Software Style Guide
for Writing Programs in C++
SY DE 121 – Digital Computation

Department of Systems Design Engineering
University of Waterloo

1 Introduction

This handout presents important information pertaining to program style for labs in SY DE 121 – Digital Computation. Follow this closely, as proper style is expected for all lab submissions.

The following is a brief introduction to the program style in SY DE 121. This handout is written to indicate what the T.A.'s are looking for with respect to program style when marking assignments. Keep this document handy you will need to refer to it when writing your programs. Do not worry too much if you do not completely understand the example code just yet; as the course progresses you can read over new sections to get an idea of what is expected out of your lab.

Good program style improves readability and makes the program easier to maintain (and to mark!). One of the most important tasks in programming (and engineering in general) is *communication*. Proper program style is an important part of ensuring that someone who is not familiar with the code can understand the program you have written. That someone may even be you in a few weeks time!

Please remember that these are general guidelines and not hard and fast rules. For example, individual companies in industry will have their own norms and specifications for style. The most important thing to remember is to try to keep things as clear and simple as possible. Striving for clarity and simplicity will lead to more reliable programs, fewer bugs to weed out, and programs that are easier to debug and maintain. As with all rules, there are exceptions so if you are unsure about what to do, ask your T.A. and he or she will be glad to help you out.

These guidelines are not complete and cover only basic style and layout; more instruction and direction will be given as you cover more advanced topics in the course and in future software courses.

1.1 Why is style so important?

Appropriate programming style serves two purposes:

1. It improves readability and hence clarity and maintainability. The compiler does not care about this part. It is done for humans: both you and future programmers.
2. It leads to programming conventions that reduce the chance of insidious bugs that may be hard to track down, such as logic errors and other run-time bugs. Often in these cases the program may work, but not as intended.

1.2 Style and Marking

Failure to follow a consistent, appropriate style will result in a loss of marks not only on your assignments, but also on some midterm and final exam problems.

2 Program Layout

Proper program layout is one of the best ways to make your code easier to read and understand. When laying out your code, you want elements considered as part of a group to look like a group.

2.1 Indenting

Indent one tab stop every time you enter a new block of code, and every time you begin a new conditional statement.

There are only a few rules to remember when indenting. First, you should always indent whenever you reach a new block of code. This usually means indenting whenever you open a curly brace (i.e. a '{'). Also indent one level for the code following each conditional statement (`if` statements, `while` statements, `for` statements, etc.) You should decrease your indent back again when your block of code or conditional statement ends.

Most IDE's (Integrated Development Environments such as Microsoft Developer Studio) and editors (such as EMACS) attempt to handle indenting for you.

2.2 Brackets

Brackets are placed on separate lines and paired brackets should have the same level of indentation.

When using curly brackets, make sure that each bracket is on its own line and that each pair of brackets have the same indentation. This makes it quite easy to find pairs of bracket within nested code. See example at end of this Style Guide.

2.3 Blank Lines

Insert a blank line before each new block of code.

Spacing is another important way of making your program more readable. Putting a blank line after a small group of related statements (a group of variable declarations, cin statements, etc.) breaks the code up into smaller chunks and makes the program more readable. Each group is often started with a comment indicating the commonality between these statements (see section 3.2 for more details). This grouping of related statements is roughly analogous to a paragraph in writing, where statements in a paragraph convey details pertaining to a single topic or concept.

2.4 Whitespace

*Use spaces around keywords, symbols, and operators (i.e. +, -, *, /, <<, >>, <, <=, etc.).*

When writing statements, judicious use of whitespace (spaces, tabs, etc.) around mathematical expressions, keywords, symbols, and operators will help make your code more readable. You should insert a space before and after operators such as +, -, <, (), etc.

For example,

```
hypotenuse = sqrt ( side1 * side1 + side2 * side2 );
```

is preferred over:

```
hypotenuse=sqrt(side1*side1+side2*side2);
```

2.5 Line Length

Break up lines that run over the edge of the page.

