

```
thread_local int i = 0;
void bar(int id){
    i += id;
}
void foo(int id) {
    bar(id);
    std::cout << std::addressof(i) << "\n";
    std::cout << i << "\n";
}
int main() {
    i = 42;
    std::thread t1(foo, 1);
    std::thread t2(foo, 2);

    t1.join(); t2.join();
    std::cout << std::addressof(i) << "\n";
    std::cout << i << "\n";
    return 0;
}
```

0x7fd4ec402b20

1

0x7fd4ec500000

2

0x7fd4ec400710

42

```

- struct chirp{
    int id = 0;
-   chirp(){
        std::cout << "created" << "\n";
    };
    thread_local chirp obj;
- void sometimes_called(){
    obj.id = 42;
}
- void flip(){
    if(std::rand()%2)
        sometimes_called();
}
- int main(){
    std::srand(std::time(nullptr));
    std::thread t1(flip);
    t1.join();
}

```

# Megszakítás kívülről

- C: `int pthread_cancel(pthread_t tid);`
- Java: `java.lang.Thread.interrupt()`
- C++-ban régebb nem volt lehetőség a szál megszakítására más szálból
- C++20
  - „P0660: Cooperatively Interruptible *Joining* Thread”
  - `std::jthread`, `std::stop_token`

## `std::jthread`

Defined in header `<thread>`

`class jthread;` (since C++20)

The class `jthread` represents a [single thread of execution](#). It has the same general behavior as `std::thread`, except that `jthread` automatically rejoins on destruction, and can be cancelled/stopped in certain situations.

# Példa

- A szál ellenőrzi ha érkezett megszakítási kérés

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

using namespace std::literals::chrono_literals;

void f(std::stop_token stop_token, int value)
{
    while (!stop_token.stop_requested())
    {
        std::cout << value++ << ' ' << std::flush;
        std::this_thread::sleep_for(200ms);
    }
    std::cout << std::endl;
}

int main()
{
    std::jthread thread(f, 5); // prints 5 6 7 8... for approximately 3 seconds
    std::this_thread::sleep_for(3s);
    // The destructor of jthread calls request_stop() and join().
}
```

Possible output:

```
5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

# std::jthread

- Automatikus join, kooperatívan megszakítható

## std::jthread

Defined in header `<thread>`

```
class jthread;           (since C++20)
```

The class `jthread` represents a [single thread of execution](#). It has the same general behavior as `std::thread`, except that `jthread` automatically rejoins on destruction, and can be cancelled/stopped in certain situations.

Threads begin execution immediately upon construction of the associated thread object (pending any OS scheduling delays), starting at the top-level function provided as a [constructor argument](#). The return value of the top-level function is ignored and if it terminates by throwing an exception, `std::terminate` is called. The top-level function may communicate its return value or an exception to the caller via `std::promise` or by modifying shared variables (which may require synchronization, see `std::mutex` and `std::atomic`).

Unlike `std::thread`, the `jthread` logically holds an internal private member of type `std::stop_source`, which maintains a shared stop-state. The `jthread` constructor accepts a function that takes a `std::stop_token` as its first argument, which will be passed in by the `jthread` from its internal `std::stop_source`. This allows the function to check if stop has been requested during its execution, and return if it has.

`std::jthread` objects may also be in the state that does not represent any thread (after default construction, move from, detach, or join), and a thread of execution may be not associated with any `jthread` objects (after detach).

No two `std::jthread` objects may represent the same thread of execution; `std::jthread` is not [CopyConstructible](#) or [CopyAssignable](#), although it is [MoveConstructible](#) and [MoveAssignable](#).

```

int main()
{
    // A sleepy worker thread
    std::jthread sleepy_worker(
        [](std::stop_token token)
        {
            for (int i = 10; i; --i)
            {
                std::this_thread::sleep_for(300ms);
                if (token.stop_requested())
                {
                    std::cout << "Sleepy worker is requested to stop\n";
                    return;
                }
                std::cout << "Sleepy worker goes back to sleep\n";
            }
        });

    // A waiting worker thread
    // The condition variable will be awoken by the stop request.
    std::jthread waiting_worker(
        [](std::stop_token token)
        {
            std::mutex mutex;
            std::unique_lock lock(mutex);
            std::condition_variable_any().wait(lock, token, []{ return false; });
            std::cout << "Waiting worker is requested to stop\n";
            return;
        });

    // std::jthread::request_stop() can be called explicitly:
    std::cout << "Requesting stop of sleepy worker\n";
    sleepy_worker.request_stop();
    sleepy_worker.join();
    std::cout << "Sleepy worker joined\n";

    // Or automatically using RAII:
    // waiting_worker's destructor will call request_stop()
    // and join the thread automatically.
}

