CEEDLING

(Unit testing in C)

S.Lochani Vilehya 43317

Introduction to unit testing

- A unit is simply a small piece of code for any single function. So when we test those units, it is called a unit test.
- The unit test is a short script or piece of code designed to verify the behavior of a particular unit independently to produce a pass or fail result.
- Unit Testing is performed during the application development phase.
- It is a white box testing technique

Introduction to ceedling

- Ceedling is a build system and **test framework for C**, designed to streamline the process of **Test-Driven Development (TDD)** in C projects.
- It's distributed as a Ruby gem and provides an automated way to set up, build, and run unit tests in C, making it easier to apply TDD principles.
- Ceedling includes a **test framework (Unity), a mocking framework (CMock), and CException.**
- Ceedling is highly useful for developers working in embedded systems or low-level C programming.

Unity

- Unity is a simple and efficient **unit testing framework for C**
- It is written completely in **C** and is portable, quick, simple, expressive, and extensible. It is designed to especially be also useful for **unit testing for embedded systems.**
- It provides a **set of assertions** to validate the behavior of your code.
- These assertions are designed to test various conditions and report failures when expectations are not met.
- Refer the below link to know the set of test assertions available in unity:
    - https://embetronicx.com/tutorials/unit_testing/unit-testing-in-c-testing-with-unity/#TEST_ASSERT_XXX_Functions

CMock

- CMock is a tool used in C programming to automatically create "mock" versions of functions for testing purposes.
- It creates fake versions of the functions defined in your code’s header files.
- These **fake functions** can be used in tests to simulate how the real functions behave without having to use the actual implementation.
- **Features :**
  - You can expect a function to be called with specific values.
  - You can set the return value of a mocked function.
  - You can tell CMock to ignore certain values if they’re not important for the test.
  - You can create a custom action when a mock function is called,

CException

- C doesn’t have built-in exception handling like some other languages. If something goes wrong, like a function failing, C just returns an error code.
- CException is a tool that provides exception handling in C, similar to how you might use try, catch, and throw in other programming languages (like Java or Python)
- **Key concepts :**
  - **Try{...}**: This is the part of your code where you want to check if an error happens.
  - **Throw(e)**: If an error happens, you "throw" an exception to signal that something went wrong.
  - **Catch(e) {}**: After you "throw" an exception, you can "catch" it and decide how to handle the error.
  - **ExitTry()** : It is a method used to immediately exit your current Try block but NOT treat this as an error. Don't run the Catch. Just start executing from after the Catch as if nothing had happened.

ceedling installation

$ sudo apt install ruby

$ sudo gem install ceedling

Verification of the ceedling installation

vlab@HYVLAB8:~/lochu/unity_testing_practice/test_math$ ceedling
Welcome to Ceedling!
Commands:
  ceedling example PROJ_NAME [DEST] # new specified example project (i...
  ceedling examples   # list available example projects
  ceedling help [COMMAND]   # Describe available commands or o...
  ceedling new PROJECT_NAME   # create a new ceedling project
  ceedling upgrade PROJECT_NAME   # upgrade ceedling for a project (...
  ceedling version   # return the version of the tools ...

vlab@HYVLAB8:~/lochu/unity_testing_practice/test_math$ ceedling version
Welcome to Ceedling!
      Ceedling:: 0.31.1
      CMock:: 2.5.4
      Unity:: 2.5.4
      CException:: 1.3.3

Creating a new project

- Create the project using ceedling new proj_name using the command terminal (command prompt) in your desired directory (folder).
    - **ceedling new <dir_name>**

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling$ ceedling new
simple_pro
Welcome to Ceedling!
      create  simple_pro/project.yml

Project 'simple_pro' created!
 - Execute 'ceedling help' from simple_pro to view available test
& build tasks

vlab@vlab-OptiPlex-3040:~/lochu/ceedling$ cd simple_pro/
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$ ls
project.yml  src  test

vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$ tree
.
├── project.yml
├── src
└── test
    └── support

