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

```
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking app

```
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(
              experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking app

```
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(
                experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking app

```
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking app

```cpp
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking app

```
std::pair<blocking::session, experiment> receive();

void program_main_loop()
{
    thread_pool tp;
    while(true)
    {
        auto [session, experiment] = receive();
        tp.execute(
            [s = std::move(session), ex = std::move(experiment)]() {
                perform_experiment(ex, s);
            });
    }
}
```

my networking algorithm

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

my networking algorithm

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

my networking algorithm

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

my networking algorithm

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

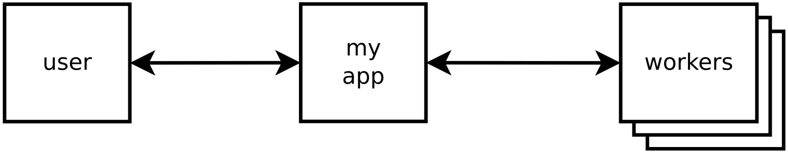
my networking algorithm

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

my networking algorithm

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

my networking algorithm



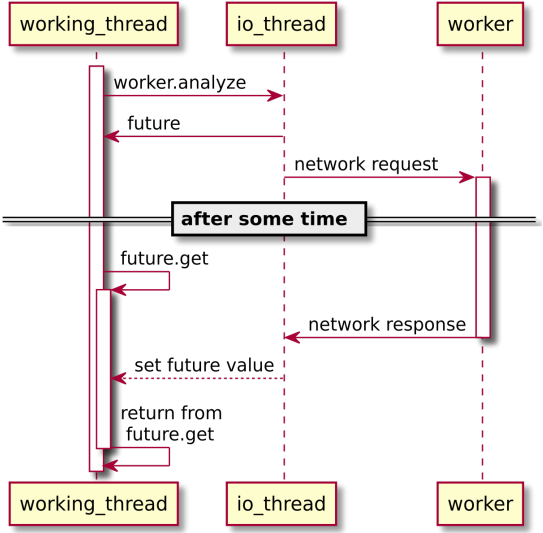
problem

```cpp
using namespace blocking;
void perform_experiment(experiment const& ex, session& s)
{
    std::vector<sample_result> results;
    for( auto const& sample : ex.samples)
        results.push_back(worker.analyze(sample));
    s.respond(results);
};
```

problem

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

partial solution



partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

partial solution

```
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

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 sacrafice readability

what I want from solution

- reduce the amount of IO bound threads
- dont sacrafice 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

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

lets start with subroutines

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

lets start with subroutines

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

lets start with subroutines

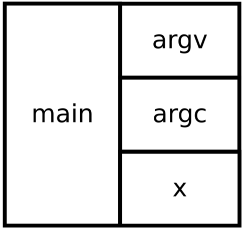
```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

lets start with subroutines

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

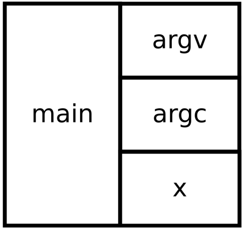
function call procedure

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```



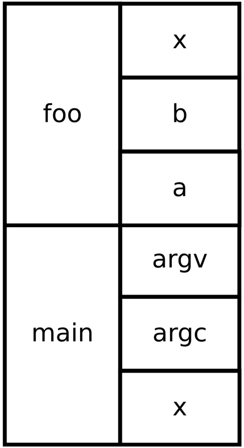
function call procedure

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```



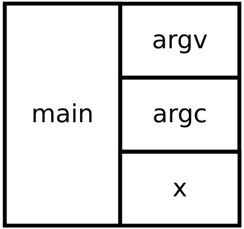
function call procedure

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```



function call procedure

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```



function call procedure

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

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

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
int foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    int x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    return x;
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    return x;
}
```

coroutine

```cpp
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    x.resume();
    return x.result();
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    x.resume();
    return x.result();
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    x.resume();
    return x.result();
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    x.resume();
    return x.result();
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    x.resume();
    x.resume();
    return x.result();
}
```

coroutine

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```

coroutine

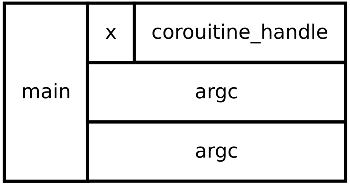
```cpp
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```

coroutine call procedure

```cpp
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```

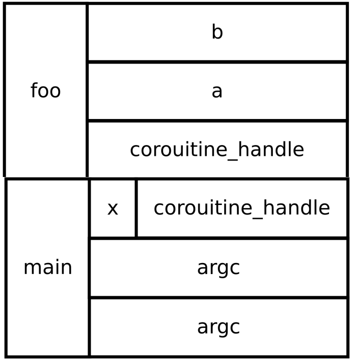
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



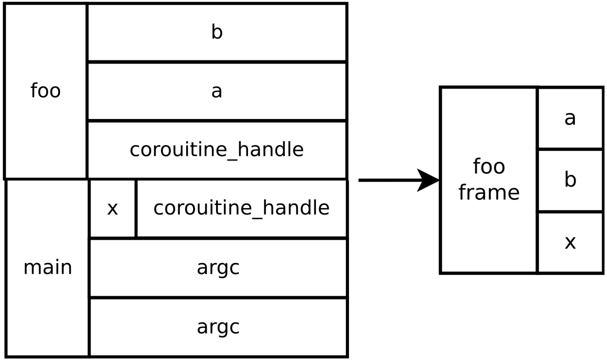
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



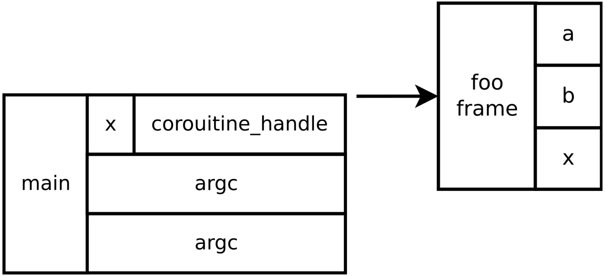
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



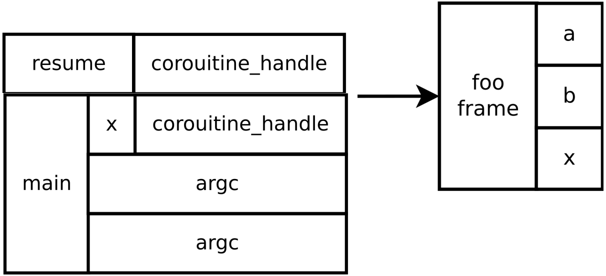
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



