What does malloc do?

allocates some memory

What is the argument that goes into malloc?

size_t size } int how many bytes you want to allocate
type value

What does free do?

lets go of the memory

What is the argument that goes into free?

free(*ptr)
↑pointer

→ malloc not as much memory as printing
→ print garbage values

→ Segmentation Fault
→ using memory that you haven't allocated

CS50 Section 4

Hexadecimal

In writing, we can also indicate a value is in hexadecimal by prefixing it with 0x, as in 0x10, where the value is equal to 16 in decimal, as opposed to 10.

Recover

- matching headers in hexadecimal
  - → use 0x!

Pointers

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

int main(void)
{
    // Get two strings
    string s = get_string("s: ");
    string t = get_string("t: ");

    // Compare strings' addresses
    if (s == t)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

s = "hi"
t = "bye"
} different

s = "hi"
t = "hi"
} different

What does this return and why?

whenever C compares 2 strings, they are comparing memory addresses

* s's pointer is at s[0]

Pointers

We can see the address with the & operator,

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

What would this look like? 0x.... Garbage :-

memory address

Pointers

The * operator lets us “go to” the location that a pointer is pointing to.

What does this do? print 50

```c
#include <stdio.h>

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

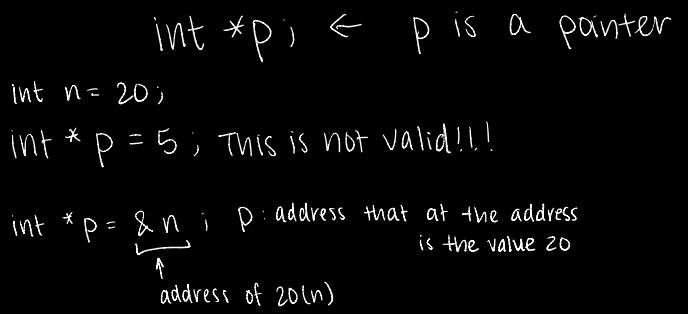
*&n is equiv. n

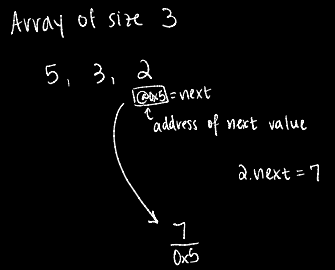
pointer

value at the pointer

Pointers

How do we declare a variable that we want to be a pointer?





Pointers

How do we declare a variable that we want to be a pointer?

```c
#include <stdio.h>

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

address

prints the address

printf("%i\n", *p); prints 50

value

We use int *p to declare a variable, p, that has the type of *, a pointer, to a value of type int, an integer. What would this print?

Strings

How are strings stored in memory?

memory address corresponding to s[0]

