**BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI**

***CS-F211: Data Structures and Algorithms***

**Reference Sheet: Debugging with VSCode**

# Introduction

Welcome to this reference sheet on using the Visual Studio Code Debugger. Debugging is something that can't be avoided. Every programmer will at one point in their programming career have to debug a section of code. There are many ways to go about debugging, from printing out messages to the screen, using a debugger, or just thinking about what the program is doing and making an educated guess as to what the problem is.

Before a bug can be fixed, the source of the bug must be located. For example, with segmentation faults, it is useful to know on which line of code the seg fault is occurring. Once the line of code in question has been found, it is useful to know about the values in that method, who called the method, and why (specifically) the error is occurring. Using a debugger makes finding all of this information very simple.

We know that the errors we face may either be compile-time errors or run-time errors. Compile time errors are errors in the way the code was written or your different code files were links. They are usually issues with the syntax. We shall briefly look at compile-time errors and then move on to using the debugger to resolve errors that arise when the program runs - ie runtime errors.

# 

# Compile-Time Errors

First, let's distinguish between the types of compile-time errors: most compilers will give three types of compile-time alerts: compiler warnings, compiler errors, and linker errors.

Although you don't want to ignore them, compiler warnings aren't something severe enough to actually keep your program from compiling. Usually, compiler warnings are an indication that something might go wrong at runtime. How can the compiler know this at all? You might be making a typical mistake that the compiler knows about. A common example is using the assignment operator ('=') instead of the equality operator ('==') inside an if statement. Your compiler may also warn you about using variables that haven't been initialized and other similar mistakes. Generally, you can set the warning level of your compiler..

Nevertheless, compiler warnings aren't going to stop you from getting your program working (unless you tell your compiler to treat warnings as errors), so they're probably a bit less frustrating than errors. Errors are conditions that prevent the compiler from completing the compilation of your files. Compiler errors are restricted to single source code files and are the result of 'syntax errors'. What this really means is that you've done something that the compiler cannot understand. For instance, the statement "for(;)" isn't correct syntax because a for loop always needs to have three parts. Although the compiler would have expected a semicolon, it would also have expected a conditional expression, so the error message you get might be something like "line 53, unexpected parenthesis ')'". Note, also, that compiler errors will always include a line number at which the error was detected.

