CS 142 Style Guide

CAREFULLY READ THIS GUIDE; THESE ITEMS ARE WORTH A SIGNIFICANT PORTION OF YOUR LAB AND EXAM GRADES.

How the guide works: In the English language, there are many different ways to convey a certain message or idea: some of these ways are acceptable, whereas others are not. Similarly, there are acceptable and unacceptable methods to coding a program. In order to help you establish good coding practices we have implemented some style requirements. While there may be more than one acceptable way to do things, most companies, etc. will have a coding style guide which all programmers will follow in order to make it much easier to understand and adapt code. We do the same for this class and expect everyone to follow the style guide below. This will also make it so that TAs can know how to grade your style and you can know what to expect. Normally an organization would want just one approved style. For this semester, we have sections being taught from two different books which have chosen different conventions. Thus, there are a few cases below where we will have an "or" allowing one of two possible styles. However, you must be consistent in these cases, and once you choose one style for a particular program, you must stay with that choice for the entire program. At the end of chapter 2 in the zyBooks text (and in the appendix of the traditional section text) is a section on style which you should also read and which is similar to this one. However, THIS guide is the one we will use for this class and which the TAs will use when grading your labs and exams.

Get in the habit of using correct style and use correct style right when you start writing your code, rather than think you will come back later and fix it up. Take time to look over your code and make sure it looks "good." You should be proud of your code and want to show it off to your neighbors.

The guide progresses alongside the class. For example, the first portion of the guide will be applicable from the moment you type “Hello World” up to your final project; conversely, the portion on proper “function” style will not be relevant until you start coding functions (about halfway into the course). Next to each style category is found a list of the labs which require implementation of that category.

Point system: Each main lab or exam you turn in is worth 100 points. 75 points come from the auto-graded score you receive from zyBooks. The other 25 "qualitative points" are based on 1) did you fulfill any "Special Requirements" mandated in the lab specification, and 2) did you use proper programming style and practice. If a lab has special requirements, they will be explained in the lab specification, along with the number of points that will be taken off if not done. There are 8 *style categories* that will be graded.

1. Header Comments (Labs 1-10)
2. Format (Labs 2-10)
3. Test Cases (Labs 2-10)
4. Variables and Constants (Labs 2-10)
5. Constants and Magic Numbers (Labs 2-10)
6. Miscellaneous (Labs 2-10)

7. Functions (Labs 6-10)
8. Objects & Classes (Labs 8-10)

Most of the 8 style categories are made up of specific *styles* which are explained in detail under each category and are summarized as bullet points at the end of this style guide. Grading will be done independently for each style within a style category. For each lab, there will be a 3 point deduction for the first infraction of a style within a style category. There will be a 1 point deduction for each subsequent infraction of that style up to a maximum of 6 points per style. To help you see our goals with this, consider the style "use << endl rather then \n" under the Miscellaneous style category. If you used "\n" 25 times in your program, rather than "<< endl", we would not want you to lose all 25 qualitative points based on that one repeated error. We want to take off enough for the first incident (3 points) to get your attention so you will improve next time, and take off some for repeated mistakes. Thus, in this case you would lose 6 points for this particular style. Note that certain style categories outline the point breakdown in more detail for the given category.

In the case of the the style of *descriptive naming* the infraction comes when you declared the variable. If you declare a variable with name "x" you will have 3 points deducted for bad naming, but not get another point off for each time that *same* variable "x" appears in your code. (If you declare "x" in another scope you would lose another point). If you have another poorly named variable (e.g. "y") you will have another point deducted for its declaration, up to 6 points. There are a couple styles, such as descriptive names and test cases, which are somewhat subjective. You might have a variable name like "distance" which is not terrible but which is still not sufficiently descriptive. In a case where no blatant infractions of a style have occurred, we give the TA the latitude to take off 1 or 2 points rather than 3. If you are not sure about a subjective case like this, just communicate with the TAs before you submit so that you can both learn better and get your best score.

