CLUE++ Documentation

Release 0.2.6

Dahua Lin

Sep 06, 2017

Contents

CLUE++ is a light-weight extensions to the C++ standard library. It provides a collection of components that are widely useful in practical development. These components include a subset of functions and classes introduced in C++14 or later technical specifications (but are also useful for C++11), and some new facilities that we feel are useful in practice.

All components provided by this library are implemented in a way that closely follow the convention of the standard library. In particular, for those components that are *backported* from C++14 or new technical specifications, we strictly follow the standard specification whenever possible (with in the capability of C++11).

CHAPTER 1

Contents:

Basic utilites

Optional (Nullable)

In the C++ Extensions for Library Fundamentals (N4480), a class template `optional` is introduced, which represents objects that may possibly contain a value. Such types are widely provided by modern programming languages (*e.g.* `Nullable` in *C#*, `Maybe` in *Haskell*, `Optional` in *Swift*, and `Option` in *Rust*), and have shown their important utility in practice. This library *"backports"* the `optional` type to C++11 (within the namespace `clue`).

Here is a simple example that illustrates the use of the `optional` class.

```
#include <cmath>
#include <clue/optional.hpp>

using namespace clue;

inline optional<double> safe_sqrt(double x) {
    return x >= 0.0 ?
        make_optional(std::sqrt(x)) :
        optional<double>();
}

auto u = safe_sqrt(-1.0);  // -> optional<double>()
(bool)u;           // -> false
u.value();         // throws an exception
u.value_or(0.0);   // -> 0.0

auto v = safe_sqrt(4.0);  // -> optional<double>(2.0)
(bool)v;           // -> true
v.value();         // -> 2.0
v.value_or(0.0);   // -> 2.0
```

The standard documentation of the `optional` type is available here. Below is a brief description of this type.

Types

The class template `optional` is declared as:

class `optional`

Formal

```
template <typename T>
class optional;
```

Parameters **`T`** – The type of the (possibly) contained value.

The class `optional<T>` has a member typedef `value_type` defined as `T`.

In addition, several helper types are provided:

class `in_place_t`
A tag type to indicate in-place construction of an optional object. It has a predefined instance `in_place`.

class `nullopt_t`
A tag type to indicate an optional object with uninitialized state. It is a predefined instance `nullopt`.

Constructors

An optional object can be constructd in different ways:

constexpr `optional``()`
Constructs an *empty* optional object, which does not contain a value.

constexpr `optional`(*nullopt_t*)
Constructs an *empty* optional object (equivalent to `optional<T>()`).

`optional`(**const** *optional*&)
Copy constructor, with default behavior.

`optional`(*optional*&&)
Move constructor, with default behavior.

constexpr `optional`(**const** value_type &*v*)
Construct an optional object that contains (a copy of) the input value `v`.

constexpr `optional`(value_type &&*v*)
Construct an optional object that contains the input value `v` (moved in).

constexpr `optional`(*in_place_t*, Args&&... *args*)
Construct an optional object, with the contained value constructed inplace with the initializing arguments `args`.

Modifiers

After an `optional` object is constructed, its value can be re-constructed later using `swap`, `emplace`, or the assignment operator.

void **`swap`**(*optional* &*other*)
Swap with another optional object `other`.

void **`emplace`**(Args&&... *args*)
Re-construct the contained value using the provided arguments `args`.

Observers

Note: This class provides `operator->` to allow the access of the contained vlaue in a pointer form, and `operator*` to allow the access in a dereferenced form. One must use these operators when the `optional` object actually contains a value, otherwise it is *undefined behavior*.

A safer (but slightly less efficient) way to access the contained value is to use `value` or `value_or` member functions described below.

---

explicit constexpr operator bool () const noexcept
Convert the object to a boolean value.

Returns `true` when the object contains a value, or `false` otherwise.

constexpr value_type **const** &**value** () **const**
Get a const reference to the contained value.

Throw an exception of class `bad_optional_access` when the object is empty.

value_type &**value** ()
Get a reference to the contained value.

Throw an exception of class `bad_optional_access` when the object is empty.

constexpr value_type **value_or** (U &&*v*) **const** &
Get the contained value, or a static convertion of `v` to the type `T` (when the object is empty).

value_type **value_or** (U &&*v*) &&
Get the contained value, or a static convertion of `v` to the type `T` (when the object is empty).

Non-member Functions

void **swap** (*optional*<T> &*x*, *optional*<T> &*y*)
Swap two optional objects `x` and `y`. Equivalent to `x.swap(y)`.

constexpr *optional*<R> **make_optional** (T &&*v*)
Make an optional object that encapsulates a value `v`.

Returns An optional object of class `optional<R>`, where the template parameter `R` is defined as `typename std::decay<T>::type`.

Comparison

Comparison operators `==, !=, <, >, <=, >=` are provided to compare optional objects.

Two optional objects are considered as *equal* if they meet either of the following two conditions:

- they are both empty, or
- they both contain values, and the contained values are equal.

An optional object `x` are considered as *lesss than* another optional object `y`, if either of the following conditions are met:

- `x` is empty while `y` is not.
- they both contain values, and `x.value() < y.value()`.

Note: Comparison between an optional object and a value `v` of type `T` is allowed. In such cases, `v` is treated as an optional object that contains a value `v`, and then the rules above apply.

Timing

Timing, namely to measure the run-time of a piece of code, is a common practice in development, especially in contexts where performance is critical (*e.g.* numerical computation). *CLUE++* provides timing facilities to facilitate this practice. All these facilities are in the namespace `clue`.

Representation of duration

A class `duration` is introduced to represent time durations.

class `duration`
A wrapper of `std::chrono::high_resolution_clock::duration` that exposes more user friendly interface to work with duration.

The `stop_watch` class use a `duration` object to represent the elapsed time. The `duration` class has several member functions to retrieve the duration in different units.

explicit constexpr `duration()` noexcept
Construct a zero duration.

constexpr `duration`(const value_type *&val*) **noexcept**
Construct a duration with an object of class `value_type`, namely `std::chrono::high_resolution_clock::duration`.

constexpr double **`get()` const noexcept**
`dur.get<U>()` gets the duration in unit `U`. Here, `U` should be an instantiation of the class template `std::ratio`.

The following table lists the correspondence between `U` and physical time units.

type `U`	**physical unit**
`std::ratio<1>`	seconds
`std::milli`	milliseconds
`std::micro`	microseconds
`std::nano`	nanoseconds
`std::ratio<60>`	minutes
`std::ratio<3600>`	hours

A set of convenient member functions are also provided to make this a bit easier:

constexpr double **`secs()` const noexcept**
Get the duration in seconds.

constexpr double **`msecs()` const noexcept**
Get the duration in milliseconds.

constexpr double **`usecs()` const noexcept**
Get the duration in microseconds.

constexpr double **`nsecs()` const noexcept**
Get the duration in nanoseconds.

constexpr double **`mins()` const noexcept**
Get the duration in minutes.

constexpr double **hours**() **const noexcept**
Get the duration in minutes.

Stopwatch

A `stop_watch` class is introduced to measure running time.

class stop_watch
Stop watch class for measuring run-time, in wall-clock sense.

> **Note** Internally, it relies on the class `std::chrono::high_resolution_clock` introduced in C++11 for timing, and hence it is highly portable.

The class `stop_watch` has the following members:

explicit stop_watch(bool *st* = false) **noexcept**
Construct a stop watch. By default, it is not started. One can set `st` to `true` to let the stop watch starts upon construction.

void **reset**() **noexcept**
Reset the watch: stop it and clear the accumulated elapsed duration.

void **start**() **noexcept**
Start or resume the watch.

void **stop**() **noexcept**
Stop the watch and accumulates the duration of last run to the total elapsed duration.

duration **elapsed**() **const noexcept**
Get the total elapsed time.

Here is an example to illustrate the use of the `stop_watch` class.

```
#include <clue/timing.hpp>

using namespace clue;

// simple use

stop_watch sw(true);  // starts upon construction
run_my_code();
std::cout << sw.elapsed().secs() << std::endl;

// multiple laps

stop_watch sw1;
for (size_t i = 0; i < 10; ++i) {
    sw1.start();
    run_my_code();
    sw1.stop();
    std::cout << "cumulative elapsed = "
              << sw1.elapsed().secs() << std::endl;
}
```

Timing functions

We also provide convenient functions to help people time a certain function.

duration **simple_time** (F &&*f*, size_t *n*, size_t *n0* = 0)

Run the function `f()` for `n` times and return the total elapsed duration.

Parameters

- **f** – The function to be timed.
- **n** – The number of times `f` is to be executed.
- **n0** – The number of pre-running times. If `n0 > 0`, it will *pre-run* `f` for `n0` times to *warm up* the function (for certain functions, the first run or first several runs may take substantially longer time).

calibrated_timing_result **calibrated_time** (F &&*f*, double *measure_secs* = 1.0, double *calib_secs* = 1.0e-4)

Calibrated timing.

This function may spend a little bit time (around `calib_secs` seconds) to roughly measure the average running time of `f()` (*i.e.* calibaration), and then run `f()` for more times for actual measurement such that the entire duration of measurement is around `measure_secs` seconds.

Parameters

- **f** – The function to be timed.
- **measure_secs** – The time to be spent on actual measurement (in seconds).
- **calib_secs** – The time to be spent on calibration (in seconds).

Returns the timing result of class `calibrated_timing_result`.

class calibrated_timing_result

A struct to represent the result of calibrated timing, which has two fields:

- `count_runs`: the number of runs in actual timing.
- `elapsed_secs`: elapsed duration of the actual timing process, in seconds.

Examples:

```cpp
// source file: examples/ex_timing.hpp

#include <clue/timing.hpp>
#include <cstdio>
#include <cstring>

using namespace clue;

static char src[1000000];
static char dst[1000000];

void unused(char c) {}

// copy 1 million bytes
void copy1M() {
    std::memcpy(dst, src, sizeof(src));

    // ensure the copy actually happens in optimized code
    volatile char v = dst[0];
    unused(v);   // suppress unused warning
}

int main() {
    std::memset(src, 0, sizeof(src));
```

```
    auto r = calibrated_time(copy1M);

    std::printf("Result:\n");
    std::printf("    runs    = %zu\n", r.count_runs);
    std::printf("    elapsed = %.4f secs\n", r.elapsed_secs);

    double gps = r.count_runs * 1.0e-3 / r.elapsed_secs;
    std::printf("    speed   = %.4f Gbytes/sec\n", gps);

    return 0;
}
```

Value Range

It is a very common pattern in C/C++ programming to write loops that enumerate values within a certain range, such as

```
for (int i = 0; i < n; ++i) {
    // do something
}
```

In C++11, the range for-loop syntax is introduced, which allow concise expression of the looping over elements in a container. However, one has to resort to the old pattern when enumerating values. Here, we provide a class template `value_range` that wraps a range of values to a light-weight *container-like* object. Below is an example:

```
#include <clue/value_range.hpp>

using namespace clue;

size_t n = 10

// enumerate i from 0 to n-1
for (size_t i: vrange(n)) {
    // do something on i
}

double a = 2.0, b = 9.0;
// enumerate v from 2.0 to 8.0
for (auto v: vrange(a, b)) {
    // do something on i
}

std::vector<int> a{1, 2, 3, 4, 5};
std::vector<int> b{5, 6, 7, 8, 9};
std::vector<int> r

// enumrate i from 0 to a.size() - 1
for (auto i: indices(a)) {
    r.push_back(a[i] + b[i]);
}
```

Documentation of `value_range` and relevant functions are given below.

The `value_range` and `stepped_value_range` class templates

Formally, the class template `value_range` is defined as:

class `value_range`

Formal

```
template<typename T,
         typename D=typename default_difference<T>::type,
         typename Traits=value_range_traits<T, D>>
class value_range;
```

Classes to represent continuous value ranges, such as `1, 2, 3, 4, ...`.

Parameters

- **`T`** – The value type.
- **`D`** – The difference type. This can be omitted, and it will be, by default, set to `default_difference<T>::type`.
- **`Traits`** – A traits class that specifies the behavior of the value type `T`. This class has to satisfy the *EnumerableValueTraits* concept, which will be explained in the section *enumerable_value_traits*. In general, one may omit this, and it will be, by default, set to `value_type_traits<T, D>`.

class `default_difference`

`default_difference<T>` provides a member typedef that indicates the *default* difference type for `T`.

In particular, if `T` is an unsigned integer type, `default_difference<T>::type` is `std::make_signed<T>::type`. In other cases, `default_difference<T>::type` is identical to `T`.

To enumerate non-numerical types (*e.g.* dates), one should specialize `default_difference<T>` to provide a suitable difference type.

class `stepped_value_range`

Formal

```
template<typename T,
         typename S,
         typename D=typename default_difference<T>::type,
         typename Traits=value_range_traits<T, D>>
class stepped_value_range;
```

Classes to represent stepped ranges, such as `1, 3, 5, 7, ...`.

Parameters

- **`T`** – The value type.
- **`S`** – The step type.
- **`D`** – The difference type. By default, it is `default_difference_type<T>::type`.
- **`Traits`** – The trait class for `T`. By default, it is `value_type_traits<T, D>`.

Note: For `stepped_value_range<T, S>`, only unsigned integral types for `T` and `S` are supported at this point.

Member types

The class `value_range<T>` or `stepped_value_range<T, S>` contains a series of member typedefs as follows:

types	definitions
`value_type`	`T`
`difference_type`	`D`
`step_type`	`S`
`traits_type`	`Traits`
`size_type`	`std::size_t`
`pointer`	`const T*`
`const_pointer`	`const T*`
`reference`	`const T&`
`const_reference`	`const T&`
`iterator`	implementing `RandomAccessIterator`
`const_iterator`	`iterator`

Note: For `value_range<T>`, the `step_type` is the same as `size_type`.

Construction

The `value_range<T>` and `stepped_value_range<T, S>` classes have simple constructors.

constexpr value_range(**const** T *&vbegin*, **const** T *&vend*)

Parameters

- **vbegin** – The beginning value (inclusive).
- **vend** – The ending value (exclusive).

For example, `value_range(0, 3)` indicates the following sequence `0, 1, 2`.

stepped_value_range(**const** T *&vbegin*, **const** T *&vend*, **const** S *&step*)

Parameters

- **vbegin** – The beginning value (inclusive).
- **vend** – The ending value (exclusive).
- **step** – The incremental step.

For example, `stepped_value_range(0, 2, 5)` indicates the following sequence `0, 2, 4`.

Note: These classes also have a copy constructor, an assignment operator, a destructor and a `swap` member function, all with default behaviors.

Note: For stepped ranges, the **step must be positive**. Zero or negative step would result in undefined behavior. The size of a stepped range is computed as `(e - b + (s - 1)) / s`.

In addition, convenient constructing functions are provided, with which the user does not need to explictly specify the value type (which would be infered from the arguments):

constexpr *value_range*<T> **vrange**(**const** T &*u*)
Equivalent to `value_range<T>(static_cast<T>(0), u)`.

constexpr *value_range*<T> **vrange**(**const** T &*a*, **const** T &*b*)
Equivalent to `value_range<T>(a, b)`.

value_range<Siz> **indices**(**const** Container &*c*)
Returns a value range that contains indices from `0` to `c.size() - 1`. Here, the value type `Siz` is `Container::size_type`.

Properties and element access

The `value_range<T>` and `stepped_value_range<T, S>` classes provide a similar set of member functions that allow access of the basic properties and individual values in the range, as follows.

constexpr size_type **size**() **const noexcept**
Get the size of the range, *i.e.* the number of values contained in the range.

constexpr bool **empty**() **const noexcept**
Get whether the range is empty, *i.e.* contains no values.

constexpr size_type **step**() **const noexcept**
Get the step size.