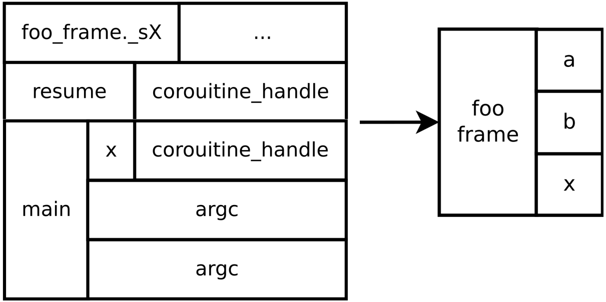
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



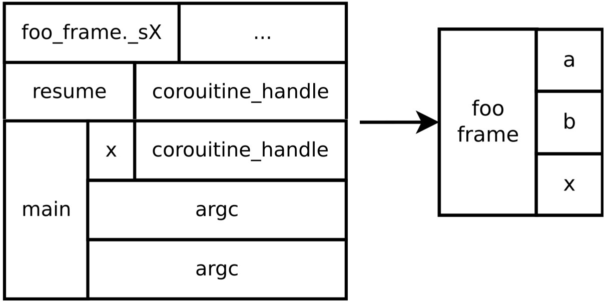
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



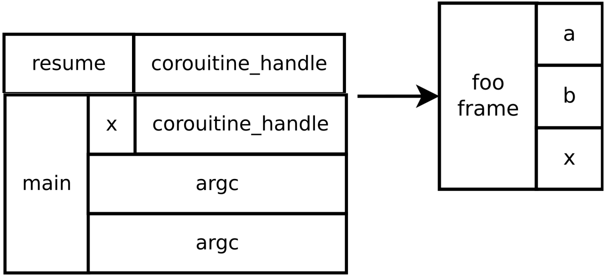
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



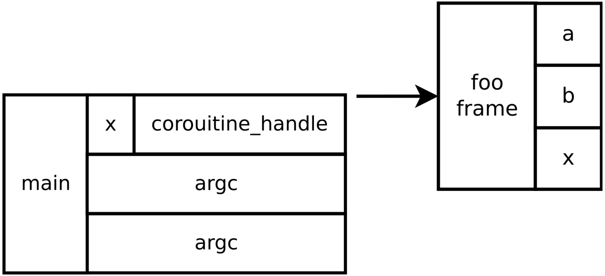
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



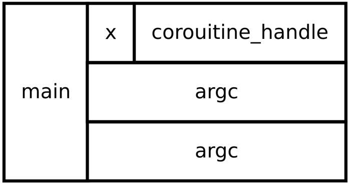
coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```



coroutine call procedure

```
task<int> foo(int a, int b) {
    int x = a + b;
    co_await suspend_always{};
    co_return x;
}

int main() {
    task<int> x = foo(5, 7);
    while(not x.is_ready())
        x.resume();
    return x.result();
}
```

previous slide was nightmare to make

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

previous slide was nightmare to make

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

previous slide was nightmare to make

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

previous slide was nightmare to make

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

previous slide was nightmare to make

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

previous slide was nightmare to make

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

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

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  int a;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  int a;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  int a;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ int b = 15; }
  int a;
};
```

coroutine transformation

```cpp
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```cpp
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ int b = 15; }
  int a;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  int a, b;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){ a = b * a; }
  int a, b;
};
```

coroutine transformation

```cpp
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```cpp
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){ a = b * a; }
  int a, b;
};
```

coroutine transformation

```cpp
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```cpp
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){
    a = b * a;
    int c = b * 2;
  }
  int a, b;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){
    a = b * a;
    int c = b * 2;
  }
  int a, b;
};
```

coroutine transformation

```cpp
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```cpp
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){
    a = b * a;
    c = b * 2;
  }
  int a, b, c;
};
```

coroutine transformation

```
task<int> bar(int a)
{
  int b = 15;
  co_await suspend_always{};
  a = b * a;
  int c = b * 2;
  co_await suspend_always{};
  co_return a + b + c;
}
```

```
struct _bar{
  _bar(int a_) : a{a_}{};
  void _s1(){ b = 15; }
  void _s2(){
    a = b * a;
    c = b * 2;
  }
  void _s3(){
    promise->return_value(
      a + b + c);
  }
  int a, b, c, v;
  promise_type* promise;
};
```

coroutine – recap

coroutine – recap

- coroutine body is transformed into coroutnine frame

coroutine – recap

- coroutine body is transformed into coroutnine 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

```cpp
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – task

```cpp
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};

```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};
```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};

```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};
```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};

```

implementing coroutine type – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};

```

implementing coroutine type – usage

```cpp
#include "implementing_simple_task_task-m.hpp"
task_ foo()
{
    std::cout << "I am ..." << std::endl;
    co_await suspend_always{};
    std::cout << "...a COROUTINE!!!" << std::endl;
}
```

implementing coroutine type – usage

```cpp
#include "implementing_simple_task_task-m.hpp"
task_ foo()
{
    std::cout << "I am ..." << std::endl;
    co_await suspend_always{};
    std::cout << "... a COROUTINE!!!" << std::endl;
}
```

implementing coroutine type – usage

```
#include "implementing_simple_task_task_m.hpp"
task_ foo()
{
    std::cout << "I am ..." << std::endl;
    co_await suspend_always{};
    std::cout << "...a COROUTINE!!!" << std::endl;
}
```

implementing coroutine type – usage

```
#include "implementing_simple_task_task-m.hpp"
task_ foo()
{
    std::cout << "I am ..." << std::endl;
    co_await suspend_always{};
    std::cout << "...a COROUTINE!!!" << std::endl;
}
```

implementing coroutine type – usage

```cpp
#include "implementing_simple_task_task-m.hpp"
task_ foo()
{
    std::cout << "I␣am␣..." << std::endl;
    co_await suspend_always{};
    std::cout << "...a␣COROUTINE!!!" << std::endl;
    co_return;
}
```

implementing coroutine type – usage

```cpp
#include "implementing_simple_task_task-m.hpp"
task_ foo()
{
    std::cout << "I am ..." << std::endl;
    co_await suspend_always{};
    std::cout << "...a COROUTINE!!!" << std::endl;
    co_return 5; // error, no return_value(int)
                 // in promise type
}
```

returning value – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};
```

returning value – promise

```
struct promise {
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void return_void() {}
  void unhandled_exception() {
    std::terminate();
  }
};
```

returning value – promise

```
struct promise {
  //...
  void return_void() {}
};
```

returning value – promise

```
struct promise {
  // ...
  void return_void() {}
};
```

returning value – promise

```
template <typename T>
struct promise {
  T value_;
  // ...
  void return_void() {}
};

