NPRG041 – C++

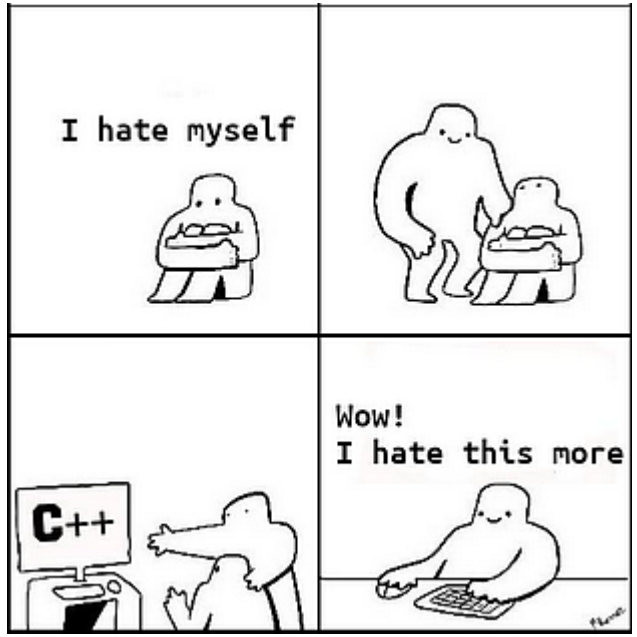
cvičení – Jiří Klepl

mattermost: ulita/2425ZS: nprg041-cpp-klepl (inv na SIS nástěnce)

Klepl@d3s.mff.cuni.cz

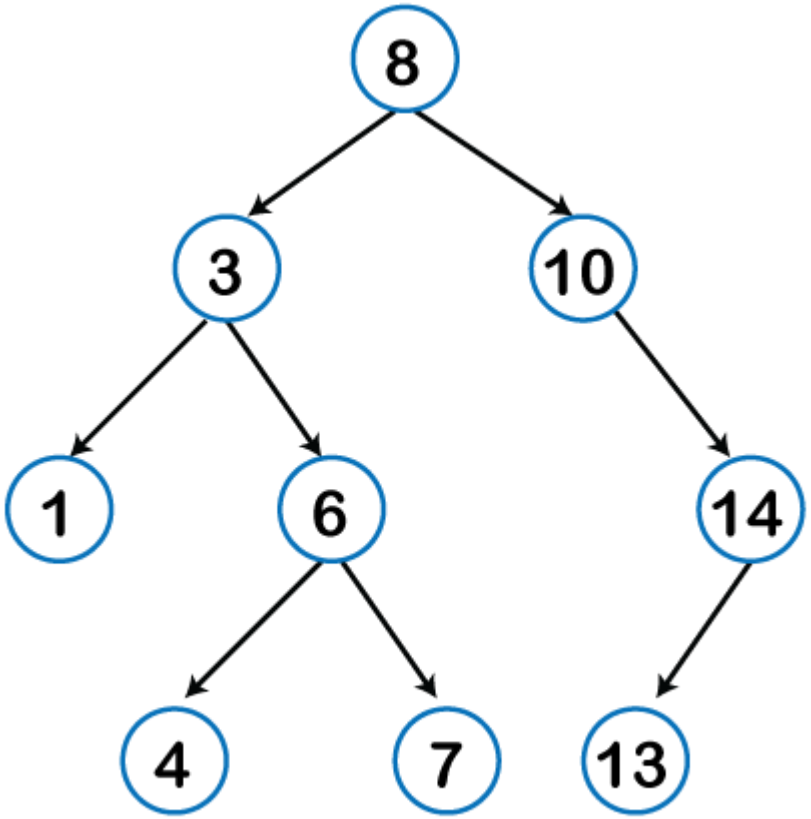
Agenda

- Modularita, objekty
- Vlastnictní a RAII
- Pointery a reference
  - Raw pointery
  - std::unique_ptr
  - std::shared_ptr
- std::move



