NPRG 041 – cvičení 7
Programování v C++

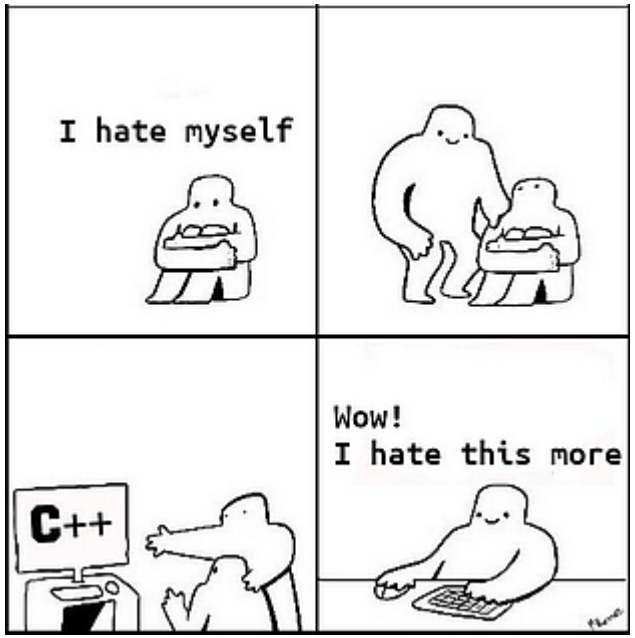
Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

mattermost:
https://ulita.ms.mff.cuni.cz/mattermost/ar2324zs/channels/nprg041-cpp-klepl

Agenda

- **Šestá úloha: Interpret Výrazů**



Dynamický polymorfismus

struct Nepolymorfni { void foo() { } };

- Nedefinuje **žádnou virtuální metodu**
  - Objekty té třídy přesně odpovídají definici (nemají nic navíc)
  - Všechny volání jsou efektivní a dohledatelné podle typu objektu

struct Polymorfni { virtual void foo() { } };

- Definuje **alespoň jednu virtuální metodu**
  - Díky tomu dostane tzv. „v-pointer“, který ukazuje na tzv. „v-table“
    - V-pointer je neviditelným fieldem té třídy, každý objekt je o něco větší, než bychom čekali
    - Ve v-table jsou správné implementace metod pro ten daný objekt (třída definuje jen interface)
  - Objekty té třídy **ukazované pointerem** nemusí být nutně té třídy, ale i **Derived**
  - Virtuální metody jsou pak typicky volány přes ten v-table

Dynamický polymorfismus

- **Virtuální metody**
  - Mohou být overridovány derived třídami
  - Volají se speciálním mechanismem (vtable)
    - Podle skutečného typu objektu (ne jak je v kódu):

    Engine* x = make_diesel();
    decltype(x) == Engine*

    // calls DieselEngine::start:
    x->start();
- **Virtuální destruktor**
  - **Povinnost**, jinak program neví, jak správně objekt zničit (musí být volán přes vtable)

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

    virtual void start() { ... }
    virtual void stop() = 0;
    virtual void rev() = 0;
    ...
};
class DieselEngine : public Engine {
public:
    void start() override { ... }
    void stop() final { ... }
    void rev() { ... }
    ...
};
class PetrolEngine : public Engine { ... };
class ElectricEngine : public Engine { ... };
```

Engine je **abstraktní třída** – nejde ho vyrobit (protože má aspoň jednu pure-virtualmetodu)

Deklarace virtuálního destruktoru
Ale implementaci necháme na compileru

Virtuální metoda s default implementací

„Pure virtual“ metody (abstraktní)

Explicitní override

Final override

Pozor! Implicitní override

Sklad motorů

std::unordered_map<int, std::unique_ptr<Engine>> warehouse{…};

- **Engine nejde vyrobit, tak jak ho přidáme do skladu?**
  - **unique_ptr** (i **shared_ptr**) naštěstí chápe dynamický polymorfismus:
    warehouse.insert({last_id++, **std::make_unique<DieselEngine>(…)**});
    warehouse.insert({last_id++, **std::make_unique<PetrolEngine>(…)**});
- Reference také fungují s polymorfismem: **void start(Engine& engine)** přijme libovolný motor a bude volat správné verze virtuálních metod
- **for (auto&& engine : warehouse) engine->rev();** // zavolá správné

Auto s polymorfním motorem

- **Auto má pointer na motor**, obojí je polymorfní a tak můžeme mít **Škodovku s Dieselem** a nebo **elektrický Seat** (různé kombinace)
  - Tím můžeme rozdělit implementační detaily různých komponent:
    3 + 4 je udržitelnější než 3 * 4

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

    virtual void start() = 0;
    virtual void stop() = 0;
    ...
private:
    std::unique_ptr<Engine> engine_;
};

class Volkswagen : public Car { ... };
class Audi : public Car { ... };
class Skoda final : public Volkswagen { ... };
class Seat final : public Volkswagen { ... };
```

