THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

CS330

C: Arrays and Strings and malloc()

Spring 2022

Lab 6

Today's Agenda

- Arrays
- Strings
- Malloc()

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

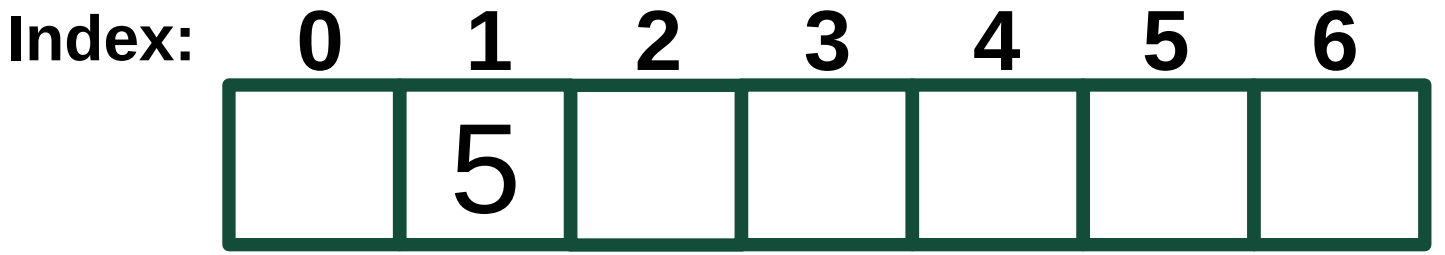
Arrays

Arrays – standard approach

- **One-Dimensional Arrays**
  - An array is a data structure that contains a number of values, of the same type, stored in elements.
  - Each element can be accessed by its position within the array: index
    - C array indexes begin at 0 (just like Python, Java)
  - Always declare the array before use
    <type> <name>[<number_of_elements>];

    ```
    int sampleArray[100];
    float anotherArray[250];
    ```
  - Don't use magic / mystery numbers. Better to define the size like:

    ```
    #define ARRAY_SIZE = 100;
    int sampleArray[ARRAY_SIZE];
    ```



Arrays – standard approach (cont'd)

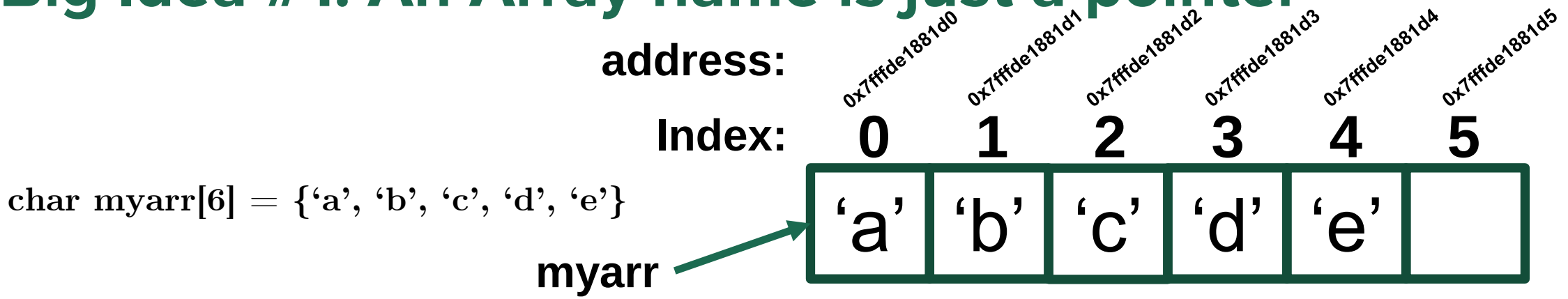
- **Can initialize at the same time as the array declaration:**
  `int arr[10] = {1, 2, 3, 4};`
- **If the array is initialized, the size is not required:**
  `int arr[] = {1, 2, 3, 4};`
- **Can assign or retrieve values using the array index:**
  ```
  printf("%d", arr[1]);  // prints 2
  arr[1] = 5;  // changes 2 to 5
  ```

