Programozási nyelvek és módszerek

4. ELŐADÁS – TÍPUSABSZTRAKCIÓ, TÍPUSPARAMÉTER

Procedurális- vagy adatabsztrakció

Egy szoftver rendszer mindig végrehajt

- bizonyos tevékenységeket
- bizonyos adatokon.

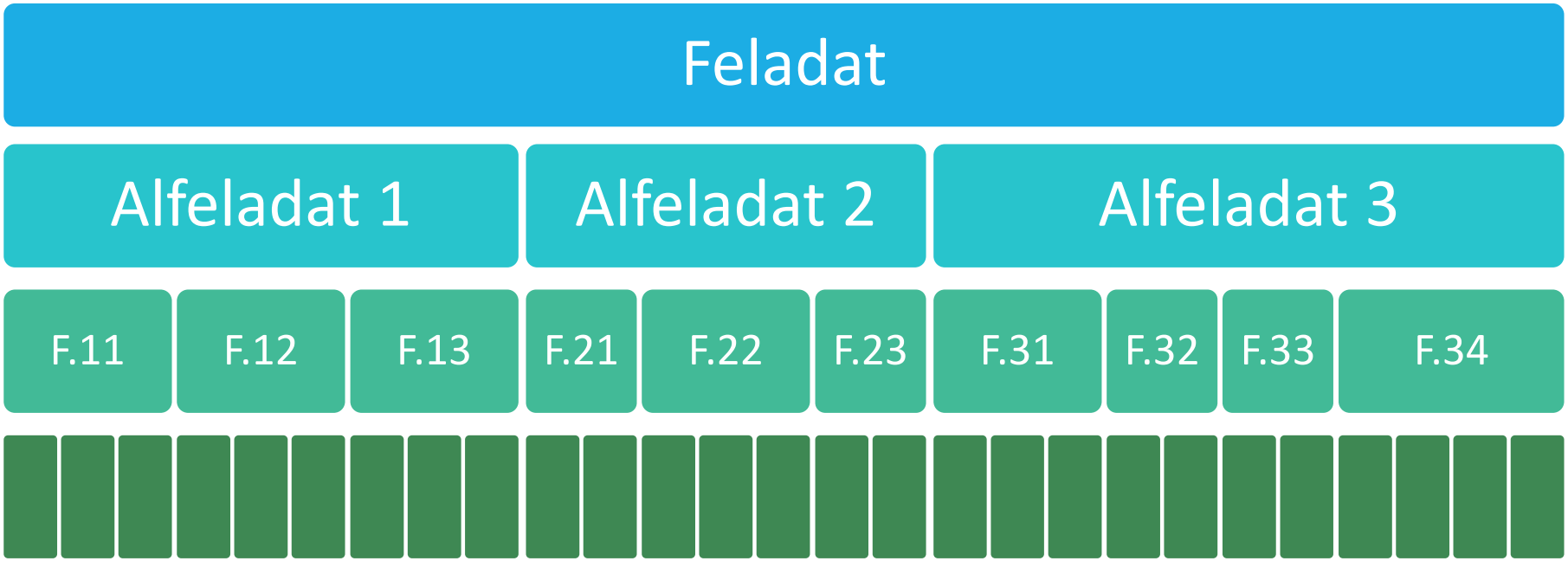
A tervezés központi kérdése, hogy a rendszer felépítését mire alapozzuk

- a végrehajtandó tevékenységekre, vagy
- az adatokra?