3 directories, 1 file
```

Add the dependencies

- Create a directory name **“vendor”** in you project directory
- Git clone the **cmock and unity** dependencies
  - **git clone https://github.com/ThrowTheSwitch/CMock**
  - **git clone git@github.com:ThrowTheSwitch/Unity.git unity**
- After cloning unity from git , get the unity related files (.c and .h) present in **“src”** folder to your local project directory
  - **mv vendor/unity/src/unity* .**

Create a source template

- To create source code and test cases, we need .c and .h files. Using ceedling also we can create the source template.
- We can create a module using :
  - **ceedling module:create[module_name].**

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$
ceedling module:create[bit_manipulation]
File src/bit_manipulation.c created
File src/bit_manipulation.h created
File test/test_bit_manipulation.c created
Generate Complete
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$
tree
.
├── project.yml
├── src
│   ├── bit_manipulation.c
│   └── bit_manipulation.h
└── test
    ├── support
    └── test_bit_manipulation.c

3 directories, 4 files
```

Default unit test code

- Open the test_bit_manipulation.c under the test directory.
- In that test_bit_manipulation.c, there might be some default code available already like beside.

```
#include "unity.h"

#include "bit_manipulation.h"

void setUp(void)
{
}

void tearDown(void)
{
}

void
test_bit_manipulation_NeedToImplement(void)
{
    TEST_IGNORE_MESSAGE("Need to Implement
bit_manipulation");
}
```

What is setup function?

- It is a type of start function (like Constructor function) that is used to initialize some variables and allocate memory if we want to.
- This setUp function will be executed before each test function.
- If you have three test functions in your test file, setUp gets called three times.

- Example :

```
void setUp(void)
{
 a = 0x00;
 b = 0xFF;
 c = 0x00;
}
```

What is teardown function?

- This is a type of end function (like destructor function) that is used to free the memory.
- This tearDown function will be executed after each test function.
- If you have three test functions in your test file, tearDown gets called three times.

Writing a sample test case

- Let's write a test case to test the function **“check_bit(uint8_t pos)”**
- So , create a test function called **“test_bit(void)”** . This function name can be anything (should be unique) but make sure it starts with “test_”
- This test case is negative test. When I pass more than 7 , it should return -1 and it should not modify any of global variables

test/test_bit_manipulation.c

```
#ifdef TEST
#include "unity.h"
#include "bit_manipulation.h"

extern uint8_t a;
extern uint8_t b;
extern uint8_t c;

void setUp(void)
{
      a=0x00;
      b=0xFF;
      c=0x00;
}

void tearDown(void)
{
}
```

```
void test_bit(void)
{
      int8_t result;
      result = do_bit_man( 15 );
      TEST_ASSERT_EQUAL_INT8( -1, result );
      TEST_ASSERT_EQUAL_INT8( 0x00, a );
      TEST_ASSERT_EQUAL_INT8( 0xFF, b );
      TEST_ASSERT_EQUAL_INT8( 0x00, c );

}

#endif // TEST
```

src/bit_manipulation.c

```
#include "bit_manipulation.h"

uint8_t a = 0x00;
uint8_t b = 0xFF;
uint8_t c = 0x00;


int8_t check_bit (int8_t pos)
{
    if( ( pos < 0 ) || ( pos > 7 ) )
    {
        //pos should be 0 to 7. Because we are going to modify 8 bit value.
        return -1;
    }

    return 0;
}
```

src/bit_manipulation.h

```
#ifndef BIT_MANIPULATION_H
#define BIT_MANIPULATION_H

#include <stdio.h>
#include <stdint.h>

int8_t check_bit (int8_t pos);

#endif // BIT_MANIPULATION_H
```

Testing the code using ceedling

- To verify test, you can run **“ceedling test:all”** command
- Make sure that you are running the command terminal on the directory where the project.yml file is present.

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$ ceedling test:all

Test 'test_bit_manipulation.c'
------------------------------
Generating runner for test_bit_manipulation.c...
Compiling test_bit_manipulation_runner.c...
Compiling test_bit_manipulation.c...
Linking test_bit_manipulation.out...
Running test_bit_manipulation.out...

