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

generikus szerkezetek, template

Szeberényi Imre
BME IIT
<szebi@iit.bme.hu>

Hol tartunk?

- C → C++ javítások
- OBJEKTUM: konkrét adat és a rajta végezhető műveletek összessége
- OO paradigmák
  - egységbezárás (encapsulation), többarcúság (polymorphism) , példányosítás (instantiation), öröklés (inheritance), generikus szerkezetek
- OO dekompozíció, tervezés
- A C++ csupán eszköz
- Elveket próbálunk elsajátítani
- Újrafelhasználhatóság

Hol tartunk? /2

- objektum megvalósítása
  - osztály (egységbe zár, és elszigetel),
  - konstruktor, destruktor, tagfüggvények
  - inicializáló lista (tartalmazott obj. inicializálása)
  - operátor átdefiniálás (függvény átdefiniálás)
  - barát és konstans tagfüggvények
  - dinamikus adat külön kezelést igényel
  - öröklés és annak megvalósítása
  - védelem enyhítése
  - virtuális függvények és osztályok
  - absztrakt osztályok

Mi az objektummodell haszna?

- A valóságos viselkedést írjuk le
- → Könnyebb az analógia megteremtése
- Láttuk a példát (dig. áramkör modellezése)
  - Digitális jel: üzenet objektum
  - Áramköri elemek: objektumok
  - Objektumok a valós jelterjedésnek megfelelően egymáshoz kapcsolódnak. (üzennek egymásnak)
- Könnyen módosítható, újrafelhasználható
- Funkcionális dekompozícióval is így lenne?

Ismétlés (alakzat)

```
class Alakzat {
protected:
  Pont p0;      ///< alakzat origója
  Szin sz;      ///< alakzat színe
public:
  Alakzat(const Pont& p0, const Szin& sz)
      :p0(p0), sz(sz) {}
  const Pont& getp0() const { return p0; }
  virtual void rajzol() const = 0;
  void mozgat(const Pont& d);
  virtual ~Alakzat() {}
};
```

Ismétlés (alakzat) /2

```
class Poligon : public Alakzat {
  Pont *pontok;
  unsigned int np;
  Poligon(const Poligon&);
  Poligon& operator=(const Poligon&);
public:
  Poligon(const Pont& p0, const Szin sz)
    :Alakzat(p0, sz), np(1)
                { pontok = new Pont[np-1]; }
  int getnp() const { return np; }
  Pont getcsp(unsigned int i) const;
  void add(const Pont& p);
  void rajzol();
  ~Poligon() { delete[] pontok; }
};
```

Ismétlés (alakzat) /3

```
Pont Poligon::getcsp(unsigned int i) const {
    if (i >= np) throw "Poligon: nincs ilyen";
    if (i == 0) return p0;
    return pontok[i-1] + p0;
}
void Poligon::add(const Pont& p) {
    Pont *tp = new Pont[np];
    for (unsigned int i = 0; i < np-1; i++)
        tp[i] = pontok[i];
    delete[] pontok;
    pontok = tp;
    pontok[np-1] = p - p0;
    ++np;
}
```

Lehet-e tovább általánosítani?

- Általánosíthatók-e az adatszerkezetek?
  Már a komplexes első példán is észrevettük, hogy bizonyos adatszerkezetek (pl. tárolók) viselkedése független a tárolt adattól.
  Lehet pl. adatfüggetlen tömböt vagy listát csinálni?
- Általánosíthatók-e az algoritmusok?
  Lehet pl. adatfüggetlen rendezést csinálni ?

elemzés: Din. tömb

- Tároljunk T-ket egy tömbben!

Műveletek:

  - Létrehozás/megszüntetés
  - Indexelés
  - Méretet a létrehozáskor (példányosításkor) adjuk
  - Egyszerűség kedvéért nem másolható, nem értékadható és nem ellenőriz indexhatárt!

