Introductions, C Refresher

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra Khoi Dinh Sydney Fitzgerald Rania Souissi

Jeffrey Oduman Abdullah Goher Monisha Krothapalli Veer Kakar

Kevin Xue Shreya Mukunthan Daniel Price Wesley Liu

Leo Zhang

pollev.com/tqm

- Favorite thing you ate over break?

Administrivia

- First Assignment (penn-vector)
  - Releases After Class
    - Expect an announcement on Ed sometime tonight!
  - **"Due" next Friday, 9/04**
  - Mostly a C refresher

- Pre-Semester Survey
  - Anonymous
  - Short!
  - Releases Tonight with penn-vec
  - Due Tuesday, 9/08

Lecture Outline

- **Introduction & Logistics**
  - **Course Overview**
  - **Assignments & Exams**
  - **Policies**
- C "Refresher"
  - memory
  - Pointers
    - Output Parameters
  - Arrays
  - Structs

Instructor: Travis McGaha

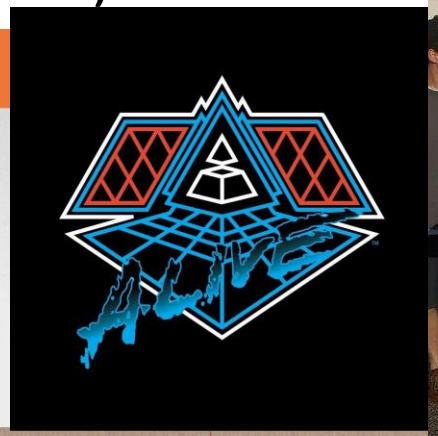
- UPenn CIS faculty member since August 2021, originally from Tampa FL & Houston TX

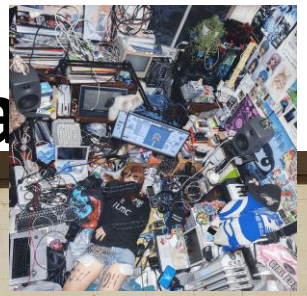
- Education & Teaching
  - Undergrad & Masters in Computer Engineering @ UW Seattle
  - 4th time instructor for CIS 2400 :)

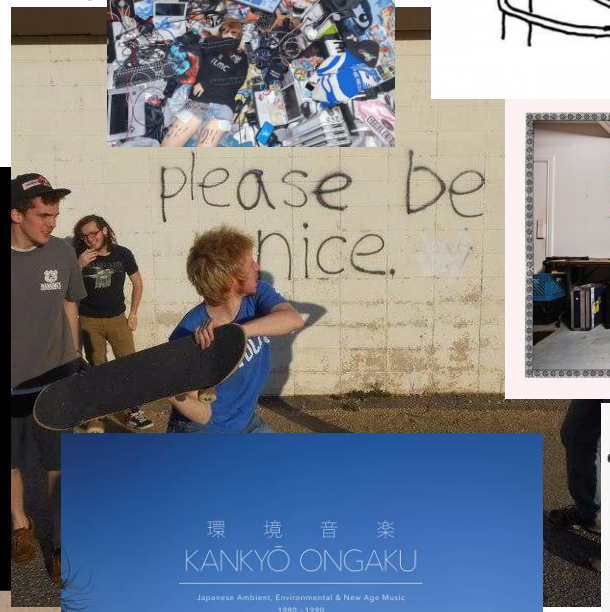
- Outside of teaching…
  - Executive committee for AAUP-Penn
  - Hiking (when I get the chance), playing games (Magic The Gathering, video games, board games), my cat, and…

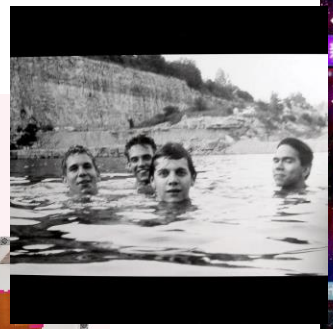
Instructor: Travis McGaha

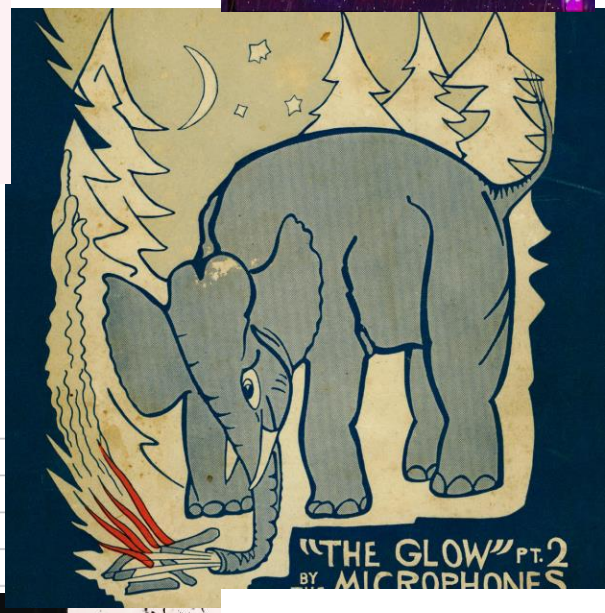
- I like most music :)

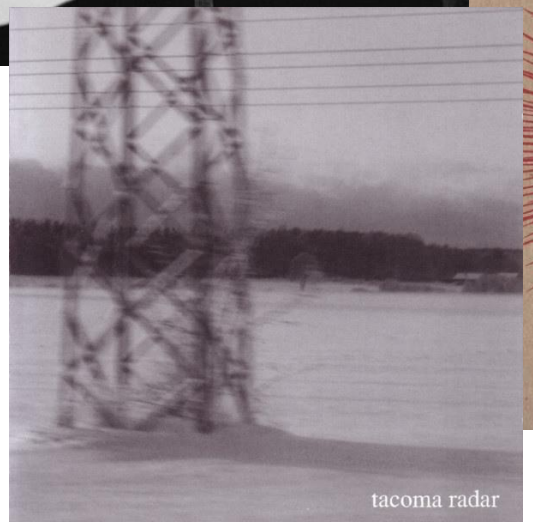


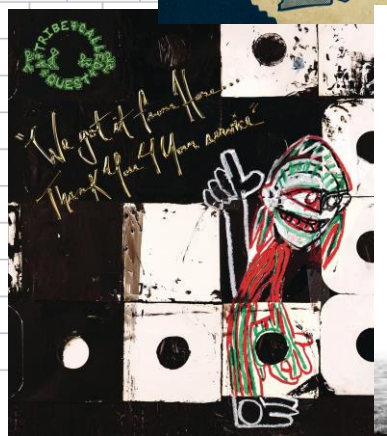












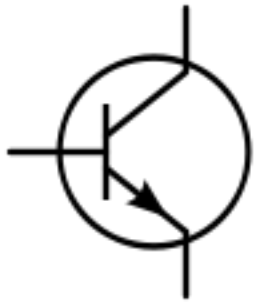
We Care

- We care a lot about your actual learning and that you have a good experience with the course.

- We are human beings, and we know that you are one too. If you are facing difficulties, please let us know and we can try and work something out.

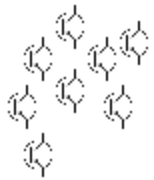
- We are always willing to chat about anything. Book a time to meet with us!
  - https://www.cis.upenn.edu/~tqmcgaha/#meetings
  - Or come to office hours!

Course Overview

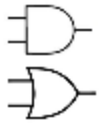


Course Overview

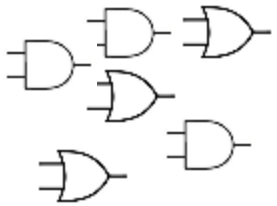
Course Overview



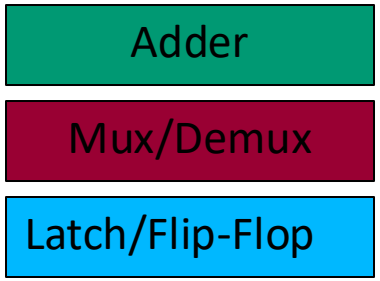
Course Overview



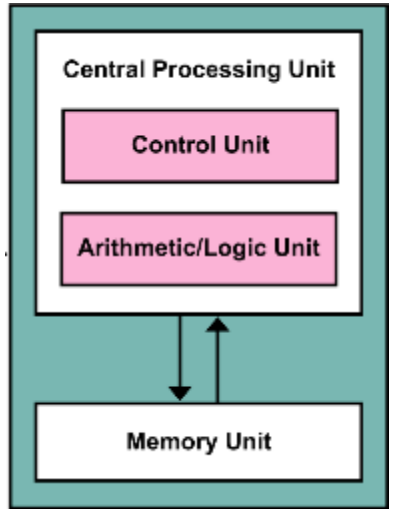
Course Overview



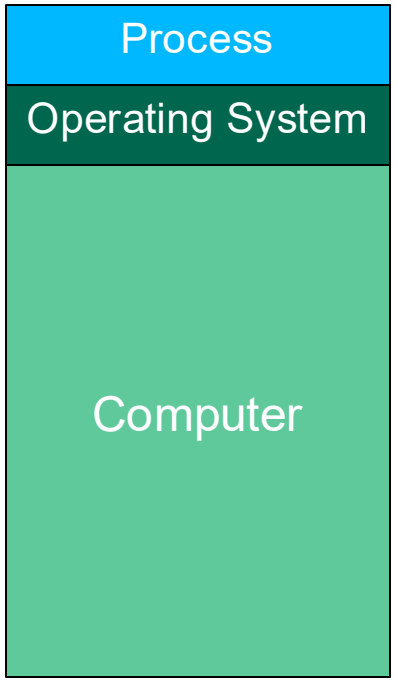
Course Overview



Course Overview



Course Overview



“Lies-to-children”

- "The necessarily simplified stories we tell children and students as a foundation for understanding so that eventually they can discover that they are not, in fact, true."
  - Andrew Sawyer (Narrativium and Lies-to-Children: 'Palatable Instruction in 'The Science of Discworld' ‘)

“Lies-to-children”

- "A lie-to-children is a statement that is false, but which nevertheless leads the child's mind towards a more accurate explanation, one that the child will only be able to appreciate if it has been primed with the lie"
  - Terry Pratchett, Ian Stewart & Jack Cohen (The Science of Discworld)

Question

- What color is the sky?

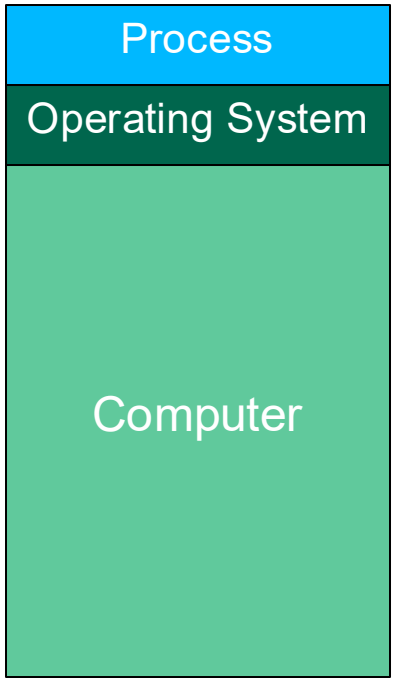
Question

- What color is the sky?

We lied to you (but in a good way)

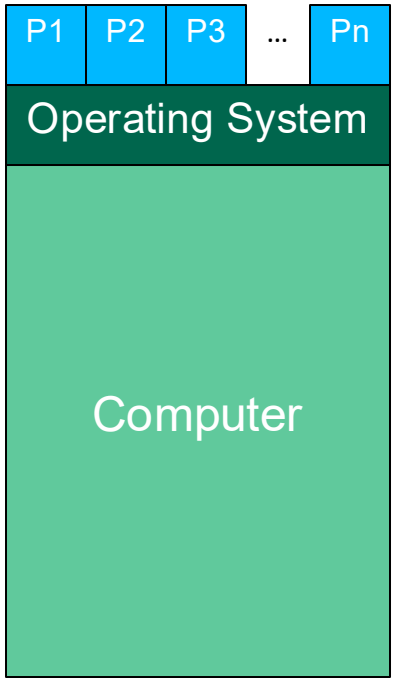
- Is memory one giant array of bytes? **Eh……. no**
- Is this a useful model? **Yes**

Course Overview



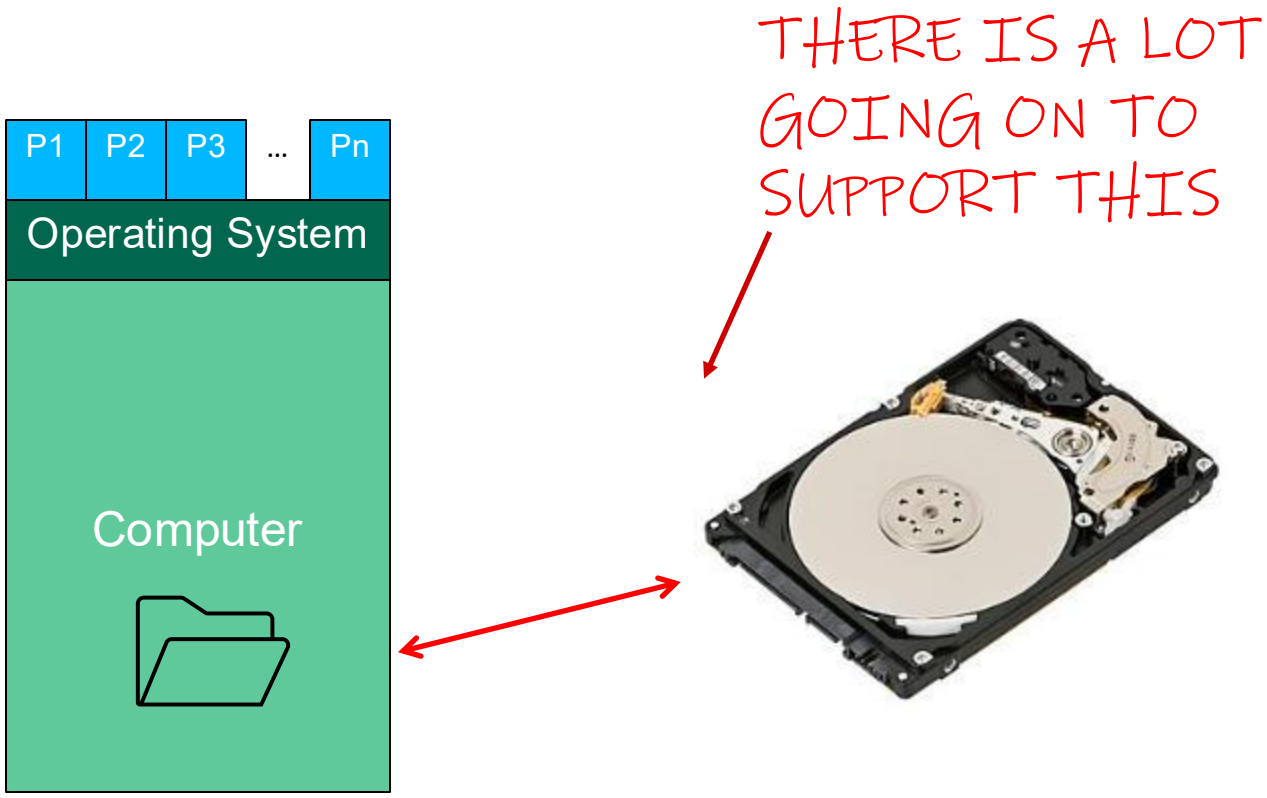
OS does A LOT more than just printing, reading input, video display, and timer

Course Overview



THERE IS A LOT GOING ON TO SUPPORT THIS

Course Overview



We're going to lie to you (but in a good way)

- "All models are wrong, but some are useful."
  - Same source as below.

