C Pointers and Arrays

Types

- Programs deal with all sorts of values
  - `12, 3, 1.3, "name?", 'x', {"grapes", 3.99, 40}`
  - Some of these you cannot add 1 to like a regular number
    - Strings concatenate when you add an int
  - You can use [ ] on strings, not on a single char
  - You can use . on structs
  - You cannot use the same kind of variable for all of these
- Types are how we categorize values
  - Based on what we can do with those values
  - Types are what you put in front of the variable names
    - int, double, etc.

Type constructors:

- Different from Java constructors
- If you have a value, you can make a new value based on it
  - Ex: if you increment 5, you get 6

Pointers:

- Memory is a big, one-dimensional array of bytes
- Every byte value has an address
  - This is its array index
  - Addresses start at 0
- For values bigger than a byte:
  - Use consecutive bytes
    - A chunk of bytes in a row
- The address of any value, regardless of size, is the address of the first byte
  - The SMALLEST address
  - `sizeof(int) = 4`
  - int address is `CODE`
  - It's value?
    - `DECOFBE/BEEFCODE` depending on endian
- Lockers:
  - Store stuff in a locker, you know which one is yours based on the number, numbered sequentially
  - How do you access a locker?
    - Knowing the locker number and combination
    - Assume no locks for example
    - How do you give someone access to a locker?
      - Give them the locker number
  - Similar to lockers,
    - Variables contain values

- But a variable is a thing itself
- Each variable is like a locker
    - Number = address
    - Contains something = value
    - Belongs to someone = owner (scope)
    - Give someone else access to your variable?
        - Give them the memory address
    - What if we put the slip of paper inside a locker itself?
        - Now we can access:
            - The locker itself (3)
            - The locker it points to (2)
            - A pointer is a variable which holds another variable's memory address
                - Can access two things
                    - The pointer variable itself and the variable it points to

- Pointers in C
    - Pointer variables and values
        - C has * as a type constructor
            - `int x;  //integer`
            - `int* p;  //pointer to an int`
            - `int** pp;  //pointer to a pointer to an int`
                - `char** argv`
        - You get the address of the variable with the address-of operator (`&`)
            - Give something, and gives the address of something
                - `int* p = &x;`
                - `int** pp = &p;`
            - You can use it on just about anything with a name
                - `&x  //address of x`
                - `&arr[10]  //address of item 10 in arr`
                - `&main  //address of main function`
            - Can't do:
                - `&5, &&x`
                - Can't get address of a temporary
        - A pointer can point to one or more values
        - A char* may point to a single char or to an array of characters
    - Multi-dimensional arrays
        - Why do we want double pointers?
        - `int** arr2D = …`
            - Each item in the array points to an array of integers
            - In Java: `int[][]`
        - Can pass char dict[][20] to functions
    - Printing pointers

        - `%p`
        - `printf("address of x = %p\n", &x);`
            - Hexadecimal
        - Pointers can be `null`
            - `int* p = NULL;`
            - `printf("p = %p\n", p);`
                - Prints null
                    - Could be different depending on system
            - In C its possible to have a pointer that is not null, but is invalid
                - Accessing an undefined pointer gives undefined behavior
- Arrays are weird with `&`
    - You can get their address by using their name alone or with the address-of operator
- Accessing values at a pointer
    - Value-at (dereference) operator *
        - Inverse of `&`
        - Takes an int pointer and gives you an int
        - Each time you use it, you remove a *
        - Access the variable that a pointer points to
            - Dereferences a pointer
            - `*p = 15;  //changes x to 15`
            - Printing `*p` prints `15`
    - The `->` operator
        - If you have a pointer to a struct, you must access its fields with `->`
            - `grapes.stock--;`
            - `food * pgrapes = &grapes;`
            - `pgrapes->price = 2.99;`
            - When we pass structs to other functions, we use pointers so we can change the struct
    - Array indexing operator
        - `p[n]` means "access the nth item pointed to by p"
            - Can do `2[s]` which is weird
- Pointer arithmetic
    - Pointers hold memory addresses, which are just numbers
    - Can do arithmetic on memory addresses
    - Calculate a new pointer based on an old one
        - `printf("%c\n", *t);  //prints an !`
        - `t[0]  //is !`
        - `t[-1]  //prints i`
        - `t[-3]  //prints undefined behavior`
    - What the brackets really do
        - `p[n]` in C means dereference address p + n

        - `s[2] → *(s + 2)`
    - “string” + x does pointer arithmetic and not concatenation
    - Original pointer is base and number we’re adding is the offset
- Array of ints
    - Memory addresses go up by size of int when you index into it
    - When you 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
        - Called scaling