Programozás alapjai II.
(9. ea) **C++**

hibakezelés és STL bevezető

Szeberényi Imre

BME IIT

<szebi@iit.bme.hu>

MŰEGYETEM 1782

Hol tartunk?

- OO alapismeretek, paradigmák
  - egységbezárás (encapsulation)
    - osztályok (adatszerkezet + műveletek)
  - többarcúság (polymorphism)
    - műveletek paraméter függőek, tárgy függőek (kötés)
  - példányosítás (instantiation)
  - öröklés (inheritance)
  - generikus adatok és algoritmusok
- Konkrét C++ szintaxis

Mit ír ki?

```
struct Base {
  Base() { f(); }
  virtual void f() { cout << "Base::f()";}
  ~Base() { f(); }
};
struct Der : Base {
  Der() {}
  void f() { cout << "Der::f()"; }
};

int main() { Der d; return 0; } // ??
```

Konstruktor feladatai (ism.)

- Öröklési lánc végén hívja a virtuális alaposztályok konstruktorait.
- Hívja a közvetlen, nem virtuális alaposztályok konstruktorait.
- Létrehozza a saját részt:
  - beállítja a virtuális alaposztály mutatóit
  - beállítja a virtuális függvények mutatóit
  - hívja a tartalmazott objektumok konstruktorait
  - végrehajtja a programozott törzset

Mit ír ki? #2

```
struct Base {
  int a;
  Base(int i) :a(0) {}
  int f() const { return a; }
};
struct Der : Base {
  int b;
  Der() : Base( b = f()) { cout << b; }
};

int main() { Der d; return 0; } // ??
```

Ma

- Kivételkezelés
  - működés részletei,
  - hatása az objektumok élettartamára
  - Konstruktorban, destruktorban
- Létrehozás/megsemmisítés működésének felhasználása (auto_ptr, fájlkezeléshez)
- STL bevezető

Erőforrás foglalás-felszabadítás

- Gyakori, hogy erőforrásként kell kezelni valamit (memória, fájl, eszköz, stb.):
  - lefoglalás
  - feldolgozás
  - felszabadítás
- Az ilyen esetekben külön figyelmet kell fordítani arra, hogy a feldolgozás közben észlelt hiba esetén is gondoskodjunk a felszabadításról.

Erőforrás foglalás-felszabadítás/2

```
FILE *fp = fopen("x.txt", "r");          // megnyitás

try {
                                 // file feldolgozása
} catch (...) {
   fclose(fp);                   // lezárás hiba esetén
   throw;                        // tovább
}
fclose(fp);                      // lezárás normál esetben
```

Erőforrás foglalás-felszabadítás/3

```
class FILE_ptr {
   FILE *p;
public:
   FILE_ptr(const char *n, const char *m) {
        p = fopen(n, m); }
   ~FILE_ptr { fclose(p); }
   operator FILE*() { return p; }
};
```

cast operator

```
{
    FILE_ptr fp("x.txt", "r");
        fprintf(fp, "Hello");
}
```

automatikus cast

destruktor megszüntet (bezár)

Hiba és kivételkezelés

Hagyományos hibakezelési módszerek:

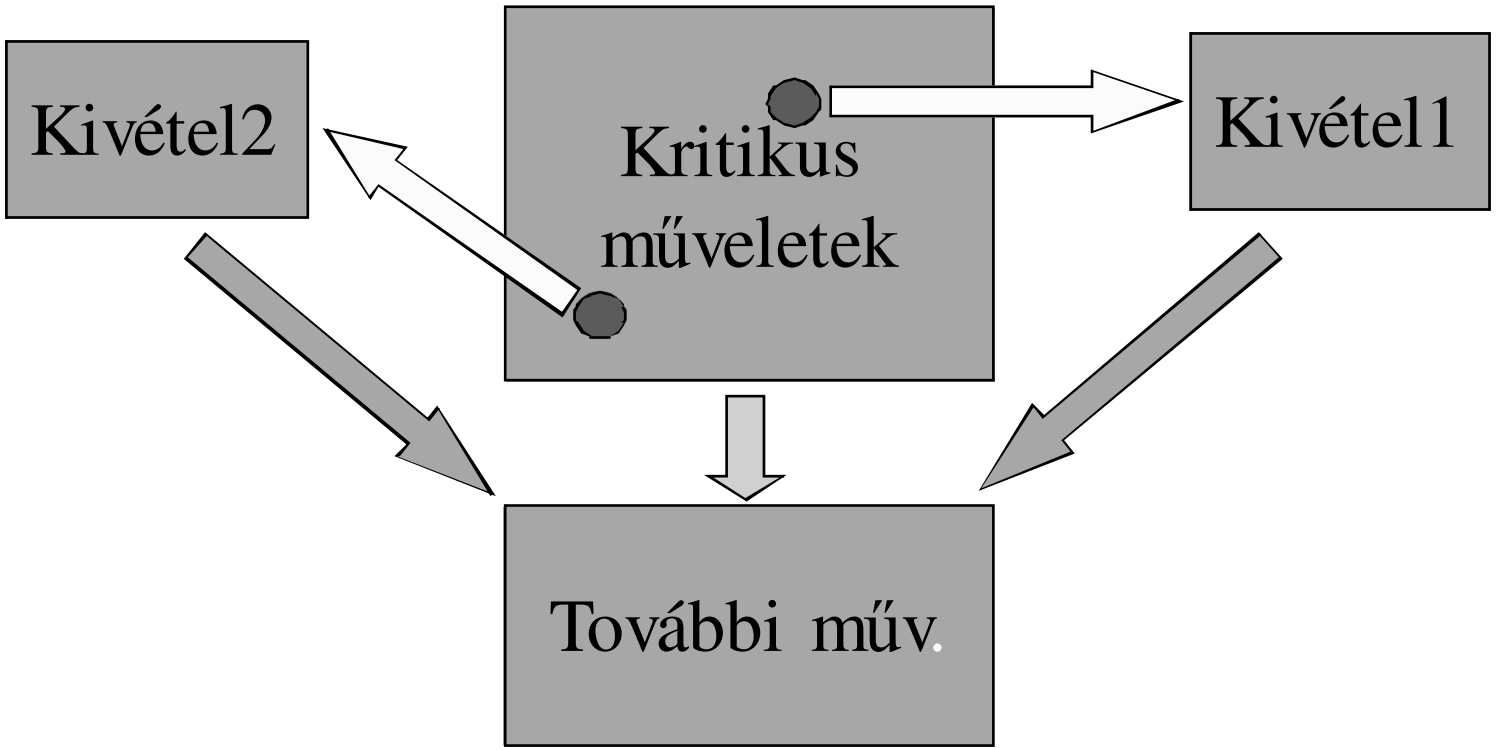
- Befejezi a program futását
- Hibakódot ad vissza
- Hibakezelő fv. meghívása

A C++ kivételkezeléssel a fentiek többé-kevésbé megoldhatók, sőt...

