NPRG 041 – cvičení 8
Programování v C++

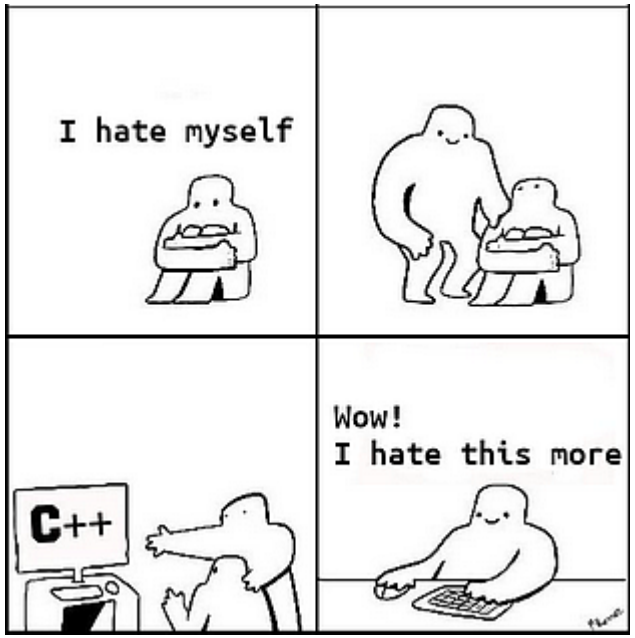
Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

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

Agenda

- **Sedmá úloha: Interpret Výrazů**



Statický polymorfismus - šablony

Budu používat slova „šablona“ a „template“ zaměnitelně

C++ templates

```
template<typename TypePar>
TypePar function_template(TypePar par) {
    return par;
}

template<typename TypePar>
struct ClassTemplate {
    TypePar value;
}

template<typename TPar>
using alias_template = std::map<int, TPar>;

template<typename TPar>
constexpr TPar default_value = {};
```

- Dovolují napsat:
  - Funkci
  - Třídu
  - Alias
  - Konstantu

  Pro víc typů
- **Instanciace**
  - Když použijeme s konkrétními typy (zavoláme `function_template(5)`)
    - Explicitně: `function_template<int>`, `ClassTemplate<std::string>`
  - **Compiler** za nás napíše kopii té definice s dosazenými parametry:
    - **Musí být viditelná celá definice**

```
int __function_template_i(int par) {
    return par;
}
```

C++ templates

- Dovolují napsat:
  - Funkci
  - Třídu
  - Alias
  - Konstantu

  Pro víc typů
- **Instanciace**
  - Když použijeme s konkrétními typy (zavoláme `function_template(5)`)
    - Explicitně: `function_template<int>`, `ClassTemplate<std::string>`
  - **Compiler** za nás napíše kopii té definice s dosazenými parametry:
    - **Musí být viditelná celá definice**

```
template<class TypePar>
TypePar function_template(TypePar par) {
    return par;
}

template<class TypePar>
struct ClassTemplate {
    TypePar value;
}

template<class TPar>
using alias_template = std::map<int, TPar>;

template<class TPar>
constexpr TPar default_value = {};
```

class má zde stejný význam jako typename

```
int __function_template_i(int par) {
    return par;
}
```

Otemplatované třídy definujeme celé naráz

Non-template

```
// class.hpp

class Class {
public:
    Class() : field1(0), field2(0) {}
    void method();
private:
    int field1, field2;
}
```

```
// class.cpp

Class::method() {
    // long body
}
```

díky inline lze includovat do různých .cpp

Template

```
// class.hpp

template<typename T, typename U>
class Class {
public:
    Class() : field1(0), field2(0) {}
    void method();
private:
    T field1; U field2;
}
// inline = same as in other modules
template<typename T, typename U>
inline Class<T, U>::method() {
    // long body
}
```

Celá definovaná v jednom souboru

Typy fieldy podle parametrů

Musíme připomenout typové parametry

Implicitní templatové argumenty

Následující template s explicitními typovými parametry

```
template<typename TypePar>
TypePar function_template(TypePar par) {
    return par;
}
```

můžeme napsat mnohem jednodušeji

```
auto function_template(auto par) {
    return par;
}
```

Compiler tento zjednodušený zápis chápe stejně jako ten explicitní

Konkrétnější overload má přednost před templatem

```
void foo(int i) {
    std::cout << "int: " << i << std::endl;
}
template<typename T>
void foo(T a) {
    std::cout << "generic: " << a << std::endl;
}
void foo(float f) {
    std::cout << "float: " << f << std::endl;
}

int main() {
    foo(5);
    foo("hi");
    foo(.5f);
    foo(.5); // is not float
}
```

Output:

```
int: 5
generic: hi
float: 0.5
generic: 0.5
```

- Pořadí funkcí zde nehraje roli
- Template má ale přednost před implicitními casty

Non-type templatové argumenty

- Argumentem templatu může být i třeba číslo
  - Musí být compile-time konstanta
- Užitečné, pokud implementace třídy závisí na konkrétním počtu
- Jinak to není ničím moc speciální

```
template<std::size_t N>
struct PackedInt {
    int &operator[](std::size_t n) {
        return values[n];
    }
    const int &operator[](std::size_t n) const {
        return values[n];
    }

    std::array<int, N> values;
};
```

