Objektumorientált programozás

Objektumalapú programozás a C++ programozási nyelvben

A standard könyvtár felépítése

Darvay Zsolt

A standard könyvtár

17. A standard könyvtár felépítése
18. Adatfolyamok
19. Algoritmusok

17. A standard könyvtár

Áttekintés

17.1. A könyvtár szerkezete
17.2. A tárolók osztályozása
17.3. Sorozatok és asszociatív tárolók
17.4. Bejárók
17.5. Memóriafoglalók
17.6. Karakterláncok és számok

17.1. A könyvtár szerkezete

- A standard könyvtár tartalmazza:
- a C régebbi könyvtári függvényeit (a megfelelő fejállomány általában c-vel kezdődik, például math.h helyett cmath lesz),
- az adatfolyamokra vonatkozó osztályhierarchiát,
- a szabványos sablon könyvtár elemeit (STL = standard template library).
- A továbbiakban ez utóbbival foglalkozunk.

A szabványos sablon könyvtár

- A szabványos sablon könyvtár tartalmaz:
  - tárolókat (container);
  - bejárókat (iterator);
  - algoritmusokat.
- A tárolók egy bizonyos adatszerekezetet valósítanak meg. Objektumok tárolására alkalmasak.
- A bejáró a tároló elemeit szolgáltatja.

Statikus és dinamikus tárolók

- Ha a tárolóban elhelyezhető objektumok maximális száma futási időben változhat, akkor dinamikus tárolóról beszélünk, ellenkező esetben pedig statikus tárolóról.
- A standard könyvtár tárolói általában dinamikus tárolók. Kivétel az array C++11-től kezdődően.
- A klasszikus tömbök statikus tárolóknak tekinthetők.

17.2. A tárolók osztályozása

- A tárolók lehetnek:
  - sorozatok (sequences),
    más néven: szekvenciális tárolók;
  - asszociatív tárolók (associative container);
  - sorozat-átalakítók (sequence adapter)
    más néven: tároló-átalakítók
    (container adapter)
  - majdnem-tárolók

Ábrázolási mód

- Milyen adatszerkezettel valósítják meg az egyes tárolókat?
- A szabvány ezt nem írja elő, csak a tároló felületét (interface) rögzíti, illetve azt is, hogy az egyes műveletek bonyolultsága (hatékonysága) milyen legyen.

17.3. Sorozatok és asszociatív tárolók

- Sorozatok:
  - vector, list, deque
  - C++11-től: array, forward_list
- Asszociatív tárolók:
  - set, multiset, map, multimap

C++11-től: nem rendezett asszociatív tárolók:
unordered_set, unordered_map,
unordered_multiset, unordered_multimap

Sorozatok

- Az objektumok lineárisan vannak tárolva.
- Nincsenek rendezve egy bizonyos kulcs szerint.
- Sorozatok:
  - vector<T>
  - list<T>
  - deque<T>

A vector tároló

- Az objektumokhoz direkt hozzáférést (közvetlen elérést, random access) biztosít (a klasszikus tömbökhöz hasonlóan).
- Nincs rögzítve a tárolóban elhelyezhető objektumok maximális száma. Ez a futtatás közben változhat (dinamikus tároló).
- A hozzáadási és törlési műveletek, valós időben, a tároló végén történnek.
- Akkor használhatjuk, ha nem rendezett objektumok egy sorozatát szeretnénk tárolni.
- Ábrázolási mód: általában egy hagyományos tömb.

A vector tároló megvalósítása

```
template < class T, class A = allocator<T> >
class vector {
   // ...
};
```

az elemek típusa

memóriafoglaló (nem kötelező)
ha hiányzik, akkor az
alapértelmezett **allocator** lesz
felhasználva

A vector tároló tagjai

- Típusdeklarációk
- Tagfüggvények
- Operátor:
  - a [] operátor: referenciát térít vissza a vektor megfelelő eleméhez.