Arrays – looping through Arrays in C

- There is nothing to indicate the end of an array
  - And it's very easy to iterate past the end of the array
- It's helpful to know the size: sizeof()
- Syntax is:
  - Size of the array (in bytes):
    $\text{sizeof}(\text{arr})$
  - Capacity of the array (number of elements):
    $\text{sizeof}(\text{arr}) \ / \ \text{sizeof}(\text{arr}[0])$

```
13      int arr[10] = {5,8,9,12};
14      printf("\n Size of the array :%lu", sizeof(arr));                40
15      printf("\n Capacity the array :%lu", sizeof(arr)/sizeof(arr[0])); 10
16
17      int arr2[] = {5,8,9,12};
18      printf("\n Size of the array2 :%lu", sizeof(arr2));               16
19      printf("\n Capacity the array2 :%lu", sizeof(arr2)/sizeof(arr2[0]));4
20      return 0;
21  }
```

Arrays in C: The Use of Pointers
Big Idea #1: An Array name is just a pointer



- Since every byte in memory has an address (See Pointers, Big Idea #1) and since we can know the address (See Pointers, Big Idea #2) the array variable is essentially a pointer to the array base address
- Furthermore, in languages like Python and Java you can return an array from a function, but in C you get a pointer instead
- Therefore, you can get values from the array in either of these two ways:

standard approach: $\mathrm{myarr[i]}$

via the pointer: $*(\mathrm{myarr} + \mathrm{i})$

Big Idea #2: Can access an Array element via its pointer

We're going to focus on the second of those two methods:

$$*(\text{myarr} + \text{i})$$

We know that * is essentially fetching the value at a pointer, and we know that myarr is a pointer to the start of an array.

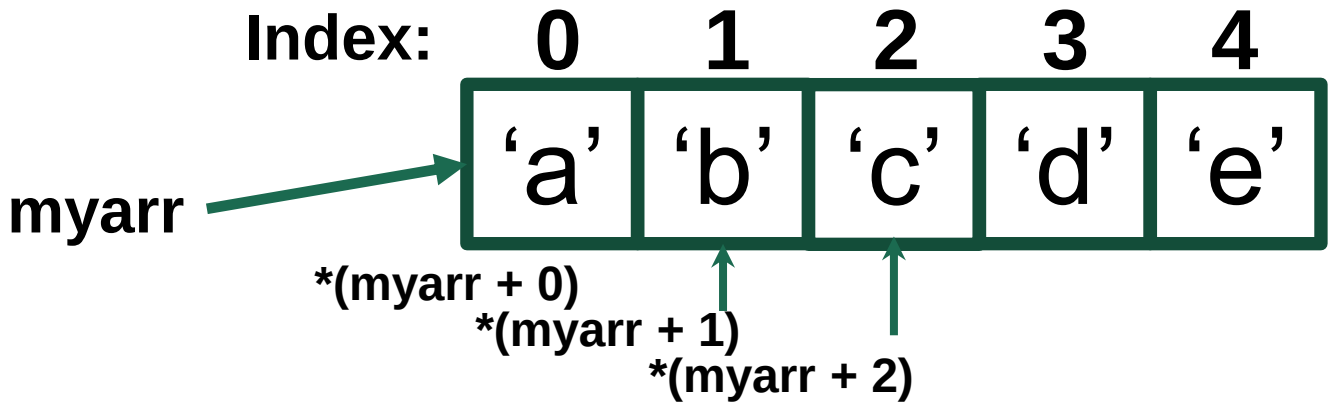
We also know that the i stands for an index.

But being able to add i to a pointer is one of the most important and powerful capabilities of pointers: **Pointer Arithmetic**.

