Introductions, Pointers, Arrays

Computer Systems Programming, Spring 2023

Instructor: Travis McGaha

TAs:

Kevine Bernat	Jialin Cai
Mati Davis	Donglun He
Chandravaran Kunjeti	Heyi Liu
Shufan Liu	Eddy Yang

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

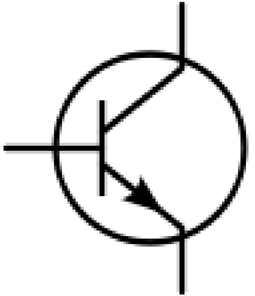
Lecture Outline

- **Introduction & Logistics**
  - **Course Overview**
  - **Assignments & Exams**
  - **Policies**
- Pointers
- Arrays

Instructor: Travis McGaha

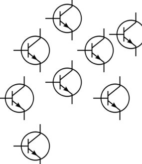
- UPenn CIS faculty member since… August 2021
  - Currently my fourth semester at UPenn
  - Second Semester with CIT 595
- Education: University of Washington, Seattle
  - Masters in Computer Science in March 2021
  - Bachelors in Computer Engineering in June 2019
  - Instructed a course that covers very similar material
- More on my personal website: https://www.cis.upenn.edu/~tqmcgaha/

Course Overview

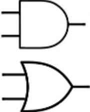


Course Overview

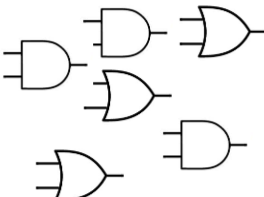
Course Overview



Course Overview



Course Overview



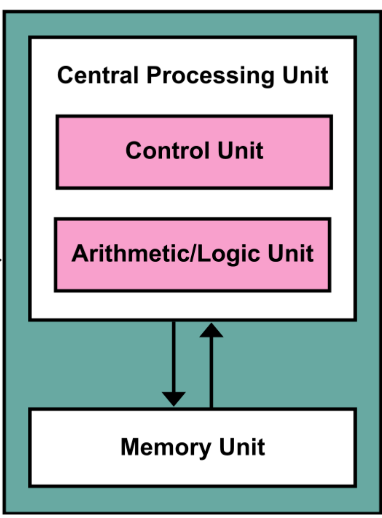
Course Overview

Adder

Mux/Demux

