

# OO Programozás

## Iterátorok, string, STL konténerek

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

[https://github.com/wajzy/GKxB\\_INTM119](https://github.com/wajzy/GKxB_INTM119)

2024. december 4.

## Tervezési minta (design pattern)

„Az informatikában programtervezési mintának (angolul Software Design Patterns) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.”\*

## Első jelentős irodalom

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns (*Elements of Reusable Object-Oriented Software*), Addison-Wesley, 1994

- Létrehozási minták
- Szerkezeti minták
- Viselkedési minták

## Iterátor

„Az Iterátor (...) minta lényege, hogy segítségével szekvenciálisan érhetjük el egy aggregált objektum elemeit, a mögöttes megvalósítás megismerése nélkül.”\*



## Forrás

## Feladat:

- Készítsünk saját iterátor interfészt, és azt megvalósító tényleges iterátorokat, melyekkel bejárható egy karakterlánc összes betűje, vagy egy láncolt lista elemei!
- A megvalósítás során törekedjünk a C/C++ programozók által jól ismert, mutatókhoz kapcsolódó operátorok alkalmazására!

## hasNext()

operator!= → két iterátor ugyanazt az elemet teszi-e elérhetővé?

next()

operator++ → iterátor léptetése a következő elemre

`operator*` → az iterátor által kijelölt elem lekérése



100

— — — — —

100

```
3 #include <cstring>
4 #include <stdexcept>
5 #include "Iterator.h"
6
7 class Messageliterator : public Iterator<char> {
8     char* p;
9     public:
10     Messageliterator(char* s) {
11         p = s;
12     }
```



100

```

13
14     bool operator!=(const Iterator<char>& it) const override {
15         return p != static_cast<const MessageIterator&>(it).p;
16     }
17
18     MessageIterator& operator++() override {
19         ++p;
20         return *this;
21     }
22
23     char& operator*() const override {
24         return *p;
25     }

```

override

Azt állítjuk, hogy a tagfüggvény (felül)definiálja az öröklött függvényt  $\rightarrow$  ha nem így van (pl. elgépelés), akkor hibaüzenettel leáll a fordítás.

## static\_cast

Az implicit és a felhasználó által definiált típuskonverzió kombinációja, szükség esetén hívja a konverziós konstruktort. *Fordítási időben* ellenőrzi a típusokat és eldönti, hogy az átalakítás végrehajtható-e. Használható primitív típusok közötti átalakításhoz, a származtatási hierarchiában történő fel- és lefelé lépéshez, vagy void mutatóról/ra történő konverzióhoz is.

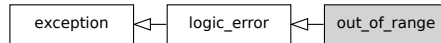
100

```

28 class Message {
29     private:
30         char* pStr;
31         int len; // The length of str is also stored
32     public:
33         Message() { // default ctor
34             pStr = new char('\0');
35             len = 0;
36         }
37
38         Message(const char* s) { // conversion ctor
39             len = strlen(s);
40             pStr = new char[len + 1];
41             strcpy(pStr, s);
42         }

```





# Message5.h

```
61     int length() const {
62         return len;
63     }
64
65     MessageIterator begin() const {
66         return MessageIterator(pStr);
67     }
68
69     MessageIterator end() const {
70         return MessageIterator(pStr + len);
71     }
72 };
```

\_\_\_\_\_



```

18     bool operator!=(const Iterator<T>& it) const override {
19         return p != static_cast<const ListIterator<T>&>(it).p;
20     }
21
22     ListIterator<T>& operator++() override {
23         p = p->next;
24         return *this;
25     }
26
27     T& operator*() const override {
28         return p->value;
29     }
30 };

```



```
32 template<class T>
33 class LinkedList {
34     private:
35         ListItem<T>* front;
36         ListItem<T>* tail;
37     public:
38         LinkedList() {
39             front = tail = nullptr;
40         }
41     };
42 }
```



```

50 void append(const T& i) {
51     ListItem<T>* latest = new ListItem<T>;
52     *latest = { i, nullptr };
53     if(tail == nullptr) {
54         front = latest;
55     } else {
56         tail->next = latest;
57     }
58     tail = latest;
59 };

