

Memory and Arrays

CSE 333 Summer 2020

Welcome – please set up your Zoom session. We'll start the actual class meeting at 10:50 am pdt

Instructor: Travis McGaha

Teaching Assistants:

Jeter Arellano

Ramya Challa

Kyrie Dowling

Ian Hsiao

Allen Jung

Sylvia Wang



pollev.com/cse33320su

About how long did Exercise 0 take?

- A. 0-1 Hours
- B. 1-2 Hours
- C. 2-3 Hours
- D. 3-4 Hours
- E. 4+ Hours
- F. I didn't submit / I prefer not to say

Administrivia (1)

- ❖ Exercise 0 due this morning*

 - *but if you're still getting your Linux setup working, finish ex0 and submit it in the next day or so. At latest, Friday (hopefully).

 - Sample solution will be posted over the weekend

- ❖ Exercise 1 out today, due Friday morning, 10:30 am**

 - ** We'll cut off submissions Saturday mid-day so we can post sample solutions late Saturday for you to check before ex2 is due on Monday.

 - You'll want to check your code style for this and later exercises using the clint tool distributed with hw0 and described there

- ❖ If you don't have a gradescope account, send a note to cse333-staff (or a private ed post) with your name, student id# and uw email address (xyzy@uw.edu) so we can get you set up

Administrivia (2)

- ❖ Office hours: See the Website, Canvas calendar or Canvas/Zoom meeting list for times/days and Zoom links.
 - Probably will be more additions or changes over next several days
 - Use the discussion board to report problems or contribute suggestions

- ❖ Sections: TA assignments posted on website. Links to section zoom meetings on canvas (May be put on website)
 - There might be some moving around on the admin side of things. Remember you can show up to any section if you need to.

Administrivia (3)

- ❖ Homework 0 out last night
 - Logistics and infrastructure for projects
 - Gitlab email sent last night when repos created – no action needed
 - Demos and setup in section tomorrow – we're still figuring out how best to do this, but we will have sections tomorrow. See Canvas calendar or Canvas/Zoom page for Zoom links
 - Please email us or make a private ed post if you are missing a repo or hw0 files.
- ❖ Reference system for grading is CSE lab/attu/current VM
 - For both exercises and homework (project) code
 - It's your job to be sure your solution(s) work on them
 - Just because it works on ReallyCoolLinuxDistribution[®] doesn't mean it for sure works on other Linux systems, including ours.

Administrivia (4)

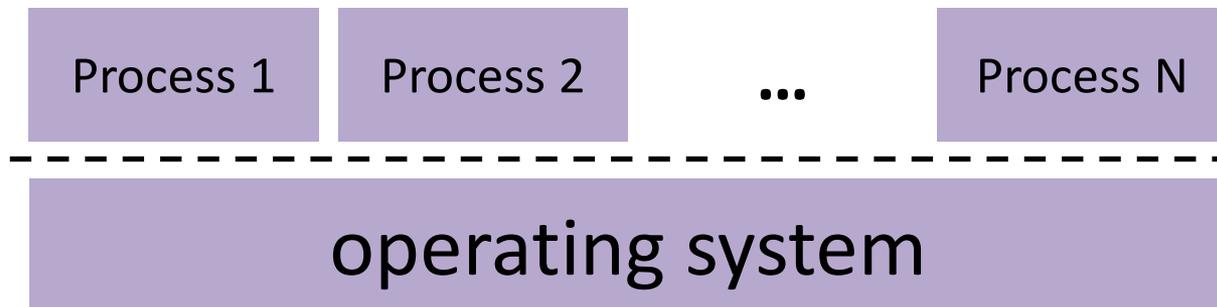
- ❖ Pre-quarter survey out
 - Anonymous survey to collect information about how to best handle things this quarter.
 - Due Friday June 26th at 11:59 pm.

Lecture Outline

- ❖ **C's Memory Model** (CSE 351 refresher)
- ❖ Pointers (CSE 351 refresher)
- ❖ Arrays

