Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

7. előadás

Strukturált programozás: adatszerkezetek megvalósítása

Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Adatszerkezetek megvalósítása

Gyűjtemények

- *Adatszerkezet*nek nevezzük adatok tárolási célokat szolgáló strukturális, formai elrendezését
- A legáltalánosabb adatszerkezetek nevezzük *gyűjtemények*nek (*collection*), amelyek sok nyelven eleve adottak, pl.:
  - *tömb* (vektor): rögzített hosszú, egészekkel indexelt sorozat
  - *verem* (LIFO): bővíthető sorozat, ahol mindig csak az utolsó elem érhető el
  - *sor* (FIFO): bővíthető sorozat, ahol mindig csak az első elem érhető elem
  - *asszociatív tömb*: speciális értékekkel indexelt sorozat, amely bárhol bővíthető

Adatszerkezetek megvalósítása

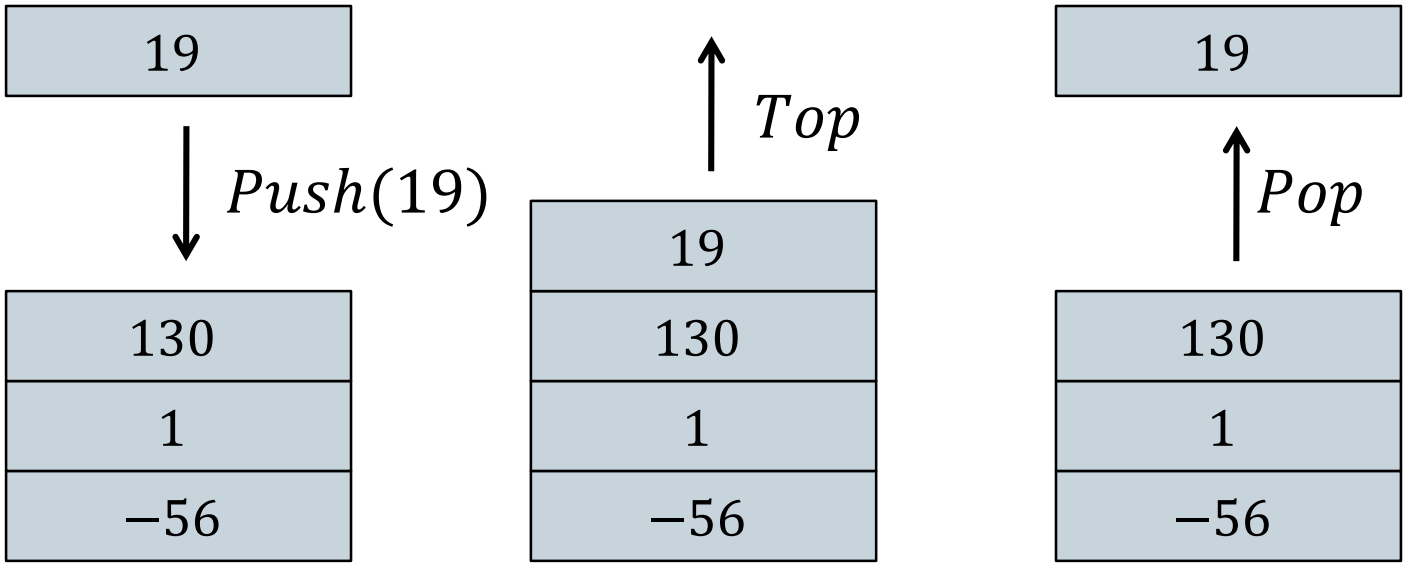
Adatszerkezetek megvalósítása

- Adatszerkezetek megvalósításának szintjei:
  - *absztrakt adattípus* (ADT): definiálja a típus értékhalmazát és műveletalmazát, valamint a viselkedés paramétereit (pl. specifikáció segítségével)
  - *absztrakt adatszerkezet* (ADS): megadja az alapvető struktúrát, azaz a rákövetkezési kapcsolatot (pl. irányított gráffal)
  - *reprezentáció*: megadja az ábrázolási modellt, azaz milyen primitív elemekből épül fel az adatszerkezet, és azokon miként hajtódnak végre a műveletek, két fő típusa:
    - *aritmetikai*: tömböt veszünk alapul
    - *láncolt*: dinamikus memóriakezelést használ

