Table of Contents

Week 1

Scratch vs. C

- Now that we've explored some basic programming concepts with Scratch, we can try to use the same ideas with a more traditional language, C.
- Recall that last week, to run our program in Scratch, we would begin with a block that read `when green flag clicked`.
- Our example of having Scratch say `hello, world` can be translated to the following C:

```
#include <stdio.h>

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

  - `printf` is the equivalent of `say` in Scratch, and it will print whatever is inside the parentheses.
  - We notice a bit of syntax, like the double quotes and the semicolon, but we can focus on one piece at a time.
- In Scratch, `say` was a function that took an argument, or parameter, and the equivalent line in C is:

```
printf("hello, world\n");
```

  - The `\n` prints a new line, like pressing enter after typing out that message.

- And in the case of loops, in Scratch we might have a `forever` block that does something over and over again. In C, we would have this:

```
while (true)
{
    printf("hello, world\n");
}
```

  - The statements inside the braces will be executed again and again `while` the expression inside the parentheses is true, and since `true` will always be true, the loop will continue forever.

- To repeat a loop a certain number of times, we have something a little more complex:

```
for (int i = 0; i < 50; i++)
{
    printf("hello, world\n");
}
```

  - We'll come back to this again in a bit, but know that `for` is the special word we use to start a loop.

- In Scratch, we used blocks for variables like `set [i] to [0]` to store values. To do the same in C, we'd do this:

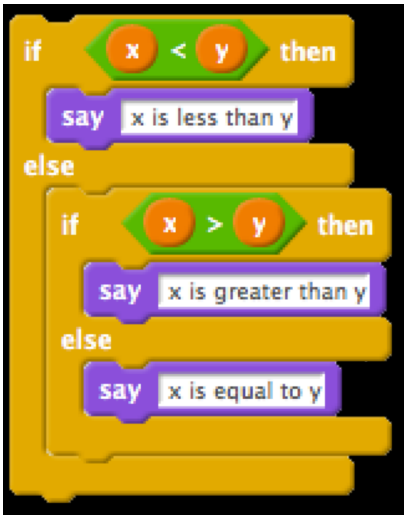
```
int i = 0;
```

  - `int` stands for integer, where we are creating a variable to store whole numbers, `i` is the name of our new variable, and `0` is the value we will initially set it to.
  - And the semicolon just ends this statement.

- Boolean expressions were questions that would either be true or false, and in Scratch they might have looked like `i < 50`, is `i` less than `50`. And in C, it's just as simple:

```
i < 50
```

- `x < y`, too, is the same in Scratch and C, as long as `x` and `y` are both variables we've created and assigned values to.
- We can use conditions, too, to create forks in the road. Recall that last time we demonstrated this in Scratch:



- The same might look even a little simpler in C:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

- Scratch had lists, too, where we could store multiple values together. The equivalent in C is something called an array, where we store lots of items back-to-back.
- Last time in Scratch we saw blocks like `item (1) of [argv]`, which took the first item from a list called `argv`, and in C (we start counting from 0 in C, since that's the smallest non-negative value we can represent), we would use `argv[0]`.

---

hello, C

- So, going back to our original example:

```
#include <stdio.h>

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

- `main` is the equivalent of `when green flag clicked`, and marks the *main* chunk of code that should be executed.

- To go from this code, which is readable to humans, need to be translated first to **machine code**, that look something like this:

