1. Design
   1. Design Rules (Software Practice)

1.1.1．The Principal of DRY

* + In principle, the first letter "Don't repeat yourself."
  + The principle is "Avoid repeated work in software development and duplication of artifacts".
  + The programmer's workflow in software development is as follows.
    - Get latest source code from VCS (abbreviation of version control system, subversion, git etc)
    - Understanding the specifications as needed, detailed design
    - Programming
    - Build and deploy source code
    - Test new source code
    - Regression test around new source code
    - Committing source code to VCS
  + With duplication of deliverables,
    - Clone code
    - Comments that describe in natural language what is written in the source code
    - Detailed design document that can be easily understood by reading the source code.

1.1.2．Boy Scout Rule

* The first step to follow this rule is to review the changes before committing to VCS.

1.1.3．The principle of YAGNI

* The function (requirement) is not implemented until it is actually needed.

1.1.4．Inside-Out principle

* + This principle is used when developing a GUI application with an MVC architecture
  + It is a restriction to the development procedure that "Development does not start from GUI (View), start from Model".
  + Start development from Model and make it possible to execute it from external "regression test software". Then, if you make it possible to execute it from View, architecture based on MVC, you can get software that can test it automatically.

1.1.5．Name and Conquer

* + If the object is not properly named, it is difficult for us to remember it correctly. If you cannot remember exactly, you cannot understand exactly. Even if understood, it is difficult to share that understanding with others. On the other hand, by giving it a name, the understanding of the object starts and it becomes possible to share the understanding with others.

1.1.6．Sharpening the saw blade

* + Wear touch typing and development tool shortcuts.
  + Use a good editor or IDE.
  + Don't rely solely on printf debugging, use a good debugger.
  + Implement "[0.0 DRY principle] (---)" and introduce and develop tools to eliminate repetitive manual work.
  + Find bugs easily with minimal effort using static and dynamic analysis tools.
  + Use open source.
  + Learn new languages ​​and the latest features of familiar languages.
  + Understand and implement efficient new processes.
  + Get the latest technical information from books and the web regularly.

1.1.7．The Law of Demeter

* + An object has knowledge of other structures and properties (relative to other objects), then the premise should be minimized.

1.1.8．Conway’s Law

* + Build an architecture that divides the entire software horizontally into packages.
  + The team based on the domain organization is responsible for developing this package.
  + Modification of package interfaces and responsibilities requires coordination meetings between organizations.

1.1.9．SOLID

SOLID is a principle of five object orientation (OOD / OOP) as shown below.

1.1.9.1．The principle of single responsibility (SRP)

* + One class must have only one responsibility (function).
  + One class must be created to be changed for one single reason.

1.1.9.2．Open-Closed Principle (OCP)

* + Class must be open for the extension,
  + Class must be closed for modification

1.1.9.3．Liskov's substitution principle (LSP)

* + Even when passing derived class instead of base class to X (function or class) using base class, X must be able to operate normally without knowing its actual type.
  + Preconditions cannot be enhanced with derived classes.

In other words, you should not create a derived class with a precondition that is stronger than the base class.

* + Postcondition cannot be weakened by derived class.

In other words, you should not create a derived class that has a weaker postcondition than the base class.

1.1.9.4．Interface separation law (ISP)

* + Classes should not force their clients to rely on methods they do not use.
  + Do not make class interface huge
  + Don't define irrelevant classes in the header file.
  + Header files should not include header files that are unnecessary for compiling the file.

1.1.9.5．Dependency reversal principle (DIP)

* + Higher level modules should not depend on lower level modules.
  + Abstracts should not depend on figurative.
  1. Design Patterns

There are 14 design patterns:

1.2.1．Pimpl Ideom

* Pimpl Ideom used when you do not want to propagate the implementation details of class A (A.cpp, A.h) to the class that uses it.
* In general, parsing of the STL library consumes a lot of CPU time. If class A uses STL as a member and includes that STL header file in A.h: Each time you compile a file that includes A.h, its STL is parsed. This will consume CPU time and slow down the overall project build. It is an effective means to avoid these problems in advance, but As the trade-off, the execution speed is slower.

1.2.2．clone Ideom

It is an ideom to avoid slicing by object copy.

1.2.3．NVI Ideom

* + Assuming that there is a class A that provides services by a member function A :: f () and a client class B that uses it:

class B strongly depends on A :: f ().

* + On the other hand, if A :: f () is virtual and class AD derived from class A overrides A :: f (),

AD :: f () strongly depends on A :: f ().

* + Under this condition, dependence on A :: f () is concentrated, and it is assumed that the cost of A :: f () modification and function addition will be high.
  + This ideom of "do not make virtual member functions public" alleviates this problem.

1.2.4．New overload

* In embedded software development in particular, it may be desirable to limit the use of new throughout the system.

1.2.5．DI

* + - Generate class NotDIDepended in constructor Suppose that class NotDI exists.
    - In this case, class NotDI depends on the instance of class NotDIDepended.
    - Such dependencies reduce the availability and testability of class NotDI.
    - This is "if class NotDIDepended to be the class that wraps the database,
    - It can be easily understood from the fact that a database is required to test class NotDI.

1.2.6．Singleton

This pattern facilitates the disciplined use of global objects.

