CS2002 Practical 1 - Introduction to C

By: 150013565 Tutor: Juliana Bowles

Date: 13th February 2018

Abstract

The objective of this practical was to build a program, written in three stages, to calculate the "Padovan sequence". The specification required me to create multiple C files along with a Makefile. This report documents my project.

The Padovan sequence, P, is generated by:

$$P_n = P_{n-2} + P_{n-3}|_{n>3}$$

Where the first 3 terms of the Padovan sequence is defined as (1,1,1).

The first stage of the practical was to generate the sequence recursively, the second stage was to generate the sequence iteratively and the final stage was to implement a custom Padovan sequence generator where the user could define the first 3 terms.

As an extension project, continuing with the theme of famous sequences, I have built a program which prints n lines of Pascal's triangle.

1 Stage 1 - Recursive Calculation

The first stage of the practical was to implement a recursive definition of the Padovan sequence and print it out the values generated with recursion, separate each value with commas and wrap it all with (square) brackets.

Implementing the recursive definition of Padovan sequences was quite straight forward, I did not run into any major issues. The syntax for writing recursion in C is the same in Java - have a defined and reachable base case and for all inputs.

However, I did run into issues with printing the calculated values. Unlike python, where you can store all the calculated values in a list and simply call print(list), C was not the same. C has no inbuilt functions that will print the array. To solve this problem, I had to write out the print statements manually. When the array is empty print out a pair of (square) brackets. When the array is non-empty print out the open (square) bracket and then for each value up to the last value in the the array, print out the value along with a comma and space. For the last value in the array, just print out the value and the close (square) bracket.

2 Stage 2 - Iterative Calculation

The second stage of the practical was to implement a iterative definition of the Padovan sequence and print, using the same formatting as stage one, the calculated values.

Implementing the iterative definition of Padovan sequences was also quite straight forward. For each value of i up to n, if i is less than three return one. If i is greater than three, initialize an array of size i and declare the first three values in that array all as one. Then, to calculate any i, use the definition of the Padovan sequence to calculate what i is. For example, for i = 4, arr[3] = arr[1] + arr[0] which is the fourth Padovan number.

This is quite an inefficient solution as for each new value of i, I calculate each value in the array multiple times. For example, when i = 4, I calculate the values in the array up to i = 4 and once I calculate that value, I return the value and the array disappears. When I need to calculate i = 5, it would be much more efficient to store the array and calculate the next value in that array rather than calculating it all from scratch. A possible solution to this inefficiency would be to use pointers, however at the time of writing this practical, I was not familiar with pointers and did not have the time required to research and implement it.

3 Stage 3 - Repeated Calculation

The second stage of the practical was to allow users to input their own starting values instead of the default (1,1,1). I have also implemented a input validation check to ensure that the user enters data that the program can compute.

3.1 Custom Starting Sequence

To implement this section, I took my implementation from stage two and adapted it so rather than having the function return one when n is less than three, it returns what the user specified. The method used to calculate further values was the same as mentioned in stage 2.

3.2 Optional: Validating User Input

A validation checker was implemented so that the user does not enter a number which would overflow C's inbuilt integer data type. C does not have a simple try-catch like Java, instead to implement this check, I read the user inputs as data type long-long. I then passed user input numbers into a function which takes that long-long value and performs a subtraction from the max limit of the integer data type - which is 2,147,483,647. If the result is positive, the user input value is greater than the number integer can hold, which would mean that the number would overflow the memory allocated. If the result is negative, then the user has input a valid number and can be cast to the integer data type with not problems.

Another validation check that was implemented is number of inputs. For the custom starting sequence, the 'scanf' method should read exactly three inputs, if it does not read exactly three inputs then the user has done something wrong and an error message is displayed. The same applies for the length.