Kivételkezelés újból

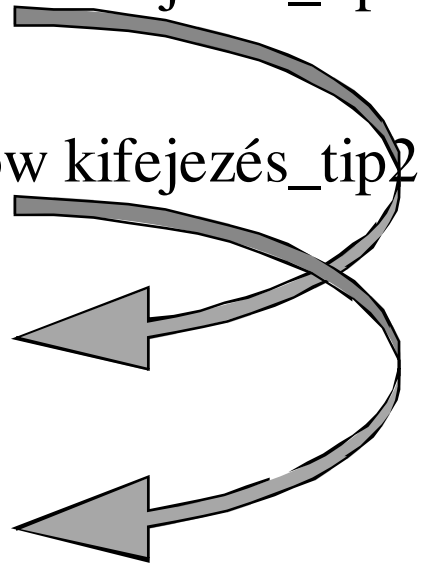
- Hibák/kivételek kezelése gyakran nem a hiba keletkezésének helyén történik.
- Legtöbbször a hiba keletkezésének helyén nem is tudjuk, hogy mit kell tenni, mert csak a programrészlet (függvény, könyvtár) felhasználója tudja ezt.
- Első C++ órán láttuk, hogy a hibát felfedező kódrészlet típusorientált hibát dobhat, amit a hívó képes feldolgozni.

Kivételkezelés = globális goto

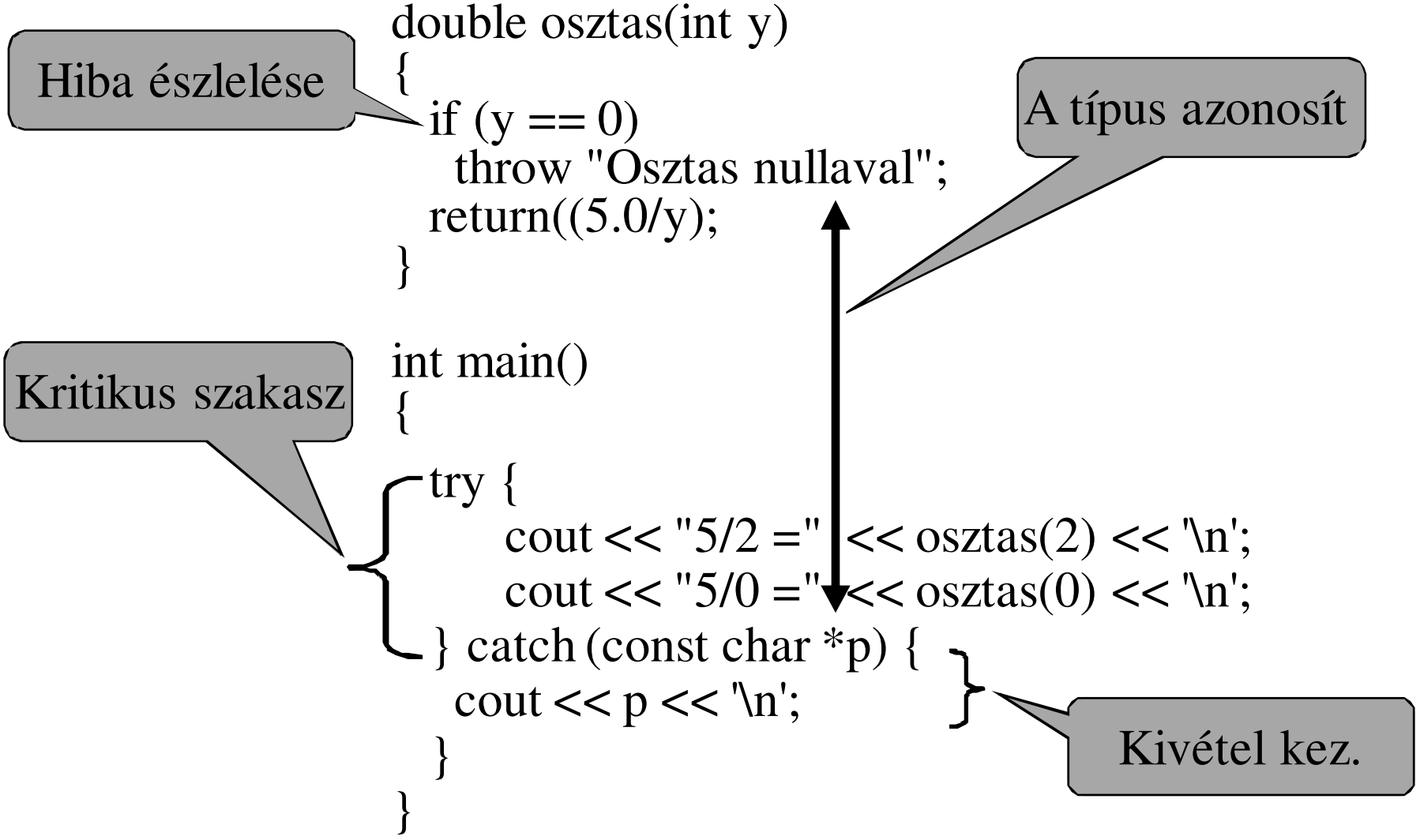


Kivételkezelés/2

```
try {
  ..... Kritikus művelet1
                if (hiba) throw kifejezés_tip1;
  ..... Kritikus művelet2
                if (hiba) throw kifejezés_tip2;
} catch (típus1 param) {
  ..... Kivételkezelés1
} catch (típus2 param) {
  ..... Kivételkezelés2
}
... további utasítások
```



Kivételkezelés példa



Újdonságok a korábbiakhoz

- A dobott kivétel alap ill. származtatott objektum is lehet.

```
try {
    throw E();
} catch(H) {
   // mikor jut ide ?
}
```

1. H és E azonos típusú,
2. H bázisosztálya E-nek,
3. H és E mutató és teljesül rájuk 1. vagy 2.,
4. H és E referencia és teljesül rájuk 1. vagy 2.

