Enhanced String Library

A

Project Report

Submitted in partial fulfillment of the

Requirements for the award of the degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE & ENGINEERING

Specialization

Business Analytics and Optimization

By

Student Name	Roll Number
Aswin S	**R103218027**
Indrakshee Roy Choudhury	**R103218060**
Milisha Gupta	**R103218082**
Sarthak Saraiya	**R103218129**

Under the guidance of

Dr. Ravi Tomar

Assistant Professor (Selection Grade)

Department of Informatics
School of Computer Science
University of Petroleum & Energy Studies
Bidholi, Via Prem Nagar, Dehradun, UK

December– 2020

DECLARATION

We hereby certify that the project work entitled **"Enhanced String Library"** in partial fulfillment of the requirements for the award of the Degree of BACHELOR OF TECHNOLOGY in COMPUTER SCIENCE AND ENGINEERING specialization **Business Analytics and Optimization** and submitted to the Department of Informatics, School of Computer Science, University of Petroleum & Energy Studies, Dehradun, is an authentic record of our work carried out during a period from **August, 2020 to December, 2020** under the supervision of **Dr. Ravi Tomar.**

The matter presented in this project has not been submitted by us for the award of any other degree of this or any other University.

Milisha Gupta	**Sarthak Saraiya**	**Indrakshee Roy Choudhury**	**Aswin S**
R103218082	**R103218129**	**R103218060**	**R103218027**

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

Date: 01-12-2020

Dr.Thipendra Pal Singh

Head**,** Department of Informatics

School of Computer Science

University of Petroleum & Energy Studies

Dehradun – 248 001 (Uttarakhand)

Dr. Ravi Tomar

Assistant Professor

(Selection Grade)

ACKNOWLEDGEMENT

We wish to express our deep gratitude to **Dr. Ravi Tomar**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thank to our respected Head of Department, Informatics, **Dr. Thipendra Pal Singh,** for his guidance and support as and when required.

We are also grateful to **Dr. Manish Prateek, Dean SCS**, for providing the necessary facilities to carry out our project work successfully.

We would like to thank all our friends for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our **Parents** who have shown us this world and for everything they have given to us.

Milisha Gupta	**Sarthak Saraiya**	**Indrakshee Roy Choudhury**	**Aswin S**
R103218082	**R103218129**	**R103218060**	**R103218027**

ABSTRACT

The C programming language has its own standard library string.h but still does not cover some of the functionalities that deal in dynamic strings.

A string can be described as an array of character data type, implying that they cannot be progressively expanded or diminished in size. To make an array larger, the information must be duplicated and cloned into another array, which is placed into a new block of memory. In the event where a string is altered, the string itself is not really changed, rather a new memory reference for the modified string is created. This would in turn lead to the loss of the memory space used to allocate the original string and leading to different performance issues.

Also, while looking into the definition of a C string does not contain the size of the memory allocated for that string therefore the user must remember this buffer size in some other variable and must check sizes on every buffer operation. As C language does not provide any automatic bounds checking and no automatic growth. This can be error-prone, and the result is the famous class of security vulnerability such as buffer overflows and other memory management problems. To rectify these issues, a new modified library is proposed which would operate on dynamic strings as well as improvise the existing functions of the standard string library in C.

TABLE OF CONTENTS

-

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

String manipulation is an important concept in computer science, and it is something that comes up very often in systems programming. The C programming language has its own standard library which has some excellent string manipulation facilities but there are several problems that are encountered while using the existing library functions.

Strings in C are defined as a stream of contiguous bytes, terminated by a byte with the value zero. The C standard library has many functions that deal with this type of string, but they suffer from one major problem. The definition of a C string does not contain the size of the memory allocated for that string. Thus, the user must remember this buffer size in some other variable. Obviously, this can be error-prone, and the result is the famous class of security vulnerability such as buffer overflows and format string attacks. [1]

One of the reasons behind vulnerability to buffer overflows are errors when copying and concatenating strings because the standard `strcpy()` and `strcat()` functions perform unbounded copy operations .There are several problems encountered when `strncpy()` and `strncat()` are used as safe versions of `strcpy()` and `strcat()`. Both functions deal with NULL-termination and the length parameter in different and non-intuitive ways that confuse even experienced programmers. They also provide no easy way to detect when truncation occurs. Finally, `strncpy()` zero-fills the remainder of the destination string, incurring a performance penalty. Of all these issues, the confusion caused by the length parameters and the related issue of NULL-termination are most important.

They can be eliminated by programmers through careful programming and by rigorous checking of array bounds. However, it is unrealistic to assume that all programmers will follow such a strict programming practice since it would decrease the benefits of convenience and performance, just as it is unrealistic to assume that they will not make any programming mistakes.

The Enhanced String Library includes replacement to standard string library functions as well as a number of additional useful routines in which we will try to rectify the above-stated issues and provide enhanced built-in functions to dynamically allocate memory to strings, free the used memory of the string that is copied to a new location, provide safer alternatives to functions such as `strncpy(),strncat(),snprintf(),` and other functions which are currently not present in the standard string library.

All these functions will be created in a way to prevent the problem of memory leak. This library can be used by developers as it can help them to save their time and will have the capability of more accurate memory management leading to fewer performance issues, more crisp code following the principle of DRY.

2. RELATED WORK

The difficulties of managing memory and efficient string manipulation in C are well known, several attempts at addressing these issues have been made. Nevertheless, few C and C++ libraries have succeeded in providing comprehensive solutions and none to our knowledge has addressed both memory leaks and string manipulation anomalies.

Authors proposed an alternative to strncpy() and strncat() in their research paper " strlcpy and strlcat Consistent, Safe, String Copy and Concatenation" that was needed, primarily to simplify the job of the programmer, but also to make code auditing easier. This is a minimalist, statically sized buffer approach that provides C string copying and concatenation with a different (and less error-prone) interface. [2]

The GNU C Library provides two useful functions that are widely used to get things done in C without buffer overflows. When using C (not C++), a dynamic memory allocation approach can be used, the asprintf and vasprintf functions are an especially useful way to solve the problem of safety. The asprintf() and vasprintf() functions are analogues of sprintf(3) and vsprintf(3), except that they automatically allocate a new C string and return a pointer to that string. [3]

One problem with them is that they are not actually standard (they are not in C11). r. Another problem is that their wide use can easily lead to memory leaks; as with any C function that allocates memory, you must manually deallocate the allocated memory.

The developers of Lucent Technologies have developed Libsafe, a wrapper of several library functions known to be vulnerable to security attacks. This wrapper is a simple dynamically loaded library that contains modified versions of C library functions such as strcpy(3). These modified versions implement the original functionality, but in a manner that ensures that any buffer overflows are contained within the current stack frame. Their initial performance analysis suggests that this library's overhead is very small. [4]

Libsafe is a useful mechanism to support defence-in-depth but it does not really prevent buffer overflows. Libsafe only protects a small set of known functions with obvious buffer overflow issues. At the time of this writing, this list is significantly shorter than the list of functions in this book known to have this problem. It also will not protect against code you write yourself (e.g., in a while loop) that causes buffer overflows. Libsafe only protects against buffer overflows of the stack onto the return address; you can still overrun the heap or the other variables in that procedure's frame. LibSafe seems to assume that saved frame pointers are at the beginning of each stack frame. This is not always true. Compilers (such as gcc) can optimize away things, and in particular the option "-fomit-frame-pointer" removes the information that libsafe seems to need. Thus, libsafe may fail to work for some programs.

3. PROBLEM STATEMENT

Most programming languages have a string library that relieves programmers from writing their own string operations for every program. The C programming language also has its own standard library "string.h" which has some excellent string manipulation facilities but still does not cover some of the functionalities that deal in dynamic strings and it has many errors in it.