Note that your qualitative points lost will never exceed 25 points. For example, assume a lab had 3 special requirements each with a maximum deduction of up to 10 points. Assume you received all 75 points for auto-grade, missed 8 points based on style, and you did not do one of the special requirements. You would have an lose 18 point of the possible 25 qualitative points and receive a total score of 82. Assume you received 60 points for auto-grade, missed 15 points based on style, and you did not do any of the special requirements. You would not receive any of the 25 qualitative points and receive a total score of 60. The TAs will correct all style infractions in your code, so that you can learn and improve, even though you may have already exceeded the 25 point qualitative maximum.

To compute the total grade the TA will start with the number of points (up to 75) received from the zyBook auto-grade. Then any special requirements and style points will be deducted from the 25 point qualitative maximum as explained above, and the remaining qualitative points will be added to your auto-grade base. Then the TA will add early points or subtract late points, add completed extra credit (when available), and record the grade in Learning Suite. The TA will give clear feedback on points deducted, specifying where errors occurred, and giving any other commentary needed to help you learn and do better the next time. You can view feedback from the TA in the feedback box for the assignment in learning suite. When Ycode alerts you that your grading has been completed it will identify which TA did the grading in case you have any questions.

Style Criteria

1. Header Comments: Applies to labs 1-10

If any of the 4 required fields are missing or incorrect it will be a 3 point deduction. Additional errors of any kind result in a 1 point deduction. This style category caps at a max deduction of 4 points.

Some of your programs will contain one file; others will contain multiple. At the very top of each file, there must be a comment block with the following information:

1. Your name, class section number, and learning suite e-mail
2. A description of the purpose of the program
3. A description of the inputs needed for the program (if there are none, simply type "No inputs")
4. A description of the output of the program

If you want to use any of your allotted late days for a program, you should note that at the beginning of the header comments of your "main" file. Below is an example of reasonable header comments for main lab #2 - Pizza:

```
/*
Sandra Dean, Section 3, Sandra@hotmail.com
Purpose:  Calculate how many pizzas of various sizes a user should buy
Input:
   Number of people eating
   Amount of tip
Output:
   Number of large, medium, and small pizzas necessary
   Total area of pizza for everyone
   Area of pizza per person
   Total cost including tip
*/
```

At the end of the header comments for the main program file, you will usually include 3 test cases (see Test Cases criteria below).

Labs 8-10 have multiple files. For files other than main you should also have header comments. However, in these cases the comments just need to briefly describe the purpose of the file.

2. Format: Applies to labs 2-10

This section is focused on how your code looks, not on its functionality. Although sloppy code may accomplish the same thing neat code does, it is much harder to debug and to understand. Furthermore, sloppy code has a higher potential for errors. In the coding world, programmers are constantly working with code written by others. Code that is very difficult to understand hinders this process. Therefore, we expect you to develop good format practices early on.

a) Each statement should appear on its own line.

DO:

```
x = 25;

y = x + 1;

if (x == 5){

   y = 14;

}
```

DON'T:

```
x = 25; y = x + 1;

if (x == 5) { y = 14; }
```

b) There are widely-adhered to rules regarding brace placement. In this class, we will using either "K&R" or "Stroustrop" style for braces. For K&R branches, loops, functions, and classes, opening braces appear at the end of the item's line; closing braces appear under the item's start. For Stroustrop, the first brace is place on the next line under the item's start. Do not use any other style. Also, once you choose one style for your lab, use that same style throughout! In the style sections which follow, we will use "K&R" style when showing braces, but know that in all those cases "Stroustrup" style would also be fine.

DO:

```
if (a < b) {

   // “K&R” style
}

while (x < y) {

   // “K&R” style
}

MyFunction(int myInputParameter) {

  // “K&R” style

}
```

or DO:

```
if (a < b)

{

  //“Stroustrup” style which is popular

}
```

c) Sub-statements are indented 3 or 4 spaces from their parent statements, but once you pick 3 or 4, be consistent (zyBook default is 3, and Visual Studio default is 4). One way to do this is to use the tab key in the system you use to write your code (your IDE-- Integrated Development Environment). Most IDEs will convert to spaces and help you get things lined up properly, but please be careful, in the end you are responsible for the alignment and indenting for your code.

