coding\_rules\_en.txt

# 1 [Introduction] (coding\_rules\_intro.md)

# 2 [SOLID] (coding\_rules\_solid.md)

# 3 [programming rule] (coding\_rules\_practice.md)

# 4 [coding style] (coding\_rules\_style.md)

# 5 [Naming rule] (coding\_rules\_naming.md)

# 6 [Comment] (coding\_rules\_comment.md)

# 7 [Design Pattern] (coding\_rules\_design\_pattern.md)

# 8 [Prohibited Function] (coding\_rules\_prohibited\_func.md)

# 9 [Others] (coding\_rules\_etc.md)

# 6 [Comment] (coding\_rules\_comment.md)

* What should be written : You should write information that can not be read from the code or a summary of the code.
* 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

\* /

# 7 [Design Pattern] (coding\_rules\_design\_pattern.md)

* The program can be implemented more simply if it is applied in the right place. If it is applied in an inappropriate place, it will increase the complexity of the program unnecessarily
* 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 such problems in advance, but the trade-off is the execution speed Will be late.

* clone ideom

It is an ideom to avoid slicing by object copy.

* NVI (non virtual interface) ideom

Can separate interface and implementation

Avoid clone code

* new overload

Example of overloading new: [op\_new.cpp] (sample / op\_new.cpp)

template for convenient overloading of new: [op\_new.h] (sample / op\_new.h)

Code to create memory pool (local heap) simply: [mpool.h] (sample / mpool.h) [mpool\_fixed.h] (sample / mpool\_fixed.h)

* DI (dependency injection)

Suppose there is a class B that generates class A in its constructor. In this case, class B is an instance of class A

Depends on Such dependencies reduce class B availability and testability.

This is "If class A is the class that wraps the database, the database is required for class B testing.

* Singleton

This pattern facilitates the disciplined use of global objects i.e. "Singleton is close to global variables". When defining Singleton, define the following two. Static member function that returns an instance (Inst ())

Static member function that returns const instance (InstConst ())

* 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, with regard to the addition of an object's public method, the correction will be clear.

* Observer

There is an object Subject and an object set Observer, and the relationship between the two is as follows:

Observer receives subject change notification.

Subject must not depend on Observer.

Model for implementing a GUI application by MVC is Subject, and View is Observer.

* Null Object

Conditional branching is prone to bugs and leads to an increase in the number of test steps. Therefore, It is better to eliminate the conditional branching that can be done. This pattern is one of the conditional branches

if (a! = nullptr) {

a. xxx ();

}

* Factory

A pattern that uses a dedicated method (Factory method) to create an object.

Used when you do not want to directly depend on an object for classes and functions that create objects.

* Template Method

Define a method (template method) that defines the form (form, etc.) of the template, and the behavior and data for filling it

The pattern to use when separating methods.

* Proxy

If you implement a class that provides services across CPU space, threads, and processes, provide that service.

Invisible clients are not aware of the location and internal structure of the service provision class (socket, pipe, msg\_q etc.)

Pattern to prevent the occurrence of unnecessary dependencies.

* Strategy

The behavior of 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 specifies "partial behavior" (Strategy object)

A pattern that can be divided into and passed from the outside as a parameter of g ().

Use this pattern if there are many variations in the Strategy object.

* RAII

RAII stands for "Resource Acquisition Is Initialization", and secures and releases resources.

A pattern or ideom that is tied to the initialization and destruction of an object.

In particular, when memory allocation using new is performed, preventing memory leaks if you do not follow RAII

It is difficult.

In RAII of memory, std :: unique\_ptr works extremely well.

coding\_practice\_en.txt

# 3 Programming Code

It aims to provide practices for realizing safer , easier-to-read C ++ source code.

\*Types and Instances :

