KIV/TSI - Seminář C++

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

Martin Úbl

KIV ZČU

2020/2021

Poznámky

- `++itr` versus `itr++`
- `math.h` versus `cmath`
- „`map` is not a member of `std`“
  - `#include <map>`
- poznámky odevzdávání
  - MSVS solution, CMake, Makefile, Sconstruct, ...
  - odebrat pracovní a dočasné soubory („Clean solution“ nebo analogie)
  - odevzdávaný archiv má max. pár kilobajtů
- precompiled header

Třída, objekt

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

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

Rect obdelnik;
```

Virtuální metody

- virtuální metoda – polymorfismus
- klíčové slovo `virtual`
- tabulka virtuálních metod
  - za běhu volí, co za metodu se zavolá
  - 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
  - `const` kvalifikátory

Virtuální metody

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

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

Virtuální metody

- 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
  - jisté urychlení

Virtuální metody

```
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)
- „neimplementovaná“ metoda
- deklarována pomocí `= 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
- typicky takto lze „nahrazovat“ neexistenci rozhraní

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

Výchozí a smazané konstruktory a destruktory

- výchozí implementace
  - default, move, copy konstruktory
  - move-assign a copy-assign operátory
  - destruktor
  - 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

- 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

Explicitní vyloučení

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

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) {
}
```

- klíčové slovo `const`
- neměnná hodnota, neměnný objekt
  - `const` proměnná - neměnná hodnota proměnné
  - `const` metoda - nemění vnitřní stav objektu
- fyzicky má místo v paměti
- 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`
  - používat jen výjimečně a jen tam, kde se hodí

- konstantní hodnota

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

- konstantní metoda - `GetName()` nezmění stav objektu, jen vrací nějaké jméno

```
class Rodic {
  public:
    virtual 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
- 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 = 3.14159265358979323384;
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
- jinak fallback na vyhodnocení za běhu

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. znamínkové a bezznamínkové varianty)

Operátory

- 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`

Přetěžování operátorů

- 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;
    }
...
```

Přetěžování operátorů

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

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

Přetěžování operátorů

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

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

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

Přetěžování operátorů

```
...
    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 subjektu „nahlédnout“ do private/protected části třídy

Přetěžování operátorů

- 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 konstruktoru pro int, pak stačí

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

Přetěžování operátorů

- návratová hodnota = „výsledek“ operace
- může být více samozřejmá
  - např. Cislo + Cislo bude vracet určitě instanci Cislo
- nebo méně samozřejmá
  - výsledkem přiřazení by mělo být také Cislo
  - pak dodržuje konvence s okolním kódem
  - (výsledkem přiřazení integeru je přiřazovaný integer)
- nebo vůbec samozřejmá
  - funktory

Funktor

- funktor je třída, kterou „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);
```

Přetížené operátory v STL

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