CSE 3100: Systems Programming

Lecture 6 : Memory Allocation

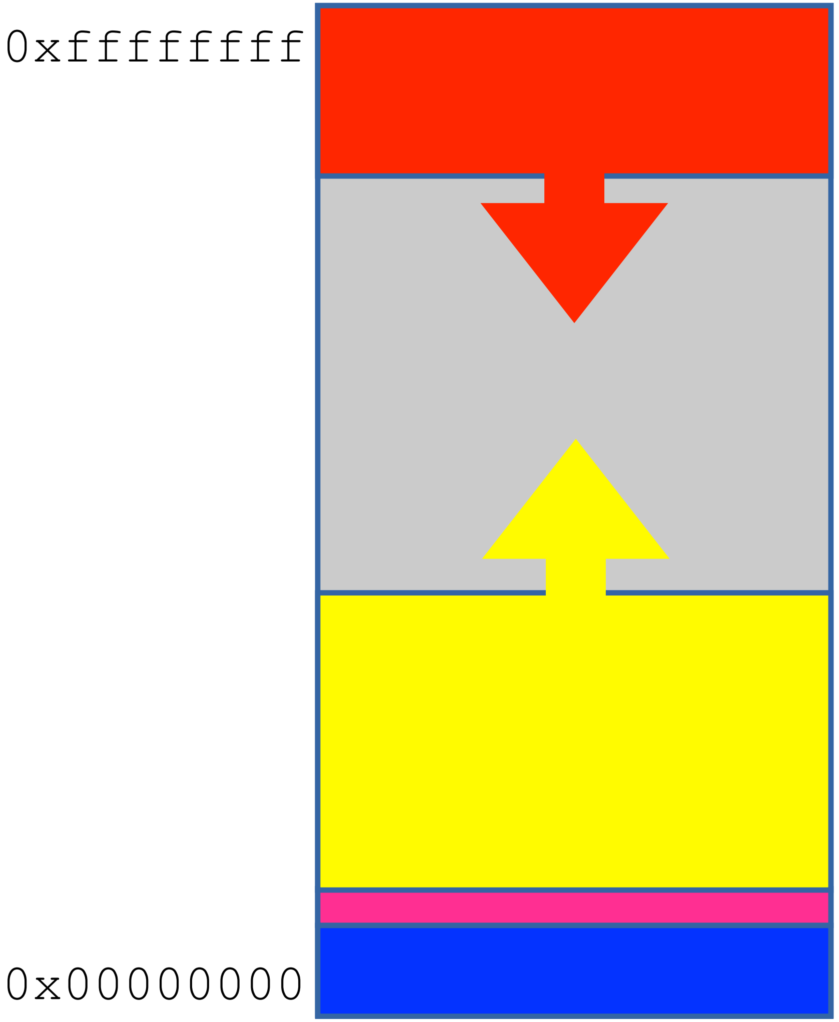
Professor Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

Recall: Memory Organization (1)

- Three pools of memory
  - Static/global
  - Stack
  - Heap
- Each pool features
  - Different lifetime
  - Different allocation/deallocation policy

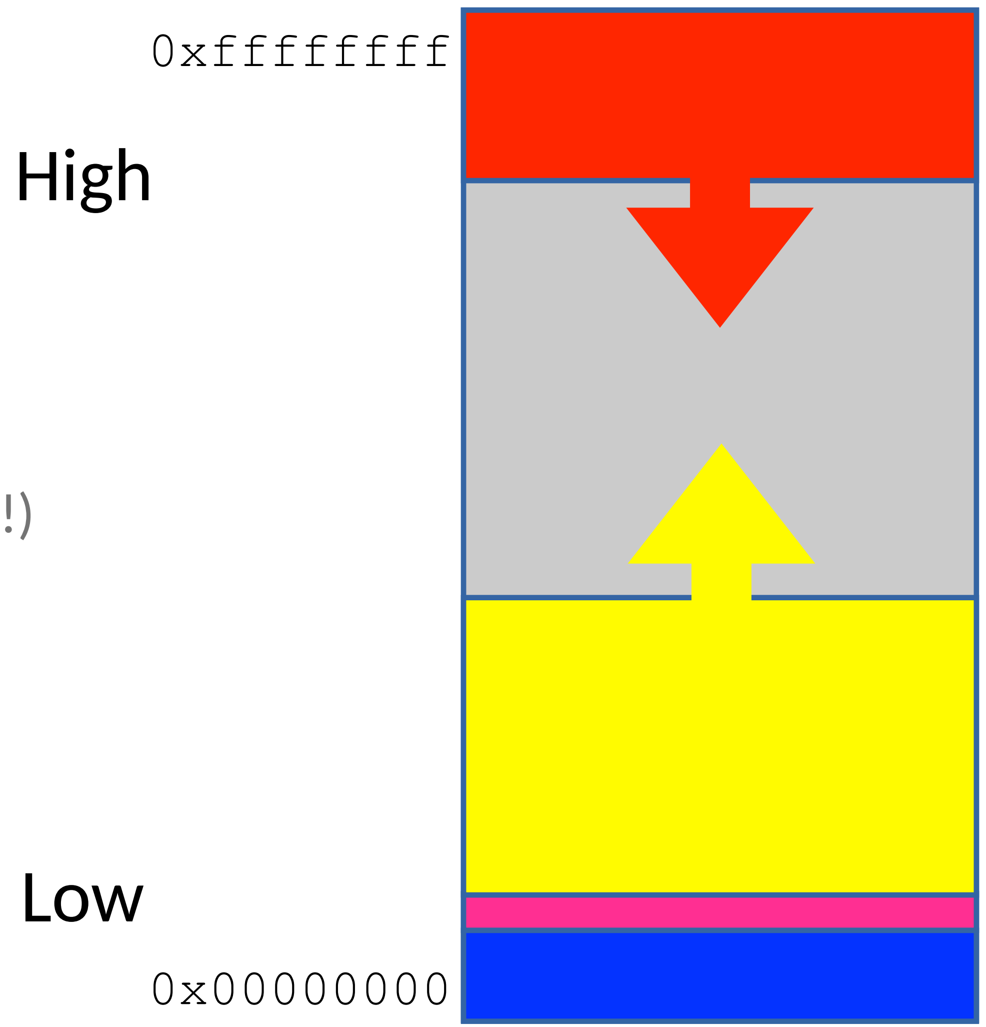


Recall: Memory Organization (2)

- Memory….
  - Every *Process* has an
    - **Address Space**
  - **Executable** code is at the bottom
  - **Statics** and globals are just above
  - **Stack** is at the top (going down!)
  - **Heap** grows from the bottom (going up!)
  - **Gray** no-man's land is up for grab

Low end may not start from 0.

High end may not be the 0xff…ff.



Static/global memory pool

- This is where
  - All constants (including string literals) are held
  - Global variables
  - All variables declared “static” are held
- Allocated when
  - The program starts
- Deallocated when
  - The program terminates
- FIXED SIZE
  - Compiler needs to know the size and make reservations

Stack

- This is where….
  - Memory comes from for local variables in functions!
- Easy to manage because it is automatic!
  - Allocated automatically when entering the function
  - De-allocated automatically when you leave the function
    - Scope is that of function
    - Should not be used after the function returns
      - For example, indirectly used via a pointer
- Default stack size using gcc is 2 MBytes.
- Need to increase stack size for large arrays and deep recursion

Heap

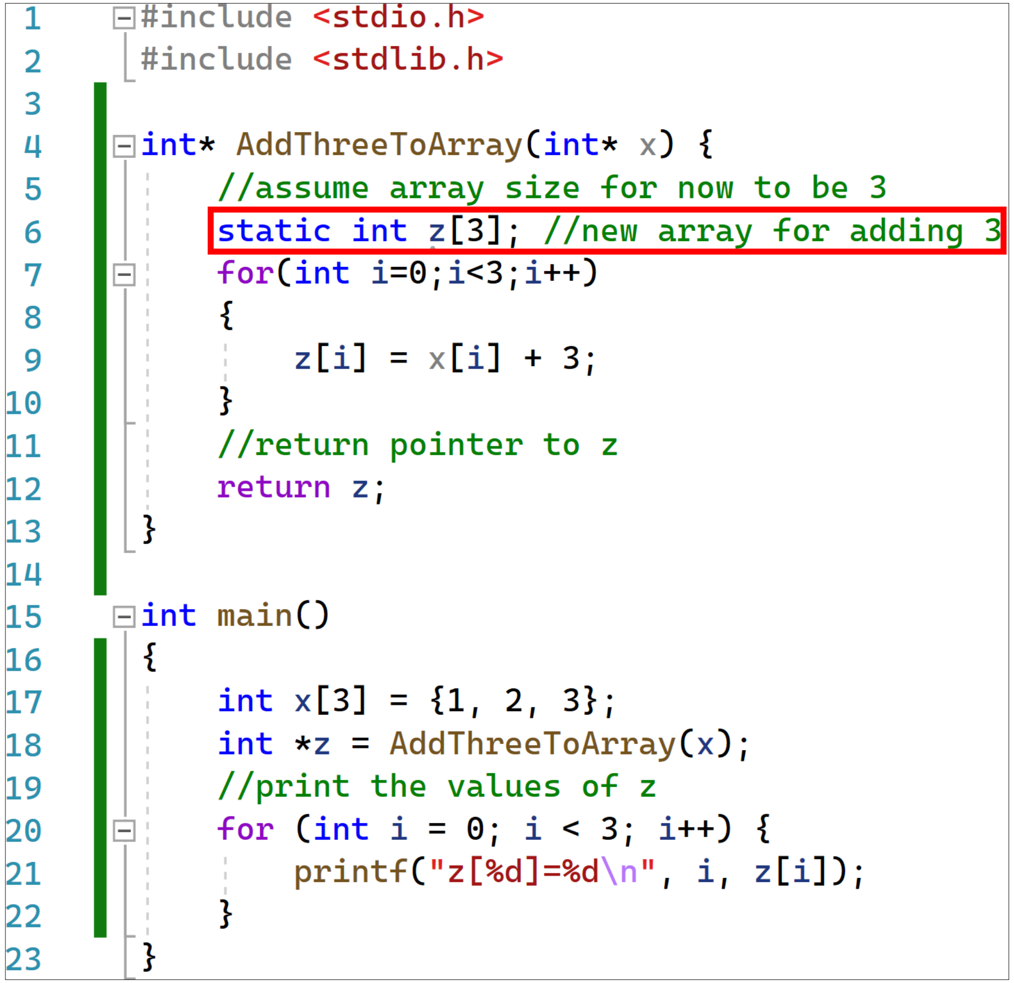
- This is where…
  - Memory comes from for manual “on-the-fly” allocations
- Who is in charge ?
  - **The programmer** for both allocation / deallocation
- Lifetime of memory blocks ?
  - As long as they are not freed!

Manipulating Memory in C

- In the previous lecture we asked a very basic question:

In C how do I get my functions to return multiple values (arrays)?

