Template-ek. Generikus programozás. Template metaprogramozás.

C++ programozás – 7. óra
Széchenyi István Egyetem

`http://dropbox.com/...`

2017

Template-ek: bevezetés

- Mire jók a **template**-ek, milyen célt szolgálnak?
  - **Osztályok, függvények és nevek (aliasok) típussal történő paraméterezését**, hogy ne kelljen többször ugyanazt lekódolni.
  - **Általános koncepciók tömör reprezentálását és kombinálását** (pl. "*minden, amire adott operátorok értelmezhetőek*", "*minden, ami összeszorozható*" stb.)
- Gondoljunk a template-ekre úgy, mintha típus-absztrakciók lennének! Adott template csak azoktól a (közös) dolgoktól függ, amelyeket bizonyos típusokból meg akarunk ragadni
  - másszóval: egyébként semmiféle kapcsolatot nem feltételez a felhasználható típusok között
  - Pl. azon kívül, hogy mátrixok és vektorok is skalár-szorozhatóak, semmilyen kapcsolat nem kell, hogy közöttük legyen (nem kell egymásból származniuk, vagy ilyesmi)

Template-ek: bevezetés II

- A template-ek egyik nagy pozitívuma, hogy **a tervezőnek nem kell lekötnie, hogy csak adott típus használható, más nem**.
  - A kód így rugalmasabb lesz, hiszen nem kell *n* különböző típusra *n*-szer majdnem ugyanazt lekódolni
- Pl. az összes STL-absztrakció tulajdonképpen template class, ahogy a rajtuk végezhető műveletek is
  - *string*, *ostream*, *regex*, *complex*, *list*, *map*, *unique_ptr*, *thread*, ...
  - stringek komparálása, a << operátor, stb.
- Az, hogy ezek a típusok önmagukban is használhatók, egy *using*... utasításnak köszönhető
  - Pl. *using string = std::basic_string<char>;*

Template-ek: bevezetés III

- Ugyanakkor a történet nem ilyen egyszerű. Lesznek bizonyos dolgok, amikre template-használat esetén jobban oda kell figyelnünk
- Az egyik, hogy nem adható meg explicit módon, hogy egy típus-argumentumra milyen megkötések érvényesek.
  - Pl. nincs olyan, hogy "*minden olyan T típus, amelyre igaz hogy ...*"
  - Konkrétan a template-ek használata a típushelyesség ellenőrzését elodázza: kevésbé a deklaráció számít, mint inkább a definíció (ahogy a template-elt osztályt, függvényt használjuk!)
  - A tervező csak reménykedni tud, hogy az interfész használója megfelelő tulajdonságú típusokat használ
- Fontos az is, hogy a template-használatot ne vigyük túlzásba. Figyelmetlenül használva sok felesleges kódot is generálhat a fordító.

Template osztályok

- Példa egyszerű string template-re:

```
template<typename C>
class String{
public:
  String();
  explicit String(const C*);
  String(const String&);
  String operator=(const String&);
  //...
  C& operator[](int n){
    return ptr[n]; //ellenorizetlen hozzaferes
  }

  String& operator+=(C c); //c hozzadasa a string vegehez
  //...
private:
  static const int short_max = 15;
  int sz;
  C* ptr; //ptr sz darab C-re mutat
};
```

- Kvíz: miért kell *const* módosító a konstruktorokba?

Template osztályok II

```
template<typename C>
class String{
public:
  String();
  explicit String(const C*);
  String(const String&);
  String operator=(const String&);
  //...
  C& operator[](int n){
    return ptr[n]; //ellenorizetlen hozzaferes
  }

  String& operator+=(C c); //c hozzadasa a string vegehez
  //...
private:
  static const int short_max = 15;
  int sz;
  C* ptr; //ptr sz darab C-re mutat
};
```

- Válasz: mivel a konstruktor nem módosítja az argumentumot, ezért az lehet valamilyen temp object is (nem muszáj, lehet).
- Kvíz 2: miért explicit a második konstruktor?

Template osztályok III

- Válasz: így garantálni tudjuk, hogy azokat a függvényeket, melyek *String* típust várnak, ne lehessen implicit módon *constC** típussal meghívni.
- Pl. *concat* függvény MyString típust vár:

```
#include<iostream>

template<typename C>
class MyString{
  public:
    MyString();
    explicit MyString(const C*);
    MyString(const MyString&);
    ~MyString(){delete[] ptr;}
    C& operator[](int n);
    void concat(MyString);
    void print();

  private:
    int sz;
    C* ptr;
};
```

```
#include "mystring.hpp"

int main(){
  MyString<char> mstr2("baba");
  //mstr2.concat("papa"); //mukodik ha konstruktor nem
        explicit
  mstr2.concat(MyString<char>("papa")); //ha ctor explicit,
        muszaj igy
  mstr2.print();

  return 0;
}
```

Template osztályok IV

- *template<typename C>* azt jelenti, hogy *C*-t mint típust használhatjuk a kódban.
- Ekvivalens: *template<class C>*. Mindkét esetben *C* egy típus neve.
- Ez alapján jónéhány string típus példányosítható:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

struct JChar {/* ... */} //meg japan karakter is hasznalhato
String<JChar> js;

int main() //count the occurrences of each word on input
{
  map<String<char>, int> m;
  for (String<char> buf; std::cin >> buf; ){
    ++m[buf]
  }
  // write out result
}
```

Template osztályok V - A kód kisebb lehet

- Template-tel létrehozott osztály ugyanolyan osztály mint bármely másik, csak a típus-paraméter összes előfordulása behelyettesítésre kerül.
  - Ellenben előfordulhat, hogy a generált kód rövidebb lesz!
  - Ha olyan típussal példányosítjuk az osztályt, melyre bizonyos függvényeket nem használunk, akkor kisebb lesz a generált kód.
- Egy osztályt akkor és csak akkor kell adott template-tel legenerálni, ha létezik példányosítása!

```
String<JChar> jcs(``valami-japan-szoveg'') //amig nincs ilyen, nincs szukseg ra hogy String<JChar>
      osztalyt legeneraljuk
```

- Ugyanígy ha egy függvényt nem használunk, felesleges betenni a legenerált kódba! (a fordítónak)
- Ha viszont le kell generálni, azt **specializációnak** nevezzük

Template osztályok VI - A kód kisebb lehet

- Például a *List* < *Glob* > specializációban nem lesz *sort*() függvény, csak a *List* < *String* > specializációban

```
template<typename T>
class List {
  //...
public:
  void sort();
};

class Glob {
  //... semmilyen komparalo operator nincs
};

void f(List<Glob>& lb, List<string>& ls)
{
  ls.sort();
  // ... lb-vel is csinalunk dolgokat, de lb.sort()-ot nem hivjuk meg
}
```

Template osztályok VII - Deklaráció és definíció

- Ezzel együtt template osztály definiálására kicsit más szabályok vonatkoznak, mint a sima osztályokra:
  - A template sem nem osztály, sem nem fv, csak egy "minta" amit a fordító felhasznál
  - A kód generálásához a fordítónak látnia kell a template definícióját is (nemcsak a deklarációját)!
    - Pl. ha `Foo<int>` típusú változót szeretnénk használni, a fordítónak látnia kell a template definícióját is, hogy `int` típust bele tudja helyezni.
  - Mivel a fordítók (általában) egyszerre csak egy .cpp fájlt tudnak nézni, ezért a template osztály definícióinak egyazon .h fájlban kell lenniük, mint a deklarációja.

Template osztályok VIII - Deklaráció és definíció

- Például:

```
//foo.h
template<typename T>
class Foo{
public:
  Foo();
  void someMethod(T, x);
private:
  T x;
};

template<typename T>
Foo<T>::Foo() {
  ...
}

template<typename T>
void Foo<T>::someMethod(T x) {
  ...
}
```

Tegyük fel, hogy ilyen kódunk van:

```
//Bar.cpp

void valami(){
  ...
  Foo<int> f;
  f.someMethod(5);
}
```

Világos, hogy `Foo` template-et `int` típusra kell specializálni. De ezt csak akkor lehet, ha a fordító most látja a definíciót is (ha lenne egy Foo.cpp fájl, amit külön fordítottunk le, akkor linkelési hiba!)

Template osztályok IX

- Hasznos tanács: amikor template osztályt tervezünk, először template nélkül debuggoljuk, és csak utána adjuk hozzá a template-et.
  - Így ha valami nem működik, tudjuk hogy nem a template miatt van a probléma
  - Ráadásul olyan hibákat elkaphatunk, amelyek valószínűleg minden specializációnál előfordulnának
- Ugyanígy, ha kapunk egy kódot amiben template is van, először egy példán keresztül próbáljuk meg megérteni. Utána sokkal könnyebb dolgunk lesz!
- Template osztály tagjait ugyanúgy definiáljuk, mint más osztálynál. Nincs szükség arra sem, hogy az osztályon belül definiáljunk mindent. Viszont ettől még osztályon kívül is meg kell mondanunk, hogy egy függvény (metódus) template-elt!

Template osztályok X

- Például a .cpp fájlban:

```
template<typename C>
String<C>::String() : sz{0}, ptr{nullptr}
{
}

template<typename C>
String& String<C>::operator+=(C c)
{
  //add c to the end of this string...
  return *this;
}
```

Template osztályok XI - Típusellenőrzés nehézségei

- Template-ek használata hatékony kódot eredményezhet, ugyanakkor a típusellenőrzést és típushibákról való jelentések elkészítését ez a fajta rugalmasság megnehezíti.
  - A kódgenerálás ugyanis a build folyamatnak egy viszonylag késő fázisában történik (ha már tudjuk pl. hogy az összes többi template-et milyen típusokkal használjuk)...
  - Sok olyan részlet lesz ezért a kódban, amire a template-et készítő programozó nem is számít!
- Mondtuk, hogy nem adhatunk meg kényszereket template-argumentumokra — ez pl. reménytelen (nem tudjuk megmondani a fordítónak, hogy *Elem* típus kompatibilis kell hogy legyen a *Container*-rel):

```
template<Container Cont, typename Elem>
requires Equal_comparable<Cont::value_type, Elem>()
int find_index(Cont& c, Elem e);
```

Template osztályok XII - Dinamikus polimorfizmus nem működik

- Ezért ha olyan paraméterekkel példányosítunk egy template-et, amire a template létrehozója nem gondolt, csúnya hibát kaphatunk (ráadásul nem a példányosításkor, hanem a felhasználáskor)
- Ember legyen a talpán, aki az ezután kapott hibaüzenetből visszafejti, hogy a template-et ilyen vagy olyan típussal miért nem lehet használni.
- Kérdés: mi történik akkor, ha egymásból származó osztályokat használunk template paraméterként? Érvényes marad a kapcsolat? Sajnos nem!

```
Shape* p {new Circle(p, 100)}; //Circle* atkonvertalodik Shape*-ra
vector<Shape>* q {new vector<Circle>{}}; //hiba: vector<Circle>* nem konvertalodik vector<Shape>*-ra
vector<Shape> vs {vector<Circle>{}}; //hiba: vector<Circle> nem konvertalodik vector<Shape>-re
vector<Shape*> vs {vector<Circle*>{}}; //hiba: vector<Circle*> nem konvertalodik vector<Shape*>-ra
```

Template osztályok XIII - Dinamikus polimorfizmus nem működik

- De miért nem!? Az ember azt hinné, hogy ha a kör egyfajta alakzat, akkor egy köröket tartalmazó vektor alakzatokat tartalmazó vektor is egyben.
- Van azonban egy gyakorlati oka, hogy miért nincs így. Tfh van körünk és háromszögünk, és:

```
void f(set<Shape*>& s)
{
  //...
  s.insert(new Triangle{p1, p2, p3});
}

void g(set<Circle*>& s)
{
  f(s); //hiba, tipusutkozes: s set<Circle*>, nem set<Shape*>
  //problema: Triangle nem egyfajta Circle, ezert f() fv-ben gond lenne
}
```

Template osztályok XIV - Dinamikus polimorfizmus nem működik

- Látszik: két függvény összefűzésekor elveszítenénk a dinamikus polimorfizmus nyújtotta előnyöket.
- Amikor sima *Shape*∗, *Circle*∗ és *Triangle*∗ típusokat használunk, akkor sem az a cél hogy az alakzat típusát kihasználjuk, hanem az, hogy a Shape osztály nyújtotta interfészt kihasználjuk.
  - Egy ilyen fv meghívásakor sem elegáns belenézni az argumentumba, hogy az objektum tulajdonképpen Kör vagy Háromszög-e (ezért léteznek virtuális fv-ek).
- Persze az más kérdés, hogy amik egy konkrét konténerben benne vannak, azok mindenféle alakzatra mutató pointerek lehetnek. Ez pl. $f()$ függvény önmagában nem volt hiba
- Az elemek tehát egyenként vizsgálhatóak. En bloc viszont a konténerek nem felcserélhetőek!

Template osztályok XV - Dinamikus polimorfizmus nem működik

- A témát folytatva: még rosszabb lenne, ha tömbökre ugyanez a hiba előfordulhatna
- A beépített tömb ugyanis nem tartalmaz annyi garanciát sem, mint a konténer osztályok

```
void maul(Shape* p, int n) // Veszelyes osdi megkozelites
{
  for(int i=0; i!=n; ++i) p[i].draw(); //artatlannak tunik, de nem az!
}

void user()
{
  Circle image[10];
  //...
  maul(image, 10); //ld. lent
}
```

Template osztályok XVI - Dinamikus polimorfizmus nem működik

- *maul*() meghívásakor:
  - először implicit módon *Circle*∗ típusúra változik a tömb (a pointer annak első elemére mutat)
  - ezt követően *Circle*∗-ból *Shape*∗ lesz
  - de egy Shape objektum ugye kisebb, tehát a *user*() és *maul*() függvények mást látnak
- Ezzel egy objektum esetén nincs probléma, de a tömbhasználat már nem OK!
  - A pointerek által mutatott mem.terület nem "menedzselt", *maul*() nem tudja, hol vannak az elemek határai
  - Nem a *p*[0]-ra történő hívással van gond: az egy *Circle* objektum és annak a vtáblája megmutatja, hol van a meghívott *draw* fv. A második *Circle* objektum viszont nem ott kezdődik, ahol *maul*() hiszi!
  - Csak remélhetjük, hogy a program azonnal összeomlik.

Template osztályok XVII - Dinamikus polimorfizmus nem működik

- Ezért is jobb, ha a containereket preferáljuk a beépített tömbök helyett
- Legyünk óvatosak, ha ilyet látunk, hogy *void f(T* p, int count)*.
  - Ha *T* szülőosztály lehet, *count* pedig az elemek számára vonatkozik, bármikor gond lehet.
- Az előző példában, ha *maul*() egy $vector < Shape* >$ típust várna, akkor a vektor minden eleme külön pointer lenne, ami bármilyen (*Shape*-ből származó) objektumra mutathatna. Úgy már akár elemenként is meghívhatnánk a draw fv-t.
- $vector < Shape >$ viszont ugyanezen okokból nem működne (nem ismerjük *Vector* osztály belső működését, de elképzelhető hogy igazából tömböt használ).

Template osztályok XVIII

- Végezetül: template osztályokra nagyjából ugyanazok a dolgok vonatkoznak, mint a sima osztályokra
- Pár kivétel azért van:
  - *constexpr* nem használható template osztályban, hiszen az csak egy minta, ami direktben nem fordítódik le
  - Template osztályban lehetnek virtuális függvények, viszont azok nem lehetnek template-eltek. (A legenerált osztályokhoz más template típus esetén külön vtábla kéne, ami kezelhetetlenül bonyolult lenne, ezért nem engedi meg a nyelv)
    - Persze a kikötés csak arra vonatkozik, amikor a virtuális fv új template-elt paramétert definiálna.
    - Az osztálynak magának is lehetnek template-jei, amiket a fv felhasználhat, hiszen az osztályt amikor generáljuk, akkor az egész osztály típusát már ismerjük.
    - Pl. az STL-ben is van öröklés, pl. *Vector* meg *List* meg ilyenek mind a *Container* osztályból örökölnek.

Template függvények

- Ugyanúgy ahogy más esetekben, template-elt osztályokból / függvényekből is csak egy definíció létezhet egy programban.
- Ellenben ha speciaizációtól függő implementációt szeretnénk adott típusra, ezt megtehetjük. Például:

```
template<typename T>
T sqrt(T);

template<typename T>
complex<T> sqrt(complex<T>);

double sqrt(double);

void f(complex<double> z){
  sqrt(2.0); //sqrt(double) alapbol a template nelkulit preferalja a fordito!
  sqrt(2); //sqrt<int>(int) kiveve ha implicit konverzio kellene!
  sqrt(z); //sqrt<double>(complex<double>) itt pedig bonyibb a szitu, ld. lent
}
```

Template függvények II

- Ilyenkor nem triviális eldönteni, melyik specializációt kell meghívni, de a fordító mindig azt választja, amelyik argumentumai szempontjából a többi lehetőségnél specializáltabb
  - Például *sqrt<double>(complex<double>)* specializáltabb, mint *sqrt<complex<double»(complex<double>)*, mert az utóbbi template-je *T sqrt(T)*, ahol *T*-be bármi behelyettesíthető (pl. *complex* $< T >$ is).
- Összességében elmondható, hogy:
  - Template-ek lehetővé teszik, hogy nagyon rövid forrásból temérdek mennyiségű kódot generáljunk. Ez veszélyes is lehet, ha pl. sok, majdnem egyforma osztály kerül legenerálásra.
  - Ugyanakkor ha jól használjuk őket, rengeteg fáradtságtól kímélhetjük meg magunkat. Ezen kívül jó minőségű kódot kapunk. Pl. sok indirekt fv-hívás megspórolható, ha template-eket és inline-olt kódot használunk.

Template függvények III

- Itt például a *Compare* osztálynak ha van *operator*() nevű függvénye, a fordító inline-olni fogja:

```
template<typename T, typename Compare = std::less<T>>
void sort(vector<T>& v){
  //Knuth Shell sort-ja
  Compare cmp; //default Compare objektum
  const int n = v.size();

  for(int gap = n/2; 0<gap; gap/=2){
    for(int i = gap; i < n; i++){
      for(int j = i-gap; 0 <= j; j-=gap){
        if(cmp(v[j+gap], v[j]))
          swap(v[j], v[j+gap]);
      }
    }
  }

}
```

Template függvények IV

- A kódot saját komparátor-típus definiálását követően szabadon felhasználhatjuk:

```
struct No_case{
  bool operator()(const string& a, const string& b) const;
};
```

- Például az STL könyvtárban van egy *sort*() függvény kétféle változatban:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator fst, RandomAccessIterator lst);

template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator fst, RandomAccessIterator lst, Compare c);
```

Template függvények V – paraméterek kikövetkeztetése

- Fontos, hogy **a fordító ki tudja következtetni a template argumentumainak típusát**. *Ez csak akkor lehetséges, ha a paraméter-lista egyértelműen meghatározza a típust.* Például:

```
template<typename T, int max>
struct Buffer{
  T buf[max];
};

template<typename T, int max>
T& lookup(Buffer<T, max>& b, const char* p);

Record& f(Buffer<string, 128>& buf, const char* p){
  return lookup(buf, p); //nem fontos megmondani, hogy melyik lookup fv kell
}
```

- Ezzel együtt **template osztály inicializálásakor a paramétereket mindig explicite meg kell adni**. Ennek oka, hogy több konstruktor esetén igen bonyolult lehet kideríteni, hogy melyikről is van szó.

Template függvények VI – paraméterek kikövetkeztetése

- Kicsit olyan téma, mint a függvény-felültöltés (lehetne bonyolult szabályokat kreálni), csak konstruktorok esetében duplán zavaró lenne (hiszen az implicit konverzió ott amúgy is szerephez jut)
- Ha kikövetkeztetett típusú objektumot szeretnénk létrehozni, ezt gyakran megtehetjük valamilyen segédfüggvénnyel. Pl.:

```
template<typename T1, typename T2>
pair<T1, T2> make_pair(T1 a, T2 b){
  return {a, b};
}

auto x = make_pair(1,2); // x pair<int, int>
auto y = make_pair("Gyor",2.1); // x pair<string, double>
```

- Amikor viszont a paraméterekből nem derülhet ki a típus (pl. mert nincs paraméter, vagy nem egyértelmű), kötelező megadni:

Template függvények VII – paraméterek kikövetkeztetése

```
template<typename T>
T* create(); //csinaljunk egy T objektumot es adjunk vissza ra pointert

void f(){
  vector<int> v;
  int* p = create<int>();
}
```

- Hasonló módon működnek a castok (`static_cast`, `dynamic_cast`, stb.).
- Hogy pontosan hogyan működik a template fv-ek paramétereinek feloldása, az elég bonyolult. Mindenesetre érdemes tudni, hogy általában működni fog.

Template-ek és aliasok

- Típus aliast definiálhatunk a `using` és `typedef` kulcsszavakkal is. A korábbi általánosabb abban a fontos értelemben, hogy segítségével template-ek esetén olyan aliast definiálhatunk, amelyben néhány (nem az összes) típust lekötünk. Pl:

```
template<typename T, typename Allocator = allocator<T>> vector;
using Cvec = vector<char>; // mindket argumentum specifikalt
Cvec vc = {'a', 'b'};

template<typename T>
using Vec = vector<T,My_alloc<T>> //csak a 2. argumentum specifikalt
Vec<int> fib = {0, 1, 1, 2, 3, 5, 8, 13};
```

- Ami ebben jó, hogy az alias használata teljesen ekvivalens az eredeti template használatával:

```
vector<int,My_alloc<int>> verbose = fib;
```

Template-ek és aliasok II

- Ez a fajta működés azt jelenti, hogy template-et az alias használatával is specializálhatunk (C++11-től kezdve!)

```
template<int>
struct int_exact_traits{ //alapotlet: int_exact_traits<N>::type pontosan N-bites tipus
  using type = int;
};

template<>
struct int_exact_traits<8>{ //specializacio 8-cal... template kulcsszo utan mar ures!
  using type = char;
};

template<>
struct int_exact_traits<16>{ //specializacio 16-tal... template kulcsszo utan mar ures!
  using type = short;
};

template<int N>
using int_exact = typename int_exact_traits<N>::type; //egyszerubb jeloles, visszakapjuk a
      tipusnevet!

int_exact<8> a = 7; //int_exact<8> 8-bites int tipus
```

Generikus programozás: bevezetés

- Mire jók a template-ek? Milyen programozási stílust támogatnak?
  - Típusokat (valamint értékeket és template-eket is) átadhatunk paraméterként, információveszteség nélkül. Ez remek lehetőséget teremt inline-olásra, és ezt hatékonyan kihasználják a jelenlegi implementációk.
  - Késleltetetten történik meg a típusellenőrzés (példányosításkor!), ezért különböző kontextusokból fűzhetünk össze információkat.
  - Konstans értékeket is átadhatunk paraméterként. Ez lehetőséget teremt fordításidejű számításokra.
- Mindez segíthet kompakt és hatékony kód írásában.
- Kiemelkedően fontos alkalmazás: **generikus programozás**. Ilyenkor általános algoritmusokat készítünk, nem tudjuk pontosan milyen típusra.

Generikus programozás: bevezetés

- A másik fontos paradigma: **template metaprogramozás**. Ebben a paradigmában a template-re úgy tekintünk, mint típusok és kód generálásának lehetőségére.
  - Persze a generikus programozásban is jelen van, de az inkább a típus általánosításáról, ez pedig kódrészletek fordításidejű generálásáról szól
- Mivel a template-ek típusellenőrzése a definícióban történik (nem a deklarációban), ezért rugalmasabb és elősegíti az ún. *duck typing* módszert ("*If it walks like a duck and quacks like a duck, it's a duck*"). Az a kérdés: lefordul-e a kód. Nem az a kérdés, hogy megfelelő típust használtunk-e.
- Két fontos téma: algorithm lifting, és koncepciók.
  - hogyan általánosíthatunk úgy egy algoritmust, hogy (ésszerűen) a bemenő típusok lehető legtágabb halmazára működjön?
  - hogyan tudjuk a lehető legpontosabban megszabni, hogy milyen típusokra működik egy algoritmus?

Algoritmusok általánosítása

- A legjobban tesszük, ha template függvényeinket konkrét megvalósítás alapján alakítjuk ki. Ezt nevezzük *lifting*-nek, vagy magyarul (jobb szó híján) általánosításnak.
- Fontos, hogy az ésszerűség határain belül maradjunk. Ha túl általánosra választjuk az algoritmust, lehet hogy feleslegesen rontjuk a hatékonyságot. Ezért is jobb, ha már van egy példánk, mint ha absztrakt elvekből indulnánk ki rögtön.
- Nézzünk egy konkrét példát:

```
double add_all(double* array, int n)
//konkret algoritmus double tipusu tombon
{
  double s {0};
  for(int i=0; i<n; ++i){
    s = s+array[i]
  }
  return s;
}
```

Algoritmusok általánosítása

- Egyértelmű, hogy ez a kód mit csinál. De nézzünk egy hasonló példát:

```
struct Node{
  Node* next;
  int data;
};

int sum_elements(Node* first, Node* last)
//ujabb konkret algoritmus, inteket tarolo listakra
{
  int s = 0;
  while(first!=last){
    s += first->data;
    first = first->next;
  }
  return s;
}
```

Algoritmusok általánosítása

- A két példa stílusában és részleteiben eltér, de gyakorlottabb programozók rögtön mondanák, hogy valójában mindkettő az *akkumulátor* algoritmus megvalósítása. A módszernek több neve van: *reduce*, *fold*, *sum*, *aggregate*, stb.
- Az általánosításhoz először megállapítjuk, hogy a felhasznált típusok tekintetében a `double` illetve `int` típus, illetve a tömb és láncolt lista típusok részletkérdések. Először írjunk valamilyen pszeudo-kódot:

```
T sum(data)
{
  T s = 0;
  while(not at end){
    s = s + current value
    get next data element
  }
  return s
}
```

Algoritmusok általánosítása

- A megvalósításhoz három dolgot kell konkretizálnunk:
  - Mi az, hogy *not at end*?
  - Mi az, hogy *get current value*?
  - Mi az, hogy *get next data element*?
- Az adatok tekintetében szükségünk van továbbá egy *inicializáló*, *összeadó* és *return* műveletre. Ez alapján már írhatjuk, hogy:

```
//STL-szeru kod
template<typename Iter, typename Val>
Val sum(Iter first, Iter last){
  Val s = 0;
  while(first!=last){
    s = s + *first;
    ++first;
  }
  return s;
}
```

