C Programming

Computer Operating Systems, Spring 2024

Instructors: Joel Ramirez Travis McGaha

Head TAs: Adam Gorka Daniel Gearhardt
Ash Fujiyama Emily Shen

TAs:

Ahmed Abdellah
Angie Cao
August Fu
Caroline Begg
Cathy Cao
Claire Lu
Eric Sungwon Lee
Ethan Weisberg
Garrett O'Malley Kirsch
Hassan Rizwan
Iain Li
Jerry Wang
Juan Lopez
Keith Mathe
Maya Huizar
Meghana Vasireddy
Perrie Quek
Sidharth Roy
Sydnie-Shea Cohen
Vivi Li
Yousef AlRabiah

Poll Everywhere

pollev.com/tqm

- What questions do you have for me?

Administrivia (pt. 1)

- HW00 Posted
  - Should have everything you need after this lecture
  - Autograder Posted over the weekend
  - Due: Friday 9/6 @ Midnight
  - Start ASAP since you need to setup the environment
  - Short HW00 Demo in a second

- Survey00: Pre-semester Survey
  - Anonymous Survey, live now
  - On Canvas (So that it can be anonymous)
  - Due Wednesday 9/11 @ midnight

Administrivia (pt. 2)

- Check-in 00
  - Short questions about C
  - Due before lecture on Tuesday
  - Releases tonight or sometime tomorrow
  - Will be on gradescope

HW00 Demo

- Demonstrate how to run it
- Compiling it is something you need to figure out

- **`clang-15`** is the compiler you should use for the assignment
  - If it is not installed, try running this in the terminal:

    ```
    apt-get install -y clang-15
    ```

Website & Infra Demo

- Website: https://www.seas.upenn.edu/~cis2400/current/
- Canvas site: https://canvas.upenn.edu/courses/1811752
- Docker: see setup doc on course website

Lecture Outline

- **C Intro**
  - **Cont. from last time:**
    - **Arrays**
    - **Command line args**
  - Pointers
    - Box & Arrow Diagrams
    - Arrays vs pointers
    - C Strings
  - ~~Structs~~
  - ~~The Stack & Pass-by-value~~
  - More on Compiling

Sample C program: sum evens

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

int sum_evens(int n) {
  int sum = 0;
  for (int i = 0; i < n; i++) {
    if (i % 2 == 0) {
      sum += i;
    }
  }
  return sum;
}

int main() {
  int sum = sum_evens(5);
  printf("sum: %d\n", sum);
  return EXIT_SUCCESS;
}
```

Function declarations & parameters

Variables local to the function

For loops & if statements look similar

Print statements are different to format output. This replaces %d with the value of sum, more later in lecture

Another Similarity: Scope

- Variables declared inside of a function are local to that function and are not visible outside of that scope.

- Variables can also be declared outside of a function – these variables typically have global scope but there are some subtleties

C vs Java Similarities Overview

- C and Java are very similar syntactically

- Similarities:
  - Control Structures (if/else/for/while/…)
  - Variables and data types (int/char/float/double/…)
  - Arrays and strings exist in both
    (but are also different implementation wise)
  - Statements & Expressions
    x = (y + z) / 2

C vs Java

- C and Java are Syntactically Similar, but …
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**
  - **do not assume everything in C is like Java**

- From my experience, a common source for making mistakes in C is forgetting that things are not like Java

C vs Java: Differences

- C is functionally very different than Java

- Some differences:
  - C doesn’t default initialize anything
  - C doesn’t have objects
  - C compiles down to machine code
  - C runs really fast
  - C doesn’t check much in terms of safety, no nice error messages like Java has
  - C is “just above” assembly in terms of abstraction
  - C allows for direct memory access
  - **Java has implicit references, C is explicit with pointers**

More on this in a second

Arrays

- Definition: `type name[size]`
  - Allocates `size` *contiguous* elements of type `type`
  - Normal usage is a compile-time constant for `size` (*e.g.* `int scores[5];`)
  - Initially, array values are "garbage" == Random values

value	10	9	9	9	10

- 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
  - The array name 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 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

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

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

C command line args

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

int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++) {
    printf("%s\n", argv[i]);
  }
  return EXIT_SUCCESS;
}
```

