

# Standard Template Library alapok

OOP és DB: OOP 6. hét

# Az C++ Standard Library története

1993-ban Alexander Stepanov bemutatott egy új könyvtárat az ANSI/ISO C++ szabványtestületének - amit **Standard Template Library (STL)**-nek nevezték. A könyvtár fogadtatása kitörően pozitív volt.

1994 nyarára Stepanov és Meng Lee által összeállított javaslatok belekerültek a C++ szabvány vázlatába (draft version). Augusztusra a Hewlett Packard is felkarolta a projektet, és az ő megvalósításuk (amiben Stepanov is részt vett) széleskörben elterjedt.

A végleges szabványba soha nem került bele, de a **C++ Standard Library** igen, aminek erős alapot ad az STL. A C++ SL sem nem rész-, sem pedig nem bővített halmaza az STL-nek, de rengeteg a közös pont bennük. A köznyelvben ma is sokan mondanak STL-t C++ Standard Library helyett.

# Standard Template Library

Érdemes nekünk is használnunk a Standard Template Library nevet - bár tudnunk kell, hogy itt az eredetinek arra a részére gondolunk, ami be is került a C++ Standard Library-be.

Azért is érdemes az STL-hez mint névhez ragaszkodunk, mert a C++ Standard Library-ben más célú osztályok / függvények is helyet kapnak, de a mai órán inkább az STL-lel közös részből lesz szó.

# De lássuk, mik is azok a template-ek!

Mire jók a template-ek, milyen célt szolgálnak?

Osztályok, függvények és egyéb nevek (aliasok) **típussal történő paraméterezést** teszik lehetővé, hogy ne kelljen többször ugyanazt lekódolni

*Pl. egy vektor tárolhat inteket meg stringeket is - felesleges lenne kétféle vektor osztályt készíteni.*

**Általános koncepciók tömör reprezentálására** is jók a template-ek (pl. lehet egy template osztályt készíteni ami bármilyen típussal működik, ami támogatja a szorzat operátort, vagy valami más operátort.

# Mire jók a template-ek?

Sok esetben gondolhatunk a template-ekre úgy, mintha típus-absztrakciók lennének. Az adott template működése csak azoktól a közös dolgoktól függ, amiket bizonyos típusokból meg akarunk ragadni.

Másszóval: a közös dolgokon túl semmi egyéb kapcsolatot nem feltételezünk közöttük.

Például:

Azon kívül, hogy mátrixok és vektorok is skalár-szorozhatóak, semmilyen kapcsolat nem kell hogy közöttük legyen ahhoz, hogy általános skalárszorzat-függvényt írjunk (nem kell egymásból sem származniuk)

# Mire jók a template-ek? Mire kell odafigyelni?

A template-ek egyik fő előnye, hogy a tervezőnek nem kell lekötnie, hogy csak adott típus használható, más nem.

A kód így rugalmasabb lesz, hiszen nem kell  $n$  különböző típusra  $n$ -szer (majdnem) ugyanazt lekódolni.

Ugyanakkor template-ek készítésekor pár dologra oda kell figyelnünk.

Ha olyan paraméterrel példányosítunk egy template-et, ami nem teljesít minden követelményt (pl. a template osztály egyik metódusa meghív egy olyan másik fv-t, ami nem értelmezett a típusra), akkor a hiba nem a példányosításkor, hanem felhasználáskor fog keletkezni.

# Mire kell odafigyelni template-eknél?

Ugyanakkor template-ek készítésekor pár dologra oda kell figyelnünk.

A másik, hogy template osztály (vagy függvény) önmagában nem egy teljes osztály (vagy függvény), hanem inkább egy mintázat arra, hogyan kell egy konkrét osztályt (vagy függvényt) legenerálni (amikor már tudja a fordító, hogy milyen típussal paraméterezzük).

**Ez azt is jelenti, hogy template osztály (vagy függvény) megvalósítása sosem lehet külön fordítási egységben (külön cpp fájlban) - csak include-olt header file-ban.** Ugyanis ha a main.cpp-ben készítek egy ilyen, hogy `std::vector<int> x;` - akkor a fordítónak hozzá kell férnie a teljes implementációhoz (nemcsak a deklarációhoz), hogy az adott osztályt legenerálja.

