Dr inż. Jolanta Kawulok, pokój: 503

PK3 Laboratorium 3 (Iteratory)

Iteratory - interfejs pomiędzy algorytmami, a kontenerami

https://refactoring.guru/pl/design-patterns/iterator/cpp/example
https://refactoring.guru/pl/design-patterns/iterator
https://infotraining.bitbucket.io/cpp-stl/iteratory.html
https://en.cppreference.com/w/cpp/iterator

Iteratory:

- zachowują się jak zwyczajne wskaźniki
- obiekty, których zadaniem jest iteracja po elementach sekwencji
- działają dla wszystkich kontenerów
- nie istnieje żadna klasa bazowa iteratora

Istnieje 5 kategorii iteratorów

1. **Iterator wejściowy** (ang. *input iterator*) — umożliwia odczyt w przód (np. w strumieniach wejściowych)
   Służą do JEDNORAZOWEGO odczytywania informacji
2. **Iterator wyjściowy** (ang. *output iterator*) — umożliwia zapis w przód (np. w strumieniach wyjściowych)
   Służą do JEDNORAZOWEGO zapisu – informacje nie są nadpisywane, a zapisywane w nowym miejscu
3. **Iterator postępujący** (ang. *forward iterator*) — umożliwia odczyt i zapis w przód (kombinacja iteratorów wejściowych i wyjściowych)
   Każdy z elementów można przetwarzać wielokrotnie (następuje nadpisanie wartości przy zapisie)
4. **Iterator dwukierunkowy** (ang. *bidirectional iterator*) — umożliwia odczyt i zapis w przód i wstecz (np. iteratory listy)
   Mają wszystkie cechy iteratorów postępujących i dodatkową możliwość cofania się do poprzedniego elementu
5. **Iterator dostępu swobodnego** (and. *random access iterator*) — umożliwia odczyt i zapis z dostępem swobodnym
   Iterator dla wektora, kolejki o dwóch końcach, napisów (string)
   Jest to odpowiednik wskaźnika.

	input	output	forward	bidirectional	rand.
Odczyt V=*p p->x	T	N	T	T	T
Zapis *p=v	n	t	t	t	t
Inkrementacja ++p p++	t	t	t	t	t
Dekrementacja p--	n	n	n	t	t
Porównania p1==p2 p1!=p2	t	n	t	t	t
Przypisanie p1=p2	n	n	t	t	t
*p — dereferencja (jesli p == q, to *p jest równowazne *q)	n	n	t	t	t
Jesli p == q to ++p == ++q	n	n	t	t	t
Inne. np.: Porównania: p>q q+=n, --q, q-=n Swobodny dostęp (indeksowanie): p[n]	n	n	n	n	t
Konstr bezparametrowy	t	n	t	t	t
Konstr kopiujący	T	t	t	t	t

Iteratory „zwykłe”

Iteratory pomagają nam przejść przez wszystkie elementy danego kontenera.

Przykład:

```
vector<int> myvec = { 1,1,2,2,3,3 };
vector<int>::iterator myit1;                          // zwykły iterator
vector<int>::const_iterator myit2;                    // stałe elementy
const vector<int>::iterator myit3 = myvec.begin()+2;  // stały iterator - możemy tylko
                                                            przy deklaracji go zdefiniować

for (myit1 = myvec.begin(); myit1 < myvec.end(); ++myit1) {
       cout << *myit1 << endl; // można wyświetlić elementy
       *myit1 = 1; // można elementy zmienić
}

for (myit2 = myvec.begin(); myit2 < myvec.end(); ++myit2) {
       //*myit2 = 1; // nie możemy zmieniać elementów
       cout << *myit2 << endl; // można wyświetlić elementy
}

//for (myit3 = myvec.begin(); myit3 < myvec.end(); myit3++) // nie możemy iterować, bo stały
                                                            iterator
cout << "myit3 " << *myit3 << endl;  // można wyświetlić dany element
```

Iteratory wstawiające

„Zwykłe iteratory” nie nadają się, aby wstawiać (dodawać) NOWE elementy do kontenerów, do tego celu trzeba stosować iteratory wstawiające.