- **argc** is the number of arguments given to the program.
  - **The name of the program is counts as an argument.**
- **argv is an array of char*'s** (strings) that are the arguments
  - **The name of the program is the first argument.**

C command line args [Live Demo]

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

int main(int argc, char* argv[]) {
  for (int i = 0; i < argc; i++) {
    printf("%s\n", argv[i]);
  }
  return EXIT_SUCCESS;
}
```

- **Let’s see an example…**

- Note how everything arg is a string, will need to do conversion to other types if you want that.

Lecture Outline

- C Intro
  - Cont. from last time:
    - Arrays
    - Command line args
  - **Pointers**
    - **Box & Arrow Diagrams**
    - **Arrays vs pointers**
    - **C Strings**
  - ~~Structs~~
  - ~~The Stack & Pass-by-value~~
  - More on Compiling

Pointers

POINTERS ARE EXTREMELY IMPORTANT IN C

- Variables that are explicit “references”
  - Holds the location to some data in computer memory
  - Must specify a type so the data being referred to can be interpreted

equivalent

- Generic definition: `type* name;` or `type *name;`
  - Example: `int *ptr;`
    - Declares a variable that can refer to an int
    - 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 data
  - Example:

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

- Get the “reference” of a variable with `&`
  - `&foo` gets a “reference” to foo in memory
  - Example:

```
int a = 240;
int *ptr = &a;
*ptr = 2; // 'a' now holds 2
```

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

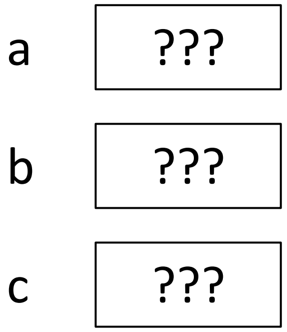
```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```



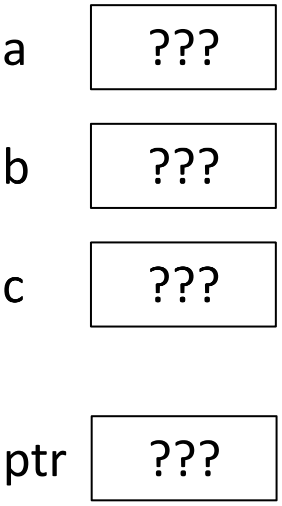
a	???
b	???
c	???

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```



a ???

b ???

c ???

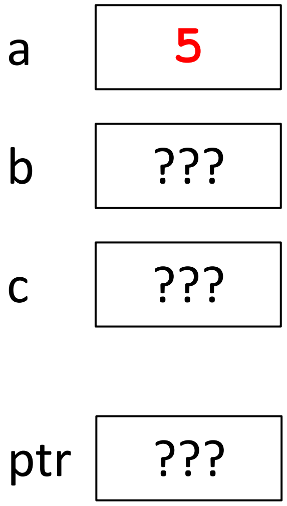
ptr ???

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

  a = 5;
→ b = 3;
  ptr = &a;

  *ptr = 7;
  c = a + b;

