# Functors (1)

Functors or function objects are objects that can be treated as though they are functions. An object of a class is a functor if in its class the function call is overloaded. Example:

```
class FunctorClass {
private:
   double Value;
public: // a class may include several operator functions having different signatures.
   FunctorClass(double d) : Value(d) { }
   double operator() () { return Value; } // overloads call to function that has no parameters
                                          // and returns a double
   void operator() (double); // overloads call to function that has a double parameter
                             // and returns nothing
};
void FunctorClass::operator() (double d) {
  Value = d;
}
```

Now

```
FunctorClass fn(5.0), *pfn = new FunctorClass(10.0);
fn(6); // actually fn.operator() (6) and Value is now 6, fn is an object treated as function
(*pfn)(6); // *pfn gives us an object
cout << fn() << endl; // actually cout << fn.operator()() and prints 6
cout << (*pfn)() << endl; // prints 6
```

# Functors (2)

Generally the operator that overloads the function call is written as:

return_value_type operator() (input_parameter_list) { function_body }

or if we have separate *.*h* and *.*cpp* files:

return_value_type operator() (input_parameter_list); // prototype in *.h
return_value_type class_name::operator() (input_parameter_list)
{ function_body } // definition is *.cpp

As a functor is an object, it has state (the collection of values of attributes). A function using variables with global lifetime has also state but ... A function has only one instance and the global variables it uses are freely attached and maybe modified by the other components of application or by the function itself:

```
int x = 0;
void fun() {
static int y = 0;
…….  // may modify x and / or y
}
int main() {
fun(); // after each call the state may be changed
……..
```

Advantage of functors: we may create any number of functors and each of them has its own encapsulated state.

# Functors (3)

```
class FunctorModifier {
private:
    int Coeff;
public:
    FunctorModifier(int i) : Coeff(i) { }
    int operator() (int i) { return i + Coeff; }
};
FunctorModifier fm1(1);  // modifies with coefficient 1
FunctorModifier fm2(2); // modifies with coefficient 2
cout << fm1(10) << ' ' << fm2(20) << endl;  // prints 11 and 22
                                            // actually fm1.operator() (10) and
                                            // fm2.operator() (20) were called
```

When using functions:

```
int Coeff = 1; // global
int FunModifier(int i)
{
    return i + Coeff;
}
cout << FunModifier(10) << endl;  // prints 11
Coeff = 2; // changes the global coefficient, serious side effects may occur if Coeff is also
           // used elsewhere in the application
cout << FunModifier(20) << endl; // prints 22
```

# Functors (4)

Functors may be used instead of pointers to functions. Let us have

```
void ProcessArray(int *p, int n, int i1, int i2, function<void(int)>pf)
{  // pf is a function wrapper
  ……..…… // checks input, throws exception if errors
  for (int i = i1;  i <= i2;  i++)   {
        (pf)(*(p + i)); // do something with each member of array
   }
}
class FunctorPrint {
  public: // no constructor, here we have just one method – the operator overloading
        void operator() (int x) const { cout << x << ' '; }
 };
```

Now:

```
int arr[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
FunctorPrint print; // compiler-created default constructor is applied
ProcessArray(arr, 9, 1, 5, print); // prints 2, 3, 4, 5
```

Value of argument *pf* is *print* – a functor, i.e. object of class in which call to function is overloaded. From this class an *operator()* with single argument of type *int* and without return value is searched and called.

# Functors (5)

More examples:

```
void QuadEq(double a, double b, function<double()>pf, double *px1, double *px2)
{
  double d = b * b – 4 * a * (pf)();
  ……..……..……..……..……..……..
}
FunctorClass fn(6.0), *pfn = new FunctorClass(6.0); // see slide Functors (1)
double x1, x2;
QuadEq(1, 5, fn, &x1, &x2); // actually d = b * b – 4 * a * fn()
                                    // or d = b * b – 4 * a * fn.operator()()
QuadEq(1, 5, fn() , &x1, &x2); // error
QuadEq(1, 5, *pfn, &x1, &x2); // dereferencing is applied
QuadEq(1, 5, FunctorClass(6.0), &x1, &x2); // creates nameless functor object, see
                                                    // slide Initializing (7)
QuadEq(1, 5, FunctorClass { 6.0 }, &x1, &x2);
QuadEq(1, 5, FunctorClass ffn = { 6.0 }, &x1, &x2); // error
```

Similarly:

```
ProcessArray(arr, 9, 1, 5, FunctorPrint());
ProcessArray(arr, 9, 1, 5, FunctorPrint { });
```

# Functors (6)

C++ defines several templates for functors that may replace simple functions and lambdas.
Example:

```
void QuadEq(double a, double b, function<double(double, double)>pf, double x, double y,
                    double *px1, double *px2)
{ // Here pf is a wrapper for functions with two double arguments, it returns also a double
   double d = b * b - 4 * a * (pf)(x, y);
  ……..
 }
plus<double> add; // template plus for adding two doubles, add is the functor name
double x1, x2;
try {
  QuadEq(1, 9, add, 7, 13, &x1, &x2);
  // here (pf)(x, y) is actually add(7, 13) and we get 20
}
catch (exception ex) {
   cout << ex.what() << endl;
}
```

There are also standard functors for other arithmetical operations (*minus*, *multiplies*, etc.), for comparison (*equal_to*, *less*, *greater*, etc.), logical operations and bitwise operations. See more on http://www.cplusplus.com/reference/functional/

# Random numbers (1)

Software-based random number generators rely on some mathematical formulas and are therefore pseudo-random numbers. To get truly random numbers we need some hardware attached to the computer.

Random number engine *random_device* tries to find a hardware generator and in case of failure selects a software algorithm. The standard does not specify which algorithm: the choice is up to the library designer. The other engines generate only pseudo-random numbers. To declare an engine without serious mathematical background is very difficult. Therefore C++ has several predefined engines.

In addition to engines we need also distributions that describe how the random numbers are distributed within a range. The C++ standard specifies 20 distribution classes.