Subscript operátor
Definuje indexaci

Verze pro readonly
PackedInt

Dynamický vs Statický polymorfismus

Dynamický polymorfismus

- Definujeme abstraktní třídu
  - a pak konkrétní třídy, které ji dědí
- **Runtimový overhead:**
  - Volání virtuálních funkcí
  - Polymorfní třídy mají vpointery
- **Při překladu nemusíme vědět konkrétní typ každého objektu**
  - Ale vyžadujeme příbuznost

Statický polymorfismus

- Píšeme definici s type parametry
  - Instanciace s konkrétními typy
- **„Žádný runtimový overhead“**
  - Vygenerování spousty strojového kódu
- **Za překladu musíme znát každý typ**
  - Nevyžadujeme příbuznost

Kombinace statického a dynamického polymorfismu

```
class AbstractValue {
public:
    using ptr = unique_ptr<AbstractValue>;
    virtual ~AbstractValue() = default;
    virtual void print(ostream& out) const = 0;
    friend ostream& operator<<(ostream& out,
                               const AbstractValue& val) {
        val.print(out); return out;
    }
};
template<class T>
class ConcreteValue : public AbstractValue {
public:
    explicit ConcreteValue(T val) : val_(val) {}
    void print(ostream & out) const override {
        out << val_;
    }
private:
    T val_;
};
```

Type erasure

- Design pattern
- Odstranění detailů o polymorfismu z kódu

```
int main() {
    vector<Value> values;

    values.push_back(5);
    values.push_back("hi");
    values.push_back(.5f);

    for (auto&& value : values) {
        cout << value << endl;
    }
}
```

```
class Value {
    class AbstractValue {
    public:
        using ptr = unique_ptr<AbstractValue>;
        virtual ~AbstractValue() = default;
        ...
    };

    template<class T>
    class ConcreteValue : public AbstractValue {
    ...
    private:
        T val_;
    };

public:
    template<class T>
    Value(T t) : val_(make_unique<ConcreteValue<T>>(t)) {}
    ...
private:
    AbstractValue::ptr val_;
};
```

std:variant a std::visit

#include <variant>

- Kombinuje výhody a nevýhody dynamického i statického polymorfismu
- **Za překladu nemusíme znát konkrétní typy**
  - **Typy nemusí být příbuzné**
- **Hodnoty nemusí být za pointerem**
  - **Efektivitnější než dyn. polymorfismus**
  - **Méně efektivní než statický**
- **Ale musíme vypsat všechny přípustné typy**

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);
}
```

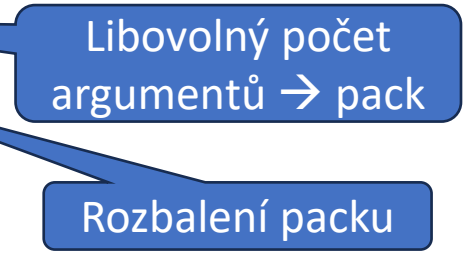
Reprezentuje různé funkce

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

Overload pattern

```
template<class... Ts>
struct overload : Ts... {
  using Ts::operator()...;
};

int main() {
  using item_type =
    variant<int, const char*, float>;
  vector<item_type> items{5, "hi", .5f};
  for (auto&& item : items)
    visit(overload {
      [](int i) {
        cout << "int: " << i << endl; },
      [](auto a){
        cout << "generic: " << a << endl; },
      [](float f){
         cout << "float: " << f << endl; }
    }, item);
}
```




```
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);
}
```

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>

```
#include <optional>
```

- Užitečný pro typy, které nemají ekvivalent nullu
- Příklady vytvoření:
  - empty: **std::optional<T> opt** nebo **std::optional<T> opt = {}**
  - s hodnotou: **std::optional<T> opt = value**
- Získání hodnoty přes dereferenci ***opt** nebo **.value**
- Zjištění, jestli má hodnotu: **if(opt)** nebo **if (opt.has_value())**
- Podporuje **operator=** a **emplace** pro update
- Podporuje **reset** a **= nullopt** pro vynulování

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

std::any

#include <any>

- Může obsahovat skoro jakýkoliv typ
- Používá nějakou implementaci type erasure
- Přívětivost není moc velká
  - Hodnotu musíme získat přes **any_cast**
  - Hází **výjimku**, pokud se spleteme nebo **nullptr**, používáme-li pointery: `std::any_cast<float>(&a)`
- Použití: **pokud to nejde jinak**
  - Typicky tam, kde by se v C použil void*
  - Třeba management zpráv

```
// any type
any a = 1;
cout << any_cast<int>(a) << '\n';
a = 3.14;
cout << any_cast<double>(a) << '\n';

// bad cast
try {
    a = 1;
    cout << std::any_cast<float>(a) << '\n';
} catch (const std::bad_any_cast& e) {
    cout << e.what() << '\n';
}
```

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

Constexpr

- Už víme, že u proměnných constexpr znamená, že ta proměnná je compile-time konstanta
- U funkcí to znamená něco trochu jiného: **“lze ji provést za kompilace“**

```
constexpr auto foo(auto i) {
    return i * 2;
}
```

  - Její výsledek můžeme uložit do compile-time konstanty
  - Ale stále ji lze aplikovat i na runtimové hodnoty → pak se provede za runtimu
- Co když ji určitě chceme provést za kompilace? → **consteval**
- Constexpr i consteval funkce mají spoustu omezení