  return 0;
}
```

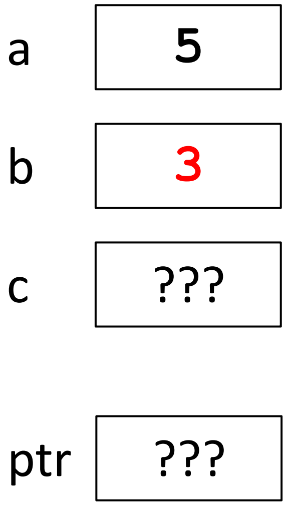


Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```c
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```



a 5

b 3

c ???

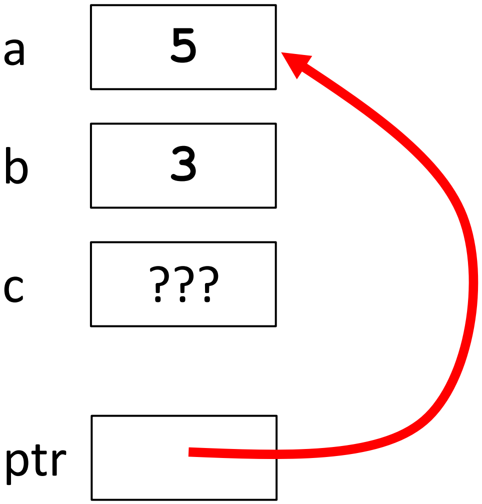
ptr ???

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```c
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```



a 5

b 3

c ???

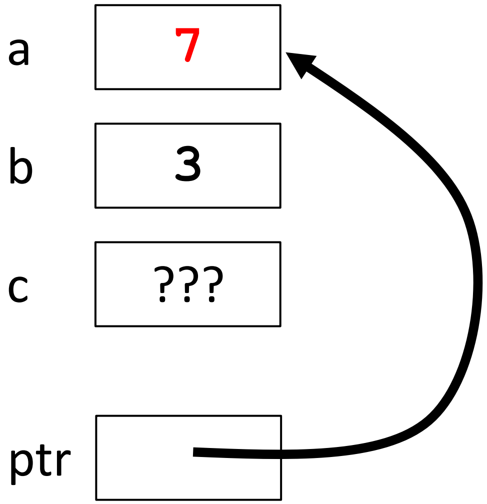
ptr

Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

  return 0;
}
```

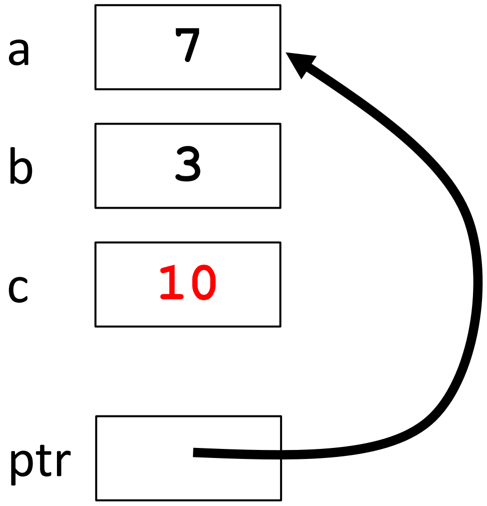


Box and Arrow Diagrams

Red arrow is the next line to execute

- Really Really Really useful thing to visualize C code is to draw diagrams with boxes and arrows to visualize what is going on.

```
int main(int argc, char* argv[]) {
  int a, b, c;
  int* ptr;

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

  *ptr = 7;
  c = a + b;

→ return 0;
}
```



