

# Programování se šablonami

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

© Karel Richta, Martin Hořeňovský, Aleš Hrabalík, 2021

Programování v C++, B6B36PCC

11/2024, Lekce 10a

<https://cw.fel.cvut.cz/wiki/courses/b6b36pcc/start>



# Pojmy

- Budeme využívat tzv. *šablony* k tomu, abychom vytvořili *generické algoritmy*, tj. algoritmy nezávislé na použitých typech.
- Tento styl programování se označuje:
  - *Generické programování (generic programming)*
  - *Metaprogramování se šablonami (template metaprogramming)*
- Existují dva druhy šablon:
  - *Šablony funkcí*
  - *Šablony tříd*

# Šablony funkcí

# Šablona funkce minimum

Mějme funkci `minimum`, která ze dvou čísel vybere to menší tak, že je porovná operátorem `<`:

```
double minimum(double a, double b) {  
    if (a < b) {  
        return a;  
    }  
    else {  
        return b;  
    }  
}
```

*// nebo:*

```
double minimum(double a, double b) { return (a < b)? a: b; }
```

# Šablona funkce `minimum`

- Funkce `minimum` v 99 % případů funguje.
- Překvapení nastává pouze v případě, že si neuvědomíme, že po použití `minimum` najednou pracujeme s hodnotou typu `double`.
- Navíc, pokud *nechceme* pracovat s typem `double`, `minimum` je neefektivní.
  - Převést hodnotu na typ `double` stojí čas a peníze.
  - Operace s reálnými čísly jsou významně dražší než operace s celočíselnými typy.
- Také dojde ke ztrátě přesnosti, pokud chceme pracovat s typy `long double`, `long long`, nebo `unsigned long long`.
- Proto je žádoucí, aby funkce `minimum` podporovala více typů.

# Šablona funkce minimum

- Aby funkce `minimum` podporovala více typů, můžeme ji přetížit:

```
double minimum(double a, double b) { return (a < b)? a: b; }  
int minimum(int a, int b) { return (a < b)? a: b; }
```

- Aby `minimum` podporovala všechny typy, nahradíme funkci `minimum` šablonou funkce `minimum`:

```
template<typename T>  
T minimum(T a, T b) { return (a < b)? a: b; }
```

- Přidali jsme řádek `template<typename T>`, `T` je šablonový parametr.
- Všechny výskyty typu `double` jsme nahradili parametrem `T`.

# Šablona funkce `minimum`

- Šablona funkce `minimum` se chová, jako kdybychom funkci `minimum` přetížili pro všechny typy.

```
#include <iostream>
```

```
template<typename T>
```

```
T minimum(T a, T b) { return (a < b)? a: b; }
```

```
int main() {
```

```
    std::cout << minimum(1, 2) << "\n";           // int
```

```
    std::cout << minimum(4u, 3u) << "\n";         // unsigned
```

```
    std::cout << minimum(2.3f, 3.4f) << "\n";     // float
```

