C++17 i STL

Algorytmy

Iteratory

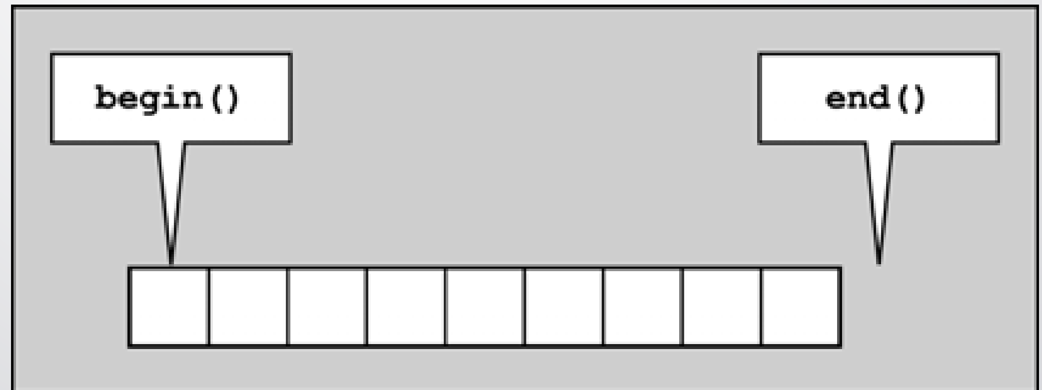
- Iterator to obiekt reprezentujący pozycję elementu w kontenerze.
- Zachowanie iteratora definiują podstawowe operacje:
  - Operator * zwraca element z aktualnej pozycji. Jeśli elementy posiadają składowe, dostęp do nich można uzyskać bezpośrednio z poziomu iteratora za pomocą operatora ->.
  - Operator ++ pozwala iteratorowi przejść do następnego elementu. Większość iteratorów pozwala również na wykonanie kroku wstecz za pomocą operatora --.
  - Operatory == oraz != zwracają wartość logiczną informującą, czy dwa iteratory reprezentują tę samą pozycję.
  - Operator = przypisuje iterator (pozycję elementu, do którego on się odnosi).
- Operatory te są interfejsem zwykłych wskaźników języka C++ wykorzystywanych do iterowania po elementach tablicy.

Iteratory

- Iteratory są inteligentnymi wskaźnikami, które iterują po bardziej skomplikowanych strukturach danych niż tablice.
- Wewnętrzne zachowanie iteratorów zależy od struktury danych, po której one iterują – każdy typ kontenerowy zapewnia swój własny rodzaj iteratora (iteratory współdzielą ten sam interfejs, lecz posiadają różne typy).
- Wszystkie klasy kontenerowe udostępniają te same podstawowe funkcje składowe, które umożliwiają im wykorzystanie iteratorów do nawigowania po ich elementach:
  - begin() – zwraca iterator reprezentujący początek elementów w kontenerze (początkiem tym jest pozycja pierwszego elementu, jeśli kontener nie jest pusty).
  - end() – zwraca iterator reprezentujący koniec elementów w kontenerze (końcem tym jest pozycja za ostatnim elementem).

Iteratory

- Funkcje begin() oraz end() definiują zakres półotwarty zawierający pierwszy element, lecz wyłączający ostatni.
- Zakres półotwarty posiada dwie zalety:
  - Istnieje proste kryterium zakończenia pętli iterującej po elementach — wykonywanie pętli kontynuowane jest tak długo, dopóki nie zostanie osiągnięta wartość funkcji end().
  - Unikamy konieczności specjalnej obsługi zakresów pustych. Dla zakresów pustych wartość funkcji begin() równa jest wartości funkcji end().



