

Wątki

CODERS SCHOOL

<https://coders.school>



Łukasz Ziobroń
lukasz@coders.school

Łukasz Ziobroń & Bartosz Szurgot - autorzy



Łukasz Ziobroń

Not only programming experience:

- C++ and Python developer @ Nokia & Credit Suisse
- Scrum Master @ Nokia & Credit Suisse
- Code Reviewer @ Nokia
- Webmaster (HTML, PHP, CSS) @ StarCraft Area

Training experience:

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

Public speaking experience:

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



Wątki - Agenda

- Wątek - `std::thread`
- Podstawowe użycie
- Podstawowe użycie z `lambda`
- Zadanie 1: przekazywanie parametrów
- Zadanie 2: przekazywanie referencji
- Zadanie 3: przekazywanie metody klasy
- Przekazywanie parametrów
- Puste wątki (Not-A-Thread)
- `join()` czy `detach()`?
- RAI
- Zadanie 4: `scoped_thread`
- Copy ellision (RVO)
- Problem - wisząca referencja
- Problem - wyjątki w wątkach
- Zadanie 5: wątki w kolekcjach

Wątek – `std::thread`

- Wątek jest obiektem
- `#include <thread>`
- `std::thread`
- Najważniejsze operacje:
 - constructor – uruchamia wątek
 - `get_id()` – pobranie identyfikatora wątku
 - `join()` – przyłączenie wątku
 - `detach()` – odłączenie wątku
 - `joinable()` – czy można przyłączyć wątek

Wątek – `std::thread`

- Funkcje i klasy pomocnicze dla wątków w bibliotece standardowej
 - `std::thread::hardware_concurrency()`

zwraca liczbę dostępnych wątków współbieżnych. Funkcja ta może zwrócić 0, jeśli taka informacja nie będzie możliwa do uzyskania. Utworzenie większej liczby wątków jest możliwe i nazywa się *oversubscription*. Efekty przełączania kontekstu mogą mieć jednak negatywny wpływ na wydajność.
 - `std::this_thread`
 - `sleep_for(const chrono::duration<Rep, Period>& sleep_duration)`

wstrzymuje wykonanie bieżącego wątku na (przynajmniej) określony interwał czasu
 - `sleep_until(const chrono::time_point<Clock, Duration>& sleep_time)`

blokuje wykonanie wątku przynajmniej do podanego jako parametr punktu czasu
 - `yield()`

funkcja umożliwiająca podjęcie próby wywłaszczenia bieżącego wątku i przydzielenia czasu procesora innemu wątkowi
 - `get_id()`

zwraca obiekt typu `std::thread::id` reprezentujący identyfikator bieżącego wątku

Podstawowe użycie

```
#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
```

Podstawowe użycie z lambda

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

int main()
{
    thread t([]
    {
        cout << "Hello ";
        cout << this_thread::get_id();
        cout << " thread" << endl;
    });
    // can do other stuff here
    t.join();
    return 0;
}
```

```
$> g++ 02_hello_lambda.cpp -lpthread
```

```
$> ./a.out
```

```
Hello 47937732544256 thread
```


Zadanie 1: przekazywanie parametrów

```
#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;
}
```

- Utwórz wątek i odpal w nim funkcję `add()` przekazując do niej liczby 3 i 4

Zadanie 1 - rozwiązanie

```
#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;
}
```

- Utwórz wątek i odpal w nim funkcję `add()` przekazując do niej liczby 3 i 4
- Jak przekazać wynik obliczeń z powrotem do funkcji `main()`?
 - Nie da się poprzez `return`, wątki do tego nie służą
 - Można zapisać coś w globalnej zmiennej, ale to proszenie się problemy – synchronizacja
 - Właściwy sposób to przekazanie jako parametr referencję do zmiennej, którą zmodyfikujemy w wątku

Zadanie 2: przekazywanie referencji

```
#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;
}
```

- Utwórz wątek i odpal w nim funkcję `add10()` przekazując do niej liczbę 5
- Wypisz wynik na ekran

Zadanie 2 - rozwiązanie

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

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;
}
```