Latch/Flip-Flop

Course Overview



Course Overview

Course Overview

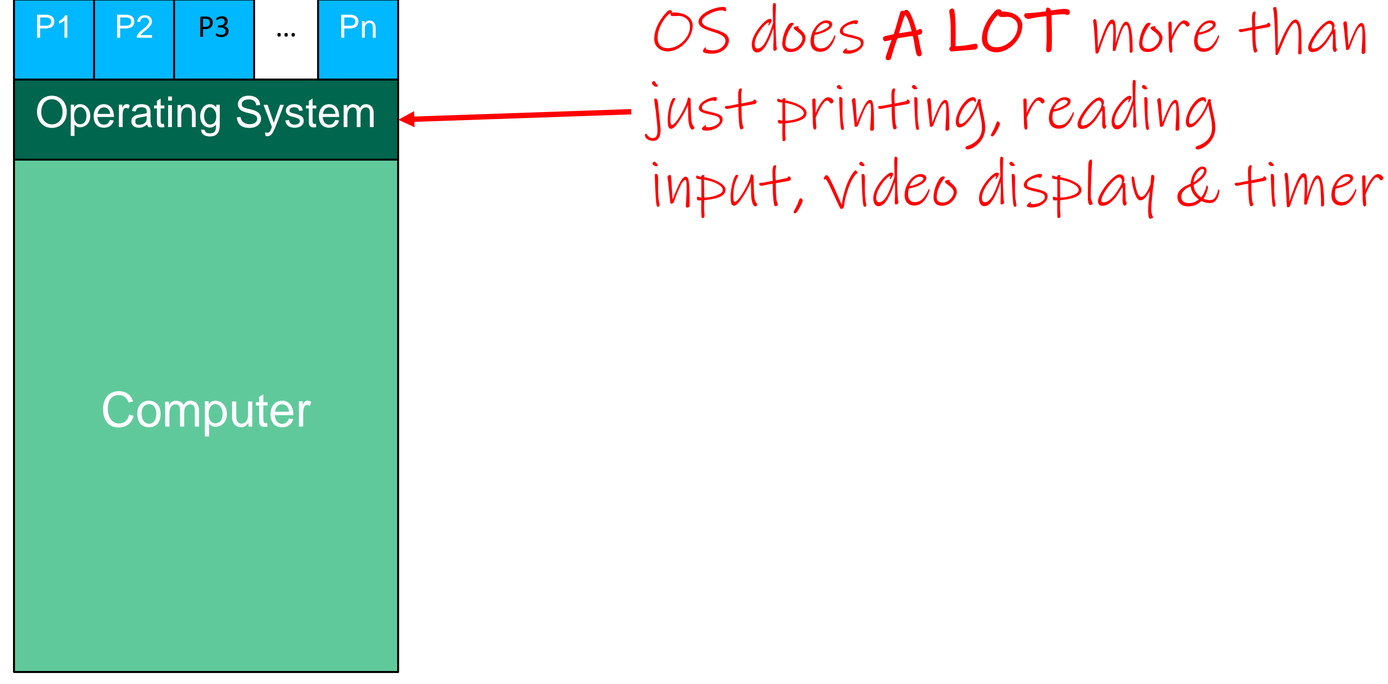
Course Overview

Process

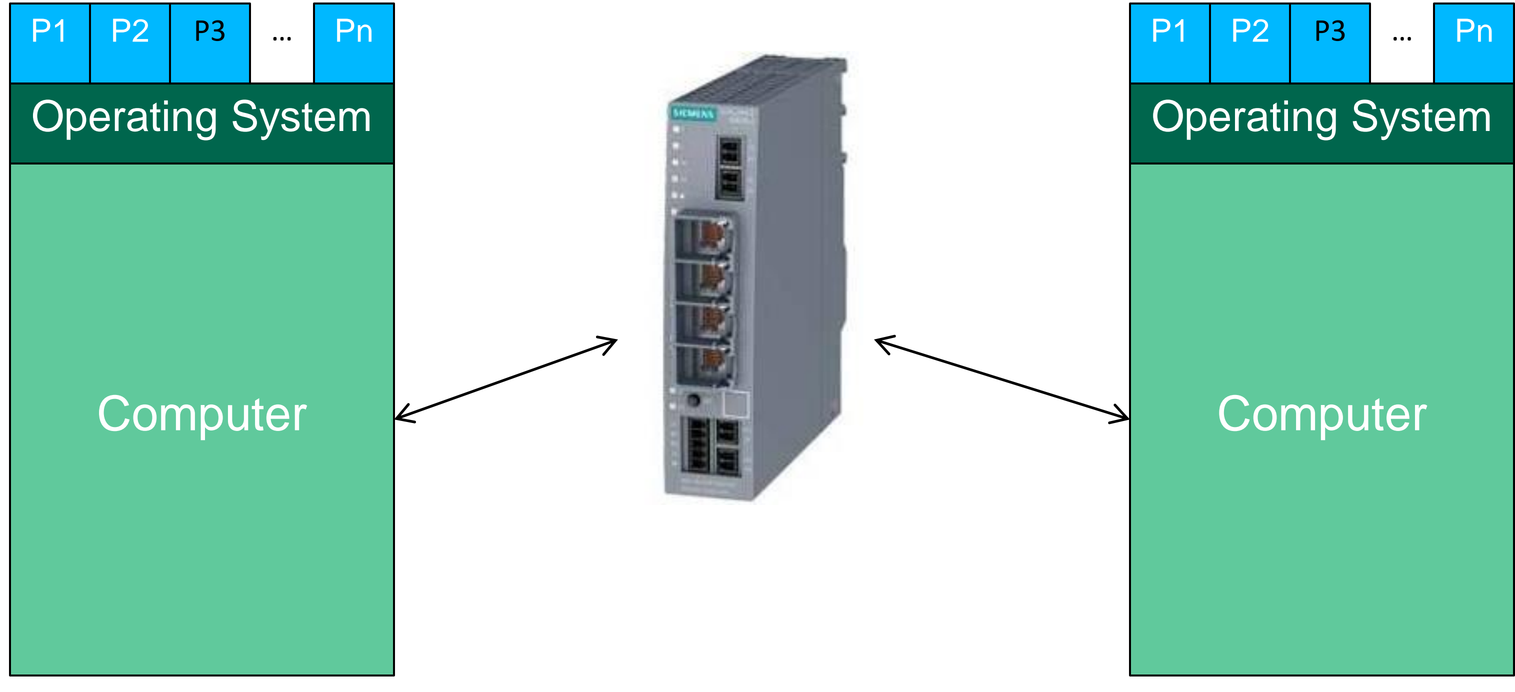
Operating System

Computer

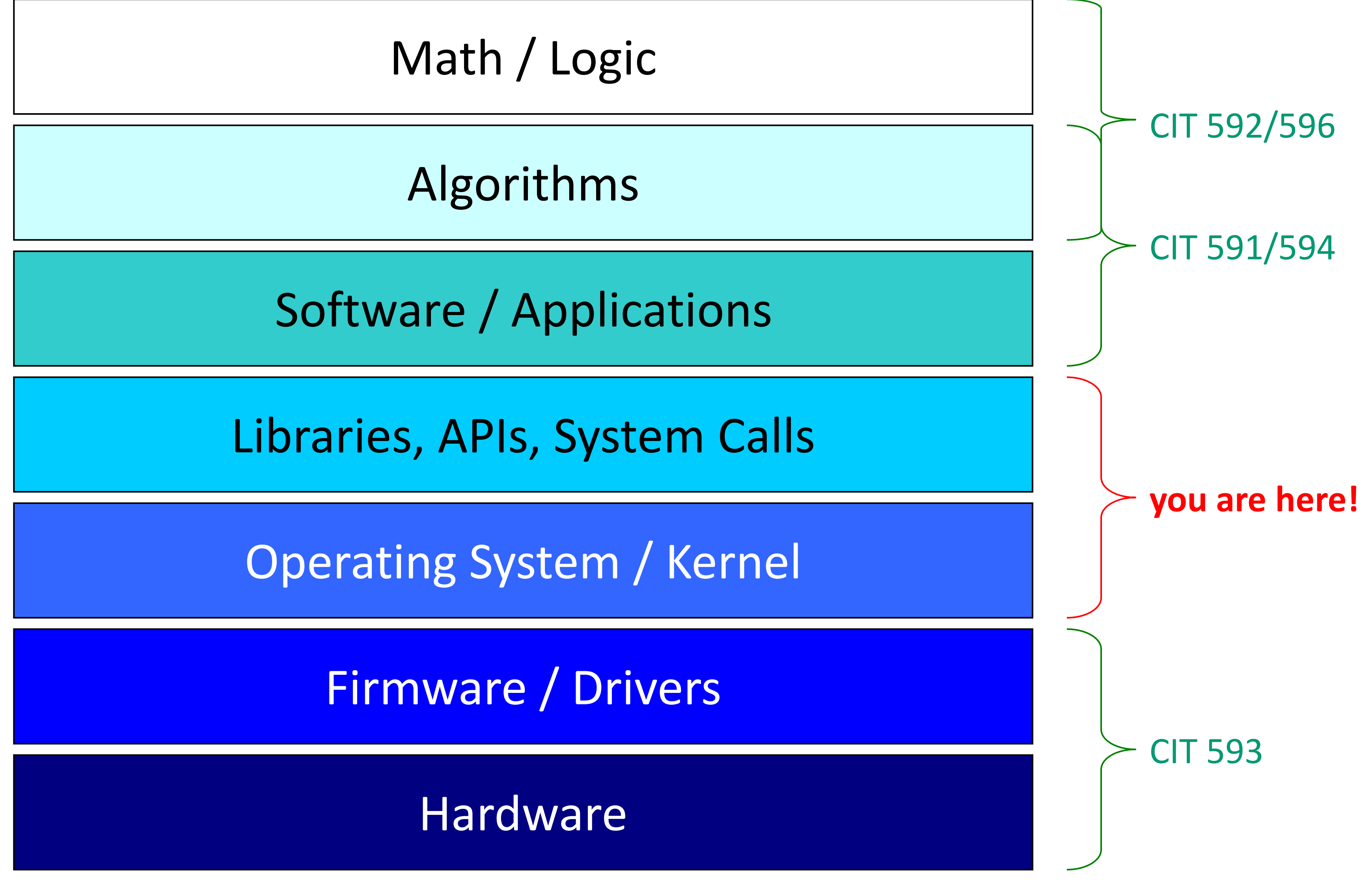
Course Overview



Course Overview



Course Overview



Prerequisites

- Course Prerequisites:
  - CIT 5930
- What you should be familiar with already:
  - C programming
  - C Memory Model
  - Computer Architecture Model
  - Basic UNIX command line skills
- Will still cover some of these lightly with the beginning of the semester ☺

CIT 5950 Learning Objectives

- To leave the class with a better understanding of:
  - How software "interfaces" with Operating System
  - How a computer runs/manages multiple programs
  - Various system resources and how to apply those to code
    - Threads, networking, file I/O
  - C++

- Topics list/schedule can be found on course website
  - Note: These may be slightly tweaked

Disclaimer

- This is a digest, **READ THE SYLLABUS**
  - https://www.seas.upenn.edu/~cit5950/23sp/documents/syllabus

Course Components pt. 1

- Lectures (28)
