Basics of C Programming

CS 0449: Introduction to System Software

CS0449 Teaching Assistants

University of Pittsburgh
School of Computing and Information

Meta-Notes

- These slides were adapted heavily from recitation slides created by *Martha Dixon* who was a teaching assistant (TA) for this course in Fall of 2020. They contain materials which were obtained from various sources, including, but not limited to, the following:

[1] J. Misurda, CS 0449: Introduction to Systems Software, 3rd ed. Pittsburgh, PA: University of Pittsburgh, 2017.

[2] S. J. Matthews, T. Newhall, and K. C. Webb, Dive into Systems: A Gentle Introduction to Computer Systems. San Francisco, CA: No Starch Press, 2022.

[3] R. Bryant, D. R. O'Hallaron, and M. S., Computer Systems: A Programmer's Perspective. Princeton, NJ: Pearson, 2016.

[4] L. Oliveira, V. Petrucci, and J. Misurda, in Introduction to Systems Software, 2022

Agenda

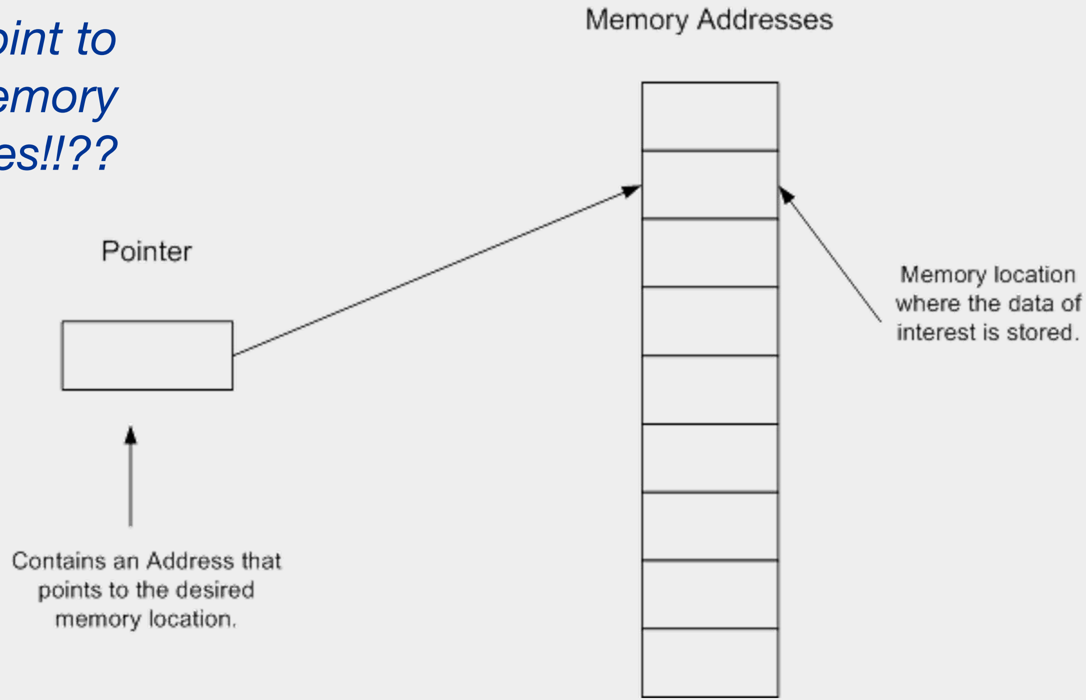
- Course News!
- Pointer Lab
- File I/O in C
  - Standard integer sizes
  - Reading/writing files
- Project 1
- Tophat

Course News

- Project check-ins
- Due date: 9th
- Lab slides: https://sites.pitt.edu/~shk148/teaching/CS0449-2241/#handouts

Pointers

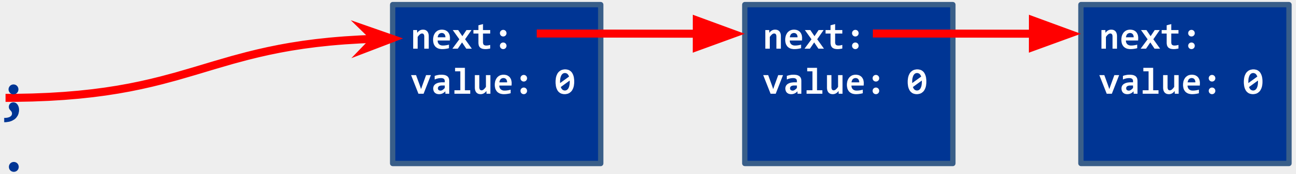
Point to here, point to there, point to that, point to this, and point to nothing! well, they are just memory addresses!!??



