};
char target =  ' ';
std::cin >> target;

int index = -1;
for (int i = 0; i < 5; ++i)
{
    if (x[i] == target)
    {
        index = i;
        break;
    }
}

std::cout << index << std::endl;
```

Exercise. The above code scans the array from the smallest index value to the largest. Rewrite it so the scanning starts from the largest index value.

Exercise. Now write code to count the number of occurrences of a value in an array.

Note that if a value is not in an array, in the worse case scenario, you have to scan the whole array to realize that the value is not present. On the average you need to scan n/2 elements where n is the size of the array.

Random access

It's important to understand that there's no reason that you must scan the array from the first index to the last or from the last to the first.

You can and should think of the array as a collection of values and you have the freedom of accessing any value in the array.

This leads us to a useful idea. Recall some of the ASCII art problems. One of the difficulties is that print follows a strict direction. On the output window, your print cursor must move top-to-bottom, left-to-right.

Here's an ASCII problem. Prompt the user for two integers for variables i and j (from 0 to 9). The program prints two X's at column i and j. For instance if the user enters 0 and 3, the program prints

```
X  X
```

And if the user enters 2 7 the program prints

```
  X    X
```

Now your program's logic should be like this:

```
print i spaces
print 'X'
print j – i – 1 spaces
print 'X'
```

Correct? For instance when the user enters 2 and 7, we have i = 2 and j = 7. So the above pseudocode will actually execute

```
print 2 spaces
print 'X'
print 4 spaces
print 'X'
```

which does give this

```
  X    X
```

But WAIT!!! What if the user enters 7 and then 2? Then the first two lines of your pseudocode executes:

```
print 7 spaces
print 'X'
...
```

But now you need to go backwards ... and you can't! The print cursor can only go forward! So you really need to sort i and j. The correct algorithm is

```
sort i, j in ascending order
print i spaces
print 'X'
print j – i – 1 spaces
print 'X'
```

But there's another way to solve this problem. You can create an array of characters, put the 'X' in the array of characters and then print the array. Why would you do that? Because you have complete control over where you want to put characters into the array!

```
declare array a of characters
set a[i] to 'X'
set a[j] to 'X'
print all the values in a
```

In C++ you have

```
std::cin >> char a[10];
for (int i = 0; i < 10; i++)
{
    a[i] = ' ';
}

int i = 0, j = 0;
std::cin >> i >> j;
a[i] = 'X';
a[j] = 'X';

for (int i = 0; i < 10; i++)
{
    std::cout << a[i];
}
```

The point here is to illustrate the fact that an array gives you random access to its values. At this point in the code:

```
a[i] = 'X';
a[j] = 'X';
```

we don't really care if i is greater than j.

Exercise. Do the same problem but allow the user to input four integers. (Try to do it without arrays!)

Why arrays?

The above explains only the syntax and how to work with arrays. It also explains that if you need to work with a lot of variables, performing the same operation on these variables (such as printing their values), then you might want to use an array.

But … probably the most important "view" of an array is that you should view it as a **storage container** of values. Why is that important?

You see containers of values everywhere. For instance when you buy something online (say at amazon.com), amazon has to retrieve information on the product. The product information is in a container – the products catalog database. At this point we only have a container (or array) of integer values or double values or boolean values or characters. Much later you will learn about containers of not just single values but aggregate values such as a container of customer data made up of firstname, lastname, email address, cellphone number, etc.

An array allows us to store computation so that a program can save time when the program runs and avoid recomputation all the time.

Read the above several times!!!

So how does an array "store computation" and "avoid recomputation"?

Remember that we already know how to check if a number is prime: A positive integer n greater than 1 is a prime if we test n % d for d = 2, 3, 4, …, n – 1. If n % d is 0, then d is a divisor and therefore n is not a prime. If n % d is not zero for all d = 2, 3, 4, …, n – 1, then n must be a prime.

The next thing I told you is that you can improve on the speed of the above algorithm by testing only d = 2, 3, 4, … up to including the square root of n. For the case of n = 101, you only need to test d = 2, 3, 4, …, 10 instead of d = 2, 3, 4, …, 100. That's a saving of 90% … !!!

But … there's *yet* another way to speed up the program: you only need to test n % d for d running through *primes* up to the square root of n. Therefore for n = 101, you only need to test n % d for d = 2, 3, 5, 7. So from d = 2, 3, 4, …, 10 you go down to d = 2, 3, 5, 7, from 9 values of d to 4. That's about 50% … !

How would an array help?

One way is to compute an array of 1000 primes before you run your program. Let's say the array is called:

