

Rychle a svižně

C++11 STANDARD

Osnova

- ⦿ Podpora napříč překladači
- ⦿ Klíčová slova
- ⦿ Konstrukce jazyka
- ⦿ STL Knihovna
- ⦿ Bonus

- ⦿ značení
 - detail – technický detail, pro zajímavost
 - navíc – nebudeme probírat dopodrobna

Motivace

- ⦿ Proč rozšiřovat C++?
 - nové postupy
 - chybějící funkcionalita v jazyku
 - snaha aplikovat „trendy“ věci
 - jazyky jako C# a Java je již implementují
- ⦿ Z toho plyne rozšiřování
 - jazyka
 - standardní knihovny

Překladače

⦿ GCC

- verze 4.7.2 nebo 4.7.3 – téměř celý standard
- verze 4.8.1 vše (pouze ohlášeno)

⦿ clang

- verze 3.2 nebo 3.3 téměř vše

⦿ MSVC

- verze 2012 – nedostatky v jazyku
- verze 2013 – téměř vše



Klíčová slova

- ⦿ nově přidaná (`decltype`, `constexpr`, ...)
- ⦿ změna významu (`auto`)
- ⦿ rozšíření významu (`using`, `delete`, ...)
- ⦿ nově přidaná v kontextu (`final`, `override`)

Klíčová slova – auto

⦿ problém 1:

- použijete ve své třídě `std::vector`
- ten je parametrizován šablonovým typem

⦿ řešení v C++03 (používané v PB161)

```
typename std::vector< Type >::iterator it;
```

- nepřehledné, složité
- nutnost přepsat typ `Type` při změně typu

Klíčová slova – auto

⦿ problém 2

- máte šablonovanou funkci
- nad šablonovým parametrem voláte metodu
- neznáte datový typ výsledku volání
 - může se netriviálně měnit

⦿ řešení v C++03

- použít typedef ze šablonové třídy

```
typename Type::return_t ret = t.call();
```

- nepřehledné, složité

Klíčová slova – auto

- označuje odvoditelný typ proměnných
- používat s rozvahou

```
double foo() {  
    return 3.14;  
}  
int main() {  
    auto a = 0; // int  
    std::vector< long > v;  
    auto i = v.begin(); // std::vector< long >::iterator  
    auto pi = foo(); // double  
    return 0;  
}
```


Klíčová slova – char*_t

- nové typy
- char16_t pro UTF-16 znaky
- char32_t pro UTF-32 znaky

Klíčová slova – constexpr



- značí, že výraz je konstantní a lze vypočítat v době překladu
- lze použít na proměnné, funkce a metody

```
constexpr int factorial( int f ) {  
    return f <= 1 ? 1 : f * factorial( f-1 );  
}  
template< int N >  
void testPrint() {  
    std::cout << N << std::endl;  
}  
  
testPrint< factorial( 4 ) >(); // 4! == 24
```

Klíčová slova – decltype

- ◉ `decltype(výraz)`
- ◉ stává se typem, který vznikne vyhodnocením výrazu
- ◉ použití v situacích, kdy není možné zjistit typ výrazu
- ◉ vysvětlení použití později

```
double pi = 3.14;  
decltype( pi ) alsoDouble = 4;  
decltype( 0 ) IamInt = 10;
```

Klíčová slova – default, delete



- ⦿ lze poznačit konstruktory a přiřazovací operátor
- ⦿ default vynutí vygenerování překladačem
 - pozor na reference
- ⦿ delete zakáže volání
 - v C++03 řešeno umístěním do private sekce
 - neintuitivní

```
class MyClass {  
public:  
    MyClass() = default;  
    MyClass( const MyClass &) = delete;  
    MyClass operator=(MyClass) = delete;  
    ...  
};
```

Klíčová slova – final, override

⦿ problém

- v C++03 není možné ověřit, zda překrýváme virtuální metodu, nebo ne
 - klasifikátor `const`, `volatile`
 - přepsání v písmenu
 - viditelnost
- přitom snadné odhalit překladačem
 - Java i C# kontrolu mají

