NOTE FOR CPPCOREGUIDELINES

This is my note for CppCoreGuidelines. Where: F.60

TYPE SAFETY

Problem areas

- **narrowing conversions**

  Minimize their use and use `narrow` or `narrow_cast` (from the GSL) where they are necessary.
- **range errors**

  Use `span`
- **array decay**

  Use `span` (from the GSL)
- **casts**

  Minimize their use. `Templates` can help.
- **unions**

  Use `variant` (in C++17)

gsl::index / std::ptrdiff_t

Index for container and array.

Report the use of void* as a parameter or return type

Report the use of more than one bool parameters

Because these bool parameters could be enum type.

If an interface is a template, document its parameters using concepts

Warn if any non-variadic template parameter is not constrained by a concept (in its declaration or mentioned in a requires clause).

Declare a pointer that must not be null as gsl::not_null

Use gsl::zstring to describe a pointer to a zero terminated C-style string

`zstring` is a zero terminated `char` string. `czstring` is a const zero terminated `char` string. `wzstring` is a zero terminated `wchar_t` string. `cwzstring` is a const zero terminated `wchar_t` string. `u16zstring` is a zero terminated `char16_t` string. `cu16zstring` is a const zero terminated `char16_t` string. `u32zstring` is a zero terminated `char32_t` string. `cu32zstring` is a const zero terminated `char32_t` string.

Do not pass an array as a single pointer

use `std::string_view` or `span<char>` from the GSL to prevent range errors.

RESOURCE SAFETY

gsl::owner / std::unique_ptr / std::shared_ptr

Use them to mark the ownership of a pointer.

A example using gsl::owner

```
gsl::owner<X*> compute(args)    // It is now clear that ownership is
transferred
{
    gsl::owner<X*> res = new X{};
    // ...
    return res;
}
```

SHARED RESOURCE

Avoid non-const global variables

- A function should not make control-flow decisions based on the values of variables declared at namespace scope.
- A function should not write to variables declared at namespace scope.

Avoid singletons

Singletons are basically complicated global objects in disguise.

EXCEPTIONS

Use exceptions to signal a failure to perform a required task

A good rule for performance critical code is to move checking outside the critical part of the code.

ASSERT

Gsl::Expects() / gsl::Ensures() / static_assert() / assert() / std::terminate()

These are as pre/postcondition.

INTERFACE

Pimple idiom

Reason:

- For `stable library ABI` , consider the Pimpl idiom.
- `Private data members` participate in class layout and private member functions participate in overload resolution, changes to those implementation details require `recompilation` of all users of a class that uses them.
- A `non-polymorphic interface class` holding a pointer to `implementation` (Pimpl) can isolate the users of a class from changes in its implementation at the cost of an indirection.

Prefer empty abstract classes as interfaces to class hierarchies

Reason:

- Abstract classes that are `empty` (have no non-static member data) are more likely to be `stable` than base classes with state.

Keep the number of function arguments low

Reasons:

- `Missing an abstraction` . There is an abstraction missing, so that a compound value is being passed as individual elements instead of as a single object that enforces an invariant. This not only expands the parameter list, but it leads to errors because the component values are no longer protected by an enforced invariant.

- “`Violating "one function, one responsibility` . The function is trying to do more than one job and should probably be refactored.

Method:

- Grouping arguments into " `bundles` " is a general technique to reduce the number of arguments and to increase the opportunities for checking.

Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning

Reason:

- Avoid adjacent parameters that can be invoked by the same arguments in either order with different meaning.

Method:

- Define a `struct` as the parameter type and name the fields for those parameters accordingly.

A function should perform a single logical operation

Keep function `short` and `simple` .

If a function might have to be evaluated at compile time, declare it constexpr

`constexpr` does not guarantee `compile-time` evaluation; it just guarantees that the function can be evaluated at compile time for constant expression arguments if the programmer requires it or the compiler decides to do so to optimize.

If a function is very small and time-critical, declare it inline

- Specifying `inline` (explicitly, or implicitly when writing member functions inside a class definition) encourages the compiler to do a better job.
- `Constexpr` implies inline.

If your function must not throw, declare it noexcept

- If an `exception` is not supposed to be thrown, the program cannot be assumed to cope with the error and should be `terminated` as soon as possible. Declaring a function noexcept helps optimizers by reducing the number of `alternative execution paths` . It also speeds up the exit after failure.
- If you know that your application code cannot respond to an `allocation failure` , it could be appropriate to add `noexcept` even on functions that allocate.
- **Destructors, swap functions, move operations, and default constructors should never throw.**

FOR

Std::ranges::for_each

```
for_each(v, [](int x) { /* do something with the value of x */ });
```