Griffin Hurt
Undergraduate Teaching Fellow
griffhurt@pitt.edu
https://griffinhurt.com

Spring 2024, Term 2244
Friday 2 PM Recitation
Feb 2nd, 2024

Slides adapted from
Shinwoo Kim, Martha Dixon, and Vinicius Petrucci

Department of Computer Science
School of Computing & Information
University of Pittsburgh

Recitation 3: Pointers

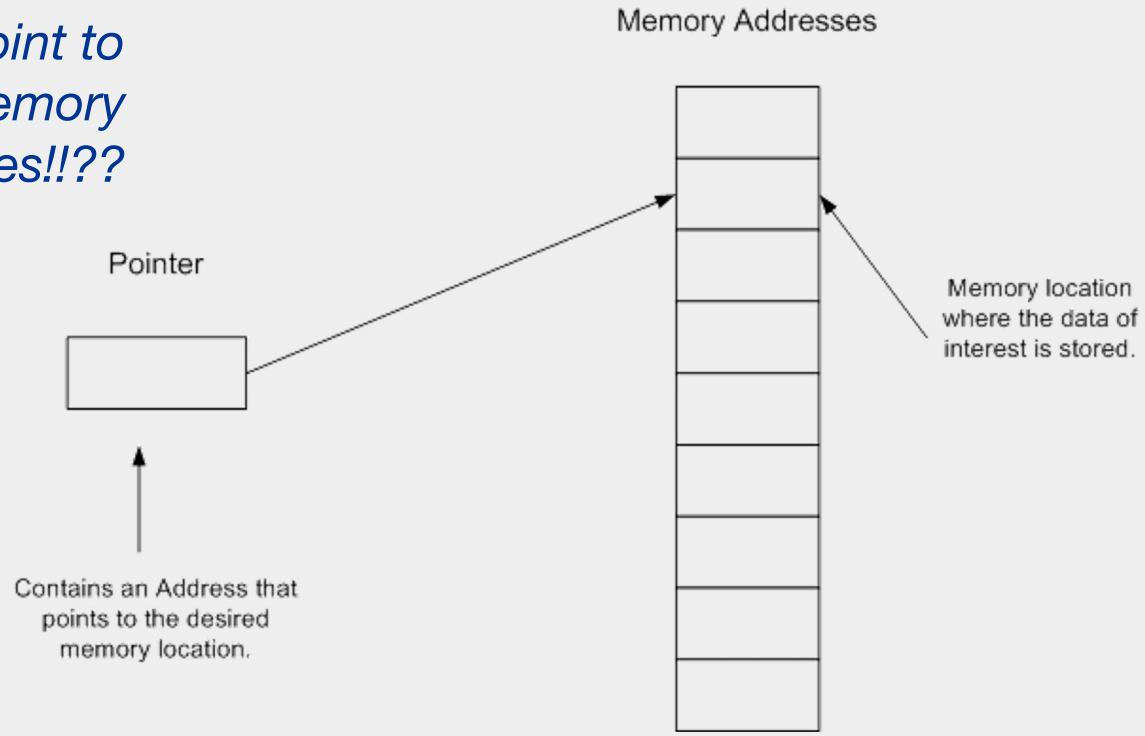
- Course News
- Pointers
- Quiz (for completion)
- Lab 2: Pointer Lab: Looking at Pointers!

Course News

Lab 2 is out, due February 8th 5:59PM

Pointers

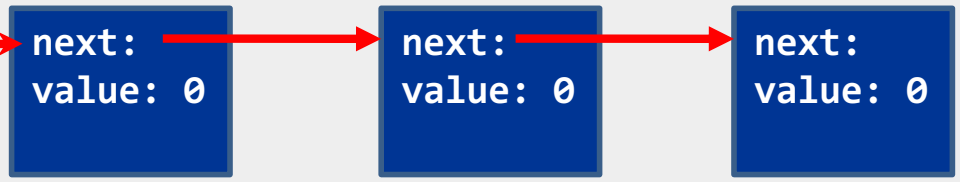
Point to here, point to there, point to that, point to this, and point to nothing! well, they are just memory addresses!!??



