Lecture 2

Compiling

- We started the course with Scratch, and then learned C.
- Recall that we write our source code in C, but needed to compile it to machine code, in binary, before our computers could run it.
  - `clang` is the compiler we learned to use, and `make` is a utility that helps us run `clang` without having to indicate all the options manually.
  - If we wanted to use CS50's library, via `#include <cs50.h>`, and use `clang` instead of `make`, we also have to add a flag: `clang hello.c -lcs50`. The `-l` flag *links* the `cs50` file, which was installed into the CS50 Sandbox.
- "Compiling" source code into machine code is actually made up of smaller steps:
  - preprocessing
  - compiling
  - assembling
  - linking
- *Preprocessing* involves looking at lines that start with a `#`, like `#include`, before everything else. For example, `#include <cs50.h>` will tell `clang` to look for that header file first, since it contains content that we want to include in our program. Then, `clang` will essentially replace the contents of those header files into our program:

```
...
string get_string(string prompt);
int printf(const char *format, ...);
...
int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- *Compiling* takes our source code, in C, and converts it to assembly code, which looks 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
    xorl %eax, %eax
    movl %eax, %edi
    movabsq $.L.str, %rsi
    movb $0, %al
    callq get_string
    movabsq $.L.str.1, %rdi
    movq %rax, -8(%rbp)
    movq -8(%rbp), %rsi
    movb $0, %al
    callq printf
    ...
```

  - These instructions are lower-level and can be understood by the CPU more directly, and generally operate on bytes themselves, as opposed to abstractions like variable names.
- The next step is to take the assembly code and translate it to instructions in binary by *assembling* it.
- Now, the final step is *linking*, where the contents of linked libraries, like `cs50.c`, are actually included in our program as binary.

Debugging

- Let's say we wrote this program, `buggy0`:

```
int main(void)
{
    printf("hello, world\n")
}
```

  - We see an error, when we try to `make` this program, that we didn't include a missing header file.
  - We can also run `help50 make buggy0`, which will tell us, at the end, that we should `#include <stdio.h>`, which contains `printf`.
  - We do that, and see another error, and realize we're missing a semicolon at the end of our line.
- Let's look at another program:

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

```c
int main(void)
{
    for (int i = 0; i <= 10; i++)
    {
        printf("#\n");
    }
}
```

- Hmm, we intended to only see 10 `#`s, but there are 11. If we didn't know what the problem is (since our program is working as we wrote it), we could add another print line to help us:

```c
#include <stdio.h>

int main(void)
{
    for (int i = 0; i <= 10; i++)
    {
        printf("i is %i\n", i);
        printf("#\n");
    }
}
```

- Now, we see that `i` started at 0 and continued until it was 10, but we should have it stop once it's at 10.

- If we wrote our program without any whitespace, like the below, it would still be correct:

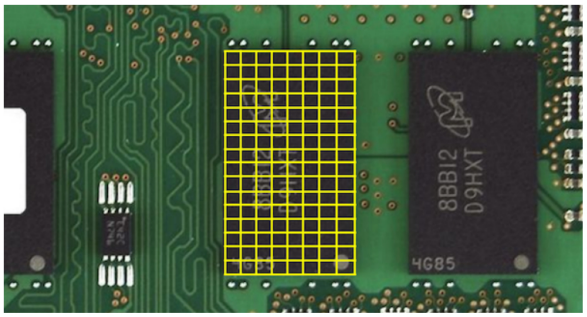
```c
#include <stdio.h>

int main(void)
{
for (int i = 0; i < 10; i++)
{
printf("i is %i\n", i);
printf("#\n");
}
}
```

- But, our program is much harder to read, and so it's poorly styled. With indentation for our loops, it'll be easier to see the nesting of our lines of code.
- We can run `style50 buggy2.c`, and see suggestions for what we should change.