TArray megvalósítás

```
class TArray {
    T *tp; // elemek tömbjére mutató pointer
    unsigned int n;   // tömb mérete
    TArray(const TArray&); // másoló konstr. tiltása
    TArray& operator=(const TArray&); // tiltás
public:
    TArray(int n=5) :n(n) { tp = new T[n]; }
    T& operator[](unsigned int);
    const T& operator[](unsigned int) const; // most nem impl.
    ~TArray() { delete[] tp; }
};
T& TArray::operator[](unsigned int i) { return tp[i]; }
```

privát, így nem érhető el

Mit kell változtatni?

- Minden T-t át kell írni a kívánt (pl, int, double, Komplex stb.) típusra.
- Neveket le kell cserélni (névelem csere)
- Más különbség láthatóan nincs.

- (Meg kellene valósítani tisztességesen, hogy használható legyen, de most nem ez a lényeg)

Lehet-e általánosítani?

Típusokat és neveket le kell cserélni -->

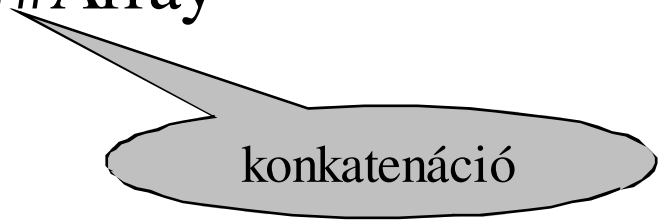
Generikus adatszerkezetek:

- Olyan osztályok, melyekben az adattagok és a tagfüggvények típusa fordítási időben szabadon paraméterezhető.
- Megvalósítás:
  - preprocesszorral
    - define + névkonkatenáció ##
  - nyelvi elemként (template)

Preprocesszoral

Típus és névelem csere makrókkal:

- Névelem csere:
  - #define Array(T)
- Osztály deklarációk:
  - #declare_Array(T) makro
- Külső tagfüggvény definíciók:
  - #implement_Array(T) makro



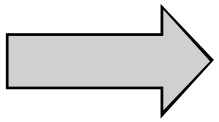
Hogyan működik?

névelem csere

```
class TArray {
    T *tp;
    unsigned int n;
public:
    TArray(int n=5)
    :n(n) { tp = new T[n]; }
...
```

```
#define Array(T)      T##Array
#define declare_Array(T)          \
class Array(T) {                  \
    T *tp;                        \
    unsigned int n;               \
public:                           \
    Array(T)(int n=5)             \
    :n(n) { tp = new T[n]; }      \
...
```

```
declare_Array(int)
```



```
class intArray {
    int *tp;
    unsigned int n;
public:
    intArray(int n=5)
    :n(n) { tp = new int[n]; }
...
```

Array(T)

```
#define Array(T)        T##Array
```

```
#define declare_Array(T)                                    \
class Array(T) {                                            \
    T *tp;                                                  \
    unsigned int n;                                         \
    Array(T)(const Array(T)&);                              \
    Array(T)& operator=(const Array(T)&);                   \
public:                                                     \
    Array(T)(int n=5) :n(n) { tp = new T[n]; }              \
    T& operator[](unsigned int);                            \
    ~Array(T)() { delete[] tp; }                            \
};
```

```
#define implement_Array(T)                                  \
     T& Array(T)::operator[](unisgned int i) { return tp[i]; }
```

Használata

```
#include "gen_array_m.hpp" //declare+implement makró
class Valami { . . . };          // valamilyen osztály

declare_Array(int)               // intArray deklarációja
implement_Array(int)             // intArray def. csak 1-szer !
declare_Array(Valami)            // ValamiArray deklarációja
implement_Array(Valami)          // ValamiArray def.  1-szer !

   Array(int) a1, a2, a3;
   Array(Valami) v1, v2, v3;

   a2[0] = 5;
   v2[1] = Valami;
```

Minden rendben ?

