

# ENGG1003 - Lab 1

Brenton Schulz

## 1 Introduction

This laboratory exposes you to the fundamental tools required to write computer programs in C. No prior programming experience is assumed but if you don't have basic computer literacy you're gonna have a bad time.

## 2 C Programming Basics

In order to write programs in C (and most other languages) the following software tools are required:

- An *editor*, to create and edit raw text files.
- A *compiler*, to convert your text files into an *executable* file.

A programming editor is very different to a *word processor* (eg: Microsoft Word) in that it displays and stores raw ASCII text only. What you see printed to the screen represents the *actual data* stored in the file. By contrast, Word will store a combination of text and display formatting and, as such, is not suitable for writing code.

Programming editors will generally have features optimised for coding, such as:

- Syntax highlighting
- Line numbering
- Auto completion
- Pre-emptive error notifications
- Communication with the compiler to highlight errors
- Automatic indenting
- Highlighting of matching blocks
  - ie: an easy method to find matching pairs of ( ), { }, " ", etc.

It is hoped that you will discover these features and learn to work with them. In time you will learn which features work well with your style and which simply get in the way.

For the time being the “compiler” noun will be used to colloquially reference a highly complex set of software tools which turn your source code into an executable binary file. You will be shielded from the details until otherwise necessary.

### 2.1 Introduction to OnlineGDB

OnlineGDB is a basic (*very* basic) browser-based development environment for a variety of programming languages. It gives you access to an editor, a small amount of cloud storage, compiler, and standard input / output. It also contains a *debugging* feature however for technical reasons<sup>1</sup> we won't be using it.

All compilation and execution is performed on the OnlineGDB server. As such, the service has an incredibly low barrier to entry: there is (almost) zero installation/configuration required to get started running code.

**Task:** Open a web browser and navigate to <http://www.onlinegdb.com>.

---

<sup>1</sup>It only allows one debug session per IP address. The entire campus uses the same public IP so we can't use it in labs.

**NB:** If a demonstrator sees you using Edge or IE they may instinctively think you need more help than students using Chrome or Firefox.

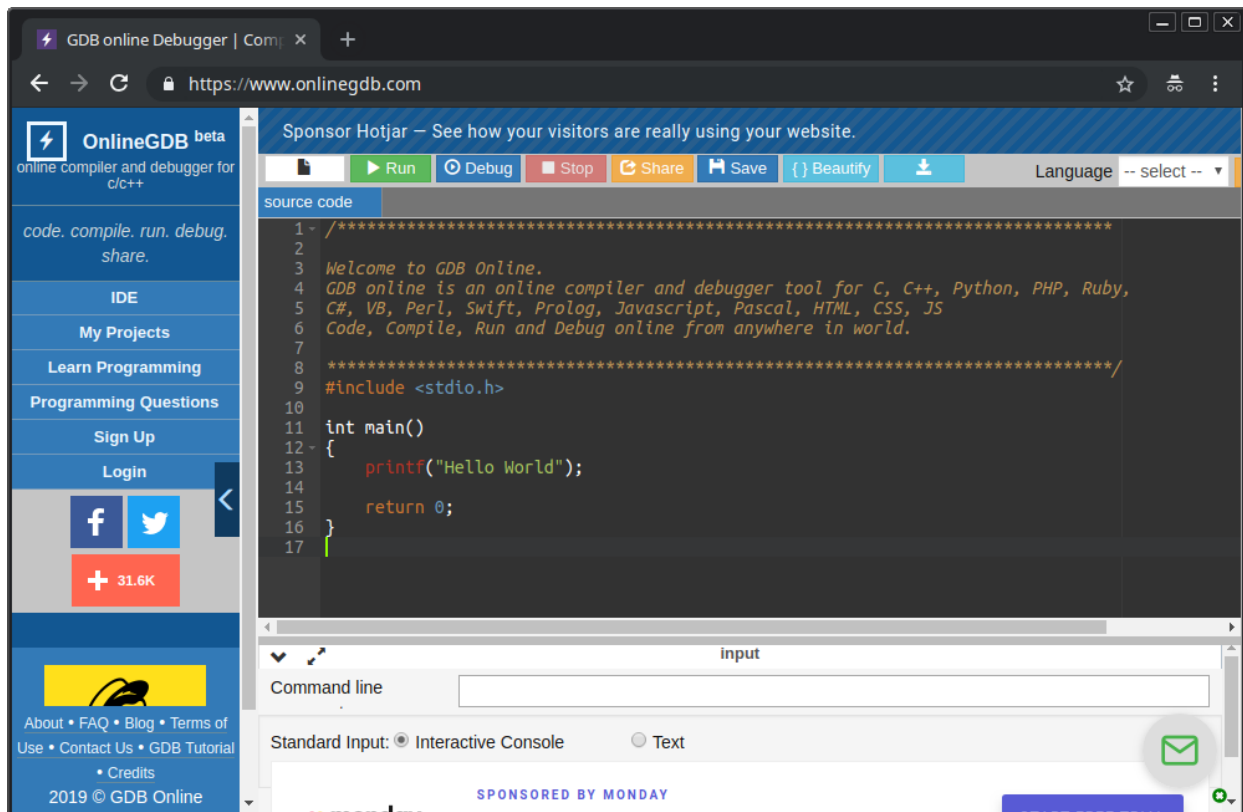


Figure 1: The view when OnlineGDB is first opened.