Example:

```
#include <random> // see http://www.cplusplus.com/reference/random/
default_random_engine generator; // the simplest predefined engine, no parameters needed
int lower_bound = 0, upper_bound = 100;
uniform_int_distribution<int> distribution(lower_bound, upper_bound);
for (int i = 0;  i < 10;  i++)
   cout << distribution(generator) << endl;
                  // prints 10 pseudo-random numbers from range 0…100
```

# Random numbers (2)

Example:

```
mt19937 generator(static_cast<unsigned long int>(time(nullptr)));
    // mt19937 is a predefined engine of type Mersanne_twister_engine
    // Mersanne_twister_engine is considered to generate the highest quality of randomness
    // It needs seed, here the current time from computer clock
double mean = 0, deviation = 1.0;
normal_distribution<double> distribution(mean, deviation);
for (int i = 0;  i < 10;  i++)
   cout << distribution(generator) << endl;
                // prints 10 pseudo-random numbers
```

# Algorithms *for_each* and *for_each_n* in C++ before v.20 (1)

STL algorithm *for_each* is defined in the following way:

```
for_each(container. iterator_to_first_element_of_range,
         container. 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 operations on the element.

Examples (see also https://en.cppreference.com/w/cpp/algorithm/for_each.html):

```
#include <algorithm>
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
for (auto& i : data) {
   cout << i << ' ';
}
cout << endl; // prints the complete vector without applying STL algorithms
class Print {
    public: void operator() (auto& i) const { cout << i << ' '; }
};
for_each(data.begin(), data.end(), Print());
cout << endl; // prints the complete vector, operation is implemented by functor
for_each(data.begin(), data.end(), [] (const auto& i) { cout << i << ' '; });
cout << endl;  // prints the complete vector, , operation is implemented by lambda
```

# Algorithms *for_each* and *for_each_n* in C++ before v.20 (2)

More examples:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
for_each(data.begin() + 4, data.begin() + 9, [] (const int& i) { cout << i << ' '; });
cout << endl;  // prints 3 -8 8 12
int n = 0;
for_each(data.begin(), data.end(), [&] (const auto& i) { if (i < 0) n++});
cout << n << endl;  // prints 3

list<Date> deadlines = { …. };
for_each(deadlines.begin(), deadlines.end(), [](const Date& d) { d.SetDay(d.GetDay() + 1); });
```

From C++ v. 17:

```
for_each_n(container. iterator_to_first_element_of_range, number of elements,
                operation_to_perform);
```

The operation is performed on the specified number of elements.

Example (see https://en.cppreference.com/w/cpp/algorithm/for_each_n.html):

```
for_each_n(data.begin(), 4, [] (const int& i) { cout << i << ' '; });
cout << endl;  // prints 1 4 5 -7
```

# Ranges and views (1)

As an abstraction, a range (introduced in C++ v. 20) is something that you can iterate over. A range is represented by an iterator that marks the beginning of the range and a sentinel that marks the end of the range. The sentinel may be the same type as the begin iterator (so called common ranges), or it may be different.

All the STL containers (vectors, lists, maps, etc.) may be considered as ranges. Their iterators marking the beginning and the end are of the same type. C-style arrays are also ranges.

Ranges have their own namespace *std::ranges*. In it several non-member methods as *begin, end, cbegin, cend, rbegin, rend, crbegin, crend, size, empty,* etc. are defined. The argument of those functions is the range name.

Example:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
```

Here *data.begin()* is the same as *ranges::begin(data)*, *data.empty()* is the same as *ranges::empty(data), data.begin() + 3* is the same as *ranges::begin(data) + 3,* etc.

Most (but not all) of the STL standard algorithms have an equivalent from the ranges namespace.

See also:

https://en.cppreference.com/w/cpp/ranges/range.html
https://www.geeksforgeeks.org/cpp-20-ranges-library/
https://hannes.hauswedell.net/post/2019/11/30/range_intro/

# Ranges and views (2)

So, from C++ v. 20 we have:

```
for_each(range. iterator_to_first_element_of_range,
            range. iterator_to_first_element_not_in_range, operation_to_perform);
ranges::for_each(range, operation_to_perform);
```

Examples:

```
#include <algorithm>
#include <ranges>
for_each(data.begin(), data.end(), [] (const auto& i) { cout << i << ' '; });
cout << endl; // vector data is a range, prints the complete vector
for_each(ranges::begin(data), ranges::end(data), [] (const auto& i) { cout << i << ' '; });
cout << endl; // iterators from ranges library, prints the complete vector
ranges::for_each(data, [] (const auto& i) { cout << i << ' '; });
cout << endl; // method from ranges library, prints the complete vector
```

```
for_each_n(range. iterator_to_first_element_of_range, number of elements,
                operation_to_perform);
ranges::for_each_n(range. iterator_to_first_element_of_range, number of elements,
                        operation_to_perform);
```

Here the both functions are actually identical. Example:

```
ranges::for_each_n(data.begin(), 4, [] (const int& i) { cout << i << ' '; });
cout << endl;  // prints 1 4 5 -7
```

# Ranges and views (3)

A view is a range containing elements by some rules filtered from its base range. Views are specified by view adapters. For example, view adapter filter

```
views::filter(data, [] (const auto& x) { return x > 0; }); // namespace views is nested into
                                                           // namespace ranges
```

gives us a range containing the positive elements of vector *data*. Consequently, instead of

```
for_each(data.begin(), data.end(), [] (const auto& i) { if (i > 0) cout << i << ' '; });
```

we can write

```
ranges::for_each(views::filter(data, [] (const auto& x) { return x > 0; }),
                 [] (const int& x) { cout << x << ' '; });
```

A view may have name:

```
auto view1 = views::filter(data, [](const int& i) { return i > 0; });
```

or using alternative syntacs:

```
auto view2 = data | views::filter([](const int& i) { return i > 0; });
```

Constructing a view does not perform any operations on the range, i.e. a view is just a rule and does not own any elements. The operations on a view are applied only when the iterating over the elements is going on.

# Ranges and views (4)

Examples of different view adapters (the complete list of view adapters is on https://en.cppreference.com/w/cpp/ranges):

