Prezentacja PK4

Temat: Algorytmy i Iteratory STL

Autorzy:
1. Michał Jankowski
2. Jakub Klimek

Iteratory

- Podstawowe narzędzie do wykonywania operacji na kolekcjach
- Możliwość poruszania się po elementach bez znajomości typu rzeczywistego
- “**Uogólnienie wskaźnika**”
- Biblioteka standardowa dostarcza gotowe iteratory (#include<iterator>)
- Dozwolone wykorzystanie typedef

```
vector<int>some_vector = { 1,2,3 };
auto some_iterator = some_vector.begin();// pozycja (poczatek)
auto some_value = *some_iterator; //wartosc
```

Umożliwia nam to przeciążony **operator*** i **operator=**

```
typedef vector<int>::iterator  IT;

IT iter, koniec = some_vector.end();
```

Iteratory wejścia (“Input”)

- Jednokrotny przebieg kolekcji do przodu (single pass)
- Pozwalają tylko na odczyt elementu wskazywanego (uwaga na zakres-end())
- Posiada domyślny kostruktor kopiujący
- Związany ze strumieniem wejściowym

```
a == b
a!=b
*a
a->m
++a
a++
```

Iteratory wyjścia (“Output”)

- Jednokrotny przebieg kolekcji do przodu (single pass)
- Pozwalają tylko na zapis elementu wskazywanego
- Posiada domyślny kostruktor kopiujący
- Związany ze strumieniem wyjściowym

t->typ obiektu wskazywanego przez typ iteratora

```
*a = t
*a++ = t
 ++a
 a++
```

Iteratory przejścia do przodu (“Forward”)

- Połączenie operatora wejścia wyjścia
- Posiada konstruktor kopiujący
- Możliwy zarówno odczyt i zapis
- Poruszanie się po kolekcji do przodu tylko poprzez inkrementacje iteratora
- Posiada domyślny konstruktor

```
a == b
a! = b
*a
*a = t
*a++ = t
a->m
++a
a++
```

Iteratory dwukierunkowe (“Bidirectional”)

- Zachowanie podobne do iteratora przejścia do przodu
- Można dekrementować iterator

```
 a == b
 a! = b
 *a
 *a = t
 *a++ = t
 a->m
 ++a
 a++
 --a
a--
*a--
```

Iteratory bezpośredniego dostępu (Random Access)

- Dziedziczy wszystko po iteratorach dwukierunkowych
- Posiada możliwość dostępu do wybranego składnika struktury
- Wykorzystuje operatory : <, > , +=, -=,+,-
- Także operator inedksacji []

```
a == b
a! = b
*a
*a = t
*a++ = t
a->m
++a
a++
--a
a--
*a--
```

```
*a--
a+n
n+a
a-n
a-b
a>b
a<=b
a+=n
a-=n
a[n]
```

Generatory iteratorów

- **back_inserter**-> dodawanie nowych elementów od końca kolekcji “x”
- **front_inserter**-> dodawanie nowych elementow od początku kolekcji “x”
- **inserter**-> tworzy insert_iterator
- **make_move_inserter** (C++11)-> tworzy **move_iterator**

```
vector<int> some_vec1 = { 1, 2, 3 };

vector<int> some_vec2 = { 4, 5, 6 };

std::copy(some_vec1.begin(), some_vec1.end(), std::back_inserter(some_vec2));
```

```
some_vec1 = 1 2 3
some_vec2 = 4 5 6 1 2 3
```

```
deque<int> some_vec1 = { 1, 2, 3 };

deque<int> some_vec2 = { 4, 5, 6 };

std::copy(some_vec1.begin(), some_vec1.end(), std::front_inserter(some_vec2));
```

```
some_vec1 = 1 2 3
some_vec2 = 3 2 1 4 5 6
```

