NPRG041 – C++

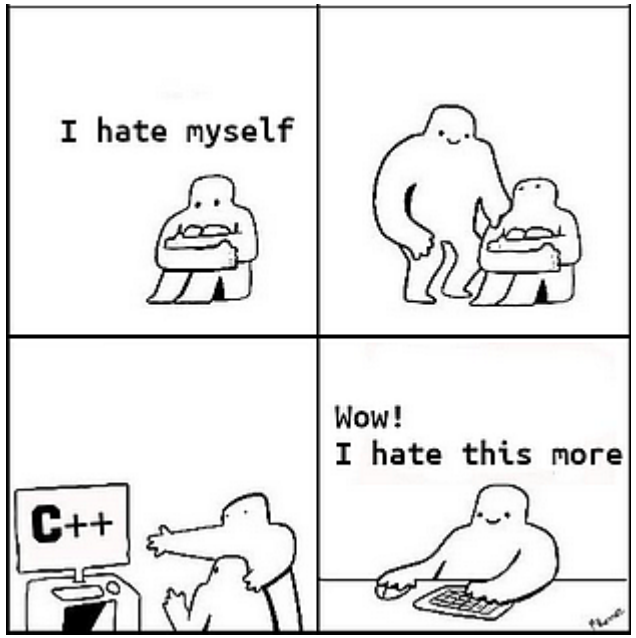
cvičení – Jiří Klepl

mattermost: ulita/2425ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

Klepl@d3s.mff.cuni.cz

Agenda

- Dědičnost
- Dynamický polymorfismus
  - virtual, override, final
- The Rule of Three, The Rule of Five, The rule of Zero



Dědičnost

```
class Base {
public:
    static bool is_base() {
        return true;
    }
};

class Derived : public Base { };

void accept_base(const Base& b) {
    assert(b.is_base());
}

int main() {
    Base b;
    Derived d;

    accept_base(b);
    accept_base(d);
}
```

Dědičnost

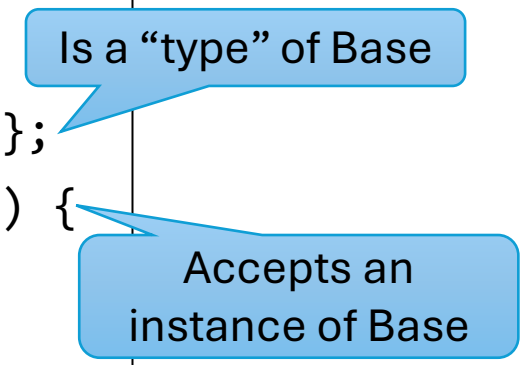
```
class Base {
public:
    static bool is_base() {
        return true;
    }
};

class Derived : public Base { };

void accept_base(const Base& b) {
    assert(b.is_base());
}

int main() {
    Base b;
    Derived d;

    accept_base(b);
    accept_base(d);
}
```



Dědičnost

```
class Base {
public:
    static bool is_base() {
        return true;
    }
};

class Derived : public Base { };

void accept_base(const Base& b) {
    assert(b.is_base());
}

int main() {
    Base b;
    Derived d;

    accept_base(b);
    accept_base(d);
}
```