```
    int prime[1000];
```

After that for every n you want to test for primality, you simply test n %

prime[i] for i = 0, 1, 2, … and stop when prime[i] if greater than the square root of n. The only caveat is that this array is finite. So if you need to check for n % d when d is larger than the 1000th prime then you need to perform some tests without using the `prime` array. For instance, since the 1000th prime is 7919, and the square root of 1357931 is 1165.30, to check if 1357931 is a prime we test

```
1357931 % 2
1357931 % 3
1357931 % 5
…
1357931 % 1637
```

(the prime after 1637 is 1657 which is greater than 1165.30) and of course the primes 2, 3, 5, …, 1637 are taken from our `prime` array. By the way 1637 is the 259th prime. Therefore there are 259 checks:

```
1357931 % 2
1357931 % 3
1357931 % 5
…
1357931 % 1637
```

Using our crude method of testing

```
1357931 % 2
1357931 % 3
1357931 % 4
1357931 % 5
...
1357931 % 1357930
```

required 1357929 checks. If we test up to the square root of 1357929:

```
1357931 % 2
1357931 % 3
1357931 % 4
1357931 % 5
...
1357931 % 1165
```

we have to perform 1164 checks. In summary:

- Test 1357931 % d for d = 2, 3, 4, …, 1357930: 1357929 tests
- Test 1357931 % d for d = 2, 3, 4, …, 1165: 1164 tests
- Test 1357931 % d for d = 2, 3, 5, …, 11637: 259 tests

See the difference?

The square of 7919 is 62710561. Therefore with an array of 1000 primes, we can check if n is a prime for n up to 62710561. If n is greater than 62710561, then we need a larger `prime` array.

This is also the reason why Google works so fast. When you search for web pages containing “dog”, Google would send you the relevant URLs in a split second because Google continually scans for billions of web pages, precomputes, and stores keywords of the web pages before you even ask for “dog” related web pages. The only difference is that Google stores the results in databases and not arrays.

Bubblesort

Recall that we have the follow earlier. Suppose you have four integer variables a, b, c, d. You want to sort the values in a, b, c, d into ascending order and put them into a, b, c, d. For instance if we initially have this:

```
a = 5    b = 3    c = 6    d = 0
```

we want to end up with this:

```
a = 0    b = 3    c = 5    d = 6
```

Recall that I gave you the **bubblesort algorithm** for doing that:

```
Pass 1:
swap the values in a, b so that a <= b
swap the values in b, c so that b <= c
swap the values in c, d so that c <= d
// at this point the largest value of a, b, c, d must be in d

Pass 2:
swap the values in a, b so that a <= b
swap the values in b, c so that b <= c
// at this point the largest value of a, b, c must be in c

Pass 3:
swap the values in a, b so that a <= b
// at this point the largest value of a, b must be in b
```

[Don't recall any of the above? Quickly review your previous notes on bubblesort!!! Now!!!]

Of course you know what's going to happen next.

We're going to sort values in an array!!! (cheers ... clapping ... and wild whistling).

Let's do this slowly. Using the above pseudocode but with a, b, c, d replaced by a[0], a[1], a[2], a[3], we have the following:

```
Pass 1:
swap the values in a[0], a[1] so that a[0] <= a[1]
swap the values in a[1], a[2] so that a[1] <= a[2]
swap the values in a[2], a[3] so that a[2] <= a[3]
// at this point the largest value of a[0], ..., a[3] must be in a[3]

Pass 2:
swap the values in a[0], a[1] so that a[0] <= a[1]
swap the values in a[1], a[2] so that a[1] <= a[2]
// at this point the largest value of a[0], ..., a[2] must be in a[2]
```