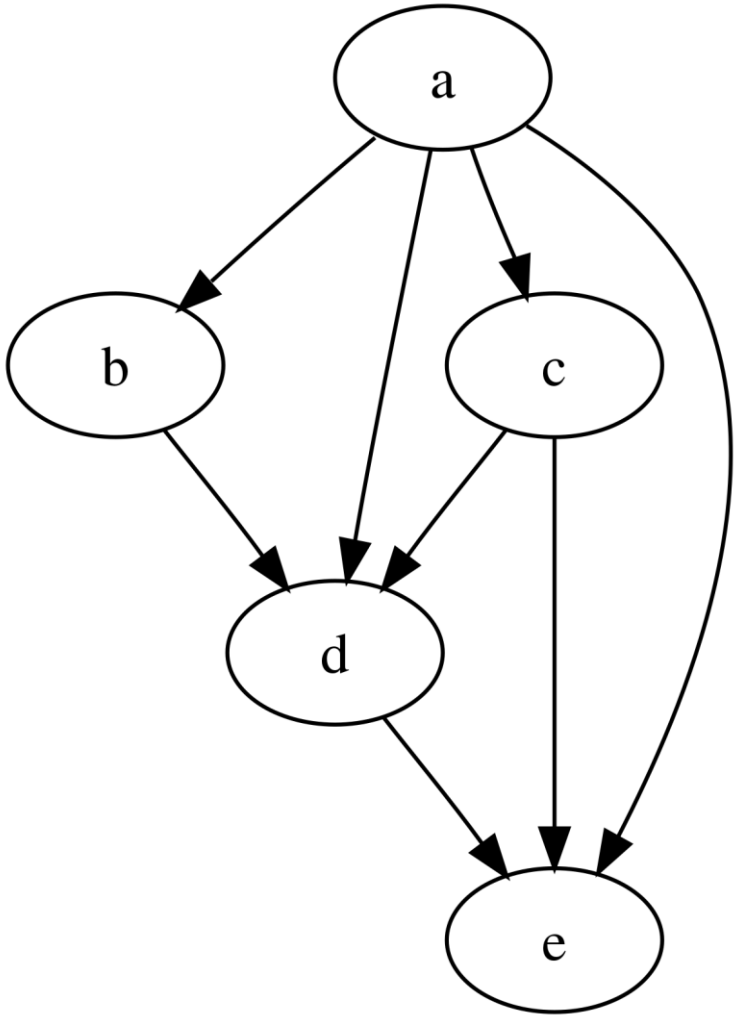
Typický use-case pro unique_ptr: stromy

- 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[]>`

- Pointer na pole konstantní délky
- Vytvoření:
  - `std::make_unique<T[]>(n)`
  - `std::make_shared<T[]>(n)`
- Automatická alokace + dealokace
- indexace prvku pomocí []
- ⚠️ nepamatují si délku
- ⚠️ Málokdy užitečnější než `std::vector<T>`

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

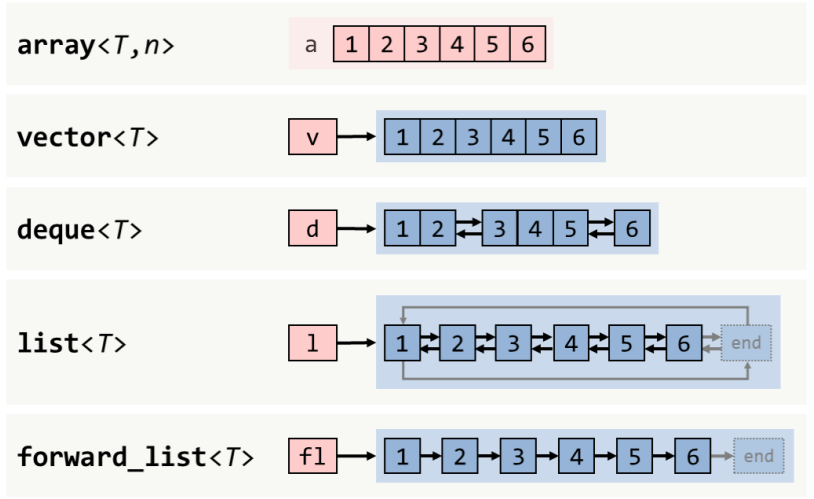
Vlastníci a jejich (chytré) reference

- 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**

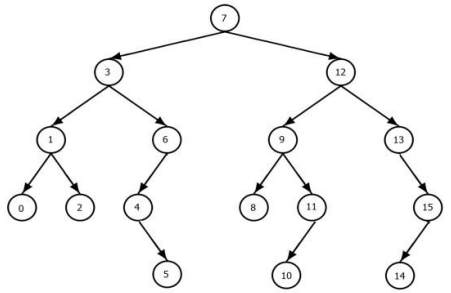
- Reference na objekt:
  - **T& obj** – read+write reference
  - **const T& obj** – read-only ref.
  - **T&& obj** – ref. na temporary (nebo move)
  - **auto&& obj** – automatická reference
  - **T* ptr** – observer pointer na obj
- Reference na hodnoty v pair/tuple:
  - **auto&& [v1, …, vn] = pair/tuple**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

