

**Programming in C**

**Lab 1 – Algorithm Development**

Joakim Bajoul Kakaei

2nd of September 2017

**A lab report in the course DT555A Programming in C**

Table of Contents

[1. Introduction 1](#_Toc459883776)

[2. Design 3](#_Toc459883777)

[3. Implementation and Test 8](#_Toc459883778)

[4. Results and discussion 1](#_Toc459883779)2

[5. References](#_Toc459883780) 13

# 1. Introduction

In this lab we are tasked with improving how we approach a problems as developers; rather than implementing a solution to a problem, we focus on the methods that should be considered before coding. This is mainly done through developing a (few) flowcharts, some pseudo code and finally test-cases to ensure you can walk through the design without breaking it.

Due to financial reasons I’ve decided to make my flowcharts on draw.io (part of google-drive) and most of my pseudo code is written in notepad++.

Test cases are only added on task 1 and task 4, as required in the task outline. But the test cases are more than robust enough to handle most edge cases (all that I created!), so I went for quality over quantity.

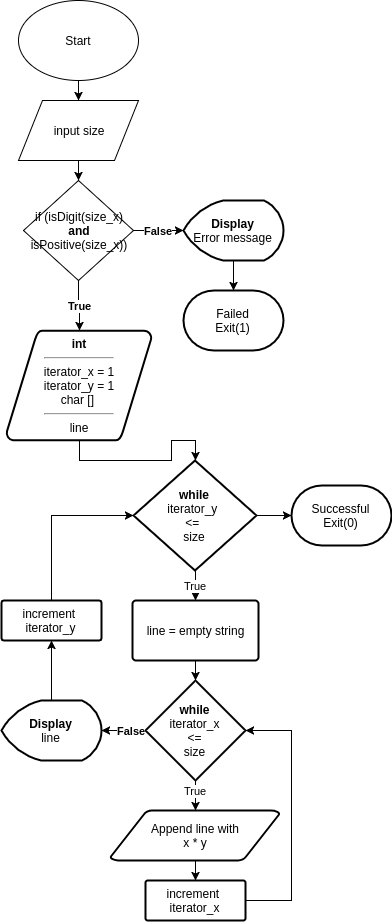
# 2. Design

**Task 1. Multiplication Table:**

Introduction:

The objects of this task seem clear enough; take a user’s input and display its possible multiplications in a table-structure like we used when we were children. However, even though the task might seem clear, I had to make one assumption; what would happen if a user enters a non-numeric string? In my case I’ve decided to check if the input is numerical and if not, the algorithm prints an error message and exits with status code 1 (failure).

Other than that the code executes as you would expect, it creates three variable, one to hold the size and an iterator for each axis on the graph (x, y). Then it spins through two loops, one to print each line, and one to iterate through the lines.

 Design and pseudo code:

1. The user is prompted for a *size*

2. Size is numeric and positive?

T: Go to step 3.

F: Display error message and exit with status 1.

3. Create two iterators for the y- and x-axis *y-iterator, x-iterator* and a character array *line*.

4. While *y-iterator* is lower or equal to *size*

T: Go to step **5**

F: End with successful status 0.

5. Set *line* to the empty string.

6. While the *x-iterator* is lower or equal to the size.

T:

\* Append the string with the product of the *y-iterator* and the current *x-iterator*.

\* Increment the *x-iterator*

F:

\* Display the *line*

\* Increment the *y-iterator*.

\* Go to step 4.

7. End with a successful status code

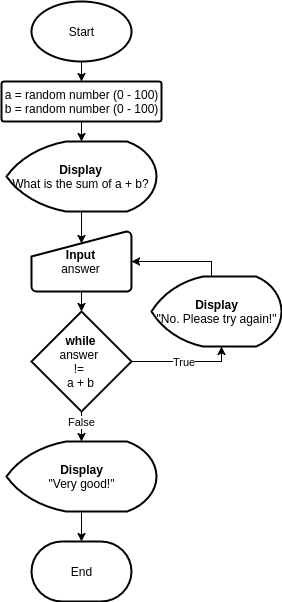
**Task 2. Computer Assisted Instruction (CAI)**

Introduction:

This task is far more straight-forward than the previous one, as no assumptions are required to be made in terms of the input.

