

---

# tl Documentation

**Sy Brand**

**Apr 08, 2021**



---

## Contents

---

<b>1 About</b>	<b>1</b>
<b>2 Index</b>	<b>3</b>
2.1 API Reference . . . . .	3
<b>3 Getting the Code</b>	<b>27</b>
<b>4 License</b>	<b>29</b>
<b>Index</b>	<b>31</b>



# CHAPTER 1

---

## About

---

*tl* is a collection of public-domain (CC0) generic C++ libraries written by [Sy Brand](#). Some parts are backports of more recent standard library features to C++11 (e.g. *integer\_sequence*), some are extensions of existing standard library components (e.g. *optional*), others are handy utilites (e.g. *overload*).



## 2.1 API Reference

### 2.1.1 `expected`

Source code

C++11/14/17 `std::expected` with functional-style extensions and reference support.

#### ***tl::expected***

```
template<class T, class E>  
class expected
```

A *tl::expected*<*T*, *E*> object is an object that contains the storage for another object and manages the lifetime of this contained object *T*. Alternatively it could contain the storage for another unexpected object *E*. The contained object may not be initialized after the expected object has been initialized, and may not be destroyed before the expected object has been destroyed. The initialization state of the contained object is tracked by the expected object.

*T* must not be a reference type, but it may be *void*.

#### **Member Types**

```
using value_type = T
```

```
using error_type = E
```

```
using unexpected_type = unexpected<E>
```

#### **Special Members**

```
constexpr expected ()
```

Default-constructs the expected value.

Only available if *T* is default-constructible.

```
constexpr expected (expected const&)
constexpr expected (expected&&)

template<class ...Args>
expected (tl::in_place_t, Args&&...)

template<class U, class ...Args>
expected (tl::in_place_t, std::initializer_list<U>, Args&&...)
    Constructs the expected value in-place using the given arguments.

template<class G = E>
constexpr expected (unexpected<G> const &e)

template<class G = E>
constexpr expected (unexpected<G> &&e)
    Constructs the unexpected value.

    explicit if e is not implicitly convertible to E.

template<class ...Args>
expected (tl::unexpected_t, Args&&...)

template<class U, class ...Args>
expected (tl::unexpected_t, std::initializer_list<U>, Args&&...)
    Constructs the unexpected value in-place using the given arguments.

template<class U, class G>
constexpr expected (expected<U, G> const &rhs)

template<class U, class G>
constexpr expected (expected<U, G> &&rhs)
    Converting copy/move constructors.

    explicit if U and G are not implicitly convertible to T and E.

template<class U = T>
constexpr expected (U &&v)
    Constructs the expected value with the given value.

template<class U = T>
explicit expected &operator= (U &&v)
    If this* in the expected state, assigns v to the expected value. Otherwise destructs the unexpected value and constructs the expected value with v.

template<class G = E>
expected &operator= (tl::unexpected<G> const &e)

template<class G = E>
expected &operator= (tl::unexpected<G> &&e)
    If this* in in the unexpected state, assigns e to the unexpected value. Otherwise destructs the expected value and constructs the unexpected value with e.
```

### Standard Interface

This part of the interface is based on the proposed `std::expected`.

```
template<class ...Args>
void emplace (Args&&... args)

template<class U, class ...Args>
void emplace (std::initializer_list<U>, Args&&... args)
    If this* in in the expected state, assigns a T constructed in-place from args... to the expected value. Otherwise destructs the unexpected value and constructs the expected value in-place from args...
```



void **swap** (*expected* &*rhs*)

Swaps *\*this* with *rhs*.

*noexcept* if *T* and *E* are nothrow-swappable and -move-constructible.

**constexpr T \*operator->** ()

**constexpr T const \*operator->** () **const**

Returns a pointer to the expected value. Undefined behaviour if *\*this* is in the unexpected state.

Use `tl::expected::value()` for checked value retrieval.

**constexpr T &operator\*** () **&**

**constexpr T const &operator\*** () **const &**

**constexpr T &&operator\*** () **&&**

**constexpr T const &&operator\*** () **const &&**

Returns the expected value. Undefined behaviour if *\*this* is in the unexpected state. Use

`tl::expected::value()` for checked value retrieval.

**constexpr T &value** () **&**

**constexpr T const &value** () **const &**

**constexpr T &&value** () **&&**

**constexpr T const &&value** () **const &&**

If *\*this* is in the expected state, returns the expected value. Otherwise throws

`tl::bad_expected_access`.

**constexpr E &error** () **&**

**constexpr E const &error** () **const &**

**constexpr E &&error** () **&&**

**constexpr E const &&error** () **const &&**

If *\*this* is in the unexpected state, returns the unexpected value. Undefined behaviour if *\*this* is in the expected state. Use `tl::expected::has_value()` or

`tl::expected::operator bool()` to check the state before calling.

**constexpr bool has\_value** () **const noexcept**

**explicit constexpr operator bool** () **const noexcept**

Returns whether or not *\*this* is in the expected state.

template<class U>

**constexpr T value\_or** (U &&*u*) **const &**

template<class U>

**constexpr T value\_or** (U &&*u*) **&&**

If *\*this* is in the expected state, returns the expected value. Otherwise returns *u*.

## Extensions

These functions are all extensions to the proposed `std::expected`.

template<class F>

**constexpr auto and\_then** (F &&*f*) **&**

template<class F>

**constexpr auto and\_then** (F &&*f*) **const &**

template<class F>

**constexpr auto and\_then** (F &&*f*) **&&**

template<class F>

**constexpr auto and\_then** (F &&*f*) **const &&**

Used to compose functions which return a `tl::expected`. If *\*this* is in the expected state,

applies *f* to the expected value and returns the result. Otherwise returns *\*this* (i.e. the unexpected value bubbles up).

*Requires:* Calling the given function with the expected value must return a specialization of `tl::expected`.

```
template<class F>
constexpr auto map (F &&f) &
template<class F>
constexpr auto map (F &&f) const &
template<class F>
constexpr auto map (F &&f) &&
template<class F>
constexpr auto map (F &&f) const &&
template<class F>
constexpr auto transform (F &&f) &
template<class F>
constexpr auto transform (F &&f) const &
template<class F>
constexpr auto transform (F &&f) &&
template<class F>
constexpr auto transform (F &&f) const &&
```

Apply a function to change the expected value (and possibly the type). If *\*this* is in the expected state, applies *f* to the expected value and returns the result wrapped in a `tl::expected<ResultType, E>`. Otherwise returns *\*this* (i.e. the unexpected value bubbles up).

```
template<class F>
constexpr auto map_error (F &&f) &
template<class F>
constexpr auto map_error (F &&f) const &
template<class F>
constexpr auto map_error (F &&f) &&
template<class F>
constexpr auto map_error (F &&f) const &&
```

Apply a function to change the unexpected value (and possibly the type). If *\*this* is in the unexpected state, applies *f* to the unexpected value and returns the result wrapped in a `tl::expected<T, ResultType>`. Otherwise returns *\*this* (i.e. the expected value bubbles up).

```
template<class F>
expected<T, E> constexpr or_else (F &&f) &
template<class F>
expected<T, E> constexpr or_else (F &&f) const &
template<class F>
expected<T, E> constexpr or_else (F &&f) &&
template<class F>
expected<T, E> constexpr or_else (F &&f) const &&
```

If *\*this* is in the unexpected state, calls `f(this->error())` and returns the result. Otherwise returns *\*this*.

