Pokročilé programování v c++

Tomáš Turek

Přednáška 1

Výjimky

- Výjimky jsou oproti původnímu stylu odchytávání chyb lepší, že pokud celý kód proběhne bez chyb, tak je méně náročný na runtime. Za to když nastanou problémy, tak je runtime naopak o dost náročnější.
- *Původní způsob byl pomocí vracení kódů. Tedy if-else větve.*
- Obecně odchytávání výjimek je poměrně náročný problém. V průběhu místo vracení typu se musí najít přes hashovací tabulku pro nalezení správného místa pro návrat a následné ukončení a smazání veškerých předchozích proměnných.
- Velké chyby jako dělení nulou nejsou výjimkami. Pokud by to tak bylo, tak se bude muset vytvořit podstatně více kódů, protože dané errory mohou nastat víceméně skoro všude.
- Přirozený postup jak pracovat s výjimkami: 1. v `main` mít vždy `catch(...)`, 2. také předem dát `catch(const std::exception& e)`, 3. nové výjimky dědit z `std::exception`.

Exception Handling

- stack - return address při zpracování je používán jako nějaký hash klíč pro hledání správného místa v try blocku atd atd. - hash table - pomocí návratové adresy se musím dozvědět vše potřebné o místě, kam se vracím (její return address, velikost na stacku, počet proměnných atd..) - při překladu - fragmenty pro úklid z funkcí kde mohli vzniknout výjimky a data hash table
- hard error - not signalized exception :)

Rules:

- destructor neskončí výjimkou
- constructor static vars neskončí výjimkou
- move constr nevypouští výjimky

Překladače generují svoje implicit trycatch blocky (*compound object* - array/class allocation/constructon)

Z **MAINU** nevypouštět výjimky - microsoft třeba neuklízí stack pokud výjimka vyskočí z main (OS trochu uklízí, ofstream je třeba problém - žádný flush, ztracená poslední data, která se třeba neodeslala) **Neplatí** pro vlákna - exceptions v std::thread jsou chyceny v library - objeví se když další vlákno volá **join**

Přednáška 2

exceptions a locks `std::lock_guard` - constructor zamyká, destructor odemyká - RAII kategorie - RAII může vyžadovat přesně specifikované bloky `std::optional<std::lock_guard<std::mutex>> ...` - použití lock guards bez potřeby bloků

Exception safe programming

- **weak** exc. safety - funkce je weak safe, pokud po exception nechá data v konzistentním stavu
  - žádné memory leaks
  - všechny pointery null nebo valid
  - app-level invariants valid
- **strong** exc. safety - funkce je strong safe, pokud po exception nechá data v tom samém (observable) stavu jako když volána
  - = **Commit or rollback semantics**

Most parts std lib jsou strong exception safe. Příklad - `std::vector::insert`

- mark procedures which cannot throw by noexcept

Best practices

- CONSTR
  - Default constructor - lightweight
    - make `noexcept` if possible
  - většina non-trivial constr potřebují nějakou alokaci
    - nemůžou být `noexcept`
  - constr bez alokace můžou být `noexcept`
  - nezapomínat - single-param constr `explicit`
- DESTR
  - pro inherit hierarchy - `virtual destructor`
  - avoid data elements needing cleanup
  - pokud opravdu třeba - vždy noexcept
- Copy constr, Copy assignment
  - avoid explicit implement if possible
  - exception-safe implement of copy assignment
    - can reuse code already implemented

