Лекція 14

Алгоритми

У цій главі...

Узагальнені алгоритми, описані в стандартній бібліотеці , розділяються на чотири категорії: алгоритми, що не змінюють уміст контейнерів; алгоритми, що модифікують уміст послідовності; алгоритми сортування і пошуку; і чисельні алгоритми. Перші три групи алгоритмів описані в заголовку `<algorithm>`, а чисельні алгоритми — у заголовку `<numeric>`. Усі вони знаходяться в просторі імен `std`.

14.1. Алгоритми, що не змінюють уміст контейнерів

Розглянемо першу групу алгоритмів.

14.1.1. Алгоритм for_each

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function f);
```

Застосовує функцію `f()` до результатів розіменування кожного ітератора з діапазону, обмеженого ітераторами `first` і `last`. Повертає функцію `f()`.

Приклад алгоритму for_each

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

void ToHex (int n)
{
   printf("%x ",n); // Выводим целое число в шестнадцатеричном виді
}

int main()
{
   vector <int> array (15);
   for (int i = 0; i < array.size (); ++i) array[i] = i+1;
   for_each (array.begin (), array.end (), ToHex);
   printf("\n");
   return 0;
}
```

Результат

```
1 2 3 4 5 6 7 8 9 a b c d e f
```

14.1.2. Алгоритм find

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Виконує пошук елемента, що має значення `value`, у послідовності, обмеженої ітераторами `first` і `last`. Якщо шуканий елемент знайдений, повертає ітератор, установлений на його перше входження. У противному випадку він повертає ітератор `last`.

Приклад алгоритму find

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   vector <int> array (15);
   for (int i = 0; i < array.size (); ++i)
```

```
    {
        array[i] = 15-i;
        printf("%d ",array[i]);
    }
    printf("\n");
    vector<int>::iterator where = find (array.begin (), array.end (), 5);

    // Виводимо позицію числа 5 относительно початку вектора
    printf("Відносна позиція числа 5: %d \n", where-array.begin());

    return 0;
}
```

Результат

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Відносна позиція числа 5:10
```

14.1.3. Алгоритм find_if

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate
pred);
```

Повертає перший ітератор `i` з діапазону `[first,last)`, для якого виконуються умови `*i == value` і `pred(*i) != false`. Якщо елемент не знайдений, повертає `last`.

Приклад алгоритму find_if

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

bool IsMultiply7 (int n)
{
   return n%7==0; // Якщо число кратне 7, повертаємо true
}

int main()
{
   vector<int> array (15);
   for (int i = 0; i < array.size (); ++i) array[i] = i+1;

   vector<int>::iterator where;
   where = find_if(array.begin (), array.end (), IsMultiply7);
   printf("Шукане число: %d\n",*where);
   return 0;
}
```

Результат

```
Шукане число: 7
```

14.1.4. Алгоритм find_end

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
  ForwardIterator1 find_end(ForwardIterator1 first1, ForwardIterator1 last1,
                            ForwardIterator2 first2, ForwardIterator2 last2,
                            BinaryPredicate pred);
```

Перша версія алгоритму `find_end` знаходить у контейнері підпослідовність, що складається з однакових елементів. Повертає останній ітератор з діапазону `[first1, last1- (last2- first2))`, для якого виконується умова `*(i+n) == *(first1+n)` для всех `0 < n < last2-first2`. Інакше кажучи, повертає ітератор, установлений на останнє входження підпослідовності, обмеженої ітераторами `first2` і `last2`, у діапазон, заданий ітераторами `first1` і `last1`. У противному випадку він повертає ітератор `last1`.

Друга версія алгоритму дозволяє задавати бінарний предикат, що визначає умову збігу: `pred(*(i+n),*(first2+n))!= false` для всех `0 < n < last2-first2`.

Приклад алгоритму find_end

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[3] = {4,5,6};
   int i;
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   vector<int>::iterator where;
   where = find_end(first.begin(), first.end(), second.begin(), second.end());
   printf("Шукане число: %d\n",*where);
   return 0;
}
```

Результат

```
  1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
  4   5   6
Шукане число: 4
```

14.1.5. Алгоритм find_first_of

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 find_first_of(ForwardIterator1 first1,
                                 ForwardIterator1 last1,
                                  ForwardIterator2 first2,
                                 ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
         class BinaryPredicate>
  ForwardIterator1 find_first_of(ForwardIterator1 first1,
                                 ForwardIterator1 last1,
                                 ForwardIterator2 first2,
                                 ForwardIterator2 last2,
                                 BinaryPredicate pred);
```

Перша версія алгоритму `find_first_of` знаходить у контейнері перший елемент підпослідовності, що збігає з елементом другої підпослідовності. Повертає перший ітератор `i` з діапазону `[first2, last2)`, для якого виконується умова `*i == *j`. У противному випадку він повертає ітератор `last1`.

Друга версія алгоритму дозволяє задавати бінарний предикат, що визначає умову збігу: `pred(*i,*j)!= false`.

Приклад алгоритму find_first_of

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[3] = {4,5,6};
   int i;
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
```

```
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   vector<int>::iterator where;
   where = find_first_of(first.begin(), first.end(),
                         second.begin(), second.end());
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
  1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
  4   5   6
Шукане число:  5
Позиція: 3
```

14.1.6. Алгоритм adjacent_find

```
template<class ForwardIterator>
  ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator adjacent_find(ForwardIterator first, ForwardIterator last,
                                BinaryPredicate pred);
```

Перша версія виконує пошук однакових сусідніх елементів у послідовності, обмеженої ітераторами `first` і `last`. Повертає перший ітератор, що задовольняє умові `*i == *(i+1)`. Якщо шукана пара не знайдена, алгоритм повертає ітератор `last`.

Друга версія дозволяє задати умову збігу елементів: `pred(*i, *(i+1)) != false`.

Приклад алгоритму adjacent_find

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");

   vector<int>::iterator where;
   where = adjacent_find(first.begin(), first.end());
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
Шукане число: 5
Позиція: 3
```

14.1.7. Алгоритм count

```
template<class InputIterator, class T>
  typename iterator_traits<InputIterator>::difference_type
  count(InputIterator first, InputIterator last, const T& value);
```

Повертає кількість ітераторов `i` з діапазону `[first,last)`, для яких виконується умова `*i == value`.

Приклад алгоритму count

```
#include <vector>
#include <algorithm>
#include <iostream>
```

```
using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");

   int HowMatch = count(first.begin(), first.end(),5);
   printf("Кількість п'ятірок: %d\n", HowMatch);
   return 0;
}
```

Результат

```
1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
Кількість п'ятірок: 4
```

14.1.8. Алгоритм count_if

```
template<class InputIterator, class Predicate>
  typename iterator_traits<InputIterator>::difference_type
  count_if(InputIterator first, InputIterator last, Predicate pred);
```

Повертає кількість ітераторов `i` з діапазону `[first,last)`, для яких виконуються умови `*i == value` і `pred(*i) != false`.

Приклад алгоритму count_if

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

bool odd(int n)
{
 return n%2?true:false;
}

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");

   int HowMatch = count_if(first.begin(), first.end(),odd);
   printf("Кількість непарних %d\n", HowMatch);
   return 0;
}
```

Результат

```
1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
Кількість непарних чисел: 9
```

14.1.9. Алгоритм mismatch

```
template<class InputIterator1, class InputIterator2>
  pair<InputIterator1, InputIterator2>
  mismatch(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  pair<InputIterator1, InputIterator2>
```

```
mismatch(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, BinaryPredicate pred);
```

Перша версія виконує пошук першої розбіжності між двома послідовностями. Повертає пари ітераторов `i` і `j`, що задовольняють умовам: `j == first2 + (i- first1)`, де `i` — перший ітератор з діапазону `[first1,last1)`, що задовольняє умові `!(*i == *(first2+(i- first1)))`. Інакше кажучи, алгоритм `mismatch` повертає ітератори, установлені на початок і кінець послідовності.

Друга версія алгоритму дозволяє задати бінарний предикат, що визначає умову збігу:
`pred(*i, *(first2+(i-first1))) == false`.

Приклад алгоритму mismatch (перший варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[15] = {1,2,3};
   int i;
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
         printf("\n");

   pair<vector<int>::iterator,vector<int>::iterator> where;
   where = mismatch(first.begin(), first.end(), second.begin());
   if(where.first==first.end())
       printf("Однакові вектори");
   else
   {
       printf("Розбіжність у позиції %d\n",where.first-first.begin());
   }
   return 0;
}
```

Результат

```
  1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
  1   2   3
Розбіжність у позиції 3
```

Приклад алгоритму mismatch (другий варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>

using namespace std;

bool square(int n1, int n2)
{
   return n1*n1==n2?true:false;
}

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[15] = {1,4,9};
   int i;
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
```

```
        printf("%3d ",second[i]);
          printf("\n");

   pair<vector<int>::iterator,vector<int>::iterator> where;
   where = mismatch(first.begin(), first.end(), second.begin(),square);
   if(where.first==first.end())
       printf("Однакові вектори");
   else
   {
       printf("Розбіжність у позиції %d\n",where.first-first.begin());
   }
   return 0;
}
```

Результат

```
1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
  1   4   9
Розбіжність у позиції 3
```

14.1.10. Алгоритм equal

```
template<class InputIterator1, class InputIterator2>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2);

template<class InputIterator1, class InputIterator2,
         class BinaryPredicate>
  bool equal(InputIterator1 first1, InputIterator1 last1,
             InputIterator2 first2, BinaryPredicate pred);
```

Перша версія перевіряє, ли збігається діапазон, обмежений ітераторами `first1` і `last1`, з послідовністю, на яку встановлений ітератор `first2`. Якщо діапазони збігаються, алгоритм повертає значення `true`, у противному випадку — значення `false`.

Друга версія дозволяє задати бінарний предикат, що визначає умову збігу:
`pred(*i, *(first2 + (i - first1))) != false`.