# Végül: hogyan hozunk létre template-et?

*template<typename C>* - C típust használhatjuk joker típusként a kódban

Azt is írhatjuk, hogy *template <class C>*. Példa template függvények:

```
template <class C>
int f(C a) {
    return a + 1;
}

template <> // mivel itt C típusa std::string, uresen hagyjuk a template argot
int f(std::string s) {
    return 0;
}

int main()
{
    int* k = new int { 5 };
    std::cout << f(5) << std::endl; // 6
    std::cout << f<std::string>("hah") << std::endl; // 0
    delete k;
}
```



# Végül: hogyan hozunk létre template-et?

Osztályra példa:

```
template <typename T>
class ValuePrinter {
private:
    T val;
public:
    // template osztaly minden fv-et a h fajlban kell implementalni!
    ValuePrinter(T v) {
        val = v; // azert, mert amikor peldanyositjuk, be kell
        // helyettesiteni a tipust mindenhova,
        // de a fordito egyszerre csak egy cpp fajlt lat!
    }
    void print();
};
```

```
template <typename T>
void ValuePrinter<T>::print() {
    // minden T tipusra mukodik, ami stream operatorral hasznalhato
    std::cout << "value is " << val << std::endl;
}
```

# Végül: hogyan hozunk létre template-et?

Adott típusra az osztály is specializálható definíció szerint:

```
template <>
class ValuePrinter<char> {
private:
    char val;
public:
    ValuePrinter(char v) { // template osztaly minden fv-et a h fajlban kell implementalni!
        val = v; // azert, mert amikor peldanyositjuk, be kell
        // helyettesiteni a tipust
        // mindenhova, de a fordito egyszerre csak egy cpp fajlt lat!
    }
    void print() {
        std::cout << "character value is " << val << std::endl;
    }
};
```

# Az STL szerkezete

Térjünk vissza a Standard Template Library-re!

4 fő komponenst érdemes megkülönböztetni:

Containers (kontérek) - értékek általános tárolására

Iterators (iterátorok) - konténerekben levő értékek általános iterálására

Functions (függvények) - gyakran újra és újra megvalósított függvények

Algorithms (algoritmusok) - felhasználási mintázatok, mint rendezés, keresés, stb.

# Containerek

Attól függően, hogy milyen struktúrában tárolunk adatokat, más és más típusú konténert érdemes használni.

A **szekvencia-konténerek** valamilyen sorrendbe rendezett adatok tárolására jók. Ilyen container a *vector*, *list*, *deque* (ejtsd: dekk - jelentése: **double-ended queue**), *array*, *forward\_list*.

Ezeknek az osztályoknak mind nagyon hasonló az interfészük, de vannak finomságbeli különbségek. Például a *list* osztály egy duplán láncolt lista (mindegyik elem tartalmaz hivatkozást a következő és előző elemre), ezért a beszúrás nagyon gyors, de az a random access lassú. *vector* esetén pont a beszúrás lassabb lehet, ha éppen át kell méretezni a vektort mert betelt a lefoglalt memória!

# Példát már sokat láttunk...

```
std::list<int> mylist = { 1,2,3,4,5 };
// nincs indexalas, csak vegig menni lehet rajta.
// std::cout << "Element 3 of list is: " << ????? << std::endl;
// viszont: konnyu beszurni!
mylist.insert(
    std::next(mylist.cbegin(), 3),
    10); //4. elem ele szurjuk be
std::cout << "Elements of list are: ";
for (int i : mylist) {
    std::cout << i << ", ";
}
std::cout << std::endl;
```

```
Elements of list are: 1, 2, 3, 10, 4, 5,
Element 3 of vector is: 4
Elements of vector are: 1, 2, 3, 8, 4, 5,
```

```
std::vector<int> myvec = { 1,2,3,4,5 };
// van indexalas is!
std::cout << "Element 3 of vector is: " << myvec[3] << std::endl;
myvec.insert(
    myvec.cbegin() + 3, // ez is mukodik, mert random access
    8); //4. elem ele szurjuk be
std::cout << "Elements of vector are: ";
for (int i : myvec) {
    std::cout << i << ", ";
}
std::cout << std::endl;
```

# Deque

Csak, hogy ezzel az osztállyal is megismerkedjünk: a *deque* sok szempontból hasonlít a vektorhoz, de nem folytonos mem.területen tárolja az elemeit (folytonos részletekben tárolja).

