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

September 23, 2007

Missouri S&T CS Department C++ Coding Standard

Table of Contents

Missouri S&T CS Department C++ Coding Standard

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 (in the case of a class header, use the name of the class).

2.1 Filename extensions

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

2.2 Examples of valid filenames:

Filename:	**Expected contents:**
`main.cpp`	`int main()` function for the assignment
`hw1.cpp`	`int main()` function for assignment 1
`vector.cpp`	Function implementations for a non-templated vector class
`vector.tpp`	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 header 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
```

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

```
#ifndef VECTOR_HPP
#define VECTOR_HPP
```

The last line of every header file should be:

```
#endif
```

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.

3.2 Implementation Files

`.cpp` files should NEVER BE `#included`. `.tpp` files should be `#included` at the end of the header file in which the templated class is defined.

4 Comments

4.1 Files

The very top of every file must contain the following comment block.

```
////////////////////////////////////////////////////////////////////
/// @file header.h
/// @author Your name and class section here
/// @brief A brief description of the file
////////////////////////////////////////////////////////////////////
```

4.2 Classes

In header files a class comment block should follow directly after a file comment block. Class comment blocks are not included in source files.

```
////////////////////////////////////////////////////////////////////
/// @class className
/// @brief A brief description of the class
////////////////////////////////////////////////////////////////////
```

4.3 Functions

In header files the function comment blocks should follow directly after the class comment block. In source files function comment blocks must directly precede the function description.

```
////////////////////////////////////////////////////////////////////
/// @fn functionName
/// @brief A brief description of the file
/// @pre The function’s precondition in terms of the program variables
/// and process statuses.
/// @post The function’s postcondition in terms of program variables
/// and process statuses
/// @param paramName1 description of the parameter
/// @param paramName2 description of the parameter
/// @return description of the return
////////////////////////////////////////////////////////////////////
```

4.4 Class Member Variables

In header files the class member variables should be followed by a brief description of their purpose.

```
class list
{
    ...
    private:
    int m_size; ///< holds the size of the list
};
```

4.5 Loops (CS253 only)

In source files loops should be preceded by the following comment blocks structure.

```
////////////////////////////////////////////////////////////////////
/// loop description
/// loop precondition: loop's precondition
/// loop postcondition: loop's postcondition
/// invariant: loop's invariant
/// proof: outline of initialization, maintenance, and termination
////////////////////////////////////////////////////////////////////
```

5 Program code

5.1 Indenting

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

Do not use tabs to indent—use spaces instead. You may indent with two, three, or four spaces for each level of indentation, but **you must be consistent** throughout all files of your program.

5.2 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 (`test_vect`) or mixed case (`testVect`) names.

One-letter names should only be used if:

- The purpose is obvious from the name (for coordinates, `X` and `Y`; for sizes, `W` = width and `H` = height)
- A variable is being used as a counter for a for or while loop
  `for (int i = 0; i < 10; i++)`

Global variables should not appear in your program, but they would be named with all uppercase letters (`float PI = 3.1415926`). Names of all other variables and functions always begin with a lowercase letter.

By convention, class names should always begin with an uppercase letter.

Template types should be named descriptively. `T` is acceptable as it is a conventional name. `generic` is acceptable because it is descriptive.

6 Statements

6.1 Simple statements

Each line should contain at most one statement. Example:

```
argv++; argc--; // DON'T DO THIS!
```

Do not use the comma operator to group multiple statements unless it is for an obvious reason. Example:

```
if (error)
{
    cerr << "error", exit(1); // DON'T DO THIS EITHER!
}
```

6.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`". See the following sections for examples.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even singletons, when they are part of a control structure, such as an `if-else` or a `for` statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

6.3 return Statements

A `return` statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;

return myDisk.size();

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

6.4 if, if-else, if-else-if-else Statements

The `if-else` class of statements should have the following form:

```
if (condition)
{
    statements;
}

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

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

It is strongly encouraged that `if` statements always use braces {}. Avoid the following form:

```
if (condition) // AVOID! THIS OMITS THE BRACES {}!
    statement;
```

6.5 for Statements

A `for` statement should have the following form:

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

An empty `for` statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
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).

6.6 while Statements

A `while` statement should have the following form:

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

An empty `while` statement should have the following form:

```
while (condition);
```

6.7 do-while Statements

A `do-while` statement should have the following form:

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

6.8 switch Statements

A `switch` statement should have the following form:

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

Every time a case falls through (doesn't include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if later another `case` is added.

Example

temperature.hpp

```
#ifndef TEMPERATURE_HPP
#define TEMPERATURE_HPP

////////////////////////////////////////////////////////////////////////
/// @file temperature.hpp
/// @author Matt Buechler CS153A
/// @brief This header file declares the Temperature class.
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
/// @class Temperature
/// @brief This class stores the value of one temperature.  Programmers
/// can use this class to convert between degrees Celsius, Fahrenheit,
/// or Kelvin.
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
/// @fn Temperature::setCelsius( double )
/// @brief This function sets the temperature using degrees Celsius
/// @pre The parameter must be in degrees Celsius
/// @post The Celsius temperature is converted into degrees Kelvin and
/// stored in m_temperature
/// @param x holds a temperature value in degrees Celsius
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
/// @fn Temperature::setFahrenheit( double )
/// @brief This function sets the temperature using degrees Fahrenheit
/// @pre The parameter must be in degrees Fahrenheit
/// @post The  temperature is converted into degrees Kelvin and
/// stored in m_temperature
/// @param x holds a temperature value in degrees Fahrenheit
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
/// @fn Temperature::setKelvin( double )
/// @brief This function sets the temperature using degrees Kelvin
/// @pre The parameter must be in degrees Kelvin
/// @post The Kelvin temperature is stored in m_temperature
/// @param x holds a temperature value in degrees Kelvin
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
/// @fn double Temperature::getCelsius()
/// @brief This function returns the value of m_temperature in degrees
/// Celsius
/// @pre none
/// @post A copy of the value of m_temperature is converted into
/// degrees Celsius and is returned
/// @return The value of m_temperature in degrees Celsius
////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////////////////////////
/// @fn double Temperature::getFahrenheit()
/// @brief This function returns the value of m_temperature in degrees
/// Fahrenheit
/// @pre none
/// @post A copy of the value of m_temperature is converted into
/// degrees Fahrenheit and is returned
/// @return The value of m_temperature in degrees Fahrenheit
////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////
/// @fn double Temperature::getKelvin()
/// @brief This function returns the value of m_temperature
/// @pre none
/// @post A copy of the value of m_temperature is returned
/// @return The value of m_temperature in degrees Kelvin
////////////////////////////////////////////////////////////////////

class Temperature
{
    public:
        void setCelsius( double x );
        void setFahrenheit( double x );
        void setKelvin( double x );
        double getCelsius();
        double getFahrenheit();
        double getKelvin();
    private:
        m_temperature; ///< holds the temperature value in Kelvins
};

#endif
```

main.cpp

```
////////////////////////////////////////////////////////////////////
/// \file temperature.hpp
/// \author Matt Buechler
/// \brief This header file declares the Temperature class.
////////////////////////////////////////////////////////////////////

#include “temperature.hpp”

int main()
{
    Temperature t;

    t.setCelsius( 53.23 );
    cout << t.getKelvin(); // outputs 326.23

    return 0;
}
```