NPRG041 – C++

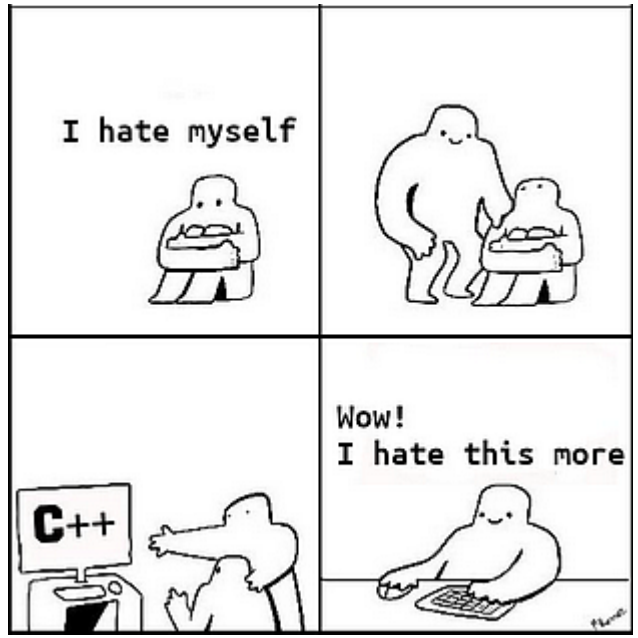
cvičení – Jiří Klepl

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

Klepl@d3s.mff.cuni.cz

Agenda

- Cast
  - „C-style cast“
  - static_cast a dynamic_cast
  - const_cast
  - reinterpret_cast
- Parsování
- Varianty a visitování
- std::optional a std::expected



Přetypovávání

- **(C-style)cast** a **functional_style(cast)**
- **const_cast<DestType>(arg)**
- **static_cast<DestType>(arg)** (a **dynamic_cast<DestType>(arg)**)
- **reinterpret_cast<DestType>(arg)**

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

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

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

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

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

⚠️ C-Style a functional-style

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

- Dědictví z C, compiler se pokusí vše následující:
  1. const_cast
  2. static_cast
  3. static_cast + const_cast
  4. reinterpret_cast
  5. reinterpret_cast + cost_cast
- Není v kódu jasné, co udělá (musíme odvodit z cílového typu a z typu proměnné)

static_cast

- **static_cast<DestType>(arg)**
- Compile-time typová konverze
- Pro hodnoty
  - Zkusí najít cestu konverzí z argumentu do **DestType**
    - Konstruktory a `operator Type()`
- Na pointerech/referencích
  - ✅ Z **Derived*** na **Base*** (upcast)
  - ⚠️ Z **Base*** na **Derived*** (downcast)
  - Musíme si být jisti, že **arg** je **Derived**

```
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);
```

dynamic_cast

- Funguje jen na **polymorfních třídách** (= mají virtuální metodu)
- Použijeme, pokud si nejsme jisti, jaká konkrétní **derived** třída to je

✅ **Stará se o bezpečnost**

- Selhání na **pointeru**
  - **static_cast<Derived*>(arg)**
  - Výsledek je **nullptr**
- Selhání na **referenci**
  - **static_cast<Derived&>(arg)**
  - Vyhodí výjimku **std::bad_cast**

```
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);
```

const_cast

- Můžeme použít na **přidání const k typu**
- ⚠️ **Můžeme použít na odebrání const od typu**
- Zjednodušuje implementaci **const** a non-**const** verzí metod kontejnerů
- Přidáním **const** můžeme něco chránit před omylným zápisem
- ⚠️ U odebírání **const** stále nesmíme měnit tu hodnotu

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

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

Jednoduché parsování výrazů

1. Seřadíme operátory podle priority (zde od těch nejmíň prioritních):
   I. +, -
   II. *, /, %
   III. čtení hodnoty a uzávorkovávání
2. Z operátorů uděláme funkce (nebo metody parseru)
   I. Expression::pointer parse_add_expression(std::istream& in)
   II. Expression::pointer parse_mul_expression(std::istream& in)
   III. Expression::pointer parse_value_expression(std::istream& in)
3. Pro jednotlivé typy výrazů si připravíme polymorfní třídy
   (a pomocné make_<typ> funkce ať nemusíme řešit pointery)

(pokračování na dalším slajdu)

Ukázka funkce pro parse_add _expression

```
Expression::pointer parse_add(std::istream &in) {
    auto lhs = parse_mul_expression(in);

    while (in >> std::ws && (in.peek() == '+' || in.peek() == '-')) {
        char op = (char)in.get();
        auto rhs = parse_mul_expression(in);

        if (op == '+') {
            lhs = make_add_expression(std::move(lhs),
                                      std::move(rhs));
        } else {
            lhs = make_sub_expression(std::move(lhs),
                                      std::move(rhs));
        }
    }

    return lhs;
}
```

Ukázka funkce pro parse_add_expression

Pokusíme se přečíst první operand

```
Expression::pointer parse_add(std::istream &in) {
    auto lhs = parse_mul_expression(in);

    while (in >> std::ws && (in.peek() == '+' || in.peek() == '-')) {
        char op = (char)in.get();
        auto rhs = parse_mul_expression(in);

        if (op == '+') {
            lhs = make_add_expression(std::move(lhs),
                                      std::move(rhs));
        } else {
            lhs = make_sub_expression(std::move(lhs),
                                      std::move(rhs));
        }
    }

    return lhs;
}
```

Ukázka funkce pro parse_add_expression

```
Expression::pointer parse_add(std::istream &in) {
    auto lhs = parse_mul_expression(in);

    while (in >> std::ws && (in.peek() == '+' || in.peek() == '-')) {
        char op = (char)in.get();
        auto rhs = parse_mul_expression(in);

        if (op == '+') {
            lhs = make_add_expression(std::move(lhs),
                                      std::move(rhs));
        } else {
            lhs = make_sub_expression(std::move(lhs),
                                      std::move(rhs));
        }
    }

    return lhs;
}
```

