STL: Библиотека алгоритмов

- В STL для работы с контейнерами имеется *библиотека алгоритмов* с функциями для сортировки, поиска, копирование ...
- Алгоритмы – глобальные функции оперирующие с итераторами

☞ Существуют «специализированные» версии алгоритмов, например `list.sort()`, `map.find()`, которые более эффективны по сравнению с «глобальными»

- Для настройки алгоритмов предусмотрен механизм «подключения» пользовательских функций

☞ Основная задача при работе с алгоритмами состоит в написании пользовательских функций

Алгоритмы: простейшая классификация

- Немодифицирующие алгоритмы: поиск, подсчёт числа элементов ...
- Модифицирующие: удаление, вставка, замена ...
- Алгоритмы сортировки и работы с сортированными последовательностями

Заголовочные файлы

```
#include <algorithm>  // main header file
#include <numeric>    // numeric operations
// auxiliary headers
#include <memory>     // operations on uninitialized memory
#include <execution>  // execution policies, C++17
```

Немодифицирующие алгоритмы

☞ Функции которые не меняют контейнер

`for_each()`	вызывает заданную функцию для каждого элемента
`for_each_n()`	для первых N элементов
`find()`	находит первый заданный элемент
`find_if()`	находит первый элемент удовлетворяющий условию
`count()`	число элементов равных заданному
`count_if()`	число элементов удовлетворяющих условию
`min_element()`	минимальный элемент
`max_element()`	максимальный элемент
`minmax_element()`	возвращает пару *min,max*
`all_of()`	проверяет, истинен ли предикат для всех,
`any_of()`	для любого, или
`none_of()`	ни для одного из элементов

for_each() и for_each_n()

1. `for_each(beg, end, UnaryFunction op)`
   - «функция» `op(elem)` вызывается для всех элементов из `[beg,end)`, а возвращаемое значение `op()` игнорируется
   - `for_each` возвращает копию функционального объекта `op()` после выполнения последнего вызова

2. `for_each_n(beg, size_t N, UnaryFunction op)`, (C++17)
   - то же, что `for_each(beg,beg+N,op)`, но возвращает итератор на `beg+N`

☞ если функция `op(elem)` модифицирует элемент то `for_each()` изменит содержимое контейнера

☞ `for_each()` это один из самых универсальных алгоритмов

for_each() для печати элементов контейнера «в старом стиле»

```
struct print { // класс содержащий operator()
  print(int c=0) : count(c) {}
  void operator()(int x) {cout<<x<<' '; ++count;}
  int count; // counter of operator() calls
};
std::vector<int> V {1,2,3,4,5};
print P = std::for_each(V.cbegin(), V.cend(), print());
cout << endl; // 1 2 3 4 5
cout << P.count << " items printed\n"; // 5 items printed
```

...то же самое с лямбда функцией

```
int count = 0; // counter for lambda
auto lf = std::for_each( cbegin(V), cend(V),
      [&count](const auto& x) {cout<<x<<' '; ++count;});
lf("< " + std::to_string(count) + " items printed\n");
// 1 2 3 4 5 < 5 items printed
```

пример для for_each_n()

```
// print N elements and return iterator to first+N
size_t N = 3;
auto itN = std::for_each_n(cbegin(V), N, // number of elements
      [](const auto& x) {cout<<' '<<x;});
cout << endl; // 1 2 3
cout << "*itN=" << *itN << endl; // *itN=4
```

☞ обратите внимание

- for_each возвращает «функцию»
- for_each_n возвращает итератор
- проверка выхода за пределы контейнера не выполняется
- print() – это вызов конструктора класса print
- старайтесь везде использовать константные итераторы [cbegin,cend)

- **модифицирующий `for_each()`** **«в старом стиле»**

```
struct add_value {
   add_value(int v=1) : the_value(v) {}
   void operator() (int& elem) const {elem+=the_value;}
   int the_value; // value to add to elements
};
std::for_each(V.begin(), V.end(), add_value(2));
std::for_each(V.cbegin(), V.cend(), print());
cout << endl; // 3 4 5 6 7
```