(Bad) Solution 1: Use Static Memory



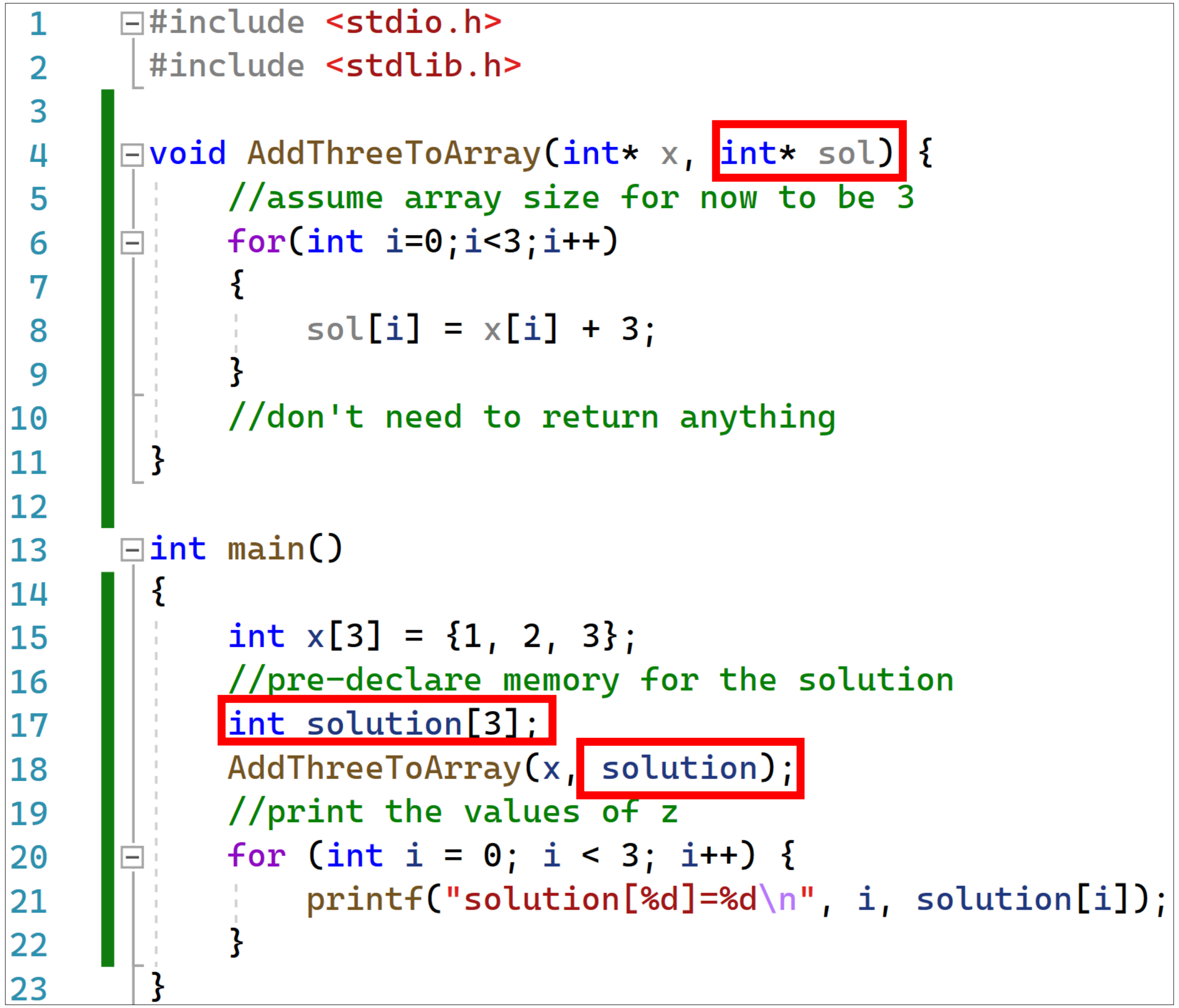
```
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x) {
    //assume array size for now to be 3
    static int z[3]; //new array for adding 3
    for(int i=0;i<3;i++)
    {
        z[i] = x[i] + 3;
    }
    //return pointer to z
    return z;
}

int main()
{
    int x[3] = {1, 2, 3};
    int *z = AddThreeToArray(x);
    //print the values of z
    for (int i = 0; i < 3; i++) {
        printf("z[%d]=%d\n", i, z[i]);
    }
}
```

- Are there any problems with this approach?

A. If we want to call the function multiple times, old solution will be over written.

B. The static memory is allocated WHEN the program starts and is fixed. What if we don’t know the size of the solution before run time?

(Bad) Solution 2: Pass the empty solution as input



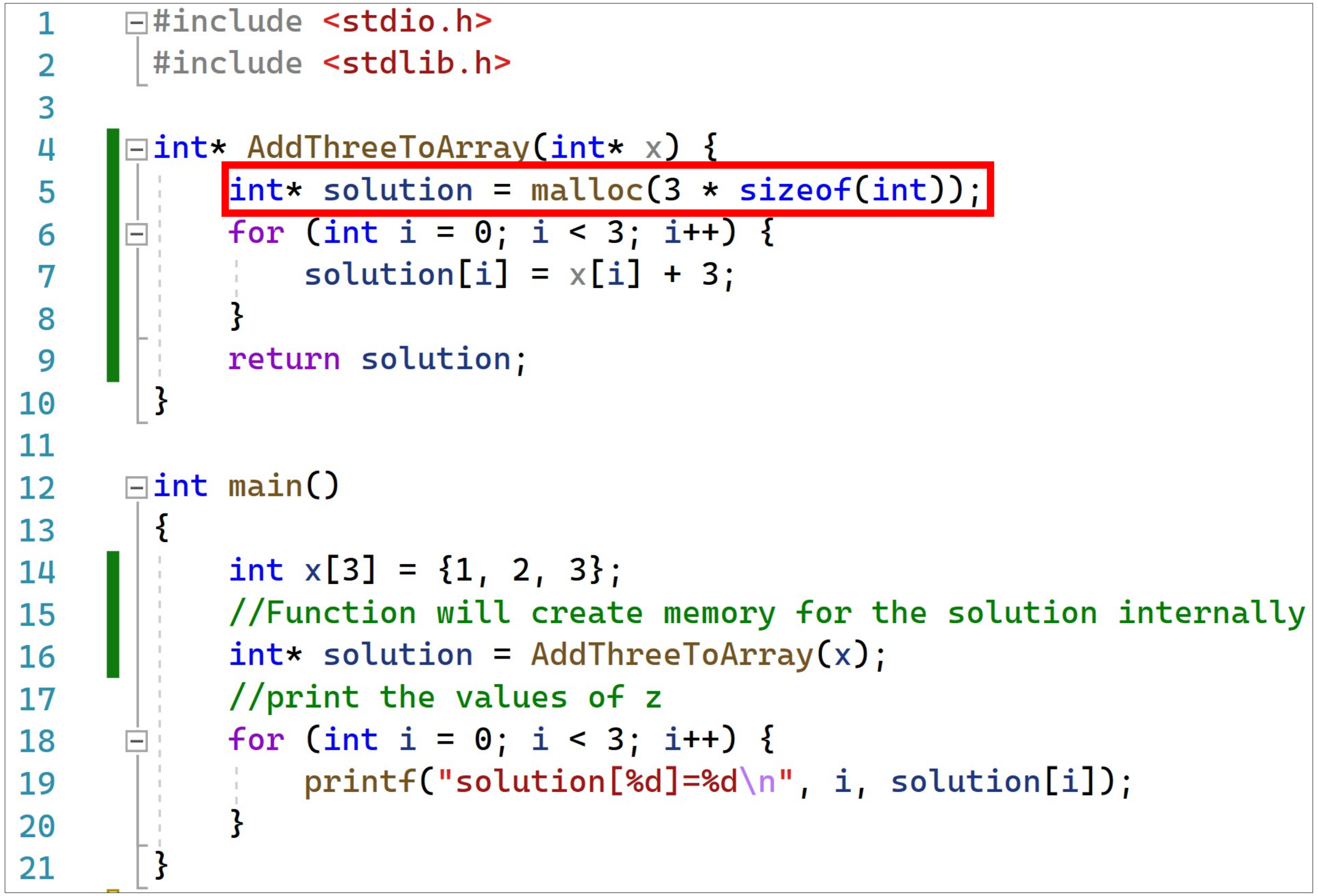
```
#include <stdio.h>
#include <stdlib.h>

void AddThreeToArray(int* x, int* sol) {
    //assume array size for now to be 3
    for(int i=0;i<3;i++)
    {
        sol[i] = x[i] + 3;
    }
    //don't need to return anything
}

int main()
{
    int x[3] = {1, 2, 3};
    //pre-declare memory for the solution
    int solution[3];
    AddThreeToArray(x, solution);
    //print the values of z
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

- Are there any problems with this approach?

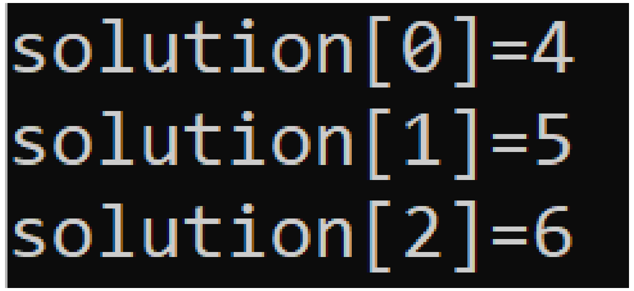
What if we don’t know the size of the solution BEFORE the method is called?

Clearly we need a way to allocate memory during run time.

Good Solution: Allocate Memory Dynamically



```c
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x) {
    int* solution = malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main()
{
    int x[3] = {1, 2, 3};
    //Function will create memory for the solution internally
    int* solution = AddThreeToArray(x);
    //print the values of z
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```



```
solution[0]=4
solution[1]=5
solution[2]=6
```

Requesting Memory on the Heap

```
#include <stdlib.h>
void* malloc(size_t size);
```

  - size_t is an unsigned integer data type defined in <stdlib.h>
  - used to represent sizes of objects in bytes

- **If** successful, a call to malloc(n) returns a generic pointer (void *)
  - It points to a memory block of n bytes on the heap
- If not successful, NULL is returned

Generic pointers: void*

Pointer to a memory block whose content is “un-typed”

- Use for raw memory operations or in generic functions
- Automatic casting when assigned to other pointer types

```
int * pox = malloc(6 * sizeof(int));
```

- Requires casting before dereferencing for read / write

```
    *(int *)pv;  // use pv as an int *
```

- NULL, a special pointer value useful for initializations, error handling

  #define NULL ((void*)0)

The Rolling Stones once famously sang:

“You can’t always get what you want.”

What where they singing about?

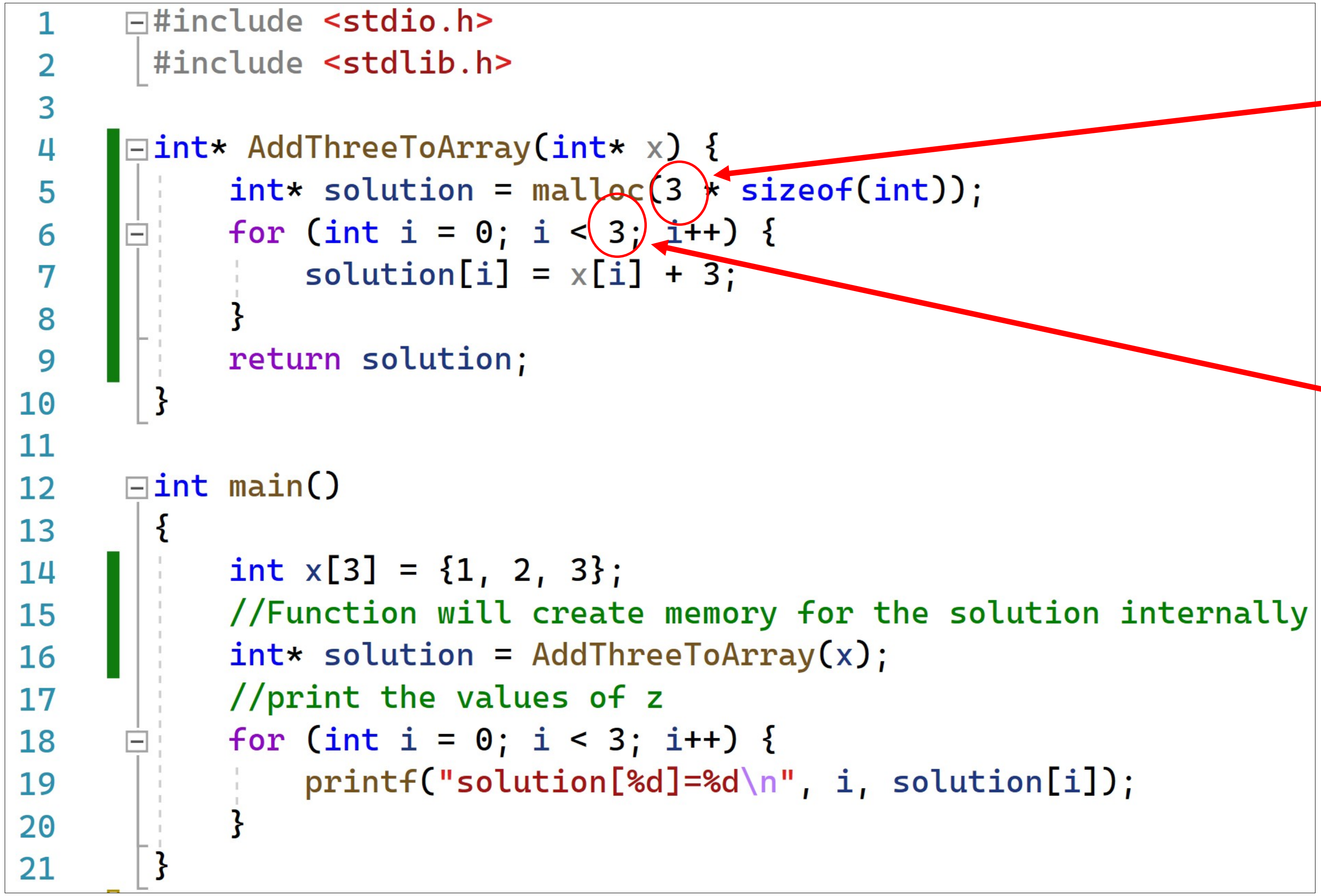
1. About the disappointments in life. (unlikely)
2. About a call to **malloc** failing because you don’t have enough computer memory.

