C++ Programming Style Guide

Background:

C++ is one of the main development languages used by many open-source projects. As every C++ programmer knows, the language has many powerful features, but this power brings with it complexity, which in turn can make code more bug-prone and harder to read and maintain.

The goal of this guide is to manage this complexity by describing in detail the dos and don'ts of writing C++ code. These rules exist to keep the code base manageable while still allowing coders to use C++ language features productively.

Style, also known as readability, is what we call the conventions that govern our C++ code. The term "style" is a bit of a misnomer, since these conventions cover far more than just source file formatting.

Most open-source projects developed by Google, Intel, and other industry giants conform to the requirements in this guide.

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.

All student code submitted for grading should adhere to the style specified in this guide to earn full points.

Goals

Why do we have this document?

There are a few core goals that we believe this guide should serve. These are the fundamental **why**s that underlie all the individual rules. By bringing these ideas to the fore, we hope to ground discussions and make it clearer to our programming students why the rules are in place and why particular decisions have been made. It is important that you understand the rules and the rationale for them.

The goals of the style guide as we currently see them are as follows:

Style rules should pull their weight

The benefit of a style rule must be large enough to justify asking all students to remember it. The benefit is measured relative to the codebase we would get without the rule, so a rule against a very harmful practice may still have a small benefit if people are unlikely to do it anyway.

Optimize for the reader, not the writer

You should assume that your code is expected to continue for quite some time. As a result, more time will be spent reading most of your code than writing it. We explicitly choose to optimize for the experience of our average software engineer reading, maintaining, and debugging code in our codebase rather than ease when writing said code. "Leave a trace for the reader" is a particularly common sub-point of this principle: When something surprising or unusual is happening in a snippet of code, leaving textual hints for the reader at the point of use is valuable.

Be consistent with existing code

Using one style consistently for all students lets us focus on other (more important) issues. Consistency also allows for automation: tools that format your code or adjust your #includes only work properly when your code is consistent with the expectations of the tooling. In many cases, rules that are attributed to "Be Consistent" boil

down to "Just use the Style Guide and stop worrying about it"; the potential value of allowing flexibility on these points is outweighed by the cost of having people argue over them.

Be consistent with the broader C++ community when appropriate

Consistency with the way other organizations use C++ has value for the same reasons as consistency within your student code. If a feature in the C++ standard solves a problem, or if some idiom is widely known and accepted, that's an argument for using it.

Avoid surprising or dangerous constructs

C++ has features that are more surprising or dangerous than one might think at first glance. Some style guide restrictions are in place to prevent falling into these pitfalls.

Avoid constructs that our average C++ programmer would find tricky or hard to maintain

C++ has features that may not be generally appropriate because of the complexity they introduce to the code. In widely used code, it may be more acceptable to use trickier language constructs, because any benefits of more complex implementation are multiplied widely by usage, and the cost in understanding the complexity does not need to be paid again when working with new portions of the codebase.

Be mindful of our scale

With a codebase of hundreds of students writing thousands of lines of code, some mistakes and simplifications for one programmer can become costly for many (when working on group projects, especially).

The intent of this document is to provide maximal guidance with reasonable restriction. As always, common sense and good taste should prevail. By this we specifically refer to the established conventions of the MHCC student community, not just your personal preferences or those of your team. Be skeptical about and reluctant to use clever or unusual constructs: the absence of a prohibition is not the same as a license to proceed.

----------CS160 & CS161 & CS260----------

C++ Version

Code should conform to C++14 and not use features from C++17 or C++2x.

File Names

All courses: All files should be saved with the extension, .cpp.

CS162 & CS260:

.h - extension for header files

.cpp - extension for implementation files

.cpp - extension for driver (test) files

.tcc - extension for template files

<u>Layout and Comments</u>:

Order of Code Components

All courses: All driver programs must contain needed components in this order:

1. Opening comments (described in more detail below)
2. Include directives
3. Namespace block opening
4. Global variables (very limited - see below) & constants
5. Function prototypes, with comments
6. Main function
7. Other functions

8. Namespace block closing

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

```
/ ************************************************************
*  wk3_operators_jsmith.cpp
*
*  Summary:  Program to demonstrate use of various mathematical operators
*  Specifications: Program will prompt the user to enter 6 separate integer
*                      values. Mathematic operators + - * / % will be used
*                      to generate output of a single mathematical expression.
*  Test cases:   0, 0, 0, 0, 0, 0              error message (divide by zero)
*                10, 5, 2, 6, 3, 7             5
*                 0, -12, 34, -2, 11, 3        2
*
*  Author: Joe Smith
*  Created: Jun 2000
*  Summary of Modifications:
*           14 Jul 2000 – JMS – corrected bug in Multiply routine
*           24 Jul 2000 – JMS – added Rational::reduce method
*
*  ************************************************************/
```

