

Introductions, C refresher

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang

Haoyun Qin

Kevin Bernat

Ryoma Harris

Audrey Yang

Jason hom

Leon Hertzberg

Shyam Mehta

August Fu

Jeff Yang

Maxi Liu

Tina Kokoshvili

Daniel Da

Jerry Wang

Ria Sharma

Zhiyan Lu

Ernest Ng

Jinghao Zhang

Rohan Verma



pollev.com/tqm

❖ How are you?

Lecture Outline

- ❖ **Introduction & Logistics**
 - **Course Overview**
 - **Assignments & Exams**
 - **Policies**
- ❖ **C Refresher**
 - Memory Layout
 - Demo (make, man pages)
 - Malloc, free, pointers
 - stdin, stdout

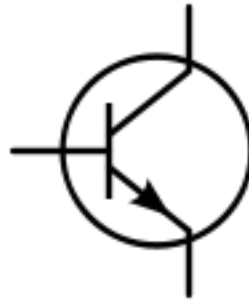
Instructor: Travis McGaha

- ❖ UPenn CIS faculty member since August 2021
 - First Semester with CIS 3800
 - CIS 2400 in 21fa & 22fa
 - CIT 5950 in 22sp & 23sp

- ❖ More on my personal website:
<https://www.cis.upenn.edu/~tqmcgaha/>

- ❖ Schedule meeting w/ me
- ❖ Unofficial office hours right after class
- ❖ Official office hours TBD

