IST163-C++ Programming Style Guide

The purpose of this document is to provide you with a set of guidelines to following concerning the formatting and documenting of C++ programs written for IST courses. The goal is to create programs that are easier to read, understand and, most importantly for programming and development jobs in the real world, easy to maintain.

Documentation/Commenting

Take appropriate (or give appropriate) credit for your work. Each file should begin with a comment section giving the name of the file, the author's name, the date the file was last modified, and a description of its purpose. In addition, it is also consider good programming practice to include any assumptions you have made (such as assumptions about reasonable input) and any error-checking that has been provided. For example,

```
/*****************************
Name: Brian Morgan
Course: IST 163 – Programming Practicum w/C++
Date: January 11, 2014 (due date)
*************************
Files for Assignment: lab1-1.cpp, etc.
Purpose of File: this is a generic example of a C++ program
Assistance Received:  Jennifer Morgan, Michael Morgan
*****************************/
```

Each function prototype and function definition should be fully commented. The comments should include a summary of what the function does (when necessary), its preconditions (conditions assumed when the function is called) its postconditions (describing the effect of the function call), and the dates the function was coded. Below is a sample structure of the documentation that should accompany each function.

```
/*****************************
Function: Name
Precondition: what has to be true before the function executes, & the name
and type of inputs in the form
     name - definition
     name - definition
Postcondition: what the function does (purpose), as well as the name and type
output in the form
     return type, name - definition
Dates of Coding:
Modifications: Any listing of major modifications made to code after first
debugging
*************************/

EXAMPLE:
/*****************************
Function: Fahrenheit_to_Celsius
Precondition: the value of intTempFar must be set prior to the function call
              intTempFar – temperature in degrees F that will be converted
Postcondition: this function will convert the temperature from degrees F
               (intTempFar parameter) to degrees Celsius & return the degrees
               Celsius to the calling function.
Dates of Coding: January 13, 2003
Modifications: None
*************************/
double Fahrenheit_to_Celcius (double intTempFar);
```

Additional Documentation Tips

- Your program should be self-documenting; that is, a listing of your source code should provide the key information in English, such that the information will help others (and often yourself!) understand your program.

- You should document heavily those portions of programs where you spent more than 15 minutes thinking about its logic to help you better understand the code and your thoughts if there should happen to be an error.

- Insert comments in your program. A comment should appear after each declaration of variables, unless the names of variables have clear enough meaning to what they should contain, which is a goal of good programming. Comments should also appear before loops or decision structures, and after groupings of statements to provide information which is not obvious. For example:

```
int intTotalCost
int intItemCount;
double dblAverageCost;          //average cost of an item, to be computed
```

  You will also use comments to precede a block of code:

```
//perform linear search of costs array for a matching value
int i = 0;
while ( i < SIZE && costs[i] != search_val )
        i++;
```

Naming Conventions
Choosing good identifier names is more of an art than a science, but here are some guidelines that may help.

Variables
Variables should be assigned meaningful names. Defining variables with names like X, Y or Z, gives no insight into how the variable is to be used or the types of values that can be assigned. Although it may take a little longer to type out a descriptive variable name, doing so can save hours of frustration when trying to debug a program that doesn't work or when trying to modify a program that you haven't worked on for awhile.
Examples: a variable to store a person's first name – strFirstName. A variable to store a person's salary – dblSalary;

One exception to the descriptive naming rule is that variables used solely for iteration purposes (an index in a for loop), may be named with single letters. Preferred letters for this purpose are i, j, and k. A comment should exist where these variables are declared that explains that they are for use as index variables. Such variables should never be used for any other purpose.