- "If it were necessary for us to understand how every component of our daily lives works in order to function - we simply would not."
  - AnRel (UNHINGED: A Guide to Revolution for Nerds & Skeptics)

- This course will reveal more details, but there is still a ton we are leaving out. Even what we say that is accurate, will likely change in the future.

Prerequisites

- Course Prerequisites:
  - CIS 2400 (or equivalent previous experience)
  - Teamwork & Willingness
- What you should be familiar with already:
  - C Programming
  - C Memory Model
  - Computer Architecture Model
  - Basic UNIX command line skills
- HW00 and HW01 are tuned so that it will help refresh you on these.
  - Even if you think you know C, get started sooner rather than later.

CIS 5480 Learning Objectives

- To leave the class with a better understanding of:
  - How a computer runs/manages multiple programs
  - How the previous point may affect the code we write
    - Extra relevant with hardware prices and performance concerns
  - How to read documentation
  - Experience writing a massive programming project FROM SCRATCH with others
  - More comfortable writing C code

- Topics list/schedule can be found on course website
  - Note: This is tentative

Disclaimer

- Parts of the course are tentative
  - Some of the project milestones & demos
  - Recitation attendance
  - Schedule of topics

- This is a digest, **READ THE SYLLABUS**
  - https://www.seas.upenn.edu/~cis5480/current/documents/syllabus
  - Note: Syllabus is mostly finalized but may undergo a few tweaks still

Course Components: Textbook

- Textbook (0)
  - Textbooks recommended in pasts
    - A.S. Tanenbaum. Modern Operating Systems (4th Edition onwards). Prentice-Hall.
    - W. Richard Stevens and Stephen A. Rago. Advanced Programming in the UNIX Environment (2/e or 3/e). Addison-Wesley Professional.
  - Systems for all: https://diveintosystems.org/book/
    - Free online textbook, pretty well written
  - Linux Man pages:
    - https://linux.die.net/man/
    - https://www.man7.org/linux/man-pages/
    - The **`man`** command in the terminal
    - DEMO:
      - name a C function
      - tcsetpgrp

Course Components: Part 1

- Lectures (2.5%)
  - Introduces concepts, slides available on website, and recordings available on Canvas
  - In lecture polling (graded on completion, need to complete 80% of them).
    - Polls will be left open until the next lecture so you can complete them afterwards.
  - Some lectures are dedicated to exam review and exams

- Recitations (2.5%)
  - Fridays @ 10:15 – 11:45 AM or 1:45 – 3:15 pm in Levine 101
  - Extra coding and exam style practice, often relevant to the programming projects
  - Worksheets collected for attendance, graded on the **attempt** not correctness.
    By "attempt" we don't mean that you put your name, we mean that it looks like you tried to do the work.
  - Need to get credit for 80% of recitations.

Course Components: Part 2

- Exams (45%)
  - Quiz on 9/29 in class, covers lectures 0 (today) through 7 (5%)
  - Midterm on 10/22 in class, covers lectures 0 through 14 (15%)
  - Final during Final Exam Period, cumulative (25%)
- Pre-recorded videos
  - Entirely optional
  - Goes over lecture material or demonstrates something for projects
- Projects (50%)
  - See next couple slides

Programming Facilities

- Docker
  - Same environment as the Autograders
  - Instructions for setup on the website
  - WSL and orbstack are also available if you want, but primary support is for Docker.

- Speclab cluster, as a fallback incase Docker does not work
  - Instructions on course website
  - To see status: https://www.seas.upenn.edu/checklab/?lab=speclab

- **DO NOT use Eniac machines to develop projects for this class!**

Project 0 – penn-vector

- Project 0 (2.5%)
  - Basic data structure in C: A dynamically resizable array (e.g. Vector or ArrayList)
  - Optional Extension: make an easier to use generic version w/ macros
  - Idea is to help you get comfortable with coding in C
    - C
    - Structs
    - Pointers
    - Allocation
  - Done Individually
  - **Will be posted after class!!**

Project 1 – penn-shell

- Project 1 (10%)
  - Unix “Shell” – command interpreter (e.g. sh, bash, etc)
  - Learn how system calls are supported and used
  - Redirection, pipelines, background/foreground processing, job control
  - Groups of two
  - Three weeks
  - Will have to demo your code at the end of the project to 1 or 2 TAs.
    In the demo you will have to answer questions about your work but also demonstrate you actually learned the content and can apply it to different (but related) scenarios.

PennOS – Projects 2 & 3

- Best way to learn about an operating systems is to build one!

- By the end of the project, you will:
  - Learn about how different subsystems in Unix interact with each other
  - Learn about priority scheduling, file systems, user shell interactions
  - Become a really good and confident systems programmer

- There is a paper on this in an ACM OS journal :
  - http://netdb.cis.upenn.edu/papers/pennos.pdf

PennOS – Projects 2 & 3

- Project 2 – penn-fat (12.5%)
  - Build a FAT file system and a stand-alone program for interacting with it
  - Groups of two
  - Three weeks

- Project 3 – penn-os (25%)
  - Build a kernel complete with a scheduler and shell and integrate it with your file system
  - Groups of two
  - Four weeks

PennOS – Projects 2 & 3

- Demos with the TA staff are done during the final week of the semester
  - Each group member will be expected to be able to answer questions about different parts of the project
  - In the demo you will have to answer questions about your work but also demonstrate you actually learned the content and can apply it to different (but related) scenarios.

- **Group evaluation done by the end of semester**
  - **Team members with lower than 15% contribution to the group will get their course grade downgraded**
  - **Team members who do almost nothing will get a failing grade in the course**

Project Policies

- **Students who did not contribute to group projects will get F grade regardless of overall score.**

- Late Policy
  - Due dates given on the schedule are the expected date of completion
  - All students get 5 late tokens to use on project
  - Can use at max two tokens per project, each token is 24 hours late
  - No submissions will be accepted after the 48-hour window has passed, baring extenuating circumstances

Collaboration Policy Violation

- Generative AI
  - I am skeptical of its usefulness for your learning and for your success in the course
  - Not banned, but I strongly do not recommended. Use your best judgement

- You will not help your overall grade and happiness:
  - Quizzed individually during project demos, exams on project in finals
  - If you can't explain your code in OH, we can turn you away
    - This is different than being confused on a bug or with C, this is ok
  - Personal lifelong satisfaction from completing PennOS

Course Grading

- **Participation & Engagement (5%)**
  - Lecture Polls 2.5%
  - Recitation Attendance: 2.5%

- **Projects (50%)**
  - Project 0 **penn-vector**: 2,5%
  - Project 1 **penn-shell**: 10%
  - Project 2 **penn-fat**: 12.5%
  - Project 3 **PennOS**: 25%

- **Exams (45%)**
  - Midterm I Exam: 5%
  - Midterm II Exam: 15%

Course Infrastructure

- Course Website: www.engineering.upenn.edu/~cis5480/current/
- Docker or Speclab
- Gradescope
- Poll Everywhere
- Ed Discussion

Getting Help

- Ed
  - Announcements will be made through here
    - When you show up and the lecture hall is empty, go to Ed to find out why…
  - Ask and answer questions

- Office Hours:
  - Can be found on calendar on front page of course website

- 1-on-1's:
  - Can schedule 1-on-1's with Travis (or TA's once we set that up)
  - OH and Ed are "first stop", but this is an option for when OH and Ed can't meet your needs or if you need more nuanced help

pollev.com/tqm

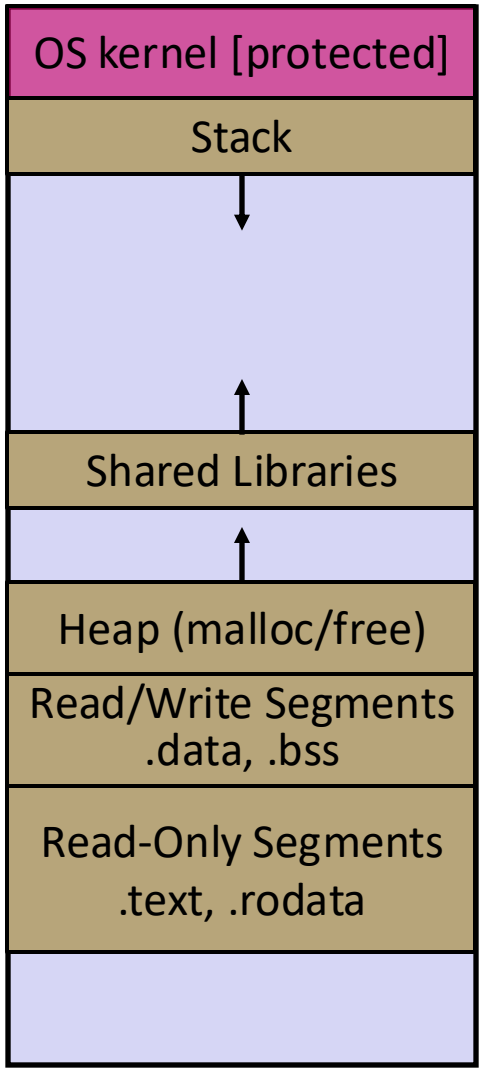
- Any questions, comments or concerns so far?

Lecture Outline

- Introduction & Logistics
  - Course Overview
  - Assignments & Exams
  - Policies
- **C "Refresher"**
  - **memory**
  - **Pointers**
    - **Output Parameters**
  - Arrays
  - Structs

Memory

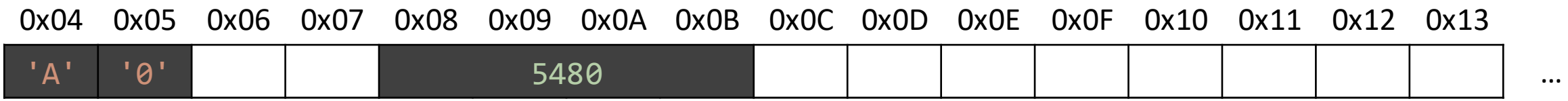
- Where all data, code, etc are stored for a program
- Broken up into several segments:
  - The stack
  - The heap
  - The kernel
  - Etc.
- Each “unit” of memory has an address



Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits
- Memory is a giant array of bytes where everything* is stored
  - Each **byte** has its own address (“index”)
- Some types take up one byte, others more

```
int main() {
  char c = 'A';
  char other = '0';
  int x = 5480;
}
```



Memory is Huge

- Modern computers are called “64-bit”
  - Addresses are 64-bits (8-bytes)
  - There are 2^{64} possible memory locations, each location is 1-byte
  - 2^{64} is **18,446,744,073,709,551,616.**
  - Pointers must be 64-bits (8-bytes) to be able to hold any address on the computer.

Pointers

POINTERS ARE EXTREMELY IMPORTANT IN C

- Variables that store addresses
  - It stores the address to somewhere in memory
  - Must specify a type so the data at that address can be interpreted
- Generic definition: `type* name;` or `type *name;` (equivalent)
  - Example: `int *ptr;`
    - Declares a variable that can contain an address
    - Trying to access that data at that address will treat the data there as an int

Pointer Operators

- *Dereference* a pointer using the unary * operator
  - Access the memory referred to by a pointer
  - Can be used to read or write the memory at the address

- Get the address of a variable with &
  - `&foo` gets the address of variable foo in memory

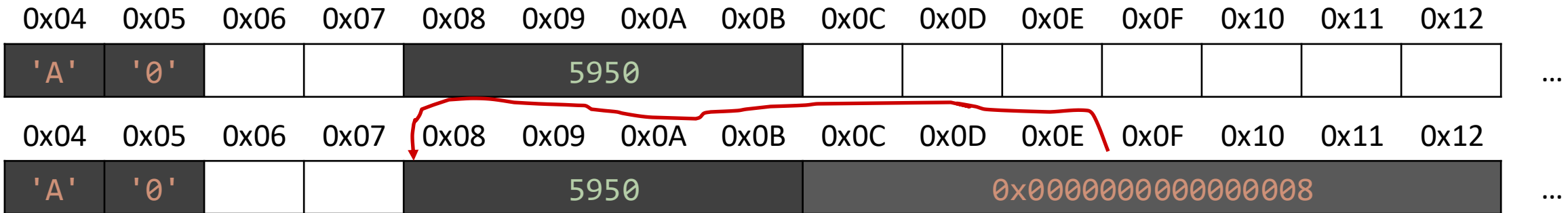
- Example:

```
int a = 5480;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Memory as an array of bytes

- Everything in memory is made of bits and bytes
  - Bits: a single 1 or 0
  - Byte: 8 bits
- Memory is a giant array of bytes where everything* is stored
  - Each byte has its own address (“index”)
- Some types take up one byte, others more

```
int main() {
  char c = 'A';
  char other = '0';
  int x = 5950;
  int* ptr = &x;
}
```



Pointers as References

- The exact value stored in a pointer *almost* never matters, we treat them more like references
- In this class we will never hardcode in an address into a pointer. We will never do something like:

```
int *ptr = 0x7fffff5194;
```

  - Read as: "**ptr** contains the address **0x7fffff5194**"
  - *with the exception of NULL
- Instead, we write code that is more often like:

```
int example = 5;
int *ptr = &example;
```

  - Read as: "**ptr** refers to the integer **example**"
  - Or "**ptr** contains the address of the integer **example**"

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