Приклад алгоритму equal (перший варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>

using namespace std;

int main()
{
   int initial[3] = {1,2,3};
   int pattern[3] = {1,2,5};
   int i;
   vector<int> first (initial, initial+3), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   if(equal(first.begin(), first.end(), second.begin()))
       printf("Однакові вектори");
   else
   {
       printf("Неоднакові вектори");
   }
   return 0;
}
```

Результат

```
Неоднакові вектори
```

Приклад алгоритму equal (другий варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <functional>

using namespace std;

bool square(int n1, int n2)
{
   return n1*n1==n2?true:false;
}

int main()
{
   int initial[3] = {1,2,3};
   int pattern[3] = {1,4,9};
   int i;
   vector<int> first (initial, initial+3), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   if(equal(first.begin(), first.end(), second.begin(), square))
       printf("Однакові вектори");
   else
   {
       printf("Неоднакові вектори");
   }
   return 0;
}
```

Результат

```
Однакові вектори
```

14.1.11. Алгоритм search

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
  ForwardIterator1 search(ForwardIterator1 first1, ForwardIterator1 last1,
                          ForwardIterator2 first2, ForwardIterator2 last2,
                         BinaryPredicate pred);
```

Виконує пошук підпослідовності, обмеженої ітераторами `first2` і `last2`, елементи якої збігаються з елементами діапазону `[first1, last1)`. Повертає ітератор, установлений на перший елемент шуканої підпослідовності, т.е перший ітератор з діапазону `[first1, last1- (last2- first2))`, що задовольняє умові `*(i+n) == *(first2+n)` для всех `0<n<last2-first2`. Якщо підпослідовність не знайдена, повертається ітератор `last1`.

Друга версія алгоритму дозволяє задавати бінарний предикат, що визначає умову збігу елементів: `pred(*(i+n), *(first2+n)) != false`.

Приклад алгоритму search (перший варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[3] = {4,5,6};
   int i;
```

```
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   vector<int>::iterator where;
   where = search(first.begin(), first.end(), second.begin(), second.end());
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
  1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
  4   5   6
Шукане число: 4
Позиція: 9
```

Приклад алгоритму search (другий варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

bool IsEqual (int n1, int n2)
{
   return n1*n1 == n2? true: false;
}

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int pattern[3] = {25,25,36};
   int i;
   vector<int> first (initial, initial+15), second(pattern,pattern+3);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");
   for (i = 0; i < second.size(); ++i)
       printf("%3d ",second[i]);
       printf("\n");

   vector<int>::iterator where;
   where = search(first.begin(), first.end(), second.begin(),
                  second.end(), IsEqual);
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
  1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
 25  25  36
Шукане число: 5
Позиція: 3
```

14.1.12. Алгоритм search_n

```
template<class ForwardIterator, class Size, class T>
  ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                           Size count, const T& value);

template<class ForwardIterator, class Size, class T, class BinaryPredicate>
  ForwardIterator1 search_n(ForwardIterator first, ForwardIterator last,
                            Size count, const T& value, BinaryPredicate pred);
```

Виконує пошук підпослідовності, що складає з `count` елементів, що мають значення `value`, у діапазоні `[first,last)`. Повертає ітератор, установлений на ее початок, тобто ітератор, що задовольняє умові `*(i+n) ==value` для всех `0<n<count`. У випадку невдачі повертається ітератор `last`.

Друга версія алгоритму дозволяє задавати бінарний предикат, що визначає умови збігу елементів: `pred(*(i+n),value) != false` для всех `0<n<count`.

Приклад алгоритму search_n (перший варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,1,2,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");

   vector<int>::iterator where;
   // Шукаємо підпослідовність, що складається з двох п'ятірок
   where = search_n(first.begin(), first.end(), 2,5);
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
1   2   3   5   5   6   1   2   3   4   5   6   3   4   5
Шукане число: 5
Позиція: 3
```

Приклад алгоритму search_n (другий варіант)

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

bool twice (int element, int value)
{
   return element == value*2? true: false;
}

int main()
{
   int initial[15] = {1,2,3,5,5,6,10,10,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15);
   for (i = 0; i < first.size(); ++i)
       printf("%3d ",first[i]);
       printf("\n");

   vector<int>::iterator where;
   // Шукаємо підпослідовність, що складається з двох десяток (10 = 2*5)
   where = search_n(first.begin(), first.end(), 2,5, twice);
   printf("Шукане число: %d\n",*where);
   printf("Позиція: %d\n",where-first.begin());
   return 0;
}
```

Результат

```
1   2   3   5   5   6  10  10   3   4   5   6   3   4   5
Шукане число: 10
Позиція: 6
```

14.2. Алгоритми, що модифікують

Перейдемо до опису другої категорії алгоритмів.

14.2.1. Алгоритм copy

```
template<class InputIterator, class OutputIterator>
  OutputIterator copy(InputIterator first, InputIterator last,
                      OutputIterator result);
```

Копіює діапазон `[first,last)` у діапазон `[result,result+(last- first))`, виконуючи операцію `*(result+n)=*(first+n)` для всех `0<n<last-first`. Повертає ітератор `result+(last- first)`. Ітератор `result` не повинний належати діапазону `[first,last)`.

Приклад алгоритму copy

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,10,10,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15), second(15);
   for (i = 0; i < first.size(); i++)
       printf("%3d ",first[i]);
       printf("\n");
   copy(first.begin(),first.end(),second.begin());
   for (i = 0; i < second.size(); i++)
       printf("%3d ",second[i]);
       printf("\n");

   return 0;
}
```

Результат

```
  1   2   3   5   5   6  10  10   3   4   5   6   3   4   5
  1   2   3   5   5   6  10  10   3   4   5   6   3   4   5
```

14.2.2. Алгоритм copy_backward

```
template<class BidirectionalIterator1, class BidirectionalIterator2>
  BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
                                       BidirectionalIterator1 last,
                                       BidirectionalIterator2 result);
```

Копіює діапазон `[first,last)` у діапазон `[result- (last- first),result)`, починаючи з ітератора `last-1` і закінчуючи ітератором `first`, за допомогою операції `*(result- n)=*(first- n)` для всех `0<n=<last-first`. Повертає ітератор `result+(last- first)`. Ітератор `result` не повинний належати діапазону `[first,last)`.

Приклад алгоритму copy_backward

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main()
{
   int initial[15] = {1,2,3,5,5,6,10,10,3,4,5,6,3,4,5};
   int i;
   vector<int> first (initial, initial+15), second(15);
   for (i = 0; i < first.size(); i++)
       printf("%3d ",first[i]);
       printf("\n");
   copy_backward(first.begin(),first.end(),second.end());
   for (i = 0; i < second.size(); i++)
       printf("%3d ",second[i]);
       printf("\n");
```

```
   return 0;
}
```

Результат

```
 1   2   3   5   5   6  10  10   3   4   5   6   3   4   5
 1   2   3   5   5   6  10  10   3   4   5   6   3   4   5
```

14.2.3. Алгоритм swap

```
template<class T>
  void swap(T& a, T& b);
```

Змінює местами значення, на які встановлені ітераторы a і b.

Приклад алгоритму swap

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   int initial1[5] = {1, 2, 3, 4, 5};
   int initial2[5] = {2, 4, 6, 8, 10};
   int i;
   vector<int> first(initial1, initial1+5), second(initial2,initial2+5);

    // Змінюємо масиви местами, обмінюючи елементи
   for(i=0;i<first.size();i++)
   swap(first[i],second[i]);

   for(i=0;i<first.size();i++)
   printf("%d ",first[i]);
   printf("\n");

   for(i=0;i<second.size();i++)
   printf("%d ",second[i]);
   printf("\n");

   return 0;
}
```

Результат

```
2 4 6 8 10
1 2 3 4 5
```

14.2.4. Алгоритм swap_ranges

```
template<class ForwardIterator1, class ForwardIterator2>
  ForwardIterator2 swap_ranges(ForwardIterator1 first1,
                               ForwardIterator1 last1,
                               ForwardIterator2 first2);
```

Змінює местами елементи діапазонів [first1,last1) і [first2,last2), виконуючи операцію swap(*(first1+n),*(first2+n)). Діапазони [first1,last1) і [first2,last2) не повинні перекриватися.

Приклад алгоритму swap_ranges

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   int initial1[5] = {1, 2, 3, 4, 5};
   int initial2[5] = {2, 4, 6, 8, 10};
   int i;
   vector<int> first(initial1, initial1+5), second(initial2,initial2+5);
```

```
   swap_ranges(first.begin(),first.end(),second.begin());

   for(i=0;i<first.size();i++)
   printf("%d ",first[i]);
   printf("\n");

   for(i=0;i<second.size();i++)
   printf("%d ",second[i]);
   printf("\n");

   return 0;
}
```

Результат

```
2 4 6 8 10
1 2 3 4 5
```

14.2.5. Алгоритм iter_swap

```
template<class ForwardIterator1, class ForwardIterator2>
  void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Змінює местами значення, на які встановлені ітераторы a і b.

Приклад алгоритму iter_swap

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   int initial1[5] = {1, 2, 3, 4, 5};
   int initial2[5] = {2, 4, 6, 8, 10};
   int k;
   vector<int> first(initial1, initial1+5), second(initial2,initial2+5);

   vector<int>::iterator i=first.begin();
   vector<int>::iterator j=second.begin();

   do{
   iter_swap(i,j);
   i++;
   j++;
   } while (i != first.end() && j != second.end());

   for(k=0;k<first.size();k++)
   printf("%d ",first[k]);
   printf("\n");

   for(k=0;k<second.size();k++)
   printf("%d ",second[k]);
   printf("\n");

   return 0;
}
```

Результат

```
2 4 6 8 10
1 2 3 4 5
```

14.2.6. Алгоритм transform

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
  OutputIterator transform(InputIterator first, InputIterator last,
                           OutputIterator result, UnaryOperation op);

template<class InputIterator1, class InputIterator2,
        class OutputIterator, class BinaryOperation>
```

```
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

Перший варіант алгоритму `transform` застосовує унарную операцію `op(*(first+(i- result))` для каждого ітератора `i` з діапазону елементів `[result,result+(last1-first1))`, тобто до разыменованным значень кожного ітератора з діапазону `[first,last)`. Повертає ітератор `result+(last1-first1)`.

Другий варіант алгоритму виконує операцію `binary_op(*(first1+(i- result),*(first2+(i-result)))`.

Приклад алгоритму transform (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int square(int n)
{
   return n*n;
}

int main()
{
   const int SIZE = 5;
   int first [5] = { 1, 2, 3, 4, 5};
   int second[5];

   transform (first, first + SIZE, second, ptr_fun(square));

   for (int i = 0; i < SIZE; i++)
     printf("%d ", second[i]);
     printf("\n");
   return 0;
}
```

Результат

```
1 4 9 16 25
```

Приклад алгоритму transform (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 5;
   int first [5] = { 1, 2, 3, 4, 5};
   int second[5] = { 10, 20, 30, 40, 50};
   int result[5];

   transform (first, first + SIZE, second, result, multiplies<int>());

   for (int i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");
   return 0;
}
```

Результат

```
10 40 90 160 250
```

14.2.7. Алгоритм replace

```
template<class ForwardIterator, class T>
  void replace(ForwardIterator first, ForwardIterator last,
               const T& old_value, const T& new_value);
```

Заміняє елементи, що мають значення old_value, на які посилаються ітераторы з діапазону [first,last), елементами, що мають значення new_value.

Приклад алгоритму replace

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int array [10] = { 1, 5, 3, 4, 5, 5, 7, 10, 3, 14};
   int i;

   printf("До заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Заменяем п'ятірки нулями
   replace(array, array + SIZE, 5,0);

   printf("Після заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");
   return 0;
}
```

Результат

```
До заміни:
1 5 3 4 5 5 7 10 3 14
Після заміни:
1 0 3 4 0 0 7 10 3 14
```

14.2.8. Алгоритм replace_if

```
template<class ForwardIterator, class Predicate, class T>
  void replace_if(ForwardIterator first, ForwardIterator last,
                  Predicate pred, const T& new_value);
```

Привласнює елементам контейнера, на які посилаються ітераторы i з діапазону [first,last), значення new_value, якщо виконується умова pred(*i)!=false.

Приклад алгоритму replace_if

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int array [10] = { 1, 5, 3, 4, 5, 5, 7, 10, 3, 14};
   int i;

   printf("До заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   replace_if(array, array + SIZE, bind2nd (less<int> (), 6), 0);

   printf("Після заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");
```

```
   return 0;
}
```

Результат

```
До заміни:
1 5 3 4 5 5 7 10 3 14
Після заміни:
0 0 0 0 0 0 7 10 0 14
```

14.2.9. Алгоритм replace_copy

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator replace_copy(InputIterator first, InputIterator last,
                              OutputIterator result, const T& old_value,
                              const T& new_value);
```

Копіює елементи, на які посилаються ітераторы з діапазону `[first,last)`, у послідовність, адресуемую ітератором *result*, заміняючи елементи, що мають значення `old_value`, елементами, що мають значення `new_value`. Вихідний діапазон не змінюється. Повертається ітератор `result+(last-first)`.