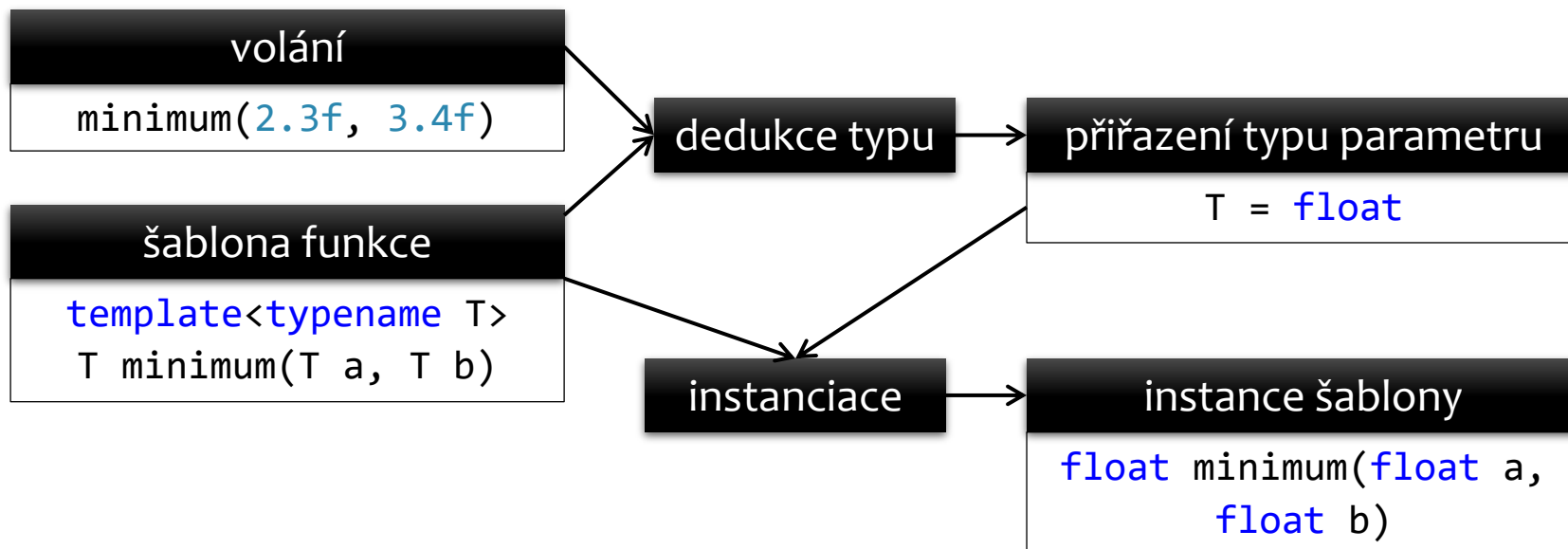
```
    std::cout << minimum(4.3, 3.2) << "\n";       // double
```

```
    std::cout << minimum('a', 'A') << "\n";      // char
```

```
}
```

# Šablona funkce `minimum`

- Tím, že zavoláme `minimum`, donutíme kompilátor odvodit, kterou variantu funkce `minimum` chceme. Toto odvození se nazývá *dedukce typu* (*template parameter type deduction*).
- Kompilátor posléze vytvoří funkci, která se má zavolat. Tento proces se nazývá *instanciace*. Vytvořené funkce už pracují s konkrétními typy a nazývají se *instance*.



# Šablona funkce `minimum`

- Stejně jako přetěžování, **instanciace probíhá během kompilace**.
- Podobně jako u přetěžování musí být *jednoznačně* určeno, kterou instanci chceme:

```
minimum(1.1, 2); // Chyba, argument šablony nelze odvodit
```

- Zde kompilátor prohlásí, že si není jistý; chtěli jsme instanci `double minimum(double, double)`, nebo `int minimum(int, int)`? Toto dilema můžeme vyřešit tak, že explicitně určíme typ parametru T:

```
minimum<int>(1.1, 2); // OK, výsledek je 1  
minimum<double>(1.1, 2); // OK, výsledek je 1.1
```

# Šablona funkce `minimum`

- Volání `minimum` funguje pro všechny typy, které umějí operaci `<`.

```
minimum<std::string>("abc", "def"); // OK, výsledek je abc
```

- Takový typ si umíme i vytvořit. Autor šablony `minimum` nemusí mít žádnou informaci o tom, že někdo v budoucnu vytvoří typ `Date`:

```
struct Date {  
    int year, month, day;  
    Date(int y, int m, int d): year(y), month(m), day(d) {}  
};  
bool operator<(const Date& l, const Date& r) { ... }
```

```
minimum(Date(2002, 9, 2), Date(2003, 1, 1)); // OK
```

Šablona třídy

# Šablona třídy Vector

- Třídu Vector známe ze cvičení.

```
class Vector {  
public:  
    Vector();  
    Vector(const Vector&);  
    ~Vector();  
    void push_back(double d);  
    void pop_back();  
    double& back();  
    double back() const;  
    double& operator[](int i);  
    double operator[](int i) const;  
    int size() const;  
    bool empty() const;  
private:  
    // ...  
};
```

# Šablona třídy Vector

- Nevýhodou třídy Vector je, že pracuje pouze s prvky typu double. Rádi bychom se přiblížili třídě `std::vector` a mohli typ obsažených prvků stanovit.

```
Vector<double> vd;  
vd.push_back(2.3);  
vd.push_back(3.4);
```

```
Vector<std::string> vs;  
vs.push_back("abc");  
vs.push_back("def");
```

- Proto musíme třídu Vector nahradit šablonou třídy Vector se šablonovým parametrem T.

