



pollev.com/naomila



How is course setup going for you?

Vote for *each* of CSE Linux environment, text editor, and Gitlab/git.

- A. **Done! Went (relatively) smoothly.**
- B. **Done! Was tough to set up.**
- C. **Still working on it.**
- D. **Haven't tried to set it up yet.**

CSE333 Systems Programming

Memory, Data, Parameters

Instructors:

Naomi Alterman

Teaching Assistants:

Ann Baturytski

Barbora Buzkova

Mendel Carroll

Camden Harris

Hannah Hempstead

Rishabh Jain

Irene Lau

Jonathan Nister

Advay Patil

Aviel Wood

Angela Wu

Jennifer Xu

Administrivia

- ❖ Pre-quarter survey due Friday, 11:59 PM (link on course website)
- ❖ Exercise 1 due Friday morning, 11:00 AM
 - Submission via Gitlab (contact us if you don't have access)
 - Set up an SSH key and clone repos ASAP
 - Clone your repo before section tomorrow!
 - Make sure your solution compiles & runs properly on the CSE Linux environment before submitting
 - No late days for exercises
- ❖ Style and linting
 - We're going to push our class linting tool, `cppLint.py`, to your exercise repos today
 - Style grading will initially be lenient and get stricter as the quarter goes on

333 Workflow Aids/Upgrades

- ❖ See [Linux → Text Editors](#) on website for how to configure vim or VS Code for use in this class
 - From vi/vim, can compile and execute code without ever leaving the editor using ":!
<cmd>"
 - For VS Code, can connect to attu remotely and take advantage of the IDE features
 - From either text editor, you will want to get comfortable navigating and editing multiple files *simultaneously*
- ❖ We will learn the basics of Makefiles to simplify the compilation steps into the command make
- ❖ *Required* as of 26wi: [Husky OnNet VPN](#) for off-campus attu access

Lecture Outline

- ❖ **C Declarations and Definitions**
 - Toolchain considerations
- ❖ Memory Management (351 refresher)
- ❖ C Data Considerations
- ❖ C Parameters

Printing in C

❖ `int printf(const char* format, ...);`

- How do we remind ourselves how this works?
- DOCUMENTATION 🐱 !
 - Online: <https://www.cplusplus.com/reference/cstdio/printf/>
 - Terminal: `man 3 printf`
- Very important to use correct format specifier for the value you want to print, otherwise implicit casting will occur

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000



Error Handling

❖ Errors and Exceptions

- C does not have exception handling (no try/catch)
- Errors are returned as **integer error codes** from functions
 - Because of this, error handling is ugly and inelegant
 - For readability, `CONSTANT_NAMES` are defined to abstract away the actual integer values – need to look up in documentation
- Global variable **`errno`** holds value of last system error

❖ Status codes and signals

- Processes exit (e.g., **`return`** from **`main`**) with status code
 - Standard codes found in `stdlib.h`:
`EXIT_SUCCESS` (usually 0) and **`EXIT_FAILURE`** (non-zero)
- “Crashes” trigger signals from OS (e.g., `SIGSEGV` for segfault)

Function Definitions

❖ Generic format:

```
ReturnType FuncName(type param1, ..., type paramN) {  
    // statements  
}
```

```
// sum of integers from 1 to max  
int sumTo(int max) {  
    int i, sum = 0;  
  
    for (i = 1; i <= max; i++) {  
        sum += i;  
    }  
  
    return sum;  
}
```

Function Ordering

- ❖ You *shouldn't* call a function that hasn't been declared yet

Note: code examples from slides are posted on the course website for you to experiment with!

sum_badorder.c

```
int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return EXIT_SUCCESS;
}

// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

Solution 1: Reverse Ordering

- ❖ Simple solution; however, imposes ordering restriction on writing functions (who-calls-what?)

sum_betterorder.c

```
// sum of integers from 1 to max
int sumTo(int max) {
    int i, sum = 0;

    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}

int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return EXIT_SUCCESS;
}
```



Solution 2: Function Declaration

- ❖ Teaches the compiler the arguments and return types; function definitions can then be in a logical order
 - Function comment usually by the *prototype*

sum_declared.c