Következtetések

- Célszerű kivétel osztályokat alkalmazni, (pl. std::exceptions) amiből származtatással újabb kivételeket lehet létrehozni.
- A dobás értékparamétert dob, ezért az elkapáskor számolni kell az alaposztályra történő konverzióval (adatvesztés).

➔ Célszerű pointert, vagy referenciát alkalmazni.

➔ Kell másoló konstruktor.

Kivételek specifikálása

- Függvény deklarálásakor/definíciójakor megadható, hogy milyen kivételeket generál az adott függvény.
- Ha mást is generálna, akkor az automatikusan meghívja az unexpected() handlert.

```
void f1() throw (E1, E2);     // csak E1, E2
void f2() throw();            // semmi
void f3();                    // bármi
```

Továbbdobás

```
try {
    throw E();
} catch(H) {
    if (le_tudjuk kezelni) {
        ....
    } else {
        throw;
    }
}
```

Az eredeti dobódik tovább, nem csak az elkapott változat.

Paraméter nélkül

Minden elkapása

```
try {
    throw E();
} catch(...) {
    // szükséges feladatok
    throw;
}
```

A kezelők sorrendje fontos!

Roll back

```
try {
    A a;
    B b;
    C *cp = new C;
    if (hiba) throw "Baj van";
    delete cp;
} catch(const char *p) {
    // A létezik ?
    // B létezik ?
    // *cp által mutatott obj. létezik ?
}
```

Minden a blokkban deklarált, "létező" objektum destruktora meghívódik.

Hiba esetén C nem szabadul fel, de **cp** megszűnik.

Melyik obj. létezik ?

- Csak az az objektum számít létezőnek, amelynek a konstruktora lefutott.
- Ha a konstruktor nem fut le, akkor a roll back során a destruktor sem fog végrehajtódni.
- Előző példában C konstruktora lefutott ugyan, de nem deklarációval hoztuk létre, hanem dinamikusan.

Kivétel a konstruktorban

- Lényegében a kivételkezelés az egyetlen mód arra, hogy a konstruktor hibát jelezzen.
- Hiba esetén gondoskodni kell a megfelelő obj. állapot előállításáról.

Inicializáló listán keletkező kivétel elfogása:

```
class A {
        B b;
public:
        A() try
                :b() { // konstruktor programozott része
        } catch (...) { // kivételkezelés
        }
};
```

Kivétel a destruktorban

Destruktor hívás oka:

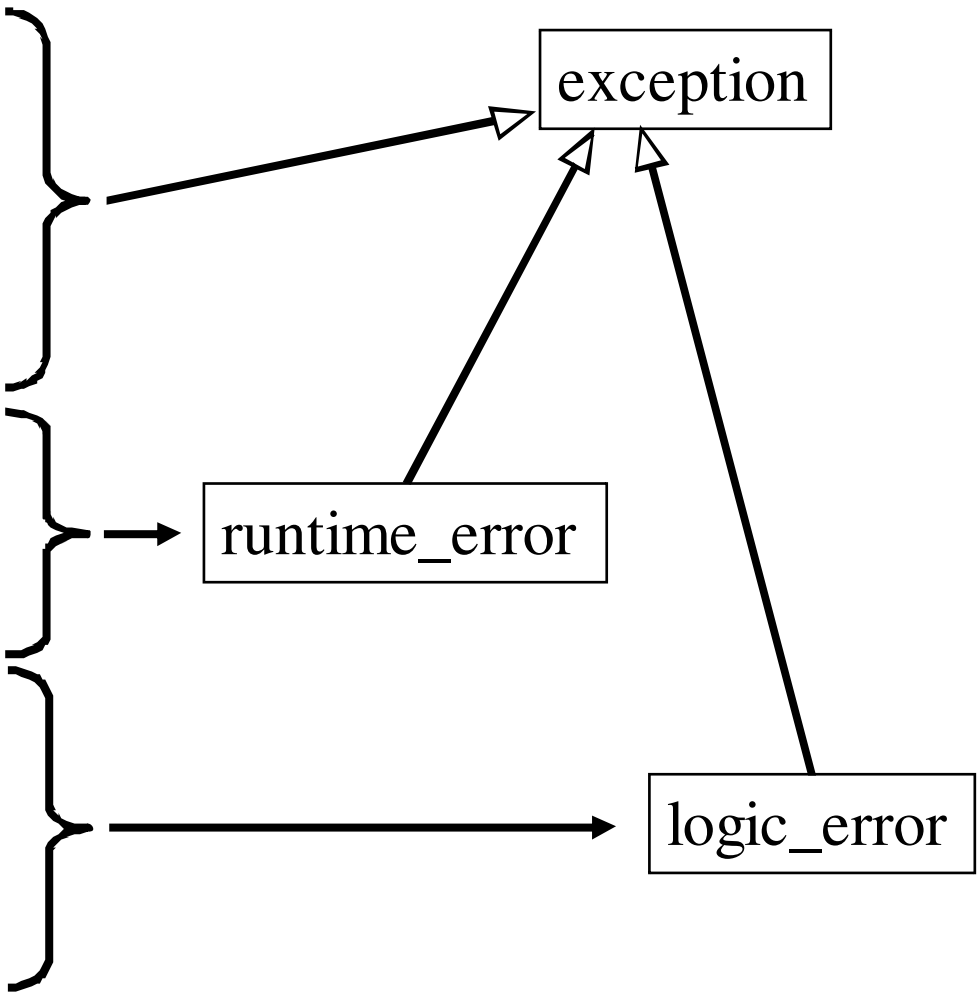
1. Normál meghívás
2. Kivételkezelés (roll back) miatti meghívás. Ekkor a kivétel nem léphet ki a destruktorból.