Algoritmusok általánosítása

- Itt kihasználtuk, hogy ismerjük az STL reprezentációs stílusát. Minden szekvenciához tartozik egy iterátor-pár, amely támogatja a pointer-dereferencián keresztüli elérést, az inkrementáló-operátort és a nem-egyenlő operátort annak észlelésére, hogy a szekvencia végére értünk.
- Ez az algoritmus mostmár tömöbkre és láncolt listákra is működhet. A tömbök iterátornak számítanak. A `Node` típusunkat pedig átalakíthatjuk iterátorrá. Az első ötletünk lehet az alábbi:

```
double ad[] = {1,2,3,4};
double sd = sum<double*, double>(ad, ad+4);

Node n3 = Node(nullptr, 5);
Node n2 = Node(&n3, 2);
Node n1 = Node(&n2, 1);
int si = sum<Node*, int>(&n1, end(&n1));
```

Algoritmusok általánosítása

- Ezzel az a gond, hogy míg tömb használata esetén gond nélkül használhattunk pointert, addig láncolt listás megvalósításnál a pointer inkrementálásakor nem feltétlenül a következő elemet fogjuk kapni (mivel nem szomszédos memóriaterületeket foglalnak el a lista elemei).
- Gondolhatjuk: akkor definiáljuk felül az inkrementáló operátort! Sajnos ezt pointerre nem tehetjük meg:

```
//nem mukodik mert operator argumentumai kozul legalabb az egyiknek tipusnak kell lennie
//(pointer nem tipus)
Node* operator++(Node* n){
  return n->next;
}
```

- Ezért olyan típust kell definiálnunk, ami iterátorként működik láncolt listás elemeken. Akkor ez a kód működni fog:

Algoritmusok általánosítása

```
Node n3 = Node(nullptr, 5);
Node n2 = Node(&n3, 2);
Node n1 = Node(&n2, 1);
//Node osztalynak van egy iteratora
int si = sum<Node::iterator, int>(n1.begin(), n1.end());//iterator inkrementalas feluldefinialhato!
```

- Tisztább lenne, ha nem Node osztálynak lenne iterátora, hanem egy olyan konténer osztálynak, amely Node-ok alapján valósít meg láncolt listát (pl. `NodeLinkedList`), ami memóriakezelés szempontjából is jobb lenne... ahogy pl. STL osztályokban vannak ilyenek:

```
std::vector<int> myvector;
//...
for(std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++i){
  //na, az iterator objektum mar inkrementalhato, mert UDT-nek szamit!
  //az iterator vector osztalyhoz tartozik, nem pedig a vektor elemeihez
}
```

Algoritmusok általánosítása

- De ezt most hagyjuk... Node struct így módosul:

```
struct Node{
  Node* next; int data;
  Node(Node* nxt, int i){next = nxt; data = i;}

  class iterator{
    Node* currentNode;
  public:
    iterator(Node* n){currentNode = n;}
    int operator*(){return currentNode->data;} //kell, hogy az osszeadas mukodjon
    iterator& operator++(){
      currentNode = currentNode->next;
      return *this;
    }//inkrementalashoz
    friend bool operator!=(const iterator&, const iterator&); //while ciklushoz
  };

  friend bool operator!=(const iterator& a, const iterator& b){
    return (a.currentNode != b.currentNode);
  }

  iterator& getIterator(){return iterator(this);}
};
```

Algoritmusok általánosítása

- A program kódja pedig így néz ki:

```
Node n3 = Node(nullptr, 5);
Node n2 = Node(&n3, 2);
Node n1 = Node(&n2, 1);
int si = sum<Node::iterator, int>(n1.getIterator(), nullptr);
//ha most az iterator inkabb kontener-osztalyhoz tartozna, lehetne begin() meg end(), es
//end() az utolso utani elemre mutatna (hiszen a kontener pl. tombot hasznalna)
//ehelyett most ugy teszunk mintha end() mindig nullptr-t adna vissza. Ha n3 Node-ot tartalmazo
//iteratort inkrementaljuk, ugyanugy nullptr-t kapunk.
```

- Ez jóval általánosabb, mint legtöbb gyakorlatban futó kód. Pl. a függvényt meghívhatjuk leb.pontos számokat tartalmazó `List` konténerekkel, `int[]` tömbökkel, `vector<char>` típusokkal, stb.

Algoritmusok általánosítása

- Ami még fontosabb: a kód ugyanolyan hatékony, mint eredetileg volt.
- Ami viszont kényelmetlen, hogy az eredmény típusát a fv nem tudja kikövetkeztetni. Ezen könnyen változathatunk (emlékszünk? paraméter alapján ki lehet következtetni!) – ezért tegyük lehetővé default érték megadását!

```
template<typename Iter, typename Val>
Val accumulate(Iter first, Iter last, Val s){
  while(first!=last){
    s = s + *first;
    ++first;
  }
  return s;
}

double ad[] = {1,2,3,4};
double s1 = accumulate(ad, ad+4, 0.0); //automatikusan double
double s2 = accumulate(ad, ad+4, 0); //akkumulaljunk int-be
```

Algoritmusok általánosítása

- Ha idáig eljutottunk, jön a következő kérdés: miért használjunk mindig '+' operátort? Olyan sosincs, hogy inkább szoroznánk? Újabb általánosítás:

```
template<typename Iter, typename Val, typename Oper>
Val accumulate(Iter first, Iter last, Val s, Oper op){
  while(first!=last){
    s = op(s,*first);
    ++first;
  }
  return s;
}

double ad[] = {1,2,3,4};
double s1 = accumulate(ad, ad+4, 0.0, std::plus<double>()); //kell hozza #include<functional>
double s2 = accumulate(ad, ad+4, 1.0, std::multiplies<double>());
```

- Itt látszik, miért jó még a kezdeti értéke megadásának lehetősége: a szorzás és 0.0 érték nem passzolnak egymáshoz.

Koncepciók tervezése

- Milyen követelményeknek kell megfelelnie egy típusnak ahhoz, hogy template-be behelyettesíthető lehessen? A lehetőségek végtelenek, mert tetszőleges tulajdonságokat elképzelhetünk:
  - Típusok, melyek összeadhatóak, de nem kivonhatóak egymásból
  - Típusok, melyek másolhatnak, de nem mozgathatnak értékeket
  - Típusok, melyekre értelmezett másolási műveletek valójában nem másolnak (pl. *shallow copy*)
  - Típusok, melyekre a == operátor összehasonlítást végez, illetve olyan típusok, melyekre ehelyett egy `compare()` fv-t defiálunk
  - Típusok, melyeket `plus()` fv segítségével adhatunk össze, illetve olyan típusok, melyekre ehelyett egy `operator+()` fv-t definiálunk
- Az eredmény totális káosz. Minden osztálynak egyedi interfésze lehet, innentől kezdve nehéz a lehetőségeket kordában tartani. Ha viszont minden template követelménye egyedi, akkor nehéz olyan típusokat definiálni, melyek sok template-be behelyettesíthetőek.

Koncepciók tervezése

- Célunk ezért egy minimális számú *koncepció* kidolgozása, melyek *plug-compatible* tulajdonságúak.
- Vegyünk ismét egy példát: a `String` template-et. Mit várunk el `X` típustól ahhoz, hogy argumentumként szerepelhessen `String:String<X>`-ben? Milyen feltételeknek kell megfelelnie `X`-nek ahhoz, hogy karakteréként tekintsünk rá?

```
template<typename C>
class String{
  // ...
};
```

- Gyakorlott programozóknak lehet, hogy kisszámú halmazból kerülne ki egy-egy gyorsan adott válaszuk.

Koncepciók tervezése

- Mi viszont próbáljunk meg messzebbről kiindulni. A tervezést három lépésre bonthatjuk:
  - [1] Nézzünk bele az eredeti implementációba és határozzuk meg, milyen tulajdonságait (operátorok, fv-ek, tagváltozók, stb.) használjuk fel a template paraméter-típusnak!
  - [2] Ezt követően vizsgáljunk meg néhány hihető alternatív megvalósítást, és soroljuk fel ezeknek a template paraméter-típussal szemben támasztott követelményeit! Ezáltal esetleg rájöhetünk arra, hogy tágabb vagy szűkebb lehetőségeket kell(ene) inkább lehetővé tenni a template-típusok számára.
  - [3] Fentiek alapján hasonlítsuk össze a megkövetelt tulajdonságokat olyan koncepciókkal, melyeket más template-ek esetén felhasználtunk. Olyan egyedi koncepciókat keresünk, melyek segítségével ez a mostani lista lerövidíthető.

