# PA101 – Learning the Ropes

## Synopsis

This first PA is going to be rather unusual. In the upcoming week, you will generally have a single program to write for the whole assignment. This time we are going to work on a few programs but their size will be much smaller.

Each of the following programs should be written in their own file which name is specified in the title of the section describing what the programs should be doing.

You will be able to download “empty” versions of each file which you will then only have to fill up with your own solutions.

## Quick Advice

If your pre-requisite “First intro to Programming” already taught you how to program in Java, you should feel at home here. On the other hand, keep it simple. For instance, even if you already know how to use functions, do not use them here. Focus on what we studied last week instead.

Some of you may wonder how to get started writing code since the instructions detail only the requirements of the problem without providing you with the step-by-step solution. If you’re in this situation, this means your pre-requisite intro to programming didn’t teach you how to program, in the sense of solving computational problems, but how to read program & translate an already available solution, expressed in plain English, into a program. E.g.

| **Plain English solution in the instructions** | | **Expected program** |
|---|---|---|
| When x is 42 then display “hello” on the screen | | if (x==42) printf(“hello”); |

That’s not programming. We are left to teach you how to program in one offering instead of two now so let’s get started!

## prog1.c – Getting some data from the user

We are going to write a program which reads two integer values from the user, makes sure they are within acceptable ranges, and display patterns of star symbols on the screen.

Before you even start programming, open the empty *prog1.c* provided to you by your instructor then add the following comments at the start of every file you will be modifying.

```
/*
Assignment  PA101
File        prog1.c


Student:    [Your Full Name] [USF email]
Date:       [date]


Features:   [Type your summary of what your program does]
Bugs:       [Specify here remaining bugs at time of submission]
*/
```

**Remark** – *Do this every time you start working on a program this semester.*

We are now ready to make your program do something interesting.

First, write your program so that it declares two integer variables **nbStars** and **nbLines** which will be both initialized to 0. As you declare the variables, add some comments for each of them specifying their role, if any.

**Remark** – *This is something you will have to do every time you introduce a new variable. In future PAs, you will have to add comments for every variable already there, even if you didn't add it yourself.*

As a rule of thumb, it is useless to write comments which simply state what you already wrote but in plain English instead of in a programming language. Take a look at the following for an example of good commenting;

```
int datakey = 42; /* key number to decrypt the user data files */
```

The following illustrates what redundant commenting would be;

```
int datakey = 42; /* datakey is an int variable initialized to 42 */
```

**Remark** – *Later, as we use to program with functions, you should comment the role of each function and all of its parameters. This will have to be done in all PAs whether or not you wrote the function yourself.*

Your program will then prompt the user for an integer value to be stored in **nbStars** and read it. If **nbStars** is lesser than 1 or larger than 3, the program will prompt again the user for a value for **nbStars**. This will happen until the value is between 1 and 3 inclusive. Your program will then do the same thing for the variable **nbLines**, accepting values between 1 and 9, inclusive.

**Remark** – *Again, you need to comment here. The idea is to show whoever is reading your program, the main steps of your work. E.g.*

```
/* Handling variable nbStars - we keep reading until it's in range */
...
/* Handling variable nbLines- we keep reading until it's in range */
...
```

*Like with the previous advice, you should not simply restate in plain English what the program does. Your readers are assumed to know how to read the programming language you're using. Here is an example of what not to do;*

```
/* testing if datakey is 42 */
if (datakey == 42) {
        /* datakey is 42 */
        ...
} else {
        /* datakey is not 42 */
        ...
}
```

*While the above-example is bad as far as commenting goes, it illustrates the kind of indentation you're expected to write programs with. The textbook and apprenticeship videos illustrate this too. The following would be an example of something no programmer would want to read, and no TA will grade.*

```
if(datakey==42){
x = 3* y /9; }
     else {
     x = y;
}
```

Here is how a session with your program should look; user input is in **blue-bold**:

```
Enter a value for nbStars   0
Enter a value for nbStars   -1
Enter a value for nbStars   5
Enter a value for nbStars   3
Enter a value for nbLines   999
Enter a value for nbLines   9
```

As part of your work, verify that your program accepts all valid pair of values and rejects any value which is out of range. Keep track of the values you tested in a text file named *prog1-tests.txt*

```
Tests used for prog1;
```

| nbStars | nbLines | expected behavior | observed? |
|---|---|---|---|
| 0 | n/a | refuses nbStars, prompts again | ok |
| 0 then 2 | 0 | refuses nbStarts, accepts 2nd value | ok |
| 2 | 0 | refuses nbLines & prompts again after accepting nbStars | ok |

Please note that this prog1-test.txt file you are creating will be submitted as part of your submission. The file is not provided with your assignment, and you can generate it with notepad, but ensure it is a .txt file.

**Remark –** *This first program should have been easy. Did you notice how the description of the problem actually told you what you had to do step by step? Now let us work on developing your ability to take a description of the outcome & figure out how to implement it – i.e. program.*

## prog2.c – Displaying L lines of N stars

When you have *prog1.c* of project *PA101-prog1* working, cut and paste its contents inside the *prog2.c* file inside project *PA101-prog2*. This will require you to follow the steps below;

- Open project *PA101-prog1*
- Select the text of the source file in *prog1.c*
- Copy it using the edit menu
- Close project *PA101-prog1* without leaving the IDE
- Open project *PA101-prog2*
- Open the *prog2.c* file
- Paste the contents using the edit menu

Now that you have two values **nbStars** and **nbLines**, we are going to use them to display a pattern of star symbols, '*', on the screen. Your program will display **nbLines** lines on the screen. Each line will start with a star symbol followed by its number, starting at 1, and a space. After that, your program will display **nbStars** star symbols.

