GAMECENTRIC

Alberto Barbati
alberto@gamecentric.com

Coroutines in practice

Italian C++ Conference 2019

June 15, Milan

www.italiancpp.org

HOST

PATRON

Community Crumbs

SPONSORS

Who am I?

C++ programmer since 1990

Works in videogame industry since 2000

Core Trainer of Game Programming at Digital Bros Game Academy

Follows the works of the C++ Committee since 2008

What will we see today?

- A skeleton implementation of a very simple coroutines framework
  - Single-threaded, cooperative multitasking
- An application of the framework for inter-process communication
- A few parts are Windows-specific (named pipes, events)
- Written for Visual Studio 2019 with /std:c++latest and /await to enable experimental support for the Coroutine TS

Coroutine

- A coroutine is a function that can be suspended and subsequently resumed
- While the coroutine is suspended, the thread is not stopped, but another function is resumed instead

Final goal (server side)

```
Coroutine<> server()
{
    Pipe pipe = co_await Pipe::createPipe(L"PipeName", 4096);

    std::vector<std::byte> data = makeData();

    co_await pipe.write(data);
}
```

Final goal (client side)

```
Coroutine<> client()
{
    Pipe pipe = co_await Pipe::openPipe(L"PipeName");

    std::vector<std::byte> buffer(pipe.bufferSize());
    auto msglen = co_await pipe.read(buffer);

    // do something with buffer[0, msglen-1]
}
```

Reasons for suspension

- We identify two main reasons for suspending a coroutine:
  - Waiting for another coroutine to complete
  - Waiting for an asynchronous operation to complete

Reasons for suspension (on Windows)

- We identify two main reasons for suspending a coroutine:
  - Waiting for another coroutine to complete
  - Waiting for a HANDLE object to become signaled

Bootstrapping coroutines

```
int main()
{
    CoroLoop loop;

    // TODO: add one or more coroutines to the loop

    loop.run();
}
```

Library support

```
#include <experimental/coroutine>
using std::experimental::coroutine_handle;
```

- The `coroutine_handle<T>` class template encapsulate a handle to an invocation of a coroutine
- `coroutine_handle<void>` or simply `coroutine_handle<>` objects can represent any kind of coroutines
- `coroutine_handle<T>` objects are tied to a specific kind of coroutines, they are always convertible to `coroutine_handle<>`

class CoroLoop

```
class CoroLoop
{
public:
    CoroLoop();
    static CoroLoop& getLoop();

    void schedule_blocked(HANDLE handle, coroutine_handle<> coro);
    void schedule_ready(coroutine_handle<> coro);

    void run();

private:
    std::vector<HANDLE>              m_handles;
    std::vector<coroutine_handle<>>  m_blocked;
    std::vector<coroutine_handle<>>  m_ready;
};
```

CoroLoop construction

```
CoroLoop::CoroLoop()
{
    s_loop = this;
}

static CoroLoop& CoroLoop::getLoop()
{
    return *s_loop;
}
```

Adding “blocked” coroutines to the pool

```
private:
    std::vector<HANDLE>                m_handles;
    std::vector<coroutine_handle<>> m_blocked;

public:
    void schedule_blocked(HANDLE handle, coroutine_handle<> coro)
    {
        m_handles.push_back(handle);
        m_blocked.push_back(coro);
    }
```

Adding “ready” coroutines to the pool

```
private:
    std::vector<coroutine_handle<>>  m_ready;

public:
    void schedule_ready(coroutine_handle<> coro)
    {
        m_ready.push_back(coro);
    }
```

CoroLoop::run()

```
void CoroLoop::run()
{
    while (!m_blocked.empty() && !m_ready.empty())
    {
        // process "blocked" coroutines

        // process "ready" coroutines
    }
}
```

Processing the blocked coroutines

```
DWORD result = WaitForMultipleObjectsEx(
    m_handles.size(), m_handles.data(), // HANDLEs to wait for
    false,
    m_ready.empty() ? INFINITE : 0, // timeout
    true);
```

Resuming blocked coroutines

```
if (result >= WAIT_OBJECT_0
    && result < WAIT_OBJECT_0 + m_handles.size())
{
    unsigned index = result - WAIT_OBJECT_0;
    auto coro = m_blocked[index];
    m_handles.erase(m_handles.begin() + index);
    m_blocked.erase(m_blocked.begin() + index);
    coro.resume();
}
```

What does resume() do?

