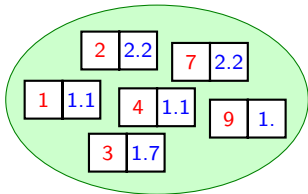


# STL: map<>

👉 Ассоциативный контейнер `std::map<Tkey, Tvalue>`:  
контейнер для хранения пар (ключ, величина) (`key, value`)

`std::map<int, double>`

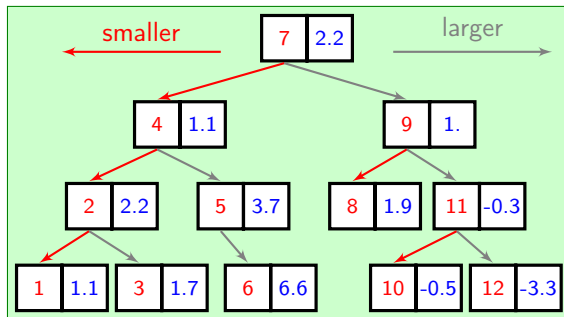


ассоциативный массив: `m[key] = value`

- ключ играет роль индекса массива
- величина — то что хранится в элементе с этим индексом

- ✓ В **map** не может быть двух элементов с одинаковыми ключами
- ✓ Элементы **map** автоматически сортируются по ключу

# STL map<>: сбалансированное бинарное дерево



- ✓ сортировка обеспечивает быстрое нахождение элемента по ключу используя бинарный поиск
- ✓ для ключа нужна операция сравнения, например `оператор<()`
- ☞ значение ключа изменить нельзя, сортировка нарушится

# STL map<>: основное

```
#include <map>           // заголовочный файл
```

## Параметры шаблона для map<>

map<Tkey, Tval>            ключи сортируются с `Tkey::operator<()`

map<Tkey, Tval, TOp>    TOp – тип (type) функции сортировки `Op()`

### конструкторы

map<> m                    пустой map

map<> m1(m2)              копирующий конструктор

map<> m {{k1,v1},{k2,v2},...}    инициализация списком

map<> m(Op)                пустой map с функцией сортировки Op

### размер

m.size()                    количество элементов map

m.empty()                   true для пустого map

## Пример `std::map<int,double>`

```
using Mmap = std::map<int,double>; // shorten the writing

Mmap mm {{5,0.2}, {7,0.6}, {1,0.75}};
mm[2] = 1.0;           // insert one more element

// std::map<int,double>::const_iterator it = mm.cbegin();
for( auto it = cbegin(mm); it != cend(mm); ++it ) {
    cout << "mm[" << it->first << "]= " << it->second << ", ";
}
cout << endl; // mm[1]=0.75, mm[2]=1, mm[5]=0.2, mm[7]=0.6,
```

## Обратите внимание

📖 порядок вывода отсортирован по ключу

`it->first` – доступ к ключу

`it->second` – доступ к значению

# STL map<>: operator[]

• `map::operator[const Key& key]` — доступ по ключу

- ✓ если ключ имеется, возвращается ссылка на ассоциированное с ключом значение
- ✓ если ключа нет, создается новая пара с этим ключом, для значение вызывается конструктор по умолчанию и возвращается ссылка на него

Так делать не надо, иначе появятся новые элементы!

```
auto mt = mm; // make a copy of 'mm' for this test
for( int i = 1; i <= 7; i++ ) {
    cout << "mt[" << i << "]= " << mt[i] << ", ";
}
cout << endl; // mt[1]=0.75, mt[2]=1, mt[3]=0, mt[4]=0,
               // mt[5]=0.2, mt[6]=0, mt[7]=0.6,
```

• функция `at(key)`: `[key]` с проверкой существования `key`

- ✓ возвращает ссылку на значение элемента с ключом `key`
- ✓ если ключ отсутствует, возбуждает исключение `std::out_of_range`

```
try {  
    cout << "mm[100]=" << mm.at(100) << endl;  
} catch(const std::out_of_range& e) {  
    cout << "Exception: " << e.what() << endl;  
} // mm[100]=Exception: map::at: key not found  
  
for( int i = 1; i <= 7; i++ ) {  
    try { cout << "mm[" << i << "]= " << mm.at(i) << ", ";  
    } catch (std::out_of_range) { continue; }  
}  
cout << endl; // mm[1]=0.75, mm[2]=1, mm[3]=mm[4]=mm[5]=0.2,  
              // mm[6]=mm[7]=0.6,
```