```

\_\_\_\_\_

```
61     ListIterator<T> begin() {
62         return ListIterator<T>(front);
63     }
64
65     ListIterator<T> end() {
66         return ListIterator<T>(nullptr);
67     }
68 };
```



```

14 // range-based for loop, C++11
15 for (const auto& c : m) {
16     std::cout << c;
17 }
18 std::cout << std::endl;
19
20 // operator[]
21 try {
22     for (auto i = 0; i <= m.length(); i++) {
23         std::cout << m[i];
24     }
25 } catch (const std::out_of_range& e) {
26     std::cerr << "\nException caught: " << e.what() << std::endl;
27 }

```

```
29     LinkedList<int> l;  
30     l.append(1); l.append(2); l.append(3);  
31     for (auto i = l.begin(); i != l.end(); ++i) {  
32         std::cout << *i << '\t';  
33     }  
34     std::cout << std::endl;  
35 }
```

```
Hello C++ world!  
Hello C++ world!  
Hello C++ world!  
Exception caught: Message::operator[]  
1 2 3
```

## Range-based for loop (C++11)

- Bejárhatók vele tömbök,
- iterátort (`begin()`, `end()`) biztosító *gyűjtemények* (ld. később),
- és kapcsos zárójelek között felsorolt értékek.
- A soron következő elem elérhető érték szerint és referenciával is.



## range.cpp

```

3  int main() {
4      int array[] = {1, 2, 3};
5      for(const auto& i : array) { // reference
6          std::cout << i << '␣';
7      }
8      std::cout << std::endl;
9
10     for(auto i : array) { // value
11         std::cout << i << '␣';
12     }
13     std::cout << std::endl;
14
15     for(auto i : {4, 5, 6}) { // braced-init-list
16         std::cout << i << '␣';
17     }
18     std::cout << std::endl;

```

## range.cpp

```
20  std::string text = "C++ is so cool!\n";
21  // explicit iterator usage
22  for(auto i=text.begin(); i!=text.end(); ++i) {
23      std::cout << *i;
24  }
25
26  // implicit iterator usage, range-based for loop
27  for(const auto& c : text) {
28      std::cout << c;
29  }
30
31  // overloaded [] operator
32  for(size_t i=0; i<text.length(); ++i) {
33      std::cout << text[i];
34  }
```

Az `std::size_t` típus

- Előjel nélküli egész típus, ami tetszőlegesen nagy objektum méretét ( $\rightarrow$  `sizeof`) képes megadni bájtokban mérve.
- Általában indexelésre és ciklusszámlálóként használják az ilyen típusú változókat.

## C++ iterátorok

„Iterátor bármely olyan objektum, amely adatok (például egy tömb vagy egy gyűjtemény) valamely elemére mutatta operátorok egy halmazának (ami legalább a növelő (++) és indirekció \* műveleteket tartalmazza) segítségével képes az adatok között iterálni.”\*

A legkézenfekvőbb iterátor típus a *mutató*, de az összetett adatszerkezetek általában bonyolultabb megoldásokat igényelnek.

Minden iterátor közös jellemzői:

- Azonos típusú értékből vagy referenciából másolással létrehozható (copy constructible).
- Azonos típusú érték vagy referencia hozzárendelhető (copy assignable).
- Megsemmisítható (destructible), azaz skalár típus (felültöltés nélkül is értelmezhető rajta az összeadás művelet) vagy elérhető destruktorkkal rendelkező osztály, melynek minden nem statikus tagja is megsemmisíthető.
- Értelmezett rajta a növelés (++) operátor (prefix és suffix alakban is).



## Bidirectional

A Forward iterátorok képességein túl a csökkentés ( $--$ ) operátort is támogatja, bejárás hátrafelé.

## Random access

Mint Bidirection, de támogatja még az összeadást ( $+$ ,  $+=$ ), kivonást ( $-$ ,  $-=$ ), az egyenlőtlenségi relációkat ( $<$ ,  $<=$ ,  $>$ ,  $>=$ ) és az indexelést ( $[]$ ) is.

