Sablonok *–template*

- **Függvénysablonok**

```
inline double max(double a, double b)
{
        return a > b ? a : b;
}
```

komplexre:

```
inline Complex max(Complex a, Complex b)
{
        return a>b ? a :b;
}
```

>> a függvény törzse ugyanaz marad, csak a típus változik!

cél: írjunk olyan függvény sablont, amiben a típus paraméterként szerepel, felhasználás során lehet megadni… >> **függvénysablon**

```
template <class t>
inline T max(T a, T b)
{
        return a > b ? a : b;
}
```

- függvény neve elé → template kulcsszó
- <…> -ben felsoroljuk a sablon paramétereket (a példában egy paraméter volt, ennek a T nevet adtuk
- ha a paraméter egy típus akkor a „class” vagy a „typename” kulcsszót kell eléírni

felhasználás:

1.) implicit példányosítás

```
int n = max(3,4);   //1-eset
double d = max(3.2, 4.2);  //2-eset
```

- nem adtuk meg explicit a paramétereket
- 1 esetben class T = int, míg a 2. esetben double → **a fordító a paraméterekből következtet, hogy melyik sablont használja!**

- a sablon fordítási időben fejtődik ki!
- a felhasználás során behelyettesíti a sablon paramétereket és legyárt egy közönséges függvényt

következmény: minden paraméter kombinációra legenerálódik a kód → nagyon menőhet a kód mérete → „code bloat” ~~ kód burjánzás

a sablon paraméter – esetünkben T csak egy adott típust jelölhet (nincs konverzió!)

double d = max(3.1, 2); //??? … double és int → **NEM fordul le**

2.) *explicit példányosítás*

double d = max<double>(3.1, 2); //lefordul, mert normális fgv.-ként fordul → van típuskonverzió!

>> explicit felsoroljuk a sablonparaméterek értékét

Osztálysablonok

- gyakrabban használjuk, mint a függvénysablonok

pl.:

Stack osztály → int elemeket tárolt
Complex elemeket, hogy tárolhatunk?

>> osztálysablont írunk, ahol az elem típusa paraméter, amelyet a felhasználás során lehet megadni…

Eredeti stack osztály:

```
class Stack
{
        int* pData;
        int elementsNum;
public:
        void push(int n){…}
        int pop(){…}

        Stack(const Stack& s) {…}
};
```

→ *templateként*

```
template <class T> //T: elem típusa
class Stack
{
        T* pData;
        int elementsNum;
public:
        void push(T item){…}
        …
```

```
        Stack(int size)
        {
                pData = new T[size];
        }

        Stack (const Stack<T>& s) {…}
        T pop(); // ezt máshol adjuk meg!
};

//Stack.cpp
#include „Stack.h”
…
template <class T>
T Stack<T>::pop() {…}
```

1.) class kulcsszó elé >> template <paraméterek felsorolása>
2.) sablon paraméterek bevezetése a kódba (int helyébe T)
3.) az osztályt mint típus felhasználva a név után fel kell sorolni a paramétereket <> között. Pl.: Stack<T>& → ez csak a régebbi fordítóknál kötelező
konstruktor destruktor neve NEM változik!
4.) ha egy műveletet az osztály definícióban csak deklarálunk: a művelet definíciója csak a fenti példa szerint adható meg!

használata:

```
Stack<int> s(3); // felsoroljuk a sablonparaméterek értékét, fordításkort T->int
…
Stack<Complex> s2(3);
…
Stack<Stack<int>> s3(4);
```

~~teljes értékű osztálynak számít!

T az elem típusa bármilyen típus lehet?

T-nek kell legyen default konstruktora!

T-nek a másoló konstruktora és az = operátora jól kell működnie!

Max függvénytípusnál a T bármilyen típus lehet?

```
{
        return a>b ? a:b;
}
```

→ csak olyan típusokra ahol a > operátor meg van írva! /nem fordul le ahol nincs megírva!/

Sablonparaméterek

```
template<class T, class U, int N>

T f(U u)
{
        T var;
        for(int i=0; i < N; i++) {…}
        …
}
```

fgv.sablon → 3 paraméter: -T, U típus

- **N pedig int konstans paraméter → csak konstans lehet**

Fordítási kifejtás következményei

1.) A fordító csak akkor fejt ki, fordít le egy függvény/osztály sablont, ha használjuk
pl.: megírjuk a Stack sablonunkat, 0 hibéval fordul le, pedig lehet, hogy van benne hiba, ha nem használjuk sehol!

```
Stack<int> s1(3);
```

valójában függvényenként kell fordítani → az előző sorral csak a konstrukor fordult le…