You've kinda used pointers in Java...

- **remember writing linked lists?**

```
class Link {
    Link next;
    int value;
}
Link list = new Link();
list.next = new Link();
list.next.next = new Link();
```



what about a reference that doesn't refer to anything?

C has `null` too, but you have to yell it: `NULL`!

A pointer is a variable that contains a memory address

Pointers are variables, so they have a type

- **The type describes what kind of data it points to**
  - An `int` has type `int`
  - A pointer to an `int` has type `int*`
  - A pointer to a pointer to an `int` has type `int**`
- **Expressions also have a type**
  - If `x` has type `int`, then `x+4` also has type `int`
  - If `x` has type `int`, then `&x` has type `int*`
  - If `p` has type `int*`, then `*p` has type `int`
  - If `p` has type `int*`, then `&p` has type `int**`

Pointers are variables, so they store data

- **a variable is a named piece of memory**
- **a pointer is a variable that holds a memory address**

```
int x = 0x100;
int y = 0x200;
int* px = &x;
int* py = &y;
```

Name	Address	Value
x	DC00	0100
y	DC04	0200
px	DC08	DC00
py	DC0C	DC04

since pointers are variables, can you get their addresses?

the addresses of these variables are given to us automatically by the compiler-ish

Declaring pointers

- **in Java, how do you declare an array of any type X?**
  - you put **square brackets** after the type: `X[]`

`int[]`

an **array** that holds **ints.**

`int*`

an **pointer** to an **int.**

`int[][]`

an **array** that holds **arrays,** and each of those holds **ints.**

`int**`

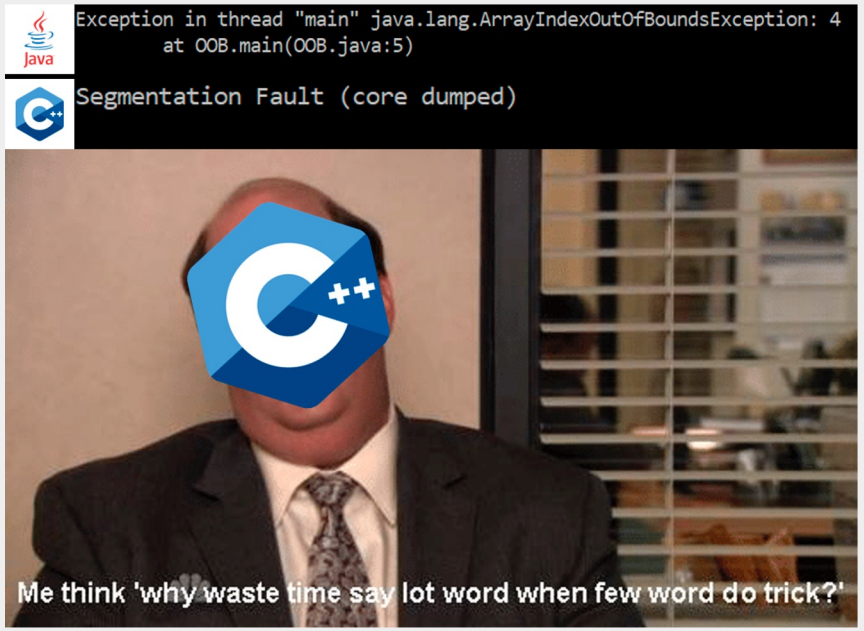
a **pointer** to a **pointer,** which points to an **int.**

a C pointer can point to **either a single value or an array of that type.**

The address-of operator (&)

- **when used as a prefix operator, & means "address of"**
  - it gives you the memory address of any variable, array item, etc.
- **the address is given to you as a pointer type**
  - i.e. it **"adds a star"** I know it seems backwards, why wouldn't they make * add a star, or name pointers int& right?
  - use it on an **int**?
    - you get an **int***
  - use it on an **int***?
    - you get an **int****
  - YOU GET THE IDEA I hope
- **you can use it on just about anything with a name**
  - `&x`
  - `&arr[10]`
  - `&main` (yep!) google function pointers in C!

