

VYBRANÉ FUNKCIONALITY STANDARDU C++11

Studijní materiály

OBSAH

1	ÚVOD	3
2	POPIS VYBRANÝCH ZMĚN	4
2.1	INICIALIZACE	4
2.1.1	Uniformní inicializace	4
2.1.2	Seznam inicializátorů	5
2.2	LITERÁLY	6
2.2.1	Řetězcové literály (raw string literal)	6
2.2.2	Uživatelsky nadefinované literály	6
2.3	NOVÁ KLÍČOVÁ SLOVA	7
2.3.1	override – překrytí	7
2.3.2	final – omezení překrytí a dědění	7
2.3.3	constexpr – konstantní výrazy	8
2.3.4	noexcept – vyhazování výjimek	8
2.4	NULLPTR	9
2.5	SILNĚ TYPOVÉ ENUMERACE	9
2.6	INLINE NAMESPACE	10
2.7	AUTO A DECLTYPE	11
2.7.1	Range-for	11
2.7.2	auto	11
2.7.3	decltype	12
2.8	VARIADICKÉ ŠABLONY	12
2.9	LAMBDA	13
2.10	REFERENCE NA R-HODNOTU, MOVE SEMANTICS A PERFECT FORWARDING	14
2.10.1	Reference na r-hodnotu (rvalue)	14
2.10.2	Move semantics, std::move	14
2.10.3	Perfect forwarding, std::forward	16
2.11	VLÁKNA	17
2.11.1	Funkce join() a detach()	17
2.11.2	mutex	18
3	ZADÁNÍ SAMOSTATNÉ PRÁCE	20
3.1	ZADÁNÍ 1	20
3.2	ZADÁNÍ 2	20

1 ÚVOD

Standard C++11 započíná éru moderních C++ standardů. Byl navržen tak, aby byl kompatibilní s C++98 a C++03. Výjimku tvoří nová klíčová slova, jejichž název nesmí být jméno proměnné. Nemůžeme si proto nadefinovat například proměnnou `int noexcept`; jelikož `noexcept` je klíčové slovo.

2 POPIS VYBRANÝCH ZMĚN

2.1 Inicializace

2.1.1 Uniformní inicializace

Doposud v C++ chyběla jednotná inicializace proměnných/objektů. Inicializace se mohla v závislosti na inicializovaném objektu provést pomocí kulatých závorek, složených závorek nebo operátorů přiřazení. C++11 přidává *uniform initialization*, což znamená, že proměnnou nebo objekt můžeme inicializovat stejným způsobem pomocí složených závorek. Tento zápis zároveň provede *value initialization*, takže proměnná nikdy nebude ve stavu *undefined* – bude 0 pro datové typy a *nullptr* pro ukazatele.

```
int i{};           // bude inicializováno jako 0
```

```
int values[] { 1, 2, 3 };
```

```
std::vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
```

Nemůžeme ovšem používat *narrowing initialization* jako u inicializace pomocí operátorů, která „ořeže“ hodnotu, pokud se do proměnné nevejde.

```
int x = 5.3;       // x bude 5
```

```
int y{5.3};        // ERROR
```

2.1.2 Seznam inicializátorů

V C++11 můžeme předat funkci seznam parametrů pomocí `std::initializer_list<>`. To se hodí např. pokud chceme nad každým prvek vykonat nějakou funkci, nebo při inicializaci prvku.

Funkce se seznamem inicializátorů:

```
void foo (std::initializer_list<int> values)
{
    for (auto v : values) { std::cout << v << "\n"; }
}
```

Volání funkce se seznamem inicializátorů:

```
foo ({12,3,5,7,11,13,17});
```

Pokud v třídě máme více konstruktorů a některý z nich využívá seznam inicializátorů, překladač bude preferovat konstruktor se seznamem. Při volání konstruktoru musíme ovšem využít zápisu se složenými závorkami.

```
class Cislá
{
public:
    Cislá(int,int){return;};
    Cislá(std::initializer_list<int>){return;};
};

Cislá a(1,2);           // zavolá Cislá(int,int)
Cislá b({1,2});         // zavolá Cislá(initializer_list)
Cislá c{1,2};           // zavolá Cislá(initializer_list)
Cislá d{1,2,3};         // zavolá Cislá(initializer_list)
Cislá e = {1,2};        // zavolá Cislá(initializer_list)
```

