20. Array Parameters

Objectives

- Write functions with array parameters
- Pass array to function
- Use `sizeof()` to compute the size of an array variable
- Modify array values via a function
- Introduction to 1D cellular automata

Array parameters

So far we talked about parameters of basic type (or none at all). C++ functions can handle arrays too.

Try this:

```
void print(int x[3])
{
    for (int i = 0; i < 3; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


int main()
{
    int a[] = {1, 2, 3};
    print(a);

    return 0;
}
```

Actually you need not specify the size of the array parameter:

```
void print(int x[])
{
    ...
}
...
```

(You'll understand why once you have studied pointers.)

If you want your function to work for an array of any size, you want to pass in the size of the array. For instance the above program can be improved as follows:

```
void print(int x[], int x_size)
{
    for (int i = 0; i < x_size; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


int main()
{
    int a[] = {1, 2, 3};
    print(a, 3);
    int b[] = {5, 6, 7, 8, 9};
    print(b, 5);
```

```
    return 0;
}
```

Frequently programmers also call such a variable a length variable. So another suitable name for the `x_size` parameter is **`x_len`**.

One **very important** thing to note is that it's OK to process **only part of the array** by passing in a smaller size:

```
...
int main()
{
    int a[] = {1, 2, 3};
    print(a, 2);
    ...

    return 0;
}
```

Even though array `a` has 3 values, the `print()` function will only print the first 2. So the 2 in `print(a, 2)` does not mean that the array `a` has 2 values. It just means that you want the `print()` function to print the first 2 values.

You can even have a function that prints values in the array starting at any index position:

```
void print(int x[], int xSize)
{
    for (int i = 0; i < xSize; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


void print_from(int x[], int start, int end)
{
    for (int i = start; i <= end; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


int main()
{
    int a[] = {5, 3, 5, 2, 42, 8, 2};
    print_from(a, 2, 6);

    return 0;
}
```

In the **`print_from()`**, you can specify the starting and ending index of the values in `x` to print. Frequently when you want a function to work on

only a part of the array (i.e., a subarray), you usually specify the starting index and the index that is **one beyond the last index** to process:

```
...

void print_from(int x[], int start, int end)
{
    for (int i = start; i < end; ++i)
    {
        std::cout << x[i] << ' ';
    }
}

...
```

In that case, the number of values processed (in this case printed) will be **`end - start`**.

Let's try another example. I want a function `sum()` that accepts an array of `double`s and a size and returns the sum of the array. The return type must of course be `double`. So the function looks like this:

```
double sum(double x[], int x_size)
{
    ...
}


int main()
{
    double a[] = {1.1, 2.2, 3.3};
    std::cout << sum(a, 1) << std::endl;
    std::cout << sum(a, 2) << std::endl;
    std::cout << sum(a, 3) << std::endl;

    return 0;
}
```

The code summing up the array x is just

```
double s = 0.0;
for (int i = 0; i < x_size; ++i)
{
    s += x[i];
}
```

So putting everything together we get the following … and of course you should test your code ...

```
double sum(double x[], int xSize)
{
    double s = 0.0;
    for (int i = 0; i < x_size; ++i)
    {
        s += x[i];
```

```
    }
    return s;
}


int main()
{
    double a[] = {1.1, 2.2, 3.3};
    std::cout << sum(a, 1) << std::endl;
    std::cout << sum(a, 2) << std::endl;
    std::cout << sum(a, 3) << std::endl;

    return 0;
}
```

Exercise. Write a function that returns the product of all doubles in the array that is passed in to the function.

```
_____________ product(double x[], int x_size)
{

}


int main()
{
    double a[] = {1, 2, 3};
    std::cout << product(a, 1) << '\n'; // 1
    std::cout << product(a, 2) << '\n'; // 2
    std::cout << product(a, 3) << '\n'; // 6

    return 0;
}
```