*Requires:* `std::invoke_result_t<F>` must be `void` or convertible to `tl::expected<T,E>`.

```
template<class T, class E, class U, class F>
constexpr bool operator==(expected<T, E> const &lhs, expected<U, F> const &rhs)
template<class T, class E, class U, class F>
```

**constexpr** bool **operator!=** (*expected*<*T*, *E*> const &*lhs*, *expected*<*U*, *F*> const &*rhs*)

Compare two `tl::expected` objects. They are considered equal if they are both in the same expected/unexpected state and their stored objects are equal.

template<class **T**, class **E**, class **U**>

**constexpr** bool **operator==** (*expected*<*T*, *E*> const &*e*, *U* const &*u*)

template<class **T**, class **E**, class **U**>

**constexpr** bool **operator!=** (*expected*<*T*, *E*> const &*e*, *U* const &*u*)

template<class **T**, class **E**, class **U**>

**constexpr** bool **operator==** (*U* const &*u*, *expected*<*T*, *E*> const &*e*)

template<class **T**, class **E**, class **U**>

**constexpr** bool **operator!=** (*U* const &*u*, *expected*<*T*, *E*> const &*e*)

Compare a `tl::expected` to an expected value. Only true if *e* stores has an expected value which is equal to *u*.

template<class **T**, class **E**>

**constexpr** bool **operator==** (*expected*<*T*, *E*> const &*e*, *tl::unexpected*<*E*> const &*u*)

template<class **T**, class **E**>

**constexpr** bool **operator!=** (*expected*<*T*, *E*> const &*e*, *tl::unexpected*<*E*> const &*u*)

template<class **T**, class **E**>

**constexpr** bool **operator==** (*tl::unexpected*<*E*> const &*u*, *expected*<*T*, *E*> const &*e*)

template<class **T**, class **E**>

**constexpr** bool **operator!=** (*tl::unexpected*<*E*> const &*u*, *expected*<*T*, *E*> const &*e*)

Compare a `tl::expected` to an unexpected value. Only true if *e* stores has an unexpected value which is equal to *u*.

template<class **T**, class **E**>

void **swap** (*tl::expected*<*T*, *E*> &*lhs*, *tl::expected*<*T*, *E*> &*rhs*)

Calls *lhs.swap(rhs)*.

*noexcept* if *lhs.swap(rhs)* is *noexcept*.

### *tl::unexpected*

template<class **E**>

**class** `tl::unexpected`

Used as a wrapper to store the unexpected value.

*E* must not be *void*.

**unexpected** () = delete

**explicit constexpr** `unexpected` (*E* const &*e*)

**explicit constexpr** `unexpected` (*E*&&)

Copies/moves the stored value.

**constexpr** *E* const &**value** () const &

**constexpr** *E* &**value** () &

**constexpr** *E* &&**value** () &&

**constexpr** *E* const &&**value** () const &&

Returns the contained value.

template<class **E**>

**constexpr** bool **operator==** (const `unexpected`<*E*> &*lhs*, const `unexpected`<*E*> &*rhs*)

template<class **E**>

**constexpr** bool **operator!=** (const `unexpected`<*E*> &*lhs*, const `unexpected`<*E*> &*rhs*)

```
template<class E>
constexpr bool operator< (const unexpected<E> &lhs, const unexpected<E> &rhs)
template<class E>
constexpr bool operator<= (const unexpected<E> &lhs, const unexpected<E> &rhs)
template<class E>
constexpr bool operator> (const unexpected<E> &lhs, const unexpected<E> &rhs)
template<class E>
constexpr bool operator>= (const unexpected<E> &lhs, const unexpected<E> &rhs)
    Compares two unexpected objects by comparing their stored value.
```

```
template<class E>
unexpected<std::decay_t<E>> tl::make_unexpected(E &&e)
    Create an unexpected from e, deducing the return type
```

Example:

```
auto e1 = tl::make_unexpected(42);
tl::unexpected<int> e2 (42); //same semantics
```

```
static constexpr unexpected_t tl::unexpected
    A tag to tell tl::expected to construct the unexpected value.
```

Example:

```
tl::expected<int, int> a(tl::unexpected, 42);
```

## Related Definitions

```
template<class E>
class bad_expected_access : public std::exception
    Thrown when checked value accesses fail, e.g.:
```

```
tl::expected<int, bool> a(tl::unexpected, false);
a.value(); //throws bad_expected_access<bool>
```

```
explicit bad_expected_access(E)
```

```
const char *what () const noexcept override
    Returns “Bad expected access”
```

```
static constexpr tl::in_place_t tl::in_place
    A tag to tell expected to construct its value in-place
```

## 2.1.2 function\_ref

Source code

An implementation of [function\\_ref](#).

```
template<class F>
class tl::function_ref
    A lightweight non-owning reference to a callable.
```

Example:

```

void foo (function_ref<int(int)> func) {
    std::cout << "Result is " << func(21); //42
}

foo([](int i) { return i*2; });

```

```

template<class R, class ...Args>
class tl::function_ref<R(Args...)>
    Specialization for function types.

```

### Special Members

**constexpr function\_ref () noexcept** = delete

**constexpr function\_ref (function\_ref const &rhs) noexcept**  
Creates a *tl::function\_ref* which refers to the same callable as *rhs*.

template<typename F>

**constexpr function\_ref (F &&f) noexcept**  
Creates a *tl::function\_ref* which refers to *f*.

*f* must be invocable with *Args...*, returning a type convertible to *R*.

**function\_ref &operator= (function\_ref const &rhs) noexcept**  
Makes *\*this* refer to the same callable as *rhs*.

template<typename F>

**constexpr function\_ref &operator= (F &&f) noexcept**  
Makes *\*this* refer to *f*.

*f* must be invocable with *Args...*, returning a type convertible to *R*.

**constexpr void swap (function\_ref &rhs) noexcept**  
Swaps the callables referred to by *\*this* and *rhs*.

**R operator () (Args... args) const**  
Invoke the stored callable with the given arguments.

```

template<typename R, typename ...Args>
constexpr void swap (function_ref<R> Args...
    > &lhs, function_ref<RArgs...> &rhs) noexcept Swaps the callables referred to by *this and rhs.

```

## 2.1.3 generator

### Source code

ranges-compatible generator type built on C++20 coroutines.

```

template<class T>

```

```

class tl::generator

```

A generator allows implementing sequence producers which are terse and avoid creating the whole sequence in memory.

