

```
template <
    int D, typename S1, typename S2, typename B2,
    typename E1, typename E2, typename T1, typename T2
>
struct for_id_impl<D, S1, S2, E1, B2, E1, E2, T1, T2> {
```

POKROČILÉ VYUŽITÍ C++ ŠABLON

DANIEL LANGR

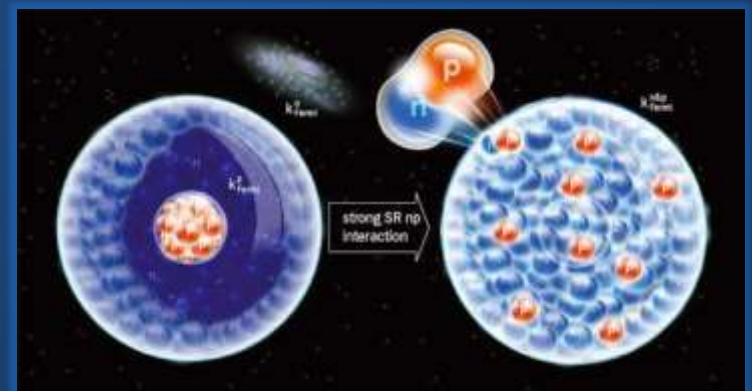
ČVUT FIT / VZLÚ

daniel.langr@fit.cvut.cz

```
template <typename T>
static void execute(T& f, int id1, int id2 = 0) {
    if (pos<S2, typename deref<B2>::type>::value == id2)
        executor<D, T1, typename deref<B2>::type>::execute(f);
    // f.template operator()<T1, typename deref<B2>::type>();
```

Modelování interakcí nukleonů v atomových jádrech

- Výpočetně náročné problémy \Rightarrow využití **superpočítačů** (masivně paralelní systémy)
- Nejvýkonnější dnešní superpočítače \Rightarrow **lehká jádra (těžší jádra s nízkou věrohodností výsledků)**
- Těžší jádra:
 - výkonnější superpočítače
 - **chytřejší přístup**



Chytřejší přístup k problému

- Jerry P. Draayer (LSU), 70. léta 20. století
 - využití **SU(3) symetrií**
- Výrazně složitější matematický model \Rightarrow **složitější algoritmizace / implementace:**
 - pokročilé datové struktury (asociativní pole, hash tabulky, LRU cache, vyhledávací stromy, ...)
- Realizace: Tomáš Dytrych (FJFI, LSU), 2007+, **C++**



Příklad: proton-neutronová interakce

C.1 P-N Interaction

$$\langle r i_p i_n \omega' S' \alpha'_p \alpha'_n \rho' \kappa' L' || H || r j_p j_n \omega S \alpha_p \alpha_n \rho \kappa L \rangle =$$

$$\Pi_{J L' S'} \times \Pi_{S_p S_n} \times \sum_{\substack{n_p n_n \omega_p^p \omega_n^p \\ n_p n_n \omega_p^n \omega_n^n \\ \rho_0 \omega_0 S_0 \kappa_0}} c^{L-1} \times \Pi_{S_0} \times \begin{Bmatrix} L & S & J \\ S_0 & S_0 & 0 \\ L' & S' & J \end{Bmatrix} \times \begin{Bmatrix} S_p & S_n & S \\ S_0^p & S_0^n & S_0 \\ S_p' & S_n' & S' \end{Bmatrix} \times \sum_{\beta} \langle \omega \kappa L; \omega_0 \kappa_0 L_0 || \omega' \kappa' L' \rangle_{\beta} \times$$

$$\sum_{\rho_p \rho_n} \begin{Bmatrix} \omega_p & \omega_p^p & \omega_p^n & \rho_p \\ \omega_n & \omega_n^p & \omega_n^n & \rho_n \\ \omega & \omega_0 & \omega' & \rho \\ \rho & \rho_0 & \rho' & \end{Bmatrix} \times \langle \alpha'_p \omega'_p S'_p || \{ a_{n_p}^\dagger \times \tilde{a}_{n_p} \}_{S_0^p}^{\omega_p^p} || \alpha_p \omega_p S_p \rangle_{\rho_p} \times \langle \alpha'_n \omega'_n S'_n || \{ a_{n_n}^\dagger \times \tilde{a}_{n_n} \}_{S_0^n}^{\omega_n^n} || \alpha_n \omega_n S_n \rangle_{\rho_n}$$

1. rovnice
2. algoritmus řešení (pseudokód)

Main Algorithm

2.1 Begin—Function main()

```

1: H ← 0
2: for all unique (i_p, i_n) in pbasisI ordered by pos do
3:   Irow ← pbasisI[i_p, i_n].pos
4:   S_p' ← pconfsl[i_p, S_p]
5:   ω_p' ← pconfsl[i_p, ω_p]
6:   α_p,mas ← pconfsl[i_p, α_p,mas]
7:   S_n' ← nconfsl[i_n, S_n]
8:   ω_n' ← nconfsl[i_n, ω_n]
9:   α_n,mas ← nconfsl[i_n, α_n,mas]
10: for all unique (j_p, j_n) in pbasisJ ordered by pos do
11:   Jcol ← pbasisJ[j_p, j_n].pos
12:   S_p ← pconfsl[j_p, S_p]
13:   ω_p ← pconfsl[j_p, ω_p]
14:   α_p,mas ← pconfsl[j_p, α_p,mas]
15:   S_n ← nconfsl[j_n, S_n]
16:   ω_n ← nconfsl[j_n, ω_n]
17:   α_n,mas ← nconfsl[j_n, α_n,mas]
18:   resPN ← resolvePN(...)
19:   if i_n = j_n then calculate resPP
20:   if i_p = j_p then calculate resNN
21:   for all unique (ω', S') in pbasisI[i_p, i_n] do
22:     ρ_mas ← (ω' ⊗ ω_n = ω')
23:     for all unique (ω, S) in pbasisJ[j_p, j_n] do
24:       ρ_mas ← (ω ⊗ ω_p = ω)
25:       contribPN(...)
26:       if i_n = j_n then contribPP(...)
27:       if i_p = j_p then contribNN(...)
28:       Jcol ← Jcol + [pbasisJ[j_p, j_n][ω, S]]
29:   Irow ← Irow + [pbasisI[i_p, i_n][ω', S']]

```

2.4 Function calculateAPN(...)

```

1: APN ← 0
2: for ρ_0 in 0, ..., (ρ_0,mas - 1) do
3:   for β in 0, ..., (β_mas - 1) do
4:     for ρ_p in 0, ..., (ρ_p,mas - 1) do
5:       for ρ_n in 0, ..., (ρ_n,mas - 1) do
6:         APN[ρ_0, β] ← APN[ρ_0, β] +
7:           resPN[ω_0^p, ω_0^n, ω_0][S_0^p, S_0^n, S_0].c[||||]_p[α'_p, α_p, ρ_p]
8:           { ω_p, ω_p^p, ω_p^n } [ρ_p, ρ_0, ρ', β, ρ_p, ρ_n] ×
9:           resPN[ω_0^p, ω_0^n, ω_0][S_0^p, S_0^n, S_0].c[||||]_n[α'_n, α_n, ρ_n]

```

2.5 Function contribAPN(...)

```

1: for all (κ', L') in pbasisI[i_p, i_n][ω', S'] do
2:   for all (κ, L) in pbasisJ[j_p, j_n][ω, S] do
3:     B ← 0
4:     for κ_0 in 0, ..., (κ_0,mas - 1) do
5:       for ρ_0 in 0, ..., (ρ_0,mas - 1) do
6:         D ← 0
7:         for β in 0, ..., (β_mas - 1) do
8:           D ← D + (ω ⊗ L; ω_0 ρ_0 L_0 || ω' κ' L')_β × APN[ρ_0, β]
9:           B ← B + D × resPN[ω_0^p, ω_0^n, ω_0][S_0^p, S_0^n, S_0].c[||||]_n[α'_n, α_n, ρ_n]
10:        H[Irow, Jcol] ← H[Irow, Jcol] + B × Π_{J L' S'} × Π_{J L S} = { L S J } { L' S' J }
11:        Jcol ← Jcol + 1
12:        Irow ← Irow + 1

```

2.2 Function resolvePN(...)

```

1: resPN.clear()
2: for all i_m in pmas[i_p, j_p] do
3:   k[||||]_p ← pmas[i_p, j_p][i_m]
4:   ω_0^p ← adta[i_m, ω_0^p]
5:   S_0^p ← adta[i_m, S_0^p]
6:   for all i_n in nmas[i_n, j_n] do
7:     k[||||]_n ← nmas[i_n, j_n][i_m]
8:     ω_0^n ← adta[i_m, ω_0^n]
9:     S_0^n ← adta[i_m, S_0^n]
10:    for all (ω_p, S_p) in print[i_p, i_m] do
11:      k.c ← pconfsl[i_p, i_m][ω_p, S_p]
12:      resPN.insert[ω_0^p, ω_0^n, ω_0, S_0^p, S_0^n, S_0, k[||||]_p, k[||||]_n, k.c]

