NPRG 041 – cvičení 10
Programování v C++

Jiří Klepl

mail:
klepl@d3s.mff.cuni.cz

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

Bonusová témata

Ranges, Concepts
Budoucnost C++?

Ranges

#include <ranges>

- Rozšíření a zobecnění algoritmů+iterátorů (a sekvenčních kontejnerů)
- Range ≡ **[begin, end)** – typický range (třeba vector)
  - nebo **begin + [0, size)** – range určitého rozsahu
  - nebo **[begin, pred)** – range zakončený podmínkou
  - nebo **[begin, …)** – nekonečný range

https://en.cppreference.com/w/cpp/algorithm/ranges

- Algoritmy s argumety (begin, end, …) mají ranges:: variantu pro range
- Adaptéry: např. **take_view**, **zip_view** (aka. **views::take**, **views::zip**)
  - Vytvoří nový pohled (nějak upravený) na již existující range
  - Řetězení pomocí operátoru **|**
    for(auto&& item : range | std::views::filter(even) | std::views::take(5))

Zpět k templatům

```cpp
template<class T>
class Class {
public:
    Class(T t) : t_(t) {}
    // does not offer a reset() function
private:
    T t_;
};
```

Primární template

Ten nejobecnější default, pokud neexistuje specifičtější

```cpp
template<class T>
class Class<T*> {
public:
    Class(T* t) : t_(t) {}
    void reset() { t_ = nullptr; }
private:
    T* t_;
};
```

```cpp
template<>
class Class<int> {
public:
    Class(int i) : i_(i) {}
    void reset() { i_ = 0; }
private:
    int i_;
};
```

```
template<class T>
class Class {
public:
    Class(T t) : t_(t) {}
    // does not offer a reset() function
private:
    T t_;
};
```

Primární template

Ten nejobecnější default, pokud neexistuje specifičtější

```
template<>
class Class<int> {
public:
    Class(int i) : i_(i) {}
    void reset() { i_ = 0; }
private:
    int i_;
};
```

```
template<class T>
class Class<T*> {
public:
    Class(T* t) : t_(t) {}
    void reset() { t_ = nullptr; }
private:
    T* t_;
};
```

Parciální specializace

Specifičtější definice pro nějaké typy odpovídající patternu (zde T*)

```
template<class T>
class Class {
public:
    Class(T t) : t_(t) {}
    // does not offer a reset() function
private:
    T t_;
};
```

Primární template

Ten nejobecnější default, pokud neexistuje specifičtější

```
template<class T>
class Class<T*> {
public:
    Class(T* t) : t_(t) {}
    void reset() { t_ = nullptr; }
private:
    T* t_;
};
```

Parciální specializace

Specifičtější definice pro nějaké typy odpovídající patternu (zde T*)

Prázdné parametry

```
template<>
class Class<int> {
public:
    Class(int i) : i_(i) {}
    void reset() { i_ = 0; }
private:
    int i_;
};
```

Explicitní specializace

Definice pro konkrétní typové parametry (zde int)

```
template<class T>
class Class {
public:
    Class(T t) : t_(t) {}
    // does not offer a reset() function
private:
    T t_;
};
```

Primární template

Ten nejobecnější default, pokud neexistuje specifičtější

```
template<class T>
class Class<T*> {
public:
    Class(T* t) : t_(t) {}
    void reset() { t_ = nullptr; }
private:
    T* t_;
};
```

Parciální specializace

Specifičtější definice pro nějaké typy odpovídající patternu (zde T*)

Prázdné parametry

```
template<>
class Class<int> {
public:
    Class(int i) : i_(i) {}
    void reset() { i_ = 0; }
private:
    int i_;
};
```

Explicitní specializace

Definice pro konkrétní typové parametry (zde int)

```
template class Class<float>;
```

Explicitní instanciace

Prostě vyžádá výrobu podle zadaných parametrů

Pokročilejší templaty

Templaty na steroidech (jen ukázka)

Definice linked-listu

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

```
int main() {
  using L = Cons<int,
            Cons<double,
             Cons<char,
              Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

Templaty na steroidech (jen ukázka)

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

Definice linked-listu

Deklarace implementace "funkce" member

"Funkce" member

```
int main() {
  using L = Cons<int,
                 Cons<double,
                  Cons<char,
                   Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

Templaty na steroidech (jen ukázka)

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

Definice linked-listu

Deklarace implementace "funkce" member

"Funkce" member

List začíná hledaným itemem

```
int main() {
  using L = Cons<int,
                 Cons<double,
                  Cons<char,
                   Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

Templaty na steroidech (jen ukázka)

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

Definice linked-listu

Deklarace implementace "funkce" member

"Funkce" member

List začíná hledaným itemem

List nezačíná hledaným itemem

Posouváme se na další item

