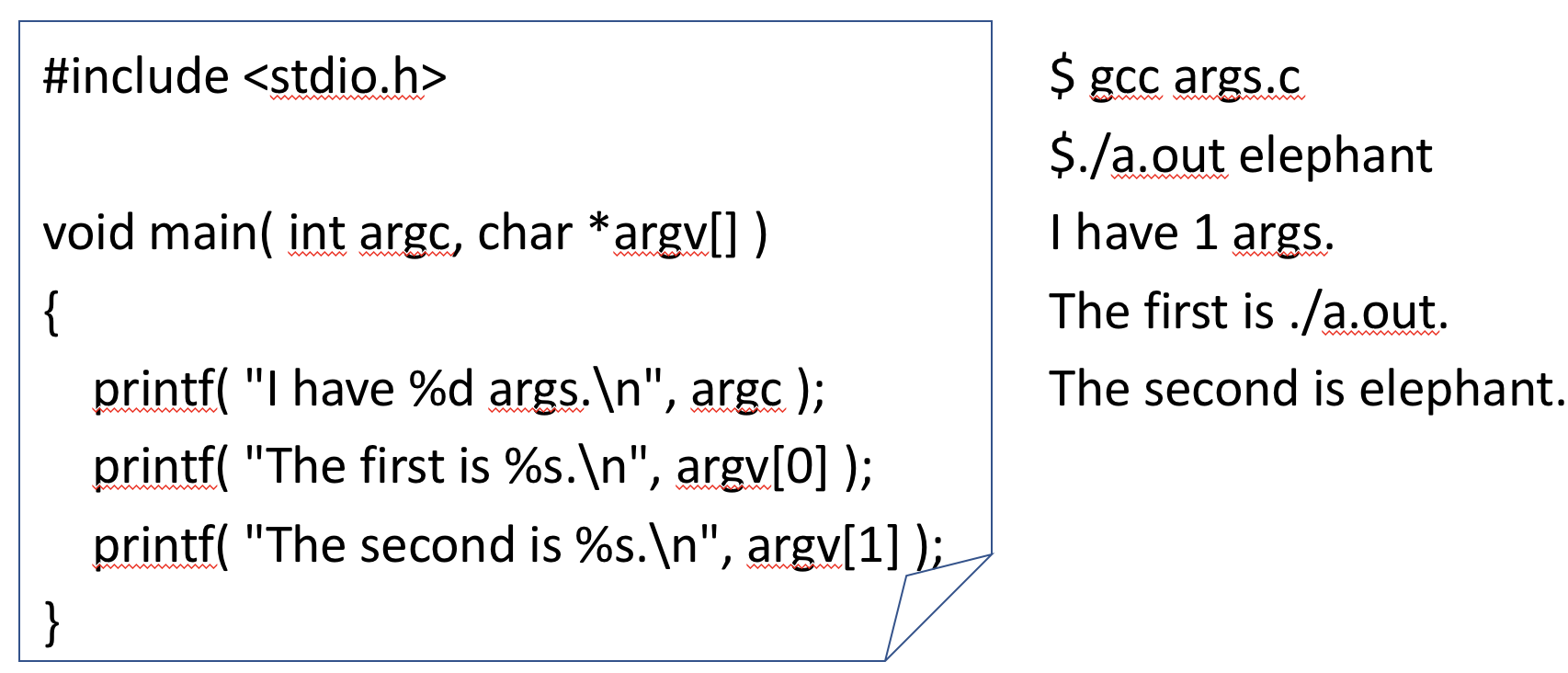
# COMP206-C

* Datatype
  + Integer numbers: **int**
  + Floating point numbers of single precision: **float**
  + Floating point numbers of double precision: **double**
  + Characters to represent text: **char**

Modifiers

* + Short and long: change the number of bits used to represent an int
  + Signed and unsigned: determine whether one can represent negative numbers
  + Pointers store addresses in the computer's memory
* Accessing Program Argument



Function

* A return type (can be void)
* A name (must be unique in the program)
* An argument list with types and variable names (can be empty)
* A function body enclosed in {}

Array

In order to store more than one value within the same variable, C provides simple arrays:

int hourly\_prices[24];

**We must provide the size when declaring the variable**, but it can be left off when specifying a function argument:

int main( int argc, char \*argv[] )