```

2.3 Function contribPN(...)

```

1: for all unique (ω_0^p, ω_0^n, ω_0) in resPN do
2:   β_mas ← (ω_0 ⊗ ω_0 = ω')
3:   if 0 = β_mas then continue
4:   calculate { ω_p, ω_p^p, ω_p^n } for all [ρ_p, ρ_0, ρ', β, ρ_p, ρ_n]
5:   for all unique (S_0^p, S_0^n, S_0) in resPN[ω_0^p, ω_0^n, ω_0] do
6:     if 0 = (S_0 ⊗ S_0 = S') then continue
7:     L_0 ← S_0
8:     κ_0,mas ← SUB::max[ω_0, L_0]
9:     calculate { S_p, S_n, S } { S_0^p, S_0^n, S_0 }
10:    Irow ← Irow
11:    for α'_p in 0, ..., (α'_p,mas - 1) do
12:      for α'_n in 0, ..., (α'_n,mas - 1) do
13:        for ρ' in 0, ..., (ρ'_mas - 1) do
14:          Jcol ← Jcol
15:          for α_p in 0, ..., (α_p,mas - 1) × 0 do
16:            for α_n in 0, ..., (α_n,mas - 1) do
17:              for ρ in 0, ..., (ρ_mas - 1) do
18:                APN ← calculateAPN(...)
19:                contribAPN(...)

```

Programování pro superpočítače

- ~~PHP, Python, C#, Perl, Ruby, ...~~

- Java ???

- Fortran, C, C++

- vysoce optimalizované překladače

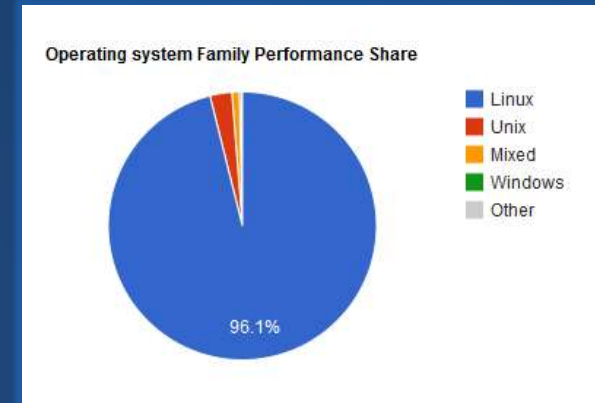
- Fortran/C: jazyky „nižší úrovně“

- Fortran oblíbený mezi matematiky a fyziky

- C++

- OOP, statický/dynamický polymorfismus, generické programování, metaprogramování, ...

- knihovny STL, Boost (algoritmy, datové struktury)



Blue Waters/ Hopper / Notebook

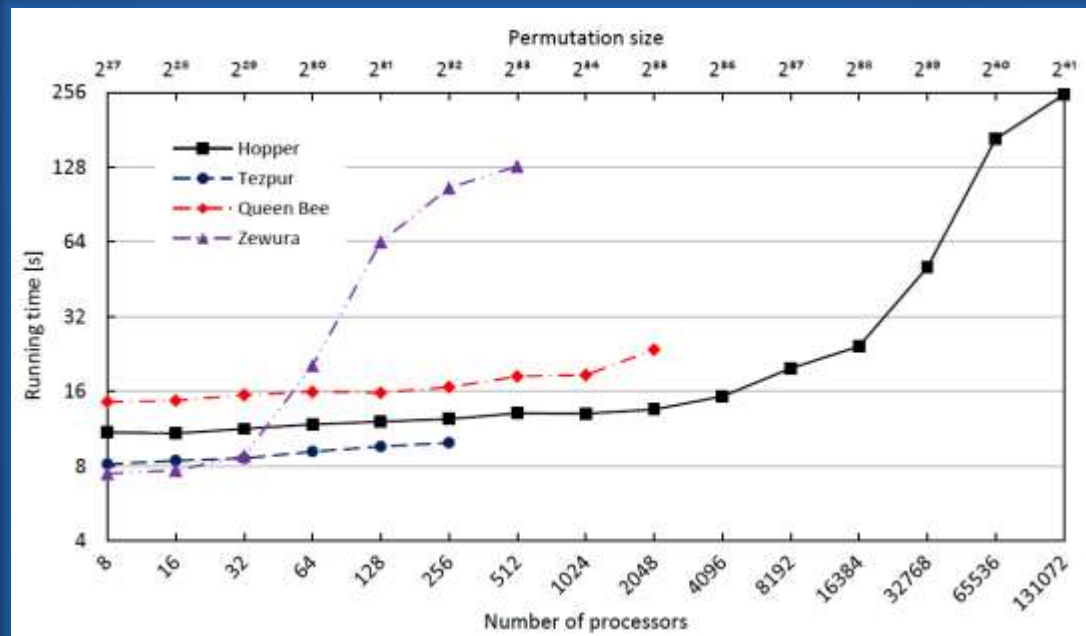
	Blue Waters	Hopper	Notebook
Počet CPU jader	386 816	153 216	2
Operační paměť [GB]	1 382 096	217 000	8
Počet GPU (CUDA) jader	8 847 360	0	8
Diskový prostor [TB]	26 400	2 000	0,256
Propustnost I/O [GB/s]	> 1 000	35	0,160
Výkon [GFLOPS]	≈ 11 500 000	≈ 1 000 000	≈ 2
Fond (core hours/rok)	1 000 000	29 000	neomezený



Paraperm: paralelní generování náhodných permutací

$$\pi : \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}, \quad (0, \dots, n-1) \rightarrow (\pi(0), \dots, \pi(n-1))$$

- Promíchání čísel 0, ..., n-1
- Velké množství čísel \Rightarrow výsledná permutace je distribuována mezi lokální paměti jednotlivých procesorů
- Při generování dochází ke komunikaci mezi procesory (převážná většina času běhu algoritmu)



Paraperm: datový typ pro permutovaná čísla?

- Univerzální použití 64-bitového datového typu
 - pro $n \leq 2^{32} \Rightarrow 0x00000000\bullet\bullet\bullet\bullet\bullet\bullet$
 - plýtvání pamětí a (drahým/omezeným) časem (komunikace)
- `uint64_t` pokud $n > 2^{32}$
- jinak `uint32_t` (~~`uint16_t`, `uint8_t`~~)
- **Implementace**: C++, generátor = šablona, datový typ její parametr

Paraperm: API / příklad použití

- Implementace formou knihovny
- Paralelní programovací model MPI
 - komunikace mezi procesory zasíláním zpráv

```
namespace paraperm
{
    template <typename T = uintmax_t>
    class Paraperm : boost::noncopyable
    {
    public:
        typedef T value_type;
        typedef std::vector<T> vector_type;

        Paraperm();
        ~Paraperm();

        void generate(MPI_Comm comm, T n);

        const vector_type& perm() const;
        T pos() const;
        T count() const;

    private:
        struct Impl;
        Impl* pimpl_;
    };
}
```

```
#include <mpi.h>
#include <cstdint>
#include <paraperm/Paraperm.h>

typedef paraperm::Paraperm<uint64_t> Paraperm;

int main(int argc, char* argv[])
{
    MPI_Init(&argc, &argv);
    int N;
    MPI_Comm_size(MPI_COMM_WORLD, &N);

    Paraperm paraperm;

    const Paraperm::value_type n = (1UL << 24) * N;
    paraperm.generate(MPI_COMM_WORLD, n);

    const Paraperm::vector_type& perm = paraperm.perm();
    const Paraperm::value_type pos = paraperm.pos();
    const Paraperm::value_type count = paraperm.count();

    // do whatever with the generated permutation

    MPI_Finalize();
    return 0;
}
```

Komunikační rutiny MPI

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, ...)
```

- **MPI_Datatype:**
 - výčtová konstanta určující typ elementů v poli buf
 - hodnoty MPI_INT, MPI_UNSIGNED, MPI_FLOAT, ...