- Calling `resume()` on a coroutine handle resumes the corresponding coroutine invocation
- When the coroutine eventually reaches a suspension point, the `resume()` function will simply returns and control gets back to the loop

Resuming ready coroutines

```
std::vector<coroutine_handle<>> ready;
ready.swap(m_ready);
for (auto coro : ready)
{
    coro.resume();
}
```

Example of a coroutine

```
Coroutine<> client()
{
    Pipe pipe = co_await Pipe::openPipe(L"PipeName");

    std::vector<std::byte> data = makeData();

    co_await pipe.write(data);
}
```

Example of a coroutine

```
Coroutine<Pipe> Pipe::openPipe(const wchar_t* name)
{
    // creates a pipe, co_awaiting until
    // it's connected to the server

    co_return Pipe { pipe };
}
```

Await-expression

`co_await` *`expression`*

- An expression, not a statement: it may have a value
- Introduces a potential suspension point for a coroutine
- *`expression`* is evaluated, let's call the result *`e`*
- By calling methods on *`e`* the compiler generates custom code related to suspending and resuming the coroutine
- If the type of *`e`* has all required methods, we call it an "awaitable" type

Suspending

- The compiler calls `e.await_ready()`
  - If the result is true, *e* is considered to be “ready”, meaning that there’s nothing to wait for: the coroutine is not suspended
  - Otherwise, the coroutine is suspended: the compiler calls `e.await_suspend(h)`, where `h` is `a coroutine_handle<>` that refers to the coroutine being suspended

Resuming

- Eventually, the coroutine will be resumed and execution is yielded back to the coroutine
- We are not yet finished with the await expression!
- The compiler calls `e.await_resume()` that returns the value of the await-expression (or void if there's no such value)

Our first awaitable type

- Let's write our first awaitable type to allows a coroutine to be suspended and resumed when a `HANDLE` object becomes signaled

Handle

```
class Handle
{
    HANDLE m_handle;

public:
    // RAII wrapper around HANDLE
    // ctor, dtor, movable, non-copiable

    bool await_ready();
    void await_suspend(coroutine_handle<> h);
    void await_resume();
};
```

These methods make the type “awaitable”

Making Handle awaitable (ready)

```
bool Handle::await_ready()
{
    return false;
}
```

Making Handle awaitable (suspend)

```
void Handle::await_suspend(coroutine_handle<> coro)
{
    CoroLoop::getLoop().schedule_blocked(m_handle, coro);
}
```

Making Handle awaitable (resume)

```
void Handle::await_resume()
{ }
```

Next goal: write our coroutine type

```
template <class Result = void>
class [[nodiscard]] Coroutine
{
    // TODO
};
```

Promise type

- When the compiler compiles a coroutine, first of all it selects the *promise* type
- If *C* is the return type of the coroutine function, by default the promise type is `C::promise_type`
- The selection can be customized by specializing the `std::coroutine_traits` template, we won't see that today

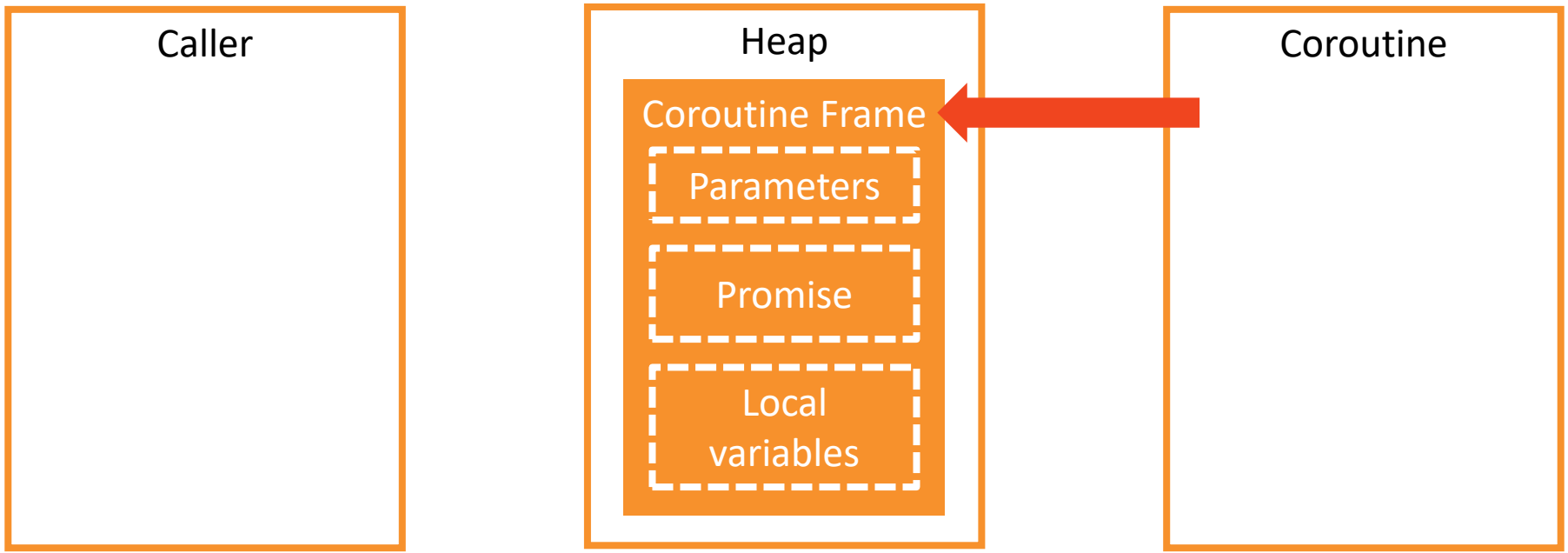
Declaring the promise

```
template <class Result>
class Promise
{
    // ...
};

template <class Result = void>
class [[nodiscard]] Coroutine
{
public:
    using promise_type = Promise<Result>;

    // ...
};
```

