Chetu Inc.

Chetu
Delivering World-Class IT Solutions

C++ CODING GUIDELINES

A Generic Recommendation to all

From ETC

Disclaimer of Content

The content of this document is the compilation of different industry standards and guidelines related information found through different available open channels. This document is created by formatting and putting all together the Information collected through different mediums of internet blogs, forums, user feedbacks, project development team experience and feedback along with the generic recommendations of different technology providers to use them in industry standards projects or applications.

CONTENTS

1.0 INTRODUCTION

The intent of this document is to provide direction and guidance to C++ programmers that will enable them to employ good programming style and proven programming practices leading to safe, reliable, testable, and maintainable code.

Note that the guidelines contained within this document will not guarantee the production of an error-free, safe product. However, adherence to these guidelines will help programmers produce clean designs that minimize common sources of mistakes and errors.

Note that the guidelines contained within this document is C++ inhancement to C coding guideline updated on backbone at "http://backbone:9090/Education/Standards/Technologies/C%20and%20C++/C_Coding_Guidelines.pdf". We recommend everyone to read C coding guideline along with this document for complete C++ guideline/recommendation.

2.0 GUIDELINE

2.1 Header Files

- Every .cpp file should have an associated .h file.
- All header files should be self-contained. Specifically, a header should have header guards, should include all other headers it needs, and should not require any particular symbols to be defined.
- If a template or inline function is declared in a .h file, define it in that same file.
- All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be <*PROJECT*>_<*FILE*>_H_.
- Avoid using forward declarations where possible. Just #include the headers you need.
- Define functions inline only when they are small, say, 10 lines or less.
- Use standard order for readability and to avoid hidden dependencies: Related header, C library, C++ library, other libraries' .h, your project's .h.
- All of a project's header files should be listed as descendants of the project's source directory without use of directory shortcuts.

2.2 Scoping

2.2.1 Namespaces

- Place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Unnamed namespaces in .cpp files are encouraged. Do not use *using-directives*. Do not use inline namespaces.
- Unnamed namespaces are allowed and even encouraged in .cpp files, to avoid link time naming conflicts.
- Do not use unnamed namespaces in .h files.
- Do not declare anything in namespace std, including forward declarations of standard library classes.
- Do not use using-declarations in .h files except in explicitly marked internal-only namespaces, because anything imported into a namespace in a .h file becomes part of the public API exported by that file.
- Do not use inline namespaces.

2.2.2 Functions and Variables

- Prefer placing nonmember functions in a namespace, putting nonmember functions in a namespace avoids polluting the global namespace.
- 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. We encourage you to declare them in as local a scope as possible, and as close to the first use as possible.
- Variables needed for if, while and for statements should normally be declared within those statements, so that such variables are confined to those scopes.
- Variables of class type with static storage duration are forbidden.
- Objects with static storage duration, including global variables, static variables, static class member variables, and function static variables, must be Plain Old Data (POD): only ints, chars, floats, or pointers, or arrays/structs of POD.
- If you need a static or global variable of a class type, consider initializing a pointer (which will never be freed), from either your main() function or from pthread_once().

2.3 Classes

2.3.1 Constructors

- Avoid virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error.
- Constructors should never call virtual functions.

2.3.2 Implicit Conversions

- Do not define implicit conversions. Use the explicit keyword for conversion operators and single-argument constructors.
- Type conversion operators, and constructors that are callable with a single argument, must be marked explicit in the class definition.
- Constructors that cannot be called with a single argument should usually omit explicit. Constructors that take a single std::initializer_list parameter should also omit explicit, in order to support copy-initialization.

2.3.3 Copyable and Movable Types

- Support copying and/or moving if it makes sense for your type. Otherwise, disable the implicitly generated special functions that perform copies and moves.
- Due to the risk of slicing, avoid providing an assignment operator or public copy/move constructor for a class that's intended to be derived.
- If you do not want to support copy/move operations on your type, explicitly disable them using = delete or whatever other mechanism your project uses.

2.3.4 Structs vs. Classes

- Use a struct only for passive objects that carry data; everything else is a class.

- All inheritance should be public. If you want to do private inheritance, you should be including an instance of the base class as a member instead.
- Make your destructor virtual if necessary. If your class has virtual methods, its destructor should be virtual.
- Make data members private, unless they are static const.