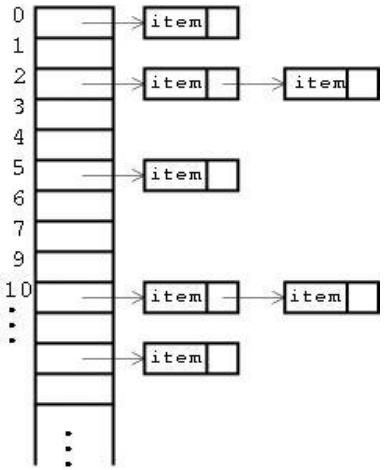
- Reference na řetězec:
  std::string_view(std::string from)
  std::string_view(const char *from)
  std::string_view(c. char *from, size_t len)

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
- **Asociativní kontejnery**
  - **Setříděné** – podle operátoru <
    - **`set/multiset`** – množina (multi: s opakováním)
    - **`map/multimap`** – asociativní pole
  - **Nesetříděné** – používají hashování, operátor ==
    - **`unordered_set/map/multiset/multimap`**



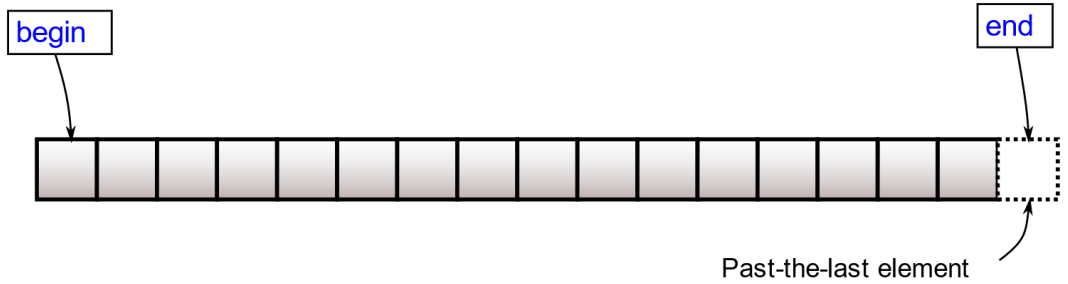


Iterátory

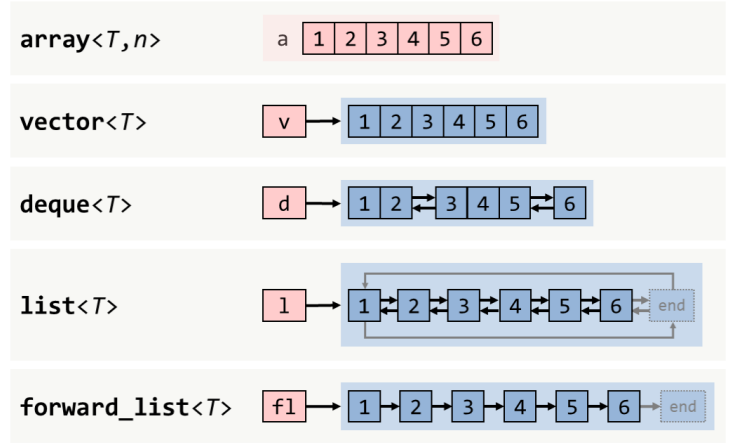
- Odkaz na prvek v kontejneru (nebo imaginární `.end() na konci`)
- Výroba typicky container.begin(), container.end()
  - Pro read-only verzi container.cbegin(), container.cend()
  - Existuje pak ještě obrácené procházení container.rbegin(), container.rend()
- Jako pointery
- Typ definován v kontejneru
  - ::iterator
  - ::const_iterator

```
std::vector<std::string> animals{
    "dog",
    "cat",
    "bat",
    "rat",
    "aligator"
};

for (auto it = animals.cbegin(); it != animals.cend(); ++it)
{
    std::cout << *it << '\n';
}
```