```
// sum of integers from 1 to max
int sumTo(int max); // func prototype

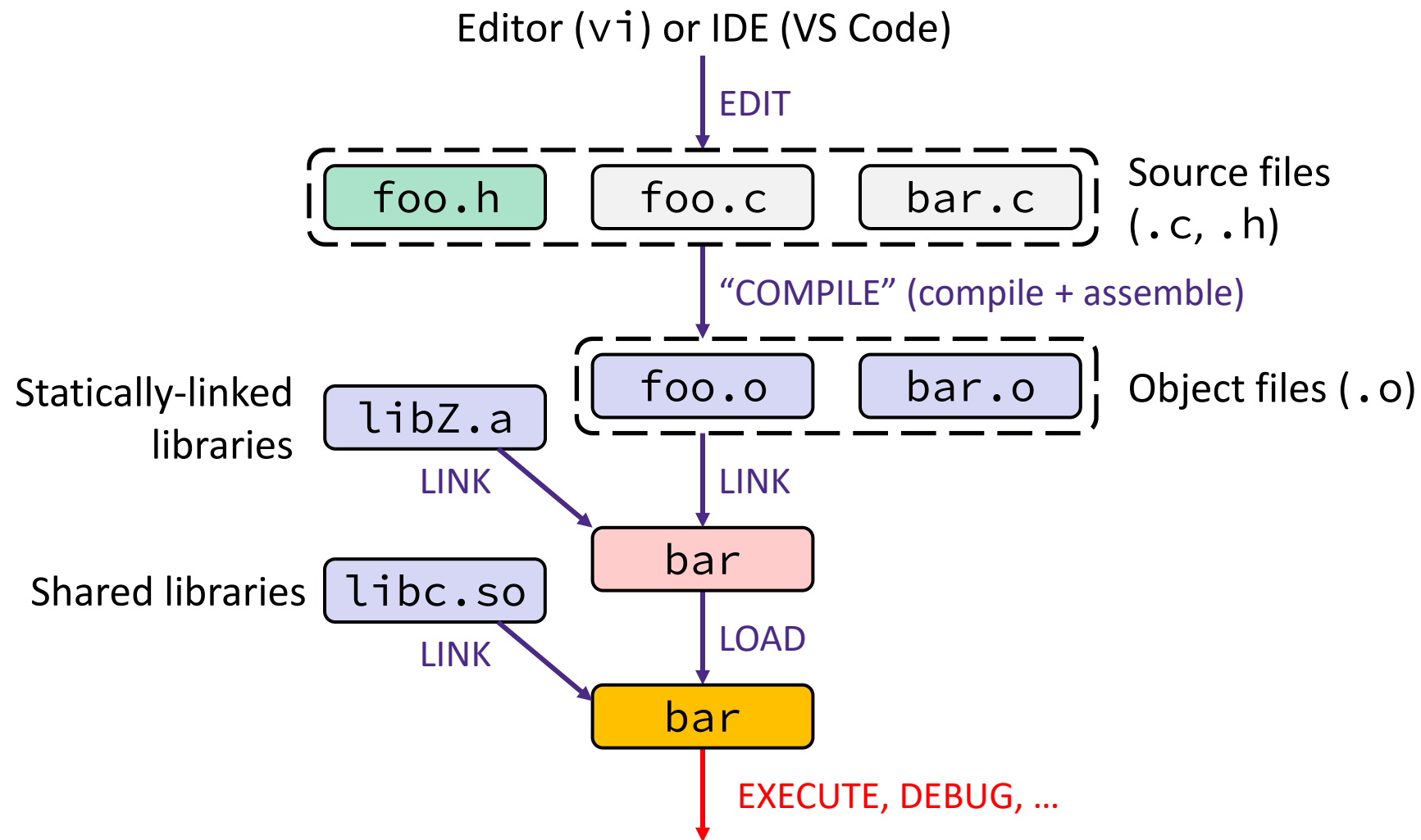
int main(int argc, char** argv) {
    printf("sumTo(5) is: %d\n", sumTo(5));
    return EXIT_SUCCESS;
}

int sumTo(int max) {
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

Function Declaration vs. Definition

- ❖ C/C++ make a careful distinction between these two
- ❖ **Definition:** The thing itself
 - *e.g.*, code for function, variable definition that creates storage
 - Must be **exactly one** definition of each thing (no duplicates)
- ❖ **Declaration:** Description of a thing
 - *e.g.*, function prototype, external variable declaration
 - Often in header files and incorporated via `#include`
 - Should also `#include` declaration in the file with the actual definition to check for consistency
 - Needs to appear in **all files** that use that thing
 - Should appear before first use

C Compilation Workflow



Multi-file C Programs

Note: This example has poor style for code split. More on multiple files in Lecture 5.

