CPS 188

Computer Programming Fundamentals

Prof. Alex Ufkes

Topic 7.1: Arrays

Toronto Metropolitan University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

Today

Arrays

Declaration & Usage

Arrays & Loops

Arrays & Functions

2D Arrays, Enums

Suppose you want to store 5 integers.

```
int n1, n2, n3, n4, n5;
```

Easy.

Suppose you want to store 50 integers.

```
int n01, n02, n03, n04, n05, n06, n07, n08, n09, n10;
int n11, n12, n13, n14, n15, n16, n17, n18, n19, n20;
int n21, n22, n23, n24, n25, n26, n27, n28, n29, n30;
int n31, n32, n33, n34, n35, n36, n37, n38, n39, n40;
int n41, n42, n43, n44, n45, n46, n47, n48, n49, n50;
```

Tedious, but easy.

Suppose you want to store **500000** integers.

We need something new!

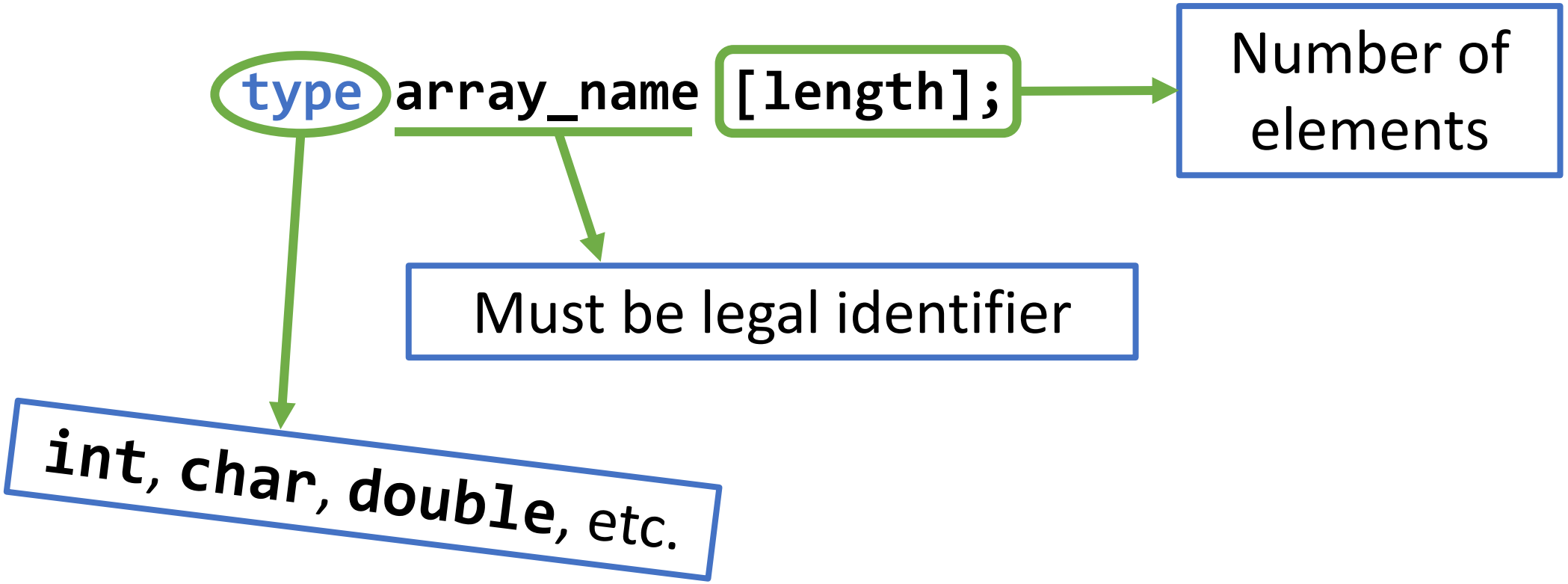
Arrays

An array is a sequence of values of the same type:

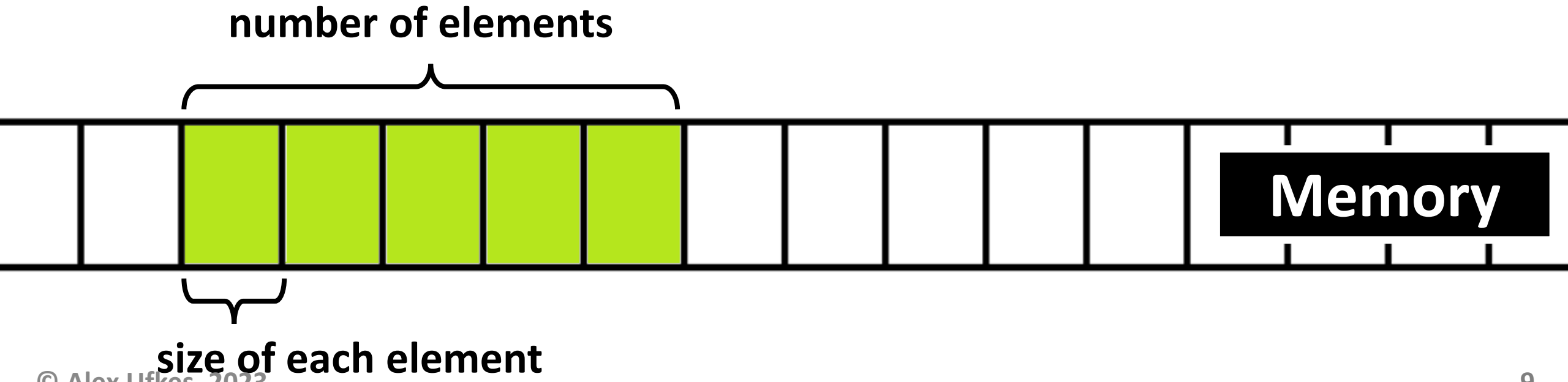
```
#include <stdio.h>

int main()
{
    int nums[100];
    return 0;
}
```

An array of 100 integers

Array Syntax



```c
#include <stdio.h>

int main()
{
    int nums[5];
    return 0;
}
```



```
#include <stdio.h>

int main()
{
    int nums[5] = {2, 4, 6, 8, 10};
    return 0;
}
```

Declare and initialize:

Declare and initialize:

```
#include <stdio.h>

int main()
{
    int nums[] = {2, 4, 6, 8, 10};
    return 0;
}
```

Size will be equal to the number of initialized elements.