```

# “Joining” szál - RAII

```
class scoped_thread{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_): t(std::move(t_)){
        if ( !t.joinable())
            throw std::logic_error("No associated thread!");
    }
    ~scoped_thread(){
        t.join(); // waits for the thread to finish
    }
    scoped_thread(scoped_thread&)= delete;
    scoped_thread& operator=(scoped_thread const &)= delete;
};
```

# Szál leválasztás

- *detach()*
- Nincs lehetőség a szál visszacsatolására
- *joinable()* tulajdonság hamis lesz
- Fontos, hogy a szál ne hivatkozzon objektumokra melyeknek élettartama megszűnik!



# Példa

```
class MessagePrint
{
private:
    int &n; //n is changed to a reference
public:
    MessagePrint(int &n): n(_n){};
    void operator()(const std::string &msg) {
        for (auto i=0; i<this->n; i++){
            std::cout << msg << " " << i << "/" << n << '\n';
            std::this_thread::sleep_for(std::chrono::milliseconds(1));
        }
        std::cout << n << '\n';
    }
};
```

```

- void trouble(){
  int local_state = 1000000;
  std::string s{"Hello!"};
  MessagePrint mp(local_state);
  std::thread t = std::thread(mp, s);
  std::this_thread::sleep_for(std::chrono::seconds(1));
  t.detach();
- }// local variables go out of scope while the thread
- // might still be running

- int main() {
  trouble();
  std::cout << "Sleeping for 1 minute, please wait!\n";
  std::this_thread::sleep_for(std::chrono::seconds(60));
  return 0;
- }

```

```

...
Hello! 800/1000000
Hello! 801/1000000
Hello! 802/1000000
Hello! 803/1000000
Hello! 804/1000000
Sleeping for 1 minute, please wait!
Hello! 805/32766
Hello! 806/32766
Hello! 807/32766
...

```

clang



```

...
Hello! 796/1000000
Hello! 797/1000000
Sleeping for 1 minute, please wait!
0

```

gcc 8

# Szálgűjtemény

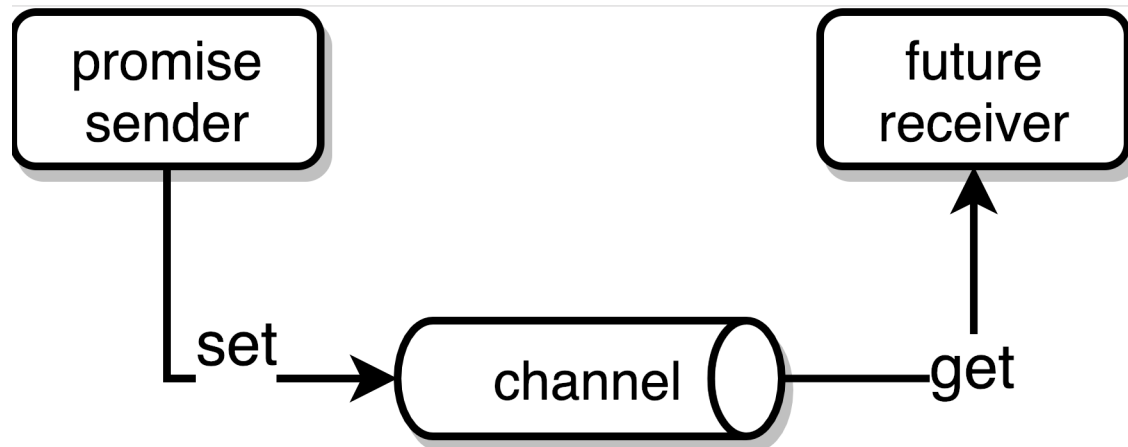
```
 int main() {  
    std::vector<std::thread> threads;  
    auto lambda = [](int id) {  
        std::cout << id << "\\n";  
    };  
    for (auto i = 0; i < 16; i++)  
        threads.emplace_back(lambda, i);  
    std::for_each(std::begin(threads),  
                  std::end(threads),  
                  std::mem_fn(&std::thread::join));  
    return 0;  
 }
```

