Algorithm *find* (1)

Let us have

```
list <Date> deadlines = { ….. };
```

To check does the list contains date "January 5, 2019" we may write a loop:

```
bool found = false;
for (auto& d : deadlines)
{
    if (d == Date(5, 2, 2019))
    {
        found = true;
        break;
     }
}
cout << (found ? "Found" : "Not found") << endl;
```

But it is more easy to write:

```
#include <algorithm> // See www.cplusplus.com/reference/algorithm/find/
auto it = find(deadlines.begin(), deadlines.end(), Date(5, 1, 2019));
cout << (it == deadlines.end() ? "Not found" : "Found") << endl;
```

Algorithm *find* (2)

```
iterator_to_result =  find(
                      container_name. iterator_to_first_element_of_range,
                      container_name. iterator_to_first_element_not_in_range,
                      element_to_find);
```

C++ standard algorithm *find* is able to search from any container. It returns the iterator to the first element it has found or, if the searching has failed, the iterator to the first element not in range. Of course, the elements must be of type that supports comparing.

For maps and sets it is better to apply their own built-in method *find*.

Algorithm *find_if* (1)

iterator_to_result = find_if(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, predicate);

The predicate may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. The body of predicate must check does the input value satisfies the search condition and return *true* or *false*.

find_if returns the iterator to the first element for which the predicate returns *true* or, if the searching has failed, the iterator to the first element not in range.

Example (see also http://www.cplusplus.com/reference/algorithm/find_if/):

```
list<Date> deadlines = { …. };
for (auto& d : deadlines)
{
    if (d.GetDay() == 5)
    {
        cout << "Found" << endl;
        break;
    }
}
```

The same with *find_if*:

```
auto it = find_if(deadlines.begin(), deadlines.end(),
                                []( const Date& d)->bool { return d.GetDay() == 5; });
cout << (it1 == deadlines.end() ? "Not found" : "Found") << endl;
```

Algorithm *find_if* (2)

Example:

```
map<string, Date> deadlines = { { "Mathematics", Date(5, 1, 2022) }, { "Chemistry",
Date(10, 1, 2022) },  { "Physics", Date(15, 1, 2022) } };
string subject = "";
for (auto& d: deadlines)
{  // here we do not use keys for searching
    if (d.second == Date(10, 1, 2022))
   {  // we are searching a key corresponding to the specified value
        subject = d.first;
        break;
    }
}
cout << (subject.empty() ? "Not found"  : subject.c_str()) << endl; // prints "Chemistry"
```

The same with algorithm *find_if*:

```
auto it = find_if(deadlines.begin(), deadlines.end(),
          [](const pair<string, Date>& x)->bool { return x.second == Date(10, 1, 2022); });
cout << (it == deadlines.end() ? "Not found" : it->first.c_str()) << endl; // prints "Chemistry"
```

or simply:

```
auto it = find_if(deadlines.begin(), deadlines.end(),
          [](auto x)->bool { return x.second == Date(10, 1, 2022); });
```

Algorithm *find_if* (3)

Instead of lambda we may use a separate function

```
bool CompareDates(const pair<string, Date>& x)
{
    return x.second == Date(10, 1, 2022);
}
auto it = find_if(deadlines.begin(), deadlines.end(), CompareDates);
cout << (it == deadlines.end() ? "Not found" : it->first.c_str()) << endl; // prints "Chemistry"
```

or a functor:

```
class CompareDates
{
private:
  Date date;
public:
  Compare(Date d) : date(d) { }
  bool operator() (pair<string, Date> x) const { return x.second == date; }
};
auto it = find_if(deadlines.begin(), deadlines.end(), CompareDates(Date(15, 1, 2022)));
cout << (it == deadlines.end() ? "Not found" : it->first.c_str()) << endl; // prints "Chemistry"
```

Algorithm *find_if _not*

```
iterator_to_result =  find_if_not(container_name. iterator_to_first_element_of_range,
                                  container_name. iterator_to_first_element_not_in_range, predicate);
```

find_if returns the iterator to the first element for which the predicate return *true* or, if the searching has failed, the iterator to the first element not in range. *find_if_not* returns the iterator to the first element for which the predicate returns *false*.