Accessing Elements

Accessing Elements

```
#include <stdio.h>

int main()
{
    int nums[5];

    nums[0] = 17; /* first element  */
    nums[1] = -3; /* second element */
    nums[2] = 0;  /* third element  */
    nums[3] = 57; /* fourth element */
    nums[4] = 3;  /* fifth element  */

    return 0;
}
```

subscript/offset/index

Numbering starts at zero!

Accessing Elements

```
#include <stdio.h>

int main()
{
    int nums[5];

    nums[0] = 17;
    nums[1] = 3 + nums[0];
    nums[2] = nums[1] + nums[0];
    nums[3] = nums[2]*nums[1];
    nums[4] = -nums[3];

    return 0;
}
```

Elements of an integer array can be treated like any other integer!

Numbering starts at <u>zero</u>!

Accessing Elements

```
#include <stdio.h>

int main()
{
    int nums[5];

    scanf("%d", &nums[0]);
    scanf("%d", &nums[1]);
    scanf("%d", &nums[2]);
    scanf("%d", &nums[3]);
    scanf("%d", &nums[4]);

    return 0;
}
```

Address of first element

Elements of an integer array can be treated like any other integer!

Numbering starts at zero!

IMPORTANT!

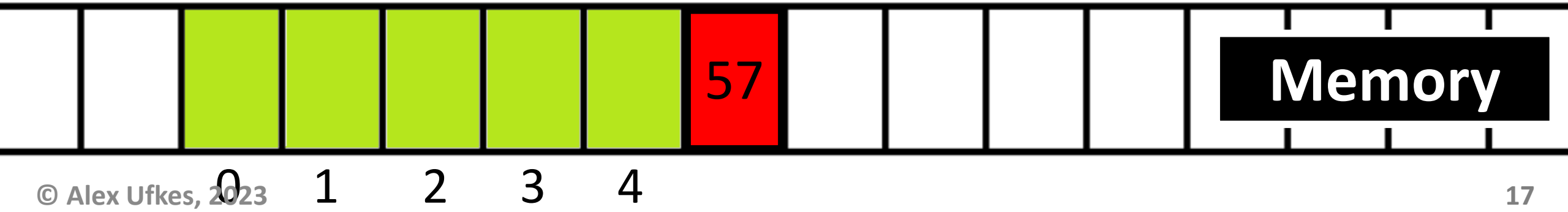
If the array has 5 elements, the valid indexes are 0, 1, 2, 3, 4

```
#include <stdio.h>

int main()
{
    int nums[5];
    nums[5] = 57;   /* BAD! Out of bounds! */
    return 0;
}
```

This ***will*** compile but may crash during run-time.

IMPORTANT!

```
#include <stdio.h>

int main()
{
    int nums[5];
    nums[5] = 57; /* BAD! Out of bounds! */
    return 0;
}
```



So now that we know how to store 500000 numbers easily using an **array**…

How do we assign values to that array?

Writing 500000 assignment statements is just as impossible as declaring 500000 variables.

Arrays & Loops

```
#include <stdio.h>

int main()
{
        int i, nums[5];

        for (i = 0; i < 5; i++)
        {
                nums[i] = i*i;
        }

        return 0;
}
```

i goes from 0 to 4

Perfect!
The valid subscripts in an array with 5 elements are 0 to 4.

Numbering starts at zero!

Arrays & Loops

```
#include <stdio.h>

int main()
{
    int i, nums[5];

    for (i = 0; i < 5; i++)
    {
        scanf("%d", &nums[i]);
    }

    return 0;
}
```

We can fill an array using `scanf` as well!

Numbering starts at <u>zero</u>!

Arrays & Loops

Use a **for** loop to print all elements in an array:

```
int i, nums[5] = {1, 2, 3, 4, 5};

for (i = 0; i < 5; i++)
{
     printf("%d\n", nums[i]);
}
```

Console

```
1
2
3
4
5
```

More Array Rules & Properties

Size can be variable (in newer versions of C):

```
int x;
printf("Enter size: ");
scanf("%d", &x);
int nums[x];
return 0;
```

OK!

Declaration

Other statements

Another Declaration

More Array Rules & Properties

<u>Once the size is set, it cannot be changed</u>

```
int x, nums[100];
nums = nums[200];
return 0;
```

BAD!

- **Once declared, nums is stuck having 100 elements.**
- **Arrays declared with variable length are still fixed in size**

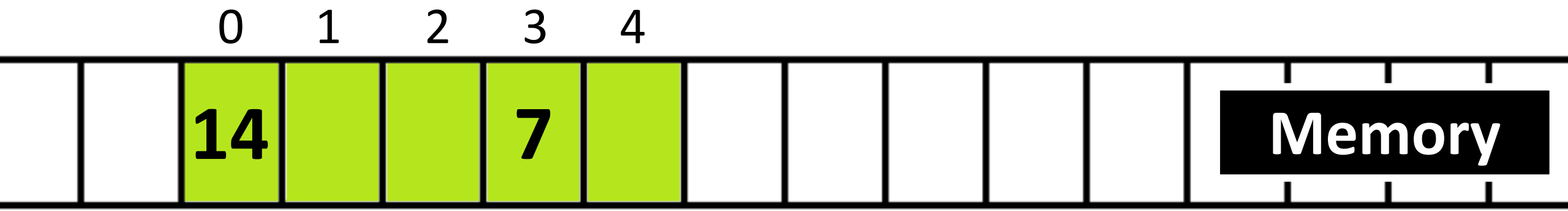
Arrays, Pointers, Addresses

Array name ***without*** an index is the address of the first element.