2.2 Literály

2.2.1 Řetězcové literály (raw string literal)

V C++11 je nově možno používat řetězce, které nezpracovávají speciální znaky jako `\n`, `\r` atp. Pokud bychom chtěli mít řetězec `"\\n"`, běžný řetězec by vypadal takto:

```
"\\\\n"
```

Nově však můžeme řetězec obalit zezáčátku `R“(` a na konci `)“` a ve výsledku zapsat takto:

```
R"(\n)"
```

Pokud chceme mít v řetězci závorky, můžeme k obalovacím závkám přidat nějaké značení, tzv. *delimiter*. Může být až 16 znaků dlouhý a můžeme v něm použít cokoliv kromě zpětného lomítka, mezer a uvozovek:

```
R"delim(...)delim"
```

2.2.2 Uživatelsky nadefinované literály

Nově je možné nadefinovat si vlastní literály. Díky tomu můžeme za hodnotu napsat příponu označující, co hodnota znamená – nabízejí se např. jednotky, ovšem fantazii se meze nekladou. Hodnoty literálů mohou být těchto typů:

- unsigned long long int (reprezentující integer)
- long double (reprezentující desetinnou čárku)
- char (reprezentující znak)
- const char* (reprezentující řetězec)
- const char*, size_t size (taktéž reprezentující řetězec)

```
// kilometry
```

```
long double operator"" _km( long double x ) { return x*1000; }
```

```
// metry
```

```
long double operator"" _m( long double x ) { return x; }
```

```
cout << ( 1.0 _km + 20.0 _m ) << endl;    // vypíše 1020
```

2.3 Nová klíčová slova

2.3.1 `override` – překrytí

Klíčové slovo *override* označuje virtuální funkci, která překrývá jinou virtuální funkci. Pokud funkce nepřepisuje virtuální funkci rodičovské třídy, překladač vyhodí chybu. Funkcionalita je užitečná hlavně proto, že ihned zjistíme, jestli jsme např. v názvu funkce udělali překlep.

```
void foo() override;
```

2.3.2 `final` – omezení překrytí a dědění

Klíčové slovo *final* je možno použít ve dvou případech: k označení virtuální funkce, nebo k označení třídy.

Pokud *final* označuje virtuální funkci, znamená to, že nemůže být překryta v dědicí třídě. V tom případě se zapisuje na konci deklarace nebo definice funkce.

Pokud *final* označuje třídu, značí to, že z není možno dědit. V takovém případě se zapisuje ihned za jméno třídy.

```
struct Base
```

```
{  
  
    virtual void foo();  
  
};
```

```
struct A : Base
```

```
{  
  
    void foo() final; // Base::foo is overridden and A::foo is the final override  
  
    void bar() final; // Error: bar cannot be final as it is non-virtual  
  
};
```

```

struct B final : A // struct B is final
{
    void foo() override; // Error: foo cannot be overridden as it is final in A
};

struct C : B // Error: B is final
{ };

```

2.3.3 constexpr – konstantní výrazy

Klíčové slovo *constexpr* se používá pro označení proměnných, funkcí a jiných výrazů jako konstantní, tedy že se mají vyhodnotit už během překladač.

```

constexpr int square (int x) { return x * x; }

float a[square(9)]; // a má 81 prvků

```

2.3.4 noexcept – vyhazování výjimek

Klíčové slovo *noexcept* specifikuje, zda funkce může vyhazovat výjimky. Nevyhazující funkce mohou volat vyhazovací funkce. Pokud je vyhozena výjimka a vyhledávač její obsluhy narazí na nejvnější nevyhazující funkci, je zavolána funkce `std::terminate`.

```

void foo() noexcept;

```

2.3.5 default – výchozí konstruktory

Od C++11 můžeme v deklaraci funkce uvést klíčové slovo *default*, kterým explicitně říkáme, že chceme, aby překladač použil výchozí konstruktor, i když už máme v třídě jiné uživatelem nadefinované konstruktory.

```

class C{
    C(int i);

    C() = default;
}

```


2.4 nullptr

Pokud chceme nadefinovat ukazatel, který nemá hodnotu, můžeme mu nově přiřadit *nullptr*. V minulosti pokud jsme chtěli nadefinovat prázdný ukazatel, použili bychom 0 nebo NULL – zde ale mohlo dojít k chybě při překladu, jelikož 0 (nebo NULL převedený na 0) je chápána jako datový typ integer.