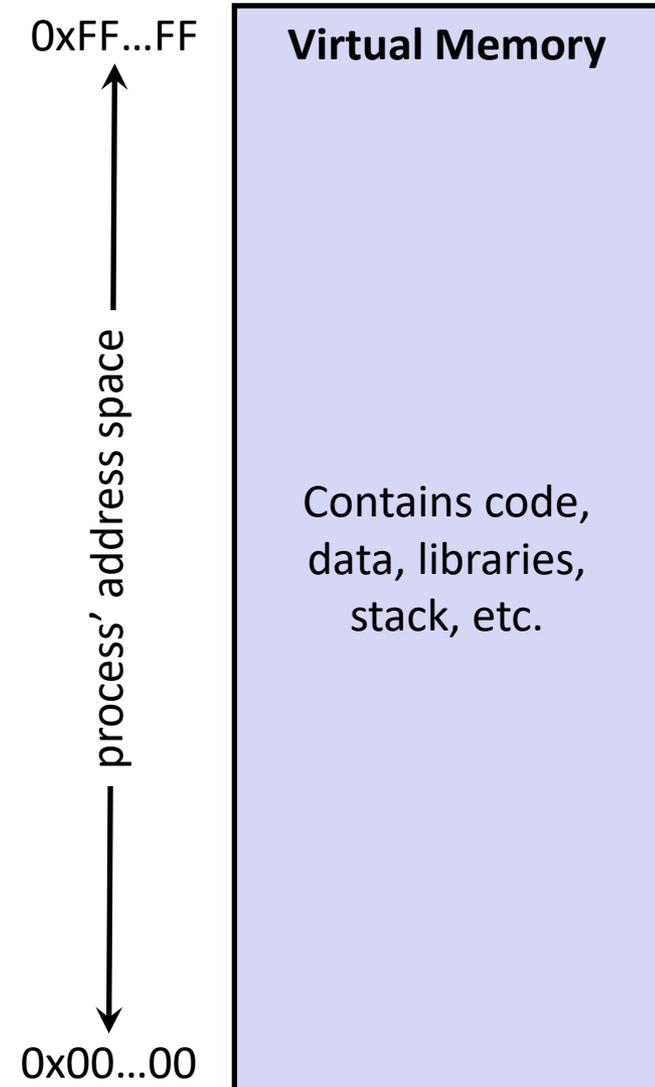
OS and Processes

- ❖ The OS lets you run multiple applications at once
 - An application runs within an OS “process”
 - The OS timeslices each CPU between runnable processes
 - This happens *very quickly*: ~100 times per second



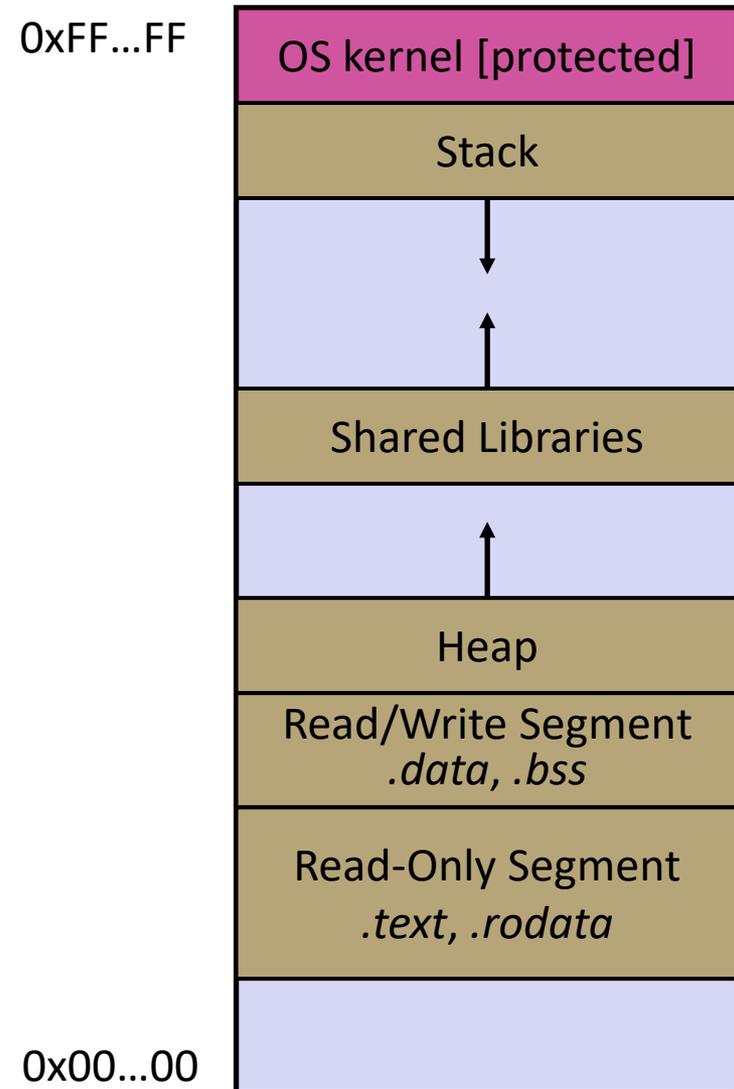
Processes and Virtual Memory

- ❖ The OS gives each process the illusion of its own private memory
 - Called the process' **address space**
 - Contains the process' virtual memory, visible only to it (via translation)
 - 2^{64} bytes on a 64-bit machine