```c
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

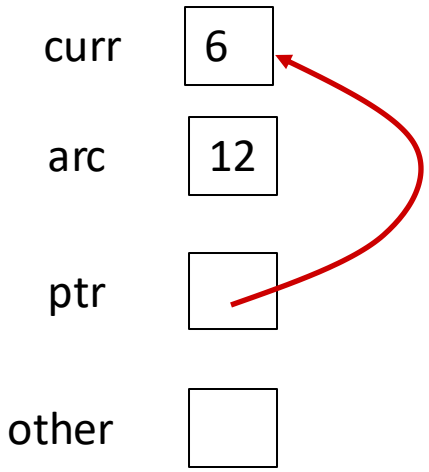
curr	6
arc	12
ptr	
other	

```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

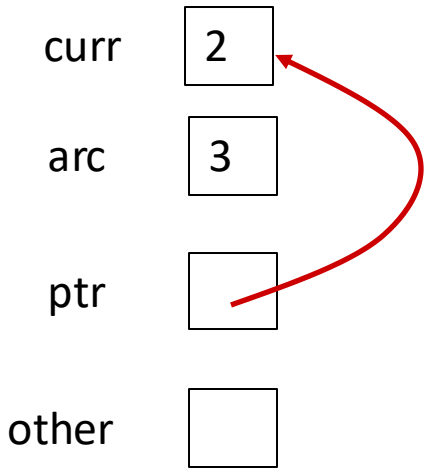



```c
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

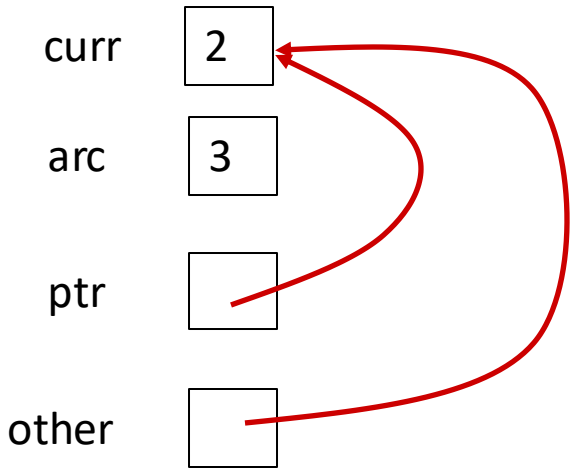



```c
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

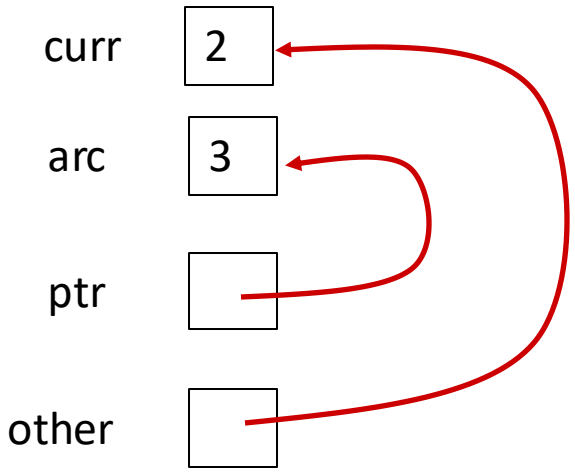



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

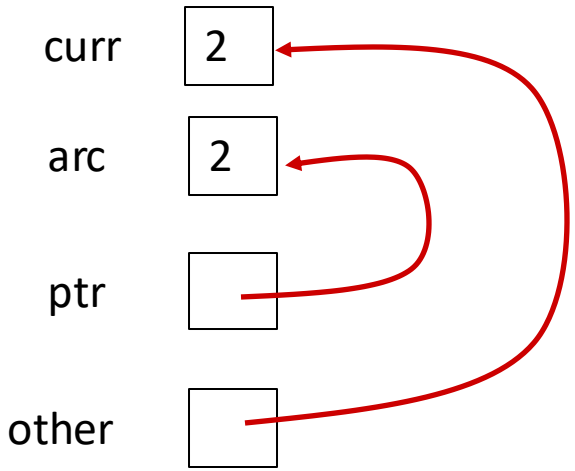



```c
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!

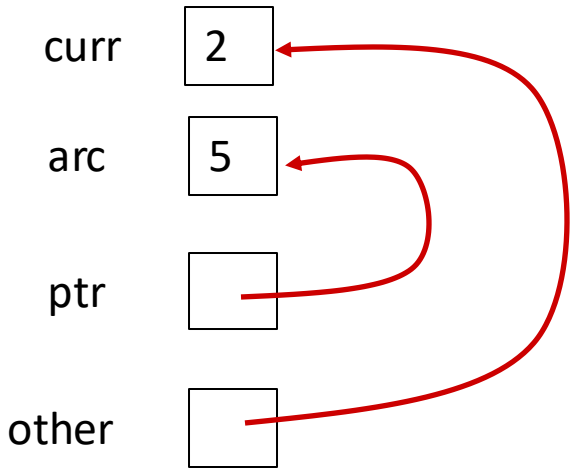



```
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Poll Everywhere

pollev.com/tqm

- What does this print?
  - You can assume this compiles and the print syntax is correct.
  - Try drawing with boxes and arrows!




```c
int main() {
  int curr = 6;
  int arc = 12;

  int* ptr = &curr;
  *ptr = 2;
  arc = 3;

  int* other = ptr;
  ptr = &arc;
  *ptr = *other
  *ptr += 3;

  // print curr and arc
  printf("%d\n", curr);
  printf("%d\n", arc);
}
```

Aside: `NULL` / `nullptr`

- `NULL` or `nullptr` is an address that is guaranteed to be invalid
  - In C on Linux, `NULL` / `nullptr` is `0x0` and an attempt to dereference `NULL` *causes a segmentation fault*
  - **In C++ (and modern C) we use `nullptr` instead of `NULL` (both are fine in this class)**

- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
  - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {
  int* p = nullptr;
  *p = 1;  // causes a segmentation fault
  return EXIT_SUCCESS;
}
```

Structured Data

- A `struct` is a C datatype that contains a set of fields
  - Similar to a Java class, but with no methods or constructors or really much else…
  - Useful for defining new structured types of data
  - Acts similarly to primitive variables
- Generic declaration:

```c
// declaring the struct type
struct point {
  float x;
  float y;
};

// declaring a variable
struct point pt;
```

```c
// declaring the struct type
typedef struct point_st {
  float x;
  float y;
} point;

// declaring a variable
point pt;
```

Structured Data Initialization

- A `struct` is a C datatype that contains a set of fields
  - Acts similarly to primitive variables
- Generic declaration:

```
typedef struct point_st {
  float x;
  float y;
} point;

point pt;
point origin = {0.0f, 0.0f};
point other = (point) {
  .x = 3.14f,
  .y = 3.800f,
};

pt = origin; // pt now contains 0.0f, 0.0f
```

Default values are still garbage!

<- Initializer List

<- with designators

^ same as `pt.x = origin.x;` `pt.y = origin.y;`

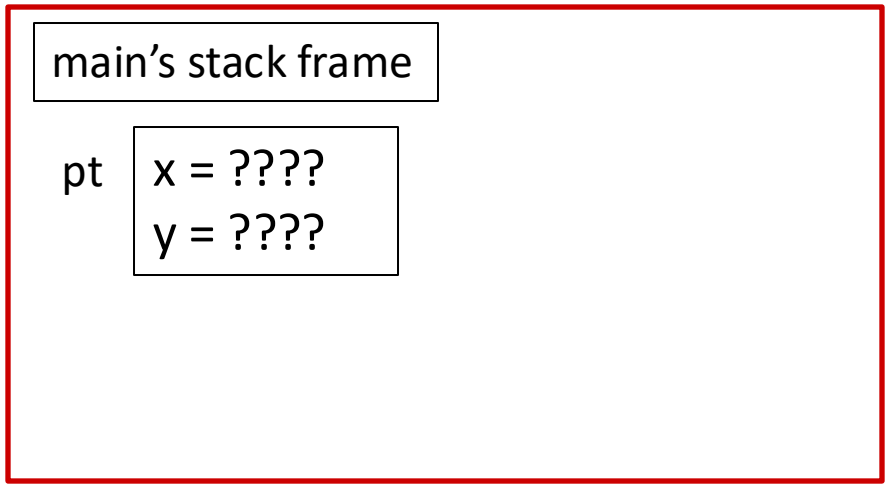
Accessing struct Fields

- Use "." to refer to a field in a struct
- Use "->" to refer to a field from a struct *pointer*
  - Dereferences pointer first, then accesses field

```
typedef struct point_st {
  float x, y;
} Point;

int main(int argc, char** argv) {
  Point p1 = {0.0, 0.0};
  Point* p1_ptr = &p1;

  p1.x = 1.0;
  p1_ptr->y = 2.0;  // equivalent to (*p1_ptr).y = 2.0;
  return 0;
}
```

Structs: Copied not Referenced

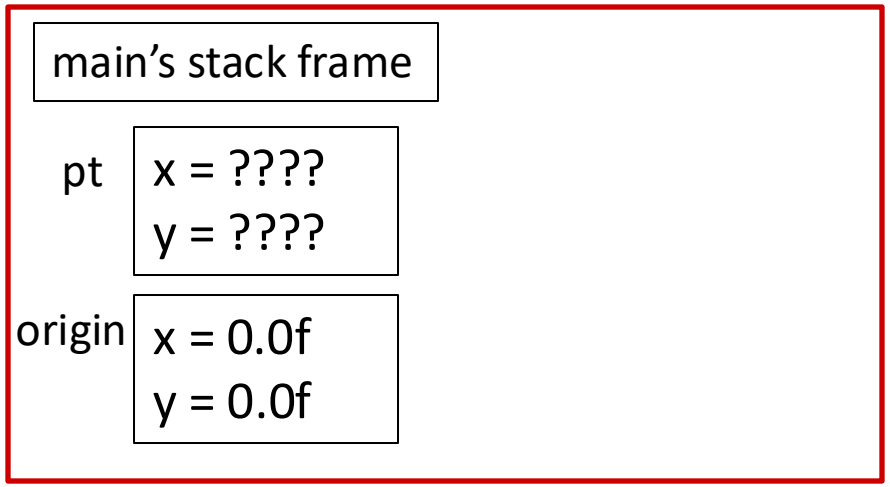
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

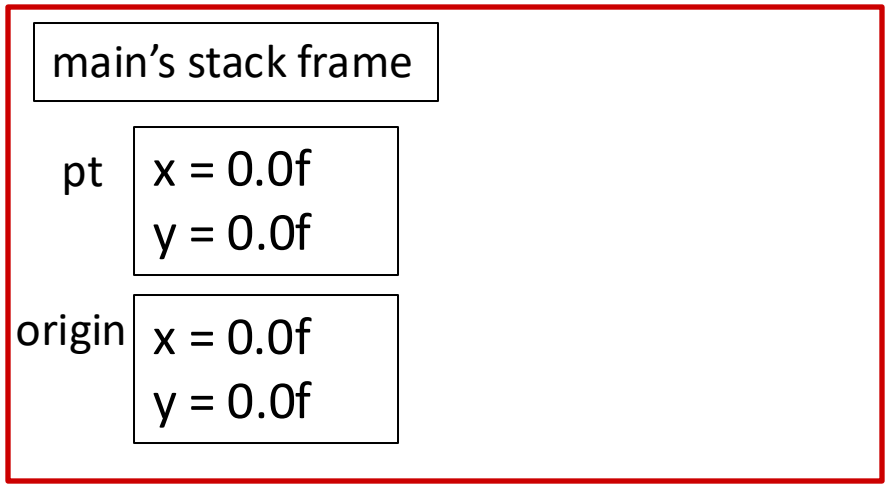
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

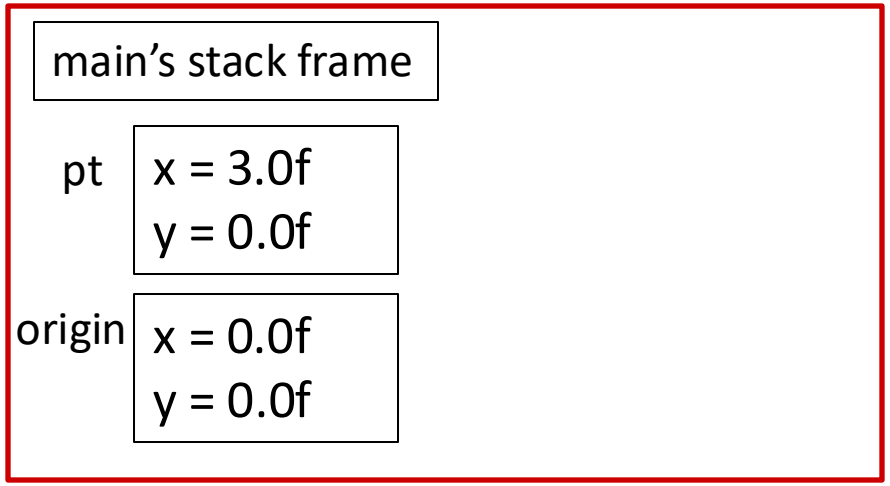
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

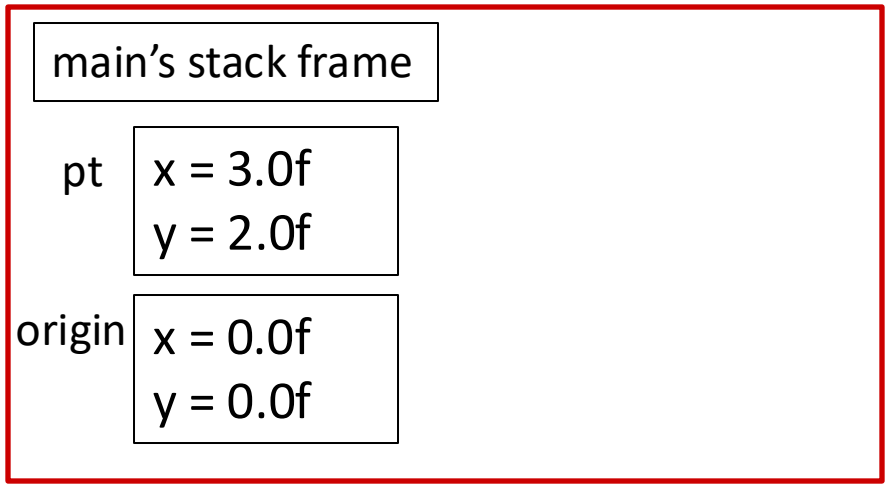
- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Structs: Copied not Referenced

- When we have two struct variables, we have two structs.
  - Objects in languages like Java or Python are references




```c
typedef struct point_st {
  float x;
  float y;
} Point;

int main() {
  Point pt;
  Point origin = {0.0f, 0.0f};
  pt = origin; // pt now contains 0.0f, 0.0f

  pt.x = 3.0f;
  pt.y = 2.0f;
}
```

Raise Hands

- What does this code print?

