COMP 2710
C++ Programming Style Guide

Introduction

As you are aware, compilers do not care what your source code looks like as long as it is correct syntax for the language. So, why does it matter what the source code looks like? It matters because humans will also be reading your source code. In this class your program needs to be easily understood by you and by your instructor/grader. In other environments your program will need to be understood by other members of your programming team, by members of a testing team, and by programmers who will be maintaining your code after you leave a project. Thus, good programming style is an important ingredient to good software engineering. The more readable your code, the more efficient the software production process will be.

Think of your source code as a document describing your implementation. Like writing a paper, article, or blueprint, you have the freedom to make choices regarding how to present your document. However, you also know that if you stay within the guidelines of popular convention for the type of document you are creating, then people reading your document will have an easier time understanding it. There is a tension between your creative freedom and the conventions dictating your style. It is your responsibility to find an appropriate balance.

What is Style?

Style is the format or layout of a program as commented source code. In the context of COMP 2710, we will concentrate on the following aspects:

- Indentation
- Placement of Curly Brackets
- Whitespace
- Comments
- Identifier Names

Indentation

Fixed Width Indentation

Almost all indentation occurs in fixed width intervals, usually using an interval of a single tab or two or more spaces. The following example program is indented by four spaces per indentation.

```
// Program 0:    The Bean Counter
int main ()
{
    int number_of_beans;                 // Number of beans (from user)
    const double BEAN_WEIGHT = 2.051;    // Average bean weight in grams
    cout << "Please enter the number of beans" << endl;
    cin >> number_of_beans;
    if (number_of_beans >= 0)
    {
        cout << "Number of Beans: " << number_of_beans << endl;
        cout << "Average Weight: " << BEAN_WEIGHT << endl;
        cout << "Total Weight: " << number_of_beans * BEAN_WEIGHT << endl;
    }   |
}   |   |
|   |   |
|   |   |
|   |   +---- Level 2 (indented eight spaces)
|   |
|   +-------- Level 1 (indented four spaces)
|
+------------ Level 0 (not indented)
```

I don't care how much you indent, but you must be consistant in your indenting. If you indent four spaces from level 0 to level 1, you must indent four more spaces from level 1 to level 2. Furthermore, if you indent four spaces anywhere in your code, you must indent four spaces everywhere in your code.

Level 0

The following items begin at level 0:

- Preprocessor directives (e.g. `#include`)
- Function Headers and Class Headers
- Global Variables (and other global declarations)

Level 1+

Other statements are indented. Indent once for being in a function, and once more for each control statement the statement is nested within. Always indent whether or not the control statement uses curly brackets. It is not necessary to indent case labels within a switch statements and public/private labels within class definitions, but you may do so if you wish (see below).

```
if

    if (beans == bacon)
    {
        beans++;
        bacon++;
    }


if/else

    if (beans == bacon)
        beans++;
    else
        bacon++;


while

    while (beans == bacon)
        beans++;


do while

    do
    {
        beans++;
    } while (beans == bacon);


switch (#1)

    switch (number_beans)
    {
        case 0:
            cout << "What?  No beans???" << endl;
            break;
        default:
            cout << "Ahhhh" << endl;
            break;
    }


switch (#2)

    switch (number_beans)
    {
    case 0:
        cout << "What?  No beans???" << endl;
        break;
    default:
        cout << "Ahhhh" << endl;
        break;
    }
```

Placement of Curly Brackets

There are several common styles used for lining up curly brackets. The most important thing is to be consistent. If you use one style anywhere in your code, you must use the same style everywhere in your code. I prefer the following style:

```
Option 1

    header
    {
        body
        body
    }
```

Whitespace

Vertical Whitespace

A program should have two or more blank lines between functions. Further, it should have a single blank line to break up statements within a function into logical units.

```
    // Function:     Eat_Beans
    // Description:  Notice how there are two blank lines between this
    //               function and the next function.
    void Eat_Beans (void)
    {
      cout << "Burp!" << endl;
    }

    // Function:     Count_Beans
    // Description:  Notice how the two logical steps of the algorithm
    //               are separated using blank lines.
    int Count_Beans (Carton carton)
    {

      // Sample the beans database to get the average weight
      int i;
      double average = 0;          // Average weight within sample
      for (i = 0; i < SAMPLE_SIZE; ++i)
        average += beans_database[rand() % NUM_BEANS].weight();
      average /= SAMPLE_SIZE;

      // Return total number of beans in carton
      int total_beans = int (Carton.weight() / average);
      return total_beans;
    }
```

