

Programozás alapjai II.

(7. ea) C++

generikus szerkezetek, template újból

Szeberényi Imre, Somogyi Péter

BME IIT

<szebi@iit.bme.hu>



MŰEGYETEM 1782

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. és ősz osztály inicializálása)
 - függvény túlterhelés és felüldefiniálás (overload, override)
 - barát és konstans tagfüggvények
 - dinamikus adat (erőforrás) külön kezelést igényel
 - öröklés és annak megvalósítása
 - védelem enyhítése
 - virtuális függvények, 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) {}  
    Pont getp0() const { return p0; }  
    Szin getsz() const { return sz; }  
    virtual void rajzol() const = 0;  
    void mozgat(const Pont& d);  
    virtual ~Alakzat() {}  
};
```

Tartalmazott objektumok

Adattagok inicializálása

Virtuális destruktork

Virtuális rajzol.
Leszármazottban
valósul meg.

Ismétlés (alakzat) /2

```
class Poligon : public Alakzat {
    size_t np;
    Pont *pontok;
    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(size_t i) const;
    void add(const Pont& p);
    void rajzol();
    ~Poligon() { delete[] pontok; }
};
```

Ős inicializálása

Ne legyen
használható

Dinamikus területet

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 ?

KomplexTar (emlékeztető)

```
class KomplexTar {
    static const size_t nov = 3; // növekmény értéke
    Komplex *t;                // pointer a dinamikusan foglalt adatra
    size_t db;                 // elemek száma
    size_t kap;                // tömb kapacitása
public:
    KomplexTar(int m = 0) :db(m), kap(m+nov) {
        t = new Komplex[kap]; }
    KomplexTar(const KomplexTar&);
    unsigned int capacity() const { return kap; }
    unsigned int size() const { return db; }
    Komplex& operator[](unsigned int i);
    const Komplex& operator[](unsigned int i) const ;
    KomplexTar& operator=(const KomplexTar&);
```

...

Elemzés: Din. tömb

- Tároljunk **T**-ket egy tömbben!

Műveletek:

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

TArray megvalósítás

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

privát, így
nem érhető el

T helyett legyen int! Mi változik

- Minden **T**-t át kell írni int-re, azaz név elem csere kell:

```
class intArray {  
    size_t n;  
    int *tp;  
    intArray(const intArray&);  
    ....
```

- Más különbség láthatóan nincs.

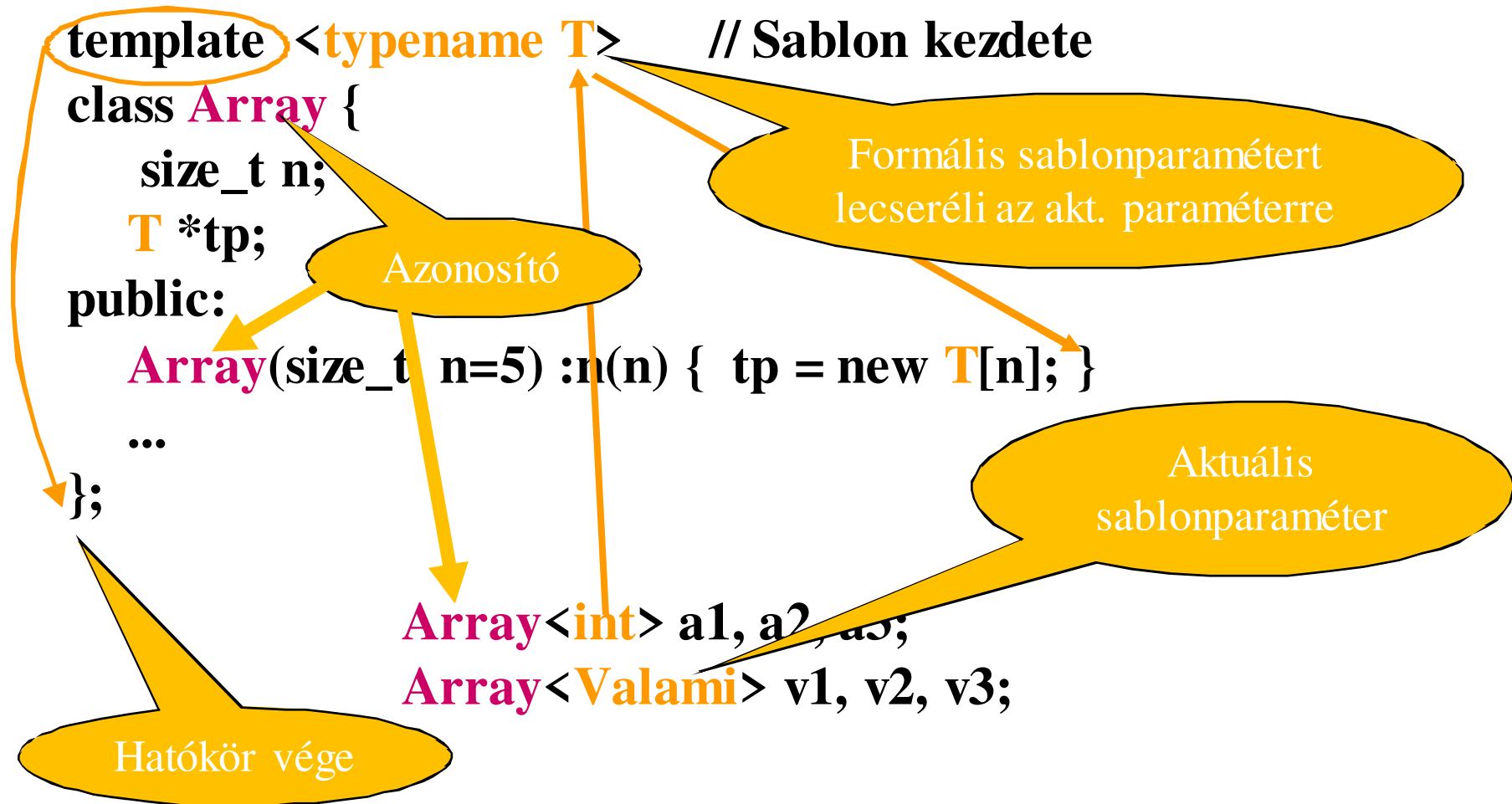
Lehet-e általánosítani?

Típusokat és neveket le kell cserélni -->

Generikus adatszerkezetek:

- Olyan osztályok, melyekben az adattagok és a tagfüggvények típusa **fordítási időben szabadon paraméterezhető**.
- Megvalósítás:
 - preprocesszorral
 - nem mindig lehetséges, nem típusbiztos
 - nyelvi elemként: **template**
 - Függvényeknél már használtunk. Miért ne lehetne osztályokra is?

Osztálysablon



Array osztály sablonja

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

Sablonparamétertől függő nevet generál
(név elem csere)

Array tagfüggvényeinek sablonja

