STL - Standard Template Library

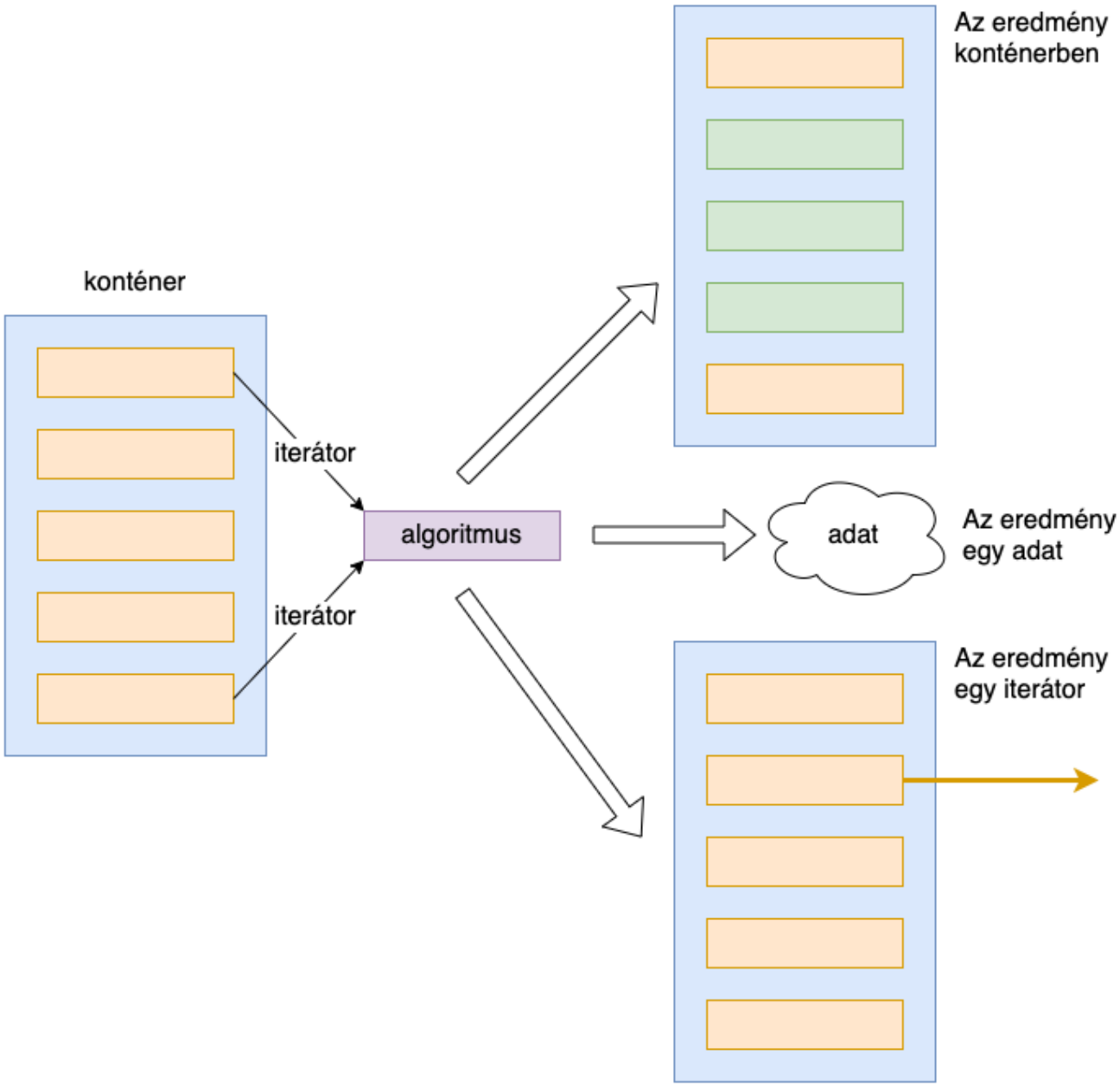
Bevezetés az STL-be

A C++ nyelv Szabványos Sablonkönyvtára (STL) osztály- és függvénysablonokat tartalmaz, amelyekkel elterjedt adatstruktúrákat (vektor, sor, lista stb.) és algoritmusokat (rendezés, keresés, összefésülés stb.) építhetünk be a programunkba.

A sablonos megoldás lehetővé teszi, hogy az adott néven szereplő osztályokat és függvényeket (majdnem) minden típushoz felhasználhatjuk, a program igényeinek megfelelően.

Az STL alapvetően három csoportra épül, a **konténerekre** (tárolókra), az **algoritmusokra** és az **iterátorokra** (bejárókra). Egyszerűen megfogalmazva az algoritmusokat a konténerekben tárolt adatokon hajtjuk végre az iterátorok felhasználásával.

A végrehajtott algoritmus működésének eredményét többféleképpen is megkaphatjuk (konténerben, iterátorként vagy valamilyen egyéb adatként).



A konténerek, az iterátorok és az algoritmusok kiegészítéseként az STL-ben további szabványos elemeket is találunk: helyfoglalókat (*allocators*), adaptereket (*adaptors*), függvényobjektumokat (*function objects* – *functors*)

Függvényobjektumok - funktorok (*functors*, *function objects*)

Az STL sok szabványos algoritmust biztosít a programozók számára, amelyek feldolgozzák a konténerekben tárolt adatokat. Az algoritmusok működése függvényobjektumok (function objects, functors) megadásával testre szabható. Ily módon meghatározhatjuk, hogy milyen műveletek hajtódjanak végre a kollekció elemein. A függvényobjektum hagyományos függvénymutató is lehet, azonban az esetek többségében objektumokat alkalmazunk.

A függvényobjektum olyan típus, amely megvalósítja a függvényhívás (`()`) operátorát. Két lényeges előnye van a közönséges függvényekhez képest:

1. megőrizheti a működés állapotát
2. mivel a függvényobjektum egy típus, megadhatjuk sablon paraméterként

Amikor az egyoperandusú (*unáris*) függvényobjektum `bool` típusú értéket ad vissza **predikátum**-nak nevezzük. **Bináris predikátum**-ról akkor beszélünk, ha egy kétoperandusú függvényobjektum ad vissza **bool** típusú értéket, a két paraméter összevetésének eredményeképp.

A C++11-től használt lambda kifejezések tulajdonképpen névtelen függvénybjektumok.

A függvényobjektumokkal kapcsolatos STL osztálysablonokat a `<functional>` fejállományban találjuk. Egy egyszerű példa a funktorokra:

```
#include <iostream>

struct Add()
{
  int x;
  int operator()(int y) const { return x + y; }
};

int main()
{
  Add a1;
  a1.x = 1;
  std::cout << a1(2) << std::endl;
}
```

> kimenet: 3

A későbbiekben nézni fogunk arra is példát, hogy hogyan lehet funktorok segítségével pl. konténert rendezni.

Konténerek

A konténerek olyan objektumok, amelyekben más, azonos típusú objektumokat tárolhatunk. A tárolás módja alapján a konténereket három csoportba sorolhatjuk:

1. **szekvenciális** (*sequence*) tárolóról beszélünk, amikor az elemek sorrendjét a tárolás sorrendje határozza meg.
2. **asszociatív** (*associative*) konténerek ezzel szemben az adatokat egy kulcssal azonosítva tárolják, melyeket tovább csoportosíthatunk kulcs alapján
    1. **rendezett** (*ordered*) és
    2. **nem rendezett** (*unordered*) tárolókra.

A konténerek sokféleképpen különböznek egymástól:

- a memóriahasználat hatékonysága
- a tárolt elemek elérési ideje
- az új elem beszúrásának, illetve valamely elem törlésének időigénye
- új elem konténer elejére, illetve végére történő beillesztésének ideje
- stb

Szekvenciális tárolók

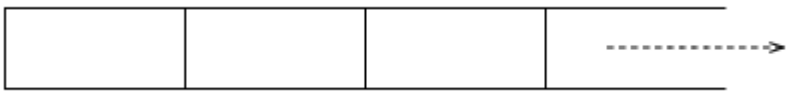
Jellemzőjük, hogy megőrzik az elemek beviteli sorrendjét, azaz a beszúrás ideje határozza meg az elemek sorrendjét. Az `array` kivételével tetszőleges pozícióba beszúrhatunk elemet, illetve törölhetünk onnan. Ezek a műveletek általában a tárolók végein a leggyorsabbak.

`array` - a sablonparaméterben megadott konstans elemszámmal létrejövő, egydimenziós tömbök osztálysablonja.

```
#include <array>

int main()
{
  std::array<int, 5> arr1{{1, 2, 3, 4, 5}};
  std::array<int, 5> arr2 = {1, 2, 3, 4, 5};
}
```

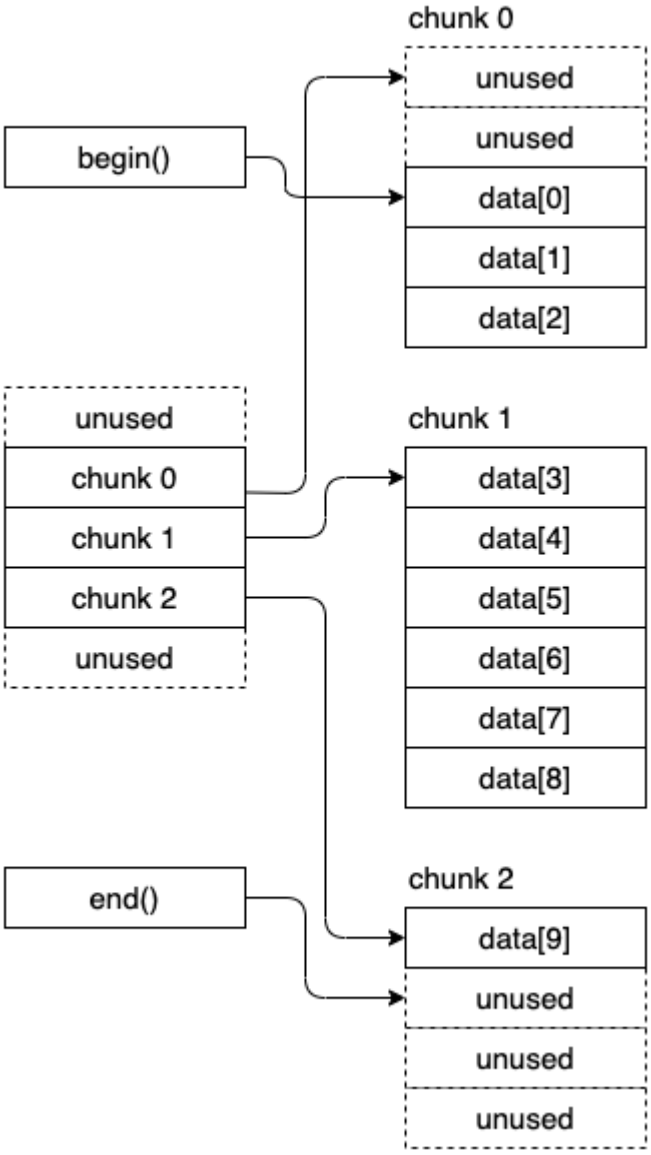
`vector` - a vektor **dinamikus tömbben** tárolódik sorfolytonos memóriaterületen, amely a végén növekedhet, viszont azt már tudjuk, hogy egy tömb mérete nem növelhető. Ezt az `std::vector` a következő képpen oldja meg: ha több elemet szeretnénk beszúrni, mint amennyi az adott vektor kapacitása, akkor lefoglal egy nagyobb memóriaterületet - általában kétszer akkorát - és minden elemet egyesével átmásol erre a memória területre. Ezért a vektor végéhez való elem hozzáadásának műveletigénye amortizált konstans: általában konstans, de ha új memóriaterületet kell lefoglalni és a meglevő elemet átmásolni, akkor lineáris.

Az elemeket indexelve is elérhetjük konstans (*O(1)*) idő alatt. Elemek eltávolítása a (`pop_back()`), illetve hozzáadása (`push_back()`) a vektor végéhez szintén konstans (*O(1)*) idő alatt lehetséges, míg az elején vagy a közepén ezek a műveletek (`insert()`, `erase()`) lineáris (*O(n)*) végrehajtású idejűek. Rendezetlen vektorban egy elem megkeresésének ideje szintén lineáris (*O(n)*).



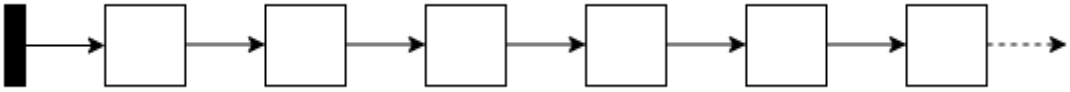
```
#include <vector>

int main()
{
  std::vector<int> vec1{1, 2, 3, 4, 5};
  std::vector<int> vec2 = {1, 2, 3};
}
```

`deque` - **kettősvégű sor**-t megvalósító adatszerkezet, amely mindkét végén növelhető, egydimenziós tömböket tartalmazó listában tárolódik. Elemeket mindkét végén konstans (*O(1)*) idő alatt adhatunk (`push_front()`, `push_back()`) a kettősvégű sorhoz, illetve távolíthatunk el onnan (`pop_front()`, `pop_back()`). Az elemek index segítségével konstans időben (*O(1)*) is elérhetőek.



