Table of Contents

Lecture 1

C

- Last week we used Scratch to better understand ideas and concepts in programming, like loops and functions.
- Today, we'll get an introduction to C, an older language, as another way of familiarizing ourselves with the tool of programming.
- We can start translating our simple programs from Scratch to C, the code of which is written purely in text.
- The `say` block in Scratch is a function, which the equivalent in C is the following:

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

  - Notice that the function itself is called `printf`, and that the arguments, or parameters, to the function are wrapped inside symmetrical parentheses, `(` and `)`.
  - The double quotes, `"`, are also symmetrical and surround words, or any sequence of characters, in C. We'll start calling these sequences of characters **strings**.
  - The line also ends with a semicolon, which new programmers like us need to remember to include, but will come more naturally with practice!
- To make this a complete program, we can add the following:

  ```
  #include <stdio.h>

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

  - Just like how we need the `when green flag clicked` block in Scratch to start our program, our C program won't run unless we write a few lines to set it up.
  - We notice that there's a `int main(void)` line, and `main` is the standard name in C to indicate that it is the default function in a program that should be run.
  - The top line is harder to guess, but `include` is a keyword that indicates we want to include some other file in our program. `stdio.h` contains (and we only know from searching online and looking at documentation) the standard input/output library, which means that it deals with input

(like from the keyboard) and output (printing characters to the screen). In fact, it contains the code of `printf` that we are using. There is no equivalent in Scratch, since by default the functions are already defined and created for us.

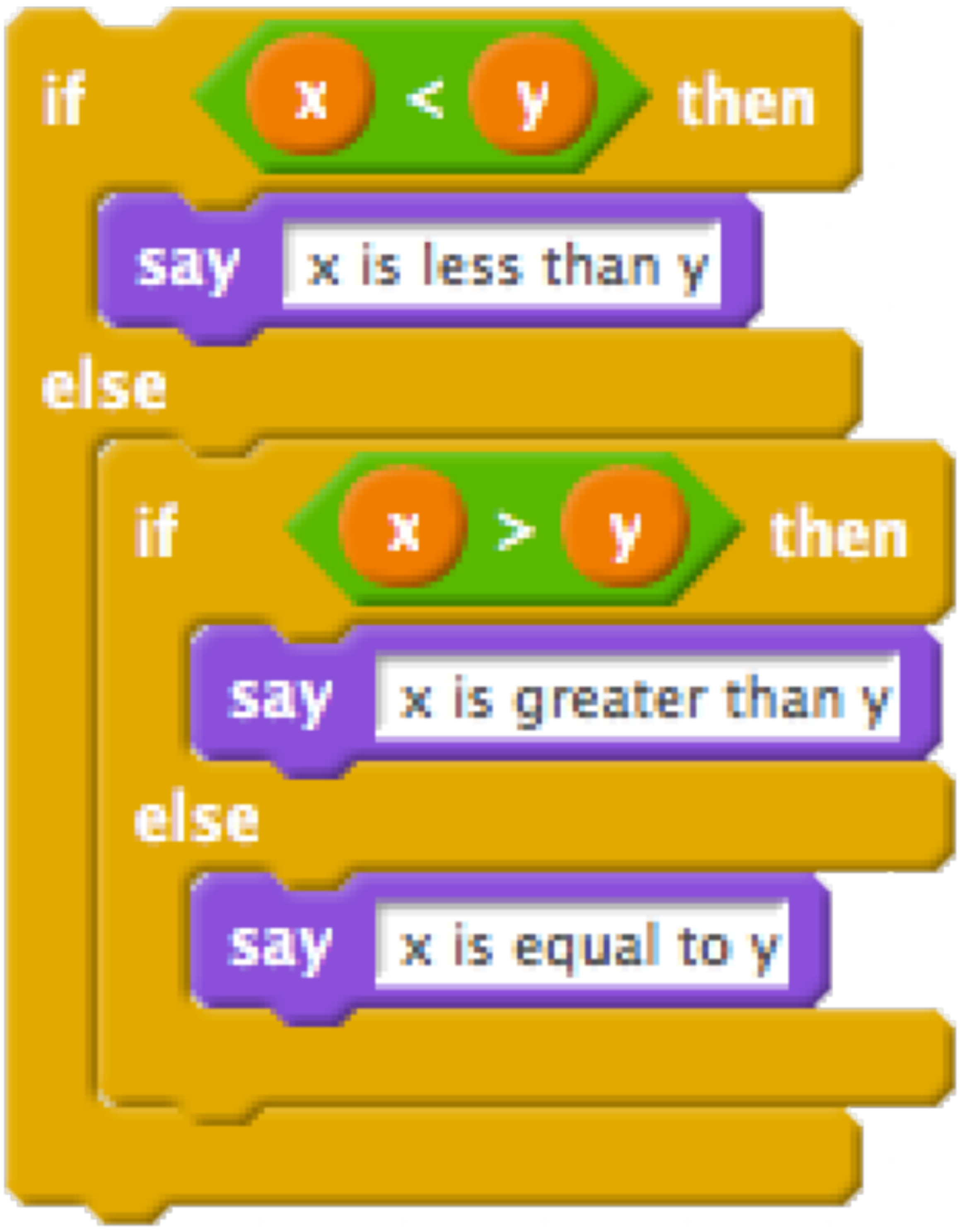
- A `forever` block can be translated to C like this:

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

  - The `while` keyword means that the loop will run as long as the Boolean expression inside the parentheses is true. And since `true` will always be true, the loop will run forever.
- To `repeat` something a certain number of times, we can use this:

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

  - This is a little harder to figure out, but we can go through step by step. `for` is another keyword in C that indicates a loop.
  - `int i = 0` is an initialization of a variable, which means that we created a variable with the name `i`, of the type `int`, or integer, and set its initial value to `0`. In C, each variable has a type of value.
  - Then `i < 50` is the Boolean expression that the `for` loop checks, to determine if it will continue or not. Since this condition is true, the `for` loop will run the `printf` line. And since we started `i` at 0, stopping before `i` reaches 50 will mean this runs exactly 50 times, as we intended.
  - Finally, `i++` is an expression in C that adds 1 to the value of `i`. Then, the `for` loop will check `i < 50`, and repeat this process until the Boolean expression is no longer true.
- Last time, we also created a nested set of conditions in Scratch:



```
if x < y then
    say x is less than y
else
    if x > y then
        say x is greater than y
    else
        say x is equal to y
```

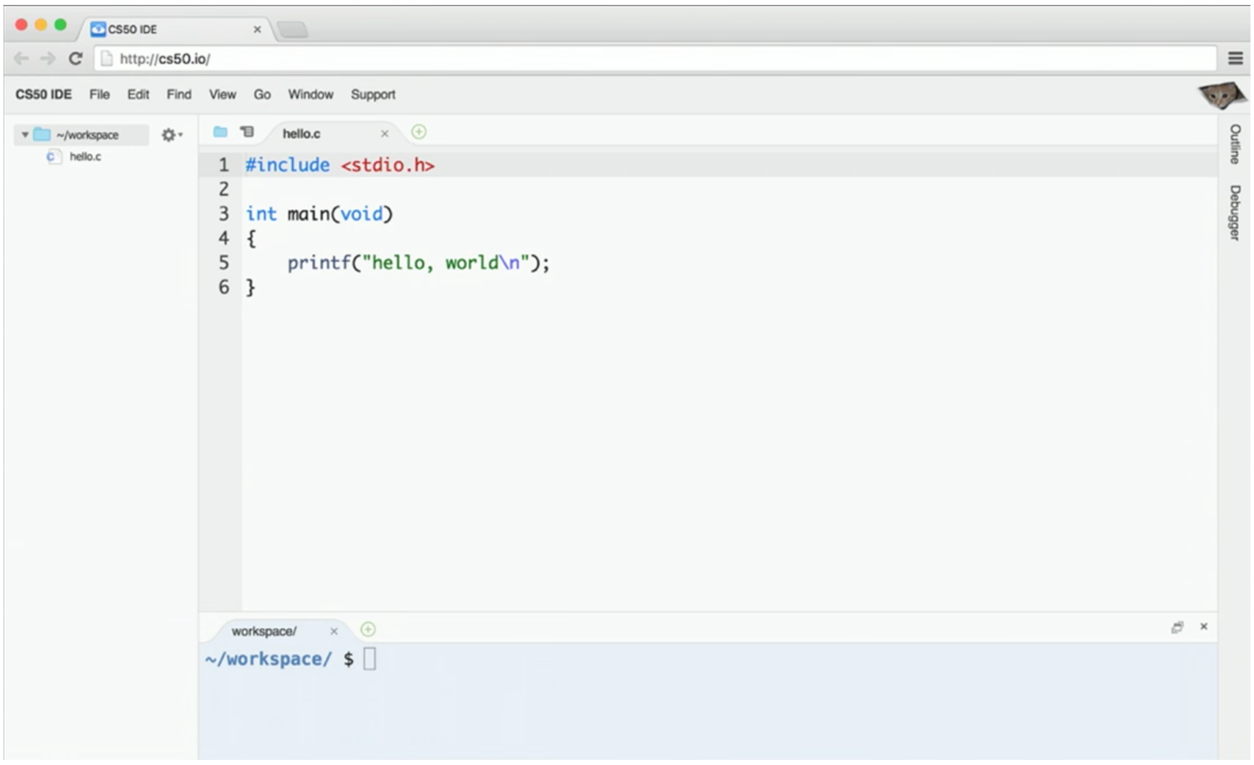
- In C, the equivalent code will look like this:

```
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");
}
```

- In our code, we assume that `x` and `y` have already been initialized or set to some other values beforehand.
- We use the `if`, `else if`, and `else` keywords to denote the forks in the road, based on Boolean expressions. `else` simply captures all the cases that haven't fit into a previous condition.
- Notice that curly braces, `{` and `}`, are used to wrap the lines of code that we want to run for each of the conditions if they are true. We also use indentation to make the lines of code more readable.

Compiling

- So now we can write fairly basic programs in C. But computers only understand binary, so the **source code** that we write need to be converted to 0s and 1s, or **machine code** that a computer actually uses.
- This conversion is done by software called a **compiler**. We'll set everyone up with the same environment, or the same platform.
- This environment is the CS50 IDE (Integrated Development Environment), at cs50.io. We can think of it as a web application, inside which we can write, compile, and run code:

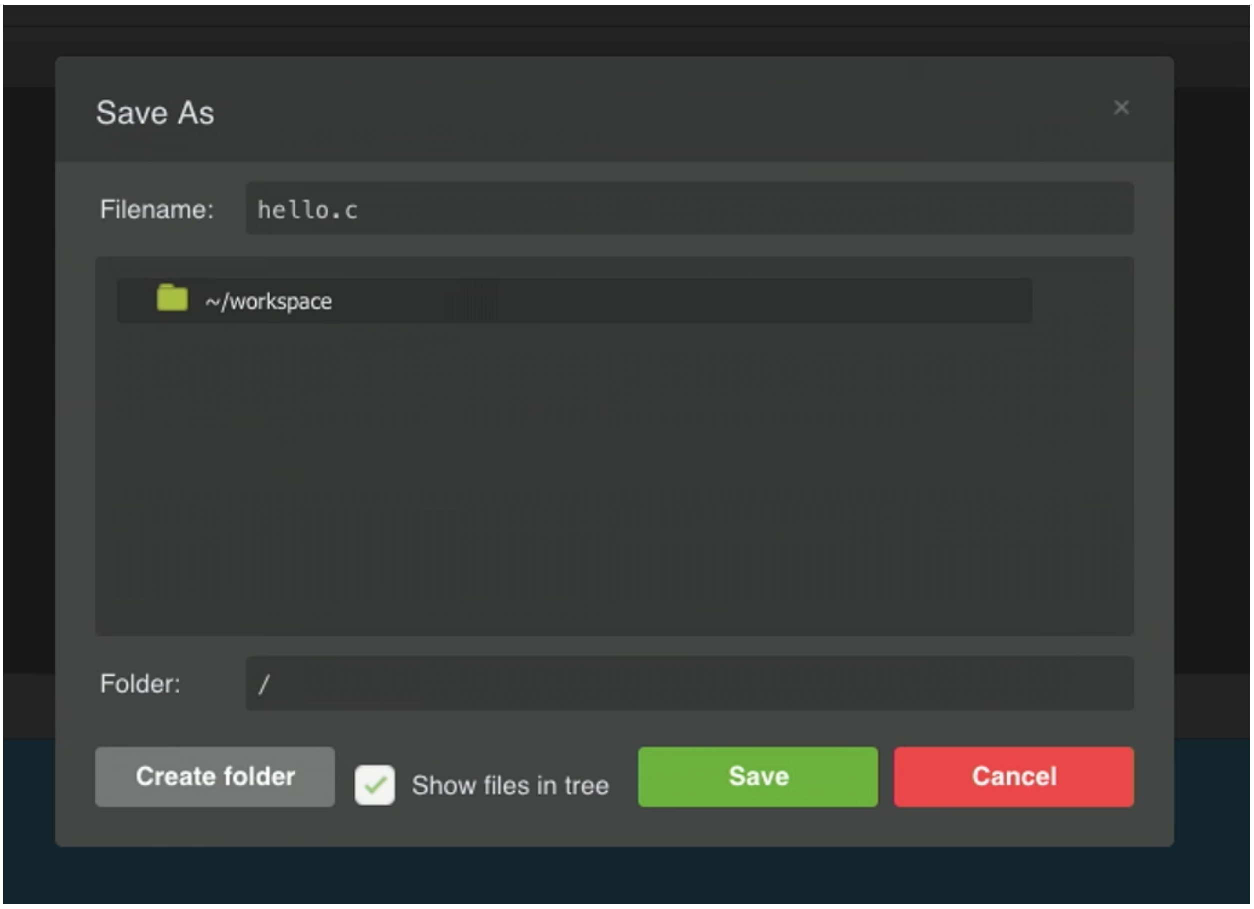


