Little Languages

Lecture 09:

Arrays & Dynamic Memory "The Heap"

Before lecture: **Start VM and pull 590 materials from upstream**.
Then…

```
$ cd 590-material-<you>
$ git pull upstream master
$ cd 590-material-<you>/lecture/<today>
```

Disclaimer of Overloaded Terminology

- "The Heap" and "A heap data structure" are *completely unrelated*.

- When we are discussing the stack and the heap in a program's memory:

  1. The Stack is short for "call stack" and stores each function call's parameters, local variables, return address, and a few more stateful details.
     - Memory on the stack is always managed automatically for you in the semantics of calling (pushing frames) and returning (popping frames).

  2. The Heap, also commonly referred to as dynamic storage or a program's "free store", is a region of program's memory with fewer restrictions than the stack.
     - Unlike stack memory, whose management semantics are largely the same between programming languages, with heap memory the rules vary widely.

Using a stack/heap diagram, draw an approximation of how you believe this program is represented in memory after the two variables a and b are initialized.

```rust
fn main() {
    // Declare a 2-element vec and array
    let a: Vec<u32> = vec![110, 110];
    let b: [u32; 2] = [590, 590];

    // Print the address of each value
    println!("&a:    {:p}", &a);
    println!("&b:    {:p}", &b);

    // Print the address of first elems
    println!("&a[0]: {:p}", &a[0]);
    println!("&a[1]: {:p}", &a[1]);
    println!("&b[0]: {:p}", &b[0]);
    println!("&b[1]: {:p}", &b[1]);
}
```

Rust

Here are sample memory addresses of each `println`:

```
&a:    0x7ffece7c2e08
&b:    0x7ffece7c2e20
&a[0]: 0x7fc532821008
&a[1]: 0x7fc53282100c
&b[0]: 0x7ffece7c2e20
&b[1]: 0x7ffece7c2e24
```

Stack/Heap Diagram of Previous Example

Technical Interview Practice Question: Why are values on the stack fixed in size?

- In systems languages, like C and Rust, *array* and *struct values* are stored on the stack by default (just like primitive values).
- In both languages, *arrays* and *structs* <u>*cannot*</u> be dynamically resized.
- Pair up with a neighbor and defend reasons why must stack values be fixed in size? Alternative viewpoint: what extra work would be required if stack values *were not* fixed in size?
- Rank your top three reasons and submit the one you feel strongest about.

Arrays in C

```c
// Initialize two 2-element arrays
uint16_t a[2] = { 110, 110 };
uint16_t b[2] = { 590, 590 };

// Print their addresses and contents
printf("&a:   %p\n", &a);
printf("a[0]: %d\n", a[0]);
printf("a[1]: %d\n", a[1]);
printf("a[2]: %d\n", a[2]);

printf("&b:   %p\n", &b);
printf("b[0]: %d\n", b[0]);
printf("b[1]: %d\n", b[1]);
printf("b[2]: %d\n", b[2]);
printf("b[-1]: %d\n", b[-1]);
```

C

- Does the program compile? If so, what is the output of **`a[2]`**?

- Example Output:

Aside: Arrays *are a useful lie we tell ourselves*...

- Just like variables!
  - Alternative definition for *abstraction:* a useful lie with good intentions.

- Programming languages give us syntax and data types to help us believe.
  - But the CPU only has registers and memory addresses...

- Languages differ in the lengths they go to conceal their lies.
  - C lies transparently and doesn't care if it gives you lifelong trauma.
  - Rust takes a nice middle ground opting for safe transparency.
  - Memory managed languages (Java, Python, JS) go to great lengths to be opaque.

Arrays in C are really just pointers to memory

- The C compiler allocates stack space to hold the # of elements requested

- The array's identifier (name) is assigned the first element's address
  - No info about the *size* of the array exists at runtime by default, that's on *you*

- Find the exact location of an element in an array at <INDEX> in memory:

```
element_addr = array_addr + <INDEX> * size_of_an_element
```

<u>This is why programming language designers like indexing from 0!</u>

In C, the *array index operator* is *syntactic sugar* for pointer arithmetic and dereferencing.

```c
// Initialize a 3-element arrays
uint16_t a[3] = { 101, 110, 590 };

// Arrays are pointers? Prove it.
printf("a:          %p\n", a);
printf("*a:         %d\n", *a);
printf("*(a + 0):   %d\n", *(a + 0));
printf("*(a + 1):   %d\n", *(a + 1));
printf("*(a + 2):   %d\n", *(a + 2));
```

C

- You would lose no capability in C without the array indexing operator. You'd just have to access elements via pointer arithmetic:

```
a:          0x7fff88649ff2
*a:         101
*(a + 0):   101
*(a + 1):   110
*(a + 2):   590
```

Syntactical sugar is a nicety for humans. It provides an improved user experience by allowing for more natural expressivity that is usually less verbose.

Fundamentals of *Unmanaged* Dynamic Memory

- When programs need dynamic storage for data structures that can flexibly grow and/or are too large to store on the stack, they need heap space outside of the stack.