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

Checking Code Coverage

- To verify the code coverage install “gcovr” using pip
  - **pip install gcovr**
- Now add **gcov plugin** in the project.yml file created in your project directory
  - Add **“-gcov”** in the **“enabled”** section of **plugins**

```
:plugins:

  :load_paths:

    - "#{Ceedling.load_path}"

  :enabled:

    - stdout_pretty_tests_report

    - module_generator
```

Usage and output of gcov

- To run the gcov use the command
  - **ceedling gcov:all**
- If you have previous reports generated by the gcov you clean them using following commands
  - **ceedling gcov:clean**
  - **rm -rf ./build/artifacts/gcov**

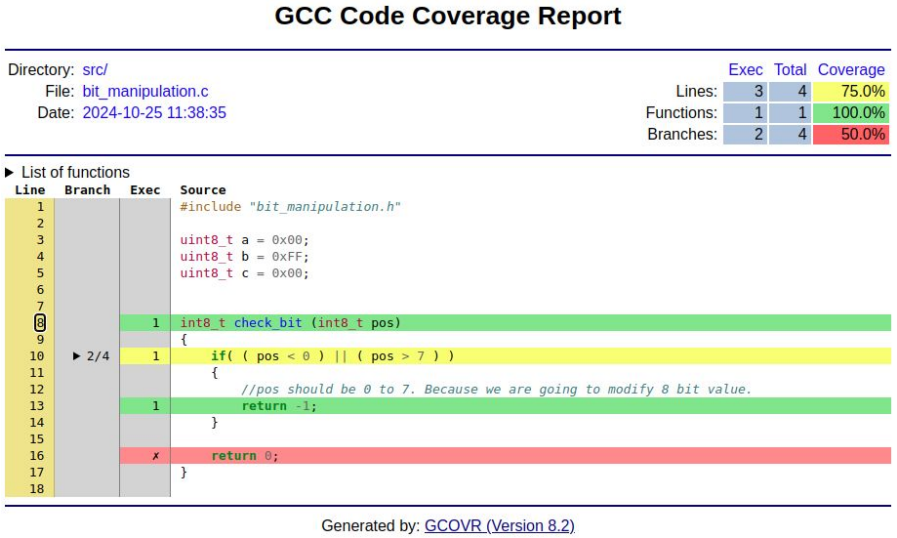
```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$ ceedling gcov:all

Test 'test_bit_manipulation.c'
------------------------------
Compiling test_bit_manipulation_runner.c...
Compiling test_bit_manipulation.c...
Compiling unity.c...
Compiling bit_manipulation.c with coverage...
Compiling cmock.c...
Linking test_bit_manipulation.out...
Running test_bit_manipulation.out...
--------------------------
GCOV: OVERALL TEST SUMMARY
--------------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
--------------------------
GCOV: CODE COVERAGE SUMMARY
--------------------------
bit_manipulation.c Lines executed:75.00% of 4
bit_manipulation.c Branches executed:100.00% of 4
bit_manipulation.c Taken at least once:50.00% of 4
bit_manipulation.c No calls
```

Html report generation using gcov

- If you want to generate a detailed HTML review, then please use the below command after ceedling gcov:all.
    - **ceedling utils:gcov**
- After running the command you will get the prints like this :