Is a “type” of Base

Accepts an instance of Base

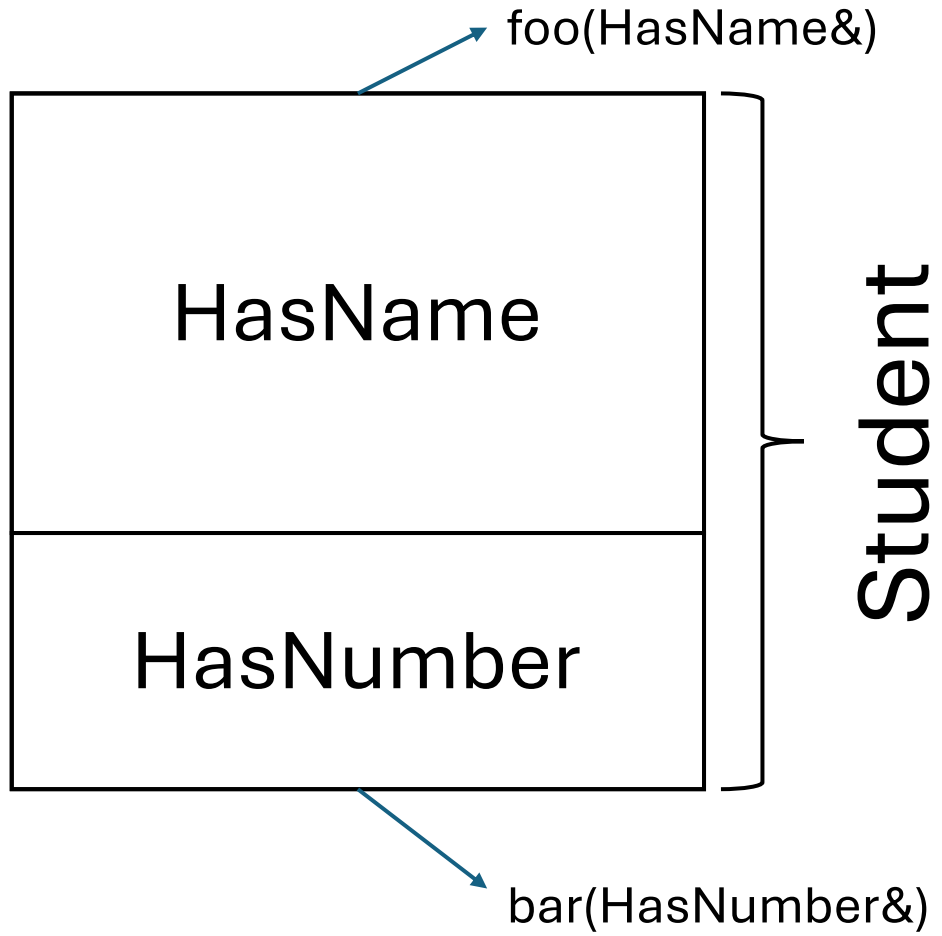
➔ Accepts a Derived

Derived obsahuje Base

```
class HasNumber {
public:
    HasNumber(int n) : number_{n} {}
    int number() const { return number_; }
private:
    int number_;
};

class HasName {
public:
    HasName(const char* n): name_{n} {}
    const std::string& name() const { return name_; }
private:
    std::string name_;
};

class Student : public HasName, public HasNumber {};
```



Dynamický polymorfismus

```
class Printable {
public:
    virtual ~Printable() noexcept = default;

    friend std::ostream& operator<<(std::ostream& out,
                                    const Printable& self) {
        self.print(out);
        return out;
    }

private:
    virtual void print(std::ostream& out) const = 0;
};
```

Dynamický polymorfismus

```
class Printable {
public:
    virtual ~Printable() noexcept = default;

    friend std::ostream& operator<<(std::ostream& out,
                                    const Printable& self) {
        self.print(out);
        return out;
    }

private:
    virtual void print(std::ostream& out) const = 0;
};
```

⚠ Virtuální destruktor

Virtuální (overridovatelná metoda)

Bez defaultní implementace

Dynamický polymorfismus

```
class Printable {
public:
    virtual ~Printable() noexcept = default;

    friend std::ostream& operator<<(std::ostream& out,
                                    const Printable& self) {
        self.print(out);
        return out;
    }

private:
    virtual void print(std::ostream& out) const = 0;
};
```

⚠ Virtuální destruktor

Virtuální (overridovatelná metoda)

Bez defaultní implementace

Metoda nemá defaultní implementaci („pure virtual“) == abstraktní metoda
Třída obsahuje pure virtual funkci bez overridu == abstraktní třída

Overridování

```
template<class Base>
class DefaultPrintable : public Printable {
private:
    void print(std::ostream& out) const override {
        out << typeid(Base).name();
    }
};
```

Říká compileru:
„Printable by mělo obsahovat virtual print“

Overridování

```
template<class Base>
class DefaultPrintable : public Printable {
private:
    void print(std::ostream& out) const override {
        out << typeid(Base).name();
    }
};
```

Říká compileru:
„Printable by mělo obsahovat virtual print“

⚠️ virtual ~Printable() se overriduje automaticky
☠️ pokud je virtuální

Overridování

```
template<class Base>
class DefaultPrintable : public Printable {
private:
    void print(std::ostream& out) const override {
        out << typeid(Base).name();
    }
};
```

Říká compileru:
„Printable by mělo obsahovat virtual print“

⚠️ virtual ~Printable() se overriduje automaticky
☠️ pokud je virtuální

```
public:
    ~DefaultPrintable() noexcept override = default;
```

„Rodič musí mít virtuální destruktor“

Final overridování

```
class Int : public Printable {
public:
    Int(int i) : value{i} {}

    int value;

private:
    void print(std::ostream& out) const final {
        out << value;
    }
};
```

Říká compileru:
„ Chci overridovat a nechci to povolit dětem“

The Rule of Five

Pokud třída Class obsahuje

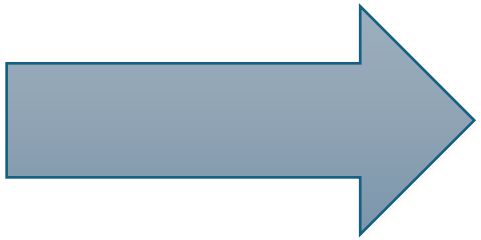
- User-defined destructor
  `~Class()`
- nebo User-defined copy constructor
  `Class(const Class&)`
- nebo User-defined copy assignment
  `Class& operator=(const Class&)`
- nebo User-defined move constructor
  `Class(Class&&)`
- nebo User-defined move assignment
  `Class& operator=(Class&&)`

https://en.cppreference.com/w/cpp/language/rule_of_three

The Rule of Five

Pokud třída Class obsahuje

- User-defined destructor
  `~Class()`
- nebo User-defined copy constructor
  `Class(const Class&)`
- nebo User-defined copy assignment
  `Class& operator=(const Class&)`
- nebo User-defined move constructor
  `Class(Class&&)`
- nebo User-defined move assignment
  `Class& operator=(Class&&)`