Avoid writing lines that are longer than the width of a printed page (80 characters). If necessary split long lines. The computer might split long lines for you when printing, but it will always make your code more readable if you split them yourself.

3 Commenting your Code

Comments are used to document your program and improve readability. Comments do not cause the computer to perform any action when the program is run. The general rule for commenting is that comments in your program should be complete, yet minimal. You will find in this course that our emphasis is greater on being complete rather than minimal. Our intent is to get you thinking about appropriate programming style early, so that when you move to larger programs, appropriate habits picked up in this course will help you in the future.

Later in the course, you will be taught how to comment more advanced features of C++, such as classes and structs, as you learn about them.

3.1 Comment Styles

Except for very long comments (such as file headings) prefer '//' style comments over the '/ */' kind.*

C++ accepts two styles of comments. One is the '//' kind and the other is the '/* */' kind. With the '//' kind, everything after the double slashes up until the end of the line is ignored by the compiler.

```
// everything after the slashes is ignored on this line
```

With the '/* */' kind, everything between the two comment indicators is ignored.

```
/*
Everything between these two
Comment indicators is ignored
*/

/*  it works on one line too! */
```

Both work fine, but in general we prefer the '//', because these comments can be nested (note that this is also the commenting style used by the texbook). With the '/* */' kind, the compiler considers the comment finished the first time it reaches a '*/'. The '/* */' kind are useful for comments that span many lines (like file headings), commenting

something in the middle of a line, and commenting out a section of your program in order to find a bug.

3.2 Sectioning

Divide your code into sections and put a comment before each section.

When commenting your code, you should not comment every line since this obscures your program code and makes it difficult to follow. Related statements should be grouped together into "paragraphs", roughly analogous to paragraphs in writing. The statements in a "paragraph" should together perform a single, simple task. "Paragraphs" should be separated from each other by blank lines, and should be preceded by a comment describing what task they are performing.

Programs in this course normally get input from the user, performs some calculations on the data, call a few functions, send some output to the screen, etc. Your program sections should reflect this sort of organization. Break your program up into logical sections and give a comment for each section.

```
// variable declarations
…
…

// get inputs from user
…
…
```

Except for brief comments (such as commenting a variable declaration), comments should generally appear on separate lines, and should be indented to the same level as the code that follows it.

Avoid obvious comments, like

```
int value;        // declare an int called value
count++;          // increment the counter
```

These serve no purpose and only act to clutter the code.

3.3 File Headings

Begin each file with a heading containing pertinent information about your program.

A file heading is a comment at the start of each file that tells some information about the author and about the program. You should have a file heading at the beginning of *every* file (not just every exercise!) Later on you'll be writing programs using multiple files.

For the purposes of SY DE 121, your file heading should contain the following information: your name, student number, date the project was written, course, lab room, assignment number, exercise number, filename, and the name you gave the C++ project. After the file header, the purpose of the program (or that particular file) should be stipulated.

Note that when we say purpose, we are looking for the *purpose of the program* (i.e. *what the program does*) NOT the purpose (or stated objective) of the exercise. This is an important distinction. Try not to be too verbose when writing the purpose of the program. Succinctness is the key!

3.4 Function Comments

Comment each function declaration with a statement about its purpose.

Comment your declarations with a statement saying what your function does. The comment for your function prototype should appear either immediately before or immediately after the function prototype. Your declaration comment should be written in such a way that someone reading your code could use and understand your function without reading the function definition. Note that the function comment method outlined in Savitch is also acceptable.

Comment function declarations with a description of their inputs, side effects, and return value.

- Input: describes each argument that is passed to the function in the variable list, and any limitations on the variable (e.g. must be a positive number, etc.)
- Side Effects: describes what effects it has on the running of the program. This includes any output to the screen (or any other device) and how each pass by reference variable is modified by the function.
- Return Value: describes what the value returned represents. No comment is needed for void functions, all other functions will have one.