3.3 Improvements

As mentioned in the previous section, the problem with this iterative implementation is that it calculates the same array multiple times. Another

problem is that the method does not check whether calculated Padovan values will overflow the integer data type. This could have been checked but I have ommitted it out as the specification was vague and only implied that the input should be checked and not generated results.

4 Testing

The program passes stacscheck:

```
pc3-024-l:~/Documents/cs2002/Practical1     $ stacscheck /cs/studres/CS2002/Practicals/Practical1/stacscheck/
Testing CS2002 C1
- Looking for submission in a directory called 'Practical1': Already in it!
* BUILD TEST - build-clean : pass
* BUILD TEST - stage1/build-stage1 : pass
* COMPARISON TEST - stage1/prog-stage1-length-0.out : pass
* COMPARISON TEST - stage1/prog-stage1-length-1.out : pass
* COMPARISON TEST - stage1/prog-stage1-length-10.out : pass
* COMPARISON TEST - stage1/prog-stage1-length-3.out : pass
* BUILD TEST - stage2/build-stage2 : pass
* COMPARISON TEST - stage2/prog-stage2-length-0.out : pass
* COMPARISON TEST - stage2/prog-stage2-length-1.out : pass
* COMPARISON TEST - stage2/prog-stage2-length-10.out : pass
* COMPARISON TEST - stage2/prog-stage2-length-3.out : pass
* BUILD TEST - stage3/build-stage3 : pass
* COMPARISON TEST - stage3/prog-stage3-empty.out : pass
* COMPARISON TEST - stage3/prog-stage3-multiple-inputs.out : pass
* COMPARISON TEST - stage3/prog-stage3-standard-values.out : pass
* COMPARISON TEST - stage3/prog-stage3-straight-exit.out : pass
* COMPARISON TEST - stage3/invalid-inputs/prog-stage3-cheese.out : pass
* COMPARISON TEST - stage3/invalid-inputs/prog-stage3-cheese2.out : pass
* COMPARISON TEST - stage3/invalid-inputs/prog-stage3-one-number.out : pass
19 out of 19 tests passed
pc3-024-l:~/Documents/cs2002/Practical1 jy50$
```

Stage 1 testing was done as follows;

Test No.	Input	Expected Output	Actual Output
1	10	(1, 1, 1, 2, 2, 3, 4, 5, 7, 9)	(1, 1, 1, 2, 2, 3, 4, 5, 7, 9)
2	abc	N/A	N/A
3	-10	N/A	N/A
4	0	[]	[]

Stage 2 testing was done as follows;

Test No.	Input	Expected Output	Actual Output
1	10	(1, 1, 1, 2, 2, 3, 4, 5, 7, 9)	(1, 1, 1, 2, 2, 3, 4, 5, 7, 9)
2	abc	N/A	N/A
3	-10	N/A	N/A
4	0	[]	[]

Stage 3 testing was done as follows;

Test No.	Input values	Length	Expected Output	Actual Output
1	1,1,1	5	(1, 1, 1, 2, 2)	(1, 1, 1, 2, 2)
2	2,3,10	5	(2, 3, 10, 5, 13)	(2, 3, 10, 5, 13)
3	0,0,0	N/A	Exit	Exit
4	-10,-20,-40	N/A	Invalid input	Invalid input
5	abc	N/A	Invalid input	Invalid input
6	1293207391023971200,1,1	N/A	Overflow	Overflow

Testing was conducted manually, with inputs, expected output and actual output recorded. These tests were conducted to show that the program runs as intended and according to specification. These tests showcase the robustness of the program especially for Stage 3.

5 Conclusion

Overall this practical was fairly straight forward and was a good introduction to C. It exposed me to some of the complexities involved with C and I learned a lot from it. The biggest take-away from this practical, for me, is that nothing in C should be taken for granted i.e. there is not predefined printing method for arrays/lists you must implement it yourself.

Given more time, I would have liked to implement a pointer based solution for stages two and three. As mentioned earlier, my implementation is not efficient and could be made more efficient with the use of a pointer.