



**Yrkes  
Akademin**  
Vi hjälper dig att lyckas!

# C Programming

GDB, Unity & TDD

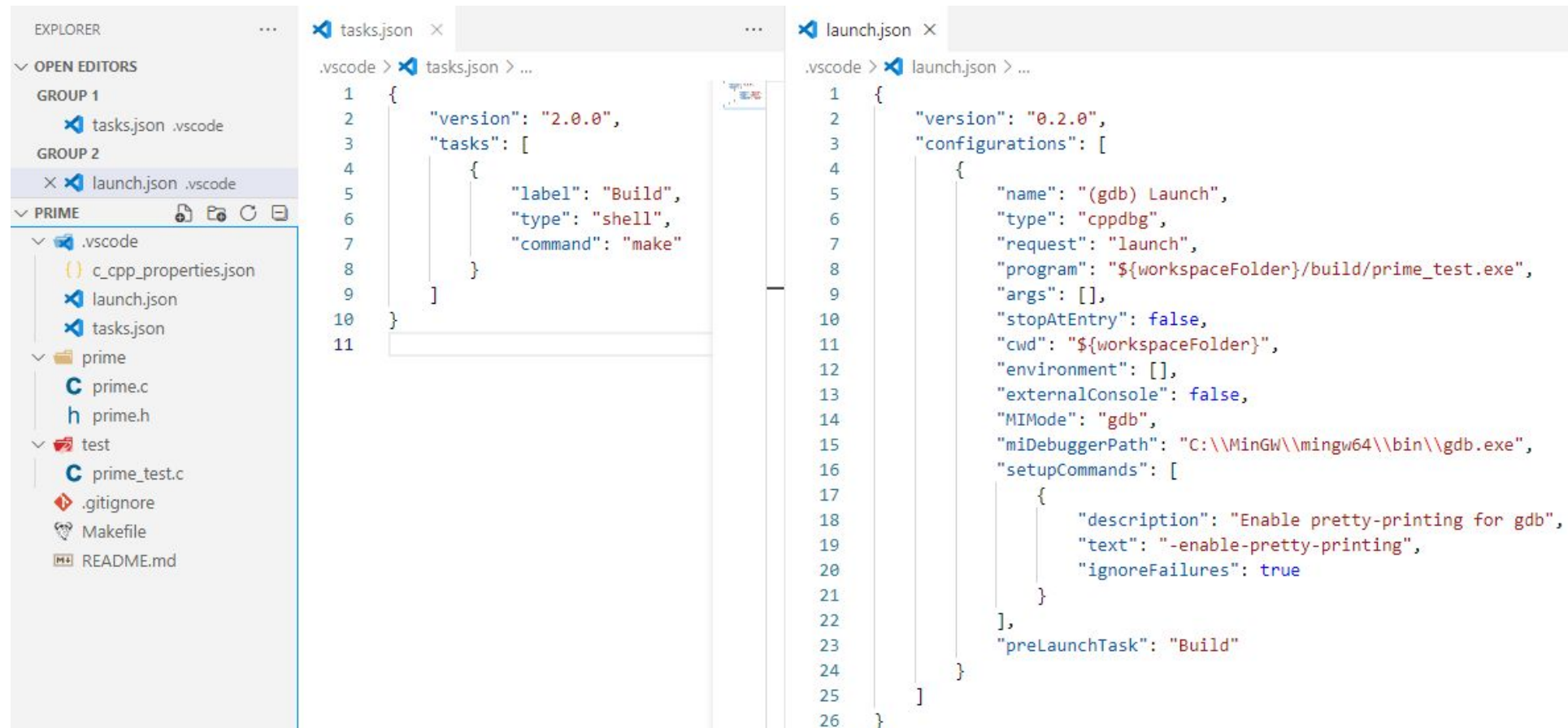
# GDB

---

- ❖ Debugging is the process of finding and resolving problems within a program using a debugger.
- ❖ **GDB** is the [GNU Project debugger](#) which can be used to debug programs written in C language.
- ❖ To compile your program for debugging you need to use **-g**. E.g. `gcc -g main.c -o main`
- ❖ You can use **gdb** to debug your program in [command line](#) or in Visual studio code.
- ❖ To use gdb & gcc to build, debug and run in visual studio code
  - Click on the run icon and then click on the **Run and Debug** button
  - Select C++(GDB/LLDB) in the opened list. And then Select **Default Configuration**
  - A json file (**launch.json**) is opened. Then set the path to your program: **"program": "path/to/your/program"**
  - Then set the path of the debugger(gdb): **"miDebuggerPath": "C:\\MinGW\\mingw64\\bin\\gdb.exe"**
  - To make a task to compile & run your program click on **Terminal > Configure Tasks..** in the main menu.
  - Then click on **Create tasks.json file from template** and then **Others Examples to run an arbitrary external command.**
  - The **tasks.json** file is created. Change the label from **echo** to **Build**
  - In the **command** write the shell commands to compile and run your program.
  - Then open **launch.json** and add **"preLaunchTask": "Build"** to the configurations.

# Visual Studio Code - GDB

## ❖ An example of launch.json and tasks.json



The screenshot shows the Visual Studio Code interface with three panels. The left panel is the Explorer, showing the file structure of a project named 'PRIME'. The middle panel shows the 'tasks.json' file, and the right panel shows the 'launch.json' file.

**Explorer Panel:**

- GROUP 1
  - tasks.json .vscode
- GROUP 2