Course Overview

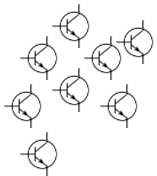




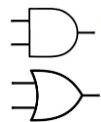
Course Overview



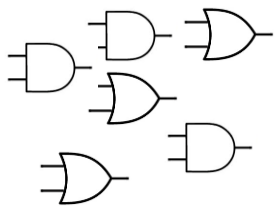
Course Overview



Course Overview



Course Overview



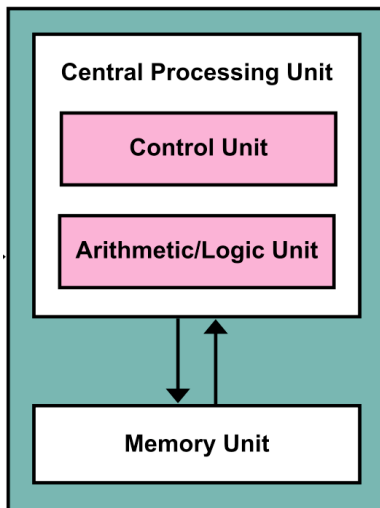
Course Overview

Adder

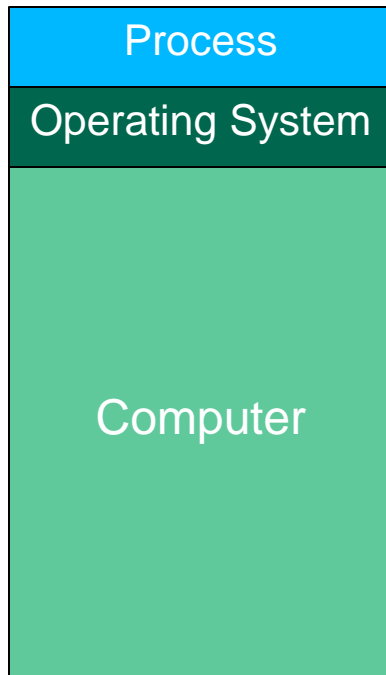
Mux/Demux

Latch/Flip-Flop

Course Overview



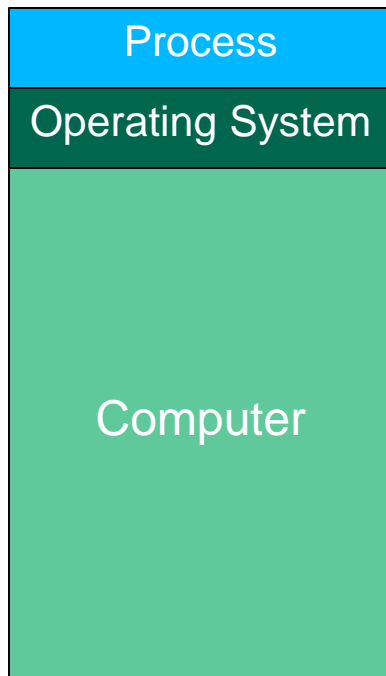
Course Overview



Course Overview

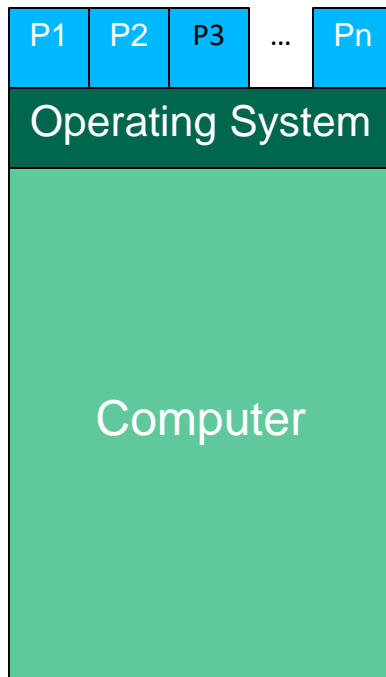


Course Overview



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

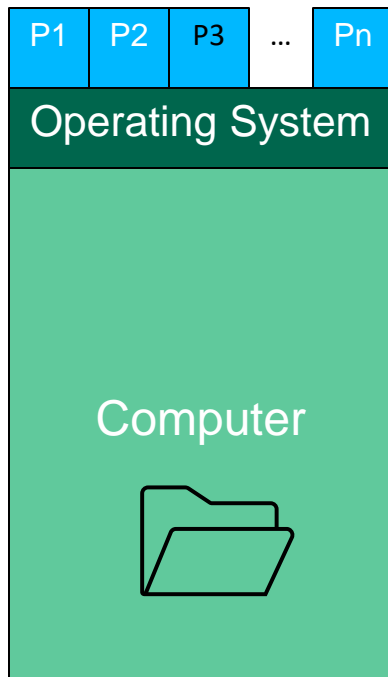
Course Overview



THERE IS A LOT
GOING ON TO
SUPPORT THIS



Course Overview



Prerequisites

- ❖ Course Prerequisites:
 - CIS 2400
 - Teamwork & Willingness/happy to spend substantial time coding

- ❖ What you should be familiar with already:
 - C programming
 - C Memory Model
 - Computer Architecture Model
 - Basic UNIX command line skills

- ❖ HW0 is tuned so that it will help refresh you on these.
 - But it still covers new content!
 - Even if you think you know C, get started sooner rather than later. 17

CIS 3800 Learning Objectives

- ❖ To leave the course with a better understanding of:
 - How a computer runs/manages multiple programs
 - How the previous point may affect the code we write
 - 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: These topics may be tweaked

Disclaimer

- ❖ This is a digest, **READ THE SYLLABUS**
 - <https://www.seas.upenn.edu/~cis3800/current/documents/syllabus>

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 (~26)
 - Introduces concepts, slides & recordings available on canvas
 - In lecture polling. Polls are not counted towards credit
- ❖ Pre-recorded videos (many)
 - Entirely optional
 - Goes over lecture material or demonstrates something for projects
- ❖ Check-ins “Quizzes” (~12)
 - Unlimited attempt low-stake quizzes on canvas to make sure you are caught up with material
 - Lowest two are dropped
- ❖ Midterms (2)
 - Two in-person exams, two pages of notes allowed
 - Details TBD

Programming Facilities

- ❖ Docker
 - Same environment as the autograder
 - Instructions for setup to be posted soon
- ❖ 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 & 1

❖ Project 0

- Unix “Shell” – command interpreter (e.g. sh, bash, etc)
- Excellent way to learn about how system calls are supported and used.
- Done individually
- Code review
- **Will be posted soon!!**

❖ Project 1

- Unix “Shell” – the real deal
- Redirection, pipelines, background/foreground processing, job control
- Groups of two.

PennOS

- ❖ Best way to learn about an operating systems is to build one.
- ❖ Build all the main features of an OS (in emulation)
- ❖ Groups of 4.
- ❖ 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

PennOS

- ❖ There is a paper on this:
<http://netdb.cis.upenn.edu/papers/pennos.pdf> at an ACM OS journal.
- ❖ Group evaluation done by the end of semester. Team member with lower than 15% contribution to the group will get grade downgrade.

