



Uniwersytet
Wrocławski

Programowanie współbieżne w C++11: *promise & future*

Zbigniew Koza

Wydział Fizyki i Astronomii

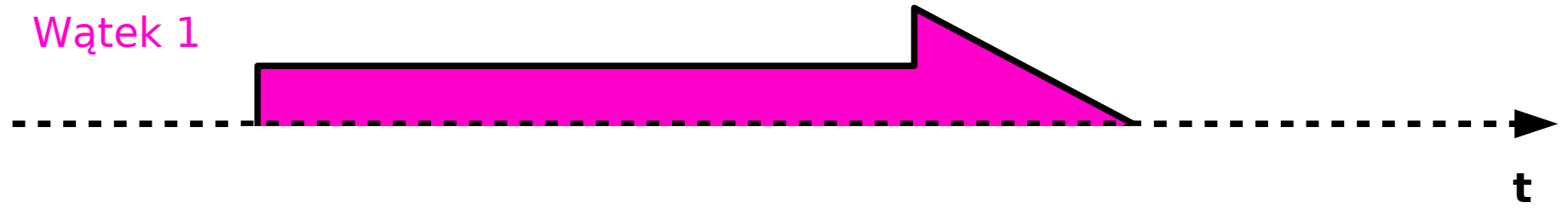
promise and future

Jak przekazać dane
z wątku roboczego?