- **. . . то же самое с лямбда функцией**

```
std::for_each( begin(V), end(V), [](int& x) {x+=9;});
std::for_each( cbegin(V), cend(V), [](int x) {cout<<x<<' ';});
cout << endl; // 12 13 14 15 16
```

обратите внимание, что в модифицирующем варианте:

- надо использовать не константные итераторы `[begin,end)`
- тип аргумента модифицирующей функции: `int&`

Минимальный и максимальный элементы

`min_element(), max_element(), minmax_element()`

- параметры функций одинаковы:

```
min_element    (beg, end, [CompFunc op])
max_element    (beg, end, [CompFunc op])
minmax_element (beg, end, [CompFunc op])
```

- функции `min_,max_` возвращают итератор, а `minmax_` пару итераторов, на минимальный/максимальный элемент из `[beg,end)`

- функция сравнения `op(elem1,elem2)` должна возвращать `true` если `elem1 < elem2`; версия без `op()` использует `operator<()`

пример для min_, max_

```
std::vector<int> V {11,2,5,36,4,14};
std::vector<int>::const_iterator it1 =
                          std::min_element(cbegin(V), cend(V));
cout << "min_element= " << *it1 << endl; // min_element= 2
std::vector<int>::const_iterator it2 =
                          std::max_element(cbegin(V), cend(V));
cout << "max_element= " << *it2 << endl; // max_element= 36
```

пример на minmax_ с лямбда функцией для сравнения

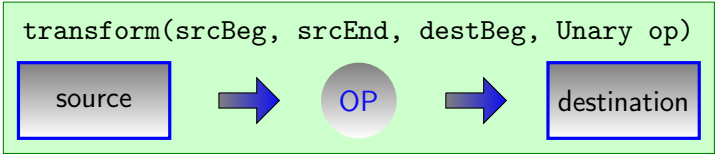
```
auto [itt1,itt2] = std::minmax_element( cbegin(V), cend(V),
      [](int x,int y) {return (x%13 < y%13);} );
cout << "min_max(mod 13)= (" << *itt1 << ", " << *itt2 << ")\n";
// min_max(mod 13)= (14, 11)
```

Модифицирующие алгоритмы

`transform()`	применяет функцию к элементам из заданного диапазона и сохраняет результат в целевом диапазоне
`replace(),..`	заменяет элементы удовлетворяющие условию
`copy()`	копирует, начиная с первого элемента
`copy_n()`	копирует `n` элементов
`copy_if()`	копирует элементы удовлетворяющие условию
`move(),..`	перемещает элементы на новое место
`remove(),..`	удаляет элементы удовлетворяющие условию
`unique()`	удаляет соседние эквивалентный элементы
`generate()`	заменяет элементы результатом операции
`fill()`	заполняет одним элементом
`iota()`	заполняет возрастающей серией

transform()

1. `transform(srcBeg, srcEnd, destBeg, UnaryFunction op)`
   - выполняет функцию `op(elem)` для элементов из `[srcBeg,srcEnd)` и записывает возвращаемое значение в `[destBeg,...)`
   - возвращает позицию последнего элемента в принимающем контейнере
   - `srcBeg` и `destBeg` могут принадлежать одному контейнеру



☞ принимающий контейнер должен быть достаточно большим чтобы вместить входящие элементы

☞ Вспомогательная функция (generic lambda) печати

```
auto prt = [](std::string_view msg, auto&& ss) {
   cout << msg << "{ ";
   for( const auto& s : ss ) {cout << s << " ";}
   cout << "}\n";
};
```

● источник и целевой диапазоны совпадают

```
std::vector V {2,3,5,7};
// change the sign of all elements
std::transform( cbegin(V),cend(V), // source range
      begin(V),                    // destination
      [](int x) {return -x;} );    // operation
prt("after: ",V); // after: { -2 -3 -5 -7 }
```