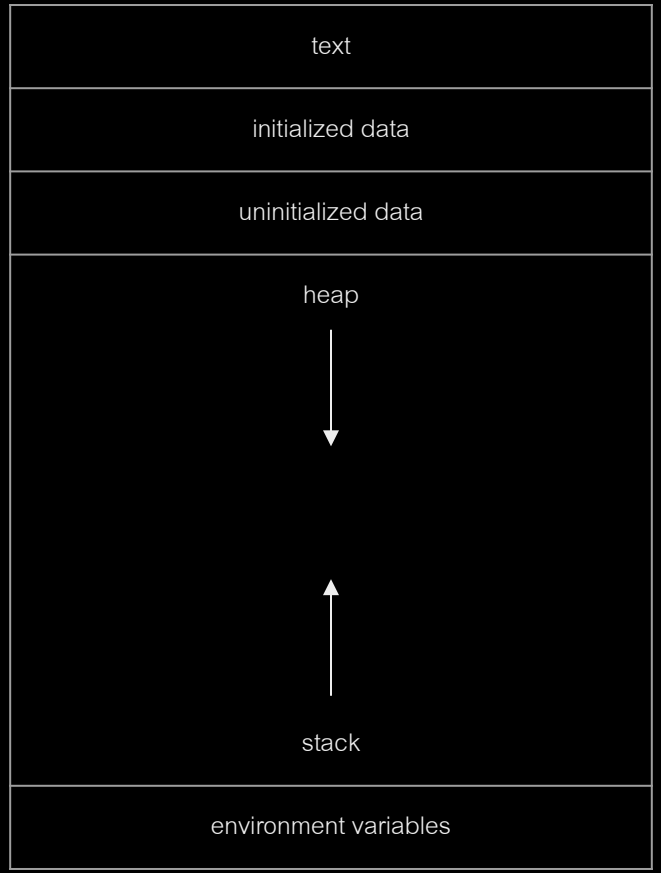
Dynamic Memory Allocation

- We know one way to use pointers -- connecting a pointer variable by pointing it at another variable that already exists in our program.

```
int *p = &n
```

- But what if we don't know in advance how much memory we'll need at compile time? How do we access more memory at runtime?

- Pointers can also be used to do this. Memory allocated *dynamically* (at runtime) comes from a pool of memory called the heap. Memory allocated at compile time typically comes from a pool of memory called the stack.


text
initialized data
uninitialized data
heap
stack
environment variables


text
initialized data
uninitialized data
heap
stack
environment variables

- We get this dynamically-allocated memory via a call to the function malloc(), passing as its parameter the number of *bytes* we want. malloc() will return to you a **pointer** to that newly-allocated memory.
  - If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer.

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


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

pointer to newly allocated memory

- We get this dynamically-allocated memory via a call to the function malloc(), passing as its parameter the number of *bytes* we want. malloc() will return to you a **pointer** to that newly-allocated memory.
  - If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer.

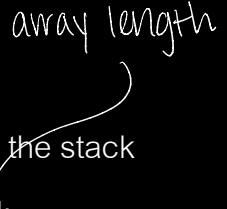
```
// Statically obtain an integer
int x;


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

```
// Get an integer from the user
int x = get_int();

// Array of floats on the stack
float stack_array[x];

// Array of floats on the heap
float *heap_array = malloc(x * sizeof(float));
```



- There's a catch: Dynamically allocated memory is not automatically returned to the system for later use when no longer needed.

- Failing to return memory back to the system when you no longer need it results in a memory leak, which compromises your system's performance.

- All memory that is dynamically allocated must be released back by free()-ing its pointer.

→ if you malloc,
run valgrind

allocating memory for a word w/ 50 chars

```
char *word = malloc(50 * sizeof(char));


// do stuff with word


// now we're done
free(word);
```

- Every block of memory that you malloc(), you must later free().
- **Only** memory that you obtain with malloc() should you later free().
- Do not free() a block of memory more than once.

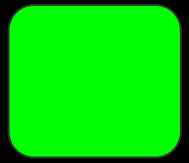
```
int m;
```

m

