# Bevezetés a programozásba 2

7. Előadás: STL konténerek, sablonok

http://digitus.itk.ppke.hu/~flugi/

# Vector

```
int main() {
   vector<int> v(10);
   …
   int sum=0;
   for (int i=0;i<v.size();i++) {
      sum+=v[i];
   }
   cout << ”osszeg:” << sum << endl;
}
```

# Általános konténer

- Elemeket tartalmaz
- A konténer deklarációjánál az elemek típusa megadható
  - template
- Most használni tanuljuk meg a konténereket
  - Írni majd adatszerkezetek tárgyból lesz feladat

# STL

- Standard Template Library
- A C++ szabvány része
- Típussal paraméterezhető konténereket és algoritmusokat tartalmaz
- A legtöbb modern programozási nyelvben létezik megfelelője

# Vector

```
template<typename _Tp, … >
class vector … {

  …

  int size() const {  …  }
  _Tp& operator[](int __n) {  … }

  …

};
```

Közelítő leírás, hogy kiférjen. A pontos definíciót megtalálod a fordítóprogramod könyvtáraiban, ami tartalmaz még néhány olyan technikát, amit még nem tanultunk

# Konténerek műveletei

- Következő elem olvasása, írása
- Tetszőleges elem olvasása, írása
- Méret, üres-e
- Elem hozzáadása
  - elejére, végére, bárhova
- Elem törlése
  - elejéről, végéről, bárhonnan

# Konténerek műveletei

- Következő elem olvasása, írása
- Tetszőleges elem olvasása, írása

v[ i ]

- Méret, üres-e

v.size()

v.empty()

- Elem hozzáadása
  - elejére, végére, bárhova

v.push_back(e)

- Elem törlése
  - elejéről, végéről, bárhonnan, mindet

v.pop_back(e)

# vector hatékonysága

- Következő elem olvasása, írása
- Tetszőleges elem olvasása, írása

v[ i ]

- Méret, üres-e

v.size()  v.empty()

- Elem hozzáadása
  - elejére, végére, bárhova

v.push_back(e)

- Elem törlése
  - elejéről, végéről, bárhonnan, mindet

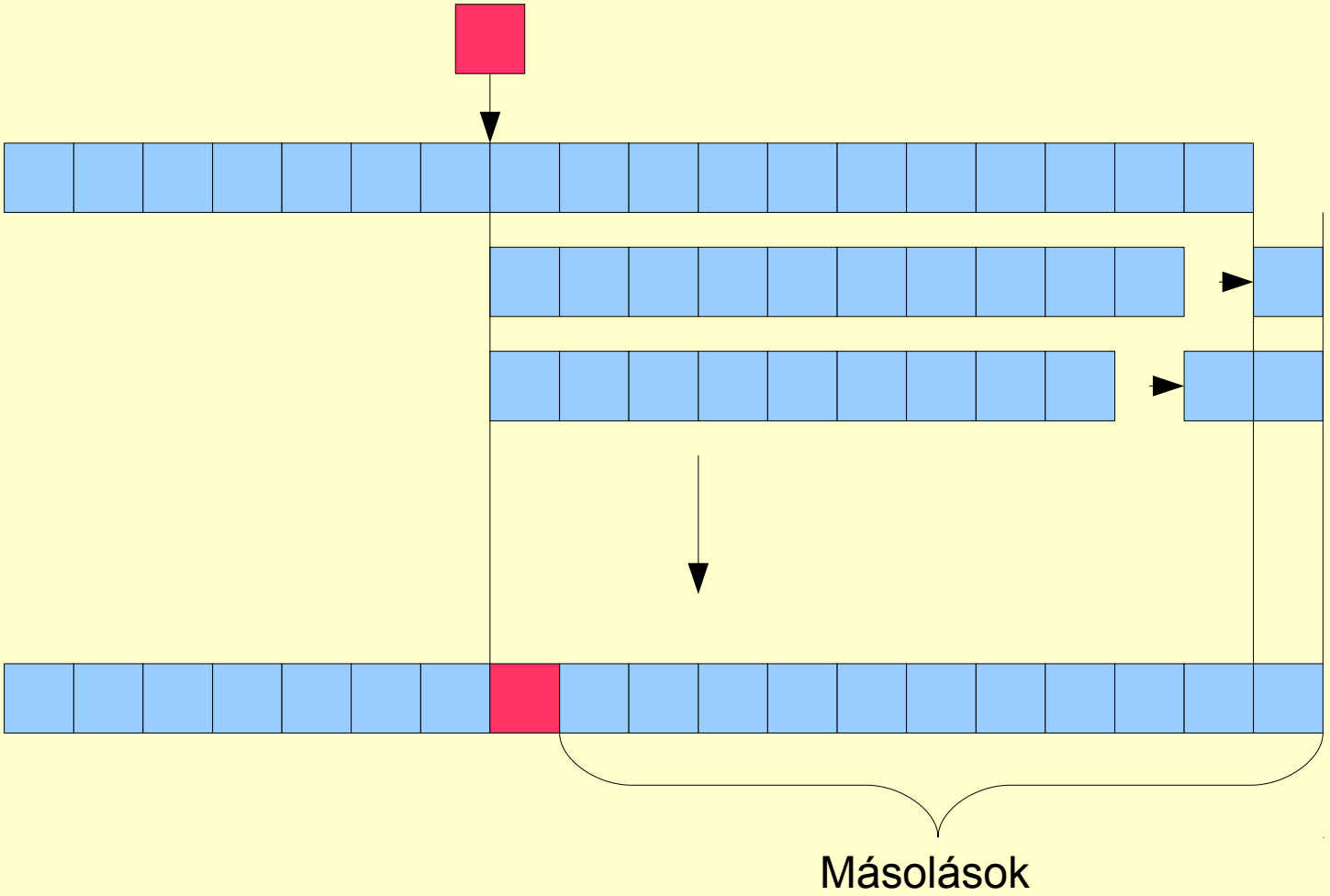
v.pop_back(e)