Poll Everywhere

pollev.com/tqm

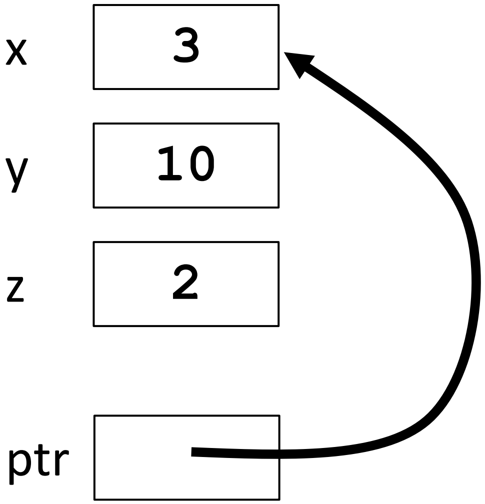
- What does this program print?

```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```

Poll Solution

- What does this program print?

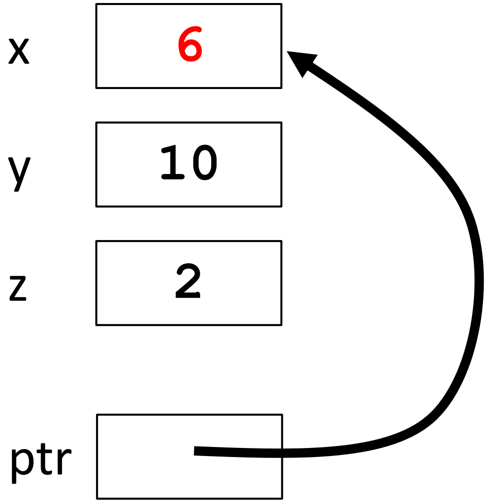
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

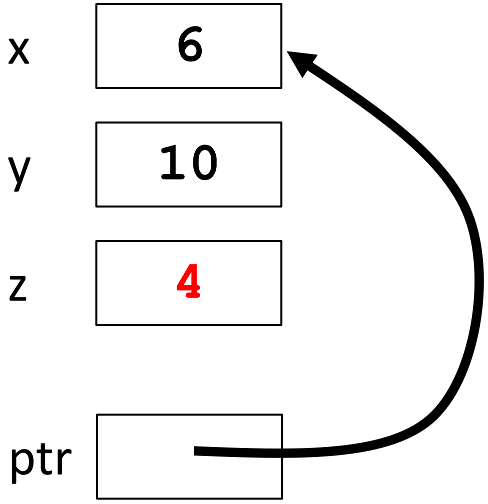
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

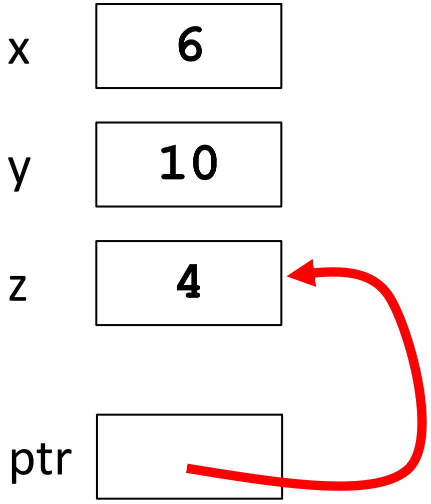
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



x 6

y 10

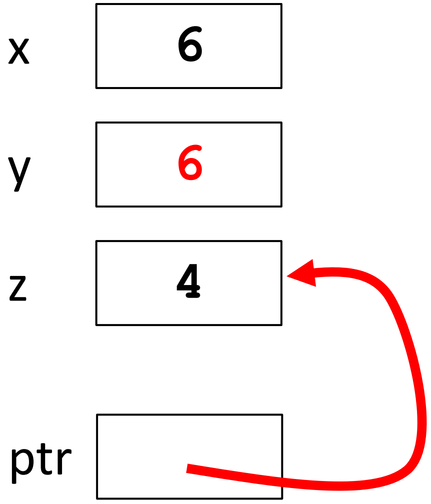
z 4

ptr

Poll Solution

- What does this program print?

```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *ptr = &x;
  *ptr = z * *ptr;
  z = 4;
  ptr = &z;
  y = *ptr + 2;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Pointers to Pointers to Pointers to Pointers

- You can have pointers to pointers:
- A pointer to a pointer to an int:

```
int x = 3;
int *ptr = &x;
int **ptr_ptr = &ptr;
```

- Can dereference more than one at a time:
  - Deference's twice: `**ptr = 3;`

- Can have pointers to pointers to pointers to pointers to pointers… `int ********************ptr;`

Poll Everywhere

pollev.com/tqm

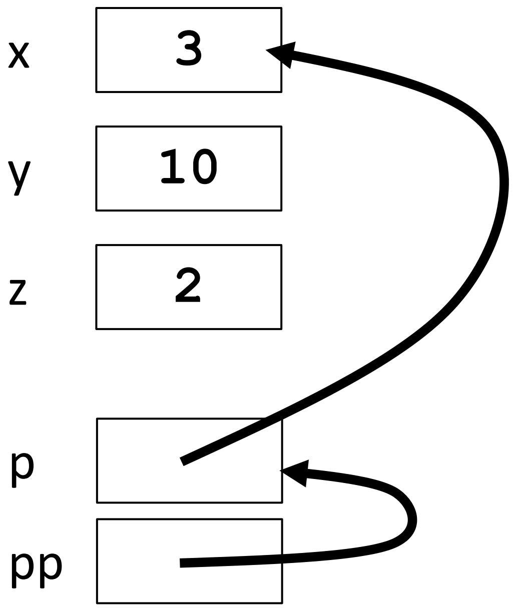
- What does this program print?

```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```

Poll Solution

- What does this program print?

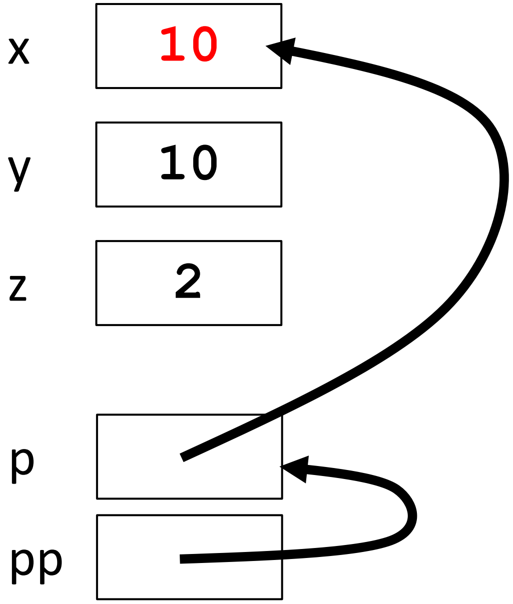
```c
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