***myarr == *(myarr + 0) == 'a'**

***(myarr + 1) == 'b'**

***(myarr + 2) == 'c'**



Pointer Arithmetic

By summing a pointer with an integer, we can create a pointer to another location in memory.

This isn't useful for data types like int, but for arrays, it's vitally important.

Pointers are how you index an array, if we couldn't use bracket notation.

In assembly, you cannot index an array with brackets. But you can fetch data from a pointer, and you can sum to pointers.

So long as you know the length of the array you're trying to index, you can index it like so: $*(\mathrm{myarr} + \mathrm{i})$

HW4 NOTE

- When accessing an index of an array:

We should only see: $*(\mathrm{myarr} + \mathrm{i})$

We should not see: $\mathrm{myarr}[\mathrm{i}]$

Exam

lab05 > C array.c > main()

```c
#include<stdio.h>

int* fillArray(int exampleArray[]){  // could also be: (int *exampleArray)
    int i;
    for(i = 0; i < 10; i++){
        exampleArray[i] = i+1;
        printf("inside function, exampleArray[%d] == %d\n", i, exampleArray[i]);
    }
    return exampleArray; // we return an 'array', but it's just a pointer
}

int main(){
    int exArray[10];
    int *ptr;
    ptr = fillArray(exArray);
    int i;
    // WARNING: C will not stop you from indexing outside the array
    for(i=0; i < 10; i++){
        // in line below, we use pointer arithmatic to get value at index i
        printf("in main, *(ptr + %d) == %d\n", i, *(ptr + i));
    }

    printf("exArray has %ld elements\n", (sizeof(exArray)/sizeof(exArray[0])));

    return 0;
}
```

Modified from: https://pastebin.com/pvyAAJuf

lab05 > C array.c > main()

```c
#include<stdio.h>

int* fillArray(int exampleArray[]){  // could also be: (int *exampleArray)
    int i;
    for(i = 0; i < 10; i++){
        exampleArray[i] = i+1;
        printf("inside function, exampleArray[%d] == %d\n", i, exampleArray[i]);
    }
    return exampleArray; // we return an 'array', but it's just a pointer
}

int main(){
    int exArray[10];
    int *ptr;
    ptr = fillArray(exArray);
    int i;
    // WARNING: C will not stop you from indexing outside the array
    for(i=0; i < 10; i++){
        // in line below, we use pointer arithmatic to get value at index i
        printf("in main, *(ptr + %d) == %d\n", i, *(ptr + i));
    }

    printf("exArray has %ld elements\n", (sizeof(exArray)/sizeof(exArray[0])));

    return 0;
}
```

```
inside function, exampleArray[0] == 1
inside function, exampleArray[1] == 2
inside function, exampleArray[2] == 3
inside function, exampleArray[3] == 4
inside function, exampleArray[4] == 5
inside function, exampleArray[5] == 6
inside function, exampleArray[6] == 7
inside function, exampleArray[7] == 8
inside function, exampleArray[8] == 9
inside function, exampleArray[9] == 10
in main, *(ptr + 0) == 1
in main, *(ptr + 1) == 2
in main, *(ptr + 2) == 3
in main, *(ptr + 3) == 4
in main, *(ptr + 4) == 5
in main, *(ptr + 5) == 6
in main, *(ptr + 6) == 7
in main, *(ptr + 7) == 8
in main, *(ptr + 8) == 9
in main, *(ptr + 9) == 10
exArray has 10 elements
```

lab05 > C array.c > main()