- In C, programs request heap space with the functions **`malloc`** or **`calloc`**
  - "Memory allocation" - `malloc` - reserves some *uninitialized* space in memory for you
  - "Clear memory" - `calloc` - reserves some space in memory for you initialized to 0s

- Both functions return the **starting address** of the memory requested to you
  - *Or fail due to lack of sufficient free memory*

- When a C program no longer needs the heap space, the program must call **`free`.**
  - Forgetting to do so leads to memory leaks.

- "Unmanaged" languages like C and C++ require you to ***allocate*** and ***free*** manually.

Example of C arrays on **stack vs heap**

```c
printf("======== stack space  ========\n");
uint32_t stack_array[2] = { 101, 110 };
printf("stack_array:   %p\n", stack_array);
printf("stack_array[0]:%d\n", stack_array[0]);
printf("stack_array[1]:%d\n", stack_array[1]);

// calloc reserves space on heap for based
// on N values * M sizeof a value. It returns
// a pointer to the starting address of the
// memory allocation. The memory is zeroed.
printf("======== calloc heap space ========\n");
uint32_t* heap_array = calloc(2, sizeof(uint32_t));
heap_array[1] = 590;
printf("heap_array:    %p\n", heap_array);
printf("heap_array[0]: %d\n", heap_array[0]);
printf("heap_array[1]: %d\n", heap_array[1]);

// Release the allocated heap memory
free(heap_array);
```

- Notice in the output just how far apart the addresses of these two values are…
  - 46,166,263,395,872 memory addresses apart!
  - That's 46,166 billion bytes away!
  - The sun is 491 billion feet away.
- The important thing to recognize is the stack and heap are disjoint regions in a process's memory.
  - The actual distance is not important. In fact, it's another lie. We'll leave that for COMP530 OS.

```
======== stack space  ========
stack_array:   0x7ffcb7d23c90
stack_array[0]:101
stack_array[1]:110
======== calloc heap space ========
heap_array:    0x55ffcc174670
heap_array[0]: 0
heap_array[1]: 590
```

C

Aside: Where do **`malloc`/`calloc`** go to gain additional plots of memory? To the *operating system*!

```c
// Request memory for 1000 32-bit ints on the heap.
// Generate random # between 100-199 and assign
// to first element. Never free memory.
// Loop infinitely.
for (;;) {
    uint32_t* ptr = malloc(1000*sizeof(uint32_t));
    *ptr = (random() % 100) + 100;
    printf("%d\n", *ptr);
}
```

C

- The library functions you rely on ultimately have to ask the operating system for some extra space via *system function calls* or *system calls*.

- We will talk more about system calls later this semester, but to give a fun preview, let's try running the program left and ***spying*** on all of the calls it makes to the operating system…

Example 03

- $ make && strace ./memory-leak.o
- Each call to **brk** function is requesting additional allocations of memory from the operating system.

Ctrl+Z to kill the process.

Vectors in Rust are Smart Pointers to Heap Arrays

```rust
fn main() {
    // Declare a 2-element vec and array
    let a: Vec<u32> = vec![110, 110];

    // Print the address of each value
    println!("&a:    {:p}", &a);

    // Print the address of first elems
    println!("&a[0]: {:p}", &a[0]);
    println!("&a[1]: {:p}", &a[1]);
}
```

Rust

A `Vec` is a struct (on the stack!) with:

1. A pointer to the elemental array on the heap
2. Total capacity allocated on heap
3. Current utilization of capacity

   When 100%, Vec internals request bigger heap space, move values, and release old.

Notice the address of the Vec a versus the address of its elements…

- 247 billion bytes apart

```
&a:    0x7ffece7c2e08
&a[0]: 0x7fc532821008
&a[1]: 0x7fc53282100c
```

How would we think about the following code in terms of the stack and heap? Its output is revealing.

```rust
let mut a_vec: Vec<u64> = Vec::with_capacity(1);

for i in 0..9 {
    a_vec.push(i);
    println!("{} {:p}", i, &a_vec[0]);
}
```

Output:

```
0 0x7f7db6621008
1 0x7f7db662a010
2 0x7f7db6620060
3 0x7f7db6620060
4 0x7f7db6615040
5 0x7f7db6615040
6 0x7f7db6615040
7 0x7f7db6615040
8 0x7f7db661b000
```

What are some benefits of a Vec's smart pointer design?

1. The smart pointer on the stack is *fixed size* but its storage space can *grow dynamically* on the heap.

2. The heap space does not actually just "grow" like the abstraction leads us to believe. When allocated capacity for the vec is overrun it must reserve a new block of space and copy all elements out.
   - In light of the last lecture on mutable references, try to appreciate how critical it is during this kind of operation no other references/threads are attempting to read or write to the vec simultaneously.

3. If ownership of the Vec is transferred somewhere else (passed as a parameter, returned as a value, stored in another data structure, and so on) notice only the smart pointer needs to move. The much larger data array on the heap can remain fixed in place.

4. Rust is able to *enforce for out-of-bounds access rules* and panic! This prevents a huge class of accidental bugs and security holes in C program arrays.

- Note: We could implement the exact same semantics of a smart pointer in C! If you're working with arrays in C, you probably should… (or rely upon a library that has taken care of it for you).