CS50: Week-04-Memory

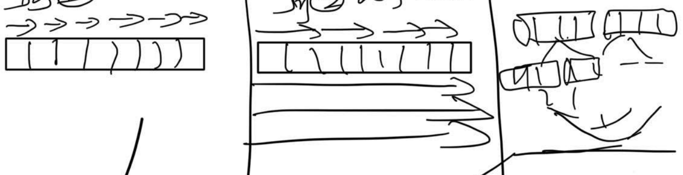
Lecture Notes: https://cs50.harvard.edu/x/2025/notes/4/

Pixel Art

RGB - Hexa decimal is shorthand of the RGB

Selection sort | Bubble sort | Merge sort

By ① | By ② adjacent



O - n | $O = (n^2)$ | $O = n\log n$

Ω - (n) | $\Omega = (n)$ | $\Omega = n\log n$

Big "O" ← O - worset case senerio.

Omega ← Ω Best case seneria.

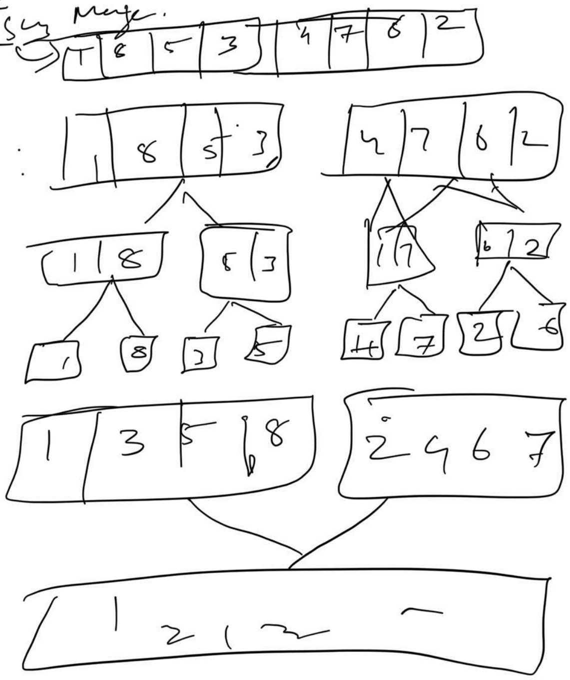
SS 7 8 1 2 3 4 5 6

1 2

BS. 1 2 7 8 | 3 4 5 6

1 2 3 4 5 | 6 5 6

Merge.



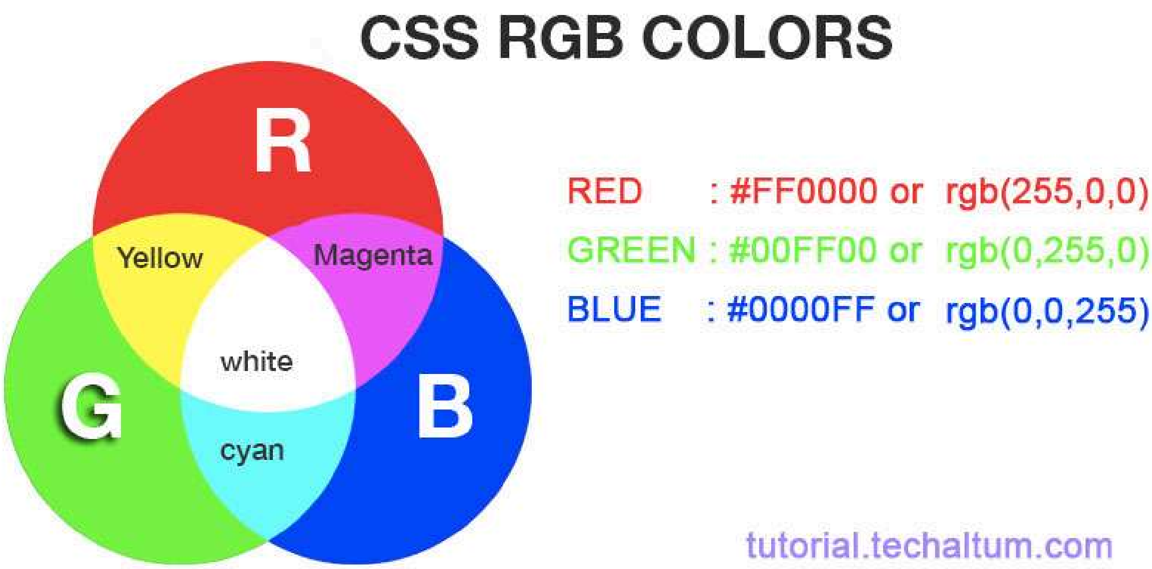
7 3 8 2 1 ... 7

1 2 3 4 5 6 7 8.

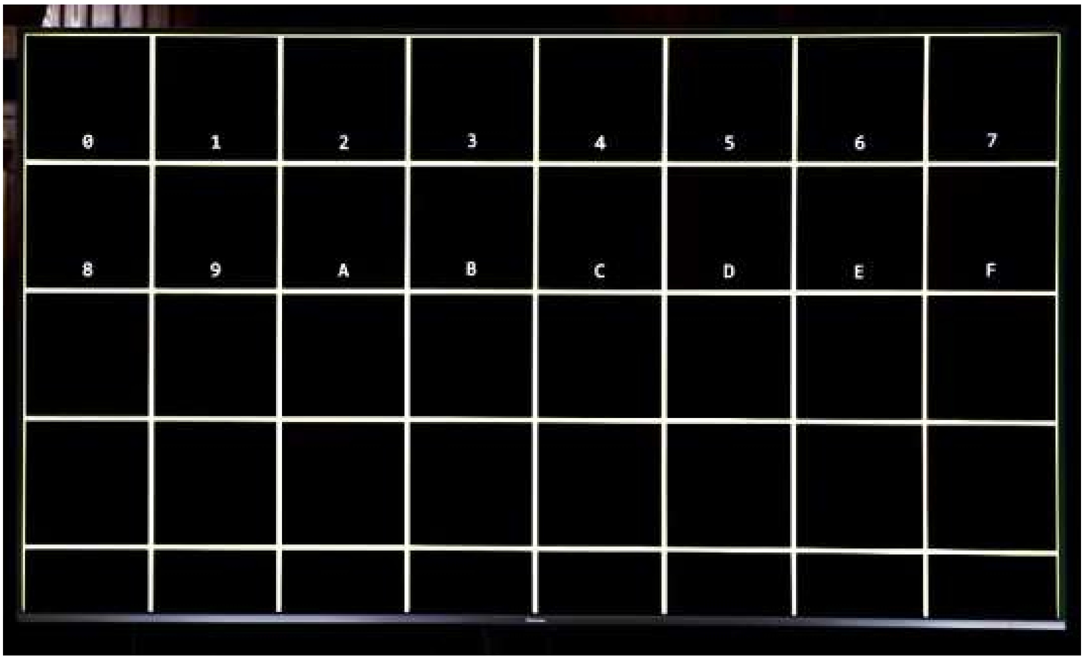
$n(\log n)$

n 8

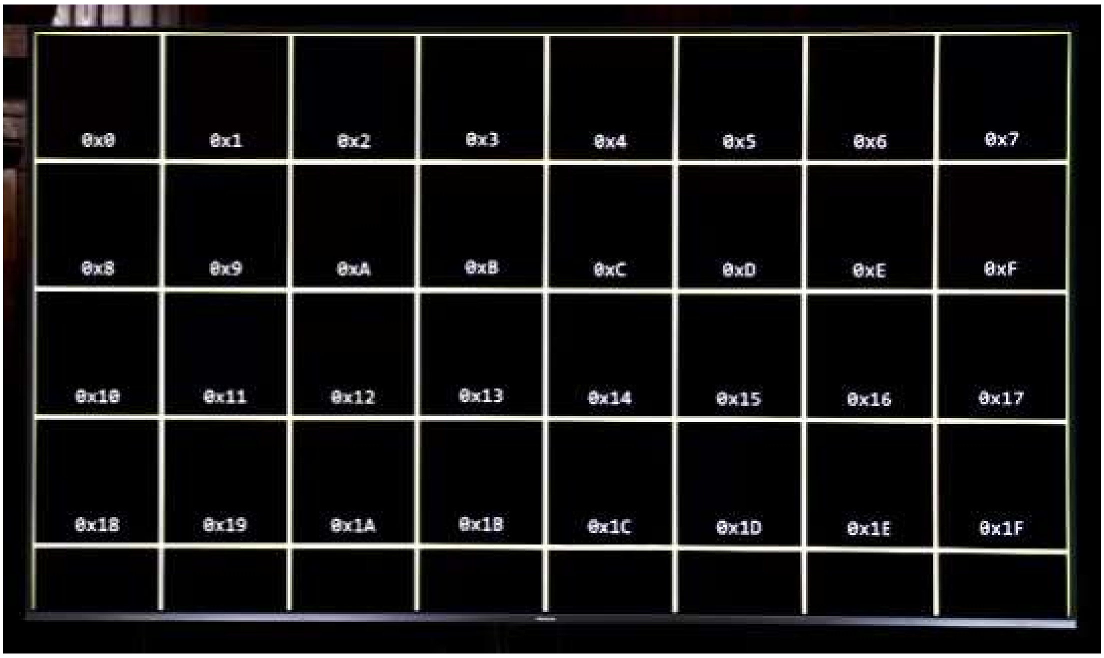

Color Picker (Foreground Color)
OK
Cancel
Add to Swatches
Color Libraries
new
current
H: 0 °
S: 0 %
B: 0 %
R: 0
G: 0
B: 0
L: 0
a: 0
b: 0
C: 75 %
M: 68 %
Y: 67 %
K: 90 %
000000
Only Web Colors