Sekvenční kontejnery



- **vector** – souvislé pole na haldě, v.push_back(item)
  - Indexace v[0] až v[v.size()-1]
  - Přidávání prvků může způsobit relokaci – ⚠️ **zneplatní všechny odkazy**
- **deque** – double-ended queue, podporuje přidávání z obou stran
  - Podobné jako vector, ale prvky nemusí být souvisle za sebou

    ℹ️ **Přidávání nikdy neinvaliduje (nezneplatňuje) reference**
- **forward_list**, **list** – prostě linked list

  ℹ️ nikdy neinvalidují reference na nesmazané prvky
  - Nepodporuje indexaci přes [], ale podporuje vkládání/mazání uprostřed
- **array** – pole pevné velikosti, data jsou součástí arraye (žádná indirekce)
- **basic_string** – string, wstring

Minipříklad

1. Načíst čísla ze stdin (oddělená whitespacy, konec: EOF)
2. Čísla na sudých pozicích odzadu ([last], [last – 2], ...) vynásobit 2
3. Vypsat čísla na pozicích i takových, že i % 3 == 0 ([0], [3], ...)

ℹ️ použijte sekvenční operátor, iterátory vždycky posouvejte **dopředu**

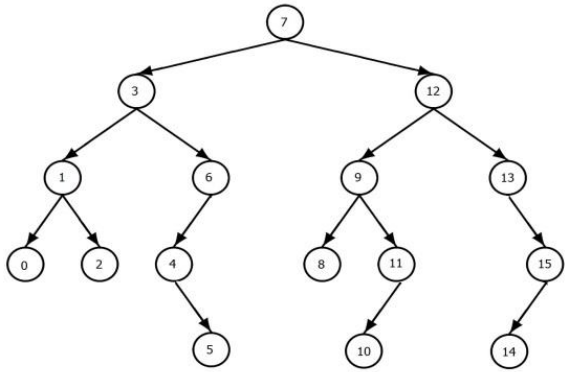
(používejte výrazy jako **++it** nebo **it += 3**; vector::iterator umí **<**)

⚠️ trénujeme správné použití: **begin**, **end**, **cbegin**, **cend**, **rbegin**, **rend**

☠️ je jednoduché udělat chybu

Asociativní kontejnery - setříděné

- Třídění podle **operátoru <** (pro vlastní třídy dodefinovat)
- **set<V>** - množina
- **multiset<V>** - množina s opakováním
- **map<K, V>** - asociativní pole
- V mapách jsou prvky uloženy jako **std::pair<K, V>**
  - **K key = item.first**; nebo **K key = it->first**
  - **V value = item.second**; nebo **V value = it->second**

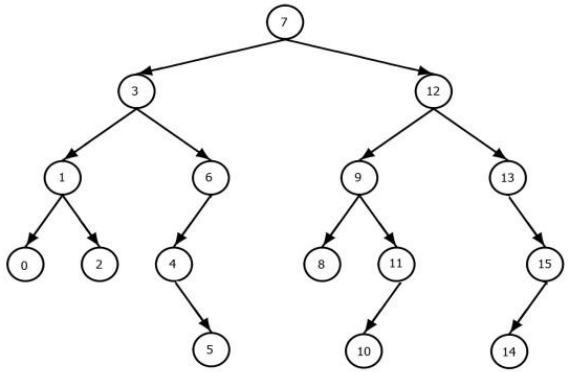



```
std::map<std::string, std::size_t>
animals{{"dog", 3},
        {"cat", 7},
        {"bat", 0},
        {"rat", 100'000},
        {"aligator", 1}};
```

```
for (auto it = animals.begin();
          it != animals.end(); ++it) {
    const std::size_t& count = it->second;
    const std::string& name = it->first;
    std::println("There are {} {}s", count, name);
}
```

Asociativní kontejnery - setříděné

- Třídění podle **operátoru <** (pro vlastní třídy dodefinovat)
- **set<V>** - množina
- **multiset<V>** - množina s opakováním
- **map<K, V>** - asociativní pole
- V mapách jsou prvky uloženy jako **std::pair<K, V>**
  - **K key = item.first**; nebo **K key = it->first**
  - **V value = item.second**; nebo **V value = it->second**