- So to recap, we have three tools to help us improve our code:
  - `help50`
  - `printf`
  - `style50`

Memory

- Inside our computers, we have chips called RAM, random-access memory, that stores data for short-term use. We might save a file to our hard drive (or SSD) for long-term storage, but when we open it and start making changes, it gets copied to RAM. Though RAM is much smaller, and temporary (until the power is turned off), it is much faster.
- We can think of bytes, stored in RAM, as though they were in a grid:



- In reality, there are millions or billions of bytes per chip.

- In C, when we create a variable of type `char`, which will be sized one byte, it will physically be stored in one of those boxes in RAM. An integer, with 4 bytes, will take up four of those boxes.

Arrays

- In memory, we can store variables one after another, back-to-back. And in C, a list of variables stored, one after another in a contiguous chunk of memory, is called an *array*.
- It turns out, we can do interesting things with just array.
- Let's look at `scores0.c`:

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

int main(void)
{
    // Get scores from user
    int score1 = get_int("Score 1: ");
    int score2 = get_int("Score 2: ");
    int score3 = get_int("Score 3: ");

    // Generate first bar
    printf("Score 1: ");
    for (int i = 0; i < score1; i++)
    {
        printf("#");
    }
```

```
    printf("\n");

    // Generate second bar
    printf("Score 2: ");
    for (int i = 0; i < score2; i++)
    {
        printf("#");
    }
    printf("\n");

    // Generate third bar
    printf("Score 3: ");
    for (int i = 0; i < score3; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- We get 3 scores from the user, and print bars for each score.
- Our 3 integers, `score1`, `score2`, and `score3` will be stored somewhere in memory.

- We can use a loop, but we can start factoring out pieces:

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

void chart(int score);

int main(void)
{
    // Get scores from user
    int score1 = get_int("Score 1: ");
    int score2 = get_int("Score 2: ");
    int score3 = get_int("Score 3: ");

    // Chart first score
    printf("Score 1: ");
    chart(score1);

    // Chart second score
    printf("Score 2: ");
    chart(score2);

    // Chart third score
    printf("Score 3: ");
    chart(score3);
}

// Generate bar
void chart(int score)
{
    // Output one hash per point
    for (int i = 0; i < score; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- Now, we have a `chart` function that can print each score.
- Remember that we need our prototype, `void chart(int score);`, to be at the top. We could also have the entire `chart` function at the top, before we use it, but eventually our `main` function would be pushed down too far, and be harder and harder to find.

- With an array, we can collect our scores in a loop, and access them later in a loop, too:

```
// Generates a bar chart of three scores using an array

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

void chart(int score);

int main(void)
{
    // Get scores from user
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
        scores[i] = get_int("Score %i: ", i + 1);
    }

    // Chart scores
    for (int i = 0; i < 3; i++)
    {
        printf("Score %i: ", i + 1);
        chart(scores[i]);
    }
}

// Generate bar
void chart(int score)
{
    // Output one hash per point
    for (int i = 0; i < score; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

- Notice that we use `int scores[3]` to initialize an array for 3 integers. Then, we use `scores[i] = ...` to store values into that array,

using some index `i` that goes from `0` to `2` (since there are 3 elements).
    - Then, we use `scores[i]` to access the values stored, at each index.
- We repeat the value `3` in a few times, so we can factor that out to a *constant*, or a number we can specify and use globally:

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

const int COUNT = 3;

void chart(int score);

int main(void)
{
    // Get scores from user
    int scores[COUNT];
    for (int i = 0; i < COUNT; i++)
    {
        scores[i] = get_int("Score %i: ", i + 1);
    }

    // Chart scores
    for (int i = 0; i < COUNT; i++)
    {
        printf("Score %i: ", i + 1);
        chart(scores[i]);
    }
}

// Generate bar
void chart(int score)
{
    // Output one hash per point
    for (int i = 0; i < score; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

    - At the top, we use the `const` keyword to indicate that this value shouldn't change. And we can use this throughout our code, so if we wanted this value to change, we only need to change it once. Finally, `COUNT` is in all capital letters, to indicate that it's a constant (by convention).
- We can have our `chart` function print the entire chart, not just one bar at a time:

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

const int COUNT = 3;

void chart(int count, int scores[]);

int main(void)
{
    // Get scores from user
    int scores[COUNT];
    for (int i = 0; i < COUNT; i++)
    {
        scores[i] = get_int("Score %i: ", i + 1);
    }

    // Chart scores
    chart(COUNT, scores);
}

// Generate bars
void chart(int count, int scores[])
{
    // Output one hash per point
    for (int i = 0; i < count; i++)
    {
        for (int j = 0; j < scores[i]; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

    - By passing in the entire `scores` array, as well as the `count` of scores we want to print, we can have the `chart` function iterate over `scores`. In fact, `chart` doesn't know how big the `scores` array actually is, so we necessarily have to pass in a `count`.

Strings

- Strings are actually just arrays of characters. We can see this 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]);
    }
}
```

