



# C++

---

7-8. HÉT – `STD::CHRONO`, SZÁLAK,  
SZEMAFOROK, TASKOK, C++20  
COROUTINE-OK

# std::chrono - Időkezelés sztenderd módon

---

Az std::chrono egy nagyon hasznos kis könyvtár, amely a C++11 óta létezik

Az std::chrono lehetővé teszi órák, időpillanatok és időtartamok reprezentálását (*clocks*, *time points*, *durations*).

Az órák (*clocks*) definíciójához hozzá tartozik:

- egy epoch (kezdeti időpont, mint pl. 1970. január 1-e) és egy
- tick rate (lehet pl. másodpercenkénti vagy milliszekundumonkénti tick rate)

Minden óra osztálynak létezik egy static *now()* függvénye, melynek segítségével az éppen aktuális időpillanat (std::chrono::time\_point) lekérdezhető

- Jelenleg 3 óra típus létezik a chrono könyvtárban: system\_clock, steady\_clock és high\_resolution\_clock.

# std::chrono - Időkezelés sztenderd módon

---

Minden óráról lekérdezhető egy adott időpillanat (*std::chrono::time\_point*) a *now()* függvény segítségével

Időpillanatot csak egy konkrét óra és az azon eltelt időtartam típus (*duration*) alapján lehet értelmezni. (ld. [https://en.cppreference.com/w/cpp/chrono/time\\_point](https://en.cppreference.com/w/cpp/chrono/time_point))

## std::chrono::time\_point

Defined in header <chrono>

```
template<
    class Clock,
    class Duration = typename Clock::duration    (since C++11)
> class time_point;
```

Két időpillanatot egymásból kivonva időtartamot (*std::chrono::duration*) kapunk.

- Időtartamot könnyen példányosíthatunk mi magunk is, ami hasznos, ha pl. egy szálat egy fix időre el szeretnénk altatni (ld. később).

Érdemes a dokumentációt példákkal együtt részletesen átnézni. A C++20-szal már kiíratható output stream operátorral a *duration* típus, ennek alternatívája a *count()* metódus használata.

# std::chrono - Időkezelés sztenderd módon

---

Egy egyszerű példa:

```
#include <iostream>

#include <chrono>

#include <ctime>

long fibonacci(unsigned n) {

    if (n < 2) return n;

    return fibonacci(n - 1) + fibonacci(n - 2);

}
```

```
int main() {

    auto start = std::chrono::system_clock::now();

    std::cout << "f(42) = " << fibonacci(42) << '\n';

    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end -
start;

    std::time_t end_time =
std::chrono::system_clock::to_time_t(end);

    std::cout << "finished computation at " <<
std::ctime(&end_time)

    << "elapsed time: " << elapsed_seconds.count() <<
"s\n";

}
```

# std::chrono - Időkezelés sztenderd módon

---

Ha Visual Studio-ban errort kapunk, az azért van, mert a felhasznált `std::ctime` (ami ekvivalens a lenti `asctime`-mal) függvény static adatokra ad vissza pointert, ami nem thread safe.



4



The reason those functions have different parameters is that the lack of safety is caused by only having a single parameter. In particular, `asctime()` uses a single buffer to return the time. So if you do something like:

```
char *s1 = asctime((time_t)0); // 1-Jan-1970 00:00:00 or something like that.
time_t t = time();
char *s2 = asctime(t);
cout << "1970: " << s1 << " now:" << s2 << endl;
```

Then you will not see two different times printed, but the current time printed twice, as both `s1` and `s2` point to the same string.

The same applies to `localtime`, which returns a pointer to `struct tm` - but it's always the same `struct tm`, so you if you do:

```
struct tm* t1 = localtime(0);
struct tm* t2 = localtime(time());
```

you will get the same values in `t1` and `t2` (with the "current" time, not 1970).

So, to fix this problem, `asctime_s` and `localtime_s` have an extra parameter that is used to store the data into. `asctime_s` also has a second extra parameter to tell the function how much space there is in the storage buffer, as otherwise, it could overflow that buffer.



