Document Number: P2506R0
Date: 2022-02-02
Audience: Library Evolution Working Group
Author: Casey Carter <Casey@Carter.net>

`std::lazy`: a coroutine for deferred execution

Abstract

This paper proposes adding a new type, `std::lazy<T>`, to the standard library to enable creation and composition of coroutines representing asynchronous computation.

[*Note 1*: SG1 wants to reserve the name `std::task` for another use. SG1 requested a different name, and LEWG chose `lazy`. — *end note*]

1 Design

1.1 Motivation

The Coroutines TS introduced a language capability that allows functions to be suspended and later resumed. One of the key applications for this new feature is to make it easier to write asynchronous code. However, the TS did not include any concrete coroutine types to do so.

We believe it was a mistake to not include such a type in C++20, and it would be an even greater mistake not to include one in C++23. We further believe that the simple type presented here - which astute readers will notice has changed very little since P1056R1 [2]) - is not ideal, but is the minimal useful type we can provide without presuming support for executors.

As such, we do not provide any of the features desired for a lazy coroutine type to integrate with P2300's design for executors [4], despite having a list of requirements available. We think it best for WG21 to commit to having at least this minimal coroutine "task" support library in C++23, regardless of whether an executor proposal makes the C++23 train. There are coroutine users today who would benefit from a standardized coroutine task wrapper which they could incorporate into non-standard execution contexts even if WG21 doesn't ship executor support in C++23.

We plan to write a followup paper with design changes to integrate `lazy` with P2300 (in collaboration with the authors of P2300) so that C++23 has *some* type to represent coroutine tasks in any event, but the *best* type to represent coroutine tasks if we also get P2300 `std::execution`.

1.1.1 Acknowledgements

The authors would like to thank Lewis Baker and Gor Nishanov, whose proposal P1056R1 "Add lazy coroutine (coroutine task) type" [2] was looted shamelessly for this proposal. This paper is presented as a new revision of P1056 for continuity of design.

1.1.2 Example code

```
#include <lazy>
#include <string>

struct record {
  int id;
  std::string name;
  std::string description;
};

std::lazy<record> load_record(int id);
std::lazy<> save_record(record r);
```

```
std::lazy<void> modify_record() {
  record r = co_await load_record(123);
  r.description = "Look, ma, no blocking!";
  co_await save_record(std::move(r));
}
```

1.2 Proposal

The interface of lazy is intentionally minimal and designed for efficiency. In fact, the only operation you can do with the lazy is to await on it:

```
template<class T, class Allocator = void>
class [[nodiscard]] lazy {
public:
  lazy(lazy&& that) noexcept;

  ~lazy();

  unspecified operator co_await();

  T sync_await();
};
```

While such a small interface may seem unusual at first, subsequent sections will clarify the rationale for this design.

1.2.1 Why not use futures with `future.then`?

The `std::future` type is inherently inefficient and cannot be used for efficient composition of asynchronous operations. The unavoidable overhead of futures is due to:

— allocation/deallocation of the shared state object,

— atomic increment/decrement for managing the lifetime of the shared state object,

— synchronization between setting of the result and getting the result, and

— (with `.then`) scheduling overhead of starting execution of subscribers to `.then`.

Consider the following example:

```
lazy<int> coro() {
  int result = 0;
  while (int v = co_await async_read())
    result += v;
  co_return result;
}
```

where `async_read()` is some asynchronous operation that takes, say, 4ns to perform. We would like to factor out the logic into two coroutines:

```
lazy<int> subtask() {
  co_return co_await async_read();
}

lazy<int> coro_refactored() {
  int result = 0;
  while (int v = co_await subtask())
    result += v;
  co_return result;
}
```

