Task 1

The program can be build using the makefile, for example

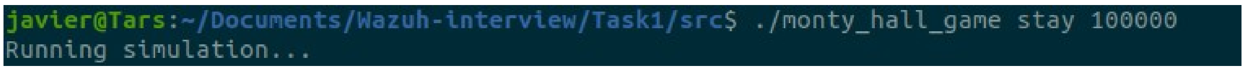
make all: build the program and generate the executable

The program accepts 2 arguments the first one is the type of simulation or in this case the strategy the input should be a string:

- "***stay***" to simulate stay strategy where the player always stays with their initial choice of door.
- "***switch***" to simulate switch strategy where the player always switches their choice of door after a non-prize door is revealed.
- "**both**" to simulate both strategies

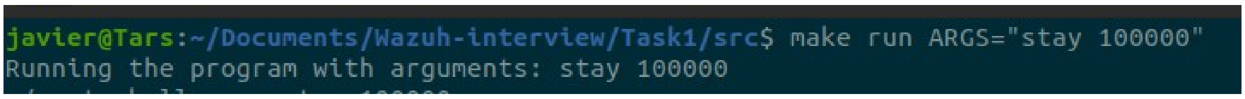
The second one argument is the number of games to run or simulations.

- ***Running from terminal after build:***



```
javier@Tars:~/Documents/Wazuh-interview/Task1/src$ ./monty_hall_game stay 100000
Running simulation...
```

- ***Running using make:***



```
javier@Tars:~/Documents/Wazuh-interview/Task1/src$ make run ARGS="stay 100000"
Running the program with arguments: stay 100000
```

Repository: https://github.com/CyberSoul21/Wazuh-interview

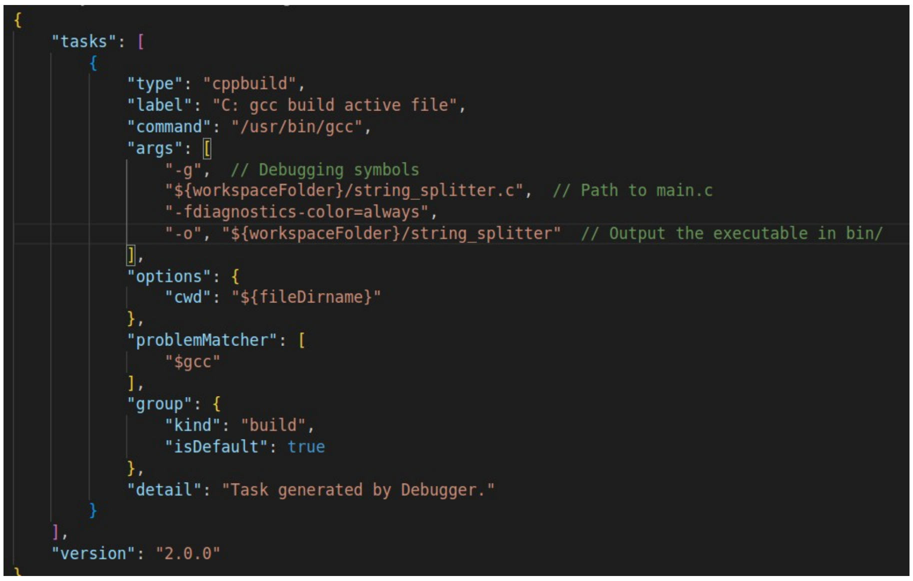
Task 2

Part a)

Procedure:

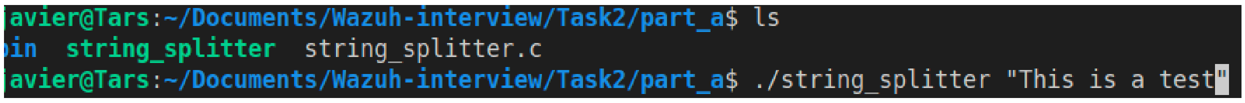
1. Understand the code, this code is a string splitter program in C that splits a string into individual words based on a delimiter (space).
   The program takes an string as argument from command line, then split into individual words and prints in a new line each word.