Přednáška 3

==Poznámky od Marka==

Templates - šablony template - generic piece of code Parametrizován typem, classou, integer const. - Class template - Global class - nested in others - Func templates - Global func - Member, inc constructor - Tzpe templates [C++11] - Var template [C++14] - `inline` needed when in .hpp - většinou jako global const **Template instancing** - použití šablony s daným typem - const parametrem Funkce často explicitně odvozené z argumentů (bez `<>`) ##### Templates a compilace ##### - compiler may check template code when defined - compiler generuje kód pouze pokud je template instanciován - různé instance nesdílejí kód (někdy size explosion) - kód generovaný pro templates je totožný pro non-templated kod - **no performance penalty** pro generičnost Implicitní instanciace - pokud je referencována nějaká specializace, která je zrovna potřeba (pokud je potřeba), template je instanciován (kompilace kódu) - instanciace member funkcí class pouze tehdy, když je funkce actually používaná - **definice templatu musí být visible at point of implicit instantiation** (constexpr trik ze cvičení zima) - **všechny šablonované metody** musí být vidět - v hlavičkovém souboru - definice templatu musí být visible v čas instanciace - classes a typy - visibility definic potřebná i v non-template - většina def class a typů v header - function templates - template visibility same as for `inline` func - most **template function def** musí být v header files ##### Multiple templates se stejným jménem ##### - Class - one *master* template `template<typename T> class vector {};` - any number of specializations které overridují master - partial `template<typename T, std::size_t n> class unique_ptr<T[n]> {};` - explicit `template<> class vector<bool> {};` - Func - any number of templates with same size - shared with non-template func - Type and var templates, concepts - pouze jedna def ##### psaní templatů ##### #### Validity of templates, concepts #### - Templates checked for validity - On definition: syntactit corectness, correctness of independent names - On instantiation: All rules of the language - Template nemusí být korektní pro všechny kombinace argumentů - compiler checkuje correctness pouze pro argumenty použité při instanciaci - před C++20 neexistoval žádný mechanismus, specifikující **requirements** pro argumenty templatů (**mechanismus concepts**) - C++20 - `requires` klauzule a `concepts` for constraining template args - překladače to umí, ale std concepty zatím nepoužívá #### C++20 Requires clauses #### **requires-clause** - constraint na template parametrech - evaluace compilerem v moment instanciace

```
template<typename IT, typename F>
requires std::is_invocable_v<F, typename std::iterator_traits<IT>::reference>
```

- C++17 - std::is_invocable_v variable template defined as

```
template<typename F, typename ... ArgTypes>
using is_invocable_v = is_invocable<F, ArgTypes...>::value;
```

- std::is_invocable is class template defined to llok like this:

```
template<typename F, typename ... ArgTypes> class is_invocable
{static constexpr bool value = /*...*/; };
```

- pokud nesplněna, deklarace funkce bude ignorována při hledání správného overloadu - ... #### C++20 Concepts #### - **concept** is logically a boolean function, jejíž argumenty jsou typy, templates nebo consts - většinou pouze jeden typename arg - eval by compiler

- definice Conceptu
  - může být pomocí **requires-expression**

  ```
  template<typename T> concept Deferencable = requires (T x) { *x; };
  ```

  - v tomto případě **requires expression** říká, že expression `*x` musí být sémanticky validní **pro x typu T**

  ```
  template<typename F, typename ... AL> concept Callable = requires (F t, AL ..
  al) { f(al ...); };
  ```

  - concept may also be defined using other concept or constant bool expression (`&&`, `||`) "'cpp template concept Reference = std::is_reference_v;

template concept ConstReference = Reference && std::is_const_v<std::remove_reference_t>;

```
- umí to *and* a *or* ale ne negaci
##### příklady conceptů ... #####
- concepty s explicit argumenty
- concepty s 1. implicit argumentem inferred z contextu

#### Variadic templates ####
- template heading - allows variable number of type arguments
```cpp
template <typename ... TList>
... doplnit
```

- named **function parameter pack**
- může být referencováno ve funkci - vždy pomocí suffixu `...` ...
- TList, Plist - překladačem držený seznam Typů a parametrů typů ... doplnit #### Perfect forwarding - motivace variadických ####

==Konec==

Šablony

- Mnoho využití i např. `auto && v` je vlastně využití šablony.
- Jak třídy, tak i funkce mohou být šablonované. Potom i globální proměnné a statické atributy.
- Překladač je schopen dedukovat parametry v `<>` pokud je hned instaciovaná.
```

- Vždy se generuje separátní kód pro speciální typ, ale není to na úkor výkonu.
- Vše musí být v **jednom stejném hlavičkovém souboru**.
- Definice musí být vždy instanciována, ale to mlže vést k problému. To lze obejít, pokud se donutí instanciace.

Specializace

- Specializované šablony pro dané typy. Pak jak v prologu se používá speciálnější definice, která je podobné dané instanci.
- To pak může vést k tomu, že je v šabloně více funkcí, respektive mnéně.
- To se týká pouze tříd.
- Pokud je řečená specializace, ale není definovaná, tak nelze použít.

Koncepty

- Pokud chceme v šabloně specifikovat, že daný parametr musí mít speciální funkci, nebo atribut. Pak následná kontrola kódu.
- Až v `C++20` pomocí `requires` klauzule a `concespts`.

Requires

```
template<typename It, typename F>
requires std::is_invocable<F, typename std::iterator_traits<IT>::reference>
// Chová se vlastně jako if. Nejdříve se dosadí a pak projde kontrola requires.
// Pokud nesplní, hledá se jiná možná funkce, jinak se vyhodí výjimka.
F for_each(IT a, IT b, F f);
```

- Také slouží k dokumentaci.

