NPRG041 – C++

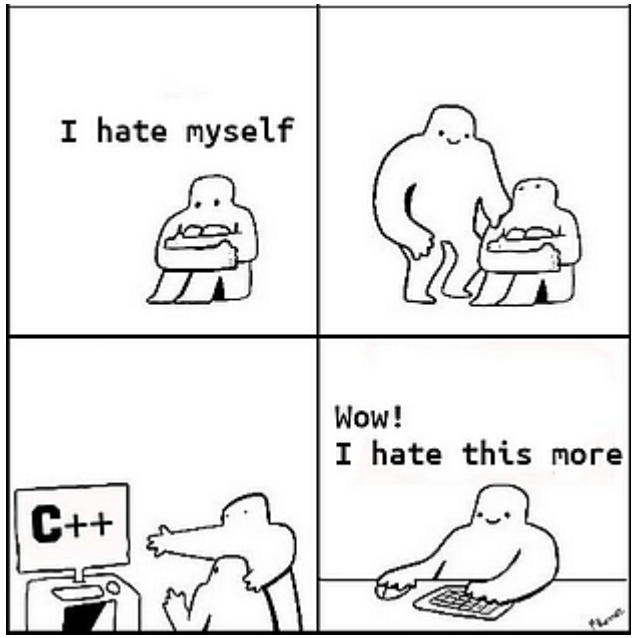
cvičení – Jiří Klepl

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

Klepl@d3s.mff.cuni.cz

Agenda

- Šablony



C++ templates

- Function templates

```
template<typename T> T add(const T& a, const T& b);
```

- Class templates

```
template<typename F, typename S> struct Pair { F first;
                                               S second; };
```

- Alias templates

```
template<typename T> using Counter = std::unordered_map<T,int>;
```

- Variable templates

```
template<int Num> constexpr int why_not = Num;
```

- (Concepts)

Použití šablon

compiler ze šablon udělá instanci s konkrétními typy

Explicitně dosazený typ

```
Counter<std::string> items{
    {"rohlik", 4},
    {"houska", 3},
    {"chleba", 1},
    {"uran", 42},
};
```

Dedukce typu (T = int)

```
auto sum = add(42, 37);
```

```
template<class T, class S>
void foo(T t, S s);

// …

foo<double>(12, 'x'); // foo<double, char>
```

Kombinace obojího

Použití šablon

compiler ze šablon udělá instanci s konkrétními typy

Explicitně dosazený typ

```
Counter<std::string> items{
    {"rohlik", 4},
    {"houska", 3},
    {"chleba", 1},
    {"uran", 42},
};
```

Dedukce typu (T = int)

```
auto sum = add(42, 37);
```

```
int _Z3addIiE(int, int);
```

```
template<class T, class S>
void foo(T t, S s);

// …

foo<double>(12, 'x'); // foo<double, char>
```

Kombinace obojího

Otemplatované třídy často definujeme celé v jednom souboru

Non-template

```hpp
// class.hpp

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

```cpp
// class.cpp

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

Problematický template

```hpp
// class.hpp
template<typename T, typename U>
class Class {
public:
    Class() : field1(0), field2(0) {}
    void method();
private:
    T field1; U field2;
}
```

```cpp
// class.cpp
template<typename T, typename U>
inline Class<T, U>::method() {
    // long body
}
```

Otemplatované třídy často definujeme celé v jednom souboru

Non-template

```
// class.hpp

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

```
// class.cpp

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

Problematický templ

Uživatel (includující) sice umí vyrobit třídu

```
// class.hpp
template<typename T, typename U>
class Class {
public:
    Class() : field1(0), field2(0) {}
    void method();
private:
    T field1; U field2;
}
```

```
// class.cpp
template<typename T, typename U>
inline Class<T, U>::method() {
    // long body
}
```

Otemplatované třídy často definujeme celé v jednom souboru

Non-template

```
// class.hpp

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

```
// class.cpp

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

Problematický templ

Uživatel (includující) sice umí vyrobit třídu

```
// class.hpp
template<typename T, typename U>
class Class {
public:
    Class() : field1(0), field2(0) {}
    void method();
private:
    T field1; U field2;
}
```

Ale neumí vytvořit metodu

```
// class.cpp
template<typename T, typename U>
inline Class<T, U>::method() {
    // long body
}
```

Otemplatované třídy často definujeme celé v jednom souboru

Non-template

```hpp
// class.hpp

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

```cpp
// class.cpp

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

Neproblematický template (celý v hpp)

```hpp
// 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
}
```

