

# SLOŽENÉ TYPY A HODNOTY

PB161 PROGRAMOVÁNÍ V JAZYCE C++

---

Nikola Beneš

27. února 2024

## Jmenné prostory

- umožňují lepší koexistenci různých knihoven
- mimo záběr předmětu

## Jmenný prostor standardní knihovny `std`

- všechny typy, objekty, funkce standardní knihovny začínají `std::`

Musíme všude psát `std::`?

## DIREKTIVA `using namespace` A DEKLARACE `using`

```
using std::cout;
// odteď můžeme psát jen cout
// ...
using namespace std;
// odteď můžeme psát všechno bez std::
```

- platnost do konce aktuálního bloku
  - pokud není uvnitř bloku, nekončí nikdy!

### Doporučení a pravidla

- preferujte `using std::jmeno;` před `using namespace std;`
- **nikdy** nepište `using namespace std;` nebo `using std::jmeno;` do *hlavičkového souboru*, pokud to není uvnitř bloku (funkce)

## SOUČINOVÉ TYPY

---

II

## Součinnový typ $A \times B$

- hodnoty typu mají dvě složky
  - první je typu  $A$
  - druhá je typu  $B$

## Obecněji: Součinnový typ $A_1 \times \dots \times A_n$

- hodnoty typu mají  $n$  složek
  - $i$ . složka je typu  $A_i$

## Součinnové typy v C++

- bez pojmenování složek: `std::tuple`
- s pojmenováním složek (záznam): `struct`
- všechny složky stejného typu: `std::array`

`std::tuple< typy složek ... >`

- přístup k jednotlivým položkám `std::get<INDEX>(t)`
  - `INDEX` je číselný literál (0, 1, 2, ...)
  - výsledkem je *l-hodnota*, je-li `t` *l-hodnota* (tedy do ní můžeme přiřadit)

`std::pair< typ1, typ2 >`

- dvojice, chová se jako `std::tuple` se dvěma položkami
- dědictví staršího C++;  
občas v návratových hodnotách knihovních funkcí
- (navíc přístup k položkám pomocí `p.first`, `p.second`)

## Inicializace

- deklarace bez inicializace: všechny složky jsou inicializovány
  - jednoduché typy na „nulu“ (0, false, 0.0, ...)
  - složené typy jako = {} (uvidíme za chvíli)

- deklarace s inicializací:

```
std::tuple< typy ... > jméno = { hodnoty ... };
```

- hodnoty musí odpovídat typům (i počtem)

- deklarace s dedukcí typů složek:

```
std::tuple jméno = { hodnoty ... };
```

- inicializace kopií:

```
std::tuple< ... > jméno = jiná_ntice;
```

- nebo použijte **auto**

- podobně pro **std::pair**
- = před { se dá vynechat

## Přiřazení

- `ntice1 = ntice2;` kopíruje jednotlivé položky
  - jako posloupnost přiřazení jednotlivých položek

## Porovnávání

- `==`, `!=` po položkách
- `<`, `<=`, `>`, `>=` – lexikograficky
- funguje i pro ntice různých typů (ale stejné velikosti)



## Rozbalení do existujících proměnných – `std::tie`

- obecněji libovolných *l-hodnot*
- `std::tie(proměnné ...)` = `ntice`;
- `std::tie` vytváří *ntici referencí*
- ignorování některých položek pomocí `std::ignore`

## Rozbalení s novými jmény – *structured binding*

- `auto` [jména ...] = `ntice`; (kopie)
- `auto&` [jména ...] = `ntice`; (reference)
- `const auto&` [jména ...] = `ntice`; (ref. jen pro čtení)
- není možné specifikovat konkrétní typ

## Klíčové slovo **struct**

- vytvoření vlastního datového typu
  - záznam (ntice s pojmenovanými položkami)
  - (může mít i metody apod.; uvidíme příště)
- syntax: **struct** jméno { deklarace položek ... };
  - definuje záznamový typ jméno
  - deklarace mohou mít i inicializaci
  - přístup k položkám pomocí . a jména položky<sup>1</sup>

```
struct point {  
    double x, y;  
    int colour = -1;  
};  
point p1;  
p1.x = 1.0;
```