> **Note** For `value_range<T>`, the step size is always `1`.

constexpr T **front**() **const noexcept**
Get the first value within the range.

constexpr T **back**() **const noexcept**
Get the last value **within** the range.

constexpr T **begin_value**() **const noexcept**
Get the first value in the range (equivalent to `front()`).

constexpr T **end_value**() **const noexcept**
Get the value that specifies the end of the value, which is the value next to `back()`.

constexpr T **operator[]**(size_type *pos*) **const**
Get the value at position `pos`, withou bounds checking.

constexpr T **at**(size_type *pos*) **const**
Get the value at position `pos`, with bounds checking.

> **Throw** an exception of class `std::out_of_range` if `pos >= size()`.

Iterators

constexpr const_iterator **cbegin**() **const**
Get a const iterator to the beginning.

constexpr const_iterator **cend**() **const**
Get a const iterator to the end.

constexpr *iterator* **begin**() **const**
Get a const iterator to the beginning, equivalent to `cbegin()`.

constexpr *iterator* **end**() **const**
Get a const iterator to the end, equivalent to `cend()`.

Note: A value range or stepped value range does not actually store the values in the range. Hence, the iterators are *proxies* that do not refer to an existing location in memory. Instead, `*iter` returns the value itself instead of a reference. In spite of this subtle difference from a typical iterator, we find that it works perfectly with most STL algorithms.

The *EnumerableValueTraits* concept

The class template `value_range` has a type parameter `Traits`, which has to satisfy the following concept.

```
// x, y are values of type T, and n is a value of type D

Traits::increment(x);       // in-place increment of x
Traits::decrement(x);       // in-place decrement of x
Traits::increment(x, n);    // in-place increment of x by n units
Traits::decrement(x, n);    // in-place decrement of x by n units

Traits::next(x);        // return the value next to x
Traits::prev(x);        // return the value that precedes x
Traits::next(x, n);     // return the value ahead of x by n units
Traits::prev(x, n);     // return the value behind x by n units

Traits::eq(x, y);       // whether x is equal to y
Traits::lt(x, y);       // whether x is less than y
Traits::le(x, y);       // whether x is less than or equal to y

Traits::difference(x, y); // the difference between x and y, i.e. x - y
```

By default, the builtin `value_range_traits<T, D>` would be used and users don't have to specify the traits explicitly. However, one can specify a different trait class to provide special behaviors.

Predicates

CLUE++ provides a series of higher-order functions for generating predicates (functors that returns `bool`), in the header `<clue/predicates.hpp>`. These predicates can be very useful in programming for expressing certain conditions.

Take a look of the following example, where we want to determine whether all elements are positive. With C++11, this can be accomplished as:

```
std::all_of(s.begin(), s.end(), [](int x){ return x > 0; });
```

This is convenient enough. However, still expressing simple conditions like positiveness using a full-fledged lambda expression remains cumbersome, especially when there are many conditions to express. *CLUE* provides a higher-order function `gt`, with which `[](int x){ return x > 0; }` above can be simplified as `gt(0)`, and consequently the code above can be rewritten as:

```
std::all_of(s.begin(), s.end(), gt(0));
```

Generic predicates

The following table lists the predicates provided by *CLUE*. Let `x` be the value to be tested by the predicates. Note that all these predicates are in the namespace `clue`.

functors	conditions
`eq(v)`	`x == v`
`ne(v)`	`x != v`
`gt(v)`	`x > v`
`ge(v)`	`x >= v`
`lt(v)`	`x < v`
`le(v)`	`x <= v`
`in(s)`	`x` is in `s`, *i.e.* `x` is equal to one of the elements of `s`
`in_range(l, r)`	`x >= l && x <= r`

Note: For `in(s)`, `s` can be a C-string. In this case, the generated predicate returns `true`, when the input character `x` equals one of the character in `s`.

CLUE also provides `and_` and `or_` to combine conditions.

and_(p1, p2, ...)
Return a predicate, which returns `true` for an argument `x` when `p1(x) && p2(x) && ...`.

Example: To express the condition like `a < x < b`, one can write `and_(gt(a), lt(b))`, or if it is a closed interval as `a <= x <= b`, then one can write `and_(ge(a), le(b))`.

or_(p1, p2, ...)
Return a predicate, which returns `true` for an argument `x` when `p1(x) || p2(x) || ...`.

Example: `or_(eq(a), eq(b), eq(c))` expresses the condition that `x` is equal to either `a`, `b`, or `c`.

Char predicates

CLUE provides several predicates for testing characters (of type `char` or `wchar_t`) within the namespace `clue::chars`, as follows. These functors can be very useful in text parsing.

functors	conditions
`chars::is_space`	`std::isspace(x)`
`chars::is_blank`	`std::isblank(x)`
`chars::is_digit`	`std::isdigit(x)`
`chars::is_xdigit`	`std::isxdigit(x)`
`chars::is_alpha`	`std::isalpha(x)`
`chars::is_alnum`	`std::isalnum(x)`
`chars::is_punct`	`std::ispunct(x)`
`chars::is_upper`	`std::isupper(x)`
`chars::is_lower`	`std::islower(x)`

Note: All these `is_space` etc are typed functors. Unlike the C-function such as `isspace`, these functors are likely to be inlined when passed to higher-level algorithms (*e.g.* `std::all_of`, `std::find`, etc). Also these functors work with both `char` and `wchar_t`. For example, `char::is_space(c)` calls `std::iswspace` internally when `c` is of type `wchar_t`.

Float predicates

CLUE also provides predicates for testing floating point numbers, within the namespace `clue::floats`.

functors	conditions
`floats::is_finite`	`std::isfinite(x)`
`floats::is_inf`	`std::isinf(x)`
`floats::is_nan`	`std::isnan(x)`

Note: These functors work with `float`, `double`, and `long double`.

(Demangled) Type Names

CLUE provides facilities to obtain (demangled) names of C++ types. All following functions are in the header `<clue/type_name.hpp>`, and they are in the namespace `clue`.

bool **`has_demangle`**`()`
Whether *CLUE* provides demangling support.

Note: At this point, demangling is supported with GCC, Clang, and ICC.

std::string **`type_name`**`()`
`typename<T>()` returns a (demangled) name of type `T`.

Note: It returns the demangled name when `has_demangle()`, otherwise it returns the name as given by `typeid(T).name()`.

std::string **`type_name`**`(x)`
Returns the (demangled) name of the type of `x`.

std::string **`demangle`**`(`**const** char *_name_`)`
Demangles the input name (the one returned by `typeid(T).name()`).

Note: When `has_demangle()` is true, namely, CLUE has demangling support, this returns the demangled name, otherwise it returns a string capturing the intput name.

Miscellaneous Utilities

CLUE also provides some utilities that are handy in programming practice. These utilities are provided by the header `<clue/misc.hpp>`.

`make_unique``(args...)`
`make_unique<T>(args...)` constructs an object of type `T` and wraps it in a unique pointer of class `std::unique_ptr<T>`.

Here, `args` are the arguments to be forwarded to the constructor.

It is equivalent to `unique_ptr<T>(new T(std::forward<Args>(args)...))`.

`pass``(args...)`
Accepts arbitrary arguments and does nothing.

The purpose of this function is mainly to trigger the execution of all the argument expressions in a variadic context.

class `temporary_buffer`

Formal

```
template<typename T>
class temporary_buffer
```

Tempoary buffer.

An object of this class invokes `std::get_temporary_buffer` on construction, and `std::return_temporary_buffer` on destruction.

Note: A temporary buffer is supposed to be used locally, and it is not copyable or movable.

Examples:

```
#include <clue/misc.hpp>

void myfun(size_t n) {
    // calls std::get_temporary_buffer to acquire a buffer
    // that can host at least n integers.
    temporary_buffer<int> buf(n);

    size_t cap = buf.capacity();  // get the size that are actually allocated
    int *p = buf.data();  // get the memory address

    // do somthing with buf ...

    // upon exit, the buffer will be returned by calling
    // std::return_temporary_buffer
}
```

Containers and Views

Array View

An array view is a light-weight *container-like* wrapper of a pointer and a size. It provides a convenient way to turn a contiguous memory region into a container-like object. In practice, such an object maintains the efficiency of a raw pointer while providing richer API to work with memory regions. Below is an example to illustrate this.

```
#include <clue/array_view.hpp>

using namespace clue;

int a[] = {1, 2, 3, 4, 5};

for (int& v: aview(a, 5)) {
    std::cout << v << std::endl;
    v += 1;
}
```

In practice, it is not uncommon that you maintain a vector in your object and would like to expose the elements to the users (without allowing the users to refer to the vector directly). In such cases, it would be a good idea to return an array view.

```
using namespace clue;

class A {
public:
    // ...

    array_view<const T> elements() const {
        return aview(elems_.data(), elems_.size());
    }

private:
    std::vector<T> elems_;
};

// client code

A a( /* ... */ );

std::cout << "# elems = " << a.elements().size() << std::endl;
for (const T& v: a.elements()) {
    std::cout << v << std::endl;
}
```

The `array_view` class template

class `array_view`

Formal

```
template<typename T>
class array_view;
```

Parameters **`T`** – The element type.

Note: In general, `array_view<T>` allows modification of the elements, *e.g* `a[i] = x`. To provide a readonly view, one can use `array_view<const T>`.

Member types

The class `array_view<T>` contains a series of member typedefs as follows:

types	definitions
value_type	std::remove_cv<T>::type
size_type	std::size_t
difference_type	std::ptrdiff_t
pointer	T*
const_pointer	const T*
reference	T&
const_reference	const T&
iterator	implementing RandomAccessIterator
const_iterator	iterator
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

Construction

constexpr **array_view**() **noexcept**
Construct an empty view, with null data pointer.

constexpr **array_view**(*pointer* *data*, size_type *len*) **noexcept**
Construct an array view, with data pointer data and size len.

Note: It also has a copy constructor, an assignment operator, a destructor and a swap member function, all with default behaviors. It is worth noting that the copy construction/assignment of a view is *shallow*, meaning that only the pointer and the size value are copied, the underlying content remains there.

A convenient function aview is provided for constructing array views without the need of explicitly articulating the value type.

constexpr *array_view*<T> **aview**(T *p, size_t *n*) **noexcept**
Construct an array view, with data pointer p and size n.

> **Note** If p is of type T*, it returns a view of class array_view<T>, and if p is a const pointer of type const T*, it returns a view of class array_view<const T>, which is a read-only view.

Basic properties and element access

constexpr size_type **size**() **const noexcept**
Get the size of the range, *i.e.* the number of elements referred to by the view.

constexpr bool **empty**() **const noexcept**
Get whether the view is empty, *i.e.* refers to no elements.

constexpr const_pointer **data**() **const noexcept**
Get a const pointer to the base address.

pointer **data**() **noexcept**
Get a pointer to the base address.

constexpr const_reference **front**() **const**
Get a const reference to the first element within the view.

reference **front**()
Get a reference to the first element within the view.

constexpr const_reference **back**() **const**
Get a const reference to the last element within the view.

reference **back**()
Get a reference to the last element within the view.

constexpr const_reference **operator[]**(size_type *pos*) **const**
Get a const reference to the element at position `pos`, without bounds checking.

reference **operator[]**(size_type *pos*)
Get a reference to the element at position `pos`, without bounds checking.

constexpr const_reference **at**(size_type *pos*) **const**
Get a const reference to the element at position `pos`, with bounds checking.

Throw an exception of class `std::out_of_range` if `pos >= size()`.

reference **at**(size_type *pos*)
Get a reference to the element at position `pos`, with bounds checking.

Throw an exception of class `std::out_of_range` if `pos >= size()`.

Iterators

constexpr const_iterator **cbegin**() **const**
Get a const iterator to the beginning.

constexpr const_iterator **cend**() **const**
Get a const iterator to the end.

constexpr const_iterator **begin**() **const**
Get a const iterator to the beginning, equivalent to `cbegin()`.

constexpr const_iterator **end**() **const**
Get a const iterator to the end, equivalent to `cend()`.

iterator **begin**()
Get an iterator to the beginning.

iterator **end**()
Get an iterator to the end.

constexpr const_iterator **crbegin**() **const**
Get a const reverse iterator to the reversed beginning.

constexpr const_iterator **crend**() **const**
Get a const reverse iterator to the reversed end.

constexpr *iterator* **rbegin**() **const**
Get a const reverse iterator to the reversed beginning, equivalent to `crbegin()`.

constexpr *iterator* **rend**() **const**
Get a const reverse iterator to the reversed end, equivalent to `crend()`.

iterator **rbegin**()
Get a reverse iterator to the reversed beginning.

iterator **rend**()
Get a reverse iterator to the reversed end.

Reindexed View

In practice, people often want to work on a selected subset of elements of a sequence. A typical approach is to copy those elements to another container, as

```
std::vector<int> source{1, 2, 3, 4, 5, 6};
std::vector<size_t> selected_inds{5, 1, 4};

std::vector<int> selected;
selected.reserve(selected_inds.size());
for (size_t i: selected_inds) {
    selected.push_back(source[i]);
}
```

This approach is cumbersome and inefficient. *CLUE++* introduces a class template `reindexed_view` to tackle this problem. An reindexed view is an object that *refers to* the selected elements while providing container-like API to work with them. Below is an example:

```
#include <clue/reindexed_view.hpp>

using namespace clue;

std::vector<int> source{1, 2, 3, 4, 5, 6};
std::vector<size_t> selected_inds{5, 1, 4};

for (auto v: reindexed(source, selected_inds)) {
    // do something on v
}
```

Below is the documentation of this class template and relevant functions.

The `reindexed_view` class template

class `reindexed_view`

Formal

```
template<class Container, class Indices>
class reindexed_view;
```

Parameters

- **`Container`** – The type of the element container.
- **`Indices`** – The type of the indices container.

Both `Container` and `Indices` need to be random access containers.

Note: Here, the `Container` type can be a const type, *e.g.* `const vector<int>`. Using a constant container type as the template argument would lead to a read-only view, which are often very useful.

Member types

The class `reindexed_view<Container, Indices>` contains a series of member typedefs as follows:

types	definitions
container_type	std::remove_cv<Container>::type
indices_type	std::remove_cv<Indices>::type
value_type	container_type::value_type
size_type	indices_type::size_type
difference_type	indices_type::difference_type
const_reference	container_type::const_reference
const_pointer	container_type::const_pointer
const_iterator	container_type::const_iterator

There are also other member typedefs, whose definitions depend on the *constness* of `Container`.

type `reference`
Defined as `container_type::const_reference` when `Container` is a const type, or `container_type::reference` otherwise.

type `pointer`
Defined as `container_type::const_pointer` when `Container` is a const type, or `container_type::pointer` otherwise.

type `iterator`
Defined as `container_type::const_iterator` when `Container` is a const type, or `container_type::iterator` otherwise.

Construction

constexpr `reindexed_view`(Container &*container*, Indices &*indices*) **noexcept**
Construct a reindexed view, with the given source container and index sequence.

Note: A reindexed view only maintains references to `container` and `indices`. It is the caller's responsibility to ensure that the `container` and `indices` remain valid while using the view. Otherwise, undefined behaviors may result.

A convenient function `reindexed` is provided for creating reindexed views, without requiring the user to explicitly specify the container type and the indices type:

constexpr *reindexed_view*<Container, Indices> **`reindexed`**(Container &*c*, Indices &*inds*)
Construct a reindexed view, with the given source container and index sequence, where the types `Container` and `Indices` are deduced from arguments.

