


This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

Donate  (<https://cs50.harvard.edu/donate>)

David J. Malan (<https://cs.harvard.edu/malan/>)

malan@harvard.edu

 (<https://www.facebook.com/dmalan>)  (<https://github.com/dmalan>) 

(<https://www.instagram.com/davidjmalan/>)  ([https://www.linkedin.com](https://www.linkedin.com/in/malan/)

[in/malan/](https://www.linkedin.com/in/malan/))  (<https://www.reddit.com/user/davidjmalan>)  @

(<https://www.threads.net/@davidjmalan>)  (<https://twitter.com/davidjmalan>)

Lecture 4

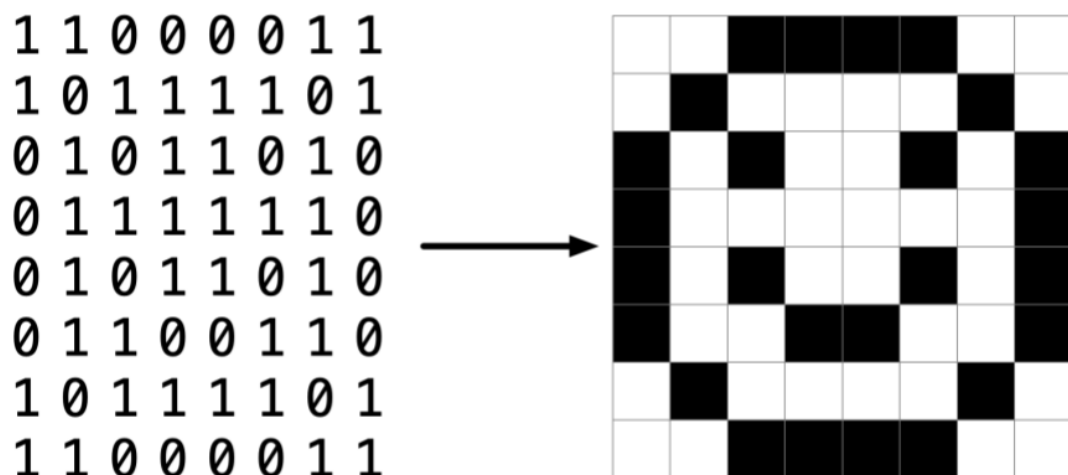
- [Welcome!](#)
- [Pixel Art](#)
- [Hexadecimal](#)
- [Memory](#)
- [Pointers](#)
- [Strings](#)
- [Pointer Arithmetic](#)
- [String Comparison](#)
- [Copying](#)
- [malloc and Valgrind](#)
- [Garbage Values](#)
- [Pointer Fun with Binky](#)
- [Swap](#)
- [Overflow](#)
- `scanf`
- [File I/O](#)
- [Summing Up](#)

Welcome!

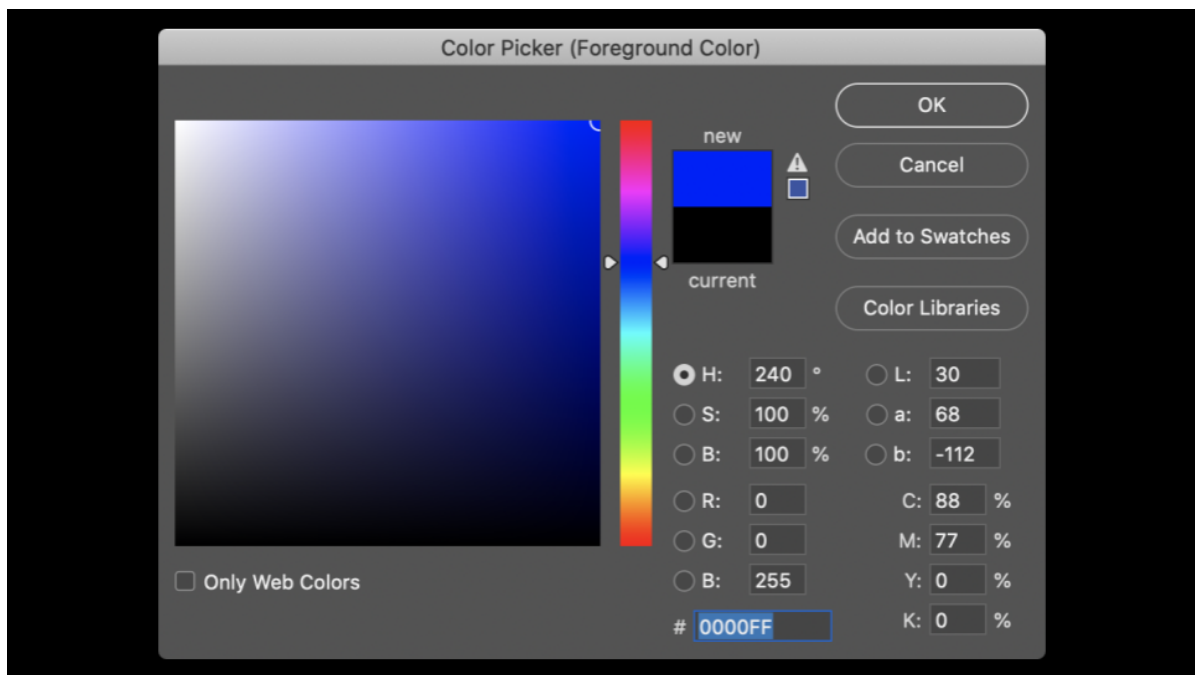
- In previous weeks, we talked about images being made of smaller building blocks called pixels.
- Today, we will go into further detail about the zeros and ones that make up these images. In particular, we will be going deeper into the fundamental building blocks that make up files, including images.
- Further, we will discuss how to access the underlying data stored in computer memory.

