Memory and Arrays
CSE 333

Instructor: Hannah C. Tang

Teaching Assistants:

Deeksha Vatwani	Hannah Jiang	Jen Xu
Justin Tysdal	Leanna Nguyen	Sayuj Shahi
Wei Wu	Yiqing Wang	Youssef Ben Taleb

Administrivia (1)

- Exercise 0 was due this morning
  - Any significant problems getting it done?
    - If unusual situation, please contact the staff with an email message so we can help
  - Sample solution will be posted late today and linked to calendar
    - Requires CSE login; please do not distribute
      - Non-CSE students should have received guest accounts for the quarter. Let us know (email to cse333-staff) if you're not set up, but we'll probably need for you to contact support[at]cs to get it resolved
- Exercise 1 out today, due Friday morning @ 11 **am**

Administrivia (2)

- Reference system for grading is *current* CSE lab/attu/VM
  - For both exercises and homework (project) code
  - It's your job to be sure your solution(s) work there
    - Just because it works on ReallyCoolLinuxDistribution® doesn't mean it necessarily works on other Linux systems, including ours – there are lots of subtle differences between Linux systems that can cause problems

Administrivia (3)

- Homework 0 gitlabs out now, spec later today
  - Due Monday @ 10 **pm**
  - Logistics and infrastructure for projects – should be quick

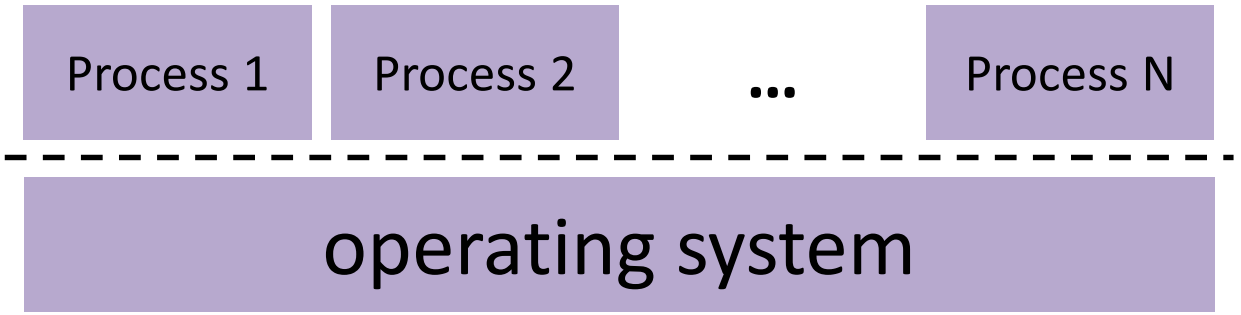
- Homework 1 will be posted and pushed to repos this weekend – read and get started as soon as it's out
  - Linked list and hash table implementations in C
  - Please read the spec and start looking at the code this weekend
    - For large projects, you must pace yourself so if something baffling happens, you can let it go for the day and come back to it tomorrow