See also http://www.cplusplus.com/reference/algorithm/find_if_not/

Algorithm *find_first_of* (1)

```
iterator_to_result =  find_first_of(
                      container_1_name. iterator_to_first_element_of_range,
                      container_1_name. iterator_to_first_element_not_in_range,
                      container_2_name. iterator_to_first_element_of_range,
                      container_2_name. iterator_to_first_element_not_in_range);
```

find_first_of returns the iterator to the first element from the range of *container_1* that is equal to any element from the range of *container_2*. In case of failure returns the iterator to the first element not in range from *container_1*.

Example (see also http://www.cplusplus.com/reference/algorithm/find_first_of/):

```
list<Date> deadlines = { Date(5, 1, 2019), Date(10, 1, 2019), Date(15, 1, 2019), Date(27, 1, 2019), Date(19, 1, 2019), Date(17, 1, 2019) }; // deadlines in January 2019
list<Date> sundays = { Date(6, 1, 2019), Date(13, 1, 2019), Date(20, 1, 2019), Date(27, 1, 2019) }; // Sundays in January 2019
auto it = find_first_of(deadlines.begin(), deadlines.end(), sundays.begin(), sundays.end());
if (it != deadlines.end())
    cout << it->GetDay() << endl; // prints 27
```

Algorithm *find_first_of* (2)

```
iterator_to_result =  find_first_of(
                      container_1_name. iterator_to_first_element_of_range,
                      container_1_name. iterator_to_first_element_not_in_range,
                      container_2_name. iterator_to_first_element_of_range,
                      container_2_name. iterator_to_first_element_not_in_range,
                      predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its first argument must be element from the range of *container_1* and the second argument must be element from the range of *container_2*. The body of predicate must check does the input values satisfy the search condition and return *true* or *false*.

Example:

```
map<string, Date> deadlines = {
        { "Mathematics", Date(5, 1, 2019) },
        { "Chemistry", Date(13, 1, 2019) },
        { "Physics", Date(15, 1, 2019) }
};
list<Date> sundays = { Date(6, 1, 2019), Date(13, 1, 2019), Date(20, 1, 2019), Date(27, 1, 2019) }; // Sundays in January 2019
auto it = find_first_of(deadlines.begin(), deadlines.end(), sundays.begin(), sundays.end(),
        [](const pair<string, Date>& x, const Date& d)->bool { return x.second == d; });
cout << it->first.c_str() << endl; // prints Chemistry
```

Algorithms *search* and *find_end* (1)

```
iterator_to_result =  search(
                          container_1_name. iterator_to_first_element_of_range,
                          container_1_name. iterator_to_first_element_not_in_range,
                          container_2_name. iterator_to_first_element_of_range,
                          container_2_name. iterator_to_first_element_not_in_range);
iterator_to_result =  find_end(
                          container_1_name. iterator_to_first_element_of_range,
                          container_1_name. iterator_to_first_element_not_in_range,
                          container_2_name. iterator_to_first_element_of_range,
                          container_2_name. iterator_to_first_element_not_in_range);
```

search tries to find the first occurrence of range of *container_2* from the range of *container_1*. *find_end* tries to find the last occurrence. In case of failure those methods return the iterator to the first element not in range from *container_1*. Actually, *container_2* specifies a pattern.

Example (see also http://www.cplusplus.com/reference/algorithm/search/ and http://www.cplusplus.com/reference/algorithm/find_end/):

```
vector<int> data = { 1, 4, 5, 7, 3, 8, 9, 12, 56, 45, 7 };
vector<int> pattern = { 7, 3, 8 };
auto it = search(data.begin(), data.end(), pattern.begin(), pattern.end());
cout << it - data.begin() << endl; // prints 3 (index of the found pattern)
```

Algorithms *search and find_end* (2)

```
iterator_to_result =  search(
                      container_1_name. iterator_to_first_element_of_range,
                      container_1_name. iterator_to_first_element_not_in_range,
                      container_2_name. iterator_to_first_element_of_range,
                      container_2_name. iterator_to_first_element_not_in_range,
                      predicate);
iterator_to_result =  find_end(
                      container_1_name. iterator_to_first_element_of_range,
                      container_1_name. iterator_to_first_element_not_in_range,
                      container_2_name. iterator_to_first_element_of_range,
                      container_2_name. iterator_to_first_element_not_in_range,
                      predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its first argument must be element from the range of *container_1* and the second argument must be element from the range of *container_2*. The body of predicate must check does the input values satisfy the search condition and return *true* or *false*.

```
vector<int> data = { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int> pattern = { 7, 3, 8 };
auto it = search(data.begin(), data.end(), pattern.begin(), pattern.end(),
            [](int &i1, int &i2)->bool { return abs(i1) == abs(i2); });
cout << it - data.begin() << endl; // prints 3 (index of the found pattern)
```

