clu

Chlorie

Sep 20, 2023

CONTENTS

Chlorie's Utilities (clu) is a library consisting of many seemingly unrelated independent components. It started out being a header-only library, but then I decided to separate some of the implementation details into source files.

The library covers a range of different use cases, from tiny utilities that the standard library does not provide (or hides as implementation details, why), to more complex data structures and algorithms that are handy to use in everyday programming.

Warning: This library is mainly created by Chlorie for personal use, so a large portion of this library is still untested and lacking documentation. Please use at your own risk.

Have fun!

CHAPTER

ONE

COMPONENTS

Most of the components are under the main include directory `clu`. Some of the core headers (like `type_traits.h`) are used by other ones, while most of the other headers aren't depended upon. Things under `clu/experimental` are highly unstable, some of them may not even compile.

A list of all the headers and their contents is presented as follows:

Header any_unique.h
: Provides a type-erasure class to store objects of unknown types.

`assertion.h`
: Defines a macro `CLU_ASSERT(expr, msg)`, which functions roughly the same as `assert` from `<cassert>`, but takes an additional message parameter.

`buffer.h`
: A buffer class template, similar to `std::span` but more suitable for raw buffer manipulation.

Header c_str_view.h
: View types for null-terminated strings.

`chrono_utils.h`
: Utilities related to `std::chrono`. Providing a centralized namespace for all the constants in `std::chrono` like `std::chrono::January`. Also some helper functions.

`concepts.h`
: Useful concepts for meta-programming.

`expected.h`
: A sum type storing a value or an error `Ok | Err`, like `Result` in Rust. A rough implementation of P0323R10: `std::expected`.

Header file.h
: Contains a function `read_all_bytes` that reads all contents of a binary file into a `std::vector`.

`fixed_string.h`
: A fixed string class that can be used as class non-type template parameters.

`flags.h`
: A wrapper for flag enumeration types.

`forest.h`
: An STL-styled hierarchical container type.

`function_ref.h`
: Non-owning type-erasure types for invocables.

`function_traits.h`
: Type traits for function types. Provides meta-functions for extracting parameter types or result type from function signatures.

Header hash.h
: Hash-related utilities. Provides `constexpr` implementations of FNV-1a and SHA1 hash functions, and a `hash_combine` function.

`indices.h`
: A random-access range for multi-dimensional indexing, so that you could use `for (const auto [i, j, k] : indices(3, 4, 5))` instead of nested `for` loops.

`integer_literals.h`
: UDLs for fixed-length integral literals, like `255_u16`.

`iterator.h`
: Adapter class template for iterators. Provides default implementations for the iterator concepts.

`manual_lifetime.h`
: Helper class for manually controlling lifetime of an object.

`meta_algorithm.h`
: Algorithms for type lists.

`meta_list.h`
: Type lists and value lists for meta-programming.

`new.h`
: Helpers for aligned heap allocation.

`oneway_task.h`
: A fire-and-forget coroutine type which starts eagerly.

`optional_ref.h`
: `std::optional` like interface for optional references.

`overload.h`
: Overload set type. Useful for `std::visit`.

`scope.h`
: Scope guards for running arbitrary code at destruction. Useful for ad-hoc RAII constructs.

`static_for.h`
: Loop over compile time indices. Can also be used as a way to force loop unrolling.

`static_vector.h`
: `std::vector` like container allocated on the stack, with a fixed capacity.

`string_utils.h`
: String-related utilities.

`type_traits.h`
: Useful type traits for meta-programming.

`piper.h`
: "Pipeable" wrappers for invocables, like those ranges adapters.

`polymorphic_value.h`
: A smart pointer type for copyable polymorphic types. A rough implementation of P0201R3: A polymorphic value-type for C++.

`polymorphic_visit.h`
: `std::visit`, but for polymorphic types.