  - Introduces concepts, slides & recordings available on canvas
  - In lecture polling. Polls remain open until the next lecture
- Sections (12)
  - Reiterates lecture content, lecture clarifications, assignment & exam preparation
- Programming Projects (5)
  - Due every ~2 weeks
  - Applications of course content
- Check-ins “Quizzes” (12)
  - Unlimited attempt low-stake quizzes on canvas to make sure you are caught up with material
  - Lowest two are dropped

Course Components pt. 2

- Final Project (1)
  - Due at the end of the semester
  - Can be done solo or in partners (tentatively)
  - Further Details TBD
- Take home Exams (2)
  - Two virtual take-home exams
  - Midterm will be the week before spring break
  - Final will be the week of finals
- Textbook (0)
  - No Textbook, but using a C++ reference would probably be useful
  - https://cplusplus.com/
  - https://en.cppreference.com/w/

Course Policies

- HW Late Policy
  - Late days given on request
    - (Request usually granted)
  - No cap on the number of late days per assignment
    - More than 3 on an assignment requires approval from Travis

- Midterm Clobber Policy
  - Final is cumulative
  - If you do better on the “midterm section” of the final, your midterm grade can be overwritten.

Course Grading

- Breakdown:
  - Homeworks (55%)
  - Final Project (15%)
  - Exams (25%)
    - Midterm 10%
    - Final 15%
  - Check in Quizzes (5%)

- 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

Course Infrastructure

- Course Website
  - Schedule, syllabus, materials …
- Codio
  - Coding environment for hw's
- Gradescope
  - Used for exams & HW submissions
- Poll Everywhere
  - Used for lecture polls
- Ed
  - Course discussion board
- Canvas
  - grades, lecture recordings, surveys & quizzes

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 canvas page
  - Starts next week
- 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 Piazza can't meet your needs

We Care

- We care about you and your experience with the course
  - There is a pre-semester survey available on canvas now. Please fill this out honestly and we will do our best to incorporate people's answers
  - There are pretty much unlimited extensions
  - 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

Questions?

- Any questions for me about ANYTHING?