In other words:

`&nums[0] == nums`

```
int nums[5];
nums[3] = 7
*nums = 2*nums[3];
/* is the same as: */
nums[0] = 2*nums[3];
```



Quincy 2005 - [arrayPointer]

File Edit View Project Debug Tools Window Help

```c
#include <stdio.h>

int main()
{
    int nums[5] = {1, 2, 3, 4, 5};

    printf("%d \n", nums);
    printf("%d \n", &nums[0]);

    printf("%d \n", *nums);
    printf("%d \n", nums[0]);

    return 0;
}
```

Press F1 for help

Quincy 2005

```
6356752
6356752
1
1

Press Enter to return to Quincy...
```

Pointer Arithmetic

These loops do the same thing!

```
int i, nums[5];

for (i = 0; i < 5; i++)
{
     nums[i] = i*i;
}
```

```
int i, nums[5];

for (i = 0; i < 5; i++)
{
     *(nums + i) = i*i;
}
```

Address of first element

```
     nums == &nums[0]
  nums + i == &nums[i]
*(nums + i) == nums[i]
```

`nums` is the *address* of the first element.

`nums+i` is the *address* of the `i`th element.

Quincy 2005 - [test.c]

File Edit View Project Debug Tools Window Help

```c
#include <stdio.h>

int main()
{
    int i, nums[5];

    for (i = 0; i < 5; i++)
    {
        *(nums + i) = i*i;
    }

    for (i = 0; i < 5; i++)
    {
        printf("value: %2d    Address: %d \n",
            *(nums + i), nums + i);
    }

}
```

Press F1 for help Ln 12, Col 28 NUM

Quincy 2005

```
value:  0    Address: 6356748
value:  1    Address: 6356752
value:  4    Address: 6356756
value:  9    Address: 6356760
value: 16    Address: 6356764

Press Enter to return to Quincy...
```

Addresses go up by 4. Why?

Arrays as function arguments

```
#include <stdio.h>

int increment (int i)
{
    return i + 1;
}

{
    int x, k = 4;
    x = increment(k);
    printf("k+1 = %d\n", x);
    return 0;
}
```

Return value gets copied into **x**

Recall:

Argument gets ***copied*** into parameter **i**.
k and **i** are **different** variables!

<u>Write a user-defined void function that does the following:</u>

Changes the first element of an array to a specified value.

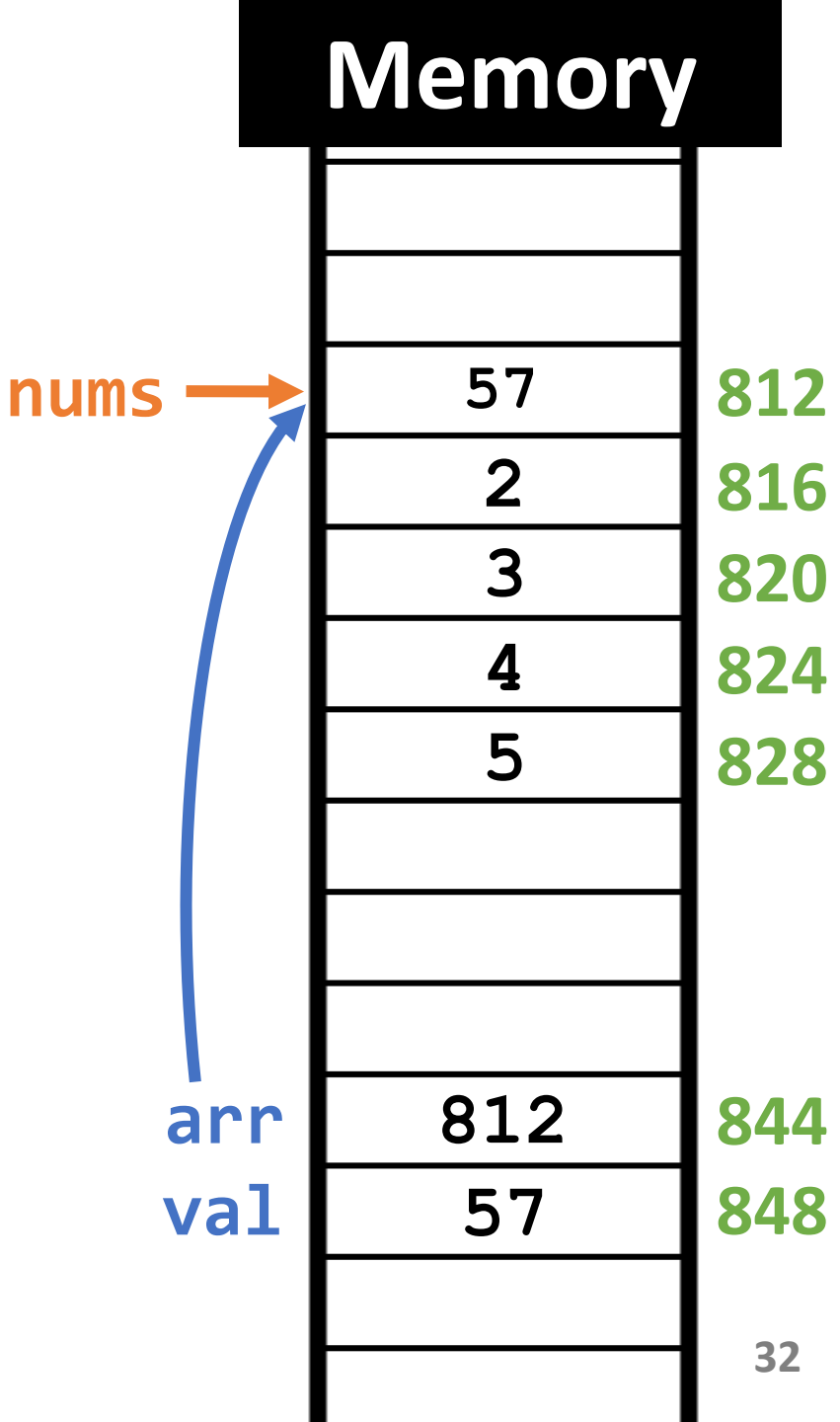
The function will take two arguments: the array, and the value to be changed.

