This is CS50

CS50's Introduction to Computer Science

OpenCourseWare

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

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan)
(https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/)
(https://www.reddit.com/user/davidjmalan)
(https://www.threads.net/@davidjmalan) (https://twitter.com/davidjmalan)

Lecture 1

Welcome!

- In our previous session, we learned about Scratch, a visual programming language.
- Indeed, all the essential programming concepts presented in Scratch will be utilized as you learn how to program any programming language. Functions, conditionals, loops, and variables found in Scratch are fundamental building blocks that you will find in any programming language.
- Recall that machines only understand binary. Where humans write *source code*, a list of instructions for the computer that is human readable, machines only understand what we can now call *machine code*. This machine code is a pattern of ones and zeros that produces a desired effect.
- It turns out that we can convert *source code* into machine code using a very special piece of software called a *compiler*. Today, we will be introducing you to a compiler that will allow you to convert source code in the programming language *C* into machine code.
- Today, in addition to learning how to program, you will be learning how to write good code.

Visual Studio Code for CS50

- The integrated development environment (IDE) that is utilized for this course is *Visual Studio Code*, affectionately referred to as , which can be accessed via that same URL or simply as *VS Code.*
- One of the most important reasons we utilize VS Code is that it has all the software required for the course already pre-loaded on it. This course and the instructions herein were designed with VS Code in mind.
- Manually installing the necessary software for the course on your own computer is a cumbersome headache. Best always to utilize VS Code for assignments in this course.
- You can open VS Code at cs50.dev (https://cs50.dev/).
- The compiler can be divided into a number of regions:

Notice that there is a *file explorer* on the left side where you can find your files. Further, notice that there is a region in the middle called a *text editor* where you can edit your program. Finally, there is a `command line interface`, known as a *CLI*, *command line*, or *terminal window*, where we can send commands to the computer in the cloud.

- In the terminal window, some common command-line arguments we may use include:
  - `cd`, for changing our current directory (folder)
  - `cp`, for copying files and directories
  - `ls`, for listing files in a directory
  - `mkdir`, for making a directory
  - `mv`, for moving (renaming) files and directories
  - `rm`, for removing (deleting) files
  - `rmdir`, for removing (deleting) directories
- The most commonly used is `ls` which will list all the files in the current directory or directory. Go ahead and type `ls` into the terminal window and hit `enter`. You'll see all the files in the current folder.
- Because this IDE is preconfigured with all the necessary software, you should use it to complete all assignments for this course.

Hello World

- We will be using three commands to write, compile, and run our first program:

```
code hello.c

make hello
```

```
./hello
```

The first command, `code hello.c` creates a file and allows us to type instructions for this program. The second command, `make hello`, *compiles* the file from our instructions in C and creates an executable file called `hello`. The last command, `./hello`, runs the program called `hello`.

- We can build your first program in C by typing `code hello.c` into the terminal window. Notice that we deliberately lowercased the entire filename and included the `.c` extension. Then, in the text editor that appears, write code as follows:

```
// A program that says hello to the world

#include <stdio.h>

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

Note that every single character above serves a purpose. If you type it incorrectly, the program will not run. `printf` is a function that can output a line of text. Notice the placement of the quotes and the semicolon. Further, notice that the `\n` creates a new line after the words `hello, world`.

- Clicking back in the terminal window, you can compile your code by executing `make hello`. Notice that we are omitting `.c`. `make` is a compiler that will look for our `hello.c` file and turn it into a program called `hello`. If executing this command results in no errors, you can proceed. If not, double-check your code to ensure it matches the above.
- Now, type `./hello` and your program will execute saying `hello, world`.
- Now, open the file explorer on the left. You will notice that there is now both a file called `hello.c` and another file called `hello`. `hello.c` is able to be read by the compiler: It's where your code is stored. `hello` is an executable file that you can run but cannot be read by the compiler.

From Scratch to C

- In Scratch, we utilized the `say` block to display any text on the screen. Indeed, in C, we have a function called `printf` that does exactly this.
- Notice our code already invokes this function:

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

Notice that the printf function is called. The argument passed to printf is `hello, world\n`. The statement of code is closed with a `;`.

- Errors in code are common. Modify your code as follows:

```
// \n is missing

#include <stdio.h>

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

Notice the `\n` is now gone.