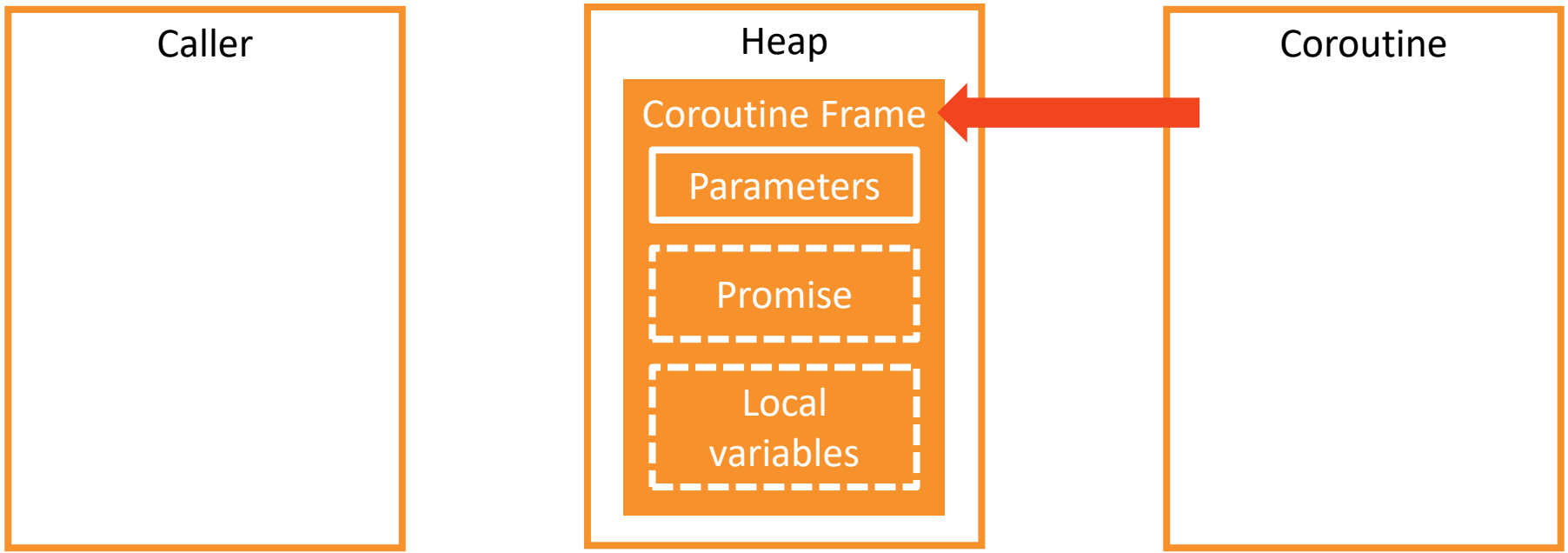
The coroutine frame

- Upon entering the coroutine function, the compiler allocates the coroutine frame, a block of memory suitable to contain:
  - All local variables, including function parameters
  - An object of the promise type
  - Implementation-defined stuff
- By default, the memory is allocated from the heap, using a normal call to `::operator new`, but the allocation can be customized