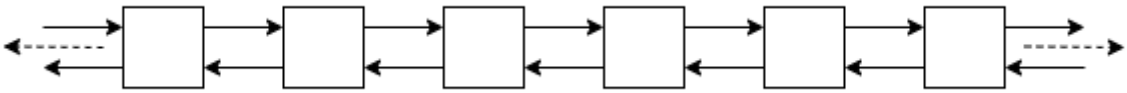

```cpp
#include <deque>

int main()
{
  std::deque<int> deq1{1, 2, 3, 4, 5};
  std::deque<int> deq2 = {1, 2, 3};
}
```

`forward_list` (C++11 óta) - **egyszeres láncolású lista**, melyet csak az elején lehet bővíteni. Azonos elemtípus esetén az elemek helyigénye kisebb, mint a kettős láncolású listáé. Az elemek beszúrása (`insert_after()`) és törlése (`erase_after()`) konstans (*O(1)*) időt igényel.



```cpp
#include <forward_list>

int main()
{
  std::forward_list<int> myflist1{1, 2, 3};
  std::forward_list<int> myflist2 = {3, 2, 1, 4};
}
```

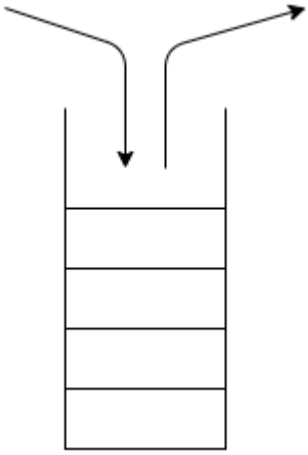
`list` - **kettős láncolású lista**, melynek elemei nem érhetőek el az indexelés operátorával. Tetszőleges pozíció esetén a beszúrás (`insert()`) és a törlés (`erase()`) művelete konstans (*O(1)*) idő alatt végezhető el. A lista mindkét végéhez adhatunk elemeket (`push_front()`, `push_back()`), illetve törölhetünk (`pop_front()`, `pop_back()`) onnan.



```cpp
#include <list>

int main()
{
  std::list<int> mylist1;
  std::list<int> mylist2;

  for (int i = 0; i < 10; ++i)
  {
    mylist1.push_back(i);
    mylist2.push_front(i*2);
  }
}
```

A szekvenciális tárolókra épülő, konténerillesztő osztálysablonok a tároló adapterek. Az alábbi konténer adapterek elemein nem lehet iterátorok segítségével végig lépkedni, ezért semmilyen algortimus hívásakor nem használhatjuk azokat.

`stack` - a **last in first out** (*LIFO*) működésű **verem** adatszerkezet típusa. A verem csak a legfelső pozícióban lévő elem módosítását (felülírás, kivétel, behelyezés) teszi lehetővé. Alapértelmezés szerint a **deque** konténerre épül, azonban a **vector** és a **list** is használható a megvalósításához.

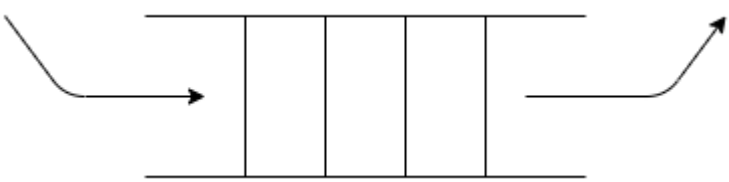


```
#include <iostream>
#include <stack>

int main()
{
  std::stack<int> mystack;
  mystack.push(0);
  mystack.push(1);
  mystack.push(2);

  while (!mystack.empty())
  {
    std::cout << mystack.top() << " ";
    mystack.pop();
  }
}
```

> kimenet: 2 1 0

---

queue - **sor** adatszerkezetet megvalósító típus, amely csak az utolsó pozícióra való beszúrást és az első pozícióról való eltávolítást teszi lehetővé (**first in first out** *FIFO*). Ezeken túlmenően az első és az utolsó elem lekérdezése és módosítása is megengedett. Az alapértelmezetten a **deque** mellett a **list** szekvenciális tárolóra épülve is elkészíthető.



```
#include <iostream>
#include <queue>

void showq(std::queue<int> myq)
```

```
{
  std::queue<int>tmpq = myq;
  while (!tmpq.empty())
  {
    std::cout << tmpq.front() << ' ';
    tmpq.pop();
  }
  std::cout << std::endl;
}

int main()
{
  std::queue<int> myq;
  myq.push(10);
  myq.push(20);
  myq.push(30);

  showq(myq);

  std::cout << "myq.size(): " << myq.size() << std::endl;
  std::cout << "myq.front(): " << myq.front() << std::endl;
  std::cout << "myq.back(): " << myq.back() << std::endl;
  myq.pop();
  showq(myq);
}
```

kimenet:

10 20 30

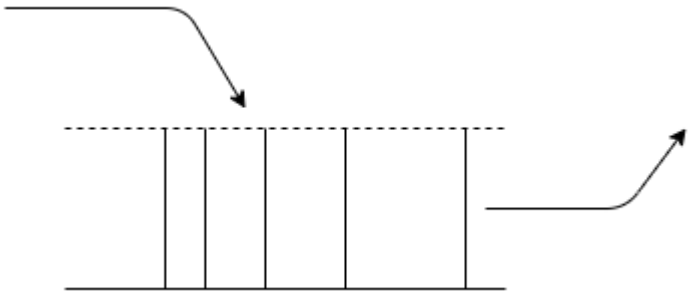
myq.size(): 3

myq.front(): 10

myq.back(): 30

20 30

`priority_queue` - a **prioritásos sorban** az elemek alapértelmezett módon a < (kisebb) operátorral hasonlítva, redezetten tárolódnak. A prioritásos sort csak az egyik, legnagyobb elemet tartalmazó végén érjük el. Ez az elem szükség esetén módosítható, vagy kivehető a sorból. Alapértelmezés szerint a **vector** konténer fölött jön létre, azonban a **deque** is alkalmazható.



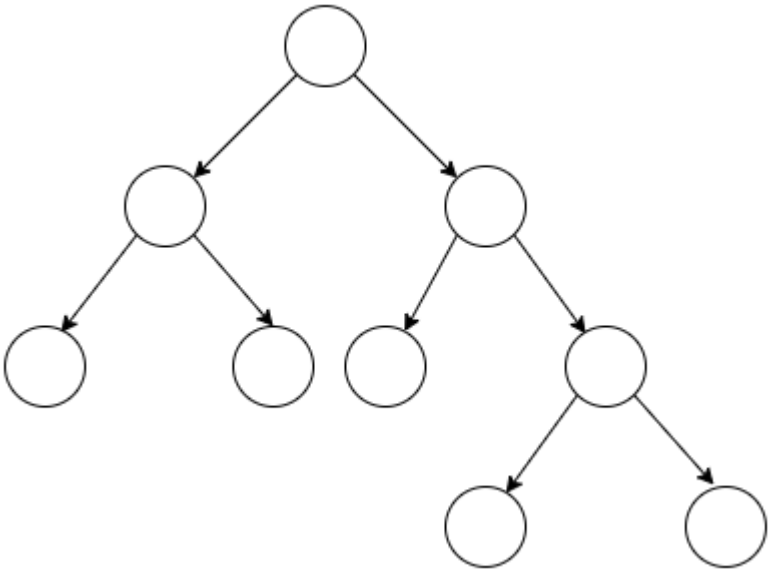
```cpp
#include <iostream>
#include <queue>

void showpq(std::priority_queue<int> prQ)
{
  std::priority_queue<int> tmpPrQ = prQ;
  while (!tmpPrQ.empty())
  {
    std::cout << tmpPrQ.top() << ' ';
    tmpPrQ.pop();
  }
  std::cout << std::endl;
}

int main ()
{
  std::priority_queue<int> myPrQ;
  myPrQ.push(10);
  myPrQ.push(30);
  myPrQ.push(20);
  myPrQ.push(5);
  myPrQ.push(1);

  showpq(myPrQ);

  std::cout << "myPrQ.size(): " << myPrQ.size() << std::endl;
  std::cout << "myPrQ.top(): " << myPrQ.top() << std::endl;
  myPrQ.pop();

  showpq(myPrQ);
}
```

> kimenet:
>
> 30 20 10 5 1
>
> myPrQ.size(): 5
>
> myPrQ.top(): 30
>
> 20 10 5 1

Megjegyzés: a C++ alapértelmezetten **maximum kupacot** (tehát egy olyan kupacot, amelyben bármely elem kulcsa nagyobb vagy egyeblő, mint a gyerekeinek kulcsa) hoz létre a prioritásos sornak.

Asszociatív tárolók

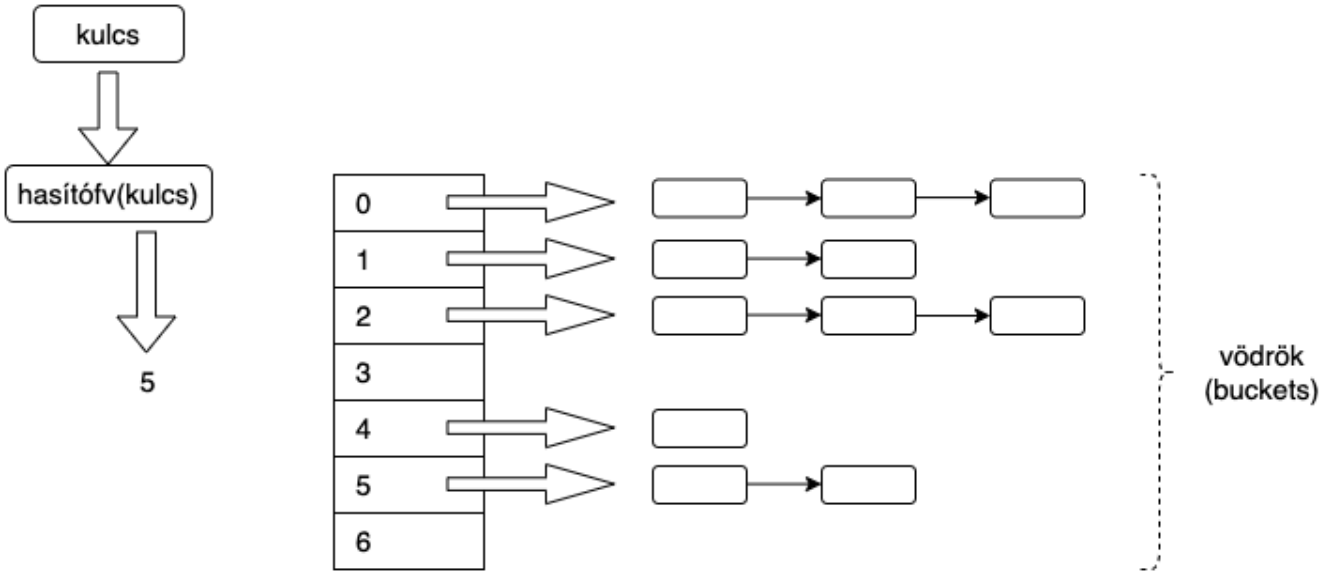
Az asszociatív konténerekben az elemekhez való hozzáférés nem az elem pozíciója, hanem egy kulcs értéke alapján megy végbe. A **rendezett asszociatív tárolók** esetén biztosítani kell a rendezéshez használható kisebb (<) műveletet. Az elemek fizikailag egy önkiegyensúlyozó bináris keresőfa (piros-fekete fa) adastruktúrában helyezkednek el.



A rendezett konténerek esetén általában logaritmikus a végrehajtási idő (*O(log(n))*), a rendezettségnek köszönhetően azonban hatékony algoritmuskkal dolgozhatunk. Ebbe a csoportba tartozik az egyedi kulcsokkal működő halmaz (**set**) és a szótár (asszociatív tömb: **map**), valamint ezek kulcsismétlődést megengedő változatai a **multiset** és a **multimap**.

Megjegyzés: kulcsismétlődés esetén a keresés végehatási ideje lineáris (*O(n)*).

Más a helyzet a **rendezetlen** (*unordered*) asszociatív konténerek esetében. Ebben az esetben az elemek gyors elérése érdekében minden elemhez egy hasító érték tárolódik egy *hash* -táblában. Az elemek elérésekor ismét kiszámítódik a hasító érték, és ez alapján majdnem konstans idő alatt lehet elérni az elemeket.



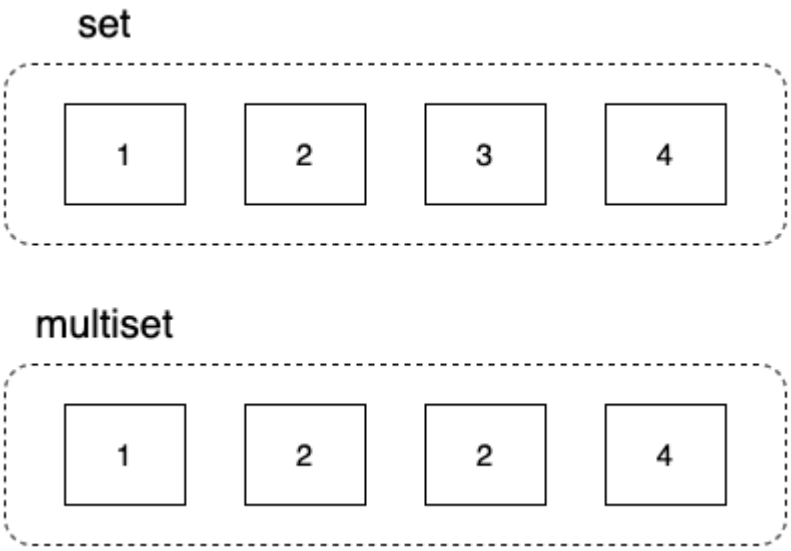
A hasító (*kulcstranszformációs*) függvény a kulcsobjektumot egy indexszé (*hasító kód*) alakítja, amely a hasító táblában kijelöl egy elemet (indexeli azt). A hash-tábla minden eleme az objektumok egy csoportjára (bucket – vödör, kosár) hivatkozik, amennyiben az adott hash-kódhoz tartoznak objektumok. Kereséskor a megfelelő kosár objektumait egymás után a kulcshoz hasonlítva találjuk meg a kívánt objektumot, amennyiban az létezik. Látható tehát, hogy a hasító tábla működésének hatékonysága nagyban függ a

hasító függvénytől. Egy jó hash-függvény véletlenszerűen és egyenletesen osztja szét az objektumokat a „vödrökbe”, minimalizálva ezzel a lineáris keresés lépéseit.

A C++ nyelv alaptípusaihoz az STL biztosítja a megfelelő `hash()` függvényeket (`<functional>` fejléc). A fentebb látható négy asszociatív konténer nem rendezett változatai az **unordered_set**, **unordered_multiset**, **unordered_map**, **unordered_multimap**.

