Knihovna algoritmů

PB161 Programování v jazyce C++

Nikola Beneš

23. dubna 2024

Knihovna algoritmů

Klasické algoritmy

- založena na principu *generického programování* (šablony)
- pracují s iterátory; typicky:
  - dva iterátory (na začátek, za konec) pro vstup
  - jeden iterátor pro případný výstup

Knihovna *ranges* (C++20)

- založena na tzv. *konceptech*
- obecnější pojem *range* – iterovatelný objekt
- „hezčí“ syntax
- možnosti „líného“ zpracování
- lepší skládání algoritmů bez nutnosti uchovávání mezivýsledků

Klasické algoritmy

Algoritmy

- funkcionální styl programování v C++
- jednotný způsob práce s různými druhy kontejnerů
- používají iterátory
- typicky dva iterátory pro vstup
  - polouzavřené intervaly
  - iterátor na první prvek vstupu
  - iterátor *za* poslední prvek vstupu
- typicky jeden iterátor pro výstup
  - na první prvek místa, kam má přijít výstup
    (je třeba zajistit, že místa je dostatek)
  - nebo speciální iterátor, který automaticky vkládá
    (tzv. *inserter*)
- různé algoritmy mohou mít různé požadavky na druh iterátorů
  - nebo fungovat různě efektivně podle druhu

Hledání

find.cpp

Algoritmy `std::find`, `std::find_if`

- první dva parametry – iterátory
- třetí parametr
  - hledaná hodnota (`std::find`)
  - predikát (`std::find_if`)

```
auto it1 = std::find(letters.begin(),
                     letters.end(), 'q');

auto it2 = std::find_if(
    letters.begin(), letters.end(),
    [](unsigned char c) {
        return std::isupper(c);
    });
```

Hledání

Co je tady špatně?

```
std::set s = { … };

auto it = std::find(s.begin(), s.end(), something);
```

- `std::find` vždy lineárně prochází kontejner
  - používá pouze rozhraní iterátorů
  - neví nic o detailech implementace kontejneru
- metody asociativních kontejnerů jsou efektivnější
  - preferujte je *vždy*

Doing linear scans over an associative array is like trying to club someone to death with a loaded Uzi. *(Larry Wall)*

sort.cpp

Algoritmus `std::sort`

```cpp
std::vector vec1{ 17, 42, 1, 3, 3, 7, 0 };
std::sort(vec1.begin(), vec1.end());

std::vector vec2{ 17, 42, 1, 3, 3, 7, 0 };
std::sort(vec2.begin() + 1, vec2.end() - 2);

auto it = std::find(vec2.begin(), vec2.end(), 42);
std::sort(it, vec2.end());
```

sort.cpp

Porovnávání

- implicitně dle operátoru <
- můžeme dodat vlastní funkční objekt (funkci, lambdu, ...)

```
std::sort(vec2.begin(), it,
          [](int x, int y) { return y < x; });

std::sort(vec2.begin(), vec2.end(),
          std::greater{});
```

Stabilita

- `std::sort` nemusí být stabilní (typická impl. *introsort*)
- v knihovně existuje i `std::stable_sort`
- podobně existují `stable_` verze některých jiných algoritmů

