C++ Coding Standard

Wei Kian Chen, PhD
Brian R. Hall, ScD
Li Jun Zhang, PhD

Revision Date: June 25, 2014

Table of Contents

1. INTRODUCTION ... 1
2. DOCUMENTATION ... 1
2.1 Header Documentation ... 1
2.2 Comments ... 1
2.2.1 Block Comment ... 1
2.2.3 Single Line Comment ... 2
2.2.4 Commenting a Segment of Code ... 2
2.2.5 Commenting a Block of Variables ... 2
2.2.6 Commented Blocks of Code ... 2
2.2.7 Trivial Comments ... 2
3. SPACING ... 3
3.1 Indentation and Spacing ... 3
3.2 Line Wrapping ... 5
3.3 Blank Lines ... 6
4. NAMING CONVENTIONS ... 6
4.1 General Requirements ... 6
4.2 Constant Identifiers (including enumerated constant) ... 7
4.3 Variable Identifiers ... 7
4.4 Functions ... 7
4.5 Structure: struct ... 7
4.6 Class ... 7
5. FUNCTIONS ... 8
6. DATA TYPES ... 8
6.1 Struct ... 8
6.2 Class ... 9
7. REFERENCES ... 9

1. INTRODUCTION

This document describes the standards that all students **MUST** follow for programming Labs and Assignments. This document may be amended as needed during the course of the semester.

2. DOCUMENTATION

2.1 Header Documentation

Each source file within the project must bear the following header documentation:

```
Author:             Your-Name
Class:              Course_Code-Course_Number-Section_Number
Assignment:         PA x or Lab x
Date Assigned:      Date-Assigned
Due Date:           Due-Date and Time

Description:
   A brief description of the purpose of the program.

Certification of Authenticity:
   I certify that this is entirely my own work, except where I have given fully-
   documented references to the work of others. I understand the definition and
   consequences of plagiarism and acknowledge that the assessor of this assignment
   may, for the purpose of assessing this assignment:
      -  Reproduce this assignment and provide a copy to another member of
         academic staff; and/or
      -  Communicate a copy of this assignment to a plagiarism checking service
         (which may then retain a copy of this assignment on its database for
         the purpose of future plagiarism checking)
```

- If section number is not available, it can be omitted.
- Example for Class: CSI-140-01.
- The certificate of authenticity is your signature that indicates this is your own code. Your program will not be graded without this signature.

2.2 Comments

2.2.1 Block Comment

Use /* */ to comment out a multiple lines of information. Example:

```
/*
  The purpose of this code is to compute the average grade for a student
  where assignment worth 30%, midterm worth 30% and the final worth 40%
*/
```

2.2.3 Single Line Comment

Use // to comment out a single line of information. Example:

```
 // The following code will prompt the user for information
```

2.2.4 Commenting a Segment of Code

Each section of code should be preceded by an inline comment. Example:

```
// collect package information
cout << "Enter package width:  ";
cin >> width;

cout << "Enter package length:  ";
cin >> length;

cout << "Enter package height:  ";
cin >> height;
```

2.2.5 Commenting a Block of Variables

Declaration and other implementation code should be followed by a sidebar comment. Example:

```
double width,
       length,
       height,
       weight;  // package information variables
```

2.2.6 Commented Blocks of Code

All **commented code** must be deleted prior to submission. If you wish to keep sections of code you have attempted or are in development then create a copy of the file, remove commended code, and submit the updated copy.

2.2.7 Trivial Comments

Avoid simply repeating what is in the code with a comment. Example:

```
a = b + c;  // add b to c and place the result in a
```

3. SPACING

3.1 Indentation and Spacing

3.1.1 Whenever a section of code is indented, it must be indented with **tab**. You can set up a default amount of space for a tab (e.g., 3 spaces) for your development environment (Visual Studio, Xcode, etc). Most system defaults are 4 spaces.

3.1.2 Space must be provided **before** and **after** each operator for readability. Example:

- Instead of:
  - i. `cout<<"Enter Name:  ";`
  - ii. `num1=num2+num3;`

- Use spacing:
  - i. `cout << "Enter Name:  ";`
  - ii. `num1 = num2 + num3;`

3.1.3 Block (compound statement) markers ({} – curly braces) are not indented. The left curly brace { and right curly brace } should be placed on their own line.

3.1.4 All code inside a function body or compound statement (such as in a function, loop, **if**, **else**, **switch**, **struct** declaration or **class** declaration) is indented.