Míg a **set** konténerekben a tárolt adat jelenti a kulcsot, addig a **map** tárolókban (kulcs/érték) adatpárokat helyezhetünk el. Az adatpárok típusa a `pair` struktúrasablon, amely lehetővé teszi, hogy egyetlen objektumban két (akár különböző típusú) objektumot tároljunk. A tárolt objektumok közül az elsőre a `first`, míg a másodikra a `second` névvel hivatkozhatunk. (A `first` jelenti a kulcsot a `second` a hozzá tartozó értéket.)

`set`, `multiset` - mindkét rendezett halmaz konténer a tárolt adatokat kulcsként használja. A **set** -ben a kulcsok (tárolt adatok) egyediek kell legyenek, míg a **multiset** -ben ismétlődhetnek. A két osztálysablon műveletei a `count()` és az `insert()` tagfüggvényektől eltekintve megegyeznek. A **set** esetében a beszúrás logaritmikus idejű (*O(log(n))*), abban az esetben, ha az `insert()` -et használjuk, iterátorral a beszúrás konstans idejű. A keresés szintén logaritmikus idejű (a műveletek kihasználják a rendezettséget).

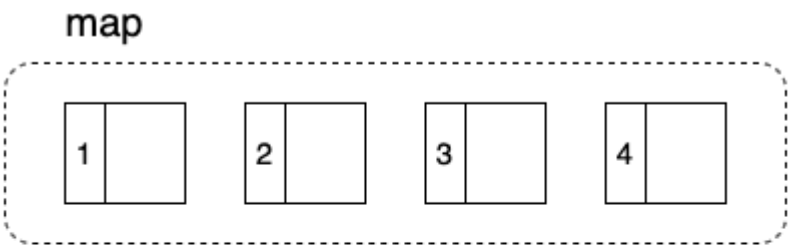


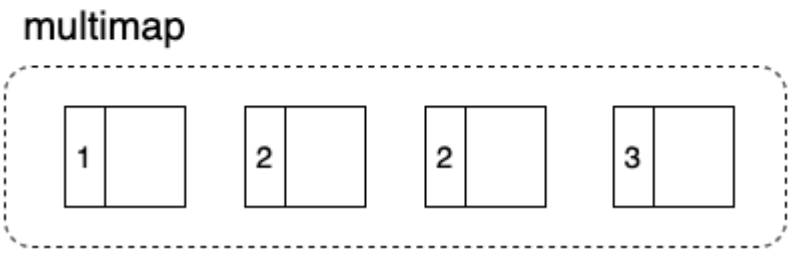

```
#include <iostream>
#include <iterator>
#include <set>

int main()
{
  std::set<int> mySet;
  std::multiset<int> myMultiSet;

  // elemek hozzáadása a set-hez, véletlen sorrendben
  mySet.insert(40);
  mySet.insert(30);
  mySet.insert(60);
  mySet.insert(20);
  mySet.insert(50);
  mySet.insert(50); // csak az első 50-es érték lesz hozzáadva
```

```
    mySet.insert(10);

    // elemek hozzáadása a multiset-hez véletlen sorrendben
    myMultiSet.insert(40);
    myMultiSet.insert(30);
    myMultiSet.insert(60);
    myMultiSet.insert(20);
    myMultiSet.insert(50);
    myMultiSet.insert(50);
    myMultiSet.insert(10);

    // set elemeinek kiírása
    std::set<int>::iterator itr = mySet.begin();
    for (; itr != mySet.end(); ++itr)
    {
      std::cout << *itr << ' ';
    }

    std::cout << std::endl;

    // multiset elemeinek kiírása
    std::multiset<int>::iterator mItr = myMultiSet.begin();
    for (; mItr != myMultiSet.end(); ++mItr)
    {
      std::cout << *mItr << ' ';
    }
  }
```

> kimenet:
>
> 10 20 30 40 50 60
>
> 10 20 30 40 50 50 60

`map`, `multimap` - mindkét asszociatív tömb elemei `pair` típusúak, és kulcs/érték párokat tartalmaznak. A tárolók elemei a kulcs alapján rendezettek. A kulcsok a **map** esetén egyediek, míg **multimap** esetén ismétlődhetnek. A halmazhoz hasonlóan a két osztálysablonnak csak a `count()` és az `insert()` tagfüggvényei különböznek egymástól. Az elemek indexelve vannak nem feltétlen nullától és nem is feltétlenül egymás utáni indexek. A kulcsok értékét nem tudjuk módosítani. Keresés logaritmikus idejű (*O(log(n))*).






```cpp
#include <iostream>
#include <iterator>
#include <map>

int main()
{

  std::map<int, int> myMap;
  std::multimap<int, int> myMultiMap;

  // elemek hozzáadása a map-hez  véletlen sorrendben
  myMap.insert(std::pair<int, int>(7, 10));
  myMap.insert(std::pair<int, int>(2, 30));
  myMap.insert(std::pair<int, int>(1, 40));
  myMap.insert(std::pair<int, int>(5, 50));
  myMap.insert(std::pair<int, int>(2, 30)); // csak az első 2-es kulcsú
kulcs/érték pár lesz beírva
  myMap.insert(std::pair<int, int>(3, 60));
  myMap.insert(std::pair<int, int>(6, 50));
  myMap.insert(std::pair<int, int>(4, 20));

  // elemek hozzáadása a multimap-hez  véletlen sorrendben
  myMultiMap.insert(std::pair<int, int> (5, 50));
  myMultiMap.insert(std::pair<int, int> (1, 40));
  myMultiMap.insert(std::pair<int, int> (6, 50));
  myMultiMap.insert(std::pair<int, int> (3, 60));
  myMultiMap.insert(std::pair<int, int> (4, 20));
  myMultiMap.insert(std::pair<int, int> (6, 10));
  myMultiMap.insert(std::pair<int, int> (2, 30));

  // elemek kiírása kulcs/ érték alapján
  std::map<int, int>::iterator itr = myMap.begin();
  for (; itr != myMap.end(); ++itr) {
      std::cout << itr->first << ' ' << itr->second << std::endl;
  }
```

```
    std::cout << std::endl;

    std::multimap<int, int>::iterator mItr = myMultiMap.begin();
    for (; mItr != myMultiMap.end(); ++mItr)
    {
      std::cout << mItr->first << ' ' << mItr->second << std::endl;
    }
  }
```

kimenet:

1 40

2 30

3 60

4 20

5 50

6 50

7 10

1 40

2 30

3 60

4 20

5 50

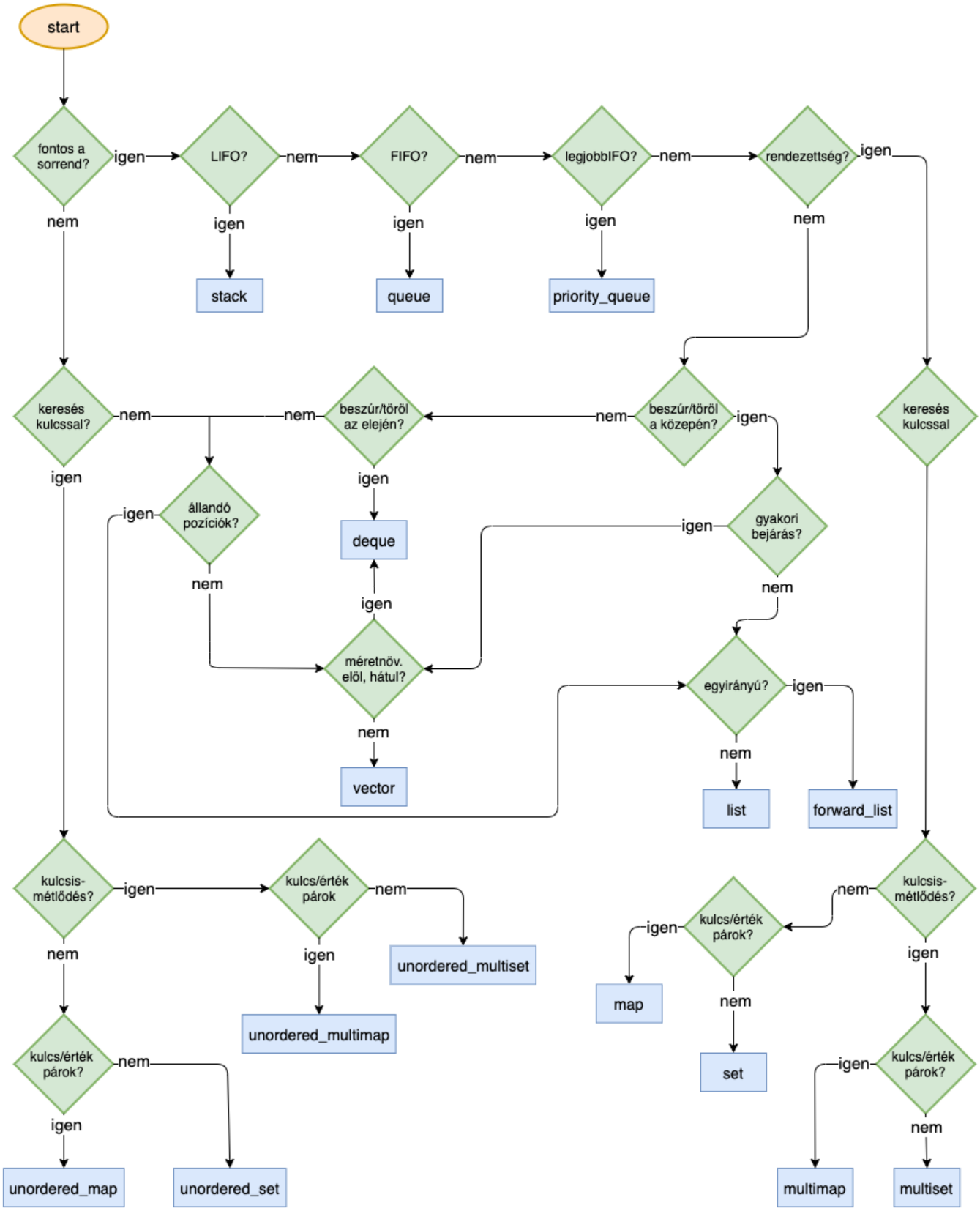
6 50

6 10

`unordered_set`, `unordered_multiset`, `unordered_map`, `unordered_multimap` - jó hash-függvény esetén egy rendezetlen konténerben a keresés konstans idejű (*O(1)*). A beszúrás szintúgy. **unordered_set** és **unordered_multiset** esetén az adatok étéke nem változhat. **unordered_map** és **unordered_multimap** esetén a kulcsok értéke nem változhat. A használatukhoz az `<unordered_set>` és az `<unordered_map>` fejlécfájl használata szükséges.

Az alábbi összehasonlítás megállja a helyét mind a négy, rendezett, illetve nem rendezett asszociatív konténer esetén:

- egy rendezett konténer kevesebb memóriát foglal ugyanannyi tárolt elem esetén,
- kevés elem esetén a keresés gyorsabb lehet a rendezett tárolókban,
- a műveletek többsége gyorsabb a rendezetlen asszociatív konténerekkel,
- a rendezetlen konténerek nem definiálják a lexikografikus összehasonlítás műveleteit: `<`, `<=`, `>` és `>=`.



Bejárók - Iterátorok

Az iterátorok elkülönítik egymástól a konténerelemekhez való hozzáférés módját a konténer fajtájától. Ezzel a megoldással lehetővé vált olyan általánosított algoritmusok készítése, amelyek függetlenek a konténerek eltérő elemhozzáférési megoldásaitól (`push_back()`, `insert()`). Az iterátor egy pozíciót határoz meg a tárolóban. Használatához az `<iterator>` fejállomány szükséges. Az iterátorokra tekinthetünk úgy, mint mutatókra, amik egy konténer adott elemére mutatnak.

Megjegyzés: az iterátorosztályokat ellátták a hagyományos mutatók operátoraival, az iterátorokat paraméterként fogadó algoritmus függvénysablonok többsége C tömbökkel is működik.

Mivel a konténerek működése lényeges eltéréseket mutat, egyetlen általános iterátor helyett a szabvány négy egymásra épülő és egy ötödik, különálló iterátort vezetett be. A különböző algoritmusok is más-más kategóriájú iterátor(oka)t várnak paraméterként.

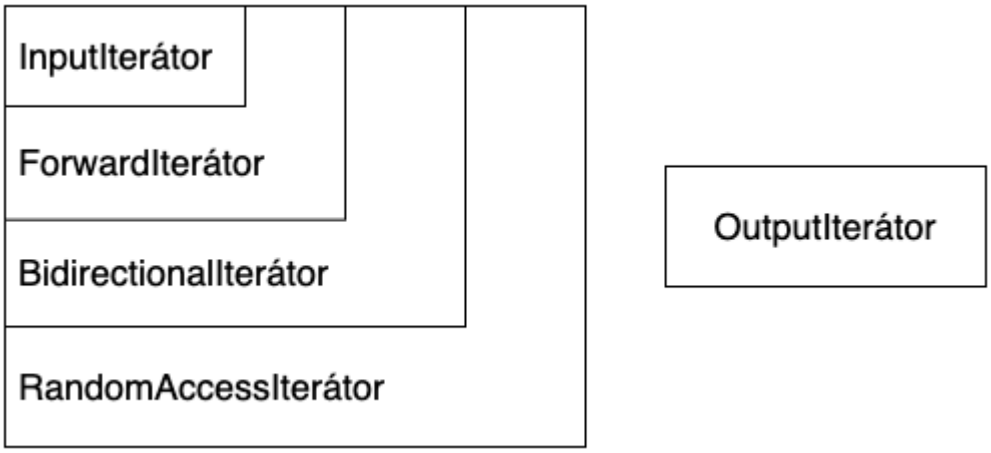
Legyenek `p` és `q` iterátorok és `n` pedig egy nemnegatív egész szám.