DO:

```
if (a < b) {

   x = 25;

   y = x + 1;

}
```

DON’T:

```
if (a < b) {

  x = 25;

    y = x +1;

}

if (a < b) {
    x = 25;

    }
```

d) Most items (operators, assignment operator, etc) are separated by one space (not less or more). While this is a general rule, there are common exceptions. No space precedes an ending semicolon. No space precedes the parameter list of a function or array index, etc. We will see these as we go.

DO:

```
    x = y + 3;

    if (x > y) {
```

DON'T:

```
    x=y+    3 ;

    if(x >y){
```

e) Use "Stroustrup/modified K&R" for if then else bracing.

f) For if-else statements, the else appears on its own line:

DO:

```
    if (a<b) {

        …..

    }

    else {

      // "Stroustrup" style, aka modified K&R
    }
```

DONT:

```
    if (a < b)

    {

        ....
```

```
} else {
  // Original K&R style, do not use
}
```

g) Even if there is currently just one sub-statement, braces are always used.

DO:

```
if (a < b) {
  x = 25;
}
```

DON’T:

```
if (a < b) x = 25;
```

This format works, but it can lead to errors later. Do not use it.

f) Lines of code are typically less than 100 characters wide. Code is more easily readable when lines are kept short (and all visible on the screen). One long line can usually be broken up into several smaller ones. Even if they line must be long you can do a 1 or more newlines and indent the line so that it is easier to read.

DO:

```
    cout << "This is a string" << endl;
    cout << "This is a second string" << endl;
    cout << "This is a third string" << endl;
```

DON’T:

```
    cout << "This is a string" << endl << "This is a second string" << endl << "This is a
third string" << endl;
```

3. Test Cases: Applies to labs 2-10

Each test case is worth 3 points. This style category caps at a max deduction of 9 points.

The goal of test cases is to do our best to ensure that our code is working properly. We want to test for simple situations and also for less common cases. When testing a video game, testers do not simply play the game through; instead, they do obscure things such as walk into walls and corners, jump up trees, and cross invisible boundaries. Their goal is to do what we call "break the code": they want to find every possible error in the game, expose them, and then fix them. When you test your code, you want to consider where your code could fail, and then test your program such that you are testing the possible "problem areas."

You will include 3 example test cases for main labs at the end of the file header comments (except where specifically noted: lab #1, #5 Hot Plate, etc.). Note that normal function header comments would not include test cases. We have them here for the purpose of our class and to allow us to grade your testing efforts. Three test cases are never enough to check all combinations as your programs become more complex. However, we will consider the test cases you include as examples and evidence of your more thorough testing which you did above and beyond your three examples. Therefore, we expect your test cases to be diverse, attempting to test and explore different issues.

Furthermore, we strongly suggest that you write at least two of your test cases, INCLUDING THE OUTPUT (worked out by hand), BEFORE you code up your solution. One of the most helpful things about writing a test case before coding is that it will give you a chance to think about the steps you have to take (as a human) to produce the required output. You may even want to take notes about what you did so that you will know what steps your code will also have to take. Generally if you do not understand the steps a human must follow in order to generate the required output, it will be impossible to write a program that can do so. It is critical to note that a significant portion of your test case outputs should be values that you worked out by hand (or otherwise verified) to see if your program does them right. If you just run your program and then plug in what the program output into your test cases, then you have not tested at all whether you program is getting correct answers.

For example, on the Pizza lab you will test with inputs that test different interesting combinations such as just one of each type of pizza, all types of pizza, different sizes of tips, etc. For example on the Pizza lab you might first try with 1, 3, and 7 guests to test the "one pizza each" variations. Then, maybe test for one large pizza and one medium. Did that check out? If so, then test for one large, one medium, and one small. If that checks out, you move on to larger cases. The three example test cases you chose from your overall testing should reflect this effort. Note that your descriptions and test cases will need to get more complex as your programs get more complex.