# Šablona třídy Vector

- Stejně jako u funkcí: přidej template a výskyty double nahrad' T.

```
class Vector {  
public:  
    Vector();  
    Vector(const Vector&);  
    ~Vector();  
    void push_back(double d);  
    void pop_back();  
    double& back();  
    double back() const;  
    double& operator[](int i);  
    double operator[](int i) const;  
    int size() const;  
    bool empty() const;  
private:  
    double* data;  
    ...  
};
```



```
template<typename T>  
class Vector {  
public:  
    Vector();  
    Vector(const Vector&);  
    ~Vector();  
    void push_back(T d);  
    void pop_back();  
    T& back();  
    T back() const;  
    T& operator[](int i);  
    T operator[](int i) const;  
    int size() const;  
    bool empty() const;  
private:  
    T* data;  
    ...  
};
```

# Šablona třídy Vector

- Navíc pokud máme nějaké definice metod mimo deklaraci třídy, musíme pro každou z nich provést to samé.

```
void Vector::push_back(double d) { ... }
```

```
└─ template<typename T>  
   void Vector<T>::push_back(T d) { ... }
```

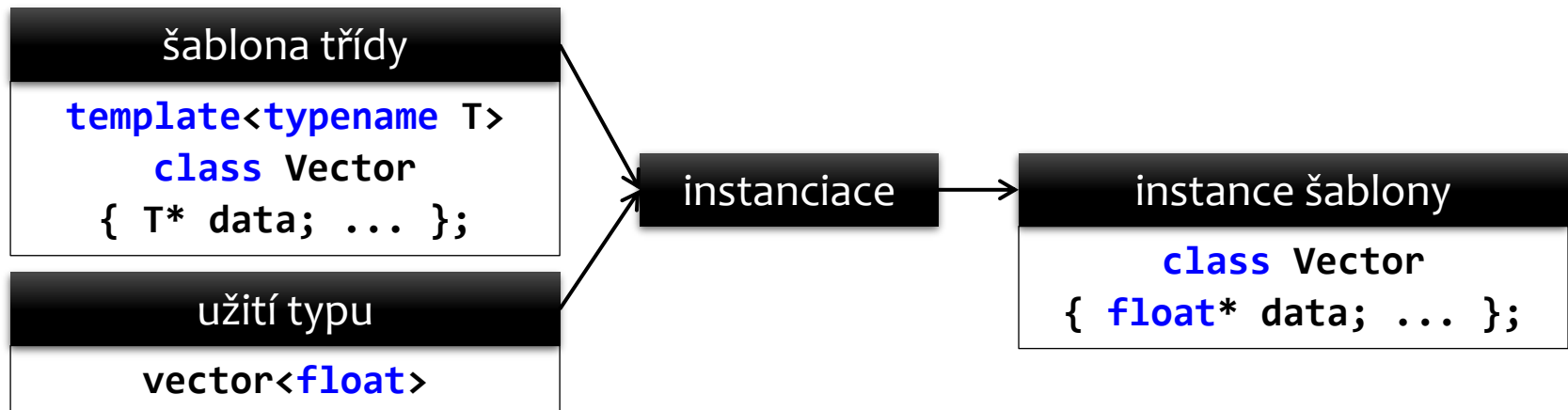
```
double Vector::back() const { ... }
```

```
└─ template<typename T>  
   T Vector<T>::back() const { ... }
```

- Vlastně jsme každou metodu nahradili šablonou metody.
- Všimněte si detailu, že `Vector::` jsme nahradili `Vector<T>::`.

