



Uniwersytet
Wrocławski

Programowanie współbieżne w C++11

Zbigniew Koza

Wydział Fizyki i Astronomii

„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`

joinable?

- 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();
}
```

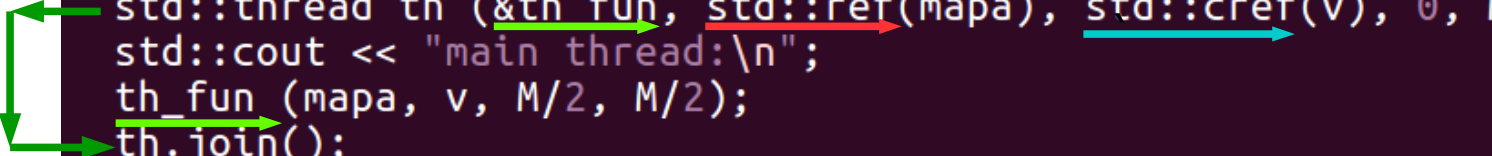
10 wątkom przekazano referencję do tej samej zmiennej,
która w dodatku jest niszczona szybciej niż te wątki
⇒ **race condition + wątki mogą operować na śmieciach**

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 szablono**w** 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**

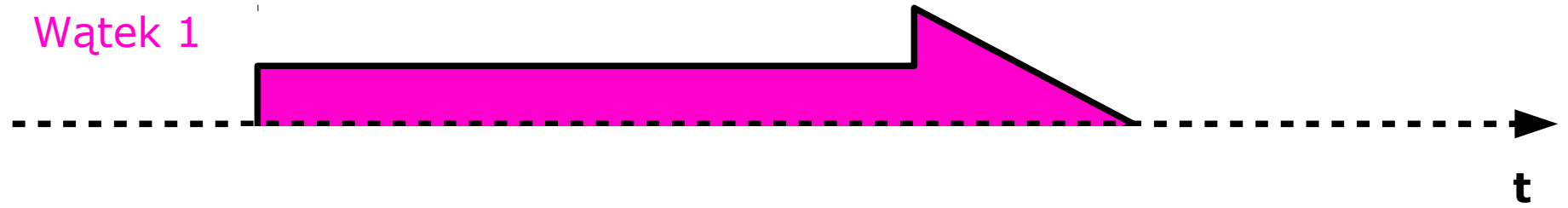
Jak przekazać dane  
z wątku roboczego?