Operacje na Iteratorach

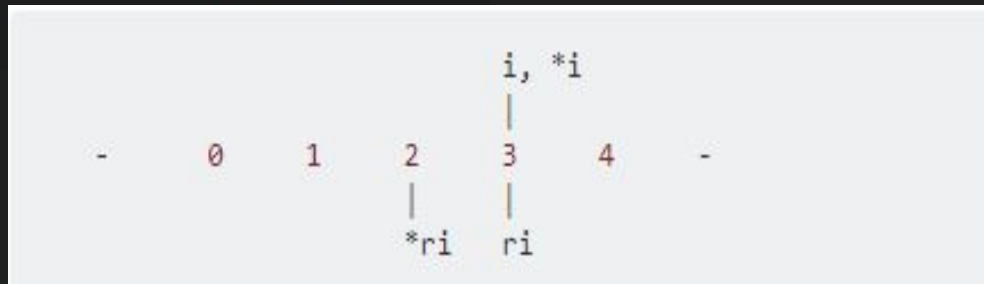
- advance()->przesuwanie iteratora o n pozycji
- distance()->odległość pomiędzy dwoma iteratorami
- begin()->iterator na pierwszy elem.
- end()->iterator na ostatni elem.
- prev()->iterator na poprzedni elem.
- next()->iterator na nastp. elem.
- base()-> zamienia reverse_itrerator na odpowiadający forward iterator

```
std::advance(it, 5);
```

```
std::vector<int>::iterator first = some_vector.begin();
std::vector<int>::iterator last = some_vector.end();
std::distance(first, last);
```

```
cout << "\nsome_vec = ";
for (auto it = begin(some_vector); it !=end(some_vector); ++it)
    std::cout << ' ' << *it;
```

```
some_vec =  1 2 3
```



Predefiniowane iteratory

Wyróżniamy kilka takich iteratorów:

- reverse_iterator-> Zachowanie przeciwne do zwykłego iteratora
- move_iterator (C++11)
- back_insert_iterator
- front_insert_iterator
- istream_iterator
- ostream_iterator
- const_iterator -> korzystamy jeśli kolekcja nie jest modyfikowalna albo tylko do odczytu

Predefiniowane iteratory

Odpowiedniki end() i begin():

- cbegin(), cend()->const_iterator
- rbegin(),rend()->reverse_iterator
- crbegin(),crend()->const_reverse_iterator

```
std::vector<int> some_vector4(4);

int i = 0;

std::vector<int>::reverse_iterator rit = some_vector4.rbegin();
for (; rit != some_vector4.rend(); ++rit)
    *rit = ++i;
```

```
4 3 2 1
```

Predefiniowane iteratory

```
std::vector<std::string> test(3);
std::vector<std::string> some_string{ "jeden","dwa","trzy" };

typedef std::vector<std::string>::iterator It;

std::copy(std::move_iterator<It>(some_string.begin()),
    std::move_iterator<It>(some_string.end()),
    test.begin());

// some_string zawiera nieokreslona wartość
some_string.clear();

std::cout << "foo:";
for (std::string& x : test) std::cout << ' ' << x;
std::cout << '\n';
```

```
test: jeden dwa trzy
```

Wyrażenia lambda

- Pozwala na tworzenie anonimowych funkcji (“ciało bez nazwy”)
- Można przypisać do zmiennej

```
int dod(int a, int b)
{
    return a + b;
}
```

=

```
[](int a, int b)->int { return a + b; };
```

```
auto foo = [](int a, int b)->int { return a + b; }(3, 4);
```

```
vector<int> wektor(32);
vector<int>::iterator it;
generate(wektor.begin(), wektor.end(), []() {return rand() % 11; });
```

Budowa wyrażenia lambda

```
[ capture ]( params ) mutable(optional) throw()(optional)-> ret { body }( set_params );
```

- [capture] -> lista przechwytywania wyrażenia
- {body}->ciało wyrażenia
- mutable ->stosowane, jeżeli zamierzamy modyfikować zmienne przekazywane przez wartość (z listy przechwytywania)
- throw -> zwraca wyjątki
- (params)->parametry wyrażenia
- ->ret-> określamy zwracany typ
- (set_params)-> czy mają posiadać domyślne wartości