```
auto print = [](const int& x) { cout << x << ' '; };
ranges::for_each(views::all(data), print); // the view contains the complete range
cout << endl;  //  prints 1 4 5 -7 3 -8 9 12 56 -45 7
ranges::for_each(views::take(data, 2), print); // the view contains the first two elements of
                                               // range
cout << endl; //  prints 1 4
ranges::for_each(views::drop(data, 2), print); // the view contains all the elements of range
                                               // except the first two
cout << endl; //  prints 5 -7 3 -8 9 12 56 -45 7
ranges::for_each(views::take_while(data, [](const int& i) { return i > 0; }), print); // the view
                                               // contains elements until the first negative value
cout << endl; //  prints 1 4 5
```

As the views are ranges, we can also use them in range-based *for* loops (see chapter *Containers* in course IAX0586):

```
for (const auto& i : views::filter(data, [](const int& i) { return i > 0; })) {
    cout << i << ' ';
}
cout << endl; // prints 1 4 5 3 9 12 56 7 (negative values are filtered out)
```

# Ranges and views (5)

Turn attention that as the view never owns any elements, using view adapter *transform* you cannot store the changes:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto view1 = views::transform(data, [](const int& i) { return i > 0 ? i * 2 : i; });
ranges::for_each(view1, print);
cout << endl; // prints 2 8 10 -7 6 -8 18 24 112 -45 14 (positive values multiplied by 2)
ranges::for_each(data, print);  // prints still 1 4 5 -7 3 -8 9 12 56 -45 7
```

But the following code works:

```
auto view2 = data | views::filter([](const int& i) { return i > 0; });
for (auto& i : view2) {
    i *= 2;
}
ranges::for_each(data, print); // prints 2 8 10 -7 6 -8 18 24 12 -45 14
cout << endl;
```

Similarly:

```
ranges::for_each(views::reverse(data), print);
cout << endl; // prints: 7 -45 56 12 9 -8 3 -7 5 4 1
ranges::for_each(data, print); // prints still 1 4 5 -7 3 -8 9 12 56 -45 7
```

# Ranges and views (6)

Ranges and views can be connected into pipeline:

```
for (const auto& i : data | views::filter([](const int& i) { return i > 0; }) |
                            views::filter([](const int& i) { return i % 2;  } )) {
    cout << i << ' ';
}
cout << endl; // result: 1, 5, 3, 9, 7 (only positive and odd numbers)
```

There are some possibilities to generate a view as a sequence of values without filtering an existing range:

```
auto view1 = views::iota(0);  // endless sequence 0, 1, 2, …. (remember that a view does not
                              // own elements)
for(int x : view1 | views::take(10) {
    cout << x << ' '; // prints 0 1 2… 9
}
auto view2 = views::iota(0, 10); // sequence 0 1 2 …9
for(int x : view2) {
    cout << x << ' '; // prints 0 1 2 … 9
}
auto view3 = views::repeat("Help me!") | views::take(2); // from C++ 23
for(auto x : view3) {
    cout << x << ' '; // prints "Help me! Help me!
}
```

# Ranges and views (7)

*ranges::subrange* is a view containing a section from range.

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
```

and we want to print the middle section of vector. The needed subrange may be defined as follows:

```
ranges::subrange sub1 { data.begin() +4, data.begin() +9 };
    // specify by the iterator to beginning of the subrange and the sentinel
auto sub2 { views::counted(data.begin() + 4, 5) };
    // specify using view adapter counted, arguments are  the iterator to beginning of the
    //  subrange and the number of elements
ranges::for_each(sub1, [] (auto &x) { cout << x << ' '; });
cout << endl;  // prints 3 -8 9 12 56
ranges::for_each(sub2, [] (auto &x) { cout << x << ' '; });
cout << endl; // prints 3 -8 9 12 56
```

View adapters *keys* and *values* allow to separate keys and values of maps:

```
map<string, Date> deadlines = { { "Mathematics", Date(5, 1, 2022) }, { "Chemistry",
Date(10, 1, 2022) },  { "Physics", Date(15, 1, 2022) } };
ranges::for_each(deadlines | views::keys, [] (auto &x) { cout << x << ' '; });
cout << endl; // prints Chemistry Mathematics Physics
ranges::for_each(views::values(deadlines), [] (auto &x) { cout << x.GetDay() << ' '; });
cout << endl; // prints 10 5 15
```

# Ranges and views (8)

As the views are ranges you can iterate through them. Example:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
for_each(views::take(data, 4).begin(), views::take(data, 4).end(), [](const int& x)
                                                                    { cout << x << ' '; });
cout << endl; // prints 1 4 5 -7
```

You can convert a range (or view) into a new container:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto view = views::transform(data, [](const int& i) { return i > 0 ? i * 2 : i; });
vector<int> new_data1(view.begin(), view.end()); // define new vector
ranges::for_each(new_data1, print);
cout << endl; // results: 2 8 10 -7 6 -8 18 24 112 -45 14
```

C++ v.23 has even better method:

```
vector<int> new_data2 = ranges::to<vector>(view);
```

# 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 https://en.cppreference.com/w/cpp/algorithm/find.html
auto it = find(deadlines.begin(), deadlines.end(), Date(5, 1, 2019));
cout << (it == deadlines.end() ? "Not found" : "Found") << endl;
```

# Algorithm *find* (2)

Generally:

```
iterator_to_result =  find(range. iterator_to_first_element_of_range,
                           range. iterator_to_first_element_not_in_range,
                          element_to_find);
iterator_to_result =  ranges::find(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*.

You can use method *find* for searching from C-style arrays:

```
int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto res = find(&array[0], &array[10], 11);
if (res != &array[5])
   cout << *res << endl;  // prints 5
else
   cout << "Failed" << endl;
```

# Algorithm *find_if* (1)

```
iterator_to_result =  find_if(range. iterator_to_first_element_of_range,
                              range. iterator_to_first_element_not_in_range, predicate);
iterator_to_result =  ranges::find_if(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 https://en.cppreference.com/w/cpp/algorithm/find.html):

```
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 = ranges::find_if(deadlines,
                          [](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 = ranges::find_if(deadlines,
                          [](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(range. iterator_to_first_element_of_range,
                                  range. iterator_to_first_element_not_in_range,
                                  predicate);
iterator_to_result =  ranges::find_if_not(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*.

Example:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto it = ranges::find_if_not(data, [](auto x)->bool { return x > 0; });
if (it == data.end())
    cout << "All the elements are positive" << endl;
else
    cout << *it << " is the first negative element" << endl;  // prints -7
```

See also https://en.cppreference.com/w/cpp/algorithm/find.html.