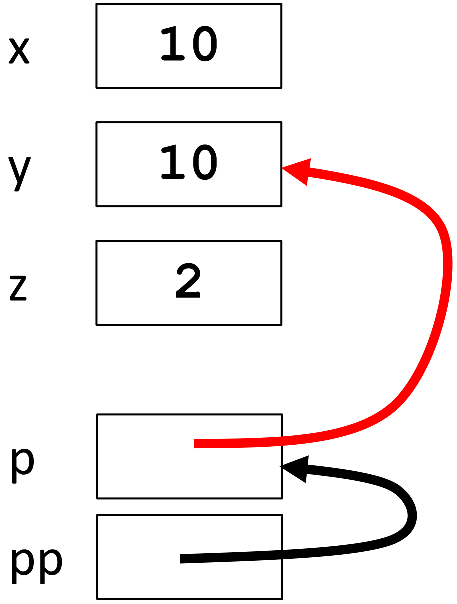
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

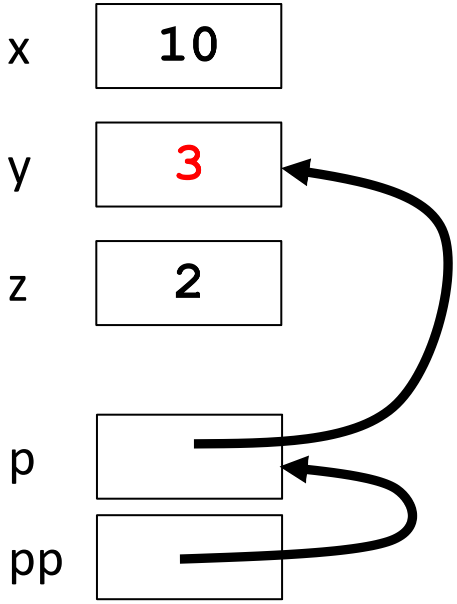
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

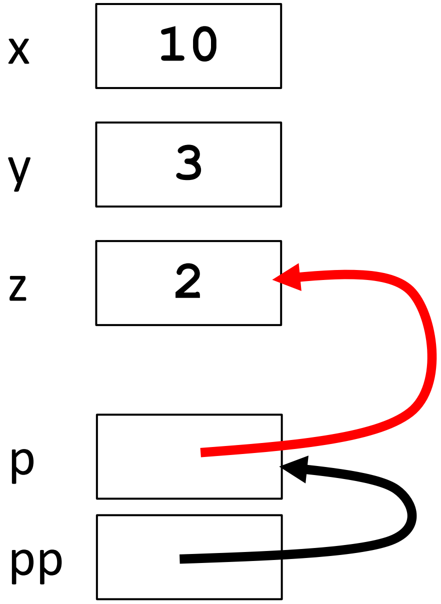
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

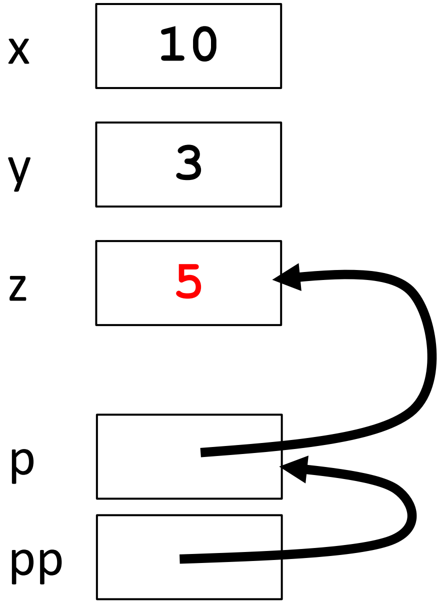
```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



Poll Solution

- What does this program print?