Example:

```

tl::generator<int> iota(int i = 0) {
    while(true) {
        co_yield i;
        ++i;
    }
}

```

(continues on next page)

```

    }
}

```

### Member Types

```
class promise_type
```

```
class sentinel
```

```
class iterator
```

#### Member Types

```
using value_type = std::remove_reference_t<T>
```

```
using reference_type = value_type&
```

```
using pointer_type = value_type *
```

```
using difference_type = std::ptrdiff_t
```

#### Special Members

```
iterator () = delete
```

The iterators should only be created by `tl::generator<T>::begin`

```
iterator (iterator const&)
```

```
iterator &operator= (iterator const&)
```

Coroutine handles point to a unique resource, so the iterators are not copyable.

```
iterator (generator &&rhs) noexcept
```

```
iterator &operator= (generator &&rhs) noexcept
```

Takes the coroutine handle from `rhs`, making `rhs` not tied to a coroutine.

#### Member Functions

```
friend bool operator== (iterator const &it, sentinel) noexcept
```

Returns `true` if the iterator has been moved from, or if the coroutine it is tied to has completed.

```
iterator &operator++ ()
```

```
void operator++ (int)
```

Resumes the coroutine until `return/co_yield/exception` and returns an iterator which can be used to retrieve the yielded value and drive the coroutine forward again.

Rethrows the exception if one occurred.

```
reference_type operator* ()
```

Returns the last value yielded.

```
using promise_type = promise
```

```
using handle_type = std::coroutine_handle<promise_type>
```

#### Special Members

```
generator ()
```

Creates a generator which is not tied to a coroutine.

```
generator (generator const&) = delete
```

```
generator &operator= (generator const&) = delete
```

Coroutine handles point to a unique resource, so generators are not copyable.

```
generator (generator &&rhs) noexcept
```

*generator* &operator= (*generator* &&*rhs*) noexcept

Takes the coroutine handle from *rhs*, making *rhs* not tied to a coroutine.

### Member Functions

*iterator* begin ()

Resumes the coroutine until return/co\_yield/exception and returns an iterator which can be used to retrieve the yielded value and drive the coroutine forward again.

Rethrows the exception if one occurred.

Calling *begin* twice on the same generator is undefined behaviour.

*sentinel* end () const noexcept

void swap (*generator* &*other*) noexcept

```
template <class T> inline constexpr bool std::ranges::enable_view<tl::generator<T>> = true
tl::generator<T> is a view.
```

## 2.1.4 optional

Source code

C++11/14/17 *std::optional* with functional-style extensions and reference support.

### tl::optional

**class** tl::optional

An optional object is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

### Member Types

using value\_type = T

### Special Members

constexpr optional () noexcept

constexpr optional (tl::nullopt\_t) noexcept

Constructs an optional that does not contain a value.

constexpr optional (*optional* const &*rhs*)

constexpr optional (*optional* &&*rhs*)

Copy and move constructors. If *rhs* contains a value, the stored value is direct-initialized with it. Otherwise, the constructed optional is empty.

template<class ...Args>

explicit constexpr optional (tl::in\_place\_t, Args&&...)

template<class U, class ...Args>

explicit constexpr optional (tl::in\_place\_t, std::initializer\_list<U>, Args&&...)

Constructs the stored value in-place using the given arguments.

template<class U = T>

constexpr optional (U &&*u*)

Constructs the stored value with *u*.

*explicit* if *u* is convertible to *T*.

template<class **U**>  
**optional** (*optional*<**U**> **const** &*rhs*)

template<class **U**>  
**optional** (*optional*<**U**> &&*rhs*)

Converting copy and move constructors. If *rhs* contains a value, the stored value is direct-initialized with it. Otherwise, the constructed optional is empty.

*explicit* if the value of *rhs* is convertible to *T*.

*optional* &**operator**= (*nullopt\_t*) **noexcept**

Makes the optional empty, destroying the stored value if there is one.

*optional* &**operator**= (*optional* **const** &*rhs*)

*optional* &**operator**= (*optional* &&*rhs*)

Copy and move assignment operators. Copies/moves the value from *rhs* if there is one. Otherwise makes the optional empty, destroying the stored value if there is one.

template<class **U** = **T**>

*optional* &**operator**= (*U* &&*u*)

Assigns the stored value from *u*, destroying the old value if there was one.

template<class **U**>

*optional* &**operator**= (**const** *optional*<**U**> &*rhs*)

Converting copy/move assignment operators. Copies/moves the value from *rhs* if there is one. Otherwise makes the optional empty, destroying the stored value if there is one.

**~optional** ()

Destroys the stored value if there is one.

### Standard Optional Features

These features are all the same as *std::optional*.

template<class ...**Args**>

**T** &**emplace** (*Args*&&... *args*)

Constructs the value in-place, destroying the current one if there is one.

void **swap** (*optional* &*rhs*)

Swaps this optional with the other.

If neither optionals have a value, nothing happens. If both have a value, the values are swapped. If one has a value, it is moved to the other and the movee is left valueless.

*noexcept* if *T* is nothrow swappable and move constructible.

**constexpr** **T** \***operator**-> ()

**constexpr** **T** **const** \***operator**-> () **const**

Returns a pointer to the stored value. Undefined behaviour if there is no value. Use *tl::optional::value()* for checked value retrieval.

**constexpr** **T** &**operator**\* () &

**constexpr** **T** **const** &**operator**\* () **const** &

**constexpr** **T** &&**operator**\* () &&

**constexpr** **T** **const** &&**operator**\* () **const** &&

Returns the stored value. Undefined behaviour if there is no value. Use *tl::optional::value()* for checked value retrieval.

**constexpr** **T** &**value** () &

**constexpr** **T** **const** &**value** () **const** &

**constexpr** **T** &&**value** () &&



**constexpr T const &&value () const &&**

Returns the stored value if there is one, otherwise throws `tl::bad_optional_access`.

**constexpr bool has\_value () const noexcept**

**explicit constexpr operator bool () const noexcept**

Returns whether or not the optional has a value.

### Extensions

These features are all extensions to `std::optional`.

template<class **F**>

**constexpr auto and\_then (F &&f) &**

template<class **F**>

**constexpr auto and\_then (F &&f) const &**

template<class **F**>

**constexpr auto and\_then (F &&f) &&**

template<class **F**>

**constexpr auto and\_then (F &&f) const &&**

Used to compose functions which return a `tl::optional`. Applies *f* to the value stored in the optional and returns the result. If there is no stored value, then it returns an empty optional.

*Requires:* Calling the given function with the stored value must return a specialization of `tl::optional`.

template<class **F**>

**constexpr auto map (F &&f) &**

template<class **F**>

**constexpr auto map (F &&f) const &**

template<class **F**>

**constexpr auto map (F &&f) &&**

template<class **F**>

**constexpr auto map (F &&f) const &&**

template<class **F**>

**constexpr auto transform (F &&f) &**

template<class **F**>

**constexpr auto transform (F &&f) const &**

template<class **F**>

**constexpr auto transform (F &&f) &&**

template<class **F**>

**constexpr auto transform (F &&f) const &&**

Apply a function to change the value (and possibly the type) stored. Applies *f* to the value stored in the optional and returns the result wrapped in an optional. If there is no stored value, then it returns an empty optional.

template<class **F**>

*optional*<T> **constexpr or\_else (F &&f) &**

template<class **F**>

*optional*<T> **constexpr or\_else (F &&f) const &**

template<class **F**>

*optional*<T> **constexpr or\_else (F &&f) &&**

template<class **F**>

*optional*<T> **constexpr or\_else (F &&f) const &&**

Calls *f* if the optional is empty and returns the result. If the optional already has a value, returns *\*this*.