# Algorithm *find_first_of* (1)

```
iterator_to_result =  find_first_of(
                      range_1. iterator_to_first_element_of_range,
                      range _1. iterator_to_first_element_not_in_range,
                      range _2. iterator_to_first_element_of_range,
                      range _2. iterator_to_first_element_not_in_range);
iterator_to_result =  ranges::find_first_of(range_1, range_2);
```

*find_first_of* returns the iterator to the first element from the range of *range_1* that is equal to any element from the range of *range_2*. In case of failure returns the iterator to the first element not in range from *range_1*.

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

```
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 = ranges::find_first_of(deadlines, sundays);
if (it != deadlines.end())
    cout << "One of the deadlines is on Sunday: " << it->GetDay() << ".Jan" << endl;
```

# Algorithm *find_first_of* (2)

```
iterator_to_result =  find_first_of(
                      range_1. iterator_to_first_element_of_range,
                      range_1. iterator_to_first_element_not_in_range,
                      range_2. iterator_to_first_element_of_range,
                      range_2. iterator_to_first_element_not_in_range,
                      predicate);
iterator_to_result =  ranges::find_first_of(range_1,  range_2, predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its first argument must be element from the range of *range_1* and the second argument must be element from the range of *range_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 = ranges::find_first_of(deadlines, sundays,
          [](const pair<string, Date>& x, const Date& d)->bool { return x.second == d; });
cout << "Exam of " << it->first.c_str() << " is on Sunday" << endl; // prints Chemistry
```

Turn attention that the containers are from different types.

# Algorithms *search* and *find_end* (1)

```
iterator_to_result =  search( range_1. iterator_to_first_element_of_range,
                              range_1. iterator_to_first_element_not_in_range,
                              range_2. iterator_to_first_element_of_range,
                              range_2. iterator_to_first_element_not_in_range);
```

*search* tries to find the first occurrence of range of *range_2* from the range of *range_1*. Actually, *range_2* specifies a pattern to be found from *range_1*.

*find_end* has same the arguments. The difference is that it tries to find the last occurrence.

In the following example (see also http://www.cplusplus.com/reference/algorithm/search/ and http://www.cplusplus.com/reference/algorithm/find_end/ ) we search the first occurrence of pattern *{ -7, 3, 8 }* in vector *data:*

```
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());
if (it == data.end())
    cout << "Not found" << endl;
else
     cout << "Found, starts from position " << (it - data.begin()) << endl;
                                                        // prints 3 (index of the found pattern)
```

# Algorithms *search* and *find_end* (2)

```
ranges::subrange result =  ranges::search(range_1, range_2);
```

Here the result is not an iterator but a subrange of range_1.

Examples:

```
vector<int> data1 = { 1, 4, 5, -7, 3, 8, 9, 12, 56, -45, 7 };
vector<int> pattern1 = { -7, 3, 8 };
auto res = ranges::search(data1, pattern1);
if (res.empty())
    cout << "Not found" << endl;
else
    cout << "Found, starts from position " << distance(data1.begin(), res1.begin()) << endl;
                                                    // prints 3 (index of the found pattern)
vector<int> data2 = { 1, 4, 5, -7, 3, 8, 9, 12, 1, 4, 5, 3, 8, 9 };
vector<int> pattern2 = { -7, 3, 8 };
auto res2 = ranges::find_end(data2, pattern2);
if (res.empty())
    cout << "Not found" << endl;
else
    cout << "Found, starts from position " << distance(data2.begin(), res2.begin()) << endl;
                                                    // prints 8 (index of the found pattern)
```

# Algorithms *search and find_end* (3)

```
iterator_to_result =  search(
                      range_1. iterator_to_first_element_of_range,
                      range_1. iterator_to_first_element_not_in_range,
                      range_2. iterator_to_first_element_of_range,
                      range_2. iterator_to_first_element_not_in_range,
                      predicate);
ranges::subrange result =  ranges::search(range_1, range_2, predicate);
```

The predicate may be a pointer to function, lambda expression or functor. Its first argument must be element from the range of *range_1* and the second argument must be element from the range of *range_2*. The body of predicate must check does the input values satisfy the search condition and return *true* or *false.*

Example:

```
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); });
if (it == data.end())
    cout << "Not found" << endl;
else
     cout << "Found, starts from position " << (it - data.begin()) << endl;
                                                    // prints 3 (index of the found pattern)
```

# Algorithms *lower_bound* and *upper_bound* (1)

```
iterator_to_result = lower_bound(range. iterator_to_first_element_of_range,
                                 range. iterator_to_first_element_not_in_range,
                                 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*<. The range must be sorted.

```
iterator_to_result = lower_bound(range. iterator_to_first_element_of_range,
                                 range. iterator_to_first_element_not_in_range,
                                 lower_bound_value, comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its first argument must be element from the range 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 https://en.cppreference.com/w/cpp/algorithm/lower_bound.html):

```
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)
```

# Algorithms *lower_bound* and *upper_bound* (2)

iterator_to_result = upper_bound(range. iterator_to_first_element_of_range,
range. iterator_to_first_element_not_in_range,
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*<. The range must be sorted.

iterator_to_result = upper_bound(range. iterator_to_first_element_of_range,
range. iterator_to_first_element_not_in_range,
upper_bound_value, comparator);

The comparator may be a pointer to function, lambda expression or functor. Its second argument must be element from the range range and the actual value of first 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 > i2 *i2; });
cout << it2 - data.begin() << endl; // prints 2 (index of 5 as 5*5 is first number greater than  20)
```

# Algorithms *lower_bound* and *upper_bound* (3)

```
iterator_to_result = ranges::lower_bound(range, lower_bound_value);
iterator_to_result = ranges::lower_bound(range, lower_bound_value, comparator);
iterator_to_result = ranges::upper_bound(range, upper_bound_value);
iterator_to_result = ranges::upper_bound(range, upper_bound_value, comparator);
```

Examples:

```
vector<int> data = { 1, 4, 5, 7, 8, 9, 12, 56, 145, 734 };
auto it1 = ranges::lower_bound(data, 10);
cout << it1 - data.begin() << endl; // prints 6 (index of 12 as the first number exceeding 10)
auto it2 = ranges::lower_bound(data, 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)
auto it3 = ranges:: upper_bound(data.begin(), data.end(), 60);
cout << it3 - data.begin() << endl; // prints 8 (index of 145 as the first number greater than 60)
auto it4 = ranges:: upper_bound(data, 20, [] (const int &i1, const int i2) { return i1 > i2 *i2; });
cout << it4 - data.begin() << endl; // prints 2 (index of 5 as 5*5 is first number greater 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 is the first of them.