```
int m;
int *a;
```

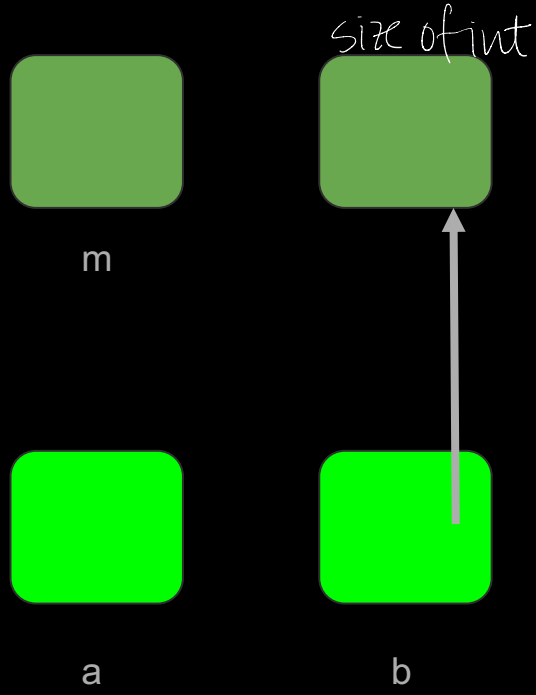
m

pointer!



a

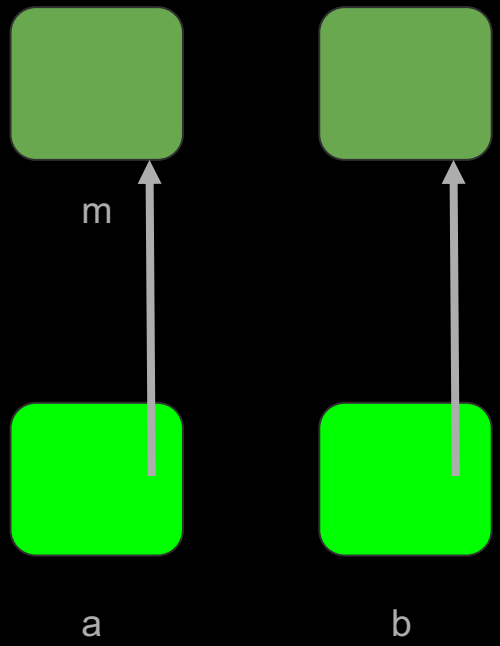
```
int m;
int *a;
int *b = malloc(sizeof(int));
```



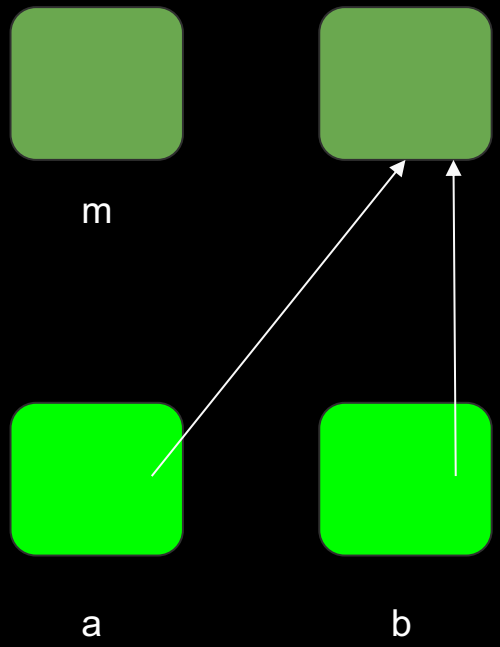
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
```

address

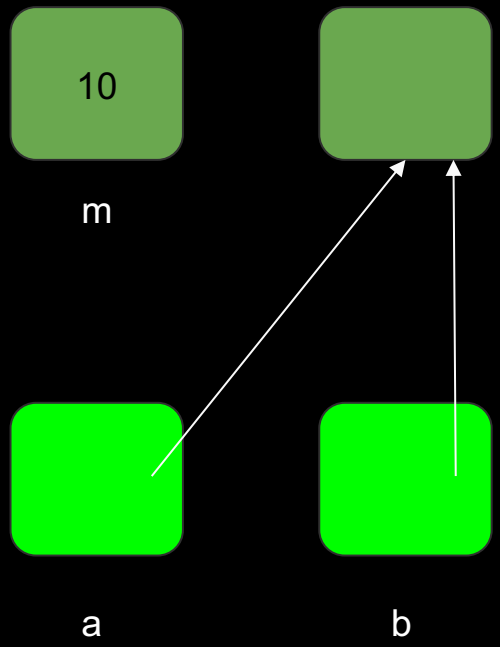
a = address of m



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

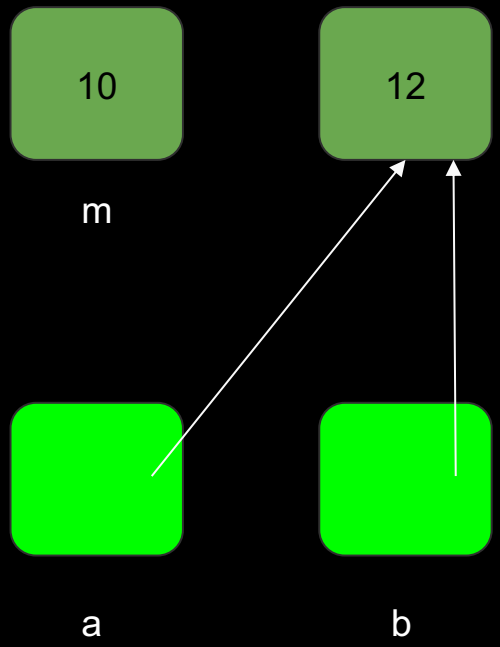


```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
```