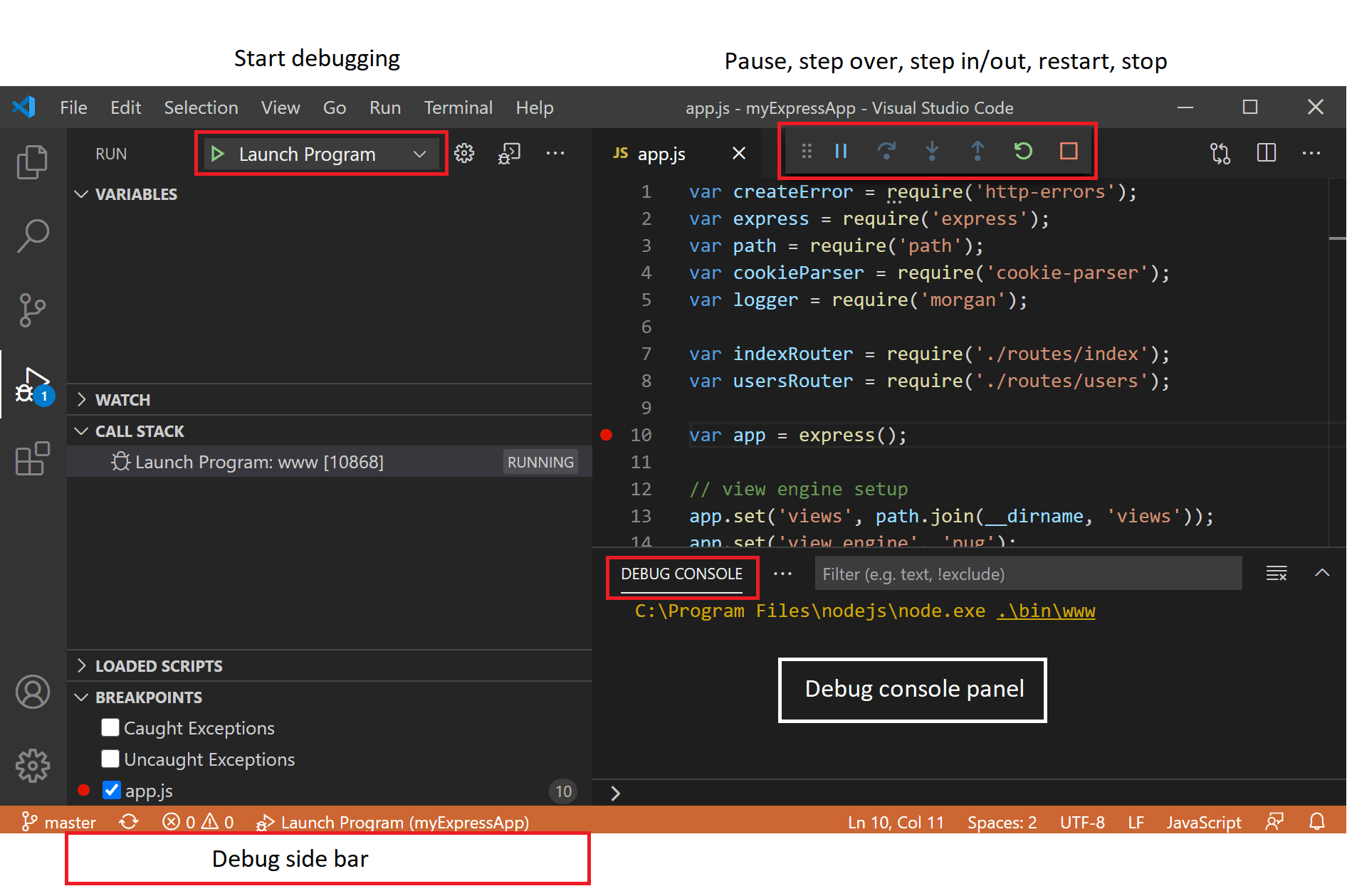
Even if you make it through the compilation process successfully, you may run into linker errors. Linker errors, unlike compiler errors, have nothing to do with incorrect syntax. Instead, linker errors are usually problems with finding the definitions for functions, structs, classes, or global variables that were declared, but never actually defined, in a source code file. Generally, these errors will be of the form "could not find definition for X".

Usually, the compilation process will begin with a series of compiler errors and warnings and, once you've fixed all of them, you'll then be faced with any linker errors.

You can learn more about handling compile time errors here — <https://www.cprogramming.com/tutorial/compiler_linker_errors.html>. In this sheet, we shall focus on using Visual Studio Code's debugger for handling run-time errors.