promise and future

# Kanał komunikacyjny

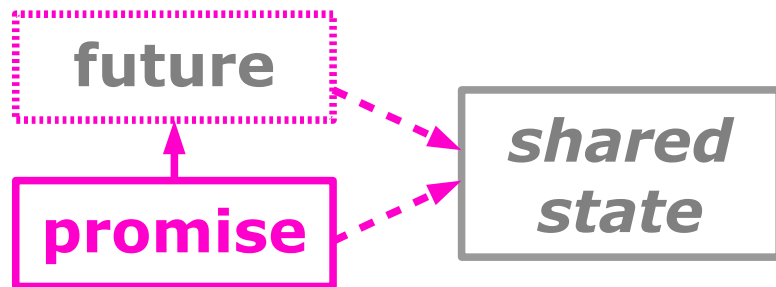
promise

Wątek 1

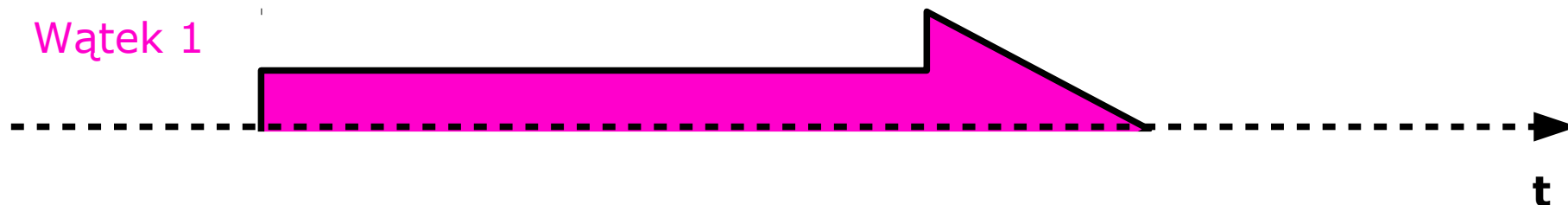




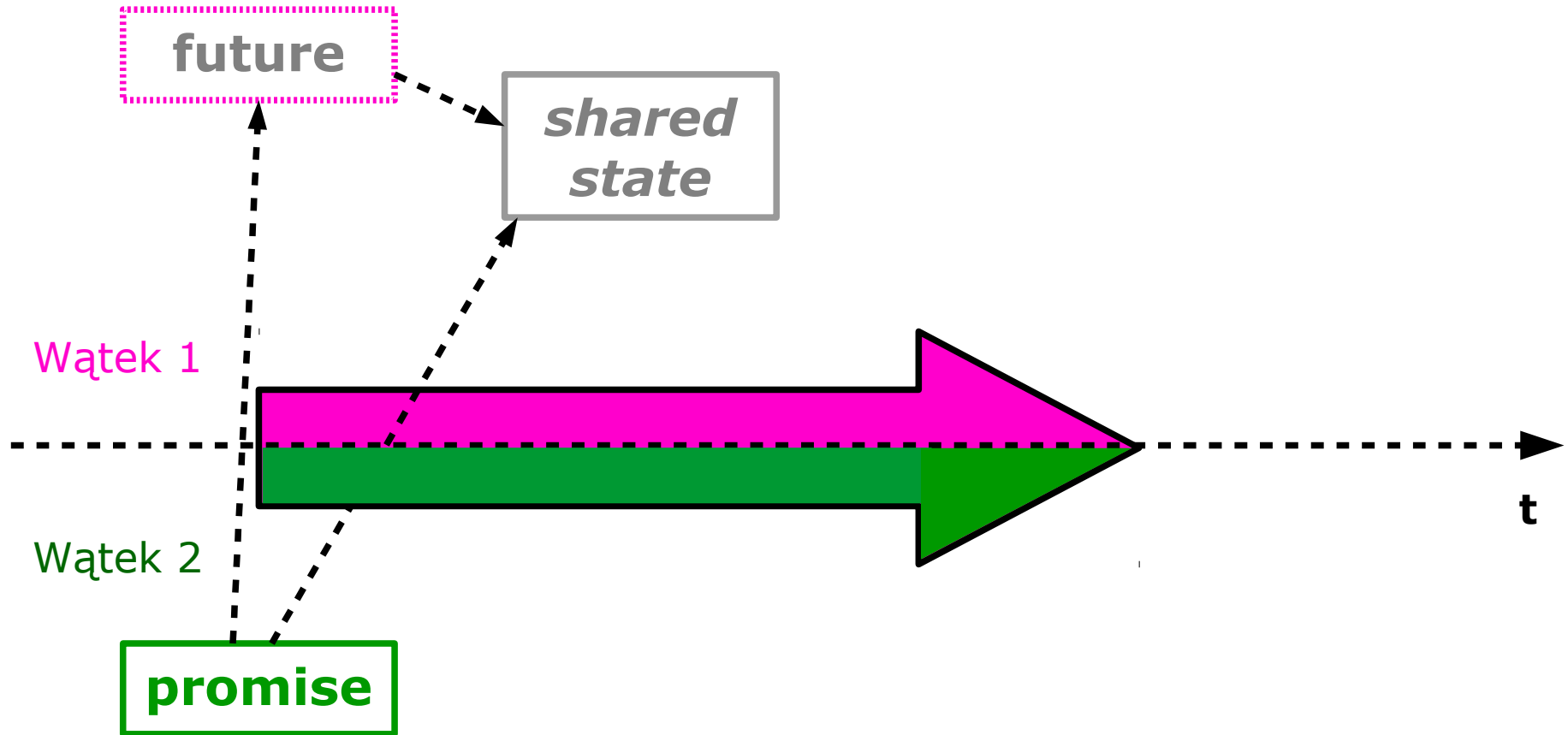
# Kanał komunikacyjny



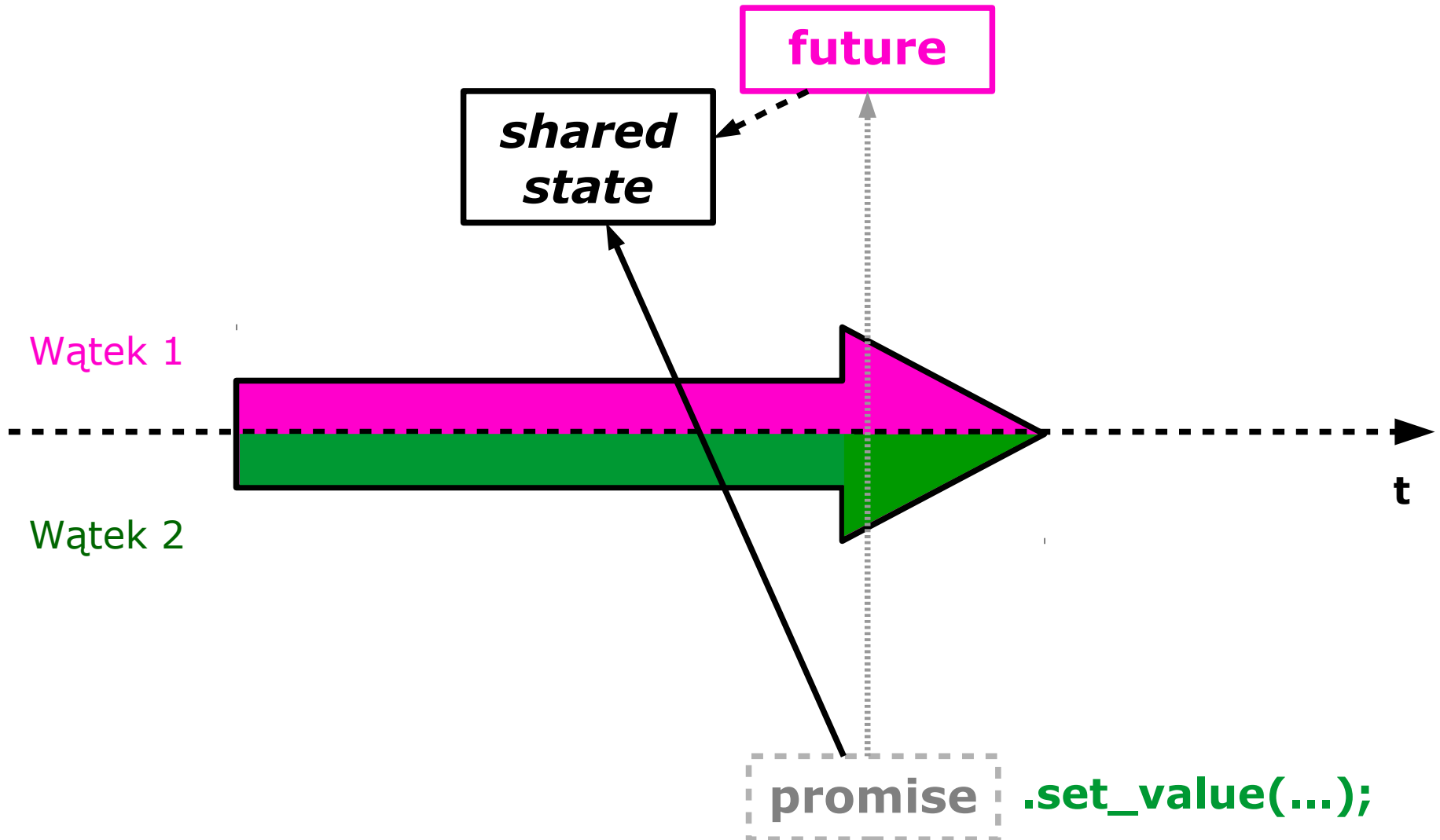
Wątek 1



# Kanał komunikacyjny



# Kanał komunikacyjny



# Kanał komunikacyjny

