Pointers & Memory In C

March 11, 2025

1 Terms

- **Address**: Refers to a **location** in memory. These are **unique** byte sequences. We typically represent addresses in hexadecimal (denoted by 0x prefix).
  For example, on a 32-bit system, let's say my first address is 0x00000000.
  Each digit here is 4 bits (0 = 0000), and there are 8 digits (4x8=32 bits).
  The next address would be 0x00000001, then 0x00000002, and so on.
  Each address stores a **single byte** of memory. You an think of your computer's memory (stack/heap) as an array. The address is like the index to this array.

- **Pointer**: Pointers are used to **directly manipulate memory**. Pointers store the following key information (along with others):
  - **Address (Value)**: where in memory does this pointer point to? ex. address 0x00000003.
  - **Type**: what kind of data does this pointer actually point to? (int, char, ...). This is important, as we need to know how many bytes to look beyond our address. If we're dealing with a char (1 byte), we get its value just by looking at our pointer address. If we're dealing with an int (4 bytes), we need to look at 4 bytes starting from our address.

- **Reference**: Referencing means returning the address of a variable. We use the **&** operator. Let `x` be a variable in my code. Then `&x` tells me the address in memory where the **first** byte of `x` is stored.

- **De-Reference**: The opposite operation. Given an address, go to that address and return the stored value (by looking at the necessary bytes starting from that address, depending on type). To de-reference, we use the ***** operator.

2 Examples

2.1 What You've Seen Before

```c
int x = 2;
```

What the compiler/system understands at runtime:

- **int**: The data type is integer, so we need to allocate 4 bytes (on most systems), which equals 32 bits. Find an unused memory address and store 32 bits of information for this variable.
- **x**: The variable name. The program does not "see" your variable name, instead it uses a symbol table which maps variable names to their addresses.
- **2**: The 32 bits starting from the **address** in memory will be set to the binary value for 2, which is 0000 0000 0000 0000 0000 0000 0000 00**10**

Here is a table to illustrate. Imagine that the compiler found 4 free bytes of memory starting from address 0x12300001. The binary data for the integer are now placed into these memory addresses 0x12300001 to 0x12300004.

Address (Hex)	**Value (Binary)**
0x12300001	0000 00**10**
0x12300002	0000 0000
0x12300003	0000 0000
0x12300004	0000 0000

You may be wondering, why are the bytes placed in reverse order? This is because the order of byte storage depends on your system architecture. Many systems use **little-endian**, where the least significant byte (LSB) is stored in the **first**/**smallest** address. You'll probably see this come up if you take any Computer Architecture or Operating Systems courses, but it's not too important for this course.

2.2 Int Pointer

```c
int x = 3;
int *pX = &x; // pointer to x by referencing (&)
int y = *pX;  // value of x by de-referencing (*)

// %p is the identifier for printing
printf("Address of x: %p\n", pX);
```

- **int**: Dealing with integer data (4 bytes)
- ***pX**: The * symbol between a type and variable, means **pointer to** an **address** where an integer is stored (4 bytes starting from this address).

- **&x**: The & symbol means **address of ....** This will return the address that the compiler assigned to x. This is called **referencing**.
- **int y = *pX**: Create a new integer called `y`. Assign its value by **de-referencing** pX. Basically, retrieve the value that pX points to (by going to its address). The value of `y` now equals the value of `x`, directly retrieved from memory.

2.3 Arrays

```
int arr[] = {1, 2, 3, 4, 5};
```

Assume that our system stores 4 bytes of data per integer. Also, the system has chosen a free block of memory starting from address 0x12300000 for **&arr** above. Check your understanding by filling in the table below:

Array Index Reference	**Address (Hex)**
&arr[0]	0x123000**00**
&arr[1]	0x123000_ _
&arr[2]	0x123000_ _
&arr[3]	0x123000_ _
&arr[4]	0x123000_ _

Answer: Memory addresses must increment by 4, since there are 4 bytes of data used for each index.

