

Concurrency: Birlikte bir zaman aralığında birden fazla iş yapmak \rightarrow Bağımsız iki veya bir arada yapılması \rightarrow Anadod tek osardaki menmen olmasa da birden fazla iş aynı zamanda yürütülebilir.

Parallelism: İşler sırasıyla依次 yapılış \rightarrow Birden fazla görevin kontrolü.

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

İşletim sisteminde thread kurmanın temel eden bir yöntemdir.
- Bir testi, thread olarak okutturmak istiyorsak, onu böyle bir thread nesnesi ile ilişkilendirebilir. Thread sınıfı, tarafından nesne görelidir!

```
int main()
{
    std::thread t;

    std::cout << std::boolalpha;
    std::cout << t.joinable() << "\n"; boolean değer döndürür. → false: ońka başlı oldugu bir iş yok!
```

```
oid func()

std::cout << "func\n";
std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main()
{
    std::cout << std::boolalpha;
    std::thread t{ func }; simdi iş开端!
    std::cout << t.joinable() << "\n"; true
    t.join(); iş gerçelleştirilir.
    std::cout << t.joinable() << "\n"; false
```

```
void func()
{
    for (int i = 0; i < 1000000; ++i) {
        std::cout << '*';
    }
}
```

```
int main()
{
    using namespace std;

    thread t{ func };
    for (int i = 0; i < 1000000; ++i)
        std::cout << '.';
    t.join();
    ///
}
```

\rightarrow Burada lorsque 2 thread var. Fakat her biri sırası / sırası deterministik değil
(Hem yıldız hem nokta basar)

\rightarrow Fakat gerki olan tek sey:

Jorn分工ının olindan kodlar bloke edilebilir.

* Concurrency in Action - Antony Williams

* Journable bir thread nesnesinin destroy edilmesi olgusunda terminate fonksiyonun çağrılmasını sağlar. Oyordan ya join
ya da detach → thread'in durumunu
Dönüşü döndür / tekrarı
soluk!

```
int main()
{
    using namespace std;

    thread tx; → join edilebilir bir thread değil
    → yine yok

    try {
        tx.join();
    }
    catch (const std::exception& ex) { → invalid argument exception
        std::cout << "exception caught: " << ex.what() << '\n';
    }
}
```

* Argument olarak callable tırn thread: - varsayıla elektronik argumentler thread ctor'a gecir. Argument'un işaretçi (pointing to callable) olması tırtılıdır.

```
void foo(int x, int y, int z)
{
    std::cout << "foo x = " << x << "\n";
    std::cout << "foo y = " << y << "\n";
    std::cout << "foo z = " << z << "\n";
}

int main()
{
    using namespace std;

    thread tx{ foo, 10, 20, 30};

    tx.join();
}
```

```
void foo(const std::vector<int> & ivec) → Fonksiyon nesnesinin kendisini taşıyır.

{
    for (const auto val : ivec)
        std::cout << val << " ";
}

int main()
{
    using namespace std;

    std::vector<int> ivec{ 1, 2, 3, 4, 5 };

    thread tx{ foo, ivec}; → Tırtılı bir şekilde işlevsellik semantiği olmaz gereklidir. Çünkü Fonksiyon  
gelen nesnenin kendisi değil. Bunun gerekçisi reference wrapper  
hüllünmeli!!! Reference wrapper ile nesnenin kendisi gider.
}
```

```

void foo(const std::vector<int> & ivec)
{
    for (const auto val : ivec)
        std::cout << val << " ";
}

int main()
{
    using namespace std;

    std::vector<int> ivec{ 1, 2, 3, 4, 5 };

    thread tx{ foo, ref(ivec) };
    tx.join();
}

```

↓
std::ref die Argumente aus dem Gedenkmann!

* Hölle C++:

The diagram shows two code snippets side-by-side. The left snippet has a red box around the parameter `ivec` in the `foo` function declaration, with a red arrow pointing to it labeled "L value reference". The right snippet has a red box around the argument `ivec` in the `tx{ foo, ivec };` line, with a red arrow pointing to it labeled "R value". A green arrow points from the "L value reference" label to the "R value" label, indicating they are related.

```

void foo(std::vector<int> & ivec)
{
    for (const auto val : ivec)
        std::cout << val << " ";
}

int main()
{
    using namespace std;

    std::vector<int> ivec{ 1, 2, 3, 4, 5 };

    thread tx{ foo, ivec };
    tx.join();
}

```

→ Syntax error. Da ist L.value.ref nicht gute R.value.
Funktionsname → const L.value.ref kann gar nicht

```

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

int main()
{
    using namespace std;

    auto fp = foo;
    thread tx{ fp, 21 };
    tx.join();
}

```

Function pointer das
keine Objekt!

```

class Functor {
public:
    void operator()(int x) const
    {
        std::cout << "x = " << x << "\n";
    }
};

int main()
{
    using namespace std;
    Functor f;
    thread tx{ f, 12 };
    tx.join();
}

```

Functor class das
keine Objekt!

* Mülakat Soru:

```
class Functor {  
public:  
    void operator()()const  
    {  
        std::cout << "operator()()\n";  
    }  
  
};  
  
int main()  
{  
    using namespace std;  
  
    thread tx(Functor());  
  
    //tx.join(); → bu satırda sorun yok  
}
```

most vexing parse!!!
Düzenli biri, bir koyan
declaration close deklarasyon
değilidir!!!

