/unc/comp211

Systems Fundamentals

&Addresses and *Pointers!

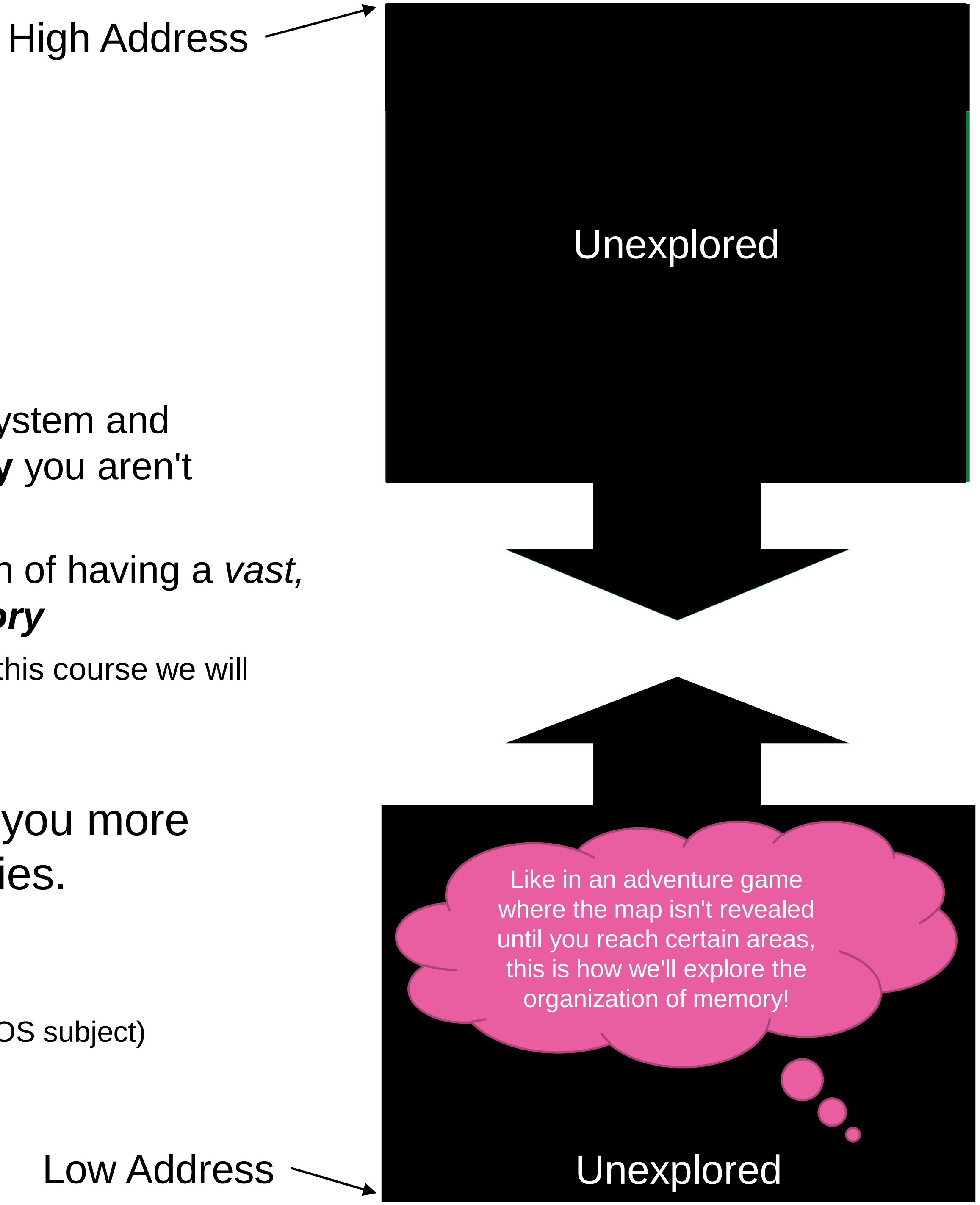
Map of a Process' Memory

- A ***process*** is a program amid execution
  - Many processes can run the same program code

- Each process' memory is isolated* from others'
  - This isolation is provided and enforced by the operating system and hardware. Trying to read or write to **segments of memory** you aren't allowed leads to a ***segmentation fault (program crash)***.
  - The operating system (OS) gives each process the illusion of having a *vast, contiguous* memory address space through ***virtual memory***
    - Virtual memory is an important topic taught in the OS course. In this course we will embrace the abstraction of virtual memory!

