C++ Programming Style Guide

© Copyright 2010 by Peter C. Chapin

Last Revised: December 27, 2009

Introduction

This document describes my personal style for C++ programming. Although the style described here has many traditional elements, some of its features are unique to me.

In this document "shall" is intended to define a requirement, "should" is intended to define a recommendation, and "may" is intended to define an option. Making exceptions to this style is acceptable; exceptional circumstances are common.

1 Naming

1.1 Abstract types, classes, structs enumerations, and class templates shall be given names consisting of a capital letter followed by all lowercase letters. Multiple words in such names shall each start with a capital letter but there shall be no space or other punctuation marks between words. For example: `WordBuffer`.

1.2 Objects, functions (including methods), function templates, function parameters, and class data members, shall be given names consisting of all lower case letters. Multiple words in such names shall be separated by a single underscore character. For example: `letter_count`.

1.3 Type names introduced with a typedef declaration outside of class scope shall be given names consisting of all lower case letters with a "`_t`" suffix. Multiple words in such names shall be separated by a single underscore character. For example: `index_t`.

1.4 Type names introduced with a typedef declaration inside class scope shall be given names consisting of all lower case letters with a "`_type`" suffix. Multiple words in such names shall be separated by a single underscore character. For example: `size_type`.

1.5 Names of entities that are intended to extend or mimic the standard library should follow the same naming convention as the standard library even if that would violate rules 1.1 through 1.4 above. For example: `netbuff`.

1.6 Names of constant objects and enumerators shall be given names using mixed case where the first letter of each word in the name is uppercase and all other letters are lowercase. Multiple words in such names shall be separated by a single underscore character. For example: `Red_Light`, `Pi`.

1.7 Macros, both object-like and function-like, shall consist entirely of uppercase letters. Multiple words in such names shall be separated by a single underscore. For example: `BUFFER_SIZE`.

1.8 Abbreviations should be avoided in names except for certain very well known exceptions. Abbreviations used by the standard library are acceptable for names of things that are intended to extend or mimic the library. For example: `line_buffer` (not `line_buf`), but `max_count` is okay because "max" is a widely used abbreviation for maximum.

1.9 Names should be long and meaningful. However, a special exception is made for loop index variables: such names can be one of `i`, `j`, `k`, `m`, or `n` as is traditional. Also it is recognized that short names are meaningful in some contexts. For example, `x_coordinate` is probably the best but `x_coord` might be acceptable under some circumstances. Even `x` could be acceptable as, for example, a cartesian coordinate in a graphing program. Short names may also acceptable if they follow the names used in a reference (for example, an algorithms text book) *provided* the reference is listed in code comments.

1.10 Names, particularly of library components should be introduced into an appropriate name space. No names should be added to the std name space except for specializations of standard templates (this exception is required by the language).

1.11 Except for the std name space, using directives shall not be used. In any case, using directives shall never appear in a header file. Using declarations should be used sparingly and shall never be used at global scope. In general names should be qualified by their name space where appropriate. For example: `std::cout`.

2 Spacing/Formatting

2.1 The braces that delimit a block, class definition, array or structure initialization, or enumeration definition shall follow this format:

```
if( x == y ) {
    // Blah...
}
```

2.2 The braces that delimit a function or method definition shall follow this format:

```
void f( )
{
    // Blah...
}
```

2.3 Material inside a block shall be indented by four spaces. Declarations beneath the access specifiers in a class definition shall be intended by four spaces past that of the access specifiers. Statements beneath case labels in a switch statement shall be indented by four spaces past that of the case labels. These rules apply recursively to nested blocks, class definitions, and switch statements.

2.4 Lines that are continued on the next line should be indented four spaces past that of the continued line.

2.5 Function calls or function declarations that are broken over multiple lines should treat the parenthesis in the function call or declaration like braces for purposes of formatting. In such a case, each parameter of the call or declaration should be on a line by itself (although closely related parameters may be grouped if desired).

2.6 If there is only one statement inside a block, the braces are optional. However, the indentation rules shall still apply if that statement is on a separate line from the control structure that opens the block.

2.7 Extremely short functions (particularly in-line functions) may have bodies that are all on one line even though this would violate rule 2.2 above. For example

```
inline int max( int a, int b )
    { return ( a > b ) ? a : b; }
```

2.8 There should be no space between a unary operator and its operand.

2.9 There should be at least one space between a binary operator and its operands. Exception: the '`.`', '`->`', '`.*`', and '`->*`' operators shall not have any spaces between them and their operands.