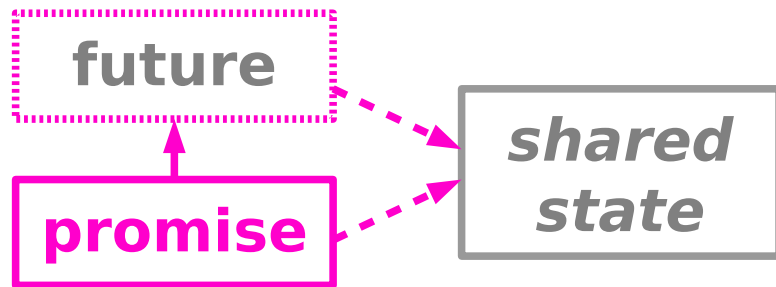
Kanał komunikacyjny

promise

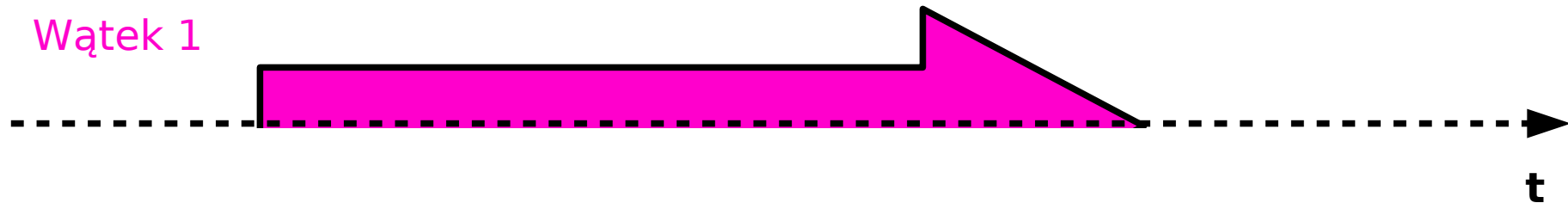
Wątek 1



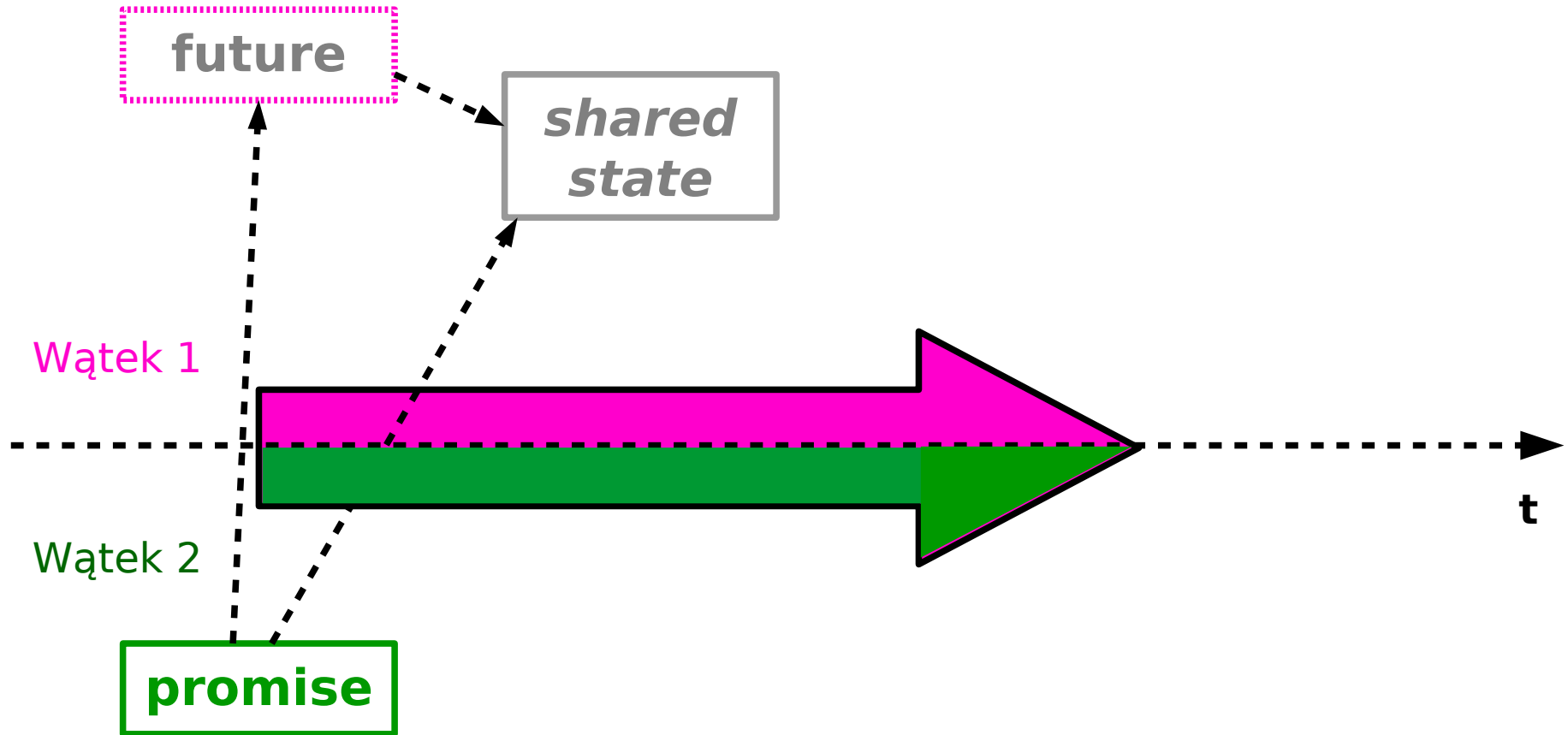
Kanał komunikacyjny



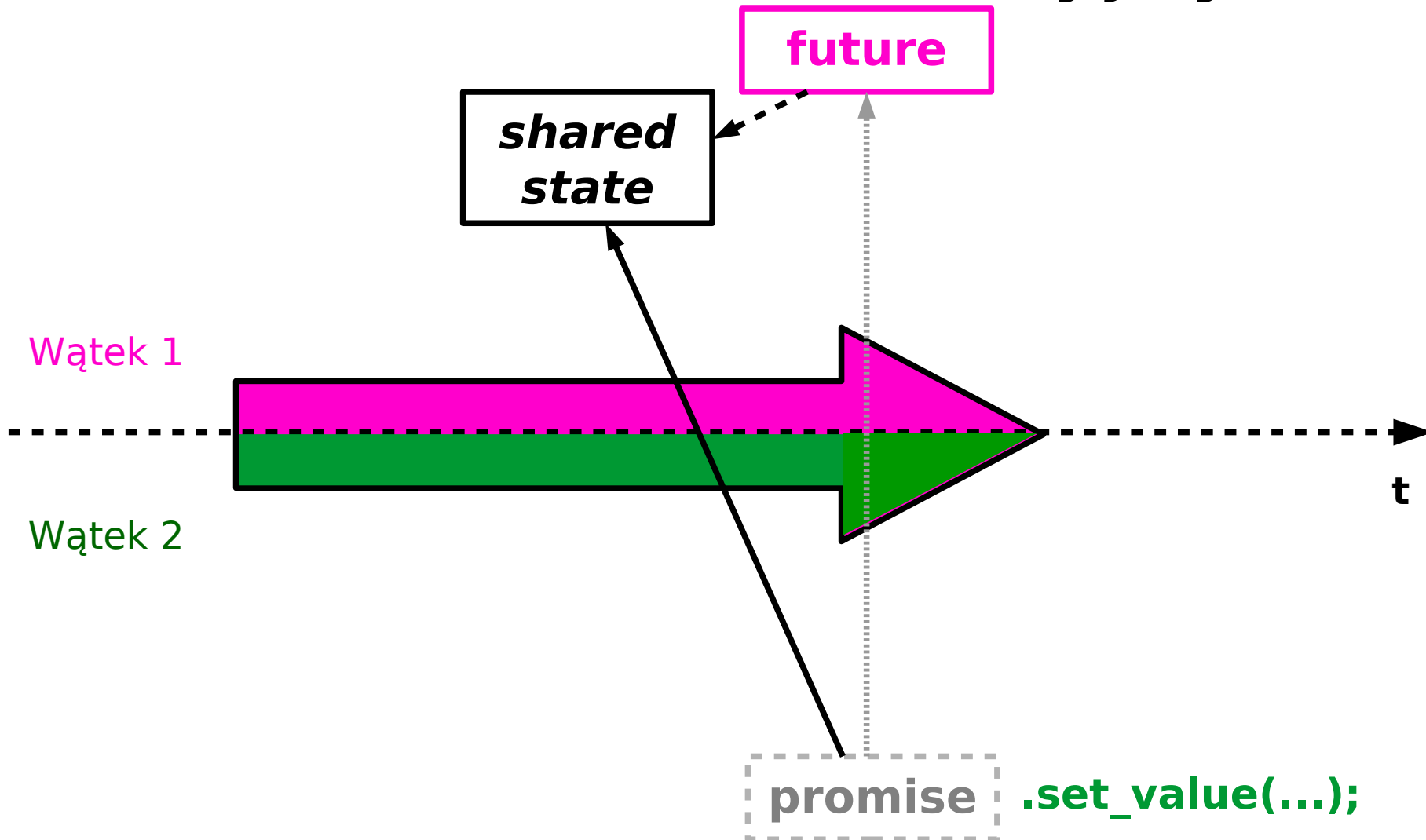
Wątek 1



Kanał komunikacyjny



Kanał komunikacyjny



Kanał komunikacyjny

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

*shared