Приклад алгоритму replace_copy

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int array [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int result[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
   int i;

   printf("До заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

    // Заміняємо всі п'ятірки числом 50
   replace_copy(array, array + SIZE, result, 5, 50);

   printf("Після заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");
   return 0;
}
```

Результат

```
До заміни:
1 2 3 4 5 6 7 8 9 10
Після заміни:
1 2 3 4 50 6 7 8 9 10
```

14.2.10. Алгоритм replace_copy_if

```
template<class InputIterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(InputIterator first, InputIterator last,
                               OutputIterator result, Predicate pred,
                               const T& new_value);
```

Копіює елементи, на які посилаються ітераторы з діапазону `[first,last)`, у послідовність, адресуемую ітератором *result*, заміняючи елементи, що задовольняють предикат `pred`, елементами, що мають значення `new_value`. Вихідний діапазон не змінюється. Повертається ітератор `result+(last-first)`.

Приклад алгоритму replace_copy_if

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;
```

```
bool odd(int n)
{
   return n%2?true:false;
}

int main()
{
   const int SIZE = 10;
   int array [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int result[10] = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20};
   int i;

   printf("До заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Заміняємо всі непарні елементи нулями
   replace_copy_if(array, array + SIZE, result, odd, 0);

   printf("Після заміни: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");
   return 0;
}
```

Результат

```
До заміни:
1 2 3 4 5 6 7 8 9 10
Після заміни:
0 2 0 4 0 6 0 8 0 10
```

14.2.11. Алгоритм fill

```
template<class ForwardIterator, class T>
  void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Привласнює значення `value` всім елементам контейнера, на які посилаються ітератори з діапазону `[first, last)`.

Приклад алгоритму fill

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int array [10];

   fill(array, array+SIZE, 1);
   printf("Масив: \n");
   for (int i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив:
1 1 1 1 1 1 1 1 1 1
```

14.2.12. Алгоритм fill_n

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

Привласнює значення `value` першим `n` елементам контейнера, на які посилаються ітераторы з діапазону `[first, last)`.

Приклад алгоритму fill_n

```
#include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   const int N   = 5;
   const int NEW = 0;
   int i;
   int array [SIZE]={1,2,3,4,5,6,7,8,9,10};

   printf("Масив до заповнення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   fill_n(array, N, NEW);
   printf("Масив після заповнення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до заповнення:
1 2 3 4 5 6 7 8 9 10
Масив після заповнення:
0 0 0 0 0 6 7 8 9 10
```

14.2.13. Алгоритм generate

```
template<class ForwardIterator, class Generator>
  void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

Привласнює елементам контейнера, на які посилаються ітераторы з діапазону `[first,last)`, значення, що повертається функцією `gen()`, що не має параметрів.

Приклад алгоритму generate

```
#include <iostream>
#include <algorithm>
#include <time.h>

using namespace std;

int random()
{
   static long k = clock();
   k++;
   srand(k);      // Стартова крапка генератора
   return rand(); // Випадкове число
}

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE];

   generate(array, array+SIZE, random);
   printf("Заповнений масив: \n");
   for (i = 0; i < SIZE; i++)
```

```
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив після заповнення:
21448 5518 22356 6427 23265 7335 24174 8244 25082 9153
```

14.2.14. Алгоритм generate_n

```
template<class OutputIterator, class Size, class Generator>
  void generate_n(OutputIterator first, Size n, Generator gen);
```

Привласнює елементам контейнера, на які посилаються ітераторы з діапазону `[first,first+n)`, значення, що повертається функцією `gen()`, що не має параметрів.

Приклад алгоритму generate_n

```
#include <iostream>
#include <algorithm>
#include <time.h>

using namespace std;

int random()
{
  static long k = clock();
  k++;
  srand(k);       // Стартова крапка генератора
  return rand();  // Випадкове число
}

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,5,6,7,8,9,10};

   printf("Масив до заповнення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Заполняем перші п'ять позицій випадковим числом

   generate_n(array, 5, random);
   printf("Масив після заповнення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до заповнення:
1 2 3 4 5 6 7 8 9 10
Масив після заповнення:
15267 32106 16176 246 17085 6 7 8 9 10
```

14.2.15. Алгоритм remove

```
template<class ForwardIterator, class T>
  ForwardIterator remove(ForwardIterator first, ForwardIterator last,
                         const T& value);
```

Видаляє з контейнера всі елементи, що мають значення `value`, на які посилаються ітераторы, що належать діапазону `[first,last)`. Повертає ітератор, установлений на кінець результуючого діапазону.

Приклад алгоритму remove

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,5,6,5,8,5,10};

   printf("Масив до видалення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Удаляем з масиву п'ятірки

   int* where = remove(array, array+SIZE, 5);

   printf("Масив після видалення: \n");
   for (i = 0; i < where-array; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до видалення:
1 2 3 4 5 6 5 8 5 10
Масив після видалення:
1 2 3 4 6 8 10
```

14.2.16. Алгоритм remove_if

```
template<class ForwardIterator, class Predicate>
  ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
                            Predicate pred);
```

Видаляє з контейнера елементи, на які посилаються ітераторы з діапазону `[first,last)` і для який є щирим предикат `pred`. Повертає ітератор, установлений на останній з елементів, що залишилися.

Приклад алгоритму remove_if

```
#include <iostream>
#include <algorithm>

using namespace std;

bool odd(int n)
{
   return n%2?true:false;
}

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,5,6,5,8,5,10};

   printf("Масив до видалення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Удаляем з масиву непарні числа

   int* where = remove_if(array, array+SIZE, odd);

   printf("Масив після видалення: \n");
```

```
   for (i = 0; i < where-array; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до видалення:
1 2 3 4 5 6 5 8 5 10
Масив після видалення:
2 4 6 8 10
```

14.2.17. Алгоритм remove_copy

```
template<class InputIterator, class OutputIterator, class T>
  OutputIterator remove_copy(InputIterator first, InputIterator last,
                             OutputIterator result, const T& value);
```

Копіює елементи контейнера, що мають значення `value`, на які посилаються ітераторы, що належать діапазону `[first,last)`, у послідовність, адресуемую ітератором `result`. Повертається ітератор, установлений на останній зі скопійованих елементів у діапазоні `result`.

Приклад алгоритму remove_copy

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,5,6,5,8,5,10};
   int result[SIZE]={-1,-2,-3,-4,-5,-6,-7,-8,-9,-10};

   printf("Масив до копіювання: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Копіюємо п'ятірки в масив result

   int* where = remove_copy(array, array+SIZE, result, 5);

   printf("Масив після копіювання: \n");
   for (i = 0; i < where-result; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до копіювання:
1 2 3 4 5 6 5 8 5 10
Масив після копіювання:
1 2 3 4 6 8 10
```

14.2.18. Алгоритм remove_copy_if

```
template<class InputIterator, class OutputIterator, class Predicate>
  OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                                OutputIterator result, Predicate pred);
```

Копіює елементи контейнера, для яких є щирим предикат `pred` і на який посилаються ітераторы, що належать діапазону `[first,last)`, у послідовність, адресуемую ітератором `result`. Повертається ітератор, установлений на останній зі скопійованих елементів у діапазоні `result`.

Приклад алгоритму remove_copy_if

```
#include <iostream>
#include <algorithm>
```

```
using namespace std;

bool odd(int n)
{
   return n%2?true:false;
}

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,5,6,5,8,5,10};
   int result[SIZE]={-1,-2,-3,-4,-5,-6,-7,-8,-9,-10};

   printf("Масив до копіювання: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Копіюємо парні елементи в масив result

   int* where = remove_copy_if(array, array+SIZE, result, odd);

   printf("Масив після копіювання: \n");
   for (i = 0; i < where-result; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до копіювання:
1 2 3 4 5 6 5 8 5 10
Масив після копіювання:
2 4 6 8 10
```

14.2.19. Алгоритм unique

```
template<class ForwardIterator>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
  ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                         BinaryPredicate pred);
```

Перший варіант алгоритму виключає сусідні дублікати елементів, на які посилаються ітераторы з діапазону `[first,last)`.

Другий варіант дозволяє задавати бінарний предикат порівняння. Умова виключення елемента, на який посилається ітератор `i`, має наступний вид: `pred(*i, *(i-1)) != false`.

Алгоритм `unique` повертає ітератор, установлений на кінець результуючого діапазону.

Приклад алгоритму unique

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,5,5,5,5,5,5,10};

   printf("Масив до видалення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");
```

```
   // Удаляем зайві п'ятірки з масиву

   int* where = unique(array, array+SIZE);

   printf("Масив після видалення: \n");
   for (i = 0; i < where-array; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до видалення:
1 2 3 5 5 5 5 5 5 10
Масив після видалення:
1 2 3 5 10
```

14.2.20. Алгоритм unique_copy

```
template<class InputIterator, class OutputIterator>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
  OutputIterator unique_copy(InputIterator first, InputIterator last,
                             OutputIterator result, BinaryPredicate pred);
```

Перша версія алгоритму копіює елементи контейнера, на які посилаються ітераторы з діапазону `[first1,last1)`, у послідовність `[result, result+(last- first))`, крім сусідніх дублікатів.

Друга версія дозволяє задавати бінарний предикат порівняння. Умова виключення дубліката, на який посилається ітератор `i`, має наступний вид: `pred(*i,*(i-1)) != false`.

Алгоритм `unique_copy` повертає ітератор, установлений на кінець результуючого діапазону.

Приклад алгоритму unique_copy (перший варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,5,5,5,5,5,5,10};
   int result[SIZE]={0,0,0,0,0,0,0,0,0,0};

   printf("Массиив до копіювання: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Удаляем зайві п'ятірки з масиву

   int* where = unique_copy(array, array+SIZE, result);

   printf("Масив після копіювання: \n");
   for (i = 0; i < where-result; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до копіювання:
1 2 3 5 5 5 5 5 5 10
Масив після копіювання:
1 2 3 5 10
```

Приклад алгоритму unique_copy (другий варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

bool twice(int n1, int n2)
{
   return 2*n1==n2?true:false;
}

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};
   int result[SIZE]={0,0,0,0,0,0,0,0,0,0};

   printf("Масив до копіювання: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Удаляем сусідні дублікати, якщо друге число вдвічі більше першого
    int* where = unique_copy(array, array+SIZE, result, twice);

   printf("Масив після копіювання: \n");
   for (i = 0; i < where-result; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до копіювання:
1 2 3 4 8 6 12 8 9 10
Масив після копіювання:
1 3 4 6 8 9 10
```

14.2.21. Алгоритм reverse

```
template<class BidirectionalIterator>
  void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Змінює порядок проходження елементів, на який посилаються ітераторы з діапазону `[first,last)`, на протилежний.

Приклад алгоритму reverse

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};

   printf("Масив до перетворення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Змінюємо порядок на протилежний

   reverse(array, array+SIZE);
```

```
   printf("Масив після перетворення: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Масив до перетворення:
1 2 3 4 8 6 12 8 9 10
Маиссв після перетворення:
10 9 8 12 6 8 4 3 2 1
```

14.2.22. Алгоритм reverse_copy

```
template<class BidirectionalIterator, class OutputIterator>
  OutputIterator reverse_copy(BidirectionalIterator first,
                              BidirectionalIterator last,
                              OutputIterator result);
```

Змінює порядок проходження елементів, на які посилаються ітераторы з діапазону `[first,last)`, на протилежний, копіюючи результат у діапазон `[result, result+(last- first))`. Повертає ітератор, установлений на кінець результуючої послідовності.