---

<sup>1</sup>je-li p l-hodnota, pak i p.x je l-hodnota

## Inicializace

- deklarace bez inicializace: **neinicializované položky**
  - pokud nemají inicializaci v definici typu
  - nebo pokud nejsou typu, který se vždy inicializuje (**std::tuple**, kontejnery – uvidíme za chvíli, ...)
- deklarace s inicializací: **typ jméno = { hodnoty ... };**
- od C++20 s pojmenováním položek **.položka = hodnota**
  - možno vynechat ty inicializované v definici
- inicializace = {}: inicializace položek „na nulu“ / jako = {}
  - pokud nemají inicializaci v definici typu
- inicializace kopií: **typ jméno = hodnota stejného typu**
  - nebo použijte **auto**

## Přiřazení

- kopie jednotlivých položek

## Rozbalení (structured binding)

- funguje i pro **struct**
- pořadí dané deklarací

`std::tie` pro **struct** nefunguje

Porovnávání pomocí `==`, `<`, apod.

- pro **struct** samo o sobě nefunguje
- ale umíme to vyřešit (příští přednáška)

`std::array< typ, počet položek >`

- počet položek musí být konstantní výraz (např. literál)
- typ musí být hodnotový (tj. ne reference)
- přístup k položkám pomocí [`index`]
  - *reference* na odpovídající položku
  - `index` mimo rozsah pole – nedefinované chování
- přístup k položkám s kontrolou mezí – `.at(index)`
  - `index` mimo rozsah pole způsobí vyhození výjimky
  - pomalejší

*Poznámka:* Máme i pole ve stylu C, ale preferujeme `std::array`.

## Inicializace

- deklarace bez inicializace: **neinicializované položky**
  - pokud nejsou typu, který se vždy inicializuje

- deklarace s inicializací:

```
std::array< typ, počet > jméno = { hodnoty ... };
```

- vynechané hodnoty inicializovány „na nulu“ / jako = { }
  - zejména = { } takto inicializuje všechny položky
- deklarace s dedukcí typu a počtu položek:  

```
std::array jméno = { hodnoty ... };
```
  - inicializace kopií: `typ jméno = hodnota stejného typu`
    - nebo použijte **auto**

## Přiřazení

- kopie jednotlivých položek

## Rozbalení (structured binding)

- funguje i pro `std::array`

`std::tie` pro `std::array` nefunguje

## Porovnávání pomocí `==`, `<`, apod.

- funguje, lexikografické

Cyklus **for** po prvcích (*range-based*)

- **for** (typ jméno : výraz) příkaz
- výraz se musí vyhodnotit na něco *iterovatelného*
- hodnoty typu **std::array** jsou iterovatelné
  - (další uvidíme za chvíli)
- od C++20 můžeme přidat inicializaci

**for** (inicializace; typ jméno : výraz)

- hodnotový **typ**: vytváříme kopie položek
- **reference**: odkazujeme se na položky přímo
- **const** **reference**: jako reference, ale nesmíme položky měnit



```
typ{ hodnoty pro inicializaci ... }
```

- výraz, který znamená dočasnou hodnotu
- vznikne jako inicializací `typ tajné_jméno = { ... }`, ale
  - *není l-hodnota*
  - přestane existovat na konci příkazu<sup>2</sup>
- **typ** můžeme vynechat, je-li odvoditelný z kontextu
  - parametr ne-generické funkce
  - návratová hodnota funkce

---

<sup>2</sup>s nějakými výjimkami – více o životních cyklech hodnot později

## Použití `const`

- `const` u jednotlivé položky znamená, že je jen pro čtení
  - `std::tuple<const int, double>`
  - `struct typ { const int i; double d; };`
- `const` u celého typu znamená neměnnost všech položek
  - jen „plytká“ – můžeme měnit odkazované hodnoty referencí (a ukazatelů, uvidíme později)

## Alias pomocí deklarace `using`

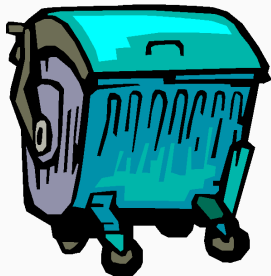
- jako `typedef` (možná znáte z C), ale s lepší syntaxí

```
using point = std::tuple<int, int>;
using circle = std::tuple<point, double>;
```

```
circle c = { { 0, 0 }, 3.14 };
```