## ● цикл for-range для map

```
template<typename Map>
inline void PrtMap(std::string_view msg, Map&& M) {
    cout << msg << "{ ";
    for( const auto& p : M ) {
        cout << "{" << p.first << ", " << p.second << "} ";
    }
    cout << "}\n";
};

PrtMap("mm=",mm); // mm={ {1,0.75} {2,1} {5,0.2} {7,0.6} }
PrtMap("mt=",mt); // «испорченный» map
// mt={ {1,0.75} {2,1} {3,0} {4,0} {5,0.2} {6,0} {7,0.6} }
```

👉 здесь p – объект специального класса для пары:

- p.first – ключ, всегда константа
- p.second – значение

# STL: класс `pair<>`

## Класс-шаблон `std::pair<T1,T2>`

- Класс `std::pair<T1,T2>` содержит два элемента типа `T1` и `T2` соответственно
- В `map` используется `std::pair<Key,Value>` для хранения пары объектов как единого целого

## • члены класса `pair<T1,T2>`

<code>pair(const T1&amp;, const T2&amp;)</code>	конструктор
<code>first</code>	первый объект пары
<code>second</code>	второй объект пары



## • функции в классе pair

- ❶ Глобальная функция `make_pair()` для создания пары, эквивалентна вызову конструктора `pair<T1,T2>(x,y)` и существует «для удобства»

```
template <class T1, class T2>  
pair<T1,T2> make_pair(const T1& x, const T2& y);
```

- ❷ Оператор сравнения:

```
template <class T1, class T2>  
bool operator<(const pair<T1,T2>& x,const pair<T1,T2>& y);
```

- ✓ возвращает `true` если `x.first < y.first` и `false` если `y.first < x.first`
- ✓ если первые элементы «равны», сравниваются вторые элементы

## 👉 Обратите внимание

Для сравнения между элементами пары используется только `operator<()`

# STL map<>: функция сравнения ключей

Функцию для сортировки ключей можно определить:

- 1 с помощью `operator<()` для типа `Key`:

```
map<Key,Elem> Map1;
```

- 2 с помощью объекта-функции: создается класс `Compare` в котором определяется оператор `bool operator()(TKey k1,TKey k2)`

```
map<Key,Elem,Compare> Map2;
```

☞ в шаблоне указывается имя класса

- 3 с помощью «внешней функции» `bool FunCmp(TKey k1,TKey k2)`

```
map<Key,Elem,TypeOfFunCmp> Map3(FunCmp);
```

☞ в шаблоне указывается тип функции сравнения, а сама функция передается как аргумент конструктора

## 1 объект-функция для `std::map<const char*, int>`

- задаем класс `ClsCmp` с оператором() для сравнение двух строк:

```
struct ClsCmp {  
    bool operator()(const char* s1, const char* s2) const  
        {return strcmp(s1, s2) < 0;}  
};
```

- в шаблоне `std::map<>` указываем имя класса:

```
std::map<const char*, int, ClsCmp> Months; // имя месяца:число дней
```

## 👉 объекты-функции (function object or functor):

объекты имеющие свойства функций, так как для них определён `operator()`

## 2 внешняя функция для `std::map<const char*, int>`

```
bool FunCmp(const char* s1, const char* s2) {  
    return strcmp(s1, s2) < 0;  
}  
  
using FUNCPTR = bool (*)(const char*, const char*); // type of function  
// typedef bool (*FUNCPTR)(const char*, const char*); // old  
std::map<const char*, int, FUNCPTR> Months(FunCmp);
```

## 3 лямбда функция для `std::map<const char*, int>`

```
auto LymCmp = [](const char* s1, const char* s2) -> bool {  
    return strcmp(s1, s2) < 0;  
};  
  
std::map<const char*, int, decltype(LymCmp)> Months(LymCmp);
```