Néhány típusdeklaráció

- iterator: egy típus, amely egy direkt hozzáférésű bejárót határoz meg annak érdekében, hogy a tároló elemeit lekérdezni, illetve módosítani lehessen.
- reverse_iterator: az elemeket fordított sorrendben szolgáltató bejáró típusa (visszafelé haladó bejáró).
- reference: egy elemre hivatkozó referecia típus.
- value_type: a vektor elemeinek a típusa.

Néhány tagfüggvény

- begin: az első elemre hivatkozó bejáró.
- end: az utolsó utáni elemre hivatkozó bejáró.
- rbegin: az első elemre hivatkozó visszafelé haladó bejáró (reverse_iterator).
- rend: az utolsó utáni elemre hivatkozó visszafelé haladó bejáró (reverse_iterator).
- empty: ellenőrzi, hogy üres-e a vektor?
- size: az elemek számát adja vissza.

Néhány tagfüggvény

- push_back: hozzáad egy elemet a vektor végére.
- pop_back: kitörli az utolsó elemet.
- at: az adott indexszel jelemzett értékhez térít vissza referenciát. A [] operátorral szemben indexhatár-ellenőrzést is végez. Egy out_of_range kivételt vált ki, ha az index a tartományon kívül esik.
- back: referencia az utolsó elemre
- front: referencia az első elemre

A list tároló

- A hozzáadási és törlési műveletek akárhol hatékonyan elvégezhetők.
- Ábrázolási mód:

  általában kétszeresen láncolt lista.

A deque tároló

- deque: kétvégű sor (double-ended queue)
- A hozzáadási és törlési műveletek a tároló elején, és a végén is, valós időben végezhetők el (a list tárolóhoz hasonló hatékonysággal).
- Az tároló belsejében az elemek beszúrása és törlése már lassú lesz.
- A indexelés (direkt hozzáférés az elemekhez) ugyanolyan hatékony mint a vector esetén.

Asszociatív tárolók

- Lehetőség van az objektumok gyors visszakeresésére, bizonyos kulcsok alapján.
- Általában rendezve van (kivéve a C++ 11-től bevezetett nem rendezett asszociatív tárolók).
- A rendezés módját egy predikátummal (függvényobjektummal) adjuk meg.
- Rendezett asszociatív tárolók:
  - set<T, Pred>
  - multiset<T, Pred>
  - map<Key, T, Pred>
  - multimap<Key, T, Pred>

A binary_function osztálysablon

```
template<class Arg1, class Arg2, class Result>
   struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

- Alaposztály, amelyből függvényobjektumok származtathatók. A functional fejállomány része.

A „less” függvényobjektum

```
template<class T>
struct less : public binary_function<T, T, bool>
{
   bool operator()(const T& x, const T& y) const;
   { return x < y; }
};
```

- A functional fejállományban van definiálva.

Asszociatív tömb

- Az asszociatív tömb egy hasznos felhasználói típus. Egyes nyelvekben beépített típusként szerepel.
- Leképezések (map) esetén meghatározott párokat tárol: (kulcs, érték).
- Például a szótár is egy leképezés.
- A kulcs alapján a hozzárendelt értéket határozza meg.
- Egyedi kulcs: minden kulcshoz pontosan egy érték tartozik.

Az asszociatív tároló és az asszociatív tömb

- A C++-beli asszociatív tároló az asszociatív tömb fogalmát terjeszti ki.
- A map az asszociatív tömb megfelelője.
- A multimap: asszociatív tömb, amelyben egy kulcshoz több érték is tartozhat.
- A set és multiset olyan asszociatív tömbök, amelyek nem határoznak meg hozzárendelt értéket.

A set<Key, Pred> tároló

- Egyedi kulcsok (Key): egyértelmű kulcsokat tartalmaz (a kulcs minden értékére egyetlen objektum van a tárolóban).
- A Pred predikátum alapértelmezett értéke a less függvényobjektum.
- Akkor használhatjuk, ha egyetlen rendezési kulcs van, például egy egész elemekből álló halmaz esetén.

A multiset<Key, Pred> tároló

- Egy adott kulcsra több objektumunk lehet a tárolóban.
- Akkor használhatjuk, ha több rendezési kulcs van.

A map<Key, T, Pred> tároló

- Az asszociatív tömbnek felel meg.
- Egyedi kulcsokat tartalmaz.
- A kulcs (Key) alapján egy T típusú elemet határoz meg, tehát nem magát a kulcsot, mint a set esetében.
- Egy egyszerű adatbázis esetén használhatjuk, abban az esetben, ha egy kulcs van.

A multimap<Key, T, Pred> tároló

- Egy adott kulcsra több objektumunk lehet a tárolóban.
- A kulcs (Key) alapján egy T típusú elemet határoz meg.
- Egy bonyolultabb adatbázis esetén használhatjuk, amely több kulccsal rendelkezik.

Sorozat-átalakítók

- Átalakító (adapter): egy korlátozott felületet biztosít egy adott tárolóhoz.
- Az átalakítókhoz nem rendelünk bejárókat, mivel a szűkebb felület az elvárásainknak eleget tesz.
- Sorozat-átalakítók:
  - stack (verem)
  - queue (sor)
  - priority_queue (prioritásos sor)

A verem átalakító

Az STL-beli megvalósítás egy része:

```
template <class T, class Container = deque<T> >
class std::stack {
protected:
   Container c;
public:
   // ...
   void pop() { c.pop_back(); }
};
```

Majdnem-tárolók

- hagyományos tömbök
- karakterláncok: basic_string
- numerikus számítások: valarray (a számokkal végezhető műveleteket támogató vektor), complex (komplex számok), ratio (törtek, C++11-től)
- bithalmazok: bitset

Példa a vector tárolóra

- A vector tárolót úgy fogjuk használni, hogy egy veremmel tudjunk dolgozni.

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

A fő függvény

```
int main() {
    int n = 10;
    vector<int> v;
    for (int i = 0; i < n; i++)
        v.push_back(i);
    while (!v.empty()) {
        cout << v.back() << " ";
        v.pop_back();
    }
}
```

Kimenet: 9 8 7 6 5 4 3 2 1 0

Az előző példa a stack tárolóval

- A stack tároló a vector, deque, vagy list védett objektumát tartalmazza, és ennek a megfelelő tagfüggvényeit hívja meg.
- Az előző program így módosul:

```
#include <iostream>
#include <stack>
using namespace std;
```

A fő függvény

```
int main() {
    int n = 10;
    stack<int> v;
    for (int i = 0; i < n; i++)
        v.push(i);
    while (!v.empty()) {
        cout << v.top() << " ";
        v.pop();
    }
}
```

Kimenet: 9 8 7 6 5 4 3 2 1 0

A stack másképp

- vector illetve list adatszerkezetre alapozva:

```
stack<int, vector<int> > v;
```

- vagy

```
stack<int, list<int> > v;
```

- A vector illetve list fejállományokat be kell ékelni.
- A kimenet ugyanaz lesz.

17.4. Bejárók

- A bejáró a mutató fogalmának az általánosítása.
- A bejárók segítségével végighaladhatunk egy tároló elemein. A szolgáltatott elemekkel különböző műveleteket végezhetünk el.
- A bejárók osztályozása
- Műveletek bejárókkal

A bejárók osztályozása

- Bemeneti (input)
- Kimeneti (output)
- Előre haladó (léptető, forward)
- Kétirányú (bidirectional)
- Közvetlen elérésű (direkt hozzáférésű, random-access)

Tárolók, bejárók és algoritmusok

- A tároló osztály meghatározza azokat a bejáró-fajtákat, amelyek használhatók az illető tároló esetén.
- A szabványos sablon könyvtár minden algoritmusának a leírása tartalmazza mindazokat a tároló-, illetve bejáró-kategóriákat, amelyekre alkalmazható az algoritmus.

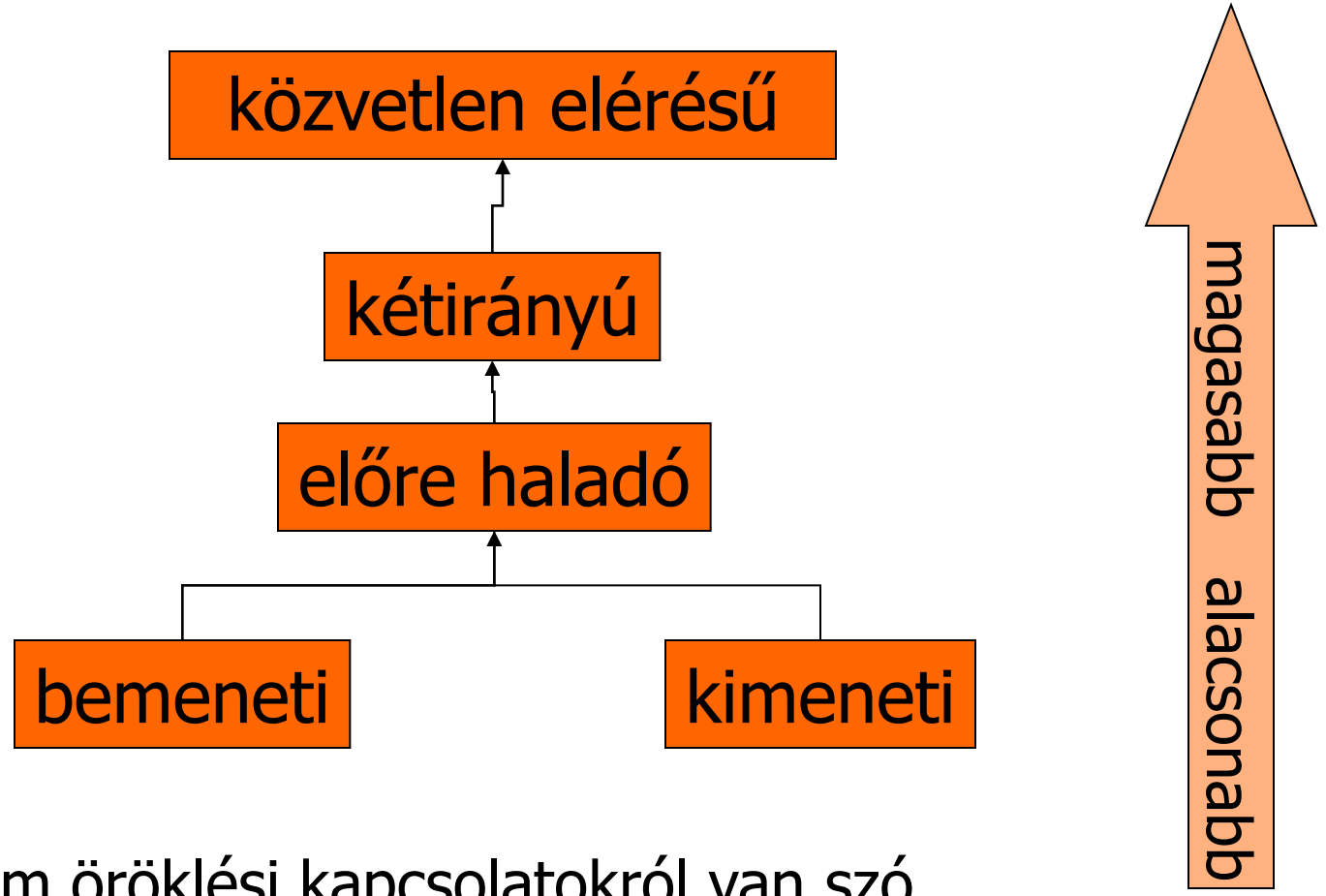
Tárolók bejárói

- vector<T>::iterator közvetlen elérésű
- deque<T>::iterator közvetlen elérésű
- list<T>::iterator kétirányú
- rendezett asszociatív tárolók kétirányú

Kapcsolat a bejárók között

- Az alacsonyabb rendű bejárók mindig helyettesíthetők magasabb rendű bejárókkal.
- A legmagasabb rendű a közvetlen elérésű bejáró.
- Egy objektumra hivatkozó mutatót közvetlen elérésű bejárónak tekinthetünk, ezért ez tetszőleges bejárót helyettesíthet.

Helyettesíthetőség



- nem öröklési kapcsolatokról van szó

Előre haladó bejáró

- Egy adott irányban lehet vele végigjárni a tároló objektumait.
- A léptetéshez a ++ operátort használjuk.

Bemeneti és kimeneti bejáró

- Az előre haladó bejáróhoz hasonló, de nem biztos, hogy annak az összes tulajdonságával rendelkezik.
- A bemeneti bejáróval adatokat tudunk bevinni a tárolóba.
- A kimeneti bejáróval adatokat olvashatunk ki a tárolóból.

Kétirányú bejáró

- A tároló objektumait nem csak előre haladva, hanem fordított irányban is bejárhatjuk.
- A ++ és -- operátorokat használjuk.

Közvetlen elérésű bejáró

- a kétirányú bejáró jellegzetességein kívül a mutatókra vonatkozó műveletek is alkalmazhatók:
- bejáró növelése (csökkentése) egy egész számmal (+= és -=)
- egész szám hozzáadása és kivonása (+ és -)
- bejárók különbsége (az eredmény egy egész szám)
- összehasonlítás (< > <= >=)

Egyenlőség és különbözőség

- A == és != operátorok a kimeneti bejárón kívül minden egyes bejáróra alkalmazhatók.

Példa közvetlen elérésű bejáróra