```

returning value – promise

```
template <typename T>
struct promise {
  T value_;
  //...
  void return_void() {}
};

```

returning value – promise

```cpp
template <typename T>
struct promise {
  T value_;
  //...
  void return_value(T t) { value_ = t; }
};

```

returning value – promise

```
template <typename T>
struct promise {
  T value_;
  auto get_return_object() {
    return coroutine_handle<promise>::from_promise(*this);
  }
  auto initial_suspend() { return suspend_always(); }
  auto final_suspend() { return suspend_always(); }
  void unhandled_exception() {
    std::terminate();
  }
  void return_value(T t) { value_ = t; }
};

```

returning value – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

returning value – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  task_(coroutine_handle<promise> handle)
    : handle_(handle)
  {}
  ~task_() { handle_.destroy(); }
private:
  coroutine_handle<promise> handle_;
};
```

returning value – task

```cpp
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  //...
private:
  coroutine_handle<promise> handle_;
};
```

returning value – task

```
#include "implementing_simple_task_promise-m.hpp"
class task_{
public:
  using promise_type = promise;
  // ...
private:
  coroutine_handle<promise> handle_;
};
```

returning value – task

```cpp
#include "implementing_returning_task_promise-m.hpp"
template<typename T>
class task_{
public:
  using promise_type = promise<T>;
  //...
private:
  coroutine_handle<promise<T>> handle_;
};
```

returning value – task

```cpp
#include "implementing_returning_task_promise-m.hpp"
template<typename T>
class task_{
public:
  using promise_type = promise<T>;
  //...
  T value() { return handle_.promise().value_; }
private:
  coroutine_handle<promise<T>> handle_;
};
```

using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    x.??;
    return x.value();
}
```

using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    x.??;
    return x.value();
}
```

using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    x.??;
    return x.value();
}
```

using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    x.??;
    return x.value();
}
```

usable task

```
#include "implementing_returning_task_promise-m.hpp"
template<typename T>
class task_{
public:
  using promise_type = promise<T>;
  //...
  T value() { return handle_.promise().value_; }
private:
  coroutine_handle<promise<T>> handle_;
};
```

usable task

```
#include "implementing_returning_task_promise-m.hpp"
template<typename T>
class task_{
public:
  using promise_type = promise<T>;
  // ...
  T value() { return handle_.promise().value_; }
  T resume() { return handle_.resume(); }
private:
  coroutine_handle<promise<T>> handle_;
};
```

usable task

```
#include "implementing_returning_task_promise-m.hpp"
template<typename T>
class task_{
public:
  using promise_type = promise<T>;
  // ...
  T value() { return handle_.promise().value_; }
  T resume() { return handle_.resume(); }
  T is_ready() { return handle_.done(); }
private:
  coroutine_handle<promise<T>> handle_;
};
```

actual using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    x.??;
    return x.value();
}
```

actual using from subroutine

```
#include "using_returning_task-m.hpp"
int main()
{
    auto x = foo();
    while( not x.is_ready() )
        x.resume();
    return x.value();
}
```

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

```
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

prefetching

```cpp
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

prefetching

```
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

prefetching

```
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

prefetching

```
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

prefetching

```
struct data {};
int costly_calculation(data&);
task<int> prefetched_execution(data* d)
{
    __buildin_prefetch(d, 1, 1);
    co_await suspend_always{};
    co_return costly_calculation(*d);
};
```

generator!

```
#include "generator-m.hpp"
generator<int> foo() {
    for(int i = 0; i < 10; ++i)
        co_yield i;
}

int main() {
    auto x = foo();
    std::optional<int> i;
    while((i = x.get_next()) != std::nullopt)
        std::cout << *i << std::endl;
}
```

generator!

```
#include "generator-m.hpp"
generator<int> foo() {
    for(int i = 0; i < 10; ++i)
        co_yield i;
}

int main() {
    auto x = foo();
    std::optional<int> i;
    while((i = x.get_next()) != std::nullopt)
        std::cout << *i << std::endl;
}
```

generator!

```
#include "generator-m.hpp"
generator<int> foo() {
    for(int i = 0; i < 10; ++i)
        co_yield i;
}

int main() {
    auto x = foo();
    std::optional<int> i;
    while((i = x.get_next()) != std::nullopt)
        std::cout << *i << std::endl;
}
```

generator!

```
#include "generator-m.hpp"
generator<int> foo() {
    for(int i = 0; i < 10; ++i)
        co_yield i;
}

int main() {
    auto x = foo();
    std::optional<int> i;
    while((i = x.get_next()) != std::nullopt)
        std::cout << *i << std::endl;
}
```

generator!

```
#include "generator-m.hpp"
generator<int> foo() {
    for(int i = 0; i < 10; ++i)
        co_yield i;
}

int main() {
    auto x = foo();
    std::optional<int> i;
    while((i = x.get_next()) != std::nullopt)
        std::cout << *i << std::endl;
}
```

more generators!

```cpp
#include "generator-m.hpp"
generator<int> fib() {
    int a = 0, b = 1;
    while(true){
        co_yield b;
        int c = a + b;
        a = b;
        b = c;
    };
}
```

more generators!

```
#include "generator-m.hpp"
generator<int> fib() {
    int a = 0, b = 1;
    while(true){
        co_yield b;
        int c = a + b;
        a = b;
        b = c;
    };
}
```

more generators!

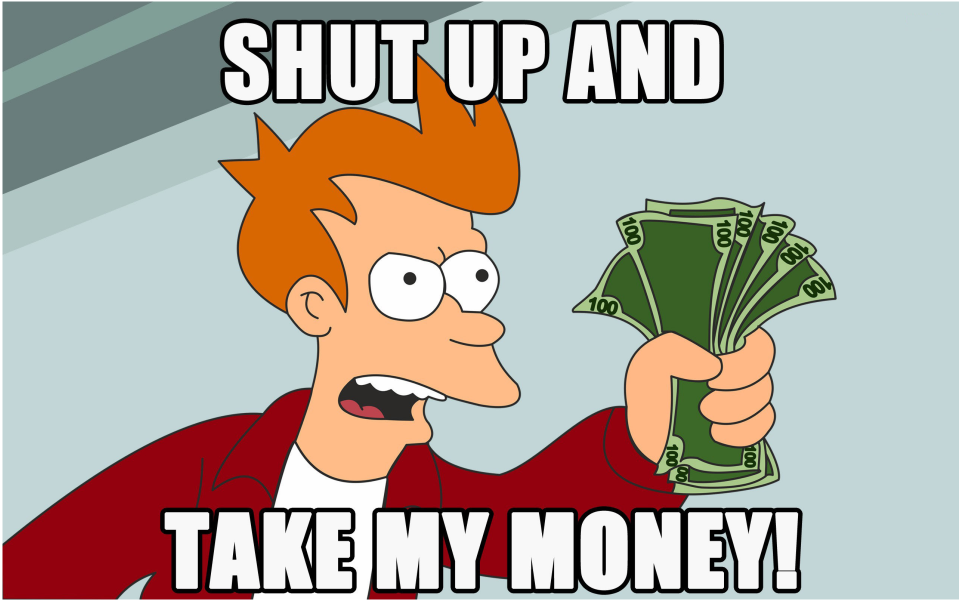
```cpp
#include "generator-m.hpp"
generator<int> fib() {
    int a = 0, b = 1;
    while(true){
        co_yield b;
        int c = a + b;
        a = b;
        b = c;
    };
}
```

more generators!