* Basic Types:
* Use char to hold ascii characters.
* Use size \ \_t to hold the value of sizeof and the length of the array.
* For others, use the type alias defined in cstdint.
* 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.
* Char Type:
* char is used only to hold ascii characters.
* char \ \* is only used to hold a pointer to an ascii string.
* Since char depends on the compiler implementation whether it is singed or unsigned, don't use char type for operation.
* bool type :
* Do not use bool type in expressions other than logical expressions (such as arithmetic operations and comparison operations).
* In particular, do not use bool type ++ or postfix ++ (-can not compile).
* Floating point type
* Allow use of these types only if you need float or double dynamic range.
* Do not use INF or NAN in calculations.
* Do not compare float or double instances with ==.
* Do not mix float and double in one expression.
* Use fixed point instead of floating point as much as possible.
* Pointer Type
* Do not use 0 or NULL as a pointer literal that represents a null pointer. Use nullptr.
* 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 definition outside the old enum (non scoped enum) class.
  + Use scoped enum when defining enum outside class.
  + 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 defined outside class as an 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. Instead its type uses constexpr.
* This will reveal the type of the constant.
* bit field
* Prohibit use other than to access hardware registers.
* 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 for member functions generated by the compiler.
* Prohibit inheritance. final is unnecessary.
* Union
* Prohibit use other than to access hardware registers.
* definition of union should not be included in the interface publishing header.
* type alias
* Use using instead of typedef except for type aliases shared with C.
* Alias ​​template is only used in template class.
* 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 forward declaration, from the .cpp file that defines it
  + You must include a header that contains the declaration.
* const / constexpr instances
* Understand the difference between const and constexpr attached to an instance.
* The value of const is determined at runtime initialization. Then unchanged.
* 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 can not 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).

* + String literals ("xxx") must be const.
    - Also in the iterator, if possible, const or use const \ \_iterator.

* Type Inference
* 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.
* Decltype

void g ()

{

int32\_t x = 3;

int32\_t & r = x;

auto a = r; // The type of a is int32\_t

decltype (r) b = r; // The type of b is int32\_t &

decltype (auto) c = r; // The type of c is int32\_t &

// decltype (auto) is the syntax for giving declytype the right-hand expression (C ++ 14)

...

}

* 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.
* Do not perform initialization directly dependent on the linked object, because the initialization order is undefined.

(There is no problem with doing static object dependent initialization above the same file).

* Define and initialize basic type constants whose values ​​are determined at compile time with constexpr.
* uniform initialization
* Do not use uniform initialization for basic types, references, and pointer initialization.
* Use uniform initialization to initialize arrays and structures.
* Do not use uniform initialization for non-container class initialization.
* For container classes,

Use uniform initialization to initialize each element with its own object.

Do not use uniform initialization for other initializations (such as specifying the vector length).

* Do not use uniform initialization for the return value of the function.
* class initialization with non-static member variable initializer ([0.0 non-static member variable / constant initialization] (---))
* If you use to initialize member variables,

And if its member variables are initialized with the default constructor,

You may use uniform initialization.

* classes and instances
* How to use files
* In principle, one class is one header file that declares it

It consists of one .cpp file that defines it.

* A class that supports only one header file (a.h) and one class consisting of one .cpp (a.cpp)

(A struct used for an argument, a class implementing a pimpl pattern, etc.) is declared and defined in a.h and a.cpp.

* Class Size
* Number of Lines
* Except when there is no other way around, the class declaration should be around 200 lines including comments.
* Member functions defined in class must not exceed 10 lines.
* Number of Members
* Except when there is no other way, the number of public member functions can be up to about seven.
* The number of variables holding the state of the object can be up to about four.
* Cohesion
* 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 then, use setter sparingly.

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 idiom. 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.
* Prohibit protected inheritance.
* 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.
* Even in initialization of base class by default constructor call,

Write it in the constructor's initializer list.

* Inheritance of Interface
* Interface inheritance is done in public. In that case,

[0.0 Liskoff's substitution principle (LSP)] (---) must be observed.

* Multiple Inheritance
* Do not use multiple inheritance unless it is inevitable.
* 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

* Be aware that 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 there is even one member variable that requires initialization method 1,

Initialization of all variables is performed in 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 can only be defined if they are dependent in the header file in which the class is declared.
* If the static constexpr member constant is not dependent in the header file,

It is defined and initialized in the unnamed namespace of .cpp.

(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 \ \_ etc.).
* Slicing
* Do not slice objects.
* Assignment as above (hold an instance of D as a reference or pointer to B,

Define B \ \* B :: Clone (), D \ \* D :: Clone () if you want to copy it to an instance of B).

* If you want to prohibit the above assignment, delete the copy constructor of B, operator = or

Make it private.

* Object Ownership
* The object or function that created object A has ownership over object A

(Or own the object A).

* An object or function that owns object A has a duty to release object A.
* An object without ownership of object A

Avoid holding object A's handle in member variables as much as possible

(In many cases this rule is unavoidable, such as the observer pattern).