Lecture Outline

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

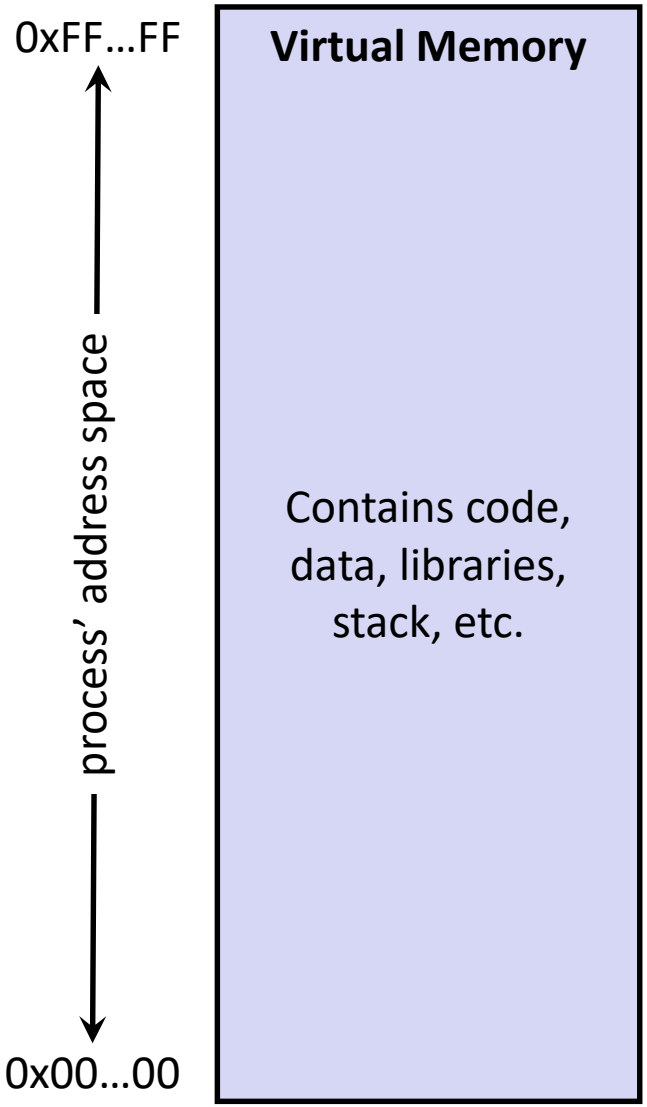
OS and Processes

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



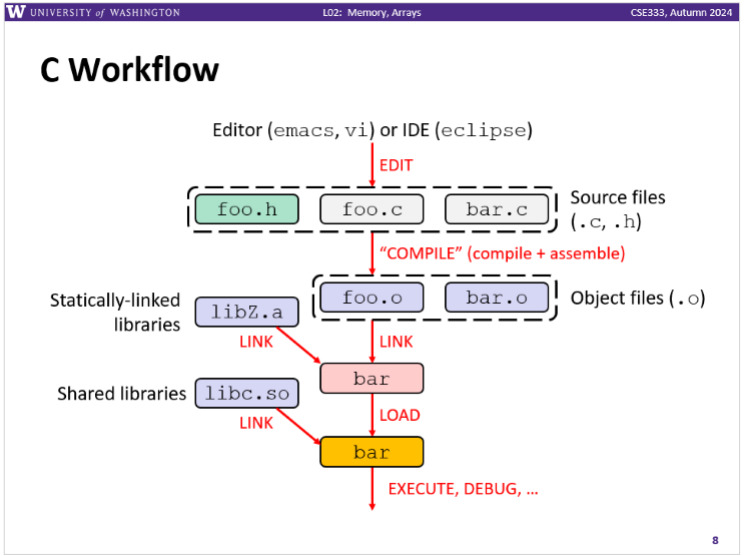
Processes and Virtual Memory

- 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



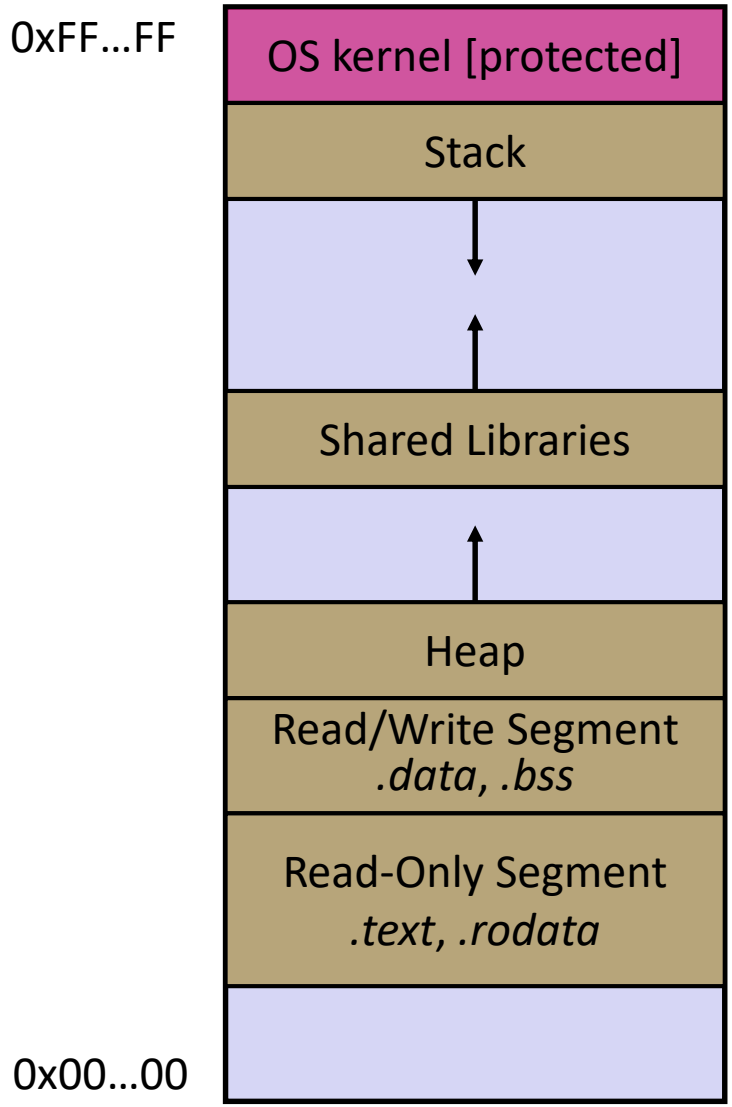
Loading

- 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



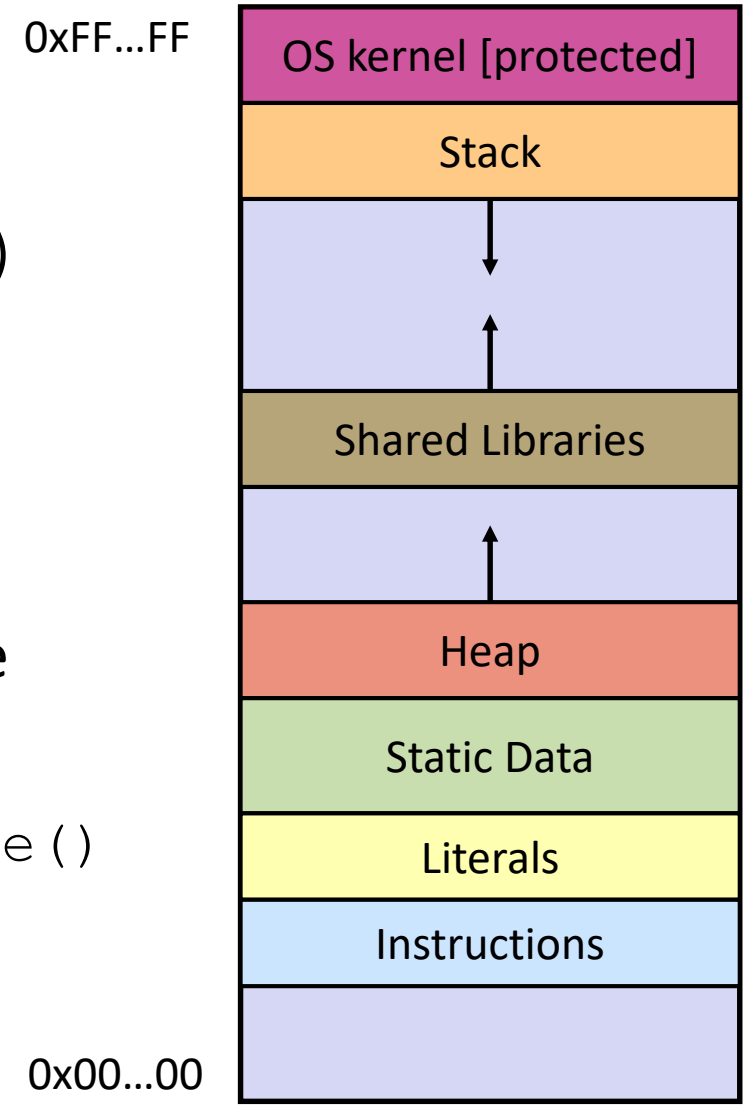
Loading

