Pointers to functions (1)

Pointer to a variable holds the address of the first byte of memory field on which the variable is located. Pointer to a function holds the address of the byte from which the function code starts.

Declaring a pointer to variable we have to specify the type of data to which it will point. Declaring a pointer to function we have to specify:

- the type of function return value
- the number of parameters
- the types of parameters

Generally, the declaration to a pointer to function is:

return_value_type (*pointer_name)(parameter_list);

Examples:

```
void (*pf)(char *); // pf will point to functions with prototype void XXX(char *)
                    // where XXX is any identifier
double **(*pfn)(int, int); // pfn will point to functions with prototype double **XXX(int, int)
```

Pointers to functions (2)

To assign values to pointer to functions use function names:

```
pointer_to_function = function_name;
```

Example: suppose we have

```
void ToUpper(char *);
void ToLower(char *);
```

then we may write

```
void (*pf)(char *);
pf = ToLower;
```

or

```
pf = ToUpper;
```

Call to a function using pointer:

```
(pointer_to_function)(parameter_list);
```

Example:

```
char Buf[81];
cout << "Type some text" << endl;
gets_s(Buf);
cout << "press '\u\' to convert the text to uppercase or any other key to lowercase" << endl;
pf = _getche() == 'u' ? ToUpper : ToLower;
(pf)(Buf);
```

Pointers to functions (3)

Suppose we have to write a function that is able to sort array containing records of any type. There are several well-known algorithms (insertion sort, bubble sort, quick sort, etc.) but they all need to compare the records. As the values in array may be of any type, we cannot build the comparison directly into the code. The only way to solve the problem is to implement the comparison with pointer to function that can compare two records:

```
void sort(void *pArray, int RecordLength, int nRecords, int (*pCompare)(void *, void *));
```

If the records are of type

```
struct Student
{
  char *pName;
  ……
};
```

the comparing function may be:

```
int CompareStudentNames(void *pStud1, void *pStud2)
{
  return strcmp((char *)((Student *)pStud1)->pName, (char *)((Student *)pStud2)->pName);
}
```

and the call to sorting function may be like:

```
sort(pStudentGroup, sizeof(Student), nGroup, CompareStudentNames);
```

Pointers to functions (4)

Let us have a function for solution of quadratic equation $ax^2 + bx + c$, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
void QuadEq(double a, double b, double (*pf)(), double *px1, double *px2)
{ // coefficient c is the output of any function with no parameters and double as return value
  double d = b * b – 4 * a * (pf)();
  if (d < 0 || !a)
     throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage example:

```
double tester() {   return 6.0;  }
double x1, x2;
try {
  QuadEq(1, 5, tester, &x1, &x2); // roots are -2 and -3
  // QuadEq(1, 5, tester(), &x1, &x2); // error, here tester is a pointer, not function
 }
catch (const exception &e) {
  cout << e.what() << endl;
}
```

Pointers to functions (5)

Another example:

```
void QuadEq(double a, double b, double (*pf)(double, double), double d1, double d2,
                    double *px1, double *px2)
{ // coefficient c is the output of any function with no parameters and double as return value
  double d = b * b – 4 * a * (pf)(d1, d2);   // In function sort from slide Pointers to functions 3
   // the input parameters for pCompare are in pArray. Here we need to specify the input
   // parameters for pf as input parameters of QuadEq
  if (d < 0 || !a)
      throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage:

```
double tester(double d1, double d2) {    return d1 + d2;  }
double x1, x2;
try {
  QuadEq(1, 5, tester, 3, 3, &x1, &x2); // roots are -2 and -3
 }
catch (const exception &e) {
  cout << e.what() << endl;
}
```

Pointers to functions (6)

But if we have

```
class Tester
{
private:
        double Value = 6;
public:
        double GetValue() const { return Value; }
        void SetValue(double d) { Value = d; }
};
```

we cannot call function *QuadEq* from slide *Pointers to functions (4)*:

```
QuadEq(1, 5, Tester::GetValue, &x1, &x2); // error
```

because to use a member function we must also specify the object.