1) Operacje ++it oraz it++ dla iteratorów wstawiających są operacjami pustymi, stąd nie stosujemy ich
2) Przy zapisie bezpieczniej robić przerwę pomiędzy ostatnimi dwoma znakami ‘>’
   a. **POPRAWNE:** **back_insert_iterator<list<int> >it(L);**
   b. **ZAMIAST:** **back_insert_iterator<list<int>>it(L);**
3) Podział iteratorów wstawiających:
   a. końcowe (ang. *back inserters*)
      i. wykorzystują metodę kontenera push_back(v)
      ii. są dostępne dla wszystkich kontenerów sekwencyjnych (bo mają one metodę push_back)
   b. początkowe (ang. *front inserters*)
      i. wykorzystują metodę kontenera push_front(v)
      ii. są dostępne dla list i kolejek o dwóch końcach (bo tylko one mają metodę push_front)
   c. ogólne (ang. *inserters*)
      i. wykorzystują metodę kontenera insert(v, it)
4) Deklaracje:

```
a. back_insert_iterator  <MYTYPcon<MYtyp2> > iter(cont);    // wstawia na końcu kontenera
b. front_insert_iterator <MYTYPcon<MYtyp2> > iter(cont);    // wstawia na początek kontenera
c. insert_iterator       <MYTYPcon<MYtyp2> > iter(cont, pos); // wstawia w określonym miejscu
```

gdzie:

```
MYTYPcon<MYtyp2> - kontener zawierający elemety typu MYtyp2 np. vector<int>
iter;            - deklarowany wstawiacz
cont;            - nazwa konkretnego kontenera
pos;             - pozycja (wykorzystujemy zwykły iterator)
```

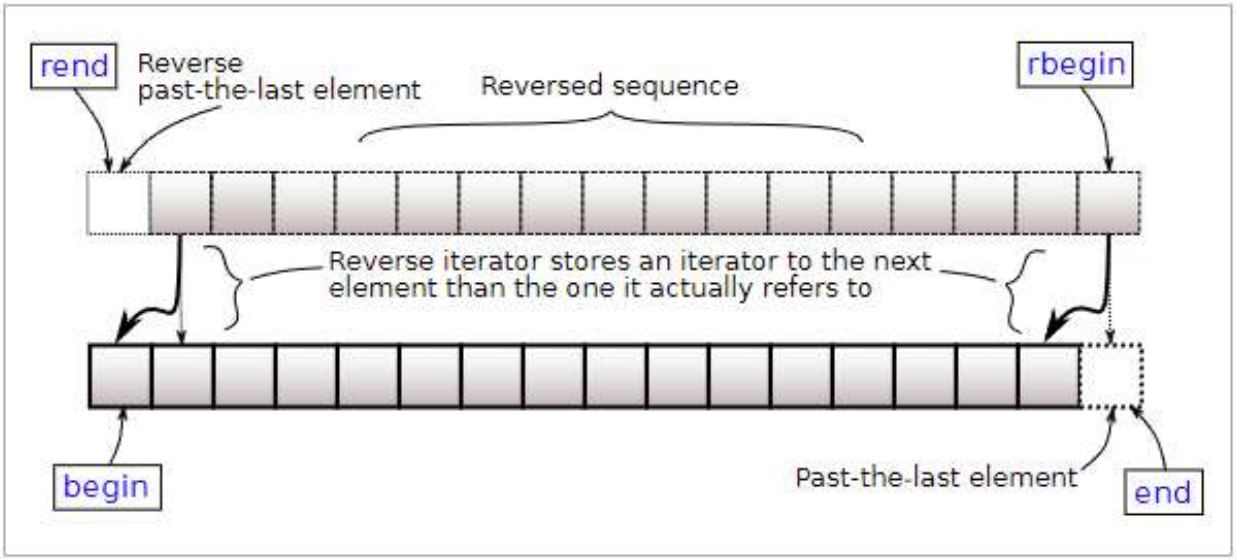
Przykład cd:

```
vector <int> V;
vector <int>::iterator itBAD = V.begin();
for (int i = 0; i < 3; ++i, ++itBAD)
       *itBAD = myvec[i]; // błąd! - próba nadpisania  na elemencie, który nie istnieje

back_insert_iterator <vector<int> > it(V);
for (int i = 0; i < 3; ++i) // ++it pominięte
{
        it = i;
        *it = myvec[i]; //  (*it oraz samo it jest poprawne)
}
// Wektor myvec na koniec będzie zawierał 6 elementów

// ==================
list<int> coll = { 1,2,3,4 };
insert_iterator<list<int> > iter(coll, --coll.end())  ; // Wstawienie na PRZEDOSTANIE miejsce
iter = 10;
iter = 20;
iter = 30;

// Wynik: 1,2,3,10,20,30,4
```