- Utwórz wątek i odpal w nim funkcję `add10()` przekazując do niej liczbę 5
- Wypisz wynik na ekran
- `std::ref()` powoduje, że przekazujemy obiekt przez referencję

```
$> g++ zadanie2.cpp -lpthread
```

```
$> ./a.out
```

```
5
```

```
15
```

Zadanie 3: przekazywanie metody klasy

```
#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 thread
    toyota.print();
    return 0;
}
```

- Utwórz wątek i odpal w nim metodę setData(), która ustawi w obiekcie toyota rok produkcji na 2015, a model na "Corolla"

Zadanie 3 - rozwiązanie

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

- Utwórz wątek i odpal w nim metodę `setData()`, która ustawi w obiekcie `toyota` rok produkcji na 2015, a model na "Corolla"
- Dlaczego przy parametrze "Corolla" nie ma `std::ref()`?
 - obiekty tymczasowe można podpiąć pod const &
 - użycie `std::ref("Corolla")` da nam referencję do zmiennej tymczasowej (w tym przypadku jest to bezpieczne)
 - istnieje niebezpieczeństwo przekazania wiszącej referencji (dangling reference)

```
$> g++ zadanie3.cpp -lpthread
```

```
$> ./a.out
```

```
Corolla 2015
```

Przekazywanie parametrów

- Konstruktor wątku jako pierwszy parametr dostaje jakikolwiek "wołalny" obiekt (callable) - lambda, funkcja, funktor. Callable jest **kopiowane** do pamięci wątku.
- Kolejne parametry konstruktora wątku są przekazywane do funkcji (lambdy)
- Parametry są forwardowane (**kopiowane** lub przenoszone) do pamięci wątku.
- Przekazanie referencji odbywa się przez użycie `std::ref()`
- Przekazanie **stałej** referencji odbywa się przez użycie `std::cref()`
- Metoda klasy odpalana w wątku jako pierwszy ukryty parametr przyjmuje wskaźnik do obiektu, na którym ma zostać wywołana

Przekazywanie parametrów

```
#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));
```


Puste wątki (Not-A-Thread)

```
#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;
}
```

- Wątki są odpalane od razu po ich utworzeniu, o ile tylko przekażemy do nich tzw. thread of execution lub callable (funkcja, funktor, lambda). Są one powiązane z wątkami systemowymi.
- Wątki są przypięte do swojej zmiennej w wątku, który go stworzył. Stworzenie pustego wątku `std::thread t;` nie odpala niczego.
- Pusty wątek (Not-A-Thread) nie jest powiązany z żadnym wątkiem systemowym i nie woła się na nim `join()` lub `detach()`

```
$> 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)
```

join() czy detach()?

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

void casualJob() {
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        this_thread::sleep_for(1s);
        cout << "Thread job done" << endl;
    });
    casualJob();
    t.join();
    return 0;
}
```

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

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

void casualJob() {
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        this_thread::sleep_for(1s);
        cout << "Thread job done" << endl;
    });
    t.detach();
    casualJob();
    return 0;
}
```

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

join() czy detach()?

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

void casualJob() {
    this_thread::sleep_for(1s);
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        cout << "Thread job done" << endl;
    });
    casualJob();
    t.join();
    return 0;
}
```

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

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

void casualJob() {
    this_thread::sleep_for(1s);
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        cout << "Thread job done" << endl;
    });
    t.detach();
    casualJob();
    return 0;
}
```

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

join() czy detach()?

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

void casualJob() {
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        cout << "Thread job done" << endl;
    });
    // no join() or detach()
    casualJob();
    return 0;
}
```

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


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

void casualJob() {
    cout << "Doing something in casualJob" << endl;
}

int main() {
    thread t([] {
        cout << "Thread job done" << endl;
    });
    casualJob();
    t.join();
    t.detach();
    return 0;
}
```

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

join() czy detach()?