```
std::vector<int> buf;
...
MPI_Send(&(buf[0]), buf.size(), MPI_INT, ...);

// Paraperm implementation
template <typename T>
class Paraperm_Impl {
public:
    void generate(...) {
        std::vector<T> buf;
        ...
        MPI_Send(&(buf[0]), buf.size(), MPI_???, ...);
        ...
    }
};
```

Řešení = šablony / specializace

```
// primární šablona (pouze deklarace)
template <typename T>
struct GetMpiDataType;

// specializace pro konkrétní typy
template <>
struct GetMpiDataType<unsigned char> {
    enum { value = MPI_UNSIGNED_CHAR };
};

template <>
struct GetMpiDataType<unsigned short> {
    enum { value = MPI_UNSIGNED_SHORT };
};

... // podobně pro ostatní typy

// obálky (wrappers)
template <typename T>
constexpr MPI_Datatype get_mpi_datatype(
    const T& arg) {
    return GetMpiDataType<T>::value;
}

template <typename T>
constexpr MPI_Datatype get_mpi_datatype() {
    return get_mpi_datatype(T());
}
```

```
template <typename T>
class Paraperm_Impl {
public:
    Paraperm_Impl() :
        // mdt_(GetMpiDataType<T>::value) { }
        mdt_(get_mpi_datatype<T>()) { }

    void generate(...) {
        std::vector<T> buf;

        // MPI_Datatype mdt = get_mpi_datatype(buf[0]);
        ...
        MPI_Send(&(buf[0]), buf.size(), mdt_, ...);
        ...
    }

private:
    MPI_Datatype mtd_;
};
```

Výčtový typ vs statické konstanty

- Statická konstanta je **l-hodnota (l-value)** \Rightarrow v případě předávání parametru odkazem musí překladač alokovat pro konstantu paměť a předat její adresu

```
...  
  
// specializace pro konkrétní typy  
template <>  
struct GetMpiDataType<unsigned char> {  
    static const int value = MPI_UNSIGNED_CHAR;  
    enum { value = MPI_UNSIGNED_CHAR };  
};  
  
... // podobně pro ostatní typy  
  
void f(int const&);  
...  
f(GetMpiDataType<T>::value);
```

Omezení možných typů pro permutovaná čísla

```
// get_mpi_datatype.h

#ifndef GET_MPI_DATATYPE_H
#define GET_MPI_DATATYPE_H

/* MPI_CHAR:          signed char
 * MPI_SHORT:         signed short int
 * MPI_INT:           signed int
 * MPI_LONG:          signed long
 * MPI_LONG_LONG:     signed long long
 * MPI_UNSIGNED_CHAR: unsigned char
 * MPI_UNSIGNED_SHORT: unsigned short int
 * MPI_UNSIGNED:       unsigned int
 * MPI_UNSIGNED_LONG:  unsigned long
 * MPI_UNSIGNED_LONG_LONG: unsigned long long
 * MPI_FLOAT:          float
 * MPI_DOUBLE:         double
 * MPI_LONG_DOUBLE:    long double */

namespace mpi
{
    template <typename T>
    struct GetMpiDataType;

    ...
}

#endif
```

```
#include <cstdint>
#include <get_mpi_datatype.h>

template <typename T>
struct GMDT_Restricted;

// specializace pouze pro požadované typy
template <> struct GMDT_Restricted<uint32_t> {
    enum {
        value = mpi::get_mpi_datatype<uint32_t>();
    };
};

template <> struct GMDT_Restricted<uint64_t> {
    enum {
        value = mpi::get_mpi_datatype<uint64_t>();
    };
};

...

template <typename T>
class Paraperm_Impl {
public:
    Paraperm_Impl() :
        mdt_(GMDT_Restricted<T>::value) { }

    ...
};
```

Omezení možných typů pro permutovaná čísla

```
template <typename T>
struct GMDT_Restricted;

template <> struct GMDT_Restricted<uint32_t> {
    enum {
        value = mpi::get_mpi_datatype<uint32_t>() };
};

template <> struct GMDT_Restricted<uint64_t> {
    enum {
        value = mpi::get_mpi_datatype<uint64_t>() };
};

...

template <typename T>
class Paraperm_Impl {
public:
    Paraperm_Impl() :
        mdt_(GMDT_Restricted<T>::value) { }
    ...
};
```

```
int main()
{
    Paraperm_Impl<float> paraperm;

    return 0;
}
```

```
get_mpi_datatype.cpp: In instantiation of 'Paraperm_Impl<T>::Paraperm_Impl() [with T = float]':
get_mpi_datatype.cpp:49:26:   required from here
get_mpi_datatype.cpp:41:57: error: incomplete type 'GMDT_Restricted<float>' used in nested name specifier
```


typedef \approx ~~nový typ~~ alias typu

```
// get_mpi_datatype.h
```

```
/* MPI_CHAR:          signed char
 * MPI_SHORT:         signed short int
 * MPI_INT:           signed int
 * MPI_LONG:          signed long
 * MPI_LONG_LONG:     signed long long
 * MPI_UNSIGNED_CHAR: unsigned char
 * MPI_UNSIGNED_SHORT: unsigned short int
 * MPI_UNSIGNED:       unsigned int
 * MPI_UNSIGNED_LONG:  unsigned long
 * MPI_UNSIGNED_LONG_LONG: unsigned long long
 * MPI_FLOAT:          float
 * MPI_DOUBLE:         double
 * MPI_LONG_DOUBLE:    long double */
```

```
...
```

```
// stdint.h
```

```
// příklad pro konkrétní architekturu:
```

```
...
```

```
typedef unsigned int  uint32_t;
```

```
typedef unsigned long uint64_t;
```

```
...
```

```
#include <stdint>
```

```
#include <get_mpi_datatype.h>
```

```
template <typename T>
```

```
struct GMDT_Restricted;
```

```
// specializace pouze pro požadované typy
```

```
template <> struct GMDT_Restricted<uint32_t> {
```

```
    enum {
```

```
        value = mpi::get_mpi_datatype<uint32_t>()
```

```
        // mpi::GetMpiDataType<uint32_t>::value
```

```
    };
```

```
};
```

```
template <> struct GMDT_Restricted<uint64_t> {
```

```
    enum {
```

```
        value = mpi::get_mpi_datatype<uint64_t>()
```

```
        // mpi::GetMpiDataType<uint32_t>::value
```

```
    };
```

```
};
```

Knihovna Boost

<http://www.boost.org>



- „...one of the most highly regarded and expertly designed C++ library projects in the world.“
— Herb Sutter and Andrei Alexandrescu, *C++ Coding Standards*
- „co chybí v STL“
- desítky (pod)knihoven
— většina z nich jsou typu „header-only“
- mnoho zakladatelů je členy výboru pro C++ standard
- některé knihovny byly převzaty do TR1 a následně do standardu C++11

Boost / MPI

```
#include <boost/mpi/datatype.hpp>

template <typename T>
class Paraperm_Impl {
public:
    Paraperm_Impl() : mdt_(boost::mpi::get_mpi_datatype<T>()) { }

    ...

private:
    MPI_Datatype mtd_;
};
```

- Výhoda: úspora práce, prevence chyb
- Nevýhoda: závislost Parapermu na knihovně Boost
- Omezení možných typů?
 - porovnání typů
 - Boost static assert

Porovnání typů / Boost static assert

```
#include <boost/static_assert.hpp>
```

```
int main()
{
    BOOST_STATIC_ASSERT(false);

    return 0;
}
```

/ výstup překladač:*

*In function 'int main()':
error: static assertion failed: false*

**/*

```
#include <boost/static_assert.hpp>
```

```
template <typename T>
class Paraperm_Impl {
    BOOST_STATIC_ASSERT((is_same<T, uint32_t>::value ||
                        is_same<T, uint64_t>::value));

public:
    Paraperm_Impl() : mdt_(boost::mpi::get_mpi_datatype<T>())
    { }
    ...
};
```

```
int main() {
    Paraperm_Impl<uint32_t> paraperm1;
    std::cout << std::endl;
    Paraperm_Impl<float> paraperm2;

    return 0;
}
```

/ výstup překladač:*

*In instantiation of 'class Paraperm_Impl<float>':
error: static assertion failed: (is_same<T, uint32_t>::value ||
is_same<T, uint64_t>::value)*

**/*

Boost type_traits

- získání informací o typech, manipulace s typy
 - is_array, is_class, is_base_of, is_floating_point, is_pointer, is_same, is_unsigned, ...
 - has_plus, has_copy_constructor, has_new_operator, ...
 - add_const, add_pointer, ...

```
#include <boost/mpi/datatype.hpp>
#include <boost/type_traits/is_same.hpp>
#include <boost/static_assert.hpp>

template <typename T>
class Paraperm_Impl {
    BOOST_STATIC_ASSERT((boost::is_same<T, uint32_t>::value || boost::is_same<T, uint64_t>::value));

public:
    Paraperm_Impl() : mdt_(boost::mpi::get_mpi_datatype<T>())
    { }
    ...
};
```

Paramerm: generátor náhodných čísel

- C++11 (Boost): generátor + distribuce
- Generátor: Mersenne-Twister
 - 32-bitová čísla: `mt19937`
 - 64-bitová čísla: `mt19937_64`
- Distribuce:
 - `uniform_int_distribution`, `uniform_real`, `normal_distribution`, ...
- Boost `lexical_cast`: převod hodnot různých typů na text a naopak

```
#include <random>

template <typename T>
struct rng_traits;

template <>
struct rng_traits<uint32_t> {
    typedef std::mt19937 type;
};

template <>
struct rng_traits<uint64_t> {
    typedef std::mt19937_64 type;
};
```

```
#include <random>

template <typename T> // T je uint32_t nebo uint64_t
class Paraperm_Impl {
public:
    void generate(T n) {
        typename rng_traits<T>::type rng;
        std::uniform_int_distribution<T> dist(0, n-1);

        for (...) { T random_number = dist(rng); ... }
        ...
    }
};
```


Paraperm: datový typ?