After OnlineGDB has loaded you will be greeted with the screen seen in Figure 1. The large area in the middle is the editor screen, this is where you will type C code. Immediately you can observe that this editor supports line numbering and syntax highlighting.

Above the editor is a toolbar which, from left to right, performs the following functions:

1. Create a blank new file
2. Run the project
3. Debug the project (not used in ENGG1003)
4. Stop execution of a running program
5. Share - Generates a link to your current source code
6. Save - When logged in this saves the project files to your personal cloud storage
7. { } Beautify - Modifies your code's whitespace to adhere to the OnlineGDB indenting style (NB: I tried this at time of writing and it didn't work on my personal computer. Go figure.)
8. Download - Downloads the currently viewed file.

The area below the editor is where standard output is written to and standard input read from. When the code is run its appearance changes to that of a basic console (ie: the GUI elements disappear and it becomes just text).

**Task:** Configure OnlineGDB to run C code by selecting “C” from the “Language” drop-down box in the upper-right. This website supports many languages<sup>2</sup> so feel free to come back here later if you're interested

<sup>2</sup>MATLAB is not one of them because it is a *very* expensive commercial package

in learning any of the others. Python, although not taught in an Engineering degree, is a common choice for Engineering PhD students as a free MATLAB alternative and is probably worth playing around with.

**Task:** Click the green Run button. The box at the bottom of the screen will produce a “Compiling” animation and, after execution of the template code, will produce the output seen in Figure 2.

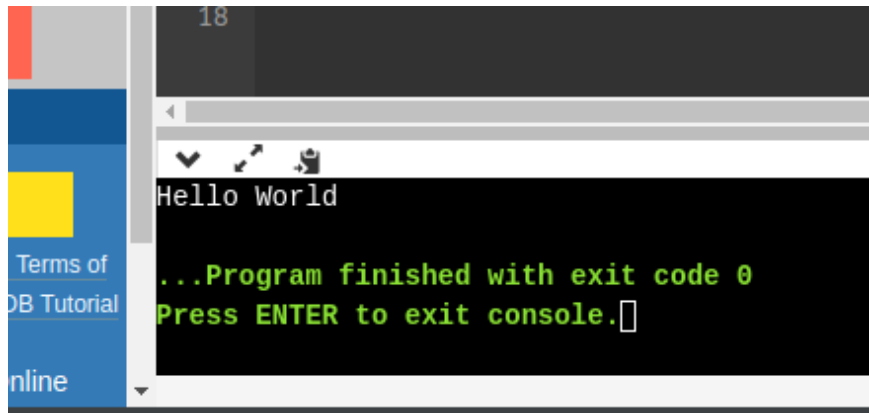


Figure 2: A cropped screenshot showing the “Hello World” program output.

**Off-topic note:** Remember the “returns zero to the operating system” comment in lecture 1? Well that’s what the text “...Program finished with exit code 0” is referencing. The 0 is the number that `main()` returned. We will learn about *function return values* later in the semester.

#### Tasks:

1. Running the default template demonstrates *standard output*, modify the code to match that in Listing 1. While making the changes you will observe OnlineGDB’s auto-complete features. When, for example, you type a double quote “ character it *automatically* types two and places the cursor between them. It will also provide auto-complete suggestions, although many of them will be inappropriate (it is, after all, just a computer program; not a science fiction grade artificial intelligence).

What other “helpful” editor behaviour did you notice? Some of it will be annoying at first (some of it will be annoying *forever*) but learning to work with the editor’s features will improve your coding speed in the long term.

```

1 #include <stdio.h>
2
3 int main() {
4     int k;
5     scanf("%d", &k);
6     printf("You entered: %d\n", k);
7     return 0;
8 }
```

Listing 1: A basic C program which demonstrates input and output.

2. After editing the code press Run. After it is compiled you will notice that the console is just displaying a cursor. This is because `scanf()` waits for data to be typed (specifically, it will wait until a new line character, ASCII value 10, is sent).
3. Type an integer and press enter / return. There will be some “lag” because the data is being sent to OnlineGDB’s server before being displayed.
4. After pressing enter the console should display the text “You typed: 123”.
5. Run the program again except this time don’t type just an integer, try typing a word, or a word containing a number, or a number followed by letters (with and without a space). What is the behaviour each time? Are you getting annoyed by the slow compile time and lag yet? OnlineGDB may be simple but it is, at times, a compromise.