```cpp
#include "generator-m.hpp"
generator<int> fib() {
    int a = 0, b = 1;
    while(true){
        co_yield b;
        int c = a + b;
        a = b;
        b = c;
    };
}
```

more generators!

```
#include "generator-m.hpp"
generator<int> fib() {
    int a = 0, b = 1;
    while(true){
        co_yield b;
        int c = a + b;
        a = b;
        b = c;
    };
}
```


SHUT UP AND
TAKE MY MONEY!

implementing generator - interface

```
template <typename T>
struct generator {
    struct promise_type;
    coroutine_handle<promise_type> handle_;
    generator(coroutine_handle<promise_type> handle)
        : handle_{handle}
    {}
    ~generator() {
        if(handle_) handle_.destroy();
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    struct promise_type;
    coroutine_handle<promise_type> handle_;
    generator(coroutine_handle<promise_type> handle)
        : handle_{handle}
    {}
    ~generator() {
        if(handle_) handle_.destroy();
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    struct promise_type;
    coroutine_handle<promise_type> handle_;
    generator(coroutine_handle<promise_type> handle)
        : handle_{handle}
    {}
    ~generator() {
        if(handle_) handle_.destroy();
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    struct promise_type;
    coroutine_handle<promise_type> handle_;
    generator(coroutine_handle<promise_type> handle)
        : handle_{handle}
    {}
    ~generator() {
        if(handle_) handle_.destroy();
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    struct promise_type;
    coroutine_handle<promise_type> handle_;
    generator(coroutine_handle<promise_type> handle)
        : handle_{handle}
    {}
    ~generator() {
        if(handle_) handle_.destroy();
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    //...
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    // ..
    std::optional<T> get_next() {
        handle_.resume();
        if(handle_.done())
            return std::nullopt;
        return handle_.promise().current_value_;
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    // ..
    std::optional<T> get_next() {
        handle_.resume();
        if(handle_.done())
            return std::nullopt;
        return handle_.promise().current_value_;
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    // ..
    std::optional<T> get_next() {
        handle_.resume();
        if(handle_.done())
            return std::nullopt;
        return handle_.promise().current_value_;
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    // ..
    std::optional<T> get_next() {
        handle_.resume();
        if(handle_.done())
            return std::nullopt;
        return handle_.promise().current_value_;
    }
};
```

implementing generator - interface

```
template <typename T>
struct generator {
    // ..
    std::optional<T> get_next() {
        handle_.resume();
        if(handle_.done())
            return std::nullopt;
        return handle_.promise().current_value_;
    }
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

implementing generator - promise

```
template <typename T>
struct generator<T>::promise_type {
    T current_value_;
    auto initial_suspend() { return suspend_always{}; }
    auto final_suspend() { return suspend_always{}; }
    void unhandled_exception() {std::terminate();}
    auto get_return_object() {
        return coroutine_handle<promise_type>::from_promise(*this);
    }
    auto yield_value(T value) {
        this->current_value_ = value;
        return suspend_always{};
    }
    void return_void(){};
};
```

its composable!

```cpp
#include "generator-m.hpp"
template<typename T>
generator<T> take_at_most_n(generator<T> g,int n)
{
    for(auto&& v : g)
        if(n-- > 0)
            co_yield v;
        else
            break;
}
```

its composable!

```cpp
#include "generator-m.hpp"
template<typename T>
generator<T> take_at_most_n(generator<T> g,int n)
{
	for(auto&& v : g)
		if(n-- > 0)
			co_yield v;
		else
			break;
}
```

its composable!

```
#include "generator-m.hpp"
template<typename T>
generator<T> take_at_most_n(generator<T> g,int n)
{
    for(auto&& v : g)
        if(n-- > 0)
            co_yield v;
        else
            break;
}
```

its composable!

```
#include "generator-m.hpp"
template<typename T>
generator<T> take_at_most_n(generator<T> g,int n)
{
    for(auto&& v : g)
        if(n-- > 0)
            co_yield v;
        else
            break;
}
```

its composable!

```cpp
#include "generator-m.hpp"
template<typename T>
generator<T> join(generator<T> g1, generator<T> g2)
{
    for( auto&& v : g1 )
        co_yield v;
    for( auto&& v : g2 )
        co_yield v;
}
```

its composable!

```cpp
#include "generator-m.hpp"
template<typename T>
generator<T> join(generator<T> g1, generator<T> g2)
{
    for( auto&& v : g1 )
        co_yield v;
    for( auto&& v : g2 )
        co_yield v;
}
```

its composable!

```
#include "generator-m.hpp"
template<typename T>
generator<T> join(generator<T> g1, generator<T> g2)
{
    for( auto&& v : g1 )
        co_yield v;
    for( auto&& v : g2 )
        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

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```

awaiting

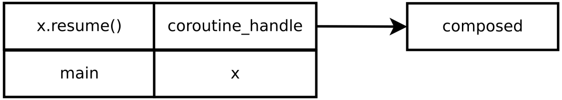
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```

awaiting