C source file 1
(sumstore.c)

```
void SumStore(int x, int y, int* dest) {  
    *dest = x + y;  
}
```

C source file 2
(sumnum.c)

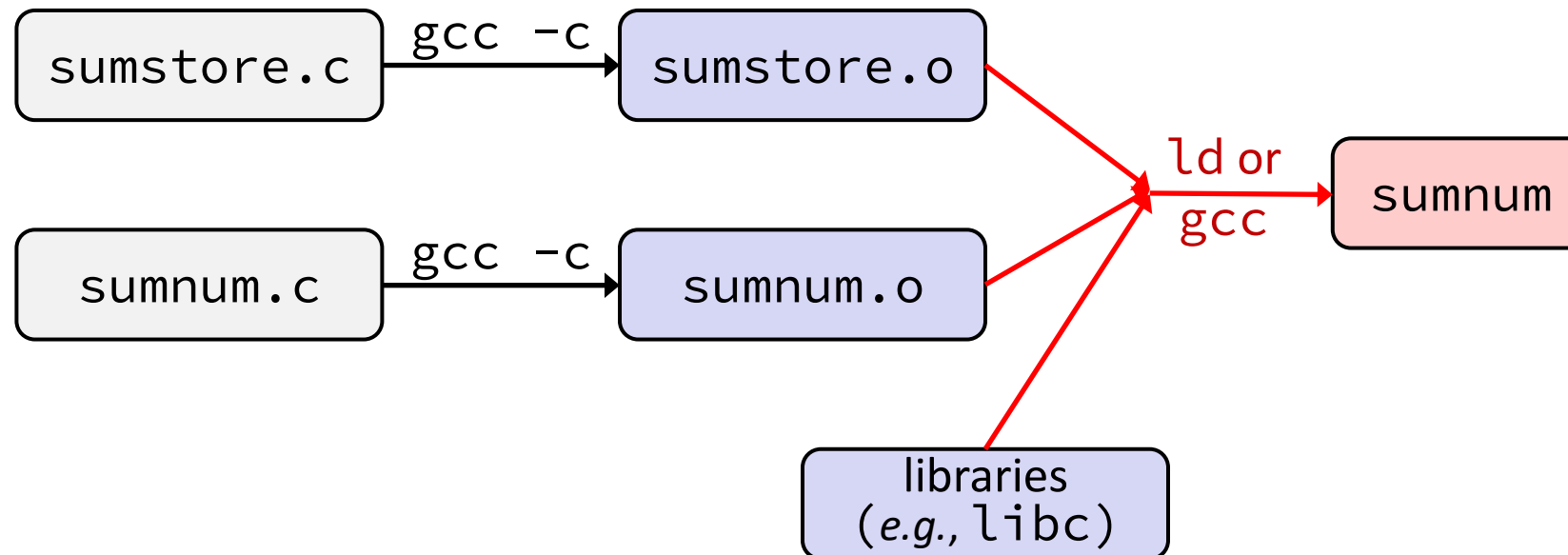
```
#include <stdio.h>  
#include <stdlib.h>  
  
void SumStore(int x, int y, int* dest);  
  
int main(int argc, char** argv) {  
    int z, x = 351, y = 333;  
    SumStore(x, y, &z);  
    printf("%d + %d = %d\n", x, y, z);  
    return EXIT_SUCCESS;  
}
```

Compile together:

```
$ gcc -o sumnum sumnum.c sumstore.c
```

Compiling Multi-file Programs

- ❖ The **linker** combines multiple object files plus statically-linked libraries to produce an executable
 - Includes many standard libraries (*e.g.*, `libc`, `crt1`)
 - A *library* is just a pre-assembled collection of `.o` files