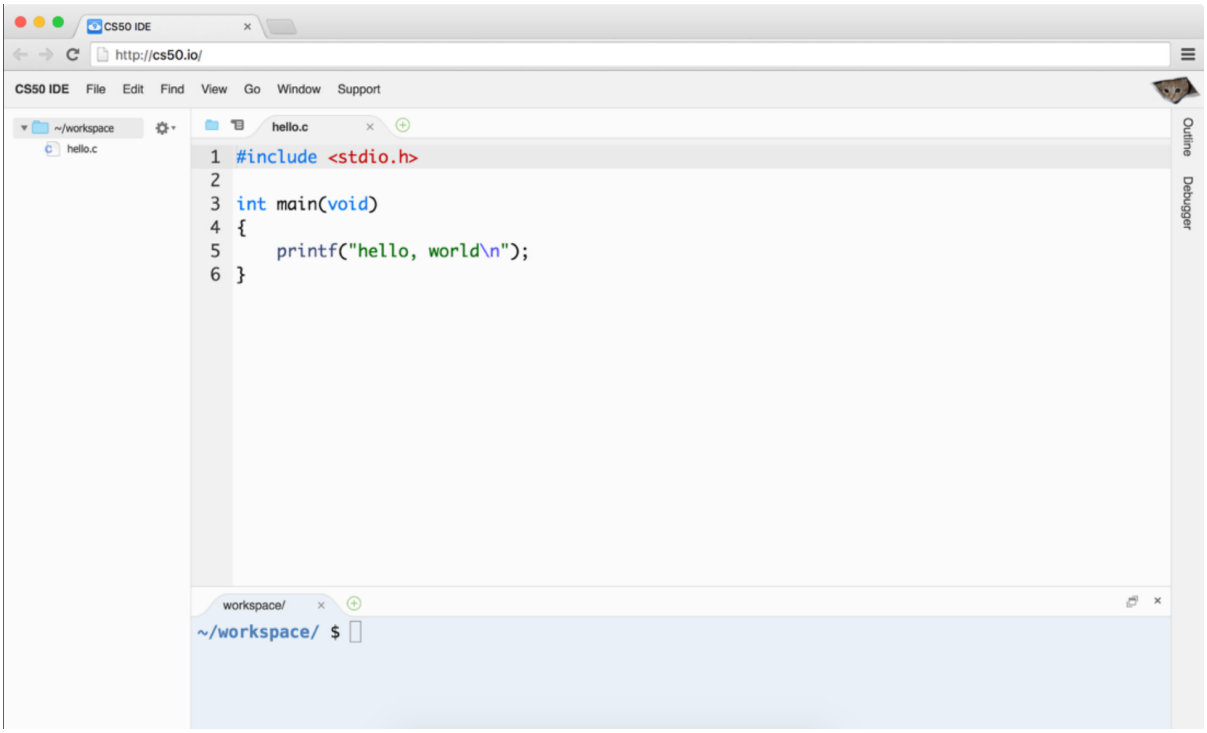
```
01111111 01000101 01001100 01000110 00000010 00000001 00000001 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000010 00000000 00111110 00000000 00000001 00000000 00000000 00000000
10110000 00000101 01000000 00000000 00000000 00000000 00000000 00000000
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
11010000 00010011 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 01000000 00000000 00111000 00000000
00001001 00000000 01000000 00000000 00100100 00000000 00100001 00000000
00000110 00000000 00000000 00000000 00000101 00000000 00000000 00000000
01000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
01000000 00000000 01000000 00000000 00000000 00000000 00000000 00000000
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000
11111000 00000001 00000000 00000000 00000000 00000000 00000000 00000000
00001000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000011 00000000 00000000 00000000 00000100 00000000 00000000 00000000
00111000 00000010 00000000 00000000 00000000 00000000 00000000 00000000
00111000 00000010 01000000 00000000 00000000 00000000 00000000 00000000
00111000 00000010 01000000 00000000 00000000 00000000 00000000 00000000
00011100 00000000 00000000 00000000 00000000 00000000 00000000 00000000
...
```

- You'll be asked to write this from memory for the test, so start memorizing now! Just kidding.