# Šablona třídy Vector

- Nyní můžeme používat typy `Vector<int>`, `Vector<std::string>`, atd. Jedná se o *instance* šablony `Vector`, které vznikly při kompilaci *instanciací*.
- Samotný typ `Vector` (bez špičatých závorek `<>`) přestane být platný.
- Na rozdíl od šablon funkcí, u šablon tříd neprobíhá dedukce typu parametru. U funkcí nám pomáhaly argumenty volání, zde nemáme z čeho typ odvodit.



Příklad: šablona `Complex` pro reprezentaci komplexních čísel.

```
#include <iostream>
#include <iomanip>
```

```
template<typename T>
struct Complex {
    T real
    T imag;
    Complex(T r, T i): real(r), imag(i) {}
};
```

```
template<typename T>
std::ostream& operator<<(std::ostream& out, const Complex<T>& c) {
    out << c.real;
    out << std::showpos << c.imag << std::noshowpos;
    return out << "i";
}
```

- Šablonou funkce `operator<<` jsme přetížili operaci `<<` pro všechny instance šablony `Complex`. Operace vypíše komplexní číslo ve tvaru `2.34+1.23i`.

Příklad: šablona read pro čtení ze standardního vstupu.

**// definice:**

```
template<typename T>
T read(std::string prompt) {
    ...
}
```

**// použití:**

```
std::string jmeno = read("Jmeno: "); // error
std::string prijmeni = read("Prijmeni: "); // error
int vek = read("Vek: "); // error

std::string jmeno = read<std::string>("Jmeno: "); // OK, C++03
std::string prijmeni = read<std::string>("Prijmeni: "); // OK, C++03
int vek = read<int>("Vek: "); // OK, C++03

auto jmeno = read<std::string>("Jmeno: "); // OK, C++11
auto prijmeni = read<std::string>("Prijmeni: "); // OK, C++11
auto vek = read<int>("Vek: "); // OK, C++11
```

```

#include <iostream>
#include <stdexcept>

template<typename T>
T read(std::string prompt) {
    T result;

    while (true) {
        std::cout << prompt;
        std::cin >> result;

        if (std::cin) { // čtení dopadlo dobře
            return result; // vrátíme výsledek
        }
        else if (std::cin.fail()) { // něco se nepovedlo
            std::cin.clear(); // reset chybového stavu std::cin
            std::cin.ignore(999, '\n'); // zahod', co je na vstupu
            continue; // zkusíme číst znovu
        }
        else { // něco se hodně nepovedlo
            throw std::runtime_error("read(): std::cin je bad nebo eof");
        }
    }
}

```

Šablony – práce s parametry

# Předpokládej, že je nákladné zkopírovat T

- Zatím jsme měli za to, že typ T je (stejně jako double) malý a levně kopírovatelný.
- Tento předpoklad přestává platit, když se místo double v parametru T vyskytne něco velkého, nebo zacházejícího s drahými prostředky (paměť, soubory, sockety, mutexy, ...).
- Např. při vytvoření `std::string` se alokuje paměť, a při zničení `std::string` se paměť uvolňuje. Zbytečné kopie objektu typu `std::string` jsou nežádoucí.

```
Vector<std::string> vs;  
vdd.push_back("abc");           // zbytečná kopie  
vdd.push_back("def");          // zbytečná kopie  
std::cout << vdd.back() << "\n"; // zbytečná kopie  
std::cout << vdd[1] << "\n";    // zbytečná kopie
```

# Předpokládej, že je nákladné zkopírovat T

- Řešení: pro objekty neznámého typu preferuj předávání const referencí před předáváním hodnotou.

```
template<typename T>
class Vector {
public:
    void push_back(T d);
    T back() const;
    T operator[](int i) const;
    ...
private:
    ...
};
```



```
template<typename T>
class Vector {
public:
    void push_back(const T& d);
    const T& back() const;
    const T& operator[](int i) const;
    ...
private:
    ...
};
```