- a `*p` kifejezés megadja a konténer `p` által jelölt pozícióján álló elemet. Amennyiben az elem egy objektum, akkor annak tagjaira a `(*p).memberName` vagy a `p->memberName` formában hivatkozhatunk.
- a `p[n]` kifejezés megadja a konténer `p + n` kifejezés által kijelölt pozícióján álló elemet - megegyezik a `(*p+n)` kifejezéssel, ami a pointer aritmetikából ismerős lehet.
- a `p++`, illetve `p--` kifejezések hatására a `p` iterátor az aktuális pozíciót követő, illetve megelőző elemre lép. (Prefixes alakban is használható `++p`, `--p`.)
- a `p == q` és a `p != q` kifejezések segítségével ellenőrizhetjük, hogy `p` és `q` iterátorok a tárolón belül ugyanarra az elemre hivatkoznak-e vagy sem.
- a `p < q`, `p <= q`, `p > q` és a `p >= q` kifejezések segítségével ellenőrizhetjük, hogy a tárolón belül `p` által mutatott elem megelőzi-e a `q` által mutatott elemet, illetve fordítva.
- a `p + n`, `p - n`, `p += n` és a `p -= n` kifejezésekkel a `p` által mutatott elemhez képest `n` pozícióval távolabb álló elemre hivatkozhatunk, előre (`+`, `+=`), illetve visszafelé (`-`, `-=`).
- a `q - p` kifejezés megadja a `q` és `p` iterátorok által mutatott elemek pozíciókban mért távolságát.

Iterátorok kategorizálása

Mint azt korábban olvashattuk a szabvány az iterátorokat 4 + 1 kategóriába sorolja, mégpedig:

1. **Input Iterátor**
2. **Forward Iterátor**
3. **Bidirectional Iterátor**
4. **RandomAccess Iterátor**
5. **Output Iterátor**



Megjegyzés: a **forward iterátor**-tól kezdve, minden kategória helyettesítheti a megelőző kategóriákat.

Iterátor kategória	Leírás	Műveletek	Alkalmazási terület
output	írás a konténerbe, előre haladva	*p=, ++	ostream
input	olvasás a konténerből, előre haladva	=*p, ->, ++, ==, !=	istream
forward	írás és olvasás előre haladva	=*p, ->, ++, ==, !=, *p=	forward_list unordered_set unordered_multiset unordered_map unordered_multimap
bidirectional	írás és olvasás előre vagy visszafelé haladva	=*p, ->, ++, ==, !=, *p=, --	list, set, multiset, map, multimap
random-access	írás és olvasás előre vagy visszafelé haladva, illetve indexelve is	=*p, ->, ++, ==, !=, *p=, --, [], +, -, +=, -=, <, >, <=, >=	vector, deque, array

Input Iterátor

A legegyszerűbb iterátor, amely csak a konténerek olvasására szolgál. A bemeneti iterátorral csak az `istream_iterator` osztálysablon tér vissza. A bemeneti iterátor tetszőleges más iterátorral helyettesíthető, kivéve az **output** iterátort. Az alábbi példában három, szóközzel tagolt számot olvasunk be:

```
#include <iostream>
#include <iterator>

int main()
{
  double data[3] = {0};
  std::cout << "Adj meg három számot: ";

  std::istream_iterator<double> pReader(std::cin);
  for (int i = 0; i < 3; ++i)
  {
    data[i] = *pReader;
    if (i < 2) pReader++;
  }

  for (int elem : data)
  {
    std::cout << elem << " ";
```

```
  }
}
```

> kimenet: Adj meg három számot: 1 2 3
>
> 1 2 3

Output Iterátor

A kimeneti iterátorral mindenütt találkozhatunk, ahol valamilyen adatfeldolgozás folyik az STL eszközeivel, pl. másolás vagy összefűzés. Output iterátort a kimeneti adatfolyam iterátor adapterek (`ostream_iterator`) és a beszúró iterátor adapterek (`insterter`, `front_inserter`, `back_inserter`) szolgálnak. A kimeneti adatfolyam iterátorra való másolás az adatok kiírását jelenti:

```
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> data = {1, 2, 3, 4, 5, 6, 7};
  std::copy(data.begin(), data.end(), std::ostream_iterator<int>
(std::cout, "\t"));
}
```

> kimenet: 1 2 3 4 5 6 7

Forward Iterátor

Amennyiben egyesítjük a bemeneti és a kimeneti iterátorokat, megkapjuk az előrehaladó iterátort, amellyel a konténerben tárolt adatokon csak előre irányban lépkedhetünk. Az előrehaladó iterátor műveleteivel minden további nélkül készíthetünk elemeket új értékkel helyettesítő függvénysablont:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

template<typename FwdIter, typename Type>
void swap(FwdIter first, FwdIter last, const Type& old_t, const Type&
new_t)
{
  while (first != last)
  {
    if (*first == old_t)
    {
      *first = new_t;
```

```
    }
    ++first;
  }
}

int main()
{
  std::vector<int> data = {1, 2, 3, 12, 23, 34};
  swap(data.begin(), data.end(), 2, 22);
  swap(data.begin(), data.end(), 1, 111);
  std::copy(data.begin(), data.end(), std::ostream_iterator<int>
(std::cout, "\t"));
}
```

> kimenet: 111 22 12 23 34

Bidirectioanl Iterátor

A bidirectional iterátorral a konténerben tárolt adatokon előre és visszafelé is lépkedhetünk. Több algoritmus is kétirányú iterátorokat vár paraméterként, mint például az adatok sorrendjét megfordító `reverse()` algoritmus.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main()
{
  std::vector<int> data = {1, 2, 3, 12, 23, 34};
  std::reverse(data.begin(), data.end());
  std::copy(data.begin(), data.end(), std::ostream_iterator<int>
(std::cout, "\t"));
}
```

> kimenet: 34 23 12 3 2 1

Random-Access Iterátor

A random-access iterátorok lehetőségei teljes egészében megegyeznek a normál mutatókéval. A `vector` és `deque` tárolókon túlmenően a C-s tömbök esetén is ilyen iterátorokat használhatunk. Az alábbi példa programban egy függvénysablont készítünk a random-access iterátorokkal kijelölt tartomány elemeinek véletlenszerű átrendezésére. (Az elempárok cseréjét az `iter_swap()` algoritmussal végezzük.)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
```

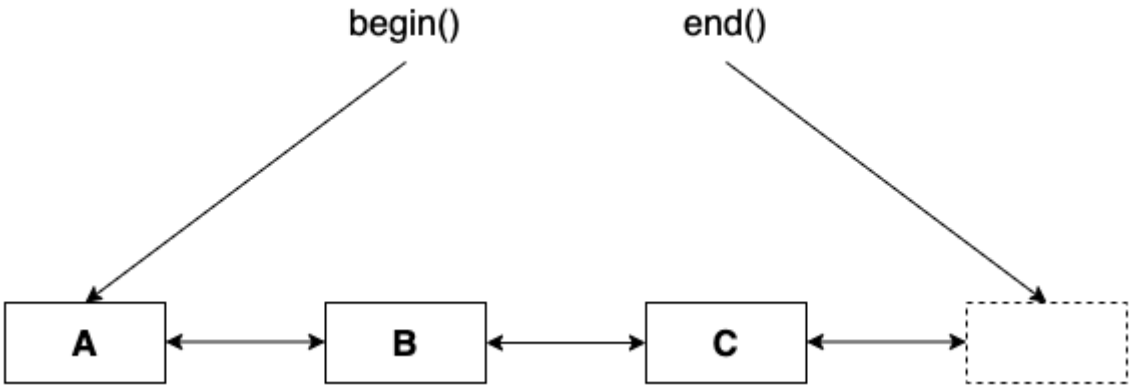
```
#include <ctime>
#include <iterator>

template<typename RandIter>
void swap(RandIter first, RandIter last)
{
  while (first < last)
  {
    std::iter_swap(first, first + std::rand() % (last-first));
    ++first;
  }
}

int main()
{
  std::vector<int> data = {1, 2, 3, 12, 23, 34};
  std::srand(unsigned(time(nullptr)));
  swap(data.begin(), data.end());
  std::copy(data.begin(), data.end(), std::ostream_iterator<int>
(std::cout, "\t"));
}
```

> kimenet: 34 23 1 3 12 2

Megjegyzés: a fentebb látott kódrészletekben használt `data.begin()` és `data.end()` helyett használható a `std::begin(data)` és `std::end(data)` alak is.

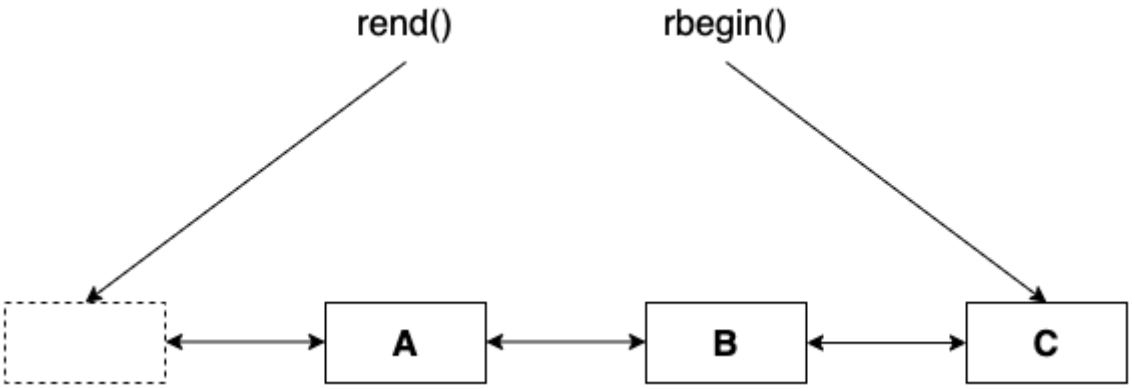
Iterátorok konténerekben tárolt elemekhez

Normál és konstans iterátorokkal térnek vissza a konténerek `begin()` és `cbegin()` tagfüggvényei, míg az utolsó elem utáni pozícióra hivatkoznak az `end()` és `cend()` tagfüggvények. A bejáráshoz előre kell léptetnünk (**++**) a `begin()` és `cbegin()` tagok hívásával megkapott iterátorokat. A függvények által visszaadott iterátor és konstans iterátor típusú bejárók kategóriáját a konténer fajtája határozza meg.



Az `array`, `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` konténerek esetén fordított irányú bejárást is lehetővé tesznek az `rbegin()`, `crbegin()`, illetve az `rend()`, `crend()` tagfüggvények által visszaadott iterátorok. Ezek a függvények `reverse_iterator`, illetve `const_reverse_iterator` típusú

értékkel térnek vissza. A bejáráshoz ebben az esetben előre kell léptetnünk (**++**) az `rbegin()`, `crbegin()` tagok hívásával megkapott iterátorokat.



STL Algoritmusok

Az STL algoritmusok az `<algorithm>` könyvtárban találhatóak, és számos jól ismert fontos függvényt foglalnak magukba, mint pl. adott tulajdonságú elem keresése, törlése stb. Numerikus algorismusok esetén a `<numeric>` deklarációs fájlra van szükség.

Az egyik legnagyobb erősségük, hogy nagyon jól olvasható kódot eredményez a használatuk. Nagyon hasznosak és megbízhatóvá tehetik a C++ programok fejlesztését. Az algoritmusok egy részét arra tervezték, hogy módosítsák egy kijelölt adatsor elemeit, azonban soha nem változtatják meg magukat az adatokat tároló konténereket.

Az algoritmusok nem tagfüggvényei a konténereknek, globális függvénysablonok, amelyek iterátorok segítségével férnek hozzá a konténerben lévő adatokhoz. Az algoritmusok teljesen függetlenek a konténerektől, a paraméterként megkapott iterátorok feladata a konténerek ismerete.

Az algoritmusok közötti eligazodásban segít, ha a különböző műveleteket a viselkedésük és működésük alapján csoportokba soroljuk. Egy lehetséges kategorizálás - ahol egy algortimus akár több csoportban is megjelenhet:

Nem módosító algoritmusok:

Ezek az algoritmusok nem változtatják meg sem az adatelemeket, sem pedig azok tárolási sorrendjét.

- `adjacent_find()`
- `find_first_of()`
- `max_element()`
- `min_element()`
- `for_each()`
- `all_of()` `any_of()` `none_of()`
- `count()` `count_if()`
- `lexicographical_compare()`
- `minmax_element()`
- `equal()`
- `max()`
- `min()`

- `mismatch()`
- `search()`
- `find()` `find_if()` `find_if_not()`
- `find_end()`
- `minmax()`
- `search_n()`

Módosító algoritmusok

Az adatmódosító algoritmusokat arra tervezték, hogy megváltoztassák a konténerekben tárolt adatelemek értékét. Ez megtörténhet közvetlenül, magában a konténerben, vagy pedig az elemek más konténerbe való másolásával. Néhány algoritmus csupán az elemek sorrendjét modosítja, és ezért került ide.

- `copy()` `copy_if()`
- `copy_backward()`
- `copy_n()`
- `fill()`
- `fill_n()`
- `for_each()`
- `generate()`
- `generate_n()`
- `iter_swap()`
- `merge()`
- `move()`
- `move_backward()`
- `replace()` `replace_if()`
- `replace_copy()` `replace_copy_if()`
- `swap()`
- `swap_ranges()`
- `transform()`

Eltávolító algoritmusok

Ezek valójában módosító algoritmusok, azonban céljuk az elemek eltávolítása egy konténerből, vagy másolása egy másik tárolóba.

- `remove()` `remove_if`
- `remove_copy()` `remove_copy_if()`
- `unique()`
- `unique_copy()`

Átalakító algoritmusok

Ezek is módosító algoritmusok, azonban kimondottan az elemsorrend megváltoztatásával jár a működésük.

- `is_partitioned()`
- `is_permutation()`
- `next_permutation()`

- `partition()`
- `partition_copy()`
- `partition_point()`
- `prev_permutation()`
- `random_shuffle()` `shuffle()`
- `reverse()`
- `reverse_copy()`
- `rotate()`
- `rotate_copy()`
- `stable_partition()`

Rendező algoritmusok

Az itt található módosító algoritmusok feladata a teljes konténerben, vagy tároló egy tartományában található elemek rendezése.

- `is_heap()`
- `is_heap_until()`
- `is_partitioned()`
- `is_sorted()`
- `is_sorted_until()`
- `make_heap()`
- `nth_element()`
- `partial_sort()`
- `partial_sort_copy()`
- `partition()`
- `partition_copy()`
- `pop_heap()`
- `push_heap()`
- `sort()`
- `sort_heap()`
- `stable_partition()`
- `stable_sort()`

Rendezett tartomány algoritmusok

Ezek az algoritmusok az elemek rendezettségét kihasználva igen hatékonyan működnek.

- `binary_search()`
- `equal_range()`
- `includes()`
- `inplace_merge()`
- `lower_bound()`
- `merge()`
- `set_differences()`
- `set_intersection()`
- `set_symmetric_difference()`
- `set_union()`

- `upper_bound()`

Numerikus algoritmusok

Számokat tároló konténerek elemein műveleteket végző algoritmusok csoportja.

- `accumulate()`
- `adjancent_difference()`
- `inner_product()`
- `iota()`
- `partial_sum()`

Néhány tároló rendelkezik az algoritmusok némelyikével megegyező nevű tagfüggvénnyel. Ezek létezésnek oka, hogy kihasználva a konténerek speciális adottságait, hatékonyabb és biztonságosabb tagfüggvény készíthető, mint az általános algoritmus. Egyetemes szabályként megfogalmazható, hogy részesítsük előnyben a taggfüggvényeket a program készítése során.

Az algoritmusok végrehajtási ideje

A konténerműveletek időigénye mellett a felhasznált algoritmusok időigénye együtt határozza meg az adott programrész futásidejét. Az algoritmusok végrehajtásához szükséges időigényt a feldolgozandó adatsor elemeinek számával (*n*) jellemezhetjük:

- **O(1)** - `swap()`, `iter_swap()`
- **O(log(n))** - `lower_bound()`, `upper_bund()`, `equal_range()`, `binary_search()`, `push_heap()`, `pop_heap()`
- **O(n log(n))** - `inplace_merge()` (legrosszabb esetben), `stable_partition()` (legrosszabb esetben),
- **O(n^2)** - `find_end()`, `find_first_of()`, `search()`, `search_n()`
- **O(n)** - minden más algoritmus

Lambda kifejezések

A lambda függvények lehetővé teszik, hogy egy vagy több soros névtelen függvényeket definiáljunk a forráskódban, ott ahol éppen szükség van rájuk. A lambda kifejezések szerkezete nem kötött, a fordító feltételezésekkel él a hiányzó részekkel kapcsolatban. Nézzünk erre egy példát:

```
int a = []{ return 12 * 23; } ();
```

A bevezető szögletes zárójel jelzi, hogy lambda következik. Ez után áll a függvény törzse, ahol a `return` utasításból a fordító meghatározza a függvény értékét és típusát. Az utasítást záró kerek zárójelpár a függvényhívást jelenti.

Amennyiben paraméterezni kívánjuk a lambdát, a szögletes és a kapcsos zárójelek közé egy hagyományos paraméterlista is beékelődik:

```
int a = [](int x, int y){ return x * y; } (12, 23);
```

Szükség esetén a függvény visszatérési típusát is megadhatjuk a C++11-ben bevezetett formában:

```
int a = [](int x, int y) -> int { return x * y; } (12, 23);
```

A lambda függvények legfontosabb alkalmazási területe az STL algoritmusok hívása.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
  std::vector<int> data = {1, 2, 3, 5, 8, 13, 21};
  int quantity = std::count_if(data.begin(), data.end(), [] (int x) {
return x % 2; });

  std::cout << "Quantity: " << quantity << std::endl;

  std::for_each(data.begin(), data.end(), [](int& e){ e *= 2; });

  std::sort(data.begin(), data.end(), [](int e1, int e2){ return e1 > e2;
});

  std::for_each(data.begin(), data.end(), [](int x){ std::cout << x << '
'; });
}
```

Első lépésként megszámoljuk a `data` vektor páratlan elemeit, majd minden elemet a duplájára növelünk, csökkenő sorrendben rendezzük a vektort, végül pedig megjelenítjük az elemeket. Ezekben a példákban a lambda függvények csak a paramétereken keresztül tartották a kapcsolatot a környezetükben elérhető változókkal. Ellentétben a hagyományos függvényekkel, a lambda kifejezésekben elérhetjük a lokális hatókör változóit.

A fájl szintű és lokális statikus élettartamú nevek elelérése minden további nélkül működik:

```
const double PI = 3.1415;

int main()
{
  static int b = 11;
  double x = [](){ return PI * b; } ();
}
```

A lokális, nem statikus függvényváltozók, illetve az osztály adattagjai esetén intézkedhetünk az elérés módjáról. Amennyiben a fenti példában `PI` és `b` lokális változók, a velük azonos hatókörben megadott

lambda a következőképpen módosul:

```
const double PI = 3.1415;
int b = 11;
double x = [PI, b](){ return PI * b; } ();
```

A lambdát és az elért változókat együtt szokás **closure**-nek nevezni, míg a felhasznált változókat, mint elkapott vagy **captured** változókra hivatkozhatunk. A változókat érték, illetve referencia szerint is elkaphatjuk.

- `[]` egyetlen helyi változót sem kívánunk elkapni
- `[=]` az összes helyi változót érték szerint kapjuk el
- `[&]` az összes helyi változót referencia szerint kapjuk el
- `[a, b]` csak az `a` és `b` változókat kapjuk el érték szerint
- `[a, &b]` az `a` változót érték, míg `b`-t referencia szerint kapjuk el
- `[=, &b]` az összes helyi változót érték szerint kapjuk el, kivéve `b`-t, őt referenciával
- `[&, a]` az összes helyi változót referenciával kapjuk el, kivéve `a`-t, amelyet érték szerint
- `[this]` osztályon belül definiált lambda kifejezésekben használhatjuk a `this` mutatót, vagyis elérhetjük az osztály adattagjait

Megjegyzés: C++17 óta `[*this]` is lehetőségünk van elkapni.

A fordító nem engedi az érték szerint elkapott változó módosítását.

Amennyiben a lambda függvényt többször szeretnénk használni, hozzárendelhetjük egy függvény mutatóhoz.

```
void (*myLambda) (int) = [](int i){ i *= i; };

auto myLambda = [](int i){ i *= i; }; // C++11 auto kulcsszó

std::for_each(data.begin(), data.end(), myLambda);
```

Nézzünk egy példát, ahol egy vektor elemeit négyzetre emeljük először egy funktor segítségével, majd egy lambda segítségével:

```
#include <iostream>
#include <vector>
#include <algorithm>

struct Square
{
  void operator()(int& i) const { i *= i; }
};

int main()
{
```

```
  Square sq;
  std::vector<int> data{1, 2, 3, 4, 5, 6}; // C++11-es vagy annál újabb
fordító szükséges

  std::for_each(data.begin(), data.end(), sq); // funktorral

  std::for_each(data.begin(), data.end(), [](int& i){ i *= i; }); //
lambdával
}
```

Sablon - template

A C++ sablonok alatt olyan **osztálysablonokat** (*class template*) és **függvénysablonokat** (*function template*) értünk, melyek esetében az adott osztály, illetve függvény definiálásakor bizonyos elemeket nem adunk meg, hanem **paraméterként** kezelünk. Ezen paraméterek megadása explicit vagy implicit módon az adott osztálysablon, illetve függvénysablon **felhasználásakor** történik. Egy osztályt osztálysablonná alakítva az elemek típusát paraméterként kezelhetjük, és a kívánt elemtípust az osztálysablon felhasználása során adhatjuk meg.

Gondolhatunk a sablonokra úgy, mint olyan osztályokra, illetve függvényekre, melyek nem teljesek, és a felhasználásukkor a paraméterek megadásával válnak teljes értékű osztályokká vagy függvényekké.

A C++ sablonok tulajdonképpen a generikus típusok a C++ nyelvbeli megfelelői. Jellemző alkalmazási területük olyan tárolóosztályok létrehozása, amelyek tetszőleges típusú elem tárolására szolgálnak, mégpedig típusbiztos (*type safe*) módon. A fordító egészen addig ignorálja a sablonokat - nem fordítja bele az assembly kódba - , amíg azok a kód valamely részén felhasználásra nem kerülnek.

Függvénysablon - function template

Legyen a feladat két függvény megírása: az első két egész szám (`int`), a második két lebegőpontos szám (`double`) közül a nagyobbat adja vissza.

```
inline int max(int lhs_, int rhs_)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}

inline double max(double lhs_, double rhs_)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}
```

Jól látható, hogy a két függvény törzse pontosan megegyezik. Ha újabb típusokra szeretnénk használni a `max()` függvényünket, akkor azokra is külön meg kellene írni. Ezen kódduplikálás elkerülésére eddigi ismereteink szerint egyetlen megoldás kínálkozik, a makrók használata. (kerülendő!). A C++ nyelvben a probléma hatékony és biztonságos megoldására a **függvénysablonokat** használhatjuk. A `max()` függvényre vonatkozóan ez azt jelenti, hogy olyan függvénysablonná alakítjuk át, amelyben a típus, amelyen

dolgozik, nincs rögzítve, paraméterként kezeljük, és ezt a paramétert a függvénysablon felhasználsakor adjuk meg. A `max()` függvénysablon implementációja a következő:

```
template <typename T>
inline T max(T lhs_, T rhs_)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}
```

A függvény átalakítását a következő lépésekben végezzük:

- a függvénysablon fejlécét a `template` kulcsszóval kell kezdeni. Ezt követően `< >` között kell felsorolni a sablonparamétereket, vesszővel elválasztva. A példában egy sablonparaméter szerepel, `T`. A sablonparamétereknek tetszőleges nevet adhatunk. Példánkban `T` egy típust jelöl, ezt a `typename` kulcsszóval jelezzük a fordító felé. A `typename` helyett a `class` kulcsszó is használható. Sablonok esetében a kettő teljesen ekvivalens. A `class` kulcsszó némiképp megtévesztő lehet, az adott paraméter nem csak osztály, hanem tetszőleges típus lehet. Sablon paraméter konstans is lehet.
- mindenhol, ahol eddig az `int` vagy `double` beégetett típust használtuk, a `T` paramétert szerepeltetjük. Ez esetünkben a függvény visszatérési értékének és bemeneti paramétereinek típusát jelenti. Használhatnánk lokálisan definiált vagy dinamikusan lefoglalt változók létrehozására is.

Megjegyzés: a `typename` kulcsszó később került be a nyelvbe, mint a `class`, ezért eleinte azt használták sablonparaméterek megadásánál.

A példában a függvény, illetve a függvénysablon a függvénytörzs rövidsége miatt célszerűen `inline`, de természetesen nem `inline` függvénysablonok írására is van lehetőség.

Eddig csak arról esett szó, hogyan lehet függvénysablont **létrehozni**. A következőkben arra lesz példa, hogyan lehet a függvénysablonokat **felhasználni**. A legegyszerűbb mód a **függvénysablon implicit példányosítása** (*implicit function template instantiation*):

```
int i = max(3, 5);
double d = max(2.3, 4.2);
```

A kódrészlet első sora `int`, a második sora `double` típussal **példányosítja** a függvénysablont. Ennek semmi köze az objektumok példányosításához. Ez azt jelenti, hogy **fordítás során** az első esetben a `T` paraméter helyébe `int`, a második esetben `double` típus helyettesítődik. A behelyettesítést követően a függvénysablonból közönséges függvény készül, melyet a fordító gépi kódra fordít. Jelen esetben azért beszélünk implicit példányosításról, mert a fordító a paraméterek típusából kikövetkezteti, milyen típust kell behelyettesíteni a sablonparaméterek helyére, vagyis a paramétereket nem adtuk meg explicit módon. Ezt a fajta kikövetkeztetést nevezzük **template argumentum dedukciónak** (*template argument deduction*).

Implicit példányosítás esetén egy adott sablonparaméter csak egy típust jelölhet. A példára vonatkozóan ez azt jelenti, hogy a fordító kikényszeríti, hogy a `max()` függvény mindkét paraméterének típusa pontosan megegyezzen. Ennek megfelelően például a

```
double d = max(1, 3.5);
```

kód nem fordul le, mert az első függvényparaméter esetében a `T` `int`, a második esetében `double` típust jelentene. A fordító a függvénysablon implicit példányosítása esetén nem engedélyezi az automatikus konverziót. Természetesen ugyanígy fordítási hibát eredményezne, ha a függvényparaméterek nem konstansok, hanem változók lennének. Sőt, az `unsigned` és a `signed` típusok közötti konverzió sem megengedett. A probléma megoldása a következő lehet:

```
double d = max(1.0, 3.5);
```

Itt mindkét paraméter típusa `double`. Változókkal szemléltetve:

```
int i;
double d;
...
double d = max((double)i, d);
```

A másik megoldást a konverziós problémára az jelenti, ha a `max()` függvénysablont **explicit módon példányosítjuk** (*explicit function template instantiation*):

```
double d = max<double>(1, 3.5);
```

Ebben az esetben a függvénysablon használatkor a függvénysablon neve utána `< >` között, vesszővel elválasztva felsoroljuk a paraméterekbehelyettesítési értékét. A példában ez a `T` helyére `double` megadását jelenti. Az expliciten példányosított függvények esetében a paraméterekre működik az automatikus konverzió, vagyis a `max<double>(1, 3.5)` nem okoz fordítási hibát: az első `int` paraméter automatikusan `double`-ra konvertálódik. Ez logikus következménye annak, hogy az explicit módon példányosított függvénysablonok ugyanúgy viselkednek, mint a közönséges függvények, ennek megfelelően pontosan ugyanazok a konverziós szabályok érvényesek rájuk is.