Koncepciók tervezése

- Az első két pont érthető okokból szorosan összefügg az algoritmus-általánosítás módszerével. A harmadik pont viszont azt a célt szolgálja, hogy ne szaladjon el velünk a ló: ne tágítsuk értelmetlenül az alkalmazhatóságot, mert sok lesz a koncepció
- Visszatérve a `String` osztályra, a szükséges műveletek kb. ezek:
  1. `C` típust copy assignmenttel és copy inicializálással másoljuk.
  2. `String` a `C` típusú változókat az == operátorral és `!=` operátorral hasonlítja össze
  3. `String` `C` típusú tömböket hoz létre (ez megmutatja azt is, mi a default konstruktor feladata)
  4. `String` `C` típusú objektumok címeit várja
  5. Amikor `String`-et megszűntetünk, a mögötte álló `C` típusú objektumokat is megszűntetjük
  6. `String` rendelkezik « és » operátorokkal `C` típusú objektumok olvasására és írására.

Koncepciók tervezése

- A 4. és 5. követelmény nem igényel magyarázatot, általában feltételezzük őket adattípusok esetén.
- Az 1. követelmény néhány olyan fontos típusra nem áll fenn (pl. `std::unique_ptr`-re), amelyek valódi erőforrásokat reprezentálnak. Legtöbb hétköznapi típusra azonban fennáll, ezért itt is megköveteljük. Ezzel szorosan összefügg a 2. pont: ha egy típust másolunk, azt szeretnénk ha ugyanúgy viselkedne, mint az eredeti (kivéve persze hogy más címen helyezkedik el).
- Azáltal, hogy a copy assignmentet is megköveteljük, az is világos hogy konstans típusokat nem használhatunk template argumentumként. Pl. `String<const char>` típust nem példányosíthatunk, mert ez nem lehetne copy assignolható.

Koncepciók tervezése

- A copy assignment megteremtése viszont azt is jelenti, hogy az algoritmus felhasználhatja a típus temp változóit, készíthet belőle konténereket, stb. Továbbá a const interfészek létrehozását sem zárja ki (az csak ígéret, hogy a fv nem nyúl bele a paraméterbe):

```
template<typename C>
bool operator==(const String<C>& s1, const String<C> &s2)
{
  if (s1.size()!= s2.size()) return false;
  for (auto i = 0; i!=s1.size(); i++){
    if (s1[i]!=s2[i]) return false;
  }
  return true;
}
```

- Mi a helyzet a mozgatással? String<C> típusra definiálhatunk move operátorokat, C-re nem kell (ha pl. fv-ből adjuk vissza, a rendszer implicit módon úgyis move-ot használ.)

Koncepciók tervezése

- Ha viszont `String<C>`-re van move, sok érdekes esetet eleve lefedünk. Pl. a `String<String<char>>` típus kezelése hatékony lesz
- Mi a helyzet a 6. követelménnyel? Nem felesleges ez? Pl. lehetne úgy is, hogy csak akkor legyen kiírható egy `String<C>`, ha olvasható is. Ez nem mindegy: ha megtartjuk a követelményt, az egész `String` template osztályra vonatkozik a követelmény. Ha finomítjuk, akkor külön template-elt fv-eken keresztül a `C` típusra vonatkozik.
- Összefoglalva: mi az ami hiányzik? Nincs pl. rendezés (pl. `<` operátor), és nincs egész típusra való konverzió sem.
- Végül nézzük: milyen jól ismert koncepcióra emlékeztet mindez? Nevezhetjük *reguláris* koncepciónak.

Koncepciók tervezése

- A *reguláris* típus olyan típus, melyet:
  - másolhatunk (assignmenttel vagy inicializálással) a helyes copy szemantikával
  - alapértelmezett konstruktorral példányosíthatunk
  - felhasználhatunk alapértelmezett műveletek operandusaként (pl. lekérdezhetjük a címét)
  - komparálhatunk a == és != operátorokkal
- Az utolsó kérdés, hogy mennyire fontos, de ha már másolni lehet, nem árt ha komparálni is tudunk.
- Ami viszont kimaradt az a rendezés. Pedig ez fontos: ha sztringekre gondolunk, simán eszünkbe juthat a sorrendezhetőség igénye. Ezért inkább használjuk az *rendezett* koncepciót!

Koncepciók tervezése

- Ezért a korábbi feltételeket megtoldjuk azzal, hogy `String<C>`-ben:
  - `C` legyen rendezhető
  - Legyenek « és » operátorok `C`-re akkor és csak akkor, ha `String<C>` ezen két operátorát is fel akarjuk használni
  - Legyen egész típusra konverzió akkor és csak akkor, ha `C` típusra is létezik ilyen
- Mostmár csak annyi a gond, hogy mindennek fordításidejű garantálásához nincs nyelvi funkció (jelenleg!). Mi azt szeretnénk, ha lehetne ilyet mondani (sajnos nem lehet):

```
template<Ordered C>
class String{
  // ...
}
```

Koncepciók betarttatása

- Ezért kerülőúton fogalmazhatjuk meg a koncepciókra vonatkozó előírásokat, fordításidőben kiértékelhető (constexpr) mechanizmusok útján. Például:

```
template<typename C>
class String{
  static_assert(Ordered<C>(), "String's character type is not ordered");
  // ...
};

template<typename C>
constexpr bool Ordered(){
  return Regular<T>() && Totally_ordered<T>();
}

template<typename T>
constexpr bool Totally_ordered(){
  return Equality_comparable<T>() && Has_less<T>() && Boolean<Less_result<T>>() &&
  Has_greater<T>() && Boolean<Greater_result<T>>() && Has_less_equal<T>() &&
  Boolean<Less_equal__result<T>>() && Has_greater_equal<T>() &&
  Boolean<Greater_equal_result<T>>();
} //es igy tovabb...
```

Koncepciók betarttatása

- A láncolat végén axiómák is találhatóak (pl. copy szemantika garantálására, stb.)
- Ennek a módszernek több hátulütője van:
  - A követelményeket definíciókban ellenőrizzük, pedig deklarációkban kéne
  - Nincs garancia arra, hogy nem felejtük el az ellenőrzést.
  - Nincs az ellenőrzés pontjának kötelezően meghatározott helye.
  - A fordító nem tudja ellenőrizni, hogy a template megvalósítása tényleg csak az adott koncepciónak megfelelő tulajdonságokat használja ki
  - Az ellenőrzés eredményét sem érti a fordító (csak az emberi olvasó)
  - A helyzetet tovább bonyolítja, hogy lehetnek többparaméteres koncepcióink is. Ezért nem mindig igaz az, hogy a koncepció egy adott *típus típusa* (pl. mi történik, ha kapcsolatot szeretnénk paraméterek között garantálni, pl. `Iter find(Iter b, Iter e, Val x)` fv-ben).

További témák – I: felhasználó általi specializáció

- Eddig olyan eseteket vettünk, melyekben bármilyen típus esetén ugyanazt a kódot generálta a template (a felhasznált típust leszámítva).
- Mi történik, ha azt szeretnénk mondani, hogy bizonyos típus esetén másképpen szeretnénk ha az implementáció működne? Pl. "*ha a paraméter pointer, használjuk A implementációt; ha nem az, használjuk B implementációt*", vagy "*ha a paraméter nem My_Base osztályból származik, dobjunk kivételt*". Ilyenkor **felhasználó általi specializációra** van szükség.
- Általában a specializáció automatikusan történik... de mi könnyen felülírhatjuk. A specializált változatban a template fejlécben üresen hagyjuk a kacsacsőröket, és az implementációban kitöltjük a típust. Később származtathatunk is belőle. Példa:

További témák – I: felhasználó általi specializáció

```
template<typename T>
class Vector{//altalanos vektortipus
  T* v;
  int sz;
public:
  Vector();
  explicit Vector(int);
  T& operator[](int i);
  //...
};

Vector<int> vi;
Vector<char*> vpc;
Vector<Node*> vpn;
```

- Általában valamilyen pointereket tartalmazó vektort célszerű példányosítani, hogy a futásidejű polimorfizmust kihasználjuk.
- Nem annyira jó, h eltérő típusú pointerekre majdnem ugyanazt a kódot többször legenerálja a fordító. Az lenne a célszerű, ha az összes pointer típus egyazon implementációt használná.

További témák – I: felhasználó általi specializáció

```
//eloszor definialunk egy specializalt valtozatot void pointereket tartalmazo kontenerekre
template<>
class Vector<void*>{
  void** p;
  void*& operator[](int i);
  //...
};
```

- A kacsacsőrök üresek, ami azt jelenti hogy a specializáció template paraméter nélkül meghatározható. Persze amikor meghívjuk, azt kell írni pl. hogy `Vector<void*> vpv;`
- A példa *teljes specializáció*, mert semmilyen típust sem kell megadni vagy kikövetkeztetni a kód belsejében.
- Ugyanez történt korábban az exact_int-es példában.