In this case, the size is decided by the calling function (e.g., here we can type any number of arguments

String

* Single Characters

**Single quotes for literals**

* + char char\_variable = ‘w’;

**Math allows moving alphabetically forward or backwards, finding relative positions**

* + char\_variable ++; (it now = ‘x’);
  + char\_variable - ‘a’ (tells you what position in alphabet, 23 here)

**Logic works via alphabetical order**

* + char\_variable == ‘x’ (evals true)
  + char\_variable == ‘X’ (evals false)
  + char\_variable > ‘z’ (evals false)
* Strings

char name[at least 6]

**Arrays of characters** (no such type “String” in C)

* + E.g. char name[100] = “David”;

**Each element is a character stored using the ASCII table**

* + A mapping between our printable letters and the 0’s and 1’s in memory

**Must be “null terminated” with the special ‘\0’ NULL value, with ASCII integer representation 0.** We get \0 at the end of our string for free when we create it with the “…” syntax. But this can lead us to start forgetting to add it when we create, e.g.,

char str[10];

str[0] = ‘h’;

str[1] = ‘i’;

str[2] = ‘\0’;

Claim: Without the \0, printf could not work.

**Double quotes for literals:**

* + char str\_var[100] = “hello”;

**Doesn’t work with math**

* + str\_var++; (error);
  + str\_var – “jello” (nothing related to ’h’ – ‘j’)

**Logic operators don’t work**

* + str\_var == “hello” (incorrect)
  + str\_var > “jello” (incorrect)
* Break down the string into its characters

char str\_var[100] = ”hello”;

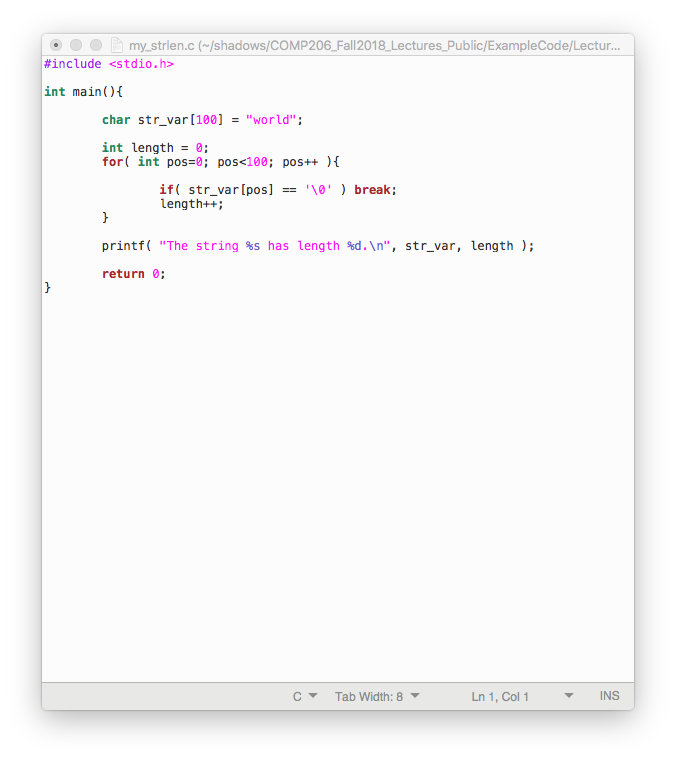
for( int pos=0; pos<100; pos++ ){

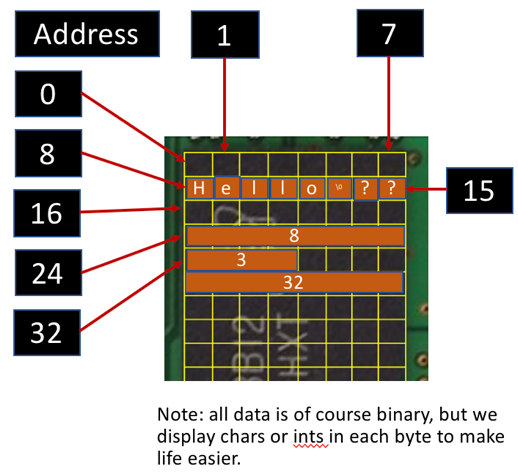
if( str\_var[pos] == ‘\0’ ) break;

printf( “%c”, str\_var[pos] );

