# Concatenating Two User Input Strings

Cooper Union

ECE-251

Professor Billoo

Ali Ghuman, Xue Ru Zhou, Husam Almanakly, Layth Yassin

March 2021

# Contents

# 1 Objective

The objective of this project is to leverage understanding of computer architecture to concatenate two strings. The strings must be within the specified range of 12 characters, otherwise an error is to be thrown with an error code of seven for the first string and eight for the second. If the two strings are according to the size constraints, they are to be concatenated and output to the terminal with an error code corresponding to the size of the combined string.

# 2 Architecture

## 2.1 Overall Flowchart Diagram

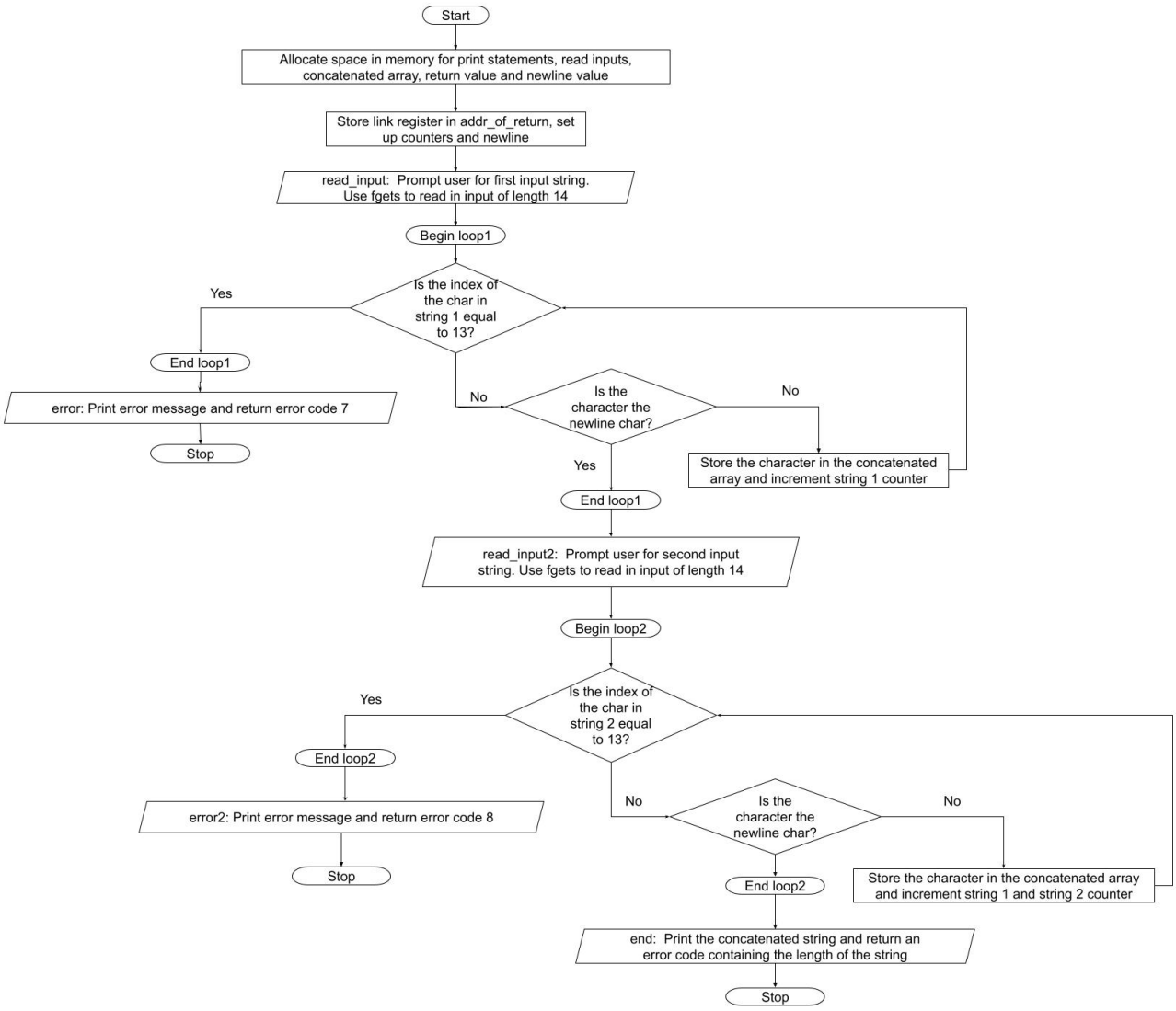


Figure 1: Project 1 Flowchart

## 2.2 Allocating memory

In the `.data` section, memory is allocated for the print statements (prompts and error messages), input strings, concatenated array, return value and the newline value. `.balign n` is a directive that aligns the given values to bytes that are multiples of `2`. For the print statements and input strings, we used `.balign 4` and declared the type of the value with `.asciz`. For arrays, each char is 1 byte we used `.balign 1` and used `.skip` the allocate the necessary space.

