C++ Programming Style Guide

Computer Science Program
Cedarville University

Goal: Our goal is to produce well-written code that can be easily understood and will facilitate life-cycle maintenance. These guidelines lay out principles that C++ programmers have found useful for producing code that contains fewer bugs, is easier to debug and maintain, and that others can understand well. It is important to remember these are just guidelines, and efficiency and understandability should not be sacrificed to blindly follow them. Remember: there are no style points for "slick code."

Layout and Comments:

1. Code should be split into separate files in a logical manner. In general, only one class should be included per file, with the class specification in a header file (.h suffix) and the class implementation in a code file (.cpp suffix).

2. Each file will begin with a file header block. This block should contain, as a minimum, an overview of the code in the file, the author, date created, date and summary of modifications (as applicable).

```
// Author:    Joe Smith
// Purpose:  This file includes the implementation for the Rational class (defined
// in Rational.h).
// Date:       23 Jun 2000
// Summary of Modifications:
//         14 Jul 2000 – JS     – corrected bug in multiply method
//         24 Jul 2000 – KAS – added Rational::reduce method
```

3. Each user-defined class specification (in the header file) will begin with a class header block. This block should contain, as a minimum, the name of the class, a description of the class, and an explanation of the class hierarchy (who derived from, who derives from this class). The block should also explain anything about the class which may be out of the ordinary; e.g., why it uses private inheritance or why a data member had to be made public.

```
// Class Name: Rational
//
// Description:  This class provides an ADT for fractions.  Users of this class are
// provided methods to manipulate fractions in an intuitive manner.  For example,
// arithmetic (+, -, *, /), comparison (<=, ==), and I/O (<<, >>) operators work for
// Rationals.  Rational does not inherit from other classes, and is not anticipated to
// be a base class for other classes.
```

4. All header files must include **#ifndef** statements to prevent multiple inclusion. Statements within the header file will be ordered as follows:
   a. Multiple inclusion protection
   b. Prototypes for functions whose use is outside the file they are implemented in
   c. Any required #include statements
   d. Class specifications
   e. Declaration of global variables (use of globals should be avoided)

5. Statements within code files (.cpp) should be ordered as follows:
   a. Any required #include statements
   b. Declaration of static variables (whose scope is within this file only)
   c. Other required functions
   d. Declaration of static function prototypes
   e. Class implementations

6. Functions within code files will be grouped logically, and ordered in a top-down fashion; i.e., top-level functions such as *main*() or *APImain*() will come first, followed by functions called by the top-level functions.

7. An appropriate amount of comments should be used throughout your code. Although it is most common for programmers to provide too few comments, over-commenting can also negatively impact the readability of the code. It is important to comment on any sections of code whose function is not obvious (to another person). For algorithmic-type code which follows a sequence of steps, it may be appropriate to summarize the algorithm at the beginning of the section (perhaps in the function header) and then highlight each of the major steps throughout the code. Either the C or C++ style comments may be used (C++ style is preferred). However, the same style should be used throughout your program. Comments should be indented at the same level as and precede the code they describe.

8. Functions should always have a specified return type. For functions not returning a value, use *void* type.

<u>Naming Conventions</u>:

1. Use descriptive names for variables and functions in your code, such as xPosition, distanceToGo, or findLargest(). The only exception to this may be loop iteration variables, where no descriptive name makes sense. In this case, use the lower case letters beginning with *i* (i, j, k, etc).

2. For variable and function/method names, the first word should be lowercase, and subsequent words should be capitalized. For example, upperLimit, averageValue, makeConnection(), addBody(). Whenever possible, function/method names should be verbs: draw(), getX(), setPosition().

3. For constants, capitalize all letters in the name, and separate words in the name using an underscore, e.g., PI, MAX_ARRAY_SIZE. Any numeric constants needed in your code (other than very simple ones like –1, 0, and 1) should be replaced by a named constant.

4. For boolean variables and functions, the name should reflect the boolean type, e.g., isEmpty (), isLastElement, hasChanged. Names should always be positive; i.e., use hasChanged rather than hasNotChanged.

Statements:

1. Only one statement is allowed per line, and each line of code will be no more than 80 characters in length (to prevent line-wrap). If a statement requires more than one line, subsequent lines will use a hanging indent to make it obvious that the statement extends over multiple lines, as shown below.

```
int myFunction (int variableA, int variableB, int variableC, int variableD,
                int variableE, int variableF );
```

2. A single declaration per line is preferred. Never mix multiple types on the same line. Avoid initialized and un-initialized variables being declared on the same line.

```
int id, grades[10];          // bad
int height, weight = 5;      // bad
```

3. Braces can be done in one of two styles: the opening brace can be put on the end of the line defining the block, or it can be on a separate line by itself, as shown below. Code inside the braces will always be indented. Always use braces with selection (i.e., if or switch statements) and looping, even when they are not strictly required by the compiler.

```
for (i = 0; i <= maxSize; i++)   {
  …
  …
}
```

```
for (i = 0; i <= maxSize; i++)
{
   …
}
```

4. Make wide use of horizontal white space: e,g, include spaces after commas, and between operands and operators in expressions. Avoid using tabs to create white space.

5. Use vertical white space to separate your code into “paragraphs” of logically connected statements.

6. Use parentheses liberally for clarity.

7. Avoid implicit tests for zero, except for boolean variables
   OK: if (isEmpty) … if (counter == 0) …
   Avoid: if(!counter)

8. Avoid "slick code." For example, *for* statements should typically only include initialization, test, and increment of iteration variable, not additional statements which belong in the loop body. Avoid: for (i = 0, sum=0; i <= maxCount; i++, sum+=i*4) {…}.

9. Prefer *for* loops for counting loops and *while* loops for loops having an indefinite number of iterations.

10. For *if* statements, the nominal or more frequent case should be placed in the "then" block, and the exceptional or less frequent case in the *else* block.

11. The *goto* statement should be used only if no other approach will produce straightforward code. Only in extremely rare, if any, cases should it become necessary. The *break* and *continue* statements should also be used sparingly. Although their need is much more common than for the *goto*, they should not be used if another straightforward approach can be used.

11. Large objects being passed as function parameters should be passed by reference. If the called function does not need to modify the object, it should be declared as a *const* reference parameter. For variables passed by value, the variable should not be declared as *const*.

Classes:

1. Member data should always be *private*, and *public* or *protected* mutators and inspectors [e.g., getX(), setX()] provided.

2. Avoid use of implementation code within the class specification in the header file. Again, an exception would be in critical sections of the code.

3. If an enumeration is required, it may be declared in the *public* section of the class rather than as a global variable outside the class.

4. Derived classes should use *public* inheritance if at all possible. Use of *protected* or *private* inheritance should be explained in source comments.

5. Use of the *const* modifier should be maximized. Any member function which does not modify the object should be marked as *const*. Any reference parameter not modified should also be marked as *const*.

6. Within the class specification, member functions should be provided in the following order:

    a. Default constructor
    b. Other constructors
    c. Copy constructor
    d. Destructor
    e. Assignment operator
    f. Other operators
    g. Mutators/inspectors
    h. Other member functions

7. Explicit declaration of the destructor, copy constructor, and assignment operators is optional, if no references appear among the class attributes. If references appear in the class attributes and they are not provided, a comment should be made indicating that the defaults are purposefully being used.