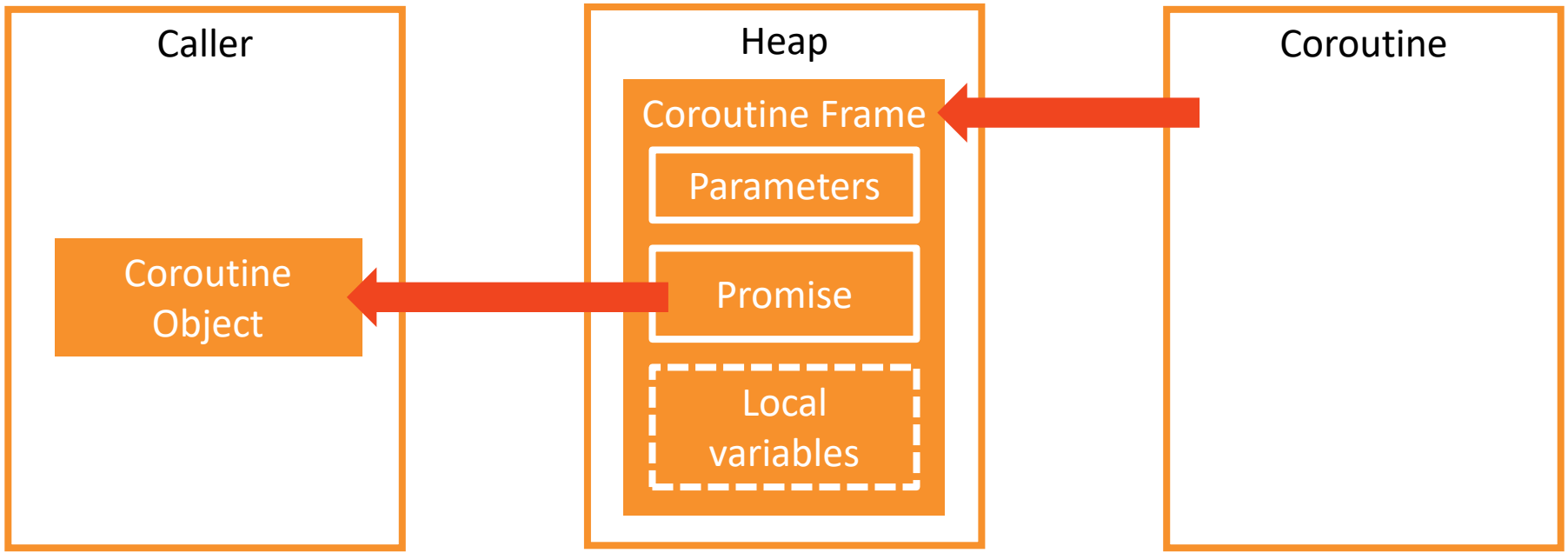
Coroutine frame



Coroutine frame



Coroutine frame



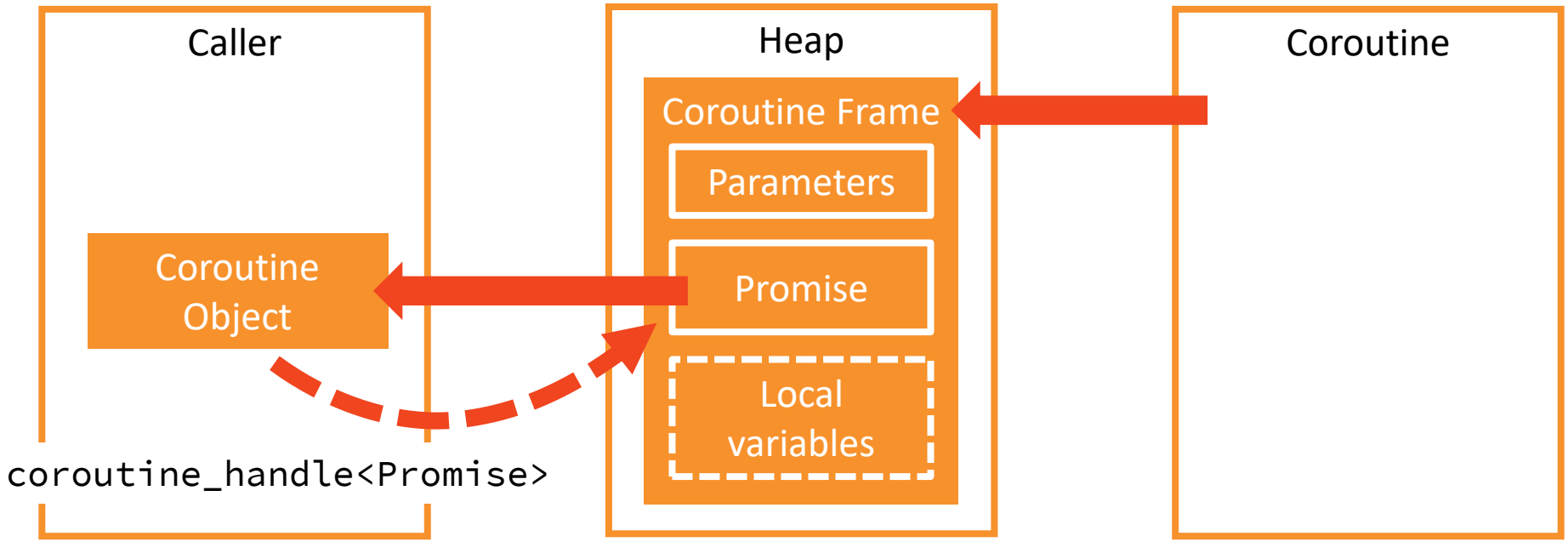
How is the result object created?