  - launch.json .vscode
- PRIME
  - .vscode
    - c\_cpp\_properties.json
    - launch.json
    - tasks.json
  - prime
    - prime.c
    - prime.h
  - test
    - prime\_test.c
    - .gitignore
    - Makefile
    - README.md

**tasks.json:**

```
1 {
2   "version": "2.0.0",
3   "tasks": [
4     {
5       "label": "Build",
6       "type": "shell",
7       "command": "make"
8     }
9   ]
10 }
11
```

**launch.json:**

```
1 {
2   "version": "0.2.0",
3   "configurations": [
4     {
5       "name": "(gdb) Launch",
6       "type": "cppdbg",
7       "request": "launch",
8       "program": "${workspaceFolder}/build/prime_test.exe",
9       "args": [],
10      "stopAtEntry": false,
11      "cwd": "${workspaceFolder}",
12      "environment": [],
13      "externalConsole": false,
14      "MIMode": "gdb",
15      "miDebuggerPath": "C:\\MinGW\\mingw64\\bin\\gdb.exe",
16      "setupCommands": [
17        {
18          "description": "Enable pretty-printing for gdb",
19          "text": "-enable-pretty-printing",
20          "ignoreFailures": true
21        }
22      ],
23      "preLaunchTask": "Build"
24    }
25  ]
26 }
```

# Unity

---

- ❖ [Unity](#) is a portable unit testing framework written in standard C and supports testing of embedded systems. [Unity Test](#)
- ❖ Unity uses assertions to test the actual values and the expectations
- ❖ Assertions are statements of what we expect to be true about the code under test.
  - E.g. `TEST_ASSERT_EQUAL_UINT8(expected, actual);`
    - Checks if the expected value and the actual value are equal or not
    - If they are not equal, it means the test is failed
- ❖ Unity is used to test modules. Means that the code under test shall be module(s).
  - A module is a source(.c) and a header(.h) file.
  - To test a module you also need to create a test file (.c) and include unity.h
    - The test cases are implemented in this file.

