

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 basic computer literacy is required. Advanced PC gaming skills such as:

- Editing .cfg or .ini files
- Configuring command line arguments (eg: via a shortcut or Steam)
- Debugging complicated driver problems
- Using a game engine's debug console (typically accessed via the tilde ~ key)
- Use of keyboard shortcuts
- Hosting LAN parties and knowing how to configure the network

are also excellent preparation for your programming adventure. If you lack this experience then hopefully today's lab gets you up to speed.

The ENGG1003 labs involve sitting at a PC with 20 to 40 other students and working through lab exercises. Most weeks the lab exercises are there for you to learn, others there will be an assessment task which must be marked by a lab demonstrator before you leave.

The lab demonstrator's job is to answer questions you have about the material and help you work through unexpected problems. There will be limits to how much help they can provide during assessment tasks (default assumption is zero) but during non-assessed labs you can ask them anything you want.

Their job is not *necessarily* to directly answer your question but to provide information which best supports your learning. This means that, instead of giving you an answer, they may guide you through the process of using Google/reference documentation/etc to help you become an independent learner.

While you are going through this course it is a near certainty that you will encounter difficult problems which take hours (or even days) to solve. This is normal. Engineering is *hard* and watching other students solve problems faster than you is demoralising. Don't hesitate to ask for help. Talk to people and ~~copy from them~~¹ solve problems together. Don't suffer in silence.

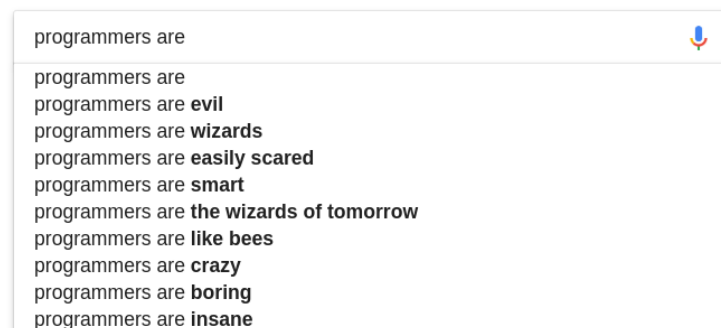


Figure 1: This image isn't really relevant, I just thought it was funny. Yes, it is real.

¹Seriously, don't do this.

Contents

1	Introduction	1
2	Software Installation	3
3	C Programming Basics	6
3.1	Introduction to Code::Blocks	7
3.2	Introduction to OnlineGDB - Optional	14
4	Compiler Errors and Warnings	16
4.1	Missing a semicolon	16
4.2	Missing a Quote Symbol	17
4.3	Failing to Define <code>main()</code> 's Return Type	18
5	Comments	19
5.1	Intrinsic Documentation	20
6	Basic Arithmetic in C	21
6.1	Operator Precedence Basics	22
7	A Very Brief Introduction to Git	24

2 Software Installation

This section of ENGG1003 introduces you to the “C” programming language. Before getting started you should check that the software below is installed on the PC you are using. Links are provided for installation on a personal computer. If you are using a university PC use the Software Center, documented below.

Software List:

- Code::Blocks - Download “codeblocks-17.12mingw-setup.exe” from <http://www.codeblocks.org/downloads/26>
- Git for Windows - Download “Git-2.25.1-32-bit.exe” from <https://github.com/git-for-windows/git/releases/tag/v2.25.1.windows.1>

To find both packages on a lab PC:

- Code::Blocks - Search “codeblocks” in the Start menu and look for the result shown in Figure ??.

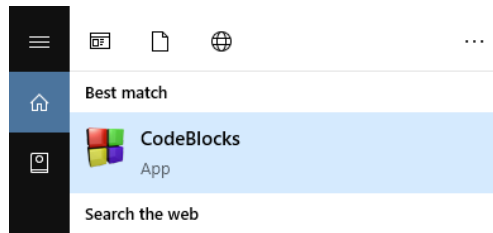


Figure 2: The Start menu search result for Code::Blocks.

- Git for Windows - There is no Start menu entry for this software, it is only accessed by right clicking in a location in Windows Explorer.

Test for its installation by opening Windows Explorer (shortcut is windows button + E), navigating to U:\, and right click somewhere. The menu which appears should list “Git GUI Here” and “Git Bash Here” as shown in Figure 3.

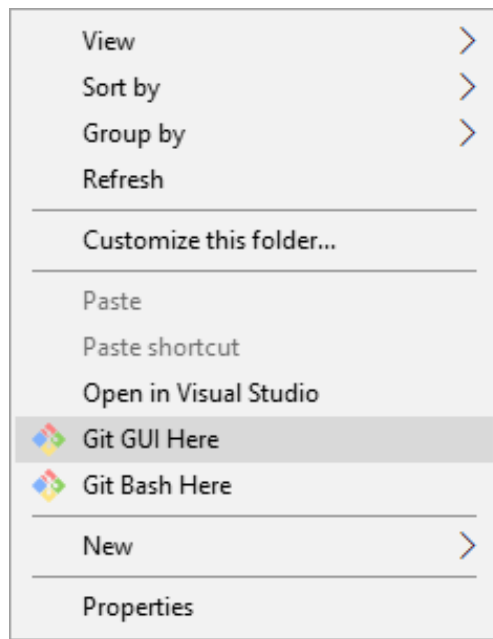


Figure 3: Git for Windows is listed when right click'ing in (or on) a folder when browsing Windows Explorer.

In case software is not installed all university PCs contain a “Software Center” app which allows students to install software on university PCs from a university managed package repository.

To install software:

1. Use the Start menu to search for the Software Center as shown in Figure 4.

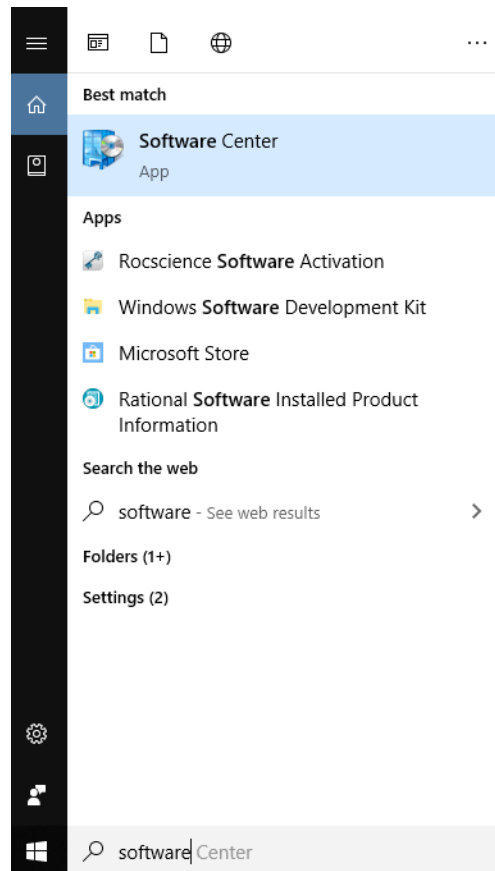


Figure 4: Use the Start menu to find the Software Center app.

2. After the Software Center opens, search for the package you need (In this case, either “codeblocks” or “git for windows”) using the search bar in the top right as shown in Figure 5.

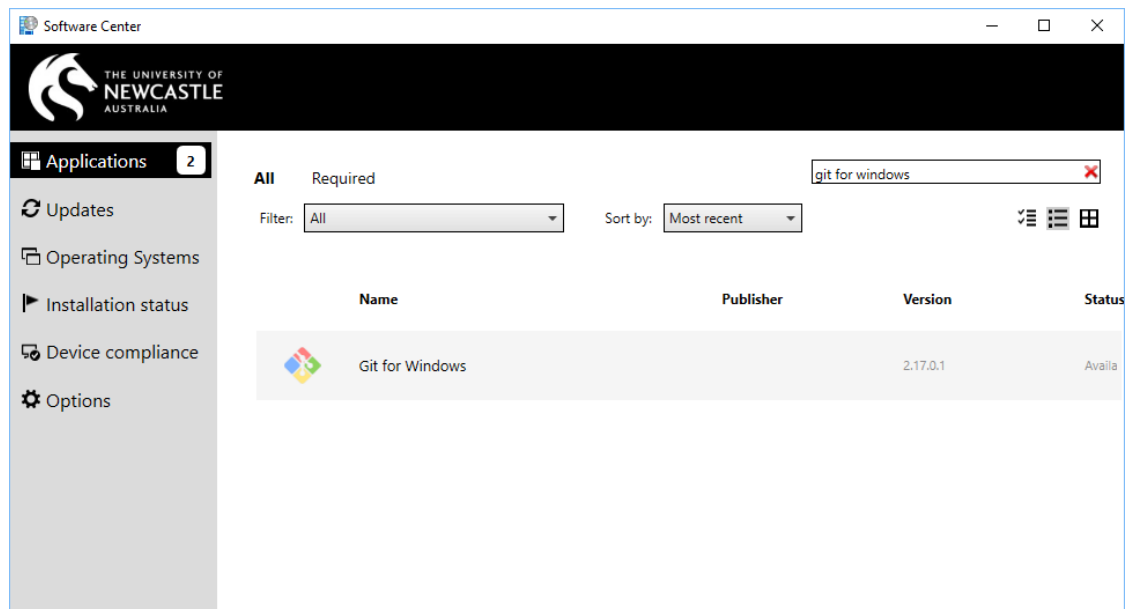


Figure 5: Searching for Git for Windows using the Software Center. Note that it is listed as “available”; this implies that it is not installed.

