

KIV/CPP – Programování v jazyce C++

03. Objekty, virtuální metody, const, constexpr, final, přetěžování funkcí a operátorů

Martin Úbl

KIV ZČU

2023/2024

- třída – deklaráce, implementace
- objekt – instance třídy
 - už má místo v paměti

```
class Rect
{
    // ...
};
```

```
Rect obdelnik;
```

- virtuální metoda – polymorfismus
- klíčové slovo `virtual`
- tabulka virtuálních metod
 - program za běhu volí, kterou metodu bude volat
 - dynamický polymorfismus
- přepisování metod v potomcích
 - v rodiči musí být `virtual`
 - v potomkovi může být `override` (měl by být)
- přepisovaná metoda musí mít identickou signaturu
 - návratová hodnota
 - název
 - parametry
 - kvalifikátory (`const`, `volatile`, `register`)
 - volací konvence

```
class Rodic
{
    public:
        virtual std::string GetName();
};

class Potomek : public Rodic
{
    public:
        virtual std::string GetName() override;
};
```

- klíčové slovo `final`
- deklaruje, že entita nesmí být dále přepsána
 - třída – nesmí mít potomka
 - metoda – nesmí být potomkem přepsána
- sémantická kontrola
- optimalizace – tzv. devirtualizace
 - kompilátor se může „zbavit“ dynamického polymorfismu, pokud si je jistý, že to jde
 - drobné urychlení

```
class Rodic {
    public:
        virtual std::string GetName();
};

class Potomek : public Rodic {
    public:
        virtual std::string GetName()
            override final;
};

class PotomekPotomka final : public Potomek {
    // ...
};
```

Čistě virtuální metody

- čistě virtuální metody (pure virtual)
- úmyslně neimplementované metody
- deklarována s uvedením = 0 za signaturou metody
- její přítomnost dělá z třídy abstraktní
- některému z potomků ukládá povinnost metodu přepsat a implementovat
 - nelze instancovat třídu s čistě virtuální metodou
- třídy, které obsahují jen čistě virtuální metody, jsou vlastně rozhraním
 - C++ nezná pojem *rozhraní* (jako např. Java nebo C#)

```
class IDrawable {  
    public:  
        virtual void Draw() const = 0;  
};
```

- výchozí implementace
 - default, move, copy konstruktory
 - move-assign a copy-assign operátory
 - destruktory
 - uvození pomocí = default za signaturou
 - programátor „dovolí“ kompilátoru vygenerovat výchozí implementaci (indikuje záměr)
- smazané implementace
 - move, copy konstruktory
 - move-assign a copy-assign operátory
 - typové konverze
 - uvození pomocí = delete za signaturou
 - zamezí tomu, aby kompilátor tyto konstrukty generoval za nás

- výchozí implementace

```
class Objekt {  
    public:  
        Objekt() = default;  
        Objekt(int parametr);  
        virtual ~Objekt() = default;  
};
```

Smazaná implementace

- smazaná implementace
- můžeme např. zakázat kopírování

```
class Objekt {  
    public:  
        Objekt();  
  
        // zakazani kopie konstruktorem  
        Objekt(const Objekt& obj) = delete;  
        // zakazani kopie prirazenim  
        Objekt& operator=(const Objekt&) = delete;  
};
```

- pozn. operátory později, teď jen pro představu

- v C++ je konstruktor s jedním parametrem považován za konverzní
- explicitní vyloučení
 - uvození pomocí `explicit` před deklarací
 - zamezí automatickému použití konverzního konstruktora

Explicitní vyloučení

```
class Objekt {  
    public:  
        Objekt(int a);  
};
```

- versus

```
class Objekt {  
    public:  
        explicit Objekt(int a);  
};
```

- v prvním případě projde, ve druhém ne:

```
void NejakaFunkce(Objekt a);  
NejakaFunkce(42);
```

Výchozí parametr

- parametry funkce nebo metody mohou mít výchozí hodnotu
- po nich nesmí následovat parametry co ji nemají
- v oddělené definici se už neuvádějí

```
class TextObject {
public:
    TextObject(Color color, int size = 12) {
        // ...
    }
    void Mod(Color color = Color::White,
             int size = 12);
};

void TextObject::Mod(Color color, int size) {
}
```

- statické metody

```
class Rectangle
{
    public:
        static double Calc_Surface(double a,
                                   double b);
};
...
double s = Rectangle::Calc_Surface(2.0, 3.0);
```

- statické atributy
 - definovány ve třídě
 - deklarovány v implementaci

```
class Rectangle
{
    public:
        static size_t Instance_Counter;
};

...
size_t Rectangle::Instance_Counter = 0;
```

Dopředná deklarace

- dopředná deklarace (obdobně v C)
 - víme, že je třída někde definována
 - nemůžeme předřadit (includovat) její definici
 - např. kvůli kruhové závislosti

```
class Map;
```

```
class GameObject {  
    protected:  
        std::shared_ptr<Map> mMap;  
};
```

```
class Map {  
    protected:  
        std::vector<std::unique_ptr<GameObject>  
            mObjects;  
};
```


