OPERÁTOROK, ITERÁTOROK, ÉS FÜGGVÉNY OBJEKTUMOK

OBJEKTUMORIENTÁLT PROGRAMOZÁS

- Operátorok definiálása saját típusokhoz
  - + friend tagelérés
- Iterátorok működése
  - + belső osztályok
- Az <algorithm> függvényei
  - + függvény objektumok használata

SZÉCHENYI EGYETEM
UNIVERSITY OF GYŐR

OPERÁTOROK

Operátor ≈ műveleti jel / relációs jel

- + egyéb operátorok, pl.: sizeof, new, delete, feltételes operátor (?:), címképzés (&), indirekció (*), tagelérés (. és ->), indexelés ([]), scope (::)

Egyszerű típusokra a működést a C ill. C++ szabvány adja meg

- Operátor szimbóluma
- Operandusok száma
- Műveleti sorrend: precedencia és asszociativitás
- Eredmény típusa
- Műveletet elvégző utasítás(ok)

Saját, összetett típusokra nekünk kell definiálnunk a működést

- Az utolsó 2 aláhúzott pontot tudjuk definiálni, a többi adott

OPERÁTOR FÜGGVÉNY

Az operátorok működését speciális operátor függvényekkel lehet definiálni

- Név: operator kulcsszó, majd az operátor szimbóluma
- Paraméterek: operandus(ok)
  - Számuk az operátortól függ (ugyanannyi, mint egyszerű típusoknál)
  - Egy operátorhoz lehet többféle paraméterezést (típusokat) is megadni
- Visszatérési érték: szabadon meghatározható
- Lehet tagfüggvény is, ekkor az első operandus metódusaként hívódik
  - Eléri a privát adattagokat is, míg egy külső függvény csak a publikus interfészhez fér hozzá

PÉLDA: OPERÁTOR FÜGGVÉNYEK

```
struct Date {
    int d, m, y;

    Date(int day, int month, int year);
    void add_years(int n);
    void add_months(int n);
    void add_days(int n);
    bool operator==(const Date& op2) { // as member function
        return d == op2.d && m == op2.m && y == op2.y;
    }
};

Date operator+(const Date& op1, int op2) { // as global function
    Date result = op1;
    result.add_days(op2);
    return result;
}

std::ostream& operator<<(std::ostream& op1, const Date& op2) {
    // members need to be accessible!
    return op1 << op2.y << ". " << op2.m << " ." << op2.d;
}
```

OPERÁTOROK LISTÁJA

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

Nem átdefiniálható operátorok			
::	.*	.	?:

FRIEND OSZTÁLYOK ÉS FÜGGVÉNYEK

Külső függvények csak a publikus interfészen érik el az osztályt

Operátor viszont csak akkor lehet tagfüggvény, ha az osztály példánya az első (bal oldali) operandus

- Probléma: `operator<<(ostream& op1, const MyClass& op2)`
- Néha nem publikus adatokat is ki szeretnénk írni (pl. debug info)

Más kivételes esetekben is hasznos lehet, ha egy külső függvény vagy osztály (pl. Tester) hozzáfér a privát tagokhoz

- De nem szeretnénk emiatt publikussá tenni, más ne férjen hozzá

Osztálydefinícióban megadhatók friend osztályok és függvények, amik hozzáférnek a privát adattagokhoz is

- `friend class FriendClass;`
- `friend void friendFunction(const MyClass&);`

PÉLDA: FRIEND OSZTÁLYOK ÉS FÜGGVÉNYEK

```
class Date {
    int d, m, y;
    friend class Test::DateTester;
    friend std::ostream& operator<<(std::ostream&, const Date&);
public:
    Date(int day, int month, int year);
    void add_years(int n);
    void add_months(int n);
    void add_days(int n);
    bool operator==(const Date& op2) { // has access to own private members
        return d == op2.d && m == op2.m && y == op2.y;
    }
};

Date operator+(const Date& op1, int op2) {
    Date result = op1;
    result.add_days(op2); // public method
    return result;
}
// friend has access to private members
std::ostream& operator<<(std::ostream& op1, const Date& op2) {
    return op1 << op2.y << ". " << op2.m << " ." << op2.d;
}
```

