Programozási nyelvek és módszerek

SABLONOK, KIVÉTELKEZELÉS

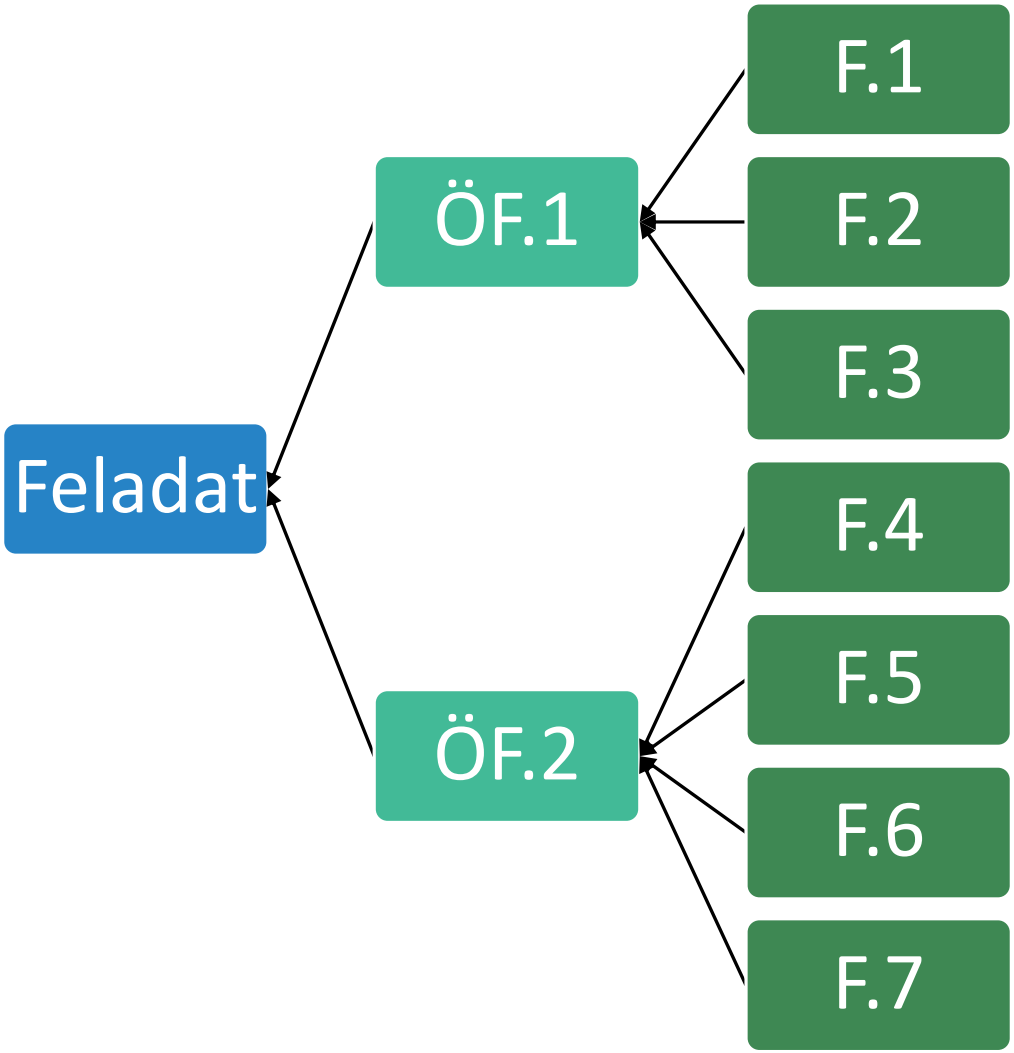
Adatabsztrakció

Adatabsztrakció

Ha elég magas szintről analizáljuk a problémát, akkor úgy tűnik, hogy

- a rendszerek jobban jellemezhetőek hosszú távon objektumaikkal, mint a rájuk alkalmazott tevékenységekkel.
- Ezt a megközelítést választva, először megpróbáljuk jellemezni az objektumokat, és az objektumok osztályait (~az adattípusokat) amelyek szerepet játszanak a rendszerben.
- Az új adattípusokat a már meglévők felhasználásával tervezzük – ez a bottom-up tervezés.
- Ez azt jelenti, hogy az elérhető komponensek szintjéről indulunk, abból építkezünk.
- Ezután oldjuk meg a problémát.

Alulról felfelé építkezés



Típus a programozó szemszögéből

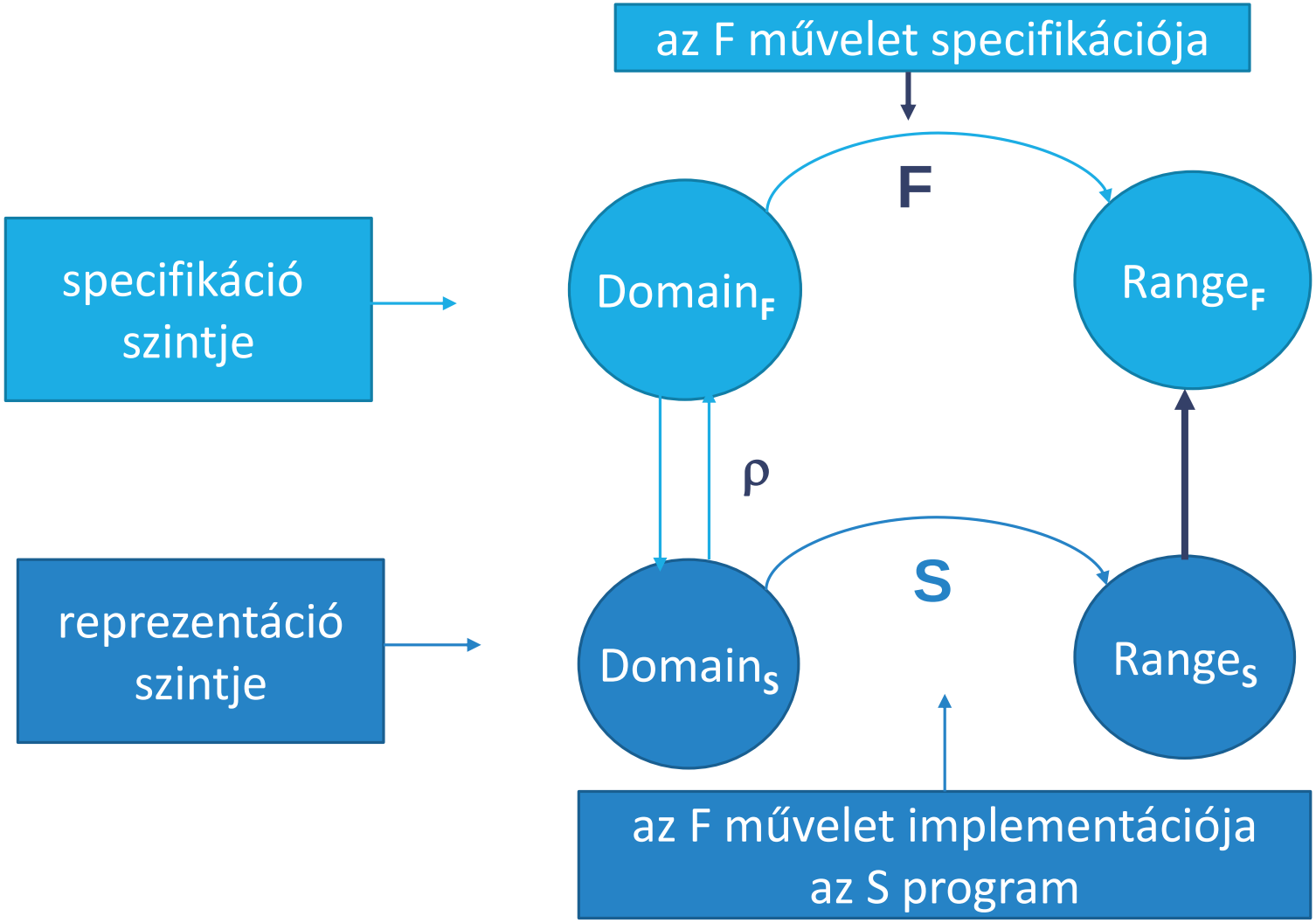
Típusspecifikáció („mit”)

- alaphalmaz: a valós világ minket érdeklő objektumainak halmaza
- specifikációs invariáns (I_S) - ezt a halmazt szűkíti
- típusműveletek specifikációja

Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

A típus specifikáció és a típus kapcsolata



Elvárások és eszközök

Elvárások a programozási nyelvekkel szemben

Absztrakt típusok megvalósításának eszközei

Elvárások a programozási nyelvekkel szemben

Modularitás

- Az egyes típusokat önálló fordítási egységekben lehessen megvalósítani!
- Ez biztosítja a típusok újrafelhasználhatóságát valamint a hatékony programfejlesztést
  - az egyes modulok könnyedén átvihetők más programokba,
  - a különböző egységeket más-más programozó is fejlesztheti
    - nem zavarva egymás munkáját

Absztrakt típusok megvalósításának eszközei

Modulokra bontás

- A professzionális használatra szánt programozási nyelvek mindegyikében megjelenik a modularizáció támogatása.
- A modularizáció alapját egyre inkább a típusokra bontás jelenti
  - azaz egy modul egy típust implementál
- De
  - sokszor van szükség „alprogram könyvtárak”-ra
  - szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is
    - lehetőség szerint az azt használó adattípus moduljában rejtünk el

Elvárások a programozási nyelvekkel szemben

Adatrejtés

- A nyelv támogassa a reprezentáció elrejtését.
- Így a nyelv biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.
- Lehetővé téve a reprezentáció, illetve az implementáció megváltoztatását, a változások a programban felfelé gyűrűzése nélkül.

