1. Resolution = The number of dots that are present on the screen.
2. We can represent most colors using red, green and blue.
3. RGB, however, is not represented using decimal values. We instead use hexadecimal. That's why, say white is represented as #FFFFFF
4. Also remember that is the RGB format, the first two represent the R value, next two represent the G value and the last two represent B value.
5. So using the above point, we can conclude red is represented as: #FF0000, green is: #00FF00 and blue is: #0000FF
6. Binary - 0 and 1. Decimal - 0,1,2,3,4,5,6,7,8,9. Hexadecimal - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
7. Hexadecimal is aka base 16.
8. So in hexadecimal, 1 would be represented as (16^1) + (16^0), where the bit under (16^1) would be 0, and the bit under (16^0) would be 1.
9. Similarly, 9 would be represented as (16^1) + (16^0), where the bit under (16^1) would be 0, and the bit under (16^0) would be 9.
10. 10 would be represented as (16^1) + (16^0), where the bit under (16^1) would be 0, and the bit under (16^0) would be A.
11. 12 would be represented as (16^1) + (16^0), where the bit under (16^1) would be 0, and the bit under (16^0) would be C.
12. FF would represent (16 \* 15) + (1 \* 15) = 255
13. The first F represents 4 bits, and the next 4 represents the next 4. So in total, it represents 8 bits.
14. We use hexadecimal in the world of graphics, for convenience. Each bit in hexadecimal represents 4 bits in binary. So when we put 2 bits of hexadecimal together (example: FF), we actually have 8 bits in binary, which represents 1 byte.
15. Hexadecimal therefore allows us to represent information more succinctly.

Memory:

1. In the above example, we represent 255 as FF. The number 16 would be represented as 10.
2. This notation is confusing for us, so we always prefix hexadecimal values with 0x.
3. Therefore, 16 would be 0x10, FF would be 0xFF etc. It's just a notation.
4. & - the "address of" operator. It allows us to ask the computer the question "Hey, at what address is this variable?" The format specifier to print this is "%p" (p for pointer)
5. \* - the "dereference" operator. Allows us to take an address and go to it.

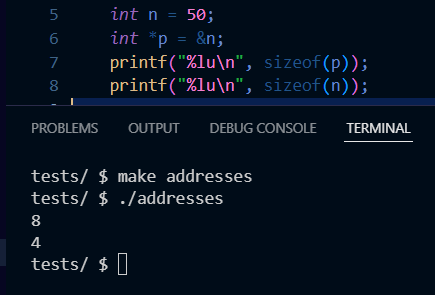
Pointers:

1. Pointers main purpose is to store addresses.

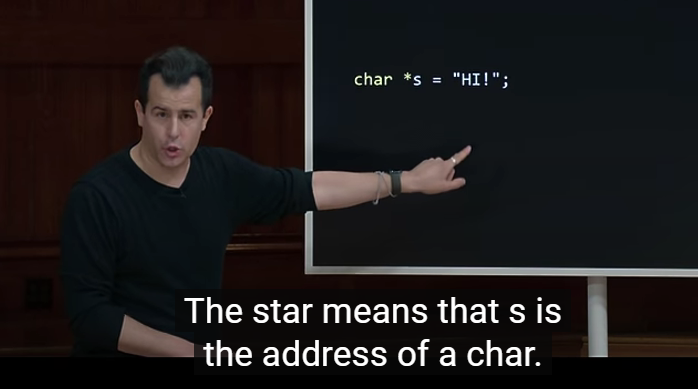
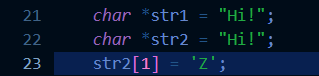
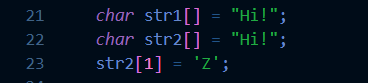
Syntax :

int \*p = &n;

printf("%p", p);

1. pointers tend to take up 8 bytes, and they store the entire memory address. In the example below, p is a pointer and n is an int variable.
2. 
3. When we write char \*p, we are creating a variable p, that will store an address of a character/string type variable.
4. Similarly, when we type int \*p, we are creating a variable p, that will store the address of an int type variable.
5. When we declare a string s = "Hi!', and create a pointer char\* p to it, p actually stores the address of the first letter of the string s. So p stores the address of H.
6. When it comes to printing strings, printf just loops through the entire string character by character.
7. int main(void)
8. {
   1. string s = "HI!";
   2. printf("%p \n", s);
9. }
10. The program above will give you the memory address of "H".
11. We can also write, printf("%p \n", &s[0]);
12. The typedef keyword allows us to create our own data type.
13. typedef struct {int number; char\* name;} person;
14. The above line basically says, "Hey, give me a data type called person, which has a number option and a name option.
15. So in CS50, when we declare a variable with the data type "string", it's actually a typedef char\* string;