2. Reproduce the bug:
   To reproduce the code I configure the environment with Visual studio code using GCC compiler:



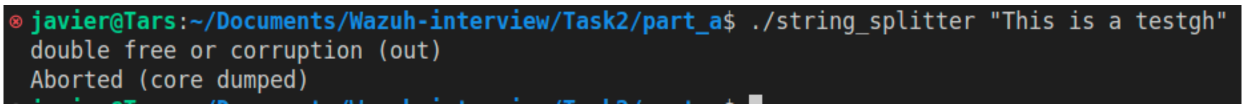
```
{
    "tasks": [
        {
            "type": "cppbuild",
            "label": "C: gcc build active file",
            "command": "/usr/bin/gcc",
            "args": [
                "-g",  // Debugging symbols
                "${workspaceFolder}/string_splitter.c",  // Path to main.c
                "-fdiagnostics-color=always",
                "-o", "${workspaceFolder}/string_splitter"  // Output the executable in bin/
            ],
            "options": {
                "cwd": "${fileDirname}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Task generated by Debugger."
        }
    ],
    "version": "2.0.0"
```

   and launch with the example:



```
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ ls
bin  string_splitter  string_splitter.c
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ ./string_splitter "This is a test"
```

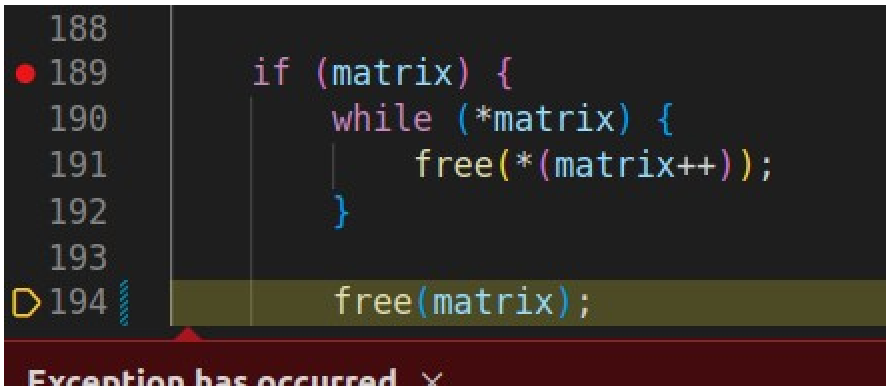
3. Locate the error. In this case shows a error related with the memory



```
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ ./string_splitter "This is a testgh"
double free or corruption (out)
Aborted (core dumped)
```

4. Warning were enabled during compilation time, but it does not show nothing.

5. The next step was use GNB debugger on Visual Studio Code, here the error was located in the function ***matrixfree()***



```
188
189     if (matrix) {
190         while (*matrix) {
191             free(*(matrix++));
192         }
193
194         free(matrix);
```
Exception has occurred

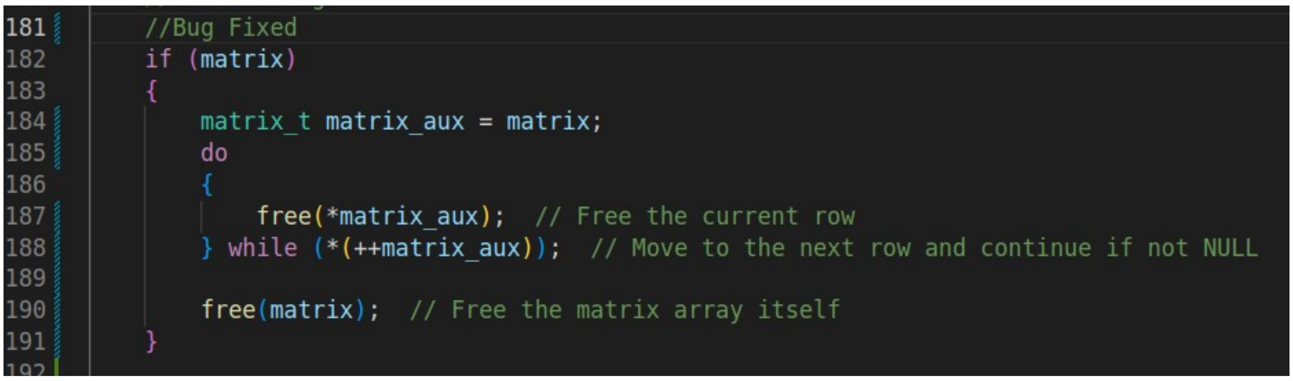
Bug 1:

Analyzing the function, it was designed to free memory dynamically allocated ***matrix,*** The function goes through each row of a dynamically allocated ***matrix***, then, frees each row, and then moves on to the next row.

