

C++20 Coroutines

Introduction

Marcin Grzebieluch

grzebieluch@me.com

21.11.2019

C++20 Coroutines

Introduction

Marcin Grzebieluch

grzebieluch@me.com

21.11.2019



Plan

- ① why do we need coroutines
- ② what is a coroutine
- ③ under the hood
- ④ implementing task coroutine

1 why do we need coroutines

2 what is a coroutine

3 under the hood

4 implementing task coroutine

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↗ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↗ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↪ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↪ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↪ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking app

```
1 std::pair<blocking::session, experiment> receive();
2
3 void program_main_loop()
4 {
5     thread_pool tp;
6     while(true)
7     {
8         auto [session, experiment] = receive();
9         tp.execute(
10             [s = std::move(session), ex = std::move(
11                 ↪ experiment)]() {
12                 perform_experiment(ex, s);
13             });
14 }
```

my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

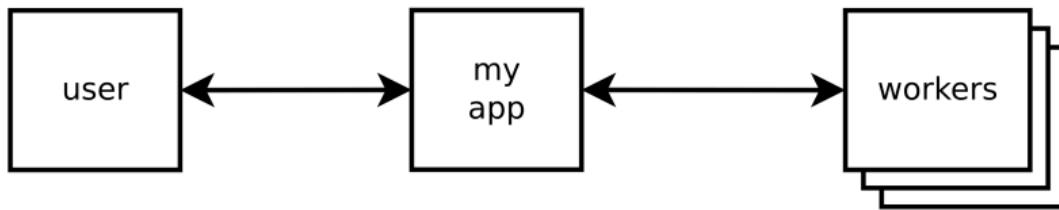
my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

my networking algorithm

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

my networking algorithm



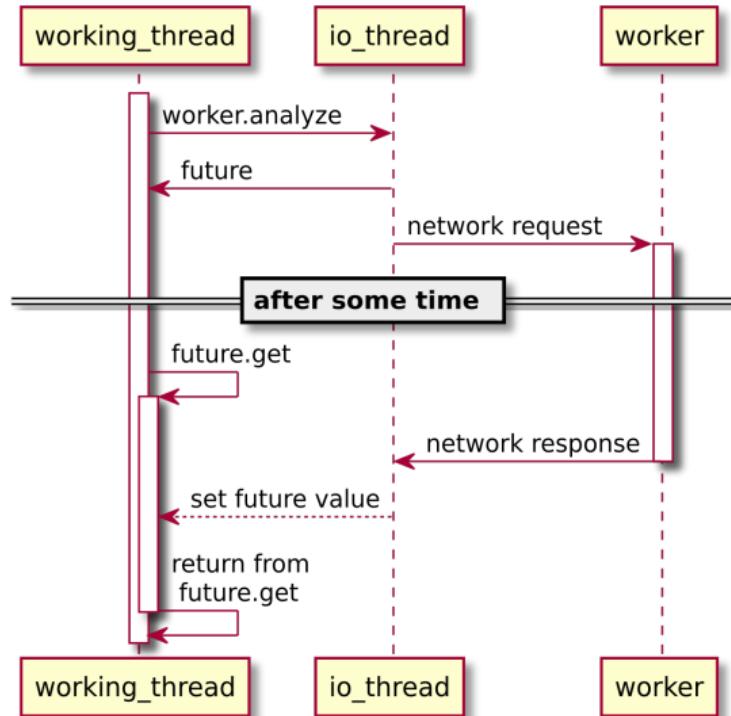
problem

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

problem

```
1 using namespace blocking;
2 void perform_experiment(experiment const& ex, session& s)
3 {
4     std::vector<sample_result> results;
5     for( auto const& sample : ex.samples)
6         results.push_back(worker.analyze(sample));
7     s.respond(results);
8 }
```

partial solution



partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

partial solution

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

how my application feels like



what I want from solution

what I want from solution

- reduce the amount of IO bound threads

what I want from solution

- reduce the amount of IO bound threads
- dont sacrifice readability

what I want from solution

- reduce the amount of IO bound threads
- dont sacrifice readability
- don't force me to split my algorithm into artificial functions

1 why do we need coroutines

2 what is a coroutine

3 under the hood

4 implementing task coroutine

lets start with subroutines

lets start with subroutines

also known as functions

lets start with subroutines

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

lets start with subroutines

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

lets start with subroutines

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

lets start with subroutines

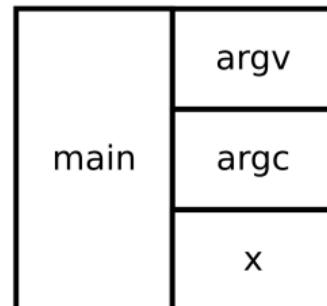
```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

lets start with subroutines

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

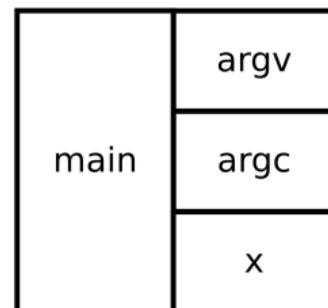
function call procedure

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```



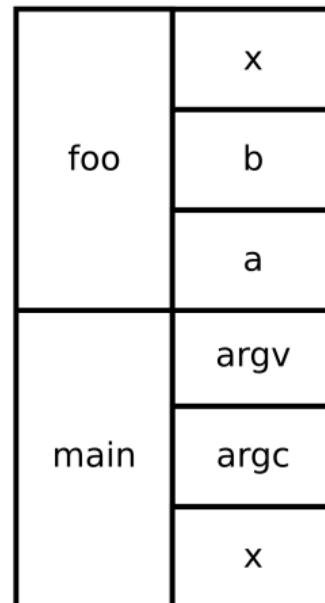
function call procedure

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```



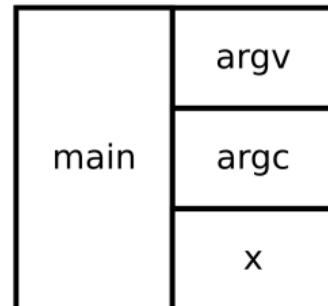
function call procedure

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```



function call procedure

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```



function call procedure

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

subroutine call actions

subroutine call actions

- create stack frame

subroutine call actions

- create stack frame
- execute function code

subroutine call actions

- create stack frame
- execute function code
- return value

subroutine call actions

- create stack frame
- execute function code
- return value
- delete stack frame

coroutine call actions

- create stack frame
- execute function code
- return value
- delete stack frame

coroutine call actions

- create stack frame
- execute function code
- return value
- delete stack frame
- **suspend execution**

coroutine call actions

- create stack frame
- execute function code
- return value
- delete stack frame
- **suspend execution**
- **resume execution**

coroutine

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 int foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_return x;  
4 }  
5  
6 int main() {  
7     int x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_return x;  
4 }  
5  
6 int main() {  
7     task<int> x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_return x;  
4 }  
5  
6 int main() {  
7     task<int> x = foo(5, 7);  
8     return x;  
9 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_return x;  
4 }  
5  
6 int main() {  
7     task<int> x = foo(5, 7);  
8     x.resume();  
9     return x.result();  
10 }
```

coroutine

```
1 task<int> foo(int a, int b) {
2     int x = a + b;
3     co_return x;
4 }
5
6 int main() {
7     task<int> x = foo(5, 7);
8     x.resume();
9     return x.result();
10 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     x.resume();  
10    return x.result();  
11 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     x.resume();  
10    return x.result();  
11 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     x.resume();  
10    x.resume();  
11    return x.result();  
12 }
```

coroutine

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```

coroutine

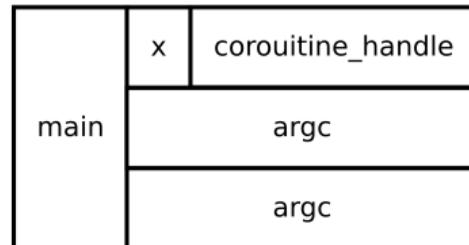
```
1 task<int> foo(int a, int b) {
2     int x = a + b;
3     co_await suspend_always{};
4     co_return x;
5 }
6
7 int main() {
8     task<int> x = foo(5, 7);
9     while(not x.is_ready())
10         x.resume();
11     return x.result();
12 }
```

coroutine call procedure

```
1 task<int> foo(int a, int b) {
2     int x = a + b;
3     co_await suspend_always{};
4     co_return x;
5 }
6
7 int main() {
8     task<int> x = foo(5, 7);
9     while(not x.is_ready())
10         x.resume();
11     return x.result();
12 }
```

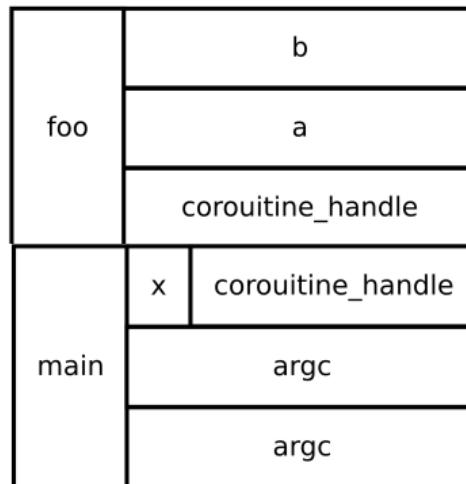
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



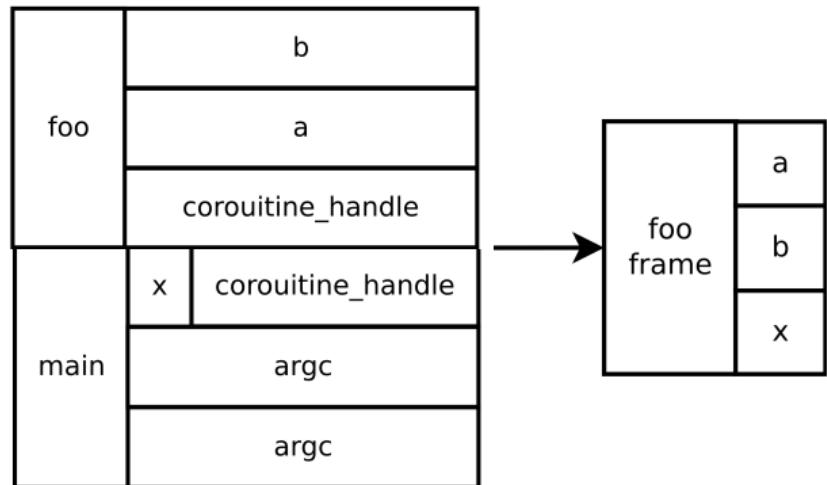
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