- 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



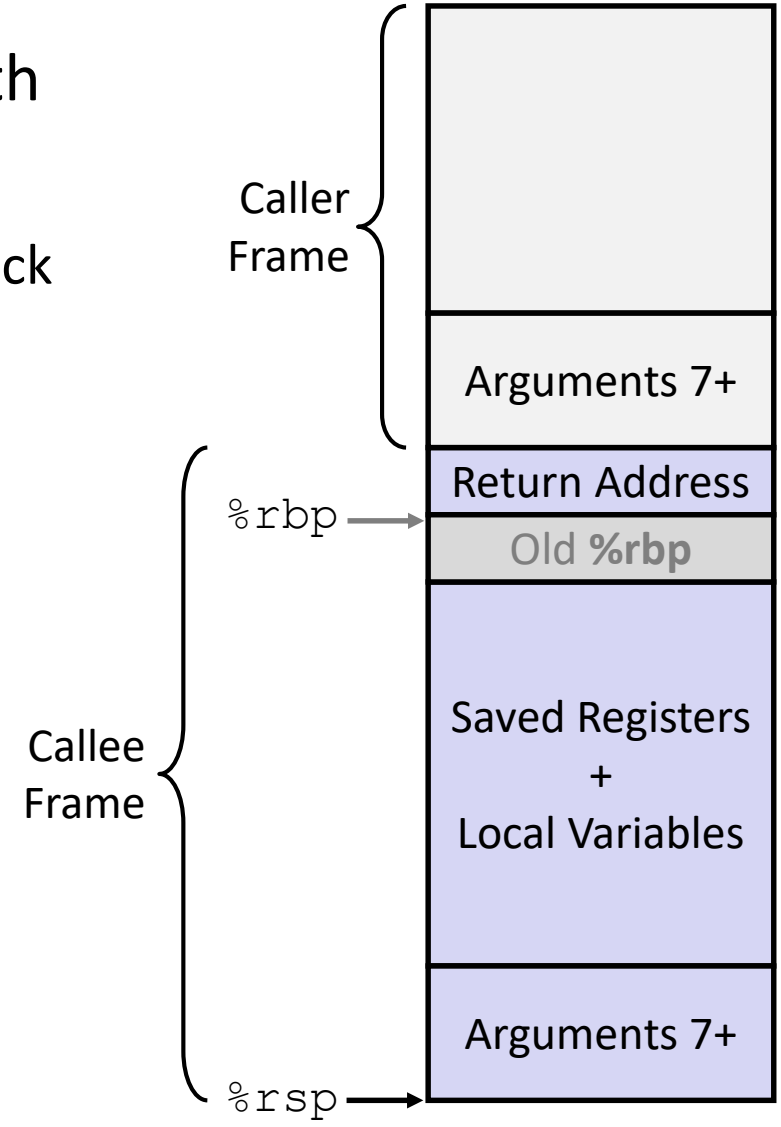
Memory Management

- *Local* variables on the Stack
  - **Automatically** allocated and freed via calling conventions (`push`, `pop`, `mov`)
- *Global* and *static* variables in Data
  - **Statically** alocated/freed when the process starts/exits
- ***Dynamically**-allocated* data on the Heap
  - `malloc()` to request; must call `free()` to release, 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