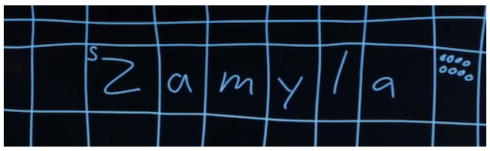
- First, we need a new library, `string.h`, for `strlen`, which tells us the length of a string. Then, we use the same syntax to access elements in arrays, `s[i]`, to print each individual character of the string `s`.

- We can improve the design of our program. `string0` was a bit inefficient, since we check the length of the string, after each character is printed, in our condition. But since the length of the string doesn't change, we can check the length of the string once:

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

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

  - Now, at the start of our loop, we initialize both an `i` and `n` variable, and remember the length of our string in `n`. Then, we can check the values each time, without having to actually calculate the length of the string.
  - `n` will only be accessible in the scope of the `for` loop, though we could initialize it outside of the loop, if we wanted to reuse it later.

- When a string is stored in memory, each character is placed into one byte into the grid of bytes. Somewhere, for example, `Zamyla` is stored in 6 bytes. But one more byte is needed, to indicate the end of the string:



  - The byte in memory where the first character of the string, `Z`, is stored, is labeled `s`, since we called our string `s` in the code above. Then, after the last character, `a`, we have one byte with all `0`s, to indicate the end of the string. And the byte of all `0`s is called a null character, which we can also write as `\0`.

- If we wanted to write our own version of `strlen`, for example, we would need to know this:

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

int main(void)
{
    // Prompt for user's name
    string s = get_string("Name: ");

    // Count number of characters up until '\0' (aka NUL)
    int n = 0;
    while (s[n] != '\0')
    {
        n++;
    }
    printf("%i\n", n);
}
```

  - Here, we iterate over each character of the string `s` with the syntax we use to access elements in arrays, and we increment a counter, `n`, as long as the character isn't the null character, `\0`. If it is, we're at the end of the string, and can print out the value of `n`.

- And, since we know that each character has a numeric, ASCII value, we can even print that:

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

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

  - With `(int) s[i]`, we take the value of `s[i]` and convert that character type to an integer type. Then, we can print out both the character and its numeric value.
  - Technically, we can even do `printf("%c %i\n", s[i], s[i]);`, and `printf` will interpret the value of `s[i]` as an integer.

- We can now combine what we've seen, to write a program that can capitalize letters:

```
#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 get a string `s`. Then, for each character in the string, if it's lowercase (its value is between that of `a` and `z`), we convert it to uppercase. Otherwise, we just print it.
- We can convert a lowercase letter to its uppercase equivalent, by subtracting the difference between a lowercase `a` and an uppercase `A`. (We know that lowercase letters have a higher value than uppercase letters, and so we can subtract that difference to get an uppercase letter from a lowercase letter.)

