

# REKAPITULACE, VÝHLED DO BUDOUCNA

PB161 PROGRAMOVÁNÍ V JAZYCE C++

---

Nikola Beneš

14. května 2024

## REKAPITULACE

---

## Hodnotová sémantika C++

- fixní vazba proměnné na objekt (po dobu její existence)
  - objekt – abstrakce „místa v paměti“
- důsledek: kopírování při inicializaci / přiřazení, volání funkce hodnotou
- možnost definovat, jak se kopírují uživatelsky definované typy
  - kopírovací konstruktor, kopírovací operátor přiřazení
- deterministický životní cyklus objektů
  - přesně dané pořadí inicializace a destrukce
  - život lokálních / globálních proměnných
  - život podobjektů spjatý s životem objektu
  - život dočasných objektů  
(může být prodloužen vazbou k referenci)

## Vlastnictví

- „kdo je zodpovědný za úklid“
- „úklid“ = volání destruktoru
- syntaktické (implicitní)
  - položky složených typů (podobjekty), lokální proměnné
  - úklid zajišťuje přímo sémantika jazyka
- sémantické (explicitní)
  - zajišťuje programátor uživatelsky definovaných typů

## Reference

- není skutečná hodnota, ale *alias*
  - jiné jméno pro existující objekt
- slabší, ale bezpečnější forma ukazatele
  - nedá se přesměrovat, nemůže být *null*, vždy je inicializovaná
- nekonstantní lvalue reference (**typ&**) se váže jen k l-hodnotám
- konstantní lvalue reference (**const typ&**) se váže k čemukoli
- rvalue reference (**typ&&**) se váže jen k pomíjivým objektům

## Doporučení

- pokud možno dávejte přednost referencím před ukazateli
  - dává smysl *null*? dává smysl přesměrování?
  - jsou-li odpovědi *NE*, pak chcete použít referenci

Pozor na *dangling reference* (odkaz na již neexistující objekt)!

## Klíčové slovo `const`

- deklaruje záměr neměnit hodnotu
- deklarace neměnných proměnných
- deklarace ukazatelů / referencí, skrze které nebude možno měnit odkazovaný objekt
  - pozor na rozdíl mezi `const int*` a `int* const`
- kvalifikace metod, které nehodlají měnit aktuální objekt
  - přetížení metod lišících se jen v `const`

## Doporučení

- používat všude, kde nechcete modifikaci
- ne v návratových hodnotách funkcí
  - je-li to skutečně hodnota, ne reference
- nebývá zvykem u hodnotových parametrů funkcí

## Přesouvací (move) sémantika

- chceme se vyhnout zbytečnému kopírování *pomíjivých* objektů
  - *r-value*
  - dočasná hodnota (po *materializaci*)
  - objekt, který je explicitně označen jako pomíjivý pomocí `std::move`
- realizuje se pomocí *rvalue* referencí (`typ&&`)
  - vážou se pouze k pomíjivým objektům
- hojně využívána standardní knihovnou
- přesouvací konstruktor, přesouvací operátor přiřazení
  - možnost „vykrást“ zdroje objektu „na pravé straně“

## Vynechání kopie (*copy elision*) – povinné (od C++17)

- inicializace objektu dočasnou hodnotou
- **return** dočasné hodnoty z funkce (RVO)
- dáno pravidly pro materializaci dočasných hodnot

## Vynechání kopie – nepovinné

- **return** lokální hodnotové proměnné z funkce (NRVO)
  - (tj. ne reference)

## Implicitní použití přesunu

- **return** lokální proměnné / parametru z funkce
  - hodnotového typu nebo typu r-hodnotové reference
  - pokud nedojde k NRVO
  - jakoby obalení **std::move**

## Automatická dedukce typu

- klíčové slovo **auto**
  - dedukce typu podle inicializace
  - **auto** bez & znamená vždy hodnotu
  - použití pro funkční parametry – generické funkce
- klíčové slovo **decltype**
  - **decltype**(proměnná) je typ, se kterým byla deklarována proměnná
  - **decltype**(výraz) je typ výrazu

### **std::decay\_t**<typ>

- odstraní z typu **const** a reference
- užitečné, chceme-li získat hodnotový typ z výrazu typu reference (dereference iterátoru, indexování kontejneru)

## Třídy v C++

- uživatelsky definované typy
  - členské proměnné – položky typů
  - členské funkce – metody (skrytý parametr **this**)
- klíčová slova **class, struct**
  - liší se jen implicitními přístupovými právy
- konstruktory
  - inicializační sekce
- přístupová práva **public, protected, private**
  - deklarace přátel **friend**
- dědičnost (v tomto předmětu pouze veřejná)

## Typové konverze

- implicitní
  - číselné povýšení (*promotion*), číselné konverze
  - potomek → předek
  - uživatelsky definované konverze – konstruktory, operátory
- explicitní
  - **static\_cast**
  - **dynamic\_cast** v kontextu podtypového polymorfismu