```
template <typename T> // tagfüggvénytípus sablon  
T& Array<T>::operator[](size_t i) {  
    return tp[i];  
}
```

A scope-hoz a név a paraméterből generálódik

```
template <class T> // tagfüggvénytípus sablon  
const T& Array<T>::operator[](size_t i) const {  
    return tp[i];  
}
```

Sablonok használata (példányosítás)

```
#include "gen_array1.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;  
}
```

sablon példányosítása aktuális template paraméterrel

Array osztály másként

```
template <class T, size_t s> // osztálysablon
class fixArray {
    T t[s]; // elemek tömbje
    void chk (size_t i) const { if (i >= s) throw "Index hiba"; }
public:
    T& operator[](size_t i) {
        chk(i); return t[i]; }
    const T& operator[](size_t i) const {
        chk(i); return t[i]; }
};
```

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

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

default sabl. paraméter is lehet

```
template <class T, size_t s = 10 > // osztálysablon
class fixArray {
    T t[s]; // elemek tömbje
    void chk (size_t i) const { if (i >= s) throw "Index hiba"; }
public:
    T& operator[](size_t i) {
        chk(i); return t[i]; }
    const T& operator[](size_t i) const {
        chk(i); return t[i]; }
};

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

Lehet-e tovább általánosítani?

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

// max példa (ism.)

```
template <typename T>
```

```
T max(T a, T b) {
```

```
    return a < b ? b : a;
```

```
}
```

```
cout << max(40, 50);
```


```
cout << max("alma", "korte"); // "alma" → const char *
```

Sablón specializáció (ism.)

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

// Specializáció $T ::= \text{const char}^*$ esetre

```
template <>
const char* Max(const char* a, const char* b) {
    return strcmp(a,b) > 0 ? a : b;
}
```

The diagram consists of three yellow arrows pointing from the text 'Specializáció T ::= const char*' to the corresponding parts of the specialized template function signature: 'const char*', 'const char*', and 'const char*'.

