

Memory, Pointers, Arrays

CSE 333 Winter 2020

Instructor: Justin Hsia

Teaching Assistants:

Andrew Hu

Austin Chan

Brennan Stein

Cheng Ni

Cosmo Wang

Diya Joy

Guramrit Singh

Mengqi Chen

Pat Kosakanchit

Rehaan Bhimar

Renshu Gu

Travis McGaha

Zachary Keyes

Administrivia

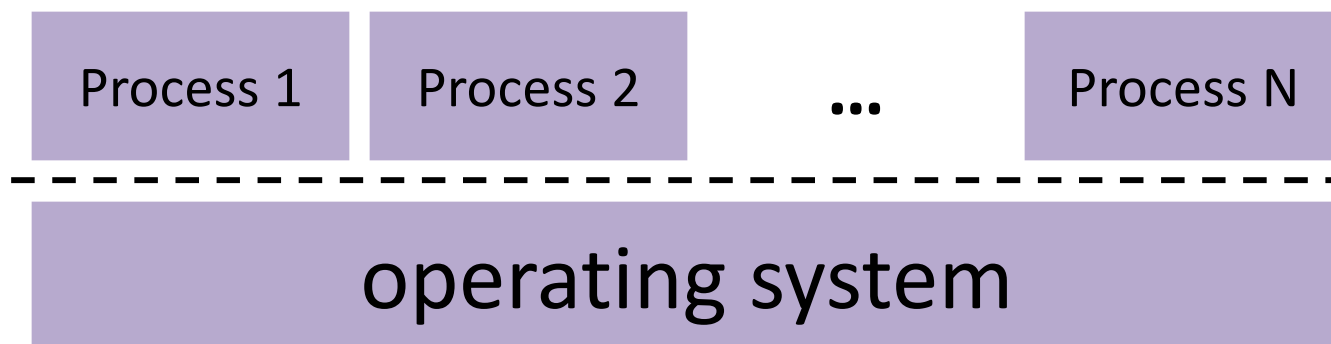
- ❖ Pre-Course Survey & Mini-Bio due tomorrow night
- ❖ Exercise 0 was due this morning
 - Solutions posted this afternoon, grades back early next week
 - Some output newly-visible on Gradescope
 - If you haven't been added to Gradescope yet, email your ex0 submission to Justin ASAP
- ❖ Exercise 1 out today and due Friday morning @ 11 am
- ❖ Homework 0 released today
 - Logistics and infrastructure for homework
 - **Set up GitLab (and VM) before section** – bring laptop if any issues
 - Using the **20wi** CSE VM this quarter

