- Jinlang Wang
- Office hour: Thurs 2:15pm – 5:00pm

Pointers

Memory

- Before we talk about what pointers are… it's helpful to understand memory a little more!

- Memory is …. a huge one-dimensional array of bytes
- Every byte has an associated address
  - That address is its **array index**
- For values **bigger than a byte**, we scale up, and use consecutive bytes!
- But… regardless of size, the address of any value is the **address of the first byte**!

The sizeof() function

- sizeof() is a **compile-time** operation which tells you how many **bytes** something takes up!

EXTREMELY IMPORTANT:

- C does not know how big an array of a pointer is!!
- All pointers in C will be 8 bytes.
- NEVER INVOKE SIZEOF ON A POINTER

Pointers: An Analogy

Lockers

1) Let’s think about lockers… what’s their purpose?
   - … to store things…

2) How do we identify lockers from one another?
   - Using their locker numbers…

3) How do you access a locker?
   - By knowing the locker number and combination….

4) If I wanted to give someone else access to my locker, what would I do?
   - give them the locker number!

Locker Rooms

Just like locker **stores things**

- A variable stores things…
- But a variable is a thing itself…

Each variable is just like a locker!

- It has a **number**: its **address**
- It stores something: its **value**
- It **belongs** to someone: its **owner**

How would I **give someone else access** to my variable?

- Give'm the locker number…
- Which is… the **memory address**!
- So what's a variable?
  - It's just a way to conveniently refer to their memory address!

Pointers!!!

- A **pointer** is a variable which holds **another variable's memory address**.
- Once you have a pointer, you have access to two things:
  1) The pointer **itself**
  2) The variable it **points to**

Pointers

What is a Pointer?

- A **pointer** is a variable that holds a memory address of another variable.
- Essentially, anytime you would use an array in Java, you'd use a pointer in C.
- You can access a value a pointer points to using the **dereference operator (*)**.
- C uses pointers because it's easier to say, "that's the place that has that data" rather than saying "This is the entire thing that which includes the data I'm interested in!"

Another thing… Arrays don’t Exist in C!

- Basically, arrays are considered as local variables in C.
- Meaning, we can’t return an array in C!
- So, what do we do instead?
  - We return a pointer to the array instead!
- BIG POINT: **Arrays become pointers when passed into functions!**

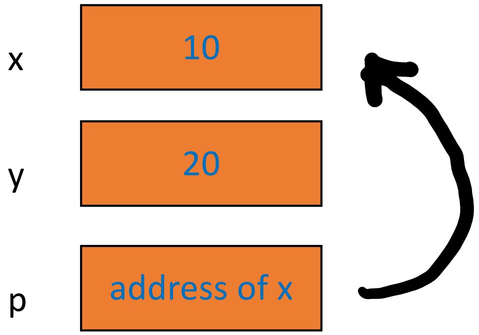
Address-OF (&) Operator

- You can get the address of a variable via the **address-of operator (&)**

```
int x;
int* p = &x;
```

Pointers Example

```
int x = 10;
int y = 20;
int* p = &x;
```

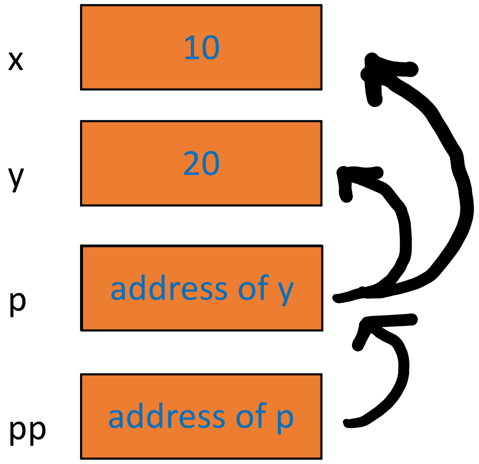


x	10
y	20
p	address of x

We say "p points to x"

Pointers Example

```
p = &y;
int** pp = &p;
```



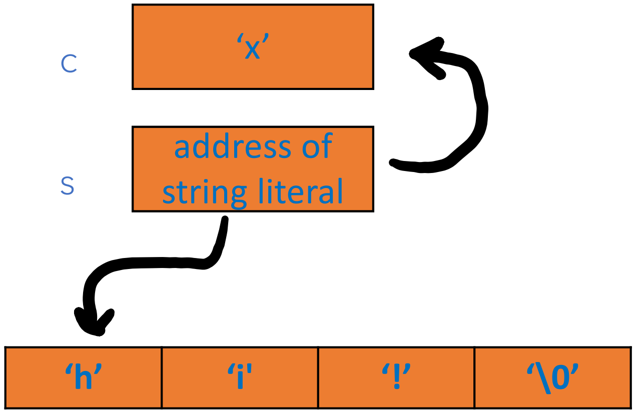
Pointers and Arrays

Pointers and Arrays

- A pointer can point to **one or more values**.
- A **char*** may point to a single **char**, or to an **array of chars**.