```
Pass 3:
swap the values in a[0], a[1] so that a[0] <= a[1]
// at this point the largest value of a[0], a[1] must be in a[1]
```

We can rewrite Pass 1 as a for-loop:

```
Pass 1:
for i = 0, 1, 2:
    swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

See it? Pass 2 looks like this:

```
Pass 2:
for i = 0, 1:
    swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

Hmmm ... Pass 2 looks like Pass 1 ... You can combine them like this:

```
Pass 1 and 2:
for j = 2, 1:
    for i = 0, ..., j:
        swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

See that? What about Pass 3? Pass 3 looks like this (I'm writing this as a for-loop just to see the pattern):

```
Pass 3:
for i = 0:
    swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

AHA! This can also be combined with Pass 1 and Pass 2:

```
Pass 1 and 2 and 3:
for j = 2, 1, 0:
    for i = 0, ..., j:
        swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

Note that this is for an array of size 4. What about an array of size 5? Do you see that the first value of j in the above (i.e. 2) comes from 4 – 2. In general if you have an array with size SIZE, the bubblesort to sort in ascending order is this:

```
for j = SIZE – 2, ..., 1, 0:
    for i = 0, ..., j:
        swap the values in a[i], a[i+1] so that a[i] <= a[i+1]
```

Now we translate this to C++:

```
for (int j = SIZE - 2; j >= 0; --j)
{
    for (int i = 0; i <= j; ++i)
    {
        if (a[i] > a[i+1])
        {
            int t = a[i];
            a[i] = a[i+1];
            a[i+1] = t;
        }
    }
}
```

Make sure you understand and memorize this!!!

Exercise. Using the above code, create an array of 100 random integers from 0 to 1000. Sort the array in ascending order and print all the integers. Check visually that the output does give you integers in ascending order.

Exercise. Redo the previous exercise, sorting the array in ***descending*** order.

Exercise. Note that the bubblesort algorithm (ascending or descending version) or course works for an array of doubles as well. Create an array of 100 random doubles from 0.0 to 1.0, sort it in ascending order and print the values of the array so that you can visually check that the values in the array is in ascending order.

Binary search

One reason for sorting an array is to improve search. Once an array is sorted in ascending order we can perform a **binary search** on the array. Let's do a simple example before we talk about code.

Suppose you have the following list of numbers:

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15

and you want to search for the number 10; 10 is our target. You can start from the leftmost and scan to the right until the number is found. Here's another way of doing it. Put your left finger at index 0 and your right finger at index 7:

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left							right

Look at the middle index between left and right. This is

(left + right) / 2 = (0 + 7) / 2 = 7 / 2 = 3

(don't forget it's integer division!). So put your nose at index 3 (or one of your toes): nose = 3

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left			nose				right

Now ask yourself this question:

a[nose] = a[3] = 8 which is less than the target (i.e. 10)

where should you look? a[left], ..., a[nose-1] or a[nose+1],...,a[right]?

Is it obvious (because the array is ascending) that the target (if present in the array) must be in a[nose+1], ..., a[right]. Right?

So let's put our left finger at nose + 1, i.e. left = nose + 1. We now have

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
					left			right

Let's repeat the process:

nose = (left + right) / 2 = (4 + 7) / 2 = 11 / 2 = 5

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
					left	nose		right

And

a[nose] is target

TARGET IS FOUND!

What about a scenario where the target is ***not*** found? Then at some point left and right will cross over each other. Let me explain with an example.

Say we look for target = 7. Again we start with this:

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left							right

And

nose = (left + right) / 2 = 3

and the picture is:

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left			nose				right

Since

a[nose] > target

we look search for target in a[left], ..., a[nose-1], i.e. we set

right = nose – 1 = 3 – 1 = 2

and the picture is now

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left		right					

Again we compute the midpoint between left and right:

nose = (left + right) / 2 = (0 + 2) / 2 = 1

i.e.,

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
	left	nose	right					

We have

a[nose] = a[1] = 5 is less than target

So we look in a[nose+1], ..., a[right], i.e.

left = nose + 1 = 1 + 1 = 2

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
			left					
			right					

We look for the mid point again:

nose = (left + right) / 2 = (2 + 2) / 2 = 2

and the picture becomes:

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
			left					
			nose					
			right					

Since

a[nose] = 6 is less than the target

We set

left = nose + 1 = 3

Now note that left is actually greater than right!!!

i	0	1	2	3	4	5	6	7
a[i]	1	5	6	8	9	10	11	15
			right	left				

This means that the target cannot be found.