Lecture Outline

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

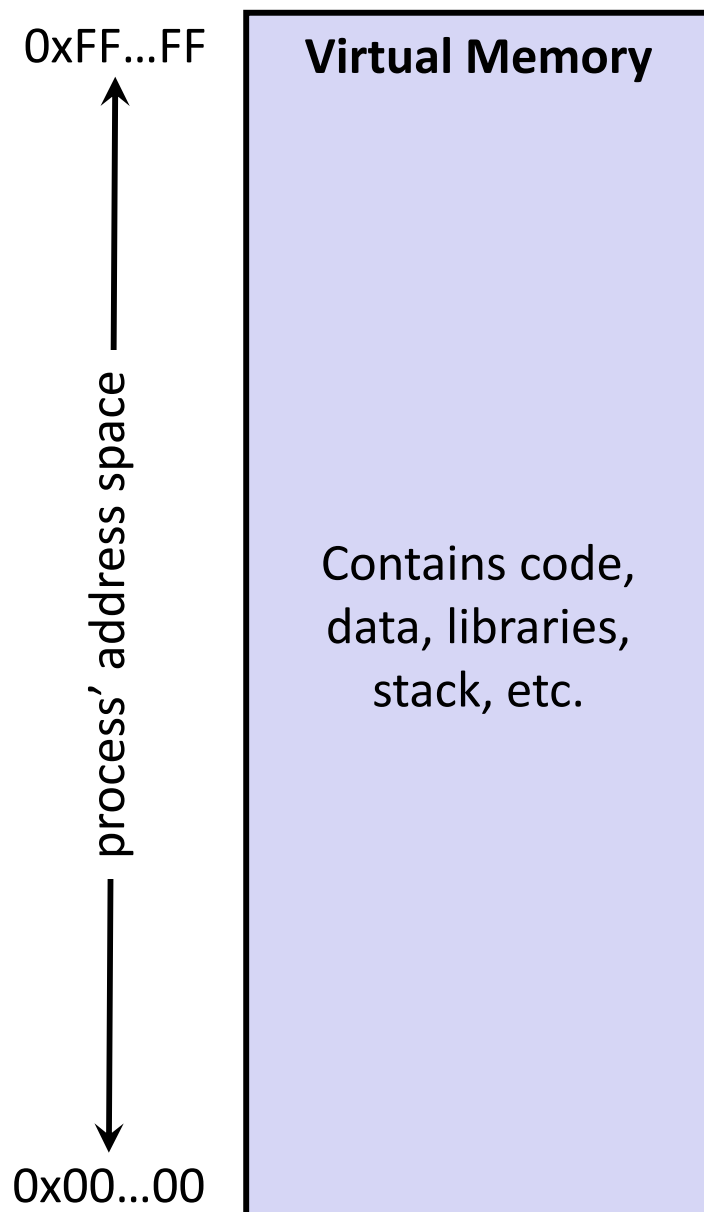
OS and Processes

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



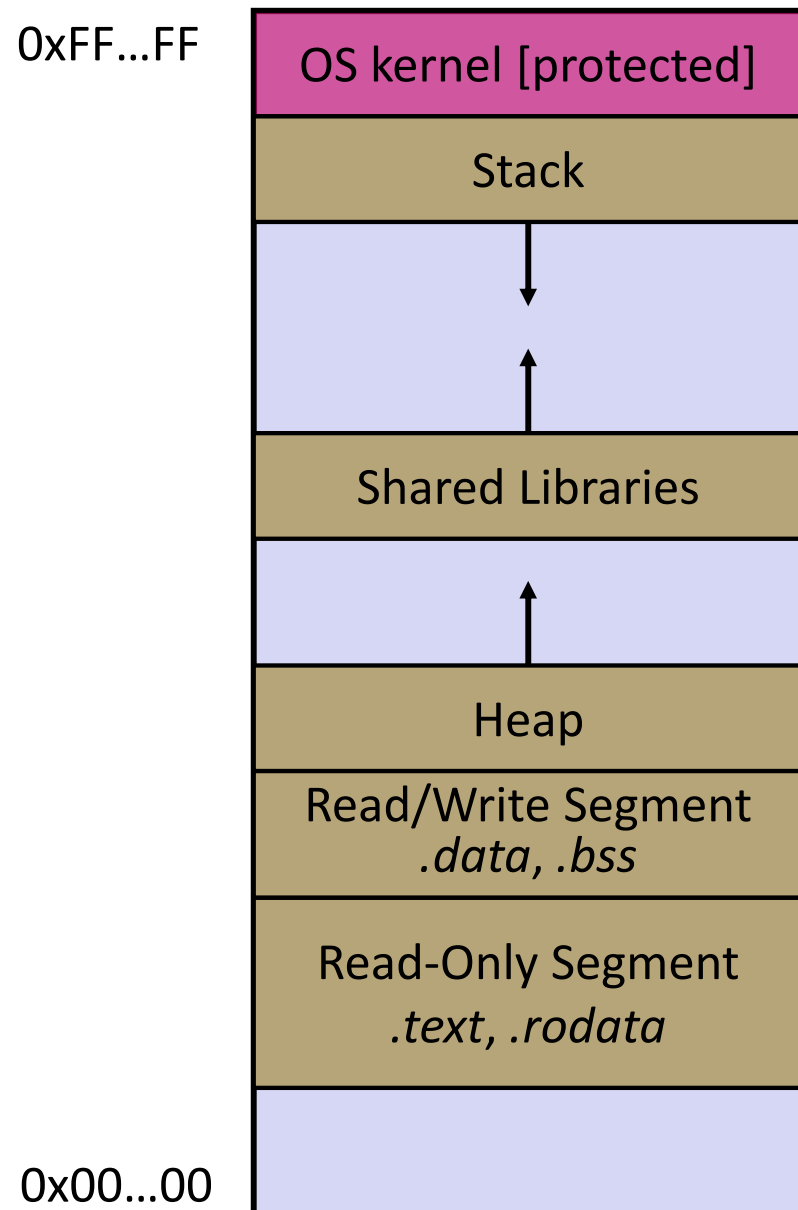
Processes and Virtual Memory

- ❖ 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