- **сохранение в другом контейнере**

```
std::list<int> L( size(V) );       // list of required size
std::transform( cbegin(V),cend(V), // source range
      begin(L),                    // destination
      [](int x) {return x+8;} );   // operation
prt("V= ",V); // V= { -2 -3 -5 -7 } без изменений
prt("L= ",L); // L= { 6 5 3 1 }
```

- **итератор вставки в конец контейнера:** `back_inserter(Container)`

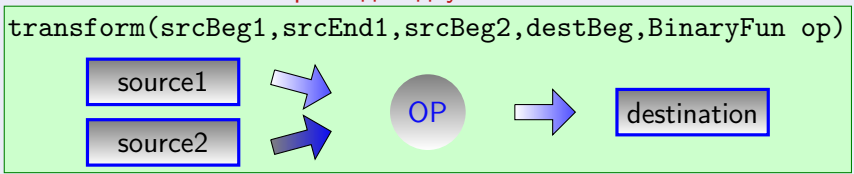
```
std::transform( cbegin(V)+2,cend(V), // source range
      std::back_inserter(L),         // insert into the end of L
      [](int x) {return -x/2;} );    // operation
prt("after back_inserter: L= ",L); // { 6 5 3 1 2 3 }
```

- **итератор вставки общего вида:** `inserter(Container,InsertIter)`

```
std::transform( cbegin(V),cend(V)-2, // source range
      std::inserter(L,begin(L)),     // insert before begin of L
      [](int x) {return 2+x;} );     // operation
prt("after inserter: L= ",L); // { 0 -1 6 5 3 1 2 3 }
```

2 transform(srcBeg1,srcEnd1, srcBeg2, destBeg, BinaryFun op)

☞ версия для двух источников



Пример с двумя источниками

```
std::vector V1 {1, 2, 3, 4};     // <int>
std::vector V2 {5.1, 6.2, 7.3}; // <double>
std::vector<double> Vres;        // destination vector
std::transform( cbegin(V2),cend(V2), // source-1 range
      cbegin(V1),                    // source-2 begin
      std::back_inserter(Vres),      // insert into the end of Vres
      [](auto x1,auto x2) -> double {return x1+x2;} ); // operation
prt("Vres= ",Vres); // Vres= { 6.1 8.2 10.3 }
```

iota() и generate()

- `iota(beg, end, T startValue)` — определена в `<numeric>`

заполняет контейнер приращениями: `startValue,startValue+1,..`

- `generate(beg, end, GeneratorFun op)`

заменяет элементы контейнера на результат вызова `op()` (без аргумента)

```
std::array<long,10> A;
// fill 'A' with sequentially increasing numbers using the iota()
std::iota(begin(A),end(A), 1); // starting with 1
prt("iota: A= ",A); // iota: A= { 1 2 3 4 5 6 7 8 9 10 }

// fill 'A' with the Fibonacci sequence using the generate()
long f1 = 1, f2 = 0;
std::generate( begin(A),end(A), // source range
      [&f1,&f2]() {auto f3=f1+f2; f1=f2; f2=f3; return f2;} ); // op()
prt("A=",A); // A={ 1 1 2 3 5 8 13 21 34 55 }
```

Алгоритмы сортировки

сортировка	
`sort()`	сортировка в порядке возрастания
`partial_sort()`	сортировка первых `n`-элементов
`stable_sort()`	сортировка с сохранением порядка равных элементов
разбиение	
`partition()`	делит элементы на две группы относительно заданного предиката: `(true-els,false-els)`
`stable_partition()`	`partition` с сохранением порядка равных элементов
двоичный поиск в сортированном контейнере	
`binary_search()`	проверка элемента
`lower_bound()`	позиция первого элемента не меньшего заданного
`upper_bound()`	позиция первого элемента большего заданного
`equal_range()`	возвращает пару: `lower_bound(),upper_bound()`

sort() и partial_sort()

1. `sort(beg, end, Compare op)`

   `sort()` сортирует элементы в промежутке `[beg,end)` используя `op(elem1,elem2)` как критерий сортировки