3.1.5 All conditional executed code is indented.

3.1.6 There must be a space before and after the bracket of the condition (see do-loop).

3.1.7 **Exception**: nested-if-statements are not indented but the conditionally executed statements internal to it are indented (see nested if-statement).

3.1.8 **Switch-statement**: each case should be indented. The first statement after a case colon (case #:) should begin after the colon and subsequent statements in the case should line up with the first statement.

Examples:

if-statement 1:

```
/*
  Ideally, the number 16 should be a constant.  We will leave it as is for
  simple illustration purposes. The same applies to all examples used in
  this guideline.
*/
if (currentYear - YEAR > 16)
{
   cout << "It must be at least 2001! " << endl;
        << "Please re-enter your input!" << endl;
}
else
{
   cout << "Are you sure about the year? (Y/N)  ";
}
```

if-statement 2:

```
if (grade >= 90)
{
   cout << "Excellent\n";
}
else if (grade >= 75)
{
   cout << "Good\n";
}
else if (grade >= 60)
{
   cout << "Pass\n";
}
else
{
   cout << "Failed\n";
}
```

nested-if-statement:

```
if (num1 < 0)
{
   if (num2 < 0)
   {
      cout << "Case 1\n";
   }
   else if (num2 == 0)
   {
      cout << "Case 2\n";
   }
   else
   {
      cout << "Case 3\n";
   }
}
else
{
   cout << "Case 4\n";
}
```

switch-statement:

```
switch (gpa)
{
   case 0:  cout << "Excellent\n";
            break;
   case 1:  cout << "Good\n ";
            break;
   case 2:  cout << "Average\n ";
            break;
   default:  cout << " Poor\n";
}
```

for-loop:

```
for (i = 0; i < 100; i++)
{
   cout << "i = " << i << endl;
   square = pow(i, 2);
   cout << i << " square is " << square << endl;
}
```

while-loop:

```
i = 0;
while (i < 100)
{
   cout << "i = " << i << endl;

   square = pow(i, 2);
   cout << i << " square is " << square << endl;
}
```

do-loop:

```
i = 0;
do {
   cout << "i = " << i << endl;

   square = pow(i, 2);
   cout << i << " square is " << square << endl;
} while (i < 100);
```

3.2 Line Wrapping

3.2.1 When a statement is too long, wrap them around so that it is easier to read.

3.2.2 Since most development environments do not perform this automatically, you must format manually.

Instead of:

a.
```
cout << "There are a circle with diameter = " << diameter << " is " <<
area << endl;
```

b.
```
gpa = (grade1 * gradePoint1) + (grade2 * gradePoint2) + (grade3 *
gradePoint3) + (grade4 * gradePoint4);
```

Use line wrapping:

a.
```
cout << "There are a circle with diameter = " << diameter
     << " is " << area << endl;
```

b.
```
gpa = (grade1 * gradePoint1) + (grade2 * gradePoint2) +
      (grade3 * gradePoint3) + (grade4 * gradePoint4);
```

3.3 Blank Lines

Use blank lines to identify code sections in order to enhance readability. Example:

```
const double PI = 3.14;

int main()
{
   double radius, area;

   cout << "Enter the radius:  ";
   cin >> radius;

   area = PI * pow(radius, 2);

   cout << "The area is  " << area << endl;

   return 0;
}
```

4. NAMING CONVENTIONS

4.1 General Requirements

4.1.1 Variables start with a lowercase letter.
4.1.2 Numeric values are allowed after the first character of the variable.
4.1.3 Each word after the first word starts with an uppercase letter (only the first letter, camelCase), if underscore is not used to connect the different words.
4.1.4 A name should:

- Be descriptive: short yet meaningful. For example:

  Instead of: `abc, s`
  Be descriptive: `letter, salary`

- Be long enough to avoid name conflicts, but not excessive in length.