state*

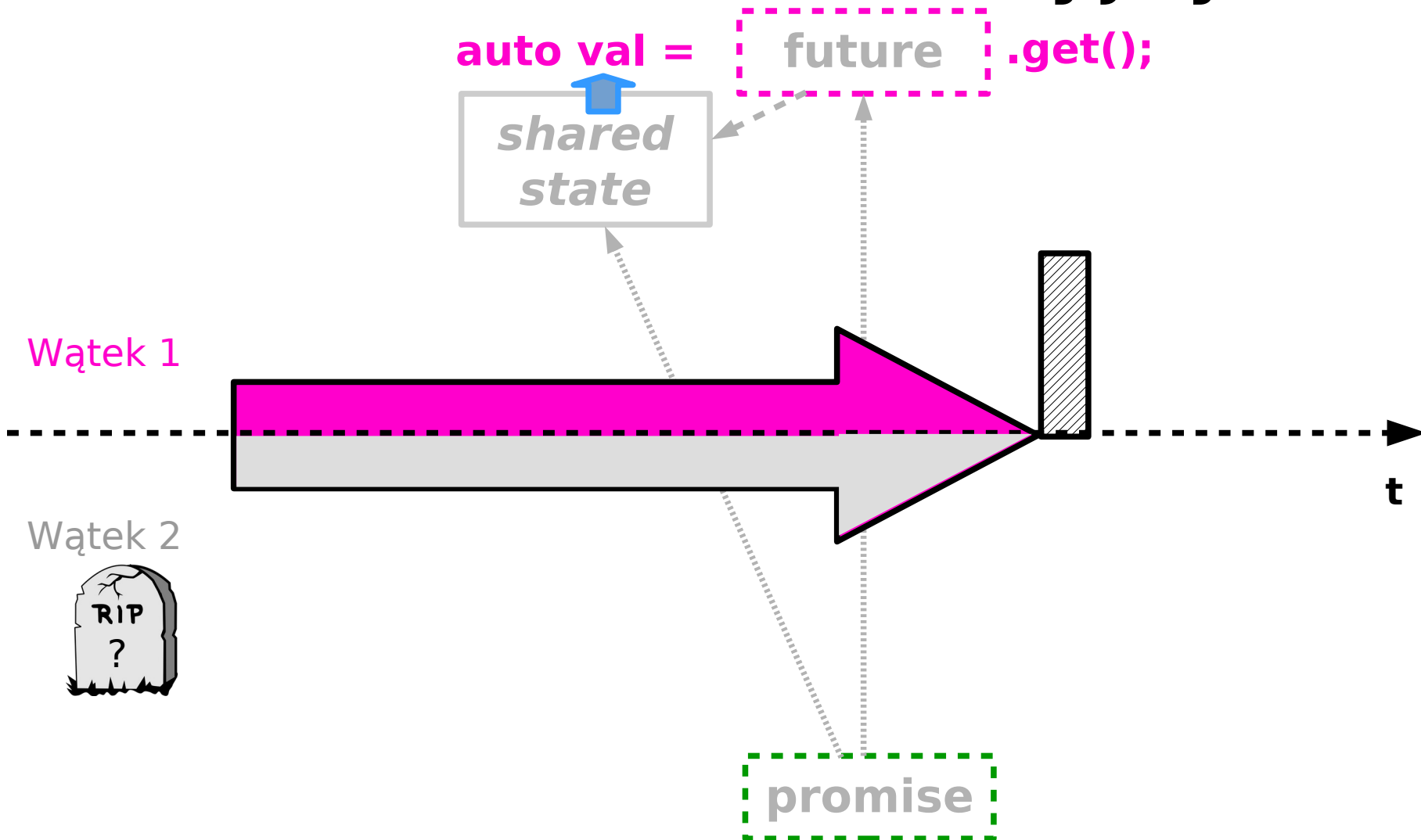
Wątek 1

Wątek 2



t

promise



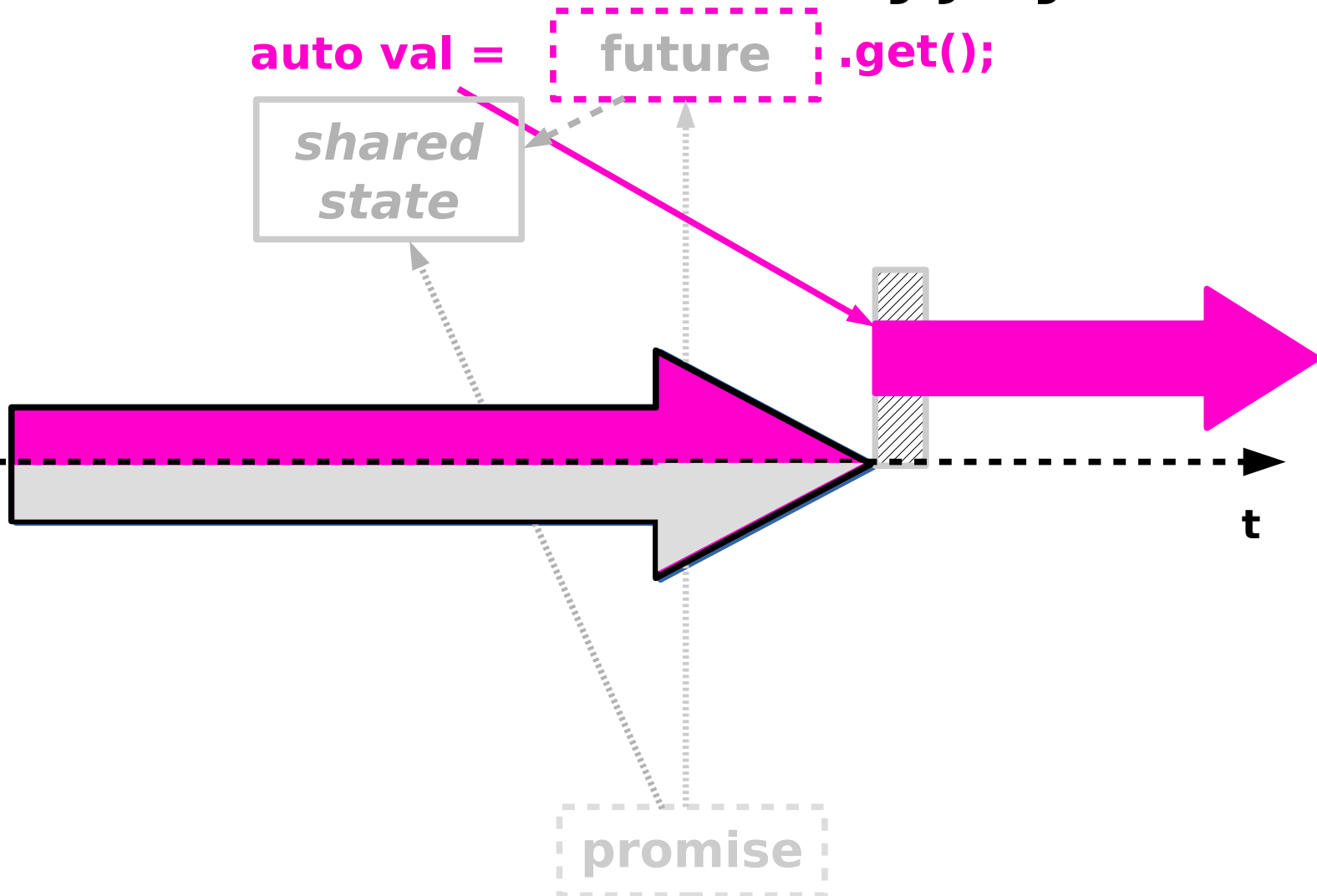
Kanał komunikacyjny

```
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()
{
1 ➡ std::promise<std::string> prms;
2 ➡ std::future<std::string> fut = prms.get_future();
3 ➡ std::thread th (&my_fun, std::move(prms));
4 ➡ std::string str = fut.get();
    std::cout << str << "\n";
    th.join();
}
```

 **promise/future**  **shared state**  **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);  
}  
catch(std::exception &)  
{  
    std::cout << "Huston, mamy problem!\n";  
}
```

throw std::future_error

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...

`std::async`


std::async

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

double my_fun(int n)
{
    double s = 0.0;
    for (int i = 1; i <= n; i += 2)
        s += 1.0/i - 1.0/(i+1);
    return s;
}

int main()
{
    constexpr int n = 1000000000;
    auto fut = std::async(my_fun, n);
    std::cout << "Czekam na wynik...\n";
    auto value = fut.get();
    std::cout << value << "\n";
}
```