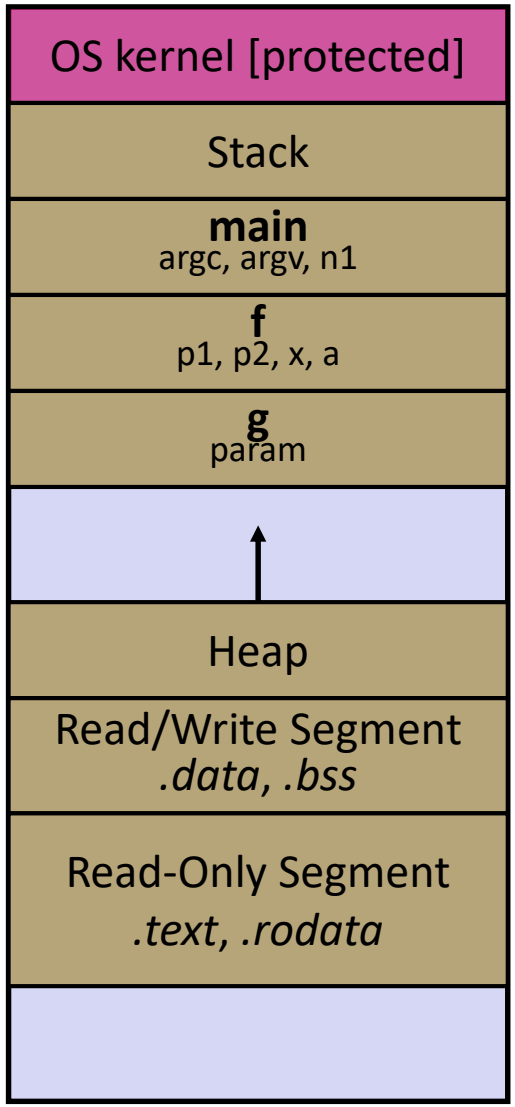
```
#include <stdint.h>

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;
}
```



Poll Everywhere

pollev.com/uwcse333

- Draw what the stack looks like at this line

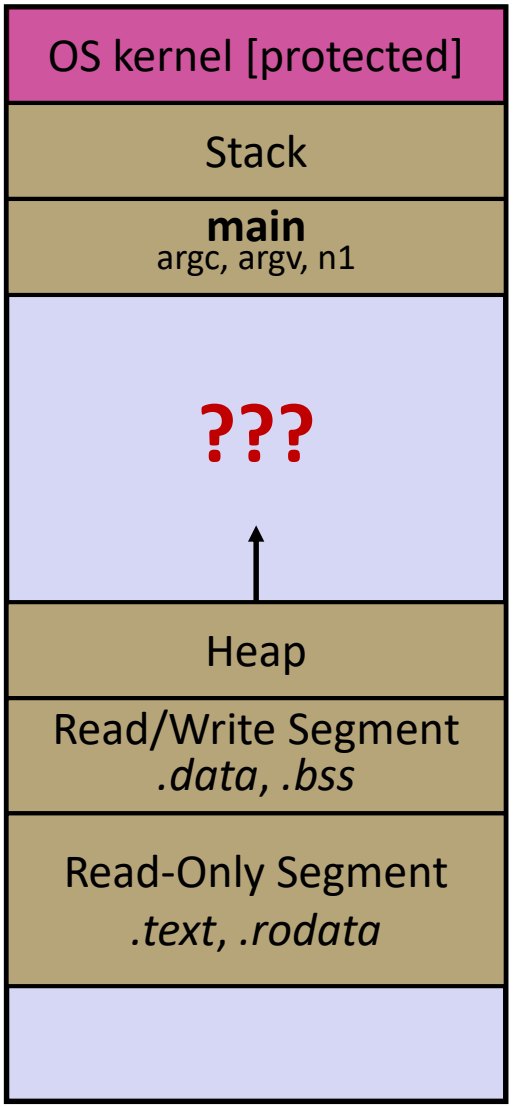
```
#include <stdint.h>

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

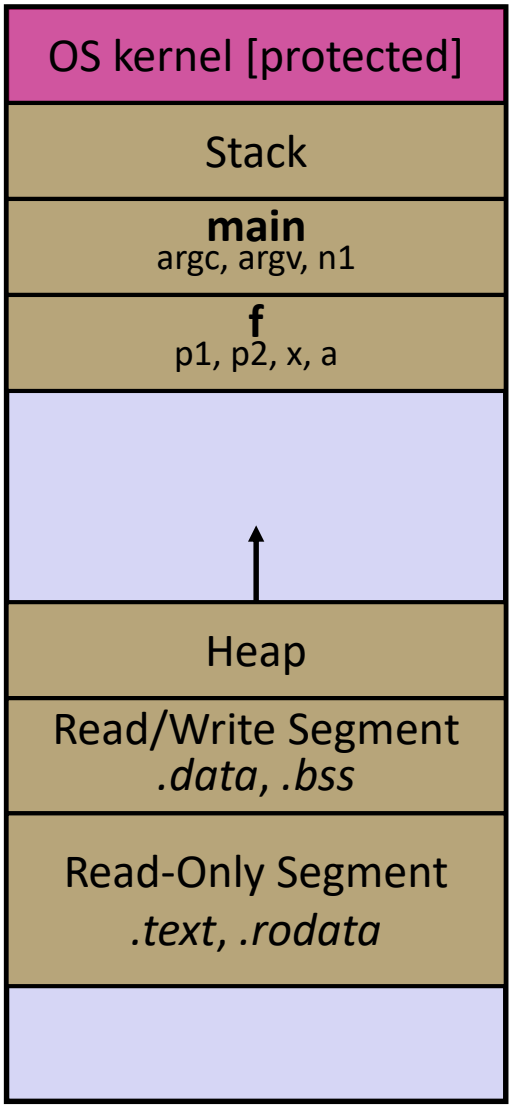
```
#include <stdint.h>

