MULTITHREADING

ASYNCHRONOUS TASKS

CODERS SCHOOL

BARTOSZ SZURGOT

ŁUKASZ ZIOBROŃ

AGENDA

1. problems with `std::thread`
2. `std::promise` and `std::future`
3. `std::shared_future`
4. `std::async`
5. Launch policies
6. `std::packaged_task`

SOMETHING ABOUT YOU

- One of the most interesting things from Cpp Core Guidelines pre-work?
- Do you prefer threads or async?

ŁUKASZ ZIOBROŃ

EXPERIENCE NOT ONLY IN PROGRAMMING

- Trainer and DevOps @ Coders School
- C ++ and Python developer @ Nokia and Credit Suisse
- Team leader and trainer @ Nokia
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Web developer (HTML, PHP, CSS) @ StarCraft Area

EXPERIENCE AS A TRAINER

- C ++ courses @ Coders School
- Practical Aspects of Software Engineering @ PWr & UWr
- Nokia Academy
- Internal corporate training

EXPERIENCE IN PUBLIC SPEAKING

- code::dive conference
- code::dive community
- Polish Academic Championship in Team Programming

INTERESTS

- StarCraft Brood War & StarCraft II
- Motorcycles
- Photography
- Archery
- Andragogy
- Marketing 🆕

CONTRACT

- 🎰 The Vegas rule
- 🗣️ Discussion, not lecture
- ☕ Additional breaks on demand
- ⌚ Punctuality

PRE-TEST

```
#include <future>
#include <iostream>

int main() {
    int x = 0;
    auto f = std::async(std::launch::deferred, [&x]{
        x = 1;
    });

    x = 2;
    f.get();
    x = 3;
    std::cout << x;
    return 0;
}
```

1. the type of f is `promise<int>`
2. the type of f is `future<void>`
3. `async()` without a launch policy may never be called
4. this program always prints 3
5. `x = 2` assignment cause a data race
6. if async was run with `std::launch::async`, there would be a data race
7. `x = 3` assignment is safe, because it happens after synchronization with async task
8. `future<void>` may be used to synchronize tasks