jelmagyarázat: konstans  logaritmikus  lineáris  nincs

# Hatékonysági kategóriák

- Konstans: fordítási időben ismert fix számú lépés
- Függő kategóriák: valamilyen n paramétertől függ a lépésszám, tipikusan elemszámtól, az alábbi képletekkel *legfeljebb konstansszorosan* becsülhető
  - logaritmikus: log n lépés
  - lineáris: n lépés
  - négyzetes, polinomiális: $n^2$, $n^p$ lépés
  - exponenciális: $2^n$ lépés

# Példa: beszúrás vektorba



# Vektor

- A tetszőleges helyre beszúrás lineáris idejű, amilyen hosszú a vektor, annyi ideig tart
- A végére beszúrás átlagosan logaritmikus idejű
  - Mivel mindig kétszerezi a kapacitást, n beszúrás log2 n átméretezést jelent, nincs szükség másolásra
- Lekérdezés, elem olvasása egy lépéses művelet

# Az STL konténerei

- Szekvenciális konténerek
  - vector
  - list
- Asszociatív konténerek
  - map
  - set
- Adapterek
  - queue
  - stack

# A list

- Minden eleme tudja, ki a következő
- Az n-edik elem megkeresése tehát lassú, n lépést igényel
- Beszúrás, törlés viszont bárhol egy lépéses művelet
- Iterátorokkal kell kezelni:
- ```
  list<int> l;
  for (list<int>::iterator
  it=l.begin();it!=l.end();++it){
    cout << *it;
  }
  ```
  ```

# Lista implementáció

- A (láncolt) listák lényege, hogy minden elem tartalmaz egy mutatót a következő elemre
- struct elem {
  T adat;
  elem * kov;
  };
- Az utolsó elem általában nullmutatót tartalmaz, vagy körbeláncol az első elemre.
- Beszúrás, törlés: néhány mutatót kell módosítani

# Iterátorok

- Konténerek közös szintaxisú bejáró típusa
- ```
  list<int> l;
  for (list<int>::iterator
  it=l.begin();it!=l.end();++it){
    cout << *it;
  }
  ```
- ```
  vector<int> v;
  for (vector<int>::iterator
  it=v.begin();it!=v.end();++it){
    cout << *it;
  }
  ```

# Konténerek

- Szekvenciális
  - Értelmezhető az elem indexe, sorszáma
  - Fontos lehet a sorrend az elemek között
- Asszociatív
  - set: nincs rendezettség, csak az a fontos, hogy egy elem létezik-e, vagy sem
  - map: kulcs-érték párok, az értéket nem index, hanem kulcs alapján keressük

# Asszociatív konténerek: map

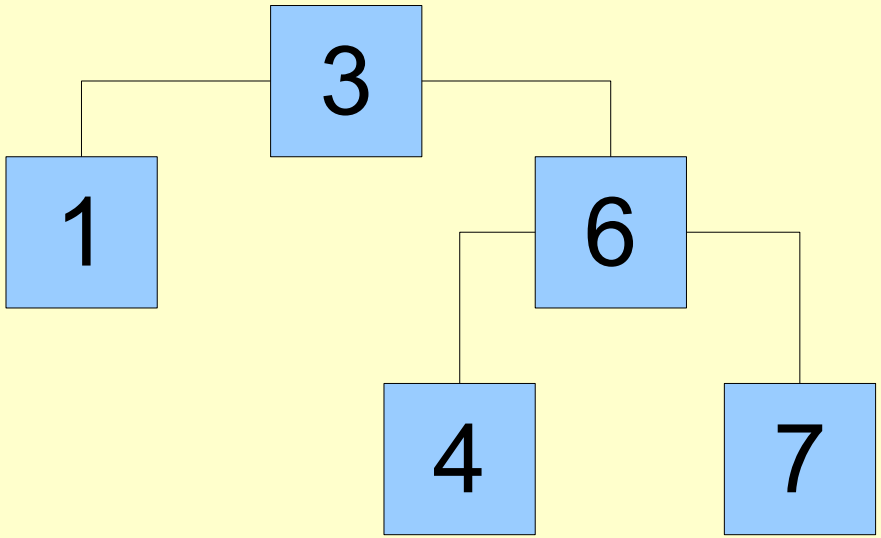
```
int main() {
  map <string, int> honapokhossza;
  honapokhossza[”január”]=31;
  …
  honapokhossza[”december”]=31;
  string s; cin >> s;
  map<string,int>::iterator it;
  it = honapokhossza.find(s);
  if (it!=honapokhossza.end()) {
    cout << honapokhossza.at(s);
  } else {
    cout << ”nincs ilyen hónap”;
  }
}
```

# Asszociatív konténerek

- Elem/kulcs létezésének eldöntése logaritmikus sebességgel történik
- Elem hozzáadása és törlése is logaritmikus idejű