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 <u>pointer</u> 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**`

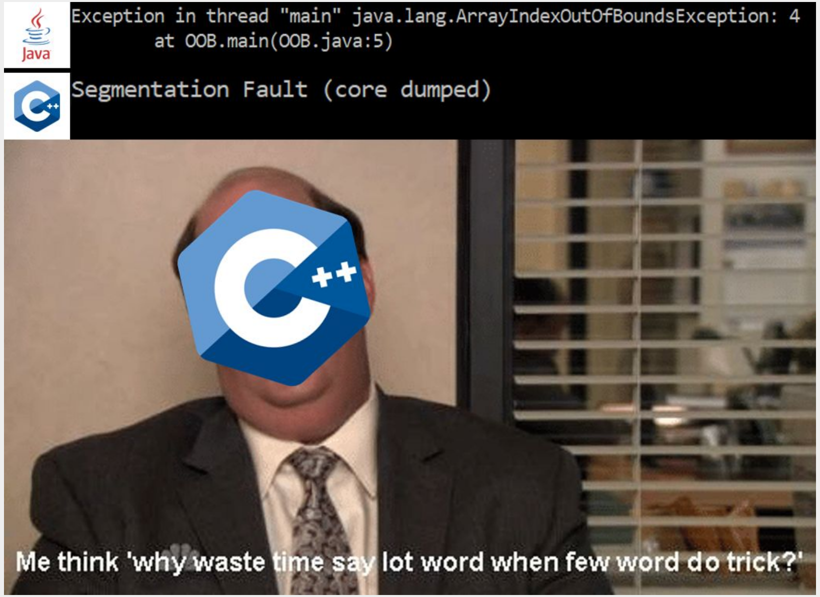
a **pointer** to a **pointer,** which points to an **int.**

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;
```

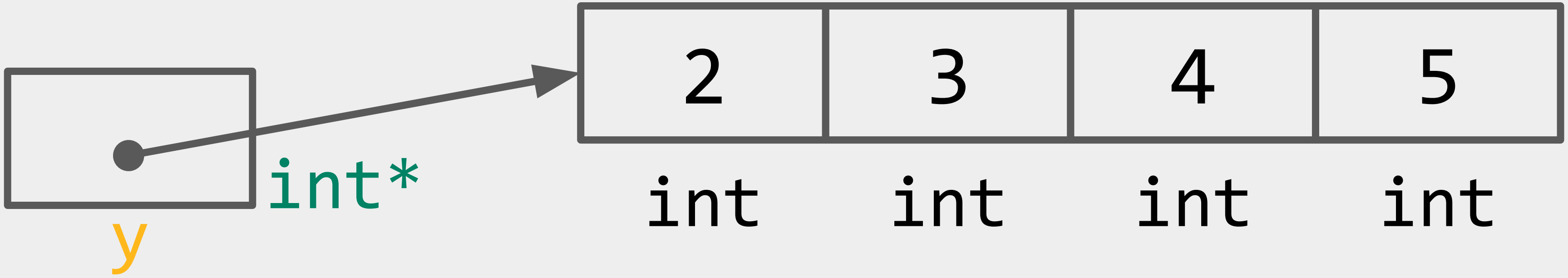
goes to the address that **`ppx`** contains, and gets the **`int*`** there

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

Arrays are just pointers *well…sort of*

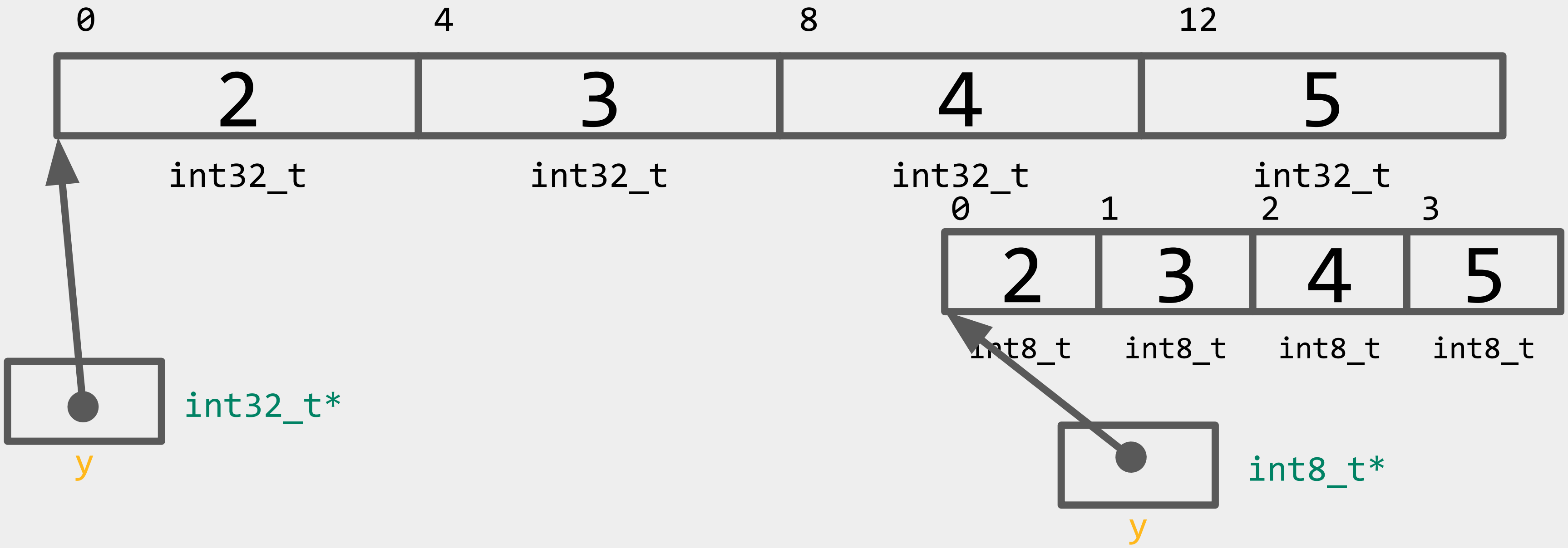
- **In C, array names are just aliases that can be used as pointers**
  - `int y[] = {2, 3, 4, 5}; // these two are`
  - `int *y = {2, 3, 4, 5}; // roughly equivalent`
- **Indexing and dereferencing pointers are equivalent**
  - Side note: you can do math with pointers…this is called **pointer arithmetic.**
  - 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:**

