Threadler ve Exception Handling:

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

void foo()
{
    std::cout << "foo called!\n";
}

void func()
{
    std::cout << "func called!\n";
    throw std::runtime_error{ "runtime error exception from func\n" };
}

int main()
{
    std::thread t{ foo };

    func();

    t.join(); //will not be executed
}
```

→ Artık thread'in bir iş yürüttüğü için ya join'i ya da detach etmeliyiz.!

→ thread join olmayacak.

→ Thread destructor'ı çağırıldığında detach / terminate ! Call / thread need

↓ A better implementation

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

void foo()
{
    std::cout << "foo called!\n";
}

void func()
{
    std::cout << "func called!\n";
    throw std::runtime_error{ "runtime error exception from func\n" };
}

int main()
{
    std::thread t{ foo };

    try {
        func();
        t.join();
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
        t.join(); //will not be executed
    }
}
```

→ Callable içinde exception:

```cpp
void func()
{
    std::cout << "func cagrildi\n";
    if (1) {
        throw std::runtime_error{ "error from func\n" };
    }
}


int main()
{

    try {

        std::thread t{ func };
        t.join();
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << '\n';
    }
}
```

→ Callable'ın gönderdiği exception try bloğu içinde yakalanamaz ✗
Doğrudan terminate çağırılır. ✓
↳ terminate de abort'u çağırır.

→ Bunu düzgünce çalıştırabilmek için rethrow exception yapılabilir, Bunun için de thread'in gönderdiği exception'ı current exception olarak kaydedip, onu throw edilip, istenmesine göre rethrow etmeliyiz.

```cpp
#include <stdexcept>
#include <thread>

std::exception_ptr exptr = nullptr;

void func(int x)
{
    std::cout << "func(int x) cagrildi x = " << x << '\n';
    try {
        if (x % 2 == 0)
            throw std::invalid_argument{ "gecersiz arguman" };
    }
    catch (...) {
        exptr = std::current_exception();
    }
    std::cout << "func(int x) sona erdi x = " << x << '\n';
}


int main()
{
    std::thread t{ func, 10 };
    t.join();
    try {
        if (exptr) {
            std::rethrow_exception(exptr);
        }
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught : " << ex.what() << '\n';
    }
}
```

eğer exception throw edilmediyse nullptr

- Syncstream:
  - threadlerin interleave olmasının önüne geçiyor.
    - threadler sıralıdır
    - output'a yan yanalar
    - Artık threadlerin çıktılarının birleşmesi !

- Cpp 20 ile eklendi!

```cpp
using namespace std;


void func()
{
    using namespace std::literals;

    std::this_thread::sleep_for(2000ms);
    std::osyncstream os{ cout };

    for (int i = 0; i < 10000; ++i) {
        os << 13 << "alican" << 34.7564 << "mustafa\n";

    }
}

void foo()
{
    using namespace std::literals;
    std::this_thread::sleep_for(2000ms);

    for (int i = 0; i < 10000; ++i) {
        std::osyncstream{ std::cout } << 25 << "necati" << 98.87234 << "ergin\n";
    }
}

int main()
{
    std::thread t1{ func };
    std::thread t2{ foo };
```

*Threadlerin Containerda Saklanması:

```cpp
#include <iostream>
#include <syncstream>
#include <chrono>

using namespace std;


void cprint(char c)
{
    using namespace std::literals;

    for (int i = 0; i < 1000; ++i) {
        std::cout << c;
        std::this_thread::sleep_for(50ms);
    }
}

int main()
{
    std::thread ta[26];

    for (int i{};  auto & t : ta) {
        t = std::thread{ cprint, 'A' + i++ };
    }


    for (auto &t : ta) {
        t.join();
    }
}
```

array ile olduğu gibi vector ile de saklanabilir.

```cpp
#include <vector>

using namespace std;


void cprint(char c)
{
    using namespace std::literals;

    for (int i = 0; i < 1000; ++i) {
        std::cout << c;
        std::this_thread::sleep_for(50ms);
    }
}

int main()
{
    std::vector<std::thread>  tvec;


    for (int i = 0; i < 26; ++i) {
        tvec.push_back(std::thread{ cprint, i + 'A' });
    }

    for (auto& t : tvec) {
        t.join();
    }
}
```