3. Clicking a search result brings up some package information and an install button as shown in Figure 6.
6. Click the install button.

[Applications](#) > [Application details](#)



Git for Windows

Install

Status: Available

Version: 2.17.0.1

Date published: Not specified

Restart required: Might be required

Download size: 38 MB

Estimated time: 5 minutes

Total components: 1

Date Modified: 8/01/2019

Figure 6: Click “Install” and wait a minute for the package to be installed.

3 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.

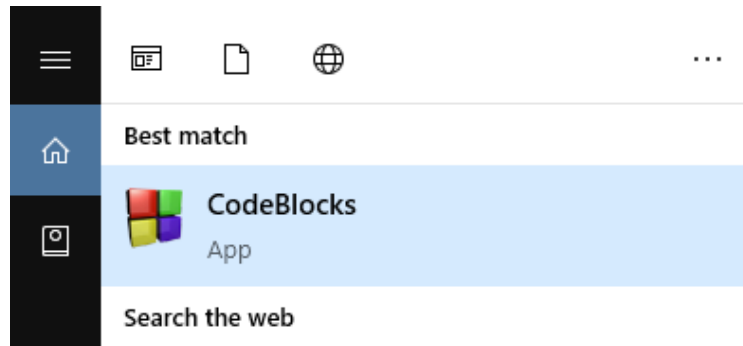
For the time being the “compiler” noun will be used to colloquially reference a highly complex set of software tools which turn your source code into an executable binary file. You will be shielded from the details until otherwise necessary.

3.1 Introduction to Code::Blocks

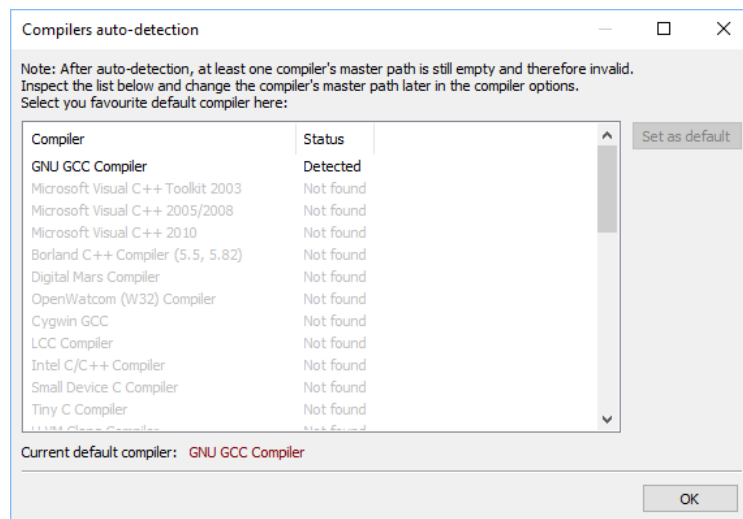
Code::Blocks is a relatively simple *integrated development environment* (IDE). It contains a coding-oriented editor, build system which runs a pre-configured compiler, and debug interface. It is the recommended development environment for ENGG1003.

To open Code::Blocks and create a new “hello world” template project perform the following steps:

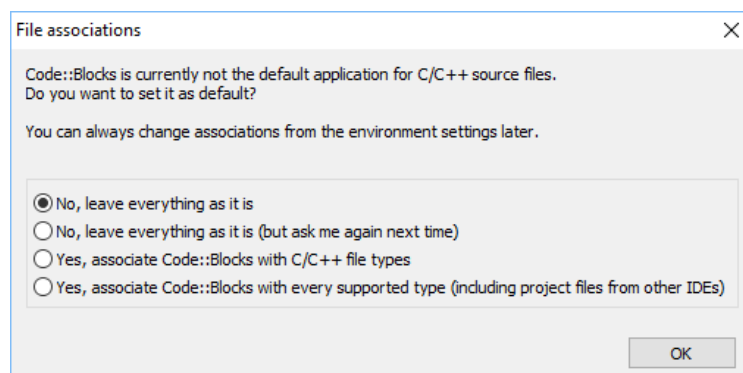
1. Find Code::Blocks in the Start menu and click it:



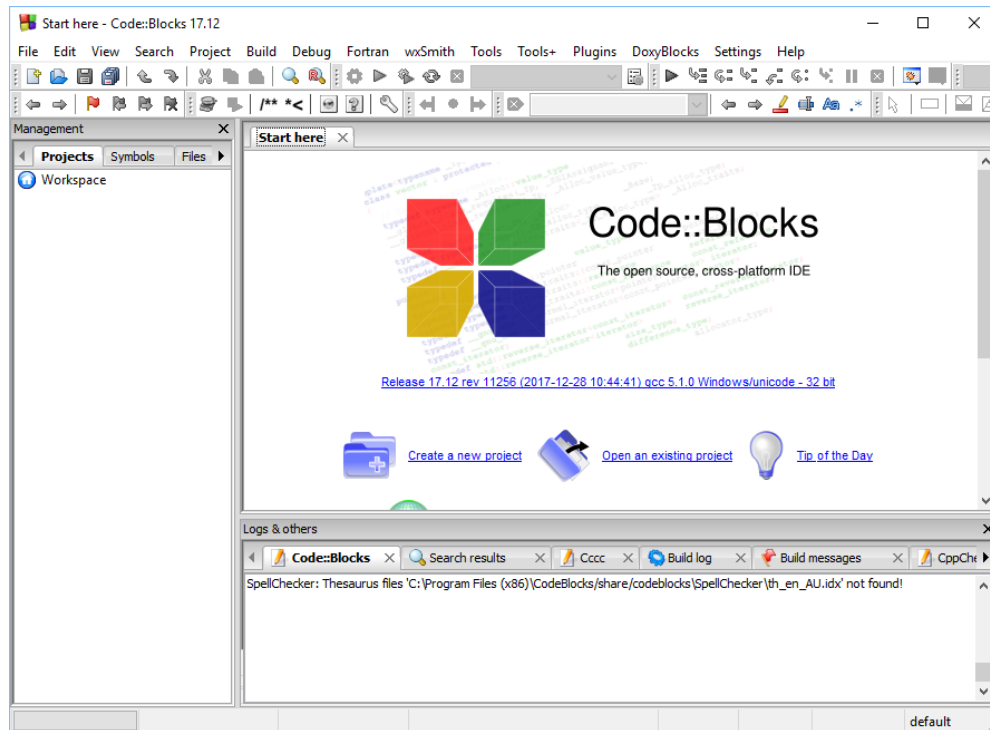
2. Code::Blocks searches for installed compilers on startup. Make sure GNU GCC Compiler is detected and click OK.



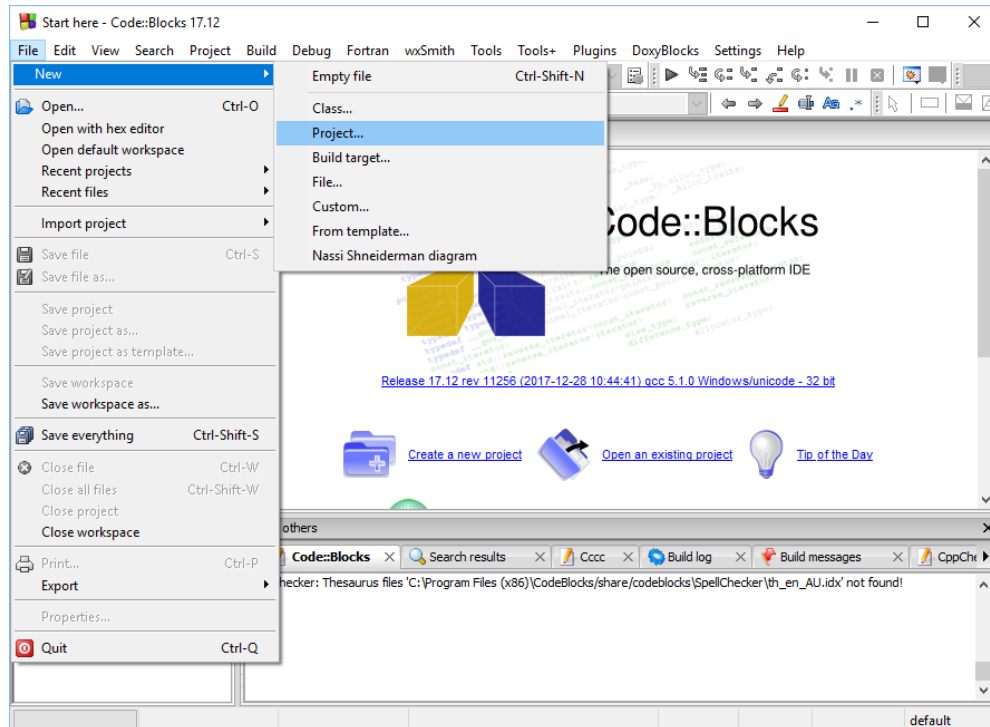
3. If this comes up make your own selection and click OK. I'd recommend the 3rd option so that clicking a .c file opens it in Code::Blocks.