}

* Find length



pointer

*A pointer is variable that stores an address.*

Pointers allow us to move around our strings (think iterators, lists indices):

char str\_var[100] = “Hello”;

char \*start = str\_var;

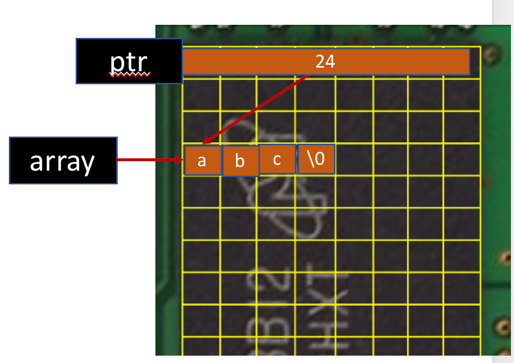
char \*mid = str\_var+3;

* Declare a pointer with:

TYPE \* VARNAME;

Star can be anywhere between the type and var

VARNAME holds the value “the address of a variable or array of TYPE”

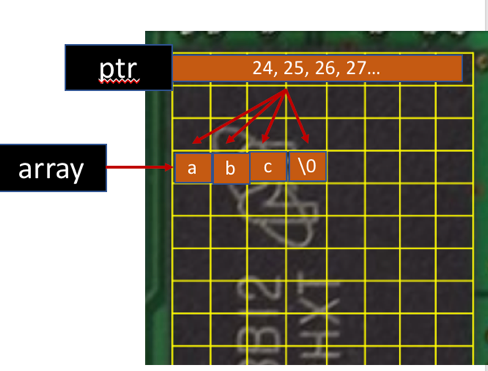


* Pointers and Arrays:
* In some ways, they are interchangeable

An array in C is implemented as the address of its first entry. So, we “point to” the array

char array[4] = “abc”;

char \*ptr = array;

* In some ways they differ:

The array variable holds the address to the start of this memory always, while the pointer is more flexible

char array[4] = “abc”;

char \*ptr = array;

ptr++; ptr++; ptr++; ptr++; // These work

array++; // This is an error!

**Pointing to start of literal works**

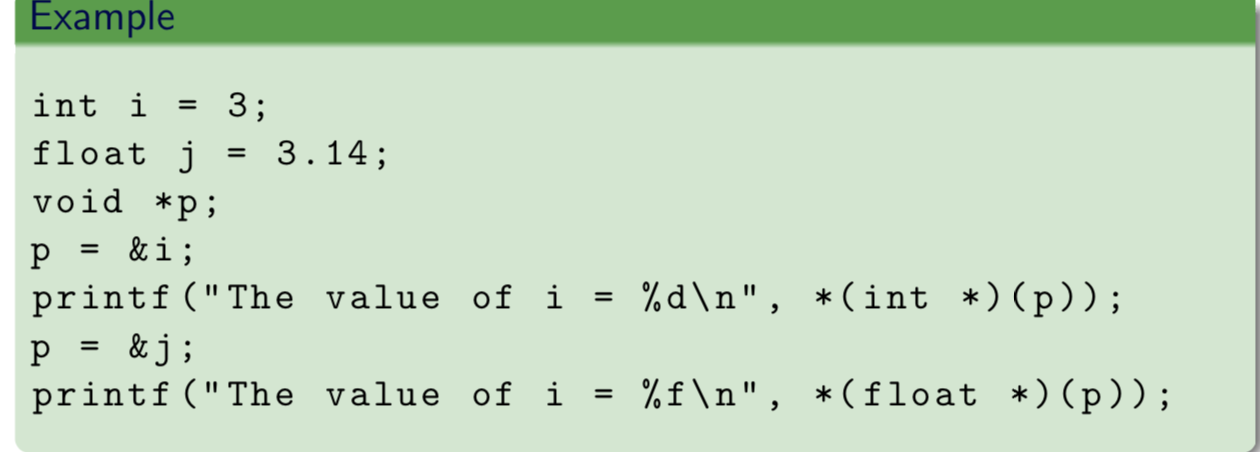
* char str\_array[100] = “hello”;
* char \*ptr = str\_array;

**Pointer math moves us around the string and computes distances**

* ptr = ptr + 3; // Now points to lo
* ptr = ptr – 1; // Back to llo
* char \*ptr2 = str\_array;
* ptr – ptr2; // Gives position difference, 2 here

