

PB161 – 10. přednáška (23. listopadu 2015)

Šablony

Motivace

- snaha o co nejmenší duplikaci kódu
- co když máme kód, který chceme použít pro různé typy?
 - generická funkce (maximum, minimum, swap, ...)
 - kontejnery
 - algoritmy
- možné řešení (C): makra
 - žádná typová bezpečnost
 - různé druhy problémů (vzpomeňte si na makra v C)
 - kontejnery: škaredé triky s `#define` a `#include`
- možné řešení (C): používání `void *`
 - používá např. `qsort`
 - není typově bezpečné
 - objekty v kontejnerech: jen jako ukazatele
- možné řešení: OOP polymorfismus
 - vyžaduje pozdní vazbu, což může být pomalé
 - vyžaduje, aby všechny myslitelné typy byly součástí jedné hierarchie (např. `Object` v Javě)
 - není možné použít pro primitivní typy (`int`, `double`)
- řešení v C++: šablony
 - jiný druh polymorfismu
 - generické programování
 - metaprogramování
- standardní knihovna v C++ z velké části používá šablony

Šablony

- můžeme je chápat jako „lepší makra“ (ovšem výrazně lepší)
- umožňují psát kód, v němž jsou „díry“ pro typy, které se doplní později (a mohou se doplnit víckrát různě)
 - tzv. instanciac
 - a nemusí jít jen o typy, jak uvidíme později
- instanciac probíhá při kompilaci
 - narozdíl od generik v Javě
- šablony mohou využívat funkce i třídy
 - od C++11 i typové aliasy pomocí `using` (viz dále)
 - (od C++14 i proměnné)

Syntax

- syntax šablonové deklarace: `template <seznam parametru>`
- parametr může být jeden nebo více
 - C++11 variadické parametry (viz speciální přednášku)

- šablonové parametry: `druh jmeno` (jméno je nepovinné)
- druh může být
 - typ: `typename`
 - hodnota: celočíselný typ (`int`, `bool`, ...), reference, ukazatel, výčtový typ (definovaný pomocí `enum`)
 - šablona
- parametr může mít implicitní hodnotu (pomocí `=`)
- místo `typename` je možno použít i slovo `class`
 - má úplně stejný význam (tj. označuje libovolný typ, ne jen třídu)
 - proto je možná lepší používat spíš `typename`

Příklad:

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

- za deklarací šablony následuje entita, která je danou šablonou parametrizována
 - typicky funkce nebo třída

Šablony funkcí

- cokoli, co bude řečeno o funkcích, platí stejně i pro metody
- definice šablonové funkce

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

- použití šablonové funkce

```
#include <iostream>
```

```
int main() {
    int a, b;
    std::cin >> a >> b;
    // explicitní hodnota šablonového parametru
    std::cout << myMax<int>(a, b) << std::endl;
    // překladač může sám dedukovat
    std::cout << myMax(a, b) << std::endl;
    // ovšem typy musí sedět přesně
    unsigned long c = 17;
    std::cout << myMax(a, c) << std::endl; // CHYBA
    // tohle je OK:
    std::cout << myMax<unsigned long>(a,c) << std::endl;
}
```

- šablonové funkce se mohou přetěžovat

```

#include <vector>
#include <algorithm>

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;
}

```

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

```

#include <cstring>

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;
}

```

- šablonové funkce se sice mohou i specializovat (viz specializaci u šablonových tříd), ale nedoporučuje se to používat
 - neintuitivní chování při přetěžování
 - viz <http://www.gotw.ca/publications/mill17.htm>
- syntax pro automatickou dedukci šablonového typu
 - jen jméno funkce: umožňuje přetížení nešablonovou verzí

- funkce a prázdný seznam šablonových parametrů <>: neumožňuje přetížení nešablonovou verzí
- 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() {
    // tohle nebude fungovat, není jak dedukovat oba parametry:
    std::cout << convert(3.14) << std::endl; // CHYBA
    // tohle je OK:
    std::cout << convert<int, double>(3.14) << std::endl;
    // částečná dedukce:
    std::cout << convert<int>(3.14) << std::endl;
}
```

Šablony tříd

- definice šablonové třídy

Příklad: Jednoduchý kontejner

```
#include <iterator>

template <typename T>
class MyContainer {
    size_t size;
    T * array;
public:
    MyContainer(size_t n) : size(n), array(new T[n]()) {}
    ~MyContainer() { delete array; }
    MyContainer(const MyContainer & other)
        : size(other.size), array(new T[size]) {
        std::copy(other.array, other.array + size, array);
    }
    void swap(MyContainer & other) {
        using std::swap;
        swap(size, other.size);
        swap(array, other.array);
    }
    MyContainer & operator=(MyContainer other) {
        swap(other);
        return *this;
    }
}
```

```

    }
    void output() const {
        std::copy(array, array+size,
            std::ostream_iterator<T>(std::cout, " "));
    }
};

```

- použití šablonové třídy

```

int main() {
    MyContainer<double> cont(10);
    cont.output();
    std::cout << std::endl;
}

```

- odbočka k zamyšlení: jaký je rozdíl mezi `new int[10]` a `new int[10]()`?
- u šablonových tříd není žádná automatická dedukce
 - to je 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);
}

```

- 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
}

```

Specializace

- můžeme chtít speciální implementaci pro konkrétní šablonové parametry
 - jiný způsob zacházení s daty
 - efektivnější implementace
- funkce můžeme přetěžovat, ale u tříd nemáme jak
- proto je možné šablony specializovat
- úplná specializace šablony (pro třídy)
 - 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)

```

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 <nespecializovane parametry>`
 - nespecializované parametry je pak možno použít v seznamu parametrů za názvem třídy

```
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
```

- doporučení pro specializace
 - specializace by se navenek neměly chovat jinak než v obecném případě
- příklad specializace ve standardní knihovně
 - `std::vector<bool>`
 - trochu kontroverzní

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>>
}
```

- 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 nejen typy, ale i (některé) hodnoty
 - celočíselných typů (včetně **bool**, **char**)
 - výčtových typů
 - reference, ukazatele
 - šablony
- instance parametrů musí být konstantní výrazy (jejich hodnota musí být známa při překladu)

Příklad použití ze standardní knihovny:


```

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

// 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>

```

Instanciac

- šablona samotná nedeklaruje žádnou funkci nebo třídu
- ty vznikají až instanciací
- instanciac (vytvoření konkrétní instance)
 - jakmile se v kódu objeví konkrétní použití šablony
 - prototyp nebo použití funkce
 - použití třídy
 - explicitní instanciac (`class Container<int>;`)

- pro instanciaci je třeba, aby kompilátor viděl celou definici šablonové třídy
- důsledek
 - linker musí vědět o šablonách a musí je umět instanciovat podle potřeby
 - nebo musí být šablonové třídy a funkce v hlavičkovém souboru
- realita: šablony musí být v hlavičkovém souboru
- 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 nezkompile
}
```

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();
}
```