```
 class Tester
{
  public: static double GetValue() const { return 6.0; }
};
```

Now

```
QuadEq(1, 5, Tester::GetValue, &x1, &x2);
```

works because *GetValue()* is now *static* and for static member function it's enough to specify just the class.

Pointers to functions (7)

Pointers to member functions are defined in another way:

return_value_type (class_name::*pointer_name)(parameter_list);

Example:

```
double (Tester::*pf)(); // pf points to functions from class Tester, those functions
                        // have no arguments and they return a double value
```

To assign value to a pointer to member function you must specify also the class:

pointer_name = &class_name::member_function_name

Example:

```
pf = &Tester::GetValue;
```

Calls using the pointers to member functions:

(object_name.*pointer_name)(parameter_list);

(pointer_to_object->*pointer_name)(parameter_list);

Examples:

```
Tester t, *pt = new Tester;
cout << (t.*pf)() << endl;
cout << (pt->*pf)() -> endl;
```

Problem: we have no pointers that can point to functions from any class.

Pointers to functions (8)

Consequently, we cannot use function *QuadEq* from slide *Pointers to functions (4)* with member functions. The proper definition is:

```
void QuadEq(double a, double b, double(Tester::*pf)(), Tester *pt, double *px1, double *px2)
{// Problem: function QuadEx is applicable only for class Tester
 double d = b * b - 4 * a * (pt->*pf)();
 if (d < 0 || !a)
     throw exception ("No solution“);
 *px1 = (-b + sqrt(d)) / 2 * a;
 *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage example:

```
Tester *pt = new Tester;
double x1, x2;
try
{
  QuadEq(1, 5, &Tester::GetValue, pt, &x1, &x2); // roots are -2 and -3
 }
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Lambda expressions (1)

The lambda (the term is from LISP language) is a short nameless function defined in the body of another function.

The simplest lambda definition is:
[ ] (formal_parameter list) { body }

To execute lambda expression immediately add the list of actual parameters:
[ ] (formal_parameter list) { body } (actual_parameter list);

The type of return value is deduced by the expression following the *return* keyword. If there is no *return* statement, the return type is *void*. If necessary, the programmer may specify the return type explicitly:

[ ] (formal_parameter list) -> return_type { body }

Examples:

```
int x = []() { return 6; } (); // define lambda and execute immediately, x gets value 6
double x1 = -2, x2 = -3;
double max = [](double a, double b) { return a <= b ? b : a; } ( x1, x2);
                                    // define lambda and execute immediately, max gets value -2
int arr[] = { 1, 2, 3, 4, 5, 6 };
cout << boolalpha << [](int *p, int n, int m) -> bool
{ int i = 0;  for (; i < n && *(p + i) != m;  i++);  return i != n; } (arr, 6, 10) << endl;
  // prints false because 10 was not found
```

Lambda expressions (2)

To execute a lambda several times declare pointers to lambda expressions:

```
auto pointer_name = lambda_definition;
```

Example:

```
auto pl = [](double a, double b) { return a <= b ? b : a; };
// auto is very useful here because we do not need to guess the type
```

To call a lambda expression by its pointer:

```
pointer_name(actual_parameter_list);
```

Example:

```
double x = pl(x1, x2);
```

Lambda expressions may use variables from the enclosing scope. The brackets at the beginning of lambda are to define the capture block.

Capture block *[=]* means that all the variables may be used by value. Example:

```
double x1 = -2, x2 = -3;
double max = [=]() { return x1 <= x2 ? x2 : x1; } (); // max is -2
```

Capture block *[&]* means that all the variables may be used by reference. Example:

```
double x1 = -2, x2 = -3;
double max = [&]() { return x1 <= x2 ? x2 : x1; } (); // max is -2
```

Lambda expressions (3)

Call by value means that

```
double x1 = -2, x2 = -3;
double max = [=]() { return x1 <= x2 ? x2 : x1; } ();
```

brown and magenta variables have the same names but they are not the same: x1 is the copy of x1. Also, magenta variables are constants:

```
double max = [=]() {x1 = -1; return x1 <= x2 ? x2 : x1; } (); // error, x1 cannot be changed
```

To specify the copies as not constants use keyword *mutual*:

```
[ =] (formal_parameter list) mutual -> return_type { body }
```

Example:

```
double x1 = -2, x2 = -3;
double max = [=]() mutual{x1 = -1; return x1 <= x2 ? x2 : x1; } (); // max is -1
cout << x1 << endl; // still -2 because x1 is just the copy of x1.
```

Call by reference means that lambda can change the values defined in the enclosing block.

Example:

```
double x1 = -2, x2 = -3;
double max = [&]() { x1 = -1; return x1 <= x2 ? x2 : x1; } (); // max is -1
cout << x1 << endl; // x1 is now -1
```

Lambda expressions (4)

Capture blocks *[=]* and *[&]* allow the lambda to use all the variables defined in the enclosing scope. To decide selectively which variables the lambda may capture, specify the capture list.

Examples:

```
double x1 = -2, x2 = -3;
double max = [x1](double b)  {  return x1 <= b ? b : x1; } (x2);
        // lambda can use the copy of x1
max = [&x1](double b)  {  return x1 <= b ? b : x1; } (x2);
        // lambda can use the reference to x1
max = [&x1, x2]() {return x1 <= x2 ? x2 : x1; } ();
        // lambda can use the reference to x1 and the copy of x2
max = [&, x2]() {return x1 <= x2 ? x2 : x1; } ();
        // lambda can use all the variables by reference except x2 that is captured by value.
max = [=, &x2]() {return x1 <= x2 ? x2 : x1; } ();
        // lambda can use all the variables by value except x2 that is captured by reference.
max = [=, &x1, &x2]() {return x1 <= x2 ? x2 : x1; } ();
        // lambda can use all the variables by value except x1 and x2 that are captured by
        // reference.
```

Capture block *[this]* allows lambda to access all the members of the current class.

Lambda expressions (5)

Lambda expressions are often used to replace the pointers to functions. Examples:

```
double x1, x2;
auto pl = [] () -> double { return 6; };
try
{   // QuadEq is defined on slide Pointers to functions (4)
    // especially convenient for testing QuadEx: no additional test functions needed
  QuadEq(1, 5, []() -> double { return 6; }, &x1, &x2); // lambda is defined in call statement
  QuadEq(1, 5, pl, &x1, &x2); // alternative, pointer to lambda is used
}
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Of course, the lambda used in call statement must have the types and number of input parameters as well as the type of return value that correspond to the function prototype. For example, to call function *QuadEx* we can use only lambdas that have no parameters and return a double value.

However, lambda expressions with capture cannot replace the pointers to functions. Example:

```
Tester *pt = new Tester; // defined on slide Pointers to functions (5)
QuadEq(1, 5, [pt]() -> double { return pt->GetValue(); }, &x1, &x2); // error
```

Function wrappers (1)

```
#include <functional> // see also http://www.cplusplus.com/reference/functional/
```

Let us rewrite QuadEq defined on slide *Pointers to functions (4):*

```
void QuadEq(double a, double b, function<double()>pf, double *px1, double *px2)
{  // pointer to function is replaced by function wrapper
  double d = b * b – 4 * a * (pf)();
  if (d < 0 || !a)
      throw exception("No solution");
   *px1 = (-b +sqrt(d)) / 2 * a;
   *px2 = (-b - sqrt(d)) / 2 * a;
}
```

function<double()>pf means that, using standard class templates, we build a wrapper object *pf* for any callable object (function, lambda with or without capture) that has no input parameters and returns a double number. Wrapper object is used (i.e. the corresponding function or lambda is called) as a regular pointer to function.

Generally:

```
function < return_value_type (list_of_input_parameter_types) > wrapper name
```

Function wrappers (2)

```
double tester()
{
   return 6.0;
}

double x1, x2;
Tester *pt = new Tester; // defined on slide Pointers to functions (5)
try
{
  QuadEq(1, 5, tester, &x1, &x2); // normal function out of classes
  QuadEq(1, 5, []() -> double { return 6; }, &x1, &x2); // lambda without capture
  QuadEq(1, 5, [pt]() -> double { return pt->GetValue(); }, &x1, &x2);
                                                          // lambda with capture
}
catch (const exception &e)
{
  cout << e.what() << endl;
}
```

Thus, we have now instruments for transferring functions out of classes as well as member functions. To transfer a function out of classes we may use its name. To transfer member functions we need to create a simple lambda.

Function wrapper (3)

Let us also rewrite QuadEq defined on slide *Pointers to functions (5):*

```
void QuadEq(double a, double b, function<double(double, double)>pf, double d1, double d2,
                    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)(d1, d2);
 if (d < 0 || !a)
     throw exception("No solution");
 *px1 = (-b + sqrt(d)) / 2 * a;
 *px2 = (-b - sqrt(d)) / 2 * a;
}
```

Usage:

```
double x1, x2;
try
{
  QuadEq(1, 5, [](double z1, double z2) { return z1 <= z2 ? z2 : z1; }, 1, 6, &x1, &x2);
}
catch (const exception &e)
{
   cout << e.what() << endl;
}
```

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/

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 *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
```

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.