👉 обратите внимание на оператор `decltype`

✓ для внешней функции его тоже можно использовать

## Пример:

```
std::map<const char*, int, decltype(LymCmp)> Months(LymCmp);  
// std::map<const char*, int, FUNCPTR> Months(FunCmp);  
// std::map<const char*, int, ClsCmp> Months;  
Months = {  
    {"January", 31},  
    {"February", 28},  
    {"March", 31},  
    {"April", 30}  
};  
for( const auto& m : Months ) {  
    cout << m.first << " has " << m.second << " days\n";  
}  
// April has 30 days  
// February has 28 days  
// January has 31 days  
// March has 31 days
```

# STL map<>: вставка и удаление элементов

## методы для вставки и удаления

<code>insert(elem)</code>	вставляет элемент и возвращает пару: позицию нового элемента и код выполнения
<code>insert(beg,end)</code>	вставляет элементы из диапазона <code>[beg,end)</code>
<code>insert_or_assign()</code>	вставляет элемент или меняет его
<code>emplace()</code>	вставляет элемент «по месту»
<code>erase(elem)</code>	удаляет все элементы со значением <code>elem</code> и возвращает число удалённых элементов
<code>erase(pos)</code>	удаляет элемент на позиции <code>pos</code>
<code>erase(beg,end)</code>	удаляет элементы в диапазоне <code>[beg,end)</code>
<code>clear()</code>	удаляет все элементы

## ❶ вставка с помощью `operator[]`

- используя `[]` невозможно понять вставили элемент или изменили  
👉 с функцией `at()` можно выделить случай «только изменение»
- вставка с `[]` невозможна если у `Tval` нет конструктора по умолчанию

---

```
using Mmap = std::map<int,double>; // shorten the writing
Mmap id = {{1,1.}, {2,1.}};
id[3] = 3.; // new pair {3,3}
id[2] = 2.; // no inserting, change value
PrtMap("id=",id); // id={ {1,1} {2,2} {3,3} }
```

---

## ❷ ВСТАВКА С ПОМОЩЬЮ `map::insert()`

- вставлять надо правильно подготовленную пару
- пара не вставится если ключ уже имеется в `std::map`
  - 👉 `insert()` возвращает пару: итератор на элемент с запрошенным ключом и успешность вставки (`bool`)

---

```
std::pair<Mmap::iterator,bool> mret =  
    id.insert(std::pair<int,double>(2,4.));  
cout << mret.second << endl; // 0 - fail to insert  
  
auto [it,ok] = id.insert(make_pair(4,4.)); // C++17  
cout << ok << endl; // 1 - success  
  
// map::value_type() is typedef for std::pair<Tkey,Tval>  
id.insert(Mmap::value_type(5,5.)); // ignore the return value  
  
print_Mmap("",id); // { {1,1} {2,2} {3,3} {4,4} {5,5} }
```



### 3 вставка с помощью `map::emplace()`

C++11...C++17

- ✓ работает как `map::insert()`, но пару создает «по месту» избегая ненужного копирования или перемещения
- ✓ `map::try_emplace()` гарантирует, что сначала проверяет ключ и если он есть, то пара даже не создается

---

```
auto [it6,ok6] = id.emplace(6,6.); // creating pair in place  
cout << ok6 << endl; // 1 - success
```

```
id.try_emplace(6,7.); // pair guaranteed not created
```

```
print_Mmap("",id); // { {1,1} {2,2} {3,3} {4,4} {5,5} {6,6} }
```

---

#### 4 вставка с помощью `map::insert_or_assign()`

C++17

- ✓ работает как `map::insert()`, но если ключ уже имеется – меняет значение в соответствующей паре

```
if( auto [it,isins] = id.insert_or_assign(7,6.); isins ) {  
    cout << it->first << " entry was inserted\n";  
} else {  
    cout << it->first << " entry was updated\n";  
} // 7 entry was inserted  
if( auto [it,isins] = id.insert_or_assign(7,7.); isins ) {  
    cout << it->first << " entry was inserted\n";  
} else {  
    cout << it->first << " entry was updated\n";  
} // 7 entry was updated
```