Algorithm *lower_bound*

iterator_to_result = lower_bound(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range,
element_specifying_the_lower_bound_value);

Returns the iterator to the first element that is not less than the bound or, if the searching has failed, the iterator to the first element not in range. The elements are compared using method *operator<*.

iterator_to_result = lower_bound(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range,
element_specifying_the_lower_bound_value, comparator);

The comparator may be a pointer to function, lambda expression or functor. Its first argument must be element from the container range and the actual value of second argument is always the element specifying the lower bound. The body of comparator must check does the first argument is considered to exceed the second (return value *true*) or not (return value *false*).

Example (see also http://www.cplusplus.com/reference/algorithm/lower_bound/):

```
vector<int> data = { 1, 4, 5, 7, 8, 9, 12, 56, 145, 734 };
auto it1 = lower_bound(data.begin(), data.end(), 10);
cout << it1 - data.begin() << endl; // prints 6 (index of 12 as the first number exceeding 10)
auto it2 = lower_bound(data.begin(), data.end(), 50
                [] (const int &i1, const int i2) { return i1 * i1 < i2; });
cout << it2 - data.begin() << endl; // prints 4 (index of 8 as 8*8 is first number exceeding 50)
```

Algorithm *upper_bound*

iterator_to_result = upper_bound(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range,
element_specifying_the_upper_bound_value);

Returns the iterator to the first element that is greater than the bound or, if the searching has failed, the iterator to the first element not in range. The elements are compared using method *operator<*.

iterator_to_result = upper_bound(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range,
element_specifying_the_upper_bound_value, comparator);

The comparator may be a pointer to function, lambda expression or functor. Its first argument must be element from the container range and the actual value of second argument is always the element specifying the upper bound. The body of comparator must check does the first argument is considered not to exceed the second (return value *true*) or not (return value *false*).

Example (see also http://www.cplusplus.com/reference/algorithm/upper_bound/):

```
vector<int> data = { 1, 4, 5, 7, 8, 9, 12, 56, 145, 734 };
auto it1 = upper_bound(data.begin(), data.end(), 60);
cout << it1 - data.begin() << endl; // prints 8 (index of 145 as the first number greater than 60)
auto it2 = upper_bound(data.begin(), data.end(), 20
                [] (const int &i1, const int i2) { return i1 * i1 > i2; });
cout << it2 - data.begin() << endl; // prints 0 (index of 1 as 1*1 is first number less than  20)
```

Finding minimum and maximum (1)

minimum_value = min(first_argument, second_argument);

The elements are compared using method *operator*<.

minimum_value = min(first_argument, second_argument, comparator);

The comparator may be a pointer to function, lambda expression or functor. The body of comparator must check does the first argument is less than the second (return value *true*) or not (return value *false*).

minimum_value = min(initializer_list);

minimum_value = min(initializer_list, comparator);

If more than one element has the smallest value, the output iterator points to the first of them.

Example (see also http://www.cplusplus.com/reference/algorithm/min/):

```
int a = 5, b = 6;
int c = min(a, b);
cout << c << endl; // prints 5
int i = 5, j = 1, k = 3, l = -10;
int x = min({ i, j, k, l });
cout << x << endl; // prints -10
int y = min({ i, j, k, l }, [](const int &i1, const int &i2)->bool { return abs(i1) < abs(i2); });
cout << y << endl; // prints 1
```

Standard method max is analogous. See http://www.cplusplus.com/reference/algorithm/max/.