The error shows above is "Corruption out " due **free(matrix)**, it is attempting to free an invalid pointer, the pointer beyond the last element in the matrix.

This is happening here because the operation priority:
matrix++ first returns the current value of the pointer, and then increments matrix to point to the next element, it means before it is incremented the dereferencing operator (*) is applied to the value of matrix.

Solution:

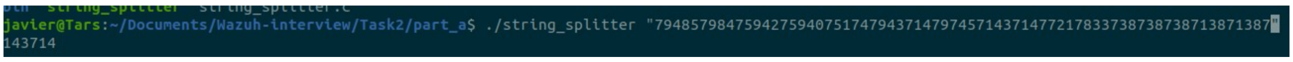


```
181     //Bug Fixed
182     if (matrix)
183     {
184         matrix_t matrix_aux = matrix;
185         do
186         {
187             free(*matrix_aux);  // Free the current row
188         } while (*(++matrix_aux));  // Move to the next row and continue if not NULL
189
190         free(matrix);  // Free the matrix array itself
191     }
192
```

To avoid the bug it is necessary keep the original pointer matrix before incrementing it. In this way when free the array the pointer in line 190, it refer to the original memory address and not the incremented pointer.

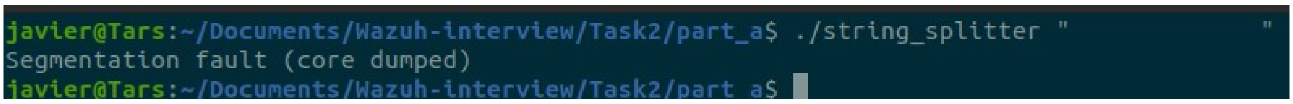
Bug 2:

For find out the bug 2 was necessary test the application with different inputs, for example a large test, only numbers, character specials, a text without spaces...etc.