Klíčová slova – final, override

- ⦿ klíčová slova pouze na správném místě
 - závisí na kontextu použití
- ⦿ takto poznačené virtuální metody překladač zkontroluje
 - metoda poznačená jako `final` nesmí být překryta
 - metoda poznačená jako `override` musí překrývat metodu z děděné třídy

```
struct A { virtual void foo() {} };  
struct B : A { virtual void foo() override {} };  
struct C : B { virtual void foo() final override {} };  
//struct D : C { virtual void foo() {} }; ← error
```

Klíčová slova – noexcept

- náhrada za throw v hlavičkách funkcí/metod
- noexcept jako operátor
 - Detekuje, zda je výraz označen jako noexcept
- noexcept jako specifikátor
 - Označuje funkce/metody a pomáhá programátorovi určit, zda může očekávat výjimku, nebo ne

```
void foo( const T &t ) noexcept( noexcept( t = t ) )  
{  
    t = t;  
}
```

specifikátor

operátor

Klíčová slova – nullptr

- ⦿ nulový pointer, náhrada za NULL
- ⦿ typově bezpečný
 - NULL je typu `int`
 - problém při přetěžování, aplikaci šablonových parametrů
 - `nullptr` má nedefinovaný typ (záleží na překladači), ovšem je plně konvertibilní pouze na ukazatele
- ⦿ NULL nebylo odstraněno z důvodu zpětné kompatibility

Klíčová slova – static_assert

- ⦿ `static_assert(výraz, chybová hláška);`
- ⦿ při překladu dojde k vyhodnocení výrazu
- ⦿ pokud se hodnota výrazu rovná `false`, dojde k zobrazení chybové hlášky a ukončení kompilace
- ⦿ využití
 - kontrola šablonových parametrů
 - kontrola `constexpr` výrazů

```
static_assert( sizeof( bool ) > 1, "bool is bigger than one byte" );
```

Klíčová slova – using



- ⦿ rozšíření významu
 - umožní definovat vlastní typy
- ⦿ podobné jako typedef, ale lepší
 - umí pracovat se šablonami
 - v C++03 bylo nutné udělat wrapper

```
using uint = unsigned int;  
  
template< typename T >  
using Matrix = std::vector< std::vector< T > >;  
  
Matrix< int > matrix;
```

Konstrukce jazyka

- ⦿ R-value reference
- ⦿ variadické šablony
- ⦿ range-for cyklus
- ⦿ inicializační seznam
- ⦿ nový zápis funkce
 - podmíněná definice funkce
- ⦿ lambda

Konstrukce jazyka – R-value reference – motivace (+ opakování)

```
std::string foo() { return "mooooc dlouuuuuhyyyy teeeeext"; }  
std::string text = foo() + foo();
```

1. dojde k vytvoření dočasného řetězce (2x)
2. dojde ke spojení dočasných řetězců
3. zavolá se kopírovací konstruktor
4. řetězec se zkopíruje
5. dočasný řetězec se zahodí

Konstrukce jazyka – R-value reference – připomenutí

⦿ L-hodnota

- „to, co může být na levé straně přiřazení“
 - proměnná, reference, bitfield, dereferencovaný ukazatel

⦿ R-hodnota

- „to ostatní“
 - dočasné objekty, čísla, řetězce, ...

Konstrukce jazyka – R-value reference

- ⊙ výkonnostní optimalizace
 - reference na dočasný objekt
 - dočasný objekt → končí platnost → lze vykrást
- ⊙ přímo se neseťkáte často
 - vnitřně používají STL kontejnery
 - move konstruktor
 - metody požadující dočasné hodnoty
 - funkce `std::move`

Konstrukce jazyka – R-value reference – příklad použití

```
std::vector< std::string > texts;
std::string t( "dloooooouuuuhyyyyyy teeeext" );

// nevolá se kopirovací konstruktor
// přesunutí textu může mít výrazný vliv
// na rychlost aplikace
texts.push_back( std::move( t ) );
// t je v tomto okamžiku prázdné
// a již by se nemelo používat

std::string tt( "dloooooouuuuhyyyyyy teeeext" );

std::string o1 = tt; // kopirovací konstruktor
std::string o2 = std::move( tt ); // přesouvací konstruktor

// funkce s parametrem vynucujícím přesunutí
void foo( std::string &&text ) { ... }
foo( std::move( o1 ) );
//foo( o2 ); ← chyba
```



R-value reference se zapisuje jako &&

Konstrukce jazyka – variadické šablony



- ⊙ problém
 - chci volat funkci/metodu s neomezeně mnoho parametry
 - printf
- ⊙ řešení v C++03:
 - není
 - nutnost použití extern “C”
 - problém s typovou kontrolou
- ⊙ řešení v C++11:
 - variadické šablony

Konstrukce jazyka – variadické šablony

Navíc



```
template< typename T >
class Adapter {
    int _id;
    T _obj;
public:
    template< typename... Args >
    Adapter( Args... args ) :
        _obj( std::forward< Args >( args )... )
    {}
};
```

nevieme čo je tu za zdrojový kód

Jak zaručit předání všech parametrů?