Accessing the value(s) at a pointer



The value-at (or "dereference") operator

- *** is the value-at operator**
  - it **dereferences** a pointer
  - that is, it **accesses the memory** that a pointer points to
- **it's the inverse of &**
  - every time you use it, you *remove* a star again, this feels backwards?

```
int** ppx = ...
int*  px = *ppx;
int   x = *px;
```

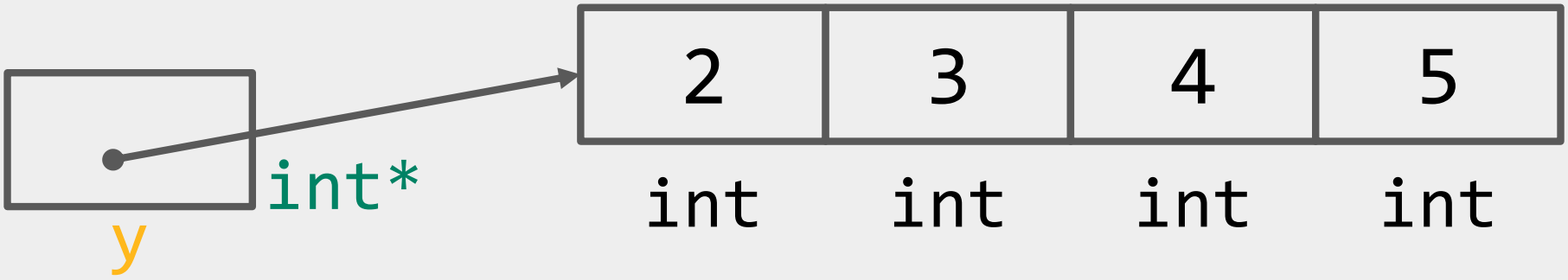
goes to the address that **`ppx`** contains, and gets the **`int*`** there

goes to the address that **`px`** contains, and gets the **`int`** there

Arrays are just pointers *well…sort of*

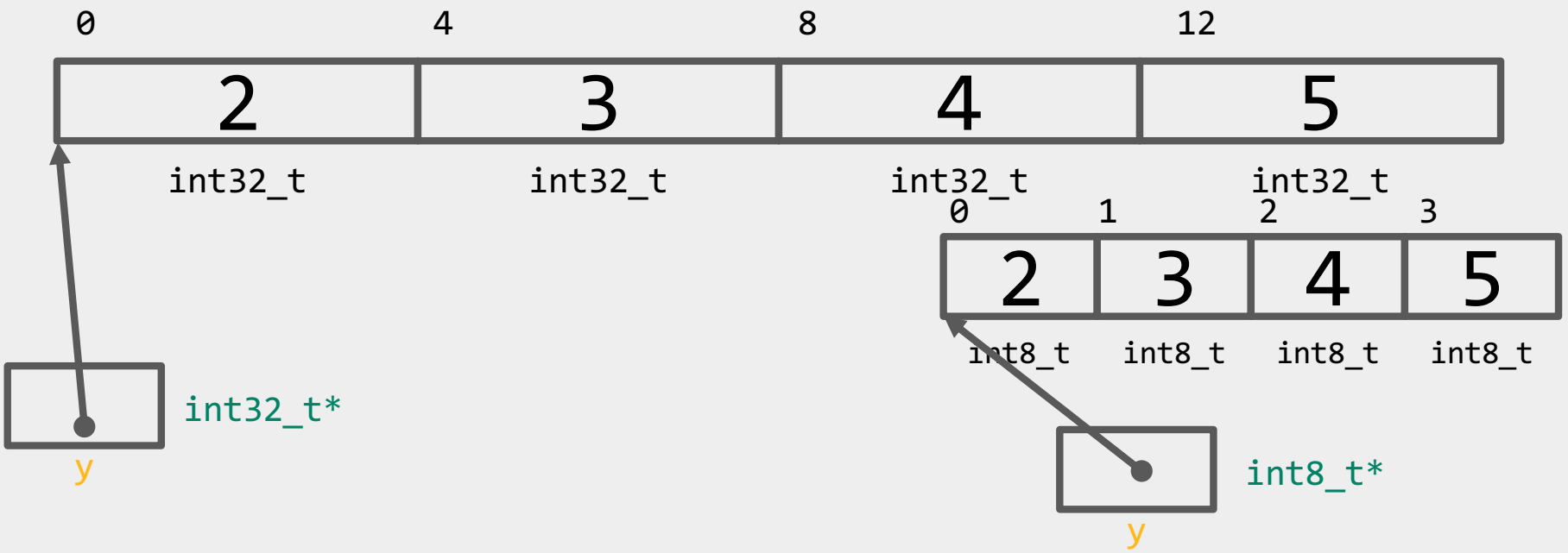
- **In C, array names are just aliases that can be used as pointers**
  - `int y[] = {2, 3, 4, 5}; // these two are`
  - `int *y = {2, 3, 4, 5}; // roughly equivalent`
- **Indexing and dereferencing pointers are equivalent**
  - Side note: you can do math with pointers…this is called **pointer arithmetic.**
  - when you use the array indexing operator, you're really just adding an offset to the pointer, and using that as the address to access.