### range.cpp

```
36 // random access iterator
37 auto it = text.begin();
38 for (size_t i=0; i<text.length(); ++i) {
39     std::cout << it[i];
40 }
41 }
```

Az `std::string` osztály néhány érdekes tulajdonsága:

- Fejfájl: `<string>`
- Az `std::basic_string<char>` sablonpéldány szinonimája (typedef)
- Inicializálható C-stringgel (`char*`), konverzió visszafelé: `string::c_str()`
- Dinamikus memóriakezelést használ
- Tartalma megváltoztatható (mutable)
- Számos felültöltött operátor, pl. összefűzés `+`, indexelés `[]`
- Megvalósítja a `RandomAccessIterator`t



## stringDemo.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <cctype>

15 int main() {
16     int i;
17     std::cout << "Enter an integer: ";
18     std::cin >> i;
19
20     std::string str;
21     std::cout << "Enter a line of text: ";
22     // getline(std::cin, str); // 'new line' after number cannot be converted
23     // to 'int' and remains in input buffer
24     getline(std::cin >> std::ws, str); // clears leading whitespaces
25     std::cout << "You've entered: " << i << ", " << str << std::endl;
```

`std::ws` → figyelmen kívül hagyja a kezdeti fehér karaktereket

### Kimenet #1

```
Enter an integer: 5
Enter a line of text: hello
You've entered: 5, hello
```

### Kimenet #2

```
Enter an integer: 5cats
Enter a line of text: You've entered: 5, cats
```

## stringDemo.cpp

```
27  std::cout << "Capacity of the string is"
28      << str.capacity() << " chars.\n";
29  int len = str.length();
30  std::cout << "Length of the string is: " << len << " chars.\n";
31  const char fillChar = '!';
32  str.resize(len+3, fillChar);
33  std::cout << "Resized and filled with " << fillChar
34      << ": " << str << std::endl;
35  str.resize(len);
36  std::cout << "Undo resize: " << str << std::endl;
37  std::cout << "Shrinking string ... ";
38  str.shrink_to_fit();
39  std::cout << "capacity is now " << str.capacity() << " chars.\n";
```

## Kimenet #1

```
Enter an integer: 1
Enter a line of text: hello
You've entered: 1, hello
Capacity of the string is 15 chars.
Length of the string is: 5 chars.
Resized and filled with !: hello!!!
Undo resize: hello
Shrinking string... capacity is now 15 chars.
```

```
Enter an integer: 2
Enter a line of text: TheQuickBrownFoxJumpsOverTheLazyDog
You've entered: 2, TheQuickBrownFoxJumpsOverTheLazyDog
Capacity of the string is 60 chars.
Length of the string is: 35 chars.
Resized and filled with !: TheQuickBrownFoxJumpsOverTheLazyDog!!!
Undo resize: TheQuickBrownFoxJumpsOverTheLazyDog
Shrinking string... capacity is now 35 chars.
```

`capacity()`

A jelenleg lefoglalt memóriaterület mérete.

`length()`

A szöveg hossza.

`resize()`

Adott hosszúságúra alakítja a stringet. Ha eredetileg rövidebb volt, megadható, hogy milyen jelekkel töltsse fel. Ha hosszabb volt, a végét levágja.

`shrink_to_fit()`

A lefoglalt, de nem használt memóriaterület felszabadítása iránti igény jelzése. Nem feltétlenül lesz teljesítve.

## stringDemo.cpp

```
41 // to allow usage of the 's' suffix
42 using namespace std::string_literals;
43 std::cout << "std::string_literal_length:_"
44           << "std::string"s.length() << std::endl;
45 // std::string literal("std::string", 11);
```

Az `std::string_literals` névtér és az `s` végződés használatával egy `std::string` literál formálisan közvetlenül is létrehozható.

## Kimenet

```
Enter an integer: 5
Enter a line of text: hello
...
std::string literal length: 11
```

```

47 std::cout << "Displaying the text char-by-char with constant iterator:\n";
48 for(auto it=str.cbegin(); it!=str.cend(); ++it) {
49     std::cout << *it;
50 }
51 std::cout << std::endl;
52 std::cout << "Reverse direction:\n";
53 for(auto it=str.crbegin(); it!=str.crend(); ++it) {
54     std::cout << *it;
55 }
56 std::cout << std::endl;
57 std::cout << "Ciphertext: " << caesar(str, 4) << std::endl;
58 std::cout << "Plaintext: " << caesar(str, 26-4) << std::endl;

```



## stringDemo.cpp