Adatszerkezetek megvalósítása

A verem

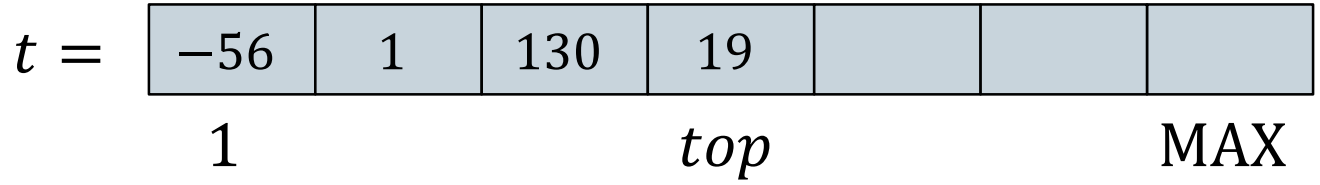
- A *verem* (*stack*), más néven LIFO (Last-In-First-Out) egy olyan bővíthető adatszerkezet, amelyben
  - csak a tetejére (végére) tudunk helyezni új elemet (*Push*)
  - csak a tetőelemet kérdezhetjük el (*Top*), és vehetjük ki (*Pop*)



Adatszerkezetek megvalósítása

A verem

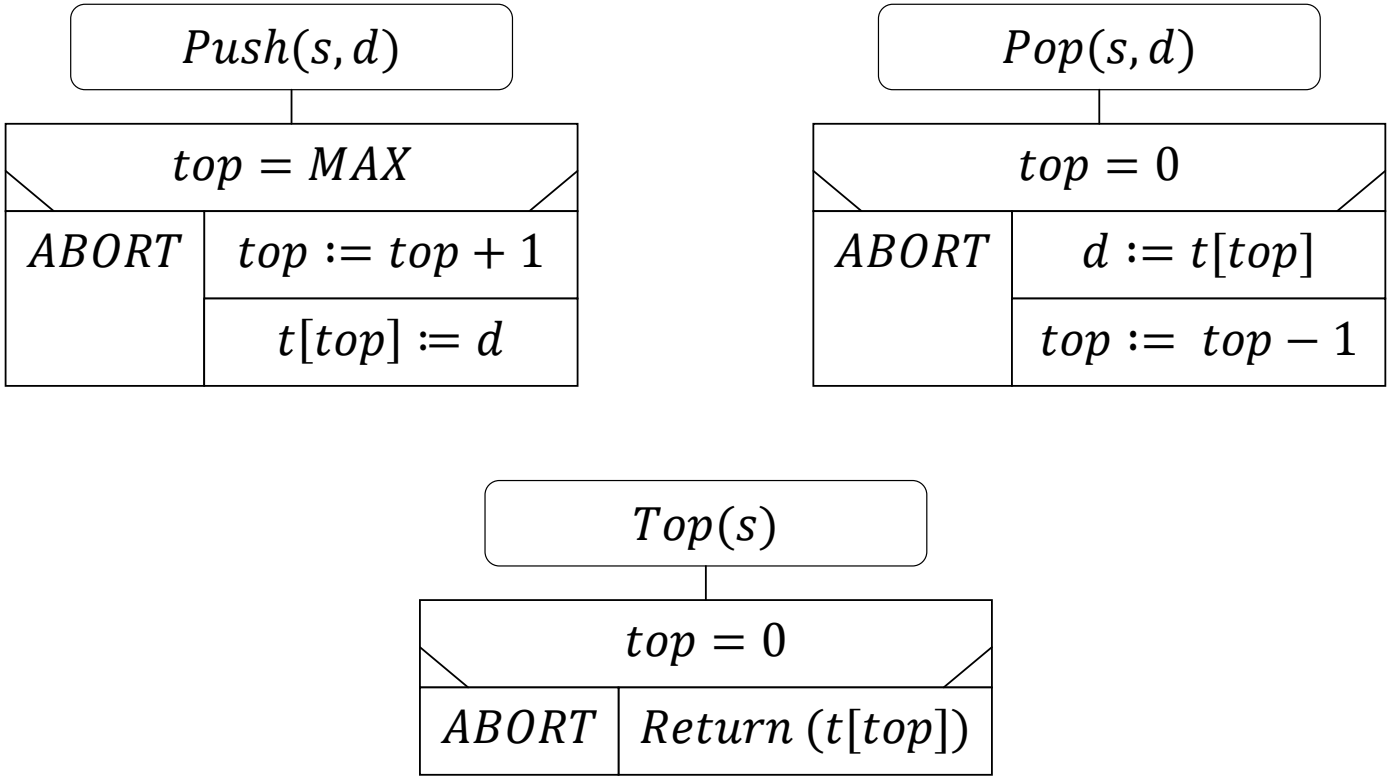
- A verem absztrakt adattípusa
  $S = (D^n, \{Push, Pop, Top, IsEmpty\})$, ahol
  - $Push: S \times D \rightarrow S$
  - $Pop: S \rightarrow S \times D$
  - $Top: S \rightarrow D$
  - $IsEmpty: S \rightarrow \mathbb{L}$
- A verem aritmetikai reprezentációja esetén az elemeket egy tömbbe helyezzük el, amely egy adott indexig (top) tartalmazza a vermet:



Adatszerkezetek megvalósítása

A verem

- A verem műveletei aritmetikai reprezentációval:



Adatszerkezetek megvalósítása

Példa

Feladat: Valósítsuk meg az egész számokat tartalmazó verem (**`Stack`**) típusát aritmetikai reprezentáció mellett.

- lehessen elemet behelyezni (**`push`**), kivenni (**`pop`**), kitörölni a teljes vermet (**`clear`**), lekérdezni a tetőelemet (**`top`**), üres-e (**`isEmpty`**), illetve tele van-e a verem (**`isFull`**), valamint mi az aktuális mérete (**`size`**)
- használjuk fel a típust egy olyan programban, amelyben megfordítjuk a bemenetről kapott 20 szám sorrendjét
- valósítsuk meg a típust külön fordítási egységben, a megvalósítása legyen rejtett a külvilág elől
- a reprezentációhoz használjunk egy 100 elemű tömböt (**`_values`**), valamint egy **`_top`** indexet

Adatszerkezetek megvalósítása

Példa

Megoldás (`stack.hpp`*):*

```
class Stack { // egész számokat tároló verem típus
private:
   int _values[100]; // értékek tömbje
   int _top; // a tetőelem indexe
public:
   Stack(); // konstruktor
   bool isEmpty() const; // üres-e a verem
   bool isFull() const; // tele van-e a verem
   bool push(int value); // elem behelyezése
   int pop(); // elem kivétele
   int top() const; // tetőelem lekérdezése
   void clear(); // elemek törlése
   int size() const; // méret lekérdezése
};
```

Adatszerkezetek megvalósítása

Példa

Megoldás (`stack.cpp`):

```
#include "stack.hpp"

Stack::Stack(){
    _top = 0; // kezdetben 0 elem van a veremben
}

bool Stack::isEmpty() const {
    return _top == 0;
}
bool Stack::isFull() const {
    return _top == 100;
}
…
```

Adatszerkezetek megvalósítása

Sablonok

- Sokszor előfordul, hogy egy adott osztályt, különösen adatszerkezetet több elemtípussal is meg akarunk valósítani (pl. számmal, szöveggel, vagy saját típussal)
  - ehhez több, művelethalmazában megegyező, de értékhalmazában különböző típus szükséges
- A megoldás az, hogy felveszünk egy behelyettesíthető típust, egy úgynevezett *sablon*t (*template*)
  - ezt a sablont tetszőleges helyen felhasználhatjuk a típusban
  - példányosításakor behelyettesítünk a használni kívánt konkrét típussal
  - használatát `template<class <név> >` jelöli a típusnál

Adatszerkezetek megvalósítása

Sablonok

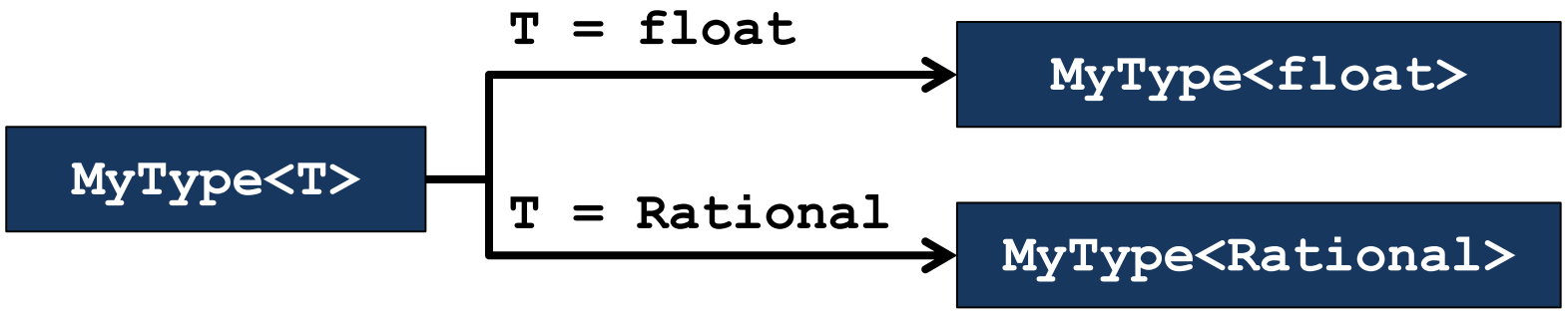
- Pl.:

```
template <class T> // T sablon használata
class MyType {
private:
   int _intValue; // rögzített típusú érték
   T _tValue;     // T típusú érték, ahol T sablon
public:
   void tMethod(T param){ // T paraméterű metódus
      _tValue = param;
   }
};
…
MyType<float> mtf;  // T helyettesítése float-tal
mtf.tMethod(3.5); // a paraméter float típusú lesz
```