# Test-Driven Development (TDD)

---

- ❖ Test-Driven Development is a technique for building software incrementally
  - Small steps help form better components
- ❖ A failing test is followed immediately by code satisfying the test
  - Healthy code growth, components are less prone of bugs.
- ❖ The focus is on the requirements instead of coding
  - Tests shall be based on the requirements
  - Tests shall enforce a specific behaviour
  - Tests shall encourage refactoring and make components easier to understand
    - Refactoring is the process of restructuring code without changing its external behavior.
- ❖ Test automation is key to TDD
  - Testing is automatically performed by machine over and over. No need for manual testing.

# Test-Driven Development (TDD)

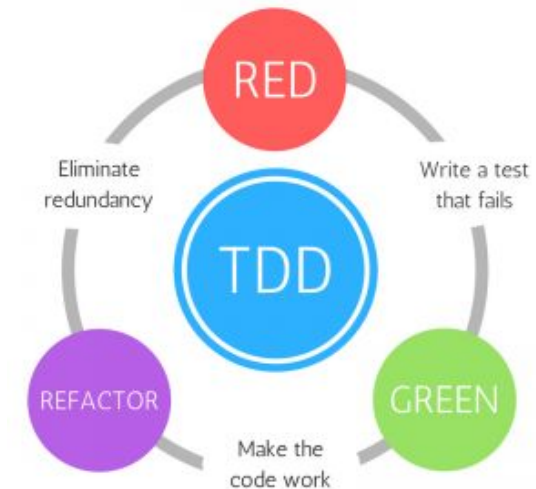
---

## ❖ Three rules of TDD

- Write production code only to pass a failing unit test.
- Write no more of a unit test than sufficient to fail.
  - Compilation failures are failures
- Write no more production code than necessary to pass the one failing unit test.