- In your terminal window, run `make hello`. Typing `./hello` in the terminal window, how did your program change? This `\` character is called an *escape character* that tells the compiler that `\n` is a special instruction to create a line break.
- There are other escape characters you can use:

```
\n  create a new line
\r  return to the start of a line
\"  print a double quote
\`  print a single quote
\\  print a backslash
```

- Restore your program to the following:

```
// A program that says hello to the world

#include <stdio.h>

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

Notice the semicolon and `\n` have been restored.

Header Files and CS50 Manual Pages

- The statement at the start of the code `#include <stdio.h>` is a very special command that tells the compile that you want to use the capabilities of a *library* called `stdio.h`, a *header file*. This allows you, among many other things, to utilize the `printf` function.
- A *library* is a collection of code created by someone. Libraries are collections of pre-written code and functions that others have written in the past that we can utilize in

our code.

- You can read about all the capabilities of this library on the Manual Pages (https://manual.cs50.io). The Manual Pages provide a means by which to better understand what various commands do and how they function.
- It turns out that CS50 has its own library called `cs50.h`. There are numerous functions that are included that provide *training wheels* while you get started in C:

```
get_char
get_double
get_float
get_int
get_long
get_string
```

- Let's use this library in your program.

Hello, You

- Recall that in Scratch we had the ability to ask the user, "What's your name?" and say "hello" with that name appended to it.
- In C, we can do the same. Modify your code as follows:

```
// get_string and printf with incorrect placeholder

#include <stdio.h>

int main(void)
{
    string answer = get_string("What's your name? ");
    printf("hello, answer\n");
}
```

The `get_string` function is used to get a string from the user. Then, the variable `answer` is passed to the `printf` function.

- Running `make hello` again in the terminal window, notice that numerous errors appear.
- Looking at the errors, `string` and `get_string` are not recognized by the compiler. We have to teach the compiler these features by adding a library called `cs50.h`. Also, we notice that `answer` is not provided as we intended. Modify your code as follows:

```
// get_string and printf with %s

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

int main(void)
```

```
{
    string answer = get_string("What's your name? ");
    printf("hello, %s\n", answer);
}
```

The `get_string` function is used to get a string from the user. Then, the variable `answer` is passed to the `printf` function. `%s` tells the `printf` function to prepare itself to receive a `string`.

- Now, running `make hello` again in the terminal window, you can run your program by typing `./hello`. The program now asks for your name and then says hello with your name attached, as intended.
- `answer` is a special holding place we call a *variable*. `answer` is of type `string` and can hold any string within it. There are many *data types*, such as `int`, `bool`, `char`, and many others.
- `%s` is a placeholder called a *format code* that tells the `printf` function to prepare to receive a `string`. `answer` is the `string` being passed to `%s`.

Types

- `printf` allows for many format codes. Here is a non-comprehensive list of ones you may utilize in this course:

```
%c
%f
%i
%li
%s
```

`%s` is used for `string` variables. `%i` is used for `int` or integer variables. You can find out more about this on the Manual Pages (https://manual.cs50.io)

- These format codes correspond to the many data types that are available within C:

```
bool
char
float
int
long
string
...
```

- We will be using many of C's available data types throughout this course.

Conditionals

- Another building block you utilized within Scratch was *conditionals*. For example, you might want to do one thing if x is greater than y. Further, you might want to do something else if that condition is not met.
- We look at a few examples from Scratch.
- In C, you can compare two values as follows:

```
// Conditionals that are mutually exclusive

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

  Notice how if `x < y`, one outcome occurs. If `x < y`, then another outcome occurs.

- Similarly, we can plan for three possible outcomes:

```
// Conditional that isn't necessary

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 not all these lines of code are required. How could we eliminate the unnecessary calculation above?

- You may have guessed that we can improve this code as follows:

```
// Compare integers

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");
}
```

Operators

- *Operators* refer to the mathematical operations that are supported by your compiler. In C, these mathematical operators include:
  - `+` for addition
  - `-` for subtraction
  - `*` for multiplication
  - `/` for division
  - `%` for remainder
- We will use all of these operators in this course.

Variables

- In C, you can assign a value to an `int` or integer as follows:

  ```
  int counter = 0;
  ```

  Notice how a variable called `counter` of type `int` is assigned the value `0`.
- C can also be programmed to add one to `counter` as follows:

  ```
  counter = counter + 1;
  ```

  Notice how `1` is added to the value of `counter`.
- This can be also represented as:

  ```
  counter = counter++;
  ```

  Notice how `1` is added to the value of `counter`. However, the `++` is used instead of `counter + 1`.
- You can also subtract one from `counter` as follows:

  ```
  counter = counter--;
  ```

  Notice how `1` is removed to the value of `counter`.

compare.c

- Using this new knowledge about how to assign values to variables, you can program your first conditional statement.
- In the terminal window, type `code compare.c` and write code as follows:

```
// Conditional, Boolean expression, relational operator

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

int main(void)
{
    // Prompt user for integers
    int x = get_int("What's x? ");
    int y = get_int("What's y? ");

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

  Notice that we create two variables, an `int` or integer called `x` and another called `y`. The values of these are populated using the `get_int` function.
- You can run your code by executing `make compare` in the terminal window, followed by `./compare`. If you get any error messages, check your code for errors.
- *Flow charts* are a way by which you can examine how a computer program functions. Such charts can be used to examine the efficiency of our code.
- Looking at a flow chart of the above code, we can notice numerous shortcomings.
- We can improve your program by coding as follows:

```
// Conditionals

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

int main(void)
{
    // Prompt user for integers
    int x = get_int("What's x? ");
    int y = get_int("What's y? ");

    // Compare integers
    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");
    }
}
```

Notice that all potential outcomes are now accounted for.

- You can re-make and re-run your program and test it out.
- Examining this program on a flow chart, you can see the efficiency of our code design decisions.

agree.c

- Considering another data type called a `char`, we can start a new program by typing `code agree.c` into the terminal window.
- Where a `string` is a series of characters, a `char` is a single character.
- In the text editor, write code as follows:

```
// Comparing against lowercase char

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

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

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

Notice that single quotes are utilized for single characters. Further, notice that `==` ensures that something *is equal* to something else, where a single equal sign would have a very different function in C.

- You can test your code by typing `make agree` into the terminal window, followed by `./agree`.
- We can also allow for the inputting of uppercase and lowercase characters:

```
// Comparing against lowercase and uppercase char
```

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

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

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

Notice that additional options are offered. However, this is not efficient code.

- We can improve this code as follows:

```c
// Logical operators

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

int main(void)
{
    // Prompt user to agree
    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");
    }
}
```

Notice that `||` effectively means *or*.

Loops and cat.c

- We can also utilize the loop building block from Scratch in our C programs.
- In your terminal window, type `code meow.c` and write code as follows:

```
// Opportunity for better design

#include <stdio.h>

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

  Notice this does as intended but has an opportunity for better design. Code is repeated over and over.

- We can improve our program by modifying your code as follows:

```
// Better design

#include <stdio.h>

int main(void)
{
    int i = 3;
    while (i > 0)
    {
        printf("meow\n");
        i--;
    }
}
```

  Notice that we create an `int` called `i` and assign it the value `3`. Then, we create a `while` loop that will run as long as `i > 0`. Then, the loop runs. Every time `1` is subtracted to `i` using the `i--` statement.

- Similarly, we can implement a count-up of sorts by modifying our code as follows:

```
// Print values of i

#include <stdio.h>

int main(void)
{
    int i = 1;
    while (i <= 3)
    {
        printf("meow\n");
        i++;
```

```
    }
}
```

Notice how our counter `i` is started at `1`. Each time the loop runs, it will increment the counter by `1`. Once the counter is greater than `3`, it will stop the loop.

- Generally, in computer science, we count from zero. Best to revise your code as follows:

```
// Better design

#include <stdio.h>

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

Notice we now count from zero.

- Another tool in our toolbox for looping is a `for` loop.
- You can further improve the design of our `meow.c` program using a `for` loop. Modify your code as follows:

```
// Better design

#include <stdio.h>

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

Notice that the `for` loop includes three arguments. The first argument `int i = 0` starts our counter at zero. The second argument `i < 3` is the condition that is being checked. Finally, the argument `i++` tells the loop to increment by one each time the loop runs.

- We can even loop forever using the following code:

```
// Infinite loop

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

int main(void)
```

```
{
    while (true)
    {
        printf("meow\n");
    }
}
```

Notice that `true` will always be the case. Therefore, the code will always run. You will lose control of your terminal window by running this code. You can break from an infinite by hitting `control-C` on your keyboard.