Adatszerkezetek megvalósítása

Sablonok működése

- A sablonok biztosítják adatszerkezetek esetén a típusfüggetlenséget, hiszen így az elemtípus tetszőleges lehet, pl. `vector<float> fVector;`
- Sablonos típus létrehozásával igazából *típusmintá*t hozunk létre, amely a sablon behelyettesítésével válik igazán típussá
  - azaz egy típusmintából több típus is létrejöhet
  - a behelyettesítésből fakadó hibák fordításkor kiderülnek



Adatszerkezetek megvalósítása

Sablonok alkalmazása

- Mivel a sablonos típus nem igazi típus, csak minta, a kódja nem helyezhető el forrásfájlban, csak fejlécfájlban (ezért a sablonos típusok egy fájlból állnak)
- Minden típusbeli hivatkozásnál a sablont is meg kell adnunk (vagy általános, vagy konkrét típussal)
- Amennyiben leválasztjuk a metódusok definícióját, a definícióban ismét kell jelölnünk a sablont, pl.:

```
template <class T> // T sablon használata
class MyType { … };

template <class T> // jelöljük a sablont
void MyType<T>::tMethod(T param){ … }
```

Adatszerkezetek megvalósítása

Sablonok lehetőségei

- Egy típus több sablonnal is rendelkezhet, ekkor azokat vesszővel választjuk el, pl.:

```
template <class T1, class T2> class MyType { … };
```

- A sablonok számos további lehetőséget kínálnak
  - nem csak típusok, de alprogramok is megvalósíthatóak sablonosra
  - értékek is megadhatóak, amik lehetnek sablonosak is
  - a sablonos művelet specializálható konkrét típusokra
- Egy másik lehetséges sablonmegoldás a generikus típus (*generic*, pl. Java), ahol a sablon behelyettesítése futási időben történik

Adatszerkezetek megvalósítása

Példa

Feladat: Módosítsuk úgy a verem típust, hogy tetszőleges elemtípussal használható legyen, azaz alakítsuk át sablonossá.

Megoldás (`stack.hpp`):

