COMP 206 – Introduction to Software Systems (guest lecture by Prof. Greg Dudek)

Lecture 11 – 2D Arrays

October 5th, 2018

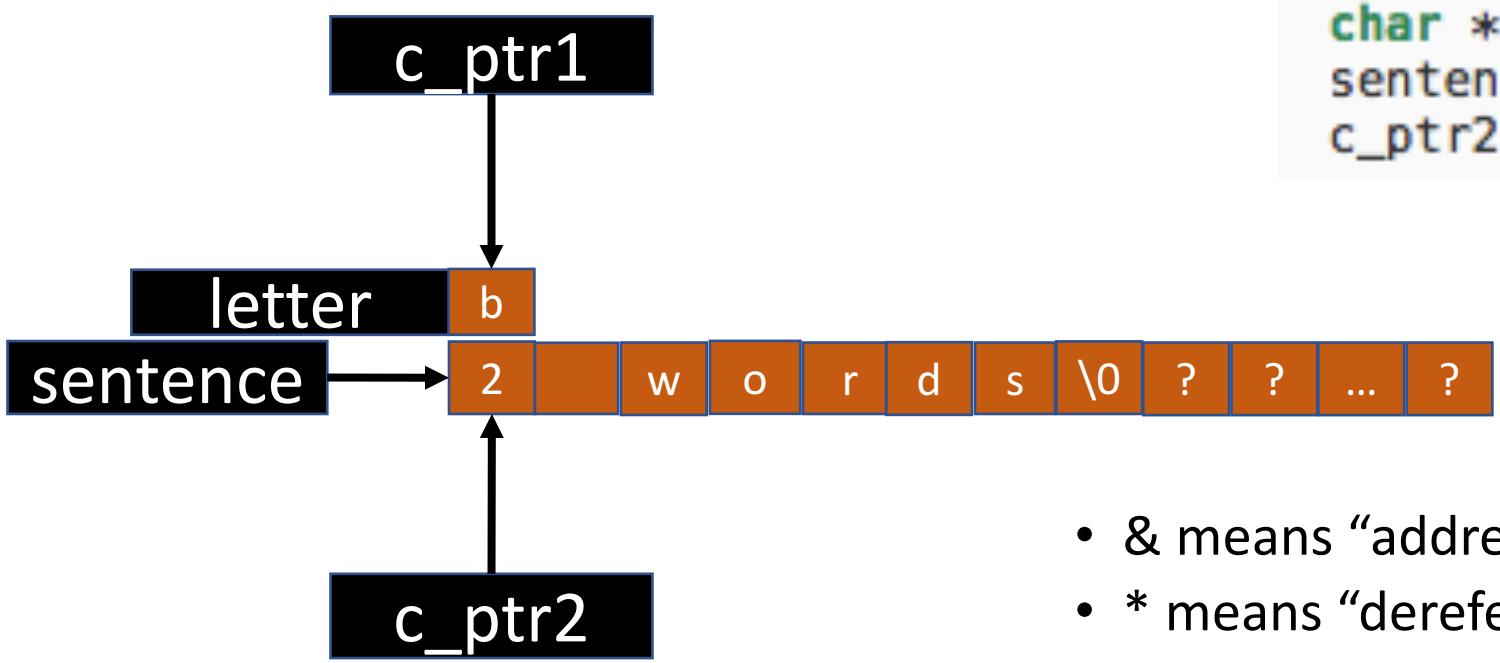
Quick Review

- The code on the right should be starting to feel comfortable.
  - Available on Github under ExampleCode/Lecture11-2DArrays/pointer_review.c
- Read through and think it over, then we'll review to get started.

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

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

        char letter = 'b';
        char sentence[100] = "2 words.";
        char *c_ptr1 = &letter;
        char *c_ptr2 = sentence;
        sentence[3] = 'i';
        c_ptr2[2] = *c_ptr1;

        printf( "letter holds: %c\n", letter );
        printf( "sentence holds: %s\n", sentence );
        printf( "c_ptr1 holds: %c\n", *c_ptr1 );
        printf( "c_ptr2 holds: %s\n", c_ptr2 );

        int number = 42;
        int lotto_picks[7] = { 8, 18, 28, 38, 48, 58, 68 };
        int *i_ptr1 = &number;
        int *i_ptr2 = lotto_picks;
        lotto_picks[3] = number;
        i_ptr1 = i_ptr2;
        i_ptr2[1] = *i_ptr1;

        printf( "number holds: %d\n", number );
        printf( "lotto_picks holds:");
        for( int pos=0; pos<7; pos++ )
                printf( "%d ", lotto_picks[pos] );
        printf( "\n" );

        printf( "i_ptr1 holds: %d\n", *i_ptr1 );
        printf( "i_ptr2 holds:");
        for( int pos=0; pos<7; pos++ )
                printf( "%d ", i_ptr2[pos] );
        printf( "\n" );

