MULTITHREADING

THREADS

ŁUKASZ ZIOBROŃ

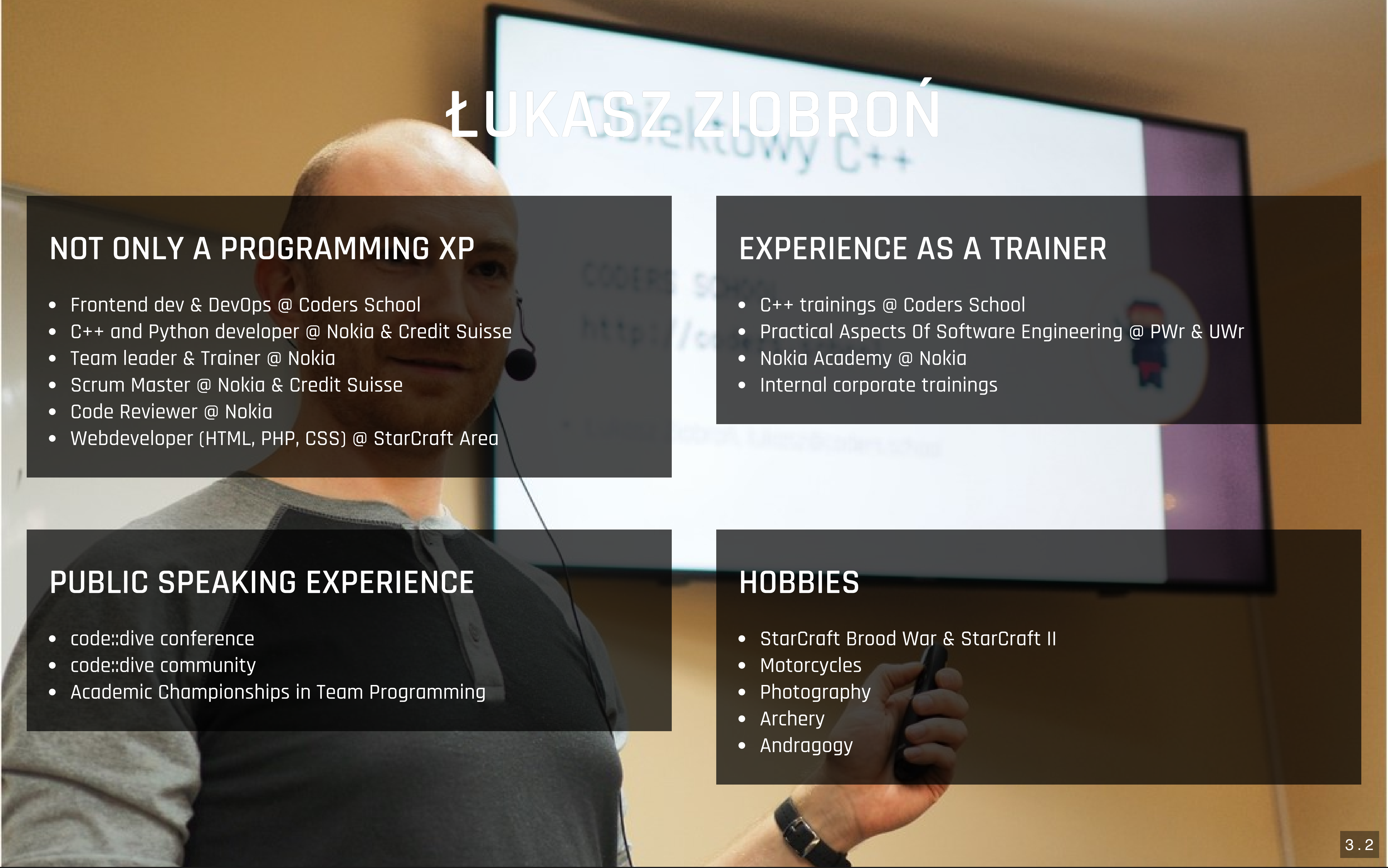
MATEUSZ ADAMSKI

THREADS - AGENDA

- intro
- concurrency
- `std::thread`
- passing arguments to threads
- `join()` or `detach()`?
- RAII
- copy elision (RVO)
- dangling reference
- exceptions in threads
- auxiliary functions
- threads in collections
- homework

SOMETHING ABOUT YOU

- On how many of my trainings have you already been?
- How many cores does your machine have?
- Which concurrent mechanisms have you already used?