Zwykła funkcja C++



- **std::async**
to wrapper dla
std::thread
i **std::promise**
- Zwraca
std::future
- **future::get**
zwraca wartość
zwróconą w funkcji
wywołanej
asynchronicznie

Cechy `std::async`

- Zalety:
 - Prostota
- Wady:
 - Liczne niespodzianki w dokumentacji i dostępnych implementacjach
 - Łatwość napisania kodu, który zachowuje się wbrew zdroworozsądkowym oczekiwaniom...

std::launch::async/ deferred

```
void print_ten (char c, int ms) {
    for (int i=0; i<10; ++i) {
        std::this_thread::sleep_for (std::chrono::milliseconds(ms));
        std::cout << c;
    }
}

int main ()
{
    std::cout << "with launch::async:\n";
    std::future<void> foo = std::async (std::launch::async, print_ten, '*', 100);
    std::future<void> bar = std::async (std::launch::async, print_ten, '@', 200);
    // async "get" (wait for foo and bar to be ready):
    foo.get();
    bar.get();
    std::cout << "\n\n";

    std::cout << "with launch::deferred:\n";
    foo = std::async (std::launch::deferred, print_ten, '*', 100);
    bar = std::async (std::launch::deferred, print_ten, '@', 200);
    // deferred "get" (perform the actual calls):
    foo.get();
    bar.get();
    std::cout << '\n';

    return 0;
}
```

std::launch::async/ deferred

```
void print_ten (char c, int ms) {
    for (int i=0; i<10; ++i) {
        std::this_thread::sleep_for (std::chrono::milliseconds(ms));
        std::cout << c;
    }
}

int main ()
{
    std::cout << "with launch::async:\n";
    std::future<void> foo = std::async (std::launch::async, print_ten, '*', 100);
    std::future<void> bar = std::async (std::launch::async, print_ten, '@', 200);
    // async "get" (wait for foo and bar to be ready):
    foo.get();
    bar.get();
    std::cout << "\n\n";

    std::cout << "with launch::deferred:\n";
    foo = std::async (std::launch::deferred, print_ten, '*', 100);
    bar = std::async (std::launch::deferred, print_ten, '@', 200);
    // deferred "get" (perform the actual calls):
    foo.get();
    bar.get();
    std::cout << '\n';

    return 0;
}
```

Bariery

Tu faktyczne uruchomienie

synchronicznie (szeregowo)

Possible output:

```
with launch::async:
**@**@**@*@**@*@@@@@

with launch::deferred:
*****@@@@@@@@@@@@@
```

std::launch:async/ deferred

- **std::launch::async** – natychmiastowe uruchomienie funkcji w osobnym wątku
- **std::launch::deferred** – uruchomienie funkcji przy pierwszym wywołaniu `wait` lub `get` na obiekcie `std::future`
- Brak tego parametru – sposób wywołania funkcji zależy od implementacji :-(
 - Przykład: Ubuntu 16.04 64 bit:
deferred is preferred
 - Manjaro 21.2.0: *async is preferred*

`std::async` „bez wartości”

```
std::cout << "with launch::async and temporaries:\n";  
std::async (std::launch::async, print_ten, '*', 100);  
std::async (std::launch::async, print_ten, '@', 200);  
std::cout << "\n";
```

- Powyższe dwa wywołania **`std::async`** spowodują:
 - Asynchroniczne uruchomienie funkcji **`print_ten`**
 - I natychmiastową blokadę na destruktorze *tymczasowego* **`std::future`** do zakończenia **`print_ten`**
- Efektywnie będą więc synchroniczne