2. `partial_sort(beg, endSort, end, Compare op)`

   в `partial_sort()` сортированным будет лишь промежуток `[beg,endSort)` (`endSort` $\leqslant$ `end`)

Эффективность алгоритмов

- `sort()`: $\sim n \times \log(n)$ вызовов op(), $n = (\text{end} - \text{beg})$
  - ☞ в C++98 неудачная реализация и n^2 в худшем случае
- `partial_sort()`: $\sim n \times \log(m)$ вызовов op(), $m = (\text{endSort} - \text{beg})$
- `stable_sort()`: $\sim n \times \log^2(n)$ вызовов op()

Сортировка «по умолчанию»: в возрастающем порядке

```
std::vector<int> V(5,0);
std::generate(begin(V),end(V), []() {return rand()%100;});

prt("before:  V= ",V); // { 7 49 73 58 30 }
std::sort(begin(V),end(V)); // sort in ascending order (default)
prt("sort(<): V= ",V); // { 7 30 49 58 73 }
```

Сортировка в убывающем порядке

```
std::sort(begin(V),end(V), [](int x,int y) {return x>y;} );
prt("sort(>): V= ",V); // { 73 58 49 30 7 }
```

пример на partial_sort()

```
std::vector W {3,8,3,6,7,9,1,5,10,6};
// select first M largest elements
size_t M = 5; // must be < size(W)
std::partial_sort( begin(W),           // begin of the range
      begin(W)+M,                      // end of sorted range
      end(W),                          // end of full range
      [](int x,int y) return x>y; );   // comparison function
cout << "sort in [beg,beg+"<<M<<"): ";
prt("",W); // sort in [beg,beg+5): { 10 9 8 7 6 3 1 3 5 6 }
```

Обратите внимание

- выражения `begin(W)+5` или `end(W)-2` допустимы лишь для контейнеров с произвольным доступом: `vector<>, array<>`
- приращение итератора общего вида:

```
increments iterator it by 5 elements
auto it = begin(W);
std::advance(it,5); // the same meaning as it+=5
```

partition()

- `partition(beg, end, UnaryPredicate op)`
  - упорядочивает элементы так, что сначала идут элементы для которых `(op(el)==true)`, а затем те, для которых `(op(el)==false)`
  - возвращает итератор на начало второй группы

```
std::vector<int> U(10,0);
std::generate(begin(U),end(U), []() {return rand()%100;});
prt("U= ",U); // U= { 72 44 78 23 9 40 65 92 42 87 }
size_t mid = 50;
// reorders the vector so that (elements < mid) precede all others
auto itm = std::partition(begin(U),end(U),     // range
      [mid](const auto& e) {return e<mid;});   // predicate
prt("after: ",U); // after: { 42 44 40 23 9 78 65 92 72 87 }
prt("true:  ",std::vector(begin(U),itm)); // { 42 44 40 23 9 }
prt("false: ",std::vector(itm, end(U))); // { 78 65 92 72 87 }
```

Параллельные алгоритмы в C++17

Политика выполнения алгоритмов (execution policy)

☞ основная идея — с помощью дополнительного параметра указать, как какой либо алгоритм должен выполняться: последовательно, параллельно или векторизованно

```
std::algorithm_name(policy, /* normal args... */);
```

- параметр `policy` может принимать значения:
  - `std::execution::seq` – последовательное выполнение: sequenced_policy
  - `std::execution::par` – параллельное выполнение: parallel_policy
  - `std::execution::par_unseq` – параллельное и векторизованное: parallel_unsequenced_policy
  - `std::execution::unseq` – векторизованное: unsequenced_policy C++20
- не все алгоритмы имеют `policy` параметр и «поведение» некоторых алгоритмов немного отличается от «стандартной версии»