Приклад алгоритму reverse_copy

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};
   int result[SIZE]={0,0,0,0,0,0,0,0,0,0};

   printf("Вихідний масив: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Змінюємо порядок на протилежний

   reverse_copy(array, array+SIZE,result);

   printf("Зворотна копія: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
1 2 3 4 8 6 12 8 9 10
Зворотна копія:
10 9 8 12 6 8 4 3 2 1
```

14.2.23. Алгоритм rotate

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last);
```

Для кожного ненегативного цілого числа `i`, що задовольняє умові `i < last- first`, заміняє елемент, що знаходиться в позиції `first+i`, елементом, записаним у позиції `first+(i+(last- middle))%(last-first)`. Таке перетворення називається *лівим обертанням*.

Приклад алгоритму rotate

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};

   printf("Вихідний масив: \n");
   for (i = 0; i < SIZE; i++)
     printf(" %d ", array[i]);
     printf("\n");

   // Выполняем ліве обертання

   rotate(array, array+SIZE/2,array+SIZE);

   printf("Ліве обертання: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
1 2 3 4 8 6 12 8 9 10
Ліве обертання:
6 12 8 9 10 1 2 3 4 8
```

14.2.24. Алгоритм rotate_copy

```
template<class ForwardIterator, class OutputIterator>
  OutputIterator rotate_copy(ForwardIterator first,
                             ForwardIterator middle,
                             ForwardIterator last,
                             OutputIterator result);
```

Виконує ліве обертання елементів контейнера, на які посилаються ітераторы з діапазону `[first,last)`, і копіює результат у послідовність `[result, result+(last- first))`. Повертає ітератор, установлений на кінець результуючої послідовності.

Приклад алгоритму rotate_copy

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};
   int result[SIZE]={0,0,0,0,0,0,0,0,0,0};

   printf("Вихідний масив: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Змінюємо порядок на протилежний

   rotate_copy(array, array+SIZE/2,array+SIZE,result);

   printf("Ліве обертання: \n");
   for (i = 0; i < SIZE; i++)
```

```
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
1 2 3 4 8 6 12 8 9 10
Ліве обертання:
6 12 8 9 10 1 2 3 4 8
```

14.2.25. Алгоритм random_shuffle

```
template<class RandomAccessIterator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last);

template<class RandomAccessIterator, class RandomNumberGenerator>
  void random_shuffle(RandomAccessIterator first,
                      RandomAccessIterator last,
                      RandomNumberGenerator& rand);
```

Перетасовує елементи, на які посилаються ітераторы з діапазону `[first,last)`, використовуючи генератор рівномірно розподілених випадкових чисел.

Друга версія алгоритму дозволяє задавати генератор випадкових чисел `rand`. Функція `rand()` повинна одержувати як аргумент позитивне ціле число `n`, що має тип `iterator_traits<RandomAccessIterator>::difference_type`, а повертати — випадкове число з інтервалу `[0, n-1)`.

Приклад алгоритму random_shuffle (перший варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};

   printf("Вихідний масив: \n");
   for (i = 0; i < SIZE; i++)
     printf(" %d ", array[i]);
     printf("\n");

   // Перетасовываем елементи

   random_shuffle(array, array+SIZE);

   printf("Перетасований масив: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
1 2 3 4 8 6 12 8 9 10
Перетасований масив:
8 4 1 3 12 8 9 10 6 2
```

Приклад алгоритму random_shuffle (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <stdlib.h>
```

```
#include <time.h>

using namespace std;

class RandomGenerator
{
public:
    int operator() (int n)
    {
        srand( (unsigned)time( NULL ) );
        return rand() % n;
    }
};

int main()
{
   const int SIZE = 10;
   int i;
   int array [SIZE]={1,2,3,4,8,6,12,8,9,10};
   RandomGenerator objGen;

   printf("Вихідний масив: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Перетасовываем елементи

   random_shuffle(array, array+SIZE, objGen);

   printf("Перетасований масив: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
1 2 3 4 8 6 12 8 9 10
Перетасований масив:
8 4 1 3 12 8 9 10 6 2
```

14.2.26. Алгоритм partition

```
template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator partition(BidirectionalIterator first,
                                  BidirectionalIterator last,
                                  Predicate pred);

template<class BidirectionalIterator, class Predicate>
  BidirectionalIterator stable_partition(BidirectionalIterator first,
                                         BidirectionalIterator last,
                                         Predicate pred);
```

Обидві версії упорядковують послідовність, обмежену ітераторами `first` і `last`, щоб всі елементи, для яких унарный предикат `pred` є щирим, передували тим, для яких цей предикат помилковий. Повертають ітератор, установлений на початок елементів, для яких предикат є помилковим. Алгоритм `stable_partition` дотримує первісного порядку в групах елементів.

Приклад алгоритму partition

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;
```

```
int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи вектора
   partition(array.begin(), array.end(), bind2nd(less<int>(), 7));

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
5 4 1 3 7 8 9 10 6 2
Упорядкований вектор:
5 4 1 3 2 6 9 10 8 7
```

Примітка: виділені елементи менше семи, а інші — чи більше рівні семи.

Приклад алгоритму stable_partition

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи
   stable_partition(array.begin(), array.end(), bind2nd(less<int>(), 7));

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
5 4 1 3 7 8 9 10 6 2
```

```
Упорядкований вектор:
5 4 1 3 6 2 7 8 9 10
```

14.3. Алгоритми сортування і пошуку

Третя група узагальнених алгоритмів складається із шаблонових функцій, що реалізують процедури сортування і пошуку, а також зв'язані з ними операції.

14.3.1. Алгоритм sort

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first,
            RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first,
            RandomAccessIterator last,
            Compare comp);
```

Перша версія цього алгоритму упорядковує елементи в діапазоні `[first, last)`.

Друга версія дозволяє задати функцію порівняння елементів, тобто порядок проходження елементів в упорядкованому контейнері.

Приклад алгоритму sort (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи
   sort(array.begin(), array.end());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
5 4 1 3 7 8 9 10 6 2
Упорядкований вектор:
1 2 3 4 5 6 7 8 9 10
```

Приклад алгоритму sort (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
```

```
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи в порядку убування
   sort(array.begin(), array.end(), greater<int>());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
5 4 1 3 7 8 9 10 6 2
Упорядкований вектор:
10 9 8 7 6 5 4 3 2 1
```

14.3.2. Алгоритм stable_sort

```
template<class RandomAccessIterator>
  void stable_sort(RandomAccessIterator first,
                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void stable_sort(RandomAccessIterator first,
                   RandomAccessIterator last,
                   Compare comp);
```

Перша версія упорядковує елементи в діапазоні `[first, last)`, не змінюючи местами еквівалентні елементи.

Друга версія дозволяє задати функцію порівняння елементів.

Приклад алгоритму stable_sort (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи
   stable_sort(array.begin(), array.end());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
```

```
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Упорядкований вектор:
1 1 3 4 4 5 5 6 7 10
```

Приклад алгоритму stable_sort (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи по убуванню
   stable_sort(array.begin(), array.end(), greater<int>());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Упорядкований вектор:
10 7 6 5 5 4 4 3 1 1
```

14.3.3. Алгоритм partial_sort

```
template<class RandomAccessIterator>
  void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
 void partial_sort(RandomAccessIterator first,
                    RandomAccessIterator middle,
                    RandomAccessIterator last,
                     Compare comp);
```

Перша версія упорядковує елементи в діапазоні `[first, last)`, але після виконання алгоритму упорядкованими виявляються лише елементи в діапазоні `[first,middle)`. Друга версія дозволяє задати функцію порівняння `comp()`.

Приклад алгоритму partial_sort (перший варіант)

```
#include <iostream>
#include <algorithm>
```

```
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем елементи
   partial_sort(array.begin(), array.begin()+5, array.end());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Упорядкований вектор:
1 1 3 4 4 7 5 10 6 5
```

Приклад алгоритму partial_sort (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем вектор по убуванню
   partial_sort(array.begin(), array.begin()+5, array.end(), greater<int>());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Упорядкований вектор:
10 7 6 5 5 1 3 4 4 1
```

14.3.4. Алгоритм partial_sort_copy

```
template<class InputIterator, class RandomAccessIterator>
  RandomAccessIterator partial_sort_copy(InputIterator first,
                                         InputIterator last,
                                          RandomAccessIterator result_first,
                                         RandomAccessIterator result_last);

template<class InputIterator, class RandomAccessIterator, class Compare>
  RandomAccessIterator partial_sort_copy(InputIterator first,
                                         InputIterator last,
                                          RandomAccessIterator result_first,
                                         RandomAccessIterator result_last,
                                         Compare comp);
```

Перша версія копіює `min(last-first,result_last-result_first)` упорядкованих елементів у діапазон `(result_first, result_first+min(last-first,result_last-result_first))`. Повертає найменший серед двох ітераторов: `min(result_last, result_last+(last-first))`.

Друга версія дозволяє задати функцію порівняння.

Приклад алгоритму partial_sort_copy (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);
   vector<int> result(10);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Копіюємо й упорядковуємо вектор
   partial_sort_copy(array.begin(), array.end(),
                     result.begin(), result.end());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Упорядкований вектор:
1 1 3 4 4 5 5 6 7 10
```

Приклад алгоритму partial_sort_copy (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>
```

```
using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);
   vector<int> result(10);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Упорядочиваем вектор по убуванню
   partial_sort_copy(array.begin(), array.end(),
                     result.begin(), result.end(), greater<int>());

   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", result[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований масив:
4 4 1 3 7 5 5 10 6 1
Упорядкований масив:
10 7 6 5 5 4 4 3 1 1
```

14.3.5. Алгоритм nth_element

```
template<class RandomAccessIterator>
  void nth_element(RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void nth_element(RandomAccessIterator first,
                   RandomAccessIterator nth,
                   RandomAccessIterator last,
                    Compare comp);
```

Перша версія упорядковує елементи в діапазоні `[first,last)` так, щоб елементи, що не перевищують значення параметра `nth`, передували йому, а елементи, що більше значення параметра *element*, випливали за ним. Таким чином, після виконання алгоритму елемент `nth` виявляється на тім місці, на якому він був би в цілком упорядкованому контейнері. Друга версія дозволяє задати функцію порівняння елементів.

Приклад алгоритму nth_element (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);
   vector<int> result(10);
```

```
   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Вычисляем місце n-го елемента
   nth_element(array.begin(), array.begin()+3, array.end());

   printf("Після перестановки: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Після перестановки:
1 1 3 4 4 5 5 6 7 10
```

Примітка. n-м елементом є число 3. В упорядкованому масиві перед ним розташовані числа, менше 3, а за ним — більше 3.

Приклад алгоритму nth_element (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,1,3,4,4,6,7,5,5,10};
   vector<int> array(initial,initial+SIZE);
   vector<int> result(10);

   random_shuffle(array.begin(),array.end());

   printf("Перетасований вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Вычисляем місце n-го елемента у векторі, упорядкованому по убуванню
   nth_element(array.begin(), array.begin()+3, array.end(),greater<int>());

   printf("Після перестановки: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   return 0;
}
```

Результат

```
Перетасований вектор:
4 4 1 3 7 5 5 10 6 1
Після перестановки:
10 7 6 5 5 4 4 3 1 1
```

Примітка. n-м елементом є число 3. В упорядкованому масиві перед ним розташовані числа, більше 3, а за ним — менше 3.