- Let's call *p* the promise object in the coroutine frame
- The return object in the calling context is initialized by evaluating `p.get_return_object()`

coroutine_handle<Promise>

- The template parameter of the `coroutine_handle<>` template is the type of the promise
- By using `coroutine_handle<Promise>` you can always retrieve a reference to the corresponding promise object and vice-versa

Coroutine frame



Keeping track of the promise

```
template <class Result = void>
class [[nodiscard]] Coroutine
{
    coroutine_handle<Promise<Result>> m_coro;

public:
    Coroutine(coroutine_handle<Promise<Result>> coro)
        : m_coro { coro }
    {}

    // ...
};
```

The return object is created

```
template <class Result>
Coroutine<Result> Promise<Result>::get_return_object()
{
    return { coroutine_handle<Promise>::from_promise(*this) };
}
```

The coroutine body

```
co_await p.initial_suspend();

try
{
    function-body
}
catch ( ... )
{
    p.unhandled_exception();
}

co_await p.final_suspend();
```

Initial suspend point

```
template <class Result>
auto Promise<Result>::initial_suspend()
{
    return std::experimental::suspend_never {};
}
```

Final suspension point

- The final suspension poin is a bit more complicated, since we need to implement the interaction between two coroutines
- The caller, which uses `co_await` to suspend and wait for the callee to complete
- The callee, which may produce a return value
- When the callee completes, the return value must be passed to the caller and the caller must be resumed

Reminder

```
Coroutine<> client()
{
    Pipe pipe = co_await Pipe::openPipe(L"PipeName");
    // ...
}

Coroutine<Pipe> Pipe::openPipe(const wchar_t* name)
{
    // ...
    co_return Pipe { pipe };
}
```

Making Coroutine<> awaitable (ready)

```
template <class Result>
bool Coroutine<Result>::await_ready() const
{
    return m_coro.done();
}
```

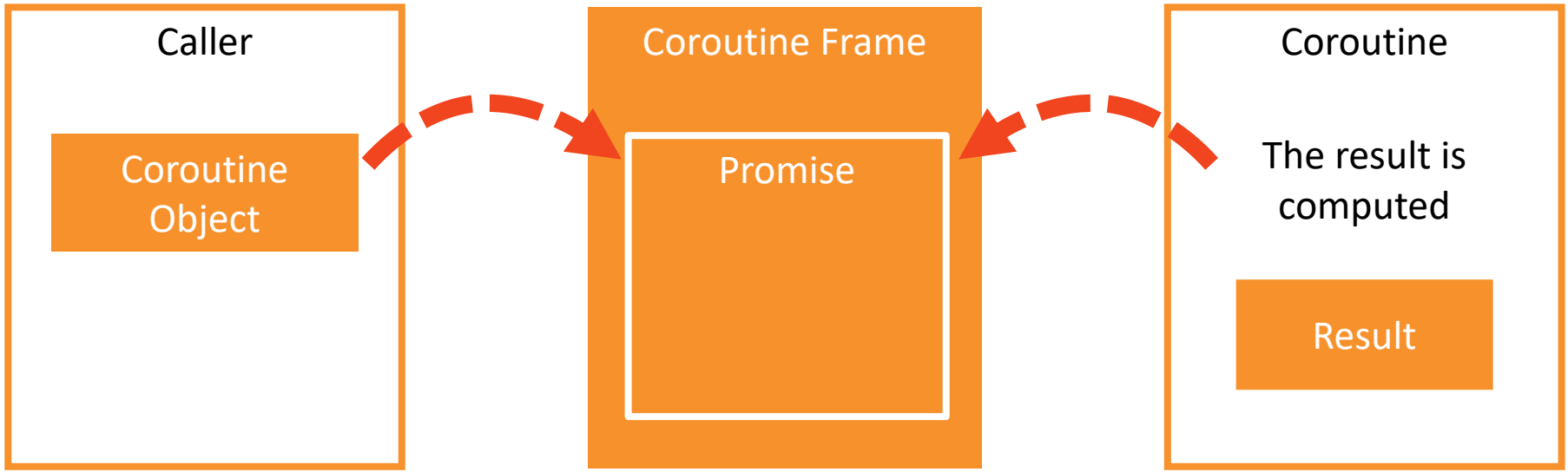
Making Coroutine<> awaitable (suspend)

```
template <class Result>
void Coroutine<Result>::await_suspend(coroutine_handle<> h)
{
    m_coro.promise().m_awaitingCoro = h;
}
```