```
#include <vector>
using namespace std;
int main() {
    unsigned n;
    cout << "n = ";
    cin >> n;
    vector<int> vect(n);
```

Bejárás

```
    cout << "A vektor merete: " << vect.size() << endl;
    for (unsigned k = 0; k < n; ++k)
        vect[k] = k * k;
    cout << "Az elemek:\n";
    for (vector<int>::iterator i = vect.begin();
                                    i != vect.end(); ++i)
        cout << " " << *i;
    cout << endl;
}
```

Lehetséges kimenet

n = **10**

A vektor merete: 10

Az elemek:

0 1 4 9 16 25 36 49 64 81

Bejárás fordított sorrendben

```
cout << "Az elemek forditott sorrendben:\n";
for (vector<int>::reverse_iterator i = vect.rbegin();
                                   i != vect.rend(); ++i)
        cout << " " << *i;
cout << endl;
```

- A vector esetén az iterator és a reverse_iterator közvetlen elérésű bejáró.

Nagyobb lépések

```
cout << "Minden harmadik elem:\n";
for (vector<int>::iterator i = vect.begin(); i < vect.end();)
{
        cout << " " << *i;
        if (vect.end() - i > 3) i += 3;
        else break;
}
cout << endl;
```

- Ez a művelet csak közvetlen elérésű bejáró esetén használható.

Példa kétirányú bejáróra

```
#include <iostream>
#include <iomanip>
#include <list>
using namespace std;
int main() {
    unsigned n;
    cout << "n = ";
    cin >> n;
    list<int> lista;
```

Hozzáadás a listához

```
cout << "A lista merete: "
     << lista.size() << endl; //méret: 0
cout << "Elemeket adunk hozza.\n";
for (unsigned k = 0; k < n; ++k)
     lista.push_back(k);
 //lista[k] = k;   // hiba: list-re nem alkalmazható
cout << "A lista merete: "
     << lista.size() << endl;   // méret: n
```

Bejárás

```
for (list<int>::iterator i = lista.begin();
                                    i != lista.end(); ++i)
   cout << " " << *i;
cout << endl;
cout << "Az elemek forditott sorrendben:\n";
for (list<int>::reverse_iterator i = lista.rbegin();
                                    i != lista.rend(); ++i)
   cout << " " << *i;
```

Bejárás fordított sorrendben (más módszerrel)