ŁUKASZ ZIOBROŃ

NOT ONLY A PROGRAMMING XP

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

EXPERIENCE AS A TRAINER

- C++ trainings @ Coders School
- Practical Aspects Of Software Engineering @ PWr & UWr
- Nokia Academy @ Nokia
- Internal corporate trainings

PUBLIC SPEAKING EXPERIENCE

- code::dive conference
- code::dive community
- Academic Championships in Team Programming

HOBBIES

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

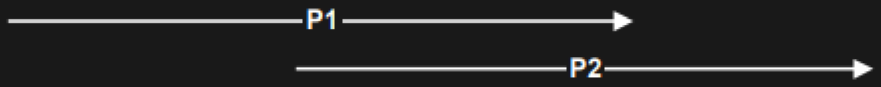
CONTRACT

- 🎰 Vegas rule
- 🗣️ Discussion, not a lecture
- ☕ Additional breaks on demand
- ⌚ Be on time after breaks

CONCURRENCY

WHAT IS CONCURRENCY?

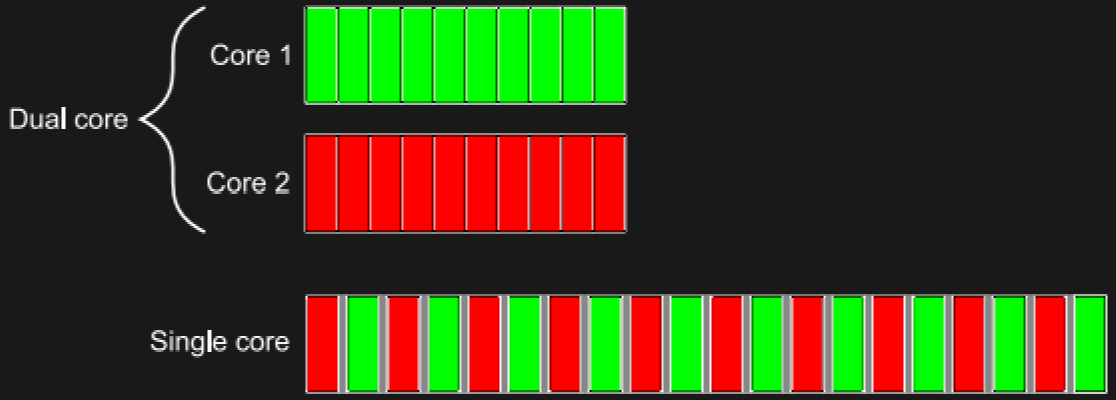
- Two processes are concurrent if one of them starts before the end of the other



- Concurrency is the process of performing two or more activities simultaneously

CONCURRENCY IN COMPUTER SYSTEMS

- Multiprocessor/multicore computers supporting multiple applications simultaneously
- Simulation of concurrency in single-processor systems using the mechanism of task switching



MODELS OF CONCURRENCY

	Threads	**Processes**
Stack	Separate	Separate
Heap	Common	Separate
Creation	Fast	Slow
Communication	Fast (heap) - pointers, data structures	Slower (IPC) - sockets, files, signals, pipes, shared memory

PROCESS

- The process is used to organize the execution of the program. One program consists of one or more processes. Thus, the process is the entire context necessary to execute the program.
- As a result of the process execution, the data segment, stack segment, and processor registers state can be changed.
- When a process is executed, the operating system allocates the necessary resources to the process (memory, CPU time, etc.).
- Synchronization, the way processes are handled, etc. is controlled by the operating system.
- There are one or more threads within each process.