# std::chrono - Időkezelés sztenderd módon

Egy workaround lehet, hogy beállítjuk a preprocesszor paramétereinél, hogy ne figyelmeztessen erre a fordító.

Ha végképp thread-safe megoldást szeretnénk, arra van a jól bevált módszer a `ctime_s` használata (ez a `time.h` headerben található meg):

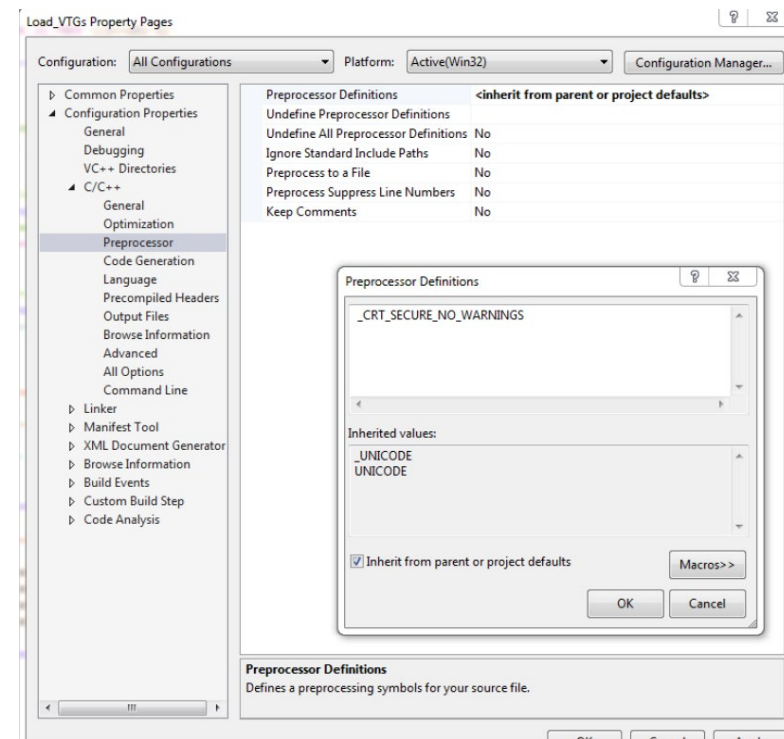
```
// std::cout << "finished computation at " << std::ctime(&end_time)

//<< "elapsed time: " << elapsed_seconds.count() << "s\n";

char str[26];

ctime_s(str, sizeof str, &end_time);

std::cout << "finished computation at " << str << "elapsed time: " <<
elapsed_seconds.count() << "s\n";
```



Select Microsoft Visual Studio Debug Console

```
f(42) = 267914296
finished computation at Mon Mar  8 14:06:41 2021
elapsed time: 17.4239s
```

# Szálak

---

A szálkezelés (threads, multithreading) alapvető absztrakció az ún. konkurrens programozásban

Minden program, amit a gépen elindítunk, egy processzben fut. A processz tartalmazza egyrészt a futtatható kódot, másrészt a program futásához szükséges memóriát (stack, heap, adatszegmens) és úgy általában az operációs rendszer által biztosított kontextust (a CPU regiszterek tartalmát is!).

- Alapesetben a processzek egy szálát tartalmazznak. De olyan is lehet, hogy egy processz többszálú. Ilyenkor “párhuzamosan” (valójában időosztott multiplexeléssel) futnak a különböző szálak, de osztozhatnak a memórián / regisztereken
- A szálak közötti kommunikáció általában gyorsabb, mint a processzek közötti, mivel a szálak egy “futtatási környezetben” futnak

A szálak különböző állapotokban lehetnek attól függően, hogy futásra készen állnak, blokkolnak vagy végeztek a bennük elindított számításokkal (terminated). A szálak működését általában az adott programozási nyelv által biztosított API-n keresztül tudjuk egymáshoz képest és a “main thread”-hez képest szinkronizálni.