Finding minimum and maximum (2)

```
pair_of_iterators_to_min_and_max = minmax_element(
                    container_name. iterator_to_first_element_of_range,
                    container_name. iterator_to_first_element_not_in_range);
```

The elements are compared using method *operator*<.

```
pair_of_iterators_to_min_and_max = minmax_element(
                    container_name. iterator_to_first_element_of_range,
                    container_name. iterator_to_first_element_not_in_range,
                    comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its both arguments must be elements from the container range. The body of comparator must check does the first argument is less than the second (return value *true*) or not (return value *false*).

If more than one element has the smallest (largest) value, the output iterator points to the first (last) of them.

Examples (see also http://www.cplusplus.com/reference/algorithm/minmax_element/):

```
vector<int> data = { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto res1 = minmax_element(data.begin(), data.end());
cout << *res1.first << ' ' << *res1.second << endl; // prints -45 56
auto res2 = minmax_element(data.begin(), data.end(),
                [](const int &i1, const int &i2)->bool { return abs(i1) < abs(i2); });
cout << *res2.first << ' ' << *res2.second << endl; // prints 1 56
```

Finding minimum and maximum (3)

Method *clamp()* was introduced in C++ version 17:

```
result= clamp(value_to_clamp, lower_limit, upper_limit);
```

and

```
result= clamp(value_to_clamp, lower_limit, upper_limit, comparator);
```

If the value to clamp is between the lower and upper limit, it is also the result. If the value to clamp is less than the lower limit, the return value is the lower limit. If the value to clamp is greater than the upper limit, the return value is the upper limit. So, here we make sure is the value we are interested in located in the specified range.

Example (see also https://en.cppreference.com/w/cpp/algorithm/clamp):

```
Date d1(1, 1, 2021), d2(20, 1, 2021), exam(15, 1, 2021);
Date dd = clamp(exam, d1, d2);
cout << dd.GetDay() << endl; // prints 15
```

Algorithms *all_of*, *any_of* and *none_of*

bool_value = all_of(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, predicate);

Returns *true* if the predicate returns *true* for all the elements in range.

bool_value = any_of(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, predicate);

Returns *true* if the predicate returns *true* for at least one of the elements in range.

bool_value = none_of(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, predicate);

Returns *true* if the predicate returns *false* for all the elements in range.

The predicate may be a pointer to function, lambda expression or functor. Its argument must be element from the container range.

Example (see also http://www.cplusplus.com/reference/algorithm/all_of/ , http://www.cplusplus.com/reference/algorithm/any_of/ , http://www.cplusplus.com/reference/algorithm/none_of/):

```
vector<int> data = { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
bool b1 = all_of(data.begin(), data.end(), [](const int &i)->bool { return i < 0; });
bool b2 = any_of(data.begin(), data.end(), [](const int &i)->bool { return i < 0; });
bool b3 = none_of(data.begin(), data.end(), [](const int &i)->bool { return i < 0; });
cout << boolalpha << b1 << ' ' << b2 << ' ' << b3 << endl; // prints false true false
```

Algorithm *equal*

```
bool_value =  equal(container_1_name. iterator_to_first_element_of_range,
                    container_1_name. iterator_to_first_element_not_in_range,
                    container_2_name. iterator_to_first_element);
```

Returns *true* if the contents of ranges match. Range in the first container is specified by two iterators. For the second container only the beginning of the range is specified. The elements are compared using method *operator*==.

```
bool_value =  equal(container_1_name. iterator_to_first_element_of_range,
                    container_1_name. iterator_to_first_element_not_in_range,
                    container_2_name. iterator_to_first_element, comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its both arguments must be elements from the container range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

Example (see also http://www.cplusplus.com/reference/algorithm/equal/):

```
vector<int> data1= { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int> data2 = { 1, 4, 5, 7, 3, 8, 9, 12, 56, 45, 7 };
bool b1 = equal(data1.begin(), data1.end(), data2.begin());
bool b2 = equal(data1.begin(), data1.end(), data2.begin(),
                    [](const int &i1, const int &i2)->bool { return abs(i1) == abs(i2); });
cout << boolalpha << b1 << ' ' << b2 << endl; // prints false true
```

Algorithm *mismatch* (1)

```
pair_of_iterators_to_not_matching_elements =  mismatch(
                container_1_name. iterator_to_first_element_of_range,
                container_1_name. iterator_to_first_element_not_in_range,
                container_2_name. iterator_to_first_element);
```

Range in the first container is specified by two iterators. For the second container only the beginning of the range is specified. The elements are compared using method *operator*==. The output value is pair containing iterators to the first occurence of non-matching elements.

```
pair_of_iterators_to_not_matching_elements =  equal(
                container_1_name. iterator_to_first_element_of_range,
                container_1_name. iterator_to_first_element_not_in_range,
                container_2_name. iterator_to_first_element, comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its both arguments must be elements from the container range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

If all the elements match, the pair consists of iterators pointing to first element not in range.

Algorithm *mismatch* (2)

Example (see also http://www.cplusplus.com/reference/algorithm/equal/):