Note that Pizza lab did not ask you to test for incorrect input. Most labs will ask for that, and part of your testing should test to make sure your program handles "bad" inputs. In the first couple of labs you could actually just show the exact expected output that your program should give. However, as you go on, this would be annoying as the full output could be very long and you really just want to focus on whether your program is getting all the key calculated values right. In these cases you do not need to write down all the formatted output text and can focus on the information calculated and manipulated by the program. You can format your test cases however you would like as long as they are clear and easy to read, and clearly specify the inputs and calculated output values. In the pizza lab one might show the following three test cases at the bottom of their header comments in main. (Do not use these exact cases as your examples).

Test 1:

   Input: People: 11, Tip: 10%

   Expected Output: 1 large, 1 medium, 1 small, Sq in total: 628.318, Sq in per guest: 57.1198, Cost: $37

   Actual Output: ...

Test 2:

   Input: People: 4, Tip: 0%

   Expected Output: 0 large, 1 medium, 1 small, Sq in total: 314.159, Sq in per guest: 78.5397, Cost: $19

   Actual Output: ...

Test 3:

   Input: People: 452, Tip: 15%

   Expected Output: 64 large, 1 medium, 1 small, Sq in total: 20420.3, Sq in per guest: 45.1777, Cost: $1102

   Actual Output: ...

Another resource to help you understand good test cases is to look at the test cases your code is tested with in the zybooks submission system. We are doing test cases specifically to see if your code works and you could pattern some of your test cases after these. You should notice that the submission test cases try a variety of bad input and input which is intended to test what might be unusual cases (for example, places where you might have used ">" where you should have used ">="). In most cases it is impossible to test all possible input, so that is not what we are looking for, but rather an honest attempt to validate your code. Also once you start using functions, some of your tests cases may just be testing to make sure a certain function works correctly.

One more example: Read this part once you get to the Plinko labs. Many wonder how to do tests with the Plinko lab where there are many more possibilities and there is also a random aspect to dropping a chip. Here is an approach I would probably use if I were in your place. I would want one test case that tested the possible types of input errors and makes sure there is an appropriate re-request when an error is made.

Test Case #1

Input: The following test case is a stream of inputs which tests the possible erroneous input situations: -1, 4, 1, -3, 9, 4, 2, …. (you fill in the rest)

Output: Verify that an appropriate error message is given for each error and that there is an appropriate re-request until a correct input is given.

Note that this communicates the full goal of this test case without having to show the full long textual output which will actually be output.

For the next two test cases I would want to make sure that I was getting correct outputs for drop one and drop multiple chips (in these cases not worrying about erroneous input which is tested in a different test case). With use of rand() we can always set a fixed seed during testing so we can know exactly what the random sequence will be. One approach would be to right a separate tiny program to just seed rand with a fixed value, and then run a loop which gives you the first *n* (left, right) choices. Then I know exactly which direction will be chosen at each step. Then I could test what should happen when I drop a chip in any slot. Note that I try a drop in the middle but also one at each edge to make sure I am handling border cases.

Test Case #2:

Input: (using fixed seed of x) 1, 4, 1, 8, 1, 1, 0

Output:

```
[4.0 4.5 5.0 5.5 5.0 5.5 5.0 4.5 4.0 3.5 4.0 4.5 5.0]  Winnings: $0.00

[8.0 7.5 7.0 6.5 7.0 6.5 6.0 5.5 6.0 6.5 7.0 6.5 7.0]  Winnings: $500.00

[1.0 1.5 1.0 1.5 2.0 2.5 3.0 3.5 3.0 3.5 4.0 4.5 4.0]  Winnings: $10000.00
```

Test Case #3:

Input: (using fixed seed of x) 2, 2, 4, 0

Output: Total winnings: $1000.00 Average winnings per chip: $500.00

<u>4. Variables and Constants</u>: Applies to labs 2-10

When you program, you will often store information (names, numbers, etc.) for subsequent use. To do this, you will create variables. These variables will hold the data you wish to use. Sometimes you will store values into these variables before even running the program; sometimes these variables will only have real meaning after a user inputs a value.

a) Variable/parameter names must be written in camelCase (think of the hump in a camel's back): the variables' beginning word is not capitalized, but the other words are. Alternatively, you may use all lower case with words separated by underscore characters. We show camelCase in our other example in the style guide, but you may choose either option for your variables, but once you choose an option, stick with it for the entire program.