```cpp
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```

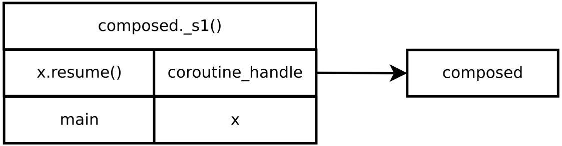
awaiting

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



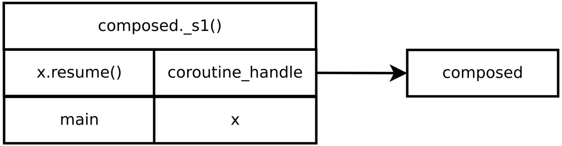
awaiting

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



awaiting

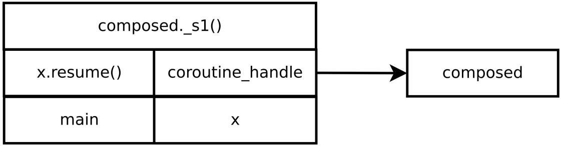
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello

awaiting

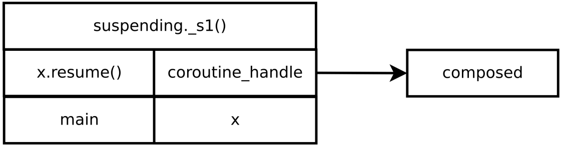
```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello

awaiting

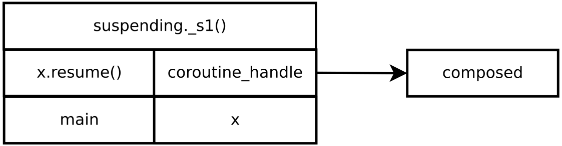
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello

awaiting

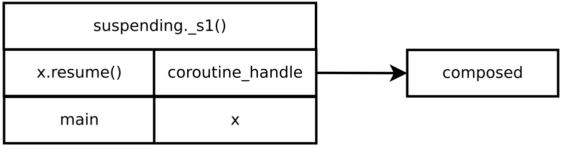
```cpp
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello,

awaiting

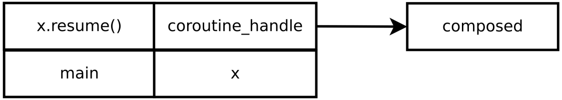
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello,

awaiting

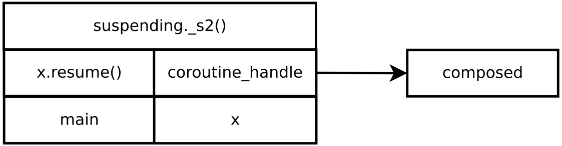
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello,

awaiting

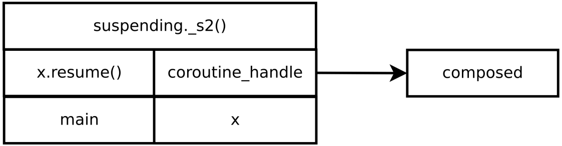
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello,

awaiting

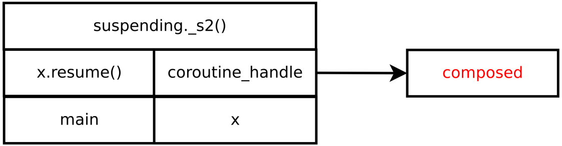
```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



output: hello, !

awaiting

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



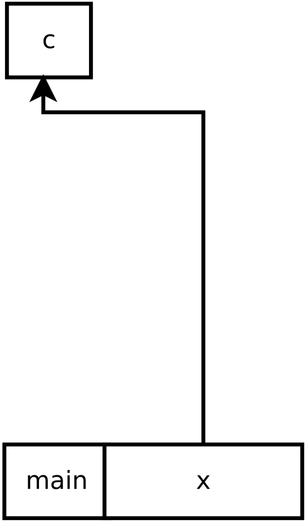
output: hello, !

how do we want to compose tasks

```cpp
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```

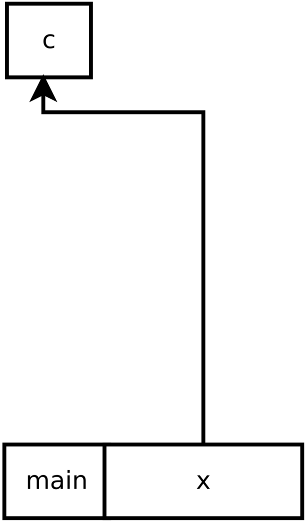
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



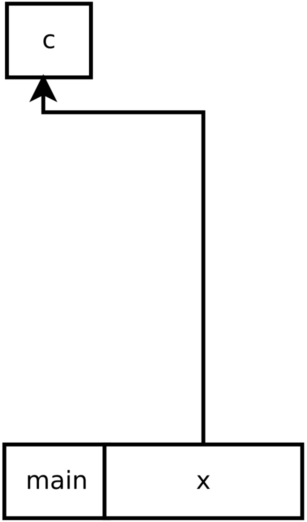
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



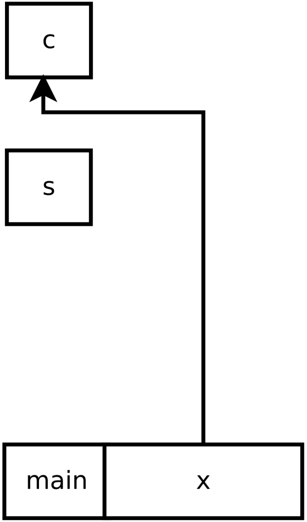
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



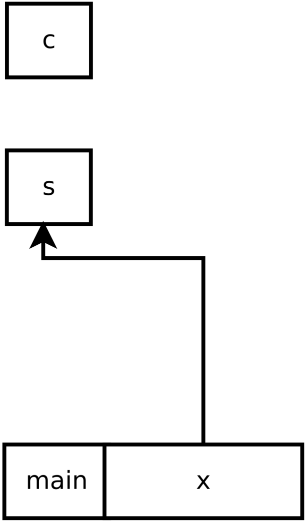
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



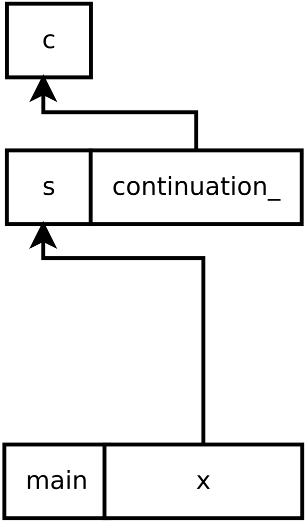
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



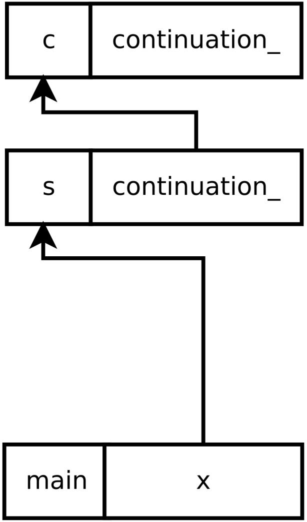
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



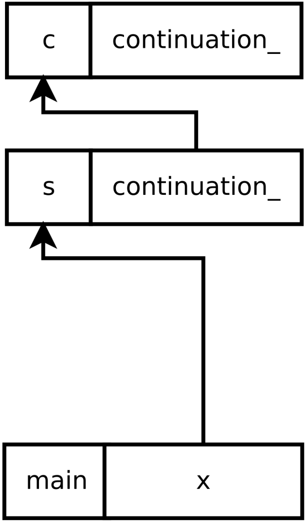
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