Constants
Constants should be given names in ALL CAPITAL LETTERS (after the type identifier). Constants that apply only to a specific class should be defined within the class, to ensure that they are not overlooked by programmers wishing to reuse your class. Such constants should be defined as "static const" values.
Example: The tax rate for a given application's calculations – dblSALESTAX = .065;

More about Constants
Programs should not have "magic numbers" in them. An example of a magic number is the number 1.07 in the following statement:
dblFinalCost = dblPrice * 1.07;

What is 1.07? In this case, it is the sales tax rate for the region that this program is being used in. What happens if someone somewhere else wants to use the program but their sales tax is different? Well, they have to first figure out that your sales tax rate is 7 percent, then figure out that you compute the finalcost by multiplying the price by 107%. Finally, they have to look for and change every instance of 1.07 in your program. In instances like this, it is best to represent values that may change, or standard values like PI, with constants.

Unless it is blatantly obvious why a numeric constant is being used (for instance, if you're trying to determine if a number is evenly divisible by 2), a constant should be defined that has a meaningful, descriptive name. This makes your programs easier to understand by others (and yourself in the future) and also makes them easier to maintain, should the sales tax rate change. A better version of the example statement is:
dlbFinalCost = dblPrice * dblSALESTAX;
where dblSALESTAX is a constant defined somewhere in the program.

Functions
Functions that you write should have uppercase letters at word boundaries. Some examples are SelectionSort, GetMenuSelection and Usage. This distinguishes your functions from standard library functions and also prevents variable name and function name conflicts.

Reserved words
C++ is a case sensitive language, so all built-in terms and library functions that you access must match exactly with their C++ definitions. By default, all C++ reserved words are in lower case.

Classes
Classes that you define should contain the word class to designate that your class will represent a new Type. All classes should be defined using the "class" keyword (not as struct). The opening brace should be on the following line, by itself, and placed directly under the 'c' in "class." The public, private and protected keywords should all line line up even with the brace. Constructors, Modification, and Constant member functions should all be grouped together in their appropriate sections (public/private/protected). A comment should precede each of these groupings. Member variables should be grouped together and not intermixed with member functions. Member functions and variables should be indented two spaces in from the section tag (public/private/protected).

Abbreviate with care
Don't think you're saving time for the reader by removing all the vowels from a name. Some abbreviations (min , max , num , ptr -- for "pointer") are so common that they're considered OK. When combining several words, capitalize the first letter of each word after the first word for variables. Constants should be all capitals with an underscore separating the combination of more than one word. All declarations should have only one identifier per line with a comment explaining the purpose and whether it is a Constant, Input, Output, or Processing identifier. **No Global variables are allowed** unless you cannot avoid them (i.e., when creating a multithreaded program). When in doubt, **PLEASE ASK!**

```
char chrSelection;                  // Input – menu selection from the user
int i;                              // Processing – loop counter
double dblAverage;                  // Output – average grade
const int intTEN = 10;              // Constant - multiplier
const double dblSALESTAX =  0.65;   // Constant – amount of state sales tax
```

Boolean variables and boolean function names should state a fact that can be true or false
That way, they will read smoothly in an **if()** or **while()** statement. Use of the verb "is" can be quite helpful. For example:

if (stringIsMissing)

Class names should be nouns. Member functions names should be verbs. For example:

```
class counter_class
{
public:
      void incrementCounter ();
      int getCounter();
private:
      int intCount;
};
```

Formatting Your Code

Line Wrapping

Try to avoid this when at all possible, but sometimes there's nothing that can be done to avoid this. When it happens, there are two options:

1. Write helper functions or use temporary variables to hold intermediate results.
2. Break the expression up over several lines. Do this at the operator boundary:

```
dblTax = (dblIncome * dblTAXRATE)
      + dblStateTax + dblLocalTax
```

(Note that the operators are on the left and that the continued expression is indented.) When **cin** and **cout** statements get too long they can also be broken up. (Note where the semicolon falls.)

```
cout << "At " <<  dblTime  << " The temperature is "  << dblTemperature
     << " and the humidity is " << dblHumidity << endl;
```

Expressions

All operators should be surrounded by spaces, with the exception of unary operators (operators that work on just one argument.) This includes **++, --, !,** etc.

```
intSum = a + b;
intSum += 12;
```

NOTE: Long math expressions can lead to potential stack overflows, so please ensure your formulas are correct if you are experiencing run-time errors.