```
int main(int argc, char* argv[]) {
  int x = 3;
  int y = 10;
  int z = 2;

  int *p = &x;
  int **pp = &p; // ptr to a ptr

  *p = 10;
  *pp = &y;
  y = 3;
  p = &z;
  **pp = *p + 3;

  printf("%d %d %d \n", x, y, z);
  return 0;
}
```



NULL

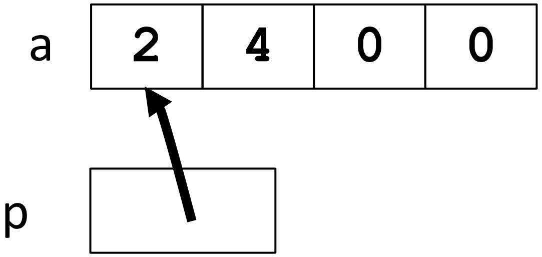
- NULL is a reference that is guaranteed to be invalid
  - an attempt to dereference NULL *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer
  - It's better to cause a segfault than to allow the corruption of memory!
  - If you can't give a pointer an initial value yet, give it NULL!

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

Arrays vs Pointers

- Arrays and pointers are very similar:
  - A pointer can refer “point to “ the first element in an array

```
int a[] = {2, 4, 0, 0};
int *p = a;
```



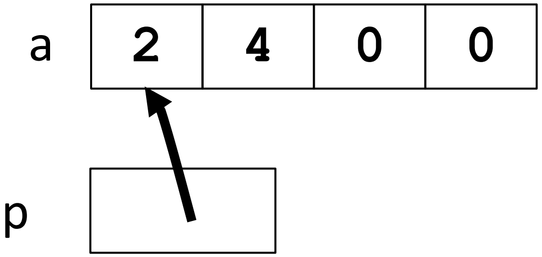
  - Because of this, we can access the “index of” a pointer

```
int a[] = {2, 4, 0, 0};
int *p = a;

printf("%d\n", p[1]); // prints 4
```

Arrays vs Pointers

- Arrays and pointers are very similar:
  - A pointer can refer “point to “ the first element in an array

```
int a[] = {2, 4, 0, 0};
int *p = a;
```



  - Pointers can be reassigned, arrays cannot

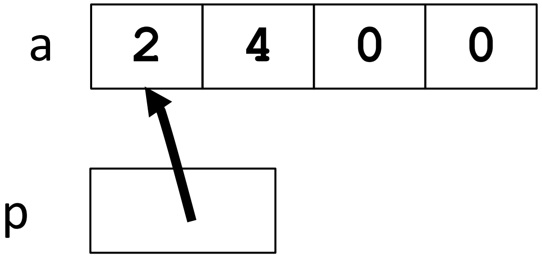
```
int a[] = {2, 4, 0, 0};
int *p = a;
int x = 3;

p = &x;
// a = &x;  // does not compile
p = &(a[1]);

printf("%d\n", p[1]); // prints ???
```

pollev.com/tqm

Arrays vs Pointers

- Arrays and pointers are very similar:
  - A pointer can refer “point to “ the first element in an array

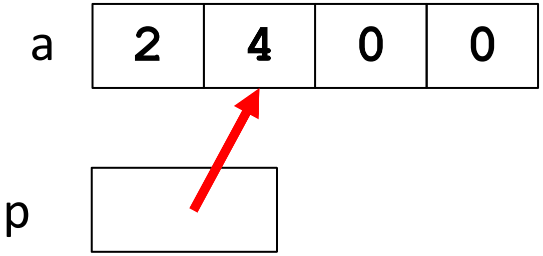
```
int a[] = {2, 4, 0, 0};
int *p = a;
```



  - Pointers can be reassigned, arrays cannot

```
int a[] = {2, 4, 0, 0};
int *p = a;
int x = 3;

p = &x;
// a = &x;  // does not compile
p = &(a[1]);

printf("%d\n", p[1]); // prints ???
```

pollev.com/tqm



Prints "0"

Arrays as Parameters (Pointer Decay)

- 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
  - **Arrays are secretly passed as pointers to the 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 using pointer arithmetic
  - E.g.