Strings:

1. In order to compare strings, we use strcmp.
2. The reason we cannot use the == sign is because we would only be comparing the addresses of the variables in which we stored our string.
3. Even if we used \*(variable name), we would only compare the first letter. (unless we did a loop but at that point, use strcmp)
4. Remember that a the variable that holds a string actually just holds its beginning character address.
5. The actual string is stored elsewhere.
6. When we use the double quotes, the \0 is automatically added to the end of your string, and the & operator is added to the beginning.
7. Single quotes (i.e just a character) do not have or need a \0.
8. 
9. writing printf(“%c”, str[0]) is the same as writing printf(“%c”, \*str)
10. printf(“%c”, str[2]) == printf(“%c”, \*(str + 2))
11. 
12. The above lines of code would cause a SEG FAULT. This is because when we use the char\* method to initialize a string, the string is stored in a Read-only section of the memory.
13. Also, in the above example, both the variables point to the same address in memory. So when we write str1 == str2, we actually get TRUE. However, if instead of directly hardcoding the initialization part, if we tried to accept user input for both as “Hi!”, then they would point to different sections in read only memory.
14. To be able to modify our string, we would have to write it as:
15. 
16. In the above case, the strings are stored in modifiable sections of memory.
17. the printf function ultimately uses the write function underneath it. Basically, it takes in the output to be printed and the format it is in, then it runs a few checks to make sure that your input was valid, then it calls the write function after performing the necessary conversions.
18. 
19. In the above example, when we declare and initialize a string with the syntax mentioned in line 20, then the name of the variable is just an alias of the address the first character is stored at.
20. That is, the above is the same thing as declaring an int or declaring a char.
21. However, if we declare a string as in line 27, then we get a pointer named d, which is located at a particular address. And this pointer d stores another address, which is where the string is actually stored.
22. Suppose we declare a char\* of length 4 as follows:
23. char\* string[4];
24. This string has the ability to store three characters and a \0
25. However, if we were to use scanf to accept user input into this string, and the user were to enter a string bigger than size 4 like "Hello There", then the first four characters would be stored side by side (Since we told the computer hey I need a section of side by side memory that is of size 4) and the remaining would be stored in random locations of memory.

Question: Why can't I store an integer address to a pointer of type char\* even though all pointers are meant to store addresses and all addresses take up 8 bytes of space to store in C?

Answer: The type of a pointer affects how the compiler interprets the data at the memory address it points to. For instance, dereferencing a char\* reads a single byte, while dereferencing an int\* reads multiple bytes (typically 4 or 8 bytes).

In summary, while all pointers store memory addresses and have the same size on a given architecture, the type of the pointer dictates how the compiler and runtime system interpret the data at the address.

malloc:

1. In order to copy one string onto another, we would have to use malloc.
2. malloc and free are two of the most important functions in C.
3. They are found in the stdlib header file.
4. When we type strlen(string name), we actually get the length of the string only, and it doesn't include the '\0'
5. If we do not NULL terminate our created string, then we can corrupt our own data.
6. the return type of the malloc function is an address (or NULL, depending on memory availability). So, we can only use it with pointers.
7. We should always error handle the malloc function, just in case it could not find enough memory.
8. Example: char\* p = malloc(sizeof(char) \* 8);
   1. if (p == NULL)
   2. {
   3. return 1;
   4. }

free:

1. free is the opposite of malloc. It frees up the memory that you used.
2. Syntax : char \*p;
   1. p = malloc(sizeof(char) \* 10);
   2. (perform operations);
   3. free(p);

NULL:

1. It just points to the address zero.
2. If you try to print the address of NULL, you get nil or NULL or 0.
3. This should not be confused with NUL, which is the terminating symbol.

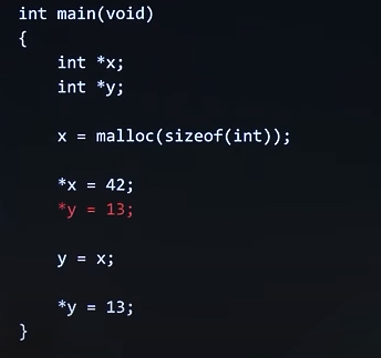
Note: sizeof is an operator

valgrind:

1. Just a program to check your programs memory usage.
2. Syntax = valgrind ./outputfilename

- A memory leak is when you keep calling malloc without freeing any of the memory. This leads the computer to believe that certain memory locations are being used, when in reality, their usage is over. So when you repeatedly keep calling malloc without freeing up memory, you are effectively using more and more memory, which in turn would slow down the computer/program.