Procedurális absztrakció

Top-down megoldás

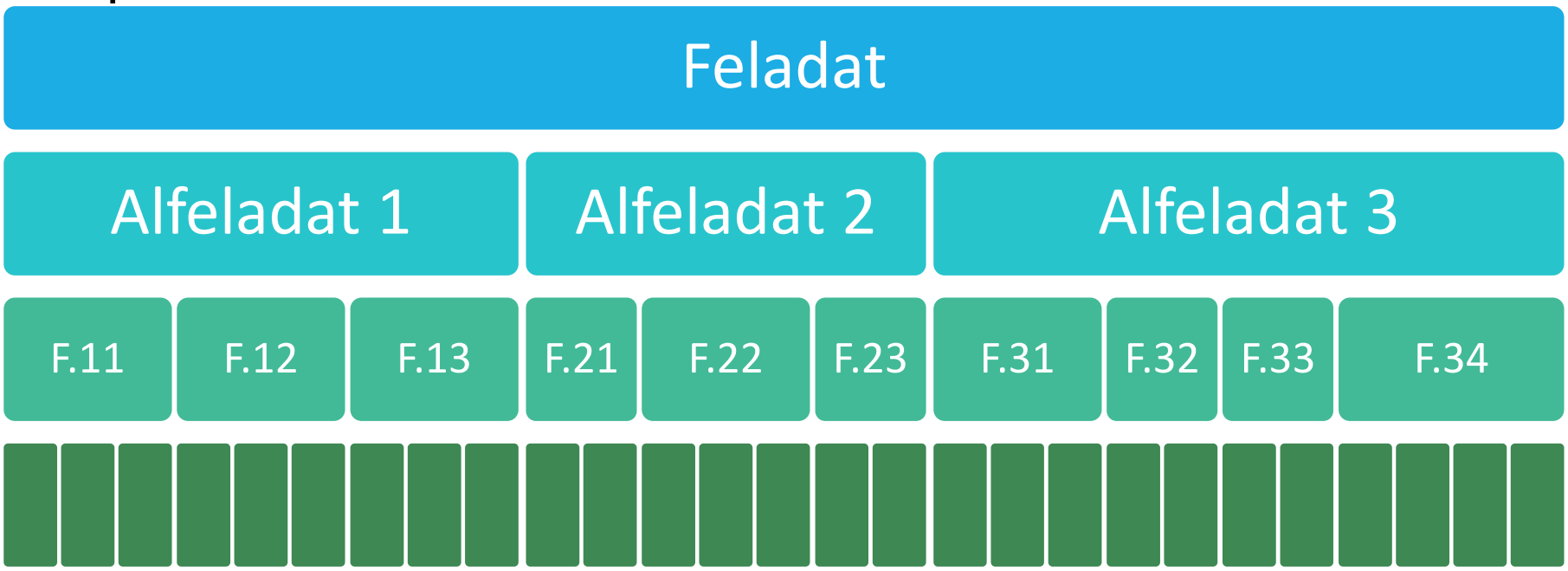
- A megközelítés hátránya: ha változik a specifikáció …



Procedurális absztrakció

Mivel egy alprogram specifikáció a magasabb szintű specifikációtól függ, így a változás tovább gyűrűzik lefelé, és végül egy egész kis változásnak nagyon nagy hatása (és költsége) lehet