Последовательное, параллельное и векторизованное выполнение

```
// pseudocode example
auto mul = [](float x,float y) {return x*y;};
std::transform( policy,              // execution policy
                cbegin(x), cend(x),  // 1st vector
                cbegin(y),           // 2nd vector
                begin(z),            // output vector
                mul);                // operation
// sequential execution:
load x[i]; load y[i]; mul(x[i],y[i]); store into z[i]

// parallel execution: multiple threads
new_thread(load x[i]; load y[i]; mul(x[i],y[i]); store into z[i])

// vectorized execution: SIMD (Single-Instruction, Multiple-Data)
load x[i...i+3]; load y[i...i+3];
mul(four elements at once); store into z[i..i+3]
```

пример с сортировкой

```
void TimeSort(std::vector<double> v, size_t ipolicy) {
   const auto start = std::chrono::steady_clock::now();
   std::string spolicy("Error");
   switch (ipolicy) {
      case 0: spolicy = "whitout policy";
         std::sort(begin(v), end(v)); break;
      case 1: spolicy = "sequenced policy";
         std::sort(std::execution::seq, begin(v), end(v)); break;
      case 2: spolicy = "parallel policy";
         std::sort(std::execution::par, begin(v), end(v)); break;
      case 3: spolicy = "parallel unsequenced policy";
         std::sort(std::execution::par_unseq, begin(v), end(v)); break;
   }
   const auto finish = std::chrono::steady_clock::now();
   auto tm = std::chrono::duration_cast<std::chrono::milliseconds>
      (finish - start).count();
   std::cout << spolicy << ": " << tm << "ms\n";
};
```

продолжение примера с сортировкой

```
std::vector<double> v(10'000'000);
std::generate(begin(v),end(v), []() {return double(rand())/RAND_MAX;});
for( size_t ipolicy : {0,1,2,3} ) {TimeSort(v,ipolicy);}
```

output: gcc(14.2.0) -std=c++20 -O -ltbb (4-core i686)

```
whitout policy: 845ms
sequenced policy: 844ms
parallel policy: 276ms
parallel unsequenced policy: 277ms
unsequenced policy: 843ms
```

Замечания

- Поддержка параллельных алгоритмов есть не у всех компиляторов
- Если выполнение невозможно в параллельном виде, все политики могут вернуться к последовательному выполнению
- При использовании параллельных алгоритмов на программисте лежит ответственность избегать data races и deadlocks

☞ data-raice: в примере с сортировкой sort заменен на for_each

```
// calculating the sum of the elements of V (uniformly random on [0,1))
double s {0.}; // ERROR: data raice for parallel execution
// std::atomic<double> s {0.};
auto acc = [&s](const auto& v) { s += v;};
...
   case 1: spolicy = "sequenced policy";
      std::for_each(std::execution::seq,cbegin(V),cend(V),acc);
      break;
   case 2: spolicy = "parallel policy";
      std::for_each(std::execution::par,cbegin(V),cend(V),acc);
      break;
...
std::cout<<spolicy<<": sum="<<sum/size(V)<<" T="<<tm<<"ms\n";
```

double s; // ERROR: data raice

```
sequenced policy: sum=0.500034 T=22ms
parallel policy:  sum=0.169102 T=8ms
par-unseq policy: sum=0.0804907 T=8ms
```

std::atomic<double> s; // slow

```
sequenced policy: sum=0.500034 T=99ms
parallel policy:  sum=0.500034 T=739ms
par-unseq policy: sum=0.500034 T=739ms
```

STL: Библиотека файловой системы

The Filesystem Library — since C++17

- Класс описывающий путь к файлу (path object)
  - манипуляции с путями, информация о пути
  - информация о типе объекта связанного с путем: файл, директория, ссылка ...
  - манипуляции с файлами: создание, копирование, перемещение
- Итераторы по директории, записи `directory_entry`
- А так же: права доступа, временные файлы, специальные файлы ...