Note the following when using this pattern.

* Singleton is particularly susceptible to pattern monkey disease among design patterns.

Singleton should be used sparingly, knowing that it is "almost a global variable".

* When defining Singleton, define the following two.
  + Static member function that returns an instance (Inst ())
  + Static member function that returns const instance (InstConst ())

Because constructor calls to Singleton objects were not thread safe prior to C ++ 98,

"Use Double Checked Locking to avoid conflicts" or

"InstConst () of each Singleton is called from the main thread before starting other threads"

Although it was necessary, the constructor call of Singleton object from C ++ 11 like the following example,

Because it became thread safe, this kind of black art became unnecessary.

1.2.7．State

* A pattern to separate and describe the state of an object and its associated behavior.
* This limits the scope of code modification accompanying addition and reduction of states (OCP).
* Also, the corrections become clear regarding the addition of the public member function of the object.

1.2.8．Observer

* There is a class Subject and multiple class Observer N (N = 0, 1, 2 ...),
* It is a pattern used when this relation must satisfy the following conditions.
  + ObserverN receives notification of Subject change.
  + Subject must not depend on ObserverN.
* Model for implementing a GUI application by MVC is Subject, and View is ObserverN.
* Subject is not dependent on ObserverN even in file dependencies

(If you replace it with MVC, you can say that Model is not dependent on View).

1.2.9．Null Object

* Conditional branching is prone to bugs and leads to an increase in the number of test steps.

Therefore, as a matter of course, it is better to eliminate the conditional branch that can be eliminated.

1.2.10．Factory

* It is a pattern that uses a dedicated function (Factory function) to create an object.
* Used when you do not want to directly depend on an object for classes and functions that create objects.

Often used in combination with DI.

1.2.11．Template method

* A member function (template method) that defines the form (form etc.) of the template, the behavior to fill it,

It is a pattern used when separating member functions that define data.

1.2.12．Proxy

* When implementing a class that provides services across CPU space, threads, and processes, for clients who receive the service, by not being aware of the location of the service provision class and the internal structure (socket, pipe, msg\_q, etc.)

It is a pattern to prevent the occurrence of unnecessary dependencies and implicit dependencies.

1.2.13．Strategy

* The behavior of the function f () is
  + Overall control
  + Partial behavior (finding some condition etc)
* Function f () if you split it (pattern like std :: qsort ())
  + Function g () that performs "overall control"
  + Object that specifications "partial behavior" (Strategy object)

Strategy is a pattern in which the object can be externally passed as a parameter of g (). Use this pattern if there are many variations in the Strategy object.

* "Function to track directories recursively and return a list of files matching the attribute specified by the argument"

1.2.14．RAII

* RAII is an abbreviation for "Resource Acquisition Is Initialization",
* It is a pattern or ideom that links resource allocation and release to object initialization and destruction.
* Especially when memory allocation using new

It is difficult to prevent memory leaks if you do not follow RAII.

1. Coding Guidelines
   1. Coding Rules Solid

2.1.1．Single Responsibility Principle

* One class must have only one responsibility (function) and must be created to be changed for one single reason.

2.1.2．Open-Closed Principle

* Class must be open for the extension and closed for modification.

2.1.3．Liskov Substitution Principle

* Where you are using the base class, you should work without problems if you use the derived class instead.
* Preconditions cannot be enhanced with derived classes.
* In other words, you should not create a derived class with a precondition that is stronger than the base class.
* Postcondition cannot be weakened by derived class.
* In other words, you should not create a derived class that has a weaker postcondition than the base class.

2.1.4．Interface Segregation Principle

* Clients should not be forced to rely on methods they don't use.
  + Do not make class interface huge
  + Don't define irrelevant classes in the header file.
  + Header files should not include header files that are unnecessary for compiling the file.

2.1.5．Dependency Inversion Principle

* Higher level modules should not depend on lower level modules.
* Abstraction must not depend on details. The details should depend on the abstraction.
  1. Coding Rules Practice

2.2.1．Types and Instances

Basic Type:

* Do not use basic types (int, char, etc.) as they are, but use the type alias defined in cstdint.
* For holding string literals, use const char \*.
* Use size\_t to hold the value of sizeof and the length of the array.
* Use unsigned type if you know that the value is not negative, if there is an operation with signed type, make it signed type even if the variable is not negative.
* Use int32\_t for integer operations, unless you have a specific reason.
* Prohibits the use of char and bool in arithmetic operations.

In particular, prohibit implicit conversion of variables of type char to other integer types (C ++ undefined behavior).

Floating point type:

* Allow use of Floating point types only if you need float or double dynamic range. If you just want to deal with decimals, use fixed point.
* Do not use INF or NAN in calculations.
* Do not compare float or double instances with ==

Enum:

* To use type checking and compiler static analysis (case omission at switch) effectively use enum to define unnecessary constants of concrete values ?? and continuous constants.
* Prohibit the use of the old enum unless there is a special reason (such as applying to array indexing). Use scoped enum instead.
* When setting values, write them first, if you do not need to set the value (if the specific value does not make sense), do not set the value.
* When using enum as array index,
  + Define old enum in struct instead of scoped enum.
  + The first defined enum member is initialized to 0.
  + The last element ends with \_MAX and shows the maximum value.
* Don't cast to enum.
* Don't use enum instead of integer constant for operation. Use constexpr instead.

