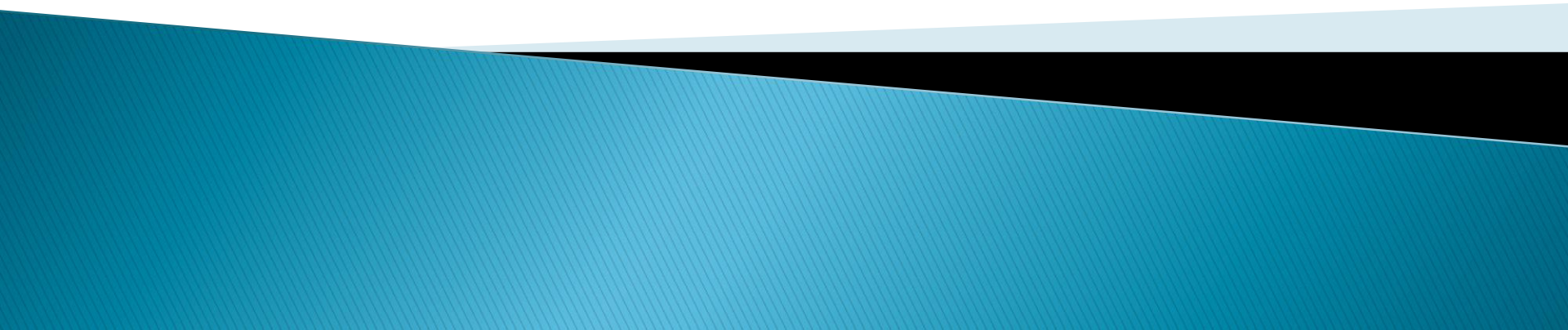


NPRG051

Pokročilé programování v C++

Domácí úkoly 2013/14



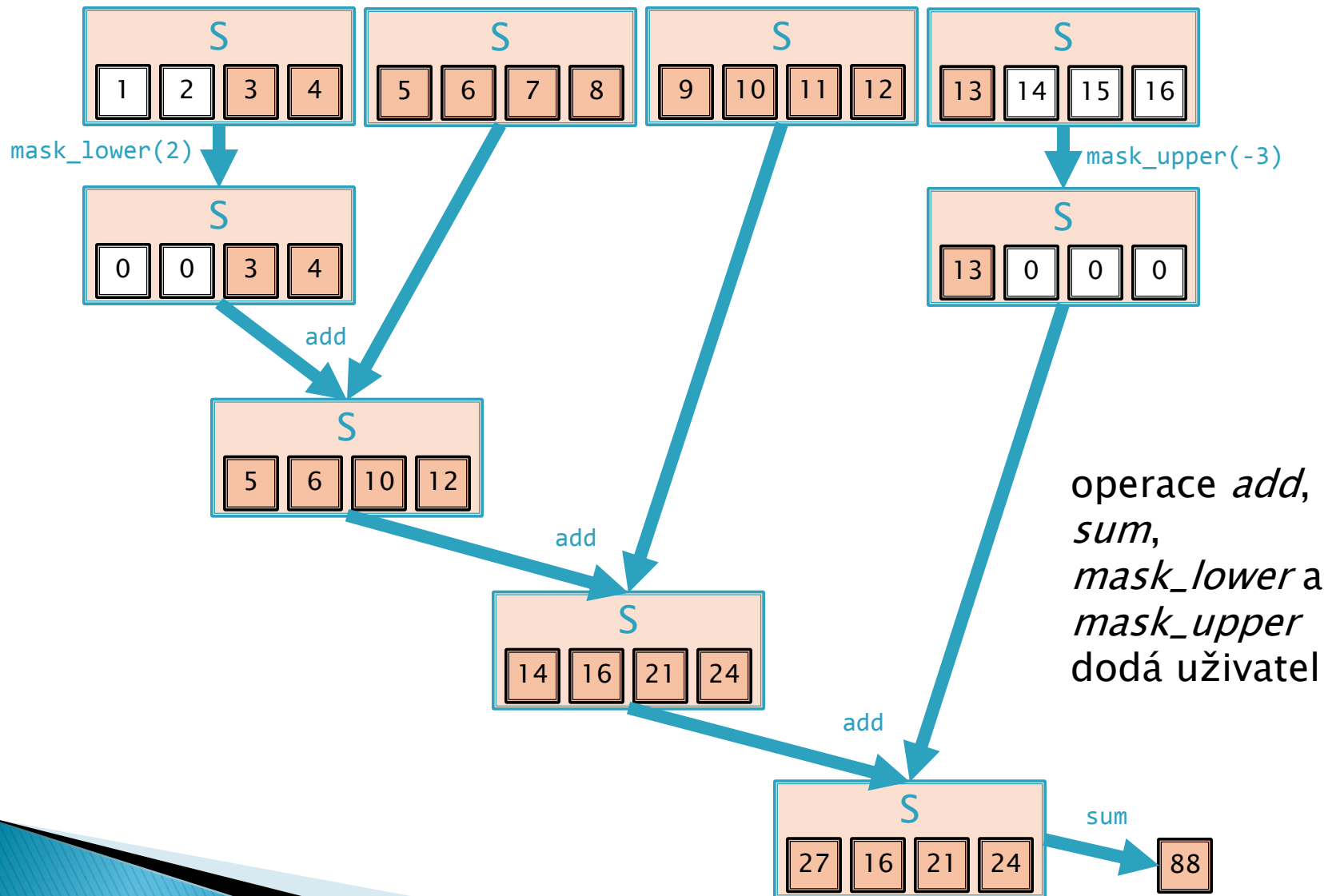
DÚ 1

SIMD kontejner

DÚ 1 – SIMD kontejner

- ▶ kontejner odpovídající poli s podporou SIMD operací
 - Single Instruction Multiple Data
 - `simd_vector< T, S>`
- ▶ `T` = logický prvek kontejneru
 - pouze jednoduchý datový typ (bez konstruktorů atp.)
- ▶ `S` = typ reprezentující `K`-tici prvků typu `T`
 - `K = sizeof(S)/sizeof(T)`
 - obvykle mocnina dvojky
 - tentýž blok paměti lze korektně chápat jako:
 - jako pole `T[K*N]` i jako pole `S[N]`
 - tj. reinterpret_cast mezi `S*` a `T*` je korektní
- ▶ kontejner má být zpřístupněn pomocí dvou druhů iterátorů
 - jeden s granularitou `T`, druhý s granularitou `S`
