Work Contracts:

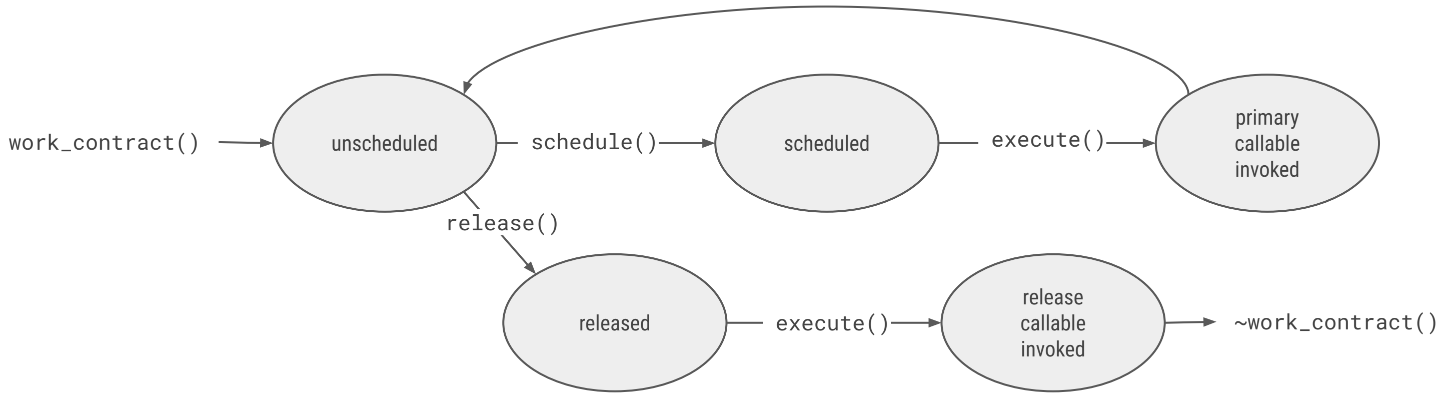
Simple, efficient concurrency and parallelism for low latency C++

Overview:

A work contract consists of a single primary callable (the contract), an optional release callable (dtor) and the contract's state. Both the primary contract callable and the optional release callable are invoked asynchronously by worker threads. A work contract is either unscheduled, scheduled, or released. An unscheduled work contract is an inactive contract and can not be executed.

Scheduling a work contract will transition it to the active state, at which point, the primary callable will be asynchronously invoked by a worker thread. After the callable has been invoked, the work contract is transitioned back to the unscheduled state.

Releasing a work contract will also transition it to the active state. However, when asynchronously executed, the release callable will be invoked rather than the primary callable, after which, the work contract is destroyed rather than transitioned back to the unscheduled state.



Work Contracts are not Tasks:

Work contracts differ from traditional tasks in several key ways:

- **Logic and Data:** Traditional tasks typically combine both the logic (the code to be executed) and the data that is being processed. These tasks are often designed for single execution and are meant to perform specific work using the provided input data.

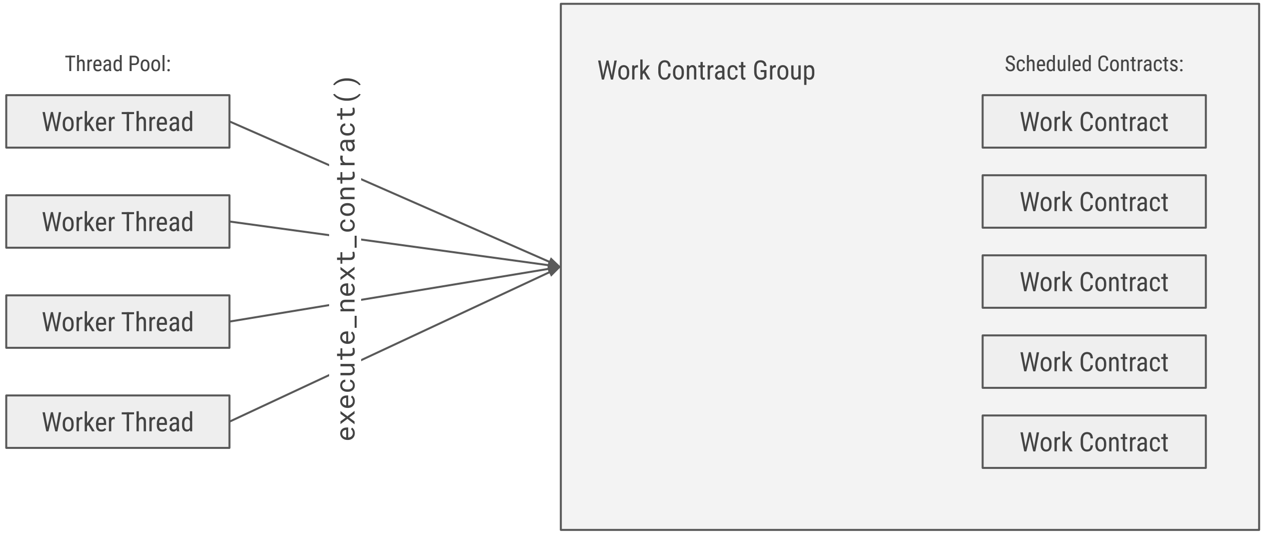
- **Scheduling and Synchronization:** Traditional tasks are usually scheduled by placing them into task queues. This approach often requires additional mechanisms to manage the scheduling, execution, and synchronization of tasks, especially when the task is expected to return computed values. Handling dependencies, synchronization, and returning results demands further design and implementation to ensure proper coordination.