CSS RGB COLORS
R
G
B
Yellow
Magenta
white
cyan
RED : #FF0000 or rgb(255,0,0)
GREEN : #00FF00 or rgb(0,255,0)
BLUE : #0000FF or rgb(0,0,255)
tutorial.techaltum.com

Name	Color	Code	RGB	HSL
white		#ffffff or #fff	rgb(255,255,255)	hsl(0,0%,100%)
silver		#c0c0c0	rgb(192,192,192)	hsl(0,0%,75%)
gray		#808080	rgb(128,128,128)	hsl(0,0%,50%)
black		#000000 or #000	rgb(0,0,0)	hsl(0,0%,0%)
maroon		#800000	rgb(128,0,0)	hsl(0,100%,25%)
red		#ff0000 or #f00	rgb(255,0,0)	hsl(0,100%,50%)
orange		#ffa500	rgb(255,165,0)	hsl(38.8,100%,50%)
yellow		#ffff00 or #ff0	rgb(255,255,0)	hsl(60,100%,50%)
olive		#808000	rgb(128,128,0)	hsl(60,100%,25%)
lime		#00ff00 or #0f0	rgb(0,255,0)	hsl(120,100%,50%)
green		#008000	rgb(0,128,0)	hsl(120,100%,25%)
aqua		#00ffff or #0ff	rgb(0,255,255)	hsl(180,100%,50%)
blue		#0000ff or #00f	rgb(0,0,255)	hsl(240,100%,50%)
navy		#000080	rgb(0,0,128)	hsl(240,100%,25%)
teal		#008080	rgb(0,128,128)	hsl(180,100%,25%)
fuchsia		#ff00ff or #f0f	rgb(255,0,255)	hsl(300,100%,50%)
purple		#800080	rgb(128,0,128)	hsl(300,100%,25%)




0
1
2
3
6
7
8
9
A
B
C
D
E
F
11
12
13
14
15
18
19
16 Binary Bit
=
2 Hexa Bit


0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xA
0xB
0xC
0xD
0xE
0xF
0x10
0x11
0x12
0x13
0x14
0x15
0x16
0x17
0x18
0x19
0x1A
0x1B
0x1C
0x1D
0x1E
0x1F

```c
int n = 50;

int *p = &n;
```

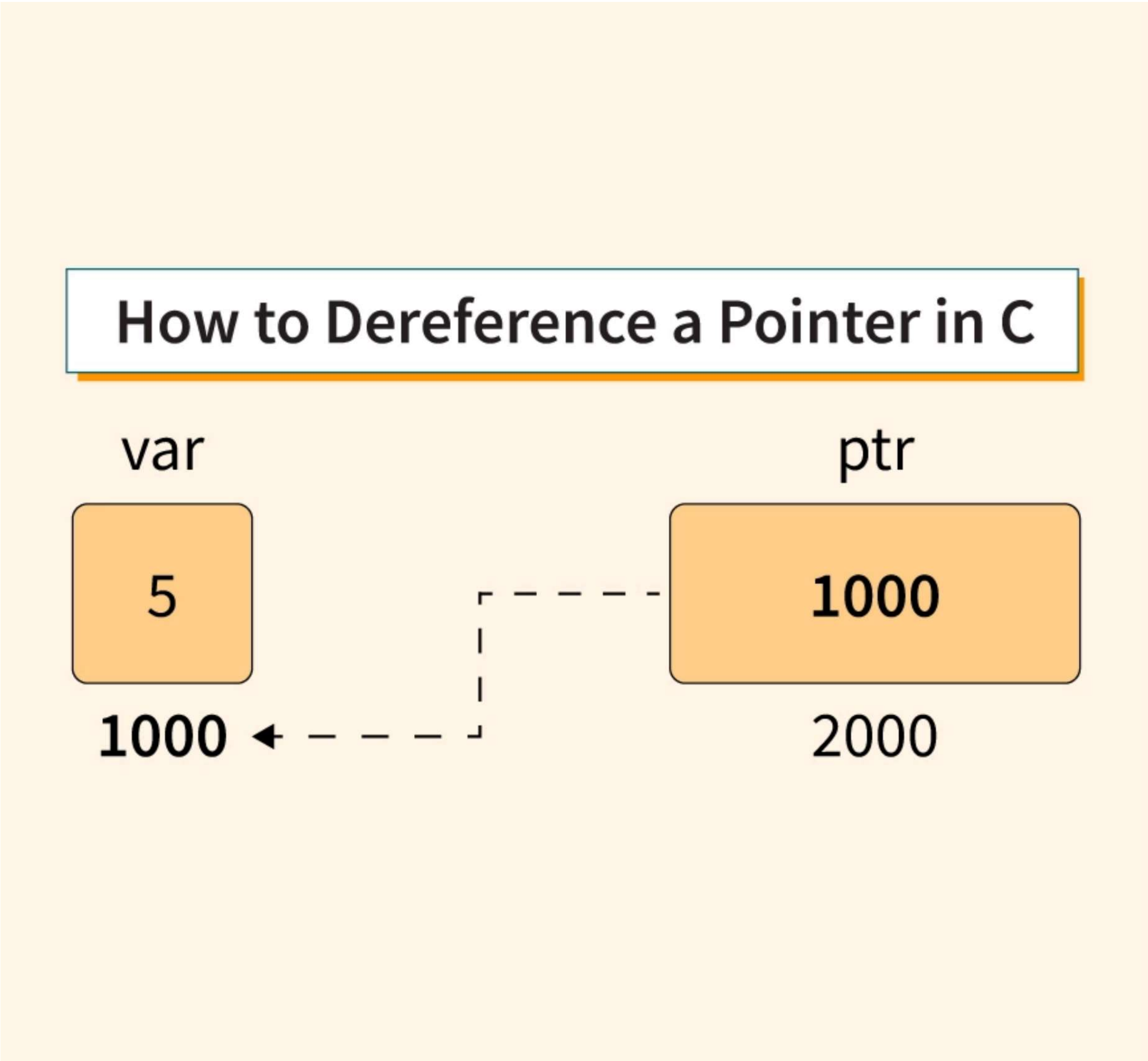
addresses.c

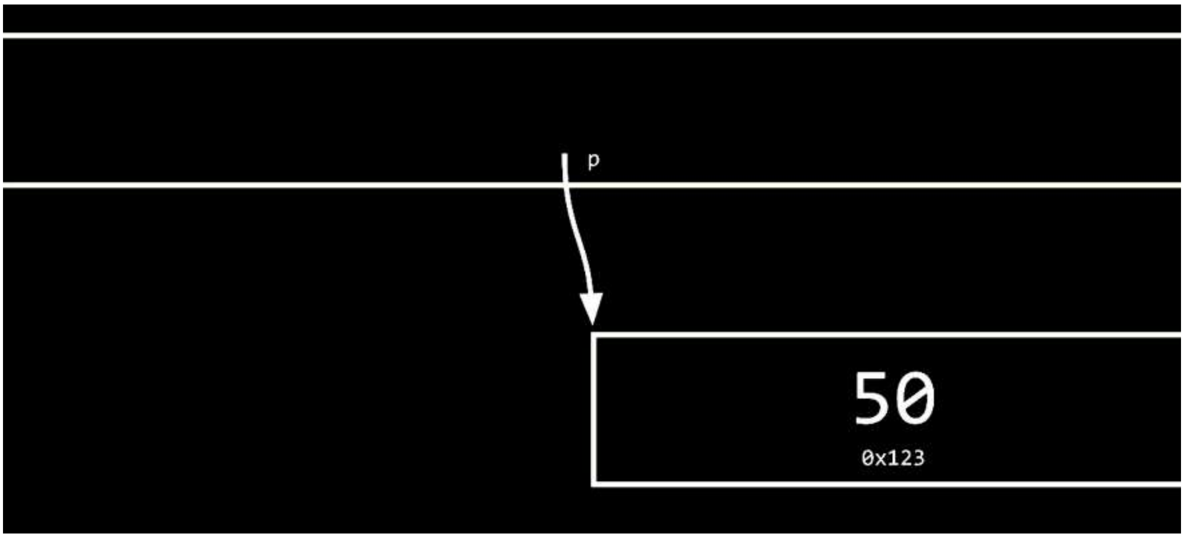
```c
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%p\n", &n);
}

