Some Suggestions on Good Programming Style for CS10003 Students

NOTE: *These suggestions are to inculcate some good programming practices as you embark on your programming journey. The goal is to write more readable and understandable programs that leads to easier debugging and maintenance. Some of them are particularly useful only when you handle large codes, which you will not in this introductory course, but good to inculcate a habit. None of these affect the correctness of the program if you don't make mistakes, you can still write a correct program without following any of these. Just that they help you in reducing your chance of mistakes and time taken to find out what you did wrong if you make a mistake (debug your program).*

For CS10003, you are not required to follow this for any exams. We will not test this in the theory and no marks will be deducted if you do not follow this.

The suggestions are from experience of coding, handling beginning-level programmers over the years, and some usual practices common among many programmers. Please feel free to ignore if you do not agree, or come argue/discuss your issue with the teacher, you are always welcome.

---

1. Once you are given a problem, before you write a single line of C code, think and write the logic on paper (steps in English, flow chart, pseudo-code, whatever you are comfortable with). Your code will come out of your logic. If you immediately start writing code and get the logic wrong, it is much harder to repair your program.

2. Give mnemonic, meaningful names to variables wherever possible so that looking at the name tells you something about the meaning of the value the variable will hold. For example, if a variable is supposed to carry sum of numbers, you can name it "sum" (do not just name it say "x"), if it is meant to carry average of some numbers name it "average" or "avg" etc. Also, it is customary to use all-caps for constants only. So usually, SUM is not good for a non-constant variable that will hold the sum of numbers. On the other hand, if you are using the value of pi in several places in your program, it is good to define a constant called PI at the beginning and initialize it to the proper value, and then use this constant whenever you need it in your program.

3. Try declaring all variables used in a function (including main()) at the beginning of the function, and not wherever you need it. Declaring it only when you need it is also useful in some cases, but as a beginner, you should declare everything at the beginning so that you can easily look them all up in one place.

4. As a beginner, it is good to put brackets, (), in expressions to explicitly enforce precedence rules you have in mind, as precedence rules are easy to forget and can give tricky errors sometimes. So in your logic, if you want to add a and b first, then add c and d, and then

multiply the two sums, write (a+b)*(c+d), not a+b*c+d which is wrong. Compiler will not give you error, and though this one is a simple expression, it is hard to debug if the expression is more complex.

5. Try avoiding writing very long expressions in a single statement. If anything goes wrong (for ex., incorrect precedence from what you want), it is harder to debug. Use temporary variables to break into shorter expressions in multiple statements as per the order you have in mind. This makes it easy to check intermediate values (evaluation of partial expressions) if needed.

6. Always indent your program as you type (not at the end after finishing your program). Each block (beginning of a function, statements inside an if/else, for/while etc.) should start with another level of indent (typically one tab or 4-5 spaces, no hard-and-fast rule).

7. As a beginner, it is always good to type { } for each if/else/for/while first before filling in statements inside it. For more than one statement (compound statement), {} are necessary. But even if you have a single statement under say an if, it is good to give the {}. That way, if you add another statement later under that same if condition, for whatever reason, you can never go wrong. It also avoids unwanted dangling-else problems as shown in class.

   As examples of 6 and 7 above, if you had two nested for loops and an if-else statement inside the inner for loop, it should look like (just a dummy program, do not worry about what it does):

```
int main()
{
        int x, y, sum = 0 cnt, n;
        scanf("%d", &n);
        for (x = 0; x < n; x++)
        {
                cnt = 0;
                for (y = x + 1; y < n - 1; y++)
                {
                        if (x + y < n)
                        {
                                sum += x + y;
                        }
                        else
                        {
                                sum = x;
                        }
                }
         }
}
```