Lecture Outline

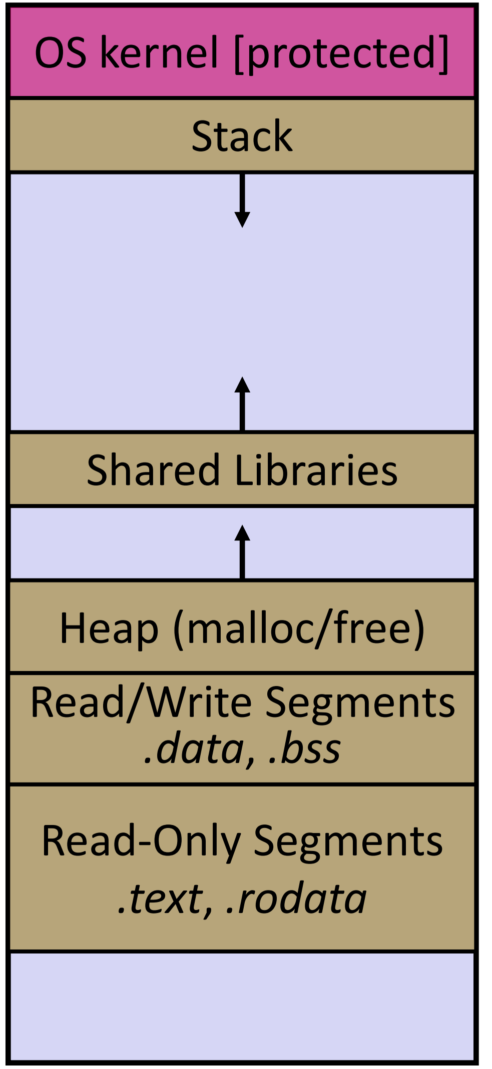
- Introduction & Logistics
  - Course Overview
  - Assignments & Exams
  - Policies
- **Pointers**
- Arrays

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 & 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

equivalent

- Generic definition: `type* name;` or `type *name;`
  - 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
```

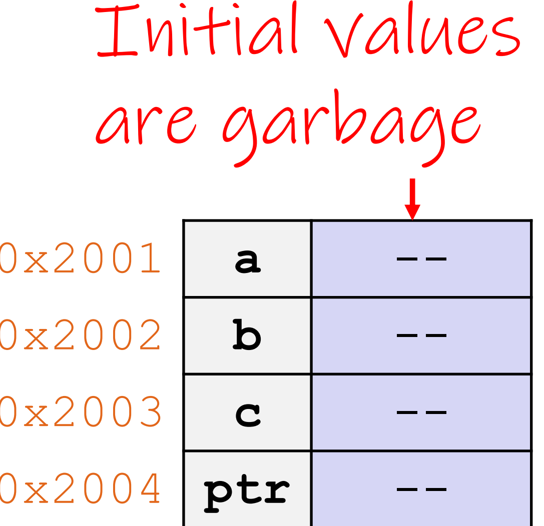
Pointer Example

```
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;
}
```



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;
}
```