- Specifikáció változásának hatása

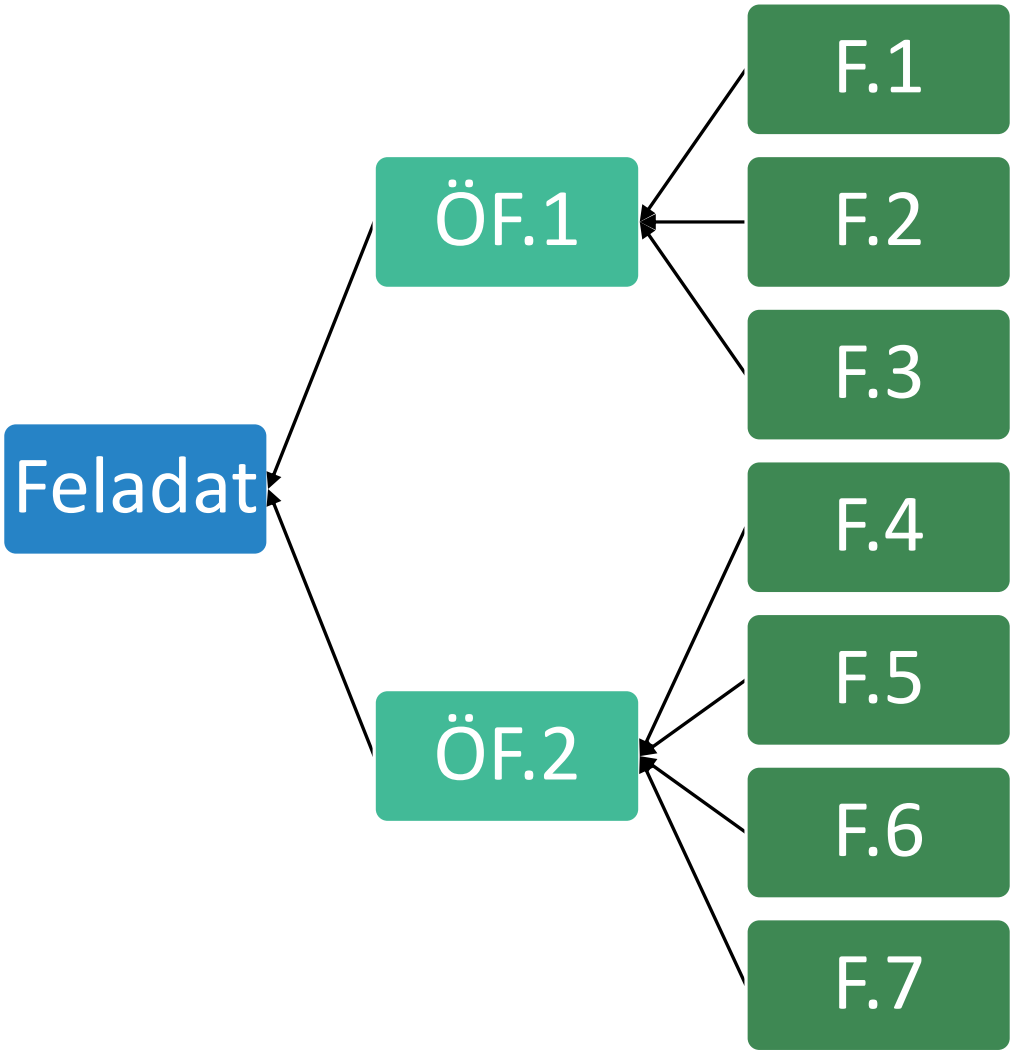


Adatabsztrakció

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

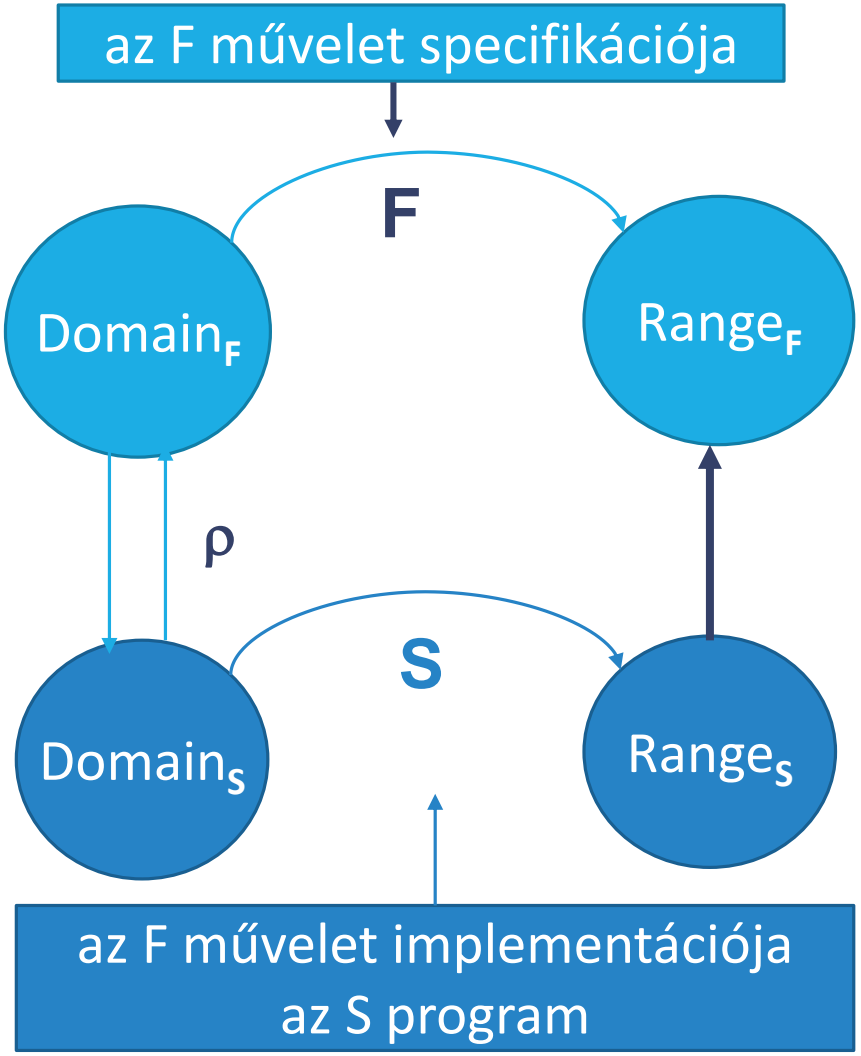
Típus megvalósítás („hogyan”)

- Reprezentációs függvény
- Típus invariáns
- Típusműveletek megvalósítása

A típus specifikáció és a típus kapcsolata

specifikáció szintje

reprezentáció szintje



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, és 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 modernebb nyelvekben a modularizáció alapját egyre inkább a típusokra bontás jelenti, azaz egy modul egy típust implementál.
- Teljesen ezen az alapon csak kevés nyelv működik
  - sokszor van szükség “alprogram könyvtárak”-ra
  - szükség lehet csak implementációs célokat szolgáló típusok megvalósítására is, amelyeket ilyenkor lehetőség szerint az azt használó adattípus moduljában rejtünk el.