DO:

```
int numPeople;

int num_people;
```

DON'T:

```
int NumPeople;

int numpeople;
```

b) Variable names should be descriptive and useful. When coding, you will often create a program that has various options. Variable names such as "firstOption", "first", "seven", or "optionA" are not useful; to make sense of these names you need other knowledge, which is not discernible from the name alone. Instead, give your variables names such as "averageClassSize" or "exitOption". Furthermore, avoid using abbreviations and single-letter variables; a good practice is to think "Would a random person understand what this variable name represents?" Variable names are usually at least two words, except in exceptional cases like indexes (e.g. i and j).

DO:

```
int numberOfBoxes;

char userName;
```

DON'T:

```
int boxes;

int b;

char k;
char usrKey;
```

c) You should use const for any variable whose value will not be changing. Constants are written in upper case with underscores (and are usually at least two words, except in a few obvious cases like PI, etc.).

DO:

```
const int MAXIMUM_WEIGHT = 300;
```

DON'T:

```
const int MAXIUMUMWEIGHT = 300;

const int maximumWeight = 300;

const int MAXIMUM = 300;
```

d) Always initialize a variable or constant (meaning give it value) when it is first declared.

DO:

```
int numPizzas = 0;

char userKey = '-';
```

DON'T:

```
int numPizzas;

char userKey;
```

Notice that these variables were defined but not given values in the "Don't" version. Avoid this. In some cases it will not seem necessary to initialize the variable. However, for example, by initializing numbers to 0 you can avoid potential difficult bugs when your code is mistakenly using an uninitialized variable which contains an unpredictable value. Some types, like "char", may not have a canonical agreed upon initial value, but by you choosing a consistent initial value that you use, you can avoid the difficult bugs coming from unpredictable values. Some types, like strings, are automatically initialized. Thus, you do not need to initialize them.

e) Variables are declared at the top of their enclosing function (including main) except iterator variables in for loops; for example:

```
for (int i = 0; i < MAX_PIZZAS; ++i) {
```

Always declare "for loop" iterator variables in the for as shown above, except in cases where the iterator variable is needed after the loop.

See the functions section below for variable and constant style when there are multiple functions.

5. Constants and Magic Numbers: Applies to labs 2-10

Numbers other than 0 and 1 should rarely occur in the body of your code. Use constants or variables to replace "magic numbers." Using constants has two important advantages. First, they make the code easier to read. Your code should read more like English prose with numbers replaced by descriptively named constants. Second, in cases where the programmer wants to later change the value of a constant, it need only be changed in one place at the declaration, and it will be updated in all places throughout the program.

For example, you would never have a line of code in you program like:

```
totalCost  = numberOfLargePizzas  * 14.88;
```

In this case 14.88 is a magic number, where someone reading the code might not know what the 14.88 stands for. Rather, you would declare and initiate a const variable at the top of your function and then later use it in your code.

```
const double COST_OF_LARGE_PIZZA = 14.88;

totalCost  = numberOfLargePizzas  * COST_OF_LARGE_PIZZA;
```

As discussed above, doing this 1) makes the code easier to read, and 2) if/when later we need up update pizza cost, we need only change it at the declaration.

The only exception for using arbitrary numbers in the body of your code is for common mathematical formulas such as:

```
circumference = 2 * PI * radius.
```

In this case you would be hard pressed to even think of a good name to replace the 2, and if you did it would probably make the formula less readable. Also, since it is a mathematical axiom, we would not be changing the value in the future, thus we would gain neither of the advantages of constants. These situations occur rarely in our labs (in Pizza lab you do find the area).