Bit field:

* The type of bit field must be unsigned int.

Struct:

* Use only as a data holder (POD).
* Do not have member functions other than constructors
* It is not necessary to specify default or delete for member functions generated by the compiler.
* Prohibit inheritance. final is unnecessary.

type alias:

* Use using instead of typedef except for type aliases shared with C.

Declare and Define Identifiers:

* Identifiers must not have more than one definition.
* An identifier must not have more than one declaration except for a type forward declaration.
* If the identifier has a declaration other than a forward declaration, the header containing that declaration from the .cpp file that defines it, then it must be included.

const / constexpr instances:

* The value of const is determined at runtime initialization, so it is unchanged and the value of constexpr is fixed at compile time. Naturally, it is immutable at runtime.
* For instance, pointer to instance, reference to instance etc, always add constexpr to an instance to which constexpr can be added. You cannot add constexpr, but always add const to an instance to which const can be added.

(It is especially important to add const in references and pointers).

Auto:

* Allow creation of objects by auto only if the type of the right-hand expression is clear in the source code.
* Be careful not to add &, \*, const, etc. when using auto.

Instance Initialization:

* All instances must be initialized at the same time as the definition. If the initial value of the pointer variable is not decided at definition, initialize with nullptr.
* Define and initialize basic type constants with constexpr.
* Do not perform initialization directly dependent on the linked object, because the initialization order is undefined.

Uniform initialization:

* Do not use uniform initialization to initialize built-in types, references, pointers, for non-container class initialization and for the return value of the function.
* Use uniform initialization to initialize arrays and structures and for container classes, to initialize each element with its own object.
* For container classes, do not use uniform initialization for other initializations (such as specifying the vector length).
* If a class initializes a member variable using "initialize with non-static member variable initializer", and if its member variables are initialized in the default constructor, you may use it.

2.2.2．Classes and instances

Class Size:

* Except when there is no other way around, the class declaration should be around 200 lines including comments and member functions defined in class must not exceed 10 lines.
* Except when there is no other way, the number of public member functions can be up to about seven and the number of variables holding the state of the object can be up to about four.
* Cohesion is measured by the metric Lack of Cohesion in Methods.
  + The degree of aggregation is low if this value is close to 1 and the degree of aggregation is high if this value is close to 0.
  + Cohesion is likely to be low if there are many member variables
  + Cohesion is likely to be low if there are many member functions
* Except for simple data holders and classes that delegate most of their behavior to other classes, it must be designed to have a high degree of cohesion.

Access level and hiding:

* Specify the access level of class. Unless there is a special reason, the access level is describe in public, protected and private in order from the top.
* Make all member variables private.
  + Prohibit public member variables.
  + Prohibit protected member variables. However, the unit test class is an exception.
* If you want to access member variables, make them pass accessor member functions. Even so, setter is modest used for define a protected getter if you need the value of the base class variable from the derived class.
* Do not return handlers for member variables, as access level encapsulation is broken. If that is unavoidable, return a handler with const.
* If there is any other implementation method, friend is prohibited. However, the following are exceptions.
  + Operator overload I / O operator
  + Arithmetic operator
  + Comparison operator
  + Unit test class
* Follow the NVI Ideom. In other words, it prohibits virtual public member functions. Make a virtual function private or protected and call it from a public non-virtual member function.

Inheritance / Derivation:

* The number of derivations is up to about 2 times.
* Implementation inheritance is private, but reconsider if implementation inheritance is really necessary. When implementing IS-IMPLEMENTED-IN-TERMS-OF, use inheritance, stratification, HAS-A, and delegation instead of inheritance.
* Add final to classes that you do not want to derive. Most classes should not be derived, so most classes have final attached.
* Inheriting a class with a non-virtual destructor is prohibited unless it is inevitable.
* In initialization of base class by default constructor call, write it in the constructor's initializer list.
* Interface inheritance is done in public. In that case, you must follow Liskov's substitution principle.
* When using multiple inheritance, do not have member variables except for one of multiple base classes.
* When using multiple inheritance, the same base class must not appear more than once in the lineage of inheritance.

Non-static member variable / constant initialization:

* All non-static member variables must be initialized at the end of the constructor.
* There are three initializations of non-static member variables, but the above one is used with priority.
  + Initialization method 0: Initialization with non-static member variable initializer
  + Initialization method 1: Initialization by constructor non-static member initializer
  + Initialization method 2: Initialization of non-static member variables in constructor
* Non-static member variables are initialized according to the order defined in class (Do not initialize other variables using variables before being initialized).
* Do not mix initialization method 0 and initialization method 1 in one class.

(When the initialization method 0 and the initialization method 1 are performed on the same variable, the initialization method 0 is not performed). Therefore, if at least one member variable requires initialization method 1, initialization of all variables is performed by initialization method 1.

Static member variable / constant initialization:

* Static (and not constexpr) member variables are declared in the header file and defined and initialized in .cpp.
* If the value of a built-in static member variable is 0, no initialization is necessary (initialized to 0).
* A static const member constant initialized in class is not defined because its meaning was lost in C ++ 11. Instead, it is defined and initialized as a static constexpr member constant

(Only member constants initialized outside of class are set to static const).