### 3 Compiler Errors and Warnings

This section will demonstrate several common compiler *errors* and *warnings*.

An error occurs when the code does not meet the rigorous and unambiguous syntax rules specified in the ANSI C standard and, as such, the compiler does not know how to interpret the code. For example, if you miss a " symbol inside a `printf()` where does the text to be printed end? This can't be assumed because there are not enough rules about what you can and can't write in this context. As such, missing a " will produce a *syntax error*.

By contrast, a warning occurs when the code is in some way "mildly" problematic but the compiler is still able to make assumptions about what the code should do and produce a binary executable. For example, if the `return 0;` at the end of `main()` is missing the compiler will throw it in for you because, well, what else is it going to do at the end of `main()`?<sup>3</sup>

Unfortunately compiler errors can be *highly* technical and difficult to interpret. Furthermore, they often report an error on a line *after* the actual mistake! The exercises below will give you some experience with interpreting compiler errors but this is a topic in which you will likely engage in "life-long learning".

#### 3.1 Missing a semicolon

##### Task:

1. If you have not done so already, get the code shown in Listing ?? working correctly.
2. Remove the semicolon (the ; character) from the end of line 4.
3. Compile the code, what error did the compiler produce? Which line does it say the error is on?

The compiler errors will look something like Listing 2.

Listing 2: The multiple errors produced by removing a *single* semicolon.

```
main.c: In function main :
main.c:14:5: error: expected '=', ',', ';', 'asm' or '__attribute__'
        before 'scanf'
        scanf("%d", &k);
        ^
main.c:14:18: error:      k      undeclared (first use in this function)
        scanf("%d", &k);
        ^
main.c:14:18: note: each undeclared identifier is reported only once for
        each function it appears in
```

Lets break this down a bit. Each error starts with a location, the syntax is: `file:line:column`. So `main.c:14:5` means the file `main.c` at line 14, column<sup>4</sup> 5.

The first error (`error: expected '=', ',', ';', 'asm'... etc.`) is telling you that something was missing and the compiler noticed the omission at line 14. In this case we removed a ; so that's what is missing. Note, however, that while the error was reported to be on line 14 the omission was actually on line 13! Also observe the extra technical jargon that you don't understand yet (what is `__attribute__`?). Some of these are, in fact, beyond the scope of ENGG1003, you have to get used to reading text you don't understand and extracting the small pieces of information you actually need.

Let's look at the second error: `'k' undeclared`. Wait, wasn't `int k` still in our source code? Why is the compiler complaining that it is undeclared when it is *right there*? The problem is that by removing the

<sup>3</sup>The programmer could want `main()` to return a value other than zero but it is overwhelmingly common to just return zero here. Historically, a program returning zero means "the program finished without error" and non-zero indicates some kind of error code.

<sup>4</sup>*Column* means the number of characters since the start of the line

semicolon the declaration syntax was incorrect, so the compiler did not interpret that line as a declaration. It didn't find the expected ; threw its hand in the air and gave up.

The final line (each undeclared identifier... etc.) is just telling you that if you forget to declare a variable the compiler will only tell you about it *once*. This is because it could appear multiple times and producing the same error over and over is redundant and confusing.

Lets try a different syntax error.

### 3.2 Missing a Quote Symbol

**Task:** Remove the closing " from the printf line so it reads:

```
1 printf("You entered: %d\n, k);
```

You will see an immediate change to the syntax highlighting; all characters between where the " was and the end of the line are now green, as if they were still inside the double quotes. Take-home message: *pay attention to syntax highlighting!*

Again, removing a *single character* generated a slew of errors:

```
main.c: In function main :
main.c:15:12: warning: missing terminating " character
    printf("You entered: %d\n, k);
           ^
main.c:15:5: error: missing terminating " character
    printf("You entered: %d\n, k);
    ^
main.c:16:5: error: expected expression before return
    return 0;
    ^
main.c:17:1: error: expected ; before } token
    }
    ^
```

Observe how missing the closing " generates both an error and a warning at different locations. Why does it do this? To be honest, this is beyond my experience; I've only been using gcc for 20 years, there's always *something* that you haven't learned yet. In the end it doesn't matter, an error exists that needs to be fixed before the compiler can output an executable.

Missing the " generated two other errors. By missing the " the printf *expression* was not completed. In C an expression can be thought of as "a complete line of code that does something unambiguous". This is a *very* informal definition but it will do for now. In order to complete the printf expression the syntax rules require the closing ", a closing right parenthesis ) [to match the opening left parenthesis (] and a semicolon ;. Removing the " character makes the compiler think that the existing ");" string at the end of the line is actually part of the data to be printed, not part of the C expression syntax. As such, the expression was never finished, leaving the compiler with an unresolved tension that will stay with it *all day*<sup>5</sup>.