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

void modify_int(int x) {
  x = 5;
}

int main() {
  int num = 3;
  modify_int(num);
  printf("%d\n", num);
  return EXIT_SUCCESS;
}
```

Raise Hands

- What does this code print?

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

typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point p) {
  p.x = 3800;
  p.y = 4710;
}

int main() {
  Point p = {1100, 2400};
  modify_point(p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Raise Hands

- What does this code print?

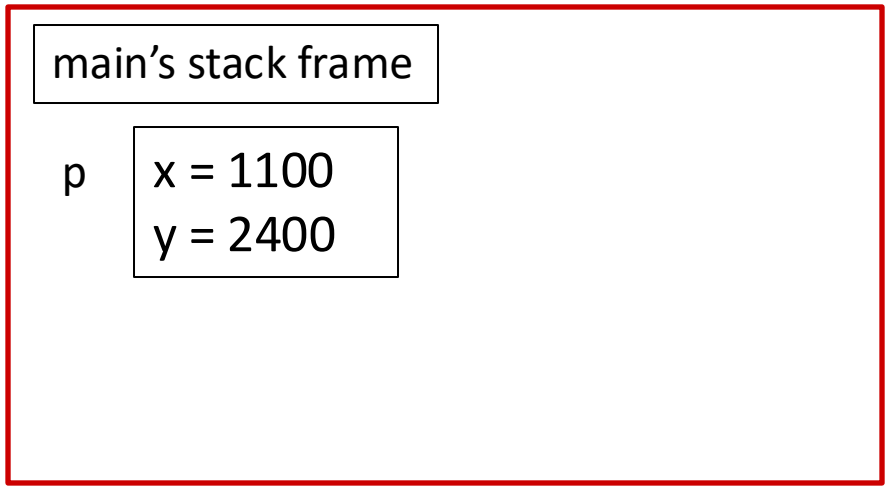
- How could we fix it?
  E.g. make modify_point *actually* modify a point

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

typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point p) {
  p.x = 3800;
  p.y = 4710;
}

int main() {
  Point p = {1100, 2400};
  modify_point(p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Demo: pass_by.c

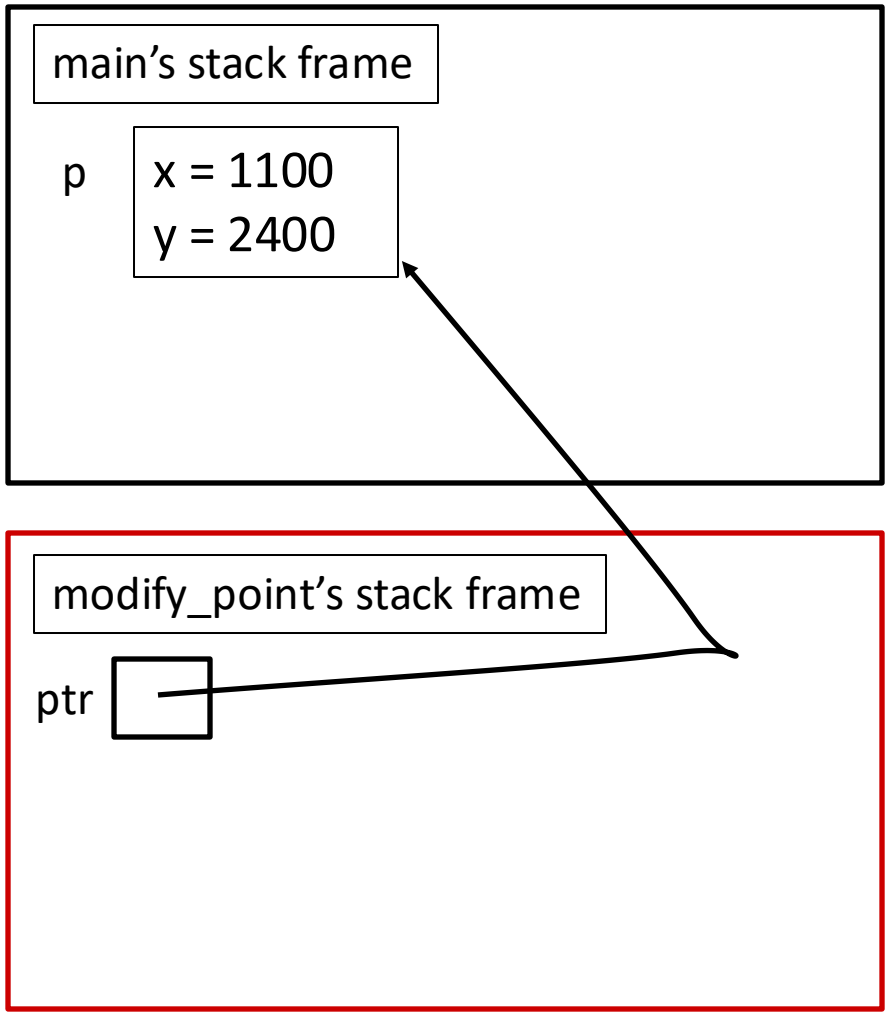
- Everything in C is pass-by value (e.g. a copy is passed to the function)
- HOWEVER, we can pass a copy of a pointer (e.g. a reference to something) to mimic pass-by-reference.
- Demo pass_by.c
  - Note: most lecture code will be available on the course website

Visualization: faulty pass by reference



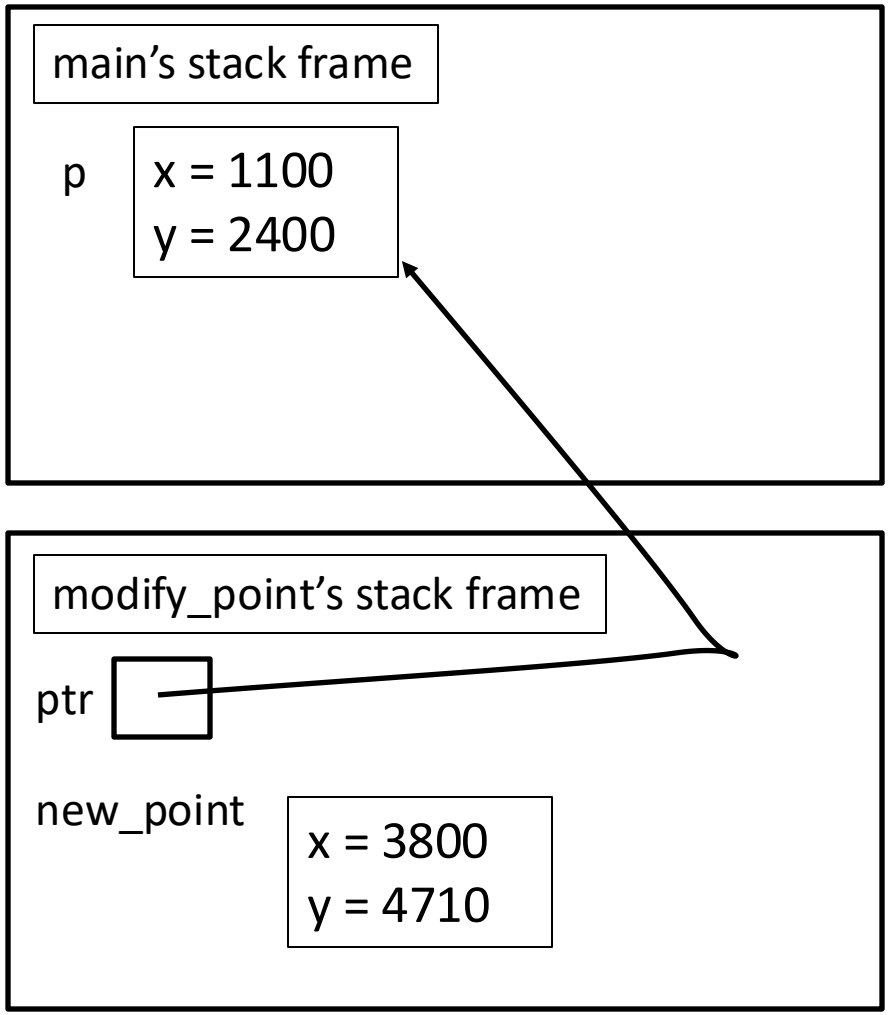

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference



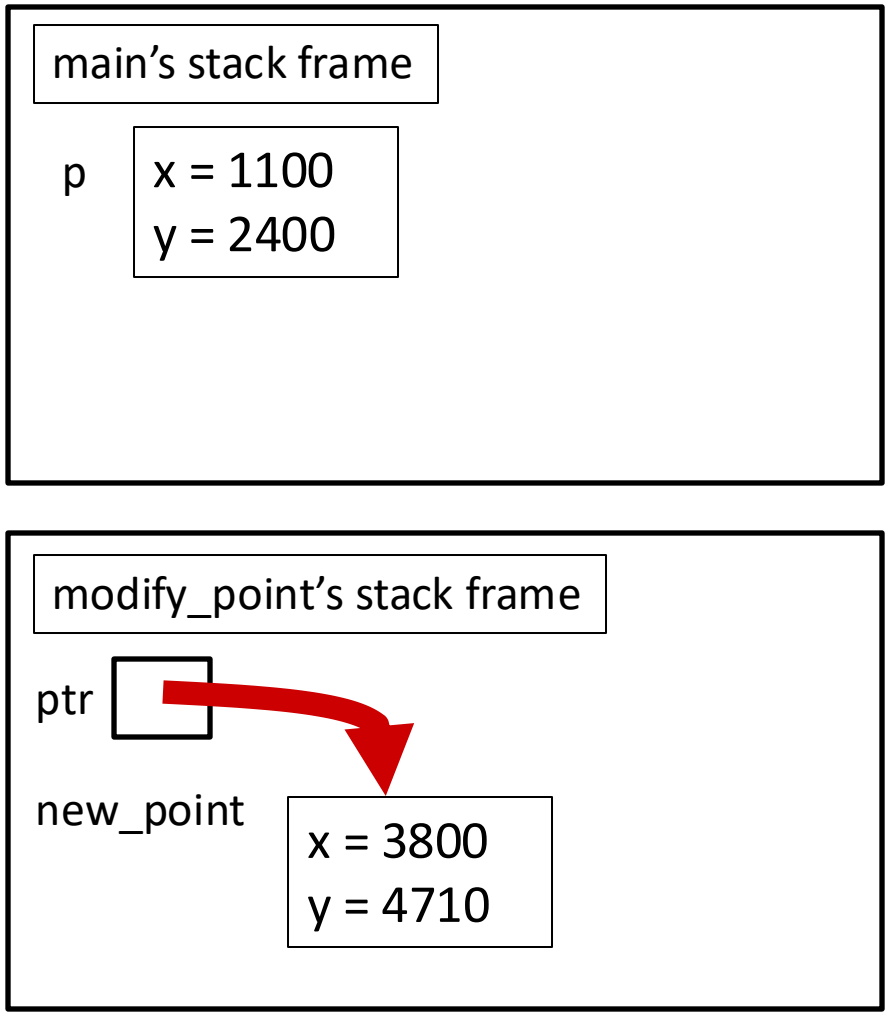

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference



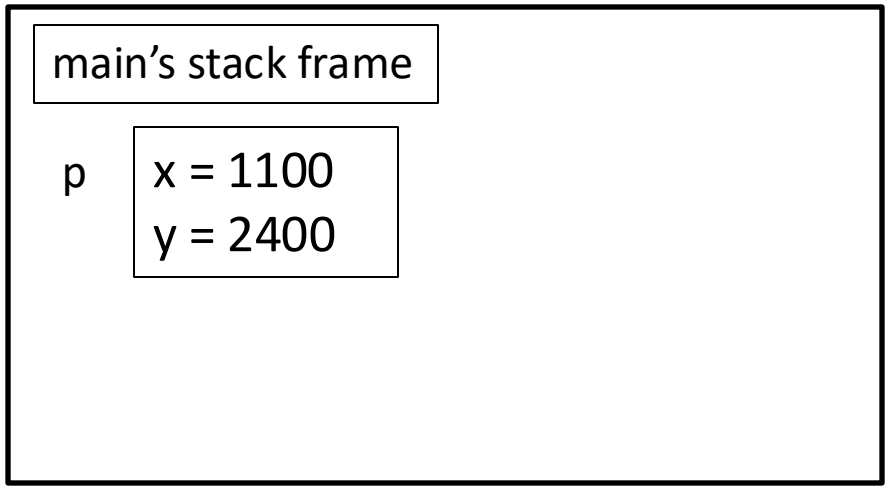

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: faulty pass by reference



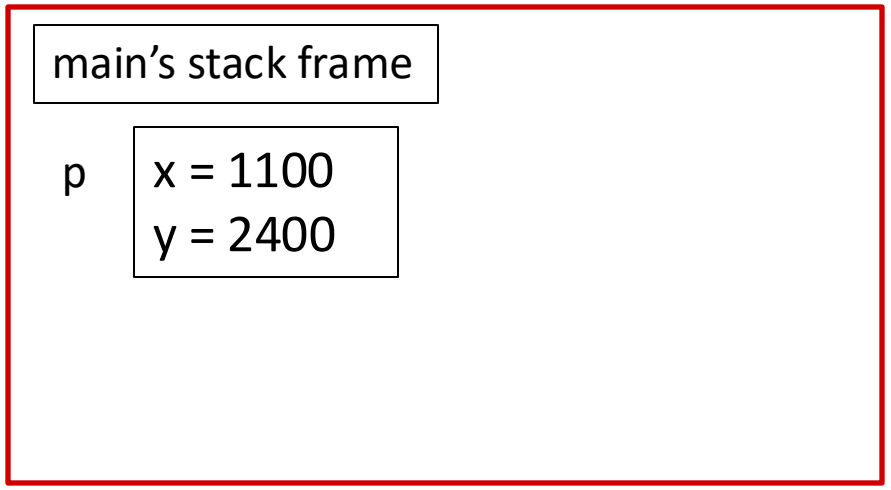

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  ptr = &new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)

Buggy version said:
`ptr = &new_point`

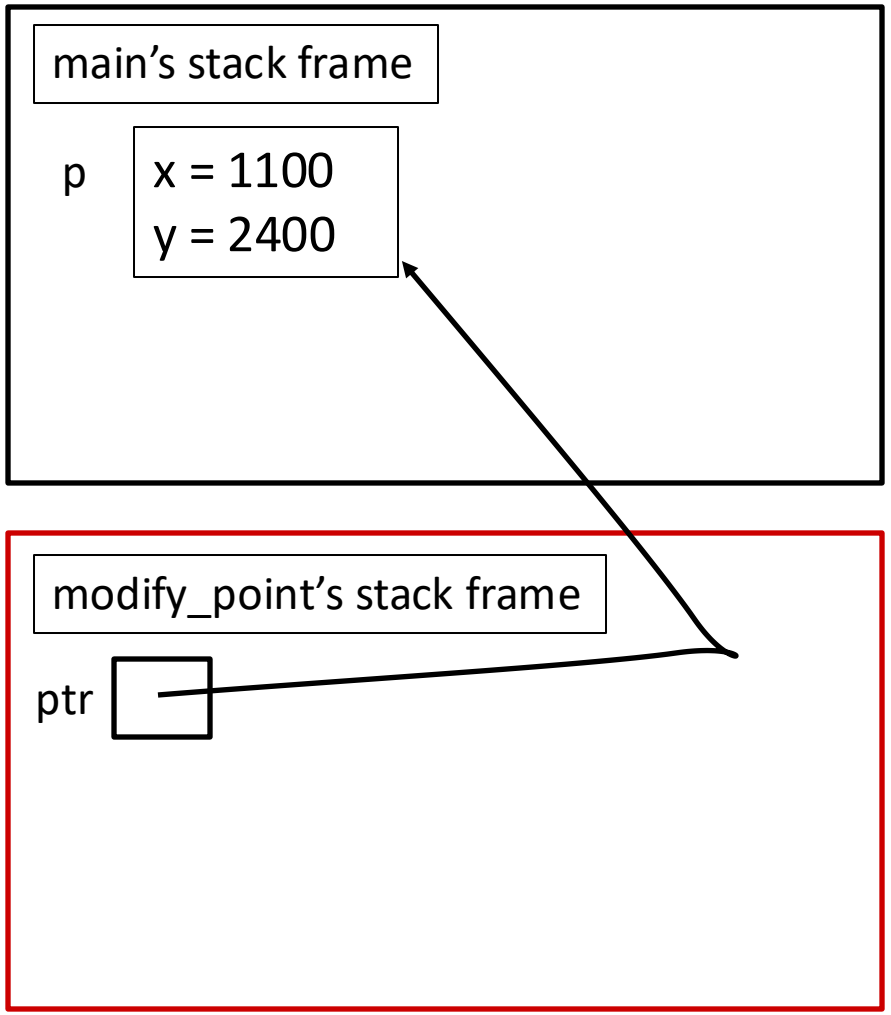
```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