Koncepty

- Je *logicky* boolovská funkce vyhodnocovaná překladačem.
- Parametry jsou typy, šablony a konstanty.
- Pro definici lze použít *requires expression*.

```
template<typename T> concept Deferencable = requires (T x) {*x};
// Test zda-li T lze použít operátor *.
template<typename F, typename .... AL> concept Callable =
    requires (F f, AL ... al) {f(al ...);};
```

- Lze také definovat pomocí jiných konceptů, boolovských proměnných a kmobinací použitím `||` a `&&`.

```
template<typename T> concept Reference = std::is_reference_v<T>;
template<typename T> concept ConstReference =
    Reference<T> && stdLLis_const_v<std::remove_reference_t<T>>;
```

- Je jedno jestli se to přeloží nebo ne, jedná se jen o boolovskou funkci.

- Lze ihned psát koncepty přímo do hlavičky šablony.

```
template< Iterator IT, Callable<typename IT::reference> F>
void for_each( IT a, IT b, F f);
```

- Nebo při kontrole používání `auto`.

```
Iterator auto it = k.find(x);
```

Variadická šablona

```
template<typename ... TList>
void f(TList ... plist);
```

- Použití funkce s několika typy v závorkách. Poté to není nějaký kontejner, ale je to jen seznam daných parametrů, které jsou zvlášť uložené, jako separátní paramtery.
- Lze `TList` a `plist` obalit. Jako `const TList & ... plist`.
- Možnost *fold.*??

Přednáška 4

==Poznámky od Marka==

Class template argument deduction - CTAD

- ++efficient compilation, ++efficient generated code
- jinak user defined deduction guides - vlastní guide pomocí templates
- CTAD zlepšuje čitelnost #### Deducing this [C++23] #### > Hromada blbůstek, které stejně ještě nikde nejsou a nejdou použít

C++20:

```
class X
{
    T& get();
    const T& get() const; // invoked on const object - "const this"
}
```

```
class X
{
    T& get();
    T&& get() &&; // && invoked on temporary *this
}
```

WTF to jde

```
class X
{
    T& get();
```

```
    const T& get() const&;
    T&& get() &&;
    const T&& get() const&&;
}
```

Změny, hezčí v C++23

```
class X
{
    T& get( this X& );
    const T& get( this const X& );
    T&& get(this X&);
    const T&& get( this const X&&); // r value ref
}

class X
{
template<typename Self>
auto&& get(this Self&& self); // uni ref!
}
```

- `&&` ve funkci r value ref
- `&&` v templatech - uni ref (forwarding)

CRTP - Curiously recurring template pattern

- compile time polymorphism > ukázka využití dedukce this pro hezčí kód

Initializers

Initializer_list

- initialization of structures/containers

```
vector<int> v = {1, 2, 3};
vector<int> v  ({1, 2, 3});
vector<int> v   {1, 2, 3};
```

- template initializer_list
  - init pro user-defined classes

`... příklad`

Při kompilaci se init_list expanduje do lokální nepojmenovaného pole - unnamed array (lightweight object). Co když chceme vracet init_list - **reference na lokální array** - nebezpečné

```
initializer_list<int> f()
{
   return {1,2,3}; // dangerous
```

```
}
auto x = f();
```

Aggregate initialization

- init without explicit constructors

```
class Person
{
long id;
string name;
int age;
};
Person p = {123214, "James", 21};
```

- **Designated initilizers** [C++20]
  - abychom neztratili význam jaký parametr zadáváme

```
Point p(.x=3, .y=4); // struct/class
enum color( red, green, blue);
string color_names[]
{
[red] = "red",
[green] = "green",
...
};
```

- ++ pravidla aby toto ^^ fungovalo

Uniform initialization

- pravidlo - prostě chceme používat `{ }` pro init, protože jiné varianty někdy mohou mít trochu jiný outcome
- příklady → prezentace

Constructors and initializer_list

- `{ }` init vs init_list vs constructor overloading
  - `{ }` init **strictly** prefers init_list
  - ! pokud existuje lib způsob jak využít init_list, bude použito

```
vector<int> x(3, 1); // 1,1,1 - vector init
vector<int> y{3, 1}; // 3,1   - init list
```

- Guidances
  - "uniform" is not always uniform
  - syntax neintuitivní
  - při designu knihoven/tříd
    - design constructors carefully
    - overloads more carefully
  - pro uživatele knihoven/tříd