Elvárások a programozási nyelvekkel szemben

Adatrejtés

- A nyelv támogassa a reprezentáció elrejtését.
- Ezen eszköz segítségével a nyelv maga biztosítja, hogy az adott típus használója csak a specifikációban megadott tulajdonságokat használhassa ki.
- Ez a megszorítás lehetővé teszi a reprezentáció, illetve az implementáció megváltoztatását anélkül, hogy a változások a programban felfelé gyűrűznének.

Absztrakt típusok megvalósításának eszközei

A reprezentáció elrejtése

- Az adatrejtés támogatására a nyelvek gyakran az összetett típusok komponenseinek láthatóságát szintekre bontják, ezek meghatározzák, hogy pontosan kik férhetnek hozzá az adott komponenshez.
- Leggyakrabban három szintű:
  - 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

Adatrejtéshez új láthatósági szintek bevezetése

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

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, nem csak a közvetlen kódismétlések elkerülésére, hanem a magasabb szintű megoldási struktúrák többszörös megírásának elkerülésére.
- Ez nagyban javítja a kód olvashatóságát és karbantarthatóságát.

Absztrakt típusok megvalósításának eszközei

Generikus programsémák támogatása

- …

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.

A C/C++ nyelvekben 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.

```
 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;
}
```

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

Vagy: `friend`

```
…
#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

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...

Alprogramok

- Alprogramok paraméterezése

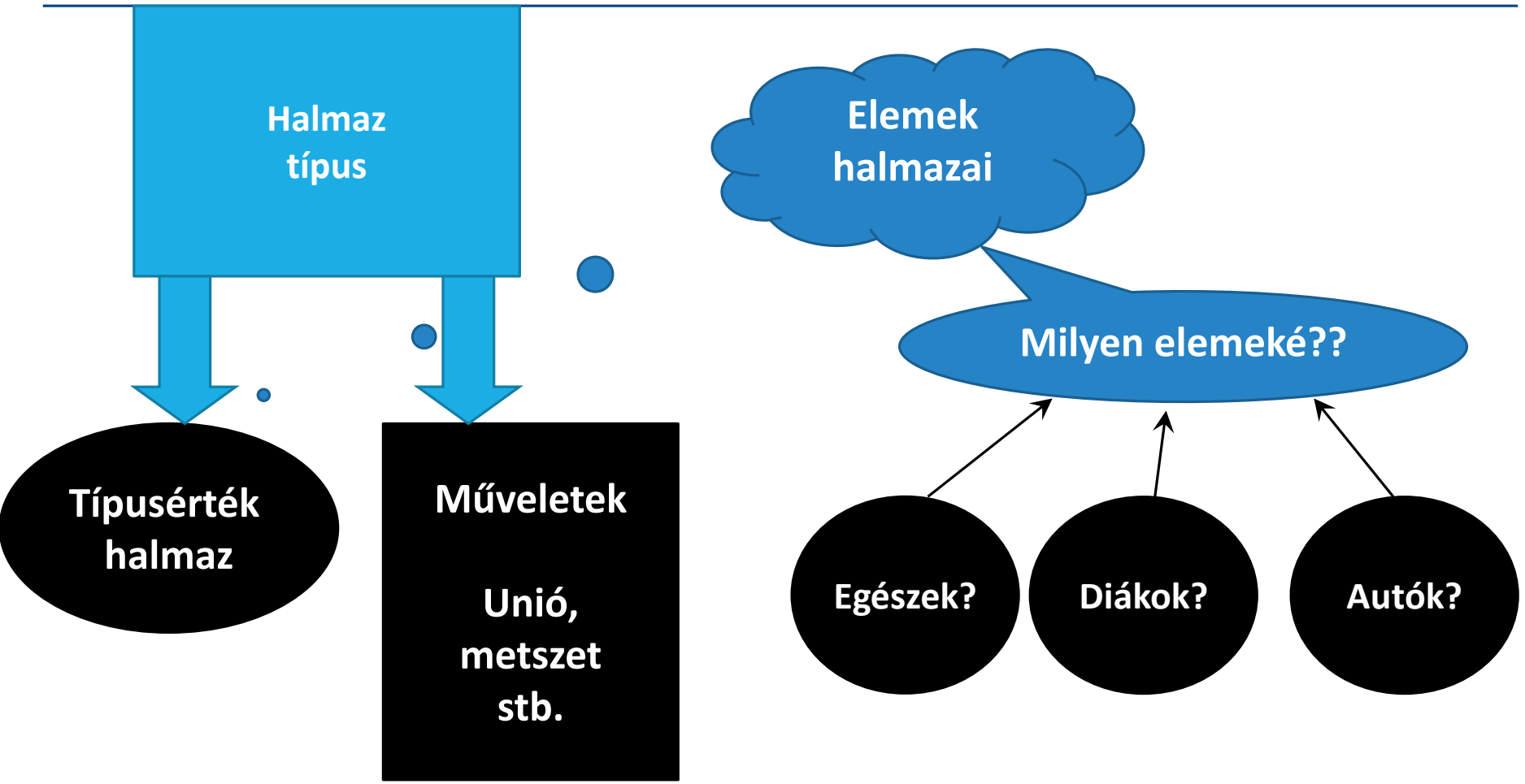
Típusok paraméterezése

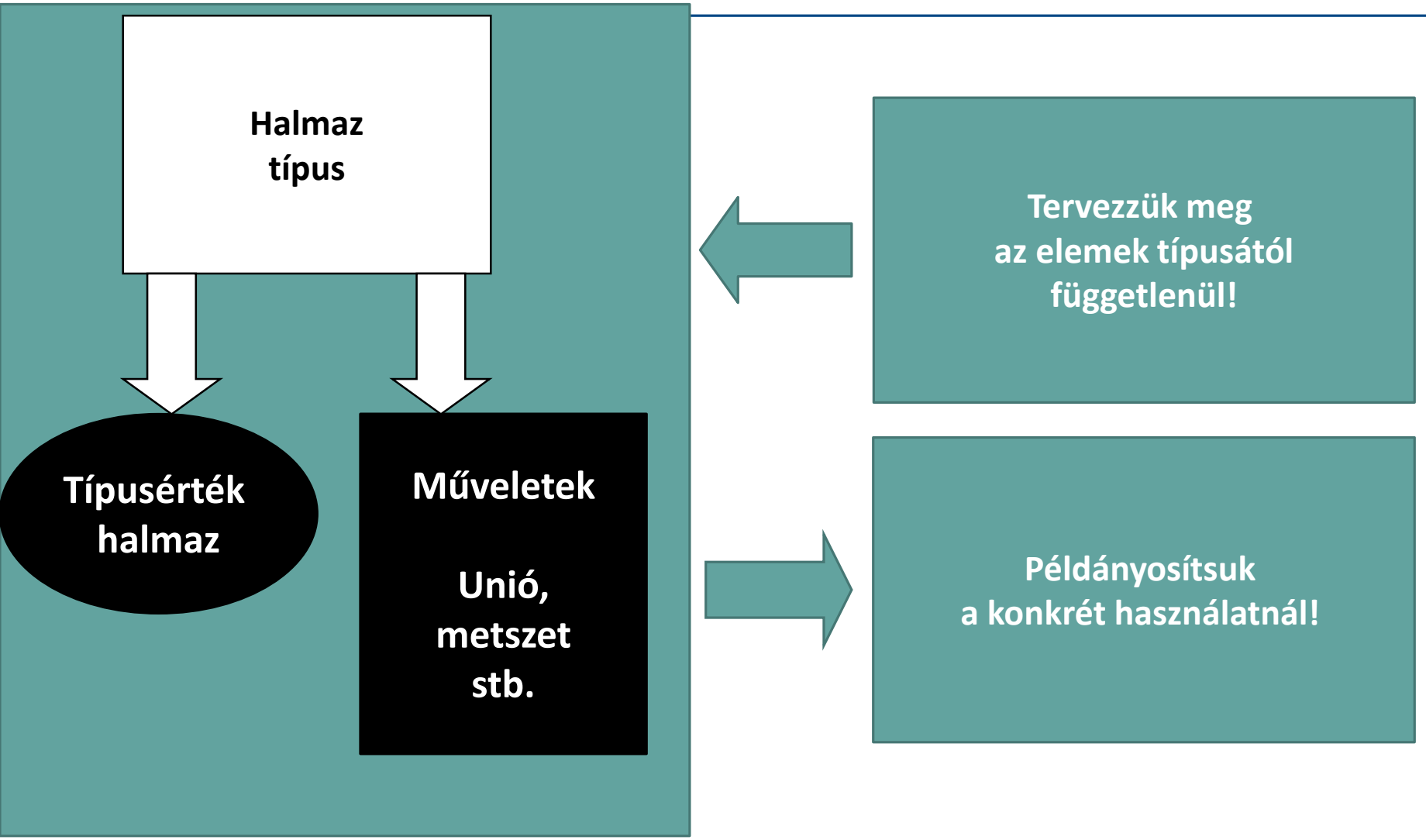
