

SOUČTOVÉ TYPY

PB161 PROGRAMOVÁNÍ V JAZYCE C++

Nikola Beneš

16. dubna 2024

PRŮBĚŽNÁ REKAPITULACE

Syntaxe a sémantika

- základní syntaxe, příkazy, výrazy
- hodnotová sémantika, reference (l/r-hodnotové), `const`
- životní cyklus objektu
- kopírování a přesouvání, konstruktory, destruktory
- základní typy, typové konverze, `static_cast`
- funkce, generické funkce s `auto`, `decltype`

Složené datové typy

- součinnové typy: ntice (`std::tuple`, `std::pair`), pole (`std::array`)
- kontejnery: `std::vector`, `std::set`, `std::map`
- základní práce s iterátory
- vlastní typy: `struct`, `class`, metody, operátory

Podtypový polymorfismus

- dědičnost
- časná a pozdní vazba
- virtuální metody

Správa zdrojů

- správa paměti: ruční, automatická pomocí chytrých ukazatelů
- správa zdrojů obecně; princip RAII
- výjimky a jejich vztah ke správě zdrojů

Funkční objekty

- objekty s přetíženým operátorem volání ()
- lambda výrazy
- práce s funkčními objekty pomocí generických funkcí

SOUČTOVÉ TYPY



Součtový typ $A + B$

- hodnoty typu jsou buď hodnoty typu A nebo hodnoty typu B
 - s informací o tom, ke kterému typu patří
- *disjunktní* sjednocení

Obecněji: Součtový typ $A_1 + \dots + A_n$

- hodnoty typu jsou index i + hodnota typu A_i

Ještě obecněji: Součtový typ $\Sigma_{i \in I} A_i$

- I může být nekonečná množina

Výčtový typ `enum`, `enum class`

- konečné množství pojmenovaných hodnot

Typ s volitelnou hodnotou `std::optional`

- buď *žádná hodnota* nebo hodnota zadaného typu

Disjunktní sjednocení `std::variant`

- hodnota jednoho z konečně mnoha zadaných typů

Volatelný objekt `std::function`

- hodnotou je (kopírovatelné) cokoli, co se dá volat jako funkce

Cokoli `std::any`

- libovolná hodnota libovolného (kopírovatelného) typu

Klasický **enum** (podobně jako v C)

- zadán výčtem jmen (volitelně i s jejich číselnou hodnotou)
 - **enum** jméno { jméno1, jméno2 = hodnota, ... }
 - jména bez hodnoty: předchozí hodnota + 1
- velikost (**sizeof**) je implementačně závislá
 - *rozsah* je hypotetický celočíselný typ s co nejmenším počtem bitů, který pokrývá všechny deklarované hodnoty
- implicitní konverze na číselné typy / **bool**
- na rozdíl od C *nemáme* implicitní konverzi číslo → **enum**
 - explicitní konverze pomocí **static_cast**
 - konverze mimo *rozsah* je UB
- přístup k pojmenovaným hodnotám *přímo* nebo pomocí **jméno_enumu :: jméno**
 - pojmenované hodnoty jsou efektivně deklarované konstanty (v aktuálním bloku, jmenném prostoru, apod.)

enum s explicitním podkladovým typem (*underlying type*)

- **enum** jméno : typ { ... }
- podkladový typ musí být celočíselný
- *rozsah* je přesně podkladový typ
- jinak funguje stejně jako klasický **enum**

enum class (nebo **struct**)

- s podkladovým typem (jako na předchozím slajdu) nebo bez
- typově bezpečný **enum**
 - tj. žádné implicitní konverze
- explicitní konverze pomocí **static_cast**
- přístup k pojmenovaným hodnotám standardně pouze pomocí **jméno_enumu::jméno**
- (od C++20) možnost použít **using enum jméno_enumu;** pro zpřístupnění položek v aktuálním bloku apod.