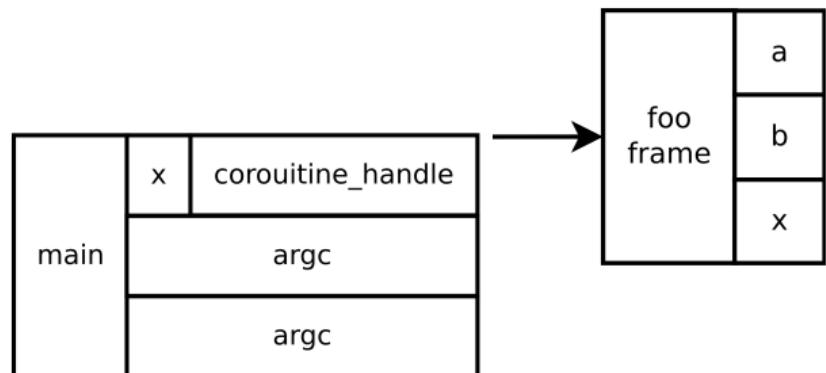
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



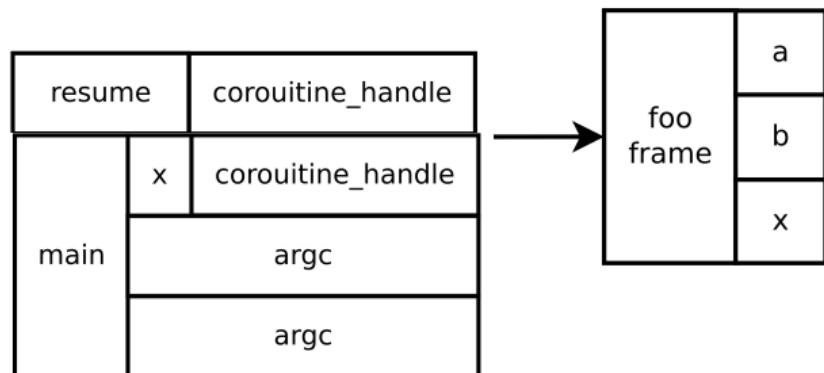
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



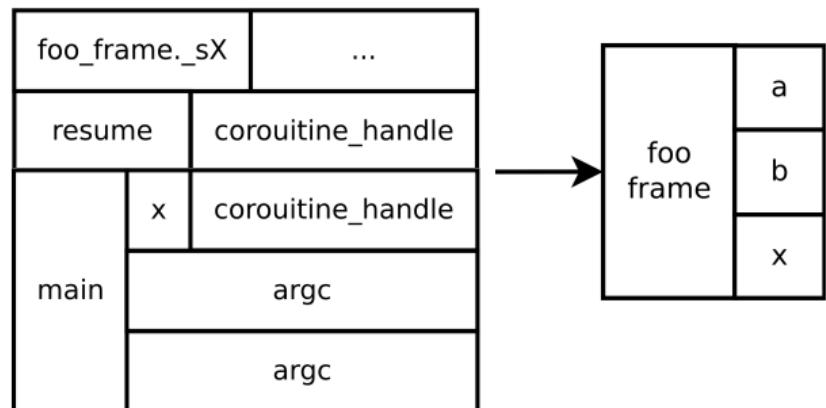
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



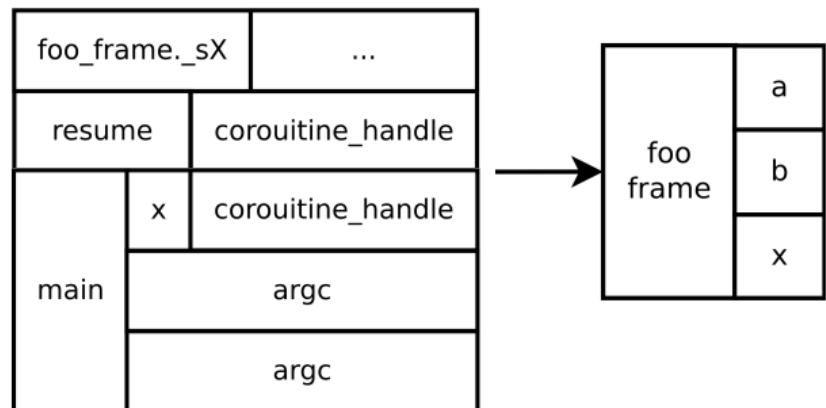
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



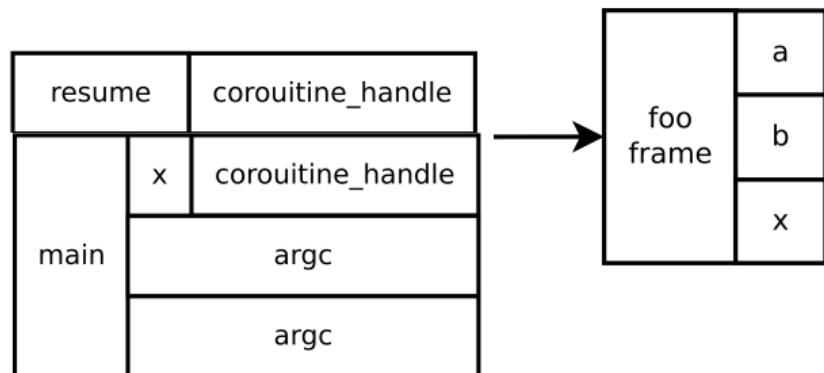
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



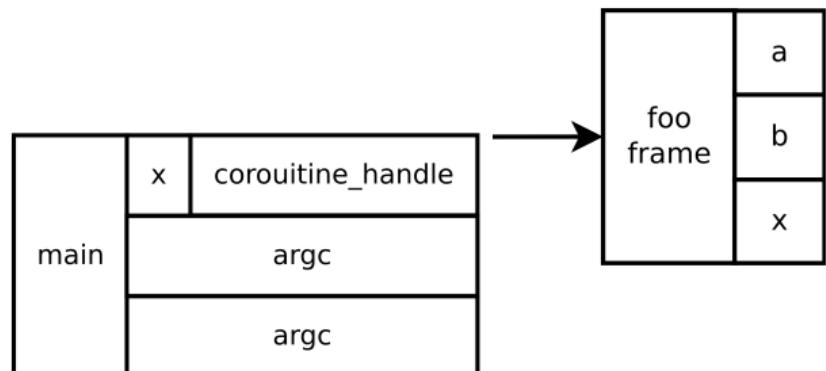
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



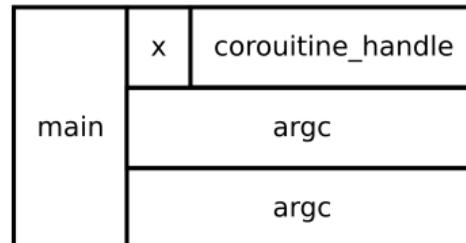
coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```



coroutine call procedure

```
1 task<int> foo(int a, int b) {  
2     int x = a + b;  
3     co_await suspend_always{};  
4     co_return x;  
5 }  
6  
7 int main() {  
8     task<int> x = foo(5, 7);  
9     while(not x.is_ready())  
10         x.resume();  
11     return x.result();  
12 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

previous slide was nightmare to make

```
1 \only<6>{\highlightedListing{7}{10}{cpp/coroutine_call-m.hpp}}
2   \begin{textblock*}{5cm} (7cm,6.3cm)
3     \includegraphics[scale=0.4]{dia/coroutine1_main}
4   \end{textblock*}
5   \begin{textblock*}{5cm} (7.1cm,5.35cm)
6     \includegraphics[scale=0.38]{dia/coroutine1_resume}
7   \end{textblock*}
8   \begin{textblock*}{5cm} (11.61cm,5.05cm)
9     \includegraphics[scale=0.43]{dia/coroutine1_foo_frame}
10  \end{textblock*}
11 }
```

take away

take away

- coroutine is a generalization of a function

take away

- coroutine is a generalization of a function
- it can be suspended and resumed

take away

- coroutine is a generalization of a function
- it can be suspended and resumed
- suspending and resuming is lightweight

take away

- coroutine is a generalization of a function
- it can be suspended and resumed
- suspending and resuming is lightweight
- creating coroutine frame might require memory allocation
- coroutines are stackless

1 why do we need coroutines

2 what is a coroutine

3 under the hood

4 implementing task coroutine

coroutine transformation

```
1 task<int> bar(int a)
2 {
3     int b = 15;
4     co_await suspend_always{};
5     a = b * a;
6     int c = b * 2;
7     co_await suspend_always{};
8     co_return a + b + c;
9 }
```

coroutine transformation

```
1 task<int> bar(int a)
2 {
3     int b = 15;
4     co_await suspend_always{};
5     a = b * a;
6     int c = b * 2;
7     co_await suspend_always{};
8     co_return a + b + c;
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                           2     int a;  
3     int b = 15;             3 };  
4     co_await suspend_always{};  
5     a = b * a;  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     int a;  
4     co_await suspend_always{}; 4 };  
5     a = b * a;  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     int a;  
4     co_await suspend_always{}; 4 };  
5     a = b * a;  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ int b = 15; }  
4     co_await suspend_always{}; 4     int a;  
5     a = b * a;                 5 };  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ int b = 15; }  
4     co_await suspend_always{};  4     int a;  
5     a = b * a;                 5 };  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     int a, b;  
5     a = b * a;                  5 };  
6     int c = b * 2;  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){ a = b * a; }  
5     a = b * a;                  5     int a, b;  
6     int c = b * 2;              6 };  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){ a = b * a; }  
5     a = b * a;                  5     int a, b;  
6     int c = b * 2;              6 };  
7     co_await suspend_always{};  
8     co_return a + b + c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){  
5     a = b * a;                5     a = b * a;  
6     int c = b * 2;              6     int c = b * 2;  
7     co_await suspend_always{};  7     }  
8     co_return a + b + c;        8     int a, b;  
9 }                                9 };
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){  
5     a = b * a;                  5         a = b * a;  
6     int c = b * 2;              6         int c = b * 2;  
7     co_await suspend_always{};  7     }  
8     co_return a + b + c;        8     int a, b;  
9 }                                9 };
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){  
5     a = b * a;                5         a = b * a;  
6     int c = b * 2;              6         c = b * 2;  
7     co_await suspend_always{};  7     }  
8     co_return a + b + c;        8     int a, b, c;  
9 }
```

coroutine transformation