Absztrakt típusok megvalósításának eszközei

A reprezentáció elrejtése

- Az adatrejtés támogatására az összetett típusok komponenseinek láthatóságát szintekre bontásával érhető el gyakran
  - Meghatározva, hogy pontosan kik férhetnek hozzá az adott komponenshez.
- Leggyakrabban három szint:
  - nyilvános (public) – az adott komponens mindenki számára látható
  - védett (protected) – az adott komponens csak a leszármazottak számára látható
  - privát (private) – a reprezentáció teljesen rejtett része, csak a műveletek implementációjában használható komponensek

Absztrakt típusok megvalósításának eszközei

Adatrejtéshez új láthatósági szintek bevezetése

- Java
  - package-private
- Delphi
  - published (futás idejű típusinformáció)
  - strict private, strict protected
- C++
  - friend
- stb.

Kérdések

Típus vagy objektum szinten szabályoz-e?
(C++ - SmallTalk)

Vannak olyan nyelvek ahol egyáltalán nincs ilyen jellegű szabályozás

- Python

Egyes nyelvekben a láthatóság még ennél is finomabban szabályozható

- a típus megvalósításakor rendelkezhetünk arról, hogy mely osztály (és leszármazottai) férhessenek hozzá az adott komponenshez.

```
class A
  feature {B, C}
     X: INTEGER;
  feature {ANY}
    make(p_name : STRING ) is
       require
            p_name /= ""
       do
            name := p_name ….
  feature {NONE}
     Y: ARRAY[INTEGER];
end
```

B és C látja (és utódaik)

public

private

Elvárások a programozási nyelvekkel szemben

Konzisztens használhatóság

- A felhasználói és a beépített típusok ne különbözzenek a használat szempontjából!
- A típusokat minél inkább a valós világban megszokott, „természetes” módon lehessen kezelni!
- Ugyanúgy lehessen összetett típusokat (tömböket, rekordokat stb.) definiálni a segítségükkel, ugyanúgy lehessen változókat definiálni a saját típusokkal stb.

Absztrakt típusok megvalósításának eszközei

Konzisztens használat

- Az új típust be lehessen illeszteni a nyelv logikájába!
  - Ha például az adott nyelvben az a konvenció, hogy egy típust a Read művelet olvas be, akkor fontos, hogy a saját típusokhoz is definiálhasson a programozó egy Read nevű műveletet
  - Azaz legyen lehetőség a Read azonosító túlterhelésére vagy átlapolására.
    - Egy azonosító túlterhelése vagy átlapolása (overloading) azt jelenti, hogy a programszöveg egy adott pontján az azonosítónak több definíciója is érvényben van.
  - Speciálisan az operátor-túlterhelésnek nagy jelentősége van abban, hogy a felhasználói típusokat természetes használatuknak megfelelően használhassuk.

Elvárások a programozási nyelvekkel szemben

Generikus programsémák támogatása.

- A programozó lehetőleg minél általánosabban írhassa meg programjait!
- A nyelv adjon lehetőséget az ismétlések minimalizálására.
  - Közvetlen kódismétlések elkerülésére
  - Magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére.
- Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

Elvárások a programozási nyelvekkel szemben

Specifikáció és implementáció szétválasztása külön fordítási egységbe

- Ekkor az adott típust használó más modulok a típus specifikáció birtokában elkészíthetők, függetlenül a tényleges implementációtól.

Modulfüggőség kezelése

- A fordító program kezelje maga a modulok közötti relációkat (egyik használja a másikat stb.).

Absztrakt típusok megvalósításának eszközei

Specifikáció és implementáció szétválasztása

- Azon nyelvekben, melyek támogatják az „egy modul egy típus” elvet, néha lehetőség van a típusspecifikációnak az implementációtól külön, önálló fordítási egységben történő leírására.
- Ez segíthet abban, hogy az egyes modulok egymástól függetlenül elkészíthetőek legyenek.

C/C++

A specifikációt fizikailag be kell másolni minden olyan fordítási egységbe, amely az adott típust használni akarja

- Így nincs igazi szétválasztásról.
- A bemásolás megkönnyítésére a specifikáció egy külön, speciális forrásfájlban (header fájl) leírható
  - az előfordító segítségével azt beemelhetjük a megfelelő fordítási egységekbe.
- Ha egy típus specifikációját többször is beírjuk egy fordítási egységbe, az hibát jelent
  - ennek kivédéséről a header fájl megírásakor a programozónak gondoskodnia kell.

C/C++

```
typedef double Angle;
class Complex {
public:
 Complex(double r=0, double i=0){ R = r; I = i;}
 Complex operator =(Complex z){ R = z.R; I = z.I; return *this; }
 Complex operator +(Complex z) { return  Complex(R+z.R,I+z.I);}
 Complex operator +(double x) { return Complex(R+x,I);}
 Complex operator *(Complex z);
 Complex operator *(double x);
 double Re();
 double Im();
 double Abs();
 Angle  Phi();
private:
   double R;
   double I;
}
```

C/C++

```
Complex operator + (double x, Complex z) {
    return z+x
}
```

Vagy: **friend**

C/C++

```
#include "complex.h"
Complex Exp(Complex z, double eps = 0.0001)
{
    Complex zi = Complex(1.0);
    Complex sum;
    double  i  = 1.0;

    while(zi.Abs() >= eps )
    {
        sum = sum+zi;
        i  += 1.0;
        zi  = (zi*z)/i;
    }
    return sum;
}
```

Más nyelvekben a fizikai szétválasztás nem lehetséges

- a fejlesztő eszköz támogatja a kód többszintű nézetét
- a felhasználás szempontjából lényegtelen részletek eltakarhatók.

A Java, C# stb. nyelvekben a specifikáció és implementáció összemosódik.

- Segítséget a dokumentációs lehetőségek jelentenek, amelyek a megfelelően dokumentált programkódból elő tudják állítani a specifikáció szöveges leírását.

Modulfüggőség kezelése

- Egy program legmagasabb szintű építőkövei a modulok.
- A program működése ezen modulok interakciója.
- Minden modul igényel szolgáltatásokat és segítségükkel más szolgáltatásokat valósít meg, így a modulok között egyfajta függőségi reláció alakul ki, s egy modul megváltozása esetén szükségessé válhat a tőle függő modulok újrafordítása.

Némelyik programozási nyelv (például a C vagy C++) teljes egészében a programozóra bízza ezen függőségek kezelését, minden fordítási egységet önállóan kezel.

- Éppen ezért vannak speciális eszközök (pl. make), amelyek kizárólagos feladata ennek a feladatnak a megkönnyítése.
- Ezek a megoldások nem tökéletesek, előfordul, hogy túl sokszor fordítanak újra valamit, vagy éppen nem veszik észre az újrafordítás szükségességét stb.

Más nyelvekben a függőségek kezelése a fordító program feladata.

- Az Ada nyelvben például a with utasítással kell megadnunk, hogy milyen más fordítási egységektől függ a szóban forgó modul.
- Ez garantálja azt, hogy a modul fordítása előtt mindazon modulok specifikációs része lefordításra kerül (ha az szükséges), amelyektől az adott modul függ.

Kérdések

Saját adattípus önálló fordítási egységként megvalósítható?

Reprezentáció elrejtése megvalósítható?

Milyen láthatósági szintek vannak?

Beépített típusokkal megegyező módon használható?

Van-e azonosító túlterhelés/ operátor túlterhelés/ free operátor?

Specifikáció és implementáció különválasztható?

Sablonok

Sablonok – bevezetés

Újrafelhasználható, könnyen karbantartható programokra van szükség.