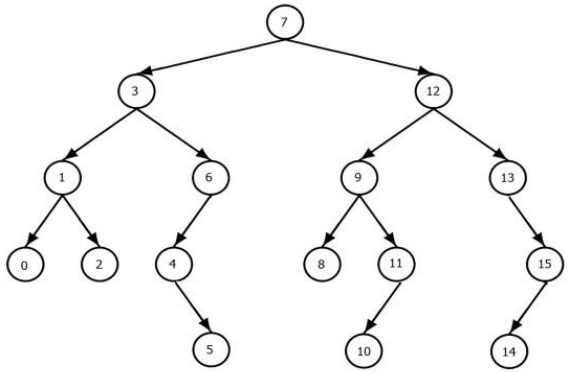



```
std::map<std::string, std::size_t>
animals{{"dog", 3},
        {"cat", 7},
        {"bat", 0},
        {"rat", 100'000},
        {"aligator", 1}};
```

```
for (auto it = animals.begin();
          it != animals.end(); ++it) {
    // structured binding (jde i auto/auto&&)
    const auto& [name, count] = *it;
    std::println("There are {} {}s", count, name);
}
```

Asociativní kontejnery - setříděné

- Třídění podle **operátoru <** (pro vlastní třídy dodefinovat)
- **set<V>** - množina
- **multiset<V>** - množina s opakováním
- **map<K, V>** - asociativní pole
- V mapách jsou prvky uloženy jako **std::pair<K, V>**
  - **K key = item.first**; nebo **K key = it->first**
  - **V value = item.second**; nebo **V value = it->second**



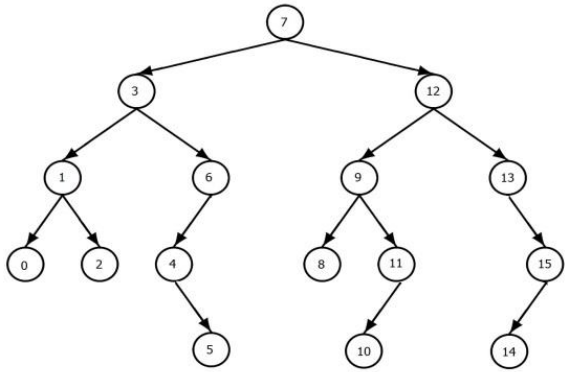

```
std::map<std::string, std::size_t>
animals{{"dog", 3},
        {"cat", 7},
        {"bat", 0},
        {"rat", 100'000},
        {"aligator", 1}};
```

```
for (auto it = animals.begin();
          it != animals.end(); ++it) {
    // structured binding (jde i auto/auto&&)
    const auto& [name, count] = *it;
    std::println("There are {} {}s", count, name);
```

const auto& je **const std::pair<K, V>&**

Asociativní kontejnery - setříděné

- Třídění podle **operator<** (pro vlastní třídy dodefinovat)
- **set<V>** - množina
- **multiset<V>** - množina s opakováním
- **map<K, V>** - asociativní pole
- V mapách jsou prvky uloženy jako **std::pair<K, V>**
  - **K key = item.first**; nebo **K key = it->first**
  - **V value = item.second**; nebo **V value = it->second**




```
std::map<std::string, std::size_t>
animals{{"dog", 3},
        {"cat", 7},
        {"bat", 0},
        {"rat", 100'000},
        {"aligator", 1}};
```

```
// structured binding, range-based for
for (const auto& [name, count] : animals) {
    std::println("There are {} {}s", count, name);
}
```

Hledání shody v asociativních kontejnerech

```
auto it = animals.find(name);

// kontrola: našli jsme prvek?
if (it != animals.end()) {
    std::size_t count = it->second;
    std::println("There are {} {}s", count, name);
} else {
    std::println(R"(Unknown: "{}")", name);
}
```

Raw string

Najdi první item s klíčem >= name

```
auto it = animals.lower_bound(name);
```

Najdi první item s klíčem > name

```
auto it = animals.upper_bound(name);
```

Najdi itemy s klíči == name

```
auto [start, end] = animals.equal_range(name);
```

Hledání shody v asociativních kontejnerech

```cpp
auto it = animals.find(name);

// kontrola: našli jsme prvek?
if (it != animals.end()) {
    std::size_t count = it->second;
    std::println("There are {} {}s", count, name);
} else {
    std::println(R"(Unknown: "{}")", name);
}
```