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

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>{ ... };
```

Sablon specializáció (összef.)

Függvények különböző változatai: túlterhelés (overload)

Sablonok esetében a túlterhelés mellett 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()) {}
};
```

T default
konstruktora

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

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



```
v1.a1 ← 0;
```

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



```
v2.a1 ← 3.14;
```

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; }
};
```

default

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

```
// paraméterként kapott típus és konstansa, ami
// csak integrális, vagy pointer típus lehet
template <typename T, T c = T()>
struct V1 {
    T val;
    V1() { val = c; }
};
V1<int, 30> v30; // v30.val ← 30;
V1<int*> v0; // v0.val ← NULL;
```

Sablon, mint paraméter

```
template <class T> struct Pont    { T x, y; };
template <class T> struct Pont3D { T x, y, z; };

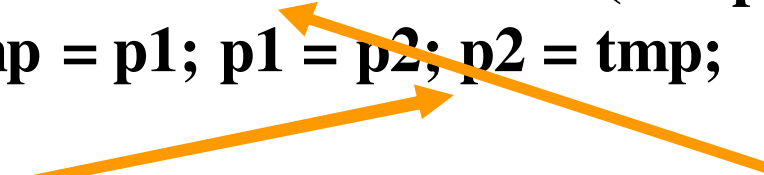
template <class S,
         template <class T> class P = Pont >
struct Idom {
    P<S> origo; // idom origója
};

Idom<int> sikidom1;
    sikidom1.origo;           ← Pont<int>
Idom<double> sikidom2;
    sikidom2.origo;         ← Pont<double>
Idom<int, Pont3D> test;
    test.origo;             ← Pont3D<int>
```


Függvénysablonok paraméterei

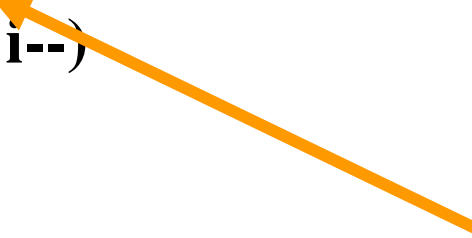
A sablonparaméterek általában **levezethetők** a függvényparaméterekből. Pl:

```
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);
```



Mi is a sablon ?

- Típusbiztos nyelvi elem az általánosításhoz.
- Gyártási forma: a sablonparamétereiktől függően példányosodik:
 - **osztály** vagy **függvény** (valamilyen dekl.) jön belőle létre.
- Paraméter: típus, konstans, függvény, sablon, def.
- A példányok specializálhatók, melyek eltérhetnek az eredeti sablontól.
- Feldolgozása **fordítási idejű** ezért a példányosítás helyének és a sablonnak egy fordítási egységben kell lennie. → gyakran **header fájlba tesszük (.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 };
};
```

Template
metaprogram

```
std::cout << fakt<3>::fval << std::endl;
```

Fordítási időben 4 példány (3,2,1,0) keletkezik

Nehéz nyomkövetni, de nagyon sok könyvtár (pl. boost) használja.

Algoritmus módosítása

- Előfordulhat, hogy egy algoritmus (pl. rendezés) működését módosítani akarjuk egy függvénnyel.
- **Predikátum** (ism): Logikai függvény, ami befolyásolja az algoritmus működését.
- **Sablon-, vagy függvényparaméterként** egy eljárásmodot (függvényt) is átadhatunk, ami lehet:
 - osztály tagfüggvénye, vagy
 - önálló függvény

Példa: leg... elem kiválasztása

```
template <typename T, class S>
T leg1(T a[], int n) {
    T tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (S::select(a[i], tmp)) tmp = a[i];
    return tmp;
}
```

Az **S** sablonparaméter egy olyan osztály, melynek van egy **select** logikai tagfüggvénye, ami az algoritmus működését befolyásolja. pl.:

```
struct kisebb_int {
    static bool select(int a, int b) { return a < b; }
};
```

Példa: leg... elem kiválasztása /2

// Lehet egy sablonból generálódó osztály is

```
template<typename T>
```

```
struct nagyobb { // szokásos nagyobb reláció  
    static bool select(T a, T b) { return a > b; }  
};
```

// Használat:

```
int it[9] = {-5, -4, -3, -2, -1, 0, 1, 2, 3};
```

```
double dt[5] = { .0, .1, .2, .3, 4.4 };
```

```
cout << leg1<int, kisebb_int>(it, 9) << endl; // -5
```

```
cout << leg1<int, nagyobb<int> >(it, 9) << endl; // 3
```

```
cout << leg1<double, nagyobb<double> >(dt, 5); // 4.4
```

Problémák, kérdések

- Megszorítónak tűnik, hogy a tagfüggvényt **select-nek** hívják.
- Miért kell az osztály? Így nem lenne jó?

```
template <typename T, bool select (T, T)>
T leg2 (T a[], int n) {
    T tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (select (a[i], tmp)) tmp = a[i];
    return tmp;
}
template<class T> bool kisebbFv (T a, T b) { return a < b; }
cout << leg2<int, kisebbFv>(it, 9);
```

Funktor

- Függvényként használható objektum.

