# NPRG 041 – cvičení 4
# Programování v C++

Jiří Klepl

**mail**:
klepl@d3s.mff.cuni.cz

**mattermost**:
https://ulita.ms.mff.cuni.cz/mattermost/ar2324zs/channels/nprg041-cpp-klepl

# Feedback k úloze BST

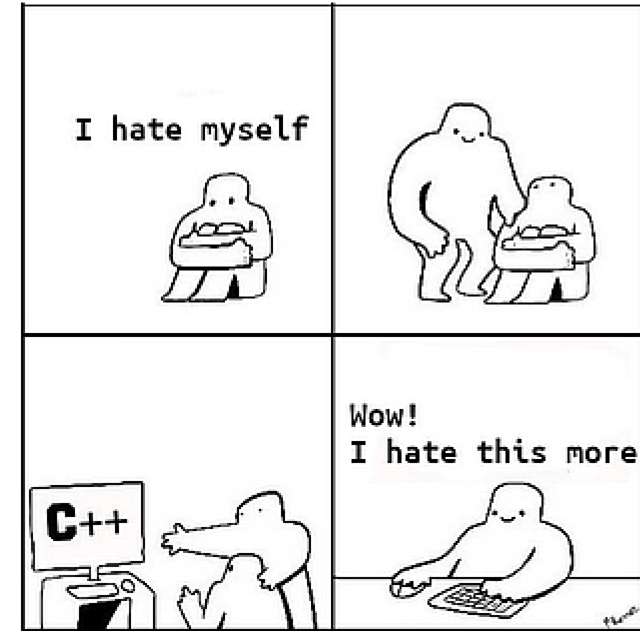
TODO

# Prostor na otázky z přednášky

# Agenda

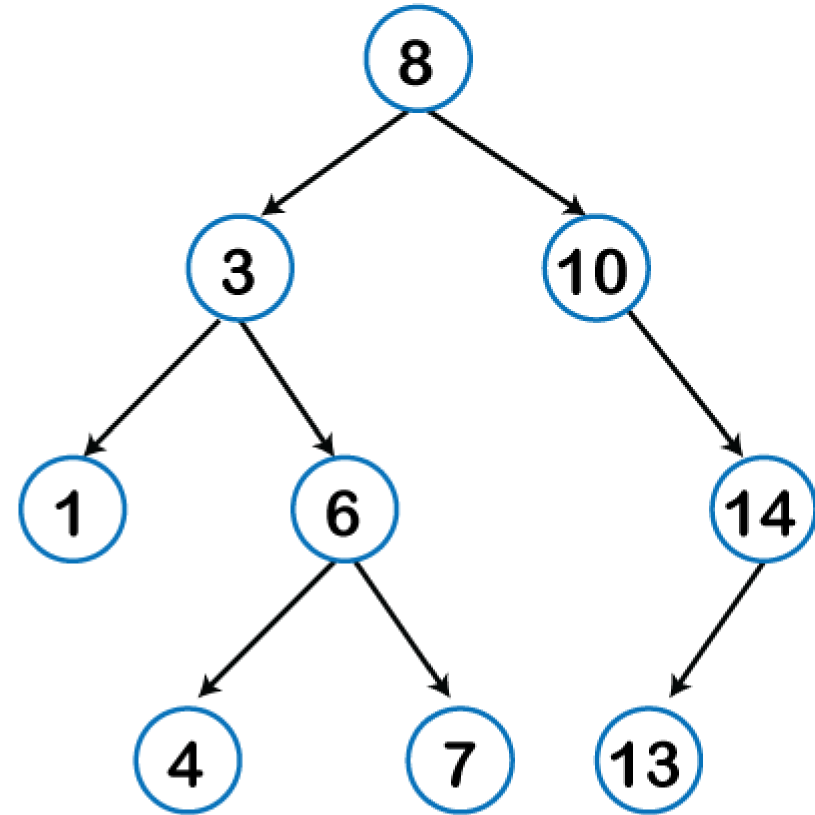
- **Shrnutí úvodních témat**

- **Třetí úloha: Práce s kontejnery**
  - Úvod do kontejnerů
  - Sekvenční kontejnery
  - Asociativní kontejnery