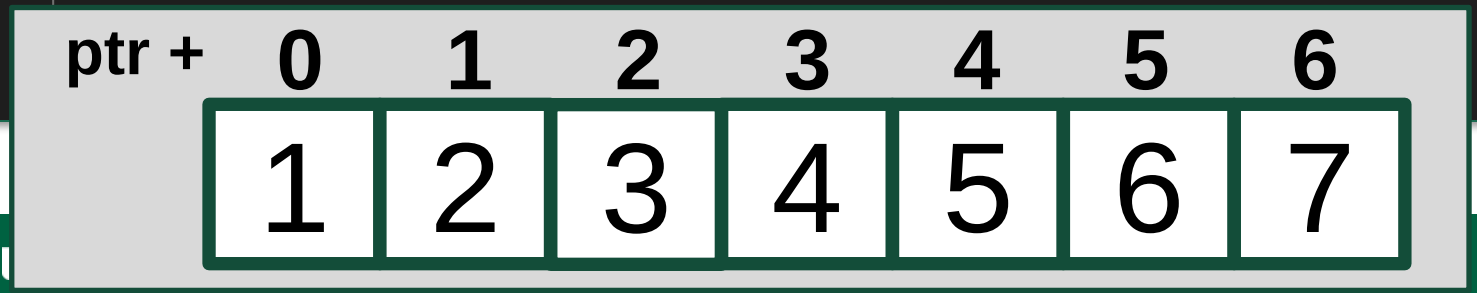
```c
#include<stdio.h>

int* fillArray(int exampleArray[]){  // could also be: (int *exampleArray)
    int i;
    for(i = 0; i < 10; i++){
        exampleArray[i] = i+1;
        printf("inside function, exampleArray[%d] == %d\n", i, exampleArray[i]);
    }
    return exampleArray; // we return an 'array', but it's just a pointer
}

int main(){
    int exArray[10];
    int *ptr;
    ptr = fillArray(exArray);
    int i;
    // WARNING: C will not stop you from indexing outside the array
    for(i=0; i < 10; i++){
        // in line below, we use pointer arithmatic to get value at index i
        printf("in main, *(ptr + %d) == %d\n", i, *(ptr + i));
    }

    printf("exArray has %ld elements\n", (sizeof(exArray)/sizeof(exArray[0])));
```

```
inside function, exampleArray[0] == 1
inside function, exampleArray[1] == 2
inside function, exampleArray[2] == 3
inside function, exampleArray[3] == 4
inside function, exampleArray[4] == 5
inside function, exampleArray[5] == 6
inside function, exampleArray[6] == 7
inside function, exampleArray[7] == 8
inside function, exampleArray[8] == 9
inside function, exampleArray[9] == 10
in main, *(ptr + 0) == 1
in main, *(ptr + 1) == 2
in main, *(ptr + 2) == 3
in main, *(ptr + 3) == 4
in main, *(ptr + 4) == 5
in main, *(ptr + 5) == 6
in main, *(ptr + 6) == 7
in main, *(ptr + 7) == 8
in main, *(ptr + 8) == 9
in main, *(ptr + 9) == 10
exArray has 10 elements
```



THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Strings

Strings and Arrays

- **Strings are char arrays in C**, not a type of their own. This means they're subject to all the same idiosyncrasies of arrays. They can still be indexed, but there's an extremely important reason to not do so:
- Strings in C end with a null terminator: $'\backslash 0'$
  This is a character that is used to know when an array of characters ends.
- However, if we index one character, we don't have the null terminator on that new one-char string.
- Code example following simply demonstrates that strings are arrays, and how to print one character of a string.

```
char str[8];
```

An array can be initialized with a string, when it is declared. For example, with the declaration:

```
char str[8] = "message";
```

```
char *str = “message”;
```

the compiler copies the characters of the "message" into the str array and adds the null character. In particular, str[0] becomes 'm', str[1] becomes 'e', and the value of the last element str[7] becomes '\0'. In fact, this declaration is equivalent to:

```
char str[8] = {'m', 'e', 's', 's',
'a', 'g', 'e', '\0'};
```

String Example

lab05 > C string_ex.c > main()

```c
#include<stdio.h>

int main(){
    char myString[5] = "test";
    // size should be num char + 1 to hold null character '\0' at the end
    printf("%s \n", myString);
    printf("%c \n", myString[3]);
    // printf("%s \n", myString[3]);  // error since this is just one char
    // and C expects a null character at the end
    return 0;
}
```

```
test
t
```