```
int main() {
  using L = Cons<int,
            Cons<double,
             Cons<char,
              Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

Templaty na steroidech (jen ukázka)

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

Definice linked-listu

Deklarace implementace "funkce" member

"Funkce" member

List začíná hledaným itemem

List nezačíná hledaným itemem

Empty list ☹

Posouváme se na další item

```
int main() {
  using L = Cons<int,
            Cons<double,
             Cons<char,
              Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

Templaty na steroidech (jen ukázka)

```
template<class Item, class Next>
struct Cons {}; // a list node
struct Nil; // empty list

template<class, class>
struct member_impl; // the result stored in `value`

template<class List, class Item>
constexpr bool member = member_impl<List, Item>::value;

template<class Item, class Next>
struct member_impl<Cons<Item, Next>, Item> {
    static constexpr bool value = true;
};

template<class Item, class Next, class T>
struct member_impl<Cons<Item, Next>, T> {
    static constexpr bool value = member_impl<Next, T>::value;
};

template<class T>
struct member_impl<Nil, T> {
    static constexpr bool value = false;
};
```

Definice linked-listu

Deklarace implementace "funkce" member

"Funkce" member

List začíná hledaným itemem

List nezačíná hledaným itemem

Empty list ☹

Posouváme se na další item

```
int main() {
  using L = Cons<int,
            Cons<double,
             Cons<char,
              Nil>>>;
  static_assert(member<L, int>);
  static_assert(member<L, double>);
  static_assert(member<L, char>);
  static_assert(!member<L, bool>);
}
```

To podstatné:
Specializace class templatů

Intermezzo: functions with trailing return type

```
auto foo() -> int {
    return 42;
}
```

Return na konci deklarace funkce

```
class Expression {
public:
    using pointer = std::unique_ptr<Expression>;
    virtual ~Expression() {}
    virtual pointer eval() const = 0;
};

auto Expression::eval() const -> pointer {
    return nullptr;
}
```

Nemusíme psát Expression::

+ řada dalších výhod
(můžeme v typu používat argumenty)

SFINAE (ukázka)

SUBSTITUTION
FAILURE
IS
NOT
AN
ERROR

= když se nám něco nepodaří odvodit, zkusíme další overload (u funkcí) nebo specializaci (u tříd)

Oba templaty get_name jsou stejně obecné takže by měly být ambiguous, ale…

```
struct Person {
    std::string name;
};

template<class T>
auto get_name(const T& t) -> const decltype(t.name)& {
    return t.name;
}

template<class T>
auto get_name(const T* t) -> const decltype(t->name)& {
    return t->name;
}

int main() {
    Person p;
    p.name = "John";
    std::cout << get_name(p) << std::endl;
    std::cout << get_name(&p) << std::endl;
    return 0;
}
```

SFINAE (ukázka)

SUBSTITUTION
FAILURE
IS
NOT
AN
ERROR

= když se nám něco nepodaří odvodit, zkusíme další overload (u funkcí) nebo specializaci (u tříd)

```cpp
struct Person {
    std::string name;
};

template<class T>
auto get_name(const T& t) -> const decltype(t.name)& {
    return t.name;
}

template<class T>
auto get_name(const T* t) -> const decltype(t->name)& {
    return t->name;
}

int main() {
    Person p;
    p.name = "John";
    std::cout << get_name(p) << std::endl;
    std::cout << get_name(&p) << std::endl;
    return 0;
}
```

Oba templaty get_name jsou stejně obecné takže by měly být ambiguous, ale…

Podaří se nám odvodit, když předáme Person& (p)

SFINAE (ukázka)

SUBSTITUTION
FAILURE
IS
NOT
AN
ERROR

= když se nám něco nepodaří odvodit, zkusíme další overload (u funkcí) nebo specializaci (u tříd)

```cpp
struct Person {
    std::string name;
};

template<class T>
auto get_name(const T& t) -> const decltype(t.name)& {
    return t.name;
}

template<class T>
auto get_name(const T* t) -> const decltype(t->name)& {
    return t->name;
}

int main() {
    Person p;
    p.name = "John";
    std::cout << get_name(p) << std::endl;
    std::cout << get_name(&p) << std::endl;
    return 0;
}
```

Oba templaty get_name jsou stejně obecné takže by měly být ambiguous, ale…

Podaří se nám odvodit, když předáme Person& (p)

Podaří se nám odvodit, když předáme Person* (&p)

SFINAE (ukázka)

SUBSTITUTION
FAILURE
IS
NOT
AN
ERROR

= když se nám něco nepodaří odvodit, zkusíme další overload (u funkcí) nebo specializaci (u tříd)