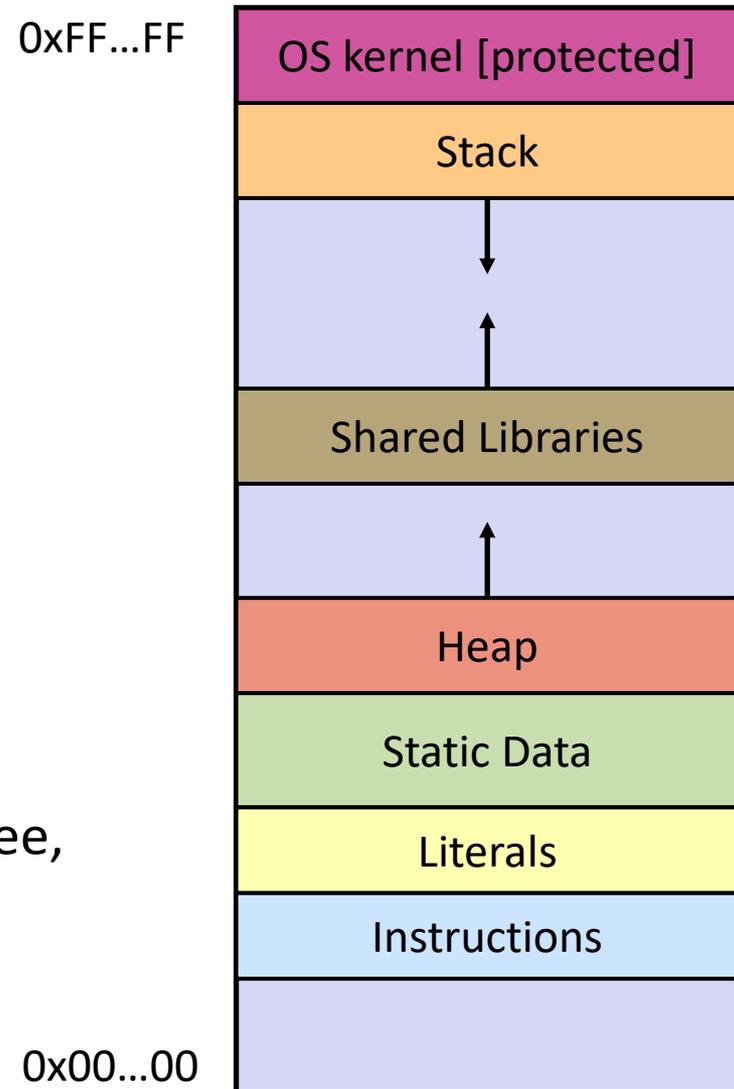
Loading

- ❖ When the OS loads a program it:
 - 1) Creates an address space
 - 2) Inspects the executable file to see what's in it
 - 3) (Lazily) copies regions of the file into the right place in the address space
 - 4) Does any final linking, relocation, or other needed preparation



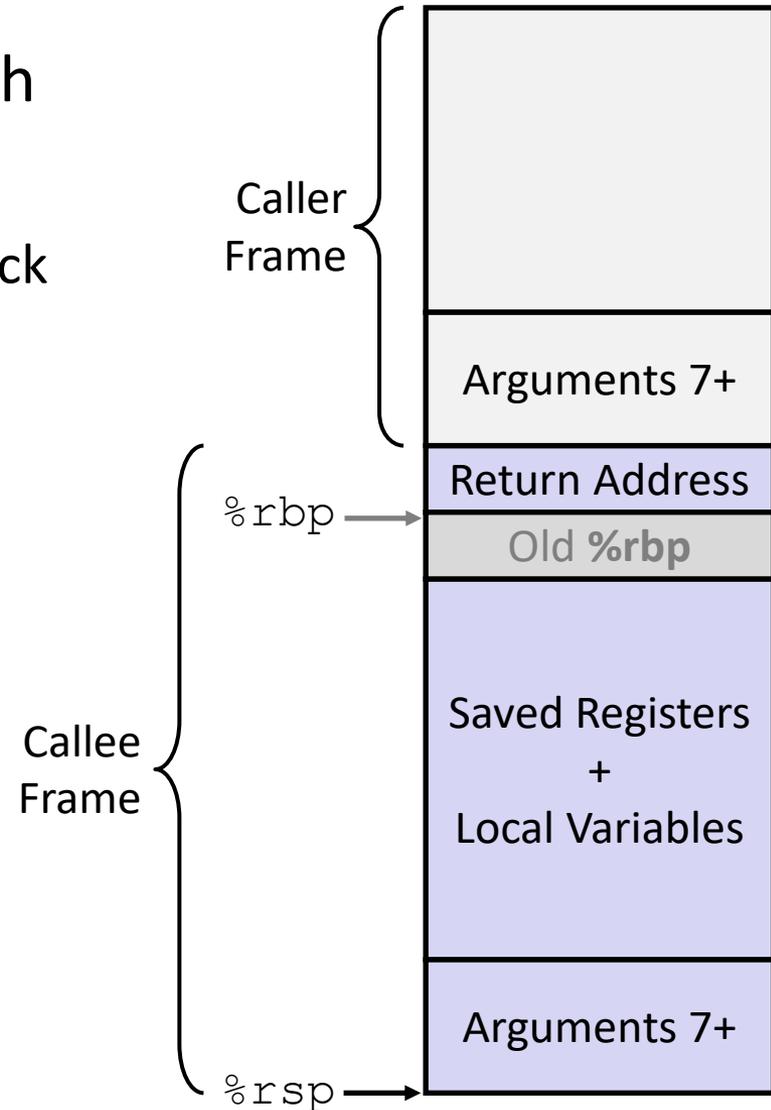
Memory Management

- ❖ *Local* variables on the Stack
 - Allocated and freed via calling conventions (`push`, `pop`, `mov`)
- ❖ *Global* and *static* variables in Data
 - Allocated/freed when the process starts/exits
- ❖ *Dynamically-allocated* data on the Heap
 - `malloc()` to request; `free()` to free, otherwise **memory leak**
 - More about this on Monday



Review: The Stack

- ❖ Used to store data associated with function calls
 - Compiler-inserted code manages stack frames for you
- ❖ Stack frame (x86-64) includes:
 - Address to return to
 - Saved registers
 - Based on calling conventions
 - Local variables
 - Argument build
 - Only if > 6 used



Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

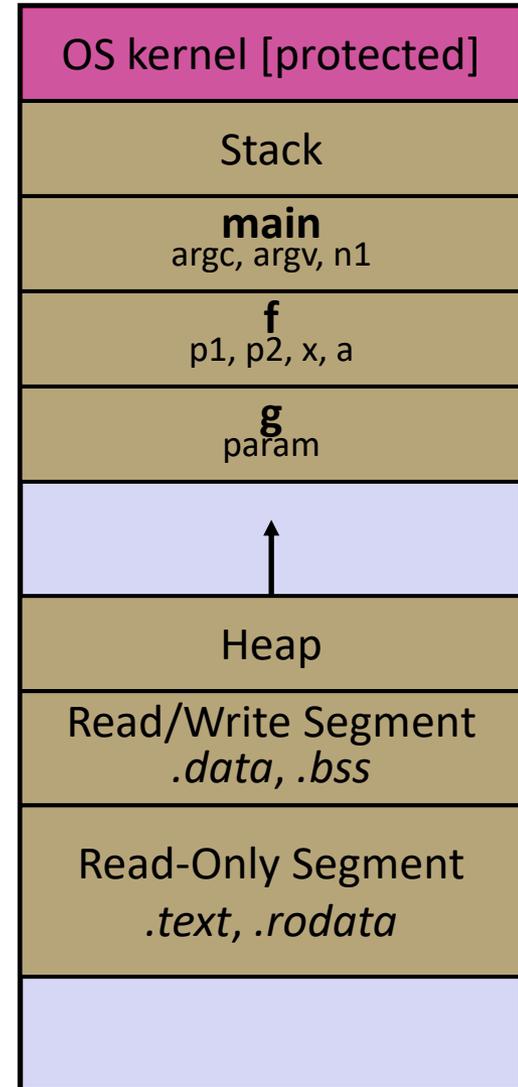
```
#include <stdint.h>

int f(int, int);
int g(int);

→ int main(int argc, char** argv) {
→   int n1 = f(3, -5);
   n1 = g(n1);
}

→ int f(int p1, int p2) {
   int x;
   int a[3];
   ...
→   x = g(a[2]);
   return x;
}

→ int g(int param) {
→   return param * 2;
}
```



Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

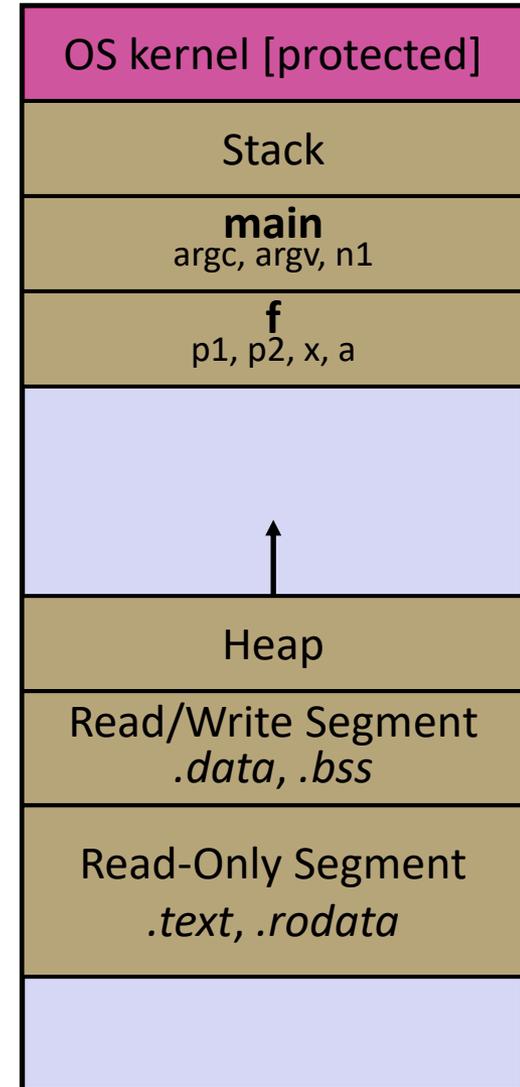
```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

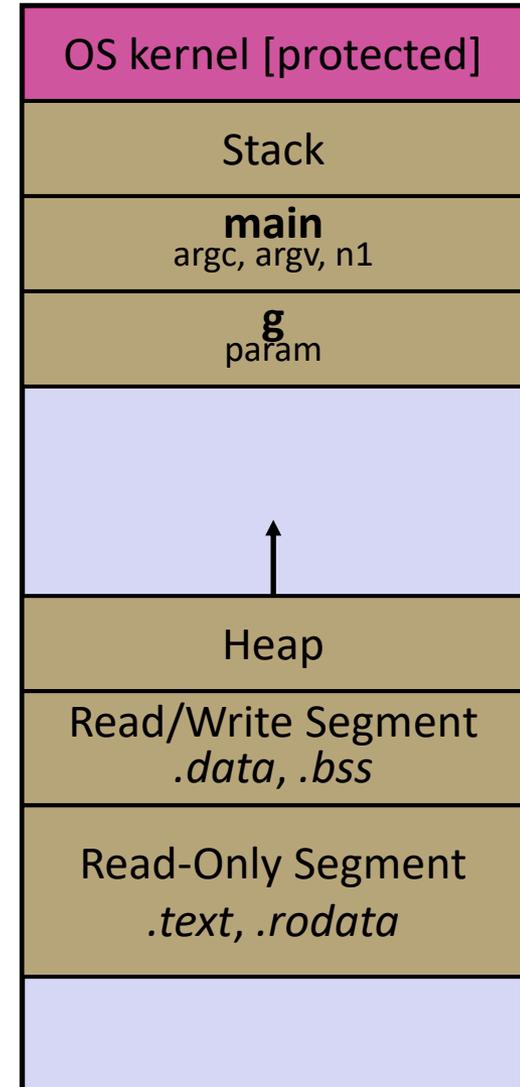
```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



Stack in Action

Note: arrow points to *next* instruction to be executed (like in gdb).