* If object A is created and generated for object B as new,
  + - The pointer of object A is held as unique\_ptr <type compatible with decltype (A)>.

Object B retains ownership of object A.

* + - Object B is responsible for releasing object A (automatic release by unique\_ptr).

Object C which does not hold ownership of object A must not release object A.

* + - When moving the ownership of object A from object B to object C,

Use unique\_ptr and move.

* 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.
* When object C which does not have ownership of object A holds the handle of object A as a member variable,

Before the end of the lifetime of object A

Object C must be prevented from accessing object A

(eg detach member function of observer pattern).

* Do not disclose the handle of an object on the stack to the outside of the function (the handle is

It 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
* 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.

* 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.
* For static member functions that can be added with constexpr (those whose return value is determined at compile time)
* If const can be added to a member function (if the object's state is not changed), it is always added.

Always set const to getXXX (getter).

Do not add const to functions like SetPtr.

* Do not use handles (pointers and references) inside class as return values.
  + If that is unavoidable, be sure to add const to the return value handle.
  + If you must return a non-const handle, the function is not const.
* A function that returns a handle inside class performs lvalue modification.
* 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
* For member functions automatically generated by the compiler, one of the following must be selected.

| How to define the above member function |

| ----------------------------------------------------------

| Do not use | = delete |

| Use compiler generation | = default |

Other than that | Implement by yourself |

* When declaring and defining a class, refer to ClassStationery (model of class) below,

Avoid creating unnecessary compiler-generated functions.

* Constructor
* If multiple constructors are required, consider whether it can be aggregated into one constructor with default arguments.
* If it can not be consolidated into one constructor, consider using a delegated constructor to prevent duplication of processing.
* Virtual functions of derived class can not be called until object initialization is complete.

Therefore, you should not call virtual functions in the constructor.

* When defining copy constructor and move constructor,

If the class has pointer type member variables, deep copying must be done.

* For constructors with one argument other than copy constructor and move constructor,

Add explicit to prevent unintended implicit type conversion.

* If you want to use the basic class constructor as the derived class constructor as it is,

Use an inherited constructor.

* 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.
  + When calling a function that may cause an exception in the destructor, it is always in the destructor

try-catch.

* If you do not use the compiler generated destructor (if you define it),

Compiler generated copy constructor, copy assignment operator, move constructor,

Do not use move assignment operator.

* Overriding
* Understand the difference between overriding and overriding.
* The overriding member function follows the meaning of the function of the original member function.
* Overriding member functions add virtual and override properly.
* If you do not want to override the overridden member function further, Add final to that function.
* All functions out of class / member functions
* Cyclomatic Complexity
* The cyclomatic complexity is preferably 15 or less.
* Unless there is a special reason, make cyclomatic complexity 20 or less.
* Number of Lines
* About 10 lines are preferable.
* Describe on 30 lines or less, unless there is a special reason.
* Overload
* Understand the difference between overriding and overriding.
* Overloaded member functions should have the same execution purpose.
* Prepare functions with different names for different purposes.
* When overloading a member function of the base class with a derived class, do not reduce the visible range of the member function.
* 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.
  + Define type conversion to bool with explicit.
* Note the symmetry of the \* operator.
  + If you implement operator ==, also implement operator! = (as well as other examples such as <and>).
  + If you implement operator +, you should also implement operator + = (same except for +).
  + If you define an assignment operator, you must also define a copy constructor.

In that case, avoid the clone code.

* Argument
* The number of arguments is limited to four.
* Basic types and their type aliases, enum, pass by value.
* Other objects are passed by reference.
* Do not rewrite the object pointed to by that reference with that function (do not call non-const member functions)
* Pass by const reference if.
* 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.
* copy constructor, copy assignment operator, move constructor,
* The parameter name of the move 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 value
* It is prohibited to use void \ \* for return value of functions / member functions other than memory allocator.
* Do not use "function declaration syntax for postfixing return types" for unavoidable reasons.
* 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.
* Receive by const reference if try-catch is inevitable.
* \* RAII (Resource Acquisition Is Initialization) to avoid resource leak due to exception
* Busy Loop
* Disable busy loop. Replace with event driven.
* Syntax

Compound statement

* Always use compound statements after if, else, for, while, and do.
* In the empty compound statement, put only ";" to express the intention that nothing is done,

Specifies that it is an empty compound statement.

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 simpler if you fall through.