```c
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

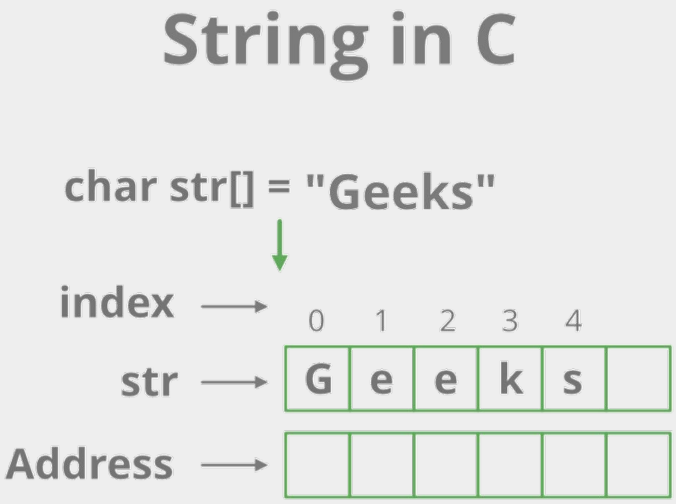
- **if you have a pointer to a struct, you must access its fields with: ->**

```
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*



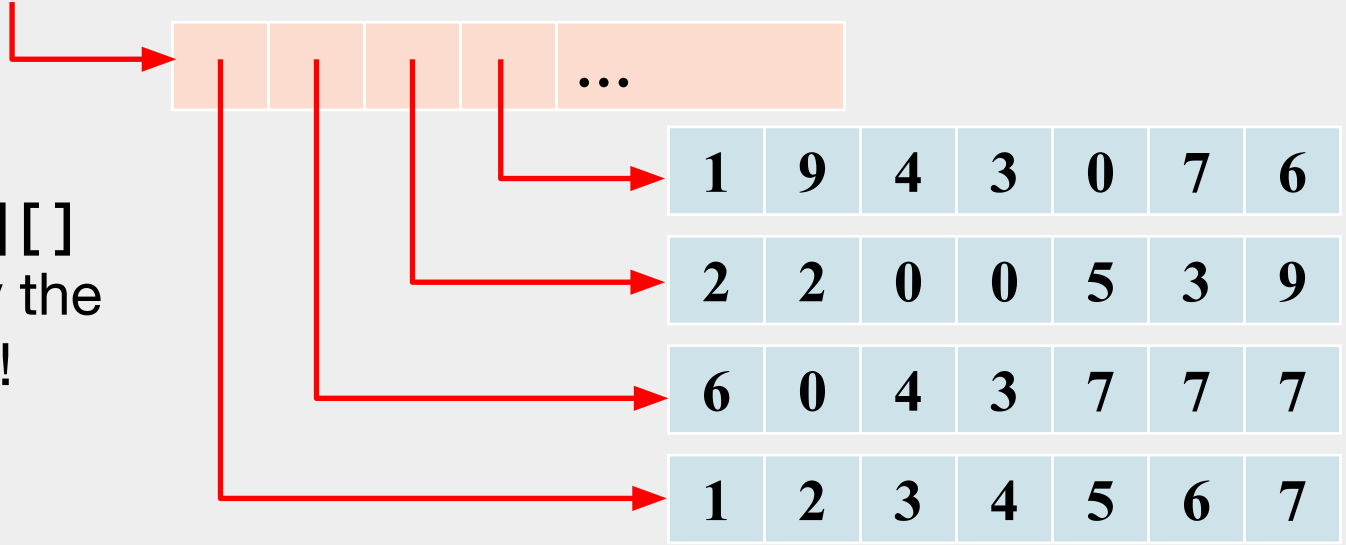
Every problem in CS...

- **...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
```