Exercise. Write a `min()` function that accepts an array of `double`s and the size of the array and returns the minimum value of the values in the array. Here's a skeleton:

```
_____________ min(_____________, int x_size)
{
    ...
}


int main()
{
    double a[] = {1.2, -2.5, -7.3, 0.0};
    std::cout << min(a, 1) << std::endl; // 1.2
    std::cout << min(a, 2) << std::endl; // -2.5
    std::cout << min(a, 3) << std::endl; // -7.5
    std::cout << min(a, 4) << std::endl; // -7.5

    return 0;
```

```
}
```

Exercise. Write a function `find()` that accepts an array of integers, the size of the array, a `target` integer, and returns the smallest index in the array where target occurs. If `target` is not found, `-1` is returned. Test it with this

```
int find(int x[], int x_size, int target)
{
    ...
}


int main()
{
    int a[] = {1, 3, 5, 2, 4, 6, 1, 3, 5, 2, 4, 6};
    std::cout << find(a, 2, 3) << std::endl; // 1
    std::cout << find(a, 2, 9) << std::endl; // -1
    std::cout << find(a, 10, 2) << std::endl; // 3

    return 0;
}
```

Exercise. Write `find_from()` function such that `find_from(x, target, start, end)` returns the index of the value of `target` in array `x`, scanning left-to-right from index value `start` to index value `end - 1`. If the value is not found, `-1` is returned.

Exercise. Write `reversefind()` function such that `reversefind(x, target, start, end)` will return the index of the value of `target` in array `x`, scanning right-to-left from index value `end - 1` to index value `start`. If the value is not found, `-1` is returned.

Exercise. Write a function `count()` that accepts an array of integers, the size of the array, and an integer value `target` and returns the number of times `target` occurs in the array. For instance if the array passed in is `{1, 42, 3, 42, 42, 1}` with size 5, and 42 is passed to `target`, then 3 is returned (because 42 occurs 3 times in the array.)

```
int count(int x[], int x_size, int target)
{
    ...
}


int main()
{
    ...
}
```

Exercise. Refer to your notes on the binary search algorithm. Write a function `binarySearch()` that accepts an array of integers, the size of the array, and a value for parameter `target`, and returns the index of the value of `target` in the array. If the value of `target` is not in the array -1 is returned. Test your code by passing in an array that is sorted in the ascending order, its size and a target in the array. Next test it with a target value that's not in the array.

Modification of values in an array through a function

There is something different between the way functions work with an array and a variable of basic type (`int` or `double` or `bool` or `char`). You already know that the following will not change the value of `x` in `main()`:

```
void inc(int x)
{
    ++x;
}

int main()
{
    int x = 42;
    inc(x);
    std::cout << x << std::endl; // still 42, right?

    return 0;
}
```

Try this:

```
void inc(int x[])
{
    ++x[0];
}


int main()
{
    int x[] = {42, 43, 44, 45};
    inc(x);
    std::cout << x[0] << std::endl;

    return 0;
}
```

Functions **can modify** the **values** in an **array that's passed in**. In other words, by default, **arrays are pass-by-reference**.

Remember that!!!

For the reason why arrays are so different from variables of basic type you will have to wait till we talk about pointers.

Here's another example. This function sets all the values in the array parameter to zero:

```
void zero_out(int x[], int size)
```

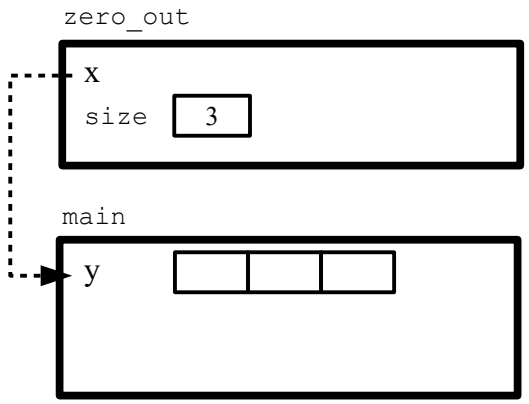
```
{
    for (int i = 0; i < size; i++)
    {
        x[i] = 0;
    }
}

int main()
{
    int y[] = {1, 2, 3};
    zero_out(y, 3);

    for (int i = 0; i < 3; i++)
    {
        std::cout << y[i] << ' ';
    }
    std::cout << '\n';

    return 0;
}
```

Here's a picture to keep in mind. Right after all parameters of the `zero_out()` function are initialized (but before the for-loop) the memory of the functions look like this:



As you can see the parameter `size` (of `zeroOut()`) has its own memory. It's just like any regular variable. The initialization of `size` involves copying the value of 3 to `size`. Recall that this type of parameter passing is called pass-by-value.