ITERÁTOROK

Az STL tárolók elemeinek bejárását az iterátor interfész egységesíti

Az iterátor a mutatók viselkedését utánozza

- *, -> operátorokkkal az elemet ill. annak adattagját/metódusát érjük el
- ++ operátorral tudjuk a következő elemre léptetni

Különböző iterátor típusok a lehetséges műveletek alapján

- Forward iterator: `++` `==` `!=`
- Bidirectional: `++` `--` `==` `!=`
- Random-access: `++` `--` `+` `-` `+=` `-=` `[]` `==` `!=` `<` `>` `<=` `>=`
- A tárolási logikától függ, hogy egy tároló milyet biztosít

Konstans iterátor a nem-módosító hozzáféréshez

ITERÁTOROK HASZNÁLATA

A tárolók több metódusa iterátort add vissza vagy vár paraméterben

- `begin(), cbegin()`: az első elemre mutató (const) iterátor
- `end(), cend()`: az elemek végét jelző iterátor
  - Nem az utolsó elemre mutat! (kb. az utolsó utáni elemre mutat)
- `insert(pos, value), insert(pos, first, last)`: a pos iterátor elé szúrja be az elemet, ill. a first, last iterátorok közti elemeket

Példa: lista bejárása

```
list<int> numbers;
for (int i=1; i<=5; i++) numbers.push_back(i);
for (list<int>::iterator it = numbers.begin();
        it != numbers.end(); ++it)
    cout << *it << '\n';
```

ITERÁTOR IMPLEMENTÁCIÓJA

Az iterátorok az STL tárolók belső osztályaiként (nested class) vannak deklarálva

- Ahogy adattagokat és tagfüggvényeket, osztályokat is lehet deklarálni egy osztályon belül
- Ekkor a külső osztály névtérként viselkedik (`list<int>::iterator`)
- A private/protected/public láthatóság a belső osztályokra is vonatkozik

Saját tároló osztályunknak is készíthetünk iterátort, hogy kompatibilis legyen a standard libraryvel

- ~~Csak származtatnunk kell az `std::iterator` osztályból~~
- C++17: Csak meg kell adni az iterátor típusát és a használt típusokat
- Meg kell valósítani az adott iterátor típus interfészét

PÉLDA: RANGE OSZTÁLY SAJÁT ITERÁTORRAL

A saját iterátor egy egyszerűbb példán lesz bemutatva:

- A `Range<A,B>` az `A, A+1, A+2, …, B-1, B` sorozatnak felel meg
- A template argumentumok itt nem osztályok vagy típusok, hanem konstans értékek
- Nincs dinamikus tároló, amit kezelni kell, csak konstans határok

```
#include <iostream>
#include <vector>
#include "Range.h"
using namespace std;

int main() {
    for (int i = 4; i <= 10; ++i)
        cout << i << '\n';
    Range<4,10> range;
    for (Range<4,10>::iterator it = range.begin(); it != range.end(); ++it)
        cout << *it << '\n';
    vector<int> vec(range.begin(), range.end()); // create vector from range
    for (vector<int>::iterator it = vec.begin(); it != vec.end(); ++it)
        cout << *it << '\n';
}
```

```
// provides an iterator for the integer sequence between FROM and TO
template<long FROM, long TO>
struct Range {
    class iterator {
        long num = FROM;
    public:
        iterator(long _num = 0) : num(_num) {}
        // prefix increment: advance and return self
        iterator& operator++() {
            num = TO >= FROM ? num + 1: num - 1; // increasing or decreasing sequence
            return *this;
        }
        // postfix increment (dummy int argument)
        iterator operator++(int) {
            iterator retval = *this; // create copy
            ++(*this);               // advance self
            return retval;           // return copy (this is why prefix is preferred)
        }
        bool operator==(iterator other) const { return num == other.num; }
        bool operator!=(iterator other) const { return !(*this == other); }
        long operator*() { return num; }
        using iterator_category = std::forward_iterator_tag; // iterator traits
        using difference_type = long; // type aliases (similar to typedef)
        using value_type = long;
        using pointer = const long*;
        using reference = const long&;
    };
    iterator begin() { return FROM; } // conversion by iterator(long) constructor
    iterator end() { return TO >= FROM? TO+1 : TO-1; }
};
```

STANDARD ALGORITMUSOK

A C++ standard library nem csak konténer osztályokat biztosít, hanem számos rajtuk végezhető művelet függvényét is