- **Work Contracts and Logical Graphs:** In contrast, work contracts are designed to form logical graphs, where each contract acts as a node within the graph. These nodes represent individual contracts that need to be executed as part of a larger workflow. The contracts may depend on each other, but their execution can proceed independently without the need for explicit synchronization or returning values between them. This approach enables more flexible, modular, and parallel execution models, particularly in complex workflows.

- **Scheduling:** Unlike traditional tasks, work contracts do not require task queues for scheduling. Instead, they are independent and can be self-scheduling. This means that they can be activated or invoked directly, often asynchronously, without relying on a task queue or external mechanism. The primary focus of a work contract is on executing the logic itself, rather than managing complex scheduling or synchronizing results. As a result, work contracts are lighter and more efficient for use cases where simplicity, latency and parallelism are key.

Work Contract Groups:

Work Contracts are not created independently. Instead, they are created via a Work Contract Group container class. Similarly, the execution of scheduled (and released) work contracts is done via that same parent work contract group by invoking work_contract_group::execute_next_contract(). Typically, a thread pool and each worker thread repeated invokes work_contract_group::execute_next_contract.



Work Contracts are Lock Free:

By design, work contracts are thread-safe and are guaranteed to be invoked by at most one thread at a time. As a result, it is often possible to implement any necessary data ingress and egress queues as lock-free, single-producer/single-consumer queues. This design choice helps eliminate the need for locks, improving performance and reducing contention between threads.

Moreover, scheduling a work contract is both lock-free and wait-free, ensuring that no thread is blocked or delayed while scheduling a contract. Additionally, invoking execute_next_contract on the parent work contract group is also lock-free, which ensures that the entire work contract architecture operates without the need for locks. This makes the entire system more efficient and scalable, as it avoids thread contention and guarantees smooth execution even under high concurrency.

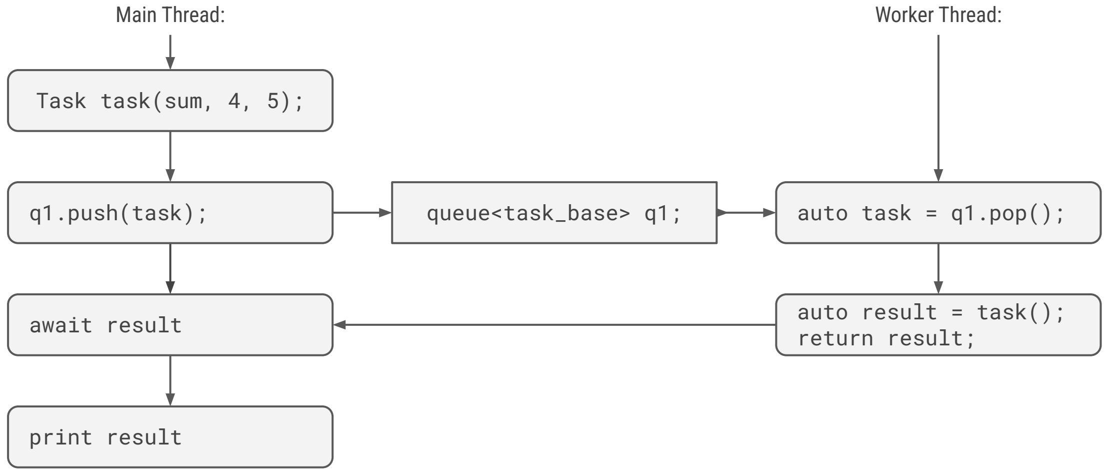
Passing data into work contracts:

Although Work Contracts represent only the logic to be executed, they often require data during invocation. This typically involves using an ingress queue that holds the next set of data for the contract to process. Similarly, if the work contract generates a result needed by subsequent contracts, that output is passed through an egress queue. It is also common to combine the actions of placing data into the queue and scheduling the work contract that will process that data.

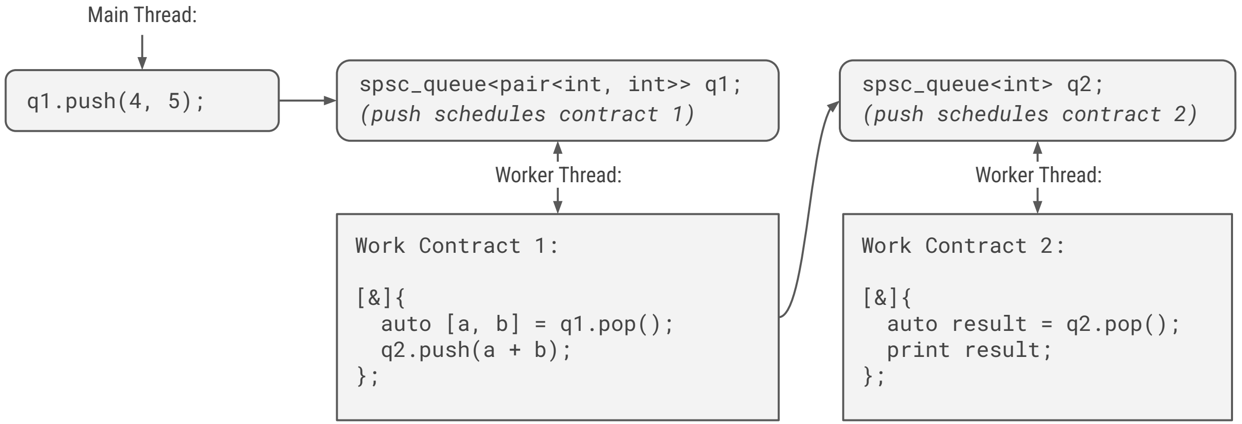
By design, work contracts are thread-safe and are guaranteed to be invoked by at most one thread at a time. As a result, it is often possible to implement any necessary ingress and egress queues as lock-free, single-producer/single-consumer queues. This design choice helps eliminate the need for locks, improving performance and reducing contention between threads.

The following demonstrates a simple asynchronous process for calculating the sum of two integers:

Summing two integers with traditional tasks:



Summing two integers with work contracts:



```
q1.push(4, 5);
```

```
spsc_queue<pair<int, int>> q1;
(push schedules contract 1)
```

```
spsc_queue<int> q2;
(push schedules contract 2)
```

```
Work Contract 1:

[&]{
  auto [a, b] = q1.pop();
  q2.push(a + b);
};
```

```
Work Contract 2:

[&]{
  auto result = q2.pop();
  print result;
};
```

Example 1: The Fundamentals - Hello World

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;});
    workContract.schedule();
    return 0;
}
```

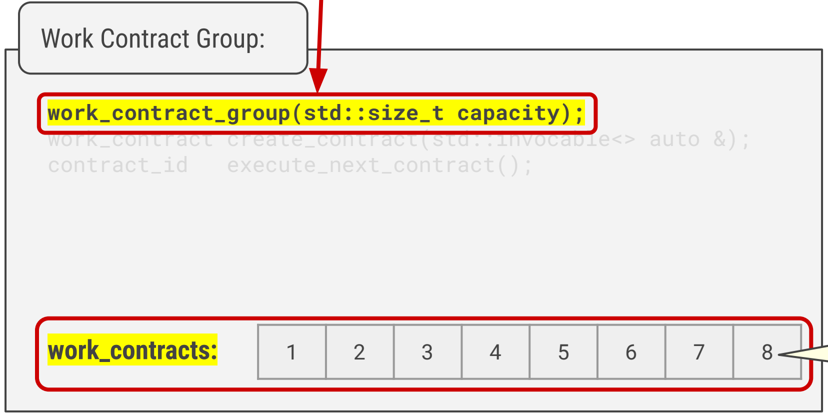
Include the `work_contract` header.

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token s
    auto workContract = wcGroup.create_contra
    workContract.schedule();
    return 0;
}
```

Create a `work_contract_group`. This instance has a maximum capacity of eight `work_contracts`.