`*y ≡ y[0]` `*(y+1) ≡ y[1]`



Pointer types are important!

- If `x` is an `int*8_t*`, `x[3]` access elements at byte offset 3 × 1 = 3
- If `x` is an `int*32_t*`, `x[3]` access elements at byte offset 3 × 4 = 12



Pointer arithmetic

- **if we write this:**
  ```
  int array[] = {0, 1, 2, 3};
  ```
- **memory looks like this:**
- **if we want to access array[2]...**
  - what is that equivalent to?
  - ***(array + 2)**
- **but how big is each item in the array? (what is sizeof(int)?)**

Name	Address	Value
array[3]	DC0C	3
array[2]	DC08	2
array[1]	DC04	1
array[0]	DC00	0

- when we write **array + 2**, we don't get **0xDC02,** we get **0xDC08**
- **it adds the size of 2 *items* to the address**
- **when you add or subtract offsets to pointers, C "scales" the offsets by multiples of the size of the type they point to.**

Oh yeah, and that stupid -> operator

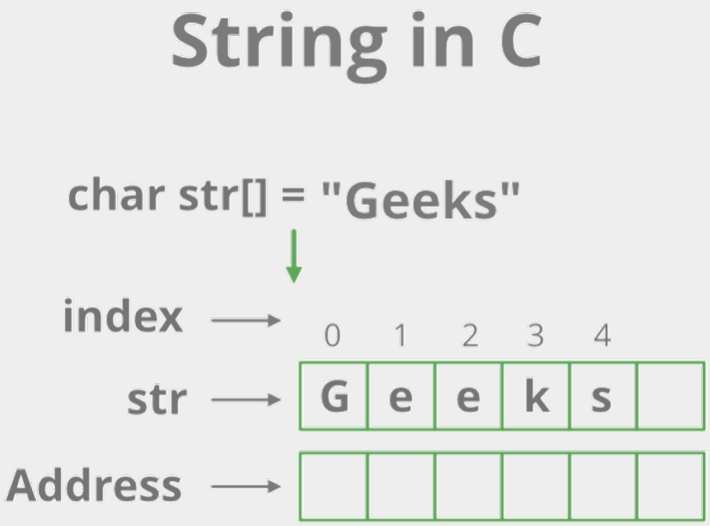
- **if you have a pointer to a struct, you must access its fields with: `->`**

```
Food* pgrapes = &produce[0];
pgrapes->price   = 2.99;
(*pgrapes).price = 2.99;
```

these are identical in meaning.