The first and the foremost problem was of **memory wastage**. To get a clear idea let us consider a scenario to know how memory wastage occurs in string manipulation. Consider a character string buff[] in which a string consists of 10 letters. For example, the string “Indrakshee” is read from the user. This would be declared as

Figure-1 Array declared for the string “Indrakshee”

“The memory allocated buff would be 11 bytes including the null character.”

As known earlier, a string can be described as an array of character data type, implying that they cannot be progressively expanded or diminished in size. Now consider the situation in which the user is required to alter buff[] with another string .In the event of string alteration using the library “string.h”, we can understand that the string itself is not changed rather a new memory reference for the modified string is created. For better understanding, let's say that the user altered the buff[] with a string of 11 bytes.

Now think of a situation in which the user does the alteration for 10,000 times.

Total bytes=10000*11

=110,000 bytes would be wasted

On proceeding further, it leads us to the next major issue in string.h library called as the **buffer overflow**. To understand the concept of buffer overflow, let us take the help of the following code snippet:

```
char name[5]="";

prinf("enter your name");

scanf("%s",name);
```

Considering a situation in which the user enters a name which contains more character than the space allotted. For example, AswinSarat

Figure-2 Array declared for the string "AswinSarat"

"What happened above is that the string name[] that the user entered has corrupted the data present in the adjacent memory location(here in the location 1031)"

Wrapping up we found that the C programming language neither provides any automatic bounds checking nor does it provide any automatic growth. This can also lead to some other common errors such as **unbounded string copies** and **string truncation errors**

When the strcpy() function from the library string.h is used we get the below shown error:

```
char a[16];

strcpy(a,"0123456789abcdef");
```

Due to the lack of boundary checking, the null character gets declared beyond the scope of the array as shown in the figure below:

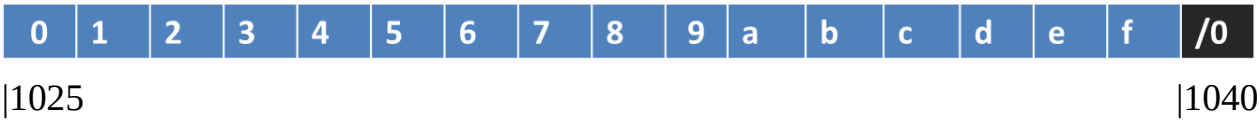


Figure-3 Array declaration of the string "0123456789abcdef"

"Therefore, the allocated memory space for string a[]ends at 1040 leading to the null terminator not being declared at all."

This leads to another common error of **null termination error.**

4. OBJECTIVE

Our objective is to create a static string library which will:

I. Provide built-in function to dynamically allocate memory to strings.

II. Fix the existing issues in the built-in functions of string.h library thus creating the improvised versions of them.

III. Create functions that are not prone to the problem of memory leak.

IV. Create functions that do not perform unbounded string operations.

V. Create methods that are currently not present in string.h.

5. DESIGN

To build the Enhanced String Library (ESL), Agile software development methodology is deployed as shown in Fig 1. Agile software development refers to an approach to project management that prioritizes incremental, feedback-driven changes into software development.

With the help of Agile methodology and by using time-boxed, fixed schedule sprints/Iterations of 2-3 weeks, we were able to add new functions in the Enhanced string library frequently based on constant feedback. Using Agile in ESL gives the team constant opportunities to build, deliver, learn, and adjust.

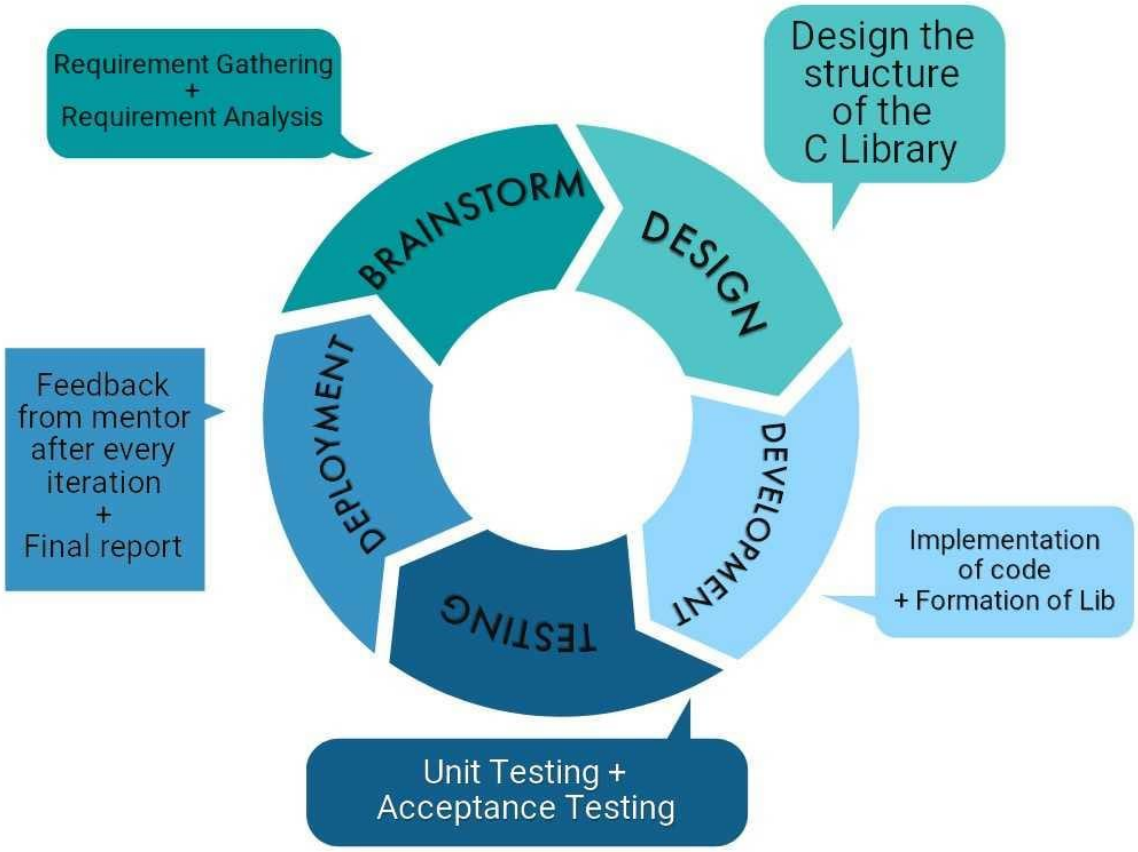


Figure-4 Agile Methodology for ESL

Several works related to our problem statement were analyzed by the team to find out possible shortcomings and succeeded in finding solutions to most of the issues that existed in the standard string library in C. But none to our knowledge addressed all the problems in a single library.
So, after the requirement analysis, we designed the basic structure of ESL by creating three main components:

• A **C source file(*.c)**- This file contains the definitions i.e., the body of all the functions that are included in our library. After the final source file is created, an object file for the same is created.

• A **C header file(*.h)**-This file is basically an interface of the ESL library , it contains the definitions or references of the functions included in the library.

• A **test file**- This file shows the actual implementation of the ESL. It contains the test code for the implementation of the library functions. In the last step of our library creation, we link this test file to the object file created earlier to ensure the proper functioning of the library. Fig 5 below shows the step wise design of the Enhanced String Library.