A vektor végére általában konstans a beszúrás (kivéve, ha újra kell méretezni a vektort, és akkor ráadásul minden már meglevő elemét új helyre fogja másolni a futtatási környezet) - ezért azt mondják, hogy *amortized constant*

Deque elejére és végére is nagyon gyorsan be lehet szúrni. Átméretezéskor nem másol semmit, csak újabb területeket foglal le. Viszont az indexálás kicsit lassabb, mivel nem teljesen folytonos.

# Deque - kiváló fák bejárásához

Fa bejárásánál a látogatott listának mindig egyik végéről szedjük ki a következő elemet, és annak gyermekeit egyik vagy másik végére tesszük.

Ha a gyermekeket ugyanarra a végére szúrjuk be, ahonnan kiolvasunk, akkor mélységi bejárást kapunk

Ha ellenkező végére, akkor szélességi bejárást

```
std::deque<Node*> visitedNodes = { root };
while (visitedNodes.size() > 0) {
    Node* node_to_process = visitedNodes.back();
    node_to_process->print();
    visitedNodes.pop_back();
    for (Node* ch : node_to_process->getChildren()) {
        visitedNodes.push_front(ch);
    }
}
```

# A teljes példa

```
class Node {  
    std::string val;  
    std::vector<Node*> children;  
public:  
    Node(std::string val) : val(val) {}  
    void addChild(Node* np) { children.push_back(np); }  
    std::vector<Node*>& getChildren() { return children; }  
    void print() { std::cout << val << std::endl; }  
};
```



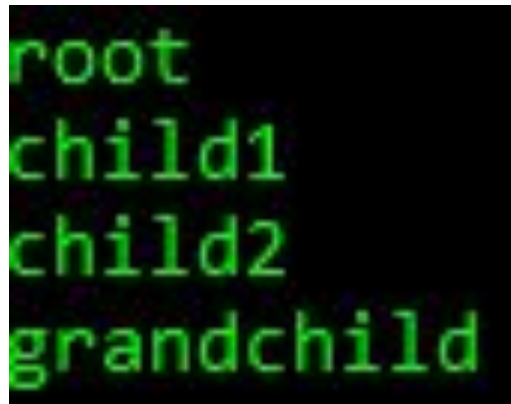
A teljes példa, folyt.

```
class Tree {  
    Node* root;  
public:  
    Tree() : root(new Node("root")) {}  
    Node * getRoot() { return root; }  
    Node * addNode(Node* parent, std::string val) {  
        Node* ch = new Node(val);  
        parent->addChild(ch);  
        return ch;  
    }  
};
```

# A teljes példa, folyt.

```
Tree tree;
Node* root = tree.getRoot();
Node* ch1 = tree.addNode(root, "child1");
Node* ch2 = tree.addNode(root, "child2");
Node* gc1 = tree.addNode(ch1, "grandchild");

std::deque<Node*> visitedNodes = { root };
while (visitedNodes.size() > 0) {
    Node* node_to_process = visitedNodes.back();
    node_to_process->print();
    visitedNodes.pop_back();
    for (Node* ch : node_to_process->getChildren()) {
        visitedNodes.push_front(ch);
    }
}
```



```
root
child1
child2
grandchild
```

# Containerek, folyt.

A szekvencia-konténerek mellett érdemes ismerni az **asszociatív konténereket** is. Ezekre példa a *set* és a *map*. Az ilyen konténerek értékeket illetve érték-párokat tárolnak rendezett módon. A rendezettség megkönnyíti az érték szerinti keresést.

```
typedef std::pair<int, std::string> jatekos;

// list:
//  a doubly linked list; elements are not stored in contiguous memory.
//  Opposite performance from a vector. Slow lookup and access (linear time), but once a position
//  has been found, quick insertion and deletion (constant time).
std::list<jatekos> psg = { jatekos(7, "Mbappe"), jatekos(9, "Cavani") };
psg.push_front(jatekos(1, "Buffon"));
psg.push_back(jatekos(11, "Di Maria"));
```

