This is CS50x

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/in/malan/)
(https://orcid.org/0000-0001-5338-2522)
(https://www.quora.com/profile/David-J-Malan)
(https://www.reddit.com/user/davidjmalan) (https://twitter.com/davidjmalan)

Lecture 1

- C
- CS50 IDE
- Compiling
- Functions and arguments
- main, header files
- Tools
- Commands
- Types, format codes,
- Operators, limitations, truncation
- Variables, syntactic sugar
- Conditions
- Boolean expressions, loops
- Abstraction
- Mario
- Memory, imprecision, and overflow

C

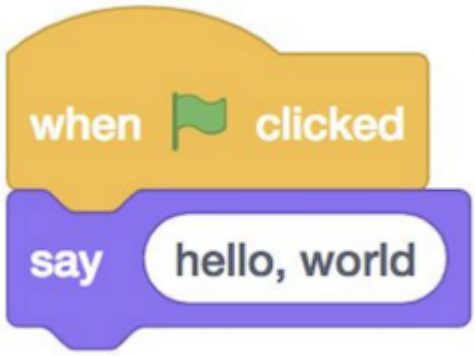
- Today we'll learn a new language, **C**: a programming language that has all the features of

Scratch and more, but perhaps a little less friendly since it's purely in text:

```c
#include <stdio.h>

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

- Though at first, to borrow a phrase from MIT, trying to absorb all these new concepts may feel like drinking from a fire hose, be assured that by the end of the semester we'll be empowered by and experienced at learning and applying these concepts.
- We can compare a lot of the programming features in C to blocks we've already seen and used in Scratch. The details of the syntax are far less important than the ideas, which we've already been introduced to.
- In our example, though the words are new, the ideas are exactly as same as the "when green flag clicked" and "say (hello, world)" blocks in Scratch:

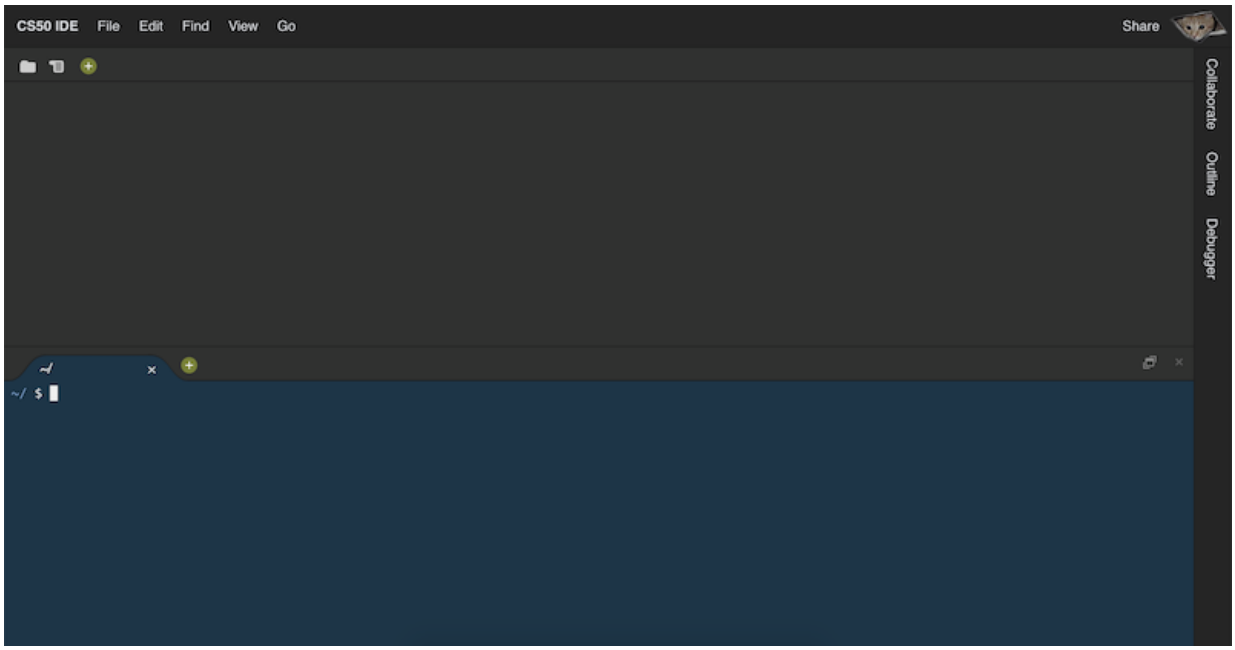


- When writing code, we might consider the following qualities:
  - **Correctness**, or whether our code works correctly, as intended.
  - **Design**, or a subjective measure of how well-written our code is, based on how efficient it is and how elegant or logically readable it is, without unnecessary repetition.
  - **Style**, or how aesthetically formatted our code is, in terms of consistent indentation and other placement of symbols. Differences in style don't affect the correctness or meaning of our code, but affect how readable it is visually.

CS50 IDE

- To start writing our code quickly, we'll use a tool for the course, **CS50 IDE** (https://ide.cs50.io/), an *integrated development environment* which includes programs and features for writing code. CS50 IDE is built atop a popular cloud-based IDE used by general programmers, but with additional educational features and customization

- We'll open the IDE, and after logging in we'll see a screen like this:



  - The top panel, blank, will contain text files within which we can write our code.
  - The bottom panel, a **terminal** window, will allow us to type in various commands and run them, including programs from our code above.

- Our IDE runs in the cloud and comes with a standard set of tools, but know that there are many desktop-based IDEs as well, offering more customization and control for different programming purposes, at the cost of greater setup time and effort.
- In the IDE, we'll go to File > New File, and then File > Save to save our file as `hello.c`, indicating that our file will be code written in C. We'll see that the name of our tab has indeed changed to `hello.c`, and now we'll paste our code from above:

```c
#include <stdio.h>

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

- To run our program, we'll use a CLI, or **command-line interface**, a prompt at which we need to enter text commands. This is in contrast to a **graphical user interface**, or GUI, like Scratch, where we have images, icons, and buttons in addition to text.

Compiling

- In the terminal in the bottom pane of our IDE, we'll **compile** our code before we can run it. Computers only understand binary, which is also used to represent instructions like printing something to the screen. Our **source code** has been written in characters we can read, but it needs to be compiled: converted to **machine code**, patterns of zeros and ones that our computer can directly understand.

- A program called a **compiler** will take source code as input and produce machine code as output. In the CS50 IDE, we have access to a compiler already, through a command called **make**. In our terminal, we'll type in `make hello`, which will automatically find our `hello.c` file with our source code, and compile it into a program called `hello`. There will be some output, but no error messages in yellow or red, so our program compiled successfully.
- To run our program, we'll type in another command, `./hello`, which looks in the current folder, `.`, for a program called `hello`, and runs it.
- The `$` in the terminal is an indicator of where the prompt is, or where we can type in more commands.

Functions and arguments

- We'll use the same ideas we've explored in Scratch.
- **Functions** are small actions or verbs that we can use in our program to do something, and the inputs to functions are called **arguments**.
    - For example, the "say" block in Scratch might have taken something like "hello, world" as an argument. In C, the function to print something to the screen is called `printf` (with the `f` standing for "formatted" text, which we'll soon see). And in C, we pass in arguments within parentheses, as in `printf("hello, world");`. The double quotes indicate that we want to print out the letters `hello, world` literally, and the semicolon at the end indicates the end of our line of code.
- Functions can also have two kinds of outputs:
    - **side effects**, such as something printed to the screen,
    - and **return values**, a value that is passed back to our program that we can use or store for later.
        - The "ask" block in Scratch, for example, created an "answer" block.
- To get the same functionality as the "ask" block, we'll use a **library**, or a set of code already written. The CS50 Library will include some basic, simple functions that we can use right away. For example, `get_string` will ask the user for a **string**, or some sequence of text, and return it to our program. `get_string` takes in some input as the prompt for the user, such as `What's your name?`, and we'll have to save it in a variable with:

```
string answer = get_string("What's your name? ");
```

    - In C, the single `=` indicates **assignment**, or setting the value on the right to the variable on the left. And C will call the `get_string` function in order to get its output first.
    - And we also need to indicate that our variable named `answer` has a **type** of string, so our program knows to interpret the zeros and ones as text.

- Finally, we need to remember to add a semicolon to end our line of code.

- In Scratch, we also used the "answer" block within our "join" and "say" blocks. In C, we'll do this:

```
printf("hello, %s", answer);
```

  - The `%s` is called a **format code**, which just means that we want the `printf` function to substitute a variable where the `%s` placeholder is. And the variable we want to use is `answer`, which we give to `printf` as another argument, separated from the first one with a comma. (`printf("hello, answer")` would literally print out `hello, answer` every time.)