```
    cout << "\nAz elemek forditott sorrendben:\n";
    list<int>::iterator j = lista.end();
    while (j != lista.begin())
        cout << " " << *--j;
    cout << endl;
}
```

- A list egy i bejárója esetén például az i += 3 nem alkalmazható, mivel kétirányú, és nem közvetlen elérésű bejáróról van szó.

Lehetséges kimenet

n = 10
A lista merete: 0
Elemeket adunk hozza.
A lista merete: 10
Az elemek:
 0 1 2 3 4 5 6 7 8 9
Az elemek forditott sorrendben:
 9 8 7 6 5 4 3 2 1 0
Az elemek forditott sorrendben:
 9 8 7 6 5 4 3 2 1 0

17.5. Memóriafoglalók

- A memória kezelését (lefoglalását és felszabadítását) valósítja meg (nem kell ezt maga a tároló, vagy az algoritmus elvégezze).
- Szabványos felületet biztosítanak a memóriakezelési műveletekre.
- Szabványos memóriafoglaló: allocator. A legtöbb alkalmazáshoz ez megfelelő lesz.

A szabványos memóriafoglaló

- Az allocator sablont a <memory> fejállományban vezetik be.
- A lefoglalásra a new operátort használja.
- A delete segítségével végzi a felszabadítást.
- Felhasználói memóriafoglaló is definiálható.

Az allocator felülete

```
template <class T> class std::allocator {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;
    pointer address(reference r) const;
```

Az allocator felülete

```
  const_pointer address(const_reference r) const;
  allocator();
  template <class U> allocator(const allocator<U>&);
  pointer allocate(size_type n, const void* hint = 0);
  void deallocate(pointer p, size_type n);
  void construct(pointer p, const T& val);
  void destroy(pointer p);
  size_type max_size() const;
  template <class U>
  struct rebind { typedef allocator<U> other; };
};
```

17.6. Karakterláncok és számok

- Karakterláncok:
  - A basic_string osztálysablon
  - A string és wstring tároló
  - A basic_string tagjai
- Számok
  - numeric_limits
  - cmath
  - valarray

A basic_string osztálysablon

```
template <
class CharType,
class Traits = char_traits<CharType>,
class Allocator = allocator<CharType> >
```

- CharType: az elemek típusa (char vagy wchar_t szokott lenni)
- Traits: a CharType jellemzőit adja meg
- Allocator: memóriafoglaló

A string és wstring tároló

- Megvalósítás:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

A basic_string tagjai

- Típusdeklarációk
- Tagfüggvények
- Operátorok

Típusdeklarációk

- iterator: közvetlen elérésű bejáró
- reverse_iterator: visszafelé haladó bejáró
- const_iterator: egy const elem elérésére
- const_reverse_iterator: const elem visszafelé haladó bejáróval
- pointer: mutató egy elemhez
- reference: referencia egy elemre

Tagfüggvények

- append: karaktereket fűz hozzá a végéhez
- begin: az első elemre hivatkozó bejáró
- clear: törli az összes elemet
- compare: összehasonlít két karakterláncot
- copy: egy hagyományos tömbbe másol
- data: a karakterláncot egy hagyományos tömbbé alakítja
- empty: ellenőrzi, hogy üres-e a karakterlánc

Tagfüggvények

- end: az utolsó utáni pozícióra hivatkozó bejáró
- erase: kitöröl egy elemet, vagy egy tartománynak megfelelő elemet
- find: megkeresi egy részlánc első előfordulását
- find_first_of: megkeresi egy tömb egyik karakterének első előfordulását
- find_last_of: megkeresi egy tömb egyik karakterének utolsó előfordulását
- insert: beszúrás
- length: hossz

Tagfüggvények

- push_back: hozzáad a végére egy karaktert
- rbegin: az első elemre hivatkozó visszafelé haladó bejáró (reverse_iterator).
- rend: az utolsó utáni elemre hivatkozó visszafelé haladó bejáró (reverse_iterator).
- size: méret
- substr: egy részláncot térít vissza
- swap: kicseréli két karakterlánc tartalmát

Operátorok

- += operátor: hozzáfűz a karakterlánchoz
- = operátor: értékadás
- [] operátor: egy adott indexű elemhez térít vissza referenciát
- A string esetén egyéb operátorok is használhatók (wstring esetén már nem):

összehasonlító operátorok: == != < > <= >=

bemeneti/kiviteli műveletek: << >>

összefűzés: +

Műveletek hagyományos tömbökkel

- A tömb mérete fordításkor rögzítve lesz.
- Az összefűzést és átmásolást a strcat és strcpy függvényekkel végezhetjük.