```
5  std::string& caesar(std::string& str, int shift) {
6      for(auto it=str.begin(); it!=str.end(); it++) {
7          if(isalpha(*it)) {
8              char alpha = islower(*it) ? 'a' : 'A';
9              *it = alpha + (*it-alpha+shift)%('z'-'a'+1);
10         }
11     }
12     return str;
13 }
```

## Kimenet

Enter an integer: 5

Enter a line of text: hello

...

Displaying the text char-by-char with constant iterator:

hello

Reverse direction:

olleh

Ciphertext: lipps

Plain text: hello

Iterátorok:

`begin()`, `cbegin()`

A string elejére mutató (konstans) iterátor.

`end()`, `cend()`

A string végére mutató (konstans) iterátor.

`rbegin()`, `crbegin()`

A string végére mutató (konstans) iterátor, bejárás fordított irányban.

`rend()`, `crend()`

A string elejére mutató (konstans) iterátor, bejárás fordított irányban.

Caesar-rejtjelezés

## stringDemo.cpp

```
60     std::cout << "Indices of letter 'e':\n";
61     for(auto i=str.find("e", 0); i!=std::string::npos;
62         i=str.find("e", i)) { // or try to use rfind()
63         std::cout << i++ << ' ';
64     }
65     std::cout << std::endl;
66     std::cout << "First half:" << str.substr(0, str.length()/2)
67         << std::endl;
68     str.insert(0, "<"); str.insert(str.length(), ">");
69     std::cout << "Inserting:" << str << std::endl;
70     str.erase(0, 1); str.erase(str.length()-1);
71     std::cout << "Erasing:" << str << std::endl;
72 }
```

## Kimenet

```
Enter an integer: 42
Enter a line of text: Reverse engineering
...
Indices of letter 'e':
1 3 6 8 13 14
First half: Reverse e
Inserting: <Reverse engineering>
Erasing: Reverse engineering
```

`find(str, pos), rfind(str, pos)`

Rész-karakterlánc (`str`) keresése adott helyről (`pos`) indulva. Ha nincs találat, `npos`-sal tér vissza.

`substr(pos, count)`

Rész-karakterlánc előállítás `pos` helyről indulva, `count` hosszban.

`insert(index, s)`

Az `s` karakterlánc beszúrása `index` helyre.

`erase(index, count)`

Karakterek törlése az `index` helytől kezdve, `count` mennyiségben.

## Standard Template Library (STL)

### Gyűjtemények (containers)

dinamikus tömb (`vector`), láncolt lista (`list`, `forward_list`), leképezés (asszociatív tömb, szótár: `map`), halmaz (`set`), verem és sorok (`stack`, `deque`), stb.

### Algoritmusok

pl. keresés, csere.

### Iterátorok

elemek bejárása különféle irányokban és módokon

### Függvény objektumok (functors)

Függvények paramétereként átadható objektumok. Ezek tagfüggvényeivel a hívott függvény viselkedése befolyásolható, pl. egy gyűjtemény elemei átalakíthatók a kívánt (paraméterezésnek megfelelő) módon.

### Adapterek (adapters)

Más komponensek viselkedését módosítják, pl. egy iterátor bejárési irányát megfordítja.

## A `vector` osztály

- Fej fájl: `<vector>`
- Dinamikus tömb osztálysablon
- Speciális eset: `vector<bool>`  $\rightarrow$  bithalmaz
- Tartalma megváltoztatható (mutable)
- Véletlen elérés:  $\mathcal{O}(1)$ , beszúrás/törlés a végén: amortizált  $\mathcal{O}(1)$ , tetszőleges helyen  $\mathcal{O}(n)$ .



## vectorDemo.cpp

```
1  #include <iostream>
2  #include <vector>

12 int main() {
13     std::vector<int> iv1 = {1, 2, 3, 4, 5}; // initializer list
14     for(const auto& i : iv1) {
15         std::cout << i << ' ';
16     }
17     std::cout << std::endl;
18
19     std::vector<int> iv2 {6, 7, 8, 9, 10}; // uniform initialization
20     for(const auto& i : iv2) {
21         std::cout << i << ' ';
22     }
23     std::cout << std::endl;
```

## vectorDemo.cpp

```
25     std::vector<int> iv3(5, 42); // constructor #1
26     for(auto i = iv3.cbegin(); i!=iv3.cend(); ++i) {
27         std::cout << *i << '␣';
28     }
29     std::cout << std::endl;
30
31     std::vector<int> iv4(5); // constructor #2
32     for(size_t i=0; i<iv4.size(); i++) {
33         std::cout << iv4[i] << '␣';
34     }
35     std::cout << std::endl;
```

## vectorDemo.cpp