Now I'm ready to give you the code. This will set `index` to the index where the target is found. If it is not found `index` is set to -1. The name of the array is `x` and the size of the array is `SIZE`.

```
int lower = 0;
int upper = SIZE - 1;
int mid = 0;
int index = -1;

while (lower <= upper)
{
    mid = (lower + upper) / 2;

    if (x[mid] < target)
    {
        lower = mid + 1;
    }
    else if (x[mid] > target)
    {
        upper = mid - 1;
    }
    else
    {
        index = mid;
        break;
    }
}
```

Now if you have an array of length n that is not sorted, linear searching will require on the average n/2 searches. The worse case requires n searches. What about binary search? In the worse case scenario, it takes $\log_2 n$. Here's a table of comparison:

n	linear search (n)	binary search ($\log_2 n$)
10	10	3.3
100	100	6.6
1000	1000	10.0
1000000	1000000	19.9
1000000000	1000000000	29.9

See the difference? Better organization and a good search algorithm implies less time in data retrieval.

[For math geeks, if you have taken Calculus you know that the gap between n and $\log_2 n$ widens. Specifically, by l'Hôpital's rule, the limit of $n/(\log_2 n)$ as n approaches infinity is in fact infinity.]

Why is the worse case $\log_2 n$? Well look at the algorithm. In each iteration, you cut down the search space by ½ right? Supposing the worse case, your search space reaches one value. So the sizes of the search spaces looks like this:

$$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow ... \rightarrow 1$$

and you go through say k iterations. In other words you have

$$n/2^k = 1$$

i.e.

$$2^k = n$$

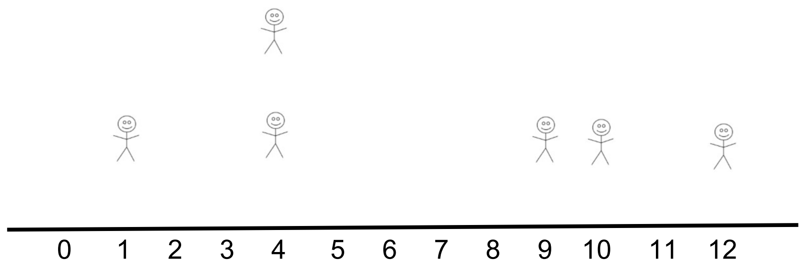
If you take log-to-base-2 you get

$\log_2 2^k = \log_2 n$, i.e., the number of iteration $k = \log_2 n$

Details on the study of algorithms and measure of what we call “time complexity” is found in CISS350 (Data Structure and Advanced algorithms) and CISS358 (Algorithmic Analysis). Deep studies in the extent of computability is found in CISS362 (Automata theory) where one is even interested in questions like “What are computers? Are there computational problems that cannot be solved by an algorithm or by a computer?”

Example: Party Planning Problem

You are planning a party for a group of friends who live on the same street:



Here (including yourself), you have people living at 1, 4, 4, 9, 10, 12. (Note: there are two people living at 4.) Suppose you host the party at 9. Then the total distance traveled by everyone to the house at 9 is

$$|1 - 9| + |4 - 9| + |4 - 9| + |10 - 9| + |12 - 9| = 8 + 5 + 5 + 1 + 3 = 22$$

You want to be a good host. So your goal is to find an address (i.e., an integer) so the that total distance traveled by everyone to the address is the smallest possible. If there are two address that gives the same minimal total distance, you pick the smaller address.

You may assume there are at most 100000 addresses.

Your program first accepts an integer n for the total number of addresses. It then accepts the n addresses (i.e., integers) and then prints the best address to hold the party.

For instance for the above test case, your console would look like this:

```
6
1 4 4 9 10 12
9
```

if your program thinks address 9 is the best.

WARNING: Incoming hint on next page.