The main point of this task is to understand how to **keep asking a question** until a correct answer has been provided, regardless of if the answer is numeric or not.

In my case I’ve decided to implement a while-loop that keeps looping until the answer is the same as a + b. The test-case is simply *while answer != (a + b)*.

 Design and pseudo code:

1. a = random number (0 - 100)

2. b = random number (0 - 100)

3. Display "What is the sum of a + b?

4. Input answer

5. While answer is not a + b

T:

\* Display "No. Please try again!"

\* Input answer

\* Return to loop conditional.

F:

\* Continue to step 6

6. Display "Very good!"

7. Exit with status code (0)

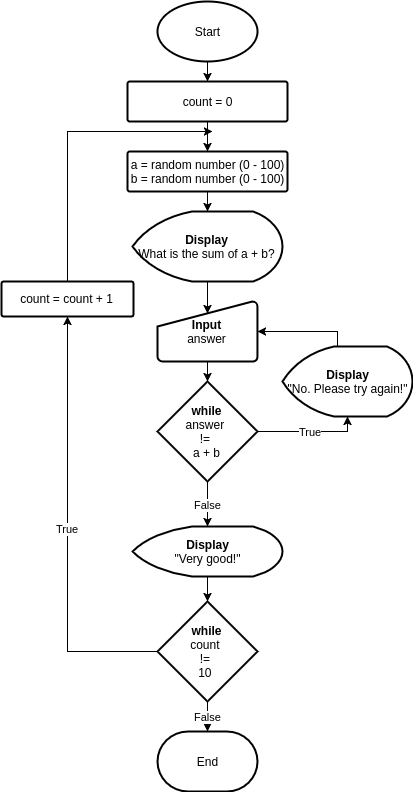
**Task 3b. Computer Assisted Instruction (CAI) – Part 2.**

Introduction:

This task’s solution will be almost identical to the previous task, with the obvious difference that we need create 10 instances of the previous solution.

The requirements are rather clear; create a question, prompt the user, check if the answer is correct. If not correct, we prompt the user again, if it is correct we check if the user has answered 10 questions correctly. If 10 correct answers haven’t been provided, we go back to the first step of creating a question. If the condition has been satisfied, we exit the program.

No assumptions are required from our side as the main design has been implemented in the previous task and the main purpose of this task is to extend that solution.



Design and pseudo code:

1. count = 0

2. a = random number (0 - 100)

3. b = random number (0 - 100)

4. Display "What is the sum of a + b?

5. Input answer

6. While answer is not a + b

T:

\* Display "No. Please try again!"

\* Input answer

\* Return to loop conditional.

F:

\* Continue to step 7

7. Display "Very good!"

8. While count is not 10 go back:

T:

\* Increment count by 1

\* Go to step 2

F:

\* Go to step 9

9. Exit with status code (0)

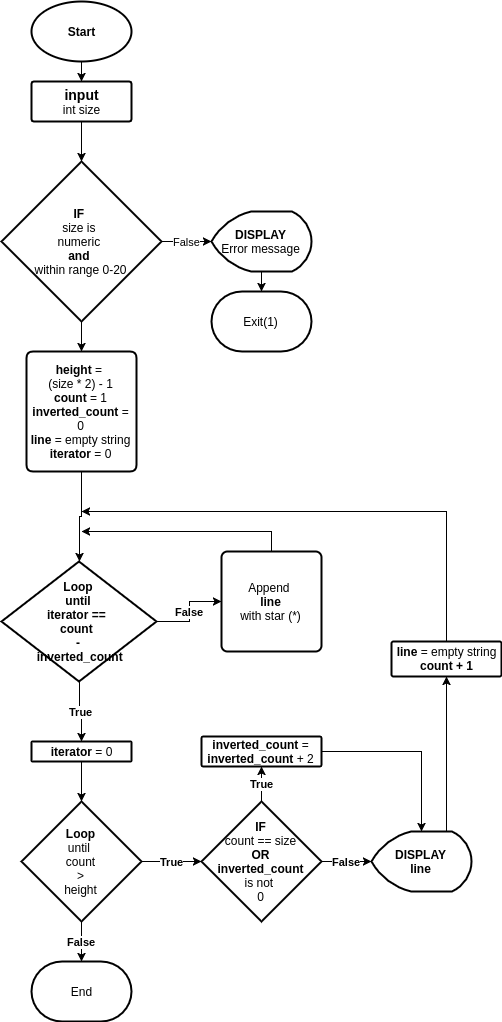
**Task 4b. Pyramid**

Introduction:

In this exercise we are tasked with creating a pyramid by using stars (\*) based on a **numeric** input size which serves as the top of the pyramid. For example, if we receive 5 as input then we are going to build 9 lines with *n* stars where *n* is an iterator lower or equal to *size times 2 and higher or equal to 1.*

The requirements also specify that the input needs to be within the range 5-20, but it doesn’t entail what will happen if the input is out of range. In my case I’ll assume that we simply terminate the program with an error message and an erroneous exit status (1).

I’ve also decided to only incorporate two loops in my program (one inner and one outer), and use an inverted count which increments for every loop where the count is higher than the size. This inverted count is then used to decrement the amount of times we append stars. Say we are using the example above, in most solution you will see three or four separate loops – one that handles the steps **up to 5** and a separate that **steps down from 4 to 1**. In my case I’ve got a single loop that when it hits the size, starts decrementing the width of the pyramid by the inverted count, which is incremented for each step after we have hit the top of the pyramid.

 So on step 5 we check if we have hit the top of the pyramid (which we have) and we then set inverted count to 2. For step 6 we then iterate over count (6) – inverted count (2) = 4. Any further loop will then increment inverted count with 2 so that on step 7 we have 4 and on 8 we have 6.

1. input size

2. if size is numeric and within range 0-20

T:

\* Go to stop 3

F:

\* Display Error message

\* Exit with erroneous status code (1)

3. set height = (size \* 2) - 1

4. set count = 1

5. set inverted\_count = 0

6. line = empty string

7. loop until count equals height

7a. if count equals size or inverted\_count is not 0.

T:

\* inverted\_count + 2

\* continue to 7b.

F:

\* continue to 7b.

7b. loop until iterator equals count - inverted\_count

T:

\* Continue to 7c

F:

\* Append line with "\*"

\* Continue to 7c.

7c. Display line

7d. set line to empty string

7e. iterator = 0

7f. count++

8. Exit with successful status (0)

# 3. Implementation and Test

Introduction:

All my test cases are made in the tripple-A (standard):

* **A**rrange – We set up the test with the variables (in this case only input)
* **A**ct – We step through the pseudo code step-by-step
* **A**ssert – We assert if the expected output is the same as the actual output.

In the case of pseudo code it’s a bit more difficult as we don’t have any real code to test. Instead I’m then stepping through each part of the pseudo to check if we in any case are breaking the code, or if something unexpected happens. This is rather verbose, but necessary.

The tests are divided by task and test cases. **According to the requirements of the tasks that I performed in Lab 1 we were only required to write test cases for task 1 and task 4**. I’ve tried to check all the corner / edge-cases for each of these designs to ensure that we have covered all the basics.

**Task 1:**

**Test case 1:**

**Input:** “Multiplication”

**Expected output:** Error message “Sorry, we only allow numeric input, please restart to try again!”.

**Actual pseudo:**

1. The user is prompted for a *size >> “Multiplication”*

2. Size is numeric? (NO)

3. Print error code

4. Exit with status 1.

**Test**: Successful.

**Test case 2:**

**Input:** 2

**Expected output:** A correctly formatted multiplication table.

**Actual pseudo:**

1. The user is prompted for a *size >> 2*

2. Size is numeric? (YES)

3. y-iterator = 1, x-iterator = 1, char line []

4. While *y-iterator* is lower or equal to *size (YES)*

5. line = ””

