

# PB161 Programování v jazyce C++

## Přednáška 10

### Šablony

Nikola Beneš

23. listopadu 2015

# K zamyšlení

Jaký je rozdíl mezi `new int[10]` a `new int[10]()`?

# Šablony

- co nejmenší duplikace kódu
- stejný (podobný) kód pro různé typy
  - kontejnery
  - algoritmy
- možná řešení:
  - makra
  - `void *`
  - OOP polymorfismus

- co nejmenší duplikace kódu
- stejný (podobný) kód pro různé typy
  - kontejnery
  - algoritmy
- možná řešení:
  - makra
  - `void *`
  - OOP polymorfismus
- řešení v C++: šablony
  - jiný druh polymorfismu
  - generické programování
  - metaprogramování

- „lepší makra“
- generický kód, do nějž se později doplní typy
  - tzv. instanciacce
  - a nejen typy
- instanciacce
  - vytvoření konkrétní entity ze šablony
  - probíhá při kompilaci
  - rozdíl proti generikům v Javě
- šablony mohou využívat funkce i třídy
  - od C++11 i typové aliasy

`template` <seznam parametru>

- jeden nebo více parametrů
  - C++11 variadické parametry (viz speciální přednášku)
- druhy parametrů
  - typ: `typename`
  - hodnota: celočíselný typ (`int`, `bool`, ...), reference, ukazatel, výčtový typ (definovaný pomocí `enum`)
  - šablona
- parametr může mít implicitní hodnotu (pomocí =)
- místo `typename` možno použít `class`

Příklad:

```
template <typename T, int size = 100, typename U = bool>
```

# Šablony funkcí

```
template <typename T>
const T & myMax(const T & x, const T & y) {
    return x < y ? y : x;
}

int main() {
    int a, b;
    std::cin >> a >> b;
    std::cout << myMax<int>(a, b) << std::endl;
    std::cout << myMax(a, b) << std::endl;
    unsigned long c = 17;
    std::cout << myMax(a, c) << std::endl; // CHYBA
    std::cout << myMax<unsigned long>(a,c) << std::endl;
}
```



# Šablony funkcí (pokr.)

- šablonové funkce se mohou přetěžovat

```
template <typename T>
const T & myMax(const T & x, const T & y, const T & z) {
    return myMax( myMax(x, y), z );
}

template <typename T>
const T & myMax( const std::vector<T> & vec ) {
    return *max_element(vec.begin(), vec.end());
}

int main() {
    std::cout << myMax(2.0, 3.14) << std::endl;
    std::cout << myMax(2.0, 3.14, 42.0) << std::endl;
    std::vector<unsigned> v{ 10, 40, 20, 70, 30 };
    std::cout << myMax(v) << std::endl;
}
```

# Šablony funkcí (pokr.)

- funkce může být přetížená šablonovou i nešablonovou verzí
  - nešablonová verze má přednost

```
const char * myMax(const char * x, const char * y) {  
    return strcmp(x,y) < 0 ? y : x;  
}
```

```
int main() {  
    // zavolá se šablonová funkce myMax<int>  
    std::cout << myMax(20, 70) << std::endl;  
    // zavolá se nešablonová funkce myMax  
    std::cout << myMax("ahoj", "hello") << std::endl;  
    // co se zavolá teď?  
    std::cout << myMax<const char*>("ahoj", "hello") << std::endl;  
    // a co teď?  
    std::cout << myMax<>("ahoj", "hello") << std::endl;  
}
```

# Šablony funkcí (pokr.)

- automatická dedukce šablonového typu
  - jen jméno funkce
  - funkce a prázdný seznam šablonových parametrů <>
  - částečná automatická dedukce: vynechání některých parametrů

```
template<typename T, typename U>
T convert(const U & value) {
    T x = static_cast<T>(value);
    return x;
}
```

```
int main() {
    std::cout << convert(3.14) << std::endl; // CHYBA
    std::cout << convert<int, double>(3.14) << std::endl;
    // částečná dedukce:
    std::cout << convert<int>(3.14) << std::endl;
}
```

**Příklad:** Jednoduchý kontejner

```
#include <iterator>
```

```
template <typename T>
```

```
class MyContainer {
```

```
    size_t size;
```

```
    T * array;
```

```
public:
```

```
    // ...
```

```
};
```

```
int main() {
```

```
    MyContainer<double> cont(10);
```

```
    // ...
```

```
}
```

## Šablony tříd (pokr.)

- u šablonových tříd není žádná automatická dedukce
  - důvod pro funkce jako `std::make_pair`

```
template <typename T>
class Foo {
public:
    Foo(const T &);
    // ...
};

template <typename T>
Foo<T> make_foo(const T & val) {
    return Foo<T>(val);
}

int main() {
    auto foo = make_foo(1);
}
```

## Šablony tříd (pokr.)

- uvnitř šablonových tříd mohou být šablonové metody

```
template <typename T>
class Foo {
    T value;
public:
    Foo(const T & val) : value(val) {}
    template <typename U>
    void print(const U & thing) {
        std::cout << value << " : " << thing << std::endl;
    }
};

int main() {
    Foo<double> foo(3.14);
    foo.print("example");
    // 3.14 : example
}
```

## Úplná specializace šablony (pro třídy)

- speciální implementace pro konkrétní šablonové parametry
- uvozená deklarací `template <>`
- za názvem třídy konkrétní parametry v `< >`

```
template <typename T>  
class X { /* ... */ };
```

```
template <>  
class X<int> { /* speciální implementace pro int */ };
```

