Missouri S&T Computer Science Department
C++ Coding Standard

August 21, 2016

Contents

1 Introduction

1.1 Why do we need a coding standard?

A coding standard is desirable for a couple of reasons:

- Most software companies enforce some kind of coding standard, so this is good experience preparing for a "real-world" environment.
- Using a coding standard makes code more readable, making it easier for the graders to assist students with coding problems and allowing them to evaluate the code and return grades more quickly.
- All code will be graded against the same standard, allowing for more uniform grading.

1.2 Acknowledgements

Some elements of this document are inspired by Code Conventions for the Java™ Programming Language, revised April 20, 1999, available at: `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`.

2 File Names

The purpose of the file should be easy to determine from its name. You may call your main implementation file `hw1.cpp` (Homework 1) or `main.cpp`, but other files (class header or implementation files) should be given more descriptive names. For instance, in the case of a class header, use the name of the class.

2.1 File extensions

Header files:	`.h`
Implementation files:	`.cpp`
Template implementation files:	`.hpp`

2.2 Examples of valid filenames

Filename	Contents
`main.cpp`	`int main()` function for the assignment
`hw1.cpp`	`int main()` function for assignment 1
`vector.cpp`	Member function implementations for a non-templated vector class
`vector.hpp`	Member function implementations for a templated vector class
`sl_list.h`	Header file for a singly-linked list class

3 Project Files

The files in your program will either be header or implementation files. Header files will contain all function prototypes and struct or class declarations. Implementation files contain the implementations of all functions (with the exception of set and get functions).

Filenames for headers and implementations must match except for the extension. If your function is prototyped in a file called `foobar.h`, it must be implemented in `foobar.cpp`.

At most, every line in your file should be less than 80 characters wide (including indentations). This is a standard display width for most terminals.

3.1 Header files

Header files must always contain lines similar to the following before any other lines of code:

```
#ifndef FILENAME_EXTENSION
#define FILENAME_EXTENSION
```

Adding these lines ensures that if a header file is `#include`d by more than one implementation file, the code it contains is only included and compiled once.

By standard C++ programming conventions, the `#define` value is "FILENAME underscore EXTENSION". For example, in the `vector.h` above, the appropriate lines are:

```
#ifndef VECTOR_H
#define VECTOR_H
```

The last line of every header file should be:

```
#endif
```

Alternatively, you may place the following at the top of a header file:

```
#pragma once
```

This eliminates the need for the `#ifndef` / `#define` as well as the `#endif` at the end of the header file. While `#pragma once` is not standard, it is supported by every commonly-used compiler.

3.2 Implementation files

`.cpp` files should NEVER be `#include`d. `.hpp` files should be `#include`d at the end of the header file that defines the associated template class immediately before the `#endif`.

4 Comments

4.1 Files

The very top of every file must have a comment block like the following:

```
/*
 * File: filename.cpp
 * Author: Your name and your class and section number
 * A brief description of the file's contents
 */
```

4.2 Classes

Before each class declaration (i.e. in the header file for each class), the following comment block must appear:

```
/*
 * Class: class_name
 * A brief description of what the class does
 */
```

4.3 Functions

Both regular and member functions must be documented at the point of declaration(i.e. in a header file). When defining a class, the comment blocks will appear inside the class definition immediately above the member function they describe.

```
/*
 * Function: function_name
 * A description of the function that includes any
 *   information needed to use it
 * Pre: The function's preconditions, in terms of
 *   program variables and system state
 * Post: The function's postconditions
 * Param paramname1: A description of the first parameter
 * Param paramname2: A description of the second parameter
 * Return: A description of the return value
 */
```

4.4 Class member variables

Each class member variable should have a short comment describing its purpose:

```
class list
{
  private:
  int m_size; // Stores the number of nodes in the list
};
```

5 Naming conventions

The purpose of any variable, function, class, or struct should be obvious from its name. You are not required to use Hungarian notation (`intSize`, `strName`), but you are required to use descriptive names.

You may choose to use underscores (`line_count`, also known as 'snake case') or mixed case (`lineCount`, also known as 'camel case') names.

You may only use single-letter variable names under the following conditions:

- The purpose is obvious from the name, e.g. `x` and `y` for coordinates, or `w` and `h` for width and height
- The variable is being used as a counter in a for or while loop, e.g. `for(int i = 0; i < 10; i++)`

Global variables should not appear in your programs. Global constants ought to be named with all uppercase letters and underscores; for instance, `float PI=3.14159`.

Names of variables and functions should begin with lower-case letters. Class names begin with upper-case letters.

Template types should be named descriptively. `T` is acceptable as it is a common convention. `generic` is also acceptable since it is descriptive.

6 Formatting Statements

6.1 Indenting

Every line inside a code block should be indented one level for easier readability.

Do not use tabs to indent; use spaces instead. You may indent two or four spaces for each level, but you must be consistent across all the files in your program. Most editors can be configured to automaically indent your code as you type.

6.2 Simple statements

Each line should contain no more than one statement. For instance, this is not acceptable:

```
argv++; argc--; // <-- NO!
```

Don't use the comma operator to group unrelated statements.

```
cerr << "error", exit(1); // THIS IS ALSO BAD
```

6.3 Blocks

Blocks consist of a list of statements enclosed in curly braces (`{}`).

- The opening and closing braces should appear on their own lines at the indent level of code outside the block.
- The statements inside the block should be indented one more level.
- Braces should be included around all blocks, including blocks containing only one statement. This prevents bugs caused by forgetting to add braces around a block when adding additional statements.