```
1 task<int> bar(int a)          1 struct _bar{  
2 {                                2     _bar(int a_) : a{a_}{};  
3     int b = 15;                  3     void _s1(){ b = 15; }  
4     co_await suspend_always{};  4     void _s2(){  
5     a = b * a;                  5     a = b * a;  
6     int c = b * 2;              6     c = b * 2;  
7     co_await suspend_always{};  7     }  
8     co_return a + b + c;       8     void _s3(){  
9 }                                9     promise->return_value(  
                                10    a + b + c);  
                                11    }  
                                12    int a, b, c, v;  
                                13    promise_type* promise;  
                                14 };
```

coroutine – recap

coroutine – recap

- coroutine body is transformed into coroutnine frame

coroutine – recap

- coroutine body is transformed into coroutine frame
- each suspend point **might** end up as a separate function

1 why do we need coroutines

2 what is a coroutine

3 under the hood

4 implementing task coroutine

implementing coroutine type

implementing coroutine type

- interface type - task/generator/etc.

implementing coroutine type

- interface type - task/generator/etc.
- promise type

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12
```

implementing coroutine type – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12 
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7 }
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7 }
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7 }
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7 }
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7     co_return;
8 }
```

implementing coroutine type – usage

```
1 #include "implementing_simple_task_task-m.hpp"
2 task_ foo()
3 {
4     std::cout << "I_am_..." << std::endl;
5     co_await suspend_always{};
6     std::cout << "...a_COROUTINE!!!" << std::endl;
7     co_return 5; // error, no return_value(int)
8                     // in promise type
9 }
```

returning value – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12 }
```

returning value – promise

```
1 struct promise {
2     auto get_return_object() {
3         return coroutine_handle<promise>::from_promised(*this);
4     }
5     auto initial_suspend() { return suspend_always(); }
6     auto final_suspend() { return suspend_always(); }
7     void return_void() {}
8     void unhandled_exception() {
9         std::terminate();
10    }
11 };
12 }
```

returning value – promise

```
1 struct promise {  
2     //...  
3     void return_void() {}  
4 };
```

returning value – promise

```
1 struct promise {  
2     // ...  
3     void return_void() {}  
4 };
```

returning value – promise

```
1 template <typename T>
2 struct promise {
3     T value_;
4     // ...
5     void return_void() {}
6 };
7
```

returning value – promise

```
1 template <typename T>
2 struct promise {
3     T value_;
4     //...
5     void return_void() {}
6 };
7
```

returning value – promise

```
1 template <typename T>
2 struct promise {
3     T value_;
4     //...
5     void return_value(T t) { value_ = t; }
6 };
7
```

returning value – promise

```
1 template <typename T>
2 struct promise {
3     T value_;
4     auto get_return_object() {
5         return coroutine_handle<promise>::from_promised(*this);
6     }
7     auto initial_suspend() { return suspend_always(); }
8     auto final_suspend() { return suspend_always(); }
9     void unhandled_exception() {
10         std::terminate();
11     }
12     void return_value(T t) { value_ = t; }
13 };
14
```

returning value – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

returning value – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     task_(coroutine_handle<promise> handle)
6         : handle_(handle)
7     {}
8     ~task_() { handle_.destroy(); }
9 private:
10    coroutine_handle<promise> handle_;
11};
```

returning value – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     //...
6 private:
7     coroutine_handle<promise> handle_;
8 };
```

returning value – task

```
1 #include "implementing_simple_task.promise-m.hpp"
2 class task_{
3 public:
4     using promise_type = promise;
5     //...
6 private:
7     coroutine_handle<promise> handle_;
8 };
```

returning value – task

```
1 #include "implementing_returning_task.promise-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     using promise_type = promise<T>;
6     //...
7 private:
8     coroutine_handle<promise<T>> handle_;
9 };
```

returning value – task

```
1 #include "implementing_returning_task.promise-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     using promise_type = promise<T>;
6     //...
7     T value() { return handle_.promise().value_; }
8 private:
9     coroutine_handle<promise<T>> handle_;
10};
```

using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     x.??;
6     return x.value();
7 }
```

using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     x.??;
6     return x.value();
7 }
```

using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     x.??;
6     return x.value();
7 }
```

using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     x.??;
6     return x.value();
7 }
```

usable task

```
1 #include "implementing_returning_task.promise-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     using promise_type = promise<T>;
6     //...
7     T value() { return handle_.promise().value_; }
8 private:
9     coroutine_handle<promise<T>> handle_;
10};
```

usable task

```
1 #include "implementing_returning_task.promise-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     using promise_type = promise<T>;
6     // ...
7     T value() { return handle_.promise().value_; }
8     T resume() { return handle_.resume(); }
9 private:
10    coroutine_handle<promise<T>> handle_;
11 };
```

usable task

```
1 #include "implementing_returning_task.promise-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     using promise_type = promise<T>;
6     // ...
7     T value() { return handle_.promise().value_; }
8     T resume() { return handle_.resume(); }
9     T is_ready() { return handle_.done(); }
10 private:
11     coroutine_handle<promise<T>> handle_;
12 };
13
```

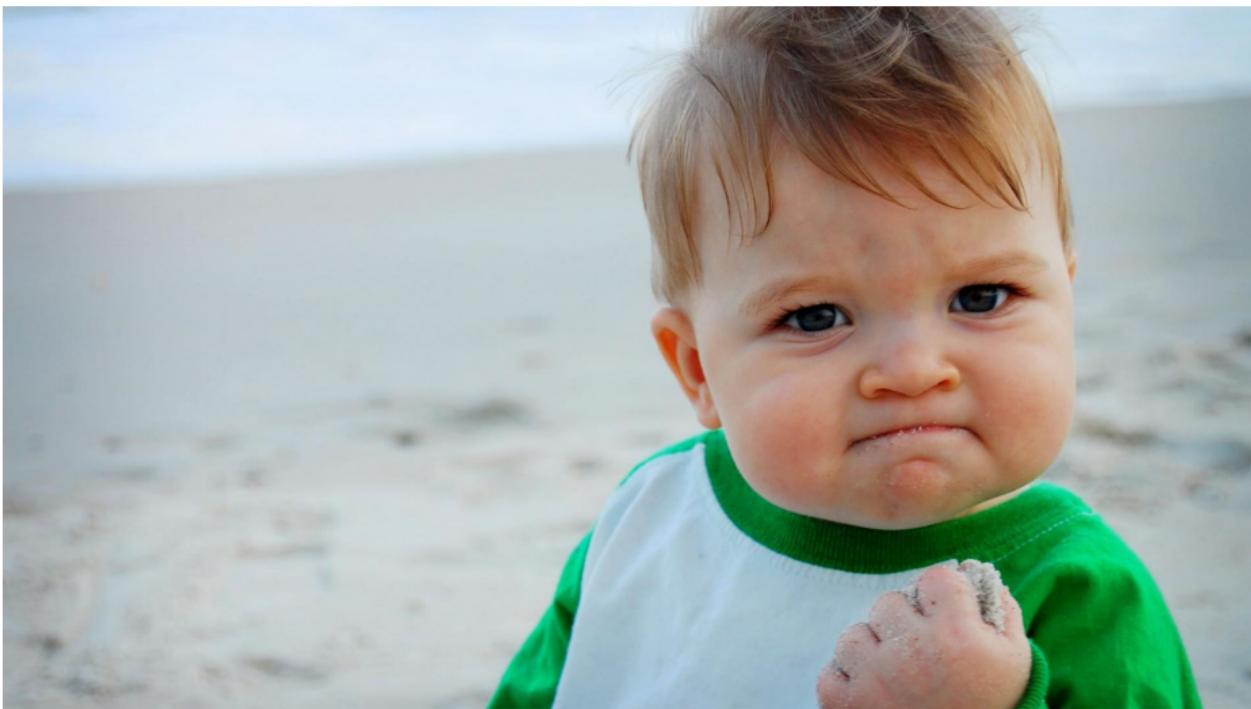
actual using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     x.??;
6     return x.value();
7 }
```

actual using from subroutine

```
1 #include "using_returning_task-m.hpp"
2 int main()
3 {
4     auto x = foo();
5     while( not x.is_ready() )
6         x.resume();
7     return x.value();
8 }
```

we did it!



we have the power!

we have the power!

- interface type - operate on coroutine

we have the power!

- interface type - operate on coroutine
- promise type - control coroutine behaviour

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __buildin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __buildin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __buildin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __builtin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __builtin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

prefetching

```
1 struct data {};
2 int costly_calculation(data&);
3 task<int> prefetched_execution(data* d)
4 {
5     __builtin_prefetch(d, 1, 1);
6     co_await suspend_always{};
7     co_return costly_calculation(*d);
8 }
```

generator!

```
1 #include "generator-m.hpp"
2 generator<int> foo() {
3     for(int i = 0; i < 10; ++i)
4         co_yield i;
5 }
6
7 int main() {
8     auto x = foo();
9     std::optional<int> i;
10    while((i = x.get_next()) != std::nullopt)
11        std::cout << *i << std::endl;
12 }
```

generator!

```
1 #include "generator-m.hpp"
2 generator<int> foo() {
3     for(int i = 0; i < 10; ++i)
4         co_yield i;
5 }
6
7 int main() {
8     auto x = foo();
9     std::optional<int> i;
10    while((i = x.get_next()) != std::nullopt)
11        std::cout << *i << std::endl;
12 }
```

generator!

```
1 #include "generator-m.hpp"
2 generator<int> foo() {
3     for(int i = 0; i < 10; ++i)
4         co_yield i;
5 }
6
7 int main() {
8     auto x = foo();
9     std::optional<int> i;
10    while((i = x.get_next()) != std::nullopt)
11        std::cout << *i << std::endl;
12 }
```

generator!