Indentation

To improve the readability of programs, and to aid you in debugging your code, there should be indentation that indicates different blocks of statements (see the documentation example at the end of this document).

Curly Braces

There are two types of C++ programmers. One type puts the opening curly brace at the end of the line that starts the block of code. For example:

```
for (i = 0; i < itrMAXITER; i++) {
  statements;
  if (x) {
    statements;
  }
}
```

The other type puts the curly braces immediately beneath the line, lined up at the same indentation level. For example:

```
for (i = 0; i < itrMAXITER; i++)
{
  statements;
  if (x)
  {
```

```
    statements;
  }
}
```

This latter type, lining the opening and closing braces up beneath the control construct, is the preferred method. First of all, it is easy to see that every opening brace has a closing brace at the same level of indentation. This makes it easier to determine which block a brace is closing. Also, it is MUCH easier to verify that every closing brace has a matching opening brace. You may think you have started a block but may have inadvertently left off the opening brace at the end of a statement. You don't want to spend hours trying to figure out why, such as the code below:

```
for (i = 0; i < itrMAXITER; i++)
  do something in the loop;
  A couple hundred lines of code;
  do something else in the loop;
}
```

won't compile.

If you do use the first method, be sure to put in the beginning and ending braces (same for using parentheses) at the same time so that you know you have a matching set to work with.

Parentheses and Precedence Order

The C++ operator precedence order is a rather tricky thing. Rather than relying on obscure rules of precedence, mathematical and relational expressions should make use of parentheses to clarify how the expression is to be evaluated. For example, the statement:

a = x - y * z;

should be written as:

a = x - (y * z);

rather than relying on precedence rules. Although the program will compile and run properly, you or someone required to maintain your program may forget the order of evaluation, or simply make a mistake, and cause the program to stop functioning properly. While the previous example is simple and most people remember Please My Dear Aunt Sally, the following statement is a little less easy to understand:

x++ = y - z * k % 2 + ++z << 2;

Programming Style Tips

Software developers should write code that is:

- Reliable
- Easy to understand
- Robust (considers all contingencies)
- Easy for other developers to use
- Easy to maintain and update

Coding standards and guidelines help to achieve these goals.

General

- C++ programs should be written in a straightforward manner. Good programmers strive for simple, easy-to-read, and understandable code. Avoid bizarre or overly cute usages.
- Each function should perform a single, well-defined task.
- Replace repetitive statements by calls to a common function whenever the function-call overhead is not too costly in terms of efficiency. This will improve readability and maintainability (easier to find bugs, make changes, etc.) of the code.
- In general, opt to pass modifiable arguments by reference, small non-modifiable arguments by value, and large non-modifiable arguments by using references to constants. This balances performance and clarity since call-by-reference exposes an argument to corruption and the

copying involved in call-by-value of large objects can degrade performance. That is, for passing large objects, use a constant reference parameter to simulate the security of call-by-value and the efficiency of call-by-reference.

Formatting (reiterated)

- Group related code together.
- Use consistent and reasonable indentation conventions throughout your program to improve program readability. Avoid using several spaces for an indent - instead use a tab. If there are several levels of indentation, each level should be indented the same additional amount of space.
- Indent the body of a control statement (if else, switch, for, while, do while) to emphasize structure and enhance readability.
- Place braces {} to emphasize program structure.
- Nested *if* statements tend to quickly migrate to the right following a simple indent each control structure approach. If you have a situation which requires multiple nested if statements, instead of doing:

```
if ( dayOfWeek == 1 )
    // do this;
else
   {
   if ( dayOfWeek == 2 )
       // do that;
   else
      {
      if ( dayOfWeek == 3 )
          // do something else;
       etc...
```