Hodnota *nullptr* je speciálního typu *std::nullptr_t*, proto ho můžeme použít jako argument funkce nebo konstruktor. Může se automaticky konvertovat na ukazatel, ale ne na integer.

```
void foo(char*);
```

```
void foo(int)
```

```
foo(NULL);    //do C++11 – není jisté na co se převede
```

```
foo(nullptr); //od C++11
```

2.5 Silně typové enumerace

Enumerace je výčtový typ, jehož hodnoty jsou omezeny na určitou množinu dat. V minulosti ovšem enum fungoval v podstatě jako integer, a jakýkoliv *enum* mohl být porovnán s integerem nebo enum jiného typu. Jelikož všechny *enum* jsou v základu integer, tak úplně jiné *enum* mohly být vyhodnoceny jako stejné, jelikož v základu měly stejnou int hodnotu. Hodnoty enum jsou navíc unscoped, a nemohly tedy existovat enumerace se stejným jménem.

```
// tento kód se nepřeloží
```

```
enum Meny {DOLAR, EURO, KORUNA};
```

```
enum PokryvkyHlavy {CEPICE, KSILTOVKA, KORUNA};
```

Silně typový enum, nebo *enum class*, je v C++11 nový výčtový typ který tyto problémy řeší. Odlišné výčtové typy spolu nelze porovnávat, a jejich hodnoty jsou vázány na jméno konkrétní *enum class*. Silně typový enum navíc může explicitně “dědit” z datového typu.

```
enum class Barva : unsigned int { Zelena = 0x00FF00, Cerna = 0x000000, Bila = 0xFFFFFFFF };
```

```
enum class Semafor : bool { Cervena, Zelena };
```

```
Barva b = Barva::Zelena;
```

```
Semafor s = Semafor::Zelena;
```

2.6 Inline namespace

Namespace slouží pro seskupení proměnných, funkcí a podprogramů do logického bloku. Ve velkých projektech slouží k předcházení konfliktů mezi jmény objektů – tedy umožňují vytvořit různé objekty se stejným názvem, pokud nejsou ve stejném jmenném prostoru.

Inline namespace je namespace, jehož členové jsou brány jako členové obalujícího namespace. Toto chování je přechodné pro více “úrovní” inline namespace – pokud máme namespace A, v něm inline namespace B, a v něm inline namespace C, tak členové z namespace C budou dostupné i v namespace A.

```
namespace A {  
    namespace B {  
        int foo() { return 1; }  
    }  
    inline namespace I {  
        int foo() { return 2; }  
    }  
}  
  
int getFoo {A::foo()};           //zavolá foo() z I  
int getFooB {A::B::foo()};      // zavolá foo() z B
```

2.7 Auto a decltype

2.7.1 Range-for

V C++11 můžeme nově procházet prvky v kolekci pomocí speciálního zápisu for smyčky:

```
for ( prvek : kolekce ) { ... }
```

2.7.2 auto

Pokud chceme nadefinovat proměnnou bez uvedení jejího typu, můžeme nově použít klíčové slovo *auto*. Překladač si potom za nás odvodí typ proměnné při inicializaci (tedy z přiřazené hodnoty, nebo typu proměnné kterou nám vrací funkce.) Nemůžeme proto proměnnou pouze deklarovat bez inicializace.

Používání *auto* má několik výhod:

- máme jistotu, že nedojde k implicitní konverzi (např. ze *size_t* na *int*)
- kód je obecnější, a proto je jednodušší provádět v něm změny
- při iterování, kde nás často typ iterátoru nezajímá, nám *auto* zajistí kratší a čitelnější kód

Je ale nutné mít na paměti že:

- *auto* udává pouze typ, ne specifikace jako *const* nebo *reference*
- nelze je použít pro datové proměnné, jejichž název je složen z více slov, jako např. *long long* nebo *long double*

Klíčové slovo *auto* můžeme také použít před názvem funkce. V takovém případě ale musíme také v deklaraci specifikovat koncový návratový typ (trailing return type).

```
auto func1(int const i) -> int  
{ return 2*i; }
```

Často můžeme *auto* vidět ve for smyčce kdy procházíme kolekce:

```
for (auto i : v) {...}
```