# Debugging Run-Time Errors with VSC

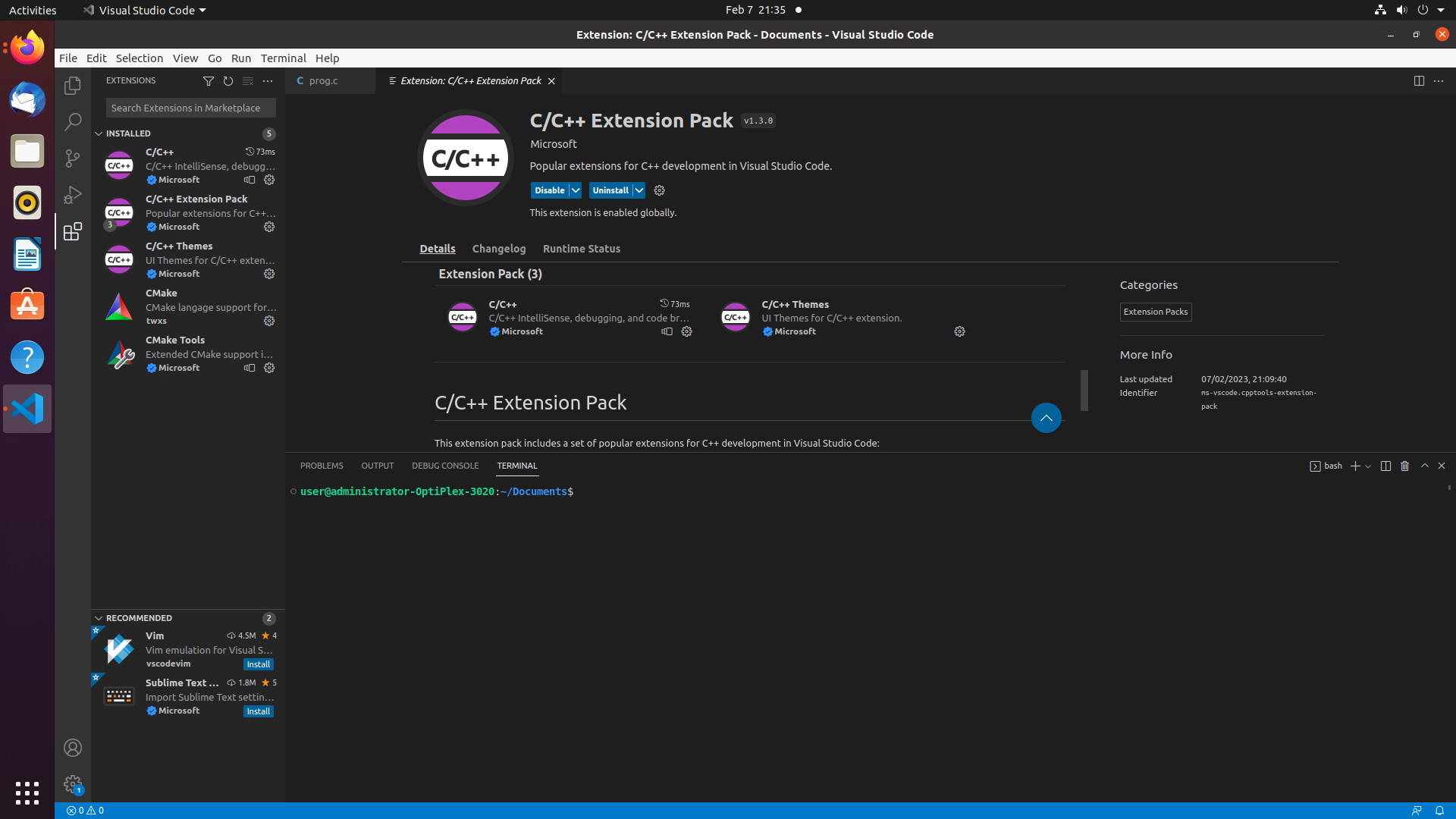
One of the key features of Visual Studio Code is its great debugging support. VS Code's built-in debugger helps accelerate your edit, compile, and debug loop.



## Debugger extensions

For debugging different languages and runtimes, different extensions are available in the VSC marketplace...

For C, the C/C++ Extension Pack by Microsoft includes a debugger which is the one we would be using.