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/fa2023/labs/02/pointerlab-handout.zip -O pointerlab-handout.zip
```

1. Unzip to your <u>private</u> 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

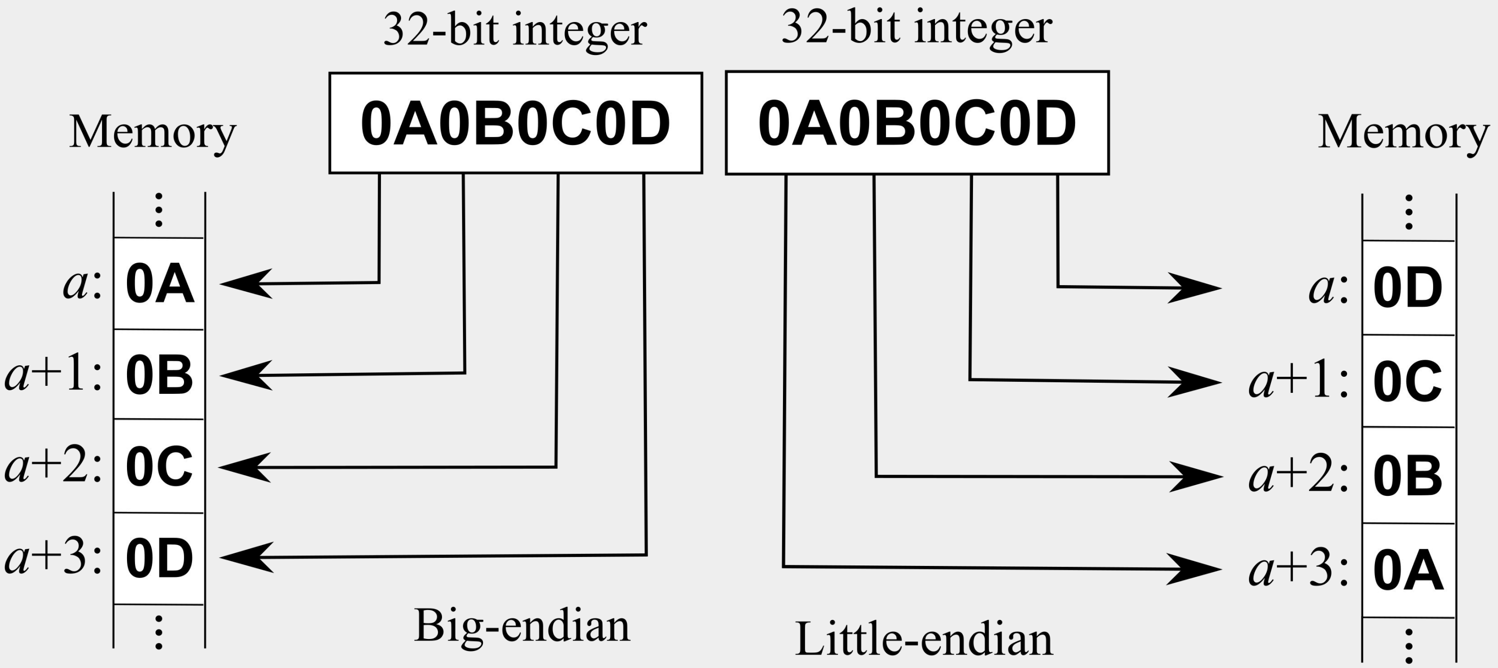
- **Arrays of `int`s allocate contiguous space in memory so that one element follows the next.**

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

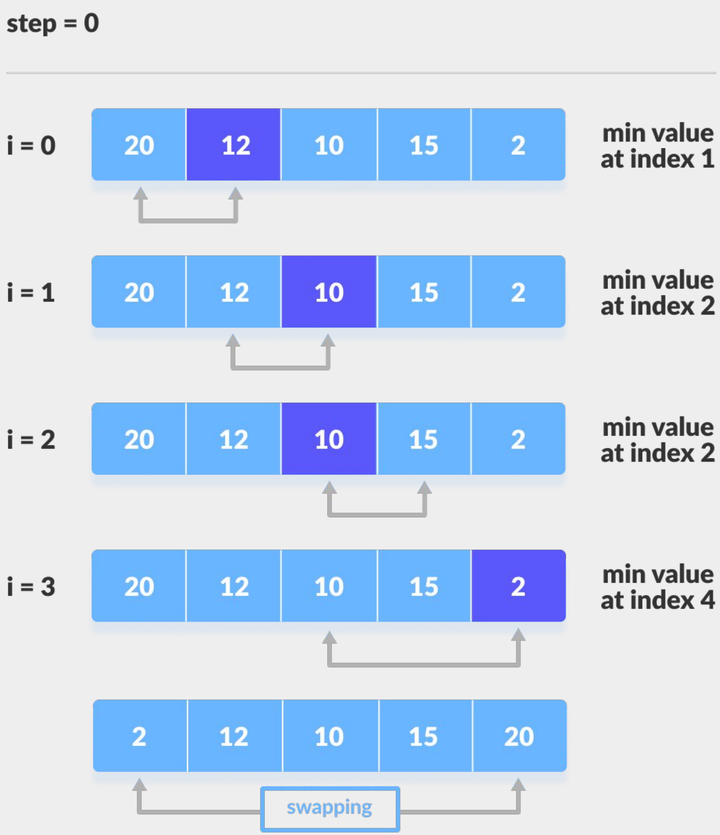
- **C strings do not not how '*long*' they are (No `.length()` method).**
  - We need to calculate this ourselves.
  - All C strings are arrays of characters that end with a null terminator, `\0`.

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.

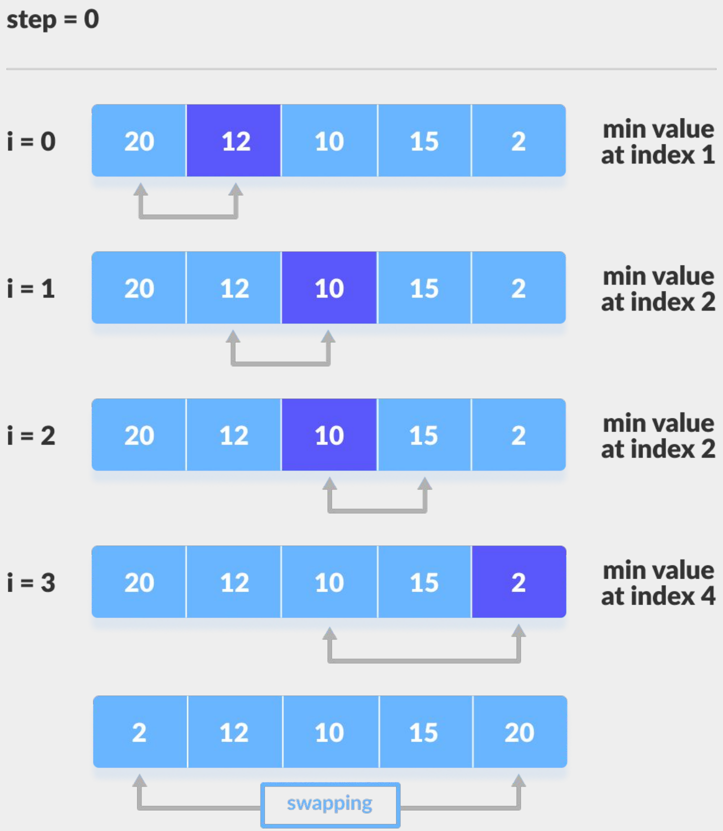
TASK: Selection Sort

- **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 **<u>functional correctness</u>:** *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**

Basics of File I/O

Reading and writing files in C

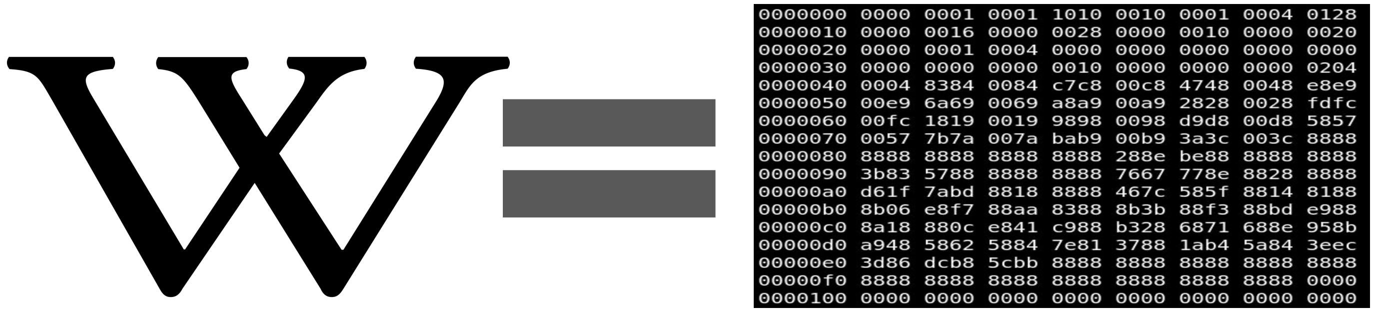
```
[[ ~ ]$ hexdump -C binary_file_example
00000000  41 00 41 00 00 00 42 00  42 00 00 00 43 00 43 00  |A.A...B.B...C.C.|
00000010  00 00 44 00 44 00 00 00  45 00 45 00 00 00 46 00  |..D.D...E.E...F.|
00000020  46 00 00 00 47 00 47 00  00 00 48 00 48 00 00 00  |F...G.G...H.H...|
