Kurs języka C++

13. Algorytmy

Spis treści

Zakresy w algorytmach STL

- W pliku nagłówkowym `<algorithm>` zdefiniowanych jest około 100 standardowych algorytmów działających na zakresach definiowanych przez pary iteratorów (dla wejścia) lub pojedyncze iteratory (dla wyjścia).
- Niektóre algorytmy (na przykład `sort()`) wymagają iteratorów o dostępie swobodnym, a inne (na przykład `find()`) przeglądają sekwencje po kolei, więc wystarcza im iterator jednokierunkowy.
- Wiele algorytmów fakt nieodnalezienia elementu standardowo oznacza zwróceniem końca zakresu.

Zakresy w algorytmach STL

- Algorytmy pracują na kolekcjach i na tablicach.
- Argumentami algorytmów STL są zakresy (iteratory w kolekcjach albo wskaźniki w tablicach)
- Po stronie funkcji wywołującej leży obowiązek zapewnienia poprawności zakresów – oznacza to, że początek musi odnosić się do wcześniejszego lub tego samego elementu tego samego kontenera co koniec.
- Algorytmy działają w trybie nadpisywania, a nie wstawiania – funkcja wywołująca musi więc zapewnić, aby zakresy docelowe posiadały odpowiedni rozmiar.

Parametry funkcyjne

- Niektóre algorytmy umożliwiają przekazanie operacji zdefiniowanych przez użytkownika, które są następnie przez nie wewnętrznie wywoływane.
- Operacje te to funktory – mogą być zwykłymi funkcjami lub obiektami funkcyjnymi lub lambdami.
- Funktory służyć mogą do realizacji następujących zadań:
  - predykat jednoargumentowy jako kryterium wyszukiwania lub wybierania elementów;
  - predykat dwuargumentowy jako kryterium sortowania czy wyszukiwania w uporządkowanym zbiorze;
  - funktor aplikowany do wszystkich elementów z podanego zakresu;
  - funktor dla algorytmów numerycznych.

Klasyfikacja algorytmów

- Algorytmy dzielą się na niemodyfikujące (tylko czytające dane) i modyfikujące.
- Przeznaczenie algorytmu można wywnioskować po jego nazwie:
  - Przyrostek/sufiks `_if` używany jest wtedy, gdy istnieją dwie postacie pewnego algorytmu posiadające tę samą liczbę parametrów, lecz jedna wymaga podania wartości (wersja bez przyrostka) a druga funkcji lub obiektu funkcyjnego (wersja z przyrostkiem). Algorytm `find()` na przykład szuka elementu o określonej wartości, podczas gdy algorytm `find_if()` szuka elementu spełniającego podane kryterium.
  - Przyrostek/sufiks `_copy` wskazuje, że elementy podlegają nie tylko manipulacji, lecz również kopiowaniu do zakresu docelowego. Algorytm `reverse()` na przykład odwraca kolejność elementów wewnątrz danego zakresu, podczas gdy algorytm `reverse_copy()` kopiuje elementy w odwrotnej kolejności do innego zakresu.

Algorytmy niemodyfikujące

- Algorytmy niemodyfikujące nie zmieniają ani kolejności, ani wartości przetwarzanych elementów.
- Algorytmy niemodyfikujące współpracują z iteratorami wejściowymi i postępującymi, można je więc wywołać dla wszystkich kontenerów standardowych.

Algorytm `for_each`

- Algorytm `for_each()` wywołuje wobec każdego elementu operację podaną przez funkcję wywołującą.
- Wywołanie:
  for_each(iterator_pocz, iterator_kon, funkcja)
- Algorytm for_each() zwraca obiekt funkcyjny stosowany do elementów kolekcji.
- Przykład 1:

```
void echo(short num) {
    cout << num << endl;
}
…
vector<short> vect;
…
for_each(vect.begin(), vect.end(), echo);
```

Algorytm for_each

- Przykład 2:

```
struct Sum {
    void operator()(int n) { sum += n; }
    int sum{0};
};
…
    std::vector<int> nums{3, 4, 2, 8, 15, 267};
…
    auto print = [](const int& n)
        { cout << n << " "; };
    for_each(nums.cbegin(), nums.cend(), print);
    cout << '\n';
…
    std::for_each(nums.begin(), nums.end(),
        [](int &n){ n++; });
…
    Sum s = std::for_each(nums.begin(), nums.end(), Sum());
```

Algorytmy niemodyfikujące wyszukujące

- Fukcja `find()` znajduje pierwsze wystąpienie zadanej wartości.
- Fukcja `find_end()` znajduje ostatnie wystąpienie zadanego ciągu wartości.
- Fukcja `search()` znajduje pierwsze wystąpienie zadanego ciągu wartości.
- Fukcja `min_element()` znajduje element o najmniejszej wartości.
- Fukcja `max_element()` znajduje element o największej wartości.