Pixel Art

- Pixels are squares, individual dots, of color that are arranged on an up-down, left-right grid.
- You can imagine as an image as a map of bits, where zeros represent black and ones represent white.



- *RGB*, or *red*, *green*, *blue*, are numbers that represent the amount of each of these colors. In Adobe Photoshop, you can see these settings as follows:



Notice how the amount of red, blue, and green changes the color selected.

- You can see by the image above that color is not just represented in three values. At the bottom of the window, there is a special value made up of numbers and characters. `255` is represented as `FF`. Why might this be?

Hexadecimal

- Hexadecimal* is a system of counting that has 16 counting values. They are as follows:

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

Notice that `F` represents `15`.

- Hexadecimal is also known as *base-16*.
- When counting in hexadecimal, each column is a power of 16.
- The number `0` is represented as `00`.
- The number `1` is represented as `01`.
- The number `9` is represented by `09`.
- The number `10` is represented as `0A`.
- The number `15` is represented as `0F`.
- The number `16` is represented as `10`.
- The number `255` is represented as `FF`, because 16×15 (or `F`) is 240. Add 15 more to make 255. This is the highest number you can count using a two-digit hexadecimal system.
- Hexadecimal is useful because it can be represented using fewer digits. Hexadecimal allows us to represent information more succinctly.

Memory

- In weeks past, you may recall our artist rendering of concurrent blocks of memory. Applying hexadecimal numbering to each of these blocks of memory, you can visualize these as follows:

0	1	2	3	4	5	6	7
8	9	A	B	C	D	E	F
10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F

- You can imagine how there may be confusion regarding whether the `10` block above may represent a location in memory or the value `10`. Accordingly, by convention, all hexadecimal numbers are often represented with the `0x` prefix as follows:

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

- In your terminal window, type `code addresses.c` and write your code as follows:

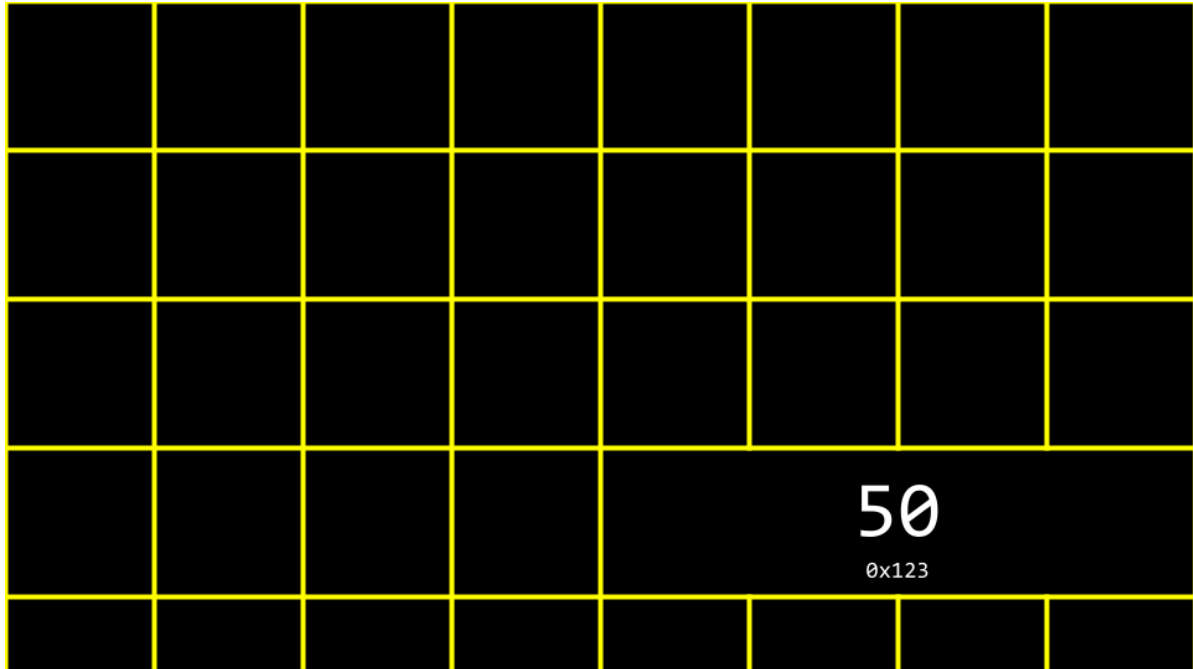
```
#include <stdio.h>

int main(void)
```

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

Notice how `n` is stored in memory with the value `50`.

- You can visualize how this program stores this value as follows:



- The `C` language has two powerful operators that relate to memory:

```
& Provides the address of something stored in memory.  
* Instructs the compiler to go to a location in memory.
```

- We can leverage this knowledge by modifying our code as follows:

```
#include <stdio.h>  
  
int main(void)  
{  
    int n = 50;  
    printf("%p\n", &n);  
}
```

Notice the `%p`, which allows us to view the address of a location in memory. `&n` can be literally translated as “the address of `n`.” Executing this code will return an address of memory beginning with `0x`.

Pointers

- A *pointer* is a variable that contains the address of some value. Most succinctly, a pointer is an address in your computer’s memory.
- Consider the following code:

```
int n = 50;

int *p = &n;
```

Notice that `p` is a pointer that contains the address of an integer `n`.