- Wątek należy zawsze przyłączyć - `join()` lub odłączyć - `detach()`. Zawsze.
- Destraktor wątku nie przyłącza go ani nie odłącza (brak RAI )
- Brak przyłączenia lub odłączenia wątku spowoduje zawołanie `std::terminate()`, które ubija aplikację
- Metoda wątku `joinable()` zwraca `true`, jeśli można zrobić `join()`
- `join()` można zrobić tylko raz i wyklucza się on z `detach()`. Należy użyć albo jednego albo drugiego
- Jeśli wątek odłączamy, to zazwyczaj robimy to od razu po jego utworzeniu. Po odłączeniu nie możemy się już odwołać do wątku używając jego zmiennej
- Jeśli wątek przyłączamy, to musimy wybrać właściwe miejsce na jego przyłączenie. `join()` jest operacją blokującą, która czeka, aż wątek zakończy pracę, więc zazwyczaj robi się to na końcu funkcji odpalającej wątek. Jeśli funkcja ta zwraca wątek, to można go przyłączyć jeszcze później.

RAII

- Resource Acquisition Is Initialization
- Idiom (wzorzec) języka C++ gwarantujący bezpieczeństwo obsługi zasobów
- Pozyskanie zasobu w konstruktorze
- Zwolnienie zasobu w destruktorze
- Automatyczne zwolnienie zasobu przy wystąpieniu wyjątku, dzięki mechanizmowi odwijania stosu
- Znane klasy implementujące RAII:
 - `unique_ptr` – wrapper na zwykły wskaźnik
 - `shared_ptr` – wrapper na zwykły wskaźnik
 - `unique_lock` – wrapper na mutex
 - `fstream` – wrapper na plik
- `std::thread` nie implementuje RAII ☹️
- `std::thread` ma zablokowaną operację kopiowania
- `std::thread` może być przenoszony tak jak `unique_ptr` (semantyka przenoszenia, `std::move`)

Zadanie 4: scoped_thread

```
#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
    auto st3 = move(st);
    try {
        do_sth_unsafe_in_current_thread();
    } catch (const exception & e) {
        cout << e.what() << endl;
    }
    return 0;
} // thread is safely destroyed
```

- Napisz mechanizm RAI na wątek - `scoped_thread`
- Jakie operacje powinny zostać uwzględnione?
 - konstruktor przyjmuje zasób - `std::thread`
 - konstruktor nie pozwala utworzyć obiektu, jeśli prześlemy pusty wątek
 - destruktor woła `join()`
 - kopiowanie jest zabronione
 - przenoszenie jest dozwolone
- Kopiowanie wątków jest operacją usuniętą, kompilator na to nie pozwoli
- Przenoszenie wątków jest dozwolone

Zadanie 4 - rozwiązanie

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

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");
        }
    }
    ~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;
};
```

```
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
    auto st3 = move(st);
    try {
        do_sth_unsafe_in_current_thread();
    } catch (const exception & e) {
        cout << e.what() << endl;
    }
    return 0;
} // thread is safely destroyed
```


Copy ellision (RVO)

```
#include <thread>

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

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

- Kopiowanie wątku jest zabronione
- Zwracanie kopii z funkcji podlega zasadom copy ellision – kompilator optymalizuje kod, poprzez wyrzucenie zbędnego kopiowania
- RVO (Return Value Optimisation) to szczególny przypadek copy ellision
- Jeśli zmienna lokalna utworzona w funkcji jest zwraca przez kopię nastąpi RVO
- Zmienna zostanie od razu utworzona w odpowiednim miejscu na stosie, gdzie jest możliwy dostęp do niej z poziomu wyższej ramki stosu
- Dzięki RVO można zwracać wątki z funkcji poprzez kopię

Problem - wisząca referencja

```
#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;
}
```