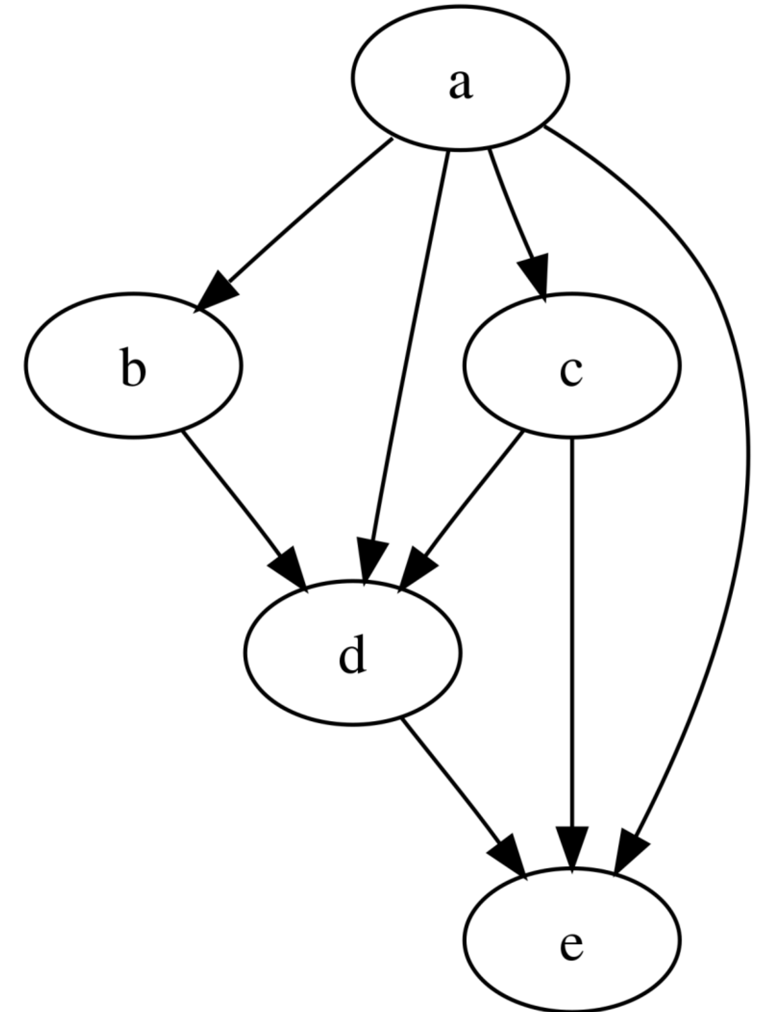
# Typický use-case pro unique_ptr: stromy

- Viděli jsme např. v úloze BST
- Nedovolí kopírování
  - Žádné zdvojování větví!
- Jednoduché přelinkovávání
- Děti umřou s rodičem
  - Snadno smažeme (pod)strom



# Typický use-case pro shared_ptr: DAGy

- Kde najdeme DAGy? Příklady:
  - Grafy dependencí
  - Rozdělení výrazu na podvýrazy
    - U dosazování hodně pomohou, mějme:
      x = 3a + b
      můžeme dosadit do:
      x^2 + 3x
      -> (3a + b)^2 + 3 (3a + b)
      máme dva podvýrazy (3a + b)
- Počítají si počet referencí
  - Dealokování, až když objekt nikdo nepotřebuje



# std::unique_ptr<T[]> a std::shared_ptr<T[]>

- Specializace pro **statické** pole (jako std::array<T, N>, ale velikost určena v run-time)
- Vytvoření: **std::make_unique<T[]>(n)** a **std::make_shared<T[]>(n)**
- Samy vytvoří pole n objektů (volají jejich konstruktory jako new T[n])
  - Samy je pak zničí (jako delete[])
- **! Pozor:** nepamatují si délku
- Málokdy užitečnější než **std::vector<T>** - ten navíc umí push_back
  - Protože často stejně musíme znát délku
- Podporují operátor indexace []