# Szálak C++-ban

---

A 2011-es C++ szabvány előtt nem voltak alapból thread-ek a C++-ban. Külső könyvtárakat kellett használni (pl. POSIX-thread könyvtárakat) a többszálú programozáshoz. Ez elég fájdalmas volt, mivel a portabilitást megnehezítette

A C++11-be szerencsére bekerült az `std::thread` osztály (`<thread>` header), ami viszont nyelvi (szabvány) szinten támogatja a szálak létrehozását és menedzselését

Egy új szál elindításához C++-ban mindössze annyit kell tennünk, hogy:

- példányosítjuk az `std::thread` osztályt
- átadunk neki egy meghívható (callable) egyedet (ez lehet függvény pointer, függvény objektum vagy akár lambda kifejezés is)

A szál befejezését megvárhatjuk a szál objektum `join()` metódusának meghívásával.

- Ha a szál végtelen ciklusban van, kívülről nem tudjuk szépen leállítani (nincs `stop()` vagy `terminate()` metódus hogy ne jöjjön létre az erőforrások szempontjából inkonzisztens állapot). Egy lehetséges megoldás, hogy a szálban futó kód pollingol egy változót, hogy mikor lépjen ki a ciklusból (ld. később)



# Szálak C++-ban

---

A C++ szálak alapesetben nem adhatnak vissza értéket, de erre vonatkozóan vannak workaroundok:

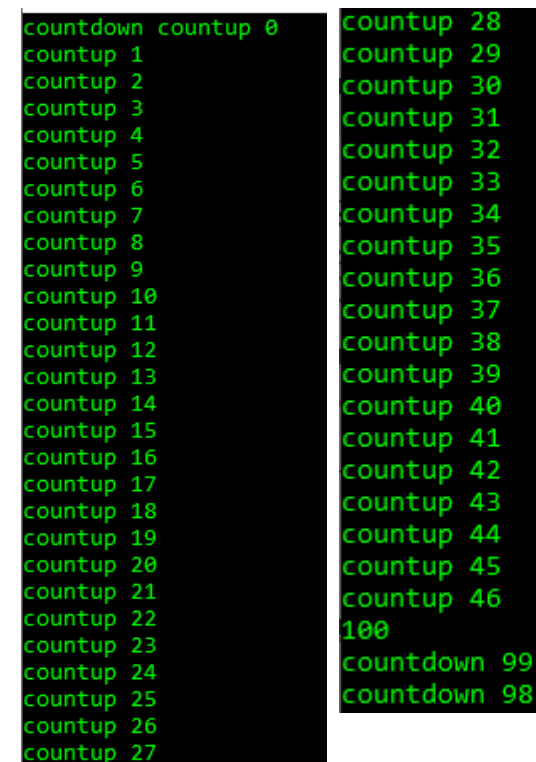
- Egyrészt referenciaként is átadhatunk a threadnek paramétereket (alapesetben pass-by-value szemantika érvényesül, de lambda függvények tudnak referenciaként “elkapni” változókat, vagy használhatjuk az `std::ref` függvényt is)
- Másrészt felhasználhatjuk az `std::promise` osztályt (ld. később)
- Harmadrészt használhatunk a szálak között megosztott memóriát (de ilyenkor – ha a memóriát több szál írja és / vagy olvassa, a hozzáférést szemaforokkal kell védeni, magyarul a “thread safety-t” mi magunk kell, hogy biztosítsuk)

Ha egy szálban futó kód kivételt dob, a program kilép (meghívódik az `std::terminate`, mint egy el nem kapott kivételnél)

# Szálak C++-ban – egy egyszerű példa

```
int main() {  
  
    auto countdown_lambda = [](int lim) {for (int i = lim;  
i > 0; i--) { std::cout << "countdown " << i <<  
std::endl; }};  
  
    auto countup_lambda = [](int lim) { for (int i = 0; i  
< lim; i++) { std::cout << "countup " << i << std::endl;  
}};  
  
    std::thread thread1(countdown_lambda, 100); // a  
parametereket a callable után adjuk meg  
  
    std::thread thread2(countup_lambda, 100);  
  
    thread1.join(); thread2.join();  
}
```