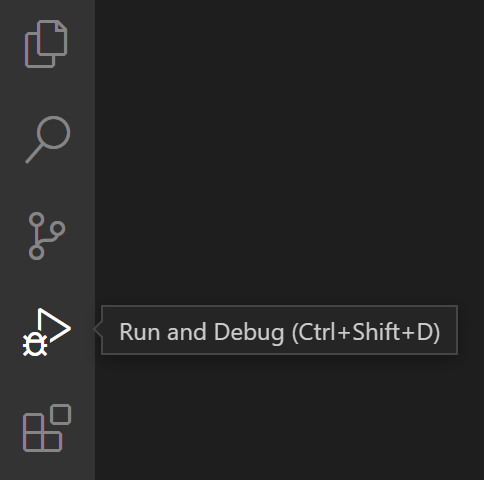
## Start debugging

It is helpful to first create a sample C application before reading about debugging.

You can use the *prog1.c* program provided to you for this purpose.

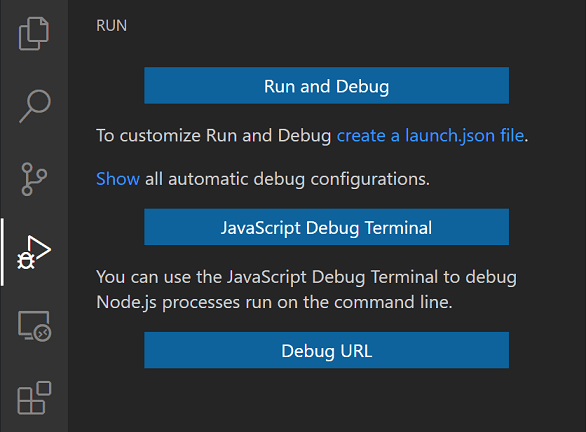
## Run and Debug view

To bring up the **Run and Debug** view, select the **Run and Debug** icon in the **Activity Bar** on the side of VS Code. You can also use a keyboard shortcut. (Ctrl+Shift+D by default. May vary from system to system.)



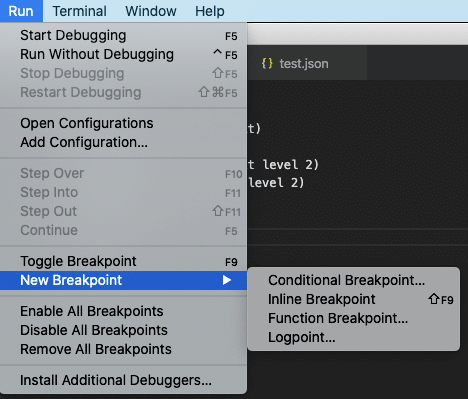
The **Run and Debug** view displays all information related to running and debugging and has a top bar with debugging commands and configuration settings.

If running and debugging is not yet configured (no launch.json has been created), VS Code shows the Run start view.



## Run menu

The top-level **Run** menu has the most common run and debug commands:



## Debug actions

Once a debug session starts, the **Debug toolbar** will appear on the top of the editor.



| **Action** | **Explanation** |
| --- | --- |
| Continue / Pause  F5 | Continue: Resume normal program/script execution (up to the next breakpoint).  Pause: Inspect code executing at the current line and debug line-by-line. |
| Step Over  F10 | Execute the next method as a single command without inspecting or following its component steps. |
| Step Into  F11 | Enter the next method to follow its execution line-by-line. |
| Step Out  Shift+F11 | When inside a method or subroutine, return to the earlier execution context by completing remaining lines of the current method as though it were a single command. |
| Restart  Ctrl+Shift+F5 | Terminate the current program execution and start debugging again using the current run configuration. |
| Stop  Shift+F5 | Terminate the current program execution. |

### Run mode

In addition to debugging a program, VS Code supports **running** the program. The **Debug: Run (Start Without Debugging)** action is triggered and it uses the currently selected launch configuration. Many of the launch configuration attributes are supported in 'Run' mode. Pressing the **Stop** button terminates the program.

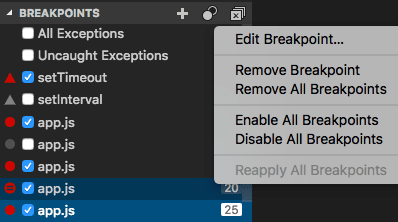
## Breakpoints

Breakpoints can be toggled by clicking on the **editor margin** on the current line. Finer breakpoint control (enable/disable/reapply) can be done in the **Run and Debug** view's **BREAKPOINTS** section.