Functions

- While we will provide much more guidance later, you can create your own function within C as follows:

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

The initial `void` means that the function does not return any values. The `(void)` means that no values are being provided to the function.

- This function can be used in the main function as follows:

```
// Abstraction

#include <stdio.h>

void meow(void);

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

// Meow once
void meow(void)
{
    printf("meow\n");
}
```

Notice how the `meow` function is called with the `meow()` instruction. This is possible because the `meow` function is defined at the bottom of the code, and the *prototype* of the function is provided at the top of the code as `void meow(void)`.

- Your `meow` function can be further modified to accept input:

```c
// Abstraction with parameterization

#include <stdio.h>

void meow(int n);

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

// Meow some number of times
void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

Notice that the prototype has changed to `void meow(int n)` to show that `meow` accepts an `int` as its input.

- Additionally, we can get user input:

```c
// User input

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

void meow(int n);

int main(void)
{
    int n;
    do
    {
        n = get_int("Number: ");
    }
    while (n < 1);
    meow(n);
}

// Meow some number of times
void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

Notice that `get_int` is used to obtain a number from the user. `n` is passed to `meow`.

- We can even test to ensure that the input we get provided by the user is correct:

```c
// Return value

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

int get_positive_int(void);
void meow(int n);

int main(void)
{
    int n = get_positive_int();
    meow(n);
}

// Get number of meows
int get_positive_int(void)
{
    int n;
    do
    {
        n = get_int("Number: ");
    }
    while (n < 1);
    return n;
}

// Meow some number of times
void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

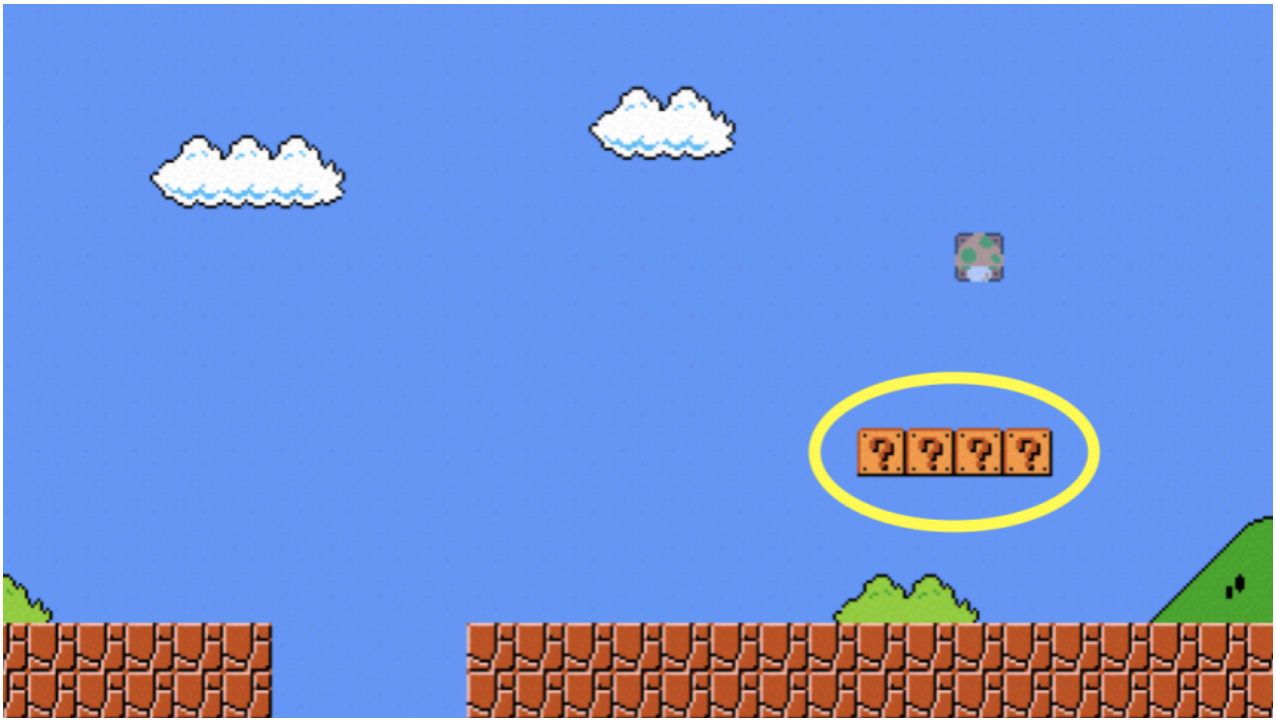
Notice that a new function called `get_positive_int` asks the user for an integer while `n < 1`. After obtaining a positive integer, this function will `return n` back to the `main` function.

Correctness, Design, Style

- Code can be evaluated upon three axes.
- First, *correctness* refers to "Does the code run as intended?" You can check the correctness of your code with `check50`.
- Second, *design* refers to "How well is the code designed?" You can evaluate the design of your code using `design50`.
- Finally, *style* refers to "How aesthetically pleasing and consistent is the code?" You can evaluate the style of your code with `style50`.

Mario

- Everything we've discussed today has focused on various building blocks of your work as an emerging computer scientist.
- The following will help you orient toward working on a problem set for this class in general: How does one approach a computer science-related problem?
- Imagine we wanted to emulate the visual of the game Super Mario Bros. Considering the four question blocks pictured, how could we create code that roughly represents these four horizontal blocks?



- In the terminal window, type `code mario.c` and code as follows:

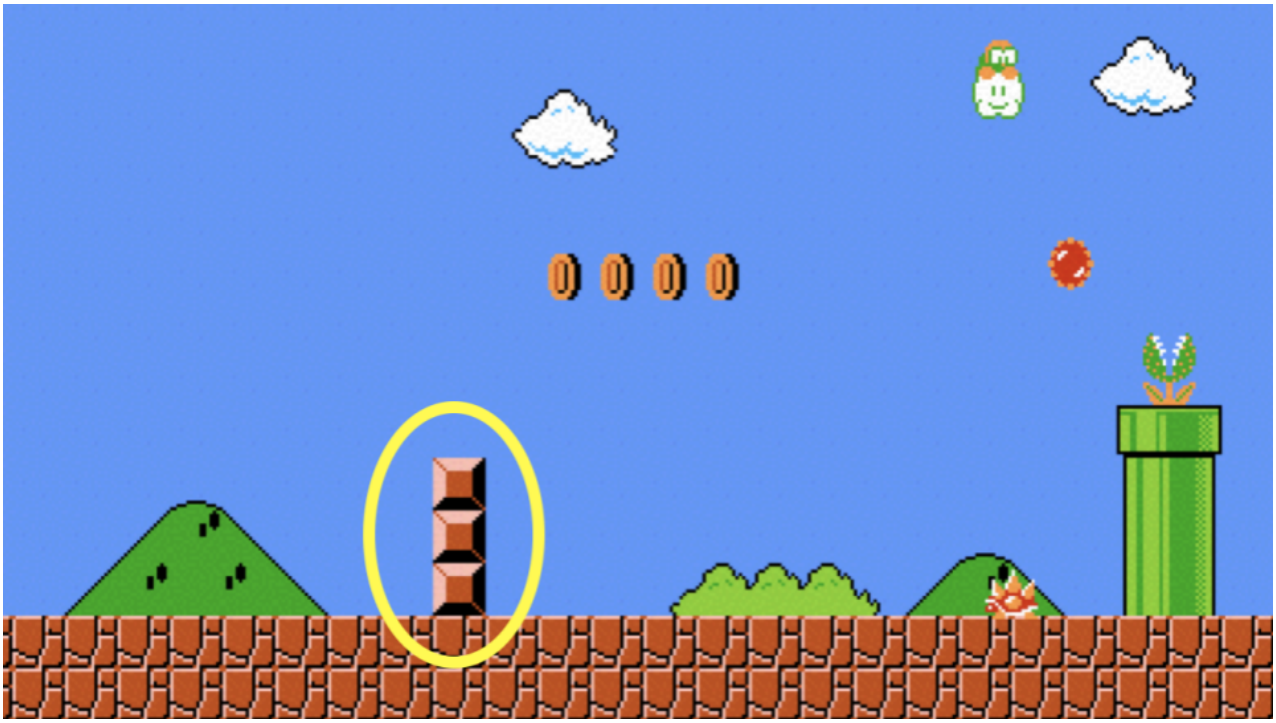
```c
// Prints a row of 4 question marks with a loop

#include <stdio.h>

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

  Notice how four question marks are printed here using a loop.