Array Index Reference	**Address (Hex)**
&arr[0]	0x123000**00**
&arr[1]	0x123000**04**
&arr[2]	0x123000**08**
&arr[3]	0x123000**0c**
&arr[4]	0x123000**10**

Also, arrays can decay into pointers to their first element. Here is an example which shows the initialization of arrays and passing them into functions using explicit pointers.

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

// pointer as argument
void printInts(int *arr, int size) {
    for (int i=0; i<size; i++) {
        printf("%d ",arr[i]);
    }
    printf("\n");
}

// pointer as argument
void printChars(char *arr) {
    // arr can be directly treated as string (character array)
    int len = strlen(arr);
    for (int i=0; i<len; i++) {
        printf("%c ",arr[i]);
    }
    printf("\n");
}

int main() {
    int arr1[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr1)/sizeof(int);
    // arr2 is now a pointer to the same address as arr1
    int *arr2 = arr1;
    // arrays decay to pointers when passed into function
    // this is why we can pass (arr2) and don't need (&arr2)
    printInts(arr2, size);

    char arr3[] = "Hello";
    // arr4 is now a pointer to the same address as arr3
    char *arr4 = arr3;
    // again, pass arr4 (decays to pointer) directly into function
    printChars(arr4);

    return 0;
}
```

The main takeaways are that arrays can decay into pointers automatically. So, we do not need **&arr** when passing into a function, and we do not ***arr** inside each function to de-reference values.

3 What's the Point? :)

3.1 Pass by Reference

Pointers are essential for code optimization. Let's walk through an example. Imagine we want to create a function that should update a variable by doubling it. One option would be to **pass by value**, which means we pass a copy of our original variable into this function, compute the doubled result, and finally update our original variable. So, it's a 3 step process like below:

```
#include <stdio.h>

// takes in an integer value (not pointer)
int update(int x) {
    return x*2;
}

void main() {
    int x = 5;
    printf("%d\n",x);
    // pass by value, then update based on result
    x = update(x);
    printf("%d\n",x);
}
```

We did not use any explicit pointers here. Note: you may be thinking, why not just double the value inside main()? Well, yes for a simple operation like this, we could do it in main. But what if our update() function was more complex. In general for cleaner code, we want to keep each function (including main()) minimal. Pointers can help us with this.

Look at the optimized example below. Now, we **pass by reference**. This means the update function takes in a pointer to an integer, and directly doubles it in memory through de-referencing. In main(), where we call the function, we pass in a **reference** to x using **&x** (**address** of x), like we have seen before:

```c
#include <stdio.h>

// takes in integer pointer as argument
void update(int *pX) {
    // de-reference the pointer
    // directly double the value in memory
    *pX *= 2;
}

void main() {
    int x = 5;
    printf("%d\n",x);
    // pass reference to x
    update(&x);
    printf("%d\n",x);
}
```

Now, x has been updated directly without making any copies, which we can confirm by printing in main(). This is a very important concept, and you'll see pass by reference used frequently in C programs online.

3.2 Dynamic Allocation

Another way that pointers are used is for **dynamic memory allocation**. When you create an array in C, a block of memory is allocated starting from address **&arr**, depending on the array type and size. After this, the size is fixed, i.e. **static**. The dimensions of the array must be known at compilation (hard-coded int or #define/macro).
But what if we don't know how big our array should be? Maybe the size depends on something that is not known during compilation, for example user input or output from previous functions. This requires **dynamic** memory allocation, i.e. the amount of memory required can change dynamically.
In C, there are several functions within **stdlib.h** for handling dynamic allocation:

```c
#include <stdio.h>
// need stdlib.h for memory allocation functions
#include <stdlib.h>

void main() {
    int n;
    printf("Enter the size: ");
    scanf("%d", &n);

    // need to know how many bytes of memory to allocate
    // use sizeof(int) and multiply by number of array elements (n)
    int nBytes = n * sizeof(int);

