



Uniwersytet  
Wrocławski

# Programowanie współbieżne w C++11

Zbigniew Koza

Wydział Fizyki i Astronomii

Co to jest współbieżność?

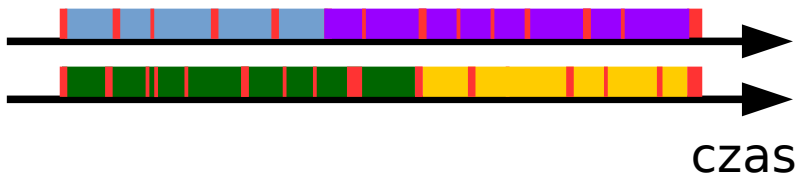
# Przetwarzanie:



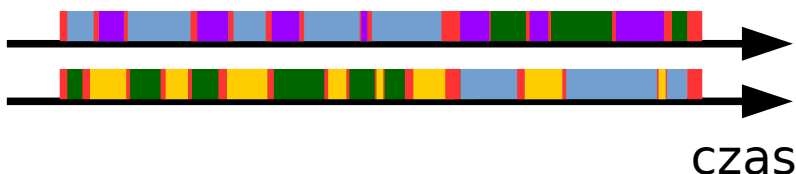
- Szeregowo



- **Współbieżne**



- Równoległe



- **Równoległe  
i współbieżne**

# Problemy

- Synchronizacja
- Bezpieczna wymiana informacji
- Nigdy nie wiemy, kiedy nasz wątek zostanie wywłaszczony
- Potrzebujemy mechanizmów pozwalających stwierdzić, czy/kiedy zasoby współdzielone z innymi wątkami zostały zmodyfikowane

```
#include <thread>
```

# Pierwszy program

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

void my_fun_pl()
{
    std::cout << u8"Witaj, świecie!\n";
}

int main()
{
    std::thread th(&my_fun_pl);
    std::cout << "Hello, world!\n";
    th.join();
}
```

# Pierwszy program

- **#include <thread>**

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

void my_fun_pl()
{
    std::cout << u8"Witaj, świecie!\n";
}

int main()
{
    std::thread th(&my_fun_pl);
    std::cout << "Hello, world!\n";
    th.join();
}
```

# Pierwszy program

- `#include <thread>`
- **Definicja obiektu zarządzającego funkcją wywoływaną współbieżnie**

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

void my_fun_pl()
{
    std::cout << u8"Witaj, świecie!\n";
}

int main()
{
    std::thread th(&my_fun_pl);
    std::cout << "Hello, world!\n";
    th.join();
}
```



# Pierwszy program

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

void my_fun_pl()
{
    std::cout << u8"Witaj, świecie!\n";
}

int main()
{
    std::thread th(&my_fun_pl);
    std::cout << "Hello, world!\n";
    th.join();
}
```

- `#include <thread>`
- Definicja obiektu zarządzającego funkcją wywoływaną wspólnie
- **Synchronizacja z wątkiem głównym**