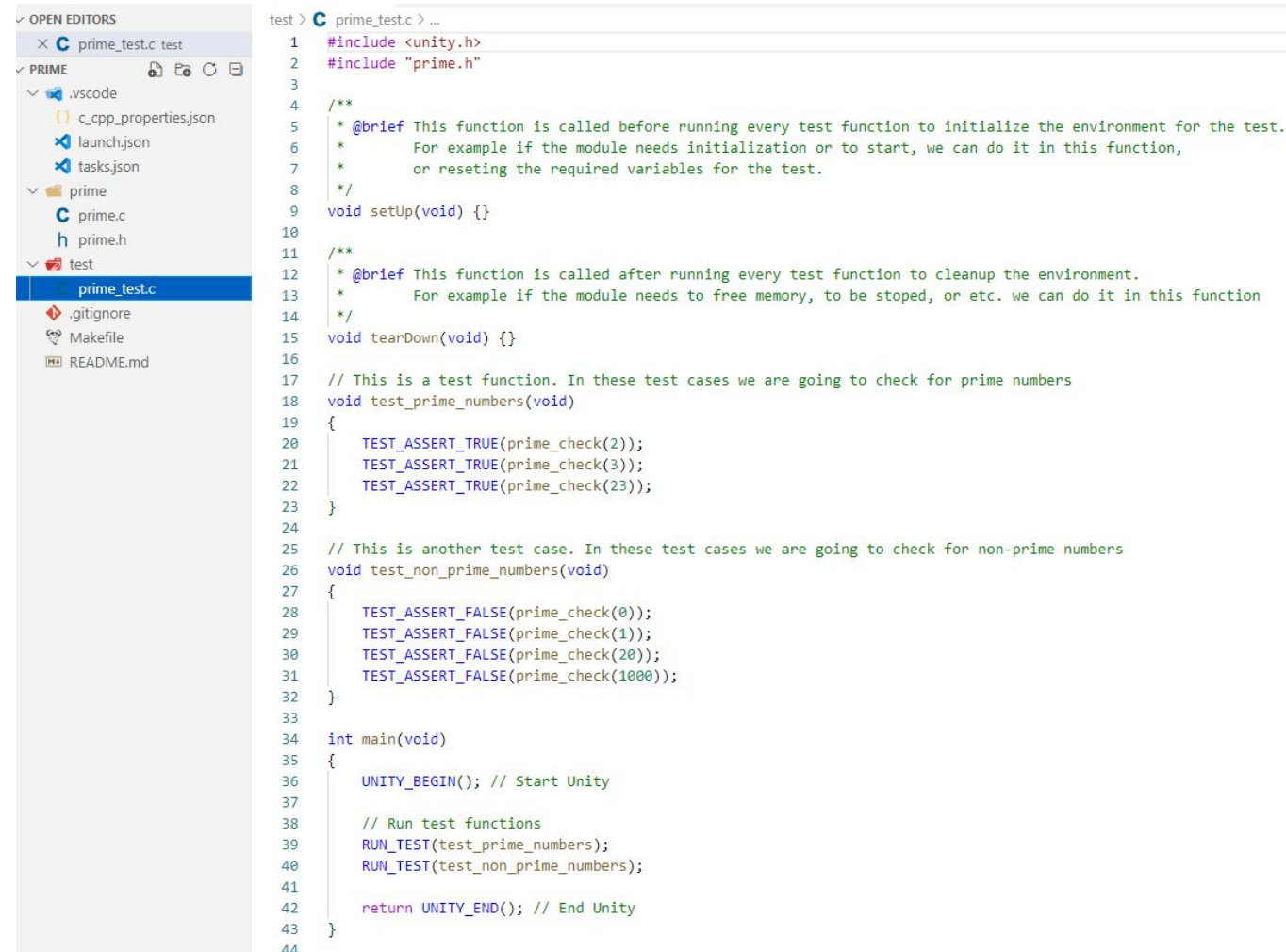
## ❖ The steps of the TDD cycle

- Add a small test
- Run all the tests and see the new one fails
- Make the small changes needed to pass the failing test
- Run all the tests and see the new one is passed
- Refactor to remove duplication and improve expressiveness





# Unity - Example



The image shows a VS Code editor interface. On the left, the 'EXPLORER' sidebar displays a project structure for 'PRIME'. The 'test' folder is expanded, showing 'prime\_test.c' selected. The main editor area displays the contents of 'prime\_test.c', which is a C file using the Unity testing framework. The code includes headers for 'unity.h' and 'prime.h', and defines two test functions: 'test\_prime\_numbers' and 'test\_non\_prime\_numbers'. The 'main' function initializes Unity, runs the tests, and ends the session.

```
test > C prime_test.c > ...
1  #include <unity.h>
2  #include "prime.h"
3
4  /**
5   * @brief This function is called before running every test function to initialize the environment for the test.
6   *        For example if the module needs initialization or to start, we can do it in this function,
7   *        or resetting the required variables for the test.
8   */
9  void setUp(void) {}
10
11 /**
12  * @brief This function is called after running every test function to cleanup the environment.
13  *        For example if the module needs to free memory, to be stopped, or etc. we can do it in this function
14  */
15 void tearDown(void) {}
16
17 // This is a test function. In these test cases we are going to check for prime numbers
18 void test_prime_numbers(void)
19 {
20     TEST_ASSERT_TRUE(prime_check(2));
21     TEST_ASSERT_TRUE(prime_check(3));
22     TEST_ASSERT_TRUE(prime_check(23));
23 }
24
25 // This is another test case. In these test cases we are going to check for non-prime numbers
26 void test_non_prime_numbers(void)
27 {
28     TEST_ASSERT_FALSE(prime_check(0));
29     TEST_ASSERT_FALSE(prime_check(1));
30     TEST_ASSERT_FALSE(prime_check(20));
31     TEST_ASSERT_FALSE(prime_check(1000));
32 }
33
34 int main(void)
35 {
36     UNITY_BEGIN(); // Start Unity
37
38     // Run test functions
39     RUN_TEST(test_prime_numbers);
40     RUN_TEST(test_non_prime_numbers);
41
42     return UNITY_END(); // End Unity
43 }
44
```