```c
#include <stdio.h>

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

```
~/workspace/ $
```

  - We could each do this on our own computers, but setting up the editors, compilers, and other necessary software is a lot of work
  - Notice that we have a file browser to the left, where we can upload or download files from it, the code editor on the right, and the terminal at bottom right, into which we can type commands that our virtual environment will run.
  - We'll have instructions to log in for Problem Set 1, but for now, just follow along!
  - By default, we have a `~/workspace` folder where we can save files to.
- First, let's create a new file. We'll use `File > Save`, and call our new file `hello.c`:


Save As

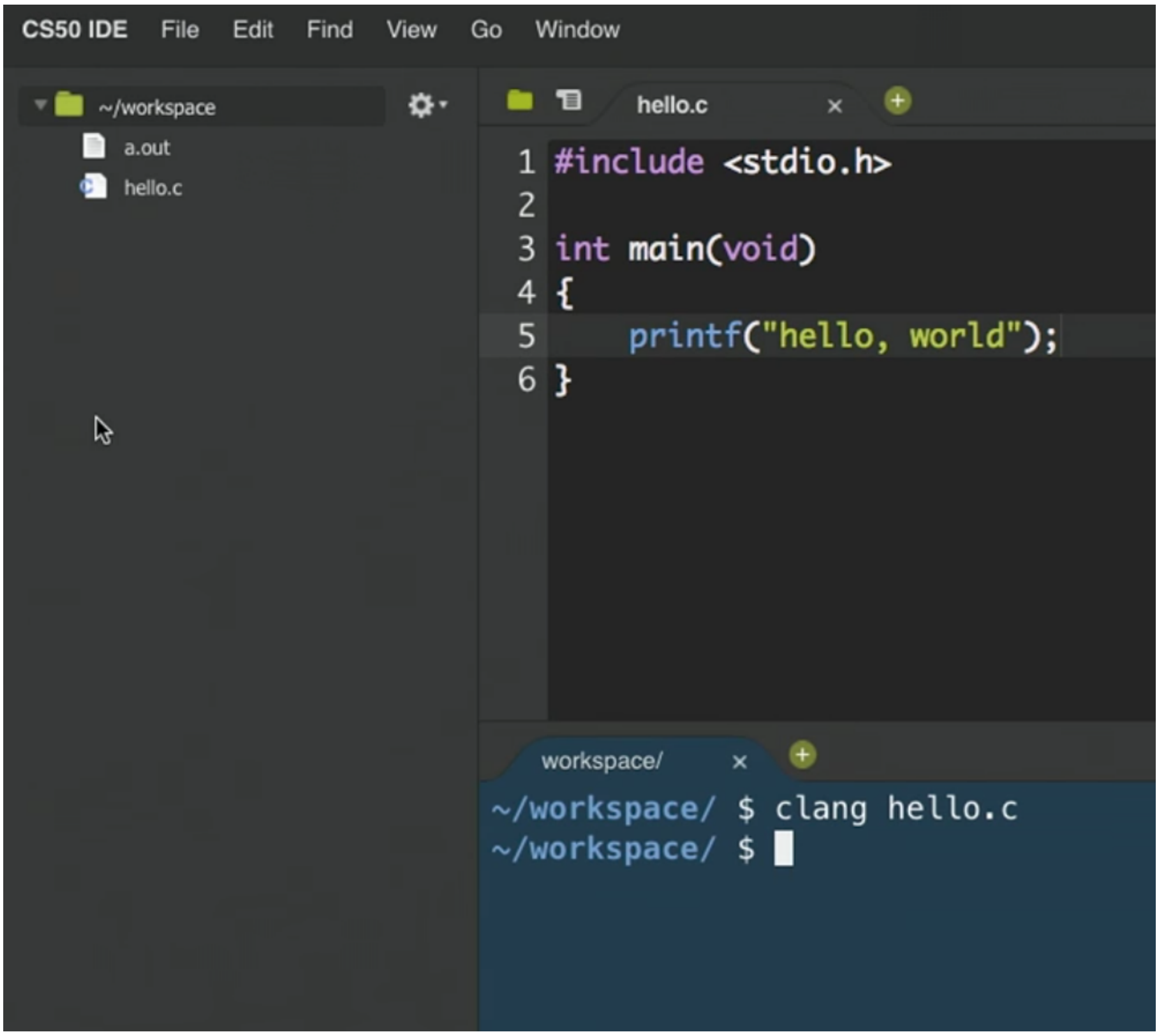
Filename: hello.c

~/workspace

Folder: /

- Since we are going to write a program in C, we will end our file with the `.c` extension. And we'll only use lowercase, with underscores or hyphens instead of space, by convention.