`std::shared_future`

- Jest to „future” wielokrotnego dostępu
- Ale po co, skoro promise może stan shared memory ustalić tylko raz?
- Przydatne, jeśli do tych samych danych (zapisanych przez jakiś promise) należy dać dostęp kilku wątkom

Wątek roboczy z **shared_future**

```
void th_fun(std::shared_future<void> sfut, int id)
{
    std::ofstream F("parallel.txt", std::ios::app);
    while(sfut.wait_for(std::chrono::seconds(0)) != std::future_status::ready)
    {
        for (volatile long int i = 0; i < id*1'000'000'000L+ 390'005'000; i++)
            continue;
        F << std::to_string(id) + "\n" << std::flush;
    };
    std::cout << "bye from " + std::to_string(id) + "\n";
}
```

- W powyższym kodzie wątek roboczy w pętli „coś robi”, od czasu do czasu monitorując stan swojego **shared_future**

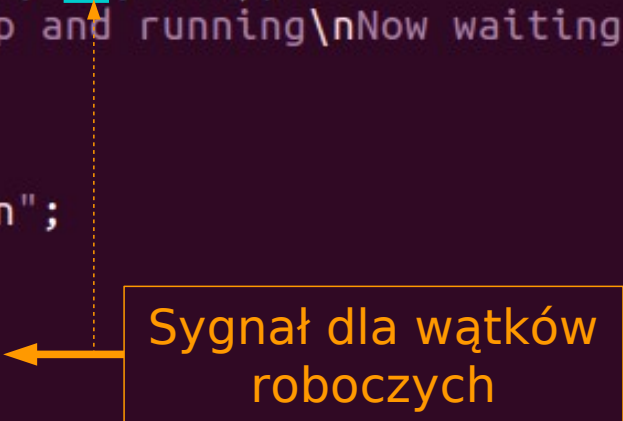
Dygresja

```
void th_fun(std::shared_future<void> sfut, int id)
{
    std::ofstream F("parallel.txt", std::ios::app);
    while(sfut.wait_for(std::chrono::seconds(0)) != std::future_status::ready)
    {
        for (volatile long int i = 0; i < id*1'000'000'000L+ 390'005'000; i++)
            continue;
        F << std::to_string(id) + "\n" << std::flush;
    };
    std::cout << "bye from " + std::to_string(id) + "\n";
}
```

- **volatile** – zakaz optymalizacji operacji na obiektach z tym modyfikatorem
- 1'000'000 – lukier składniowy C++14
- 10000000000L – typ literału (tu: **long** int)

Wątek główny z `shared_future`

```
int main()
{
    constexpr int N = 4;
    std::promise<void> prms;
    std::shared_future<void> sf = prms.get_future().share();
    std::vector<std::thread> v;
    for (int i = 0; i < 4; i++)
        v.emplace_back(th_fun, sf, i+1);
    std::cout << "all is up and running\nNow waiting for q...\n";
    char c;
    while (std::cin >> c)
    {
        std::cout << c << "\n";
        if (c == 'q')
        {
            prms.set_value();
            break;
        }
    }
    for (auto & th: v)
        th.join();
}
```



Sygnal dla wątków roboczych

The diagram consists of a yellow dashed arrow pointing from the `prms.set_value();` line in the code to a yellow box containing the text "Sygnal dla wątków roboczych".

Promise-future

- W poprzednim przykładzie:
 - Kierunek przepływu informacji był **odwrotny**:
od wątku głównego do wątków roboczych
 - Tę informację wątek główny przekazał **z opóźnieniem** (czego nie da się zrobić np. poprzez argumenty funkcji)

Dygresja

```
v.emplace_back(th_fun, sf, i+1);
```

- Składowe kontenerów z nazwą **emplace** np. **emplace_back**, **emplace**, etc. służą do jednoczesnej konstrukcji i dodania obiektu do kontenera (bez osobnej konstrukcji i kopiowania!)
- Charakterystyczne dla kontenerów STL w C++11 (vector, map, set,...)