# Kompilacja

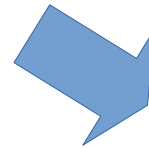
```
> g++ 1.cpp -pthread
```

# Wynik

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

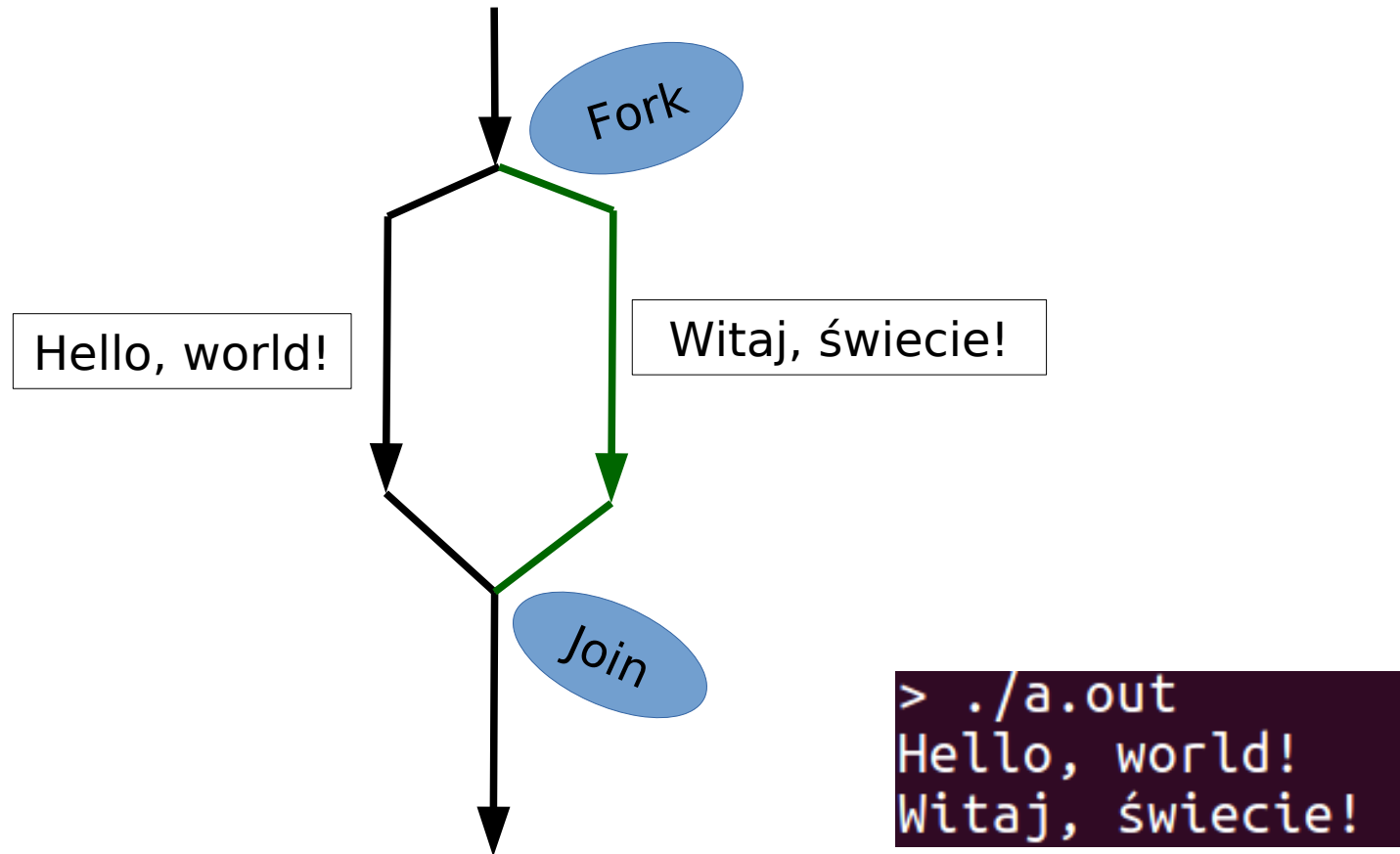
void my_fun_pl()
{
    std::cout << u8"Witaj, świecie!\n";
}