- **Tároljunk stringekre mutató pointereket:**

  declare_Array(char *) implement_Array(char *)

- **Mi lesz a makrókból?**

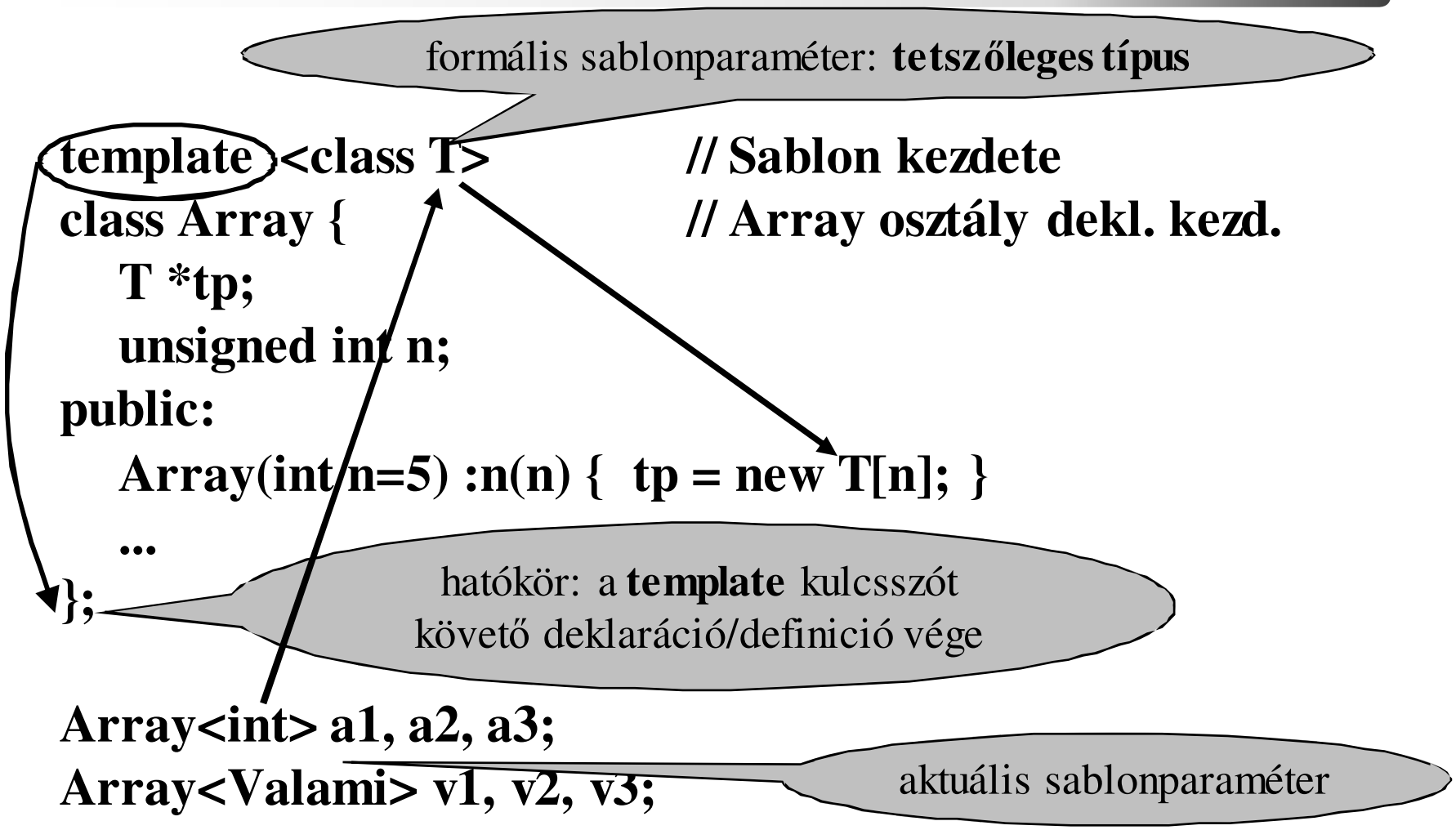
  pl: #define Array(T) T##Array

  typedef-fel talán megoldható lenne

- **Más, ennél a példánál nem jelentkező problémák is adódnak az egyszerű szöveghelyettesítésből, ezért jobb lenne nyelvi elemmel.**