00000030  49 00 49 00 00 00 4a 00  4a 00 00 00 4b 00 4b 00  |I.I...J.J...K.K.|
00000040  00 00 4c 00 4c 00 00 00  4d 00 4d 00 00 00 4e 00  |..L.L...M.M...N.|
00000050  4e 00 00 00 4f 00 4f 00  00 00 50 00 50 00 00 00  |N...O.O...P.P...|
00000060  51 00 51 00 00 00 52 00  52 00 00 00 53 00 53 00  |Q.Q...R.R...S.S.|
00000070  00 00 54 00 54 00 00 00  55 00 55 00 00 00 56 00  |..T.T...U.U...V.|
00000080  56 00 00 00 57 00 57 00  00 00 58 00 58 00 00 00  |V...W.W...X.X...|
00000090  59 00 59 00 00 00 5a 00  5a 00 00 00              |Y.Y...Z.Z...|
0000009c
```

What we have seen so far ...

- **In lab 0, you (maybe unknowingly) used command line arguments to interact with your program**
  - When you ran `./calculator 4 5 +`
- **In lab 1, you used the standard I/O stream(s)**
  - `printf()`, `scanf()`, and other `<stdio.h>` functions
- **This week, we'll learn to read and write from files on your computer**
  - which you will need to do for the first project

What is a file?

- **In C, a file is simply a sequence (*stream*) of bytes:**
  - Text files (or ASCII file) is sequence of ASCII code, i.e., each byte is the 8 bit code of a character (*.txt, *.c, etc.)
  - Binary files contains the original binary number as stored in memory (*.pdf, *.doc, *.jpg, etc.)



A hex dump of the 318 byte Wikipedia favicon

Opening files with fopen()