int main()
{
    std::thread th(&my_fun_pl);
    std::cout << "Hello, world!\n";
    th.join();
}
```

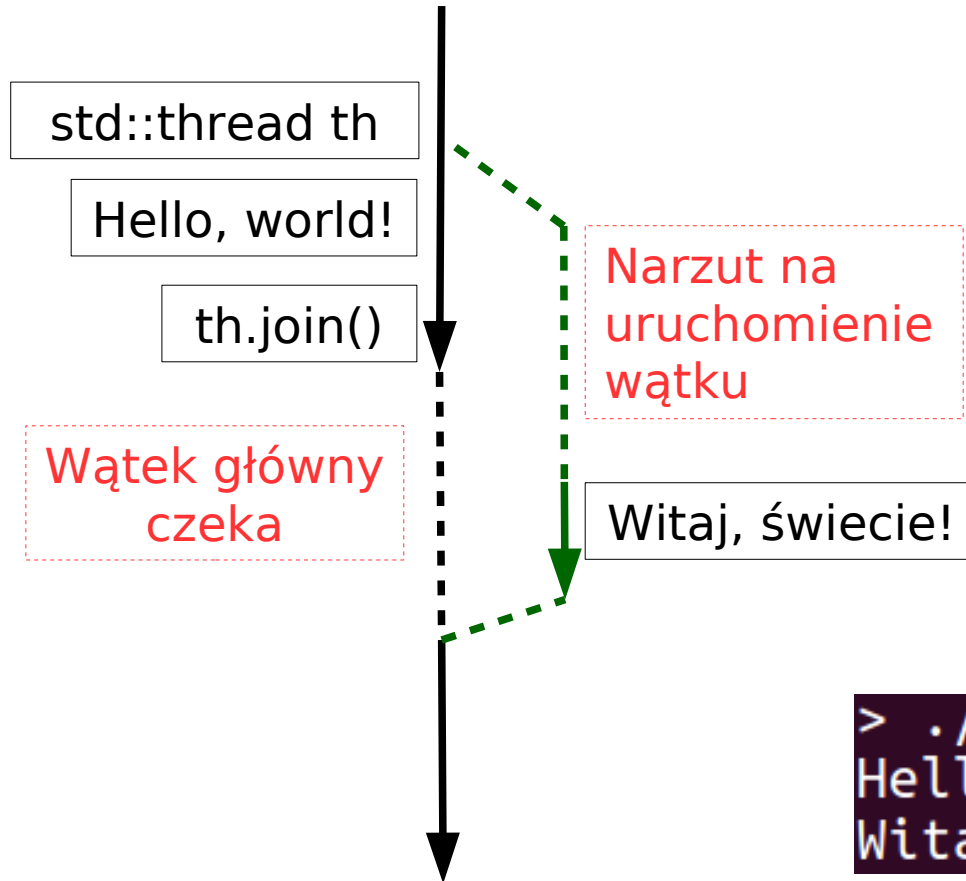


```
> ./a.out
Hello, world!
Witaj, świecie!
```

# Fork & join



# Fork & join



```
> ./a.out  
Hello, world!  
Witaj, świecie!
```

# Uwaga

- Potrafimy uruchomić drugi wątek, ale wciąż nie znamy mechanizmów synchronizacji i wymiany danych...
- Wciąż nie potrafimy w bezpieczny i efektywny sposób korzystać ze współbieżności...

# Przykład 2

```
#include <iostream>    // std::cout
#include <thread>        // std::thread, std::this_thread::sleep_for
#include <chrono>        // std::chrono::seconds

void pause_thread(int n)
{
    std::this_thread::sleep_for (std::chrono::seconds(n));
    std::cout << "pause of " << n << " seconds ended\n";
}

int main()
{
    std::cout << "Spawning 3 threads...\n";
    std::thread t1 (pause_thread, 1);
    std::thread t2 (pause_thread, 2);
    std::thread t3 (pause_thread, 3);
    std::cout << "Done spawning threads. Now waiting for them to join:\n";
    t1.join();
    t2.join();
    t3.join();
    std::cout << "All threads joined!\n";
    return 0;
}
```

// źródło: <http://www.cplusplus.com/reference/thread/thread/join/>

# Przykład 2

```
#include <iostream>    // std::cout
#include <thread>       // std::thread, std::this_thread::sleep_for
#include <chrono>       // std::chrono::seconds

void pause_thread(int n)
{
    std::this_thread::sleep_for (std::chrono::seconds(n));
    std::cout << "pause of " << n << " seconds ended\n";
}

int main()
{
    std::cout << "Spawning 3 threads...\n";
    std::thread t1 (pause_thread, 1);
    std::thread t2 (pause_thread, 2);
    std::thread t3 (pause_thread, 3);
    std::cout << "Done spawning threads. Now waiting for them to join:\n";
    t1.join();
    t2.join();
    t3.join();
    std::cout << "All threads joined!\n";
    return 0;
}
```