        * prefer `{ }` init (bez =)
            · some professionals prefer copy-init pro primitiv typy

const X constexpr X consteval X -init

- const
  - logical immutability
- constexpr
  - compile-time constant value
  - can be used in const expressions
  - can be applied to functions - tělo vyhodnotitelné za překladu > jaký je rozdíl mezi constexpr a céčkovým define
    > ...
    > týpek: jaký je rozdíl mezi automobilem a stromem ...
- consteval [C++20]
  - immediate function
  - applied only to **functions**
  - every call **must** produce compile-time constant "'cpp constexpr int fact(int n) {return n < 2 ? 1 : n*fact(n-1);} consteval int combination(int m, int n) {return ... };

int a = g(); fact(4); // OK Compile-time fact(a); // OK Runtime

combination(4,8); // OKOK Compile-time combination(a,8); // ERROR

```
- constinit [C++20]
    - mutable
    - **forces constant init** of static or thread-local vars
    - it can help to limit static order init nondeterminism

```cpp
int f() {...};
constexpr int g(bool p) { return p ? 42 : f(); }
constinit x = g(true); // OK
constinit y = g(false); // ERROR
```

Modules

Headers & include

- separate compilation
  - 50 years old
  - `#include`
- C++20 - long awaited
  - coexistence **include/import**
  - :| gcc 11+ incomplete
  - :(std::lib [C++23]
```

- **binary module interface** (BMI)
  - compiler/platform specific
  - compiler options
- export/import
- possibility to partition to submodules (asi jako python)
- velký speedup pro compilaci

==Konec==

Perfect forwarding - rules

- Reference collapsing rules
  - Používané pouze při template inferenci.
- "Forwarding reference" = "universal ref"
  - T && kde T je template arg

```
template<typename T> void f(T && p)
{
    g(p);
}

X lv;

f(lv);
```

- lvalue of type X
  - kompilátor používá `T = X &`, typ p je `X &` kvůli collapsing rules

```
f(std::move(lv));
```

- rvalue of type X
  - kompilátor používá `T = X`, typ p je `X &&`
- pokud dále v **f** voláme **g** a posíláme tam **p**, v obou variantách se jedná o lvalue referenci - neefektivní v druhém případě (není perfect forwarding)- **move lost**
  - pro perfect chceme rádoby podmíněný move
    - `std::forward<T>(p)`

```
template<typename T> void f(T && p)
{
    g(std::forward<T>(p));
}
```

- `T = X &`
  - reference collapsing - forward vrací X & - **lvalue**
- `T = X`
  - forward vrací X && - **rvalue**
  - forward se zde chová jako move

Typy			
L-value	proměnné X, X&, X&&	fce vracející X&	cast X &
R-value	konstanta	Fce vracející X, X&&	cast X, X &&

R-value vracející funkce neobvyklé nebo velmi známé (`std::move`, `std::forward`)

... correct implementation of emplace ...

Removing references when storing values

Př - storing values of any type

- [C++17] - deduction guides

```
template<typename T2>
ftor( T2 && p ) -> ftor<std::remove_reference_t<T2>>;

// umožňuje syntax
std::string s = "hello";
ftor x(s);
auto y = ftor(s);
```

- umožňuje z parametrů v ( ) závorkách odvodit, co dát do < >
- př `std::pair` - nemusíme explicitně psát parametry dvojice

Useful std lib - type traits

```
#include <type_traits>
```

- Type properties
  - `is_void_v`, `is_enum_v`, `is_pointer_v`, ...
  - logicky: **compile-time** functions vracející bool parametrizován typem
  - technicky: **constexpr** bool variable templates parametrized by type
    - `xxx_v<T>` zkratka pro `xxx<T>::value`
- Type transformations
  - `remove_reference_t`, `remove_cv_t`, `remove_cvref_t`[C++20]
  - logicky: **compile-time** functions vracející typ parametrizován typem
  - technicky: **type alias** (using) templates parametrized by type
    - `xxx_t<T>` - `typename xxx<T>::type`

Přednáška 5

auto - range-based for

- `auto` - C++11

- **structured return values**
  - tuple -> tie
  - C++14 auto return type ještě hezčí
  - C++17 **structured binding**

```
auto f(...) {... return {a,b,c};}
auto [x,y,z] = f();
```

- structured bindings - stejně jako tie, bez nutnosti deklarace proměnných
- return value syntax `->`
- auto vs decltype

Přednáška 6

Typy

Polymorfismus

- runtime - dynamický polymorfismus
  - flexibilní, ale přidává runtimový overhead
- kompilační polymorfismus
  - šablony a lambda výrazy
  - vše se řeší během překladu
  - nemá runtime overhead