 - při přístupu přes druhý iterátor lze aplikovat **vektorové operace** nad `S`

Příklad – sečtení prvků ve výřezu



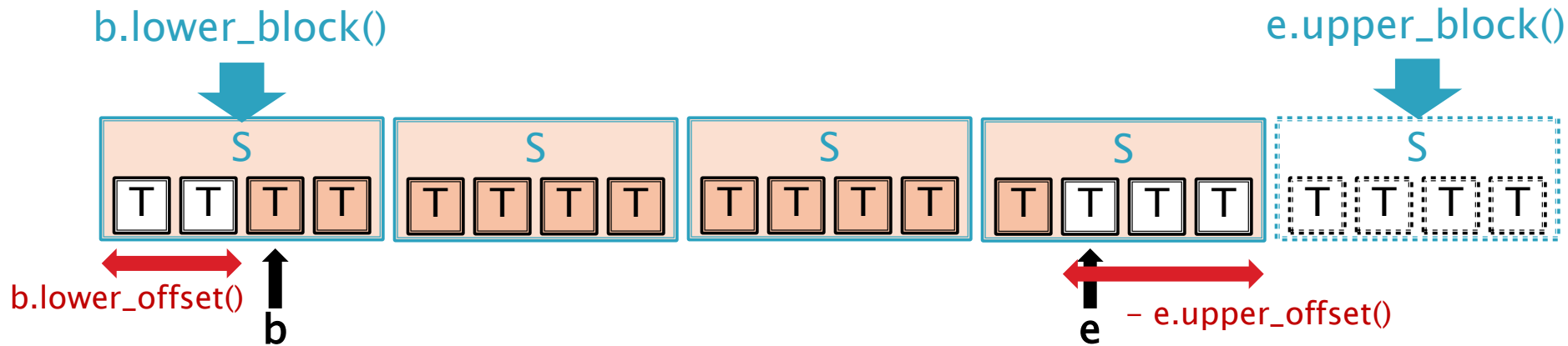
Reálné použití

```
#include "du1simd.hpp"
#include <xmmintrin.h>    // de-facto standard, obsahuje SSE3 instrukce
                          // __m128 odpovídá double[2]/float[4]
simd_vector< float, __m128> my_vector( 1000000);

namespace simd_op {
    __m128 add( __m128 a, __m128 b)
    {
        return _mm_add_ps( a, b);    // instrukce ADDPS
    }
    float sum( __m128 a)
    {
        float x;
        __m128 b = _mm_hadd_ps( a, a);    // HADDPS
        __m128 c = _mm_hadd_ps( b, b);    // HADDPS
        _mm_store_ss( & x, c);            // MOVSS
        return x;
    }
    /*...*/
};
```