# Shrnutí úvodních témat – hlavní body

- Stringy
  - C-stringy – char*, C++-stringy – std::string
  - Práce s chary: isdigit, isalpha, isalnum, …
- Streamy
  - Input streamy (podporují >>) a output streamy (podporují <<)
  - fstreamy, stringstreamy
- RAII – resource(vlastnictví) representován objektem, zahození => úklid
  - std::move – povolení k převzetí vlastnictví, std::swap
- Pointery: raw pointery a smart pointery

# Vlastníci a jejich (chytré) reference

- Jeden objekt:
  - **T obj** – na stacku (preferováno)
    - auto obj = výraz
  - **unique_ptr<T>** ptr – na haldě
  - **shared_ptr<T>** ptr – shared, halda
- Více objektů:
  - **std::pair** (dva objekty)
  - **std::tuple** (libovolný počet objektů)
- Statické pole:
  - Na stacku: **std::array<T, N>**
  - Na haldě (ale neví délku – pozor):
    std::unique_ptr<T[]>
    std::shared_ptr<T[]>
- Dynamické pole: **std::vector<T>**
- Řetězec: **std::string**

- Reference na objekt:
  - **T& obj** – read+write reference
  - **const T& obj** – read-only ref.
  - **T&& obj** – ref. na temporary (nebo move)
  - **auto&& obj** – automatická reference
  - **T* ptr** – observer pointer na obj
- Reference na hodnoty v pair/tuple:
  - **auto&& [v1, …, vn] = pair/tuple**
- Reference na pole (pro všechny):
  - **std::span<T>, std::span<const T>**

- Reference na řetězec:
  **std::string_view**(std::string from)
  **std::string_view**(const char *from)
  **std::string_view**(c. char *from, size_t len)

# Undefined behavior (UB)

“**there are no restrictions on the behavior of the program**… the compiled program is not required to do anything meaningful”

- Největší zlo v C a C++; typické příklady UB:
  - Přístup k prvkům mimo rozsah pole (outside-bounds memory access)
  - Přetečení *signed* integeru
  - nullptr dereference, read after delete, double delete, …
- Překladač může předpokládat, že UB nenastane -> **optimalizace**
  - vector[idx] nikdy není mimo rozsah
  - for(int i = s; i < e; ++i) a[i] = a * i + b; se spočítá bez přetečení
  - Dereferencovaný pointer nebude nikdy null, …
- Jak je najdeme?
  - Někdy třeba segmentation fault (např. null dereference) a puštěním v debuggeru
  - Jindy: nástroje na statickou analýzu, simulátory (valgrind), sanitizéry, …

```
int foo(int x) { return x + 1 > x; /* either true or UB */ }
```

Možný překlad

```
foo(int):
 mov eax, 1
 ret
```

# Nástroje na hledání bugů v kódu hledání nejen UB

Pokročilejší nástroje

- **Warningy**: -Wall -Wextra -pedantic (gcc, clang) nebo /W4 (msvc)
- **Debugger** – máme vestavěný v IDE, nebo např. gdb (na Linuxu)
- Valgrind
  - Simulátor, nefunguje jen na debugging, ale umí i profiling cache atd.
  - Pustíme v něm neupravený program (valgrind ./binárka) s debug flagy (např. zkompilovaný v debug módu nebo stačí –g flag v gcc a clangu)
  - Program může být typicky 50x pomalejší
- Sanitizéry
  - Vkládají se do programu při překladu: -fsanitize=… (gcc), /fsanitize=… (msvc)
  - Při běhu kontrolují různé věci (např address sanitizer kontroluje přístupy mimo rozsahy polí)
  - Program spustíme úplně normálně: ./binárka

Compiler pozná chybu:

Compiler nepozná chybu:

Memory sanitizer pozná chybu

Ukázka, kdy address sanitizer objeví UB:
správný kód, kód s UB, a verze, kde je UB opraveno