14.3.6. Алгоритм lower_bound

```
template<class ForwardIterator, class T>
  ForwardIterator lower_bound(ForwardIterator first,
                              ForwardIterator last,
                              const T& value);

template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first,
                            ForwardIterator last,
                            const T& value,
                            Compare comp);
```

Перша версія повертає ітератор, установлений на позицію, у яку можна вставити параметр `value`, не порушуючи порядок проходження елементів у діапазоні `[first, last)`. Друга версія дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму lower_bound (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,5,5,9,9,10};
   vector<int> array(initial,initial+SIZE);

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Обчислюємо позицію для вставки

   vector<int>::iterator where;
   where = lower_bound(array.begin(), array.end(), 5);

   printf("Позиція для вставки: %d\n",where-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
1 2 3 4 5 5 5 9 9 10
Позиція для вставки: 4
```

Приклад алгоритму lower_bound (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {10, 9, 8, 7, 6, 5, 5, 4, 3, 2};
   vector<int> array(initial,initial+SIZE);

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
```

```
     printf("%d ", array[i]);
     printf("\n");

   // Обчислюємо позицію для вставки

   vector<int>::iterator where;
   where = lower_bound(array.begin(), array.end(), 5,greater<int>());

   printf("Позиція для вставки: %d\n",where-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
10 9 8 7 6 5 5 4 3 2
Позиція для вставки: 5
```

14.3.7. Алгоритм upper_bound

```
template<class ForwardIterator, class T>
  ForwardIterator upper_bound(ForwardIterator first,
                              ForwardIterator last,
                              const T& value);

template<class ForwardIterator, class T, class Compare>
  ForwardIterator upper_bound(ForwardIterator first,
                              ForwardIterator last,
                              const T& value,
                              Compare comp);
```

Перша версія виконує пошук останнього елемента в діапазоні `[first, last)`, значення якого не перевищує значення параметра `value`. Це дозволяє вставити значення `value`, не порушуючи порядок проходження елементів. Повертає ітератор, установлений на шуканий елемент.

Друга версія дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму upper_bound (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {1,2,3,4,5,5,5,9,9,10};
   vector<int> array(initial,initial+SIZE);

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Обчислюємо позицію для вставки

   vector<int>::iterator where;
   where = upper_bound(array.begin(), array.end(), 5);

   printf("Позиція для вставки: %d\n",where-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
1 2 3 4 5 5 5 9 9 10
```

```
Позиція для вставки: 7
```

Приклад алгоритму upper_bound (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {10, 9, 8, 7, 6, 5, 5, 4, 3, 2};
   vector<int> array(initial,initial+SIZE);

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Обчислюємо позицію для вставки

   vector<int>::iterator where;
   where = upper_bound(array.begin(), array.end(), 5,greater<int>());

   printf("Позиція для вставки: %d\n",where-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
10 9 8 7 6 5 5 4 3 2
Позиція для вставки: 7
```

14.3.8. Алгоритм equal_range

```
template<class ForwardIterator, class T>
  pair<ForwardIterator, ForwardIterator>
  equal_range(ForwardIterator first,
              ForwardIterator last,
              const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator>
   equal_range(ForwardIterator first,
               ForwardIterator last,
               const T& value,
               Compare comp);
```

Знаходить у діапазоні `[first,last)` найбільшу підпослідовність `[i,j)`, що допускає вставку значення `value` без порушення порядку. Нехай `k` — ітератор, установлений на новий елемент. Вставка виконується, якщо справедлива умова `!(*k<value)&&!(value<*k)`

Друга версія дозволяє програмісту самому задати функцію порівняння, що визначає критерій пошуку. Вставка виконується, якщо справедлива умова `comp(*k,value)==false&&comp(value,*k)==false`.

Алгоритм повертає пару, що містить ітераторы, установлені на початок і кінець шуканого діапазону.

Приклад алгоритму equal_range (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
```

```
   int i;
   int initial[SIZE] = {1, 2, 3, 4, 5, 5, 5, 6, 7, 8};
   vector<int> array(initial,initial+SIZE);
   pair<int*,int*> range;

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Вычисляем діапазон для вставки

   range = equal_range(array.begin(), array.end(), 5);

   printf("Перша позиція для вставки:      %d\n", range.first-array.begin());
   printf("Остання  позиція для вставки: %d\n", range.second-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
1 2 3 4 5 5 5 6 7 8
Перша позиція для вставки:    4
Остання позиція для вставки: 7
```

Приклад алгоритму equal_range (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 10;
   int i;
   int initial[SIZE] = {10, 9, 5, 5, 5, 4, 3, 2, 1, 0};
   vector<int> array(initial,initial+SIZE);
   pair<int*,int*> range;

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Вычисляем діапазон для вставки

   range = equal_range(array.begin(), array.end(), 5, greater<int>());

   printf("Перша позиція для вставки:      %d\n", range.first-array.begin());
   printf("Остання  позиція для вставки: %d\n", range.second-array.begin());
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний масив:
10 9 5 5 5 4 3 2 1 0
Перша позиція для вставки:    2
Остання позиція для вставки: 5
```

14.3.9. Алгоритм binary_search

```
template<class ForwardIterator, class T>
```

```
  bool binary_search(ForwardIterator first,
                     ForwardIterator last,
                     const T& value);

template<class ForwardIterator, class T, class Compare>
  bool binary_search(ForwardIterator first,
                     ForwardIterator last,
                     const T& value,
                     Compare comp);
```

Виконує бінарний пошук елемента, що має значення `value`, у діапазоні `[first,last)`. Повертає значення `true`, якщо ітератор `i` з діапазону `[first,last)` задовольняє умовам `!(*i<value)&&!(value<*i)` чи `comp(*i,value) == false && comp(value,*k)==false`.

Приклад алгоритму binary_search (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 10;
   const int NUM = 5;
   int i;
   int initial[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   vector<int> array(initial,initial+SIZE);

   printf("Вихідний вектор: \n");
   for (i = 0; i < SIZE; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Выполняем пошук елементів

   if(binary_search(array.begin(), array.end(), NUM))
     printf("Число %d міститься у векторі\n",NUM);
     else
     printf("Число %d не міститься у векторі\n",NUM);
   printf("\n");

   return 0;
}
```

Результат

```
Вихідний     масив:
1 2 3 4 5 6 7 8 9 10
Число 5 міститься в масиві
```

Приклад алгоритму binary_search (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string.h>

using namespace std;

bool Equal(const char* element, const char* value)
{
   return strcmp(element, value)<0? 1:0;
}

int main()
{
   const int SIZE = 6;
   bool Is;
   char* array[6] = {"AAA", "BBB", "CCC", "DDD", "FFF", "GGG"};
   vector<char*> str(array,array+SIZE);
```

```
   printf("Вихідний вектор: \n");
   for (int i = 0; i < SIZE; i++)
     printf("%s ", str[i]);
     printf("\n");

   // Выполняем пошук елемента

   Is = binary_search(str.begin(), str.end(), "ABC", Equal);

   if(Is) printf("У контейнері є рядок %s\n","ABC");
     else printf("У контейнері немає рядка  %s\n","ABC");

   Is = binary_search(str.begin(), str.end(), "AAA", Equal);

   if(Is) printf("У контейнері є рядок %s\n","AAA");
     else printf("У контейнері немає рядка  %s\n","AAA");

   printf("\n");

   return 0;
}
```

Результат

```
Вихідний вектор:
AAA BBB CCC DDD FFF GGG
У контейнері немає рядка  ABC
У контейнері є рядок AAA
```

14.3.10. Алгоритм merge

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                       InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

Поєднує два упорядкованих діапазони `[first1,last1)` і `[first2,last2)`, поміщаючи результат у діапазон `[result, result+(last1- first1)+(last2- first2)`. Результуючий діапазон упорядковується по зростанню відповідно до відношення порівняння, заданим функцією `comp()`, і не повинний перетинатися з вихідними діапазонами. Алгоритм `merge` повертає ітератор `result+(last1- first1)+(last2- first2)`.

Приклад алгоритму merge (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 5;
   const int SIZE2 = 7;
   int i;

   int first[SIZE1] = {1,2,3,4,5};
   int second[SIZE2]= {6,7,8,9,10,11,12};

   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);

   printf("Перший вектор: \n");
```

```
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array1[i]);
     printf("\n");

   printf("Другий вектор: \n");
   for (i = 0; i < SIZE2; i++)
     printf("%d ", array2[i]);
     printf("\n");

   // Выполняем злиття

   merge(array1.begin(),array1.end(),array2.begin(),
          array2.end(),result.begin());

   printf("Об'єднаний вектор: \n");
   for (i = 0; i < SIZE1+SIZE2; i++)
     printf("%d ", result[i]);
     printf("\n");
   printf("\n");

   return 0;
}
```

Результат

```
Перший вектор:
1 2 3 4 5
Другий вектор:
6 7 8 9 10 11 12
Об'єднаний вектор:
1 2 3 4 5 6 7 8 9 10 11 12
```

Приклад алгоритму merge (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE1 = 5;
   const int SIZE2 = 7;
   int i;

   int first[SIZE1] = {1,2,3,4,5};
   int second[SIZE2]= {6,7,8,9,10,11,12};

   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);

   printf("Перший вектор: \n");
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array1[i]);
     printf("\n");

   printf("Другий вектор: \n");
   for (i = 0; i < SIZE2; i++)
     printf("%d ", array2[i]);
     printf("\n");

   // Выполняем злиття

   merge(array1.begin(),array1.end(),
         array2.begin(),array2.end(),result.begin(),
         greater<int>());
```

```
   printf("Об'єднаний вектор: \n");
   for (i = 0; i < SIZE1+SIZE2; i++)
     printf("%d ", result[i]);
     printf("\n");
   printf("\n");

   return 0;
}
```

Результат

```
Перший вектор:
1 2 3 4 5
Другий вектор:
6 7 8 9 10 11 12
Об'єднаний вектор:
6 7 8 9 10 11 12 1 2 3 4 5
```

Примітка. Всі елементи другого вектора передують елементам першого вектора, оскільки елементи другого вектора більше елементів першого вектора.

14.3.11. Алгоритм inplace_merge

```
template<class BidirectionalIterator>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  void inplace_merge(BidirectionalIterator first,
                     BidirectionalIterator middle,
                     BidirectionalIterator last,
                      Compare comp);
```

Поєднує діапазон `[first,last)` з діапазоном `[middle, last)`. Обидва діапазони належать однієї і тієї ж послідовності і повинні бути упорядковані по зростанню. Результуюча послідовність є упорядкованої по зростанню.

Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму inplace_merge (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 7;
   const int SIZE2 = 5;
   int i;

   int first[SIZE1+SIZE2]  = {6,7,8,9,10,11,12,1,2,3,4,5};

   vector<int> array(first,first+SIZE1+SIZE2);

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array[i]);
     printf("\n");

   printf("Другий діапазон: \n");
   for (i = SIZE1; i < SIZE1+SIZE2; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Выполняем злиття

   inplace_merge(array.begin(),array.begin()+SIZE1,array.end());

   printf("Упорядкований вектор: \n");
```

```
   for (i = 0; i < SIZE1+SIZE2; i++)
     printf("%d ", array[i]);
     printf("\n");
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
6 7 8 9 10 11 12
Другий діапазон:
1 2 3 4 5
Упорядкований вектор:
1 2 3 4 5 6 7 8 9 10 11 12
```

Приклад алгоритму inplace_merge (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE1 = 7;
   const int SIZE2 = 5;
   int i;

   int first[SIZE1+SIZE2]  = {6,7,8,9,10,11,12,1,2,3,4,5};

   vector<int> array(first,first+SIZE1+SIZE2);

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array[i]);
     printf("\n");

     printf("Другий діапазон: \n");
     for (i = SIZE1; i < SIZE1+SIZE2; i++)
     printf("%d ", array[i]);
     printf("\n");

   // Выполняем злиття

   inplace_merge(array.begin(),array.begin()+SIZE1,
                 array.end(), greater<int>());


   printf("Упорядкований вектор: \n");
   for (i = 0; i < SIZE1+SIZE2; i++)
     printf("%d ", array[i]);
     printf("\n");
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
6 7 8 9 10 11 12
Другий діапазон:
1 2 3 4 5
Упорядкований вектор:
```
6 7 8 9 10 11 12 1 2 3 4 5

Примітка. Всі елементи другого діапазону передують елементам першого діапазону, оскільки елементи другого діапазону більше елементів першого діапазону.