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

```
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 https://en.cppreference.com/w/cpp/algorithm/max.html

# Finding minimum and maximum (2)

```
iterator_to_result = min_element(range. iterator_to_first_element_of_range,
                                 range. iterator_to_first_element_not_in_range);
iterator_to_result = min_element(range. iterator_to_first_element_of_range,
                                 range. iterator_to_first_element_not_in_range,
                                 comparator);
iterator_to_result = ranges::min_element(range);
iterator_to_result = ranges:: min_element(range, comparator);
```

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

Examples (see also https://en.cppreference.com/w/cpp/algorithm/ranges/min_element.html):

```
vector<int> data = { 12, 9, -8, 145, -56, 734, 1, 4, 7, 5 };
auto it1 = min_element(data.begin(), data.end());
cout << *it1 << endl;  // prints -56
auto it2 = ranges::min_element(data);
cout << *it2 << endl; // prints -56
auto it3 = min_element(data.begin(), data.end(), [](const int& i1, const int& i2)->bool
                                                 { return abs(i1) < abs(i2); });
cout << *it3 << endl; // prints 1
auto it4 = ranges::min_element(data, [](const int& i1, const int& i2)->bool
                                     { return abs(i1) < abs(i2); });
cout << *it14 << endl; // prints 1
```

See also https://en.cppreference.com/w/cpp/algorithm/max_element.html

# Finding minimum and maximum (3)

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

The elements are compared using method *operator*<.

```
pair_of_iterators_to_min_and_max = minmax_element(
                                   range. iterator_to_first_element_of_range,
                                   range. 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 range 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 https://en.cppreference.com/w/cpp/algorithm/minmax_element.html):

```
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 (4)

result = ranges::minmax_element(range);

The elements are compared using method *operator*<.

result= ranges::minmax_element(range, comparator);

Here the result is an object specified by template *ranges::minmax_elemet_result*. It has two attributes*: min* is iterator to minimum and *max* is the iterator to maximum.

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

Examples:

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

# Finding minimum and maximum (5)

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.

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

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* (1)

bool_value = all_of(range. iterator_to_first_element_of_range,
range. 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(range. iterator_to_first_element_of_range,
range. 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(range. iterator_to_first_element_of_range,
range. 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 range range.

bool_value = ranges::all_of(range, predicate);

bool_value = ranges:: any_of(range, predicate);

bool_value = ranges:: none_of(range, predicate);

# Algorithms *all_of*, *any_of* and *none_of* (2)

Examples (see also https://en.cppreference.com/w/cpp/algorithm/all_any_none_of.html):

```
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
bool b4 = ranges::all_of(data, [](const int &i)->bool { return i < 0; });
bool b5 = ranges::(data, [](const int &i)->bool { return i < 0; });
bool b6 = ranges::(data, [](const int &i)->bool { return i < 0; });
cout << boolalpha << b4 << ' ' << b5 << ' ' << b6 << endl; // prints false true false
```

# Algorithm *equal* (1)

```
bool_value =  equal(range_1. iterator_to_first_element_of_range,
                    range_1. iterator_to_first_element_not_in_range,
                    range_2. iterator_to_first_element);
```

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

```
bool_value =  equal(range_1. iterator_to_first_element_of_range,
                    range_1. iterator_to_first_element_not_in_range,
                    range_2. 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 range range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

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

```
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 *equal* (2)

```
bool_value =  ranges::equal(range_1, range_2);
bool_value =  ranges::equal(range_1, range_2, comparator);
```

Examples:

```
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,data2);
bool b2 = equal(data1.data2, [](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(
range_1. iterator_to_first_element_of_range,
range_1. iterator_to_first_element_not_in_range,
range_2. iterator_to_first_element);

Range in the first range is specified by two iterators. For the second range 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. If all the elements match, the pair consists of iterators pointing to first element not in range.

pair_of_iterators_to_not_matching_elements = mismatch(
range_1. iterator_to_first_element_of_range,
range_1. iterator_to_first_element_not_in_range,
range_2. 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 range range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

result = ranges:: mismatch(range_1, range_2);
result = ranges:: mismatch(range_1, range_2, comparator);

Here the result is an object specified by template *ranges::in_in_result*. It has two attributes: *in1* is the iterator to first occurence of non-matching element from *range_1* and *in2* is the iterator to its counterpart from *range_2*.

# Algorithm *mismatch* (2)

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

```
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
auto res3= ranges::mismatch(data1, data2);
cout << *res3.in1 << ' ' << *res3.in2 << endl; // prints -4 4
auto res4= mismatch(data1, data2,  [](const int &i1, const int &i2)->bool
                                    { return abs(i1) == abs(i2); });
cout << *res4.in1 << ' ' << *res4.in2 << endl; // prints 56 55
```

# Algorithms *count* and *count_if*

```
int counter = count(range. iterator_to_first_element_of_range,
        range. 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.

```
int counter = count_if(range. iterator_to_first_element_of_range,
        range. 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*. *R*eturns the number of elements for which the predicate has returned *true*.

```
int counter = ranges::count(range, value_to_match);
int counter = ranges:: count_if(range, predicate);
```

Examples (see also https://en.cppreference.com/w/cpp/algorithm/count.html):

```
vector<int> data = { 1, 4, 5, -7, 3, 8, 9, -12, 1, 4, 5, 3, 8, -9 };
cout << count(data.begin(), data.end(), 4) << endl; // prints 2
cout << count_if(data.begin(), data.end(), [](const int& i)->bool { return i < 0; }) << endl;
                                                                    // prints 3
cout << ranges::count(data, 4) << endl; // prints 2
cout << ranges::count_if(data, [](const int& i)->bool { return i < 0; }) << endl; // prints 3
```

# Algorithms *fill* and *fill_n* (1)

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

The specified element is assigned to the elements in range.

