C++ Style Rules

CSULB, CECS 282
Neal Terrell

The style rules below are mandatory for all submitted labs and assignments. They are not optional, and violating a rule will result in a rejection of your submission.

1. Spacing

1. No line may exceed 80 characters in length. If you must break a line in two, indent the second line by one additional space beyond the first line's indentation. If a third line is needed, keep it at the same indentation as the second line. *Example*:
```
This is a long line...
 it continues on the next line
 indented by an additional space.
```
2. Use only one statement per line. You may not compact multiple statements onto one line.
3. EXCEPTION: a method declaration and body may be put onto one line if it consists of a single statement and does not violate the 80 character length restriction. *Example*:
`int GetTotal() {return QUARTER_VALUE * mQuarters;}`
4. You may compact multiple variable declarations onto one line when appropriate. *Example*:
`double speed, time, distance;`
5. Put spaces after a comma or the semicolons in a for-loop. Never put a space *before* a comma or semicolon. Put spaces around each keyword ("if", "while", etc.). Put spaces around operators, except for `++` and `--`.
6. Do not put spaces between a function name and the parenthesis which follows. Do not put spaces between parentheses and the code which follows.
Bad: `CallFunction ( "Hello" );`
Good: `CallFunction("Hello");`
7. Do place a space between a control statement (`if`, `else`, `while`, and `for`) and the parenthesis which follows.
8. There must be one space between a { and the code preceding it.
9. Use single empty lines to break up your code so it is more readable. In general, your code should fall into groups of 5-8 lines, separated by a blank line or a brace. Put a blank line after your local variable definitions in a method.

2. Identifiers

1. Give meaningful names to variables. Another programmer should be able to read your variable name and have an idea of what it is used for. Names like X, blah, and no are not as meaningful as daysPerYear, firstItem, and location.
2. Avoid one-letter variable names, except as counters in for-loops, or when they are domain appropriate (like the X coordinate of a point on a graph).
3. Avoid abbreviations in names unless the abbreviation is commonly known or obvious, and you use it everywhere in your code.
4. Use these rules for casing and multi-word identifiers:

(a) All variable names start with a lowercase letter. Capitalize the first letter of the second, third, and later words in the identifier name. Do not separate words with underscores. This is called "camel casing." *Example*: daysPerYear

(b) All class and method names follow the rules for variable names, except the first letter of the first word is also capitalized. This is called "Pascal casing." *Example*: TextFile, ThisIsAMethod

(c) Class nonpublic (private/protected) member data must start with the letter "m" followed by the variable name using the rules for variable names. *Example*: mFirstName

(d) Const variables must be named using all capital letters, with underscores ("_") separating words. *Example*: SECONDS_PER_MINUTE

5. Local variables in methods should be declared when they are needed, and groups of related variables should be declared in sequential lines. For example, if you are reading three sides of a triangle from the user, you should declare all three variables together, as in:

```
int side1, side2, side3;
// read input from user
```

or

```
int side1;
int side2;
int side3;
// read input from user
```

and **not** as

```
int side1;
// read side1 from user
int side2;
// read side2 from user
```

3. Indentation

1. Use spaces for indentation. Each indentation level should be 3 spaces.
2. Indent one level for each {, but **do not** indent access modifier labels (public, private, protected):

```
class Hello {
   int x;
public:
   Hello(int);
   char* GetName() {return "Neal";}
};
```

excepting the provision in rule 1.3.

3. Place opening braces ("{") on the same line as the control statement it belongs to. Do not place a { on its own line. If a control statement runs more than one line, place the brace at the end of the final line of the statement.
4. Place closing braces ("}") on their own line, in the same column as the beginning of the statement containing the opening {, again excepting the provision in rule 1.3.

4. Constant values

1. Constant values (that do not change between program runs) must be expressed using const variables; **not** with #defines.
2. Your code may not use "magic numbers" except when declaring const variables. Only the numbers 0, 1, and 2 may appear in your code outside of final variables; even these might need to be in const variables if they might change in later versions of your code.

Example:
Bad:
```
double money = quarters * 0.25;
```
Good:
```
const double QUARTER_VALUE = 0.25;
double money = quarters * QUARTER_VALUE;
```

3. Define parameters, variables, return values, and member methods to be const whenever possible. Use const references when passing complex types as parameters by value.

4. Do not use references as parameters if you intend to modify the actual parameter in the method. Use pointers instead.
Example:
Bad:
```
void ModifyDouble(double &x) {x = 10;}
```
Good:
```
void ModifyDouble(double *x) {*x = 10;}
```

5. Class design

1. When a type is just a simple set of data, use a struct. If you need a type with methods other than a constructor/destructor, use a class.

2. All member data in classes must be declared private, with the exception of public static constants. Member data in structs may be declared public.

3. If a private member variable needs to be accessible outside your class, declare appropriate public accessor and mutator methods ("getters and setters"):
```
class Wallet {
   int mQuarters;
public:
   void SetQuarters(int quarters) {mQuarters = quarters;}
   int GetQuarters() {return mQuarters;}
};
```

4. Organize header files (*.h) for classes in the following order:

   (a) A public block with enums, static const variables, constructors, destructor, and public methods.

   (b) A private block with private member variables, methods, and friend declarations.
```
class Wallet {
public:
   enum Currency {DOLLARS, YEN};
   static const double QUARTER_VALUE;

   Wallet();
   Wallet(double amount);
   virtual ~Wallet();

   void AddMoney(double);
   void GetAmount();

private:
   double mAmount;
   double ConvertToYen(double);
   friend class DebitCard;
};
```

6. Large Scale Organization

1. All .h files should be protected with #ifndef/#define pairs to prevent multiple-inclusion.
2. All classes should come in .h/.cpp pairs, with the .h file containing the class declaration and the .cpp containing definitions for that class.
3. Do not put a *using namespace* statement in a .h file.
4. Do not #include a non-system .h file in another .h file when a forward-declaration would suffice.

7. Elegance

1. Do not use the statement "`== true`" or "`== false`" in code. Use boolean variables as direct conditions.
   Bad: `if (mFinished == true)`
   Good: `if (mFinished)`
2. Do not assign simple values to member variables in a constructor bodies. Use an initializer list instead.
3. Do not make accessor and mutator methods for private member variables unless there is a legitimate reason for non-class code to need to access those members.
4. Replace if-else statements with boolean expressions when appropriate. Example:

```
if (value % 2 == 0)
   isEven = true;
else
   isEven = false;
```

   should instead be

```
isEven = value % 2 == 0;
```