Iterátory a okraje intervalů

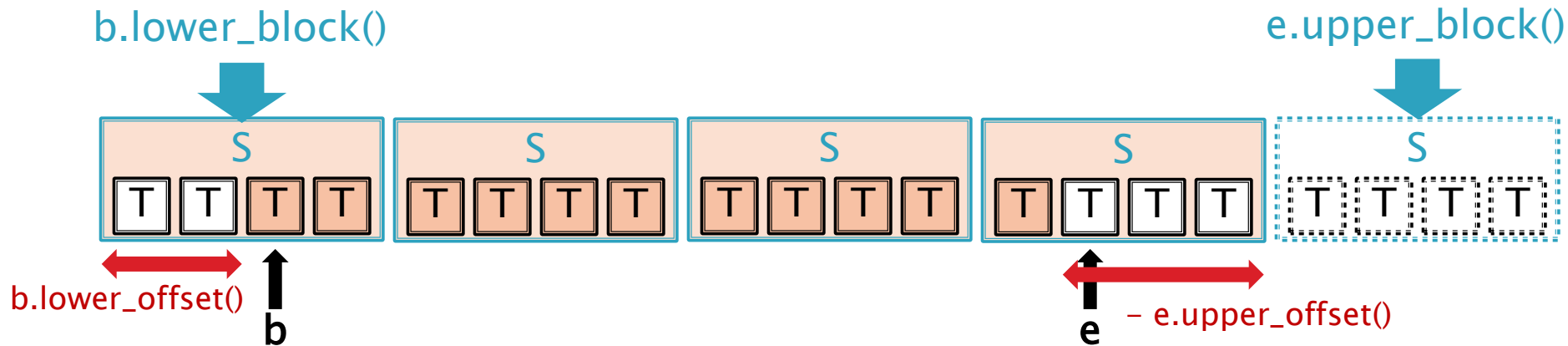
- ▶ uživatel může chtít procházet část kontejneru
 - hranice interval nemusí být dělitelné K!
 - při použití SIMD operací musí hranice intervalu řešit speciálně
 - simd iterátory musí zpřístupnit i několik sousedních prvků
 - ty do intervalu nepatří
 - odmaskování za-hraničních prvků vyřeší uživatel vlastní SIMD operací



Příklad použití – sečtení prvků

```
T sum( simd_vector<T,S>::iterator b, simd_vector<T,S>::iterator e)
{
    assert( e - b >= K + 1);           // kratke intervaly nutno resit specialne
    simd_vector<T,S>::simd_iterator bb = b.lower_block();
    simd_vector<T,S>::simd_iterator ee = e.upper_block() - 1;

    S acc = simd_op::mask_lower( *bb, b.lower_offset());
    for ( ++bb; bb != ee; ++bb) {
        acc = simd_op::add( acc, *bb);
    }
    acc = simd_op::add( acc, simd_op::mask_upper( *bb, e.upper_offset()));
    return simd_op::sum( acc);
}
```