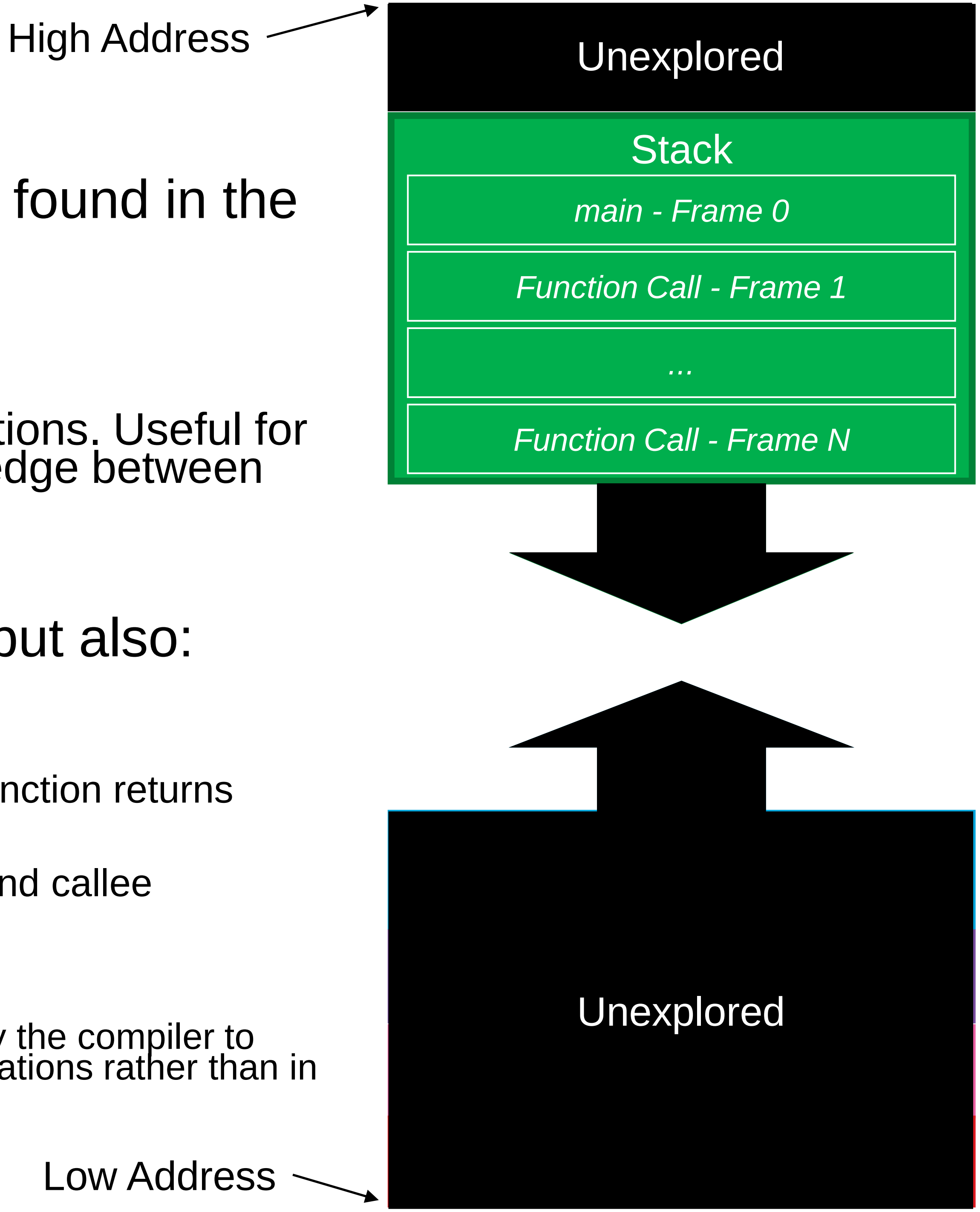
- A key characteristic of a *systems* language is it gives you more direct access to memory-level concerns and capabilities.
  - *With great power comes great responsibility!*

* Processes *can <u>intentionally</u>* share memory if they want to (shared memory also an OS subject)



The Call Stack

- Variables local to function calls are typically* found in the "Stack" or *function call stack*
- Each function *call* has a *frame* on the stack.
  - This enables separation of storage between functions. Useful for separating concerns between and limiting knowledge between functions. Critical for generalized recursion.
- Frames not only contains variable's values, but also:
  - Argument values
  - Return Address
    - Where in the program to resume execution once a function returns
  - Return Value
    - Space where return value is shared between caller and callee
  - Additional CPU State*



* In 311 you will learn about CPU registers which can also be used by the compiler to store the contents of a variable directly in a CPU's limited storage locations rather than in the virtual memory system.