Assume the array has at least one element.

```c
#include <stdio.h>

void change_first( int arr[], int val )
{
    arr[0] = val;
}

int main (void)
{
    int nums[5] = {1, 2, 3, 4, 5};
    printf("before: %d\n", nums[0]);
    change_first(nums, 57);
    printf("after: %d\n", nums[0]);
    return 0;
}
```

nums is the *address* of the first element.
This address is *copied* into **arr**.
The *elements* of the array are **NOT** copied!

```c
#include <stdio.h>

void change_first(int arr[], int val)
{
    arr[0] = val;
}

int main (void)
{
    int nums[5] = {1, 2, 3, 4, 5};
    printf("before: %d\n", nums[0]);
    change_first(nums, 57);
    printf("after: %d\n", nums[0]);
    return 0;
}
```



<u>Write a user-defined void function that does the following:</u>

Print all the elements in an array.

```c
#include <stdio.h>

void print_array(int arr[], int size)
{
    int i;
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
}

int main (void)
{
    int nums[5] = {1, 2, 3, 4, 5};
    print_array(nums, 5);
    return 0;
}
```

Why do we pass the size of the array into the function?

If we change the size of the array, only the main() function has to be modified.

This way, our program stays modular!

<u>Write a user-defined function that does the following:</u>

Finds the largest element in an array and returns it.

```c
#include <stdio.h>

int find(int arr[], int size)
{
    int i, largest = arr[0];

    for (i = 1; i < size; i++)
    {
        if (arr[i] > largest)
            largest = arr[i];
    }

    return(largest);
}

int main (void)
{
    int nums[6] = {1, -2, 0, 4, -9, 3};
    printf("largest: %d", find(nums, 6));
    return (0);
}
```

Initialize largest to the first element

Compare each element to largest. If arr[i] is bigger than largest, set largest to arr[i]

Return largest

Array Length with `sizeof()`

Returns the size, in bytes, of a given data type.

Quincy 2005 - [Text1]

File Edit View Project Debug Tools Window Help

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

int main ()
{
    printf("%d\n", sizeof(char));
    printf("%d\n", sizeof(int));
    printf("%d\n", sizeof(float));
    printf("%d\n", sizeof(double));

    return 0;
}
```

Press F1 for help Ln 9, Col 35

Quincy 2005

```
1
4
4
8

Press Enter to return
to Quincy...
```

Array Length with `sizeof()`

Turns out it works on arrays, too!

Quincy 2005 - [main.c]

File Edit View Project Debug Tools Window Help

```c
#include <stdio.h>

int main(void)
{
    int arr[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    int sizeBytes = sizeof(arr);
    int nElements = sizeBytes/sizeof(int);

    printf("Size of array in bytes: %d\n", sizeBytes);
    printf("Number of elements: %d\n", nElements);

    return 0;
}
```

Press F1 for help

Quincy 2005

```
Size of array in bytes: 32
Number of elements: 8

Press Enter to return to Quincy...
```

HOWEVER:

- Does **NOT** work when the array is passed into a function!
- Only works in the scope the array was declared.
- Still have to pass array size when using a function.

<u>Write a user-defined function that does the following:</u>

Takes an array and creates a second array whose elements are each three times bigger than the corresponding elements in the original array. Return the tripled array.

Returning (Static) Arrays

Can’t do it.

Static arrays can never be returned by a function.

```
int arr[5] = {1, 2, 3, 4, 5};
return(arr);
```

Why?

Returning (Static) Arrays

```
int* getArray()
{
    int arr[5] = {1, 2, 3, 4, 5};
    return(arr);
}
```

Build

```
"D:\Programs\quincy\MinGW\bin\gcc.exe" -std=iso9899:199409 -pedantic-errors -W -no-long-long
g:\quincy\code\callocsquare.c: In function 'getArray':
g:\quincy\code\callocsquare.c:7: warning: function returns address of local variable
"D:\Programs\quincy\MinGW\bin\gcc.exe" -mconsole -o G:\quincy\code\callocsquare.exe G:\
Successful build
```

Returning (Static) Arrays

```
int* getArray()
{
    int arr[5] = {1, 2, 3, 4, 5};
    return(arr);
}
```

We can only return <u>one</u> thing. We ***cannot*** return the whole array.

When a function ends, local variables are lost.

We can return the ***address*** of **`arr`**, but the elements themselves are lost

Returning (Static) Arrays