- But there are library functions that we can use, to accomplish the same thing:

```
#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 two functions, among others, from a library called `ctype`, that we can use. (And we would only know this from reading the documentation, for that library, that other people wrote.)
- We can use a command-line program, `man`, to read manual information for other programs, if it exists. For example, we can run `man toupper` to see some documentation about that function. Then, we'll see that `toupper` will return the character as-is, if it's not a lowercase letter, and so we can simply have:

```
for (int i = 0, n = strlen(s); i < n; i++)
{
    printf("%c", toupper(s[i]));
}
```

Command-line arguments

- We've used programs like `make` and `clang`, which take in extra words after their name in the command line. It turns out that programs of our own, can also take in *command-line arguments*.
- In `argv0.c`, we change what our `main` function looks like:

```
#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");
    }
}
```

- `argc` and `argv` are two variables that our `main` function will now get, when our program is run from the command line. `argc` is the argument count, or number of arguments, and `argv` is an array of strings that are the arguments. And the first argument, `argv[0]`, will be the name of our program (the first word typed, like `./hello`). In this example, we'll check if we have two arguments, and print out the second one if so.

- We can print every argument, one at a time:

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

int main(int argc, string argv[])
{
    for (int i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
}
```

- We can print out each character of each argument, too:

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

int main(int argc, string argv[])
{
    for (int i = 0; i < argc; i++)
    {
        for (int j = 0, n = strlen(argv[i]); j < n; j++)
        {
            printf("%c\n", argv[i][j]);
        }
```

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

- With `argv[i]`, we get the current argument from the array of arguments, and with `argv[i][j]`, we get a character from that string.

Encryption

- If we wanted to send a message to someone, we might want to *encrypt*, or somehow scramble that message so that it would be hard for others to read. The original message is called *plaintext*, and the encrypted message is called *ciphertext*.
- A message like `HI!` could be converted to ASCII, `72 73 33`. But anyone would be able to convert that back to letters.
- We look at examples, from World War I, to a poem about Paul Revere's ride, of historical codes.
- *Encryption* generally requires another input, in addition to the plaintext. A *key* is needed, and sometimes it is simply a number, that is kept secret. With the key, plaintext can be converted, via some algorith, to ciphertext, and vice versa.
- For example, if we wanted to send a message like `I L O V E Y O U`, we can first convert it to ASCII: `73 76 79 86 69 89 79 85`. Then, we can encrypt it with a key of just `1` and a simple algorithm, where we just add the key to each value: `74 77 80 87 70 90 80 86`. Then, someone converting that ASCII back to text will see `J M P W F Z P V`. To decrypt this, someone might have to guess the value of each letter, through trial-and-error, but they wouldn't be sure, without knowing the key. In fact, this algorithm is known as a Caesar cipher.

Exit codes

- It turns out that we can indicate errors in our program, by returning a value from our `main` function:

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

int main(int argc, string argv[])
{
    if (argc != 2)
    {
        printf("missing command-line argument\n");
        return 1;
    }
    printf("hello, %s\n", argv[1]);
    return 0;
}
```

- The return value of `main` in our program is called an exit code, and we can actually see this in our command line. If we ran this program with `./exit`, we can then type `echo $?`, which will print the last program's return value.
- As we write more complex programs, error codes like this will help us determine what went wrong, even if it's not visible or meaningful to the user.

Sorting

- With arrays, we can solve more interesting problems than before. We can think of arrays like lockers, where, behind the doors of each locker, is some value, like an integer or character. Indeed, computers can only look at one locker, or value at a time.
- If we had a list of numbers, and we wanted to find a number in that list, the best we could do is look through it, one at a time, or randomly.
- But if we knew the list was sorted, we could look in the middle first, and move left or right accordingly.
- With some volunteers, we demonstrate how we might sort a list.
- Our volunteers start in the following random order:

```
6 5 1 3 7 8 4 2
```

- We look at the first two numbers, and swap them so they are in order:

```
5 6 1 3 7 8 4 2
```

- Then we look at the next pair, `6` and `1`, and swap them:

```
5 1 6 3 7 8 4 2
```

- We repeat this, until, after our first pass, the largest number ended up furthest on the right:

```
5 1 6 3 7 4 2 8
```

- (In the lecture, the 1 accidentally moved a spot too far!)