# Předpokládej, že je nákladné zkopírovat T

- Řešení: pro objekty neznámého typu preferuj předávání `const` referencí před předáváním hodnotou.

```
template<typename T>
struct Complex {
    T real
    T imag;
    Complex(T r, T i): real(r), imag(i) {}
};
```



```
template<typename T>
struct Complex {
    T real
    T imag;
    Complex(const T& r, const T& i): real(r), imag(i) {}
};
```

# Předpokládej, že je nákladné zkopírovat T

- Řešení: pro objekty neznámého typu preferuj předávání `const` referencí před předáváním hodnotou.

```
template<typename T>  
T minimum(T a, T b) { return (a < b)? a: b; }
```



```
template<typename T>  
const T& minimum(const T& a, const T& b) { return (a < b)? a: b; }
```

# Předpokládej, že je nákladné zkopírovat T

- Řešení: pro objekty neznámého typu preferuj předávání const referencí před předáváním hodnotou.

```
template<typename T>
```

```
T read(std::string prompt) {
```

```
    ...
```

```
}
```



```
template<typename T>
```

```
const T& read(std::string prompt) { // takto ne
```

```
    T result;
```

```
    ...
```

```
    return result; // ouha, vracíme referenci na lokální proměnnou
```

```
    ...
```

```
}
```

POZOR

NEPOKOUŠEJ SE  
PŘEDAT REFERENCI  
TAM, KDE MUSÍŠ  
PŘEDAT HODNOTU

# Předpokládej, že je nákladné zkopírovat T

- Řešení: pro objekty neznámého typu preferuj předávání `const` referencí před předáváním hodnotou.

```
template<typename T>  
T read(std::string prompt) {  
    ...  
}
```



```
template<typename T>  
T read(const std::string& prompt) { // takhle ano  
    ...  
}
```



Do funkce ale vstupuje `std::string`, který je drahé zkopírovat. Použijme referenci.

Takto se alespoň vyhneme drahé kopii známého typu.



# Proč používáme const& a ne jen &?

- Použití konstantní reference má dvě výhody
  - Sémantickou: pokud bereme const T&, slibujeme že nebudeme předaný parametr měnit, ale pouze číst.
  - Syntaktickou: pokud bereme const T&, funkce bere i dočasné hodnoty

```
template <typename T>
void write(T& t) {
    std::cout << t;
}
```

```
int main() {
    int i = 234;
    write(i); 
    write(123); 
}
```

```
template <typename T>
void write(const T& t) {
    std::cout << t;
}
```

```
int main() {
    int i = 234;
    write(i); 
    write(123); 
}
```

Složitější šablony

# Více než jeden parametr

- Šablony tříd i funkcí mohou mít libovolný počet šablonových parametrů.

```
template <typename U, typename V>  
struct Pair {  
    U first;  
    V second;  
}
```

```
Pair<int, std::string> fooPair;  
Pair<bool, bool> barPair;
```

# Proměnný počet parametrů

- Lze deklarovat šablony s proměnným počtem parametrů.
- Příkladem je šablona třídy `std::tuple` a šablona funkce `std::make_tuple`.

```
#include <tuple>
```

```
std::tuple<int, int, int, int> quadTuple;  
std::tuple<int, std::string> pairTuple;  
std::tuple<bool> monoTuple;  
std::tuple<> emptyTuple;
```

```
// ...
```

```
auto myTuple = std::make_tuple(42, '\0', false, "ahoj");  
std::cout << std::get<0>(myTuple) << "\n"; // 42  
std::cout << std::get<3>(myTuple) << "\n"; // ahoj
```

```
// v deklaraci je toto  
template <class... Types>
```

# class namísto typename

- Klíčové slovo `class` má před názvem šablonového parametru stejný význam, jako klíčové slovo `typename`.

```
template <class U, class V>
struct Pair {
    U first;
    V second;
}
```

```
Pair<int, std::string> fooPair;
Pair<bool, bool> barPair;
```

# Jiné parametry než `class/typename`

- Šablonový parametr nemusí být typ, může to být i hodnota. Hodnota, kterou pak dosadíme jako argument, musí být vyčíslitelná během kompilace (protože instanciací šablon probíhá při kompilaci).

```
template<typename T, int N>  
struct Array {  
    T data[N];  
};
```

```
Array<double, 20> fooArr;  
Array<int, 30> barArr;
```