THREAD

- Threads of the same process share most of the address space (code and data segment, open files, etc.).
- Switching thread context is relatively fast and doesn't burden operating system.
- Creating a thread requires less resources to run and is faster than creating a process.
- Easy (but also dangerous) communication between threads within one process.
- Each thread has a separate stack (function return address and local variables).

std::thread

THREAD - `std::thread`

- The thread is an object
- `#include <thread>`
- `std::thread`
- The most important operations:
  - constructor - starts a thread
  - `get_id()`
  - `join()`
  - `detach()`
  - `joinable()`

BASIC USAGE

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

void action() {
    cout << "Hello ";
    cout << this_thread::get_id();
    cout << " thread" << endl;
}

int main() {
    thread t(action);
    // can do other stuff here
    t.join();
    return 0;
}
```

```
$> g++ 01_hello.cpp –lpthread
$> ./a.out
Hello 47082117789440 thread
```

WHAT TO PASS TO `std::thread`?

Any callable, like:

- Function
- Functor (function object) - an object with `operator()`
- Lambda expression
- `std::function`
- Pointer to a function or pointer to a member function

The callable is **copied** to the memory area belonging to the newly created thread

WHAT TO PASS TO `std::thread`?

```cpp
struct Bar {
    void operator()() {
        std::cout << "Hello world\n";
    }
}

void foo() {
    std::cout << "Hello world\n";
}

int main() {
    std::thread t1(foo);
    Bar bar;
    std::thread t2(bar);
    std::thread t3([]() {
        std::cout << "Hello world\n";
    });
    t1.join();
    t2.join();
    t3.join();
}
```

```cpp
void foo() {
    std::cout << "Hello world\n";
}

class Bar {
public:
    void foo() {
        std::cout << "Hello world\n";
    }
};

int main() {
    std::thread t(&foo);
    t.join();

    class Bar bar;
    std::thread t1(&Bar::foo, bar);
    t1.join();
    return 0;
}
```

EMPTY THREADS (NOT-A-THREAD)

- Threads are started immediately after their creation, as long as we pass the so-called thread of execution or callable (function, functor, lambda). They are related to system threads.
- Threads are pinned to their variable in the thread that created it. Creating an empty thread `std::thread t;` does not start anything.
- An empty thread (Not-A-Thread) is not associated with any system thread and `join()` or `detach()` can't be called on it

EMPTY THREADS (NOT-A-THREAD)

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

int main() {
    thread t;
    t.join();   // not allowed on an empty thread
    t.detach(); // not allowed on an empty thread
    return 0;
}
```

```
$> g++ 03_join_empty_thread.cpp –lpthread
$> ./a.out
terminate called after throwing an instance of 'std::system_error'
    what():  Invalid argument
Aborted (core dumped)
```

PASSING PARAMETERS

TASK 1: PASSING PARAMETERS

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

int add(int a, int b)
{
    return a + b;
}

int main()
{
    // run add function in a thread
    // pass 3 and 4 as arguments
    return 0;
}
```

- Create a thread and run `add()` function on it, passing the numbers 3 and 4 to it

TASK 1 - SOLUTION

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

int add(int a, int b)
{
    return a + b;
}

int main()
{
    thread t(add, 5, 6);
    t.join();
    return 0;
}
```

- How to pass the result of a calculation back to `main()` function?
  - You can't do it through `return`, threads don't do that
  - You can write something in a global variable, but it's asking for trouble - synchronization
  - The correct way is to pass a reference to the variable that we will modify in the thread as a parameter

EXERCISE 2: PASSING REFERENCES

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

void add10(int & a)
{
    a += 10;
}

int main()
{
    // run add10 function in a thread
    // pass 5 as an argument and read it's value
    return 0;
}
```

- Create a thread and run `add10()` function on it, passing the number 5 to it
- Display the result on the screen in main thread

TASK 2 - SOLUTION

```
void add10(int & a) {
    a += 10;
}

int main() {
    int five = 5;
    thread t(add10, ref(five));
    cout << five << endl;
    t.join();
    cout << five << endl;
    return 0;
}
```