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

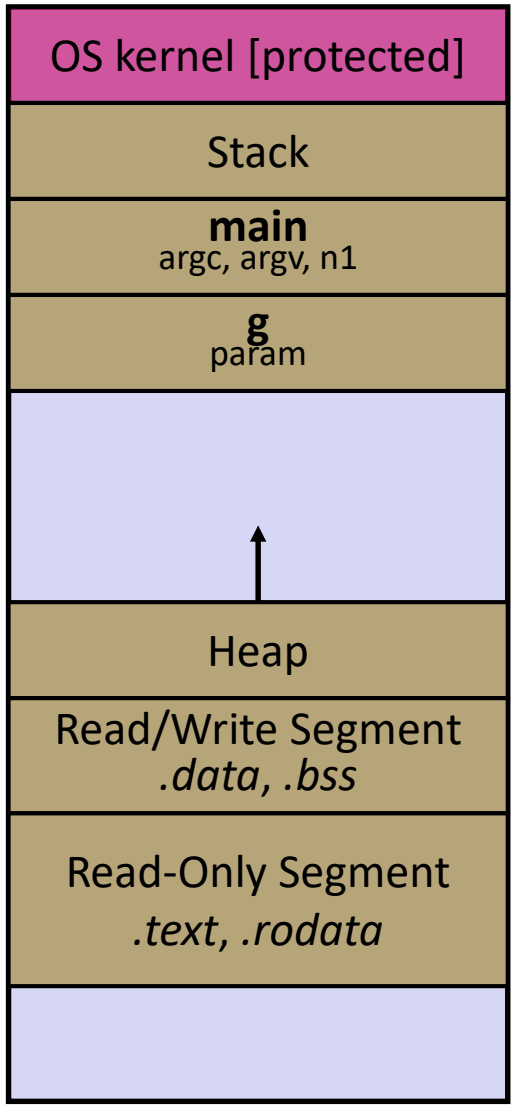
```
#include <stdint.h>

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

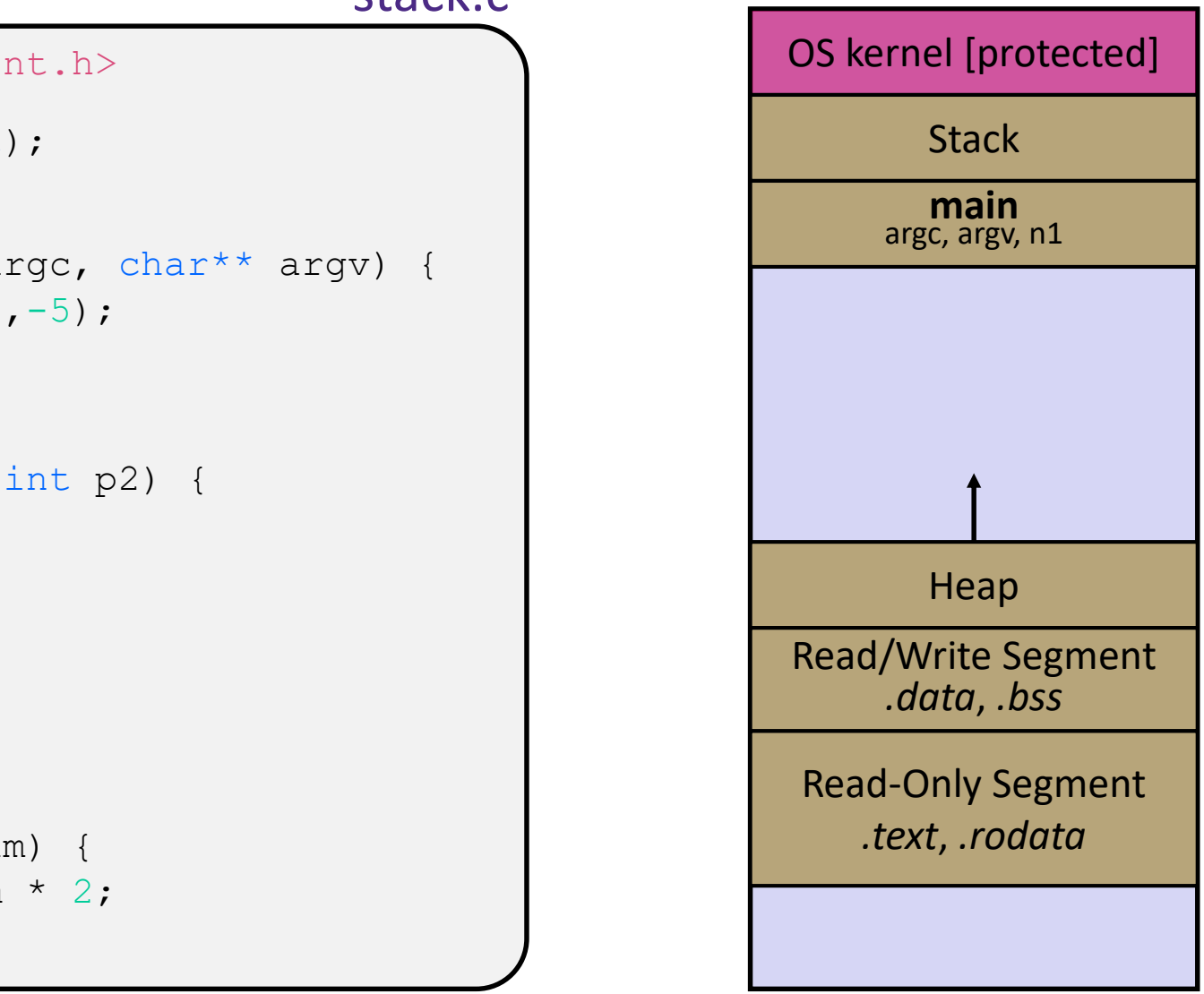
```
#include <stdint.h>