```
vector<int> data1= { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int> data2 = { 1, 4, 5, 7, 3, 8, 9, 12, 55, 45, 7 };
auto res1= mismatch(data1.begin(), data1.end(), data2.begin());
cout << *res1.first << ' ' << *res1.second << endl; // prints -4 4
auto res2= mismatch(data1.begin(), data1.end(), data2.begin(),
                    [](const int &i1, const int &i2)->bool { return abs(i1) == abs(i2); });
cout << *res2.first << ' ' << *res2.second << endl; // prints 56 55
```

Algorithm *count*

```
int counter = count(container_name. iterator_to_first_element_of_range,
        container_name. iterator_to_first_element_not_in_range, value_to_match);
```

Returns the number of elements from the specified range that are equal with the value to match.

Example (see also http://www.cplusplus.com/reference/algorithm/count/):

```
list<Date> deadlines = {….};
int n = 0;
for (auto& d : deadlines)
{
      if (d == Date(10, 1, 2019)
          n++;
}
```

The same with *count*:

```
int n = count(deadlines.begin(), deadlines.end(), Date(10, 1, 2019));
```

Algorithm *count_if*

```
int counter = count_if(container_name. iterator_to_first_element_of_range,
        container_name. iterator_to_first_element_not_in_range, predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. The body of predicate must check does the input value satisfies the condition to be counted and to return *true* or *false*.

count_if returns the number of elements for which the predicate has returned *true*.

Example (see also http://www.cplusplus.com/reference/algorithm/count_if/):

```
list<Date> deadlines = {….};
int n = 0;
for (auto& d : deadlines)
{
      if (d.GetMonth() == 1)
          n++;
}
```

The same with *count_if*:

```
int n = count_if(deadlines.begin(), deadlines.end(),
    [] (const Date& d)->bool { return d.GetMonth() == 1; } );
```

To find the total number of elements in range:

```
int n = count_if(deadlines.begin(), deadlines.end(), [] (auto)->bool { return true; } );
```

Algorithm *for_each*

for_each(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, operation_to_perform);

The operation_to_perform may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. Its body performs some operation on the element.

Example (see also http://www.cplusplus.com/reference/algorithm/for_each/):

```
list<Date> deadlines = { …. };
for (auto& d : deadlines)
{
      d.SetDay(d.GetDay() + 1);
}
for (auto& d : deadlines)
{
       cout << d.ToString() << endl;
}
```

The same with *for_each*:

```
for_each(deadlines.begin(), deadlines.end(), [](Date& d) { d.SetDay(d.GetDay() + 1); });
for_each(deadlines.begin(), deadlines.end(), [](Date& d) { cout << d.ToString() << endl;});
```

Algorithms *fill* and *fill_n*

```
fill(container_name. iterator_to_first_element_of_range,
     container_name. iterator_to_first_element_not_in_range, element_to_assign);
fill_n(container_name. iterator_to_first_element_of_range,
     number_of_elements_to_fill, element_to_assign);
```

The specified element is assigned to the elements in range.

Example (see also http://www.cplusplus.com/reference/algorithm/fill/ and http://www.cplusplus.com/reference/algorithm/fill_n/):

```
list<Date> deadlines(0);
for (int i = 0; i < 7; i++)
   deadlines.push_back(Date(1, 1, 2019));
```

The same with *fill*:

```
list<Date> deadlines(7);
fill(deadlines.begin(), deadlines.end(), Date(1, 1, 2019));
```

The same with *fill_n*:

```
list<Date> deadlines(7);
fill_n(deadlines.begin(), 7, Date(1, 1, 2019));
```

Algorithms *generate* and *generate_n*

```
generate(container_name. iterator_to_first_element_of_range,
         container_name. iterator_to_first_element_not_in_range, generator);
generate_n(container_name. iterator_to_first_element_of_range,
          number_of_elements_to_generate, generator);
```

The generator may be a pointer to function, lambda expression or functor. Its may not have any arguments. Its return values are one after another assigned to the elements in the container.