```
#include <iostream>
#include <cstring>
using namespace std;
```

A fő függvény

```
int main() {
    char s[20]; char t[20]; char w[40];
    cout << "s = "; cin >> s;
    cout << "t = "; cin >> t;
    strcpy(w, s);
    strcat(w, t);
    cout << "w = " << w << endl;
    cout << "s = " << s << endl;
    cout << "t = " << t << endl;
}
```

A string osztállyal

- A karakterlánc mérete futási időben változhat.
- Az összefűzést és átmásolást a + illetve az = operátorokkal végezzük.

```
#include <string>
#include <iostream>
using namespace std;
```

A fő függvény

```
int main() {
    string s, t, w;
    cout << "s = "; cin >> s;
    cout << "t = "; cin >> t;
    w = s + t;
    cout << "w = " << w << endl;
    cout << "s = " << s << endl;
    cout << "t = " << t << endl;
}
```

Számok

- numeric_limits
- cmath
- valarray
- complex
- ratio

A valarray osztály

- Vektorműveletek számokkal (gyorsabb, hatékonyabb mint a vector)
- Típusdeklaráció:
  - value_type: az elemek típusa
- Tagfüggvények
- Operátorok (pl. op= ahol op egy bináris aritmetikai, vagy bitenkénti operátor).

Tagfüggvények

- apply: egy adott függvényt alkalmaz az összes elemre
- cshift: ciklikus eltolása az elemeknek
- shift: eltolás
- max: legnagyobb elem
- min: legkisebb elem
- size: méret
- sum: összeg

Példa (valarray)

```
// Elemenkénti hozzáadás a += operátorral.
// Az összes elem négyzetreemelése
// az apply tagfüggvény segítségével.
#include <valarray>
#include <iostream>
#include <time.h>
using namespace std;
```

A kiir függvénysablon

```
template <class T>
void kiir(const char* s, valarray<T>& w)
{
   cout << s << endl;
   for (unsigned i = 0; i < w.size(); ++i)
        cout << w[i] << " ";
   cout << endl;
}
```

Az f függvény

```
int f(int x)
{
   return x * x;
}
```

A fő függvény

```
int main() {
    valarray<int> v(10);
    valarray<int> z(10);
    valarray<int> t;
    srand((unsigned)time(0));
    for (int i = 0; i < 10; ++i)
        v[i] = rand() % 10;
    for (int i = 0; i < 10; ++i)
        z[i] = i;
```

Műveletek

```
    kiir("Elso valarray:", v);
    kiir("Masodik valarray", z);
    v += z;
    kiir("Az elsohoz hozzaadva a masodikat:", v);
    t = v.apply(f);
    kiir("Negyzetreemelve:", t);
}
```

Lehetséges kimenet

Elso valarray:

7 8 3 6 5 5 0 8 6 5

Masodik valarray

0 1 2 3 4 5 6 7 8 9

Az elsohoz hozzaadva a masodikat:

7 9 5 9 9 10 6 15 14 14

Negyzetreemelve:

49 81 25 81 81 100 36 225 196 196

Más példa (atan)

```
#include <iostream>
#include <cmath>
#include <valarray>
using namespace std;
int main() {
    valarray<double> v_tomb(3);
    valarray<double> eredmeny;
    v_tomb[0] = 1;
    v_tomb[1] = sqrt(3);
    v_tomb[2] = sqrt(3) / 3;
```

Az atan függvény „valarray”-re

```
    eredmeny = atan(v_tomb);
    //eredmeny = v_tomb.apply( atan );  // ugyanaz
    cout.precision(17); // pontossag: 17
    int t[] = { 4, 3, 6 };
    for (int i = 0; i < 3; i++)
        cout << t[i] * eredmeny[i] << endl;
    cout << typeid(eredmeny).name() << endl;
}
```

Az eredmény

3.141592653589793

3.141592653589793

3.141592653589793

class std::valarray<double>