```

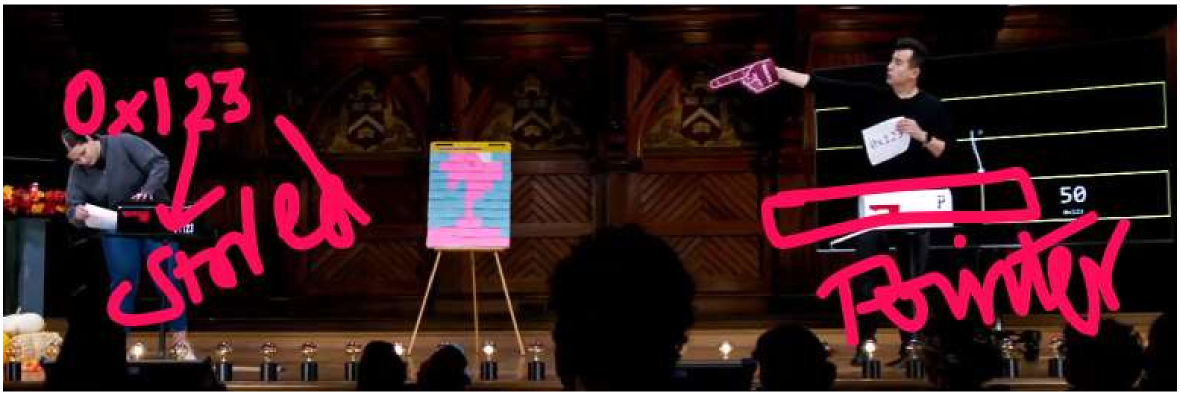
TERMINAL

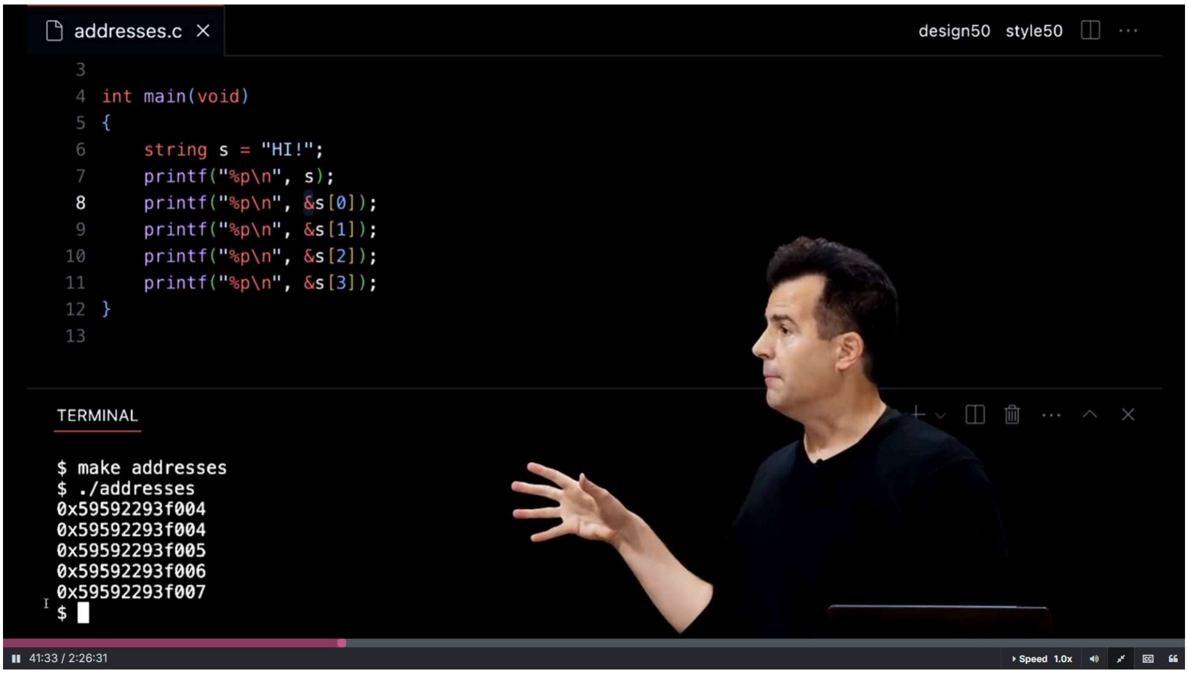
```
$ make addresses
$ ./addresses
0x7ffef3c2925c
$ make addresses
$ ./addresses
0x7ffc5f13d28c
$
```

Dereferencing is used to access or find out the data which is contained in the memory location which is pointed by the pointer. The * (asterisk) operator which is also known as the C dereference pointer is used with the pointer variable to dereference the pointer.2


How to Dereference a Pointer in C
var
ptr
5
1000
1000
2000


p
50
0x123

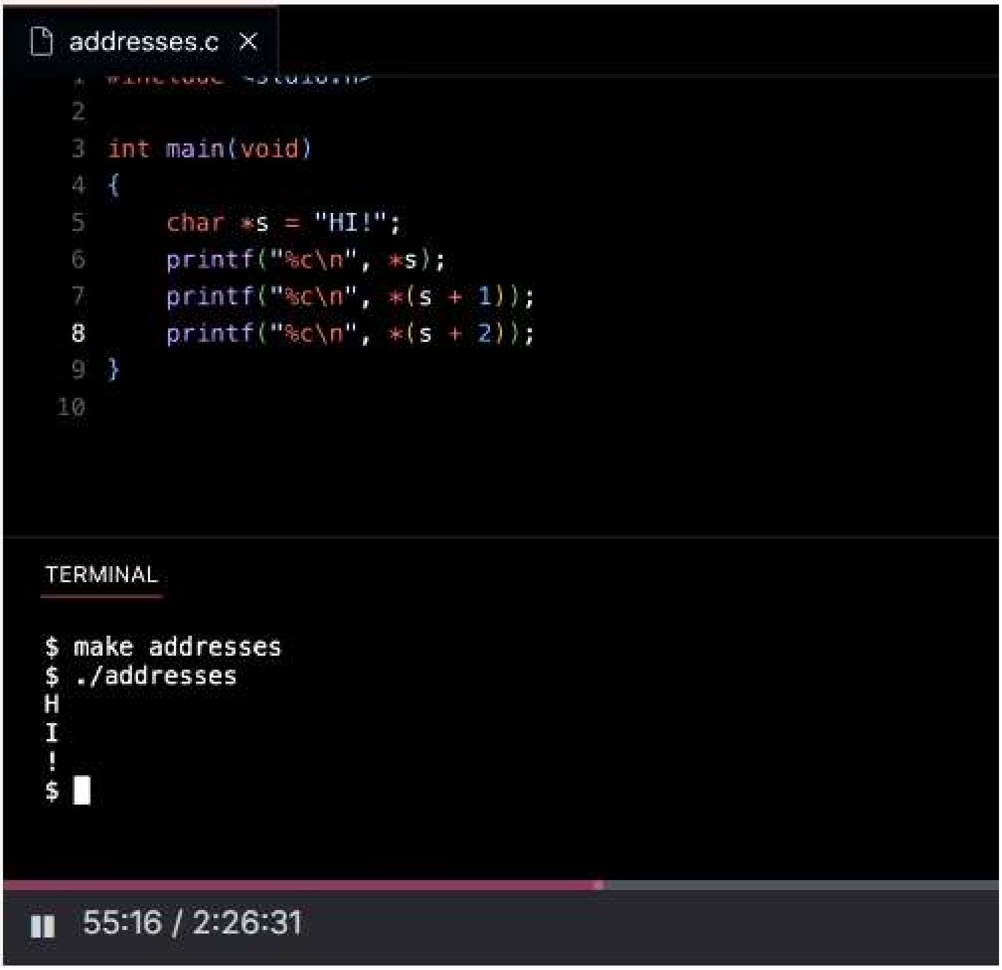





```
#include <stdio.h>

int main(void)
{
int x = 42; // A normal integer variable
int *ptr = &x; // A pointer storing the address of x

printf("Address of x: %p\n", &x); // Prints memory address of x
printf("Value of x: %d\n", x); // Prints the actual value of x

printf("Value of ptr: %p\n", ptr); // Prints the memory address stored in
ptr
printf("Dereferenced ptr: %d\n", *ptr); // Dereferencing: Accessing value at
ptr

return 0;
```

```
}
```