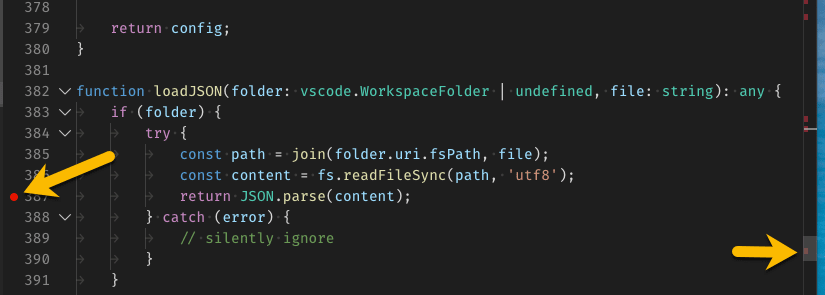
* Breakpoints in the editor margin are normally shown as red-filled circles.
* Disabled breakpoints have a filled grey circle.
* When a debugging session starts, breakpoints that cannot be registered with the debugger change to a grey hollow circle. The same might happen if the source is edited while a debug session without live-edit support is running.

If the debugger supports breaking on different kinds of errors or exceptions, those will also be available in the **BREAKPOINTS** view.

The **Reapply All Breakpoints** command sets all breakpoints again to their original location.