- Ennek a jegyében készít a programozó minél általánosabb típusokat és alprogramokat, hogy azokat mind az éppen aktuális, mind egyéb programjaiban a lehető legszélesebb körben használhassa.
- Maga a programozási módszertan is olyan programozási megoldásokat tanít, amelyek általánosan használhatóak.
- Ezt a törekvést a programozási nyelvek is támogatják kisebb-nagyobb mértékben
  - Most ezeket az eszközöket gyűjtjük össze.
  - Az eszközök némelyike olyan megszokott és hétköznapi, hogy talán nem is vesszük észre, hogy ebbe a kategóriába tartoznak...

Sablonok – bevezetés

Alprogramok

- Alprogramok paraméterezése

Típusok paraméterezése

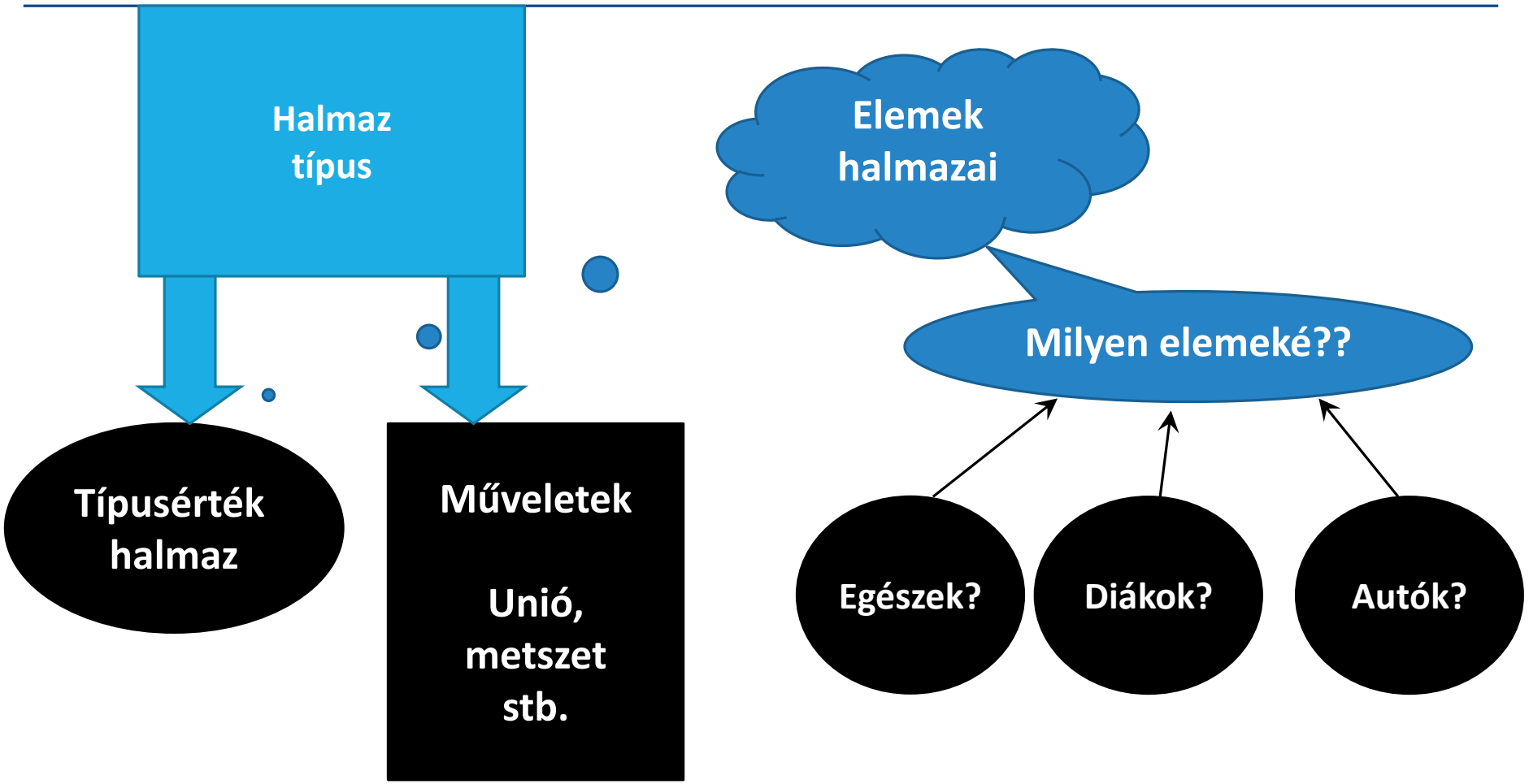
- Például az Ada-ban a diszkriminánsos rekordok, vagy a határozatlan indexhatárú tömb típusok

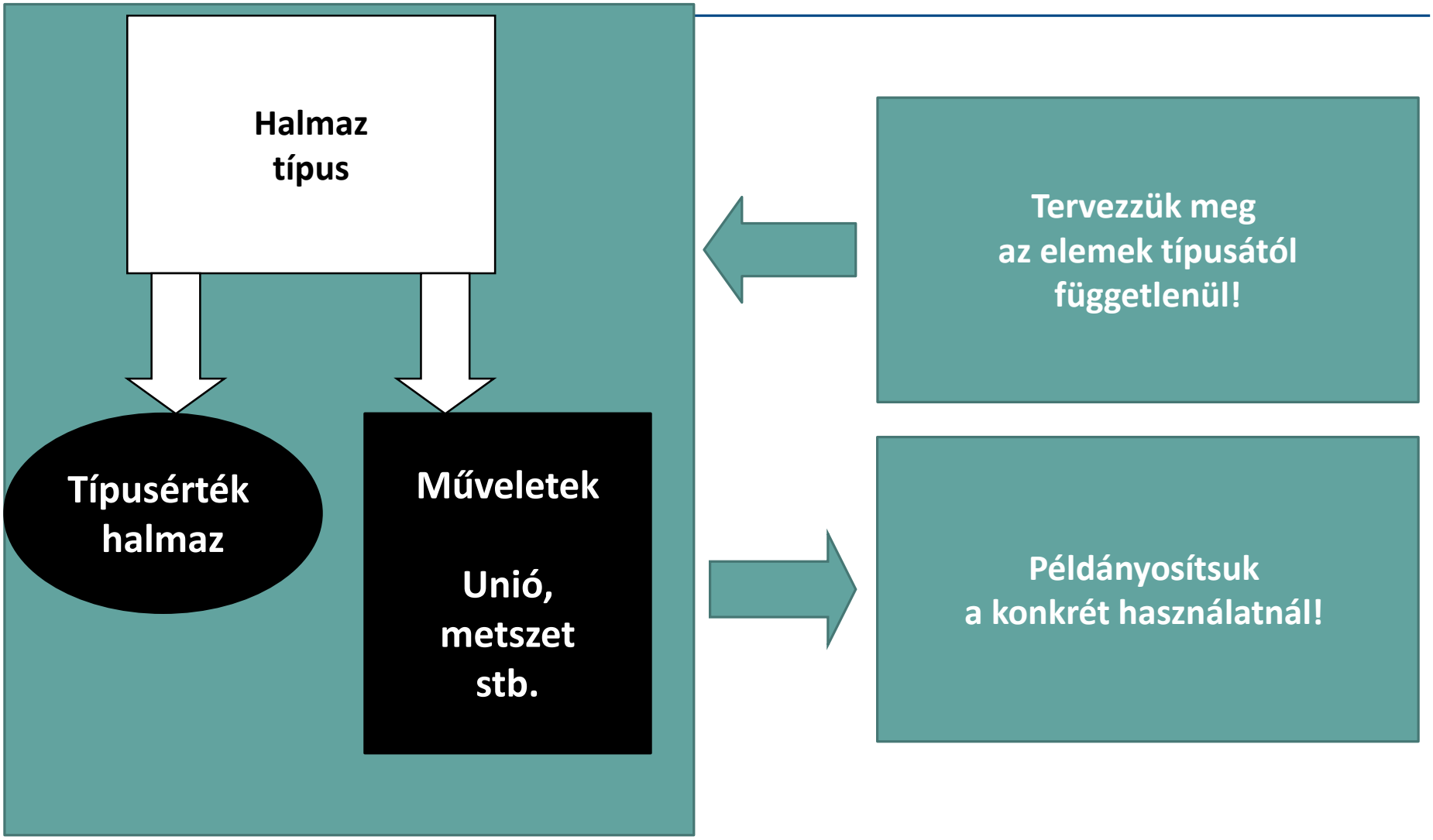
Alprogrammal történő paraméterezés

Sablonok – cél

Típusokkal történő paraméterezés

- Programozási tételeink, alapvető adatszerkezeteink és adattípusaink is „absztrakt” fogalmakat használnak
  - a számukra feltétlenül szükséges megszorításokat teszik ezekre a fogalmakra.
- Ennek következménye, hogy általában nem egyetlen típus elégíti ki ezen megszorításokat, hanem sok.
- Szeretnénk a fenti tételek, adatszerkezetek, típusok implementációját csak egyszer megadni és azt a leírást általánosan az összes, a feltételeket kielégítő típussal használni.