PROBLEMS WITH THREADS

`std::thread` IS A LOW-LEVEL MECHANISM

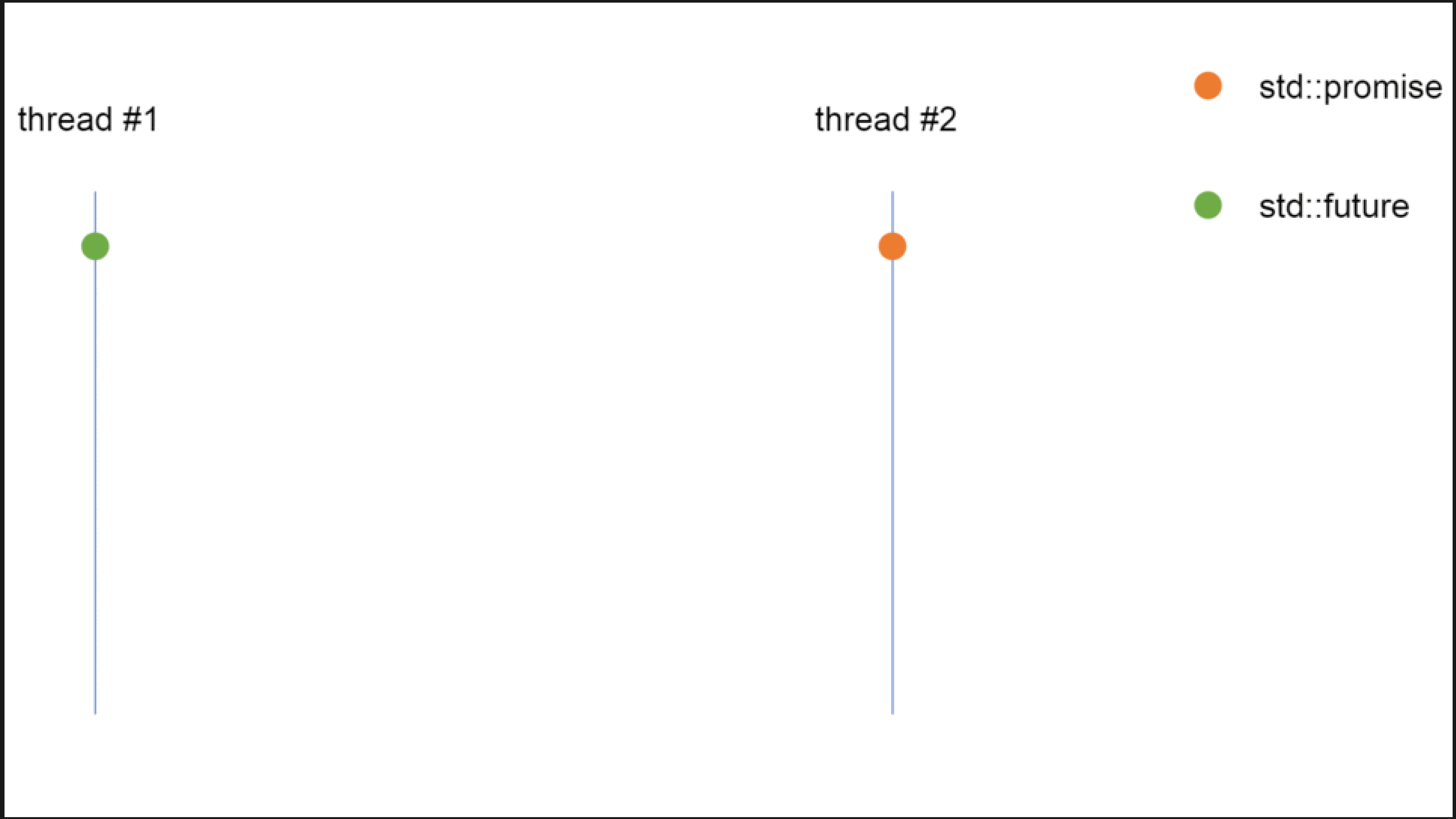
- How to return something from it?
  - Pass variable by reference (use `std::ref`)
- How to forward an exception?
  - Use `std::exception_ptr` and `std::current_exception`
- Should be manually joined/detached?
  - Lack of `join()` or `detach()` terminates program execution

ONE WAY COMMUNICATION

`std::promise`

`std::future`

HOW `std::promise/std::future` WORKS



BASIC `std::promise/std::future` USAGE

```
std::promise<int> promise;
std::future<int> future = promise.get_future();      // connected pair
auto function = [] (std::promise<int> promise) {
    // ...
    promise.set_value(10);
};
std::thread t(function, std::move(promise));
std::cout << future.get() << std::endl;
t.join();
```

- `std::promise/std::future` pair is used to create one-way communication channel
- `std::promise` is used for setting value
- `std::future` is used for getting value
- `std::promise/std::future` pair can be used only once

QUIZ

FIND BROKEN PIECE OF CODE

```
#1 auto future = promise.get_future();
#2 promise.set_value(10);
#1 future.get();
```

```
#2 promise.set_value(10);
#1 auto future = promise.get_future();
#1 future.get();
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 promise.set_value(10);
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 // set wasn't called
```

FIND BROKEN PIECE OF CODE

```
#1 auto future = promise.get_future();
#2 promise.set_value(10);
#1 future.get();
```

```
#2 promise.set_value(10);
#1 auto future = promise.get_future();
#1 future.get();
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 promise.set_value(10);
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 // set wasn't called
```

FIND BROKEN PIECE OF CODE

```
#1 auto future = promise.get_future();
#2 promise.set_value(10);
#1 future.get();
```

```
#2 promise.set_value(10);
#1 auto future = promise.get_future();
#1 future.get();
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 promise.set_value(10);
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 // set wasn't called
```