Vzorové přeposlání všech parametrů do konstrukturu objektu typu T.

Konstrukce jazyka – variadické šablony



- ⦿ ve standardní knihovně mnoho využití
 - `std::make_shared`
 - `std::thread`
 - `std::make_tuple`
 - vytvoří „anonymní strukturu“
- ⦿ VS2012 řešení
 - šablonové parametry až do cca 20 parametrů

Konstrukce jazyka – range-for cyklus

⦿ Proč další druh cyklu?

- často se prochází datové kontejnery
- iterování přes index nelze použít u všech (a navíc je pomalejší)
- procházení pomocí iterátorů má krkolomný zápis
- `std::for_each` nemá nic jako příkaz `break` v cyklech
 - řešit pomocí vyhazování výjimky je škaredé

Konstrukcje języka – range-for cyklus

```
std::vector< std::string > names;  
...  
for ( const std::string &name : names ) {  
    ...  
}
```

Diagram illustrating the range-for loop structure:

- Prvek z kontejneru** (Element from container) points to the element being iterated over.
- Kontejner** (Container) points to the container being iterated over.

```
for ( int i : { 1, 2, 3, 4, 5, 6 } ) {  
    ...  
}
```

Diagram illustrating the range-for loop structure:

for (*expr* : *container*) *body*

↓

```
{  
    auto &&__c = container;  
    auto __i = beginExpr,  
           __e = endExpr;  
    for ( ; __i != __e; ++__i ) {  
        expr = *__i;  
        body  
    }  
}
```

Konstrukce jazyka – range-for cyklus

- ⦿ chci použít i pro vlastní třídu
 - metoda `begin`
 - metoda `end`
 - implementace iterátoru

Konstrukce jazyka – inicializační seznam



- ⦿ možnost inicializovat třídu pomocí více prvků
 - typicky u kontejnerů z STL

```
std::vector< int > primes{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 };
```

- ⦿ v souvislosti s tím došlo k možnosti použít syntaxi složených závorek i k volání běžných konstruktorů
 - lze tak explicitně volat bezparametrické konstruktory elementárních typů
 - int, char, double, ...

Konstrukce jazyka – inicializační seznam



⦿ realizace

- `std::initializer_list< type >`
 - chová se jako datový kontejner
 - metody `begin`, `end`, `size`

⦿ chci použít ve vlastní třídě

- implementovat konstruktor
- implementovat přiřazovací operátor
- brát hodnotou
 - nemusí být datovým kontejnerem v paměti

Konstrukce jazyka – inicializační seznam



```
class MyClass {
    std::vector< int > d[2];
public:
    MyClass( std::initializer_list< int > list )
    {
        int index = 0;
        for ( int i : list ) {
            d[index].push_back( i );
            index = (index + 1) % 2;
        }
    }
};

MyClass c{ 1, 2, 3, 4, 5, 6 };
```


Konstrukce jazyka – nový zápis funkce

```
return_type name( parameters ) { body }
```

```
auto name( parameters ) -> return_type { body }
```

- ⦿ výhody?
- ⦿ viditelnost jmen parametrů při definici návratové hodnoty
- ⦿ lepší přehlednost
 - komplikovaný typ za šipku
 - snadný standardně

```
template< typename T, typename U >  
auto foo( T &t, U &u )  
    -> decltype( t + u )  
{  
    t.Prepare();  
    u.Prepare();  
    return t + u;  
}
```

Konstrukce jazyka – podmíněná definice funkce

```
void foo( unsigned u ) {  
    std::cout << "uint: " << u;  
}  
void foo( double d ) {  
    std::cout << "double: " << d;  
}
```

```
int main() {  
    foo( 1u );  
    foo( 3.14 );  
    foo( 1 );//problém  
    // 1 je typu int, nelze se rozhodnout, kterou funkci zavolat  
  
    return 0;  
}
```

Problém vznikne vždy, pokud neexistuje přesná typová shoda v parametrech a existuje víc funkcí s jedním přetypováním parametru.