Halmaz típus
Típusérték halmaz
Műveletek
Unió, metszet stb.
Elemek halmazai
Milyen elemeké??
Egészek?
Diákok?
Autók?


Halmaz
típus
Típusérték
halmaz
Műveletek
Unió,
metszet
stb.
Tervezzük meg
az elemek típusától
függetlenül!
Példányosítsuk
a konkrét használatnál!

Példák

Keresés vektorban lineárisan

Maximális elem keresése vektorban

Vektor elemeinek rendezése

Verem, Sor

Prioritásos sor

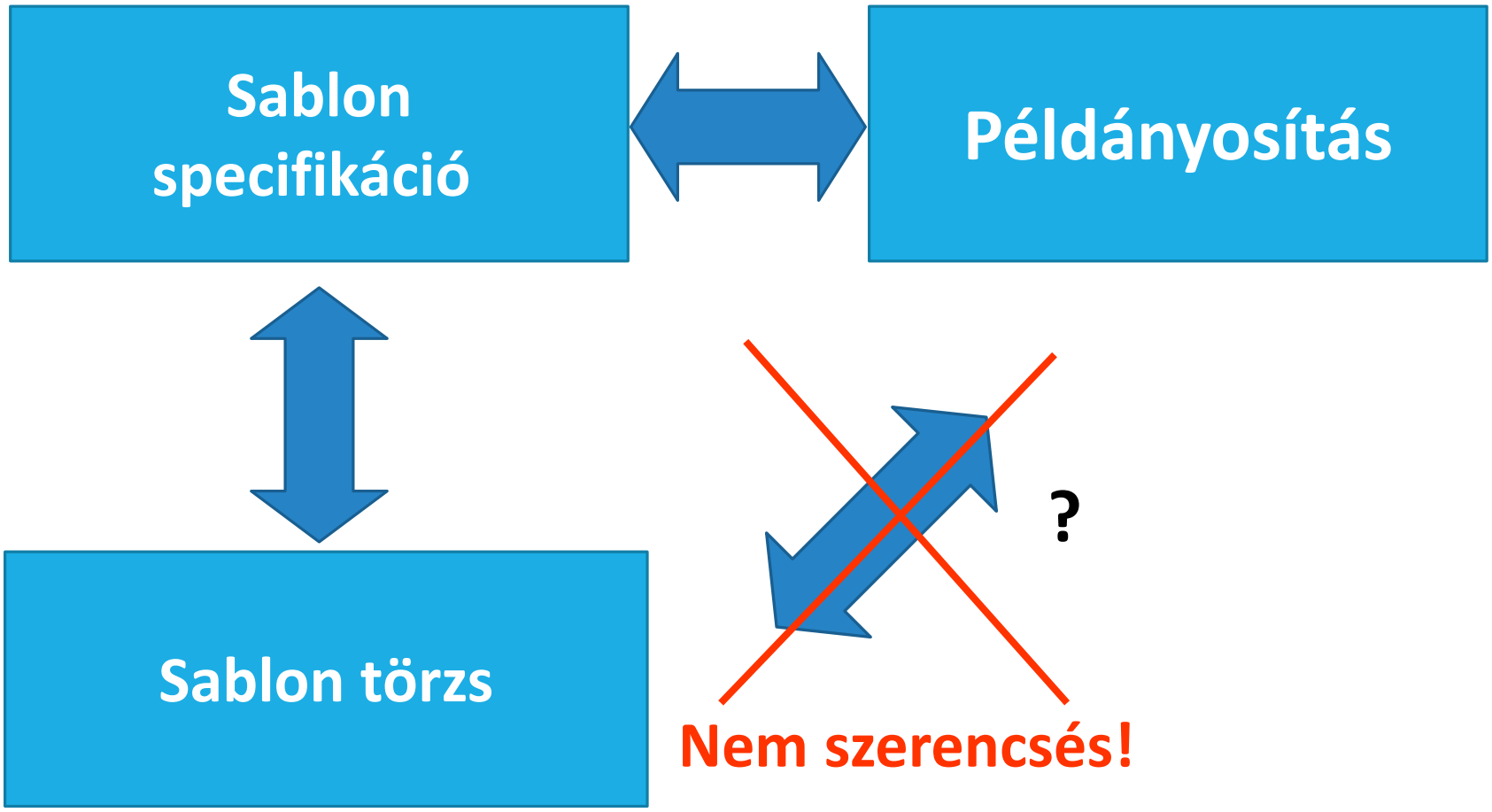
Fa

Keresőfa

Hash-tábla, stb.

Mi a különbség a példák között?

Példányosítás – összefüggések



Ezen igény kielégítésére többféle eszköz kínálkozik.

- A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten
  - hasonló hatások – korlátozott mértékben – elérhetők az előfeldolgozó rendszer használatával.
- Az előfeldolgozó rendszer szerves része a C nyelv specifikációjának
  - Nem tekinthető igazán nyelvi eszköznek, mivel a forrásnyelvű programot manipulálja
  - Közelebb áll a megíráshoz használt szövegszerkesztőhöz.

```
typedef struct cell (ELEMENT data; struct cell *next) CELL;
typedef CELL *LIST;

int find_item(ELEMENT find, LIST head, LIST *scan, LIST *prior)
{
        scan=head;
        prior=NULL;

        while (GREATHAN(find, (*scan)->data)

        {

                *prior=*scan;

                *scan=(*scan)->next;

        };

        return EQUALS(find, (*scan)->data);

}
```

Ez a megoldás független az aktuális ELEMENT típustól

El kell készíteni a megfelelő makrókat, és működik

```
/* alfabetikus eset */
#ifdef ALPHA
#define ELEMENT char*
#define LESSTHAN(x,y) (strcmp((x),(y))>0)
#define GREATHAN(x,y) (strcmp((x),(y))<0)
#define EQUALS(x,y) (strcmp((x),(y))==0)
#define ASSIGN(x,y) (strcpy((x),(y)))
#define MAXVALUE '\377'

/* short integer használata*/
#elif SHORT
#define ELEMENT short int
#define LESSTHAN(x,y) ((x)<(y))
#define GREATHAN(x,y) ((x)>(y))
#define EQUALS(x,y) ((x)==(y))
#define ASSIGN(x,y) ((x)=(y))
#define MAXVALUE 0x7fff
```

CLU

```
generic_proc_name = proc [ t : type ] (...)
                                returns (...) signals (...)
        where t has op1 : proctype (...) returns (...)
        where t has op2 : proctype (...) returns (...)
          ...
        where t has opk : proctype (...) returns (...)

sort = proc[ t : type ] ( A : array[ t ] )
                                returns ( array[ t ] )
   where t has lt : proctype (t, t) returns (bool)
```

OOP megoldás

Objektumorientált programozási nyelvekben a hatás elvileg elérhető kizárólag az öröklődés és a polimorfizmus használatával

- A szükséges közös tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba – vagy interfészbe –
- Ez közös őse lehetne az összes, a feltételeknek eleget tevő osztálynak.

A megoldás problémája, hogy

- a nyelv alapvető osztályhierarchiájának kialakításakor ismerni kellene az összes számításba jövő feltételkombinációt

Típussal paraméterezés

A típussal paraméterezést támogató nyelvek általában valamilyen önálló konstrukciót vezetnek be.

Ezekben a struktúrákban megadhatunk formális paraméterként típusokat, amelyeket aztán más típusokhoz hasonlóan használhatunk a struktúra belsejében.

Amikor a formális paramétereknek aktuális értéket adva létrehozzuk a struktúra megadott típusokhoz tartozó változatát, akkor a struktúra példányosításáról beszélünk.

C++

```
#define MAX(a,b) a > b ? a : b
MAX( x, y )*2 --> x > y ? x : y*2
#define MAX(a,b) ((a) > (b) ? (a) : (b))
MAX( ++x, y ) --> ++x > y ? ++x : y

void swap( double& x, double& y ) {
  double temp = x;
        // !! Hogy határozzuk meg x típusát?
  x = y;
  y = temp;
}
```

C++

A C++ nyelvben a típussal paraméterezés lehetőségét a template-ek biztosítják.

- Több, mint az előző makróhelyettesítés, ahol a fordító a megadott aktuális típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását.
- Bizonyos minimális szintaxisellenőrzést végez a fordító, de a formális paraméternél csak annyit tudok meghatározni, hogy az adott paraméter egy típust jelöl.

C++

Nem tudom előírni a megvalósítandó alprogram vagy típus számára fontos tulajdonságok meglétét

- Bizonyos, szükséges műveletek
- Az ebből származó hibák így a példányosításkor jelentkeznek.
- Előnye a nyelvnek, hogy a template példányosítása teljesen automatikusan történik.