```
auto val = future.get();
```

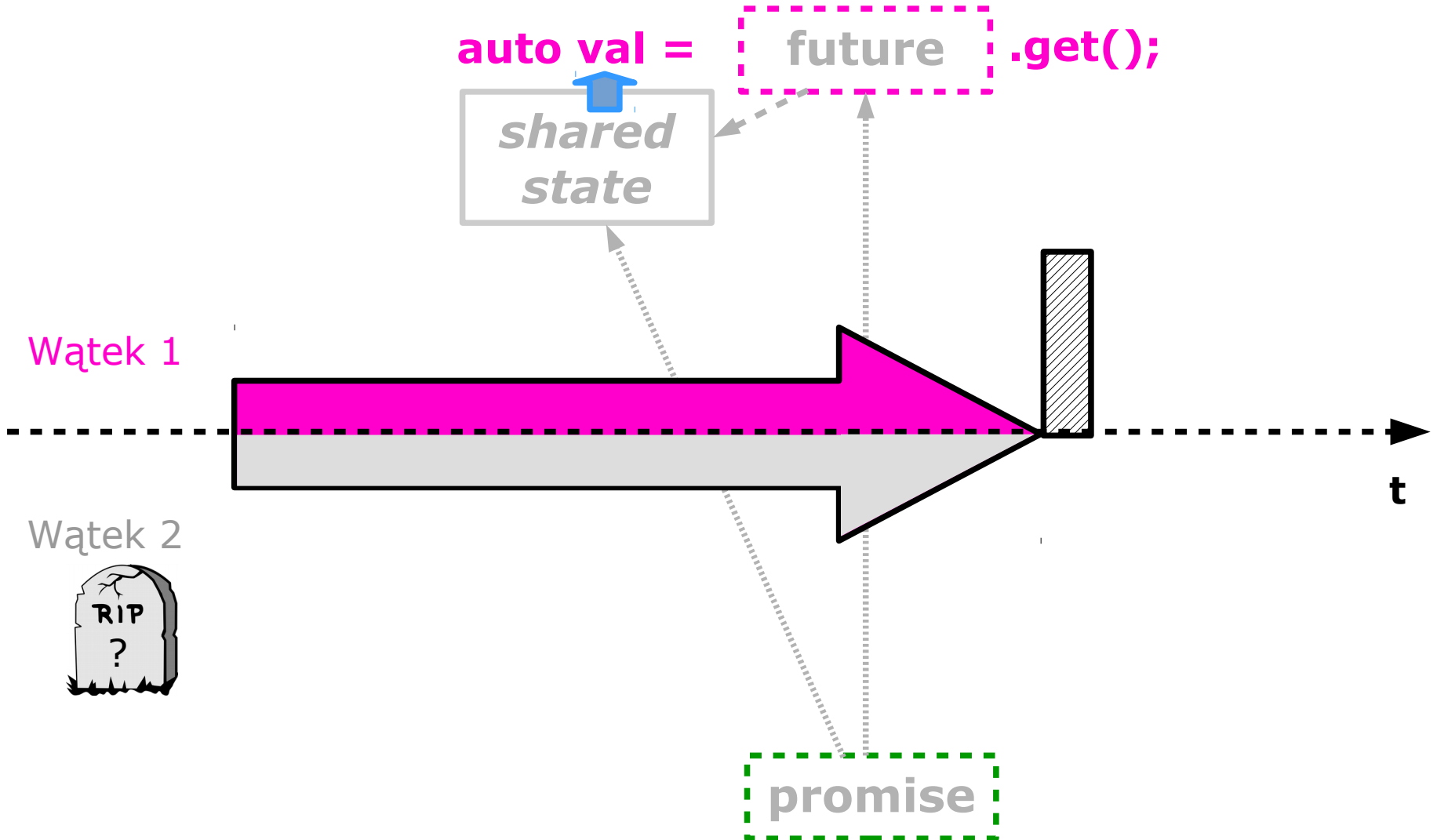
*shared  
state*