- Modify your code as follows:

```
#include <stdio.h>

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

Notice that this code has the same effect as our previous code. We have simply leveraged our new knowledge of the `&` and `*` operators.

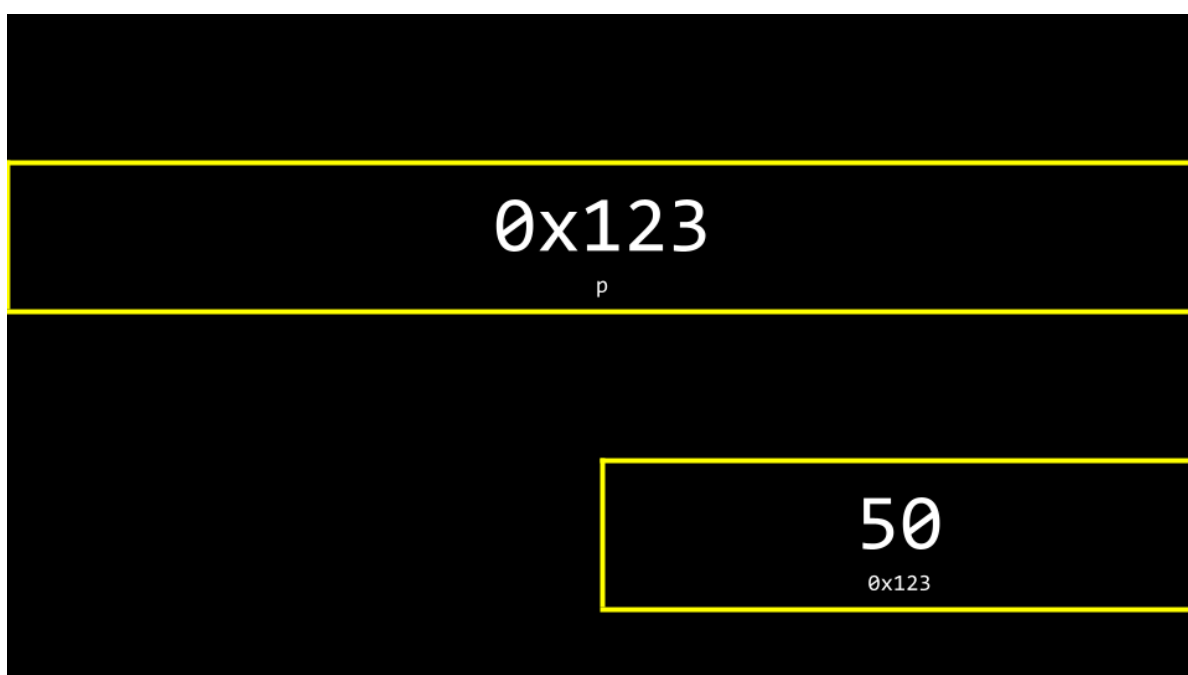
- To illustrate the use of the `*` operator, consider the following:

```
#include <stdio.h>

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

Notice that the `printf` line prints the integer at the location of `p`. `int *p` creates a pointer whose job is to store the memory address of an integer.

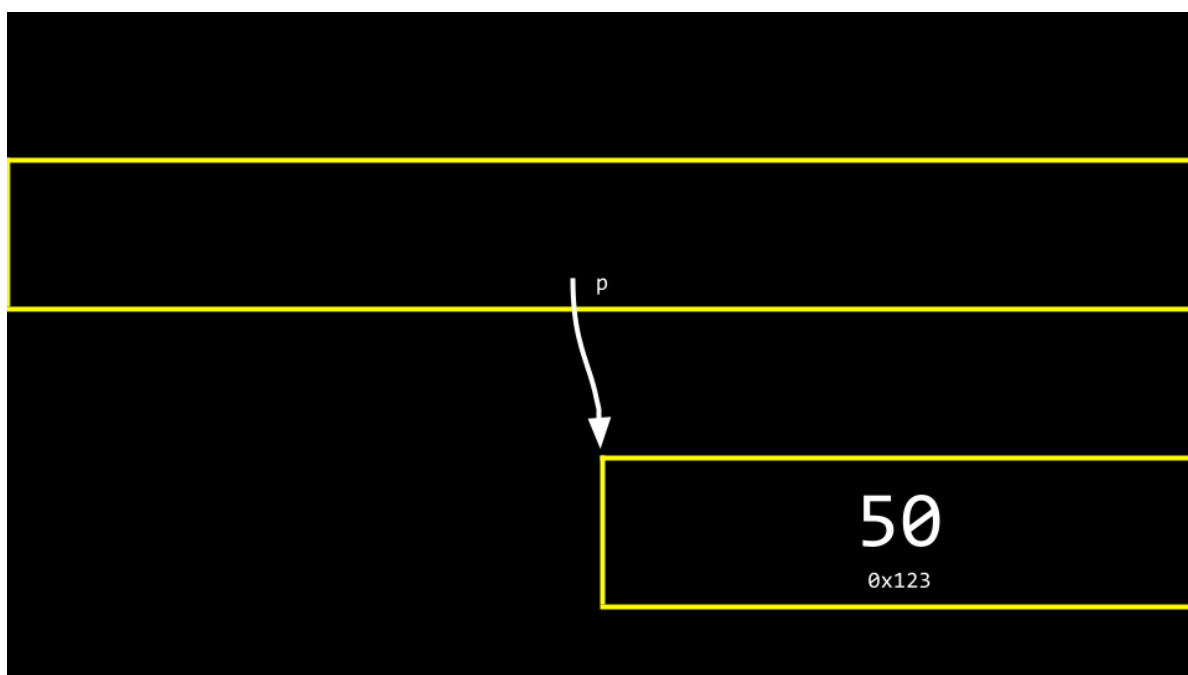
- You can visualize our code as follows:



Notice the pointer seems rather large. Indeed, a pointer is usually stored as an 8-byte

value. `p` is storing the address of the `50`.

