

# KIV/TSI - Seminář C++

## 07. Šablony II. (variadické šablony), statický polymorfismus, CRTP

Martin Úbl

KIV ZČU

2020/2021

- opakování
- generické programování
- parametrizace typem nebo hodnotou
- instancování šablony při použití (parametrizací)
- existuje tolik instancí, kolik různých sad parametrů

```
template<typename T>
void Print(const T& what)
{
    std::cout << what << std::endl;
}
```

```
Print(20);
```

- lze dědit parametrizovanou šablonu třídy
- opět jako šablonu nebo jako specializaci

```
class CharVector : public std::vector<char>
{
    // ...
};
```

```
template<typename T>
class BetterVector : public std::vector<T>
{
    // ...
};
```

- musíme ale dodefinovat konstruktor(y)

```
class CharVector : public std::vector<char>
{
    public:
        CharVector(std::initializer_list<T> li)
            : std::vector<char>(li)
        {
        }
};
```

```
// ...
```

```
CharVector cv{ 1,2,3,4 };
```

- nebo přejmout všechny konstruktory

```
class CharVector : public std::vector<char>
{
    public:
        using std::vector<char>::vector;
};
```

- pozn. konstruktor šablonový není, jen třída

- nebo přejmout všechny konstruktory - obecněji

```
class B : public A
{
    public:
        using A::A;
};
```

- takto přejímáme vždy všechny
  - nelze přejmout jen vybrané
- toto není výsadou pouze šablon, ale jakékoliv dědičnosti

- variadické šablony
- dovolují definovat proměnlivý počet šablonových parametrů
- „statická analogie“ `va_args` z C
- specializace
  - rekurzivní
  - přímá (forwarding)

```
template <typename ... T>
void DoSomething(T... args)
{
    std::cout << "Doing..." << std::endl;
    Something(args...);
}
```

- rekurzivní specializace, např. suma šablonou

```
template<typename T1>
T1 TemplateSum(T1 fa)
{
    return fa;
}
```

```
template<typename T1, typename... T>
T1 TemplateSum(T1 fa, T... args)
{
    return fa + TemplateSum(args...);
}
```

```
TemplateSum(1,2,3,4,5);
```



- rekurzivní specializace
- ne vždy dobrý nápad

```
TemplateSum(1,2,3,4,5);
```

- generuje všechny specializace funkce

```
TemplateSum<int>
```

```
TemplateSum<int, int>
```

```
TemplateSum<int, int, int>
```

```
TemplateSum<int, int, int, int>
```

```
TemplateSum<int, int, int, int, int>
```

- důsledek: roste velikost binárky, v runtime 5 funkčních volání
  - někdy lze zredukovat použitím klíčového slova `inline`

- syntaxe s třemi tečkami se nazývá *parameter pack*
- reprezentuje jeden a více šablonových parametrů
- v definici „balí“ parametry, v implementaci „rozbaluje“

```
template<typename... Args>
void DoSomething(Args... args)
{
    SomethingElse(args...);
}
```

- `args...` rozbalí parametry – `args1, args2, args3, ...`

- fold expressions
- dovoluje aplikovat binární operátor na parameter pack při jeho rozvinutí
- např. fold expression na výpis

```
template<typename ...Args>
void Print(Args&&... args) {
    (std::cout << ... << args) << std::endl;
}
```

- args... rozbalí parametry – args1, args2, args3, ...

- perfect forwarding
- v kombinaci s r-value referencí, principem reference-collapsing a funkcí `std::forward` dokážeme předávat libovolný šablonový řetěz dál

```
class Student {
public:
    Student(const std::string& name,
            int birthyear) { ... }
};

template <typename T1, typename T2>
Student MakeStudent(T1&& a, T2&& b) {
    return Student(std::forward<T1>(a),
                  std::forward<T2>(b));
}
```

- perfect forwarding s variadickými šablonami

```
class Student {
public:
    Student(const std::string& name,
            int birthyear) { ... }
};

template <typename... Args>
Student MakeStudent_2(Args&&... args) {
    return Student(std::forward<Args>(args)...);
}
```