- Például az Ada-ban a diszkriminánsos rekordok, vagy a határozatlan indexhatárú tömb típusok

Alprogrammal történő paraméterezés

Típusokkal történő paraméterezés

- Programozási tételeink, alapvető adatszerkezeteink és adattípusaink is “absztrakt” fogalmakat használnak, azaz csupán 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

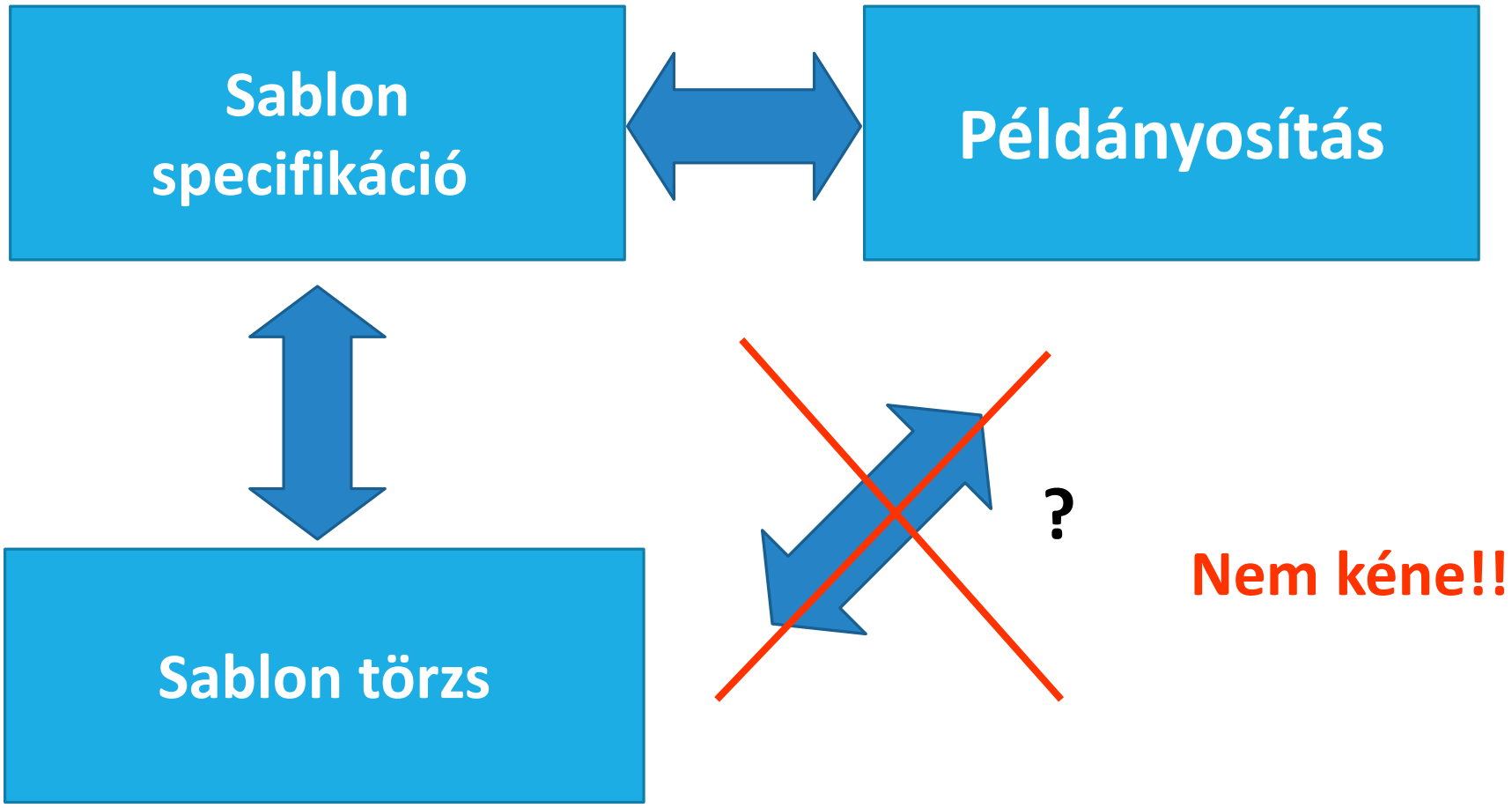
Fa

Keresőfa

Hash-tábla, stb.

Mi a különbség a példák között?

Példányosítás



Ezen igény kielégítésére többféle eszköz kínálkozik.

- A C nyelvben például nem áll rendelkezésre ilyen eszköz nyelvi szinten, ám hasonló hatások – korlátozott mértékben – elérhetők az előfeldolgozó rendszer használatával.

Az előfeldolgozó rendszer szerves része a C nyelv specifikációjának

- Mégsem tekinthető igazán nyelvi eszköznek, hiszen 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)
```