Wątek 1

Wątek 2



t



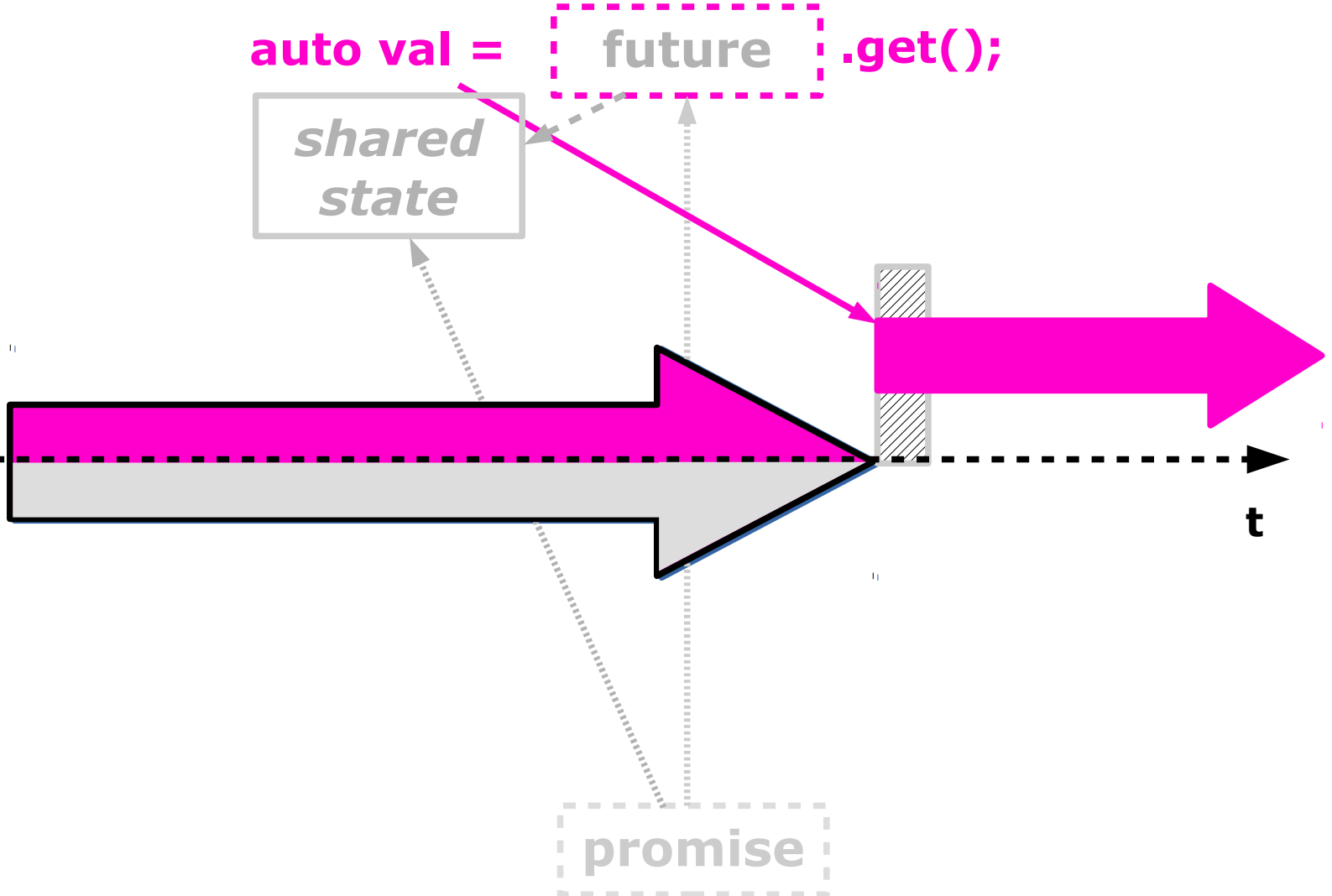
# Kanał komunikacyjny