Pak

By měla obsahovat všech 5

```
class Class {
public:
  ~Class() noexcept;
  Class(const Class&);
  Class& operator=(const Class&);
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;
};
```

https://en.cppreference.com/w/cpp/language/rule_of_three

The Rule of Five

Pokud třída Class obsahuje

- User-defined destructor
  `~Class()`
- nebo User-defined copy constructor
  `Class(const Class&)`
- nebo User-defined copy assignment
  `Class& operator=(const Class&)`
- nebo User-defined move constructor
  `Class(Class&&)`
- nebo User-defined move assignment
  `Class& operator=(Class&&)`

Pak

By měla obsahovat všech 5

```
class Class {
public:
  ~Class() noexcept;
  Class(const Class&);
  Class& operator=(const Class&);
  Class(Class&&) noexcept;
  Class& operator=(Class&&) noexcept;
};
```

Kolik toho je potřeba implementovat???

https://en.cppreference.com/w/cpp/language/rule_of_three

Malá záchrana

```
class Class {
public:
    ~Class() noexcept = default;
    Class(const Class&) = delete;
    Class& operator=(const Class&) = delete;
    Class(Class&&) noexcept { ... }
    Class& operator=(Class&&) noexcept { ...
}
};
```

Některé funkce většinou budou **default**ované nebo **delete**d

https://en.cppreference.com/w/cpp/language/rule_of_three

Malá záchrana

```
class Class {
public:
    ~Class() noexcept = default;
    Class(const Class&) = delete;
    Class& operator=(const Class&) = delete;
    Class(Class&&) noexcept { ... }
    Class& operator=(Class&&) noexcept { ...
}
};
```

Některé funkce většinou budou **default**ované nebo **delete**d

ℹ️ `delete` se hodí, když implementujeme třeba handle nějakého resourcu (např unique_ptr)

⚠️ `default` je zrádná věc
→ pokud compiler rozhodne, že to nejde implementovat, tak to bez upozornění defaultuje na = `delete`

https://en.cppreference.com/w/cpp/language/rule_of_three

Co když na některou funkci zapomenu

Class.cpp

```
class Class {
public:
    ~Class() noexcept = default;
    Class(const Class&) = delete;
    Class& operator=(const Class&) = delete;
    Class& operator=(Class&&) noexcept {
        /* ... */
        return *this;
    }
};
```

```
/home/jirka/nprg041/nprg041-web/data/06/Class.cpp:1:7: warning: class 'Class' defines a default destructor, a copy constructor, a copy assignment operator and a move assignment operator but does not define a move constructor [cppcoreguidelines-special-member-functions]
    1 | class Class {
      |       ^
```

```
$ clang-tidy -checks=cppcoreguidelines* Class.cpp -- -Wall -Wextra -pedantic
```

Zapnutí checků podle **core guidelines**

Klasické parametry pro konstruktor

https://clang.llvm.org/extra/clang-tidy/

https://learn.microsoft.com/en-us/cpp/code-quality/clang-tidy

https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

The rule of zero

Nechat děti, aby se spravovaly samy

```
class rule_of_zero
{
    std::string cppstring;
public:
    // redundant, implicitly defined is better
    // rule_of_zero(const std::string& arg) : cppstring(arg) {}
};
```

The rule of zero

Nechat děti, aby se spravovaly samy

```
class rule_of_zero
{
    std::string cppstring;
public:
    // redundant, implicitly defined is better
    // rule_of_zero(const std::string& arg) : cppstring(arg) {}
};
```

Nejhorší >>>>>> Nejlepší

User-provided (= custom)

```
class Class
{
  Class(const Class& other)
    : child{other.child} {}

private:
  std::string child;
};
```

User-defaulted (= explicit default)

```
class Class
{
  Class(const Class&)
    = default;

private:
  std::string child;
};
```

Implicitly-defined

```
class Class
{
private:
  std::string child;
};
```

Příklad: labs/expression

- Vycházejte ze souboru **expression.hpp** (libovolně upravte podle potřeby)
  - Obsahuje implementace **Expression**, **Value**
  - Třídy implementují strom reprezentující aritmetický výraz
- Implementaci napište do **expression.cpp**, **main.cpp** testuje funkčnost
- **Expression.evaluate()** vrátí pointer na Value, která odpovídá hodnotě výrazu
  - Každý derived Expression definuje chování vyhodnocování
  - Expression používá čísla pomocí například: *left + *right (hodnoty definují operátory)
- Value implementuje fukce **add**, **mul** a **print**
  - Value může obsahovat **int** nebo **double**
  - Pravidla operací **add** a **mul**:
    aspoň jeden operand je **double** → výsledek je **double**, jinak **int**
  - **Bude nejspíš potřeba využít „double-dispatch“**