HW Policies

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

- ❖ Late Policy
 - You are given 5 late tokens.
 - Tokens are counted per student and can only be used on some assignments.
 - Two tokens used at max per assignment
 - Each token grants 48 hours of extra time
 - If there are extenuating circumstances, please let me know

Collaboration Policy Violation

- ❖ You will be caught:
 - Careful grading of all written homeworks by teaching staff
 - Measure of Software Similarity (MOSS):
<http://theory.stanford.edu/~aiken/moss/>
 - Successfully used in several classes at Penn
- ❖ Zero in assignment, zero for class participation (3%). F grade if caught twice.
 - First-time offenders will be reported to Office of Student Conduct with no exceptions. Possible suspension from school
 - Your friend from last semester who gave the code will have their grade retrospectively downgraded.

Collaboration Policy Violation

❖ Generative AI

- I am skeptical of its usefulness for your learning and for your success in the course
- Some articles on the topic:
 - <https://www.aisnakeoil.com/p/chatgpt-is-a-bullshit-generator-but>
 - <https://www.aisnakeoil.com/p/gpt-4-and-professional-benchmarks>
- Not banned, but not recommended. Use your best judgement.

❖ You will not help your overall grade and happiness:

- Quizzed individually during project demo, 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

❖ Breakdown:

- Project 0 penn-shredder: (8%)
- Project 1 penn-shell: (18%)
- Project 2 PennOS: (37%)
- Exams (34%)
 - 17% each
- Check-in Quizzes(3%)

❖ Final Grade Calculations:

- I would LOVE to give everyone an A+ if it is earned
- Final grade cut-offs will be decided privately at the end of the Semester. What is used in previous semester is in the syllabus

Course Infrastructure

- ❖ Course Website: www.seas.upenn.edu/~cis3800/23fa/
 - Materials, Schedule, Syllabus ...
- ❖ Docker or Speclab
 - Coding environment for hw's
- ❖ Gradescope
 - Used for HW Submissions
- ❖ Poll Everywhere
 - Used for lecture polls
- ❖ Ed Discussion
 - Course discussion board

Getting Help

- ❖ Ed
 - Announcements will be made through here
 - Ask and answer questions
 - Sign up if you haven't already!

- ❖ Office Hours:
 - Can be found on calendar on front page of course website
 - Starts next week for all TAs

- ❖ 1-on-1's:
 - Can schedule 1-on-1's with Travis
 - Should attend OH and use Ed when possible, but this is an option for when OH and Ed can't meet your needs

We Care

- ❖ I am still figuring things out, but we do care about you and your experience with the course
 - Please reach out to course staff if something comes up and you need help

- ❖ **PLEASE DO NOT CHEAT OR VIOLATE ACADEMIC INTEGRITY**
 - We know that things can be tough, but please reach out if you feel tempted. We want to help
 - Read more on academic integrity in the syllabus



pollev.com/tqm

- ❖ Any questions, comments or concerns so far?

Lecture Outline

❖ Introduction & Logistics

- Course Overview
- Assignments & Exams
- Policies

❖ **C refresher**

- **Pointers**

- Arrays

I Will go through parts of this relatively fast.

Review this on your own

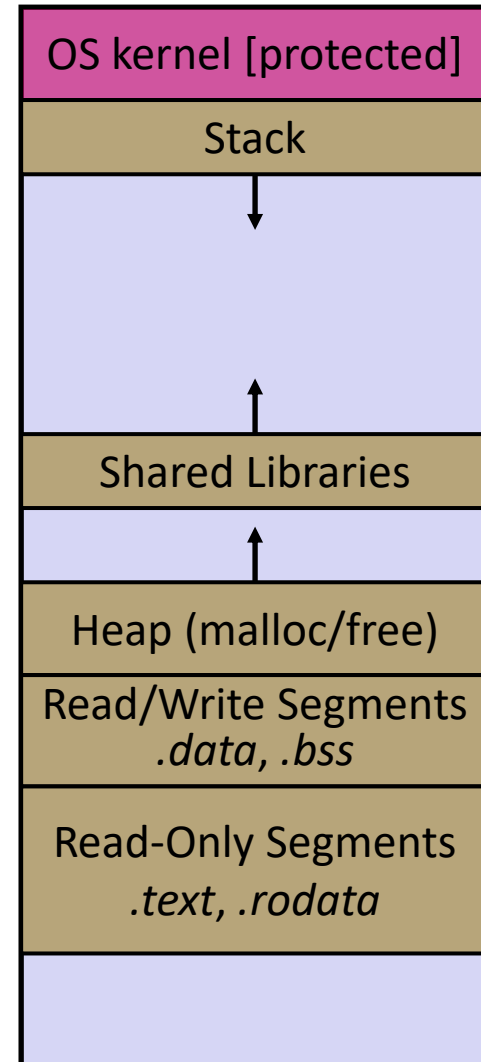


Hello World in C

Hello World: read()