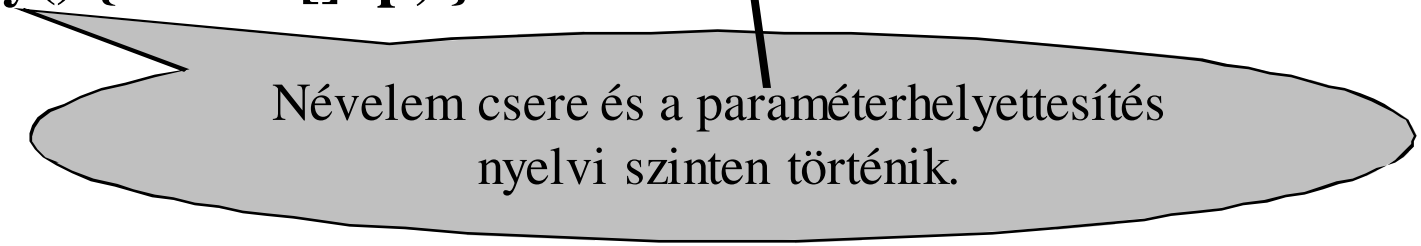
- **Megoldás: template**

Megoldás: template – nyelvi elem

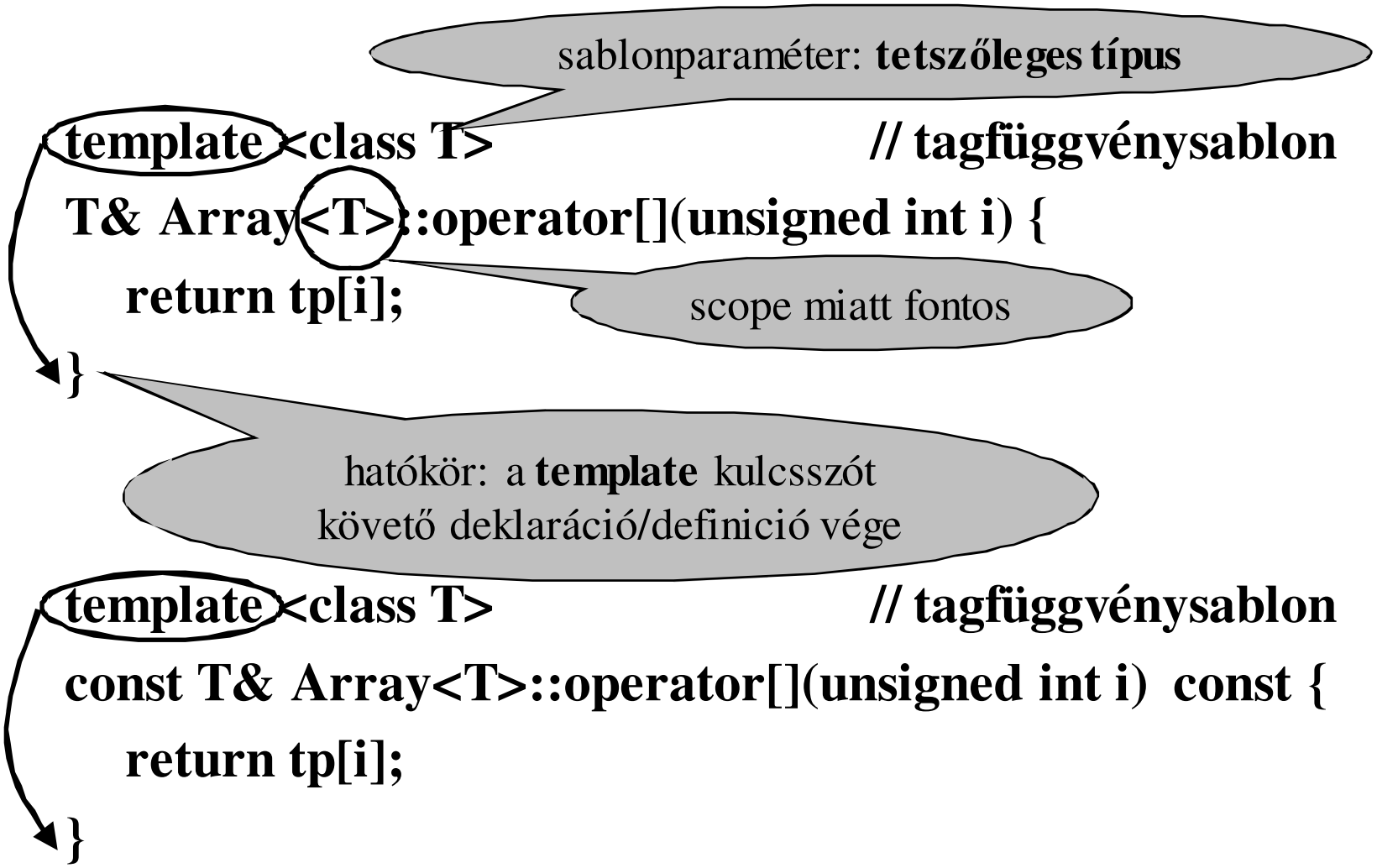


Array osztály sablonja

```
template <class T>          // osztálysablon
class Array {
    T *tp;                  // elemek tömbjére mutató pointer
    unsigned int n;         // tömb mérete
    Array(const Array&); // másoló konstr. tiltása
    Array& operator=(const Array&); // tiltás
public:
    Array(int n=5) :n(n) { tp = new T[n]; }
    T& operator[](unsigned int);
    const T& operator[](unsigned int) const;
    ~Array() { delete[] tp; }
};
```



Tagfüggvények sablonja

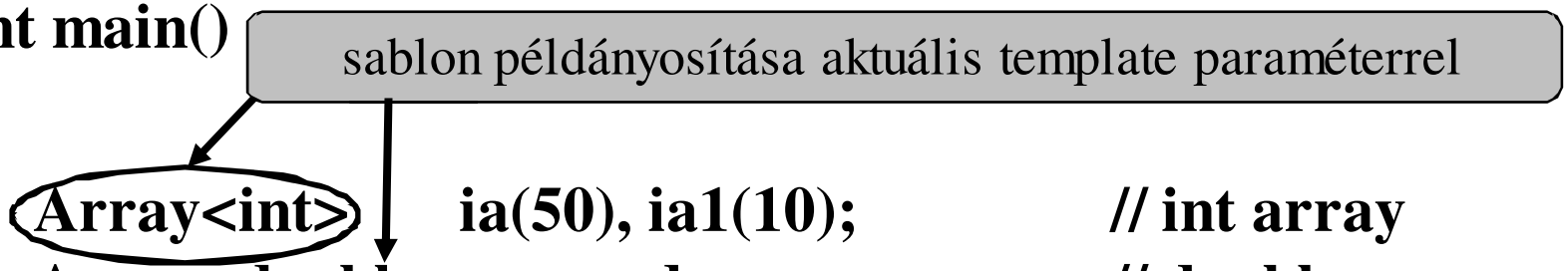


Sablonok használata (példányosítás)

```
#include "generic_array.hpp" // sablonok

int main()
{
   Array<int>          ia(50), ia1(10);        // int array
   Array<double>       da;                     // double array
   Array<const char*>  ca;                     // const char* array

   ia[12] = 4;
   da[2] = 4.54;
   ca[2] = "Hello Template";
   return 0;
}
```



Array osztály másként

```
template <class T, unsigned int s>   // osztálysablon
class Array {
    T t[s];                          // elemek tömbje
public:
    T& operator[](unsigned int i) {
        if (i >= s) throw "Indexelési hiba";
        return t[i];
       }
    const T& operator[](unsigned int i)  const;
};
```