Remember how missing a ; caused errors on lines *below* where the error actually was? Well the same thing has happened here. Because the printf() line was malformed an error occurred on the line below it (expected expression before return) *and* the one below that (expected ; before } token); and you thought your *ex* was highly strung!

<sup>5</sup><https://xkcd.com/859/>

### 3.3 Failing to Define `main()`'s Return Type

Enough about errors, the final example in this section demonstrates a compiler *warning*. Unfortunately, OnlineGDB doesn't display the compiler output if only warnings are generated. If you are running `gcc` in Linux (as described at the end of this lab document) you won't have this problem<sup>6</sup>.

**Task:** Remove `int` before `main` and the semicolon in `return 0`, as per Listing 3

```

1 main()
2 {
3     int k;
4     scanf("%d", &k);
5     printf("You entered: %d\n", k);
6     return 0
7 }
```

Listing 3: Example code which generates a warning.

Attempting to compile this code generates the following compiler output:

```

main.c:11:1: warning: return type defaults to   int   [-Wimplicit-int]
main()
^
main.c: In function   main   :
main.c:17:1: error: expected      ;      before      }      token
}
^
```

The error can be ignored here, let's focus on `warning: return type defaults to 'int'`. The keyword which goes before `main()` specifies its *return type*. This is the type of data which it sends back to the operating system on program exit, we will study function return types in later weeks. To my knowledge, operating systems only support the `int` return type (ie: an integer) so if you leave it out the compiler can quite safely *assume* that's what should be there and only issue a warning instead of an error.

Generally speaking, warnings should be fixed when you see them. With experience you will know when they *need* to be fixed (because the compiler is assuming something incorrectly) and when they can be ignored (because you have more important stuff to fix first).

No doubt you will see *many* other errors and warnings during your foray into C programming. I still occasionally see new ones! If in doubt, throw the compiler output into Google, chances are someone on Stack Overflow<sup>7</sup> has written a good explanation about it.

<sup>6</sup>Yeah, I'm very biased towards Linux. It gets a bad rap because games, Microsoft Office, Adobe, and Autodesk don't support it but it is otherwise *overwhelmingly* dominant. It runs: mobile phones (Android is a Linux derivative, iOS is a Unix variant), servers, supercomputers, "internet of things" devices, etc. As engineers you will probably see it in many places throughout your career, just hidden from the end user.

<sup>7</sup>Why are you even taking this course? Stack Overflow is all you *really* need, right?

## 4 Comments

So far all the code examples have been very basic and (hopefully) easy enough to read. In “real” projects, however, this is rarely the case and the code needs some kind of explanation for the reader to quickly, and accurately, understand what it does<sup>8</sup>.

You will hopefully gain experience with code comments as you progress through this course (and the rest of your career!) but for now we will just see the basics. Any text in the source files<sup>9</sup> which:

- Is between `/*` and `*/`, or
- Is between `//` and the end of the line

is *totally ignored* by the compiler and called a “comment”.

Code comments are a place for you to explain how your code works, or what it does, to future people who work with it. It is also a great place to leave little memos to yourself, typically in the form of:

```
// TODO: Fix this because <reasons>.
```

In fact some code editors (like the Linux editor `vim`) will automatically highlight the text `TODO` so it is easy to find.

**Task:** Take whatever source code is currently shown in OnlineGDB and add some comments in various places. Compile the code and observe that they have no effect.

Students frequently ask what should and should not be commented. In this course your comments should be written such that a student who is on track to achieve 50% can understand what your code does without having to consult external reference material. This will seem overly verbose but it will be used in lieu of a better standard. As a general rule: if you needed to look up something in a reference manual when writing the code write a comment explaining it. For example:

```
1 printf("%d\n", x); // %d formats an integer
```

Anything which, at first glance, appears to be at least a little bit cryptic should probably get a comment.

---

<sup>8</sup>Despite what you might believe experienced engineers and programmers are not wizards, our understanding is not *magic*, it is based on experience and frequently needs supplementing with code comments.

<sup>9</sup>Did I forget to define what source code, or source files, are? It is just another word for any programming code. It can also be called a source *listing*.

## 4.1 Intrinsic Documentation

As a supplement to comments, *intrinsic documentation* is the idea behind choosing informative names for variables and functions<sup>10</sup>. Compare, for example, the following two code listings:

```
1 int main()  
2 {  
3     float x, y;  
4     scanf("%f", &x);  
5     y = x*9.0/5.0 + 32.0;  
6     printf("%f\n", y);  
7 }
```

```
1 int main()  
2 {  
3     float tempFahrenheit, tempCelcius;  
4     scanf("%f", &tempCelcius);  
5     tempFahrenheit = tempCelcius*9.0/5.0 + 32.0;  
6     printf("%f\n", tempFahrenheit);  
7 }
```

The first one, especially without comments, is cryptic and not possible to understand without more information. The second, even if you don't understand all the code, quite intuitively converts Celcius to Fahrenheit.