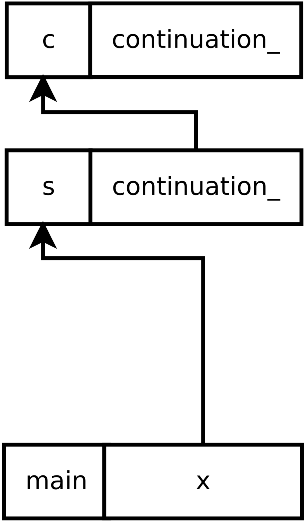
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



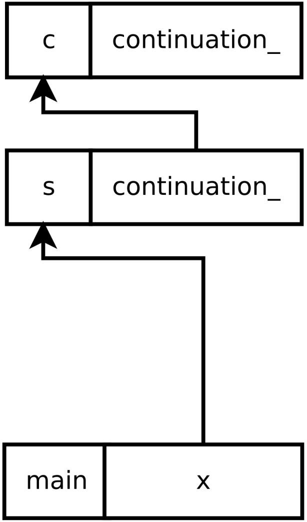
how do we want to compose tasks

```cpp
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



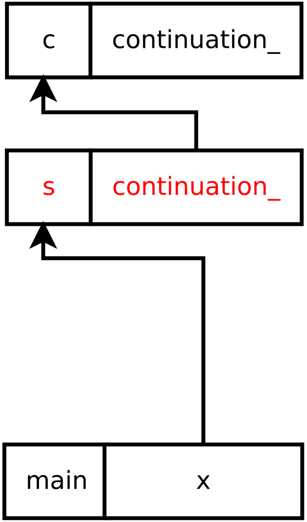
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



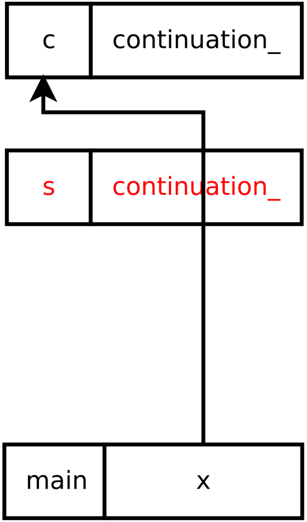
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



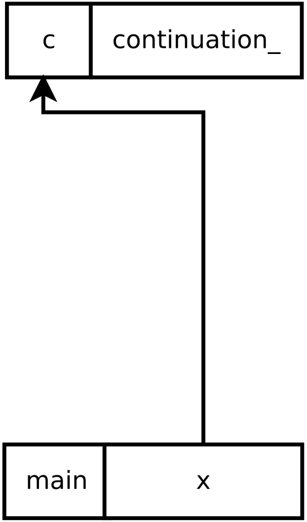
how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ",␣";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



how do we want to compose tasks

```
task<void> suspending() {
    std::cout << ", ";
    co_await suspend_always{};
    std::cout << "word";
};

task<void> composed() {
    std::cout << "hello";
    co_await suspending();
    std::cout << "!";
};

int main() {
    task<void> x = composed();
    while(not x.is_ready())
        x.resume();
}
```



begin of

slides removed from live presentation

trivial awaitable

```
struct suspend_always_
{
    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<>) {}
    constexpr void await_resume() {}
};
```

trivial awaitable

```
struct suspend_always_
{
    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<>) {}
    constexpr void await_resume() {}
};
```

trivial awaitable

```
struct suspend_always_
{
    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<>) {}
    constexpr void await_resume() {}
};
```

trivial awaitable

```
struct suspend_always_
{
    bool await_ready() { return false; }
    void await_suspend(coroutine_handle<>) {}
    constexpr void await_resume() {}
};
```

make task awaitable

```
#include "awaitable-m.hpp"
template<typename T>
class task_{
public:
  // ...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```

make task awaitable

```cpp
#include "awaitable-m.hpp"
template<typename T>
class task_{
public:
  //...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```

make task awaitable

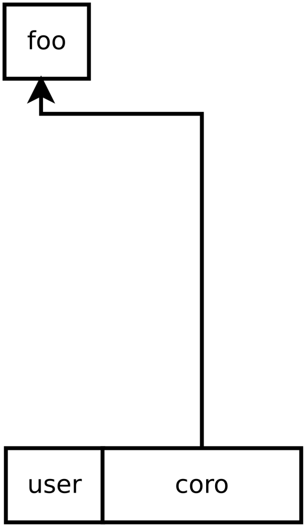
```
#include "awaitable-m.hpp"
template<typename T>
class task_{
public:
  //...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```

make task awaitable

```
#include "awaitable-m.hpp"
template<typename T>
class task_{
public:
  //...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```

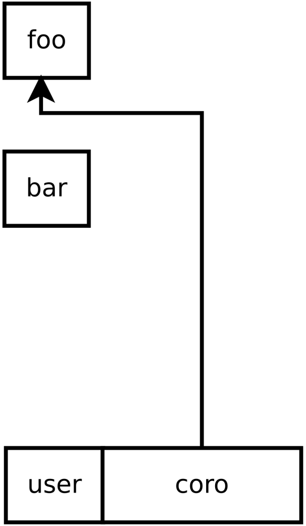
make task awaitable

```
#include "awaitable-m.hpp"
template<typename T>
class task_{
public:
  //...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```



make task awaitable

```
#include "awaitable_m.hpp"
template<typename T>
class task_{
public:
  //...

  auto operator co_await() const & noexcept
  {
      return awaitable<T>{handle_};
  }

private:
  coroutine_handle<promise<T>> handle_;
};
```



our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

our awaitable

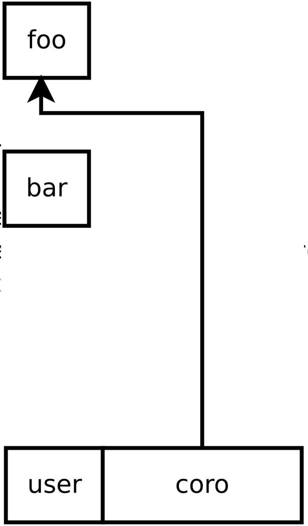
```cpp
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

our awaitable

```cpp
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

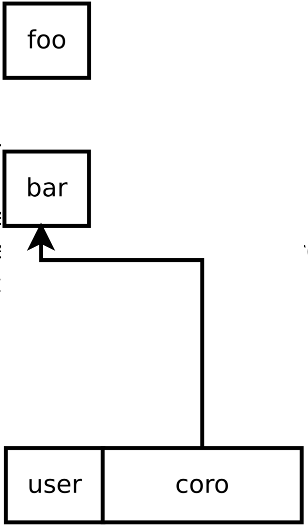
our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                          o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```