2.7.3 decltype

Při použití nového klíčového slova *decltype* překladač zjistí datový typ výrazu. Na rozdíl od klíčového slova *auto*, které se používá pro odvození datového typu proměnné z její inicializace, *decltype* se používá spíše v případech, kdy inicializace proměnné není jednoduchá – nejčastěji se jedná o dedukování návratového typu funkce.

```
int a = 1;           // a je typu int

decltype(a) b = a;   // decltype(a) je int
```

```
template <typename T1, typename T2>

auto add(T1 x, T2 y) -> decltype(x+y);
```

2.8 Variadické šablony

Variadické šablony jsou šablony, které mají parametry přebírající proměnné množství argumentů. Těmto argumentům se říká *Parameter pack* (balík parametrů). Časté použití je oddělení prvního předaného argumentu od zbytku.

```
void foo() { } // přetížení pro situaci kdy dojdou argumenty v „hlavní“ funkci f
```

```
template<typename T, typename ... X>

void foo(T prvni, X... zbytek)

{

    g(prvni);           // něco se provede s prvním prvek

    f(zbytek...);       // funkce se zavolá nad zbytkem argumentů

}
```

Pokud chceme funkci přetížit, musíme přetížení nadeklarovat před definováním hlavní funkce které předáváme argumenty. Variadická funkce „zná“ pouze funkce, které jsou nadefinované před ní. Pokud bychom *void f()* nadefinovali až po variadické funkci, překladač by vyhodil chybu a hlásil že funkci *f()* nemůže najít.

2.9 Lambda

V C++11 se nově objevují lambdy – funkcionalita, které můžeme nadefinovat uvnitř výrazů. Jedná se o výrazy, které můžeme použít jako inline funkce. Nejčastěji se používají jako parametr vstupující do jiné funkce, ale mohou pouze něco vykonávat:

```
[] { std::cout << "hello world" << std::endl; }
```

Lambda funkce vždy začíná hranatými závorkami, což je tzv. *lambda introducer*. V něm můžeme uvést *captures* – proměnné vyskytující se mimo lambda funkci, ke kterým chceme mít v lambdě přístup.

```
int i = 1;
```

```
auto a = [i] { return i * 2; }();
```

Lambda funkci můžeme uložit do proměnné a využít ji později v kódu. Můžeme také v kulatých závorkách před tělem funkce uvést parametry, podobně jako u normální funkce:

```
int number = 3;
```

```
auto lambda = [number] (int i) { return i * number; };
```

```
std::cout << lambda(2) << std::endl;    //vypíše 6
```

2.10 Reference na r-hodnotu, move semantics a perfect forwarding

C++11 umožňuje pracovat s r-hodnotami pomocí referencí, a díky tomu vyřešit problémy nazvané *move semantics* a *perfect forwarding*.

Přibývají také 2 nové funkce: *std::move* a *std::forward*. Navzdory jejich názvu nic nikam nepřesouvají ani neposílají – jsou to šablony funkcí které umožňují pracovat s r-hodnotami.

2.10.1 Reference na r-hodnotu (rvalue)

lvalue = l-hodnota = výraz který ukazuje na specifickou adresu v paměti, a můžeme ho teda adresovat & operátorem

rvalue = r-hodnota = výraz který na specifickou adresu neukazuje, nemůžeme jej adresovat a nemůžeme mu tedy ani přiřadit hodnotu. Typicky jsou to dočasné proměnné.

Na l-hodnotu se doteď mohla vytvářet reference pomocí ampersandu (&). Nově můžeme vytvořit refence i na r-hodnotu, a to pomocí dvou ampersandů:

```
int && i = 3; // i je l-hodnota, její typ je reference na r-hodnotu; 3 je r-hodnota
```

2.10.2 Move semantics, std::move

V C++98 bylo možné objekty pouze kopírovat, ne přesunovat objekty nebo hodnoty mezi nimi. Díky referenci na r-hodnotu nyní můžeme ovšem hodnotu rovnou přiřadit bez kopírování:

```
template <typename T>
```

```
class something {
```

```
public:
```

```
    std::vector<int> buffer;
```

```
    something (const something <T>& other) {
```

```
        // Copy konstruktor = Doposud jediný způsob s kopírováním
```

```
        // Zkopírování other.buffer.
```

```
        // Uvolnění this.buffer.
```

```
        // Přiřazení kopie do this.buffer
```

```
    }
```

```

something (something <T>&& other){

    // Move konstruktor = Nový způsob – předávání reference na r-hodnotu

    // Uvolnění this.buffer.

    // Přřadit other.buffer do this.buffer.

    // other.buffer odkazuje na nic (nullptr), this.buffer odkazuje na původní
    other.buffer

}

};