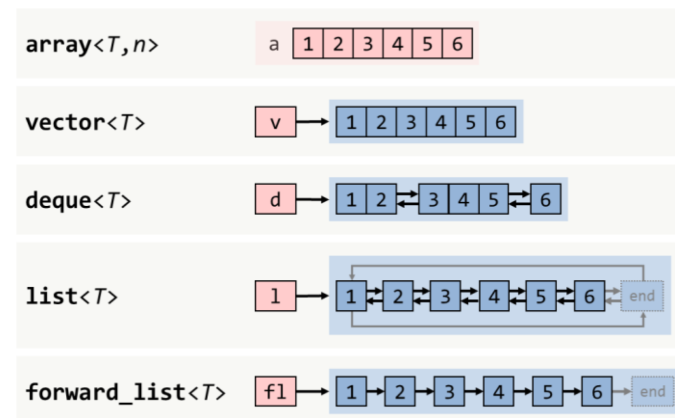
# Dnešní úloha: Práce s kontejnery

- 4 zdrojové soubory (pořadí podle témat) a jeden CMakeLists.txt:
  - https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/containers/containers.cpp
  - https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/containers/sequential.cpp
  - https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/containers/multimap.cpp
  - https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/containers/map_translations.cpp
  - https://www.ksi.mff.cuni.cz/teaching/nprg041-klepl-web/data/sources/containers/CMakeLists.txt
- Všechny zdrojáky mají drobná TODO na procvičení přímo v kódu

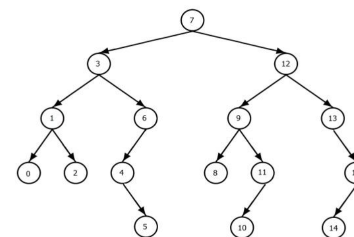
# Odevzdání Úlohy Práce s kontejnery

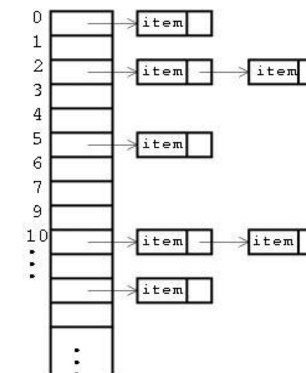
- Soubory úlohy dejte do složky „containers“ (nebo labs-4) v adresáři „labs“ vašeho repozitáře
  - Necommitujte binární soubory ani žádné jiné vygenerované při sestavování
  - Úloha předpokládá 4 zdrojové soubory, ale součástí mohou být projektové soubory Visual Studia, CMakeLists.txt a nebo README.md
- Doma nepovinně dále experimentujte s kontejnery

# Kontejnery




- **Sekvenční kontejnery**
  - **`vector`** – pole prvků z přidáváním na konec
  - **`deque`** – pole s přidáváním na oba konce
  - **`list`**, **`forward_list`** – obousměrně/jednosměrně vázaný seznam
  - **`array`** – pole pevné velikosti, data žijí přímo v objektu (u VŠECH ostatních na haldě)
- **Asociativní kontejnery**
  - **Setříděné** – podle operátoru <
    - **`set/multiset`** – množina (multi: s opakováním)
    - **`map/multimap`** – asociativní pole
  - **Nesetříděné** – používají hashování, operátor ==
    - **`unordered_set/map/multiset/multimap`**



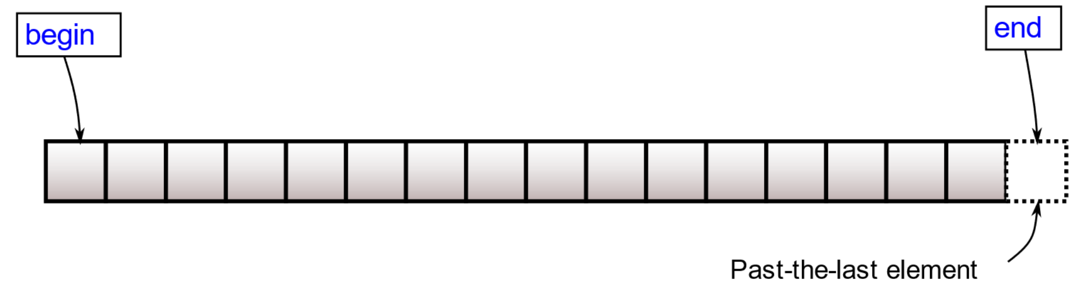


# Konstrukce s std::initializer_list

- Už jsme viděli s vectorem, ale použitelné u kontejnerů obecně
- Vynutit voláním konstruktoru se složenými závorkami, např.: std::vector{values…}
- Vytvoření kontejneru přesně s určenými prvky:

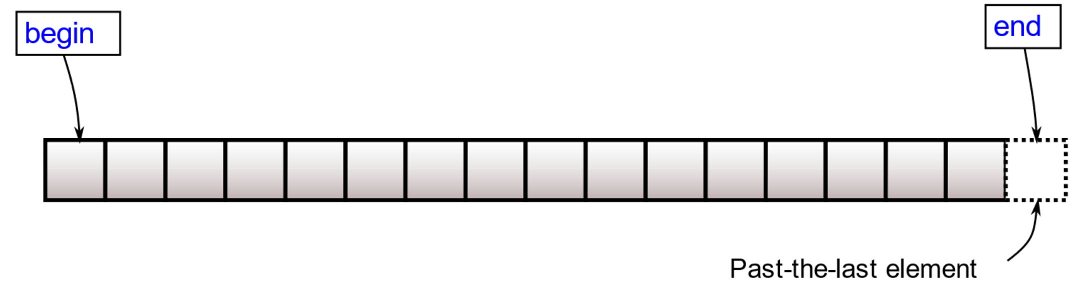
```
std::vector<std::string>{"hi", "hello", "howdy", "ciao"};
std::map<std::string, int>{{"hi", 0}, {"hello", 1}, {"ciao", 2}};
```

# Iterátory



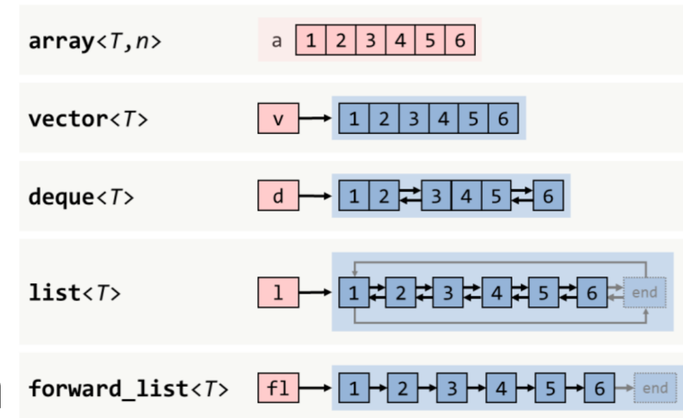

- Typické použití: **for (auto it = kontejner.begin(); it != kontejner.end(); ++it)**
- Odkaz na prvek v kontejneru (u map: **std::pair<key, value>**)
- Typ: **Kontejner<T>::iterator**, **Kontejner<T>::const_iterator**
  - **Proč dva různé typy iterátorů?** – ze stejného důvodu jako const T* ≢ T* const
    - **const Kontejner<T>::iterator je neiterovatelný iterátor na modifiable hodnotu**
    - **Kontejner<T>::const_iterator je iterátor na readonly hodnotu**
- U včech kontejnerů k: **k.begin(), k.end(), k.cbegin(), k.cend()**
  - (c)end odkazuje ZA kontejner (neplatný prvek, proto ve for smyčce: **it != k.end()**)
- Chovají se jako pointery:
  - Inkrementace: **++it** ukazuje na další prvek, **it += 2** přeskočí následující prvek
  - *it – získání prvku; it->field, it->metoda() – získání fieldu a volání metody
  - Pozor: it++ (post-increment) vrátí kopii iterátoru, ne it

# Procházení kontejnerů (všechny kontejnery)



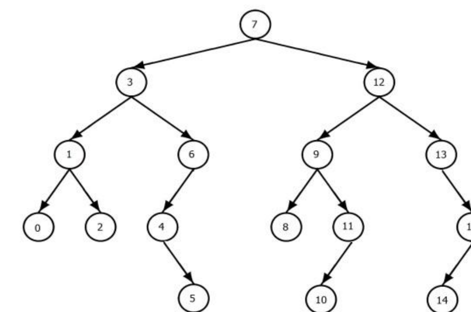

- Pomocí iterátorů (Kontejner::iterator a Kontejner::const_iterator):
  - Odkazované prvky jsou const právě když kontejner je const
    **for (auto it = k.begin(); it != k.end(); ++it) *it = new_value;**
  - Pokud chceme vždy const hodnotu (**když jen čteme**):
    **for (auto it = k.cbegin(); it != k.cend(); ++it) std::cout << *it;**
  - Máme varianty s **(c)rbegin** a **(c)rend**: procházení pozpátku (a opět: (c) = const)
- Pomocí range-based for:
  - Pro sekvenční a sety (jakékoliv) typicky: **for (auto&& value : kontejner)**
  - Pro mapy (jakékoliv) typicky: **for (auto&& [key, value] : map)**