& The Address of Values in Memory

- The C programming language has an "address of" operator which evaluates to the address of its operand in memory.
  - A memory address depends on the *word size* of a computer. On your 64-bit laptop the word size is 64 bits. Think of it as a (very large!) unsigned integer.

- To print a memory address with **printf**, use the **%p** format specifier
  - The **p** stands for **pointer**, which is a name for a memory address value

- Consider the output right:
  - Notice how **enormous** these hex values are! As an unsigned integer **&b** is 140,736,771,505,430
  - Also notice since a **char** is a single byte, the addresses of a and b are right beside one another.

- Takeaway: The **stack** is in **high** memory addresses. Local variables are collocated within their stack frame.

Consider the following code...

```c
#include <stdio.h>

int main()
{
    char a = 'a';
    char b = 'b';

    printf("&a: %p\n", &a);
    printf("&b: %p\n", &b);
}
```

... and it's output:

```
&a: 0x7fffd545bd16
&b: 0x7fffd545bd17
```

What are the 2 missing hex digits?

```c
uint16_t a = 1;
uint16_t b = 2;

printf("&a: %p\n", &a);
printf("&b: %p\n", &b);
```

```
&a: 0x7ffc1a7e9524
&b: 0x7ffc1a7e9526
```

```c
uint32_t a = 1;
uint32_t b = 2;

printf("&a: %p\n", &a);
printf("&b: %p\n", &b);
```

```
&a: 0x7ffff088d700
&b: 0x7ffff088d7 ?
```

Visualization of Type Widths

- Notice that the *type* of a variable establishes its *bit-width* in memory.
- Variables whose types are larger than 1-byte span *multiple* addresses in memory!
  - We will assume "little endian" meaning when a value spans multiple addresses, its low-order bits will be in the low-address end of the range.
- Reminder: variable names are for humans only! The compiler does the bookkeeping to produce machine programs that operate in terms of memory addresses with no memory representation of variable names.

```
uint16_t a = 1;
uint16_t b = 2;

printf("&a: %p\n", &a);
printf("&b: %p\n", &b);
```

	Address	$Contents_2$	$Contents_{16}$	$Contents_{10}$
	XXX7	00000000	00	0
	XXX6	00000000	00	0
	XXX5	00000000	00	0
	XXX4	00000000	00	0
b	XXX3	00000000	00	0
b	XXX2	00000010	02	2
a	XXX1	00000000	00	0
a	XXX0	00000001	01	1

```
uint32_t a = 1;
uint32_t b = 2;

printf("&a: %p\n", &a);
printf("&b: %p\n", &b);
```

	Address	$Contents_2$	$Contents_{16}$	$Contents_{10}$
b	XXX7	00000000	00	0
b	XXX6	00000000	00	0
b	XXX5	00000000	00	0
b	XXX4	00000010	02	2
a	XXX3	00000000	00	0
a	XXX2	00000000	00	0
a	XXX1	00000000	00	0
a	XXX0	00000001	01	1

Aside: Why do addresses change each time a program runs?

```
learncli$ ./a.out
&a: 0x7ffc02bf4836
&b: 0x7ffc02bf4837
learncli$ ./a.out
&a: 0x7ffea9951166
&b: 0x7ffea9951167
learncli$ ./a.out
&a: 0x7ffffb676f66
&b: 0x7ffffb676f67
learncli$ ./a.out
&a: 0x7fffec4261d6
&b: 0x7fffec4261d7
learncli$ ./a.out
&a: 0x7ffd59137866
&b: 0x7ffd59137867
```

- Running any of the previous programs results in different output *every single execution.* Why?

- The operating system *intentionally* randomizes the starting address of the call stack every time a program runs
  - ASLR - Address Space Layout Randomization

- Why? Hacking programs becomes more difficult when the exact addresses of data have some randomness.