## Typové aliasy

- **using** nové\_jméno = typ;
- „lepší **typedef**“

## Přetěžování

- funkcí, metod, operátorů, konstruktorů
- musí se lišit alespoň
  - počtem nebo typem parametrů
  - u metod kvalifikátorem **const**
- konstruktor může delegovat inicializaci na jiný

## Přetěžování operátorů

- volné (**friend**) funkce vs. metody
- speciální operátor  $\leq$   $\Rightarrow$  pro porovnávání
  - možno nechat vygenerovat automaticky
  - různé druhy uspořádání

## Funkční objekty

- s přetíženým operátorem volání ( )
- použití s generickými funkcemi – **auto**
- chceme-li někam ukládat – **std::function**
  - dražší volání, používat jen pokud nutno

## Uzávěry (lambda funkce) (C++11)

- `[kontext](parametry) -> návratový typ { tělo }`
- funkční objekt anonymního typu
- možnost zachytávat kontext
  - hodnotou: `x, =, *this`
  - referencí: `&x, &, this`
  - (pozor na život objektů chycených referencí!)
- **mutable** chceme-li v těle měnit hodnotou chycené proměnné

## Podtypový polymorfismus

- pozdní vazba metod – klíčová slova **virtual, override**
  - implementace pomocí virtuálních tabulek a ukazatelů na ně
  - statický vs. dynamický typ objektu
- veřejná dědičnost – vztah IS-A (*Liskov Substitution Principle*)
- často raději preferujeme kompozici – vztah HAS-A
  - pořadí volání konstruktorů a destruktorek

## Na co si dát pozor

- dealokace skrz ukazatel na předka bez virtuálního destruktorek je nedefinované chování
- při kopírování do objektu předka dochází k ořezání (*slicing*)
  - polymorfnní objekty v kontejnerech / položkách složených typů musí být dynamicky alokovány

Toto je jedna z možností realizace polymorfismu v C++, ale *ne jediná!*

## Výjimky

- používáme k řešení chyb / výjimečných situací za běhu
- *NE pro běžný tok řízení (control flow)!*
- doporučení: házejte hodnotou, chyťte referenci
- moderní implementace:
  - běžný běh programu není existencí výjimek zpomalen, ale jakmile je výjimka skutečně vyhozena, dojde ke zpomalení
- specifikace **noexcept** – úmysl nevyhazovat z funkce výjimku
  - přesouvací konstruktor / operátor přiřazení by měly být pokud možno **noexcept** – kvůli chování standardní knihovny
- nemáme **finally** – pro úklid používáme *destruktory*
- hierarchie výjimek ve standardní knihovně `<stdexcept>`
- záruky standardní knihovny (*exception safety*)
  - *basic / strong / nofail*

## Princip RAII

- obecný způsob práce se zdroji (už od pradávných dob C++)
- zdroj je spjat s životním cyklem objektu
  - jedna třída pro jeden typ zdroje
  - jedna instance třídy pro jednotku zdroje
  - inicializace objektu = získání zdroje
  - konec života objektu = uvolnění zdroje

## Rule of Zero / Five

- pokud třída nespravuje zdroj, není důvod psát kopírovací / přesouvací konstruktor / operátor přiřazení a destruktork
- pokud třída spravuje zdroj, pak je vhodné implementovat všechny zmíněné metody
  - nedává-li smysl, = **delete**
  - stačí-li implicitní, = **default**

## Práce s pamětí

- nízkourovňová: **new** a **delete**
  - v moderním C++ pokud možno **nepoužíváme**
- vysokoúrovňová – použití standardní knihovny
  - kontejnery
  - chytré ukazatele
  - využívá principu RAII

## Doporučení – první použitelná možnost v tomto pořadí

- žádná dynamická alokace (hodnotové proměnné, položky)
- **std::unique\_ptr** (jeden vlastník)
- **std::shared\_ptr** (více vlastníků, poslední uklízí)

## Standardní knihovna

- `std::pair`, `std::tuple`, `std::array`
- kontejnery, iterátory
- algoritmy (klasické, *ranges*)
- chytré ukazatele: `std::unique_ptr`, `std::shared_ptr`
- `std::function`
- `std::optional`, `std::variant`, `std::any`
- `std::string`, `std::string_view`
- vstup a výstup, formátování

## Dvojice a ntice

- `std::pair` pro dvojice
  - položky `first` a `second`
- `std::tuple` (C++11) pro libovolné ntice
  - přístup k položkám pomocí `std::get<N>(...)`
  - vytvoření ntice referencí pomocí `std::tie`
- rozbalování ntic (C++17) – *structured bindings*
  - funguje i pro pole, `std::array`,  
vlastní složené typy s veřejnými položkami