# Metaprogramování

# Metaprogramování

- Šablony umožňují i metaprogramování. To je programování, kde nepracujeme se vstupními daty, ale se vstupním programem (typy).
- Umožňují se chovat k různým typům stejně, ale zároveň optimálně.
- Například `std::advance(iter, sz)` posune iterátor `iter` o `sz` pozic. Každý iterátor můžeme `sz`-krát inkrementovat, ale `RandomAccess` iterátor můžeme posunout o `sz` operátorem `+`.
- Metaprogramování pomocí šablon se řeší v čase kompilace, stejně jako všechny šablony.

# Příklad: Faktoriál

```
template<int n> struct Factorial {  
    enum { val = Factorial<n-1>::val * n };  
};  
template<> struct Factorial<0> {  
    enum { val = 1 };  
};  
  
std::cout << Factorial<12>::val << std::endl;
```

Vytiskne:

479001600

Spočte se v době překladu, ještě před spuštěním programu (rekurzivním sestupem).

# Techniky metaprogramování

- Ukážeme si 2 klasické techniky:
  - Tag dispatch (těžko se překládá)
  - SFINAE ( + `std::enable_if`)
- Tag dispatch je technika, kdy použijeme přetěžování funkcí a jejich volání dle vlastností typů.
  - Obvykle implementováno jako funkce, kterou může uživatel zavolat a sada přetížení, které dělají práci.
  - Například již zmíněný `std::advance(InputIt, Distance)`.
- SFINAE – Substitution Failure Is Not An Error
  - Za určitých okolností kompilační chyby nejsou kompilační chyby... pouze vedou k vyřazení funkce ze sady funkcí, které mohou být použity k uskutečnění volání funkce (tzv. overload set)

# Srovnání „tag dispatch“ a statické funkce

- Předpokládejme, že chceme vytvořit generickou funkci `void f<T>()`,
- která dělá cosi, pokud T je typ splňující predikát POD (`template<class T> struct is_pod<T>; plain old data type`) a něco jiného, jestliže T je typ nesplňující predikát POD (nebo libovolný jiný predikát).
- Můžeme toho dosáhnout pomocí vzoru „tag-dispatch“ podobně jak to dělají standardní knihovny s iterátory (viz dále).
- Alternativní řešení použije statickou členskou funkci a částečně specializované typy (viz dále).

# Řešení podle vzoru „tag dispatch“

```
template <bool> struct podness {};  
typedef podness<true> pod_tag;  
typedef podness<false> non_pod_tag;  
template <typename T> void f2(T, pod_tag) { /* POD */ }  
template <typename T> void f2(T, non_pod_tag) {  
    /* non-POD */  
};  
template <typename T>  
void f(T x) {  
    // Dispatch to f2 based on tag.  
    f2(x, podness<std::is_pod<T>::value>());  
};
```

# Řešení pomocí statických funkcí

```
template <typename T, bool> struct f2;
```

```
template <typename T>  
struct f2<T, true> { static void f(T) { /* POD */ } };
```

```
template <typename T>  
struct f2<T, false> { static void f(T) { /* non-POD */ } };
```

```
template <typename T>  
void f(T x) {  
    // Vyber korektní částečně specializovaný typ.  
    f2<T, std::is_pod<T>::value>::f(x);  
}
```

# tag dispatch -- příklad

```
template <typename Iter, typename Dist>
void td_advance(Iter& it, Dist d) {
    _advance(it, d,
    typename std::iterator_traits<Iter>::iterator_category{});
}
```

Tag



Dispatch



```
template <typename Iter, typename Dist>
void _advance(Iter& it, Dist d,
              std::input_iterator_tag) {
    for (Dist i = 0; i < d; ++i) { ++it; }
}
```

```
template <typename Iter, typename Dist>
void _advance(Iter& it, Dist d,
              std::random_access_iterator_tag)
{ it += d; }
```

