Table of Contents

Lecture 2

C, continued

- Last time, we learned how to use the CS50 IDE to write and compile source code, and run the compiled program in our terminal.
- Recall that `clang` is one compiler that can take source code written in C, and turn it into machine code, instructions in 0s and 1s, that our computer can actually run.
- And recall that, in order to use functions like `get_string`, we need to `#include <cs50.h>` at top, or `#include <stdio.h>` to use `printf`, among others.
- If we see any errors when compiling, we should always look at the first one, try to fix it, and recompile, since the following errors might simply be a result of the first error.
- If we see an error we don't understand, we can use a command written by CS50 staff, `help50`. For example, we could run `help50 clang hello.c`, and see the following:

```
Helping with...

hello.c:5:5: error: use of undeclared identifier 'string'; did you mean 'stdin'?

By "undeclared identifier," clang means you've used a name string on line 5 of
hello.c which hasn't been defined. Did you forget to #include <cs50.h> (in which
string is defined) atop your file?
```

  - By reading the yellow text, we get a hint to indeed `#include` the library that we initially forgot to add.
- Now if we run `clang` again, we get a different error. When we add a header file to our source code, it only indicates that the library exists somewhere. Since the CS50 library wasn't built in (like `stdio.h` is), we also need to point `clang` to the file that contains the library's code:

```
~/workspace/ $ clang hello.c -lcs50
~/workspace/ $
```

Compiling

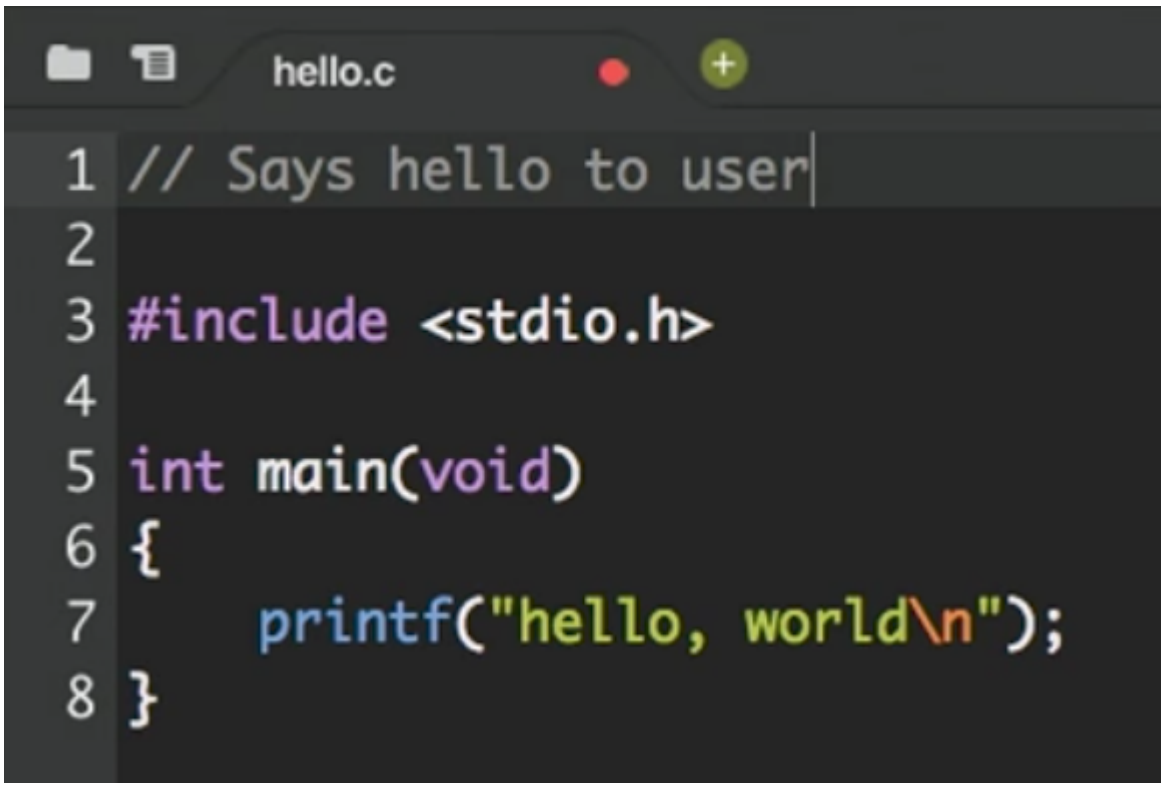
- So far, when we've used the term compiling, we were actually referring to a process that is made of four steps:
  - preprocessing
    - In C, preprocessing involves replacing the lines that start with `#include` with the contents of the actual file.
  - compiling
    - The compiler takes the complete source code and converts it to assembly code, much simpler instructions that look like this:

      ```
      main:                                 #   @main
          .cfi_startproc
      # BB#0:
          pushq   %rbp
      .Ltmp0:
          .cfi_def_cfa_offset 16
      .Ltmp1:
          .cfi_offset %rbp, -16
          movq %rsp, %rbp
      .Ltmp2:
          .cfi_def_cfa_register %rbp
          subq $16, %rsp
          movabsq $.L.str, %rdi
          movb $0, %al
          callq   printf
      ...
      ```

