Continuation optional, expected, and future

John Franklin Rickard | john@moduleworks.com

Overview

- `std::optional`
- `std::expected`
- Monadic continuation methods
- The `std::future`?

Header optional

Added in C++17

- `optional` class itself
- `bad_optional_access` exception for checked access
- `nullopt` object of type `nullopt_t`
- `make_optional` for consistent inplace_construction
- operators for std::optional

cppreference

Why std::optional

```cpp
// Throws if string is not a valid number
int ParseInt(std::string_view str);

// get string as input
// parse input to number
try {
    return ParseInt(str);
}
catch(const std::exception& e) {
    // log, throw, whatever
    // notify user/caller this did not work
}
```

Why std::optional

```
// Returns true if string contains int.
// Assigns parsed number to given reference.
bool TryParseInt(std::string_view, int&);

// Throws if string is not a valid number
int ParseInt(std::string_view str) {
    int result = 0;
    bool success = TryParseInt(str, result);
    if (success)
        return result
    else
        throw ...;
}
```

Why std::optional

```cpp
// first is true on success
// second only valid on success
std::pair<bool, int> TryParseInt(std::string_view);

// Throws if string is not a valid number
int ParseInt(std::string_view str) {
    auto parseResult = TryParseInt(str);
    if (parseResult.first)
        return parseResult.second;
    else
        throw ...;
}
```

Why std::optional

```
template <class T>
struct optional {
    union {
        char m_null_state;
        T m_value;
    };
    bool m_engaged;

    explicit operator bool() const {
        return m_engaged;
    }
}
```

Why std::optional

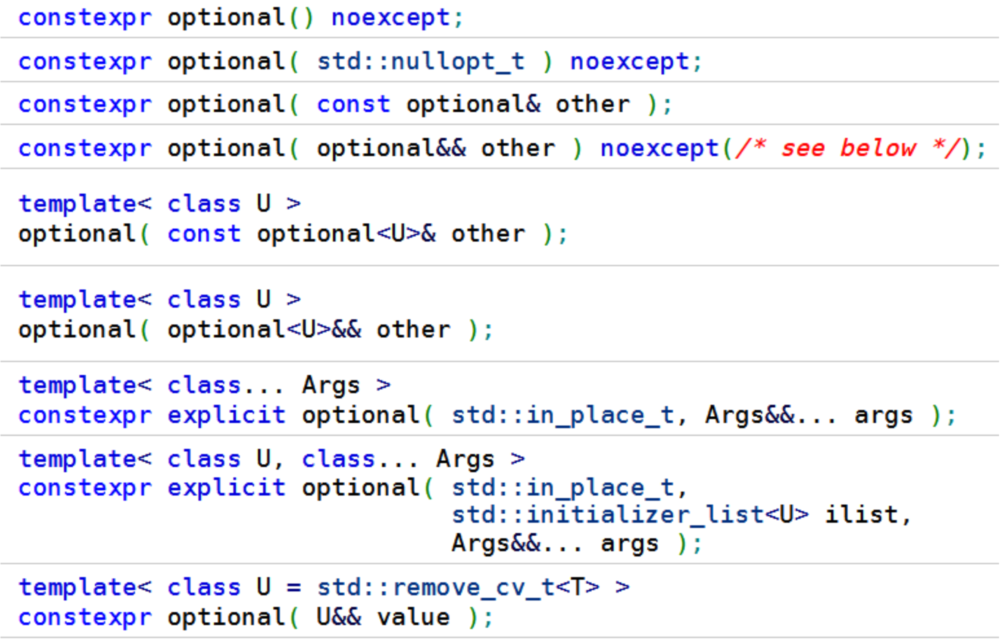
```cpp
std::optional<int> TryParseInt(std::string_view);

// Throws if string is not a valid number
int ParseInt(std::string_view str) {
    if (auto result = TryParseInt(str))
        return *result;
    else
        throw ...;
}
```