2.3.5 Operator Overloading

- Overload operators judiciously. Do not create user-defined literals.
- Define overloaded operators only if their meaning is obvious, unsurprising, and consistent with the corresponding built-in operators.
- Define operators only on your own types. More precisely, define them in the same headers, .cpp files, and namespaces as the types they operate on.
- Prefer to define non-modifying binary operators as non-member functions.
- Don't go out of your way to avoid defining operator overloads.
- Do not overload &&, ||, , (comma), or unary &. Do not overload operator"", i.e. do not introduce user-defined literals.

2.3.6 Declaration Order

- Class definition should start with its public: section, followed by its protected: section and then its private: section. If any of these sections are empty, omit them. Within each section, the declarations generally should be in the following order:
    - Typedefs and Enums
    - Constants (static const data members)
    - Constructors
    - Destructor
    - Methods, including static methods
    - Data Members (except static const data members)
- Do not put large method definitions inline in the class definition. Usually, only trivial or performance-critical, and very short, methods may be defined inline.

2.4 Functions

- Prefer small and focused functions.
- All parameters passed by reference must be labeled const.
- Within function parameter lists all references must be const.
- If you want to overload a function, consider qualifying the name with some information about the arguments.
- Do not use default function parameters..

2.5 Other C++ Features

2.5.1 Rvalue References

- Use rvalue references only to define move constructors and move assignment operators,

2.5.2 Variable-Length Arrays and alloca()

- Do not use variable-length arrays or alloca(). Use a safe allocator instead, such as vector or std::unique_ptr<T[]>.

2.5.3 Friends

- Friends should usually be defined in the same file so that the reader does not have to look in another file to find uses of the private members of a class

2.5.4 Casting

- Use C++ casts like static_cast<>(). Do not use other cast formats like int y = (int)x; or int y = int(x);.
- Do not use C-style casts. Instead, use these C++-style casts.

2.5.5 Preincrement and Predecrement

- Use prefix form (++i) of the increment and decrement operators with iterators and other template objects.

2.5.6 Use of const

- If a function does not modify an argument passed by reference or by pointer, that argument should be const.
- Declare methods to be const whenever possible. Accessors should almost always be const. Other methods should be const if they do not modify any data members, do not call any non-const methods, and do not return a non-const pointer or non-const reference to a data member.
- Consider making data members const whenever they do not need to be modified after construction.

2.5.7 Integer Types

- You should always use types like int16_t, uint32_t, int64_t in preference to short, unsigned long long and the like, when you need a guarantee on the size

of an integer. The sizes of integral types in C++ can vary based on compiler and architecture.

- You should not use the unsigned integer types such as uint32_t, unless there is a valid reason such as representing a bit pattern rather than a number, or you need defined overflow modulo 2^N. In particular, do not use unsigned types to say a number will never be negative. Instead, use assertions for this.
- If your code is a container that returns a size, be sure to use a type that will accommodate any possible usage of your container. When in doubt, use a larger type rather than a smaller type.
- Use care when converting integer types. Integer conversions and promotions can cause non-intuitive behavior.

2.5.8 64-bit Portability

- Code should be 64-bit and 32-bit friendly. Bear in mind problems of printing, comparisons, and structure alignment.
- Use the LL or ULL suffixes as needed to create 64-bit constants.

2.5.9 Preprocessor Macros

- Be very cautious with macros. Prefer inline functions, enums, and const variables to macros.
- Instead of using a macro to inline performance-critical code, use an inline function.
- Instead of using a macro to store a constant, use a const variable. Instead of using a macro to "abbreviate" a long variable name, use a reference.
- Instead of using a macro to conditionally compile code ... well, don't do that at all (except, of course, for the #define guards to prevent double inclusion of header files). It makes testing much more difficult.

- The following usage pattern will avoid many problems with macros; if you use macros, follow it whenever possible:
    - Don't define macros in a .h file.
    - #define macros right before you use them, and #undef them right after.
    - Do not just #undef an existing macro before replacing it with your own; instead, pick a name that's likely to be unique.
    - Try not to use macros that expand to unbalanced C++ constructs, or at least document that behavior well.
    - Prefer not using ## to generate function/class/variable names.

2.5.10 0 and nullptr/NULL

- Use 0 for integers, 0.0 for reals, nullptr (or NULL) for pointers, and '\0' for chars.

2.5.11 sizeof

- Prefer sizeof(*varname*) to sizeof(*type*).

2.5.12 auto

- Use auto to avoid type names that are just clutter. Continue to use manifest type declarations when it helps readability, and never use auto for anything but local variables.

2.5.13 Braced Initializer List

- Never assign a *braced-init-list* to an auto local variable. In the single element case, what this means can be confusing.