```
Spawning 3 threads...
Done spawning threads. Now waiting for them to join:
pause of 1 seconds ended
pause of 2 seconds ended
pause of 3 seconds ended
All threads joined!
```



# Funkcja składowa join

- Metoda `join` czeka na zakończenie funkcji, którą zarządza dany obiekt, i dopiero wtedy sama kończy swoje działanie
- Metoda ta tworzy więc **barierę** dla wątku głównego
- Jest to jedna z metod **synchronizacji** wątku głównego z wątkami pobocznymi

# Alternatywa dla join

- `thread.detach()`
- Tworzy „wątek działający w tle”  
(*daemon thread*)

# Nie ma alternatywy dla `join`

- ~~• `thread.detach()`~~
- ~~• Tworzy „wątek działający w tle”  
(*deamon thread*)~~
- Nie kombinuj, nie komplikuj sobie życia

# Co, jeśli zapomnimy o join?

```
~thread()  
{  
    if (joinable())  
        std::terminate();  
}
```

- Program „pada”



```
Spawning 3 threads...  
Done spawning threads. Now waiting for them to join:  
pause of 1 seconds ended  
pause of 2 seconds ended  
pause of 3 seconds ended  
All threads joined!  
terminate called without an active exception  
Przerwane (zrzut pamięci)
```

# Dlaczego join nie jest wywoływane w destruktorze?

- Jeśli w wątku głównym zgłoszono by **wyjątek**, to automatyczne uruchomienie join w destruktorze `std::thread` mogłoby zablokować wątek główny (bariera!), a więc i obsługę zgłoszonego wyjątku, zwłaszcza jeśli wątek poboczny oczekiwałby na dane z zawieszzonego wątku głównego
- => *deadlock* (zakleszczenie)

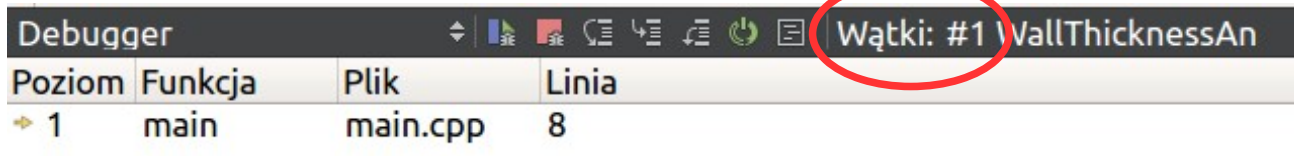
```
~thread()  
{  
    if (joinable())  
        std::terminate();  
}
```

# Dygresja: debugging

- gdb:

```
(gdb) info threads
Id      Target Id      Frame
* 1      Thread 0x7ffff7fb3740 (LWP 12369) "a.out" main () at p1a_join.cpp:20
 3      Thread 0x7ffff674d700 (LWP 12371) "a.out" 0x00007ffff7632c1d in nanosleep ()
    at ../sysdeps/unix/syscall-template.S:84
 4      Thread 0x7ffff5f4c700 (LWP 12372) "a.out" 0x00007ffff7632c1d in nanosleep ()
    at ../sysdeps/unix/syscall-template.S:84
```

- QtCreator:



- Valgind + helgrind

```
Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
  at 0x400606: main (simple_race.c:13)
```

# Zamiast funkcji może być lambda

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

int main()
{
    std::thread th([]()
    {
        std::cout << u8"Witaj, świecie!\n";
    });
    std::cout << "Hello, world!\n";
    th.join();
}
```

# Grupa wątków

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread ([]{
            std::cout << "Witaj, świecie!\n";
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

- **std::thread**  
kopiowany jest  
z użyciem  
*move semantics*
- **std::thread**  
nie może  
posiadać  
prawdziwej kopii!



# Grupa wątków

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread ([]{
            {
                std::cout << "Witaj, świecie!\n";
            }
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```



```
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
Hello, world!
Witaj, świecie!
Witaj, świecie!
Witaj, świecie!
```

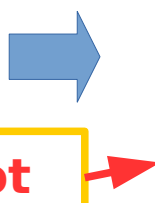
# Grupa wątków

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread([i]()
        {
            std::cout << "Witaj, świecie!\n";
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

**Przeplot**



Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Hello, world!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!

