Lifetime analysis for C++20 Coroutines

LLVM MUC Meetup
Jan 29, 2024

Presenter: Utkarsh Saxena
usx@google.com
usx95@ LLVM discourse

Overview

- Coroutines in C++20 (what and why ?)
- Edges and pitfalls
- Solutions
- Rough edges

Not in this talk

- Implementing coroutines
- Understanding codegen

What are coroutines ?

- Suspendable functions
  - Can suspend themselves.
  - Other entities can resume them.
- Stateful
  - Stores the state (local variable, resume points)
- Stackless (in C++20)

Why coroutines ?

- Readability of async code:
  - Simpler: normal straight line function
  - Better than registering nested callbacks.

```
// Sync
std::string GetFileContents(const absl::string_view path) {
    const FileHandle file_handle = OpenFile(path);
    const std::string data = file_handle.Read(path);
    return data;
}
```

```
// Async with callbacks (before C++20)
void GetFileContents(const absl::string_view path, Callback callback) {
    OpenFile(path, [&](FileHandle file_handle) {
        file_handle.Read(path, [&](std::string data) { callback(data); })
    });
}
```

```
// Async with Coroutines
task<std::string> GetFileContents(const absl::string_view path) {
    const FileHandle file_handle = co_await OpenFile(path);
    const std::string content = co_await file_handle.Read(path);
    co_return content;
}
```

Example: Fibonacci generator

```
// Generates the fib seq: 1,1,2,3,5,..
Gen makeFibGen() {
    int a = 1;
    int b = 1;
    while(true) {
        co_yield a;
        int c = a + b;
        a = b;
        b = c;
    }
}
```

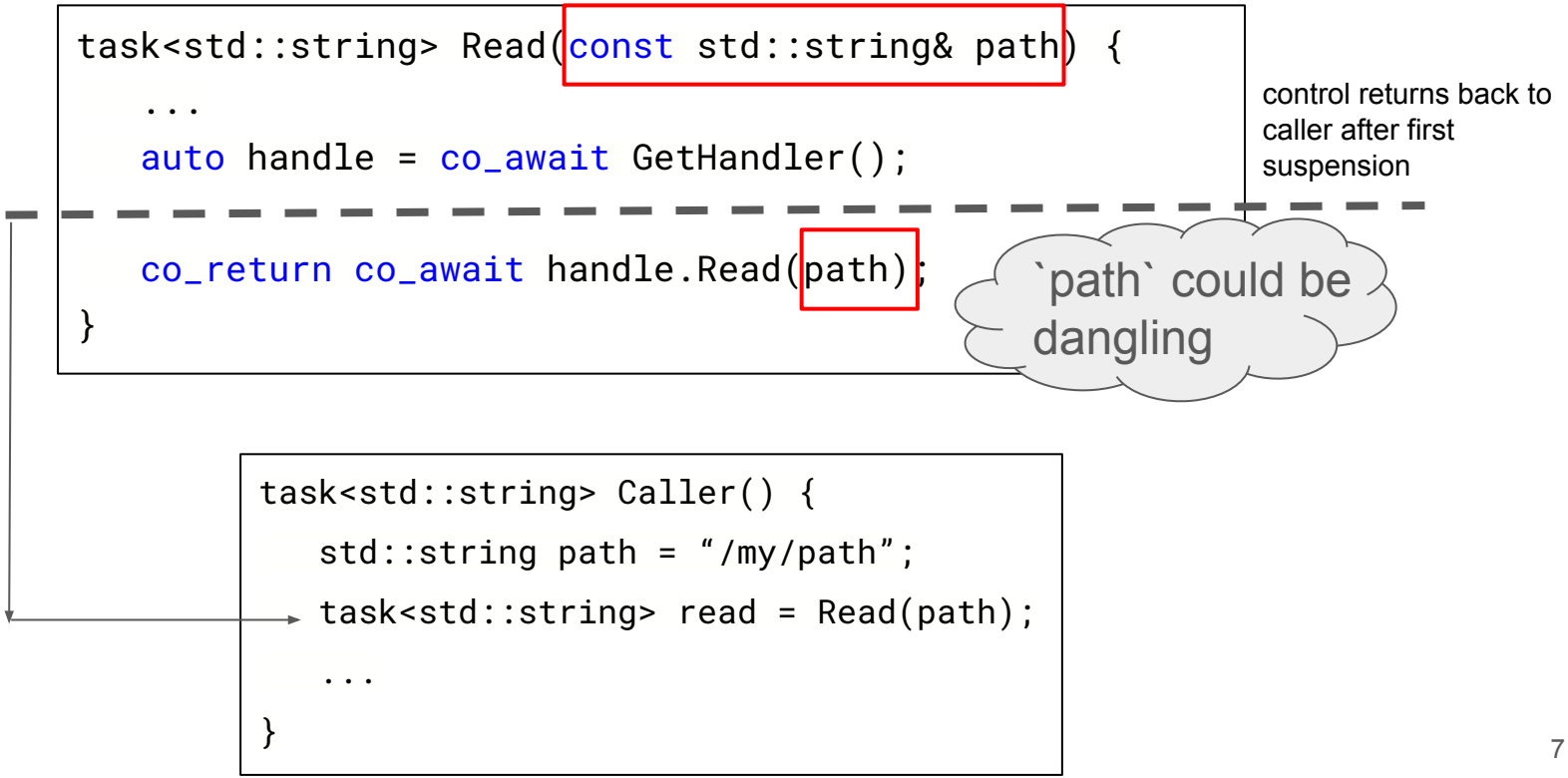
```
struct Gen {
    // Successive calls returns the fib seq: 1,1,2,3,5,..
    int next();
    ...
};

Gen makeFibGen();

void caller() {
    auto gen = makeFibGen();
    for (int i = 1; i <= 10; ++i) {
        std::cout << gen.next() << "\n";
    }
}
```

Pitfalls:
What can go wrong ?