> **Note** If `c` is a const reference, then `Container` will be deduced to a const type. The same also applies to `indices`.

Basic properties and element access

constexpr bool **`empty`**() **const noexcept**
Get whether the view is empty (*i.e.* contains no selected elements). It is equal to `indices.empty()`.

constexpr size_type **`size`**() **const noexcept**
Get the number of *selected* elements. It is equal to `indices.size()`.

constexpr size_type **`max_size`**() **const noexcept**
Get the maximum number of elements that a view can possibly refer to.

constexpr const_reference **`front`**() **const**
Get a const reference to the first element within the view.

reference **front**()
Get a reference to the first element within the view.

constexpr const_reference **back**() **const**
Get a const reference to the last element within the view.

reference **back**()
Get a reference to the last element within the view.

constexpr const_reference **operator[]**(size_type *pos*) **const**
Get a const reference to the element at position `pos`, without bounds checking.

reference **operator[]**(size_type *pos*)
Get a reference to the element at position `pos`, without bounds checking.

constexpr const_reference **at**(size_type *pos*) **const**
Get a const reference to the element at position `pos`, with bounds checking.

reference **at**(size_type *pos*)
Get a reference to the element at position `pos`, with bounds checking.

Iterators

constexpr const_iterator **cbegin**() **const**
Get a const iterator to the beginning.

constexpr const_iterator **cend**() **const**
Get a const iterator to the end.

constexpr const_iterator **begin**() **const**
Get a const iterator to the beginning, equivalent to `cbegin()`.

constexpr const_iterator **end**() **const**
Get a const iterator to the end, equivalent to `cend()`.

iterator **begin**()
Get an iterator to the beginning.

iterator **end**()
Get an iterator to the end.

Fast Vector

Sequential containers (e.g. `std::vector`) are very widely used in engineering practice, and therefore its efficiency can have a notable impact to a system's overall performance. *CLUE++* provides an optimized implementation, namely `clue::fast_vector`, can aims to serve as a drop-in replacement of `std::vector` in performance-critical paths.

Features

Conforming inteface

From the standpoint of interface, `clue::fast_vector` implements all API for `std::vector` as specified in the C++11 standard. One may refer to the documentation of std::vector for detailed information. Hence, it can be directly used as a replacement of `std::vector`.

```
class A {
    // ...
};

using myvec_t = std::vector<A>;
//
// to leverage fast vector, one can simply
// rewrite this definition as
//
//  using myvec_t = clue::fast_vector<A, 4>;
//
// suppose most vectors have a length below 4.
//

// ...

myvec_t a;
a.xxx(); // call certain member functions
```

Optimized implementation

Compared to `std::vector`, the implementation of `clue::fast_vector` is optimized in several aspects:

- Allows users to specify a customized *static capacity* `SCap`, when the number of elements is below `SCap`, they can be stored in a static array directly embedded in the object (without the need of dynamic allocation). This can substantially speed up the cases that involve a large number of short vectors (but with varying sizes).
- For element types that are declared as *relocatable*, it directly calls `memcpy` or `memmove` when performing batch insertion or erasion.
- It grows the capacity by a factor of about `1.625 = 1 + 1/2 + 1/8` instead of `2`. The choice of this a smaller growth factor is inspired by fbvector.

The `fast_vector` class template

class `fast_vector`

Formal

```
template<class T,
         size_t SCap=0,
         bool Reloc=is_relocatable<T>::value,
         class Allocator=std::allocator<T> >
class fast_vector final;
```

Parameters

- **`T`** – The element type.
- **`SCap`** – The static capacity.
- **`Reloc`** – Whether the elements are bitwise relocatable.
- **`Allocator`** – The underlying allocator type.

Note:

- The internal implementation of `fast_vector` optionally comes with a static array of size `SCap`. When the number of elements is below `SCap`, they can be stored in the static array without dynamic memory allocation. By default, `SCap == 0`, which indicates using dynamic memory whenever the vector is non-empty.
- **Bitwise relocatability** means that an instance of type `T` can be moved around in the memory without affecting its own integrity. This is the case for most C++ types used in practice. However, for certain types that maintain pointers or references to members, their instances are not relocatable.

  CLUE uses a traits struct `clue::is_relocatable` to determine whether a type is relocatable. For safety, *CLUE* adopts a conservative approach, that is, to assume all types are NOT relocatable except scalar types. However, users can overwrite this behavior to enable fast movement for a customized type `T`, either specializing `clue::is_relocatable<T>` or simply specifying the third template argument `Reloc` to be `true`.

Ordered Dict

An *ordered dict* is an associative container (like a hash table) that preserves the input order of the entries, namely, the order of pairs that one visits when traversing from `begin()` to `end()` is the same as the order of those pairs being inserted to the dict.

Note: The order-preserving behavior is similar to that of Python's OrderedDict.

```
#include <clue/ordered_dict.hpp>

using namespace clue;

ordered_dict<string, int> d;
d["a"] = 1;
d["b"] = 3;
d["c"] = 2;

for (const auto& e: d) {
    std::cout << e.first << " -- " << e.second << std::endl;
}

// This snippet prints:
//
// a -- 1
// b -- 2
// c -- 3

// The dict can also be constructed in other ways,
// e.g. initializer list.

ordered_dict<string, int> d2{{"a", 1}, {"b", 3}, {"c", 2}};

// key/values can be accessed via several methods:

d.at("b");   // -> 3
d.at("x");   // throws std::out_of_range

d.find("b");  // returns a iterator pointing to {"b", 2}
d.find("x");  // returns d.end()
```

```
// entries can be added in several different ways:

ordered_dict<string, int> d3;
d3.insert({"a", 1});
d3.emplace("b", 2);       // construct a pair then
                          // decide whether to insert
d3.try_emplace("c", 3);   // when "c" is not found,
                          // it then construct a pair and insert.

// Note: the subtle differences between the behaviors of emplace
// and try_emplace follows that of std::unordered_map.

d3.emplace("a", 5);      // no insertion happens as "a" already existed.
d3.update("a", 5);       // updates the value of d3["a"] to 5
d3["a"] = 5;             // updates the value of d3["a"] to 5

d.insert({{"a", 10}, {"b", 20}}); // insert a series of pairs,
                                  // entries with repeated keys will be
                                  // ignored.

d.update({{"a", 10}, {"b", 20}}); // update from a series of pairs,
                                  // entries with repeated keys will be
                                  // used to overwrite current values.
```

The `ordered_dict` class template

class `ordered_dict`

Formal

```
template<class Key,
         class T,
         class Hash = std::hash<Key>,
         class KeyEqual = std::equal_to<Key>,
         class Allocator = std::allocator< std::pair<Key,T> >
        >
class ordered_dict;
```

Parameters

- **`Key`** – The Key type (copy-constructible).
- **`T`** – The mapped type.
- **`Hash`** – The hash functor type.
- **`KeyEqual`** – The functor type for key equality comparison.
- **`Allocator`** – The allocator type.

Note: The implementation of `ordered_dict` contains a vector of key-value pairs (of class `std::pair<Key, T>`), and a map from key to index.

The API design of `ordered_dict` emulates that of `std::unordered_map`, except that it is a grow-only container, namely, one can insert new entries but cannot remove existing ones.

Member types

The class `ordered_dict<Key, T, Hash, KeyEqual, Allocator>` contains a series of member typedefs as follows:

types	definitions
`key_type`	`Key`
`mapped_type`	`T`
`value_type`	`std::pair<Key, T>`
`size_type`	`std::size_t`
`difference_type`	`std::ptrdiff_t`
`hasher`	`Hash`
`key_equal`	`KeyEqual`
`allocator_type`	`Allocator`
`reference`	`T&`
`const_reference`	`const T&`
`pointer`	`std::allocator_traits<Allocator>::pointer`
`const_pointer`	`std::allocator_traits<Allocator>::const_pointer`
`iterator`	`std::vector<std::pair<Key, T>, Allocator>::iterator`
`const_iterator`	`std::vector<std::pair<Key, T>, Allocator>::const_iterator`

Construction

`ordered_dict``()`
Default constructor. Constructs an empty dict.

`ordered_dict`(InputIter *first*, InputIter *last*)
Constructs a dict from a range of key-value pairs, given by `[first, last)`.

`ordered_dict`(std::initializer_list<value_type> *ilist*)
Constructs a dict from an initializer_list that contains a series of key-value pairs.

Note: `ordered_dict` also has a copy constructor, an assignment operator, a destructor and a `swap` member function, all with default behaviors.

Basic Properties

bool **`empty`**`()` **const noexcept**
Get whether the dict is empty (i.e. containing no entries).

size_type **`size`**`()` **const noexcept**
Get the number of key-value entries contained in the dict.

size_type **`max_size`**`()` **const noexcept**
Get the maximum number of entries that can be put into the dict.

bool **`operator==`**(**const** *ordered_dict* &*other*) **const**
Test whether two dicts are equal, *i.e.* their underlying list of key-value pairs are equal.

bool **`operator!=`**(**const** *ordered_dict* &*other*) **const**
Test whether two dicts are not equal.

Lookup

The elements in a dict can be retrieved by a key or a positional index.

const T &**at**(**const** Key &*key*) **const**
Get a const reference to the corresponding mapped value given a key.

Throw An exception of class `std::out_of_range` when the given key is not in the dict.

T &**at**(**const** Key &*key*)
Get a reference to the corresponding mapped value given a key.

Throw An exception of class `std::out_of_range` when the given key is not in the dict.

const value_type &**at_pos**(size_type *pos*) **const**
Get a const reference to the `pos`-th key-value pair.

value_type &**at_pos**(size_type *pos*)
Get a reference to the `pos`-th key-value pair.

T &**operator[]**(**const** Key &*key*)
Return a reference to the mapped value corresponding to `key`. When the `key` is not in the dict, it inserts a new entry (where the key is copied, and the mapped value is constructed by default constructor).

Note This is equivalent to `try_emplace(key).first->second`.

T &**operator[]**(Key &&*key*)
Return a reference to the mapped value corresponding to `key`. When the `key` is not in the dict, it inserts a new entry (where the key is moved in, and the mapped value is constructed by default constructor).

Note This is equivalent to `try_emplace(std::move(key)).first->second`.

const_iterator **find**(**const** Key &*key*) **const**
Locate a key-value pair whose key is equal to `key`, and return a const iterator pointing to it. If `key` is not found, it returns `end()`.

iterator **find**(**const** Key &*key*)
Locate a key-value pair whose key is equal to `key`, and return an iterator pointing to it. If `key` is not found, it returns `end()`.

size_type **count**(**const** Key &*key*) **const**
Count the number of occurrences of those keys that equal `key`.

Modification

void **clear**()
Clear all contained entries.

void **reserve**(size_type *c*)
Reserve the internal storage to accomodate at least `c` entries.

std::pair<*iterator*, bool> **emplace**(Args&&... *args*)
Construct a new key-value pair from `args` and insert it to the dict if the key does not exist.

Returns a pair comprised of an iterator to the inserted/found entry, and whether the insertion occurs.

std::pair<*iterator*, bool> **try_emplace**(**const** key_type &*k*, Args&&... *args*)
If the given key `k` is not found in the dict, insert a new key-value pair whose mapped value is constructed from `args`, otherwise, no construction and insertion would happen.

Returns a pair comprised of an iterator to the inserted/found entry, and whether the insertion occurs.

Note: There exist differences between the behaviors of `emplace` and `try_emplace`. Specifically, `emplace` first constructs a key-value pair from `args`, and then look-up the key and decide whether to insert the new pair; while `try_emplace` first look-up the key and then decide whether to construct and insert a new pair. Generally, `try_emplace` is more efficient when the key already existed.

std::pair<*iterator*, bool> **insert** (**const** value_type &*v*)
Insert a copy of the given pair to the dict if the key `v.first` is not found.

Returns a pair comprised of an iterator to the inserted/found entry, and whether the insertion occurs.

std::pair<*iterator*, bool> **insert** (value_type &&*v*)
Insert a move-in pair to the dict if the key `v.first` is not found.

Returns a pair comprised of an iterator to the inserted/found entry, and whether the insertion occurs.

std::pair<*iterator*, bool> **insert** (P &&*v*)
Equivalent to `emplace(std::forward<P>(v))`.

void **insert** (InputIter *first*, InputIter *last*)
Insert a range of key-value pairs to the dict.

Note Those pairs whose keys already exist will not be inserted.

void **insert** (std::initializer_list<value_type> *ilist*)
Insert a series of key-value pairs from a given initializer list `ilist`.

Note Those pairs whose keys already exist will not be inserted.

void **update** (**const** value_type &*v*)
Update an entry based on the given key-value pair. Insert a new entry if the key `v.first` is not found.

Note `d.update(v)` is equivalent to `d[v.first] = v.second`.

void **update** (InputIter *first*, InputIter *last*)
Update entries from a range of key-value pairs.

void **update** (std::initializer_list<value_type> *ilist*)
Update entries from a series of key-value pairs given by an initializer list `ilist`.

Iterators

constexpr const_iterator **cbegin** () **const**
Get a const iterator to the beginning.

constexpr const_iterator **cend** () **const**
Get a const iterator to the end.

constexpr const_iterator **begin** () **const**
Get a const iterator to the beginning, equivalent to `cbegin()`.

constexpr const_iterator **end** () **const**
Get a const iterator to the end, equivalent to `cend()`.

iterator **begin** ()
Get an iterator to the beginning.

iterator **end** ()
Get an iterator to the end.

Note: These iterators are pointing to key-value pairs, of type `std::pair<Key, T>`.

Keyed Vector

A *keyed vector* is a sequential container that allows constant-time random access (similar to `std::vector`), where the elements can be accessed by positional indexes or associated keys (*e.g.* the names).

```
#include <clue/keyed_vector.hpp>

using namespace clue;

keyed_vector<int, string> v;

v.push_back("a", 10);
v.push_back("b", 20);
v.push_back("c", 30);

// The code above makes a keyed-vector that contains three
// elements: 10, 20, and 30. They are respectively associated
// with keys "a", "b", and "c".

// The elements can be accessed using positional indexes:

v[0];     // -> 10, without bounds-checking
v.at(1);  // -> 20, with bounds-checking
v.at(2);  // -> 30
std::vector<int>(v.begin(), v.end()); // -> {10, 20, 30}

// They can also be accessed using keys:

v.by("a");   // -> 10
v.by("b");   // -> 20
v.by("c");   // -> 30

// Elements can be appended in batch:

keyed_vector<int, string> v2;
v2.extend(v.begin(), v.end());
v2.extend({{"a", 10}, {"b", 20}, {"c", 30}});

// The vector can be directly constructed from ranges/initializers

keyed_vector<int, string> u(v.begin(), v.end());
keyed_vector<int, string> u2{{"a", 10}, {"b", 20}, {"c", 30}};
```

The `keyed_vector` class template

class `keyed_vector`

Formal

```
template<class T,
         class Key,
         class Hash=std::hash<Key>,
```

```
        class Allocator=std::allocator<T>
    >
class keyed_vector;
```

Parameters

- **T** – The element type.
- **Key** – The key type.
- **Hash** – The hashing functor of keys.
- **Allocator** – The allocator type.

Note: The implementation of `keyed_vector` contains a standard vector of type `std::vector<T, Allocator>` and a hash map that associates keys with positional indexes.

The API design of this class emulates that of `std::vector`, except: (1) it allows elements to be accessed by key, using the method `by`, and (2) it is grow-only, namely, one can add new elements, but cannot remove existing ones.