```

Můžeme tedy využít move konstruktor nebo move operátor přiřazení. Ty se automaticky vygenerují, pokud je splněno že v třídě:

- není uživatelem nadeklarovaný copy konstruktor
- není uživatelem nadeklarovaný copy operátor přiřazení
- není uživatelem nadeklarovaný move operátor přiřazení
- není uživatelem nadeklarovaný destruktork

Kopírování předané hodnoty použijeme, pokud chceme přiřazovaný objekt *other* zachovat. U move je přiřazovaná hodnota zahozena, vyhneme se zbytečným kopiím a program je tak rychlejší. Move se také využívá u datových typů které nelze kopírovat (např. `std::unique_ptr`), kdy místo zkopírování přesuneme vlastnictví hodnoty.

Objekt, ze kterého jsme data přesunuli (v dřívější ukázce *other*), bude validní, ale ocitne se ve stavu `unspecified`, tj. pokud se jedná o pole prvků můžeme mu např. přiřadit novou hodnotu, nebo zkontrolovat jestli je prázdné, ale nemůžeme si procházet jednotlivé prvky, protože nevíme kolik jich objekt obsahuje (většinou 0).

Pokud chceme z proměnné udělat r-hodnotu, můžeme využít funkce `std::move(x)`. Ta předanou hodnotu přetypuje na r-hodnotovou referenci. Pokud chceme zavolat funkci bez kopírování hodnoty, můžeme to provést následovně:

```

void bar(std::vector<int> x);

void foo() {
    std::vector<int> v;
    // ...
    bar(std::move(v));
}

```

2.10.3 Perfect forwarding, std::forward

Perfect forwarding umožňuje implementovat šablony funkcí, které vezmou předané parametry a předají je dalším funkcím tak, aby dostaly přesně ty stejné parametry jako šablonová funkce se zachovanými l-hodnotami a r-hodnotami.

Pokud tedy v následujícím kódu dostane funkce *wrapper* l-hodnotu, zavolá funkci A, pokud dostane r-hodnotu zavolá funkci B.

```

void f(X& p);           // A
void f(X&& p);          // B

```

```

template <typename T>
void wrapper(T&& p) {
    // Do some stuff
    f(std::forward<T>(p));
}

```

```

X y;

wrapper(y);           // calls f(X& p)
wrapper(X());         // calls f(X&& p)

```


Pokud máme typ `T` a vytvoříme referenci `T&&`, neznamená to, že je to vždy r-hodnotová reference. Za určitých okolností se totiž `T&&` může chovat i jako l-hodnotová reference (která má normálně zápis `T&`). Díky tomu mohou být navázány na cokoliv. Říká se jim *forwarding reference*.

Funkce `std::forward<T>()` provádí přetypování – pokud je předaný parametr r-hodnota, přetypuje ji na r-hodnotu; pokud je předaný parametr l-hodnota, přetypuje jej na základě `T` na l-hodnotu nebo r-hodnotu.

2.11 Vlákna

V každé aplikaci je jeden hlavní proces – funkce `main`. Tento proces můžeme rozdělit na více paralelně běžících podprogramů pomocí vláken. Vlákna můžeme vytvořit ze samostatných funkcí, funktorů, lambda funkcí a členských funkcí třídy.

```
#include <thread>
```

```
std::thread t(f);           // f je funkce, funktor nebo lambda
```

```
ClassExample ce;
```

```
std::thread t_class(&ClassExample::function, &ce, "parametr"); // zápis pro členskou funkci třídy
```

2.11.1 Funkce `join()` a `detach()`

Pokud vytvoříme vlákno, měli bychom zároveň pro něj ve funkci `main` zavolat funkci `join()` nebo `detach()`. Po ukončení své práce jsou totiž data vlákna stále někde uložena, a pokud neurčíme co s nimi chceme dělat, může docházet k memory leaku.