Pointer Arithmetic



- **Pointer Arithmetic** simply means performing mathematical operations (like addition and subtraction) on pointers in programming languages like C and C++.
- Since pointers store memory addresses, pointer arithmetic allows you to move through memory efficiently. The key operations include:
  1. **Increment (** `ptr+` **)** → Moves the pointer to the next memory location.
  2. **Decrement (** `ptr--` **)** → Moves the pointer to the previous memory location.
  3. **Addition (** `ptr + n` **)** → Moves the pointer forward by `n` elements.
  4. **Subtraction (** `ptr - n` **)** → Moves the pointer backward by `n` elements.
  5. **Difference (** `ptr2 - ptr1` **)** → Finds the number of elements between two pointers.

- Example in C:

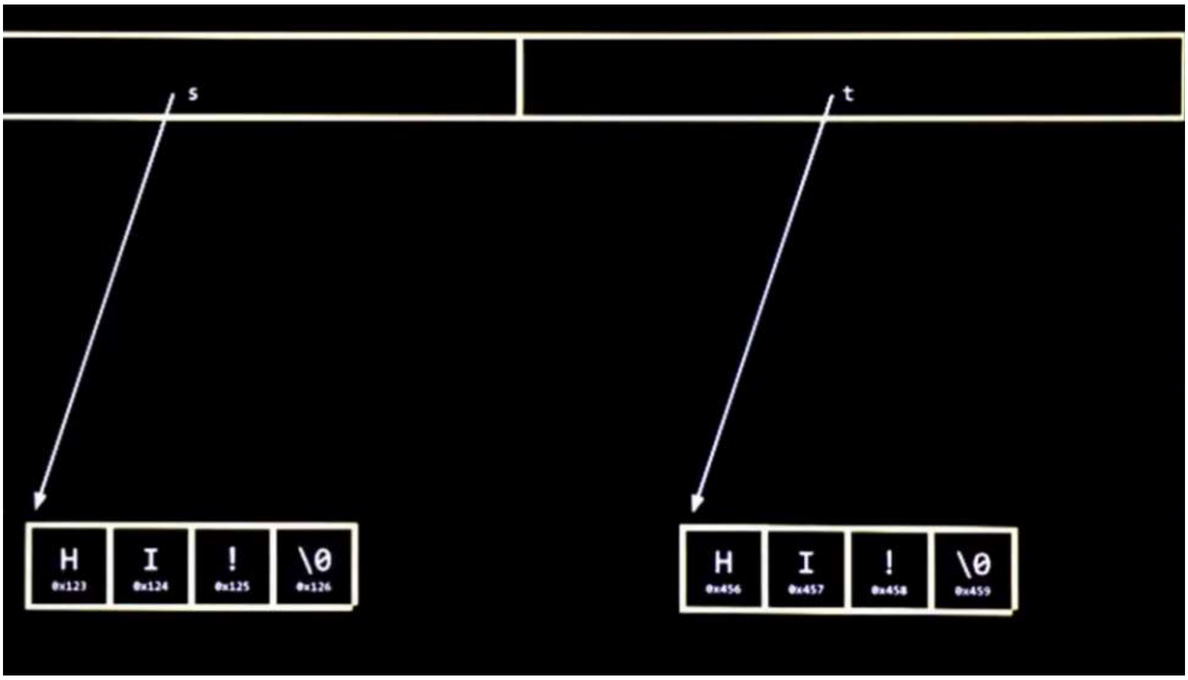
```
#include <stdio.h>

int main() {
int arr[] = {10, 20, 30, 40, 50};
int *ptr = arr; // Pointer to first element

printf("Value at ptr: %d\n", *ptr); // 10
ptr++; // Move to next element
printf("Value at ptr: %d\n", *ptr); // 20

return 0;
}
```

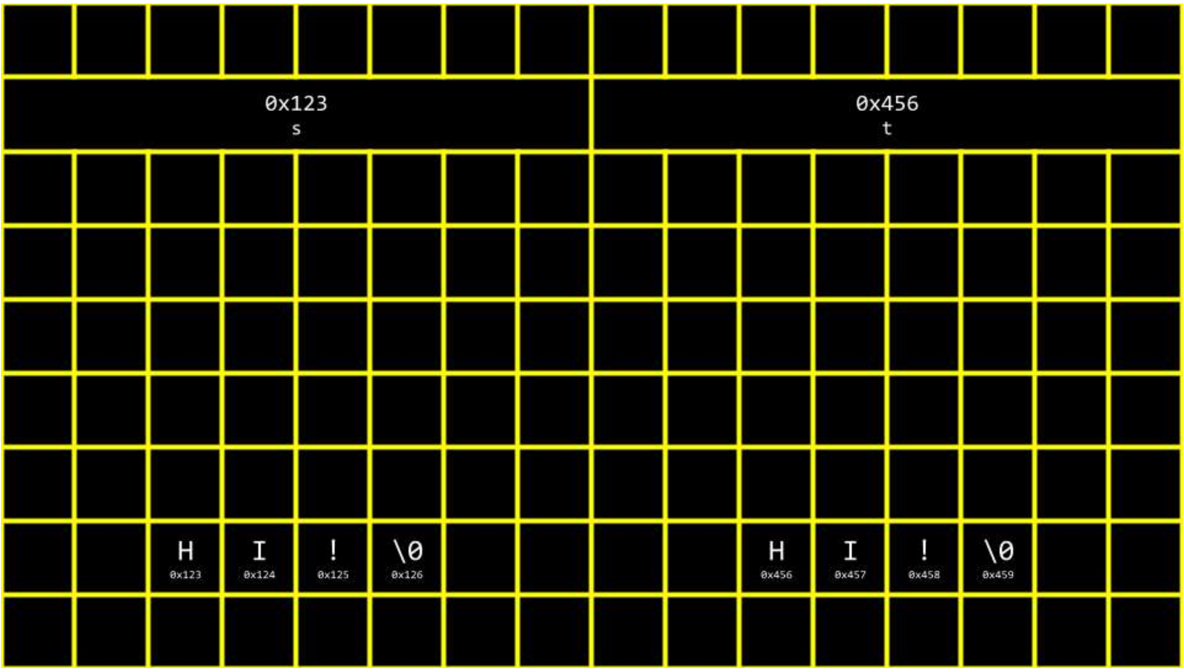
1. Since `ptr` points to an integer (which is 4 bytes on most systems), `ptr++` increases its address by 4 bytes.
2. Would you like me to explain any specific aspect in detail? 🚀