You will observe that my notes break intrinsic documentation rules all the time. Sorry about that (the examples in notes will also tend to not do anything particularly useful, so it gets difficult when you want to document "nothing").

---

<sup>10</sup>No, you don't need to know what a function is yet. It is in week 5 or so.



## 5 Basic Arithmetic in C

So, it is Page 9 of these notes and we haven't done anything *useful* yet! Ok *fine*, lets do some data processing.

In this section we will do some basic arithmetic on numbers which are read from the console (ie: read from `stdin`). Since we haven't learnt much C (or even much programming in general) these examples are either going to be a bit boring (because they don't do much) or look like black magic (because you haven't learned how they work yet). I have to ask you just to go through the motions at this stage, hopefully exposure to the code below now will make it easier to understand how it works in the coming weeks.

Lets start with the basic C arithmetic operators:

Operation	C Symbol
Addition	+
Subtraction	-
Multiplication	*
Division	/

Table 1: Basic arithmetic operators in C

These are all *binary* operators, meaning they operate on two *operands*<sup>11</sup>. This may feel obvious, but C includes several *unitary* operators and even a *ternary* one (that nobody uses because it's confusing). Each operand could be a variable (eg: `a + b`), a constant (eg: `a + 5`) or some complicated expression (eg: `(2*a + 6) / (12 + b - x)`)<sup>12</sup>.

We begin with a basic example:

Modify your code to match that of Listing 4.

```

1 #include <stdio.h>
2
3 int main() {
4     int k;
5     printf("Enter an integer: ");
6     scanf("%d", &k);
7     k = 2*k;
8     printf("That integer doubled is: %d\n", k);
9     return 0;
10 }
```

Listing 4: A basic arithmetic example

Notice that this code has a *slightly* improved user experience to previous examples; it produces a prompt (Enter an integer: ) which tells you what to do. Also notice that the first `printf()` does *not* end with a `\n` (newline), so the number you type appears on the same line as the prompt.

The line `k = 2*k` takes the value of `k`, multiplies it by 2, then assigns that result back into the variable `k`. This is a *crucial* concept in programming: the `=` symbol is **not** equality, `k = 2*k` is **not** an equation, it is *assignment*. Assignment takes what's on the right, evaluates it, then stores it into the variable on the left.

### 5.1 Operator Precedence Basics

Just like in “normal” algebra different operators take precedence over others (PEMDAS / BODMAS anyone? How about those Facebook posts where *everybody* gets this wrong?). You can view the full C op-

<sup>11</sup>An operand is one of the “things” that a mathematical operator operates on. Eg: In `a + b` the variables `a` and `b` are operands.

<sup>12</sup>I contemplated leaving this closing parenthesis out to give you unresolved tension but I'm not quite *that* evil.

erator precedence here: [https://en.cppreference.com/w/c/language/operator\\_precedence](https://en.cppreference.com/w/c/language/operator_precedence) but you don't need to learn all of it now, it will be covered more in future lectures.

For now we will observe some basic examples and look at how engineers can deal with the complexity of full C operator precedence.

The predominant engineer's approach to this topic is to vaguely learn how the language behaves and then throw parentheses everywhere just to be *absolutely sure* that the intention is not ambiguous. In the end it is *other people* who will have trouble reading your code, not the compiler, so you might as well make their job easy.

**Task:** Implement the following equation in C:

$$y = 2x + 3 \times 5 \quad (1)$$

using the template in Listing 5.

```

1 #include <stdio.h>
2
3 int main() {
4     float x;
5     float y;
6     printf("Enter a number: ");
7     scanf("%f", &x); // Note change of %d to %f
8     // y = ??? uncomment this line and write your answer instead
9     printf("y: %f\n", y);
10    return 0;
11 }
```

Listing 5: A basic arithmetic example

You will notice a few new things about Listing 5. Firstly, it uses the `float` datatype. This type will store any real number with a magnitude of  $1.2 \times 10^{-38}$  to  $3.4 \times 10^{38}$  with a precision of approximately 6 decimal digits. Its bigger brother, the `double` will be seen later.

The other major change is that inside `scanf` and `printf` the `%d` has changed to `%f`. The `f` stands for floating point, which is a standard method<sup>13</sup> for storing fractional numbers using only binary integers. The details are beyond this course, but you can read the details on Wikipedia: [https://en.wikipedia.org/wiki/IEEE\\_754](https://en.wikipedia.org/wiki/IEEE_754).