```
FILE *fopen(const char * pathname, const char*mode);
```

```
> FILE* pt = fopen("E:\\PATH\program.txt","w");
```

- opens the file whose name is the string pointed to by pathname and associates a stream with it.
- returns a pointer (of type FILE) to the stream

Opening Files with `fopen()`

```
*fopen(const char * filename, const char * mode );
```

Modes:

- `r`: opens an existing file for reading.
- `w`: opens a file for writing.
  - If `filename` does not exist, new file is created.
  - starts writing at the beginning of file.
- `a`: opens a text file for writing in appending mode.
  - If `filename` does not exist, new file is created.
  - start appending content in the existing file content.
- `r+`: opens a file for both reading and writing.
- `b`: indicates file is a binary file
- and more…
  - Use **`man fopen`** to learn more

fread() lets us read, fwrite() lets us write

```
fread(void *ptr, size_t size, size_t nmemb, FILE* stream);
```

- reads nmemb items of data each size bytes long
- from stream
- stores them at the location given by ptr.

```
fwrite(const void *ptr, size_t size, size_t nmemb, FILE * stream);
```

- writes nmemb items of data each size bytes
- to the stream
- from the location given by ptr.

Reading and writing moves the pointer

```
File * stream File * stream   File * stream

10100110101111111011111010111111100100011
10010000110000100100010010010000101100100
10100110101111111011111010111111100100011
10010000110000100100010010010000101100100
10100110101111111011111010111111100100011
10010000110000100100010010010000101100100
```

```
> fread(ptr1, 1, 1, stream)
...
> fwrite(ptr1, 1, 1, stream)
```

Example

```
> fread(ptr1, 1, 1, stream)
```

This reads 1 byte and moves the file position indicator by 1 byte (8 bits).

```
> fread(ptr1, 4, 1, stream)
```

This reads 1 block of 4 bytes, moving the file position indicator by 4 bytes (4 * 8 = 32 bits).

```
> fread(ptr1, 4, 2, stream)
```

This reads 2 blocks of 4 bytes each from the file stream, moving the file position indicator by $4 \times 2 = 8$ bytes (8 * 8 = 64 bits).

We can rewind or fast-forward with `fseek()`