A kimeneten láthatjuk, hogy néha az adott kódsoron belül (mivel nyilván több CPU utasításból áll) is történhet context switch a szálak között:



```
countdown countup 0  
countup 1  
countup 2  
countup 3  
countup 4  
countup 5  
countup 6  
countup 7  
countup 8  
countup 9  
countup 10  
countup 11  
countup 12  
countup 13  
countup 14  
countup 15  
countup 16  
countup 17  
countup 18  
countup 19  
countup 20  
countup 21  
countup 22  
countup 23  
countup 24  
countup 25  
countup 26  
countup 27  
countup 28  
countup 29  
countup 30  
countup 31  
countup 32  
countup 33  
countup 34  
countup 35  
countup 36  
countup 37  
countup 38  
countup 39  
countup 40  
countup 41  
countup 42  
countup 43  
countup 44  
countup 45  
countup 46  
100  
countdown 99  
countdown 98
```

# Szálak C++-ban – referenciák használata

---

```
int pass_this_by_ref = 0;

auto incr_ref_n_times = [&pass_this_by_ref](int ntimes) {
    for (int i = 0; i < ntimes; i++) {
        std::cout << "incr,"; pass_this_by_ref++;
    }
};

auto double_ref_n_times = [&pass_this_by_ref](int ntimes) {
    for (int i = 0; i < ntimes; i++) {
        std::cout << "dbl,"; pass_this_by_ref *= 2;
    }
};
```

```
std::thread thread1(incr_ref_n_times, 50);
std::thread thread2(double_ref_n_times, 5);
```

```
thread1.join();
```

```
thread2.join();
```

```
std::cout << "final value is always different  
(right now it's " << pass_this_by_ref << ")" <<  
std::endl;
```

# Szálak C++-ban – referenciák használata

Microsoft Visual Studio Debug Console

[illegible]

Microsoft Visual Studio Debug Console

```
dbel,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,  
incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,incl,  
incl,incl,incl,dbel,dbel,dbel,dbel,final value is always different (right now it's 800)
```

Microsoft Visual Studio Debug Console

```
incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,  
incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,  
incr,incr,dbel,dbel,dbel,dbel,dbel,final value is always different (right now it's 1600)
```

Microsoft Visual Studio Debug Console

```
incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,  
incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,incr,dbel,dbel,dbel,dbel,dbel,incr,  
incr,incr,incr,incr,incr,incr,incr,final value is always different (right now it's 1352)
```

# Szálak C++-ban – referenciák használata

---

Ahogy említettük, referencia `std::ref`-fel is használható. Ilyenkor a szálban futó callable referencia argumentumot is vár:

```
// szinten jó lett volna:
//auto mylambda = [&counter_val](int starting_val) {
auto mylambda = [](int starting_val, int& cntval) {
    std::this_thread::sleep_for(std::chrono::seconds(1));
    for (int i = starting_val; i < starting_val + 1000; i++) {
        // counter_val = i;
        cntval = i;
    }

    std::cout << "Thread finished" << std::endl;
};

// std::thread th {mylambda, starting_val};
std::thread th{ mylambda, starting_val, std::ref(counter_val) };
```

# Szálak C++-ban – promise-szal

---

A promise-t mint koncepciót érdemes megismerni, mert nemcsak a C++-ban használatos (az utóbbi években alapvető jelentőségűvé vált például a Javascript / NodeJS világban is)

A javascriptes világban egy promise objektum lényegében egy ígéretet reprezentáló objektum egy számítás elvégzésére, amelyből később az eredményt lekérdezhetjük.

Általában van valamilyen mechanizmus annak kezelésére, ha a számítást nem sikerült végrehajtani (reject() callback / catch függvény), illetve az eredmény automatizált tovább processzálására (resolve() callback, then() függvény)



# Szálak C++-ban – promise-szal

---

C++-ban kicsit mások a `<future>` headerben megtalálható `std::promise` körüli alapfogalmak, de hasonló a szemantikája.

C++-ban egy `std::promise` objektum létrehoz egy `std::future` objektumot (ez utóbbi lekérdezhető a `get_future()` metódus segítségével)

- Miután lekérdeztük a `future` objektumot, a `promise` objektumot átadhatjuk akár egy másik `thread`-nek is (fontos, hogy ezt `std::move()`-val kell megtennünk!)
- Az a `thread`, amely ezután birtokában van a `promise` objektumnak, beállíthat rajta akármilyen eredményt
- A kiinduló szálból pedig a korábban megszerzett `future` objektumon keresztül le tudjuk kérdezni hogy van-e eredmény, illetve magát az eredményt.