* Static constexpr member constants are dependent in the header file in which the class is declared definition is allowed.
* If the static constexpr member constant is not dependent in the header file, .cpp unnamed in namespace is defined and initialized

(In other words, don't define it as a member of class. This will prevent unnecessary compilation).

Mutable:

* The use of mutable is prohibited except for member variables for exclusive control (std :: mutex or \_ etc.).

Slicing:

* Do not slice objects.

Object Lifetime:

* Objects have the following lifetimes:
  + The lifetime of statically generated objects
  + The lifetime of the object created in thread\_local
  + The lifetime of the object created by new
  + Lifetime of object on stack
  + Lifetime of temporary object (rvalue)
* Do not access objects before or after the lifetime.
* Do not disclose the handle of an object on the stack to the outside of the function (the handle is points to an object whose lifetime has expired).
* Prohibit the use of thread\_local except for debugging purposes such as logging.
* Do not assign rvalues to non-const references (this is an error for most compilers). Use of rvalue reference (type &) is limited to function arguments only.
* Do not hold temporary object references in reference or pointer type member variables. A good way to protect this is to not hold the handle of an unowned object in a member variable.

2.2.3．Function / Member Function

Out-of-class functions:

* Prohibit out-of-class functions except in the following cases.
  + Linkage with C language is required.
  + Define overloads for binary operators.

Member Functions:

* Consider actively making member functions static.
* If const can be added to a member function, be sure to add it.
  + Always set const to getXXX (getter).
  + Do not add const to functions like SetPtr.

Compiler-Generated Member Functions:

* The member functions automatically generated by the compiler are as follows.
  + Default constructor
  + copy constructor
  + copy assignment operator (operator =)
  + move constructor
  + move assignment operator
  + Destructor

Constructor:

* Do not define multiple constructors if they can be aggregated into a constructor with default arguments.
* You cannot call the derived class virtual function until the object initialization is complete. Do not call virtual functions in the constructor.
* If you define copy constructor and move constructor, the class has pointer type member variable, then you have to do deep copy.
* Implicit type conversion is not performed for single argument constructors that are not copy constructors, add explicit.

move constructor, move assignment operator:

* Don't throw an exception outside.
* move constructor, move assignment operator must be noexcept

(If you put it in a std container, the copy constructor may be called if it is not added).

Destructor:

* Do not call virtual functions in destructors.
* Destructors must not propagate exceptions outside.
  + Do not throw in a destructor.
  + If there is a member function call that throws an exception in the destructor. Always try-catch in the destructor.
* If you do not use the compiler generated destructor (if you define it), compiler generated copy constructor, copy assignment operator, move constructor, move assignment operator, then do not use.

Overriding:

* The overriding member function follows the meaning of the function of the original member function.
* Overriding member functions add virtual and override properly.
* Add final if you do not want to override the overridden member function any more.

Overload:

* Overloaded member functions should have the same execution purpose.
* Prepare functions with different names for different purposes.
* When overloading a member function of base class with derived class, reduce the visible range of member function it must not be.
* Do not create overload functions whose arguments can be cast implicitly

Operator Overloading

* Use operator overloading sparingly.
  + Do not overload the comma (,). Because the evaluation order of the left and right sides of the operator is not constant.
  + Use implicit type conversion sparingly.

Argument:

* The number of arguments is limited to four.
* Arguments are built-in types and their type aliases, enum, pass by value.
* Pass const reference if you do not rewrite the object with the function.
* The pointer argument is used only when "the function performs processing when the argument is nullptr".
* Do not give names to arguments that must be defined even though they are not used due to reasons such as inheritance.
* Parameter name of copy constructor and copy assignment operator is rhs (right-hand side).
* The reason for passing by reference or passing by pointer is
  + Faster processing at runtime.
  + Source code dependencies can be reduced if the type of the object passed by reference is forward-declared, so compilation may be faster.
* Name the parameters of the signature to help understand the meaning of the parameters.
* Do not write anything to member functions without parameters (functions that perform linkage with C have an argument of void).
* Prohibits making an array an argument (implicit type conversion to a pointer). Use a reference to an array instead.

Return:

* Do not use handle (pointer or reference) inside class for return value. If it is unavoidable, be sure to add const.
* Prohibit using void \* for return value of function / member function other than memory allocator.
* Do not return objects other than built-in types and unique\_ptr.

Constexpr:

* Add to functions that can be added by constexpr.

Extern:

* Prohibit extern prototype declaration in .cpp.
* When using an external function, include the header file in which the function is declared.

Assertion:

* Use assert () positively to detect logically impossible states.
* Use static \_assert for logic conflicts that can be determined at compile time, not at runtime.
* If both static \_assert and assert can be used, use static \_assert in preference.

Reentrant:

* Implement functions and member functions as reentrant as possible.
* Member functions of functions or objects referenced by multiple threads must be reentrant.

Exception Handling:

* Functions should not throw exceptions unless it is unavoidable.
* Exceptions thrown by the standard library (string, vector, stream etc) do not try-catch, crash the program. There is nothing you can do with try-catch.
* Always make constructor calls succeed. Do not do things that may cause errors (such as network connections).
* Prohibit the use of "exception specifications with the throw keyword" (as it often violates LSP).
* If you specify that the function does not throw an exception, add noexcept.

int getValue () const noexcept;

* Receive by const reference if try-catch is inevitable.
* In RAII (Resource Acquisition Is Initialization) to avoid resource leak due to exception, manage resources.