# Závislá jména

- Parsování výrazů v C++ je v některých případech velmi složité, například `t * f`; může být buďto deklarace ukazatele, nebo násobení dvou proměnných.
  - Záleží na tom, jestli je `t` typ, nebo proměnná.
- V šablonách může dokonce být parsování závislé na typu dosazeném do šablony (a tudíž se nedá rozhodnout během parsování).
  - `T::x` může být typ, nebo statická proměnná.
- Existuje proto jednoduché pravidlo: „Výraz není typ, pokud tento výraz není již známý jako typ, nebo před ním není klíčové slovo `typename`“

# Závislá jména -- příklad

```
template <typename Container>
void foo(const Container& c) {
    Container d;
    Container::value_type* temp;
    typename Container::value_type* temp;
}
```

Ok, Container je známé jako typ

Násobení!

temp je ukazatel, explicitně říkáme, že Container::value\_type je typ

```
template <typename Iter, typename Dist>
void td_advance(Iter& it, Dist d) {
    _advance(it, d,
    typename td::iterator_traits<Iter>::iterator_category{});
}
```

Proč typename?

SFINAE

# SFINAE – Substitution Failure Is Not An Error

- Za určitých okolností kompilační chyby nejsou kompilační chyby... pouze vedou k vyřazení funkce z overload setu (sady funkcí, které mohou být použity k uskutečnění volání funkce)
- Kompilační chyby nejsou kompilační chyby, pokud se vyskytnou v tzv. závislých částech funkce
  - Argument funkce
  - Další šablonový argument
  - Návratová hodnota
- Za tímto účelem se používá `std::enable_if<Pred, type>::type`, což je typ ze standardní knihovny, který se dá zkompilovat pouze pokud je predikát `Pred` pravdivý.
  - Pokud `Pred` pravdivý není, `::type` neexistuje.
  - Existuje kratší varianta: `std::enable_if_t<Pred, type>`

# SFINAE – příklad

```
template <typename Iter, typename Dist>
void sfinae_advance(Iter& it, Dist d,
typename std::enable_if_t<
    !std::is_same<
        typename std::iterator_traits<Iter>::iterator_category,
        std::random_access_iterator_tag
    >::value
>* = 0) {
    for (Dist i = 0; i < d; ++i) { ++it; }
}
```

```
template <typename Iter, typename Dist>
void sfinae_advance(Iter& it, Dist d,
typename std::enable_if_t<
    std::is_same<
        typename std::iterator_traits<Iter>::iterator_category,
        std::random_access_iterator_tag
    >::value
>* = 0) { it += d; }
```

## std::enable\_if\_t

- `std::enable_if` se též dá zapsat bez specifikace typu
  - `std::enable_if_t<Pred>` je pak bráno jako `void`
  - Bez typu se obvykle používá, pokud je součástí argumentů funkce
- V šablonách se návratový typ též účastní přetěžování
  - Takže se `enable_if_t` dá použít i místo návratového typu

```
template <typename Num1, typename Num2>
std::enable_if_t<std::is_unsigned<Num1>::value
    && std::is_signed<Num2>::value, Num1>
safeCast(Num2 num) {
    if (num > std::numeric_limits<Num1>::max() ||
        num < 0) { throw "num outside range"; }
    return static_cast<Num1>(num);
}
```

# Traits (vlastnosti)

Příklad: rozlišení znakových sad v <string>:

```
template <class charT> struct char_traits;
template <> struct char_traits<char>;
template <> struct char_traits<wchar_t>;
template <> struct char_traits<char16_t>;
template <> struct char_traits<char32_t>;
template < class charT,
        class traits = char_traits<charT>,      //
        basic_string::traits_type
        class Alloc = allocator<charT>         //
        basic_string::allocator_type
> class basic_string;
```

# Vlastnosti typů - `type_traits`

- Standardní knihovna obsahuje tzv „type traits“, což jsou šablonové třídy umožňující zjišťovat a měnit vlastnosti typů.
- Například `std::is_polymorphic<T>::value` je `true` iff `T` obsahuje funkci označenou jako `virtual`.
- `std::make_signed<T>::type` je znamínková varianta typu `T`
  - `std::make_signed<unsigned int>::type` je `signed int`
  - `std::make_signed<char>::type` je `signed char`
  - ...
- Dají se použít pro tag dispatch nebo SFINAE
- Dají se vytvořit i vlastní...