Destruktorban keletkező kivétel elfogása:

```
A::~A() try {
                // destruktor törzse
                } catch (...) { // kivételkezelés
}
```

Szabványos kivételek (stdexcept)

bad_alloc
bad_cast
bad_typeid
bad_exception
ios_base::failure
range_error
overflow_error
underflow_error
lenght_error
domain_error
out_of_range
invalid_argument

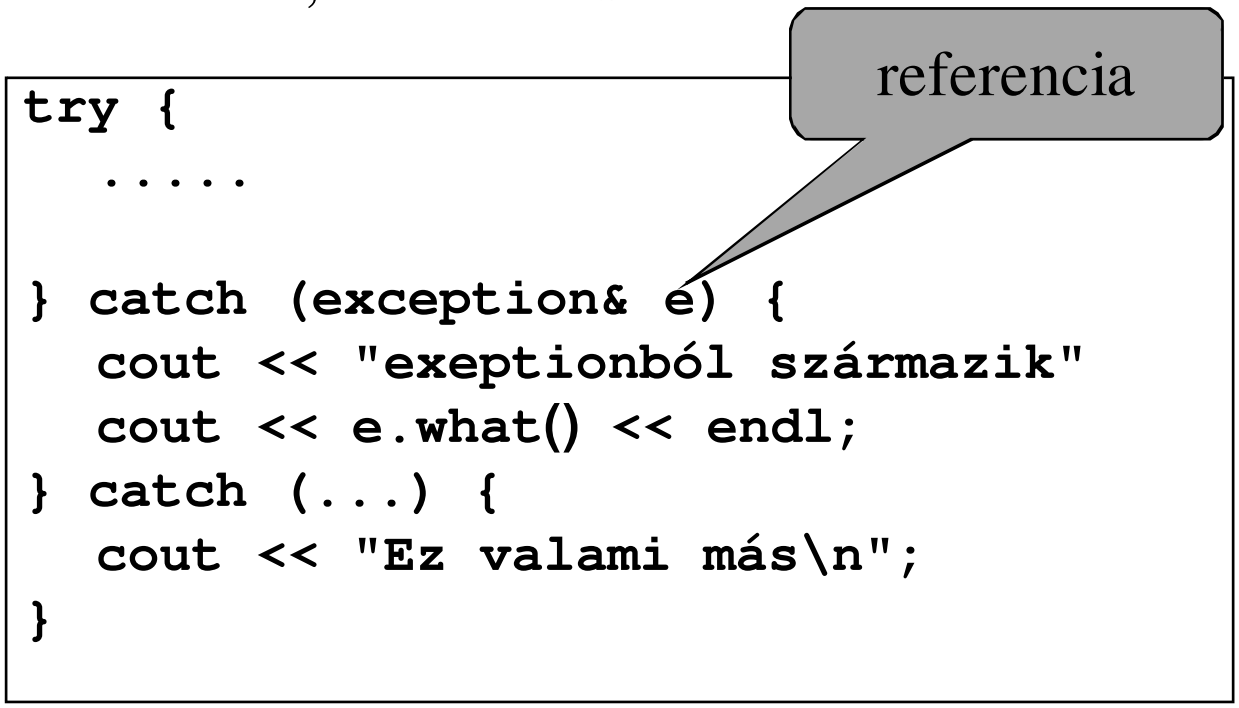


Szabványos kivételek/2

```
class exception {
...
public:
   exception() throw();
   exception(const exception&) throw();
   execption& operator=(const exception&) throw();
   virtual ~exception() throw();
   virtual const char *what() const throw();
};
class runtime_error : public exception {
public:
   explicit runtime_error(const string& what_arg);
};
class logic_error : public exception {
public:
   explicit logic_error(const string& what_arg );
};
```

Szabványos kivételek/3

- A standard könyvtár nem bővíti az *exception* osztály függvényeit, csak megfelelően átdefiniálja azokat.
- A felhasználói programnak nem kötelessége az *exception-ből* származtatni, de célszerű.



Kapcsolódó függvények

- terminate_handler set_unexpected(voif f()) throw();
  - bellálítja az unexpected handlert.
- void unexpected();
  - meghívja az unexpected handlert. Ha ez nem várt kivételt dob, és a bad_exception lehetséges, akkor bad_exeption kivétel keletkezik.

    void f4() throw (E1, bad_exception);
- terminate_handler set_terminate(void f()) throw();
  - beállítja a terminate handlert
- void terminate();
  - meghívja a terminate handlert
- bool uncaugth_exception() throw();
  - kivételkezelés folyamatban van, még nem talált rá a megfelelő handlerre (nem kapták el). Destruktorban lenne szerepe, de …

unexpected kezelés példa

```
void myhandler() {
  throw;      // továbbdob -> bad_exception
}
void valami() throw (int, bad_exception) {
  throw 'c'; // karaktert dob, -> unexpected
}

int main (void) {
  set_unexpected(myhandler);// saját handlert állít
  try {
    valami();
  } catch (exception &e) {
    cerr << e.what() << endl;
  }
  return 0;
}
```

Szabványos könyvtár (STL)

Általános célú, újrafelhasználható elemek:

- tárolók, majdnem tárolók
- algoritmusok
- függvények
- bejárók
- kivételek
- memóriakezelők
- adatfolyamok

http://www.sgi.com/tech/stl/
http://www.cppreference.com/cppstl.html
http://www.cplusplus.com/reference/stl/

std::auto_ptr (memory)

- Objektum sablon a <memory> -ban
- Egy mutatót tárol. Megsemmisülésével a mutatott objektum is törlődik.
- Műveletei:
  - konstruktor, destruktor
  - get(), reset(), release()
  - * ->, =
  - konvertáló operatorok

std::auto_ptr /2

```
{   auto_ptr<C> cp1, cp2(new C);
    cp2->valami();
    cp1 = cp2;
    (*cp1).valami();
    auto_ptr<int> ip(new int);
    *ip.get() = 10;
    ip.reset(new int);
    int *ip2 =  ip.release();
    delete ip2;
}
// cp1 és ip által mutatott obj. is megsemmisül
```