```
#include <paraperm/paraperm.h>

int main(int argc, char* argv[]) {
    uint64_t n = boost::lexical_cast<uint64_t>(argv[1]);
    ...
    paraperm::Paraperm<??> permutation;
    permutation.generate(n);
    ...
}
```

- Výběr datového typu závisí na velikosti generované permutace:
 - `uint64_t` pokud $n > 2^{32}$, jinak `uint32_t`
- Velikost generované permutace je známa až za běhu programu (runtime)
- Argument šablony musí být znám při překladu (compile time)
- Obecně výběr typu za běhu programu:
 - dynamicky polymorfismus (dědičnost, virtuální funkce)
 - single dispatch, runtime overhead
- Jiné řešení: `template metaprogramming` (metaprogramování)

Metaprogramování

- Vyšší úroveň nad základním jazykem
- „Normální kód“ – výsledkem překladu je strojový kód spustitelný na procesoru
- „Metakód“ – výsledkem překladu je normální kód
- Metakód slouží ke generování „normálního“ kódu, který implementuje požadovanou funkcionalitu
- Překlad metakód – normální kód – strojový kód je pro uživatele většinou transparentní
- Metaprogram je vykonáván v čase překladu (compile time)
- Příklad: **preprocessor**

```
#define MAX(a, b) ((a) < (b) ? (b) : (a))

int main() {
    int i = 5, j = 10;
    int k = MAX(i, j); // int k = ((i) < (j) ? (j) : (i));

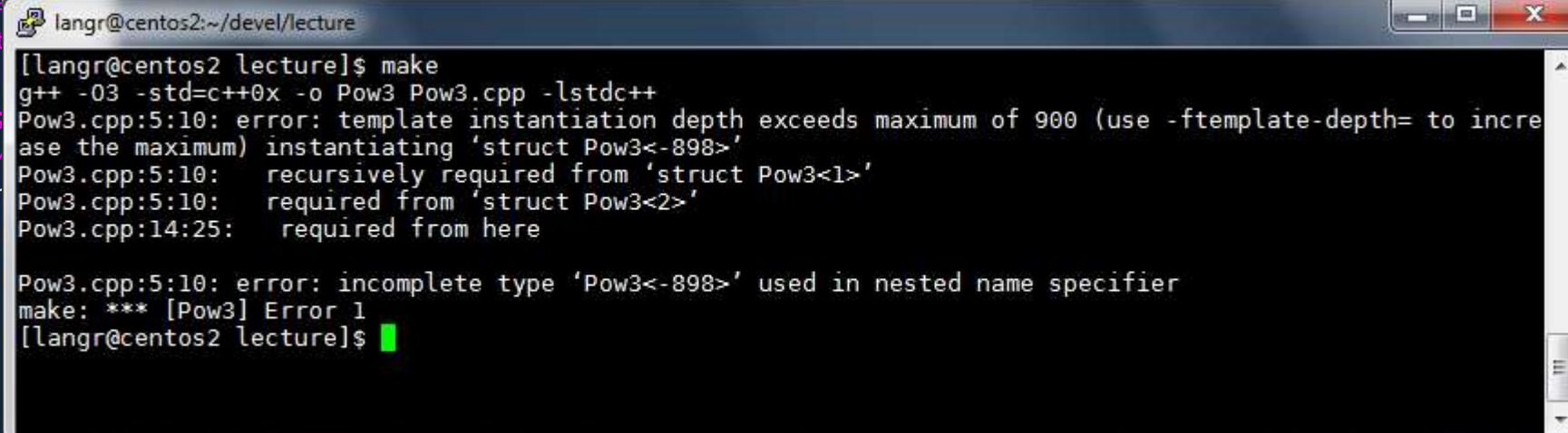
    return 0;
}
```

Template metaprogramming

- Metaprogramování pomocí šablon
- Využití rekurze a částečné / úplné specializace
- Příklad: **výpočet $3^N \approx \text{Pow3}<N>::\text{value}$**

```
// primární šablona
template <int N>
struct Pow3 {
    enum { value = 3 * Pow3<N-1>::value };
};
/*
// ukončení rekurze - úplná specializace
template <int N>
struct Pow3 {
    enum { value = 1 };
};
```

1. Překladač vytvoří instanci `Pow3<2>`
 - `Pow3<2>::value = 3 * Pow3<1>::value`
2. Překladač vytvoří instanci `Pow3<1>`
 - `Pow3<1>::value = 3 * Pow3<0>::value`
3. Překladač vytvoří instanci `Pow3<0>`
 - `Pow3<0>::value = 1`



```
langr@centos2:~/devel/lecture
[langr@centos2 lecture]$ make
g++ -O3 -std=c++0x -o Pow3 Pow3.cpp -lstdc++
Pow3.cpp:5:10: error: template instantiation depth exceeds maximum of 900 (use -ftemplate-depth= to increase the maximum) instantiating 'struct Pow3<-898>'
Pow3.cpp:5:10: recursively required from 'struct Pow3<1>'
Pow3.cpp:5:10: required from 'struct Pow3<2>'
Pow3.cpp:14:25: required from here
Pow3.cpp:5:10: error: incomplete type 'Pow3<-898>' used in nested name specifier
make: *** [Pow3] Error 1
[langr@centos2 lecture]$
```

Rozbalení smyčky (loop unrolling)

skalární součin vektorů

```
// primární šablona
template <int DIM, typename T>
struct DotProduct {
    // enum { value = ??? };
    static T value(const T* a, const T* b) {
        return (*a) * (*b)
            + DotProduct<DIM-1, T>::value(a+1, b+1);
    }
};
```

// částečná specializace - ukončení rekurze

```
template <typename T>
struct DotProduct<1, T> {
    static T value(const T* a, const T* b) {
        return (*a) * (*b);
    }
};
```

// obal (wrapper) - dedukce argumentů šablony

```
template <int DIM, typename T>
inline T dot_product(const T* a, const T* b) {
    return DotProduct<DIM, T>::value(a, b);
}
```

```
/*
template <typename T>
inline T dot_product(size_t dim, T* a, T* b) {
    T result = T();
    for (size_t i = 0; i < dim; ++i)
        result += a[i] * b[i];
    return result;
}
*/

int main() {
    int a[3] = {1, 2, 3};
    int b[3] = {4, 5, 6};
    //std::cout << dot_product(3, a, b) << std::endl;
    std::cout << dot_product<3>(a, b) << std::endl;
    return 0;
}

/*
dot_product<3>(a, b)
= DotProduct<3, int>::value(a, b)
= (*a)*(*b) + DotProduct<2, int>::value(a+1, b+1)
= (*a)*(*b) + (*(a+1))*(*(b+1))
    + DotProduct<1, int>::value(a+2, b+2)
= (*a)*(*b) + (*(a+1))*(*(b+1)) + (*(a+2))*(*(b+2))
= a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
*/
```

Metafunkce

- Vstupní parametry: parametry šablony (**typy, celá čísla**)
- Výstupní parametry: **typy, celá čísla**
- Vstupní i výstupní parametry známé v **čase překladu**
 - metafunkce je vyhodnocena v **čase překladu**
- Implementace: struct (class)
- Standardní syntaxe:

```
template <typename /* int, bool, ... */ par1, ..., typename /* int, bool, ... */ parN>
struct some_type_metafunction {
    typedef ... type;
};
```

```
template <typename /* int, bool, ... */ par1, ..., typename /* int, bool, ... */ parN>
struct some_numeric_metafunction {
    enum { value = ... };
};
```

```
typedef some_type_metafunction<arg1, ..., argN>::type result_type;
int result_value = some_numeric_metafunction<arg1, ..., argN>::value;
```

Příklad: metafunkce IfThenElse

výběr typu podle podmínky (známé v době překladu)

```
// primární šablona (pouze deklarace)
template <bool COND, typename T1, typename T2>
struct IfThenElse;
```

```
// částečná specializace pro COND == true
template <typename T1, typename T2>
struct IfThenElse<true, T1, T2> {
    typedef T1 type;
};
```

```
// částečná specializace pro COND == false
template <typename T1, typename T2>
struct IfThenElse<false, T1, T2> {
    typedef T2 type;
};
```

```
/*
template <typename T>
class Paraperm { ... };
...
Paraperm<float> generator; // chyba překladu
*/

// Paraperm: rozlišní uint32_t a uint64_t pomocí
// přepínače
template <bool NUM_64BIT>
class Paraperm {
    typedef typename IfThenElse<
        NUM_64BIT,
        uint64_t,
        uint32_t
    >::type num_type;

    ...
};

int main() {
    Paraperm<true> generator;
}
```


Boost MPL

Boost Metaprogramming Library

- „Ekvivalent“ STL na úrovni metaprogramování
 - sekvence: **vector**, list, deque, set, map, vector_c, ...
 - iterátory: **begin**, end, **next**, prior, **deref**, distance, ...
 - algoritmy: transform, **find**, replace, max_element, ...