Konstrukce jazyka – podmíněná definice funkce

```
template< typename T >
typename std::enable_if<
    std::is_floating_point< T >::value,
    void
>::type
foo( T f ) {
    std::cout << "floating point: " << f;
}

void foo( unsigned i ) {
    std::cout << "otherwise: " << i;
}

int main() {

    foo( 1u );
    foo( 3.14 );
    foo( 1 );

    return 0;
}
```

Podmínka, která musí platit

Typ, který se použije jako
návratová hodnota, pokud platí

Nepřehledné při zápisu před
jménem funkce.

Konstrukce jazyka – nový zápis funkce

```
template< typename T >
auto foo( T d )
    -> typename std::enable_if<
        std::is_floating_point< T >::value,
        void
    >::type
{
    std::cout << "floating point: " << d;
}

void foo( unsigned i ) {
    std::cout << "otherwise: " << i;
}

int main() {

    foo( 1u );
    foo( 3.14 );
    foo( 1 );
    return 0;
}
```

Přepsání do varianty s
novým způsobem zápisu
funkce.

Konstrukce jazyka – lambda

⊙ situace

- používáme algoritmy z STL
- mnohé z nich potřebují funktor jako parametr
- funktor = vytvořit třídu

⊙ problém

- kostrbatá syntaxe
- nutnost vymýšlet názvy
- víc funktorů → ztráta přehlednosti kódu

Konstrukce jazyka – lambda

- ⊙ známé z funkcionálního paradigmatu
 - anonymní funkce
 - má schopnost zachytávat svůj kontext
 - může vidět na proměnné okolo sebe
 - capture sekce
- ⊙ syntaxe:

```
[ capture ] ( parameters ) -> return_t { body }
```
- ⊙ některé části definice mohou být vynechány

Konstrukce jazyka – lambda

◎ capture

- možnost zachycení proměnných okolo lambdy
- [] – nezachytává kontext (normální funkce)
- [=] – zachytí vše hodnotou (konstantní)
- [&] – zachytí vše referencí (teoreticky)
- [a,&b] – zachycení a hodnotou, b referencí

```
std::vector< int > v{ 1, 2, 3, 4, 5, 6, 42 };
int numberOfEven{}; // inicializace defaultním ctorem, = 0
std::for_each( v.begin(), v.end(),
    [&] ( int n ) {
        if ( n % 2 == 0 ) ++numberOfEven;
    }
);
```

Konstrukce jazyka – lambda

⊙ jaký má typ?

- pokud nechytá kontext, je typem funkce
- pokud zachytává kontext, není možné vědět typ
 - třída generovaná překladačem

⊙ jak lze uložit?

- do proměnné typu `auto`
- do šablonové proměnné
- do proměnné typu `std::function<signature>`

Konstrukce jazyka – lambda

```
#include <functional>
#include <iostream>

template< typename C, typename A >
void foo( C callback, A argument ) {
    callback( argument );
}

void bar( std::function< void(int) > c
    callback( argument );
}

int main() {
    int context = 0;
    auto lambda1 = [&] ( int n ) -> int {
        ++context;
        return n*n;
    }
    std::function< void(int) > lambda2 = [&] ( int n ) { context += n; }

    lambda1(42);
    lambda2(42);
    foo( lambda1, 5 );
    foo( [] ( int n ) { std::cout << n << std::endl; }, 2 );
    bar( [] ( int n ) { std::cout << n << std::endl; }, 2 );
    return 0;
}
```

Předání čehokoliv volatelného do funkce pomocí šablonového parametru.

Předání čehokoliv volatelného do funkce pomocí std::function s požadovanou signaturou.

Uložení do promenne

Za Zavolání s lambdou definovanou v místě volání.