- Back in the CS50 IDE, we'll add what we've discovered:

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

int main(void)
{
    string answer = get_string("What's your name? ");


    printf("hello, %s", answer);
}
```

  - We need to tell the compiler to include the CS50 Library, with `#include <cs50.h>`, so we can use the `get_string` function.
  - We also have an opportunity to use better style here, since we could name our `answer` variable anything, but a more descriptive name will help us understand its purpose better than a shorter name like `a` or `x`.

- After we save the file, we'll need to recompile our program with `make hello`, since we've only changed the source code but not the compiled machine code. Other languages or IDEs may not require us to manually recompile our code after we change it, but here we have the opportunity for more control and understanding of what's happening under the hood.

- Now, `./hello` will run our program, and prompt us for our name as intended. We might notice that the next prompt is printed immediately after our program's output, as in `hello, Brian~/ $`. We can add a new line after our program's output, so the next prompt is on its own line, with `\n`:

```
printf("hello, %s\n", answer);
```

  - `\n` is an example of an **escape sequence**, or some text that represents some other text.

main, header files

- The “when green flag clicked” block in Scratch starts what we would consider to be the main program. In C, the first line for the same is `int main(void)`, which we’ll learn more about over the coming weeks, followed by an open curly brace `{`, and a closed curly brace `}`, wrapping everything that should be in our program.

```
int main(void)
{

}
```

  - We’ll learn more about ways we can modify this line in the coming weeks, but for now we’ll simply use this to start our program.
- **Header files** that end with `.h` refer to some other set of code, like a library, that we can then use in our program. We *include* them with lines like `#include <stdio.h>`, for example, for the *standard input/output* library, which contains the `printf` function.

Tools

- With all of the new syntax, it’s easy for us to make mistakes or forget something. We have a few tools written by the staff to help us.
- We might forget to include a line of code, and when we try to compile our program, see a lot of lines of error messages that are hard to understand, since the compiler might have been designed for a more technical audience. `help50` is a command we can run to explain problems in our code in a more user-friendly way. We can run it by adding `help50` to the front of a command we’re trying, like `help50 make hello`, to get advice that might be more understandable.
- It turns out that, in C, new lines and indentation generally don’t affect how our code runs. For example, we can change our `main` function to be one line, `int main(void) {printf("hello, world");}`, but it’s much harder to read, so we would consider it to have bad style. We can run `style50`, as with `style50 hello.c`, with the name of the file of our source code, to see suggestions for new lines and indentation.
- Additionally, we can add **comments**, notes in our source code for ourselves or other humans that don’t affect how our code runs. For example, we might add a line like `// Greet user`, with two slashes `//` to indicate that the line is a comment, and then write the purpose of our code or program to help us remember later on.
- `check50` will check the correctness of our code with some automated tests. The staff writes tests specifically for some of the programs we’ll be writing in the course, and instructions for using `check50` will be included in each problem set or lab as needed. After we run `check50`, we’ll see some output telling us whether our code passed relevant tests.
- The CS50 IDE also gives us the equivalent of our own computer in the cloud, somewhere

on the internet, with our own files and folders. If we click the folder icon in the top left, we'll see a file tree, a GUI of the files in our IDE:

- To open a file, we can just double-click it. `hello.c` is the source code that we just wrote, and `hello` itself will have lots of red dots, each of which are unprintable characters since they represent binary instructions for our computers.

Commands

- Since the CS50 IDE is a virtual computer in the cloud, we can also run commands available in Linux, an operating system like macOS or Windows.

- In the terminal, we can type in `ls`, short for list, to see a list of files and folder in the current folder:

```
~/ $ ls
hello*  hello.c
```

  - `hello` is in green with an asterisk to indicate that we can run it as a program.
- We can also *remove* files with `rm`, with a command like `rm hello`. It will prompt us for a confirmation, and we can respond with `y` or `n` for yes or no.
- With `mv`, or *move*, we can rename files. With `mv hello.c goodbye.c`, we've renamed our `hello.c` file to be named `goodbye.c`.
- With `mkdir`, or *make directory*, we can create folders, or directories. If we run `mkdir lecture`, we'll see a folder called `lecture`, and we can move files into directories with a command like `mv hello.c lecture/`.
- To *change directories* in our terminal, we can use `cd`, as with `cd lecture/`. Our prompt will change from `~/` to `~/lecture/`, indicating that we're in the `lecture` directory inside `~`. `~` stands for our home directory, or our account's default, top-level folder.
- We can also use `..` as shorthand for the parent, or containing folder. Within `~/lecture/`, we can run `mv hello.c ..` to move it back up to `~`, since it's the parent folder of `lecture/`. `cd ..`, similarly, will change our terminal's directory to the current one's parent. A single dot, `.`, refers to the current directory, as in `./hello`.
- Now that our `lecture/` folder is empty, we can remove it with `rmdir lecture/` as well.

Types, format codes

Types, format codes,

- There are many data **types** we can use for our variables, which indicate to the computer what type of data they represent:
  - `bool`, a Boolean expression of either `true` or `false`
  - `char`, a single ASCII character like `a` or `2`
  - `double`, a floating-point value with more digits than a `float`
  - `float`, a floating-point value, or real number with a decimal value
  - `int`, integers up to a certain size, or number of bits
  - `long`, integers with more bits, so they can count higher than an `int`
  - `string`, a string of characters
- And the CS50 library has corresponding functions to get input of various types:
  - `get_char`
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long`
  - `get_string`
- For `printf`, too, there are different placeholders for each type:
  - `%c` for chars
  - `%f` for floats, doubles
  - `%i` for ints
  - `%li` for longs
  - `%s` for strings

Operators, limitations, truncation

- There are several mathematical operators we can use, too:
  - `+` for addition
  - `-` for subtraction
  - `*` for multiplication
  - `/` for division
  - `%` for remainder
- We'll make a new program, `addition.c`:

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

int main(void)
{
```

```
{
    int x = get_int("x: ");

    int y = get_int("y: ");

    printf("%i\n", x + y);
}
```

- We'll include header files for libraries we know we want to use, and then we'll call `get_int` to get integers from the user, storing them in variables named `x` and `y`.
- Then, in `printf`, we'll print a placeholder for an integer, `%i`, followed by a new line. Since we want to print out the sum of `x` and `y`, we'll pass in `x + y` for `printf` to substitute in the string.
- We'll save, run `make addition` in the terminal, and then `./addition` to see our program working. If we type in something that's not an integer, we'll see `get_int` asking us for an integer again. If we type in a really big number, like `4000000000`, `get_int` will prompt us again too. This is because, like on many computer systems, an `int` in CS50 IDE is 32 bits, which can only contain about four billion different values. And since integers can be positive or negative, the highest positive value for an `int` can only be about two billion, with a lowest negative value of about negative two billion, for a total of about four billion total values.

- We can change our program to use the `long` type:

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

int main(void)
{
    long x = get_long("x: ");

    long y = get_long("y: ");

    printf("%li\n", x + y);
}
```

  - Now we can type in bigger integers, and see a correct result as expected.
- Whenever we get an error while compiling, it's a good idea to scroll up to the top to see the first error and fix that first, since sometimes a mistake early in the program will lead to the rest of the program being interpreted with errors as well.
- Let's look at another example, `truncation.c`:

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

int main(void)
```

```
{
    // Get numbers from user
    int x = get_int("x: ");
    int y = get_int("y: ");

    // Divide x by y
    float z = x / y;
    printf("%f\n", z);
}
```

- We'll store the result of `x` divided by `y` in `z`, a floating-point value, or real number, and print it out as a float too.
- But when we compile and run our program, we see `z` printed out as whole numbers like `0.000000` or `1.000000`. It turns out that, in our code, `x / y` is divided as two integers *first*, so the result given back by the division operation is an integer as well. The result is **truncated**, with the value after the decimal point lost. Even though `z` is a `float`, the value we're storing in it is already an integer.
- To fix this, we **cast**, or convert, our integers to floats before we divide them:

```
float z = (float) x / (float) y;
```

- The result will be a float as we expect, and in fact we can cast only one of `x` or `y` and get a float as well.

Variables, syntactic sugar

- In Scratch, we had blocks like "set [counter] to (0)" that set a **variable** to some value. In C, we would write `int counter = 0;` for the same effect.
- We can increase the value of a variable with `counter = counter + 1;`, where we look at the right side first, taking the original value of `counter`, adding 1, and then storing it into the left side (back into `counter` in this case).
- C also supports **syntactic sugar**, or shorthand expressions for the same functionality. In this case, we could equivalently say `counter += 1;` to add one to `counter` before storing it again. We could also just write `counter++;`, and we can learn this (and other examples) through looking at documentation or other references online.