Iteratory wsteczne (odwrotne) – reverse iterator

1. Umożliwiają przeglądanie kontenera wstecz
2. Mają interfejs zwykłych iteratorów
3. Metoda `base()` zwraca iterator bazowy dla odwrotnego
4. Deklaracje:

```
a. MYTYPcon <MYtyp2>::reverse_iterator rit;
b. MYTYPcon <MYtyp2>::reverse_iterator rit(it1); // inicjowany iteratorem it
//gdzie
    MYTYPcon<MYtyp2> - kontener np. vector <int>
```



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

```
// Przyklad CD

vector <int> v3;
back_insert_iterator <vector<int> > it3(v3);
for (int i = 1; i <= 5; ++i)
       it3 = i;

vector<int>::iterator it3a(v3.begin() + 2);
cout << *it3a << endl; // wynik: 3

vector<int> ::reverse_iterator rit3(it3a);
cout << *rit3 << endl; // wynik: 2

vector <int> ::iterator it3b;
it3b = rit3.base(); // base( ) zwraca iterator bazowy dla odwrotnego
cout << *it3b << endl; // wynik: 3
```

Iteratory strumieniowe

1. Zachowują się jak zwyczajne iteratory
    a. Mają interfejs zwykłych iteratorów
2. Operują na strumieniach i/o
    a. Iteratory strumieniowe wyjściowe:
        i. analogiczne do iteratorów wstawiających, z tym, że kontenerem jest strumień wyjściowy
        ii. `ostream_iterator <mytype> iter(os); // iterator związany z os (np. cout, plik wyj.)`
        iii. `ostream_iterator <mytype> iter(os, mySYMBOL); // rozdziela dodawane elementy wykorzystując mySYMBOL (np. "\t")`

    b. Iteratory strumieniowe wejściowe:
        i. czytamy ze strumienia wejściowego
        ii. `istream_iterator <mytype> iter(is); // iterator związany z is (np. cin, plik wej.)`
        iii. `istream_iterator <mytype> iter;  // iterator końca strumienia`

```
// Przykład

vector <int> vecIN;
back_insert_iterator <vector<int> > bIt(vecIN);
istream_iterator <int> myIN(cin);  // iterator wczytujący z ekranu
istream_iterator <int> OUT_end;    // iterator końca strumienia

do // z wejścia przekopiowanie danych do wektora
{
    bIt = *myIN;
    ++myIN;
}  while (myIN != OUT_end);


ostream_iterator < int > wy(cout, "\n"); // iterator wyjścia, który rozdziela dodawane
                                         elementy znakiem nowej linii
copy(vecIN.begin(), vecIN.end(), wy);



// ====================================================================

// PRZYKLAD CZYTANIA Z PLIKU

ifstream mojPLIK("input.txt");
istream_iterator<string> Start(mojPLIK);
istream_iterator<string> Koniec;
ostream_iterator<string> WYJ(cout, " xx \n");

// WYSWIETLENIE NA 2 SPOSOBY:
// 1) sposób:
for (istream_iterator<string> it = Start; it != Koniec; ++it)
{
    cout << *it << "\n";
}
// 2) sposób:
copy(Start, Koniec, WYJ);
```

Iteratory na własny kontener

1. Po stworzeniu własnego kontenera, należy zdefiniować do niego iteratory.
2. Iteratory te powinny zawierać cechy iteratorów umożliwiające korzystanie z wybranych algorytmów. (najlepiej, aby zawierał cechy minimum jak iteratory postępujące)

https://en.cppreference.com/w/cpp/iterator
https://refactoring.guru/pl/design-patterns/iterator/cpp/example

Zadania

Treści zadań dostępne są w komentarzach pliku cpp

- Dla łatwość sprawdzania zadania (do wszystkich ćwiczeń), proszę wszystko umieszczać w 1 pliku głównym cpp. Jednak osobno część z deklaracjami i z definicjami. W późniejszym projekcie należy program rozbić na pliki.
- Odpowiedzi do pytań proszę napisać w komentarzach w programie.
- Wszystkie zadania wykonujemy w ramach jednego programu – **proszę edytować zadany kod „pk3my_21_lab03-studenci.cpp**", a następnie stopniowo odkomentować linijki kodu podczas kolejnych stworzonych przez siebie zadań.