Example (see also http://www.cplusplus.com/reference/algorithm/generate/ and http://www.cplusplus.com/reference/algorithm/generate_n/):

```
list<Date> deadlines(0);
for (int i = 0; i < 10; i++)
  deadlines.push_back(CreateRandomDate(Date(1, 1, 2018), Date(31, 12, 2018)));
```

The same with *generate*:

```
list<Date> deadlines(10);
generate(deadlines.begin(), deadlines.end(),
   []()->Date { return CreateRandomDate(Date(1, 1, 2018), Date(31, 12, 2018)); });
```

The same with *generate_n*:

```
list<Date> deadlines(10);
generate_n(deadlines.begin(), 10,
   []()->Date { return CreateRandomDate(Date(1, 1, 2018), Date(31, 12, 2018)); });
```

Algorithms *copy* and *copy_n* (1)

copy(container_name_1. iterator_to_first_element_of_range,
container_name_1. iterator_to_first_element_not_in_range,
container_name_2. iterator_to_initial_position);

Copies all the elements from the range of *container_1* into *container_2* starting from specified position.

Example (see also http://www.cplusplus.com/reference/algorithm/copy/):

```
vector<int> data1 = { 1, 2, 3, 4, 5, 6, 7 };
vector<int> data2 = { 10, 20, 30, 40, 50, 60, 70 };
copy(data1.begin() + 2, data1.begin() + 4, data2.begin() + 2);
for_each(data2.begin(), data2.end(), [](int i) { cout << i << ' '; }); // get 10, 20, 3, 4, 50, 60, 70
```

copy does not insert new elements into the destination *container_2*. It simply replaces the existing ones with values from *container_1*.

Example:

```
vector<int> data3 = { 1, 7 };
copy(data1.begin(), data1.end(), data3.begin() + 1); // error – seven elements from data1
                                                     // cannot replace one element from data3
vector<int> data4(7);  // dimension is 7, filled with zeroes
copy(data1.begin(), data1.end(), data4.begin());
for_each(data4.begin(), data4.end(), [](int i) { cout << i << ' '; });  // get the full copy of data1
```

Algorithms *copy* and *copy_n* (2)

container_2 and *container_1* may be the same containers.

Example:

```
vector<int> data = { 1, 2, 3, 4, 5, 6, 7 };
copy(data.begin(), data.begin() + 2, data.begin() + 3);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; }); // get 1, 2, 3, 1, 2, 6, 7
```

Overlapping is allowed, i.e. the initial position may be in the specified range. Example:

```
vector<int> data = { 1, 2, 3, 4, 5, 6, 7 };
copy(data.begin(), data.begin() + 4, data.begin() + 2);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; }); // get 1, 2, 1, 2, 3, 4, 7
```

In *copy_n* the range is specified with the iterator to the first element and the number of elements:

```
copy_n(container_name_1. iterator_to_first_element_of_range, number of elements,
        container_name_2. iterator_to_initial_position);
```

Example (see also http://www.cplusplus.com/reference/algorithm/copy_n/):

```
vector<int> data = { 1, 2, 3, 4, 5, 6, 7 };
copy(data.begin(), 3, data.begin() + 3);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; }); // get 1, 2, 1, 2, 3, 6, 7
```

Algorithm *copy_if*

```
copy_if(container_name_1. iterator_to_first_element_of_range,
        container_name_1. iterator_to_first_element_not_in_range,
        container_name_2. iterator_to_initial_position, predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. The body of predicate must check does the input value satisfies some condition and return *true* or *false*.

The difference between *copy* and *copy_if* is that an element is copied only if the predicate for it returns *true*.

Example (see also http://www.cplusplus.com/reference/algorithm/copy_if/):

```
vector<int> data1 = { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int> data2(11);
copy_if(data1.begin(), data1.end(), data2.begin(), [](int i)->bool { return i >= 0;  });
for_each(data2.begin(), data2.end(), [](int i) { cout << i << ' '; });
                    // get 1, 5, 3, 9, 12, 56, 7, 0, 0, 0, 0
```

Insert iterators (1)

The copy algorithms overwrite data. With insert iterators we may force them to insert. There are three insert iterators: *back_insert_iterator*, *front_insert_iterator* and *insert_iterator*.