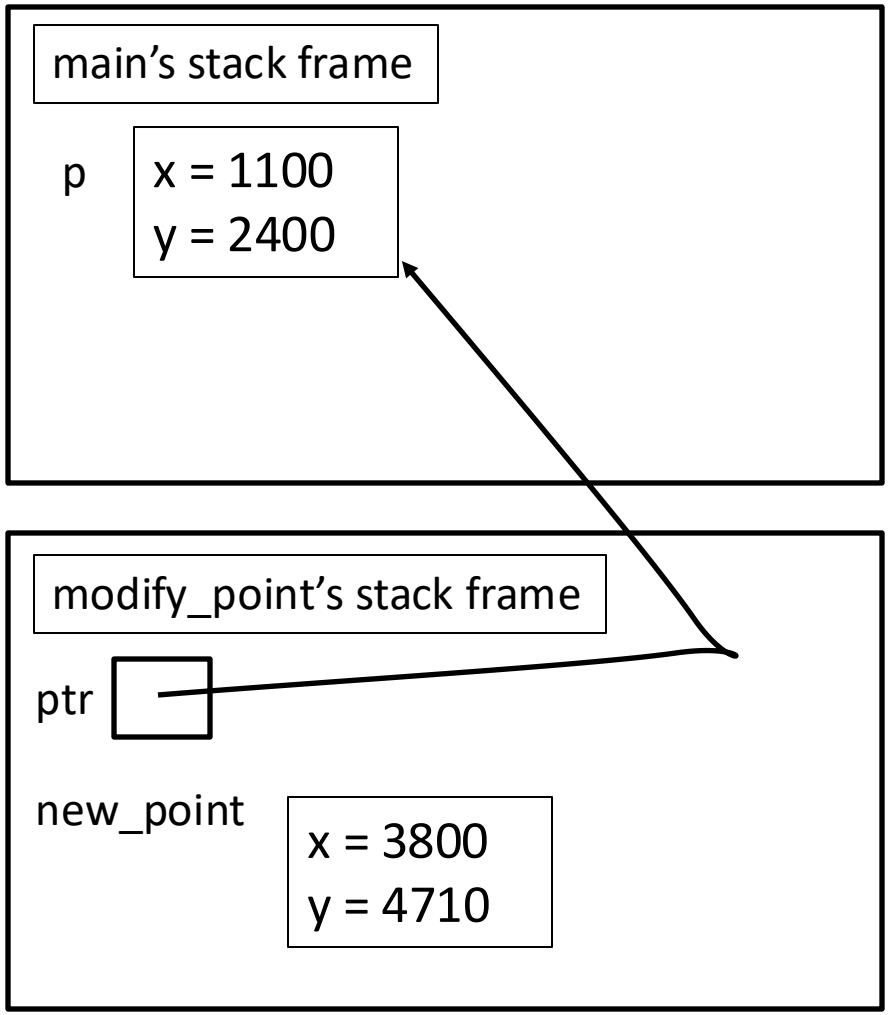

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



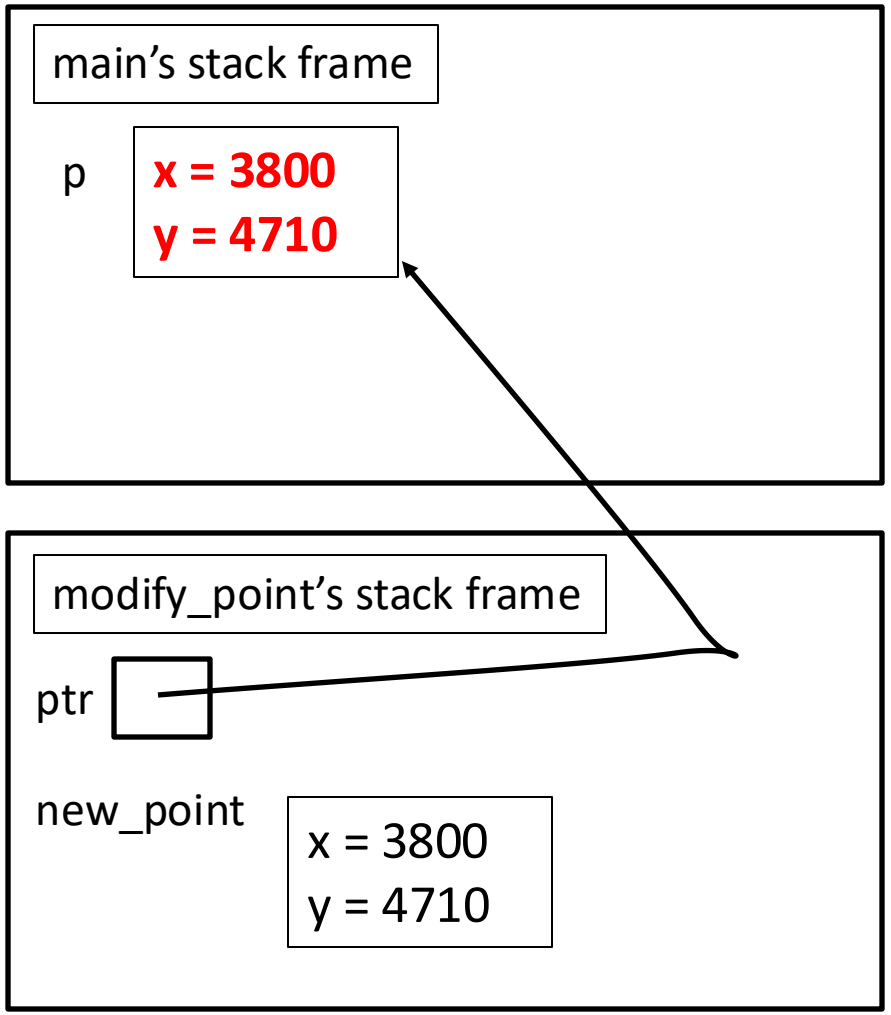

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)



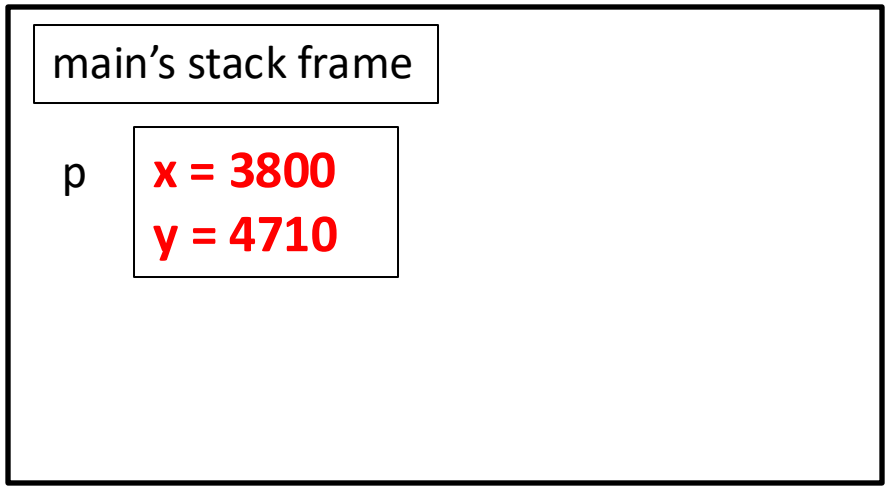

```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Visualization: fixed pass by reference (Output Parameters)




```c
typedef struct point_st {
  int x;
  int y;
} Point;

void modify_point(Point* ptr) {
  Point new_point = (Point) {
    .x = 3800,
    .y = 4710,
  };
  *ptr = new_point;
}

int main() {
  Point p = {1100, 2400};
  modify_point(&p);
  printf("%d, %d\n", p.x, p.y);
  return EXIT_SUCCESS;
}
```

Lecture Outline

- Introduction & Logistics
  - Course Overview
  - Assignments & Exams
  - Policies
- **C "Refresher"**
  - memory
  - Pointers
    - Output Parameters
  - **Arrays**
  - **Strings**
  - **Structs**

Arrays in C

- Definition: `type` `type name[size]`
  - Allocates `size` x `sizeof(type)` bytes of *contiguous* memory
  - Normal usage is a compile-time constant for `size`
    (*e.g.* `int scores[175];`)
  - Initially, array values are "garbage"

- Size of an array
  - **Not stored anywhere** – array does not know its own size!
  - The programmer will have to store the length in another variable or hard-code it in
  - No bounds checking!

Using Arrays

Optional when initializing

- Initialization: `type name[size] = {val0,…,valN};`
  - `{ }` initialization can *only* be used at time of definition
  - If no size supplied, infers from length of array initializer

- Array name used as identifier for "collection of data"
  - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
  - Array name (by itself) produces the address of the start of the array
    - Cannot be assigned to / changed

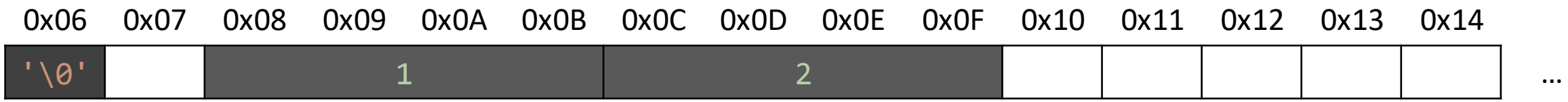
```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;   // memory smash!
```

No IndexOutOfBounds
Hope for segfault

Arrays in C

- Here is a memory diagram example:

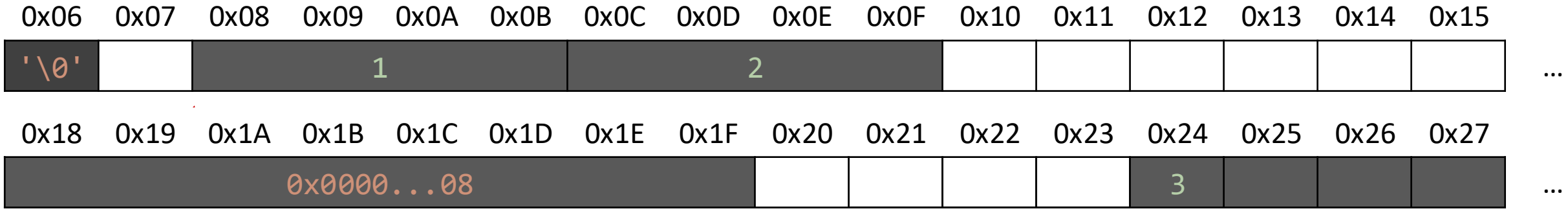
```
int main() {
  char c = '\0';

  int arr[2] = {1, 2};
}
```



Pointers as C arrays

- Pointers can be set to an array
- Pointers can always be indexed into like an array
  - Pointers don't always have to point to the beginning of an array!
- `ptr[i]` is equivalent to `*(ptr + i)`
  - Pointer arithmetic makes this work

```
int main() {
  char c = '\0';

  int arr[2] = {1, 2};

  int* ptr = arr;

  int x = ptr[1] + 1;
}
```



Poll Everywhere

pollev.com/tqm

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

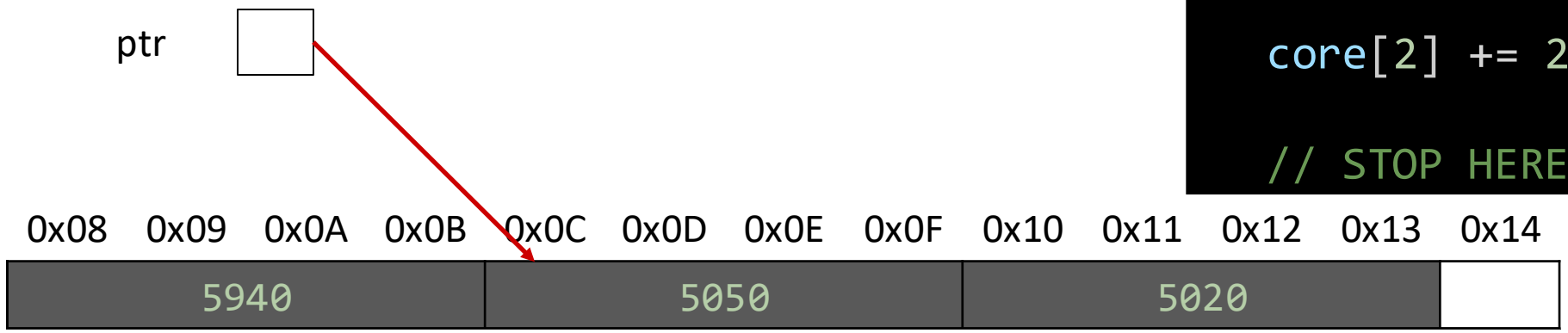
```
void foo() {
  int core[3] = {5940, 5930, 5960};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
}
```

Poll Everywhere

pollev.com/tqm

- What is the final value of core after this code is run? Where is ptr pointing to after this code is run?
  - Hint: Draw it out!