Algorytmy niemodyfikujące wyszukujące

- Przykład 1:

```
const int N = 7;
int myints[N] = {3,7,2,5,6,4,9};
…
// using default comparison:
cout << "The smallest element is "
  << *min_element(myints, myints+N) << endl;
cout << "The largest element is "
  << *max_element(myints, myints+N) << endl;
```

Algorytmy niemodyfikujące wyszukujące

- Przykład 2:

```
int n;
cin >> n;
…
std::vector<int> v {0, 1, 2, 3, 4};
…
auto result = find(begin(v), end(v), n);
if (result != end(v))
    cout << "v contains: " << n << '\n';
else
    cout << "v does not contain: " << n << '\n';
```

Algorytmy niemodyfikujące sprawdzające

- Fukcja `count_if()` zlicza wystąpienia zadanej wartości w określonym zakresie.
- Fukcja `equal()` sprawdza czy wartości z podanych zakresów są sobie równe.
- Fukcja `missmatch()` znajduje pierwsze wystąpienie różnicy w podanych ciągach wartości (wynikiem jest para iteratorów).
- Fukcja `is_permutation()` sprawdza czy jeden zakres jest permutacją innego zakresu.
- Fukcja `is_sorted()` sprawdza czy jeden zakres jest posortowany.

Algorytm is_prmutation

```
static constexpr auto v1 = {1,2,3,4,5};
static constexpr auto v2 = {3,5,4,1,2};
static constexpr auto v3 = {3,5,4,1,1};

cout << v2 << " is a permutation of " << v1 << ": " << boolalpha
     << is_permutation(v1.begin(), v1.end(), v2.begin()) << endl
     << v3 << " is a permutation of " << v1 << ": " << boolalpha
     << is_permutation(v1.begin(), v1.end(), v3.begin()) << endl;
```

Algorytmy modyfikujące

- Algorytmy modyfikujące zmieniają wartość elementów. Mogą one bezpośrednio modyfikować elementy z danego zakresu lub modyfikować je podczas kopiowania do innego zakresu.

- Algorytm `for_each()` dopuszcza operację modyfikującą swój argument - zatem argument ten musi być przekazywany przez referencję.
- Przykład:

```
void square (int &elem) { elem *= elem; }
…
for_each(coll.begin(), coll.end(), square);
```

Algorytmy modyfikujące

- Algorytm `transform()` wykorzystuje operację zwracającą modyfikowany argument (wynik operacji można przypisać do pierwotnego elementu).
- Przykład:

```
int square (int elem) { return elem * elem; }
…
transform(coll.begin(), coll.end(),
    coll.begin(), square);
```

- Funkcja `copy()` kopiuje zakres począwszy od pierwszego elementu; funkcja `copy_backward()` kopiuje zakres począwszy od ostatniego elementu.
- Funkcja `move()` przenosi zakres począwszy od pierwszego elementu; funkcja `move_backward()` przenosi zakres począwszy od ostatniego elementu.

Algorytmy modyfikujące

- Funkcja fill() zastępuje każdy element z zadanego zakresu podaną wartością.
- Funkcja replace() zastępuje elementy o określonej wartości z zadanego zakresu inną wartością.
- Funkcja generate() zastępuje każdy element z zadanego zakresu wartością wygenerowaną przez podaną funkcję bezargumentową.
- Funkcja merge() scala dwa zakresy.
- Funkcja swap_ranges() zamienia miejscami elementy z dwóch zakresów.

Algorytmy usuwające

- Algorytmy usuwające są specjalną postacią algorytmów modyfikujących. Mogą one usuwać elementy albo z pojedynczego zakresu, albo przy jednoczesnym kopiowaniu do innego zakresu. Tak jak w przypadku algorytmów modyfikujących, jako kontenera docelowego nie możemy użyć kontenera asocjacyjnego ani nieuporządkowanego.

- Funkcja `remove()` usuwa elementy o podanej wartości.
- Funkcja `remove_if()` usuwa elementy spełniające zadany predykat.
- Funkcja `unique()` usuwa elementy powtarzające się (sąsiednie).

Algorytmy mutujące

- Algorytmy mutujące to algorytmy, które zmieniają kolejność elementów (a nie ich wartości) poprzez operacje przypisania i zamiany ich wartości.

- Funkcja `reverse()` odwraca kolejność elementów.
- Funkcja `rotate()` przesuwa cyklicznie elementy.
- Funkcja `random_shuffle()` losowo zmienia kolejność elementów.
- Funkcja `partition()` dzieli zakres na elementy spełniające predykat (na początku kolekcji) i te niespełniające (na końcu kolekcji – funkcja zwraca iterator do początku drugiego przedziału.

Algorytmy sortujące

- Algorytmy sortujące są specjalnym rodzajem algorytmu mutującego, ponieważ także zmieniają kolejność elementów. Sortowanie jest jednak bardziej skomplikowane niż proste operacje mutujące i zabiera zwykle więcej czasu.

- Funkcja `sort()` sortuje elementy.
- Funkcja `stable_sort()` sortuje elementy w sposób stabilny.

Algorytmy pracujące na posortowanych danych

- Algorytmy przeznaczone dla zakresów posortowanych wymagają, aby zakresy, na których one operują, były posortowane zgodnie z ich kryterium sortowania.

- Funkcja `binary_search()` sprawdza, czy dany zakres zawiera określony element.

Literatura

- [1] B.Stroustrup: C++. Kompendium wiedzy. Wydanie 4. Helion 2013. Rozdział 32: Algorytmy STL.
- [2] N.M.Josuttis: C++. Biblioteka standardowa. Wydanie 2. Helion 2014. Rozdział 11: Algorytmy STL.