/unc/comp211

Systems Fundamentals

Static Memory & Static Lifetimes!

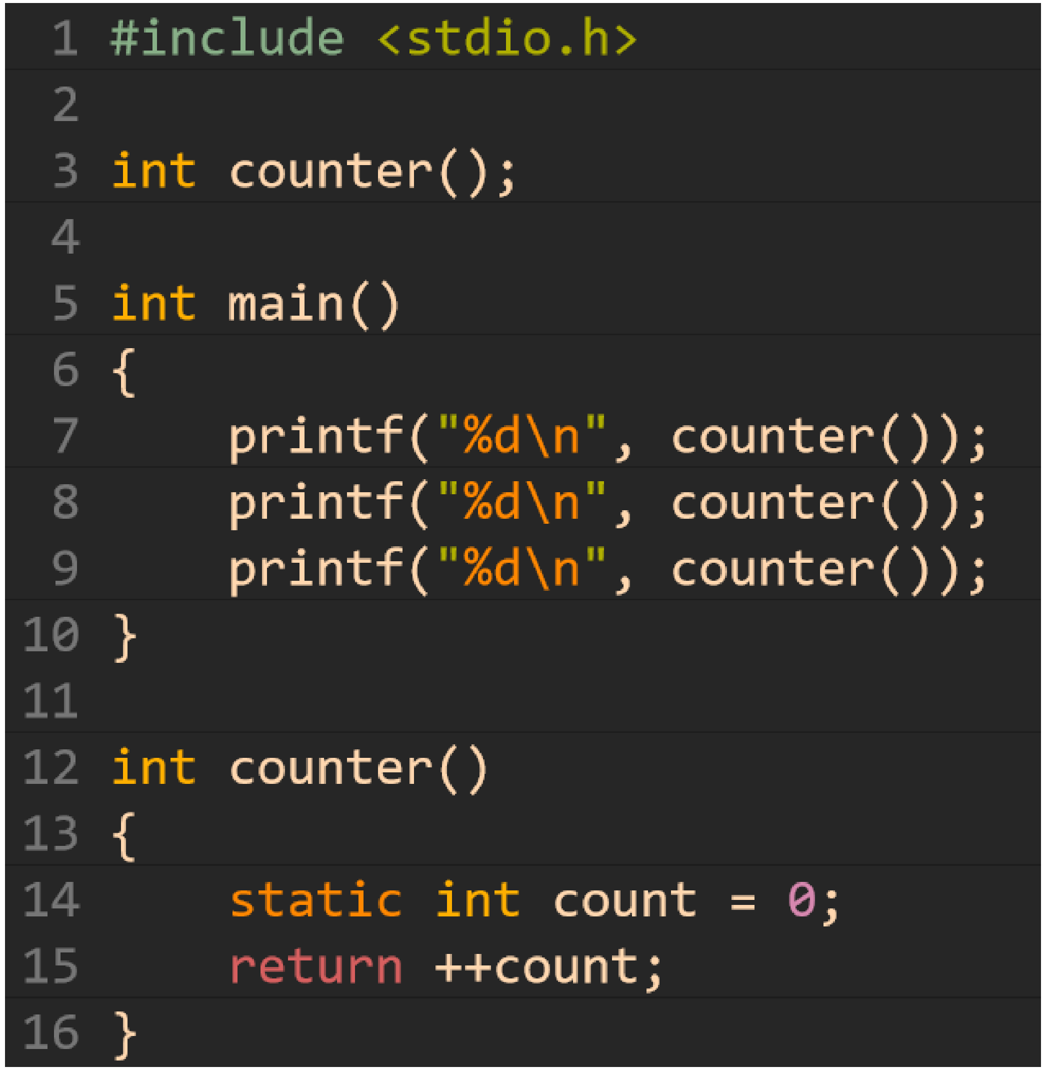
Compile Time vs. Runtime

- The process of executing your program's code has many stages
  - **Compile Time** - When a compiler takes source code as input and converts it to a more primitive representation closer to machine instructions.
    - This occurs when you run gcc or javac
  - **Runtime** - When your compiled program is executed by the processor.
    - This is your program "in motion"
    - This occurs when you run a.out or whatever you've named your executable
  - There are additional stages we'll introduce later, as well.