C++

```
template <typename T>
void swap( T& x, T& y) {
   T temp = x;
   x = y;
   y = temp;
}
```

C++

```
template <class T>
T max(T x, T y){
     return (x>y) ? x : y;
}
```

Ennek a használata:

```
int i;
Own_typ a,b;
int j=max(0,i);       // T –t helyettesíti "int"-tel
Own_typ m=max(a,b); // Own_typ -pal
```

C++

```
double x, y = 5.1, z = 3.14;
x = max(y, z);
int i = 3, j = 4, k;
double x = 3.14, y = 4.14, z;
const int ci = 6;
k = max(i, j);  // max(int, int)
z = max(x, y);  // max(double, double)
k = max(i, ci); // max(int, int) - triviális
z = max(i, x);  // Nem egyértelmű!
```

C++

```
template <class T, class S>
T max( T a, S b) {
  if ( a > b ) return a;
  else return b;
}
int i = 3;
double x = 3.14;
double z = max( i, x);   // Ez ok, megtörténik
z == 3.0;              // A konverziók miatt a 3.14
                       //  az 3.0 lesz
```

Explicit specializáció

```
template <class R, class T, class S>
R max(T a, S b) {
  if ( a > b ) return a;
  else return b;
}
z = max<double>(i, x);   // ok, 3.14
k = max<long, long, int>(i, x);
                // Konvertál long és int -re
k = max<int, int, int>(i, j);
                // Fölösleges
```

C++

```
#include <iostream.h>
template <class T> class Vector {
     T* data;
     int size;
public:
     Vector(int);
     ~Vector() { delete[] data; };
     T& operator[](int i) { return data[i]; }
};
template <class T> Vector<T>::Vector(int n) {
     data=new T[n];
     size=n;
};
```

C++

```
main(){
   Vector <int> x(5);
   for (int i=0; i<5;++i) {
       x[i]=i;
   }
   for (i=0; i<5;++i) {
      cout  << x[i] << ' ';
   }
   cout << '\n';
}
```

C++ concept

A concept-ek lehetősége

- megfogalmazhatjuk elvárásainkat egy típussal szemben
- ezt még példányosítás előtt tud ellenőrizni a fordítóprogram
- meg tudja nevezni a hiba okát olvasható formátumban

C++20-ban bevezetésre került!

C++ concept

```
template<typename T>
concept EqualityComparable =
  requires(T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
};
```

Megjegyzés

- draft cpp szabvány esetén (8.x gcc-vel pl) a szintaxis eltérő lehet több esetben
- Itt: `concept bool EqualityComparable =`

C++ concept

```
template<EqualitiyComparable T>
void equals(T x, T y) {
    if (x==y)
        std::cout << "Azonosak";
    else
        std::cout << "Nem azonosak";
}

equals<int>(42, 42); // "Azonosak"

equals<int>(42, 10); // "Nem azonosak"
```

C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};
};

ownClass o1(10), o2(20);
equals<ownClass>(o1, o2); // HIBA!!!
```

No matching function for call to 'equals' candidate template ignored: constraints not satisfied [with T = ownClass] because 'ownClass' does not satisfy 'EqualitiyComparable' because 'a == b' would be invalid: invalid operands to binary expression ('ownClass' and 'ownClass')

C++ concept

```
class ownClass {
public:
    int member;
    ownClass(int init): member(init) {};

    bool operator == (ownClass const &other) {
        return member == other.member;
    }

    bool operator != (ownClass const &other) {
        return member != other.member;
    }
};
```

C++ concept

További megadási lehetőség

```
template<typename T> requires EqualitiyComparable<T>
void equals(T x, T y) …
```

Sőt, akár

```
template<typename T> requires !EqualitiyComparable<T>
```

Követelmények együttes megadása is lehetséges

```
template<typename T>
requires floating_point<T> && integral<T>;


template<typename T>
requires floating_point<T> || integral<T>;
```

C++ concept

Előzőekhez az összetevők:

```
template < class T >
concept integral = std::is_integral_v<T>;

template < class T >
concept floating_point = std::is_floating_point_v<T>;
```

Egymásba ágyazással:

```
template<typename T>
concept MyConcept = requires(T a) {
    requires floating_point<T>;
    requires integral<T>;
};
```

Eiffel: generic

Objektum orientált megközelítés.

A formális paraméterei osztályok lehetnek ahol a szükséges speciális tulajdonságokat azáltal lehet meghatározni, hogy megadható, mely osztályból kell származnia az aktuális paraméternek.

- Előnye: A generic belsejében használt műveletek a megadott ősosztály műveletei lehetnek.
  - A használat helyessége a generic megírásakor ellenőrizhető.
- Hátránya: a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közös ős definiálható legyen.
  - Ekkor viszont a problémák jórészt megoldhatóak az öröklődés és a polimorfizmus eszközeivel.

Eiffel

Megszorítás nélküli generic

```
deferred class TREE [G] ...
class LINKED_LIST [G] ...
class ARRAY [G] ...
```

Az osztályoknak akárhány formális generic paramétere lehet.

Típus létrehozásához egy generic osztályból:

- Minden formális generic paraméterhez kell egy típus, az aktuális generic paraméter.
- Ez egy generikusan származtatott (példányosított) típust eredményez.

Eiffel

Generic származtatások - példányosítás

```
TREE [INTEGER]
TREE [PARAGRAPH]
LINKED_LIST [PARAGRAPH]
TREE [TREE [TREE [PARAGRAPH]]]
ARRAY [LINKED LIST [TREE [LINKED LIST
[PARAGRAPH]]]
```

Eiffel

Korlátozott generic

- `class HASH TABLE [G, KEY -> HASHABLE] ...`
  - KEY –t megszorítjuk a HASHABLE osztállyal.
  - Minden T aktuális generic paraméter bázisosztálya, amit a KEY-hez használunk a HASHABLE megszorító osztály leszármazottja kell legyen.
- `HASH TABLE [PARAGRAPH, STRING]`
- A Korlátozott formális generic paraméterek általános szintaxisa:
  - `T -> Class-type`

Ada

A sablonok paraméterezhetősége sokkal szélesebb körű és rugalmasabb

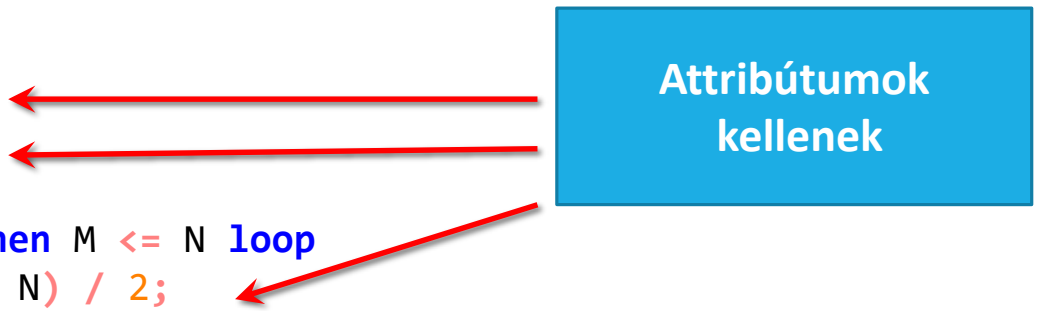
- Lehet:
  - megadott őstípusból származó típussal paraméterezni,
  - kiköthető, hogy az aktuális paraméter valamilyen típusosztályba (diszkrét típus, felsorolási típus, tetszőleges típus stb.) tartozzon.
  - megadhatók a használni kívánt további műveletek, amelyek segítségével kikerülhető a kötelező közös ős megléte.
  - az explicit megadott műveletek segítségével a generic még rugalmasabb használatára nyílik lehetőség, megtehető például, hogy egy rendezési relációt használó sablont az egész számokkal példányosítva nem a szokásos rendezési relációt adjuk meg, hanem például az oszthatóság parciális rendezési relációját.

Ada

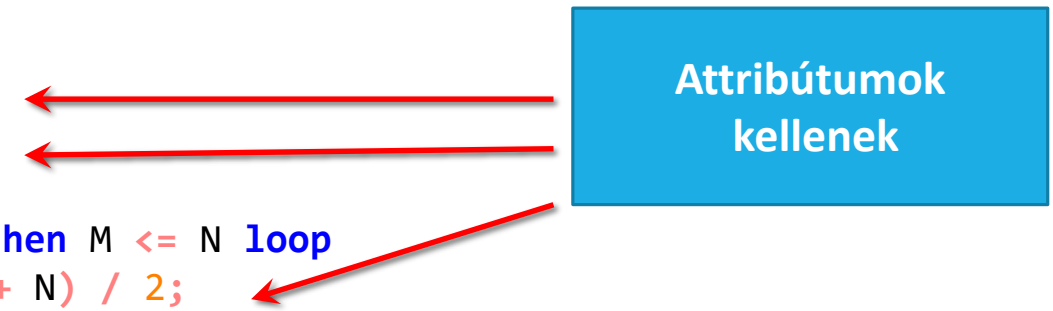
```
generic
      type Item is private;
      type Index is (<>);
      type Vector is array (Index range <>) of Item;
      with function "<"(X, Y : Item) return Boolean is <> ;

procedure Log_Search(V: in Vector; X: in Item;
                     Found: out Boolean; Ind: out Index);
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```

```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