„header-only“

⚠ Je potřeba zde použít inline

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
}
```

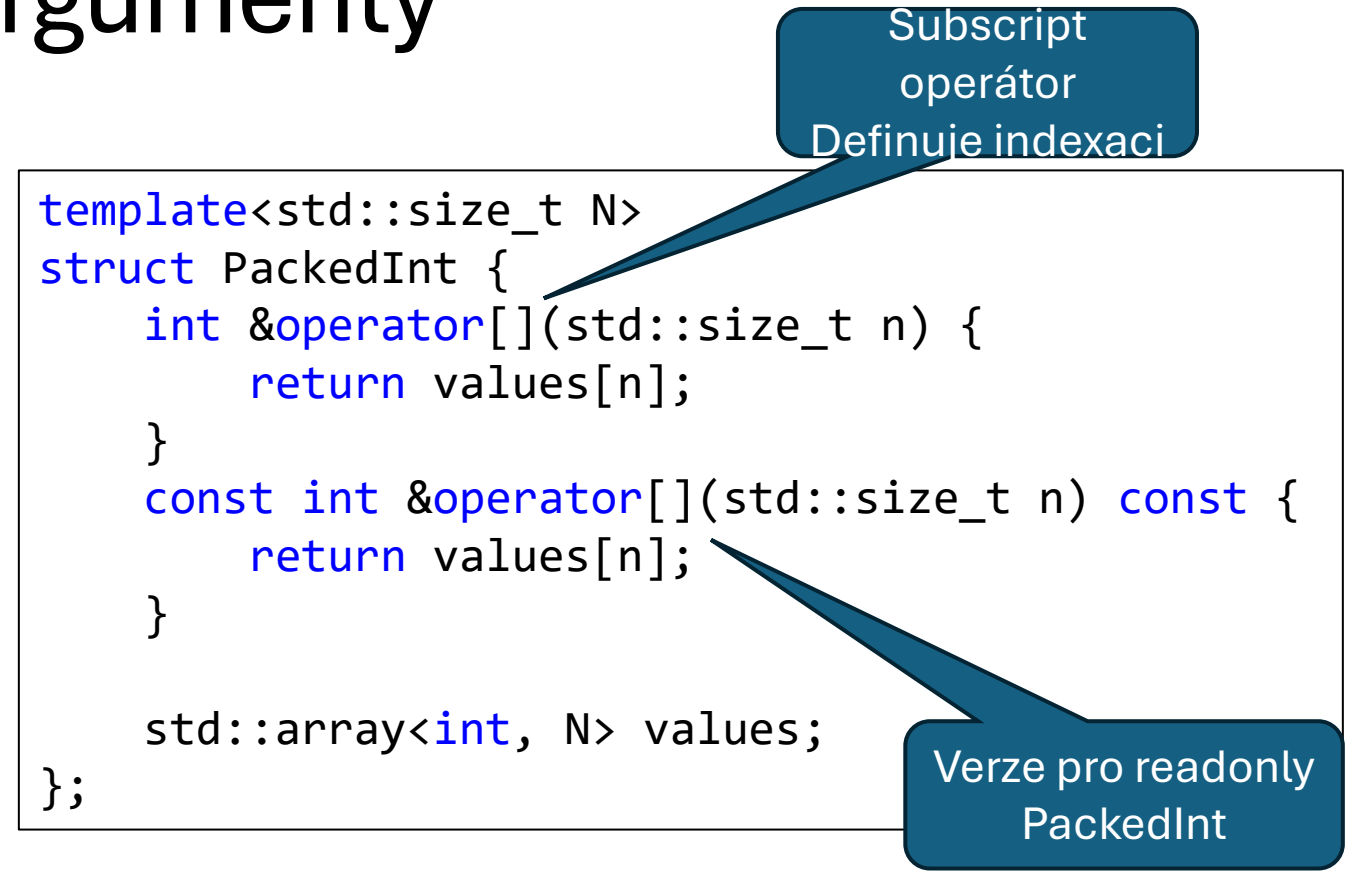
Output:

```
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



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_;
};
```

Příklad: Fraction<T>

- Máme typ **T**, který podporuje **+** a ***
- Chceme definovat typ **Fraction<T>**, který podporuje sčítání, odečítání, násobení a dělení s dalším **Fraction<T>** podle pravidel s operacemi na zlomcích
- Chceme definovat srovnávání podle **<**
- **Fraction<T>** musí podporovat přiřazování, kopírování, movování, atd. (pokud je podporuje **T**)

- Uživatel si bude moct includnout header s implementací **Fraction<T>** a volně jej používat

Šablony - nejen typ dat

- lze využít všech vlastností typu
  - duck typing - kompatibila
  - kontrola kompilátorem