# Szálak C++-ban – promise-szal

---

Huhh, nézzük ezt meg egy példán keresztül. Itt thread2 és thread3 nyomán ugyanúgy 1-et ír ki a program, mert thread1 a kettő között végez (és állítja false-ra control\_var-t)

```
bool control_var = true;

auto wait_n_flip = [&control_var](std::promise<int> res, int secs) {

    std::this_thread::sleep_for(std::chrono::seconds(secs));

    if (control_var) { control_var = false; res.set_value(1); }

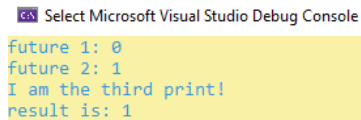
    else { control_var = true; res.set_value(0); } };
```

```
std::promise<int> res1, res2, res3;

std::future<int> res1future = res1.get_future();

std::future<int> res2future = res2.get_future();

std::future<int> res3future = res3.get_future();
```



```
Select Microsoft Visual Studio Debug Console
future 1: 0
future 2: 1
I am the third print!
result is: 1
```

```
std::thread thread1(wait_n_flip, std::move(res1), 2);

std::thread thread2(wait_n_flip, std::move(res2), 1);

std::thread thread3(wait_n_flip, std::move(res3), 10);

std::cout << "future 1: " << res1future.get() << std::endl; // var, amig "el nem készült" az eredmény

std::cout << "future 2: " << res2future.get() << std::endl; // var, amig "el nem készült" az eredmény

std::cout << "I am the third print!" << std::endl;

if (int result = res3future.get()) { // amikor kész, akkor megy be ide:

    std::cout << "result is: " << result << std::endl;

}

thread1.join(); thread2.join(); thread3.join();
```

# Szálak C++-ban – promise-szal

---

Adott future állapotát a wait\_for() metódussal tesztelhetjük. Ez vár x időt majd megmondja, hogy mi a future állapota:

```
bool control_var = true;

auto wait_n_flip = [&control_var](std::promise<int> res, int
secs) {

    std::this_thread::sleep_for(std::chrono::seconds(secs));

    if (control_var) { control_var = false; res.set_value(1); }

    else { control_var = true; res.set_value(0); }

};

std::promise<int> res1, res2;

std::future<int> res1future = res1.get_future();
```

```
std::future<int> res2future = res2.get_future();

std::thread thread1(wait_n_flip, std::move(res1), 2);

std::thread thread2(wait_n_flip, std::move(res2), 1);

if (res1future.wait_for(std::chrono::milliseconds(5)) !=
std::future_status::ready) {

    std::cout << "future 1 not ready at all yet..." << std::endl;

}
```

# Szálak C++-ban – promise-szal

---

Egy szálát leállíthatunk úgy is, hogy átadunk neki egy `std::future` objektumot és a hozzá tartozó `promise`-t később beállítjuk arra az értékre, amivel a `future`-ön keresztül üzeni tudunk:

```
std::promise<void> exitSignal; // ez a promise fogja a szalat
leallitani

std::future<void> futureObj = exitSignal.get_future(); // a promise
letrahoz egy future objektumot

auto threadFunc = [](std::future<void> futureObj) {

    std::cout << "thread started" << std::endl;

    // ha nem ready az allapota, akkor timeout:

    while (futureObj.wait_for(std::chrono::milliseconds(1)) ==
std::future_status::timeout) {

        std::cout << "working..." << std::endl;

        std::this_thread::sleep_for(std::chrono::milliseconds(1000));

    }
}
```

```
std::cout << "thread finished" << std::endl;

};

std::thread th(threadFunc, std::move(futureObj)); // elinditjuk
a szalat

th.detach(); // levalaszthatjuk a main thread-rol hogy ne
kelljen join()-t hivni

std::this_thread::sleep_for(std::chrono::seconds(5));

exitSignal.set_value(); // itt leallitjuk ("uzenunk" neki)
```