There are occasional debatable cases such as in the Hot Plate lab where you take the average of your 4 neighbors:

```
average = (northNeighbor + eastNeighbor + southNeighbor + westNeighbor) / 4;
```

In this case, either using 4 as is or replacing 4 with a constant such as NUM_NEIGHBORS could theoretically be reasonable. However, in this course (with our style guide), whenever there is a debatable situation like this, use a constant! (Note: One reason we have this style guide is so that TAs will know how to grade and students can know what to expect. Thus, use a constant in a debatable case and you will be all right.)

0/1 Exceptions: 0 and 1 will often occur in your code but only in the following cases:

- Initializing variables when declared, including iterator variables in loops
- Initializing/re-initializing counts or indices
- Incrementing/decrementing variables – although whenever possible use ++ or --
- Simple boundary and index adjustment of variables (e.g. for (unsigned int i = 0; i < str.size() – 1; ++i))
- return(0) or return(1) at the end of main.

Note that just because a value is 0 and 1 does not mean it should never be a constant. In fact, in cases except for the above, a 0 and 1 should be a constant. For example, in main lab #2 (Pizza) the spec states that 1 large pizza feeds 7, a medium 3, and a small 1. 7 and 3 are obvious magic numbers. But so is 1, both because the meaning of 1 in the coding context may not be obvious, and we may want to change the number of people that a small pizza feeds.

Be careful NOT to use constants inappropriately as you will lose points if you do. Two common mistakes beginning students make are:

1. const int ZERO = 0; This does nothing of use. It does not make the code more readable (0 and ZERO are equivalent) and it does not make things more adaptable. You would never want to later set ZERO = 1!!
2. const int FIVE_YEAR_LOAN = 5; Note in this case the variable cannot be changed to another value (e.g. 10) without needing to change the name of the variable. Instead, name it something like LIFE_OF_LOAN, which is both descriptive AND can be changed later without needing to change the name.

Besides the obvious uses of const as in the COST_OF_LARGE_PIZZA example above, examples of other places where you would use a constant include:

- Random number functions: a constant seed if using one, the random range and the offset, e.g. rand() % RAND_RANGE + RAND_OFFSET
- setprecision, setw, iomanip operators
- Comparing floats e.g. if (x – y < FLOAT_EPSILON)
- User input options. Rather than if (userOption == 0), use if (userOption == QUIT_OPTION), with constants for QUIT_OPTION, PRINT_OPTION, etc.
    - Note that there are cases where you could have “magic strings.” If for example you used (userOption = “quit”), “quit” is a magic string. In most of these situations the string is already readable, and just used in one place, so we will not require you to worry about magic strings, but we wanted to mention them so you are aware.

In summary, numbers should not occur in your code except in const declarations and for the exceptions discussed above.

6. Miscellaneous: Applies to labs 2-10

a) Self documenting code: An important goal is to have your program be “Self-Documenting” such that a person reading your code can understand what it is doing without need of extra comments.

Comments however are important. Your code should have clear concise comments that explain what each section of code does. Comments should not be applied to every line or even every few lines of code, but instead to sections or functions of code that summarizes what that specific section or function does. In those few areas of code where a particularly complex algorithm was applied, you should state why it was implemented that way. But comments should never be used as an excuse to write bad or complex code that could be written better or simpler otherwise.

b) Do not use global variables. Global constants are OK.

c) Never use the C++ command “goto” as this can lead to serious problems in the code. If you use it you will lose all 25 qualitative points for the lab!

d) Use "endl" rather than "\n" (except in lab 1).

e) Make sure you end your "int main()", which is your main code function, with "return 0;". This signals that your program finished correctly.

7. Functions: Applies to labs 6-10

Functions should:

1. Have a descriptive name for the function and the function parameters.
2. Be short, typically less than 20 lines and no more than 40.
3. Do one task. For example, you would not want a function which both a) returns the total number of words in a document and b) capitalizes first letters of all the words. Those should be done by separate functions, making them simpler and more general to use in the future.
4. Be structured to avoid duplication of code. Thus, you should not have places in your lab where you have duplicated snippets of code many times, which would have been better handled by a function call each time.