Common pointer patterns

I.e., `String = char[] = char*`



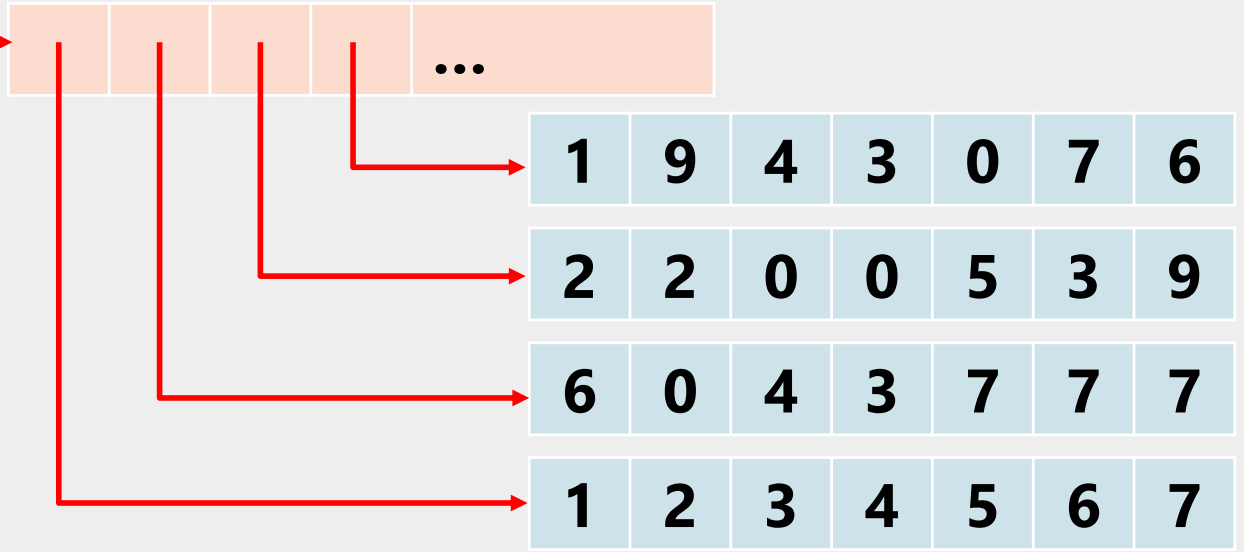
Every problem in CS...

- **...can be solved with another level of indirection/references/pointers.**
- **pointers are the basis of:**
  - strings
  - arrays
  - object-oriented programming
  - dynamic memory management
  - pretty much everything your operating system does
  - pretty much everything... *everything* does.
- **higher level languages often give you more abstract, safer ways of achieving the same things that you can do with pointers**

Multi-dimensional arrays

- **we already saw single-dimensional arrays, but…**

```
int** arr2d = ...
```



a Java `int[][]` works exactly the same way!

Pass-by-reference

- **often you want to give *another function* access to your variables.**

```
fgets(buffer, 100, stdin);
int x, y;
function_that_returns_two_values(&x, &y);
```

since these functions *have access to* buffer, x, and y, they can change their values.

Pass-by-reference (example)

```c
#include <stdio.h>

// Function that modifies the value using a pointer
void modifyValue(int *x) {
    *x = (*x) * 2;
}

int main() {
    int number = 5;

    printf("Original value: %d\n", number);

    // Passing the address of 'number' to modifyValue
    modifyValue(&number);

    printf("Modified value: %d\n", number);

    return 0;
}
```

```
Original value: 5
Modified value: 10
```

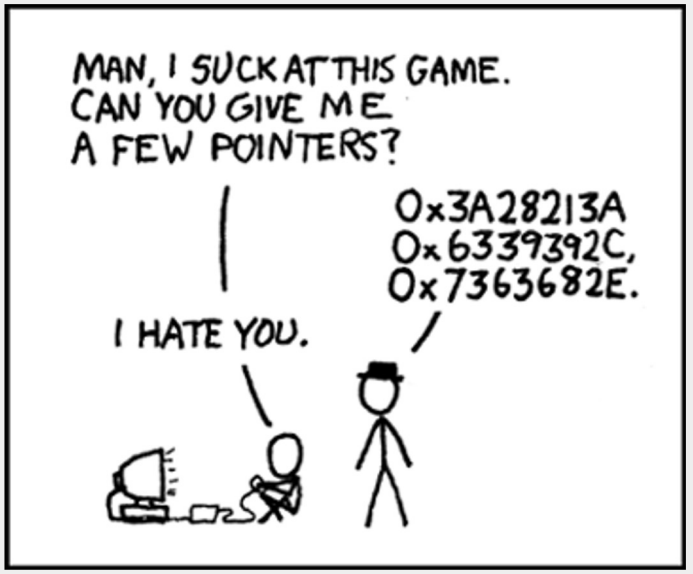
Quiz Time!