`unique_coroutine_handle.h`
: An RAII type for `std::coroutine_handle<T>` which destroys the handle on destruction.

vector_utils.h
A `make_vector` function to remedy the fact that we cannot use `std::initializer_list` to initialize containers with move-only elements.

1.1 Header any_unique.h

Provides a type-erasure class to store objects of unknown types.

1.1.1 Examples

```cpp
#include <clu/any_unique.h>

int main()
{
    clu::any_unique a(1); // a holds an int
    a.emplace(std::in_place_type<double>, 3.14); // now a holds a double, and the previous int is destroyed
    a.reset(); // a holds nothing

    struct immovable
    {
        immovable() = default;
        immovable(immovable const&) = delete;
        immovable(immovable&&) = delete;
        immovable& operator=(immovable const&) = delete;
        immovable& operator=(immovable&&) = delete;
    };

    clu::any_unique b(std::in_place_type<immovable>); // immovable objects can also be stored, using in_place_type
}
```

class **any_unique**
A type erasure class that could store literally anything.

This class type erases objects and always store them on the heap. It is almost of no use but to provide a way to manage lifetime of objects with compile-time unknown types.

Public Functions

constexpr **any_unique**(const *any_unique*&) = delete
any_unique is not copyable.

1.2 Header `buffer.h`

namespace **clu**

Typedefs

using **mutable_buffer** = *basic_buffer*<std::byte>

Mutable buffer using `std::byte` as the element type.

using **const_buffer** = *basic_buffer*<const std::byte>

Constant buffer using `std::byte` as the element type.

Functions

template<alias_safe **T**>
constexpr void **memmove**(*T* *dst, const *T* *src, const size_t size) noexcept

Copies data between addresses, safe even when the destination overlaps the source.

Parameters

- **dst** – Destination address.
- **src** – Source address.
- **size** – The size (counting T's, not bytes) to copy.

template<buffer_safe **T**>
mutable_buffer **trivial_buffer**(*T* &value) noexcept

template<buffer_safe **T**>
const_buffer **trivial_buffer**(const *T* &value) noexcept

template<typename **T**>
void **trivial_buffer**(const *T*&&) = delete

Variables

template<typename T> concept alias_safe = same_as_any_of<T, unsigned char, char, std::byte>

Specifies that a type is safe to alias.

An alias-safe type could be used as buffer elements.

template<typename T> concept buffer_safe =trivially_copyable<T> ||(std::is_array_v<T> && trivially

Specifies that a type is safe to be seen through by a buffer.

Buffer-safe types can be read from/written into buffers.

template<typename T> concept trivial_range =std::ranges::contiguous_range<T> &&std::ranges::sized_

template<typename T> concept mutable_trivial_range = trivial_range<T> && (! std::is_const_v<std::ranges::range_value_t<T>>)

template<typename **T**>

class **basic_buffer**

#include <buffer.h> Non-owning buffer view, suitable for raw byte manipulations.

Template Parameters

T – The buffer element type, must be alias-safe.

Public Types

using **value_type** = *T*

using **mutable_type** = *basic_buffer*<std::remove_const_t<*T*>>

using **const_type** = *basic_buffer*<std::add_const_t<*T*>>

Public Functions

constexpr **basic_buffer**() noexcept = default

Creates an empty buffer.

constexpr **basic_buffer**(const *basic_buffer*&) noexcept = default

constexpr **basic_buffer**(*basic_buffer*&&) noexcept = default

constexpr *basic_buffer* &**operator=**(const *basic_buffer*&) noexcept = default

constexpr *basic_buffer* &**operator=**(*basic_buffer*&&) noexcept = default

inline constexpr **basic_buffer**(*T* *ptr, const size_t size) noexcept

Constructs a buffer from a start pointer and a size.