# Sekvenční kontejnery



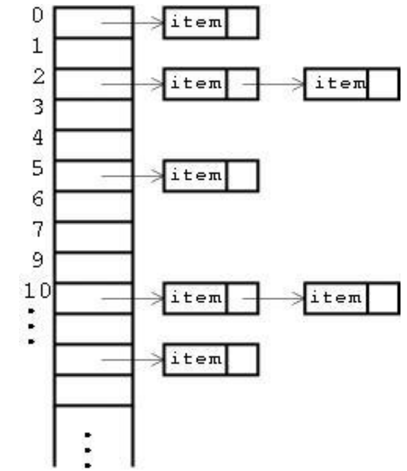

- **vector** – souvislé pole prvků na haldě s přidáváním zprava
  - Podpora indexace
  - Přidávání prvků může způsobit relokaci – zneplatní všechny reference
- **deque** – double-ended queue, podporuje přidávání z obou stran
  - Podobně efektivní jako vector, ale prvky nemusí být souvisle za sebou
    - Přidávání nikdy neinvaliduje reference
- **forward_list**, **list**
  - Zachovávají umístění prvků (dokud není smazaný, reference na něj je platná)
  - Nepodporuje indexaci přes [], ale podporuje vkládání doprostřed
- **array** – pole pevné velikosti, data jsou součástí arraye (žádná indirekce)
- **basic_string** – string, wstring

# Asociativní kontejnery - setříděné




- Třídění podle operátoru < (pro vlastní třídy je potřeba dodefinovat)
- **set<V>** - množina
- **multiset<V>** - množina s opakováním
- **map<K, V>** - asociativní pole
    - Hledání/čtení prvku přes **it = find(K)** – vrátí **map.end(), pokud prvek nenalezne**
    - **it = lower_bound(K)**, **it = upper_bound(K)** – vrátí iterátor na první klíč >= K, resp. > K
        - K čemu to je? Klíč by mohl být třeba floating point číslo, pak se přesná shoda hledá blbě
- **multimap<K, V>** - relace s rychlým hledáním podle K
    - Opět **lower_bound(K), upper_bound(K)**
    - Také **equal_range(K)** – vrátí dvojici iterátorů [začátek, konec) reprezentující všechny hodnoty s ekvivalentním klíčem (rovné nebo neporovnatelné)
- Všechny hodnoty (multi)map uloženy v **std::pair<K, V>** s fieldy **.first** a **.second**
    - Ale často bývá pohodlnější: **for (auto&& [k, v] : map)** nebo **auto&& [k, v] = *it**;

# Asociativní kontejnery - nesetříděné




- **std::unordered_(multi)set**, **std::unordered_(multi)map**
- Opět hledání/čtení přes **it = find(K)**, porovnání s **.end()**
- Implementace přes hash tabulku – do binů podle jejich hashe
  - U vlastních tříd je potřeba (dobře) dodefinovat:
    std::size_t std::hash<X>(const X&)
    - Například u struct X { std::string name, surname; } není vhodný hash:
      std::hash(name) + std::hash(surname) ani std::hash(name) ^ std::hash(surname)
      - CppReference radí kombinovat hashe alespoň přes hash1 ^ (hash2 << 1)
    - Špatná hash funkce znamená pomalý lookup
  - Pro naše potřeby není nutné umět správně hashovat, ale dávejte si pozor
- Ekvivalence prvků podle operátoru ==
  - Opět, u vlastních tříd potřeba dodefinovat