Typová kompatibilita

- kompatibilita
- nominativní typing
- nominative subtyping
  - děděnní
- strukutrální typovaní
  - kompatibilita přes strukturu a ne dědičnost
  - pomocí šablon
  - kontextově založené: nějaké jsou si kompatibilní a nějaké ne
- přetěžování funkcí

type erasure

- je třeba v `std::function`, `std;:any` a `std::variants`
- vlastně kombinace šablon a dědičnosti

```
class Animal {
    template<typename T> Animal(const T& obj)
        : pet_(make_unique<Holder<T>>(obj)) {}
    string say() { return pet_->say(); }
```

```
    struct HolderBase {
        virtual ~HolderBase() {}
        virtual string say() =0;
    };
    template<typename T> struct Holder : public HolderBase {
        Holder(const T& obj) : obj_(obj) {}
        string say() override { return obj_.say(); }
        T obj_;
    };
    unique_ptr<HolderBase> pet_;
};

class Dog { auto&& say() { return "haf"s; } };
class Cat { auto&& say() { return "mnau"s;} };

vector< Animal> zoo;

zoo.emplace_back( Dog{});
zoo.emplace_back( Cat{});

for (auto&& a : zoo)
    cout << a.say();
```

`std::any`

- Do hodnoty můžeme přidat cokoliv.
- `type()` a `has_value()` metody.

`std::variant`

- Nadefinuje se, jaké hodnoty může mít. Jinak je dost podobné jako `any`.
- Funktor `std::visit` jako parametr dostane cokoliv, co je **Callable**.
  - Pokud chceme přetížit funkce, tak lze využít `overload`.

`optional`

`expected`

Paralelní programování

Hlavním problémem při paralelizaci je **race condition**. Nastává to při sdíleném *stavu*. Záleží to na plánovači.

Vlákna

- Jsou v hlavičkovém souboru `<thread>`. Má *Frok-join paradigma*. Třída `thread` a namespace `this_thread`.

– Konstruktor vytvoří nové a vlákno a ihned ho spustí. !Vytváření vlákna je hodně pomalé!
– Destruktor hlídá stav a pokud je `joinable`, tak ukončí pomocí `terminate()`.
– Implicitně má vlákno, že je `joinable`.
– Pokud další vlákno, běží, tak `join` se čeká.
– Pomocí `detach` se vlákno odpojí a osamostatní, potom nelze zavolat `join`.

Futures

Přednáška 7

Synchronizační primitiva

Mutual exclusion *MUTEX*

- Jen na kritickou část kódu. Je v `<mutex>` a `<shared_mutex>`.
- Hlavní funkce `lock` a `unlock`, která zamiká danou kritickou část.
- Jen jeden může být uvnitř kritické části.
- Také funkce `try_lock` ale to je *nedoporučené*! Protože se pak **aktivně** čeká oproti `lock`.
- Pro použití je lepší ud1lat třídu pro zabalení, která má vlastní mutex a pro veřejné funkce se na začátku zamkne a na konci odemkne. Také se tomu říká *monitor*.
- Pokud bych, ale volal uvnitř funkce jinou funkci s mutexem, tak už to nepůjde. Pro toto použití se lze využít `recursive_mutex`. Lepší je vyhodit funkčnost dané funkce ven do privátní funkce bez zámku a pak jej volat.
- V případě výjimky se neodemče!

Další varianty

- `timed_mutex` obdobně jako standardní `mutex`, ale `try_lock` mají i verze pro `try_lock_for` a `try_lock_until`.
- `recursive mutex` zase obdobný, ale pokud se udělá `lock` ze stejného vlákna, tak se jen sčítá počet, kolikrát tam je a pak při `unlocku` se odečítá. Je ale pomalejší.
- `recursive_timed_mutex` je kombinací obou dvou.
- `shared_mutex` oproti normálnímu `mutexu` umožní více vláknům se "dívat" na sdílené prostředky. Pro to lze použít funkci `lock_shared`. Normální `lock` stáe funguje stejně a čeká na exklusivní zámek a ani nepouští žádné jiné vlákna ani do `lock_shared`.
- Pak je i `shared_timed_mutex`.

Mutex wrapers

- `std::unique_lock` zamykací třídá s více funkcemi a vlastnostmi.
- `std::lock_quard` je to scope based lock neboli **RAII**. Je jen na `lock` nikoliv na `shared_lock`.

```
{
    // Automaticky se ihned zamkne.
    std::lock_quard<std::mutex> lk(mtx);
    lst.push_front(x);
} // Automaticky se odemkne i když se vyhodí výjimka.
```

- `std::shareed_lock` pro shared mutex.
- Pak je i variadický wrapper `template<typename ... MutexTypes> scope_lock`, který může mít několik zámků najednou. Opět **RAII** a je dobrý pro vyvarování se dedalocku. Deadlocku lze také zabránit pokud budeme zamykat mutexy vždy ve stejném pořadí.
- Ještě se hodí `std::size_t hardware_destructuve_interference_size;` je kontanta pro velikost cache line. Problém je že se může do jedné cache liny načíst dvě proměnné. Proměnné mohou být vedle sebe a protože cache musí být v rámci procesorů konzistentní, tak se pořád zapisuje a čte cache do a z paměti, říká se tomu *ping pong*. Lze se tomu vyvarotat pomocí volání `align()`. Nejlépe se tomu vyvarovat obecně.