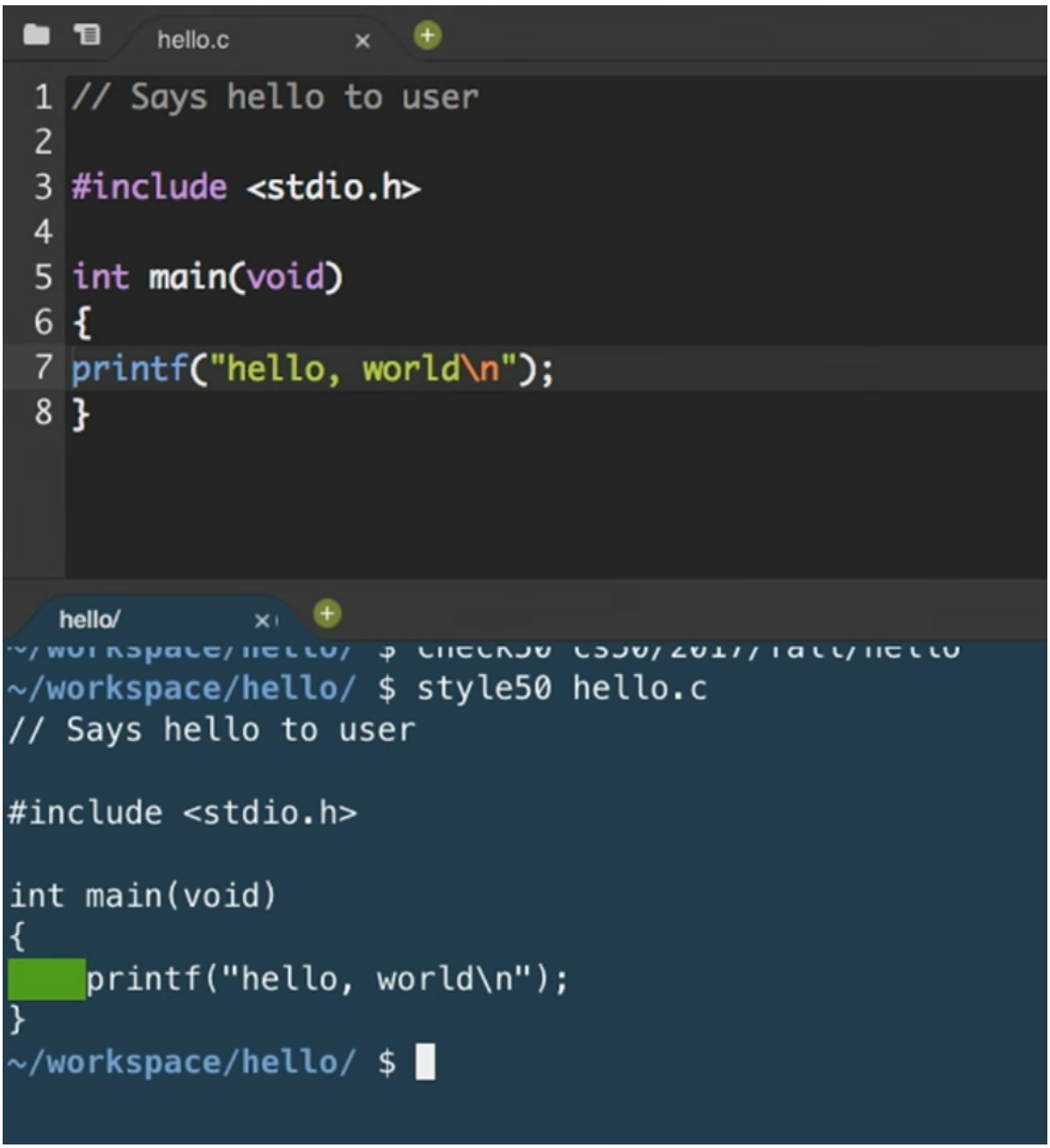
      These lines are single-step arithmetic or memory management instructions that CPUs can perform.
  - assembling
    - Finally, these lines of assembly are converted to 0s and 1s that the CPU can directly understand.
  - linking
    - We also need to combine into our program the binary file for standard I/O library that we call functions from, and this last step does exactly that. Recall that we only included `stdio.h`, which is just the header file that declares the functions, not the actual code for them.
- By having different stages, we can more closely examine, debug, and work with each layer, so the more complicated systems that we build atop them are cleaner, more secure, and better-designed.

Tools

- For the problem sets, you'll be able to use two other tools written by staff, `check50` and `style50`.
  - `check50` will run test cases against our program, by uploading it to CS50's servers.
  - `style50` will check for style, like indentation, variable naming, and comments. For our `hello` program, we might add a one-line summary at the top:



If we forgot to indent or indented too much, `style50` will also indicate that with color: green for spacing it suggests adding, and red for spacing we should remove:



Like with compiler errors, if we see a lot of output, we can try fixing one or a few things at once, and re-running to see our progress.

- As for design, we'll learn from practice, examples, and feedback from human TFs!

Printing, Debugging

- Super Mario Bros. is a classic video game from the 1980s, where the main character Mario runs across the screen, jumping into bricks and enemies.
- We can quickly write a program, with what we already know, to print 4 question marks in our terminal:

```c
#include <stdio.h>

int main(void)
{
    printf("????");
}
```

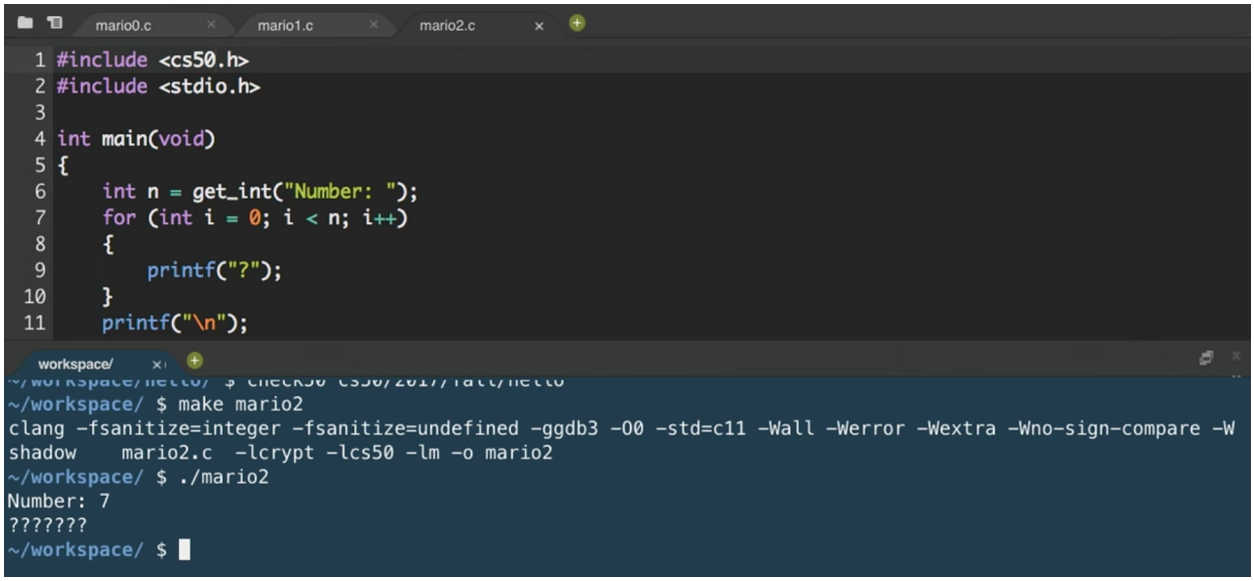
```
~/workspace/ $ make mario0
clang -fsanitize=integer -fsanitize=undefined -ggdb3 -O0 -std=c11 -Wall -Werror -Wextra -Wno-sign-compare -W
shadow    mario0.c  -lcrypt -lcs50 -lm -o mario0
~/workspace/ $ ./mario0
????~/workspace/ $
```