```
#include <boost/mpl/begin_end.hpp>
#include <boost/mpl/find.hpp>
#include <boost/mpl/next_prior.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_same.hpp>

typedef boost::mpl::vector<char, short, int, long> types;
typedef boost::mpl::begin<types>::type iter1; // iterátor pro začátek sekvence types = iterátor pro char
typedef boost::mpl::next<iter1>::type iter2; // iterátor pro další typ po iter1 = iterátor pro short
typedef boost::mpl::deref<iter2>::type t2; // obsah (derefernce) iterátoru iter2 = typ short
typedef boost::mpl::find<types, short>::type iter2_; // alternativní iterátor pro typ short

BOOST_STATIC_ASSERT((boost::is_same<t2, short>::value)); // OK :)
BOOST_STATIC_ASSERT((boost::is_same<iter2_, iter2>::value)); // OK :)
```

Příklad: pozice typu v sekvenci

pozice = vzdálenost typu od začátku sekvence

```
#include <boost/mpl/begin_end.hpp>
#include <boost/mpl/distance.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/static_assert.hpp>
/*
template <typename S, typename T>
struct pos {
    enum {
        value = boost::mpl::distance< // distance: vzdálenost mezi dvěma iterátory
            typename boost::mpl::begin<S>::type, // iterátor pro první prvek sekvence
            typename boost::mpl::find<S, T>::type // iterátor pro hledaný typ
        >::value
    };
};
*/
template <typename S, typename T>
struct pos : public boost::mpl::distance<
    typename boost::mpl::begin<S>::type,
    typename boost::mpl::find<S, T>::type
>
{ };

typedef boost::mpl::vector<char, short, int, long> types;

BOOST_STATIC_ASSERT((pos<types, int>::value == 2));
```

Příklad: typově zobecněná funkce `max`

```
typedef boost::mpl::vector<
    char, short, int, long, float, double
> types;

/*
- porovnání pozic typů T a U v sekvenci priorit
- výsledkem metafunkce WiderType je typ, který má
  v sekvenci priorit vyšší pozici (prioritu)
*/
template <typename T, typename U>
struct WiderType {
    typedef typename IfThenElse<
        (pos<types, U>::value > pos<types, T>::value),
        U, T
    >::type type;
};
```

```
template <typename T, typename U>
typename WiderType<T, U>::type max(T a, U b) {
    if (b > a)
        return b;
    return a;
}

int main() {
    int i = 5;
    double d = 3.5;

    auto max1 = max(i, d);
    auto max2 = max(d, i);

    std::cout << typeid(max1).name() << std::endl;
    std::cout << typeid(max2).name() << std::endl;
    // typ výsledku v obou případech je double

    std::cout << max1 << std::endl;
    std::cout << max2 << std::endl;
    // výsledek v obou případech je 5
}
```

WiderType: “Širší typ”

Je double širší typ než long?

- `sizeof(long) == 8 == sizeof(double)`
- oba typy jsou schopny reprezentovat (rozlišit) **stejný** počet čísel
- typ `double` reprezentuje čísla v pohyblivé řádové čárce
 - **nemůže reprezentovat všechna celá čísla typu `int64_t`**

```
#include <iostream>

int main() {
    double a = 1.0;
    double b = a;

    for (int i = 0; i < 1000; ++i)
        b += 1.0;

    std::cout
        << (b == a) << ", " << (b - a)
        << std::endl;
}
```

*// výstup: 0, 1000
// výstup by měl být nezávislý na hodnotě a*

```
#include <iostream>

int main() {
    double a = 1.0e50;
    double b = a;

    for (int i = 0; i < 1000; ++i)
        b += 1.0;

    std::cout
        << (b == a) << ", " << (b - a)
        << std::endl;
}
```

*// výstup: 1, 0
// !!!!!!!!*

Algoritmus `for_each`

- Aplikace funktoru na každý typ v sekvenci (**runtime**)
- **Sekvence typů**
- **Funkční objekt (funktör)**
 - operátor volání funkce parametrizovaný typem (šablona)
- Při překladač musí vzniknout instance operátoru pro všechny typy

```
typedef boost::mpl::vector<char, short, int, long> types;

struct {
    template <typename T>
    operator()() { ... }
} f;

int main() {
    for_each<types>(f);

    /* algoritmus for_each provede (za běhu programu!):
    f.operator()<char >(); // keyword operator f<char>(); f()<char>;
    f.operator()<short>();
    f.operator()<int >();
    f.operator()<long >();
    */
}
```

for_each: příklad použití

```
#include <cstdint>
#include <iostream>
#include <typeinfo>
#include <boost::mpl::vector.hpp>

struct {
    template <typename T> // parametrizace typem
    void operator()() {
        std::cout
            << "type name: " << typeid(T).name()
            << ", byte size: " << sizeof(T)
            << std::endl;
    }
} f; // funktor (funkční objekt)

typedef boost::mpl::vector<
    char, signed char, short, int, long, long long
> c_types;

typedef boost::mpl::vector<
    int8_t, int16_t, int32_t, int64_t
> stdint_types;

int main() {
    for_each<c_types>(f);
    std::cout << std::endl;
    for_each<stdint_types>(f);
}
```

Výstup pro GNU g++ 4.7.2, Linux, x86_64:

```
type name: c, byte size: 1
type name: a, byte size: 1
type name: s, byte size: 2
type name: i, byte size: 4
type name: l, byte size: 8
type name: x, byte size: 8
```

```
type name: a, byte size: 1
type name: s, byte size: 2
type name: i, byte size: 4
type name: l, byte size: 8
```

Výstup implikuje:

```
typedef signed char int8_t
typedef short      int16_t
typedef int        int32_t
typedef long       int64_t
```

char a signed char jsou různé typy
long a long long jsou různé typy

for_each: iterace přes typy v sekvenci

```
using namespace boost::mpl;

// primární šablona (PŠ): průchod sekvencí
template <typename B, typename E>
struct ForEach {
    static void execute() {
        ForEach<typename next<B>::type, E>::execute();
    }
};

// částečná specializace (ČS): ukončení průchodu
template <typename E>
struct ForEach<E, E> {
    static void execute() { }
};

// wrapper (obálka) - pohodlnější spouštění
// místo ForEach<S>::execute(); stačí for_each<S>();
template <typename S>
void for_each() {
    ForEach<
        typename begin<S>::type, typename end<S>::type
    >::execute();
}

typedef vector<int, long> typ;

int main() {
    for_each<types>();
}
```

- Rekurzivní volání funkce execute() struktury ForEach s různými argumenty šablony
- Argumenty šablony = iterátory pro danou sekvenci
- vector<int, long> má 3 iterátory:
 1. iterátor pro int
 2. iterátor pro long
 3. koncový iterátor
- Překlad programu:
 - rekurzivní vznik 3 instancí ForEach
- Běh programu
 - rekurzivní volání funkce execute 3 instancí ForEach

Instance	B	E
I. (PŠ)	begin<S>::type ≡ iterátor pro int	end<S>::type ≡ koncový iterátor
II. (PŠ)	next::type ≡ iterátor pro long	end<S>::type ≡ koncový iterátor
III. (ČS)	next::type ≡ koncový iterátor	end<S>::type ≡ koncový iterátor

for_each: „čistší“ definice

```
using namespace boost::mpl;

// primární šablona (PŠ): průchod sekvencí
template <typename B, typename E>
struct ForEach {
    static void execute() {
        ForEach<typename next<B>::type, E>::execute();
    }
};

// částečná specializace (ČS): ukončení průchodu
template <typename E>
struct ForEach<E, E> {
    static void execute() { }
};

// wrapper (obálka) - pohodlnější spouštění
// místo ForEach<S>::execute(); stačí for_each<S>();
template <typename S>
void for_each() {
    ForEach<
        typename begin<S>::type, typename end<S>::type
    >::execute();
}

typedef vector<int, long> types;

int main() {
    for_each<types>();
}
```

```
using namespace boost::mpl;

// primární šablona (PŠ) - průchod sekvencí
template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForEach {
    static void execute() {
        ForEach<S, typename next<B>::type, E>::execute();
    }
};

// částečná specializace (ČS) - ukončení průchodu sekvencí
template <typename S, typename E>
struct ForEach<S, E, E> {
    static void execute() { }
};

// wrapper (obálka) - pohodlnější spouštění
template <typename S>
void for_each() {
    ForEach<S>::execute();
}

typedef vector<int, long> types;

int main() {
    for_each<types>();
}
```

for_each: aplikace funkčního objektu

```
using namespace boost::mpl;

// primární šablona (PŠ) - průchod sekvencí
template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForEach {
    static void execute() {
        ForEach<S, typename next<B>::type, E>::execute();
    }
};