  Bad example: `iReallyDoNotKnowWhyAmIDoingThisButIThinkThisWillWork`

4.1.5 Avoid using characters in a name that might be confusing, if possible. For example:

- The letter ‘`O`’ might be read as the number `0` or the letter ‘`D`’
- The letter ‘`I`’ might be read as the number `1` or the letter ‘`l`’
- The letter ‘`S`’ might be read as the number `5`
- The letter ‘`Z`’ might be read as the number `2`
- The letter ‘`n`’ might be read as the letter ‘`h`’

4.1.6 There are two types of words to consider:

i. Common words listed in a language dictionary should never be abbreviated. Do **NOT** use:
   - **cmd** instead of **command**
   - **cp** instead of **copy**
   - **comp** instead of **computer**

ii. Domain specific phrases that are more naturally known by abbreviations/acronyms should be kept abbreviated. Do **NOT** use:
   - **HypertextMarkupLanguage** instead of **html**
   - **SocialSecurityNumber** instead of **ssn**
   - **CentralProcessingUnit** instead of **cpu**

4.2 Constant Identifiers (including enumerated constant)

4.2.1 All letters must be in uppercase.

4.2.2 Underscore is used between words.

Examples: `MAXIMUM, FIST_DAY, MAX_LOAD`

4.3 Variable Identifiers

4.3.1 Avoid one letter variable names, except for loop counters, which are almost always i, j and k.

4.3.2 Different words for the identifier can be combined, but the first character of the second word onward should be capitalized.

Examples: `firstName, lastName, dateOfBirth`

4.4 Functions

Follow the general naming convention (camelCase).

Examples: `sayHi(), sayBye()`

4.5 Structure: struct

The first character of each word must be capitalized.

Examples: `Account, User, Item`

4.6 Class

The first character of each word must be capitalized.

Examples: `Account, User, Item`

5. FUNCTIONS

5.1 Function prototypes should be defined in the header file.[1]
5.2 Function definitions should be defined in the implementation file.[1]
5.3 All function prototypes and definitions must be sorted in ascending order based on function names.[2]
5.4 Functions should be small and concise to avoid developing overly complex functions.
5.5 Functions must be separated at least one blank line.
5.6 Functions must have the following statements.

i. Pre: What are the conditions needed before calling this function?
ii. Post: What are the conditions after the function is completed?
iii. Purpose: What is this function supposed to do?
iv. Author: Who implemented this function? This is needed only if the project is completed in a team setting

Examples:

```
/*       Pre:  None
 *      Post:  Welcome displayed to the screen
 *   Purpose:  Display Greeting message to the user
 *********************************************************/
void sayHi()
{
   cout << "Welcome\n";
}
```

5.7 Parameters for *pass-by-reference* and *pointer* must be in the following format, where the & and * symbols must be with the data type and not the identifier

- Incorrect:
  i. Pass-by-reference: `void function(int &number);`
  ii. Pointer: `void function(int *ptr);`
- Correct:
  i. Pass-by-reference: `void function(int& number);`
  ii. Pointer: `void function(int* ptr);`

6. DATA TYPES

6.1 Struct

6.1.1 All fields must be indented.

[1] If the project is large enough, otherwise not needed.
[2] For class member functions, the functions should be list in ascending order within their own section.

6.1.2 It must be declared in the header file.

Example:

```
struct Account
{
   string accountNumber;
   double balance;
};
```

6.2 Class

6.2.1 The private, protected, and public members of a class must be declared in this order: Private members must be declared before the protected members, and the protected members must be declared before the public members. Each section of members must be indented.

6.2.2 All private members must start with a prefix "m" that indicates a member.

6.2.3 Each function must be separated by a blank line.

6.2.4 Member functions must be declared in the header file and member function definitions must be written in the implementation file.

6.2.5 Each class should have its own header and implementation file to allow for abstraction.

6.2.6 All member functions names should follow the naming convention.

Example:

```
class Example
{
   private:
      int mValue;

   protected:
      int mNum;

   public:
      Example();
};
```

7. REFERENCES

1. Corporation, Lockheed Martin. "Joint Strike Fighter: C++ Coding Standard." *Joint Strike Fighter: C++ Coding Standard*. http://www.research.att.com/~bs/JSF-AV-rules.pdf (asccessed Dec 21, 2008)
2. Department of Mathematics and Computer Science, Denison University. "Department of Mathematics and Computer Science." *Department of Mathematics and Computer Science*. http://personal.denison.edu/~havill/style.pdf (accessed December 21, 2008)
3. Services, Geothechnical Software. "C++ Programming Style Guidelines." *C++ Programming Style Guidelines*. http://geosoft.no/development/cppstyle.html (accessed May 21, 2014)
4. Stroustrup, Bjarne. "Stroustrup's C++ Style and Technique FAQ." Stroustrup's C++ Style and Technique FAQ. http:// http://www.stroustrup.com/bs_faq2.html (accessed May 21, 2014)