- klíčové slovo `const`
- neměnná hodnota, neměnný objekt
 - `const` proměnná – neměnná hodnota proměnné
 - `const` metoda – nemění vnitřní stav objektu
- `const` hodnota má vždy fyzicky místo v paměti
 - narozdíl od `constexpr`, viz dále
- sémantická kontrola
- prostor pro optimalizace kompilátorem
- u metod lze částečně obejít klíčovým slovem `mutable`
 - typicky pro atributy co se mění dočasně v rámci volání
 - např. `std::mutex` a zamčení po čas volání
 - používat jen výjimečně a jen tam, kde se hodí

- konstantní hodnota

```
const size_t size = vec.size();
```

- konstantní metoda – GetName() nezmění stav objektu, jen vrátí nějaké jméno

```
class Person {  
    public:  
        std::string GetName() const;  
};
```

- konstantní parametr

```
void print_var(const int value);
```

- konstantní proměnná – ukazatel
- lze měnit adresu kam ukazuje, nelze měnit paměť, na kterou ukazuje

```
const int* values;
```

- konstantní ukazatel na proměnná data
- nelze měnit adresu kam ukazuje, lze měnit paměť, na kterou ukazuje

```
int* const values;
```

- reference na konstantní paměť

```
const int& cr_value;
```

- konstantní reference na proměnnou paměť je nesmysl
- „cíl“ reference už je z podstaty konstantní
 - warning C4227: anachronism used: qualifiers on reference are ignored

```
int& const rc_value;
```

- constexpr
- výraz nebo funkce, která se může vyhodnotit v čase kompilace
- konstanty
- výpočet „magic“ konstant
- deterministické výpočty s konstantními parametry

```
constexpr double PI = 3.1415926535897932384;  
constexpr double Inv_PI = 1.0 / PI;
```

```
constexpr uint64_t faktorial(uint64_t n) {  
    return (n > 1) ? n * faktorial(n - 1) : 1;  
}
```

- pozn.: faktoriál rekurzí je stále špatně, jen demonstruje možnost použití constexpr

- od C++17 už nemusí být constexpr funkce jen jednořádková s jedním return
- výraz/funkce se jako constexpr vyhodnotí jen tehdy, když
 - je výsledek přiřazen do další constexpr proměnné
 - parametry jsou constexpr
- jinak fallback na vyhodnocení za běhu
- náhrada #define
 - constexpr je typově bezpečné
 - constexpr funkce se vyhodnotí pokud možno v čase překladač
 - preferovat vždy před makry
- od C++17 navíc if constexpr – o té až později

- od C++20 klíčové slovo `constexpr` pro funkce
- varianta `constexpr`, ale funkce **musí** být vyhodnotitelná v čase překladače

```
constexpr int magic(int n) {  
    return n*n*2 + 15;  
}
```

- od C++20 klíčové slovo `constexpr` pro proměnné a konstanty
- varianta `constexpr`, pro konstanty
- proměnná/konstanta **musí** být inicializována hodnotou, která se vypočte v čase překladač

```
// OK:
```

```
constexpr double MyMagic = magic(16) + 8;
```

```
// Fail:
```

```
// (time() není constexpr, ani constexpr)
```

```
constexpr double MyMagic = time(nullptr) + 8;
```


Přetěžování funkcí a metod

- princip známý z mnoha jazyků
- funkce/metoda má stejný název, ale liší se vždy parametry
- může se lišit i návratová hodnota (ale ne výhradně)
- pochopitelně každá deklarace má vlastní implementaci

```
class RGBColor {
public:
    void Set(int r, int g, int b);
    void Set(const RGBColor& color);
    bool Set(const std::string& str);
}
```

- pozn.: pozor na blízké typy (např. znaménkové a bezznaménkové varianty)

- klasické matematické operátory
 - +, -, *, /, %, ++, --
- bitové
 - &, |, ^, ~, <<, >>
- porovnávací
 - ==, <=, >=, <, >, !=
- přiřazovací (+ compound)
 - =, +=, *=, /=, ...
- přístupové
 - [], *, &, ->, .., ...
- další...
 - fnc(...), comma, ternární operátor, casty, new, delete, sizeof, ...
- <https://en.cppreference.com/w/cpp/language/operators>

- třídě lze definovat, co bude operátor dělat
- počet parametrů je dán aritou operátoru
- volíme
 - návratovou hodnotu
 - typ parametru
 - `const`
 - reference
- nepřetěžovat operátory jen proto, že můžeme
 - (špatný) příklad z produkce: operátor `+` odesílal data přes síťový socket