FIND BROKEN PIECE OF CODE

```
#1 auto future = promise.get_future();
#2 promise.set_value(10);
#1 future.get();
```

```
#2 promise.set_value(10);
#1 auto future = promise.get_future();
#1 future.get();
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 promise.set_value(10);
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 // set wasn't called
```

FIND BROKEN PIECE OF CODE

```
#1 auto future = promise.get_future();
#2 promise.set_value(10);
#1 future.get();
```

```
#2 promise.set_value(10);
#1 auto future = promise.get_future();
#1 future.get();
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 promise.set_value(10);
```

```
#1 auto future = promise.get_future();
#1 future.get();
#2 // set wasn't called
```

MORE ABOUT `std::promise` AND `std::future`

MORE ABOUT `std::promise`

```
std::thread t(function, std::move(promise));
```

- `std::promise` and `std::future` can be moved only

```
std::future<int> future = promise.get_future();
```

- returns a future associated with the promised result
- second call will throw

HOW TO "SET"?

- set value

```
promise.set_value(10);
```

- set exception

```
promise.set_exception(std::make_exception_ptr(e));
```

```
try {
    // ...
} catch(...) {
    promise.set_exception(std::current_exception());
}
```

`std::promise` can be "set" only once.

`std::future_errc::promise_already_satisfied` exception is thrown when it is set multiple times.

MORE ABOUT `std::future`

- `future.valid()`
  - Checks if the future can be used
  - Using invalid future cause Undefined Behavior
- `future.wait()`
  - waits for the result to become available

HOW TO "GET"?

`future.get()`

- waits for the results to become available and returns the result
- will automatically throw stored exception
- will invalidate the future

`std::future` can be "get" only once.

`std::future_errc::future_already_retrieved` exception is thrown when it is get multiple times.

EXERCISES

EXERCISE

`exercises/01_get_number_async.cpp`

Implement `get_number_async()` function. It should call `get_number()` asynchronously on another thread and return `std::future` which will hold the result.

```cpp
int get_number() {
    return 10;
}

int main() {
    auto future = get_number_async();
    return future.get();
}
```

EXAMPLE ON SLIDE 3.3

SOLUTION

```
std::future<int> get_number_async() {
    std::promise<int> p;
    std::future<int> f = p.get_future();
    auto wrapped_func = [] (std::promise<int> p) {
        p.set_value(get_number());
    };
    std::thread t(wrapped_func, std::move(p));
    t.detach();
    return f;
}
```

EXERCISE

`exercises/02_schedule.cpp`

Implement a `schedule()` function. It should be able to take a function like `get_number()` as a parameter and call it asynchronously on another thread. It should return `std::future` which will hold the result.

```cpp
int get_number() {
  return 10;
}

int main() {
  auto future = schedule(get_number);
  return future.get();
}
```

Does it work on the below function? 🧐

```cpp
int throw_sth() {
    throw std::runtime_error{"Sorry"};
}
```

SOLUTION

```
std::future<int> schedule(std::function<int()> func) {
    std::promise<int> p;
    std::future<int> f = p.get_future();
    auto wrapped_func = [func] (std::promise<int> p) {
        try {
            p.set_value(func());
        } catch(...) {
            p.set_exception(std::current_exception());
        }
    };
    std::thread t(wrapped_func, std::move(p));
    t.detach();
    return f;
}
```

BETTER SOLUTION

- Use template for function
- Use variadic templates for function arguments
- Use `std::invoke_result_t` to get return type
- Use perfect forwarding
- Do not pass arguments by ref into lambda - lifetime issues

It has more to do with templates rather than multithreading, so this is your homework.