# Kolejne uruchomienie...

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread ([])()
        {
            std::cout << "Witaj, świecie!\n";
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

[illegible]

# Kolejne uruchomienie...

- Niemal za każdym razem inny wynik...

```
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Hello, world!  
Witaj, świecie!
```

```
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Hello, world!  
Witaj, świecie!  
Witaj, świecie!
```

```
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Witaj, świecie!  
Hello, world!  
Witaj, świecie!  
Witaj, świecie!
```

- To jest bardzo niekorzystne zjawisko!

# Skomplikujmy funkcję...

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread ([i]()
        {
            std::cout << "Witaj, świecie nr " << i << "\n";
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```



# Wynik...

```
Witaj, świecie nr Witaj, świecie nr Witaj, świecie nr 6  
Witaj, świecie nr 7  
Witaj, świecie nr 4  
Witaj, świecie nr 5  
1  
Witaj, świecie nr 3  
Hello, world!  
Witaj, świecie nr 02  
Witaj, świecie nr 9  
Witaj, świecie nr 8
```

- Za każdym razem coś innego
- „Przeplot” jest nieznośny
- Program jest niedeterministyczny :-(

# Wyścig (race condition)

- Wyścig to sytuacja, gdy wynik działania kilku procesów lub wątków zależy od względnej kolejności wykonywania poszczególnych operacji w każdym z nich
  - kilka wątków/procesów w tym samym czasie usiłuje modyfikować (i być może odczytywać) ten sam zasób, np. plik lub pamięć współdzieloną
  - Przykład: dwa niesynchronizowane programy/wątki wyświetlające dane na tej samej konsoli

# Wyścig - przykład

Thread 1	Thread 2		Integer value
			0
read value		←	0
increase value			0
write back		→	1
	read value	←	1
	increase value		1
	write back	→	2

OK

Thread 1	Thread 2		Integer value
			0
read value		←	0
	read value	←	0
increase value			0
	increase value		0
write back		→	1
	write back	→	1

wyścig



# Wyścig = błąd

- Program, w którym zachodzi wyścig:
  - jest niedeterministyczny
  - jest niezwykle trudny do testowania
  - jest **błędny**
- Diagnostyka wyścigu jest bardzo trudna, dlatego lepiej zapobiegać niż leczyć
- Zapobieganie polega na stosowaniu określonych reguł/idiomów programowania (język nie daje żadnych gwarancji)

# Diagnostyka wyścigu

```
4 void my_fun_pl()
5 {
6   std::cout << u8"Witaj, świecie!\n";
7 }
8
9 int main()
10 {
11   std::thread th(&my_fun_pl);
12   std::cout << "Hello, world!\n";
13   th.join();
14 }
```

np. Valgrind

Dołącz kod  
źródłowy

```
> g++ -pthread -g p1.cpp
> valgrind --tool=helgrind ./a.out
```

```
Hello, world!
==14608== -----
==14608==
==14608== Possible data race during write of size 8 at 0x604158 by thread #1
==14608== Locks held: none
==14608==   at 0x4F4EE25: std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<
r> >&, char const*, long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==14608==   by 0x4F4F236: std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::
const*) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==14608==   by 0x400F3F: main (p1.cpp:12)
==14608==
==14608== This conflicts with a previous write of size 8 by thread #2
==14608== Locks held: none
==14608==   at 0x4F4EE25: std::basic_ostream<char, std::char_traits<char> >& std::__ostream_insert<
r> >&, char const*, long) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==14608==   by 0x4F4F236: std::basic_ostream<char, std::char_traits<char> >& std::operator<< <std::
const*) (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.21)
==14608==   by 0x400EFA: my_fun_pl() (p1.cpp:6)
==14608==   by 0x40249C: void std::Bind_simple<void (*())()>::M_invoke<>(std::Index_tuple<>) (fun
==14608==   by 0x4023E5: std::Bind_simple<void (*())()>::operator()() (functional:1520)
```