Hint

There are 2 ways to solve this problem. Based on what you know, the following is the slower algorithm …

Suppose the number of addresses is stored in **`n`** and the addresses are stored in array **`x`**.

You test every house **`x[i]`**. In other words for each **`x[i]`**, compute the total distance traveled by everyone going to address **`x[i]`**. You compute a running minimum of these distances.

Of course there's a loop over all the potential party address **`x[i]`** for **`i`** running from 0 to **`n - 1`**. And for each address **`x[i]`**, you'll need to compute the total distance traveled to **`x[i]`**, i.e., you need to add up all the |**`x[j]`** – **`x[i]`**| for j running over 0,1,…,n-1. The |**`x[j]`** – **`x[i]`**| is the absolute value of **`x[j]`** – **`x[i]`**. You use the **`fabs`** function. I have already used this in the computation of the roots of a function.

Since you are performing a running minimum computation, you will need a **`min`** variable. Note that your problem does not print the **`min`** – you actually want the address **`x[i]`**. This is very similar to the algorithm for computing the maximum value of a function in the chapter on for-loops. So besides variable **`min`**, need also need a **`min_i`** where **`x[min_i]`** is the desired address.

Example: Cellular automata

A **cellular automata (CA)** is simply a grid of values where the value at a point in the grid can change its value. The way such a value v changes depends on the values near v.

CAs are studied in math, CS, physics, biology, social science, etc. You name it. They can be as practical as image processing where they are used to remove noise from images to create cleaner images. Yet they can be as abstract and complex as you like – they appear in AI, dynamical systems, and chaos theory. You can find lots of information about cellular automata on the web – go ahead and check out the CA entry at wikipedia.

Let me be more specific by looking at a simple example. Suppose we look at a 1-dimensional CA with these values:

```
0,0,1,1,0,1,1,1
```

This CA is made up of 8 cells. The values are either 0 or 1. So a CA can be as simple as an array.

Now suppose the value at a cell changes according to these rules:

- If the value is 1 and together with the values on its left and right, there are three or one 1s, then the value becomes 0. Otherwise it stays as 1. In other words, if the value is 1 and either the left or right neighbor is 1 (but not both), then 1 stays as 1. Otherwise it becomes 0. You can think of it this way:
    - Companionship: If 1 has exactly one companion, he/she/it lives on. If 1 has no companion, it dies.
    - Overcrowding: If 1 has too many companions, overcrowding kills he/she/it.
- If this value is 0, and together with the values on its left and right, there are two 1s, then the value becomes 1. Otherwise it stays as 0. You can think of it this way:
    - Reproduction: If a spot is available, then the 1 on the left and right produces a 1. A reproduction (i.e., 0 → 1) occurs only when there are two adjacent 1s next to the 0.

For instance, the overcrowding rule gives is this:

```
1,1,1
  ↓
  0
```

So applying this rule to the value at index 6 we get

```
0,0,1,1,0,1,1,1
            ↓
0,0,1,1,0,1,0,1
```

I'll write the rule as `111 → 0` instead of

```
1,1,1
  ↓
  0
```

Here are more examples:

- Using the rule `001 → 0`, we get
```
0,0,1,1,0,1,1,1
  ↓
_,0,_,_,_,_,_,_
```
- Using the rule `011 → 1`, we get
```
0,0,1,1,0,1,1,1
    ↓
_,_,1,_,_,_,_,_
```
- Using the rule `101 → 1`, we get
```
0,0,1,1,0,1,1,1
        ↓
_,_,_,_,1,_,_,_
```

In general the new i-th value depends on the current (i-1)-th, i-th, (i+1)th values:

```
_,_,_,_,?,_,_,_
      ↓ ↓ ↓
      \ ↓ /
       \↓/
        ↓
_,_,_,_,?,_,_,_
```

The value at the ends (the first and last) do not have two neighbors. So we will not change the values at the left and right end points:

```
0,0,1,1,0,1,1,1
↓ ↓
0,_,_,_,_,_,_,1
```

(There are other ways to compute the value of the cells at end points. For instance you can view the CA as being wrapped around at the end points so that the new value for the first cell depends on the values of first, second, and last cell).

