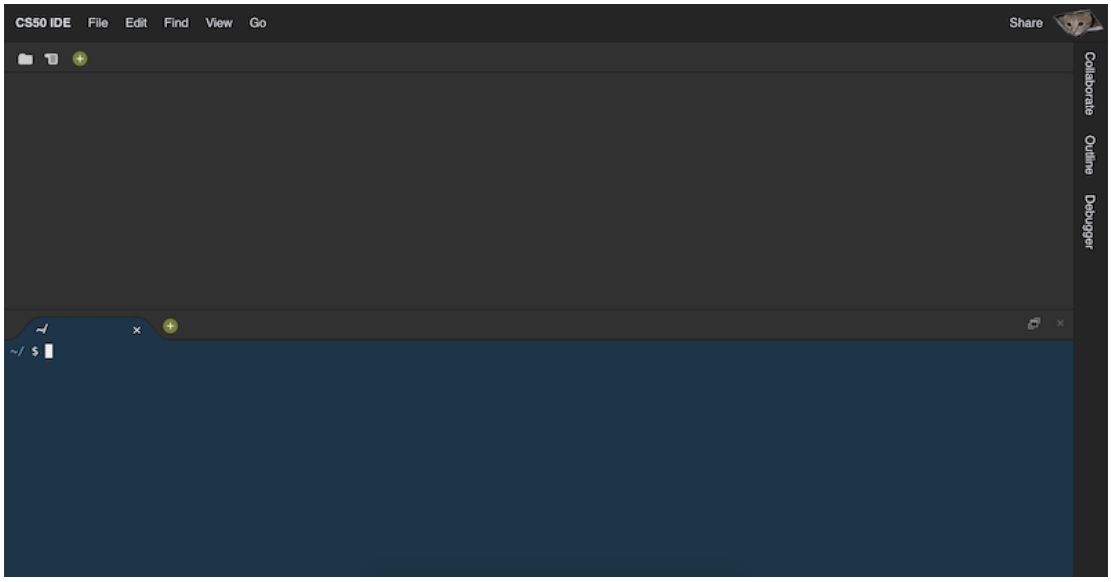
Please read the following to the best of your abilities. Some key terms are highlighted. You will be tested on this content upon completion. Inform the proctor when you finish the passage.

CS50 IDE

To start writing our code quickly, we'll use a tool for the course, CS50 IDE, 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:

```
#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.