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

"Program testing can be used to show the presence of bugs, but never to show their absence!" – Edsger W. Dijkstra

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 full range of valid input values to generate normally 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 or fail condition. Sometimes a test case will need prior inputs to establish a program's internal state; e.g. *When testing to verify the integrity of values in a string containing its minimum or maximum lengths, a certain sequence of inputs is required …*

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 result could be recreated.

Negative test cases 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. Output of validation messages in response to incorrect or unexpected inputs is usually 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 and the monitoring of system events should be identified, prevented, or caught 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 input can cause errors
    - values just outside the edge cases
    - values causing an error resulting in potentially unexpected, undefined, or unpredictable results,
      e.g. overflow or a system error message.

- Test case Comments identify what validation logic or 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.

Test case components

Case Description:

The reason, purpose, and intent of each test; this is similar to the rational for source code comments.
+ means a positive case which expects a successful result for inputs in normal range of use and to illustrate minimum & maximum values that can be processed, i.e. edge cases.
− means a negative case to generate a validation message or error handling [PASS] or to explore unexpected / undefined behaviours beyond the edge cases [FAIL].

Test Input:

exact value or instructions to create input that illustrates Case Description. This is similar to source code in a program: it describes exactly what and how it will be done but not why. (Why is in the Case Description.)

expected Output or Response:

reference value to confirm the result of Test Input which illustrates the Case Description.

This is similar to the programming debug process by which results of processing are examined to confirm expectations and assumptions (PASS) or to identify an unexpected issue (FAIL).

Comments:

If the test fails, document the unexpected output/response, and 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.

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.

Type a string:

- **aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa**

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

- **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/

reminder: an array in C is really a pointer

C char array in memory

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

`char letters[27] = {'a','b','c','d','e',...,\0};`
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
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[28] == *(pToLetters + 28) == *(letters + 28) == ?
```
// whatever is in the byte in the memory location immediately after end of array

```
letters[-1] == *(pToLetters + -1) == *(letters + -1) == ?
```
// whatever is in the byte in the memory location immediately before start of 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[21] = {'0','1','2','3','4','5','6','7','8','9',')','!','@','#','$','%','^','&','*','(',\0};

```
~~~OS~memory~~~*charArray_____~~~OS~memory~~~
~~~~~~~~~~~~~~~0123456789)!@#$%^&*(0~~~~~~~~~~~~~~~
```
\0 shown as 0

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++)  { . . . }
```
// this may continue beyond the array bounds

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?
    - If there are no failed tests, 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.