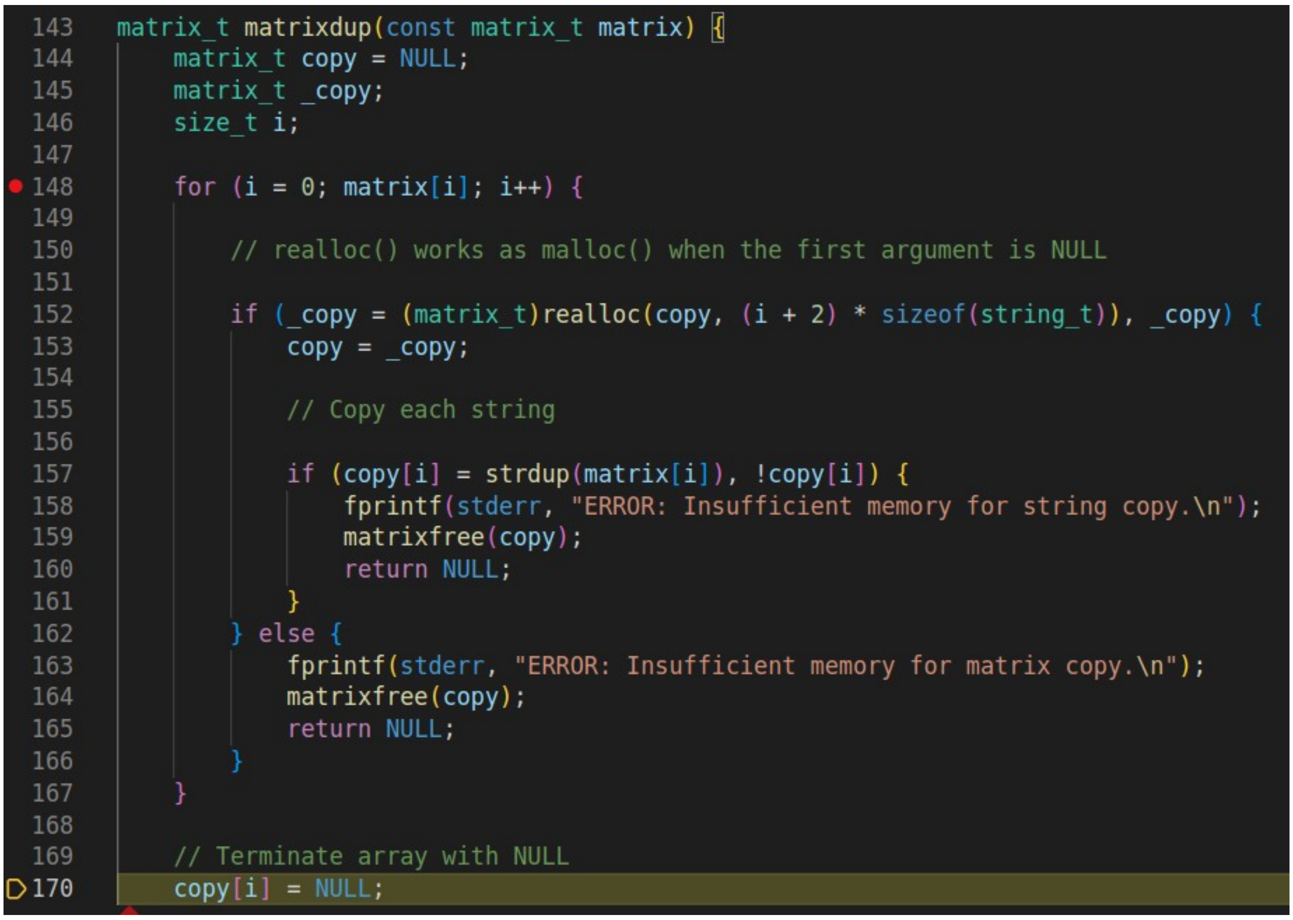
```
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ ./string_splitter "79485798475942759407517479437147974571437147721783373873873871387"
143714
```

Finally the application fail if it has space as input:



```
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ ./string_splitter "                "
Segmentation fault (core dumped)
javier@Tars:~/Documents/Wazuh-interview/Task2/part_a$ 
```