  - We can immediately improve this by adding `\n` to the end of the string of question marks: `"????\n"`.
- We can try using a `for` loop to improve our program:

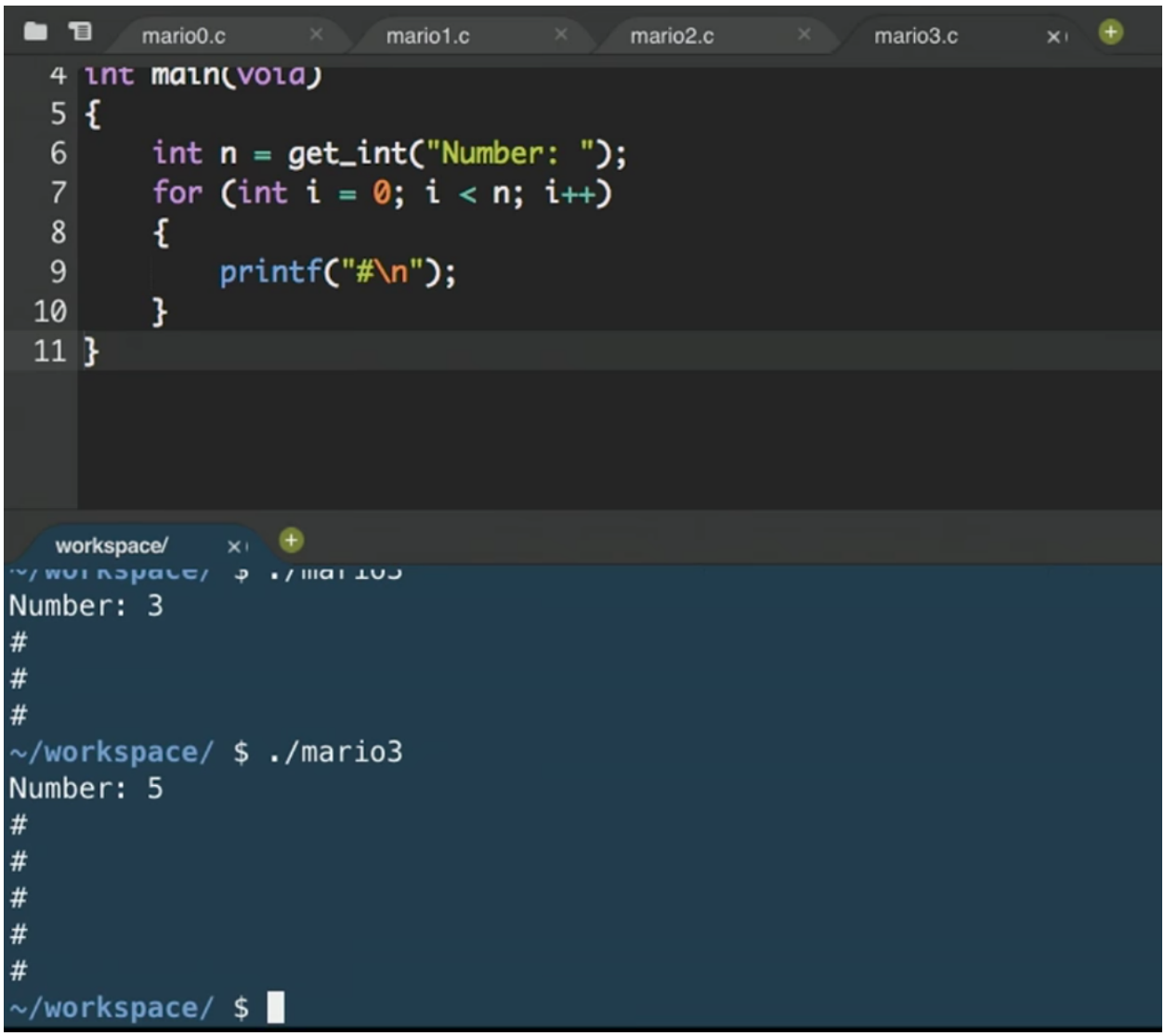
```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 4; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