```
ptr[3] = ...;
```

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 use an 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
  - How else would printf know how to stop printing?

```
char str[] = "Hello";
```

- Example:
  - Takes up 6 characters, 5 for “Hello” and 1 for the null terminator

value	'H'	'e'	'l'	'l'	'o'	'\0'

Poll Everywhere

Discuss

- What are all the bugs in this code?

```
int main(int argc, char* argv[]) {
  // TODO
  char str[] = "Ho";
  char* str_two = "Hi";

  printf("%s and %s\n", str, str_two);

  str = str_two;
  str_two = "Hey";

  printf("%s and %s\n", str, str_two);

  char arr[2];
  arr[0] = 'y';
  arr[1] = 'a’;
  printf("%s\n", arr);

  return 0;
}
```

Lecture Outline

- C Intro
  - Cont. from last time:
    - Arrays
    - Command line args
  - Pointers
    - Box & Arrow Diagrams
    - Arrays vs pointers
    - C Strings
  - ~~Structs~~
  - ~~The Stack & Pass-by-value~~
  - **More on Compiling**

Compilation: Basics & Running

- As we saw last time, we need to compile our code in this class.

- We use **`clang-15`** in this class
- Simplest Compilation: `clang-15 example.c`
  - By default produces an executable called `a.out`

- If we want to run the executable, we type: `./a.out`
  - If it was in a directory (called "test" for example)

```
./test/a.out
```

  - The first `.` is used to say "start looking in the current directory"

Compilation: Options

- a.out is not what we usually want to call our programs

- Can use the compiler flag `-o`
  to specify what the output should be called
  - After the –o, (letter o) need to specify what we want the output to be called

- If we want to compile the file `hello.c` into an executable called `hello`, we can do:

```
clang-15 -o hello hello.c
```

Compilation: More Options

- We will eventually use a debugger, covered more in a later class. Add the `-g3` flag to have the compiler output have the maximum debugging info

- Compiler is pretty good at telling us when something looks wrong. To turn on "all" warnings, use `-Wall`
  - Not "all" warnings.
  - Wall stands for **W**arnings **all**

- If we want to compile the file `hello.c` into an executable called `hello`, with these options we can do:

```
clang-15 -g3 -Wall -o hello hello.c
```

How to Read (warnings)

- You should fix all warnings you have during compilation, autograder will deduct for warnings AND errors

- Some warnings may have "cascading effects"
  - Try fixing them top to bottom
  - If you fix one error and still have more errors, try recompiling and see if the first error fixed others

- Demo: missing_semi.c

How to Read (warnings)

- Demo: missing_semi.c

```
missing_semi.c:22:18: error: expected ';' at end of declaration
  album copy = *a
                 ^

                 ;
missing_semi.c:42:23: error: use of undeclared identifier 'a1'
  printf("a1 = %s\n", a1);
                      ^
missing_semi.c:45:50: error: no member named 'title' in 'album'
  printf("madvillainy.title = %s\n", madvillainy.title.data);
                                     ~~~~~~~~~~~ ^

3 errors generated.
```

- General Structure:

```
file_name.c  :  line num : column num : error/warning description
      line from source code
```

Poll Everywhere

Discuss

- What’s this trying to say? How do you think we should fix it?

```
mystery.c:7:9: error: expected ';' after top level declarator
} string
        ^

        ;
mystery.c:19:8: warning: missing terminating '"' character [-
Winvalid-pp-token]
  mf = "DOOM;
       ^
mystery.c:19:8: error: expected expression
mystery.c:23:5: warning: expression result unused [-Wunused-value]
  x + 2;
  ~ ^ ~
mystery.c:29:1: warning: type specifier missing, defaults to 'int';
ISO C99 and later do not support implicit int [-Wimplicit-int]
weird_func() {
^
int
3 warnings and 2 errors generated.
```

Action Items

- Get things setup
- HW00
- Check-in00
- Pre-semester Survey