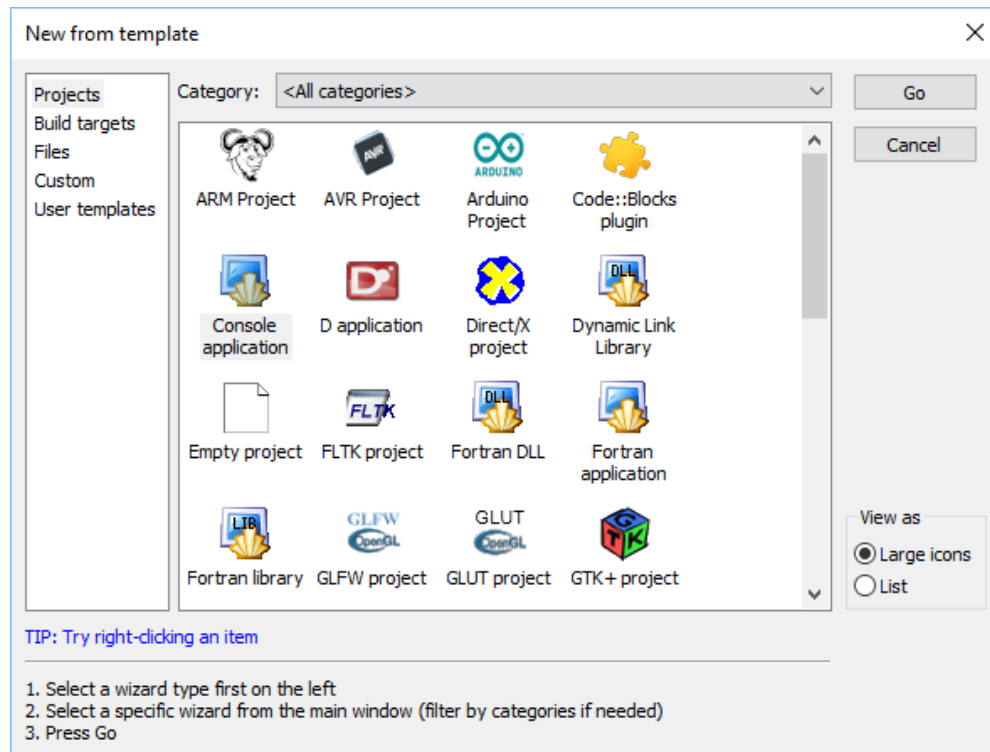
4. After closing the previous dialogue box we are finally greeted by the main screen. It currently doesn't have a project open so there is no code to edit and no files listed in the "Workspace" (left side).



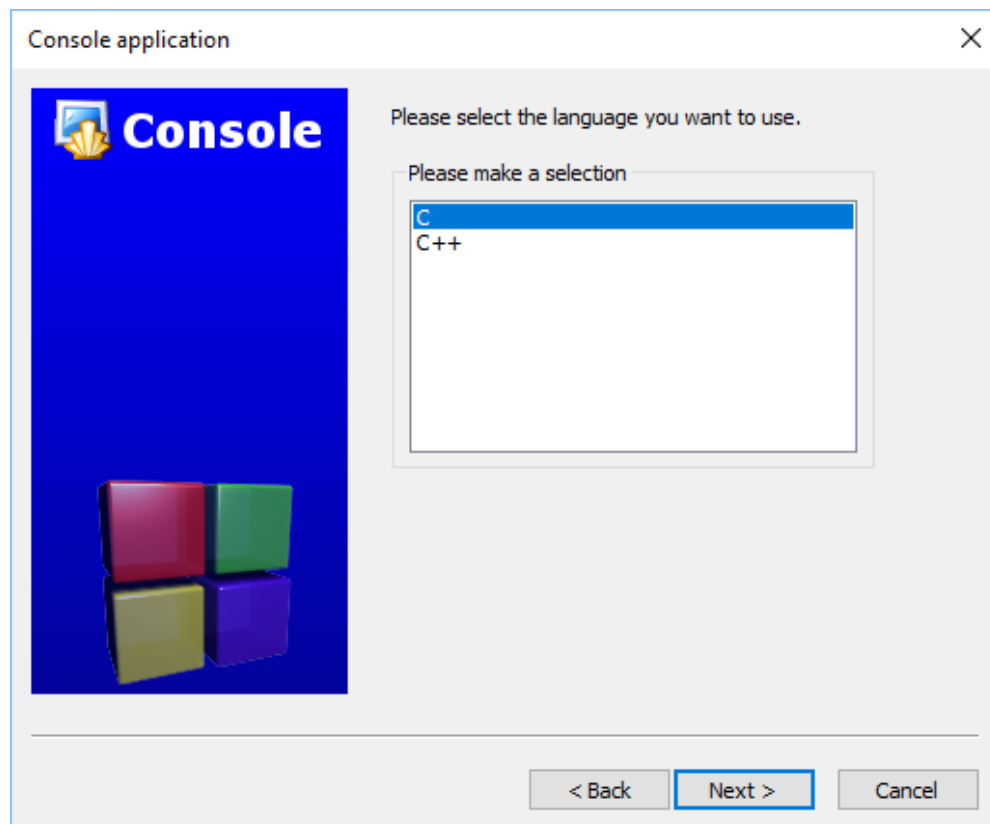
5. Create a new project by clicking File \rightarrow New \rightarrow Project...



6. Select “Console application” (middle left) and click “Go” (upper right).

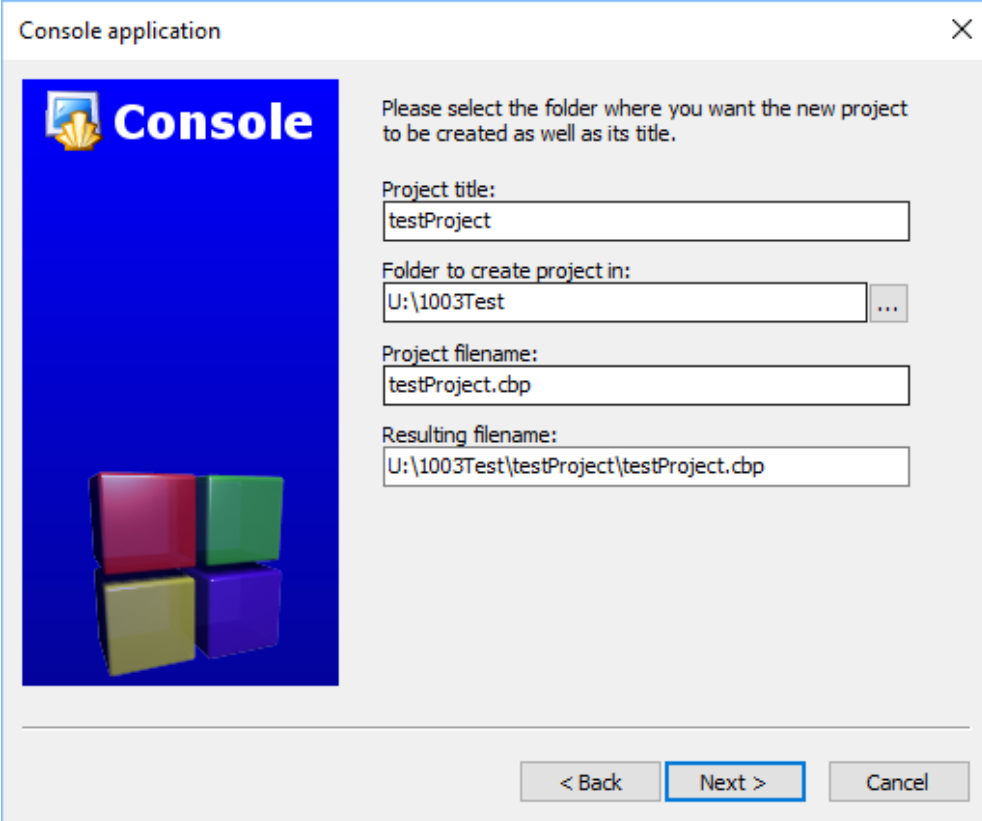


7. Select “C” in the middle list and click “Next”



8. Give your project a title and select a folder to store it by clicking the “...” button to the right of the “Folder to create project in” text box. I recommend storing it on U:\ as this is a network share mounted on any university PC when you log in. Later, we will use the version control software “git” and host our project files on the cloud service “GitHub”.

Click “Next” when you are done.