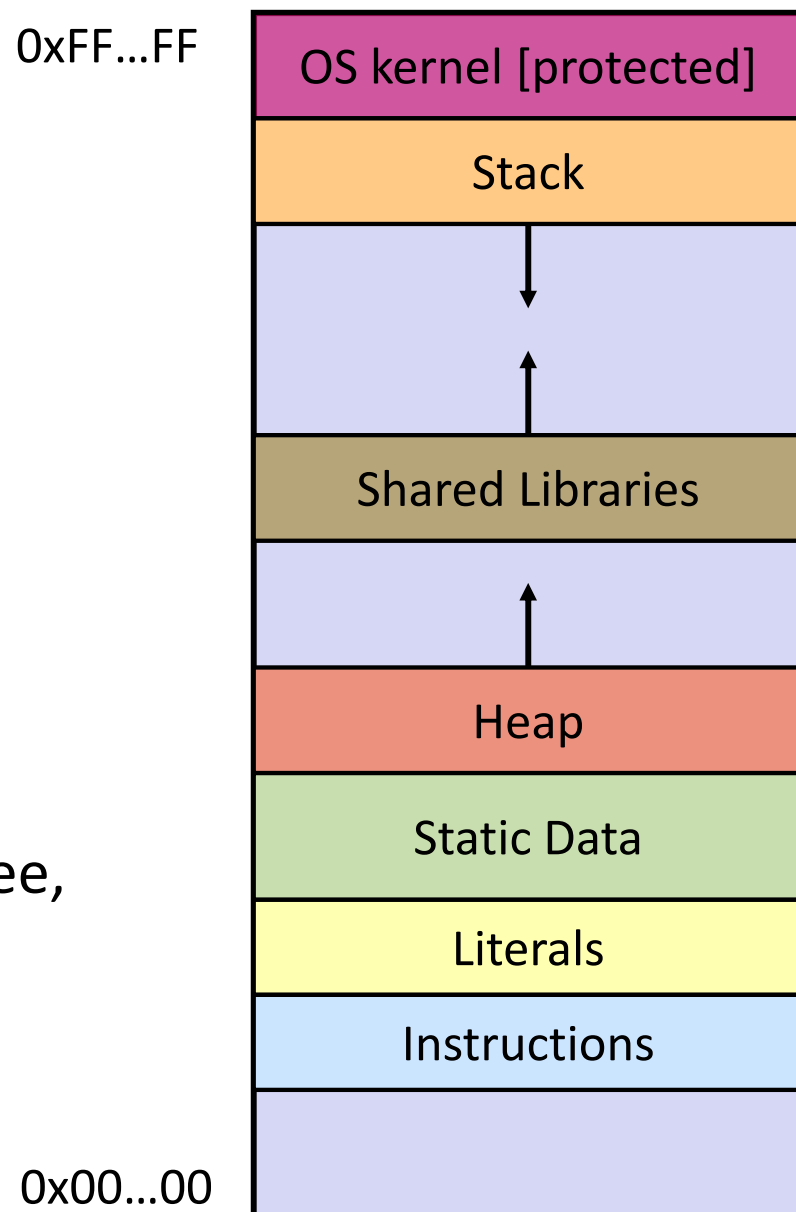
Loading

- ❖ 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



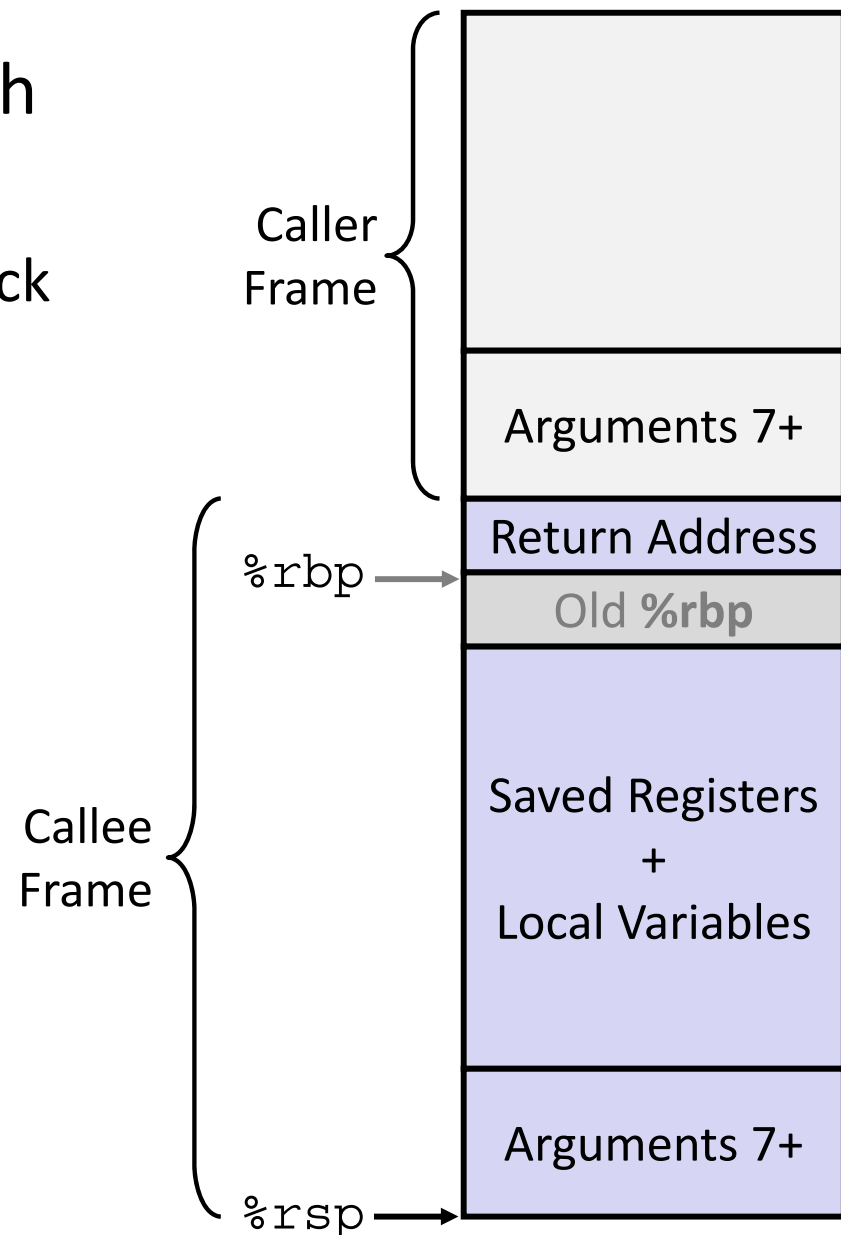
Memory Management

- ❖ *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**



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

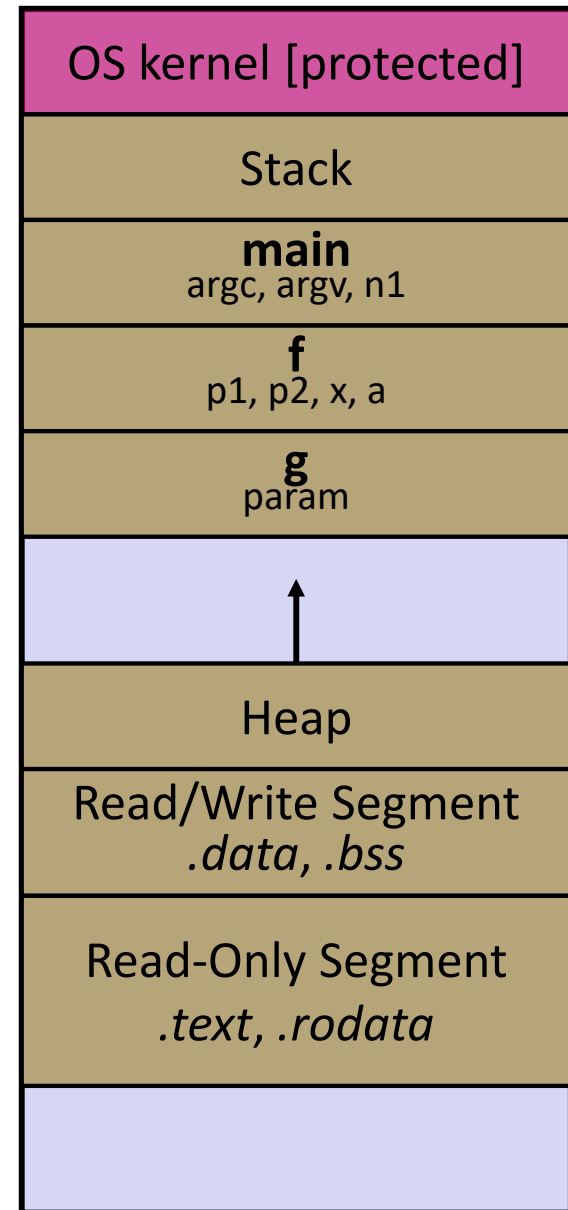
```

int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
    int32_t n1 = f(3, -5);
    n1 = g(n1);
    return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
    int32_t x;
    int32_t a[3];
    ...
    x = g(a[2]);
    return x;
}

int32_t g(int32_t param) {
    return param * 2;
}
    
```