Examples (see also https://en.cppreference.com/w/cpp/algorithm/fill.html and https://en.cppreference.com/w/cpp/algorithm/fill_n.html):

```
fill(v.begin(), v.end(), 1);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 1 1 1 1 1
fill(v.begin() + 1, v.begin() + 3, 2);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 1 2 2 1 1
fill_n(v.begin(), 5, 3);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 3 3 3 3 3
v.clear();
cout << v.size() << endl;   // prints 0
fill_n(v.begin(), 5, 1); // ERROR, fill_n does not expand the range
```

# Algorithms *fill* and *fill_n* (2)

```
ranges::fill(range, element_to_assign);
ranges::fill_n(iterator_to_first_element_of_range, number_of_elements_to_fill,
                element_to_assign);
```

Examples:

```
vector<int> v(5); // initialized with zeroes
ranges::fill(v, 1);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 1 1 1 1 1
ranges::fill(views::counted(v.begin() + 1, 2), 2);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 1 2 2 1 1
ranges::fill_n(v.begin(), 5, 3);
ranges::for_each(v, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 3 3 3 3 3
v.clear();
cout << v.size() << endl; // empty
ranges::fill (v, 1); // not error but does nothing
```

# Algorithms *generate* and *generate_n* (1)

```
generate(range. iterator_to_first_element_of_range,
         range. iterator_to_first_element_not_in_range, generator);
generate_n(range. 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 range.

Examples (see also https://en.cppreference.com/w/cpp/algorithm/generate.html and https://en.cppreference.com/w/cpp/algorithm/generate_n.html):

```
vector<int>* pv = new vector<int>(5);
default_random_engine gen;
uniform_int_distribution<int> distr(0, 100);  // random numbers between 0 and 100
generate(pv->begin(), pv->end(), [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl; // prints 82 13 91 84 12
generate(pv->begin() + 1, pv->begin() + 3, [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl; // prints 82 97 92 84 12
generate_n(pv->begin(), 5, [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl; // prints 22 63 31 9 55
generate_n(pv->begin(), 6, [&]()->int { return distr(gen);  }); // ERROR, no expanding
```

# Algorithms *generate* and *generate_n* (2)

```
ranges::generate(range, generator);
ranges::generate_n(iterator_to_first_element_in_range, number_of_elements_to_generate,
                   generator);
```

Examples:

```
vector<int>* pv = new vector<int>(5);
default_random_engine gen;
uniform_int_distribution<int> distr(0, 100);  // random numbers between 0 and 100
ranges::generate(*pv, [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl;  // prints 28 19 55 100 96
ranges::generate(views::counted(pv->begin() + 1, 2), [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl; // prints 28 100 97 100 96
ranges::generate_n(pv->begin() , 5, [&]()->int { return distr(gen);  });
ranges::for_each(*pv, [](const auto& i) { cout << i << ' ';  });
cout << endl; // prints 97 15 73 98 99
```

# Algorithms *copy*, *copy_n* and *copy_if* (1)

```
copy(range_1. iterator_to_first_element_of_range,
     range_1. iterator_to_first_element_not_in_range,
     range_2. iterator_to_initial_position);
```

Copies all the elements from the range of *range_1* into *range_2* starting from the specified position.

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

```
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);
ranges::for_each(data2, [](int i) { cout << i << ' '; }); // prints 10 20 3 4 50 60 70
```

*copy* does not insert new elements into the destination *range_2*. It simply replaces the existing ones with values from *range_1*.

Example:

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

# Algorithms *copy*, *copy_n* and *copy_if* (2)

*range_2* and *range_1* may be the same ranges.

Example:

```
vector<int> data = { 1, 2, 3, 4, 5, 6, 7 };
copy(data.begin(), data.begin() + 2, data.begin() + 3);
ranges::for_each(data, [](int i) { cout << i << ' '; }); // prints  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);
ranges::for_each(data, [](int i) { cout << i << ' '; }); // prints 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(range_1. iterator_to_first_element_of_range, number of elements,
        range_2. iterator_to_initial_position);
```

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

```
vector<int> data = { 1, 2, 3, 4, 5, 6, 7 };
copy_n(data.begin(), 3, data.begin() + 3);
ranges::for_each(data, [](int i) { cout << i << ' '; }); // prints 1 2 3 1 2 3 7
                                                         // operations on  the same range with overlapping
```

# Algorithms *copy*, *copy_n* and *copy_if* (3)

```
copy_if(range_1. iterator_to_first_element_of_range,
        range_1. iterator_to_first_element_not_in_range,
        range_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 https://en.cppreference.com/w/cpp/algorithm/copy.html):

```
vector<int> data1 = { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int> data2(11);  // filled with zeroes
copy_if(data1.begin(), data1.end(), data2.begin(), [](int i)->bool { return i >= 0;  });
ranges::for_each(data2, [](int i) { cout << i << ' '; });  // prints1 5 3 9 12 56 7 0, 0, 0, 0
                                                           // only non-negative values were copied
```

# Algorithms *copy*, *copy_n* and *copy_if* (4)

```
ranges::copy(source_range, iterator_to_initial_position_in_destination);
```

Examples:

```
vector<int> *pData1 = new vector<int> { 1, 2, 3, 4, 5, 6, 7 };
vector<int>* pData2 = new vector<int>(7);
vector<int> *pData3 = new vector<int> { 10, 20, 30, 40, 50, 60, 70 };
vector<int>* pData4 = new vector<int>{ 1, 2, 3, 4, 5, 6, 7 };
vector<int>* pData5 = new vector<int>{ 1, 2, 3, 4, 5, 6, 7 };
ranges::copy(*pData1, pData2->begin());
ranges::for_each(*pData2, [](int i) { cout << i << ' '; });
cout << endl; // prints the full copy of source vector
ranges::copy(views::counted(pData1->begin() + 2, 2), pData3->begin() + 2);
ranges::for_each(*pData3, [](int i) { cout << i << ' '; });
cout << endl; // prints 10 20 3 4 50 60 70
ranges::copy(views::take(*pData4, 2), pData4->begin() + 3);
ranges::for_each(*pData4, [](int i) { cout << i << ' '; });
cout << endl; // prints  1 2 3 1 2 6 7, operations on  the same range
ranges::copy(views::take(*pData5, 4), pData5->begin() + 2);
ranges::for_each(*pData5, [](int i) { cout << i << ' '; });
cout << endl; // prints 1 2 1 2 3 4 7 , operations on  the same range with overlapping
```