In this case, a comment stating that there is no problem even if there is no break statement

// fallthrough

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.
  + If it can be a switch statement, use switch.

Range for statement

* A for statement that searches all elements of an array or container such as std :: vector

Use the range for statement introduced from C ++ 11.

Control Statement Nesting

* Do not write switch in if, for, while, do-while, switch because it is difficult to understand the relationship with break.
* 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.
* 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.
* Logical operation
* The right operand of the && or || logical operator must not contain side effects
* Ternary Operator
* Use ternary operators in preference to simple if statements.
* In particular, conditional initialization of variables may not be possible with if statements.
* If the types in the second and third terms are incompatible (if implicit type conversion can not be performed), the expression can not be compiled.

Even in such cases, if you use cast, you may be able to use ternary operators.

Use the if statement without such a brutal use.

* Memory Allocation
* 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,

It is applied only when the object created by new can not be managed by unique\_ptr.

* Do not delete if the destructor of class is not visible (incomplete type class).
* Do not delete void \ \*.
* 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.

* + - (note)
* The target of this rule is

Less memory available and rarely rebooted

I use a 3rd party library (not STL etc) and that library is new / delete, using malloc / free

* 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).

* The operands of the sizeof operator must not contain expressions that seem to have side effects.
* At first glance, do not use sizeof to a pointer that looks like an array.
* Operation between pointers
* Divisions or comparisons between pointers must only be applied to pointers to elements of the same array.
* RTTI
* Watch for confusion with RAII.
* 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.
* Use of typeid (RTTI (Runtime Type Conversion)) is not used except for unit test.
* cast, implicit type conversion
* The code that requires a cast is a design review, as it is mostly due to design level issues.
* 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.

* Preprocessor Instructions
* It causes deterioration of the readability of the source code and an obstacle to unit testing.

Macro Function

* + Preferentially use the template function, not the macro function.
* Do not use macro functions unless there is another way to implement it.

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.
* 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, if the package provides services to the outside,

The header file defined in the package should do only 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
    - Use pimpl idiom to hide implementation details if you want to avoid dependency propagation
    - Apply Observer, DI, Factory, etc. if you do not want the superordinate concept to depend on the subordinate concept.

Dual loading protection

* Include #include guards on header files to prevent double inclusion.

The macro for guard is <path name> \ \_ <file name> \ \_ H \ \_.

#include in header file

* If the header file includes other header files, to reduce unnecessary dependencies,

Do not include header files that are unnecessary for compiling header files.

* If you do not need to dereference the types used in header files, use forward declarations to reduce dependencies.
* In the following cases, the header file can be compiled if there is a preceding declaration of class,

This makes it possible to break the dependencies between header files.

* + In the header file, only pointers and references of that class are used
  + In the header file, the class is used as parameter type or function return value

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.
* #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 <>
* To include from header file or .cpp file

Prohibit path specification using "../" (move up directory).

include other than .h

* Do not include .c or .cpp.
* 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

* + An identifier with a scope of 2 has the same scope as 1 according to the C ++ conventions,
* The identifier is declared and defined in the package external private header file, so

Not accessible from outside the package (in a decent way).

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.
    - Do not use 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),

Allow use of using namespace at the beginning of UT source code.

* Prohibit use of inline namespace.

namespace alias

* Nested long namespaces may create aliases at file scope.
* Runtime Efficiency
* If there is a trade-off between run-time efficiency and source code readability
* Basically, priority is given to readability of source code.
* For code optimization that reduces 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

It is inefficient because the prefix operator is executed. Use either a prefix operator if either is acceptable.

* The same rules apply to base types that do not have this overhead, for code consistency.
* operator X, operator x = selection
* Use operator X = instead of operator X if possible.
* The same rules apply to base types that do not have this overhead, for code consistency.
* Return object
* Objects other than built-in types, enum and unique\_ptr (large objects)

Avoid using 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.

* extern template
* It is instantiated many times, which causes expansion of ROM and expansion of build time

Use extern temaplate for the temaplate class.

* 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.

Design\_pattern.txt

# 7 Design Pattern

* 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.
* clone ideom

It is an ideom to avoid slicing by object copy.

* NVI (non virtual interface) 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.
* New overload
* In embedded software development in particular, it may be desirable to limit the use of new throughout the system.
* 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.

* 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.

* 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.
* 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).

* 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.

* 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.

* 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.

* + 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.

* 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)

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"
* 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.