Stack in Action

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

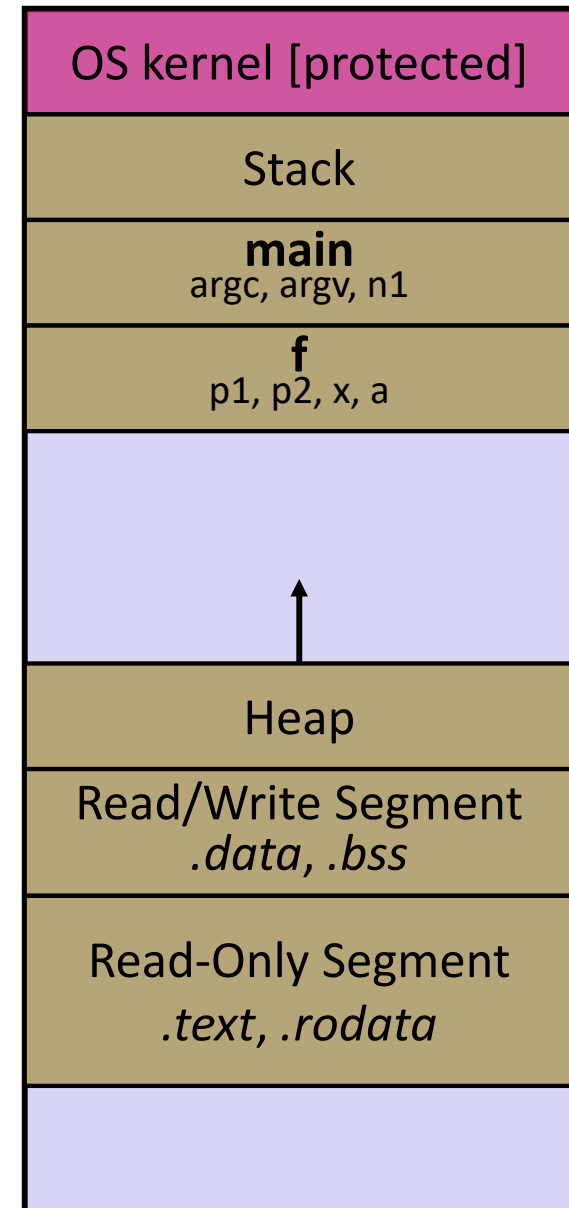
stack.c

```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
    int32_t n1 = f(3, -5);
    n1 = g(n1);
    return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
    int32_t x;
    int32_t a[3];
    ...
    x = g(a[2]);
    return x;
}

int32_t g(int32_t param) {
    return param * 2;
}
```