# Korlátok

- Hibakezelés
  - Nehézkes, globális *std::exception\_ptr* keresztül
- Eredmények visszatérítése referenciákon keresztül
- Részeredmények lekérdezése
  - Nincs támogatás
- Megoldás -> taszk alapú programozás
  - *future, promise, packaged\_task, async*

# Taszk alapú programozás

- 1976 - future, promise, eventual, delay, deferred
- szétválasztja a (jövőbeli) értéket attól, hogy hogyan (és mikor) számítódik ki (ígéret)

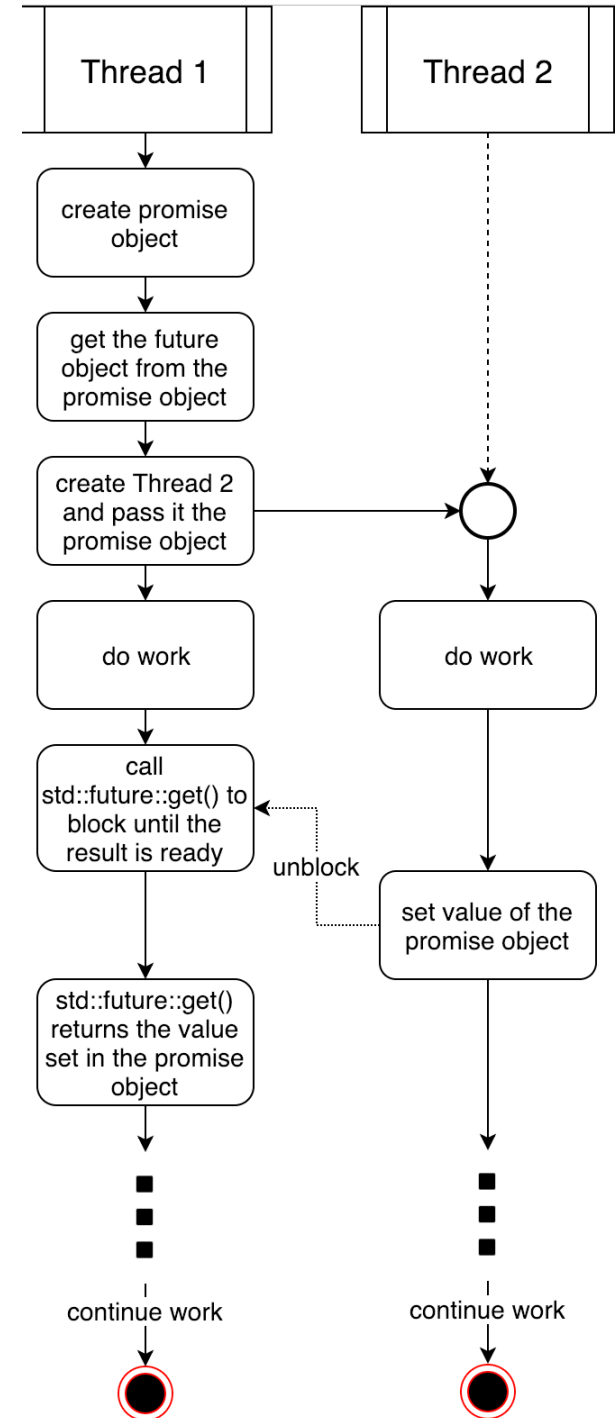


# C++-ban

- *<future>* fejléc
- *future* - osztálysablon, melynek objektuma egy jövőbeli értékét tárol
- *promise* - ígéret, hogy az értéket a jövőben, valamikor, valaki beállítja
  - Minden *promise* objektumnak van egy társított *future* objektuma, amellyel lekérhetjük az értéket, amelyre a *promise* objektum egyszer beállítódik.