Raw string

Najdi první item s klíčem >= name

```cpp
auto it = animals.lower_bound(name);
```

Najdi první item s klíčem > name

```cpp
auto it = animals.upper_bound(name);
```

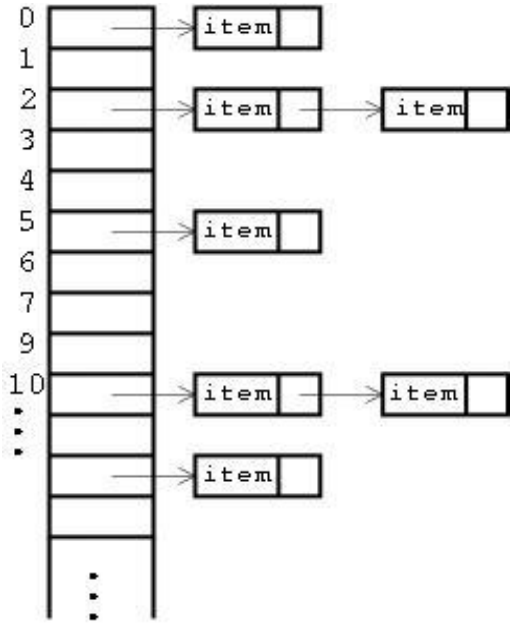
Najdi itemy s klíči == name

```cpp
auto [start, end] = animals.equal_range(name);
```

Není to prostě find???
NENÍ (click here)

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 hashe klíče
  - ℹ️ **std::size_t std::hash<K>(const K&)**
- Ekvivalence prvků podle **operator==**
  - 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, nepráznost
  - pop_back(), pop_front() — odebrání
  - erase(it) — odstranění prvku
  - clear() — vymazání všech prvků

Další: ***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

Konstruktory a vkládání do kontejneru

```
class MyClass {
  MyClass( X x, Y y);
  MyClass(const  MyClass& mc);
  MyClass(MyClass&& mc) noexcept;
  MyClass& operator=(const MyClass& mc);
  MyClass& operator=(MyClass&& mc) noexcept;
  ~MyClass();
};
```

typická sada konstruktorů

automaticky generované

```
vector<MyClass> v;
MyClass m{ x, y };
```

push	emplace		
v.push_back(m)	v.emplace_back(m)	(ctor) copy_ctor (dtor)	☹
v.push_back(move(m)) v.push_back(MyClass{x,y})	v.emplace_back(move(m)) v.emplace_back(MyClass{x,y})	ctor, move_ctor, dtor	😐
	v.emplace_back(x,y)	ctor	☺

☹

```
vector<BigClass> v;
sort( v);
```

```
vector<unique_ptr<BigClass>> v;
sort( v);
```

☺

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
    - ⚠️ nezapomeňte na **std::move** pro předcházení kopií
- **U map pozor – emplace** je jako konstruktor **std::pair<Key, Value>**
  - ⚠️ nejdřív vyrobí ten pair, pak ho teprve zkouší přidat
  - ✅ **try_emplace** je u map často **bezpečnější** – vyrábí pair jen, když v mapě není

⚠️ 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
}
```

⚠️ 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**

Překladový slovník (basic) : gitlab: ./labs/dictionary/

k jednomu slovu může být více překladů

- operace
  - přidat slovo a jeho překlad
    akceptovat **více** překladů jednoho slova
  - odebrat jeden překlad slova
  - odebrat všechny překlady slova
  - nalézt všechny překlady slova

- k rozmyšlení
  - kontejner(y) pro ukládání dat
    - efektivita operací
  - jak 'vracet' více slov
    - kontejner hodnot ※ rozmezí

- Rozhraní: třída s metodami
  - API - public metody
    - volání metod s konstantami z 'main'
    - metody třídy nijak nekomunikují s cin/cout
  - až potom parsování cin
    - console UI
    - řádkově orientovaný vstup

```
add(slovo, cizi);
del(slovo, cizi);
del(slovo);
?? find(slovo); // -> cizi cizi cizi
```

```
string r, cmd, arg;
for( ;;) {
  getline( f, r);
  if( f.fail())  break;
  istringstream line(r);
  line >> cmd >> arg;
}
```

```
add slovo cizi
find slovo
....
```