stack.c

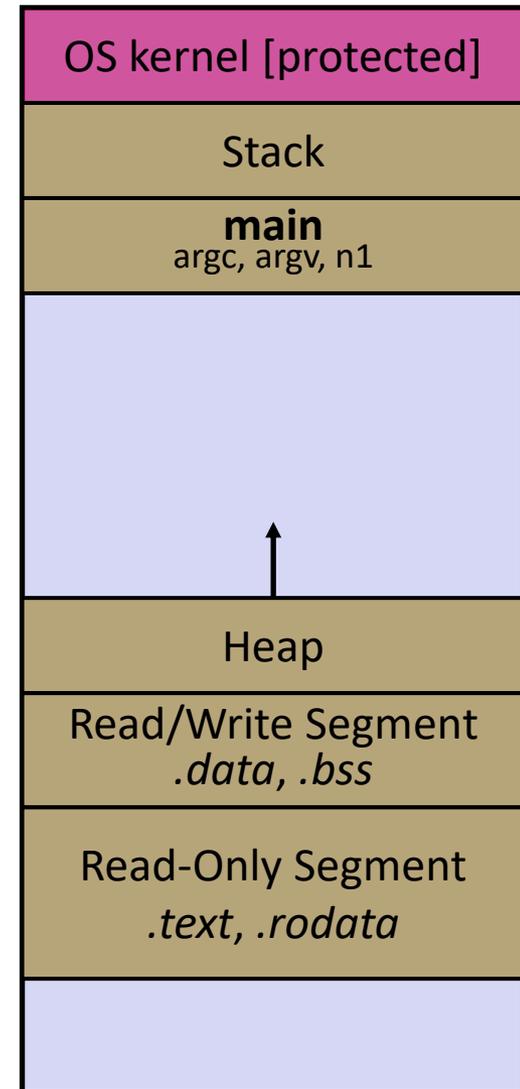
```
#include <stdint.h>

int f(int, int);
int g(int);

int main(int argc, char** argv) {
    int n1 = f(3, -5);
    n1 = g(n1);
}

int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

int g(int param) {
    return param * 2;
}
```



Lecture Outline

- ❖ C's Memory Model (refresher)
- ❖ **Pointers** (refresher)
- ❖ Arrays



Pointers

Try to be consistent
with ptr declaration
syntax

❖ Variables that store addresses

- It points to somewhere in the process' virtual address space
- `&foo` produces the virtual address of `foo`

equivalent

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

- Recommended: do not define multiple pointers on same line:

`int *p1, p2;`

not the same as

`int *p1, *p2;`

- Instead, use:

`int *p1;`

`int *p2;`

❖ *Dereference* a pointer using the unary `*` operator

- Access the memory referred to by a pointer

Pointer Example

pointy.c

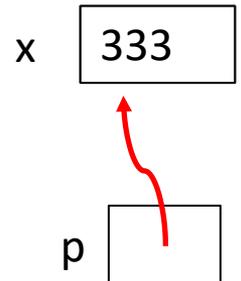
```
#include <stdio.h>
#include <stdint.h>

int main(int argc, char** argv) {
    int x = 351;
    int* p;    // p is a pointer to a int

    p = &x;    // p now contains the addr of x
    printf("&x is %p\n", &x);
    printf(" p is %p\n", p);
    printf(" x is %d\n", x);

    *p = 333;  // change value of x
    printf(" x is %d\n", x);

    return 0;
}
```



Something Curious

- ❖ What happens if we run `pointy.c` several times?

```
bash$ gcc -Wall -std=c11 -o pointy pointy.c
```

Run 1:

```
bash$ ./pointy
&x is 0x7ffff9e28524
p is 0x7ffff9e28524
x is 351
x is 333
```

Run 2:

```
bash$ ./pointy
&x is 0x7ffffe847be34
p is 0x7fffe847be34
x is 351
x is 333
```

Run 3:

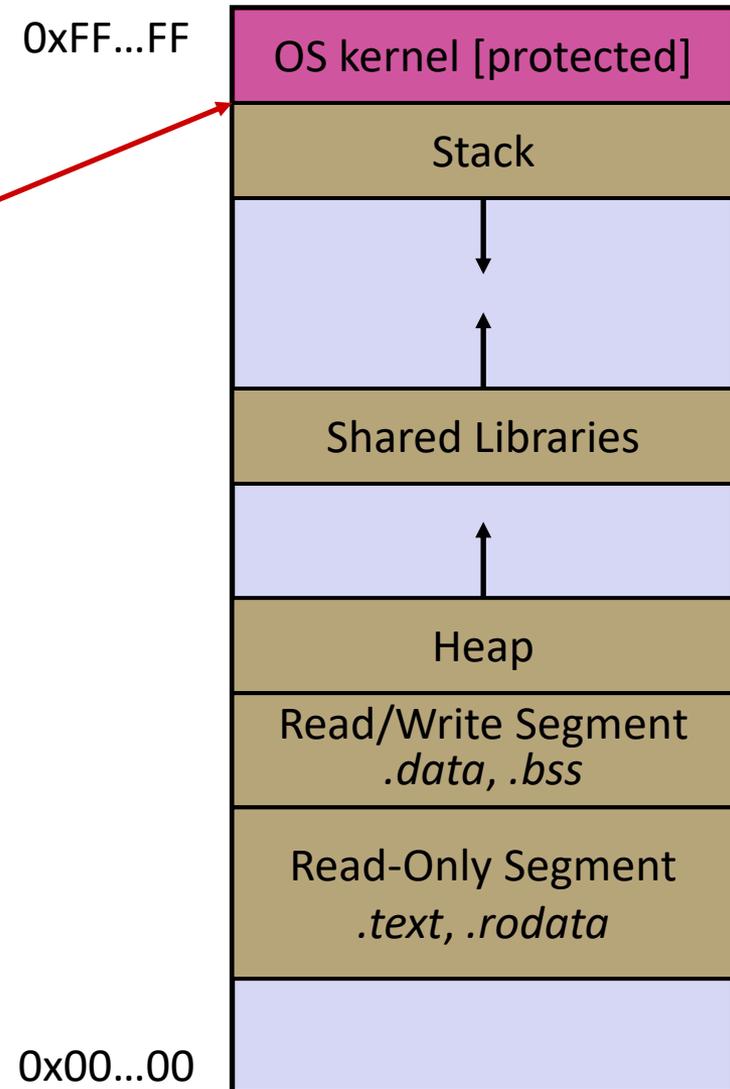
```
bash$ ./pointy
&x is 0x7ffffe7b14644
p is 0x7fffe7b14644
x is 351
x is 333
```