Stack in Action

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

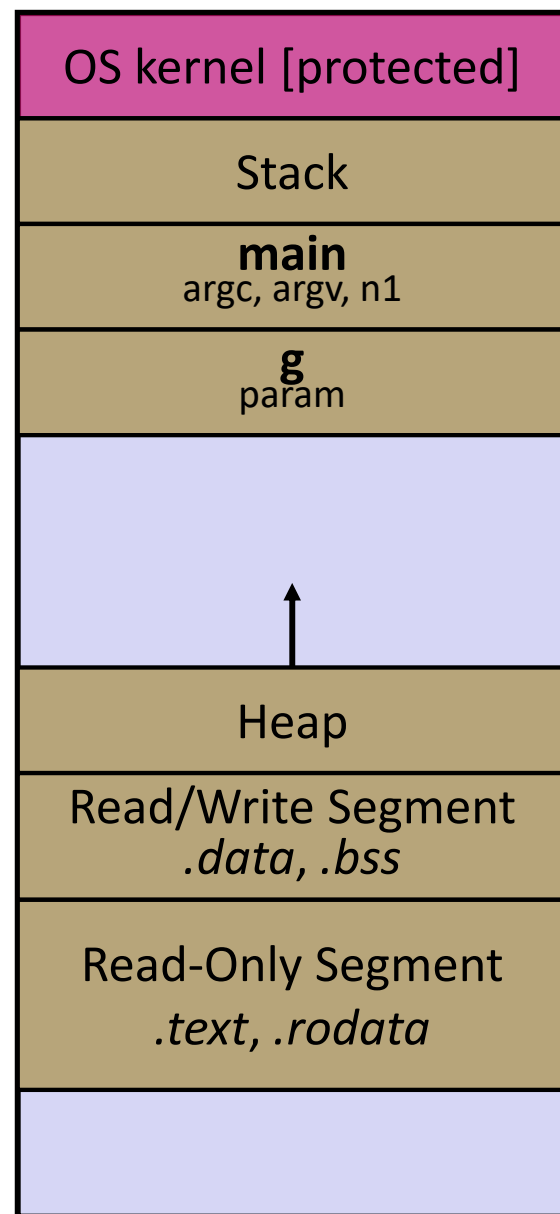
stack.c

```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
    int32_t n1 = f(3, -5);
    n1 = g(n1);
    return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
    int32_t x;
    int32_t a[3];
    ...
    x = g(a[2]);
    return x;
}

int32_t g(int32_t param) {
    return param * 2;
}
```



Stack in Action

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

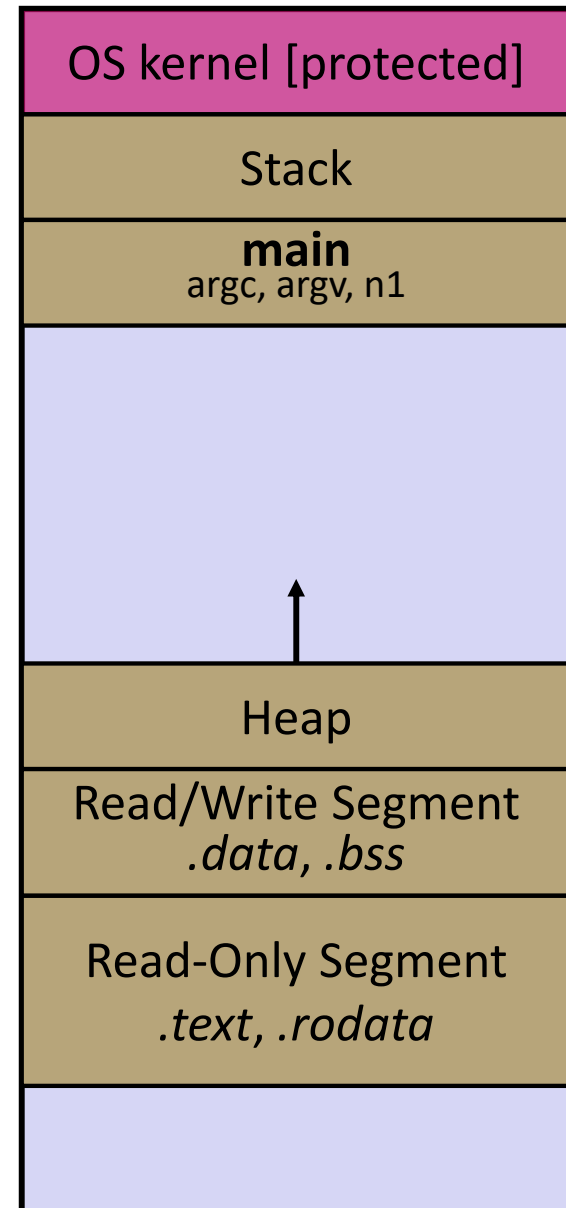
stack.c

```
int32_t f(int32_t, int32_t);
int32_t g(int32_t);

int main(int argc, char** argv) {
    int32_t n1 = f(3, -5);
    n1 = g(n1);
    return EXIT_SUCCESS;
}

int32_t f(int32_t p1, int32_t p2) {
    int32_t x;
    int32_t a[3];
    ...
    x = g(a[2]);
    return x;
}

int32_t g(int32_t param) {
    return param * 2;
}
```