You Can’t Always Get What You Want

- A call to malloc() may fail
  - For example, if you are out of memory
- In this case you get back a NULL value
  - Not much to do except report the error (and terminate nicely)

```
char* p = malloc(100); // request 100 bytes
if (p == NULL) {
      // report error and finish
    perror("Not enough memory");
    exit(1);
}
```

Any lingering issues with our code?



```c
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x) {
    int* solution = malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main()
{
    int x[3] = {1, 2, 3};
    //Function will create memory for the solution internally
    int* solution = AddThreeToArray(x);
    //print the values of z
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

We still have hard coded variables!

We can further break this problem down into two cases:

1. We are working with an array that has a pre-declared size:

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

2. We are working with an array created using malloc:

```c
int* solution = malloc(3 * sizeof(int));
```

Case 1: Counting Elements in Pre-Declared Arrays

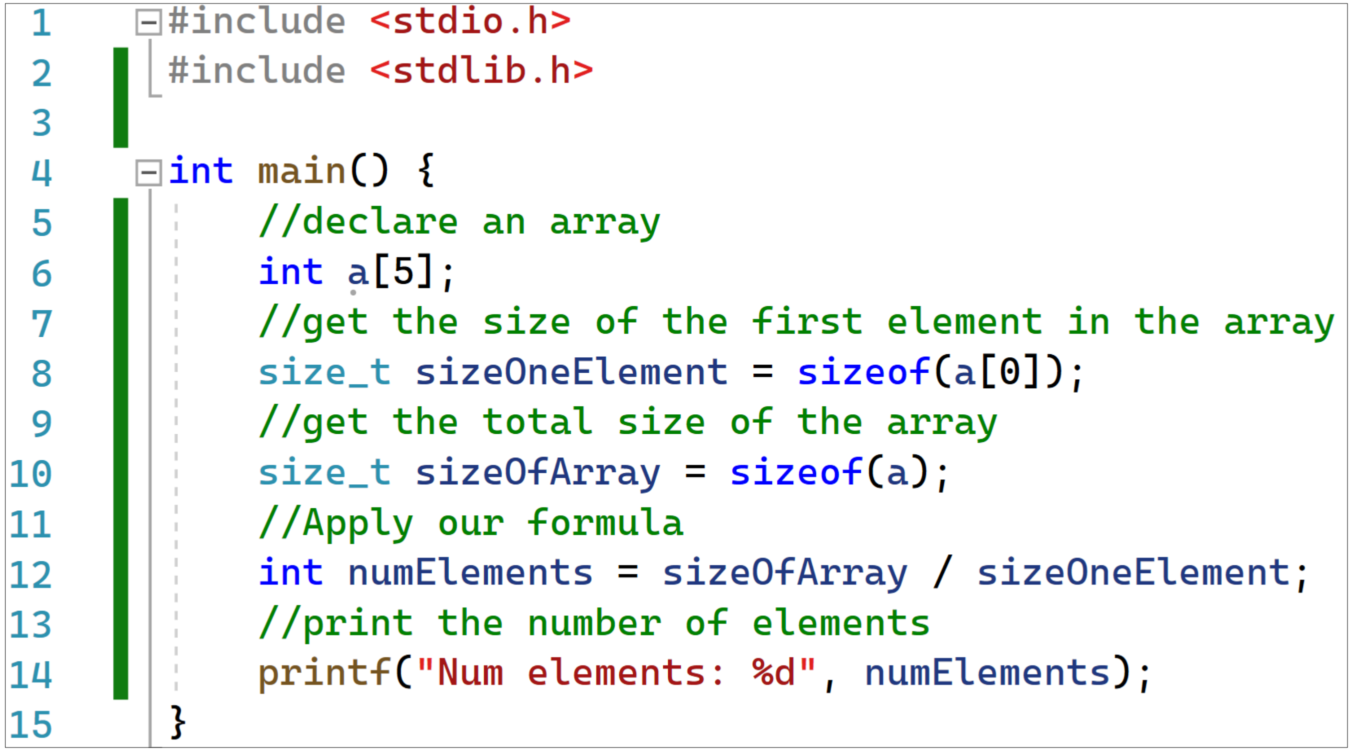
- When passed an array not declared using malloc, how do we know its size?

1. We need to know what type of data the array holds.
2. If we know the type of data we can divide the total size of the array by the size of one element of data. That gives us the number of elements in the array.

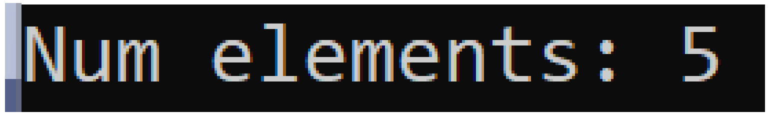
$$Number\ of\ elements \in Array = \frac{\text{¿}\, the\ Array\ (\text{¿}\, bytes)}{\text{¿}\, 1\ Array\ Element\ (\text{¿}\, bytes)}$$

Case 1: Counting Elements in Pre-Declared Arrays

$$Number\ of\ elements \in Array = \frac{\text{¿}\,the\ Array\ (\text{¿}\,bytes)}{\text{¿}\,1\ Array\ Element\ (\text{¿}\,bytes)}$$



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //declare an array
    int a[5];
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(a[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(a);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```



```
Num elements: 5
```

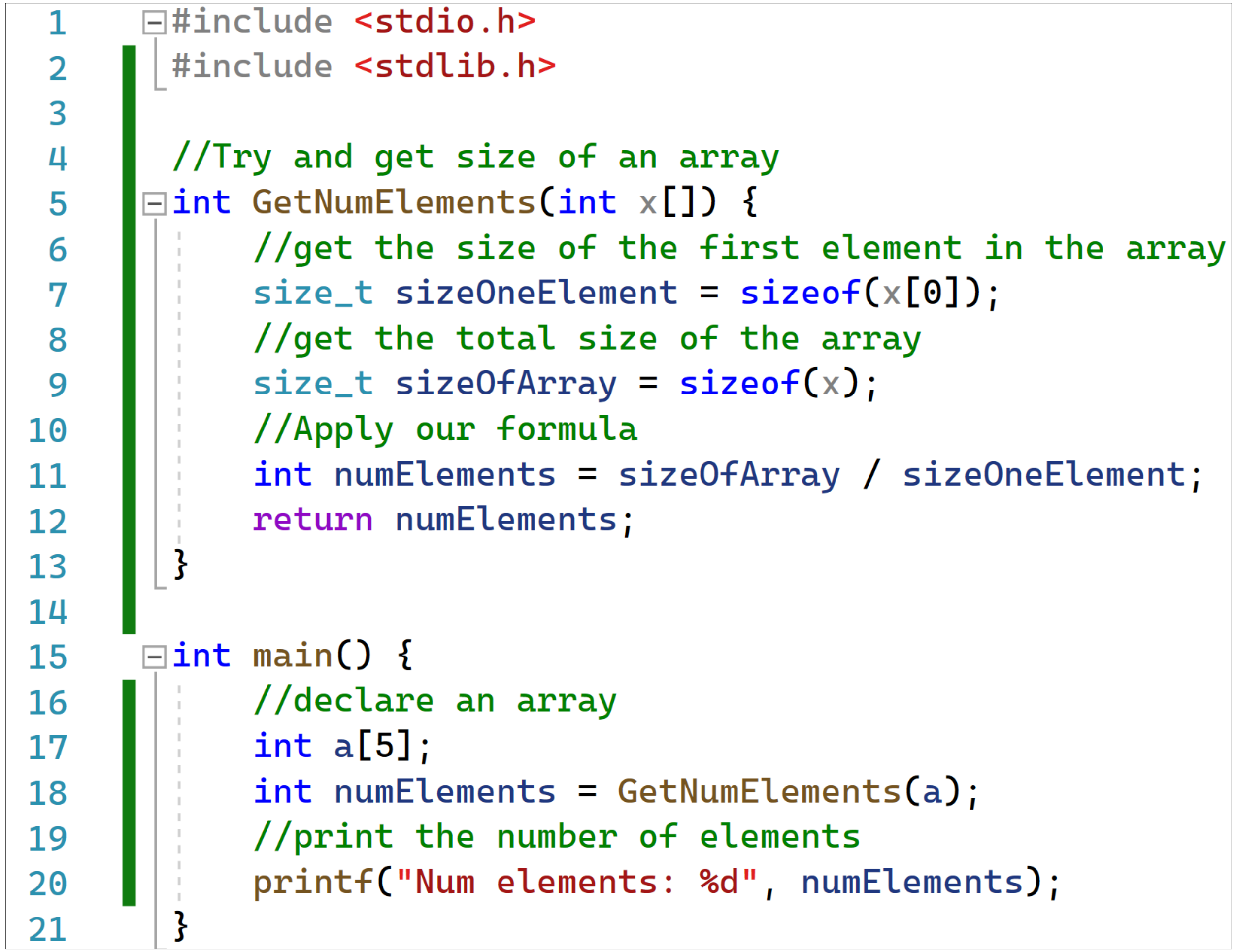
Follow up Question:

$$Number\ of\ elements \in Array = \frac{¿\, the\ Array\ (¿\, bytes)}{¿\, 1\ Array\ Element\ (¿\, bytes)}$$

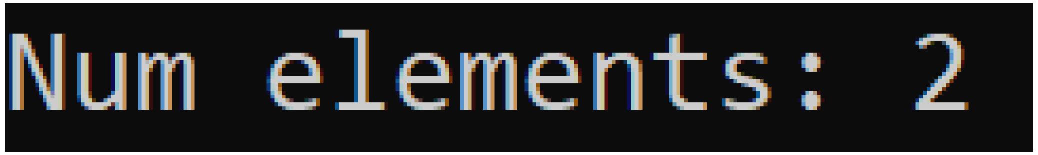
If all we need is the simple formula above why isn’t it already built into C as a function?

Most of the class right now:

Trying to build our own function to find number of elements...



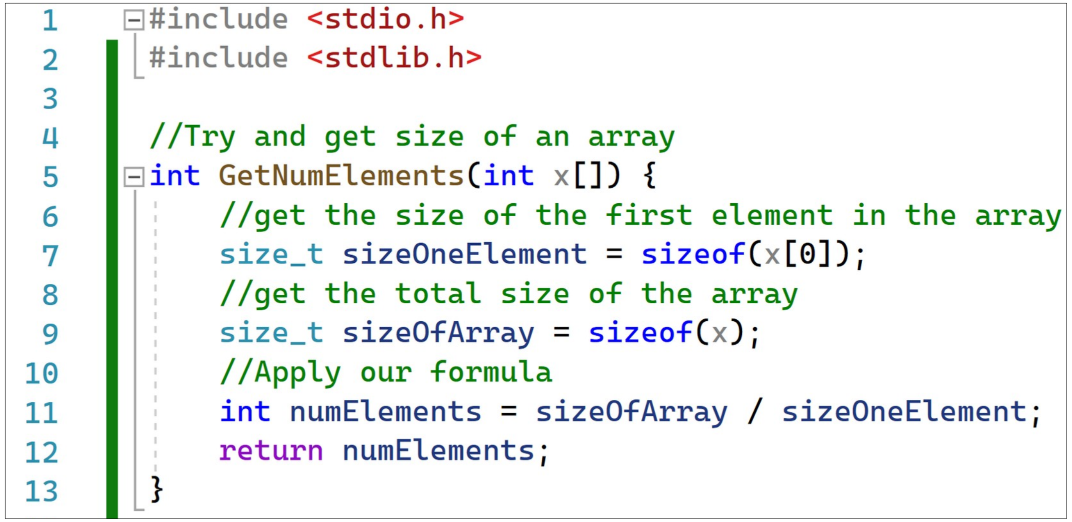
```c
#include <stdio.h>
#include <stdlib.h>