Breaking a single `co_await` into its own function may seem silly, but it allows us to measure the overhead of composition of tasks. With the proposed `lazy`, our per-operation cost grew from 4ns to 6ns and did not incur any heap allocations. Moreover, this overhead of 2ns is not inherent to `lazy` and we anticipate that with improved coroutine optimization technology we will be able to drive the overhead close to zero. To estimate the cost of composition with `std::future`, we used the following code:

```
int fut_test() {
  int count = 1'000'000;
  int result = 0;
```

```
  while (count > 0) {
    promise p;
    auto fut = p.get_future();
    p.set_value(count--);
    result += fut.get();
  }
  return result;
}
```

As measured on the same system (Linux, clang 6.0, libc++), we get *133ns* per operation! Here is the visual illustration.

```
          op cost: **
   lazy overhead: *
future overhead: **********************************************************************
```

Being able to break apart bigger functions into a set of smaller ones and being able to compose software by putting together small pieces is fundamental requirement for a good software engineering since the 60s. The overhead of `std::future` and types similar in behavior makes them unsuitable as coroutine types.

1.2.1.1 Removing `future` overhead: Memory Allocation

Consider the only operation that is available on a `lazy`, namely, awaiting on it.

```
lazy<X> g();
lazy<Y> f() {
  // ...
  X x = co_await g();
  // ...
}
```

The caller coroutine `f` owns the lazy object for `g` that is created and destroyed at the end of the full expression containing `co_await`. This allows the compiler to determine the lifetime of the coroutine and apply Heap Allocation eLision Optimization [9], which allocates a coroutine's state as if it were a local variable in its caller.

1.2.1.2 Removing `future` overhead: Reference counting

The coroutine state is not shared. The `lazy` type only allows moving pointer to a coroutine from one `lazy` object to another. Lifetime of a coroutine is linked to its `lazy` object, and the `lazy` object's destructor destroys the coroutine, thus, no reference counting is required.

In a later section about cancellation we will cover the implications of this design decision.

1.2.1.3 Removing `future` overhead: Set/Get synchronization

The `lazy` coroutine always starts suspended. This allows not only to avoid synchronization when attaching a continuation, but also enables solving via composition how and where coroutine needs to get executed and allows to implement advanced execution strategies like continuation stealing.

1.2.1.4 Removing `future` overhead: Scheduling overhead

Consider the following code fragment:

```
int result = 0;
while (int v = co_await async_read())
  result += v;
```

Let's say that `async_read` returns a `future`. That `future` cannot resume directly the coroutine that is awaiting on it as it will, in effect, transform the loop into unbounded recursion.

On the other hand, coroutines have built-in support for symmetric coroutine transfer [7]. Since `lazy` object can only be created by a coroutine and the only way to get the result from a coroutine is by awaiting on it from another coroutine, the transfer of control from completing coroutine to awaiting coroutine is done in symmetric fashion, thus eliminating the need for extra scheduling interactions.

1.3 Destruction and cancellation

`lazy` destroys any associated coroutine in its destructor. It is safe to do, only if the coroutine has finished execution (at the final suspend point) or it is in a suspended state (waiting for some operation to complete), but, somehow, we know that the coroutine will never be resumed by the entity which was supposed to

resume the coroutine on behalf of the operation that coroutine is awaiting upon. That is only possible if the underlying asynchronous facility support cancellation.

We strongly believe that support for cancellation is a required facility for writing asynchronous code and we struggled for awhile trying to decide what is the source of the cancellation, whether it is the `lazy` that must initiate cancellation - and therefore every await in every coroutine needs to understand how to cancel a particular operation it is being awaited upon - or every async operation is tied to a particular lifetime and cancellation domain and operations are cancelled in bulk by cancellation of the entire cancellation domain [6].

We experimented with both approaches and reached the conclusion that not performing cancellation from the `lazy`, but, pushing it to the cancellation domain leads to more efficient implementation and is a simpler model for users.

1.4 Why no move assignment?

This is rather unorthodox decision and even authors of the paper did not completely agree on this topic. However, going with more restrictive model initially allows us to discover if the insight that lead to this decision was wrong. Initial design of the lazy, included move assignment, default constructor and swap. We removed them for two reasons.