- Similarly, we can apply this same logic to create three vertical blocks.



- To accomplish this, modify your code as follows:

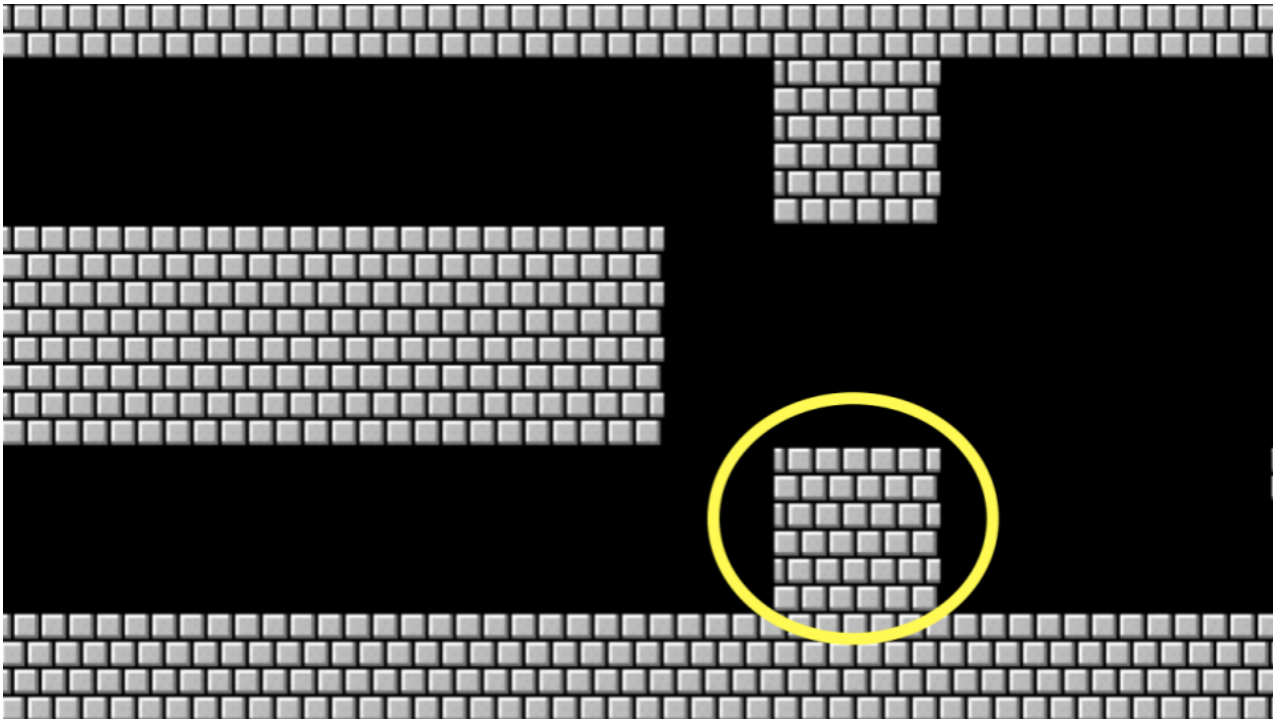
```c
// Prints a column of 3 bricks with a loop

#include <stdio.h>

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

Notice how three vertical bricks are printed using a loop.

- What if we wanted to combine these ideas to create a three-by-three group of blocks?



- We can follow the logic above, combining the same ideas. Modify your code as follows:

```
// Prints a 3-by-3 grid of bricks with nested loops

#include <stdio.h>

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

Notice that one loop is inside another. The first loop defines what vertical row is being printed. For each row, three columns are printed. After each row, a new line is printed.

- What if we wanted to ensure that the number of blocks is *constant*, that is, unchangeable? Modify your code as follows:

```
// Prints a 3-by-3 grid of bricks with nested loops using a constant

#include <stdio.h>

int main(void)
{
    const int n = 3;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
```

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

Notice how `n` is now a constant. It can never be changed.