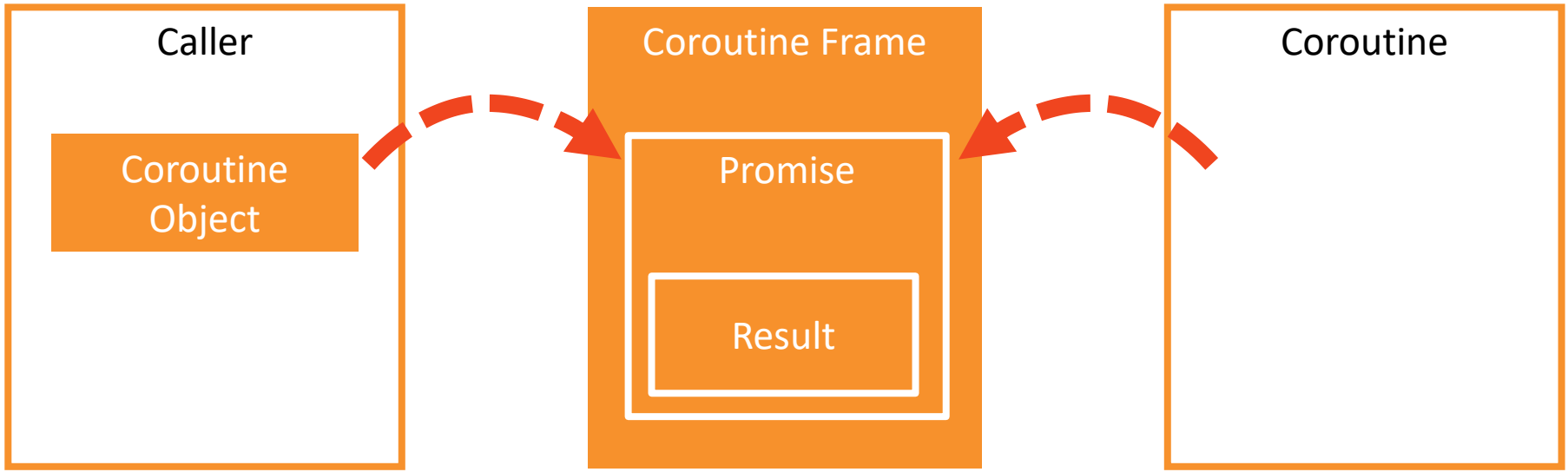
The promise keeps track of an awaiting coroutine

```
template <class Result>
class Promise
{
    coroutine_handle<> m_awaitingCoro;

    // ...
};
```

Return value from a coroutine



Return value from a coroutine



The co_return statement

```
co_return expression;
```

```
p.return_value(expression);
```

```
co_return;
```