Requires: `std::invoke_result_t<F>` must be `void` or convertible to `tl::optional<T>`.

```
template<class F, class U>
U map_or (F &&f, U &&u) &
```

```
template<class F, class U>
U map_or (F &&f, U &&u) const &
```

```
template<class F, class U>
U map_or (F &&f, U &&u) &&
```

```
template<class F, class U>
U map_or (F &&f, U &&u) const &&
```

Maps the stored value with `f` if there is one, otherwise returns `u`.

```
template<class U>
constexpr optional<std::decay_t<U>> conjunction (U &&u) const
Returns u if *this has a value, otherwise an empty optional.
```

```
constexpr optional disjunction (const optional &rhs) &
constexpr optional disjunction (const optional &rhs) const &
constexpr optional disjunction (const optional &rhs) &&
constexpr optional disjunction (const optional &rhs) const &&
Returns rhs if *this is empty, otherwise the current value.
```

```
optional take ()
Takes the value out of the optional, leaving it empty
```

```
template<class T, class U>
constexpr bool operator==(tl::optional<T> const&, tl::optional<U> const&)
```

```
template<class T, class U>
constexpr bool operator!=(tl::optional<T> const&, tl::optional<U> const&)
```

```
template<class T, class U>
constexpr bool operator<(tl::optional<T> const&, tl::optional<U> const&)
```

```
template<class T, class U>
constexpr bool operator<=(tl::optional<T> const&, tl::optional<U> const&)
```

```
template<class T, class U>
constexpr bool operator>(tl::optional<T> const&, tl::optional<U> const&)
```

```
template<class T, class U>
constexpr bool operator>=(tl::optional<T> const&, tl::optional<U> const&)
```

If both optionals contain a value, they are compared with `T`'s relational operators. Otherwise `lhs` and `rhs` are equal only if they are both empty, and `lhs` is less than `rhs` only if `rhs` is empty and `lhs` is not.

```
template<class T>
constexpr bool operator==(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
constexpr bool operator!=(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
constexpr bool operator<(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
constexpr bool operator<=(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
constexpr bool operator>(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
constexpr bool operator>=(tl::optional<T> const&, tl::nullopt_t)
```

```
template<class T>
```

**constexpr** bool **operator==** (tl::nullopt\_t, tl::optional<T> const&)

template<class T>

**constexpr** bool **operator!=** (tl::nullopt\_t, tl::optional<T> const&)

template<class T>

**constexpr** bool **operator<** (tl::nullopt\_t, tl::optional<T> const&)

template<class T>

**constexpr** bool **operator<=** (tl::nullopt\_t, tl::optional<T> const&)

template<class T>

**constexpr** bool **operator>** (tl::nullopt\_t, tl::optional<T> const&)

template<class T>

**constexpr** bool **operator>=** (tl::nullopt\_t, tl::optional<T> const&)

Equivalent to comparing the optional to an empty optional

template<class T>

void **swap** (tl::optional<T> &lhs, tl::optional<T> &rhs)

Calls lhs.swap(rhs).

*noexcept* if lhs.swap(rhs) is *noexcept*

### tl::optional<T&>

template<class T>

**class** tl::optional<T&>

Specialization for when *T* is a reference. *optional<T&>* acts similarly to a *T\**, but provides more operations and shows intent more clearly.

*Examples:*

```
int i = 42;
tl::optional<int&> o = i;
*o == 42; //true
i = 12;
*o = 12; //true
&*o == &i; //true
```

Assignment has rebind semantics rather than assign-through semantics:

```
int j = 8;
o = j;

&*o == &j; //true
```

**using** value\_type = T&

#### Special Members

**constexpr** optional () *noexcept*

**constexpr** optional (tl::nullopt\_t) *noexcept*

Constructs an optional that does not contain a reference.

**constexpr** optional (optional const &rhs)

**constexpr** optional (optional &&rhs)

Copy and move constructors. If *rhs* contains a reference, makes the stored reference point at the same object. Otherwise, the constructed optional is empty.

template<class U = T>

**constexpr optional** (*U* &&*u*)

Makes the stored reference point at *u*.

*u* must be an lvalue.

template<class **U**>

**optional** (*optional*<*U*> **const** &*rhs*)

Converting copy constructor. If *rhs* contains a reference, makes the stored reference point at the same object. Otherwise, the constructed optional is empty.

*optional* &**operator=** (*nullopt\_t*) **noexcept**

Makes the optional empty.

*optional* &**operator=** (*optional* **const** &*rhs*)

Copy assignment operator. If *rhs* contains a reference, makes the stored reference point at the same object. Otherwise, the constructed optional is empty.

template<class **U** = *T*>

*optional* &**operator=** (*U* &&*u*)

Makes the stored reference point at the same object.

*u* must be an lvalue.

template<class **U**>

*optional* &**operator=** (**const** *optional*<*U*> &*rhs*)

Converting copy assignment operator. If *rhs* contains a reference, makes the stored reference point at the same object. Otherwise, the constructed optional is empty.

**~optional** ()

No-op

### Standard Optional Features

These features are modelled after those in *std::optional*.

void **swap** (*optional* &*rhs*) **noexcept**

Swaps this optional with the other.

If neither optionals have a reference, nothing happens. If both have a reference, the references are swapped.

If one has a reference, it is moved to the other and the movee is left referenceless.

**constexpr T \*operator->** ()

**constexpr T const \*operator->** () **const**

Returns a pointer to the stored value.

**constexpr T &operator\*** () &

**constexpr T const &operator\*** () **const** &

**constexpr T &&operator\*** () &&

**constexpr T const &&operator\*** () **const** &&

Returns the stored value. Undefined behaviour if there is no value. Use

*tl::optional*<*T*>::*value*() for checked value retrieval.

**constexpr T &value** () &

**constexpr T const &value** () **const** &

**constexpr T &&value** () &&

**constexpr T const &&value** () **const** &&

Returns the stored value if there is one, otherwise throws *tl::bad\_optional\_access*.

**constexpr bool has\_value** () **const noexcept**

**explicit constexpr operator bool () const noexcept**

Returns whether or not the optional has a value.

### Extensions

These features are all extensions to `std::optional`.

```
template<class F>
constexpr auto and_then (F &&f) &
template<class F>
constexpr auto and_then (F &&f) const &
template<class F>
constexpr auto and_then (F &&f) &&
```

```
template<class F>
constexpr auto and_then (F &&f) const &&
```

Used to compose functions which return a `tl::optional`. Applies `f` to the value stored in the optional and returns the result. If there is no stored value, then it returns an empty optional.