String Comparison

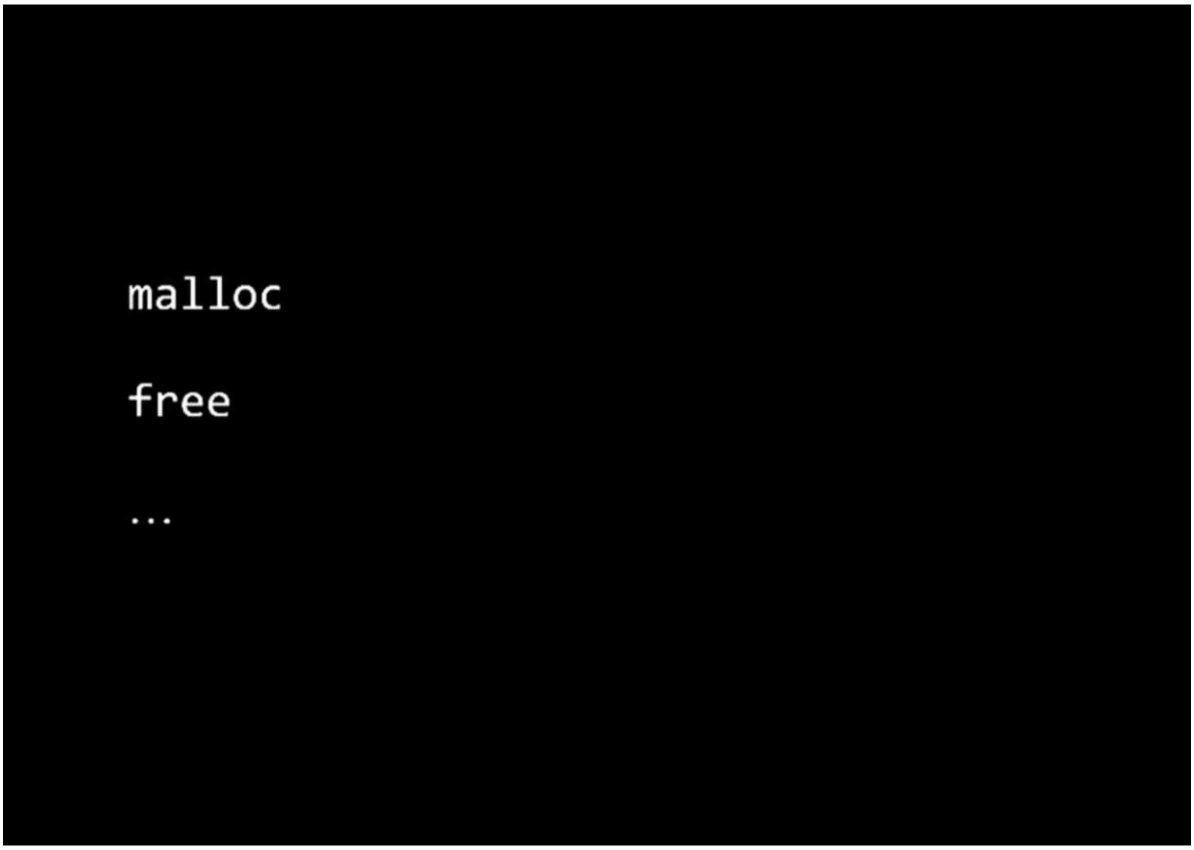
compare.c

```c
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    if (s == t)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}

```




0x123
s
0x456
t
H
0x123
I
0x124
!
0x125
\0
0x126
H
0x456
I
0x457
!
0x458
\0
0x459

Malloc


malloc
free
...

Valgrind, Garbage Values, Pointer Fun with Billy, File I/O

Valgrind

- *Valgrind* is a tool that can check to see if there are memory-related issues with your programs wherein you utilized `malloc`. Specifically, it checks to see if you `free` all the memory you allocated.
- Consider the following code for `memory.c`:

```
// Demonstrates memory errors via valgrind

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
}
```

  Notice that running this program does not cause any errors. While `malloc` is used to allocate enough memory for an array, the code fails to `free` that allocated memory.
- If you type `make memory` followed by `valgrind ./memory`, you will get a report from valgrind that will report where memory has been lost as a result of your program. One error that valgrind reveals is that we attempted to assign the value of `33` at the 4th position of the array, where we only allocated an array of size `3`. Another error is that we never freed `x`.
- You can modify your code to free the memory of `x` as follows:

```
// Demonstrates memory errors via valgrind

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
    free(x);
}
```

  Notice that running valgrind again now results in no memory leaks.

Garbage Values

- When you ask the compiler for a block of memory, there is no guarantee that this memory will be empty.
- It's very possible that the memory you allocated was previously utilized by the computer. Accordingly, you may see *junk* or *garbage values*. This is a result of you getting a block of memory but not initializing it. For example, consider the following code for `garbage.c`:

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

int main(void)
{
    int scores[1024];
    for (int i = 0; i < 1024; i++)
    {
        printf("%i\n", scores[i]);
    }
}
```

  Notice that running this code will allocate `1024` locations in memory for your array, but the `for` loop will likely show that not all values therein are `0`. It's always best practice to be aware of the potential for garbage values when you do not initialize blocks of memory to some other value like zero or otherwise.

Pointer Fun with Binky

- We watched a video from Stanford University that helped us visualize and understand pointers.

File I/O

- You can read from and manipulate files. While this topic will be discussed further in a future week, consider the following code for `phonebook.c`:

  ```
  // Saves names and numbers to a CSV file

  #include <cs50.h>
  #include <stdio.h>
  #include <string.h>

  int main(void)
  {
      // Open CSV file
      FILE *file = fopen("phonebook.csv", "a");

      // Get name and number
      char *name = get_string("Name: ");
      char *number = get_string("Number: ");

      // Print to file
      fprintf(file, "%s,%s\n", name, number);

      // Close file
      fclose(file);
  }
  ```

  Notice that this code uses pointers to access the file.

- You can create a file called `phonebook.csv` in advance of running the above code or download phonebook.csv. After running the above program and inputting a name and phone number, you will notice that this data persists in your CSV file.

- If we want to ensure that `phonebook.csv` exists prior to running the program, we can modify our code as follows:

  ```
  // Saves names and numbers to a CSV file

  #include <cs50.h>
  #include <stdio.h>
  #include <string.h>

  int main(void)
  {
      // Open CSV file
      FILE *file = fopen("phonebook.csv", "a");
      if (!file)
      {
          return 1;
      }

      // Get name and number
      char *name = get_string("Name: ");
      char *number = get_string("Number: ");

      // Print to file
      fprintf(file, "%s,%s\n", name, number);
  ```

notice that this data persists in your CSV file.

- If we want to ensure that `phonebook.csv` exists prior to running the program, we can modify our code as follows:

```
// Saves names and numbers to a CSV file

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Open CSV file
    FILE *file = fopen("phonebook.csv", "a");
    if (!file)
    {
        return 1;
    }

    // Get name and number
    char *name = get_string("Name: ");
    char *number = get_string("Number: ");

    // Print to file
    fprintf(file, "%s,%s\n", name, number);

    // Close file
    fclose(file);
}
```

Notice that this program protects against a `NULL` pointer by invoking `return 1`.

- We can implement our own copy program by typing `code cp.c` and writing code as follows:

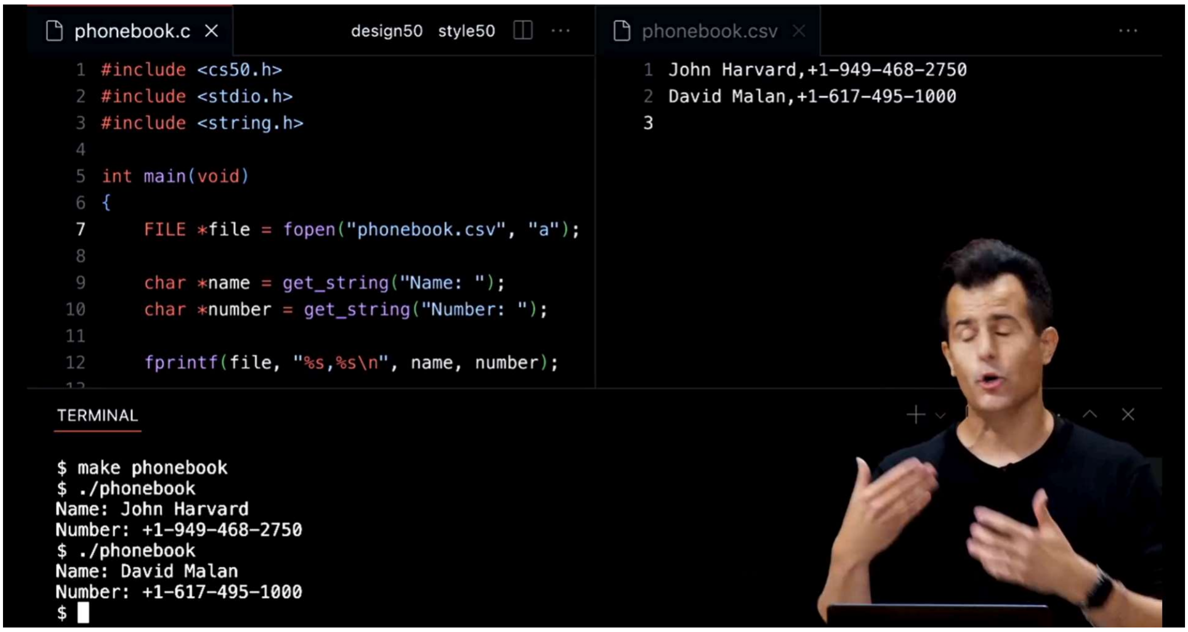
```
// Copies a file