    // malloc allocates nBytes of memory and returns a void pointer
       to the starting address
    // you should explicitly cast the returned pointer to the correct
        type (int*)
    // it would still compile without (int*) due to implicit casting
    int *arr = (int*) malloc(nBytes);

    // An alternative using calloc
    // (worse performance but guarantees bits are set to 0)
    // Note it takes in the number of elements and size of data in 2
       arguments
    int *arrC = (int*) calloc(n, sizeof(int));

    // Important, check for memory allocation failure right away
    if (arr == NULL || arrC == NULL) {
        printf("Memory allocation failed\n");
        return;
    }

    // Simple assignment and print
    for (int i = 0; i < n; i++) {
        arr[i] = i;
        printf("Address: %p, Value: %d\n", arr + i, arr[i]);
        // Note arr + i is equivalent to &arr[i] due to pointer
           arithmetic
        // Basically, the compiler scales i by sizeof(data): For our
           int arr, arr+i becomes arr+0, arr+4, arr+8, ...
    }
    printf("\n");

    // Resize example
    int n2;
    printf("Enter the new size: ");
    scanf("%d", &n2);
    // Resize the array using realloc()
    // Pass the pointer to first address of arr and new size
    // What is it doing?
        // 1) If there's space in the same continuous block, it
           preserves old memory and allocates the extra (n2 - n)
           bytes
        // 2) If there's no space, it will copy the existing n bytes
           and allocate the total n2 bytes somewhere else (fresh
           address)
    int *arr2 = realloc(arr, n2*sizeof(int));

    // Check for errors
    if (arr2 == NULL) {
        printf("Memory re-allocation failed\n");
        return;
    }
    // If no error, let's update our original array to new one
    arr = arr2;

    // Print values in resized array
    for (int i = 0; i < n2; i++) {
        printf("Address: %p, Value: %d\n", arr + i, arr[i]);
    }

    // Free any allocated (heap) memory at the end
    // Otherwise, prone to memory leaks
    free(arr);
    free(arrC);
}
```

Output

```
Enter the size: 4
Address: 0x564cb9ef4ac0, Value: 0
Address: 0x564cb9ef4ac4, Value: 1
Address: 0x564cb9ef4ac8, Value: 2
Address: 0x564cb9ef4acc, Value: 3

Enter the new size: 8
Address: 0x564cb9ef4b00, Value: 0
Address: 0x564cb9ef4b04, Value: 1
Address: 0x564cb9ef4b08, Value: 2
Address: 0x564cb9ef4b0c, Value: 3
Address: 0x564cb9ef4b10, Value: 0
Address: 0x564cb9ef4b14, Value: 0
Address: 0x564cb9ef4b18, Value: 0
Address: 0x564cb9ef4b1c, Value: 0
```

4 Practice

Memory allocation and pointers will be useful for completing Assignment 3. You should go through the template code provided to you. The added complexity is that in 2-dimensions, we are now dealing with pointers to pointers (**)! But it's not too bad: Make sure you understand how pointers and the C syntax works in 1-D. int* represents a pointer to a bunch of integers (1-D array). Then you can think of (int*)* as a pointer to a 1-D array. Basically, everything before the final * in your declaration is the type of data you're storing. Then we can access elements using the usual array notation, arr2D[i][j] after allocation.

```c
// Allocate rows of my array
// Using sizeof(int*) since we have pointers to integers stored here
int** arr2D = (int**) malloc(rows * sizeof(int*));

// Check for allocation success (important)
...

// Allocate a row of my array
// Here, we are using the usual 1-D syntax to fill a row
// Using sizeof(int) since we have integers stored here
arr2D[0] = (int*) malloc(cols * sizeof(int));
...

// Check for allocation success again!
...

// Do something with your array
...

// When finished, think about what you need to free. Is freeing each
   row sufficient?
free(...)
...
```

If you want more practice, try re-doing some past exercise or quiz questions using functions which take in pointers rather than values. Continue testing in main() like usual. Good luck.