- Notice it's not all random, though. The *high-order* bits keep the stack starting in the same general vicinity of addresses:
  - 1.406e14 through 1.407e14
  - Full precision: 140,668,768,878,592 through 140,737,488,355,327

How are addresses of local variables related across multiple stack frames?

```c
#include <stdio.h>
#include <stdint.h>

uint64_t sum(uint64_t x)
{
    printf("x: %lu - &x: %p\n", x, &x);
    if (x <= 1) {
        return x;
    } else {
        return sum(x - 1) + x;
    }
}

int main()
{
    printf("sum(3): %lu\n", sum(3));
}
```

Pointers are Addresses to Memory

- In systems programming languages pointers are a first-class data type
  - Can be stored in variables, passed as parameters, returned from functions
  - You can *dereference* a pointer to read that memory address's contents

- What is the point of pointers? Why haven't you needed them before?
  - It turns out you *have* needed them before.
    - Java: Reference types (arrays, objects) are opaque pointers to heap values.
    - In "memory managed" languages you have limited control of and visibility into pointers, but they're very much there!
  - Big Idea: Pointers enable sharing data structures between function calls without having to copy the structure
    - It would be expensive to copy large data structures as arguments to a function call only to have to copy it back to the caller's frame
    - Keep this in mind because our early demos will show pointers to *simple, primitive values* for illustrative purposes
  - Other use cases: Many!
    - Efficient iteration through arrays.
    - Sorting strings and objects without having to move their values in memory.
    - Dynamic dispatch of functions.

Follow Along - pointer_demo.c

```c
#include <stdio.h>

int main()
{

    char a_char = 'z';

    // Declare a pointer to a char value
    char *a_char_ptr;

    // Assign an address to a pointer
    a_char_ptr = &a_char;

    // Read the address stored in a pointer
    printf("a_char_ptr: %p\n", a_char_ptr);

    // Read the address referenced by the pointer
    printf("*a_char_ptr: %c\n", *a_char_ptr);

    // Write to the address referenced by the pointer
    *a_char_ptr = 'y';

    // We changed the referenced value!
    printf("a_char: %c\n", a_char);

}
```

C Pointers - Basic Operators and Operations

- Declaring a pointer variable:
```
<type> *<identifier>;
```

- Example:
```
char *a_char_ptr;
```

- Assigning the address of a variable to a pointer:
```
char a_char = 'z';
a_char_ptr = &a_char;
```

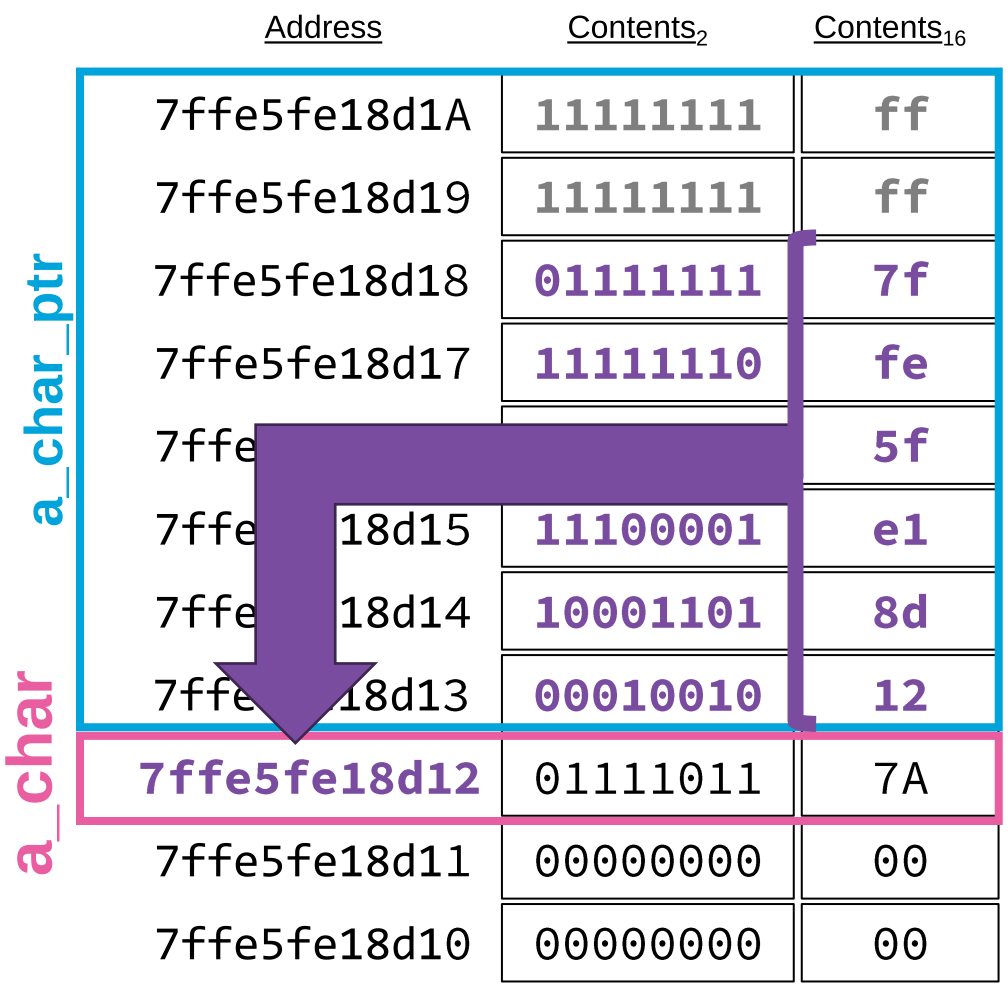
- Access the pointer's value
```
printf("%p\n", a_char_ptr); // Prints the address of a_char
```