További témák – I: felhasználó általi specializáció

```
//ezek utan jojjon a pointer Vector-ok implementacioja (kizarolag pointereket tartalmazo vektorokra)
template<typename T>
class Vector<T*> : private Vector<void*>{//reszleges specializacio
public:
  using Base = Vector<void*>;
  Vector(){}
  explicit Vector(int i): Base(i){}
  T*& operator[](int i) {return reinterpret_cast<T*&>(Base::operator[](i));}
  //...
};
```

- A fenti kód *részleges specializáció*, mert a void típusú pointerekre létrehozott Vector template-ből származik, ezért csak pointer típusokra lesz alkalmazható.
- A `Vector<T*>` osztály csupán interfész a `Vector<void*>` osztályhoz.
- Enélkül a megoldás nélkül még átlagos méretű programok esetében is több Mb-nyi kódnövekedést eredményezne a sok példányosítás.

További témák – I: felhasználó általi specializáció

- Persze ez nem jelenti azt, hogy `Vector<T*>` osztályt ne kellene minden `T` típusra újra példányosítani. Az viszont biztos, hogy ennek a költsége kisebb lesz: csak származtatás és inline-olás szerepel benne. Az implementáció a közös, `Vector<void*>` osztályban kap helyet.
- Megjegyzés: `Vector<void*>` osztályt nem bonyolult implementálni, mivel az osztálybeli tömb void pointereket tartalmaz.
  - ezek mérete pedig fix (az architektúra címhosszához lesz köze, nem ahhoz, hogy mi a konkrét `T` típus!)
  - így nem nehéz pl. megkeresni az i. pointert a tömbben.

További témák – II: Template-ek és öröklés

- A template-ek és a származtatás is azt a célt szolgálják, hogy:
  - létező típusokból új típusokat hozzunk létre
  - interfészeket specifikáljunk
  - lehetővé tehessük, hogy több kódrészletben megragadjunk valami közöset
- Pontosabban: template osztállyal interfészt definiálhatunk.
  - A template megvalósítása, illetve a specializációinak megvalósítása ezen interfészen keresztül érhetőek el.
  - A megvalósítás kódja template osztályban minden paraméter-típus esetén ugyanaz.
  - A különböző specializációk implementációja viszont nagyban el is térhet (ld. korábbi példa), azonban az elsődleges template szemantikáját kell nekik is megvalósítaniuk.
  - A specializáció funkcióban is bővítheti az elsődleges template-et.

További témák – II: Template-ek és öröklés

- Szülőosztályok segítségével is interfészt definiálhatunk.
  - Az osztály implementációja és a származtatott osztályok implementációja egyazon interfészen keresztül érhetőek el.
  - A különböző származtatott osztályok implementációi nagyban eltérhetnek, de a szülőosztály szemantikáját kell nekik is megvalósítaniuk.
  - A származtatott osztály funkcióban is bővítheti a szülőosztályt.
- Tehát: a template-ekkel és a származtatással sok szempontból hasonló eredményt érhetünk el! Ezért a két technikát érdemes közös névvel illetni.
- Mindkét technika az ún. *polimorfizmus* példája. A kettő megkülönböztetésére a **fordításidejű polimorfizmus** és **parametrikus polimorfizmus**, illetve a **futásidejű polimorfizmus** kifejezéseket használjuk.

További témák – II. Template-ek és öröklés

- A kétfajta polimorfizmus közötti különbség (célok tekintetében) gyakran árnyalatnyi.
- Gyakran keveredik is a kettő. Pl. `vector<Shape*>` osztály parametrikus polimorfizmust használó konténer, amely futásidőben polimorfikus objektumokat tartalmaz.
- Fontos, hogy mindkét technikát elsajátítsuk.

Template-ek és öröklés

- Mikor használjunk futásidejű, és mikor parametrikus polimorfizmust?
- Nézzünk egy lebutított és absztrakt példát:

```
template<typename X>
Class Ct{
  X mem;
public:
  X f();
  int g();
  void h(X);
};

template<>
class Ct<A>{ //specializacio A tipusra
  A* mem; //a reprezentacio lehet mas, mint az elsodleges template-ben!
public:
  A f(); int g();
  void h(A);
  void k(int); //bovebb funkcionalitas
};

Ct<A> cta; Ct<B> ctb; //ketfele specializacio
```

Template-ek és öröklés

- Ugyanez (kb.) osztály-hierarchia útján:

```
class Cx{
  X mem;
public:
  virtual X& f();
  virtual int g();
  virtual void h(X&);
};

class Da: public Cx{//szarmaztatott osztaly
public:
  X& f(); int g(); void h(X&);
};

class Db: public Cx{
  B* p; //bovebb reprezentacio
public:
  X& f(); int g(); void h(X&);
  void k(int); //bovebb funkcionalitas
};

Cx& cxa {*new Da};
Cx& cxb {*new Db}; //nem eletszeru, viszont jelezzuk: fut.ideju pol.-hoz mindig ptr vagy ref kell
```

Template-ek és öröklés

- Mindkét esetben olyan objektumokat használjunk, amelyek közös műveletekkel rendelkeznek. Megfigyelhetjük azonban, hogy:
  - Ha az interfészben és származtatott osztályban használt típus különbözik, hatékonyabb a template
  - Ha az interfészben és származtatott osztályban levő implementációk csak egyetlen paraméterben, vagy nagyon kevés, speciális esetben eltérőek, hatékonyabb a template.
  - Ha az objektumok tényleges fajtái fordításidőben nem ismertek, csak osztályhierarchia jó.
  - Ha fontos, hogy a típusok között hierarchikus viszony legyen, kellenek a hierarchiák. A szülőosztály közös interfészt definiál. Template specializációk esetén a konverziókat a programozónak kell megadni! (ld. korábban: `vector<Circle>` nem egyfajta `vector<Shape>`).
  - Ha nem szeretnénk dinamikus mem.allokációkra támaszkodni, jobb a template.
  - Ha futásidejű hatékonyság kell (pl. inline), a template sokkal jobb.

Template-ek és öröklés

- Nem könnyű feladat elérni, hogy a szülőosztályok kicsik legyenek és típus-biztonságosak. Nehéz garantálni pl., hogy az interfészben meghatározott típusok ne változzanak a származtatott osztályokban.
- Gyakran kötünk olyan kompromisszumokat, melyekben vagy:
  - túlságosan behatároljuk az interfészt ("gazdag" interfészt csinálunk, melyet minden származtatott osztálynak implementálnia kell)
  - túlságosan tágra csináljuk meg az interfészt (pl. `void*` vagy `Object*` típusokat használunk)
- A kettő közötti egyensúly elérése gyakran csak template-ekkel ésszerű.
- Pl. szülőosztályra mutató pointer felhasználható template argumentumként (futásidejű polimorfizmus), template paraméter felhasználható szülőosztály-interfész specifikálásához (típusbiztonság)

Template-ek és öröklés

- Fontos odafigyelni, hogy mikor használjunk template-et egy egész osztály, és mikor csak egy-egy függvény keretében.
- Ha pl. van egy renderelhető alakzatunk, gondolhatjuk, hogy az egész osztályt paraméterezzük:

```
template<typename Color_scheme, typename Canvas> //megkerdojelezheto
class Shape{ //...};

template<typename Color_scheme, typename Canvas> //megkerdojelezheto
class Circle: public Shape{
  //...
};

template<typename Color_scheme, typename Canvas> //megkerdojelezheto
class Triangle: public Shape{
  //...
};

void user(){
  auto p = new Triangle<RGB, Bitmapped>({0,0},{0,60},{30,20});
}
```

Template-ek és öröklés

- Mi ezzel a probléma? Ha csak egy *Color_scheme-Canvas* kombinációt használunk, a generált kód majdnem akkora lesz, mint template nélkül (azért majdnem, mert használattól függően lehet hogy némelyik nem-virtuális fv-t sosem használjuk.
- Általában azonban a Color_scheme és Canvas osztálynak rengeteg-féle származtatott variánsa lehet. Ilyenkor *N* kombinációhoz *N*-szer generálunk le minden virtuális fv-t, mert a fordító nem tudhatja, hogy futásidőben melyiket fogjuk meghívni! Általában jobb ötlet:

```
class Shape{
  template<typename Color_scheme, typename Canvas>
  void configure(const Color_scheme&, const Canvas&);

  //...
};
```

Template-ek és öröklés

- Ettől függetlenül persze más kérdés, hogyan osszunk meg különböző osztályok és objektumok mögött konfigurációra vonatkozó, `Color_scheme`-től és `Canvas`-tól függő információkat.
- Egy lehetőség, hogy a `configure()` fv lefordítja valamilyen szabványos típusú értékekre a konkrét objektumokból kinyert információt.
- A másik lehetőség, hogy `Shape` osztályba belefoglalunk egy `Configuration*` pointert, ahol `Configuration` a konfigurációs interfészt definiáló szülőosztály.

Template-ek és öröklés – template paraméterek mint szülőosztályok

- Klasszikus osztályhierarchia esetében a különböző implementációkban eltérő dolgokat a származtatott osztályokban helyezzük el, a szükséges implementációt pedig a szülőosztály virtuális fv-ein keresztül hívjuk meg.
- Ez a módszer azonban nem teszi lehetővé, hogy:
  - Az interfészben használt típusokat változtassuk
  - Hatékonyan (pl. inline-olva) hívjuk meg a virtuális fv-t
- Ezeken a problémákon segíthet, ha a specializált információt és műveleteket template argumentumok útján adjuk át a szülőosztálynak.