STL Knihovna

- ⦿ statické pole
- ⦿ automatické pointry
- ⦿ vlákna
 - mutex
- ⦿ regulární výrazy
- ⦿ „anonymní struktury“
- ⦿ (a mnohé další)

STL Knihovna – statické pole

⊙ problém

- chci zvětšovací pole
 - použiji `std::vector`
- chci nezvětšovací statické pole
 - musím použít pole z jazyka C

⊙ řešení C++11

- `std::array< typ, velikost >`
 - chová se jako pole
 - má metody jako kontejner
 - `begin`, `end`, `size`, `operator[]`

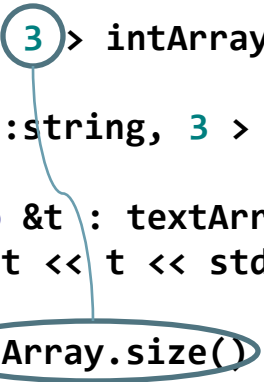
STL Knihovna – statické pole

```
#include <array>
#include <iostream>
#include <string>

int main() {
    std::array< int, 3 > intArray{ { 1,2,3 } }; // double-braces required
    // except after =
    std::array< std::string, 3 > textArray = { "text", "texz2", "text3" };

    for ( const auto &t : textArray )
        std::cout << t << std::endl;

    std::cout << intArray.size() << std::endl;
    return 0;
}
```

A blue line connects the number 3 in the template argument of the first array declaration to the argument of the size() method call in the output statement.

STL Knihovna – automatické pointry

⦿ problém

- v C a C++ je potřeba alokovanou paměť uvolňovat
- je potřeba podchytit všechna možné situace, kdy je potřeba dealokovat

⦿ řešení C++

- princip RAII
 - dynamicky alokované zdroje spravuje třída
 - v destruktoru třída dealokuje zdroje

STL Knihovna – automatické pointry

- ⦿ `std::unique_ptr< typ >`
 - symbolizuje unikátnost dynamické paměti
 - lze pouze přesouvat
 - `std::move`
- ⦿ `std::shared_ptr< typ >`
 - symbolizuje vlastnictví dynamické paměti
 - počítá si reference, poslední dealokuje
- ⦿ `std::weak_ptr< typ >`
 - symbolizuje běžný ukazatel
 - před použitím nutné konvertovat na `std::shared_ptr`

STL Knihovna – automatické pointry


```
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void f() {
    // Has to be copied into a shared_ptr before usage
    if (auto spt = gw.lock())
        std::cout << *spt << "\n";
    else
        std::cout << "gw is expired\n";
}

int main() {
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;
        f(); // 42
    }

    f(); // gw is expired
}
```



Variadická šablona funkce.
Bere přesně takové argumenty,
jaké potřebuje konstruktor alokovaného
objektu.

STL Knihovna – vlákna

- ⦿ C a C++03 neznalo nic jako pojem vlákno
- ⦿ v C++11 je pojem vlákno zaveden
 - C# a Java ho mají zaveden již dlouho
 - operační systémy umožňují pracovat s vlákny
 - problém s rozdílným API
 - pthread API nefunguje všude
 - standardizováno rozhraní

STL Knihovna – vlákna

```
#include <iostream>
#include <thread>
#include <vector>

void foo( int i ) {
    std::cout << "Hello world from thread " << i << std::endl;
}

int main() {
    std::vector< std::thread > threads( 12 );
    int i{};
    for ( auto &t : threads ) {
        t = std::thread( foo, i++ );
    }
    for ( auto &t : threads ) {
        t.join();// wait for thread t
    }
}
```



Variadická šablona konstruktoru.
Bere přesně takové argumenty,
jaké potřebuje operátor volání
spouštěného objektu/funkce.

STL Knihovna – synchronizační primitiva

⦿ problém

- výpis nebude atomický
 - může se stát (a stávat se bude), že budou proložené elementární jednotky výstupu mezi vlákny

⦿ řešení

- použít kritickou sekci
 - část kódu, kde je zaručené, že bude provádět pouze jedno vlákno současně

STL Knihovna – synchronizační primitiva

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
```

```
std::mutex mutex;
```

RAII princip

Zamknutí zámku.

```
void foo( int i ) {
    std::lock_guard< std::mutex > lock( mutex );
    std::cout << "Hello world from thread " << i << std::endl;
}
```

Odemknutí v destruktoru objektu lock.