```
$> g++ zadanie2.cpp –lpthread
$> ./a.out
5
15
```

- `std::ref()` causes the object to be passed by reference

EXERCISE 3: PASSING A CLASS METHOD

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

class Car {
    int production_year;
    string model_name;
public:
    void setData(int year, const string & model) {
        production_year = year;
        model_name = model;
    }
    void print() {
        cout << model_name << " " << production_year << endl;
    }
};

int main() {
    Car toyota;
    // set year to 2015, model to "Corolla" in a new thread
    toyota.print();
    return 0;
}
```

- Create a thread and run `setData()` method on it, which will set year of production to 2015 and the model to "Corolla" on the toyota object.

TASK 3 - SOLUTION

```cpp
class Car {
    int production_year;
    string model_name;
public:
    void setData(int year, const string & model) {
        production_year = year;
        model_name = model;
    }
    void print() {
        cout << model_name << " " << production_year << endl;
    }
};

int main() {
    Car toyota;
    thread t(&Car::setData, &toyota, 2015, "Corolla");
    t.join();
    toyota.print();
    return 0;
}
```

```
$> g++ zadanie3.cpp –lpthread
$> ./a.out
Corolla 2015
```

TASK 3 - SOLUTION

- Why there is no `std::ref()` next to "Corolla" parameter?
  - temporary objects can be attached to `const &`
  - using `std::ref("Corolla")` will give us a reference to a temporary variable (in this case it's safe)
  - there is a danger of passing a dangling reference

PASSING PARAMETERS

- The constructor of the thread as the first parameter gets any callable object - lambda, function, function pointer, functor. Callable is copied into thread memory.
- Successive parameters of the thread's constructor are passed to the callable
- Parameters are forwarded (copied or moved) to thread memory
- Passing a reference is done by using `std::ref()`
- Passing a constant reference is done by using `std::cref()`
- A class method that is called in a thread takes a hidden pointer to the object on which it is to be called as the first parameter.

PASSING PARAMETERS

```
#include <thread>

void foo() { /* ... */ }
// foo() – function without parameters
std::thread t1(&foo);

void bar(int a, int & b) { /* ... */ }
int field = 5;
// bar(1, field) – function with parameters
std::thread t2(&bar, 1, std::ref(field));

struct SomeClass {
    void method(int a, int b, int & c) { /* ... */ }
};
SomeClass someObject;
// someObject.method(1, 2, field) – class method
std::thread t3(&SomeClass::method, &someObject, 1, 2, std::ref(field));
```

TRAPS WHEN PASSING ARGUMENTS

- What are the pitfalls of the code below?

```
void f(int i, std::string const& s);
void oops(int arg)
{
    char buffer[1024];
    sprintf(buffer, "%i", arg);
    std::thread t(f, 3, buffer);
    t.detach();
}
```

The implicit conversion takes place in a thread, so the object that is being converted may no longer exist.

TRAPS WHEN PASSING ARGUMENTS

- Pay attention to the passed arguments
  - If the variable is a pointer/reference, its lifespan should be longer than lifespan of the thread that operates on it
  - If there is a risk of implicit conversion, it's best to pass the converted argument straight away
- Solution - explicit conversion takes place in a current thread

```cpp
void f(int i, std::string const& s);
void not_oops(int arg)
{
    char buffer[1024];
    sprintf(buffer, "%i", arg);
    std::thread t(f, 3, std::string(buffer));
    t.detach();
}
```

TEMPLATE FUNCTIONS IN THREADS

```cpp
#include <iostream>
#include <thread>
#include <chrono>
using namespace std::chrono_literals;

template <typename Time>
void daemon(int number, Time time) {
    for (int i = 0 ; i < number; ++i) {
        std::cout << "Hi I'm thread with id: " << std::this_thread::get_id()
                  << " Number: " << number << '\n';
        std::this_thread::sleep_for(time);
    }
}

int main() {
    std::thread t1(daemon<decltype(1s)>, 20, 1s);
    // equivalent of std::thread t1(daemon<std::chrono::second>, 20, 1s);
    std::thread t2(daemon<decltype(1500ms)>, 15, 1500ms);
    std::thread t3(daemon<decltype(2700ms)>, 10, 2700ms);

    t1.detach();
    t2.detach();
    t3.detach();

    daemon(4, 7s);
```

`join()` OR `detach()`?

join() OR detach()?