## Pole

- `std::array`
- obalené Céčkové pole s metodami a chováním jako kontejnery

## Kontejnery

- sekvenční: `std::vector`
  - dynamické pole (se zvětšováním)
  - základní obecně užitečný kontejner
- asociativní: `std::set`, `std::map`
  - množina / slovník
  - uspořádané dle klíče
  - `multi`-verze s opakováním klíčů
- přístup pomocí iterátorů
- invalidace (iterátorů, referencí, ukazatelů)
- `std::span` – pohled do pole
  - pro libovolný kontejner nad souvislým úsekem paměti
  - pro čtení i pro modifikaci

## Algoritmy – klasické

- rozsáhlá knihovna užitečných algoritmů
- vstupem typicky interval – dvojice iterátorů
- výstup dán jedním iterátorem

## Ranges

- pojem iterovatelného objektu (*range*)
- „hezčí“ verze klasických algoritmů
  - vstupem dvojice iterátorů nebo *range*
- pohledy (*views*) pro „líné“ zpracování

## Doporučení

- nevynalézat kolo
- má-li kontejner přímo metodu, preferovat tu

## Chytré ukazatele

- automaticky ve svém destruktoru dealokují paměť
- `std::unique_ptr` – unikátní vlastník
  - nekopírovatelný, přesouvatelný
  - nulová režie za běhu
- `std::shared_ptr` – sdílené vlastnictví
  - založený na počítání odkazů (*reference counting*)
  - kopírování je sdílení; poslední vlastník dealokuje
  - komplikovanější, nutná režie pro počítání vlastníků (musí fungovat i v paralelním prostředí)
- (v tomto předmětu jen základní použití)

### `std::function`

- uchovává libovolný (kopírovatelný) funkční objekt
- doporučení: používat jen, je-li to nutné

### `std::optional`

- objekt s volitelnou hodnotou
- přístup přes metody nebo operátory `*` a `->`

### `std::variant`

- disjunktní sjednocení – typově bezpečný **union**
- přístup přes `std::get` nebo `std::get_if`
- volání funkčního objektu pomocí `std::visit`

### `std::any`

- drží libovolný (kopírovatelný) objekt

## Práce s řetězci

- různé typy pro „znaky“ (`char`, `wchar_t`, `char32_t`, ...)
  - odpovídající řetězcové literály
  - odpovídající verze `std::string` apod.
  - (na mnoha místech jen `char` / `wchar_t` verze)
- `std::string` – funguje podobně jako vektor znaků
  - moderní implementace typicky optimalizují pro malé řetězce
  - jiná pravidla pro invalidaci
- `std::string_view` – pohled na řetězec nebo jeho část
  - drží jen ukazatel a délku
  - jen pro čtení nebo `substr`
  - vždy předáváme hodnotou (nekopíruje se obsah řetězce)
  - zneplatňuje se podobně jako iterátory / reference
- `std::format` – nový způsob formátování řetězců

## Proudový vstup a výstup

- abstrakce pomocí tzv. proudů (*streams*)
- standardní vstup a výstupy
  - `std::cin`, `std::cout`, `std::cerr`, `std::clog`
- operátory vstupu a výstupu: `>>`, `<<`
- načtení řádku do `std::string` pomocí volné funkce `std::getline(std::istream&, std::string&)`
- chybové příznaky `eofbit`, `failbit`, `badbit`
  - test pomocí typové konverze na `bool`
- souborové proudy
  - `std::ifstream`, `std::ofstream`, `std::fstream`
- řetězcové (paměťové) proudy `std::stringstream` apod.
- I/O manipulátory
- proudové iterátory

# POKROČILÁ TÉMATA C++

---

## CO JSME V TOMTO PŘEDMĚTU VYNECHALI

- generické programování – *šablony*
  - generické funkce, třídy, atd.
  - od C++20 navíc s koncepty
  - variadické šablony, *fold expressions*
- programování v době kompilace, metaprogramování
  - pomocí šablon i jinak
  - *type traits*
- universální reference, *perfect forwarding*
- vícenásobná a virtuální dědičnost; detaily RTTI
- pokročilejší práce s chytrými ukazateli; `std::weak_ptr`
- paralelní programování v C++, vlákna, synchronizace, ...
- různé části standardní knihovny
  - `<chrono>`, `<random>`, `<regex>`, `<filesystem>`, ...
- různé užitečné idiomy (moderního) C++
- korutiny
- a mnoho dalšího...

## PV264 Seminar on programming in C++

- praktičtěji zaměřený
- méně témat, více do hloubky
  - šablony, generické programování
  - metaprogramování, spouštění kódu za kompilace
  - koncepty, užitečné idiomy
- forma seminářů + domácích úkolů
  - jako cvičení z PB161
- kolokvium, závěrečný programovací test
- 3 + 1 kredity

*(v semestru podzim 2024 se nevypisuje)*

## PV294 Advanced C++

- větší šířka záběru, pokrývající velkou část (moderního) C++
- přednášky s demonstračními ukázkami
- zápočet, závěrečný (odpovědníkový) test
- 2 kredity