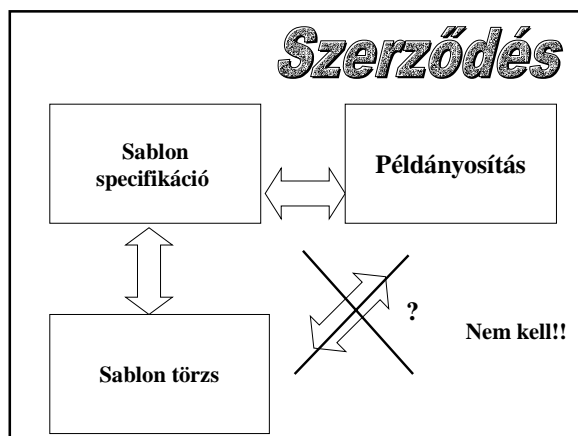
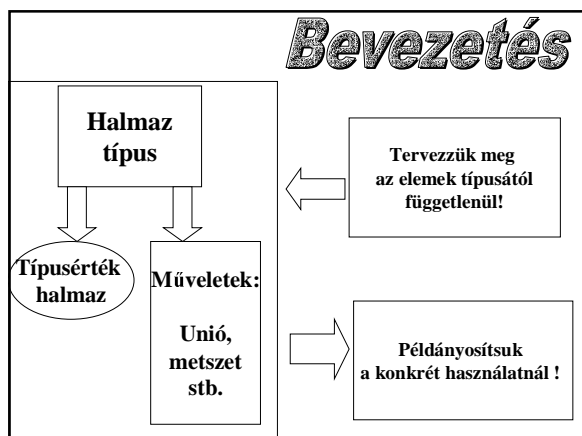
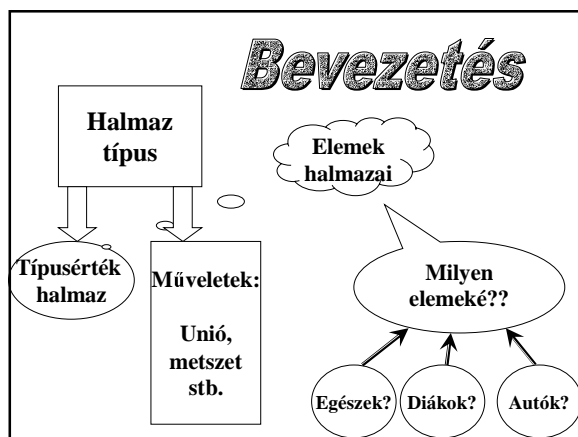


Sablonok

- 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 (pl. az Ada-ban a diszkriminációs rekordok, vagy a határozatlan indexhatáru tömb típusok)
- Alprogrammal történő paraméterezés
- Típusokkal történő paraméterezés
programozási tételeink, alapvető adatszerkezetek 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.



- 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 ha úgy tetszik, 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. Pl.:

```
/* 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
....
```

Alphard:

```
form simpleset(maxsize: integer, thin: form <=,>)
beginform
specification
requires maxsize >=0;
...
representation
...
implementation
...
endform
```

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)
```

- Objektorientált programozási nyelvekben a hatás elvileg elérhető lenne kizárólag az öröklődés és a polimorfizmus használatával, hiszen a szükséges közös tulajdonságokat össze lehetne fogni egyetlen absztrakt osztályba – vagy interfészbe –, amely aztán közös öse 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 létrehozuk 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( int& x, int& y) {
int temp = x; // !! Hogy határozzuk meg x
                                     típusát?
```

```
x = y;
y = temp; }
```

- A **C++** nyelvben a típussal paraméterezés lehetőségét a **template**-ek biztosítják.
- Egy **template** alig több, mint valamiféle 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.

- 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.

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

```
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
```

```
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); // ->
    „ambiguous, no standard conversion”
```

```
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); // ok, de..
z == 3.0 // ??
```

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
```

```
#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;
};
```

```
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';
    return 0;
};
```

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 osztály műveletei lehetnek, így a használat helyessége a generic megírásokor ellenőrizhető.
- Hátránya: a szükséges közös tulajdonságokat jóval előre tudni kell, hogy a megfelelő közösős 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.

- 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

- Ada95: 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 ősléte.
- 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.

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;
```

Attribútumok
kellenek

```

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;

```

Attribútumok
kellenek

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 5.0: generic bevezetése

Eddig:

```

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 5.0:

Ehhez a java.util-ban pl.:

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

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

■ java.util-ban még pl.:

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

■ java.lang-ban pl. :

```
public interface Comparable<T> ...
```

- Figyeljük meg, hogy míg a C++ esetén a típussal való paraméterezésnél a 'típus' fogalmat inkább a programozási nyelv szemszögéből használjuk, ahol igazából csak a típusérték-halmaz (vagyis a reprezentáló elemiérték-sorozatok halmaza) számít, addig az Ada és az Eiffel már inkább a programozási módszertan szemszögéből közelít, azaz a típust típusértékek és a hozzájuk tartozó műveleteket egységeként kezeli.

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?

- 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 (statikus) vagy futási időben (dinamikus) történik?