What does malloc do?

What is the argument that goes into malloc?

What does free do?

What is the argument that goes into free?

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.

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");
    }
}
```

What does this return and why?

Pointers

We can see the address with the & operator,

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

What would this look like?

Pointers

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

What does this do?

```
#include <stdio.h>

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

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?

```
#include <stdio.h>

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

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

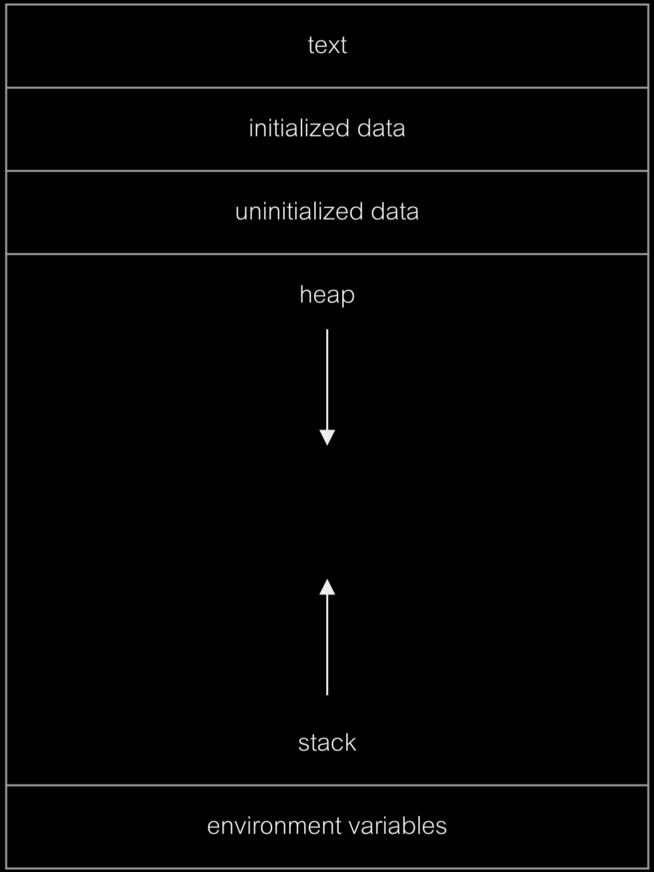
How are strings stored in memory?

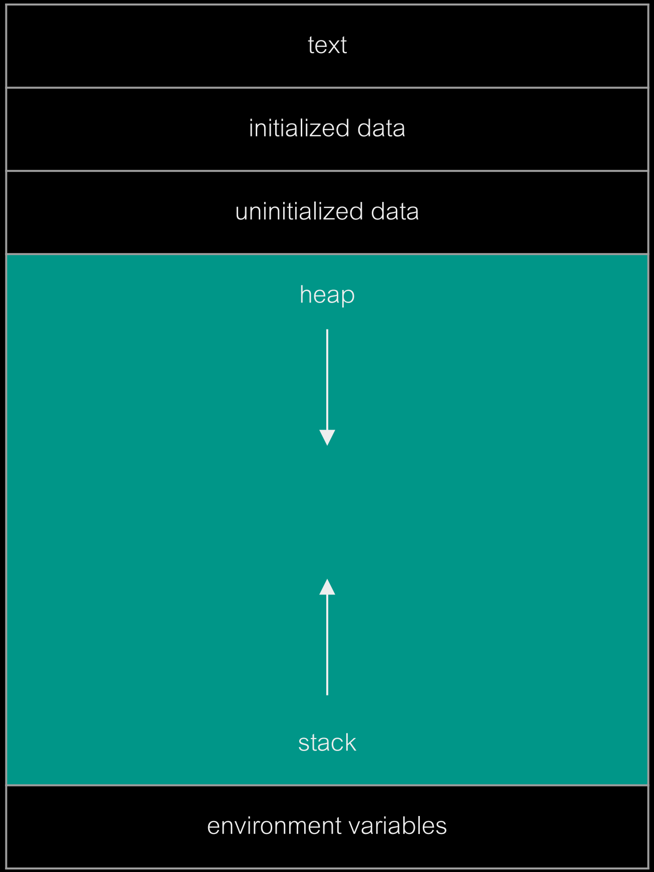
Dynamic Memory Allocation

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

- 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);
```

- 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.