- Nincs kitüntetett sorrend, bár végigjárható minden elem, a sorrend nem mindig definiált
- Nagyon rugalmas eszköz, apró veszélyekkel
  - honapokhossza[”blabla”] lekérés mellékhatása egy beszúrás definiálatlan elemre, ”blabla” kulccsal, fontos a find() és at() használata

# Asszociatív konténerek

- A kulcs típusa bármi lehet, amire létezik operator< művelet
- Ennek hátterében egy rendezőfa nevű szerkezet van



# Konténerekhez illő tipikus feladatok

- vector: fix, indexelt adatok, tetszőleges hozzáférés, pl.: logaritmikus keresés
- list: nagyon változó számú, fontos sorrendbeli adatok, főleg helyi műveletekkel, pl.: szövegszerkesztőben a szöveg sorai
- set: elemek beszúrására, törlésére, létezésének eldöntésére van szükség, pl.: egy fájl azon szavai, amik többször is előfordulnak
- map: kulcshoz rendelhető értékek halmaza, pl.:angol-magyar szótár `map<string,string>`

# Adatszerkezetek

- A konténerek megvalósításának tudománya az adatszerkezetek és algoritmusok
- Mi csak használjuk ezeket, ehhez viszont érdemes tudni, hogy melyik miben jó, amihez segít tudni, hogyan működik
- A konténerek kombinálhatóak, a kombinációk hatékonyságának végiggondolásához is szükség van erre az ismeretre
  - `vector<list<int> > listakvektora;`

# Hatékonyság táblázat

| | vector | list | map |
|---|---|---|---|
| elem elérés | 1 | n | log n |
| beszúrás | n | 1 | log n |
| törlés | n | 1 | log n |

# Konténerek használata

- A feladat specifikációjából megsejthető, mely műveletek milyen gyakoriságúak, ez kellőképpen meghatározza a hatékony választások körét
- Itt is igaz a „keep it simple” szabály: ha valamit egyszerűen is elég jól meg lehet csinálni, érdemes úgy csinálni
  - legfeljebb kiderül, hogy nem elég gyors, akkor átalakítjuk, de addig is működik
- A konténerek a tagfüggvényeiken keresztül kezelhetőek, van dokumentáció

# STL algoritmusok

- Az STL konténerek egy része feltételezi, hogy létezik néhány függvény a paraméterül adott típusokra
  - például a map a kulcsoknál az operator<()
- Ennek segítségével megoldhatóak bonyolultabb algoritmusok, a konténer fajtájának rögzítése nélkül

# sort

```
#include <algorithm>

int main() {
    vector<int> v;
    for (int i=0;i<20;i++) v.push_back(rand()%100);

    for (vector<int>::iterator it=v.begin();
                                it!=v.end();it++) {
        cout << *it << " ";
    }
    cout << endl;

    sort(v.begin(),v.end());

    for (vector<int>::iterator it=v.begin();
                                it!=v.end();it++) {
        cout << *it << " ";
    }
}
```

# Template osztály szintaxis