Automatic Variables and the Call Stack

- Whether a function is called can be **controlled by *side-effects* (files, events)**
  - Input side-effects are *unknown* until **runtime**!
- A **call to a function can occur from *anywhere***, thus **from any stack depth**
  - Automatic variable locations depend on the depth of call stack
  - Example: Consider printf being called from both main and a subroutine function, printf's automatic variable's addresses will be different for the two calls.
- There can be ***multiple instances*** *of* ***a single automatic variable***
  - This must be true for recursion to work!
- The address of automatic variables is only generally decidable at runtime.

What is the output?



```c
#include <stdio.h>

int counter();

int main()
{
    printf("%d\n", counter());
    printf("%d\n", counter());
    printf("%d\n", counter());
}

int counter()
{
    static int count = 0;
    return ++count;
}
```

Static Analysis

- **Static analysis** of your code is typically performed at compile time.
  - Type checking - do you have type errors?
  - Warning checks - examples:
    - variable use before initialization
    - function calls before declaration
    - unused variables
    - code paths in functions that do not return a value

- **Static** implies "**at rest**" and usually **decidable at compile time**.
  - **Static memory** is reserved for **singleton variables** and values whose addresses are "at rest" at runtime and, thus, are decided* at compile time.
  - * Due to address space layout randomization, the *static memory offset* is decided at compile time, but the exact location of the segment is randomized within a range at runtime for security purposes.

