Pointers on Pointers: They're A-MAZE-ing!

A CS 33 Helpsession

What is a pointer?

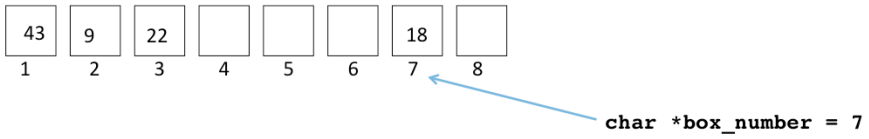
- A typed variable that holds an address.
- A pointer is a *reference*.
- To get data, you must *dereference* a pointer.

What is a pointer? A pointer is a variable that holds the address of some typed data. A pointer is a reference to the data. You must dereference the pointer to get the data itself.

Practical Example

- Imagine a group of boxes on floor.
- Each box is labeled with a number.
- Inside **box 7** is an integer I'm interested in.
- I want to tell my program to open **box 7**, and add 1 to the integer.

```
char *box_number = 7; // declare pointer to point to box 7
*box_number = (*box_number) + 1; // add one to box contents
```



Imagine a group of boxes on the floor. They each have an integer in them. To get my program to modify a particular integer in a box, I must tell it in which box to get the integer itself. For example, suppose my program is interested in box 7. To get my program to access box 7, I first give it a pointer to box 7. The program can then dereference the pointer by going to the box labeled "7." The program can open it and modify the integer inside of it.

How Is That Computer Science?

- The box number represents the **memory address** of the integer in the computer.
- The **memory address** tells the computer where a value physically exists.
- In a computer, the box number would actually look like `0xc0b86704`
- Computers are byte addressable, so a box holds one byte of data.
- Larger pieces of data are spread out over many boxes.
- This is why we needed pointers to be typed! We must know how many boxes to look in to fetch all the data we want.
- `sizeof(int) = 4`, so an integer spans 4 boxes.
- NOTE: All pointers are the same size (they are the size of a word in the machine). The type of pointer dictates the size of the data being pointed to.

Let's look at some code!

- A pointer variable is declared with an `*`.
- A pointer is dereferenced with an `*`.
- A pointer is gotten with an `&`. We also call this "getting a reference."

```
int example = 9001; // just an int
int *int_pointer = &example; // a pointer to an int (declaring a pointer)
int my_int = *int_pointer; // the int being pointed to (dereferencing a pointer)
int *another_pointer = &my_int; // the pointer to my_int (getting a reference to my_int)
```

Why do we even need pointers?

- In Java, you might write

```
static void makeACatWalk(Cat myCat) {
   myCat.walk();
}

Cat greg = new Cat();
makeACatWalk(greg);
```

In Java, you might write a function that takes a Cat and calls its walk method. You would expect the cat you passed in to have walked.

Why do we even need pointers?

- You are actually writing

```
static void makeACatWalk(Cat *myCat) {
    (*myCat).walk(); // equivalent to writing, myCat->walk();
}

Cat *greg =  (Cat*) new Cat();
makeACatWalk(greg);
```

But in order for this to happen, everything must be passed by reference with pointers! If it wasn't, the Cat object would have to be copied every time it was passed into a function. and the original Cat being passed into the function would not be modified. We call the process of modifying data from outside a function inside a function, "mutation." By passing in greg by reference, we allow makeACatWalk() to mutate greg. If greg had been copied into makeACatWalk, the original greg would remain un-mutated.

Arrays

- Array variables are pointers to the first element in the array.
- Arrays don't have a "length" field
  - Use `sizeof` (be careful!)
  - Pass around length of array
- The `[ ]` operator does two things
  - Adds an offset to the pointer
  - Dereferences the pointer

Array variables are merely pointers which point to the first element in an array of data.

They do not remember their own size. To figure out the size of an array you have two choices: (1) you may use the sizeof function or (2) you may pass around the array length with the array itself. Though using sizeof saves you some trouble, it tends to have some caveats that you must be careful to avoid. If you are interested in this function, I recommend looking up the man page for sizeof. You are safer for this introductory project just passing the size of the array around.

The [] operator supplies convenient access to array elements. It first adds the proper byte offset to the pointer representing your array, such that you now have a pointer to the element specified. It then dereferences this pointer an returns the element itself.