# C++-ban

- Az ígért érték beállítása illetve lekérése
  - *set\_value()* és *get()*
- Időzített várakozás
  - *wait\_for*, *wait\_until*
- *future* objektumot a *get\_future()* metódus segítségével kapjuk meg az ígéretből
- Hibakezelés
  - *std::promise::set\_exception()*



# Példa

$$f(x, a) = \frac{1}{\sqrt{x} - a}$$

```
float f(float x, float a)
{
    if (x < 0)
        throw std::domain_error("Square root of negative number!");
    auto d = std::sqrt(x) - a;
    if (std::abs(d) < std::numeric_limits<float>::epsilon())
        throw std::domain_error("Divide by zero!");
    return 1/d;
}
```



```
void f(float x, float a, std::promise<float> &pr)
{
    if (x < 0) {
        pr.set_exception(
            std::make_exception_ptr(
                std::domain_error("Square root of negative number!")));
        return;
    }
    auto d = std::sqrt(x) - a;
    if (std::abs(d) < std::numeric_limits<float>::epsilon()) {
        pr.set_exception(
            std::make_exception_ptr(
                std::domain_error("Divide by zero!")));
        return;
    }
    // everything OK, set the result
    pr.set_value(1/d);
}
```

```
int main() {  
    auto pr = std::promise<float>{};  
    std::thread(f, 4, 1.99, std::ref(pr)).detach();  
    auto fv = pr.get_future(); // retrieve future obj.  
    try {  
        // do some other tasks  
        auto res = fv.get(); // blocks until ready or exception  
        std::cout << res << '\n';  
    }  
    catch (const std::exception& e) {  
        std::cout << "Exception: " << e.what() << '\n';  
    }  
    return 0;  
}
```

# packaged\_task

- Wrapper egy funkcionál köré
- Visszatéríti a funkcionál kiértékelési eredményét vagy kivételt mely
  - Egy *std::future* objektumon keresztül érhető el
- Gyakorlatban, elrejtí az *std::promise* részét az aszinkron számításnak!

# Példa

```
float f(float x, float a)
{
    if (x < 0)
        throw std::domain_error("Square root of negative number!");
    auto d = std::sqrt(x) - a;
    if (std::abs(d) < std::numeric_limits<float>::epsilon())
        throw std::domain_error("Divide by zero!");
    return 1/d;
}
```

```
int main() {  
    auto pt = std::packaged_task<decltype(f)>(f);  
    auto fv = pt.get_future();  
    std::thread(std::move(pt), 4, 1.99).detach();  
    try {  
        // do some other work  
        auto res = fv.get();  
        // blocks until ready or exception  
        std::cout << res << '\n';  
    }  
    catch (const std::exception& e) {  
        std::cout << "Exception: " << e.what() << '\n';  
    }  
    return 0;  
}
```

# async

- Ugyancsak wrapper egy funkcionál köré
- Még magasabb szintű absztrakció
- Futtatási (kiértékelési) lehetőségek (launch policy)
  - *launch::async* – a kód egy új programszámban fut (alapértelmezett)
  - *launch::deferred* – a kód az aktuális számban hajtódig végre aszinkron módon („lusta végrehajtás”)
  - *launch::async* / *launch::deferred* – implementáció függő
- Egyenesen egy *std::future* objektumot térít vissza

# Példa

```
int main() {  
    // auto pt = std::packaged_task<decltype(f)>(f);  
    // auto fv = pt.get_future();  
    // std::thread(std::move(pt), 4, 1.99).detach();  
    auto fv = std::async(std::launch::async, f, 4, 1.99);  
    try {  
        // do some other work  
        auto res = fv.get();  
        // blocks until ready or exception  
        std::cout << res << '\n';  
    }  
    catch (const std::exception& e) {  
        std::cout << "Exception: " << e.what() << '\n';  
    }  
    return 0;  
}
```

# Szál automatikus bevárása

- Ha külön szálon fut
  - az async által visszatérített objektum destruktora blokkol, bevárja (join) a szálat!

```
int main() {  
    {  
        auto fv = std::async(std::launch::async,  
            [](){std::this_thread::sleep_for(std::chrono::seconds(5));});  
    } // destructor of fv blocks here until spawned thread completes  
    std::cout << "Main thread continues with work\n";  
    // work performed by the main thread has to wait  
    return 0;  
}
```



# std::futures from std::async aren't special!

- Scott Meyers
  - <http://scottmeyers.blogspot.com/2013/03/stdfutures-from-stdasync-arent-special.html>

```
{  
    std::future<...> fv = fn_that_returns_a_future();  
} // does the destructor block or not?
```

# packaged\_task vs. async

- Az *async* külön szálon, azonnal megpróbálja lefuttatni a funkcionált
- *packaged\_task* - task előkészítése
  - manuális elindítás később
  - vagy explicit módon egy adott szálhoz rendelés