Namespaces
All programs should be written in a specified namespace. The namespace label should include the name of the assignment and the name of the user. For example:

```
namespace wk2_formOutput_pwiese {
  //....
}
```

2. Appropriate 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 reading your code. 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. Any C++ style comments may be used. However, the same style should be used throughout your program. Comments should be indented at the same level as the code they describe.

<u>Naming Conventions</u>:

1. Use descriptive names for variables and functions in your code, such as x_Position, distance_to_go, 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). Do not use cute names.

2. Variable names should be lowercase, with subsequent words separated by an underscore. Ex: wages, hourly_rate, peas_in_a_pod.

3. No variables and functions should have the same names in any parts of the program (including in multiple files).

4. 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.

Functions - All function labels (tags) must begin with a upper case letter, use camel case, and have a name that has meaning in the context of the purpose of the function.

5. Example: float ComputeEarnings()

6. For Boolean variables and functions, the name should reflect the Boolean type, e.g., IsEmpty (), is_last_element, has_changed. Names should always be positive; i.e., use has_changed rather than has_not_changed.

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 uninitialized variables.

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

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.

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

4. Make wide use of horizontal white space. Always include spaces after commas, and between operands and operators in expressions. Do not using tabs to create white space.

```
x=y*2;          // bad
x = y * 2;      // good
```

5. Use vertical white space to separate your code into "paragraphs" of logically connected statements. Never have more than one blank line of white space.

6. Use parentheses liberally for clarity.

7. Do not use implicit tests for zero, except for Boolean variables

   OK: if (is_empty) …
   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. For *if* statements, the nominal or most frequent case should be placed in the *if* block. Place remaining conditions in order from top to bottom as expected in frequency. The least frequently expected condition should be last.

10. The *goto* statement must not be used.

11. The *break* and *continue* statements should be used sparingly. They should be used only if another straightforward approach can be used (rare). (Switch structures are an exception - *break* will be used.)

12. 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 does not need to be declared as **const**.

13. Lines of code to be executed in the block of a conditional or repetition statement should *not* appear on the same line as the condition.

```
  if (value > 1000) { cout << "big"; }        // bad
if (value > 1000) {                           // good
        cout << "big";
}
if (value > 1000)                   // good
{
        cout << "big";
}
if (value > 1000)                   // good
        cout << "big";
```

Global Variables

Global variables are not allowed. (Global constants are fine.)

Exception Handling and Assertions

Exception handling should be used during development to help identify potential causes of failure of a program. It should be removed prior to submission of code, unless specifically required for an assignment.

Functions:

5. Functions within code files will be grouped logically, and ordered in a top-down fashion; i.e., *main*() will come first, followed by functions called by the higher-level functions. Therefore, prototypes for all user-defined functions are required prior to the *main*() function.

6. Never use the convention of a trailing return type.

7. Functions must always have a return statement. For functions not returning a value, use *void* type. All functions must end with an explicit return statement.

8. Function names should begin with an upper case letter, with subsequent words appearing in camel case. Ex: MakeConnection(), AddBodyToShape(). Whenever possible, function names should be verbs: Draw(), GetX(), SetPosition().
9. Use pass by value (with a return value), unless necessary to use pass by reference. Pass by value is always preferred.
10. If a function has a combination of pass by value and pass by reference parameters, pass by value parameters should be listed first, followed by pass by reference parameters.
11. Functions should be no longer than 20 lines of code. Call other functions from within a function, as needed, to adhere to this rule. Smaller is better.
12. Using *auto* as a return type:
    a. CS161 & CS162 - Functions should not use *auto* as a return type.
    b. CS260 - Functions may use the *auto* return type. However, it may be used *only* when the return type is not fixed. Never use *auto* as a return type if the data type of the return value is fixed.

13. Variables that are member variables of a class should have a trailing underscore in their label and use camel case. Ex: age_, numVehicles_

14. Variable declarations should be separated from function calls.

```
int value = GetValue();          // bad

int value = 0;                   // good, though variable declarations should be in their own
value = GetValue();              //       section, if possible
```

Local Variables

Place a function's variables in the narrowest scope possible and initialize variables in the declaration.