Iteratory

- Przykład:

```
list<char> coll; // kontener list na elementy znakowe
// …
// wypisz wszystkie elementy – iteruj po elementach
list<char>::const_iterator pos;
for (pos = coll.cbegin(); pos != coll.cend(); ++pos)
    cout << *pos << ' ';
 cout << endl;
```

Przesuwanie iteratora po kolekcji

- Do przesunięcia iteratora do następnego elementu można użyć prefiksowego operatora inkrementacji (++iter) albo postfiksowego (iter++).
- Należy używać prefiksowego operatora inkrementacji ponieważ może on mieć lepszą wydajność niż operator postfiksowi – ten ostatni wewnętrznie wykorzystuje obiekt tymczasowy, gdyż musi zwrócić starą pozycję iteratora.

Zachowanie pierwotnych wartości w kolekcji

- Można zainicjalizować iterator wywołaniem funkcji składowej begin() kontenera i dzięki temu pominąć jawną deklarację jego typu używając słowa `auto`.
- Przykład:

```
for (auto it = coll.begin(); it != coll.end(); ++it) {
    cout << *it << ' ';
}
```

- Zaletą stosowania `auto` jest zdecydowane polepszenie przejrzystości kodu.
- Wadą stosowania `auto` jest utrata jawnie deklarowanej niemodyfikowalności iteratora, ponieważ po zastosowaniu inicjalizacji za pomocą `coll.begin()` otrzymamy iterator modyfikujący (bez `const`).

Zachowanie pierwotnych wartości w kolekcji

- Wartość zwracana z funkcji składowej `begin()` jest obiektem typu *kontener::iterator*.
- Aby zachować możliwość stosowania iteratorów niemodyfikujących i równocześnie zapewnić wygodę stosowania `auto`, w klasach kontenerów w C++11 udostępniono funkcje składowe `cbegin()` i `cend()` – obie zwracają obiekt iteratora typu *kontener::const_iterator*.
- Iterator typu const_iterator nie pozwala na modyfikację elementów kolekcji.

Iteratory a pętle zakresowe

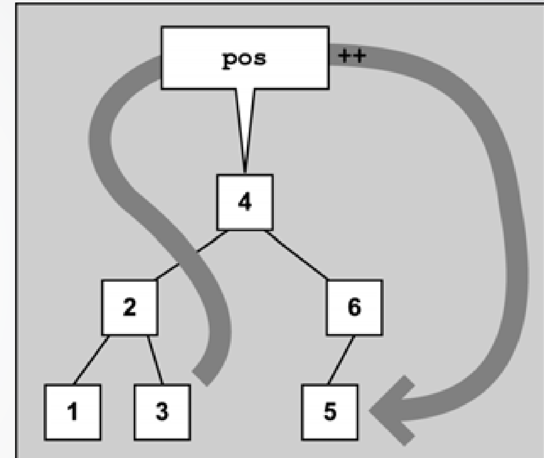
- W przypadku kontenerów zakresowa pętla for jest niczym innym jak wygodnym interfejsem, zdefiniowanym do iterowania po wszystkich elementach przekazanego zakresu kolekcji (od początku do końca).
- Wewnątrz ciała pętli bieżący element jest inicjalizowany wartością, do której odnosi się bieżący iterator.
- Przykład:
  ```
  for (auto elem: coll) { … }
  ```
  jest równoważna z
  ```
  for (auto it = coll.begin(); it != coll.end(); ++it) {
      auto elem = *pos;
      …
  }
  ```
- W zakresowej pętli for warto deklarować `elem` jako niemodyfikującą referencję – w taki sposób eliminuje się konieczność tworzenia kopii elementów przeglądanego kontenera.
- Przykład:
  ```
  for (const auto &elem: coll) { … }
  ```

Użycie kontenerów – przykład 1

```
using IntSet = set<int, greater<int>>;
…
IntSet coll;
…
coll.insert({3, 1, 5, 4, 1, 6, 2});
…
IntSet::const_iterator pos;
for (pos = coll.begin(); pos != coll.end(); ++pos)
    cout << *pos << ' ';
cout << endl;
```

Użycie kontenerów – przykład 1

- Ponieważ iterator zdefiniowany jest przez kontener, działa on prawidłowo nawet w przypadku, gdy wewnętrzna struktura kontenera jest bardziej skomplikowana.
- Jeśli na przykład iterator odnosi się do trzeciego elementu, operator ++ przenosi go do elementu czwartego znajdującego się na wierzchołku drzewa.
- Po następnym wywołaniu operatora ++ iterator będzie się odnosić do piątego elementu umieszczonego u spodu drzewa.