```
fseek(FILE *stream, long offset, int whence);
```

- sets the file position indicator for the stream
- new position (measured in bytes) = `offset` + `whence`.

`whence`:

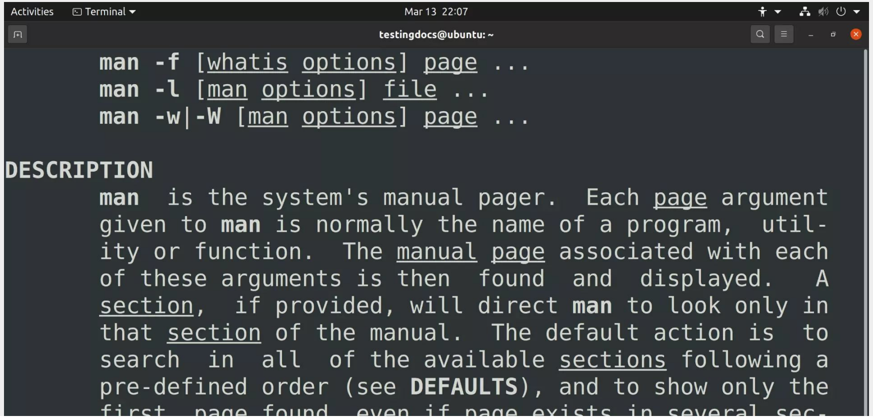
- `SEEK_SET` - from start-of-file
- `SEEK_CUR` - from current position
- `SEEK_END` - from end-of-file

Example

- **fseek(file, 10, SEEK_SET)**
  moves the file position indicator 10 bytes from the beginning of the file.
- **fseek(file, 10, SEEK_CUR)**
  moves the file position indicator 10 bytes forward from the current position in the specified file stream.
- **fseek(file, 10, SEEK_END)**
  moves the file position indicator 10 bytes before the end of the specified file stream.

Always remember to save (and close) your files!

- **Just like memory leaks, you may also get file handle leaks**
  - If you use `fopen()`, always remember to `fclose()`
    - `int fclose(FILE* filePointer)`
      - returns `0` on success!
- **If you are confused about these functions → Consult the `MANual`**



Thoth `man` errors: *try* `MANPATH= man 3 fopen`

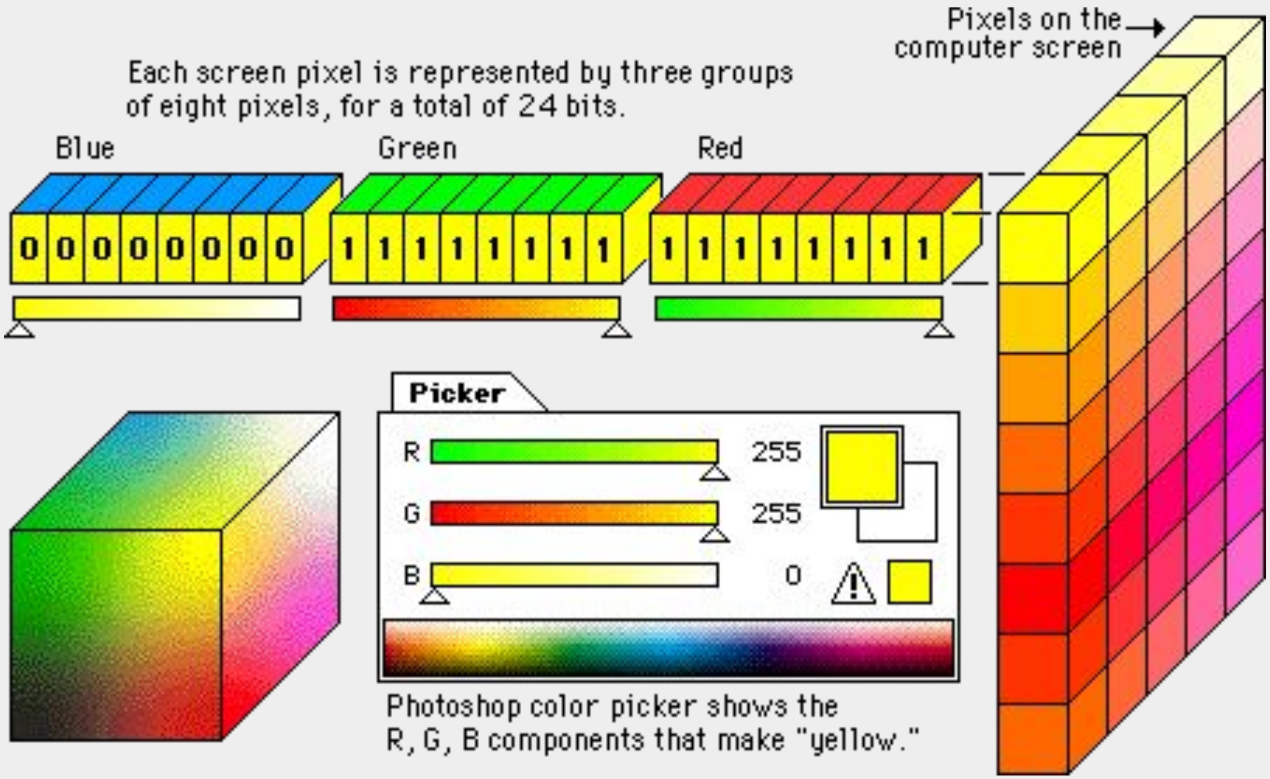
Project 1

Quick Guide

Project Brief

- The goal of this project is to convert a CBM file into a BMP file
  - CBM is a custom file format made by Dr.Luis
  - BMP is a standard image format
- `*.BMP` ⇒ Bitmap Image File
  - Container format for a big array of pixels (picture cells)
  - Each pixel is represented by a 24-bit number:
    - 8 bit for Red (0-255)
    - 8 bit for Green (0-255)
    - 8 bit for Blue (0-255)

Pixels



Pixels on the computer screen
Each screen pixel is represented by three groups of eight pixels, for a total of 24 bits.
Blue
Green
Red
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1
Picker
R
255
G
255
B
0
Photoshop color picker shows the R, G, B components that make "yellow."

CBM file

- Each CBM file consists of:
  - **A header**
    - Which contains metadata about the file (image size, number of colors, etc.)
  - **Color palette**
    - n RGB values
  - **Image**
    - Each pixel is represented as a single byte which indexes the color from the palette
    - E.g., `pixel`$_i$ `= 7`
      - $\Rightarrow$ `pixel`$_i$ `= palette[7]`

Phase 1. Read the CBM file

- Your task is to read the CBM header & palette and display it to the terminal
  - Hint: defines `structs` and read the structs using `fread(&stuct,...)`
- How many colors in palette?
  - See number of colors from header.