Difference from `unordered_dict`

Whereas both `unordered_dict` and `keyed_vector` implement key-value mapping and preserve input order, they are different kinds of containers. Specifically, `unordered_dict` is a container of `std::pair<Key, T>` with an API similar to `std::unordered_map`, while `keyed_vector` is a container of `T` with an API similar to `std::vector` (while additionally allowing indexing by key).

Member types

The class `keyed_vector<T, Key, Hash, Allocator>` contains the following member typedefs:

types	definitions
key_type	Key
value_type	T
size_type	std::size_t
difference_type	std::ptrdiff_t
hasher	Hash
allocator_type	Allocator
reference	T&
const_reference	const T&
pointer	std::allocator_traits<Allocator>::pointer
const_pointer	std::allocator_traits<Allocator>::const_pointer
iterator	std::vector<T, Allocator>::iterator
const_iterator	std::vector<T, Allocator>::const_iterator
reverse_iterator	std::vector<T, Allocator>::reverse_iterator
const_reverse_iterator	std::vector<T, Allocator>::const_reverse_iterator

Construction

keyed_vector()
Construct an empty keyed vector.

keyed_vector(InputIter *first*, InputIter *last*)
Construct a keyed vector from a range of entries (of type `std::pair<Key,T>`).

keyed_vector(std::initializer_list<std::pair<Key, T>> *ilist*)
Construct a keyed vector from a list of initial entries (of type `std::pair<Key, T>`).

Note: `keyed_vector` also has a copy constructor, an assignment operator, a destructor and a `swap` member function, all with default behaviors.

Basic Properties

bool **empty**() **const noexcept**
Get whether the vector is empty (i.e. containing no elements).

size_type **size**() **const noexcept**
Get the number of elements contained in the vector.

size_type **max_size**() **const noexcept**
Get the maximum number of elements that can be put into the vector.

size_type **capacity**() **const noexcept**
The maximum number of elements that the current storage can hold without reallocating memory.

bool **operator==**(**const** *keyed_vector* &*other*) **const**
Test whether two keyed vectors are equal, *i.e.* the sequence of elements and their keys are equal.

bool **operator!=**(**const** *keyed_vector* &*other*) **const**
Test whether two keyed vectors are not equal.

Element Access

const T ***data**() **const noexcept**
Get a const pointer to the base of the internal element array.

T ***data**() **noexcept**
Get a pointer to the base of the internal element array.

const T &**front**() **const**
Get a const reference to the first element.

T &**front**()
Get a reference to the first element.

const T &**back**() **const**
Get a const reference to the last element.

T &**back**()
Get a reference to the last element.

const T &**at**(size_type *i*) **const**
Get a const reference to the `i`-th element.

> **Throw** an exception of class `std::out_of_range` if `i >= size()`.

T &**at**(size_type *i*)
Get a reference to the `i`-th element.

> **Throw** an exception of class `std::out_of_range` if `i >= size()`.

const T &**operator[]**(size_type *i*) **const**
Get a const reference to the `i`-th element (without bounds checking).

T &**operator[]**(size_type *i*)
Get a reference to the `i`-th element (without bounds checking).

const T &**by**(**const** key_type &*k*) **const**
Get a const reference to the element corresponding to the key `k`.

Throw an exception of class `std::out_of_range` if the key `k` is not found.

T &**by**(**const** key_type &*k*)
Get a reference to the element corresponding to the key `k`.

Throw an exception of class `std::out_of_range` if the key `k` is not found.

const_iterator **find**(**const** key_type &*k*) **const**
Return a const iterator pointing to the element corresponding the key `k`, or `end()` if `k` is not found.

iterator **find**(**const** key_type &*k*)
Return an iterator pointing to the element corresponding the key `k`, or `end()` if `k` is not found.

Modification

void **clear**()
Clear all contained elements.

void **reserve**(size_type *c*)
Reserve the internal storage to accomodate at least `c` elements.

void **push_back**(**const** key_type &*k*, **const** value_type &*v*)
Push a new element `v` with key `k` to the back of the vector.

Both `k` and `v` will be copied.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **push_back**(**const** key_type &*k*, value_type &&*v*)
Push a new element `v` with key `k` to the back of the vector.

Here, `k` will be copied, while `v` will be moved in.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **push_back**(key_type &&*k*, **const** value_type &*v*)
Push a new element `v` with key `k` to the back of the vector.

Here, `k` will be moved in, while `v` will be copied.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **push_back**(key_type &&*k*, value_type &&*v*)
Push a new element `v` with key `k` to the back of the vector.

Both `k` and `v` will be moved in.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **emplace_back**(**const** key_type &*k*, Args&&... *args*)
Construct a new element at the back of the vector with arguments `args`.

Here, the associated key `k` will be copied.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **emplace_back**(key_type &&*k*, Args&&... *args*)
Construct a new element at the back of the vector with arguments `args`.

Here, the associated key `k` will be moved in.

Throw an exception of class `std::invalid_argument` if `k` already existed.

void **extend**(InputIter *first*, InputIter *last*)
Append a series of keyed values to the back. An element of the source range should be a pair of class `std::pair<Key, T>`.

Throw an exception of class `std::invalid_argument` when attempting to add a value with a key that already existed.

void **extend**(std::initializer_list<std::pair<Key, T>> *ilist*)
Append a series of keyed values (from an initializer list) to the back.

Throw an exception of class `std::invalid_argument` when attempting to add a value with a key that already existed.

String and text processing

String View

In the C++ Extensions for Library Fundamentals (N4480), a class template `basic_string_view` is introduced. Each instance of such a class refers to a constant contiguous sequence of characters (or *char-like objects*). This class provides a light-weight representation (with only a pointer and a size) of a *sub-string* that implements many of the methods available for `std::string`.

The string views are very useful in practice, especially for those applications that heavily rely on sub-string operations (but don't need to modify the string content). For such applications, string views can be a drop-in replacement of standard strings (*i.e.* instances of `std::string`) as they provide a similar set of interface, but are generally much more efficient (they don't make copies).

This library provides string view classes, where our implementation strictly follows the Technical Specification (N4480), except that all the classes and functions are within the namespace `clue` (instead of `std::experimental`). The standard document for this class is available here.

Below is brief description of the types, their members, and other relevant functions.

The `basic_string_view` class template

The signature of the class template is as follows:

class `basic_string_view`

Formal

```
template<class charT, class Traits>
class basic_string_view;
```

Parameters

- **`charT`** – The character type.
- **`Traits`** – The traits class that specify basic operations on the character type. Here, `Traits` can be omitted, which is, by default, set to `std::char_traits<charT>`.

Four typedefs are defined:

typedef *basic_string_view*<char> **string_view**

typedef *basic_string_view*<wchar_t> **wstring_view**

typedef *basic_string_view*<char16_t> **u16string_view**

typedef *basic_string_view*<char32_t> **u32string_view**

For ASCII strings with character type `char`, one should use `string_view`.

Member types and constants

The class `basic_string_view<charT, Traits>` contains a series of member typedefs as follows:

types	definitions
`traits_type`	`Traits`
`value_type`	`charT`
`pointer`	`const charT*`
`const_pointer`	`const charT*`
`reference`	`const charT&`
`const_reference`	`const charT&`
`iterator`	implementing `RandomAccessIterator`
`const_iterator`	`iterator`
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`
`size_type`	`std::size_t`
`difference_type`	`std::difference_type`

It also has a member constant `npos`, defined as `size_t(-1)`, to indicate a certain kind of characters or sub-strings are not found in a finding process. (This is the same as `std::string`).

Constructors

The class `basic_string_view<charT, Traits>` provides multiple ways to construct a string view. Below is a brief documentation of the member functions. For conciseness, we take `string_view` for example in the following documentation. The same set of constructors and member functions apply to other instantations of the class template similarly.

constexpr string_view() noexcept
Construct an empty string view

constexpr string_view(const *string_view* &*r*) **noexcept**
Copy construct a string view from `r` (default behavior)

> **Note** The copy constructor only sets the size and the base pointer, without copying the characters that it refers to.

string_view(const std::string &*s*) **noexcept**
Construct a view of a standard string `s`.

constexpr string_view(const charT **s*, size_type *count*) **noexcept**
Construct a view with the base address *s* and length *count*.

constexpr string_view(const charT **s*) **noexcept**
Construct a view of a null-terminated C-string.

The `string_view` class also has destructor and assignment operators, with default behaviors.

Basic Properties

The `string_view` class provides member functions to get basic properties:

constexpr bool **`empty`**`()` **const noexcept**
Get whether the string view is empty (*i.e.* with zero length).

constexpr size_type **`length`**`()` **const noexcept**
Get the length (*i.e.* the number of characters).

constexpr size_type **`size`**`()` **const noexcept**
Get the length (the same as `length()`).

constexpr size_type **`max_size`**`()` **const noexcept**
Get the maximum number of characters that a string view can possibly refer to.

Element Access

constexpr const_reference **`operator[]`**`(`size_type *pos*`)` **const**
Get a const reference to the character at location `pos`.

> **Note** The member function `operator []` does not perform bound checking.

const_reference **`at`**`(`size_type *pos*`)` **const**
Get a const reference to the character at location `pos`, with bounds checking.

> **Throw** an exception of class `std:out_of_range` if `pos >= size()`.

constexpr const_reference **`front`**`()` **const**
Get a const reference to the first character in the view.

constexpr const_reference **`back`**`()` **const**
Get a const reference to the last character in the view.

constexpr const_pointer **`data`**`()` **const noexcept**
Get a const pointer to the base address (*i.e.* to the first character).

> **Note** For views constructed with default constructor, this returns a null pointer.

Iterators

constexpr const_iterator **`cbegin`**`()` **const noexcept**
Get a const iterator to the beginning.

constexpr const_iterator **`cend`**`()` **const noexcept**
Get a const iterator to the end.

constexpr *iterator* **`begin`**`()` **const noexcept**
Get a const iterator to the beginning, equivalent to `cbegin()`.

constexpr *iterator* **`end`**`()` **const noexcept**
Get a const iterator to the end, equivalent to `cend()`.

constexpr const_iterator **`crbegin`**`()` **const noexcept**
Get a const reverse iterator to the reversed beginning.

constexpr const_iterator **`crend`**`()` **const noexcept**
Get a const reverse iterator to the reversed end.

constexpr *iterator* **`rbegin`**`()` **const noexcept**
Get a const reverse iterator to the reversed beginning, equivalent to `crbegin()`.

constexpr *iterator* **rend**() **const noexcept**
Get a const reverse iterator to the reversed end, equivalent to `crend()`.

Modifiers

void **clear**() **noexcept**
Clear the view, resetting the data pointer and the size to `nullptr` and `0` respectively.

void **remove_prefix**(size_type *n*) **noexcept**
Exclude the first `n` characters from the view.

void **remove_suffix**(size_type *n*) **noexcept**
Exclude the last `n` characters from the view.

void **swap**(*string_view* &*other*) **noexcept**
Swap the view with `other`.

Note: An external `swap` function are provided for string views, which invokes the member function `basic_string_view::swap` to perform the swapping.

Conversion, Copy, and Sub-string

explicit operator std::string() **const**
Convert the string view to a standard string (by making a copy).

std::string **to_string**() **const**
Convert the string view to a standard string (by making a copy).

size_type **copy**(charT **s*, size_type *n*, size_type *pos* = 0) **const**
Copy the part starting at `pos` to a buffer `s` of length `n`.

Returns The number of characters actually copied, which is equal to `min(n, size() - pos)`.

constexpr *string_view* **substr**(size_type *pos* = 0, size_type *n* = npos) **const**
Get a view of a sub-string (with length bounded by `n`) that begins at `pos`.

Returns With `pos < size()`, it returns a view of a sub-string, whose length is equal to `min(n, size() - pos)`.

Throw an exception of class `std::out_of_range` if `pos >= size()`.

Comparison

int **compare**(*string_view sv*) **const noexcept**
Compare with another string view `sv`.

Returns `0` when it is equal to `sv`, a negative integer when it is less than `sv` (in lexicographical order), or a positive integer when it is greater than `sv`.

int **compare**(size_type *pos1*, size_type *n1*, *string_view sv*) **const**
Equivalent to `substr(pos1, n1).compare(sv)`.

int **compare**(size_type *pos1*, size_type *n1*, *string_view sv*, size_type *pos2*, size_type *n2*) **const**
Equivalent to `substr(pos1, n1).compare(sv.substr(pos2, n2))`.

int **compare**(**const** charT **s*) **const**
Compare with a null-terminated C-string `s`.

int **compare**(size_type *pos1*, size_type *n1*, **const** charT **s*) **const**
Equivalent to `substr(pos1, n1).compare(s)`.

int **compare**(size_type *pos1*, size_type *n1*, **const** charT **s*, size_type *n2*) **const**
Equivalent to `substr(pos1, n1).compare(string_view(s, n2))`.

Note: These many `compare` methods may seem redundant. They are there mainly to be consistent with the interface of `std::string`.

In addition to the `compare` methods, all comparison operators (including `==, !=, <, >, <=, >=`) are provided for comparing string views. These operators return values of type `bool`.

Find Characters

Similar to `std::string`, string view classes provide a series of member functions to locate characters or sub-strings. These member functions return the index of the found occurrence or `string_view::npos` when the specified character or sub-string is not found within the view (or part of the view).

size_type **find**(charT *c*, size_type *pos* = 0) **const noexcept**
Find the first occurrence of a character `c`, starting from `pos`.

size_type **rfind**(charT *c*, size_type *pos* = npos) **const noexcept**
Find the last occurrence of a character `c`, in a reverse order, starting from `pos`, or the end of the string view, if `pos >= size()`.

size_type **find_first_of**(charT *c*, size_type *pos* = 0) **const noexcept**
Find the first occurrence of a character `c`, starting from `pos` (same as `find(c, pos)`).

size_type **find_first_of**(*string_view* *s*, size_type *pos* = 0) **const noexcept**
Find the first occurrence of a character that is in `s`, starting from `pos`.

size_type **find_first_of**(**const** charT **s*, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `find_first_of(string_view(s, n), pos)`.

size_type **find_first_of**(**const** charT **s*, size_type *pos* = 0) **const noexcept**
Equivalent to `find_first_of(string_view(s), pos)`.

size_type **find_last_of**(charT *c*, size_type *pos* = npos) **const noexcept**
Find the last occurrence of a character `c`, in a reverse order, starting from `pos`, or the end of the string view, if `pos >= size()` (same as `rfind(c, pos)`).

size_type **find_last_of**(*string_view* *s*, size_type *pos* = npos) **const noexcept**
Find the last occurrence of a character that is in `s`, in a reverse order, starting from `pos` (or the end of the string view, if `pos >= size()`).

size_type **find_last_of**(**const** charT **s*, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `find_last_of(string_view(s, n), pos)`.

size_type **find_last_of**(**const** charT **s*, size_type *pos* = npos) **const noexcept**
Equivalent to `find_last_of(string_view(s), pos)`.

size_type **find_first_not_of**(charT *c*, size_type *pos* = 0) **const noexcept**
Find the first occurrence of a character that is not `c`, starting from `pos`.

size_type **find_first_not_of**(*string_view* *s*, size_type *pos* = 0) **const noexcept**
Find the first occurrence of a character that is not in `s`, starting from `pos`.

size_type **find_first_not_of**(**const** charT **s*, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `find_first_not_of(string_view(s, n), pos)`.

size_type **find_first_not_of**(**const** charT *s, size_type *pos* = 0) **const noexcept**
Equivalent to `find_first_not_of(string_view(s), pos)`.

size_type **find_last_not_of**(charT *c*, size_type *pos* = npos) **const noexcept**
Find the last occurrence of a character that is not `c`, in a reverse order, starting from `pos`.

size_type **find_last_not_of**(*string_view s*, size_type *pos* = npos) **const noexcept**
Find the first occurrence of a character that is not in `s`, in a reverse order, starting from `pos`.

size_type **find_last_not_of**(**const** charT *s, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `find_first_not_of(string_view(s, n), pos)`.

size_type **find_last_not_of**(**const** charT *s, size_type *pos* = npos) **const noexcept**
Equivalent to `find_first_not_of(string_view(s), pos)`.