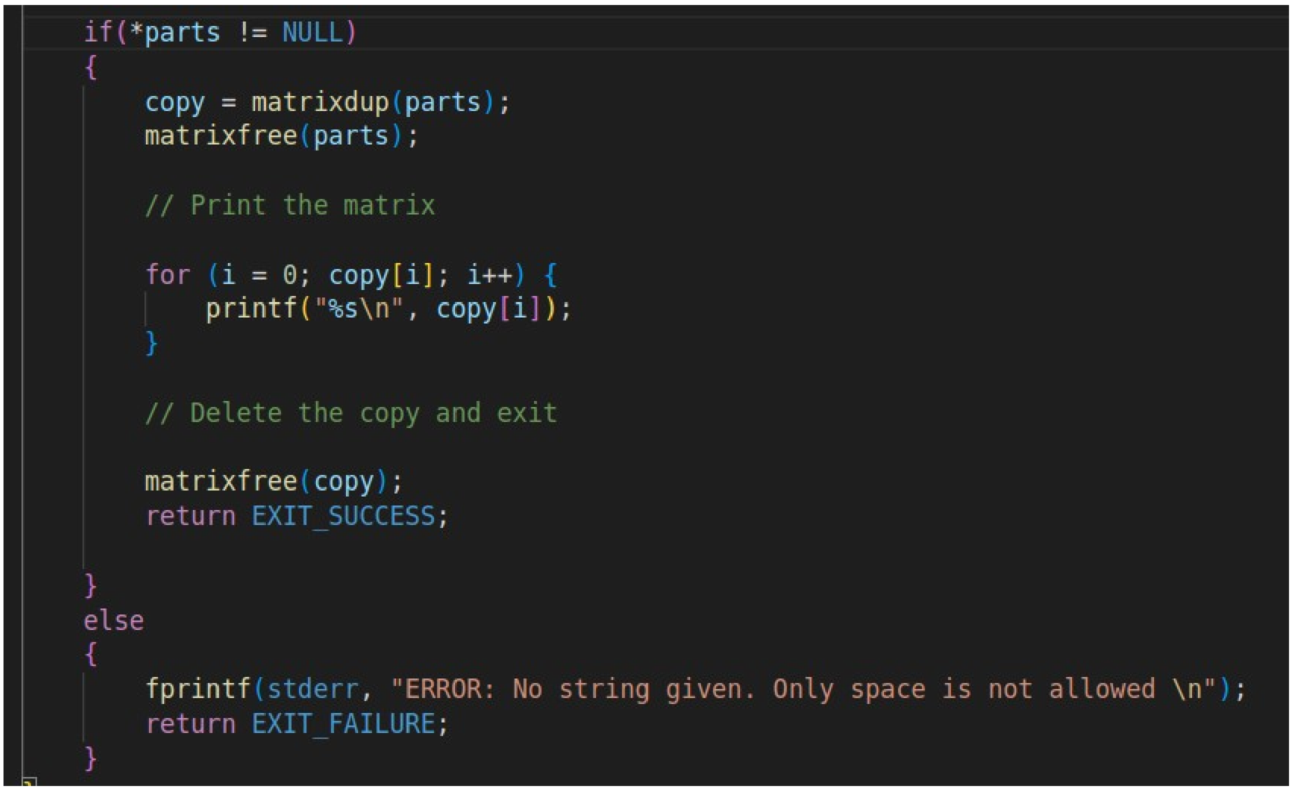
The application fail in the function ***matrixdup*** due ***copy*** has not memory allocated because it never enter to the for loop because ****matrix*** is ***NULL***



```c
matrix_t matrixdup(const matrix_t matrix) {
    matrix_t copy = NULL;
    matrix_t _copy;
    size_t i;

    for (i = 0; matrix[i]; i++) {

        // realloc() works as malloc() when the first argument is NULL

        if (_copy = (matrix_t)realloc(copy, (i + 2) * sizeof(string_t)), _copy) {
            copy = _copy;

            // Copy each string

            if (copy[i] = strdup(matrix[i]), !copy[i]) {
                fprintf(stderr, "ERROR: Insufficient memory for string copy.\n");
                matrixfree(copy);
                return NULL;
            }
        } else {
            fprintf(stderr, "ERROR: Insufficient memory for matrix copy.\n");
            matrixfree(copy);
            return NULL;
        }
    }

    // Terminate array with NULL
    copy[i] = NULL;
```

Solution:

In order to avoid this bug, I added a conditional to check pointer matrix, and letting know to the user about only text is allowed.