Arrays

```
int my_array[10]; // declares an array of size 10
int *my_ptr = my_array; // my_array is of type int*

//the following are equivalent
int first_element = my_array[0]; // adds (0 * sizeof(int)) to my_array, dereferences
int first_element = *my_array; // my_array points to the first element!

// the following are equivalent
int fifth_element = my_array[4]; // adds (4 * sizeof(int)) to my_array, dereferences
int fifth_element = my_ptr[4]; // adds (4 * sizeof(int)) to my_ptr, dereferences
int fifth_element = *(my_array + 4); // adds (4 * sizeof(int)) to my_array, dereferences
```

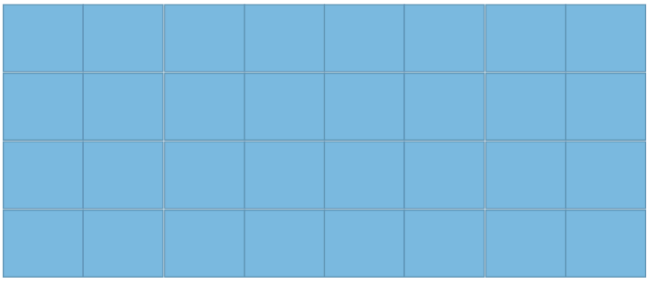
Here are some example of how to use arrays in code. Note how many different ways there are to access the same data.

2D Arrays

- A 2D array is an array of arrays.
- An array name is essentially a pointer to its first element.
- A 2D array is a pointer to the first array.

```
int two_d_array[4][10]; // declare 2D int array of size 10 x 10
int **two_dimensions = two_d_array; // declare variable as pointer to 2D int array
int *first_row = two_d_array[0]; // get first row
int first_element = two_d_array[0][0]; // get first element
int first_element = **two_d_array; // get first element
```

2D Array

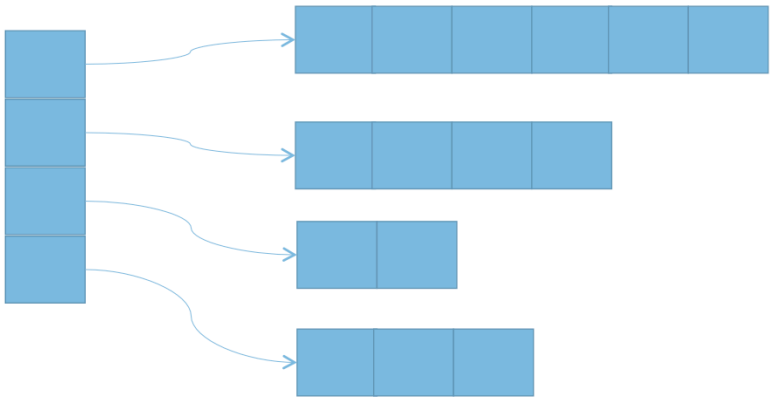


You've seen char ** before…

```
int main(int argc, char **argv) {
  // ...
}
```

The main function of a C program uses a 2D array to hold the program arguments. A char * is equivalent to a string (an array of characters), so a char ** is an array of strings or a 2D array of characters.

Array of pointers



How does this relate to Maze?

- You can use a 2D array of room `structs` to represent a maze.
- Do not pass room `structs` by value, aka use a pointer!
  - Expensive and silly to copy
  - Cannot mutate

A maze can be represented as a 2D array of room structs. When using the 2D array, be careful to write proper function signatures. Passing by value (aka not with a pointer) means every time you call a function, the entire piece of data passed in is copied. This is okay for small pieces of data like ints and chars, but not great for structs which tend to be larger. Instead, you may pass structs in by pointer. Another reason to pass by pointer is to allow mutation, such that if you modify your maze in a function, the results of the modification persist after the function returns.

How does this relate to Maze?

```
typedef struct room room_t;

room_t maze[25][10]; // declare your maze

room_t *get_first_room(room_t the_maze[25][10]) {
       return &the_maze[0][0];
}

room_t *get_first_room(room_t **the_maze) {
       return &the_maze[0][0];
}
```

Hooray for pointers!

- Questions?
- More on maze