Aside: 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



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

▪ Example:

```
int *ptr = ...; // Assume initialized
int a = *ptr; // read the value
*ptr = a + 2; // write the value
```

❖ Get the address of a variable with `&`

- `&foo` gets the address of `foo` in memory

▪ Example:

```
int a = 595;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Pointer Example

```


int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

Initial values
are garbage



0x2001	a	--
0x2002	b	--
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers
each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    → a = 5;
    → b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	--

Assuming that integers and pointers
each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	5
0x2002	b	3
0x2003	c	--
0x2004	ptr	0x2001

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    → *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	7
0x2002	b	3
0x2003	c	--
0x2004	ptr	0x2001

Assuming that integers and pointers each fit into a single memory location

Pointer Example

```

int main(int argc, char** argv) {
    int a, b, c;
    int* ptr;    // ptr is a pointer to an int

    a = 5;
    b = 3;
    ptr = &a;

    *ptr = 7;
    c = a + b;

    return 0;
}
    
```

0x2001	a	7
0x2002	b	3
0x2003	c	10
0x2004	ptr	0x2001

Assuming that integers and pointers each fit into a single memory location

 **Poll Everywhere**pollev.com/tqm

❖ What does this code print?

```
#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;
}
```

Poll Everywhere

pollev.com/tqm

❖ What does this code print?

❖ How could we fix it?
E.g. make modify point
actually modify a point

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

typedef struct point_st {
    int x, 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

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```


Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```
int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}
```

```
int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
```

main

soln1

?

soln2

?

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```

main

soln1	?
soln2	?

solve_quad

a	2.0
b	4.0
c	0.0
soln1	
soln2	
d	?

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```

main

soln1	?
soln2	?

solve_quad

a	2.0
b	4.0
c	0.0
soln1	
soln2	
d	16.0

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        → *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```

main

soln1	0
soln2	?

solve_quad

a	2.0
b	4.0
c	0.0
soln1	
soln2	
d	16.0

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        → return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```

main

soln1	0.0
soln2	-2.0

solve_quad

a	2.0
b	4.0
c	0.0
soln1	
soln2	
d	16.0

Output Parameters

- ❖ Pointers can be used to “return” more than one value from a function

```

int solve_quadratic(double a, double b, double c,
                   double* soln1, double* soln2) {
    double d = b*b - 4 * a * c;
    if (d >= 0) {
        *soln1 = (-b + sqrt(d)) / (2*a);
        *soln2 = (-b - sqrt(d)) / (2*a);
        return 1;
    } else {
        return 0;
    }
}

int main(int argc, char** argv) {
    double soln1, soln2; // populated by function call
    solve_quadratic(2.0, 4.0, 0.0, &soln1, &soln2);
    // ...
}
    
```

main

soln1	0.0
-------	-----

soln2	-2.0
-------	------

Arrays

- ❖ Definition: `type name [size]`
 - Allocates `size * 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

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

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?
- Arrays do not know their own size

Passes in address of start of array

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

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


Equivalent

❖ 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

Strings without Objects

- ❖ Strings are central to C, very important for I/O
- ❖ In C, we don't have Objects but we need strings
- ❖ If a string is just a sequence of characters, we can have use array of characters as a string

- ❖ Example:

```
char str_arr[] = "Hello World!";  
char *str_ptr = "Hello World!";
```

Null Termination

DO NOT FORGET THIS. THIS IS THE CAUSE OF MANY BUGS

❖ Arrays don't have a length, but we mark the end of a string with the null terminator character.

- The null terminator has value `0x00` or `'\0'`
- Well formed strings ***MUST*** be null terminated

❖ Example: `char str[] = "Hello";`

❖ Takes up 6 characters, 5 for "Hello" and 1 for the null terminator

address	0x2000	0x2001	0x2002	0x2003	0x2004	0x2005
value	'H'	'e'	'l'	'l'	'o'	'\0'

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 is something wrong with this function
- ❖ What is it? How do we fix this function w/o changing the function signature