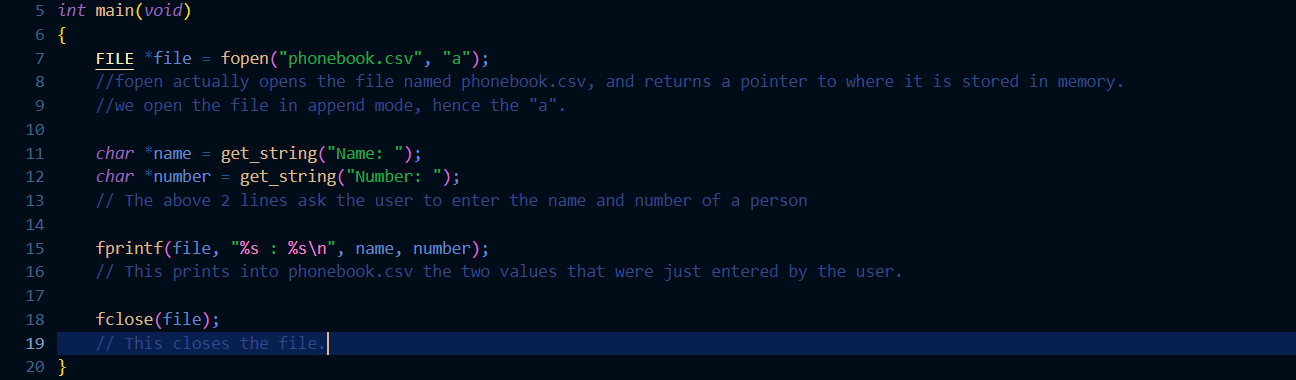
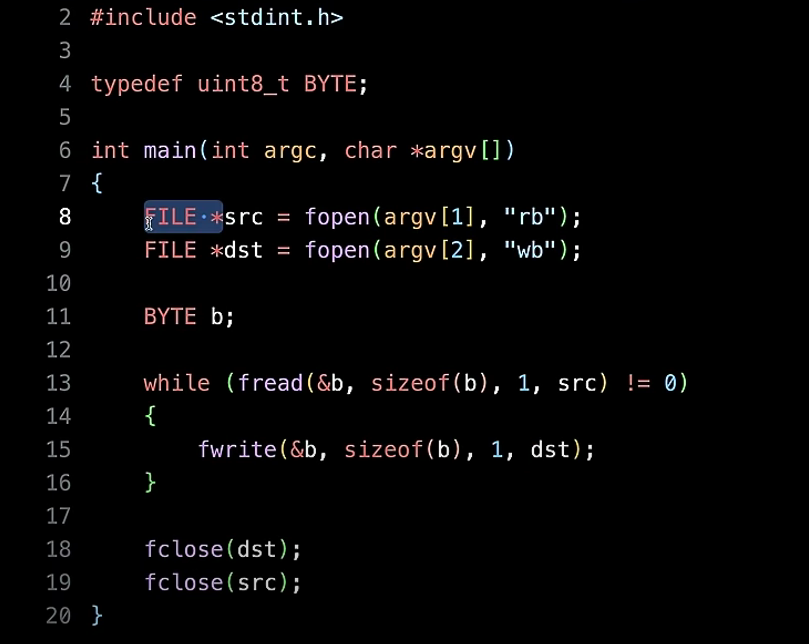
Garbage values:

1. Values of variables that you did not proactively set yourself, i.e, a remnant of a previous program.
2. When we pass a variable to a function, it creates a copy of that variable at a different memory address. This is called passing by value.
3. memory allocated using malloc comes from heap memory.
4. memory allocated to variables during program execution comes from the stack memory.
5. When we use too much heap memory, we could face heap overflow. The same situation happens for stack memory as well.
6. The above two are examples of buffer overflow.
7. 
8. 
9. In the above example, we do not initialize the pointer y with a value. So y has a number stored in it, which could be 0 (NULL), could be negative or could be positive.
10. If in case the value at y is positive, then \*y = 13 would work completely fine.
11. However, in case the value of y is 0 or negative, then the program would crash, since we’re accessing memory that we shouldn’t or that doesn’t exist.

Extra notes:

1. Passing by value is also called passing by copy.
2. Segmentation faults occur when we touch areas of memory that we did not ask the operating system for earlier.

File I/O:

1. Some common functions when it comes to file handling include fopen(), fclose(), fprintf(), fscanf(), fread(), fwrite(), fseek()
2. These allow us to manipulate files on a computer’s hard drive.
3. 
4. 
5. We use fprintf to write formatted strings into our files.
6. We use fwrite to literally write byte by byte into our file. This is what we would use for audio input or image input, where we deal with raw data.