Back to the task at hand, have a go at implementing Equation 1 and see if the result works. Observe that you don't need to force precedence with parentheses because, in C, the multiplication operations are performed before addition.

As practice, implement the following equations (what happens when you choose  $x$  to force a division by zero?):

1.  $y = \frac{9}{5}x + 32$
2.  $y = \frac{x}{1-x}$ . This one will require parentheses.
3.  $y = x^2 + 2x$ . **NB:** C does *not* include an exponent operator. Implement  $x^2$  as `x*x`:
4.  $y = \frac{x+2}{x-1}$

<sup>13</sup>The other major standard is known as *fixed point*. Details are beyond ELEC1003 but you'll probably see it if you study enough digital signal processing (DSP).

## 6 A Very Brief Introduction to Git

Programming projects typically involve two potentially problematic issues:

1. Multiple people contribute to the same project
2. Programmers frequently want to “roll-back” to an earlier code version.

The second item arises from the fact that after going down a problematic design pathway it is typically much faster to scrap the idea and start again than it is to try and fix all the problems you just created. The fastest way to do *this* is to load up known-working code from the past and go from there.

As such, programmers designed so-called *versioning* subsystems. These are ways of storing data which allows someone to access either a current or past version.

In this course it will be *recommended* that you take advantage of the modern versioning system known as Git. It was developed for tracking code in the Linux kernel and has since expanded to be an industry standard; even Microsoft purchased the website GitHub because it was *just better* than anything else they had developed internally.

**Task:** Navigate to <https://github.com/bschulznewy/engg1003/blob/master/Lectures/Wk1/Friday/LectureFriWk1.pdf>, this is the GitHub “repository” that I have been using when writing ENGG1003 content. In particular, it is the lecture notes for the Friday Week 1 Lecture (I’m subtly trying to get you to read them ahead of time, apologies for improving your study efficiency).

The link is for a PDF document and you will see that GitHub automatically renders it inside the website. It will attempt to display any file directly linked to.

But that’s not what we’re here for, I want to show you the real power of Git: version tracking. Click on the “History” button to the top right of the PDF preview. You will be greeted with a view similar to that shown in Figure 3. It lists the git *commits* made which effect that file. A commit is a snapshot of what the entire repository looked like at a particular point in time. Each commit happens manually (ie: I type a command to cause a commit to be created) and is attached to a “commit message” which (hopefully) describes the changes which were made.

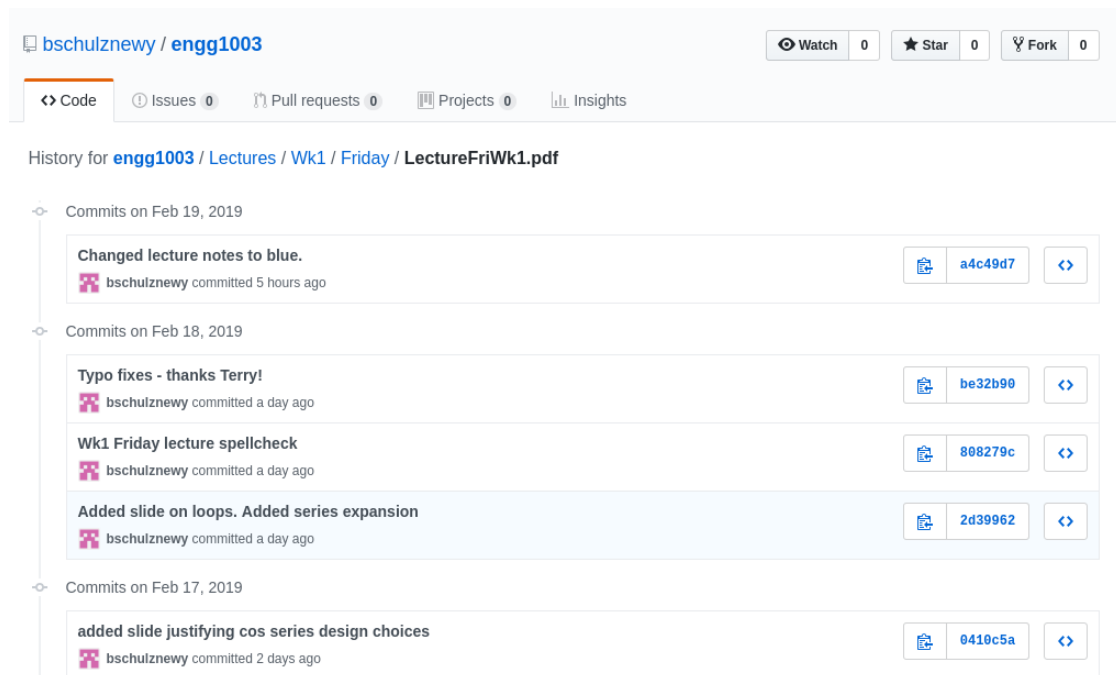


Figure 3: The git repo history for the Friday lecture notes.