At the end of the code, the addresses of these memory locations are declared. We also make it known that printf and fgets exist by declaring `.global printf` and `.global fgets`.

## 2.3 Read User Input

From main, the link register is directed to `read_input` first. The message that prompts the user to enter a string is located at the address via label `addr_of_input`. This address is loaded into register R0. Then, the link register (which returns to `addr_of_return` as initialized in `.data`) is directed to `printf` to print the message.

After the user has entered a string of ASCII characters, the C function `fgets()` is used to read in the input. The value 14 is moved into register R1 to indicate the length fgets will read in. In addition, the parameter `stdin` is loaded into R2, using R3 to fetch the memory address first.

## 2.4 Conditional Loops: Counting the length of input strings

### 2.4.1 Input String 1

After `fgets()` reads the input and stores the string in R0, the link register is directed to `loop1`.

The value in R9 is the counter for the loop, set to 0 in `main`. The first part of the loop compared the value in R9 (or the index of the character in the input string) is equal to 13, using the instruction `cmp`. If the character is at the 13th index, this means that there are 14 or more ASCII characters (including the newline character that is appended after the user hits "enter"). It is important to keep in mind that indexing starts at 0, so therefore, the 13th index would contain the 14th character. This is why `fgets()` was told to read 14 characters into the input string so that it would get to the 13th index in the string array.

If the index is 13 and the program has not detected the newline character yet, the program breaks out of `loop1` and goes to `error`. If the index is not 13, the value of the character in the input string is compared to the newline character (which is stored in R6).

`ldrb r5, [r2,r9]` loads one byte at the index indicated by R9 of R2 (register holding the input string) into R5, a register to temporarily hold the character. In the next iteration, R5 will get overwritten.

If the character is not yet the newline character, the value in R5 is stored into R1, which holds the array of size 25 (24 for characters, 1 for newline, 1 for null) to hold the concatenated array. `add r9, r9, #1` increments the counter to continue to the next character in the string array and `b loop1` continues the loop. On the other hand, if the character is the newline character, the program will jump out of this loop and go to `read_input2`.

### 2.4.2 Input String 2

`read_input_2` does essentially the same thing as `read_input`. It prints a message that prompts the user to enter a second string. Then, the parameters of `fgets()` are loaded again into registers R1 and R2. After this, the program breaks to `loop2`.

While the counter was stored in R9 for string 1, the counter for string 2 is stored in R7, which was also set to 0 in `main`. Similarly, the value in R7 is compared to 13. If they are equal, the program jumps to `error2`. If not equal, the character is compared with the newline character. If it is the newline character, the program jumps out of the loop and goes to `end`. If it is not the newline yet, the value is stored into R1, the concatenated string array. Both R9 and R7 are incremented by adding 1 to the previous value. `b loop2` repeats the set of instructions until the conditions are met to jump out of the program.

## 2.5 Return Error Codes and Print Statements

`error` and `error_2` loads the memory location of the error message into R0 and then calling `bl printf` to print a statement, letting the user know that they have exceeded the maximum amount of characters. Afterwards, the value 7 and 8, for `error` and `error2`, respectively, are moved into R0 so that entering `echo $?` in the command line will return 7 or 8 error code. Lastly, the instruction `bx lr` exits the program.

## 2.6 End

If both strings are 12 or less characters long, the concatenated string is printed, using previously mentioned instructions (loading the address into R0 and using `bl printf`). Then, the value of R9 (the counter that incremented from the beginning of string 1 until the end of string 2) is moved into R0. This is the length of the concatenated string. After the program exits, entering `echo $?` in the command line will return the length of the concatenated string.

Note – Azeria Labs: Writing ARM Assemvbly[1] and the ECE251 lecture recordings[2] were used as references for learning ARM architecture and assembly syntax.

# 3 Design Considerations

## 3.1 Assumptions

The objective of the project mentioned that the character limit is 12 ASCII characters. This was assumed to include letters, numbers and spaces. However, because the objective did not specify whether or not the newline character (that results from the user pressing "enter" after their string) was included in the 12 character limit, it was not accounted for in the limit. If it was accounted for, the experience would not be very intuitive for the user because they would get the error messages for entering 11 visible ASCII characters.