- We repeat this, and every time we make a pass, the next-largest number ends up next-furthest to the right:

```
1 5 3 6 4 2 7 8
```

- Eventually, our list becomes fully sorted. The first time, we compared 7 pairs of numbers. The second time, we compared 6 pairs.
- We shuffle our numbers again:

```
2 4 8 5 7 1 3 6
```

- And this time, we look for the smallest number each time, as we go down the list, and put that to the far left:

```
1 4 8 5 7 2 3 6
1 2 8 5 7 4 3 6
1 2 3 5 7 4 8 6
```

- Each time, we select the smallest number and swap it with the number that's in the furthest left part of the unsorted part of the list.

- With this algorithm, we still pass through the list *n - 1* times, since there are *n* people, and we do have to compare each number with the smallest number we've seen thus far.
- Let's try to figure this out a little more formally. The first algorithm, bubble sort, involved comparing pairs of numbers next to each other, until the largest bubbled up to the right. We might write that in pseudocode as:

```
repeat until no swaps
```

```
    for i from 0 to n-2
        if i'th and i+1'th elements out of order
            swap them
```

- And selection sort might be as follows:

```
for i from 0 to n-1
    find smallest element between i'th and n-1'th
    swap smallest with i'th element
```

- For the first pass, we needed to make `n - 1` comparisons, to find the smallest number. Then, in each of the following passes, we made one less comparison, since we had already moved some numbers to the left:

```
(n - 1) + (n - 2) + ... + 1
n(n - 1)/2
(n^2 - n)/2
n^2 / 2 - n/2
```

  - Each line simplifies to the next, and eventually, we get `n^2 / 2 - n/2` as the number of comparisons we need to make. In computer science, we can use *O*, big *O* notation, to simplify that further, and say that our algorithm takes *O*(*n*^2) steps, "on the order of *n* squared". This is because, as *n* gets bigger and bigger, only the *n*^2 term matters.
  - For example, if *n* were 1,000,000, we would get:

```
n^2 / 2 - n/2
1,000,000^2 / 2 - 1,000,000/2
500,000,000,000 - 500,000
499,999,500,000
```

    - which is on the same order of magnitude as n^2.

- It turns out, there are other common orders of magnitude:
  - $O(n^2)$
  - $O(n \log n)$
  - $O(n)$
  - $O(\log n)$
  - $O(1)$
- Searching through a phone book, one page at a time, has $O(n)$ running time, since we need one step for every page. Using binary search would have $O(\log n)$ running time, since we divided the problem in half each time.
- Let's take another array of numbers, but this time, use an empty array of the same size as our working space:

```
4 2 7 5 6 8 3 1
_ _ _ _ _ _ _ _
```

- Since we have 8 numbers, let's look at the first half, the first 4. We'll sort that recursively, and look at just the left half of that. With 2 numbers, `4 2`, we look at the left half of that (sorted), and the right half of that (sorted), and combine them by sorting them, `2 4`. We'll move them to our second array:

```
_ _ 7 5 6 8 3 1
2 4 _ _ _ _ _ _
```

- We repeat this, for the right half of the original half:

```
_ _ | _ _ 6 8 3 1
2 4 | 5 7 _ _ _ _
```

- Then, we merge those halves, to get a sorted left half:

```
_ _ _ _ 6 8 3 1
_ _ _ _ _ _ _ _
2 4 5 7 _ _ _ _
```

- We repeat, for the right half:

```
_ _ _ _ | _ _ _ _
_ _ _ _ | _ _ _ _
2 4 5 7 | 1 3 6 8
```

- And now, we can merge both halves:

```
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _
1 2 3 4 5 6 7 8
```

- Each number had to move 3 times, since we divided 8 by 2 three times, or log *n* times. So this algorithm takes $O(n \log n)$ to sort a list.
- We look at demos like Sorting Algorithms Animations and What different sorting algorithms sound like to conclude.