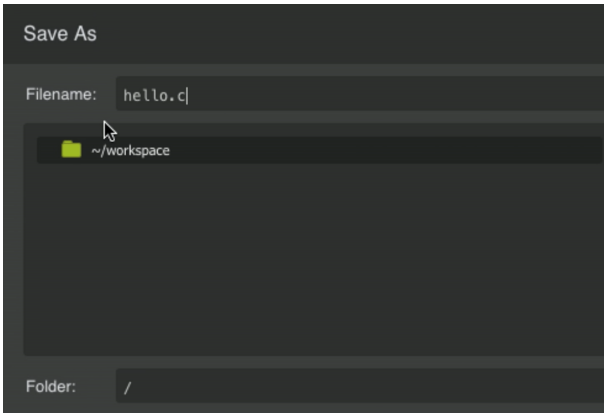
- But you do need to remember that, at the end of the day, computers only operate with binary, `0`s and `1`s, and so each of these patterns of `0`s and `1`s represent a special instruction to the CPU, central processing unit, of the computer. Some patterns will mean "print this to the screen," some patterns "add these two numbers," or any of a large number of operations.

- We don't need to create this by hand, since there is software called **compilers**, which take code written in C and readable by humans (**source code**), and translates it to machine code.

- We're all using slightly different operating systems on our computer, like macOS or Windows or others, and just so everyone is on the same page (get it?), we'll use a cloud-based integrated development environment called CS50 IDE (https://cs50.io).
    - What does that actually mean? This is a web-based programming environment based on a platform called Cloud9, which allowed us to pre-install standard software and configure it the same way for everyone.
- We can visit the page (heh), create a free account, and see something like this:



    - On the left is where we can see our files in the cloud, on the right is where we edit our code, and the strange box at the bottom is called a `terminal`, a command-line interface (CLI) where we can type in commands directly to our computer. In this case, these commands will be sent to the computer in the cloud, and we'll use it to compile our code or run our programs.
- We'll jump right in with making our first program, and first we'll save a file using `File > Save as`:



- Now we have a file called `hello.c`, since files with source code for C end in `.c` by convention. We'll type in the same example to the editor:

```
#include <stdio.h>

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

- On our own computers, we might be used to double-clicking an icon of a program to rn it. The cloud computers we use run an operating system called Linux, which oftentimes do come with a graphical user interface (GUI), but is better known for its command-line interface, and so we'll use that.
- To do that, in the bottom panel we'll type `clang hello.c` as follows:

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

  - `clang` (as in C language) is a compiler, so we're just asking it to compile our `hello.c` file.
  - `~/workspace/` just means that we're in the folder called `workspace` in which `hello.c` lives (we can verify this by looking at the file list on the left), and `$` is just a prompt, indicating that we'll be typing our command there.
- After we press enter, we don't see anything in particular:

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

- It turns out that the default name for compiled programs is `a.out`, and we can run it with:

```
~/workspace/ $ ./a.out
hello, world
~/workspace/ $
```

  - Notice that it printed what we wanted successfully, and also moved our cursor to the next line. Recall that our source code had the extra `\n` to create this new line.
- If we were to remove that from our source code, and remember to save, we can recompile our program and see this:

```
~/workspace/ $ clang hello.c
~/workspace/ $ ./a.out
hello, world~/workspace/ $
```

  - It still worked, but our next prompt ended up at the same line.

- So we change it back, and remember, every time we change our source code we also need to recompile it.
- We can also ask our new friend clang to save the program as something with a nicer name, by passing it command-line arguments (also called flags or switches):

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

  - So in the middle we've added `-o` for `output` and specified it to be `hello`.
- So now we can press enter, and be able to run `./hello`.
- But this seems like it'll be more and more of a hassle as we have bigger, more complex programs. So there's actually yet another program, called `make`, that we'll use.
- But first, some cleanup. We'll run `ls` to show all the files in our `workspace` folder:

```
~/workspace/ $ ls
a.out*  hello*  hello.c
```

  - This lists the files, which matches what we see on the left side. We could delete it with the GUI on the left side, but we could also:

```
~/workspace/ $ rm a.out
```

  - This command, `rm` removes a file. It asks us to confirm, and we'll type `y` for yes.
- Executable programs, that we can run, are also shown by `ls` with a `*` and in a special color.
- So we can run `make`:

```
~/workspace/ $ make hello
```

  - This program will create a `hello` executable program from a source code file called `hello.c`, all of which it infers from that one word.
  - After we press enter, we see a really long command that starts with `clang` but passes in a lot more options (which we'll eventually need), but notice that we again will have a `hello` file in our directory that we can run.
- Other Linux command-line, er, commands include:
  - `cd` for change directory, to move around to different folders
  - `ls` which we've seen
  - `mkdir` to make a directory

- `rm` to remove a file
- `rmdir` to remove a directory

The CS50 Library

- So let's build more interesting programs.
- To get inputs from users, we've implemented some custom functions:
  - `get_char`
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long_long`
  - `get_string`
- We'll create a file called `string.c` (a string is just a sequence of characters):

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

int main(void)
{
    string name = get_string();
    printf("hello, %s\n", name);
}
```

  - The first lines include libraries, or groups of custom functions we can use in our own code. `cs50.h` contains the custom functions above, and `stdio.h` (Standard Input and Output) contains basic C functions like `printf`. `cs50.h` also includes a special type of variable called `string`, which C doesn't have built in.
  - In our `main` functions, we first create a `string` variable called `name`, and use a function called `get_string`. We need to end it with `()` because we want to run the function, even if we don't have any arguments to pass to it. The results of `get_string` will then be stored back into `name`.
  - Then in the next line, we'll use a strange syntax, `%s`, to include the value of a variable into what gets printed out. If we just used `printf("hello, name\n")`, it would literally just print `hello, name`. But with `%s` we can include `name` as a variable.

- Now we can type `make string`, and `./string`. But it looks like nothing is happening. Well, it's just waiting for our input, waiting to get a string from us. So we'll type in `David`, press enter, and see that it replies with `hello, David` like we might expect. Cool!
- But let's make it a little less confusing. Before we `get_string`, let's print some instructions out:

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

int main(void)
{
    printf("Name: ");
    string name = get_string();
    printf("hello, %s\n", name);
}
```

  - Then the prompt that waits will be next to `Name: `.
- We've built a simple program step by step, line by line, with baby steps, and generally this is a good strategy for writing programs, since we can check our work at each stage and make sure what we've done so far works as expected.
- Let's do something a little different:

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

int main(void)
{
    int i = get_int();
    printf("hello, %i\n", i);
}
```

  - Now we're getting an integer, storing it in a variable called `i`, and giving it to `printf` as a `%i` since `%s` substitutes a string but we know `i` is an integer.
  - If we compile this, run it, and type in something like `David`, it will tell us to `Retry` until we type in something that's just a number.
- These first examples will take us (slowly but thoroughly!) through the basics, so that we can eventually build more exciting programs.
- In fact, with C we have much more control over what our computer is doing, and look under the hood a lot more easily.
- Let's write another short program:

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

int main(void)
{
    printf("x is ");
    int x = get_int();

    printf("y is ");
    int y = get_int();

    int z = x + y;

    printf("sum of x and y is %i\n", z);
}
```

  - So we've gotten two numbers from the user, `x` and `y`, made a new variable `z` that contains the sum, and printed it out.

- But we can make it a little simpler without creating a whole variable and naming it:

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

int main(void)
{
    printf("x is ");
    int x = get_int();

    printf("y is ");
    int y = get_int();

    printf("sum of x and y is %i\n", x + y);
}
```

- But let's do a little more math:

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

int main(void)
{
    printf("x is ");
    int x = get_int();

    printf("y is ");
    int y = get_int();

    printf("%i plus %i is %i\n", x, y, x + y);
    printf("%i minus %i is %i\n", x, y, x - y);
    printf("%i times %i is %i\n", x, y, x * y);
    printf("%i divided by %i is %i\n", x, y, x / y);
    printf("remainder of %i divided by %i is %i\n", x, y, x % y);
}
```

  - Notice the operations we use and how they are translated to C. `%` in particular, gets us the remainder when the first number is divided by the second.

- Well let's compile, run, and type in `1` and `10` for `x` and `y`:

```
...
1 divided by 10 is 0
...
```

  - Everything else looks good, except for that one line! The correct answer should be `0.1`, right? But remember that we're working with integers `x` and `y` and printing out integers with `%i`, so numbers after the decimal point get truncated, or cut off. (`0.1` ends up being `0`.)

- So we can fix it by using a variable type called `float`, for floating-point values (real numbers):

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

int main(void)
{
    printf("x is ");
    float x = get_float();

    printf("y is ");
    float y = get_float();

    printf("%f divided by %f is %f\n", x, y, x / y);
}
```

- Now our math is correct!

Data Types

- There are lots of data types we'll be using:
  - `bool` for a Boolean value (true or false)
  - `char` for a single character
  - `double` for a large real number with more bits than a normal `float`
  - `float`
  - `int`
  - `long long` for a large whole number with more bits than a normal `int`
  - `string`
- Let's write another program to show us how many bytes are used for each of these data types:

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

int main(void)
{
    printf("bool is %lu\n", sizeof(bool));
    printf("char is %lu\n", sizeof(char));
    printf("double is %lu\n", sizeof(double));
    printf("float is %lu\n", sizeof(float));
    printf("int is %lu\n", sizeof(int));
    printf("long long is %lu\n", sizeof(long long));
    printf("string is %lu\n", sizeof(string));
}
```

```
bool is 1
char is 1
double is 8
float is 4
int is 4
long long is 8
string is 8
```

  - It turns out, for our specific Cloud9 operating system, a `bool` is a whole byte, a character is 8 bits too, and so on.
  - But wait, strings are just 8 bytes long? Not to worry, we'll realize how a string can be longer than that, soon enough.

- And we have a limited number of bytes in memory, so we can only store a finite number of digits. In fact, imagine that we have a binary number with 8 bits:

```
1 1 1 1 1 1 1 0
```

- If we added `1` to that, we'll get `1 1 1 1 1 1 1 1`, but what happens if we add another `1` to that? We'll start carrying over all the `0`s to get `0 0 0 0 0 0 0 0`, but we don't have an extra bit to the left to actually store that larger value.
- In programs, we see this behavior with integers:

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

int main(void)
{
    int n = 1;
    for (int i = 0; i < 64; i++)
    {
        printf("%i\n", n);
        n = n * 2;
    }
}
```

- We know that `int`s have 4 bytes set aside for them, which is 32 bits, so 2^32 possible values, which is about 4 billion values. But half of them are negative, so the highest positive value is just about 2 billion.
- So in this program we're starting with `n` as `1`, and doubling it each time:

```
n is 1
n is 2
n is 4
n is 8
...
n is 1073741824
n is -2147483648
n is 0
n is 0
...
```

- So now we know that, eventually, as our number gets too big for the number of bits set aside for it, we'll have something bad happen. This is called an **overflow**.
- We can change `n` to `long long` and print it out with `%lld`, but at the last step we still see it "wrap around" to a negative number.

- In the real world, certain games might use an integer for values, but bugs might appear (http://www.geek.com/games/why-gandhi-is-always-a-warmongering-jerk-in-civilization-1608515/) as they wrap around!
- More serious bugs could occur with jets shutting off (https://www.engadget.com/2015/05/01/boeing-787-dreamliner-software-bug/), too.