Zamykací algoritmy

- `std::lock`, který zamyká více mutexů. Místo toho se používá `scope_lock`.
- Pak i `std::try_lock` opět se nedoporučuje.

Call once

- `std::once_flag` pmocný objekt pro `std::call_once`.
- `std::call_once` zavolá funkcni jen jednou i když je volaná z více vláken.

```
std::once_flag flag;
void do_once(){
    std::call_once(flag, [](){/* do somehting */});
}
std::thread t1(do_once);
std::thread t2(do_once);
```

Condition variables

- Pro čekání na kritické části, ==není zcela vhodné==.
- Nefunguje samo o sobě.
- `std::conditional_variable` může blokovat jedno nebo více vláken ve stejný čas dokud:
  - Jsme nezískali notifikaci z jiného vlákna.
  - Vyprší časovač.
  - !*spurious wakeups* - to jsou špatné probuzení i když nikdo neprobudí.

- `wait` automaticky manipuluje s mutexem (taky je třeba do něj vložit `std::unique_lock`) a `notify` pak oznamuje, ale neukládá se a tedy pokud se zavolá dřívě, než `wait`, tak je zbytečné a potom, když už bude `wait`, tak se nedočká.
- Příklad je **producer and consumer**.
- Smotný `wait` mění zámek dle toho, pokud je notifikován nebo ne.

Semafor

- V hlavičce `<semaphore>` a velmi standardní primitivum. ==Doporučený==
- Buď počítá a tedy `std::counting_sempahore` a konstruktor vytvoří počítadlo. A manipulace pomocí funkcí:
  - `acquire()` - pokud je větší než 0, tak ho dekrementuji a jdu dál. Jinak se zablokuji. Známe jako *down*.
  - `releas(count=1)` - podívá se do fronty a pokud je tam zablokované vlákno, tak ho probudím a vyskočím. Jinak zvýším počítadlo o 1. Známé jako *up*.
  - Může být buď jen nezáporné, nebo i záporné. Chová se to stejně, jen jsou jiné detaily.
- Nebo `std::binary_semaphore`.
  - Jen na 0 a 1 a vlastně dělá jen to stejné co mutex. Ale převážně se to používá na probuzení jiného vlákna.
  - Nicméně mutex je rychlejší a v krizových částí je lepší použít než semafor.

Přednáška 8

Coordination types

- Možné bariéry na koordinaci vláken.

Latches

- Koordinační mechanismus. Je to jen na jednorázové použití.
- Čeká se dokud na dané místo nedorazí dostatečný počet vláken.
- Je to v `<latch>`.

Barrirers

- Možné použít víckrát. V hlavičce `<barrier>`.
- Podobným způsobem jako `latch`.
- Sekvence fází:
  - Nejdříve je `arrive()` a dekrementuji počet vláken a `wait()` čeká.
  - Pokud je počítadlo na nule, tak se volá funkce a odblokují se vlákna.
  - Počítadlo se resetuje *oproti* `latch`.

Stop tokens

- Jak signalizovat konec výpočtu pro ostatní vlákna, že už třeba není nutnost dále pokračovat.
- `stop_token` je získán "hlavním vláknem" a pak pomocí `bool stop_requested()` a `bool stop_possible()` se sám zastavuje.
- `stop_source` generace stop tokenů pro vlákna pomocí `stop_token get_token()`. Každému vláknu se vloží jeden token. A následně lze zavolat `request_stop()`.
- `stop_callback` vyvolání callbacku.
- Místo `thread` lze pužít `jthread`, který už přímo pracuje se stop tokenama.

Thread local storage

- Jiný přístup k uchovávání proměnných. Mimo globálních (*které jsou sdílené v rámci celého programu a teky všech vláken*) a lokálních (*žije jen v nějaké době a je privátní proměnná jednoho vlákna*), pak je taky nový keyword `thread_local`.
- To se používá na globální proměnná, která se chová jako privátní proměnná pro jedno vlákno.
- Životnost je během živat celého vlákna.