Škoda je typ VW (ne jen auta)

Od škody a seatu nelze dál dědit

Auto s polymorfním motorem

- **Auto má pointer na motor**, obojí je polymorfní a tak můžeme mít **Škodovku s Dieselem** a nebo **elektrický Seat** (různé kombinace)
  - Tím můžeme rozdělit implementační detaily různých komponent:
    3 + 4 je udržitelnější než 3 * 4
- **Jak ale okopírovat auto?**

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

    virtual void start() = 0;
    virtual void stop() = 0;
    ...
private:
    std::unique_ptr<Engine> engine_;
};

class Volkswagen : public Car { ... };
class Audi : public Car { ... };
class Skoda final : public Volkswagen { ... };
class Seat final : public Volkswagen { ... };
```

Škoda je typ VW (ne jen auta)

Od škody a seatu nelze dál dědit

Auto s polymorfním motorem a klonováním

Místo std::unique_ptr<Engine> stačí psát **ptr**, popř. **Engine::ptr**

```
class Engine {
public:
    using ptr = std::unique_ptr<Engine>;

    virtual ~Engine() noexcept = default;

    virtual ptr clone() const = 0;
};

class Car {
public:
    using ptr = std::unique_ptr<Car>;

    virtual ~Car() noexcept = default;

    virtual ptr clone() const = 0;
private:
    std::unique_ptr<Engine> engine_;
};
```

Virtuální metoda na výrobu kopie

Jak bude vypadat useful instance?

Auto s polymorfním motorem a klonováním

Místo std::unique_ptr<Engine> stačí psát **ptr**, popř. **Engine::ptr**

```
class Engine {
public:
    using ptr = std::unique_ptr<Engine>;

    virtual ~Engine() noexcept = default;

    virtual ptr clone() const = 0;
};
```

Virtuální metoda na výrobu kopie

```
class Car {
public:
    using ptr = std::unique_ptr<Car>;

    virtual ~Car() noexcept = default;

    virtual ptr clone() const = 0;
private:
    std::unique_ptr<Engine> engine_;
};
```

Pro čitelnost schováme implementace

```
class DieselEngine : public Engine {
public: DieselEngine(int capacity, int power)
        : capacity_(capacity), power_(power) {}

    DieselEngine(const DieselEngine& other)
        : DieselEngine(other.capacity_, other.power_) {}
    DieselEngine& operator=(const DieselEngine& other) {
        capacity_ = other.capacity_; power_ = other.power_;
        return *this; }

    DieselEngine(DieselEngine&& other) noexcept
        : DieselEngine(other.capacity_, other.power_) {}
    DieselEngine& operator=(DieselEngine&& other) noexcept {
        capacity_ = other.capacity_; power_ = other.power_;
        return *this; }

    ~DieselEngine() noexcept {
        engine_registry.unregister(this); }

    ptr clone() const override {
         return std::make_unique<DieselEngine>(*this); }
private: int capacity_, power_;
};
```

Polymorfním motor s klonováním
The Rule of Five

Pokud třída definuje jednu z vyznačených pěti funkcí
(zde definuje destructor, který něco dělá,) tak by měla definovat všechny:
copy constructor + assignment, move constructor + assignment, destruktor

```
class DieselEngine : public Engine {
public:
  DieselEngine(int capacity, int power) { … }

  DieselEngine(const DieselEngine& other) : …
  { … }
  DieselEngine& operator=(const DieselEngine& other)
  { … }

  DieselEngine(DieselEngine&& other) noexcept : …
  { … }
  DieselEngine& operator=(DieselEngine&& other)
  { … }

  ~DieselEngine() noexcept {
        engine_registry.unregister(this); }

  ptr clone() const override {
         return std::make_unique<DieselEngine>(*this); }
private: …
};
```

The Five

Copy constructor a copy assignment

Move constructor a move assignment

Nějaký speciální desktruktor

Kvůli tomuhle musíme definovat všechny z the five

Vyrobíme ptr na nový Diesel a vrátíme ptr na obecný Engine, abychom splnili API

Klonování používá copy constructor

Polymorfním motor s klonováním
The Rule of Five s defaultováním

Pokud ty funkce nedělají nic zajímavého a compiler je dokáže odvodit, můžeme použít **= default**