Lecture Outline

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



Pointers

- ❖ Variables that store addresses
 - It points to somewhere in the process' virtual address space
 - `&foo` produces the virtual address of `foo`
- ❖ 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 <stdlib.h>
#include <stdint.h>
#include <inttypes.h>

int main(int argc, char** argv) {
    int32_t x = 351;
    int32_t* 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 %" PRIu32 "\n", x);

    *p = 333;    // change the value of x
    printf(" x is %" PRIu32 "\n", x);

    return EXIT_SUCCESS;
}
```

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 0x7ffffe847be34
x is 351
x is 333
```

Run 3:

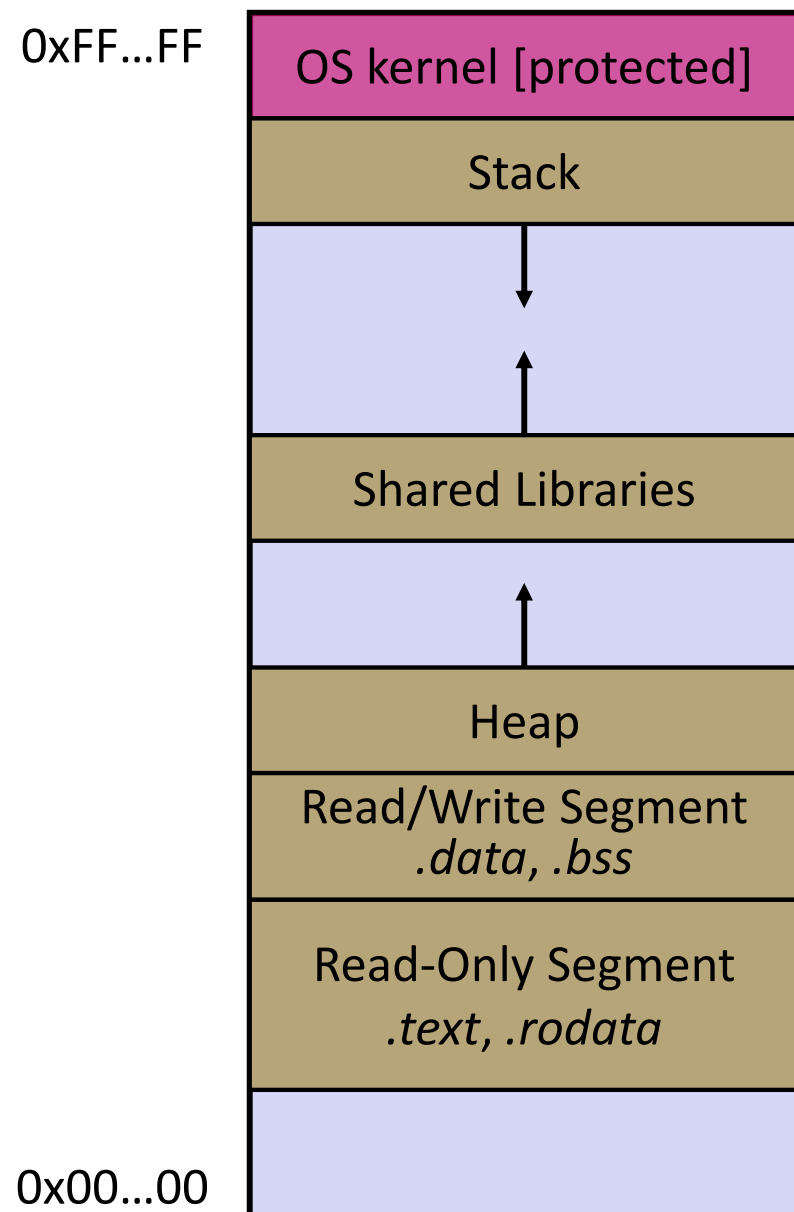
```
bash$ ./pointy
&x is 0x7ffffe7b14644
p is 0x7ffffe7b14644
x is 351
x is 333
```

Run 4:

```
bash$ ./pointy
&x is 0x7fffff0dfe54
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	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
arr [2]	4
arr [1]	3
arr [0]	2
p	0x7fff...74

0x7fff...48

0x7fff...44

0x7fff...40

0x7fff...38

Inadvisable Pointer Examples

inadvisable_pointers.c