        return 0;
}
```

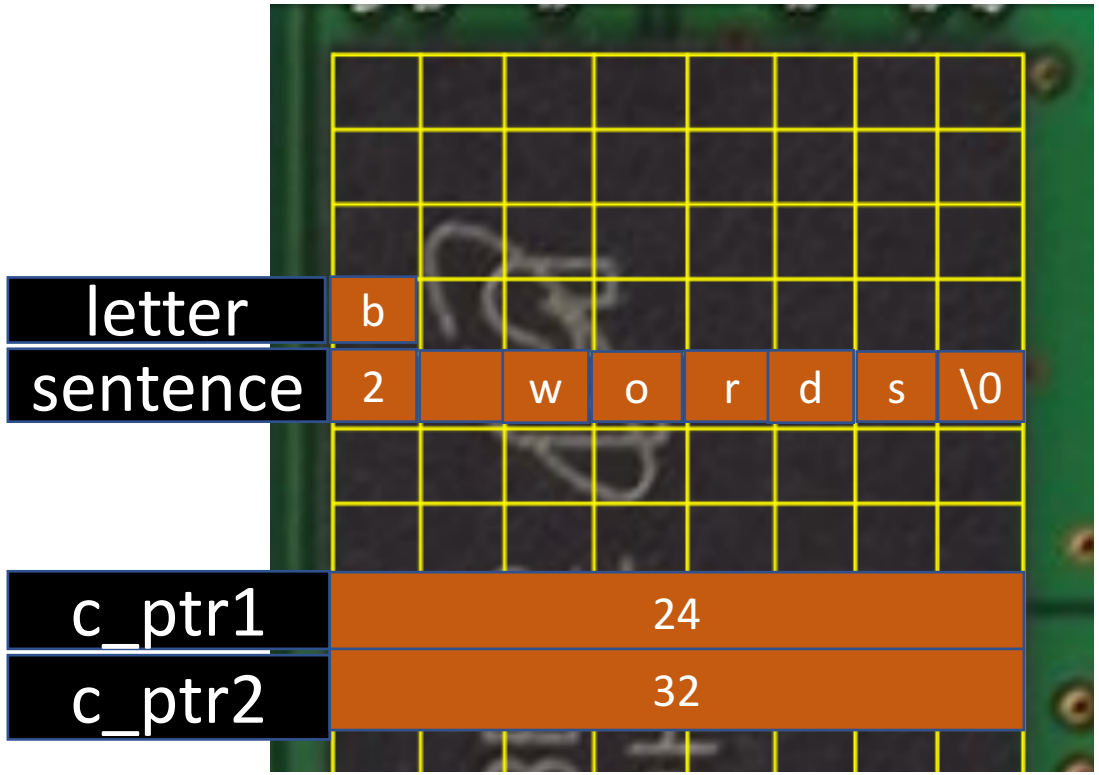
Box and Arrow

```
char letter = 'b';
char sentence[100] = "2 words.";
char *c_ptr1 = &letter;
char *c_ptr2 = sentence;
sentence[3] = 'i';
c_ptr2[2] = *c_ptr1;
```



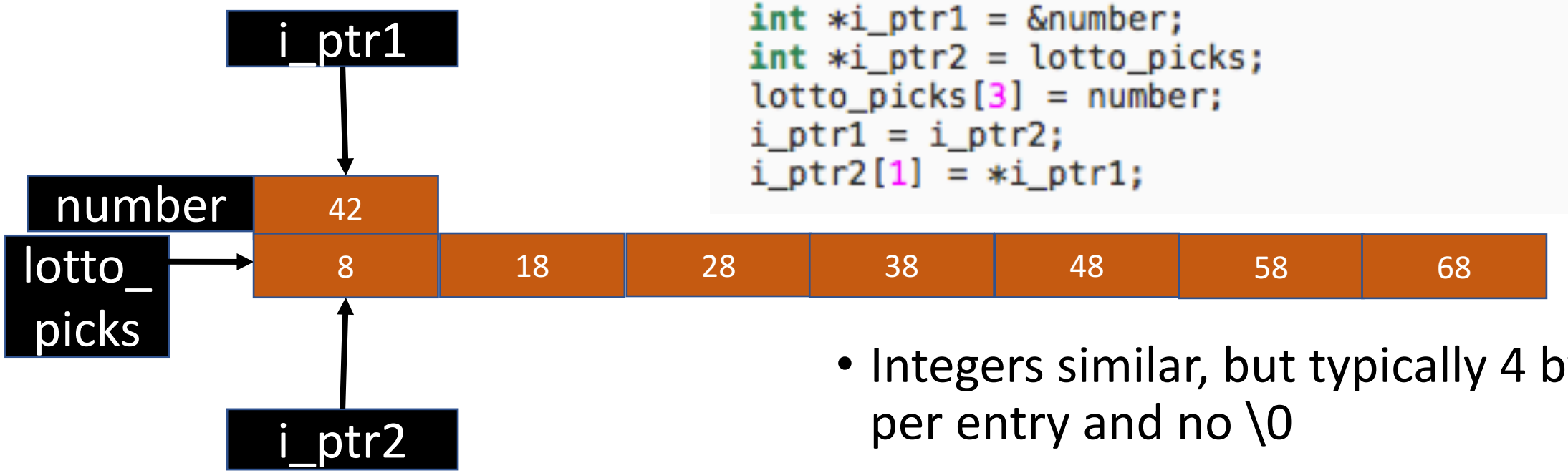
- & means “address of” or “points to” variable
- * means “dereference” or “get values at” pointer
- Array variables can be thought of as pointer always to the beginning
- Pointers can be indexed with [i] notation just like arrays

Memory and Addresses