# Programowanie współbieżne...

- **Jest trudne!** (jeśli program ma być bullet-proof)
- Gdyż trudno je *w pełni* zautomatyzować
- Żeby uniknąć nieoczywistych błędów, trzeba rozumieć (prawie) wszystkie zagadnienia języka na poziomie programów szeregowych
  - W C++: różne rodzaje funkcji, argumenty i wartości funkcji, referencje, semantyka *move*, konstruktor/destruktor, dziedziczenie, szablony, wyjątki, RAII, STL, iteratory, czas życia zmiennych, stos/sterta programu, funktory/wyrażenia lambda...

„Kopiowanie” wątków

# Potrafimy przenosić wątki nienazwane

```
#include <iostream>
#include <thread>
#include <vector>

const int N = 10;

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        workers.push_back(std::thread ([]{
            std::cout << "Witaj, świecie!\n";
        }));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

- **std::thread**  
kopiowany jest  
z użyciem  
*move semantics*
- std::thread  
nie może  
posiadać  
prawdziwej kopii!

# Wątki nazwane

- Jak pracować z wątkami nazwanymi (tzw. l-values)?

# Jak „skopiować” nazwane wątki?

```
9   std::vector<std::thread> workers;
10  for (int i = 0; i < N; i++)
11  {
12      std::thread th ([i]()
13      {
14          std::cout << "Witaj, świecie nr " << i << "\n";
15      });
16      workers.push_back(th);
17  }
```

To nie działa

Kontenery STL: przechowują wartości

**error:** use of deleted function 'std::thread::thread(const std::thread&)'


In file included from p5.cpp:2:0:  
/usr/include/c++/5/thread:126:5: **note:** declared here  
thread(const thread&) = delete;

Konstruktor kopiujący jest „zakazany”

# std::move

- Musimy jawnie wskazać, że chcemy do wektora przesunąć (a nie skopiować) nazwany obiekt (l-wartość)

```
std::vector<std::thread> workers;
for (int i = 0; i < N; i++)
{
    std::thread th ([i]()
    {
        std::cout << "Witaj, świecie nr " << i << "\n";
    });
    workers.push_back(std::move(th));
}
```



**std::move** „unieważnia” przenoszony obiekt (tu: **th**),  
dlatego jego destruktor nie wywoła **terminate**



- Zawsze można sprawdzić stan wątku

```
int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        std::thread th ([i]()
        {
            std::cout << "Witaj, świecie nr " << i << "\n";
        });
        workers.push_back(std::move(th));
        assert (!th.joinable());
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
    {
        assert (w.joinable());
        w.join();
    }
}
```

Przekazywanie parametrów  
do wątków

# Przez wartość...

```
constexpr int N = 10;

void th_fun(int i)
{
    std::cout << "Witaj, świecie nr " << i << "\n";
}

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        std::thread th (&th_fun, i);
        workers.push_back(std::move(th));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

# Przez referencję: `std::ref`

```
constexpr int N = 10;

void th_fun(int & i)
{
    std::cout << "Witaj, świecie nr " << i << "\n";
}

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        std::thread th (&th_fun, std::ref(i));
        workers.push_back(std::move(th));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

# Dygresja: jak to działa?

- Konstruktor `std::thread` zaimplementowano jako tzw. ***variadic template***: szablon o dowolnej liczbie parametrów (od C++11)

```
template <class Fn, class... Args>  
explicit thread (Fn&& fn, Args&&... args);
```

# Wyniki... (wersja z referencją)

```
Witaj, świecie nr Witaj, świecie nr Witaj, świecie nr 24
Witaj, świecie nr 5
1
Witaj, świecie nr 3
Witaj, świecie nr 7
Witaj, świecie nr 7
Witaj, świecie nr 9
Hello, world!
Witaj, świecie nr 10
Witaj, świecie nr 10
```

**Dwa światy nr 10  
(numer nieistniejący)**

**Dwa światy nr 7**

**Brak światów nr 0, 6, 8**

# Przez referencję:

```
void th_fun(int & i)
{
    std::cout << "Witaj, świecie nr " << i << "\n";
}

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        std::thread th (&th_fun, std::ref(i));
        workers.push_back(std::move(th));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

**Tu ginie zmienna i**



**Wątki są  
niszczone później**





# Przez referencję:

Tu ginie zmienna i

Wątki są  
niszczone później

```
void th_fun(int & i)
{
    std::cout << "Witaj, świecie nr " << i << "\n";
}

int main()
{
    std::vector<std::thread> workers;
    for (int i = 0; i < N; i++)
    {
        std::thread th (&th_fun, std::ref(i));
        workers.push_back(std::move(th));
    }
    std::cout << "Hello, world!\n";
    for (auto & w: workers)
        w.join();
}
```

10 wątkom przekazano referencję do tej samej zmiennej,  
która w dodatku jest niszczone szybciej niż te wątki  
⇒ **race condition + wątki mogą operować na śmieciach**




# Przykład 2

```
constexpr int N = 10;
constexpr int M = 10000;

void th_fun(std::map<int,int> &map, std::vector<int> const& v,
            int start, int len)
{
    for (int i = start; i < len + start; ++i)
        map[v[i]]++;
}

int main()
{
    srand(0);
    std::vector<int> v(M);
    std::generate(v.begin(), v.end(), [](){return rand() % N; });
    std::map<int,int> mapa;
    std::cout << "worker thread:\n";
    std::thread th (&th_fun, std::ref(mapa), std::cref(v), 0, M/2);
    std::cout << "main thread:\n";
    th_fun (mapa, v, M/2, M/2);
    th.join();
    for (auto x: mapa)
        std::cout << x.second << " ";
    std::cout << "\n";
}
```



# Przykład 2

```
void th_fun(std::map<int,int> & map, std::vector<int> const& v,  
           int start, int len)  
{  
    for (int i = start; i < len + start; ++i)  
        map[v[i]]++;  
}
```

- Program w 2 wątkach wyznacza liczbę wystąpień danej liczby w wektorze v
- Argument map przekazywany jest przez referencję
- Wyścig!

# Przykład 2

```
void th_fun(std::map<int,int> & map, std::vector<int> const& v,  
           int start, int len)  
{  
    for (int i = start; i < len + start; ++i)  
        map[v[i]]++;  
}
```



```
for (auto x: mapa)  
    std::cout << x.second << " ";  
std::cout << "\n";
```



- Wyścig!

1017 951 979 932 973 924 991 1006 962 951

1020 950 965 933 963 914 996 996 958 953

Naruszenie ochrony pamięci (zrzut pamięci)

# Dygresja: `std::ref` i `std::cref`

- Są to „wrappery” dla argumentów szablonów funkcji

```
template <typename T>
void foo(T x);
...
int x;
foo(std::ref(x));
```



**T** jest zamieniane  
z **int** na **int&**

```
template <typename T>
void foo(T x);
...
int x;
foo(std::cref(x));
```



**T** jest zamieniane  
z **int** na **const int&**

**static\_cast<...>(...)**

# Dlaczego „&” jest niebezpieczna?

- Przekazując dane przez **referencję** lub **wskaźnik**, tworzymy sytuację, gdy wątek może operować *bezpośrednio* na (lokalnych) zmiennych cudzego wątku
- Zapanowanie nad tym problemem to nawet większe wyzwanie niż bezpieczna obsługa współdzielonych zmiennych globalnych

# const& ?

- Stała referencja jest tylko protezą
- Bo skąd wiemy, czy wątek główny nie modyfikuje danych, przekazanych innym wątkom przez stałą referencję?

# Co zamiast &?

- **Kopiowanie danych** do wątków pobocznych
  - To może być bardzo kosztowne, jeśli danych jest dużo!
- **Przesuwanie danych** do wątków pobocznych

**W świecie idealnym każdy wątek pracuje na własnych, unikatowych danych**



**unikamy wyścigu**