Console application

Please select the folder where you want the new project to be created as well as its title.

Project title:
testProject

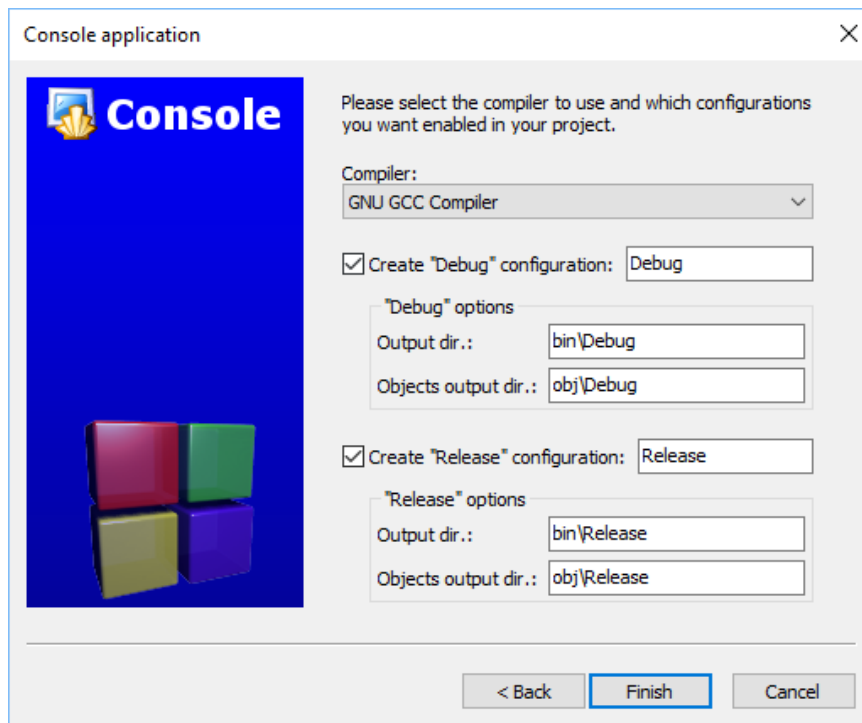
Folder to create project in:
U:\1003Test ...

Project filename:
testProject.cbp

Resulting filename:
U:\1003Test\testProject\testProject.cbp

< Back Next > Cancel

9. Don't change anything on this screen. Just click “Finish”.



10. The project is now created and you should be seeing a screen similar to Figure 7. The code listing in the main editor window is a classic “hello world” program. For unknown historical cultural reasons printing “Hello World!” to the screen is the first thing everybody must learn when studying programming.
11. Compile and run the project by clicking the “green play button and gear” icon, highlighted by the professionally drawn blob of a red arrow in Figure 7. This performs a “build and run” process. In this context “build” is just another word for “compile”; the process of turning program code into a *binary* that the computer can execute.

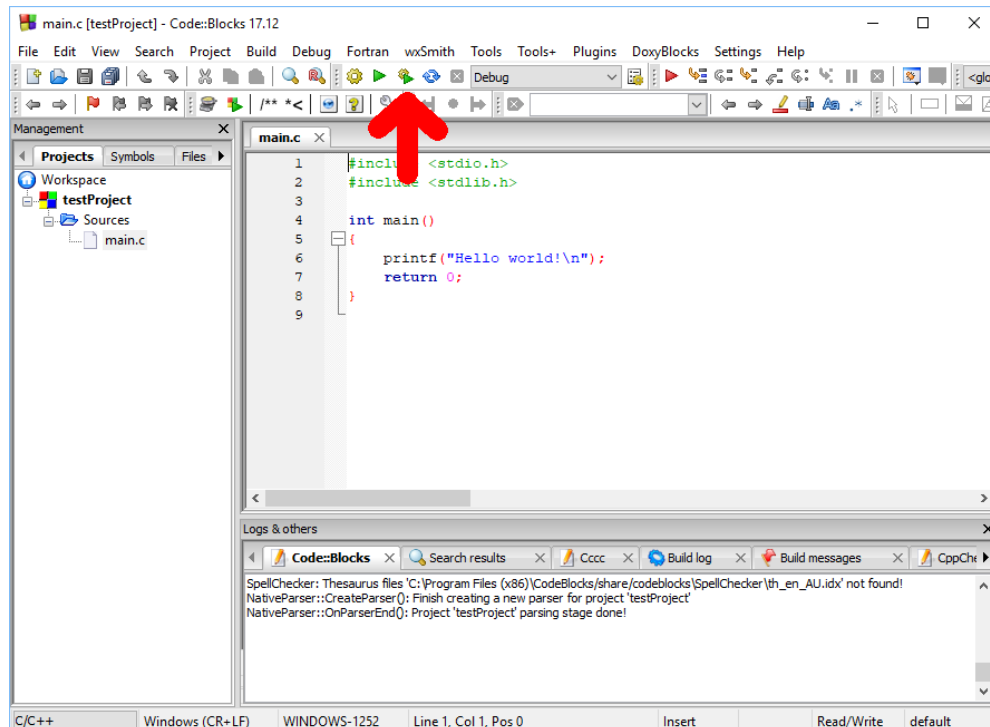


Figure 7: The “Hello World” template project created by Code::Blocks.

12. If there are no errors (if there are ask a demonstrator for help) the Windows command prompt window shown in Figure 8 will appear. This is your program running in a console, the “Hello world!” text is visible at the top.

The “Process returned 0...” text relates to the `return 0;` at the bottom of the code listing. You don’t need to worry about what this is now, we will study “return values” in later weeks. Suffice to say that if you see this everything worked.

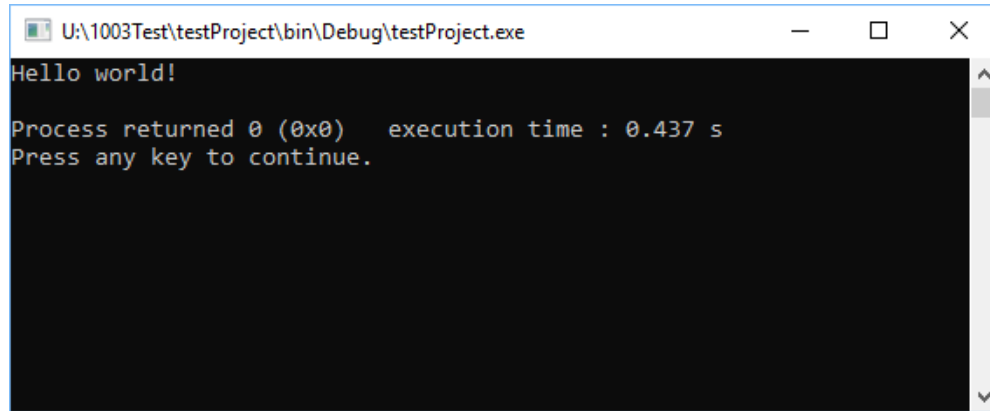


Figure 8

Task 1

1. Running the default template demonstrates *standard output*. Modify the code to match that in Listing 1.

NB: You can use either tabs or spaces for indenting. Tabs is fewer keystrokes, spaces are more well defined. Some editors change the width of a tab and this causes illogical anger in some programmers.

While making the changes you will observe Code::Blocks' auto-complete features. When, for example, you type a double quote " character it *automatically* types two and places the cursor between them. The editor will also automatically indent new lines and 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).

Also observe that when you write a *function* name (eg: `scanf()`) you will be greeted with a pop-up which shows `int scanf(const char*, ...)`. This is known as a *function prototype* and is there to help you remember how a function is used. We will study this in more detail when functions are covered later in the semester.