*Requires:* Calling the given function with the stored value must return a specialization of `tl::optional`.

```
template<class F>
constexpr auto map (F &&f) &
template<class F>
constexpr auto map (F &&f) const &
template<class F>
constexpr auto map (F &&f) &&
template<class F>
constexpr auto map (F &&f) const &&
```

```
template<class F>
constexpr auto transform (F &&f) &
template<class F>
constexpr auto transform (F &&f) const &
template<class F>
constexpr auto transform (F &&f) &&
template<class F>
constexpr auto transform (F &&f) const &&
```

Apply a function to change the value (and possibly the type) stored. Applies `f` to the value stored in the optional and returns the result wrapped in an optional. If there is no stored value, then it returns an empty optional.

```
template<class F>
optional<T> constexpr or_else (F &&f) &
template<class F>
optional<T> constexpr or_else (F &&f) const &
template<class F>
optional<T> constexpr or_else (F &&f) &&
template<class F>
optional<T> constexpr or_else (F &&f) const &&
```

Calls `f` if the optional is empty and returns the result. If the optional already has a value, returns `*this`.

*Requires:* `std::invoke_result_t<F>` must be `void` or convertible to `tl::optional<T>`.

```
template<class F, class U>
U map_or (F &&f, U &&u) &
```

```

template<class F, class U>
U map_or (F &&f, U &&u) const &
template<class F, class U>
U map_or (F &&f, U &&u) &&
template<class F, class U>
U map_or (F &&f, U &&u) const &&
    Maps the stored value with f if there is one, otherwise returns u.

template<class U>
constexpr optional<std::decay_t<U>> conjunction (U &&u) const
    Returns u if *this has a value, otherwise an empty optional.

constexpr optional disjunction (const optional &rhs) &
constexpr optional disjunction (const optional &rhs) const &
constexpr optional disjunction (const optional &rhs) &&
constexpr optional disjunction (const optional &rhs) const &&
    Returns rhs if *this is empty, otherwise the current value.

optional take ()
    Takes the reference out of the optional, leaving it empty

```

## Related Definitions

```

template<class T>
class std::hash<tl::optional<T>>

    std::size_t operator () (tl::optional<T> const&) const
        Returns the hash of the stored value if one exists. Otherwise returns 0.

class tl::monostate
    Used to represent an optional with no data; essentially a bool

static constexpr tl::in_place_t tl::in_place
    A tag to tell optional to construct its value in-place

static constexpr tl::nullopt_t tl::nullopt
    Represents an empty optional

```

*Examples:*

```

tl::optional<int> a = tl::nullopt;
void foo (tl::optional<int>);
foo(tl::nullopt); //pass an empty optional

```

```

class tl::nullopt_t
class tl::bad_optional_access : public std::exception

```

## 2.1.5 ranges

[Source code](#)

Implementations of ranges that didn't make C++20.

***tl::enumerate\_view***/***tl::views::enumerate***

```
template <class V> class tl::enumerate_view : public std::ranges::view_interface<enumerate_view<V>>
    Requires: std::ranges::input_range<V> && std::ranges::view<V>
```

A view which lets you iterate over the items in a range and their indices at the same time.

Examples:

```
std::vector<int> v;
for (auto&& [index, item] : tl::views::enumerate(v)) {
    //...
}
for (auto&& [index, item] : tl::enumerate_view(v)) {
    //...
}
for (auto&& [index, item] : v | tl::views::enumerate) {
    //...
}
```

**Member Types**

```
template<bool Const>
```

```
class sentinel
```

**Member Types**

```
private using Base = std::conditional_t<Const, const V, V>
```

**Special Members**

```
sentinel () = default
```

```
explicit constexpr sentinel (std::ranges::sentinel_t<Base> end)
```

```
constexpr sentinel (sentinel<!Const> other)
```

```
    Requires: Const && std::convertible_to<std::ranges::sentinel_t<V>,
std::ranges::sentinel_t<Base>>
```

Const-converting constructor.

**Member Functions**

```
constexpr std::ranges::sentinel_t<Base> base () const
```

Retrieve the base sentinel.

**Friend Functions**

```
friend constexpr bool operator==(const iterator<Const> &x, const sentinel &y)
```

```
friend constexpr std::ranges::range_difference_t<Base> operator- (const iterator<Const> &x,
                                                                    const sentinel &y)
```

```
friend constexpr std::ranges::range_difference_t<Base> operator- (const sentinel &x, const iterator<Const> &y)
```

```
    Requires: std::sized_sentinel_for<std::ranges::sentinel_t<Base>, std::ranges::iterator_t<Base>>
```

```
template<bool Const>
```

```
class iterator
```

**Member Types**

```
private using Base = std::conditional_t<Const, const V, V>
```

```
private count_type = std::conditional_t<std::ranges::sized_range<V>, std::ranges::...
```

```

using iterator_category = typename std::iterator_traits<std::ranges::iterator_t<Base>>::iterator_category
reference = std::pair<count_type, std::ranges::range_reference_t<Base>>;
using value_type = std::pair<count_type, std::ranges::range_value_t<Base>>
using difference_type = std::ranges::range_difference_t<Base>

```

### Special Members

**iterator** () = default

Construct an iterator not tied to a base iterator.

**explicit constexpr iterator** (std::ranges::iterator\_t<Base> *current*, *difference\_type* *pos*)

Construct an iterator wrapping the given base iterator at the given index.

**constexpr iterator** (*iterator*<!Const> *i*)

**Requires:** *Const* && *std::convertible\_to<std::ranges::iterator\_t<V>*,  
*std::ranges::iterator\_t<Base>>*

Const-converting constructor.

### Member Functions

**constexpr** std::ranges::iterator\_t<Base> **base** () **const** &

**Requires:** *std::copyable<std::ranges::iterator\_t<Base>>*

Retrieve the base iterator.

**constexpr** std::ranges::iterator\_t<Base> **base** () &&

Retrieve the base iterator.

**constexpr** reference **operator\*** () **const**

Retrieve a pair of the current index and element.

**constexpr** reference **operator[]** (*difference\_type* *x*) **const**

**Requires:** *std::ranges::random\_access\_range<Base>*

Retrieve the pair of the index and element at an offset of *x*.