(Again, just for completion)

Password: _____

Pointer Lab

***Solve* a series of short coding puzzles to better understand how pointers work!**



Getting set up

1. **Download the starter code:**

   On Thoth:

   `wget https://cs0449.gitlab.io/sp2024/labs/02/pointerlab-handout.zip -O pointerlab-handout.zip`

1. **Unzip to your private directory on Thoth**

   `unzip pointerlab-handout.zip`

- **Creates a directory called `pointerlab-handout` that contains a number of files**
- **You will modify only the file `pointer.c`**

pointer.c

- **Skeleton for some programming exercises**
- **Comment block that describes exactly what the functions must do**
  - and what restrictions there are on their implementation.

TASK: Pointer Arithmetic

Goal

- **Compute the size (how much memory a single one takes up, in bytes) of an `int`**

Hint

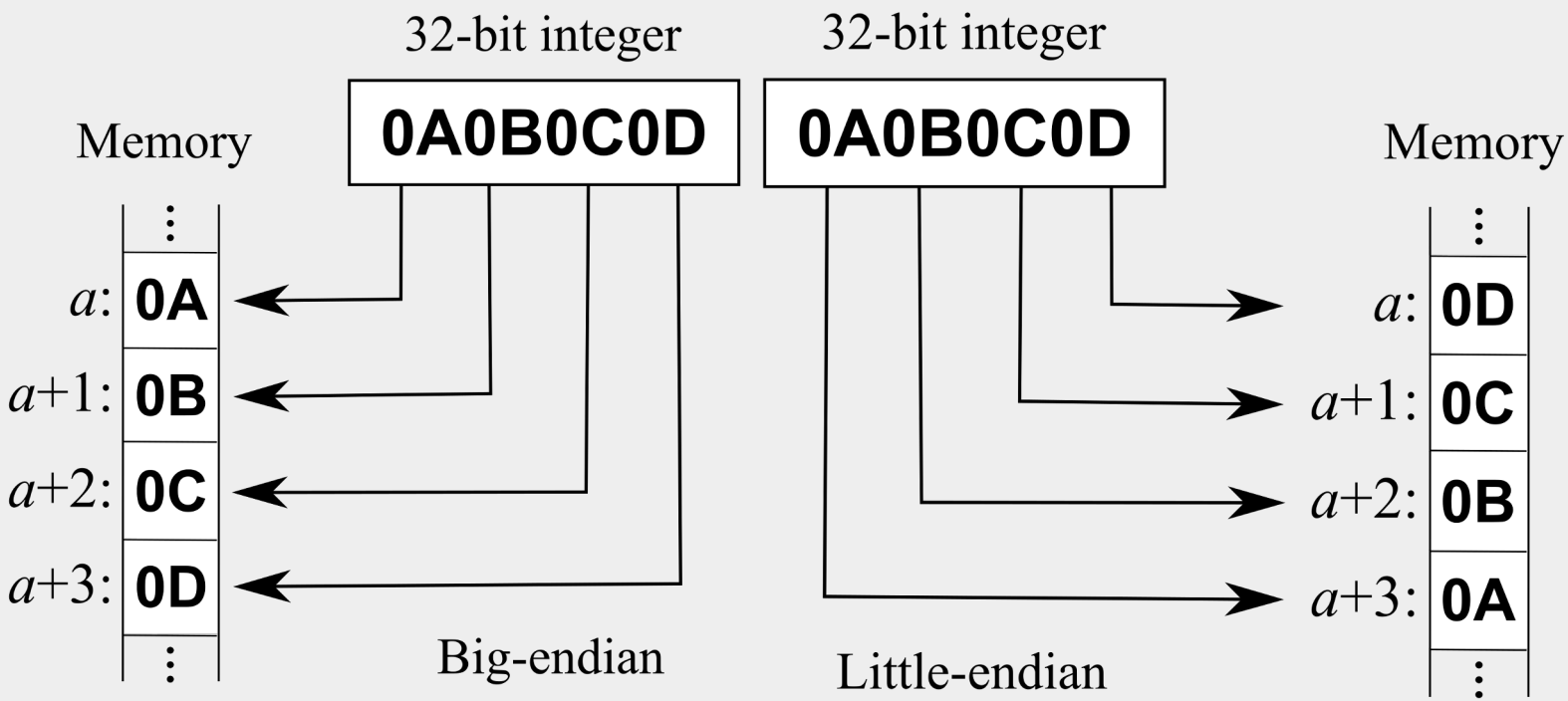
- **Arrays of `ints` allocate contiguous space in memory so that one element follows the next.**