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

void casualJob() {
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        this_thread::sleep_for(1s);
        cout << "Thread job done" << '\n';
    });
    casualJob();
    t.join();
    return 0;
}
```

```
$> g++ 04a_join.cpp –lpthread
$> ./a.out
Doing sth in casualJob
Thread job done
```

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

void casualJob() {
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        this_thread::sleep_for(1s);
        cout << "Thread job done" << '\n';
    });
    t.detach();
    casualJob();
    return 0;
}
```

```
$> g++ 04b_detach.cpp –lpthread
$> ./a.out
Doing sth in casualJob
```

join() OR detach()?

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

void casualJob() {
    this_thread::sleep_for(1s);
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        cout << "Thread job done" << '\n';
    });
    casualJob();
    t.join();
    return 0;
}
```

```
$> g++ 04c_join.cpp –lpthread
$> ./a.out
Thread job done
Doing sth in casualJob
```

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

void casualJob() {
    this_thread::sleep_for(1s);
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        cout << "Thread job done" << '\n';
    });
    t.detach();
    casualJob();
    return 0;
}
```

```
$> g++ 04d_detach.cpp –lpthread
$> ./a.out
Thread job done
Doing sth in casualJob
```

join() OR detach()?

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

void casualJob() {
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        cout << "Thread job done" << '\n';
    });
    // no join() or detach()
    casualJob();

    return 0;
}
```

```
$> g++ 05_no_join_no_detach.cpp —lpthread
$> ./a.out
Thread job done
Doing sth in casualJob
terminate called without an active exception
Aborted (core dumped)
```

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

void casualJob() {
    cout << "Doing sth in casualJob" << '\n';
}

int main() {
    thread t([] {
        cout << "Thread job done" << '\n';
    });
    casualJob();
    t.join();
    t.detach();
    return 0;
}
```

```
$> g++ 05_join_and_detach.cpp —lpthread
$> ./a.out
Doing sth in casualJob
Thread job done
terminate called after throwing an instance of
'std::system_error'
    what():  Invalid argument
Aborted (core dumped)
```

`join()` OR `detach()`?

- A thread must always be joined - `join()` or detached - `detach()`. Always.
- Thread destructor does not join nor detach it (no RAII 😔, but in C++20 we have `std::jthread`)
- Failure to join or detach a thread will result in a `std::terminate()` call which quits the application
- Thread method `joinable()` returns `true` if you can do `join()`
- `join()` can only be done once and is exclusive with `detach()`. Either one or the other should be used
- When we detach a thread, we usually do so immediately after creating it. After detaching, we can no longer refer to the thread using its name
- If we join a thread, we have to choose the right place to do it. `join()` is a blocking operation that waits for the thread to finish, so this is usually done at the end of the thread's starting function. If this function returns a thread, it can be joined even later.

RAII

RAII

- Resource Acquisition Is Initialization
- C++ language idiom (pattern) that guarantees security of resource handling
- Acquire the resource in the constructor
- Release the resource in the destructor
- Automatic release of the resource when an exception occurs, thanks to the stack unwinding mechanism
- Known classes implementing RAII:
  - `unique_ptr` - wrapper for a raw pointer
  - `shared_ptr` - wrapper for a raw pointer
  - `unique_lock` - wrapper for a `mutex`
  - `fstream` - wrapper for a file
- `std::thread` does not implement RAII 😔
- `std::thread` has a copy operation disabled
- `std::thread` can be moved like `unique_ptr` (move semantics, `std::move`)