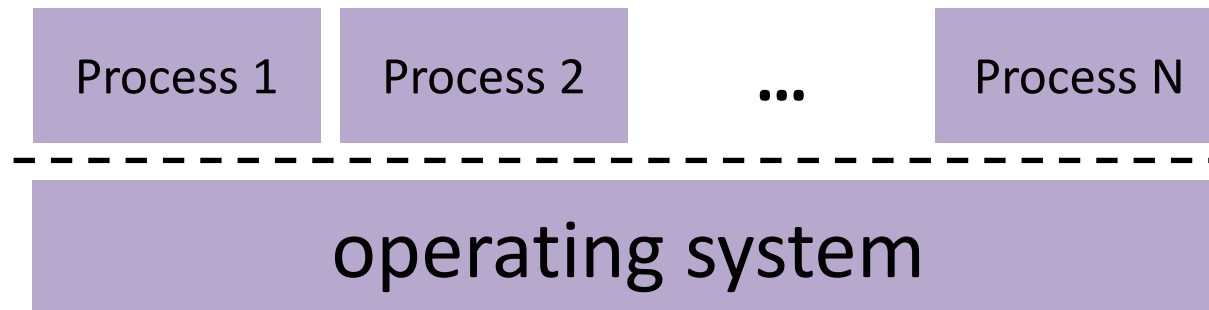


Lecture Outline

- ❖ C Declarations and Definitions
- ❖ **Memory Management** (351 refresher)
- ❖ C Data Considerations
- ❖ C Parameters