- Rendezés (`std::sort`)
- Keresés (`std::find`)
- Min-/maximumkeresés (`std::minmax_element`)
- Megszámlálás (`std::count`)

Ezeket a függvényeket az <algorithm> tartalmazza

- A paraméterként kapott [first, last) iterátorok között dolgoznak
- Az összehasonlításhoz az elemek között definiálni kell a megfelelő operátorokat (<, ==)
- Vagy paraméterben meg kell adni a komparátor/predikátum függvényt

FÜGGVÉNY, MINT PARAMÉTER

Mi is készíthetünk olyan függvényt, ami egy másik függvényt vár paraméterben

```
void apply(vector<int>& vec, int (*fn)(int)) {
    for (int i = 0; i < vec.size(); ++i)
        vec[i] = fn(vec[i]);
}
```

A 2. paraméter egy int visszatérési értékű, 1 db int paraméterű függvény

Függvény template-tel is megoldható, ekkor a híváskor átadott függvény típusa specializálja a típust

```
template<class Function>
void apply(vector<int>& vec, Function fn) {
    for (int i = 0; i < vec.size(); ++i)
        vec[i] = fn(vec[i]);
}
```

FÜGGVÉNY POINTER ÁTADÁSA

Átadáskor a függvény neve lesz a paraméter, zárójelek nélkül:

```
#include <vector>
using namespace std;

int square(int n) { return n * n; }

int main() {
    int myints[] = { 3, -7, 12, -5, -2, 0, 5, -8 };
    // fill vector from array
    vector<int> vec(myints, myints + 8);

    apply(vec, square); // or: apply(vec, &square);
}
```

FÜGGVÉNY OBJEKTUM (FUNKTOR)

Egy objektum is tud függvényként viselkedni, ha definiálva van rá az `operator()`

```
struct Multiplier {
    int mult;
    int operator()(int x) { return x * mult; }
};
int main() {
    int myints[] = { 3, -7, 12, -5, -2, 0, 5, -8 };
    vector<int> vec(myints, myints + 8);

    Multiplier multObj = { 5 };
    vec[0] = multObj(vec[0]); // behaves like a function
    apply(vec, multObj);
}
```

- A <functional> header számos függvény objektum template-et biztosít a gyakori aritmetikai, összehasonlítás és logikai műveletekhez

LAMBDA KIFEJEZÉS

A függvény objektumokat gyakran egyetlen speciális használatra hozzuk létre

- Nem használható fel újra, emiatt kár egy új osztályt csinálni

A lambda-kifejezés egy ideiglenes, névtelen függvény objektum

- Az átadáskor, helyben hozzuk létre:

```
int main() {
    int myints[] = { 3, -7, 12, -5, -2, 0, 5, -8 };
    vector<int> vec(myints, myints + 8);
    apply(vec, [](int x) { return x % 2; } );
}
```

- Formátum: **[**captures**] (**params**) -> ret_t { body }**
  - A captures részben felsorolt, aktuális scope-ban lévő változók elérhetők lesznek a függvényből (kb. mint a függvény objektum adattagjai)
  - A **->ret_t** elhagyható, ha a visszatérési érték típusa kikövetkeztethető

PÉLDA: <ALGORITHM> FÜGGVÉNYEK

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

bool lessAbs (int a, int b) { return abs(a) < abs(b); }

struct Filter {
    int limit;
    bool operator() (int x) { return x < limit; }
};

int main() {
    int myints[] = { 3, -7, 12, -5, -2, 0, 5, -8 };
    vector<int> vec(myints, myints + 8);

    // sort by absolute value with function pointer
    sort(vec.begin(), vec.end(), lessAbs);
    for (int n : vec) { cout << n << ' '; } cout << endl; // 0 -2 3 -5 5 -7 -8 12

    // count negative numbers with function object
    Filter zeroFilter = { 0 };
    int negatives = count_if(vec.begin(), vec.end(), zeroFilter);
    cout << negatives << " negative numbers\n"; // 4

    // find first odd number with lambda
    vector<int>::iterator it;
    it = find_if(vec.begin(), vec.end(), [](int x) { return x % 2 == 1; });
    cout << "First odd is " << *it << ", at index " << it-vec.begin() << endl;
}
```