Dr. T's Style Guide

Version 0.2

Trevor M. Tomesh
trevor.tomesh@uwrf.edu
University of Wisconsin - River Falls
Department of Computer, Information and Data Science
Created: September 7th 2018
Updated: February 16th 2023

Introduction

> "Even if you don't intend anybody else to read your code, there's still a very good chance that somebody will have to stare at your code and figure out what it does: That person is probably going to be you, twelve months from now.." - Raymond Chen, Microsoft Developer [1]

Programming is, in many ways, a form of creative expression. While the ultimate aim of any piece of code is to explain how to solve some sort of problem, how you express the solution is in many ways more important than the actual solution itself.

Code can be beautiful. Well written code is not only easy to read but has certain aesthetics to it. To appreciate the aesthetics and artistry of code, I would recommend the software studies book *"10 PRINT CHR$(205.5+RND(i)); : GOTO 10"* by Montfort Et Al. which provides a guided meditation on a single line of Commodore 64 code as well as "If Hemmingway Wrote Javascript" by Angus Croll.

This document is a collection of best practices recommended for my students. While it is heavily influenced by the python PEP-8 style guide, these practices can be applied to most standard programming languages – except maybe perl. Perl is probably beyond redemption.

Warning: Please note – this is an early version of a work in progress. Do not take it as gospel. It is neither thorough nor complete, but hopefully it will be generally useful!

Comment Your Code

No matter how great of a programmer you think you are, you are not too good to comment your code. Period. The key to readability is well commented code.

Prologue Comments

The "prologue" is the introductory section of your code. Generally, this is done in the form of a single block comment. The prologue should contain the following information:

- program name
- author name
- course title
- date
- assignment number
- description of program

The following is an example in C code, although the same basic principles apply to any programming language.

```
// program: hello.c
// author: Trevor M. Tomesh
// course: CS 427
// date: 2018/09/06
// assignment #1
// description: this program is a demonstrator program that shows how to format
//              code properly. Note that this prologue section is not using
//              block quotes (slash-star). This is the preferred way to do it
//              because it makes it easier to identify where this section
//              starts and stops. Note that this is followed by a series of
//              dashes to mark where the prologue ends.
//------------------------------------------------------------------------------
```

There is no ambiguity as to what this program does and who wrote it for what purpose. This should be at the head of every piece of code you write. That includes header files! Of course, you will need to use common sense as to what exactly belongs in the prologue – being that this is for a course, the course number and assignment number is listed. In a "normal" coding environment you will not list a course number or assignment number, but perhaps the name of your employer or the project that the work belongs to. Use your head.

Explaining Code

One of the main purposes of commenting is to summarize what a block of code does. According to Steve McConnell, author of canonical text on coding style "Code Complete":

> "Comments should explain, at a higher level of abstraction than the code, what you're trying to do. [2]"

A common mistake is to simply restate the code in plain English. For example:

```
// print the text "hello world" to the screen
print("hello world");
```

It is self evident that this prints "hello world" – one need not re-state it.

Consider the following examples for a basic euler's method:

Good:

```
// compute the solution of a given function with euler's method
while(x<=t)
{
    k=h*fun(x,y);
    y=y+k;
    x=x+h;
    printf("%0.3f\t%0.3f\n",x,y);
}
```

Bad:

```
while(x<=t) //while loop
{
    k=h*fun(x,y); //assign fun(x,y)*h to k
    y=y+k; //add k to y
    x=x+h; //add x to h
    printf("%0.3f\t%0.3f\n",x,y); //print result
}
```

The former is a good example because it summarizes what the code does. The latter is a bad example because it one re-iterates what the code does. In the latter example, we do not know the purpose of the code – we are only aware that it does some computations and prints some results. This is pretty much useless to us.

Documenting Methods / Functions

Document your functions. Make it clear where they start and end, what they take and what they return.

The following is an example in Python:

```
#---------------------------------------------------------------
# listSum
# purpose: calculates the sum of values in a list
# parameter(s): <1> list (sumList)
# return: sum

def listSum(sumList):
   theSum = 0
   n = len(sumList)

     for i in range(0, n):
        theSum += float(sumList[i])

     return theSum
```

Note that in the prologue to the function there is a "divider" that shows where the function starts.

The prologue gives:

1. The name of the function.
2. The purpose of the function – what it does.
3. A description of the parameters – there's `<1>` parameter and it must be of the type "list" called `sumList`.
4. What the function returns

You may choose to follow up a function with a second divider at the bottom to clearly denote where the function ends – however, the "return" at the bottom is a clear indicator. *Functions should always return something – even if that something is nothing.*

Group Your Functions!

Say that you have functions that perform basic arithmetic operations. For example, the sum of two numbers and the difference of two numbers.
These functions both perform arithmetic operations and should be grouped together!

For example:

```
//---------------------------------------------------------------------
//                  Arithmetic Operations
//   function:            sum
//   purpose:             calculates the sum of two integers
//   parameter(s):        <2> int a, int b
//   returns:             <1> int theSum

int sum(a,b)
{
   int theSum = a + b;
   return(theSum);
```

```
}

// function:               diff
// purpose:                calculates the difference of two integers
// parameter(s):           <2> int a, int b
// returns:                <1> int theDiff

int diff(a,b)
{
   int theDiff = a - b;
   return(theDiff);
}
```

etc...

Naming, Spacing and Layout

First and foremost, even if you are not a python programmer, you should read the python PEP-8 guide to get an idea of what well layed out code should look like. (https://www.python.org/dev/peps/pep-0008/)

Indentation

Even in a “bracketed” language such as C or Java, you should still use indentation to denote blocks of code. Moreover, that indentation should be four (4) spaces from the end of the last indentation layer. No tabs.

For example:

```
for(int i=0; i<100; i++){
    if(i==42){
        print("Life, the universe, whatever...");
    }
    else{
         print(i);
    }
}
```

Do NOT, for the love of all that is holy, do this:

```
for(int i=0; i<100; i++){if(i==42){print("Life, the universe, whatever...");}else{print(i);}}
```

There is a special place in hell for people that do this. You are not being clever or “concise,” you are being arrogant and annyoing.

Will it compile? Sure. But who cares – it looks like hot garbage and, unless you are doing one of those “code golfing” competitions where you are intentionally trying to be as terse as possible, this will be impossible to read and you will have no friends and probably die alone. Or worse yet, you might be a perl programmer!

Line Length

The PEP-8 calls for you to use 79 characters to a line max. Stylistically speaking, even that is pushing it. A good number to strive for is about 59 characters. That way it is easier to compare code with mutliple windows open. Moreover, sticking to about 59 characters allows for a larger font that is easier on the eyes.

To avoid having to divide up lines – which is generally preferrable to running over the character width limit – try to limit outrageously long compound control statements. See the PEP-8 for more guidance on this issue.

References

- [1] Raymond Chen. Code is read much more often than it is written, so plan accordingly, 2018.
- [2] Steve McConnell. Code Complete, Second Edition. Microsoft Press, Redmond, WA, USA, 2004.