Busy Loop:

* Disable busy loop (while(1) {}). Replace with event driven.

2.2.4．Syntax

Compound statement:

* Always use compound statements after if, else, for, while, and do.
* In order to express the intention that nothing will be done in an empty compound statement, place a sentence consisting only of ";".

switch statement:

* Make sure that the case clause and default clause of the switch statement end with a break statement. You should avoid using case statements without breaks, but in rare cases the code may be simple. In this case, a comment stating that there is no problem even if there is no break statement. Be sure to default is always inserted even if it is only break.

if-else-if statement:

* If it continues with if-else-if, end it with an else statement.

Control Statement Nesting:

* If a switch statement is required in if, for, while, do-while, and switch, make the inner switch statement a function.

return statement:

* Do not put parentheses after the return statement.

goto statement:

* Prohibit the use of goto.

Lambda Expression:

* The lambda expression should basically be a one-liner. Do not exceed 5 lines.

Statement in Macro:

* Use do-while (0) ideom if there is a sentence in the macro.

2.2.5．Operator

Priorities:

* In expressions where the order of precedence is unclear, enclose the order in parentheses ().

Bit operation:

* The handling of the sign when overflow or underflow is undefined, so prohibits bit operations on signed variables.
* Use std :: bitset for bit operation as much as possible, because it can be implemented regardless of the bit length of the variable.
* Use () properly so that you do not have to be careful about operator precedence.

Logical operation:

* The right operand of the && or || logical operator must not contain side effects.

Ternary Operator:

* Put parentheses so that there is no unintended interpretation.
* Use ternary operators in preference to simple if statements. In particular, conditional initialization of variables may not be possible with if statements.

New:

* new (nothrow), prohibit the use of placement new.
* Since the return value of new is never nullptr, the return value of the new operator is not tested.
* Do not new if you can create an object on the stack.
* A class that does not permit new has a private operator new ().
* The new object is held as unique\_ptr, do not make unique\_ptr new.

Delete:

* According to the rules of new, there is no explicit delete. This rule is for some reason that it is applied only when the object created by new cannot be managed by unique\_ptr.
* Do not delete at a place where the class destructor is not visible.
* Do not compare delete target pointer with nullptr before deleting.

Optional rules for :: operator new:

* Prohibit the use of global new. Create a local memory pool created in a separately defined method, and overload new for each class.

Sizeof:

* Prioritize sizeof (instance name) instead of sizeof (type name).
* For pointer type variables, to get the size of the instance it points to: Use sizeof (\* pointer variable name).

Pointer Arithmetic:

* Operations between pointers must only be applied to pointers to elements of the same array.

RTTI:

* The use of typeid (RTTI (Runtime Type Conversion)) is prohibited except for unit tests.
* Use virtual functions when you want to execute code in different execution paths depending on the type of derived class. When processing is outside the object, it can be realized by the Visitor pattern etc.

cast, implicit type conversion:

* Even if cast is unavoidable, C type cast, const \_cast, dynamic \_cast are prohibited. Even when using static \_cast or reinterpret \_cast, downcast is not permitted.
* It prohibits implicit conversion of arrays to pointers except in examples like strnlen and memcpy. If you want an array to be an argument of a function, use a reference to the array, this makes the length of the array clear even within the function.

2.2.6．Preprocessor Instructions

* + Preprocessor Instructions causes deterioration of the readability of the source code and an obstacle to unit testing.
  + Prohibit # if / # ifdef etc in .cpp.
  + Symbol generation by ## is prohibited.

Macro Function:

* Use template function instead of macro function.
* Do not use macro functions unless there is no other way to implement them.

Macro Type Constants:

* Give preference to constexpr uint32 \_t and enum, not macro constants.
* The use of macro constants is prohibited except when there is no other implementation method.

2.2.7．Header File

Implementation and Publishing:

* Software has the following structure in terms of package
  + Software is divided into packages.
  + The package is made of multiple header files and multiple .cpps.
  + Files that make up a package are stored in this package-specific directory.
  + Packages may be divided into subpackages. Subpackages, package requirements meet.

Based on the above premise, when a package provides services to the outside, it is defined in the package header file that should only do one of the following:

* + Publish the package service to the outside (header for interface publishing).
  + Defined for the implementation of class in the package, not exposed outside the package (implementation header).
* Place the interface publishing header in a directory that can be referenced by other packages at compile time.
* Implementation headers should not be placed in a directory that can be referenced by other packages at compile time.
* An identifier that can be defined in .cpp without cloning must not be defined in the header file.

Dependencies:

* Do not create unnecessary / inappropriate dependencies (DIP or ISP violations).
  + Properly use forward declarations to minimize dependencies.
  + Avoid cycle of dependencies. Circular dependencies between packages are prohibited.
  + Keep proper dependencies by using design patterns and ideom properly
    - If you want to avoid dependency propagation, use the pimpl ideom to hide the implementation details
    - Apply Observer, DI, Factory, etc. if you do not want the superordinate concept to depend on the subordinate concept.