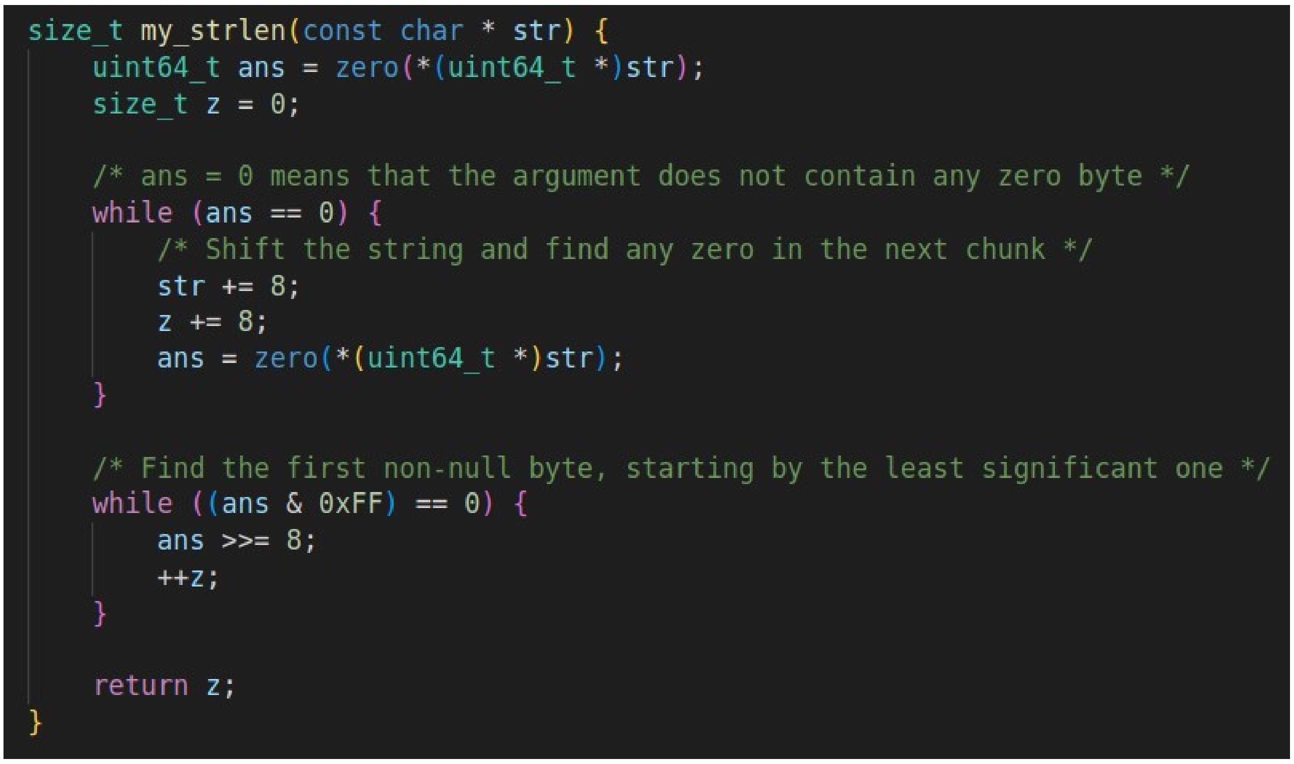


```
if(*parts != NULL)
{
    copy = matrixdup(parts);
    matrixfree(parts);

    // Print the matrix

    for (i = 0; copy[i]; i++) {
        printf("%s\n", copy[i]);
    }

    // Delete the copy and exit

    matrixfree(copy);
    return EXIT_SUCCESS;

}
else
{
    fprintf(stderr, "ERROR: No string given. Only space is not allowed \n");
    return EXIT_FAILURE;
}
```

Part b)

For this case I tested similar as the code on the part a, with a variety of inputs, but after several attempts, I was unable to break the application, so I decided to investigate in depth the code. Take into account the conditions in the comments

 *   1. str points to a valid string.
*   2. That string is zero-terminated.



```
size_t my_strlen(const char * str) {
    uint64_t ans = zero(*(uint64_t *)str);
    size_t z = 0;

    /* ans = 0 means that the argument does not contain any zero byte */
    while (ans == 0) {
        /* Shift the string and find any zero in the next chunk */
        str += 8;
        z += 8;
        ans = zero(*(uint64_t *)str);
    }

    /* Find the first non-null byte, starting by the least significant one */
    while ((ans & 0xFF) == 0) {
        ans >>= 8;
        ++z;
    }

    return z;
}
```

This function may not be meeting the condition, for example it might be added a condition to check that the string is properly null-terminated before processing it. Continue with this idea a proper checking to avoid accessing memory beyond the allocated string buffer might be added too.

This is solved adding the following lines:

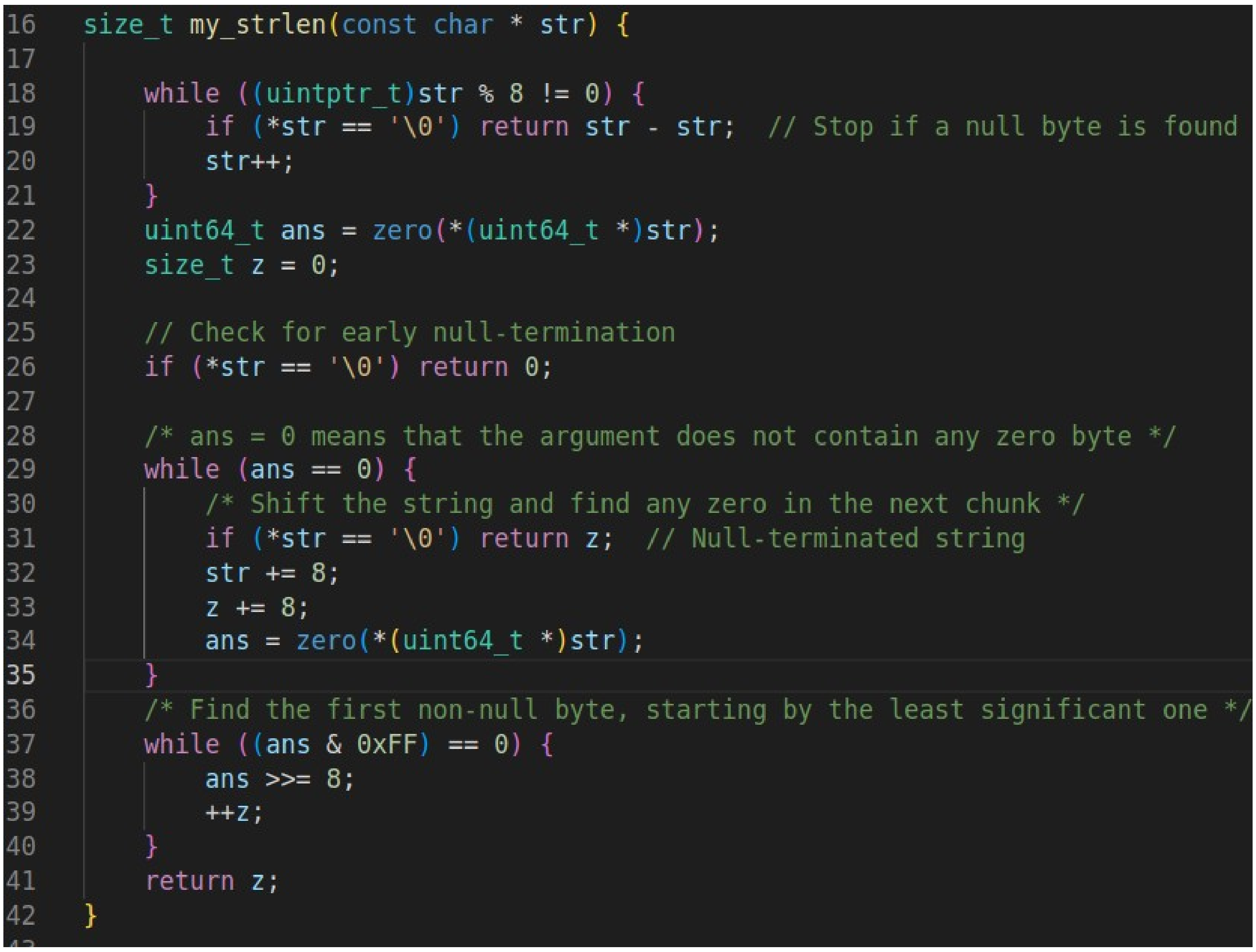
```
// Check for early null-termination
if (*str == '\0') return 0;
```

and ensure string is null-terminated:

```
if (*str == '\0') return z;  // Null-terminated string.
```

In addition investigating more about the architecture x86-64 little-endian architecture where is intended to run on, This highlights that it does not require strict alignment for data, one reason there is not exist a checking if the pointer is aligned to 8 bytes. But it can ensure proper alignment before casting the pointer to a uint64_t* adding the following line:

```
while ((uintptr_t)str % 8 != 0) {
    if (*str == '\0') return str - str;  // Stop if a null byte is found
    str++;
}
```



```
size_t my_strlen(const char * str) {

    while ((uintptr_t)str % 8 != 0) {
        if (*str == '\0') return str - str;  // Stop if a null byte is found
        str++;
    }
    uint64_t ans = zero(*(uint64_t *)str);
    size_t z = 0;

    // Check for early null-termination
    if (*str == '\0') return 0;

    /* ans = 0 means that the argument does not contain any zero byte */
    while (ans == 0) {
        /* Shift the string and find any zero in the next chunk */
        if (*str == '\0') return z;  // Null-terminated string
        str += 8;
        z += 8;
        ans = zero(*(uint64_t *)str);
    }
    /* Find the first non-null byte, starting by the least significant one */
    while ((ans & 0xFF) == 0) {
        ans >>= 8;
        ++z;
    }
    return z;
}
```