```
template<typename T> class Matrix {
private:
  vector<T> data;
  size_t M, N;
public:
  T operator[]( size_t i, size_t j)
    { return data[i*M+j]; }
};
```

uložení po řádcích

```
template<typename T> class C {
  T val_;
};
```

jakýkoliv typ

```
template<typename T> class C {
  void f() { T::f(); }
};
```

lze volat metody

- umožnit zvolit reprezentaci
  - compile-time
  - efektivita - no overhead
  - virtuální metody 〰 režie

```
template<typename T, class L=RowWise>
class Matrix {
  ....
  T operator[]( size_t i, size_t j)
    { return data[ L::offset(i,j,M,N) ]; }
};
```

- policy classes, traits, type deduction, FTAD/CTAD, generic programming, SFINAE, ..., ...
  - ↦ Advanced C++

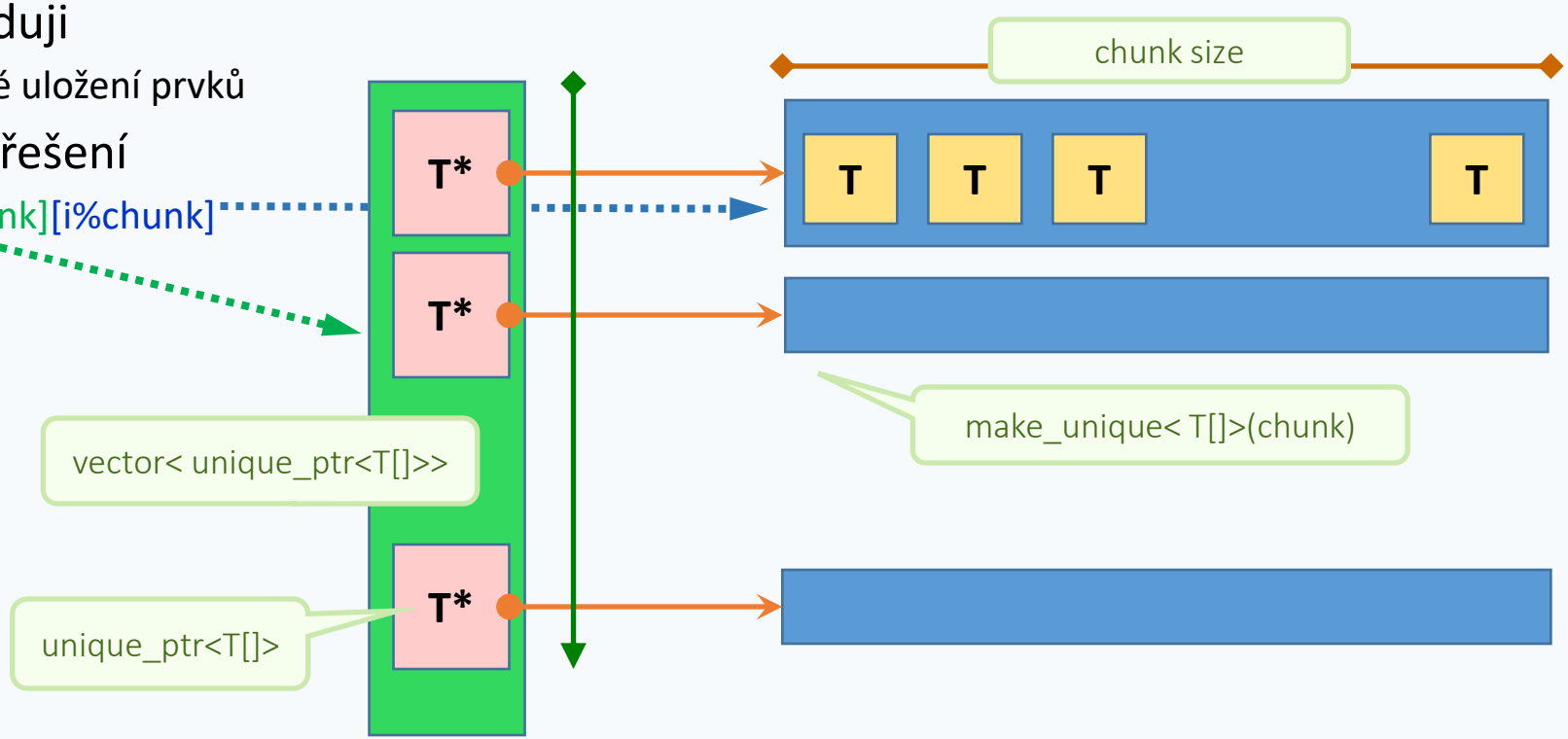
```
class ColumnWise {
public:
  static size_t offset( size_t i, j, m, n)
    { return j*n + i; }
};