```
1 #include "generator-m.hpp"
2 generator<int> foo() {
3     for(int i = 0; i < 10; ++i)
4         co_yield i;
5 }
6
7 int main() {
8     auto x = foo();
9     std::optional<int> i;
10    while((i = x.get_next()) != std::nullopt)
11        std::cout << *i << std::endl;
12 }
```

generator!

```
1 #include "generator-m.hpp"
2 generator<int> foo() {
3     for(int i = 0; i < 10; ++i)
4         co_yield i;
5 }
6
7 int main() {
8     auto x = foo();
9     std::optional<int> i;
10    while((i = x.get_next()) != std::nullopt)
11        std::cout << *i << std::endl;
12 }
```

more generators!

```
1 #include "generator-m.hpp"
2 generator<int> fib() {
3     int a = 0, b = 1;
4     while(true){
5         co_yield b;
6         int c = a + b;
7         a = b;
8         b = c;
9     };
10 }
```

more generators!

```
1 #include "generator-m.hpp"
2 generator<int> fib() {
3     int a = 0, b = 1;
4     while(true){
5         co_yield b;
6         int c = a + b;
7         a = b;
8         b = c;
9     };
10 }
```

more generators!

```
1 #include "generator-m.hpp"
2 generator<int> fib() {
3     int a = 0, b = 1;
4     while(true){
5         co_yield b;
6         int c = a + b;
7         a = b;
8         b = c;
9     };
10 }
```

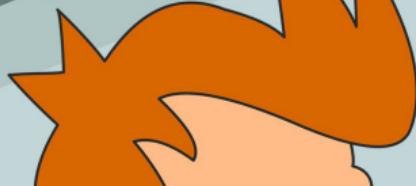
more generators!

```
1 #include "generator-m.hpp"
2 generator<int> fib() {
3     int a = 0, b = 1;
4     while(true){
5         co_yield b;
6         int c = a + b;
7         a = b;
8         b = c;
9     };
10 }
```

more generators!

```
1 #include "generator-m.hpp"
2 generator<int> fib() {
3     int a = 0, b = 1;
4     while(true){
5         co_yield b;
6         int c = a + b;
7         a = b;
8         b = c;
9     };
10 }
```

SHUT UP AND



TAKE MY MONEY!

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     struct promise_type;
4     coroutine_handle<promise_type> handle_;
5     generator(coroutine_handle<promise_type> handle)
6         : handle_{handle}
7     {}
8     ~generator() {
9         if(handle_) handle_.destroy();
10    }
11};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     struct promise_type;
4     coroutine_handle<promise_type> handle_;
5     generator(coroutine_handle<promise_type> handle)
6         : handle_(handle)
7     {}
8     ~generator() {
9         if(handle_) handle_.destroy();
10    }
11};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     struct promise_type;
4     coroutine_handle<promise_type> handle_;
5     generator(coroutine_handle<promise_type> handle)
6         : handle_{handle}
7     {}
8     ~generator() {
9         if(handle_) handle_.destroy();
10    }
11};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     struct promise_type;
4     coroutine_handle<promise_type> handle_;
5     generator(coroutine_handle<promise_type> handle)
6         : handle_{handle}
7     {}
8     ~generator() {
9         if(handle_) handle_.destroy();
10    }
11};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     struct promise_type;
4     coroutine_handle<promise_type> handle_;
5     generator(coroutine_handle<promise_type> handle)
6         : handle_{handle}
7     {}
8     ~generator() {
9         if(handle_) handle_.destroy();
10    }
11};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     //...
4 };
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     // ...
4     std::optional<T> get_next() {
5         handle_.resume();
6         if(handle_.done())
7             return std::nullopt;
8         return handle_.promise().current_value_;
9     }
10};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     // ...
4     std::optional<T> get_next() {
5         handle_.resume();
6         if(handle_.done())
7             return std::nullopt;
8         return handle_.promise().current_value_;
9     }
10};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     // ...
4     std::optional<T> get_next() {
5         handle_.resume();
6         if(handle_.done())
7             return std::nullopt;
8         return handle_.promise().current_value_;
9     }
10};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     // ..
4     std::optional<T> get_next() {
5         handle_.resume();
6         if(handle_.done())
7             return std::nullopt;
8         return handle_.promise().current_value_;
9     }
10};
```

implementing generator - interface

```
1 template <typename T>
2 struct generator {
3     // ..
4     std::optional<T> get_next() {
5         handle_.resume();
6         if(handle_.done())
7             return std::nullopt;
8         return handle_.promise().current_value_;
9     }
10};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

implementing generator - promise

```
1 template <typename T>
2 struct generator<T>::promise_type {
3     T current_value_;
4     auto initial_suspend() { return suspend_always{}; }
5     auto final_suspend() { return suspend_always{}; }
6     void unhandled_exception() { std::terminate(); }
7     auto get_return_object() {
8         return coroutine_handle<promise_type>::from_promise(*this);
9     }
10    auto yield_value(T value) {
11        this->current_value_ = value;
12        return suspend_always{};
13    }
14    void return_void(){}
15};
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> take_at_most_n(generator<T> g, int n)
4 {
5     for(auto&& v : g)
6         if(n-- > 0)
7             co_yield v;
8         else
9             break;
10 }
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> take_at_most_n(generator<T> g, int n)
4 {
5     for(auto&& v : g)
6         if(n-- > 0)
7             co_yield v;
8         else
9             break;
10 }
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> take_at_most_n(generator<T> g, int n)
4 {
5     for(auto&& v : g)
6         if(n-- > 0)
7             co_yield v;
8         else
9             break;
10 }
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> take_at_most_n(generator<T> g, int n)
4 {
5     for(auto&& v : g)
6         if(n-- > 0)
7             co_yield v;
8         else
9             break;
10 }
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> join(generator<T> g1, generator<T>
4   ↪ g2)
5 {
6     for( auto&& v : g1 )
7         co_yield v;
8     for( auto&& v : g2 )
9         co_yield v;
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> join(generator<T> g1, generator<T>
4   ↪ g2)
5 {
6     for( auto&& v : g1 )
7         co_yield v;
8     for( auto&& v : g2 )
9         co_yield v;
```

its composable!

```
1 #include "generator-m.hpp"
2 template<typename T>
3 generator<T> join(generator<T> g1, generator<T>
4   ↪ g2)
5 {
6     for( auto&& v : g1 )
7         co_yield v;
8     for( auto&& v : g2 )
9         co_yield v;
```

we almost have all the powers

- interface type - operate on coroutine
- promise type - control coroutine behaviour

we almost have all the powers

- interface type - operate on coroutine
- promise type - control coroutine behaviour
- awaiter type - control await behaviour

awaiting

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```

awaiting

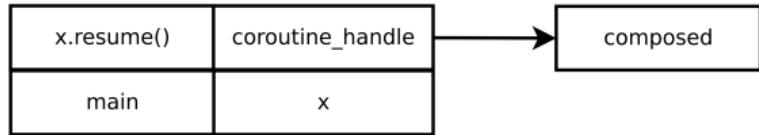
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```

awaiting

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```

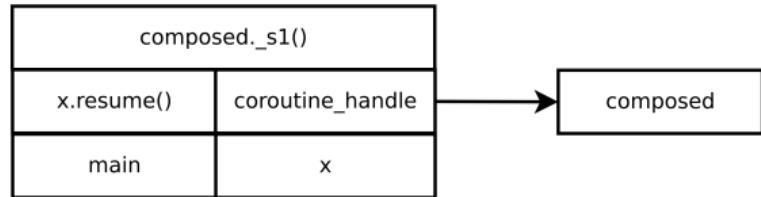
awaiting

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



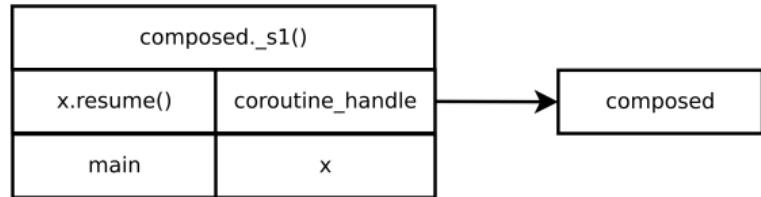
awaiting

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



awaiting

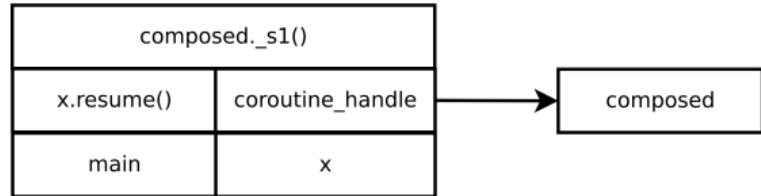
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello

awaiting

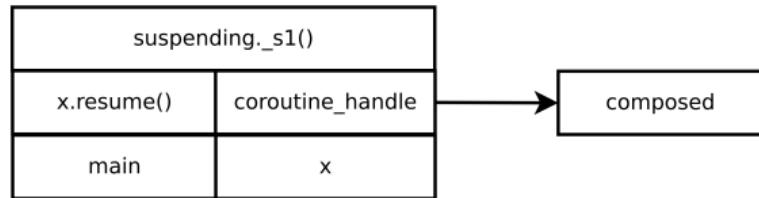
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello

awaiting

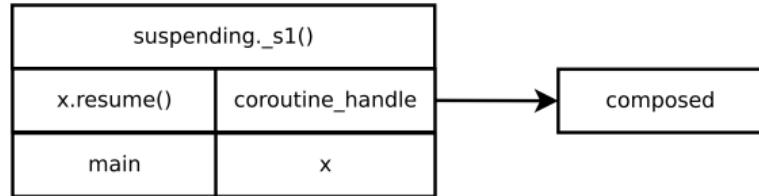
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello

awaiting

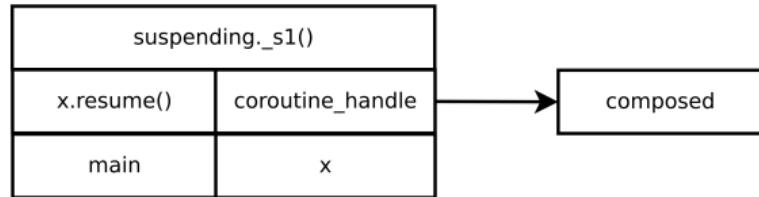
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello,

awaiting

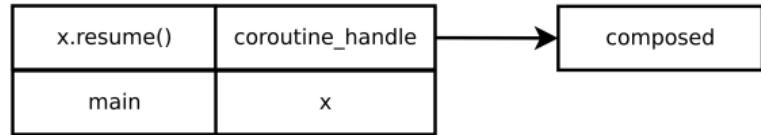
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello,

awaiting

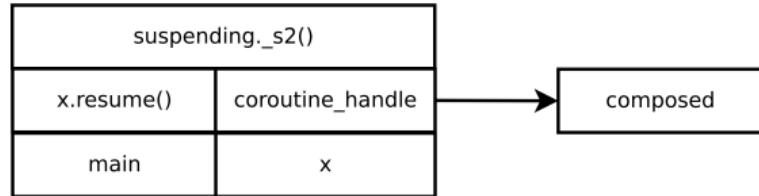
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello,

awaiting

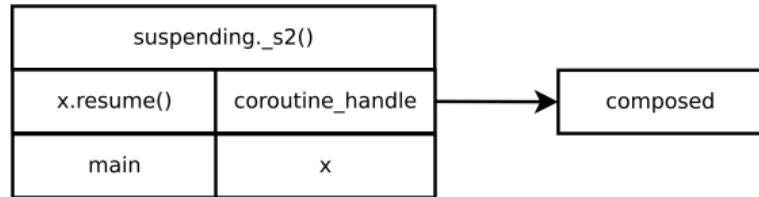
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello,

awaiting

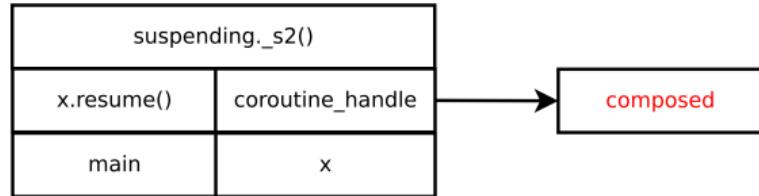
```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello, !