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
   T _values[100];
   int _top;
   …
   bool push(T value); // elem behelyezése
   T pop(); // elem kivétele
   T top() const; // tetőelem lekérdezése
   …
```

Adatszerkezetek megvalósítása

Példa

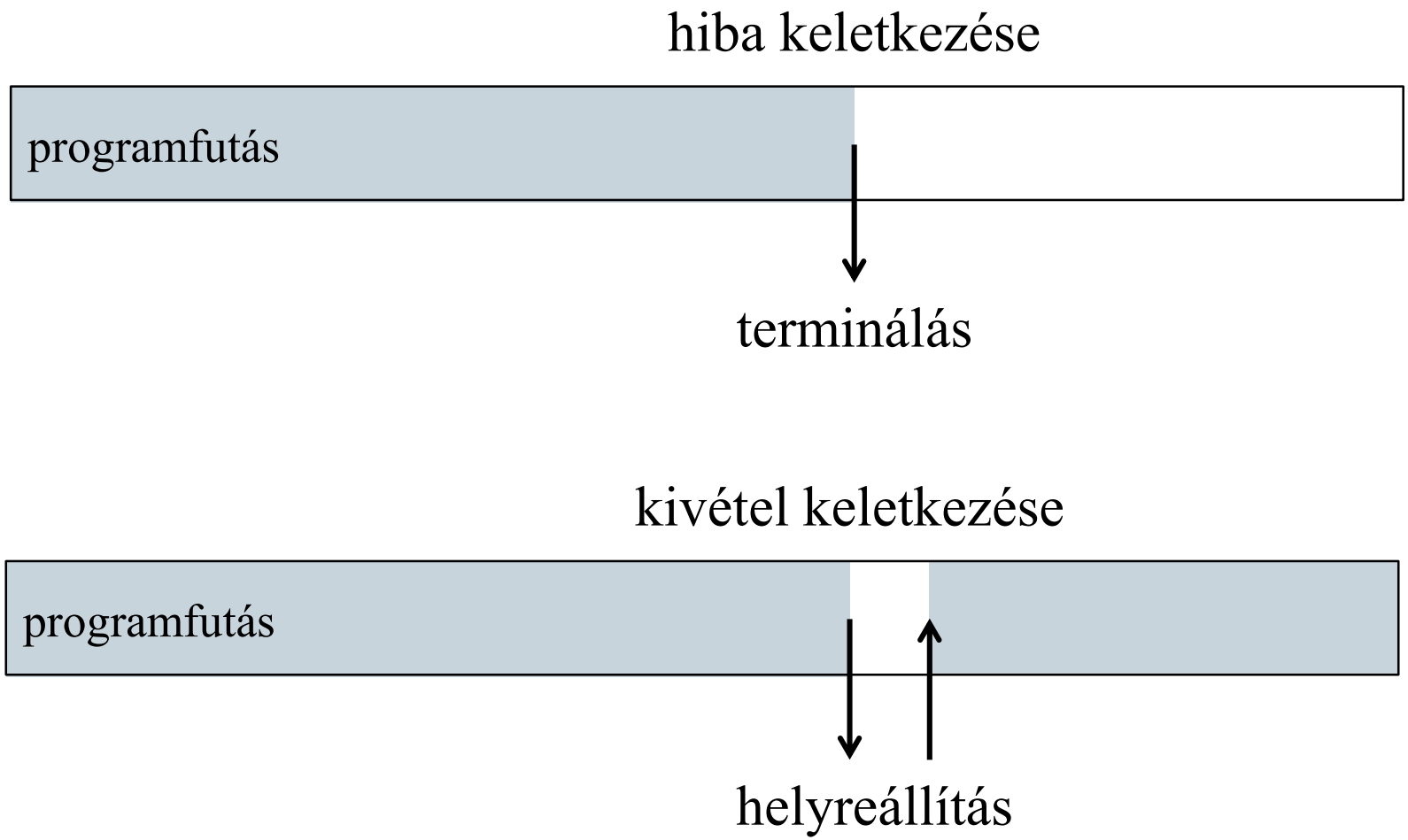
Megoldás (`stack.hpp`):

```
…
template <class T>
Stack<T>::Stack() {
    _top = 0; // kezdetben 0 elem van a veremben
}
…
template <class T>
bool Stack<T>::push(T value) {
    …
}
…
```

Adatszerkezetek megvalósítása

Hibák és kivételek

- A program futása során különböző, feladatnak megfelelő viselkedést megszakító jelenségek fordulhatnak elő, amelyek „abnormális” állapotba viszik a programfutást
- Ezeket a jelenségeket két kategóriába soroljuk:
  - *hibák* (*error*): olyan jelenségek, amelyeket a program nem tud futás közben kijavítani, és a program az abnormális állapotban terminál
  - *kivételek* (*exception*): olyan jelenségek, amelyek a program futása közben kijavíthatóak, ezért a program futása folytatódhat normál állapotban
    - általában megelőzik a tényleges hiba bekövetkeztetét


hiba keletkezése
programfutás
terminálás
kivétel keletkezése
programfutás
helyreállítás

Adatszerkezetek megvalósítása

Kivételkezelés

- A kivételek futás közben történő felismerését, feldolgozását, majd a programfutás megfelelő állapotba történő visszaállítását nevezzük *kivételkezelés*nek (*exception handling*)
  - a kivételek *kiváltód*nak (*throw*), és azokat lehetőségünk van a program valamely szintjén *lekezelni* (*catch*)
  - a kivételkezelés rendszerint külön *kivételkezelő blokk*ot igényel a kódban
    - dedikáltan figyelni a kivételek előfordulását, és biztosítja a megfelelő reakció lefuttatását
  - a kivételkezelést a régebbi nyelvek (pl. C, Pascal) nem támogatják, itt egyéb eszközöket helyettesítik a hibajelzést
    - pl. logikai visszatérési érték, extremális érték, hibakód

Adatszerkezetek megvalósítása

Kivételkezelés

- A kivételek úgy keletkeznek a programban, hogy egy hibához vezető lépést megakadályozunk egy kivétel kiváltásával, pl.:
  - mielőtt túlindexelnénk a tömböt, jelezzük, hogy az index a megadott tartományon kívül van
  - mielőtt nullával osztanánk, jelezzük, hogy azt a számítógép nem tudja értelmezni
- Kivételt mi is kiválthatunk a programegységünkben, és azt feldolgozhatjuk, vagy továbbadhatjuk más programegységnek
- A le nem kezelt kivétel ugyanúgy megszakítja a program futását, mint a hiba, de általában nem hagyja inkonzisztens állapotban a mentett adatokat (pl. fájlban, vagy adatbázisban)

Adatszerkezetek megvalósítása

Kivételkezelés

- Kivételt kiváltani a **`throw`** utasítással tudunk:
  `throw <kivétel>;`
  - hatására a kivételelfogó utasításra kerül a vezérlés
  - bármilyen konstans vagy változó kiváltható kivételként
- A kivételkezelő blokk két részből áll, a felügyelt blokkból (**`try`**), valamint a kezelő blokkból (**`catch`**):