Find Substrings

size_type **find**(*string_view s*, size_type *pos* = 0) **const noexcept**
Find a substring `s`, starting from `pos`.

size_type **find**(**const** charT *s, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `find(substr(s, n), pos)`.

size_type **find**(**const** charT *s, size_type *pos* = 0) **const noexcept**
Equivalent to `find(substr(s), pos)`.

size_type **rfind**(*string_view s*, size_type *pos* = npos) **const noexcept**
Find a substring `s`, in a reverse order, starting from `pos`, or the end of the string view if `pos >= size()`.

> **Note** A matched substring is considered as *found* if its starting position precedes `pos`.

size_type **rfind**(**const** charT *s, size_type *pos*, size_type *n*) **const noexcept**
Equivalent to `rfind(substr(s, n), pos)`.

size_type **rfind**(**const** charT *s, size_type *pos* = npos) **const noexcept**
Equivalent to `rfind(substr(s), pos)`.

Note: The reason that there are so many `find_*` methods in slightly different forms is that string views need to be consistent with `std::string` in the interface, so it can serve as a drop-in replacement.

Extensions of String Functionalities

This library provides a set of functions to complement the methods of `std::string` (or `string_view`). These functions are useful in many practical applications.

Note: To be consistent with the standard, these extended functionalities are provided as global functions (within the namespace `clue`) instead of member functions.

Combining string views together with these functionalities would make string analysis much easier. Before going into details, let's first look at a practical examples.

Suppose, we have a text file like this:

```
# This is a list of attribues
# The symbol `#` is to indicate comments

bar = 100, 20, 3
foo = 13, 568, 24
xyz = 75, 62, 39, 18
```

The following code snippet uses the functionalities provided in *CLUE++* to parse them into a list of records.

```
// a simple record class
struct Record {
    std::string name;
    std::vector<int> nums;

    Record(const std::string& name) : name(name) {}

    void add(int v) {
        nums.push_back(v);
    }
};

inline std::ostream& operator << (std::ostream& os, const Record& r) {
    os << r.name << ": ";
    for (int v: r.nums) os << v << ' ';
    return os;
}

// the following code reads the file and parses the content
using namespace clue;

// open a file
std::istringstream fin(filename)

// get first line
char buf[256];
fin.getline(buf, 256);

while (fin) {
    // construct a string view out of buffer,
    // and trim leading and trailing spaces
    auto sv = trim(string_view(buf));

    // process each line
    // ignoring empty lines or comments
    if (!sv.empty() && !starts_with(sv, '#')) {
        // locate '='
        size_t ieq = sv.find('=');

        // note: sub-string of a string view remains a view
        // no copying is done here
        auto name = trim(sv.substr(0, ieq));
        auto rhs = trim(sv.substr(ieq + 1));

        // construct a record
        Record record(name.to_string());

        // parse the each term of right-hand-side
        // by tokenizing
```

```
        foreach_token_of(rhs, ", ", [&](const char *p, size_t n){
            int v = 0;
            if (try_parse(string_view(p, n), v)) {
                record.add(v);
            } else {
                throw std::runtime_error("Invalid integer number.");
            }
            return true;
        });

        // print the record
        std::cout << record << std::endl;
    }

    // get next line
    fin.getline(buf, 256);
}
```

In this code snippet, we utilize five aspects of functionalities in *CLUE++*:

- `string_view`, which constructs a like-weight view (without making a copy) on a memory block to provide string-related API. For example, you can do `sv.find(c)` and `sv.substr(...)`. Particularly, `sv.substr(...)` results in another string view of the sub-part, without making any copies.
- `trim`, which yields another string view, with leading and trailing spaces excluded.
- `starts_with`, which checks whether a string starts with a certain character of sub-string. *CLUE++* also provides `ends_with` to check the suffix, and `prefix`/`suffix` to extract the prefixes or suffixes.
- `foreach_token_of`, which performs tokenization in a functional way. In particular, it allows a callback function/functor to process each token, instead of making string copies of all the tokens.
- `try_parse`, which trys to parse a string into a numeric value, and returns whether the parsing succeeded.

For string views, please refer to *String View* for detailed exposition. Below, we introduce other string-related functionalities provided by *CLUE++*.

Make string view

constexpr `view`(s)
Make a view of a standard string `s`.

If `s` is of class `std::basic_string<charT, Traits, Allocator>`, then the returned object will be of class `basic_string_view<charT, Traits>`. In particular, if `s` is of class `std::string`, the returned type would be `string_view`.

Prefix and suffix

constexpr `prefix`(s, size_t *n*)
Get a prefix (*i.e.* a substring that starts at `0`), whose length is at most `n`.

Parameters

- **`s`** – The input string `s`, which can be a standard string or a string view.
- **`n`** – The maximum length of the prefix.

This is equivalent to `s.substr(0, min(s.size(), n))`.

constexpr `suffix` (s, size_t *n*)
Get a suffix (*i.e.* a substring that ends at the end of `s`), whose length is at most `n`.

Parameters

- **`s`** – The input string `s`, which can be a standard string or a string view.
- **`n`** – The maximum length of the suffix.

This is equivalent to `s.substr(k, m)` with `m = min(s.size(), n)` and `k = s.size() - m`.

bool **`starts_with`** (str, sub)
Test whether a string `str` starts with a prefix `sub`.

Here, `str` and `sub` can be either a null-terminated C-string, a string view, or a standard string.

bool **`ends_with`** (str, sub)
Test whether a string `str` ends with a suffix `sub`.

Here, `str` and `sub` can be either a null-terminated C-string, a string view, or a standard string.

Trim strings

`trim` (str)
Trim both the leading and trailing spaces of `str`, where `str` can be either a standard string or a string view.

Returns the trimmed sub-string. It is a view when `str` is a string view, or a copy of the sub-string when `str` is an instance of a standard string.

`trim_left` (str)
Trim the leading spaces of `str`, where `str` can be either a standard string or a string view.

Returns the trimmed sub-string. It is a view when `str` is a string view, or a copy of the sub-string when `str` is an instance of a standard string.

`trim_right` (str)
Trim the trailing spaces of `str`, where `str` can be either a standard string or a string view.

Returns the trimmed sub-string. It is a view when `str` is a string view, or a copy of the sub-string when `str` is an instance of a standard string.

Parse values

bool **`try_parse`** (str, T &*v*)
Try to parse a given string `str` into a value `v`. It returns whether the parsing succeeded.

Parameters

- **`str`** – The input string to be parsed, which can be either a C-string, a string view, or a standard string.
- **`v`** – The output variable, which will be updated upon successful parsing.

To be more specific, if the function succeeded in parsing the number (*i.e.* the given string is a valid number representation for type `T`), the parsed value will be written to `v` and it returns `true`, otherwise, it returns `false` (the value of `v` won't be altered upon failure).

Note Internally, this function may call `strtol`, `strtoll`, `strtof`, or `strtod`, depending on the type `T`.

Note: This function allows preceding and trailing spaces in `str` (for convenience in practice), meaning that `"123"`, `"123 "`, and `" 123\n"`, etc are all considered valid when parsing an integer. However, empty strings, strings with spaces in the middle (*e.g.* `123 456`), or strings with undesirable characters (*e.g.* `123a`) are considered invalid.

For integers, the function allows base-specific prefixes. For example, `"0x1ab"` are considered an integer in the hexadecimal form, while `"0123"` are considered an integer in the octal form.

For floating point numbers, both fixed decimal notation and scientific notation are supported.

For boolean values, the function can recognize the following patterns: `"0"` and `"1"`, `"t"` and `"f"`, as well as `"true"` and `"false"`. Here, the comparison with these patterns are case-insensitive.

Examples:

```
using namespace clue;

int x;
try_parse("123", x);   // x <- 123, returns true
try_parse("a123", x);  // returns false (x is not updated)

double y;
try_parse("12.75", y);  // y <- 12.75, returns true

bool z;
try_parse("0", z);      // z <- false, returns true
try_parse("false", z);  // z <- false, returns true
try_parse("T", z);      // z <- true, returns true

// in real codes, you may write this in case you don't really know
// exactly what value type to expect

auto s = get_some_string_from_text();
bool x_bool;
int x_int;
double x_real;

if (try_parse(s, x_bool)) {
    std::cout << "got a boolean value: " << x_bool << std::endl;
} else if (try_parse(s, x_int)) {
    std::cout << "got an integer: " << x_int << std::endl;
} else if (try_parse(s, x_real)) {
    std::cout << "got a real number: " << x_real << std::endl;
} else {
    throw std::runtime_error("Can't recognize the value!");
}
```

Tokenize

Extracting tokens from a string is a basic and important task in many text processing applications. ANSI C provides a `strtok` function for tokenizing, which, however, will destruct the source string. Some tokenizing functions in other libraries may return a vector of strings. This way involves making copies of all extracted tokens, which is often unnecessary.

In this library, we provide tokenizing functions in a new form that takes advantage of the lambda functions introduced in C++11. This new way is both efficient and user friendly. Here is an example:

```
using namespace clue;

const char *str = "123, 456, 789, 2468";

std::vector<long> values;
foreach_token_of(str, ", ", [&](const char *p, size_t len){
    // directly convert the token to an integer,
    // without making a copy of the token
    values.push_back(std::strtol(p, nullptr, 10));

    // always continue to take in next token
    // if return false, the tokenizing process will stop
    return true;
});
```

Formally, the function signature is given as below.

void **foreach_token_of** (str, delimiters, f)

Extract tokens from the string str, with given delimiter, and apply f to each token.

Parameters

- **str** –

  The input string, which can be either of the following type:

  - C-string (*e.g.* `const char*`)
  - Standard string (*e.g.* `std:string`)
  - String view (*e.g.* `string_view`)

- **delimiters** – The delimiters for separating tokens, which can be either a character or a C-string (if a character `c` matches any char in the given `delimiters`, then `c` is considered as a delimiter).
- **f** – The call back function for processing tokens. Here, `f` should be a function, a lambda function, or a functor that takes in two inputs (the base address of the token and its length), and returns a boolean value that indicates whether to continue.

This function stops when all tokens have been extracted and processed *or* when the callback function `f` returns `false`.

Formatting

CLUE provides several convenience functions to facilitate string formatting. These functions are light-weight wrappers based on `snprintf` and C++'s `stringstream`. These functions are provided by the header `<clue/sformat.hpp>`.

sstr (args...)

Concatenating multiple arguments into a string, through a string stream (an object of class `std::ostringstream`).

Note: The arguments here need not be strings. The only requirement is that they can be inserted to a standard output stream.

Examples:

```
#include <clue/sformat.hpp>

sstr(12);  // -> "12"
sstr(1, 2, 3); // -> "123"
sstr(1, " + ", 2, " = ", 3); // -> "1 + 2 = 3"

struct MyPair {
    int x;
    int y;
};

inline std::ostream& operator << (std::ostream& out, const MyPair& p) {
    out << '(' << p.x << ", " << p.y << ')';
    return out;
}

sstr("a = ", MyPair{1,2}); // -> "a = (1, 2)"
```

cfmt (fmt, x)

Wraps a numeric value `x` into a light-weight wrapper of class `cfmt_t<T>`. This wrapper uses `snprintf`-formatting with pattern `fmt`, when inserted to a standard output stream.

Examples:

```
cout << cfmt("%d", 12);   // cout << "12"
cout << cfmt("%.6f", 2);  // cout << "12.000000"
```

cfmt_s (fmt, args...)

Encapsulate the result of `snprintf`-formatting into a function. This function accepts multiple arguments. It returns an object of class `std::string`.

Examples:

```
cfmt_s("%04d", 12);  // -> "0012"
cfmt_s("%d + %d = %d", 1, 2, 3); // -> "1 + 2 = 3"
```

delimits (seq, delimiter)

Wraps a sequence `seq` into a light-weight wrapper of class `Delimits<Seq>`. The elements of the sequence will be outputed with a separator `delimiter`, when the wrapper is inserted to a standard output stream.

Note: Here, `seq` can be of arbitrary collection type `Seq`. The only requirement is that `Seq` provides the `begin()` and `end()` methods.

Examples:

```
std::vector xs{1, 2, 3};
cout << delimits(xs, "+");  // cout << "1+2+3"

std::vector ys{5};
cout << delimits(ys, ",");  // cout << "5"

sstr('[', delimits(xs, ", "), ']');  // -> "[1, 2, 3]"
```

String Template

CLUE provides a light-weight string template engine, by the class `stemplate`.

This template engine uses `{{ name }}` to indicate the terms to be interpolated, and accepts a dictionary-like object for interpolation.

```
clue::stemplate st("{{a}} + {{b}} = {{c}}");

std::unordered_map<std::string, int> dct;
dct["a"] = 123;
dct["b"] = 456;
dct["c"] = 579;

std::cout << st.with(dct);  // directly write to the output stream
st.with(dct).str();         // return a rendered string
```

Note: Here, `st.with(dct)` returns a light-weight wrapper that maintains const references to both the template `st` and the value dictionary `dct`. When inserted to an output stream, the result is directly written to the output stream. One may also call the `str()` member function of the wrapper, which would return the rendered string, an object of class `std::string`.

Text IO

The library provides some convenient utilties to read/write text files. These functionalities are implemented in `<clue/textio.hpp>`.

std::string **`read_file_content`**(filename)
: Read all the content in a file into a string.

  Here, `filename` can be of type `const char*` or `std::string`.

class `line_stream`
: Line stream class. It wraps a text string into a stream of lines. So one can iterate the lines using STL-style iterators.

  The iterated values are of type `clue::string_view` that provides a view into the part of the text corresponding to the current line. **Note:** The string view includes the line-delimiter `'\n'`.

  The class has three constructors, respectively accepting a C-string toegther with a length, a C-string, or a standard C++ string.

Example: The following example reads text from a file, and print its lines with line number prefixes.

```
#include <clue/textio.hpp>
#include <iostream>

using clue::string_view;
using clue::read_file_content;
using clue::line_stream;

int main() {
    std::string text = read_file_content("myfile.txt");

    line_stream lstr(text);
    size_t line_no = 0;
    for (string_view line: lstr) {
```

```
        std::cout << ++line_no << ": " << line;
    }
}
```

Monadic Parser

We provide the class `mparser` to facilitate the implementation of parsers. Specifically, this class implements a light-weight monadic parser combination framework, that allows one to compose basic parsing rules to parse more complex patterns.

Examples

Here are two examples on how practical parsers can be implemented with `mparser`.

