Functions III

const arguments

Remember that we can declare variables as const:

```
// PI is a constant that cannot be modified
const double PI = acos(-1);
// trying to change the value of PI
PI = 7; // compile-time error!
```

You can also use const on function arguments:

```
// ‘const int x’ means we promise not to modify @x
void incrementValue(const int& x) {
    // trying to change the value of @x
    ++x; // compile-time error!
}
```

Value or Reference?

If the variable is a primitive (int, double, bool, char) or an object:

- pass-by-value (the default behavior) is creating a *copy* of a variable
- pass-by-reference (using &) is like sharing the *same variable* between functions

If the variable already holds an address (an array, for example):

- pass-by-value is still passing an address, so it will behave like pass-by-reference!
- using pass-by-reference on something that is already a reference is seldom useful

Today we'll talk about using arrays as arguments

- arrays as arguments <u>always</u> behave as if passed by reference

Arrays as Arguments

Let's say I want to write a function that takes an array as input

- this function must obviously accept an array as one of its arguments (it should be const if the function will not modify it)
- it must also accept a second argument, which specifies the number of valid elements in the array (probably should be const)

Examples:

```
// sorts all @num_elements in @array in ascending order

void bubblesort(int array[], const int num_elements);


// prints each of the elements in @array on its own line

void print_array(const string array[], const int num_elements);
```

Arrays as Arguments

A closer look:

```
// sorts all @num_elements in @array in ascending order

void bubblesort(int array[], const int num_elements);
```

Notice:

- the array argument has a pair of <u>empty</u> square brackets after it.
- the second argument, `@num_elements`, is `const`. This is an encouraged best practice.
- any array passed to this function might be changed, since arrays as arguments behave as if passed by reference (this is perfect for a sorting function!)

The array in this function *cannot* be changed (notice the `const`):

```
// prints each of the elements in @array on its own line

void print_array(const string array[], const int num_elements);
```

Arrays as Arguments

The array passed to this function *can* be changed:

```
// sorts all @num_elements in @array in ascending order
void bubblesort(int array[], const int num_elements);
```

The array passed to this one *cannot* be changed (notice the const):

```
// prints each of the elements in @array on its own line
void print_array(const string array[], const int num_elements);
```

Specifying an array as const means that the function cannot modify it

- this is always a good idea when you know the array won't change (for example, if you're just printing its values)

Arrays as Arguments

Assume this function exists:

```
// prints each of the elements in @array on its own line
void print_array(const string array[], const int num_elements);
```

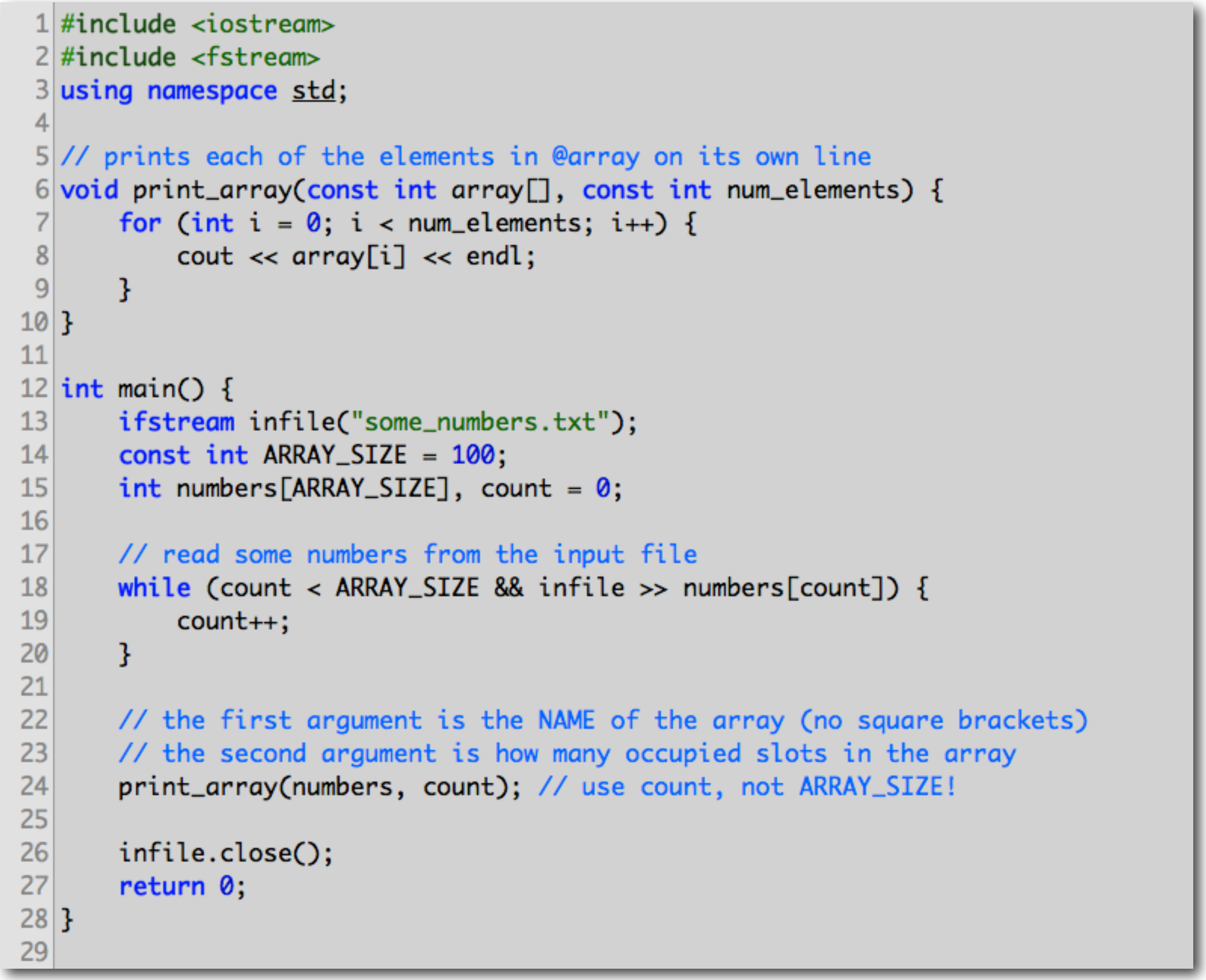
To call the function:

- provide the name of the of the array (without square brackets) as the first argument
- use a count of the number of valid elements in the array (NOT the total size of the array) as the second argument

Assume names is an array of 50 elements, of which 12 are filled…

```
print_array(names, 12); // prints the 12 names in the array
```

Arrays as Arguments



```cpp
#include <iostream>
#include <fstream>
using namespace std;

// prints each of the elements in @array on its own line
void print_array(const int array[], const int num_elements) {
    for (int i = 0; i < num_elements; i++) {
        cout << array[i] << endl;
    }
}

int main() {
    ifstream infile("some_numbers.txt");
    const int ARRAY_SIZE = 100;
    int numbers[ARRAY_SIZE], count = 0;

    // read some numbers from the input file
    while (count < ARRAY_SIZE && infile >> numbers[count]) {
        count++;
    }

    // the first argument is the NAME of the array (no square brackets)
    // the second argument is how many occupied slots in the array
    print_array(numbers, count); // use count, not ARRAY_SIZE!

    infile.close();
    return 0;
}

```

2D Arrays as Arguments

Declaring a function that takes a 2D array is slightly more involved:

```
// GLOBAL variable declaring the width of the 2nd dimension
static const int COLS = 32; // only acceptable use of global constants


// each row is assumed to have COLS columns in it, OR...
void use_2D_array(int array[][COLS], const int rows);


// each row has the specified number of columns (not necessarily COLS)
void use_2D_array(int array[][COLS], const int rows, const int cols);
```

Note the similarity to declaring 2D arrays in general:

- only the first dimension can (and should, in the case of functions) be omitted
- all other dimensions must be explicitly specified

2D Arrays as Arguments

In this function:

```
// each row is assumed to have COLS columns in it

void use_2D_array(int array[][COLS], const int rows);
```

`rows` specifies the number of valid rows in the array

- the array you pass to the function will be assumed to have `COLS` columns in it, but you have to explicitly specify the number of valid `rows`.

Example:

```
// only use the first 10 rows (each has COLS columns) of @some_array

use_2D_array(some_array, 10);
```

2D Arrays as Arguments

In this function:

```
// each row has the specified number of columns (not necessarily COLS)
void use_2D_array(int array[][COLS], const int rows, const int cols);
```

`rows` and `cols` specify the number of valid elements in the array

- if only the first 6 columns of the first 10 rows of a 20x20 array are used, specify `rows` as 10 and `cols` as 6!

Example:

```
// only use the first 6 columns of the first 10 rows of @some_array
use_2D_array(some_array, 10, 6);
```

Arrays as Arguments

Remember:

- when passing an array to a function, just use its name (no brackets)
- specify the number of valid elements when calling the function

Functions cannot return arrays as you might expect

- pass an array as a non-const argument and then directly modify it
- arrays always behave as if passed by reference

Bubble Sort

Future assignments may use this as a starting point.

You would be well-advised to complete this assignment!

Swapping Two Values

BubbleSort requires us to be able to swap two array elements…

Why does this not do what we want?

```
// swap the two values (wrong way)
array[i]   = array[i-1];  // array[i] gets overwritten
array[i-1] = array[i];    // so now both values are the same
```

You have to store one of the values into a temporary variable!

```
int temp = array[i];
// swap the two values (right way)
array[i]   = array[i-1];  // array[i] gets overwritten
array[i-1] = temp;        // but its value was saved in temp
```

Swapping Two Values

BubbleSort requires us to be able to swap two array elements…

You *could* create a function to swap to values (use pass-by-reference!)

```
// swap the values of @val1 and @val2
void swap_values(int& val1, int& val2) {
    int temp = val1;
    val1 = val2;  // val1 gets overwritten
    val2 = temp;  // but its original value was saved in temp
}
```

Then call the function to swap the array elements:

```
swap_values(array[i], array[i-1]); // swaps the two values
```