# Manuális indítás

```
int main()
{
    std::packaged_task<double(double)> task(
        [](double x) { return std::sqrt(x); });
    std::future<double> result = task.get_future();
    task(100.0); //meghívás
    std::cout << "Result: " << result.get() << << std::endl;
}
```

# Szálon

```
int main()
{
    std::packaged_task<double(double)> task(
        [](double x) { return std::sqrt(x); });
    std::future<double> result = task.get_future();

    std::thread task_td(std::move(task), 100.0);
    task_td.join();

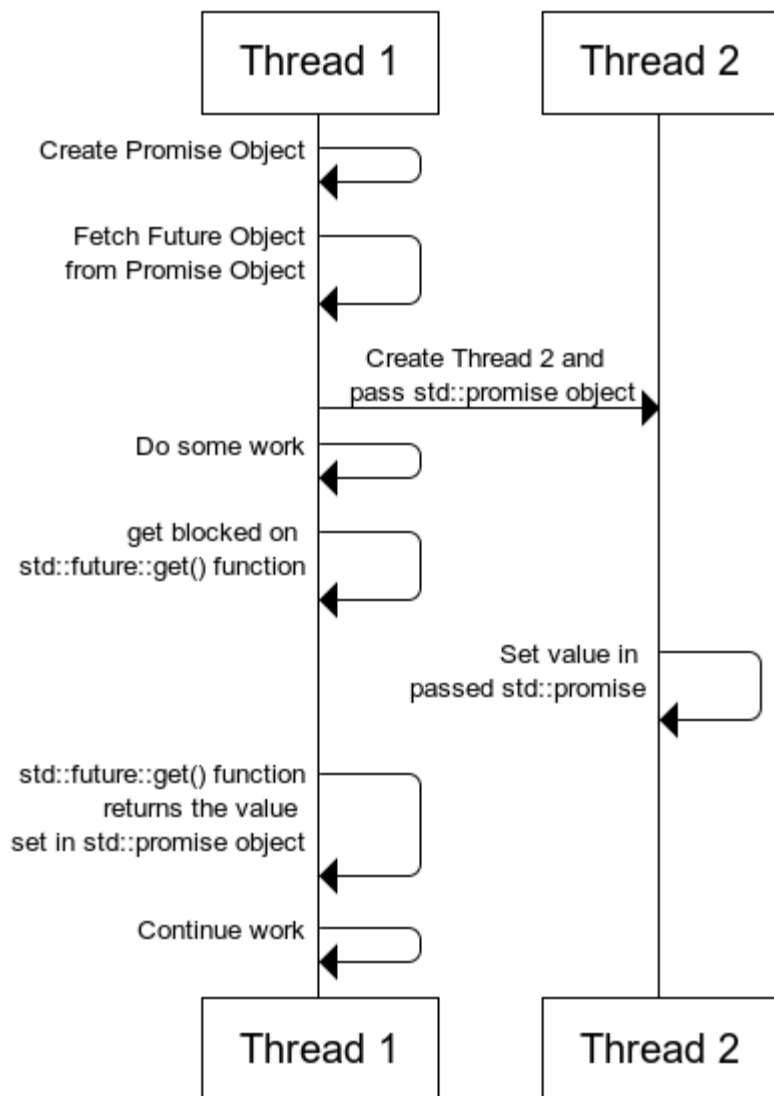
    std::cout << "Result: " << result.get() << << std::endl;
}
```

# Szálak - visszatérítési értékek

- *future* - osztálysablon, melynek objektuma egy jövőbeli értékét tárol
- *promise* - ígéret, hogy az értéket a jövőben, valamikor, valaki beállítja
  - Minden *promise* objektumnak van egy társított *future* objektuma, amellyel lekérhetjük az értéket, amelyre a *promise* objektum egyszer beállítódik.
- Az ígért érték beállítása illetve lekérése
  - *set\_value()* és *get()*
- *future* objektumot a *get\_future()* metódus segítségével kapjuk meg az ígéretből

# Szálak - visszatérítési értékek

## **std::promise and std::future work flow**



# Szálak - visszatérítési értékek

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

void initiazer(std::promise<int> * promObj)
{
    std::cout << "Inside Thread" << std::endl;
    promObj->set_value(42);
}

int main()
{
    std::promise<int> promiseObj;
    std::future<int> futureObj = promiseObj.get_future();
    std::thread th(initiazer, &promiseObj);
    std::cout << futureObj.get() << std::endl;
    th.join();
    return 0;
}
```