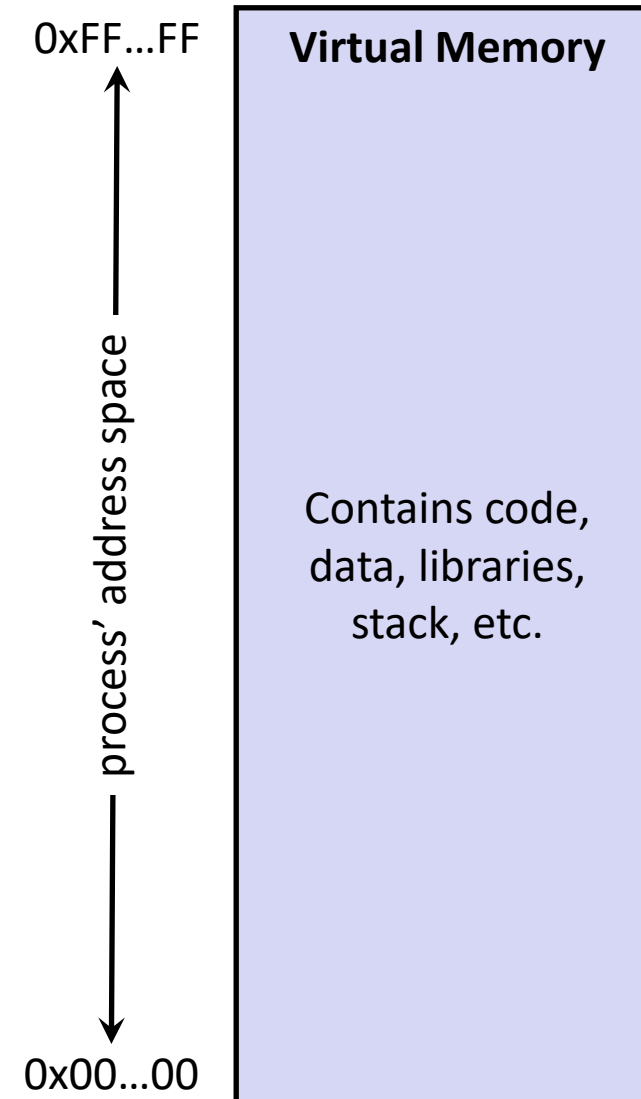
OS and Processes

- ❖ 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



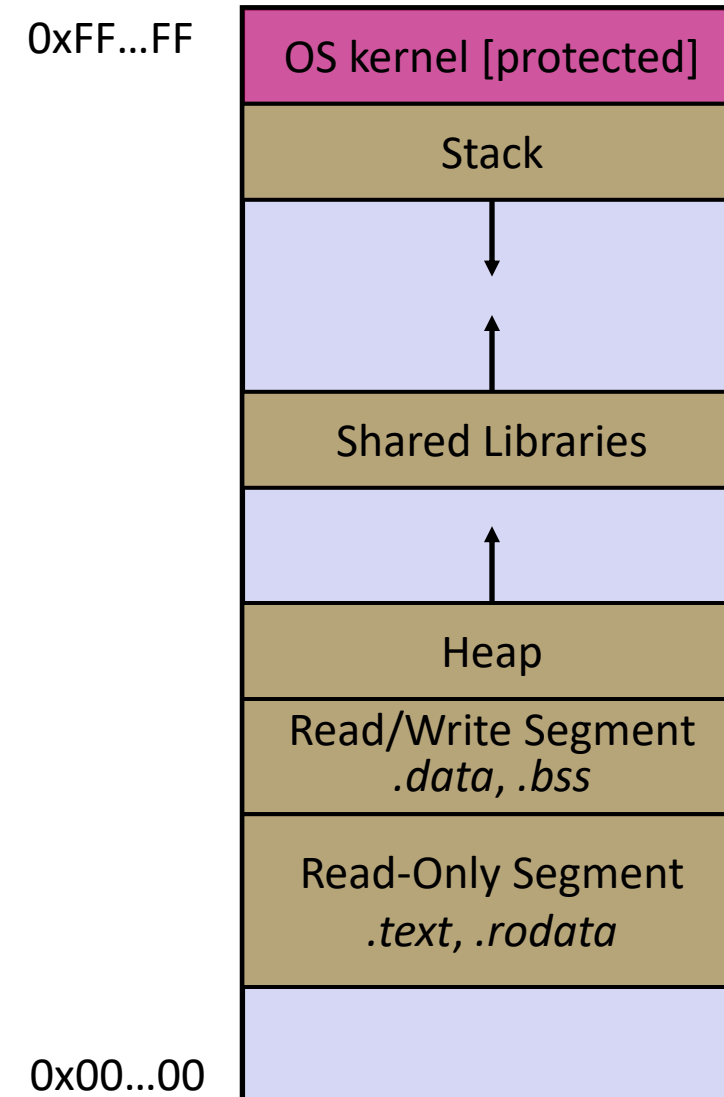
Processes and Virtual Memory

- ❖ 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



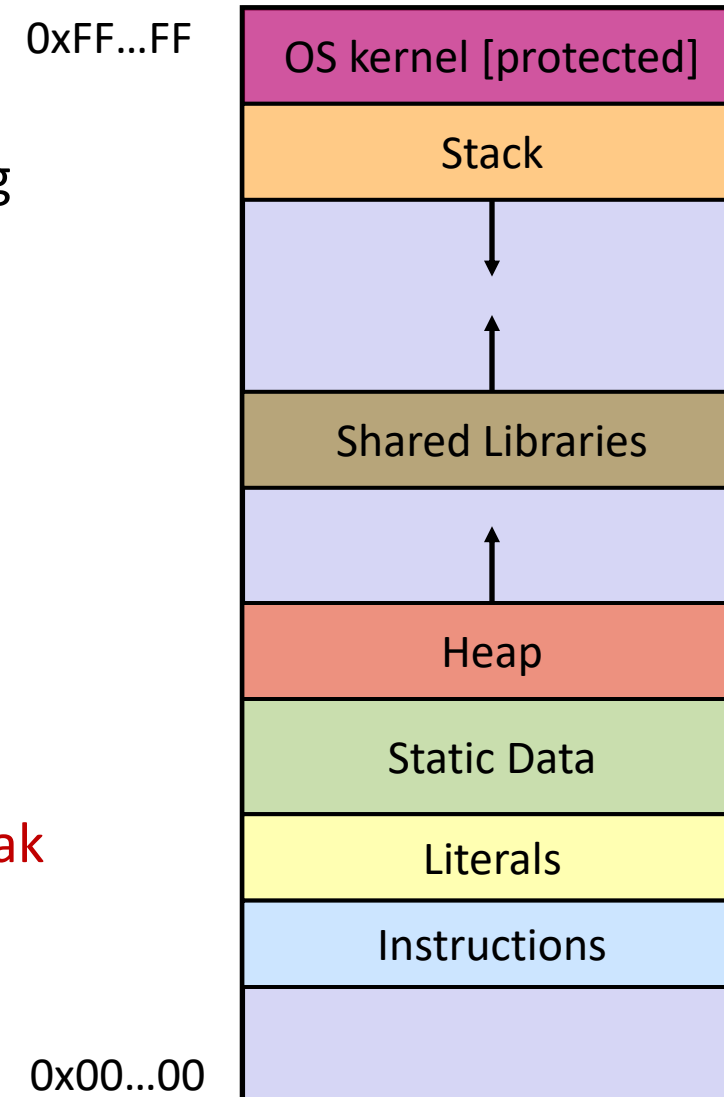
Loading

- ❖ 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



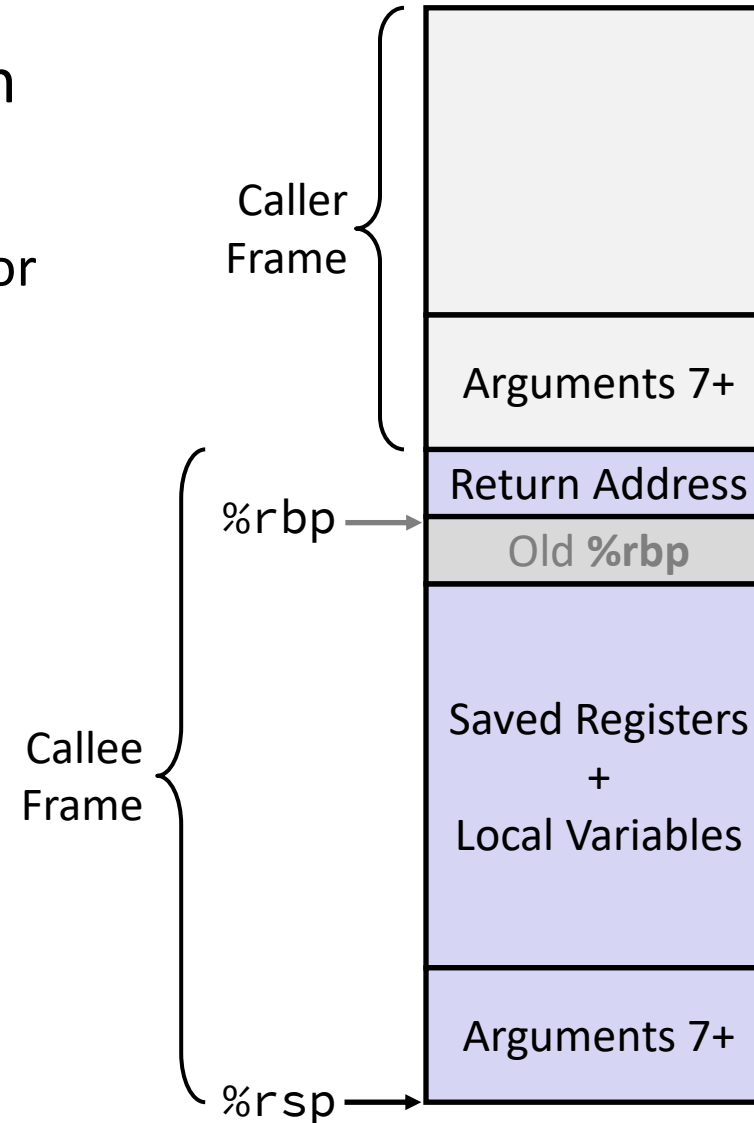
Memory Management

- ❖ *Local* variables on the Stack
 - **Automatically**-allocated and deallocated via calling conventions (push, pop, mov)
- ❖ *Global* and *static* variables in Data
 - **Statically**-allocated when the process starts and deallocated when it exits
- ❖ `malloc`-ed data on the Heap
 - **Dynamically**-allocated by process
 - Must call `free()` to free, otherwise a **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 (1/4)