TASK 4: scoped_thread

```cpp
#include <thread>
#include <stdexcept>
#include <chrono>
#include <iostream>
using namespace std;

class scoped_thread {
    // your implementation goes here
};

void do_sth(int) {
    this_thread::sleep_for(1s);
}

void do_sth_unsafe_in_current_thread() {
    throw runtime_error("Whoa!");
}

int main() {
    scoped_thread st(std::thread(do_sth, 42));
    // auto st2 = st; // copying not allowed
    [[maybe_unused]] auto st3 = move(st);
    scoped_thread st4(do_sth, 42);
    try {
        do_sth_unsafe_in_current_thread();
```

EXERCISE 4: SCOPED_THREAD

- Write RAII mechanism for thread - `scoped_thread`
- What operations should be included?
  - constructor takes a resource - `std::thread`
  - the constructor prevents us from creating an object if we pass an empty thread
  - the destructor is calling `join()`
  - copying is prohibited
  - moving is allowed
- Copying threads is a deleted operation, the compiler will not allow it
- Moving threads is allowed

TASK 4 - SOLUTION

```cpp
class scoped_thread {
    std::thread t_;
public:
    explicit scoped_thread(std::thread t)
        : t_{std::move(t)}
    {
        if (not t_.joinable()) {
            throw std::logic_error("No thread");
        }
    }

    template<typename ...Args>
    explicit scoped_thread(Args&&... args)
        : t_{ std::forward<Args>(args)... }
    { }

    ~scoped_thread() {
        if (t_.joinable()) {
            t_.join();
        }
    }
    scoped_thread(const scoped_thread &) = delete;
    scoped_thread(scoped_thread &&) = default;
    scoped_thread& operator=(const scoped_thread &) = delete;
    scoped_thread& operator=(scoped_thread &&) = default;
```

COPY ELISION (RVO)

COPY ELISION (RVO)

```
#include <thread>

std::thread f() {
    return std::thread([]{});
}

int main() {
    auto t = f();              // copy elision - ok
    // auto t2 = t;            // copying not allowed
    auto t3 = std::move(t);    // moving is ok
    t3.join();                 // join on t3, not t
    return 0;
}
```

- Copying a thread is prohibited
- Returning copies from functions is subject to the copy elision rules - the compiler optimizes the code by throwing away redundant copies
- RVO (Return Value Optimization) is a special case of copy elision
- If the local variable created in the function is returned by the copy RVO will happen
- The variable will be immediately created in the appropriate place on the stack, where it can be accessed from the upper frame of the stack
- With RVO, you can return threads from a function via a copy

RVO - CODE TRANSFORMATION

```
#include <thread>

std::thread f() {
    return std::thread([]{});
}

int main() {
    auto t = f();           // copy elision - ok
    t.join();
    return 0;
}
```

```
#include <thread>

void f(std::thread& t) {
    t{[]{}};
}

int main() {
    std::thread t;
    f(t);
    t.join();
    return 0;
}
```

DANGLING REFERENCE

PROBLEM

```
#include <thread>

void do_sth(int i) { /* ... */ }

struct A {
    int& ref_;
    A(int& a) : ref_(a) {}
    void operator()() {
        do_sth(ref_); // potential access to
                      // a dangling reference
    }
};

std::thread create_thread() {
    int local = 0;
    A worker(local);
    std::thread t(worker);
    return t;
} // local is destroyed, reference in worker is dangling

int main() {
    auto t = create_thread();  // Undefined Behavior
    auto t2 = create_thread(); // Undefined Behavior
    t.join();
    t2.join();
    return 0;
}
```

DANGLING REFERENCE/POINTER

- You have to ensure that the thread has correct access to the resources it uses during its lifetime, i.e. something is not deleted sooner. This should not be a surprise, because even in a single-threaded application you have to take care of it, otherwise we have Undefined Behavior (UB).
- This is the case when the thread is holding pointers or references to local objects and the thread is still alive when we exit the local function.
- Copying data to a thread is safe. Always prefer copying.
- See C++ Core Guidelines [CP.31]

EXCEPTIONS IN THREADS

PROBLEM

```cpp
#include <thread>
#include <iostream>

int main() {
    try {
        std::thread t1([]{
            throw std::runtime_error("WTF - What a Terrible Failure");
        });
        t1.join();
    } catch (const std::exception & ex) {
        std::cout << "Thread exited with exception: " << ex.what() << "\n";
    }
    return 0;
}
```

```
$> g++ 09_exceptions_not_working.cpp -lpthread
$> ./a.out
terminate called after throwing an instance of 'std::runtime_error'
    what():  WTF - What a Terrible Failure
Aborted (core dumped)
```

How to handle an exception from a different thread?