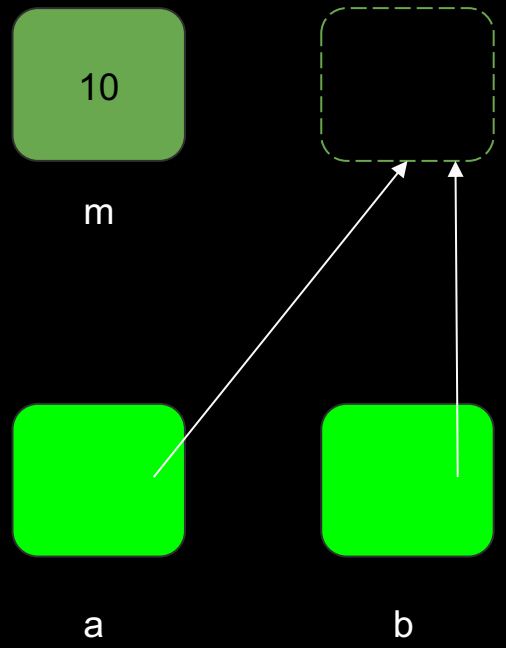


```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
```

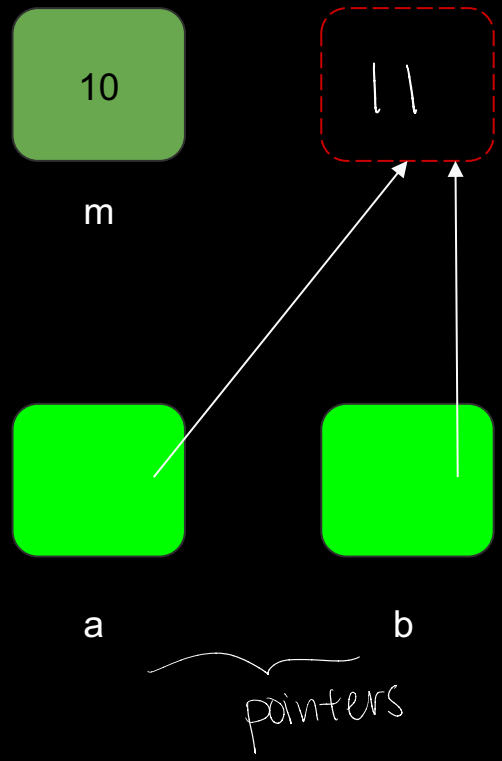
value at b = m+2 = 12



```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(a);
```



```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
free(a);
*b = 11;
```



What happens if we do not free memory that we have allocated?

memory leak

Make sure to run valgrind on your code to ensure you don't have any memory leaks.

What happens if we malloc too many times?

run out of heap space

What happens if we call functions too many times?

run out of stack space

File I/O

1. fopen : what it does and arguments and return
2. fread :
3. funte :
4. fclose :

Fopen: opens a file, returns a pointer

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

mode: "r" "w" "a"

TYPE — pointer to the file

Fread: reads a file, returns # elements read

```
int val;
fread(&val, sizeof(int), 1, input);
```

- &val: pointer to some place to store the values read
- sizeof(int): size of each sample
- 1: the # of samples you're reading
- input: file reading from

input has already read to end

```
int hi = fread(&val, sizeof(int), 1, input);
```

hi = 0

x → true : do not cont!

x → false : would execute

```
if (x==0) ⇔ if (!x)
while(fread(___) != 0) {
}
```

```
while(fread(_____)) {
    // this program will execute as long as
    fread(_____) does not return 0
}
```

fread(___) → = 0 don't execute; ≠ 0 execute

while fread ≠ 0 ⇔ while fread

- The ability to read data from and write data to files is the primary means of storing **persistent data**, which exists outside of your program.

- In C, files are abstracted using a data structure called a FILE. Almost universally, though, when working with FILEs do we actually use pointers to files (aka FILE *).

- The functions we use to manipulate files all are found in **stdio.h**.

- Every one of them accepts a FILE * as one of its parameters, except fopen() which is used to get a file pointer in the first place.