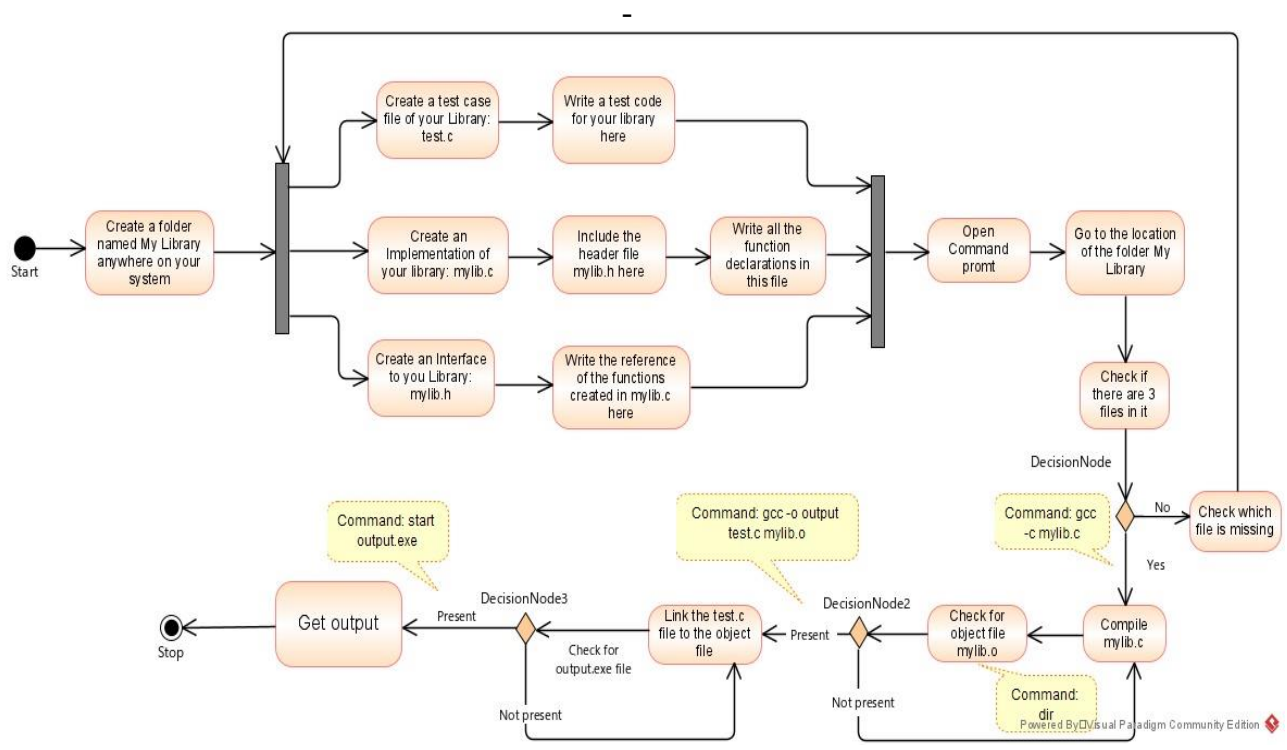


Figure-5 Step by step creation and usage of a C library

After the designing phase, Documentation of the project and making a prototype strengthened the base of the project. For quality assurance, we tested the functions in our library for numerous test cases across different platforms. At the last phase of each iteration, we will present our work to our mentor. We have worked on the feedback received from our mentor in the next iteration unless the final product was made as shown in the Fig 6 below.

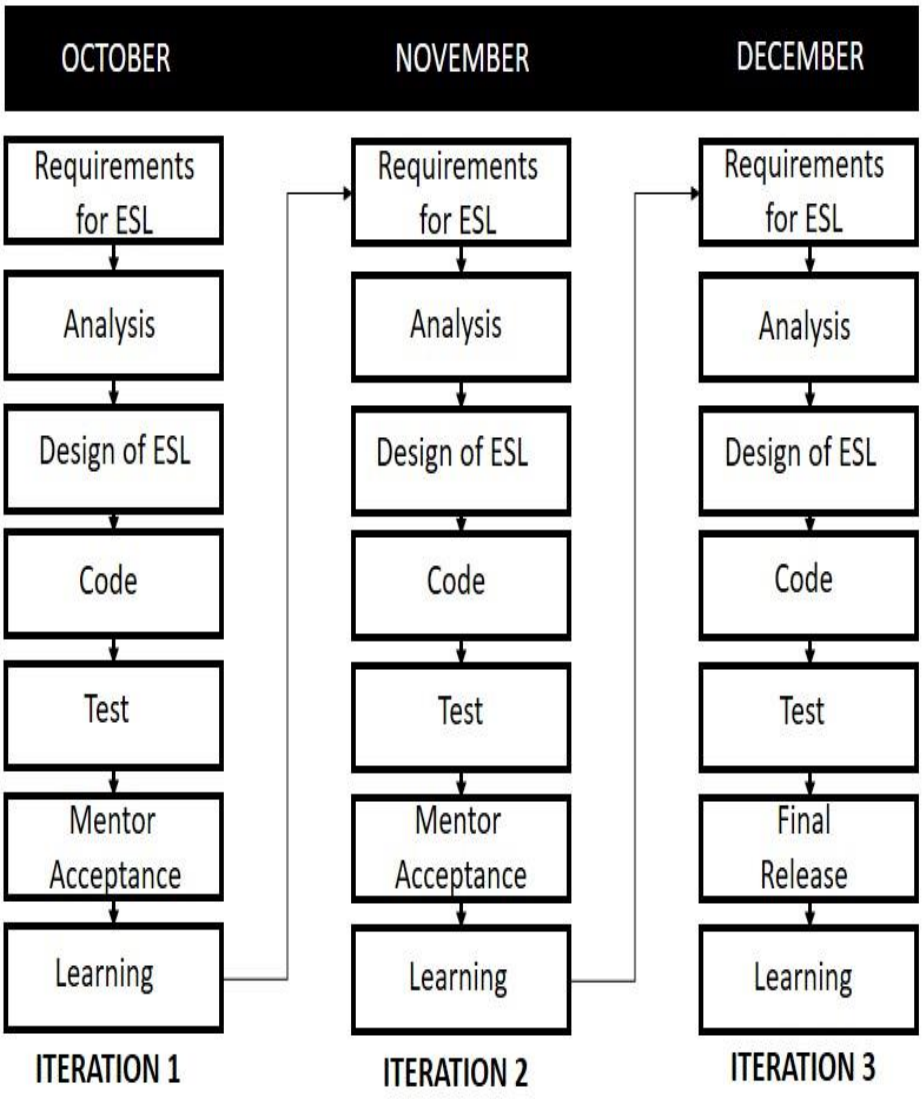


Figure-6 Iteration Cycle

6. IMPLEMENTATION

The Enhanced String Library is a static library that is joined to the main module of a program during the linking stage of compilation before creating the executable file. After a successful link of a static library to the main module of a program, the executable file will contain both the main program and the library. Below is the step-by-step compilation of .c and .h files.