```
template <typename T>
class TC {
  T mező;
  T fv(T a);
};

  ...
TC<int> tci;
```

Az eddigiekkel ellentétben ez a T tényleg így van a kódban

# Template példa

```
template <typename T>
class Tomb {
   T *_m; int _s;
public:
   Tomb(int s) : _s(s) { _m=new T[_s];}
   ~Tomb() {delete[] _m;}
   T operator[](int i) const {return _m[i];}
};
...
Tomb<int> t(10); int k = t[1];
Tomb<string> t2(10); string s = t2[1];
```

# A template működése

- Szöveg szintű helyettesítés
  - A szintaktikai hibák előre kiszűrhetőek, a szemantikaiak nem
  - Az implementációt nem lehet elrejteni, mert ha el van rejtve, akkor a helyettesítést nem lehet elvégezni
  - Ezért előre lefordítani sem lehet.
- A fordítás előtti helyettesítés csak akkor történik meg, ha meghivatkozzuk az adott template-t illetve annak adott metódusát.

Fontos:

Minden template implementációját a fejlécfájlba kell tenni. Nem lehet elrejteni.

(elméletileg az export kulcsszó segítségével meg lehet tenni. Az elmélet és a gyakorlat között elméletileg nincs különbség..)

# Template osztály szintaxis

```
template <typename T>
class TC { ...
  void fv(T t) {
    T x=t%10;
  } ...
};
TC<int> tci; tci.fv();
TC<string> tci; tci.fv();
```

Ha olyan műveletet használunk, ami a paraméter T típusnak nincs, hibát kapunk.

Ezzel meg lehet szorítani a paramétereket

OK

HIBA

# template függvény

```
template <typename T>
T maxt(const T& a, const T& b)
{
  return a > b ? a : b;
}
...
char k = maxt('a','b');
int i = maxt(3,4);
```

# template függvény

```
template <typename T>
T maxt(const T& a, const T& b)
{
   return a > b ? a : b;
}
...
char k = maxt('a','b');
int i = maxt(3,4);
```

# Tervezés template osztályokkal

- Mit érdemes típussal paraméterezni?
  - Aminek a kódja nagyon hasonló sokféle típusra
    - Konténer osztályok: vector (, list, map, stb...)
    - Algoritmusok (#include <algorithm>)
  - Amiben az egyes műveletek túlterhelése jelent csak különbséget
    - ```
      template <typename T> T max(T a, T b) {
          return a>b ? a:b;
      }
      ```
- Mire alkalmas még? Nehezen végiggondolható lehetőségek: template metaprogramming
      ```

# map példa

```
int main() {
   map<string, int> m;
   m["aa"]=1;
   m["bb"]=2;
   m["cc"]=3;
   for (map<string, int>::iterator it=m.begin()
     ;it!=m.end();it++)
   {
       cout << it->first << " "
            << it->second << endl;
    }
}
```

# map kulcs típushoz kell operator<

```
struct A {
    A(int n):a(n){}
    int a;
};
bool operator<(A a, A b) {
    return a.a<b.a;
}
int main() {
    map<A, int> m;
    m[A(1)]=4;  m[A(2)]=3;
    m[A(3)]=2;  m[A(4)]=1;
    for (map<A,int>::iterator it=m.begin();
    it!=m.end(); it++) {
        cout << it->first.a << " "
             << it->second << endl;
    }
}
```

# STL algoritmusok

- Az STL konténerek egy része feltételezi, hogy létezik néhány függvény a paraméterül adott típusokra
  - például a map a kulcsoknál az operator<()
- Ennek segítségével megoldhatóak bonyolultabb algoritmusok, a konténer fajtájának rögzítése nélkül

# sort

```
#include <algorithm>

int main() {
    vector<int> v;
    for (int i=0;i<20;i++) v.push_back(rand()%100);

    for (vector<int>::iterator it=v.begin();
                                it!=v.end();it++) {
        cout << *it << " ";
    }
    cout << endl;

    sort(v.begin(),v.end());

    for (vector<int>::iterator it=v.begin();
                                it!=v.end();it++) {
        cout << *it << " ";
    }
}
```

# sort