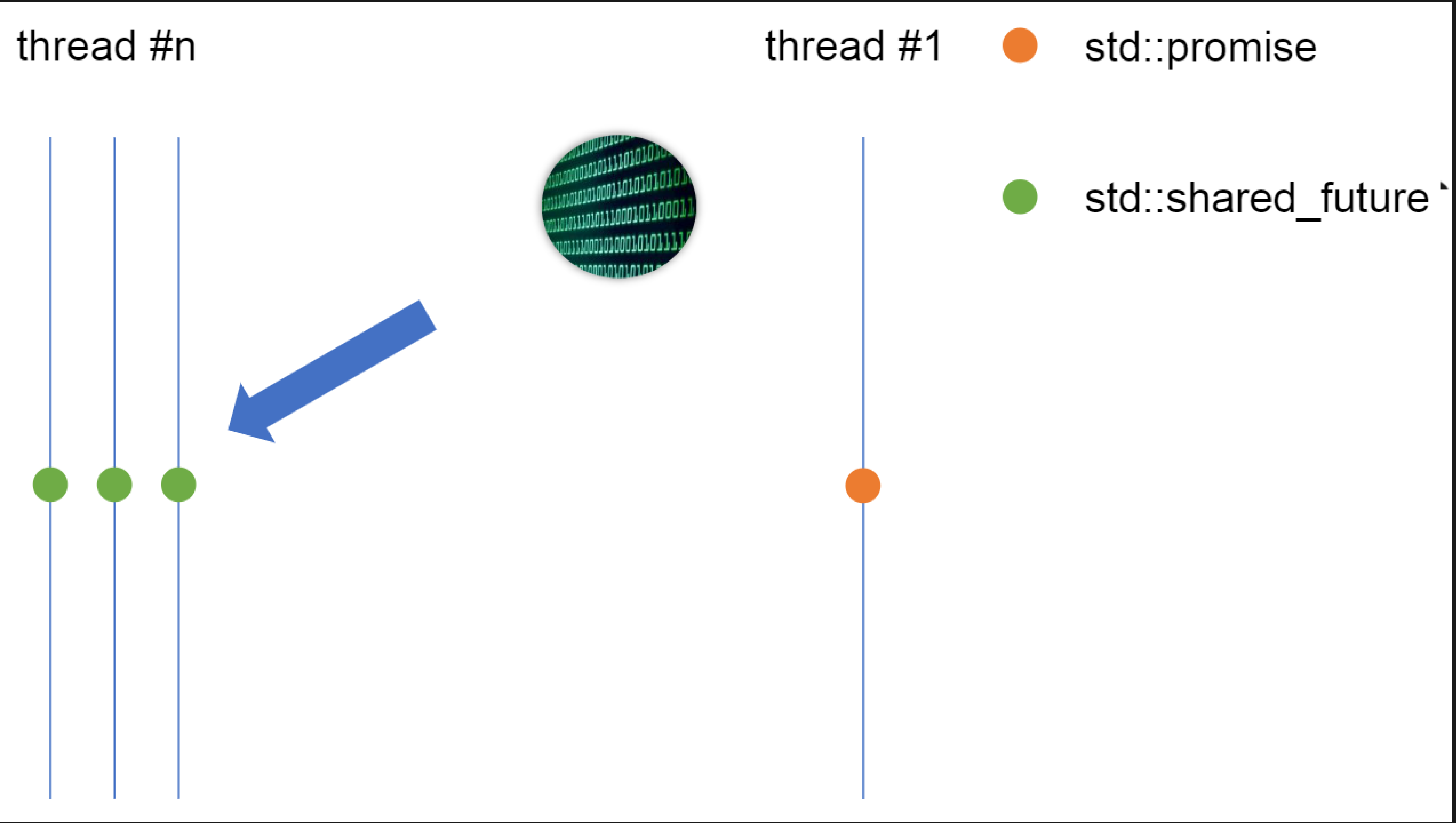
HOMEWORK

Modify the `schedule()` function, so it can take a function of any type, and behave similarly to `std::async()`