Address	Name	Value
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;
}
```

Address	Name	Value
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

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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

PRO TIP: Draw out the addresses with "Boxes & Arrows" to visualize what is going on

- 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

pollev.com/tqm

- What is printed in this program?

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

A. 10, 24, 33

B. 10, 33, 33

C. 24, 10, 10

D. 24, 33, 33

E. I’m not sure

Poll Everywhere

pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

main

a	b	c
10	24	33

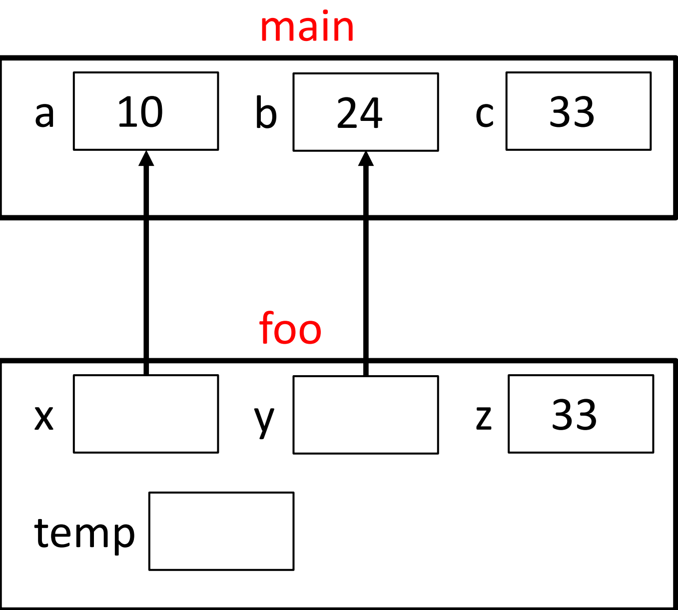
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
→ int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



Poll Everywhere

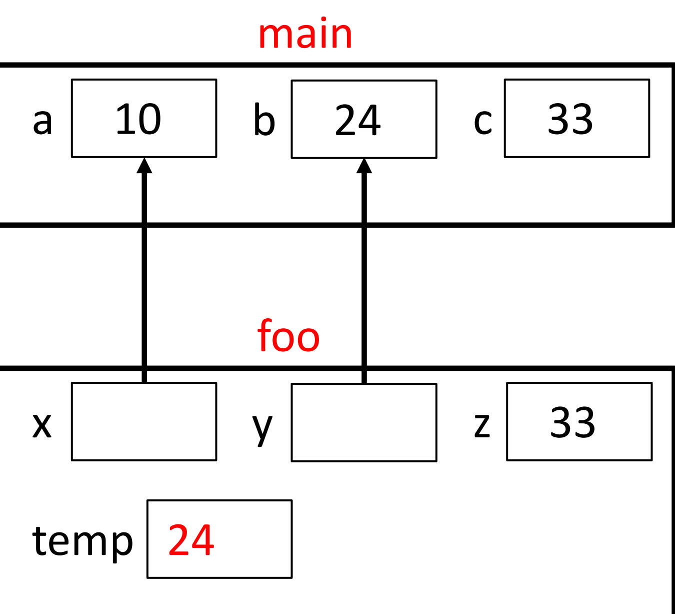