2.5.14 Lambda expressions

- Use lambda expressions where appropriate. Avoid default lambda captures when capturing this or if the lambda will escape the current scope.

2.6 Naming

2.6.1 General Naming Rules

- Names should be descriptive; eschew abbreviation.
- Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word.

2.6.2 File Names

- Filenames should be all lowercase and can include underscores (_) or dashes (-). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "_".
- C++ files should end in .cpp and header files should end in .h. Files that rely on being textually included at specific points should end in .inc .
- Do not use filenames that already exist in include, such as db.h.
- In general, make your filenames very specific.
- Inline functions must be in a .h file. If your inline functions are very short, they should go directly into your .h file.

2.6.3 Type Names

- Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass, MyExcitingEnum.
- The names of all types — classes, structs, typedefs, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores.

2.6.4 Variable Names

- The names of variables and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores.

2.6.5 Constant Names

- Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case.

2.6.6 Function Names

- Ordinarily, functions should start with a capital letter and have a capital letter for each new word (a.k.a. "upper camel case" or "Pascal case"). Such names should not have underscores. Prefer to capitalize acronyms as single words (i.e. StartRpc(), not StartRPC()).

2.6.7 Namespace Names

- Namespace names are all lower-case. Top-level namespace names are based on the project name .

2.6.8 Enumerator Names

- Enumerators should be named *either* like **constants** or like **macros**: either kEnumName or ENUM_NAME.

2.6.9 Macro Names

- If you define a macro, they're like this: MY_MACRO_THAT_SCARES_SMALL_CHILDREN.

2.7 Comments

- When writing your comments, write for your audience: the next contributor who will need to understand your code. Be generous — the next one may be you!

2.7.1 Comment Style

- Use either the // or /* */ syntax, as long as you are consistent.

2.7.2 File Comments

- Start each file with license boilerplate, followed by a description of its contents.
- Every file should contain license boilerplate. Choose the appropriate boilerplate for the license used by the project.
- If you make significant changes to a file with an author line, consider deleting the author line.
- Every file should have a comment at the top describing its contents, unless the specific conditions described below apply.
- Do not duplicate comments in both the .h and the .cpp. Duplicated comments diverge.
- A file-level comment may be omitted if the file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration.

2.7.3 Class Comments

- Every class definition should have an accompanying comment that describes what it is for and how it should be used.
- The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class. Document the synchronization assumptions the class makes, if any. If an instance of the class can be accessed by multiple threads, take extra care to document the rules and invariants surrounding multithreaded use.

2.7.4 Function Comments

- Declaration comments describe use of the function; comments at the definition of a function describe operation.
- Types of things to mention in comments at the function declaration:
    - What the inputs and outputs are.
    - For class member functions: whether the object remembers reference arguments beyond the duration of the method call, and whether it will free them or not.
    - If the function allocates memory that the caller must free.
    - Whether any of the arguments can be a null pointer.

    - If there are any performance implications of how a function is used.
    - If the function is re-entrant. What are its synchronization assumptions?
- When documenting function overrides, focus on the specifics of the override itself, rather than repeating the comment from the overriden function.
- When commenting constructors and destructors, remember that the person reading your code knows what constructors and destructors are for, so comments that just say something like "destroys this object" are not useful. Document what constructors do with their arguments, and what cleanup the destructor does. If this is trivial, just skip the comment. It is quite common for destructors not to have a header comment.

2.7.5 Variable Comments

- Each class data member (also called an instance variable or member variable) should have a comment describing what it is used for. If the variable can take sentinel values with special meanings, such as a null pointer or -1, document this.
- As with data members, all global variables should have a comment describing what they are and what they are used for.

2.7.6 Implementation Comments

- In your implementation you should have comments in tricky, non-obvious, interesting, or important parts of your code.
- Tricky or complicated code blocks should have comments before them. Example:
- Also, lines that are non-obvious should get a comment at the end of the line. These end-of-line comments should be separated from the code by 2 spaces.
- Note that there are both comments that describe what the code is doing, and comments that mention that an error has already been logged when the function returns.
- If the argument is a literal constant, and the same constant is used in multiple function calls in a way that tacitly assumes they're the same, you should use a named constant to make that constraint explicit, and to guarantee that it holds.
- Replace large or complex nested expressions with named variables.
- As a last resort, use comments to clarify argument meanings at the call site.
- Do not state the obvious. In particular, don't literally describe what code does, unless the behavior is nonobvious to a reader who understands C++ well. Instead, provide higher level comments that describe *why* the code does what it does, or make the code self describing.
- Self-describing code doesn't need a comment.