Пространство имен `std::filesystem`

```
#include <filesystem>            // header
namespace fs = std::filesystem; // shortcut for std::filesystem
```

пример операций с путем: fs::path

```
fs::path p1("/Users");
cout << p1 << endl;   // "/Users"
p1.append("nefedov"); // adds a path with a directory separator
p1 /= "test";         // synonym for append()
cout << p1 << endl;   // "/Users/nefedov/test"
p1.concat("/new_");   // only adds new characters to the end
p1 += "cpp/a.out";    // synonym for concat()
cout << p1 << endl;   // "/Users/nefedov/test/new_cpp/a.out"

for (const auto& part : p1) { // path iteration
   cout << part << ", ";
}
cout << endl; // "/", "Users", "nefedov", "test", "new_cpp", "a.out"

PathInfo(p1); // следующий слайд
```

пример: некоторые «информационные» функции для fs::path

```
void PathInfo(const fs::path& TestPath) {
   cout << "Path info for: " << TestPath << endl;
   cout << "canonical path " << fs::canonical(TestPath) << endl;
   cout << "exists=        " << fs::exists(TestPath) << endl;
   cout << "root_name=     " << TestPath.root_name() << endl;
   cout << "root_path=     " << TestPath.root_path() << endl;
   cout << "relative_path= " << TestPath.relative_path() << endl;
   cout << "parent_path=   " << TestPath.parent_path() << endl;
   if ( fs::is_regular_file(TestPath) ) {
      cout << "regular file=  " << TestPath.filename() << endl;
      cout << "   stem=       " << TestPath.stem() << endl;
      cout << "   extension=  " << TestPath.extension() << endl;
      cout << "   size(byte)= " << fs::file_size(TestPath) << endl;
      cout << "   time=       " << FileTime(TestPath) << endl;
   } // else if ( fs::is_directory(TestPath) ) { ... skip
}
```

output of PathInfo(p1)

```
Path info for: "/Users/nefedov/test/new_cpp/a.out"
canonical path "/Volumes/CaseSensitive/test/cpp/MyLectures/new_cpp/a.out"
exists=        1
root_name=     ""
root_path=     "/"
relative_path= "Users/nefedov/test/new_cpp/a.out"
parent_path=   "/Users/nefedov/test/new_cpp"
regular file=  "a.out"
   stem=       "a"
   extension=  ".out"
   size(byte)= 41272
   time=       Wed Feb 12 21:08:36 2025
```

Обратите внимание

- функции члены, такие как `TestPath.filename()`: быстрые, дешевые
- автономные (free-standing) функции, как `fs::exists(TestPath)`: затратные, обычно обращаются к реальной файловой системе
- функция `FileTime()` для красивой печати времени, следующий слайд

время последней модификации файла

```
std::string FileTime( const fs::path& FileName ) {
   std::string ret; // empty string for result value
   if( fs::exists(FileName) && fs::is_regular_file(FileName) ) {
      auto ftime = fs::last_write_time(FileName);
#if __cplusplus >= 202002L
      using namespace std::chrono_literals;
      ftime += 3h; // timezone UTC -> MSK
      ret = std::format(":%c", ftime); // C++20
#else   // C++17
      namespace cr = std::chrono;
      auto nowF = fs::file_time_type::clock::now();
      auto dsec = cr::duration_cast<cr::seconds>(nowF-ftime).count();
      const auto now = cr::system_clock::now();
      std::time_t tm = cr::system_clock::to_time_t(now) - dsec;
      ret = std::string(std::ctime(&tm)); // C-time
#endif
   }
   return ret;
}
```

...продолжение PathInfo(): информация о директории

```
} else if ( fs::is_directory(TestPath) ) {
   cout << "directory=     " << TestPath.filename()
      << " contains the following items:\n";
   size_t Nf = 0;
   // iterate over directory items but not subdirectories
   for ( auto entry : fs::directory_iterator(TestPath) ) {
      std::string fname = entry.path().filename(); // get filename
      if ( fs::is_directory(entry.status()) ) fname += "/";
      cout << "   " << fname << endl;
      Nf++;
   }
   cout << "== " << Nf << " items in total ==\n";
}
```