```
void foo() {
  int core[3] = {5940, 5930, 5960};

  core[1] += 20;

  int* ptr = &(core[1]);

  ptr[0] -= 900;

  ptr[1] = 5000;

  core[2] += 20;

  // STOP HERE
```



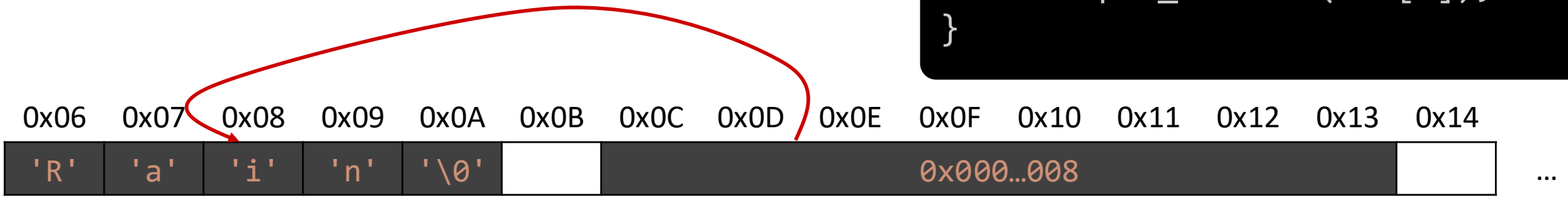
Strings in C

- Strings in C are just arrays of characters with a special character at the end to mark the end of the string: `'\0'`
  - Called the "null terminator" character

- C-strings are often referred to with a **char[]** or a **char***

- Example:
  - `print(str)  // Rain`
  - `print(ptr_str)  // in`

```
int main() {
  char c = '\0';

  char str[5] = "Rain";

  char* ptr_str = &(str[2]);
}
```



Poll Everywhere

pollev.com/tqm

- Finish this code:
  - This function takes in a string and returns the length of the string.
  - Do not call any other function
  - size_t is just an unsigned integer type
  - Remember to index into the pointer like an array!
  - What marks the end of a string?
  - You don't have to use a while loop, but I think it makes the most sense.

```
size_t strlen(char* str) {
  size_t length = 0;









  return length;
}
```

Poll Everywhere

pollev.com/tqm

- Finish this code:
  - This function takes in a string and returns the length of the string.
  - Do not call any other function
  - size_t is just an unsigned integer type
  - Remember to index into the pointer like an array!
  - What marks the end of a string?
  - You don't have to use a while loop, but I think it makes the most sense.

```
size_t strlen(char* str) {
  size_t length = 0;

  while (str[length] != '\0') {

    length += 1;

  }


  return length;
}
```

Multi-dimensional Arrays

- Generic 2D format:

```
type name[rows][cols];
```

  - Still allocates a single, contiguous chunk of memory
  - C is *row-major*
  - Can access elements with multiple indices
    - `A[0][1] = 7;`
    - `my_int = A[1][2];`
  - The entries in this array are stored in memory in row major order as follows:
    - A[0][0], A[0][1], A[0][2], A[1][0], A[1][1], A[1][2]
  - 2-D arrays normally only useful if size known in advance. Otherwise use dynamically-allocated data and pointers (later)

Arrays as Parameters

- It's tricky to use arrays as parameters
  - What happens when you use an array name as an argument? Passes in address of start of array
    - It "decays" into a pointer
  - Pointers (like arrays) do not know their length

```
int sumAll(int a[]) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

Equivalent

```
int sumAll(int* a) {
  int i, sum = 0;
  for (i = 0; i < ...???
}
```

- Note: Array syntax works on pointers
  - E.g. `ptr[3] = ...;`

Solution: Pass Size as Parameter

```
int sumAll(int a[], int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += a[i];
  }
  return sum;
}
```

- Standard idiom in C programs

Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

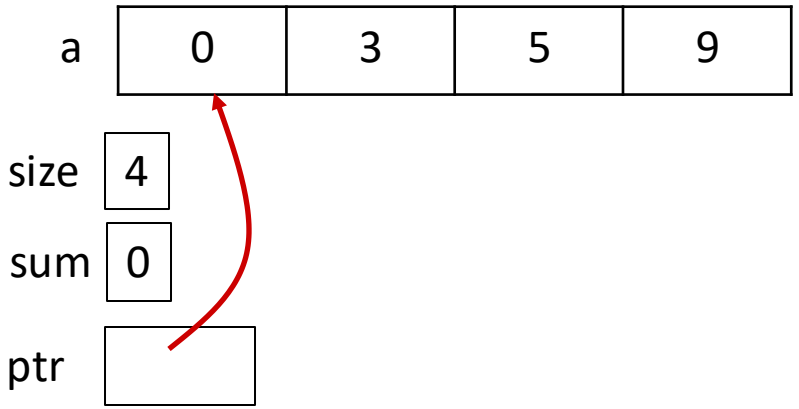
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
for (int i = 0; i < size; i++) {
  sum += ptr[i];
}
```

```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```

Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

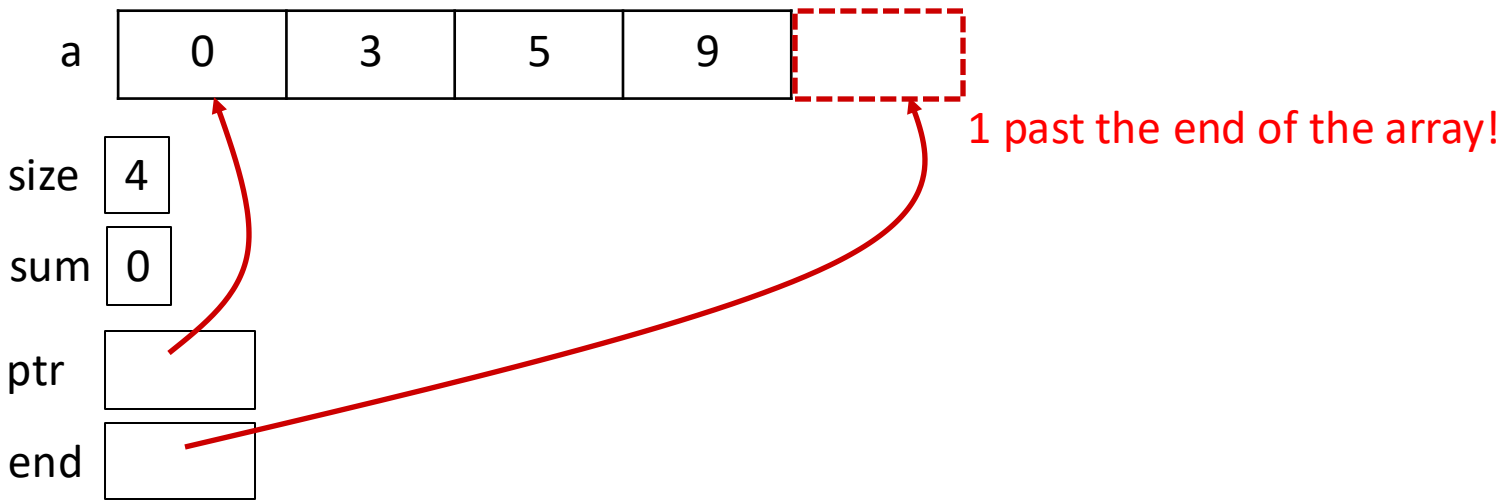
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

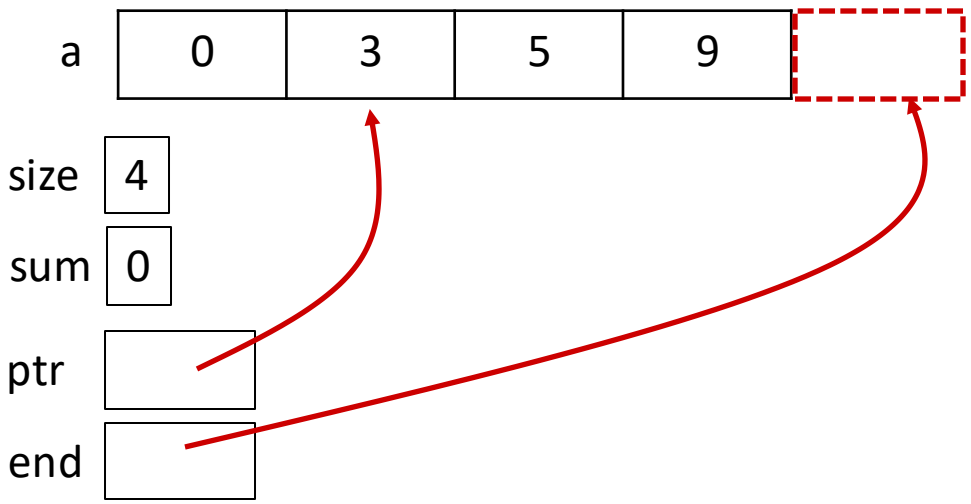
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

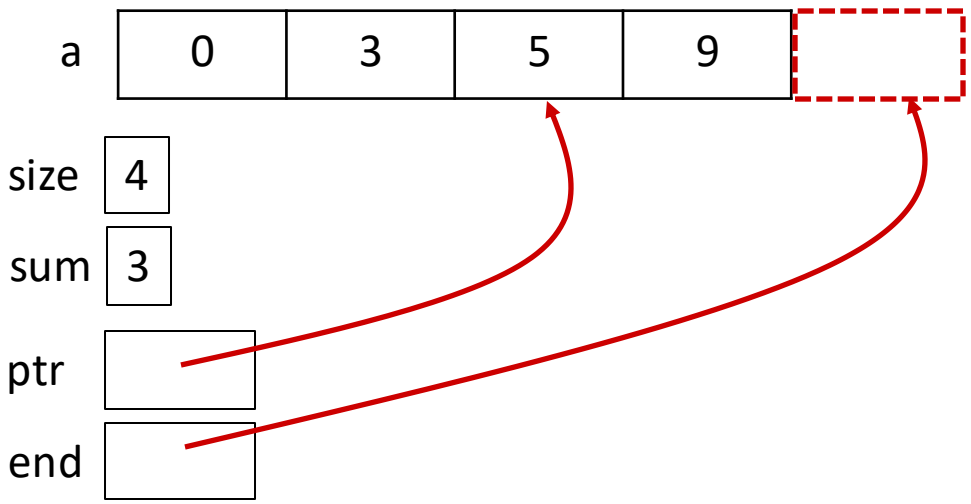
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

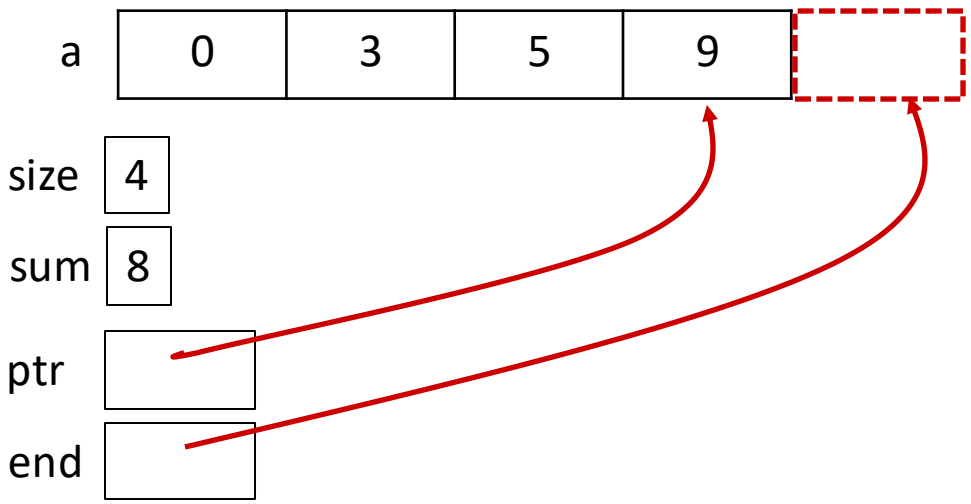
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

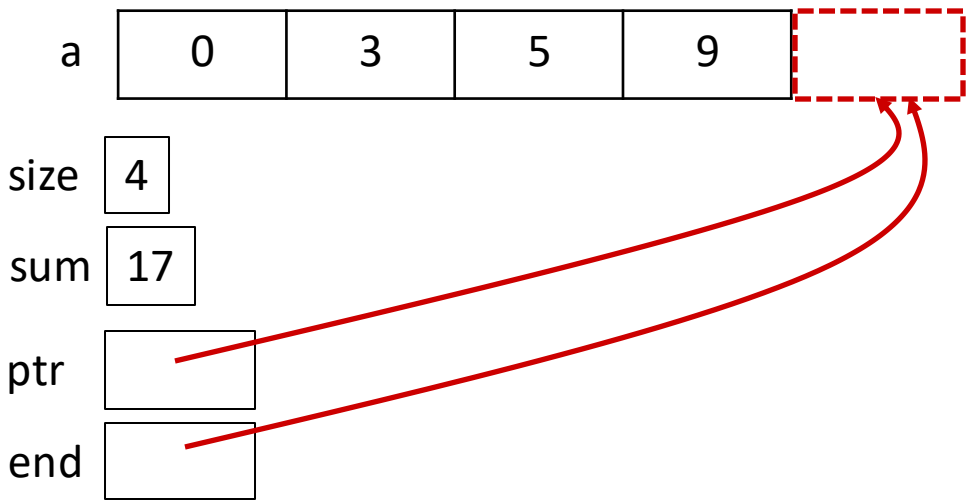
```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.