awaiting

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



output: hello, !

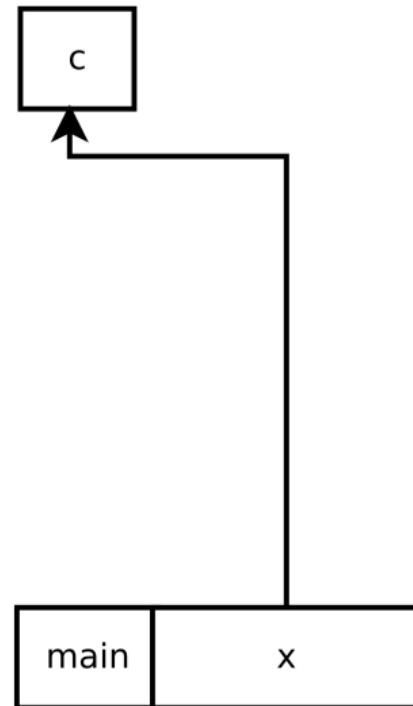
how do we want to compose tasks

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



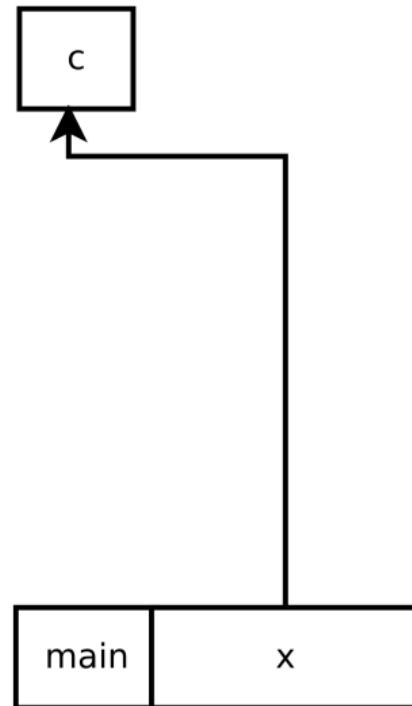
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



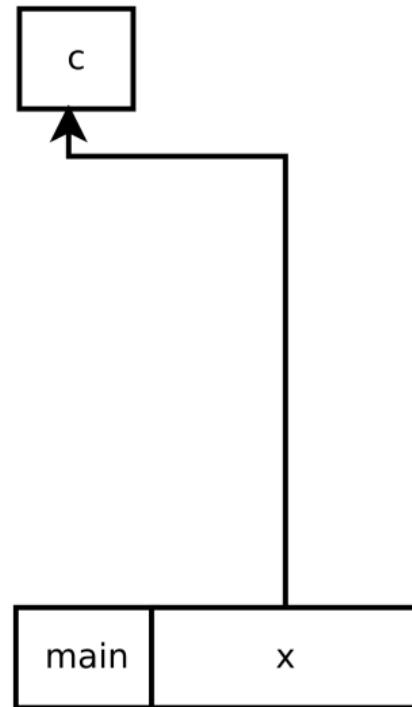
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



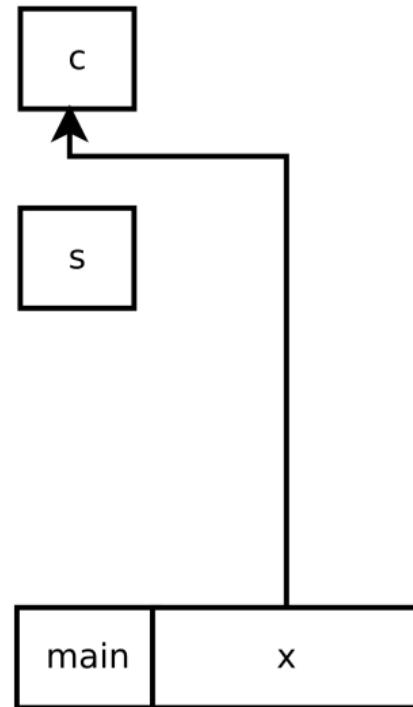
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



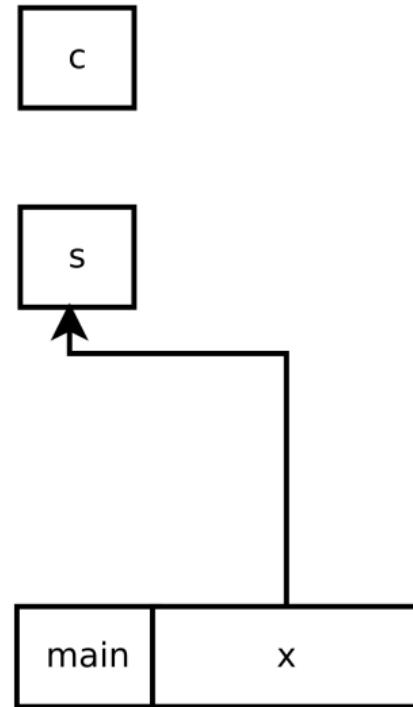
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



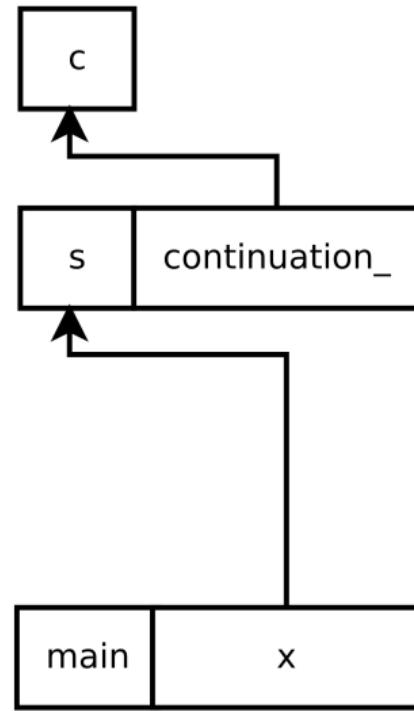
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



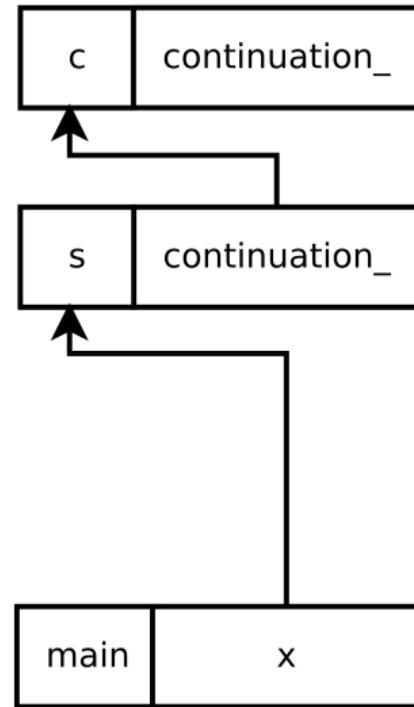
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



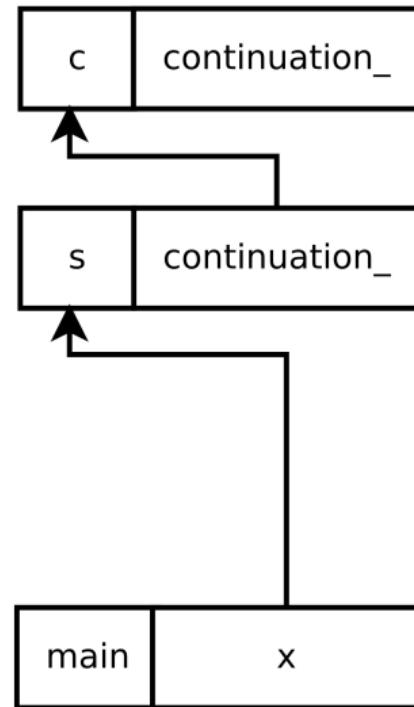
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