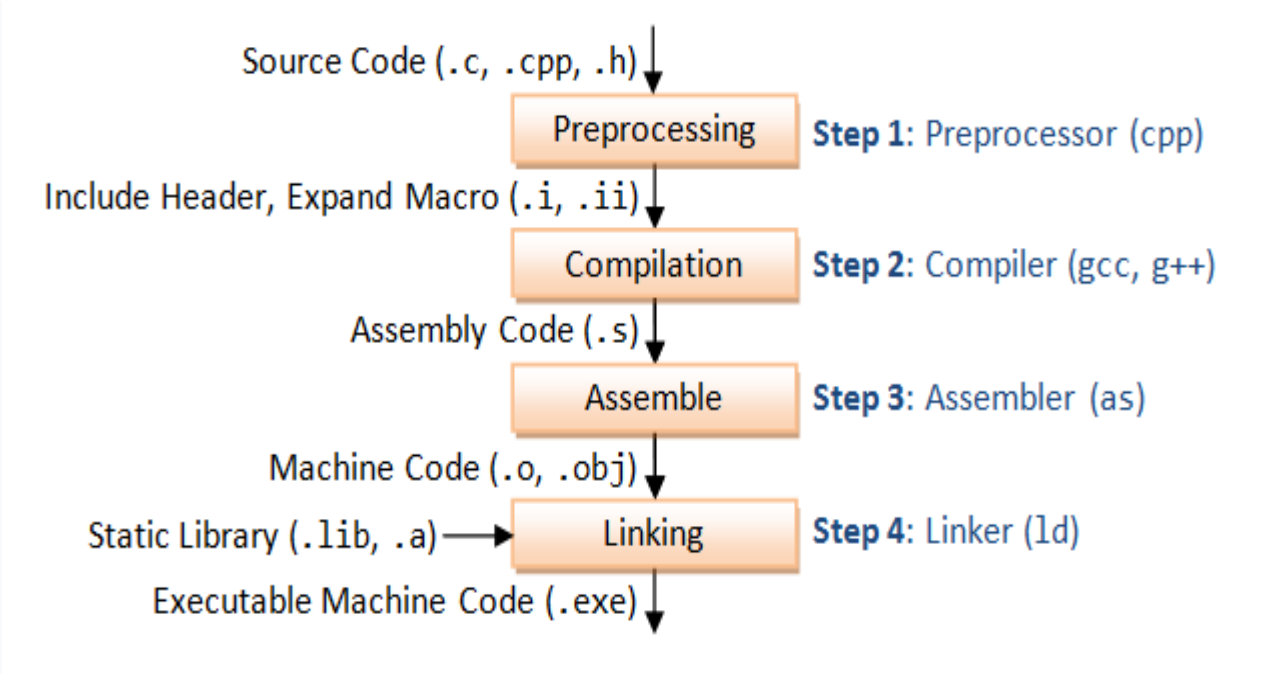
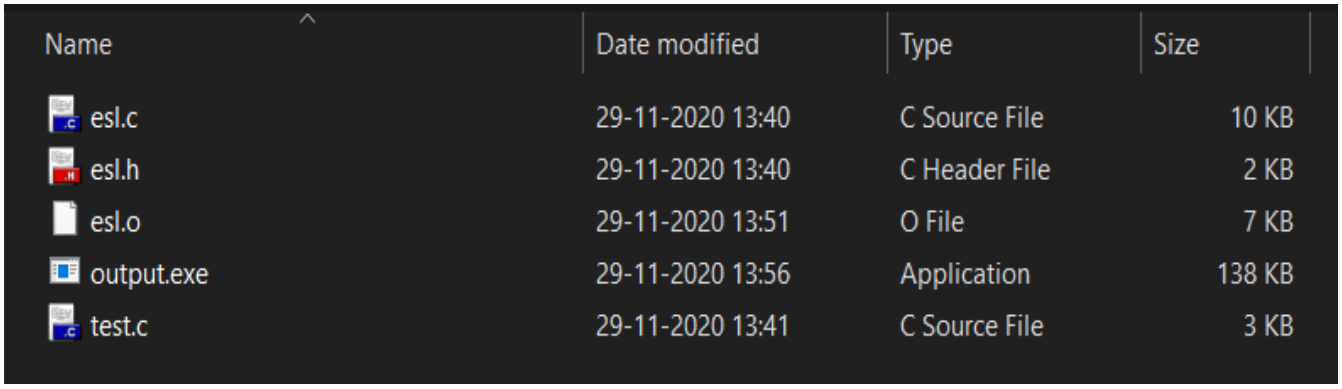


Figure-7 Compilation

6.1 EXECUTING THE LIBRARY

Total files necessary:



Name	Date modified	Type	Size
esl.c	29-11-2020 13:40	C Source File	10 KB
esl.h	29-11-2020 13:40	C Header File	2 KB
esl.o	29-11-2020 13:51	O File	7 KB
output.exe	29-11-2020 13:56	Application	138 KB
test.c	29-11-2020 13:41	C Source File	3 KB

Figure-8 Total required files

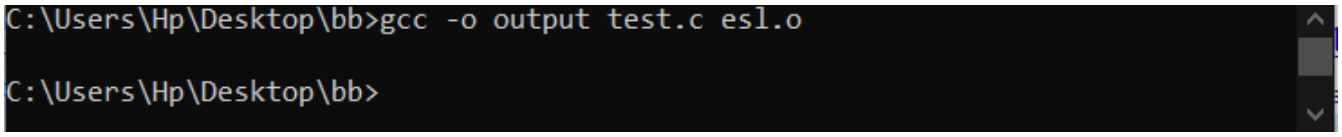
STEP 1: Compiling “esl.c” file

```
C:\Users\Hp\Desktop\bb>gcc -c esl.c

C:\Users\Hp\Desktop\bb>
```

Figure-9 Compiling of “esl.c” file

STEP 2: Linking the test file “test.c” with the object file “esl.o” file



```
C:\Users\Hp\Desktop\bb>gcc -o output test.c esl.o

C:\Users\Hp\Desktop\bb>
```

Figure-10 Command to link test and object file

STEP 3: Run the executable file “output.exe”.

```
C:\Users\Hp\Desktop\bb>output
Enter your string
```

Figure-11 Command for output

6.2 DESCRIBING THE SOURCE FILE AND THE FUNCTIONS

C provides no automatic bounds checking and no automatic growth. The programmer must always know the size of the buffer he is working with and must check sizes on every buffer operation. This is tedious and error prone. This library presents an idea that might make C -string handling a little easier.

The solution to the string problem involves special string structures and a library of functions. The structures hold buffer sizes, and the library functions grow character arrays whenever necessary.

```
struct string_data {
    str_size alloc; // stores the number of bytes allocated
    str_size length;
    char buff[];
};
```

Figure-12 Code snippet of ESL struct

alloc is the allocated size of the buffer, and **length** is the amount of the buffer currently used. Every time **data** is written to, **alloc** and **length** are checked. If the operation would cause the buffer to overrun, the buffer is grown using **realloc** first.

Following are the predefined functions that we have decided to include in our Library.

6.2.1 `string_alloc(…)`

- This function allocates the memory to the string structure at runtime using malloc.

6.2.2 `string_realloc(…)`

- This function helps the string to grow or shrink according to the given length and reallocates the memory to the string structure that was previously allocated by malloc.

6.2.3 `string_get_data(…)`

- This function returns the pointer to character array containing the data from the string structure.

6.2.4 `string_create(…)`

- This function accepts the string from the user, allocates the memory equivalent to the length of that string during runtime ensuring the string is null terminated and returns the dynamically created string.

6.2.5 `string_has_space(…)`

- This function returns True or False based on if there is any free space left.

6.2.6 `string_size(…)`

- This function returns the size of string.

6.2.7 `string_add_char(…)`

- Appends a character to the end of the string.

6.2.8 `string_Remove(…)`

- This function removes a substring of given length from the original string.

6.2.9 `string_free(…)`

- Frees up the space allocated by malloc/realloc.

6.2.10 `string_insert(…)`

- Inserts a substring or character at a desired index.

6.2.11 `string_Replace(…)`

- This function replaces any part of the string with the given character/string which may be longer/shorter than the part of the string already existing.

6.2.12 `string_cat(…)`

- The strcat() function concatenates the destination string and the source string, and the result is stored in the destination string.

6.2.13 `string_ncat(…)`

- The string_ncat() function appends the NULL-terminated string src to the end of dst. It will append at most size − strlen(dst) − 1 bytes, NULL-terminating the result.

6.2.14 `string_copy(…)`

- This function copies the string pointed by source (including the null character) to the destination.

