Programming Test Cases see https://en.wikipedia.org/wiki/Test_case

Why test cases? Because we need to ensure these:
https://en.wikipedia.org/wiki/Defensive_design
https://en.wikipedia.org/wiki/Defensive_programming

When testing a program, especially your own program, beware of confirmation bias. We like evidence proving we are right. No one likes being wrong, no one *wants* to shoot themselves in the foot. The scientific method operates more along the lines of Dijkstra's bug warning: there is no exhaustive test that proves a theory is correct. If scientists fail to prove themselves wrong, they are conditionally satisfied they might be right…until further testing proves otherwise. This is the attitude of a good tester, and of one of the great pioneers in computer science who was not also a woman.

Test cases illustrate steps to exercise all lines of code in a program. They include specific data to be input, the rational for entering it, and the expected output. Each test case step results in PASS or FAIL. In the latter case, unexpected output, the lack thereof, or system error messages must be captured and recorded. *Test cases must be specific enough to be repeatable by the programmer who receives the testing results.*

Positive test cases are done with a representative range (lowest, middle, highest) of valid input values to generate expected output demonstrating all functions of the software. The range of input includes typically expected values and edge case values (minimum, maximum, zero, null, empty, full). The test case documents the expected result so the tester can verify a Pass condition. If a positive test case fails and the input data is within the min/max range, then it indicates a bug. If the input data was outside the min/max range, then it is really a Negative test case.

Multiple tests must be done in a single session to show repeatability of passed test. A single test may be successful, but subsequent tests may reveal problems in the program's logic or housekeeping of internal variables. E.g. the first search for a value within a string may pass the test but if the program were to erroneously begin its search for another input at the last found position of a previous search instead of the string's beginning, a FAIL condition would be identified only with a second test – add comments describing how the bug could be recreated.

Negative test cases employ input values which do not illustrate the program's intended purpose and function. Negative test cases are used to explore beyond the boundaries of positive test case input: over-the-edge case values which are less than the minimum or greater than the maximum edges. Output of validation messages in response to incorrect or unexpected inputs is expected and thus will PASS a negative test case.

Any input generating incorrect output, unexpected program behaviour, or a system error is identified as a FAILed test case. These conditions are documented when test cases are run, and later used to initiate programming maintenance. Validation of user input should be done before the program or OS throws an exception/error – these are usually found just outside the edge cases.

- Negative test cases illustrate all the ways *inputs outside the Positive edge cases* can either generate validation and diagnostic messages (Pass), or cause run-time errors (Fail).
    - Test case Comments identify what validation logic and diagnostic message is required to avoid errors caused by certain inputs.
        - No programming is required in source files or in the comments.

- In a professional development environment, analysts develop, run, and document test cases. The results are sent to the programmer (sometimes offshore in a different time zone) who wrote the code. That programmer rectifies the issues, reruns the tests, and returns the results (often overnight) to the analyst.

Individual test case components

Description:

- The reason and intent of each test; this is like the rational for source code comments.

Positive or Negative Test and its Purpose:

- **"+"** indicates a positive case for inputs in normal range of use or to illustrate the minimum & maximum values that can be processed successfully, i.e. up to the edge cases. All such cases are expected to PASS.
- "–" indicates a negative case using inputs outside the normal range of use. These are expected to generate a validation message or error handling and thus PASS the test. If those inputs result in unexpected / undefined behaviours, the test case FAILs.
- Always state the intended purpose of the input. What is it meant to verify?

Data Input:

- exact value or instructions to create input that illustrates Description and Purpose.
- It specifies exactly what and how data will be input but not why. Why is in the Description.

Expected Output:

- reference value to confirm the test result – this is documented before the test is run.

Actual result if unexpected:

- any variance from the expected output

Success?

- "PASS" or "FAIL"

Comments:

- required for a test case that FAILs and/or for unexpected results.
- If the test fails, recommend a fix to prevent the failure, e.g. a validation check and/or diagnostic message.
- If the test passes, comments are needed only to clarify assumptions, constraints, or special conditions required for the input to result in the expected output.