- Another bug can arise when we have **floating-point imprecision**. Remember that floats have a finite number of bits. But there are an infinite number of real numbers, so a computer has to round and represent some numbers inaccurately.
- Let's write a simple program to see this firsthand:

```
#include <stdio.h>

int main(void)
{
    printf("%.55f\n", 1.0 / 10.0);
}
```

  - The new part, `%.55f`, just tells `printf` to print 55 digits after the decimal point.
  - And we used `1.0` and `10.0` just to ensure that the types are floats (since we didn't specify them as variables). Alternatively, we could have used `(float) 10` to cast, or specify, `10` to be a floating-point 10 and not an integer 10.
- Now when we run this, we get:

```
0.1000000000000000055511151231257 8...
```

  - Hm, it turns out the closest approximation a computer can make to `0.1` is that number.
- We watch a quick clip (https://video.cs50.net/2016/fall/lectures/1?t=78m53s) on imprecision in the real world.
- Just to recap, we now know that there are a few different data types that we can use, and also print with various symbols:
  - `%c`
  - `%f`
  - `%d`
  - `%i`
  - `%lld`
  - `%s`
  - `...`
- And in addition to `\n` for a new line, we can use certain **escape sequences**, symbols we can type, for `printf` to print tabs or quotes or others:
  - `\a`

- `\n`
- `\r`
- `\t`
- `\'`
- `\"`
- `\\`
- `\0`
- `…`

More C

- Let's write a program that uses more of the same ideas from Scratch, so we can build more complex programs in C:

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

int main(void)
{
    int i = get_int();
    if (i < 0)
    {
        printf("negative\n");
    }
    else if (i > 0)
    {
        printf("positive\n");
    }
    else
    {
        printf("equal\n");
    }
}
```

  - Since we know that by the last `else` `i` is neither greater or less than `0`, we don't need to specify `else if (i == 0)`. And note that we use `==` to compare two variables or values, since a single `=` assigns one value to the other.

- We can play with some more logic:

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

int main(void)
{
    char c = get_char();
    if (c == 'Y' || c == 'y')
    {
        printf("yes\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("no\n");
    }
    else
    {
        printf("error\n");
    }
}
```

  - We get a character `c`, and compare it to either `Y` or `y`, or `N` or `n`. We use `||` in C to represent a logical **or**, where only one of the expressions need to be true for that condition to be followed and `&&` for **and**, where both expressions must be true.
  - Note that we use single quotes around characters, to distinguish them between strings of a single character, which we use double quotes to indicate.
- Let's explore a different way to implement the same program:

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

int main(void)
{
    char c = get_char();
    switch (c)
    {
        case 'Y':
        case 'y':
            printf("yes\n");
            break;
        case 'N':
        case 'n':
            printf("no\n");
            break;
        default:
            printf("error\n");
            break;
    }
}
```

  - Here we're using something called a **switch**, which has various cases that, when it matches one of them, will execute the statements below it. It will continue until it reaches a `break;` statement to break out of the switch.

- With our compiler and editor to help us explore, we could try removing all the `break;` statements to see what happens:

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

int main(void)
{
    char c = get_char();
    switch (c)
    {
        case 'Y':
        case 'y':
            printf("yes\n");

        case 'N':
        case 'n':
            printf("no\n");

        default:
            printf("error\n");

    }
}
```

- And in this case (heh), all the statements below the first case that matches will be executed.

- Let's dive deeper into how to design code, by making our own custom function:

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

int main(void)
{
    string s = get_string();
    print_name(s);
}

void print_name(string name)
{
    printf("hello, %s\n", name);
}
```

- Notice that below our `main` function, we define a new function called `print_name` which takes in a `string` that it can refer to as `name`, and it returns no value, so we call the type of value it returns `void`. (`main`, on the other hand, returns a value of type `int`. More on that another day.)

- But now if we try to `print_name(s)` in our `main` function, we still get an error. And that's because the compiler reads from top to bottom, in order, so at the time `main` calls `print_name`, it doesn't exist yet. So we need to declare it with something called a **prototype** first:

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

void print_name(string name);

int main(void)
{
    string s = get_string();
    print_name(s);
}

void print_name(string name)
{
    printf("hello, %s\n", name);
}
```

  - That just defines the function, what it will take, and what it will return, and later our compiler will look for it and be able to link it correctly.

