Pointers

Memory as sections of a tape

We can imagine computer memory as a long string of sections of tape. Each section has
a label 0,1,..,n. Each section may also
contain some data such as the word "hello"
or the number 5.

Label (Address)	Data
0	5
1	66
2	"hello"
3	"world"
.	…
.	…
.	…
n	8

If we wanted to ask what was at location 2 we could go check memory and see that it is the world hello.

The idea of a pointer is simply a variable that
holds one of these labels or addresses of a given location in our table.

Pointers in C

In a modern computer the tape that represents memory is very long. Theoretically a 64bit CPU can address 2^64 memory locations. This is why in C a pointer, any pointer regardless of type, is represented by a 64bit integer.

You can think of each memory location as being a single byte (8 bits) in size, this by convention. Its the smallest amount of information we'll ever care to work with.

Memory addresses are most commonly written using hexadecimal as it is more compact and readable (if your used to it) then either binary or decimal.

These are all equivalent numbers

- Binary: 0b110110100100001011010010101000001101101001000010110100101010000
- Decimal: 94804327453344
- Hex: 0x5a42d2a0

We can make an integer pointer in C and look at its value like so

```
int *i;

printf("This pointer currently points to address 0x%x in memory\n", i);
```

Output

```
"This pointer currently points to address 0xfffffce in memory"
```

Pointer Semantics

What is special about a pointer in C is the semantics that the C language attaches to it. This is related to the type system.

Each type in C has a set size in bytes, which we can look at and print using the `sizeof(type)` function. This function given a type returns
the size of said type in bytes as a `long unsigned int = size_t`. Its important to note these sizes may vary from physical system to physical system,
but these are the most common values.

```
printf("Size of integer = %lu\n", sizeof(int));                             //prints 4 (4 bytes = 32 bits)
printf("Size of long integer = %lu\n", sizeof(long int));                   //prints 8
printf("Size of unsigned long integer = %lu\n", sizeof(unsigned long int)); //prints 8
printf("Size of size_t = %lu\n", sizeof(size_t));                           //prints 8
printf("Size of char = %lu\n", sizeof(char));                               //prints 1
```

Since a ptr (any pointer, of any type) is just a 64 bit integer the size of a ptr is 8 bytes no matter the type it points to. Remember it just specifies a memory location.

```
//these will all print 8
printf("Size of int* = %lu\n", sizeof(int*));
printf("Size of char* = %lu\n", sizeof(char*));
printf("Size of char** = %lu\n", sizeof(char**)); //what structure is described by char** ?
```

You can effectively think of a pointer of a given `type` as the start of a sequence of `sizeof(type)` bytes. By inspecting these bytes we can get a value of `type`.

For example if we have an `int*` and we look at the 4 bytes starting at the address (label) specified by our ptr then we will say those 4 bytes represent the value of an integer.

Operations on Pointers

Dereference (*)

When we dereference a pointer we are now referencing the value stored at that location or label or address. When referencing that value we can do anything that we could do with a variable of that type.

```
int* i = 0xff00; //so from 0xff00 to 0xff04 we have an integer

//Dereference and assign
*i = 100; //in the 4 bytes starting at 0xff00 store the binary value of 100

printf("%d\n", *i+4); //prints 104
```

Addition (+)

Recall that a pointer is an integer. However C defines addition on pointers differently then addition on integers.

Consider the following

```
int* i=0;
int j=0;

printf("Value of Int ptr: 0x%x\n",i);
printf("Value of Int: 0x%x\n",j);

i += 2; //increment ptr
j += 2*sizeof(int); //increment integer

printf("Value of Int ptr: 0x%x\n",i);
printf("Value of Int: 0x%x\n",j);
```

Output

```
Value of Int ptr: 0
Value of Int: 0
Value of Int ptr: 8
Value of Int: 8
```

This behavior implies that pointer addition is defined as adding the size of its type times the input given.

```
<type>* pointer;

pointer + x <-> pointer + x * sizeof(<type>)
```

This behavior, and the fact that pointer addition is defined at all makes sense when we consider arrays in C.

Address of (&)

One other operator that is important to consider when working with pointers is the address of operator (&).

This operator is defined on any data type in C and pointers as well. Its main use is to obtain a pointer to object that you call it on. This is useful when you want to use a function to modify something without having to return a new value.

```
int i = 5;

int* j = &i; //so j is now a ptr to i

printf("%d\n", *j); //dereference j to print i
```

Output

```
5
```

Its interesting to think about what getting the address of pointer means. Consider the above code and the pointer `j`.
What is `&j`. Well ultimately `j` is just a number. It is also stored somewhere in memory. So address of `j` is exactly that. It is the location where the value of the pointer itself (not the value at the location specified by the pointer) is kept.

Arrays

You have probably heard someone say that arrays are pointers in C. Lets explore this fact.

From what we've shown previously you may be able to see how everything you do with an array you can do with a pointer and the operations defined above.

For an example consider an array of integers

```
int A[5] = {6,7,8,9,10};

  printf("Val of base pointer A = 0x%x\n", A);

  printf("_____________________________________________________\n");
  for(int i=0; i < 5; i++){
    printf("*(A+%d) = %d | ", i, *(A+i));
    printf("A[%d] = %d | ", i, A[i]);
    printf("Address of value at (A+%d) = 0x%x\n", i, (A+i));
  }
```

Output

```
Val of base pointer A = 0x388e5690
_____________________________________________________
*(A+0) = 6 | A[0] = 6 | Address of value at (A+0) = 0x388e5690
*(A+1) = 7 | A[1] = 7 | Address of value at (A+1) = 0x388e5694
*(A+2) = 8 | A[2] = 8 | Address of value at (A+2) = 0x388e5698
*(A+3) = 9 | A[3] = 9 | Address of value at (A+3) = 0x388e569c
*(A+4) = 10 | A[4] = 10 | Address of value at (A+4) = 0x388e56a0
```

So we can see that array indexing is effectively the same as incrementing and dereferencing a pointer. Note how the pointer val increases in increments of 4. Exactly the size of the int data type.

`A` is really just a single int pointer. Its address is the start of the array in memory. By incrementing it we can access the 5 integers stored contiguously beginning at location **0x5220d690** ending at location **0x388e56a0**.