Template paraméterek mint szülőosztályok

- Tfh kiegyenlített bináris fát szeretnénk implementálni. Mivel rugalmas felhasználhatóságot szeretnénk, nem kódolhatjuk bele a fa csomópontjait reprezentáló osztályba a felhasználó típusát. Több lehetőségünk van:
  - Legyenek a felhasználói adatok egy származtatott osztályban, melyet virtuális fv-eken keresztül elérhetünk. Ennek hátulütője, hogy a virtuális fv-hívások költségesek (viszonylag), és az interfész nem ismeri a konkrét adattípust, tehát sokat kell kasztolni.
  - Legyen `void*` a csomópontokban, ami bármire mutathat. Ez azonban megkétszerezheti a szükséges allokációk számát (kell egy `UserData*` és egy `void*` is, továbbá kasztolnunk is gyakran kellene, ráadásul nem típusellenőrzött módon!
  - Lehet egy `Data` osztály, ami minden adatstruktúra szülője, és használhatunk `Data*` típust. Ez az előző két módszer kombinációja, továbbra sem jó.

Template paraméterek mint szülőosztályok

- Vannak további lehetőségek is. Nézzük pl. ezt:

```
template<typename N>
struct Node_base{
  N* left_child
  N* right_child

  Node_base();
  void add_left(N* p){
    if (left_child == nullptr){
      left_child = p;
    }else{
      //...
    }
  }
  //...
};

template<typename Val>
struct Node: Node_base<Node<Val>>{ //hasznaljuk fel a szarmaztatott osztalyt sajat szulojeben
  Val v;
  Node(Val vv);
  //...
};
```

Template paraméterek mint szülőosztályok

- Ezáltal a `Node_base` osztály annak ellenére tudja felhasználni `Node<Val>`-t, hogy nem is ismeri ezt a típust!
- Vegyük észre, hogy egy `Node` reprezentációja igen tömör lesz. Pl. egy `Node<double>` így nézne ki:

```
struct Node_base_double{
  double val;
  Node_base_double* left_child;
  Node_base_double* right_child;
};
```

- Sajnos ebben az esetben a felhasználónak ismernie kell `Node_base` osztály műveleteit és a létrehozott fa struktúráját is. Pl.

Template paraméterek mint szülőosztályok

```
using MyNode = Node<double>;

void user(const vector<double>& v){
  MyNode root;
  int i = 0;
  for(auto x : v){
    auto p = new MyNode(x);
    if(i++%2)
      root.add_left(p);
    else
      root.add_right(p);
  }
}
```

- Ezért sem kényelmes, ha a felhasználónak kell garantálnia a fa kiegyensúlyozását. Ezt szeretnénk, ha a fa maga oldaná meg. Ehhez azonban a fának hozzá kellene férnie a felhasználói adatok értékéhez!
- Egy lehetséges megoldás, hogy kikötjük, hogy a `Node<Val>` típusnak kell, hogy legyen egy kisebb-mint operátora. Ekkor:

Template paraméterek mint szülőosztályok

```
//...

void insert(N& n){
  if (n<left_child)
    //...
  else
    //...
}
```

- Ez jól működik. Minél több mindent feltételezünk a felhasználói típusról, annál könnyebben meg tudjuk írni a megvalósítást. Extrém esetben értéket is átadhatunk a `Node_base` osztálynak (ezt a megoldást használja az `std::map` is).
- Az eredeti kérdést azonban nem válaszoltuk meg. Mi van akkor, ha a felhasználó szeretne `Node` objektumokat manipulálni is (pl. áthelyezni egy másik fába)? Ekkor nem elegendő egy anonim node típust használnunk.

Template paraméterek mint szülőosztályok

- Kézenfekvő megoldás: `Node` típus kombináljon össze egy érték típust és egy "kiegyensúlyozó" típust:

```
template<typename Val, typename Balance>
struct Search_node : public Node_base<Search_node<Val, Balance>, Balance>{
  Val val;//felhasznaloi adatok
  Search_node(Val v):val(v){}
};
```

- `Balance` kétszer szerepel, mert része a `Search_node` osztálynak, és mert a szülőnek példányosítania is kell egy ilyen objektumot.

```
template<typename N, typename Balance>
struct Node_base : Balance {
  //...
  void insert(N& n){
    if(this->compare(n,left_child)) // compare() Balance osztalybol...
      //...
    else
      //...
  }
}
```

Template paraméterek mint szülőosztályok

- Lehetett volna úgy is, hogy Balance osztály alapján tagváltozót hozunk létre; ez általában azonban pazarlóbb lenne. Ha pl. nincs Balance osztálynak semmilyen tagváltozója, akkor származtatott osztály példányosítása esetén nincs extra memória-foglalás (*empty-base optimization*). Továbbá ez a megoldás hasonlít ma már kvázi-szabványos framework-ökhöz. Felhasználható pl. így:

```
struct Red_black_balance{
  // adatok es muveletek amik kellenek piros-fekete fahoz
};

template<typename T>
using Rbnode = Search_node<T, Red_black_balance>;

Rbnode<double> myRoot; //piros-fekete fa double-okkal
using Mynode = Rbnode<double>;

void user(const vector<double>& v){
  for (auto x: v)
    myRoot.insert(*new Mynode(x));
}
```

Template paraméterek mint szülőosztályok

- A node struktúra így kompakt, és minden olyan fv-t, aminek hatékonynak kell lennie, inline-olhatunk. Tehát elértük, hogy típus-biztonságos megoldásunk legyen, miközben a komponálható egyszerűségét is fenntartottuk.
- Ha a megoldást mégis túl szószátyárnak tartjuk, explicit template paraméter helyett azt is megtehetjük, hogy a Node objektum mondja meg, hogy mi a Balancer típusunk (teljesen mindegy):

```
template<typename N>
struct Node_base : N::balance_type{
  //...
};
```

Template metaprogramozás – bevezetés

- Eddig a generikus programozást abban a kontextusban vettük, hogy általános algoritmusokat készítsünk
- A metaprogramozás ezzel szemben arról szól, hogy fordításidőben generálunk programokat
  - Azért metaprogramozás, mert típusokon végzünk számításokat, nem feltétlenül értékeken
- Mire lehet jó a metaprogramozás?
  - Javított típusbiztonság – kiszámíthatóak a pontos típusok, amik kellenek, így sok esetben elkerülhető az explicit típuskonverzió
  - Javított futásidejű teljesítmény: több helyen inline-olhatunk, és több helyen fordításidőben is kiszámíthatjuk azokat a típusokat, amikre szükségünk van.

Bevezetés

- A metaprogramozásnak több szintje különböztethető meg:
  - Nincs fordításidejű számítás (csak típus és érték-argumentumok adhatók át)
  - Egyszerű komputáció (fordításidejű tesztek és iteráció nélkül)
  - Explicit fordításidejű komputáció (pl. compile-time `if`)
  - Fordításidejű iterációk (rekurzió alapján)
- De a metaprogramozás tágabb témakör, mint a *template* metaprogramozás, mert működhet pl. `constexpr` útján is.
- Megintcsak érdemes odafigyelni, hogy a súlykot ne vessük el: mértékkel jó lehet a metaprogramozás, de túlzott használata nehezen áttekinthető kódot eredményez.
- Megjegyzés: a template-ek bizonyítottan Turing-teljesek. Pl. Lisp-interpretert írtak már template-ekkel. Ezzel együtt körülményes: tisztán funkcionális programozást tesz lehetővé.

Típusfüggvények

- Típusfüggvényeknek nevezzük azokat a fv-eket, melyeknek legalább egy paramétere típus, és/vagy visszatérési értéke típus.
- Nem feltétlenül kell, hogy úgy nézzenek ki mint a fv-ek általában. Sőt, legtöbbször igen más a szintaxisuk. Pl. a standard könyvtár `is_polymorphic<T>` olyan típus, melyet ha példányosítunk akkor annak `value` adattagja adja meg a kérdésre a választ:

```
if (is_polymorphic<int>::value) cout << "Micsoda meglepetes!"
```

- Hasonló módon az a konvenció: ha egy típust szeretnénk visszakapni, azt a `type` adattagon keresztül olvashatjuk ki:

```
enum class Axis: char{x, y, z};
enum flags{off, x=1, y=x<<1, z = x<<2};

typename std::underlying_type<Axis>::type x; //x tipusa char
typename std::underlying_type<flags>::type y; //y valoszinuleg int
```

Típusfüggvények – II.