```
#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 is something wrong with this function
- ❖ What is it? How do we fix this function w/o changing the function signature

```
#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;
    }

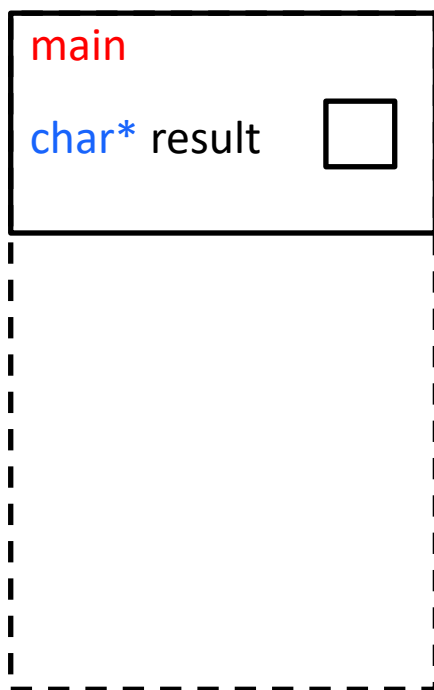
    return str;
}
```


Poll Everywhere

pollev.com/tqm

- ❖ There is something wrong with this function
- ❖ What is it? How do we fix this function w/o changing the function signature

The Stack



```
#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;
    }

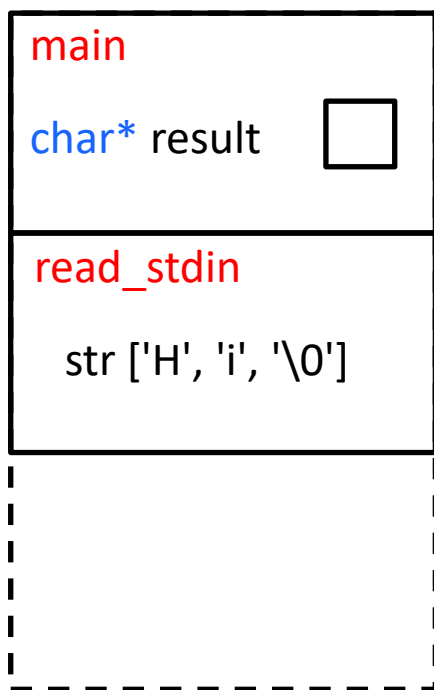
    return str;
}
```

Poll Everywhere

pollev.com/tqm

- ❖ There is something wrong with this function
- ❖ What is it? How do we fix this function w/o changing the function signature

The Stack



```
#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;
    }

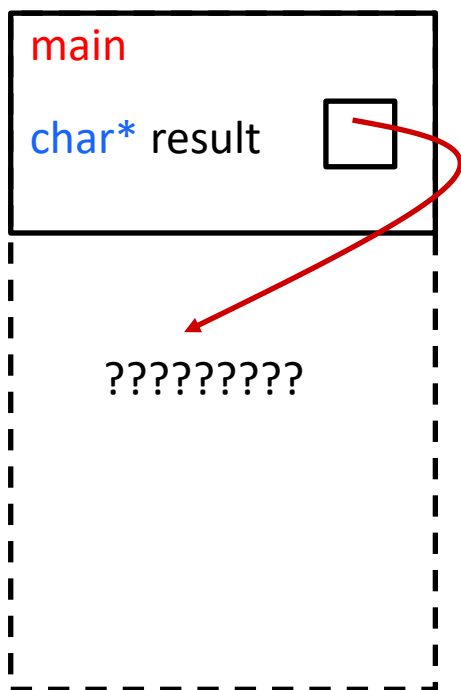
    return str;
}
```

Poll Everywhere

pollev.com/tqm

- ❖ There is something wrong with this function
- ❖ What is it? How do we fix this function w/o changing the function signature

The Stack



```
#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;
}
```

static function variables

❖ Functions can declare a variable as static

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

```
int next_num();
```

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

```
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;
}
```

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

Memory Allocation

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

```
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


```
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: NULL

- ❖ NULL is a memory location that is **guaranteed to be invalid**
 - In C on Linux, NULL is 0x0 and an attempt to dereference NULL *causes a segmentation fault*
- ❖ 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 = NULL;
    *p = 1; // causes a segmentation fault
    return EXIT_SUCCESS;
}
```

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 runtime 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

```

#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

```

#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

```

#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()

```
#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;
}
```