- You can more accurately visualize a pointer as one address that points to another:

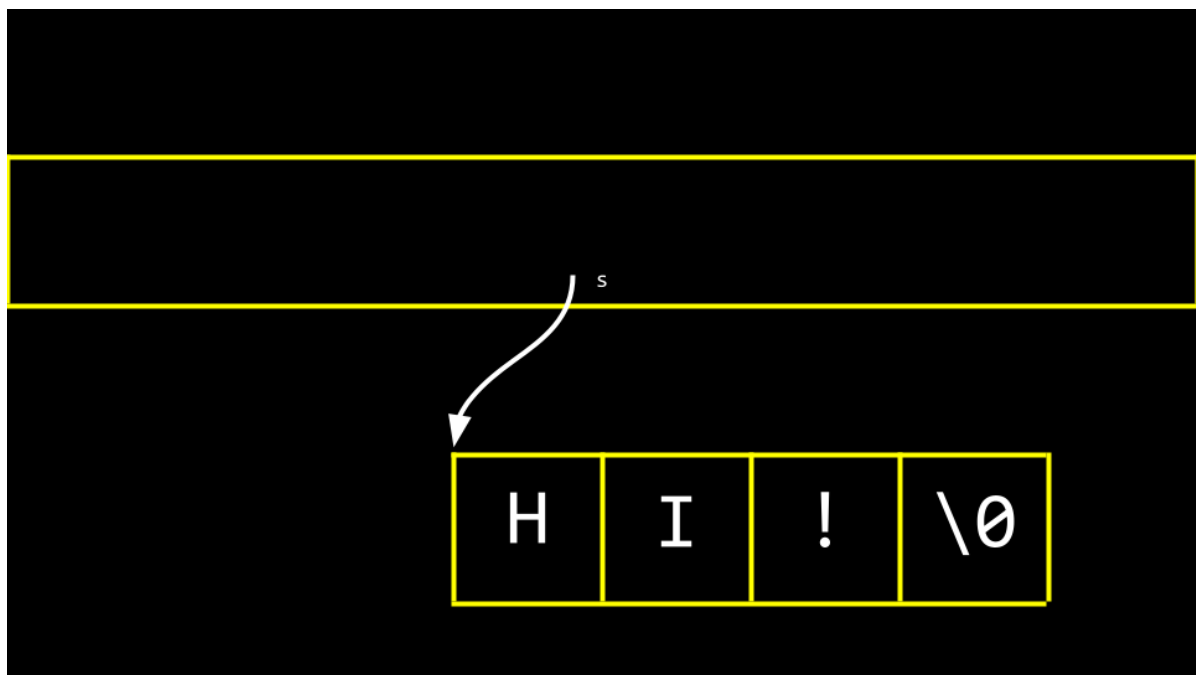


Strings

- Now that we have a mental model for pointers, we can peel back a level of simplification that was offered earlier in this course.
- Recall that a string is simply an array of characters. For example, `string s = "HI!"` can be represented as follows:



- However, what is `s` really? Where is the `s` stored in memory? As you can imagine, `s` needs to be stored somewhere. You can visualize the relationship of `s` to the string as follows:



Notice how a pointer called `s` tells the compiler where the first byte of the string exists in memory.

- Modify your code as follows:

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

int main(void)
{
    string s = "HI!";
    printf("%p\n", s);
    printf("%p\n", &s[0]);
    printf("%p\n", &s[1]);
    printf("%p\n", &s[2]);
    printf("%p\n", &s[3]);
}
```

Notice the above prints the memory locations of each character in the string `s`. The `&` symbol is used to show the address of each element of the string. When running this code, notice that elements `0`, `1`, `2`, and `3` are next to one another in memory.

- Likewise, you can modify your code as follows:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%s\n", s);
}
```

Notice that this code will present the string that starts at the location of `s`. This code effectively removes the training wheels of the `string` data type offered by `cs50.h`. This is raw C code, without the scaffolding of the `cs50` library.

- You can imagine how a string, as a data type, is created.

- Last week, we learned how to create your own data type as a struct.
- The cs50 library includes a struct as follows: `typedef char *string`
- This struct, when using the cs50 library, allows one to use a custom data type called `string`.

Pointer Arithmetic

- You can modify your code to accomplish the same thing in a longer form as follows:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", s[0]);
    printf("%c\n", s[1]);
    printf("%c\n", s[2]);
}
```

Notice that we are printing each character at the location of `s`.

- Further, you can modify your code as follows:

```
#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", *s);
    printf("%c\n", *(s + 1));
    printf("%c\n", *(s + 2));
}
```

Notice that the first character at the location of `s` is printed. Then, the character at the location `s + 1` is printed, and so on.

String Comparison

- A string of characters is simply an array of characters identified by its first byte.
- Earlier in the course, we considered the comparison of integers. We could represent this in code by typing `code compare.c` into the terminal window and writing code as follows:

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

int main(void)
{
    // Get two integers
```

```
int i = get_int("i: ");
int j = get_int("j: ");

// Compare integers
if (i == j)
{
    printf("Same\n");
}
else
{
    printf("Different\n");
}
}
```

Notice that this code takes two integers from the user and compares them.

