# Assignment 4 – Code Coverage

While writing unit tests is important, one of the ways to check if tests are effective is to use code coverage. In this assignment, we will learn how to obtain line coverage of a C++ program.

Due to limited time, you do not need to use `#include "catch.hpp"` for writing tests today. Call everything from the main function.

## Problem 1 [20]

Create a folder called 'problem1' and inside it create two more folders 'src', 'build'. The 'src' folder will contain all source cpp files while the 'build' folder will contain the executables.

Write a C++ program with the following functionality.

1. Create a new file *main.cpp* in 'src' folder.

```cpp
#include <iostream>

using namespace std;


int main()
{
   int x = 5;
   if (x == 5)
   {
      cout << "x is 5" << endl;
   }
   else
   {
      cout << "x is not 5" << endl;
   }
   return 0;
}
```

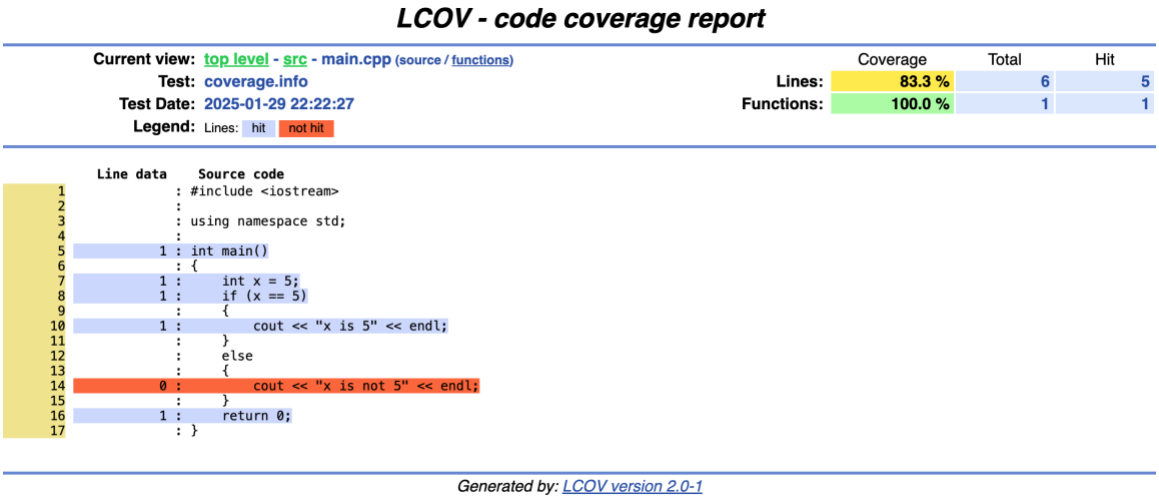
2. Compile it using
   `g++ -fprofile-arcs -ftest-coverage  -o build/program -O0 src/main.cpp`
3. For our case, we want the executable to be written to the 'build' directory

4. Now run the program using ./build/program
5. This should create .gcda and .gcno files in the build directory which stores the structure of the program and the runtime information. These files contain information about which lines of code were executed during the program's runtime.
6. There are two tools/commands available for summarizing the coverage report. One is called gcov and another is called lcov.
7. Let's use 'lcov' to summarize the results of the coverage. The following is the command.

   lcov --capture --directory ./build/ --output-file coverage.info

   This command captures the coverage information data from the ./build directory and saves it to a file called coverage.info
8. Now, to properly view the results, let's create a HTML report using the command which should write the report to a directory called 'out'.

   genhtml coverage.info --output-directory out --legend
9. Open index.html from 'out' directory in your browser. For main.cpp, it should look like the following.



**LCOV - code coverage report**

| | | | | | |
|---|---|---|---|---|---|
| Current view: | top level - src - main.cpp (source / functions) | | Coverage | Total | Hit |
| Test: | coverage.info | Lines: | 83.3 % | 6 | 5 |
| Test Date: | 2025-01-29 22:22:27 | Functions: | 100.0 % | 1 | 1 |
| Legend: | Lines: hit not hit | | | | |

```
Line data    Source code
   1           : #include <iostream>
   2           :
   3           : using namespace std;
   4           :
   5         1 : int main()
   6           : {
   7         1 :     int x = 5;
   8         1 :     if (x == 5)
   9           :     {
  10         1 :         cout << "x is 5" << endl;
  11           :     }
  12           :     else
  13           :     {
  14         0 :         cout << "x is not 5" << endl;
  15           :     }
  16         1 :     return 0;
  17           : }
```

Generated by: LCOV version 2.0-1

10. The results show the lines that hit during execution of the program.
    a. The red lines represent the missed lines during execution.
    b. Overall, the data shows line coverage of 83% which means the program execution triggered 83% of the lines.
11. As a software engineer, coverage is an important metric used to check the effectiveness of written tests.

Remove the content of the main function which was meant for this tutorial. In the same file, implement the following functionality:

- Write a class called NumberOps and implement the following functions. The required data members are for you to choose.
- Write a function that takes a vector of integers as a parameter and does not return anything. It prints the count of positive numbers, the count of negative numbers and count of the zeros. **[5]**
- Write a function that takes a vector of only positive integers as a parameter and finds the first prime number and returns it. Check if passed vector contains any invalid input. **[10]**
- Write a function that takes a vector of only negative integers as a parameter and returns the sum of all even numbers. Check if passed vector contains any invalid input. **[5]**

Now write tests such that you achieve 100% line coverage.

Note: If there are issues with running "lcov" after executing tests. For such cases, you can use gcov on the .gcno files.
Example command:

```
gcov -r ./build/tests-numOps.gcno
```

**Important: You might want to delete .gcda and .gcno files when executing the program again since the coverage information gets accumulated in multiple runs. Your submission will be judged by only a single run.**

## Problem 2 [80]

For this problem, follow the same folder structure. i.e., all source files should be kept in "src" and all executable in "build".

Create an Employee class with the following private data members:

- name – This should be dynamically allocated
- eid – This is an int
- age – This is an int
- salary – This double
- static int count_of_employees – To count the total number of employees in the company. We will learn more about static data members in future lectures. Quick definition is "static data members are class members that are shared among all objects of the class rather than being specific to a particular object".

In the class you should have at least the following methods:

- A parameterized constructor to initialize the employee details. You need to handle exceptions for invalid inputs.
    - Valid age
    - Valid and non-empty name.
    - Valid reasonable salary
- A copy constructor to create a copy of an existing Employee object
- A destructor to free any allocated memory and handle employee count.
- A display method to print employee details to the console
- Overload + to combine salaries of two employees and returns a new Employee object with salary set to the combined salary. The values of other details (such as name, age etc.) of this new Employee object is for you to decide.
- A static function get_count_of_Employees() to get the count of employees. The signature is

  ```
  static int get_count_of_Employees(){
  // Your code
  }
  ```

  You may call this using the following syntax.

  ```
  Employee:: get_count_of_Employees();
  ```
- A function to update salary of an employee while handling invalid inputs.

Now, create objects of the Employee class in the main function and properly test your application to achieve 100% line coverage.

Grade distribution:

- Parameterized constructor **[10]**
- Copy Constructor **[20]**
- Destructor **[10]**
- Display **[10]**
- Overload + **[15]**
- Static function to get count **[5]**
- Function to update salary **[10]**

Sub-parts

- Complete and correct implementation with 100% line coverage **[100%]**
- Complete and correct but does not achieve 100% line coverage. Anything less than 100. **[70%]**

- Incomplete with 100% coverage **[20%]**
- Incomplete without full coverage **[0%]**