So using the above rules we get the following behavior

```
0,0,1,1,0,1,1,1
0,0,1,1,1,1,0,1
```

You can think of the above as an evolving CA that changes with time. With three times we get

```
0,0,1,1,0,1,1,1 (time 0)
0,0,1,1,1,1,0,1 (time 1)
0,0,1,0,0,1,1,1 (time 2)
```

etc.

We will use CA to generate some ASCII art by printing the values of the cells in 1D CA: If the value is 0, we print a space and if the value is 1 we print X. We will let our CA run through a certain number of time steps, printing the CA for each time step. For instance the above CA goes through three time steps:

```
0,0,1,1,0,1,1,1
0,0,1,1,1,1,0,1
0,0,1,0,0,1,1,1
```

and we print

```
+--------+
|  XX XXX|
|  XXXX X|
|  X XXX |
+--------+
```

We will use a 1D CA of size 2 * n + 1 where n is an input from the user. We will run this for n time steps. We will start off with a CA with a 1 in the middle of the 1D array and 0 elsewhere. The set of rules to use is:

```
000 → 0
001 → 1
010 → 0
011 → 1
100 → 1
101 → 0
110 → 1
111 → 0
```

Here's the pseudocode:

```
declare an array ca of size 2*500 + 1.
note that the maximum size of the ca is 2*500 + 1.
declare an array t of size 2*500 + 1.

get n from user (at most 500), size of the ca is 2 * n + 1.
note that we will only use ca[0], …, ca[2*n].
set the values in ca to all 0s except for a 1 in the middle.

// time = 0
print ca (for value 1 print 'X' and for value 0 print ' '; use a loop).
for time = 1, 2, ..., n - 1:
{
        for each value in ca,
            apply above rules and fill corresponding value in t
        copy values in t back to ca (use a loop).
        print ca (for value 1 print 'X' and for value 0 print ' ').
}
```

Note that you can specify n = 500 for a maximum CA of size 2 * 500 + 1 = 1001. However, you can't see the ASCII art clearly on your console window because of wraparound. You can probably right-click and choose a smaller font size and larger window size and then you can see the ASCII art up to about n = 100. For larger sizes, you can save the output to a document (say MS Word), choose a really tiny font and a larger page size and print it out.

NOTE: Recall that a number such as 1425 is just 1*1000 + 4*100 + 2*10 + 5. You can extract the 1, 4, 2, and 5 from 1425 by using integer division / and integer mod %. This is viewing an integer "written in base 10". If I give you 1,0,1 or any sequence of zeroes and ones, then it's sometimes convenient to convert that into a single integer. For instance you can convert 1,0,1 into 5 using "binary representation" – I'll explain the conversion in a bit. But first, why would you want to do that? Well, compare this

```
int a = 1, b = 0, c = 1; // 1,0,1
if (a == 1 && b == 0 && c == 1)
{
      ...
}
```

with this:

```
int d = 5; // 1,0,1 converted to 5
if (d == 5)
{
      ...
}
```

Clearly the second code fragment is simpler because after the conversion there's only one variable and not three. Also, because you are comparing a single integer value, you can use a switch:

```
switch (d)
{
      case 5:
            ...
            break;
}
```

But what is a nice way to convert 1,0,1 to an integer? You use the same idea as base 10 representation of numbers but instead of powers of 10, you use powers of 2:

$1,0,1 \rightarrow 1*4 + 0*2 + 1*1$

Here's another example:

$10111 \rightarrow 1*16 + 0*8 + 1*4 + 1*2 + 1*1$

In the case of our CA, the three "bits" determining how to change a bit can be converted to an integer. The rules:

```
000 → 0
001 → 1
010 → 0
011 → 1
100 → 1
101 → 0
110 → 1
111 → 0
```

if you convert the 0s and 1s on the left to base 10 numbers, becomes

```
0 → 0
1 → 1
2 → 0
3 → 1
4 → 1
5 → 0
6 → 1
7 → 0
```

Exercises

Q1. Declare an array of 1000 integers. Set a[0] to 0 and a[1] to 1. For the remain values in the array, set each value to the sum of the previous two values. For instance set a[2] to the sum of a[0] and a[1]. Set a[3] to the sum of a[1] and a[2]. Etc. In general set a[i] to the sum of a[i-1] and a[i-2] for appropriate values of a[i]. In other words, the value at an index position is the sum of the values at two previous positions. Print all the values in your array. Visually verify the correctness of the values. The first values are of course

0, 1, 1, 2, 3, 5, 8, 13, 21, …

This is the famous Fibonacci sequence and appears in many areas of science including math, computer science, physics, etc.