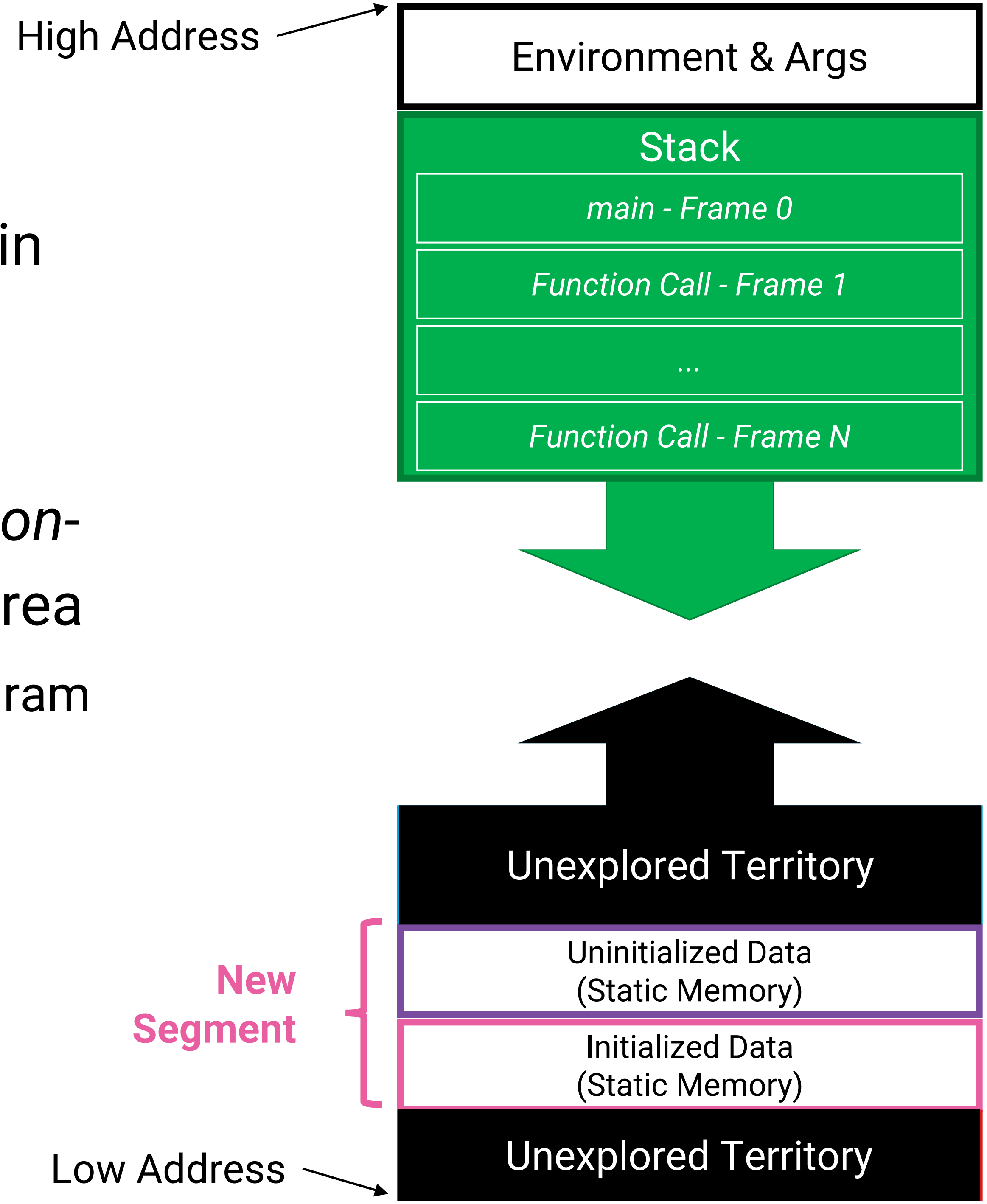
Declaration Demonstration and Notes

- Let's explore static addresses a little more closely...

```c
// global var
char global = 1;

// static global val - "private" to file
static char static_global = 2;

int main()
{
    // static local var - singleton in static memory
    static char static_local = 3;

    // automatic local var - stack allocated
    char automatic = 4;

    printf("&global: %p\n", &global);
    printf("&static_global: %p\n", &static_global);
    printf("&static_local: %p\n", &static_local);
    printf("&automatic: %p\n", &automatic);
}
```

Static Memory

- Static and global variables are allocated in the static memory segment of a process
- Those whose initialized values that are *non-zero* are allocated in the *initialized data* area
    - Values are copied in from the compiled program
- Those whose values are uninitialized, or initialized to zero, are allocated in the uninitialized data area and zeroed out.


High Address

Environment & Args

Stack

main - Frame 0

Function Call - Frame 1

...

Function Call - Frame N

Unexplored Territory

New Segment

Uninitialized Data (Static Memory)

Initialized Data (Static Memory)

Unexplored Territory

Low Address

Lifetime IS NOT Scope

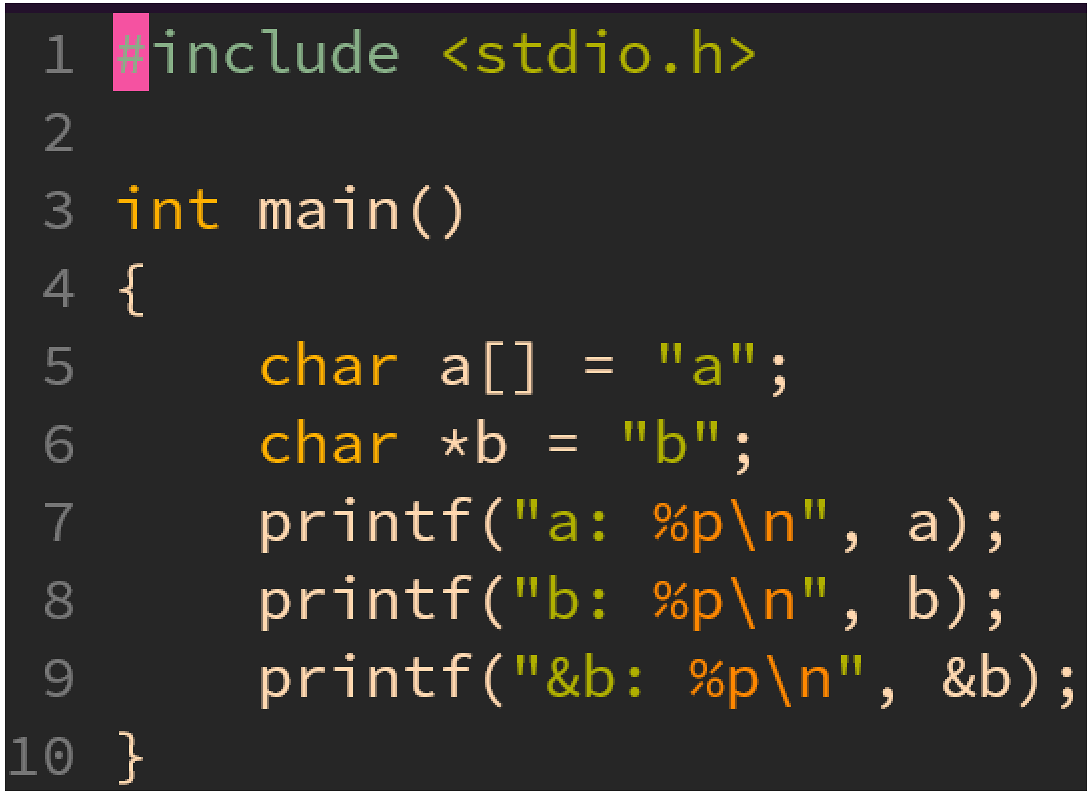
- **Lifetime**: When is a memory address safe for reading? When expired?