Pitfalls


task<std::string> Read(const std::string& path) {
...
auto handle = co_await GetHandler();
co_return co_await handle.Read(path);
}
control returns back to caller after first suspension
`path` could be dangling
task<std::string> Caller() {
std::string path = “/my/path”;
task<std::string> read = Read(path);
...
}

Dangling reference to temporaries

```
task<std::string> Read(const std::string& path) {

    auto handle = co_await GetHandler();

    co_return co_await handle.Read(path);
}
```

`path` could be dangling

```
std::string GetFilename();
task<std::string> User() {
    auto read = Read(GetFilename());

    std::string content = co_await read;
}
```

```
task<std::string> User() {
    std::string content
      = co_await Read(GetFilename());
}
```

Dangling reference to stack variable

```
task<std::string> Read(const std::string& path) {

    auto handle = co_await GetHandler();

    co_return co_await handle.Read(path);
}
```

`path` could be dangling

```
task<std::string> User(std::string path) {
    return Read(path);
}
```

```
task<std::string> User(std::string path) {
    co_return co_await Read(path);
}
```

std::function

```
struct Request { int num; };

task<int> Add(const Request& a) {
   co_return a.num + 1;
}
```

```
int main() {
   Request r{0};
   std::function<task<int>(Request)> AddFn = Add;
   sync_wait(AddFn(r));
}
```

```
template <class T>
auto wrapper(T arg) {
    // Dangling ref to 'arg' when
    // T is 'Request'
    // (not 'const Request&').
    return Add(arg);
}
```

```
==========================================================
ERROR: AddressSanitizer: stack-use-after-return on address
```

Solution:
Identify at compile time

Identify at compile time

```
struct Request { int num; };

task<int> Add(const Request& a) {
    co_return a.num + 1;
}
```

```
// Ref to stack variable.
task<int> User(Request r) {
   return Add(r);
}
```

```
// Ref to temporary.
task<int> foo = Add(Request{0});
```

`task` (coroutine return object):

…
Coroutine frame:

…
// param.
const Request &a;

The lifetime of **argument** to parameter **`a`** must outlive the return object **`task`**.

This is not new to C++

```
struct Request { int num; };
struct Response { const Request& r; };

Response Foo(const Request& r) {
    return Response{r};
}

int main() {
    Response Res = Foo(Request{0});
    return Res.r.num;
}
```

```
AddressSanitizer: stack-use-after-scope
```

This is not new to C++ : [[clang::lifetimebound]]

```
struct Request { int num; };
struct Response { const Request& r; };

Response Foo([[clang::lifetimebound]] const Request& r) {
   return Response{r};
}

int main() {
   Response Res = Foo(Request{0});
   return Res.r.num;
}
```

```
warning: temporary whose address is used as value of
local variable 'Res' will be destroyed at the end of
the full-expression [-Wdangling]
  16 |     Response Res = Foo(Request{0});
     |                          ^~~~~~~~~~
```

Introduce [[clang::coro_lifetimebound]]

```
template <typename T = void>
struct [[clang::coro_return_type, clang::coro_lifetimebound]]
co_task { /**/ };
```

"Coroutine return type"

```
co_task<int> Add(const Request& a) {
   co_return a.num + 1;
}
```

Implicitly lifetime bound

… Coroutines cannot be identified through their signature.

Arguments to a function with a "coroutine return type" would be considered to be "lifetimebound" to the return object.

Lifetime bound coroutines: Plain returns

```
co_task<int> coro(const int& n) {
   co_return n+1;
}

co_task<int> foo(int n) {
   return coro(n);
}
```

```
<source>:31:17: warning: address of stack memory
associated with parameter 'n' returned
[-Wreturn-stack-address]
  31 |     return coro(n);
```

Lifetime bound coroutines: Temporaries

```
co_task<int> coro(const Request& r) {
   co_return r.num + 1;
}
```

```
Request CreateRequest();


co_task<int> foo() {
   auto task = coro(CreateRequest());
   co_return co_await task;
}
```

```
<source>:38:22: warning: temporary whose address is
used as value of local variable 'task' will be
destroyed at the end of the full-expression
[-Wdangling]
  38 |     auto task = coro(CreateRequest());
     |                      ^~~~~~~~~~~~~~~
```

Lifetime bound coroutines: Lambda captures

```
co_task<int> lambdas() {
    int a = 1;
    auto task = [a]() -> co_task<int> {
        co_return a;
    }();
    co_return co_await task;
}
```

```
<source>:18:17: warning: temporary whose
address is used as value of local variable
'lamb' will be destroyed at the end of the
full-expression [-Wdangling]
  18 |     auto lamb = [a]() -> co_task<int> {
     |                 ^~~~~~~~~~~~~~~~~~~~~~
  19 |         co_return a;
     |         ~~~~~~~~~~~
  20 |     }();
     |     ~
```

Rough edges: False positives with R-value refs

```
co_task<int> coro(Request r) {
   co_return r.n;
}

co_task<int> forward(Request&& r) {
   return coro(r);
}

void use() {
   auto local = forward(Request{0});
}
```

```
<source>:60:25: warning: temporary whose address is
used as value of local variable 'local' will be
destroyed at the end of the full-expression
[-Wdangling]
  60 |     auto local = forward(Request{0});
     |                          ^~~~~~~~~~
```

Cannot use std::function (and others) with coroutines

Thank you

Presenter: Utkarsh Saxena
usx@google.com
usx95@ LLVM discourse