The following sections contain many examples of this formatting.

6.4 `return` statements

Every function, including `void` functions, must end with a `return` statement.

Multiple return statements are allowed when sanity-checking arguments. Otherwise, you are strongly discouraged from using multiple return statements as it makes following the flow of the function difficult. For example,

```
int divide(int num, int denom)
{
  if(denom == 0)
  {
    cerr << "Division by zero!";
    return 0;
  }

  return num/denom;
}
```

Return statements should use parentheses only when needed to make the return value clearer.

```
return;
return myDisk.size();
return (size ? size : defaultSize);
```

6.5 `if`, `if-else`, and `if-else-if-else` statements

These statements should have the following forms:

```
if(condition)
{
  statements;
}
```

```
if(condition)
{
  statements;
}
else
{
  statements;
}
```

```
if(condition)
{
  statements;
}
else if(condition)
{
  statements;
}
else
{
  statements;
}
```

Braces must be included even if there is only one statement in the block. Do not do this:

```
if(condition)
  statement;
```

6.6 `for` statements

A `for` statement should have the following form:

```
for(initialization; condition; update)
{
  statements;
}
```

If a for loop is empty; that is, all the work in the for loop is done by the initialization, condition, and update clauses, you may format it as follows:

```
for(initialization; condition; update);
```

When using the comma operator in the initialization or update clause of a for statement, avoid the complexity of using more than three variables. If needed, use separate statements before the for loop (for the initialization clause) or at the end of the loop (for the update clause).

6.7 `while` statements

A `while` statement should have the following form:

```
while(condition)
{
  statements;
}
```

An empty `while` statement may be formatted as follows:

```
while(condition);
```

6.8 `do-while` statements

`do-while` statements should have the following form:

```
do
{
  statements;
} while(condition);
```

This is the one exception to the rule that braces appear on their own lines.

6.9 `switch` statements

A `switch` statement should have the following form:

```
switch(variable)
{
  case ABC:
    statements;
    /* falls through */
  case DEF:
    statements;
    break;
  case XYZ:
    statements;
```

```
    break;
  default:
    statements;
    break;
}
```

Every case that falls through (doesn't `break`) must include a comment where the `break` statement would usually appear stating as much. This makes it easy to spot bugs where the `break` statement was accidentally omitted.

Every `switch` should contain a `default` case. In the example above, the `break` statement is redundant, but it prevents a fall-through error if more cases are added at a later time.

6.10 Class declarations

Classes must be declared in the following format:

```
class ClassName
{
  public:
    // public member function declarations

  private:
    // private member variables and functions
};
```

Example

Listing 1: temperature.h

```
#ifndef TEMPERATURE_H
#define TEMPERATURE_H

/*
 * File: temperature.h
 * Author: Nathan Jarus, CS 1570 A
 * This file contains a class for converting
 * temperature units.
 */

/*
 * Class: Temperature
 * This class stores the value of one temperature.
 * It can convert temperatures between degrees Fahrenheit,
 * Celsius, and Kelvin.
```

```
 */

class Temperature
{
  public:
    /*
     * Function: setCelsius
     * Sets the temperature using degrees Celsius
     * Pre: the parameter must be in degrees Celsius
     * Post: the parameter is converted to degrees Kelvin
     *   and stored in m_temperature
     * Param temp: the temperature to store, in degrees
     *   Celsius
     */
    void setCelsius(double temp);

    /*
     * Function: setFahrenheit
     * Sets the temperature using degrees Fahrenheit
     * Pre: the parameter must be in degrees Fahrenheit
     * Post: the parameter is converted to degrees Kelvin
     *   and stored in m_temperature
     * Param temp: the temperature to store, in degrees
     *   Fahrenheit
     */
    void setFahrenheit(double temp);

    /*
     * Function: setKelvin
     * Sets the temperature using degrees Kelvin
     * Pre: the parameter must be in degrees Kelvin
     * Post: the parameter is converted to degrees Kelvin
     *   and stored in m_temperature
     * Param temp: the temperature to store, in degrees
     *   Kelvin
     */
    void setKelvin(double temp);

    /*
     * Function: getCelsius
     * Retrieves the temperature in degrees Celsius
     * Pre: none
     * Post: The value of m_temperature is converted to
     *   degrees Celsius
     * Return: The value of m_temperature in degrees
     *   Celsius
```

```
     */
    double getCelsius();

    /*
     * Function: getFahrenheit
     * Retrieves the temperature in degrees Fahrenheit
     * Pre: none
     * Post: The value of m_temperature is converted to
     *   degrees Fahrenheit
     * Return: The value of m_temperature in degrees
     *   Fahrenheit
     */
    double getFahrenheit();

    /*
     * Function: getKelvin
     * Retrieves the temperature in degrees Kelvin
     * Pre: none
     * Post: The value of m_temperature is converted to
     *   degrees Kelvin
     * Return: The value of m_temperature in degrees
     *   Kelvin
     */
    double getKelvin();

  private:
    double m_temperature; // Stores a temperature in
                          // degrees Kelvin
};

#endif
```

Listing 2: main.cpp

```
/*
 * File: main.cpp
 * Author: Nathan Jarus, CS 1570 A
 * Demonstrates the use of the Temperature class
 */

#include "temperature.h"

using namespace std;

int main()
{
```

```
  Temperature t;

  t.setCelsius(52.3);
  cout << t.getKelvin() << endl;

  return 0;
}
```