//Try and get size of an array
int GetNumElements(int x[]) {
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(x[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(x);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    return numElements;
}

int main() {
    //declare an array
    int a[5];
    int numElements = GetNumElements(a);
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```



```
Num elements: 2
```

Explaining the Issue

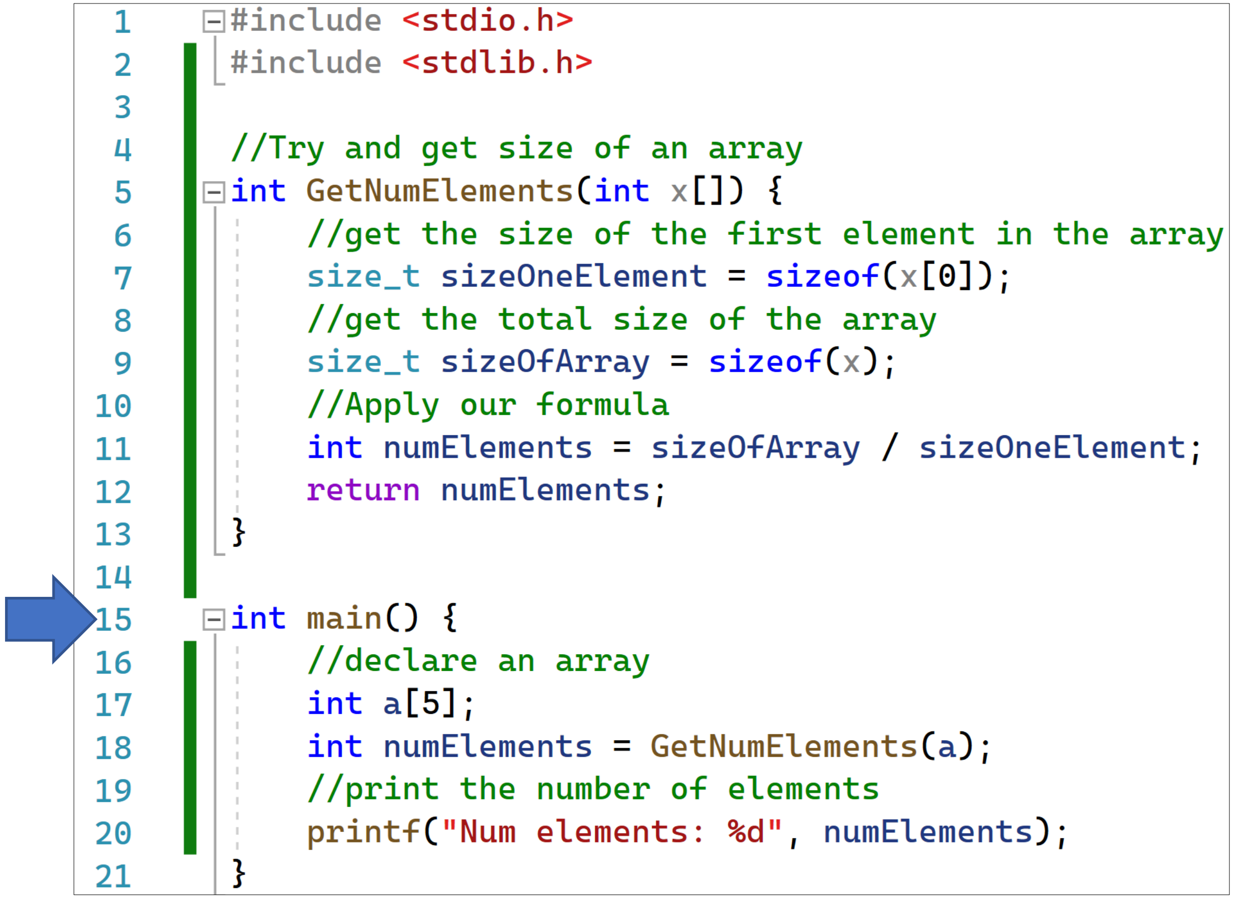
- In C how are arrays passed to functions?
- They are passed as POINTERS.
- The function doesn’t see the entire array. It only sees a pointer to the first element in the array.



```c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  //Try and get size of an array
5  int GetNumElements(int x[]) {
6      //get the size of the first element in the array
7      size_t sizeOneElement = sizeof(x[0]);
8      //get the total size of the array
9      size_t sizeOfArray = sizeof(x);
10     //Apply our formula
11     int numElements = sizeOfArray / sizeOneElement;
12     return numElements;
13 }
```

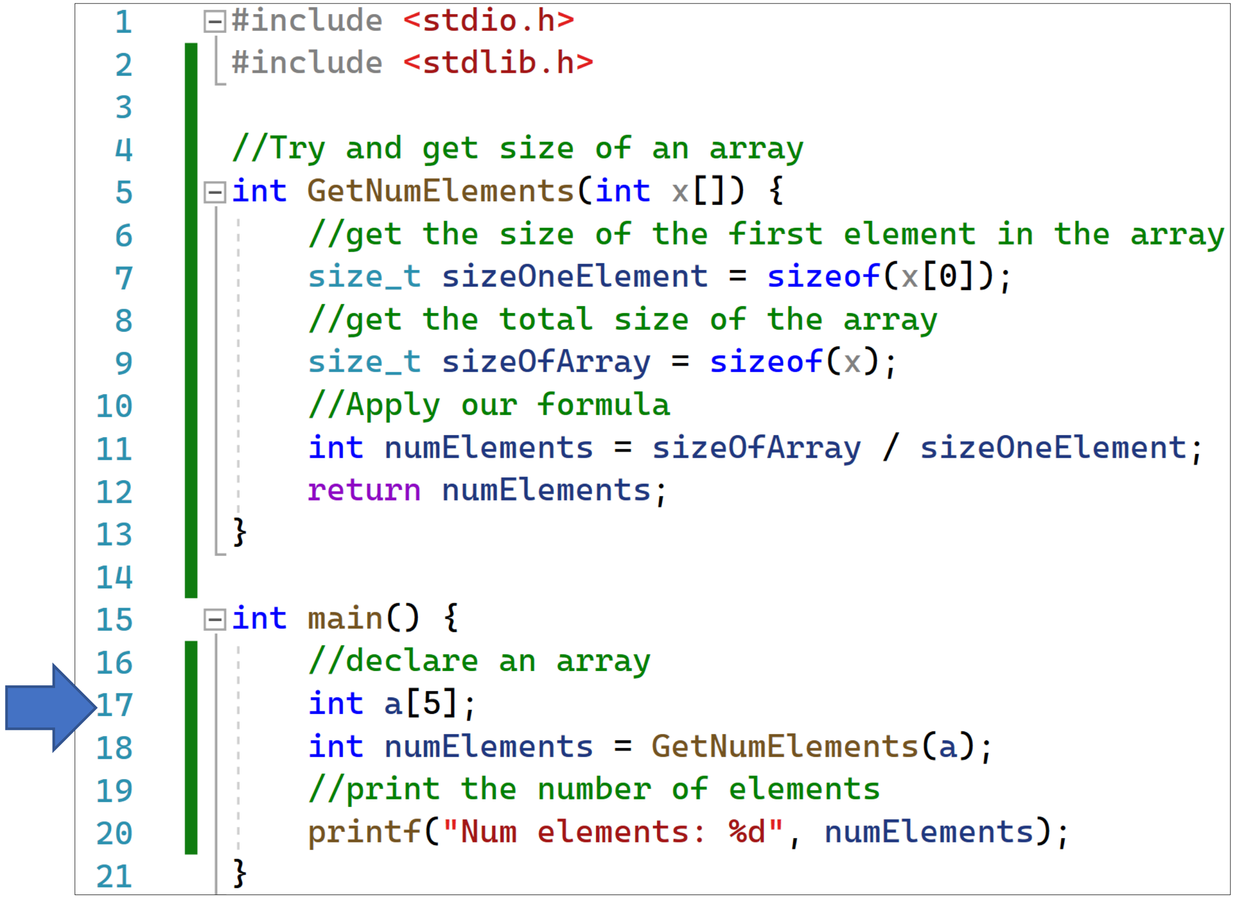
- When we are running code inside the function in line 5, who is x and what is its size?
- x is a pointer that points to the first element of x in memory. See next slide for a picture.

Pictorial View of What is Going on…

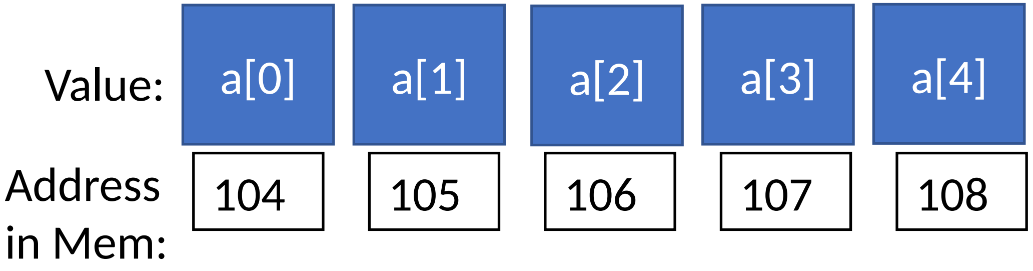


```c
#include <stdio.h>
#include <stdlib.h>

//Try and get size of an array
int GetNumElements(int x[]) {
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(x[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(x);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    return numElements;
}

int main() {
    //declare an array
    int a[5];
    int numElements = GetNumElements(a);
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```

Pictorial View of What is Going on…

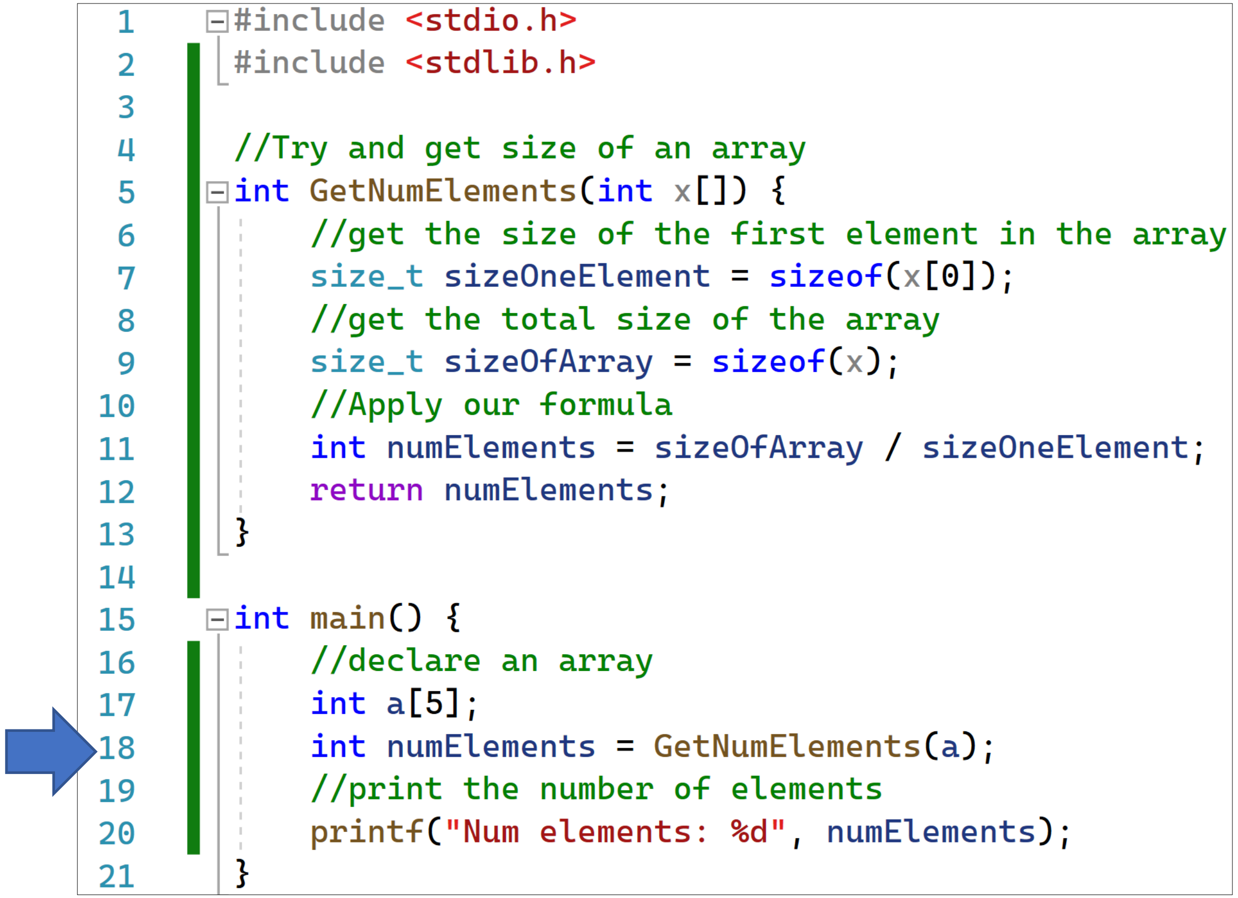