# Containerek, folyt.

```
// Set: rendezett elemek, egy elem csak egyszer szerepelhet benne
std::set<jatekos> psgJatekosok;
std::cout << std::endl;
psgJatekosok.empty() ? std::cout << "Meg nincs jatekos a halmazban" : std::cout << "Mar vannak jatekosok";
std::cout << std::endl;
for (jatekos j : psg) {
    psgJatekosok.insert(j);
}
psgJatekosok.empty() ? std::cout << "Meg nincs jatekos a halmazban" : std::cout << "Mar vannak jatekosok";
std::cout << std::endl;
for (jatekos j : psgJatekosok) {
    std::cout << j.first << ": " << j.second << std::endl;
}

std::cout << std::endl;
// mi tortenik, ha valakit ketszer rakunk bele?
psgJatekosok.insert(jatekos(3, "Kimpembe")); // kimpembe egyszer mar benne volt
psgJatekosok.insert(jatekos(13, "Dani Alves")); // Dani Alves sorrendhelyesen kerul bele
psgJatekosok.insert(jatekos(5, "Marquinhos")); // Marquinhos is
for (jatekos j : psgJatekosok) {
    std::cout << j.first << ": " << j.second << std::endl;
}
```

# Containererek, folyt.

```
std::cout << std::endl;
// mi tortenik, ha valakit ketszer rakunk bele?
psgJatekosok.insert(jatekos(3, "Kimpembe")); // kimpembe egyszer mar benne volt
psgJatekosok.insert(jatekos(13, "Dani Alves")); // Dani Alves sorrendhelyesen kerul bele
psgJatekosok.insert(jatekos(5, "Marquinhos")); // Marquinhos is
for (jatekos j : psgJatekosok) {
    std::cout << j.first << ": " << j.second << std::endl;
}
```

```
Meg nincs jatekos a halmazban
Mar vannak jatekosok
1: Buffon
3: Kimpembe
7: Mbappe
9: Cavani
11: Di Maria

1: Buffon
3: Kimpembe
5: Marquinhos
7: Mbappe
9: Cavani
11: Di Maria
13: Dani Alves
```



# Containerrek, folyt.

```
// map: kulcs-ertek parok
// It stores only unique keys and that too in sorted order based on its assigned sorting criteria.
// As keys are in sorted order therefore searching element in map through key is very fast i.e.
// it takes logarithmic time.
// In std::map there will be only one value attached with the every key.
std::map<jatekos, std::string> jatekosToCsapat;
jatekosToCsapat.insert(std::make_pair(jatekos(1, "Buffon"), "PSG"));
jatekosToCsapat.insert(std::make_pair(jatekos(6, "Pogba"), "ManU"));
jatekosToCsapat[jatekos(7, "Mbappe")] = "PSG";
std::cout << std::endl;
if (jatekosToCsapat.find(jatekos(6, "Pogba")) != jatekosToCsapat.end()) {
    std::cout << "Pogba csapata: " << jatekosToCsapat[jatekos(6, "Pogba")] << std::endl;
}
std::cout << std::endl;
std::cout << "A map-en vegig iterálva ezt kapjuk:" << std::endl;
for (std::pair<jatekos, std::string> pair : jatekosToCsapat) {
    std::cout << pair.first.second << " csapata: " << pair.second << std::endl;
}
```

# Iterátorok

Az STL általános interfészt ad konténerek elemeinek iterálására / címzésére. Ezekre már láttunk több példát.

Az alapelv: teljesen mindegy, hogy egy konténer belül hogyan tárol elemeket, végig kell tudni rajtuk iterálni.

Minden konténer osztály névterében van egy iterator nevű belső osztály. Ilyen iterátort általában le lehet kérni a *begin()*, *end()* vagy *cbegin()*, *cend()* metódusokkal (utóbbi esetekben konstans iterátort kapunk, amin keresztül értéket nem lehet módosítani).