std::optional

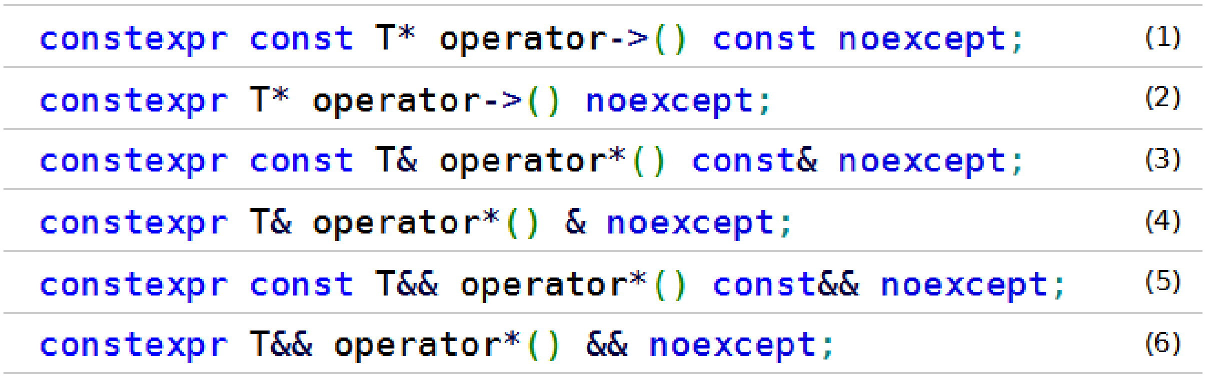
```cpp
auto opt = std::optional<Foo>();
opt = {};
opt.reset();
assert(opt.has_value() == false);

opt = std::optional<Foo>(...);
opt = Foo{...};
// in place construction methods
opt = std::optional<Foo>(std::in_place_t, ...);
opt.emplace(...);
opt = std::make_optional<Foo>(...);
assert(opt.has_value() == true);
```

std::optional

```
opt->ToString();
(*opt).ToString();

// throws bad_optional_access if empty
opt.value();

// easy default values
return GetOptional().value_or(...);

assert(opt.has_value() == true);
SinkFoo(*std::move(opt));
assert(opt.has_value() == true);
```



```
constexpr optional() noexcept;

constexpr optional( std::nullopt_t ) noexcept;

constexpr optional( const optional& other );

constexpr optional( optional&& other ) noexcept(/* see below */);

template< class U >
optional( const optional<U>& other );

template< class U >
optional( optional<U>&& other );

template< class... Args >
constexpr explicit optional( std::in_place_t, Args&&... args );

template< class U, class... Args >
constexpr explicit optional( std::in_place_t,
                             std::initializer_list<U> ilist,
                             Args&&... args );

template< class U = std::remove_cv_t<T> >
constexpr optional( U&& value );
```



```
constexpr const T* operator->() const noexcept;          (1)
constexpr T* operator->() noexcept;                      (2)
constexpr const T& operator*() const& noexcept;          (3)
constexpr T& operator*() & noexcept;                     (4)
constexpr const T&& operator*() const&& noexcept;        (5)
constexpr T&& operator*() && noexcept;                   (6)
```

std::nullopt

```cpp
// Same as default constructor
// Looks and feels similar as nullptr
std::optional a = std::nullopt;

void Dispatch(std::string); // 1
void Dispatch(std::optional<int>); // 2

Dispatch({}); // error, ambiguous
Dispatch(std::nullopt); // calls 2
```

std::optional comparisons

```cpp
std::optional<int>(5) < std::optional<int>(); // false
std::nullopt <= GetOptional(); // always true
std::optional<int>() > 5; // false
std::optional<int>(5) == 5; // true

// taken from libc++
bool operator==(const optional<_Tp>& __x,
                const optional<_Up>& __y) {
    if (static_cast<bool>(__x) != static_cast<bool>(__y))
        return false;
    if (!static_cast<bool>(__x))
        return true;
    return *__x == *__y;
}
```