```c
#include <stdio.h>
#include <stdlib.h>

//Try and get size of an array
int GetNumElements(int x[]) {
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(x[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(x);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    return numElements;
}

int main() {
    //declare an array
    int a[5];
    int numElements = GetNumElements(a);
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```

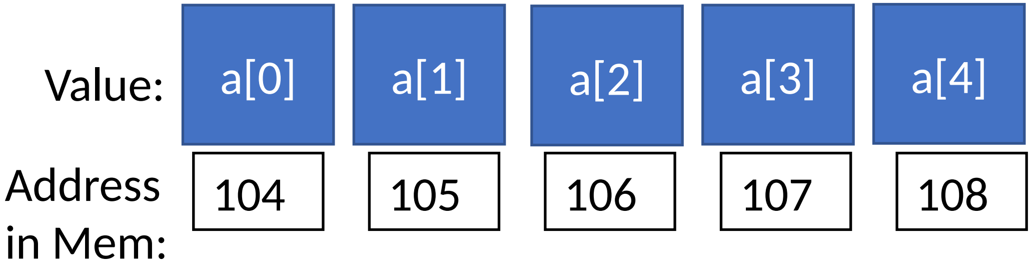


Value:	a[0]	a[1]	a[2]	a[3]	a[4]
Address in Mem:	104	105	106	107	108

Pictorial View of What is Going on…

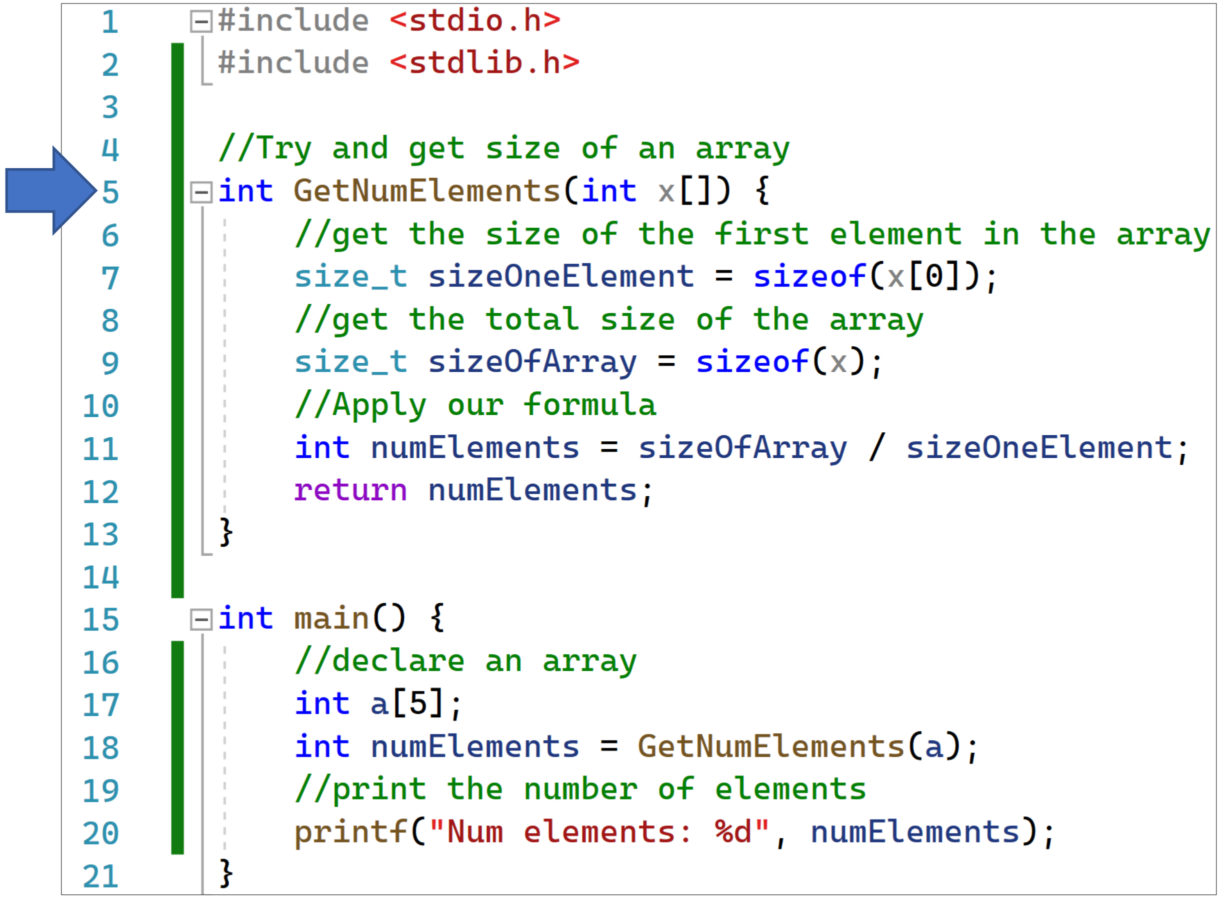


```c
#include <stdio.h>
#include <stdlib.h>

//Try and get size of an array
int GetNumElements(int x[]) {
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(x[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(x);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    return numElements;
}

int main() {
    //declare an array
    int a[5];
    int numElements = GetNumElements(a);
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```

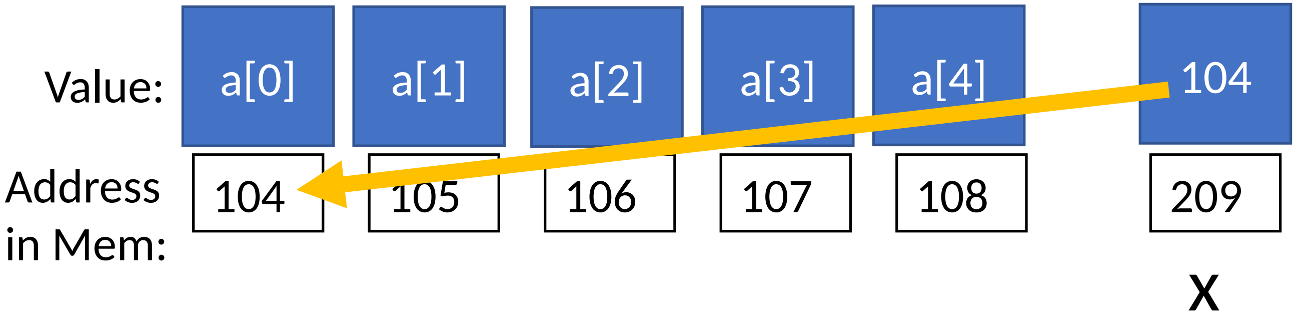


Value:	a[0]	a[1]	a[2]	a[3]	a[4]
Address in Mem:	104	105	106	107	108

Pictorial View of What is Going on…

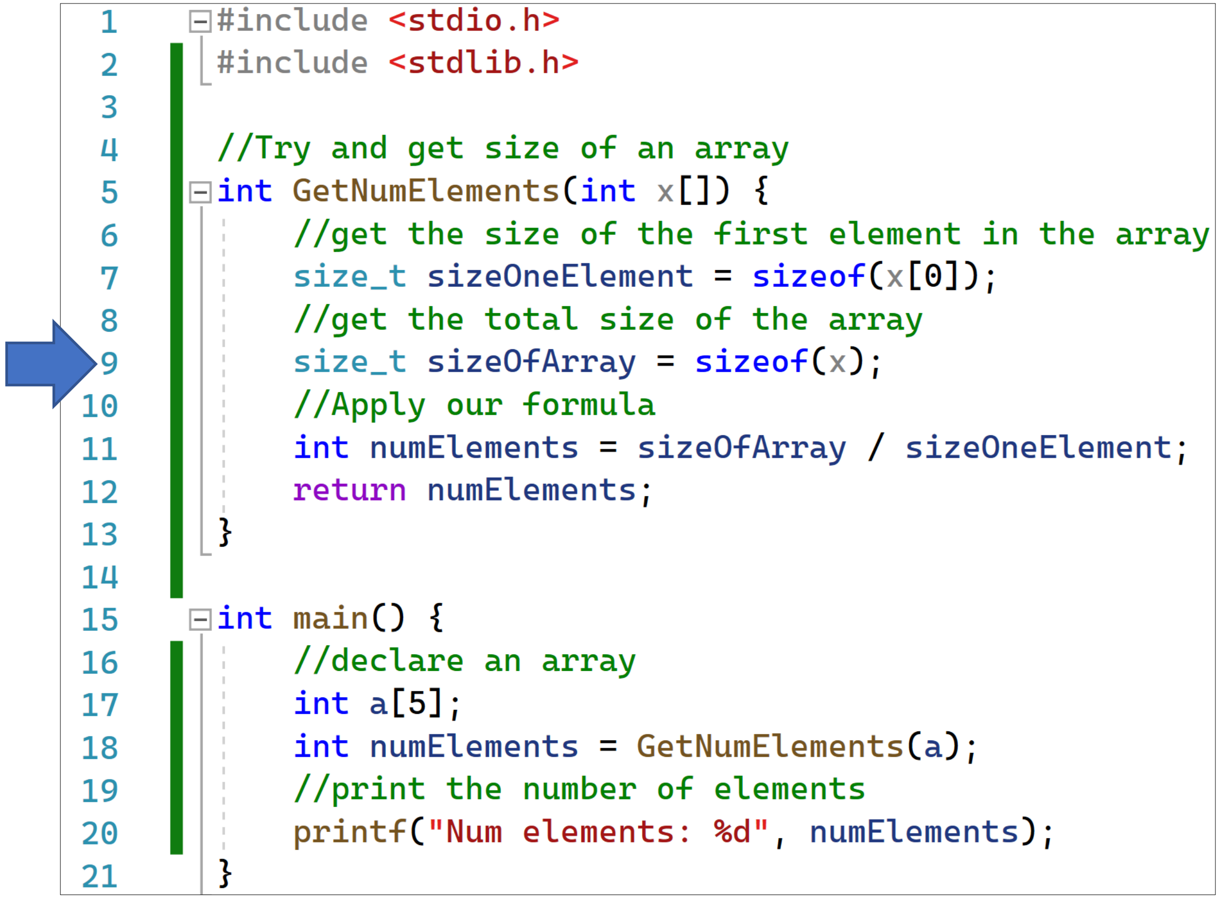


