

# C++ Gyakorlat jegyzet 7. óra.

A jegyzetet UMANN Kristóf készítette HORVÁTH Gábor gyakorlatán. (2017. május 6.)

## 1. Template

### 1.1. Függvény template-ek

Térjünk vissza a régebben megírt swap függvényünkhöz.

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

Ahogy azt láttuk, túl tudjuk terhelni ezt a függvényt, hogy más típusú objektumokat is meg tudjunk cserélni. Azonban gyorsan megállapítható, hogy állandóan egy újabb overloadot létrehozni nem épp ideális megoldás. Ez a kisebb gond, a nagyobb az, hogy a kódsímtelés áldozatai leszünk: ha bármi miatt megváltozna a `swap` belső implementációja (pl. találunk hatékonyabb megoldást), az összes létező swap függvényben meg kéne ejteni a változtatást. E probléma elkerülésére egy megoldás lehet, ha létrehozunk egy sablont, melynek mintájára a fordító maga tud generálni egy megfelelő függvényt.

```
template <typename T>
void swap(T &a, T &b)
{
    T tmp = a;
    a = b;
    b = tmp;
}
```

Az így implementált `swap` függvény egy *template*, és a template paramétere (`T`) egy típus. Ez alapján a fordító már létre tud hozni megfelelő függvényeket:

```
int main()
{
    int a = 2, b = 3;
    swap<int>(a, b);

    double c = 1.3, d = 7.8;
    swap<double>(c, d);
}
```

A fordítónak csak annyi dolga van, hogy minden `T`-t lecseréljen `int`-re, és már kész is a függvény. A fenti példában mi explicit megmondtuk a fordítónak, hogy `swap`-ot milyen template paraméterrel példányosítsa (*instantiate*), azonban függvényeknél erre nem feltétlenül van szükség: a fordító tudja `a` és `b` típusát, így ki tudja találni hogy mit kell behelyettesítenie.

```
int main()
{
    int a = 2, b = 3;
    swap(a, b);

    double c = 1.3, d = 7.8;
    swap(c, d);
}
```

Ezt a folyamatot (amikor a fordító kitalálja a template paramétert) **template paraméter dedukciónak** (*template parameter deduction*) hívjuk.

**1.1.1. Megjegyzés.** Természetesen a példányosítás jóval bonyolultabb annál, hogy a fordító minden T-t egy konkrét típusra cserél, de ebben a könyvben az egyszerűség kedvéért elégedjünk meg ennyivel.

Nem csak típus lehet template paraméter – bármi ami **nem** karakterlánc literál vagy lebegőpontos szám.

**1.1.2. Megjegyzés.** A lebegőpontos számokra vonatkozó indoklást később a template specializációknál lesz leírva, de a karakterlánc literálokra már most adhatunk választ.

Mivel a C++-ban van lehetőség függvénytúlterhelésre, ezért a fordító fordítás közben nem csak a függvény nevét, de annak paraméterlistáját, visszatérési értékét és a template paraméterekre vonatkozó információkat (stb.) is kénytelen eltárolni a függvény nevével együtt. Amennyiben karakterlánc literál is lehetne template paraméter, nagyon meg tudna nőni ennek a sztringnek a hossza, és lassíthatná a fordítási időt.

```
template <typename T, int ArraySize>
int arraySize(const T (&array)[ArraySize])
{
    return ArraySize;
}

int main()
{
    int i[10];
    std::cout << arraySize(i) << std::endl; //10
}
```

A fenti kód a 3. gyakorlat végén tett megjegyzésből lehet ismerős. Jól demonstrálja a template paraméter dedukciót.

## 1.2. Osztály template-ek

Nem csak függvények, osztályok is lehetnek template-ek melyen nagyon hasonlóan működnek. A következő kódrészletekben a template osztályok mellett megismerkedhetünk még a template-ek „lustaságával” is.

```
#include <iostream>

template <typename T>
struct X
{
    void f()
    {
        T t;
        t.foo();
    }
};

struct Y
{
    void bar() {}
};

int main() {}
```

Ez a kód úgy tűnhet, hogy nem fog lefordulni, lévén mi soha semmilyen `foo` tagfüggvényt nem írtunk, azonban mégis le fog. Ez azért van, mert a template osztályok (és függvények) csak sablonok, amiből aminek alapján a fordító generálhat egy konkrét osztályt (vagy függvényt), és mivel sose példányosítottuk, fordítás után az `X` template osztály nem fog szerepelni a kódban. Szintaktikus ellenőrzést végez a fordító, (pl. zárójelek be vannak-e zárva, pontosvessző nem hiányzik-e stb.), de azt, hogy van-e olyan `T` típus, ami rendelkezik `foo()` függvénnel, már nem.