```
int* getArray()
{
    int arr[5] = {1, 2, 3, 4, 5};
    return(arr);
}
```

Instead, we pass the array to be "*returned*" into the function as an argument.

<u>Write a user-defined function that does the following:</u>

Takes an array and creates a second array whose elements are each three times bigger than the corresponding elements in the original array. Return the tripled array.

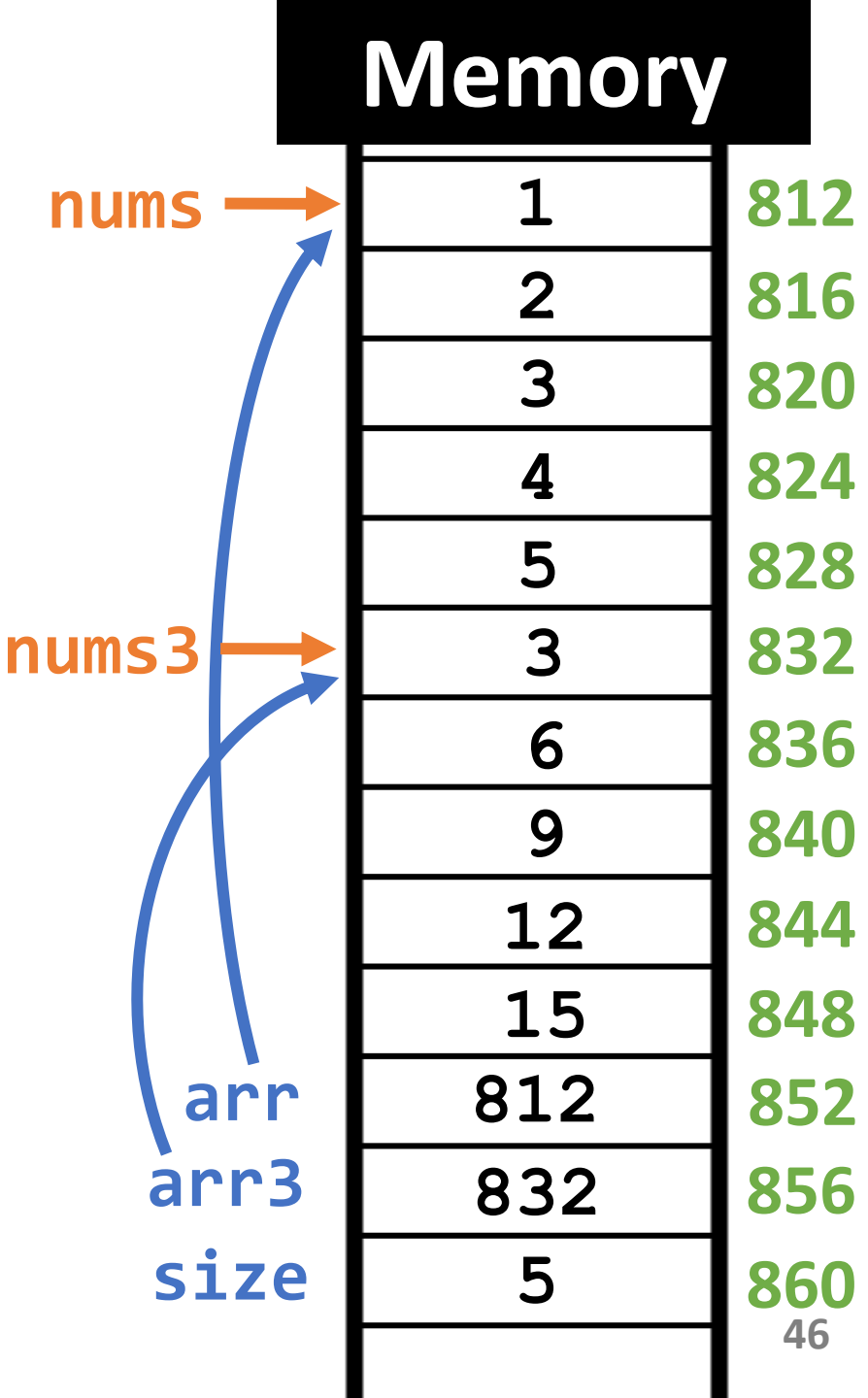
<u>Problem:</u> We can't use a `return` statement to return a static array

```c
#include <stdio.h>

void triple(int arr[], int arr3[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        arr3[i] = arr[i]*3;
}

int main (void)
{
    int i, nums3[5], nums[5] = {1, 2, 3, 4, 5};
    triple(nums, nums3, 5);
    printf("Original:\n");
    for (i = 0; i < size; i++)
        printf("%2d ", nums[i]);
    printf("Tripled:\n");
    for (i = 0; i < size; i++)
        printf("%2d ", nums3[i]);
    return (0);
}
```

`arr` is the address of the first element of `nums`.
`arr3` is the address of the first element of `nums3`.

`nums` and `nums3` are declared in **`main()`**
Their addresses get passed into **`triple()`**

```c
#include <stdio.h>

void triple(int arr[], int arr3[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        arr3[i] = arr[i]*3;
}

int main (void)
{
    int i, nums3[5], nums[5] = {1, 2, 3, 4, 5};
    triple(nums, nums3, 5);

    printf("Original:\n");
    for (i = 0; i < size; i++)
        printf("%2d ", nums[i]);
    printf("Tripled:\n");
    for (i = 0; i < size; i++)
        printf("%2d ", nums3[i]);
    return (0);
}
```



Quincy 2005 - [arrayTriple]

File Edit View Project Debug Tools Window Help

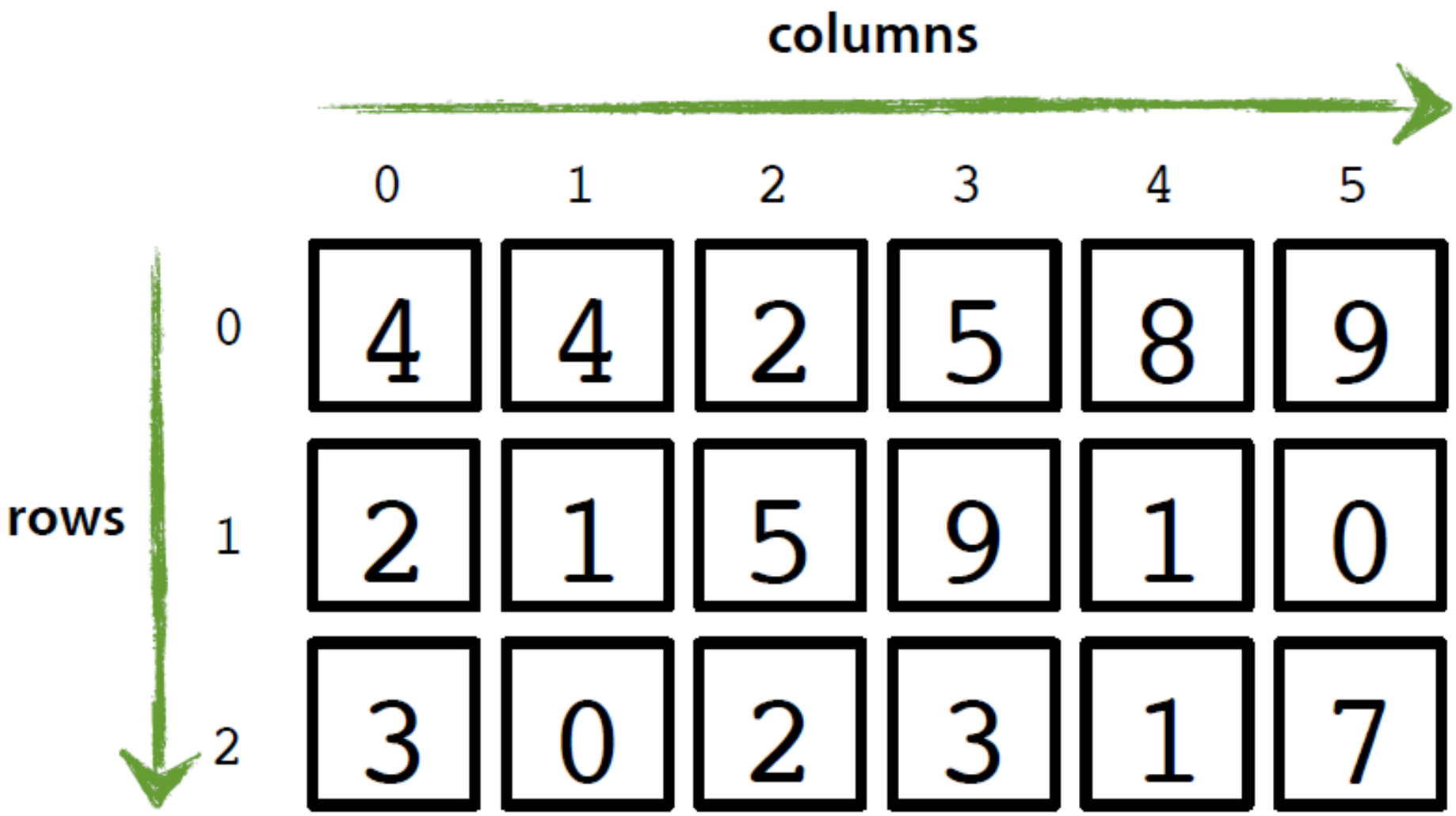
```c
#include <stdio.h>