Konstruktor inicializál

cp2 NULL lesz

Új poi, előző felszabadul

"Kézzel" szabadítunk fel

std::auto_ptr /3 !!!

```
{
    Vigyázat ! Tömbökre nem jó, mert delete-t hív!

     auto_ptr<int> ip(new int[100]);
}
// delete hívódik, delete[] helyett!!!
```

Szabványos tárolók

- vector
- list
- deque

} Sorozattárolók

- stack
- queue
- priority_queue

} Adapterek

- map
- multimap
- set
- multiset

} Asszociatív tárolók

- string
- array
- valarray
- bitset

} Majdnem tárolók

vector template

```
template <class T, class Allocator = allocator<T> >
class vector {
public:
// Types
  typedef T value_type;
  typedef Allocator allocator_type;
  class iterator;
  class const_iterator;
  typedef typename Allocator::size_type  size_type;
  typedef typename Allocator::difference_type  difference_type;
  typedef typename Allocator::reference  reference;
  typedef typename Allocator::pointer  pointer;
  typedef typename std::reverse_iterator<iterator> reverse_iterator;
....
```

vector template/2

```
// Construct/Copy/Destroy
  explicit vector(const Allocator& = Allocator());
  explicit vector(size_type, const Allocator& = Allocator ());
  vector(size_type, const T&, const Allocator& = Allocator());
  vector(const vector<T, Allocator>&);
  vector<T,Allocator>& operator=(const vector<T, Allocator>&);
  template <class InputIterator>
          void assign(InputIterator start, InputIterator finish);
  void assign(size_type, const);
  allocator_type get_allocator () const;
```

vector template/3

```
// Iterators
  iterator begin();
  const_iterator begin() const;
  iterator end();
  const_iterator end() const;
  reverse_iterator rbegin();
  const_reverse_iterator rbegin() const;
  reverse_iterator rend();
  const_reverse_iterator rend() const;
```

vector template/4

```
// Capacity
  size_type size() const;
  size_type max_size() const;    // elméleti maximum
  void resize(size_type);
  void resize(size_type, T);
  size_type capacity() const;    // jelenleg allokált
  bool empty() const;
  void reserve(size_type);
```

vector template/5

```
// Element Access
  reference operator[](size_type);
  const_reference operator[](size_type)  const;
  reference at(size_type);
  const_reference at(size_type)  const;
  reference front();
  const_reference front()  const;
  reference back();
  const_reference back() const;
```

vector template/6

```
// Modifiers
  void push_back(const T&);
  void pop_back();
  iterator insert(iterator, const T&);
  void insert(iterator, size_type, const T&);
  template <class InputIterator>
     void insert(iterator, InputIterator, InputIterator);
  iterator erase(iterator); iterator erase(iterator, iterator);
  void swap(vector<T, Allocator>&);
  void clear()
};
```

Tárolók által definiált típusok

Általános, minden tárolóra érvényes

- value_type, allocator_type,
- reference, const_reference,
- pointer, const_pointer
- iterator, const_iterator
- reverse_iterator, const_reverse_iterator
- difference_type
- size_type

Minden asszociatív tárolóra érvényes

- key_type
- key_compare

Tárolók műveletei

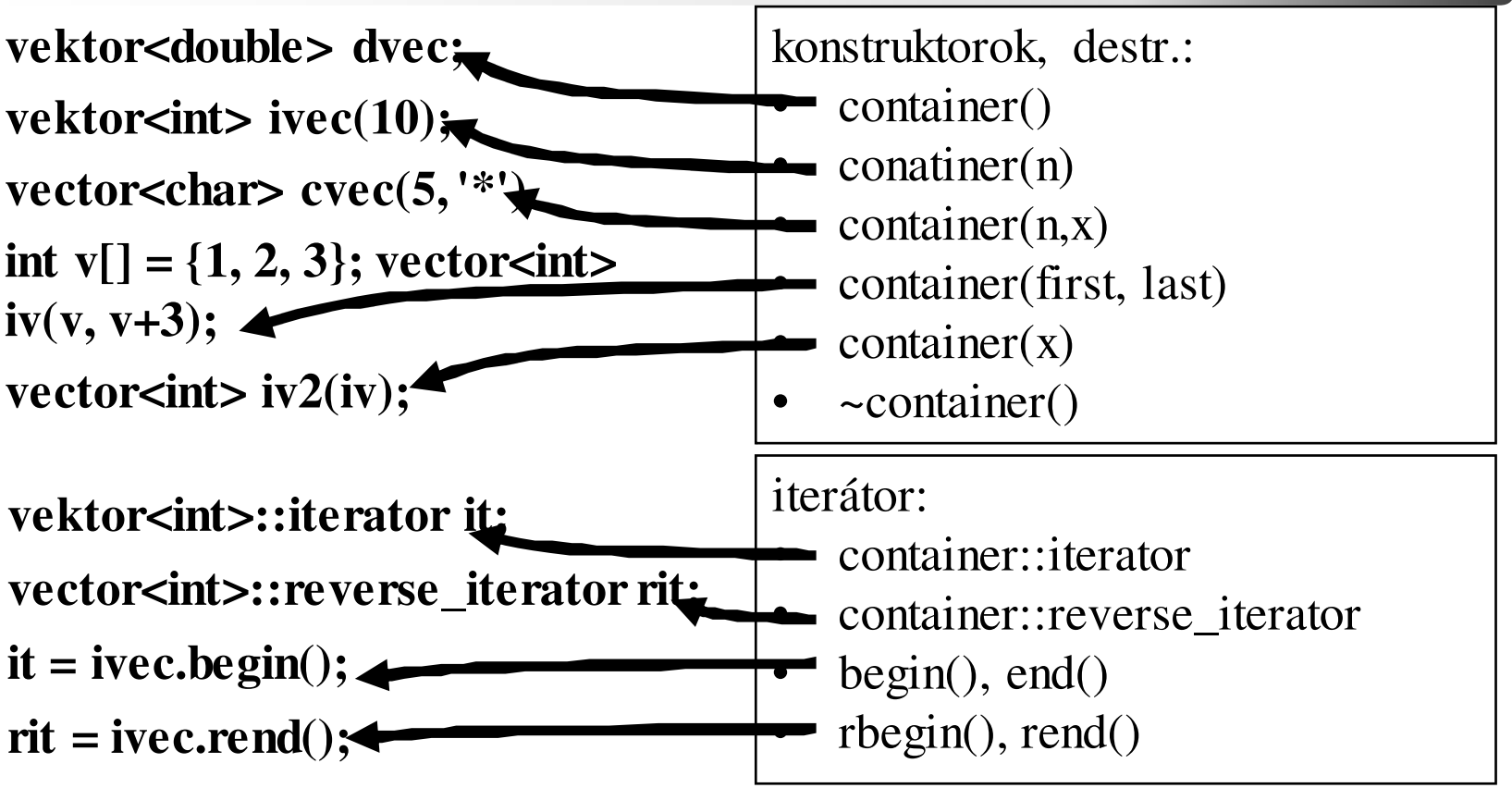
Általános, minden tárolóra érvényes
- létrehozás/megszüntetés: konstruktorok/destruktor
- értékadás: operator=, assign()
- iterátor példányosítás: begin(), end(), rbegin(), reend()
- méret lekérdezés: size(), max_size(), empty()
- módosítók: insert(), erase(), clear(), swap()