Function definitions should be commented and laid out much like the main() part of your program. Break your code into sections, and comment each section.

4 Declaring Variables and Functions

Variables should be declared one per line, with a short comment indicating their use. Note that this is not the convention used in your text, but this is the convention that we will adhere to in this course.

There are two reasons for this rule. First, it helps the reader of the program to know exactly the purpose of the variable. Second, it helps you, the writer of the program, to think about what kind of data storage a program or function will need.

4.1 Naming Variables

Declare your variables in all small letters, one per line, with a brief comment for each, and use variable names that give an indication as to their use.

Giving your variables and functions meaningful names helps a reader understand your code. Giving arbitrary variable and function names is not appropriate. When assigning variable names you should always ask yourself, "does this name convey any information about the variable usage?" If the answer is yes, good; if not, you should probably think of a better name.

Some examples of good variable names:

```
double vehicle_length;          // length of assembled vehicle
double pizza_diameter;          // diameter of cooked pizza
int num_computers;              // number of computers in classroom
double temperature;             // current temperature
```

Some examples of bad variable names:

```
int joe;
double number;          (what number??)
double other_number;
double temp;            (temperature or temporary?)
```

Variables should be named using all small letters. Compound names should be written with each element separated by an underscore (e.g. max_rain_for_year). Names using mixed upper and lower case letters will be reserved for classes and structs (which you will see later in the course). Do not begin variable names with an underscore, compilers generally use these to identify special global variables. When documenting your variables, you should not be redundant to the variable name ie.

```
int num_computers;              // number of computers in classroom
```

is preferred to :

```
int num_computers;              // number of computers
```

4.2 Naming Constants

Constant names should be written in all capital letters.

Another important piece of program style is the use of the const keyword. This allows you to make a variable of interest constant; meaning its value cannot be changed throughout the execution of the program. You should use constants whenever you have a variable that should not change throughout the length of your program. Plausible examples would be things like the number of seconds in a minute, inches in a foot, a value for pi, etc.

In C++ (not just in this course) convention dictates that names for constants be written in capital letters. You are expected to use constants and follow this naming convention in the course. Just as with variable names, the names for your constants should convey some sort of meaning as to what your constant represents. This normally doesn't tell the user the value of the constant. Names such as const int ZERO = 0; or const int NINE = 9; is considered poor style.

Constants should be declared before regular variables.

4.3 Naming Functions

Use small letters for function names, and use names that convey some information as to their use.

The general rules for naming functions are much the same as for naming variables, you want to try and pick a name that conveys meaning about the use of the function.

Caution: Make sure you pick names for your functions that are not in conflict with variable names you've already used in your program, or with C++ keywords (see Appendix 1 in Savitch for a complete list of keywords). This will generate errors in your program!

4.4 Global Variables

Don't use global variables.

Global variables are variables declared in the global scope (i.e. outside of any curly braces { }, often just before the main() part). *Do not use global variables in your programs.* Global variables can be used (and modified) by any function in your program without explicitly being passed to the function. Using global variables decreases modularity (the ability to use software components over again) and can cause unexpected behaviour if your function modifies a global variable unexpectedly. These concepts will be more fully discussed as the course progresses.

Using global variables generally indicates a deficiency in your program design, and often that a student does not have a clear understanding of how to use functions properly. If you feel that the only solution to a problem requires the use of a global variable, ask your T.A. for help.

5 Echoing User Inputs

Input should be echoed back to the user as soon as it is received. It reassures users when they receive immediate feedback after entering input. Echoing also helps programmers when they are testing their programs; errors in input statements are a common cause of defects.

6 Some Final Notes

Your textbook has a short section called "Program Style". Also, observe the way programs are laid out in the book and the program fragments that are part of your assignments. The best method to improve your programming skills is to practice, and to imitate conventional programming style.