void triple(int arr[], int arr3[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        arr3[i] = arr[i]*3;
}

int main()
{
    int i, nums3[5], nums[5] =
   triple(nums, nums3, 5);

   printf("Original:\n");
   for (i = 0; i < 5; i++)
      printf("%2d ", nums[i]);
   printf("\nTripled:\n");
   for (i = 0; i < 5; i++)
      printf("%2d ", nums3[i]);
   return (0);
}
```

Press F1 for help Ln 22, Col 2 NUM

Quincy 2005

```
Original:
 1  2  3  4  5
Tripled:
 3  6  9 12 15
Press Enter to return to Quincy...
```

2D Arrays


columns
0
1
2
3
4
5
rows
0
1
2
4
4
2
5
8
9
2
1
5
9
1
0
3
0
2
3
1
7

Recall: 1D Static Arrays

```
#include <stdio.h>

int main()
{
    int nums[100];
    return 0;
}
```

An array of 100 integers

2D Static Arrays

We simply add another set of square brackets [ ]

```
#include <stdio.h>

int main()
{
    int nums[10][10];
    return 0;
}
```

A 2D array, 10 by 10

ND Static Arrays

As many dimensions as you want:

```
#include <stdio.h>

int main()
{
    int nums[5][5][5][5][5];

    return 0;
}
```

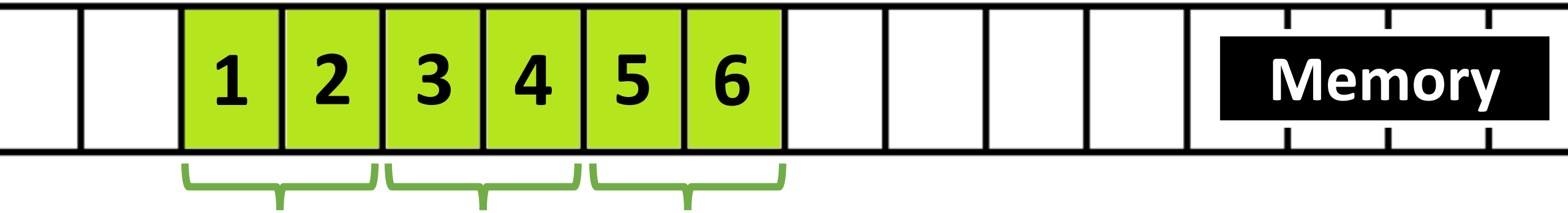
Declare and initialize:

```
#include <stdio.h>

int main()
{
        int nums[3][2] = {{1, 2}, {3, 4}, {5, 6}};
        return 0;
}
```

3 rows

2 columns

Row 1 Row 2 Row 3



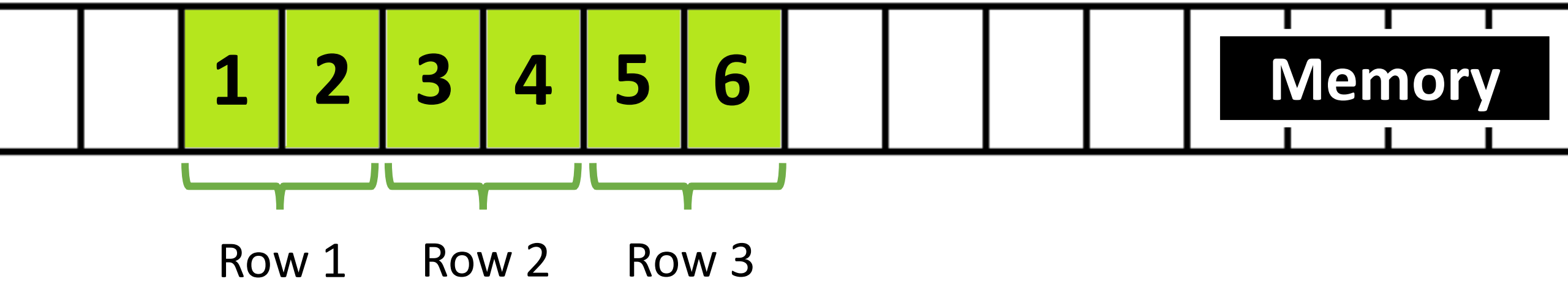
Row 1 Row 2 Row 3

2D Memory

Memory is ***NOT*** 2D!

Addressing is ***one dimensional***!

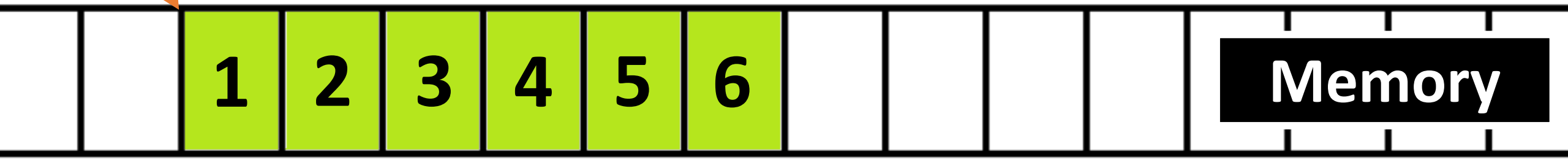
Static 2D arrays are still allocated as contiguous 1D chunks.



Declare and initialize:

```
#include <stdio.h>

int main()
{
    int nums[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    return 0;
}
```

The name of a static, 2D array is <u>STILL</u> just a pointer to the first element!