Az iterator nemcsak iterálásra, hanem pozíció megjelölésre is jó. Pl. *insert()* esetében megmondja, hova szúrjunk be. Vagy *find()* esetében visszaadja a keresett elem pozícióját (amennyiben nem találjuk, az *end()*-del lesz azonos)

# Functions (<functional> header)

A függvény objektumok olyan objektumok, amelyek megvalósítják az *operator()* metódust, ezért meghívhatóak ugyanúgy, mintha függvények lennének. Az ilyen objektumokat functoroknak is szokták nevezni.

## Function objects

*Function objects* are objects specifically designed to be used with a syntax similar to that of functions. In C++, this is achieved by defining member function `operator()` in their class, like for example:

```
1 struct myclass {  
2     int operator()(int a) {return a;}  
3 } myobject;  
4 int x = myobject (0);           // function-like syntax with object myobject
```

A <functional> headerben sok hasznos függvény elérhető functor formában. Ennek az a fő előnye hogy ezeket a függvényeket átadhatjuk más függvényeknek argumentumként anélkül, hogy meg kéne őket írni.



# Functions (<functional> header)

Például:

```
1 // plus example
2 #include <iostream>      // std::cout
3 #include <functional>    // std::plus
4 #include <algorithm>     // std::transform
5
6 int main () {
7     int first[]={1,2,3,4,5};
8     int second[]={10,20,30,40,50};
9     int results[5];
10    std::transform (first, first+5, second, results, std::plus<int>());
11    for (int i=0; i<5; i++)
12        std::cout << results[i] << ' ';
13    std::cout << '\n';
14    return 0;
15 }
```

Output:

11 22 33 44 55

# Functions (<functional> header)

Például:

```
1 // greater_equal example
2 #include <iostream>      // std::cout
3 #include <functional>    // std::greater_equal, std::bind2nd
4 #include <algorithm>     // std::count_if
5
6 int main () {
7     int numbers[]={20,-30,10,-40,0};
8     int cx = std::count_if (numbers, numbers+5, std::bind2nd(std::greater_equal<int>(),0));
9     std::cout << "There are " << cx << " non-negative elements.\n";
10    return 0;
11 }
```

Output:

```
There are 3 non-negative elements.
```

Már el is jutottunk az <algorithm> headerig. Ritka tömör kódot lehet így írni. `std::count_if()`-fel kiváltottunk egy for ciklust. `std::bind2nd()`-del elkerültük, hogy külön saját functort kelljen írunk, ami nagyobb-egyenlő 0-ra tesztel.

# <algorithm>

Ebben a headerben olyan általánosan megfogalmazható műveletsorozatok találhatók meg, amik gyakran boilerplate kódhoz vezetnek:

- valami fölött iterálni és megszámolni, hogy hány elemre igaz egy predikátum
- valami fölött iterálni és minden elemre végrehajtani egy műveletet
- valami fölött iterálni és egy görgetett eredményt kiszámolni (pl. összeg, szorzat, átlag, ...)
- valaminek az elemeit sorba rendezni
- valaminek az elemei között egy konkrét értéket keresni
- valaminek az elemeit egy predikátum (vagy többértékű fv szerint) partícionálni

# Példák

Output:

Vector is not partitioned

Now, vector is partitioned after partition operation

The partitioned vector is : 2 8 6 5 1 7

```
// C++ code to demonstrate the working of
// partition() and is_partitioned()
#include<iostream>
#include<algorithm> // for partition algorithm
#include<vector> // for vector
using namespace std;
int main()
{
    // Initializing vector
    vector<int> vect = { 2, 1, 5, 6, 8, 7 };

    // Checking if vector is partitioned
    // using is_partitioned()
    is_partitioned(vect.begin(), vect.end(), [](int x)
    {
        return x%2==0;
    })?

    cout << "Vector is partitioned":
    cout << "Vector is not partitioned";
    cout << endl;

    // partitioning vector using partition()
    partition(vect.begin(), vect.end(), [](int x)
    {
        return x%2==0;
    });

    // Checking if vector is partitioned
    // using is_partitioned()
    is_partitioned(vect.begin(), vect.end(), [](int x)
    {
        return x%2==0;
    })?

    cout << "Now, vector is partitioned after partition operation":
    cout << "Vector is still not partitioned after partition operation";
    cout << endl;

    // Displaying partitioned Vector
    cout << "The partitioned vector is : ";
    for (int &x : vect) cout << x << " ";

    return 0;
}
```