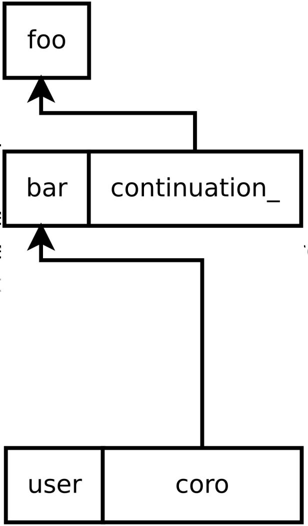
our awaitable

```cpp
#include "awaitable_promise_m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                    o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```



our awaitable

```
#include "awaitable_promise_m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                    o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

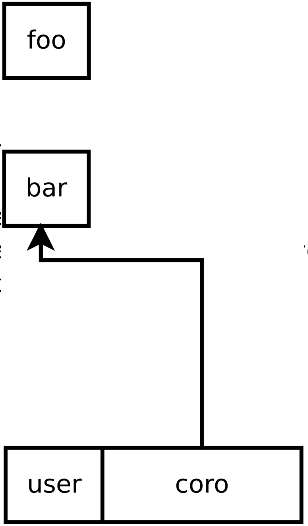


our awaitable

```cpp
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

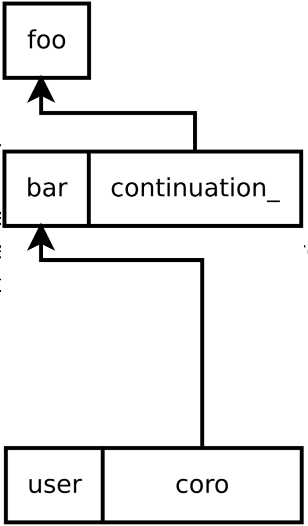
our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                        o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```



our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                    o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

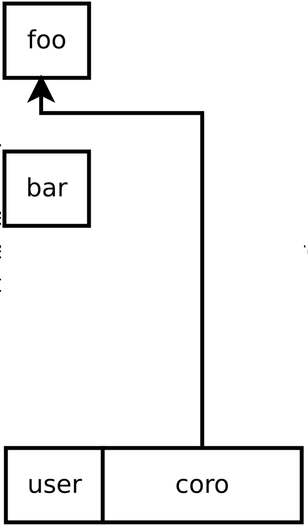


our awaitable

```cpp
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

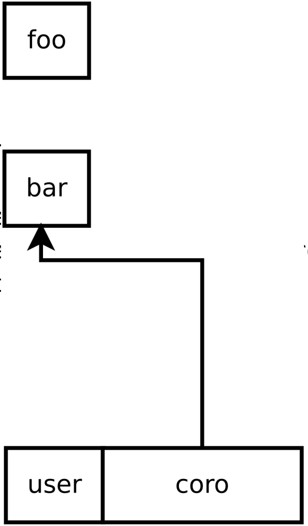
our awaitable

```
#include "awaitable_promise_m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                         o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```



our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> cor

    bool await_ready() { return not coro_.done
    coroutine_handle<> await_suspend(coroutine                    o) {
        coro_.promise().set_continuation(await
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```



our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

our awaitable

```
#include "awaitable_promise-m.hpp"
template <typename T>
struct awaitable
{
    coroutine_handle<promise<T>> coro_;
    awaitable(coroutine_handle<promise<T>> coro) : coro_{coro}{};

    bool await_ready() { return not coro_.done(); }
    coroutine_handle<> await_suspend(coroutine_handle<> awaitingCoro) {
        coro_.promise().set_continuation(awaitingCoro);
        return coro_;
    }
    decltype(auto) await_resume() {
        return coro_.promise().value_;
    }
};
```

setting continuation

```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

setting continuation

```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

setting continuation

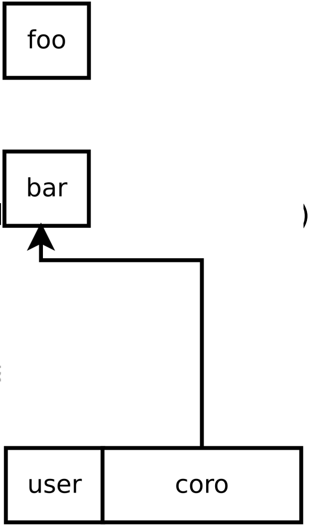
```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

setting continuation

```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

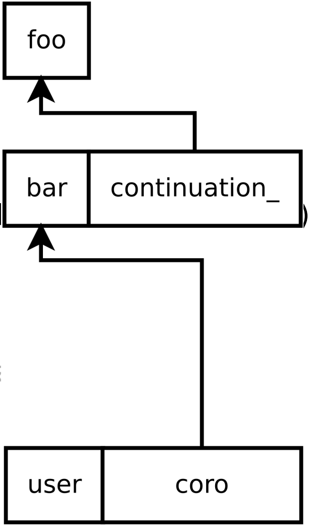
setting continuation

```
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_hand            ) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_a

};
```



setting continuation

```
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_hand                  ) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_a

};

```



setting continuation

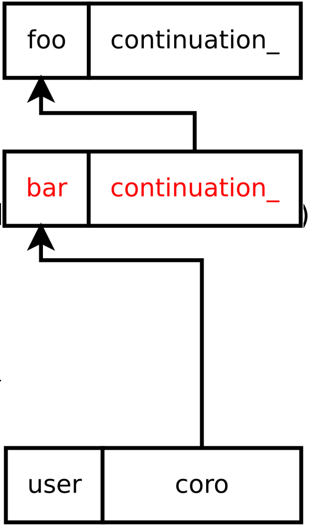
```
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

setting continuation

```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_handle<> continuation) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_awaitable{}; }

};

```

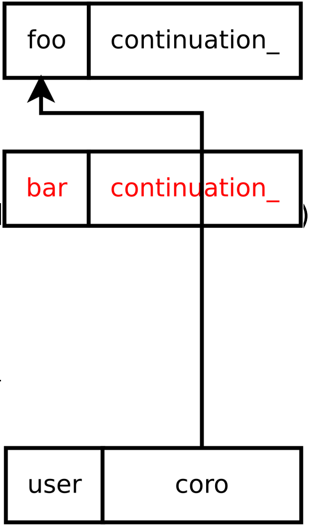
setting continuation

```
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_hand                 ) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_

};