Matrix<int, ColumnWise> m;
```

uložení po sloupcích

Gumové pole

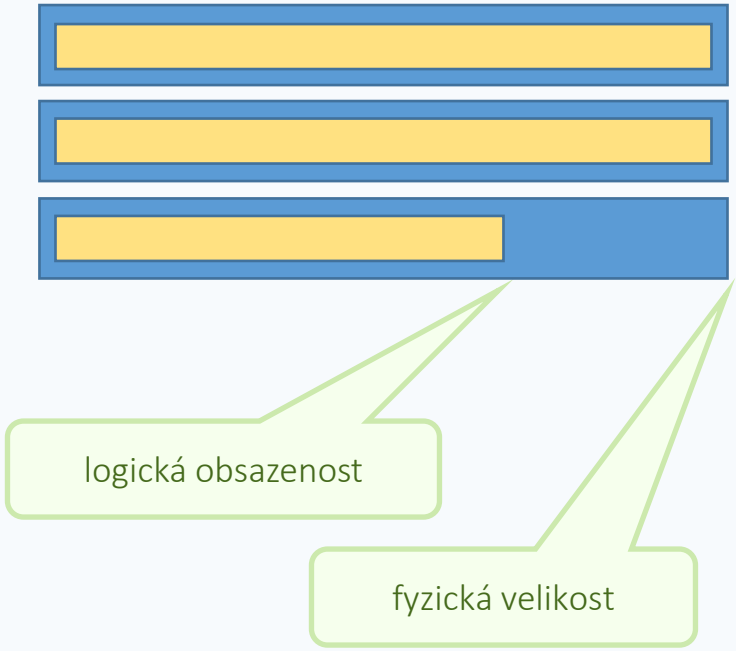
- problém
  - std::vector nezachovává umístění prvků
  - přidání → invalidace referencí, iterátorů, ...
  - vynuceno požadavkem na spojité uložení prvků
- chci
  - datová struktura zachovávající umístění
  - žádné invalidace
  - konstantní časová složitost přístupu k prvkům
- nevyžaduji
  - spojité uložení prvků
- možné řešení
  - [i/chunk][i%chunk]

x.push_back(n)
x[i]



Gumové pole - deklarace

```
template<typename T> class Pole {
public:
  Pole( size_t chunk = 100) : .... {}
  void push_back( const T& x);
  T& operator[] ( size_t i) { return ..; }
  T& at( size_t i) { check(i); return ..; }

private:
  void check( size_t i);
  void resize( size_t i);
  ....
  vector< unique_ptr<T[]>> hrabe_;
};
```



Gumové pole - iterator

```
template<typename T> class Pole {
public:
  void push_back( const T& x);
  T& operator[] ( size_t i);

  iterator begin() { return iterator{..}; }
  .... end() { return ....; }
private:
  ....
};
```

svázání iteratoru s instancí kontejneru

- iterator:
  - * - dereference prvku
  - ++ - inkrementace
- end()
  - různý od ∀ platných iterátorů
    - … i kdykoliv v budoucnosti!
  - nemusí být iterator
    - přetížený operator !=
  - na posledním prvku musí platit ++it == end()

sentinel

stejně jako std:: iterátory

```
Pole<xyz>::iterator it1;
auto it2 = p.begin();
```

```
for( auto it = p.begin();
  it != p.end(); ++it) ....
```

nemusí být stejné typy

Pole::iterator

default konstruktor
Pole<T>::iterator it;
it = p.begin();

```
class iterator {
  iterator() : .... {}
  iterator( const iterator& it) : .... {}
  iterator( ....) : .... {}
  T& operator* () { return .... }
  bool operator != ( ....) { return ....; }
  iterator operator++ () { ....; return *this; }
private:
  ....
};
```

???

Gumové pole - kopie, implementace

- základ implementace
  - x.push_back(n)
  - x[i]
- iterator
  - operator * != ++
    - ++ na posledním prvku
  - begin(), end()
    - end() -> iterator
    - operator !=
  - for(auto&& i : v) {}
- operace s kontejnerem
  - liší se od default?
    - copy-constructible
    - move-constructible
    - assignable (=)
- řádně otestujte
  - všechny funkce / kombinace
  - okrajové případy
- k rozmyšlení
  - bez default konstruktoru
  - const_iterator
  - Pole<unique_ptr<T>>

```
Pole<T> a;
....
Pole<T> b = a;
```