ENGR 151 Style Expectations

August 14, 2022

1 Summary

Use limited line length, consistent brackets, consistent indenting, descriptive variable names, and occasional comments that clarify the code. Make code easily readable rather than using every option C++ allows. If you are unsure whether your commenting is sufficient, ask yourself whether a competent coder unfamiliar with your program could efficiently determine what the whole program and each chunk of code does, using a combination of your commenting and the code itself.

The rules are not entirely arbitrary choices. Conforming to these guidelines will help reduce coding errors and make the code readable by others who are trained in code formatting, whether that is a GSI/IA helping debug your code, or a coworker later in your career.

Note that an example C++ file at the end of this document presents much of what is explained here. The same basic rules apply for MATLAB code, although syntax will differ slightly.

2 Formatting

- Formatting includes using indenting and whitespace consistently, in order to produce quickly readable code with an obvious visual hierarchy.

- Blank lines should separate important "blocks" of code within a function. You should generally separate variable declarations at the beginning of a function from the first line that is not a declaration, and leave blank lines before and after loop blocks.

- Statements inside braces `{}` should be *consistently indented*, using 3+ spaces or the tab key. Nested braces (braces inside other braces) should be indented two levels, and the indenting for each level should be the same. Every time you add a new pair of brackets, all code between those brackets should be indented one level to the right of them. If a code block has over about 6 indent levels (e.g., 5 nested loops with 2 nested if statements), it may be difficult to follow, so consider writing part of it as a separate function.

- Either the `//` (line) or `/* */` (block) style is fine for comments. The top of a program should have a comment block indicating what the program does, who wrote it, and when. For class projects, include your name, uniqname, date, laboratory section number, project number, filename, and a description of the program.

- Comments should help the reader understand the code, which may be someone else or it may be you 5 years later. Comments on variables should convey the meaning of the variable, e.g.

  ```
  int redApps; // number of red apples
  ```

- Avoid comments that state the bleeding obvious, e.g.

  ```
  a = b; // set the value of a to the value of b
  ```

- Each function should generally have a comment block unless the function is short and easily understood. If the function is long or uses a complicated algorithm, comment lines should explain the algorithm used. To avoid comments that "wrap around" when printed, put comments on their own line; only very short comments should be added to the end of other lines. Other long lines (like some `cout` statements) can be broken into shorter pieces on multiple lines. In general, try to keep the width of each line of your program under 85 characters.

3 Program Structure and Usage

- Use descriptive function and variable names and be consistent in the style. Using single letters `(i, j, k...)` for loop variables is fine, but otherwise use names that clarify what the code does, especially in longer pieces of code. Two common standards are:

  - `CamelCase` – Words are generally capitalized, with no spaces.
  - `snake_case` – the variable names are lower case and separated by underscores.

  For C++ in particular, other common naming conventions are:

  - Types start with capitals: `MyClass`
  - functions and variables start with lower case: `myMethod`
  - constants are all capital: `GRAVITY_CONSTANT`

- Functions that return a value should not overwrite the original values of any input parameters. Functions that change the values of any input parameters should be of type `void` (we call these functions "procedures"). Therefore, functions that modify arrays (e.g., sorting) should be of type `void`. A function that both overwrites parameter values and returns a value is considered bad programming practice by some. (There is an exception common to C-like programming; a procedure may be designed to return an "error code" when it is unable to work on the given parameters, such as when end-of-file (EOF) occurs during file I/O.)

- Declaring or initializing variables near where they are used will generally lead to more efficient, easily maintainable code, so this is encouraged. Variables that are used throughout a function should be declared in a list at the beginning of the function, which makes them easy to both find and change later. This is easier to appreciate once your code grows to a few hundred or thousand lines. Initializations for `while` loops should be immediately before the `while` statement itself, especially for nested `while` loops.

- The increment statement of a `while` loop should generally be at the end of the loop. Use `for` loops only for simple counting loops.