14.3.12. Алгоритм includes

```
template<class InputIterator1, class InputIterator2>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool includes(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, InputIterator2 last2,
               Compare comp);
```

Визначає, ли містить діапазон `[first1,last1)` всі елементи діапазону `[first2, last2)` у тім же порядку, тобто ли є другий діапазон підпослідовністю першого. Якщо відповідь стверджувальний, алгоритм повертає значення `true`, у противному випадку — значення `false`. Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму includes (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 7;
   const int SIZE2 = 5;
   int i;

   int first[SIZE1]  = {1,2,3,4,5,6,7};
   int second[SIZE2] = {1,2,3,4,5};

   bool Is;

   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array1[i]);
     printf("\n");

     printf("Другий діапазон: \n");
     for (i = 0; i < SIZE2; i++)
     printf("%d ", array2[i]);
     printf("\n");

   // Перевіряємо, ли містить перший діапазон всі елементи другого діапазону

   Is = includes(array1.begin(),array1.end(),
                 array2.begin(),array2.end());

   if(Is) printf("Includes\n");
     else printf("Not includes\n");

   return 0;
}
```

Результат

```
Перший діапазон:
1 2 3 4 5 6 7
Другий діапазон:
1 2 3 4 5
Містить
```

Контрпример до алгоритму includes

```
#include <iostream>
#include <algorithm>
#include <vector>
```

```
#include <functional>

using namespace std;

int main()
{
   const int SIZE1 = 7;
   const int SIZE2 = 5;
   int i;

   int first[SIZE1]  = {1,2,3,4,5,6,7};
   int second[SIZE2] = {5,4,3,2,1};

   bool Is;

   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
     printf("%d ", array1[i]);
     printf("\n");

     printf("Другий діапазон: \n");
     for (i = 0; i < SIZE2; i++)
     printf("%d ", array2[i]);
     printf("\n");

   // Перевіряємо, ли містить перший діапазон всі елементи другого діапазону

     Is = includes(array1.begin(),array1.end(),
                   array2.begin(),array2.end());

     if(Is) printf("Містить\n");
       else printf("Не містить");

     return 0;
}
```

Результат

```
Перший діапазон:
1 2 3 4 5 6 7
Другий діапазон:
5 4 3 2 1
Не містить
```

Приклад алгоритму includes (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;
   bool Is;

   char* first[SIZE1]  = {"AAA","BBB","CCC","DDD","EEE", "FFF"};
   char* second[SIZE2] = {"AAA","EEE","FFF"};
```

```
   vector<char*> array1(first,first+SIZE1);
   vector<char*> array2(second,second+SIZE2);

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%s ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%s ", array2[i]);
   printf("\n");

   // Перевіряємо, ли містить перший діапазон всі елементи другого діапазону

   Is = includes(array1.begin(),array1.end(),
                 array2.begin(),array2.end(), Comp);

   if(Is) printf("Містить\n");
     else printf("Не містить\n");

   return 0;
}
```

Результат

```
Перший діапазон:
AAA BBB CCC DDD EEE FFF
Другий діапазон:
AAA EEE FFF
Містить
```

14.3.13. Алгоритм set_union

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                           InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result, Compare comp);
```

Створює упорядковане об'єднання елементів двох діапазонів: `[first1, last1)` і `[first2, last2)`. Результат записується в діапазон `[result, result+(last1- first1)+(last2- first2))`. Алгоритм повертає ітератор `result+(last1- first1)+(last2- first2)`. Результуючий діапазон не повинний перетинатися з вихідними діапазонами. Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму set_union (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   int first[SIZE1]  = {10,20,30,40,50,60};
   int second[SIZE2] = {5,6,7};
   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);
   vector<int>::iterator where;
```

```
   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%d ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%d ", array2[i]);
   printf("\n");

   // Об'єднання

   where = set_union(array1.begin(),array1.end(),
                     array2.begin(),array2.end(),
                      result.begin());

   printf("Об'єднаний діапазон: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%d ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
10 20 30 40 50 60
Другий діапазон:
5 6 7
Об'єднаний діапазон:
5 6 7 10 20 30 40 50 60
```

Приклад алгоритму set_union (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   char* first[SIZE1]  = {"AAA","BBB","CCC","DDD","EEE", "FFF"};
   char* second[SIZE2] = {"AAA","BBB","GGG"};
   vector<char*> array1(first,first+SIZE1);
   vector<char*> array2(second,second+SIZE2);
   vector<char*> result(SIZE1+SIZE2);
   vector<char*>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%s ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%s ", array2[i]);
   printf("\n");
```

```
   // Об'єднання

   where = set_union(array1.begin(),array1.end(),
                     array2.begin(),array2.end(),
                     result.begin(), Comp);

   printf("Об'єднаний діапазон: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%s ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
AAA BBB CCC DDD EEE FFF
Другий діапазон:
AAA BBB GGG
Об'єднаний діапазон:
AAA BBB CCC DDD EEE FFF GGG
```

14.3.14. Алгоритм set_intersection

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                  InputIterator2 first2, InputIterator2 last2,
                                 OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                                 InputIterator2 first2, InputIterator2 last2,
                                 OutputIterator result, Compare comp);
```

Створює упорядкований діапазон, що представляє собою перетинання двох упорядкованих діапазонів `[first1, last1)` і `[first2, last2)`. Результат записується в діапазон `[result, result+(last1-first1)+(last2- first2))`. Алгоритм повертає ітератор `result+(last1- first1)+(last2-first2)`. Результуючий діапазон не повинний перетинатися з вихідними діапазонами.

Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму set_intersection (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   int first[SIZE1]  = {1,2,3,4,5,6};
   int second[SIZE2] = {5,6,7};
   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);
   vector<int>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%d ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
```

```
   printf("%d ", array2[i]);
   printf("\n");

   // Перетинання

   where = set_intersection(array1.begin(),array1.end(),
                            array2.begin(),array2.end(),
                            result.begin());

   printf("Перетинання: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%d ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
1 2 3 4 5 6
Другий діапазон:
5 6 7
Перетинання:
5 6
```

Приклад алгоритму set_intersection (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   char* first[SIZE1]  = {"AAA","BBB","CCC","DDD","EEE", "FFF"};
   char* second[SIZE2] = {"AAA","BBB","GGG"};
   vector<char*> array1(first,first+SIZE1);
   vector<char*> array2(second,second+SIZE2);
   vector<char*> result(SIZE1+SIZE2);
   vector<char*>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%s ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%s ", array2[i]);
   printf("\n");

   // Перетинання

   where = set_intersection(array1.begin(),array1.end(),
                            array2.begin(),array2.end(),
                            result.begin(), Comp);

   printf("Перетинання: \n");
```

```
   for (i = 0; i < where-result.begin(); i++)
   printf("%s ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
AAA BBB CCC DDD EEE FFF
Другий діапазон:
AAA BBB GGG
Перетинання:
AAA BBB
```

14.3.15. Алгоритм set_difference

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                                InputIterator2 first2, InputIterator2 last2,
                               OutputIterator result, Compare comp);
```

Створює упорядкований діапазон, що містить різницю двох упорядкованих діапазонів: `[first1, last1)` і `[first2, last2)`. Результат записується в діапазон `[result, result+(last1- first1)+(last2- first2))`. Алгоритм повертає ітератор `result+(last1- first1)+(last2- first2)`. Результуючий діапазон не повинний перетинатися з вихідними діапазонами.

Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму set_difference (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   int first[SIZE1]  = {1,2,3,4,5,6};
   int second[SIZE2] = {5,6,7};
   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);
   vector<int>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%d ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%d ", array2[i]);
   printf("\n");

   // Різниця

   where = set_difference(array1.begin(),array1.end(),
                          array2.begin(),array2.end(),
                          result.begin());
```

```
   printf("Різниця: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%d ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
1 2 3 4 5 6
Другий діапазон:
5 6 7
Різниця:
1 2 3 4
```

Приклад алгоритму set_difference (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   char* first[SIZE1]  = {"AAA","BBB","CCC","DDD","EEE", "FFF"};
   char* second[SIZE2] = {"AAA","BBB","GGG"};
   vector<char*> array1(first,first+SIZE1);
   vector<char*> array2(second,second+SIZE2);
   vector<char*> result(SIZE1+SIZE2);
   vector<char*>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%s ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%s ", array2[i]);
   printf("\n");

   // Різниця

   where = set_difference(array1.begin(),array1.end(),
                          array2.begin(),array2.end(),
                          result.begin(), Comp);

   printf("Різниця: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%s ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
```

```
AAA BBB CCC DDD EEE FFF
Другий діапазон:
AAA BBB GGG
Різниця:
CCC DDD EEE FFF
```

14.3.16. Алгоритм set_symmetric_difference

```
template<class InputIterator1, class InputIterator2, class OutputIterator>
  OutputIterator set_symmetric_difference(InputIterator1 first1,
                                          InputIterator1 last1,
                                           InputIterator2 first2,
                                          InputIterator2 last2,
                                          OutputIterator result);

template<class InputIterator1, class InputIterator2,
         class OutputIterator, class Compare>
  OutputIterator set_symmetric_difference(InputIterator1 first1,
                                          InputIterator1 last1,
                                           InputIterator2 first2,
                                          InputIterator2 last2,
                                           OutputIterator result,
                                          Compare comp);
```

Створює упорядкований діапазон, що містить симетричну різницю двох упорядкованих діапазонів: `[first1, last1)` і `[first2, last2)`. Результат записується в діапазон `[result, result+(last1-first1)+(last2- first2))`. Алгоритм повертає ітератор `result+(last1- first1)+(last2-first2)`. Результуючий діапазон не повинний перетинатися з вихідними діапазонами.

Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму set_symmetric_difference (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   int first[SIZE1]  = {1,2,3,4,5,6};
   int second[SIZE2] = {5,6,7};
   vector<int> array1(first,first+SIZE1);
   vector<int> array2(second,second+SIZE2);
   vector<int> result(SIZE1+SIZE2);
   vector<int>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%d ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%d ", array2[i]);
   printf("\n");

   // Симетрична різниця

   where = set_symmetric_difference(array1.begin(),array1.end(),
                                    array2.begin(),array2.end(),
                                    result.begin());

   printf("Симетрична різниця: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%d ", result[i]);
```

```
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
1 2 3 4 5 6
Другий діапазон:
5 6 7
Симетрична різниця:
1 2 3 4 7
```

Приклад алгоритму set_symmetric_difference (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE1 = 6;
   const int SIZE2 = 3;
   int i;

   char* first[SIZE1]  = {"AAA","BBB","CCC","DDD","EEE", "FFF"};
   char* second[SIZE2] = {"AAA","BBB","GGG"};
   vector<char*> array1(first,first+SIZE1);
   vector<char*> array2(second,second+SIZE2);
   vector<char*> result(SIZE1+SIZE2);
   vector<char*>::iterator where;

   printf("Перший діапазон: \n");
   for (i = 0; i < SIZE1; i++)
   printf("%s ", array1[i]);
   printf("\n");

   printf("Другий діапазон: \n");
   for (i = 0; i < SIZE2; i++)
   printf("%s ", array2[i]);
   printf("\n");