// částečná specializace (ČS) - ukončení průchodu sekvencí
template <typename S, typename E>
struct ForEach<S, E, E> {
    static void execute() { }
};

// wrapper (obálka) - pohodlnější spouštění
template <typename S>
void for_each() {
    ForEach<S>::execute();
}

typedef vector<int, long> types;

int main() {
    for_each<types>();
}
```

```
using namespace boost::mpl;

template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForEach {
    template <typename T>
    static void execute(T& f) {
        f.template operator()<typename deref<B>::type>();
        ForEach<S, typename next<B>::type, E>::execute(f);
    }
};

template <typename S, typename E>
struct ForEach<S, E, E> {
    template <typename T>
    static void execute(T& f) { }
};

template <typename S, typename T>
void for_each(T& f) {
    ForEach<S>::execute(f);
}

typedef vector<int, long> S;
struct { ... } f;

int main() {
    for_each<types>(f);
}
```

for_each: ukázka

```
using namespace boost::mpl;

template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForEach {
    template <typename T>
    static void execute(T& f) {
        f.template operator()<typename deref<B>::type>();
        ForEach<S, typename next<B>::type, E>::execute(f);
    }
};

template <typename S, typename E>
struct ForEach<S, E, E> {
    template <typename T>
    static void execute(T& f) { }
};

template <typename S, typename T>
void for_each(T& f) {
    ForEach<S>::execute(f);
}

typedef vector<int, long> types;
struct { template <typename T> void operator()(T&) const {} } f;

int main() {
    for_each<types>(f);
}
```

- Překlad programu:
 - rekurzivní vznik 3 instancí ForEach
 - vznik 2 instancí operátoru ()
funktoru: <int> a <long>
- Běh programu
 - rekurzivní volání funkce execute 3 instancí ForEach:
 1. f.template operator()<int>();
 2. f.template operator()<long>();
 3. {}

Instance	B	E
I. (PŠ)	begin<S>::type ≡ iterátor pro int	end<S>::type ≡ koncový iterátor
II. (PŠ)	next::type ≡ iterátor pro long	end<S>::type ≡ koncový iterátor
III. (ČS)	next::type ≡ koncový iterátor	end<S>::type ≡ koncový iterátor

Algoritmus ~~for_each~~ for_id

- Aplikace funktoru na ~~každý~~ požadovaný typ v sekvenci (runtime)
 - požadovaný typ určen jeho pozicí v sekvenci známou až za běhu programu !!!
- Sekvence typů
- Funkční objekt (funktör)
 - operátor volání funkce parametrizovaný typem (šablona)
- Při překladač musí vzniknout instance operátoru pro všechny typy

```
typedef boost::mpl::vector<char, short, int, long> types;
// pozice typů v sekvenci:  0      1      2      3

struct {
    template <typename T>
    operator>() { ... }
} f;

int main(int argc, char* argv[]) {
    int id = boost::lexical_cast<int>(argv[1]);
    for_id<types>(f, id);

    /* pro id == 2 provede algoritmus for_id (za běhu programu!):
    f.operator<int>();
    */
}
```

for_id: funkcionalita

- Generické programování: argumenty šablon musejí být známe v **čase překladu**
- Použití for_id:
 1. vytvoření více instancí šablony
 2. výběr instance podle parametru známého za **běhu programu**
- Pro uživatele je tento proces skrytý
- **Výsledný efekt \approx výběr argumentu šablony za běhu programu**
- Příklad pro Paraperm:

Omezení:

1. Generický kód musí být obalen funktorem
2. Za překladu vzniká více instancí šablon, za běhu programu je (pravděpodobně) využita jen jedna
 - \Rightarrow delší doba překladu
 - \Rightarrow větší programové soubory / kódové segmenty

```
// algoritmus využívající Paraperm ve formě funktoru
struct {
    template <typename T> void operator>()() {
        ...
        paraperm::Paraperm<T> generator;
        ...
    }
} algorithm;

typedef boost::mpl::vector<uint32_t, uint64_t> types;

int main() {
    uint64_t n = boost::lexical_cast<uint64_t>(argv[1]); // velikost permutace
    int id = (n > (1UL << 32)) ? 1 : 0; // pozice (id) požadovaného typu pro prvky permutace
    for_id<types>(algorithm, id);
}
```

for_id: definice a porovnání s for_each

```
// primární šablona (PŠ) - průchod sekvencí
template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForEach {
    template <typename T>
    static void execute(T& f) {

        f.template operator()<typename deref<B>::type>();

        ForEach<S, typename next<B>::type, E>::execute(f);
    }
};

// částečná specializace (ČS) - ukončení průchodu sekvencí
template <typename S, typename E>
struct ForEach<S, E, E> {
    template <typename T>
    static void execute(T& f) {

    }
};

// wrapper (obálka) - pohodlnější spouštění
// místo ForEach<S>::execute(f); stačí for_each<S>(f);
template <typename S, typename T>
void for_each(T& f) {
    ForEach<S>::execute(f);
}
```

```
// primární šablona (PŠ) - průchod sekvencí
template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForId {
    template <typename T>
    static void execute(T& f, int id) {
        if (distance<typename begin<S>::type, B>::value == id)
            f.template operator()<typename deref<B>::type>();
        else
            ForId<S, typename next<B>::type, E>::execute(f, id);
    }
};

// částečná specializace - ukončení průchodu sekvencí
template <typename S, typename E>
struct ForId<S, E, E> {
    template <typename T>
    static void execute(T& f, int id) {
        throw std::runtime_error( ... ); // špatný argument id
    }
};

// wrapper (obálka) - pohodlnější spouštění
// místo ForId<S>::execute(f); stačí for_id<S>(f);
template <typename S, typename T>
void for_id(T& f, int id) {
    ForId<S>::execute(f, id);
}
```

for_id: ukázka

```
template <
    typename S,
    typename B = typename begin<S>::type,
    typename E = typename end<S>::type
>
struct ForId {
    template <typename T>
    static void execute(T& f, int id) {
        if (distance<typename begin<S>::type, B>::value == id)
            f.template operator()<typename deref<B>::type>();
        else
            ForId<S, typename next<B>::type, E>::execute(f, id);
    }
};

template <typename S, typename E>
struct ForId<S, E, E> {
    template <typename T>
    static void execute(T& f, int id) {
        throw std::runtime_error( ... );
    }
};

template <typename S, typename T>
void for_id(T& f, int id) {
    ForId<S>::execute(f, id);
}

typedef vector<int, long> types;
struct { template <typename T>

int main() { for_id<types>(f, 1
```

Překlad programu:

- rekurzivní vznik 3 instancí ForId
- vznik 2 instancí operátoru () funktoru: <int> a <long>

Běh programu

- rekurzivní volání funkce execute instancí ForId:
 1. distance<iterátor pro int, iterátor pro int>::value není rovno 1 \Rightarrow přechod k další instanci
 2. distance<iterátor pro int, iterátor pro long>::value je rovno 1 \Rightarrow f.template operator()<long>();

Instance	B	E
I. (PŠ)	begin<S>::type \equiv iterátor pro int	end<S>::type \equiv koncový iterátor
II. (PŠ)	next::type \equiv iterátor pro long	end<S>::type \equiv koncový iterátor
III. (ČS)	next::type \equiv koncový iterátor	end<S>::type \equiv koncový iterátor

for_id: rozšíření do dvou dimenzí

příklad: výpočet se čtvercovou řádkou maticí

Volba datového typu pro:

1. řádkové / sloupcové indexy nenulových elementů
2. hodnoty maticových elementů

Jak zvolit typy?

- pro indexy: podle velikosti matice
 - $n \leq 256 \Rightarrow \text{uint8_t}$, $n \leq 65536 \Rightarrow \text{uint16_t}$, $n \leq 2^{32} \Rightarrow \text{uint32_t}$, jinak $\Rightarrow \text{uint64_t}$
- pro hodnoty: podle požadované přesnosti výpočtu / dostupných zdrojů (paměť, čas)

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & -0.5 \\ 0 & 1 & 0 & 0 & 0 & -0.5 \\ 0 & 0 & 1 & 0 & 0 & -0.5 \\ 0 & 0.5 & 0 & 1 & 0 & -0.5 \\ 0 & 0 & 0.5 & 0 & 1 & -0.5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
template <typename T, typename U> // T - typ pro řádkové a sloupcové indexy, U - typ pro hodnoty
struct SquareMatrix {
    uintmax_t n; // = 6
    std::vector<T> rows; // = { 0, 0, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5 }
    std::vector<T> cols; // = { 0, 5, 1, 5, 2, 5, 1, 3, 5, 2, 4, 5, 5 }
    std::vector<U> vals; // = { 1, -.5, 1, -.5, 1, -.5, .5, 1, -.5, .5, 1, -.5, 1 }
};

int main() {
    SquareMatrix<uint8_t, float> matrix;
    matrix.n = 6;
    matrix.rows.push_back(0); matrix.cols.push_back(0); matrix.vals.push_back(1.0);
    ...

    return 0;
}
```