Felmerülhet a kérdés, hogy a `max()` függvénysablonunk ténylegesen bármilyen típussal használható-e? Vessünk egy pillantást a függvénysablonunk implementációjára, ami magában rejti a választ.

```
template <typename T>
inline T max(T lhs_, T rhs_)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}
```

Ebben az implementációban két megkötés él a `T` sablonparaméterre vonatkozóan:

1. a függvény törzsében használhatjuk rá a `>` operátort, így a `T` típusra értelmezettnek kell lennie, és az általunk elvárt módon kell megvalósítania az összehasonlítást. Így ha `T` egy saját osztály vagy stuktúra, akkor gondoskodnunk kell a `>` operátor megfelelő megírásáról.
2. A függvény az `lhs_` és `rhs_` paramétereket érték szerint veszi át, valamint nem referencia típussal tér vissza, így a függvény hívásakor meghívódik a `T` másolókonstruktora. Ha `T` saját típus, gondoskodjunk megfelelő másolókonstruktor megírásáról, amennyiben a beépített másolókonstruktor nem biztosít megfelelő működést.

Nagyobb objektumok esetében a `max()` függvénysablon hatékonyabb működést eredményez, ha referencia szerint történik a paraméterátadás:

```
template <typename T>
inline const T& max(const T& lhs_, const T& rhs_)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}
```

Függvénysablon-specializáció

Mint azt az előzőekben láthattuk a `max()` függvénysablon teljesen jól működött egész és lebegőpontos számok esetében, de mi a helyzet más típusokkal? Tekintsük az alábbi kódrészletet:

```
const char* text = max("abc", "bcd");
```

Ez a program hibás, ugyanis a `max()` függvény két string összehasonlítására nem alkalmas. Ennek oka, hogy két `char*` mutató közül a nagyobbikat adja vissza, vagyis a stringeket nem a tartalmazott karakterek, hanem a memóriabeli elhelyezkedésük alapján hasonlítja össze. Szerencsére van megoldás: **függvénysablon-specializációt** (*function template specialization*) kell alkalmazni. Ennek a lényege, hogy az általános függvénysablon definíción kívül bizonyos paraméterek behelyettesítésére az általánostól eltérő implementációt adunk meg. A példánkra vonatkozóan ez azt jelenti, hogy megírjuk ugyan a fenti általános `max()` függvénysablont, de emelett meg kell írni egy `const char*`-ra specializált implementációt is. Ez szintaktikailag két módon is megtehető. Az első a "klasszikus" megoldás:

```
const char* max(const char* lhs_, const char* rhs_)
{
  return (std::strcmp(lhs_, rhs_) > 0) ? lhs_ : rhs_;
}
```

`strcmp()` - beépített függvény a `<string.h>` fejállományban, amely paraméterül vár két `string` (vagy `char*`) típusú argumentumot, majd lexikografikusan összehasonlítja őket.

A második a C++ gondolkodásmódjának jobban megfelelő megoldás:

```
template<>
const char* max<const char*>(const char* lhs_, const char* rhs_)
{
  return (std::strcmp(lhs_, rhs_) > 0) lhs_ : rhs_;
}
```

Ez esetben a következő szabályok érvényesek:

- a függvénysablon elé a `template<>` kifejezést kell írni, üres paraméterlistával.
- a függvénysablon neve után `< >` között fel kell sorolni valamennyi sablonparaméterre vonatkozóan a specializált behelyettesítési értéket, vesszővel elválasztva. A példánkra vonatkozóan ez a `const char*`.
- egy függvénysablonhoz több specializáció is készíthető, természetesen különböző paraméterbehelyettesítésekkel.
- a fordító a függvénysablon felhasználásakor egy adott specializációt alkalmaz, amennyiben a sablonparaméterek **pontosan** megegyeznek a specializációban megadottakkal. A specializációk "illesztése" során a fordító semmiféle konverziót nem alkalmaz, így például nem konstansról konstans konverziót sem.
- függvénysablonokra vonatkozóan csak teljes, valamennyi paraméterre vonatkozó specializáció írható.

Felmerülhet kérdésként, hogy miért a `const char*`-ra való specializációt írtuk meg és miért nem a `char*`-ra? A specializációk illesztésekor a fordító a sablonparaméterekre vonatkozóan szigorú típusegyezősséget alkalmaz. Az `"abc"` és `"bcd"` literálok típusa `const char*`. Így, ha a `char*`-ra specializált változatot írtuk volna meg, akkor ezen paraméterekre vonatkozóan az általános `max()` implementáció hívódott volna meg. A gondolatmenetet tovább folytatva kérdéses, hogy ha csak a `const char*`-ra specializált verziót írjuk meg, akkor a `char*` paraméterek esetében alkalmazza-e a fordító a specializált változatot. A válasz ez esetben is nem. A tanulság pedig az, hogy a specializációt `char*` és `const char*` paraméterekre is meg kell írni, másként az adott helyzetnek megfelelően meglepő eredményeket kaphatunk a `max()` függvénysablon alkalmazásakor.

Fontos: a sablonspecializáció esetén, ha az érintett paraméter mutató típusú, a konstans és nem konstans paraméterekre is írjunk specializációt.

Sablonparaméter nem csak típus lehet, hanem típusos konstans is. Nézzük meg az alábbi példát:

```
#include<iostream>

template<int N>
int square()
{
  return N * N;
}

int main()
{
  const int x = 10;
  std::cout << "square of " << x << " is " << square<x>();
}
```

> kimenet: square of 10 is 100

A `square()` függvénysablon sablonparaméterként egy `int` konstanst vár, amelynek négyzetével tér vissza. A `square()` csak konstans paraméterrel használható, hiszen a sablon kifejtése fordítási időben történik. Ez nagyban korlátozza a használhatóságát, ugyanakkor előnye, hogy négyzetre emelés már fordítási időben megtörténik, ami jelentős futás közbeni sebességnövekedést eredményezhet.

Korábban említésre került, hogy nem csak egy sablon paramétert adhatunk meg. Írjunk egy függvénysablont, ami két sablonparaméterrel rendelkezik, és összehasonlítja a sablonparaméter-típusok memóriabeli méretét: 1-el tér vissza, ha az első paraméterben megadott típus nagyobb, -1-el, ha kisebb, és 0-val, ha egyenlők.

```
#include<iostream>

template <typename T1, typename T2>
int compareTypes()
{
  if (sizeof(T1) > sizeof(T2))
  {
    return 1;
  }
  else if (sizeof(T1) <  sizeof(T2))
  {
    return -1;
  }
  else
  {
    return 0;
  }
}

int main()
{
  std::cout << "compare double to int " << compareTypes<double, int>() <<
std::endl;
  std::cout << "compare int to long " << compareTypes<int, long>() <<
std::endl;
  std::cout << "compare int to int " << compareTypes<int, int>() <<
std::endl;
}
```

> kimenet:
>
> compare double to int 1
>
> compare int to long -1
>
> compare int to int 0

Hívott függvény kiválasztása

Amennyiben egy adott függvénynek több implementációja is van, felmerül a kérdés, hogy a függvényhívás során melyik függvény fog meghívódni. A következő szabályok érvényesek a sorszámnak megfelelő prioritási sorrendben:

1. ha létezik olyan közönséges függvény, melynek paraméterei típus szerint pontosan megegyezzenek, akkor az adott függvény hívódik meg.
2. ha létezik olyan függvénysablon, melynek paraméterei típus szerint pontosan megegyeznek, akkor az adott függvény hívódik meg.
3. ha létezik közönséges függvény vagy függvénysablon, mely esetében típuskonverzióval megegyeznek a paraméterek, akkor az adott függvény hívódik meg. Az automatikus típus konverzió nem érinthet sablonparamétert. Például a `template <typename T> void myFunc(double a, T b, T c);` esetében a `b` és `c` függvényparaméterekre vonatkozóan a korábbi szabályoknak megfelelően csak akkor lehetséges az automatikus konverzió, ha a `myFunc()` függvénysablont explicit példányosítjuk, vagyis míg a `myFunc(10, 10.1, 11);` fordítási hibát okoz, mert a harmadik paraméternél automatikus `int` -> `double` konverzióra volna szükség. Az explicit példányosított `myFunc<double>(10, 10.1, 11);` hívás lefordul.

Osztálysablon - class template

A fordítónak jeleznünk kell, hogy nem közönséges osztályról, hanem osztálysablonról van szó. Ezt a korábban bemutatott függvénysablonokhoz hasonlóan a `template` kulcsszó segítségével tehetjük meg.

```
template <typename T>
class MyClass
{
public:
  MyClass(T a_);
  T getA() const { return _a; }
  void setA(T a_);
private:
  T _a
};
```

A `template` kulcsszót követően `< >` között fel kell sorolni a sablonparamétereket, vesszővel elválasztva. A paraméterekre vonatkozó szabályok megegyeznek a függvénysablonok esetében megadottakkal.

Amennyiben a tagfüggvények implementációját nem az osztálydefiníciós részben, implicit inline módon kívánjuk megadni, a következőképpen tehetjük meg:

```
// a setA() tagfüggvény definíciója
template <typename T>
void MyClass<T>::setA(T a_)
{
  _a = a_;
}
```

A definíciót a többi tagfüggvény és konstruktor esetében is hasonló módon lehet megadni. Valamennyi tagfüggvény-definíció előtt a `template` kulcsszót kell megadni, és `< >` között fel kell sorolni a template paramétereket. Ezen túlmenően az osztály neve után `< >` között vesszővel elválasztva fel kell sorolni a sablonparaméterek **nevét**. Itt a `class` vagy `typename` kulcsszókat nem írjuk ki a paraméterek elé. **A konstruktor és destruktor nevének megadására ez az átalakítás nem érvényes.**

```
// helyes
template <typename T>
MyClass<T>::MyClass(T a_): _a(a_) { }

// helytelen
MyClass<T>::MyClass<T>(T a_): _a(a_) { }
```

Míg a közönséges osztályok esetében az osztálydefiníciós rész a `.h` fejlécfájlba, a tagfüggvények definícióját a `.cpp` forrásfájlba szokás tenni, osztálysablonok esetében a tagfüggvények definícióját is a fejlécfájlba kell tenni. Ennek okait a Sablonok fordítása című rész fejti ki bővebben.

Azokon a helyeken ahol az osztálysablont metódusparaméterként, tagváltozóként vagy lokális változóként használjuk, régebbi fordítók esetében fel kell sorolni a sablonparaméterek **neveit** vesszővel elválasztva, a **class**, illetve **typename** kulcsszó nélkül. A másolókonstruktort vehetjük példánk alapjául:

```
template <typename T>
class MyClass
{
public:
  MyClass(const MyClass<T>& rhs_)
  {
    ...
  }
};
```

Ezen szabály alkalmazása modern fordítók esetében már nem kötelező. Célszerű emlékezni rá, hogy ha régebbi forráskódban látjuk.

A következőkben arra láthatunk példát, hogy hogyan tudjuk **felhasználni** az osztálysablont.

```
int main()
{
  MyClass<int> mc1;
  mc1.setA(1);

  MyClass<char> mc2;
  mc2.setA('a');
}
```

A felhasználás során az osztálysablon neve után `< >` között kell megadni a sablonparaméterek behelyettesítési értékét. Az osztálysablon felhasználására vonatkozó legfontosabb szabály a következő: **az osztálysablon paraméterek megadása után ugyanúgy használható, mint egy közönséges osztály. Másképpen fogalmazva: ahol a kódban osztálynév szerepelhet, ott szerepelhet felparaméterezett osztálysablon is.**

Az osztálysablonok felhasználásával kapcsolatban érdemes megjegyezni, hogy amennyiben egy adott módon felparaméterezett osztálysablont többször kívánunk használni, kényelmes lehet új típusként való bevezetése a `typedef`-fel:

```
typedef MyClass<int> MC;
...
MC mc;
mc.petA(1);
```

Osztályok esetében is előfordulhat, hogy a sablonparaméter típusos konstans vagy osztálysablon. Abban az estben, ha **felparaméterezett sablon**-t használunk pl. `MyClass<AnotherClass<int>>` régebbi fordítók esetén (C++11 előtt) a utolsó `>>` közé szóközt kell rakni, mert a fordító **bitshift right** operátorként ismeri fel. Modernebb fordítók esetén ez már nem jelent problémát.

Típusos konstans sablonparaméter

Mint azt korábban említettük, sablonparaméter lehet típusos konstans is. Nézzünk erre egy példát.

```
template <typename T, int Size>
class MyClass
{
public:
  MyClass(T a_, int b_): _a(a_), _b(b_) { }
private:
  T _a;
  int _b;
};

int main()
{
  MyClass<int, 100> mc;
}
```

Alternatív megoldásként használhattunk `const` kulcsszóval ellátott konstanst.

```
int main()
{
  const int size = 100;
  MyClass<int, size> mc;
}
```

Annak, hogy csak konstans adható meg elvi oka van. A sablonok a felhasználásuk során **fordítási időben fejtődnek ki**, így a sablonparaméterek behelyettesítési értékeinek már fordítási időben eldönthetőnek kell lenniük.

Alapértelmezett sablonparaméter

Akárcsak a függvénysablonok esetében, osztálysablonoknál is lehetőségünk van **alapértelmezett értéket** megadni sablonparaméternek.

```
template <typename T = int>
class MyClass { ... }

int main()
{
  MyClass<> mc;
  MyClass<char> mc2;
}
```

A `T` paraméternek az `int` alapértelmezett értéket adtuk. Így, ha az osztálysablon felhasználásakor nem adunk neki értéket, akkor az `int` értéket veszi fel. Az alapértelmezett osztálysablonok megadására ugyanazok a szabályok érvényesek, mint az alapértelmezett függvénysablonokra: jobbról balra sorban haladva kihagyás nélkül tetszőleges számú paraméternek adható alapértelmezett érték. Az alapértelmezett sablonparaméterek lehetővé teszik olyan sablonok készítését, melyek széles körben testre szabhatóak, ugyanakkor legtöbb helyzetben könnyen felhasználhatók.

Mutatók és referenciák, mint sablonparaméterek

Amennyiben az elemtípus, vagyis a `T` összetett, nagyobb méretű objektumokat eredményező típus, akkor ezeket az objektumokat referenciaként érdemes paraméterben átadni a függvénynek annak érdekében, hogy elkerüljük a paraméterátadáskori másolat készítését, s így a másolókonstruktor hívását. Ezt úgy tudjuk elérni, hogy a `T` sablonparaméternek `T&`-t adunk meg. Hasonló az helyzet mutatók esetében is, `T*`.