```
nums == &nums[0][0]
```

Accessing Elements

```
#include <stdio.h>

int main()
{
    int matrix[2][2];
```

2 rows, 2 columns

Row index

Column index

```
    matrix[0][0] = 17; /* row 0, col 0 */
    matrix[0][1] = -3; /* row 0, col 1 */
    matrix[1][0] = 0;  /* row 1, col 0 */
    matrix[1][1] = 57; /* row 1, col 1 */

    return 0;
}
```

Accessing Elements

```
#include <stdio.h>

int main()
{
    int matrix[2][2];

    scanf("%d", &matrix[0][0]);
    scanf("%d", &matrix[0][1]);
    scanf("%d", &matrix[1][0]);
    scanf("%d", &matrix[1][1]);

    return 0;
}
```

Just like 1D arrays!

2D Loop + 2D Array

```
#include <stdio.h>

int main()
{
    int matrix[][] = {{1,2,3},{4,5,6},{7,8,9}};
    int row, col;

    for (row = 0; row < 3; row++) {
        for (col = 0; col < 3; col++) {
            printf("%5d", matrix[row][col]);
        }
        printf("\n");
    }

    return 0;
}
```

2D Static Arrays as Arguments

Create a function for matrix addition. It will take in three arguments. The two matrices to be added, and a third matrix to store the result.

```c
#define M 3 /* number of matrix rows */
#define N 3 /* number of matrix columns */

void addMatrix(int a[M][N], int b[M][N], int c[M][N])
{
    int row, col;

    for (row = 0; row < M; row++) {
        for (col = 0; col < N; col++) {
            c[row][col] = a[row][col] + b[row][col];
        }
    }
}
```

```c
#define M 3 /* number of matrix rows */
#define N 3 /* number of matrix columns */

void addMatrix(int a[M][N], int b[M][N], int c[M][N])
{ /* addMatrix code from previous slide here */ }

int main()
{
    int mat1[M][N] = {{1,2,2},{2,-3,6},{1,0,-3}};
    int mat2[M][N] = {{1,1,1},{1,6,2},{1,-1,-1}};
    int result[M][N];

    addMatrix(mat1, mat2, result);

    return 0;
}
```

```c
#define M 3 /* number of matrix rows */
#define N 3 /* number of matrix columns */

void addMatrix(int a[M][N], int b[M][N], int c[M][N])
{
   /* addMatrix code here */
}
```

Notice: We are specifying the array size!

At minimum, we must specify the size of each row (number of columns)

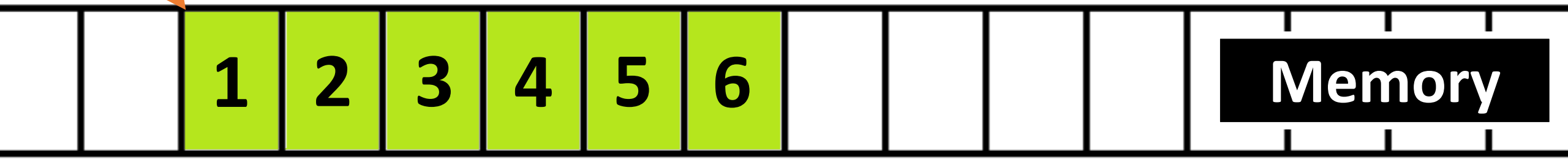
```c
#define M 3 /* number of matrix rows */
#define N 3 /* number of matrix columns */

void addMatrix(int a[][N], int b[][N], int c[][N])
{
   /* addMatrix code here */
}
```

<u>Notice:</u> We are specifying the array size!

At minimum, we must specify the size of each row (number of columns). **Why?**

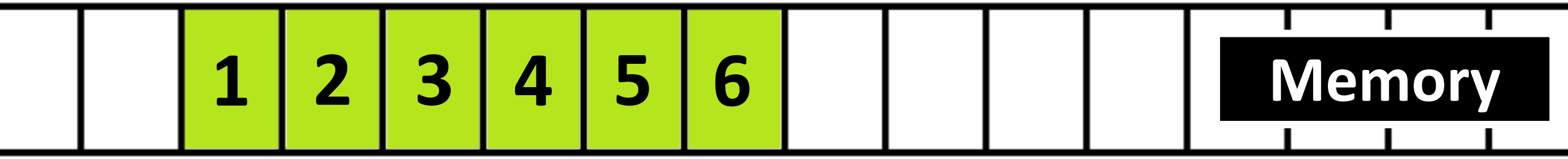
```c
#include <stdio.h>

int main()
{
    int nums[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    printf("%d", nums[1][1]);
    return 0;
}
```

If **nums** is just a pointer to the first element, how do we know where the 2nd row begins?



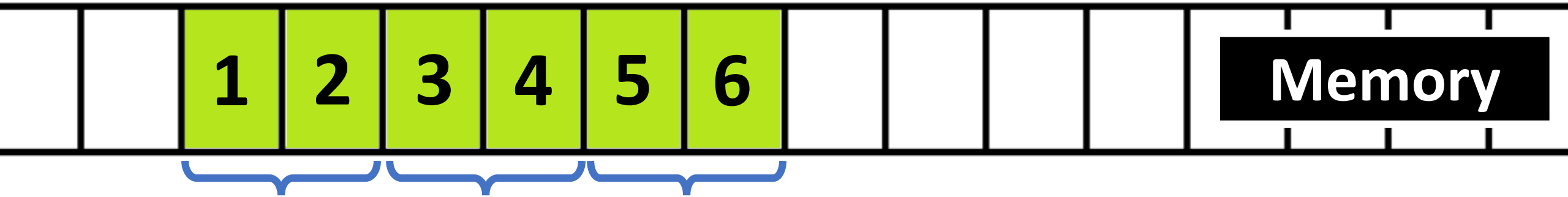
```
#include <stdio.h>

int main()
{
    int nums[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    printf("%d", nums[1][1]);
    return 0;
}
```

The compiler knows that the 2nd row starts at index 2 because it knows how many columns there are.