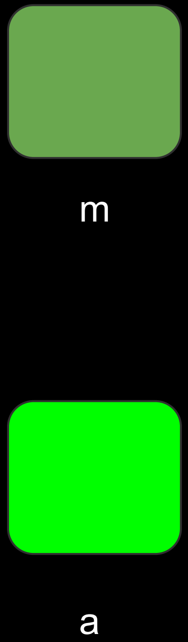
```
char *word = malloc(50 * sizeof(char));

// 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.

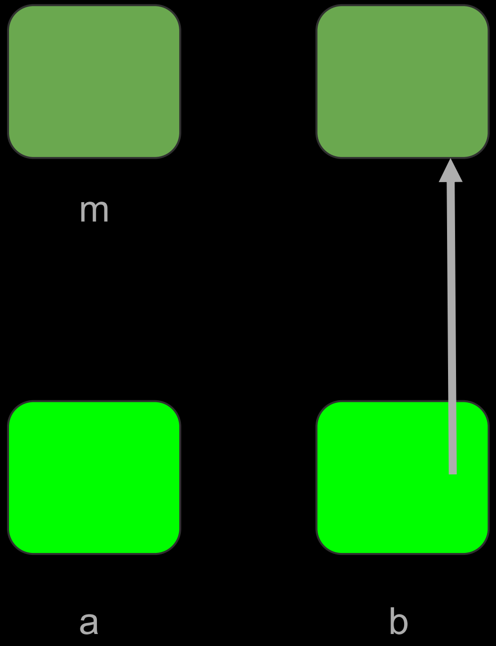
```
int m;
```

m

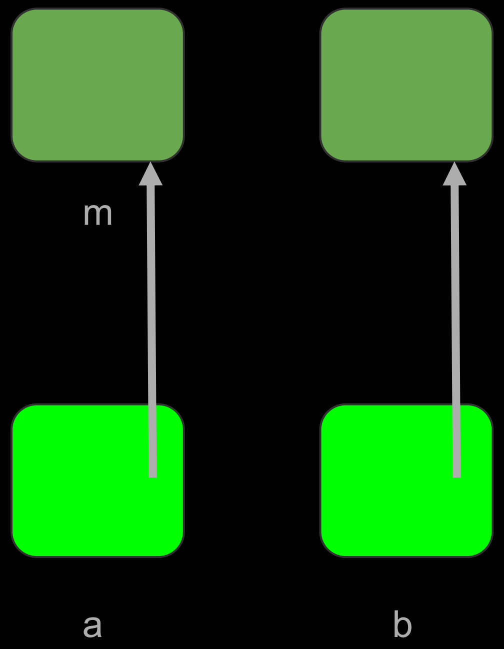
```
int m;
int *a;
```



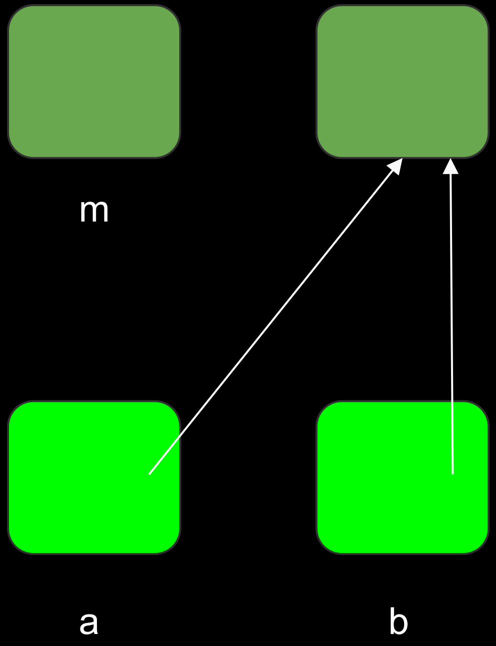
```
int m;
int *a;
int *b = malloc(sizeof(int));
```



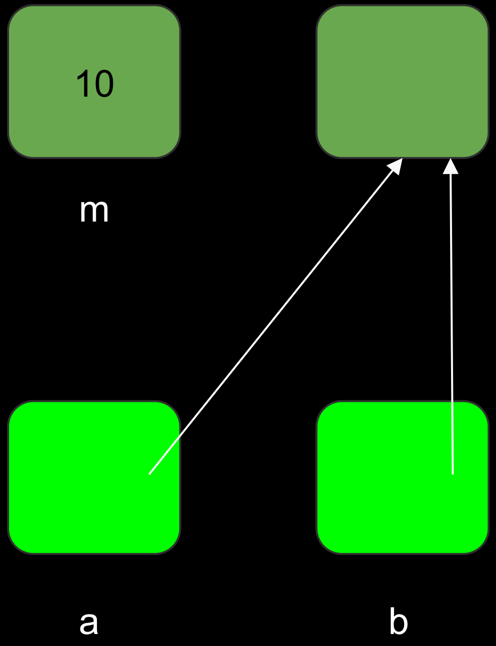
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
```