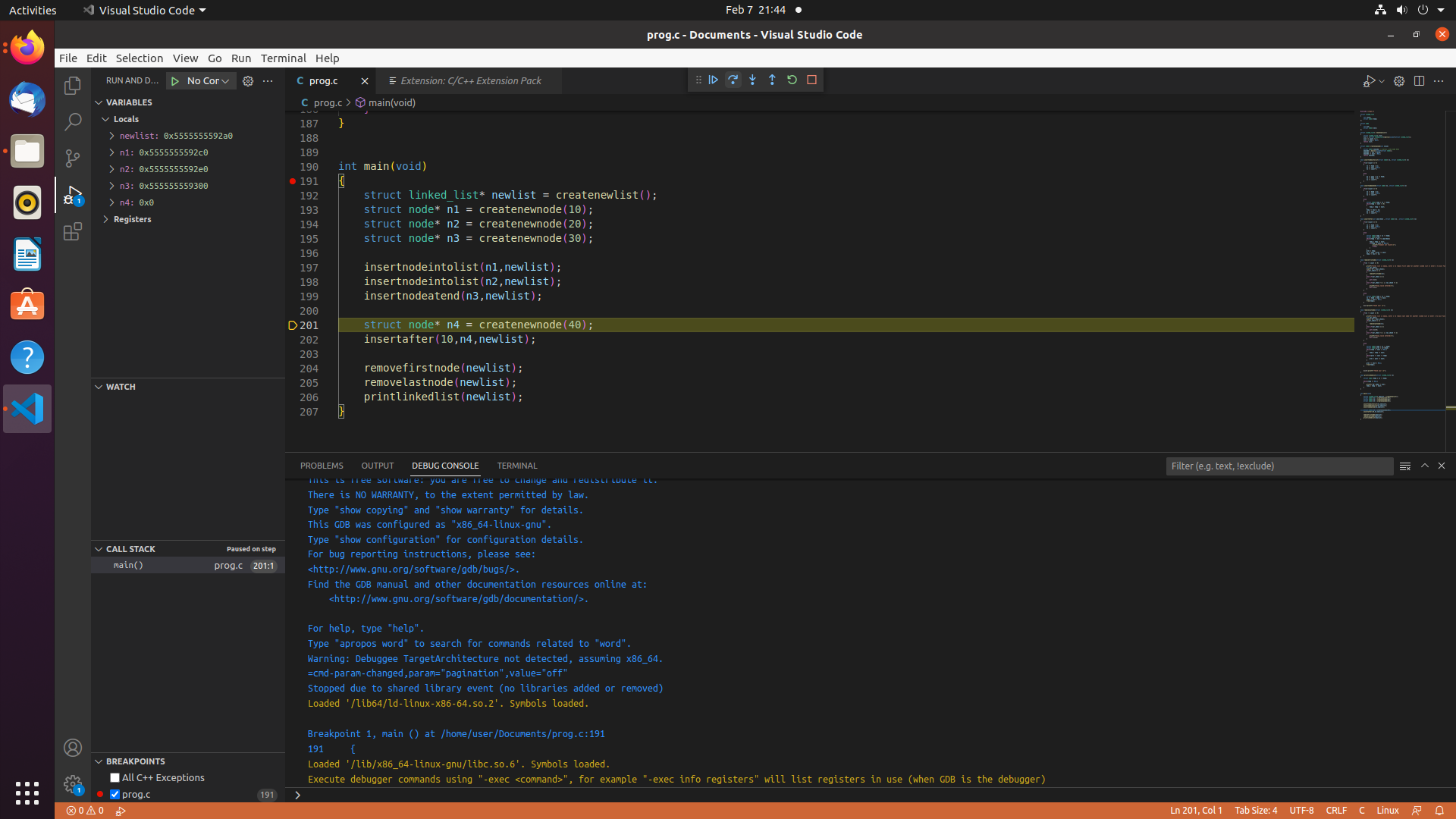


Optionally, breakpoints can be shown in the editor's overview ruler by enabling the setting debug.showBreakpointsInOverviewRuler:



## Data inspection

Variables can be inspected in the **VARIABLES** section of the **Run and Debug** view or by hovering over their source in the editor. Variable values and expression evaluation are relative to the selected stack frame in the **CALL STACK** section.



Variable values can be modified with the **Set Value** action from the variable's context menu. Additionally, you can use the **Copy Value** action to copy the variable's value, or **Copy as Expression** action to copy an expression to access the variable.