2.7.7 Punctuation, Spelling and Grammar

- Pay attention to punctuation, spelling, and grammar; it is easier to read well-written comments than badly written ones.

2.7.8 TODO Comments

- Use TODO comments for code that is temporary, a short-term solution, or good-enough but not perfect.

2.7.9 Deprecation Comments

- Mark deprecated interface points with DEPRECATED comments.

2.8 Formatting

2.8.1 Line Length

- Each line of text in your code should be at most 80 characters long.
- An #include statement with a long path may exceed 80 columns.

2.8.2 Non-ASCII Characters

- Non-ASCII characters should be rare, and must use UTF-8 formatting.
- Hex encoding is also OK, and encouraged where it enhances readability.

2.8.3 Spaces vs. Tabs

- Use only spaces, and indent 2 spaces at a time.

2.8.4 Function Declarations and Definitions

- Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.
- Some points to note:
    - Choose good parameter names.
    - Parameter names may be omitted only if the parameter is unused and its purpose is obvious.
    - If you cannot fit the return type and the function name on a single line, break between them.
    - If you break after the return type of a function declaration or definition, do not indent.
    - The open parenthesis is always on the same line as the function name.
    - There is never a space between the function name and the open parenthesis.
    - There is never a space between the parentheses and the parameters.
    - The open curly brace is always on the end of the last line of the function declaration, not the start of the next line.

- The close curly brace is either on the last line by itself or on the same line as the open curly brace.
- There should be a space between the close parenthesis and the open curly brace.
- All parameters should be aligned if possible.
- Default indentation is 2 spaces.
- Wrapped parameters have a 4 space indent.

2.8.5 Function Calls

- Either write the call all on a single line, wrap the arguments at the parenthesis, or start the arguments on a new line indented by four spaces and continue at that 4 space indent. In the absence of other considerations, use the minimum number of lines, including placing multiple arguments on each line where appropriate.

2.8.6 Braced Initializer List Format

- Format a braced initializer list exactly like you would format a function call in its place.

2.8.7 Conditionals

- Prefer no spaces inside parentheses. The `if` and `else` keywords belong on separate lines.

2.8.8 Pointer and Reference Expressions

- No spaces around period or arrow. Pointer operators do not have trailing spaces.

2.8.9 Boolean Expressions

- When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

2.8.10 Return Values

- Do not needlessly surround the `return` expression with parentheses.
- Use parentheses in `return expr;` only where you would use them in `x = expr;`.

2.8.11 Preprocessor Directives

- The hash mark that starts a preprocessor directive should always be at the beginning of the line.

- Even when preprocessor directives are within the body of indented code, the directives should start at the beginning of the line.

2.8.12 Class Format

- Sections in public, protected and private order, each indented one space.
- Things to note:
    - Any base class name should be on the same line as the subclass name, subject to the 80-column limit.
    - The public:, protected:, and private: keywords should be indented one space.
    - Except for the first instance, these keywords should be preceded by a blank line. This rule is optional in small classes.
    - Do not leave a blank line after these keywords.
    - The public section should be first, followed by the protected and finally the private section.

2.8.13 Constructor Initializer Lists

- Constructor initializer lists can be all on one line or with subsequent lines indented four spaces.

2.8.14 Namespace Formatting

- The contents of namespaces are not indented.
- Do not indent within a namespace:
- When declaring nested namespaces, put each namespace on its own line.

2.8.15 Horizontal Whitespace

- Use of horizontal whitespace depends on location. Never put trailing whitespace at the end of a line.

2.8.16 Vertical Whitespace

- Minimize use of vertical whitespace.
- Don't use blank lines when you don't have to. In particular, don't put more than one or two blank lines between functions, resist starting functions with a blank line, don't end functions with a blank line, and be discriminating with your use of blank lines inside functions. The more code that fits on one screen, the easier it is to follow and understand the control flow of the program.
- Some rules of thumb to help when blank lines may be useful:
    - Blank lines at the beginning or end of a function very rarely help readability.
    - Blank lines inside a chain of if-else blocks may well help readability.

2.9 General

- Use common sense and *BE CONSISTENT*.
- If you are editing code, take a few minutes to look at the code around you and determine its style. If they use spaces around their if clauses, you should, too. If their comments have little boxes of stars around them, make your comments have little boxes of stars around them too.