```
37  std::vector<int> iv5; // constructor #3
38  iv5.push_back(11); iv5.push_back(12); iv5.push_back(13);
39  for(size_t i=0; i<iv5.size(); i++) {
40      std::cout << iv5.at(i) << ' ';
41  }
42  std::cout << std::endl;
43
44  // 2x3 matrix of 1's
45  std::vector<std::vector<int>> im1(2, std::vector<int>(3, 1));
46  for(const auto& r : im1) {
47      for(const auto& c : r) {
48          std::cout << c << ' ';
49      }
50      std::cout << std::endl;
51  }
```

## Kimenet

```
1 2 3 4 5
```

```
6 7 8 9 10
```

```
42 42 42 42 42
```

```
0 0 0 0 0
```

```
11 12 13
```

```
1 1 1
```

```
1 1 1
```

## Inicializálás, példányosítás

- inicializáló listával
- explicit **konstruktor** hívással, pl.
  - `vector();`
  - `explicit vector( size_type count );`
  - `vector( size_type count, const T& value, const Allocator& alloc = Allocator() );`

## vectorDemo.cpp

```
53  std::vector<int> iv6 = {1, 2, 3, 4, 5};
54  std::cout << "Size_of_iv6:_" << iv6.size() << std::endl;
55  std::cout << "Capacity:_" << iv6.capacity() << std::endl;
56  std::cout << "Reserving_memory..._" ; iv6.reserve(16);
57  std::cout << "Capacity:_" << iv6.capacity() << std::endl;
58  std::cout << "Is_it_empty?_" << (iv6.empty() ? "Yes" : "No") << std::endl;
59  std::cout << "Shrinking..._" ; iv6.resize(3); printForward(iv6);
60  std::cout << "Growing..._" ; iv6.resize(6); printForward(iv6);
61  std::cout << "Further_growing..._" ; iv6.resize(9, -1); printForward(iv6);
62  std::cout << "Shrinking..._" ; iv6.shrink_to_fit();
63  std::cout << "Capacity:_" << iv6.capacity() << std::endl;
```

## vectorDemo.cpp

```
4  template<class T>
5  void printForward(const std::vector<T>& v) {
6      for(auto i=v.begin(); i!=v.end(); i++) {
7          std::cout << *i << ' ';
8      }
9      std::cout << std::endl;
10 }
```

Shrinking... Capacity: 9



`size()`

Tárolt elemek száma.

`capacity()`

Ennyi elem számára van lefoglalva memóriaterület.

`reserve(new_cap)`

Legalább `new_cap` elemszámú adatnak foglal le memóriaterületet.

`empty()`

Logikai igaz értékkel tér vissza, ha a vektor üres.

`resize(count)`, `resize(count, value)`

Ha `count` kisebb `size()`-nál, akkor az utolsó elemeket levágja. Ha nagyobb, akkor kibővíti a vektort és a `value` másolataival tölti fel az új elemeket.

`shrink_to_fit()`

Kezdeményezi (de nem feltétlenül hajtja végre) a kihasználatlan kapacitások törlését.

## vectorDemo.cpp

```
65  std::cout << "First_element:_ " << iv6.front() << std::endl;
66  std::cout << "Last_element:_ " << iv6.back() << std::endl;
67  std::cout << "Element_at_idx._1:_ " << iv6[1] << std::endl;
68  std::cout << "Element_at_idx._2:_ " << iv6.at(2) << std::endl;
69  std::cout << "Elements_in_reverse_order_(using_pointers):_";
70  for(const int* pi = iv6.data()+iv6.size()-1; pi>=iv6.data(); pi--) {
71      std::cout << *pi << '_';
72  }
73  std::cout << "\nElements_in_reverse_order_(using_rev_it.):_";
74  for(std::vector<int>::const_reverse_iterator i = iv6.crbegin();
75      i!=iv6.crend(); i++) {
76      std::cout << *i << '_';
77  }
78  std::cout << std::endl;
```

## Kimenet

First element: 1

Last element: -1

Element at idx. 1: 2

Element at idx. 2: 3

Elements in reverse order (using pointers): -1 -1 -1 0 0 0 3 2 1

Elements in reverse order (using rev. it.): -1 -1 -1 0 0 0 3 2 1

`front()`, `back()`

Visszaadják az első és utolsó tárolt elemet.

`operator[] (pos)`, `at (pos)`

Visszaadják a pos indexű elemet.

`data()`

Visszaadja a lefoglalt memóriaterület kezdőcímét.

`cbegin()`, `crend()`

Csak olvasásra alkalmas iterátorok az elemek fordított sorrendben történő eléréséhez.

## vectorDemo.cpp