Odbočka: Obrácené iterátory

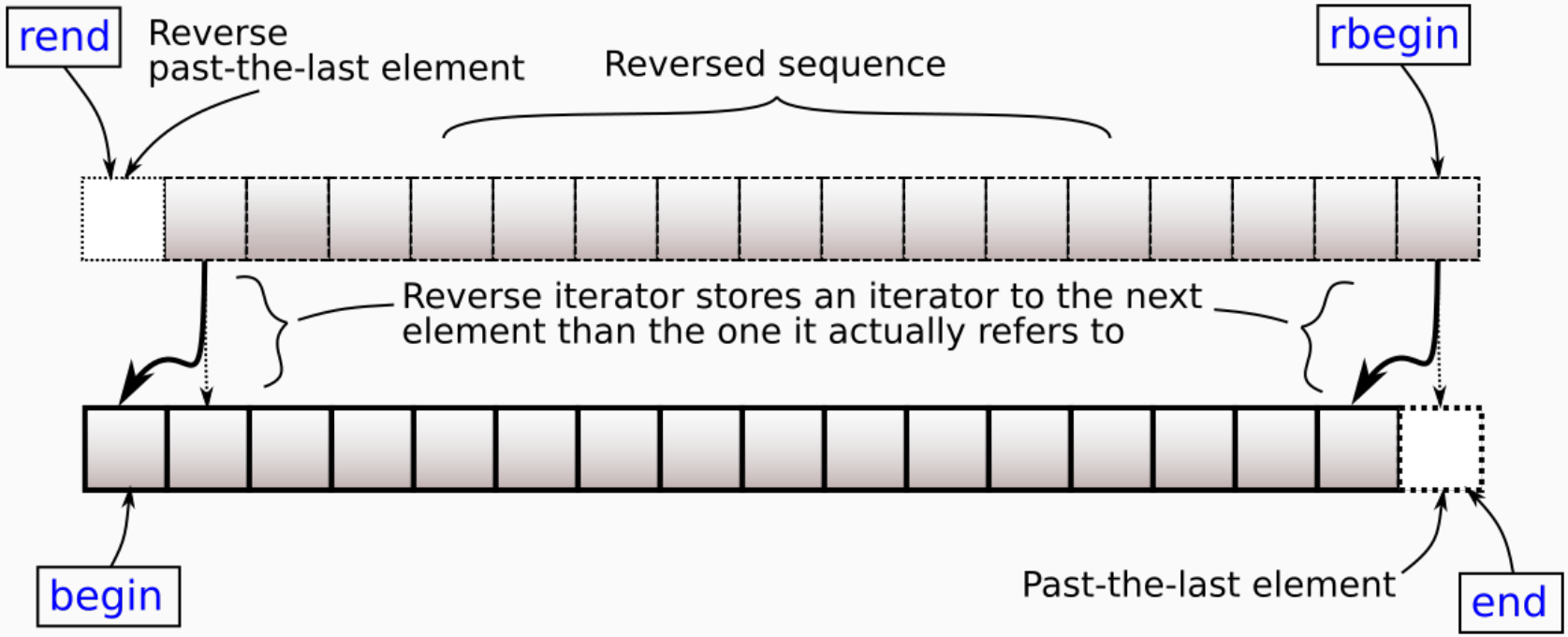
reverse.cpp

`std::reverse_iterator`

- vytváří „obrácený“ iterátor
  - obal (*wrapper*) nad původním iterátorem,
    jehož `++` volá operátor `--` původního iterátoru a naopak
  - ukazuje *o jeden prvek před* původní iterátor
    (zachovává polouzavřené intervaly iterátorů)
- u kontejnerů, jejichž iterátory jsou obousměrné
  - `rbegin()` – obrácený iterátor na konec
  - `rend()` – obrácený iterátor *před* začátek
- příklad použití pro obrácené řazení:
  `std::sort(vec.rbegin(), vec.rend())`

Odbočka: Obrácené iterátory

`std::reverse_iterator`



zdroj: https://en.cppreference.com/w/cpp/iterator/reverse_iterator

copy.cpp

Algoritmus `std::copy`

- zdrojový interval (dva iterátory)
- cílový iterátor – je třeba zajistit dostatek místa

```
std::set s = { 1, 21, -7, 4 };
std::vector<int> vec;
vec.resize(7);
std::copy(s.begin(), s.end(), vec.begin() + 2);
```

Odbočka: Vkládácí iterátory

copy.cpp

`std::inserter`, `std::back_inserter`, …

- užitečné speciální iterátory, které při zápisu volají metody kontejnerů
  - `std::inserter` volá `insert`
  - `std::back_inserter` volá `push_back`

```
std::copy(s.begin(), s.end(),
          std::back_inserter(vec));
```

- pro kopírování do kontejnerů je lepší použít přímo metodu `insert` (s přetížením, které bere interval daný iterátory)

```
vec.insert(vec.end(), s.begin(), s.end());
```

Kopírování

copy.cpp

Algoritmus `std::copy_if`

- jen objekty splňující daný *predikát*

Algoritmus `std::transform`

- kopírování s modifikací
- jeden vstup, unární funkce – jako `map` v Haskellu
- dva vstupy, binární funkce – jako `zipWith` v Haskellu
  - druhý vstup daný jen jedním iterátorem
  - počet prvků určený prvním vstupem

Algoritmus `std::move`

- jako `std::copy`, ale místo kopírování *přesouvá*
  - pomocí přesouvacího přiřazení