Példányosítsuk az `X` osztályt `Y`-nal!

```
int main()
{
    X<Y> x;
}
```

Ekkor már azt várnánk hogy fordítási hibát dobjon a fordító, hisz Y-nak nincs `foo()` metódusa, azonban mégis gond nélkül lefordul, mivel az `f()` tagfüggvényt nem hívtuk meg, így nem is példányosult az osztályon belül.

```
int main()
{
    X<Y> x;
    x.f();
}
```

Itt már végre kapunk fordítási hibát, mert példányosul `f()`. Ez jól mutatja, hogy a template-ek lusták, és csak akkor példányosulnak, ha „nagyon muszáj”.

A template-eknek adhatunk meg alapértelmezett értéket.

```
template <typename T = void> //alapértelmezett paraméter
struct X { /* ... */};

struct Y { /* ... */};

int main()
{
    X<Y> x;
    X<> x2;
}
```

Ilyenkor nem szükséges megadni template paramétert (mely esetben értelemszerűen `X<> == X<void>`).

Ahogy az említve volt korábban, szinte bármi lehet template paraméter, akár egy másik template is.

```
template <typename T>
struct X { /* ... */};

struct Y { /* ... */};

template <template <typename> class Templ>
struct Z
{
    Templ<int> t;
};

int main()
{
    Z<X> z;
}
```

Fent `Templ` egy olyan template, aminek a template paramétere egy típus. Így `Z`-nek a template paramétere egy olyan template, aminek a template paramétere egy típus. Mivel `X` egy template (és template paramétere egy típus), így megadható `Z`-nek template paraméterként.

**1.2.1. Megjegyzés.** Fent a template paraméter listában `typename` helyett `class` szerepel. Ezek gyakorlatilag ekvivalensek, mind a kettő azt jelenti, hogy az adott paraméter típus (bár a `typename` beszédesebb).

A fenti példákban mindig egy default konstruktort hívtunk meg, amikor objektumokat hoztunk létre. Helyes lenne-e az, ha explicit módon kírnánk a zárójeleket (hangsúlyozva a default konstruktor hívást)?

```
int main()
{
    X<Y> x();
    X<> y2();
    Z<X> z();
}
```

A kód helyesen lefordul, de a jelentése nem ugyanaz, mintha nem lenne ott a zárójel. Mivel a `c++` nyelvtana nem egyértelmű, más kontextusban ugyanaz a kódrészlet mást jelenthet (egyik legegyszerűbb példa a `static`

kulcsszó), így meg kellett alkotni egy olyan szabályt, miszerint amit deklarációként lehet értelmezni, azt deklarációként **kell** értelmezni. Így ezek függvénydeklarációk lesznek: Az első esetben például egy olyan függvényt deklarálunk, melynek neve `x`, `X<Y>`-al tér vissza és nem vár paramétert. Így ha default konstruktort szeretnének meghívni, semmilyen zárójelt nem szabad használni.

**1.2.2. Megjegyzés.** C++11ben lehet gömbölyű zárójel helyett kapcsos zárójelet alkalmazni konstruktorhívásnál, így ez a probléma nem fordulhat elő. pl: `X<Y> x{};`

A template-ek paramérének ismertnek kell lennie fordítási időben.

```
template <int N>
void f() {}

int main()
{
    int n;
    std::cin >> n;
    f<n>(); //hiba, n nem ismert fordítási időben
}
```

Ez nyilvánvaló, hisz a template-eknek az a funkciója, hogy a fordító generáljon belőlük példányokat, és a fordítási idő végeztével erre nincs lehetőség.

**1.2.3. Megjegyzés.** Fontos még, hogy a template-ek nagyon megnövelik a fordítási időt, így nem mindig éri meg egy olyan függvényt is template-ként megírni, melyet nem feltétlenül muszáj.

### 1.3. Template specializáció

Néha szeretnénk, hogy bizonyos speciális behelyettesítéseknél más legyen az implementáció mint az alap sablonban. Ilyenkor szokás **specializációkat** (*template specialization*) létrehozni:

```
template <class T>
struct A
{
    A() { std::cout << "general A" << std::endl; }
};

template <> //template specializáció
struct A<int>
{
    A() { std::cout << "special A" << std::endl; }
};

template <class T>
void f() { std::cout << "general f" << std::endl; }

template <> //template specializáció
void f<int>() { std::cout << "special f" << std::endl; }

int main()
{
    A<std::string> a1; //general A
    f<std::string>(); //general f
    A<int> a2; //special A
    f<int>(); //special f
}
```

Mind `A` osztályhoz, mind `f` függvényhez létrehoztunk egy specializációt arra az esetre, ha a template paraméterként `int`-et kapnak. Számos okunk lehet arra hogy ezt tegyük: a standard könyvtár megfényesebb példája az `std::vector` osztály, mely egy template, és van template specializációja `bool` esetre.

**1.3.1. Megjegyzés.** Az `std::vector<bool>` számos optimalizációt tartalmazhat (persze nem feltétlenül, hisz ez implementáció függő): általában nem `bool`-okban tárolja az adatokat, hanem bitekben. Sajnos azonban ez hátrányokkal is jár, például hogy a `[]` operátor érték és nem referencia szerint ad vissza.