**constexpr** *iterator* &**operator++** ()

**constexpr** void **operator++** (int)

**Requires:** *!std::ranges::forward\_range<Base>*

*iterator* **operator++** (int)

**Requires:** *std::ranges::forward\_range<Base>*

**constexpr** *iterator* &**operator--** ()

**constexpr** *iterator* **operator--** (int)

**Requires:** *std::ranges::bidirectional\_range<Base>*

**constexpr** *iterator* &**operator+=** (*difference\_type* *x*)

**constexpr** *iterator* &**operator-=** (*difference\_type* *x*)

**Requires:** *std::ranges::random\_access\_range<Base>*

### Friend Functions

**friend constexpr** bool **operator==** (const *iterator* &*x*, const *iterator* &*y*)

**Requires:** *std::equality\_comparable<std::ranges::iterator\_t<Base>>*

**friend constexpr** bool **operator<** (const *iterator* &*x*, const *iterator* &*y*)

**friend constexpr** bool **operator>** (const *iterator* &*x*, const *iterator* &*y*)

**friend constexpr** bool **operator<=** (const *iterator* &*x*, const *iterator* &*y*)



```
friend constexpr bool operator>= (const iterator &x, const iterator &y)
```

**Requires:** `std::ranges::random_access_range<Base>`

```
friend constexpr auto operator<=>(const iterator& x, const iterator& y)
```

**Requires:** `std::ranges::random_access_range<Base> && std::three_way_comparable<std::ranges::iterator_t<Base>`

```
friend constexpr iterator operator+ (const iterator &x, difference_type y)
```

```
friend constexpr iterator operator+ (difference_type x, const iterator &y)
```

```
friend constexpr iterator operator- (const iterator &x, difference_type y)
```

```
friend constexpr iterator operator- (difference_type x, const iterator &y)
```

**Requires:** `std::ranges::random_access_range<Base>`

### Special Members

```
enumerate_view () = default
```

```
enumerate_view (V base)
```

### Member Functions

```
constexpr iterator<false> begin ()
```

**Requires:** `!simple_view<V>`

```
constexpr iterator<true> begin () const
```

**Requires:** `simple_view<V>`

```
constexpr sentinel<false> end ()
```

```
constexpr iterator<false> end ()
```

**Requires:** `std::ranges::common_range<V> && std::ranges::sized_range<V>`

```
constexpr sentinel<true> end () const
```

```
constexpr iterator<true> end () const
```

**Requires:** `std::ranges::common_range<V> && std::ranges::sized_range<V>`

```
constexpr std::ranges::size_t<V> size ()
```

**Requires:** `std::ranges::sized_range<V>`

```
constexpr std::ranges::size_t<const V> size ()
```

**Requires:** `std::ranges::sized_range<const V>`

```
constexpr V base () const &
```

**Requires:** `std::copy_constructible<V>`

```
constexpr V base () &&
```

```
class tl::views::detail::enumerate_fn
```

```
template<class V>
```

```
constexpr tl::enumerate_view<deduced> operator () (V &&v) const
```

```
template<class V>
```

```
friend constexpr auto operator | (V &&v, enumerate_fn)
```

**Requires:** `std::ranges::viewable_range<V>`

Create a `tl::enumerate_view` from the given range.

```
constexpr inline tl::views::detail::enumerate_fn enumerate
```

Instance of `tl::views::detail::enumerate_fn` to allow piping, e.g.:

```
for (auto&& [index, item] : v | tl::views::enumerate) {  
    //...  
}
```

## 2.1.6 Miscellaneous Utilities

### apply

[Source code](#)

A C++11 implementation of C++17's `std::apply`.

```
template<class F, class Tuple>  
auto tl::apply (F &&f, Tuple &&tuple)  
    Calls f with the contents of tuple as arguments.
```

Equivalent to:

```
f (std::get<0>(tuple), std::get<1>(tuple), /*...*/ std::get<N>(tuple));
```

SFINAE-friendly.

*noexcept* if the call to *f* is *noexcept*.

### casts

[Source code](#)

A handful of handy casts.

```
template<class To, class From>  
To tl::bit_cast (From const &from)  
    Casts the bit representation of from to a To. Use this instead of type punning through a union or reinterpret_cast.  
    Essentially does:
```

```
To to;  
std::memcpy(&to, &from, sizeof(to));
```

```
template<class E>  
auto underlying_cast (E e)  
    Casts an enumerator value to its underlying type.  
  
    SFINAE-friendly.
```

### decay\_copy

[Source code](#)

An implementation of `decay_copy`.

```
template<class T>  
std::decay_t<T> tl::decay_copy (T &&t)  
    Creates a copy of t, decaying the type. Used to produce an rvalue from some expression without accidentally moving lvalues.
```

## dependent\_false

Source code

A `static_assert` helper.

## integer\_sequence

Source code

An C++11 implementation of C++14's `compile-time integer sequences`, along with some utilities.

These constructs are typically used to implement the `indices` trick

```
template<class T, std::size_t N>
using tl::make_integer_sequence = magic
    Creates a tl::integer_sequence from 0 to  $N-1$ .
```

Example:

```
tl::make_integer_sequence<int, 3>; //tl::integer_sequence<int, 0, 1, 2>
```

```
template<std::size_t... Idx>
using tl::index_sequence = integer_sequence<std::size_t, Idx...>
    Alias for integer sequences of type std::size_t, which comes up a lot with utilities like std::get.
```

```
template<std::size_t N>
using tl::make_index_sequence = make_integer_sequence<std::size_t, N>
    Alias for making an integer sequence of std::size_t s.
```

```
template<class ...Ts>
using tl::index_sequence_for = make_index_sequence<sizeof(Ts...)>
    Make an index sequence to index a parameter pack.
```

```
template<std::size_t From, std::size_t N>
using tl::make_index_range = magic
    Make in index sequence spanning the specified range.
```

Example:

```
make_index_range<3, 2>; //tl::index_sequence<3, 4>
make_index_range<5, 4>; //tl::index_sequence<5, 6, 7, 8>
```

## make\_array

Source code

Basic implementation of `make_array`

```
template<class ...Ts>
constexpr std::array<std::decay_t<std::common_type_t<Ts...>>, sizeof...(Ts)> tl::make_array (Ts&&...
    Create a std::array from the given arguments, deducing the type and size of the array.
    ts)
```

Example:

```
tl::make_array(0, 1, 2, 3); // std::array<int, 4>{0, 1, 2, 3}
```

### numeric\_aliases

Source code

Rust-style numeric aliases.

```
using i8 = std::int8_t
using i16 = std::int16_t
using i32 = std::int32_t
using i64 = std::int64_t
using u8 = std::uint8_t
using u16 = std::uint16_t
using u32 = std::uint32_t
using u64 = std::uint64_t
using uchar = unsigned char
using ushort = unsigned short
using uint = unsigned int
using ulong = unsigned long
using ullong = unsigned long long
using llong = long long
using ldouble = long double
using usize = std::size_t
```

### overload

Source code

A rudimentary implementation of `overload`.