Lista przechwytywania wyrażenia lambda

- [] - pusta lista
- [zmienna1] - oddzielone przecinkiem. Domyślnie, wartości zmiennych są kopiowane. Jeżeli chcemy modyfikować zmienne poza ciałem wyrażenia, używamy referencji (&)
- [=] - automatycznie przekazuje używane zmienne przez kopiowanie
- [&] - automatycznie przekazuje używane zmienne przez referencję
- [=, lista_referencji] - zmienne na liście przekazywane przez referencję, reszta przez wartość
- [&, lista_zmiennych] - zmienne na liście przekazywane przez kopiowanie, reszta przez referencję

Uwagi do iteratorów i zadania 3

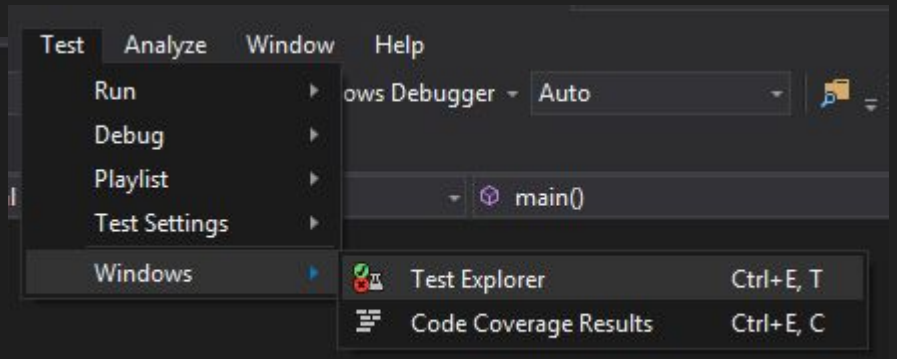
```cpp
#pragma once
#include<iterator>
#include<array>

class Iteratory
{
public:
    Iteratory();
    typedef typename std::array<int, 5> kolejka;
    typedef typename kolejka::iterator iterator;
    typedef typename kolejka::const_iterator const_iterator;
    typedef typename kolejka::reverse_iterator reverse_iterator;
    //using tablica = std::array<T, 3>;
    //using iterator = tablica::iterator;
    //using cons_iterator = tablica::const_iterator;

    inline iterator begin() noexcept { return  dque.begin(); }
    inline const_iterator cbegin() const noexcept { return dque.cbegin(); }
    inline iterator end() noexcept { return dque.end(); }
    inline const_iterator  cend() const noexcept { return dque.cend(); }
    inline reverse_iterator rbegin() noexcept { return dque.rbegin(); }
    inline reverse_iterator rend() noexcept { return dque.rend(); }

    //mozna dodac do iteratorow noexcept, nie wyrzuca wyjatku dziala szybciej

private:
    kolejka dque;
};
```



Algorytmy

- Kontenery dostarczane przez bibliotekę standardową udostępniają wąski zakres funkcji
- Biblioteka standardowa dostarcza algorytmy ***generyczne*** (nagłówek *algorithm)*
- Algorytmy te operują na iteratorach.
- Algorytmy są niezależne od kontenera
- Ale są zależne od operacji na elementach

```
vector<int> vec = { 1 , 2, 3, 4, 5, 6, 7, 21, 23, 22, 24 };
int val = 22;
auto result = find(vec.cbegin(), vec.cend(), val);
cout << "Value " << val << (result == vec.cend() ?
    " is not present" : " is present") << endl;
```

Algorytmy tylko odczytujące

- Operujące na jednym kontenerze - np. find
- Operujące na dwóch kontenerach - np. equal

```
int sum = accumulate(vec.cbegin(), vec.cend(), 0);
string v[] = { "Ala ", "ma ", "kota" };
//nie zadziała - brak operatora + dla const char*
string sen = accumulate(cbegin(v), cend(v), "");
//zamiast tego:
string sen = accumulate(cbegin(v), cend(v), string(""));
```

```
//vec2 ma przynajmniej tyle elementow, ile vec
bool isEqual = equal(vec.cbegin(), vec.cend(), vec2.cbegin());
```