Dual loading protection:

* Put #define guards on header files to prevent double inclusion. The macro for guard is

<path name> \ \_ <file name> \ \_ H \ \_.

#include in header file:

* If header file includes other header file, header file to reduce unnecessary dependency, do not include header files that are unnecessary for compiling.
* If you do not need to dereference the types used in header files, use forward declarations to reduce dependencies.

Order of files to be #included:

* Perform #include of system header file before #include of user-defined header file.
* System header files are #included in alphabetical order and #Include user-defined header files in alphabetical order.

Path name specified by #include:

* User-defined header files are enclosed in "", and system header files are enclosed in <>
* "../" (Move up directory) to include from header file or .cpp file. Prohibit path specification.

include other than .h:

* Do not include .c or .cpp.

2.2.8．Scope

Scope Definition:

* The scope covered in this chapter is defined as follows.

0. Global

1. Package External Publishing namespace

2. Package external private namespace

3. File (nameless namespace and static in function)

4. In class

5. In the function

6. In the block

Scope Principle:

* The scope of the identifier must be arranged to be minimal.
  + Prohibit the definition of an identifier with a scope of 0.
  + Prohibit the definition of static variables with scope of 1 and 2.
  + Minimize the definition and declaration of identifiers in the package external public header file.
  + Identifiers used only inside class are defined and declared as private or protected.
  + Put an identifier used only in one .cpp in an anonymous namespace (do not use static).
  + Variables used in only a single function are defined in that function.
  + The auto variable is defined in the innermost block that uses it.
* Prohibit identifiers in namespaces with overlapping scopes.
  + Identifiers within the same namespace must be unique within that namespace.

Limitations of using namespace:

* Use of using namespace is permitted only at the beginning of the function.
* For UT framework namespace (:: testing :: XXX etc), use namespace at the beginning of UT source code.
* Prohibit use of inline namespace.

namespace alias:

* Nested long namespaces may create aliases at file scope.

2.2.9．Runtime Efficiency

* If there is a trade-off between runtime efficiency and source code readability, prioritize source code readability.
* If you do code optimization to reduce the readability of the source code, be sure to profile and optimize only runtime bottlenecks.
* Prohibit code optimization early in development.

Selection of Prefix / Postfix Operator:

* The postfix operator says, "After copying the object and returning the copied object", the prefix operator is inefficient because it is executed. Use either a prefix operator if either is acceptable. For code consistency, the same rule applies to embedded types that do not have this overhead.

operator X, operator x = selection:

* Use operator X = instead of operator X if possible.
* The same rules apply to embedded types that do not have this overhead, for code consistency.

Return object:

* Avoid using huge objects for function return values.

Move:

* If class implements deep copy, such as securing a resource and holding it as a pointer, if it is assumed that there are many code patterns for which the copy source is a temporary object, consider the move constructor and the move assignment operator.

2.2.10．Others

Assembler:

* For assembler functions, define .asm for definition and a header file to declare its prototype.
* Assembler functions also follow the rules for function / member functions.
* Make sure that inline assembler or macro functions that contain it do not spread across the package.

Language Extensions:

* Do not use compiler-specific language extensions unless there is another way to do this.
* For object alignment,
  + Use alignas, alignof.
  + Do not use compiler-specific alignment functions (#pragma etc.).
* For the #pragma to be used repeatedly, use a combination of the Pragma operator and a macro.
  1. Coding Rules Style

2.3.1．Indentation

Indent characters:

* Tabs are not used because they may change appearance depending on the environment and may interfere with merging.

In particular, when viewing the source code with a browser (such as confirmation of pull-req), it becomes an inspection inhibiting factor.

Indenting if, for, while, do-while

* The statements attached to if, for, while, and do-while are indented one less than if, for, while, and do-while.

Block Indentation:

* The statement inside the block is lowered by one indent from the block start '{'

Indenting case, default:

* Indent the case and default with the switch.
* Decrease the indent of the sentence following case and default by one.
* The statement following case, default is written after a line break.
* When enclosing with {...} in order to block in the case, '{' puts 1 byte space character after: and immediately after it. Put '}' in the same column as case.

Eg. switch (var\_a) {

case 1: { // OK

var\_b = 1;

break;

}

case 2: // NG an indent of 'case' does not match that of 'switch'.

{ // NG '{' does not appear on a same line.

var\_b = 2;

break;

} // NG an indent of '}' does not match that of 'case'.

case 3:

var\_b = 3; // NG a same indent after 'case' statement.

break;

case 4: var\_c = 4; // NG there is a statement on a same line of 'case'.

break;

default: // OK

break;

}

2.3.2．block (braces ({}))

* The statement following '}' should have a line break immediately after '}' and start writing from the same column.

2.3.3．Function Argument Brackets

* Place parentheses of function arguments immediately after the function name.
* If the argument is on a separate line, place the closing parenthesis on a separate line.
* When the whole fits on one line, write on one line as it is.

2.3.4．Access level of class

* Do not indent the access level of class because the nesting of member functions becomes too deep.

2.3.5．Blank

After statement:

* Put a space after the statement keyword (for, while, do-while, switch, try, if, else etc) and no space after the function name.