2.10 There shall be at least one space after each comma or semicolon. However, there shall be no space before each comma or semicolon. (This rule is intended to mimic the normal rules of English writing).

2.11 There should be one space after a `(` and before a `)`.

2.12 There should be no space after the name of a function in a function call or declaration. This rule also applies to function-like macros.

2.13 There should be no space after the reserved words `if`, `for`, or `while`.

2.14 A source file shall be formatted assuming that a fixed width font is used. Alignment may be used between source lines to emphasize relationships between those lines. In fact such alignment is encouraged. RATIONALE: Although proportional width fonts are popular in most writing, and are even occasionally used in programming, they don't make aligning one line with another very feasible. Such alignment is an excellent device for highlighting the similarity between related, but complicated operations.

2.15 A source file shall be formatted so that no line is longer than 96 characters. RATIONALE: To maximize the likelihood that the source file will display properly in a wide range of environments, very long lines should be avoided. The 96 character limit, while arbitrary, allows reasonably long lines without getting into problems of line wrapping or truncation on most systems.

3 Usage

3.1 Every object should be initialized when it is declared. Every object's point of declaration should be close to where it is first used. For loop indexes should be declared as part of the loop except in the cases where the value of the index variable is needed outside of the loop.

3.2 Only new style casts shall be used. RATIONALE: New style casts provide more information to the reader of the program. Also, since they are more specific than the general purpose old style cast, their consistent usage may help the compiler detect inappropriate casts.

3.3 All expressions shall have appropriately matched types. Explicit casts shall be used to indicate intent in cases where the types do not match naturally. Casts shall be used even when automatic type conversion rules would cause the same conversion. RATIONALE: Although C++ provides many implicit type conversions, one should program as if it did not have those conversions. Such an approach produces more robust programs. If available a tool should be used to verify type usage strictly. See 3.4 below.

3.4 Programs should be developed using the highest reasonable warning level provided by the compiler. Code should be written so that it compiles without warnings. Exception: It is understood that some compilers provide very aggressive warning levels that might be considered "unreasonable" for general use.

3.5 An explicit return statement may be placed at the end of void functions.

3.6 Every possible value of a switch statement's controlling expression should be accounted for in the switch statement's case list. If necessary one should include an empty default case to explicitly document one's intention to ignore values not otherwise mentioned.

3.7 Any class with virtual methods shall have a virtual destructor.

3.8 Any class that has either a destructor, a non-trivial copy constructor, or a non-trivial assignment operator should have all three of these methods. If any of these methods are not actually necessary, explicit documentation to that effect shall be added to the class.

3.9 All data members of a class should be private. Protected data may be used in some cases. Structures (as opposed to classes) may have public data members.

3.10 Inline methods should be defined outside the class definition. Class definitions should only contain method declarations. RATIONALE: Although this is awkward at times, it is important to separate the interface of a type from its internal structure. Making a method inline is an implementation detail; it should not be part of a class's interface. However it is understood that the syntactic overhead of defining an inline method outside the class definition, particular in the case of template classes, is sometimes very high. In such a case, this rule can be cautiously violated.

4 Documenting

4.1 Each source file shall have a comment header containing the name of the file, the author, and a description of the file's contents. The comment header may also include a "to-do" list, a revision history (in cases where a revision control system is not being used), relevant references, and licensing information.

4.2 Each module in the program should have associated documentation that describes how to use the module and, optionally, how the module works. The documentation for a module may include pseudo-code, UML diagrams, sample code, and any other information that will aid in the understanding of the module.

4.3 Each declaration in a header file should have an associated comment that describes the interface to the declared entity. If a module has associated documentation, these comments may be sparse (perhaps just references to the documentation) otherwise they should be complete enough to serve as the documentation. An inline documentation tool such as Doxygen may be used to create documentation but the use of such tools is neither required nor necessarily recommended. RATIONALE: Inline documentation tools have their uses, but the large comments they require (if good quality documentation is desired) can be distracting in the C++ source files. Some programs may be better served with out-of-line documentation.

4.4 Each function in an implementation source file should have a comment header containing a brief description of how the function works. Note that implementation comments should not describe the interface to a function if the interface has been described elsewhere.

4.5 The body of each function shall be broken into logically distinct parts with blank lines separating the parts. A short comment may be provided for each of these parts.