# type\_traits – příklad

```
template <typename Iter, typename Dist>
void _trait_advance(Iter& it, Dist d, std::true_type) {
    it += d;
}
```

```
template <typename Iter, typename Dist>
void _trait_advance(Iter& it, Dist d, std::false_type) {
    for (Dist i = 0; i < d; ++i) {
        ++it;
    }
}
```

Proč není potřeba typename?

```
template <typename Iter, typename Dist>
void trait_advance(Iter& it, Dist d) {
    _trait_advance(it, d, is_random_access<Iter>{});
}
```

## type\_traits – příklad (2)

```
template <typename T, typename Enabled = void>  
struct is_random_access;
```

```
template <typename T>  
struct is_random_access<T,  
    typename std::enable_if_t<  
        std::is_same<typename  
            std::iterator_traits<T>::iterator_category,  
            std::random_access_iterator_tag  
        >::value  
    >  
> : std::true_type {};
```

## type\_traits – příklad (3)

```
template <typename T>
struct is_random_access<T,
    typename std::enable_if_t<
        !std::is_same<typename
            std::iterator_traits<T>::iterator_category,
            std::random_access_iterator_tag
        >::value
    >
> : std::false_type {};
```

## Bonus

- Hra Game of Life v C++ šablonách:

<https://github.com/sirgal/compile-time-game-of-life>

- Turingův stroj v C++ šablonách:

<http://coliru.stacked-crooked.com/a/de06f2f63f905b7e>

Děkuji za pozornost.

# Potíže s funkcí minimum

```
#include <iostream>
```

```
double minimum(double a, double b) { return (a < b)? a: b; }
```

```
int main() {
```

```
    // co je menší, milion nebo miliarda?
```

```
    int i = 1000000;
```

```
    int j = 1000000000;
```

```
    std::cout << minimum(i, j) << "\n"; // odpověď: 1e+06
```

```
    // min(a/c,b/c) == min(a,b)/c. nebo ne?
```

```
    int a = 52;
```

```
    int b = 41;
```

```
    int c = 10;
```

```
    if (4 != 4.1)
```

```
    if (minimum(a / c, b / c) != minimum(a, b) / c) {
```

```
        std::cout << "jejda\n";
```

```
    }
```

```
}
```

# Zbytek safeCast

```
template <typename Num1, typename Num2>
std::enable_if_t<std::is_signed<Num1>::value
                && std::is_signed<Num2>::value,
Num1> safeCast(Num2 num) {
    if (num > std::numeric_limits<Num1>::max()
        || num < std::numeric_limits<Num1>::min()) {
        throw "num is outside the range for casted-to type"; }
    return static_cast<Num1>(num);
}
```

```
template <typename Num1, typename Num2>
std::enable_if_t<std::is_unsigned<Num1>::value
                && std::is_unsigned<Num2>::value,
Num1> safeCast(Num2 num) {
    if (num > std::numeric_limits<Num1>::max()) {
        throw "num is outside the range for casted-to type"; }
    return static_cast<Num1>(num);
}
```

## Zbytek safeCast (2)

```
template <typename Num1, typename Num2>
std::enable_if_t<std::is_unsigned<Num1>::value
                && std::is_signed<Num2>::value,
Num1> safeCast(Num2 num) {
    if (num > std::numeric_limits<Num1>::max() || num < 0) {
        throw "num is outside the range for casted-to type"; }
    return static_cast<Num1>(num);
}
```

```
template <typename Num1, typename Num2>
std::enable_if_t<std::is_signed<Num1>::value
                && std::is_unsigned<Num2>::value,
Num1> safeCast(Num2 num) {
    if (num > std::numeric_limits<Num1>::max()) {
        throw "num is outside the range for casted-to type"; }
    return static_cast<Num1>(num);
}
```