- Trzeba zapewnić, że wątek ma poprawny dostęp do zasobów z których korzysta w czasie swojego życia, czyli np. coś nie jest usuwane wcześniej. To nie powinno być zaskoczeniem, bo nawet w jednowątkowej aplikacji trzeba o to dbać, inaczej mamy Undefined Behavior (UB).
- Taki przypadek zachodzi, gdy wątek trzyma wskaźniki lub referencje do lokalnych obiektów i wątek ciągle żyje, gdy wychodzimy z lokalnej funkcji.
- Kopiowanie danych do wątku jest bezpieczne. Jeśli pracujesz na małych porcjach danych nie wymagających modyfikacji zawsze preferuj kopiowanie.
- Zobacz [C++ Core Guidelines \[CP.31\]](#)

Problem - wyjątki w wątkach

```
#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)
```

Problem - wyjątki w wątkach

```
#include<iostream>
#include<thread>
#include<exception>
#include<stdexcept>

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;
}
```

- Nie można standardowo złapać wyjątków w innym wątku niż tym, który rzucił wyjątek
- Aby przechwycić wyjątek rzucony z innego wątku należy użyć wskaźnika na wyjątek - `std::exception_ptr`
- Wątek rzucający wyjątek powinien przypisać do wskaźnika na wyjątek obecny wyjątek za pomocą `std::current_exception()`
- Wątek, który chce złapać wyjątek powinien sprawdzić, czy `std::exception_ptr` został ustawiony i jeśli tak jest rzucić ten wyjątek ponownie poprzez `std::rethrow_exception()`
- Warto używać w wyjątkach funkcji `noexcept`, aby mieć pewność, że wyjątki nie będą rzucone

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

Zadanie 5: wątki w kolekcjach

- Napisz krótki program, w którym 20 wątków jest trzymane w wektorze.
- Każdy wątek ma za zadanie poczekać 1 sekundę, po czym wyświetlić swój numer, który przyjmuje jako parametr oraz znak nowej linii.
- Tworzenie wątków i ich przyłączanie powinno zostać zrealizowane w 2 oddzielnych pętlach
- Uruchom program kilka razy i zaobserwuj jakie daje wyniki

Zadanie 5 - rozwiązanie

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

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

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

- Napisz krótki program, w którym 20 wątków jest trzymane w wektorze.
- Każdy wątek ma za zadanie poczekać 1 sekundę, po czym wyświetlić swój numer, który przyjmuje jako parametr oraz znak nowej linii.
- Tworzenie wątków i ich przyłączanie powinno zostać zrealizowane w 2 oddzielnych pętlach
- Uruchom program kilka razy i zaobserwuj jakie daje wyniki

Zadanie 5 – możliwe rezultaty

\$> ./a.out	\$> ./a.out	\$> ./a.out	\$> ./a.out
0	4	18	0
1	5	016	1
3	7	2	16
2	6	3	3
5	14	4	199
6	15	5	815
8	16	6	11
10	18	15	5
12	13	8	
7	11	9	18
11	9	10	17
9	2	11	4
13	0	12	12
4	8	13	13
14	10	14	7
15	17	19	
17	12		10
16	1	1	2
19	19	7	6
18	3	17	14

- Dlaczego tak się dzieje?
- Strumień wyjściowy cout jest tylko jeden. Jest on wspólnym zasobem współdzielonym między wątkami
- Może dochodzić do przeplotów w trakcie dostępu do strumienia (jeden wątek zacznie coś wpisywać i nie skończy, a już drugi wejdzie mu w paradę i wpisze swój numer)
- Współdzielenie zasobów to typowy problem wielowątkowości
- Jak sobie z tym poradzić? To już temat na kolejną lekcję 😊

Przydatne linki

- [std::thread \(cppreference.com\)](http://cppreference.com/std/thread)
- [std::ref \(cppreference.com\)](http://cppreference.com/std/ref)
- [C++ Core Guidelines on Concurrency and Parallelism](#)
- [Top 20 C++ multithreading mistakes and how to avoid them](#)

CODERS SCHOOL

<https://coders.school>

ASK A NINJA



Łukasz Ziobroń
lukasz@coders.school