//tx.join(); → bu satırda sorun yok

* Non-Static Member Functions as Threads:

```
class Nec {  
public:  
    void func()  
    {  
        std::cout << "Nec::func()\n";  
    }  
};  
  
int main()  
{  
    using namespace std;  
  
    Nec mynec;  
  
    thread t{ &Nec::func, mynec };  
  
    t.join();  
}
```

burada mynec nesnesi KOPYALANDI!

```
int main()  
{  
    using namespace std;  
  
    Nec mynec;  
  
    thread t{ &Nec::func, std::ref(mynec) };  
  
    t.join();  
}
```

şimdilerde kopyalanmaz!

```
void func(const std::ostream& os);  
  
int main()  
{  
    using namespace std;  
  
    thread t(func, ref(std::cout));  
}
```

Kopyalanma koyuldu
osstream nesnesi,
reference wrapper ile
thread'e bindildi.

* Threadlist Kopiierung:

- Threadlist kopieren maga kopie! ohne move eddimage geht!

- Move eddimage threadlist ja nicht abmetten geht!

* Threadlist containernde sollnebiut:

```
7 using namespace std;
8
9
10 int main()
11 {
12     thread tx;
13     thread ty(tx);
14 }
```

Kopieren
schnell
haben!

ty=tx dehabe

```
7 using namespace std;
8
9
10 void func() {}
11
12 int main()
13 {
14     thread tx{ func };
15
16     std::cout << boolalpha << tx.joinable() << "\n"; → tx joinable
17
18     thread ty{ std::move(tx) }; → tx is moved
19
20     std::cout << boolalpha << tx.joinable() << "\n"; → tx is not joinable
21     std::cout << boolalpha << ty.joinable() << "\n"; → ty is joinable
22
23     ty.join();
24     std::cout << boolalpha << ty.joinable() << "\n"; → after ty joined,
25     ty will not be joinable
26 }
27 }
```

```
void func(std::thread t)
{
    t.join();
}

void workload() {}

int main()
{
    func(thread{ workload });
}
```

r value expression obigerum legal

```
using namespace std;

std::thread make_thread()
{
    thread t{} [
        std::cout << "func()\n";
    ];
    return t; → dynamik amobt neinen move eddimage
              garantiert
}

void workload() {}

int main()
{
    thread t;
    t = make_thread();
    r value expression
    t.join();
}
```

* Joining Thread Wrappers:

```
class JThread {
public:
    JThread() noexcept = default;

    template <typename Func, typename ...Args>
    explicit JThread(Func&& f, Args &&...args) : m_t(std::forward<Func>(f), std::forward<Args>(args)...)
    {

    }

    ~JThread()
    {
        if (joinable())
            join();
    }

    explicit JThread(std::thread t)noexcept : m_t(std::move(t)) {}

    JThread(JThread &&other) noexcept : m_t(std::move(other.m_t)) {}

    bool joinable()const noexcept
    {
        return m_t.joinable();
    }

    void join()
    {
        m_t.join();
    }

    void swap(JThread& other)noexcept
    {
        m_t.swap(other.m_t);
    }

    JThread& operator=(JThread&& other)
    {
        if (joinable())
            join();

        m_t = std::move(other.m_t);

        return *this;
    }

private:
    std::thread m_t;
};
```

* Get_id(): return değer thread-id taranır.

başka işçilerin ve kesişmeyen operatörleri var.

```
thread tx{ func };

auto id = tx.get_id(); ↳ hangi thread çalıştırılsa onun id'i  
oysa bkr id değer alınırlar
std::cout << typeid(id).name() << "\n";
```

```
tx.join();
```

```
using namespace std;

std::thread::id g_id;

void func()
{
    auto id = std::this_thread::get_id();
    if (id == g_id) {
        std::cout << "fonksiyon ana thread tarafından calistiriliyor\n";
    }
    else {
        std::cout << "fonksiyon chils thread tarafından calistiriliyor\n";
    }
}

int main()
{
    g_id = this_thread::get_id();

    func();
```

→ Fonksiyonun main thread mi, child thread mi?
çalışıyor - onu onlaşıp, felâkîler yaşatabiliriz?

Sleep-For / Sleep-until Fonksiyonları:

- Sleep for: bir ekran boyasına thread'ı bloke eder.
- Sleep until: bir time point'i ulaşınca hedef thread'ı bloke eder.

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

auto now() { return std::chrono::steady_clock::now(); }

auto awake_time()
{
    using std::chrono::operator""ms;
    return now() + 2000ms;
}

int main()
{
    std::cout << "Hello\n" << std::flush;           // once sayı. Cıktı
    const auto start{ now() };                      // bufferde beklenen
    std::this_thread::sleep_until(awake_time());      // süre流淌
    std::chrono::duration<double, std::milli> elapsed{ now() - start };
    std::cout << "Waited " << elapsed.count() << " ms\n";
}
```

```
using namespace std;

std::thread::id g_id;

void func(int n)
{
    std::this_thread::sleep_for(std::chrono::duration<double>{12.36});
```