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$
ceedling utils:gcov
Creating gcov results report(s) in
'build/artifacts/gcov'...
(INFO) Reading coverage data...
(INFO) Writing coverage report...
Done in 0.730 seconds.
```

- Once it is generated, then you can see the HTML file in simple_prog\build\artifacts\gcov. Open those two files which are generated and analyze.



GCC Code Coverage Report

Directory: src/
File: bit_manipulation.c
Date: 2024-10-25 11:38:35

	Exec	Total	Coverage
Lines:	3	4	75.0%
Functions:	1	1	100.0%
Branches:	2	4	50.0%

▶ List of functions

Line	Branch	Exec	Source
1			#include "bit_manipulation.h"
2			
3			uint8_t a = 0x00;
4			uint8_t b = 0xFF;
5			uint8_t c = 0x00;
6			
7			
8		1	int8_t check_bit (int8_t pos)
9			{
10	▶ 2/4	1	if((pos < 0) || (pos > 7))
11			{
12			//pos should be 0 to 7. Because we are going to modify 8 bit value.
13		1	return -1;
14			}
15			
16		✗	return 0;
17			}
18			

Generated by: GCOVR (Version 8.2)

Contd...

- Whenever you regenerate the report please clean it and regenerate or follow the steps below to get the updated report.
  - **ceedling clean** – This will clean the generated files.
  - **ceedling test:all** – build and test the test case
  - **ceedling gcov:all–** generate coverage result
  - **ceedling utils:gcov** – Generate the HTML detailed report

Mocking

- In coding , Mocking is a way to replace some functions with alternate implementations (“mocks”).
- An object that you want to test may have dependencies on other modules. To isolate the behavior of the code or function you want to test, you need to replace the other dependencies with mocks.
- Using fake functions, we can eliminate the dependencies and test the code that we want to test.
- So, Mocks are used to break module dependencies so that they can be tested in isolation.Ceedling has an inbuilt tool for doing that mocking. That is CMock. So we don’t need to install anything.

Usage and example of CMock

- CMock is a framework for generating mocks based on a header API.
- All you have to do to use CMock is add a mock header file to the test suite file.
- You can generate the mock functions using #include "mock_example.h". Here, example.h is your file to create a mock.

- From the previous that we wrote to know usage of ceedling and unity :
  - Lets create new source files temp.c and temp.h where temp_read function is present
  - In check_bit function, we need to print whether the temp is high or low
  - Add #include “temp.h” line in the test_bit_manipulation.c file

temp.c

```
#include "temp.h"

//We will assume that, temp value
will be updated in this value.
volatile int temp_value = 0;


int temp_read( void )
{
        return temp_value;
}
```

temp.h

```
#ifndef TEMP_H
#define TEMP_H

/* Read temp value */
int temp_read( void );

#endif // TEMP_H
```

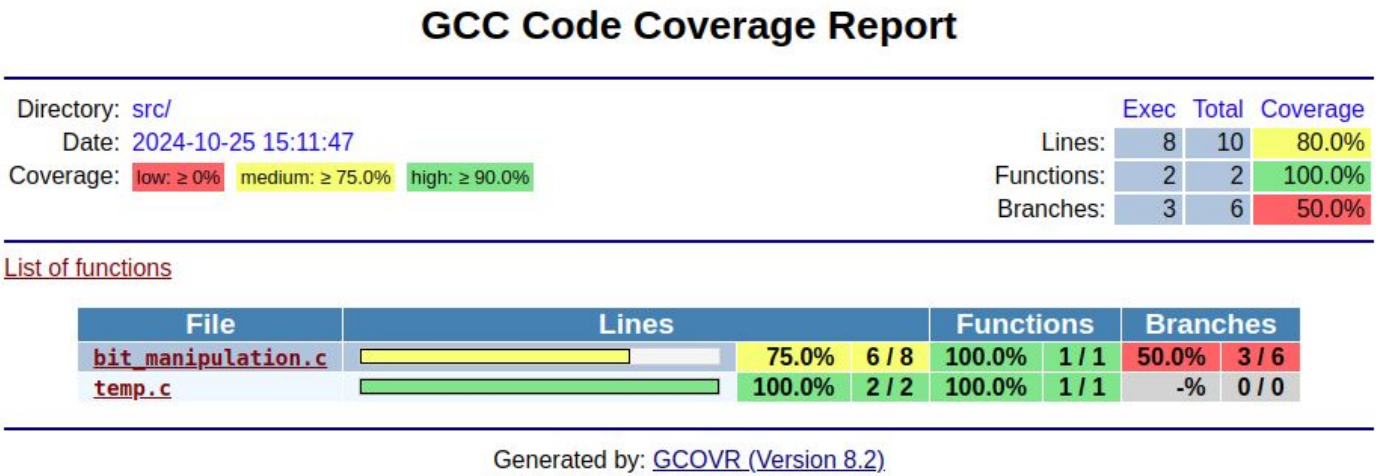
Modification of bit_manipulation.c