6.2.15 `string_ncopy(…)`

- The function copies up to n characters from the string pointed to, by src to dest. In a case where the length of src is less than that of n, the remainder of dest will be padded with null bytes. Unlike Strncpy(…) of string.h, our function, string_ncopy(…) take the full size of the buffer (not just the length) and guarantee to NULL-terminate the result (as long as size is larger than 0).

6.2.16 `string_vsprintf(…)`

- The **string_vsprintf function** writes formatted output to an object pointed to by s using arg as the variable argument list.Unlike the vsprintf, this allocate a string large enough to hold the output including the terminating null byte, and return a pointer to it via the first argument. This pointer should be passed to free() to release the allocated storage when it is no longer needed.

6.2.17 `string_sprintf(…)`

- The **string_sprintf()** works just like printf() but instead of sending output to console it returns the formatted string. The first argument to string_sprintf() function is a pointer to the target string. The rest of the arguments are the same as for printf() function. The functions string_sprintf() are analogs of sprintf(), except that they allocate a string large enough to hold the output including the terminating null byte ('\0'), And return a pointer to it via the first argument. This pointer should be passed to free() to release the allocated storage when it is no longer needed.

6.2.18 `string_sep`

- This function is used to split a string into a series of tokens based on a particular substring

6.3 OUTPUT SCREEN

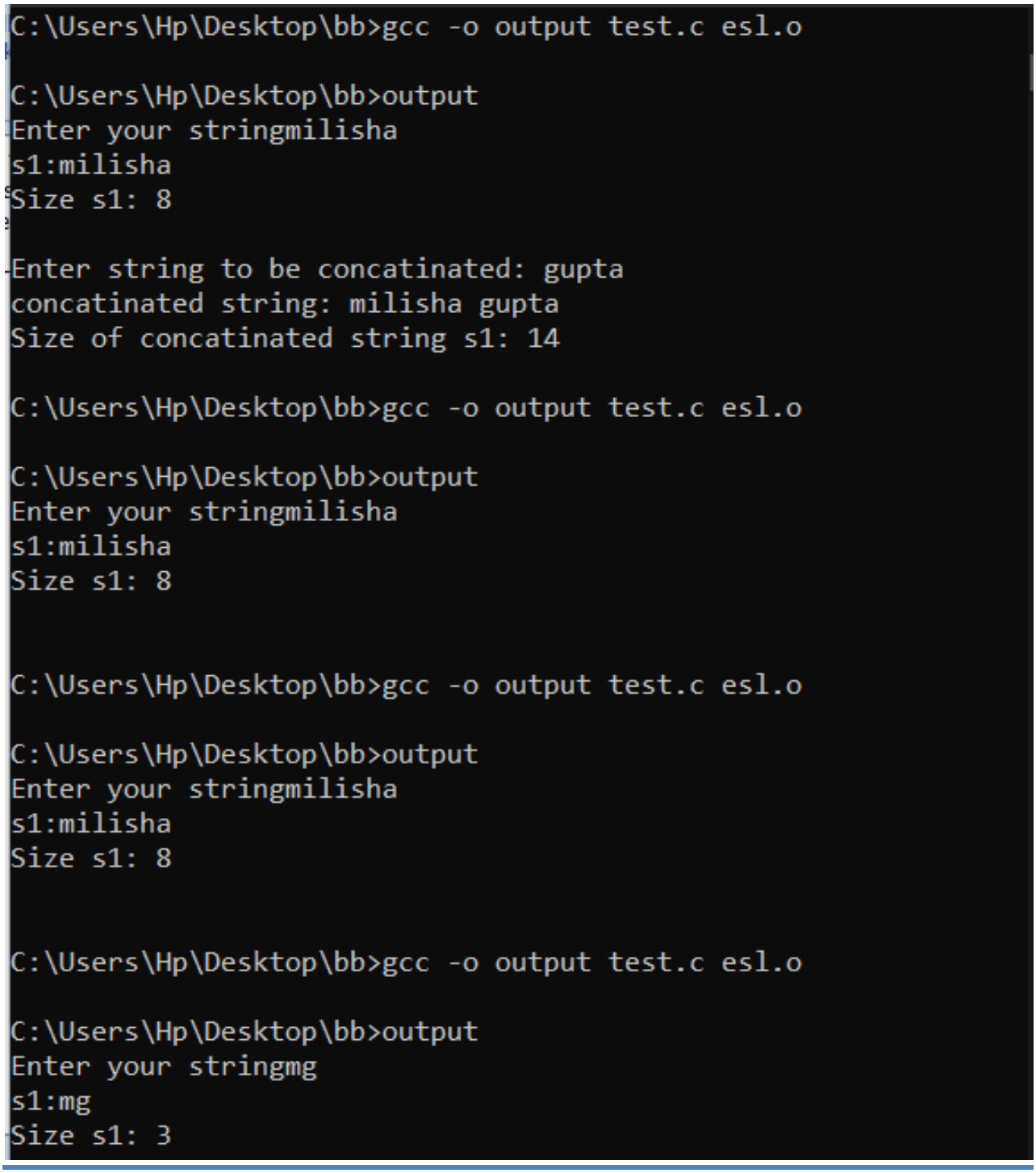


Figure-13 Output Screen 1

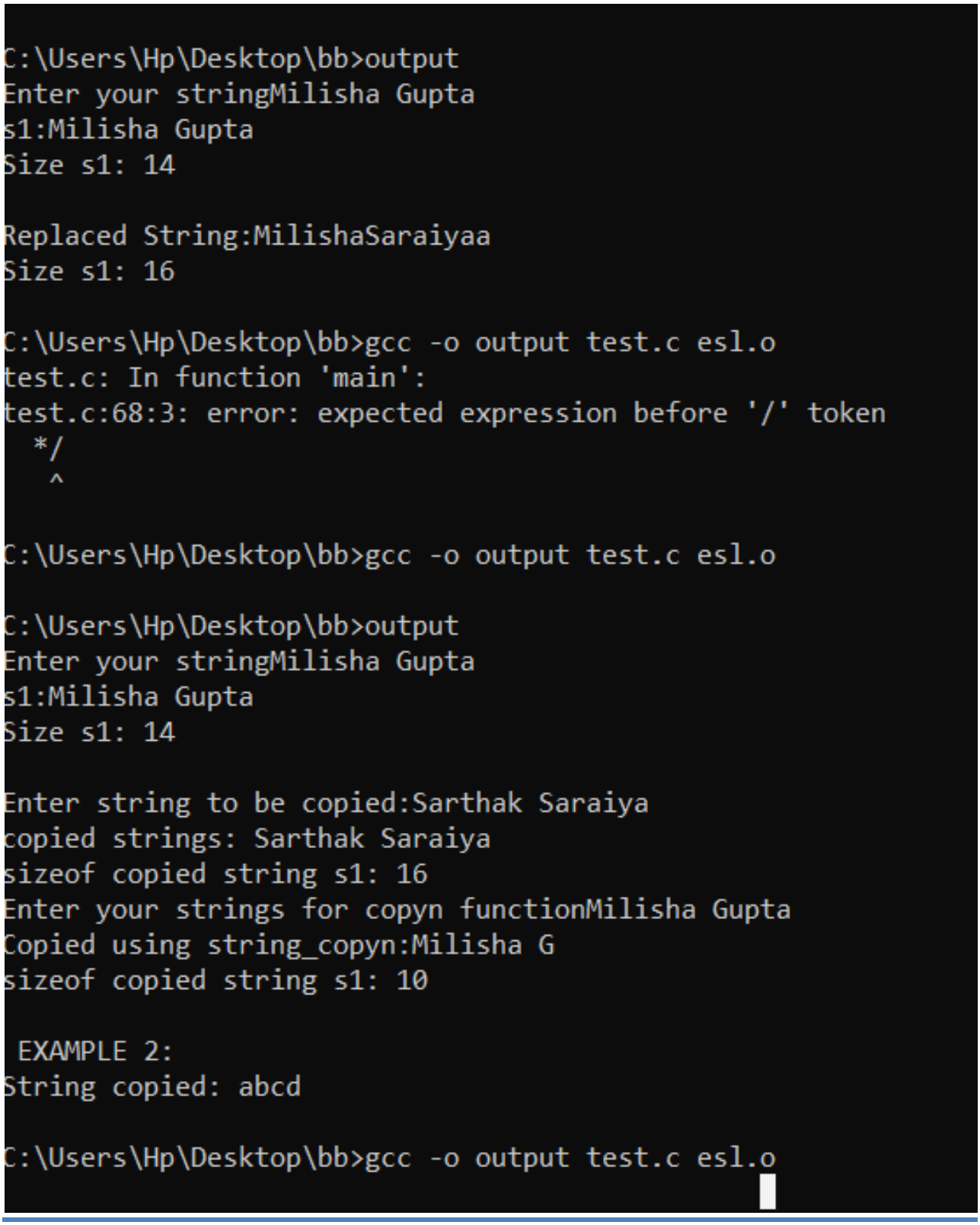


Figure-14 Output Screen 2

```
C:\Users\Hp\Desktop\bb>gcc -o output test.c esl.o

C:\Users\Hp\Desktop\bb>output
Enter your stringMilisha
s1:Milisha
Size s1: 8

Enter your strings for catn functionGupta
Concatinated using string_ncat:MilishaGu
sizeof concat string s1: 10

C:\Users\Hp\Desktop\bb>gcc -o output test.c esl.o

C:\Users\Hp\Desktop\bb>output

C:\Users\Hp\Desktop\bb>gcc -o output test.c esl.o

C:\Users\Hp\Desktop\bb>output
s1 = 100
s2 = 300.000000
this is a string
16

C:\Users\Hp\Desktop\bb>gcc -o output test.c esl.o

C:\Users\Hp\Desktop\bb>output
foo foo foo
C:\Users\Hp\Desktop\bb>
```

Figure-15 Output Screen 3

6.4 RESULT ANALYSIS

Functions designed specifically for the implementation of Dynamic strings.

```
string_alloc(…)
string_realloc(…)
string_get_data(…)
string_create(…)
string_has_space(…)
string_size(…)
string_free(…)
string_remove(…)
```

Functions designed which were currently not existing in string.h

```
string_replace(…)
string_insert(…)
```

Improvisation of functions that were present in the string.h.

strncat()	string_ncat(…)
strcat()	string_cat(…)
strncpy()	string_ncopy(…)
strcpy()	string_copy(…)
vsprintf()	string_vsprintf(…)
sprintf()	string_sprintf(…)
strtok()	string_sep(…)

strcpy() vs string_copy()

To prevent buffer overflow, in `string_copy()` size of destination string is checked, and memory is reallocated according to the string to be copied i.e. source string. In `strcpy()` there is no bounds checking thus results in overwriting the memory cells which are not allocated to the destination string.

Condition	Strcpy()		String_copy()	
	Memory Wasted	Chances of Bufferoverflow	Memory Wasted	Chances of Bufferoverflow
Sizeof(src)=sizeof(dst)	0	No	0	No
Sizeof(src)<sizeof(dst)	sizeof(dst)-sizeof(src)	No	0	No
Sizeof(src)>sizeof(dst)	0	Yes	0	No

Table 1:Comparision of strcpy() and string_copy() in terms of memory and safety.

strncpy() vs string_ncopy()

string_ncopy() avoids buffer overruns and ensures that the output string is null terminated. Unlike strncpy(), that sometimes fail to terminate destination string, it guarantees null termination and memory is reallocated according to the string to be copied if it is less than the given length. string_ncopy() return value facilitate the detection of truncation—*if* the programmer checks the return values.

Condition	Strncpy()			String_ncopy()		