You will note that your text and the code fragments for your assignments sometimes disagree with the guidelines written here (for instance Savitch does not break his code into discrete sections and put a comment for each section, nor does he declare variables one per line). When in doubt, refer to the guidelines given here.

7 Advanced Topics

This section describes some of the more advanced features of C++ and how they relate to program style. A full description of these features will be given in your lectures, the intent here is not to describe these features, but to indicate the practices we will follow when using them.

7.1 Structs

Structs should be given a meaningful name, written in mixed upper and lower case letters, with a brief comment as to its use, and a brief comment about each data member.

A struct, which you will see later in the course, is a new type, similar to a double or an int. In order to be able to use the struct, it should be given a meaningful name, written in mixed upper and lower case letters. A brief comment should precede the struct giving a description as to its use. Data members should be given meaningful names, be declared one per line, and each be given a brief description as to the use of the member.

7.2 Separate Compilation

Include a file heading comment for every module in your program.

When splitting a program up into multiple files, a file heading comment should be given for each module as described in section 3.3 (a module is the combination of a header file - .h and implementation file - .cpp). You should also include the dependencies of the header file (i.e. what .cpp file it needs, where the function definitions are located).

7.3 #define Names

Use the name of the file, written in all capital letters, substituting the period in the file name with an underscore.

When choosing names for your #define statements, you must choose a name that is unique to your program. You cannot choose a name that is the same as #defined in another file, nor can you choose a name that is the same as one of your functions in your program. Remember that

names that you #define don't obey the usual rules of program scope, so you also have to insure that it does not conflict with any variable or constant names.

For this reason when using #define / #ifndef statements to wrap your header, convention dictates that we use the name of the file, substituting an underscore for what otherwise would be a dot (note that the period (.) cannot be used when using #define).

Use all capital letters in the name, just as is done for constants. The reasoning behind this is that C++ is derived from a language named C, and C does not have the const keyword, constants are declared by using #define. Furthermore, this will help prevent name conflicts caused by using #define for the same name you have given one of your functions (just make sure it's different from any constants you declare!)!

For example for the file functions.h, we would use:

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
…
#endif
```

as our preprocessor directives.

If you have any further questions, don't hesitate to ask for help.

8 Example Program – Feet to Inches

```
//  Project:  Feet to Inches
//  Purpose:  converts a distance in feet to a distance in inches
//  By:  I.M Cool
//  Student No.:  123456
//  Course:  SY DE 121
//  Assignment #1, Exercise #1
//  File:  main.cpp
//  Due Date:  Friday, September 29, 2010

#include <iostream>
using namespace std;

int feet_to_inches ( int num_feet );
// Function converts a distance in feet to a distance in inches
//        INPUTS: num_feet - an integer number of feet
//        RETURNS:  an integer number of inches

int main( )
{
        // variable declarations
        int num_feet = 0;               // distance in feet
        int num_inches = 0;             // distance in inches

        // get input from user
        cout << "Please enter the number of feet to convert: ";
        cin >> num_feet;
        cout << endl;
        cout << "You entered: " << num_feet << endl;

        // check to see if user entered a positive number of feet
        if ( num_feet > 0 )
        {
                num_inches = feet_to_inches ( num_feet );

                // output results to screen
                cout << num_feet << " feet equals " << num_inches
                        << " inches\n";
        }

        else    // negative number of feet
                cout << "You entered 0, or a negative number of feet\n";

        return 0;
}

int feet_to_inches ( int num_feet )
{
        // variable declaration
        const int INCHES_IN_FOOT = 12;
        int inches = 0;

        inches = num_feet * INCHES_IN_FOOT;

        return inches;
}
```

File Heading with all relevant information

Function declaration with comment

Code is broken up into sections, with a comment for each section

Echo user inputs

Long line split across two lines

Constants in capital letters

Code is nicely indented