Parameters

- **ptr** – Start of the buffer region.
- **size** – Size of the buffer.

template<typename R> inline requires (std::is_const_v< T > &&trivial_range< R >)||(! std

template<typename = int> inline requires std::is_const_v< T > constexpr explicit (false) basic_

inline constexpr *T* ***data**() const noexcept

Gets a pointer to the buffer data.

inline constexpr size_t **size**() const noexcept

Gets the size of the buffer.

inline constexpr bool **empty**() const noexcept

Checks if the buffer is empty.

inline constexpr void **remove_prefix**(const size_t size) noexcept

inline constexpr void **remove_suffix**(const size_t size) noexcept

inline constexpr *T* &**operator[]**(const size_t index) const noexcept

inline constexpr *basic_buffer* **first**(const size_t size) const noexcept

inline constexpr *basic_buffer* **last**(const size_t size) const noexcept

inline constexpr *basic_buffer* &**operator+=**(const size_t size) noexcept

inline constexpr std::span<*T*> **as_span**() const noexcept

Converts the buffer into a `std::span` of the element type.

template<trivially_copyable **U**>
inline constexpr *U* **as**() const noexcept

template<trivially_copyable **U**>
inline constexpr *U* **consume_as**() noexcept

inline constexpr size_t **copy_to**(const *mutable_type* dest) const noexcept

inline constexpr size_t **copy_to_consume**(const *mutable_type* dest) noexcept

Private Members

T ***ptr_** = nullptr

size_t **size_** = 0

Private Static Functions

template<typename **U**>
static inline constexpr *T* ***conditional_reinterpret_cast**(*U* *ptr) noexcept

Friends

inline friend constexpr friend basic_buffer operator+ (basic_buffer buffer, const size_t size) noexcept

1.3 Header `c_str_view.h`

template<typename **Char**>

class **basic_c_str_view**

A string view type for null-terminated strings.

Template Parameters

Char – The character type

Public Functions

inline constexpr const_reference **`front`**`()` const noexcept

Get the first character.

inline constexpr const *Char* ***`data`**`()` const noexcept

Get a pointer to the C-string.

inline constexpr const *Char* ***`c_str`**`()` const noexcept

Get a pointer to the C-string.

inline constexpr size_t **`size`**`()` const noexcept

Get the length of the string, excluding the null terminator

Remark

This function is not $O(1)$. It calls `strlen`.

inline constexpr size_t **`length`**`()` const noexcept

Same as *`size()`*

inline constexpr bool **`empty`**`()` const noexcept

Check if the view is empty.

inline constexpr void **`remove_prefix`**`(`const size_t n`)` noexcept

Remove prefix with a certain length from the view.

inline constexpr void **`swap`**`(`*basic_c_str_view* &other`)` noexcept

Swap two views.

inline explicit constexpr **`operator std::basic_string_view<`*`Char`*`>`**`()` const noexcept

Convert this null-terminated view into a regular `std::basic_string_view<Char>`.

Friends

`inline friend constexpr friend void swap (basic_c_str_view &lhs, basic_c_str_view &rhs) noexcept`

Swap two views.

struct **`sentinel`**

Sentinel type.

Public Functions

inline constexpr bool **`operator==`**`(`const *Char* *ptr`)` const

Comparison with character pointer, compares equal when the character is null.

1.4 Header `file.h`

This header provides helper functions to read/write data from/to files easily.

1.4.1 Examples

The following code snippet reads a file and computes SHA-1 hash of the contents:

```cpp
#include <clu/file.h>
#include <clu/assertion.h>

int main()
{
    std::vector<int> data{ 1, 2, 3, 4 };
    clu::write_all_bytes("data.bin", data);
    const auto read = clu::read_all_bytes<int>("data.bin"); // read as vector of ints
    const auto bytes = clu::read_all_bytes("data.bin"); // default behavior is to read as vector of bytes
    CLU_ASSERT(read == data);

    clu::write_all_text("data.txt", "Hello, world!");
    const std::string text = clu::read_all_text("data.txt");
    CLU_ASSERT(text == "Hello, world!");
}
```

template<trivially_copyable **T** = std::byte, allocator_for<*T*> **Alloc** = std::allocator<*T*>>
std::vector<*T*, *Alloc*> *clu*::**read_all_bytes**(const std::filesystem::path &path, const *Alloc* &alloc = *Alloc*{})

Read contents of a binary file into a `std::vector`.