6. While the *x-iterator* is lower or equal to the size. (YES)

7. line = ”” + ” ” + (1 \* 1)

*8. x-iterator = 1 + 1*

*9. x-iterator(2) lower or equal to size(2)? (YES)*

10. line = ”1” + ” ” + (1 \* 2)

11. x-iteartor = 2 + 1

12. *x-iterator(3) lower or equal to size(2)*? (NO)

13. Display line

14. line = ””

15. y-iterator = 1 + 1

16. y-iterator(2) lower or equal to size(2)? (YES)

*17. x-iterator = 1 + 1*

*18. x-iterator(2) lower or equal to size(2)? (YES)*

19. line = ”1” + ” ” + (1 \* 2)

20. x-iteartor = 2 + 1

21. *x-iterator(3) lower or equal to size(2)*? (NO)

22. Display line

23. y-iteartor(3) lower or equal to size(2)? (NO)

24. Exit with successful status code.

**Test**: Successful.

**Task 4:**

**Test case 1:**

**Input:** 5

**Expected output:** The example pyramid

**Actual pseudo:**

1. input size » 5

2. if size is numeric and within range 5-20 (YES)

3. set height = (5 \* 2) – 1 = 9

4. set count = 1

5. set inverted\_count = 0

6. set line = empty string (“”)

7. set iterator = 0

7. loop until count (1) is higher than height (9)

8. loop until iterator (0) equals count (1)- inverted\_count (0) (NO)

9. Append line with (\*)

10. increment iterator (1)

11. is iterator(1) equal to count (1)? (YES)

12. display line (\*)

13. set line = empty string (“”)

14. increment count (2)

15. Go to step 7 until count is higher than height.

16. Exit with successful status (0)

**Test:** Successful

**Test case 2:**

**Input:** 1

**Expected output:** Error out of range

**Actual pseudo**:

1. **input** size >> 1

2. **if** size is numeric (YES) and within range 5-20? (NO)

3. **display** error message describing exception

4. **exit** with erroneous status code (1)

**Test:** Successful

**Test case 3:**

**Input:** 0

**Expected output:** Error out of range

**Actual pseudo**:

1. **input** size >> 0

2. **if** size is numeric (YES) and within range 5-20? (NO)

3. **display** error message describing exception

4. **exit** with erroneous status code (1)

**Test:** Successful

**Test case 4:**

**Input:** -1

**Expected output:** Error out of range

**Actual pseudo**:

1. **input** size >> -1

2. **if** size is numeric (YES) and within range 5-20? (NO)

3. **display** error message describing exception

4. **exit** with erroneous status code (1)

Test: Successful

**Test case 5:**

**Input:** “some string”

**Expected output:** Non numeric string

**Actual pseudo**:

1. **input** size >> “some string”

2. **if** size is numeric (NO) and within range 5-20? (NO)

3. **display** error message describing exception

4. **exit** with erroneous status code (1)

Test: Successful

# 4. Results and discussion

With such detailed requirements it was easy to guess what the results were going to be before even starting out with the design, especially since most of the tasks were closed-ended. (ie. didn’t leave much to my imagination).

I made assumptions where possible, mainly when it came to handling input and the possible edge cases (I mainly used 0, 1, -1 and a string of characters). All of which were handled well according to the test-cases.

I didn’t encounter any major problems with any of the tasks. However I believe I might have been overly enthusiastic with my test-cases and writing their results. One thing that I found problematic was the amount of work that had to be put into these relatively basic tasks when aiming for a higher grade.

I appreciate and understand why designing a solution might be important before moving forward with implementation. However, it seems unnecessary to design both a flowchart and pseudo code as they are both designed to solve such similar problems in the design phase, with the pseudo code being a bit more verbose and more helpful when debugging.

I hope that we won’t be required to perform such top-heavy design when going into the actual implementation of the code in later labs as it is a lot of effort for such a small yield.

# 5. References

A reference list of works used in the report. Use the Vancouver system (<https://en.wikipedia.org/wiki/Vancouver_system>) in the listing.