```
procedure Log_Search(V:in Vector; X:in Item Found: out Boolean; Ind:out Index) is
    M, N, K : Integer;
    begin
        M := Index'Pos(V'First);
        N := Index'Pos(V'Last);
        Found := False;
            while not  Found and then M <= N loop
                        K := (M + N) / 2;
                        if  X < V(Index'Val(K)) then
                                N := K - 1;
                        elsif X = V(Index'Val(K)) then
                                Ind := Index'Val(K);
                                Found := True;
                        else
                                M := K + 1;
                        end if;
                end loop;
    end Log_Search;
```



Java

Generic bevezetése

```
List<Integer> myIntList =
                              new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.iterator().next();
```

Java

Ehhez a java.util-ban példa

```
public interface List <E> extends Collection<E>
{
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E>{
    E next();
    boolean hasNext();
}
```

Java

```
public class Box<T> {
     private T t;                          // T: Típus
     public void add(T t) { this.t = t; }
     public T get() { return t; }
}
Box<Integer> integerBox;
integerBox = new Box<Integer>();
```

Java 7-től:

```
Box<Integer> integerBox = new Box<>();
```

- is engedélyezett, ha a fordító ki tudja következtetni

Java

A használata:

```
public class BoxDemo3 {
  public static void main(String[] args) {
    Box<Integer> integerBox = new Box<Integer>();
    integerBox.add(new Integer(10));
    Integer someInteger = integerBox.get();
    System.out.println(someInteger);
  }
}
```

Java

generic metódusok lehetősége is megvan:

```
public class Box<T> {
    private T t;
    public void add(T t) { this.t = t; }
    public T get() { return t; }
    public <U> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }
}
```

Java

Az eredmény:

- `T: java.lang.Integer`
- `U: java.lang.String`

Java

Típustörlés – nyers (raw) típusok

- `Box rawBox = new Box();`
  - ez is lehetséges – ezzel a generic előtti viselkedést kapjuk…
- `Box` a nyers típusa a `Box<T>`-nek

A korábbi programok így gond nélkül működhetnek

Futási időben minden generic `Object`-tel paraméterezett lesz

- Az ellenőrzés fordításkor történik

Java

Megszorított típusparaméter lehetősége:

- `<U extends Valami>`
  és akkor az aktuális típusparaméter U-ra csak a Valami leszármazottja lehet
- vagy, ha interfész megvalósítását is kérjük:
  - `<U extends Valami & MyInterface>`
- A generic törzsében számíthatunk a Valami és a MyInterface műveleteire!
- Mindig először kell az osztály, azután az interfészek

Öröklődéssel való kapcsolata később, az OOP-nél!

Java

Példa:

```
public class Gen1<A extends Number> {
  public Gen1(A aa) { a = aa; }
  public A getElem() { return a; }
  public void setElem(A aa) { a=aa; }
  public int egesz() { return a.intValue(); }
  protected A a;
}
Gen1<Double> g2 = new Gen1 <Double>(2.3);
g2.egesz();
```

Java – Típushelyettesítő

Tételezzük fel, hogy

```
public class Gen<T> {
  public Gen(T st) {storage = st; }
  public T getElem() { return storage; }
  public void setElem(T st) { storage = st; }
  protected T storage;
}
```

És készítünk egy f fv-t:

```
static <A,B> Gen f(Gen<A> ga, Gen<B> gb) {
  Object o1 = ga.getElem();
  Object o2 = gb.getElem();
  if (o1.toString().length() < o2.toString().length() )
     return ga;
  else
     return gb;
}
```

Mi legyen a visszatérési típusa?

Java – Típushelyettesítő

Probléma:

```
Gen<> g = f(new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- ez fordítási hibás!

```
Gen g = f( new Gen<Integer>(1), new Gen<Double>(2.0) );
```

- Ok, de ez „nyers” (raw) típus…

```
Gen<Number> gi = f(new Gen<Integer>(1),
                                        new Gen<Double>(2.0) );
```

- ez rendben, de warning

Típushelyettesítő

```
Gen<?> g1 = f( new Gen<Integer>(1), new Gen<Double>(2.0));
```

Java – Típushelyettesítő

Még jobb, ha már a definíciójába tesszük:

```
static Gen<? extends Number> max(
                                Gen<? extends Number> ga,
                                Gen<? extends Number> gb) {
  Number n1 = ga.getValue();
  Number n2 = gb.getValue();
  if ( n1.doubleValue() < n2.doubleValue() )
    return ga;
  else
    return gb;
}
```

Java további lehetőségek

Nemcsak felső megszorítás adható

Vegyük az alábbi függvényt

```
public void append(
            Collection<Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

A következő listát hogyan tudjuk paraméterként átadni?

```
List<Number> numbers = new ArrayList<Number>();
```

Java további lehetőségek

A felső megszorítás értelemszerűen ekkor nem jó.

```
public void append(
    Collection<? super Integer> integers, int n) {
    for (int i = 1; i <= n; i++) {
        integers.add(i);
    }
}
```

```
List<Number> numbers = new ArrayList<Number>();
append(numbers, 5);
```

Java generic és a new

A kapott típus szerinti objektumok létrehozása – probléma

```
class Dummy<T> {
    public T getInstance() throws Exception {
        new T();
        T.newInstance();
        T.class.newInstance();
    }
}
```

Egyik sem lehetséges

- Az megfejtés abban rejlik, hogy a típustörlést követően Object került a T helyére és nem az aktuális típusparaméter
  - Tehát a formális ellenőrzésen átmegy, de a törlést követően más a viselkedése

Java generic és a new

Generic tömbök létrehozása – nem lehetséges

```
T[] ts = new T[n]
```

Ennek a megfejtése más

- A Java tömbök kovariánsak (erről még lesz szó). Lehetséges:

```
Integer[] myInts = {1,2,3,4};
Number[] myNumber = myInts;
```

  - A tömbökhöz futásidőben elérhető típusinformáció tartozik, a fenti esetben a `myNumber` változó mögötti tömbbe nem lehetséges a `4.5` érték berakása. Ezt futásidőben ellenőrzi és megakadályozza a VM. (Ez rendben is van, a tömbök definícióját teljesíti.)
  - Generikus tömb esetén azonban a típusinformáció a típustörlés miatt nem áll rendelkezésre, azaz `Object` lesz minden. Így a fenti típusellenőrzés nem lehetséges futásidőben, ami baj.

C#

Sok szempontból hasonlít a Java megoldásaira

- Szintaktika
- Megszorítások

Különbség azonban, hogy C# esetén nincsen típustörlés

- E tekintetben a C++ template-hez hasonló a működés
- Minden típusparaméter-értékhez „natív” kódot generál

C#

```
public class LinkedList<TKey, TData> where TKey : IComparable
{
    …
        public bool Find(TKey key)   {
            Node<TKey,TData> current = _head;
            while (current.NextNode != null)
            {
                if (current.Key.CompareTo(key) != 0)
                    current = current.NextNode;
                else
                    break;
            }
            return (current.Key.CompareTo(key) == 0);
        }
}
```

Kérdések:

Milyen nyelvi elemekből tudunk sablonokat létrehozni?

- Új adattípusokból és/vagy alprogramokból?

Milyen paraméterei lehetnek ezeknek a mintáknak?

- Típusok, objektumok, alprogramok?
- Létrehozható paraméter nélküli sablon?

Milyenfajta típusok lehetnek aktuális típus-paraméterek?

- Csak beépített típusok és altípusaik, vagy felhasználó által definiált típusok is?

Kérdések

Adhatunk megszorításokat a formális generic paramétereknek?

Egymásba ágyazható ez a konstrukció?

A példányosítás fordítási (statikusan) vagy futási időben (dinamikusan) történik?

Kivételkezelés

Program készítésének célja

A célunk jó minőségű program készítése

Alapvető elvárás

- helyes – a specifikációnak megfelelően működik
- robosztus – nem várt helyzetekben sem történik katasztrófa

Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Hardverhibák

- elromlott winchester miatti leállások,
- elfogy a memória, stb.

Hibalehetőségek

A programok futása közben hibák léphetnek fel.

Szoftverhibák

- a futtatott programban,
  - kísérlet nullával való osztásra
  - aritmetikai túlcsordulás, alulcsordulás
  - üres pointer/referencia dereferencia kísérlete
  - tömb hibás indexelése
  - hibás típus konverzió kísérlete
  - hibás input stb.
- az operációs rendszerben meglévő programozói hibák.

A program

megbízhatóságának növelése

érdekében

figyelni kell a hibákra

és meg kell próbálni

megelőzni,

vagy ha már bekövetkeztek,

kezelni

őket!

Hibakezelés históriája

Hogyan kezelték régebben?

- Abortált a program
- Minden alprogram valamilyen jelzést (hibaértéket) ad vissza
  - A lefutás OK
  - A lefutás nem sikerült

Példa

Tegyük fel, hogy szeretnénk beolvasni egy fájlból

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real);
begin
    ReadByte(f,n1);
    ReadInt(f,n2);
    ReadLongInt(f,n3);
    ReadReal(f,n4);
end ReadFile;
```