**1.3.2. Megjegyzés.** Visszatérve egy korábbi állításhoz, miért nem lehet lebegőpontos szám template paraméter? Lévén két lebegőpontos szám könnyedén lehet csak nagyon kis mértékben eltérő, ezért könnyű egy ilyesmi hibába belefutni:

Legyen adott `d1`, `d2`, fordítási időben ismert lebegőpontos szám! Mi azt hisszük, hogy ez a kettő egyenlő, de mivel számos módosításon mentek keresztül, minimális mértékben, de nem lesznek egyenlőek. Ilyenkor ha egy template paraméterként lebegőpontos számot váró függvénynek megadnánk őket template paraméterül, kétszer kéne példányosítani az adott függvényt.

Ez a kisebb gond, de mi van ha pont erre az értékre mi létrehoztunk egy template specializációt, és csak (pl.) `d1` esetben került az a függvény meghívásra? Ez ellen a fordító se tudná a felhasználót megvédeni.

Írjunk faktoriális számoló algoritmus template-ek segítségével!

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

template<>
struct Fact<0>
{
    static const int val = 1;
};

int main()
{
    std::cout << Fact<5>::val << std::endl; //120
}
```

Fact 4szer példányosul: `Fact<5>`, ..., `Fact<1>`, majd a legvégén az általunk specializált `Fact<0>`-t hívja meg.

**1.3.3. Megjegyzés.** Ahogyan ezt korábban megállapítottunk, egy **konstans** osztályszintű változót függvény-törzsön belül is inicializálhatunk.

Ez fel is hívja a figyelmet a template-ek veszélyeit statikus változók használatakor.

```
template <class T>
class A
{
    static int count;
public:
    A()
    {
        std::cout << ++count << ' ', ' ';
    }
};

template <class T>
int A<T>::count = 0;

int main()
{
    for(int i = 0; i<5; i++)
    {
        A<int> a;
        A<double> b;
    }
}
```

Kimenet: 1 1 2 2 3 3 4 4 5 5

Bár arra számítanánk, hogy 1-től 10ig lesznek a számok kiírva, ne felejtsük, hogy itt két teljesen különböző osztály fog létrejönni: `A<int>` és `A<double>`, így a `count` adattag hiába osztályszintű, 2 teljesen különböző példánya lesz ennek is: `A<int>::count` és `A<double>::count`.

## 1.4. Dependent scope

Lehetőségünk van arra hogy osztályon belül deklaráljunk még egy osztályt. Bár erről bővebben a következő órai jegyzetben lesz szó, egy igen fontos problémát vet fel.

```
class A
{
public:
    class X {};
};

void f(A a)
{
    A::X x;
}

int main()
{
    A a;
    f(a);
}
```

Ezzel semmi probléma nincs. Legyen `A` egy template osztály!

```
template <class T>
class A
{
public:
    class X {};
};

template <class T>
void f(A<T> a)
{
    A<T>::X x;
}

int main()
{
    A<int> a;
    f(a);
}
```

Itt máris bajba jutottunk, a fordító azt a hibát fogja jelezni, hogy `X` egy ún. **dependent scope**-ban van. Ez azt jelenti, hogy attól függően, milyen template paraméterrel példányosítjuk `A`-t, `X`-nek lehet más a jelentése. Az alábbi kód ezt jól demonstrálja:

```
template <typename T>
struct A
{
    class X{};
};

template <>
struct A <int>
{
```

```

    static int X;
};

int A<int>::X = 0;

template <typename T>
void f()
{
    A<T>::X;
}

```

Itt az `f` függvényben vajon mi lesz `A<T>::X`? A válasz az hogy nem tudni, hisz ha `int`-el példányosítunk akkor statikus adattag, ha bármi mással, akkor meg egy típus. Ezért kell a fordítónak biztosítani, hogy a `template` paramétertől függetlenül garantáltan típust fog oda kerülni. Ezt a `typename` kulcsszóval tehetjük meg.

```

template <typename T>
void f()
{
    typename A<T>::X;
}

```

A `typename` garantálja a fordítónak, hogy bármi is lesz `T`, `A<T>::X` mindenképpen típus lesz. Ha mégis olyan `template` paramétert adunk meg, aminél ez nem teljesülne (ez esetben `T = int`) akkor fordítási idejű hibát kapunk.

**1.4.1. Megjegyzés.** A fordító általában szokott szólni, hogy a `typename` kulcsszó hiányzik.

**1.4.2. Megjegyzés.** A dependent scope problémája nem csupán az osztályon belüli osztályokra érvényes. Nemsokára meglátjuk, hogy a `typedef` kulcsszó is ide tud vezetni.

Így viszont felmerülhet a kérdés hogy van-e szükség `typename` kulcsszóra, ha egy `std::vector<int>::iterator` típusú objektumot akarunk létrehozni (`std::vector<int>::iterator vit;`). A válasz az hogy nem, hisz ha konkrétan megadjuk a típust, amellyel példányosítanánk, akkor a fordító arra a konkrét típusra vissza tudja keresni, hogy `std::vector<int>::iterator` típus-e, vagy sem.