```
class DieselEngine : public Engine {
public:
  DieselEngine(int capacity, int power) = default;

  DieselEngine(const DieselEngine& other) = default;
  DieselEngine& operator=(const DieselEngine& other) = default;

  DieselEngine(DieselEngine&& other) noexcept = default;
  DieselEngine& operator=(DieselEngine&& other) noexcept = default;

  ~DieselEngine() noexcept {
        engine_registry.unregister(this); }

  ptr clone() const override {
         return std::make_unique<DieselEngine>(*this); }
private:
  int capacity_, power_;
};
```

The Five

Polymorfním motor bez klonování
The Rule of Five s deletem

Co ale když nechceme klonovat a nepotřebujeme, aby ten typ šlo třeba kopírovat, použijeme **= delete**

```
class DieselEngine : public Engine {
public:
  DieselEngine(int capacity, int power) = default;

  DieselEngine(const DieselEngine& other) = delete;
  DieselEngine& operator=(const DieselEngine& other) = delete;

  DieselEngine(DieselEngine&& other) noexcept = default;
  DieselEngine& operator=(DieselEngine&& other) noexcept = default;

  ~DieselEngine() noexcept {
        engine_registry.unregister(this); }

private:
  int capacity_, power_;
};
```

Explicitně zakázané kopírování

The Five

Polymorfním motor s klonováním
The Rule of Zero

Ideál je ale **The Rule of Zero**
To speciální chování odvedeme do fieldu
a nemusíme psát žádné z the five, vše za nás udělá compiler

```
class DieselEngine : public Engine {
public:
  DieselEngine(int capacity, int power) = default;

  /* all are implemented by the compiler */

  ptr clone() const override {
          return std::make_unique<DieselEngine>(*this); }
private:
  int capacity_, power_;
  registry_handle registry_handle_;
};
```

The Five

To speciální chování destruktoru jsme
odvedli do pomocného objektu,
takže tady nemusíme řešit nic

Jiný příklad tohodle jsou:
smart pointery a kontejnery

Přetypovávání
C-Style a functional-style

```
double d = 42.;
int i = (int)d;
int j = int(d);
```

- V C++ máme 4 druhy castu:
  - **(C-style)cast** a **functional_style(cast)**
    - Dědictví z C, zkoušejí postupně casty vypsané níže (const, static, reinterpret)
    - **Nepoužívat!!!**
    - Není v kódu jasné, co udělá (musíme odvodit z cílového typu a z typu proměnné)
  - **const_cast<DestType>(arg)**
  - **static_cast<DestType>(arg)** (a **dynamic_cast<DestType>(arg)**)
  - **reinterpret_cast<DestType>(arg)**

Přetypovávání *const_cast*

- V C++ máme 4 druhy castu:
  - **(C-style)cast** a **functional_style(cast)**
  - **const_cast<DestType>(arg)**
    - Můžeme použít na **přidání const k typu** (i odebrání, ale u toho pozor!)
    - Typické použití je třeba implementace const a non-const verzí metod u kontejnerů
  - **static_cast<DestType>(arg)** (a **dynamic_cast<DestType>(arg)**)
  - **reinterpret_cast<DestType>(arg)**

```
class T {
public:
    value* find(key) {
        /* dlouhá definice */
    }

    const value* find(key) const {
        return const_cast<const value*>(
            const_cast<T*>(this)->find()
        );
    }
}
```

Přetypovávání
static_cast a dynamic_cast

- V C++ máme 4 druhy castu:
  - **(C-style)cast** a **functional_style(cast)**
  - **const_cast<DestType>(arg)**
  - **static_cast<DestType>(arg)**
    - Compile-time typová konverze
    - Musí existovat cesta z typu arg do DestType
    - Efektivní, typicky bezpečný způsob
    - Lze použít v dynamickém polymorfismu:
      - Z Derived* na Base* – to je bezpečné
      - Z Base* na Derived* – musíme si být jisti, že je skutečně Derived, takže vždy pozor!!!
  - **dynamic_cast<DestType>(arg)**
  - **reinterpret_cast<DestType>(arg)**

```
int i = 42;
float f = static_cast<float>(i);


// Enum to int
enum class Color { RED, GREEN, BLUE };
int value = static_cast<int>(Color::RED);


auto p_orig = make_unique<Derived>();

// Pointer upcasting
Base* p_b = static_cast<Base*>(p_orig);