тест для директории

```
fs::path p2("/Users/nefedov/test/new_cpp");
PathInfo(p2);
```

output of PathInfo(p2)

```
Path info for: "/Users/nefedov/test/new_cpp"
canonical path "/Volumes/CaseSensitive/test/cpp/MyLectures/new_cpp"
exists=        1
root_name=     ""
root_path=     "/"
relative_path= "Users/nefedov/test/new_cpp"
parent_path=   "/Users/nefedov/test"
directory=     "new_cpp" contains the following items:
   BitManip.cpp
   файл для теста.txt
   a.out
   dir_test/
== 4 items in total ==
```

Дополнительные слайды

- `nth_element(beg, end, IterNth, Compare op)`
  - ставит на нужное «сортированное» место E_n элемент и упорядочивает элементы E_k таким образом, что `op`(E_j, E_i)`==false` если `i<n и j>n`
  - `op(E1,E2)`: функция сравнения, как в `sort()`
  - позиция `n`-th элемента задается итератором `IterNth`

- `select first M largest elements >compare with partial_sort()`

```
std::vector W {3,8,3,6,7,9,1,5,10,6};
size_t M = 5; // must be < size(W)
std::nth_element( begin(W),            // begin of the range
      begin(W)+M,                      // iterator for M-th
      end(W),                          // end of full range
      [](int x,int y) {return x>y;} ); // comparison function
cout << "put "<<M<<"-th in right place: ";
prt("",W); // put 5-th in right place: { 9 8 10 7 6 6 5 3 3 1 }
```

- `binary_search(beg, end, Value, Compare op)`
  - проверяет наличие `Value` в отсортированном контейнере
  - `op` должен совпадать с алгоритмом который применялся для сортировки

```
auto lgt = [](int x,int y) {return x>y;}; // comparison function
std::sort(begin(W), end(W), lgt); // decreasing sort
prt("W= ",W); // W= { 10 9 8 7 6 6 5 3 3 1 }
// binary search must use exactly the same function as sorting
if( std::binary_search(cbegin(W),cend(W),3, lgt) ) {
   cout << "W contains 3" << endl;
} else {
   cout << "W does not contain 3" << endl;
} // W contains 3
```

equal_range()

- equal_range(beg, end, Value, Compare op)
  - возвращает диапазон итераторов begV,endV, таких, что вставка Value перед ними не нарушает сортировку
  - op должен совпадать с алгоритмом который применялся для сортировки

```
auto lgt = [](int x,int y) {return x>y;}; // Compare function
std::sort(begin(W), end(W), lgt);
prt("W= ",W); // W= { 10 9 8 7 6 6 5 3 3 1 }
for( size_t i : {4,3} ) {
   auto [pL,pU] = std::equal_range(cbegin(W),cend(W),i,lgt);
   cout << i << " can be inserted in interval ["
        << std::distance(cbegin(W),pL) << ","
        << std::distance(cbegin(W),pU) << "]\n";
}
// 4 can be inserted in interval [7,7]
// 3 can be inserted in interval [7,9]
```

Печать дерева файлов

```
void DirTree(const fs::path& pathToDir, int level=0) {
   if ( fs::exists(pathToDir) && fs::is_directory(pathToDir) ) {
      if( level == 0 ) { cout << "> " << pathToDir << "\n"; }
      auto shift = std::string(2+level*5,' ');
      for (const auto& entry : fs::directory_iterator(pathToDir)) {
         auto filename = entry.path().filename();
         if ( fs::is_directory(entry.status()) ) {
            cout << shift << "[+] " << filename << "\n";
            DirTree(entry, level + 1);
         } else if ( fs::is_regular_file(entry.status()) ) {
            cout << shift << filename << "\n";
         } else {
            cout << shift << " [?]" << filename << "\n";
         }
      }
   }
}
```

вызов DirTree() для рабочей директории

```
const fs::path wd { fs::current_path() }; // working directory
DirTree(wd);
```

output DirTree(wd)

```
> "/Volumes/CaseSensitive/test/cpp/MyLectures/new_cpp"
  "BitManip.cpp"
  "файл для теста.txt"
  "a.out"
  [+] "dir_test"
       [+] "dir2"
            "file2"
            [+] "dir3"
                 "file3"
       "file1"
```