```
#include "bit_manipulation.h"
#include "temp.h"

uint8_t a = 0x00;
uint8_t b = 0xFF;
uint8_t c = 0x00;


int8_t check_bit (int8_t pos)
{
    if( ( pos < 0 ) || ( pos > 7 ) )
    {
        //pos should be 0 to 7. Because we are going to modify 8 bit value.
        return -1;
    }
    int res = temp_read();
    if(res >= 30)
    {
           printf("temperature is high\n");
    }
    else
    {
           printf("temperature is low\n");
    }

    return 0;
}
```

Note : Change the position value you are passing to **check_bit()** in test_bit() function in **test_bit_manipulation.c to "5".**so it may not return -1 and continue the execution of below code of temperature

Gcov report

- Here, temp.c is 100% but bit_manipulation.c is 75%. adc.c is out of scope for us. Because we are testing check_bit not temp_read. Why did this 25% get reduced in bit_manipulation.c
- Here We have missed one line and one branch. This branch (if) is not been covered by our unit testing code.



GCC Code Coverage Report

Directory: src/
Date: 2024-10-25 15:11:47
Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

	Exec	Total	Coverage
Lines:	8	10	80.0%
Functions:	2	2	100.0%
Branches:	3	6	50.0%

List of functions

File	Lines			Functions		Branches	
bit_manipulation.c		75.0%	6 / 8	100.0%	1 / 1	50.0%	3 / 6
temp.c		100.0%	2 / 2	100.0%	1 / 1	-%	0 / 0

Generated by: GCOVR (Version 8.2)

GCC Code Coverage Report

Directory: src/
File: bit_manipulation.c
Date: 2024-10-25 15:11:47

	Exec	Total	Coverage
Lines:	6	8	75.0%
Functions:	1	1	100.0%
Branches:	3	6	50.0%

▸ List of functions

Line	Branch	Exec	Source
1			#include "bit_manipulation.h"
2			#include "temp.h"
3			
4			uint8_t a = 0x00;
5			uint8_t b = 0xFF;
6			uint8_t c = 0x00;
7			
8			
9			
10		1	int8_t check_bit (int8_t pos)
11			{
12	▸ 2/4	1	if((pos < 0) || (pos > 7))
13			{
14			//pos should be 0 to 7. Because we are going to modify 8 bit value.
15		✗	return -1;
16			}
17			
18			
19		1	int res = temp_read();
20	▸ 1/2	1	if(res >= 30)
21			{
22		1	printf("temperature is high\n");
23			}
24			else
25			{
26		✗	printf("temperature is low\n");
27			}
28			
29		1	return 0;
30			}
31			

Generated by: GCOVR (Version 8.2)

Generating Mock using CMock

- To generate a mock for the header file, all we need to do is include the mock header file name in one of our unit test files. The mock header file name is the original header name prepended with mock_.
- For example, to create mocks for the functions in **temp.h**, we just include **mock_temp.h** in our **test_bit_manipulation.c** instead.
- Ceedling automatically creates this **mock_adc.h** file and a corresponding implementation in **mock_adc.c** by using CMock.

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$
ceedling test:all


Test 'test_bit_manipulation.c'
------------------------------
Generating include list for temp.h...
Creating mock for temp...
Generating runner for test_bit_manipulation.c...
Compiling test_bit_manipulation_runner.c...
Compiling test_bit_manipulation.c...
Compiling mock_temp.c...
Compiling unity.c...
Compiling bit_manipulation.c...
Compiling cmock.c...
Linking test_bit_manipulation.out...
Running test_bit_manipulation.out...
```

Contd...