After ',':

* Put a space after ','. However, immediately after the end of the line ',', immediately after the line feed without spaces.

Before and after unary operators, binary operators and ternary operators:

* There is no space between unary operators and operands.
* Except for [], ->, “. (Period)”, “, (comma operator)” put a space before and after binary operator and ternary operator.

Unwanted Blank Characters:

* Blank characters (spaces) are for proper use as character separators. Therefore, do not put unnecessary blank characters at the end of the line. Do not use double-byte spaces except comments.

2.3.6．Writing a ternary operator

* Write the ternary operator as follows.

condition? true-expression

: false-expression

condition? true-expression: false-expression // if you can write one liner.

The following notation is also acceptable. This notation can be used like a switch statement or else-if syntax.

case1? expression1:

case2? expression2:

case3? expression3:

...

default-expression

If the execution results of the \* operator are all lvalues, you can write:

(a <b? a: b <c? b: c) = val ();

2.3.7．pointer type \ \* and reference type &

* Place pointer type \* and reference type & immediately after the type.
* In case of pointer type \* and reference type, do not declare multiple variables by one statement.

2.3.8．Assignment

* In addition to simple assignment (=), assignment includes compound assignment (+ =,-= etc.), although multiple assignments can be described in one statement but multiple assignments are prohibited in one statement in order to reduce readability.

2.3.9．Number of lines / number of columns

Number of Function Lines:

* The number of function lines (between {from}) should be within 30 lines because it is common to push test sequences into one function exclude code for unit testing.

Number of columns in a row:

* In case of number of columns in a row, the length of the line should be within 120 columns.
* If there is a function with many arguments or a complex conditional statement, and it does not fit, write the statement over multiple lines.
* Align the abstraction of each block in the code.

2.3.10．namespace

* Generally, the interval of namespace definition is vertically long, so it is necessary to indicate that the end of namespace is always at the end.

To do this, write a comment and place it before and after namespace. Do not indent because nesting is too deep.

namespace event {

namespace {

...

} // // namespace // Remember

...

} // namespace event

* 1. Coding Rules Naming

Naming principle:

* Do not abbreviate words.
* Member function names should not start with a verb (but if you can only think of nouns, add "do" at the beginning).

Prohibited matters:

* Names beginning with \_ (underscore) are not used regardless of variables or functions.
* System Hungarian naming is prohibited as it causes unwanted changes.
* Do not give more than one name to the same thing. In particular, aliasing the same type leads to unnecessary confusion.
* Do not add numbers for enumeration.
* Do not omit vowels.
* If you want to define abbreviations across projects, define them in [abbreviation list].

File Name:

* Make the Directory Name same as the package name.
* For defining file name, each word is composed of lower case letters and concatenated with '\_'
* The file name is derived from the class name as follows.

Type Name:

* Names of type (class, struct, typedef, enum) are prefixed with uppercase letters and with each new word uppercase do not use underscores (\_).
* Give a descriptive name.

Name of Constant:

* Name of Constant write in capital letters and connect words with '\_'

Member Variable Name:

* The name of the member variable of class is the end of the general variable plus the underscore (\_)
* Do not start member variables beginning with ‘m’ or ‘\_’

Member variable name of struct:

* Member variable name of struct is same as general variables. There is no '\_' at the end.

Member Functions:

* Member functions are objects that define the behavior of objects, member function names usually begin with a verb.
* Add Async to the end of asynchronous member functions.

public member function name:

* The name of the public member function is capitalized at the beginning, and each word is capitalized and separate the words with upper and lower case letters without ‘\_’.

private, protected member function name:

* Private, protected Member function names are used in the same case as automatic variable names.

Parameter Name:

* Parameter Name is same as automatic variable name rules.

Automatic variable name:

* Automatic variable names should be easy-to-understand and concise nouns in English (lowercase alphabet, numbers, ‘\_’) of 3 letters or more.
* Names such as i and j cannot be used except for loop variables.
* Use "\_" (underscore) to separate words and all lowercase letters.
* Variable names don't have to be long. Make the content a name that can be analogized.

Include Guard:

* If # pragma once is not available, header files will need include guards. The name of the include guard macro is from the project's base directory for that file.

Generate relative pathname by replacing / with \_ and appending \_ at the end.

Namespace:

* The name of namespace is the package directory name and if the directory has hierarchy, then namespace is in phase with it.

Others:

* For the collection class, make the population plural to indicate it.
* Do not use "l" or "O" for the variable name (it is difficult to distinguish between "1" and "0").
  1. Coding Rules Comment

Comment not added information:

* At the end of the file, "End of file" that indicates the end of the file, or "wait for 1 second" for sleep (1), Comments like this are unnecessary because they are obvious when you look at them.
* Prohibit code comment out (including #if 0 etc.).
* Do not allow comments (such as annotate) held by svn or git to be included in the comment.