# KONTEJNERY

---



## Sekvenční

- `std::vector` – dynamické pole (může se zvětšovat)
- (`std::deque`, `std::list`, `std::forward_list`)

## Asociativní

- `std::set` – uspořádaná množina
- `std::map` – slovník, uspořádaný dle klíče
- varianty s opakováním: `std::multiset`, `std::multimap`
- (hashovací varianty: `unordered`)

## Adaptéry

- `std::stack` – zásobník (LIFO)
- `std::queue` – fronta (FIFO)
- (`std::priority_queue`)

## Iterátory

- reprezentují pozici v kontejneru nebo za *koncem kontejneru*
- používány ve funkcích/metodách standardní knihovny
  - některé návratové hodnoty
  - občas některé parametry
- je-li `it` iterátor, který není za *konec*, pak `*it` je reference na odpovídající prvek kontejneru
- dvojice iterátorů často znamená *polouzavřený interval* [*od*, *do*)
- chceme-li uložit iterátor do proměnné, hodí se **auto**

*více o iterátorech později (v páté přednášce)*

## Společné operace

- inicializace, přiřazení, porovnávání pomocí `==` a `!=`
  - většinou i lexikografické porovnávání `<` apod.
- `empty()`, `size()`, `clear()`
- `swap()` – prohození obsahu s kontejnerem stejného typu
- `begin()` – iterátor na první prvek kontejneru
- `end()` – iterátor za *konec* kontejneru

## Typické další operace

- přístup ke konkrétnímu prvku
  - `front()`, `back()`, indexování (`[]`), `at()`
- metody pro přidávání / odebírání prvků
- asociativní kontejnery: metody pro hledání

viz tabulku dole na <https://en.cppreference.com/w/cpp/container>

`std::vector< typ položek >`

- dynamické (zvětšující se) pole
- konstantní přidávání<sup>3</sup> / odebírání prvků na konci
- pro většinu běžných použití jako seznam položek stejného typu
- má velikost a kapacitu; dosáhne-li velikost kapacity, další přidání prvku znamená realokaci a přesun položek

## Inicializace

- deklarace bez inicializace: prázdné pole (velikost 0, kapacita 0)
- deklarace s inicializací, s dedukcí typu – jako `std::array`

---

<sup>3</sup>amortizovaně konstantní



## Základní operace

- indexování: `[]` (bez kontroly mezí), `at()` (s kontrolou mezí)
- `push_back(hodnota)` přidání prvku na konec
  - varianta `emplace_back( parametry ... )` vytvoří hodnotu až uvnitř vektoru (inicializací pomocí parametrů)
- `pop_back()` odebrání z konce
  - `front()` / `back()` – reference na první / poslední prvek
- procházení po prvcích pomocí příkazu **for**

Další: <https://en.cppreference.com/w/cpp/container/vector>

- pozor na rozdíl mezi `resize()` a `reserve()`
  - to první mění velikost (nové prvky inicializovány „na nulu“)
  - to druhé mění kapacitu (typicky nepotřebujeme)

`std::vector<bool>`

- historický relikv, mnozí jej považují za omyl
- na rozdíl od všech ostatních instancí `std::vector` smí být implementován speciálně (typicky po bitech)
  - může porušovat některé vlastnosti kontejnerů
- indexování / dereference iterátoru může vracet speciální objekt (konvertovatelný na `bool`)
  - nemusí být možné mít referenci (`bool&`) na prvek

*Poznámka:* standardní knihovna obsahuje i statický bitový vektor (tj. fixní velikosti) – `std::bitset`.