```
auto val = future.get();
```

*shared  
state*

Wątek 1

Wątek 2



# Przykład

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

void my_fun(std::promise<std::string> & prms)
{
 prms.set_value(std::string("Ja bez żadnego trybu...\n"));
}

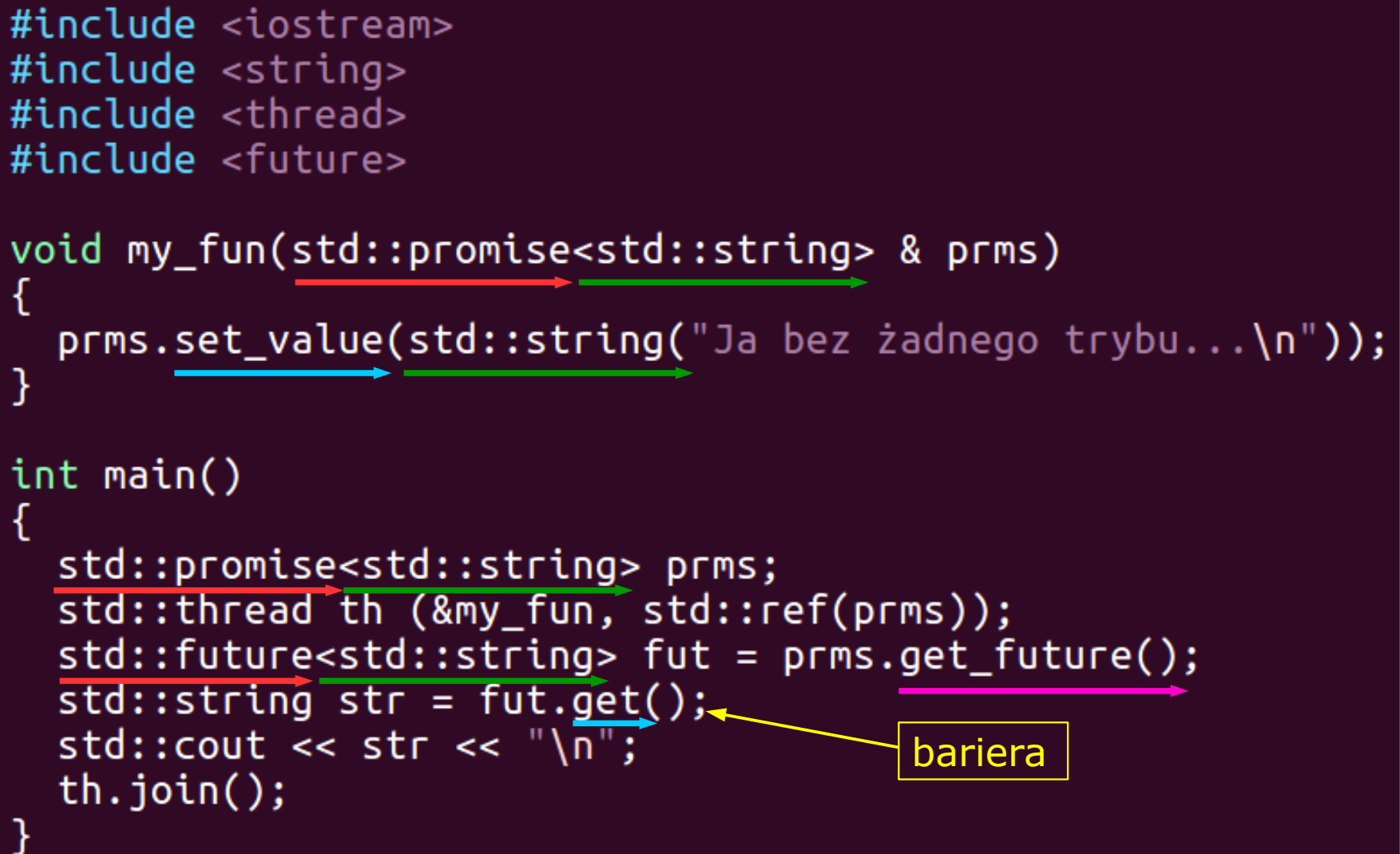
int main()
{
 std::promise<std::string> prms;
 std::thread th (&my_fun, std::ref(prms));
 std::future<std::string> fut = prms.get_future();
 std::string str = fut.get();
 std::cout << str << "\n";
 th.join();
}
```