```
80  std::vector<int> iv7;
81  std::cout << "Assigning ... "; iv7.assign(3, 1); printForward(iv7);
82  std::cout << "Pushing ... "; iv7.push_back(2); printForward(iv7);
83  std::cout << "Popping ... "; iv7.pop_back(); printForward(iv7);
84  std::cout << "Inserting ... "; iv7.insert(iv7.end(), 2); printForward(iv7);
85  std::cout << "Erasing ... "; iv7.erase(iv7.end()-1); printForward(iv7);
86  std::cout << "Swapping vectors ... \n"; iv7.swap(iv6);
87  std::cout << "\niv7: "; printForward(iv7);
88  std::cout << "\niv6: "; printForward(iv6);
89  return 0;
90 }
```

## Kimenet

```
Assigning... 1 1 1
Pushing... 1 1 1 2
Popping... 1 1 1
Inserting... 1 1 1 2
Erasing... 1 1 1
Swapping vectors...
    iv7: 1 2 3 0 0 0 -1 -1 -1
    iv6: 1 1 1
```

`assign(count, value)`

Lecseréli a vektor elemeit count darab value-ra.

`push_back(value), pop_back()`

Hozzáfűznek vagy eltávolítanak egy elemet a vektor végéhez, -ról.

`insert(pos, value), erase(pos)`

value beszúrása a pos iterátorral adott helyre, vagy egy elem törlése onnan.

`swap(other)`

Két vektor elemeinek és lefoglalt tárterületének felcserélése.

## A deque osztály

## A deque osztály

- Fejfájl: <deque>
- Hasonló a dinamikus tömbhöz, de lehetővé teszi a gyors ( $\mathcal{O}(1)$ ) beszúrást és törlést a gyűjtemény mindkét végén. Ezt jellemzően úgy éri el, hogy az elemeket nem összefüggő memóriaterületen tárolja, hanem több, egymástól függetlenül lefoglalt memóriaterületen.
- Véletlen elérés:  $\mathcal{O}(1)$ , beszúrás vagy törlés a széleket leszámítva  $\mathcal{O}(n)$ .
- Tartalma megváltoztatható (mutable).



## dequeDemo.cpp

```
1  #include <iostream>
2  #include <deque>
3
4  template<class T>
5  void printForward(const std::deque<T>& dq) {
6      for(auto i=dq.cbegin(); i!=dq.cend(); i++) {
7          std::cout << *i << ' ';
8      }
9      std::cout << std::endl;
10 }
```

## dequeDemo.cpp

```
12  int main() {  
13      std::deque<int> dq;  
14      dq.push_back(4);  
15      dq.push_front(2);  
16      dq.push_back(8);  
17      dq.push_front(1);  
18      printForward(dq);  
19  
20      dq.pop_front(); dq.pop_back();  
21      printForward(dq);  
22      return 0;  
23  }
```

## Kimenet

1 2 4 8

2 4

`push_front()`, `pop_front()`

Beszúrás / törlés a sor elején.

## A list osztály

- Fejfájl: <list>
- Két irányban láncolt lista. Egy irányban láncolt változat: `forward_list`
- Gyors ( $\mathcal{O}(1)$ ) beszúrás és törlés a gyűjtemény bármely pontján, de az elemek véletlen elérése nem támogatott, a bejárás lassú.
- Tartalma megváltoztatható (mutable).

## listDemo.cpp

```
1 #include <iostream>
2 #include <list>
3
4 template<class T>
5 void printForward(const std::list<T>& l) {
6     for(auto i=l.cbegin(); i!=l.cend(); i++) {
7         std::cout << *i << ' ';
8     }
9     std::cout << std::endl;
10 }
```

## listDemo.cpp