- And within our libraries `cs50.h` and `stdio.h` are similar prototypes, one line statements that define functions like `get_string` and `printf`, with their implementations in other files.

- And to demonstrate return values, we can write a program like this:

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

int square(int n);

int main(void)
{
    printf("x is ");
    int x = get_int();
    printf("x^2 is %i\n", square(x));
}

int square(int n)
{
    return n * n;
}
```

- `square` is a function that takes an `int` `n`, and returns something of type `int`. Within the function, it will just `return n * n`.
- Now we can use `square(x)` in our function, and `printf` the result like any other `int` since we know that's what the function will return.

- If we go back to our original `get_string` function, we can realize that `get_string` probably has a prototype that looks something like `string get_string(void)`, since it takes no arguments but returns a string for us to use:

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

int main(void)
{
    string s = get_string();
    printf("hello, %s\n", s);
}
```

- Let's improve our `cough` example from Scratch last time, step by step:

```
#include <stdio.h>

int main(void)
{
    printf("cough\n");
    printf("cough\n");
    printf("cough\n");
}
```

- Here we want to print out `cough` three times, so we've copied and pasted the code.

- So we can replace that with a loop, since each line is exactly the same:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("cough\n");
    }
}
```

- But we can write our own function now, so we can reuse it wherever we'd like:

```c
#include <stdio.h>

void cough(void);

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        cough();
    }
}

void cough(void)
{
    printf("cough\n");
}
```

    - It may seem like we've worked too hard for this particular example, but as programs get more and more complex we will need to create these blocks and custom functions.

- For example, if we wanted to `cough` a number of times, and also `sneeze` a certain number of times, we should be able to do that quite simply:

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

void cough(int n);
void say(string word, int n);
void sneeze(int n);

int main(void)
{
    cough(3);
    sneeze(3);
}

void cough(int n)
{
    say("cough", n);
}

void say(string word, int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("%s\n", word);
    }
}

void sneeze(int n)
{
    say("achoo", n);
}
```

- Notice that `main` can simply call `cough` and `sneeze` with the number of times it would like that action, and those functions call `say`, which has the actual shared implementation of a `for` loop and `printf`.
- Notice that `say`, too, is now taking two arguments, one of which is a `string` and one an `int`, and so each time it is called, both arguments need to be passed in.

- We call this concept **abstraction**, where we build layers that different people can work on, but will work together in the end since each piece will do what it is supposed to (if it's implemented correctly, of course!).
- And in fact, we've been using abstraction this whole time as we called `get_string` or `printf`, since we don't know how those functions are actually implemented in the other files that we are including, but we can use them since we know what they will do.

- So let's go back to what `make` is actually doing. Compiling, in fact, includes several steps such as:
  - preprocessing
    - Lines that start with `#`, like `#include`, are preprocessed. `#include` in particular makes our compiler look for the file somewhere on our computer and literally include them inside our files (like copying and pasting them in).
  - compiling
    - We can run `clang -S hello.c` to see our C program compiled into another language called assembly language (which you'll see more of if you take Cheng (http://ummcheng.com/)'s favorite class during his time at Harvard, CS61!) that has the very simple instructions that CPUs can understand (like adding numbers, moving values in memory, etc), but in text format so humans can attempt to decipher it too.
  - assembling
    - The intermediate assembly code is then translated into machine code, 0s and 1s, that the CPU can actually understand.
  - linking
    - This final step takes the machine code of our program, and the machine code of all the libraries we included earlier and are using, and combines them so that the final program has all the pieces we need. (The preprocessed files we `#include` are just header files, which only have prototypes of the functions we want to use. The actual implementation and thus machine code is separate and lives in other files.)
- So there was a lot there going on, but hopefully we can start getting more and more comfortable with, and understand, how something "simple" actually works!
- Eventually, you'll be able to recognize patterns, and pick up on design and abstraction to write good programs.
- We'll focus on cryptography, or scrambling information, next week. We'll take steps each week so we can write more and more interesting programs as we go along. Until next time!