	Memory Wasted	String Truncation	Chances of Buffer overflow	Memory Wasted	String Truncation	Chances of Buffer overflow
Strlen(src)=strlen(dst)	No	No	yes	No	Yes	No
Strlen(src)>strlen(dst)	Yes	No	Yes	No	Yes	No
Strlen(src)<strlen(dst)	No	Yes	No	No	yes	No

Table 2: Comparision of strncpy() and string_ncopy() in terms of memory,truncation and safety.

strcat() vs string_cat()

To prevent buffer overflow, in string_cat() size of destination string is checked, and memory is reallocated according to the length of source + destination string. In strcat() there is no bounds checking thus results in overwriting the memory cells which are not allocated to the destination string.

Condition	Strcat()		String_ncat()	
	Memory Wasted	Chances of Bufferoverflow	Memory Wasted	Chances of Bufferoverflow
Sizeof(new_dst)= sizeof(dst+src)	No	No	No	No
Sizeof(new_dst)< sizeof(dst+src)	Yes	No	No	No
Sizeof(new_dst)> sizeof(dst+src)	No	Yes	Yes	No

Table 3:Comparision of strcat() and string_cat() in terms of memory and safety.

strncat() vs string_ncat()

string_ncat() avoids buffer overruns and ensures that the output string is null terminated. Unlike strncat() ,that sometimes fail to terminate destination string, it guarantees null termination and memory is reallocated if the length of source + destination string it is less than the given length.. string_ncat() return value facilitate the detection of truncation—if the programmer checks the return values.

Condition	Strncat()			String_ncat()		
	Memory Wasted	String Truncation	Chances of Buffer overflow	Memory Wasted	String Truncation	Chances of Buffer overflow
Strlen(new_dst)= strlen(dst+src)	No	No	yes	No	Yes	No
Sizeof(new_dst)< sizeof(dst+src)	No	No	Yes	No	Yes	No
Sizeof(new_dst)> sizeof(dst+src)	Yes	Yes	No	No	yes	No

Table 4: Comparision of strncat() and string_ncat() in terms of memory,truncation and safety.

vsprintf () vs string_vsprintf()

`string_vsprintf()` works exactly like `vsprintf()` except that they allocate a string large enough to hold the output including the terminating null byte and return a pointer to it via the first argument. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed.

vprintf() VS string_sprintf()

`string_sprintf()` works exactly like `sprintf()` except that they allocate a string large enough to hold the output including the terminating null byte and return a pointer to it via the first argument. This pointer should be passed to free(3) to release the allocated storage when it is no longer needed.

strtok() VS_string_sep()

`strtok()` breaks string `str` into a series of tokens using a delimiter string**.** Unlike the `strtok()` function, the delimeter can be of more than 1 character.

7. CONCLUSION

This report has discussed the development of an enhanced C string library to overcome the defects found in the standard C string library “string.h”. The objectives of this project were to create a static string library which will provide built-in functions to dynamically allocate memory to strings, will create improved functions to fix the problems of memory leak, unbounded string copies and other errors in string.h. All the objectives mentioned in the project were successfully met and implemented as well. By creating functions to dynamically allocate memory to string in this project, we were able to solve the problem of memory wastage and efficiently use the memory for our benefit.

Also, by including bound checking in most of the functions used in this project, many common errors were successfully rectified. This project brings to our notice the numerous errors present in the standard C string library string.h and various ways to rectify these errors and use the memory more efficiently. Further, there is future scope of inclusion of other string functions such as that of pattern matching to make this library more useful. The library created in this project can be used by developers as it can help them to save their time and will have the capability of more accurate memory management leading to fewer performance issues and more crisp code following the principle of DRY.