```
int a[] = {0, 3, 5, 9};
int size = 4;

int sum = 0;
int* ptr = a; // &(a[0])
int* end = ptr + size;
for (; ptr != end; ptr++) {
  sum += *ptr;
}
```



That's all for now!

- If we got through all this, good job!!!
- You should have everything you need for the first homework assignment after next lecture (The Heap, Malloc and Free). If you want to get started now, we put some of the slides on malloc and free after this slide.

- We are going a little fast because we expect you have already seen all or most of this before!
- When we get to new material it usually won't be as fast

- Releasing today or tomorrow:
  - Project 0 – penn-vector
  - Pre-Semester Survey

Demo: get_input.c

- Lets code together a small program that:
  - Reads at max 100 characters from stdin (user input)
  - Truncates the input to only the first word
  - Prints that word out
  - Not allowed to use scanf, FILE*, printf, etc

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function signature

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?

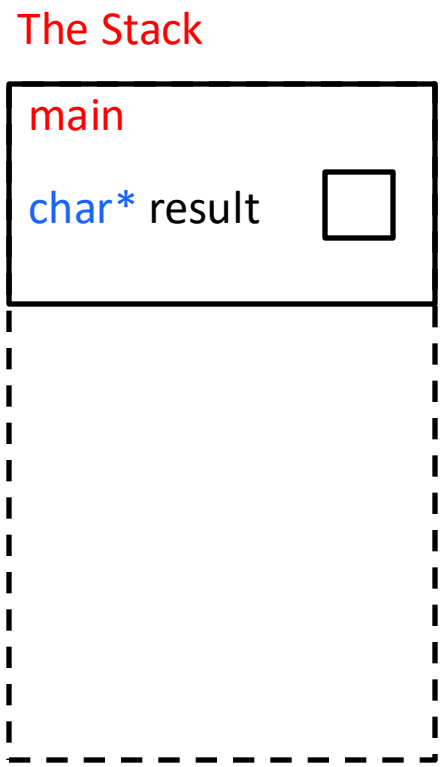
```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO,
                     str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

```c
// assuming this is how the function is called
char* result = read_stdin();
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?



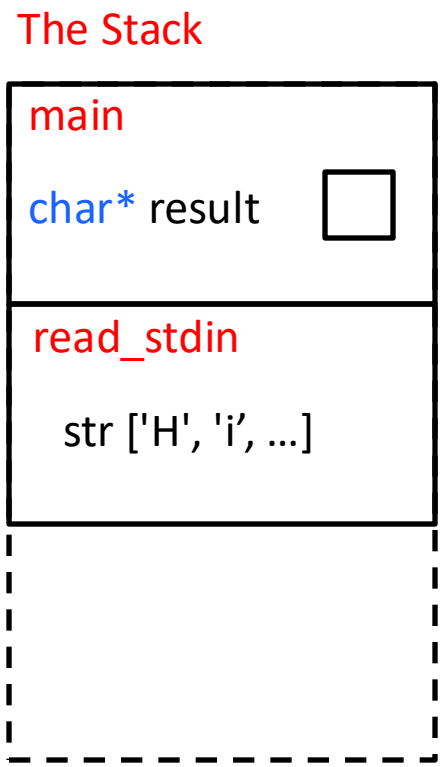

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO,
                     str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

```c
// assuming this is how the function is called
char* result = read_stdin();
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?



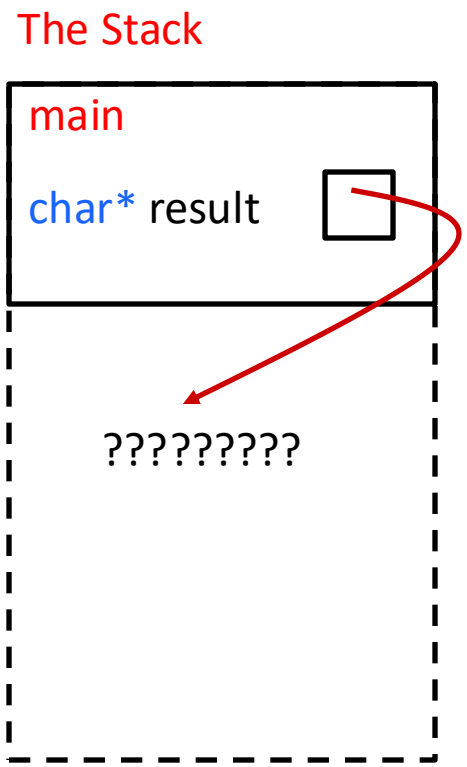

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO,
                     str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

```c
// assuming this is how the function is called
char* result = read_stdin();
```

Poll Everywhere

pollev.com/tqm

- There are two things wrong with this function
- What are they? How do we fix this function w/o changing the function sig?




```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str[MAX_INPUT_SIZE];

  ssize_t res = read(STDIN_FILENO,
                     str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

```c
// assuming this is how the function is called
char* result = read_stdin();
```

Memory Allocation

- So far, we have seen two kinds of memory allocation:

```c
int counter = 0;    // global var

int main() {
  counter++;
  printf("count = %d\n",counter);
  return 0;
}
```

- `counter` is statically-allocated
  - Allocated when program is loaded
  - Deallocated when program exits

```c
int foo(int a) {
  int x = a + 1;      // local var
  return x;
}

int main() {
  int y = foo(10);    // local var
  printf("y = %d\n",y);
  return 0;
}
```

- `a`, `x`, `y` are automatically-allocated
  - Allocated when function is called
  - Deallocated when function returns

Aside: `sizeof`

- **`sizeof`** operator can be applied to a variable or a type and it evaluates to the size of that type in bytes
- Examples:
  - **`sizeof(int)`** – returns the size of an integer
  - **`sizeof(double)`** – returns the size of a double precision number
  - **`struct my_struct s;`**
    - **`sizeof(s)`** – returns the size of the struct s
  - **`my_type *ptr`**
    - **`sizeof (*ptr)`** – returns the size of the type pointed to by ptr

- Very useful for Dynamic Memory

What is Dynamic Memory Allocation?

- We want Dynamic Memory Allocation
  - Dynamic means “at run-time”
  - The compiler and the programmer don’t have enough information to make a final decision on how much to allocate
  - Your program explicitly requests more memory at run time
  - The language allocates it at runtime, maybe with help of the OS

- Dynamically allocated memory persists until either:
  - A garbage collector collects it (automatic memory management)
  - Your code explicitly deallocates it (manual memory management)

- C requires you to manually manage memory
  - More control, and more headaches

Heap API

- Dynamic memory is managed in a location in memory called the "Heap"
  - The heap is managed by user-level runetime library (libc)
  - Interface functions found in `<stdlib.h>`
- Most used functions:
  - **`void *malloc(size_t size);`**
    - Allocates memory of specified size
  - **`void free(void *ptr);`**
    - Deallocates memory
- Note: **`void*`** is “generic pointer”. It holds an address, but doesn’t specify what it is pointing at.
- Note 2: **`size_t`** is the integer type of **`sizeof()`**

malloc()

- `void *malloc(size_t size);`
- **malloc** allocates a block of memory of the requested size
  - Returns a pointer to the first byte of that memory
    - And returns NULL if the memory allocation failed!
  - You should assume that the memory initially contains garbage
  - You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array
float* arr = malloc(10*sizeof(float));
if (arr == NULL) {
  return errcode;
}
...   // do stuff with arr
```

ALWAYS CHECK FOR NULL

free()

- Usage: `free(pointer);`
- Deallocates the memory pointed-to by the pointer
  - Pointer *must* point to the first byte of heap-allocated memory (*i.e.* something previously returned by **malloc**)
  - Freed memory becomes eligible for future allocation
  - `free(NULL);` does nothing.
  - The bits in the pointer are *not changed* by calling free
    - Defensive programming: can set pointer to NULL after freeing it

```
float* arr = malloc(10*sizeof(float));
if (arr == NULL)
  return errcode;
...             // do stuff with arr
free(arr);
arr = NULL;     // OPTIONAL
```

The Heap

- The Heap is a large pool of available memory to use for Dynamic allocation
- This pool of memory is kept track of with a small data structure indicating which portions have been allocated, and which portions are currently available.
- **`malloc`**:
  - searches for a large enough unused block of memory
  - marks the memory as allocated.
  - Returns a pointer to the beginning of that memory
- **`free`**:
  - Takes in a pointer to a previously allocated address
  - Marks the memory as free to use.

Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...             // do stuff with ptr
  free(ptr);
}
```

addr	var	value
0x2001	**ptr**	--
...	...	--
0x4000	**HEAP START**	USED
0x4001		USED
0x4002		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...             // do stuff with ptr
  free(ptr);
}
```

addr	var	value
0x2001	**ptr**	**0x4002**
...	...	--
0x4000	**HEAP START**	USED
0x4001		USED
0x4002		**USED**
0x4003		**USED**
0x4004		**USED**
0x4005		**USED**
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Dynamic Memory Example

```c
#include <stdlib.h>

int main() {
  char* ptr = malloc(4*sizeof(char));
  if (ptr == NULL)
    return EXIT_FAILURE;
  ...              // do stuff with ptr
  free(ptr);
}
```


addr	var	value
0x2001	**ptr**	**0x4002**
...	...	--
0x4000	**HEAP START**	USED
0x4001		USED
0x4002		
0x4003		
0x4004		
0x4005		
0x4006		
0x4007		
0x4008		USED
0x4009		USED

Fixed read_stdin()

```c
#define MAX_INPUT_SIZE 100

char* read_stdin() {
  char str = (char*) malloc(sizeof(char) * MAX_INPUT_SIZE);
  if (str == NULL) {
    return NULL;
  }

  ssize_t res = read(STDIN_FILENO, str, MAX_INPUT_SIZE);

  // error checking
  if (res <= 0) {
    return NULL;
  }

  return str;
}
```

Demo (continued): get_input.c

- Lets code together a small program that:
  - Reads at max 100 characters from stdin (user input)
  - Truncates the input to only the first word
  - Prints that word out
  - Not allowed to use scanf, FILE*, printf, etc

- What was the other issue? (other than not using malloc)

Dynamic Memory Pitfalls

- Buffer Overflows
  - E.g. ask for 10 bytes, but write 11 bytes
  - Could overwrite information needed to manage the heap
  - Common when forgetting the null-terminator on malloc'd strings
- Not checking for **`NULL`**
  - Malloc returns NULL if out of memory
  - Should check this after every call to malloc
- Giving **`free()`** a pointer to the middle of an allocated region
  - Free won't recognize the block of memory and probably crash
- Giving free() a pointer that has already been freed
  - Will interfere with the management of the heap and likely crash
- **`malloc`** does NOT initialize memory
  - There are other functions like **`calloc`** that will zero out memory

Memory Leaks

- The most common Memory Pitfall
- What happens if we malloc something, but don't free it?
  - That block of memory cannot be reallocated, even if we don't use it anymore, until it is **`free`**d
  - If this happens enough, we run out of heap space and program may slow down and eventually crash
- Garbage Collection
  - Automatically "frees" anything once the program has lost all references to it
  - Affects performance, but avoid memory leaks
  - Java has this, C doesn't

`static` function variables

- Functions can declare a variable as static

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

int next_num();

int main(int argc, char** argv) {
  printf("%d\n", next_num()); // prints 1
  printf("%d\n", next_num()); // then 2
  printf("%d\n", next_num()); // then 3
  return EXIT_SUCCESS;
}

int next_num() {
  // marking this variable as static means that
  // the value is preserved between calls to the function
  // this allows the function to "remember" things
  static int counter = 0;
  counter++;
  return counter;
}
```

This is how some functions (like one in proj0) can "remember" things.

Can be thought of as a global variable that is "private" to a function

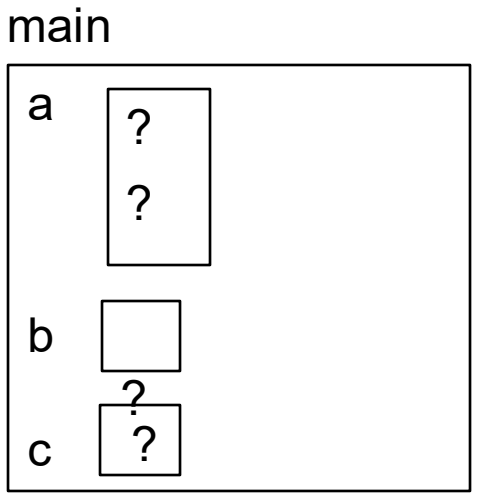
Poll Everywhere

pollev.com/tqm

- Which line below is first to (most likely) cause a crash?
  - Yes, there are a lot of bugs, but not all cause a crash ☺
  - See if you can find all the bugs!

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

  return 0;
}
```

Memory Corruption - What Happens?



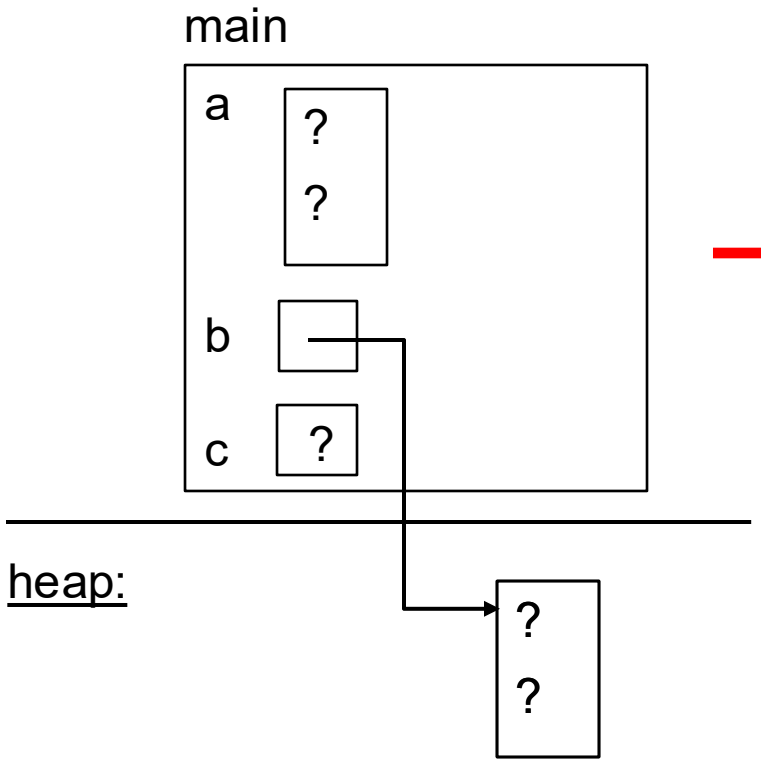
heap:

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?