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



Poll Everywhere

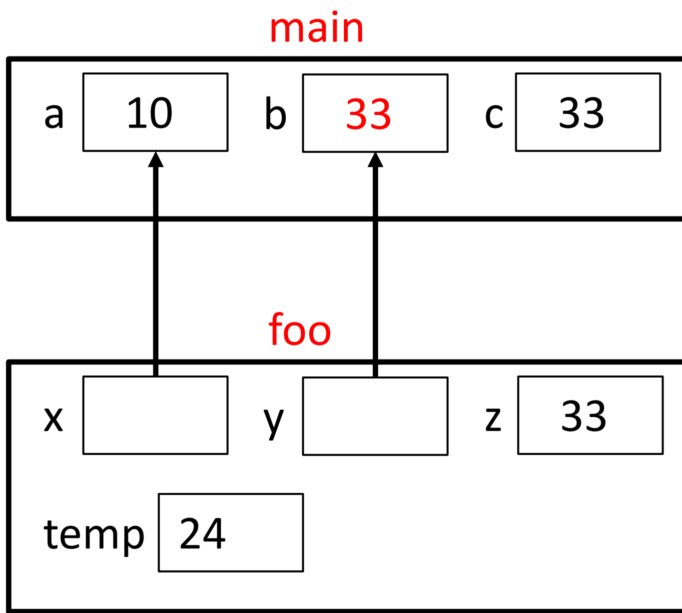
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;        <-- (red arrow)
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



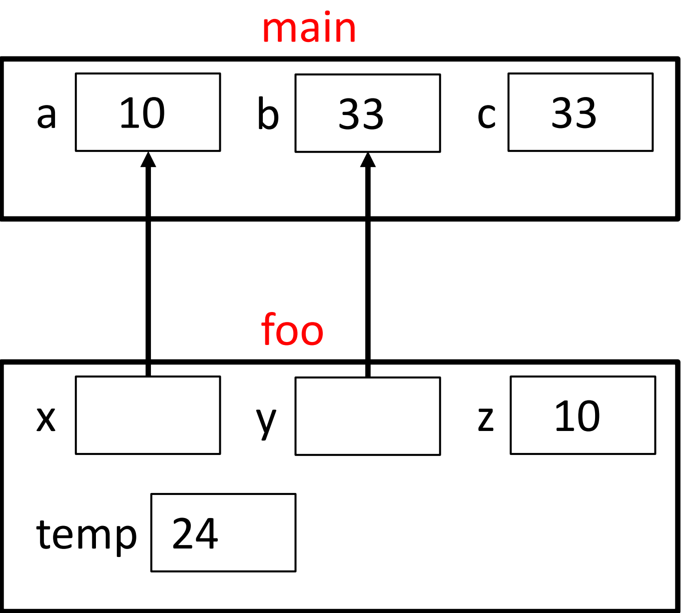
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
→ y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



Poll Everywhere

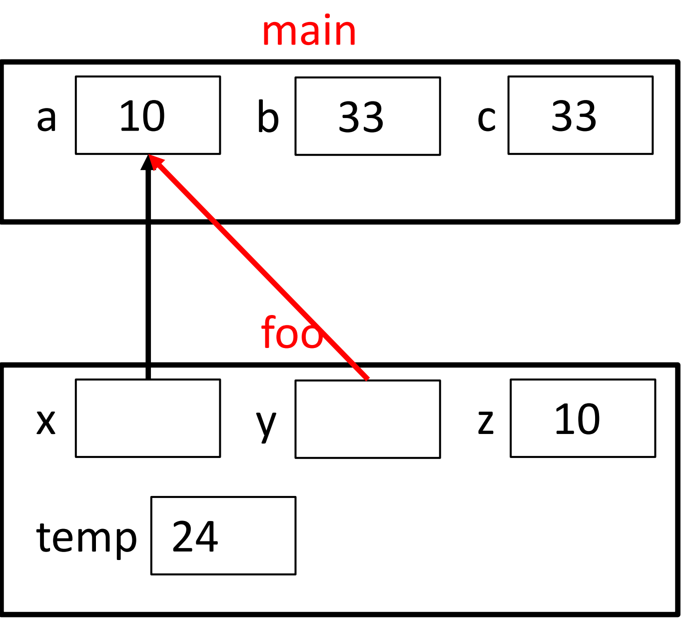
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```c
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;    // <- next line to execute
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



Poll Everywhere

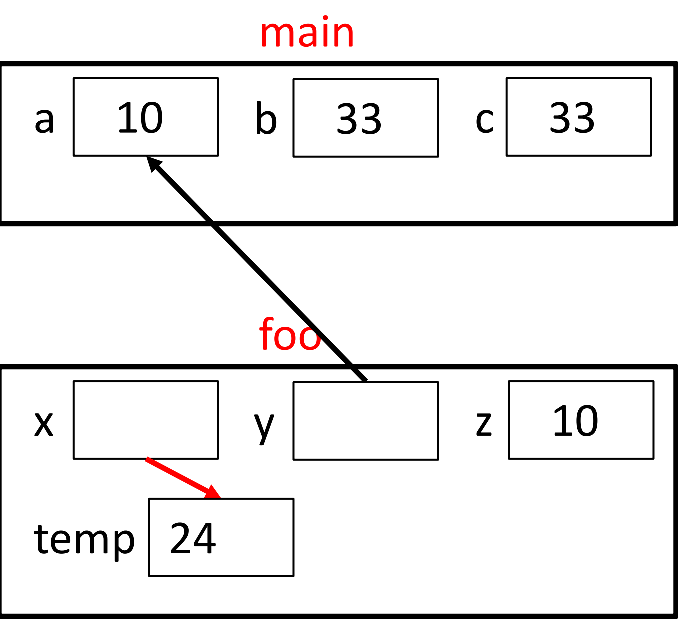
pollev.com/tqm

- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```



Poll Everywhere

pollev.com/tqm

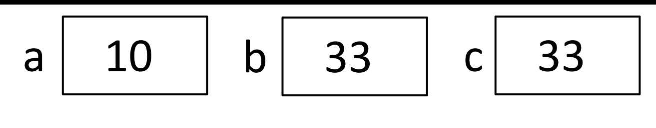
- What is printed in this program?

Red arrow indicates the NEXT line to execute