```
char letter = 'b';
char sentence[100] = "2 words.";
char *c_ptr1 = &letter;
char *c_ptr2 = sentence;
sentence[3] = 'i';
c_ptr2[2] = *c_ptr1;
```

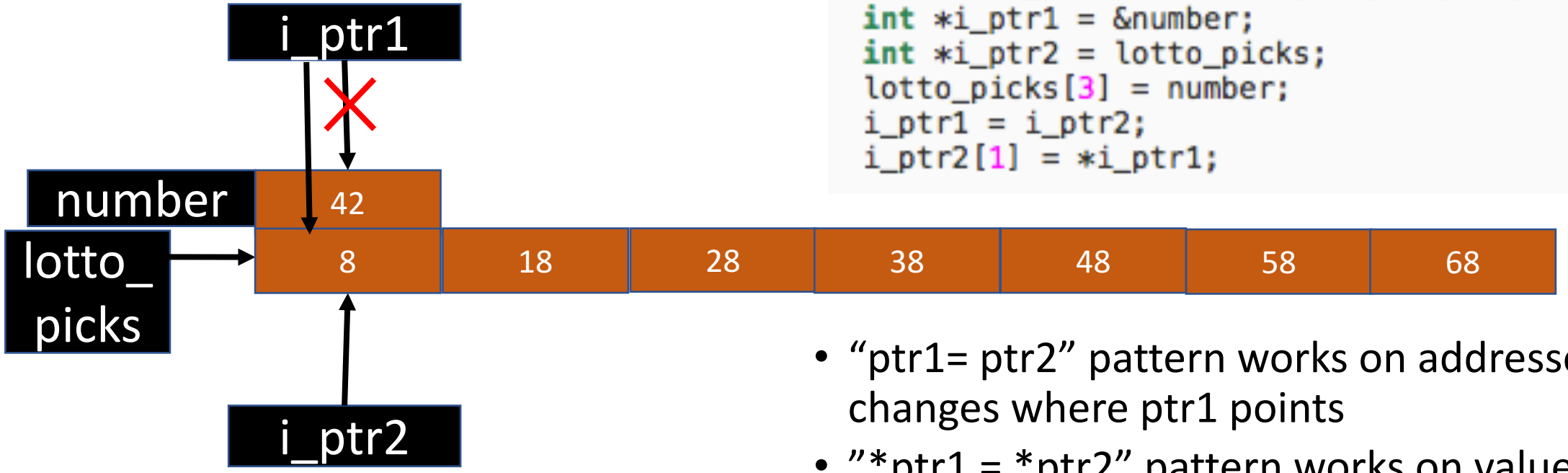
Box and Arrows: Integers




```
int number = 42;
int lotto_picks[7] = { 8, 18, 28, 38, 48, 58, 68 };
int *i_ptr1 = &number;
int *i_ptr2 = lotto_picks;
lotto_picks[3] = number;
i_ptr1 = i_ptr2;
i_ptr2[1] = *i_ptr1;
```

- Integers similar, but typically 4 bytes per entry and no \0
- *ptr is equivalent to ptr[0]
- More generally *(ptr+i) equiv ptr[i]

Box and Arrows



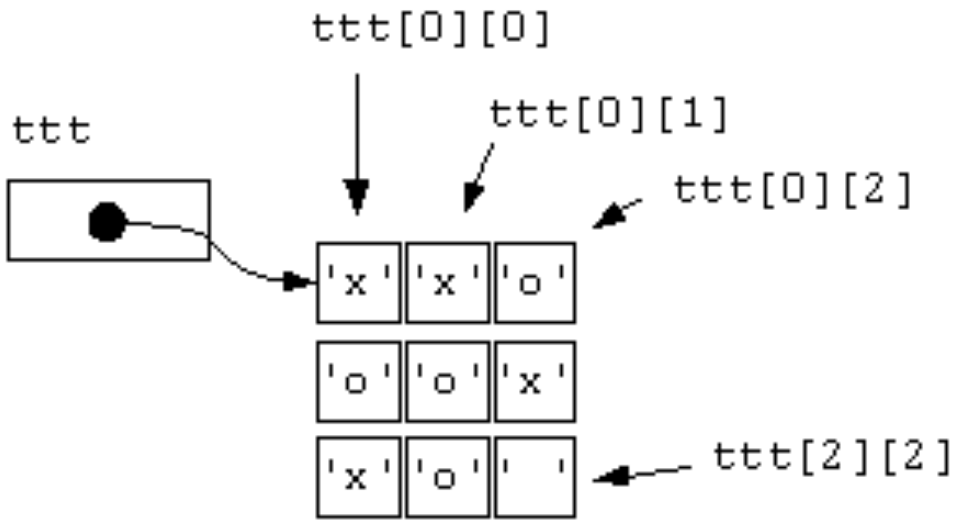