Click on the top one (specifically the text “Changed lecture notes to blue.”) and you will see what changed in this particular git commit.

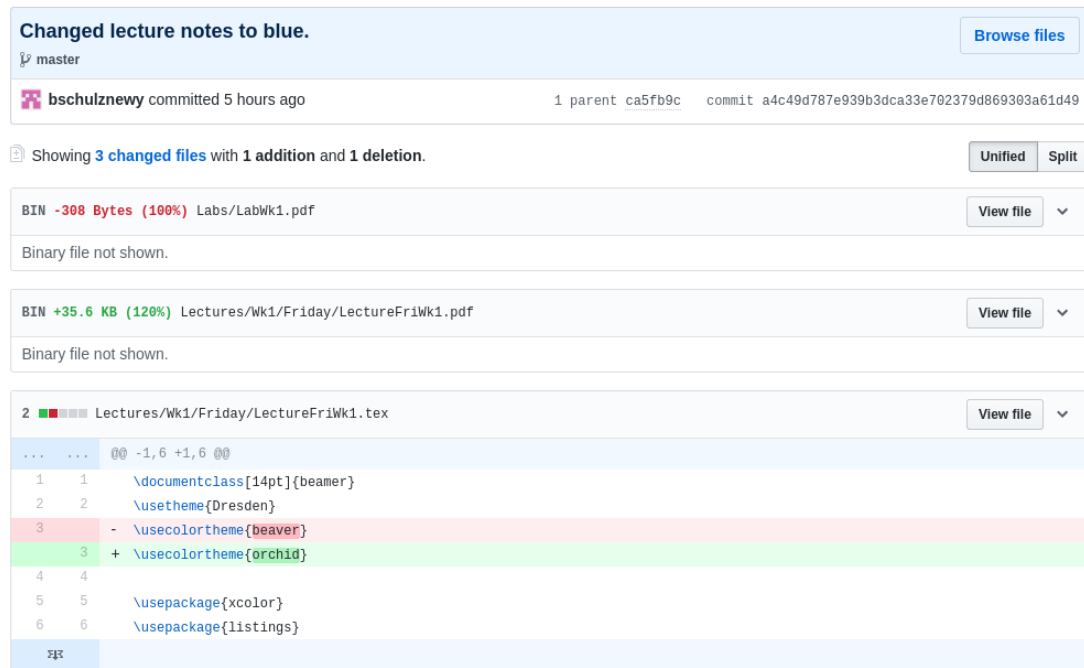


Figure 4: The changes which occurred in a particular commit.

The box at the bottom of Figure 4 shows the actual changes which occurred. In this case I changed the colour theme for the lecture notes to roughly match that of the first lecture. The changes to the PDF are not shown because they would be unintelligible garbage (it is a binary file, not text).

Browse through the other commits, feel free to also browse the other files in the repository.

**Task:** Go to [www.github.com](https://www.github.com) and create a GitHub account. You will use it in later weeks when we swap to a development environment which supports Git. You can also use a GitHub account to login to OnlineGDB, allowing you to have your source files stored on their server.

**Task:** Read more about how Git works and what it does here: [https://medium.com/@graeme\\_boy/how-to-use-git-and-github-60c4ba44ef40](https://medium.com/@graeme_boy/how-to-use-git-and-github-60c4ba44ef40).

## 7 Getting started with C in Ubuntu Linux - Optional

Linux is not supported by university IT but, personally, I find it to be a fantastic development platform. The following instructions are completely optional, only do this if you are keen to learn.

Installing Ubuntu is beyond the scope of this document (and course). If you think this is a daunting task I would recommend only using the officially supported tools to complete ENGG1003. There are many thousands of websites and YouTube videos which will guide you through the Ubuntu (or Mint, Arch, etc.) installation process. **NB:** Installing a new operating system can *very easily* destroy all existing software on a machine. Don't do this if you don't know what you're doing.

That out of the way, here's how you can get started. These steps assume a fresh Ubuntu 18.04 installation, in other distributions YMMV<sup>14</sup>:

1. The C compiler in Linux (and OnlineGDB, and *many* other platforms) is gcc (the GNU C Compiler). To install it, open a terminal (ctrl + alt + t) and type:

```
$ sudo apt install gcc libc6-dev gedit
(the $ indicates the terminal prompt, don't type that character)
```

When prompted, enter your password, wait a few seconds, press enter if it wants installation confirmation, and wait a few more seconds. An Internet connection is required for apt to download the required software.

The libc6-dev package provides all the basic C libraries (printf etc.) and gedit is a basic text editor.

2. Lets make a new directory for writing C files, type:

```
(a) $ mkdir c
(b) $ cd c
```

The first command creates a directory called "c" and the second "changes into" that directory.