TASK: Manipulating Data Using Pointers

Motive/Goal

- **Manipulate data in new ways with your new knowledge of pointers**
- **`swapInts()` - swap the values that two given pointers point to (without changing the pointers themselves)**
- **`serializeBE()` - change the value of the elements of an array to contain the data in an `int`.**
  - Use **big-endian** order.
  - You are not permitted to use `[]` syntax to access or change elements in the array anywhere in the `pointer.c` file.
- **`deserializeBE()` - does the opposite operation of `serializeBE()`.**
- **The `serializeBE()`/`deserializeBE()` functions emulate what would happen when sending an `int` through the internet.**

As an aside: Endianness



TASK: Pointers and Address Ranges

Goal

- **Determine whether pointers fall within certain address ranges, defined by an array.**
  - Determine if the address stored in `ptr` is pointing to a byte that makes up some part of an array element for the passed array. The byte does not need to be the first byte of the array element that it is pointing to.

```
intArray: 0x0             size: 4 ptr: 0x0 return: 1
intArray: 0x0             size: 4 ptr: 0xF return: 1
intArray: 0x0             size: 4 ptr: 0x10       return: 0
intArray: 0x100 size: 30        ptr: 0x12A      return: 1
intArray: 0x100 size: 30        ptr: 0x50       return: 0
intArray: 0x100 size: 30        ptr: 0x18C      return: 0
```

TASK: Byte Traversal

Motive

- **Learn to read and write data by understanding the layout of the bytes.**

Background

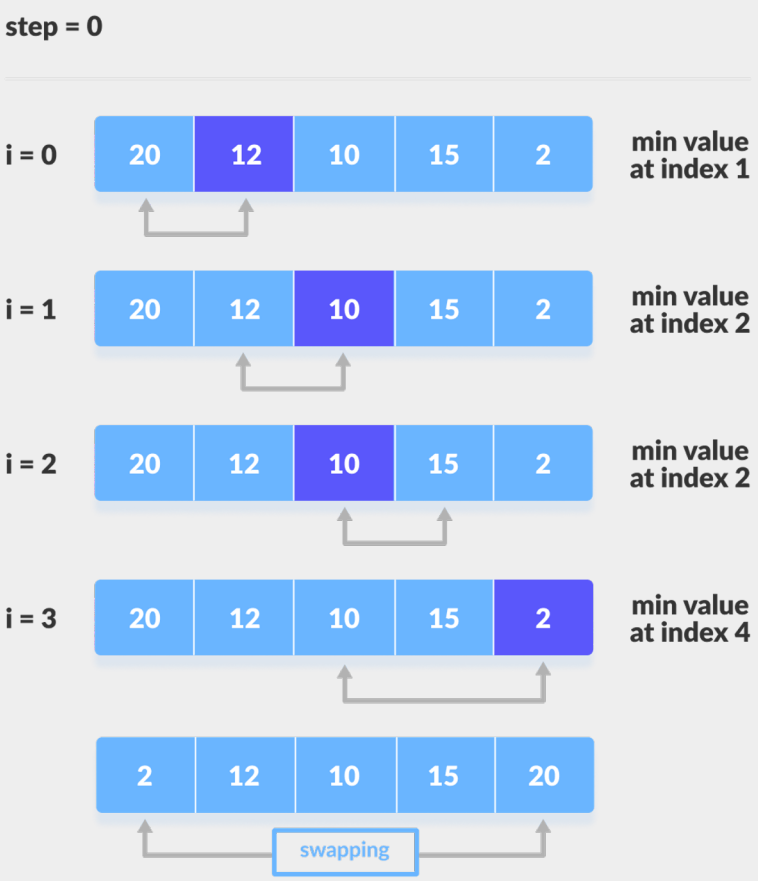
- **C strings do not not how '*long*' they are (No `.length()` method).**
  - We need to calculate this ourselves.
  - All C strings are arrays of characters that end with a null terminator, `\0`.