```
~/workspace/ $ make mario1
clang -fsanitize=integer -fsanitize=undefined -ggdb3 -O0 -std=c11 -Wall -Werror -Wextra -Wno-sign-compare -W
shadow    mario1.c  -lcrypt -lcs50 -lm -o mario1
~/workspace/ $ ./mario1
????
~/workspace/ $
```

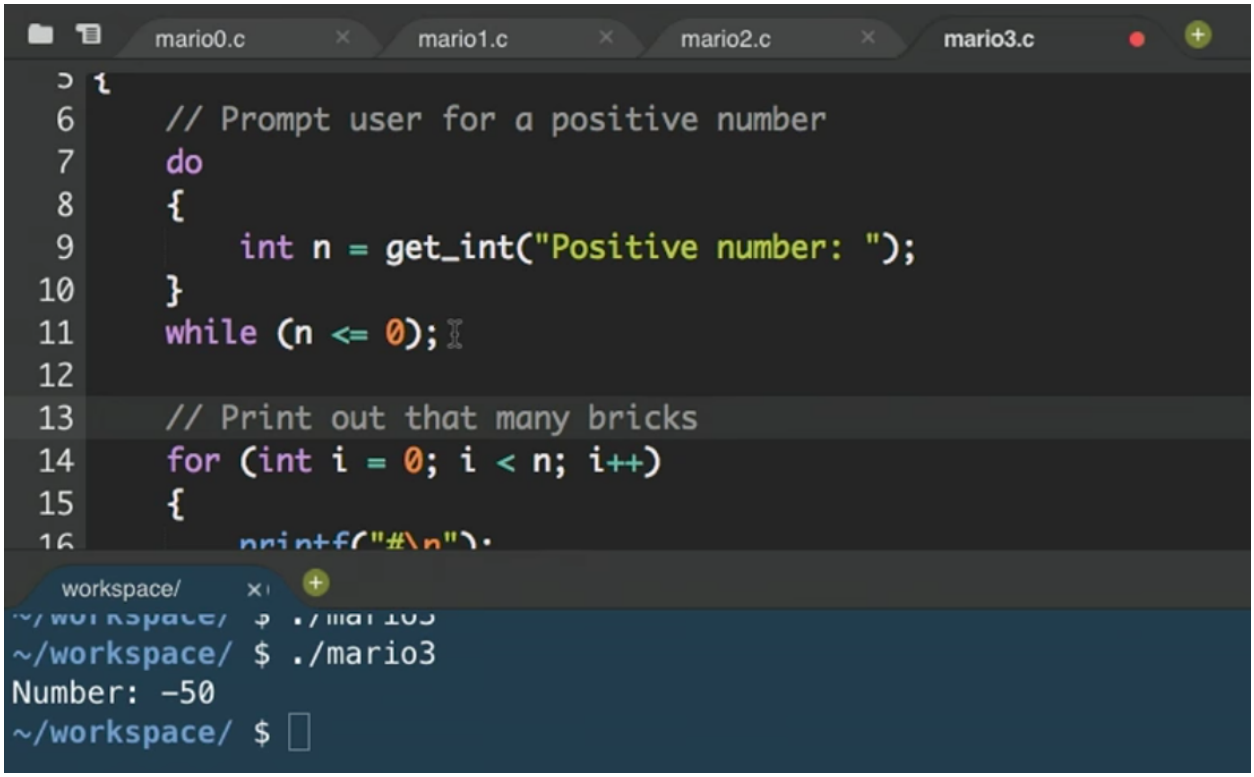
  - We start with `i = 0` and check whether `i < 4` by convention, which makes for 4 iterations of the loop.
  - We also add the `\n` to the outside of the loop, since we want just one, when we're done printing our question marks.
- Finally, we can make a version of this program that takes input from the user, and uses that to print some number of question marks:



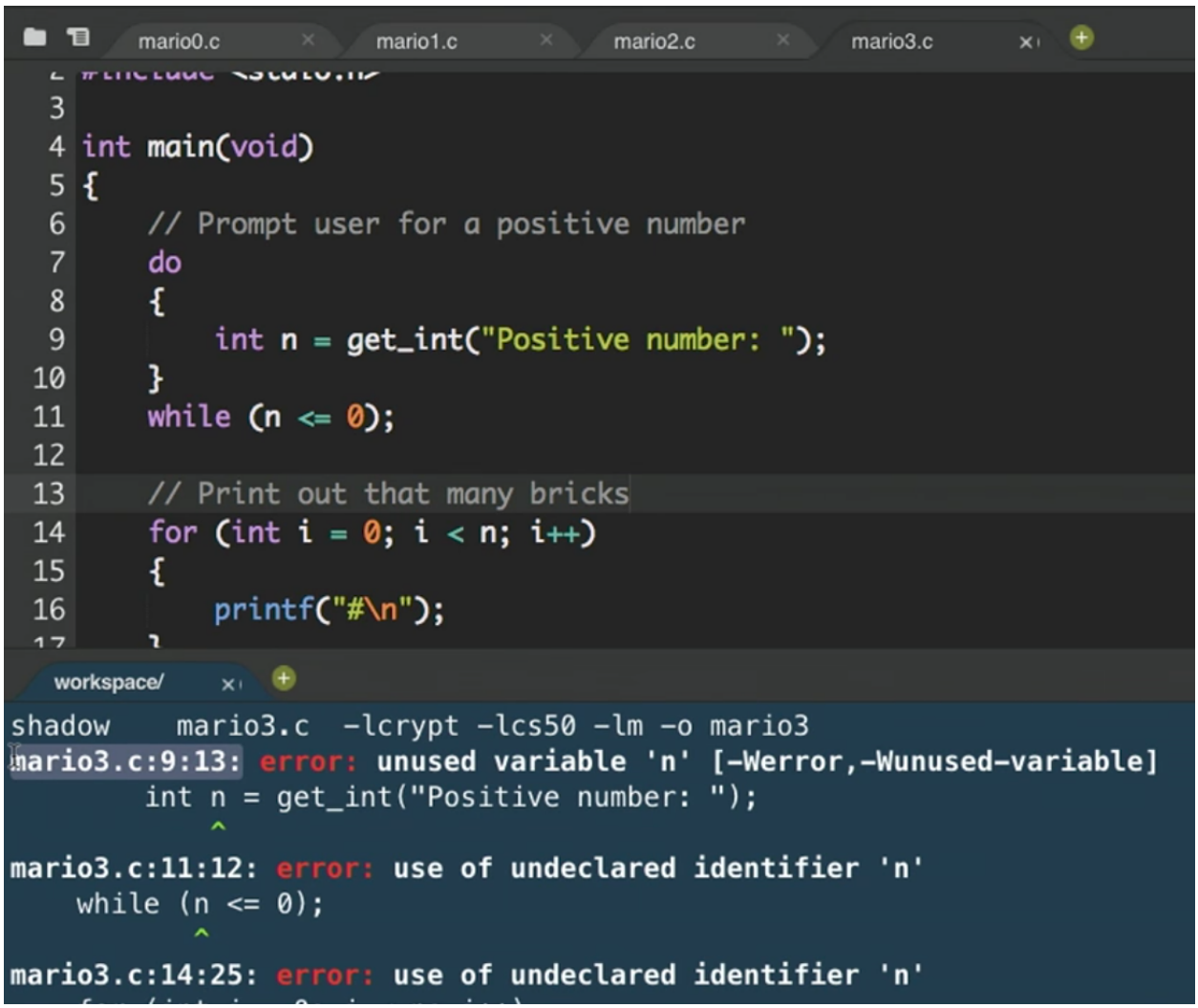
  - If we typed in a negative number as input, it would be saved to `n` since it's a valid integer, but the `for` loop would check the condition and move on immediately, since `i` would not be less than `n`.
  - And if we accidentally had a bug in our code or input that caused too many question marks to be printed, we can press Control + C on our keyboard to stop the program immediately.
- Now, if we wanted to print a vertical line of blocks, we can add `\n` to each line inside the `for` loop:



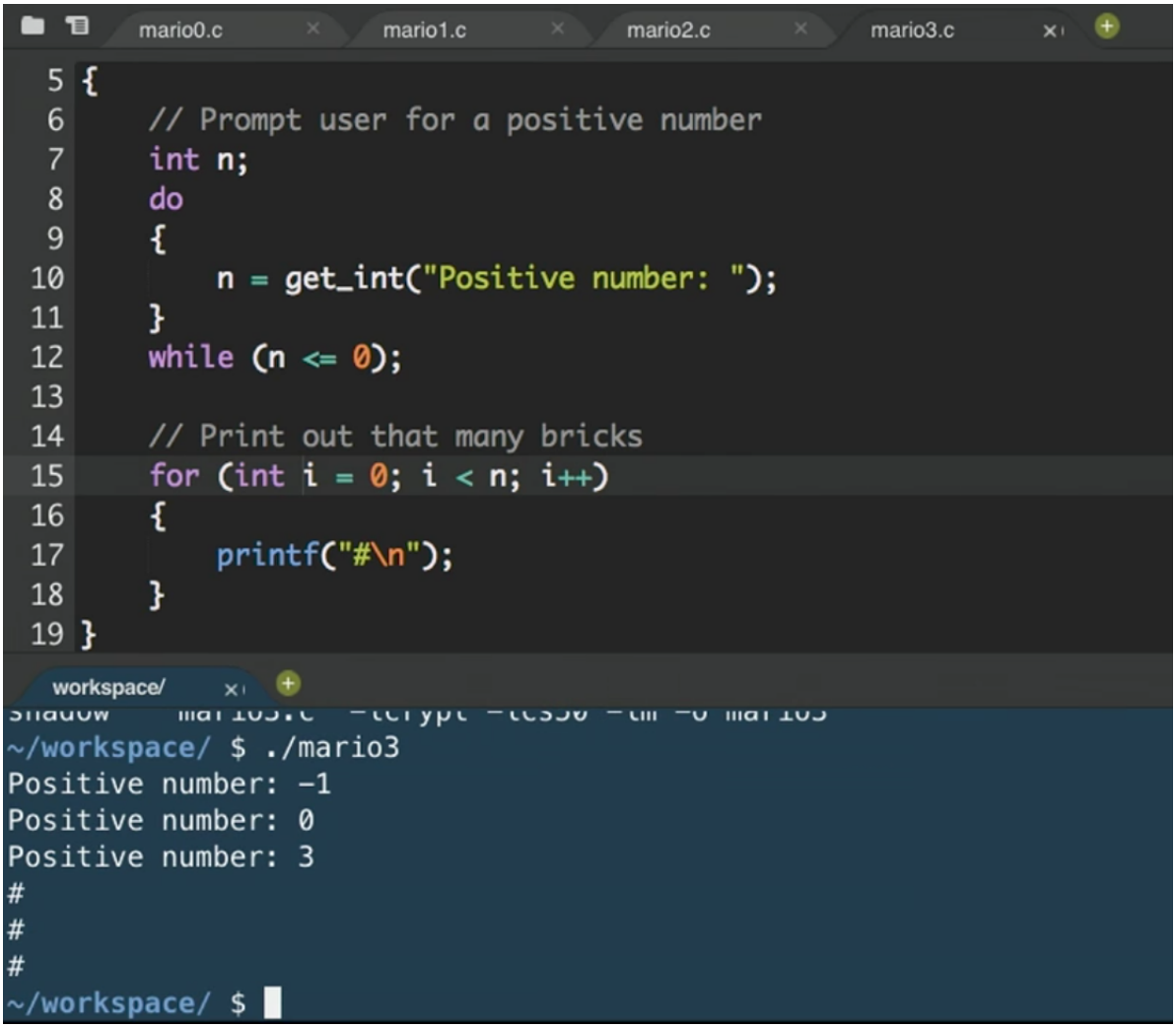
- We can also re-prompt the user for a number if the integer we get from them is negative, by using a do-while loop:



    - A do-while loop does something at least one time, then checks the condition, and repeats until the condition is no longer true.
- But when we compile this, we get lots of errors:



- First, we notice that for each error, we are told the line number and character or column where the error is.
- Somehow, our error is that `n` is not used and used when it isn't declared yet.
- In Scratch, variables created somewhere can be used anywhere. But in C, and other languages, variables have a **scope**, or level of code where it can be used, based on where it is initialized. For C, we can generally think of `scope` as being limited to being within the closest set of curly braces.
- In this case, `n` only exists within the `do` part of the loop, since it is initialized inside. To fix this, we need to make the following change:



    - We can initialize `n` outside the do-while loop with no value, and it will be accessible inside the loop since the loop is within the scope of the `main` function (the curly braces that surround the initialization of `n`).

- We could use a `while` loop if we initialize `n` to some negative value, but it's not clear where that value is from, and is considered bad design:

```c
{
    // Prompt user for a positive number
    int n = -1000;
    while (n <= 0)
    {
        n = get_int("Positive number: ");
    }

    // Print out that many bricks
    for (int i = 0; i < n; i++)
    {
        printf("#\n");
    }
}
```

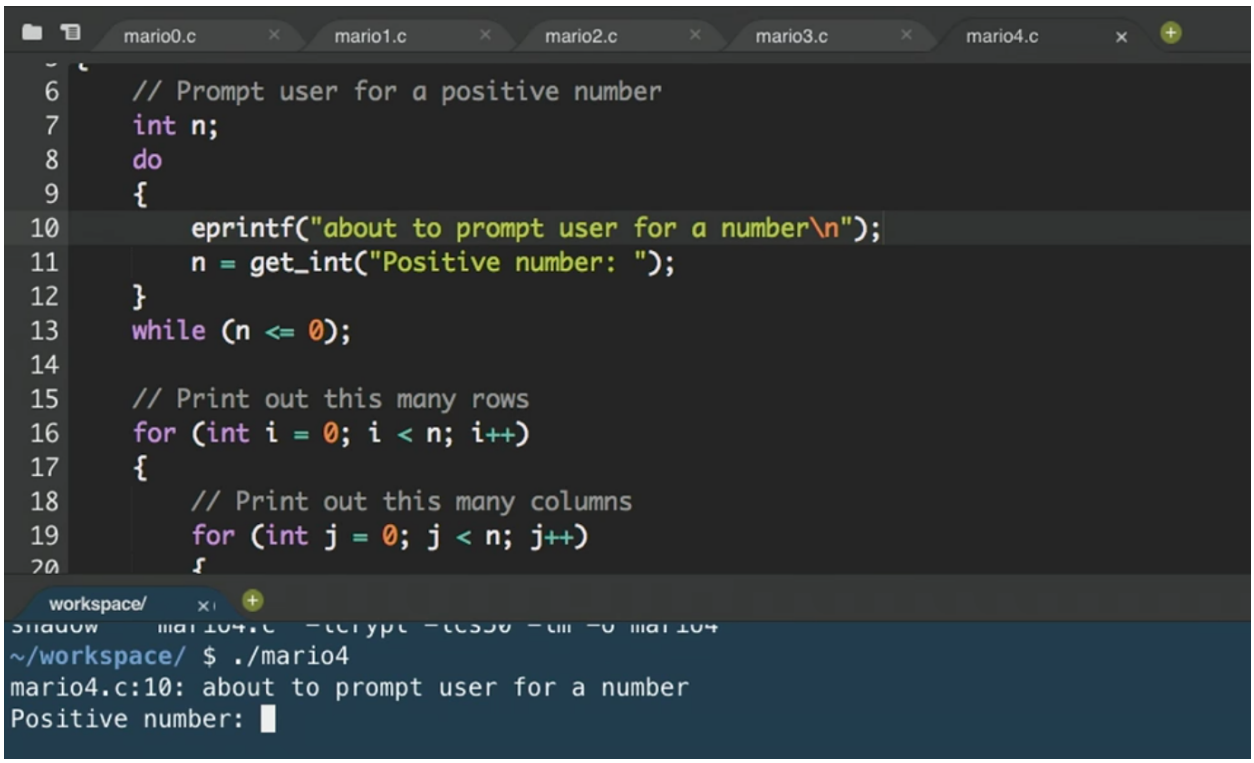
- Now let's print a square of bricks:

```c
    {
        n = get_int("Positive number: ");
    }
    while (n <= 0);