Ellenőrzés lépésenként

Ha ellenőrizni akarjuk:

```
ReadFile(f: in File; n1: out byte, n2: out integer,
                     n3: out longint, n4: out real): boolean;
success:boolean;
begin

  success:=ReadByte(f,n1);
    if success then success:=ReadInt(f,n2);
      if success then success:=ReadLongInt(f,n3);
         if success then success:=ReadReal(f,n4);

            if not success then
              // ReadReal hiba kezelése,
            end if;

         else … ReadLongInt hiba kezelése end if ;
      else … ReadInt hiba kezelése end if ;
    else … ReadByte hiba kezelése end if ;

  return success;

end ReadFile;
```

Meggondolások

Nem biztos, hogy függvényt lehet/érdemes csinálni az eljárásból

- Lehet, hogy nem visszatérési érték, hanem egy paraméter jelzi a sikert.

A kód sokkal bonyolultabb, az eredeti cél szinte elvész

- A későbbi karbantartás sokkal nehezebb!

Nem biztos, hogy a hívó figyeli a sikert jelző értéket!

Meggondolások

Lehetne, hogy globális változót használunk a lefutás helyességének figyelésére

- Osztott környezetben beláthatatlan következmények!

Hiba esetén meghívunk egy hibakezelő alprogramot

- Mi történik, ha nem tesszük meg?

Elvárások

A hibák kezelése kapcsán a következő elvárásokat lehet megfogalmazni

1. meg tudjuk különböztetni a hibákat,
2. a hibát kezelő kód különüljön el a tényleges kódtól,
3. megfelelően tudjuk kezelni - a hiba könnyen jusson el arra a helyre, ahol azt kezelni kell,
4. kötelező legyen kezelni a hibákat

KIVÉTELKEZELÉS

Megoldás elemzése

Az első kikötés, hogy meg tudjuk különböztetni a hibákat.

- Ez általában a fellépés helyén történik.

A második kikötés azt szeretné elérni, hogy a programot először látó ember is azonnal el tudja különíteni a hibakezelő és a működésért felelős részeket.

- Ez könnyebben továbbfejleszthető, karbantartható programokhoz vezet.

Példa

```
gyumolcs {
    alma();
    //...
    korte();
    //...
    barack();
}
```

Példa hagyományos módon

```
gyumolcs {
   ret1=alma();
   if (ret1==ok) {
      //...
      ret2=korte();
      if (ret2==ok) {
         //...
         ret3=barack();
         if (ret3==ok) {
            return ok;
         }
         else { /* 3. hiba kezelése */}
      }
      else { /* 2. hiba kezelése */ }
   }
   else {/* 1. hiba kezelése */ }
}
```

A példa kivételkezeléssel

```
class AlmaException extends Exception {}
class KorteException extends Exception {}
class BarackException extends Exception {}


gyumolcs {
   try {
      alma(); /*küldhet AlmaException-t*/
      korte();
      barack();
   }
   catch(AlmaException hiba1) { /* 1. hiba kezelése */ }
   catch(KorteException hiba2) { /* 2. hiba kezelése */ }
   catch(BarackException hiba3) { /* 3. hiba kezelése */ }
}
```

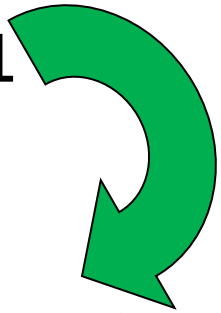
A hiba jusson el oda, ahol kezelni kell

A harmadik kikötés arra vonatkozik, hogy egy alacsonyabb szinten jelentkező hibát nem biztos, hogy az adott szint le tud kezelni, ezért a megfelelő helyre kell eljuttatni.

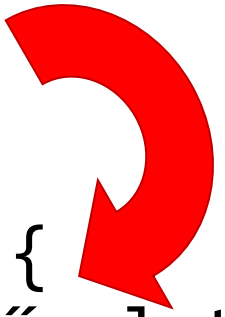
- A programok többszintűek
- Példa
  - egy fájlt nem tudunk megnyitni
  - egy üres veremből akar valaki kiolvasni

Példa

```
gyumolcs {
     alma();
     //folytatás1
}
```



```
          alma {
               jonatan();
               //folytatás2
          }
```



```
                    jonatan {
                         //műveletek
                    }
```

Hiba terjesztése hagyományosan

Tegyük fel, hogy a `jonatan()`-ban fellépő hibát a `gyumolcs()` kell lekezelje:

```
gyumolcs {
    ret=alma();
    if(ret!=ok) {
        //hibakezelés
    }
    else {
        //folytatás1
    }
}
```

```
alma {
    ret=jonatan();
    if(ret!=ok) {
        return hiba;
    }
    else {
        //folytatás2
    }
}
jonatan {
    //műveletek
    if( művelethiba ) {
        return hiba;
    }
}
```

Hiba terjesztése kivételkezeléssel

Kivételkezeléssel a felfelé terjesztést egyszerűen és kevesebb módosítással lehet megoldani:

```
gyumolcs {
    try {
        alma();
        //folytatás
    }
    catch(VmiExc) {
        //hibakezelés
    }
}
```

```
jonatan throws VmiExc {
    //műveletek
}
```

```
alma throws VmiExc {
    jonatan();
    //folytatás
}
```

Kivétel és nem hiba

```
type Napok is (Vasarnap, Hetfo, Kedd, … Szombat);

function Holnap(Ma: Napok) return Napok is
begin
   return Napok'Succ(Ma);
exception
   when Constraint_Error => return Napok'First;
end Holnap;
```

Kérdések

Hiba- vagy kivételkezelést ad-e a nyelv?

Hogyan kell kivételeket definiálni-kiváltani-kezelni?

Milyen a kivételkezelés szintaktikai formája

Mihez kapcsolódik a kezelés: utasítás-, blokk- vagy eljárás/függvényszintű?

A kivételekből lehet-e kivételcsoportokat, kivételosztályokat képezni, amelyeket a kivételkezelőben egységesen lehet kezelni?

Kérdések

Paraméterezhető-e a kivétel információ továbbadás céljából?

Támogatja-e a nyelv explicit módon valamilyen végső tevékenység („finally”) megadását?

- Ha nem, akkor tudjuk-e szimulálni azt?
- Milyen megszorításokkal?

A nyelv biztosít-e olyan nyelvi konstrukciót, amelynek nem kell explicit módon megadni, hogy mely kivételt kell továbbadni?

Kérdések

Újrakezdhető-e az alprogram (blokk) a hiba kezelése után?

Megadható-e /meg kell-e adni, hogy egy alprogram milyen lekezeletlen kivételeket küldhet?

Párhuzamos környezetben vannak-e speciális kivételek?

Mi történik a váratlan kivételekkel?

Kiváltható-e kivétel kivételkezelőben?

Melyik kivételkezelő kezeli a kivételt?

C++

Kivétel lehet egy objektum, aminek tetszőleges a típusa.

Hasznos, ha definiálunk osztályokat a felhasználó kivételeihez

Például

- `class MyException {};`

C++

Néhány predefinit kivétel:

- `bad_cast`: ha a dynamic_cast operator nem használható egy referenciára,
- `bad_typeid`: ha a typeid operátora egy null pointer dereferenciája.
  (mindkettő a typeinfo.h-ban deklarálva).

C++

Kivétel kiváltása

- `throw [expression] ";"`
  A kifejezés egy időszakos objektumba kerül.
- A normál program lefutás megszakad, a vezérlés a legközelebbi megfelelő kezelőhöz kerül
- Ha nincs kifejezés: csak kivételkezelőben, illetve innen hívott függvényben lehet, az aktuális kivétel újrakiváltása.
- A fordító nem biztos, hogy ellenőrzi, ha `throw` utasítás kivétel-objektum nélkül: `terminate` függvény hívása.