Conditions

- We can translate conditions, or "if" blocks, with:

```
if (x < y)
{
    printf("x is less than y\n");
}
```

    - Notice that in C, we use `{` and `}` (as well as indentation) to indicate how lines of code should be nested.
- We can have “if” and “else” conditions:

```
if (x < y)
{
    printf("x is less than y\n");
}
else
{
    printf("x is not less than y\n");
}
```

- And even “else if”:

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

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

    - Notice that, to compare two values in C, we use `==`, two equals signs.
    - And, logically, we don’t need the `if (x == y)` in the final condition, since that’s the only case remaining, so we can just say `else`:

```
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");
}
```

- Let’s take a look at another example, `conditions.c`:

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

```
#include <stdio.h>

int main(void)
{
    // Prompt user for x
    int x = get_int("x: ");

    // Prompt user for y
    int y = get_int("y: ");

    // Compare x and y
    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");

    }
}
```

- We've included the conditions we just saw, along with two calls, or uses, of `get_int` to get `x` and `y` from the user.
- We'll compile and run our program to see that it indeed works as intended.

- In `agree.c`, we can ask the user to confirm or deny something:

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

int main(void)
{
    char c = get_char("Do you agree? ");

    // Check whether agreed
    if (c == 'Y' || c == 'y')
    {
        printf("Agreed.\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("Not agreed.\n");
    }
}
```

- With `get_char`, we can get a single character, and since we only have a single one

in our program, it seems reasonable to call it `c`.

- We use two vertical bars, `||`, to indicate a logical "or", whether either expression can be true for the condition to be followed. (Two ampersands, `&&`, indicate a logical "and", where both conditions would have to be true.) And notice that we use two equals signs, `==`, to compare two values, as well as single quotes, `'`, to surround our values of single characters.
- If neither of the expressions are true, nothing will happen since our program doesn't have a loop.

Boolean expressions, loops

- We can translate a "forever" block in Scratch with:

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

  - The `while` keyword requires a condition, so we use `true` as the Boolean expression to ensure that our loop will run forever. `while` will tell the computer to check whether the expression evaluates to `true`, and then run the lines inside the curly braces. Then it will repeat that until the expression isn't true anymore. In this case, `true` will always be true, so our loop is an **infinite loop**, or one that will run forever.
- We could do something a certain number of times with `while`:

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

  - We create a variable, `i`, and set it to 0. Then, while `i` is less than 50, we run some lines of code, including one where we add 1 to `i` each time. This way, our loop will eventually end when `i` reaches a value of 50.
  - In this case, we're using the variable `i` as a counter, but since it doesn't serve any additional purpose, we can simply name it `i`.
- Even though we *could* do the following and start counting at 1, by convention we should start at 0:

```
int i = 1;
while (i <= 50)
{
```

```
{
    printf("hello, world\n");
    i++;
}
```

- Another correct, but arguably less well-designed solution might be starting at 50 and counting backwards:

```
int i = 50;
while (i > 0)
{
    printf("hello, world\n");
    i--;
}
```

  - In this case, the logic for our loop is harder to reason about without serving any additional purpose, and might even confuse readers.

- Finally, more commonly we can use the `for` keyword:

```
for (int i = 0; i < 50; i++)
{


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

  - Again, first we create a variable named `i` and set it to 0. Then, we check that `i < 50` every time we reach the top of the loop, before we run any of the code inside. If that expression is true, then we run the code inside. Finally, after we run the code inside, we use `i++` to add one to `i`, and the loop repeats.
  - The `for` loop is more elegant than a `while` loop in this case, since everything related to the loop is in the same line, and only the code we actually want to run multiple times is inside the loop.

- Notice that for many of these lines of code, like `if` conditions and `for` loops, we don't put a semicolon at the end. This is just how the language of C was designed, many years ago, and a general rule is that only lines for actions or verbs have semicolons at the end.