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

Funktor osztály
(Functor Class)

- Funktor osztály: **függvényként viselkedő osztály.**
- Funktor: Funktor osztály példánya (függvény objektum)

Kettő az egyben?

- A leg1(...) változat egy osztályt vár, a leg2(...) változat pedig függvényt vár sablon paraméterként.
- Hogyan lehetne mindkettőt?
- Kapjon általános típust, és ennek megfelelő példányt kapja meg függvény paraméterként is, amit használjunk függvényként.

```
template <typename T, typename S>
T legElem(const T a[], int n, S sel) {
    ...
    if ( sel(...) )
```

Mindent tudó legElem

```
template <typename T, typename S>
T legElem(const T a[], int n, S sel) {
    T tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (sel(a[i], tmp)) tmp = a[i];
    return tmp;
}
```

A függvény második paramétere logikai függvényként viselkedő valami, ami lehet önálló függvény, vagy funktor is. Pl:

```
cout << legElem<int, kisebbObj<int> >
      (it, 9, kisebbObj<int>() );
```

létre kell hozni

fixArray és legElem?

```
fixArray<int, 100> fixa;  
// használható így?  
legElem(fixa, 100, kisebbObj<int>() );
```

// problémák:

```
template <typename T, typename S>  
T legElem(const T a[], int n, S sel) {  
    T tmp = a[0]; ...
```

// Elemtípus helyett adjuk át a tömböt:

```
template <typename T, typename S>  
T legElem(const T& a, int n, S sel) {  
// Rendben, de honnan lesz elem típusunk a tmp-hez?  
// 1. Adjuk át azt is.
```

// 2. Tegyük be a fixArray osztályba egy belső típust pl:

```
template <class T, unsigned int s = 10>  
class fixArray {  
    T t[s];  
public:  
    typedef T value_type; ...
```

Mindent tudó legElem /2

// indexelhető tárolókhöz, melynek van **value_type** belső típusa

```
template <typename T, typename S>
T legElem(const T& a, int n, S sel) {
    typename T::value_type tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (sel(a[i], tmp)) tmp = a[i];
    return tmp;
}
```

Segítség a fordítónak

// hagyományos tömbökhöz

```
template <typename T, typename S>
T legElem(const T a[], int n, S sel) {
    T tmp = a[0];
    for (int i = 1; i < n; ++i)
        if (sel(a[i], tmp)) tmp = a[i];
    return tmp;
}
```

Túlterhelés

Predikátum (összefoglalás)

- Logikai függvény, **vagy függvény-objektum**, ami befolyásolja az algoritmus működését
- Predikátum megadása

– Sablonparaméterként:

```
template<typename T, bool select(T, T)>>  
T leg2(const T t[], int n);
```

– Függvényparaméterként:

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

Gyakoribb, rugalmasabb
megadási mód

Generikus példa 2

1. Olvassunk be fájl végéig maximum 10 db **valamit** (pl. valós, komplex, stb. számot)!
2. Írjuk ki!
3. Készítsünk másolatot a tömbről!
4. Rendezzük az egyik példányt nagyság szerint növekvően, a másikat csökkenően!
5. Írjuk ki mindkét tömböt.

Példa 2 kidolgozás

- Van generikus fix méretű tömbünk
fixed_size_gen_array.hpp
- Készítünk generikus rendező algoritmust!
generic_sort.hpp
- Készítünk generikus hasonlító operátorokat!
generic_cmp.hpp
- Készítünk segédsablont a beolvasáshoz és kiíráshoz!
- Rakjuk össze az építőelemeket!

Példa 2 kidolgozás /2

```
#include <iostream>

#include "fixed_size_gen_array.hpp" // fix méretű generikus
    // tömb indexelhető, másolható, értékadható
#include "generic_sort.hpp" // indexelhető típus generikus
    // rendezése
#include "generic_cmp.hpp" // generikus hasonlító függvények

// A bemutatott megoldás bármilyen indexelhető típussal,
// így pl. az alaptípusokból létrehozott tömbökkel is működik,
// ha az elemtípus másolható, értékadható és értelmezett a
// <, > <<, és >> művelet.
```


Példa 2 kidolgozás /2

/// Indexelhető típus elemeit kiírja egy stream-re

```
template <typename T>
```

```
void CopyToStream(const T& t, size_t n, std::ostream& os) {
```

```
    for (size_t i = 0; i < n; ++i)
```

```
        os << t[i] << ", ";
```

```
    os << std::endl;
```

```
}
```