Parallel algorithms

- V hlavičkách `<algorithms>` a `<numeric>`. Je možné používat skoro všechny algoritmy také paralelně.
- `seq` - počítání sekvenčně.
- `par` - ve více vláknech a nepoužívají se vektorové konstrukce.
- `par_unseq` - použití vektorových konstrukcí .
- `unseq` - jestli se používá jedno vlákno a vektory.
- `for_each`
- `reduce` - paralelní paradigma pro udělání asociativní a komutativní operace na všech prvcích paralelně. **Důležitá operace**.
- `scan`
- `transform_reduce`
- `transform_scan`
- Nesmí být vyvolaná vyjímka, jinak to celé spadne.

Paměťové metody, atomické operace a lock-free struktury

Memory models

- Jaké operace můžeme provádět s pamětí.

Weak

- **Velmi slabý model** zaručuje konzistenci v rámci jednoho vlákna. (*C++11*)
- To kdy se ty konstrukce `read` a `wirte` volají se neví a není nijak konzistentní co se týče více vláken.
- **Slabý model** zaručuje, že pokud se zapíše A do B, tak potom B je stjené jako A. (*ARM*)

Strong

- Pořadí těch zápisů odpovídá pořadí `store`.
- **Obvyklý model** (*x86/64*)
- **Sekvenční konzistence** zaručuje se, že i `load` je tak jak byl volaný. Není žádný *HW*, ale jen pomocí softwaru (v jazyce *Java* a *C#*).

Barriers (Fence)

- Acquire fence
  - Pokud je někde vložena, tak všechny `read` nesmí předběhnout danou fence pro load.
- Release barrier
  - Opačný směr por store, tedy `write` nesmí být opožděn až za daný fence.

Atomic operations

- V hlavičce `<atomics>`. Je zde možnost vytvářet lock-free algoritmy.
- Taky je možné vytvářet fence. A práce s memory ordering.
- Lock-free algoritmy jsou na lock-free sturkturách. To znamená, že jsou speciálně napsané, tak že se nijak neblokuje a není tam mutex, ale i přesto funguje v paralelním prostředí. Jsou rychlejší, ale komplikovanější. Nicméně se používá aktivní čekání.

Memory ordering

- `enum memory_order;` ten má
  - `memory_order_seq_cst` - nejvíc striktní a sekvenčně konzistentní.
  - `memory_order_relaxed` - default a nejvíc divoký.
  - `memory_order_acquire`, `memory_order_release` a `memory_order_acquire_release` které odpovídají fencím.

```
#include <atomic>

struct Counter{
    std::atomic<int> value;
    void increment() { ++value; }
    void decrement() { --value; }
```

```
    // Taky .store() a .compare_exchange(....)
    int get(){ return value.load(); }
};
```

- Šablona `atomic<T>` je pro všechny typy a pro několik jsou specializované.
- Taky možnost `wait` a `notify` na čekání získání hodnoty a odblokování vláken. *není důležité*
- `compare_exchange(....)`hodnota se porovná s další a pokud je stejná, tak se vymění za novou. Odpovídá abstraktní instrukci `CAS`.
- U `load`, `store` a `compare_exchange` je i `memory_order` jako parametr.
- `void atomic_thread_fence(memory_order order) noexcept;` je pro vložení fence do kódu.
- `atomic_flag` povolí jednobitový atomický test.

Přednáška 9

Lock-free programování

- Vytváření struktur a algoritmů pro více jader bez použití zámků.

CAS loop

```
for(;;){ // aktivní čekání
    // čtení kritických dat do lokální proměnné
    list_node *old_head = head;
        // tohle musí být atomické a použít load
        // se sémantikou memory_order_acquire
    // spekulativně měnit data
    pushed_member->next = old_head;
    // CAS - pokus o zapsání kritických dat
    if(CAS(&head, pushed_member, old_head) == old_head) return;
        // CAS je atomická operace a díky tomu to funguje
        // acquire pro load a release pro read
}
```

ABA problém

- Problém, který se může vyskytnout při lock-free programování.
- Porovnávaná data mohou být jiná i když se tváří že jsou stejné (mohou mít třeba stejný pointer).
- Spočívá to v tom, že někdo jiný může strukturu změnit, ale i tak to vypadá správně.
- Jak se tomu lze vyhnout.
  - Buď používat nějaký index, který bude zaručeně vždy jiný (index, ne pointer).
  - Vytvoření počítadla pro používání struktury.

* Použití velkého CASU, do kterého lze přidat počítadlo.

Korutiny

- Podpbné jako subrutiny. To jest, že se můžou vrátit a mohou být volány.
- Oproti subrutinám se mohou sami zastavit a někdo jiný je může probudit.
- Užitečné jsou ==generátorry==.