   // Різниця

   where = set_symmetric_difference(array1.begin(),array1.end(),
                                    array2.begin(),array2.end(),
                                    result.begin(), Comp);

   printf("Різниця: \n");
   for (i = 0; i < where-result.begin(); i++)
   printf("%s ", result[i]);
   printf("\n");

   return 0;
}
```

Результат

```
Перший діапазон:
AAA BBB CCC DDD EEE FFF
Другий діапазон:
AAA BBB GGG
Симетрична різниця:
```

```
CCC DDD EEE FFF GGG
```

14.3.17. Алгоритм make_heap

```
template<class RandomAccessIterator>
  void make_heap(RandomAccessIterator first,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void make_heap(RandomAccessIterator first,
                 RandomAccessIterator last,
                  Compare comp);
```

Створює купу з діапазону [first,last). Друга версія дозволяє задати функцію порівняння, що визначає критерій пошуку.

Приклад алгоритму make_heap (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;
   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first, first+SIZE);
   vector<int>::iterator iter;

   printf("Вектор: \n");
   for (iter=array.begin(); iter!= array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   make_heap(array.begin(), array.end());

   printf("Купа: \n");
   for (iter=array.begin(); iter!= array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Купа:
5 4 3 1 2
```

Приклад алгоритму make_heap (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{

   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
```

```
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end(),Comp);

   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Купа:
EEE DDD CCC AAA BBB
```

14.3.18. Алгоритм push_heap

```
template<class RandomAccessIterator>
  void push_heap(RandomAccessIterator first,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void push_heap(RandomAccessIterator first,
                 RandomAccessIterator last,
                 Compare comp);
```

Поміщає новий елемент у кінець купи. Передбачається, що діапазон `[first, last)` являє собою коректну купу. Друга версія дозволяє задати функцію порівняння `comp()`.

Приклад алгоритму push_heap (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;

   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end());

   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   // Вставляем новий елемент у кінець масиву

   array.push_back(10);

   printf("Вектор після вставки: \n");
```

```
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   // Помещаем новий елемент у купу

   push_heap(array.begin(),array.end());

   printf("Купа після вставки: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Купа:
5 4 3 1 2
Вектор після вставки:
5 4 3 1 2 10
Купа після вставки:
10 4 5 1 2 3
```

Приклад алгоритму push_heap (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end(),Comp);

   // Вставляем новий елемент у кінець масиву

   array.push_back("GGG");

   printf("Вектор після вставки: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   // Помещаем новий елемент у купу

   push_heap(array.begin(),array.end());

   printf("Купа після вставки: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
```

```
   printf("%s ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Вектор після вставки:
EEE DDD CCC AAA BBB GGG
Купа після вставки:
EEE DDD CCC AAA BBB GGG
```

14.3.19. Алгоритм pop_heap

```
template<class RandomAccessIterator>
  void pop_heap(RandomAccessIterator first,
                RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void pop_heap(RandomAccessIterator first,
                RandomAccessIterator last,
                Compare comp);
```

Змінює местами перший і передостанній елементи купи. Друга версія дозволяє програмісту самому задати функцію порівняння.

Приклад алгоритму pop_heap (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;

   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end());

   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   // Удаляем кореневий елемент купи

   pop_heap(array.begin(),array.end());
   printf("Купа після видалення кореневого елемента: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Купа:
```

```
5 4 3 1 2
Купа після видалення кореневого елемента:
4 2 3 1 5
```

Приклад алгоритму pop_heap (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end(),Comp);

   // Купа
   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   // Удаляем кореневий елемент

   pop_heap(array.begin(),array.end());

   printf("Купа після видалення: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Купа:
EEE DDD CCC AAA BBB
Купа після видалення кореневого елемента:
BBB DDD CCC AAA EEE
```

14.3.20. Алгоритм sort_heap

```
template<class RandomAccessIterator>
  void sort_heap(RandomAccessIterator first,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort_heap(RandomAccessIterator first,
                 RandomAccessIterator last,
                 Compare comp);
```

Упорядковує купу `[first,last)`. Друга версія дозволяє програмісту самому задати функцію порівняння елементів.

Приклад алгоритму sort_heap (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;

   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end());

   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   // Упорядковуємо купу

   sort_heap(array.begin(),array.end());

   printf("Купа після сортування: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Купа:
5 4 3 1 2
Купа після сортування:
1 2 3 4 5
```

Приклад алгоритму sort_heap (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter;

   printf("Вектор: \n");
```

```
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   make_heap(array.begin(),array.end(),Comp);

   // Купа
   printf("Купа: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   // Упорядковуємо купу

   sort_heap(array.begin(),array.end());

   printf("Купа після сортування: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Купа:
EEE DDD CCC AAA BBB
Купа після сортування:
DDD CCC AAA BBB EEE
```

14.3.21. Алгоритм min

```
template<class T>
  const T& min(const T& a, const T& b);

template<class T, class Compare>
  const T& min(const T& a, const T& b, Compare comp);
```

Обидві версії повертають найменше з двох значень. Якщо значення рівні, повертається перший аргумент. Друга версія дозволяє задати функцію порівняння.

Приклад алгоритму min (перший варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   int a = 10, b = 20;
   printf("Min (a,b) = %d \n",min(a,b));
   return 0;
}
```

Результат

```
Min(a,b) = 10
```

Приклад алгоритму min (другий варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
```

```
{
   char *a = "AAA", *b = "BBB";
   printf("Min (a,b) = %s \n",min(a,b,Comp));
   return 0;
}
```

Результат

```
Min(a,b) = AAA
```

14.3.22. Алгоритм max

```
template<class T>
  const T& max(const T& a, const T& b);

template<class T, class Compare>
  const T& max(const T& a, const T& b, Compare comp);
```

Повертає найбільше з двох значень. Якщо значення рівні, повертається перший аргумент. Друга версія дозволяє задати функцію порівняння.

Приклад алгоритму max (перший варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
   int a = 10, b = 20;
   printf("Max (a,b) = %d \n",max(a,b));
   return 0;
}
```

Результат

```
Max(a,b) = 20
```

Приклад алгоритму max (другий варіант)

```
#include <iostream>
#include <algorithm>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   char *a = "AAA", *b = "BBB";
   printf("Max (a,b) = %s \n",max(a,b,Comp));
   return 0;
}
```

Результат

```
Max(a,b) = BBB
```

14.3.23. Алгоритм min_element

```
template<class ForwardIterator>
  ForwardIterator min_element(ForwardIterator first,
                              ForwardIterator last);

template<class ForwardIterator, class Compare>
  ForwardIterator min_element(ForwardIterator first,
                              ForwardIterator last,
                              Compare comp);
```

Обидві версії повертають ітератор, установлений на мінімальний елемент у діапазоні `[first,last)`. Друга версія дозволяє задати функцію порівняння `comp()`.

Приклад алгоритму min_element (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;

   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter, where;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   where = min_element(array.begin(),array.end());

   printf("Мінімальний елемент = %d ", *where);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Мінімальний елемент = 1
```

Приклад алгоритму min_element (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter, where;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   where = min_element(array.begin(),array.end(),Comp);

   printf("Мінімальний елемент = %s ", *where);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Мінімальний елемент = AAA
```

14.3.24. Алгоритм max_element

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first,
                            ForwardIterator last);

template<class ForwardIterator, class Compare>
  ForwardIterator max_element(ForwardIterator first,
                              ForwardIterator last,
                              Compare comp);
```

Повертає ітератор, установлений на максимальний елемент у діапазоні `[first,last)`. Друга версія дозволяє задати функцію порівняння `comp()`.

Приклад алгоритму max_element (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
{
   const int SIZE = 5;

   int first[SIZE]  = {1,2,3,4,5};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter, where;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   where = max_element(array.begin(),array.end());

   printf("Максимальний елемент = %d ", *where);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
1 2 3 4 5
Максимальний елемент = 5
```

Приклад алгоритму max_element (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter, where;

   printf("Вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
```

```
   printf("%s ", *iter);
   printf("\n");

   where = max_element(array.begin(),array.end(),Comp);

   printf("Максимальний елемент = %s ", *where);
   printf("\n");

   return 0;
}
```

Результат

```
Вектор:
AAA BBB CCC DDD EEE
Максимальний елемент = EEE
```

14.3.25. Алгоритм lexicographical_compare

```
template<class InputIterator1, class InputIterator2>
  bool lexicographical_compare(InputIterator1 first1,
                               InputIterator1 last1,
                                InputIterator2 first2,
                               InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
  bool lexicographical_compare(InputIterator1 first1,
                               InputIterator1 last1,
                                InputIterator2 first2,
                               InputIterator2 last2,
                               Compare comp);
```

Порівнює діапазони `[first1, last1)` і `[first2, last2)`, використовуючи лексикографічний порядок. Якщо перша послідовність менше другої в лексикографічному змісті, алгоритм повертає значення `true`, у противному випадку — значення `false`.

Друга версія алгоритму дозволяє задати функцію порівняння елементів `comp()`.

Приклад алгоритму lexicographical_compare (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 5;
   bool Less;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   char* second[SIZE]  = {"AAB","BBC","CCD","DDE","EEF"};
   vector<char*> array1(first,first+SIZE);
   vector<char*> array2(second,second+SIZE);
   vector<char*>::iterator iter;

   printf("Перший вектор: \n");
   for (iter = array1.begin(); iter != array1.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   printf("Другий вектор: \n");
   for (iter = array2.begin(); iter != array2.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   Less = lexicographical_compare(array1.begin(),array1.end(),
                                  array2.begin(),array2.end());

   if (Less) printf("array1 >  array2");
```

```
        else  printf("array2 <= array1");
   printf("\n");

   return 0;
}
```

Результат

```
Перший вектор:
AAA BBB CCC DDD EEE
Другий вектор
AAB BBC CCD DDE EEF
array2 <= array1
```

Приклад алгоритму lexicographical_compare (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <functional>

using namespace std;

int main()
 {
   const int SIZE = 5;
   bool Less;

   char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
   char* second[SIZE]  = {"AAB","BBC","CCD","DDE","EEF"};
   vector<char*> array1(first,first+SIZE);
   vector<char*> array2(second,second+SIZE);
   vector<char*>::iterator iter;


   printf("Перший вектор: \n");
   for (iter = array1.begin(); iter != array1.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   printf("Другий вектор: \n");
   for (iter = array2.begin(); iter != array2.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   Less = lexicographical_compare(array1.begin(),array1.end(),
                                  array2.begin(),array2.end(),
                                  greater<char*>());

   if (Less) printf("array1 >  array2");
       else  printf("array1 <= array2");
   printf("\n");

   return 0;
}
```

Результат

```
Перший вектор:
AAA BBB CCC DDD EEE
Другий вектор:
AAB BBC CCD DDE EEF
array1 >  array2
```

14.3.26. Алгоритм next_permutation