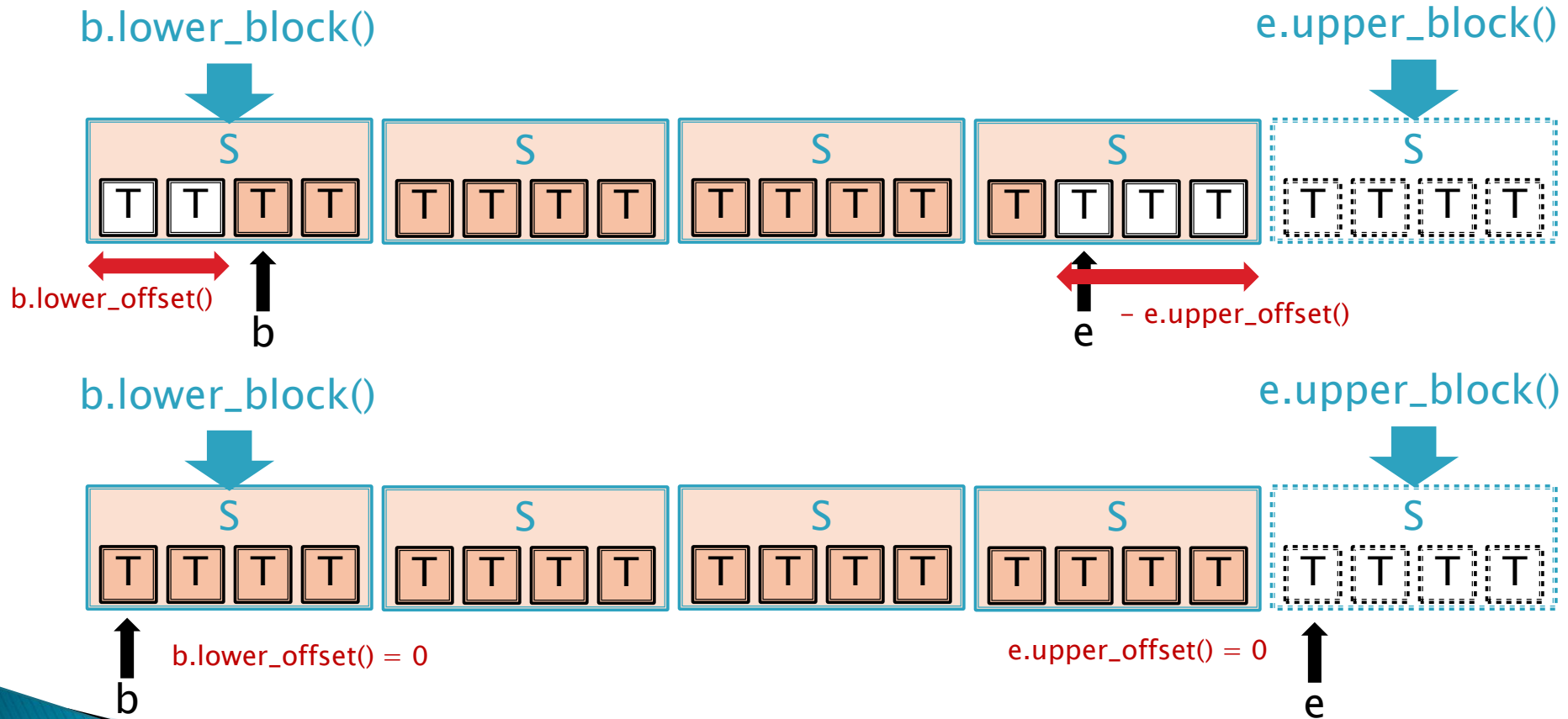
Invarianty a okrajové případy

$0 \leq b.lower_offset() < K$

$-K < b.upper_offset() \leq 0$

$\&*b == (T^*)\&*b.lower_block() + b.lower_offset()$

$\&*e == (T^*)\&*e.upper_block() + e.upper_offset()$



Upřesnění zadání

- ▶ `simd_vector< T, S>::iterator`
 - ▶ splňuje podmínky `random-iterator` category
 - ▶ `value_type = T`
 - ▶ metody `lower_block`, `upper_block` vracejí `simd_iterator`
 - ▶ metody `lower_offset`, `upper_offset` vracejí posunutí vůči `simd_iterator`
- ▶ `simd_vector< T, S>::simd_iterator`
 - ▶ splňuje podmínky `random-iterator` category
 - ▶ `value_type = S`
- ▶ `random-iterator` category (viz standard C++, `<iterator>`)
 - ▶ iterátor je ukazatel nebo třída obsahující typy
 - ▶ `typedef random_iterator_tag iterator_category;`
 - ▶ `difference_type`, `value_type`, `pointer`, `reference`
 - ▶ iterátor lze vytvořit bez inicializace (konstruktor bez parametrů)
 - ▶ kopie: `copy-konstruktor`, `operator=`
 - ▶ jsou definovány tyto operátory (*a, b iterátory, n typu difference_type*)
 - ▶ `*a`, `a[n]`
 - ▶ `a == b`, `a != b`, `a < b`, `a > b`, `a <= b`, `a >= b`
 - ▶ `a + n`, `n + a`, `a - n`, `a - b`
 - ▶ `++a`, `a++`, `--a`, `a--`, `a += n`, `a -= n`

Inicializace kontejneru

- ▶ velikost kontejneru (počet prvků typu T) se určuje parametrem konstruktoru
 - počet prvků kontejneru nemusí být dělitelný K
 - kontejner nemá metody pro zvětšování a zmenšování
 - po vzniku není obsah prvků kontejneru inicializován
- ▶ kontejner podporuje move-constructor a move-assignment
 - nepodporuje copy metody
- ▶ všechny přístupy na prvky typu S musejí být zarovnané!
 - adresa dělitelná sizeof(S)
 - `new S[N]` nezaručuje potřebné zarovnání!
 - použijte `std::align` (C++11)
 - *gcc: `_aligned_malloc` (#ifdef)*
- ▶ metody `begin()` a `end()` vracejí iterator
 - `size() = end() - begin() =` počet prvků typu T (parametr konstruktoru)