# Algorithms *copy, copy_n* and *copy_if* (5)

```
ranges::copy_n( iterator_to_first_element_in_source_range, number of elements,
                iterator_to_initial_position_in_destination);
```

Example:

```
vector<int>* pData = new vector<int>{ 1, 2, 3, 4, 5, 6, 7 };
ranges::copy_n(pData->begin(), 3, pData->begin() + 3);
ranges::for_each(*pData, [](int i) { cout << i << ' '; });
cout << endl; // prints 1 2 3 1 2 3 7, operations on  the same range with overlapping
```

```
ranges::copy_if(source_range, iterator_to_initial_position_in_destination, predicate);
```

Example:

```
vector<int> *pData1 = new vector<int> { 1, -4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
vector<int>* pData2 = new vector<int>(11);
ranges::copy_if(*pData1, pData2->begin(), [](int i)->bool { return i >= 0;  });
ranges::for_each(*pData2, [](int i) { cout << i << ' '; });
cout << endl; // prints1 5 3 9 12 56 7 0, 0, 0, 0; only non-negative values were copied
```

Other algorithms for copying: *ranges::copy_backward, ranges::reverse_copy, ranges::rotate_copy, ranges::unique_copy*.

All the copying algorithms specified in the *ranges* namespace have output value specified by template *ranges::in_out_result.*

A good overview of algorithms for copying: https://www.studyplan.dev/pro-cpp/copy-algorithms

# 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 https://en.cppreference.com/w/cpp/iterator/back_insert_iterator.html
vector<int> data1 = { 1, 2 };
back_insert_iterator<vector<int> > it1(data1); // bind the iterator and range
```

*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
ranges::for_each(data1, [] (const auto& i) { cout << i << ' '; }); // prints 1 2 3
cout << endl;
it1++; // shift the iterator to the end
*it1 = 4;
ranges::for_each(data1, [] (const auto& i) { cout << i << ' '; }); // prints 1 2 3 4
```

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

```
vector<int> data2 = { 10, 20 };
ranges::copy(data2, it1);  // it1 points to the end of destination
ranges::for_each(data1, [] (const auto& 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

```
ranges::copy(data2, 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  https://en.cppreference.com/w/cpp/iterator/back_inserter.html
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 };
ranges::copy(data2, back_inserter(data1));
ranges::for_each(data1, [] (const auto& 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;
ranges::copy(data2, front_inserter(data1));
ranges::for_each(data1, [] (const auto& i) { cout << i << ' '; }); // attention: prints 20, 10, 6, 1, 2
```

See https://en.cppreference.com/w/cpp/iterator/front_inserter.html

# 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. See https://en.cppreference.com/w/cpp/iterator/inserter.html. Examples:

```
vector<int> data1 = { 1, 3 }, data2 = { 10, 20 };
insert_iterator<vector<int> > it1(data1, data1.begin() + 1);
*it1 = 2;
ranges::for_each(data1, [] (const int& i) { cout << i << ' '; }); // prints 1 2 3
ranges::copy(data2, inserter(data1, data1.begin() + 2));
ranges::for_each(data1, [](const 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 = ranges::find_if(deadlines, [] (const Date& d) { return d.GetDay() > 8; });
vector<Date> additional = { Date(11, 1, 2020) }; // vector, not list!
ranges::copy(additional, inserter(deadlines, after_8));
ranges::for_each(deadlines.begin(), deadlines.end(), [] (const Date& d) // prints 3 8 11 15 20
                                                { cout << d.GetDay() << ' '; });
```

The insert iterators offer an alternative to methods like *vector::push_back, vector::insert, list::push_back, list::push_front, list::insert*, etc. Their advantage is the universality as they can be applied for any type of containers that allow to add new elements to the beginning, to the end or into the middle.

# Stream iterators (1)

The stream iterators allow us to use a streams like *cin* ot *cout* as a source or destination of algorithms like *copy*. They are defined as follows:

ostream_iterator<data_type> (stream)
ostream_iterator<data_type> (stream, delimiter between values)
istream_iterator<data_type> ()
istream_iterator<data_type> (stream)

Examples:

```
vector<string> names;  // into it we want to read a sequence of words from keyboard
copy(istream_iterator<string>(cin), istream_iterator<string>(), back_inserter(names));
```

From slide *Algorithms copy, copy_n and copy_if (1)* we know that

- The first argument of *copy* is the iterator to the first element is source range. Here it is *istream_iterator<string>(cin)*.
- The second argument is the iterator to the element following the end of range. Here it is *istream_iterator<string>()*.
- The third argument is our case must be the iterator pointing to the end of destination vector. It is constructed by *back_inserter* (see slide *Insert iterators (2)*).

The user must type the sequence of words with a white-space character between them. To stop the input type *ENTER, CTRL+Z, ENTER*.

# Stream iterators (2)

Examples:

```
list<string> names = { "John", "James", "Mary", "Richard" };
ostream_iterator<string> oit(cout, " ");
ranges::copy(names, oit);  // prints all the 4 names with space between them
                                    // see slide Algorithms copy, copy_n and copy_if (4)
fstream f("c:\\temp\\test.txt", fstream::in | fstream::out | fstream::trunc);
if (!f.good())
{
    cout << "Problems with file" << endl;
    return;
}
ranges::copy(names, ostream_iterator<string>(f, " "));  // write into file
vector<string> buf;
f.seekg(ios_base::beg);
copy(istream_iterator<string>(f), istream_iterator<string>(), back_inserter(buf));
                                                                    // read from file
```

# Algorithms *replace* and *replace_if* (1)

```
replace(range. iterator_to_first_element_of_range,
        range. 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(range. iterator_to_first_element_of_range,
           range. 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*.

