4. Gyakorlat

Operátor túlterhelés (operator overloading)

A programozás során gyakran használunk operátorokat a programban elvégzendő feladatok tömör, olvasható kifejezésére.

Az operátorok beépített típusok kezelésére kitűnően használhatók, a programozó által létrehozott típusok, osztályok kezelésére azonban természetesen nem alkalmasak változatlan formában. A szabványos C++ operátorokhoz előre definiált úgynevezett operátor-függvények tartoznak. Amikor tehát egy C++-ban definiált műveleti jel értelmezését ki szeretnénk terjeszteni egy saját adattípusra, akkor a kérdéses operátor operátor-függvényére adunk meg egy újabb paraméter-szignatúrájú változatot.

Megoldást az operátorok túlterhelése jelenti. A túlterhelés segítségével a programozó meghatározhatja, hogy mi történjék az általa létrehozott típusokkal az egyes operátorok hatására.

Túlterhelés során megváltoztathatjuk az egyes operátorok jelentését, de:

- nem hozhatunk létre új operátorokat
- nem változtathatjuk meg az operátorok aritását
- nem változtathatjuk meg az operátorok precedenciáját

Túlterhelhető operátorok

`+`	`-`	`*`	`/`	`%`	`^`
`&`	`|`	`~`	`!`	`=`	`<`
`>`	`+=`	`-=`	`*=`	`/=`	`%=`
`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`
`||`	`++`	`--`	`->*`	`,`	`->`
`[]`	`()`	`new`	`new[]`	`delete`	`delete[]`

A `'='` (értékadó), `'&'` (címképző) és a `','` (kiválasztó) operátorok túlterhelés nélkül is érvényesek.

Nem túlterhelhető operátorok

::	.	.*	? :	sizeof	typeid

Ezeknek az operátoroknak a túlterhelése nemkívánatos mellékhatásokkal járna, ezért nem lehet őket túlterhelni.

Túlterhelés menete

A túlterhelés során az operátort megvalósító utasításokat függvényként adjuk meg. A függvény nevében az **operator** kulcsszót maga az operátor követi. Az operátor argumentumai a függvény argumentumai és visszatérési értéke, amelyek beépített típusok, objektumok (kis méret) vagy referenciák (nagy méret) lehetnek.

Az operátorok túlterhelésére használt függvények argumentumai lehetnek objektumok, de ez nem szerencsés nagyméretű objektumok esetében. Mutatókat nem használhatunk argumentumként, mert a mutatókra alkalmazott operátorok nem terhelhetők túl. A referenciák argumentumként használva lehetővé teszik a nagyméretű objektumok kezelését anélkül, hogy lemásolnánk őket.

Iterátorok (bejárók)

Az iterátor olyan objektum, amely képes egy adathalmaz bejárására. Az adathalmaz jelentheti valamely STL konténerben tárolt elemeket, vagy azok résztartományát. Az iterátor egy pozíciót határoz meg a tárolóban. Az iterátorok használatához az **<iterator>** fejállományt kell a programunkba meghívni. Az iterátorokra tekinthetünk úgy, mint mutatókra, amik egy konténer adott elemére mutatnak. Legyenek **p** és **q** iterátorok és **n** pedig egy nemnegatív egész szám.

- a ***p** kifejezés megadja a konténer **p** által kijelölt pozícióján álló elemet. Amennyiben az elem egy objektum, akkor annak tagjaira a **(*p).tag** vagy a **p->tag** formájában hivatkozhatunk.
- a **p[n]** kifejezés megadja a konténer **p + n** kifejezés által kijelölt pozícióján álló elemet – megegyezik a ***(p+n)**-el (pointer aritmetika)
- a **p++** illetve **p--** kifejezések hatására a **p** iterátor az aktuális pozíciót követő, illetve megelőző elemre lép. (prefixes alak 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, 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, 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 egymástól.

Az iterátorok kategorizálása

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

Fontos megjegyezni, hogy a **`forward`** iterátortól kezdve minden kategória helyettesítheti a megelőző kategóriákat. (A kategóriákhoz tartozó új műveletek piros színnel vannak jelölve.)

Input iterátor

A legegyszerűbb iterátor, amely csak a konténerek olvasására használható. A bemeneti iterátorral csak az **istream_iterator** osztálysablon tér vissza. Az input iterátor tetszőleges más iterátorral helyettesíthető, kivéve az output iterátort. Az alábbi példában 3, szóközzel tagolt egész számot olvasunk be:

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

int main()
{
  double data[3] = {0};
  std::cout << "Give 3 number: ";