Konst. obj-hoz

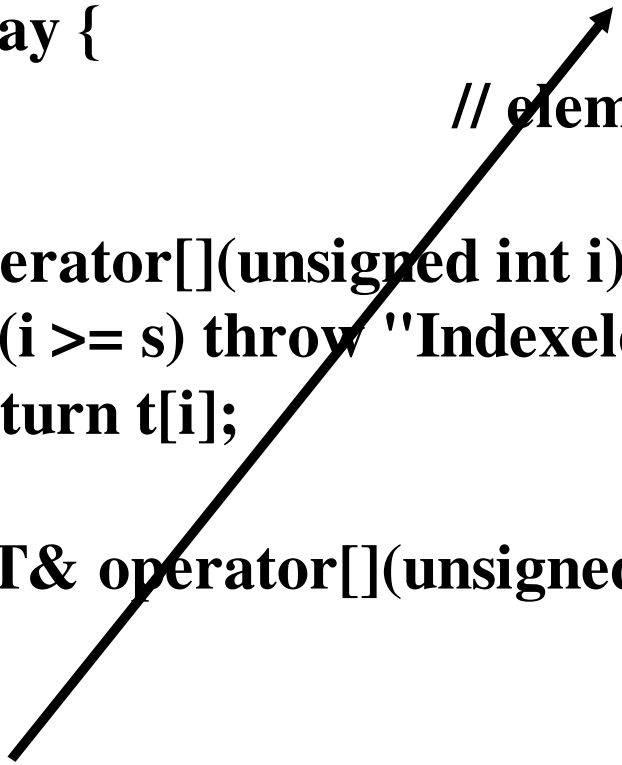
Többször példányosodik! Növeli a kódot, ugyanakkor egyszerűsödött az osztály.

```
Array<int, 10> a10; Array<int, 30> a30;
```

deafault paraméter is lehet

```
template <class T, unsigned int s = 10>   // osztálysablon
class Array {
    T t[s];                          // elemek tömbje
public:
    T& operator[](unsigned int i) {
          if (i >= s) throw "Indexelési hiba";
          return t[i];
         }
    const T& operator[](unsigned int i)  const;
};

Array<int> a10; Array<int, 30> a30;
```



Lehet-e tovább általánosítani?

- Általánosíthatók-e az adatszerkezetek? → Sablon
- Általánosíthatók-e a függvények? → Sablon

```
template <typename T>
inline T Max(T a, T b) {
    return a > b ? a : b;
};
```

std::cout << Max<int>(6, 8);

std::cout << Max (3.7, 8.9);

A sablonparaméterek többnyire **levezethetők** a függvényparaméterekből.

Függvénysablon

- Függvények, algoritmusok általánosítási eszköze.
- Hatékony, paraméterezhető, újrafelhasználható, általános.

```
template <class T>
void rendez (T a[], int n) {
    for (int i = 1; i < n; i++) {
        T tmp = a[i]; int j = i-1;
        while (j >= 0 && a[j] > tmp) {
            a[j+1] = a[j]; j--;
        }
        a[j+1] = tmp;
    }
}
.....
int t[] = { 4, 8, -2, 88, 33, 1, 4, -1 };
rendez<int>(t, 8);
```

Kérdések

1. Referecia paraméterrel hatékonyabb-e?

```
template <typename T>
const T& Max(const T& a, const T& b) {
    return a > b ? a : b;
};
```

2. Működik-e sztringel?

```
std::cout << Max("Hello", "Adam");
```

Címeket hasonlítunk össze sztringek helyett!

Megoldás: specializáció

```
template <typename T>
const T& Max(const T& a, const T& b) {
    return a > b ? a : b;
};

// Specilaizáció T::= const char* esetre
template <>
const char* Max(const char* a,const char* b){
    return strcmp(a,b) > 0 ? a : b;
};


std::cout << Max("Hello", "Adam");
```

Specializáció

Függvények különböző változatai: átdefiniálás

Sablonok esetében egy újabb eszközünk is van: specializáció. Egy sablonnal megadott osztály, vagy függvény adott változatát átdefiniálhatjuk. Ilyenkor nem a sablonban megadott módon fog példányosodni.

```
template <class T>
struct V {
   T a1;
   V() :a1(T()) { }
};
```

```
template <>
struct V<double> {
   double a1;
   V() :a1(3.14) { }
};
```

```
V<int>v1;
```



```
a1 = 0;
```

```
V<double>v2;
```



```
a1 = 3.14;
```

Mi is a sablon ?

- Nyelvi elem az általánosításhoz.
- Gyártási forma.
- A sablonparaméterektől függően példányosodik:
  - osztály vagy függvény jön belőle létre.
- Paraméter: típus, konstans, függvény, sablon
- Default paramétere is lehet.
- A példányok specializálhatók, melyek eltérhetnek az eredeti sablontól.
- A példányosítás helyének és a sablonnak egy fordítási egységben kell lennie (.hpp).