Kritéria hodnocení

- ▶ včasnost
 - za nedodržení termínu body prudce dolů
- ▶ přeložitelnost
 - přeložitelné bez chyb a (pokud možno) warningů
 - kompatibilní s vzorem použití (rozhraní je pevné)
- ▶ úplnost
 - všechny potřebné deklarace, metody a operátory
- ▶ stabilita
 - vyzkoušejte různé velikosti a meze
- ▶ kultura kódu
 - pravidla, moudra, dobré zvyky, udržitelnost, estetika, čitelnost
- ▶ odevzdávací formality a konvence
 - názvy a struktura souborů, komentáře

Pokyny pro odevzdání

- ▶ termín: středa 12.3. 10:00
- ▶ zadání a pomocné soubory:
 - <http://www.ksi.mff.cuni.cz/lectures/NPRG051/html/nprg051.html>
- ▶ v du1.zip najdete 3 soubory
 - du1test.cpp
 - kód používající vaše řešení – neměňte – **zachovat rozhraní!**
 - *(velmi doporučeno!)* můžete si přidat vlastní testy
 - du1simd.hpp, du1simd.cpp
 - zde **doplňte** vaše řešení a odevzdejte
- ▶ soubory nepřejmenovávejte
 - na začátek každého souboru vložte komentář typu

```
// DU1simd.hpp
// Karel Vomacka NPRG051 2013/2014
```
- ▶ vaše řešení vložte do Grupíčku – *neposílejte emailem!*
 - správné soubory do správných sloupečků!
 - pouze du1simd.hpp, du1simd.cpp

DÚ 2

Konstatní databáze

DÚ2 – Motivace ☹️

```
enum class barva { bila, modra, zelena };
```

```
std::string to_string(barva b)
{
    switch (b) {
        case barva::bila:
            return "bila";
        case barva::modra:
            return "modra";
        case barva::zelena:
            return "zelena";
        default:
            throw std::invalid_argument("b");
    }
}
```

```
barva from_string(const std::string &str)
{
    if (str == "bila") {
        return barva::bila;
    } else if (str == "modra") {
        return barva::modra;
    } else if (str == "zelena") {
        return barva::zelena;
    } else {
        throw std::invalid_argument("str");
    }
}
```