    // Print out this many rows
    for (int i = 0; i < n; i++)
    {
        // Print out this many columns
        for (int j = 0; j < n; j++)
        {
            printf("#");
        }
        printf("\n");
    }
```

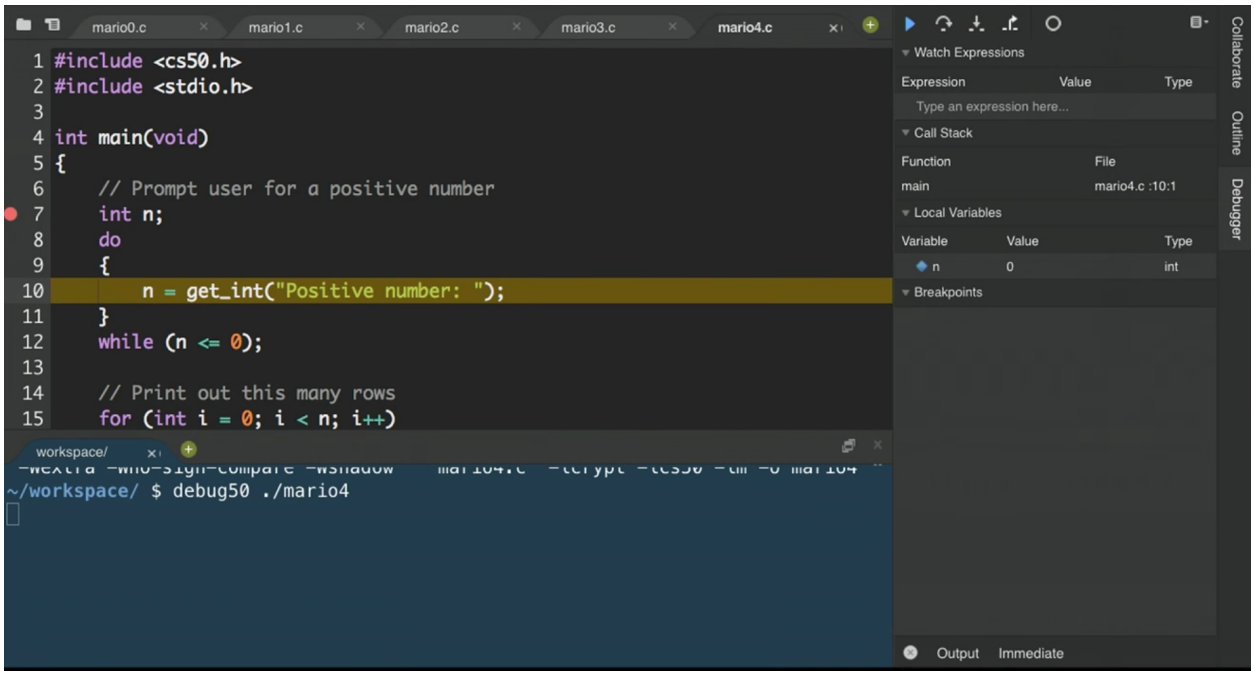
```
~/workspace/ $ ./mario4
Positive number: 5
#####
#####
#####
#####
#####
~/workspace/ $
```

  - We can think of `printf` as being able to print to the terminal like a typewriter: it can print one character after another, and use a new line to move to the next line.
  - Here we are nesting one `for` loop inside another, using `j` as our counter to avoid mixing up our counts.
  - In the inner `for` loop, we print `#` the right number of times for each row, and follow that with a new line. The outer `for` loop will then repeat that for the right number of rows.

- We can use another tool, `eprintf`, to provide information to ourselves when our program is running:



- We should also use the debugger, by clicking on line numbers to the left of our code:



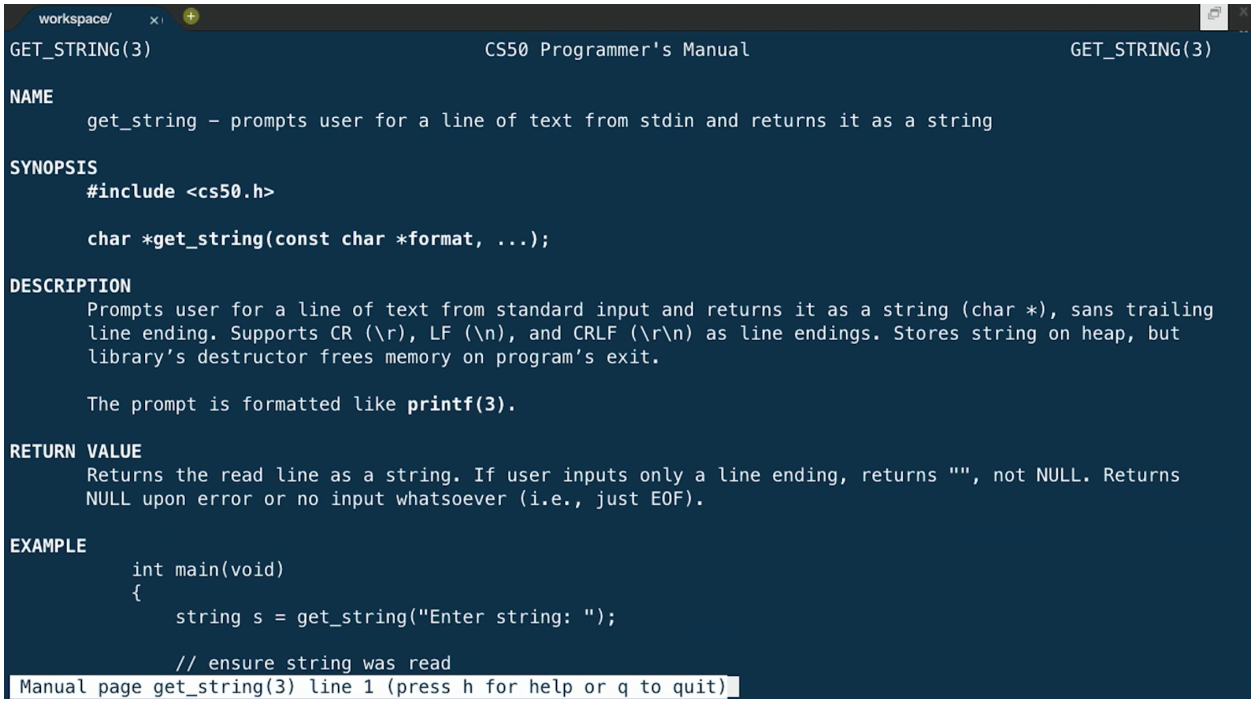
  - The red dot is called a **breakpoint**, which pauses our program at that line.
  - Then we can run `debug50 ./mario4`, and the panel on the right automatically opens.
  - We see a section called "Local Variables", underneath which we see that our only variable so far, `n` is `0` at that point in our code.
  - We can use the buttons on the top of that panel to control our program precisely. The triangle that looks like a play button will let the program resume normally until it reaches another breakpoint, if any. The next button, an arrow in the shape of a half-circle, will run the very next line of code, and pause the program again. The next button, the arrow pointing downwards, allows us

to "step into" that line of code, and finally, the last arrow pointing up and to the right allows us to step back out of the next line of code.

- We take a quick break by looking at this video of Super Mario Bros. recreated in augmented reality, with the Microsoft HoloLens headset.

Strings, Arrays

- There are several functions in the CS50 Library that allow us to get variables from the user:
  - `get_char`
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long_long`
  - `get_string`
- As implied by these function names, C supports various types of data that a variable can store:
  - `bool`
  - `char`
  - `double`
  - `float`
  - `int`
  - `long long`
  - `string`
  - ...
- Similarly, we can substitute different variable types into `printf` with different format codes:
  - `%c`
  - `%f`
  - `%i`
  - `%lld`
  - `%s`
  - ...
- To look up the documentation for these functions and format codes, among other information, we can use the built-in manual in the terminal.
- For example, we can type `man get_string` and see the following:



- But even that might be too hard to understand at first, so CS50 Staff have created https://reference.cs50.net/, with explanations that might be a bit more friendly. And we can toggle the level of comfort with the checkbox that indicates as much.
- A TF in New Haven, Stelios, can store his name, character by character, in memory like so:

```
| S | t | e | l | i | o | s |
```

- Indeed, a string is just an abstraction for a sequence of characters. Let's experiment with `string0.c`:

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

int main(void)
{
    string s = get_string("input:  ");
    printf("output: ");
    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c\n", s[i]);
    }
}
```

  - First, we start a `for` loop since we want to print each character of a string that's provided as input. The loop should end at the end of the string, so we can determine that with `strlen(s)`, a function that returns to us the length of the string.
  - Then, inside the loop, we use `%c` to print out each character. And to access each character, we use `i` as the index into the character in the string `s`: the first character is at index 0, the second at index 1, and so forth. The notation to get an item at a certain index in an array, or a list of values one after another, is what we see substituted into `printf`: `s[i]`.

    - And at top, we include a new library, `string.h`, that has various functions for working with strings, including `strlen`.
- Now, we can actually see how ASCII maps numbers to letters. **Typecasting** is casting, or converting, variables from a certain type, like `int`, to another, like `char`, or vice versa.
- Let's add this to our previous program:

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

int main(void)
{
    string s = get_string("Name: ");
    for (int i = 0; i < strlen(s); i++)
    {
        printf("%c %i\n", s[i], (int) s[i]);
    }
}
```

    - Notice that we are substituting `(int) s[i]` for the `%i` in the string we print out, and `(int)` typecasts the character at `s[i]` to an `int`.
- Now, we can work with strings directly. For example, we can capitalize a string, letter by letter:

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

int main(void)
{
    string s = get_string("before: ");
    printf("after:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            printf("%c", s[i] - ('a' - 'A'));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

    - First, we notice that the `for` loop now initializes two variables, `i` and `n`, at the start. `n` is set to the length of `s`, and we can do this once at the beginning of the loop since we know the length won't change. Then, the loop won't need to compute the length of the string on each iteration when it compares `i` to `strlen`. Instead, it can just compare it to `n`, which we already saved.

    - Within the loop, we check whether each character is lowercase. We can compare the values of one `char` with another directly, and use `&&` to indicate a logical `and` in C, such both Boolean expressions need to be true for the condition to run. And if it is indeed lowercase, we do some arithmetic to capitalize it. Fortunately, in ASCII, all lowercase letters are offset from the capital counterparts by the same amount. So we can subtract that difference from a lowercase letter, and get the number for the capital version of that letter.
- It turns out, C has built-in functions that are really helpful for doing this:

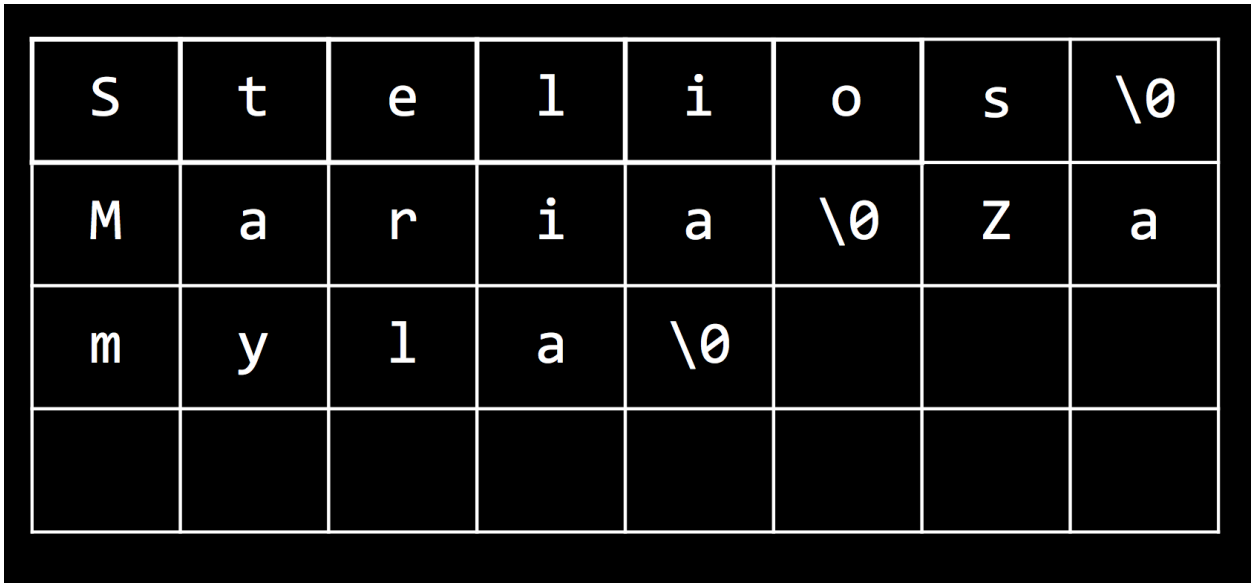
```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    string s = get_string("before: ");
    printf("after:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        if (islower(s[i]))
        {
            printf("%c", toupper(s[i]));
        }
        else
        {
            printf("%c", s[i]);
        }
    }
    printf("\n");
}
```

    - `islower` and `toupper` are functions from yet another library, `ctype.h,` that we can use to achieve the same effects as what we manually did earlier.
    - `islower` returns a Boolean value, either `true` or `false`, which we can check in our condition. And `toupper` returns the uppercase version of the character passed into it.
- We can, by looking at the documentation, realize that `toupper` will work on any character and only convert it to uppercase if it's already lowercase, so we don't even need to make that check ourselves:

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

int main(void)
{
    string s = get_string("before: ");
    printf("after:  ");
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        printf("%c", toupper(s[i]));
```

```
    }
    printf("\n");
}
```

- Now let's go in the other direction, and see if we can try to implement `strlen` ourselves.
- First, we realize that our computer's memory is just lots and lots of bytes, ordered one after another. We can represent that in a grid:



  - Each of the boxes in the grid are numbered, from 0 to some number in the billions (depending on the amount of memory we have).
  - And C stores strings in memory with one character in each byte, but also with a terminating, or ending character, at the end of each string. This special **null character**, or `\0`, is literally the number 0 (not the ASCII equivalent for the character 0).
  - If we were to store many strings in our computer's memory, they might end up being stored like this, one after another.
  - For other types of data in C, a fixed number of bytes is allocated for them every time, so they do not need a terminating character.
- Knowing that, we can write the following code:

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

int main(void)
{
    string s = get_string();
    int n = 0;
    while (s[n] != '\0')
    {
        n++;
    }
    printf("%i\n", n);
}
```

    - We create a variable `n` to store the length of our string, and check that the character at each index of `n` is not the null character, before we increment it.
    - Finally, we print out what our counter reached before the loop ended.
- In C, we can create arrays, or lists, comprised of elements of the same type of data. Strings, as we've seen, are just arrays of characters.
- To be more precise, an array is a contiguous chunk of memory of elements of the same type, stored back to back.
- So far, we've started writing our programs by defining the main function as `int main(void)`: `main` is a function that takes `void`, or nothing, as its arguments, and returns an `int`.
- We can change that so our program takes input not when it runs, but before it runs, at the command-line, as does `clang` or `style50`, so we avoid having to wait for prompts.
- We can try the following with `argv0.c`:

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

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

    - Notice that `main` now takes in two arguments. The first, `argc`, is a count of how many arguments were passed in. The second, `argv`, is an array of strings, each of which are the arguments typed at the prompt.
    - If `argc`, or the number of arguments, is 2, then we print out the second of those arguments. `argv[0]`, the first argument, will always be the name of our program.
    - We can compile and run this program with something like `./hello David`, and see as output `hello, David`.
- We can try to access an element in an array that we know doesn't exist:

- The error we get when we run our program, a "segmentation fault", means that our program tried to access our program that it should not have.

Cryptography

- We can start applying what we've learned to the domain of cryptography.
- One way to encrypt, or scramble data, so that it can be decrypted, or unscrambled, is to map each letter in the alphabet to some other letter with a key.
- Encrypted data, or ciphertext, is the scrambled version of plaintext, or the original, easily-readable data. To get from plaintext to ciphertext, and vice versa, we need to know the key, or some piece of information, like a number that indicates how many letters we need to shift each letter in our plaintext by.
- A clip from the Christmas Story shows the main character Ralphie using a toy decoder ring to decrypt a message by hand, only to be disappointed that the message is an advertisement for an old-fashioned beverage, Ovaltine.