Lista kivételével minden sorozattárolóra érvényes
- elemek elérése: front(), back(), operator[], at()
- módosítók: push_back(), pop_back()

Minden asszociatív tároló érvényes
- count(), find(), lower/upper_boud(), equal_range()

Speciális, csak egy adott tárolóra alkalmazható pl:
- elemek elérése: pop_front(), push_front(),
- módosítók: merge(), splice(), uniq(), remove(), sort(), …

Létrehozás, iterátorok

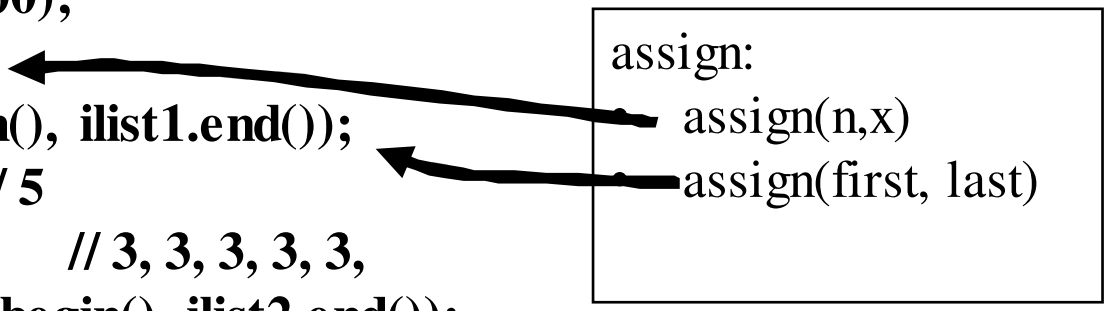



```
for (it = ivec.begin(); it != ivec.end(); ++it)
        std::cout << *it;
```

Értékadás, elérés, size

```
template <typename C> // Segédsablon a példákhoz
void print(C& co, const char* sep = ",") {
   for (typename C::iterator it = co.begin(); it != co.end(); ++it )
      std::cout << *it << sep;
}
```

```
list<int> ilist1, ilist2(100);
ilist1.assign(5, 3);
ilist2.assign(ilist1.begin(), ilist1.end());
cout << ilist2.size();  // 5
print(ilist2, ", " );          // 3, 3, 3, 3, 3,
vector<int> ivec(ilist2.begin(), ilist2.end());
ivec[0] = 6;
cout << ivec.front();  // 6
cout << ivec.back();  // 3
```



Módosítók

```
template <typename C> // Segédsablon a példákhoz
void reverse_print(C& co, const char* sep = ",") {
 for (typename C::reverse_iterator it = co.rbegin();
                                          it != co.rend(); ++it )
      std::cout << *it << sep;
}
```

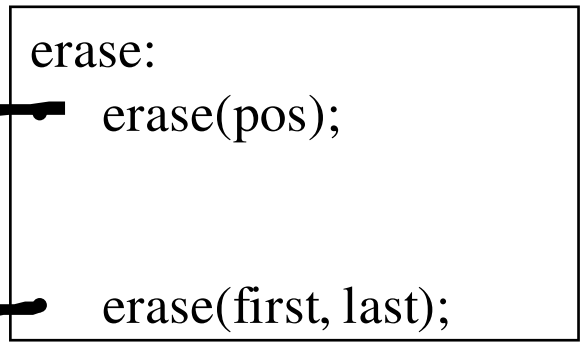
```
char duma[] = "HELLO";
vector<char> cv(duma, duma+5);
reverse_print(cv, """);   // OLLEH
cv.insert(cv.end(), ' ');
cv.insert(cv.end(), duma, duma+5);
print(cv, """);  //HELLO HELLO
cv.insert(cv.begin(), 3, '*') ;
print(cv, """);  //***HELLO HELLO
```

insert:

- insert(pos, x);
- insert(pos, first, last);
- insert(pos, n, x)

További módosítók

```
deque<int> iq1, iq2;
iq1.push_back(3);  iq1.push_back(4);
iq1.push_back(8);iq1.push_back(10);
iq1.push_back(2);iq1.push_back(7);
cout << iq1.front();              // 3
print(iq1);                       // 3,4,8,10,2,7,
iq1.pop_back();
print(iq1);                       // 3,4,8,10,2,
iq1.erase(iq1.begin()+1);
print(iq1);                       // 3,8,10,2,
iq1.erase(iq1.begin()+1, iq1.end());
print(iq1);                       // 3,
```



Asszociatív tárolók műveletei

```
set<int> set1;
set1.insert(4); set1.insert(3);
cout << set1.count(3);
set1.insert(3);  set1.insert(10);
print(set1);                          // 3,4,10
if (set1.find(4) != set1.end())
   cout << "megvan";                  // megvan
typedef set<int>::iterator myit;
myit it = set1.lower_bound(4);
cout << *it;                          // 4
it = set1.upper_bound(4);
cout << *it;                          // 10
std::pair<myit, myit> rit;
rit = set1.equal_range(4);
cout << *rit.first << ":" << *rit.second;   // 4:10
```

- count(x)
- find(x)
- lower_bound(x)
- upper_bound(x)
- equal_range(x)

vector<T, Alloc>

Speciális műveletek:
- capacity()
- reserve()
- resize(n), resize(n, val)