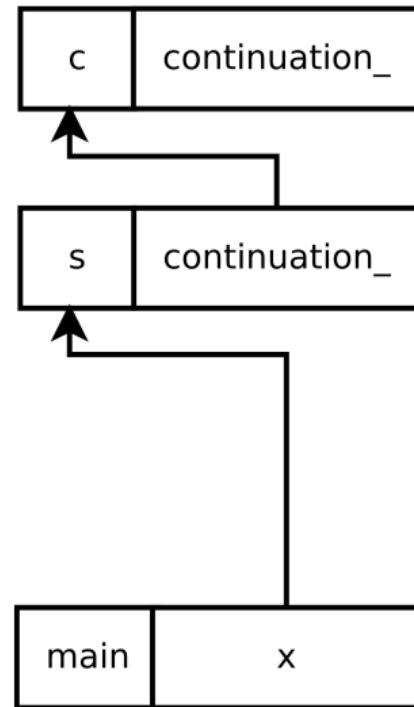
how do we want to compose tasks

```
1 task<void> suspending() {
2     std::cout << ",_";
3     co_await suspend_always{};
4     std::cout << "word";
5 };
6
7 task<void> composed() {
8     std::cout << "hello";
9     co_await suspending();
10    std::cout << "!";
11 };
12
13 int main() {
14     task<void> x = composed();
15     while(not x.is_ready())
16         x.resume();
17 }
```



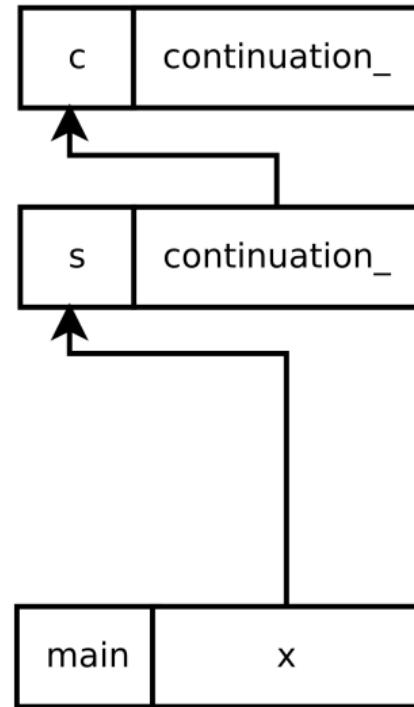
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



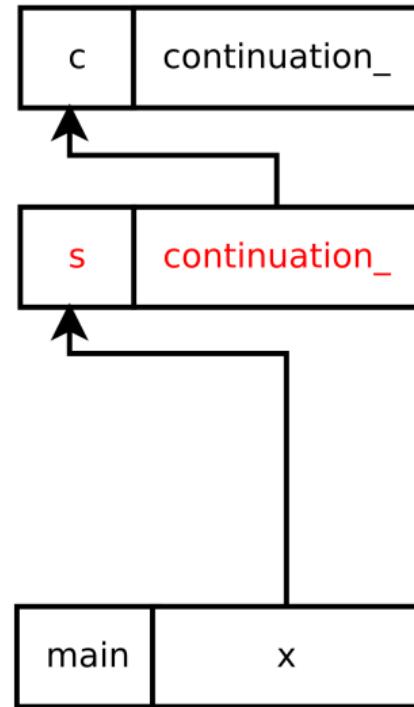
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



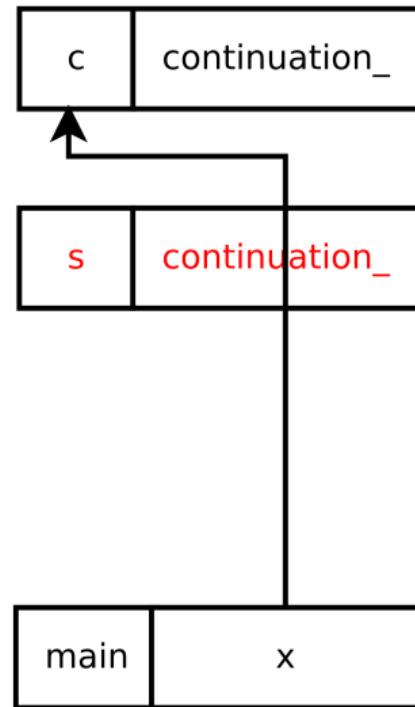
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



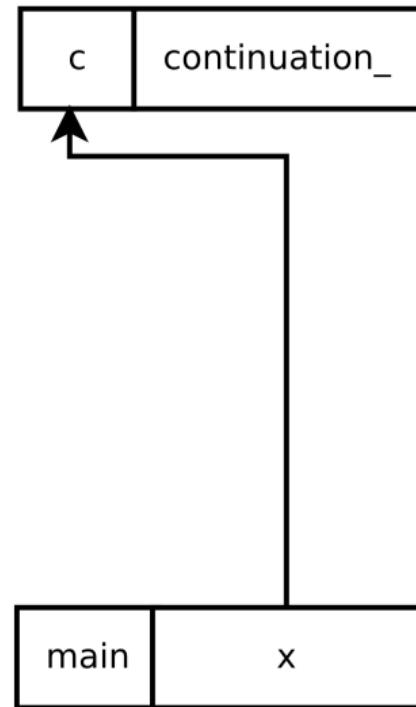
how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



how do we want to compose tasks

```
1 task<void> suspending() {  
2     std::cout << ",_";  
3     co_await suspend_always{};  
4     std::cout << "word";  
5 };  
6  
7 task<void> composed() {  
8     std::cout << "hello";  
9     co_await suspending();  
10    std::cout << "!";  
11};  
12  
13 int main() {  
14     task<void> x = composed();  
15     while(not x.is_ready())  
16         x.resume();  
17 }
```



begin of

slides removed from live presentation

trivial awaitable

```
1 struct suspend_always_
2 {
3     bool await_ready() { return false; }
4     void await_suspend(coroutine_handle<>) {}
5     constexpr void await_resume() {}
6 };
```

trivial awaitable

```
1 struct suspend_always_
2 {
3     bool await_ready() { return false; }
4     void await_suspend(coroutine_handle<>) {}
5     constexpr void await_resume() {}
6 };
```

trivial awaitable

```
1 struct suspend_always_
2 {
3     bool await_ready() { return false; }
4     void await_suspend(coroutine_handle<>) {}
5     constexpr void await_resume() {}
6 };
```

trivial awaitable

```
1 struct suspend_always_
2 {
3     bool await_ready() { return false; }
4     void await_suspend(coroutine_handle<>) {}
5     constexpr void await_resume() {}
6 };
```

make task awaitable

```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```

make task awaitable

```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```

make task awaitable

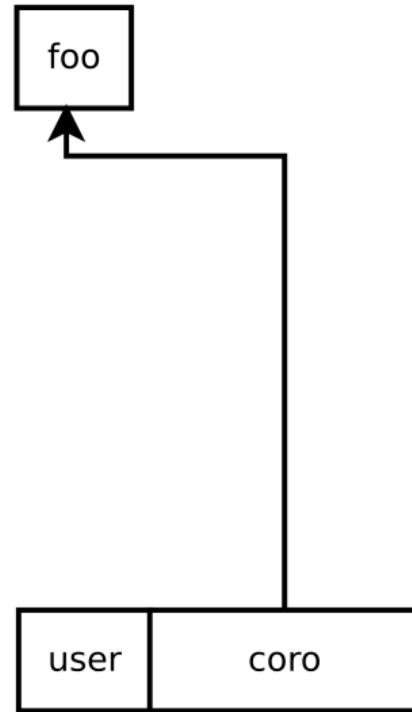
```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```

make task awaitable

```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```

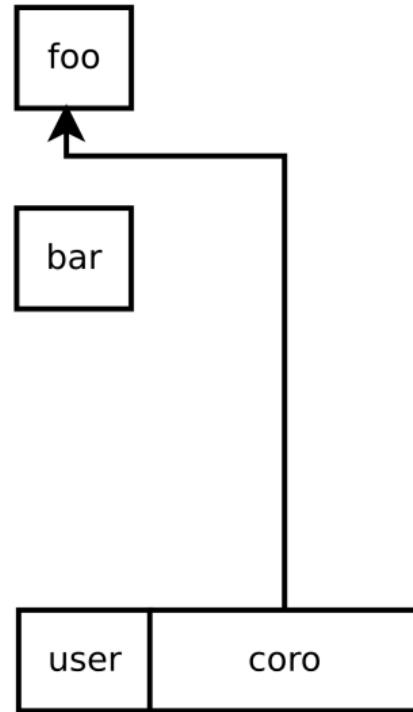
make task awaitable

```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```



make task awaitable

```
1 #include "awaitable-m.hpp"
2 template<typename T>
3 class task_{
4 public:
5     //...
6
7     auto operator co_await() const & noexcept
8     {
9         return awaitable<T>{handle_};
10    }
11
12 private:
13     coroutine_handle<promise<T>> handle_;
14 }
```



our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}(){}
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<void> await_suspend(coroutine_handle<void> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<void> await_suspend(coroutine_handle<void> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

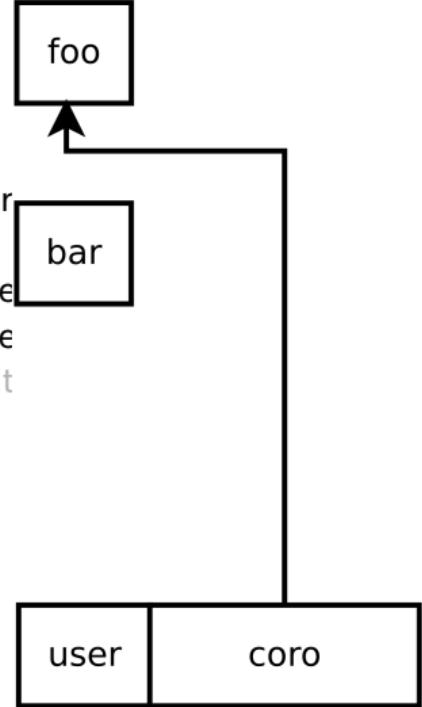
```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<void> await_suspend(coroutine_handle<void> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro)
7     : coro_(coro) {}
8     bool await_ready() { return !coro_.done(); }
9     coroutine_handle<T> await_suspend(coroutine_handle<T> promise)
10    {
11        coro_.promise().set_continuation(await_resume);
12        return coro_;
13    }
14    decltype(auto) await_resume() {
15        return coro_.promise().value_;
16    }
`o) {
```



our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) :
7         coro_(coro) {
8         bool await_ready() { return !coro_.done(); }
9         coroutine_handle<T> await_suspend(coroutine_handle<T> continuation) {
10             coro_.promise().set_continuation(continuation);
11             return coro_;
12         }
13         decltype(auto) await_resume() {
14             return coro_.promise().value();
15         }
16     };

```

The diagram illustrates the state of objects during an await operation. It features three main components represented as boxes:

- A top-level box labeled "foo".
- A middle box labeled "bar".
- A bottom-level box divided into two sections: "user" on the left and "coro" on the right.