Take an examples from: cppreference

ONE-TO-MANY CONNECTION

ONE-TO-MANY CONNECTION



std::shared_future<T>

```
std::shared_future<int> sfuture = promise.get_future().share();
```

- allows multiple getting
- copyable and movable
- each thread should have its own shared_future object

`std::shared_promise` does not exist

ASYNCHRONOUS TASKS

CODERS SCHOOL

PROMISE/FUTURE VS ASYNC APPROACH

```cpp
void promise_future_approach() {
  std::promise<int> prom;
  std::future<int> fut = prom.get_future();
  auto function = [] (std::promise<int> prom) {
    // ...
    prom.set_value(10);
  };
  std::thread t(function, std::move(prom));
  t.detach();
  std::cout << fut.get() << std::endl;
}
```

```cpp
void async_approach () {
  auto function = [] () {
    // ...
    return 20;
  };
  std::future<int> fut = std::async(function);
  std::cout << fut.get() << std::endl;
}
```

`std::async`

- `#include <future>`
- `std::async()`
- Wraps a function that can be called asynchronously
- Returns appropriate `std::future`
- Handles exceptions through `std::promise/std::future`
- Automatically throws exceptions as needed

```
$> ./02_async
10
20
```

`std::async`

- `std::async` is a high-level solution (finally!) that automatically manages asynchronous calls with basic synchronization mechanisms
- The most convenient form of launching tasks:
  - handling return values
  - exception handling
  - synchronization (blocking `get()` and `wait()` on `std::future`)
  - scheduler - automatic queuing of tasks performed by implementation of the standard library
  - ability to manually select the type of launch (immediate, asynchronous `async`, synchronous `deferred`)

EXERCISE

exercises/03_exceptions.cpp

Simplify the code by using async instead of threads.

```cpp
#include <thread>
#include <iostream>
#include <string>
#include <random>
using namespace std;

std::random_device rd;

int main() {
    std::exception_ptr thread_exception = nullptr;
    std::string result;

    auto task = [](std::exception_ptr & te, std::string & result) {
        try {
            std::mt19937 gen(rd());
            std::bernoulli_distribution d(0.5);
            if (d(gen)) {
                throw std::runtime_error("WTF");
            } else {
                result = "success";
            }
        } catch (...) {
            te = std::current_exception();
        }
    };
```

```
    std::thread t(task, std::ref(thread_exception), std::ref(result));

    std::cout << "Some heave task on main thread\n";
    std::this_thread::sleep_for(1s);
```

SOLUTION

```
#include <iostream>
#include <string>
#include <random>
#include <future>
using namespace std;

std::random_device rd;

int main() {
    auto task = []() {
        std::mt19937 gen(rd());
        std::bernoulli_distribution d(0.5);
        if (d(gen)) {
            throw std::runtime_error{"WTF"};
        } else {
            return "success";
        }
    };

    auto result = std::async(std::launch::async, task);

    std::cout << "Some heave task on main thread\n";
    std::this_thread::sleep_for(1s);

    try {
        auto value = result.get();
        std::cout << "Task exited normally with result: " << value << '\n';
    } catch (const std::exception & ex) {
        std::cout << "Task exited with an exception: " << ex.what() << "\n";
    }
```

DRAWBACKS OF `async`

It may fail due to resource exhaustion, rather than queuing up tasks to be executed later. It does not use a thread pool to reschedule a failed task.

LAUNCH POLICIES

CODERS SCHOOL

LAUNCH POLICIES - EXAMPLE

```cpp
#include <chrono>
#include <future>
#include <iostream>
#include <vector>
using namespace std;

int main() {
    auto f1 = async([]{
        cout << "f1 started\n";
        this_thread::sleep_for(1s);
        return 42;
    });
    cout << "f1 spawned\n";

    auto f2 = async(launch::async, []{
        cout << "f2 started\n";
        this_thread::sleep_for(1s);
        return 2 * 42;
    });
    cout << "f2 spawned\n";

    auto f3 = async(launch::deferred, []{
        cout << "f3 started\n";
        this_thread::sleep_for(1s);
        return 3 * 42;
    });
    cout << "f3 spawned\n";
```