First, when observing how lazy was used, we noticed that whenever, a variable-size container of tasks was created, we later realized that it was a suboptimal choice and a better solution did not require a container of tasks.

Second: move-assignment of a lazy is a ticking bomb. To make it safe, we would need to introduce per lazy cancellation of associated coroutines and it is a very heavy-weight solution.

At the moment we do not offer a move assignment, default constructor and swap. If good use cases, for which there are no better ways to solve the same problem are discovered, we can add them.

1.5 Interaction with allocators

`lazy` incorporates the work on coroutine allocators from P1681R0 [8]. Described simply:

`lazy`'s second template parameter `Allocator` can be used to statically declare an allocator type, or can be set to `void`, the default, to indicate that the allocator type is erased. An allocator may or may not be provided to the initial corouroutine call.

If the coroutine has a first parameter of type `std::allocator_arg_t`, the second argument `a` passed to the original call is:

— used to allocate the coroutine state, if the statically declared allocator type is `void`,

— converted to the statically-declared allocator type, which is used to allocate the coroutine state, otherwise.

If the provided allocator is not convertible to a statically-declared non-`void` allocator type, the program is ill-formed.

If the coroutine has no first parameter, or a first parameter with type other than `std::allocator_arg_t`, the coroutine state is allocated with either a default-constructed allocator of the statically-declared allocator type, if it is not `void`, or a default-constructed `std::allocator<byte>` otherwise. If the selected allocator type is not default-constructible, the program is ill-formed.

The actual allocator used is stored type-erased within the coroutine state if it is not both default constructible and stateless, as indicated by `allocator_traits::is_always_equal`. A pointer to a deallocation function which can recover the erased allocator type is stored as well when the statically-declared allocator type is `void`.

Deallocation of the coroutine state can then recover an allocator equivalent to the allocator used to allocate the coroutine state.

1.6 But how do we *use* it?

As we mentioned in the beginning, the only operation that one can do on a lazy is to await on it.Using an *await-expression* in a function turns it into a coroutine. But, this cannot go on forever, at some point, we have to interact with coroutine from a function that is not a coroutine itself, `main`, for example. What to do?

We provide a simple function to bridge the gap between synchronous and asynchronous worlds, `lazy::sync_await`. Calling `sync_await` starts the lazy execution in the current thread, and, if it gets suspended, blocks

until the result is available. (This functionality could be extended to generically handle arbitrary awaitables; P2300 provides just such an implementation for generic senders, and machinery for adapting awaitables into senders. The future proposal integrating `lazy` with P2300 will undoubtedly remove this limited special-case function.)

1.6.1 Implementation experience

A version of proposed type has been used in shipping software that runs on hundreds of million devices in consumer hands. Also, a similar type has been implemented in most extensive coroutine abstraction library CppCoro [1]. This proposed type is minimal and efficient and can be used to build higher level abstraction by composition.

We've also made a Compiler Explorer implementation available for experimentation that works with the three major Standard Library implementations [3].

2 Wording

The technical specifications that follow take the form of excerpts from the working draft N4901 [5] with change markings:

— ~~Text to be struck is in red with strikethough~~, and

— text to be added is "green" with underline.

17.3.2 Header `<version>` synopsis [version.syn]

[Editor's note: Add a new feature-test macro to the `<version>` synopsis, in the appropriate order, replacing `20XXYYL` with the year-and-month of merge:]

```
#define __cpp_lib_lazy                  20XXYYL // also in <lazy>
```

[Editor's note: Add definitions to the `<coroutine>` synopsis:]

17.12.2 Header `<coroutine>` synopsis [coroutine.syn]

```
#include <compare>              // see [compare.syn]

namespace std {
  [...]

  // [coroutine.trivial.awaitables], trivial awaitables
  struct suspend_never;
  struct suspend_always;

  // 17.12.7, awaitable concepts
  template<class T, class Promise = void>
  concept simple_awaitable = see below;

  template<class T, class Promise = void>
  concept awaitable = see below;
}
```