Stackful korotuny

- Také známé jako *fiber*.
- Každá má svůj vlastní zásobník.
- Mohou být přiřazené k vláknům a taky odebrané od vlákna.
- Není jazykově implementovaná, ale lze ji implemenotvat v kinhovně.

Stackless korotuny

- Nemají svůj vlastní zásobník a využívají zásobník volajícího.
- Stav se uchovává na heapu.
- Musí mít jazykovou podporu.
- Je jednodušší a rychlejší.

Užití v `C++`

- Jak detekovat korotuny. A to pouze použitím keywordů výrazů `co_await`, `co_yield` a statement `co_return`.
- `co_await` všechny lokální proměnné se uloží na heap. Následně se musí vytvořit volatelný objekt.
- **Coroutine handler** je typ pro ukládání na heap. Jedná se vlastně o `C` pointer `std::coroutine_handler<>`. Pro zničení se používá funkce `destroy`.

```
struct Awaiter {
    std::coroutine_handle<> *hp_;
    constexpr bool await_ready() const noexcept { return false; }
    void await_suspend(std::coroutine_handle<> h) { *hp_ = h; }
    constexpr void await_resume() const noexcept {}
};

ReturnObject counter(std::coroutine_handle<> *continuation_out) {
    Awaiter a{continuation_out};
    for (unsigned i = 0;; ++i) {
        co_await a;
        // use i here
    }
}
```

```
void main() {
    std::coroutine_handle<> h;
    counter(&h);
    for() {
        h();
        // unable to get i, just call
    }
    h.destroy();
}
```

- V hlavičkovém souboru `<coroutine>` a zde je `std::suspend_always` a `std::suspend_never` a ty vrací `await_ready` vždy `false` repsektive `true`.
- Vrací se speciaální struktura `ReturnObject`, která musí mít v sobě strukture `promise_type`.
- `co_yield` je dost podobná `co_await`.

```
struct ReturnObject {
    struct promise_type {
        unsigned value_;
        ReturnObject get_return_object() {
        return { // Uses C++20 designated initializer syntax
            .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
            };
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_never final_suspend() { return {}; }
        void unhandled_exception() {}
        std::suspend_always yield_value(unsigned value) {
        value_ = value;
        return {};
        }
    };
    std::coroutine_handle<promise_type> h_;
};

ReturnObject counter() {
    for (unsigned i = 0;; ++i)
        co_yield i;
    // co yield i => co_await promise.yield_value(i)
}

void main() {
    auto h = counter().h_;
    auto &promise = h.promise();
    for (int i = 0; i < 3; ++i) {
        std::cout << "counter: " << promise.value_ << std::endl;
        h();
```

```
    }
    h.destroy();
}
```

- `co_return` označuje, že korutina byla ukončena. Také vrací danou hodnotu.

```
struct ReturnObject {
    struct promise_type {
        unsigned value_;
        ~promise_type() { /* do something */ }
        ReturnObject get_return_object() {
            return {
            .h_ = std::coroutine_handle<promise_type>::from_promise(*this)
            };
        }
        std::suspend_never initial_suspend() { return {}; }
        std::suspend_always final_suspend() { return {}; }
        void unhandled_exception() {}
        std::suspend_always yield_value(unsigned value) {
            value_ = value;
            return {};
        }
        void return_void() {}
    };
    std::coroutine_handle<promise_type> h_;
};

ReturnObject counter() {
    for (unsigned i = 0; i < 3; ++i)
        co_yield i;
        // falling off end of function or co_return;
}

void main() {
    auto h = counter().h_;
    auto &promise = h.promise();
    while (!h.done()) { // Do NOT use while(h) (which checks h non-NULL)
        std::cout << "counter: " << promise.value_ << std::endl;
        h();
    }
    h.destroy();
}
```

`std::generator`

- V `C++23` je jednoduchá podpra pro generátor.

Cherry picking

Databáze

- Připojení databázi je pomocí knihovny na danou databázi.
- Víceméně všechna chování veškerých databází je stejné. (*kromě MySQL*).
- Není zatím žádný standard v `C++`.

Network

- Zatím není žádná základní podpora pro sítování.
- Lze využít **BSD** sokety.

Asynchroní I/O

- Taká není, možná v `C++26`.

Filesystem

- Už je v `C++17`.

Shared memory

- Není a nebude.

Boost

- Přenositelná **velká** knihovna. Zdrojáky a testování nových funkcí.
- Pokud je v Boostu něco dobrého, tak se to promítne do standardu.

SFINAE