Template Parameters

- **T** – Value type of the result vector. It must be trivially copyable.
- **Alloc** – Allocator type used by the vector.

Parameters

- **path** – The path to the file to read from.
- **alloc** – The allocator for allocating the vector.

std::string *clu*::**read_all_text**(const std::filesystem::path &path)

Reads contents of a text file into a `std::string`.

Parameters
path – The path to the file to read from.

void *clu*::**write_all_bytes**(const std::filesystem::path &path, *const_buffer* bytes)

Writes bytes in a given buffer into a file. The file will be overwritten.

Parameters

- **path** – The path to the file to write into.
- **bytes** – The byte buffer.

void *clu*::**write_all_text**(const std::filesystem::path &path, std::string_view text)

Writes given text into a file. The file will be overwritten.

Parameters

- **path** – The path to the file to write into.
- **text** – The text to write.

1.5 Header hash.h

This header provides hash-related helper functions and common hash algorithms. The algorithms are `constexpr` enabled if possible.

Note: Unless specified otherwise, if the digest type of a hasher is a `std::array` of unsigned integers (words), the most significant word always comes last (little endian). The individual words are in native byte order, which may not be little endian.

1.5.1 Examples

The following code snippet reads a file and computes SHA-1 hash of the contents:

```cpp
#include <iostream>
#include <format>
#include <clu/file.h>
#include <clu/hash.h>

int main()
{
    const auto bytes = clu::read_all_bytes("my_file.txt");
    const auto hash = clu::sha1(bytes);
    std::cout << std::format("SHA-1: {:08x}{:08x}{:08x}{:08x}{:08x}\n",
        hash[0], hash[1], hash[2], hash[3], hash[4]);
}
```

We cannot use strings in `switch`-es, but utilizing the `constexpr` hash algorithms we are able to imitate that feature. Hash conflicts of cases are also detected at compile time which is neat. Just don't use this on arbitrary user inputs, where weird *(read "malicious")* hash clashes may occur.

```cpp
#include <clu/hash.h>

int main()
{
    using namespace clu::literals;     // for enabling the UDLs
    switch (clu::fnv1a("second"))      // regular function call
    {
        case "first"_fnv1a: return 1;  // or use UDLs
        case "second"_fnv1a: return 2;
        case "third"_fnv1a: return 3;
    }
}
```

CHAPTER TWO

GUIDES

Since this project uses latest features of C++ aggressively, some of the concepts used in this library might seem unfamiliar to you. This part of the documentation is intended to help you to understand these concepts.

2.1 Coroutines

The concept of coroutine is nothing new, the first occurence of which dates back to the 1950s. It is an extension on the "subroutine" (or "function") concept, supporting "suspension" (or "pause") and "resumption" of execution between routines. A coroutine can be viewed as a kind of syntactic sugar of a state machine. Below is a simple illustration of coroutine execution in pseudo-C++ code.

```cpp
Coro coroutine(int& value)
{
    $suspend(); // suspension point #1
    value = 42;
    $suspend(); // suspension point #2
    return;
}

int main()
{
    int value = 0;

    // coro runs to #1 and suspends,
    // control flow returns to main
    Coro coro(value);

    // Resume the execution of coro,
    // from where it was suspended (#1).
    // It will set value to 42 and suspend again.
    coro.$resume();

    assert(value == 42);

    // This time coro resumes from #2,
    // and runs until the end (return statement).
    coro.$resume();

    return 0;
}
```

In this example the control flow ping-pongs between the coroutine and the main function, which is a typical case for generator coroutines.

2.1.1 Generators

A generator is a lazy range of values, in which each value is generated (yielded) on demand. Many other languages have some concrete implementation of generators, for example C# functions returning `IEnumerable<T>` with `yield return` statements in the function body, or Python generators with `yield` statements.