[Editor's note: Insert new subclauses at the end of [support.coroutine]:]

17.12.7 Awaitable concepts [coroutine.awaitable]

1 The `awaitable` and `simple_awaitable` concepts specify the requirements on a type that is usable in an *await-expression* ([expr.await]).

```
template<class T>
concept suspend-result = see below; // exposition only

template<class T, class Promise = void>
concept simple_awaitable = requires(T& t, const coroutine_handle<Promise>& h) {
  { t.await_ready() } -> convertible_to<bool>; // not required to be equality-preserving
  { t.await_suspend(h) } -> suspend-result; // not required to be equality-preserving
```

```
  t.await_resume(); // not required to be equality-preserving
};

template<class T, class Promise = void>
concept has-member-co_await = // exposition only
  requires(T&& t) {
    { std::forward<T>(t).operator co_await() }
      -> simple_awaitable<Promise>; // not required to be equality-preserving
  };

template<class T, class Promise = void>
concept has-ADL-co_await = // exposition only
  requires(T&& t) {
    { operator co_await(std::forward<T>(t)) }
      -> simple_awaitable<Promise>; // not required to be equality-preserving
  };

template<class T, class Promise = void>
concept awaitable = has-member-co_await<T, Promise> ||
  has-ADL-co_await<T, Promise> || simple_awaitable<T, Promise>;
```

2 For a type `T`, *`suspend-result`*`<T>` is satisfied if `T` denotes `void` or `bool`, or if `T` is a specialization of `coroutine_handle`.

3 For an expression E, if `decltype((`E`))` satisfies the `awaitable<P>` concept for some promise type `P`, then a *simple awaitable of* E is an object satisfying the `simple_awaitable<P>` concept that is either the result of evaluation of E itself or the result of an application (if available) of `operator co_await` to E.

17.12.8 Coroutine tasks [coroutine.lazy]

17.12.8.1 Overview [coroutine.lazy.overview]

1 This subclause describes components that a program can use to create coroutines representing asynchronous computations.

17.12.8.2 Header <lazy> synopsis [lazy.syn]

```
namespace std {
  template<class T = void, class Allocator = void>
  class lazy;
}
```

17.12.8.3 Class template lazy [coroutine.lazy.type]

1 The class template `lazy` defines a type for a coroutine lazy object that can be associated with a coroutine whose return type is `lazy<T>` for some type `T`. This subclause refers to such a coroutine as a *lazy coroutine* and to type `T` as the *eventual type* of the lazy coroutine.

```
template<class T = void, class Allocator = void>
  class [[nodiscard]] lazy {
  public:
    lazy(lazy&& that) noexcept;

    ~lazy();

    unspecified operator co_await();

    T sync_await();
  };
```

2 *Mandates*: `T` denotes `void`, a reference type, or a `move_constructible` object type.

3 *Preconditions*: `Allocator` denotes `void` or meets the *Cpp17Allocator* requirements ([allocator.requirements.general]).

4 The library provides specializations of `coroutine_traits` and implements `lazy`'s member `operator co_await` as necessary to provide the following behaviors.

5 If, in the definition of a lazy coroutine, the first parameter has type `allocator_arg_t`, then the coroutine shall have at least two parameters and the type of the second shall meet the *Cpp17Allocator* requirements. If

dynamic allocation is required to store the coroutine state ([dcl.fct.def.coroutine]), the implementation uses the second argument either directly, if `Allocator` denotes `void`, or indirectly, via conversion to `Allocator` if `Allocator` is non-void, to allocate and deallocate the coroutine state. The program is ill-formed if a provided allocator is not convertible as required. If no allocator is provided via arguments to the lazy coroutine, it uses a default-constructed `Allocator`, if non-`void`, or `allocator<byte>` otherwise. In any case, the implementation ensures that deallocation is performed with an allocator equivalent to the allocator used for allocation.

6 If a *yield-expression* ([expr.yield]) occurs in the suspension context of a lazy coroutine, the program is ill-formed.