for_id: rozšíření do dvou dimenzí

příklad: výpočet se čtvercovou řádkou maticí

```
typedef boost::mpl::vector<uint8_t, uint16_t, uint32_t, uint64_t> ind_types;

int get_ind_id(uint64_t n) {
    if (n <= 256)          return 0;
    else if (n <= 65536)    return 1;
    else if (n <= (1UL << 32)) return 2;
    else                   return 3;
}

typedef boost::mpl::vector<float, double> fp_types;

struct {
    template <typename T, typename U>
    void operator()() {
        SquareMatrix<T, U> matrix;
        ... // konstrukce maticových elementů
        ... // výpočet (hledání vlastních čísel, řešení lineární soustavy rovnic, ...)
    }
} algorithm;

int main(int argc, char* argv[]) {
    uint64_t n = ...
    int ind_id = get_int_id(n);
    int fp_id = ... // volba přesnosti výpočtu např. parametrem příkazové řádky

    for_id<ind_types, fp_types>(algorithm, ind_id, fp_id);
}
```

for_id 2D: implementace

// primární šablona - průchod první sekvencí

```
template < typename S1, typename S2,
          typename B1 = typename begin<S1>::type, typename B2 = ty
          typename E1 = typename end<S1>::type, typename E2 = type
          typename T1 = typename deref<B1>::type>
struct ForId2 {
    template <typename T> static void execute(T& f, int id1, int id2) {
        if (distance<typename begin<S1>::type, B1>::value == id1)
            // f.template operator()<typename deref<B1>::type, ???>();
            ForId2<S1, S2, E1, B2, E1, E2, typename deref<B1>::type>::execute(f, id1, id2);
        else
            ForId2<S1, S2, typename next<B1>::type, B2, E1, E2, T1>::execute(f, id1, id2);
    }
};
```

```
template <typename S1, typename S2, typename T>
void for_id(T& f, int id1, int id2) {
    ForEach<S1, S2>::execute(f, id1, id2);
}
```

// částečná specializace I. - průchod druhou sekvencí

```
template <typename S1, typename S2, typename B2, typename E1, typename E2, typename T1>
struct ForId2<S1, S2, E1, B2, E1, E2, T1> {
    template <typename T> static void execute(T& f, int id1, int id2) {
        if (distance<typename begin<S2>::type, B2>::value == id2)
            f.template operator()<T1, typename deref<B2>::type>();
        else
            ForId2<S1, S2, E1, typename next<B2>::type, E1, E2, T1>::execute(f, id1, id2);
    }
};
```

// částečná specializace II. - ukončení průchodu sekvencí

```
template <typename S1, typename S2, typename E1, typename E2, typename T1>
struct ForId2<S1, S2, E1, E2, E1, E2, T1> {
    template <typename T> static void execute(T& f, int id1, int id2) { throw std::runtime_error( ... ); }
};
```

for_id: rozšíření do více dimenzí

- Příklad: obdelníková matice
 - rozdílené typy pro řádkové a sloupcové indexy
- Stejný princip (vzor)
 - primární šablona + částečné specializace pro iterace pro každou sekvenci (dimenzi)
- Příliš velké množství definic
 - pro každou dimenzi D je potřeba D+1 definic (primární šablona + D částečných specializací)
 - pro dimenze 1, ..., Dmax je potřeba $D_{\max} \times (D_{\max} + 3) / 2 = O(D_{\max}^2)$ definic
- Lze implementovat pomocí $2 \times D_{\max} + 1 = O(D_{\max})$ definic
 1. aktuální dimenze problému – parametr šablony
 2. delegování volání operátoru na pomocnou třídu (Executor)

```
template <typename T, typename U, typename V>
struct RectangularMatrix {
    uintmax_t n;
    std::vector<T> rows;
    std::vector<U> cols;
    std::vector<V> vals;
};
```

for_id: sjednocení definic pro 1 a 2 dimenze

```
template < int D, // D je 1 nebo 2
          typename S1, typename S2,
          typename B1 = typename begin<S1>::type, typename B2 = ty
          typename E1 = typename end<S1>::type, typename E2 = type
          typename T1 = typename deref<B1>::type>
struct ForId12 {
    template <typename T> static void execute(T& f, int id1, int id2)
    if (distance<typename begin<S1>::type, B1>::value == id1)
        if (D == 1)
            // f.template operator()<typename deref<B1>::type>();
            Executor<D, typename deref<B1>::type, T2>::execute(f);
        else
            ForId12<D, S1, S2, E1, B2, E1, E2, typename deref<B1>::type>
        else
            ForId12<D, S1, S2, typename next<B1>::type, B2, E1, E2, T1>::
    };

template <int D, typename S1, typename S2, typename B2, typename E1
struct ForId12<D, S1, S2, E1, B2, E1, E2, T1> {
    template <typename T> static void execute(T& f, int id1, int id2)
    if (distance<typename begin<S2>::type, B2>::value == id2)
        // f.template operator()<T1, typename deref<B2>::type>();
        Executor<D, T1, typename deref<B2>::type>::execute(f);
    else
        ForId12<D, S1, S2, E1, typename next<B2>::type, E1, E2, T1>::execute(f, id1, id2);
    };

template <typename S1, typename S2, typename E1, typename E2, typename T1>
struct ForId12<D, S1, S2, E1, E2, E1, E2, T1> {
    template <typename T> static void execute(T& f, int id1, int id2) { throw std::runtime_error( ... ); }
};

template <int D, typename T1, typename T2>
struct Executor;

// částečná specializace pro 1D problém
template <typename T1, typename T2>
struct Executor<1, T2, T2> {
    template <typename T>
    static void execute(T& f) {
        f.template operator()<T1>();
    }
};

// částečná specializace pro 2D problém
template <typename T1, typename T2>
struct Executor<2, T2, T2> {
    template <typename T>
    static void execute(T& f) {
        f.template operator()<T1, T2>();
    }
};
```

využití Boost Preprocessor Library

- *“Rather than being written out by hand, mechanical-looking code should really be generated mechanically.”*
- *“The Boost Preprocessor library plays a role in preprocessor metaprogramming similar to the one played by the MPL in template metaprogramming: It supplies a framework of high-level components that make otherwise-painful metaprogramming jobs approachable.”*

[illegible]

```
#define MASCOT_MPL_FI_MAX 5
#include <mascot/for_id.h>
```

```
using namespace mascot::mpl;
```

```
typedef boost::mpl::vector<...> types1;
```

```
typedef boost::mpl::vector<...> types5;
```

```
int main() {
```

```
int id1 = ...
```

```
int id5 = ...
```

```
for_id<types1>(f, id1);
```

```
for_id<types1, types2>(f, id1, id2);
```

```
for_id<types1, ..., types4>(f, id1, ..., id4);
```

```
for_id<types1, ..., types5>(f, id1, ..., id5);
```

```
return 0;
```

}

for_id (BOOST): kompatibilita

Compiler vendor	Compiler version	Boost version	Processor architecture	Operating system	Compilation errors
Cray	8.0.7	1.47.0	AMD x86_64	Linux	YES
GNU	4.7.1	1.47.0	AMD x86_64	Linux	NO
IBM	11.1	1.40.0	IBM POWER7	AIX	YES
Intel	12.1.5	1.47.0	AMD x86_64	Linux	NO
Microsoft	16.0.0.30319 01	1.48.0	Intel x86_64	Windows	YES
PathScale	4.0.12.1	1.47.0	AMD x86_64	Linux	NO
PGI	12.5-0	1.47.0	AMD x86_64	Linux	NO

- Cray: `rvalue references`, `BOOST_STATIC_ASSERT`
- IBM, Microsoft: `f.template operator<T>()`
 - řešení: pojmenovaná funkce namísto operátoru

for_id: **template code bloat**
počet generovaných instancí operátoru ()

Length L	Problem dimension d					
	1	2	3	4	5	6
1	1	1	1	1	1	1
2	2	4	8	16	32	64
3	3	9	27	81	243	729
4	4	16	64	256	1024	4096
5	5	25	125	625	3125	15625
6	6	36	216	1296	7776	46656

for_id: **template code bloat**
čas překladu [s] (GNU g++ 4.4.6)

Length L	Problem dimension d					
	1	2	3	4	5	6
1	0.4	0.4	0.4	0.4	0.4	0.4
2	0.4	0.4	0.5	0.5	0.6	0.8
3	0.4	0.4	0.5	0.9	1.7	5.6
4	0.4	0.5	0.7	1.7	8.1	109.6
5	0.4	0.5	1.0	3.9	57.3	2405.2
6	0.4	0.6	1.4	11.0	514.5	N/A

for_id: template code bloat

paměťové nároky překladu [MB] (GNU g++ 4.4.6)