or, implicitly, if execution flows off the coroutine body

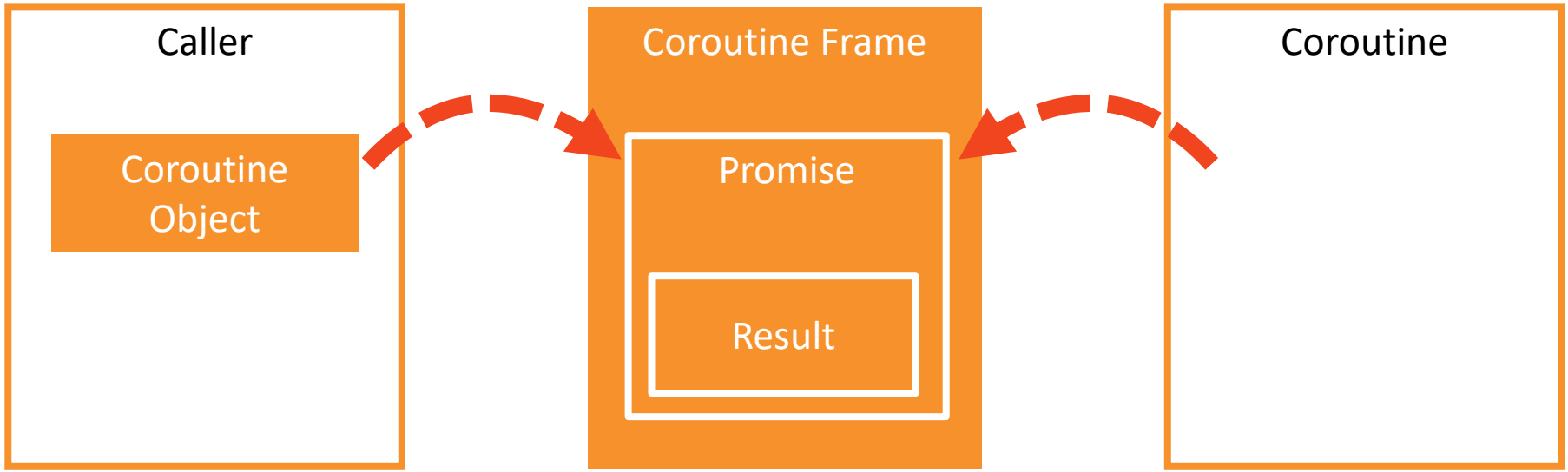
```
p.return_void();
```

class Promise<R> : public BasicPromise<R>

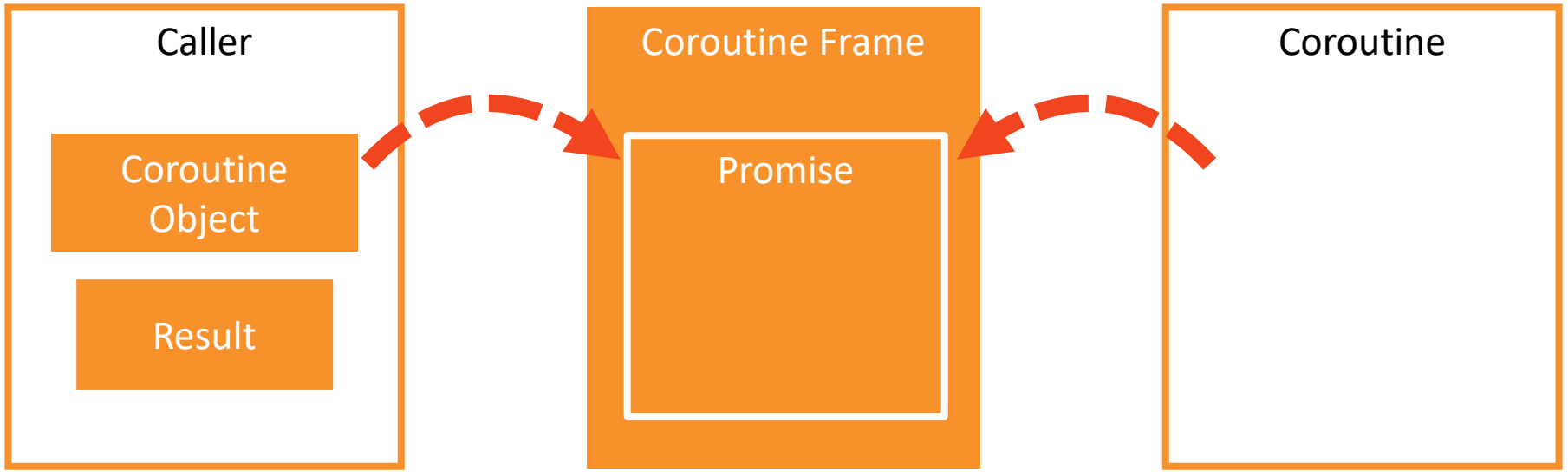
```
template <class R>
class BasicPromise
{
  std::optional<R> m_result;

public:
  void return_value(R value)
  {
    m_result = std::move(value);
  }

  R result()
  {
    return std::move(*m_result);
  }
};
```

```
template <>
class BasicPromise<void>
{
public:
  void return_void()
  {}

  void result()
  {}
};
```