/// Indexelhető típus elemeit beolvassa egy stream-ről

```
template <typename T>
```

```
int ReadFromStream(T& t, size_t n, std::istream& is) {
```

```
    size_t cnt = 0;
```

```
    while (cnt < n && is >> t[cnt]) cnt++;
```

```
    return cnt;
```

```
}
```

Példa 2 kidolgozás /3

```
int main() {
    fixArray<double, 10> be; // max. 10 elemű double tömb

    int db = ReadFromStream(be, 10, std::cin); // beolvasunk
    CopyToStream(be, db, std::cout); // kiírjuk
    fixArray<double, 10> masolat = be; // másolat készül
    insertionSort(be, db, less<double>()); // növekvő rendezés
    insertionSort(masolat, db, greater<double>()); // csökkenő
    CopyToStream(be, db, std::cout); // kiírjuk az elemeket
    CopyToStream(masolat, db, std::cout); // kiírjuk az elemeket
}
```

http://git.ik.bme.hu/prog2/eloadas_peldak/ea_07 → gen_pelda2

fixed_size_gen_array.hpp

```
#ifndef FIXED_SIZE_GEN_ARRAY_H
#define FIXED_SIZE_GEN_ARRAY_H
template <class T, unsigned int s>
class fixArray {
    T t[s];
    void chkIdx(unsigned i) const { if (i >= s) throw "hiba"; }
public:
    typedef T value_type; // a példában nem használjuk
    T& operator[](size_t i) {
        chkIdx(i); return t[i];
    }
    const T& operator[](size_t i) const {
        chkIdx(i); return t[i];
    }
};
#endif
```

generic_sort.hpp

```
#ifndef .....
```

```
template <typename T>
```

```
void swap(T& a, T& b) { T tmp = a; a = b; b = tmp; }
```

```
template <typename T, class F>
```

```
void insertionSort (T& a size_t n, F cmp) {
```

```
    for (size_t i = 1; i < n; i++) {
```

```
        size_t j = i;
```

```
        while (j > 0 && cmp(a[j], a[j-1])) {
```

```
            swap(a[j], a[j-1]);
```

```
            j--;
```

```
        }
```

```
    }
```

```
}
```

generic_cmp.hpp

```
#ifndef GENERIC_CMP_H
#define GENERIC_CMP_H
template <typename T>
struct less {
    bool operator()(const T& a, const T& b) const {
        return a < b;
    }
};

template <typename T>
struct greater {
    bool operator()(const T& a, const T& b) const {
        return a > b;
    }
};
#endif
```

Módosítsuk a kiíróát!

- Legyen predikátuma:

```
template <typename T, class F>
void CopyToStream(const T& t, size_t n,
                  std::ostream& os, F fun) {
    for (size_t i = 0; i < n; ++i)
        if (fun(t[i])) os << t[i] << ",";
    os << std::endl
}
```

- Szeretnénk kiírni elsőször az 5, majd a 15, és a 25-nél nagyobb értékeket.
- Ennyi predikátumfüggvényt kell készítenünk?

Ötlet: tároljunk egy ref. értéket

```
template <typename T>
class greater_than {
    T ref_value;
public:
    greater_than(const T& val) : ref_value(val) {}
    bool operator()(const T& a) const {
        return a > ref_value; }
};
```

//Példa:

```
greater_than<int>gt(10); // konstruktor letárolja a 10-et
gt(8); // fv. hívás op. ← false
gt(15); // fv. hívás op. ← true
```

Módosított kiíró használata

```
int main() {  
    int it[9] = {-5, -4, -3, -2, -1, 0, 1, 2, 3 };  
  
    CopyToStream(it, 9, std::cout, greater_than<int>(-1) );  
                                                    // 0,1,2,3,  
    CopyToStream(it, 9, std::cout, greater_than<int>(0) );  
                                                    // 1,2,3,  
    CopyToStream(it, 9, std::cout, greater_than<int>(1) );  
                                                    // 2,3,  
}
```


Összefoglalás

- A C-ben megtanult preprocesszor trükkökkel elvileg általánosíthatók az osztályok és függvények, de nem biztonságos, és nem mindig megoldható.
- **template**: típusbiztos nyelvi elem az általánosításhoz.
- Formálisan:

template < templ_par_list > declaration

Összefoglalás /2

- Generikus osztályokkal tovább általánosíthatjuk az adatszerkezeteket:
 - 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 algoritmusokat:
 - 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
 - Függvénytípus sablon felüldefiniálható

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. (ld. fixArray)