- In general, avoid `break` and `continue` statements.

- Do not hard-code specific values.
  Bad example:

  ```
  int a[10];
  for (int j = 0; j == 9; j = j + 1)
  {
     a[j]= 1;
  }
  ```

  Good example:

  ```
  const int SIZE = 10;
  int a[SIZE];
  for (int j = 0; j < SIZE; j = j + 1)
  {
     a[j] = 1;
  }
  ```

- In general, avoid global variables (i.e. that can change in value). Global constants are fine but should be defined as `const`, e.g.

  ```
  const double GRAVITY_CONSTANT=9.80665;
  ```

- In programming languages that are capable of it, using `and` and `or` rather than `&&` and `||` in your code can help you avoid mistakenly using the bit operators (`&` and `|`). You may use `not` or `!`, and you can write `!=` for "not equals". Use `true` and `false` where applicable. Statements like `if(test)` or `if(!test)` should be used only if `test` is declared as a Boolean. If `test` is declared as an integer, write `if(test != 0)` or `if(test == 0)`.

- When converting the type of a value (e.g., a `double` to an `int`) consider using the `static_cast` form rather than the C `(type)value` or C++ `type(value)` forms. For example, use the following:

```
double x = 2.5;
int i = static_cast<int>(x); // preferred C++ typecast
```

The following are not guaranteed to be supported in later versions of the C++ standard:

```
int j = int(x); // C++ functional typecast
int k = (int)x; // C-style typecast
```

4 Example C++ code demonstrating style guidelines

```
// Author:  Alec Thomas
// Uniqname:  agrt
// Date:  9/1/2018
// Lab Section:  <your section>
// Project #:  <the project number>

// File: FibonacciStyleExample.cpp
// Compiler: g++

// Program Description:
// This program is a tutorial on writing code according to the style
// guidelines. The program itself calculates numbers from the
// Fibonacci sequence using the Gold Ratio, Phi


// include statements
#include <iostream>
#include <cmath>

// Global constants
const double PHI_GOLDEN = (1.0 + sqrt(5.0))/2.0; // the golden ratio

// Procedure and Function declarations (defined below main)
void printFibonacci(long F,int n);
double intPow(double x,int n);
long fibonacci(int n);

// Could put ‘using namespace std;’ here to not have to
// write ‘std::’ in front of cin / cout statements

int main()
{
    int n;   // n is order of Fibonacci number

    std::cout<<"Input n = ";
    std::cin>>n;

    // Now calculate nth element of Fibonacci sequence
    int fibValue = fibonacci(n);

    printFibonacci(fibValue, n);

    return 0;
}

// This procedure prints the number from the Fibonacci
// sequence to screen
void printFibonacci(long F, int n)
{
    std::cout<<"F("<<n<<") = "<<F<<std::endl;
}

// This function calculates the nth element of the Fibonacci
// sequence using the "golden ratio" phi
long fibonacci(int n)
{
    // First calculate F(n) using golden ratio solution
    double fibApprox;
    fibApprox = intPow(PHI_GOLDEN,n) - intPow(1.0 - PHI_GOLDEN,n);
    fibApprox /= sqrt(5.0);

    // Now we round to nearest integer and typecast as int
    // as the fibApprox answer will have round off error
    fibApprox = round(fibApprox);
    long fibInt = static_cast<long>(fibApprox);

    return fibInt;
}

// This function calculates x^n for integer n
double intPow(double x, int n)
{
    double result = 1.0;

    // first deal with negative power by inverting x
    if (n < 0)
    {
        n = -n;
        x = 1.0/x;
    }
    // note indenting here
    while(n > 0)
    {
        if(n%2 == 0) // if n is even can square x for efficiency
        {
            n = n/2;
            x = x*x;
        }
        else    // simply multiply the result by x
        {
            n = n - 1;
            result = result*x;
        }
    }
    return result;
}
```