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

stack.c

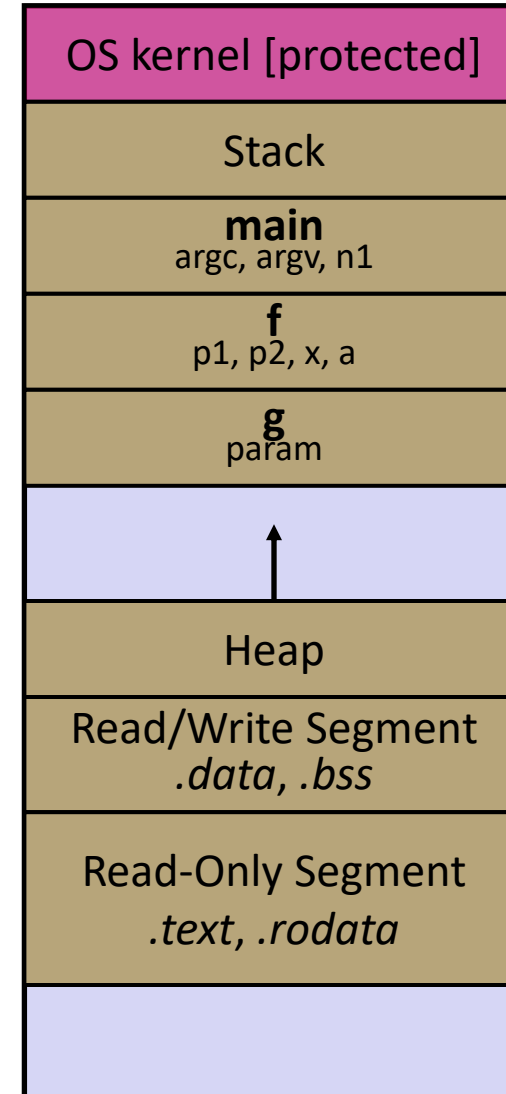
```
#include <stdlib.h>

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

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

→ 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 (2/4)

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

stack.c

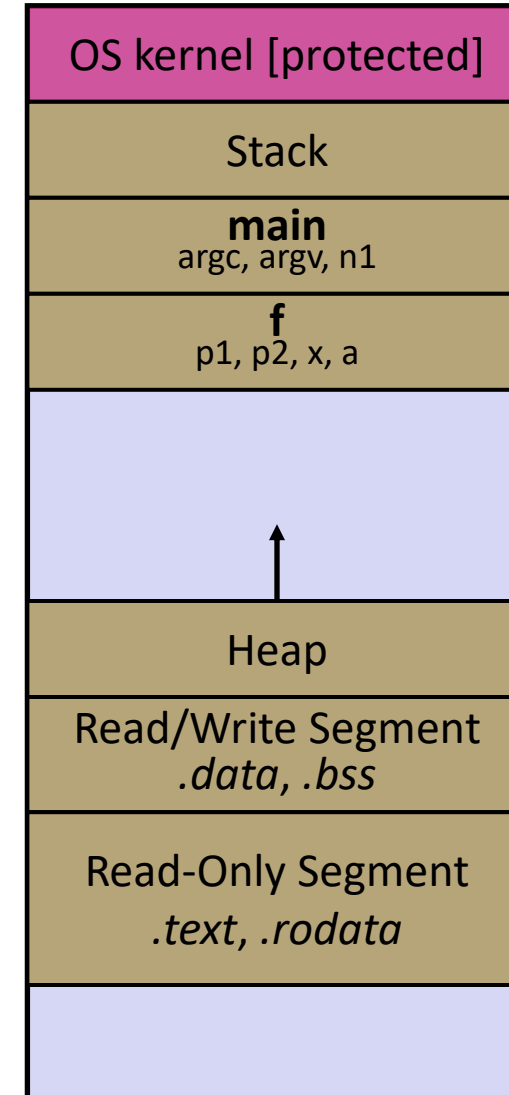
```
#include <stdlib.h>

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

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

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 (3/4)

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

stack.c

```
#include <stdlib.h>

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

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

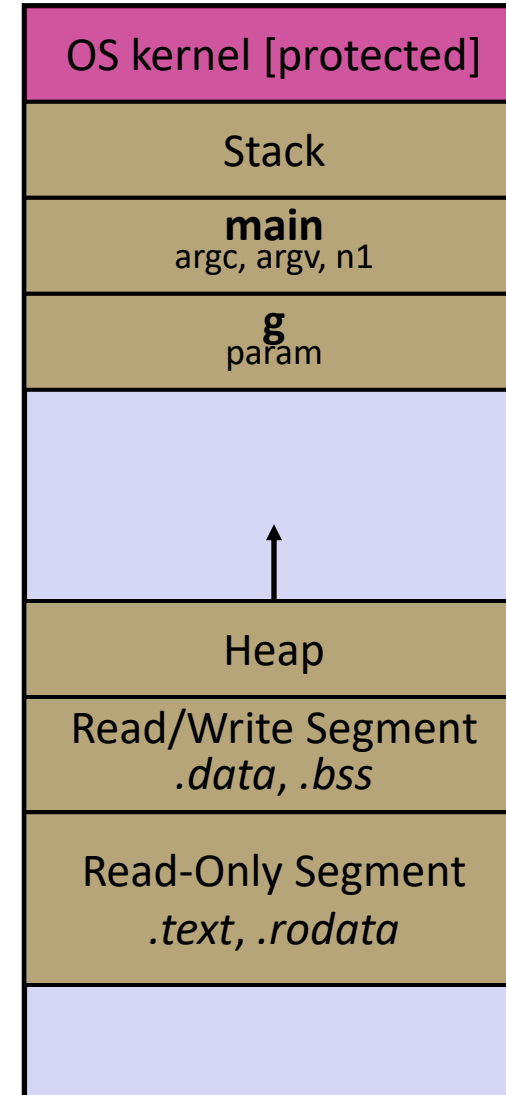
int f(int p1, int p2) {
    int x;
    int a[3];
    ...
    x = g(a[2]);
    return x;
}

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

→

→

→



Stack in Action (4/4)

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

stack.c