Goal

- **`stringLength()` - returns the length of a string, given a pointer to its beginning.**
  - Note that the null terminator character does NOT count as part of the string length.
- **`stringSpan (str1, str2)` - returns the length of the initial portion of str1 which consists only of characters that are part of str2.**
  - The search does NOT include the terminating null-characters of either strings, but ends there.

TASK: Selection Sort

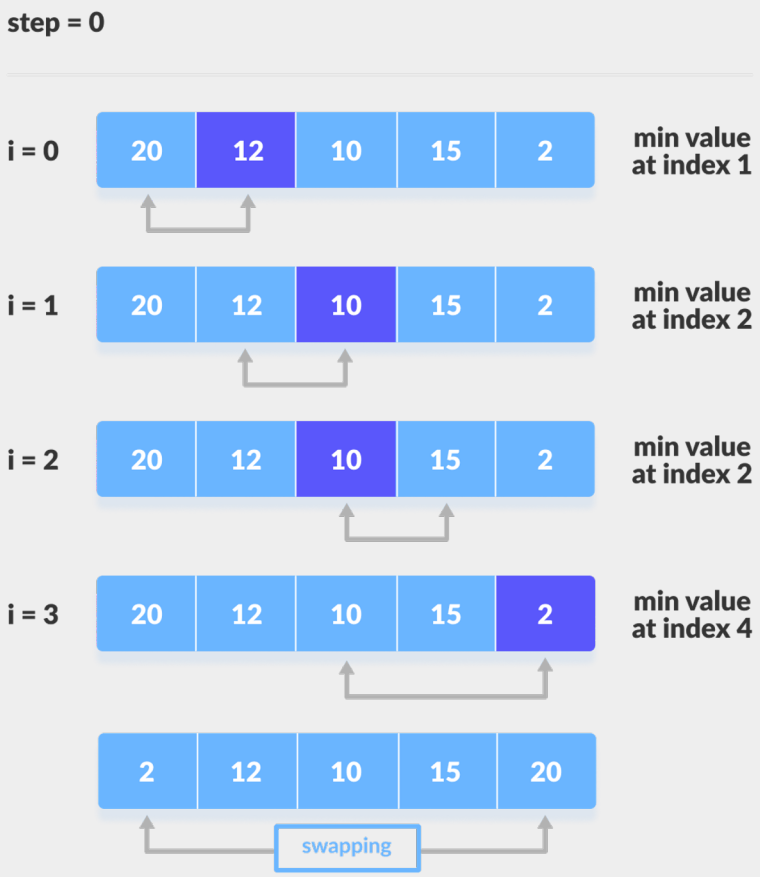
- **Your final task is to implement selection sort**
  - Just like 445... but in C
  - You **may** use *loops* and *if statements*
  - But still no array syntax (`array[]`)



In case you forgot…

```
Let:
    arr:= array
    n:= the length of arr

for i = 0 → (n-1)
    minIndex = i
    for j = (i+1) → n
        if arr[minIndex] > arr[j]
            minIndex = j
        end if
    end for
    swap(arr[i], arr[minIndex])
end for
```



Evaluation

- **The following driver program has been provided to help you check the correctness of your work:**

  `ptest`

  checks **functional correctness:** *Does your solution produce the expected result?*

  To use:

  1. Build using `make`
  2. Run using `./ptest`
     - You must rebuild each time you modify `pointer.c`

- **Gradescope Autograder may test your program on inputs that `ptest` does not check by default.**
- **Coding style (restriction) will be checked by grader TA on Gradescope**