```
int number = 42;
int lotto_picks[7] = { 8, 18, 28, 38, 48, 58, 68 };
int *i_ptr1 = &number;
int *i_ptr2 = lotto_picks;
lotto_picks[3] = number;
i_ptr1 = i_ptr2;
i_ptr2[1] = *i_ptr1;
```

- “ptr1= ptr2” pattern works on addresses, changes where ptr1 points
- ”*ptr1 = *ptr2” pattern works on values, changes the value of memory at ptr1’s address

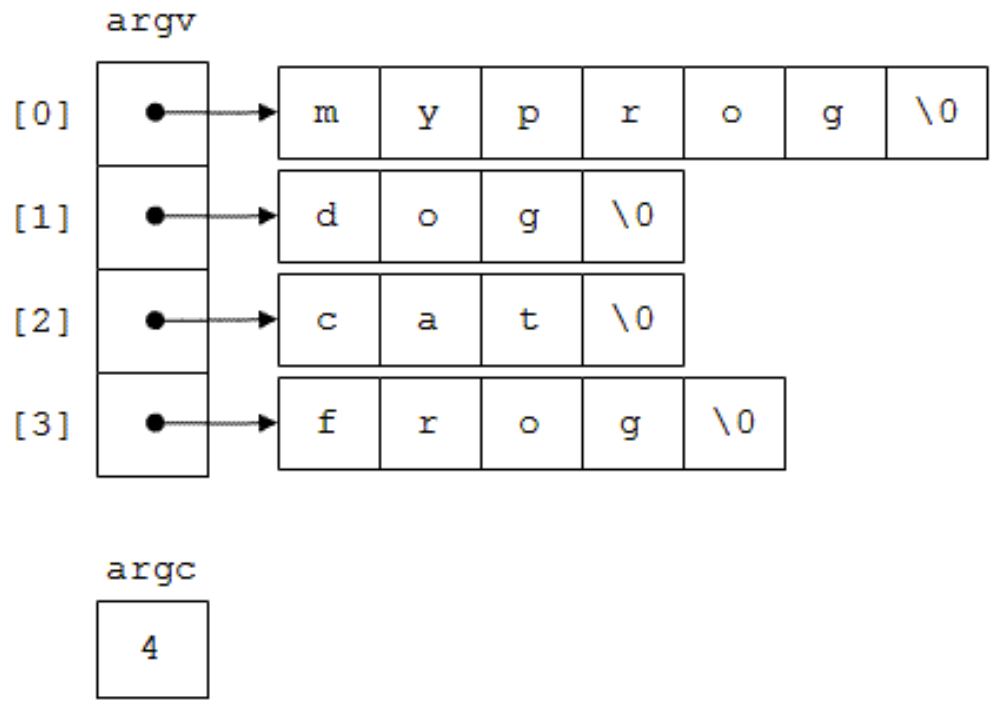
Where are we going today and next week?

- Real Software Systems that work with binary (non-text) data
- Specific example: how are images represented in:
  - Gaming
  - Computer Generated Imagery in movies
  - Satellite maps
  - Bunny-ear augmented selfies
- Today: data that has a 2D layout

2D Arrays and Arrays of Pointers

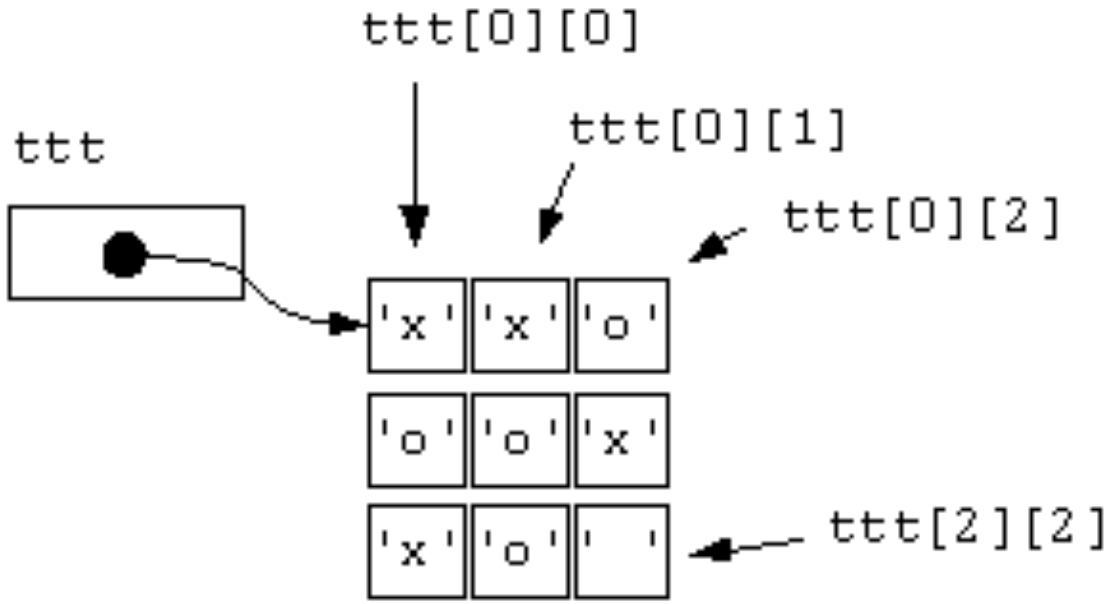