```
template<class BidirectionalIterator>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool next_permutation(BidirectionalIterator first,
                        BidirectionalIterator last, Compare comp);
```

Створює чергову перестановку елементів діапазону [first, last). Передбачається, що безліч усіх перестановок є цілком упорядкованим по відношенню “менше” чи в лексикографічному порядку. Якщо чергова перестановка існує, алгоритм повертає значення true. У противному випадку алгоритм упорядковує вихідну (найменшу) перестановку і повертає значення false. Друга версія алгоритму дозволяє задати функцію порівняння.

Приклад алгоритму next_permutation (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
 {
    const int SIZE = 5;
    bool Is;

    int first[SIZE]  = {1,2,3,4,5};
    vector<int> array(first,first+SIZE);
    vector<int>::iterator iter;


    printf("Вихідний вектор: \n");
    for (iter = array.begin(); iter != array.end(); iter++)
    printf("%d ", *iter);
    printf("\n");

    Is = next_permutation(array.begin(),array.end());

    if (Is)
    {
      printf("Наступна перестановка: \n");
      for (iter = array.begin(); iter != array.end(); iter++)
      printf("%d ", *iter);
      printf("\n");
    }
    else
      printf("Наступної перестановки не існує");

    return 0;
 }
```

Результат

```
Вихідний вектор:
1 2 3 4 5
Наступна перестановка:
1 2 3 5 4
```

Приклад алгоритму next_permutation (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
 {
    const int SIZE = 5;
    bool Is;

    char* first[SIZE]  = {"AAA","BBB","CCC","DDD","EEE"};
    vector<char*> array(first,first+SIZE);
```

```
   vector<char*>::iterator iter;


   printf("Вихідний вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   Is = next_permutation(array.begin(),array.end(),Comp);

   if (Is)
   {
     printf("Наступна перестановка: \n");
     for (iter = array.begin(); iter != array.end(); iter++)
     printf("%s ", *iter);
     printf("\n");
   }
   else
     printf("Наступної перестановки не існує");

   return 0;
}
```

Результат

```
Вихідний вектор:
AAA BBB CCC DDD EEE
Наступна перестановка:
AAA BBB CCC EEE DDD
```

14.3.27. Алгоритм prev_permutation

```
template<class BidirectionalIterator>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
  bool prev_permutation(BidirectionalIterator first,
                        BidirectionalIterator last,
                        Compare comp);
```

Створює перестановку, що передує діапазону `[first, last)`. Передбачається, що безліч усіх перестановок є цілком упорядкованим по відношенню “менше” чи в лексикографічному порядку. Якщо попередня перестановка існує, алгоритм повертає значення `true`. У противному випадку алгоритм упорядковує останню (найбільшу) перестановку і повертає значення `false`. Друга версія алгоритму дозволяє задати функцію порівняння.

Приклад алгоритму next_permutation (перший варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

int main()
 {
   const int SIZE = 5;
   bool Is;

   int first[SIZE]  = {1,2,3,5,4};
   vector<int> array(first,first+SIZE);
   vector<int>::iterator iter;


   printf("Вихідний вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%d ", *iter);
   printf("\n");

   Is = prev_permutation(array.begin(),array.end());
```

```
   if (Is)
   {
     printf("Попередня перестановка: \n");
     for (iter = array.begin(); iter != array.end(); iter++)
     printf("%d ", *iter);
     printf("\n");
   }
   else
     printf("Попередньої перестановки не існує");

   return 0;
}
```

Результат

```
Вихідний вектор:
1 2 3 5 4
Попередня перестановка:
1 2 3 4 5
```

Приклад алгоритму next_permutation (другий варіант)

```
#include <iostream>
#include <algorithm>
#include <vector>

using namespace std;

bool Comp(const char* s1, const char* s2)
{
   return strcmp(s1,s2) < 0 ? 1:0;
}

int main()
{
   const int SIZE = 5;
   bool Is;

   char* first[SIZE]  = {"AAA","BBB","CCC","EEE","DDD"};
   vector<char*> array(first,first+SIZE);
   vector<char*>::iterator iter;

   printf("Вихідний вектор: \n");
   for (iter = array.begin(); iter != array.end(); iter++)
   printf("%s ", *iter);
   printf("\n");

   Is = prev_permutation(array.begin(),array.end(),Comp);

   if (Is)
   {
     printf("Попередня перестановка: \n");
     for (iter = array.begin(); iter != array.end(); iter++)
     printf("%s ", *iter);
     printf("\n");
   }
   else
     printf("Попередньої перестановки не існує");

   return 0;
}
```

Результат

```
Вихідний вектор:
AAA BBB CCC EEE DDD
Попередня перестановка:
AAA BBB CCC DDD EEE
```

14.4. Чисельні алгоритми

Деякі чисельні алгоритми носять універсальний характер. Ця обставина дозволила реалізувати їх у виді шаблонових функцій і помістити в стандартну бібліотеку шаблонів.

14.4.1. Алгоритм accumulate

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last,
             T init, BinaryOperation binary_op);
```

Обчислює суму всіх елементів діапазону, обмеженого ітераторами `first` і `last`. Друга версія дозволяє довільно задати бінарну операцію, яку варто застосувати до елементів вектора (наприклад, послідовно перемножити їх, обчислюючи факторіал). Параметр `init` задає початкове значення, з якого починається підсумовування.

Приклад алгоритму accumulate (перший варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
   const int SIZE = 10;
   int array[10] = {1,2,3,4,5,6,7,8,9,10};
   vector<int> objVector(array, array+SIZE);;
   int  sum = accumulate(objVector.begin(), objVector.end(),0);
   printf("Сума дорівнює %d\n", sum);

   return 0;
}
```

Результат

```
Сума дорівнює 55
```

Приклад алгоритму accumulate (другий варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int factor(int n1, int n2)
{
   return n1*n2;
}

int main()
{
   const int SIZE = 5;
   int array[5] = {1,2,3,4,5};
   vector<int> objVector(array, array+SIZE);;
   int factorial = accumulate(objVector.begin(), objVector.end(), 1, factor);
   printf(" Факторіал дорівнює %d\n", factorial);

 return 0;
}
```

Результат

```
Факторіал дорівнює 120
```

14.4.2. Алгоритм inner_product

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
          class BinaryOperation1, class BinaryOperation2>
```

```
T inner_product(InputIterator1 first1,
                InputIterator1 last1,
                InputIterator2 first2,
                T init,
                BinaryOperation1 binary_op1,
                BinaryOperation2 binary_op2);
```

Обчислює скалярний добуток двох векторів: `[first1,last1)` і `[first2,last2)`. Підсумовування починається з початкового значення `init`. Друга версія дозволяє уточнити метод підсумовування і множення елементів за допомогою функцій `binary_op1()` і `binary_op2()` відповідно.

Приклад алгоритму inner_product (перший варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
   const int SIZE = 5;
   int first[5] = {1,2,3,4,5}, second[5] = {5,4,3,2,1};

   vector<int> objVector1(first, first+SIZE),
               objVector2(second, second+SIZE);

   int  inner = inner_product(objVector1.begin(),objVector1.end(),
                              objVector2.begin(),0);
   printf(" Скалярний добуток дорівнює %d\n", inner);

   return 0;
}
```

Результат

```
Скалярний добуток дорівнює 35
```

Приклад алгоритму inner_product (другий варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

double Plus(double n1, double n2)
{
   return (n1+n2)/2;
}

double Mult(double n1, double n2)
{
   return (n1*n2)/2;
}

int main()
{
   const int SIZE = 5;
   double first[5] = {1,2,3,4,5}, second[5] = {5,4,3,2,1};

   vector<double> objVector1(first, first+SIZE),
                  objVector2(second, second+SIZE);

   double  inner = inner_product(objVector1.begin(),objVector1.end(),
                                 objVector2.begin(),0,Plus,Mult);
   printf(" Скалярний добуток дорівнює %lf\n", inner);

   return 0;
}
```

Результат

```
Скалярний добуток дорівнює 2.000000
```

14.4.3. Алгоритм partial_sum

```
template <class InputIterator, class OutputIterator>
 OutputIterator partial_sum(InputIterator first,
                            InputIterator last,
                            OutputIterator result);

template <class InputIterator, class OutputIterator,
          class BinaryOperation>
 OutputIterator partial_sum(InputIterator first,
                             InputIterator last,
                             OutputIterator result,
                             BinaryOperation binary_op);
```

Обчислює часткові суми елементів діапазону `[first,last)`. Першим елементом результату є перший елемент вихідної послідовності. Результат зберігається в об'єкті `resut`. Друга версія дозволяє задати спосіб підсумовування. Повертається ітератор, установлений на початок діапазону `result`.

Приклад алгоритму partial_sum (перший варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
   const int SIZE = 5;
   double first[5] = {1,2,3,4,5};

   vector<double> objVector(first, first+SIZE), result(SIZE);
   vector<double>::iterator i;

   partial_sum(objVector.begin(), objVector.end(), result.begin());

   printf("Часткові суми: ");
   for(i=result.begin(); i!=result.end(); i++)
       printf("%7.3f ",*i);
       printf("\n");

   return 0;
}
```

Результат

```
Часткові суми:   1.000   3.000   6.000  10.000  15.000
```

Приклад алгоритму partial_sum (другий варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

double Mult(double n1, double n2)
{
   return n1*n2;
}

int main()
{
   const int SIZE = 5;
   double first[5] = {1,2,3,4,5};

   vector<double> objVector(first, first+SIZE), result(SIZE);
   vector<double>::iterator i;

   partial_sum(objVector.begin(), objVector.end(), result.begin(),Mult);

   printf("Часткові добутки: ");
   for(i=result.begin(); i!=result.end(); i++)
       printf("%7.3f ",*i);
```

```
       printf("\n");

   return 0;
}
```

Результат

```
Часткові добутки:   1.000   2.000   6.000  24.000 120.000
```

14.4.4. Алгоритм adjacent_difference

```
template <class InputIterator, class OutputIterator>
  OutputIterator adjacent_difference(InputIterator first,
                                     InputIterator last,
                                     OutputIterator result);

template <class InputIterator, class OutputIterator,
          class BinaryOperation>
  OutputIterator adjacent_difference(InputIterator first,
                                       InputIterator last,
                                       OutputIterator result,
                                        BinaryOperation binary_op);
}
```

Обчислює послідовність, що містить різниці між сусідніми елементами. Першим елементом результату є перший елемент вихідної послідовності. Друга версія дозволяє задати бінарну операцію, що застосовується до кожної пари сусідніх елементів.

Приклад алгоритму adjacent_difference (перший варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

int main()
{
   const int SIZE = 5;
   double first[5] = {1,4,9,16,25};

   vector<double> objVector(first, first+SIZE), result(SIZE);
   vector<double>::iterator i;

   adjacent_difference(objVector.begin(), objVector.end(),
                       result.begin());

   printf("Різниці: ");
   for(i=result.begin(); i!=result.end(); i++)
       printf("%7.3f ",*i);
       printf("\n");

   return 0;
}
```

Результат

```
Різниці:   1.000   3.000   5.000   7.000   9.000
```

Приклад алгоритму adjacent_difference (другий варіант)

```
#include <iostream>
#include <vector>
#include <numeric>
using namespace std;

double Mult(double n1, double n2)
{
   return n1*n2;
}

int main()
{
   const int SIZE = 5;
   double first[5] = {1,4,9,16,25};
```

```
   vector<double> objVector(first, first+SIZE), result(SIZE);
   vector<double>::iterator i;

   adjacent_difference(objVector.begin(), objVector.end(),
                       result.begin(),Mult);

   printf("Попарні добутки: ");
   for(i=result.begin(); i!=result.end(); i++)
       printf("%7.3f ",*i);
       printf("\n");

   return 0;
}
```

Результат

```
Попарні добутки:   1.000   4.000  36.000 144.000 400.000
```