C++ allows you to declare variables anywhere in a function. However, for this Style Guide, you are to declare them in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to. In particular, initialization should be used instead of declaration and assignment.

```
float int_rate;
int_rate = 0.045;          // bad - declaration and initialization on separate lines
float int_rate = 0.045;    // good - declaration and initialization on one line
```

Variables used in *if*, *switch*, *while*, *do...while*, and *for* structures should be declared immediately prior to the structure (as appropriate).

Type Casting

Use the C++ style static_cast<datatype>(variable) to cast a variable. Do not use C style casting.

Incrementing and Decrementing

Use the prefix form for incrementing and decrementing (++i and --i), unless postfix form is required for a specific purpose.

A postfix increment/decrement expression evaluates to the value *as it was before it was modified*. This can result in code that is more compact but harder to read. The prefix form is generally more readable, is never less efficient, and can be more efficient because it doesn't need to make a copy of the value as it was before the operation.

White Space

Horizontal - indentation is critical for readability!

- indent all code within a function
- indent all code within a conditional structure
- indent all code within a repetition structure
- indent the second line of a line of code that spans more than one line
- NO LINE OF CODE SHOULD EXCEED 80 characters in length

Vertical - double-space...

- after opening comments
- after preprocessor directives
- after initial variable and constant declarations
- before and after conditional and repetition structures
- between functions

----------CS162 & CS260----------

Classes:

15. 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).
16. Class names must begin with an upper case letter, use an underscore as separator, and have a name that has meaning in the context of the purpose of the class.

17. Structure and class names must begin with a capital letter.

18. Variables that are member variables of a class should have a trailing underscore in their label and use camel case. Ex: age_, numVehicles_

19. Member data should always be private, with public or protected mutators and accessors provided.

20. Even within the class, member data should usually be referenced via the mutators and accessors. One exception would be in critical sections of code (e.g., inner loops) where performance considerations might require direct reference to member data.

21. Inline functions should be used *only* if the body of the function is one executable line.

22. Derived classes should use public inheritance if at all possible. Use of protected or private inheritance should be very well documented and explained.

23. 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***.

CS162 & 260: All header files must contain needed components in this order:

1. Opening comments - extensive (see below)
2. Define guard
3. Include directives
4. Namespace block opening
5. Class definition
    a. Private members (variables & functions)
    b. Protected members (variables & functions)
    c. Public members (variables & functions)
        i. accessors
        ii. mutators
        iii. helpers
        iv. friends
    d. Friend functions
6. Non-member function prototypes
7. Template file include (if necessary)
8. Namespace block closing

CS162 & 260: All class implementation and template files must contain components in this order:

1. Opening comments
2. Include directives
3. Namespace block opening
4. Function definitions, in the same order as in the class definition
5. Namespace block closing

24. Explicit declaration of the destructor, copy constructor, and assignment operators is optional. However, if not provided, a comment should be made indicating that the defaults are purposefully being used.
25. 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 (what other class this is derived from, what other classes derive 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.
//
// ********************************************************************
```

26. All header files must include #ifndef statements (define guard) to prevent multiple inclusion.

27. Statements within the header file will be ordered as follows:
    a. Comments
    b. Define guard
    c. Any required #include statements
    d. Namespace block opening
    e. Class definition
        a static constants
        b private section - member variables and private member function prototypes
        c protected section - if used
        d public section
            i static constants
            ii constructor(s) - default first, then overloaded, then copy
            iii destructor
            iv overloaded assignment operator
            v member function prototypes (or inline functions; see below), organized by purpose
                1 accessors
                2 mutators
                3 helpers
                4 friends

f. Prototypes for non-functions

28. Statements within implementation files (.cpp) should be ordered as follows:
    a. Opening comments
    b. Any required **#include** statements (only libraries for C++ should be used)
    c. Namespaces to be accessed (unless creating your own namespace(s), only std should be included)
    d. Declaration of static variables (whose scope is within this file only)
    e. Declaration of static function prototypes
    f. Class function implementations, in the order they appear in the header file
    g. Non-member function implementations

29. Comments
    b. In header files - comments at the top of the program (in addition to the comments required for all programs (#1)
        i INVARIANTS - list of each member variable and its purpose
        ii Prototypes for each member and non-member function
            1 Preconditions
            2 Postconditions
    c. In implementation files
        i Comment at the end of each functions - i.e., // end GetYear()
        ii Comments within the code of the functions sufficient to document code that may need explanation
    d. In driver (test) files - same as in #1