```c
#include <stdio.h>

int main()
{
    int nums[3][2] = {{1, 2}, {3, 4}, {5, 6}};
    printf("%d", nums[1][1]);
    return 0;
}
```

If the compiler doesn't know the number of columns, it has no idea where one row ends and another begins. 2D indexing would not be possible.



```
void addMatrix(int a[][N], int b[][N], int c[][N])
{
    int row, col;

    for (row = 0; row < M; row++)
        for (col = 0; col < N; col++)
            c[row][col] = a[row][col] + b[row][col];
}
```

Therefore, this is sufficient, but omitting the number of rows and columns entirely is not.

Enums

Oddball topic, tangential to arrays

Enums

Enumerations allow us to define a custom type AND the values that type can take:

```
#include <stdio.h>
enum boolean {FALSE, TRUE};
int main()
{
    enum boolean f, t;
    f = FALSE;
    t = TRUE;
    printf("%d %d\n", f, t);
    return 0;
}
```

- The enum values (FALSE, TRUE) alias their numeric index in the enum
- FALSE == 0, TRUE == 1

Enums

```
enum day {Sun, Mon, Tue, Wed, Thu, Fri, Sat};

for (int i = Sun; i <= Sat; i++)
   printf("Day %d\n", i);
```

What prints? `Day 0, Day 1, …, Day 5, Day 6`

Numbering starts at 0 by default.

Enums

If we specify an integer for the first element, counting will start from there:

```
enum day {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};
```

```
for (int i = Sun; i <= Sat; i++)
   printf("Day %d\n", i);
```

What prints? `Day 1, Day 2, …, Day 6, Day 7`

enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany

File Edit Search View Document Project Build Tools Help

Symbols

- Functions
  - main [5]
- Typedefs / Enums
  - day [3]
    - Fri [3]
    - Mon [3]
    - Sat [3]
    - Sun [3]
    - Thu [3]
    - Tue [3]
    - Wed [3]

enum.c

```c
#include <stdio.h>

enum day {Sun = 1, Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);

    return 0;
}
```

Compiler

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code
Compilation finished successfully.
```

line: 12 / 14 col: 0 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: unknown

C:\WINDOWS\SYSTEM32\cmd.exe

```
Day 1
Day 2
Day 3
Day 4
Day 5
Day 6
Day 7


------------------
(program exited with code: 0)

Press any key to continue . . .
```

enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany

File Edit Search View Document Project Build Tools Help

Symbols

- Functions
  - main [5]
- Typedefs / Enums
  - day [3]
    - Fri [3]
    - Mon [3]
    - Sat [3]
    - Sun [3]
    - Thu [3]
    - Tue [3]
    - Wed [3]

enum.c

```c
#include <stdio.h>

enum day {Sun = 1, Mon, Tue, Wed = 12, Th

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);

```

What about this?
What prints?

Numbering continues from the last assigned value

Compiler

```
gcc -Wall
Compilation finished successfully.
```

line: 3 / 14 col: 37 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: unknown

C:\WINDOWS\SYSTEM32\cmd.exe

```
Day 1
Day 2
Day 3
Day 4
Day 5
Day 6
Day 7
Day 8
Day 9
Day 10
Day 11
Day 12
Day 13
Day 14
Day 15

------------------
(program exited with code: 0)

Press any key to continue . . .
```

Sun, Mon, Tue

Wed, Thu, Fri, Sat

enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany

File Edit Search View Document Project Build Tools Help

Symbols

- Functions
  - main [5]
- Typedefs / Enums
  - day [3]
    - Fri [3]
    - Mon [3]
    - Sat [3]
    - Sun [3]
    - Thu [3]
    - Tue [3]
    - Wed [3]

enum.c

```c
#include <stdio.h>

enum day {Sun = 1.6, Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %d\n", i);

    return 0;
}
```

Non-integer values?

Compiler

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples)
enum.c:3:17: error: enumerator value for 'Sun' is not an integer constant
    3 | enum day {Sun = 1.6, Mon, Tue, Wed, Thu, Fri, Sat};
      |                 ^~~
Compilation failed.
```

Nope.

line: 13 / 14 col: 0 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: unknown

enum.c - C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samples - Geany

File Edit Search View Document Project Build Tools Help

Symbols

- Functions
  - main [5]
- Typedefs / Enums
  - day [3]
    - Fri [3]
    - Mon [3]
    - Sat [3]
    - Sun [3]
    - Thu [3]
    - Tue [3]
    - Wed [3]

enum.c

```c
#include <stdio.h>

enum day {Sun = 'a', Mon, Tue, Wed, Thu, Fri, Sat};

int main()
{
    for (int i = Sun; i <= Sat; i++)
        printf("Day %c\n", i);

    return 0;
}
```

Characters work fine, they're integers behind the scenes

Compiler

```
gcc -Wall -o "enum" "enum.c" (in directory: C:\Users\aufke\Google Drive\Teaching\CPS 188\Code Samp
Compilation finished successfully.
```

line: 8 / 14 col: 22 sel: 0 INS TAB mode: CRLF encoding: UTF-8 filetype: C scope: main

C:\WINDOWS\SYSTEM32\cmd.exe

```
Day a
Day b
Day c
Day d
Day e
Day f
Day g


------------------
(program exited with code: 0)

Press any key to continue . . .
```

Questions?