Arrays Example

```
char c = ‘x’;
char* s = &c;
s = “hi!”;
```



“Pointing to a single value” is the same as “pointing to an array of length 1”!

Accessing the Value at a Pointer

The Value-At (Dereference) Operator

1) ***** is the value-at operator: it's the **inverse of &**

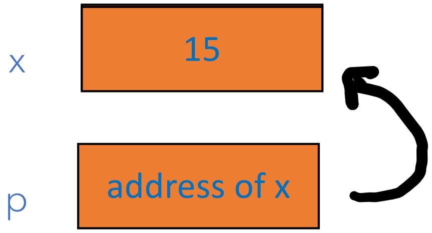
- Every time you use it, you ***remove*** a star.

2) It **accesses the variable** that a pointer **points to**

- We say that it "**dereferences**" a pointer

Pointer Example

```
int x = 10;
int* p = &x;
*p = 15; //changes x!
```



Changing the value of a pointer (via dereference) will change the original value!

Array-Indexing Operators

The Array Indexing Operator

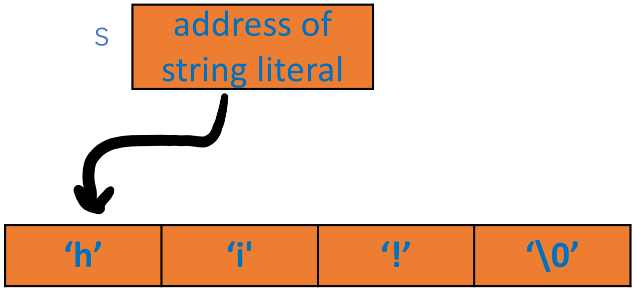
- **p[n]** means "access the **nth** item pointed to by **p**."

char* s = "hi!";

char c = s[2];

char d = 2[s];

Now c and d are the same thing…



What the Brackets Actually do

p[n] in C really means "dereference address **p + n**"

s[2]
= *(s+2)
= *(2+s)
= 2[s]

What about values **bigger than a char**?

Scaling

- When we add an offset to a pointer, the offset is **multiplied by the size of the item being pointed to** before being added to the base address.
- For int pointers, we scale by 4.

int arr[3] = {1, 2, 3};

int* p = arr;

*(arr + 1) → *(arr + 1*sizeof(int)) → *(arr + 4)

Pointer Arithmetic & Void Pointers

Memory Addresses are just Numbers

1) Pointers hold memory addresses…

- Memory addresses are just **numbers**.

2) It's incredibly useful to do **arithmetic** on **memory addresses**

- No dereferencing is involved in pointer arithmetic.
- We are operating ***on the pointer itself!***

Strings in C

Strings are just Char Arrays

- Strings are just sequences of characters!
- In C, we indicate the end of a String using a **NULL TERMINATOR: '\0'**
- If we lose track of the NULL TERMINATOR, we're pretty much screwed.

Strings in C

- The end of a string is indicated by a **NUL Terminator** (‘\0’)
- There are two ways to initialize strings in C, by a **char array**, or a **char pointer**

1) **char mystr[100] = “hello”;**

Allocates space for 100 characters, fills array with characters up to the length of the string, and fills the rest of the slots with **‘\0’**!

2) **char* mystr = “hello”;**

Allocates the string in the **static data segment**

- Allocates space for the **pointer**
- Don’t do this if you want to do **String Manipulation!**

Basic String Functions

- **strlen()**: Scans entire string for a **‘\0’** and return count of iterations to get there
- **strcmp():** compares two string and returns comparison value (compareTo in Java)
- **strcpy(a,b):** copies string from **b** into **memory** at **a**
- **strcat(a,b):** copies string from **b** into **memory AFTER a**.

- Avoid string manipulation at **ALL COSTS in C**!!

String Manipulation Example

Suppose we start out with a **char array mystr: char mystr[100];**

mystr

0	1	2	3	4	5	6	7	8	9

If we **strcpy** the string, "**this**" into **mystr**:

mystr

0	1	2	3	4	5	6	7	8	9
t	h	i	s	\0					

If we **strcat** the string, " **is**" into the modified **mystr:**

mystr

0	1	2	3	4	5	6	7	8	9
t	h	i	s		i	s	\0		

String Interning

Have you ever wondered….

- I have the following code:

```
String s1 = “537”;
String s2 = new String(“537”);
s1==s2 //returns false!!
```

Memory Pools

Here's what's actually going on in the background:

- s1 points to a pool of memory referred to as: "Non-heap Memory Pool"
- s2 points to a pool of memory referred to as: "Heap Memory Pool"

So, when we compare s1 == s2, this returns false because the pointers are pointing to different memory pools

String Interning

How would we get around this?

- String interning

We can forcefully make the pointer pointing to the Heap Memory Pool point to the Non-Heap Memory Pool.

This will work:

- s1 ==s2.intern() //returns true