```
#include <stdlib.h>

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

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

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

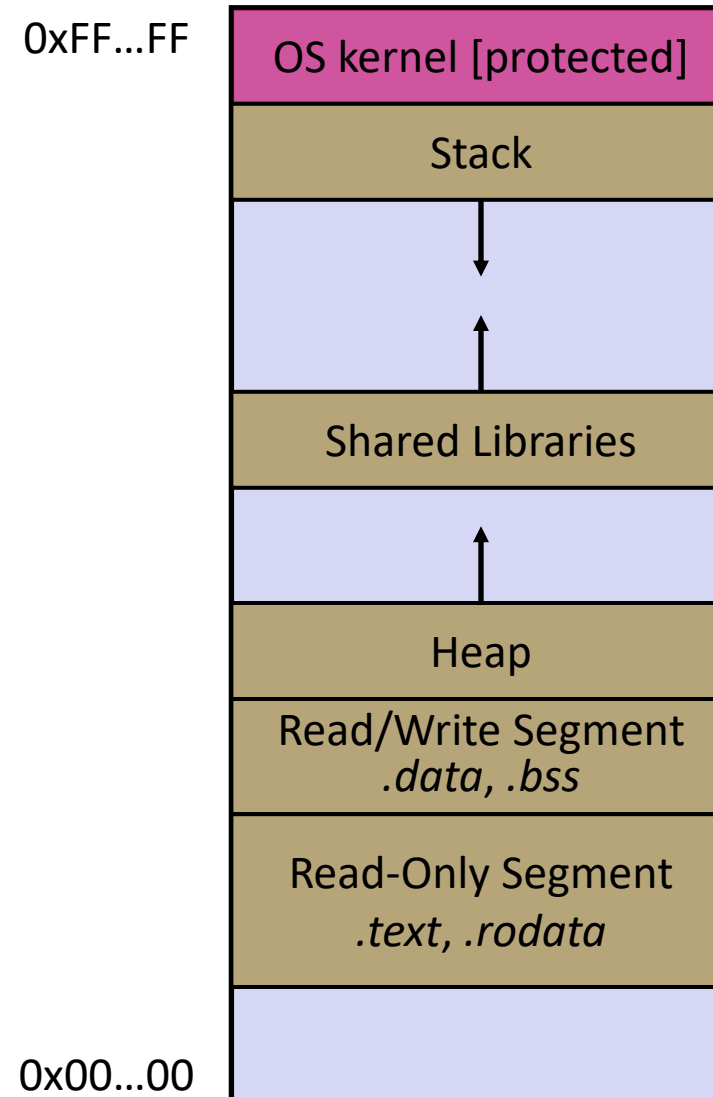
int g(int param) {
    return param * 2;
}
```

→



Neato trivia: 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 Declarations and Definitions
- ❖ Memory Management (351 refresher)
- ❖ **C Data Considerations**
- ❖ Parameters

C Primitive Types and Memory

Do not memorize, these aren't strict sizes!

❖ Integer types

- `char`, `int`

❖ Floating point

- `float`, `double`

❖ Modifiers

- `short` [int]
- `long` [int, double]
- `signed` [char, int]
- `unsigned` [char, int]

C Data Type	32-bit	64-bit	printf
char	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	12	16	%Lf
pointer	4	8	%p



C99 Extended Integer Types

- ❖ Solves the conundrum of “how big is an `long int`?”

```
#include <stdint.h>

void foo(void) {
    int8_t a; // exactly 8 bits, signed
    int16_t b; // exactly 16 bits, signed
    int32_t c; // exactly 32 bits, signed
    int64_t d; // exactly 64 bits, signed
    uint8_t w; // exactly 8 bits, unsigned
    ...
}
```

```
void sumstore(int x, int y, int* dest) {
```



```
void sumstore(int32_t x, int32_t y, int32_t* dest) {
```

Arrays

- ❖ Definition: `type name[size]` allocates `size*sizeof(type)` bytes of *contiguous* memory
 - By default, array values are “mystery” data (*i.e.*, uninitialized)
 - Normal usage is a compile-time constant for `size`
e.g., `int scores[175];`
- ❖ Size of an array
 - Not stored anywhere – array does not know its own size!
 - `sizeof(array)` only works in the 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”
 - Array name produces the address of the start of the array
 - Cannot be assigned to / changed
 - `name[index]` specifies an element of the array and can be used as an assignment target or as a value in an expression
 - Is actually `*(name+index)` with pointer arithmetic (Lecture 3)

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

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 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
 - (will need to either wait for Lecture 4 or review malloc/free on your own)