This is CS50x

OpenCourseWare

Donate (https://cs50.harvard.edu/donate) (https://community.alumni.harvard.edu/give/59206872)

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)

Hello

Getting Started

CS50 IDE is a web-based "integrated development environment" that allows you to program "in the cloud," without installing any software locally. Indeed, CS50 IDE provides you with your very own "workspace" (i.e., storage space) in which you can save your own files and folders (aka directories).

Logging In

Head to ide.cs50.io (https://ide.cs50.io) and click "Sign in with GitHub" to access your CS50 IDE. Once your IDE loads, you should see that (by default) it's divided into three parts. Toward the top of CS50 IDE is your "text editor", where you'll write all of your programs. Toward the bottom of is a "terminal window" (light blue, by default), a command-line interface (CLI) that allows you to explore your workspace's files and directories, compile code, run programs, and even install new software. And on the left is your "file browser", which shows you all of the files and folders currently in your IDE.

Start by clicking inside your terminal window. You should find that its "prompt" resembles the below.

```
~/ $
```

Click inside of that terminal window and then type

```
mkdir ~/pset1/
```

followed by Enter in order to make a directory (i.e., folder) called `pset1` inside of your home directory. Take care not to overlook the space between `mkdir` and `~/pset1` or any other character for that matter! Keep in mind that `~` always denotes your home directory and `~/pset1` denotes a directory called `pset1`, which is inside of `~`.

Here on out, to execute (i.e., run) a command means to type it into a terminal window and then hit Enter. Commands are "case-sensitive," so be sure not to type in uppercase when you mean lowercase or vice versa.

Now execute

```
cd ~/pset1/
```

to move yourself into (i.e., open) that directory. Your prompt should now resemble the below.

```
~/pset1/ $
```

If not, retrace your steps and see if you can determine where you went wrong.

Now execute

```
mkdir ~/pset1/hello
```

to create a new directory called `hello` inside of your `pset1` directory. Then execute

```
cd ~/pset1/hello
```

to move yourself into that directory.

Shall we have you write your first program? From the *File* menu, click *New File*, and save it (as via the *Save* option in the *File* menu) as `hello.c` inside of your `~/pset1/hello` directory. Proceed to write your first program by typing precisely these lines into the file:

```
#include <stdio.h>

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

Notice how CS50 IDE adds "syntax highlighting" (i.e., color) as you type, though CS50 IDE's choice of colors might differ from this problem set's. Those colors aren't actually saved inside of the file itself; they're just added by CS50 IDE to make certain syntax stand out. Had you not saved the file as `hello.c` from the start, CS50 IDE wouldn't know (per the filename's extension) that you're writing C code, in which case those colors would be absent.

Listing Files

Next, in your terminal window, immediately to the right of the prompt (`~/pset1/hello/ $`), execute

```
ls
```

You should see just `hello.c`? That's because you've just listed the files in your `hello` folder. In particular, you *executed* (i.e., ran) a command called `ls`, which is shorthand for "list." (It's such a frequently used command that its authors called it just `ls` to save keystrokes.) Make sense?

Compiling Programs

Now, before we can execute the `hello.c` program, recall that we must *compile* it with a *compiler* (e.g., `clang`), translating it from *source code* into *machine code* (i.e., zeroes and ones). Execute the command below to do just that:

```
clang hello.c
```

And then execute this one again:

```
ls
```

This time, you should see not only `hello.c` but `a.out` listed as well? (You can see the same graphically if you click that folder icon again.) That's because `clang` has translated the source code in `hello.c` into machine code in `a.out`, which happens to stand for "assembler output," but more on that another time.

Now run the program by executing the below.

```
./a.out
```

Hello, world, indeed!

Naming Programs

Now, `a.out` isn't the most user-friendly name for a program. Let's compile `hello.c` again, this time saving the machine code in a file called, more aptly, `hello`. Execute the below

more aptly, `hello`. Execute the below.

```
clang -o hello hello.c
```

Take care not to overlook any of those spaces therein! Then execute this one again:

```
ls
```

You should now see not only `hello.c` (and `a.out` from before) but also `hello` listed as well? That's because `-o` is a *command-line argument*, sometimes known as a *flag* or a *switch*, that tells `clang` to output (hence the `o`) a file called `hello`. Execute the below to try out the newly named program.