Optional references

```
// Returns null if not cached
CacheData* GetCachedResult(int id);


auto cache = GetCachedResult(SearchID);
if (cache != nullptr)
    return *cache;
```

Optional references

```
std::optional<CacheData*> GetCachedResult(int id);

auto cache = GetCachedResult(SearchID);
if (cache) {
    if (*cache != nullptr)
        return *cache.value();
}
```

Optional references

- `There are no optional references, functions, arrays, or (possibly cv-qualified) void;`
- std::optional<T&> Paper
- Missed C++26 sadly

Header expected

Added in C++23

- `expected` class itself
- `unexpected` represents an unexpected value
- `bad_expected_access` exception for checked access
- `unexpect_t` in_place_t for unexpected values

cppreference

Why expected

```cpp
enum class parse_status {
    success,
    unknown_delimiter
};
struct parse_result {
    std::optional<int> number;
    parse_status status;
};
parse_result TryParseInt(std::string_view);
```

Why expected

```
template <class T, class Err>
struct expected {
    union {
        T m_value;
        Err m_error;
    };
    bool m_has_value;
}
```

Why expected

```cpp
template <class T, class Err>
    requires is_void_v<Ty>
struct expected<T, Err> {
    union {
        Err m_error;
    };
    bool m_has_value;
}
```

std::expected

```cpp
std::expected<int, parse_status> TryParseInt(std::string_vi

// Throws if string is not a valid number
int ParseInt(std::string_view str) {
    if (auto result = TryParseInt(str))
        return *result;
    else
        throw result.error();
}
```

std::expected

```
auto exp = std::expected<int, parse_status>();
assert(*exp == 0);

exp = parse_status::unknown_delimiter;
assert(exp.has_value() == false);

auto exp2 = std::expected<Foo, int>();
exp2->Calculate();
```

std::expected

```
auto exp = std::expected<int, parse_status>();

// throws bad_expected_access on error
exp.value();
// !! UB if it contains value !!
exp.error();

// easy default value
exp.value_or(5)

// default value for errors (?)
exp.error_or(parse_status::bla);
```

std::unexpected

```cpp
auto unexp = std::unexpected<int>(5);
assert(unexp.error() == 5);

auto exp = std::expected<int, int>(unexp);
assert(exp.has_value() == false);

// Does not compile
// unexp == 5;

// works as expected :)
exp.error() == unexp;
```

Expected references

From cppreference:

```
A program is ill-formed if it
instantiates an expected with a reference
type, a function type, or a
specialization of std::unexpected. In
addition, T must not be std::in_place_t
or std::unexpect_t.
```

Monadic operations

Monadic operations

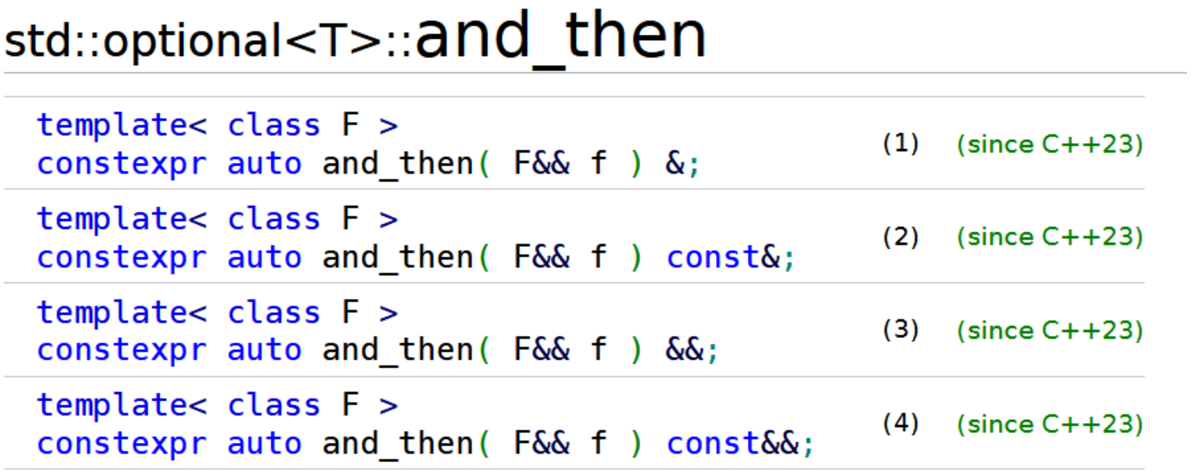
What is a Monad?