```
void foo(int *x, int *y, int z) {
  int temp = *y;
  *y = z;
  z = *x;
  y = x;
  x = &temp;
}

int main() {
  int a = 10, b = 24, c = 33;
  foo(&a, &b, c);
  printf("%d, %d, %d\n", a, b, c);
  return EXIT_SUCCESS;
}
```

main



B. 10, 33, 33

Lecture Outline

- Introduction & Logistics
  - Course Overview
  - Assignments & Exams
  - Policies
- Pointers
- **Arrays**

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

Pointer Arithmetic

- We can do arithmetic on addresses to iterate through arrays.
  - ```
    double my_array[10]; // create an array of 10 doubles
    double *ptr = my_array; // ptr has the address of the
                            // first element
    ptr = ptr + 1; // increment ptr to point to
                   // the next elemetnt
    ptr[2] = 3.14; // equivalent to *(ptr + 2) = 3.14
    ```
- Pointers are *typed*
  - Tells the compiler the size of the data you are pointing to
- Pointer arithmetic is scaled by `sizeof(*ptr)`
  - Sometimes a single array element can span multiple addresses
  - Works nicely for arrays

Size (number of bytes) of thing being pointed at
    ```

pollev.com/tqm

- What are the final values of nums?

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```

A. 2, 3, 5, 6

B. 38, 38, 5, 6

C. 2, 38, 5, 2

D. 2, 38, 5, 5

E. I'm not sure

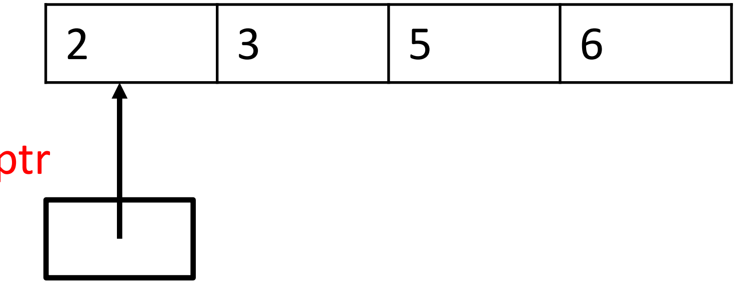
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



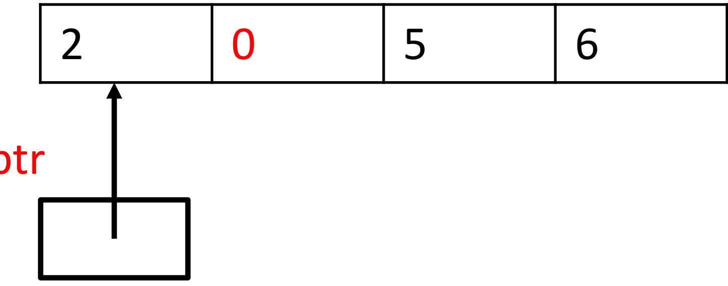
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



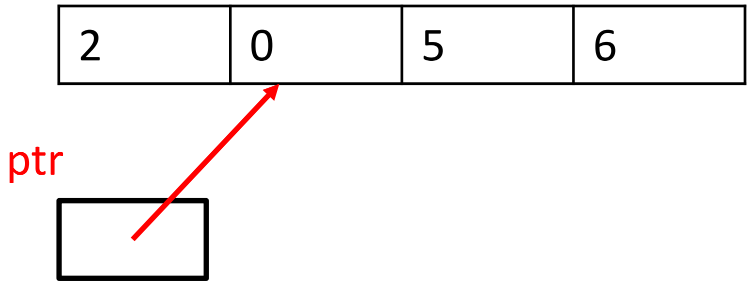
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;          <- (red arrow: next line)
ptr[1] = nums[0];
```

nums

2	38	5	6

ptr (points to nums[1])

pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];   ← (next line)
```

nums

2	38	5	6

ptr (points to nums[2])

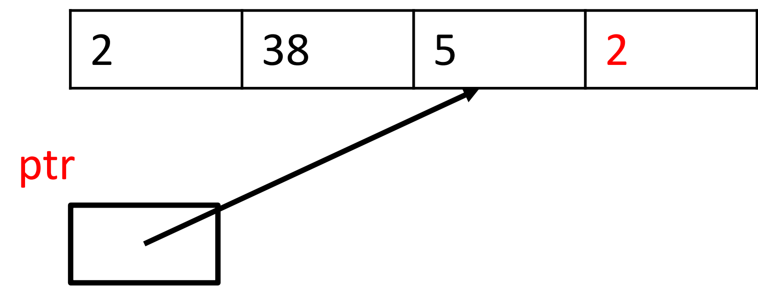
pollev.com/tqm

- What are the final values of nums?

Red arrow indicates the NEXT line to execute

```
int nums[4] = {2, 3, 5, 6};
int *ptr = nums;

ptr[1] = 0;
ptr++;
ptr[0] = 38;
ptr++;
ptr[1] = nums[0];
```



C. 2, 38, 5, 2