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 #include <stdlib.h>
3
4 int main()
5 {
6     int k;
7     scanf("%d", &k);
8     printf("You entered: %d\n", k);
9     return 0;
10 }
```

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

2. After editing the code build and run it again. 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.
4. After pressing enter the console should display the text "You entered: 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?

3.2 Introduction to OnlineGDB - Optional

NB: OnlineGDB is not necessary but you may find it useful if you are forced to use a computer which lacks a C compiler.

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 reasons² 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 2

1. Open a web browser and navigate to <http://www.onlinegdb.com>.
2. Configure OnlineGDB to run C code by selecting "C" from the "Language" drop-down box in the upper-right.

This website supports many languages^a so feel free to come back here later if you're interested 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.

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

^aMATLAB is not one of them because it is a *very* expensive commercial package.

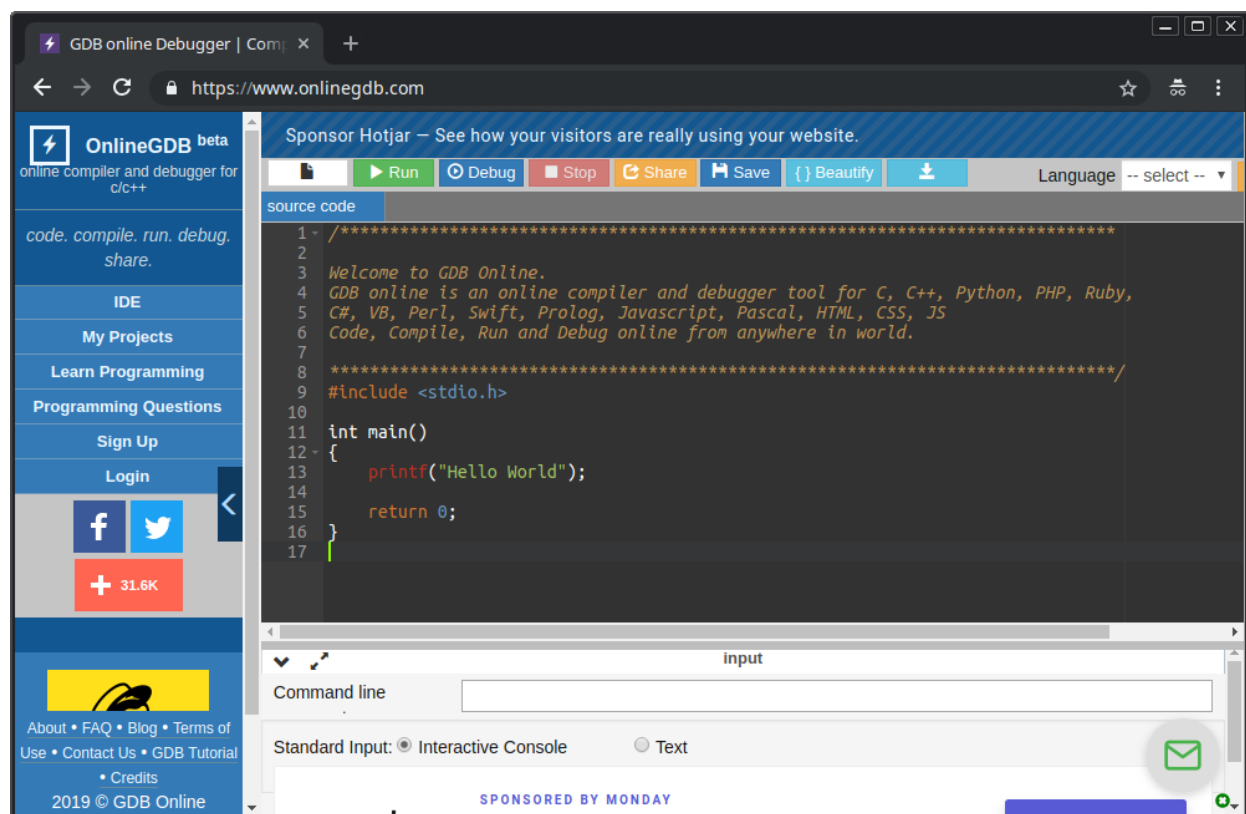


Figure 9: The view when OnlineGDB is first opened.

After OnlineGDB has loaded you will be greeted with the screen seen in Figure 9. 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

²It only allows one debug session per IP address. The entire campus uses the same public IP so we can't use it in labs.

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 3

1. Observe that there is already a template C listing in the editor.
2. 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 10.

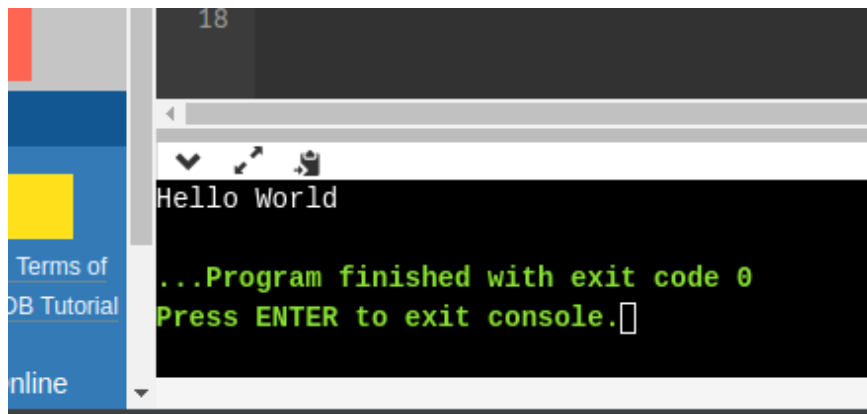


Figure 10: A cropped screenshot showing the "Hello World" program output.

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.

4 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()`?³

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".

4.1 Missing a semicolon

Task 4

1. If you have not done so already, get the code shown in Listing 1 working correctly.
2. Remove the semicolon (the ; character) from the end of line 6.
3. Compile the code, what error did the compiler produce? Which line is the error is on?

The compiler errors in Code::Blocks will look something like Listing 2, but formatted better.

```
||=== Build: Debug in testProject (compiler: GNU GCC Compiler) ===|
U:\1003Test\testProject\main.c||In function 'main':|
U:\1003Test\testProject\main.c|7|error: expected '=', ',', ';', 'asm' or
    '__attribute__' before 'scanf'|
U:\1003Test\testProject\main.c|7|error: 'k' undeclared (first use in this function)|
U:\1003Test\testProject\main.c|7|note: each undeclared identifier is reported only
    once for each function it appears in|
||=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===||
```

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

The first error (error: expected '=', ',', ';', 'asm'... etc.) is telling you that something was missing and the compiler noticed the omission at line 7. In this case we removed a ; so that's what is missing. Note, however, that while the error was reported to be on line 7 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 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.

³The 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.

4.2 Missing a Quote Symbol

Task 5

1. Fix the previous error by re-inserting the semicolon.
2. 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 blue⁴ in the Code::Blocks editor, 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:

```
| |== Build: Debug in testProject (compiler: GNU GCC Compiler) ==|
U:\1003Test\testProject\main.c|In function 'main':|
U:\1003Test\testProject\main.c|8|warning: missing terminating " character|
U:\1003Test\testProject\main.c|8|error: missing terminating " character|
U:\1003Test\testProject\main.c|9|error: expected expression before 'return'|
U:\1003Test\testProject\main.c|10|error: expected ';' before '}' token|
| |== Build failed: 3 error(s), 1 warning(s) (0 minute(s), 0 second(s)) ==|
```

Listing 3: Errors produced by removing a " character.

Observe how missing the closing " generates both an error and a warning at different locations. Why does it do this? To be honest, I don't know; 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 as without the " the printf *expression* was not completed. In C an expression can be thought of as "a syntactically 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*⁵.

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); C can be incredibly unforgiving!

⁴Yeah, I know that these notes use different syntax highlighting colours to Code::Blocks. Sorry, fixing is on my TODO list for 2021.

⁵<https://xkcd.com/859/>

4.3 Failing to Define `main()`'s Return Type

Enough about errors, the final example in this section demonstrates a compiler *warning*.

Task 6

1. Fix the previous error by re-inserting the appropriate quote symbol.
2. Remove `int` before `main` Listing 4

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

Listing 4: Example code which generates a warning.

Compiling this code generates the following compiler warning:

```
warning: return type defaults to int [-Wimplicit-int]
```

Listing 5: Compiler output produced when `main()` return type is omitted.

Observe the warning printed to both the “Build log” and “Build messages” tabs towards the bottom of the Code::Blocks window.

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 Overflow⁶ has written a good explanation about it.

⁶Why are you even taking this course? Stack Overflow is all you *really* need, right?

5 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 does⁷.

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 files⁸ 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 7

Take whatever source code is currently shown in Code::Blocks 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.

⁷Despite 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.

⁸Did 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*.

5.1 Intrinsic Documentation

As a supplement to comments, *intrinsic documentation* is the idea behind choosing informative names for variables and functions⁹. 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 try to document "nothing").

⁹No, you don't need to know what a function is yet. It is in week 5 or so.

6 Basic Arithmetic in C

So, it is Page 21 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*¹⁰. This may feel obvious, but C includes several *unary* 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)`)¹¹.

We begin with a basic example:

Task 8

Modify your code to match that of Listing 6.

```

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 6: 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.

¹⁰An operand is one of the "things" that a mathematical operator operates on. Eg: In `a + b` the variables `a` and `b` are operands.

¹¹I contemplated leaving this closing parenthesis out to give you unresolved tension but I'm not quite *that* evil.

