contents next →

Setup

Ahh, step 1. Don't you love a fresh start on a blank slate? And then selecting that singular brick onto which you will build your entire palatial estate?

Unfortunately, when you're building a *computer program*, step 1 can get... complicated. And frustrating. You have to make sure your environment is set up for the programming language you're using, and you have to figure out how to compile and run your program in that environment.

Fortunately, the program we are building doesn't depend on any external libraries, so you don't need anything beyond a C compiler and the standard library it comes with. (We will also be using the `make` program.) To check whether you have a C compiler installed, try running `cc --version` at the command line (`cc` stands for "C Compiler"). To check whether you have `make`, try running `make -v`.

How to install a C compiler...

...in Windows

You will **need to install some kind of Linux environment within Windows**. This is because our text editor interacts with the terminal at a low level using the `<termios.h>` header, which isn't available on Windows. I suggest using either Bash

on Windows or Cygwin.

Bash on Windows: Only works on 64-bit Windows 10. See the installation guide. After installing it, run `bash` at the command line whenever you want to enter the Linux environment. Inside `bash`, run `sudo apt-get install gcc make` to install the GNU Compiler Collection and the `make` program. If `sudo` takes a really long time to do anything, you may have to fix your /etc/hosts file.

Cygwin: Download the installer from cygwin.com/install.html. When the installer asks you to select packages to install, look in the `devel` category and select the `gcc-core` and `make` packages. To use Cygwin, you have to run the Cygwin terminal program. Unlike Bash on Windows, in Cygwin your home directory is separate from your Windows home directory. If you installed Cygwin to `C:\cygwin64`, then your home directory is at `C:\cygwin64\home\yourname`. So if you want to use a text editor outside of Cygwin to write your code, that's where you'll want to save to.

...in macOS

When you try to run the `cc` command, a window should pop up asking if you want to install the command line developer tools. You can also run `xcode-select --install` to get this window to pop up. Then just click "Install" and it will install a C compiler and `make`, among other things.

...in Linux

In Ubuntu, it's `sudo apt-get install gcc make`. Other distributions should have `gcc` and `make` packages available as well.

The `main()` function

Create a new file named `kilo.c` and give it a `main()` function. (`kilo` is the name of the text editor we are building.)

kilo.c **Step 1** `main`

```
int main() {
  return 0;
}
```

↗ compiles

In C, you have to put all your executable code inside functions. The `main()` function in C is special. It is the default starting point when you run your program. When you `return` from the `main()` function, the program exits and passes the returned integer back to the operating system. A return value of `0` indicates success.

C is a compiled language. That means we need to run our program through a C compiler to turn it into an executable file. We then run that executable like we would run any other program on the command line.

To compile `kilo.c`, run `cc kilo.c -o kilo` in your shell. If no errors occur, this will produce an executable named `kilo`. `-o` stands for "output", and specifies that the output executable should be named `kilo`.

To run `kilo`, type `./kilo` in your shell and press `Enter`. The program doesn't print any output, but you can check its exit status (the value `main()` returns) by running `echo $?`, which should print `0`.

Compiling with `make`

Typing `cc kilo.c -o kilo` every time you want to recompile gets tiring. The `make` program allows you to simply run `make` and it will compile your program for you. You just have to supply a `Makefile` to tell it how to compile your program.

Create a new file literally named `Makefile` with the following contents.

Makefile **Step 2** `make`

```
kilo: kilo.c
	$(CC) kilo.c -o kilo -Wall -Wextra -pedantic -std=c99
```

↗ compiles

The first line says that `kilo` is what we want to build, and that `kilo.c` is what's required to build it. The second line specifies the command to run in order to actually build `kilo` out of `kilo.c`. Make sure to indent the second line with an actual tab character, and not with spaces. You can indent C files however you want, but `Makefile`s must use tabs.

We have added a few things to the compilation command:

- `$(CC)` is a variable that `make` expands to `cc` by default.

- `-Wall` stands for "**all W**arnings", and gets the compiler to warn you when it sees code in your program that might not technically be wrong, but is considered bad or questionable usage of the C language, like using variables before initializing them.
- `-Wextra` and `-pedantic` turn on even more warnings. For each step in this tutorial, if your program compiles, it shouldn't produce any warnings except for "unused variable" warnings in some cases. If you get any other warnings, check to make sure your code exactly matches the code in that step.
- `-std=c99` specifies the exact version of the C language **st**an**d**ard we're using, which is C99. C99 allows us to declare variables anywhere within a function, whereas ANSI C requires all variables to be declared at the top of a function or block.

Now that we have a `Makefile`, try running `make` to compile the program.

It may output `make: `kilo' is up to date.`. It can tell that the current version of `kilo.c` has already been compiled by looking at each file's last-modified timestamp. If `kilo` was last modified after `kilo.c` was last modified, then `make` assumes that `kilo.c` has already been compiled, and so it doesn't bother running the compilation command. If `kilo.c` was last modified after `kilo` was, then `make` recompiles `kilo.c`. This is more useful for large projects with many different components to compile, as most of the components shouldn't need to be recompiled over and over when you're only making changes to one component's source code.

Try changing the return value in `kilo.c` to a number other than `0`. Then run `make`, and you should see it compile. Run `./kilo`, and try `echo $?` to see if you get the number you changed it to. Then change it back to `0`, recompile, and make sure it's back to returning `0`.

After each step in this tutorial, you will want to recompile `kilo.c`, see if it finds any errors in your code, and then run `./kilo`. It is easy to forget to recompile, and just run `./kilo`, and wonder why your changes to `kilo.c` don't seem to have any effect. You must recompile in order for changes in `kilo.c` to be reflected in `kilo`.

In the next chapter, we'll work on getting the terminal into *raw mode*, and reading individual keypresses from the user.