- Launch examples/04_async_policies
- Look at the source code
- Launch examples/05_async_ids
- Experiment with launch policies settings
- Observe how do programs work
- Draw conclusions :)

```
$> ./04_async_policies
f1 spawned
f1 started
f2 spawned
f3 spawned
Getting f1 result
f2 started
Got f1 result
Getting f2 result
Got f2 result
Getting f3 result
f3 started
Got f3 result
42
84
126
```

LAUNCH POLICIES

```
async(std::launch policy, Function&& f, Args&&... args);
```

- `launch::async`
  - Asynchronous call on a separate system thread
- `launch::deferred`
  - Lazily executes the function `f` the first time methods `get()` or `wait()` are called on the `future`.
  - Execution is synchronous - the caller waits for the function `f` to complete.
  - If `get()` or `wait()` is not called, the function `f` will not be executed.
- Both `launch::async | launch::deferred` (default)
  - Asynchronous execution or lazy evaluation (up to the implementation)
  - It is not known whether the `f` will be executed concurrently
  - It is not known whether the `f` will be executed on another thread or on the same thread that calls `get()` or `wait()` on `future`
  - It is impossible to predict whether the `f` will be executed at all, because there may be paths in the code where `get()` or `wait()` will not be called (eg. due to exceptions)
- Neither `launch::async` or `launch::deferred`
  - Undefined Behavior
- There are also additional, implementation defined policies allowed

EXERCISE: NO LAUNCH POLICY PROBLEM

exercises/04_async_never_called.cpp

```cpp
#include <iostream>
#include <future>
using namespace std;

void f() {
    this_thread::sleep_for(1s);
}

int main() {
    auto fut = async(f);

    while (fut.wait_for(100ms) != future_status::ready) {
        // loop until f has finished running...
        // ... which may never happen!
        cout << "Waiting...\n";
    }
    cout << "Finally...\n";
}
```

- Undefined Behavior?
- If the scheduler choose `std::launch::async` then everything is fine
- If it choose `std::launch::deferred` then `future_status` will never get `ready` value which gives us infinite loop
- The selected policy may depend on the current system load
- Can you fix this code without specifying a policy?

```
$> ./04_async_never_called
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Waiting...
Finally...
```

SOLUTION

```cpp
#include <iostream>
#include <future>
using namespace std;

void f() {
    this_thread::sleep_for(1s);
}

int main() {
    auto fut = async(f);

    if (fut.wait_for(0s) == future_status::deferred) {
        cout << "Scheduled as deferred. "
             << "Calling wait() to enforce execution\n";
        fut.wait();
    } else {
        while (fut.wait_for(100ms) != future_status::ready) {
            cout << "Waiting...\n";
        }
    }
    cout << "Finally...\n"
}
```

- There is no direct way to check how `future` will be/was run, but...
- `wait_for()` returns 1 of 3 statuses:
  - `future_status::deferred`
  - `future_status::ready`
  - `future_status::timeout`
- `wait_for()` called with 0 time returns immediately
  - `future_status::deferred` means that `deferred` was chosen
  - `future_status::timeout` means that `async` was chosen
- cppreference.com

THE RIDDLE

```
#include <iostream>
#include <string>
#include <future>

int main() {
    std::string x = "x";
    std::async(std::launch::async, [&x](){
        x = "y";
    });
    std::async(std::launch::async, [&x](){
        x = "z";
    });
    std::cout << x;
}
```

- What will be displayed on screen?
  - x
  - y
  - z
  - It depends (on what?)