- `join()` - Čekáme, než vlákno dokončí svou práci a „spojí“ se opět s hlavním vláknem. Data jsou přečtena pro získání informací o jeho stavu, a poté jsou vymazána.
- `detach()` - Vlákno osamostatníme – nezajímá nás kdy a jak se ukončí. Data jsou na konci automaticky vymazána.

Pokud `join()` ani `detach()` nepoužijeme, program se přeloží, ale když vlákno skončí program bude ukončen.

Funkce `join()` a `detach()` se na vlákne mohou zavolat jenom jednou. Pokud některou z nich zavoláme vícekrát, program se sice přeloží, ale při druhém zavolání funkce spadne. Proto je nejlepší vždy napřed otestovat pomocí funkce `joinable()`, zda lze `join` nebo `detach` provést :

```
if (t.joinable()) { t.join(); }
```

2.11.2 mutex

Vlákna programu sdílejí adresní prostor, a proto mohou přistupovat ke stejným datům. Souběh (race condition) nastane, pokud se několik vláken zároveň s daty manipulovat, což vede k nežádoucímu chování. Zjednodušeně např.

- 2 vlákna `v1` a `v2` chtějí inkrementovat proměnnou `x=0`
- obě vlákna k proměnné dorazí ve stejný okamžik, uloží si, že na začátku `x` bylo 0
- `v1` inkrementuje `x` na 1
- `v2` by mělo správně inkrementovat `x` na 2, ovšem o činnosti `v1` neví a má stále uloženo že `x=0`; proto `x` inkrementuje na 1
- ve výsledku místo požadované 2 dostaneme 1

Pro předejití souběhu se využívá **mutex** a jeho funkce **`lock()`** a **`unlock()`**. První vlákno pomocí `lock()` zablokuje sekci kódu, a další vlákna poté musí čekat, než ji první vlákno opět odblokuje pomocí `unlock()`.

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

```
int counter = 0;
```

```
void race()
```

```
{
```

```
    mutex.lock(); // vlákno 1 se sem dostane jako první, jde dál;
```

```
        //vlákno 2 přijde později, musí čekat
```

```
    counter++; // vlákno 1 je zde, vlákno 2 se sem nedostane
```

```
    mutex.unlock(); // až se sem dostane vlákno 1, vlákno 2 bude moci pokračovat}
```

Další funkcí mutexu je **trylock()**. Ta se pokusí sekci zablokovat podobně jako **lock()**, ovšem ihned vrátí **true/false** podle toho, zda se zablokování podařilo. Vláknem tedy na odblokování čekat nemusí.

```
if(mutex.try_lock()){  
  
    counter++;  
  
    mutex.unlock();  
  
}
```

Pro mutex existují také wrappery - **std::lock_guard** uzamkne blok kódu před dalšími vlákny, dokud není nezavolán jeho destruktork (tedy po dokončení funkce, když je vyhozena výjimka,...). Nelze ovšem manuálně odemknout.

```
void locker()  
  
{  
  
    std::lock_guard<std::mutex> lg(mutex);  
  
    // zbytek funkce...  
  
}
```

3 ZADÁNÍ SAMOSTATNÉ PRÁCE

3.1 Zadání 1

- Vytvořte třídu Student, která bude obsahovat ID, jméno, seznam známek (známky mohou být i např. 2,5) a průměr známek
- Implementujte funkci pro přidání známky do seznamu známek. Funkce bude přebírat referenci na r-hodnotu
- Implementujte variadickou funkci pro výpis proměnných do konzole. Funkce vypíše vždy předaný název proměnné a její hodnotu. Volána bude např. `printArgs("Student name", _name, "ID", _ID)`;
- Implementujte funkci pro výpis informací o studentovi. Funkce bude využívat variadickou šablonu z předchozího bodu.
- Využijte lambda funkci pro výpočet průměru známek

3.2 Zadání 2

- Implementujte třídu BankAccount, která bude uchovávat informace o majiteli účtu (jako ID, jméno a příjmení) a peněžní stav
- Vytvořte členské funkce pro odeslání peněz, přijetí peněz a vybrání peněz
- Vytvořte vlákna pro
 - posílání peněz mezi účty
 - vybírání peněz
- Funkce vláken demonstруйте - vytvořte několik účtů a v mnoha vláknech mezi nimi provádějte transakce