PROBLEM - EXCEPTIONS IN THREADS

```cpp
int main() {
    std::exception_ptr thread_exception = nullptr;
    std::thread t([](std::exception_ptr & te) {
        try {
            throw std::runtime_error("WTF");
        } catch (...) {
            te = std::current_exception();
        }
    }, std::ref(thread_exception));
    t.join();
    if (thread_exception) {
        try {
            std::rethrow_exception(thread_exception);
        } catch (const std::exception & ex) {
            std::cout << "Thread exited with an exception: "
            << ex.what() << "\n";
        }
    }
    return 0;
}
```

```
$> g++ 10_exceptions_working.cpp –lpthread
$> ./a.out
Thread exited with an exception: WTF
```

PROBLEM - EXCEPTIONS IN THREADS

- Each thread has its own stack so the stack unwinding mechanism works only on its stack
- You cannot normally catch exceptions from a thread other than the thread that raised the exception
- To catch an exception thrown from another thread use the exception pointer – `std::exception_ptr`
- The thread throwing the exception should assign the current exception to the exception pointer via `std::current_exception()`
- The thread that wants to catch an exception should check if `std::exception_ptr` has been set and if it is, throw that exception again with `std::rethrow_exception()`
- It is useful to use `noexcept` functions in threads to make sure no exceptions are thrown

AUXILIARY FUNCTIONS

HOW MANY THREADS SHOULD BE CREATED?

- Too many threads - the program runs slower
- Too few threads - no full use of available potential
- Use `std::thread::hardware_concurrency()`
  - Returns the number of available concurrent threads
  - May return 0 if such information is not obtainable

THREAD IDENTIFICATION

- `std::this_thread::get_id()`
  - Returns an object of `std::thread::id` type, representing the id of the current thread

```
std::thread::id master_thread;
void some_core_part_of_the_algorithm()
{
    if (std::this_thread::get_id() == master_thread) {
        do_master_thread_work();
    }
    do_common_work();
}
```

PUTTING THREADS TO SLEEP

- `std::this_thread::sleep_until(const chrono::time_point& sleep_time)`
  - blocks execution of the thread at least until the time point specified as a parameter
- `std::this_thread::sleep_for(const chrono::duration& sleep_duration)`
  - suspends execution of the current thread for (at least) the specified time interval
- `std::this_thread::yield()`
  - function to try to yield the current thread and allocate CPU time to another thread

PUTTING THREADS TO SLEEP

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    using namespace std::chrono_literals;
    std::cout << "Hello waiter\n" << std::flush;

    auto start = std::chrono::high_resolution_clock::now();
    std::this_thread::sleep_for(2s);
    auto end = std::chrono::high_resolution_clock::now();

    std::chrono::duration<double, std::milli> elapsed = end - start;
    std::cout << "Waited " << elapsed.count() << " ms\n";
}
```

Possible output:

```
Hello waiter
Waited 2000.12 ms
```

THREADS IN COLLECTIONS

TASK: THREADS IN COLLECTIONS

- Write a short program in which 20 threads are held in a vector
- Each thread has to wait 1 second, then display its number (taken as a parameter) and a space character
- Run the program several times and observe the results

TASK 5 - SOLUTION

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

void do_work(int id) {
    this_thread::sleep_for(1s);
    cout << id << ' ';
}

int main() {
    vector<thread> threads;
    for (int i = 0; i < 20; i++) {
        threads.emplace_back(do_work, i);
    }
    for (auto && t : threads) {
        t.join();
    }
    return 0;
}
```