<u>Useful test string of unique characters</u>

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!@#$%^&*()_+-=[]\{}|<>?/;'":
```

Test Data

Although you have seen the code, test cases should use the black box approach in general. That is, not to make any assumptions about what the code is expecting or assuming the user will do.
Because you have seen the code, you may be able to devise tests to cause the program to FAIL: it either continues to operate while outputting incorrect values, or it crashes with a terminal error.

When entering a string of test data, what will enable you to verify the "character found" function?

Type a string:

➢ **0123456789**

Type the character position within the string: // to be retrieved

➢ **4**

The character found at 4 position is '4'

Is that a meaningful test result? Can you verify the result is from the string or does it reveal a bug which shows the position index instead of the value at that position within the string? Is the input prompt from a human POV, what is in the fourth position? (**3**) Is the program interpreting the input as its index to the string? (**4**)

To reduce confusion, use test data which does not parallel the program's internals. In the above case, use a string of **9876543210** or **ABCDEFGHIJ**

N.B. Using repeating characters or repeating sequences for test input can hide problems.

Enter a string:

➢ **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

Enter the position within the string to be extracted:

➢ **13**

The character found at 13 position is 'a'

Is that a meaningful test result? Can you verify the result is correct?

All values entered in a single test should be different.

For alpha characters, use the Latin alphabet in sequence: abc...xyzABC...XYZ, instead of whatever comes out from mashing the keyboard. A sequence of unique alpha / digits helps to illustrate the position of data within the variable / string array / structure.

A suggestion for numbers to be input in the same test case is 123456789, then 234567891, then 345678912, ... or simply 2, then 3, then 4, …
E.g. Enter a dividend of 1, a divisor of 1, and the quotient output will appear correctly as 1. If the output is the same as the input, any number of bugs could be hiding. Numeric values of -1, 0, 1

represent edge cases. The most careful programmers input only sequences of prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, …

https://www.softwaretestinghelp.com/positive-and-negative-test-scenarios/

C char array in memory

https://www.tutorialspoint.com/cprogramming/c_pointer_to_an_array.htm

reminder: an array name in C is really just a pointer to the first array element at index [0]

```
char letters[27] = {'a','b','c','d','e',...,x,y,z,\0};  // alphabet string
```

When the program is loaded, this tells the OS to allocate a contiguous block of memory to hold char data type × 27. The OS then tells the program where it find it. "letters" becomes a pointer to the memory location of the first element in the array: `&letters[0]` See the above URL.

```
char* pToLetters; // a pointer to a char data type which can be a single char or an array of char.
pToLetters = letters;
```

```
letters == pToLetters
```

```
letters[0 ] == *(pToLetters + 0 ) == *(letters + 0 ) == 'a'
```

```
// pointer arithmetic uses the index to offset the pointer from the beginning to an element.
// that is why the first element is zero.
```

```
letters[25] == *(pToLetters + 25) == *(letters + 25) == 'z'
```

```
letters[26] == *(pToLetters + 26) == *(letters + 26) == \0
```

```
letters[27] == *(pToLetters + 27) == *(letters + 27) == ?
```

```
// whatever is in the byte in the memory location immediately after the upper bound of the array
```

```
letters[-1] == *(pToLetters + -1) == *(letters + -1) == ?
```

```
// whatever is in the byte in the memory location immediately before the lower bound of the array
```

C *assumes* the programmer will never reference an element outside the array bounds. Because C is efficient, C never checks whether the element is within bounds. C trusts the programmer.

C char array – assigning values

```
char charArray[11] = {'0','1','2','3','4','5','6','7','8','9',\0};
```

```
~~~OS~memory~~~*charArray_~~~OS~memory~~~
```

```
~~~~~~~~~~~~~~~0123456789@~~~~~~~~~~~~~~~   \0 shown as @
               ^^^^^^^^^^
```

Assign "abcdefghijklmnopqrstuvwxyz" to charArray

A string assigned to a char array will overwrite memory beginning at &charArray[0] to the end of the string (not the array) and C runtime automatically adds the terminator '\0'. C trusts the string plus terminator fits within the bounds of the char array. In this example, it does not but C carries on as if it will.

```
for (i=0;i < lengthOfString(someString); i++) {
     charArray[i] = *(someString + i);
```

```
}
charArray[i] = '\0'; // auto add terminator
~~~OS~memory~~~*charArray_~~~OS~memory~~~
~~~~~~~~~~~~~~~abcdefghijklmnopqrstuvwxyz0~~~~~~~~~
                ^^^^^^^^^^
```

C char array – retrieving values

Because C does not know where \0 is located, it simply searches until it finds \0. Because C array elements are referenced relative to the beginning of the array, and because C is efficient and trusts the programmer, results will be unpredictable if C's assumptions are not true. This is how C finds the end of a string in a char array:

```
For (i=0; *(charArray + i ) == '\0'; i++)  { printf("%c",charArray[i]; }
// this may continue beyond the array bounds
```

If the OS memory beyond the 11 bytes of `*charArray_` is not overwritten, the entire alphabet may be output. However, the program has no control over what the OS might do with the memory outside the bounds of `*charArray_` so results may be unpredictable.

Summary checklist

- Testing to confirm basic function is just that, basic. And confirms your confirmation bias.
- MIN and MAX edge cases for each kind of input?
- Less than MIN? More than MAX? generate validation messages
- If no tests fail, you may not be trying hard enough.
- When a test does FAIL, what validation logic and/or diagnostic message to the user is recommended to prevent the failure?
- Integration testing of main() : run MIN and MAX edge cases selected from each module's comprehensive tests. Purpose is to confirm essential functionality of the modules when combined into a main() program. Because no source code changes were made to the modules when combined into main() program, exhaustive retesting is not required.
- one worksheet (or file) for each module, one test per row.
- N.B. you are not testing C library functions, only the modules.

Not quite convinced of the value of test cases? See this.