```
./hello
```

Hello there again!

Making Things Easier

Recall that we can automate the process of executing `clang`, letting `make` figure out how to do so for us, thereby saving us some keystrokes. Execute the below to compile this program one last time.

```
make hello
```

You should see that `make` executes `clang` with even more command-line arguments for you? More on those, too, another time!

Now execute the program itself one last time by executing the below.

```
./hello
```

Phew!

Getting User Input

Suffice it to say, no matter how you compile or execute this program, it only ever prints `hello, world`. Let's personalize it a bit, just as we did in class.

Modify this program in such a way that it first prompts the user for their name and then prints `hello, so-and-so`, where `so-and-so` is their actual name.

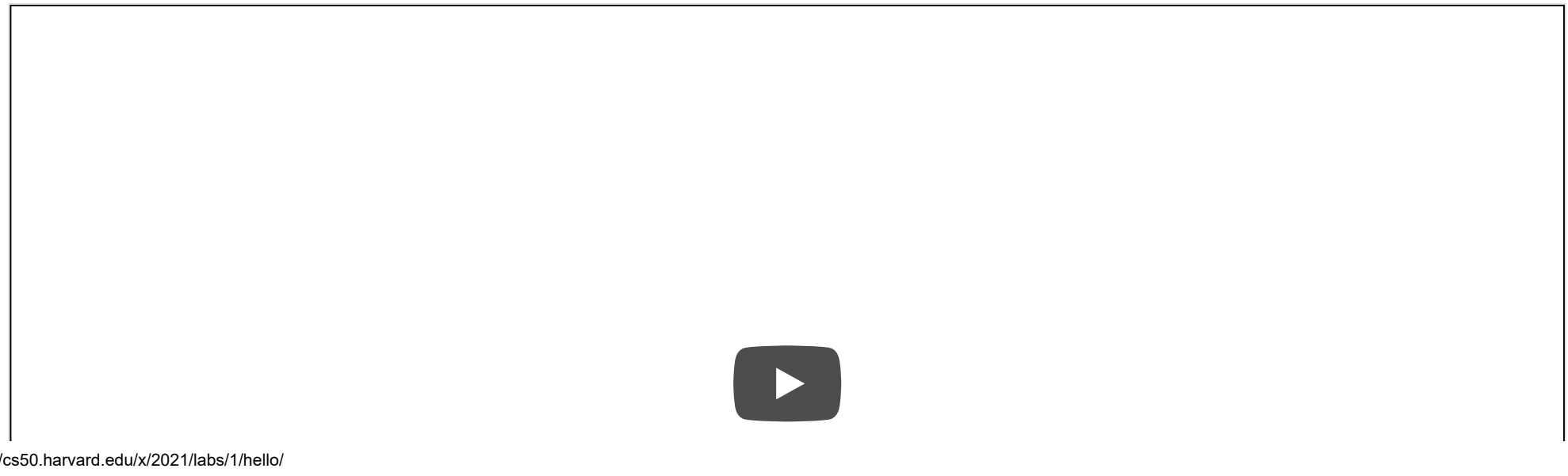
As before, be sure to compile your program with:

```
make hello
```

And be sure to execute your program, testing it a few times with different inputs, with:

```
./hello
```

Walkthrough



Hints

Don't recall how to prompt the user for their name?

Recall that you can use `get_string` as follows, storing its *return value* in a variable called `name` of type `string`.

```
string name = get_string("What is your name?\n");
```

Don't recall how to format a string?

Don't recall how to join (i.e., concatenate) the user's name with a greeting? Recall that you can use `printf` not only to print but to format a string (hence, the `f` in `printf`), a la the below, wherein `name` is a `string`.

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

Use of undeclared identifier?

Seeing the below, perhaps atop other errors?

```
error: use of undeclared identifier 'string'; did you mean 'stdin'?
```

Recall that, to use `get_string`, you need to include `cs50.h` (in which `get_string` is *declared*) atop a file, as with:

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

How to Test Your Code

Execute the below to evaluate the correctness of your code using `check50`. But be sure to compile and test it yourself as well!

```
check50 cs50/problems/2021/x/hello
```

Execute the below to evaluate the style of your code using `style50`.

```
style50 hello.c
```

How to Submit

Execute the below, logging in with your GitHub username and password when prompted. For security, you'll see asterisks (`*`) instead of the actual characters in your password.

```
submit50 cs50/problems/2021/x/hello
```