Množinové operace

set_ops.cpp

Algoritmy `std::includes`, `std::set_union`, …

- fungují nad *seřazenými posloupnostmi*
  - `std::set`, ale i třeba seřazený `std::vector`
  - implicitně podle `<`, ale můžeme dodat vlastní porovnání
- posloupnosti s unikátními prvky – množinová sémantika
  - podmnožina, sjednocení, průnik, …
- posloupnosti s opakujícími se prvky – multimnožinová
  - podmnožina – podposloupnost (ne nutně souvislá)
  - sjednocení – maximum[1] z výskytů
  - průnik – minimum z výskytů
- prvky se primárně berou z „levé“ posloupnosti
  - (stabilní algoritmy)

[1] chceme-li všechny výskyty, můžeme použít `std::merge`

generate.cpp

Algoritmus `std::fill`

- vyplní zadaný interval zadanou hodnotou

Algoritmus `std::generate`

- vyplní interval výsledky volání zadaného funkčního objektu

Algoritmus `std::iota`

- vyplní interval hodnotami od zadané,
  s postupným inkrementem

Odstranění prvků

remove.cpp

Algoritmy `std::remove`, `std::remove_if`, `std::unique`

- ve skutečnosti *nic nemažou*
- *posunou* (pomocí přesouvacího přiřazení) prvky v zadaném intervalu tak, že ty, co se nemají smazat, zůstanou na začátku
- vrátí iterátor na poslední prvek, co má zůstat
  - prvky za ním jsou *připraveny ke smazání* („vykradené“ přesouváním)
- idiom „erase-remove“
  - nejprve použijeme algoritmus `std::remove` apod.
  - pak zavoláme metodu `erase` kontejneru
  - od C++20 máme `std::erase`, `std::erase_if`, které přesně toto udělají za nás

Algoritmy

... a mnohé další

- https://en.cppreference.com/w/cpp/algorithm
- *in situ* algoritmy mají většinou verzi `_copy`, která místo modifikace původního kontejneru kopíruje
  - `std::remove_copy`, `std::remove_copy_if`, atd.

Jonathan Boccara "105 STL Algorithms in Less Than an Hour"
https://www.youtube.com/watch?v=2olsGf6JIkU (CppCon 2018)

Doporučení pro používání algoritmů

- nemá smysl vynalézat kolo – podívejte se nejprve do knihovny
  - postupem času se naučíte, které algoritmy jsou k dispozici
- zkuste raději použít algoritmus z knihovny než vlastní řešení
  - (pro problémy řešitelné *malým* cyklem `for` to není nutné)
  - menší šance na chyby typu *off-by-one*
  - dost často mohou být efektivnější
  - mnohem jasněji čitelný záměr
    (pro člověka znalého knihovny algoritmů, samozřejmě)
- má-li kontejner metodu, která problém řeší, použijte raději tu
  - typicky optimální řešení pro danou implementaci

Knihovna *ranges*

Motivace

Klasické algoritmy

- plní svou roli, ale ...
- nemají moc pěknou syntax
  - dost často chceme procházet celý kontejner
  - musíme psát `begin()` a `end()`
- neumožňují kompozici
  - mezivýsledky algoritmů je nutno někde uchovat

Knihovna `ranges` (od C++20)

Range – iterovatelný objekt

- kontejner
- interval daný dvěma iterátory, iterátorem a počtem prvků, …
- adaptér iterovatelného objektu, který na jeho prvky *líně* aplikuje nějaký algoritmus
- atd.

Algoritmy používající *ranges*

- ve jmenném prostoru `std::ranges`
- požadavky na iterátory slabší než u klasické knihovny

Knihovna `ranges` (od C++20)

Vytváření nových *ranges* – pohledy (*views*)

- pohled – lehký, snadno přesouvatelný iterovatelný objekt
- typ ve jmenném prostoru `std::ranges` (název končící `_view`)
- funkční objekty ve jmenném prostoru `std::views`
- zejména tzv. *adaptéry*
  - vytvářejí pohledy nad iterovatelné objekty
  - umožňují kompozici pomocí operátoru `|`
    (zleva doprava, jako „roura“ (*pipe*) v shellu)

Odbočka: Alias jmenného prostoru

```
namespace ranges = std::ranges
```

- vytvoří alias `ranges`, který zastupuje `std::ranges`
- např. místo `std::ranges::sort` pak můžeme psát jen `ranges::sort`
- doporučení pro použití podobné jako pro **`using`** (**`namespace`**)
  - nikdy v hlavičkových souborech, pokud to není v bloku, ve vlastním jmenném prostoru apod.