3. Create a new .c file. We will do this in gedit (because it is easy and simple) but there are many others (the more nerdy among you may want to learn vim or emacs. They are *very* powerful editors).

Type: `$ nohup gedit test.c &`

(The & symbol at the end of a command runs the command "in the background". This gives you the command line back straight away, instead of having to quit gedit first. Preceding the command with nohup stops gedit from closing if you close the terminal window)

4. Type out the code seen in Figure 5.
5. Click the Save button (or type ctrl + s).

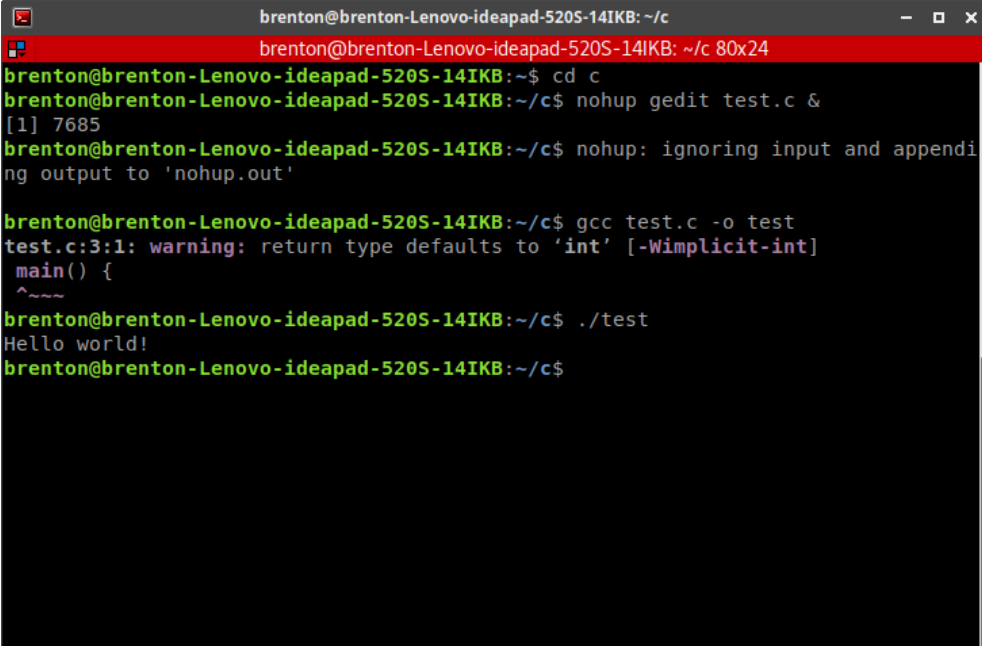


Figure 5: The gedit window with some C code typed out.

6. Move the *keyboard focus* back to the terminal (ie: click the terminal window).

<sup>14</sup>Your mileage may vary. ie: this step might not work.

7. If you can see the command prompt yet, press `enter`
8. You can type `ls` to see a list of all files in the current directory. `test.c` should be there.
9. To compile the `.c` file run: `$ gcc test.c -o test`  
This will create a binary executable called `test`. If the `-o command line argument` is not given `gcc` the binary file defaults to the name `a.out`.
10. Run `test` by typing: `$ ./test`  
The `./` is a special character string meaning “relative to the current directory”. If you try to run `test` from any other directory nothing will happen because `test` is a built-in command. With most other names you will get a “Command not found...” error.
11. When the program runs you should see something similar to Figure 6.
12. Go back to `gedit` and keep coding as you desire. Return to the command line to run `gcc` to re-compile your code. **NB:** The command line has a *history* feature, pressing the up arrow will scroll through past commands, *you don't need to type them out from scratch*.
13. If you enjoyed this you're a bit weird, welcome to the club! Recommended further reading would be a tutorial on `make` followed by investigations into more powerful editors like `vim`.



```
brenton@brenton-Lenovo-ideapad-520S-14IKB: ~/c
brenton@brenton-Lenovo-ideapad-520S-14IKB: ~/c 80x24
brenton@brenton-Lenovo-ideapad-520S-14IKB:~$ cd c
brenton@brenton-Lenovo-ideapad-520S-14IKB:~/c$ nohup gedit test.c &
[1] 7685
brenton@brenton-Lenovo-ideapad-520S-14IKB:~/c$ nohup: ignoring input and appendi
ng output to 'nohup.out'

brenton@brenton-Lenovo-ideapad-520S-14IKB:~/c$ gcc test.c -o test
test.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
main() {
^~~~
brenton@brenton-Lenovo-ideapad-520S-14IKB:~/c$ ./test
Hello world!
brenton@brenton-Lenovo-ideapad-520S-14IKB:~/c$
```

Figure 6: The complete command line sequence performed in these steps.