A feldolgozás fordítási idejű

```
template <int N>
struct fakt {
   enum  { fval =  N * fakt<N-1>::fval };
};
```

Specializálás

```
template <>
struct fakt<0> {
   enum  { fval =  1 };
};
```

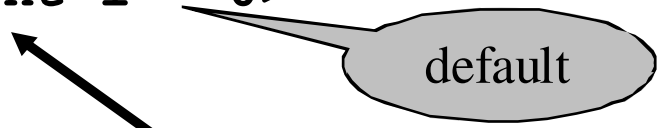
std::cout << fakt<5>::fval << std::endl;
Fordítási időben 4 példány (3,2,1,0) keletkezik

Nem akarunk így programozni! Csak szemléltetés.

Template paraméter

típus, konstans, függvény, sablon

```
template <class T1, typename T2, int i = 0>
struct V {
    T1 a1; T2 a2; int x;
    V() { x = i;}
};
```




```
V<int, char, 4> v1;
V<double, int> v2;
```

```
template <class T> struct miez { int a0; };
template <class K, template <class T> class C = miez>
struct S {
    C<K> a1;
};
S<int> vau; vau.a1.a0 = 10;
```

Részleges és teljes specializáció

```
template <class R, class S, class T>
struct A { … };
```

részleges specializálás

```
template <class S, class T>
struct A<char, S, T> { … };
```

teljes specializálás

```
template <>
struct A<char, char, char>{  … };
```

Függvénysablonok paraméterei

A sablonparaméterek általában **levezethetők** a függvényparaméterekből. Pl:

```
template<class T> void csere(T& p1, T& p2) {
   T tmp = p1; p1 = p2; p2 = tmp;
}
int x1 = 1, x2 = 2;                    csere(x1, x2);
```

Ha nem, akkor meg kell adni. Pl:

```
template<class T, int n> void fv(T t1[n], T t2[n]) {
   for (int i = n; i >= 0; i--)
      t1[i] = t2[i];
}
int it1[10], it2[10];              fv<int, 10>(it1, it2);
```

Algoritmus módosítása

- Előfordulhat, hogy egy algoritmus (pl. rendezés) működösét módosítani akarjuk egy függvénnyel (predikátum).
- Sablonparaméterként egy eljárásmódot (függvényt) is átadhatunk, ami lehet osztályban, vagy önállóan.
- Példa: Írjunk egy általános kiválasztó algoritmust, ami képes kiválasztani a legkisebb, legnagyobb, leg... elemet.

Kiválasztó algoritmus sablonnal

```
template<class T, int n, class S>
T kival(T t[n]) {
    T tmp = t[0];
    for (int i = 1; i < n; i++)
        if (S::select(t[i], tmp)) tmp = t[i];
    return tmp;
}
```

Feltételezzük, hogy van egy osztályunk, aminek van egy alkalmas select tagfüggvénye

select fv. is lehet sablon

```
template<class T>
struct Min { // szokásos min. kereséshez
   static bool select(T a, T b) { return a < b; }
};
template<class T>
struct Max { // szokásos max. keresésez
   static bool select(T a, T b) { return a > b; }
};
```

```
struct intMinAbs { // szokásostól eltérő kiválasztó
  static bool select(int a, int b) { return abs(a) < abs(b);}
};
```

Kiválasztó algoritmus használata

```
int main()  {

  int it[9] = {-5, -4, -3, -2, -1, 0, 1, 2, 3 };
  double dt[5] =  { .0,  .1,  .2,  .3,  4.4 };
  cout << kival<double, 5, Max<double> >(dt); // max
  cout << kival<int, 9, Min<int> >(it); // mimimum
  cout << kival<int, 9, intMinAbs>(it); // eltérő kiv. fv.
  return(0);
}
```

szóköz kell,
különben **>>**-nek értené!