Modified from: https://pastebin.com/r1GJFLU9

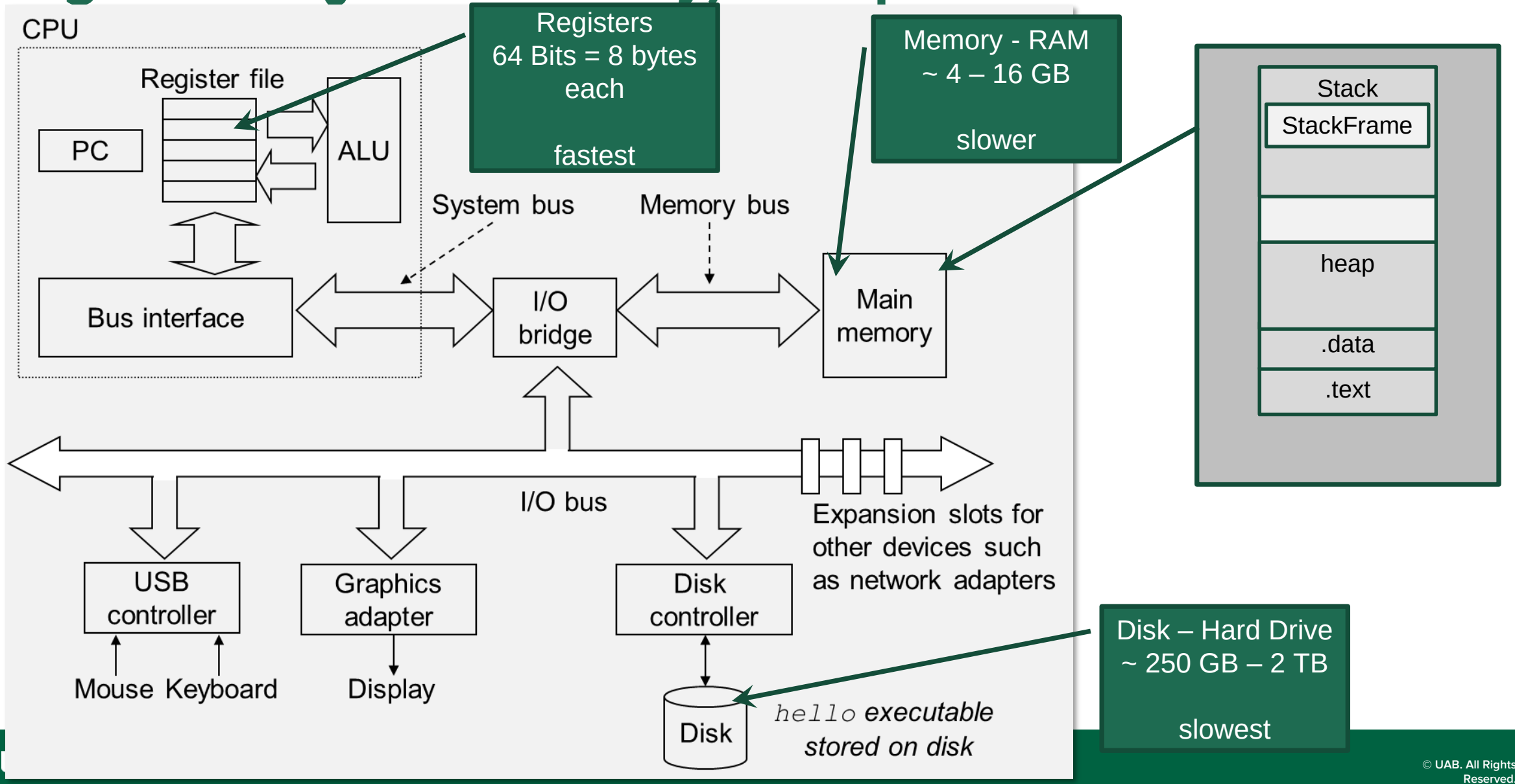
Strings

- See also string.h, lots of useful functions:
  - strlen()
  - strcpy()
  - strcat()
  - strcmp()
  - strtok()
  - Just be careful, some strip off the null character
- A string literal is a sequence of characters enclosed in double quotes.
- C treats it as a nameless character array.
- To store a string in a variable, we use an array of characters.
- Because of the C convention that a string ends with the null character, to store a string of **N** characters, the size of the array should be **N+1** at least.

THE UNIVERSITY OF ALABAMA AT BIRMINGHAM

Malloc()

High Level (just for today) Computer Architecture



Malloc()


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

- The size_t type is usually a synonym of the unsigned int type (remember typedef)
- The size parameter declares the number of **bytes** to be allocated
- If the memory is allocated successfully:
  malloc() returns a pointer to that memory
  NULL otherwise
- We must free the memory when we're done, using free()
- Allocates memory on the heap, widely accessible within our program
- Flexible memory whose size can be reallocated later as needed
  - Dynamic Memory Allocation

See also

- void *realloc(void *ptr, size_t size)
  - to change (reallocate) the memory allocation (e.g. you need more)
- void *calloc(size_t numElements, size_t sizeOfEachElement)
  - To allocate an array initialized to 0s
- void free(void *ptr)
- memcpy()
- memmove()
- memcmp()

lab05 > C array_malloc.c > main()

```c
#include<stdio.h>
#include<stdlib.h> // for malloc

int* fillArray(){
    int numElements = 10;
    int *ptrToArray;
    ptrToArray = malloc(numElements * sizeof(int));
    //ptrToArray = calloc(numElements, sizeof(int));
    int i;
    for(i = 0; i < 10; i++){