- kromě tříd se můžou úplně specializovat
  - funkce (spíš nepoužívat)
  - metody šablonových tříd
  - statické atributy šablonových tříd
  - a další (viz [http://en.cppreference.com/w/cpp/language/template\\_specialization](http://en.cppreference.com/w/cpp/language/template_specialization))



# Specializace (pokr.)

- specializace metody šablonové třídy

```
template <typename T, typename U>
class X {
public:
    void foo() { std::cout << "unspecialised!\n"; }
};

template <>
void X<int,double>::foo() { std::cout << "specialised!\n"; }

int main() {
    X<int, int> x;
    X<int, double> y;
    x.foo();
    y.foo();
}
```

## Částečná specializace šablony

- jen pro třídy
- uvozená deklarací `template` <nespecializované parametry>
- nespecializované parametry je pak možno použít v seznamu parametrů za názvem třídy

## Specializace (pokr.)

```
template <typename T, typename U>
class X { /* ... */ };           // 1
template <typename T>
class X<T, double> { /* ... */ }; // 2
template <typename T>
class X<T, T> { /* ... */ };      // 3
template <typename T>
class X<int, T> { /* ... */ };     // 4
```

```
X<double, char> x1; // použije se verze 1
X<char, double> x2; // použije se verze 2
X<char, char> x3;   // použije se verze 3
X<int, char> x4;     // použije se verze 4
X<int, int> x5;      // CHYBA! není jasné, kterou verzi použít
```

# Specializace (pokr.)

- doporučení pro specializace
  - neměnit vnější chování
- příklad specializace ve standardní knihovně
  - `std::vector<bool>`

# Typové aliasy v C++11

- šablonové

```
template<typename Element, typename Allocator>
class BasicContainer { /* ... */ };

template<typename Element>
class StandardAllocator { /* ... */ };

template<typename Element>
using Container = BasicContainer<Element,
    StandardAllocator<Element>>;

int main() {
    Container<double> c;
    // totéž, co BasicContainer<double,
    //      StandardAllocator<double>>
}
```

# Typové aliasy v C++11 (pokr.)

- nešablonové
  - fungují jako `typedef`, jen s jinou syntaxí

```
using VectorInt = std::vector<int>;
```

```
using Number = int;
```

```
using Function = void (*) (int, int);
```

```
template <typename T>  
class Container {  
    using valueType1 = T;  
    typedef T valueType2;  
};
```

# Použití šablon – funkční objekty

- jak fungují algoritmy ve standardní knihovně?
  - funkci/funkční objekt berou jako šablonový argument

```
template<typename Iterator, typename Function>
void modify(Iterator from, Iterator to, Function func) {
    for (auto it = from; it != to; ++it) {
        *it = func(*it);
    }
}
```

# Šablony nejen typové

- parametry šablon mohou být i (některé) hodnoty
  - celočíselných typů (včetně `bool`, `char`)
  - výčtových typů
  - reference, ukazatele
- instance parametrů musí být konstantní výrazy

Příklad použití ze standardní knihovny:

```
template <typename T, std::size_t N>
class array {
    T _array[N];
    // ...
};
```



## Šablony nejen typové (pokr.)

```
// klasický příklad: faktoriál při překladu
template <unsigned N>
struct Factorial {
    const static unsigned value = N * Factorial<N-1>::value;
};

template <>
struct Factorial<0> {
    const static unsigned value = 1;
};

int main() {
    // hodnota se spočítá při překladu, ne za běhu!
    return Factorial<5>::value;
}
```

# Šablony jako parametry šablon

- parametrem šablony může být opět šablona

```
template< typename Value,  
        template <typename> class Container >  
class X {  
    Container<Value> cont;  
    // ...  
};  
  
// použití  
  
X<int, std::vector> x;  
// x obsahuje atribut cont typu std::vector<int>
```

## Kdy se vytvoří konkrétní instance šablony?

- během překladu
- pokud se v kódu objeví konkrétní použití
  - prototyp nebo použití funkce
  - použití třídy
  - explicitní instanciac
- pro instanciaci je třeba vidět celou definici šablonové třídy/funkce
- důsledek
  - buď inteligentní linker
  - nebo šablonové třídy a funkce v hlavičkovém souboru
- realita: šablony musí být v hlavičkovém souboru

# Instanciace (pokr.)

Instance se vytváří jen pro to, co se skutečně použije.

```
template<typename T>
class X {
    T t;
public:
    void f() { t.f(); }
    void g() { t.g(); }
};

class A { public: void f() {} };

int main() {
    X<A> x;
    x.f(); // takhle je to OK
    // x.g(); // po odkomentování se nezkompiluje
}
```

## Nápověda pro překladač – typename

```
template <typename T>
class Container {
public:
    typedef T value_type;
    using ptr_type = T *;
    using size_type = unsigned int;
    size_type size();
    // ...
};

template <typename T>
void do_something(Container<T> & cont) {
    // nebude fungovat:
    Container<T>::size_type size = cont.size();
    // je třeba napsat:
    typename Container<T>::size_type size = cont.size();
}
```

## Šablony

- velmi mocný nástroj
- generické funkce (metody)
- generické třídy
- nejen pro typy

## Instanciace šablon

- při překladu
- jen ty instance, které je nutné vyrobit
- jen ty části, které je nutné vyrobit
- důsledek: šablony v hlavičkových souborech