```
$ ./cbm2bmp --info CBM_FILENAME
=== CBM Header ===
Magic: 0x7449
Width: 958
Height: 718
Number of colors: 16
Color array offset: 22
Image array offset: 70

=== Palette (R, G, B) ===
Color 0: (24, 36, 21)
Color 1: (37, 66, 26)
Color 2: (56, 91, 41)
Color 3: (77, 113, 63)
Color 4: (5, 9, 5)
Color 5: (53, 51, 49)
Color 6: (102, 102, 103)
Color 7: (75, 74, 72)
Color 8: (104, 139, 86)
Color 9: (176, 188, 219)
Color 10: (150, 164, 172)
Color 11: (127, 146, 128)
Color 12: (123, 78, 204)
Color 13: (198, 208, 129)
Color 14: (219, 119, 118)
Color 15: (163, 41, 75)
```

BMP File

- The beginning of the BMP is a header which contains metadata (key details about the picture)

54 bytes — Header

Image content

24-bit lena.bmp

54 bytes — b13ae35152ab5d93a49 cc112b044d2a9b13ae3

Image content —
b044d2a9b13ae35152ab
5d93a49cc112b044d2a9
251d9ed69482c88a4077
c619bcfcf227afbc0c6e
6529aac435c769696ddb
6206340827ab559e6029
a99aae4a62f622a699ea
e0750fec6f65f74d5d28
f1e9a99aae4

24-bit lena.bmp

BMP Header

Header

File header (14 bytes)

DIB Header (40 bytes)

Bitmap File Header	
Identifier (ID)	2
File Size	4
Reserved	4
Bitmap Data Offset	4
DIB Header	
Bitmap Header Size	4
Width	4
Height	4
Planes	2
Bits Per Pixel	2
Compression	4
Bitmap Data Size	4
H-Resolution	4
V-Resolution	4
Used Colors	4
Important Colors	4

Phase 2. Generate BMP Header

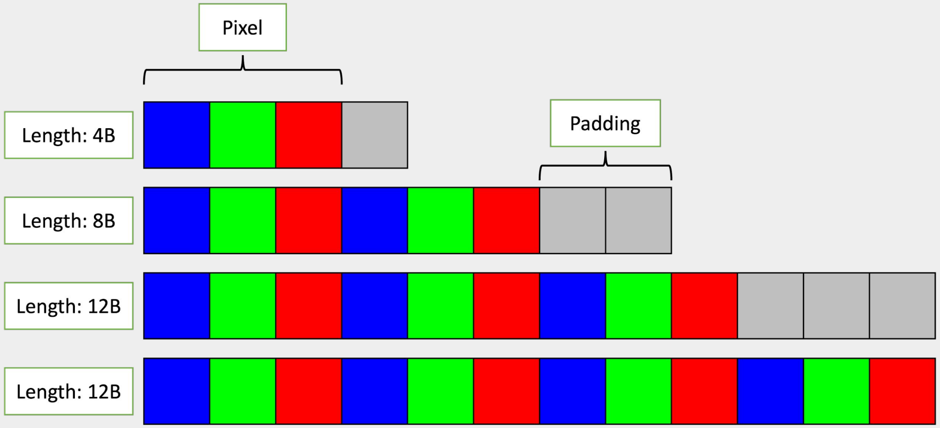
```
$ ./cbm2bmp --bmp-info CBM_FILENAME
=== BMP Header ===
Type: BM
Size: 2073654
Reserved 1: 0
Reserved 2: 0
Image offset: 54

=== DIB Header ===
Size: 40
Width: 960
Height: 720
# color planes: 1
# bits per pixel: 24
Compression scheme: 0
Image size: 0
Horizontal resolution: 0
Vertical resolution: 0
# colors in palette: 0
# important colors: 0
```

First two bytes must be BM (not Nul-terminated)

Derive size from CBM file

Size of BMP

- Size of BMP file
  - Size of Header + Size of Image
    - Size of Image = Width * Height * Size of Pixel
- Note. Width must account for padding
  - Padding is applied if length of each row is not a multiple of 4 Bytes



Phase 2. Generate BMP Header

```
$ ./cbm2bmp --bmp-info CBM_FILENAME
=== BMP Header ===
Type: BM
Size: 2073654
Reserved 1: 0
Reserved 2: 0
Image offset: 54

=== DIB Header ===
Size: 40
Width: 960
Height: 720
# color planes: 1
# bits per pixel: 24
Compression scheme: 0
Image size: 0
Horizontal resolution: 0
Vertical resolution: 0
# colors in palette: 0
# important colors: 0
```

First two bytes must be BM (not Nul-terminated)

Derive size from CBM file

Keep reserved values as zero

See handout for the rest.

Phase 3. Construct the BMP

- Combining phase 1 and 2 ⇒ Construct the BMP file
  - `fwrite()` to a file
  - Must write headers to file, then pixels
- Caveats
  - In CBM file:
    - Pixels in palette are RGB
    - Each entry in the image section is an index of the palette
    - Pixels are stored Top → Bottom
  - In a BMP:
    - Pixels are BGR; Pixels are stored directly in the image section (no indexing a palette)
    - Each row has padding
    - Pixels are stored Bottom → Top

Phase 3. Convert the image

010011000001101100101110011
00001001011001010011 0001
1001010001111100001 10 11011
0101010111110001110 10 0011
100000000101011 00
00101111101110 1 00
110011110001100 111
101001111001110 111
1111000111101001 01
0110101111010100000000010001
0000010111101100000011100
0001000000110010000010000
0111011010101101010110101011
0100101110000010110000 1011

7

Palette
(from CBM)

10001011
00111011
00111011

24-bit lena.cbm

24-bit lena.bmp

Remarks

- See handout for
  - Reading command line arguments
  - **Compactness of Structs**
    - !!!
  - Makefiles