Iteratory strumieniowe

- Iterator strumieniowy (ang. stream iterator) to adaptator iteratora, który umożliwia wykorzystanie strumienia jako źródła lub przeznaczania algorytmów.
- W szczególności iterator strumieniowy wejściowy służy do odczytywania elementów ze strumienia wejściowego, a iterator strumieniowy wyjściowy do wpisywania wartości do strumienia wyjściowego.

cdn

Algorytmy

- Struktury danych pomagają w przechowywaniu i obsłudze danych na różne sposoby, natomiast algorytmy powodują zastosowanie określonych przekształceń danych znajdujących się w strukturach danych (tablicach i kontenerach).
- Definicje różnych algorytmów znajdują się w pliku nagłówkowym <algorithm>.
- Wszystkie algorytmy STL przetwarzają najczęściej zakres danych wyznaczony przez iteratory.

Zakresy w algorytmach STL

- Wszystkie algorytmy STL przetwarzają zakres wyznaczony przez iteratory.
- Pierwszy zakres określony jest przez jego początek i koniec.
- Jeśli chodzi o zakresy dodatkowe, to w większości przypadków wystarczy podać sam ich początek, ponieważ koniec wynika z liczby elementów występujących w pierwszym zakresie.

Zakresy w algorytmach STL

- Po stronie funkcji wywołującej leży obowiązek zapewnienia poprawności zakresów – oznacza to, że początek musi odnosić się do wcześniejszego lub tego samego elementu co koniec tego samego kontenera.
- Algorytmy działają w trybie nadpisywania, a nie wstawiania – funkcja wywołująca musi więc zapewnić, aby zakresy docelowe posiadały odpowiedni rozmiar.
- Zakres podajemy [od-pierwszego, do-ostatniego),

Zakresy w algorytmach STL

- W pliku nagłówkowym `<algorithm>` zdefiniowanych jest około 80 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 kolekcje sekwencyjnie, więc wystarcza im iterator jednokierunkowy.
- Wiele algorytmów fakt nieodnalezienia elementu standardowo oznacza zwróceniem końca zakresu.

Minimum i maksimum

- Obliczanie wartości minimalnej oraz maksymalnej:

```
template <class T>
inline const T& min (const T &a, const T &b)
   { return b<a ? b : a; }
template <class T>
inline const T& max (const T &a, const T &b)
   { return a<b ? b : a; }
```

- Istnieją też wersje tych szablonów z komparatorami (funkcja lub obiekt funkcyjny):

```
template <class T, class C>
inline const T& min (const T &a, const T &b,
C comp)
   { return comp(b,a) ? b : a; }
template <class T>
inline const T& max (const T &a, const T &b,
C comp)
   { return comp(a,b) ? b : a; }
```

Minimum i maksimum

- Przykład 1:

```
bool int_ptr_less (int *p, int *q) {
  return *p < *q;
}
…
int x = 33, y = 44;
int *px = &x, *py = &y;
int *pmax = std::max(px, py, int_ptr_less);
```

- Przykład 2:

```
int i;
long l;
…
// niezgodne typy argumentów
// l = max(i,l); // BŁĄD

l = std::max<long>(i,l); // OK
```

Zamiana wartości

- Zamiana dwóch wartości:
```
template <class T>
inline void swap(T &a, T &b) {
  T tmp(move(a));
  a = move(b);
  b = move(tmp);
}
```
- Przykład:
```
int x = 33, y = 44;
…
std::swap(x,y);
```

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

- Przeznaczenie algorytmu można wywnioskować po jego nazwie:
  - Przyrostek `_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 `_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.

cdn