C++

Példák

- `throw 5;`
- `throw "An exception has occurred";`
- `throw MyException();`
- `throw;`

További információk: a kivételosztályban lehetnek nyilvános attribútumok, konstruktor beállíthatja stb.

```
class ExceptionWithParameters {
 public:
  int a,b;
  ExceptionWithParameters(int a0,int b0)
                                  { a=a0; b=b0; }
  };

  throw ExceptionWithParameters(0,1);
```

C++

Kivételek kezelése - try blokkal:

- `try compound_statement handler_list`
  `ahol`
  `handler_list = handler{handler}`
- `handler = catch "(" exception_declaration ")"`
  `compound_statement`
  `exception_declaration = type [ident] |"..."`

C++

A dobott kivételeknek megfelel egy `catch` ág, ha a következő feltételek közül valamelyik teljesül:

- A két típus pont ugyanaz.
- A `catch` ág típusa publikus bázisosztálya az eldobott objektumnak.
- A `catch` ág típusa mutató, és az eldobott objektum olyan mutató, melyet valamely standard mutató konverzióval át lehet konvertálni a `catch` ág típusára.

A kezeletlen kivételek továbbgyűrűződnek a hívó `try` blokkban, majd az azt hívóban.

- A legkülső `try` blokk után a *terminate* függvény kerül meghívásra.

C++

Például

```
class A {};
class B: public A {};
class C: public B {};
class D: public A {};
class X {};
class AX: public A, public X {};

catch (A)          // A, B, C, D, AX típusúakat kap el
catch (A*)         // A*,B*,C*,D*,AX*
catch (X&)         // X és AX
catch (const B&) // B és C
catch (char*)      // char*
catch (void*)      // tetszőleges pointer típusút
```

C++

Mindig az első kezelőt választja ki

- ha rossz sorrendet írunk, nem hiba!

```
try { // ...
  }catch (A) {/* ... */}
   catch (B) {/* ... */}//soha nem jön ide!

try {// ...
  }catch (B) { /* ... */ }
   catch (A) { /* ... */ } // így OK.
```

C++

Lehet a kivételobjektumra is hivatkozni:

```
catch (ExceptionWithParameters e) {
    cout<< e.a; // kiírja
}
```

Lehet ...

- ez bárminek megfelel – utolsó legyen a `try` blokkban

A `terminate` hívása általában abort-ot jelent

- A felhasználó a `set_terminate`-tel megadhat mást is
- ez is le kell állítsa a programot.

Miközben a vezérlés átadódik a kivételkezelőnek, destruktort hív minden automatikus objektumra, ami a `try` blokkba történő belépés óta keletkezett.

C++

Kivétel specifikációk

- `throw "(" [type {"," type}] ")"`

Például: `void f(int x) throw (A,B,C);`

Az f függvény csak A,B és C típusú kivételt generál, vagy azok leszármazottait.

- `int f2(int x) throw ()` - f2 nem válthat ki kivételt!
- `void f3(int x)` - f3 bármit kiválthat!

Ha specifikáltunk lehetséges kivételtípusokat, akkor minden más esetben a rendszer meghívja az `unexpected()` függvényt

- Az alapértelmezett viselkedése a `terminate()` függvény meghívása
- Ez a `set_unexpected` segítségével szintén átállítható.

C++

Példák:

```
class X {};
class Y: public X {};
class Z {};
void A() throw(X,Z) // A X,Y,Z-t dobhat
{ /* ... */ }
void B() throw(Y,Z) {
   A();      // unexpected, ha A X-t dob, ok Y, Z típusúra
}
void C();    // semmit sem tudunk C-ről
void D() throw() {
   C();      // ha C bármit kivált, unexpected -t hív
   throw 7; // unexpected-t hív
}
```

Java

A C++ szintaxistól nem sokban különbözik.

- Eltérések a szemantikában.

```
try {
     throw new EgyException ("parameter");
}
catch (Type variable) {

}
finally {

}
```

Java

A továbbiakban csak az eltérések:

- **`finally`** nyelvi konstrukció, ezzel a Java megbízhatóbb programok írását segíti elő.

Egy függvény által kiváltható kivételek specifikálása:

- `void f (int x) throws AException, BException;`

Egy szálon futó program esetén ha a virtuális gép nem talál a kivétel kezelésére alkalmas kódot, akkor a VM és a program is terminál.

- Ha többszálú a program, akkor egy `uncaughtException()` metódus fut le.

Java

Minden kivétel a `java.lang.Throwable` leszármazottja.

Ha olyan kivételt szeretnénk dobni, amely nem a `Throwable` leszármazottja, akkor az fordítási hibát okoz.

A kivételek két nagy csoportba sorolhatóak

- ellenőrzöttek: `Exception` leszármazottjai
- nem-ellenőrzöttek: `Error` leszármazottjai

Java

Számos olyan kivétel van, ami

- előre nem látható
- És fölösleges lenne mindenhol lekezelni őket.
- Ezeket a kivételeket nevezzük nem-ellenőrzött kivételeknek.
  - Például nem ellenőrizzük le minden utasítás végrehajtása előtt, hogy van-e elég memóriánk stb.

Ez az oka a kivételek két csoportjának.

- A csoportosítás nem konzisztens
  - Az `Exception` egyik leszármazott osztálya, a `RuntimeException` és leszármazottjai sem ellenőrzöttek.

Az ellenőrzött kivételek esetén fordítási hiba lép fel, ha nincsenek specifikálva vagy elkapva.

- Továbbá ha olyan ellenőrzött kivételt kívánunk elkapni, amely hatókörön kívül van.

Java

Néhány predefinit nem ellenőrzött kivétel

A `RuntimeException` leszármazottjai

- `ArithmeticException`
- `ClassCastException`
- `IndexOutOfBoundsException`
  - `ArrayIndexOutOfBoundsException`
  - `StringIndexOutOfBoundsException`
- `NullPointerException`.

Az `Error` leszármazottjai

- `OutOfMemoryError`
- `StackOverflowError`
- stb.

Java példa

```
class VeremException extends Exception {}
class VeremMegteltException extends VeremException
{
      private int utolso;
      public VeremMegteltException (int i) {
            utolso = i;
      }
      public int miVolt () {
            return utolso;
      }
}
```

Java példa

```
class Verem {
  final static public int MERET = 10;
  private int tarolo [] = new int[MERET];
  private int mutato = 0;
  public void betesz (int i) {
    try {
      if (mutato < MERET)
        tarolo [mutato++] = i;
      else
        throw new VeremMegteltException (i);
    } catch (VeremMegteltException e) {
      System.out.println( e.miVolt () + " nem fert be");
      throw;
    }
    finally {
      System.out.println ("finally: mindig lefutok");
    }
  }
}
```

Mi a gond ezzel?

Java

Predefint kivételek előfordulása:

```
class A { }
class B extends A { }
class C {
   void X() {
      A a = new A;
      B b = (B)a; // ClassCastException
   }
   void Y() {
     int ia[] = new int[10];
     for (int i=1; i<=10; i++)
       ia[i]=0;  // amikor i==10 lesz: ArrayIndexOutOfBoundsException
   }
   void Z() {
     C c=null;
     c.X();       // NullPointerException
   }
}
```

Java

Kivételek kezelése

```
try {
  // utasítások
}
catch (MyException e) {
  // utasítások MyException kezelésére
}
catch (AnotherException e) {
  // utasítások AnotherException kezelésére
}
finally {
  // mindig végrehajtódó utasítások
}
```

Java

```
class A extends Exception {}
class B extends A {}

try {
    // ...
}
catch (A a) { /* ... */ }
catch (B b) { /* ... */ }

void MyMethod() throws MyException {
    // utasítások
    throw new MyException();
    // utasítások
}
```

Java

```
class ExcA extends Exception {}
class ExcD extends RuntimeException {}
class ExcF extends ExcA {}

void A() throws ExcA {
    // ...
    throw new ExcA();
    // ...
}
```

Java

```
void B1() {
    A(); // Hiba: ExcA nincs lekezelve B1-ben
}
void B2() {
  try {
     A(); // OK
  } catch (ExcA e) {
     // exception kezelése
  }
 }
void B3() throws ExcA {
     A(); // OK
}
```

Java

```
void D() throws ExcD {
    // ...
}
void E() {
    D(); // ok, ExcD leszármazottja
         // RuntimeException-nak,
         // így nem kell jelezni
}
void F() throws ExcA {
      throw new ExcF();    // OK
}
```

Java try-with-resources

További hasznos lehetőség a finally használata helyett:

```
try (BufferedReader br =
        new BufferedReader(new FileReader(path)))
{
    return br.readLine();
}
```

Ehhez a `Closeable` vagy `AutoCloseable` interfészt kell megvalósítania a használandó osztálynak

OOP

KÖVETKEZŐ HÉTEN