- In the case of strings, however, one cannot compare two strings using the `==` operator.
- Utilizing the `==` operator in an attempt to compare strings will attempt to compare the memory locations of the strings instead of the characters therein. Accordingly, we recommended the use of `strcmp`.
- To illustrate this, modify your code as follows:

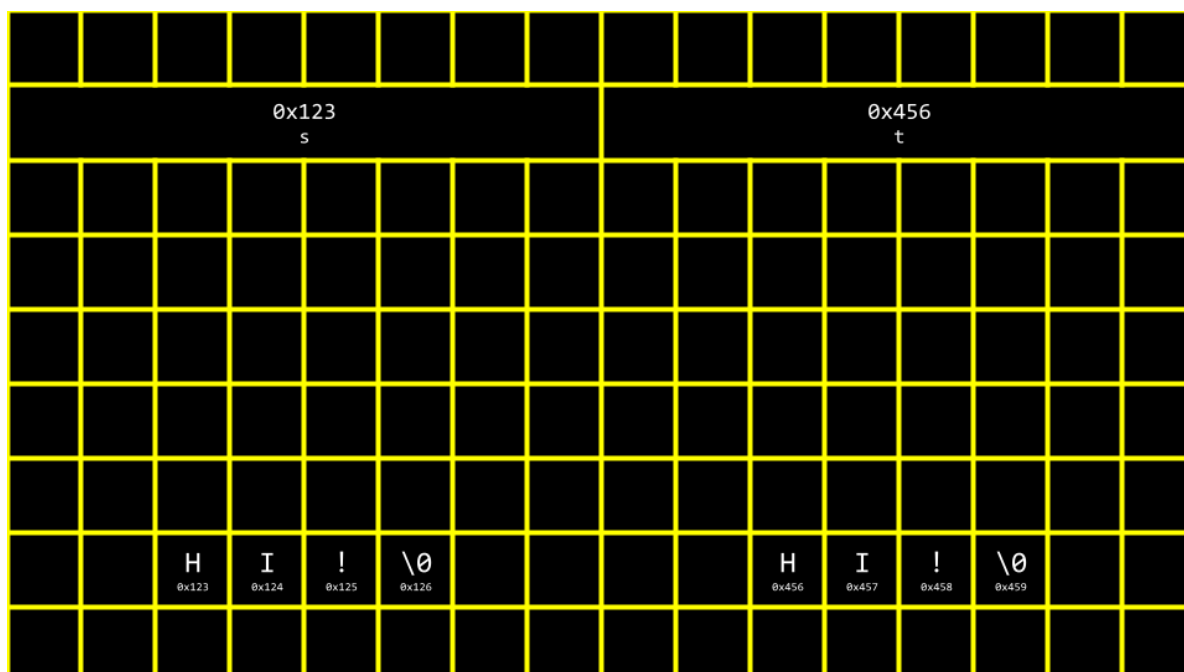
```
#include <cs50.h>
#include <stdio.h>

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

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

Noticing that typing in `HI!` for both strings still results in the output of `Different`.

- Why are these strings seemingly different? You can use the following to visualize why:



- Therefore, the code for `compare.c` above is actually attempting to see if the memory addresses are different: not the strings themselves.
- Using `strcmp`, we can correct our code:

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

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

    // Compare strings
    if (strcmp(s, t) == 0)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}
```

Notice that `strcmp` can return `0` if the strings are the same.

- To further illustrate how these two strings are living in two locations, modify your code as follows:

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

int main(void)
{
    // Get two strings
    char *s = get_string("s: ");
```

```
char *t = get_string("t: ");

// Print strings
printf("%s\n", s);
printf("%s\n", t);
}
```

Notice how we now have two separate strings stored likely at two separate locations.

- You can see the locations of these two stored strings with a small modification:

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

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

    // Print strings' addresses
    printf("%p\n", s);
    printf("%p\n", t);
}
```

Notice that the `%s` has been changed to `%p` in the print statement.

Copying

- A common need in programming is to copy one string to another.
- In your terminal window, type `code copy.c` and write code as follows:

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

int main(void)
{
    // Get a string
    string s = get_string("s: ");

    // Copy string's address
    string t = s;

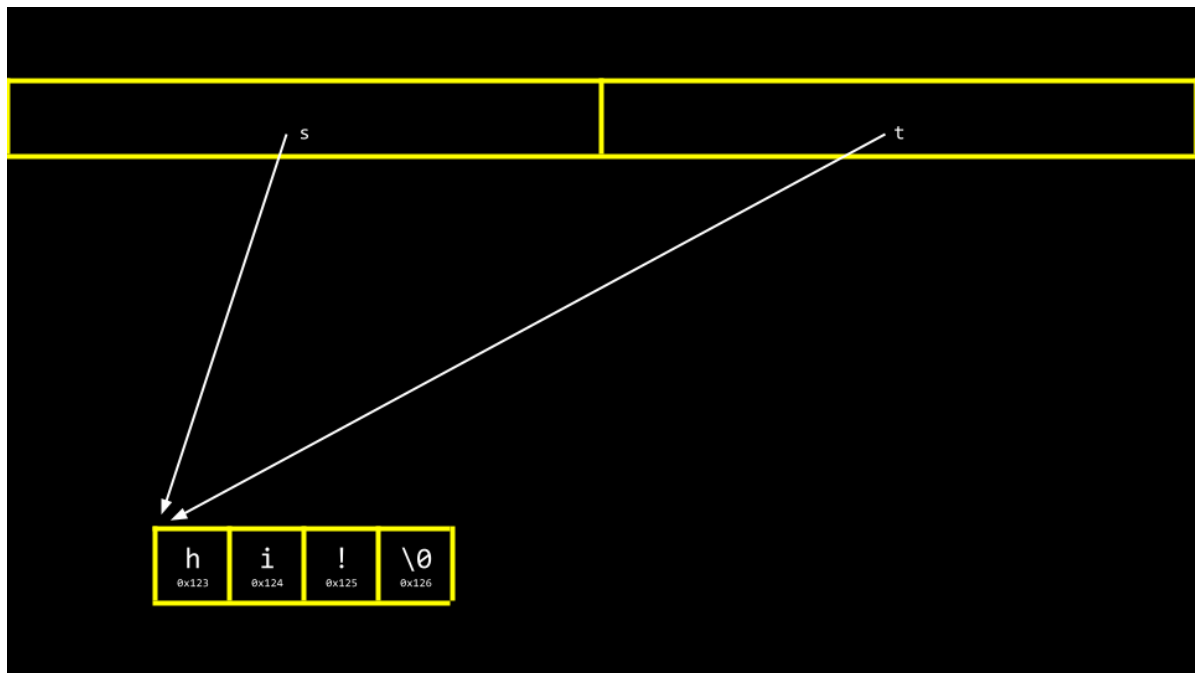
    // Capitalize first letter in string
    t[0] = toupper(t[0]);