- If you want to see all the mock control functions that CMock has generated for a particular module, you can find them in the **.h files in build/test/mocks.**
- For each function present in **temp.h**, It generates a bunch of new functions that you use to control the mock function. These new functions are based on the name and parameters of the original function.
- In my case , the below functions will be generated for temp_read() function.
    - void temp_read_CMockIgnoreAndReturn(UNITY_LINE_TYPE cmock_line, int cmock_to_return)
    - void temp_read_CMockStopIgnore(void)
    - void temp_read_CMockExpectAndReturn(UNITY_LINE_TYPE cmock_line, int cmock_to_return)
    - void temp_read_AddCallback(CMOCK_temp_read_CALLBACK Callback)
    - void temp_read_Stub(CMOCK_temp_read_CALLBACK Callback)

Mock variants

- There are multiple variants of mock functions that will be generated.
- Some of the most used variants are :
  - Expect
  - Ignore Arguments
  - Array
  - Ignore
  - ReturnThruPtr
  - Callback
- For detail information of mock variants refer below link
  - https://embetronicx.com/tutorials/unit_testing/unit-testing-in-c-mock-using-cmock-in-embedded/#Mock_Variants

The solution of the original code

- In order to achieve 100% code coverage, we have two branches which are if and else. So we are going to update our old test_bit_manipulation.c with cmock.
- I have added CMock functions to Test cases 3 and 4. Let's see the code coverage by using the below commands.

```
void test_bit(void)
{
	int8_t result;

	temp_read_CMockExpectAndReturn(__LINE__,10);
	result = check_bit( 5 );
	TEST_ASSERT_EQUAL_INT8( -1, result );
	TEST_ASSERT_EQUAL_INT8( 0x00, a );
	TEST_ASSERT_EQUAL_INT8( 0xFF, b );
	TEST_ASSERT_EQUAL_INT8( 0x00, c );

}
```

```
void test_bit_1(void)
{
	int8_t result;

	temp_read_CMockExpectAndReturn(__LINE__,35);
	result = check_bit( 5 );
	TEST_ASSERT_EQUAL_INT8( -1, result );
	TEST_ASSERT_EQUAL_INT8( 0x00, a );
	TEST_ASSERT_EQUAL_INT8( 0xFF, b );
	TEST_ASSERT_EQUAL_INT8( 0x00, c );

}
```

Output after using mock functions

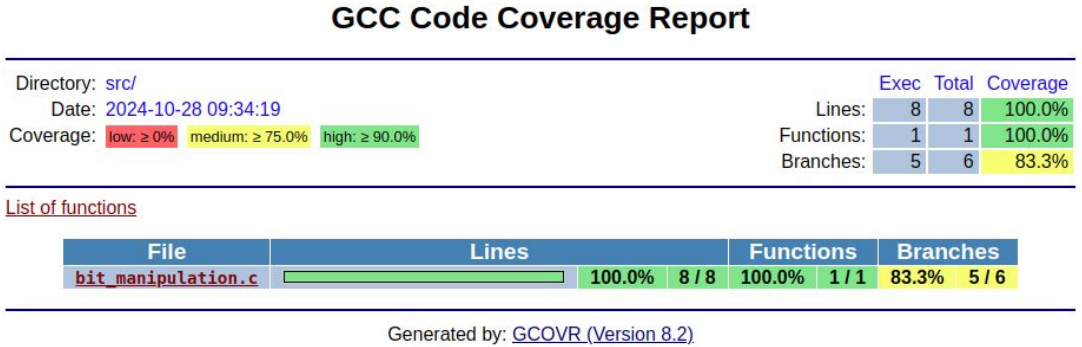
- Now we will observe that both the if and else statements are covered , we get 100% coverage

vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro$ ceedling gcov:all --trace
Test 'test_bit_manipulation.c'

Compiling test_bit_manipulation_runner.c...
Compiling test_bit_manipulation.c...
Linking test_bit_manipulation.out...
Running test_bit_manipulation.out...
.
.