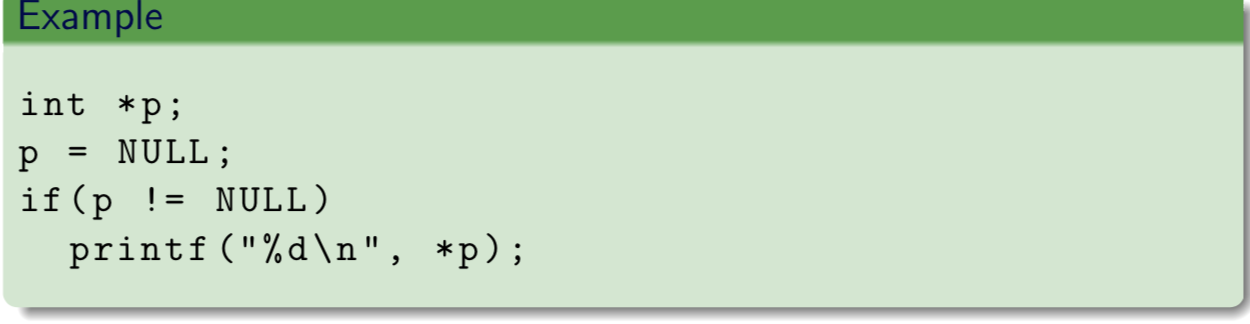
**Logic is based on the pointer position**

* ptr = ptr2; // False, not same spots
* ptr > ptr2; // True, ptr is farther along



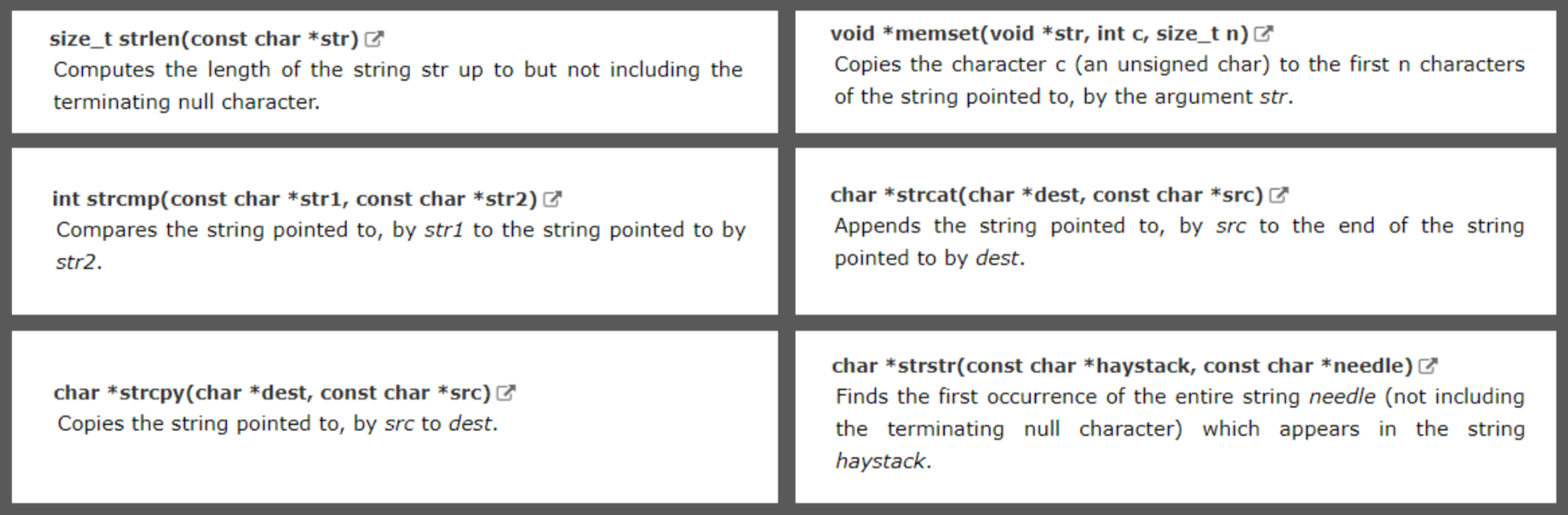
3

3.14



don’t need to be 0

sting.h



**stdio.h**