Monadic operations

Monadic operations

Monadic operations added in C++23 for optional and expected:

- `and_then`
- `transform`
- `or_else`
- `transform_error`



std::optional<T>::and_then

`template< class F >`
`constexpr auto and_then( F&& f ) &;`
	(1)	(since C++23)

`template< class F >`
`constexpr auto and_then( F&& f ) const&;`
	(2)	(since C++23)

`template< class F >`
`constexpr auto and_then( F&& f ) &&;`
	(3)	(since C++23)

`template< class F >`
`constexpr auto and_then( F&& f ) const&&;`
	(4)	(since C++23)

std::optional::and_then

```cpp
auto opt = std::optional<int>(5);

auto add2 =
    [](int i) -> std::optional<int> { return i + 2; };

assert(opt.and_then(add2) == 7);
assert(opt == 5);
assert(opt.and_then(add2).and_then(add2) == 9);

// add2 never called below
assert(std::optional<int>().and_then(add2) == std::nullopt
```

std::optional::and_then

```cpp
auto times2 =
    [](int i) -> std::optional<double> {
        return i * 2.0;
    };
auto toString =
    [](double d) -> std::optional<std::string> {
        return std::to_string(d);
    };

auto str = std::optional<int>(2)
    .and_then(times2)
    .and_then(toString)
    .value_or("6");

assert(str == "4");
```

std::expected::and_then

```cpp
auto times2 =
    [](int i) -> std::expected<double, int> {
        return i * 2.0;
    };
auto toString =
    [](double d) -> std::expected<std::string, int> {
        return std::to_string(d);
    };

auto str = std::expected<int, int>(std::unexpected(5))
    .and_then(times2)
    .and_then(toString)
    .value_or("6");

assert(str == "6");
```

std::expected::and_then

```
auto turnToInt = [] -> std::expected<int, int> { return {7
                                                        
auto exp = std::expected<void, int>()
    .and_then(turnToInt);

assert(exp == 7);
```

std::optional::or_else

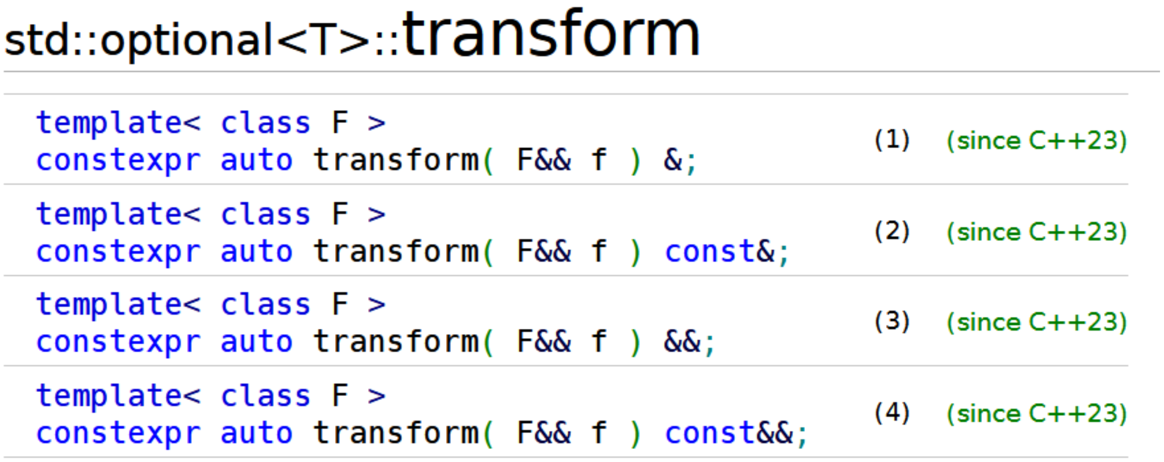
```cpp
auto opt = std::optional<int>(5);

auto answer = [] { return std::optional(42); };

assert(opt.or_else(answer) == 5);
assert(opt == 5);

auto neverWorks = [] { return std::nullopt; };
opt = std::nullopt;

assert(opt.or_else(neverWorks).or_else(answer) == 42);
```