Abstraction

- We can write a program that prints `meow` three times:

```
#include <stdio.h>

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

```
}
```

- We could use a `for` loop, so we don't have to copy and paste so many lines:

```
#include <stdio.h>

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

- We can move the `printf` line to its own function, like our own puzzle piece:

```
#include <stdio.h>

void meow(void) {
  printf("meow\n");
}


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

  - We defined a function, `meow`, above our `main` function.

- But conventionally, our `main` function should be the first function in our program, so we need a few more lines:

```
#include <stdio.h>

void meow(void);

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

void meow(void)
{
    printf("meow\n");
```

```
}
```

- It turns out that we need to declare our `meow` function first with a **prototype**, before we use it in `main`, and actually define it after. The compiler reads our source code from top to bottom, so it needs to know that `meow` will exist later in the file.

- We can even change our `meow` function to take in some input, `n`, and meow `n` times:

```
#include <stdio.h>

void meow(int n);

int main(void)
{
    meow(3);
}

void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");

    }
}
```

- The `void` before the `meow` function means that it doesn't return a value, and likewise in `main` we can't do anything with the result of `meow`, so we just call it.

- The abstraction here leads to better design, since we now have the flexibility to reuse our `meow` function in multiple places in the future.

- Let's look at another example of abstraction, `get_positive_int.c`:

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

int get_positive_int(void);

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

// Prompt user for positive integer
int get_positive_int(void)
{
    int n;
    do
    {
        n = get_int("Positive Integer: ");
```

```
    }
    while (n < 1);
    return n;
}
```

- We have our own function that calls `get_int` repeatedly until we have some integer that's *not* less than 1. With a do-while loop, our program will do something first, then check some condition, and repeat while the condition is true. A while loop, on the other hand, will check the condition first.
- We need to declare our integer `n` outside the do-while loop, since we need to use it after the loop ends. The **scope** of a variable in C refers to the context, or lines of code, within which it exists. In many cases, this will be the curly braces surrounding the variable.
- Notice that the function `get_positive_int` now starts with `int`, indicating that it has a return value of type `int`, and in `main` we indeed store it in `i` after calling `get_positive_int()`. In `get_positive_int`, we have a new keyword, `return`, to return the value `n` to wherever the function was called.

Mario

- We might want a program that prints part of a screen from a video game like Super Mario Bros. In `mario.c`, we can print four question marks, simulating blocks:

```
#include <stdio.h>

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

- With a loop, we can print a number of question marks, following them with a single new line after the loop:

```
#include <stdio.h>

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

- We can get a positive integer from the user, and print out that number of question marks, by using `n` for our loop:

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

int main(void)
{
    // Get positive integer from user
    int n;
    do
    {
        n = get_int("Width: ");
    }
    while (n < 1);

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

- And we can print a two-dimensional set of blocks with nested loops, one inside the other:

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

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

  - We have two nested loops, where the outer loop uses `i` to do everything inside 3 times, and the inner loop uses `j`, a different variable, to do something 3 times for each of *those* times. In other words, the outer loop prints 3 "rows", or lines, ending each of them with a new line, and the inner loop prints 3 "columns", or `#` characters, *without* a new line.

Memory, imprecision, and overflow

- Our computer has memory, in hardware chips called RAM, random-access memory. Our programs use that RAM to store data while they're running, but that memory is finite.

- With `imprecision.c`, we can see what happens when we use floats:

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

int main(void)
{
    float x = get_float("x: ");
    float y = get_float("y: ");

    printf("%.50f\n", x / y);
}
```

  - With `%.50f`, we can specify the number of decimal places displayed.
  - Hmm, now we get …

```
x: 1
y: 10
0.10000000149011611938476562500000000000000000000000
```

  - It turns out that this is called **floating-point imprecision**, where we don't have enough bits to store all possible values. With a finite number of bits for a `float`, we can't represent all possible real numbers (of which there are an *infinite* number of), so the computer has to store the closest value it can. And this can lead to problems where even small differences in value add up, unless the programmer uses some other way to represent decimal values as accurately as needed.
- Last week, when we had three bits and needed to count higher than seven (or `111`), we added another bit to get eight, `1000`. But if we only had three bits available, we wouldn't have a place for the extra `1`. It would disappear and we would be back at `000`. This problem is called **integer overflow**, where an integer can only be so big before it runs out of bits.
- The Y2K problem arose because many programs stored the calendar year with just two digits, like 98 for 1998, and 99 for 1999. But when the year 2000 approached, the programs had to store only 00, leading to confusion between the years 1900 and 2000.
- In 2038, we'll also run out of bits to track time, since many years ago some humans decided to use 32 bits as the standard number of bits to count the number of seconds since January 1st, 1970. But with 32 bits representing only positive numbers, we can only count up to about four billion, and in 2038 we'll reach that limit unless we upgrade the software in all of our computer systems.