However the parameter `x` (of `zero_out()`) does **NOT** receive the values of the array `y`. The point of the picture is to show you that `x` actually **refers** directly to the memory of `y`. You can and should think of `x` as another name for `y` – an alias. Or you can think of the `x` in `zero_out()` as a parasite that lives on the memory of `y`. Therefore changing `x[0]` in `zero_out()` is the same as changing the `y[0]` in `main()`. This form of parameter-passing is called **pass-by-**

reference.

Test the function to make sure it works.

Exercise. Write a function `zero_out_odd()` that accepts an array of integers and the size of the array and then replaces all the odd values in the array by zero. Test your function.

Exercise. Write a function `rand_array()` that accepts an array of integers, the size of the array, min (an int), max (an int) and puts random integers from min, min+1, …, max – 1 into the array. For instance calling `rand_array(x, 10, 1, 7)` will randomize x[0], x[1], x[2], …, x[9] with integer values from 1 to 6. Test your function.

Exercise. Write a function `two_powers()` that accepts an array of integers and the size of the array and puts powers of 2 starting with 2^0=1 into the array. For instance if you pass array x with size 5 into the function, on return, `x` has values 1, 2, 4, 8, 16, Test your function.

Exercise. Write a function `primes()` that accepts an array of integers and the size of the array and puts primes into the array starting with 2. For instance calling `primes(x, 5)` will set `x[0]` to 2, `x[1]` to 3, `x[2]` to 5, `x[3]` to 7, `x[4]` to 11. Test your function.

Exercise. Write a function `swap()` that accepts an array of integers and two index values, and swaps the values at those index positions. For instance if x is an array initialized with {1, 2, 3, 4}, then on return from calling `swap(x, 1, 3)`, the values in `x` becomes

1, 4, 3, 2

Test your function.

Bubblesort and binary search

Exercise. Write a function `bubblesort()` that accepts an array of integers and an integer for the size of the array and performs bubblesort on the array so that the values are in ascending order. Test your code: Create an array of 10 random integers, call the `bubblesort()` function, and print the values of `x` in `main()`. Test your function.

Exercise. Write a function **`bubblesort_from()`** that accepts an array **`x`** of integers and two integers **`start`** and **`end`** and performs bubblesort on from **`x[start]`** to **`x[end - 1]`** so that the values are in ascending order. Test your code: Create an array of 10 random integers, call the **`bubblesort_from()`** function to sort the values using **`start=2`** and **`end=7`**, and print the values of **`x`** in **`main()`**. Test your function.

Exercise. Write a function `binarysearch()` that performs binary search on an array. Specifically, the function accepts an array **`x`** of integers, an integer **`x_len`** for the size of the array, and an integer **`target`**. The function returns the index in `x` where **`target`** appears. The function assume that **`x`** is sorted in ascending order. Test your function.

Exercise. Write a function `binarysearch()` that accepts an integer array **`x`**, integers **`start`** and **`end`** and integer **`target`**, and then performs binary search on an array **`x`** from index **`start`** to index **`end - 1`** searching for **`target.`** The function returns the index in `x` where **`target`** appears. The function assume that **`x`** is sorted in ascending order from index **`start`** to index **`end - 1`**. Test your function.

Computing array sizes: `sizeof()`

This is a quick review.

The amount of memory (in terms of bytes – i.e. 8 bits) used by a value, a variable, or a type can be found by calling the `sizeof()` function.