  Do this instead:

```
if ( dayOfWeek == 1 )
   // do this;
else if ( dayOfWeek == 2 )
   // do that;
else if ( dayOfWeek == 3 )
    // do something else;
```

  Or, use a **switch** statement. Either will save space and be easier to read, while producing the same results as the nested ifs.
- Add white space (i.e., blank lines and or spaces) to improve readability. In general:
  - Use one blank line to separate logical chunks of code. Avoid using more than one blank line.
  - Always place a blank line before a declaration that appears between executable statements. If you prefer to place declarations at the beginning of a function, separate them from the executable statements with a blank light. This highlights where declarations end and executable statements begin.
  - Put a blank line before and after each control structure.
  - Place a space after each comma (,) to make programs more readable.
  - Place a space on either side of a binary operator. This makes the operator stand out and the program easier to read.
  - An fully documented program is at the end of this document to provide you with an example of the above
- Declare each variable on a separate line. This format allows for placing a descriptive comment next to each declaration.

Global vs. Local Variables

- Declare variables to be local to the function(s) in which they are used. Global variables and constants are counter-productive to the goal of encapsulation. Also, a global variable runs the risk of being inadvertently altered in a function in which it is not meant to be used.
- Place named constants as local as possible. When then should a constant be made global? There is no hard and fast rule to apply here. Judge the use of the constant - if it is used in many functions throughout the program (e.g. the tax rate in an accounting program) then it makes sense to declare the constant globally at the beginning of the file.
- Do not use global variables to get around the need to pass parameters.

Code Efficiency

- A nested if/else statement can be much faster than a series of non-nested if statements because of the possibility that an early if condition is satisfied.

Classes

- It's considered better style to list all the public members of a class first in one group and then list all the private members in another group. This tends to focus the attention of the user of the class on the public interface rather than on the inaccessible implementation.
- Every class should have a constructor to ensure that every object is initialized to a well-defined state.
- Declare as const all member functions that do not need to modify the current object so that you can use them on a const object if necessary.

Full Documentation Example for C++ Projects

```
/*****************************
Name: Brian Morgan
Course: IST 163 – Programming Practicum w/C++
Date: January 11, 2014 (due date)
**************************
Files for Assignment: lab1-1.cpp, etc.
Purpose of File: this is a generic example of a C++ program
*************************************************/

//header information
#include <iostream>
using namespace std;

/**************************
Function: Main
Precondition:
     none
Postcondition:
     Demo C++ simple functions, program's main function
     no returns - displays to the screen only
Dates of Coding Modifications: January 10, 2003
**************************/
void main()
{
     // variable declarations and short meaning
     int intCurrCard;  // holds the current card value
     int intCardTot;    // holds the total of your hand

     // prompt the user for the input value of the first card
     cout << "Please enter the value of your first card: ";
     // read in first card's value
     cin >> intCurrCard;
```

```
    // if card was 1, set to 11 (Ace)
    if ( intCurrCard == 1)
         intCurrCard = 11;

    // add current to the total for your hand
    intCardTot = intCurrCard;

    // loop while the total in the player's hand is less than 17
    while( intCardTot < 17 )
    {
        // ask for another card
        cout << "Please enter the value of the next card: ";
        // read in another card integer
        cin >> intCurrCard;

         // if card was 1 and current total is 10 or less, set to 11
         if( intCurrCard == 1 && intCardTot <= 10 )
              intCurrCard = 11;

         // add current to total
         intCardTot = intCardTot + intCurrCard;
    }

    // if over 21, busted
    if( intCardTot > 21 )
         cout << "Busted - You win" << endl;
    // else, hold
    else
         cout << "Sitting - show me your cards" << endl;

    // exit the program gracefully
    cout << "Type a letter and press the Enter key ";
    cin.get();

    // return from main to exit the program
    return;
}
```