Motivace

- vracení hodnoty nebo informace o její neexistenci
- typy bez rozumné implicitní hodnoty
 - nebo příliš drahé na konstrukci
- jiné jazyky: **Maybe** (Haskell), *nullable* (C# aj.)
- možná řešení:
 - `std::tuple<T, bool>` (T musíme vždy zkonstruovat)
 - `std::unique_ptr<T>` (potřebuje dynamickou alokaci)
 - **union** (korektní zacházení s ním je obtížné a náchylné k UB)
- od C++17: `std::optional`

`std::optional<T>` (C++17)

- buď drží hodnotu typu `T` nebo *nic*
 - *nic*: `std::nullopt` nebo konstruktor bez parametrů
- hodnota je uvnitř, tj. bez nutnosti dynamické alokace
- přístup k hodnotě
 - operátory `*`, `->` (nekontrolovaný přístup)
 - metoda `value()` (vyhazuje výjimku)
 - metoda `value_or()` (zadaná implicitní hodnota)
- zjištění přítomnosti hodnoty
 - explicitní konverze na `bool`
 - metoda `has_value()`
- implicitní konverze `T` → `std::optional<T>`
- `std::make_optional<T>(argumenty)` vytvoří `std::optional<T>` předáním argumentů konstruktoru `T`

`std::optional<T>` (C++17)

- je kopírovatelný / přesouvatelný / přiřazovatelný, pokud typ `T` má tutéž vlastnost
- kopie/přesun objektu s žádnou hodnotou (`std::nullopt`) vytvoří objekt s žádnou hodnotou
- jinak deleguje práci na příslušné spec. metody objektu uvnitř
- tj. zejména při přesunu:
 - dojde k přesunu odpovídajícímu typu `T`
 - „vykradený“ objekt typu `std::optional<T>` *má stále hodnotu* ve smyslu `has_value()` (je jí „vykradená“ hodnota typu `T`)

`std::optional<T>` (C++17)

- vytvoření hodnoty přímo uvnitř – `opt.emplace(argumenty)`
 - destruuje předchozí drženou hodnotu
 - vytvoří novou hodnotu uvnitř – předá argumenty konstruktoru T
- porovnejte s `opt = T{ argumenty }`
 - zavolá přesouvací operátor = typu T

Motivace

- typ, jehož hodnoty jsou některého ze zadaných typů
 - s informací o tom, hodnota kterého typu je aktuálně přítomna
 - libovolně (konečně mnoho) typů
- „typově bezpečný **union**“
 - zacházení s **union** je velmi náchylné k chybám (zejména v přítomnosti typů s netriviálními konstruktory / destruktory)
- jiné jazyky: **Either**¹ (Haskell), **enum** (Rust), ...

¹obecněji libovolný nerekurzivní ADT

`std::variant< typ1, typ2, ... >`

- drží hodnotu jednoho z vyjmenovaných typů
 - hodnota je uvnitř, podobně jako u `std::optional`
- bezparametrický konstruktor vytvoří hodnotu *prvního* typu
- zjištění, která hodnota – metoda `index()`
 - volná funkce `std::holds_alternative<T>()`
- přístup k hodnotě – `std::get<X>`
 - podle indexu nebo typu (je-li jednoznačný)
 - vyhazuje výjimku
- `std::get_if<X>` vrátí ukazatel (`nullptr` v případě chyby)
 - bere *ukazatel* na `std::variant`
- `std::visit` předá hodnotu funkčnímu objektu

`std::variant< typ1, typ2, ... >`

- je kopírovatelný / přesouvatelný / přiřazovatelný, pokud všechny typy v jeho deklaraci mají tutéž vlastnost
- kopie/přesun delegují práci na příslušné metody objektu uvnitř
- přiřazení:
 - obě strany stejný typ hodnoty: deleguje se na její operátor =
 - jinak se destruuje vnitřní objekt držžený levou stranou a použije se `emplace`
- `emplace` – podobně jako u `std::optional`, ale je třeba říct, kterou hodnotu konstruuujeme
 - `emplace<index>` nebo `emplace<typ>`

```
std::visit(visitor, variant1, variant2, ...)
```