The first example is to parse a given string `ex` in the form of `<name> = <value>`, where `name` is a typical identifier and `value` is an integer. Here, spaces are dealt with using the `skip_spaces` method.

```
// the namespace that hosts a number of basic parsing rules
using namespace mpar;

// the variables to store the views of extracted parts
string_view lhs, rhs;

// construct an mparser, and skip leading spaces
mparser mp(ex);
mp = mp.skip_spaces();

// extract left-hand-side
mp = mp.pop() >> identifier() >> pop_to(lhs);

// ensure the existence of '='
// blanks(0) means at least 0 blank characters
mp = mp >> blanks(0) >> ch('=') >> blanks(0);

// extract right-hand-side
mp = mp.pop() >> digits() >> pop_to(rhs);
```

The second example is to parse a function call, in the form of `fun(arg1, arg2, ...)`, where the arguments can be variable names, or real numbers.

```
using namespace mpar;

// the variables to store the views of extracted parts
string_view fname, arg;
std::vector<string_view> args;

// construct an mparser and skip leading spaces
auto mp = mparser(ex).skip_spaces();

// extract function name
mp = mp.pop() >> identifier() >> pop_to(fname);
assert(!mp.failed());

// locate the left bracket
mp = mp >> blanks(0) >> ch('(') >> blanks(0);
```

```
assert(!mp.failed());

// loop to extract arguments
auto term = either_of(identifier(), realnum());
mp = foreach_term(mp, term, ch(','), [&](string_view e){
    args.push_back(e);
});
assert(!mp.failed() && mp.next_is(')'));
```

Note: It is noteworthy that parsing is not a pure procedure. When parts are detected/extracted, they need to be processed by other components of a larger program, *e.g.* being fed to higher-level analysis or translated to other forms. Compared to heavy frameworks, such as ANTLR and Boost Spirit, our light-weight approach is more efficient and easier to embed into a C++ program.

The `basic_mparser` class template

The signature of the class template is:

class `basic_mparser`

Formal

```
template<typename CharT>
class basic_mparser;
```

Parameters **`CharT`** – The character type, *e.g.* `char` or `wchar_t`.

Two alias types are defined:

typedef *basic_mparser*<char> **`mparser`**

typedef *basic_mparser*<wchar_t> **`wmparser`**

Within the class, there are several useful public typedefs:

types	definitions
`value_type`	`CharT`
`iterator`	`const CharT*`
`const_iterator`	`const CharT*`
`size_type`	`std::size_t`
`view_type`	`basic_string_view<CharT>`
`string_type`	`basic_string<CharT>`

The mparser maintains three pointers, namely, *anchor*, *begin*, and *end*. The part `[anchor, begin)` is considered as the matched part, which the parser has scanned, while the part `[begin, end)` is the remaining part, which the parser may process in future. It also maintains a boolean flag to indicate whether the parsing failed.

Constructors

`basic_mparser` (*iterator* *a*, *iterator* *b*, *iterator* *e*, bool *fail* = false) **noexcept**

Construct an m-parser with all fields given.

Parameters

- **`a`** – The anchor pointer.

- **b** – The beginning pointer.
- **e** – The pass-by-end pointer.
- **fail** – Whether the parser is tagged as *failed*. Default is `false`.

basic_mparser(view_type *sv*)
Construct an m-parser from a string view.

It sets both `anchor` and `begin` to `sv.data()`, and `end` to `sv.data() + sv.size()`.

basic_mparser(**const** string_type &*s*)
Construct an m-parser from a standard string.

It is equivalent to `basic_mparser(view_type(s))`.

basic_mparser(**const** CharT **s*)
Construct an m-parser over a C-string.

It is equivalent to `basic_mparser(view_type(s))`.

basic_mparser(view_type *sv*, size_type *pos*)
Construct an m-parser from a string view, starting from `pos`.

It sets both `anchor` and `begin` to `sv.data() + pos`, and `end` to `sv.data() + sv.size()`.

basic_mparser(**const** string_type &*s*, size_type *pos*)
Equivalent to `basic_mparser(view_type(s), pos)`.

basic_mparser(**const** CharT **s*, size_type *pos*)
Equivalent to `basic_mparser(view_type(s), pos)`.

Note: The string range does not own the memory. It only maintains pointers. Hence, it is important to ensure that the underlying string remains valid throughout its lifetime.

Properties

iterator **anchor**() **const**
Get the anchor pointer (of the matched part).

iterator **begin**() **const**
Get the beginning pointer (of the remaining part).

iterator **end**() **const**
Get the pass-by-end pointer.

operator bool() **const noexcept**
Return `!failed()`.

bool **failed**() **const noexcept**
Return whether the parsing was failed.

size_type **matched_size**() **const noexcept**
Get the size of the matched part, *i.e.* `[anchor, begin)`.

bool **remain**() **const noexcept**
Get whether the remaining part is non-empty, *i.e.* `begin != end`.

size_type **remain_size**() **const noexcept**
Get the size of the remaining part.

CharT **operator[]**(size_type *i*) **const**
Get the `i`-th character of the remaining part (without bounds checking).

CharT **at**(size_type *i*) **const**
Get the `i`-th character of the remaining part (with bounds checking).

CharT **front**() **const**
Get the first character of the remaining part.

view_type **matched_view**() **const noexcept**
Convert the matched part to a string view.

string_type **matched_string**() **const**
Convert the matched part to a standard string.

view_type **remain_view**() **const**
Convert the remaining part to a string view.

bool **next_is**(CharT *c*) **const noexcept**
Test whether the next character is `c`.

Equivalent to `!failed() && remain() && front() == c`.

bool **next_is**(view_type *sv*) **const noexcept**
Test whether the parser is not failed and the remaining part starts with `sv`.

bool **next_is**(**const** char **s*) **const**
Equivalent to `next_is(view_type(s))`.

Manipulation

The class `basic_mparser` provides a series of methods to manipulate the m-parser. Note that these methods do not change the current m-parser, instead, they return the manipulated m-parser as a new one.

basic_mparser **pop**() **const noexcept**
Pop the matched part, *i.e.* move `anchor` to `begin`.

basic_mparser **pop_to**(*string_view* &*dst*) **const noexcept**
Store the matched part to `dst` and then pop.

basic_mparser **skip_to**(*iterator p*) **const**
Move `begin` to `p`.

basic_mparser **skip_by**(size_type *n*) **const**
Move `begin` forward by `n` characters.

Equivalent to `skip_to(begin() + n)`.

basic_mparser **skip**(Pred &&*pred*) **const**
Skip all characters that satisfy `pred`, *i.e.* those characters on which `pred` yields `true`.

basic_mparser **skip_spaces**() **const noexcept**
Skip spaces.

Equivalent to `skip(chars::is_space)`.

basic_mparser **skip_until**(Pred &&*pred*) **const**
Skip until it reaches the end or hits a character that satisfies `pred`.

basic_mparser **fail**() **const noexcept**
Tag the m-parser as failed.

We also provide a set of *manipulators*, which can be used with the insertion operator, to accomplish similar functionalities. The advantage of such manipulators is that they can be used in a way similar to a matching rule. These manipulators are defined within the namespace `clue::mpar`.

`mpar::pop()`
Get a manipulator that pops the matched part, moving `anchor` to `begin`.

Note `m >> mpar::pop()` is equivalent to `m.pop()`.

`mpar::pop_to(`*string_view* *&dst*`)`
Get a manipulator that pops the matched part, and stores it to `dst`.

Note `m >> mpar::pop_to(dst)` is equivalent to `m.pop_to(dst)`.

`mpar::skip_by(`size_t *n*`)`
Get a manipulator that skips `n` characters.

Note `m >> mpar::skip_by(n)` is equivalent to `m.skip(n)`.

`mpar::skip(`**const** Pred *&pred*`)`
Get a manipulator that skips all characters that satisfy `pred`.

Note `m >> mpar::skip(pred)` is equivalent to `m.skip(pred)`.

`mpar::skip_until(`**const** Pred *&pred*`)`
Get a manipulator that skips until it reaches the end or hits a character that satisfies `pred`.

Note `m >> mpar::skip_until(pred)` is equivalent to `m.skip_until(pred)`.

Matching Rules

basic_mparser **operator>>**(*basic_mparser* *&m*, Rule *&&rule*) **const**
Monadic binding with a given rule.

Generally, `rule` is a function that tries to match a pattern with the remaining part (or a leading sub-string thereof). Specifically, `rule` takes as input the beginning pointer `b` and pass-by-end pointer `e` and returns a m-parser (of class `basic_mparser<CharT>`) that indciates the parsing results.

The returned parser `rm` should satisfy the following requirement:

- `rm.anchor() == b`
- `rm.end() == e`
- `rm.begin()` indicates the pass-by-end of the matched part.
- `rm.failed()` indicates whether the matching failed.

This binding operator `>>` works as follows:

- If `m.failed()`, it returns `m` immediately.
- Otherwise, it tries to match the remaining part by calling `rm = rule(m.begin(), m.end())`. If `rm.failed()`, it returns `m.fail()`, otherwise it forwards the beginning pointer of the remaining part to `rm.begin()`, namely, returning `m.skip_to(rm.begin())`.

We provide a series of pre-defined rules and combinators. By combining these facilities in different ways, one can derive parsers for different purposes. All such facilities are within the namespace `clue::mpar`.

ch(**const** Pred *&pred*)
Get a rule that matches a character satisfying `pred`.

See *Predicates* for a set of pre-defined predicates on characters, *e.g.* `chars::is_space`, `chars::is_digit`, etc.

ch(char *c*)
Get a rule that matches a character `c`.

Note This is equivalent to `ch(eq(c))`.

ch_in(**const** char **s*)
Get a rule that matches a character containes in `s`.

Note This is equivalent to `ch(in(s))`.

chs(**const** Pred &*pred*)
Get a rule that matches one or more characters that satisfy `pred`.

chs(**const** Pred &*pred*, int *lb*)
Get a rule that matches a sub-string that with at least `lb` characters that satisfy `pred`.

If `lb` is zero, it can match no character (but still considered as a successful match).

chs(**const** Pred &*pred*, int *lb*, int *ub*)
Get a rule that matches a sub-string that with at least `lb` and at most `ub` characters that satisfy `pred`.

If `ub` is set to `-1`, there is no upper limit.

chs_fix(**const** Pred &*pred*, int *n*)
Get a rule that matches exactly `n` characters that satisfy `pred`.

alphas()
Equivalent to `chs(chars::is_alpha)`.

digits()
Equivalent to `chs(chars::is_digit)`.

alnums()
Equivalent to `chs(chars::is_alnum)`.

blanks()
Equivalent to `chs(chars::is_blank)`.

blanks(int *lb*)
Equivalent to `chs(chars::is_blank, lb)`.

term(*string_view sv*)
Get a rule that matches a given string.

term(**const** CharT **s*)
Get a rule that matches a given string.

Note It is equivalent to `term(basic_string_view<CharT>(s))`.

maybe(**const** Rule &*rule*)
Get a rule that *optionally* matches `rule`.

For a typical rule (except for example `chs(pred, 0)`), if the leading part of the remaining part is not a match, it will return a failed m-parser. This rule simply returns the current parser (without tagging it as failed) when no match is found.

either_of(**const** R1 &*r1*, ...)
Construct a rule that combines one or more rules in an either-or way.

Particularly, it tries the given rules one-by-one until it finds a match. If all given rules failed, it returns a the current m-parser tagged as failed.

chain(**const** R1 &*r1*, ...)
Construct a chain-rule that matches a sequence of patterns.

identifier()
Get a rule that matches a typical identifier.

A string is considered as an identifier, if it begins with _ or an alphabetic character, which is then *optionally* followed by a sequence of characters that are either _, alphabetic, or digits.

integer()
Get a rule that matches an integer.

An integer pattern *optionally* starts with + or −, and then it follows with a sequence of digits.

realnum()
Get a rule that matches a real number in decimal or scientific format, *e.g.* `12`, `-12.34`, `2.5e-6`, etc.

List Parsing

mparser **foreach_term**(*mparser m*, **const** Term &*term*, **const** Sep &*sep*, F &&*f*)
This function parses a delimited list.

It scans a list according to the given pattern as `term1 sep term2 sep ...` until it reaches the end or a part that does not satisfy the required pattern. Whenever it encounters a new term, it invokes the input functor `f` on the term.

Parameters

- **m** – The input m-parser.
- **term** – The rule for matching a term.
- **sep** – The rule for matching a separator.
- **f** – The functor to be invoked on each term (as a `string_view`).

It returns an m-parser skipped to the end of the matched part.

Optional spaces are allowed between terms and separators.

See the example at the beginning of this document section.

Meta-programming tools

Extensions of Type Traits

In C++11, a collection of type traits have been introduced into the standard library (in the header `<type_traits>`). While they are very useful, using these type traits in practice is sometimes cumbersome. For example, to add a const qualifier to a type, one has to write

```
using const_type = typename std::add_const<my_type>::type;
```

The need to use `typename` and `::type` introduces unnecessary noise to the code. In C++14, a set of helpers are introduced, such as

```
template<class T>
using add_const_t = typename add_const<T>::type;
```

This makes the codes that transform types more concise. In particular, with `add_const_t`, one can write:

```
using const_type = add_const_t<my_type>;
```

In *CLUE++*, we define all these helpers in the header `<clue/type_traits.hpp>`, so that they can be used within C++11 environment. In particular, the following helpers are provided. All these *backported* helpers are within the namespace `clue`.

Note: Below is just a list. For detailed descriptions of these type traits, please refer to the standard documentation.

```
// for const-volatility specifiers

template<class T>
using remove_cv_t = typename ::std::remove_cv<T>::type;
template<class T>
using remove_const_t = typename ::std::remove_const<T>::type;
template<class T>
using remove_volatile_t = typename ::std::remove_volatile<T>::type;

template<class T>
using add_cv_t = typename ::std::add_cv<T>::type;
template<class T>
using add_const_t = typename ::std::add_const<T>::type;
template<class T>
using add_volatile_t = typename ::std::add_volatile<T>::type;

// for references

template<class T>
using remove_reference_t = typename ::std::remove_reference<T>::type;
template<class T>
using add_lvalue_reference_t = typename ::std::add_lvalue_reference<T>::type;
template<class T>
using add_rvalue_reference_t = typename ::std::add_rvalue_reference<T>::type;

// for pointers
template<class T>
using remove_pointer_t = typename ::std::remove_pointer<T>::type;
template<class T>
using add_pointer_t = typename ::std::add_pointer<T>::type;

// for sign modifiers

template<class T>
using make_signed_t = typename ::std::make_signed<T>::type;
template<class T>
using make_unsigned_t = typename ::std::make_unsigned<T>::type;

// for arrays

template<class T>
using remove_extent_t = typename ::std::remove_extent<T>::type;
template<class T>
using remove_all_extents_t = typename ::std::remove_all_extents<T>::type;

// static conditions

template<bool B, class T = void>
using enable_if_t = typename ::std::enable_if<B,T>::type;
template<bool B, class T, class F>
using conditional_t = typename ::std::conditional<B,T,F>::type;

// other transformations
```

```
template<class T>
using decay_t = typename ::std::decay<T>::type;
template<class... T>
using common_type_t = typename ::std::common_type<T...>::type;
template<class T>
using underlying_type_t = typename ::std::underlying_type<T>::type;
template<class T>
using result_of_t = typename ::std::result_of<T>::type;
```

Meta-types and Meta-functions

Template meta-programming has become an indispensible part of modern C++. In C++11, new features such as *Variadic template* and *Template alias* makes meta-programming much more efficient and convenient than before. *CLUE++* provides a set of tools to facilitate meta programming, which take full advantage of these new C++ features.

For those who are not familiar with C++ meta-programming, Andrzej has a great blog that provides an excellent introduction of this topic.

Important Note: all meta-programming facilities in *CLUE++* are within the namespace `clue::meta`.