6.1 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 operator precedence here: 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 9

Using the template in Listing 7 implement the following equation in C:

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

Remember that `*` is the multiplication operator in C and that a `;` is needed at the end of the line.

```

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 7: A basic arithmetic example

You will notice a few new things about Listing 7. Firstly, it uses the `float` datatype. This type will store any real number with a magnitude of 1.2×10^{-38} to 3.4×10^{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 method¹² 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.

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.

¹²The other major standard is known as *fixed point*. Details are beyond ENGG1003 but you’ll probably see it if you study enough digital signal processing (DSP).

Task 10

Implement the following equations on line 8. What happens when you choose x to force a division by zero?

1. $y = 9x + 32$

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. $y = \frac{x+2}{x-1}$

7 A Very Brief Introduction to Git

NB: Git is not in the course outline so it isn't strictly required nor assessed. It is, however, incredibly useful and included here because several 3rd and 4th year course coordinators (and industry contacts) asked me to expose you to it. I agreed.

Programming projects typically involve two potentially problematic issues:

1. Multiple people contribute to the same project. How should their changes be “merged” into some kind of “master” source listing?
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 of a set of data.

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. I am using Git for tracking changes while writing course content and the Git host GitHub is also doubling as cloud storage as I move writing between three different computers.

The Git package can be installed on any computer, it doesn't even require a server / client architecture spread across two or more computers. GitHub is simply a website which hosts Git servers.

Task 11

Navigate to <https://github.com/bschulznewy/engg1003/blob/master/Lectures/2019/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 second Week 1 Lecture from **last year** (this year's notes are up there somewhere but they don't have an interesting history like the 2019 notes do).

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 11. It lists the git *commits* made which affect 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.