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 **freed**
 - 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

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>

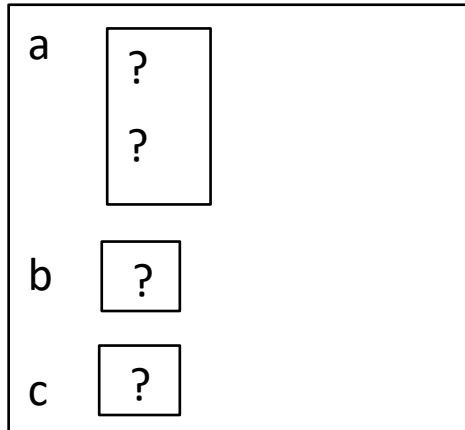
int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

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?

main



heap:

```

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

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

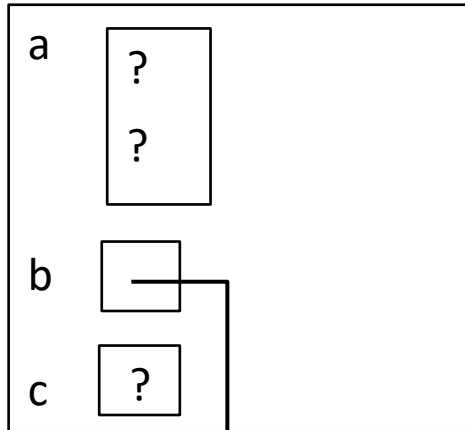
    a[2] = 5;    // assigns past the end of an array
    b[0] += 2;   // assumes malloc zeros out memory
    c = b+3;     // Ok, but if we use c, problem
    free(&(a[0])); // free something not malloc'ed
    free(b);
    free(b);     // double-free the same block
    b[0] = 5;    // use a freed (dangling)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

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

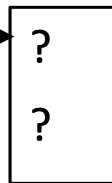
Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?

main



heap:



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

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

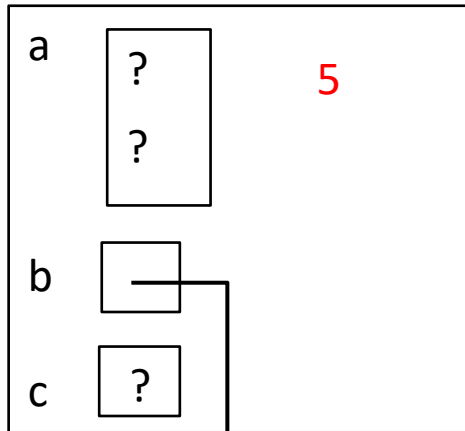
    a[2] = 5;    // assigns past the end of an array
    b[0] += 2;  // assumes malloc zeros out memory
    c = b+3;    // Ok, but if we use c, problem
    free(&(a[0])); // free something not malloc'ed
    free(b);    // double-free the same block
    b[0] = 5;   // use a freed (dangling) pointer

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

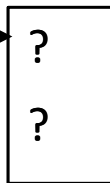
Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?

main



heap:



```

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

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

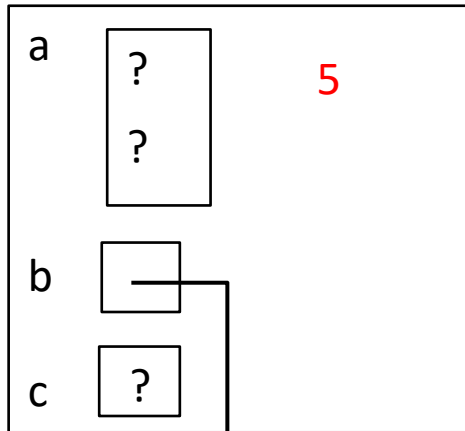
    a[2] = 5;    // assigns past the end of an array
    b[0] += 2;   // assumes malloc zeros out memory
    c = b+3;     // Ok, but if we use c, problem
    free(&(a[0])); // free something not malloc'ed
    free(b);     // double-free the same block
    b[0] = 5;    // use a freed (dangling) pointer

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

Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?

main



heap:



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

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

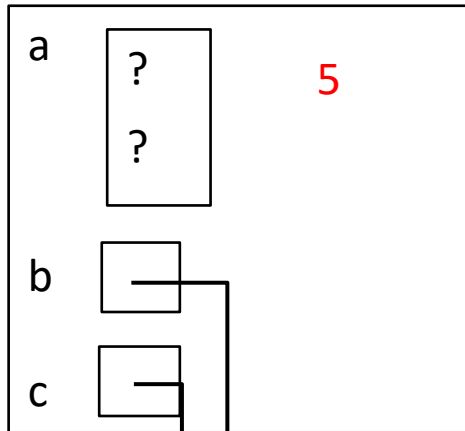
    a[2] = 5;    // assigns past the end of an array
    b[0] += 2;   // assumes malloc zeros out memory
    c = b+3;     // Ok, but if we use c, problem
    free(&(a[0])); // free something not malloc'ed
    free(b);     // double-free the same block
    b[0] = 5;    // use a freed (dangling) pointer

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

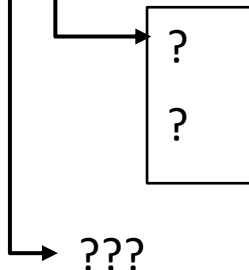
Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?

main



heap:



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

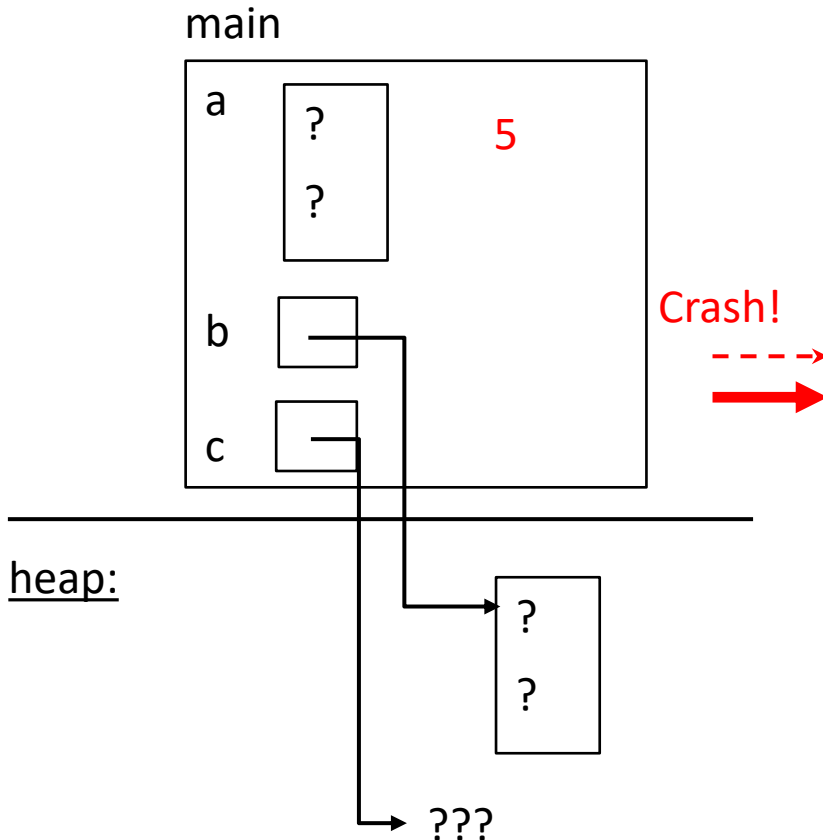
int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    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);    // double-free the same block
    b[0] = 5;   // use a freed (dangling) pointer