std::expected::or_else

```cpp
auto exp = std::expected<double, int>(std::unexpected(10));

auto shrinkError =
    [] (int i) -> std::expected<double, short> {
        return std::unexpected(5);
    };

assert(exp.or_else(shrinkError) == std::unexpected(5));
assert(exp == std::unexpected(10));
```

std::expected::or_else

```
auto exp = std::expected<void, int>(std::unexpected(5));

auto into_the_void =
    [] (int i) -> std::expected<void, short> {
        return {};
    };

assert(exp.or_else(into_the_void).has_value());
```



std::optional<T>::transform

`template< class F >`
`constexpr auto transform( F&& f ) &;`
	(1)	(since C++23)

`template< class F >`
`constexpr auto transform( F&& f ) const&;`
	(2)	(since C++23)

`template< class F >`
`constexpr auto transform( F&& f ) &&;`
	(3)	(since C++23)

`template< class F >`
`constexpr auto transform( F&& f ) const&&;`
	(4)	(since C++23)

std::optional::transform

```cpp
auto times2 = [] (int i) { return i * 2.0; };
auto to_str = [] (double d) {return std::to_string(d); };

auto str = std::optional<int>(4)
    .transform(times2)
    .transform(to_str);

assert(str == "8");
```

transform: F maps T1 to T2 and boxes it in `std::optional<T2>`

and_then: F maps directly from T1 to `std::optional<T2>`

std::expected::transform

```
auto times2 = [] (int i) { return i * 2.0; };
auto to_str = [] (double d) {return std::to_string(d); };

auto str = std::expected<int, int>(4)
     .transform(times2)
     .transform(to_str);

assert(str == "8");
```

`transform`: F maps T1 to T2 and boxes it in `std::expected<T2, E>`

`and_then`: F maps directly from T1 to `std::expected<T2, E>`

std::expected::transform

```cpp
auto return2 = [] { return 2.0; };
auto to_str = [] (double d) {return std::to_string(d); };

auto str = std::expected<void, int>()
    .transform(return2)
    .transform(to_str);

assert(str == "2");
```

`transform`: F maps T1 to T2 and boxes it in `std::expected<T2, E>`

`and_then`: F maps directly from T1 to `std::expected<T2, E>`

std::expected::transform_error

```cpp
auto times2 = [] (int i) { return i * 2.0; };
auto to_str = [] (double d) {return std::to_string(d); };

auto exp = std::expected<int, int>(4)
    .transform_error(times2)
    .transform_error(to_str);

assert(exp == 4);
```

`transform_error`: F maps E1 to E2 and boxes it in `std::expected<T, E2>`

std::expected::transform_error

```cpp
auto times2 = [] (int i) { return i * 2.0; };
auto to_str = [] (double d) {return std::to_string(d); };

auto exp = std::expected<int, int>(std::unexpected(4))
    .transform_error(times2)
    .transform_error(to_str);

assert(exp.error() == "8");
```

`transform_error`: F maps E1 to E2 and boxes it in `std::expected<T, E2>`

Monadic operations

Monadic operations added in `C++23` for optional and expected:

- `and_then`, chain successful operations
- `transform`, change value inside
- `or_else`, react to failure with alternative
- `transform_error`, change error inside