Horizontal Whitespace

There should be a single space between most operators and their operands. For example:

```
      // Do it like this:
      int i = 5 + 8 * (9 - 3);
      cout << "Hi";

      // Not like this:
      int i=5+8*(9-3);
      cout<<"Hi";
```

But some operators, like ++, [] and unary -, generally do not take horizontal whitespace.

```
      // Do it like this:
      beans[i++] = -j;

      // Not like this:
      beans [ i ++ ] = - j;
```

Comments

Program Header

Each program should have a block of comments at the top of the file that contains the main() function. These comments must contain the name of the program, your name, your course and section, the date and your email address. In addition, you must include a brief summary of the program that describes what the program does, what kinds of inputs are necessary and any bugs or deficiencies.

```
    // Program 0:    The Bean Counter
    // Name:         Bob C. Doe
    // Class:        COMP 2710 Section 1
    // Date:         Sept. 8, 2014
    // E-Mail: doebob@auburn.edu
    //
```

```
// Description:  This program calculates the number of navy beans
//               in the known universe based on a user's estimate
//               of the universal navy bean constant.
```

File Header

Every other file in a multi file program should have a header. The file header should include a description of what is contained in the file, and how the code in the file relates to the rest of the program.

```
// Program 0:    The Bean Counter
// File:         Bean.h
// Description:  Class definition of the Bean class.
```

Function Header

Most functions should have a header. The function header explains the inputs, outputs and side effects of the function, as well as a short description of what the function does.

```
// Function:     Count_Beans
// Inputs:       Weight of beans (in tons)
// Outputs:      Number of beans
// Description:  Calculates the number of beans based on the
//               average weight of a navy bean at sea level.
int Count_Beans (double weight) {
  return int (weight / Bean::Average_Weight());
}
```

A related group of small functions can share a single comment.

```
// Function:     User Input Routines
// Description:  These utility routines control simple user inputs.
int getInt (char *prompt) {
  int i;
  cout << prompt;
  cin >> i;
  return i;
}

char getYN (char *prompt) {
  char c;
  cout << prompt;
  cin >> c;
  while (c != 'n' && c != 'N' && c != 'y' && c != 'Y') {
    cout << "You fool.  You enter Y or N! ";
    cin >> c;
  }
  return c;
}
```

Comments by Object Definitions

Most objects need a comment to describe what the variable does/is.

```
double weight;        // weight of beans (in tons)
double mass;          // mass of beans (in kg)
```

Comments Embedded in the Code

Comments should be embedded in code for the following reasons.

- To introduce a logically distinct section of code.
- To explain an unusual coding technique.

The following is an example of a function which contains embedded comments.

```
// Function:     Count_Beans
// Description:  First take a sample of the beans database to aproximate
//               the weight, and then use this to estimate the total number
//               in the carton.
//               Return the number of beans.
int Count_Beans (Carton carton) {
  // Sample the beans database to get the average weight
  int i;
  double average = 0;         // Average weight within sample
  for (i = 0; i < SAMPLE_SIZE; ++i)
    average += beans_database[rand() % NUM_BEANS].weight();
  average /= SAMPLE_SIZE;

  // Return total number of beans in carton
  int total_beans = int (Carton.weight() / average);
  return total_beans;
}
```

Identifier Names

Naming Conventions

Most identifiers, including most variables and all functions and classes, should have a descriptive name. The name need not be long, but it must relate to the semantic meaning of the identifier.

```
  // Do it like this:
  double starting_location, time, velocity;
  double location = starting_location + time * velocity;

  // Not like this:
  double s, t, v;
  double l = s + t * v;
```

But on the other hand, you should use short (1 letter) identifiers for temporary variables, loop variables and input variables.

```
  // Do it like this:
  int i;
  for (i = 0; i < MAX_BEANS; i++)
    beans[i] = new Bean;

  // Not like this:
  int current_bean;
  for (current_bean = 0; current_bean < MAX; current_bean++)
    beans[current_bean] = new Bean;
```

And finally, please use all capital letters for constants.

```
  // Do it like this:
  const int MAX_BEANS = 100;

  // Not like this:
  const int max_beans = 100;
```