## Modifikace kontejneru, přes který se iteruje

- nebezpečná; potenciálně nedefinované chování
- proč? modifikace kontejneru může způsobit, že existující iterátory / reference na prvky uvnitř přestanou být platnými
  - u `std::vector` při realokaci, vkládání / mazání uprostřed
- uvnitř `for` cyklu po prvcích je schovaný iterátor
  
- další důsledky: **nedoporučujeme** dlouhodobé ukládání iterátorů / referencí na prvky kontejnerů
  
- pro podrobnosti čtěte dokumentaci
  - popis metod kontejnerů specifikuje, co se zneplatní
- přehledová tabulka na <https://en.cppreference.com/w/cpp/container>

`std::stack< typ > / std::queue< typ >`

- kontejnerové adaptéry (delegují operace na kontejner „uvnitř“)
  - implicitně `std::deque`, ale dá se změnit
- stejně pojmenované operace `push()`, `pop()`
  - LIFO pro zásobník, FIFO pro frontu
  - varianta `emplace()` – podobně jako u `std::vector`
- reference na vrchol zásobníku: `top()`
- reference na začátek / konec fronty: `front()` / `back()`

`std::set< typ >`

- prvky se neopakují
- prvky jsou uspořádané operátorem <
- prvky jsou nemodifikovatelné
  - reference na prvky jsou vždy `const`
- vkládání, mazání, hledání v  $\mathcal{O}(\log n)$ 
  - (typická implementace: červenočerné stromy)

## Inicializace

- podobně jako `std::vector`
- zadané prvky se seřadí a odstraní se duplikace

## Základní operace

- vložení: `insert` vrací `std::pair<iterator, bool>`
  - iterátor na vložený (nebo už existující) prvek
  - `true`, pokud byl prvek vložen
  - varianta `emplace` (podobně jako u `std::vector`)
- mazání: `erase`
  - různé varianty (hodnota, iterátor, rozsah iterátorů)
  - varianta s hodnotou vrací počet smazaných prvků (0 nebo 1)
  - varianty s iterátory vrací iterátor na další prvek
- hledání:
  - `contains` vrací `bool`
  - `find` vrací iterátor na prvek nebo `end()`

Další: <https://en.cppreference.com/w/cpp/container/set>

`std::map< typ klíčů, typ hodnot >`

- mapuje klíče na hodnoty, uspořádání podle klíčů (<)
- klíče se nemohou opakovat, jsou neměnné
  - typ položek je `std::pair< const klíč, hodnota >`
- funguje podobně jako `std::set` (typicky stejná implementace)
  - vkládáme celé dvojice
  - hledáme, mažeme podle klíčů
- máme-li iterátor `it` na položku, pak
  - `it->first`<sup>4</sup> je klíč
  - `it->second` je hodnota
  - nebo použijte rozbalení `auto& [k, v] = *it;`

---

<sup>4</sup>`a->b` je ekvivalentní `(*a).b`

## Vkládání

- `insert` má jako parametr dvojici (klíč, hodnota)
  - pokud už klíč existuje, hodnota se nezmění
- `emplace` bere klíč, hodnotu jako dva parametry
- `insert_or_assign` má dva parametry jako `emplace`, mění hodnotu existujícímu klíči
- `try_emplace` má jako první parametr klíč, ostatní parametry se použijí k vytvoření hodnoty uvnitř slovníku



## Operátor []

- „indexování“ klíčem
- pokud položka s klíčem neexistuje, *automaticky se vytvoří*
  - s inicializací „na nulu“ / jako = {}
  - proto není možné operátor [] použít, je-li slovník konstantní

```
wormhole[{ 0, 0 }] = { 1234, 5678 };
```

```
if (wormhole[{ 0, 7 }] == std::tuple{ 42, 17 }) {  
    std::cout << "Yes\n";  
}
```

**Metoda at** – při neexistujícím klíči selže (vyhodí výjimku)

- správné řešení v tomto případě je ale použití **find**

## `std::multiset` / `std::multimap`

- fungují podobně jako `std::set` / `std::map`, ale dovolují opakování prvků / klíčů
- `find` může najít libovolný vyhovující prvek
- `equal_range` najde rozsah všech vyhovujících prvků
  - vrací dvojici iterátorů  
(na první prvek rozsahu, za poslední prvek rozsahu)

`std::ranges::subrange`

- pohled (*view*) na rozsah prvků uvnitř kontejneru
- inicializován dvojicí iterátorů
  - na první prvek rozsahu
  - za poslední prvek rozsahu
- použití s cyklem `for` po prvcích

```
for (auto [from, to] = m.equal_range('x');
      auto& [k, v] :
          std::ranges::subrange{ from, to }) {
    v += 10;
}
```

„ochutnávka“ z knihovny `ranges`, více ve třetím bloku

*I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.* (Linus Torvalds)