- Típusfüggvény több argumentumot kaphat és több visszatérési értéke is lehet (hiszen ezek adattagok). Például:

```
template<typename T, int N>
struct Array_type{
  using type = T;
  static const int dim = N;
  //...
};

using Array = Array_type<int, 3>;
Array::type x; //x tipusa int
constexpr int s = Array::dim; //s erteke 3
```

- A típusfüggvények fordításidejű fv-ek, vagyis csak olyan argumentumuk lehet (legyenek akár típusok, akár értékek) amik fordításidőben ismertek és olyan visszatérési értékeik vannak, amik fordításidőben felhasználhatók.

Típusfüggvények – III.

- Legtöbb típusfüggvény legalább egy típus-argumentumot vár, de vannak hasznosak, amik nem. Például itt egy típusfüggvény ami olyan egész típust ad vissza, aminek megfelelő a bájtszáma:

```
template<int N>
struct Integer{
  using Error = void;
  using type = Select<N, Error, signed char, signed char, short, unsigned int, signed int, unsigned
        long, signed long, long long>;
};

typename Integer<4>::type i4 = 8; //unsigned int
typename Integer<1>::type i1 = 9; //signed char
```

- Select annyit tesz, hogy kiválasztja az N-edik elemet a listából.

Típusfüggvények – IV.

- Tehát legtöbb típusfüggvény template. Általános számításokra használhatók, típusok és értékek kapcsán is.
- Igen hasznos dolgokra használhatók. Még egy példa: típus méretétől függően foglaljunk stacken vagy halmon területet:

```
constexpr int on_stack_max = sizeof(std::string);
struct Obj_hold {
  using type = typename std::conditional<(sizeof(T) <= on_stack_max),
      Scoped<T>,
      On_heap<T>
    >::type;
};
```

- `conditional` fordításidőben tud két alternatíva között választani

Típusfüggvények – V.

- `Obj_holder` így használható:

```
void f(){
  typename Obj_holder<double>::type v1; //stack
  typename Obj_holder<array<double,200>>::type v2; //halom
  //...
  *v1 = 7.7;
  v2[77] = 9.9;
}
```

- Ez nem is olyan hajánál fogva előragadott példa, mint hihetnénk. A C++ szabvány leírása pl. megjegyzi a `function` típus definícióját követően, hogy: “Implementations are encouraged to avoid the use of dynamically allocated memory for small callable objects, for example, where f’s target is an object holding only a pointer or reference to an object and a member function pointer”

Típusfüggvények – VI.

- Hogyan implementálhatjuk Scoped-ot és On_heap-et? Nem bonyolult, és nincs szükség metaprogramozásra:

```
template<typename T>
struct On_heap{
  On_heap(): p(new T){} // allokacio
  ~On_heap(){delete p;} //deallokacio

  T& operator*(){return *p;}
  T* operator->(){return p;}

  On_heap(const On_heap&) = delete; //masolast nem engedelyezunk
  On_heap operator=(const On_heap&) = delete;
private:
  T* p; //mutato az objektumra a halmon
};
```

Típusfüggvények – VII.

- Scoped pedig:

```
template<typename T>
struct Scoped{
  T& operator*(){return x;}
  T* operator->(){return &x;}

  Scoped(const Scoped&) = delete; //itt sem masolunk
  Scoped operator=(const Scoped&) = delete;
private:
  T x; //az objektum
};
```

Egy kényelmi módosítás

- Bosszantó, hogy mindig ilyen hosszú kódot kell írnunk a típusok lekérdezéséhez. Egy kényelmi rövidítés:

```
template<typename T>
using Holder = typename Obj_holder<T>::type;

void f2(){
  Holder<double> v1;
  Holder<array<double, 200>> v2;

  *v1 = 7.7;
  v2[77] = 9.9;
}
```

Típuspredikátumok

- Ha olyan fv-eket szeretnénk készíteni, amik típusokat fogadnak el argumentumként, szükségünk lehet arra is hogy az argumentumok típusairól is időnként feltegyünk kérdéseket. Pl.: “ez előjeles típus”? “Ez polimorfikus típus (van legalább egy virtuális fv-e)”?
- Sok ilyen kérdésre a fordító tudja a választ, és standard könyvtárbeli fv-eken keresztül lehívhatjuk az információkat. Pl.:

```
template<typename T>
void copy(T* p, const T* q, int n){
  if(std::is_pod<T>::value)
    memcpy(p,q,n); //optimalizalt masolas akkor, ha plain old data-rol van szo
  else
    for(int i=0; i!=n; ++i)
      p[i] = q[i]; //masolas elemenkent
}
```

Függvény kiválasztása

- Elég bonyolult szabályok írják le, hogy valami mitől lesz POD.
  - Lényegében konstruktor, destruktor és virtuális fv nélküli osztályt vagy structot jelentenek.
- A függvény is típussal rendelkező objektum, ezért hasonló technikákkal függvényeket is kiválaszthatunk:

```
struct X{//irjuk ki X-et
  void operator()(int x){cout << "X&" << x << "!";}
  //...
};

struct Y{//irjuk ki Y-t
  void operator()(int y){cout << "Y" << y << "!";}
  //...
};

void f(){
  Conditional<(sizeof(int)>4),X,Y>{}(7); //csinaljunk egy X-et vagy Y-t es hivjuk meg
  using Z = Conditional<(Is_polymorphic<X>()),X,Y>;
  Z zz;
  zz(7);
}
```

Traits (jellemvonások)

- A standard könyvtár nagyban támaszkodik az ún. traitekre (jellemvonásokra), hogy típusokat tulajonságokkal össze tudjunk kötni. Pl. az iterátor típusok rendelkeznek az ún. iterátor-traitekkel:

```
template<typename Iterator>
struct iterator_traits{
  using difference_type = typename Iterator::difference_type;
  using value_type = typename Iterator::value_type;
  using pointer = typename Iterator::pointer;
  using reference = typename Iterator::reference;
  using iterator_category = typename Iterator::iterator_category;
};
```

- A traitek sokeredményes típusfüggényként, vagy típusfüggvények gyűjteményeként használhatóak. Például ha egy pointerre megvannak az `iterator_traitek`, akkor beszélhetünk annak `value_type` és `difference_type` tagjáról (annak ellenére, hogy pointernek nincsenek tagjai).

Traits (jellemvonások) - II.

```
template<typename Iter>
Iter search(Iter p, Iter q, typename iterator_traits<Iter>::value_type val){
  typename iterator_traits<Iter>::difference_type m = q - p;
}
```

- Itt `difference_type` arra a típusra utal, amit akkor kapunk, ha két iterátort kivonunk egymásból.

Programfutást irányító kontrollstruktúrák

- További lehetőségek rejlenek például a `Conditional` és `Select` template-ekben. A korábbi két típus közötti választást tesz lehetővé, az utóbbi pedig többféle típus közötti választást tesz lehetővé.
- Ha értékek között kell választanunk, elég egy sima feltételes kifejezés (`?:`) – ezek a template-ek viszont típusokat adnak vissza. Felmerülhet továbbá, hogy miért ne lehetne futásidejű `if`-et használni. Azért, mert van egy szabály, hogy feltételes kifejezés egyik ága sem tartalmazhat kizárólag deklarációt:

```
struct Square{
  constexpr int operator()(int i){return i*i;}
};

struct Cube{
  constexpr int operator()(int i){return i*i*i;}
};
```

Programfutást irányító kontrollstruktúrák / iteráció, rekurzió

```
if(My_cond<T>())
  using Type = Square; //hiba: ez egy deklaracio
else
  using Type = Circle; //hiba: ez egy deklaracio

Conditional<My_cond<T>(),Square,Cube>{}(99); //marad ez...
```

- Template nyelvben létezik iteráció és rekurzió is. Például:

```
template<int N>
constexpr int fac(){
  return N*fac<N-1>();
}

template<>
constexpr int fac<1>(){
  return 1;
}

constexpr int x5 = fac<5>();
```

Iteráció és rekurzió

- Mivel a template nyelv tisztán funkcionális, változók nem hozhatók létre. Ezért ha iterációt szeretnénk, marad a rekurzió, template specializációval.
- Fenti esetben viszont használhatunk `constexpr` függvényt is. Ez viszont nemcsak fordításidőben használható:

```
constexpr int fac(int i){
  return (i<2)?1:fac(i-1);
}

constexpr x6 = fac(6);
```

Iteráció és rekurzió

- Rekurzió működhet osztályokon keresztül is:

```
template<int N>
struct Fac{
  static const int value = N*Fac<N-1>::value;
};

template<>
struct Fac<1>{
  static const int value = 1;
};

constexpr int x7 = Fac<7>::value;
```

- Nem túl realisztikus, de léteznek hasznos példák is.

Mikor használjunk metaprogramozást?

- Amit eddig átnéztünk, ez alapján bármi kiszámítható fordításidőben (csak a rekurzió mélysége szab korlátot). De továbbra is kérdés: miért akarnánk metaprogramozni?
- Csak akkor, ha tisztább, jobban teljesítő és/vagy jobban karbantartható kódot kapunk.
- Nem árt, ha a template programozást `constexpr` függvények mögé rejtjük, ahol csak tehetjük.