Header future

- `promise`
- `packaged_task`
- `future`
- `shared_future`
- `async`

Why future

```
int ExpensiveOperation();

// object represents a value from the future :)
std::future<int> future =
    std::async(std::launch::async, ExpensiveOperation);

// ... do something else

// blocks until ExpensiveOperation finishes
return future.get();
```

Why future

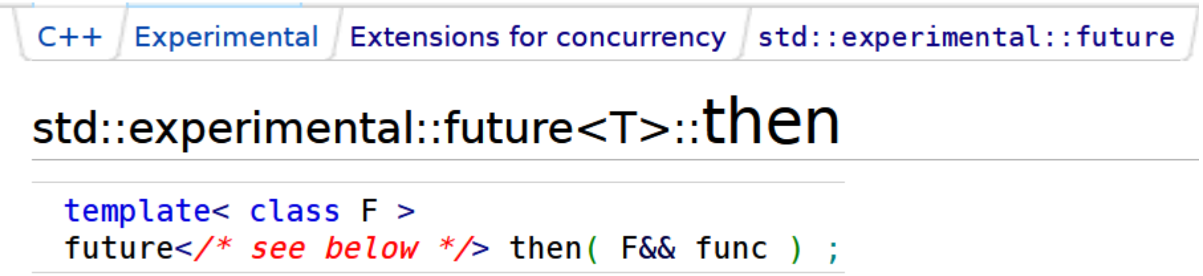
```cpp
auto future1 = std::future<int>(); // empty future
assert(future1.valid() == false);

future1 = std::async(std::launch::async, [] { return 8; });
assert(future1.valid() == true);

auto future2 = std::move(future1);
assert(future1.valid() == false);
assert(future2.valid() == true);
```

Why future

```cpp
auto future = std::async(std::launch::async, [] { return 8
using namespace std::chrono_literals;

// blocks until value available
future.wait();

// blocks at maximum the given duration
auto status = future.wait_for(1s);
// blocks at maximum until given timepoint
status = future.wait_until(
    std::chrono::system_clock::now() + 1s);

// check return status if value is avaliable or not
assert(status == std::future_status::ready);
```

Monadic operations future

Monadic operations

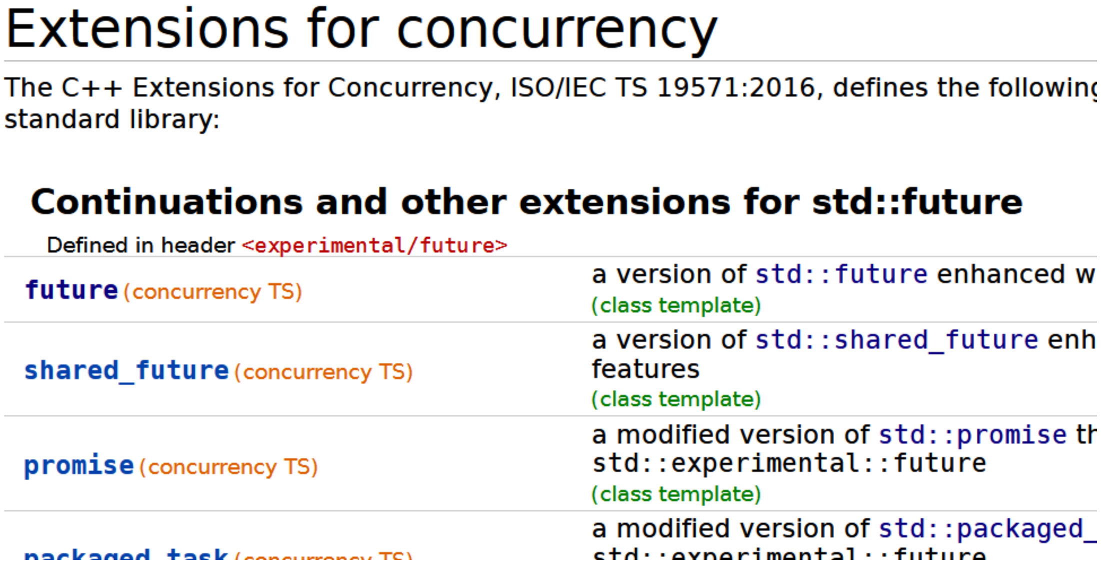


C++ / Experimental / Extensions for concurrency / std::experimental::future

std::experimental::future<T>::then

```
template< class F >
future</* see below */> then( F&& func ) ;
```