A `work_contract_group` is a container for a collection of `work_contracts`. Worker threads invoke any scheduled `work_contracts` that are contained within an instance of a `work_contract_group` by calling the `work_contract_group`'s `execute_next_contract()` function.



Work Contract Group:

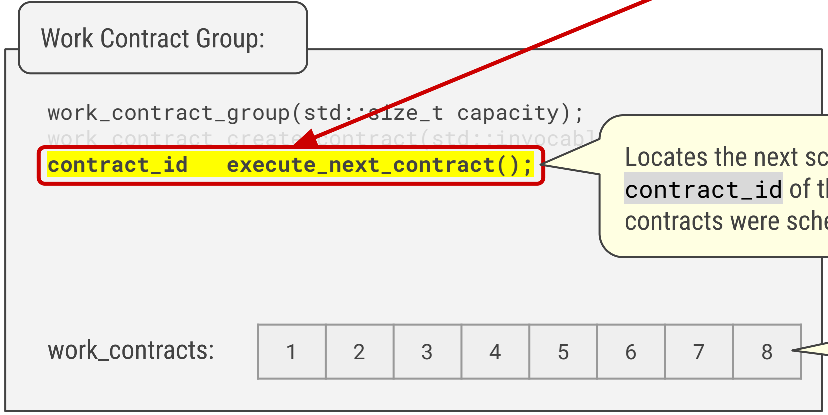
```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocable<> auto &);
contract_id   execute_next_contract();
```

Eight available `work_contracts`.

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;});
    workContract.schedule();
    return 0;
}
```

Create a worker thread which will execute `work_contract`s if they are scheduled. This is accomplished by invoking `execute_next_contract()` function on the desired `work_contract_group`.

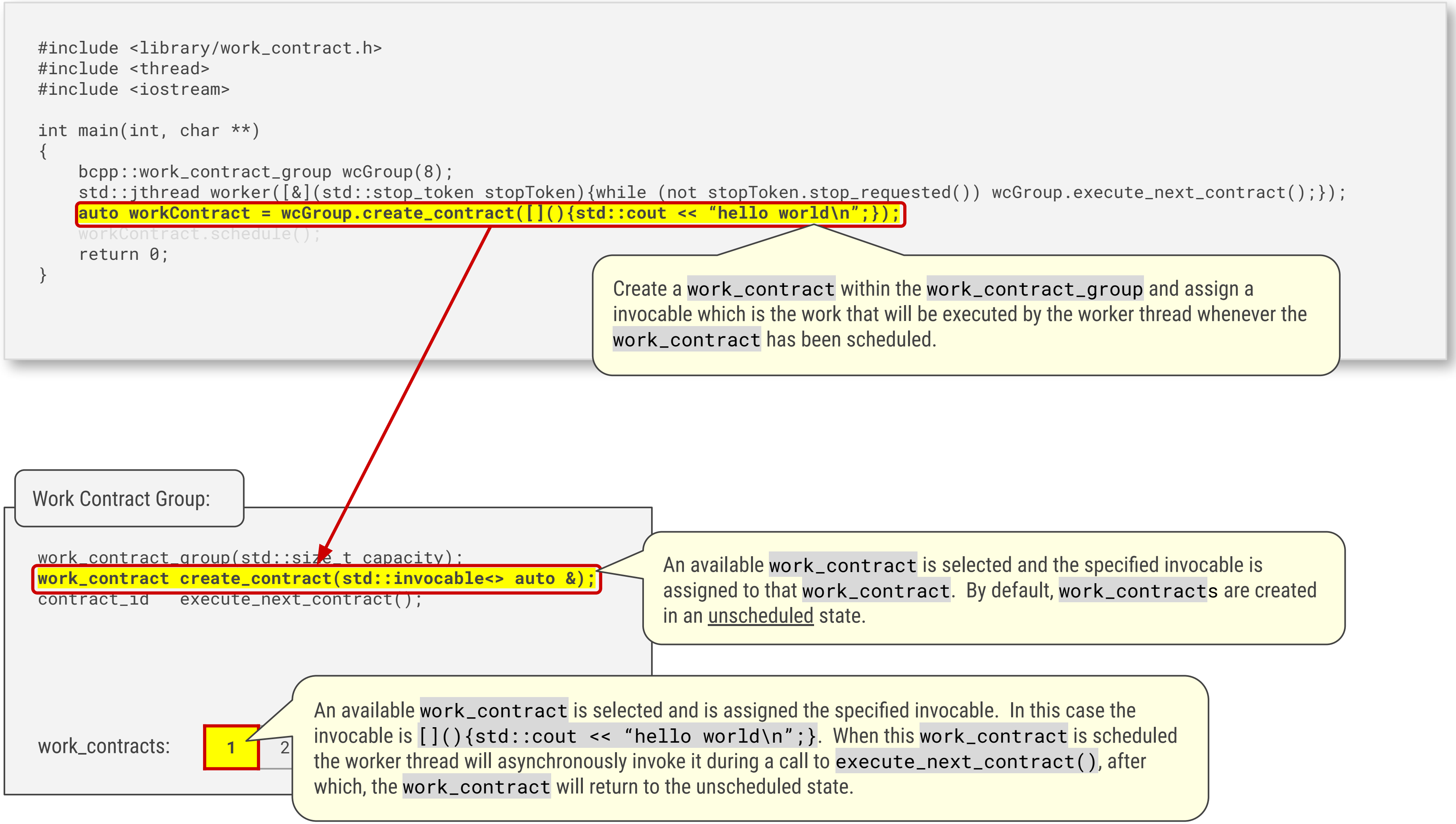


Work Contract Group:

```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocabl
contract_id   execute_next_contract();
```

Locates the next scheduled work contract, executes it, and then returns the `contract_id` of the work contract which has been executed. If no work contracts were scheduled then the invalid contract id (-1) is returned instead.

At this time, there are no scheduled work contracts.



```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;});
    workContract.schedule();
    return 0;
}
```

Create a `work_contract` within the `work_contract_group` and assign a invocable which is the work that will be executed by the worker thread whenever the `work_contract` has been scheduled.

Work Contract Group:

```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocable<> auto &);
contract_id   execute_next_contract();
```

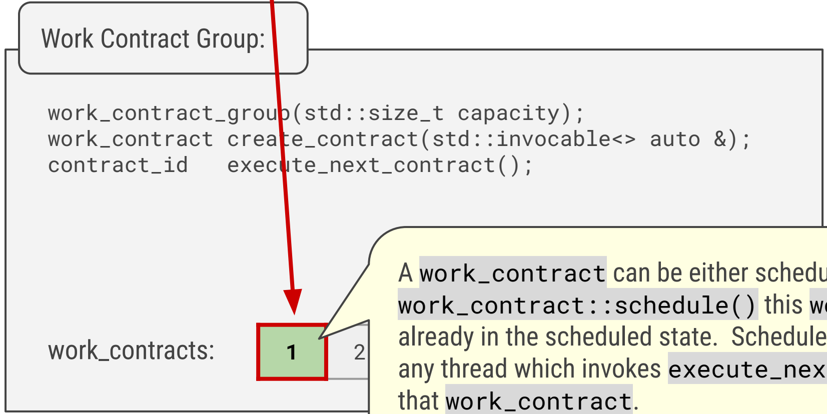
An available `work_contract` is selected and the specified invocable is assigned to that `work_contract`. By default, `work_contract`s are created in an unscheduled state.

An available `work_contract` is selected and is assigned the specified invocable. In this case the invocable is `[](){std::cout << “hello world\n”;}`. When this `work_contract` is scheduled the worker thread will asynchronously invoke it during a call to `execute_next_contract()`, after which, the `work_contract` will return to the unscheduled state.

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;});
    workContract.schedule();
    return 0;
}
```

The `work contract` is then scheduled. Once scheduled, the `work_contract` is eligible for asynchronous execution by the worker thread.