```c
#include <stdio.h>
#include <stdlib.h>

//Try and get size of an array
int GetNumElements(int x[]) {
    //get the size of the first element in the array
    size_t sizeOneElement = sizeof(x[0]);
    //get the total size of the array
    size_t sizeOfArray = sizeof(x);
    //Apply our formula
    int numElements = sizeOfArray / sizeOneElement;
    return numElements;
}

int main() {
    //declare an array
    int a[5];
    int numElements = GetNumElements(a);
    //print the number of elements
    printf("Num elements: %d", numElements);
}
```

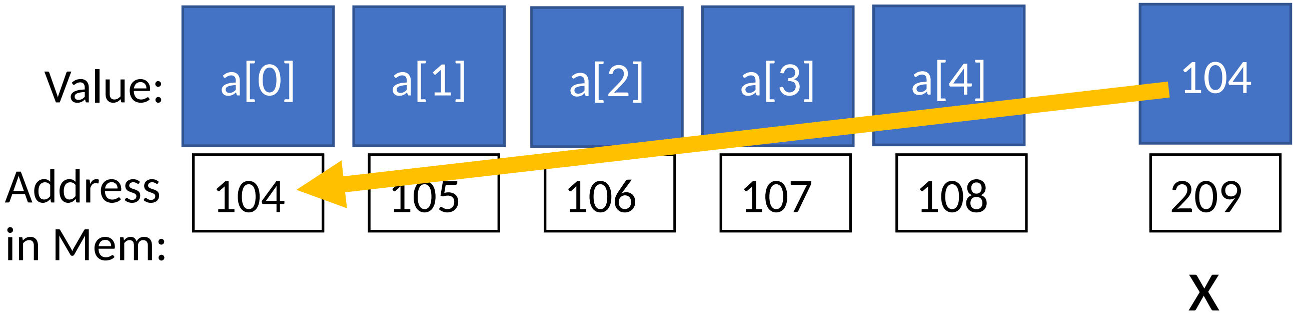


- Remember when we call the function, our array is passed in BY reference.
- This means we get a variables whose value is the first memory address of array “a”.
- Do we know where “a” ends? Not when the variable is passed to us by reference.

Pictorial View of What is Going on…



```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  //Try and get size of an array
5  int GetNumElements(int x[]) {
6      //get the size of the first element in the array
7      size_t sizeOneElement = sizeof(x[0]);
8      //get the total size of the array
9      size_t sizeOfArray = sizeof(x);
10     //Apply our formula
11     int numElements = sizeOfArray / sizeOneElement;
12     return numElements;
13 }
14
15 int main() {
16     //declare an array
17     int a[5];
18     int numElements = GetNumElements(a);
19     //print the number of elements
20     printf("Num elements: %d", numElements);
21 }
```

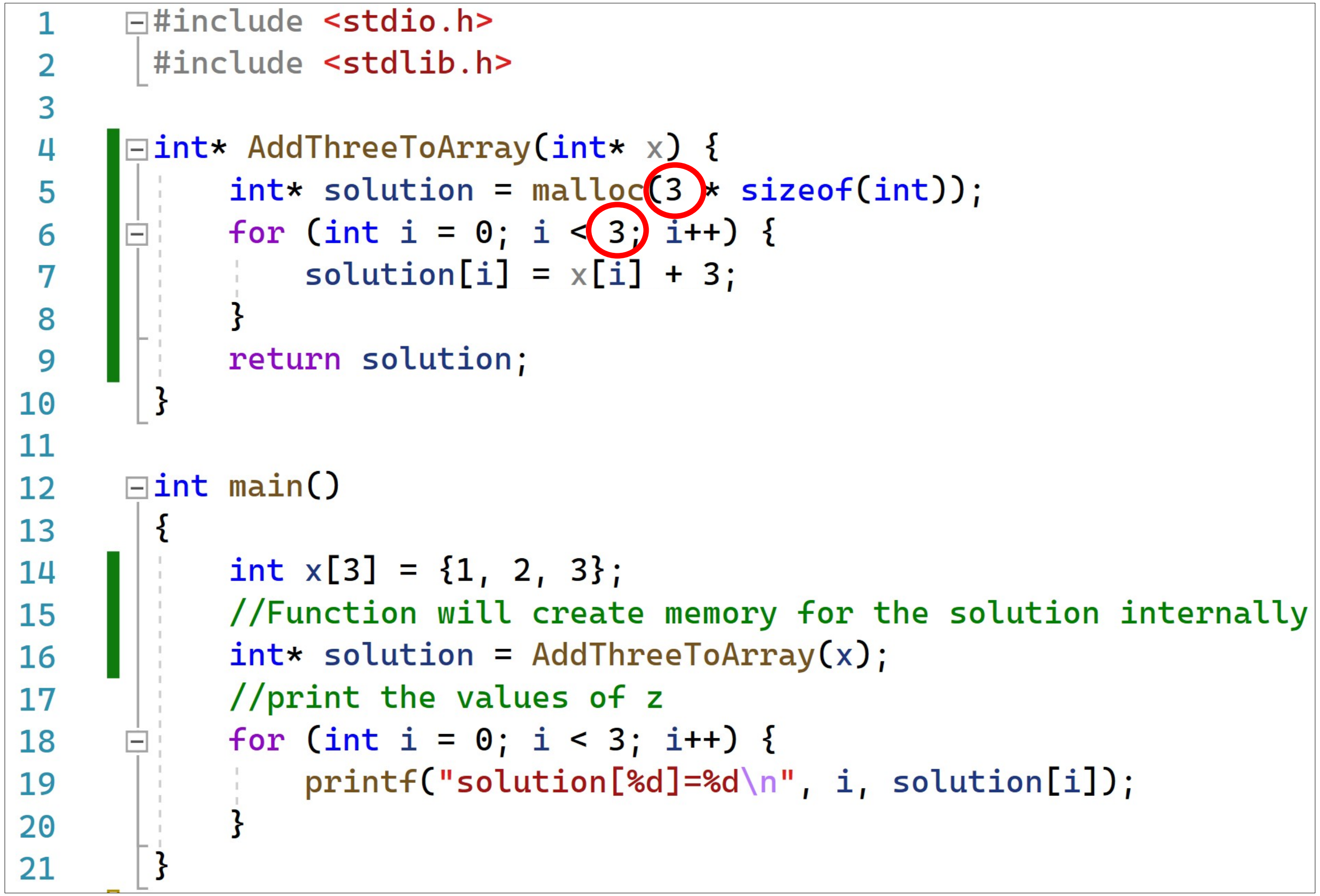


Value:	a[0]	a[1]	a[2]	a[3]	a[4]	104
Address in Mem:	104	105	106	107	108	209
						x

- Skip ahead a bit to line 9. What is the size of x?
- x is a pointer so the size of x is whatever the size of a pointer will be.
- x is NOT the size of the array “a”.

Back to our original question…

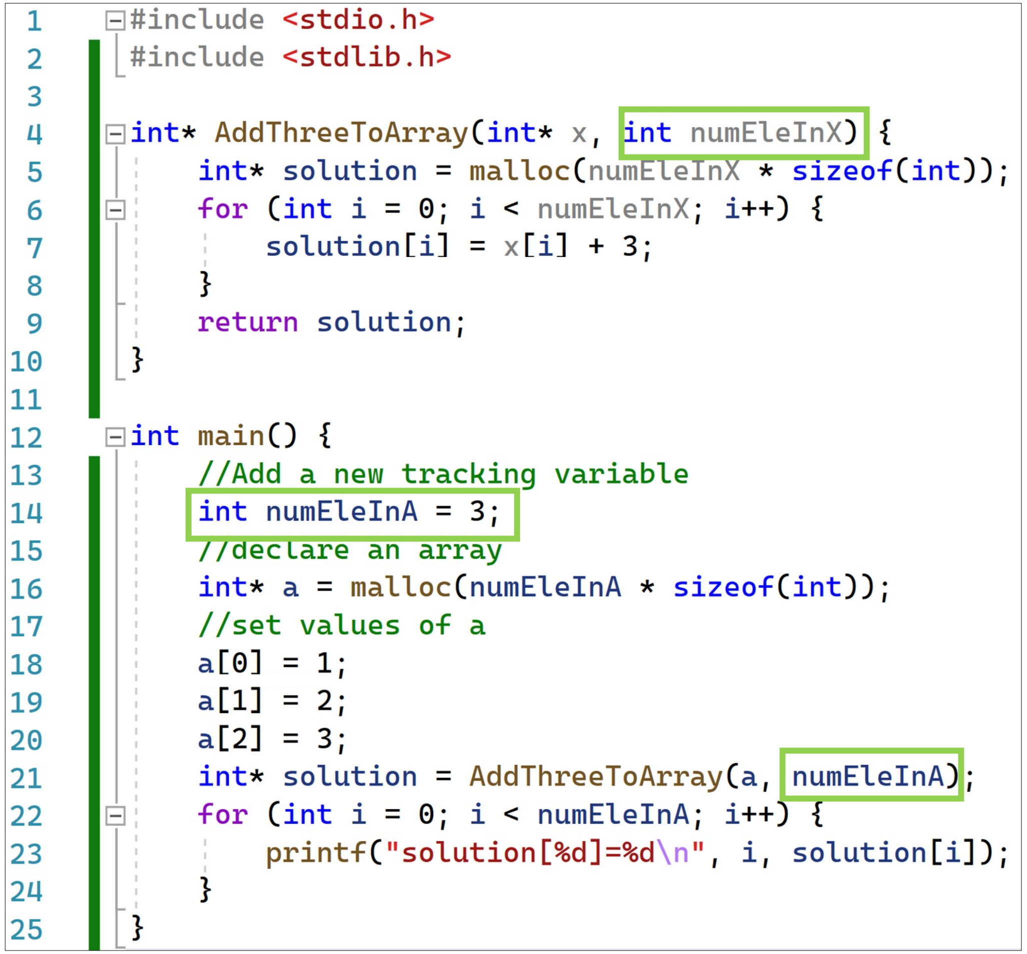
When passed an array not declared using malloc, how do we know its size and how do we remove the hard coded variables?