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

- 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 <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 0x7fffe847be34
 p is 0x7fffe847be34
 x is 351
 x is 333
```

Run 3:

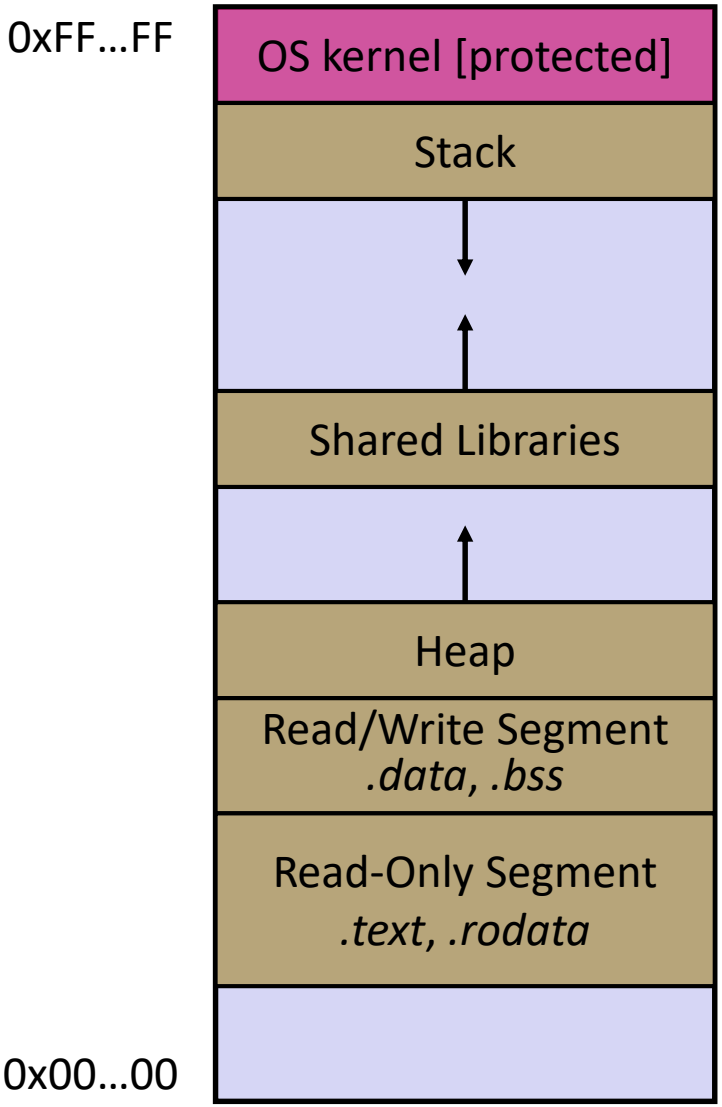
```
bash$ ./pointy
&x is 0x7fffe7b14644
 p is 0x7fffe7b14644
 x is 351
 x is 333
```

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



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" (i.e., uninitialized, unknown)

- 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++) allowed variable-length arrays
    - Uncommon; nowadays 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) evaluates to 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!
```