```
z123456@turing:~$ myprog dog cat frog
```



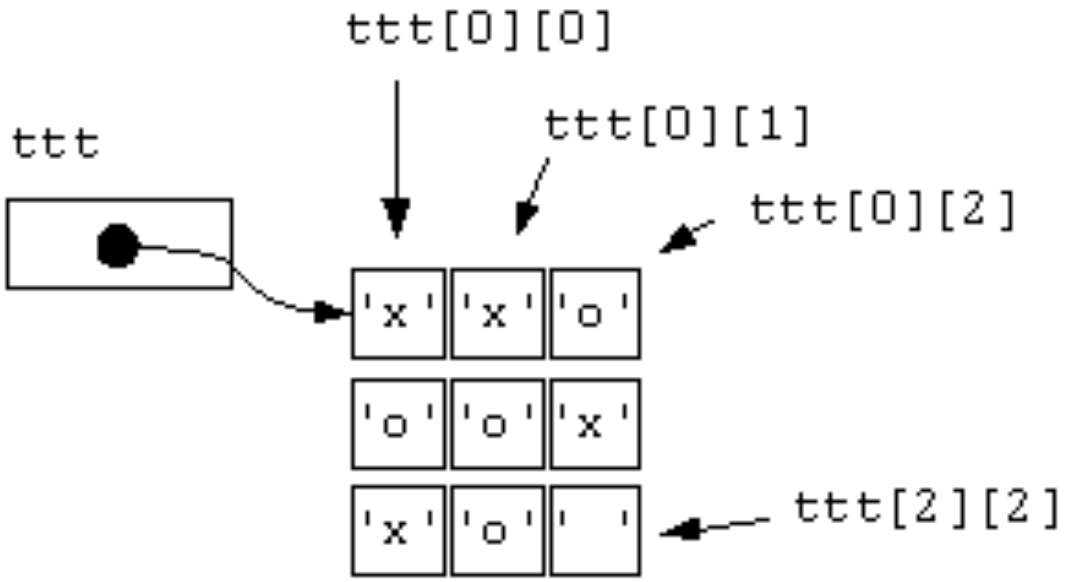
2D Arrays

- Declared by repeating the square bracket syntax:
  - char ttt[3][3];
- Creates an array where each entry is itself an array.
- The type (e.g. char) is the same for every entry of the inner array.
- Outer array behaves as we expect:
  - Fixed size
  - Always represents the memory of the first entry and cannot be moved (e.g., no ++)
  - Elements stored directly after one another



Accessing 2D Array Data

- For type array_name[N][M]
  - Syntax array_name[i][j] is used both to read and write data at entry i, j
- First index can be 0 to N-1, second index can be 0 to M-1
- In the image:
  - ttt[0][0] evaluates to ‘x’
  - ttt[2][2] = ‘o’; sets bottom-right value
- **Careful**: not [i,j]



When to use 2D Array?

- When each ”row” has the same size always:
  - int days_in_each_month[2][12] = {

  { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },

  { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 } };

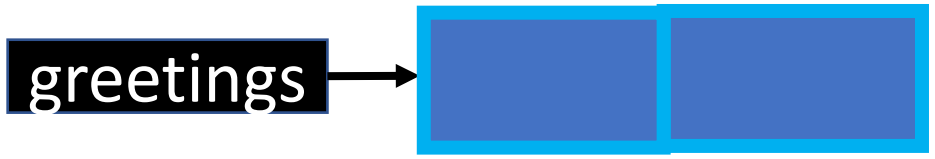
- Often when data looks like a “table”:
  - float grades[number_of_assignments][class_size] = {

  { 100.0, 99.0, 83.0 },

  { 99.9, 99.9, 99.8 } };

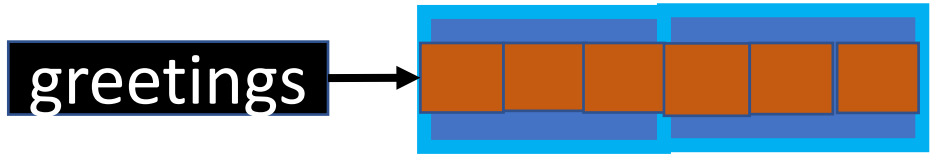
What does it look like in memory?

- char[2][3] greetings = { “hi”, ”yo” };
  - Is first off an array of length 2
  - We know those entries are placed directly in order



What does it look like in memory?

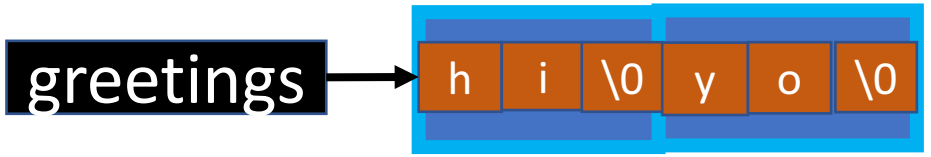
- char[2][3] greetings = { “hi”, ”yo” };
  - Is first off an array of length 2
  - Each of the outer entries contains an array of length 3, following the same rules



What does it look like in memory?