- We can type out our basic program, compile it, and run it:

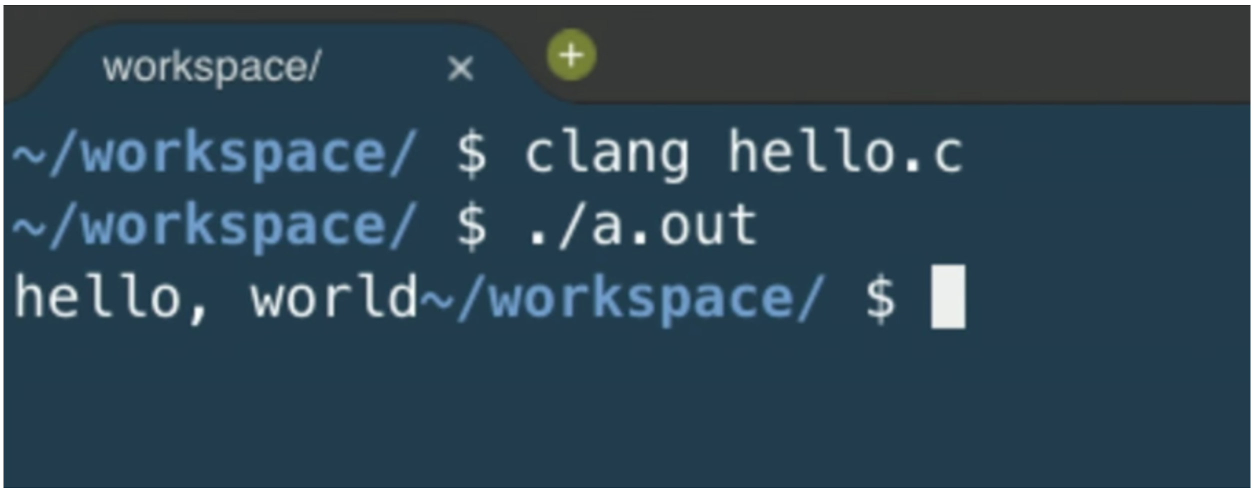


```c
#include <stdio.h>

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

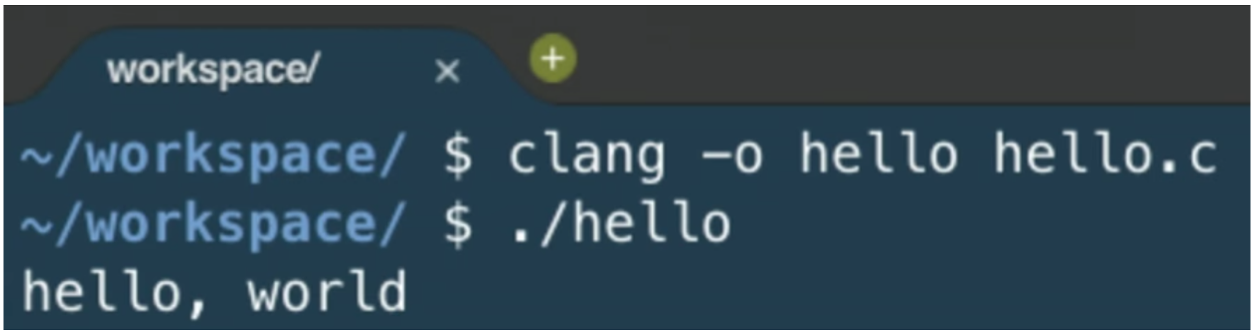
```
~/workspace/ $ clang hello.c
~/workspace/ $
```

  - Notice that the editor automatically makes our code colorful, or has syntax highlighting, to help us see patterns.
  - We'll type the command `clang hello.c` in the terminal at bottom of our IDE, to compile it. `clang` is a compiler that's been pre-installed for our use.
  - Nothing seems to happen, but no errors is good news. We can open the file browser and see that there's a new file, `a.out`, which is the machine code of our program.
- To run it, we can't just click on it. Instead, this program runs in a command-line environment, also known as the terminal. So we type `./a.out` to run it. `.` indicates the current directory:



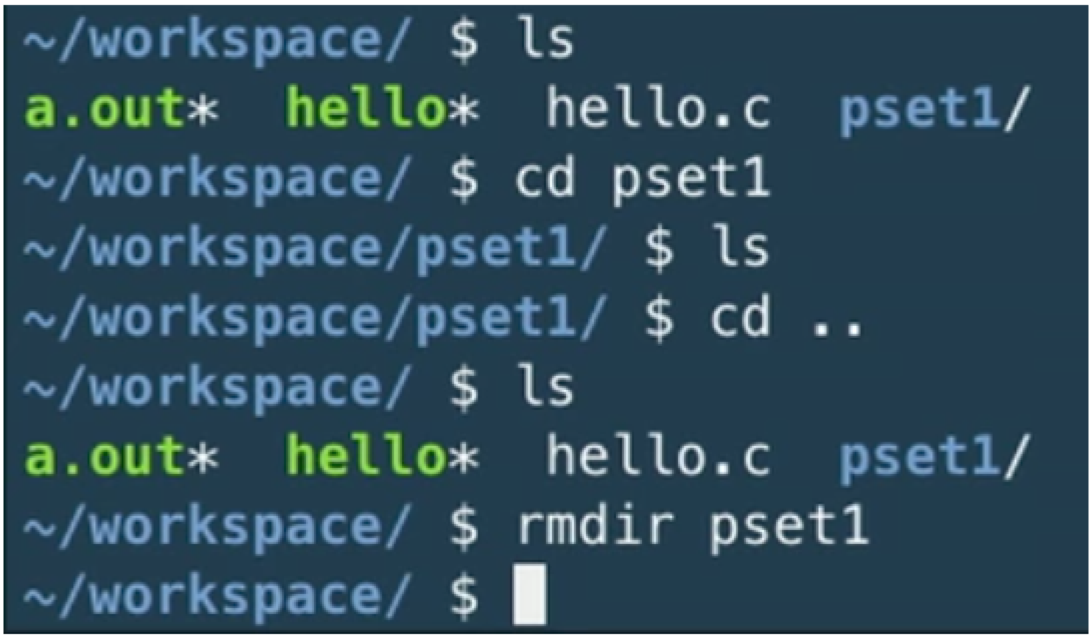
```
~/workspace/ $ clang hello.c
~/workspace/ $ ./a.out
hello, world~/workspace/ $ 
```

- We see the output of our program, but the next line of our terminal prompt is on the same line. We needed to add `\n` in our source code, which is a special, escaped character that adds a new line to what we printed to the terminal.
- Now we can save, compile, and run our program again. We can actually pass in command-line arguments to `clang`, or additional parameters that changes its behavior:



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

  - Here, we are telling `clang` to name the output file `hello`.
- There are other commands built into our environment that we can use:



```
~/workspace/ $ ls
a.out*  hello*  hello.c  pset1/
~/workspace/ $ cd pset1
~/workspace/pset1/ $ ls
~/workspace/pset1/ $ cd ..
~/workspace/ $ ls
a.out*  hello*  hello.c  pset1/
~/workspace/ $ rmdir pset1
~/workspace/ $ 
```

  - `ls` lists the files in the current directory, which we see in blue in the terminal screen above.
  - `cd` lets us change our current working directory (as in `cd pset1`, which we can create new ones of with the file browser on the left. And to change to the parent directory, we can use `cd ..` to

go up one level.
  - Finally, we can use `rmdir` to remove directories.

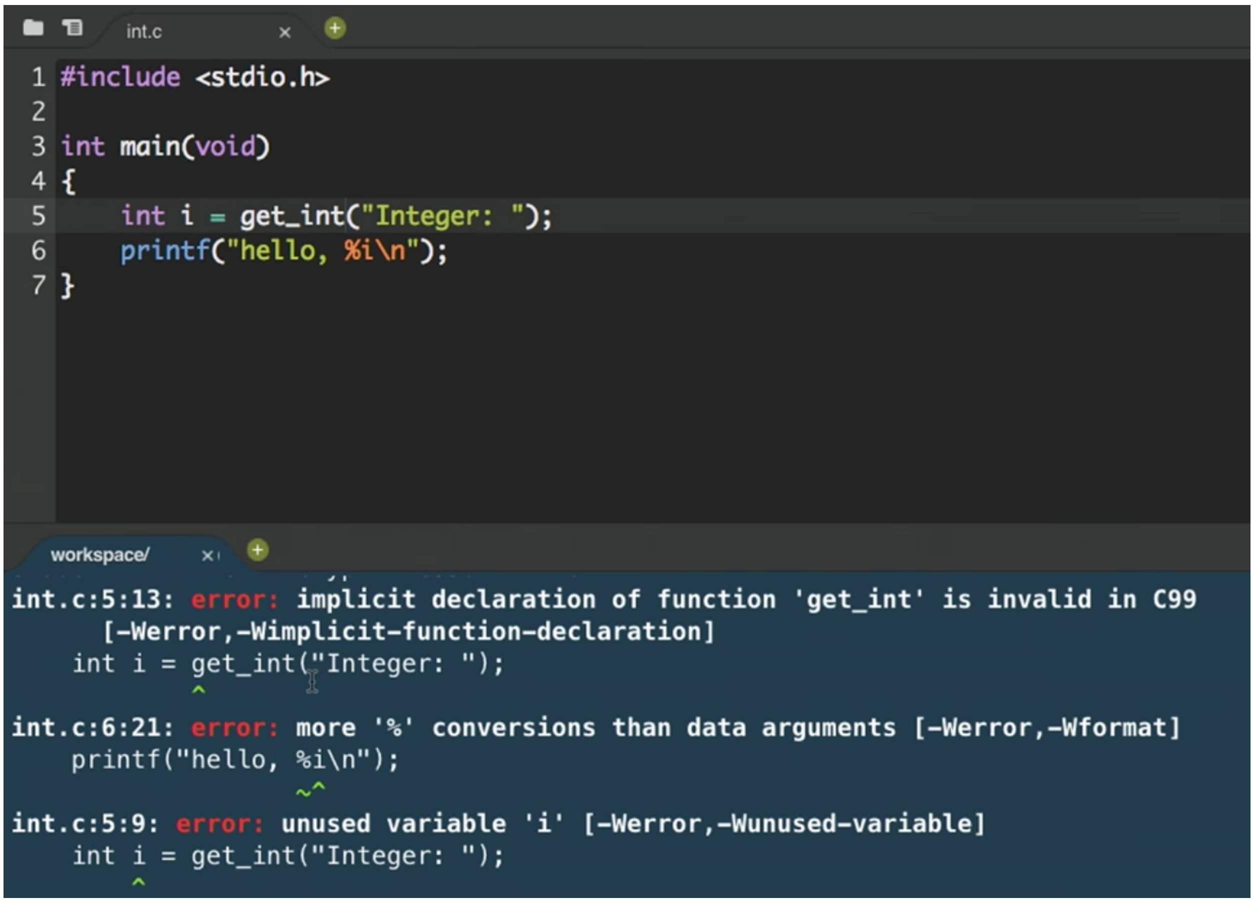
Functions

- We take a volunteer to demonstrate how `printf` is a function we pass arguments to. David hands Sam, our volunteer, a piece of paper with what he wanted to be written on the screen, and Sam copied it to the screen for David. Functions in programming, too, can be considered similar in that we can just call them and use them.
- Some functions relating to input include:
  - `get_char` - gets a character from the user
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long_long`
  - `get_string`
- We'll test out `get_string` with the following program:

```
#include <stdio.h>

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

  - On line 5, we are declaring, or creating, a new variable called `s`, of type `string`. And the value it will store is whatever `get_string` returns. Some functions like `printf` might not return any value, but other functions like `get_string` can. When we call `get_string`, we pass in `"Name: "` as an argument, so it knows what to prompt the user.
  - Next, we want to print out what was stored in our string `s`, so we use the `%s` syntax to include a string inside `printf`. And the string in question is `s`.
- Going back to our list of functions that collect input, we notice that there are other types of data built into C: `double`, `float`, and `long long`.
- Let's start with getting an integer:



```
#include <stdio.h>

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

```
int.c:5:13: error: implicit declaration of function 'get_int' is invalid in C99
      [-Werror,-Wimplicit-function-declaration]
    int i = get_int("Integer: ");
            ^
int.c:6:21: error: more '%' conversions than data arguments [-Werror,-Wformat]
    printf("hello, %i\n");
                   ~^
int.c:5:9: error: unused variable 'i' [-Werror,-Wunused-variable]
    int i = get_int("Integer: ");
        ^
```

- We can save, compile, and run this file as `int.c`. We can use another tool in the IDE called `make` to compile it. By simply running `make int`, `make` will take the file `int.c` and use a compiler to compile it into `int`, which we can run with `./int`.
- At first, we get several errors. Usually, we can start by fixing the first error, save, compile again, and repeat until our program compiles without errors.
- The first error here is telling us that `get_int` isn't actually declared. In fact, it's defined in another library, or set of code we can include, alongside `stdio.h`. `get_int`, along with other functions, live in `cs50.h`, a library written by CS50 staff to help make tedious tasks easier. So we simply need to add `#include <cs50.h>` at the top of our file. (And the source code for the library is stored in a common place in the IDE, where the compiler knows to look for it.)
- Now we can `make` our file again, and notice that we didn't provide the integer `i` into `printf` to plug into our string.
- We add it, and our program compiles and runs as we'd expect, with this final code:

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

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

- The `get_int` function prompts the user over and over, until it receives an integer.

- Let’s take a look at ints.c:

  ```
  // Integer arithmetic

  #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: ");

      // Perform arithmetic
      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);
  }
  ```

  - The first line, started with `//`, is a comment. Comment lines don’t do anything, but are notes for future programmers.
  - In our small program, we first get two integers and store them as `x` and `y`.
  - Then, we print out these variables and various expressions that involve some arithmetic on them. Addition and subtraction are what we might expect. Multiplication is `*`, division is `/`, and the modulo (remainder) operator is `%`.
- We can compile and run our program, and notice that it’s working for `x = 2` and `y = 2`. If we try `x = 1` and `y = 2`, we get a line that reads: `1 divided by 2 is 0`.
- It turns out that integers discard anything after the decimal point, if we try to store some number with a decimal into it. In this case, `1 / 2` should be `0.5`, but the decimal part is thrown away, and all we’re left with is `0`.
- We can fix this in floats.c, where we use variables of the type `float`, for floating-point arithmetic:

  ```
  // Floating-point arithmetic

  #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: ");
  ```

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

  - Notice that we use `%f` instead of `%i`, to indicate that a float should be substituted in.
  - If we wanted to control the number of decimal points printed out, we could write `%.10f` where we want the variable to be substituted in.
  - If we simply used `%f` but passed in integers, the compiler would find it to be an error.
- Let’s look at how we can use conditions:

```
// Conditions and relational operators

#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: ");

    // 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");
    }
}
```

  - All we did is what set up our program to use the example of conditions we say before.
- Let’s look at noswitch.c:

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

int main(void)
{
    char c = get_char("Answer: ");

    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 `==` for a comparison, since a single `=` assigns a value. And C uses `||` 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.
  - We could have had an `if` for `Y` and an `if` for `y`, but using one condition means that we don't need to copy and paste the code that should be run into two places. Correctness is one aspect of code, but design is another. Style, or the indentation, comments, and variable naming, is yet another aspect.
  - Note that we use single quotes around characters, to distinguish them from strings, which we use double quotes to indicate.
- Let's look at another way to implement this program:

```
// switch

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

int main(void)
{
    // Prompt user for answer
    char c = get_char("Answer: ");

    // Check answer
    switch (c)
    {
        case 'Y':
        case 'y':
            printf("yes\n");
            break;
        case 'N':
        case 'n':
            printf("no\n");
            break;
    }
}
```

  - A **switch** is another construct in C where the value of a variable is compared to various cases, and the indented code beneath a matching case will be executed.

    - Notice that we use `break` to indicate that the switch should end. Otherwise, once a matching case is found, all of the code below it will run.
- Let's write our own function that returns a value:

```
// Return value

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

int square(int n);

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

// Return square of n
int square(int n)
{
    return n * n;
}
```

    - Line 5 declares the prototype, or definition, of a function we will write, called `square`. The `int` before `square` indicate that `square` will return an `int`, and `int n` inside the parentheses indicate that `square` takes in an `int` that it will refer to as `n`. We need a prototype because our compiler for C reads in files from top to bottom, and the `main` function calls `square` before it's defined unless we have that line above it.
    - Line 10 calls `square`, passing in `x`, and the return value is not stored but passed directly to `printf`, which will substitute it in the string and print it to the screen. We could define a variable like `int squaredvalue` above, and then substitute it in, but since we are only using it once after we create it, it's considered better design to include it directly where we use it.
    - Finally, in line 14, we write the code for `square`, and return our desired value with the `return` keyword.

Overflow

- In our computers, the number of bytes in our memory is finite. As a result, we can store only so much data. In C, each type of data has a fixed number of bytes allocated to instances of it. For example, every `int` has only 4 bytes in the CS50 IDE.
- As a result, one problem we can run into is **integer overflow**. 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.

- We can see this in a program, overflow.c:

```
// Integer overflow

#include <stdio.h>
#include <unistd.h>

int main(void)
{
    // Iteratively double i
    for (int i = 1; ; i *= 2)
    {
        printf("%i\n", i);
        sleep(1);
    }
}
```

- If we compile and run this, we see:

```
1
2
4
8
16
...
1073741824
overflow.c:9:25: runtime error: signed integer overflow: 1073741824 * 2 cannot be
represented in type 'int'
-2147483648
0
0
...
```

  - We see that our program noticed an error, as we doubled `i` too many times for its value to fit into the bytes allocated for it.
- Another bug can arise when we have **floating-point imprecision**.
- 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.
- Now when we compile and run this, we get:

```
0.10000000000000000555111512312578...
```

- 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. So in this case, the closest approximation a computer can make to `0.1` is that number.
- There are several examples in the real world where these issues create limitations or even dangerous bugs.
- Next time, we'll learn how we can deal with these issues!