An arrow points from the "bar" box down to the "user" section of the bottom box, indicating that the "bar" object is currently being awaited by the "user" section of the "coro" object.

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) :
7         coro_(coro) {
8         bool await_ready() { return !coro_.done(); }
9         coroutine_handle<T> await_suspend(coroutine_handle<T> o) {
10             coro_.promise().set_continuation(await_resume(o));
11             return coro_;
12         }
13         decltype(auto) await_resume() {
14             return coro_.promise().value_;
15         }
16     };

```

```
graph TD; user[user] --> bar[bar]; bar --> foo[foo]; foo --> user;
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) {
7
8         bool await_ready() { return not coro_.done(); }
9         coroutine_handle<void> await_suspend(coroutine_handle<void> continuation) {
10             coro_.promise().set_continuation(continuation);
11             return coro_;
12         }
13         decltype(auto) await_resume() {
14             return coro_.promise().value_;
15         }
16     };

```

The diagram illustrates the state of objects during an await operation. It shows three main components:

- A rectangular box labeled "foo" at the top.
- A rectangular box labeled "bar" positioned below "foo".
- A larger rectangular box at the bottom divided into two horizontal sections: "user" on the left and "coro" on the right.

An arrow points from the "bar" box down to the "coro" section of the bottom box, indicating that the "bar" object is currently suspended within the "coro" context.

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) :
7         coro_(coro) {
8         bool await_ready() { return !coro_.done(); }
9         coroutine_handle<void> await_suspend(coroutine_handle<void> o) {
10             coro_.promise().set_continuation(await_resume(o));
11             return coro_;
12         }
13         decltype(auto) await_resume() {
14             return coro_.promise().value_;
15         }
16     };
}
```

```
graph TD; user --- bar[bar]; bar --- foo[foo]; foo --> user
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro)
7     : coro_(coro) {}
8     bool await_ready() { return !coro_.done(); }
9     coroutine_handle<void> await_suspend(coroutine_handle<void> promise_awaitable)
10    {
11        coro_.promise().set_continuation(await_suspend);
12        return coro_;
13    }
14    decltype(auto) await_resume() {
15        return coro_.promise().value_;
16    }
}
```

The diagram illustrates the state of objects during an await operation. It features four rectangular boxes: 'foo' at the top, 'bar' below it, a large vertical rectangle labeled 'user' on its left side, and a horizontal rectangle labeled 'coro' on its right side. An arrow points from 'bar' up towards 'foo'. The 'user' box is positioned to the left of the 'coro' box, with a vertical line separating them.

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) {
7
8         bool await_ready() { return not coro_.done(); }
9         coroutine_handle<void> await_suspend(coroutine_handle<void> user) {
10             coro_.promise().set_continuation(await_resume);
11             return coro_;
12         }
13         decltype(auto) await_resume() {
14             return coro_.promise().value_;
15         }
16     };
}
```

The diagram illustrates the state of objects during an await operation. It shows three main components:

- A rectangular box labeled "foo" at the top left.
- A rectangular box labeled "bar" at the middle right, with an upward-pointing arrow originating from its bottom edge.
- A horizontal rectangular bar at the bottom right divided into two equal sections: "user" on the left and "coro" on the right.

The arrow from "bar" points towards the "user" section of the bar, indicating that the continuation of the awaitable object (represented by "bar") is being passed to the user's code (represented by "user").

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

our awaitable

```
1 #include "awaitable_promise-m.hpp"
2 template <typename T>
3 struct awaitable
4 {
5     coroutine_handle<promise<T>> coro_;
6     awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};
7
8     bool await_ready() { return not coro_.done(); }
9     coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
10         coro_.promise().set_continuation(awaitingCoro);
11         return coro_;
12     }
13     decltype(auto) await_resume() {
14         return coro_.promise().value_;
15     }
16 };
```

setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

setting continuation

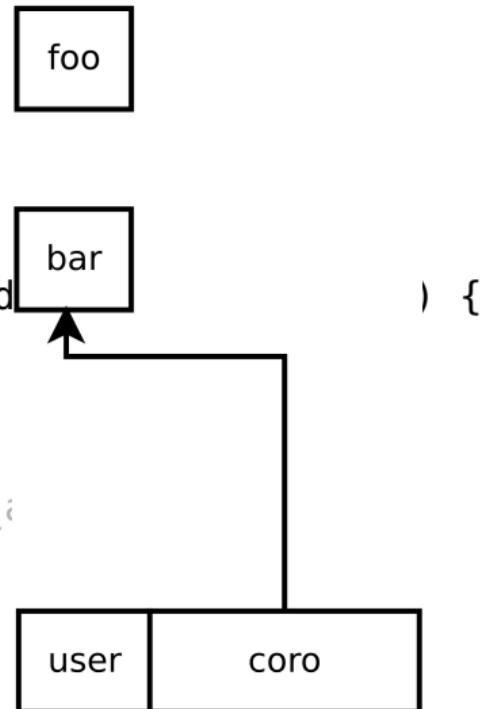
```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<T> continuation_;
7     void set_continuation(coroutine_handle<T> continuation) {
8         continuation_ = continuation;
9     }
10
11    auto final_suspend() { return final_suspend(); }
12
13 };
14
```



setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<T> continuation_;
7     void set_continuation(coroutine_handle<T> continuati
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_suspen
12
13 };
14 }
```

```
graph TD; user --- coro[coro]; coro --- bar[bar]; bar --- foo[foo]
```

setting continuation

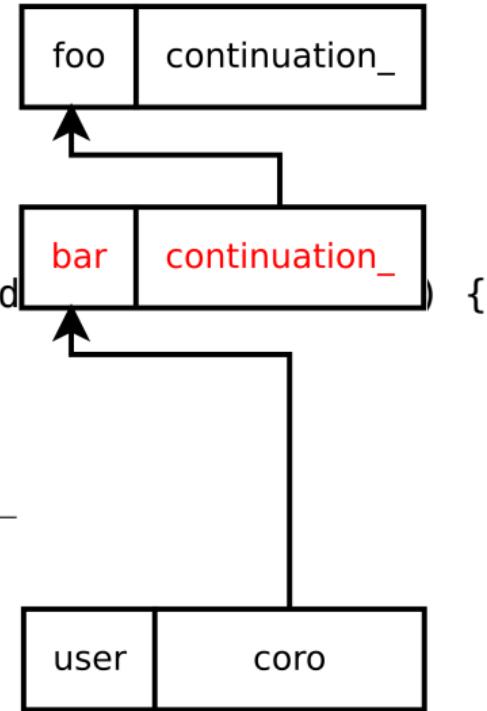
```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<> continuation_;
7     void set_continuation(coroutine_handle<> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_awaitable{}; }
12
13 };
14
```

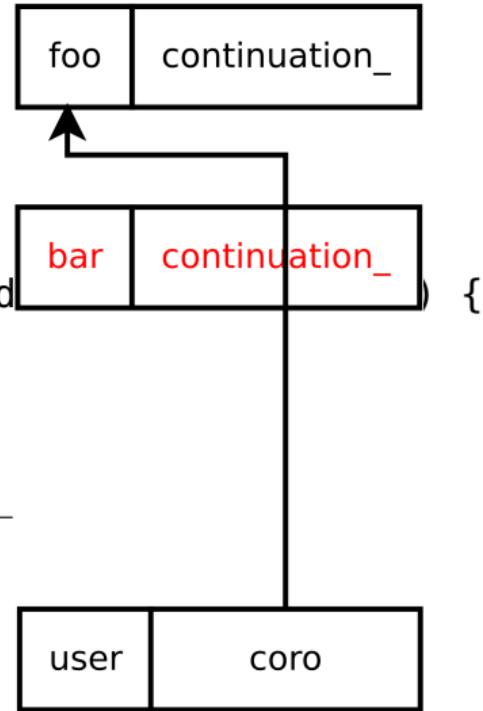
setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<T> continuation_;
7     void set_continuation(coroutine_handle<T> continu
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_
12
13 };
14 }
```



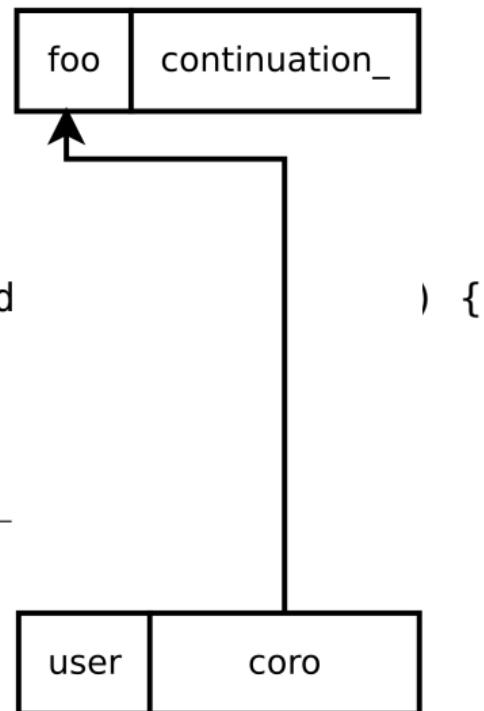
setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<T> continuation_;
7     void set_continuation(coroutine_handle<T> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_
12
13 };
14 }
```



setting continuation

```
1 #include "final_awaitable-m.hpp"
2 template <typename T>
3 struct promise {
4     // ...
5
6     coroutine_handle<T> continuation_;
7     void set_continuation(coroutine_handle<T> continuation) {
8         continuation_ = continuation;
9     }
10
11     auto final_suspend() { return final_
12
13 };
14 }
```



final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false;};
4     void await_resume() noexcept{}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