GCOV: TEST OUTPUT

[test_bit_manipulation.c]
- "temperature is low"
- "temperature is high"



GCC Code Coverage Report

Directory: src/
Date: 2024-10-28 09:34:19
Coverage: low: ≥ 0% medium: ≥ 75.0% high: ≥ 90.0%

	Exec	Total	Coverage
Lines:	8	8	100.0%
Functions:	1	1	100.0%
Branches:	5	6	83.3%

List of functions

File	Lines			Functions		Branches	
bit_manipulation.c		100.0%	8 / 8	100.0%	1 / 1	83.3%	5 / 6

Generated by: GCOVR (Version 8.2)

Limitations of CMock

- If you have created Mock and not using it in the test file , then the test case will fail like this

```
-------------------
FAILED TEST SUMMARY
-------------------
[test_bit_manipulation.c]

  Test: test_bit_1
  At line (40): "Function temp_read.  Called fewer times than expected."
```

- If you want to mock more functions in one unit test , then you have to use the mocking function in the correct order like how they have called in the source code otherwise the test case will fail

Introduction to CException

- As we discussed before , CException is a lightweight error-handling library for C programs
- It relies on two standard functions in C: setjmp and longjmp
- These functions let a program save its current point in execution (setjmp) and then jump back to that saved point later (longjmp).
- They act like a "bookmark" in the code that you can return to if something goes wrong.
- CException wraps setjmp and longjmp in simpler commands called Try and Catch, making it easier to use.
- If an error happens in a Try block, CException uses longjmp to jump to the matching Catch block to handle the error.

Contd...

- In CException, the code that you want to protect from errors is put inside a **Try { }** block. If an error happens inside, the program automatically jumps to the next **Catch block** to handle it.
- You can signal an error by using **Throw(some_number)**. The number you pass with Throw acts as an error code, which you can use in the Catch block to tell what kind of error happened or where it happened.

```
void PrintsZeroThroughFive(void) {
   volatile CEXCEPTION_T e;   // variable to store the
exception code
  int i;
  for (i = 0; i < 6; i++) {
    Try {
      Throw(i); // Throw an error with the value of 'i'
       // This line will never run because Throw jumps to
Catch
    }
    Catch(e) {
      printf("%i ", e); // prints the exception code (i)
    }
  }
}
```

Limitations of CException

- **Return and goto statements :**
  - Avoid using return or goto inside a Try block
  - Try block sets up some temporary memory and updates a global pointer, both of which are cleaned up in the Catch block.
  - If you use return or goto, you'll skip this cleanup, which could cause memory issues or other bugs
- **Local variables:**
  - If you change a local (stack) variable inside the Try block and then want to use its updated value in the Catch block, declare it as volatile
  - Without volatile, the compiler might only store the variable in a register (temporary storage), and the Catch block may not see its latest value.
- **Memory allocation :**
  - If you allocate memory inside a Try block using malloc, that memory won't be freed automatically if an error occurs.
  - Sometimes this is okay, but other times you'll need to clean it up manually in the Catch block.

Configuration of CException

- **For Multi-Tasking Environments:**
  - If your system has multiple tasks (like a real-time OS), you'll need one "stack frame" for each task to store error handling data separately for each task. You also need a way to:
    - Get a task's ID (so it can look up its specific frame).
    - Set the total number of IDs based on the number of tasks.
- If your OS has a built-in way to get task IDs (like 0, 1, 2, etc.), you're set. Otherwise, you'll need to create a method to assign IDs to each task. This method will add a bit of overhead since it's called each time a Try block starts and whenever an error (Throw) happens.
- CException library has default settings, but you can customize them if needed:
  - You can override settings via the command line or by creating a configuration file **(CExceptionConfig.h).**
  - Define **CEXCEPTION_USE_CONFIG_FILE,** and CException will look for this file where you can set custom values.