Nincs:
- push_front, pop_front
- asszociatív op.

```
int v[] = {0, 1, 2};
vector<int> iv(v, v+3);
iv.push_back(3);
iv.at(5) = 5; // hiba
```

```
iv.resize(7, -8);
iv.at(5) = 5; // nincs hiba
print(iv) // 0,1,2,3,-8,5,-8,
```

list<T, Alloc>

Speciális műveletek:

- merge(list),
- merge(list, bin_pred)
- remove(val)
- remove_if(un_pred)
- resize(n), resize(n, val)
- sort(), sort(bin_pred)
- splice(p, list)
- splice(p, list, first)
- splice(p, list, first, last)
- unique(),unique(bpred)

Nincs:

- at(), operator[]()
- asszociatív op.

```
list<int> il(2, -3);
il.push_front(9);
il.push_back(2);
il.sort();
print(il); // -3, -3, 2, 9,
il.unique();
list<int> il2(3, 4);
il.merge(il2);
print(il); // -3, 2, 4, 4, 4, 9,
```

deque<T, Alloc>

Kétvégű sor
Speciális műveletek:
- resize(n), resize(n, val)

Nincs:
- asszociatív op.

```
deque<int> dq;
dq.push_back(6);
dq.push_front(9);
print(dq); // 9, 6,
dq.resize(6, -3);
print(dq); //9, 6, -3, -3, -3,-3,
```

```
dq.back() = 1;
print(dq); // 9, 6, -3, -3, -3, 1,
dq[2] = 2;
print(dq); // 9, 6, 2, -3, -3, 1,
dq.at(3) = 0;
```

```
if (!dq.empty())
        print(dq); // 9, 6, 2, 0, -3, 1,
```

stack<T, deque>

Elrejti a kétvégű sor nem verem stílusú műveleteit.

Műveletek:

- empty()
- push()
- pop()
- top()
- stack()
- stack(cont)

```
stack<int> s;
s.push(1);
s.push(2);
s.push(3);
s.top() = 4;
s.push(13);
```

```
while (!s.empty())  {
   cout << s.top() << ", "; s.pop();
} // 13, 4, 2, 1,
```

queue<T, deque>

Elrejti a kétvégű sor nem sor stílusú műveleteit.

Műveletek:

- empty()
- push() –> push_back()
- pop() –> pop_front()
- front()
- back()
- queue(), queue(cont)

```
queue<int> q;
q.push(1);
q.push(2);
q.push(3);
q.back() = 4;
q.push(13);
```

```
while (!q.empty())  {
   cout << q.front() << ", "; q.pop();
} // 1, 2, 4, 13,
```

priority_queue<T, vector, Cmp>

Prioritásos sor. Alapesetben a < operátorral hasonlít.

Műveletek:

- empty()
- push()
- pop()
- top()
- priority_queue()

```
priority_queue<int> pq;
pq.push(1);
pq.push(2);
pq.push(3);
pq.push(-2);
pq.push(13);
```

```
while (!pq.empty())  {
   cout << pq.top() << ", "; pq.pop();
} // 13, 3, 2, 1, -2,
```

map<Key, T, Cmp, Alloc>

Asszociatív tömb
- (kulcs, érték) pár tárolása
- alapértelmezés szerint < operátorral hasonlít
- map maga is összehasonlítható

```
map<string, int> m;
m["haho"] = 8;
m["Almas"] = 23;
cout << m["haho"] << endl;
cout << m["Almas"] << endl;
map<string, int>::iterator i = m.find("haho");
```

pair<const Key, mapped_type>

Párok
- map bejárásakor párok sorozatát kapjuk
- A kulcsra first, az értékre second mezővel hivatkozhatunk

```
map<string, int> m;
m["haho"] = 8; m["Almas"] = 23; m["xx"] = 13;
map<string, int>::iterator p;
for (p = m.begin(); p != m.end(); p++) {
  cout << p->first << ": ";
  cout << p->second << ", ";
} // almas: 23, haho: 8, xx: 13
```

set<Key, Cmp, Alloc>

Halmaz
- olyan map, ahol nem tároljuk az értéket
- alapértelmezés szerint < operátorral hasonlít
- map-hoz hasonlóan összehasonlítható

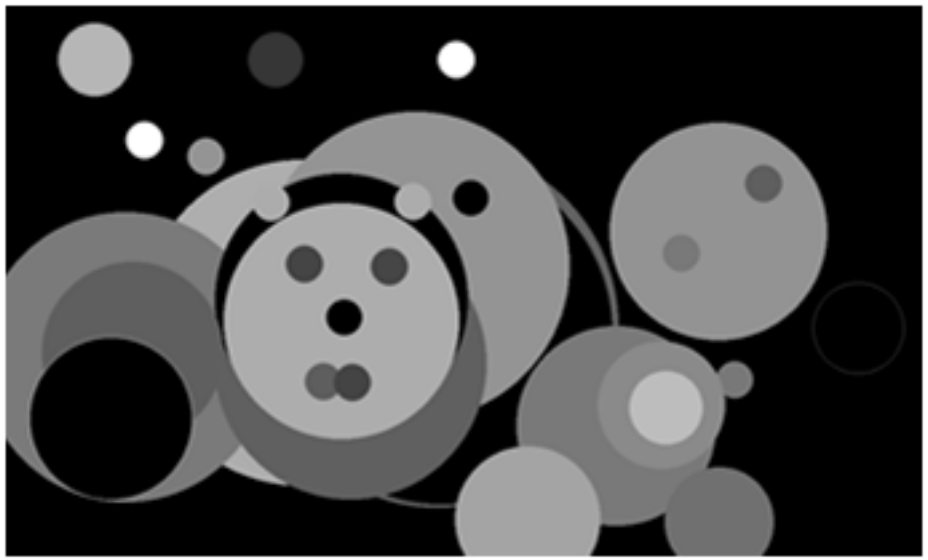
```
set<long> s;
s.insert(3); s.insert(3);
s.insert(7); s.insert(12); s.insert(8);
cout << s.count(6) << endl;  // 0
cout << s.count(3) << endl;  // 1
set<long>::iterator i = s.find(3);
print(s); // 3, 7, 8, 12,
```

Demo:Eseményvezérelt program

Demonstrációs cél: eseményvezérlés grafikus felhasználói felület működése, egyszerű SDL grafika, újrafelhasználható elemek, callback technika

Specifikáció: Eseményekre reagáló alakzatok (körök) felrakása a képernyőre. Vezérlő gombok (töröl, kilép) kialakítása.