final suspend - final awaitable

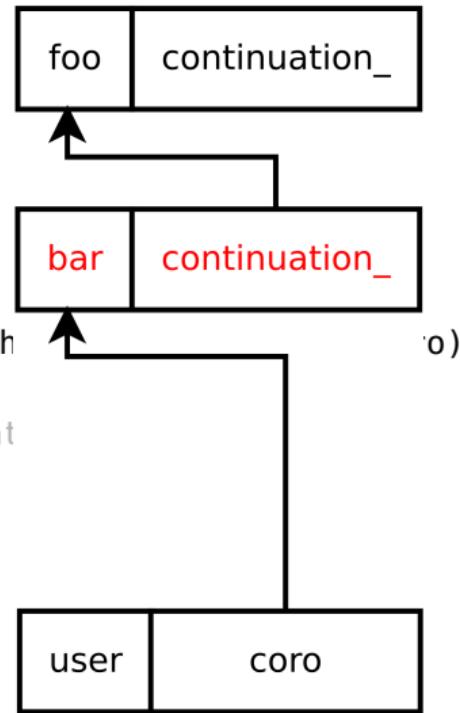
```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

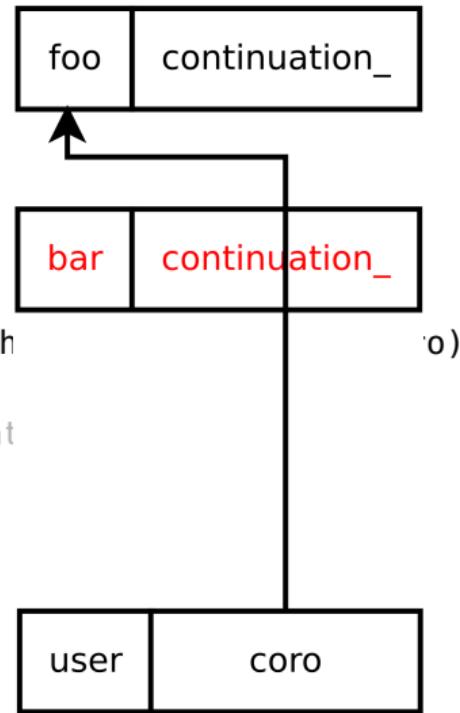
final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; };
4     void await_resume() noexcept{};
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         return finalizedCoro.promise().continuat
10    }
11};
```



final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         return finalizedCoro.promise().continuat
10    }
11};
```

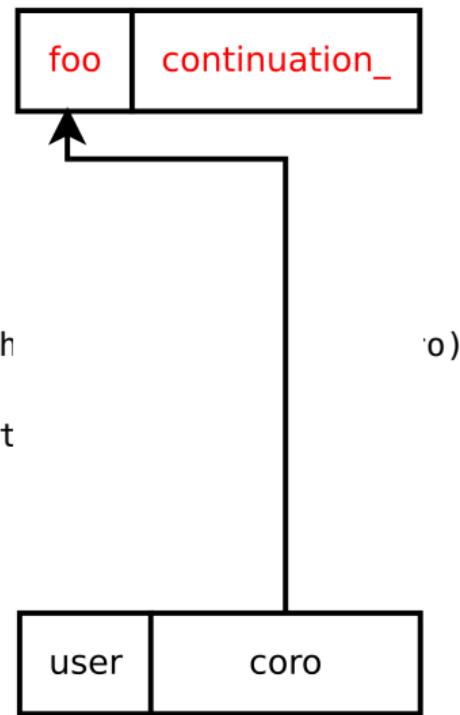


final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

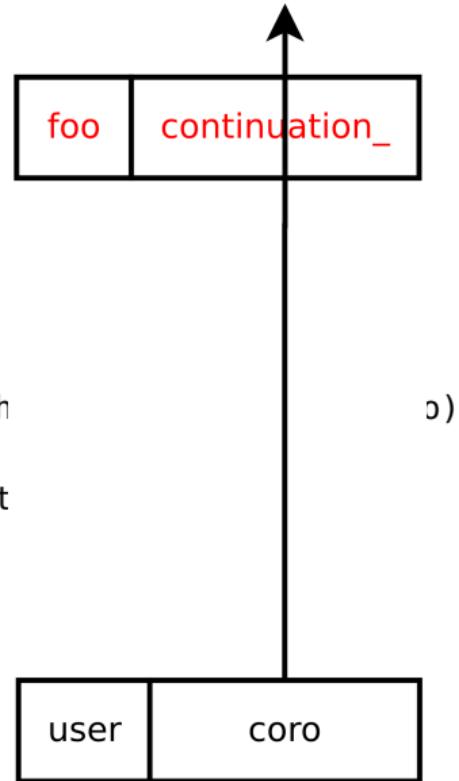
final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         return finalizedCoro.promise().continuat
10    }
11};
```



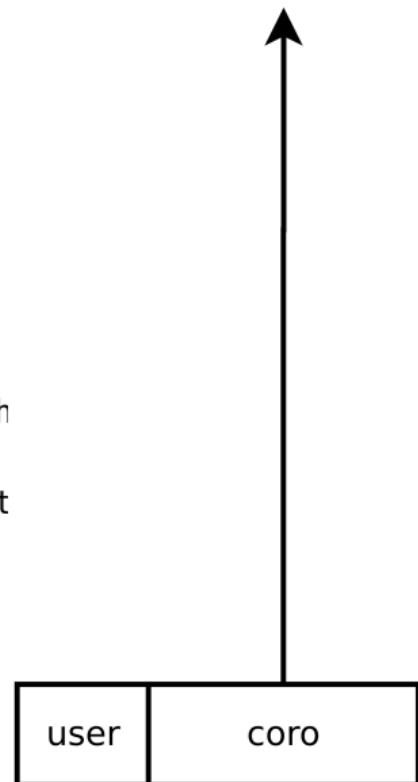
final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         return finalizedCoro.promise().continuat
10    }
11};
```



final_suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         return finalizedCoro.promise().continuat
10    }
11};
```



final suspend - final awaitable

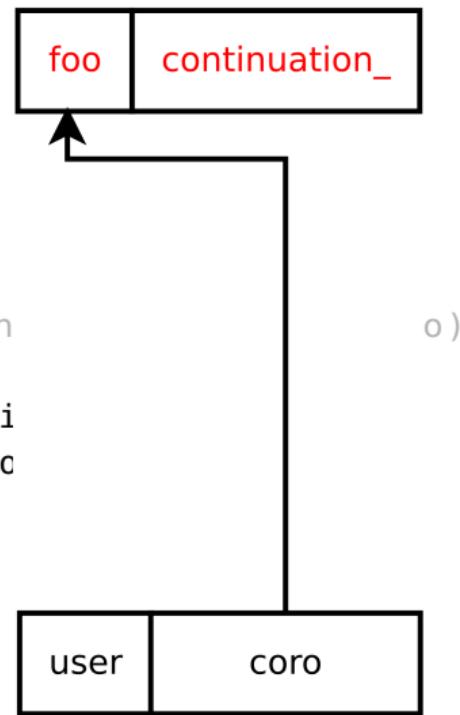
```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         return finalizedCoro.promise().continuation_;
10    }
11};
```

final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         auto& continuation = finalizedCoro.promise().continuation_;
10        return continuation ? continuation : noop_coroutine;
11    }
12};
```

final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_h
8     {
9         auto& continuation = finalizedCoro.promise_
10        return continuation ? continuation : noc
11    }
12};
```



final suspend - final awaitable

```
1 struct final_awaitable
2 {
3     bool await_ready() { return false; }
4     void await_resume() noexcept {}
5
6     template <typename P>
7     coroutine_handle<P> await_suspend(coroutine_handle<P> finalizedCoro)
8     {
9         auto& continuation = finalizedCoro.promise().continuation_;
10        return continuation ? continuation : noop_coroutine;
11    }
12};
```



take away

take away

- coroutine are flexible

take away

- coroutine are flexible
- very flexible

take away

- coroutine are flexible
- very flexible
- so flexible

take away

- coroutine are flexible
- very flexible
- so flexible
- with customization point

take away

- coroutine are flexible
- very flexible
- so flexible
- with customization point
- and more customization points

end of

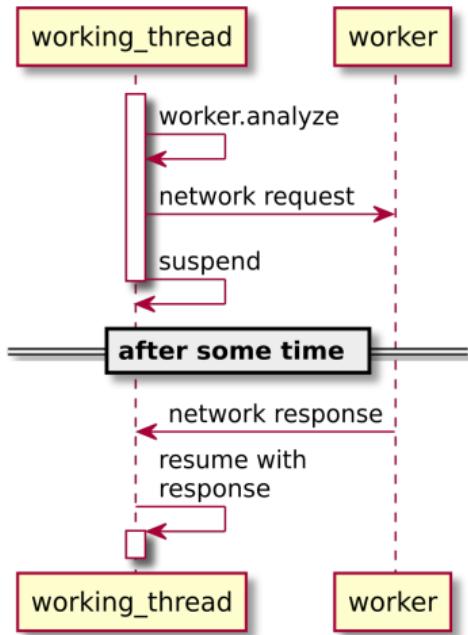
slides removed from live presentation

confession

confession



back to motivation



back to motivation

```
1 using namespace async;
2 void perform_experiment(experiment const& ex, session& s) {
3     std::vector<std::future<sample_result>> worker_futures;
4     for(auto const& s: ex.samples)
5         worker_futures.push_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_futures)
9         results.push_back(f.get());
10    s.respond(results).get();
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

back to motivation

```
1 using namespace coro;
2 eager_task<void> perform_experiment(experiment ex, session& s) {
3     std::vector<eager_task<sample_result>> worker_tasks;
4     for(auto& s: ex.samples)
5         worker_tasks.emplace_back(worker.analyze(s));
6
7     std::vector<sample_result> results;
8     for (auto& f : worker_tasks)
9         results.push_back(co_await f);
10    co_await s.respond(results);
11 }
```

any asynchronous code

any asynchronous code

- I/O access

any asynchronous code

- I/O access
- networking

any asynchronous code

- I/O access
- networking
- disks

any asynchronous code

- I/O access
- networking
- disks
- inter process communication

any asynchronous code

- I/O access
- networking
- disks
- inter process communication
- data bases

abusing!

abusing!

- sick optimization techniques

abusing!

- sick optimization techniques
- data structures!

?