Algorytmy zapisujące do elementów kontenera

- Algorytmy ***NIGDY*** nie zmieniają rozmiaru kontenera na którym operują
- Algorytmy nie sprawdzają legalności operacji zapisu
- Algorytmy kopiujące

```
//wypelnia vec zerami
fill(vec.begin(), vec.end(), 0);
```

```
//fill_n wypelnia n elementow dana wartoscia
vector<int> vec; //pusty vector
fill_n(vec.begin(), 10, 0); //KATASTROFA!!!
fill_n(back_inserter(vec), 10, 0); //OK
```

Algorytmy kopiujące

- Należą do algorytmów zapisujących
- Przyrostek _copy
- Przykład: copy, replace_copy

```
int a1[] = { 1,2,3,4 };
int a2[sizeof(a1) / sizeof(*a1)];
copy(begin(a1), end(a1), a2);
```

```
array<int, 5> a1 = { 0, 1, 2, 1, 2 };
vector<int> a2 = { 4, 4, 4, 4, 4 };
replace_copy(a1.cbegin(), a1.cend(), a2.begin(), 2, 42);
//a1 = 0,1,2,1,2
//a2 = 0,1,42,1,42
```

Algorytmy zmieniające kolejność elementów

- Należą do algorytmów zapisujących
- Są to m. in. algorytmy sortujące
- Przykład: posortowanie ciągu wyrazów i usunięcie duplikatów

```
void elimDup(vector<string> &words) {
//words = the quick red fox jumps over
//        the slow red turtle
    sort(words.begin(), words.end());
    auto newEnd = unique(words.begin(), words.end());
    words.erase(newEnd, words.end());
//words = fox jumps over quick red slow
//      the turtle
}
```

Przekazanie funkcji do algorytmu

- Algorytmy posiadają funkcje decydujące, jak klasyfikować elementy
- Algorytmy można przeładować, podając własną funkcję
- W zależności od algorytmu, funkcja przyjmuje jeden lub 2 argumenty
- Jeżeli chcemy przekazać więcej parametrów, używamy wyrażeń lambda

```
bool isShorter(const string &v1, const string &v2) {
    return v1.size() < v2.size();
}
```

```
elimDup(words);
//words = fox jumps over quick red slow the turtle
stable_sort(words.begin(), words.end(), isShorter);
//words = fox red the over slow jumps quick turtle
```

Inne przykłady

```
elimDup(words);
//words = fox jumps over quick red slow the turtle
stable_sort(words.begin(), words.end(),
    [](const string &v1, const string &v2) {
        return v1.size() < v2.size();
    });
//words = fox red the over slow jumps quick turtle
```

```
size_t size = 3;
for_each(words.cbegin(), words.cend(),
    [=](const string &v){
        if (v.size() > size){
            cout << v << endl;
        }
    });
```

Parametry algorytmów

Ze względu na parametry:

- alg(*początek, koniec, inne)*
- alg(*początek, koniec, wynik, inne)*
- alg(*początek, koniec, początek2, inne)*
- alg(*początek, koniec, początek2, koniec2, inne)*

Konwencja nazewnictwa:

- Jeżeli algorytm używa operatora == lub <, możliwe jest podanie własnego predykatu
- Jeżeli algorytm używa zewnętrznie podanej wartości, istnieje wersja *algorytm_if* przyjmująca dodatkowy predykat
- Dla algorytmów zmieniających elementy, istnieje wersja *algorytm_copy,* niemodyfikująca źródłowego ciągu

Algorytmy specyficzne dla kontenerów

- Kontenery *list* i *forwarded_list* definiują własne wersje algorytmów.
- Ze względu na implementację są one szybsze od generycznych.
- Mogą modyfikować ilość elementów kontenera.

lst.merge(lst2) lst.merge(lst2, comp)	lst.sort() lst.sort(comp)
lst.remove(val) lst.remove(pred)	lst.unique() lst.unique(pred)
lst.reverse()	
lst.splice(args) lub lst.splice_after(args) args: (p, lst2) (p, lst2, p2)	