Basic types

A set of types to support meta-programming:

```
// Note: all names below are within the namespace clue::meta

using std::integral_constant;

// Indicator of a C++ type
template<typename T>
struct type_ {
    using type = T;
};

// Extract an encapsulated type
template<typename A>
using get_type = typename A::type;

// Indicator of a nil type (nothing)
struct nil_{};

// Static boolean value
template<bool V>
using bool_ = integral_constant<bool, V>;

using true_  = bool_<true>;
using false_ = bool_<false>;

// Static integral values
template<char  V> using char_  = integral_constant<char,  V>;
template<int   V> using int_   = integral_constant<int,   V>;
template<long  V> using long_  = integral_constant<long,  V>;
template<short V> using short_ = integral_constant<short, V>;

template<unsigned char  V> using uchar_  = integral_constant<unsigned char,  V>;
```

```
template<unsigned int   V> using uint_   = integral_constant<unsigned int,   V>;
template<unsigned long  V> using ulong_  = integral_constant<unsigned long,  V>;
template<unsigned short V> using ushort_ = integral_constant<unsigned short, V>;

// Static size value
template<size_t V> using size_ = integral_constant<size_t, V>;

// Extract the value type of a static value.
template<class A>
using value_type_of = typename A::value_type;
```

Sometimes, it is useful to combine two types. For this purpose, we provide a `pair_` type to express a pair of types, as well as meta-functions `first` and `second` to retrieve them.

```
// Note: all names below are within the namespace clue::meta

template<typename T1, typename T2>
struct pair_ {
    using first_type = T1;
    using second_type = T2;
};

template<typename T1, typename T2>
struct first<pair_<T1, T2>> {
    using type = T1;
};

template<typename T1, typename T2>
struct second<pair_<T1, T2>> {
    using type = T2;
};

template<class A> using first_t  = typename first<A>::type;
template<class A> using second_t = typename second<A>::type;
```

Note: The meta-functions `first` and `second` are also specialized for other meta data structures, such as the *meta sequence*.

Static Index Sequence

The library provides useful facilities to construct static index sequence, which is useful for splatting elements of a tuples as arguments.

```
// index_seq can be used to represent a static sequence of indexes
template<size_t... Inds>
struct index_seq{};

// make_index_seq<N> constructs index_seq<0, ..., N-1>

make_index_seq<0>;  // -> index_seq<>
make_index_seq<1>;  // -> index_seq<1>
make_index_seq<4>;  // -> index_seq<0, 1, 2, 3>
```

The following example shows how one can leverage `make_index_seq` to splat tuple arguments.

```
// suppose you have a function join can accepts arbitrary number of arguments
template<class... Args>
void join(const Args&... args) { /* ... */ }

// the join_tup function can splat elements of a tuple

template<class... Args, size_t... I>
void join_tup_impl(const std::tuple<Args...>& tup, clue::meta::index_seq<I...>) {
    join(std::get<I>(tup)...);
}

template<class... Args>
void join_tup(const std::tuple<Args...>& tup) {
    join_tup_impl(tup, clue::meta::make_index_seq<sizeof...(Args)>{});
}

join_tup(std::make_tuple("abc", "xyz", 123));
```

Basic functions

The library also has a series of meta-functions to work with types or static values.

Arithmetic functions

```
// Note: all names below are within the namespace clue::meta

template<typename A>
using negate = integral_constant<value_type_of<A>, -A::value>;

template<typename A>
using next = integral_constant<value_type_of<A>, A::value+1>;

template<typename A>
using prev = integral_constant<value_type_of<A>, A::value-1>;

template<typename A, typename B>
using plus = integral_constant<value_type_of<A>, A::value + B::value>;

template<typename A, typename B>
using minus = integral_constant<value_type_of<A>, A::value - B::value>;

template<typename A, typename B>
using mul = integral_constant<value_type_of<A>, A::value * B::value>;

template<typename A, typename B>
using div = integral_constant<value_type_of<A>, A::value / B::value>;

template<typename A, typename B>
using mod = integral_constant<value_type_of<A>, A::value % B::value>;

// aliases, to cover the names in <functional>
template<typename A, typename B> using multiplies = mul<A, B>;
template<typename A, typename B> using divides = div<A, B>;
template<typename A, typename B> using modulo = mod<A, B>;
```

Comparison functions

```
// Note: all names below are within the namespace clue::meta

template<typename A, typename B> using eq = bool_<(A::value == B::value)>;
template<typename A, typename B> using ne = bool_<(A::value != B::value)>;
template<typename A, typename B> using gt = bool_<(A::value >  B::value)>;
template<typename A, typename B> using ge = bool_<(A::value >= B::value)>;
template<typename A, typename B> using lt = bool_<(A::value <  B::value)>;
template<typename A, typename B> using le = bool_<(A::value <= B::value)>;

// aliases, to cover the names in <functional>
template<typename A, typename B> using equal_to      = eq<A, B>;
template<typename A, typename B> using not_equal_to  = ne<A, B>;
template<typename A, typename B> using greater       = gt<A, B>;
template<typename A, typename B> using greater_equal = ge<A, B>;
template<typename A, typename B> using less          = lt<A, B>;
template<typename A, typename B> using less_equal    = le<A, B>;
```

Logical functions

class not_
: The member constant `not_<A>::value` is equal to `!A::value`.

class and_
: The member constant `and_<A, B>::value` is `true` iff both `A::value` and `B::value` is true.

class or_
: The member constant `or_<A, B>::value` is `true` iff either `A::value` or `B::value` is true.

Note: The meta-functions `and_<A, B>` and `or_<A, B>` implement the *short-circuit behavior*. In particular, when `A::value == false`, `and_<A, B>::value` is set to `false` without examining the internals of `B`. Likewise, when `A::value == true`, `or_<A, B>::value` is set to `true` without examining the internals of `B`.

Select

C++11 provides `std::conditional` for static dispatch based on a condition. However, using this type in practice, especially in the cases with multiple branches, is very cumbersome. Below is an example that uses `std::conditional` to map a numeric value to a signed value type.

```
#include <type_traits>

template<typename T>
using signed_type =
    typename std::conditional<
        std::is_integral<T>::value,
        typename std::conditional<std::is_unsigned<T>::value,
            typename std::make_signed<T>::type,
            T
        >::type,
        typename std::conditional<std::is_floating_point<T>::value,
            T,
            nil_t
```

```
        >::type
    >::type;
```

With the meta-function `select` and the helper alias `select_t`, this can be expressed in a much more elegant and concise way:

```
#include <clue/meta.hpp>

using namespace clue;

template<typename T>
using signed_type =
    meta::select_t<
        std::is_unsigned<T>,        std::make_signed<T>,
        std::is_signed<T>,          meta::type_<T>,
        std::is_floating_point<T>, meta::type_<T>,
        meta::type_<nil_t> >;
```

Specifically, `meta::select` is a variadic class template, described as follows:

- `select<C1, A1, R>` has a member typedef `type` which is equal to `A1::type` when `C1::value` is true, or `R::type` otherwise.
- This meta-function can accept arbitrary odd number of arguments. Generally, `select<C1, A1, C2, A2, ..., Cm, Am, R>` has a member typedef `type` which is equal to `A1::type` when `C1::value` is true, otherwise, it is equal to `A2::type` if `C2::value` is true, and so on. If no conditions are met, it is set to `R::type`.

A helper alias `select_t` is provided to further simplify the use:

```
template<typename... Args>
using select_t = typename select<Args...>::type;
```

Note: The meta-function `select` implements a *short-circuit behavior*. It examines the conditions sequentially, and once it finds a condition that is `true`, it extracts the next type, and will not continue to examine following conditions.

Variadic Reduction

A set of variadic meta-functions are provided to perform reduction over static values.

class `meta::sum`
: `meta::sum<args...>` has a member constant `value` that equals the sum of argument's member values.

class `meta::prod`
: `meta::prod<args...>` has a member constant `value` that equals the product of argument's member values.

class `meta::maximum`
: `meta::maximum<args...>` has a member constant `value` that equals the maximum of argument's member values.

class `meta::minimum`
: `meta::minimum<args...>` has a member constant `value` that equals the minimum of argument's member values.

class `meta::all`
: `meta::all<args...>` has a member constant `value`, which equals `true` if all argument's member values are `true`, or `false` otherwise.

  Note `all<>::value == true`.

class `meta::any`
: `meta::any<args...>` has a member constant `value`, which equals `true` if any of the argument's member value is `true`, or `false` otherwise.

  Note `any<>::value == false`.

class `meta::count_true`
: `meta::count_true<args...>` has a member constant `value`, which equals the number of arguments whose member value is `true`.

class `meta::count_false`
: `meta::count_false<args...>` has a member constant `value`, which equals the number of arguments whose member value is `false`.

class `meta::all_same`
: `meta::all_same<args...>` has a member constant `value`, which indicates whether all argument types are the same.

Note: The meta-functions `all` and `any` both implement the *short-circuit behaviors*. They won't look further once the resultant value can be determined.

Meta-sequence: Sequence of Types

In meta-programming, it is sometimes useful to process a list of types, in a way that is similar to `std::vector`. *CLUE++* provides facilities to support such operations in *compile-time*. Like other meta-programming tools, all these facilities are also in the namespace `clue::meta`.

Meta-sequence

In *CLUE++*, we use a variadic class template `meta::seq_`, defined below, to indicate a sequence of types.

```
// Within the namespace clue::meta:
template<typename... Elems> struct seq_;
```

We provide a series of meta-functions that emulate the `std::vector` API to work with such a sequence of types. Below is an example that illustrates the use of `seq_` and some of the meta-functions working with it.

```
using namespace clue::meta;
using meta::seq_;
using i1 = meta::int_<1>;
using i2 = meta::int_<2>;
using i3 = meta::int_<3>;
using i4 = meta::int_<4>;

// define a sequence of static values
using s = seq_<i1, i2, i3>;

constexpr size_t n = meta::size<s>::value // n = 3;
```

```
using xf = meta::front_t<s>;  // xf is i1
using xb = meta::back_t<s>;   // xb is i3
using x0 = meta::at_t<s, 0>;  // x0 is i1
using x1 = meta::at_t<s, 1>;  // x1 is i2

using r1 = meta::push_back_t<s, i4>;
// r1 is seq_<i1, i2, i3, i4>

using r2 = meta::push_front_t<s, i4>;
// r2 is seq_<i4, i1, i2, i3>

using r3 = meta::pop_front_t<s>;
// r3 is seq_<i2, i3>

using r4 = meta::pop_back_t<s>;
// r4 is seq_<i1, i2>

using rr = meta::reverse<s>;
// rr is seq_<i3, i2, i1>

using rt = meta::transform<meta::next, s>;
// rt is seq_<i2, i3, i4>

constexpr int v1 = meta::sum<s>::value; // v1 = 6;
constexpr int v2 = meta::max<s>::value; // v2 = 3;
constexpr int v3 = meta::min<s>::value; // v3 = 1;
```

Basic properties

class `meta::size`
: `meta::size< seq_<elems...> >` has a member constant `value` that equals to the number of element types.

class `meta::empty`
: `meta::empty< seq_<elems...> >` has a member constant `value` that is `true` when the number of element types is zero, or `false` otherwise.

Element type access

class `meta::front`
: `meta::front< seq_<elems...> >` has a member typedef `type` corresponding to the first element type.

class `meta::back`
: `meta::back< seq_<elems...> >` has a member typedef `type` corresponding to the last element type in the sequence.

class `meta::at`
: `meta::at< seq_<elems...>, I >` has a member typedef `type` corresponding to the `I`-th element type of the sequence.

class `meta::first`
: `meta::first< seq_<elems...> >` has a member typedef `type` corresponding to the first element type. (Equivalent to using `meta::front`).

class `meta::second`
: `meta::second< seq_<elems...> >` has a member typedef `type` corresponding to the second element

type.

Helper aliases are provided for all these meta functions:

```
// Within the namespace clue::meta:

template<class Seq> using front_t  = typename front<Seq>::type;
template<class Seq> using back_t   = typename back<Seq>::type;
template<class Seq> using first_t  = typename first<Seq>::type;
template<class Seq> using second_t = typename second<Seq>::type;

template<class Seq, size_t N>
using at_t = typename at<Seq, N>::type;
```

Modifiers

class `meta::clear`
: `meta::clear< seq_<elems...> >` has a member typedef `type = meta::seq_<>`.

class `meta::pop_front`
: `meta::pop_front< seq_<elems...> >` has a member typedef `type` which is a meta sequence with the first element type excluded.

class `meta::pop_back`
: `meta::pop_back< seq_<elems...> >` has a member typedef `type` which is a meta sequence with the last element type excluded.

class `meta::push_front`
: `meta::push_front< seq_<elems...>, X >` has a member typedef `type` which prepends a type `X` to the front of the input meta sequence.

class `meta::push_back`
: `meta::push_back< seq_<elems...>, X >` has a member typedef `type` which appends a type `X` to the back of the input meta sequence.

Helper aliases are provided for all these meta functions:

```
// Within the namespace clue::meta:

template<class Seq> using clear_t = typename clear<Seq>::type;
template<class Seq> using pop_front_t = typename pop_front<Seq>::type;
template<class Seq> using pop_back_t  = typename pop_back<Seq>::type;

template<class Seq, typename X>
using push_front_t = typename push_front<Seq, X>::type;

template<class Seq, typename X>
using push_back_t = typename push_back<Seq, X>::type;
```

Sequence reduction

All variadic reduction functions are specialized to perform reduction over a sequence, as

```
template<typename... Elems>
struct sum<seq_<Elems...>> : public sum<Elems...> {};

template<typename... Elems>
```

```
struct prod<seq_<Elems...>> : public prod<Elems...> {};

template<typename... Elems>
struct max<seq_<Elems...>> : public max<Elems...> {};

template<typename... Elems>
struct min<seq_<Elems...>> : public min<Elems...> {};

template<typename... Elems>
struct all<seq_<Elems...>> : public all<Elems...> {};

template<typename... Elems>
struct any<seq_<Elems...>> : public any<Elems...> {};

template<typename... Elems>
struct count_true<seq_<Elems...>> : public count_true<Elems...> {};

template<typename... Elems>
struct count_false<seq_<Elems...>> : public count_false<Elems...> {};
```

Algorithms

We also implement a collection of algorithms to work with meta sequences.

class `meta::cat`
: `meta::cat<S1, S2>` has a member typedef `type` that is a concatenation of two meta sequences `S1` and `S2`.

class `meta::zip`
: `meta::zip<S1, S2>` has a member typedef `type` that zips two meta sequences `S1` and `S2` of the same length.

  Example:

```
using namespace clue;
using S1 = meta::seq_<char, int>;
using S2 = meta::seq_<float, double>;

using R = typename zip<S1, S2>::type;
// meta::seq_<
//   meta::pair_<char, float>,
//   meta::pair_<int,  double>
// >
```

class `meta::repeat`
: `meta::repeat<X, N>` has a member typedef `type` which is a meta sequence that repeats the type `X` for `N` times.

  Example `meta::repeat<int, 3>::type` is `meta::seq_<int, int, int>`.

class `meta::reverse`
: `meta::reverse<S>` has a member typedef `type` which is a reversed meta sequence.

  Example `meta::reverse<meta::seq_<char, short, int>>::type` is `meta::seq_<int, short, char>`.

class `meta::transform`
: `meta::transform<F, S>` has a member typedef `type` which is the transformed sequence obtained by

applying a meta-function `F` to each element type of `S`.

class `meta::transform2`
`meta::transform2<F, S1, S2>` has a member typedef `type` which is the transformed sequence obtained by applying a meta-function `F` to each element type of `S1` and that of `S2`.

Examples:

```
using namespace clue;
using meta::int_;
using meta::seq_;

using S1 = seq_<int_<1>, int_<2>, int_<3>>;
using S2 = seq_<int_<4>, int_<5>, int_<6>>;

using U = typename meta::transform<meta::next, S1>::type;
// U is seq_<int_<2>, int_<3>, int_<4>>

using V = typename meta::transform2<meta::plus, S1, S2>::type;
// V is seq_<int_<5>, int_<7>, int_<9>>
```

class `meta::filter`
With a member typedef `type` which is the filtered sequence by retaining the element types `X` in `S` for which `Pred<X>::value` is `true`.