- char[2][3] greetings = { ”hi”, “yo” };

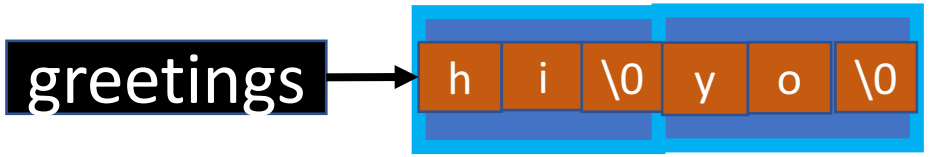
- The system memory is addressed in ascending linear order (1D). This view shows the layout in address space.



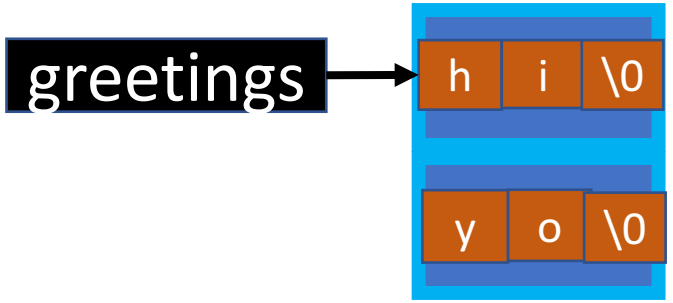
What does it look like in memory?

- char[2][3] greetings = { ”hi”, “yo” };

- The system memory is addressed in ascending linear order (1D). This view shows the layout in address space.



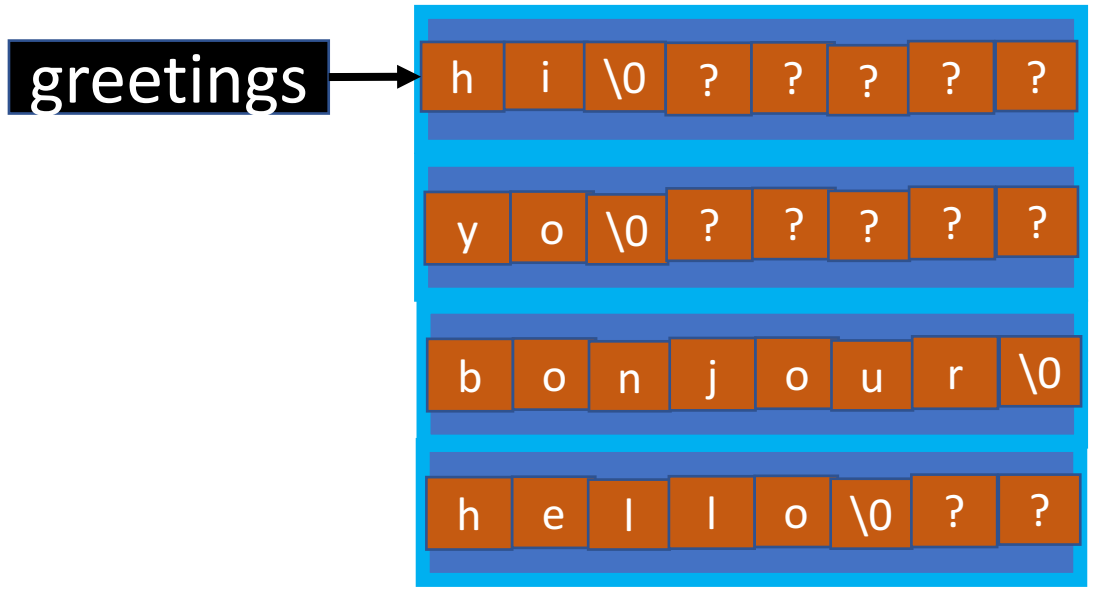
- NOTE: it’s often nice to draw it this way to understand working with images etc, but it is only for convenience.
  - Does not indicate bigger jumps in memory between ‘\0’ and ‘y’, compared to ‘y’ and ‘o’



When not to use 2D Arrays?

- If the data in each entry “row” can vary in length
- If the length of each entry “row” isn’t known when coding, for example it will change based on user input
- Handling these cases with 2D arrays forces us to use enough space for the longest possible row… for every row!

char greetings[4][8] = { “hi”, “yo”, “bonjour”, “hello” };



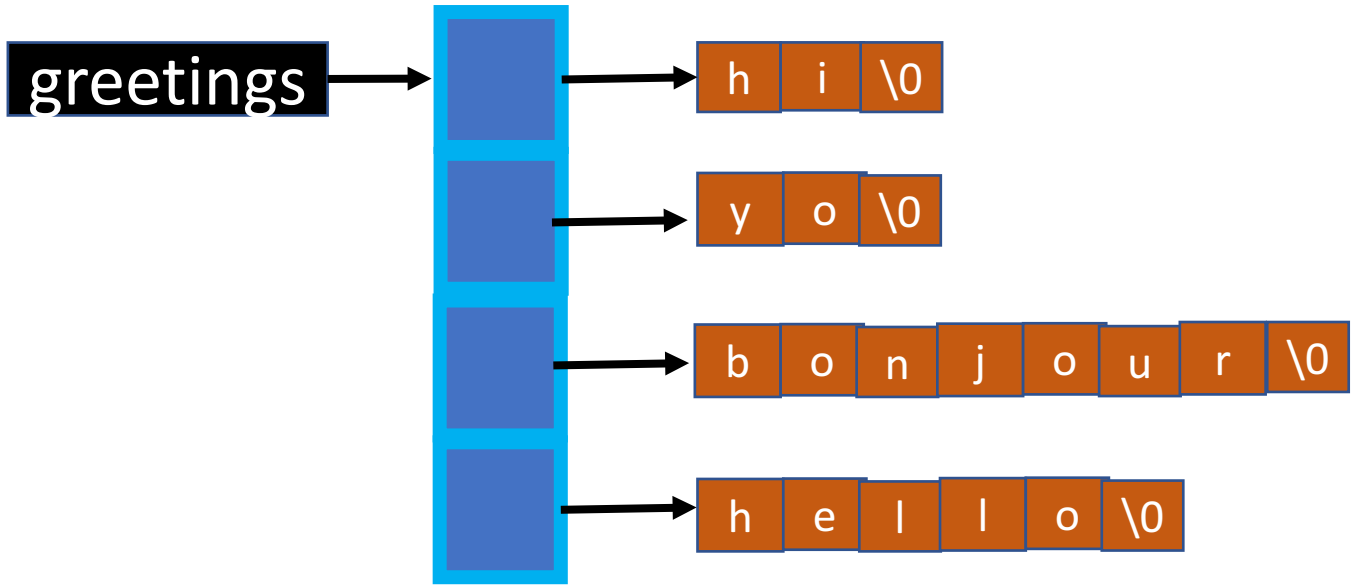
Another solution: Array of pointers

- char *greetings[4] =

{ “hi”, “yo”, “bonjour”, “hello” };

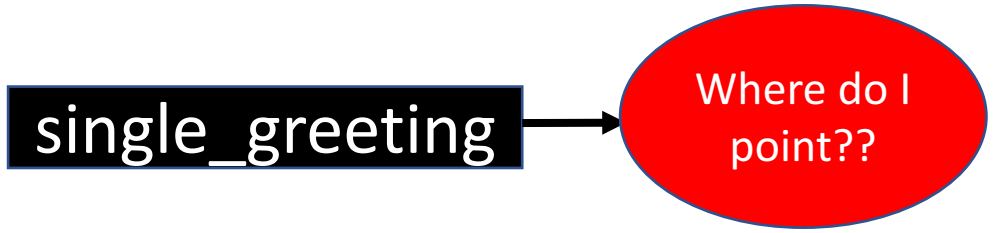
- The type of each entry is now ”char*”, a single character pointer

- This avoids wasting the space



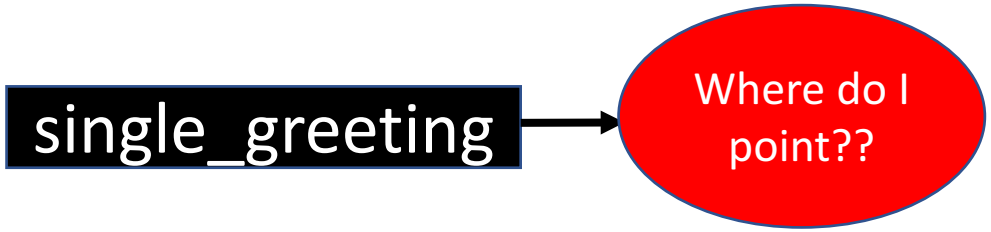
Array of pointers gotcha!

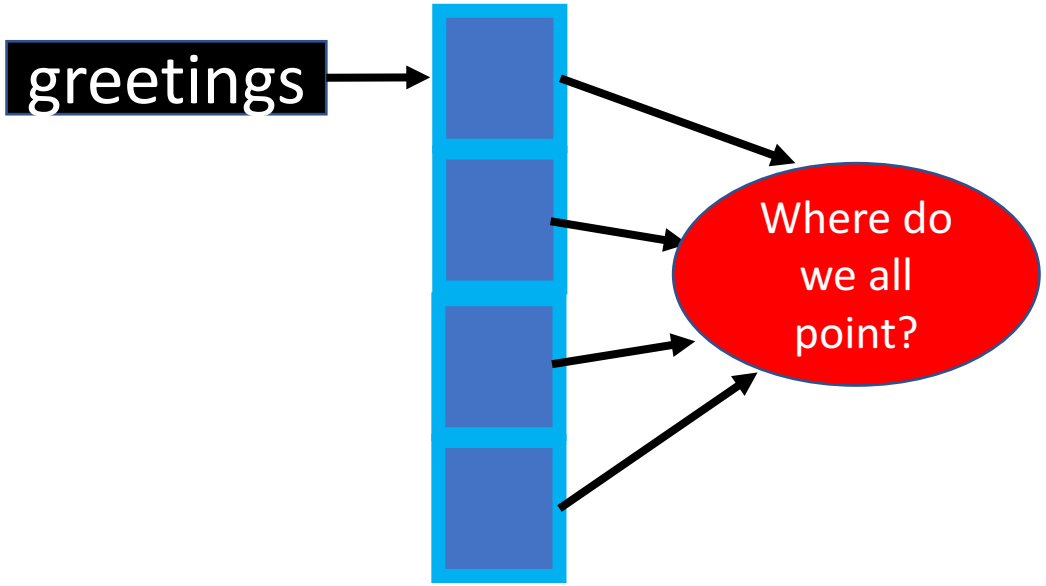
- Recall, where does an uninitialized pointer point?
  - E.g., char* single_greeting;
  - If we are lucky it might be NULL leading to segfault on use
  - We should be smart and set it to NULL ourselves
  - Otherwise, we will learn how to debug the hard way…



Array of pointers gotcha!

- The same story holds for arrays of pointers:
  - E.g., char *greetings[4];
  - All of the pointers are now undefined
- The declaration on it's own does not create any memory to hold words
- Must ensure to assign each pointer before its use:
  - To a literal like "hello"
  - To an existing pointer or array





What about calling functions: with 2D array

- Following our example, char ttt[3][3]
- Suppose we'd like a function that we can call like f(ttt)

1. void f(char current_board[3][3]){ /* function code */ }
    - Match the types exactly: it must work!

What about calling functions: with 2D array

- Following our example, char ttt[3][3]
- Suppose we'd like a function that we can call like f(ttt)

1. void f(char current_board[3][3]){ /* function code */ }
2. void f(char current_board[][3]){ /* function code */ }
   - C needs to know the entry type to read the data correctly, so the char array of length 3 must be present
   - C does not need the length of the outer array: remember, it doesn't take care of this for us anyhow!

What about calling functions: with 2D array

- Following our example, char ttt[3][3]
- Suppose we'd like a function that we can call like f(ttt)

1. void f(char current_board[3][3]){ /* function code */ }
2. void f(char current_board[][3]){ /* function code */ }
3. void f((char*)current_board[3]){ /* function code */ }
   - This says "pointer to data of type 3-length char array"
   - Same reasoning, as we can note the outer array with unknown size is equivalent to pointer
   - The brackets around (char*) are essential to distinguish this from an array of pointers (It is a good sanity check for you to be really sure you now know the difference yourself!)

Calling functions with arrays of pointers