```
#include <iterator> // see http://www.cplusplus.com/reference/iterator/back_insert_iterator/
vector<int> data1 = { 1, 2 };
back_insert_iterator<vector<int> > it1(data1); // bind the iterator and container
```

it1 points to the end of vector. As any iterator, *back_insert_iterator* supports dereferencing and incrementing:

```
*it1 = 3; // the same as data1.push_back(3) and actually push_back is called
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3
cout << endl;
it1++; // shift the iterator to the end
*it1 = 4;
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3, 4
```

As iterator, *it1* may be the third parameter of copy methods:

```
vector<int> data2 = { 10, 20 };
copy(data2.begin(), data2.end(), it1);
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3, 4, 10, 20
```

So, we have appended vector *data2* to vector *data1*.

Here we learned more about copy algorithms. Do not forget that *it1* is a back insert (not ordinary) iterator. Remark that

```
copy(data2.begin(), data2.end(), data1.end()); // crashes, data1.end() is not a back insert iterator
```

Insert iterators (2)

There is a bit more convinient mode to use insert iterators. C++ standard function presented by template

```
template <class Container> back_insert_iterator<Container> back_inserter(Container & cont)
{ ... };
```

constructs a *back_insert_iterator* for container *cont*. Example:

```
#include <iterator> // see http://www.cplusplus.com/reference/iterator/back_inserter/
vector<int> data1 = { 1, 2 };
back_inserter(data1) = 3; // creates nameless back_insert_iterator and uses it for appending
                          // the same as auto it = back_inserter(data1); *it = 3;
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3
cout << endl;
vector<int> data2 = { 10, 20 };
copy(data2.begin(), data2.end(), back_inserter(data1));
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3, 10, 20
```

front_insert_iterator and *front_inserter* are very similar. They can be applied for containers supporting *push_front* method (for lists, but not for vectors). Example:

```
list<int> data1 = { 1, 2 }, data2 = { 10, 20 };
front_inserter(data1) = 6;
copy(data2.begin(), data2.end(), front_inserter(data1));
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; });
                                        // attention: prints 20, 10, 6, 1, 2
```

Insert iterators (3)

The general *insert_iterator* and *inserter* are for inserting into middle of containers. They need two arguments: the container and the common iterator to position from which the inserting should start. Examples:

```
vector<int> data1 = { 1, 3 }, data2 = { 10, 20 };
insert_iterator<vector<int> > it1(data1, data1.begin() + 1);
*it1 = 2;
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 3
cout << endl;

copy(data2.begin(), data2.end(), inserter(data1, data1.begin() + 2));
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; }); // prints 1, 2, 10, 20, 3

list<Date> deadlines = { Date(3, 1, 2020), Date(8, 1, 2020), Date(15, 1, 2020),
                         Date(20, 1, 2020) };
auto after_8 = find_if(deadlines.begin(), deadlines.end(), [](Date d) { return d.GetDay() > 8; });
vector<Date> additional = { Date(11, 1, 2020) }; // vector, not list!
copy(additional.begin(), additional.end(), inserter(deadlines, after_8));
for_each(deadlines.begin(), deadlines.end(), [](Date d) { cout << d.GetDay() << ' '; });
                                                    // prints 3, 8, 11, 15, 20
```

Algorithms *replace* and *replace_if*

```
replace(container_name. iterator_to_first_element_of_range,
        container_name. iterator_to_first_element_not_in_range,
        old_value, new_value);
```

Replaces all the elements from the range matching the old value with the new value.

```
replace_if(container_name. iterator_to_first_element_of_range,
           container_name. iterator_to_first_element_not_in_range,
           predicate, new_value);
```

The predicate may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. The body of predicate must check does the input value satisfies the some condition and return *true* or *false*. *replace_if* replaces only those elements from the range for which the predicate returns *true*.

Example (see also http://www.cplusplus.com/reference/algorithm/replace/ and http://www.cplusplus.com/reference/algorithm/replace_if/):

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
replace(data.begin(), data.end(), 1, 0);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 0, -4, 5, -7, 0, -8, 0, 12, 56, 0, 7
replace_if(data.begin(), data.end(), [](int i)->bool { return abs(i) == 7; }, 77 );
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
                    // get 0, -4, 5, 77, 0, -8, 0, 12, 56, 0, 77
```

Algorithms *remove* and *remove_if* (1)

remove(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range,
value_to_remove);

Removes all the elements from the range matching the specified value.

remove_if(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, predicate);

The predicate may be a pointer to function, lambda expression or functor. Its argument must be an element from the specified range. The body of predicate must check does the input value satisfies the some condition and return *true* or *false*. *remove_if* removes only those elements from the range for which the predicate returns *true*.