Run 4:

```
bash$ ./pointy
&x is 0x7ffffff0dfe54
p is 0x7fffff0dfe54
x is 351
x is 333
```

Address Space Layout Randomization

- ❖ Linux uses *address space layout randomization* (ASLR) for added security
 - Randomizes:
 - Base of stack
 - Shared library (`mmap`) location
 - Makes Stack-based buffer overflow attacks tougher
 - Makes debugging tougher
 - Can be disabled (`gdb` does this by default); Google if curious



Box-and-Arrow Diagrams

boxarrow.c

```
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address

name	value
------	-------

Box-and-Arrow Diagrams

boxarrow.c

```
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	name	value
---------	------	-------

&x	x	value
&arr[2]	arr[2]	value
&arr[1]	arr[1]	value
&arr[0]	arr[0]	value
&p	p	value

stack frame for main()

Box-and-Arrow Diagrams

boxarrow.c

```
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	name	value
	x	1
&arr[2]	arr[2]	4
&arr[1]	arr[1]	3
&arr[0]	arr[0]	2
&p	p	&arr[1]

Box-and-Arrow Diagrams

boxarrow.c

```
int main(int argc, char** argv) {
    int x = 1;
    int arr[3] = {2, 3, 4};
    int* p = &arr[1];

    printf("&x: %p; x: %d\n", &x, x);
    printf("&arr[0]: %p; arr[0]: %d\n", &arr[0], arr[0]);
    printf("&arr[2]: %p; arr[2]: %d\n", &arr[2], arr[2]);
    printf("&p: %p; p: %p; *p: %d\n", &p, p, *p);

    return EXIT_SUCCESS;
}
```

address	name	value
0x7fff...4c	x	1
0x7fff...48	arr[2]	4
0x7fff...44	arr[1]	3
0x7fff...40	arr[0]	2
0x7fff...38	p	0x7fff...74

p: just an address
*p: get the data at the address

 **Poll Everywhere**pollev.com/cse33320su

❖ When run, what does this code print?

- A. 2
- B. 333
- C. 999
- D. A return address
- E. I don't know

```
int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```

Answer

local_addr.c

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

int64_t foo();
int64_t bar(int64_t);

int main() {
    int64_t* ptr = foo();
    int64_t x = bar(2);
    printf("%d\n", *ptr);
}

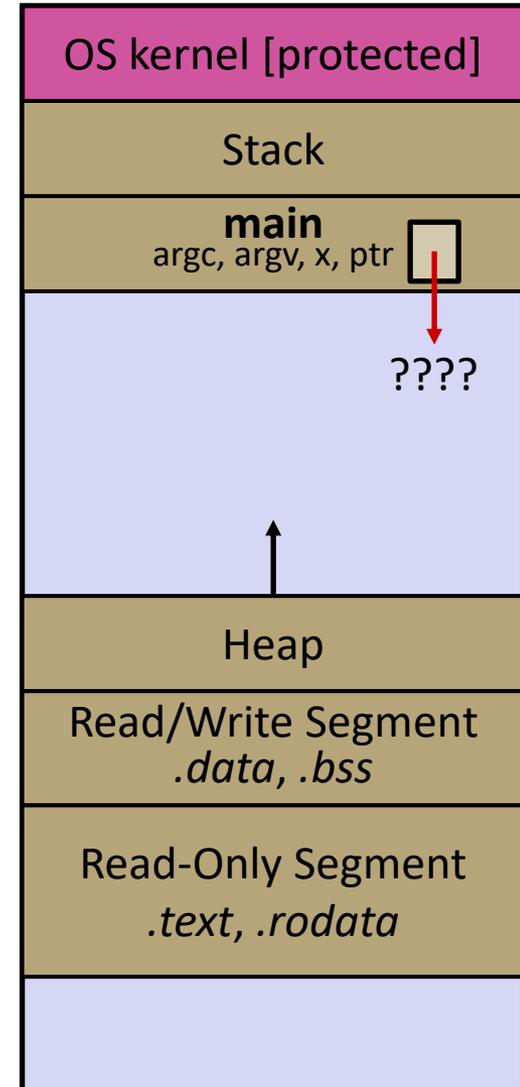
int64_t* foo() {
    int64_t x = 333;
    x += bar(x);
    return &x;
}

int64_t bar(int64_t param) {
    return param * 2;
}
```



Answer: Undefined Behaviour

E. I don't know



Lecture Outline

- ❖ C's Memory Model (refresher)
- ❖ Pointers (refresher)
- ❖ **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!
 - `sizeof (array)` only works in variable scope of array definition
- Recent versions of C (but *not* C++) allow for variable-length arrays
 - Uncommon and can be considered bad practice [*we won't use*]