Nové algoritmy

Obecně

- většina klasických algoritmů má verzi v `std::ranges`
- dají se volat
  - klasicky (iterátory jako parametry)
  - s iterovatelným objektem (místo dvou iterátorů)
- některé mají (nepovinný) parametr: projekce
  - funkční objekt, který se aplikuje na každý procházený prvek, než se na něj algoritmus „podívá“ (příklad: řazení dle klíče)
- místo iterátorů některé vracejí iterovatelné objekty
  - typicky zabalená dvojice iterátorů
  - např. `ranges::remove` vrací interval ke smazání

r_sort.cpp

Řazení – `std::ranges::sort`

- dva nepovinné parametry – porovnání, projekce
  - mají implicitní hodnotu, kterou můžeme zapsat jako `{}`
  - porovnávání – `std::less`
  - projekce – `std::identity`

```
ranges::sort(vec);

ranges::sort(vec, std::greater{});

ranges::sort(vec, {},
    [](int num) { return std::abs(num); });
```

Příklady použití

r_set_ops.cpp

Množinové operace – `std::ranges::set_union` apod.

- tři nepovinné parametry
  - porovnání
  - projekce jednotlivých složek

```
ranges::set_union(m1, m2, std::back_inserter(res));

auto fst = [](const auto& p) { return p.first; };

ranges::set_union(m1, m2, std::back_inserter(res),
                  {}, fst, fst);
```

Iterovatelné objekty – pohledy (*views*)

Obecně

- *pohledy* – lehké iterovatelné objekty
  - snadno přesouvatelné
  - nemusí být kopírovatelné, ale pokud jsou, kopie neznamená kopírování prvků
- typ pohledu v `std::ranges`, s názvem končícím `_view`
  - např. `std::ranges::transform_view`
- funkční objekt v `std::views`, název bez `_view`
  - např. `std::views::transform`
  - kompozice pomocí operátoru `|`
- „líné“ verze modifikujících algoritmů
  - vytvoření pohledu nic nemodifikuje
  - modifikace se děje „za běhu“ při použití pohledu
- jiné užitečné adaptéry iterovatelných objektů

Vytváření pohledů

v_create.cpp

`std::ranges::subrange`

- pohled vytvořený ze dvojice iterátorů

`std::views::iota`

- s jedním parametrem – nekonečná posloupnost generovaná `++`
- se dvěma parametry – konečná posloupnost

Adaptéry iterovatelných objektů

v_adapt.cpp

`std::views::take`, `std::views::drop`

- se dvěma parametry: iterovatelný objekt, kolik prvků
  - `take`: kolik prvků ze začátku vzít
  - `drop`: kolik prvků ze začátku ignorovat
- s jedním parametrem: kolik prvků
  - určený pro kompozici

`std::views::take_while`, `std::views::drop_while`

- místo počtu predikát, dokdy pokračovat

Adaptéry iterovatelných objektů

`v_adapt.cpp`

Kompozice adaptérů pomocí operátoru `|`

- `objekt | adaptér` je totéž, co `adaptér(objekt)`
- `objekt | adaptér(parametry…)` je totéž, co `adaptér(objekt, parametry…)`
- můžeme takto řetězit více adaptérů
  - přetížený operátor `|` je asociativní, tedy zejména `adaptér1 | adaptér2` vytvoří nový adaptér

Adaptéry: „Líné“ algoritmy

v_lazy.cpp

`std::views::transform`

- líně modifikuje prvky

`std::views::reverse`

- líně obrací iterovatelný objekt
- (jako použití `std::reverse_iterator`)

`std::views::filter`

- líně filtruje prvky dle zadaného predikátu

Adaptéry: Pohledy na určité složky

v_elements.cpp

`std::views::elements<N>`

- jako `std::get<N>` na každém prvku

`std::views::keys`, `std::views::values`

- aliasy pro `std::views::elements<0>`, `std::views::elements<1>`
- vhodné např. pro použití s `std::map`

Modifikace skrze pohledy

`v_modify.cpp`

Pohledy **nemusí být** jen pro čtení

- závisí na tom, jak vznikly
- pohled do modifikovatelného kontejneru umožňuje měnit prvky
- chceme-li znemožnit modifikaci
  - konverze kontejneru na `const` referenci
  - ideálně pomocí `std::as_const`

Další operace s pohledy

- závisí na kontejneru uvnitř a na typu adaptéru
- `front()`, `back()`, operátor `[]`
- přímý pohled do kontejneru: vrací referenci
- líný algoritmus (např. `transform`): vrací hodnotu

You have all these algorithms at your disposal. Learn them. You'd be surprised at how often a `rotate` *comes up in real code.*

(Sean Parent)

https://www.youtube.com/watch?v=qH6sSOr-yk8

(8:30–18:30)