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 readable code that contains fewer bugs and is easier to maintain. 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 (.c or .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, summary of modifications, and copyright statement.

```
/**********************************************************
* Summary:  This file includes the implementation for the Rational class (defined
* in Rational.h).
*
* Author: Joe Smith
* Created: Jun 2000
* Summary of Modifications [if applicable]:
*         14 Jul 2000 – JMS – corrected bug in Multiply routine
*         24 Jul 2000 – JMS – added Rational::reduce method
*
*  ©Copyright Cedarville University, its Computer Science faculty, and the
*  authors.  All rights reserved.
**********************************************************/
```

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.

```
/*********************************************************
* 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.
*
* ©Copyright Cedarville University, its Computer Science faculty, and the
*  authors.  All rights reserved.
* *********************************************************/
```

4. Functions within code files should typically be 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.

5. 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 (e.g., in a 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 recommended). However, the same style should be used throughout your program. Comments should be indented at the same level as the code they describe, and should appear before the code to which they refer.

<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 (sometimes referred to as "camelcase"). 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 typically 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 be indented 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, or initialized and un-initialized variables.

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

3. Avoid declaring global variables.

4. 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.

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

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

5. Make consistent use of horizontal white space. For example, include spaces after commas, and between operands and operators in expressions. Avoid using tabs to create white space. Note the spacing in the example in the for-loop in paragraph 3 above.

6. Use vertical white space to separate your code into "paragraphs" of logically connected statements.

7. Use parentheses as needed for clarity.

8. The conditional expression in an *if*, *for*, or *while* statement must be of type boolean. You shouldn't compare a boolean variable explicitly against true/false.
   OK: if (isEmpty) … if (counter == 0) …
   Avoid: if(!counter) … if (ifEmpty == true)

9. Write straightforward code and avoid using unnecessarily complicated 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) {…}

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

11. Looping: C++ provides two primary statements for iteration: the *while* statement and the *for* statement. The *while* statement should be preferred when the number of loop iterations is unknown at the outset of the loop or when early termination from the loop is a possibility. In addition, *while* loops are preferred when the exit condition is a non-counting condition. The *for* statement should be preferred when the number of loop iterations is known, i.e., they usually can be counted. Early termination of *for* loop is discouraged. Using *do* … *while* loops is discouraged.

12. The *break* statement should be used sparingly, except in the context of a *switch* statement.

13. The *continue* statement should be used sparingly.

14. Structured programming encourages a single exit point (i.e., a single return statement) from a function. Multiple return statements from a function should be used sparingly.

15. The *goto* statement is not to be used.

16. Large objects should be passed by reference. If the function does not 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.

17. Enumerations should be used when you need to specify a set of possible integer values. Unless the integer value is meaningful, you should not override the default enumeration numbering.

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. A possible exception would be in code where efficiency is paramount.

3. An enumeration associated with a class should not be declared globally (outside the class) unless it needs to be visible outside the class.

4. Derived classes should use public inheritance. Use of protected or private inheritance should be well documented and explained.

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
   c. Destructor
   d. Assignment operator
   e. Mutators/inspectors
   f. Other public member functions
   g. Private member functions

7. Member functions which are "helpers" for other member functions and are not part of the "interface" of the class should be declared private.

8. Explicit declaration of the destructor, copy constructor, and assignment operators is optional in situations where a shallow copy is sufficient. However, if not provided, a comment should be made indicating that the defaults are purposefully being used.

9. Make good use of comments within the class specification. Documenting the purpose and use of public methods and private variables can make your class much easier to understand.