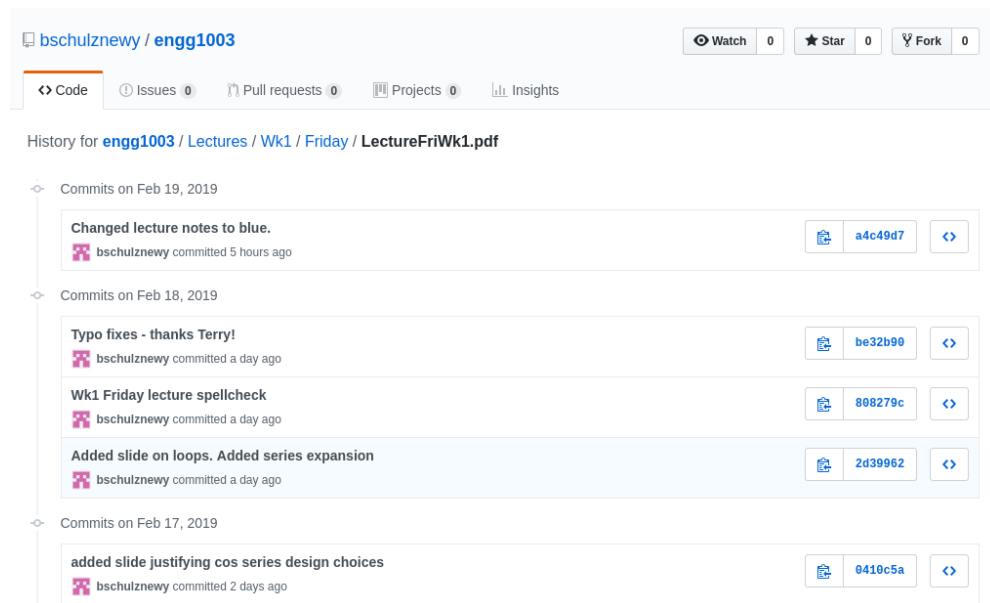


Figure 11: The git repo history for the Friday lecture notes as it was last year - It will look very different now!.

If you click on a commit message you will see what changed in this particular git commit. An example is shown in Figure 12.

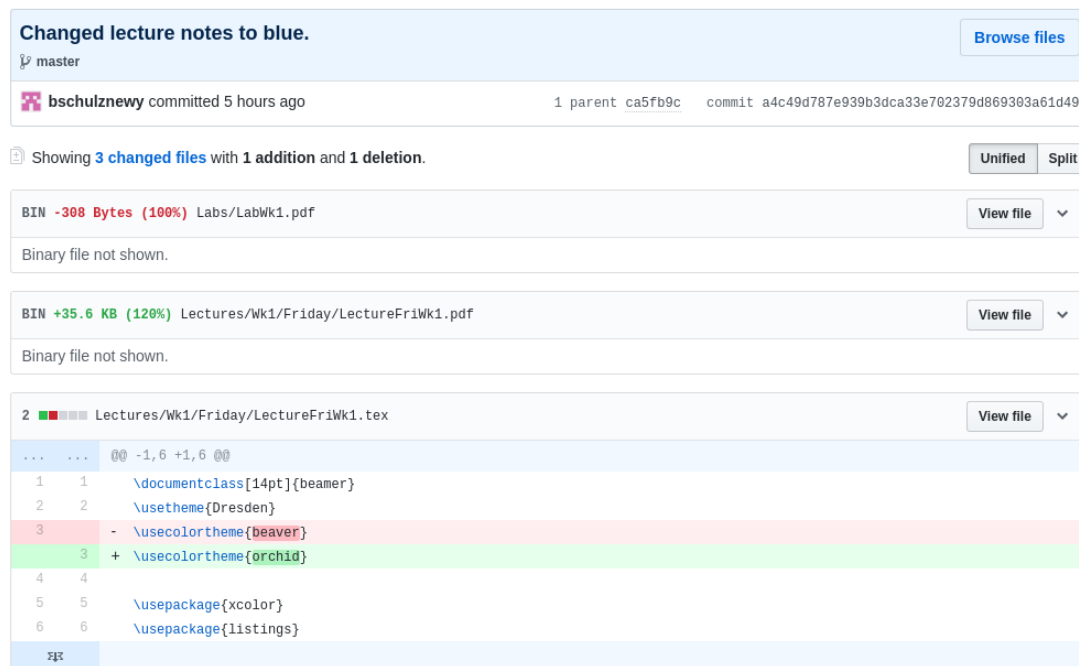


Figure 12: The changes which occurred in a particular commit.

The box at the bottom of Figure 12 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 recent commits [here](#) and browse a few commits.

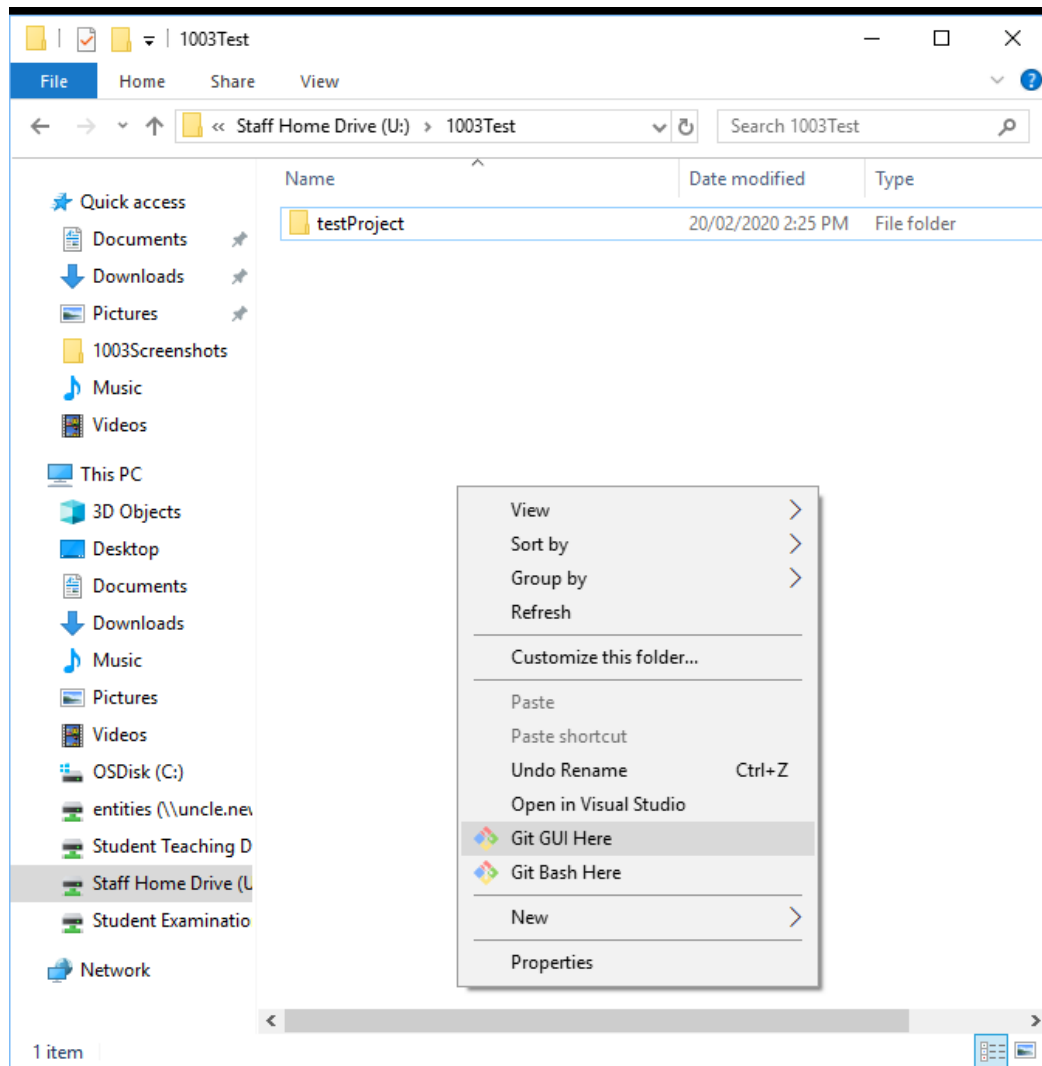
Task 12

Go to www.github.com and create a GitHub account. You can use it to store all your code from this course (and other courses!). This allows advanced version tracking and cloud storage from the same service.

Task 13

Use your new GitHub account to create a new Git repository and upload today's work to it using Git for Windows.

1. Open Windows Explorer (windows button + E) and navigate to the folder *above* where your lab 1 code is stored.
2. Right click in this folder and select "Git GUI here".



3. The dialog box shown in Figure 13 appears. Click “Create New Repository”.

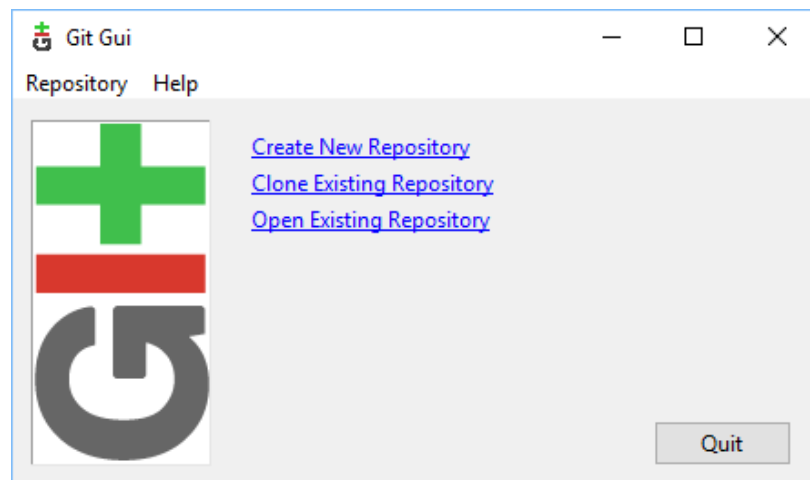
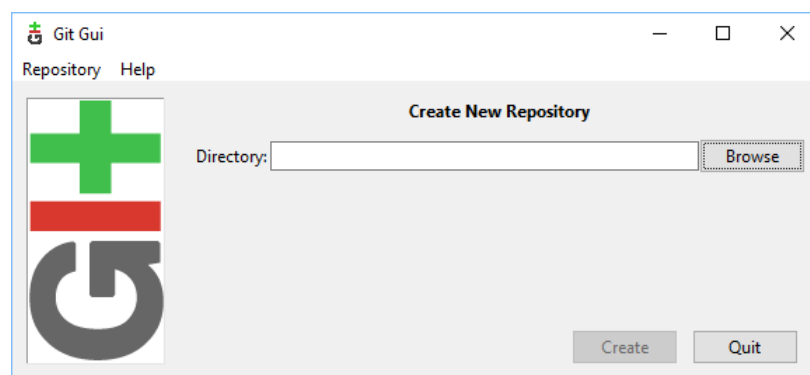
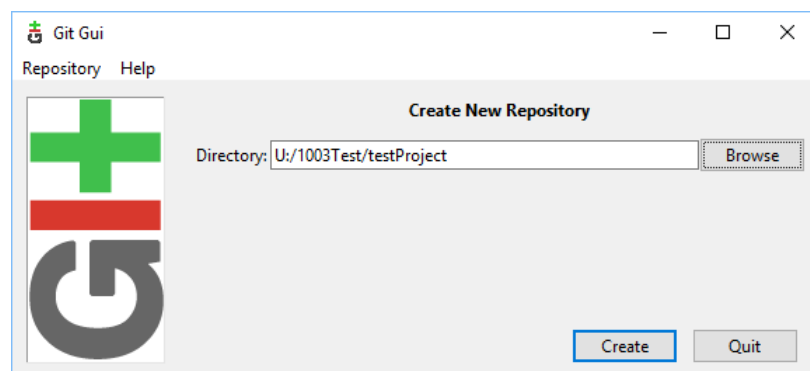


Figure 13

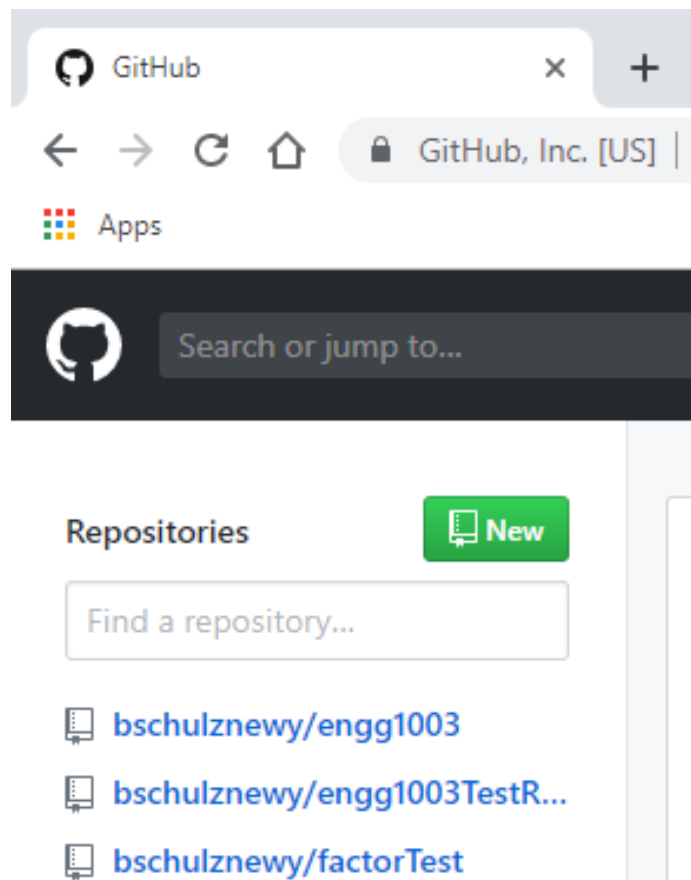
4. You are then greeted with this dialogue box:



5. Click “Browse” and select the folder which hold your lab 1 code:



6. Now navigate to <https://github.com>, log in if necessary, and click the green “New” button:



7. Give the new repository a name and click the green “Create repository” button at the bottom:

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner: bschulznewy / Repository name *: engg1003TestRepo ✓

Great repository names are short and memorable. Need inspiration? How about [upgraded-octo-disco](#)?

Description (optional)

☒ Public
Anyone can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: None | Add a license: None ⓘ

Create repository

8. In the “Quick setup” section ensure that the “HTTPS” button is selected and copy the URL in the text box next to it:

bschulznewy / engg1003TestRepo

Watch 0 Star 0 Fork 0

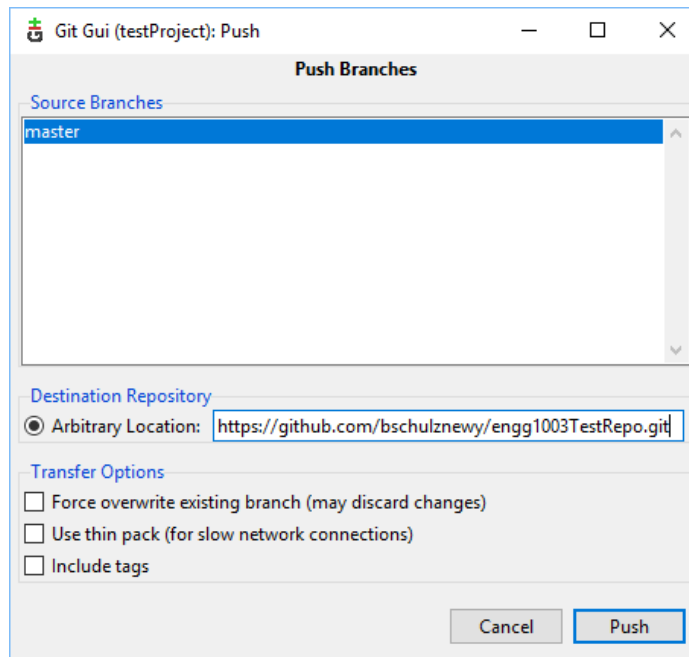
Code Issues 0 Pull requests 0 Actions Projects 0 Wiki Security Insights Settings

Quick setup — if you've done this kind of thing before

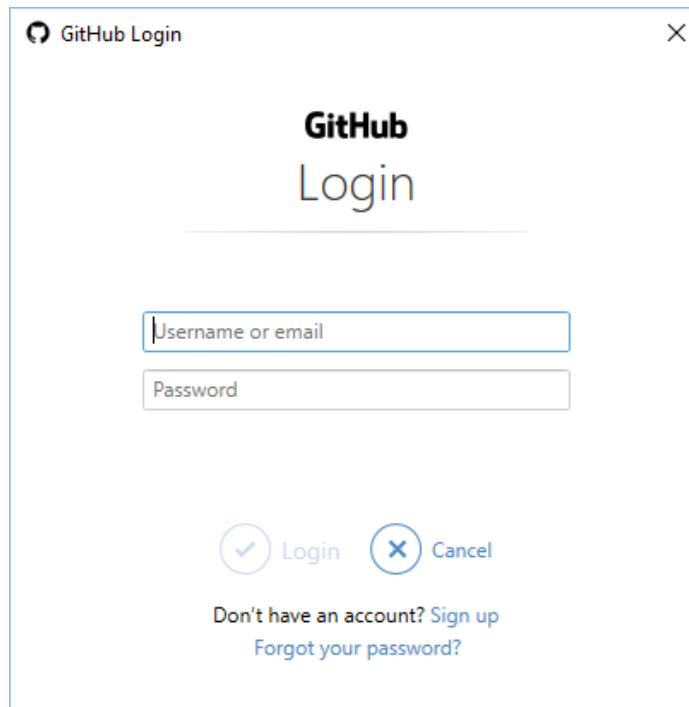
Set up in Desktop or HTTPS SSH <https://github.com/bschulznewy/engg1003TestRepo.git>

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