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

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

Az egyetlen hibája a megoldásnak, hogy előre – a nyelv alapvető osztályhierarchiájának kialakításakor – ismerni kellene az összes számításba jövő feltételkombinációt.

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.

Egy template kicsit több, mint az előző makróhelyettesítés, ahol a fordító a megadott aktuális típussal helyettesíti a hozzá tartozó formális paraméter minden előfordulását.

- Bizonyos minimális szintaxisellenőrzést ugyan 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 (például bizonyos műveletek) meglétét.

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 konverzió
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;
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++ - új szabvány

Tervezték a concept –ek lehetőségét

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

C++ concept –ek

```
template<class T> const T& min(const T &x, const T &y)
   { return y < x ? y : x; } // < értelmes T-re?
```

```
template<LessThanComparable T>
const T& min(const T &x, const T &y) { return y < x ? y : x;
   }
```

vagy

```
template<class T> const T& min(const T &x, const T &y)
   requires LessThanComparable<T> { return y < x ? y : x; }
// ez általánosabb, lehetne requires !LessThan..., ...
```

C++ concept –ek

```
auto concept LessThanComparable<class T>
{
  bool operator<(T, T);
}
```

az auto kulcsszó hatására nem kell megadni a concept_map-ban a teljesítés mikéntjét, ha az rendelkezik ilyen szignatúrájú operator< -bel

conceptek egymásba ágyazhatók

```
auto concept Comparable <class T>
{
  requires LessThanComparable<T>;
  requires GreaterThanComparable<T>;
  ...
}
```

C++ concept –ek

A concept_map segítségével megadhatjuk az elvárások teljesítésének módját:

```
concept_map LessThanComparable<MyType>

{

  bool operator<(const MyType& lhs,
                 const MyType& rhs)

  { return lhs.Less(rhs); }

};
```

C++ - új szabvány

A concepteket sajnos végül NEM vették bele!

- Majd a következő szabványba

Eiffel: generic

Erősen 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, így 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

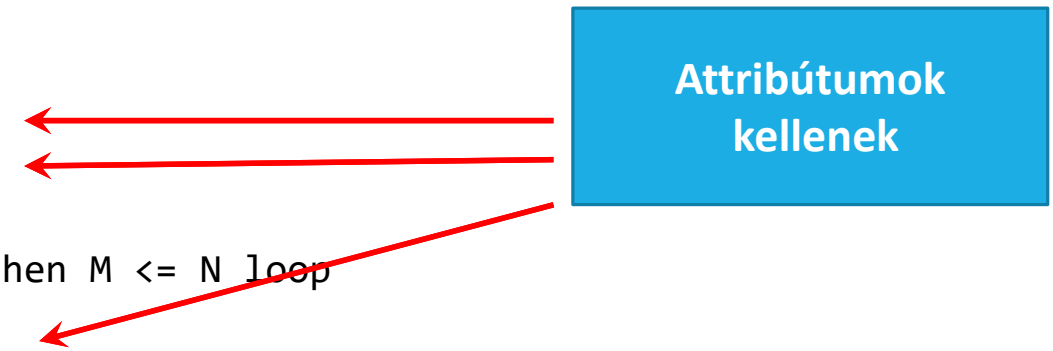
A sablonok paraméterezhetősége sokkal szélesebb körű és rugalmasabb a korábbiaknál.

- 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.

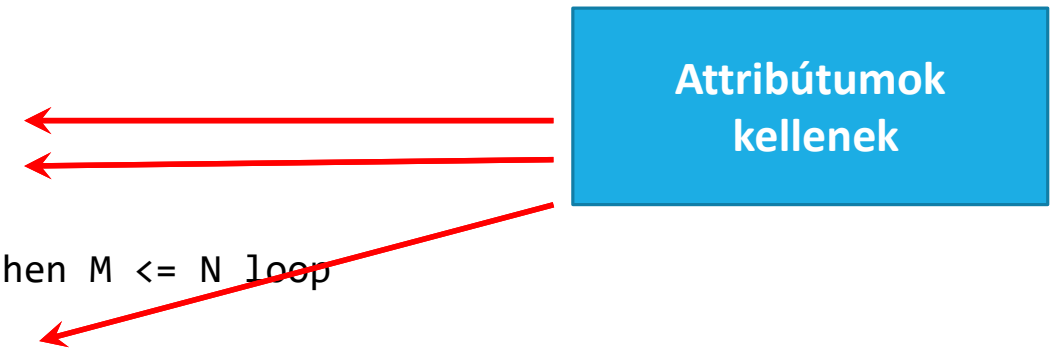
Ada

```
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;
```