```



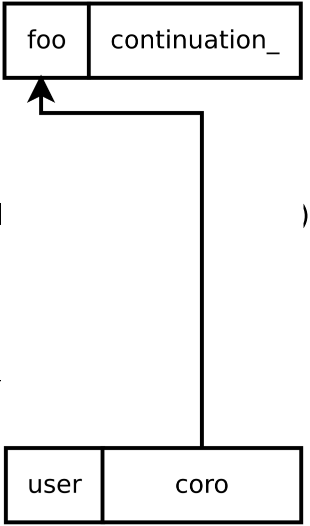
setting continuation

```cpp
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_hand) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_

};
```



setting continuation

```
#include "final_awaitable-m.hpp"
template <typename T>
struct promise {
  // ...

  coroutine_handle<> continuation_;
  void set_continuation(coroutine_hand                 ) {
    continuation_ = continuation;
  }

  auto final_suspend() { return final_

};

```



final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

final suspend - final awaitable

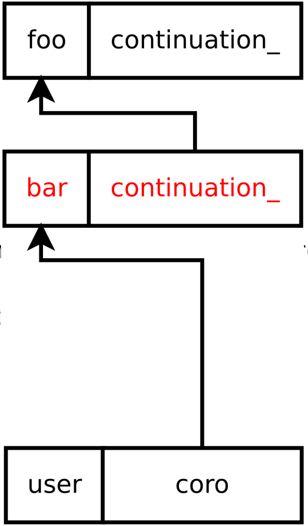
```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

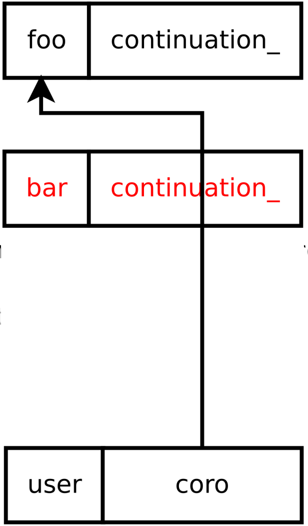
final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                      o)
  {
      return finalizedCoro.promise().continuat
  }
};
```



final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                    o)
  {
      return finalizedCoro.promise().continuat
  }
};
```

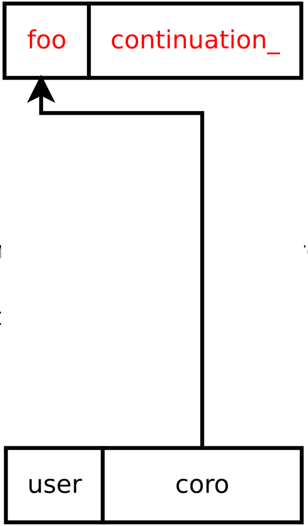


final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

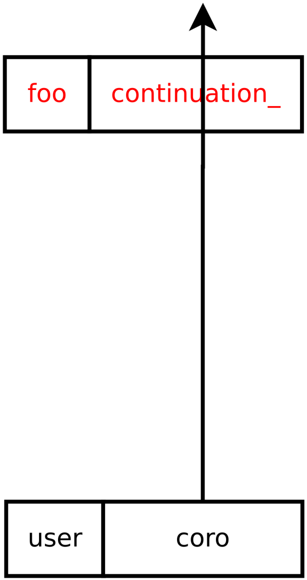
final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                    o)
  {
      return finalizedCoro.promise().continuat
  }
};
```



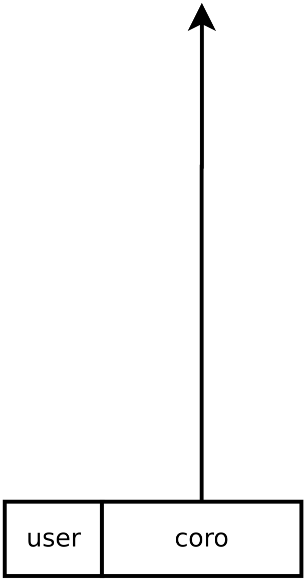
final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                    o)
  {
      return finalizedCoro.promise().continuat
  }
};
```



final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                    o)
  {
      return finalizedCoro.promise().continuat
  }
};
```



final suspend - final awaitable

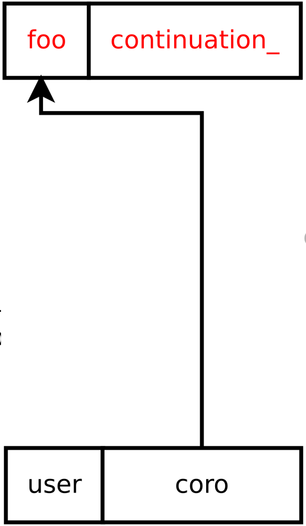
```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      return finalizedCoro.promise().continuation_;
  }
};
```

final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      auto& continuation = finalizedCoro.promise().continuation_;
      return continuation ? continuation : noop_coroutine;
  }
};
```

final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_h                  o)
  {
      auto& continuation = finalizedCoro.promi
      return continuation ? continuation : noc
  }
};
```



final suspend - final awaitable

```
struct final_awaitable
{
  bool await_ready() { return false;};
  void await_resume() noexcept{}

  template <typename P>
  coroutine_handle<> await_suspend(coroutine_handle<P> finalizedCoro)
  {
      auto& continuation = finalizedCoro.promise().continuation_;
      return continuation ? continuation : noop_coroutine;
  }
};
```

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

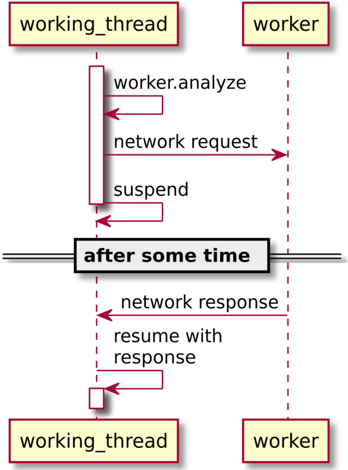
end of

slides removed from live presentation

confesion

confesion

back to motivation



back to motivation

```cpp
using namespace async;
void perform_experiment(experiment const& ex, session& s) {
    std::vector<std::future<sample_result>> worker_futures;
    for(auto const& s: ex.samples)
        worker_futures.push_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_futures)
        results.push_back(f.get());
    s.respond(results).get();
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

back to motivation

```
using namespace coro;
eager_task<void> perform_experiment(experiment ex, session& s) {
    std::vector<eager_task<sample_result>> worker_tasks;
    for(auto& s: ex.samples)
        worker_tasks.emplace_back(worker.analyze(s));

    std::vector<sample_result> results;
    for (auto& f : worker_tasks)
        results.push_back(co_await f);
    co_await s.respond(results);
};
```

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!

?