```
struct A {
    A(int n):a(n){}
    int a;
};
bool operator<(A a, A b) {
    return a.a<b.a;
}
int main() {
    vector<A> v;
    for (int i=0;i<20;i++) v.push_back(A(rand()%100));

    for (vector<A>::iterator it=v.begin(); it!=v.end();it++) {
        cout << it->a << " ";
    }
    cout << endl;

    sort(v.begin(),v.end());

    for (vector<A>::iterator it=v.begin(); it!=v.end();it++) {
        cout << it->a << " ";
    }
}
```

# Algoritmusok

- Bejáró, nem módosító algoritmusok
  - for_each, find, count, search, …
- Rendezések
  - sort, stable_sort, merge, …
- Sorozatmódosítások
  - swap, fill, copy, replace, random_shuffle, …
- Speciális
  - binary_search, sort_heap, next_permutation, …

# for_each

```
int main() {
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), mit_kell_csinalni);

}
```

# for_each

```
void mit_kell_csinalni(int a) {
   cout << a << " ";
}

int main() {
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), mit_kell_csinalni);

}
```

# for_each

- A harmadik paraméter egy függvény
- A template technika következménye, hogy bármi jó, ami mögé zárójelbe téve ott van a paraméter
  - Tehát a függvény *neve* a paraméter
  - Ez nem függvénymutató, a függvénymutató egy bizonyos típus egy adott szignatúrához.

# Függvénymutató vs template függvény paraméter

- Ez nem függvénymutató, hanem template paraméter, a függvénymutató egy bizonyos típus egy adott szignatúrához.
  - template <typename Fun>
    void fv(Fun f){
      f();
    }
  - void fv(void (*f)()) {
      f();
    }

# for_each és a funktor

- Funktor: olyan osztály vagy objektum, aminek meg van valósítva az operator() ( … ) operátora
- Tehát az objektum neve után zárójelet írva ez az operátor hívódik meg
- Ezzel „okos” függvényeket lehet csinálni, amik
  - mezőket tartalmazhatnak
  - konstruktorparamétert vehetnek át

# for_each

```
struct Mit_kell_csinalni {
    void operator()(int a) {
        cout << a << " ";
    }
};

int main() {
   Mit_kell_csinalni mit_kell_csinalni;
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), mit_kell_csinalni);

}
```

# for_each

```
struct Mit_kell_csinalni {
    Mit_kell_csinalni() {index=0;}
    void operator()(int a) {
        cout <<index <<":"<< a << "  ";
        index++;
    }
    int index;
};
int main() {
   Mit_kell_csinalni mit_kell_csinalni;
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), mit_kell_csinalni);
}
```

# for_each

```
struct Mit_kell_csinalni {
    Mit_kell_csinalni(int kezdo) {index=kezdo;}
    void operator()(int a) {
        cout <<index <<":"<< a << "   ";
        index++;
    }
    int index;
};
int main() {
   Mit_kell_csinalni mit_kell_csinalni(5);
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), mit_kell_csinalni);
}
```

# for_each

```
struct Mit_kell_csinalni {
    Mit_kell_csinalni(int kezdo) {index=kezdo;}
    void operator()(int a) {
        cout <<index <<":"<< a << "   ";
        index++;
    }
    int index;
};
int main() {
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   for_each (v.begin(), v.end(), Mit_kell_csinalni(5));
}
```

# for_each

- Mi a jó benne?
  - független a konténertől
  - biztosan működik, nem hagy ki vagy jár be kétszer elemeket
    - saját megoldásoknál ez a veszély fennáll
- Mi a hátrány?
  - Kell egy külső függvény, vagy funktor
    - A következő C++ szabványban már megoldható lesz a dolog külső függvény nélkül is
  - Tehát kicsit többet kell gépelni

# kitérő: C++11

- lambda függvények:
  - for_each(v.begin(), v.end(), [&sum](int elem){
    sum+=elem;
    });
- auto kulcsszó új jelentése
  - for(auto it=l.begin();it!=l.end();it++) {…}
- g++ -std=c++11  (4.5-)

# STL algoritmusok

- A sort és a for_each példáján keresztül minden STL algoritmus használata elsajátítható
- Közös jellemzők:
  - nem változtatják meg a konténer szerkezetét
    - a változtatások mindig értékek szintjén történnek
    - elemszámuk sem változhat
  - iterátorokon keresztül lehet elérni a konténereket

# Extra iterátorok

```
#include <iterator>

int main() {
   vector<int> v;
   for (int i=0;i<20;i++) v.push_back(rand()%100);
   copy ( v.begin(), v.end(),
      ostream_iterator<int>(cout,", ") );
}
```

# Extra iterátorok

```
int main() {
   vector<string> v;
   ifstream f("main.cpp");
   copy(istream_iterator<string>(f),
        istream_iterator<string>(),
        back_insert_iterator<vector<string> >(v));
   copy(v.begin(), v.end(),
        ostream_iterator<string>(cout,", ") );
}
```