*Yield:

```cpp
#include <iostream>
#include <thread>
#include <atomic>

std::atomic<bool> ga_start{ false };

void func(char id)
{
    using namespace std::chrono;
    while (!ga_start)
    {
        std::this_thread::yield();
    }
    std::this_thread::sleep_for(500ms);
    std::cout << id;
}

int main()
{
    std::thread ar_t[26];

    for (char i{ 'A' }; auto & t : ar_t)
        t = std::thread(func, i++);

    ga_start = true;

    for (auto& t : ar_t)
        t.join();
}
```

thread artık çizelge dışında

→ yield olduğu zaman, bu fonksiyon bir thread tarafından çalıştırıldığında, scheduler'ın çizelgeleme alınmayacak!

→ sleep'ten farkı, sleep thread'i bloke eder. yield çalışmak üzere önceliği devreder!

thread_local: modern cpp ile thread-local storage duration eklendi.

automatic storage duration
static storage duration
dynamic storage duration
→ Diğer storage türleri!

thread-local storage duration
→ thread local anahtar sözcüğü ile kullanılır.

→ n tane thread çalışıyorsa, n tane thread local nesne olur.

→ Thread'e özgü statik ömürlü değişken gibi. Her thread'e özgü farklı değerlerin saklanması gerektiğinde kullanılır.

```cpp
#include <iostream>
#include <syncstream>
#include <string>
#include <thread>
#include <mutex>

thread_local int tval{ 0 };

void func(const std::string& thread_name)
{
    ++tval; //senkronizasyon gerekmiyor
    std::osyncstream{ std::cout } << "tval in thread " << thread_name << " is " << tval << "\n";
}

int main()
{
    std::thread tx(func, "a");
    std::thread ty(func, "b");

    {
        std::osyncstream{ std::cout } << "tval in main: " << tval << '\n';
    }

    tx.join();
    ty.join();
}
```

→ main thread dışında oluşturulan tüm threadlerde bir tval olacak.
↳ Bu örnekte 3 adet tval olur.

→ main için tval = 0 çıkar
threadler de tval = 1 alıyor!

```cpp
std::mutex mtx;

void func(int id)
{
    int x = 0;
    static int y = 0;
    thread_local int z = 0;

    ++x;
    ++z;
    std::lock_guard guard(mtx);
    ++y;
    std::cout << "thread id : " << id << " x (automatic storage)      = " << x << "\n";
    std::cout << "thread id : " << id << " y (static storage)       = " << y << "\n";
    std::cout << "thread id : " << id << " z (thread local storage) = " << z << "\n\n";
}

void foo(int id)
{
    func(id);
    func(id);
    func(id);
}
```

→ tüm threadler için ortak
→ tüm threadler için ayrı ayrı var

```
thread id : 1 x (automatic storage)      = 1
thread id : 1 y (static storage)         = 1
thread id : 1 z (thread local storage)   = 1

thread id : 1 x (automatic storage)      = 1
thread id : 1 y (static storage)         = 2
thread id : 1 z (thread local storage)   = 2

thread id : 1 x (automatic storage)      = 1
thread id : 1 y (static storage)         = 3
thread id : 1 z (thread local storage)   = 3

thread id : 0 x (automatic storage)      = 1
thread id : 0 y (static storage)         = 4
thread id : 0 z (thread local storage)   = 1

thread id : 0 x (automatic storage)      = 1
thread id : 0 y (static storage)         = 5
thread id : 0 z (thread local storage)   = 2

thread id : 0 x (automatic storage)      = 1
thread id : 0 y (static storage)         = 6
thread id : 0 z (thread local storage)   = 3
```