# Jednotné API (ne každý kontejner splňuje vše) Ty nejdůležitější funkce

- Vkládání
  - push_back, push_front — přidání na konec/začátek, podporuje move
  - emplace_back, emplace, try_emplace — vytvoření na místě (najde místo, vytvoří prvek)
  - insert(V), insert(V, it) — vložení prvku (před prvek it)
  - insert(pair{K, V}) — vložení do mapy
- Přístup
  - front(), back() — reference na prvek na začátku/konci
  - operator[idx], .at(idx) — indexace (at: s kontrolou + výjimkou)
  - find(V), lower_bound, upper_bound — hledání prvku (přesné; první alespoň, větší)
- Další
  - size(), empty() — velikost, neprázdnost
  - pop_back(), pop_front() — odebrání
  - erase(it) — odstranění prvku
  - clear() — vymazání všech prvků

***Další: [https://en.cppreference.com/w/cpp/container](https://en.cppreference.com/w/cpp/container)***

| *složitost* | *přidání / odebrání na začátku* | *přidání / odebrání na i-té pozici* | *přidání / odebrání m prvků* | *přidání / odebrání na konci* | *přístup k i-tému prvku* |
|---|---|---|---|---|---|
| `funkce` | `push_front` `pop_front` | `insert` `erase` | `insert` `erase` | `push_back` `pop_back` | `begin()+i` `[i],at(i)` |
| **list** | konst | konst | m, konst přesuny mezi sezn. (splice) | konst | neex |
| **deque** | konst | min( i, n - i) | m + min( i, n - i) | konst | konst |
| **vector** | neex | n - i | m + n - i | konst (*) | konst |
| **asociativní** | ln | (s klicem k) ln | (s klicem k) ln + m | ln | nalezení podle hodnoty ln |
| **unsorted** | konst | konst | m | konst | konst |

fyzická velikost: capacity() ↭ reserve()
logická obsazenost: size() ↭ resize()

(*) při překročení kapacity rozšíření a kopie stávajících prvků

# Insertování (insert, emplace, emplace_back , emplace_front, try_emplace)

1. Varianty s _front a _back vracejí referenci na přidaný prvek
2. Jinak, u sekvenčních a multiset/multimap vracejí iterátor na přidaný prvek – protože vždy uspějí (oproti následujícím)
3. U asociativních bez *multi* vracejí std::pair<iterátor, bool> (prvek stejného klíče už existuje -> žádný prvek se nepřidá)
   - .first je iterátor na (přidaný/nalezený) prvek
   - .second řekne, jestli byl prvek skutečně přidaný (nebo jsme našli už existující)
   - Pokud prvek už existoval, tak ta metoda funguje prakticky jako find
   - Jde použít se structured bindingem: **auto [it, success] = map.emplace(…)**

# Pozor na .find(value) -> .emplace(value)

**Špatně (2x hledáme)**

```
auto it = map.find(5);
if (it != map.end()) { // already present
    // do something
} else {
    map.emplace(5, 1);
}
```

**Dobře**

```
auto [it, success] = map.emplace(5, 1);
if (!success) { // already present
    // do something
}
```

# emplace_back, emplace, try_emplace

- **Vytvoření prvku na místě** – bere argumenty konstruktoru objektu
  - Něco podobného už jsme viděli u std::make_unique<Object>(args…)
- **Někdy mohou být efektivnější** než ekvivalentní push/insert
  - Vytvoření objektu tam, kde má být, namísto toho, aby se kopíroval/přesouval
  - Můžeme emplacovat přímo předvyrobený prvek – pak se chová jako insert
    - V copy variantě i move variantě (nezapomeňte na std::move)
- **U map pozor** – emplace vyrábí std::pair<Key, Value>
  - **try_emplace** je u map **hezčí a bezpečnější** – **striktně odděluje klíče a hodnoty** – takže třeba nezahodí omylem std::unique_ptr, který se snažíme přidat do mapy jako hodnotu

# Pozor na operator[] u map

- U vectorů, když prvek neexistuje, tak vector[idx] je UB (nedefinované chování, compiler má právo dělat co chce)
  - Pokud to hrozí, můžeme použít .at(idx) – často nechceme <- málo efektivní
- U map, když prvek neexistuje, tak si ho mapa vymyslí
  - Pak má defaultní hodnotu pro ten daný typ (třeba 0)
- Nelze používat na const mapě
  - Proč? Viz výše
- Často chceme nahradit (try_)emplacem nebo findem
  - Ale: **map[key] = val** je kratší než **map.try_emplace(key).first->second = val**