```
int n = 175;
int scores[n]; // OK in C99
```

Using Arrays

❖ 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!
```

 **Poll Everywhere**pollev.com/cse33320su

- ❖ The code snippets both use a variable-length array. What will happen when we compile with C99?
 - Vote at <http://PollEv.com/cse33320su>

```
int m = 175;
int scores[m];

void foo(int n) {
    ...
}
```

- A. **Compiler Error**
- B. **Compiler Error**
- C. **No Error**
- D. **No Error**
- E. **We're lost...**

```
int m = 175;

void foo(int n) {
    int scores[n];
    ...
}
```

- Compiler Error**
- No Error**
- Compiler Error**
- No Error**

Multi-dimensional Arrays

❖ Generic 2D format:

```
type name[rows][cols] = {{values}, ..., {values}};
```

- Still allocates a single, contiguous chunk of memory
- C is *row-major*

```
// a 2-row, 3-column array of doubles
double grid[2][3];

// a 3-row, 5-column array of ints
int matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};
```

- 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

```
int sumAll(int a[]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    return 0;
}

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

Solution 1: Declare Array Size

```
int sumAll(int a[5]); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers);
    printf("sum is: %d\n", sum);
    return 0;
}

int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++) {
        sum += a[i];
    }
    return sum;
}
```

- ❖ Problem: loss of generality/flexibility

Solution 2: Pass Size as Parameter

```
int sumAll(int a[], int size); // prototype

int main(int argc, char** argv) {
    int numbers[] = {9, 8, 1, 9, 5};
    int sum = sumAll(numbers, 5);
    printf("sum is: %d\n", sum);
    return 0;
}

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

arraysum.c

- Standard idiom in C programs

Parameters: reference vs. value

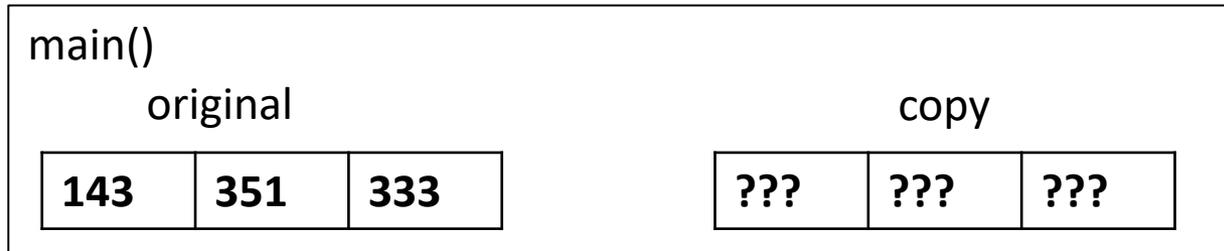
- ❖ There are two fundamental parameter-passing schemes in programming languages
- ❖ **Call-by-value**
 - Parameter is a local variable initialized with a copy of the calling argument when the function is called; manipulating the parameter only changes the copy, *not* the calling argument
 - **C, Java, C++** (most things)
- ❖ **Call-by-reference**
 - Parameter is an alias for the supplied argument; manipulating the parameter manipulates the calling argument
 - C++ references (we'll see these later)

So what's the story for arrays?

- ❖ Is it call-by-value or call-by-reference?
- ❖ Technical answer: a $T[]$ array parameter is “promoted” to a pointer of type T^* , and the *pointer* is passed by value
 - So it acts like a call-by-reference array (if callee changes the array parameter **elements** it changes the caller's array)
 - But it's really a call-by-value pointer (the callee can change the **pointer parameter** to point to something else(!))
 - This is because $\mathbf{T[i]}$ is really $\mathbf{* (T+i)}$. We aren't changing \mathbf{T} !

```
void copyArray(int src[], int dst[], int size) {
    int i;
    dst = src;    // evil!
    for (i = 0; i < size; i++) {
        dst[i] = src[i];    // copies source array to itself!
    }
}
```