## 3.2 C Functions

Throughout the program, existing C std library functions are leveraged to complete difficult tasks. To output messages prompting the user for input and to output the concatenated string, the std library function `printf()` is used. The `printf()` function outputs to the stdout (i.e., the terminal), which allows the user to see the printed outputs. Writing the code for a function that accomplishes the same tasks as `printf()` in assembly is complex and impractical. This is mainly because several cases and test conditions would have to be accounted for. In addition, writing a `printf()` function without the use of any libraries would require knowledge of machine code.

To complete the task of taking in user input from stdin (i.e., the keyboard), a few functions from the C std library were considered. The first function that was considered was `scanf()`. However, `scanf()` is not ideal for this program since it does not read white-space characters, which may be part of the string the user inputs. Another function that was considered was `getchar()`, but it was difficult to locate where the function returns the character, so it is not used in the program. The last function that was considered and was ultimately chosen is `fgets()`. The `fgets()` function reads white-space characters and allows for the specification of how many characters it should read. This solves the primary problem `scanf()` poses. Writing a function to take user input in assembly would be difficult for similar reasons as the `printf()` function.

# 4 Problems and Solutions

## 4.1 Bounds Argument for fgets()

One of the major issues encountered in this project was during the error check for words inputted with a length greater than 12 characters. This was accomplished while the user input was being stored in the output string character by character. The bounds on `fgets()` was set to 13 characters, which would allow the program to distinguish valid and invalid inputs. If the string read was 13 characters, that meant the user input was invalid.

Thus each character was compared to the null terminator character as well as the newline character to know if the end of the string had been reached. However, it was later discovered that the user input had to be cut off at 14 characters rather than 13, as the null terminator would count as the thirteenth character. The previous parameter would work correctly and classify strings less than or equal to 11 characters as valid, rather than 12. This was found through general testing and experimenting the with different parameters and conditionals that would check for the end of the string.

## 4.2 fgets() parameters

Another major concern was learning how to utilize the function `fgets()`, specifically how to pass stdin as the stream parameter. After the earlier consideration for why `scanf()` was not the appropriate function, the decision to learn how to use `fgets()` was made.

To understand how the function could be used in ARM assembly, a simple test program was written in C that simply called `fgets()`. The program passed as parameters, a character array, 14 as the input bound, and stdin as the given stream. The test program was then cross compiled into C using the -S parameter, which resulted in an ARM assembly output code. This was analyzed and ultimately a method of passing stdin and using `fgets()` correctly was deduced. A variable was named as "stdin" and this was then loaded into one of the registers to be passed as a parameter to the `fgets()` call.

## 4.3 Array Length

Additionally, there was an issue with the number of spaces of memory allocated for the concatenated array. Initially, the array was allocated 24 spots, as there would be a maximum of 24 characters inputted from the user to concatenate. However, the program crashed on specific inputs, when 12 characters were first inputted and then the bounds were overstepped on the second input, 13 characters for example. It was understood that the output array required 25 spaces of memory as there needed to be one slot for the last null terminator to be included.

# 5 Deliverables

## 5.1 README

To build and run the assembly code, use command "make all" while on the micro-lab server, and it should create an executable named a.out. To execute, run ./a.out from the current directory and enter the prompted input.

## 5.2 MakeFile

The Makefile automatically removes a pre-existing executable named "a.out" and builds the given source code. If no file named a.out exists, the remove command does nothing and the source code continues to build the source code normally.

```
# Makefile

all: a.out

a.out: proj1.S .clean
    arm-linux-gnueabi-gcc $< -o $@ -ggdb3 -static

.clean:
    rm -f a.out
```

# 6 Conclusion

Two strings were successfully concatenated, giving the appropriate error values as well. For strings of length greater than twelve, an error of seven or eight was given, whereas in the successful case, the appropriate value of the length was given. Deeper knowledge was attained concerning the workings of registers and C functions utilized in assembly. Lastly, by experimenting with various methods of reading input, debugging numerous segmentation faults, and finally arriving at success, understanding of computer architecture was heightened. There, however, remains much to be learned. Some questions that arise concern the specifics of how ARM employs C functions as well as how stdin is passed to such functions. It is not understood why stdin can be declared as it was and how ARM treats it as a memory address, or how the C function is able to take the argument and understand that stdin is what is meant by that specific address. Perhaps it is that a special address exists for stdin, stdout and stderr, and can be declared as was done. This is only speculation, however, and doubt remains to its true workings.

# 7 References

[1] *Writing ARM Assembly*. Azeria Labs, `https://azeria-labs.com/writing-arm-assembly-part-1/`

[2] Billoo, Mohammed. ECE251 Computer Architecture. 11 Feb. 2021. Class lecture, `https://tinyurl.com/3yz4bta5`