    // Print string twice
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Notice that `string t = s` copies the address of `s` to `t`. This does not accomplish

what we are desiring. The string is not copied – only the address is.

- You can visualize the above code as follows:



Notice that `s` and `t` are still pointing at the same blocks of memory. This is not an authentic copy of a string. Instead, these are two pointers pointing at the same string.

- Before we address this challenge, it's important to ensure that we don't experience a *segmentation fault* through our code, where we attempt to copy `string s` to `string t`, where `string t` does not exist. We can employ the `strlen` function as follows to assist with that:

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

int main(void)
{
    // Get a string
    string s = get_string("s: ");

    // Copy string's address
    string t = s;

    // Capitalize first letter in string
    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    // Print string twice
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Notice that `strlen` is used to make sure `string t` exists. If it does not, nothing will

be copied.

- To be able to make an authentic copy of the string, we will need to introduce two new building blocks. First, `malloc` allows you, the programmer, to allocate a block of a specific size of memory. Second, `free` allows you to tell the compiler to *free up* that block of memory you previously allocated.
- We can modify our code to create an authentic copy of our string as follows:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);

    // Copy string into memory, including '\0'
    for (int i = 0; i <= strlen(s); i++)
    {
        t[i] = s[i];
    }

    // Capitalize copy
    t[0] = toupper(t[0]);

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Notice that `malloc(strlen(s) + 1)` creates a block of memory that is the length of the string `s` plus one. This allows for the inclusion of the *null* `\0` character in our final, copied string. Then, the `for` loop walks through the string `s` and assigns each value to that same location on the string `t`.

- It turns out that there is an inefficiency in our code. Modify your code as follows:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
```

```
char *t = malloc(strlen(s) + 1);

// Copy string into memory, including '\0'
for (int i = 0, n = strlen(s); i <= n; i++)
{
    t[i] = s[i];
}

// Capitalize copy
t[0] = toupper(t[0]);

// Print strings
printf("s: %s\n", s);
printf("t: %s\n", t);
}
```

Notice that `n = strlen(s)` is defined now in the left-hand side of the `for` loop. It's best not to call unneeded functions in the middle condition of the `for` loop, as it will run over and over again. When moving `n = strlen(s)` to the left-hand side, the function `strlen` only runs once.

- The `C` Language has a built-in function to copy strings called `strcpy`. It can be implemented as follows:

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);

    // Copy string into memory
    strcpy(t, s);

    // Capitalize copy
    t[0] = toupper(t[0]);

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);
}
```

Notice that `strcpy` does the same work that our `for` loop previously did.

- Both `get_string` and `malloc` return `NULL`, a special value in memory, in the event that something goes wrong. You can write code that can check for this `NULL` condition as follows:

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

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    // Get a string
    char *s = get_string("s: ");
    if (s == NULL)
    {
        return 1;
    }

    // Allocate memory for another string
    char *t = malloc(strlen(s) + 1);
    if (t == NULL)
    {
        return 1;
    }

    // Copy string into memory
    strcpy(t, s);

    // Capitalize copy
    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    // Print strings
    printf("s: %s\n", s);
    printf("t: %s\n", t);

    // Free memory
    free(t);
    return 0;
}
```

Notice that if the string obtained is of length `0` or `malloc` fails, `NULL` is returned. Further, notice that `free` lets the computer know you are done with this block of memory you created via `malloc`.

malloc and 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`:

```
#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 as follows:

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

int main(void)
{
    int *x = malloc(3 * sizeof(int));
    x[0] = 72;
    x[1] = 73;
    x[2] = 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 this memory that 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`:

```
#include <stdio.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 \(https://www.youtube.com/watch?v=5VnDaHBi8dM\)](https://www.youtube.com/watch?v=5VnDaHBi8dM) that helped us visualize and understand pointers.

Swap

- In the real world, a common need in programming is to swap two values. Naturally, it's hard to swap two variables without a temporary holding space. In practice, you can type `code swap.c` and write code as follows to see this in action:

```
#include <stdio.h>

void swap(int a, int b);