#include <stdio.h>
#include <stdint.h>

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    FILE *src = fopen(argv[1], "rb");
    FILE *dst = fopen(argv[2], "wb");

    BYTE b;

    while (fread(&b, sizeof(b), 1, src) != 0)
    {
        fwrite(&b, sizeof(b), 1, dst);
    }

    fclose(dst);
    fclose(src);
}
```

Notice that this file creates our own data type called a BYTE , which is the size of a uint8_t. Then, the file reads a `BYTE` and writes it to a file.

- BMPs are also assortments of data that we can examine and manipulate. This week, you will be doing just that in your problem sets!

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

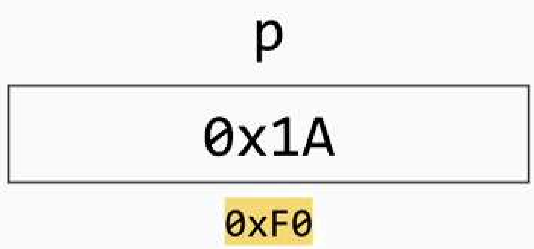
```
fopen
fclose
fprintf
fscanf
fread
fwrite
fseek
...
```



Section-04 by Yuleia

Pointers

```
int *p = 0x1A;
```



Pointer Syntax

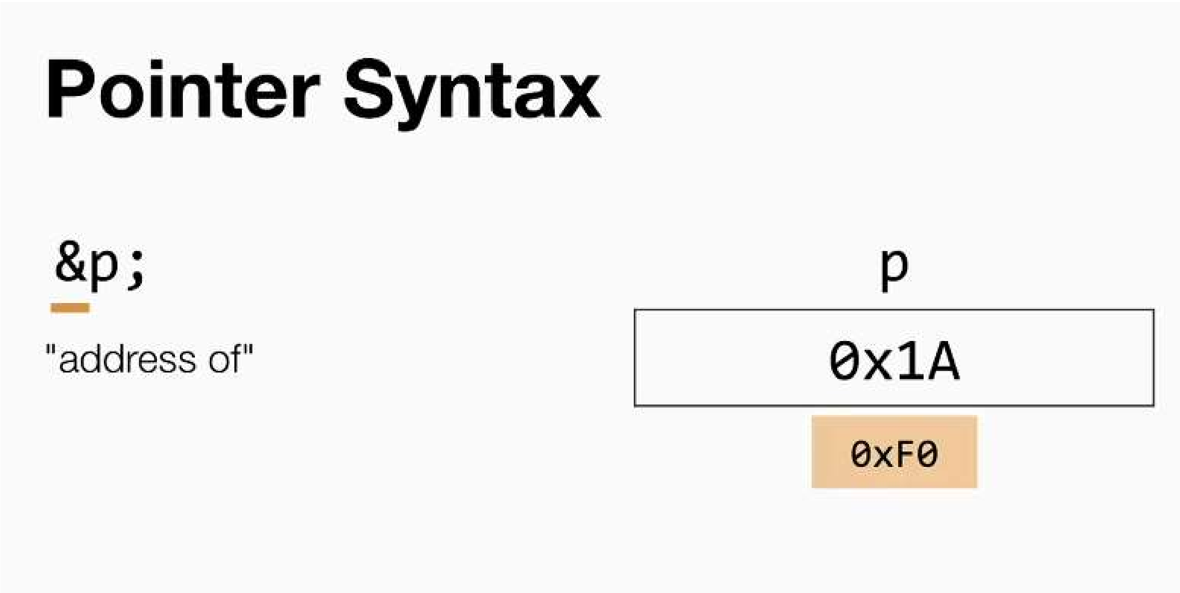
```
calls;
```

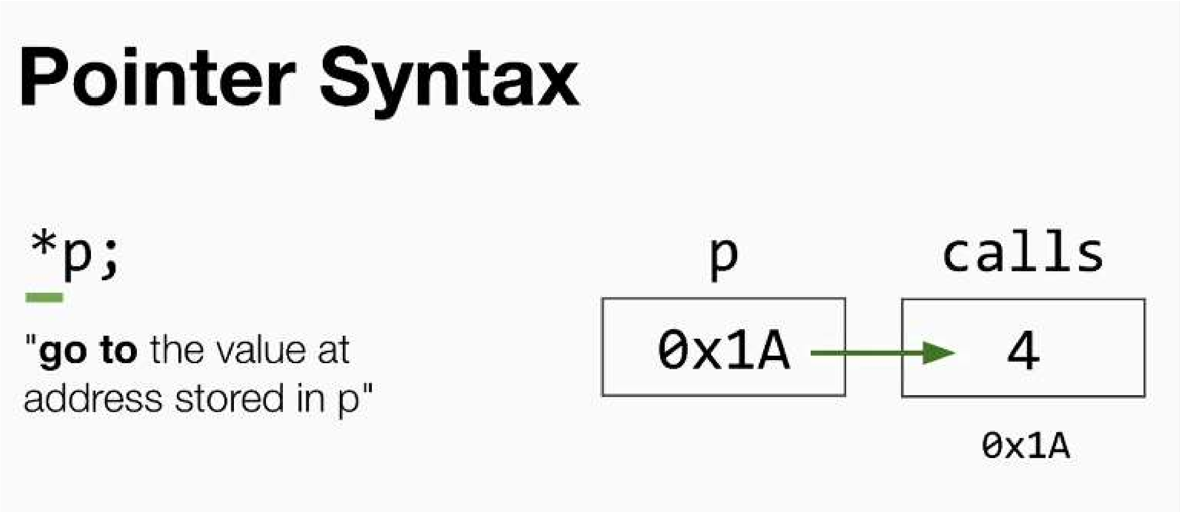
"value of"

Pointer Syntax

```
&calls;
```

"address of"


Pointer Syntax
&p;
"address of"
p
0x1A
0xF0


Pointer Syntax
*p;
"go to the value at address stored in p"
p
calls
0x1A
4
0x1A


type * is a pointer that stores the address of a type.
*x takes a pointer x and goes to the address s
that pointer.
&x takes x and gets its address.

Pointers practice

```
int x = 4;
int *p = &x;
printf("%i\n", *p);
```

p	x
0x123	4

0x123

Terminal:
4

Pointers practice

```
int x = 4;
int *p = &x;
printf("%i\n", *p);


*p = 2;
```

p	x
0x123	2

0x123

Terminal:
4

```
FILE *input = fopen("hi.txt", "r");
```

name

input

hi.txt

hi!

- `fread` reads data from a file into a buffer
- `fwrite` write data from a buffer to a file

A buffer is a chunk of memory that can temporarily store some data from the file.

File Reading Exercise

Create a program, **pdf.c**, that checks whether a file, passed in as a command-line argument, is a PDF. All PDFs will begin with a four byte sequence:

`0x25 0x50 0x44 0x46`

For example: `./pdf test.pdf` should print "yes", while `./pdf test.jpg` should print "no".

WEEK-04-Shorts

Hexadecimal

Hexadecimal

- The **hexadecimal system,** aka *base-16*, is a much more concise way to express the data on a computer's system.

```
0 1 2 3 4 5 6 7 8 9 a b c d e f
```

- Hexadecimal makes this mapping easy because a group of four binary digits (bits) is able has 16 different combinations, and each of those combinations maps to a single hexadecimal digit.

Hexadecimal

```
01000110101000101011100100111101
```

```
0100 0110 1010 0010 1011 1001 0011 1101
```

Pointers

Pointers

- Pointers provide an alternative way to pass data between functions.
  - Recall that up to this point, we have passed all data **by value**, with one exception.
  - When we pass data by value, we only pass a copy of that data.

- If we use pointers instead, we have the power to pass the actual variable itself.
  - That means that a change that is made in one function can impact what happens in a different function.
  - Previously, this wasn't possible!

Pointers

- Every file on your computer lives on your disk drive, be it a hard disk drive (HDD) or a solid-state drive (SSD).

- Disk drives are just storage space; we can't directly work there. Manipulation and use of data can only take place in RAM, so we have to move data there.

- Memory is basically a huge array of 8-bit wide bytes.
  - 512 MB, 1GB, 2GB, 4GB...

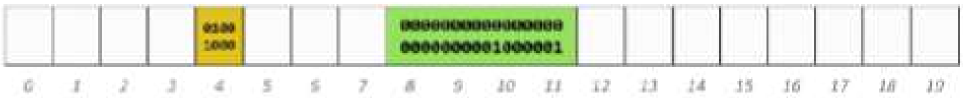
Pointers

Data Type	Size (in bytes)
`int`	4
`char`	1
`float`	4
`double`	8
`long long`	8
`string`	???

Pointers

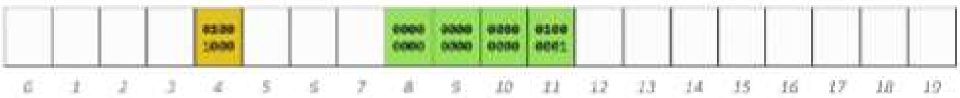
- Back to this idea of memory as a big array of byte-sized cells.
- Recall from our discussion of arrays that they not only are useful for storage of information but also for so-called **random access.**
  - We can access individual elements of the array by indicating which index location we want.
- Similarly, each location in memory has an **address.**

Pointers