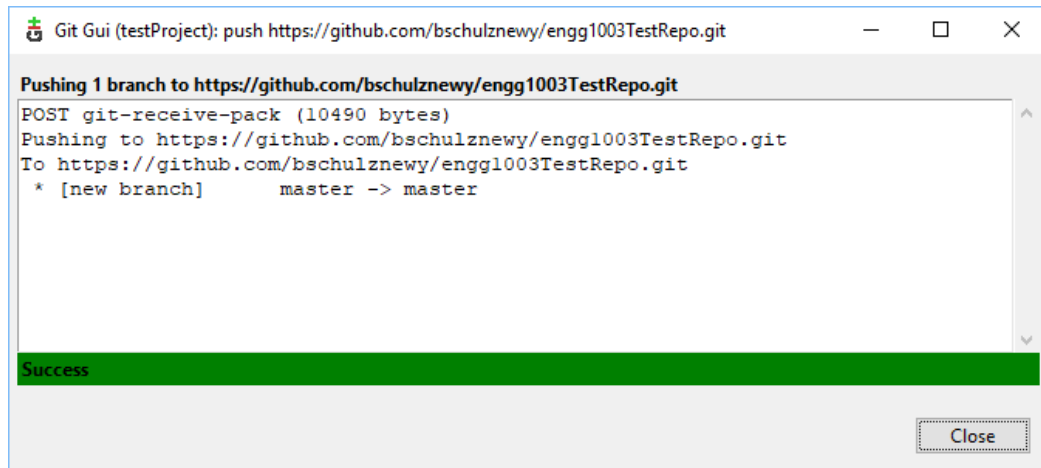
9. Back to Git for Windows, paste this URL in the text box next to “Arbitrary Location:”



10. In the dialogue box shown above, press “Push”. This attempts to make a remote (“in the cloud”) copy of you repository at the URL you supplied.
11. Since the URL is on GitHub, the following login window appears. Type in your GitHub login details and click “Login”:



12. If your login is correct the window below will appear. The text output shows the progress of the upload (in Git language the “Push” operation).

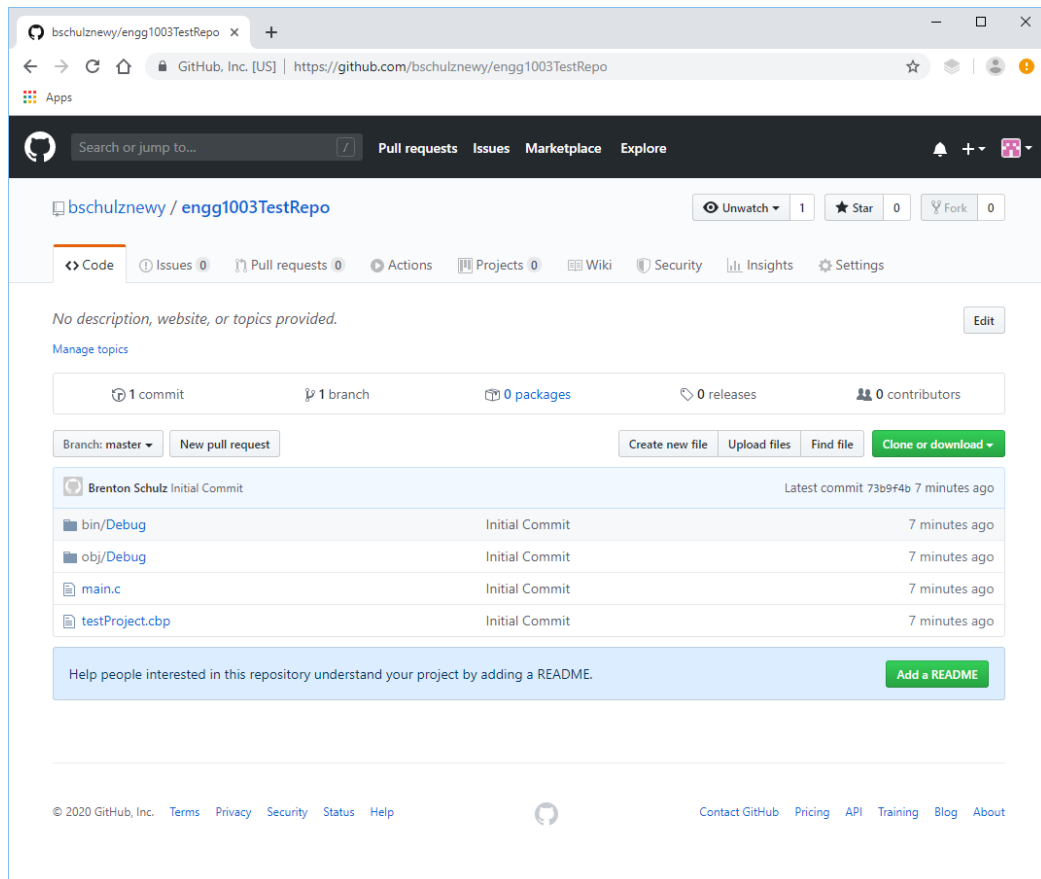


The screenshot shows a window titled "Git Gui (testProject): push https://github.com/bschulznewy/engg1003TestRepo.git". The main area displays the following text:

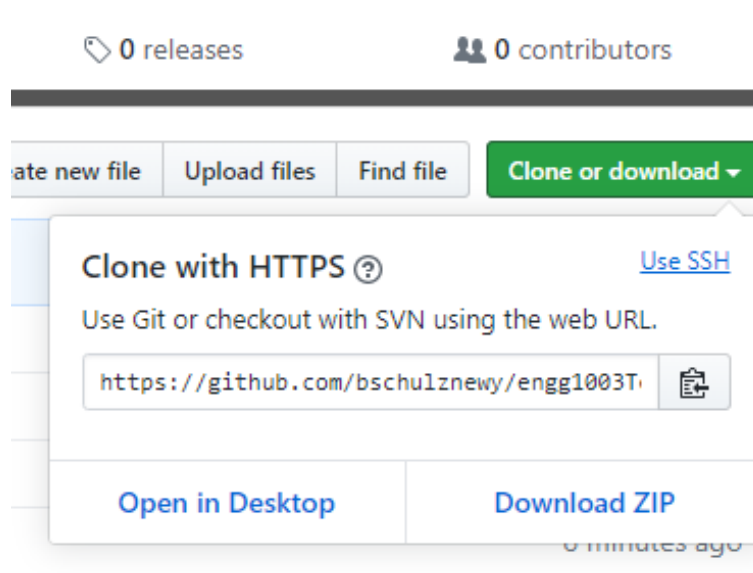
```
Pushing 1 branch to https://github.com/bschulznewy/engg1003TestRepo.git
POST git-receive-pack (10490 bytes)
Pushing to https://github.com/bschulznewy/engg1003TestRepo.git
To https://github.com/bschulznewy/engg1003TestRepo.git
 * [new branch]      master -> master
```

A green bar at the bottom indicates "Success". A "Close" button is in the bottom right corner.

13. If you navigate to the new repository on GitHub you will now see all the project files listed:



14. If you need to download (aka “clone”) this repository on another machine you will need the URL provided by clicking the green “Clone or download” button:



15. On a different computer, or in a different folder on the same computer, go back to step 1 and re-open Git Gui through the right click context menu.
16. This time select “Clone Existing Repository” and follow the prompts to create a new copy of your lab 1 code.
17. If you make changes to the code you can open the Git GUI in the repository’s top level folder and use the Git for Windows GUI to perform a “stage”, followed by a “commit” (with an appropriate message) and, lastly, a “push” to upload new changes to your GitHub hosted repository.
18. The Git repository will keep track of any changes you make to your code. At any time you can view old versions and compare what changed with each commit. Make some changes to the code in Code::Blocks editor and perform a stage/commit/push and observe the changes on GitHub and in the Git GUI.

Task 14

Read more about how Git works and what it does here: <https://developer.ibm.com/tutorials/d-learn-workings-git/>. You probably won’t understand everything, but that’s not important right now. Just get some general exposure to the ideas and we’ll fill in the practicalities later.