Multi-dimensional Arrays

- Generic 2D format:

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

  - Still allocates a single, contiguous chunk of memory
  - C stores arrays in *row-major* order

```
// 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

Poll Everywhere

pollev.com/uwcse333

- The code snippets both use a variable-length array. What will happen when we compile them *(with C99 or later)*?

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

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

```
int m = 175;

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

A.	Compiler Error	Compiler Error
B.	Compiler Error	No Error
C.	No Error	Compiler Error
D.	No Error	No Error
E.	Halp!	

Administrivia (1)

- Office hours start today! See schedule on web calendar

- Discussion board: prefer public postings to private
  - … unless it has specific code or other details that should not be shared.
  - Then the answers can help more people and we can reduce duplicate effort to answer the same question(s) multiple times.
  - Anonymous postings are fine if you're feeling bashful. ☺

- Exercise 2 due Monday @ 11 **am**
- Homework 0 due Monday @ 10 **pm**

Administrivia (2)

- You should be pretty far along in HW0 by now
  - Went over gitlab setup in sections yesterday
  - If you haven't cloned your repo yet, do it *now*! If anything is wrong *send mail to cse333-staff[at]cs* (*now*!) so we can fix accounts/repos before the weekend

- HW1 will be pushed to repos over the weekend
  - Linked list and hash table implementations in C
  - Download starter code using `git pull` in your course repo
    - Might have "merge conflict" if your local repo has unpushed changes
      - Default git merge handling will almost certainly do the right thing
      - To avoid, always do a git pull before any git commit or push
  - Please read the assignment and start looking at the code now!
    - For large projects, you want to pace yourself so if something baffling happens, you can let it go for the day and come back to it tomorrow

Administrivia (3)

- Exercise grading
  - Score is an overall evaluation: 3/2/1/0 = superior / good / marginal / not sufficient for credit
    - We expect lots of 2's and 3's at first, more 3's on later exercises
  - Then additional ±0 rubric items as needed
    - These are a quick way of communicating "why" – reasons for deductions or comments about your solution
    - Allows us to be more consistent in feedback
    - The ±0 "score" is just because that's how we have to use Gradescope to handle feedback notes – it does not contribute to "the points"

Lecture Outline

- **Arrays (cont.)**
- Pointers & Pointer Arithmetic
- Pointers as Parameters
- Pointers and Arrays
- Function Pointers

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];   // allowed in C99

  for (i = 0; i < size; i++) {
    dst[i] = src[i];
  }

  return dst; // no compiler error, but wrong!
}             // returns ptr to abandoned memory!
```

buggy_copyarray.c

Solution: Output Parameter

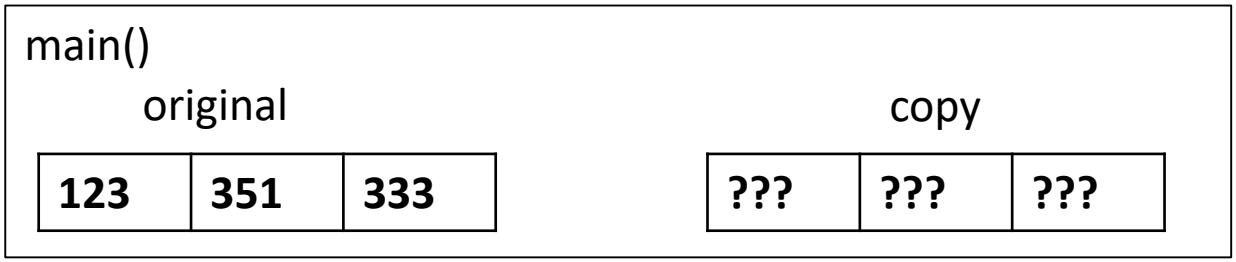
- Create the "returned" array in the caller
  - Pass it as an output parameter to `copyarray()`
    - A pointer parameter that allows the called function to store values that the caller can use
  - Works because arrays are "passed" as pointers

```
void copyArray(int src[], int dst[], int size) {
  int i;