// Pointer downcasting (with caution!!!)
Derived* p_d = static_cast<Derived*>(p_b);
```

Přetypovávání
static_cast a dynamic_cast

- V C++ máme 4 druhy castu:
  - **(C-style)cast** a **functional_style(cast)**
  - **const_cast<DestType>(arg)**
  - **static_cast<DestType>(arg)**
  - **dynamic_cast<DestType>(arg)**
    - Funguje jen na **polymorfních třídách**
      - Bere pointery nebo reference
      - Na těch, které mají virtuální metodu
    - Použijeme, pokud si nejsme jisti, jaká konkrétní derived třída to je
    - **Stará se o bezpečnost:**
      - **Cast na špatné dítě, pointer -> return nullptr**
      - **Cast na špatné dítě, reference -> throw std::bad_cast**
  - **reinterpret_cast<DestType>(arg)**

```
unique_ptr<Base> p = make_unique<Derived>();

// succeeds
Derived* d = static_cast<Derived*>(p.get());

// fails, `d2 == nullptr`
Derived2* d2 = static_cast<Derived2*>(p.get());

// succeeds
Derived& d3 = static_cast<Derived&>(*p);

// fails, throws `std::bad_cast`
Derived2& d4 = static_cast<Derived2&>(*p);
```

Přetypovávání *reinterpret_cast*

- V C++ máme 4 druhy castu:
  - **(C-style)cast** a **functional_style(cast)**
  - **const_cast<DestType>(arg)**
  - **static_cast<DestType>(arg)**
  - **dynamic_cast<DestType>(arg)**
  - **reinterpret_cast<DestType>(arg)**
    - Na pointerech a referencích
    - **Zapomene na typ argumentu** a vymyslí pro něj nový typ
      - Ten typ **MUSÍ být kompatibilní**
      - Velice nebezpečný!!!
      - Používaný typicky na převod data->byty->data

```
                    SERVER SIDE
auto server_data = std::vector<int>(100);
...
char* server_bytes = reinterpret_cast<char*>(
                       server_data.data()
                     );
-----------------------------------------
                    CLIENT SIDE
auto client_data = std::vector<int>();

// copy the bytes to data_client
client_data.resize(server_data.size());
char* client_bytes = reinterpret_cast<char*>(
                       client_data.data()
                     );

std::memcpy(client_bytes,
            server_bytes,
            server_data.size()*sizeof(int)
);
```

Cyklická dependence mezi dvěma třídami

Compiler čte postupně, takže B ještě neexistuje

```
struct A {
    void interact_with(B b) const {
        b.do_something();
    }
    void do_something() const {
        std::cout << "A::do_something()" << std::endl;
    }
};

struct B {
    void interact_with(A a) const {
        a.do_something();
    }
    void do_something() const {
        std::cout << "B::do_something()" << std::endl;
    }
}
```

Cyklická dependence mezi dvěma třídami forward deklarace třídy

```
struct A;
struct B;
struct A {
    void interact_with(B b) const {
        b.do_something();
    }
    void do_something() const {
        std::cout << "A::do_something()" << std::endl;
    }
};
struct B {
    void interact_with(A a) const {
        a.do_something();
    }
    void do_something() const {
        std::cout << "B::do_something()" << std::endl;
    }
}
```

Ok, B bylo deklarováno
tak víme, že existuje

Ale nic o B nevíme,
tak nemůžeme vyrobit objekt b

Cyklická dependence mezi dvěma třídami vyndání definice funkce ven

```
struct A;
struct B;
struct A {
    void interact_with(B b) const;
    void do_something() const {
        std::cout << "A::do_something()" << std::endl;
    }
};

struct B {
    void interact_with(A a) const;
    void do_something() const {
        std::cout << "B::do_something()" << std::endl;
    }
}

void A::interact_with(B b) const { b.do_something(); }
void B::interact_with(A a) const { a.do_something(); }
```

Teď už vše OK

v .CPP

Cyklická dependence mezi dvěma třídami vyndání definice funkce ven

```
struct A;
struct B;
struct A {
    void interact_with(B b) const;
    void do_something() const {
        std::cout << "A::do_something()" << std::endl;
    }
};

struct B {
    void interact_with(A a) const;
    void do_something() const {
        std::cout << "B::do_something()" << std::endl;
    }
}

inline void A::interact_with(B b) const { b.do_something(); }
inline void B::interact_with(A a) const { a.do_something(); }
```

- **V header souboru musíme přidat inline**
  - U všech funkcí definovaných mimo tělo nějaké třídy
- **Pozor!** inline nesouvisí s inlinováním kódu
- inline znamená: „Pokud to uvidíš i v jiném modulu, sjednoť to s tímhle“
  - To se stane kvůli includům
- Další důvod, proč funkce definovat v .cpp souborech

Teď už vše OK

v .HPP