Q2. Declare an array of 100 random doubles between 0.0 and 1.0. Now set all the elements of the array with ***<u>even</u>*** index values to 0.0. Print all the values in your array. Visually verify your work.

Q3. Declare an array of 20 characters. Initialize all the value of the array to the space character. In a while-loop continually print the characters separated by spaces, prompt the user for an index position and a character and set the value in the array to the character specified by the user. If the user enters -1 for the index, the program ends the while-loop and print the characters. Here is an execution:

```

index: 0
character: h

h
index: 1
character: e

h e
index: 2
character: l

h e l
index: 3
character: l

h e l l
index: 4
character: o

h e l l o
index: 6
character: w

h e l l o   w
index: 7
character: o

```

```
h e l l o   w o
index: 8
character: r

h e l l o   w o r
index: 9
character: l

h e l l o   w o r l
index: 10
character: d

h e l l o   w o r l d
index: 11
character: !

h e l l o   w o r l d !
index: -1

final: hello world!
```

Q4. Write a program that generates 10 random integers between 0 and 9 (inclusive), puts these values into an array, prints the integers in the array, counts the number of even integers in the array using a for-loop, and prints the number of even integers in the array.

Q5. Write a program that generates 10 random integers between 0 and 9 (inclusive), puts these values into an array, prints the integers in the array, counts the number of even integers in the array from the first index to either the last index or to the first 9 in the array, and prints the number of even integers (up to when you stop) in the array.

Q6. For a tic-tac-toe game, one could use an array of 9 characters to denote the characters on the tic-tac-toe board. In other words, if we create this variable:

```
char board[9];
```

then `board[0]` is the character at the top left corner of the tic-tac-toe board, ..., and `board[8]` is the bottom right corner of the board. Rewrite our tic-tac-toe game using this variable. In fact with arrays you can write your game for any n-by-n board:

```
const int SIZE = 5;
char board[SIZE * SIZE];
```

Here's a hint. The code for drawing the tic-tac-toe board would look something like this:

```
for (int i = 0; i < SIZE * SIZE; ++i)
{
    if (i % SIZE < SIZE - 1)
    {
        // print board[i] and '|'
    }
    else
    {
        // print board[i], newline,
        // and the row divider line
    }
}
```

Summary

An array is a variable containing values of the same type. The format of the statement to declare an array variable is as follows:

```
[type] [array name][ [size] ];
```

where *`[type]`* is the type of values you want to put into this array and *`[size]`* is the number of elements in this array. For instance to declare an array z of 10 booleans do this:

```
bool z[10];
```

You can declare with initialization. The following declares an array `y` of 5 `int`s with

```
int y[5] = {10, 11, 12, 13, 14};
```

You can omit the size:

```
int y[] = {10, 11, 12, 13, 14};
```

in that case the size is the number of initial values. You can declare and initialize partially:

```
int y[10] = {10, 11, 12, 13, 14};
```

In the case the remaining 5 are initialized to 0.

If `x` is an array of size 10, then you have access to `a[i]` for `i` = 0, 1, 2, ..., 9. You can (and should) think of `a[i]` as a variable. The value of `i` used to access an element of the array is called the **index** of that elements; the variable `i` is called an **index variable**. In general you can access an element with the integer value of an evaluated integer expression. In particular you can write loops to scan the elements of an array.

In particular you can write a for-loop to search for an element in the array, sum the values in an array, compute the maximum and minimum of an array, etc.

The bubblesort algorithm sorts an array. The algorithm “bubbles” the largest element to the largest index value of the array. You can rewrite the algorithm to bubble the smallest element to the smallest index value.

The binary search algorithm searches a sorted array more efficiently than a linear search.