```
char c = 'H';
int speedlimit = 65;
```

Pointers



```
char c = 'H';
int speedlimit = 65;
```

Pointers

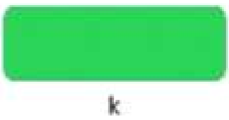

```
char c = 'H';
int speedlimit = 65;
```

Pointers


```
char c = 'H';
int speedlimit = 65;
string surname = "Lloyd";
```

Pointers

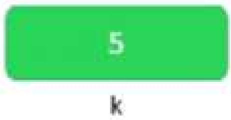
- As we start to work with pointers, just keep this image in mind:




```
int k;
```

Pointers

- As we start to work with pointers, just keep this image in mind:




```
int k;
k = 5;
```

Pointers

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int* pk;
pk = &k;
```

Pointers

- As we start to work with pointers, just keep this image in mind:


```
int k;
k = 5;
int* pk;
pk = &k;
```

Pointers

- A **pointer**, then, is a data item whose
  - *value* is a memory address
  - *type* describes the data located at that memory address

- As such, pointers allow data structures and/or variables to be shared among functions.

- Pointers make a computer environment more like the real world.

Pointers

- The simplest pointer available to us in C is the NULL pointer.
  - As you might expect, this pointer points to nothing (a fact which can actually come in handy!)

- When you create a pointer and you don't set its value immediately, you should **always** set the value of the pointer to NULL.

- You can check whether a pointer is NULL using the equality operator (==).

Pointers

- Another easy way to create a pointer is to simply **extract** the address of an already existing variable. We can do this with the address extraction operator (&).

- If x is an `int`-type variable, then &x is a pointer-to-`int` whose value is the address of x.
- If arr is an array of `double`s, then &arr[i] is a pointer-to-`double` who value is the address of the i^th element of arr.
  - An array's name, then, is actually just a pointer to its first element – you've been working with pointers all along!

Pointers

- The main purpose of a pointer is to allow us to modify or inspect the location to which it points.
  - We do this by **dereferencing** the pointer.

- If we have a pointer-to-`char` called pc, then *pc is the data that lives at the memory address stored inside the variable pc.

Pointers

- Can you guess what might happen if we try to dereference a pointer whose value is NULL?

Segmentation fault

- Surprisingly, this is actually good behavior! It defends against accidental dangerous manipulation of unknown pointers.
  - That's why we recommend you set your pointers to NULL immediately if you aren't setting them to a known, desired value.

Pointers

```
int* p;
```

- The value of p is an address.
- We can dereference p with the * operator.
- If we do, what we'll find at that location is an `int`.

Pointers

- One more annoying thing with those *s. They're an important part of both the type name **and** the variable name.
  - Best illustrated with an example.

~~`int* px, py, pz;`~~

```
int* pa, *pb, *pc;
```

Pointers

Data Type	Size (in bytes)
int	4
char	1
float	4
double	8
long long	8
string	???

Pointers

Data Type	Size (in bytes)
int	4
char	1
float	4
double	8
long long	8
char*, int*, float*, double*, ________*	4 or 8

Defining Custom Types

Defining Custom Data Types

```
typedef char* string;
```

Defining Custom Data Types

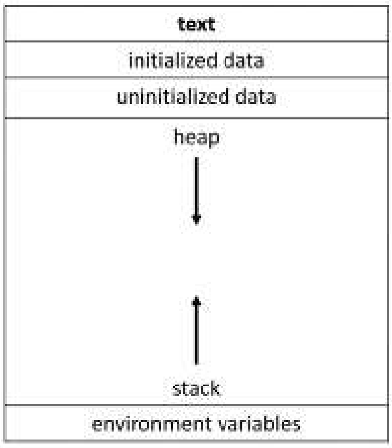
```
struct car
{
     int year;
     char model[10];
     char plate[7];
     int odometer;
     double engine_size;
};

typedef struct car car_t;
```

Dynamic Memory Allocation

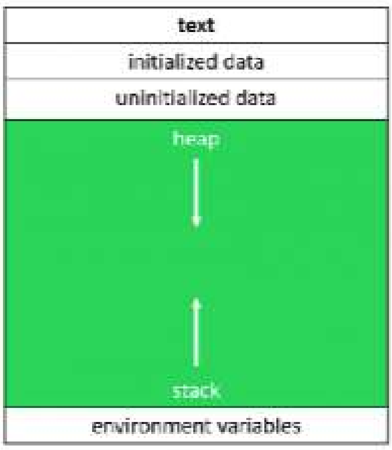
Dynamic Memory Allocation

- We can use pointers to get access to a block of **dynamically-allocated memory** at runtime.
- Dynamically allocated memory comes from a pool of memory known as the **heap.**
- Prior to this point, all memory we've been working with has been coming from a pool of memory known as the **stack.**

Dynamic Memory Allocation



Dynamic Memory Allocation



Dynamic Memory Allocation

- We get this dynamically-allocated memory by making a call to the C standard library function `malloc()`, passing as its parameter the number of bytes requested.
- After obtaining memory for you (if it can), `malloc()` will return a pointer to that memory.
- What if `malloc()` **can't** give you memory? It'll hand you back `NULL`.

Dynamic Memory Allocation

```
// statically obtain an integer
int x;

// dynamically obtain an integer
int *px = malloc(4);
```

Dynamic Memory Allocation

```
// statically obtain an integer
int x;

// dynamically obtain an integer
int *px = malloc(sizeof(int));
```

Dynamic Memory Allocation

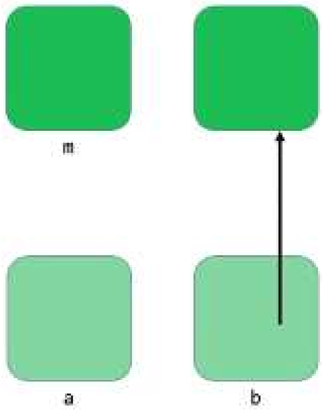
- Here's the trouble: Dynamically-allocated memory is not automatically returned to the system for later use when the function in which it's created finishes execution.
- Failing to return memory back to the system when you're finished with it results in a **memory leak** which can compromise your system's performance.
- When you finish working with dynamically-allocated memory, you must `free()` it.

Dynamic Memory Allocation

- Three golden rules:
    1. Every block of memory that you `malloc()` must subsequently be `free()`d.
    2. Only memory that you `malloc()` should be `free()`d.
    3. Do not `free()` a block of memory more than once.

Dynamic Memory Allocation

```
int m;
int* a;
int* b = malloc(sizeof(int));
```



Call Stacks

Call Stack

- When you call a function, the system sets aside space in memory for that function to do its necessary work.
  - We frequently call such chunks of memory **stack frames** or **function frames**.

- More than one function's stack frame may exist in memory at a given time. If `main()` calls `move()`, which then calls `direction()`, all three functions have open frames.

Call Stack

- These frames are arranged in a **stack**. The frame for the most-recently called function is always on the top of the stack.