Verem típus

```
generic
Max_Size : Integer;
type Elem_Type is private;

package  G_Stack_T is

   type Stack_Type is limited private;

   procedure Push(S:in out Stack_Type; Elem:in Elem_Type);

   procedure Pop(S:in out Stack_Type; Elem:out Elem_Type);

   function Is_Empty(S:in Stack_Type) return Boolean;

   function Is_Full(S:in Stack_Type) return Boolean;

   Empty, Full : exception;
```

```
private
   subtype Index is Integer
                              range 1..Max_Size+1;
   type Elements_Array is array (Index) of
                              Elem_Type;
   type Stack_Type is record
         Elements   : Elements_Array;
         First_Free : Index           := 1;
   end record;
end G_Stack_T;
```

```
package body G_Stack_T is
   procedure Push(S:in out Stack_Type;
                            Elem:in  Elem_Type) is
   begin
      if S.First_Free < Index'Last then
         S.Elements(S.First_Free):=Elem;
         S.First_Free := Index'Succ(S.First_Free);
      else
         raise Full;
      end if;
   end Push; ….
end G_Stack_T;
```

Példányosítás

```
with G_Stack_T, Text_Io; use Text_Io;
procedure Gstackdemo is
 package Intst is new G_Stack_T(15, Integer);
   use Intst;
   St : Stack_Type;
   Stel : Integer;
 begin
   Push(St,2);
   Pop(St, Stel);
```

- …

Java

Generic bevezetése

- Korábban:

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

- Most már lehet:
  - `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

java.util-ban még példa

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, Cloneable,
Serializable ….
```

java.lang-ban példa

```
public interface Comparable<T> …
```

Java

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

vagy egybe:

```
Box<Integer> 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();
                                    // no cast!
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, ellenőrzés fordításkor van

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(); ….
```

Típushelyettesítők

- Tételezzük fel, hogy

```
public class Gen<A> {
  public Gen(A aa) { a = aa; }
  public A getElem(){return a;};
  public void setElem(A aa){ a=aa;  }
  protected A a;
}
```

- 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?

Típushelyettesítők

- `Gen<A> 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 is lehet:
  - `Gen<?> g1 = f( new Gen<Integer>(1), new Gen<Double>(2.0));`

Típushelyettesítők

Még jobb, ha már a definíciójába tesszük:

```
static Gen<? extends Number> max(
            Gen<? extends Number> ga,
            Gen<? extends Number> gb) {
     Number n1 = ga.getValue();   // ok, legalább Number
     Number n2 = gb.getValue();   // ok, legalább Number
    // most már akár ezt is írhatjuk:
    if ( n1.doubleValue() < n2.doubleValue() )
      return ga;
    else
      return gb;}
```

super lehetősége

new problémája

C#

```
public class LinkedList<K,T> where K : Icomparable  {
   T Find(K key)   {
      Node<K,T> current = m_Head;
      while(current.NextNode != null) {
         if(current.Key.CompareTo(key) == 0)
            break;
         else
            current = current.NextNode;
      }
      return current.Item;
   }
   //Rest of the implementation
}
```

C#

```
public class Employee {

  private string name;

  private int id;

  public Employee(string s, int i) {

    name = s; id = i;

   }

  public string Name {
   get { return name; }
   set { name = value; }
  }

  public int ID {
   get { return id; }
   set { id = value; }
  }

}
```

C#

```
public class GenericList<T> where T : Employee {

 …

 public T FindFirstOccurrence(string s) {
  Node current = head;
  T t = null;
    while (current != null) {
       //a megszorítás elérhetővé teszi a Name propertyt:

      if (current.Data.Name == s) {
        t = current.Data; break;
      } else {
        current = current.Next;
      }
    }
    return t;
  }

}
```

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?

- Így pl. ha rendezett listák egy sablonját szeretnénk, előírható-e, hogy a lista majdani aktuális elemeinek típusára szeretnénk, hogy legyen egy „kisebb" reláció értelmezve?

Egymásba ágyazható ez a konstrukció?

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