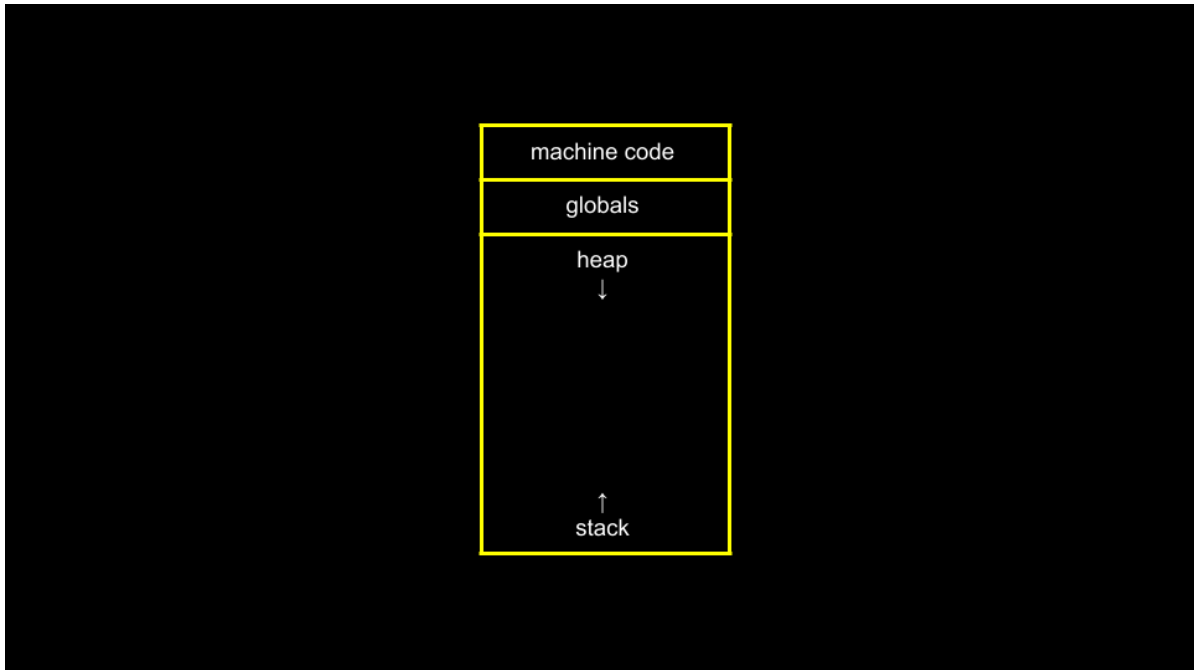
int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

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

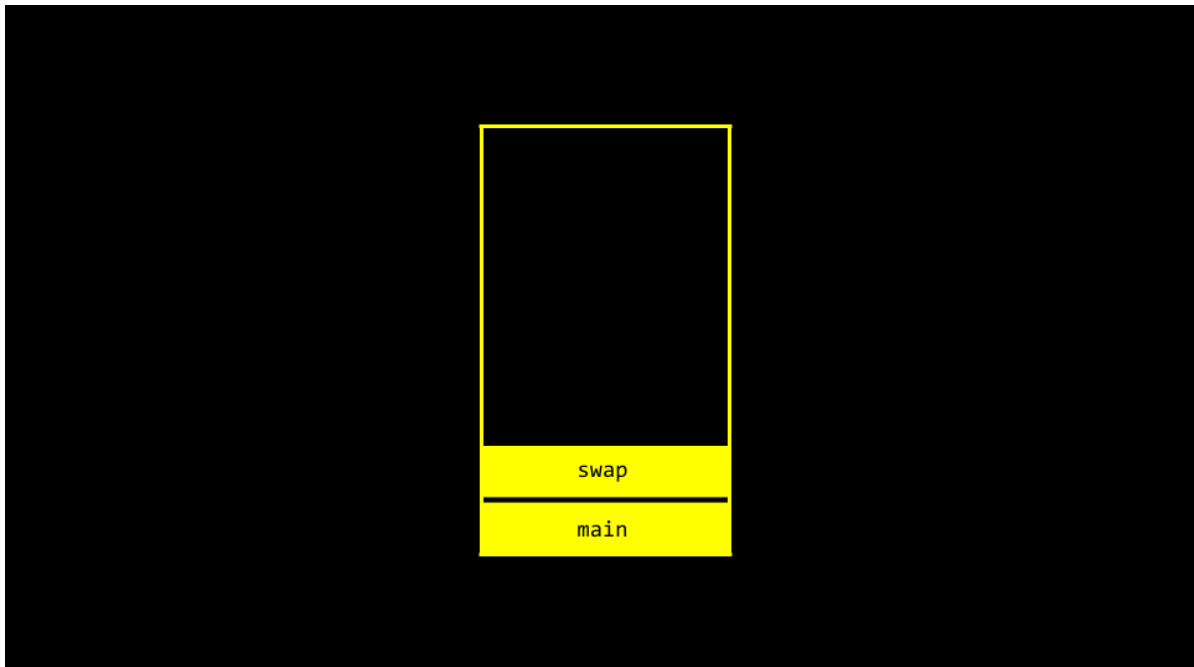
Notice that while this code runs, it does not work. The values, even after being sent to the `swap` function, do not swap. Why?

- When you pass values to a function, you are only providing copies. In previous weeks, we discussed the concept of *scope*. The values of `x` and `y` created in the curly `{}` braces of the `main` function only have the scope of the `main` function. Consider the following image:



Notice that *global* variables, which we have not used in this course, live in one place in memory. Various functions are stored in the `stack` in another area of memory.