Try this:

```
std::cout << sizeof(int) << std::endl;
std::cout << sizeof(42) << std::endl;

int x;
std::cout << sizeof(x) << std::endl;

int y[5];
std::cout << sizeof(y) << std::endl;
```

As you can see, to determine the amount of memory used to hold an int value is 4 bytes. And you get this number (I.e. 4) by using any of the following:

```
sizeof(int)       sizeof(1)          sizeof(x)
```

where `x` is an `int` variable. In the above

```
sizeof(y)
```

gives you the number of bytes used for an array of 5 integers.

Here's a useful application of the `sizeof()` function. Suppose you have an array of integers and you want to compute **the size of the array**. You can do this:

```
int y[5];
std::cout << sizeof(y) / sizeof(int) << std::endl;
```

With this, the following program:

```
#include <iostream>


void print(int x[], int x_size)
{
    for (int i = 0; i < x_size; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


int main()
{
    int a[] = {1, 2, 3};
```

```
    print(a, 3);

    return 0;
}
```

can be rewritten as

```
...

int main()
{
    int a[] = {1, 2, 3};
    int a_size = sizeof(a) / sizeof(int);
    print(a, a_size);

    return 0;
}
```

Why is this is a good thing? Because now if you change your program so that it works with an array of 5 values:

```
...
int main()
{
    int a[] = {1, 2, 3, 6, 2};
    ...
}
...
```

you don't have to worry about changing

```
...
    print(a, 3);
...
```

to

```
...
    print(a, 5);
...
```

Since with

```
...
    int a_size = sizeof(a) / sizeof(int);
    print(a, a_size);
...
```

The program works for array **a** of any size.

By the way, remember that it's OK to process only part of the array:

```
#include <iostream>


void print(int x[], int x_size)
{
    for (int i = 0; i < x_size; ++i)
    {
        std::cout << x[i] << ' ';
    }
}
```

```
int main()
{
    int a[] = {1, 2, 3};
    print(a, 2);

    return 0;
}
```

Of course you already know (see earlier notes on arrays) that you should never go outside the array so something like this is BAD ... (this has nothing to do with functions of course) ...

```
...
    int a[] = {1, 2, 3};
    print(a, 5);
...
```

Make sure you try it.

Gotchas

It's very important to remember that the `sizeof()` function is correct only when the array variable is **not a function parameter** in other words, you should only call `sizeof()` on an array variable that's not a parameters.

```
#include <iostream>


void print(int x[])
{
    int x_size = sizeof(x) / sizeof(int);
    for (int i = 0; i < x_size; ++i)
    {
        std::cout << x[i] << ' ';
    }
}


int main()
{
    int a[] = {1, 2, 3};
    print(a);

    return 0;
}
```

Here's another one ...

You **cannot return an array**!!! The following is not valid C++ code:

```
int[3] whatever()
{
    int x[3] = {1, 2, 3};

    return x;
}
```

In C/C++ you can pass an array into a function, but you cannot send an array back. Remember that!!!

Arrays and operators

Here are some **very common gotchas for arrays**.
In general all the usual operators do not work the way you expect.

Suppose you want to check if two arrays have the same values. Try this:

```
int x[] = {1, 2, 3};
int y[] = {1, 2, 3};
if (x == y)
{
    std::cout << "same" << std::endl;
}
else
{
    std::cout << "different" << std::endl;
}
```

Does it work? (Duh.) You will have to wait till we talk about pointers before you know why.

What about this?

```
int x[] = {1,2,3};
int y[3];
y = x;
```

Or:

```
int x[] = {1, 2, 3};
int y[] = {1, 2, 4};
if (x == y)
    std::cout << "same" << std::endl;
else
    std::cout << "different" << std::endl;
```

So remember this: **The comparison operators == and != and the assignment operator = does not work as "expected" for arrays.**

In general you should not expect any operator to work "in the obvious way" for arrays.

Standard operations on arrays

As mentioned in the previous section, standard operations for variables of basic types (int, double, bool, char) such as assignment (i.e., =) and equality comparison (I.e., ==) does not work for arrays.