Work Contract Group:

```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocable<> auto &);
contract_id   execute_next_contract();
```

A `work_contract` can be either scheduled or unscheduled. Upon invoking `work_contract::schedule()` this `work_contract` will transition to scheduled, if it was not already in the scheduled state. Scheduled `work_contracts` are eligible for asynchronous execution by any thread which invokes `execute_next_contract()` on the `work_contract_group` which owns that `work_contract`.

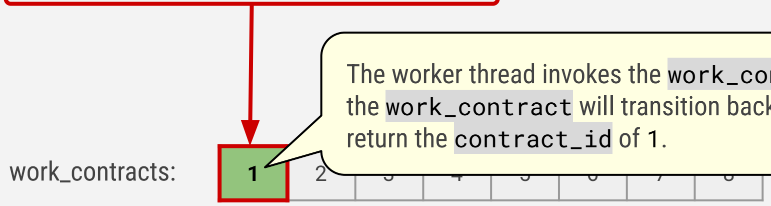
```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << "hello world\n";});
    workContract.schedule();
    return 0;
}
```

Upon invoking `execute_next_contract()` the scheduled `work_contract` is selected and the worker thread invokes the work which was assigned to that `work_contract`. In this case the work is the callable `[](){std::cout << "hello world\n";}` which was assigned at the time when the `work_contract` was created.

Work Contract Group:

```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocable<> auto &);
contract_id    execute_next_contract();
```

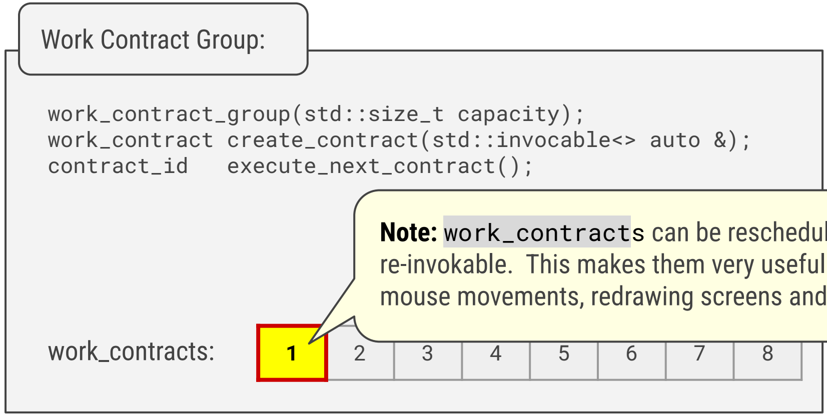


The worker thread invokes the `work_contract` and writes `hello world` to `std::cout`, after which the `work_contract` will transition back to the unscheduled state and `execute_next_contract()` will return the `contract_id` of 1.

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;});
    workContract.schedule();
    return 0;
}
```

Note: Because the `work_contract` is executed asynchronously, the main thread might terminate prior to the worker thread invoking the `work_contract`.



```
work_contract_group(std::size_t capacity);
work_contract create_contract(std::invocable<> auto &);
contract_id   execute_next_contract();
```

Note: `work_contracts` can be rescheduled an arbitrary number of times thereby making them re-invokable. This makes them very useful for repetitive asynchronous tasks such as processing mouse movements, redrawing screens and widgets, receiving packets from a socket, etc.

```
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;}, work_contract::initial_state::scheduled );
    ~~workContract.schedule();~~
    return 0;
}
```

Note: The initial state of a work_contract can be specified at the time of construction. Using this, the contract is created in the scheduled state and the subsequent instruction to explicitly schedule the work_contract can be removed.

```cpp
#include <library/work_contract.h>
#include <thread>
#include <iostream>

int main(int, char **)
{
    bcpp::work_contract_group wcGroup(8);
    std::jthread worker([&](std::stop_token stopToken){while (not stopToken.stop_requested()) wcGroup.execute_next_contract();});
    auto workContract = wcGroup.create_contract([](){std::cout << “hello world\n”;}, work_contract::initial_state::scheduled);
    return 0;
}
```

Example #1 Conclusion:

`work_contract`s eliminate the need for boilerplate code as well as a lot of verbosity. The resulting code is very succinct and easy to follow. The above example implements a completely asynchronous “hello world” example using only three lines of code.

However, `work_contract`s facilitate far more than simple one-shot asynchronous function calls. See the following for more information:

- **[WIP]**
- **[WIP]**