Tagfüggvénysablonok

Függvénysablonok nemcsak globális függvények lehetnek, hanem tagfüggvények is. Ezek közönséges osztályok vagy osztálysablonok tagfüggvényei egyaránt lehetnek.

```
class MyClass
{
public:
  template <typename T>
  void print(T val)
  {
    std::cout << val;
  }
};
```

A `print()` függvényből a fordítás során annyi különböző verzió keletkezik, ahány különböző típussal használjuk. Tagfüggvénysablon lehet konstruktor is, azonban erre van pár megkötés: a sablonkonstruktor sohasem fog másolókonstruktorként viselkedni, ezt mindig meg kell írni közönséges konstruktor formájában.

Az osztálysablonok és az öröklés

Tekinstsük az alábbi programot:

```
class ClassA { ... };

template <typename T1, typename T2>
class TemplA
{
  T1 a;
  T2 b;
};
```

A `ClassA` közönséges osztály, a `TemplA` két sablonparaméterrel rendelkező osztálysablon. **Ősosztályként minden esetben csak közönséges osztály vagy felparaméterezett osztálysablon használható.** Először nézzünk példát a `TemplA`-ból, mint ős osztálysablonból egy közönséges osztályt származtatunk:

```
class ClassB: public TemplA<int, ClassA>
{ ... };
```

Az ős osztálysablonnál a `T1` paraméter helyébe az `int` kerül, a `T2` helyébe a `ClassA` osztályt helyettesítjük. A következő esetben a `ClassA`-ból mint közönséges osztályból osztálysablont származtatunk le:

```
template <typename T1>
class TemplB: public ClassA
{
  T1 t; // T1 típusú tagváltozó
}
```

A legösszetettebb esetnek az osztálysablonból új osztálysablon leszármaztatása tekinthető. Például:

```
template <typename T3, typename T4>
class TemplC: public TemplA<int, T3>
{
  T3 t; // T3 típusú tagváltozó
}
```

A leszármaztatott `TemplC` osztálysablonnak két paramétere van: `T3` és `T4`. A szabálynak megfelelően az ős sablonosztályt itt is felparaméterezve használtuk fel. A `T1` paraméternek a fix `int` értéket adtuk meg, `T2`-nek azonban a leszármaztatott osztály `T3` paraméterét adtuk "tovább".

Osztálysablon-specializáció

Akárcsak a függvénysablonok esetében, az osztálysablonoknál is előfordulhatnak olyan esetek, amikor az általános implementáció nem megfelelő egy adott típusra.

Az osztálysablon-specializáció megírására a következő szabályok érvényesek:

- az osztálysablon neve előtt a `template` kulcsszó után `< >` között csak azokat a maradó sablonparamétereket kell felsorolni, amelyeket a továbbiakban is paraméterként kívánunk kezelni, vagyis amelyeket nem kívánunk specializálni.
- az osztálysablon neve után `< >` között fel kell sorolni valamennyi sablonparamétert: a nem specializáltakat a sablonparaméter nevének megadásával, a specializáltakat a paraméter értékével.
- egy osztálysablonhoz több specializáció is készíthető, természetesen különböző, a paraméterekre vonatkozó megkötésekkel.
- a fordító a sablon felhasználásakor mindig a lehető legspeciálisabban illeszkedő sablonverziót alkalmazza.
- a fordító az illesztés során semmiféle konverziót nem alkalmaz, így például `const`<-> nem `const` verziót sem.

Részleges sablonspecializáció-nak (*partial template specialization*) nevezzük azt, amikor csak részben kötjük meg a sablonparamétereket.

Teljes specializáció-nak (*(full) template specialization*) nevezzük, amikor minden sablonparaméterre adunk megkötést.

Sablonok fordítása

A sablonok használatával kapcsolatban alapvető fontosságú annak ismerete, hogy a **sablonok fordítási időben fejtődnek ki**. Amikor a fordító a fordítás során először "találkozik" az osztály- vagy függvénysablon egy adott példányosításával, megfelelően behelyettesíti a sablonparamétereket, és legenerálja a behelyettesítésnek megfelelő kódot. **Osztálysablonok esetében a kódgenerálás tagfüggvényenként történik**, vagyis egy tagfüggvény kódja csak akkor generálódik le, ha legalább egy helyen használjuk. Vizsgáljuk meg a korábban bevezetett `MyClass` osztálysablont.

```
template <typename T>
class MyClass
{ ... };

int main()
{
  MyClass<int> mc1;
  mc1.setA(2);
  mc1.getA();

  MyClass<double> mc2;
```

```
  mc2.setA(3.14);
}
```

Amikor a fordító a `MyClass<int> mc1;` sorhoz ér, veszi a `MyClass` osztálysablont, a `T` paraméter helyébe `int`-et helyettesít, legenerálja az alapértelmezett konstruktorának a kódját, és ennek hívására vonatkozóan generál kódot. A `mc1.setA(2);` sorhoz érve a fordító veszi a `MyClass` osztálysablont, a `T` helyére `int`-et helyettesít, legenerálja a `setA()` metódus kódját, és most ennek a hívására vonatkozóan generál kódot. A `getA()` metódus kódja a `setA()`-hoz hasonló módon generálódik. Hasonló a helyzet lép fel a `MyClass<double> mc2;` esetében is, csak `int` helyett `double` típust fog behelyettesíteni a fordító.

A sablonok ezen fordítási mechanizmusának számos kellemes és kellemetlen következménye van:

1. **Optimalizáció**: ha egy adott osztálysablon tagfüggvényét sehol sem használjuk, akkor nem fog legenerálódni a kód. Ennek következtében a generált kód mérete lényegesen kisebb lehet, mint a közönséges osztályokkal vagy függvényekkel dolgoznánk. Így alkalmazásuk a háttértáron és a memóriában is a lehető legkisebb helyet foglalja el.
2. **Kódburjánzás**: ha egy sablon - akár osztály akár függvény - több paraméterkombinációval használunk, valamennyire vonatkozóan legenerálódik a sablon tagfüggvényeinek kódja, ami a generált kód méretének növekedéséhez vezet. Ezt szokás **kódburjánzás**-nak vagy **kódfelfúvódás**-nak nevezni (*code bloat*).
3. **Fordítási hibák**: a fordítási modell talán legmeghökkentőbb következménye, hogy a **fordítási hibák rejtve maradnak**. Könnyen megeshet, hogy egy osztály- vagy függvénysablon megírását követően a fordítás során semmilyen hibát sem kapunk, és a kód mégis tele van szintaktikai és egyéb hibákkal. Valójában nem is várhatunk mást, mert a sablonok tagfüggvényei csak akkor fordulnak le, ha használjuk őket. Mindaddig nem esnek át teljes szintaktikai ellenőrzésen. Ilyen esetekben az osztálysablonok **explicit példányosítása** (*explicit template instantiation*) lehet a válasz. Az explicit példányosítás a megadott paraméterekkel valamennyi tagfüggvényre vonatkozóan kikényszeríti a kód generálást. Például `template class MyClass<int>;` Ez a sor a `MyClass` osztálysablont a `T = int` paraméter-behelyettesítéssel példányosítja.

Függvénysablonok esetén is van lehetőségünk explicit példányosításra. Ehhez az adott függvénysablont a `template` kulcsszót követően az adott paraméter-behelyettesítésekkel deklarálni kell:

```
template<typename T>
inline T max(T lhs_, T rhs,)
{
  return lhs_ > rhs_ ? lhs_ : rhs_;
}
// példányosítás T=int behelyettesítéssel
template inline int max<int>(int lhs_, int rhs_);
```

A fenti példában a `max()` függvény `inline`, így az explicit példányosítás legenerálja ugyan a kódját, a hívás helyére azonban, amennyiben lehetséges, a törzse fog behelyettesítődni.

Az explicit példányosítás jól használható arra, hogy "előcsaljuk" a fordítási hibákat, de a kód részletes tesztelését önmagában nyilvánvalóan nem oldja meg.

Az explicit példányosítást sablon osztálykönyvtárak készítésekor is felhasználhatjuk.

Ha egy sablon a paraméterek adott kombinációjával jól lefordul és működik, még nem jelenti azt, hogy más paraméterek esetén is jól fog működni.

Fejléc- és forrásfájlok

A közönséges függvények és osztályok esetében a következő elvet használjuk:

- a függvények deklarációit a fejléc- (`.h`), a definícióit a forrásfájlokba (`.cpp`) tesszük.
- az osztályok deklarációit a fejléc- (`.h`), a tagfüggvények definícióit a forrásfájlokba (`.cpp`) tesszük.

Ez az elv sablonok esetében nem használható. Tekintsük a következő kódot:

```
// MyClass.h
template <typename T>
class MyClass
{
public:
  void setA(T a_);
private:
  T _a;
};

// MyClass.cpp
#include "MyClass.h"

// a setA() tagfüggvény definíciója
template <typename T>
void MyClass<T>::setA(T a_)
{
  _a = a_;
}

// main.cpp
#include "MyClass.h"

int main()
{
  MyClass<int> mc;
  mc.setA(10);
}
```

A kód fordítási hibát okoz. Mit lát a fordító a **main.cpp** feldolgozása során? A preprocesszor kifejti az `#include "MyClass.h"`-t, és mivel ez gyakorlatilag egy egyszerű szövegszerű behelyettesítést jelent, a fordító a következő fájlon "dolgozik":

```
// main.cpp
// MyClass.h
template <typename T>
```

```
class MyClass
{
public:
  void setA(T a_);
private:
  T _a;
};

int main()
{
  MyClass<int> mc;
  mc.setA(10);
}
```

A sablonokról tudjuk, hogy **fordítási** (és **nem linkelési**) időben fejtődnek ki. A fordító a forrásfájlokat egyesével dolgozza fel, így a **main.cpp** feldolgozásakor kizárólag a fenti forráskódot látja. A sablon kifejtéséhez a teljes forráskódjának a fordító rendelkezésére kell állnia. Így az `mc.setA(10);` hívásakor ez a feltétel nem teljesül, hiszen az egy másik forrásfájlban van kifejtve. Ennek következtében a linker "*Unresolved external symbol...*" vagy valami hasonló hibaüzenetet ad. A tanulság az, hogy **az osztálysablonok esetében a felhasználás során a tagfüggvények definícióját is elérhetővé kell tenni a fordító számára**. Ezt kétféle képpen tehetjük meg. Az első megoldás szerint a tagfüggvényeket az osztálysablon-definíciójában "inline módon" adjuk meg.

```
// MyClass.h
template <typename T>
class MyClass
{
public:
  void setA(T a_)
  {
    _a = a_;
  }
private:
  T _a;
};
```

A másik megoldás szerint a tagfüggvényeket az osztálydefiníciós fejlécfájlban de nem "inline módon" adjuk meg.

```
// MyClass.h
template <typename T>
class MyClass
{
public:
  void setA(T a_);
private:
  T _a;
};
```

```
template <typename T>
void MyClass<T>::setA(T a_)
{
  _a = a_;
}
```

Megjegyzés: közönséges osztályok tagfüggvényei esetében ez utóbbi megoldás linkelési hibához vezetne a fejlécfájl többszöri `include`-olása miatt. Osztálysablonok esetében ez nem igaz, mert a linker "okosan" mindig az első definíciót teszi a lefordított kódba, és a hivatkozásokat is erre vonatkozóan oldja fel. Ugyanez igaz a globális függvénysablonokra is, a közönséges függvényekkel szemben ezek definícióját is a fejlécfájlokba tesszük.

Típuskonverziók

Az eddigiek során többször is észrevehettük, hogy a C++ típuskonverziói nagyban hasonlítanak a C nyelvre, mégis vannak különbségek. Ennek egyik oka, hogy a C++ fejlesztői már a kezdetektől fogva törekedtek rá, hogy a C-nél biztonságosabb nyelvet alkossanak. Másfelől a C++-ban megjelennek teljesen új, objektumorientált nyelvi elemek, amelyekhez új típuskonverziók tartoznak. A típustámogatás C++ alapelvét követve lehetőségünk van arra, hogy beépített automatikus típuskonverziókat írjunk. A C++ finomítja a C konverziós operátort: annak funkciója szerint négy konverziós operátort különböztet meg.

Beépített típusok közötti típuskonverzió

A C nyelvben az `enum` és az `int` típus között létezik automatikus oda-vissza implicit konverzió. Ezzel szemben C++-ban, ha `enum` típusra konvertálunk, ki kell írnunk a típuskonverziót.

```
enum days {Mon, Tue, Wed, Thu, Fri};

int main()
{
  enum days day = Mon; //*
  int d;

  // C/C++ OK
  d = Mon;

  // C OK, C++ hiba
  day = d;

  // C/C++ OK
  day = (enum days)d; //*
}
```

A C++-ban az `enum` felhasználásakor (a `*`-al jelölt sorokban) természetesen elhagyhatjuk az `enum` kulcsszót, mert az `enum` neve önmagában is típusértékű, de itt a C-vel való kompatibilitás miatt kiírtuk.

A C automatikus konverziót biztosít a `void*` típusú mutató és tetszőleges típusú mutató között oda-vissza, a C++-ban ezt a konverziót is ki kell írnunk.

A referenciája a következő szabályok érvényesek: nem konstans referenciára nincs automatikus konverzió inkompatibilis típusok referenciáiról. Tekintsük az alábbi programot:

```
void f(double& d)
{
  d = 1;
}

int main()
{
  int m = 2;
  f(m); // 1. hiba
  f((double)m) // 2. hiba
  f((double&)m) // 3. lefordul, de hibás
}
```

`int` és `double` között van automatikus típuskonverzió. Referencia esetén azonban ugyanaz a helyzet, mint a mutatóknál: mindkettő memóriacímet jelent. A mutatóhoz hasonlóan két típus kötődik hozzá: a referencia típusa, amelyet megadunk deklarációként, illetve az adott memóriaterületen található változó típusa. Ez nem meglepő hiszen referenciára is működik a polimorfizmus.