DÚ2 – Motivace 😊

```
struct barevna_db {  
    ...  
    typedef std::tuple<barva, std::string> value_type;  
    ...  
    insert(barva::bila, "bila");  
    insert(barva::modra, "modra");  
    insert(barva::zelena, "zelena");  
    ...  
};  
  
std::string to_string(barva b)  
{  
    return std::get<1>(find<barevna_db, 0>(b));  
}  
  
barva from_string(const std::string &str)  
{  
    return std::get<0>(find<barevna_db, 1>(str));  
}
```

DÚ2 – Zadání

```
struct db {  
    ...  
    typedef std::tuple<...> value_type;  
    ...  
};
```

```
template <class db, int idx>  
const typename db::value_type &find(  
    const typename std::tuple_element<idx, typename db::value_type>::type &key  
);
```

- ▶ třída db reprezentuje konstatní databázi, tj. databázi, kterou je možné pouze vytvořit a následně pouze číst
- ▶ funkce find slouží k vyhledávání v této databázi podle zadané hodnoty (key) v zadaném sloupci (idx)
 - pokud si databázi představíme jako posloupnost n-tic, pak funkce find vrátí referenci na n-tici, která má v idx-té složce hodnotu key
 - pokud taková n-tice neexistuje nebo jich je více, bude vyhozená výjimka

DÚ2 – Upřesnění zadání

- ▶ obsah třídy/struktury reprezentující databázi může být libovolný
- ▶ rozhraní funkce `find` je pevné
- ▶ primárním cílem je, aby funkce `find` opravdu fungovala
 - ▶ je možné odevzdat řešení, které místo konstantní reference na výslednou n -tici vrací tuto n -tici hodnotou
- ▶ sekundárním cílem je, aby funkce `find` fungovala rychle a úsporně
 - ▶ např. inicializovat/vytvářet databázi líně, vyhledávat n -tice pomocí efektivních datových struktur (např. `std::map`), šetřit paměť, ...

DÚ2 – Několik hintů I

- ▶ Hint 1a:

```
template <size_t>
struct B {
    int n_;
};
```

```
struct A : B<0>, B<1> {
    A()
    {
        // n_ = 1; - ambiguous access of 'n_'
        B<0>::n_ = 1; // OK
        B<1>::n_ = 2; // OK
    }
};
```

- ▶ Hint 1b:

- ▶ viz slajdy k přednášce

DÚ2 – Několik hintů II

► Hint 2a:

```
template <typename T>
struct A {
    static int n_;
};
template <typename T> int A<T>::n_;

int main()
{
    A<barevna_db>::n_ = 1;
    A<cernobila_db>::n_ = 2;
    std::cout << A<barevna_db>::n_ << A<cernobila_db>::n_ << std::endl;
    return 0;
}
```

DÚ2 – Několik hintů III

► Hint 2b:

```
template <typename T>
struct A {
    static int f() {
        static int n = 0;
        return ++n;
    }
};

int main()
{
    std::cout << A<barevna_db>::f() << A<cernobila_db>::f() << std::endl;
    return 0;
}
```

Kritéria hodnocení

- ▶ včasnost
 - za nedodržení termínu body prudce dolů
- ▶ přeložitelnost
 - přeložitelné bez chyb a (pokud možno) warningů
- ▶ jednoduchost
 - definice databáze (tj. obsah třídy/struktury db) musí být jednoduchá, intuitivní a bez zbytečné „vaty“
- ▶ kvalita návrhu
- ▶ stabilita, kultura kódu
 - pravidla, moudra, dobré zvyky, udržitelnost, estetika, čitelnost
- ▶ odevzdávací formality a konvence
 - názvy a struktura souborů, komentáře

Pokyny pro odevzdání

- ▶ termín: středa 9.4. 10:00
- ▶ na začátek každého souboru vložte komentář typu

```
// DU2cdb.hpp
// Karel Vomacka NPRG051 2013/2014
```
- ▶ vaše řešení vložte do Grupíčku – *neposílejte emailem!*
 - správné soubory do správných sloupečků!
 - du2cdb.hpp
 - implementace funkce find
 - du1cdb.cpp
 - příklad použití, tj. alespoň jedna definice konstantí databáze + několik ukázek úspěšných/neúspěšných vyhledávání v ní