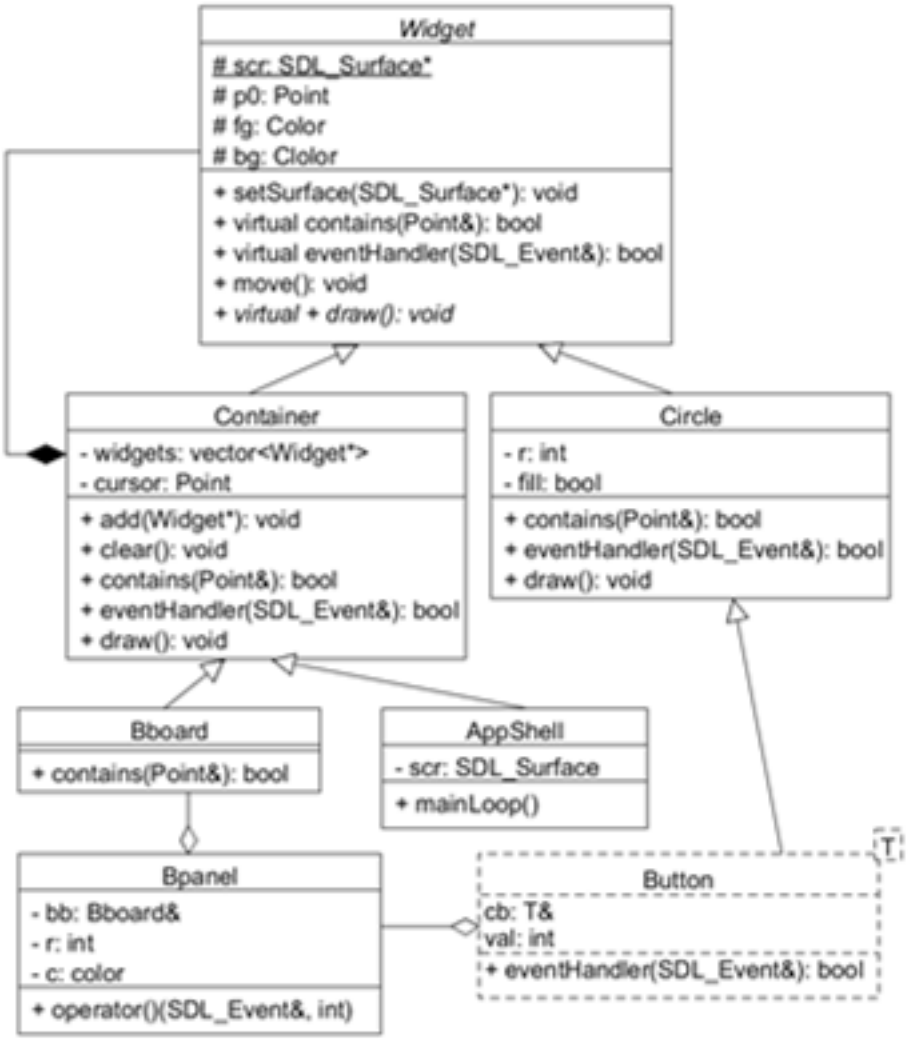
Események: Egérmozgatás, kattintás, húzás (drag)



Szereplők

- grafikus primitívek:
  - Widget, Circle, Button, ActiveBg
- összetettet grafikus elemek:
  - Container, BulletinBoard (Bboard),
  - Application Shell (AppShell)

Osztálydiagram



- A modell nem kezel ablakokat, így vágás sincs az ablakok határán.
- Minden grafikus elem a képernyő tetszőleges pontján megjelenhet.
- Egy esemény érkezésekor (pl. kattintás) a konténerbe helyezéssel ellentétes sorrendben megkérdezzük az objektumokat, hogy az adott koordináta hozzá tartozik-e (contains fv.).
- Amennyiben igen, akkor meghívjuk a eseménykezelőjét.
- Az eseménykezelő visszatérési értékel jelzi, hogy kezelt-e.
- A Bboard objektum hazudós, mert minden pozícióra „rábólint”, hogy az övé. Így végigfut a tárolóban az ellenőrzés.

Container

```
class Container : public Widget {
protected:
    std::vector<Widget*> widgets;   //< Itt tárolunk
    Point cursor;                   //< egérmozgáshoz
public:
    void add(Widget *w);
    void clear();
    bool contains(const Point& p) {
       cursor = p;          // egérmozgás követése
       return false;
    }
    bool eventHandler(const SDL_Event&);
    void draw() const;
};
```

Container bejárása

```
void Container::draw() const {
  for (size_t i = 0; i < widgets.size(); i++)
    widgets[i]->draw();
}

// Az objektumok sorrendje meghatározza a takarásokat.
bool Container::eventHandler(const SDL_Event& ev) {
// megváltozhat a container tartalma a ciklus alatt
  for (size_t i = widgets.size()-1; i >= 0
                          && i < widgets.size(); i--) {
    if (widgets[i]->contains(cursor)) {
      if (widgets[i]->eventHandler(ev))
        return true;
    }
  }
  return false;
}
```

Button template

```
template<class T = void (&)(const SDL_Event&, int) >
class Button : public Circle {
  T& cb;     //< visszahívandó objektum referenciája
  int val;   //< int érték
public:
// Konstruktor elteszi a rutin címét és a paramétert
Button(const Point& p0, int r, T& cb, int val = 0,
  const Color& fg = WHITE, const Color& bg = BLACK) :
  Circle(p0, r, fg, bg, true), cb(cb), val(val) { }
// A felhasználói felületen megnyomták a gombot
  bool eventHandler(const SDL_Event& ev) {
    if (ev.type == SDL_MOUSEBUTTONDOWN) {
      if (ev.button.button == SDL_BUTTON_LEFT) cb(ev, val);
      return true;
    }
    return Circle::eventHandler(ev);
  }
};
```