For **nbStars**=3 and **nbLines**=5 the output should look like:

```
Enter a value for nbStars   3
Enter a value for nbLines   5
#1 ***
#2 ***
#3 ***
#4 ***
#5 ***
```

**Remark –** *We are getting a bit closer to real programming problems here. You are told what the solution should look like, not how to program it.*

**Remark –** *There is another very interesting thing to learn here; in a graded assignment, I would have given you both the instructions for prog1 and prog2 to read in one shot. If you break down the original problem into small pieces, you are able to implement and test them much more easily.*

## prog3.c – Analyzing a Problem

Same as before, copy your work from the *prog2.c* file in the *PA101-prog2* project into the *prog3.c* file contained within the provided *PA101-prog3* project.

Now we want the number of stars displayed in each line to change depending on the number of the line. The following table will tell you how many star symbols we want to display given values for **nbStars** and **lineID** which will be the number of the line on which we are displaying the star symbols:

| nbStars | lineID | # of star symbols to display | nbStars | lineID | # of star symbols to display | nbStars | lineID | # of star symbols to display |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 | 1 | 3 | 1 | 1 |
| | 2 | 2 | | 2 | 3 | | 2 | 4 |
| | 3 | 3 | | 3 | 5 | | 3 | 7 |
| | 4 | 4 | | 4 | 7 | | 4 | 10 |
| | 5 | 5 | | 5 | 9 | | 5 | 13 |
| | 6 | 6 | | 6 | 11 | | 6 | 16 |
| | 7 | 7 | | 7 | 13 | | 7 | 19 |
| | 8 | 8 | | 8 | 15 | | 8 | 22 |
| | 9 | 9 | | 9 | 17 | | 9 | 25 |

Find out the formula giving you the number of star symbols to display, based on the values of **nbStars** and **lineID**. Use it to modify your program so that the number of star symbols displayed on each line follows the same logic than described in this table. Here is an example of what the interaction with your program should look like;

```
Enter a value for nbStars 3
Enter a value for nbLines 5
#1 *
#2 ****
#3 *******
#4 **********
#5 *************
```

As part of your work on part#3, you will provide a comment at the end of your program explaining how you came up with the formula you used.

**Remarks –** *A part of programming is to understand the problem to be solved. This exercise illustrates this aspect of programming.*

# prog4.c – Analyzing another Problem

**You should start working on prog4 from scratch in its empty file.**

We are going to take an imaginary game, something you have never heard about, and we are going to write a program to help the people playing this game. If you feel the game is silly, imagine you have an accountant / quantum physicist / artist explain to you, in their own jargon, what kind of program they want.

You got it; learning to program is learning to transform arbitrary requirements from a field you are not an expert in, into descriptions, which are then turned into designs before to be implemented as programs.

For this assignment, the game we will be playing with require the use of 3 six sided dices. For a roll of 1 or 2, the result is 1. For a roll of 3 or 4, the result is 2. For a roll of 5 or 6, the result is 3. As the user rolls each die one after the other, he/she obtains three values between 1 and 6 which are immediately interpreted as three results between 1 and 3. Let's call these results R1, R2 and R3. Each set of 3 rolls will award the player a certain number of points, following the rules detailed below.

Our program will not simulate the dice rolls nor will it "play this game". Instead, it will compute the reward, based on the R1, R2 and R3 values which will be entered by the user. Players told us they would like this so they don't have to figure out the reward by hand.

The first step for our program will be to compute the payoff; that is the number of points a player will get, given a set of values for *R1*, *R2* and *R3* which the user will enter when prompted to do so. The payoff will be then computed and displayed on the screen.

The rules to compute the payoff are as follows;

- To start off with, the payoff is set to the value of *R1*
- If *R2* is lesser than *R1*, we add its value to the payoff. We then look at the other values; if *R3* is lesser than *R2*, we add twice its value to the payoff. If not, we check whether *R3* is lesser than *R1*, it this is true then we add *R3* to the payoff.
- If *R2* wasn't lesser than *R1*, we check whether *R3* is lesser than *R1*. If this is true, we add *R3* to the payoff.

This is how our game works; no wonder we need a computer program. Use the forums to get help in translating what is an ambiguous language – i.e. natural language e.g. English – into something more formal – i.e. programming language. Feel free to also use the forums to get help understanding the requirements or exchanging the tests you use.

**Remark –** *Did you check R1, R2 and R3 were within acceptable range when reading them from the user?*

## Forums – Is your program really working?

As we discussed in another document, your program is not done when it compiles nor when it runs. You need to test it to make sure you did not forget something or implemented an erroneous logic. When you have a solution implemented, write a *prog4-tests.txt* file with the list of tests you have used to make sure your program works. You should not hesitate to post the test file on the forums to get help from others.

The range of values for R1, R2 and R3 are small so we might build a table with all possible combinations of values and run our program with all possible inputs. We would then compare the observed result with the one we would expect by applying the formulas by hand.

Now, if the ranges of values were [1-99] instead of [1-3], no one would really be interested in pre-computing by hand 99*99*99 payoff values.

Exhaustive testing is great but most of the time it features a bunch of redundant tests. So instead of having all possible tests, we are going to try to figure out what it the smallest list of tests which will still allow us to verify all aspects of our program.

Use the forums to discuss the tests you would use. You may post them as *prog4-test.txt*;

```
Tests used to validate in Step #3;

     R1     R2     R3           expected payoff           observed payoff

     1      3      3            something                 the same thing?

     Justification;

     Fine! I admit I picked these values at random... Partial Credit plz?
```

Make sure that, for each test, you justify whey you need them.

While it is perfectly fine to exchange tests on the forums, make sure you do not put any of your solution, source or algorithm, in your posts.