- perfect forwarding v praxi
- `std::unique_ptr` a `std::make_unique`
- předává parametry do konstruktoru

```
auto ptr = std::make_unique<Something>  
          ("Neco", 15);
```

- vyvolá např. konstruktor

```
Something(const std::string& a, int b);
```

- perfect forwarding v praxi
- `emplace_back` u kontejnerů
- vytvoří instanci rovnou v předalokovaném místě v kontejneru
- není třeba kopie nebo `move`

```
// docasna instance + move  
container.push_back({ "Marco", 1 });
```

```
// in-place vytvoreni bez move  
container.emplace_back("Polo", 2);
```

- kontejnery zakládající na variadikách
- `std::tuple`
- `#include <tuple>`
- kontejner reprezentující n-tici

```
std::tuple<int, std::string, double> ntice;
```



- `std::tuple`
- hodnoty se vyzvedají šablonově přes `std::get<n>`, kde *n* je index prvku
- nebo pomocí `std::tie` – přiřazení vnitřku do sady l-value

```
std::tuple<int, std::string, double> ntice;
```

```
int a          = std::get<0>(ntice);  
std::string b = std::get<1>(ntice);  
double c      = std::get<2>(ntice);
```

- `std::tuple`
- `std::tie` – přiřazení vnitřku do sady l-value

```
std::tuple<int, std::string, double> ntice;
```

```
int a;  
std::string b;  
double c;
```

```
std::tie(a,b,c) = ntice;
```

- další použití variadických šablon
  - databázové drivery a binding hodnot
    - někdy v kombinaci s idiomem SFINAE
  - zobecnění vytváření objektů
  - generické „průchozí“ schéma, např. cachování výsledků
  - a další...

- statický polymorfismus
- polymorfismus bez virtuálních metod
- vyhodnocený v čase kompilace
- dědičnost specializované šablony s parametrem sebe
- nenahrazuje dynamický polymorfismus, jen ho doplňuje pro určité úlohy

```
template <typename Child>
class Base {
public:
    void interface() {
        static_cast<Child*>(this)->impl();
    }
};

class Derived : public Base<Derived> {
public:
    void impl() {
        std::cout << "Derived\n";
    }
};
```

- statický polymorfismus
- generický rodič definuje rozhraní
- specifický potomek definuje implementaci
- opakovaný cast se často nahrazuje malou metodou

```
Derived& derived() {  
    return *static_cast<Derived*>(this);  
}
```

- `static_cast` namísto `dynamic_cast`
- jsme si jisti, že výsledek bude přetypování správně
  - potomek dědí předka jako šablonu
  - šablonu parametrizuje sám sebou

- konkrétní použití viz příklady
- do prezentace se nevejde

- `assert`
  - aserce za běhu
  - při ladění programu může odfiltrovat vstupy, pro které např. není funkce definována
  - vyhodnocení za běhu
  - dovoluje např. zastavit běh programu (breakpoint)
- `static_assert`
  - vyhodnocení v čase kompilace
  - ověření sémantických pravidel
  - ověření velikosti, typů, vlastností objektu, ...
  - typicky v kombinaci s nějakou standardní šablonovou deklarací



- `static_assert`

```
template<typename T>
void RequiresCopy(const T& obj)
{
    static_assert(std::is_copy_constructible_v<T>,
                  "T must be copy-constructible");

    // something
}
```

- T může být `std::string`, `int`, ...
- T nemůže být např. `std::unique_ptr`

- sada standardních statických podmínek
  - `std::is_integral`
  - `std::is_floating_point`
  - `std::is_copy_constructible`
  - `std::is_copy_assignable`
  - `std::is_class`
  - `std::is_pod`
  - `std::is_final`
  - `std::is_unsigned`
  - `std::is_base_of`
  - a další...
  - nebo jiné, staticky vyhodnotitelné podmínky
    - `sizeof(T) == ...`

- stripping
  - `std::remove_reference`
  - `std::remove_const`
  - `std::remove_volatile`
  - `std::remove_cv`
  - a další...