8. APPENDIX I PROJECT CODE

The GitHub link of our ESL project

https://github.com/indrakshee18/Minor_1_BAO_V_String

ESL.C

```
#include "esl.h"
#include <stdio.h>
#include <string.h>
#include <stdarg.h>

typedef struct string_datastring_data;

struct string_data {
      str_sizealloc; // stores the number of bytes allocated
      str_size length;
      char buff[];
};

string_data* string_alloc(str_size length)
{
      string_data*   s_data   =   malloc(sizeof(string_data)   +   (length   +
1)*sizeof(char)); // plus 1 for null terminator
      if(s_data == NULL)
printf("Couldn't allocate memory!") ;
 else {
      s_data->alloc = length+1;
s_data->length = length;}
      return s_data;
 }
string_data* string_realloc(string_data* s_data,str_size length)
 {
      s_data = realloc(s_data,sizeof(string_data) + length + 1); // plus 1 for
null terminator
      if(s_data == NULL)
printf("Couldn't allocate memory!") ;
 else {
      s_data->alloc = length+1;
s_data->length = length;}
      return s_data;
        }

string_data* string_get_data(string s) {
      return (string_data*)&s[-sizeof(string_data)];
}
string string_create(const char* str) {
      string_data* s_data = NULL;

      if (str != NULL) {
```

```
            str_sizestr_length = strlen(str);
            s_data = string_alloc(str_length);
            memcpy(&s_data->buff, str, str_length);
            s_data->buff[str_length] = '\0';
      } else {
            s_data = string_alloc(0); // will only allocate enough for a null
terminator
            s_data->buff[0] = '\0';
      }

      return s_data->buff;
}

bool string_has_space(string_data* s_data) {
      // allocate based on s_data->length + 1 to account for the null
terminator
      return s_data->alloc - (s_data->length + 1) > 0;
}

void string_add_char(string* s, char c) {
      string_data* s_data = string_get_data(*s);

      if (!string_has_space(s_data)) {
            str_sizenew_alloc = s_data->alloc * 2;
            s_data = realloc(s_data, sizeof(string_data) + new_alloc);
            if(s_data == NULL)
printf("Couldn't allocate memory!") ;
 else {
      s_data->alloc = new_alloc;}
      }
      s_data->buff[s_data->length++] = c;
      s_data->buff[s_data->length] = '\0'; // add the new null terminator

      *s = s_data->buff;
}

void string_add(string* s, const char* str) {
      string_data* s_data = string_get_data(*s);
      str_sizestr_length = strlen(str);
      str_sizenew_length = s_data->length + str_length;
      str_sizeoldL=s_data->length;

      // make sure there is enough room for new characters and null terminator
      if (s_data->alloc<= new_length + 1) {
            string_realloc( s_data,new_length);
      }
      // copy str chars
      memcpy(&s_data->buff[oldL], str, str_length);

      s_data->buff[new_length] = '\0'; // add new null terminator
      *s = s_data->buff;
}

void string_insert(string* s, str_size pos, const char* str) {
      string_data* s_data = string_get_data(*s);
```

```
	str_sizestr_length = strlen(str);

	str_sizenew_length = s_data->length + str_length;

	// make sure there is enough room for new characters and null terminator
	if (s_data->alloc<= new_length + 1) {
		string_data* new_s_data = string_alloc(new_length);

		memcpy(new_s_data->buff,  s_data->buff,  pos);  //  copy  leading
characters
		memcpy(&new_s_data->buff[pos+str_length],	&s_data->buff[pos],
s_data->length-pos); // copy trailing characters
		free(s_data);
		s_data = new_s_data;

	} else {
		memmove(&s_data->buff[pos], &s_data->buff[pos+str_length], s_data-
>length - pos); // move trailing characters
	}

	s_data->length = new_length;

	// copy str chars
	memcpy(&s_data->buff[pos], str, str_length);

	s_data->buff[new_length] = '\0'; // add new null terminator

	*s = s_data->buff;
}

void string_replace(string* s, str_size pos, str_sizelen, const char* str) {
	string_data* s_data = string_get_data(*s);

	str_sizestr_length = strlen(str);

	str_sizenew_length = s_data->length + str_length - len;

	// make sure there is enough room for new characters and null terminator
	if (s_data->alloc<= new_length + 1) {
		string_data* new_s_data = string_alloc(new_length);
		memcpy(new_s_data->buff,  s_data->buff,  pos);  //  copy  leading
characters
		memcpy(&new_s_data->buff[pos+str_length],  &s_data->buff[pos+len],
s_data->length-pos-len); // copy trailing characters
		free(s_data);
		s_data = new_s_data;

	} else {
		memmove(&s_data->buff[pos+str_length],	&s_data->buff[pos+len],
s_data->length-pos-len); // move trailing characters
	}

	s_data->length = new_length;

	// copy str chars
	memcpy(&s_data->buff[pos], str, str_length);
```

```
      s_data->buff[new_length] = '\0'; // add new null terminator

      *s = s_data->buff;
}

void string_remove(string s, str_size pos, str_sizelen) {
      string_data* s_data = string_get_data(s);
      // anyone who puts in a bad index can face the consequences on their own
      memmove(&s_data->buff[pos],   &s_data->buff[pos+len],   s_data->length   -
pos);
      s_data->length -= len;
      s_data->buff[s_data->length] = '\0';
}

void string_free(string s) {
      free(string_get_data(s));
}

str_sizestring_len(string s) {
      return ((str_size*)s)[-1];//to exclude the null character
}
str_sizestring_size(string s) {
      return ((str_size*)s)[-2];//to exclude the null character
}

str_sizestring_get_alloc(string s) {
      return ((str_size*)s)[-2];
}


str_sizestring_ncat(string* dst , const char* src , str_sizemaxlen)
{
      string_data* s_data=string_get_data(*dst) ;
      str_sizesrc_length = strlen(src);
      if (src_length+s_data->length+1 <maxlen)
            maxlen=src_length+s_data->length+1 ;
      string_realloc( s_data,maxlen-1);
      char *d = s_data->buff;
      const char *s = src;
      str_size n = maxlen;
      str_sizedlen;

      /* Find the end of dst and adjust bytes left but don't go past end */
      while (n-- != 0 && *d != '\0')
            d++;
      dlen = d - s_data->buff;
      n = maxlen - dlen;

      if (n == 0)
            return(dlen + strlen(s));
      while (*s != '\0') {
            if (n != 1) {
                  *d++ = *s;
                  n--;
            }
```

```
            s++;
      }
      *d = '\0';
      *dst=s_data->buff;
      return(dlen + (s - src));     /* count does not include NUL */
}

str_sizestring_ncopy(string* dst , const char* src , str_sizemaxlen)
{
      string_data* s_data=string_get_data(*dst) ;
      str_sizesrc_length = strlen(src);
      if (src_length+1 <maxlen)
            maxlen=src_length+1;
      string_realloc( s_data,maxlen-1);
      char *d = s_data->buff;
      const char *s = src;
      size_t n = maxlen;
      /* Copy as many bytes as will fit */
      if (n != 0) {
            while (--n != 0) {
                  if ((*d++ = *s++) == '\0')
                        break;
            }
  }
      /* Not enough room in dst, add NUL and traverse rest of src */
      if (n == 0) {
            if (maxlen != 0)
                  *d = '\0';        /* NUL-terminate dst */
            while (*s++)
                  ;
      }
      *dst=s_data->buff;
      return(s - src - 1);    /* count does not include NUL */
}

void string_copy(string* s, const char* str) {
      string_data* s_data = string_get_data(*s);

      str_sizestr_length = strlen(str);
      //s_data->length=str_length;

      // make sure there is enough room for new characters and null terminator
      s_data     =     realloc(s_data,(     sizeof(string_data)+str_length));
//reallocating the space would not lead to bufferoverflow problems
      if(s_data == NULL)
printf("Couldn't allocate memory!") ;
    else {
            s_data->length = str_length;
            s_data->alloc = str_length + 1;

      // copy str
      memcpy(&s_data->buff, str, str_length);

      s_data->buff[str_length] = '\0'; // add new null terminator

      *s = s_data->buff;}
```

```
}

int string_vsprintf (char **str, const char *fmt, va_listargs) {
  int size = 0;
va_listtmpa;

  // copy
va_copy(tmpa, args);

  // apply variadic arguments to
  // sprintf with format to get size
  size = vsnprintf(NULL, 0, fmt, tmpa);

  // toss args
va_end(tmpa);

  // return -1 to be compliant if
  // size is less than 0
  if (size < 0) { return -1; }

  // alloc with size plus 1 for `\0'
  *str = (char *) malloc(size + 1);

  // return -1 to be compliant
  // if pointer is `NULL'
  if (NULL == *str) { return -1; }

  // format string with original
  // variadic arguments and set new size
  size = vsprintf(*str, fmt, args);
  return size;
}

int string_sprintf(char **str, const char *fmt, ...)
 {

  int size = 0;
va_listargs;

  // init variadic argumens
va_start(args, fmt);

  // format and get size
  size =string_vsprintf (str, fmt, args);

  // toss args
va_end(args);

  return size;
}


char **string_sep(const char *buf, const char *sep){
    int i = 0, j = 0, k = 0, l = 0, stringCount = 0;
    /* If separator > buffer, return NULL */
    while (buf[i]) i++;
```

```
    while (sep[j]) j++;
    if (j >i) return NULL;
    char **strings = malloc(0);
    /* While we're not at the end of the buffer */
    while (*(buf+k)){
        /* If the characters match, check to see if it is the separator */
        if (*(buf+k) == *(sep)){
            for (l = 0; l < j; l++){
                /* If it isn't the separator, break */
                if (*(buf+k+l) != *(sep+l)) break;
                /* If it is the separator and the separator isn't the
                    beginning, add buf to buf+k bytes as a string */
                if (l == j-1 && k != 0){
                    strings  =  realloc(strings,  (stringCount+1)*sizeof  (char
*));
                    strings[stringCount] = malloc(k+1);
memcpy(strings[stringCount++], buf, k);
buf += (j + k), k = -1;
                /* If it is the separator, but it's the beginning
                    of the string, skip it */
                } else if (l == j-1 && k == 0) {
buf += j, k = -1;
                }
            }
        }
        k++;
    }
    /* Add a string for the left over bytes if sep isn't the end */
    if (i != k && *(buf)){
        while (buf[l]) l++;
        strings = realloc(strings, (stringCount+1) * sizeof (char *));
        strings[stringCount] = malloc(l+1);
memcpy(strings[stringCount++], buf, l);
    }
    /* Append NULL to array of strings */
    strings = realloc(strings, (stringCount+1) * sizeof (char *));
    strings[stringCount] = malloc(sizeof NULL);
    strings[stringCount] = NULL;
    return strings;
}
```