Used for in-place visitation of `std::variant`:

```
variant<int, float, std::string> my_variant;
auto do_something = overload(
    [](int i) { /* do something with int */ },
    [](float f) { /* do something with float */ },
    [](std::string s) { /* do something with string */ }
);
visit(do_something, my_variant);
```

```
template<class ...Fs>
```

```
auto tl::overload (Fs&&... fs)
```

Create a single function object with a call operator overloaded by all function objects in `fs...`

### type\_traits

Source code

Implementations of some type traits from C++17 and Lib Fundamentals v2 TS, and C++14-style type aliases for C++11.

```
template<bool B>
using tl::bool_constant = std::integral_constant<bool, B>

template<std::size_t N>
using tl::index_constant = std::integral_constant<std::size_t, N>

template<class...>
using tl::void_t = void
    Turns any number of types into void.

    This is useful as a metaprogramming trick.
```

```
template<template<class...> class Trait, class ...Args>
using tl::is_detected = magic
    Checks if Trait<Args...> is a valid specialization. Implements std::experimental::is\_detected from Lib Fundamentals V2 TS. Useful for detecting the presence of a given type member. Example:
```

```
template<class T>
using make_cute_t = decltype(std::declval<T>().make_cute());

template<class T>
using can_make_cute = tl::is_detected<make_cute_t, T>;

struct cat{ void make_cute(); };
struct abstract_concept_of_fear{};

static_assert(can_make_cute<cat>::value);
static_assert(!can_make_cute<abstract_concept_of_fear>::value);
```

```
template<typename T>
using tl::safe_underlying_type_t = magic
    Like std::underlying\_type, but if T is not an enum then there's just no type member rather than being UB. Essentially implements P0340.
```

## C++14-style alias templates

```
template<class T>
using tl::remove_cv_t = typename std::remove_cv<T>::type

template<class T>
using tl::remove_const_t = typename std::remove_const<T>::type

template<class T>
using tl::remove_volatile_t = typename std::remove_volatile<T>::type

template<class T>
using tl::add_cv_t = typename std::add_cv<T>::type

template<class T>
using tl::add_const_t = typename std::add_const<T>::type

template<class T>
using tl::add_volatile_t = typename std::add_volatile<T>::type

template<class T>
using tl::remove_reference_t = typename std::remove_reference<T>::type
```

```
template<class T>
using tl::add_lvalue_reference_t = typename std::add_lvalue_reference<T>::type

template<class T>
using tl::add_rvalue_reference_t = typename std::add_rvalue_reference<T>::type

template<class T>
using tl::remove_pointer_t = typename std::remove_pointer<T>::type

template<class T>
using tl::add_pointer_t = typename std::add_pointer<T>::type

template<class T>
using tl::make_signed_t = typename std::make_signed<T>::type

template<class T>
using tl::make_unsigned_t = typename std::make_unsigned<T>::type

template<class T>
using tl::remove_extent_t = typename std::remove_extent<T>::type

template<class T>
using tl::remove_all_extents_t = typename std::remove_all_extents<T>::type

template<std::size_t N, std::size_t AN>
using tl::aligned_storage_t = typename std::aligned_storage<N, A>::type

template<std::size_t N, class ...Ts>
using tl::aligned_union_t = typename std::aligned_union<N, Ts...>::type

template<class T>
using tl::decay_t = typename std::decay<T>::type

template<bool E, class Tvoid>
using tl::enable_if_t = typename std::enable_if<E, T>::type

template<bool B, class T, class F>
using tl::conditional_t = typename std::conditional<B, T, F>::type

template<class ...Ts>
using tl::common_type_t = typename std::common_type<Ts...>::type

template<class T>
using tl::underlying_type_t = typename std::underlying_type<T>::type

template<class T>
using tl::result_of_t = typename std::result_of<T>::type
```

### typelist

[Source code](#)

Utilities to manipulate typelists.

## CHAPTER 3

---

### Getting the Code

---

*tl* is split up over a handful of repos, where the large components are in their own repos for easy inclusion and discovery:

- optional
- expected
- function\_ref
- generator
- ranges
- Miscellaneous Utilities





## CHAPTER 4

---

### License

---

CC0:

To the extent possible under law, Sy Brand has waived all copyright and related or neighboring rights to the tl libraries.  
This work is published from: United Kingdom.



**B**

bad\_expected\_access (*C++ class*), 8  
 bad\_expected\_access::bad\_expected\_access  
   (*C++ function*), 8  
 bad\_expected\_access::what (*C++ function*), 8

**E**

expected (*C++ class*), 3  
 expected::and\_then (*C++ function*), 5  
 expected::emplace (*C++ function*), 4  
 expected::error (*C++ function*), 5  
 expected::error\_type (*C++ type*), 3  
 expected::expected (*C++ function*), 3, 4  
 expected::has\_value (*C++ function*), 5  
 expected::map (*C++ function*), 6  
 expected::map\_error (*C++ function*), 6  
 expected::operator bool (*C++ function*), 5  
 expected::operator\* (*C++ function*), 5  
 expected::operator-> (*C++ function*), 5  
 expected::operator= (*C++ function*), 4  
 expected::or\_else (*C++ function*), 6  
 expected::swap (*C++ function*), 4  
 expected::transform (*C++ function*), 6  
 expected::unexpected\_type (*C++ type*), 3  
 expected::value (*C++ function*), 5  
 expected::value\_or (*C++ function*), 5  
 expected::value\_type (*C++ type*), 3

**I**

i16 (*C++ type*), 24  
 i32 (*C++ type*), 24  
 i64 (*C++ type*), 24  
 i8 (*C++ type*), 24

**L**

ldouble (*C++ type*), 24  
 llong (*C++ type*), 24

**O**

operator!= (*C++ function*), 6, 7, 14

operator== (*C++ function*), 6, 7, 14  
 operator> (*C++ function*), 7, 14  
 operator>= (*C++ function*), 7, 14  
 operator< (*C++ function*), 7, 14  
 operator<= (*C++ function*), 7, 14

**P**

PhonyNameDueToError::base (*C++ function*), 21  
 PhonyNameDueToError::begin (*C++ function*),  
   21  
 PhonyNameDueToError::end (*C++ function*), 21  
 PhonyNameDueToError::enumerate\_view  
   (*C++ function*), 21  
 PhonyNameDueToError::iterator (*C++ class*),  
   19  
 PhonyNameDueToError::iterator::base  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::Base  
   (*C++ type*), 19  
 PhonyNameDueToError::iterator::difference\_type  
   (*C++ type*), 20  
 PhonyNameDueToError::iterator::iterator  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::iterator\_category  
   (*C++ type*), 19  
 PhonyNameDueToError::iterator::operator\*  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::operator+  
   (*C++ function*), 21  
 PhonyNameDueToError::iterator::operator++  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::operator+=  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::operator-  
   (*C++ function*), 21  
 PhonyNameDueToError::iterator::operator-  
   (*C++ function*), 20  
 PhonyNameDueToError::iterator::operator--  
   (*C++ function*), 20

PhonyNameDueToError::iterator::operator=tl::function\_ref<R (Args...)>::function\_ref  
 (C++ function), 20 (C++ function), 9

PhonyNameDueToError::iterator::operator>tl::function\_ref<R (Args...)>::operator()  
 (C++ function), 20 (C++ function), 9

PhonyNameDueToError::iterator::operator>tl::function\_ref<R (Args...)>::operator=  
 (C++ function), 20 (C++ function), 9

PhonyNameDueToError::iterator::operator<tl::function\_ref<R (Args...)>::swap (C++  
 (C++ function), 20 function), 9

PhonyNameDueToError::iterator::operator<tl::generator (C++ class), 9  
 (C++ function), 20 tl::generator::begin (C++ function), 11