ANSWER

```
$> ./06_riddle
z
```

- Explanation:
  - if the future object is temporary, it waits in the destructor until the task is over. So the second task will be run after the first one and even though they will be in other threads, their execution will be synchronized
  - cppreference.com ~future
  - Source
  - std::futures from std::async aren't special! - Scott Meyers
- Conclusions:
  - If you want to have asynchronous calls, you have to save the result in the `std::future` variable
- C++20 - changes:
  - `std::async` has marked it's return type with `[[nodiscard]]` attribute. Compiler will emit a warning when it is not assigned to a local variable.

std::packaged_task

CODERS SCHOOL

`std::packaged_task` - TRAITS

- `#include <future>`
- `std::packaged_task` is something between `std::async()` and `std::thread`.
- It's a callable class (functor), not a function, like `std::async()`
- Auxiliary object through which the `std::async()` can be implemented
- Wraps a function that can be called asynchronously
- `operator()` returns the appropriate `std::future<T>`
- Handles exceptions via `std::promise/std::future`
- Does not start automatically
- Requires explicit calling
- The call can be forwarded to another thread

```
$> ./07_packaged_task
```

std::packaged_task - EXAMPLE

```cpp
#include <iostream>
#include <cmath>
#include <future>

auto globalLambda = [](int a, int b) {
    return std::pow(a, b);
};

void remoteAsync() {
    auto result = std::async(std::launch::async, globalLambda, 2, 9);
    std::cout << "getting result:\t" << result.get() << '\n';
}

void localPackagedTask() {
    std::packaged_task<int(int, int)> task(globalLambda);
    auto result = task.get_future();
    task(2, 9);
    std::cout << "getting result:\t" << result.get() << '\n';
}

void remotePackagedTask() {
    std::packaged_task<int(int, int)> task(globalLambda);
    auto result = task.get_future();
    std::thread t(std::move(task), 2, 9);
    t.detach();
    std::cout << "getting result:\t" << result.get() << '\n';
}
```

RECAP

CODERS SCHOOL

POINTS TO REMEMBER

- Your code is high-level if you use only `std::async` and `std::future` object. Raw `std::thread`, `std::promise` or `std::packaged_task` objects means that it uses lower abstraction level, which is more complicated to understand.
- Calling `std::async` without a policy may cause unexpected behavior, like task not being called at all.
- `std::promise` can be set only once
- `std::future` can be get only once
- There is a `std::shared_future`, but there is no ~~`std::shared_promise`~~
- `std::async` does NOT always spawn a new thread. It may use a thread pool or run the task synchronously on the thread that called `get()`
- Creating `std::future` via the default constructor does not make any sense. It will be always invalid.

PRE-TEST ANSWERS

```cpp
#include <future>
#include <iostream>

int main() {
    int x = 0;
    auto f = std::async(std::launch::deferred, [&x]{
        x = 1;
    });

    x = 2;
    f.get();
    x = 3;
    std::cout << x;
    return 0;
}
```

1. the type of f is `promise<int>`
2. the type of f is `future<void>`
3. `async()` without a launch policy may never be called
4. this program always prints 3
5. `x = 2` assignment cause a data race
6. if async was run with `std::launch::async`, there would be a data race
7. `x = 3` assignment is safe, because it happens after synchronization with async task
8. `future<void>` may be used to synchronize tasks

POST-WORK

- Implement your own `async()` - upgrade `schedule()` function from this session
- Post-test
- Training evaluation

IMPLEMENT YOUR OWN `async`

Modify the `schedule()` function, so it can take a function of any type, and behave similarly to `std::async()`

- Use template parameter for function
- Use variadic templates for function arguments
- Use `std::invoke_result_t` to get a return type
- Use perfect forwarding
- Do not pass arguments by ref into lambda - lifetime issues
- Customize the behavior by using launch policies

Take examples from: cppreference

1. DESCRIBE ONE THING FROM TODAY'S SESSION
2. WHAT WAS THE MOST AMAZING FOR YOU?

USEFUL LINKS

- std::async on cppreference.com
- std::packaged_task on cppreference.com
- std::futures from std::async aren't special! - Scott Meyers
- The difference between std::async and std::packaged_task

CODERS SCHOOL