Return value from a coroutine



Return value from a coroutine



Making coroutine awaitable (resume)

```
template <class Result>
Result Coroutine<Result>::await_resume() const
{
    return m_coro.promise().result();
}
```

Final suspend point, finally

- It’s now time to implement the final suspension point in the promise type
- We just need to resume an awaiting coroutine, if any
- Since the task is not trivial, we would need to invent a new awaitable type and return an instance of it, or…

A little twist

... we can just make Promise awaitable and just return a reference to *this!

```
template <class Result>
auto& Promise<Result>::final_suspend()
{
    return *this;
}
```

Making Promise awaitable (ready)

```
template <class Result>
bool Promise<Result>::await_ready() const
{
    return false;
}
```

Making Promise awaitable (suspend)

```
template <class Result>
void Promise<Result>::await_suspend(coroutine_handle<>)
{
    if (m_awaitingCoro)
    {
        EventLoop::getLoop().schedule_ready(m_awaitingCoro);
    }
}
```

Making Promise awaitable (suspend) in C++20

```
template <class Result>
coroutine_handle<>
Promise<Result>::await_suspend(coroutine_handle<>)
{
    if (m_awaitingCoro)
        return m_awaitingCoro;
    else
        return std::noop_coroutine();
}
```

Making Promise awaitable (resume)

```
template <class Result>
void Promise<Result>::await_resume()
{
    // Will never be invoked
}
```

Destructor and copy

```
template <class Result>
Result Coroutine<Result>::~Coroutine()
{
    if (m_coro)
    {
        m_coro.destroy();
    }
}
```

We’re almost there!

The Pipe class

```
class Pipe
{
    HANDLE m_handle;

    // ctor, dtor, move-only ctor and operator=

    static Coroutine<Pipe> createPipe(const wchar_t* name,
                                      unsigned maxBufferSize);
    static Coroutine<Pipe> openPipe(const wchar_t* name);

    Coroutine<size_t> read(span<std::byte> buffer);
    Coroutine<> write(span<const std::byte> data);
}
```

```
Coroutine<Pipe> Pipe::createPipe(const wchar_t* name, unsigned maxBufferSize)
{
    HANDLE pipe = CreateNamedPipe(name,
        PIPE_ACCESS_DUPLEX | FILE_FLAG_OVERLAPPED, ...);
    if (pipe == INVALID_HANDLE_VALUE) { ReportError(GetLastError()); }

    Handle event { CreateEvent() };
    OVERLAPPED ol = makeOverlapped(event.handle());

    ConnectNamedPipe(pipe, &ol);
    DWORD error = GetLastError();

    if (error == ERROR_PIPE_CONNECTED)  { /* client connected */ }

    else if (error != ERROR_IO_PENDING) { ReportError(error); }

    else
    {
        co_await event; // wait for client to connect
    }
    co_return Pipe { pipe };
}
```

```
Coroutine<Pipe> Pipe::openPipe(const wchar_t* name)
{
    for (;;)
    {
        HANDLE pipe = CreateFile(name, ...,
                OPEN_EXISTING, FILE_FLAG_OVERLAPPED, ...);

        if (pipe != INVALID_HANDLE_VALUE)
        {
            co_return Pipe { pipe };
        }
        else
        {
            DWORD error = GetLastError();
            if (error != ERROR_FILE_NOT_FOUND) ReportError(error);
            co_await WaitFor(1s);
        }
    }
}
```

```
Coroutine<size_t> Pipe::read(span<std::byte> buffer)
{
    Handle event { CreateEvent() };
    OVERLAPPED ol = makeOverlapped(event.handle());
    if (ReadFile(m_handle, buffer.data(), buffer.size(), nullptr, &ol) == 0)
    {
        DWORD error = GetLastError();
        if (error != ERROR_IO_PENDING) ReportError(error);
         co_await event;
    }

    DWORD bytesRead;
    if (GetOverlappedResult(m_handle, &ol, &bytesRead, false) == 0)
        ReportError(GetLastError());

    co_return bytesRead;
}
```

Bootstrapping server()

```
int main()
{
    CoroLoop loop;

    auto coro = server();

    loop.run();
}
```

Thanks for your attention

Questions?

Alberto Barbati
alberto@gamecentric.com
Twitter @gamecentric

Source code at https://gitlab.com/gamecentric/coroutines-in-practice