Length L	Problem dimension d					
	1	2	3	4	5	6
1	157.0	158.0	159.2	159.0	160.2	161.3
2	158.0	160.3	162.3	166.7	173.1	185.0
3	159.1	162.3	168.9	184.0	226.5	351.1
4	160.2	165.6	179.7	225.5	401.1	635.4
5	161.3	170.0	194.8	304.6	524.9	1842.8
6	162.3	174.3	215.6	407.8	1094.0	N/A

for_id: runtime overhead

[ns] (GNU g++ 4.4.6)

- doba spuštění operátoru u 2D problému
 1. `f.operator()<uint16_t, float>();`
 2. `f.operator()<uint32_t, double>();`
 3. `for_id<ind_types, fp_types>(f, id1, id2);`
- statistické údaje z 200 měření
- aplikace: hledání vlastních čísel testovací matice (mocninná metoda)

Action	Statistic	C_{16}^f	C_{32}^d	C_*^f
invocation	mean value	191.1	191.0	477.9
	median	186.0	187.0	470.0
	standard deviation	23.0	25.1	43.1
application	mean value	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$
	median	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$	$6.1 \cdot 10^8$
	standard deviation	$6.9 \cdot 10^6$	$7.7 \cdot 10^6$	$6.6 \cdot 10^6$

Převod čísla na typ / volba funkce

Andrei Alexandrescu – Modern C++

- Výběr rozdílné funkce v závislosti na výsledku výpočtu při kompilaci
- Příklad: duplikace objektu
 1. pomocí funkce `clone()`, pokud existuje
 2. jinak pomocí `kopírovacího konstruktoru`
- Řešení pomocí větvení (if, switch) – **všechny větve musí být přeložitelné:**

```
template <typename T, bool HAS_CLONE>
class Duplicator {
public:
    static T* duplicate(T* obj) {
        if (HAS_CLONE)
            return obj->clone(); // chyba překladač pro typ T, který nemá členskou funkci clone
        else
            return new T(*obj);
    }
};
```

Převod čísla na typ / volba funkce

Andrei Alexandrescu – Modern C++

- Výběr rozdílné funkce v závislosti na výsledku výpočtu při kompilaci
- Příklad: duplikace objektu
 1. pomocí funkce `clone()`, pokud existuje
 2. jinak pomocí `kopírovacího konstruktora`
- Řešení pomocí `převodu čísla na typ`:

```
// generuje rozdílný typ pro různé argumenty I
template <int I>
struct Int2Type {
    enum { value = I };
};
```

```
template <typename T, bool HAS_CLONE>
class Duplicator {
private:
    static T* duplicate(T* obj, Int2Type<true>) {
        return obj->clone();
    }

    static T* duplicate(T* obj, Int2Type<true>) {
        return new T(*obj);
    }

public:
    static T* duplicate(T* obj) {
        return duplicate(obj, Int2Type<HAS_CLONE>());
    }
};
```

Couriously Reccuring Template Pattern (CRTP)

příklad: počítání instancí tříd

```
class SomeClass
public:
    SomeClass() { // konstruktor
        ++count_;
        ...
    }

    SomeClass(SomeClass const &) { // kopírovací konstruktor
        ++count_;
        ...
    }

    ~SomeClass() { // destruktork
        ...
        --count_;
    }

    static size_t count() {
        return count_;
    }

private:
    // počet „živých“ instancí typu SomeClass
    static size_t count_;
};

// inicializace
size_t SomeClass::count_ = 0;
```

- Nevýhoda: **nutnost implementace v každé „počítatelné“ třídě**
 - redundance kódu
 - možnost vzniku chyb

Couriously Reccuring Template Pattern (CRTP)

příklad: počítání instancí tříd

```
template <typename CountedType>
class ObjectCounter {
protected:
    ObjectCounter() { ++count_; }

    ObjectCounter(ObjectCounter<CountedType> const &) {
        ++count_;
    }

    ~ObjectCounter() { --count_; }

public:
    static size_t count() {
        return count_;
    }

private:
    // počet „živých instancí“ typu ObjectCounter
    // počet „živých instancí“ typu ObjectCounter<CountedType>
    static size_t count_;
};

// inicializace
template <typename CountedType>
size_t SomeClass<CountedType>::count_ = 0;
```

```
class SomeClass
    : public ObjectCounter<SomeClass> {
    ...
};

class SomeOtherClass
    : public ObjectCounter<SomeOtherClass> {
    ...
};

int main() {
    SomeClass obj1, obj2;
    SomeClass* obj3 = new SomeClass();

    SomeOtherClass obj4;

    std::cout << SomeClass::count() << std::endl;
    std::cout << SomeOtherClass::count()
               << std::endl;

    delete obj3;
    return 0;
}

/* výstup programu:
3
1
*/
```

Návrhový vzor Bridge

Dynamický vs statický polymorfismus

- Jeden z cílů návrhového vzoru Bridge je výběr implementace pro určité rozhraní
- Standardní implementace – implementace je odvozena od abstraktní báze třídy (**dynamický polymorfismus**)
 - výhoda: možnost výběru implementace v čase běhu programu (runtime)
 - nevýhoda: runtime overhead, ukazatele
- Implementace pomocí šablon (**statický polymorfismus**) – typ implementační třídy je dán parametrem šablony

Figure 14.3. Bridge pattern implemented using inheritance

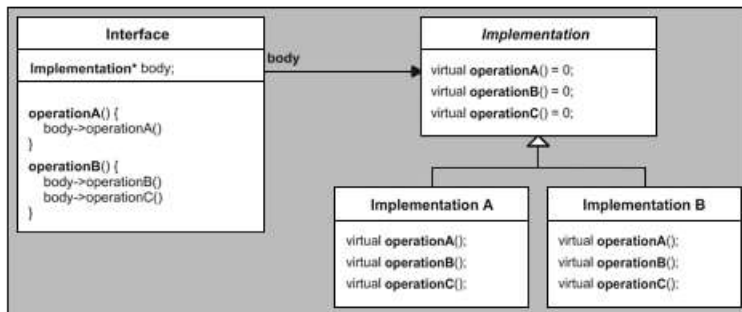
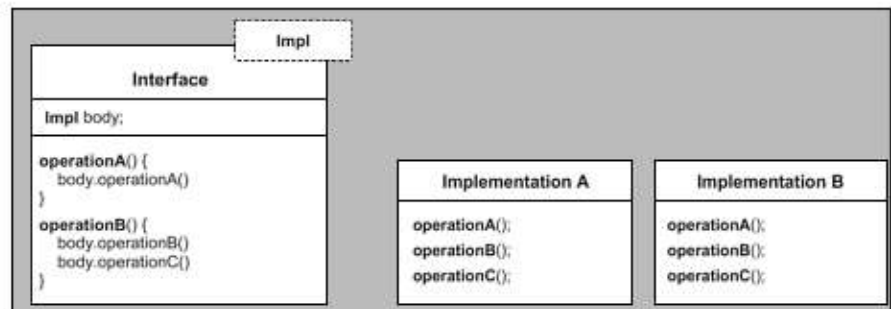


Figure 14.4. Bridge pattern implemented using templates



Knihovna Loki

Andrei Alexandrescu, <http://loki-lib.sourceforge.net/>

- „Loki is a C++ library of designs, containing flexible implementations of common *design patterns and idioms*.“
- “The library makes extensive use of C++ *template metaprogramming* and implements several commonly used tools: *typelist, functor, singleton, smart pointer, object factory, visitor and multimethods*.”
- Idiomy:
 - multiple dispatcher, pimpl (pointer to implementation), scope guard pointer, smart pointer, compile time check
- Návrhové vzory:
 - factory method, abstract factory, singleton, visitor, command
- ...
- A. Alexandrescu, Modern C++ Design (Moderní programování v C++)

Knihovna Loki / Modern C++ Design

Multimetody

- Mechanismus virtuálních funkcí v C++ umožňuje volbu volání funkce (dispatch) na základě dynamického typu objektu (**single dispatch**)
- Tzv. **multiple dispatch** – volba volání funkce na základě typů více objektů
 - v C++ není multiple dispatch přirozeně podporován
- Emulace multiple dispatch:
 1. \approx for_id
 2. **multimetody**
- Příklad:

```
class Object {  
    public: virtual void draw() = 0;  
};  
  
class Square : public Object {  
    public: virtual void draw() { ... }  
};  
  
class Circle : public Object {  
    public: virtual void draw() { ... }  
};
```

```
// vykreslení průniku dvou objektů  
void intersection(Object& obj1, Object& obj2) {  
    ?????  
}
```

Knihovna Loki / Modern C++ Design

Multimethody

- Implementace:
 - kombinace template metaprogramming a dědičnosti
 - sekvence možných typů (Square, Circle, ...)
 - iterace přes sekvence a porovnávání typů (dynamic_cast, typeid, ...)
 - volání konkrétní funkce na základě výsledku iterací a hledání v sekvencích typů