        ptrToArray[i] = i+1;
        printf("inside function, exampleArray[%d] == %d\n", i, *(ptrToArray + i));
        // can also use *(ptrToArray + i)  or ptrToArray[i]
    }
    return ptrToArray; // we return an 'array', but it's just a pointer
}

int main(){
    //int exArray[10];
    int *ptr;
    //ptr = fillArray(exArray);
    ptr = fillArray();
    int i;
    // WARNING: C will not stop you from indexing outside the array
    for(i=0; i < 10; i++){
        // in line below, we use pointer arithmatic to get value at index i
        printf("in main, *(ptr + %d) == %d\n", i, *(ptr + i));
    }

    printf("exArray has %ld elements\n", (sizeof(ptr)/sizeof(int)));

    return 0;
}
```

```
inside function, exampleArray[0] == 1
inside function, exampleArray[1] == 2
inside function, exampleArray[2] == 3
inside function, exampleArray[3] == 4
inside function, exampleArray[4] == 5
inside function, exampleArray[5] == 6
inside function, exampleArray[6] == 7
inside function, exampleArray[7] == 8
inside function, exampleArray[8] == 9
inside function, exampleArray[9] == 10
in main, *(ptr + 0) == 1
in main, *(ptr + 1) == 2
in main, *(ptr + 2) == 3
in main, *(ptr + 3) == 4
in main, *(ptr + 4) == 5
in main, *(ptr + 5) == 6
in main, *(ptr + 6) == 7
in main, *(ptr + 7) == 8
in main, *(ptr + 8) == 9
in main, *(ptr + 9) == 10
exArray has 10 elements
```

Returning Arrays, Strings, Pointers from functions

a) Declare Array as Global Variable
   Stored in .data
   Accessible from function(), main(), etc

b) Declare Array in main(), pass into function() via reference
   Stored in main()'s Stack Frame, but accessible via reference from function()'s Stack Frame

c) Declare Array using malloc() inside function() and return a pointer from the function()
   Stored in heap
   Accessible from function(), main(), etc
   Can be resized later via realloc()