```
// Leave them uninitialized!
int* int_ptr;
*int_ptr = 333;

// Use garbage values!
int_ptr = rand();
*int_ptr = 333;

// reinterpret raw bytes!
double d = 3.14;
int_ptr = (int*) &d;
*int_ptr = 333;
```

“Pointers are just variables that contain memory addresses”

```
// Uninitialized!
int*** ipp;
***ipp = 333;

// Garbage values!
ipp = rand();
***ipp = 333;

// lol, typechecking
void* vp = (void*) ipp;
void** vpp = &vp;
vpp = vp;
```

“Since pointers are variables, we can do all these things recursively!”

Lecture Outline

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

Basic Data Structures

- ❖ C does not support objects!!!
- ❖ **Arrays** are contiguous chunks of memory
 - Arrays have no methods and do not know their own length
 - Can easily run off ends of arrays in C – **security bugs!!!**
- ❖ **Strings** are null-terminated char arrays
 - Strings have no methods, but **string.h** has helpful utilities

```
char* x = "hello\n";
```

x →



- ❖ **Structs** are the most object-like feature, but are just collections of fields – no “methods” or functions

Arrays

❖ Definition: `type name [size]`

- Allocates `size * sizeof (type)` bytes of *contiguous* memory
- Normal usage is a compile-time constant for `size` (e.g. `int32_t 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*]

```
int32_t n = 175;  
int32_t 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) evaluates to the address of the start of the array
 - Cannot be assigned to / changed

```
int32_t primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0; // memory smash!
```

Challenge Question

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

```
int32_t m = 175;
int32_t scores[m];

void foo(int32_t n) {
    ...
}
```

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

```
int32_t m = 175;

void foo(int32_t n) {
    int32_t 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
int32_t matrix[3][5] = {
    {0, 1, 2, 3, 4},
    {0, 2, 4, 6, 8},
    {1, 3, 5, 7, 9}
};
```

Arrays as Parameters

- ❖ It's tricky to use arrays as parameters
 - What happens when you use an array name as an argument?
 - Recall: arrays do not know their own size

```
// prototype
int32_t sumAll(int32_t a[]);

int main(int argc, char** argv) {
    int32_t numbers[] = {9, 8, 1, 9, 5};
    int32_t sum = sumAll(numbers);
    return EXIT_SUCCESS;
}

int32_t sumAll(int32_t a[]) {
    int32_t i, sum = 0;
    for (i = 0; i < ...???)
}
```

Solution 1: Declare Array Size

```
// prototype
int32_t sumAll(int32_t a[5]);

int main(int argc, char** argv) {
    int32_t numbers[] = {9, 8, 1, 9, 5};
    int32_t sum = sumAll(numbers);
    printf("sum is: %" PRId32 "\n", sum);
    return EXIT_SUCCESS;
}

int32_t sumAll(int32_t a[5]) {
    int32_t 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

```
// prototype
int32_t sumAll(int32_t a[], int size);

int main(int argc, char** argv) {
    int32_t numbers[] = {9, 8, 1, 9, 5};
    int32_t sum = sumAll(numbers, 5);
    printf("sum is: %" PRId32 "\n", sum);
    return EXIT_SUCCESS;
}

int32_t sumAll(int32_t a[], int size) {
    int32_t i, sum = 0;
    for (i = 0; i < size; i++) {
        sum += a[i];
    }
    return sum;
}
```

arraysum.c

- This is the 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 more later)

Arrays: Call-By-Value or Call-By-Reference?

- ❖ **Technical answer:** a $T []$ array parameter is “decayed” 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(!))

```
void copyArray(int32_t src[], int32_t dst[], int32_t size) {
    int32_t i;
    int32_t copy[size]; // OK in C99, still stylistically bad
    for (i = 0; i < size; i++) {
        copy[i] = src[i];
    }
    dst = copy; // doesn't change caller's array
}
```

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?

```
int32_t* copyArray(int32_t src[], int32_t size) {  
    int32_t 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
 - “Feels” like call-by-reference, *but technically it's not*

```
void copyArray(int32_t src[], int32_t dst[], int32_t size) {  
    int32_t 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, ...);`

```
int32_t  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 reviewed 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