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

  for(const int& elem : data)
    std::cout << elem << "\t";
}
```

Output iterátor

Az output iterátorral mindenütt találkozhatunk, ahol valamilyen adatfeldolgozás folyik az STL eszközeivel, pl. másolás vagy összefűzés algoritmusok. Output iterátort az output adatfolyam-iterátor adatpterek (**ostream_iterator**) és a beszúró iterátor adapter (**inserter, front_inserter, back_inserter**) szolgálnak. A kimeneti adatfolyam iterátorra való masolás az adatok kiírását jelenti:

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

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

Forward iterátor

Amennyiben egyesítjük a bemeneti és a kimeneti iterátorokat, megkapjuk a forward iterátort, amellyel a konténerben tárolt adatokon csak előre irányban lépkedhetünk. A forward 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"));
}
```

Bidirectional iterátor

A bidirictional 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 <algorithm>
#include <vector>
#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"));
}
```

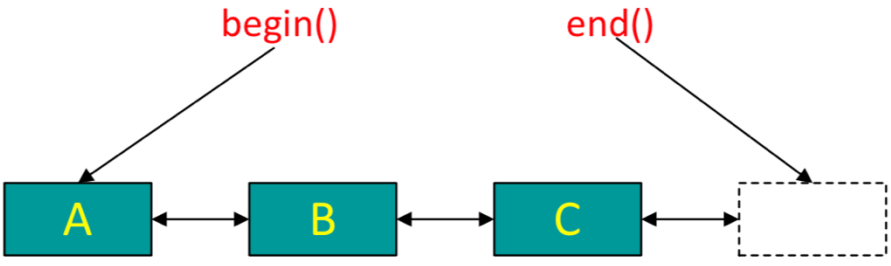
Random-access iterátor

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

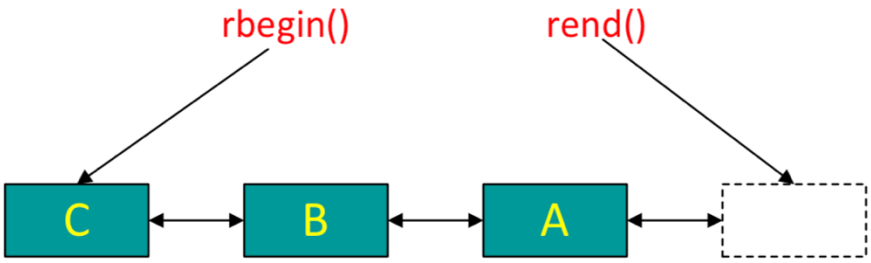
```
#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <ctime>
#include <iterator>

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"));
}
```

Iterátorok konténerekben tárolt elemekhez

Normál és konstans iterátorral 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() / cbegin()** tagok hívásával megkapott iterátorokat. A függvények által visszaadott **iterator** és **const_iterator** típusú bejárók kategóriáját a konténer fajtája határozza meg.



Az **array, vector, deque, list, set, multiset, map** és a **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 is előre kell léptetnünk (++) az **rbegin(), crbegin()** tagok hívásával megkapott iterátorokat.



Lambda kifejezések (függvények)

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. Az alábbi példában piros színnel kiemelt rész maga a lambda függvény:

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

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

Amennyiben paraméterezni kívánjuk a lambdát, a fenti két rész közé egy hagyományos paraméterlista is beékelődik:

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

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

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

A Lambda függvények legfontosabb alkalmazási területe az STL algoritmusok hívása. Az alábbi utasítások a **vector** és az **algorithm** fejállományok beépítése után működőképesek:

```
std::vector<int> data = {1, 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 = 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 << ” ”; } );
```

Először megszámláljuk 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 fájlszintű és lokális statikus élettartamú nevek elérése minden további nélkül működik:

```
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:

```
int main()
{
  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 capture-ölhetjük értékükkel, illetve referenciájukkal.

[]	egyetlen helyi változót sem kívánunk capture-ölni
[=]	az összes helyi változót érték szerint capture-ölj
[&]	az összes helyi változót referenciájával capture-öljük
[a, b]	csak az a és b változókat kapjuk el érték szerint
[a, &b]	az a változót érték, míg b-t referenciájával kapjuk el
[=, &b]	összes helyi változót érték szerint kapjuk el, kivéve b-t, őt referenciáján keresztül
[&, a]	összes helyi változót referenciájukkal kapjuk el, kivéve a-t, amelyet érték szerint
[this]	osztályon belül definiált lambda kifejezésekben használhatjuk a `this` mutatót, vagyis elérhetjük az osztály tagjait (C++17 óta capture *this)

A fordító nem engedi az érték szerint elkapott változó módosítását.

Amennyiben a lambda függvényt többször szeretnénk hívni hozzá rendelhetjük egy függvény mutatóhoz.

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

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

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

Functor

A funktorok lehetővé teszik a **()** operátor felüldefiniálását. A C++11-ben a funktorokat tudjuk kiváltani lambdákkal.

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

int main()
{
  Square sq;
  std::vector<int> v{1, 2, 3, 4, 5, 6};   // C++11-es fordító szükséges

  std::for_each(v.begin(), v.end(), sq);  // funktoral

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

  for(int& i : v)
    std::cout << i << ” ”;
}
```