* The c programming language has an “address of” operator which evaluates to the address of its operand in memory
* To print a memory address use printf with %p
* Variables whose type are larger than 1 byte span multiple addresses in memory
* Addresses change each time a program is run, but they are still within a certain range. If you look at the memory addresses each time a program runs, the higher order bits will be the same, say the top 3 for example. This is beneficial because it makes hacking more difficult.
* In system programming languages, pointers are a first-class data type. Meaning they can be stored in variables, passed as parameters, returned from functions. You can dereference a pointer to read that memory address’s contents
* In “memory managed” languages lie Java you have limited control of pointers but they are there.
* Pointers enable sharing data structures between function calls without having to copy the structure. It is expensive to copy large data structures as arguments to a fnction call and copy it back to the caller’s frame.
* Pointers can also be used for efficient iteration through arrays, sorting strings, etc
* Declaring a pointer: <type> \*<identifier>. Ex: char \*mychar. **See slide 27 for more syntax**
* Big idea: the address of a variable is what is stored as the value of a pointer to that variable.
* A char only requires 1 byte of space but pointers use 8 bytes.
* Slide 30: the output is x, y, z. because you adder adds 1 to the value of a\_char (not the addres) each time it is called. This example modified main’s local variable from outside of main.
* Slide 35: 5, 6, m, o
* Addresses are just numbers with context. Yu can perform limited arithmetic on pointers and addresses: add and subtract integers from pointers, subtract 2 pointers of the same type. This is useful when working with pointers to array elements (pointer arithmetic). The byte value that is added to the pointer by doing + 1 is based on the type of pointer (char pointer vs int pointer).
* In c, an array is just a name for the address of the array’s first element. The sizeof(array) returns the array’s total size in bytes.
* Slide 45 ex: the digits array is 4 chars which is 4 bytes. The letters array is a string which is 4 chars so 4 bytes also.
* C arrays and pointers to arrays are not the same. An array’s name is an identifier that evaluates to the address of its fuirst element, but the array’s name is not a variable and cannot be reassigned. This is like a restricted case of a pointer. A pointer is a variable that holds the address of another variable, so a poiter can be reassigned.
* Array indexing vs pointer arithmetic: a + I = &a[i]. given array a, the prev expressions are equivalent for computing the address of array element i. \*(a + i) = a[i] for reading element i.
* The sizeof operator: returns the number of bytes of its operand. This is an operator, not a function. This gives it a superpower: a type name is a valid operand. Ex: sizeof(int). the return type of sizeof is size\_t, whose size in memory depends on the machine (just like pointers). Ue format specifier %lu (long unsigned) to print a size\_t. **see slide 61 for example**.
* With strings you have an ending mark ‘\0’, but with arrays you need to know how many times to iterate to get to the end.
* Slide 63: output = 10.
* Slide 64: output = a: a, b: big, c: cake. It prints the first letter of each array followed by the entire string in each array. The size of words should be the number of elements time sthe size of each so 8 chars \* 1 byte = 8 bytes. Wrong. The size is 24 because words stores 3 pointers to char arrays and a pointer is 8 bytes so 8 \* 3 = 24. The 8 bytes taken up by the actual words is stored somewhere else. **See slide 65 for diagram**.

Structs

* A struct in c is a group of related variables. Each variable in a struct is a member (aka property). It is like a class with only public properties and no method or constructor. **See slide 68-73 for syntax**. You can zero-initialize or manually initialize each member, but always choose one or the other – don’t use a combo of both.
* Typedef keyword defines another name for another type. **See 76 for syntax**. When declaring struct arrays and variables, you don’t want to have to write the struct keyword at every declaration, so you use typedef keyword instead. This lets you give the struct a new name and then you don’t have to worry about keywords when declaring (or something like that idk). You can do all of this easily with the example syntax on **slide 82**.