- Now, consider the following image:



Notice that `main` and `swap` have two separate *frames* or areas of memory. Therefore, we cannot simply pass the values from one function to another to change them.

- Modify your code as follows:

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

```
void swap(int *a, int *b);

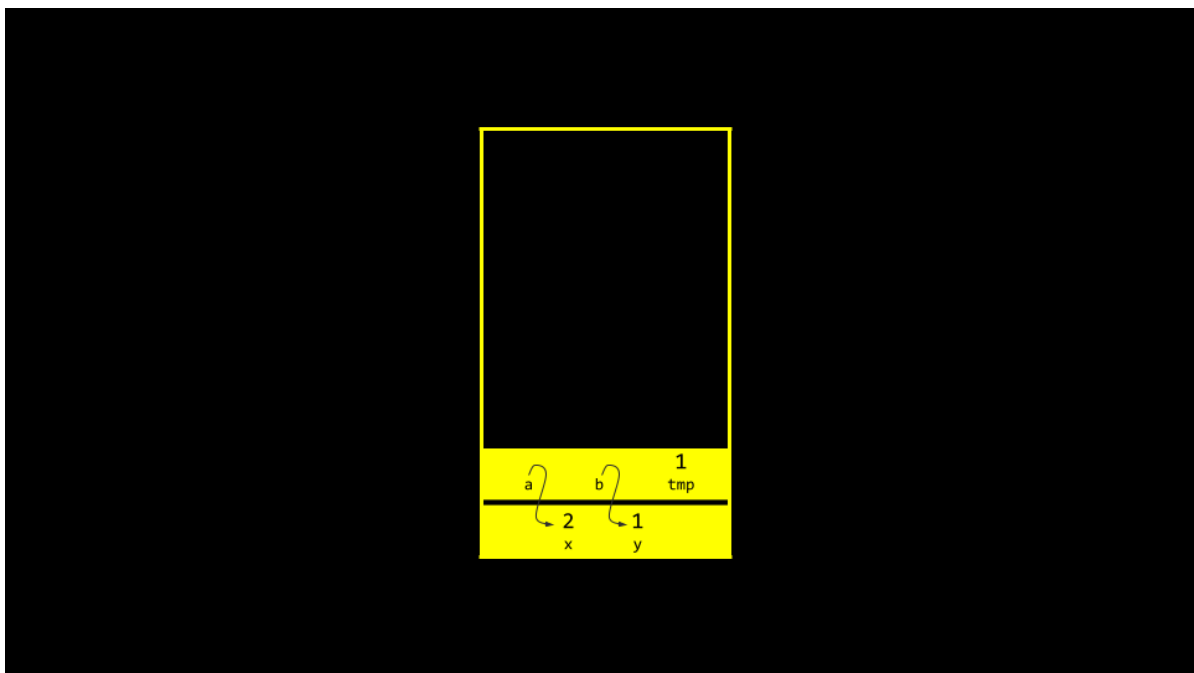
int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(&x, &y);
    printf("x is %i, y is %i\n", x, y);
}

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

Notice that variables are not passed by *value* but by *reference*. That is, the addresses of `a` and `b` are provided to the function. Therefore, the `swap` function can know where to make changes to the actual `a` and `b` from the main function.

- You can visualize this as follows:



Overflow

- A *heap overflow* is when you overflow the heap, touching areas of memory you are not supposed to.
- A *stack overflow* is when too many functions are called, overflowing the amount of memory available.
- Both of these are considered *buffer overflows*.

scanf

- In CS50, we have created functions like `get_int` to simplify the act of getting input from the user.
- `scanf` is a built-in function that can get user input.
- We can reimplement `get_int` rather easily using `scanf` as follows:

```
#include <stdio.h>

int main(void)
{
    int x;
    printf("x: ");
    scanf("%i", &x);
    printf("x: %i\n", x);
}
```

Notice that the value of `x` is stored at the location of `x` in the line `scanf("%i", &x)`.

- However, attempting to reimplement `get_string` is not easy. Consider the following:

```
#include <stdio.h>

int main(void)
{
    char *s;
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

Notice that no `&` is required because strings are special. Still, this program will not function. Nowhere in this program do we allocate the amount of memory required for our string. Indeed, we don't know how long of a string may be inputted by the user!

- Further, your code could be modified as follows. However, we have to pre-allocate a certain amount of memory for a string:

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

int main(void)
{
    char *s = malloc(4);
    if (s == NULL)
    {
        return 1;
    }
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}
```

```
    free(s);  
    return 0;  
}
```

Notice that if a string that is six bytes is provided you *might* get an error.

- Simplifying our code as follows we can further understand this essential problem of pre-allocation:

```
#include <stdio.h>  
  
int main(void)  
{  
    char s[4];  
    printf("s: ");  
    scanf("%s", s);  
    printf("s: %s\n", s);  
}
```

Notice that if we pre-allocate an array of size `4`, we can type `cat` and the program functions. However, a string larger than this *could* create an error.

- Sometimes, the compiler or the system running it may allocate more memory than we indicate. Fundamentally, though, the above code is unsafe. We cannot trust that the user will input a string that fits into our pre-allocated memory.

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`:

```
#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.

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:

```
#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:

```
#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` that 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!

Summing Up

In this lesson, you learned about pointers that provide you with the ability to access and manipulate data at specific memory locations. Specifically, we delved into...

- Pixel art
- Hexadecimal
- Memory
- Pointers
- Strings
- Pointer Arithmetic
- String Comparison
- Copying
- malloc and Valgrind
- Garbage values
- Swapping
- Overflow
- `scanf`
- File I/O

See you next time!