The correct thing to do is compare the values in the arrays manually. Try this:

```
int x[] = {1, 2, 3};
int y[] = {1, 2, 3};

bool same = true;
for (int i = 0; i < 3; i++)
{
    if (x[i] != y[i])
    {
        same = false;
        break;
    }
}
if (same)
{
    std::cout << "same" << std::endl;
}
else
{
    std::cout << "different" << std::endl;
}
```

So let's say you compare lots of arrays. You say to yourself, “I'm smart enough to write a function to do that!” So you rewrite the above as this:

```
bool array_isequal(int x[], int y[])
{
    for (int i = 0; i < 3; i++)
    {
        if (x[i] != y[i])
        {
            return false;
        }
    }
    return true;
}


int main()
{
    int x[] = {1, 2, 3};
    int y[] = {1, 2, 3};

    std::cout << array_isequal(x, y) << std::endl;

  return 0;
}
```

Of course we want to make sure our “array equal comparison function” works for general arrays of any sizes. Of course the arrays have difference sizes, they are different. (Right? The array `{1, 2, 3}` is different from `{1, 2, 3, [blah]}` regardless of the value of `[blah]`.)

```
bool array_isequal(int x[], int x_len,
                   int y[], int y_len)
{
    if (x_len == y_len)
    {
        for (int i = 0; i < x_len; ++i)
        {
            if (x[i] != y[i])
            {
                return false;
            }
        }
        return true;
    }
    else
    {
        return false;
    }
}


int main()
{
  int x[] = {1, 2, 3};
  int y[] = {1, 2, 3};

  std::cout << array_isequal(x, y) << std::endl;

  return 0;
}
```

But ... we're used to something like

```
if (x == y)
{
    std::cout << "same" << std::endl;
}
else
{
    std::cout << "different" << std::endl;
}
```

instead of

```
if (array_isequal(x, x_len, y, y_len))
{
    std::cout << "same" << std::endl;
}
else
{
```

```
    std::cout << "different" << std::endl;
}
```

It turns out that you can actually define your own version of the == operator! ... but you have to wait for CISS245 (Advanced Programming) to learn to write your own ==. In other words not only can you create your own functions, you can also create your own operators as well.

Exercise. Write a function `array_isnotequal()` (for "is not equal") that works in the obvious way. The body of the function should contain one line that calls `array_isequal()`. Test your function. [Hint: `array_isnotequal()` is the "opposite" of `array_isequal()`.]

Exercise. Write a function `array_assign()` (for "assignment operator") such that

```
array_assign(x, x_len, y, y_len)
```

will copy the first `y_len` values in the array `y` to the array `x and set x_len` to `y_len`. Test your function.

Exercise. Write a function `array_insert(x, size, v, i)` such that if x is an array with values {11,22,33,44}, `x_len` is `4`, calling

```
array_insert(x, x_len, 42, 1)
```

will insert `42` into the array `x` at index `1`. The end result is that `x` has the following values

{11, 42, 22, 33, 44}

and `x_len` is set to `5`.

Example: Cellular automata

In an earlier chapter, I talked about the concept of cellular automata. Now that we know functions, let's rewrite and clean up the code using functions. I'll repeat the information of cellular automata here so you don't have to look for the info from that earlier chapter.

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
```

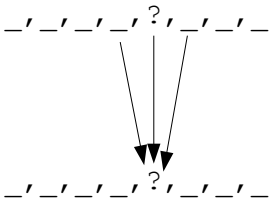
For simplicity, I might write the rule as `111 → 0` instead of

```
 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

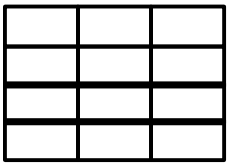
```
0,0,1,1,0,1,1,1
0,0,1,1,1,1,0,1
```

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.

(A 2D CA is similar to the 1D CA except that a value has 8 neighbors or 4 neighbors, depending on how you define neighbors. A 2D array, i.e., 2-dimensional array, is a 2-dimensional version of a 1D array. For instance here's a 4-by-3 2D array. See later notes on 2D arrays for details.