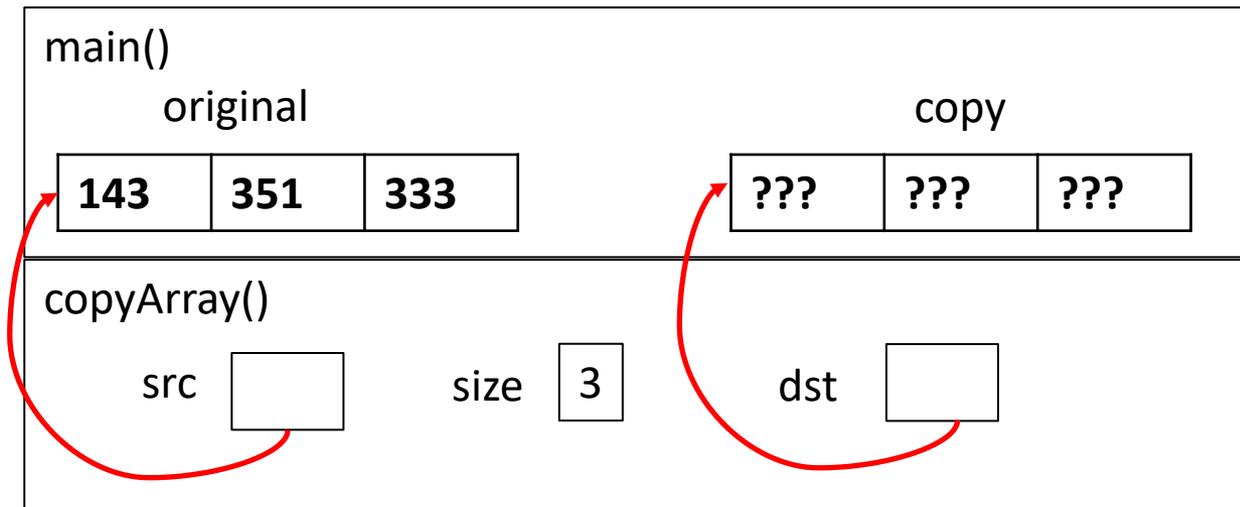
Array Memory Diagram



```
int main() {
    int original[] = {143, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

Array Memory Diagram

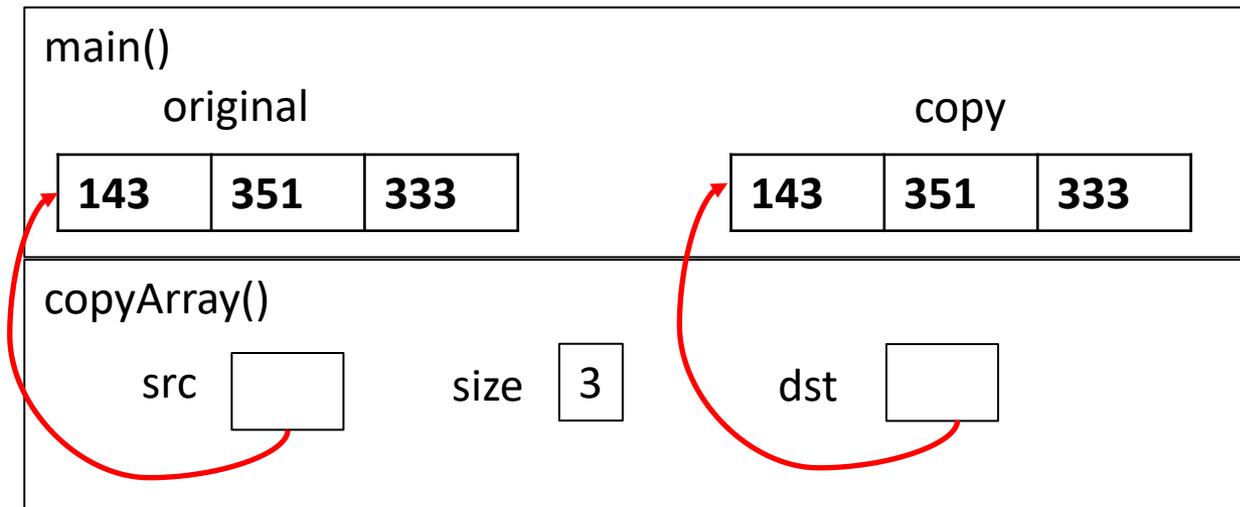


```
int main() {
    int original[] = {143, 351, 333};
    int copy[3];
    copyArray(original, copy, 3);
}

void copyArray(int src[], int dst[], int size) {
    for (int i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```

dst[i] is really
***(dst+i)**. We
aren't changing **dst**!

Array Memory Diagram



```
int main() {  
    int original[] = {143, 351, 333};  
    int copy[3];  
    copyArray(original, copy, 3);  
}  
  
void copyArray(int src[], int dst[], int size) {  
    for (int i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
}
```

dst[i] is really
***(dst+i)**. We
aren't changing **dst**!

Returning an Array

- ❖ Local variables, including arrays, are allocated on the Stack
 - They “disappear” when a function returns!
 - Can’t safely return local arrays from functions
 - Can’t return an array as a return value – why not?

```
int* copyArray(int src[], int size) {  
    int i, dst[size];    // OK in C99  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
  
    return dst;    // no compiler error, but wrong!  
}
```

buggy_copyarray.c

Solution: Output Parameter

- ❖ Create the “returned” array in the caller
 - Pass it as an **output parameter** to `copyarray()`
 - A pointer parameter that allows the called function to store values that the caller can use
 - Works because arrays are “passed” as pointers

```
void copyArray(int src[], int dst[], int size) {  
    int i;  
  
    for (i = 0; i < size; i++) {  
        dst[i] = src[i];  
    }  
}
```

`copyarray.c`

Output Parameters

❖ Output parameters are common in library functions

- `long int strtol(char* str, char** endptr, int base);`
- `int sscanf(char* str, char* format, ...);`

```
int    num, i;
char*  pEnd, str1 = "333 rocks";
char   str2[10];

// converts "333 rocks" into long -- pEnd is conversion end
num = (int) strtol(str1, &pEnd, 10);

// reads string into arguments based on format string
num = sscanf("3 blind mice", "%d %s", &i, str2);
```

outparam.c

Extra Exercises

- ❖ Some lectures contain “Extra Exercise” slides
 - Extra practice for you to do on your own without the pressure of being graded
 - You may use libraries and helper functions as needed
 - Early ones may require reviewing 351 material or looking at documentation for things we haven’t discussed in 333 yet
 - Always good to provide test cases in `main()`

- ❖ Solutions for these exercises will be posted on the course website
 - You will get the most benefit from implementing your own solution before looking at the provided one

Extra Exercise #1

- ❖ Write a function that:
 - Accepts an array of 32-bit unsigned integers and a length
 - Reverses the elements of the array in place
 - Returns nothing (`void`)

Extra Exercise #2

- ❖ Write a function that:
 - Accepts a string as a parameter
 - Returns:
 - The first white-space separated word in the string as a newly-allocated string
 - AND the size of that word
 - (probably need to wait until we look at malloc/free later)