```
12  int main() {
13      std::list<int> l = { 2, 3, 4 };
14      l.push_front(1); l.push_back(5);
15      for(auto& i : l) {
16          i *= i;
17      }
18      l.insert(l.end(), 6*6);
19      std::list<int>::iterator it = l.end(); advance(it, -l.size());
20      l.insert(it, 0);
21      std::cout << "Square numbers: "; printForward(l);
```

Lépteti az `it` iterátort `n` elemmel. (Ha az iterátor mindkét irányban bejárható, akkor `n` értéke lehet negatív.)





In reverse order: 36 25 16 9 4 1 0

After sorting: 0 1 4 9 16 25 36

After merging: -3 -2 -2 -2 -1 0 1 4 9 16 25 36

Removed duplicates: -3 -2 -1 0 1 4 9 16 25 36

```
reverse()
```

Megfordítja az elemek sorrendjét.

sort()

Növekvő sorrendbe rendezi az elemeket.

```
merge(list)
```

Átmozgatja list tartalmát az aktuális listába. Ha mindkettő rendezett, akkor az eredmény is az lesz.

unique()

Az egymást követő azonos értékek közül csak az elsőt hagyja meg.

## A set osztály

- Fej fájl: <set>
- Rendezett halmaz, ismétlődés nem megengedett.
- Keresés, beszúrás, eltávolítás:  $\mathcal{O}(\log n)$  komplexitás  $\rightarrow$  többnyire piros-fekete fával megvalósítva.
- Tartalma megváltoztatható (mutable).

setDemo.cpp

```
1 #include <iostream>
2 #include <set>
3
4 template<class T>
5 std::ostream& operator<<(std::ostream& os, const std::set<T>& s) {
6     for(const auto& item : s) {
7         os << item << ' ';
8     }
9     return os;
10 }
```

## setDemo.cpp

```
12 template<class T>
13 std::set<T> intersection(const std::set<T>& a, const std::set<T>& b) {
14     std::set<T> result;
15     for(auto item : a) {
16         if(b.find(item) != b.end()) {
17             result.insert(item);
18         }
19     }
20     return result;
21 }
```

## setDemo.cpp

```
23 int main() {
24     std::set<int> even = { 0, 2, 4, 6, 8, 10 };
25     std::cout << "Even numbers: " << even << std::endl;
26
27     std::set<int> three;
28     for(auto i=0; i<11; i++) {
29         three.insert(i);
30     }
31     for(auto it = three.begin(); it != three.end(); ) {
32         if(*it % 3 != 0) {
33             it = three.erase(it);
34         } else {
35             ++it;
36         }
37     }
38     std::cout << "Numbers divisible by three: " << three << std::endl;
```



Törli az iterátorral adott elemet, és a megváltozott adatszerkezet bejárására alkalmas új iterátort szolgáltat.

## A map osztály

## A `map` osztály

- Fej fájl: `<map>`
- Asszociatív tömb / leképezés / szótár: kulcs-érték párok halmaza (kulcs nem ismétlődhet).
- Elemei **párok**.
- A megvalósítás többnyire **piros-fekete fával** történik, ezért a keresés, beszúrás, törlés  $\mathcal{O}(\log(n))$  időt vesz igénybe.
- Tartalma megváltoztatható (mutable).



## mapDemo.cpp

```
1 #include <iostream>
2 #include <map>
3
4 template<class K, class V>
5 void printForward(const std::map<K, V>& m) {
6     for(auto i=m.cbegin(); i!=m.cend(); i++) {
7         std::cout << i->first << '\t' << i->second << std::endl;
8     }
9 }
```

## mapDemo.cpp

```
11  int main() {
12      std::map<std::string, int> students;
13      students["Johnny"] = 3;
14      students["Johnny"] = 5; // no duplicated keys
15      students.insert(std::pair<std::string, int>("Jenny", 4));
16      students.erase("Tommy"); // does nothing
17      std::cout << "Stored pairs:\n"; printForward(students);
18      std::cout << "Is Eva present? "
19                  << (students.find("Eva")==students.end() ? "No" : "Yes");
20      return 0;
21  }
```

## Kimenet

Stored pairs:

Jenny 4

Johnny 5

Is Eva present? No

`insert(value)`

Beszúr egy kulcs-érték párt.

`operator[] (key)`

Adott kulcsú elem olvasása vagy beszúrása / módosítása.

`erase(key)`

Törli az adott kulcsú elemet, ha létezik.

`find(key)`

Visszaadja a key-t kijelölő iterátort, ha a kulcs létezik, különben az `end()` iterátort.