Přetěžování operátorů

```
class Cislo {
private:
    int cislo = 0;

public:
    Cislo& operator=(int rhs) {
        cislo = rhs;    // prirazeni
        return *this;
    }

    Cislo& operator+=(int rhs) {
        cislo += rhs;  // compound pricteni
        return *this;
    }

    ...
}
```

```
...  
    Cislo& operator=(const Cislo& rhs) {  
        cislo = rhs.getCislo();  
        return *this;  
    }  
  
    Cislo& operator+=(const Cislo& rhs) {  
        cislo += rhs.getCislo();  
        return *this;  
    }  
...
```

...

```
// operatory pretypovani
operator int() const {
    return cislo;
}

operator bool() const {
    return (cislo != 0);
}

operator std::string() const {
    return std::to_string(cislo);
}
```

...

```
...
    friend std::ostream& operator<<
        (std::ostream& os, const Cislo& c);
}

std::ostream& operator<<(std::ostream& os,
                        const Cislo& c)
{
    os << "Cislo_je_" << c.cislo;
    return os;
}
```

- pozn. klíčové slovo `friend` dovoluje uvedené třídě „nahlédnout“ do `private/protected` části třídy

- operátor lze i „dodefinovat“ vně třídy

```
Cislo operator+(const Cislo& a, const Cislo& b)
{
    Cislo ret;
    ret = (int)a + (int)b;
    return ret;
}
```

- pozn. lze lépe, např. dodefinováním konverzního konstrukturu pro int, pak stačí

```
return (int)a + (int)b;
```


- návratová hodnota = „výsledek“ operace
 - může být očekávatelná
 - např. `Cislo + Cislo` bude vracet určitě instanci `Cislo`
 - nebo stanovena např. dohodou
 - výsledkem přiřazení by mělo být také `Cislo`
 - pak dodržuje konvence s okolním kódem
 - (výsledkem přiřazení integeru je přiřazovaný integer)
 - nebo silně závislá na kontextu použití
 - funktořy

Funktor

- funktor je třída, jejíž instance „lze volat jako funkci“
- ve skutečnosti obyčejná třída s přetíženým operátorem funkčního volání ()

```
class NasobFunc
{
public:
    int operator()(const int a) {
        return a*2;
    }
};

...
NasobFunc mul;

int a = 5;
int b = mul(a);
```

- `istream` a `ostream`, operátory `<<`, `>>`
 - `std::cout << cislo << retezec;`
- iterátory, operátory `++`, `-`, `+`, `-`, `==`, `!=` dereference
 - `++itr`
- `string`, operátory `==`, `!=`, `+`
 - `if (str == "hello") ...`
- a mnoho dalších

- v C++20 přibyl nový operátor pro porovnání
- `<=>`, tzv. *spaceship operator*
- trojcestné porovnání
- není binární (ano/ne), ale vrací ordering konstantu
- implicitně `std::strong_ordering`
 - `less`, `equal`, `equivalent`, `greater`
 - jeden výraz tedy vrátí výsledek několika různých porovnání

Spaceship operator

- příklad – necháme kompilátor dedukovat

```
struct IntWrapper {
    int value;
    auto operator<=>(const IntWrapper& a) const
        = default;
};
```

- příklad – definujeme sami to, co by kompilátor dedukoval

```
struct IntWrapper {
    int value;
    auto operator<=>(const IntWrapper& a) const {
        return value <=> a.value;
    }
};
```

Spaceship operator

- příklad – definujeme vlastní

```
struct IntWrapper {
    int value;
    auto operator<=>(const IntWrapper& a) const
    {
        if (value < a.value)
            return std::strong_ordering::less;
        else if (value == a.value)
            return std::strong_ordering::equal;
        else
            return std::strong_ordering::greater;
    }
};
```

Spaceship operator

- *rewritten expressions*
- kompilátor může některé výrazy nahrazovat ekvivalenty v daném kontextu
- spaceship operator je jeden z příkladů, kdy se tak může dít

```
IntWrapper a{4}, b{5};
```

$$a < b \quad \text{-->} \quad (a <=> b) < 0$$

- pak `a <=> b` vrací `std::strong_ordering::less`
 - to expanduje na `-1`, což je `< 0` a tedy je podmínka splněna

- `<=>`
- hodí se jako náhrada výčtu všech operátorů
- v podstatě umí nahradit `<`, `>`, `<=`, `>=`, `==` a `!=`
- může se vyplatit když
 - nás zajímá výsledek více druhů porovnání
 - nechceme implementovat celou sadu operátorů

- `<=>`
- procesor by pro číselné hodnoty stejně něco takového udělal
- instrukce `CMP` (a podobné) fakticky udělají „všechna porovnání najednou“
 - vlastně provedou instrukci `SUB` (odčítání) bez `write-back` do registru
 - nastaví odpovídající příznaky do `FLAGS` registru
- nejde tedy o overhead, ale naopak o přiblížení tomu, jak ve skutečnosti porovnává procesor
- pro řetězce je definováno rozhraní identické `strcmp` funkci z C
 - ideální pro lexikografické řazení