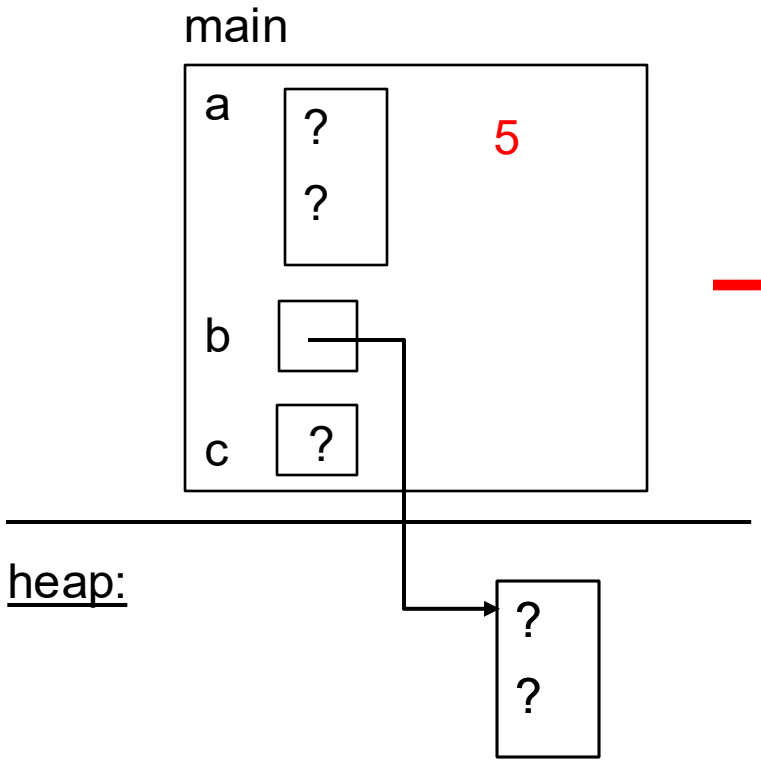



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?

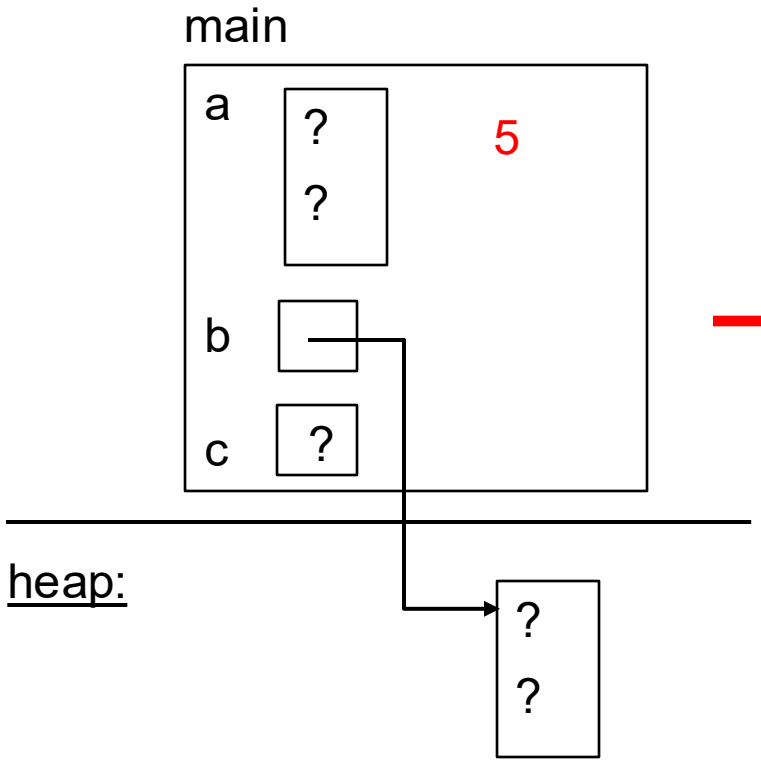



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?

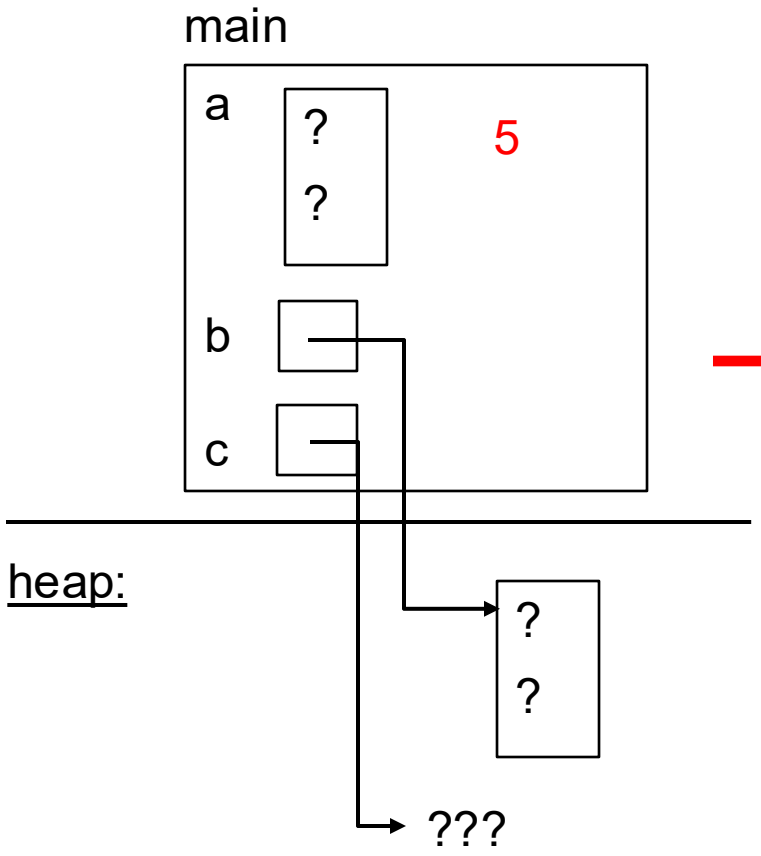



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?

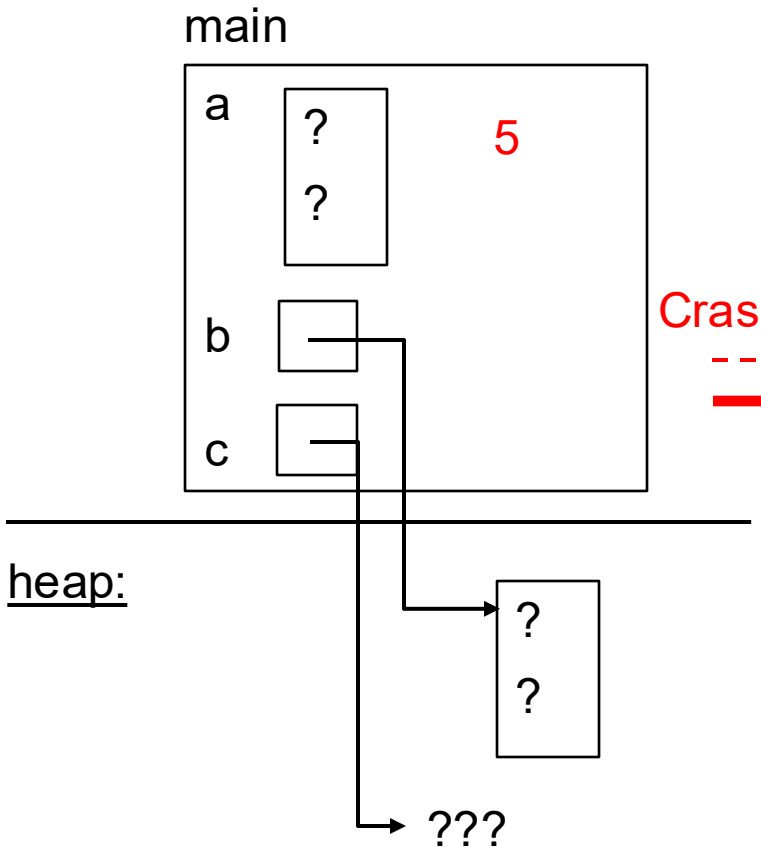



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?



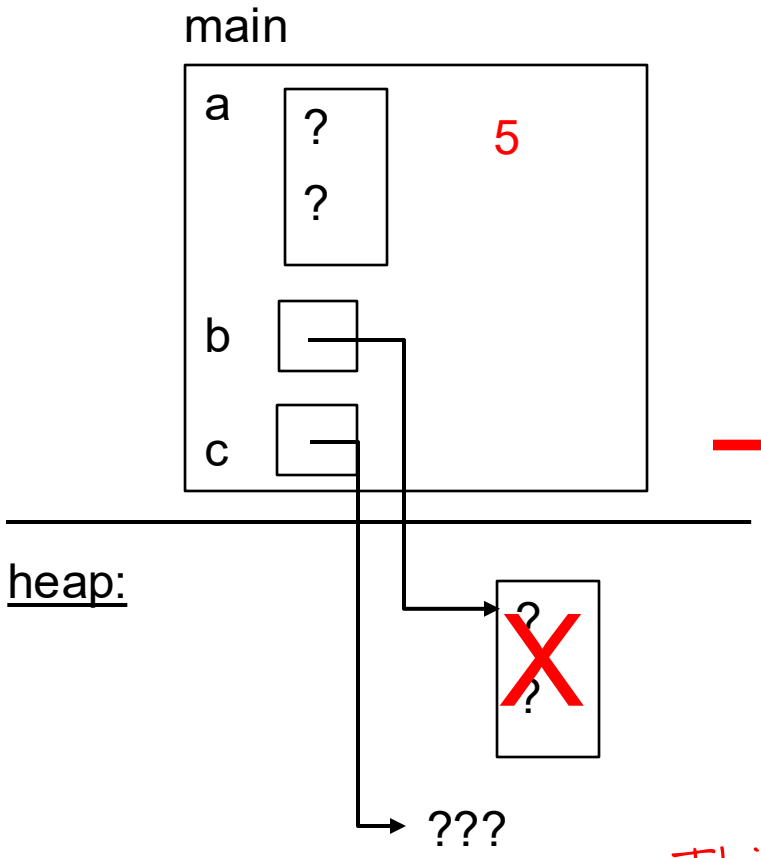
Crash!

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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?



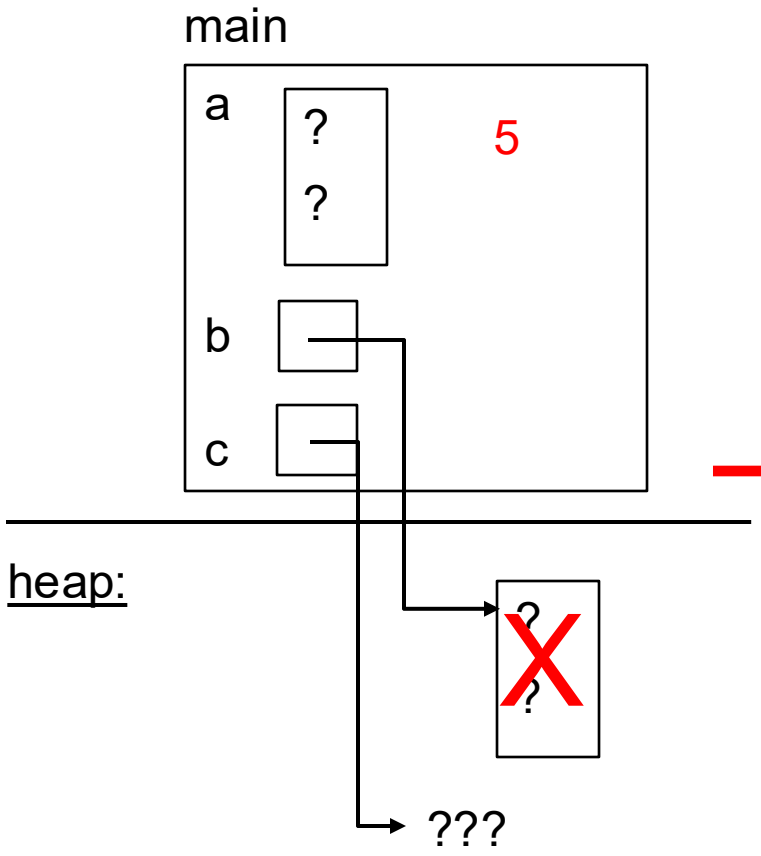

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

This "double free" would also cause the program to crash

memcorrupt.c

Memory Corruption - What Happens?

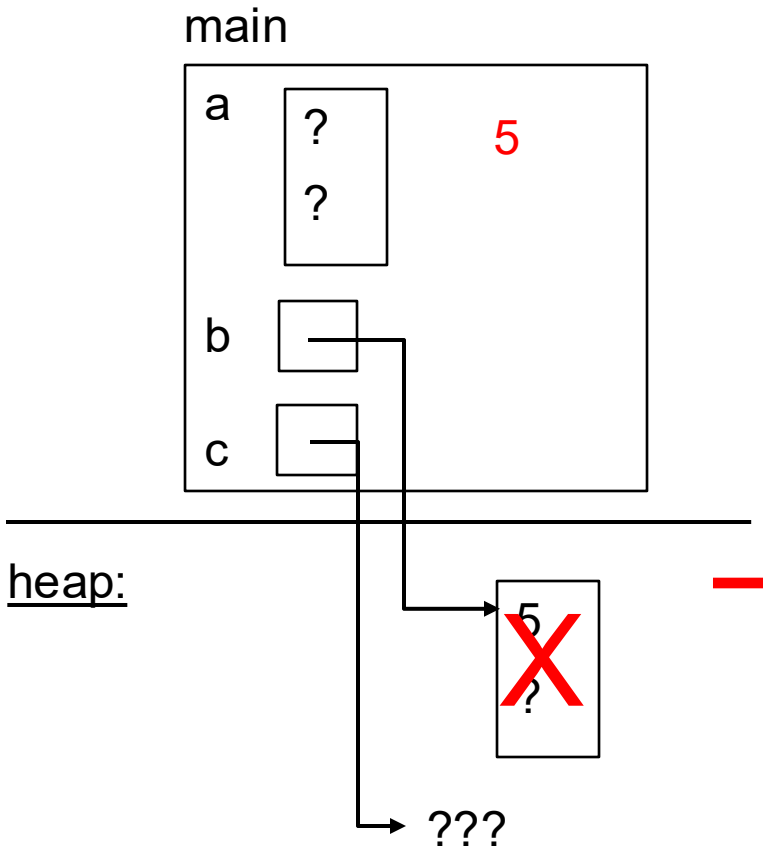



```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c

Memory Corruption - What Happens?




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

int main(int argc, char** argv) {
  int a[2];
  int* b = malloc(2*sizeof(int));
  int* c;

  a[2] = 5;    // assigns past the end of an array
  b[0] += 2;   // assumes malloc zeros out memory
  c = b+3;     // Ok, but if we use c, problem
  free(&(a[0]));  // free something not malloc'ed
  free(b);
  free(b);     // double-free the same block
  b[0] = 5;    // use a freed (dangling) pointer

  // any many more!
  return 0;
}
```

Note: Arrow points to next instruction.

memcorrupt.c