Examples (see https://en.cppreference.com/w/cpp/algorithm/replace.html):

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

# Algorithms *replace* and *replace_if* (2)

ranges::replace(range, old_value, new_value);

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

ranges::replace_if(range, predicate, new_value);

Examples:

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

# Algorithms *remove* and *remove_if* (1)

```
remove(range. iterator_to_first_element_of_range,
       range. iterator_to_first_element_not_in_range,
       value_to_remove);
```

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

```
remove_if(range. iterator_to_first_element_of_range,
          range. 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*.

Examples (see https://en.cppreference.com/w/cpp/algorithm/ranges/remove):

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
remove(data.begin(), data.end(), 1);
ranges::for_each(data, [](int i) { cout << i << ' '; });
cout << endl; // prints -4 5 -7 -8 12 56 7 12 56 1 7
remove_if(data.begin(), data.end(), [](const int& i)->bool { return abs(i) == 7; });
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
                  // prints -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 range. 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 method *erase*:

Examples:

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

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

# Algorithms *remove* and *remove_if* (3)

```
ranges::remove(range, value_to_remove);
ranges::remove_if(range, predicate);
```

Here the return value is a range, in which the return value of *begin()* points to the first element of garbage and the return value of *end()* points to the element after range.

Examples:

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
auto res1 = ranges::remove(data, 1);
ranges::for_each(data, [] (const int& i) { cout << i << ' '; });
cout << endl; // prints -4 5 -7 -8 12 56 7 12 56 1 7
data.erase(res1.begin(), res1.end());
ranges::for_each(data, [] (const int& i) { cout << i << ' '; });
cout << endl; // prints -4 5 -7 -8 12 56 7
auto res2 = ranges::remove_if(data, [] (const int& i)->bool { return abs(i) == 7; });
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
cout << endl;    // prints -4 5 -8 12 56 56 7
data.erase(res2.begin(), res2.end());
ranges::for_each(data, [] (const int& i) { cout << i << ' '; });
cout << endl; // prints -4 5 -8 12 56
```

# Algorithm *sort* (1)

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

The elements are compared using method *operator*<.

```
sort(range. iterator_to_first_element_of_range,
      range. 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 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*).

Examples (see https://en.cppreference.com/w/cpp/algorithm/sort.html?ref=leetsolve.com):

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
sort(data.begin(), data.end());
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
cout << endl; // prints -8 -7 -4 1 1 1 1 5 7 12 56
sort(data.begin(), data.end(), [](const int i1&, const int& i2)->bool
                                    { return abs(i1) < abs(i2); });
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 1 1 1 -4 5 -7 7 -8 1, 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 *sort* (2)

```
ranges::sort(range);
ranges::sort(range, comparator);
```

The comparator may be a pointer to function, lambda expression or functor. Its arguments must be elements from the 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*).

Examples:

```
vector<int> data = { 1, -4, 5, -7, 1, -8, 1, 12, 56, 1, 7 };
ranges::sort(data);
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
cout << endl; // prints -8 -7 -4 1 1 1 1 5 7 12 56
ranges::sort(data, [](const int& i1, const int& i2)->bool { return abs(i1) < abs(i2); });
ranges::for_each(data, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 1 1 1 -4 5 -7 7 -8 1, 56
```

# Algorithm *unique* (1)

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

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

unique(range. iterator_to_first_element_of_range,
range. 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 range. The body of comparator must check are the arguments equal (return value *true*) or not (return value *false*).

*unique* removes all the duplicates. Only the first of them is kept.

As *remove* and *remove_if*, *unique* does not not change the number of elements in the range. 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* returns the iterator pointig to the first element of garbage. To get rid of garbage use the container method *erase.*

# Algorithm *unique* (2)

Examples (see also https://en.cppreference.com/w/cpp/algorithm/unique.html):

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

```
vector<int> data2 = { 1, -4, 4, 4, 8, 9, 12, 56, 57, 77 };
auto it2 = unique(data2.begin(), data2.end(), [](const int& i1, const int& i2)->bool
                                                { return abs(i1) == abs(i2); });
ranges::for_each(data2, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 -4 8 9 12 56 57 77 57 77
data2.erase(it2, data2.end());
ranges::for_each(data2, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 -4 8 9 12 56 57 77
```

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

# Algorithm *unique* (3)

```
ranges::unique(range);
ranges::unique(range, comparator);
```

Here the return value is a range, in which the return value of *begin()* points to the first element of garbage and the return value of *end()* points to the element after range.

Examples:

```
vector<int> data1 = { 1, -4, -4, -4, 8, 9, 12, 56, 57, 77 };
auto res1 = ranges::unique(data1);
ranges::for_each(data1, [](const int& i) { cout << i << ' '; });
cout << endl; // prints prints 1 -4 8 9 12 56 57 77 57 77
data1.erase(res1.begin(), data1.end());
ranges::for_each(data1, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 -4 8 9 12 56 57 77
vector<int> data2 = { 1, -4, 4, 4, 8, 9, 12, 56, 57, 77 };
auto res2 = ranges::unique(data2, [](const int& i1, const int& i2)->bool
                                    { return abs(i1) == abs(i2); });
ranges::for_each(data2, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 -4 8 9 12 56 57 77 57 77
data2.erase(res2.begin(), data2.end());
ranges::for_each(data2, [](const int& i) { cout << i << ' '; });
cout << endl; // prints 1 -4 8 9 12 56 57 77
```

# Projections

Most of the algorithms from *ranges* library have one more parameter: the *projection*. Its default value is *{ }* , i.e. the empty function. If the projection is implemented, it processes each element from the range before it is handed over to the algorithm.

Example: let us have

```
struct Student
{
   string first_name, last_name;
   Student(string s1, string s2) : first_name(s1), last_name(s2) { }
};
vector<Student> group{ { "John", "Smith"}, { "Elizabeth", "Clerk" }, { "Richard", "Farmer"},
{ "James", "Sailor" } };
```

As method *operator<()* is missing, we cannot sort this vector with method *sort*. But sorting by strings is possible. The solution is to write a projection that returns the last name:

```
ranges::sort(group, less<string>(), [](const Student& s) { return s.last_name;  });
ranges::for_each(group, [](const Student& s)
                              { cout << s.first_name << ' ' << s.last_name << endl; });
// prints in alphabetical order: Elizabeth Clerk, Richard Farmer, James Sailor, John Smith
```

*less<>()* is the standard functor (the others are *less_equal, greater, greater_equal, equal_to, not_equal_to*). The standard functors can be used instead of user's lambdas, they work for classes having the corresponding operator function.