  
**promise/future** → **shared state** ← <sup>1-1</sup> **setter/getter**

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

void my_fun(std::promise<std::string> & prms)
{
 prms.set_value(std::string("Ja bez żadnego trybu...\n"));
}

int main()
{
 std::promise<std::string> prms;
 std::thread th (&my_fun, std::ref(prms));
 std::future<std::string> fut = prms.get_future();
 std::string str = fut.get();
 std::cout << str << "\n";
 th.join();
}
```



bariera

# promise & future są „jednorazowego użytku”

```
void my_fun(std::promise<std::string> prms)
{
 prms.set_value(std::string("Ja bez żadnego trybu...\n"));
 prms.set_value(std::string("Ja znów bez żadnego trybu...\n"));
}
```



Ja bez żadnego trybu...

```
terminate called after throwing an instance of 'std::future_error'
 what(): std::future_error: Promise already satisfied
Przerwane (zrzut pamięci)
```



# Obsługa wyjątków

```
try{
 std::string str("Witajcie w przyszłości!");
 prms.set_value(str);
 prms.set_value(str); ← throw std::future_error
}
catch(std::exception &)
{
 std::cout << "Huston, mamy problem!\n";
}
```

Wątek główny, main()

Wątek poboczny, catch()

Witajcie w przyszłości!  
Huston, mamy problem!

- Obsługa wyjątków - standardowa, o ile nie musimy ich przekazać do wątku głównego...

# Wyjątek przed `promise::set_value()`

```
void my_fun(std::promise<std::string> & prms)
{
 try{
 std::string str("Witajcie w przyszłości!");
 ! → throw std::runtime_error("robotnik zgłasza wyjątek");
 prms.set_value(str);
 }
 catch(std::exception &) {
 prms.set_exception(std::current_exception());
 }
}
```

Nigdy się nie wywoła

- Należy wyłapać
- Ustawić w obiekcie `std::promise` (funkcją `set_exception`)
  - Używaj `std::current_exception()`

# Wyjątek przed promise::set\_value()

```
try{
 std::promise<std::string> prms;
 std::future<std::string> fut = prms.get_future();
 th = std::thread (&my_fun, std::ref(prms));
 std::string str = fut.get();
 std::cout << str << "\n";
}
catch(std::exception & e)
{
 std::cout << "wyjątek: " << e.what() << "\n";
}
```

```
void my_fun(std::promise<std::string> & prms)
{
 try{
 std::string str("Witajcie w przyszłości!");
 throw std::runtime_error("robotnik zgłasza");
 prms.set_value(str);
 }
 catch(std::exception & e)
 {
 prms.set_exception(std::current_exception());
 }
}
```

wyjątek: robotnik zgłasza wyjątek

- Po ustawieniu wyjątku w „shared state” obiekt `std::future` przerywa blokadę i zgłasza ten wyjątek w swoim wątku

# Wyjątek ~~po~~ promise::set\_value()

```
void my_fun(std::promise<std::string> prms)
{
 try{
 std::string str("Witajcie w przyszłości!");
 prms.set_value(str);
 throw std::runtime_error("robotnik zgłasza wyjątek");
 }
 catch(std::exception &)
 {
 prms.set_exception(std::current_exception());
 }
}
```



Witajcie w przyszłości!  
terminate called after throwing an instance of 'std::future\_error'  
what(): std::future\_error: Promise already satisfied  
Przerwane (zrzut pamięci)

- **std::promise** może zapisać albo wartość, albo wyjątek, ale nie oba naraz
- Bo **std::future** już mógł zwolnić barierę...