- As illustrated earlier in this lecture, we can *abstract away* functionality into functions. Consider the following code:

```
// Helper function

#include <stdio.h>

void print_row(int width);

int main(void)
{
    const int n = 3;
    for (int i = 0; i < n; i++)
    {
        print_row(n);
    }
}

void print_row(int width)
{
    for (int i = 0; i < width; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

Notice how printing a row is accomplished through a new function.

Comments

- Comments are fundamental parts of a computer program, where you leave explanatory remarks to yourself and others who may be collaborating with you regarding your code.
- All code you create for this course must include robust comments.
- Typically, each comment is a few words or more, providing the reader an opportunity to understand what is happening in a specific block of code. Further, such comments serve as a reminder for you later when you need to revise your code.
- Comments involve placing `//` into your code, followed by a comment. Modify your code as follows to integrate comments:

```c
// Helper function

#include <stdio.h>

void print_row(int width);

int main(void)
{
    const int n = 3;

    // Print n rows
    for (int i = 0; i < n; i++)
    {
        print_row(n);
    }
}

void print_row(int width)
{
    for (int i = 0; i < width; i++)
    {
        printf("#");
    }
    printf("\n");
}
```

Notice how each comment begins with a `//`.

More About Operators

- You can implement a calculator in C. In your terminal, type `code calculator.c` and write code as follows:

```c
// Addition with int

#include <cs50.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: ");

    // Add numbers
    int z = x + y;

    // Perform addition
    printf("%i\n", z);
}
```

Notice how the `get_int` function is utilized to obtain an integer from the user twice. One integer is stored in the `int` variable called `x`. Another is stored in the `int` variable called `y`. The sum is stored in `z`. Then, the `printf` function prints the value of `z`, designated by the `%i` symbol.

- We can also double a number:

```
// int

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

int main(void)
{
    int dollars = 1;
    while (true)
    {
        char c = get_char("Here's $%i. Double it and give to next per
        if (c == 'y')
        {
            dollars *= 2;
        }
        else
        {
            break;
        }
    }
    printf("Here's $%i.\n", dollars);
}
```

Running this program, some seeming errors appear in `dollars`. Why is this?

- One of C's shortcomings is the ease by which it manages memory. While C provides you immense control over how memory is utilized, programmers have to be very aware of the potential pitfalls of memory management.
- Types refer to the possible data that can be stored within a variable. For example, a `char` is designed to accommodate a single character like `a` or `2`.
- Types are very important because each type has specific limits. For example, because of the limits in memory, the highest value of an `int` can be `4294967295`. If you attempt to count an `int` higher, an *integer overflow* will result where an incorrect value will be stored in this variable.
- The number of bits limits how high and low we can count.
- This can have catastrophic, real-world impacts.
- We can correct this by using a data type called `long`.

```
// long

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

```c
int main(void)
{
    long dollars = 1;
    while (true)
    {
        char c = get_char("Here's $%li. Double it and give to next pe
        if (c == 'y')
        {
            dollars *= 2;
        }
        else
        {
            break;
        }
    }
    printf("Here's $%li.\n", dollars);
}
```

Notice how running this code will allow for very high dollar amounts.

- Types with which you might interact during this course include:
  - `bool`, a Boolean expression of either true or false
  - `char`, a single character like a or 2
  - `double`, a floating-point value with more digits than a float
  - `float`, a floating-point value, or a 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

Truncation

- Another issue that can arise when using data types includes truncation.

```c
// Division with ints, demonstrating truncation

#include <cs50.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: ");

    // Divide x by y
    printf("%i\n", x / y);
}
```

An integer divided by an integer will always result in an integer in C. Accordingly, the above code will often result in any digits after the decimal being thrown away.

- This can be solved by employing a `float`:

```
// Floats

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

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

    // Prompt user for y
    float y = get_float("y: ");

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

Notice that this solves some of our problems. However, we might notice imprecision in the answer provided by the program.

- *Floating point imprecision* illustrates that there are limits to how precise computers can calculate numbers.
- As you are coding, pay special attention to the types of variables you are using to avoid problems within your code.
- We examined some examples of disasters that can occur through type-related errors.

Summing Up

In this lesson, you learned how to apply the building blocks you learned in Scratch to the C programming language. You learned...

- How to create your first program in C.
- How to use the command line.
- About predefined functions that come natively with C.
- How to use variables, conditionals, and loops.
- How to create your own functions to simplify and improve your code.
- How to evaluate your code on three axes: correctness, design, and style.
- How to integrate comments into your code.
- How to utilize types and operators and the implications of your choices.

See you next time!