Examples:

```
using namespace clue;
using meta::int_;
using meta::seq_;

using S = seq_<int_<1>, int_<2>, int_<3>>;

template<class A>
struct is_odd : public bool_<(A::value % 2 == 1)> {};

using R = typename meta::filter<is_odd, S>::type;
// R is seq_<int_<1>, int_<3>>;
```

class `exists`
`exists<X, S>` has a member constant `value` that indicates whether the type `X` exists as an element type of `S`.

class `exists_if`
`exists_if<Pred, S>` has a member constant `value` which is `true` if there exist element types `X` of `S` such that `Pred<X>::value` is `true`.

class `count`
`count<X, S>` has a member constant `value` which is equal to the number of occurrences of a type `X` in the sequence `S`.

class `count_if`
`count_if<X, S>` has a member constant `value` which is equal to the number of element types `X` in `S` that satisfy the condition `Pred<X>::value` is `true`.

Helper aliases are provided for all algorithms that transform types:

```
template<class S1, class S2>   using cat_t    = typename cat<S1, S2>::type;
template<class S1, class S2>   using zip_t    = typename zip<S1, S2>::type;
template<typename X, size_t N> using repeat_t = typename repeat<X, N>::type;
```

```
template<class Seq> using reverse_t = typename reverse<Seq>::type;

template<template<typename X> class F, class Seq>
using transform_t = typename transform<F, Seq>::type;

template<template<typename X, typename Y> class F, class S1, class S2>
using transform2_t = typename transform2<F, S1, S2>::type;

template<template<typename X> class Pred, class Seq>
using filter_t = typename filter<Pred, Seq>::type;
```

Concurrent programming

Shared Mutex (Read/write lock)

In C++14/C++17, a new kind of *mutex*, called *shared mutex*, is introduced.

Unlike other mutex types, a *shared mutex* has two levels of access:

- **shared:** several threads can share ownership of the same mutex.
- **exclusive:** only one thread can own the mutex.

This is useful in situations where we may allow multiple parallel readers or one writer to operate on a block of data.

As it is part of the future C++ standard, cppreference already has detailed documentation. Below is just a brief summary of the relevant classes and functions. (In *CLUE++*, we provide a C++11 implementation for these types, and the names are in the namespace `clue`, instead of `std`).

Here is an example of how shared_mutex can be used in practice.

```
using namespace clue;

class MyData {
    std::vector<double> data_;
    mutable shared_mutex mut_;   // the mutex to protect data_;

public:
    void write() {
        unique_lock<shared_mutex> lk(mut_);
        // ... write to data_ ...
    }

    void read() const {
        shared_lock<shared_mutex> lk(mut_);
        // ... read the data ...
    }
};

// --- main program ---

MyData a;

std::thread t_write([&](){
    a.write();
    sleep_for_a_while();
```

```
});

std::thread t_read1([&](){
    a.read();
});

std::thread t_read2([&](){
    a.read();
});

// t_read1 and t_read2 can simultaneously read a,
// while t_write is not writing
```

Class `shared_mutex`

class `shared_mutex`

A mutex class that allows multiple thread to maintain shared ownership at the same time, or a thread to maintain exclusive ownership.

> **Note** It is a default constructor and a destructor, while the copy constructor and assignment operator are deleted.
>
> **Note** This class is accepted to the C++17 standard.

The table below lists its member functions:

void **`lock`**`()`

Locks the mutex (acquires exclusive ownership), blocks if the mutex is not available.

bool **`try_lock`**`()`

Tries to lock the mutex.

Returns immediately. On successful lock acquisition returns true, otherwise returns false.

void **`unlock`**`()`

Unlocks the mutex.

> **Note** The mutex must be locked by the current thread of execution, otherwise, the behavior is undefined.

void **`lock_shared`**`()`

Acquires shared ownership of the mutex.

If another thread is holding the mutex in exclusive ownership, a call to lock_shared will block execution until shared ownership can be acquired.

bool **`try_lock_shared`**`()`

Tries to lock the mutex in shared mode. Returns immediately. On successful lock acquisition returns true, otherwise returns false.

void **`unlock_shared`**`()`

Releases the mutex from shared ownership by the calling thread.

> **Note** The mutex must be locked by the current thread of execution in shared mode, otherwise, the behavior is undefined.

Class `shared_timed_mutex`

class `shared_time_mutex`

Similar to `shared_mutex`, `shared_timed_mutex` allows multiple shared ownership or one exclusive ownership. In addition, it provides the ability to try to acquire the exclusive or shared ownership with a timeout.

> **Note** This class is introduced in C++14.

The class `shared_timed_mutex` provides all the member funtions as in `shared_mutex`. In addition, it provides the following members:

bool **`try_lock_for`** (**const** std::chrono::duration<Rep, Period> &*duration*)

Tries to lock the mutex (acquire exclusive ownership).

Blocks until specified `duration` has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

bool **`try_lock_until`** (**const** std::chrono::time_point<Clock, Duration> &*t*)

Tries to lock the mutex (acquire exclusive ownership).

Blocks until specified due time `t` has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

bool **`try_lock_shared_for`** (**const** std::chrono::duration<Rep, Period> &*duration*)

Tries to lock the mutex in shared mode (acquire shared ownership).

Blocks until specified `duration` has elapsed or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

bool **`try_lock_shared_until`** (**const** std::chrono::time_point<Clock, Duration> &*t*)

Tries to lock the mutex in shared mode (acquire shared ownership).

Blocks until specified due time `t` has been reached or the lock is acquired, whichever comes first. On successful lock acquisition returns true, otherwise returns false.

Class `shared_lock`

class `shared_lock`

Formal

```
template <class Mutex>
class shared_lock;
```

The class shared_lock is a general-purpose shared mutex ownership wrapper allowing deferred locking, timed locking and transfer of lock ownership.

The shared_lock locks the associated shared mutex in shared mode (to lock it in exclusive mode, `std::unique_lock` can be used)

Concurrent Counter

In concurrent programming, it is not uncommon that some function is triggered by a certain condition (*e.g.* a number grow beyond certain threshold). *CLUE* provides a class `concurrent_counter` to implement this. This class in the header file `<clue/concurrent_counter.hpp>`.

class `concurrent_counter`

Concurrent counter.

It has a default constructor that initializes the count to zero. There's another constructor that accepts an initial count.

A concurrent counter is not copyable and not movable.

This class has the following member functions:

long **get**()
Get the current value of the counter;

void **set**(long *v*)
Set a new count value, and notify all waiting threads.

void **inc**(long *x* = 1)
Increment the count by `x` (default is `1`), and notify all waiting threads.

void **dec**(long *x* = 1)
Decrement the count by `x` (default is `1`), and notify all waiting threads.

void **reset**()
Reset the count value to zero, and notify all waiting threads. Equivalent to `set(0)`.

void **wait**(Pred &&*pred*)
Waits until the count meets the specified condition (`pred(count)` returns `true`).

Note: *CLUE* has provided a series of predicates that can be useful here. (Refer to *Predicates* for details). For example, if you want to wait until when the count value goes above a certain threshold `m`, then you may write `wait( clue::ge(m) )` (or `wait( ge(m) )` when the namespace `clue` is being used).

---

void **wait**(v)
Waits until the count hits the given value `v`. Equivalent to `wait( clue::eq(v) )`.

bool **wait_for**(Pred &&*pred*, **const** std::chrono::duration &*dur*)
Waits until the count meets the specified condition or the duration `dur` elapses, whichever comes first.

It returns whether the count meets the condition upon returning.

bool **wait_for**(long *v*, **const** std::chrono::duration &*dur*)
Equivalent to `wait_for(clue::eq(v), dur)`.

bool **wait_until**(Pred &&*pred*, **const** std::chrono::time_point &*t*)
Waits until the count meets the specified condition or the time-out `t`, whichever comes first.

It returns whether the count meets the condition upon returning.

bool **wait_until**(long *v*, **const** std::chrono::time_point &*t*)
Equivalent to `wait_until(clue::eq(v), t)`.

Examples: The following example shows how a concurrent counter can be used in practice. In this example, a message will be printed when the accumulated value exceeds *100*.

```
clue::concurrent_counter accum_val(0);

std::thread worker([&](){
    for (size_t i = 0; i < 100; ++i) {
        accum_val.inc(static_cast<long>(i + 1));
    }
});

std::thread listener([&](){
    accum_val.wait( clue::gt(100) );
```

```
    std::printf("accum_val goes beyond 100!\n");
});

worker.join();
listener.join();
```

The source file `examples/ex_cccounter.cpp` provides another example.

Concurrent Queue

Concurrent queue is very useful in concurrent programming. For example, task queue can be considered as a special kind of concurrent queue. *CLUE* implements a concurrent queue class, in header file `<clue/concurrent_queue.hpp>`.

template<T>
class concurrent_queue
Concurrent queue class. `T` is the element type.

This class has a default constructor, but it is not copyable or movable. The class provides the following member functions:

size_t **size**() **const**
Get the number of elements in the queue (at the point this method is being called).

bool **empty**() **const**
Get whether the queue is empty (contains no elements).

void **synchronize**()
Block until all updating (*e.g.* push or pop) are done.

void **clear**()
Clear the queue (pop all remaining elements).

void **push**(**const** T &*x*)
Push an element `x` to the back of the queue.

void **push**(T &&*x*)
Push an element `x` (by moving) to the back of the queue.

void **emplace**(Args&&... *args*)
Construct an element using the given arguments and push it to the back of the queue.

bool **try_pop**(T &*dst*)
If the queue is not empty, pop the element at the front, store it to `dst`, and return `true`. Otherwise, return `false` immediately.

T **wait_pop**()
Wait until the queue is non-empty, and pop the element at the front and return it.

If the queue is already non-empty, it pops the front element and returns it immediately.

Note: All updating methods, including `push`, `emplace`, `try_pop`, and `wait_pop`, are thread-safe. It is safe to call these methods in concurrent threads.

Example: The following example shows how to use `concurrent_queue` to implement a task queue. In this example, multiple concurrent producers generate items to be processed, and a consumer fetches them from a queue and process.

```cpp
#include <clue/concurrent_queue.hpp>
#include <vector>
#include <thread>
#include <cstdio>

inline void process_item(double v) {
    std::printf("process item %g\n", v);
}

int main() {
    const size_t M = 2;  // # producers
    const size_t k = 10;  // # items per producer
    size_t remain_nitems = M * k;

    clue::concurrent_queue<double> Q;
    std::vector<std::thread> producers;

    // producers: generate items to be processed
    for (size_t t = 0; t < M; ++t) {
        producers.emplace_back([&Q,t,k](){
            for (size_t i = 0; i < k; ++i) {
                double v = i + 1;
                Q.push(v);
            }
        });
    }

    // consumer: process the items
    std::thread consumer([&](){
        while (remain_nitems > 0) {
            process_item(Q.wait_pop());
            -- remain_nitems;
        }
    });

    // wait for all threads to complete
    for (auto& th: producers) th.join();
    consumer.join();
}
```

Note: To emulate a typical task queue, one may also push functions as elements, and let the consumer invokes each function that it acquires from the queue.

Thread Pool

Thread pool is a very important pattern in concurrent programming. It maps multiple tasks to a smaller number of threads. This is generally more efficient than spawning one thread for each task, especially when the number of tasks is large. *CLUE* provides a `thread_pool` class in the header file `<clue/thread_pool.hpp>`.

class `thread_pool`

A thread pool class.

By default, `thread_pool()` constructs a thread pool with zero threads. `thread_pool(n)` constructs a thread pool with `n` threads. One can modify the number of threads using the `resize()` method later.

A thread pool is not copyable and not movable.

The `thread_pool` class provides the following member functions:

bool **`empty`**`()` **const noexcept**
Return whether the pool is empty (contains no threads).

size_t **`size`**`()` **const noexcept**
Get the number of threads maintained by the pool.

std::thread &**`get_thread`**`(`size_t *i*`)`
Get a reference to the `i`-th thread.

const std::thread &**`get_thread`**`(`size_t *i*`)` **const**
Get a const-reference to the `i`-th thread.

size_t **`num_scheduled_tasks`**`()` **const noexcept**
Get the total number of scheduled tasks (all the tasks that have ever been pushed to the queue).

size_t **`num_completed_tasks`**`()` **const noexcept**
Get the total number of tasks that have been completed.

bool **`stopped`**`()` **const noexcept**
Get whether the thread pool has been stopped (by calling `stop()`).

bool **`done`**`()` **const noexcept**
Get whether all scheduled tasks have been done.

void **`resize`**`(n)`
Resize the pool to `n` threads.

Note: When `n` is less than `size()`, the pool will be shrinked, trailing threads will be terminated and detached.

---

std::future<R> **`schedule`**`(`F *&&f*`)`
Schedule a task.

Here, `f` should be a functor/function that accepts a thread index of type `size_t` as an argument. This function returns a future of class `std::future<R>`, where `R` is the return type of `f`.

This function would wrap `f` into a `packaged_task` and push it to the internal task queue. When a thread is available, it will try to get a task from the front of the internal task queue and execute it.

Note: It is straightforward to push a function that accepts more arguments. One can just wrap it into a closure using C++11's lambda function.

---

void **`synchronize`**`()`
Block until all current tasks have been completed.

This function does not close the thread pool or stop any threads. After synchronization, one can continue to schedule new tasks.

Note: Multiple threads can synchronize a thread pool at the same time. However, it is not allowed to schedule a task while some one is synchronizing.

---

void **`close`**`(`bool *stop_cmd* = false`)`
Close the queue, so that no new tasks can be scheduled.

If `stop_cmd` is explicitly set to `true`, it also sends a stopping command to all threads.

Note: This function returns immediately after closing the queue (and optionally sending the stopping command). It won't wait for the threads to finish (for this purpose, one can call `join()`).

void **`close_and_stop`**`()`
Equivalent to `close(true)`.

void **`join`**`()`
Block until all threads finish.

A thread will finish when the current task is completed and then no task can be acquired (the queue is closed and empty) or when it is stopped explicitly by the stopping command.

Note: The thread pool can only be joined when it is closed. Otherwise a runtime error will be raised. Also, when all threads finish, the function, this function will clear the thread pool, resizing it to `0` threads. However, one can call `resize(n)` to reinstantiate a new set of threads.

void **`wait_done`**`()`
Block until all tasks are completed. Equivalent to `close(); join();`.

void **`stop_and_wait`**`()`
Block until all active tasks (those being run) are completed. Tasks that have been scheduled but have not been launched will remain in the queue (but won't be run by threads).

This is equivalent to `close_and_stop(); join();`.

One can later call `resize()` to re-instate a new set of threads to complete the remaining tasks or call `clear_tasks()` to clear all remaining tasks.

void **`clear_tasks`**`()`
Clear all tasks that remain in the queue. This function won't affect those tasks that are being executed.

Example: The following example shows how to schedule tasks and wait until when they are all done.

```
#include <clue/thread_pool.hpp>

void my_task(double arg) {
    // some processing ...
}

int main() {
    // construct a thread pool with 4 threads
    clue::thread_pool P(4);

    size_t n = 20;
    for (size_t i = 0; i < n; ++i) {
        double a = // get an argument;

        // tid is the index of the thread
        P.schedule([](size_t tid){ my_task(a); });
    }

    // wait until all tasks are completed
    P.wait_done();
}
```

Index

G

H

I

J

K

L

M

N

O