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

Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?



```

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

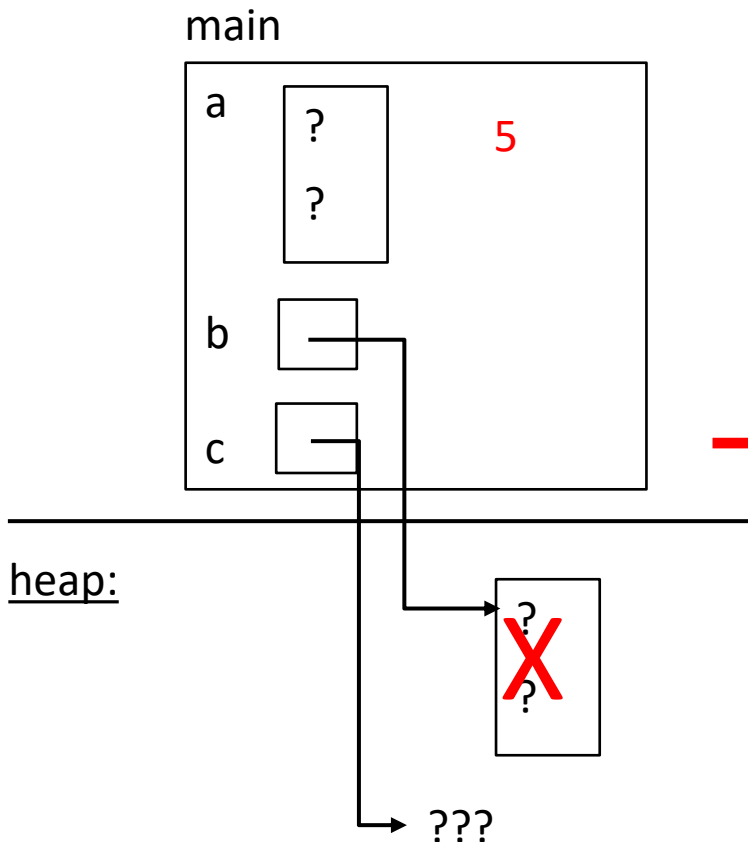
int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    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);    // double-free the same block
    b[0] = 5;   // use a freed (dangling) pointer

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

Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?



```

#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

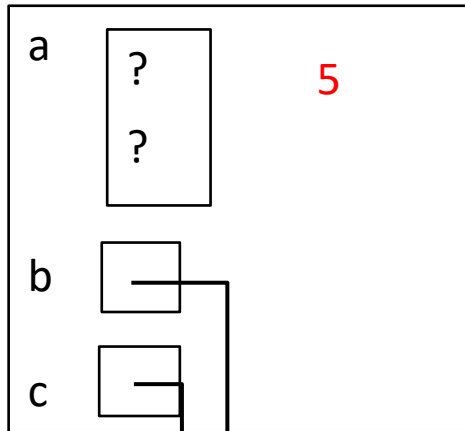
    // any many more!
    return 0;
}
    
```

Note: Arrow points to *next* instruction.

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

Memory Corruption - What Happens?

main



heap:



???

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

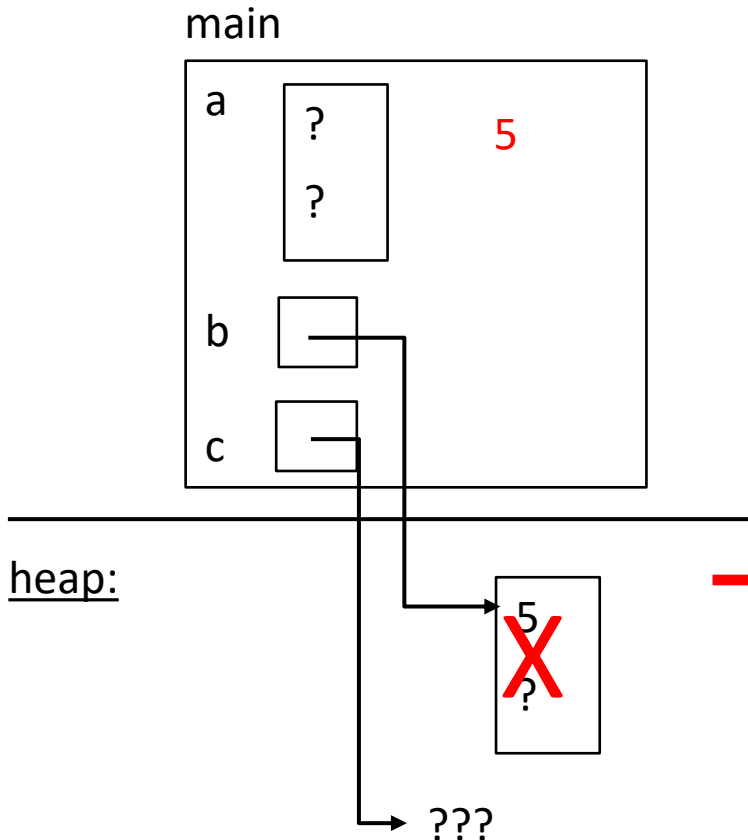
int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    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);    // double-free the same block
    b[0] = 5;   // use a freed (dangling) pointer

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

Note: Arrow points to *next* instruction.

Memory Corruption - What Happens?



```

#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);    // double-free the same block
    b[0] = 5;   // use a freed (dangling) pointer

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

Note: Arrow points to *next* instruction.