```
try { // felügyelt blokk
   <kivételkezelt utasítások>
}
catch (<elfogott kivétel>){ // kezelő blokk
   <kivételkezelő utasítások>
}
```

Adatszerkezetek megvalósítása

Kivételkezelés

```
int x, y; // nem kivételkezelt utasítások
…
try { // kivételkezelt utasítások
   cin >> x >> y;
   if (y == 0)
      throw 1; // az 1 kivétel dobása
   else
      cout << "Result: " << x / y << endl;
      // csak akkor fut le, ha y nem nulla
}
catch (int e) { // int típusú kivétel kezelése
   cout << "Exception occured: " << e << endl;
   // eredménye: "Exception occured: 1"
}
```

Adatszerkezetek megvalósítása

Kivételkezelés

```
string sx, sy; int x, y;
stringstream sstr;
try {
   cin >> sx >> sy; // biztonságos beolvasás
   sstr << sx << sy; sstr >> x >> y; // konverzió
   // különböző hibára különböző kivételek:
   if (sstr.fail())
      throw "Not number input!"; // kivétel dobás
   if (y == 0)
      throw "Division by zero!";
   cout << "Result: " << x / y << endl;
} catch (string e) {
   // string típusú kivétel kezelése
   cout << "Exception occured: " << e << endl;
}
```

Adatszerkezetek megvalósítása

Többszintű kivételkezelés

- A kivételt a kiváltás feletti bármely szinten kezelhetjük
  - pl. ha egy alprogram (metódus) meghív egy másikat, akkor az általa dobott kivételt kezelhetjük a hívó alprogramban
  - így kiválasztható az állapot helyreállításának helye
- Pl.:

```
class Rational {
public:
   Rational(int nom, int denom) {
      if (denom == 0) throw 1;
         // a művelet kivételt dobhat
      …
   }
   …
```

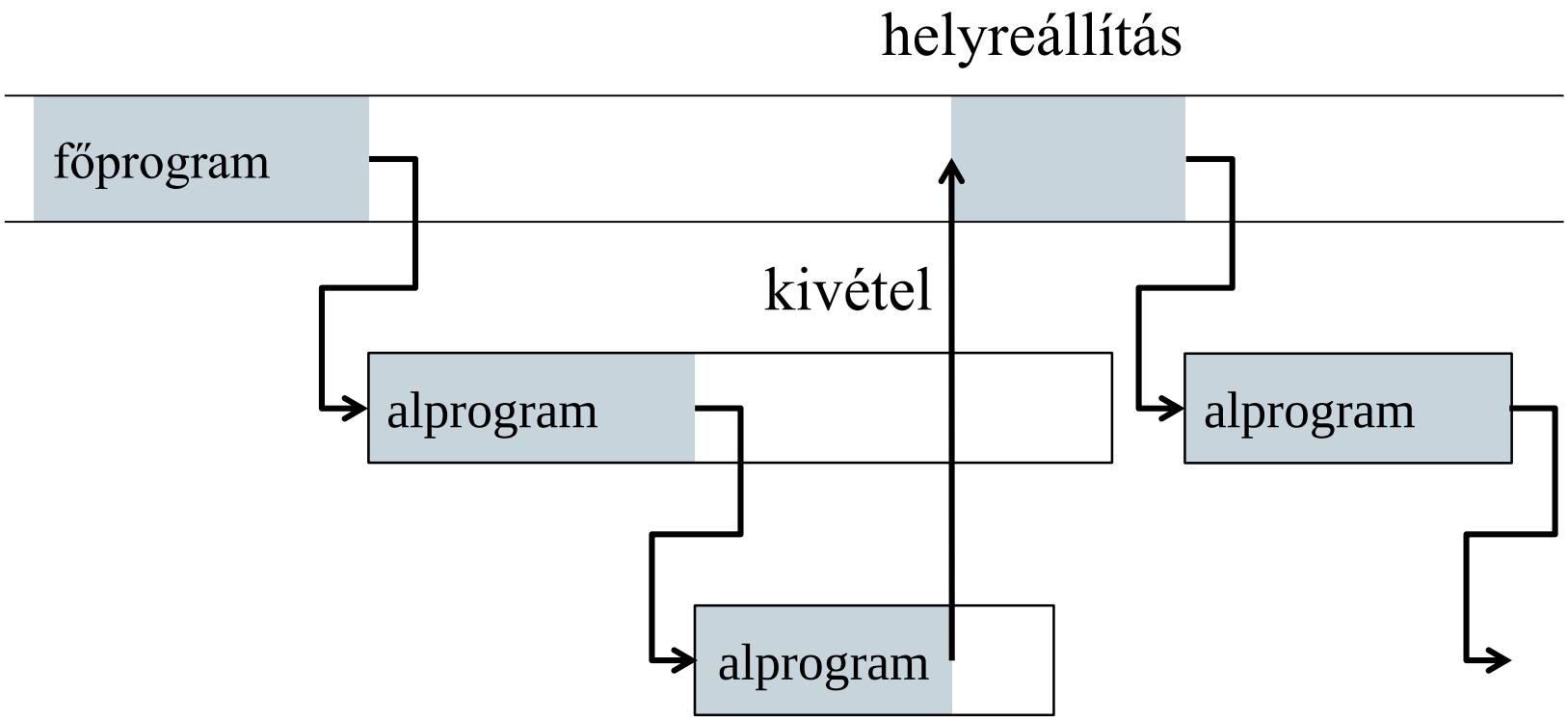
Adatszerkezetek megvalósítása

Többszintű kivételkezelés

```
bool continue = false;
do {
   try { // kivétel kezelő szakasz
      cin >> nom >> denom;
      Rational r(nom, denom);
      … // ha nem dob kivételt, tovább mehetünk
      continue = true;
        // jelezzük, hogy mehetünk tovább
   } catch(int ex) {
      if (ex == 1) // jelezzük a hibát
         cout << "The denominator is 0!";
   }
} while (!continue);
   // ismételjük, amíg hiba van
```

Adatszerkezetek megvalósítása

Többszintű kivételkezelés



Adatszerkezetek megvalósítása

Többszintű kivételkezelés

- A kivételek általában nem billentik ki a programot a normális állapotból, mert azelőtt váltódnak ki, hogy ez megtörténne, így lehetőségünk van
  - a kivétel jelzésére a felhasználónak (pl. üzenetként)
  - a kivételt okozó utasítások újrafuttatására más értékekkel (pl. bemeneti adatok újra bekérése), helyettesítésére más utasításokkal, vagy kihagyására
  - a program terminálására (ha már nem javítható az állapot), amely során
    - garantáljuk a kimentett adatok helyességét
    - biztosítjuk a még el nem mentett adatok mentését

Adatszerkezetek megvalósítása

Többszörös kivételkezelés

- Mivel több típusú kivételt is kiválthatunk, egy kivételkezelő blokk több **`catch`** blokkal is rendelkezhet, pl.:

```
try { <kivételkezelt utasítások> }
catch (<1. típusú kivétel>){ <kivételkezelés> }
catch (<2. típusú kivétel>){ <kivételkezelés> }
…
```

- Lehetőségünk van a kivételkezelő utasításban továbbdobni a megadott kivételt, vagy helyette másikat kiváltani:

```
try { <kivételkezelt utasítások> }
catch (<kivétel>){
    <kivételkezelés>;
    throw <kivétel>; // újabb kivétel kiváltás
}
```

Adatszerkezetek megvalósítása

Felsorolási típusok

- A *felsorolási típus*ok (*enumeration*) olyan speciális típusok, ahol az értékhalmaz egy rögzített, véges tartomány, amelynek értékeit felsoroljuk:

  `enum <név> { <érték 1>, <érték 2> };`

  - ez a típus csak értékhalmazból áll, és a belőle létrejött példányok csak a felsorolt értékeket vehetik fel
  - pl.:

    ```
    enum Nap { HETFO, KEDD, SZERDA, … };
    Nap n = KEDD;
    ```

  - általában segédtípusként hasznos (pl. többágú elágazásnál, kivételkezeléshez)
  - az értékei megfeleltethetőek egész számoknak

Adatszerkezetek megvalósítása

Típusok kivételei

- A legegyszerűbb megoldás kivételek definiálására saját típusokhoz felsorolási típus használata
  - a felsorolásban megadjuk a lehetséges kivételeket, pl.:

```
enum Exceptions { DENOM_ZERO, DENOM_NEGATIVE };
```

  - a kivételeket a típuson belül hozzuk létre, és váltjuk ki, pl.:

```
class Rational {
public:
    enum Exceptions { … }; // típus kivételei
    …
    Rational(int nom, int denom){
        if (denom == 0) throw DENOM_ZERO;
        if (denom < 0) throw DENOM_NEGATIVE;
        …
```

Adatszerkezetek megvalósítása

Típusok kivételei

- a kivételeket a külvilág kapja el és dolgozza fel, a típusból beazonosítva a felsorolási értékeket (a `::` operátorral), pl.:

```
try {
   Rational r(…);
}
catch (Rational::Exceptions ex){
   switch (ex) {
      // pontosan be tudjuk azonosítani a hibát
      case Rational::DENOM_ZERO:
         cout << "The denominator is 0!";
      case Rational::DENOM_NEGATIVE:
         cout << "The denominator is negative!";
   }
}
```

Adatszerkezetek megvalósítása

Példa

Feladat: Módosítsuk úgy a verem típust, hogy amennyiben nem megfelelő az állapot a művelet végrehajtására (pl. üres, vagy tele), akkor azt jelezzük kivétellel

- a kivételeket felsorolási típussal adjuk meg, két kivétel az üres, illetve tele verem (**`STACK_FULL`**, **`STACK_EMPTY`**)

Megoldás (`stack.hpp`):

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
    …
    enum Exceptions { STACK_FULL, STACK_EMPTY };
       // kivételek felsorolási típusa
    …
```

Adatszerkezetek megvalósítása

Példa

Megoldás (`stack.hpp`):

```
…
template <class T>
bool Stack<T>::push(T value) {
    if (_top < 100) { // ha még van hely
        …
    }
    else // különben kivételt dobunk
        throw STACK_FULL;

}
…
```

Adatszerkezetek megvalósítása

Példa

Megoldás (`main.hpp`):

```
…
try { // kivételkezelő blokk
   int v;
   do {
      cout << "Következő szám: "; cin >> v;
      myStack.push(v);
        // kiválthatja a STACK_FULL kivételt
   } while (v != 0);
} catch (Stack<int>::Exceptions ex) {
   // a veremből jövő kivételeket kapjuk el
   if (ex == Stack<int>::STACK_FULL) …
}
…
```

Adatszerkezetek megvalósítása

Kivétel-biztonság

- A programot *kivétel-biztos*nak (*exception-safe*) nevezünk, amennyiben garantáltan nem kerül abnormális állapotba
  - ehhez invariánsokat garantálunk a program futása során
  - ettől függetlenül a program előállíthat hibás adatokat, illetve azokat el is mentheti
- A kivétel-biztonságnak a következő szintjeit tartjuk nyilván:
  1. *kivétel-biztonság mentes*: a program nem garantálja az invariánsok teljesülését, kivétel hatására terminálhat
  2. *minimális kivétel-biztonság*: a program menthet és előállíthat hibás adatokat, de nem kerülhet abnormális állapotba (pl. főprogramban történő kivételkezeléssel)

3. *alap kivétel-biztonság*: a program nem menthet hibás adatokat, de a kivételt okozó műveletek részben lefuthatnak, és okozhatnak mellékhatásokat
4. *erős kivétel-biztonság* (változás-mentes garancia): a műveletek vezethetnek kivételhez, de azok megfelelő helyen kezelődnek, és az eredeti adatok helyreállnak
5. *kiváltás-mentes garancia*: minden kivétel a kiváltás szintjén kezelve van, tehát minden művelet hibamentessége garantált

- Sokszor a kiváltás-mentes garancia nem biztosítható, de az erős kivétel-biztonság igen, azaz mellékhatások és hibás adatok nem keletkezhetnek a futás során