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

copyarray.c

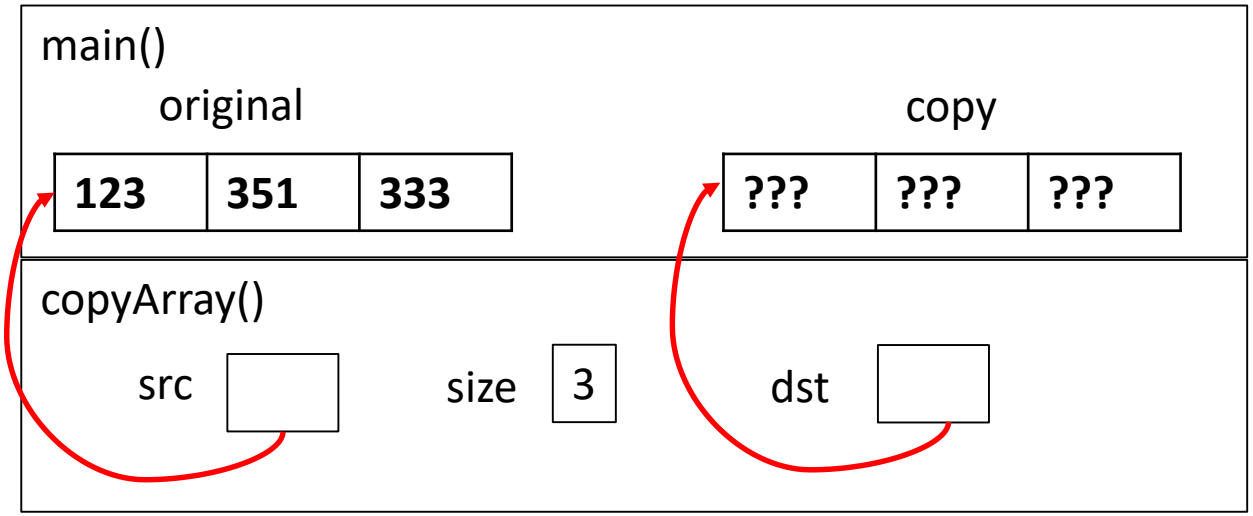
Array Memory Diagram




```
int main(){
  int original[] = {123, 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

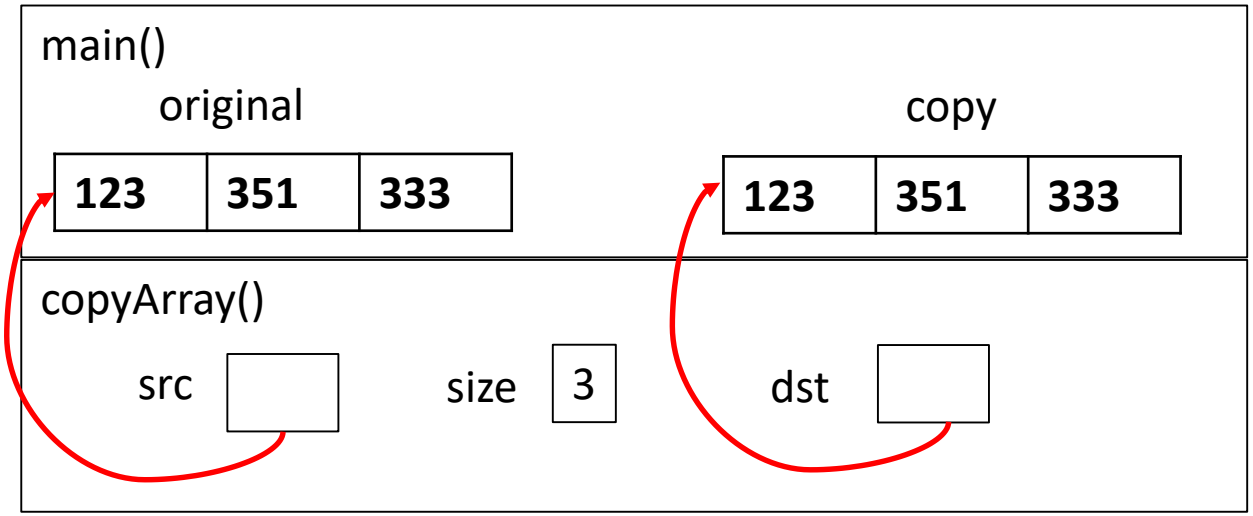



```
int main(){
  int original[] = {123, 351, 333};
  int copy[3];
  copyArray(original, copy, 3);
}

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[] = {123, 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`!

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;
char* str1 = "333 rocks";  // ptr to read-only const data
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

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(!))

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

Array Parameters – [ ] or * ?

- Array parameters are *actually* pointers to the beginning of the array
  - The [ ] syntax for parameter types is just for convenience
    - Use whichever best helps the reader

This code:

```
void f(int a[]);

int main( ... ) {
  int a[5];
  ...
  f(a);
  return EXIT_SUCCESS;
}

void f(int a[]) {...}
```

Equivalent to:

```
void f(int* a);

int main( ... ) {
  int a[5];
  ...
  f(&a[0]);
  return EXIT_SUCCESS;
}

void f(int* a) {...}
```

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)