5. If you pass in a large data structure by reference (e.g. vector, array), which will not be updated, then pass it in as const. You should name these formal parameters as with any normal variable and NOT with all CAPS as that type of constant is somewhat different.

With respect to items 2 and 3, note that one issue here is the complexity of the code. Generally shorter code is simpler to write and easier to understand than longer code. However, code that is a bit longer and simple to understand is better than being shorter but difficult to understand. Also, do not try to shorten code just by removing the whitespace used to make the code more readable (see the Format section).

Constants and variables are declared at the beginning of their enclosing function (including main), or at the top of the program when the constant is global. You will not use global variables. If the constant is local it should just be declared in that function.

Function names are CamelCase with uppercase first, <u>or</u> with lower case letters and words separated by underscores.

DO:

```
NumberOfCharacters(string inputString)

number_of_characters(string input_string)
```

DON'T:

```
nChars(string s)

Num_Characters(string str)
```

Make sure that your naming of functions and parameters are sufficiently clear that you rarely need a comment header to describe the function and parameters.

<u>8. Objects and Classes:</u> Applies to labs 8-10

- Class naming is CamelCase with uppercase first. Member functions follow the same naming rules as functions.
- Always create your own default constructor for each class, and create a parameterized constructor(s) when needed (usually). You may combine these into one constructor function by creating a default constructor with default parameters.
- Any member function which will not mutate data members should be declared as const.
- All data members and helper functions should be private.

Advice Section

In this section we mention style issues which you should follow, but for which we will not have the TAs grade. If we see that they are often violated we will add them to the "to be graded" portion of the style guide.

- A blank line separates distinct groups of statements; however, related statements do not have blank lines between them.
- Use break (and continue) very sparingly. Break can be used in switch statements. They can also be used when significant advantages are gained in time optimization or program flow, but note you can often accomplish the same thing in other ways.
- You do not need to try to align all your variable names in your declarations as that alignment usually changes if you go from one IDE to another.
- With void functions, return can be used for returning early in the function, but we do not need to put a "return;" at the end.

STYLE POINTS SUMMARY

1. Header Comments (Labs 1-10)

- Are all required names and descriptions listed?

2. Format (Labs 2-10)

- Are statements on their own lines?
- Are “if” and “while” statements "K&R" style or "Stroustrup" style?
- Are sub-statements consistently indented with either 3 or 4 spaces?
- Is there exactly one space between operators, =, and other appropriate items?
- Are “if-else” statements "Stroutrup" style?
- Even if there is just one sub-statement, are braces used?
- Are line lengths kept reasonably short?

3. Test Cases (Labs 2-10)

- Are there three diverse original test cases, demonstrating unique tests?

4. Variables and Constants (Labs 2-10)

- Are the variables camelCase (with the first letter lower-case) or with lower case letters and words separated by underscores?
- Are the variable and constant names descriptive, and more than two words except in obvious cases?
- Are the constants all caps with underscores between words?
- Are variables and constants initialized when they are declared?
- Are variables and constants declared at the appropriate place in the program?

5. Constants and Magic Numbers (Labs 2-10)

- Do numbers occur in the code, except in const declarations and for the exceptions discussed?

6. Miscellaneous (Labs 2-10)

- Is the code self-documenting with rare comments if needed?
- Are there any global variables?
- Is “goto” avoided? (25 points off if used)
- Does the code use "endl" rather than "\n" (except for lab 1)?
- Does your program end with "return 0;"?

7. Functions (Labs 6-10)

- Are the function names (CamelCase with uppercase) or with lower case letters and words separated by underscores, and are parameters descriptive?
- Are the functions less than 40 lines?
- Do the functions only do one task?
- Is duplication of code avoided by using functions?

8. Objects and Classes (Labs 8-10)

- Are class names (CamelCase with uppercase)?
- Have you created your own default constructor?
- Are member functions which will not mutate data members declared as const
- Are all data members and helper functions private?