Customization options

1. **CEXCEPTION_T:** Specifies the type of IDs used for exceptions (default is **unsigned int**).
1. **CEXCEPTION_NONE:** Sets a value that represents "no exception" in your system (default is **0x5a5a5a5a**).
2. **CEXCEPTION_GET_ID:** In multi-tasking, this should call a function to retrieve the task's ID (defaults to **0 for single-tasking systems**).
3. **CEXCEPTION_NUM_ID:** Specifies the total number of task IDs (defaults to **1 for single-tasking**).
4. **CEXCEPTION_NO_CATCH_HANDLER(id):** Sets a backup plan if an exception (Throw) is triggered without a surrounding Try...Catch block, so you can handle critical errors more gracefully.

- **Additional Hook Macros:** If you want to insert custom code at specific points in the exception handling process, you can use these optional hooks:

    1. **CEXCEPTION_HOOK_START_TRY:** Called at the beginning of a Try block.
    2. **CEXCEPTION_HOOK_HAPPY_TRY:** Called after a Try block if no error occurs.
    3. **CEXCEPTION_HOOK_AFTER_TRY:** Called after Try, or just before the Catch block if an error occurs.
    4. **CEXCEPTION_HOOK_START_CATCH:** Called right before entering the Catch block.

Example code : usage of CException

- Added a CExceptionConfig.h , add its definition in CException.h file

```
#ifdef CEXCEPTION_USE_CONFIG_FILE
#include "CExceptionConfig.h"
#endif
```

- Defined all the before mentioned hooks and macros according to our need
- Go through the source code in github
  - https://github.com/lochu-55/Company_training/tree/main/ceedling/simple_pro_CException

Changes made in bit_manipulation.c

```
#include <stdint.h>   // Include for uint8_t and int8_t
#include <stdio.h>    // Include for printf

#include "CExceptionConfig.h"
#include "CException.h"
#include "temp.h"


uint8_t a;  // Assuming this variable is defined elsewhere
uint8_t b;
uint8_t c;

int8_t check_bit(int position) {
    volatile CEXCEPTION_T e;

    StartTryHook();

    Try {
        if (position < 0 || position >= 8) {
            Throw(ERR_BIT_ERROR);  // Throw if bit position is
invalid
        }

        int temperature = temp_read();  // Assuming temp_read() is
defined elsewhere
        if (temperature < 15) {
            Throw(ERR_TEMP_LOW);  // Throw if temperature is too low
        } else if (temperature > 30) {
            Throw(ERR_TEMP_HIGH); // Throw if temperature is too high
        }

        return (a & (1 << position)) ? 1 : 0;  // Return the bit value
        HappyTryHook();

    } Catch(e) {
        StartCatchHook();
        switch (e) {
            case ERR_TEMP_LOW:
                printf("Error: Temperature too low!\n");
                return -1;
            case ERR_TEMP_HIGH:
                printf("Error: Temperature too high!\n");
                return -1;
            case ERR_BIT_ERROR:
                printf("Error: Invalid bit position!\n");
                 return -1;
            default:
                printf("Error: Unknown exception caught!\n");
                return -1;
        }
    }
    AfterTryHook();

}
```

Output after using CException

```
vlab@vlab-OptiPlex-3040:~/lochu/ceedling/simple_pro_CException$ ceedling test:all


Test 'test_bit.c'
-----------------
Running test_bit.out...

-----------
TEST OUTPUT
-----------
[test_bit.c]
  - "Starting Try block..."
  - "Handling exception in Catch block..."
  - "Error: Temperature too low!"
  - "Starting Try block..."
  - "Handling exception in Catch block..."
  - "Error: Temperature too high!"

--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  2
PASSED:  2
FAILED:  0
IGNORED: 0
```

References

- https://github.com/lochu-55/Company_training/tree/main/ceedling
- https://embetronicx.com/unit-testing-tutorials/
- https://github.com/ThrowTheSwitch/Unity
- https://github.com/ThrowTheSwitch/CMock
- https://github.com/ThrowTheSwitch/CException
- https://www.throwtheswitch.org/#download-section

THANK YOU