C++20 only provides us with the most bare-bone API of coroutines, with all the gnarly details exposed for coroutine frame manipulations and but support from the library side for user-facing APIs. There is currently a new `std::generator` class template proposed for the C++23 standard, and if you don't want to wait for another who-knows-how-many years, you can use the generator coroutine implementation in this library.

This library provides `clu::generator<T>` which supports yielding values from the coroutine body using the `co_yield` keyword. The following example shows how to use it.

```cpp
#include <iostream>
#include <clu/generator.h>

// A never-ending sequence of Fibonacci numbers
clu::generator<int> fibonacci()
{
    int a = 1, b = 1;
    while (true)
    {
        co_yield a; // Yielding a value
        a = b;
        b = a + b;
    }
}

int main()
{
    // Generators are input ranges,
    // we can use them in range-based for loops
    for (const int i : fibonacci())
    {
        std::cout << i << ' ';
        if (i > 100)
            return 0;
    }
}
```

For more details, see the documentation of `clu::generator`.

Warning: There is no documentation yet…

In the generator case, the coroutine execution is managed by the caller, since each time the caller needs another value from the generator, it resumes the coroutine from the last suspension point.

2.1.2 Asynchronous Tasks

A coroutine can also put its handle into some other execution context, for example, into a thread pool, such that the coroutine could be scheduled to run in parallel with other coroutines. This is the typical case for asynchronous tasks. This is typically implemented in other languages with the async-await construct.

If there were an async network library supporting coroutines, the code below would be an example of how to use it.

```cpp
#include <awesome_coro_lib.h>

// A coroutine that performs an asynchronous task.
// C++ coroutines don't need special `async` keyword,
// any function that contains `co_await` `co_yield` or
// `co_return` will be considered a coroutine.
task<std::string> get_response(const std::string& url)
{
    http_client client;
    // Use the co_await keyword to suspend the coroutine,
    // and after the request is completed, the coroutine
    // will the resumed with the response.
    std::string response = co_await client.get_async(url);
    co_return response;
}
```

In this example, `client.get_async(url)` would return an "awaitable" object. Applying `co_await` on the awaitable object will suspend the coroutine and wait for someone to resume. For asynchronous operations like this, the suspended coroutine's resumption is typically registered as a "callback", called after the asynchronous operation completes.

The clu library provides `clu::task<T>` to support this kind of asynchronous usage.

Note that the semantics of `task` here differs from that of some other languages that supports the async-await construct, such as C#. The C# `Task<T>` type starts eagerly, in the meaning that once a function returning `Task<T>` is called, the task is already running and runs parallel with the caller. This kind of detached execution means that we can no longer pass references to local variables as parameters to eager tasks, since the caller might never await on the callee task and thus might return sooner than the callee.

```cpp
eager_task<void> callee(const std::string& str)
{
    co_await thread_pool.schedule_after(30s);
    // Now str is dangling...
    std::cout << str << '\n'; // Boom!
}

eager_task<void> caller()
{
    std::string message = "Hello world!";
    callee(message);

    // Now we exit the coroutine, destroying
    // the message string, leaving a dangling
    // reference to the callee, since callee
    // is still waiting for the 30s delay.
    co_return;
}
```

We don't want to pay the cost of copying the parameters to every coroutine we call, nor do we want to reference count everything. Thus we should practice the idiom of "structured-concurrency", make sure that the callee is finished before the caller is done with it. `clu::task` is a lazy task type, which means that the callee is not started until the caller awaits on it.

```cpp
clu::task<void> callee(const std::string& str)
{
    co_await thread_pool.schedule_after(30s);
    std::cout << str << '\n';
}

clu::task<void> caller()
{
    std::string message = "Hello world!";

    // Since clu::task is lazy, the following does nothing.
    // callee(message);

    co_await callee(message);
    // Now we can safely destroy the message string.
}
```

2.2 Senders and Receivers

INDEX