/unc/comp211

Systems Fundamentals

Function Pointers !

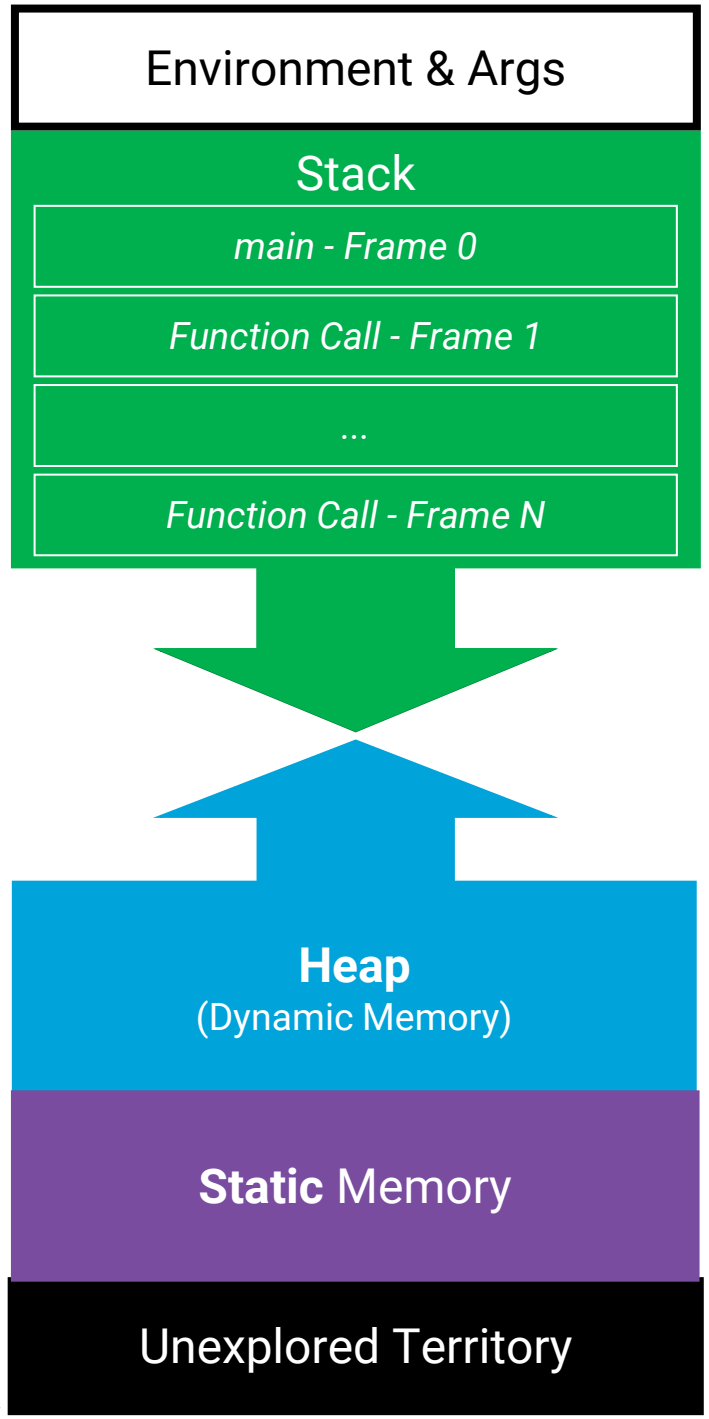
So... what *is* in memory all the way down there?

- Let's find out...

- Add the line:

```
printf("func main: %p\n", main);
```

- Your program's compiled machine code is stored in read-only memory beneath static memory!



A Function's "*Value*" in a System's Language

- At runtime, **a function name** evaluates to the **memory address** of the start of the **function's machine code instructions**.
  - The process's *machine code* is in memory just beneath its static memory.
  - The CPU carries out the instructions stored in this segment of memory.
  - *Your C code*, when compiled and executed as a process, lives in this segment!

- When a **function call** is encountered, the compiler emits machine instructions for setting up the new call frame and then the **CPU will JUMP to the address of** the function's instructions.

- If a function name is just a memory address, **can we store that address in a pointer and *then make function calls using the pointer's name?* *Yes!!!***

A Function's *Type*

- What is a function's type?
- A **function's type** is defined by its **parameter types** and its **return type**.

- Forward declarations of functions exhibit this:

```
void hello();
void world();
int add(int, int);
int sub(int, int);
```

- Which of the functions above is the same type as another? Different?

- The two void functions with no parameters are the same type (hello, world)
- The two int functions with two int parameters are the same type (add, sub)
- Anywhere you use a function of one type you could substitute the name of another of the same type.

Function Pointer Variable Types

- To declare a function pointer variable:
  1. **Declare it like you would a *forward* declaration without parameter names**
  2. **Add asterisk before the pointer variable's name**
  3. **Surround the asterisk and the pointer variable's name in parenthesis**

- Examples based on the types of functions of the previous slide:

```
void (*a_void_fn)();
int64_t (*an_int64_binop)(int64_t, int64_t);
```

- Now you have two variables, a_void_fn and an_int_fn. These two variables can be assigned the addresses of the actual functions declared on the previous slide:

```
a_void_fn = hello;
a_void_fn();
an_int64_binop = add;
printf("%d\n", an_int64_binop(2, 3)); // Prints 5
```

What are the use cases of function pointers?

- Many! Dynamic dispatch is an important mechanism under the hood of beloved features in many other programming languages.

- We can write and call functions that take functions as arguments!
  - e.g. higher-order functions such as filter, map, reduce
  - Sorting in Java, qsort in stdlib.h

- We can create "interfaces" for "object-oriented" style programming.

- And more!