ESL.H

```
#ifndef esl_h
#define esl_h
#include <stdlib.h>
#include <stdbool.h>

typedef char* string;
typedef size_tstr_size;
string string_create(const char* str);
void string_add_char(string* s, char c);
void string_add(string* s, const char* str);
void string_insert(string* s, str_size pos, const char* str);
void string_replace(string* s, str_size pos, str_sizelen, const char* str);
```

```
void string_remove(string s, str_size pos, str_sizelen);
void string_free(string s);
str_sizestring_size(string s);
str_sizestring_ncopy(string* dst , const char* src , str_sizemaxlen);
str_sizestring_get_alloc(string s);
str_sizestring_ncat(string* dst , const char* src , str_sizemaxlen);
void string_copy(string* s, const char* str);
int string_vsprintf (char **, const char *, va_list);
int string_sprintf (char **, const char *, ...);
char **string_sep(const char *buf, const char *sep);
#endif /* esl_h */
```

TEST.C

```
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include "esl.h"

int main(void)
{  /*
  char str1[]="";
  char str2[]="";
  char s0[]="";

printf("Enter your string");
  string s1 = string_create(gets(str1)); //creating the dynamic string
  //string can be inputed using gets/fgets
printf("s1:%s\n", s1);
printf("Size s1: %d\n\n", string_size(s1));//size of our string*/
/*//TESTING STRING_REPLACE FUNCTION
string_replace(&s1, 7, 5, "Saraiya");
printf("Replaced String:%s\n", s1);
printf("Size s1: %d\n", string_size(s1));*/
 /*
  //TESTING STRING_INSERT
string_insert(&s1, 3, "_Inserted_");
printf("s1:%s\n", s1);
printf("Size s1: %d\n", string_size(s1));
  *//*

  // TESTING STRING_CONCAT
printf("Enter string to be concatinated:");
  gets(s0 );
string_add(&s1,  s0);
printf("concatinated string: %s\n", s1);
printf("Size of concatinated string s1: %d\n", string_size(s1));
  */



  /*
   // TESTING STRING_COPY
```

```
printf("Enter string to be copied:");
  gets(s0 );
string_copy(&s1, s0);
printf("copied strings: %s\n", s1);
printf("sizeof copied string s1: %d\n",string_size(s1));



  //TESTING STRING_NCOPY
printf("Enter your strings for copyn function");
   gets(str1);
string_ncopy(&s1,str1,10);//alter function for memory sizeof
printf("Copied using string_copyn:%s\n", s1);
printf("sizeof copied string s1: %d\n",string_size(s1));
  *//*
  //example 2
printf("\n EXAMPLE 2: \n");
       string dst= string_create("00000");
    //char src[] = "abcdefghijk";
       char src[] = "abcd";

    int buffer_length = string_ncopy(&dst, src, string_size(dst));

    if (buffer_length>=string_size(dst)) {
printf ("String: %s too long: %d (%d expected)\n",src,buffer_length,string_size(dst )-
1);
    }

printf ("String copied: %s\n", dst);*/
/*
//TESTING STRING_NCAT
printf("Enter your strings for catn function");
  gets(str1);
string_ncat(&s1,str1,10);//alter function for memory sizeof
printf("Concatinated using string_ncat:%s\n", s1);
printf("sizeofconcat string s1: %d\n",string_size(s1));
    */


  //TESTING STRING_SPRINTF
/*
char* s1;
char* s2;
int x = 100;
float y = 300;
char *str = NULL;
char *fmt = "this is a %s";
int size = string_sprintf(&str, fmt, "string");
//convert integer and float values to strings.
string_sprintf(&s1, "%d", x);
string_sprintf(&s2, "%f", y);
printf("s1 = %s\n", s1);
printf("s2 = %s\n", s2);
//printing
```

```
printf("%s\n", str); // this is a string
printf("%d\n", size); // 16
*/
 /*
//TESTING STRING_SEP

       string s= string_create("foobarfoobarfoobar") ;
       string sep = string_create("bar") ;
    char** strings = string_sep(s, sep);

    while(*strings){
printf("%s ", *(strings));
        free(*(strings++));
    }

        return 0;
}*/
```

9. REFERENCES

[1] K. a. S. J. C. Lhee, "Buffer overflow and format string overflow vulnerabilities," 2003, pp. 423-460..

[2] T. &. R. T. Miller, "strlcpy and strlcat - Consistent, Safe, String Copy and Concatenation," in *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference, June 6-11, 1999*, Monterey, California, USA, 1999.

[3] M. Kerrisk, "asprintf(3) — Linux manual page," 19 09 2020. [Online]. Available: https://man7.org/linux/man-pages/man3/asprintf.3.html. [Accessed 10 10 2020].

[4] A. &. T. T. &. S. N. Baratloo, "Libsafe: Protecting Critical Elements of Stacks.," p. 11, 2001.