Threadlerin zaman açısından programa etkisi

```cpp
constexpr uint64_t n = 1'000'000'000;

void get_sum_odds()
{
    for (uint64_t i = 1; i < n; i += 2)
        sum_odd += i;
}

void get_sum_evens()
{
    for (uint64_t i = 0; i <= n; i += 2)
        sum_even += i;
}

int main()
{
    using namespace std::chrono;

    auto start = steady_clock::now();
    get_sum_evens();
    get_sum_odds();
    auto end = steady_clock::now();
    std::cout << "hesaplama tamamlandi toplam sure : " << duration_cast<milliseconds>(end - start).count() << " milisaniye\n";
    std::cout << "teklerin toplami   = " << sum_odd << "\n";
    std::cout << "ciftlerin toplami  = " << sum_even << "\n";
}
```

→ Bu kod yaklaşık 750 ms sürüyor!

get_sum_evens(); get_sum_odds(); → bu işlemler thread olarak çalışsaydı, 500 ms tutuyor!

* std::thread::hardware_concurrency ile, hardware da ayrı ayrı çalıştırabileceğimiz thread sayısını öğrenebiliriz. Fakat bu threadlerin farklı core larda çalışacağı garantisini vermez!

#Native Handle: → Fonksiyonun return değeri bir adres → işletim sisteminin thread'inin bir adresi

#Async: - threadleri paralel çalıştıran, workloadların return değerini alabilen, eğer onlardan biri exception gönderirse, onu da handle edebilen bir abstraction
- return değeri std::future'ın specialization'ı.
- launch policy ayarlanabilir.
  → senkron
  → asenkron (ihtiyaç duyarsa) } → std::launch
  → compiler neyi uygun görürse

```cpp
#include <thread>
#include <future>

int foo(int x)
{
    std::cout << "foo cagrildi x = " << x << "\n";

    return x * x;
}

int main()
{
    std::future<int> ft1 = std::async(foo, 10);
}
```

std::future<int> → use auto type deduction

```cpp
int main()
{
    auto ft = std::async(foo, 5);

    auto val = ft.get();

    std::cout << "val = " << val << "\n";
}
```

ft.get() → return değeri için → Future.get() func

get fonksiyonu

Çağırılana kadar, thread koşmuş ve func bitmiş olabilir
Halen çalışıyor olabilir, o zaman bloke olucak
Eğer deferred olsaydı, o an çalıştırılacaktı!

```cpp
#include <stdexcept>
#include <future>
#include <iostream>

int func()
{
    //throw std::runtime_error{ "error from func\n" };
    return 1;
}

int main()
{
    try {
        auto ftr = async(std::launch::async, func);
        auto val = ftr.get();
        std::cout << "return value is: " << val << "\n";
    }
    catch (const std::exception& ex) {
        std::cout << "exception caught: " << ex.what() << "\n";
    }
}
```

```cpp
#include <stdexcept>
#include <future>
#include <iostream>

int func();
int foo();

int main()
{
    auto ft_foo = async(std::launch::async, foo);
    auto func_val = func();

    auto x = func_val + ft_foo.get();
}
```

std namespace'inden bir argüman yollandığı için ADL kullandık. std:: async diye yazmadık

eğer thread kullanmasaydık x hesaplanması için önce func + foo koşardı. Sonra eğer foo paralelde hesaplanıyorsa süre azaldı!

→ Deferred vs Async:

```cpp
int main()
{
    auto start = std::chrono::steady_clock::now();

    //auto ftr1 = async(std::launch::deferred, get_str_letters, 20);
    //auto ftr2 = async(std::launch::deferred, get_str_digits, 20);

    auto ftr1 = async(std::launch::async, get_str_letters, 20);
    auto ftr2 = async(std::launch::async, get_str_digits, 20);

    auto s1 = ftr1.get();
    auto s2 = ftr2.get();
    auto end = std::chrono::steady_clock::now();
    std::cout << std::chrono::duration<double>(end - start).count() << " saniye\n";

    std::cout << s1 + s2 << "\n";
}
```

6 ms + overhead

3 ms + overhead

tam bitirsinler 3 saniye mi bloke edecek.

```
auto f1 = async(std::launch::async, get_str_letters, 20);
auto f2 = async(std::launch::async, get_str_digits, 20);
```

Burada future nesnelerinin get fonksiyonu f1 ve f2'nin implicit barrier çağırılır implicit olarak! Get f1 ve f2'nin destructor'ı otomatik çağırılır.