- prvním parametrem je „návštěvník“ (*visitor*)
 - volatelný objekt, který se dá zavolat se všemi typy uvnitř `std::variantů`, které jsou dalšími parametry
 - v rámci tohoto předmětu: generická lambda nebo vlastní funkční objekt s odpovídajícími přetíženími operátoru ()
- následuje jeden nebo více parametrů typu `std::variant`
- zavolá návštěvníka na objekty uvnitř `std::variantů`
- může vrátet hodnotu
 - `std::visit(...)` – všechny možnosti volání musí mít stejný návratový typ (často také nějaký `std::variant`)
 - `std::visit<typ>(…)` – provede konverzi na `typ`

Motivace

- k práci s funkčními objekty / lambdaami většinou stačí generické funkce (příp. šablony – mimo záběr předmětu)
- někdy ale potřebujeme funkční objekt někam uložit
 - položka třídy (např. pro zpětné volání – *callback*)
 - položka kontejneru (např. seznam zájemců o notifikaci)
- chceme *součtový typ* přes všechny typy volatelných objektů

```
std::function<typR(typ1, typ2, ...)>
```

- může držet libovolný *kopírovatelný*² volatelný objekt
 - ukazatele na volné funkce
 - lambda (nezachytávají-li nekopírovatelné hodnoty)
 - funkční objekty (nemají-li zakázané kopírování)
 - (a některé další typy – mimo záběr předmětu)
- může být prázdná (test pomocí konverze na **bool**)
- „malé“ objekty může držet „uvnitř“, ale obecně musí používat dynamickou alokaci
- **std::function** můžeme kopírovat, přesouvat, přiřazovat

²v C++23 bude **std::move_only_function** pro *libovolné* volatelné objekty

`std::function<typR(typ1, typ2, ...)>`

- volání skrze `std::function` je typicky pomalejší
 - není možnost inliningu
 - typická implementace – „vymazání typu“ (*type erasure*): virtuální metody nebo jiný podobný mechanismus
- používejte jen tehdy, nemáte-li jinou možnost
 - preferujte `auto` (generické funkce)

Motivace

- nutnost předat / uchovat libovolnou hodnotu bez znalosti možných typů
 - komunikace skrze neprůhledné (*opaque*) rozhraní
 - extra informace pro callback
 - rozhraní pro skriptovací jazyky
 - ...
- „typově bezpečný **void***“ s hodnotovou sémantikou
 - včetně kopírování a zaručené destrukce
- chceme *součtový typ* přes všechny hodnotové typy

std::any

- může držet libovolný *kopírovatelný* objekt
- může být prázdné
- „malé“ objekty může držet uvnitř, ale obecně musí používat dynamickou alokaci
 - typické implementace opět pomocí *type erasure*
- **std::any** můžeme kopírovat, přesouvat, přiřazovat
- **std::make_any<T>(argumenty)** vytvoří hodnotu **std::any** přímo voláním konstrukturu **T** se zadanými argumenty

`std::any_cast`

- přístup k objektu uvnitř `std::any`
- verze s hodnotou `std::any_cast<typ>(objekt_any)`
 - vrací kopii / přesun objektu uvnitř (podle toho, je-li `objekt_any` pomíjivý)
 - v případě neúspěchu vyhazuje výjimku
- verze s referencí `std::any_cast<typ&>(objekt_any)`
 - vrací referenci na objekt uvnitř
 - v případě neúspěchu vyhazuje výjimku
- verze s ukazatelem `std::any_cast<typ>(&objekt_any)`
 - bere ukazatel na `objekt_any`
 - vrací ukazatel na objekt uvnitř
 - v případě neúspěchu vrací `nullptr`
 - *preferujte* pro situace, kdy si nejste jisti obsahem