Amikor egy `int` típusú változóval inicializálunk egy referenciát, a fordító hibát jelez, hiszen az `int` memóriareprezentációja eltér a `double` memóriareprezentációjától, és az a kísérlet, hogy `int`-re jellemző memóriaterületet `double`-ként kezelünk nagy valószínűséggel programozói hiba. Nem is szólva arról, hogy a `sizeof(int)` kisebb lehet, mint a `sizeof(double)`, vagyis az `f()` függvény a memóriában az `m` után következő változók értékét is elronthatja.

A második esetben az `m` változót konvertáljuk `double` típusúra. Ennek eredménye egy ideiglenes `double` érték, amely konstans, így nem adható át paraméterként olyan függvénynek, amely nem konstans referenciát vár. Ideiglenes értéket amúgy sem túl ésszerű változtatni, hiszen változtatás után nem férünk hozzá.

A harmadik esetben bemutatott "erőszakos" típuskonverzóval fordítási időben mindig sikerrel járunk.

A felhasználói típusok konverziói

Konverzió független típusok között

Tegyük fel, hogy szeretnénk egy olyan osztályt írni, amely a C-beli lehetőségeknél könnyebbé teszi a szöveges műveleteket. A eddigiek alapján egyszerűen megírhatunk egy dinamikus karaktersorozatot tartalmazó osztályt, és az összeadás operátor megfelelő túlterhelésével lehetőséget biztosíthatunk string-ek kényelmes összefűzésére. Ugyanakkor szeretnénk, ha ez az osztály kompatibilis lenne az eddig C függvényekkel, ugyanis számos programozási felület C-ben íródott. A kompatibilitást kétféle módon szeretnénk megoldani:

1. ha egy függvény nullterminált karaktertömböt ad vissza, akkor az automatikusan konvertálható legyen az osztályunkra. Például, ha van egy függvényünk, amely ilyen osztály típusú argumentumot vár, akkor ott átadhassunk egy `char*` típusú, nullterminált, C stílusú string-et.
2. ha van egy ilyen osztályból származó objektumunk, szeretnénk, hogy átadható legyen bárhol, ahol konstans C stílusú string-et kell átadni.

Erre a két problémára a C++ két nyelvi elemet kínál. Ha egy **másik** - esetleg beépített - **típusról** szeretnénk konvertálni a mi **osztályunk típusára**, a **konverziós konstruktor** jelent megoldást. Ha az **osztályunkról** szeretnénk egy **másik típusra** konvertálni, akkor a **konverziós operátor** a megfelelő eszköz.

A konverziós konstruktor olyan egyparaméteres konstruktor, amelynek a paramétere olyan típusú, amilyen típusról konvertálni szeretnénk.

A konverziók leggyakoribb problémája, hogy bizonyos kifejezések esetén több megoldás is létezik, és a fordító nem tud választani. Ha a típuskonverziós útvonal nem egyértelmű, fordítási hibát jelent.

Útmutató:

- csak akkor írjunk konverziót, ha természetes. Sose erőltessük a konverziót.
- nagyon vigyázzunk a védett (*private*, *protected*) tagváltozók kiadására konverziós operátorok esetén, mert lehetőséget adhatunk objektumaink inkonzisztensé tételére. Ha kiadjuk ezeket a paramétereket konstansként tegyük, és írjuk elő az osztály felhasználóinak, hogy ne tárolják el a kapott értéket.
- konverziós operátor helyett használjuk a konverziós konstruktort. Az egységbe zárás alapelve miatt érdemesebb egy másik adatszerkezet alapján felépíteni az osztályt, mint kiadni az elrejtett tagváltozóit.
- mindig csak a legszükségesebb konverziót írjuk meg, mert minél több a konverzió, annál valószínűbb a kétértelműség.

Konverzió az öröklési hierarchia mentén

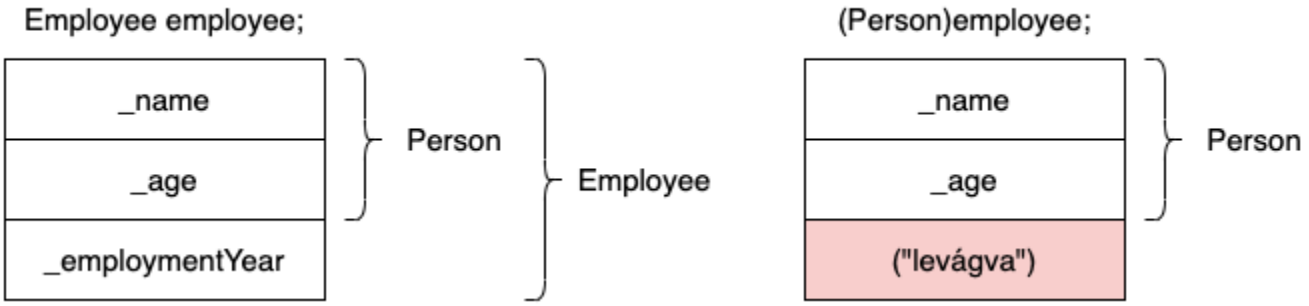
Tekintsük a leszármazottról szülőre történő típuskonverziót. Elékezzünk vissza a `Person`-`Employee` példájára.

```
Employee employee("John", 28, 3);
Person p = (Person)employee;
```

Vagy az imént említett konstruktorszintaktikával:

```
Person p = Person(employee);
```

A példa második fele jól illusztrálja, hogy itt konsruktor fog meghívódni. Felmerül a kérdés, hogy melyik konstruktor. A válasz a másolókonstruktor, hiszen a behelyettesíthetőség elve miatt a `Person&` típusú paraméter esetén megadhatunk `Employee` típusú objektumot. A másolókonstruktor viszont csak az ősosztály részével "foglalkozik", így az `employee` változó "alja" nem másolódik át, elveszik a konverzió során.



A C++ élénk fantáziájú úttörői ezt a konverzió közben történő jelenséget "szeletelés kapásból" (**slicing-on-the-fly**) névvel illették. Az eredeti, konvertált objektum megmarad, mindössze a konverzió során nem másolódik át egy rész, bár az új objektum kétségkívül úgy néz ki, mintha a réginek levágták volna az alját. Mindezt a fenti ábra szemlélteti.

Fontos: **polimorf viselkedést csak mutatón és referencián keresztül adhatunk.**

A C++ típuskonverziós operátorai

Az explicit típuskonverziót C nyelven a kifejezés elé `()` zárójelek közé írt új típus megadásával definiálhatjuk.

```
double d = (int)3.14;
```

Ezt a típuskonverziót használjuk például mutatók között, egészek között, a konstans változókat ezzel konvertáljuk nem konstanssá. A nagyobb biztonság és átláthatóság érdekében a C++ saját konverziós operátorokat definiál, amelyek jobban kifejezik a típuskonverzió jelentését. A C megoldás ugyanis egy kalap alá vesz bizonyos konverziós szándékokat. Jobb lenne, ha pontosabban meg tudnánk adni a konverzió célját. Ezeket a C++ nyelvben az alábbi operátorok segítik:

- `static_cast` (statikus típuskonverzió)
- `const_cast` (konstans típuskonverzió)
- `dynamic_cast` (dinamikus típuskonverzió)
- `reinterpret_cast` (újraértelmező típuskonverzió)

Szintaxisuk az alábbi:

```
static_cast <típusnév> (kifejezés)

const_cast <típusnév> (kifejezés)

dynamic_cast <típusnév> (kifejezés)

reinterpret_cast <típusnév> (kifejezés)
```

A C stílusú típsukonverzió helyett leggyakrabban a **statikus típuskonverziót** használjuk. Vagyis a lefele kerekítés végett eddig azt írtuk, hogy

```
double d1 = (int) d2;
```

a C++-ban ez az alábbi formát ölti:

```
double d1 = static_cast<int>(d2);
```

Ősosztály típusú mutatóról leszármazott típusú mutatóra is ezt a konverziós operátort használjuk, ha biztosak vagyunk a konverzióban. Ha nem ez a helyzet, akkor a `dynamic_cast` operátor futási időben megállapítja, hogy helyes-e a konverzió, és csak akkor hajtja végre.

A statikus típuskonverziónak megmaradtak azok a megkötései, amely a C stílusú elődjének, valamint nem konvertálhat konstans típust sem nem konstanssá, ekkor fordítási idejű hibát kapunk. Erre ugyanis egy kifejezetten erre a célra létrehozott típuskonverziós operátor áll a rendelkezésre.

A **konstans típuskonverzó** képes egyedül konstans típust nem konstanssá tenni, illetve `volatile` típust nem azzá. Ez ugyanis olyan veszélyes művelet, amelyet külön át kell gondolni, és feltűnően megjelölni a kódban. Egyéb konverziókra nem alkalmazható. Más C++ konverziós operátor nem képes végrehajtani ezt a konverziót. Példaként tekintsük meg az alábbi függvényt, amely egy third-party osztálykönyvtár része és nincs jogunkban módosítani a forráskódját.

```
void AddToStream(char* buff, unsigned len) { ... }
```

Ez a függvény egy ismeretlen formátumú állományhoz adja a változót. Mivel mentésről van szó, a `buff` változót konstansként kellett volna definiálni. Így, ha ki szeretnénk írni egy konstans tagváltozót, mert az állomány formátuma előírja, akkor típuskonverzióhoz kell folyamodnunk, amely nem konstanssá teszi az elmentendő változót. C++ nyelven erre az alábbi kódrészlet a legmegfelelőbb megoldás:

```
class Data
{
public:
  const char _c;
  Data(char c_): _c(c_) { }
  void Save()
  {
    AddToStream(const_cast<char*>(&_c), sizeof(_c));
  }
};
```

A **dinamikus típsukonverzió** szintén speciális típuskonverziót valósít meg: az öröklési hierarchián lefelé történő konverzióhoz szükséges. Az osztályoknak polimorfoknak kell lenniük, azaz a konvertálandó típusnak legalább egy virtuális függvényt kell tartalmazni. Mivel futási időben ellenőrzi, hogy tényleg végrehajtható-e a típuskonverzió, használatához a futásidejű típusinformációk kezelését be kell kapcsolnunk a fordításkor. Ezen típusinformáció segítségével a keresztbe konverziót is megoldja többszörös öröklés esetén, vagyis biztonságos. Ha a kívánt konverzió nem sikerül, akkor az operátor `bad_cast` kivételt dob. Privát öröklésnél nem használható az osztályhierarchián lefele való konverzióra, mert futási idejű hibát kapunk.

Az **újraértelmező típuskonverzió** az implementációfüggő konverziók esetén használható. Általában mutatókra alkalmazzuk, amikor lényegében csak a mutató típusát változtatjuk, vagyis azt, hogy milyen műveletek értelmezhetők egy memóriaterületen. Ezért hatása sokszor fordítófüggő. Mivel az egész típusok és a mutatók mérete fordítófüggő, ezért az egész típusok és a mutatók közötti konverziók esetén is ezt az operátort használjuk. Ugyanakkor konstans típust nem konstanssá ez az operátor sem képes átkonvertálni.

Összegezve a C++-ban explicit típuskonverzió végett használható a konstruktorszintaxisú típuskonverzió, a régi C szintaktikájú típuskonverzió, illetve az utóbbit érdemes felváltanunk a `static_cast`, `const_cast`, `reinterpret_cast` valamelyikével. A `dynamic_cast` egyedülálló: a többi típuskonverziós operátor nem képes ellátni, viszont ennek az ára, hogy futásidejű típusinformációval és annak lekérdezésével lassítjuk programunkat.

Útmutató:

- ha a konverziónak több argumentuma van, mindig a konstruktorszintaxist alkalmazzuk
- C stílusú konverzió helyett használjuk a leginkább odaillő C++ konverziós operátort
- a dinamikus típuskonverziót lehetőleg kerüljük el.

Implicit konverzió

Implicit konverzióról akkor beszélünk, ha egy adott típusú objektumnak egy másik típusú objektumot adunk értékül, és az egyik objektum a másik típusára konvertálódik anélkül, hogy mi ezt a konverziót külön (explicit módon, pl. cast-ok segítségével) „kértük” volna.

A nyelv egyik érdekessége, hogy lévén a `char` típusok tudnak számokat tárolni, így egy `char` objektum könnyedén át tud konvertálni `int` típusúra. Példaképp, az `'a' + 'b'` összeadásnál a karakterek összeadása nem definiált, azonban `int`-ek között igen, így a fordító implicit módon átkonvertálja azokat `int`-é.

Amennyiben a fordító megpróbál egy műveletet elvégezni, de típushiba miatt ez meghiúsul, megpróbálja az objektumokat átkonvertálni, lehetőleg veszteségmentesen. A fordító először implicit konverzióval próbálkozik, aztán egy felhasználó által definiált konverzióval, majd ismét egy implicit konverzióval. Amennyiben ezek után sem megvalósítható a művelet típushiba miatt, fordítási hibát kapunk.

Aritmetikai konverziók

1. ha az egyik operandus `long double`, akkor a másikat is `long double`-ra konvertáljuk.
2. különben, ha az egyik operandus `double`, akkor a másik is `dobule` lesz.
3. különben, ha az egyik `float`, akkor a másik is `float` lesz.
4. amennyiben az egyik operandus sem lebegőpontos, akkor ha az egyik operandus `unsigned long`, akkor a másik is az lesz.
5. ha az egyik operandus `long`, a másik pedig `unsigned int`, akkor előfordulhat az, hogy a két típus ugyanakkora tárterületen van tárolva. Ha az `int` és a `long` mérete megegyezik, akkor `unsigned long` lesz mindkét operandus. Különben ha a `long` nagyobb, akkor `long`-gá konvertálódik a másik.
6. különben, ha az egyik `long` a másik pedig nem `unsigned int`, akkor `long`-gá konvertálódik a másik operandus.
7. különben, ha az egyik operandus `unsigned int`, a másik is konvertálódik `unsigned int`-té.
8. különben minden `int`-té konvertálódik, ha egyik korábbi sem teljesül.

Logikai konverzió

Amikor `for` ciklusban vagy `if` ágban meg kell adnunk egy logikai értéket, akkor ott gyakran történhet implicit konverzió. Példaképp minden lebegőpontos vagy egész szám ami nullától különböző `true` értékre konvertálódik, míg a nulla érték `false`-ra. Mutatóknál minden nem `nullpointer` `true`.