- Some of the most common file input/output (I/O) functions we'll use are the following:

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

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

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

```
File * input = fopen("input.txt", "r");
if (input == NULL)
{
    printf("couldn't open file!");
    return 1;
}
```

- fopen() opens a file and returns a pointer to it. Always check its return value to make sure you don’t get back NULL.

```
FILE *ptr = fopen("test.txt", "r");

FILE *ptr2 = fopen("test2.txt", "w");

FILE *ptr3 = fopen("test3.txt", "a");
```

- fgetc() reads and returns the next character from the file, assuming the operation for that file contains "r". fputc() writes or appends the specified character to the file, assuming the operation for that pointer contains "w" or "a".

fgetc(<file pointer>);

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

- fgetc() reads and returns the next character from the file, assuming the operation for that file contains "r". fputc() writes or appends the specified character to the file, assuming the operation for that pointer contains "w" or "a".

```
char c = fgetc(ptr1);

fputc('x', ptr2);

fputc('5', ptr3);
```

- fread() and fwrite() are analogs to fgetc() and fputc(), but for a generalized quantity (qty) of blocks of an arbitrary (size), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.

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

fwrite(<buffer>, <size>, <qty>, <file pointer>);

after reading or writing, shifts the pointer by the amount you read/wrote

- fread() and fwrite() are analogs to fgetc() and fputc(), but for a generalized quantity (qty) of blocks of an arbitrary (size), holding those blocks in (or writing them from) a temporary buffer, usually an array, for local use within the program.

```
int arr[10];
fread(arr, sizeof(int), 10, ptr);
fwrite(arr, sizeof(int), 10, ptr2);
fwrite(arr, sizeof(int), 10, ptr3);
```

- fclose() closes a previously opened file pointer.

fclose(<file pointer>);

- fclose() closes a previously opened file pointer.

```
fclose(ptr);

fclose(ptr2);

fclose(ptr3);
```

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

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 to 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.

JPEG Files

different types of files have different headers

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    // Check usage
    if (argc != 2)
    {
        return 1;
    }

    // Open file
    FILE *file = fopen(argv[1], "r");
    if (!file)
    {
        return 1;
    }

    // Read first three bytes
    unsigned char bytes[3];
    fread(bytes, 3, 1, file);

    // Check first three bytes
    if (bytes[0] == 0xff && bytes[1] == 0xd8 && bytes[2] == 0xff)
    {
        printf("Maybe\n");
    }
    else
    {
        printf("No\n");
    }

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

define HEADER 0xff; } TOP OF PROGRAM!

if bytes[0] == HEADER

Exercise

In copy.c, write a program that copies a text file. Users should be able to run ./copy file1 file2 to copy the contents of text file file1 into file file2.

- ☆ check input length
  - → argc?
- ☆ fopen! → one should be "r"
  - one should be "w"
- ☆ copy over things!
  - fgetc and fputc

```
char c = fgetc(___);
if  c == EOF
{
    break;
}
```

- ☆ close both files

```
#include <stdbool.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc != 3)
    {
        printf("Bad");
        return 1;
    }
    FILE *infile = fopen(argv[1],"r");
    FILE *outfile = fopen(argv[2],"w");
    if (infile == NULL || outfile == NULL)
    {
        printf("Bad File(s)!");
        return 1;
    }
    while(true)
    {
        char c = fgetc(infile);
        if (c == EOF)
        {
            break;
        }
        fputc(c,outfile);
    }
    fclose(infile);
    fclose(outfile);
}
```

Lab

WAV: 44 byte header
data → int16_t

1. uint8_t header[44] ← HEADER SIZE ← array
copy header over
fread: header (array), HEADERSIZE, quant. 1, input

2. float factor

int16_t buffer;
while(fread(_____))
{
change buffer by factor
fwrite onto new file!
}

3. close the files

./volume input.wav output.wav Factor
0.5
2.0

HEADER {
uint8_t header[HEADER_SIZE]; ← array is called header
fread(header, HEADER_SIZE, 1, input);
fwrite _____;

int16_t buffer; ← address
while(fread(&buffer, sizeof(int16_t), 1, input))
{
change buffer by factor
fwrite _____
}

CS50 Section 4