- Read from the memory address referenced by a pointer - **dereference** read
```
printf("%c\n", *a_char_ptr); // Prints 'z' in this example
```

- Write to the memory address referenced by a pointer - **dereference** write
```
*a_char_ptr = 'y';
printf("%c\n", a_char); // Prints 'y'
```

Visualizing a Pointer in Memory

- Since 64-bit memory addresses are themselves 64-bits wide, to store an address in a pointer requires 8 bytes as shown for **a_char_ptr**
    - Aside: On today's processors only 48-bits of the possible 64 bits are used. Why? 48-bits can address up to 256 Terabytes of memory. That's 16,000+ times more memory than your laptop has. In the future those top 16-bits can be used. That's why those bytes are grayed out.



- BIG IDEA: The **address of a_char** is what is stored in the contents of **a_char_ptr**!

What is the output?

```c
#include <stdio.h>

void add1(char *char_ptr);

int main()
{
    char a_char = 'x';
    printf("%c\n", a_char);

    // Address of expression argument
    add1(&a_char);
    printf("%c\n", a_char);

    char *a_char_ptr = &a_char;
    // Pointer argument
    add1(a_char_ptr);
    printf("%c\n", a_char);
}

void add1(char *char_ptr) {
    *char_ptr = (*char_ptr) + 1;
}
```

Pointer Parameters

- Parameters can be pointers
  - In doing so, it gives functions the ability to read from and to memory addresses they otherwise would not have access to.

- Previous ex modified **`main`**'s **local** variable... ***from outside its scope!***
  - If this sentence doesn't scare you a bit, keep reading it until you're scared.

- You have *kind of* seen this before in Java / Python / TypeScript when you have parameters of type array or object (reference types)
  - In those cases what you're *actually passing* are pointers to the same array/object
  - However, in these languages it is impossible to pass pointers to primitive locals. No such restrictions exist in systems languages like C because you are working more transparently at the memory address level.

- We will see this is commonly done when writing "object-oriented style" C

What the hidden output?

```c
#include <stdio.h>

int main()
{
    char a[] = { 'c', 'o', 'm', 'p' };

    printf("&a[0]: %p\n", &a[0]);
    printf("&a[1]: %p\n", &a[1]);
    printf("&a[2]: %p\n", &a[2]);
    printf("&a[3]: %p\n", &a[3]);

    char *p = &a[1] + 1;
    printf("*p: %c\n", *p);
    p -= 1;
    printf("*p: %c\n", *p);
}
```

```
&a[0]: 0x7fff78d8e5c4
&a[1]: 0x7fff78d8e5c ?
&a[2]: 0x7fff78d8e5c
&a[3]: 0x7fff78d8e5c
*p:  ?
*p:
```

Addresses are *Just* Numbers (with Context)

- You can perform (limited) arithmetic on pointers and addresses
  - You can add and subtract integers from pointers
  - You can subtract two pointers of the same type

- Most useful when working with pointers to array elements
  - Sometimes for hackier reasons

- The actual byte arithmetic *is contextual to the pointer's type*
  - If you were adding one to the address to a uint32_t variable, such as &a_uint32 + 1, the result would increase the address by 4 bytes!
  - Implicitly, the number being added or subtracted from the pointer is being scaled by the type's byte width.

Array Indexing vs. Pointer Arithmetic

- An array variable in C is a special pointer to address of first element
  - Different than a plain pointer because it cannot be reassigned
  - Also different because sizeof(array) reports size in bytes of complete array

- Array indexing notation is *just* syntactic sugar for pointer arithmetic:
  - **`a[i]`** is the same as **`*(a + i)`**
  - **`a[0]`** is the same as **`*(a + 0)`** and the same as **`*a`**

- Since an array's name is just a pointer to its first element, you can assign it directly to a pointer of the same element type:

```
int numbers[] = { 10, 20, 30 };
int *p = numbers;
printf("%d", *p); // Prints 10
```