Monadic operations



Extensions for concurrency

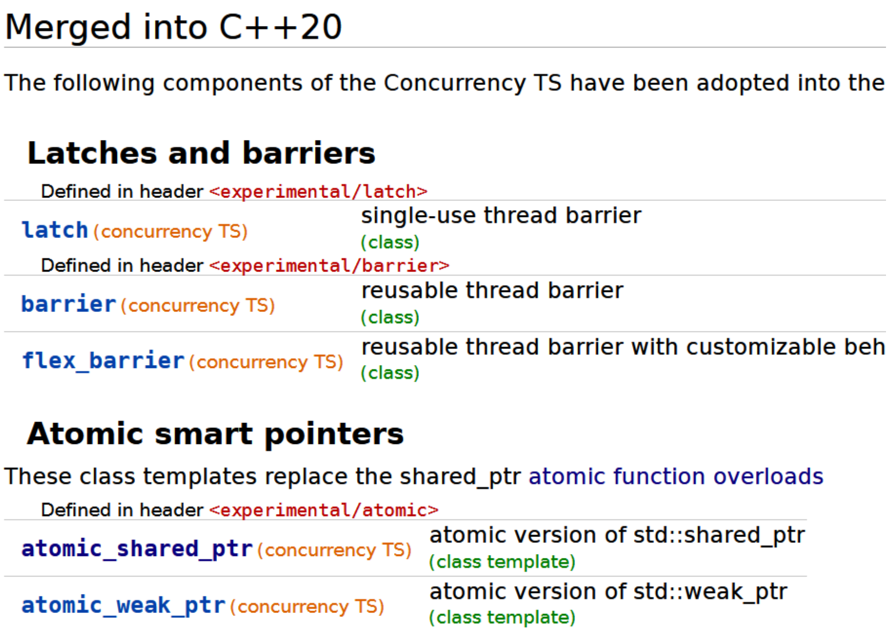
The C++ Extensions for Concurrency, ISO/IEC TS 19571:2016, defines the followin
standard library:

Continuations and other extensions for std::future

Defined in header <experimental/future>

future (concurrency TS)	a version of std::future enhanced w (class template)
shared_future (concurrency TS)	a version of std::shared_future enh features (class template)
promise (concurrency TS)	a modified version of std::promise th std::experimental::future (class template)
packaged_task (concurrency TS)	a modified version of std::packaged_ std::experimental::future

Monadic operations



Merged into C++20

The following components of the Concurrency TS have been adopted into the

Latches and barriers

Defined in header <experimental/latch>

latch (concurrency TS)	single-use thread barrier (class)

Defined in header <experimental/barrier>

barrier (concurrency TS)	reusable thread barrier (class)
flex_barrier (concurrency TS)	reusable thread barrier with customizable beh (class)

Atomic smart pointers

These class templates replace the shared_ptr atomic function overloads

Defined in header <experimental/atomic>

atomic_shared_ptr (concurrency TS)	atomic version of std::shared_ptr (class template)
atomic_weak_ptr (concurrency TS)	atomic version of std::weak_ptr (class template)

Bonus slide

std::optional is a view from C++26 onwards!

```cpp
auto opt = std::optional<int>();

for (const auto& i : opt)
{
    std::print("never runs, but isn't this cool?!");
}

auto vec = opt | std::ranges::to<std::vector>();
assert(vec.size() == 0);
```

Sadly not yet implemented by anywhere

Thank you for your attention!