```cpp
struct Person {
    std::string name;
};

template<class T>
auto get_name(const T& t) -> const decltype(t.name)& {
    return t.name;
}

template<class T>
auto get_name(const T* t) -> const decltype(t->name)& {
    return t->name;
}

int main() {
    Person p;
    p.name = "John";
    std::cout << get_name(p) << std::endl;
    std::cout << get_name(&p) << std::endl;
    return 0;
}
```

Oba templaty get_name jsou stejně obecné takže by měly být ambiguous, ale…

Podaří se nám odvodit, když předáme Person& (p)

Podaří se nám odvodit, když předáme Person* (&p)

V obou případech se nám podaří odvodit jen jeden → je to jednoznačné

SFINAE (ukázka)

SUBSTITUTION
FAILURE
IS
NOT
AN
ERROR

= když se nám něco nepodaří odvoc
zkusíme další overload (u funkcí) ne
specializaci (u tříd)

```
struct Person {
    std::string name;
};

template<class T>
auto get_name(const T& t) -> const decltype(t.name)& {
    return t.name;
}

t) -> const decltype(t->name)& {

int main() {
    Person p;
    p.name = "John";
    std::cout << get_name(p) << std::endl;
    std::cout << get_name(&p) << std::endl;
    return 0;
}
```

Oba templaty get_name
jsou stejně obecné
takže by měly být ambiguous, ale…

Podaří se nám
odvodit, když
předáme Person& (p)

Složité a generuje nekonečné errory → vysype popis každého substitution failure

Podaří se nám
odvodit, když
předáme Person* (&p)

V obou případech se nám
podaří odvodit jen jeden
→ je to jednoznačné

Nový C++ koncept: concepts

```
struct Person { std::string name; };

template<class T>
concept HasName = requires(T t) { t.name; };

template<class T>
const std::string& get_name(const T& t) requires HasName<T> {
    return t.name;
}
template<class T>
const std::string& get_name(const T* t) requires HasName<T> {
    return t->name;
}

…

Person p;
p.name = "John";
std::cout << get_name(p) << std::endl;
std::cout << get_name(&p) << std::endl;
```

Koncept
= named requirement

Nový C++ koncept: concepts

```
struct Person { std::string name; };

template<class T>
concept HasName = requires(T t) { t.name; };

template<class T>
const std::string& get_name(const T& t) requires HasName<T> {
    return t.name;
}
template<class T>
const std::string& get_name(const T* t) requires HasName<T> {
    return t->name;
}

…

Person p;
p.name = "John";
std::cout << get_name(p) << std::endl;
std::cout << get_name(&p) << std::endl;
```

Koncept
= named requirement

Pro každý objekt t typu T musí jít napsat „t.name“

Nový C++ koncept: concepts

```
struct Person { std::string name; };

template<class T>
concept HasName = requires(T t) { t.name; };

template<class T>
const std::string& get_name(const T& t) requires HasName<T> {
    return t.name;
}
template<class T>
const std::string& get_name(const T* t) requires HasName<T> {
    return t->name;
}

…

Person p;
p.name = "John";
std::cout << get_name(p) << std::endl;
std::cout << get_name(&p) << std::endl;
```

Koncept
= named requirement

Pro každý objekt t typu T musí jít napsat „t.name“

Funkce vyžadující koncept

Nový C++ koncept: concepts

```
struct Person { std::string name; };

template<class T>
concept HasName = requires(T t) { t.name; };

template<class T>
const std::string& get_name(const T& t) requires HasName<T> {
    return t.name;
}
template<class T>
const std::string& get_name(const T* t) requires HasName<T> {
    return t->name;
}

…

Person p;
p.name = "John";
std::cout << get_name(p) << std::endl;
std::cout << get_name(&p) << std::endl;
```

Koncept
= named requirement

Pro každý objekt t typu T musí jít napsat „t.name“

Funkce vyžadující koncept

Jednodušší a čitelnější než ekvivalentní SFINAE, produkuje kratší a srozumitelnější errory

Nový C++ koncept: concepts

```
struct Person { std::string name; };

template<class T>
concept HasName = requires(T t) { t.name; };

template<HasName T>
const std::string& get_name(const T& t) {
    return t.name;
}
template<HasName T>
const std::string& get_name(const T* t) {
    return t->name;
}

…

Person p;
p.name = "John";
std::cout << get_name(p) << std::endl;
std::cout << get_name(&p) << std::endl;
```

Concepty můžeme psát přímo jako „nadtypy“ typových parametrů

Concepts library

#include <concepts>

```
#include <concepts>
#include <iostream>

void print(std::integral auto i)
{
    std::cout << "Integral: " << i << '\n';
}

void print(auto x)
{
    std::cout << "Non-integral: " << x << '\n';
}
```

Koncept na auto parametru