- For **automatic**/local/**stack variables** lifetime is closely related to scope
  - Safe after initialization, expired once out of ***scope***.

- For **static variables**, ***lifetime is unrelated to scope***.
  - All static memory is initialized (either with values or 0s) *before* your main function is called.
  - **The lifetime of static memory is *the entire duration of a process!***
  - But its scope can be global, static global, or static local and only valid after declaration!

- ***Extremely Important*** concept to understand in systems programming.

Sketch how you think the program looks in memory at line 7...



```
 1 #include <stdio.h>
 2
 3 int main()
 4 {
 5     char a[] = "a";
 6     char *b = "b";
 7     printf("a: %p\n", a);
 8     printf("b: %p\n", b);
 9     printf("&b: %p\n", &b);
10 }
```

- a is a char array allocated in text's frame on the stack
- b is a pointer to a string constant allocated in read-only static memory (more on this next)
- b's address is also in text's frame

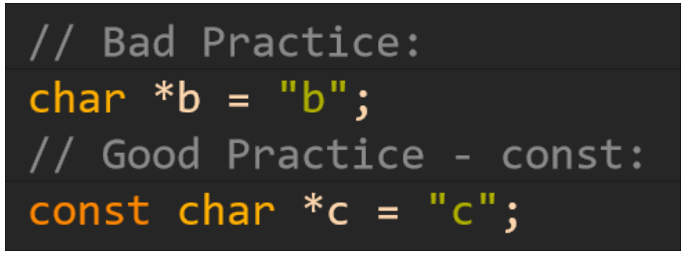
const char * pointers to *String Literals*

- When you initialize a char array with a string literal the contents are copied into each frame at the point of initialization.

- When you initialize a char * with a string literal, the char[] contents are stored in **read-only static memory**, *not* in the frame.

- **Why the difference?** An optimization. It's common you do not need to modify string literal contents at runtime, so it's a waste to copy them fresh in each frame. Storing the char[] in static memory and just initializing a pointer to it is more efficient.

- ***PSA: IF* YOU INITIALIZE A char * WITH A STRING LITERAL *ALWAYS* DECLARE IT const char ***
  - That you don't have to, without warnings, is a historical artifact.
  - If you attempt to write to read-only static memory you will get a segfault.
  - Declaring as const will give you appropriate errors at compile time if you attempt to write to its contents.



```
// Bad Practice:
char *b = "b";
// Good Practice - const:
const char *c = "c";
```

Lifetimes

Memory Segment	Safe After	Expires When	Runtime Dangers (Things that will (hopefully!) crash your program.)
Call Stack • automatic variables • parameters	Initialization	Function Call Returns	1. Returning or sharing address of automatic variable outside its scope.
Static Memory • static variables • global variables	Always* (Uninitialized variables are zeroed.)	Never	1. Writing to read-only static memory. 2. Unintended conflicting writes from many places (grave concern in multithreaded programs).