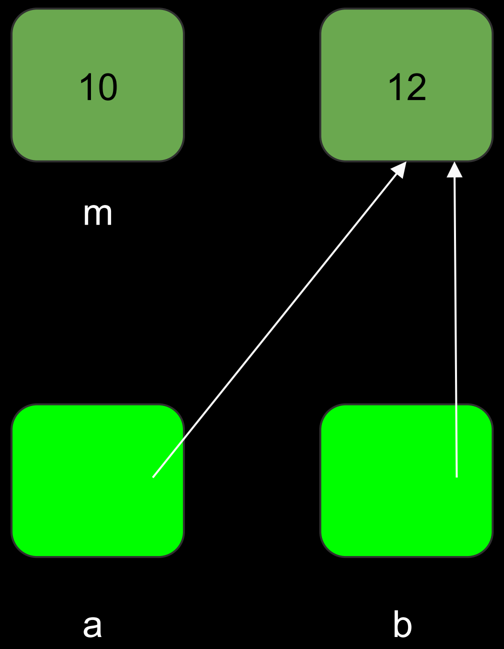
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
```



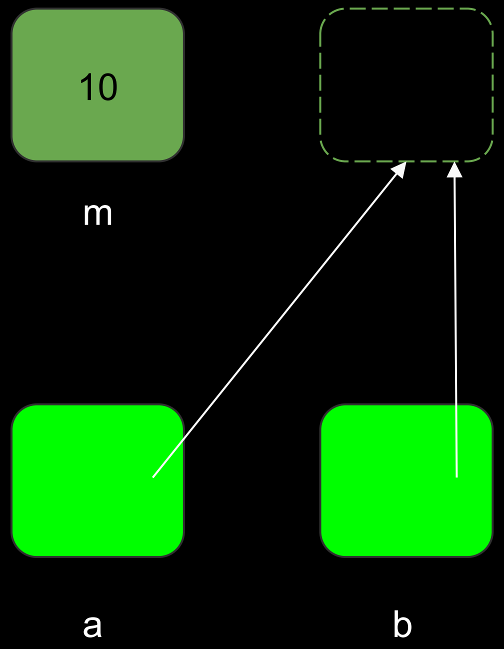
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
```



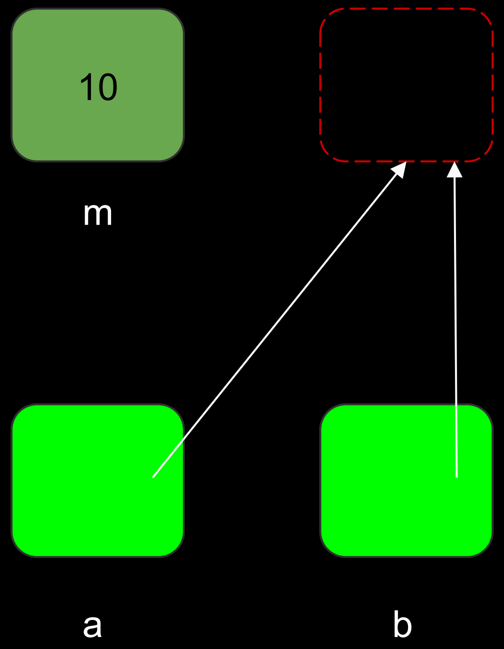
```
int m;
int *a;
int *b = malloc(sizeof(int));
a = &m;
a = b;
m = 10;
*b = m + 2;
```



```
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?

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?

What happens if we call functions too many times?

File I/O

- 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>);
```

- 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>);

- 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

```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);
}
```

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.

Lab

CS50 Section 4