Pokusíme se přečíst první operand

Přeskočíme bílé znaky a podíváme se, jestli následuje operátor + nebo -

Ukázka funkce pro parse_add_expression

```
Expression::pointer parse_add(std::istream &in) {
    auto lhs = parse_mul_expression(in);

    while (in >> std::ws && (in.peek() == '+' || in.peek() == '-')) {
        char op = (char)in.get();
        auto rhs = parse_mul_expression(in);

        if (op == '+') {
            lhs = make_add_expression(std::move(lhs),
                                      std::move(rhs));
        } else {
            lhs = make_sub_expression(std::move(lhs),
                                      std::move(rhs));
        }
    }

    return lhs;
}
```

Pokusíme se přečíst první operand

Přeskočíme bílé znaky a podíváme se, jestli následuje operátor + nebo -

Updatujeme lhs a pokračujeme

Výsledný expression

Ukázka pro parse_numeric

Pomocná funkce na parsování čísel

```
std::string parse_numeric(std::istream& in) {
    char c;
    std::string numeric;

    in >> std::ws;
    while (in.get(c)) {
        if (is_numeric_char(c))
            numeric.push_back(c);
        else
            in.unget(); // rollback the character
    }

    if (numeric.empty())
        throw std::runtime_error("expected a number");

    return numeric;
}
```

Ukázka pro parse_numeric

```
std::string parse_numeric(std::istream& in) {
    char c;
    std::string numeric;

    in >> std::ws;
    while (in.get(c)) {
        if (is_numeric_char(c))
            numeric.push_back(c);
        else
            in.unget(); // rollback the character
    }

    if (numeric.empty())
        throw std::runtime_error("expected a number");

    return numeric;
}
```

Načteme znak a podíváme se, jestli je numeric (např. isdigit nebo '.')

Vrátíme se o znak zpět

Nepodařilo se načíst nic

Věc, co můžeme zparsovat na číslo
⚠ můžeme na tom dělat další kontroly
např. že se neopakují tečky

Základní věci na funkcionální programování v C++

std:variant a std::visit

- Type-safe union
- Náhrada polymorfismu
  - + Dovoluje nesouvisející typy
  - - Musíme vyjmenovat všechny varianty

    (zatímco u dyn. polymorfismu stačí aby splňovaly nějaký interface a nemusíme o nich vědět nic víc)
- Pro naše expressions se hodí např. na reprezentaci **Value**
- **std::visit** pustí visitora na hodnotu uloženou v **std::variant**

https://en.cppreference.com/w/cpp/utility/variant

Visitor je functor

```
struct visitor {
  void operator()(int i) const {
    cout << "int: " << i << endl;
  }
  void operator()(auto a) const {
    cout << "generic: " << a << endl;
  }
  void operator()(float f) const {
    cout << "float: " << f << endl;
  }
};

int main() {
  using item_type =
    variant<int, const char*, float>;
  vector<item_type> items{5, "hi", .5f};

  for (auto&& item : items)
    visit(visitor(), item);
}
```

Reprezentuje různé funkce

Visitor s více argumenty

```
using item_type = variant<int, double>;
struct adder {
    item_type operator()(auto a, auto b) const {
        return a + b;
    }
};
int main() {
    vector<item_type> items{.4, 42, 9.6, 1, 2};

    item_type sum = 0;
    while (!items.empty()) {
        sum = visit(adder(), sum, items.back());
        items.pop_back();

        visit([](auto val) {
            cout << "sum is " << val << endl;
        }, sum);
    }
}
```

Output:

```
sum is 2
sum is 3
sum is 12.6
sum is 54.6
sum is 55
```

std::optional<T>

- Užitečný pro typy, které nemají ekvivalent **null** hodnoty
  - **std::optional<T> opt** drží hodnotu ***opt** (taky **opt.value()**), a nebo **std::nullopt**
    - Podle **if(opt.has_value())** nebo **if(opt)**

```
std::optional<int> may_fail() {
    if (failure) {
        return {}; // std::nullopt
    }

    // success
    return 42;
}
```

https://en.cppreference.com/w/cpp/utility/optional

std::optional<T>

- Užitečný pro typy, které nemají ekvivalent **null** hodnoty
  - **std::optional<T> opt** drží hodnotu ***opt** (taky **opt.value()**), a nebo **std::nullopt**
    - Podle **if(opt.has_value())** nebo **if(opt)**

```
void process_arguments() {
    std::optional<int> parallel_jobs = ...;

    run_threads(parallel_jobs.value_or(1))
}
```

```
std::optional<int> square_root(int i) {
    if (i < 0) return {};
    else return std::sqrt(i);
}

std::optional<int> inverse(int i) {
    if (i == 0) return {};
    else return 1 / i;
}

std::optional(42)
   .and_then(square_root)
   .and_then(inverse);
```

https://en.cppreference.com/w/cpp/utility/optional

std::expected<Value, Error>

- Skoro to samé jako **std::optional<Value>**
  - Ale místo **std::nullopt** máme **std::unexpected<Error>**

```
enum class parse_error
{
    invalid_input,
    overflow
};

auto parse_number(std::string_view& str) -> std::expected<double, parse_error>
{
    const char* begin = str.data();
    char* end;
    double retval = std::strtod(begin, &end);

    if (begin == end)
        return std::unexpected(parse_error::invalid_input);
    else if (std::isinf(retval))
        return std::unexpected(parse_error::overflow);

    str.remove_prefix(end - begin);
    return retval;
}
```

https://en.cppreference.com/w/cpp/utility/expected