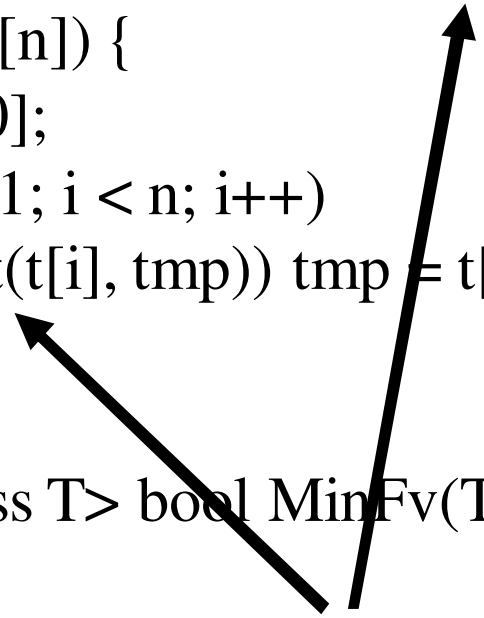
A select statikus tagfüggvény, ezért
nem kell külön példányosítani.

Tagfüggvény vagy önálló fv.

Előzőekben osztálysablonba építettük a predikátum függvényt. Önálló függvény is lehet predikátum:

```
template<class T, int n, bool select(T, T)>
T kivalFv(T t[n]) {
    T tmp = t[0];
    for (int i = 1; i < n; i++)
        if (select(t[i], tmp)) tmp = t[i];
    return tmp;
}
template<class T> bool MinFv(T a, T b) {  return a < b; }

cout << kivalFv<int, 9, MinFv>(it);
```



Függvényparaméter és fv. objektum

Leggyakrabban egy osztály függvényhívó operátorát használjuk (ún. függvény obj). Ez azonban nem lehet static:

```
template<class T,  class F>
T kivalaszt(T t[], int n, F Func) {
   T tmp = t[0];
   for (int i = 1; i < n; i++) if (Func(t[i], tmp)) tmp = t[i];
   return tmp;
}
template<class T> struct MinObj {
   bool operator()(T& a, T& b) { return a < b; }
};
cout << kivalaszt<int, MinObj<int> >(it, 9, MinObj<int>() );
```

létre kell hozni

Függvény objektum: funktor

```
template<class T>
struct HasonlitObj {
  bool operator()(T a, T b) {
     return a > b;
  }
};
```

Objektum fv. operátora.
Példány kell.

```
int t[6] = {1, 2, -8, 0, 12, 3};
kivalaszt(t, 6, hasonlitObj<int>());
```

Predikátum

- Logikai függvények, vagy függvény-objektumok, amelyek befolyásolják az algoritmus működését
- Predikátum megadása
  - Sablonparaméterként:

```
template<class T, int n, bool select(T, T)>
T keresf(T t[n]);
```

  - Függvényparaméterként:

```
template<class T,  class F>
T ker(T t[], int n, F Func);
```

Gyakoribb, rugalmasabb megadási mód

Összefoglalás

- A C-ben megtanult preprocesszor trükkökkel általánosíthatók az osztályok
- Nem biztonságos, és nem ad mindenre megolást.
- → Nyelvi elem bevezetése: template
- A preprocesszoros trükköt csak a működés jobb megértéséhez néztük meg, ma már nem illik használni.

Összefoglalás /2

- Generikus osztályokkal tovább általánosíthatjuk az adatszerkezetekről alkotott képet:
  - Típust paraméterként adhatunk meg.
  - A generikus osztály később a típusnak megfelelően példányosítható.
  - A specializáció során a sablontól eltérő példány hozható létre
  - Specializáció lehet részleges, vagy teljes

Összefoglalás /3

- Generikus függvényekkel tovább általánosíthatjuk az algoritmusokról alkotott képet:
  - Típust paraméterként adhatunk meg.
  - A generikus függvény később a típusnak megfelelően példányosítható.
  - A függvényparaméterekből a konkrét sablonpéldány
    - levezethető, ha nem, akkor
    - explicit módon kell megadni
  - Függvénysablon átdefiniálható

Összefoglalás /4

- Predikátumok segítségével megváltoztatható egy algoritmus működése
- Ez lehetővé teszi olyan generikus algoritmusok írását, mely specializációval testre szabható.
- Ügyetlen template használat feleslegesen megnövelheti a kódot (pl. széles skálán változó paramétert is template paraméterként adunk át.)