TASK 5 - POSSIBLE RESULTS

```
$> ./a.out
0 1 3 2 5 6 8 10 12 7 11 9 13 4 14 15 17 16 19 18


$> ./a.out
4 5 7 6 14 15 16 18 13 11 9 2 0 8 10 17 12 1 19 3


$> ./a.out
0 1 2 4 5 3 78 10 116 13  9  14 12 15 1618 1719


$> ./a.out
2 1 0 43 685    7 101291115   13  17 14 16  18 19
```

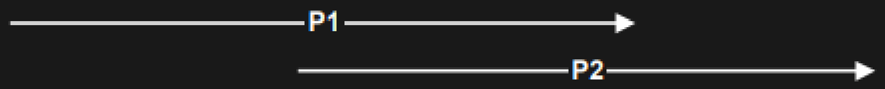
WHY IS THIS HAPPENING?

- The output stream `cout` is only one. It is a shared resource that is shared between threads
- There may be interlaces when accessing the stream (one thread will start typing something and will not finish, and the second one will interrupt it and enter its number)
- Sharing resources is a common threading problem
- How to deal with this? This is a topic for the next lesson 🙂

SUMMARY

WHAT IS CONCURRENCY?

- Two processes are concurrent if one of them starts before the end of the other



- Concurrency is the process of performing two or more activities simultaneously

WHEN TO USE CONCURRENCY?

- Distribution of tasks
- Performance:
    - Parallel tasks (division of a task into parts),
    - Data parallelization (performing the same tasks on different pieces of data)

WHEN NOT TO USE CONCURRENCY?

- When too many threads can reduce performance instead of increasing it (cost of starting a thread, cost of resources - each thread typically uses 8MB of memory),
- When the performance gain is not proportional to the effort involved and the complexity of the code (code maintenance cost is equally important).

EXERCISE 6 / HOMEWORK

- Implement the `std::accumulate` algorithm using multithreading

```
template< class InputIt, class T>
T accumulate( InputIt first, InputIt last, T init);
```

```
template<class InputIt, class T>
constexpr T accumulate(InputIt first, InputIt last, T init)
{
    for (; first != last; ++first) {
        init = std::move(init) + *first; // std::move since C++20
    }
    return init;
}
```

TIP 1

- Calculate the number of possible threads
- Create a container for storing the results

```
const size_t hardwareThread = std::thread::hardware_concurrency();
const size_t neededThreads = std::min(size / minimumSize, hardwareThread);
const size_t chunkSize = size / neededThreads;

std::cout << "NeededThreads: " << neededThreads << std::endl;
std::cout << "ChunkSize: " << chunkSize << std::endl;
std::vector<std::thread> threads(neededThreads - 1);
std::vector<T> results(neededThreads);
```

TIP 2

- Implementation of a functor for threads
  - The functor accepts successive portions of data
  - Compute and pass iterators to the beginning and end of the range on which the thread is to operate.
  - The variable holding the result should also be passed by reference

```
std::thread([](IT first, IT last, T& result) {
                result = std::accumulate(first, last, T{});
            },
            begin,
            end,
            std::ref(results[i]));
```

TIP 3

- You should also use the current thread on which the function is called, so that it does not wait idly for the other threads to finish.

```
auto begin = first;
for (size_t i = 0; i < neededThreads - 1; ++i) {
    auto end = std::next(begin, chunkSize);
    threads[i] = std::thread([](IT first, IT last, T& result) {
                                     result = std::accumulate(first, last, T{});
                                 },
                                 begin,
                                 end,
                                 std::ref(results[i]));
    begin = end;
}
results[neededThreads - 1] = std::accumulate(begin, last, T{});
```

WAS THE TASK DIFFICULT?

- What was the biggest problem?
- Can you simplify the algorithm?
- Can other mechanisms be used to facilitate implementation?

HOMEWORK

Implement the `count_if` algorithm using multithreading

USEFUL LINKS

- `std::thread` (cppreference.com)
- `std::ref` (cppreference.com)
- C ++ Core Guidelines on Concurrency and Parallelism
- Top 20 C ++ multithreading mistakes and how to avoid them

THANK YOU 🙂