- Suppose we have our char* greetings[4];
- And would like to be able to call f(greetings);

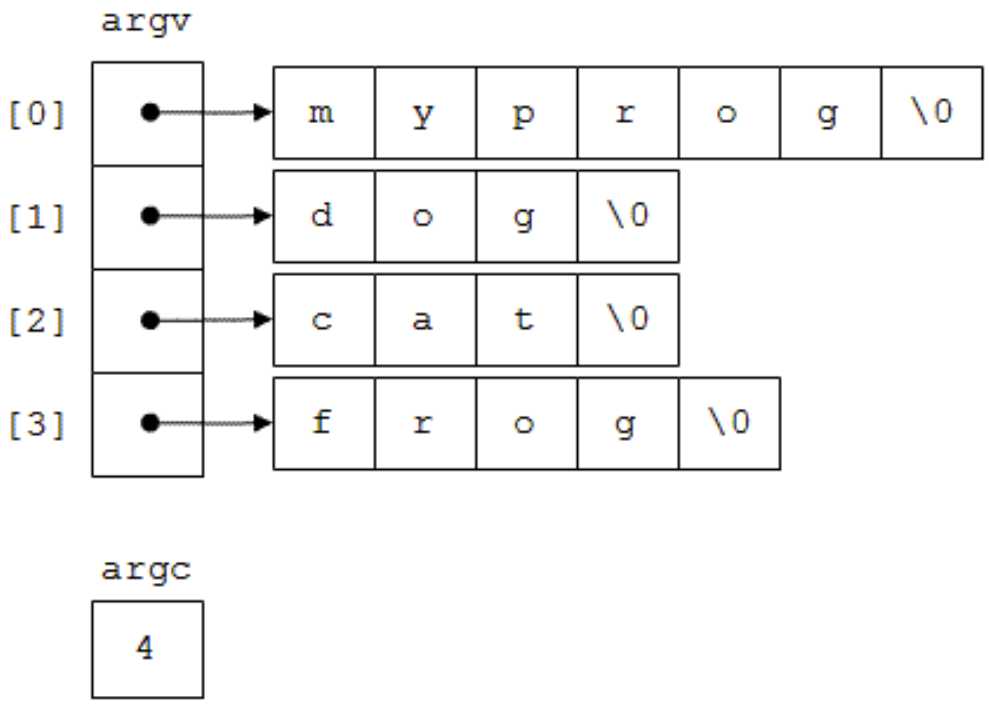
1. void f(char *greetings[4]){ /* function code */ }
2. void f(char *greetings[]) { /* function code */ }

- Same reasoning as the above, we only have 1 array to track now and C can do that without the size

Final example for today, sort argv

- Type is char* argv[], an array of pointers, each to one of the argument words

```
z123456@turing:~$ myprog dog cat frog
```



- We want the user to be able to type any number of arguments, each of any length and have the result end up properly sorted

- We can sort “in place” by working only on the pointer values within argv, no need to create a new variable

Dave’s solution on Github ExampleCode

- Lecture11-2Darrays/sort_argv.c
- I recommend you try this yourself as an exercise and compare
- At minimum, go over and understand each component

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

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

	// Consider one argument at a time
	for( int pos=0; pos<argc; pos++ ){

		// Find the "first" word alphabetically from here on
		char *current_min = argv[pos];
		int min_pos = -1;
		for( int pos2=pos+1; pos2<argc; pos2++ ){
			// Recall, str1 < str2 not OK, need strcmp
			if( strcmp( argv[pos2], current_min ) < 0 ){
				current_min = argv[pos2];
				min_pos = pos2;
			}
		}

		// If the initial min changed, we found a word out of order, swap
		if( current_min != argv[pos] ){
			char *temp = argv[pos];
			argv[pos] = current_min;
			argv[min_pos] = temp;
		}
	}

	// Print the result, should be sorted!
	for( int pos=0; pos<argc; pos++ )
		printf( "%s\n", argv[pos] );

	return 0;
}
```