```
int main() {
    std::vector< std::thread > threads( 12 );
    int i{};
    for ( auto &t : threads ) {
        t = std::thread( foo, i++ );
    }
    for ( auto &t : threads )
        t.join();
    return 0;
}
```

STL Knihovna – synchronizační primitiva

```
#include <iostream>
#include <thread>
#include <vector>
#include <mutex>
```

Přepsání funkce foo do lambdy a odstranění globální proměnné.

```
int main() {
    std::vector< std::thread > threads( 12 );
    std::mutex mutex;
    int i{};
    for ( auto &t : threads ) {
        t = std::thread( [&]( int i ) {
            std::lock_guard< std::mutex > lock( mutex );
            std::cout << "Hello world from " << i << std::endl;
        }, i++ );
    }
    for ( auto &t : threads )
        t.join();
    return 0;
}
```

STL Knihovna – regulární výrazy

- ⊙ známé z dalších jazyků
 - C#, Java, Perl, PHP, Javascript, ...
- ⊙ problém
 - validace emailové adresy
- ⊙ C++03
 - nutné volat systémové knihovny
- ⊙ C++11
 - součástí STL knihovny
 - pozor, nefunguje v libstdc++ (gcc)
 - předpokládá se, že bude korektně implementováno s verzí GCC 4.9.0

STL Knihovna – regulární výrazy

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::string filenames[] = {"foo.txt", "bar.txt", "baz.dat", "zoidberg"};
    std::regex txt_regex("[a-z+)]\\.txt");
    std::smatch match;

    for (const auto &name : filenames) {
        if (std::regex_match(name, match, txt_regex)) {
            // The first sub_match is the whole string; the next
            // sub_match is the first parenthesized expression.
            if (base_match.size() == 2) {
                std::ssub_match base_sub_match = base_match[1];
                std::string base = base_sub_match.str();
                std::cout << name << " has a base of " << base << std::endl;
            }
        }
    }
}
```

STL Knihovna – „anonymní struktury“

⊙ problém

- potřeba vrátit z funkce víc hodnot

⊙ C++03

- pro dvě hodnoty lze použít `std::pair< T1, T2 >`
- víc hodnot potřeba vytvořit třídu/strukturu

⊙ C++11

- `std::tuple`

STL Knihovna – „anonymní struktury“

```
#include <iostream>
#include <string>
#include <tuple>
```

Variadická šablona třídy.

```
std::tuple< int, std::string, double > foo() {
    return std::make_tuple( 0, std::string( "text" ), 3.14 );
}
```

```
int main() {
    auto result = foo();
    std::get< 1 >( result ) += "ik";
    std::cout << std::get< 0 >( result ) << std::endl; // 0
    std::cout << std::get< 1 >( result ) << std::endl; // textik
    std::cout << std::get< 2 >( result ) << std::endl; // 3.14

    std::string strValue;
    double dValue;

    std::tie( std::ignore, strValue, dValue ) = foo();
    std::cout << strValue << " " << dValue << std::endl;
    // text 3.14
}
```

Variadické šablony funkcí

Rule of three (+half)

- ⦿ Pokud chcete implementovat jednu z těchto metod, pravděpodobně chcete implementovat všechny
 - kopírovací konstruktor
 - přiřazovací operátor
 - destruktork
 - (swap metodu + přetížení `std::swap`)
 - prohození obsahu 2 objektů stejného typu

Rule of four (+half)

⦿ V C++11 je vhodné rozšířit seznam na

- kopírovací konstruktor
- move konstruktor

```
MyClass( MyClass &&m );
```

- přiřazovací operátor
- destruktork
- (swap metodu + přetížení std::swap)

Copy & swap idiom

Implementace operátoru přiřazení

- Problém!
 - je nutné zaručit, že objekty budou v konzistentních stavech
- Řešení (rule of three / four)

```
MyClass &operator=( MyClass m ) {  
    swap( m );  
    return *this;  
}
```



Parametr brát hodnotou

- swap nesmí vyhodit výjimku
- elegantní a efektivní řešení
 - stálý objekt se zkopíruje (zkopíroval by se stejně)
 - dočasný objekt se přesune (nedojde ke zpomalení)

Závěr

- ⊙ Přehled klíčových slov
- ⊙ Přehled jazykových konstrukcí
 - range-for cyklus
 - lambda
- ⊙ Přehled rozšíření knihovny
 - automatické ukazatele
 - vlákna
- ⊙ Rada pro implementaci operátoru
 - (to byl bonus)