Example (see also http://www.cplusplus.com/reference/algorithm/remove/ and http://www.cplusplus.com/reference/algorithm/remove_if/):

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
remove(data.begin(), data.end(), 1);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get -4, 5, -7, -8, 12, 56, 7, 12, 56, 1, 7
remove_if(data.begin(), data.end(), [](int i)->bool { return abs(i) == 7; });
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
                    // get -4, 5, -8, 12, 56, 12, 56, 1, 56, 1, 7
```

Algorithms *remove* and *remove_if* (2)

Actually, those functions does not change the number of elements in the container. Elements to be kept are shifted to the beginning of range. After return the range without not needed elements is followed by some garbage. *remove* and *remove_if* return the iterator pointing to the first element of garbage. To get rid of garbage use the container method *erase*:

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
auto it1 = remove(data.begin(), data.end(), 1);
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get -4, 5, -7, -8, 12, 56, 7, 12, 56, 1, 7
data.erase(it1, data.end());
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get -4, 5, -7, -8, 12, 56, 7
auto it1 = remove_if(data.begin(), data.end(), [](int i)->bool { return abs(i) == 7; });
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl;    // get -4, 5, -8, 12, 56, 56, 7
data.erase(it2, data.end());
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get -4, 5, -7, -8, 12, 56
```

Container *list* has its own methods *remove()* and *remove_if()*.

Algorithm *sort*

```
sort(container_name. iterator_to_first_element_of_range,
     container_name. iterator_to_first_element_not_in_range);
```

The elements are compared using method *operator*<.

```
sort(container_name. iterator_to_first_element_of_range,
     container_name. iterator_to_first_element_not_in_range, comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its arguments must be elements from the container range. The body of comparator must check whether the first argument is considered to go before the second (return value *true*) or not (return value *false*).

Example (see also http://www.cplusplus.com/reference/algorithm/sort/):

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
sort(data.begin(), data.end());
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get -8, -7, -4, 1, 1, 1, 1, 5, 7, 12, 56
sort(data.begin(), data.end(), [](int i1, int i2)->bool { return abs(i1) < abs(i2); });
for_each(data.begin(), data.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 1, 1, 1, 1, -4, 5, -7, 7, -8, 12, 56
```

Container *list* has its own method for sorting. In containers *map*, *multimap*, *set* and *multiset* the elements are always sorted by default. It is not possible to sort unordered maps.

Algorithm *unique* (1)

unique(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range);,

The elements are compared using method *operator*==.

unique(container_name. iterator_to_first_element_of_range,
container_name. iterator_to_first_element_not_in_range, comparator);

The comparator may be a pointer to function, lambda expression or functor. Its arguments must be elements from the container range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

The implementation in Visual Studio runs correctly only when the elements in the range are in sorted order.

unique removes all the duplicates from the specified range. As *remove*, it does not not change the number of elements in the container. Elements to be kept are shifted to the beginning of range. After return the range containing only unique elements is followed by some garbage. *unique* return the iterator pointig to the first element of garbage. To get rid of garbage use the container method *erase*.

Algorithm *unique* (2)

Example (see also http://www.cplusplus.com/reference/algorithm/unique/):

```
vector<int> data1 = { 1, -4, -4, -4, 8, 9, 12, 56, 57, 77 };
auto it1 = unique(data11.begin(), data11.end());
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 1, -4, 8, 9, 12, 56, 57, 77, 57, 77
data1.erase(it1, data1.end());
for_each(data1.begin(), data1.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 1, -4, 8, 9, 12, 56, 57, 77
```

```
vector<int> data12 = { 1, -4, 4, 4, 8, 9, 12, 56, 57, 77 };
auto it4 = unique(data12.begin(), data12.end(), [](const int i1, const int i2)->bool { return
abs(i1) == abs(i2); });
for_each(data12.begin(), data12.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 1, -4, 8, 9, 12, 56, 57, 77, 57, 77
data12.erase(it4, data12.end());
for_each(data12.begin(), data12.end(), [](int i) { cout << i << ' '; });
cout << endl; // get 1, -4, 8, 9, 12, 56, 57, 77
```

Container *list* has its own method *unique()*.