7 A call to a lazy coroutine `f` returns a lazy object `t` associated with that coroutine. The coroutine is suspended at the initial suspend point ([dcl.fct.def.coroutine]). Such a lazy object is considered to be in the *armed* state.

8 The type of a lazy object models the `awaitable` concept. Awaiting on a lazy object in the armed state as if by `co_await t` ([expr.await]) registers the awaiting coroutine `a` with the lazy object `t` and resume the associated coroutine `f`. At this point `t` is considered to no longer be in the *armed* state. Awaiting on a lazy object that is not in the armed state has undefined behavior.

9 Let `sa` be a simple awaitable of `t` (17.12.7). If the *compound-statement* of the *function-body* of the coroutine `f` completes with an unhandled exception, the awaiting coroutine `a` is resumed and evaluation of `sa.await_resume()` rethrows that exception.

10 If the eventual type of a coroutine `f` is `void` and the coroutine completes due to execution of a coroutine return statement ([stmt.return.coroutine]), or flowing off the end of a coroutine, the awaiting coroutine `a` is resumed and evaluation of `sa.await_resume()` shall yield `void`.

11 If the eventual type of a coroutine `f` is not `void` and the coroutine completes due to execution of a coroutine return statement ([stmt.return.coroutine]), the operand of the coroutine return statement is stored in the coroutine state, the awaiting coroutine `a` is resumed, and evaluation of `sa.await_resume()` shall yield the stored value.

[*Note 1*: If the stored value is a glvalue, users should take care to ensure the lifetime of the denoted object is sufficient to guarantee validity of the glvalue yielded from `sa.await_resume()`. — *end note*]

17.12.8.4 Members [coroutine.lazy.mem]

`lazy(lazy&& that) noexcept;`

1 *Postconditions*: `*this` is associated with the coroutine originally associated with `that`, if any, and `that` is associated with no coroutine.

`~lazy();`

2 *Preconditions*: The coroutine associated with `*this`, if any, is suspended.

3 *Effects*: Destroys the coroutine associated with `*this`, if any.

`unspecified` `operator co_await();`

4 *Preconditions*: `*this` is associated with a coroutine suspended at its initial suspend point.

Returns: An object whose type models `simple_awaitable` associated with the same coroutine as `*this`.

`T sync_await();`

5 *Preconditions*: `*this` is associated with a coroutine suspended at its initial suspend point.

Effects: Evaluates `operator co_await()` to obtain the yielded `simple_awaitable` object. If that object indicates via `await_ready` that the coroutine result is not yet ready, runs the coroutine to completion by calling `await_suspend`). Finally returns the coroutine result by evaluation of `await_resume`.

Bibliography

[1] Lewis Baker. CppCoro: A library of c++ coroutine abstractions for the coroutines ts. https://github.com/lewisbaker/cppcoro. Accessed: 2021-12-09.

[2] Lewis Baker and Gor Nishanov. P1056R1: Add lazy coroutine (coroutine task) type. https://wg21.link/p1056r1, 10 2018.

[3] Casey Carter. Implementation of `std::lazy`. https://godbolt.org/z/dxxavazPa, 2021.

[4] Michaominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, and Bryce Adelstein Lelbach. P2300R3: std::execution. https://wg21.link/p2300r3, 12 2021.

[5] Thomas Köppe. N4901: Working draft, standard for programming language c++. https://wg21.link/n4901, 10 2021.

[6] Gor Nishanov. P0399R0: Networking TS & threadpools. https://wg21.link/p0399r0, 10 2017.

[7] Gor Nishanov. P0913R1: Add symmetric coroutine control transfer. https://wg21.link/p0913r1, 3 2018.

[8] Gor Nishanov. P1681R0: Revisiting allocator model for coroutine lazy/task/generator. https://wg21.link/p1681r0, 6 2019.

[9] Richard Smith and Gor Nishanov. P0981R0: Halo: coroutine heap allocation elision optimization: the joint response. https://wg21.link/p0981r0, 3 2018.