Style of Comment:

* The doxygen format is defined as follows.
  + Class

/ \*! @class ClassName / Component (e.g. Constructor, destructor)

\* Description

\*/

* + Functions

/ \*! @fn \_name

\* @brief

\* @ param

\* @ return

\*/

* + Enum

/ \*! @enum \_name

\* conditions for find\_files\_recursively

\* /

* + Type Alias

/ \*! @typedef \_name

\* lambda expression type of condition of find\_files\_recursively

\* /

* 1. Coding Rules Prohibited Functions
* The following is deprecated in C ++ 11 or later, so use is prohibited.

unary\_function

binary\_function

pointer\_to\_unary\_function

pointer\_to\_binary\_function

ptr\_fun

mem\_fun\_t

mem\_fun1\_t

mem\_fun\_ref\_t

mem\_fun1\_ref\_t

const\_mem\_fun\_t

const\_mem\_fun1\_t

mem\_fun

mem\_fun\_ref

bind1st

binder2nd

bind2nd

auto \_ptr:

* auto \_ptr is awkward to handle, so it's a hotbed of bugs. Use unique \_ptr.

Other Deprecated STL:

* The following is deprecated in C ++ 11 or later, so use is prohibited.

std :: bind1st, std :: bind2nd, std :: ptr\_mem\_fun, std :: ptr\_mem\_fun\_ref

std :: unary\_function, std :: binary\_function

Usage limitation of std :: vector

* Use std :: vector only if you cannot define the upper limit of the managed element. If you can define upper bounds, use std :: array or an array.

Partial prohibition of std :: string

* The std :: string subscript operator [] uses string :: at () because it does not raise an out\_of\_ragen exception.
* std :: string.data () does not guarantee null termination. Use std :: string.c\_str ()

Functions that cause buffer overruns:

* Prohibit the use of the following functions that are likely to cause bugs such as buffer overflows.

gets (), scanf (), strcpy (), strcat (), sprintf (), vsprintf (), wcscat (), wcscpy ()

Command Injection Prevention:

* Prohibit use of the following functions to prevent dependency on environment variables when executing external commands.

execl (), execlp (), execv (), execvp (), popen (), system ()

Obsolete function:

* Prohibit the use of functions that are no longer maintained.

asc \ \_time (), asctime \ \_r (), ctime (), ctime \ \_r (), fattach (), fdetach (), ftw (), getitimer (), getmsg (), getpmsg (), gets (), settimer (), gettimeofday (), ioctl () in stropts.h for stream, isascii (), isastream (), \_longjmp (), pthread \ \_getconsurrency (), pthread \ \_setconcurrency (), putmsg (), putpmsg (), rand \ \_r (), \ \_setjmp (), settimer (), setpgrp (), sighold (), sigignore (), siginterrupt (), sigpause (), sigrelse (), sigset (), strlen (), \ \_tolower (), \ \_toupper (), tempnam (), tmpnam (), toascii (), ulimit (), utime ()

LEGACY Function:

* Prohibit the use of functions that have already been used.

sigstack (), cuserid (), getopt (), getw (), ttyslot (), valloc (), ecvt (), fcvt (), gcvt (), mktemp (),

bcmp (), bcopy (), bzero (), index (), rindex (), utimes (), getwd (), brk (), sbrk (), rand ()

Functions that are not thread safe:

* The following functions are prohibited because they are not thread safe.

asctime (), ctime (), getgrgid (), getgrnam (), getlogin (), getpwuid (), getpwnam (), gmtime (), localtime (), ttyname ()

ctermid (), tmpnam () (If the argument is NULL, it becomes non-reentrant and therefore cannot be used)

Nonstandard Function etc.:

* Variable-length arrays and alloca () are prohibited because they are nonstandard.

Difficult functions:

* It is extremely difficult to handle signal () and it is difficult to operate stably.

Instead, it has the advantage of being able to solve the signal () reentrant problem and having no restrictions on the functions that can be used. Use signalfd.

* Use fstream class instead of fopen / fclose to prevent resource leaks.
* Prohibit tmpfile () because you cannot open the file exclusively. Use mkstemp () instead.

Checking Symbolic Link:

* Symbolic links should not be checked with lstat () only.

Notes on strncpy () and strncat ()

* When strncpy (dst, src, sizeof (dst)-1), if sizeof (dst) <= sizeof (src), the string of dst is not null terminated.

If there is no more data to copy, there will be a performance issue as the rest of dst is filled with NULL bytes.

Both strncpy () and strncat () cannot determine that the src string has been truncated if sizeof (dst) <sizeof (src).

TOCTOU (Time Of Check, Time Of Use) Measures:

* In the pattern to check and use, such as checking the existence of the file with access () before unlink (), a problem occurs because this operation is not performed atomically.

Memory Allocation:

* Prohibit the following usage to avoid mixing new / delete and malloc / free.

If you free the pointer of the new object, the destructor of that object is not called, It leaks resources.

malloc (), realloc (), free ()

Async Signals:

* Except when processing can only be handled by signals, such as SIGCHLD for process monitoring and interruption of blocking at accept (), prohibit using asynchronous signals.
* Signals that may be used are limited to:
  + SIGHUP: daemon control
  + SIGINT: Terminal interrupt key
  + SIGILL: Incorrect hardware instruction
  + SIGBUS: hardware fault
  + SIGFPE: Arithmetic exception
  + SIGSEGV: Incorrect memory reference
  + SIGALRM: Timeout detection
  + SIGTERM: default termination signal sent by kill
  + SIGCHLD: Process monitoring

Even when handling the above signals, using sigwait () in the signal processing dedicated thread, handle asynchronous signals synchronously.