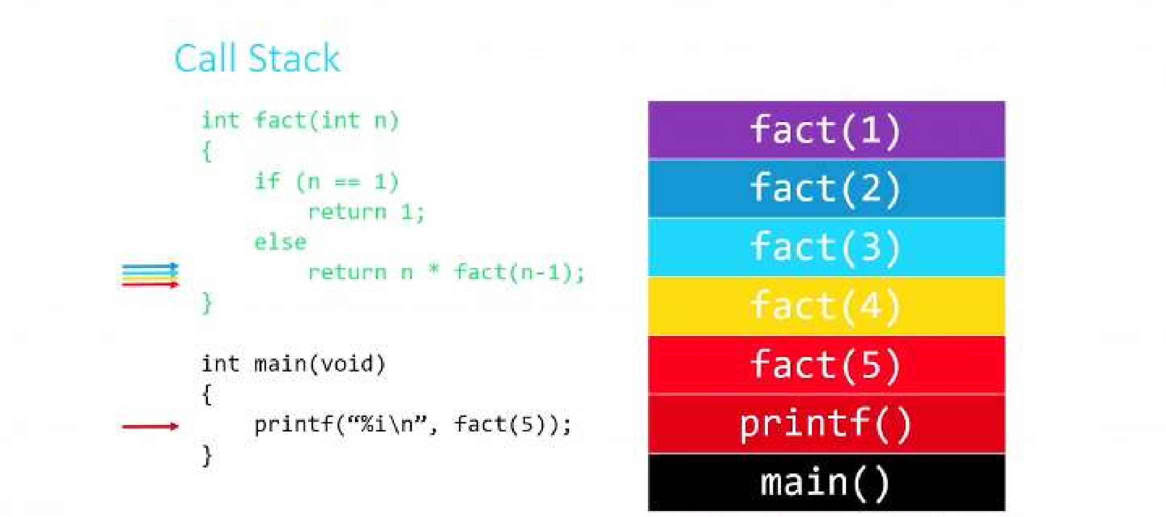
- When a new function is called, a new frame is **pushed** onto the top of the stack and becomes the active frame.

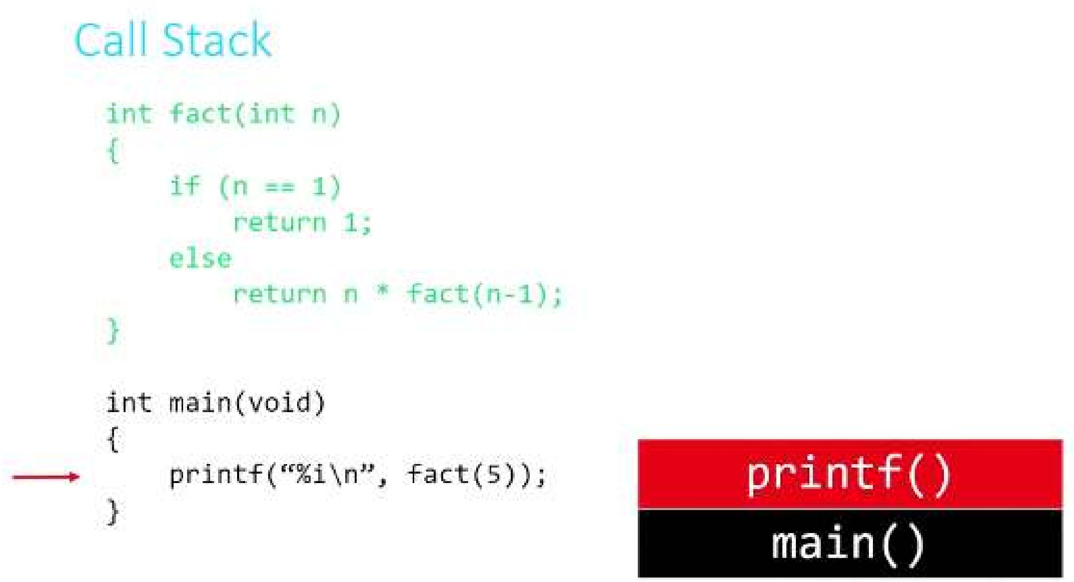
- When a function finishes its work, its frame is **popped** off of the stack, and the frame immediately below it becomes the new, active, function on the top of the stack. This function picks up immediately where it left off.

Call Stack

```
int fact(int n)
{
    if (n == 1)
        return 1;
    else
        return n * fact(n-1);
}

int main(void)
{
    printf("%i\n", fact(5));
}
```

main()





File Pointers

File Pointers

- The ability to read data from and write data to files is the primary means of storing **persistent data**, data that does not disappear when your program stops running.

- The abstraction of files that C provides is implemented in a data structure known as a `FILE`.
  - Almost universally when working with files, we will be using pointers to them, `FILE*`.

File Pointers

- The file manipulation functions all live in `stdio.h`.
  - All of them accept `FILE*` as one of their parameters, except for the function `fopen()`, which is used to get a file pointer in the first place.

- Some of the most common file input/output (I/O) functions that we'll be working with are:

fopen()	fclose()	fgetc()	fputc()	fread()	fwrite()

File Pointers

- `fopen()`
  - Opens a file and returns a file pointer to it.
  - Always check the return value to make sure you don't get back NULL.

```
FILE* ptr = fopen(<filename>, <operation>);
```

File Pointers

- `fclose()`
  - Closes the file pointed to by the given file pointer.

```
fclose(<file pointer>);
```

File Pointers

- `fclose()`
  - Closes the file pointed to by the given file pointer.

```
fclose(ptr1);
```

File Pointers

- `fgetc()`
  - Reads and returns the next character from the file pointed to.
  - Note: The operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
char ch = fgetc(ptr1);
```

File Pointers

- The ability to get single characters from files, if wrapped in a loop, means we could read all the characters from a file and print them to the screen, one-by-one, essentially.

```
char ch;
while((ch = fgetc(ptr)) != EOF)
   printf("%c", ch);
```

- We might put this in a file called `cat.c`, after the Linux command "`cat`" which essentially does just this.

File Pointers

- The ability to get single characters from files, if wrapped in a loop, means we could read all the characters from a file and print them to the screen, one-by-one, essentially.

```
char ch;
while((ch = fgetc(ptr)) != EOF)
   printf("%c", ch);
```

- We might put this in a file called `cat.c`, after the Linux command "`cat`" which essentially does just this.

File Pointers

- `fputc()`
  - Writes or appends the specified character to the pointed-to file.
  - Note: The operation of the file pointer passed in as a parameter must be “w” for write or “a” for append, or you will suffer an error.

```
fputc(<character>, <file pointer>);
```

File Pointers

- `fputc()`
  - Writes or appends the specified character to the pointed-to file.
  - Note: The operation of the file pointer passed in as a parameter must be “w” for write or “a” for append, or you will suffer an error.

```
fputc(‘A’, ptr2);
```

File Pointers

- Now we can read characters from files and write characters to them. Let’s extend our previous example to copy one file to another, instead of printing to the screen.

```
char ch;
while((ch = fgetc(ptr)) != EOF)
   printf(“%c”, ch);
```

File Pointers

- Now we can read characters from files and write characters to them. Let's extend our previous example to copy one file to another, instead of printing to the screen.

```
char ch;
while((ch = fgetc(ptr)) != EOF)
   fputc(ch, ptr2);
```

- We might put this in a file called `cp.c`, after the Linux command "`cp`" which essentially does just this.

File Pointers

- `fread()`
  - Reads `<qty>` units of size `<size>` from the file pointed to and stores them in memory in a buffer (usually an array) pointed to by `<buffer>`.
  - Note: The operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
fread(<buffer>, <size>, <qty>, <file pointer>);
```

File Pointers

- `fread()`
  - Reads `<qty>` units of size `<size>` from the file pointed to and stores them in memory in a buffer (usually an array) pointed to by `<buffer>`.
  - Note: The operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
int arr[10];
fread(arr, sizeof(int), 10, ptr);
```

File Pointers

- `fread()`
  - Reads <qty> units of size <size> from the file pointed to and stores them in memory in a buffer (usually an array) pointed to by <buffer>.
  - Note: The operation of the file pointer passed in as a parameter must be "r" for read, or you will suffer an error.

```
double* arr2 = malloc(sizeof(double) * 80);
fread(arr2, sizeof(double), 80, ptr);
```

File Pointers

- `fwrite()`
  - Writes <qty> units of size <size> to the file pointed to by reading them from a buffer (usually an array) pointed to by <buffer>.
  - Note: The operation of the file pointer passed in as a parameter must be "w" for write or "a" for append, or you will suffer an error.

```
char c;
fwrite(&c, sizeof(char), 1, ptr);
```

File Pointers

- Lots of other useful functions abound in stdio.h for you to work with. Here are some of the ones you may find useful!

Function	Description
`fgets()`	Reads a full string from a file.
`fputs()`	Writes a full string to a file.
`fprintf()`	Writes a formatted string to a file.
`fseek()`	Allows you rewind or fast-forward within a file.
`ftell()`	Tells you at what (byte) position you are at within a file.
`feof()`	Tells you whether you've read to the end of a file.
`ferror()`	Indicates whether an error has occurred in working with a file.