```c
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x) {
    int* solution = malloc(3 * sizeof(int));
    for (int i = 0; i < 3; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main()
{
    int x[3] = {1, 2, 3};
    //Function will create memory for the solution internally
    int* solution = AddThreeToArray(x);
    //print the values of z
    for (int i = 0; i < 3; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

Back to our question…

When passed an array, how do we know its size and how do we remove the hard coded variables?



```
#include <stdio.h>
#include <stdlib.h>

int* AddThreeToArray(int* x, int numEleInX) {
    int* solution = malloc(numEleInX * sizeof(int));
    for (int i = 0; i < numEleInX; i++) {
        solution[i] = x[i] + 3;
    }
    return solution;
}

int main() {
    //Add a new tracking variable
    int numEleInA = 3;
    //declare an array
    int* a = malloc(numEleInA * sizeof(int));
    //set values of a
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
    int* solution = AddThreeToArray(a, numEleInA);
    for (int i = 0; i < numEleInA; i++) {
        printf("solution[%d]=%d\n", i, solution[i]);
    }
}
```

When you find out you have to keep track of the array size in C yourself:

Recall we started by trying to break the problem down into two parts:

1. ~~We are working with an array that has a pre-declared size:~~

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

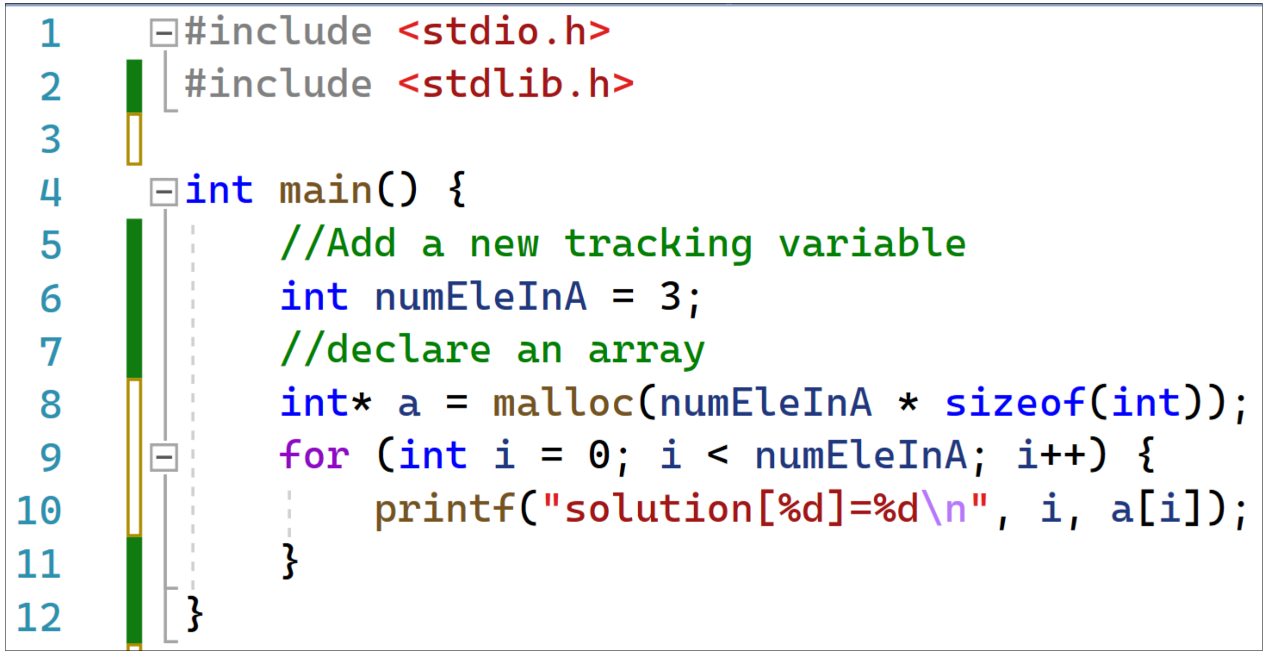
2. We are working with an array created using malloc:

```
int* solution = malloc(3 * sizeof(int));
```

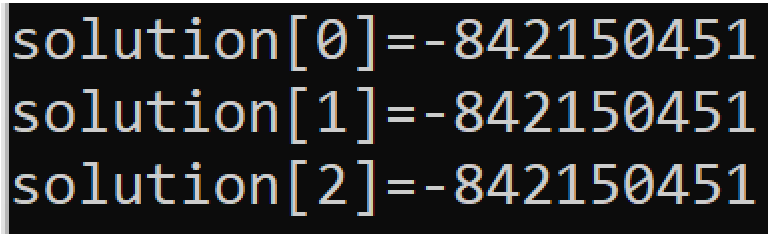
What about for the second case?

Short Answer: Malloc creates new memory by giving you a pointer to a memory address. You need to keep track, no function can double check for you later.

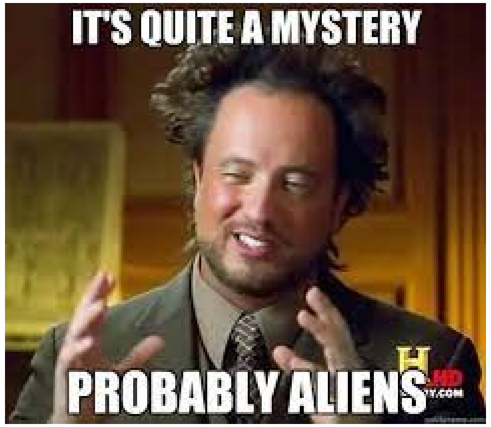
Another Question: Who lives in the empty array?



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //Add a new tracking variable
    int numEleInA = 3;
    //declare an array
    int* a = malloc(numEleInA * sizeof(int));
    for (int i = 0; i < numEleInA; i++) {
        printf("solution[%d]=%d\n", i, a[i]);
    }
}
```



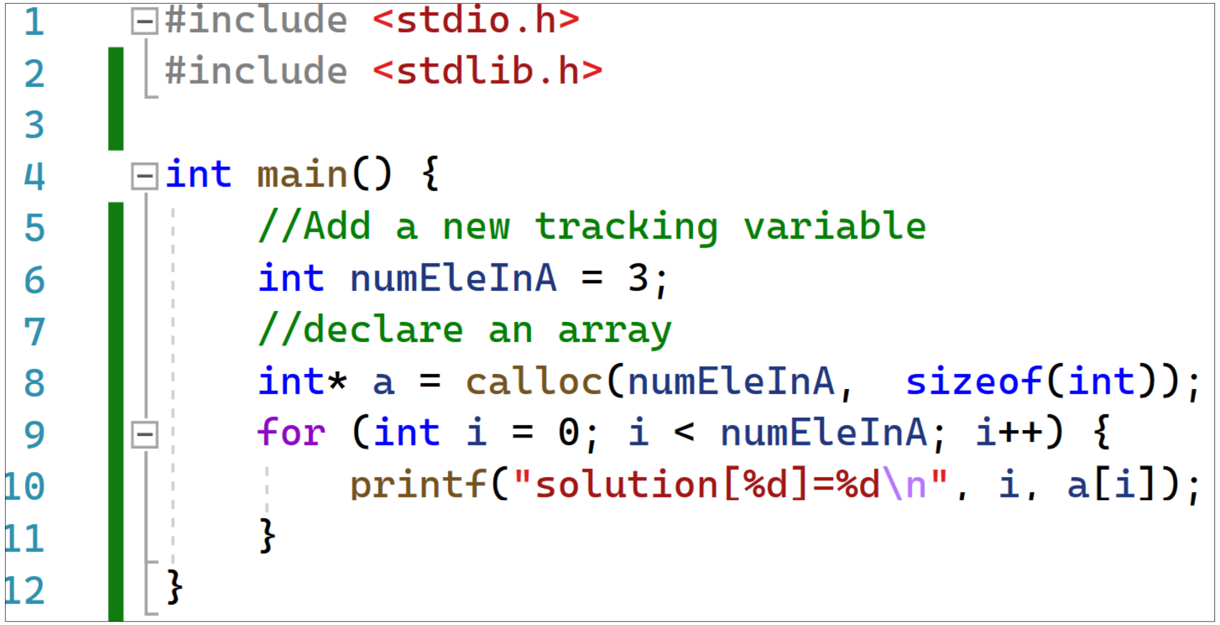
```
solution[0]=-842150451
solution[1]=-842150451
solution[2]=-842150451
```



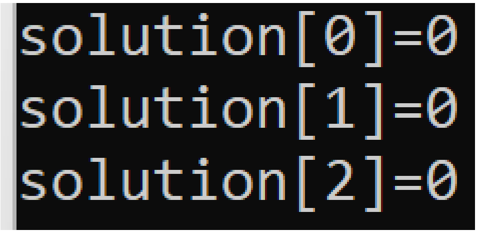
Creating arrays AND setting their values: calloc

```
#include <stdlib.h>
void* calloc(size_t nmemb, size_t size);
```

- calloc() is implemented in terms of malloc()
  - **calloc() also initializes the content to 0**



```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    //Add a new tracking variable
    int numEleInA = 3;
    //declare an array
    int* a = calloc(numEleInA,  sizeof(int));
    for (int i = 0; i < numEleInA; i++) {
        printf("solution[%d]=%d\n", i, a[i]);
    }
}
```



```
solution[0]=0
solution[1]=0
solution[2]=0
```

Adjusting array size

```
#include <stdlib.h>
void* realloc (void* ptr, size_t size);
```

What if you change your mind?

- You requested 100 bytes, but now need 200!

```
char* p = malloc(100); // request 100 bytes
...
p = realloc(p, 200);  // p may change!
```

- Before a call to realloc(p, size), p must be
  - A pointer returned by a previous malloc/calloc/realloc
  - Or NULL, in which case the call is equivalent to malloc(size)

Deallocation

```
#include <stdlib.h>
void free(void *ptr);
```

- Straightforward
  - Simply call the library function “free”
  - Takes a pointer to the block to free

```
free(ptr);
```

- Do not free a pointer twice!
  - After freeing a pointer, set it to NULL!

Two key rules for using memory

Rule 1: Everything you requested should be freed, eventually

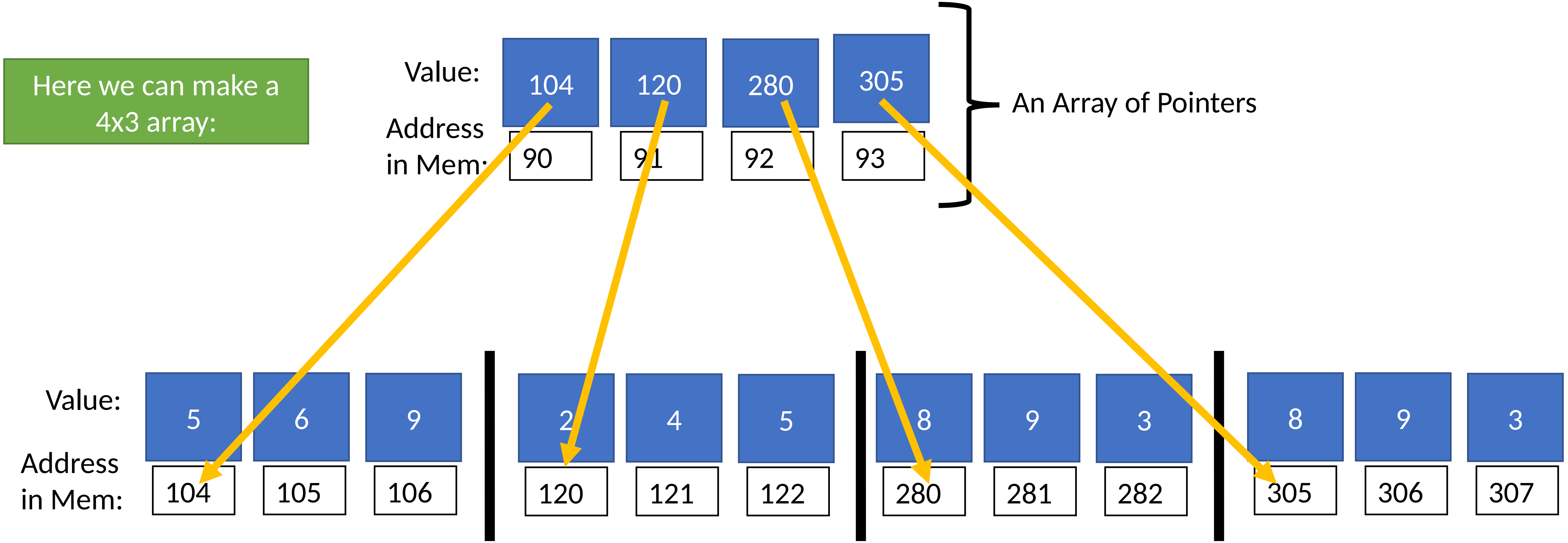
Rule 2: Only free what is allocated via malloc/calloc/realloc

- Consequences of not following the rules
  - **Memory “leaks”**
    - Your program will eventually run out of memory
  - **Undefined behavior and horrible crashes**
    - Freeing unallocated memory or already freed memory
      - May cause a memory error and a program crash
      - Worse, may corrupt the heap and cause a crash later
      - Even worse, the program may keep running, totally corrupting your data, and writing it to disk without you realising

So far we have looked at 1D arrays with pointers…

- How could we make 2D arrays with pointers? Use an array of pointers!

Here we can make a 4x3 array:



Array of Pointers

3 integers, but not in an array:

```
int  a0;
int a1;
int a2;
```

3 integers in an array:

```
// array of int's
int   a[3];
```

3 arrays each of size 10:

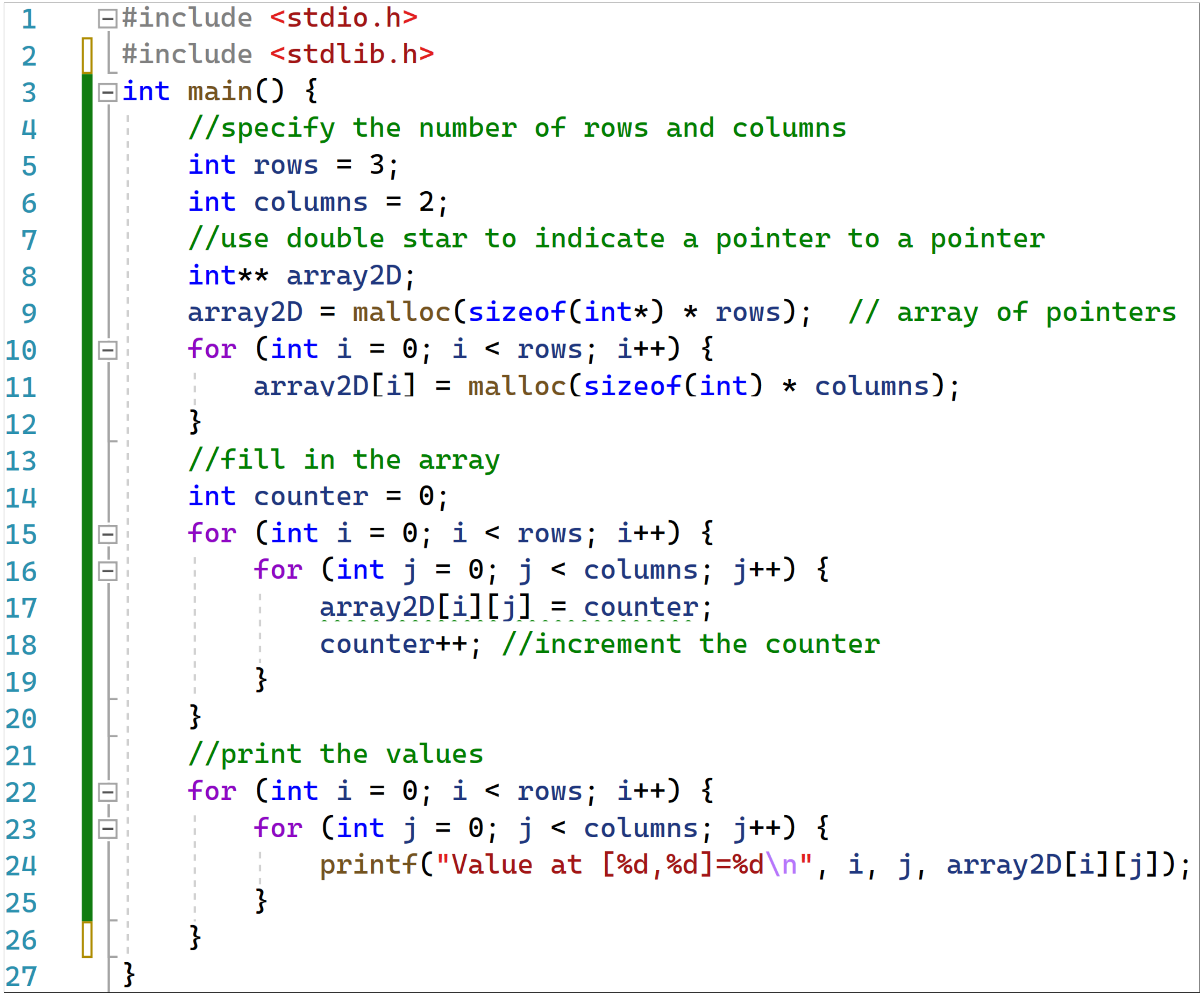
```
char *p0 = malloc(10);
char *p1 = malloc(10);
char *p2 = malloc(10);
```

One 2D array of 3x10

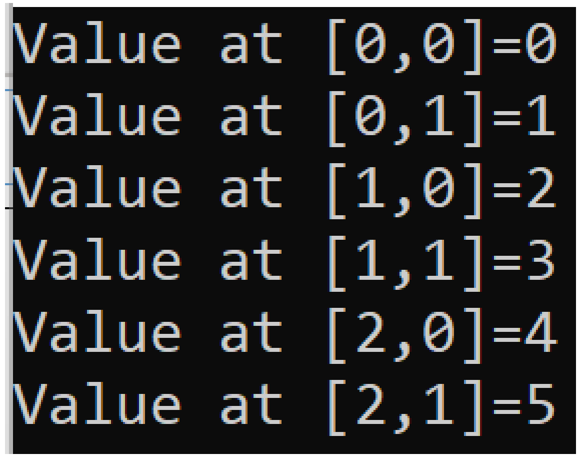
```
// array of pointers
char * p[3];

// Can also do with a loop
p[0] = malloc(10);
p[1] = malloc(10);
p[2] = malloc(10);
```

Example: allocating 2d dynamical array

```
void doSomething(int m, int n)
{ int **rows;
  rows = malloc(sizeof(int *) * m);  // array of pointers
  for (int i = 0; i < m; i++)
    rows[i] = malloc(sizeof(int)*n); // one int array for each row
  ...
  for (int i = 0; i < m; i++)
    free(rows[i]);
  free(rows);
}
```

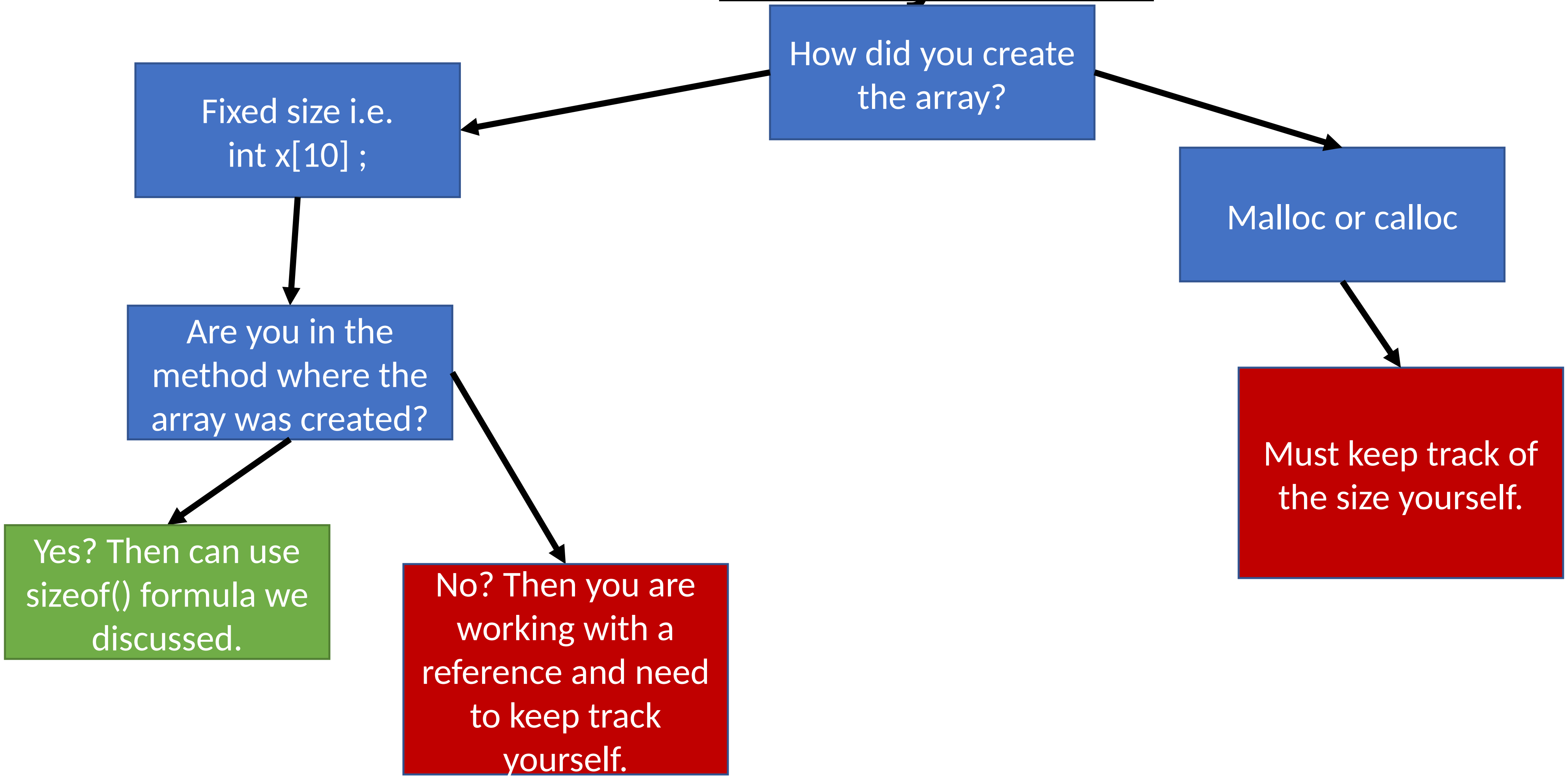


```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    //specify the number of rows and columns
    int rows = 3;
    int columns = 2;
    //use double star to indicate a pointer to a pointer
    int** array2D;
    array2D = malloc(sizeof(int*) * rows);  // array of pointers
    for (int i = 0; i < rows; i++) {
        array2D[i] = malloc(sizeof(int) * columns);
    }
    //fill in the array
    int counter = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            array2D[i][j] = counter;
            counter++; //increment the counter
        }
    }
    //print the values
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < columns; j++) {
            printf("Value at [%d,%d]=%d\n", i, j, array2D[i][j]);
        }
    }
}
```



```
Value at [0,0]=0
Value at [0,1]=1
Value at [1,0]=2
Value at [1,1]=3
Value at [2,0]=4
Value at [2,1]=5
```

Conclusions 1: How to keep track of array size



How did you create the array?

Fixed size i.e.
int x[10] ;

Are you in the method where the array was created?

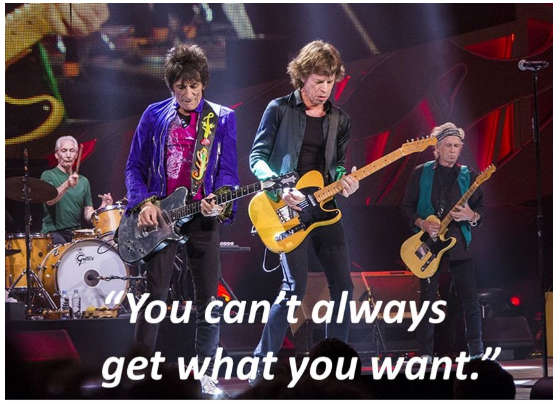
Yes? Then can use sizeof() formula we discussed.

No? Then you are working with a reference and need to keep track yourself.

Malloc or calloc

Must keep track of the size yourself.

Conclusions 2



When you find out you have to keep track of the array size in C yourself:

- Using Malloc, calloc and free are important concepts in C that you should know how to use.
- You need to understand arrays are created in different sections of the memory depending on how you instantiate the array.
- Two rules for memory:

Rule 1: Everything you requested should be freed, eventually

Rule 2: Only free what is allocated via malloc/calloc/realloc

- We can create multi-dimensional arrays using arrays of pointers.

Figure Sources

1. https://wompampsupport.azureedge.net/fetchimage?siteId=7575&v=2&jpgQuality=100&width=700&url=https%3A%2F%2Fi.kym-cdn.com%2Fentries%2Ficons%2Ffacebook%2F000%2F031%2F197%2Fvisible.jpg
2. https://upload.wikimedia.org/wikipedia/commons/1/10/The_Rolling_Stones_Summerfest_in_Milwaukee_-_2015.jpg
3. https://ychef.files.bbci.co.uk/976x549/p03lcphh.jpg
4. https://www.pngitem.com/pimgs/m/11-115218_iphone-thinking-emoji-hd-png-download.png
5. http://www.quickmeme.com/meme/357oc1
6. https://www.deviantart.com/swiftwin4ds/art/Reality-is-often-disappointing-785218817