PhonyNameDueToError::iterator::operator[tl::generator::end (C++ function), 11  
 (C++ function), 20 tl::generator::generator (C++ function), 10

PhonyNameDueToError::iterator::value\_type tl::generator::handle\_type (C++ type), 10  
 (C++ type), 20 tl::generator::iterator (C++ class), 10

PhonyNameDueToError::sentinel (C++ class), tl::generator::iterator::difference\_type  
 19 (C++ type), 10

PhonyNameDueToError::sentinel::base tl::generator::iterator::iterator (C++  
 (C++ function), 19 function), 10

PhonyNameDueToError::sentinel::Base tl::generator::iterator::operator\* (C++  
 (C++ type), 19 function), 10

PhonyNameDueToError::sentinel::operator-tl::generator::iterator::operator++  
 (C++ function), 19 (C++ function), 10

PhonyNameDueToError::sentinel::operator=tl::generator::iterator::operator= (C++  
 (C++ function), 19 function), 10

PhonyNameDueToError::sentinel::sentinel tl::generator::iterator::operator==  
 (C++ function), 19 (C++ function), 10

PhonyNameDueToError::size (C++ function), 21 tl::generator::iterator::pointer\_type  
 (C++ type), 10

tl::generator::iterator::reference\_type  
 (C++ type), 10

tl::generator::iterator::value\_type  
 (C++ type), 10

tl::generator::operator= (C++ function), 10

tl::generator::promise\_type (C++ class), 10

tl::generator::promise\_type (C++ type), 10

tl::generator::sentinel (C++ class), 10

tl::generator::swap (C++ function), 11

tl::in\_place (C++ member), 8, 18

tl::index\_constant (C++ type), 25

tl::index\_sequence (C++ type), 23

tl::index\_sequence\_for (C++ type), 23

tl::is\_detected (C++ type), 25

tl::make\_array (C++ function), 23

tl::make\_index\_range (C++ type), 23

tl::make\_index\_sequence (C++ type), 23

tl::make\_integer\_sequence (C++ type), 23

tl::make\_signed\_t (C++ type), 26

tl::make\_unexpected (C++ function), 8

tl::make\_unsigned\_t (C++ type), 26

tl::monostate (C++ class), 18

tl::nullopt (C++ member), 18

tl::nullopt\_t (C++ class), 18

tl::optional (C++ class), 11

tl::optional::~~optional (C++ function), 12

**S**

std::hash<tl::optional<T>> (C++ class), 18

std::hash<tl::optional<T>>::operator()  
 (C++ function), 18

swap (C++ function), 7, 9, 15

**T**

tl::add\_const\_t (C++ type), 25

tl::add\_cv\_t (C++ type), 25

tl::add\_lvalue\_reference\_t (C++ type), 25

tl::add\_pointer\_t (C++ type), 26

tl::add\_rvalue\_reference\_t (C++ type), 26

tl::add\_volatile\_t (C++ type), 25

tl::aligned\_storage\_t (C++ type), 26

tl::aligned\_union\_t (C++ type), 26

tl::apply (C++ function), 22

tl::bad\_optional\_access (C++ class), 18

tl::bit\_cast (C++ function), 22

tl::bool\_constant (C++ type), 25

tl::common\_type\_t (C++ type), 26

tl::conditional\_t (C++ type), 26

tl::decay\_copy (C++ function), 22

tl::decay\_t (C++ type), 26

tl::enable\_if\_t (C++ type), 26

tl::function\_ref (C++ class), 8

tl::function\_ref<R (Args...)> (C++ class), 9

tl::optional::and\_then (C++ function), 13  
 tl::optional::conjunction (C++ function), 14  
 tl::optional::disjunction (C++ function), 14  
 tl::optional::emplace (C++ function), 12  
 tl::optional::has\_value (C++ function), 13  
 tl::optional::map (C++ function), 13  
 tl::optional::map\_or (C++ function), 14  
 tl::optional::operator bool (C++ function), 13  
 tl::optional::operator\* (C++ function), 12  
 tl::optional::operator-> (C++ function), 12  
 tl::optional::operator= (C++ function), 12  
 tl::optional::optional (C++ function), 11  
 tl::optional::or\_else (C++ function), 13  
 tl::optional::swap (C++ function), 12  
 tl::optional::take (C++ function), 14  
 tl::optional::transform (C++ function), 13  
 tl::optional::value (C++ function), 12  
 tl::optional::value\_type (C++ type), 11  
 tl::optional<T&> (C++ class), 15  
 tl::optional<T&>::~~optional (C++ function), 16  
 tl::optional<T&>::~and\_then (C++ function), 17  
 tl::optional<T&>::~conjunction (C++ function), 18  
 tl::optional<T&>::~disjunction (C++ function), 18  
 tl::optional<T&>::~has\_value (C++ function), 16  
 tl::optional<T&>::~map (C++ function), 17  
 tl::optional<T&>::~map\_or (C++ function), 17  
 tl::optional<T&>::~operator bool (C++ function), 16  
 tl::optional<T&>::~operator\* (C++ function), 16  
 tl::optional<T&>::~operator-> (C++ function), 16  
 tl::optional<T&>::~operator= (C++ function), 16  
 tl::optional<T&>::~optional (C++ function), 15, 16  
 tl::optional<T&>::~or\_else (C++ function), 17  
 tl::optional<T&>::~swap (C++ function), 16  
 tl::optional<T&>::~take (C++ function), 18  
 tl::optional<T&>::~transform (C++ function), 17  
 tl::optional<T&>::~value (C++ function), 16  
 tl::optional<T&>::~value\_type (C++ type), 15  
 tl::overload (C++ function), 24  
 tl::remove\_all\_extents\_t (C++ type), 26  
 tl::remove\_const\_t (C++ type), 25  
 tl::remove\_cv\_t (C++ type), 25  
 tl::remove\_extent\_t (C++ type), 26  
 tl::remove\_pointer\_t (C++ type), 26  
 tl::remove\_reference\_t (C++ type), 25  
 tl::remove\_volatile\_t (C++ type), 25  
 tl::result\_of\_t (C++ type), 26  
 tl::safe\_underlying\_type\_t (C++ type), 25  
 tl::underlying\_type\_t (C++ type), 26  
 tl::unexpected (C++ member), 8  
 tl::unexpected (C++ class), 7  
 tl::unexpected::unexpected (C++ function), 7  
 tl::unexpected::value (C++ function), 7  
 tl::views::detail::enumerate\_fn (C++ class), 21  
 tl::views::detail::enumerate\_fn::operator () (C++ function), 21  
 tl::views::detail::enumerate\_fn::operator| (C++ function), 21  
 tl::void\_t (C++ type), 25

## U

u16 (C++ type), 24  
 u32 (C++ type), 24  
 u64 (C++ type), 24  
 u8 (C++ type), 24  
 uchar (C++ type), 24  
 uint (C++ type), 24  
 ullong (C++ type), 24  
 ulong (C++ type), 24  
 underlying\_cast (C++ function), 22  
 ushort (C++ type), 24  
 usize (C++ type), 24