So the new value at that location depends on 9 values (itself and its 8 surrounding neighbors) or 5 values (itself and its 4 neighbors on its N,S,E,W sides). 2D ca is used in for instance in image processing.)

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 skeleton code:

```
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) where the 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).

// time > 0
for time = 1, 2, ..., n - 1:
{
   for each value in ca:
   {
      apply the above rules and fill the 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.

Try different values for input. In particular start small and then try larger and larger value and you'll see a very interesting diagram,

If the diagram does not surprised you, then you are probably wrong!!!

Here are some functions that clearly appears in the above pseudocode:

```
// initialize the ca
void init(char ca[], int size);

// print the ca
void print(char ca[], int size);

// copy values in t into ca
void copy(char ca[], char t[], int size);
```

```
// update the ca
void update(char ca[], int size);
```

Test the above functions thoroughly. Then write a function to print n steps in the evolution of a CA.

```
void print_n_generations(int n)
{
    int ca[2 * 500 + 1];
    init(ca, n);
    print(ca, n);
    for (int time = 1; time < n; ++time)
    {
        update(ca, n);
        print(ca, n);
    }
}
```

Much cleaner right?

Exercise. Keep running your ca above. When will the values of the ca start to repeat? What is the state (the values) of your ca that first gets repeated (you get a cycle)? How many time steps for the first repeat? (I hope it's obvious that the state of your ca will repeat!!!)

Exercise. Now change the initial state (i.e., the intial values) of your ca. For instance instead of a 1 in the middle, try three 1s in the middle, try 1,0,1,0,1,0,…, try all 1s, etc. For each initial state, figure out shortest time for a first repeat. Print out your results. Which initial state of your ca will give you the longest cycle? How would you do this systematically? In other words, how would you write a program to go through ALL possible initial states for your ca? [Challenging: Read the section on BINARY NUMBERS]

Exercise. Find a way to iterate through all possible sets of CA rules. For each set of rules, find the number of steps for the ca to repeat itself. [Challenging: Read the section on BINARY NUMBERS]

BINARY NUMBERS: 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;
if (d == 5)
{
    ...
}
```

Clearly the second code fragment is simpler. 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:

```
1,0,1 → 1*4 + 0*2 + 1*1
```

i.e., instead of powers of 10, you use powers of 2!!! Here's another example:

```
10111 → 1*16 + 0*8 + 1*4 + 1*2 + 1*1 = 23
```

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
```

Coding involving the bits (such as 101) instead of base 10 numbers

(such as 5) is clearly going to be more tedious.

Exercise. The above converts a bunch of bits into a regular integer. What about the opposite? How would you convert an integer into bits? Of course the conversion process are compatible: From 101 you get 5 and your process for convert an integer back to bits must convert 5 to 101. Also, the bits 10111 is converted to 23 and your process must convert 23 back to 10111.

Summary

Functions can have array parameters:

```
void f(int x[])
{
    ...
}
```

The syntax for calling a function is the usual

```
int main()
{
    int a[3];
    f(a);
}
```

When a function modifies the values of an array parameter, the value in the array argument is also modified:

```
void g(double x[])
{
    x[2] = 42;
    return;
}

int main()
{
    double d[100] = {0.0}; // all values set to 0.0
    g(d);
    std::cout << d[2] << '\n'; // d[2] is changed
    return 0;
}
```

In other words, array arguments are pass by reference.

The `sizeof()` will not work in the usual way for array parameters:

```
void g(double x[])
{
    // does not print 100
    std::cout << sizeof(d) / sizeof(double) << '\n';
    return;
}

int main()
{
    double d[100] = {0.0};

    // 100 is printed
    std::cout << sizeof(d) / sizeof(double) << '\n';
    g(d);

    return 0;
}
```

To make a function work in general for an array of any size, you can pass the intended size of the array. For instance

```
void f(int x[], int size)
{
    // process x from x[0], x[1], ..., x[size - 1]
}
```

Of course you need not start with index 0. For instance you can do this:

```
void f(int x[], int start, int end)
{
    // process x from x[start] to x[end - 1]
}
```