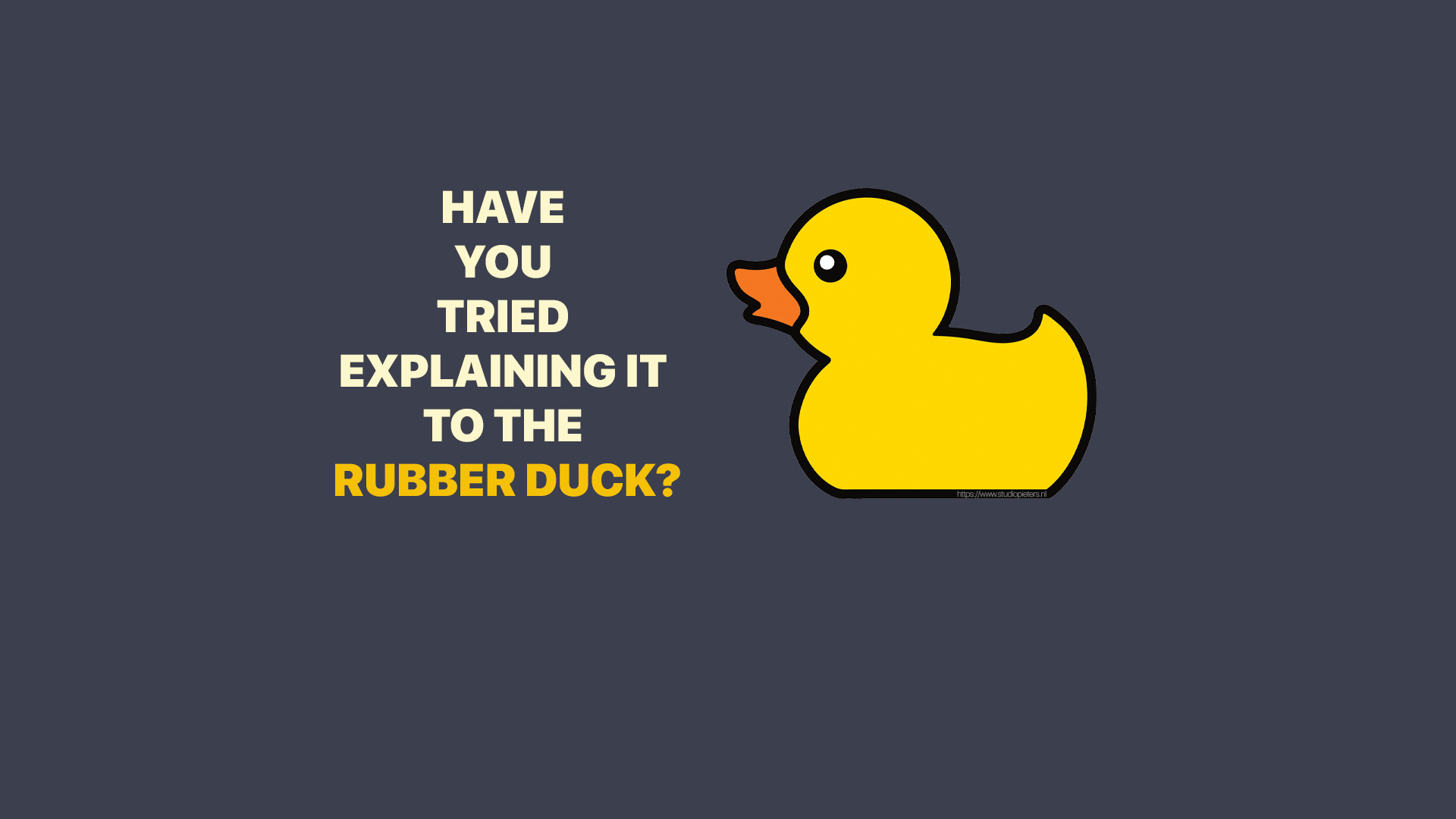
For more details refer:

<https://code.visualstudio.com/docs/editor/debugging>

<https://code.visualstudio.com/docs/cpp/cpp-debug>

# Rubber Duck Debugging

In software engineering, rubber duck debugging (or “rubberducking”) is a method of debugging code by simply articulating a problem in spoken or written natural language. Many other terms exist for this technique, often involving different (usually) inanimate objects, or pets such as a dog or a cat. Teddy bears are also widely used.



The rubber duck debugging method is as follows:

1. Beg, borrow, steal, buy, fabricate, or otherwise obtain a rubber duck (bathtub variety).
2. Place the rubber duck on the desk and inform it you are just going to go over some code with it if that’s all right.
3. Explain to the duck what your code is supposed to do, and then go into detail and explain your code line by line.
4. At some point you will tell the duck what you are doing next and then realise that that is not in fact what you are actually doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

Note: In a pinch, a coworker (or your roommate) might be able to substitute for the duck, however, it is often preferred to confide mistakes to the duck instead of your coworker.  
*Original Credit: ~Andy from lists.ethernal.org*

For more information, you may refer to <https://rubberduckdebugging.com/>.

# Try it out yourself!

Through the video tutorial shared, various techniques for debugging programs, majorly the use of the VSC debugger, have been illustrated. The source files for those programs have been made available viz. *prog1.c, prog2.c* and *prog3.c.* You are encouraged to try out the debugging process step by step as described in the video yourself for these files.

In addition to this, three practice files have also been provided viz. *practiceproblem1.c, practiceproblem2.c* and *practiceproblem3.c*. These files have buggy code written in them. Utilize the methods explained in the tutorial video and debug these files.