#### 👉 Обратите внимание

В C++17 появилась возможность инициализации внутри `if` и `switch`

## ● класс для значений не имеет конструктора по умолчанию

```
struct Nc {  
    Nc(int x) : n(x) {}  
    int n;  
};  
  
std::map<std::string,Nc> Nmap {"a",1},{"b",2};  
PrtMap("Nmap=",Nmap); // Nmap={ {a,1} {b,2} }  
  
Nmap["a"] = Nc(0); // compile error  
Nmap["c"] = Nc(3); // compile error  
  
nid.insert(Nmap::value_type("c",3)); // OK!  
PrtMap("",nid); // { {a,1} {b,2} {c,3} }  
  
if(auto [it,ok]=nid.try_emplace("c",4);!ok) {cout<<"false\n";} // false  
nid.try_emplace("d",4); // OK  
PrtMap("",nid); // { {a,1} {b,2} {c,3} {d,4} }  
  
nid.insert_or_assign("c",4); // "c" entry is updated  
PrtMap("",nid); // { {a,1} {b,2} {c,4} {d,4} }
```

## ● удаление с помощью `map::erase()`

👉 `map::erase(it)` возвращает итератор на следующий элемент

\_\_\_\_\_ удаляем все элементы с нулевым значением \_\_\_\_\_

```
PrtMap("mt= ",mt); // «испорченный» map
// mt= { {1,0.75} {2,1} {3,0} {4,0} {5,0.2} {6,0} {7,0.6} }
for( auto it = begin(mt); it != end(mt); ) {
    if( it->second == 0 ) {
        it = mt.erase(it); // C++11
        // mt.erase(it++); // C++98
    } else {
        ++it;
    }
}
PrtMap("",mt); // { {1,0.75} {2,1} {5,0.2} {7,0.6} }
```

👉 старое поведение в C++98: `map::erase(it)` ничего не возвращает

Надо использовать `map::erase(it++)`: увеличение итератора происходит до вызова функции, а удаляется старое значение `it`

# STL map<>: поиск по ключу

## методы для поиска по ключу

<code>m.find(k)</code>	позиция элемента с ключом <code>k</code> или <code>m.end()</code>
<code>m.lower_bound(k)</code>	первая позиция в которую элемент с ключом <code>k</code> может быть вставлен (первый элемент с <code>key &gt;= k</code> )
<code>m.upper_bound(k)</code>	последняя позиция в которую элемент с ключом <code>k</code> может быть вставлен (первый элемент с <code>key &gt; k</code> )
<code>m.equal_range(k)</code>	возвращает пару <code>lower_bound(k), upper_bound(k)</code>
<code>m.count(k)</code>	число элементов с ключом <code>k</code>
<code>m.contains(k)</code>	проверяет, есть ли элемент с ключом <code>k</code>

C++20

### Пример: цикл с проверкой существования ключа

```
Mmap mf = {{1,1}, {5,5}, {7,7}}; // just for test
for( int i = 1; i <= 7; i++ ) {
    if( auto it = mf.find(i); // initialization in if (C++17)
        it != end(mf) ) {    // checking that the key is found
        cout << "mf[" << it->first << "]" = " << it->second << ", ";
    }
}
cout << endl; // mf[1]=1, mf[5]=5, mf[7]=7,
```

### 👉 Обратите внимание

В C++17 появилась возможность инициализации внутри **if** и **switch**

# STL: set<>

## Ассоциативный контейнер set

📖 Неформальное определение: то же что и `map`, но хранятся только ключи, почти все функции из `map` есть и в `set` (нет `operator[]` и `at()`)

**Пример:** `#include <set> // header for set`

```
std::set<int> s1 = {1,2,3,4,5}; // инициализация списком
std::set<int> s2 = {5,3,7,9,1};
auto prt_set = [](std::string_view msg, auto&& ss) { // удобная печать
    cout << msg << "{ ";
    for( const auto& s : ss ) { cout << s << " "; }
    cout << "}\n";
};
prt_set("s1= ",s1); // s1= { 1 2 3 4 5 }
prt_set("s2= ",s2); // s2= { 1 3 5 7 9 } - отсортирован
```

### ... продолжение

```
// insert in set
auto ret = s2.insert(7); // try to insert existing key
if( !ret.second ) {cout << "false\n";} // false
prt_set("s2= ",s2); // s2= { 1 3 5 7 9 }

// intersection of sets: #include <algorithm>
std::set<int> sec12; // new set for result
std::set_intersection( cbegin(s1), cend(s1), // 1-st set
                      cbegin(s2), cend(s2), // 2-nd set
                      std::inserter(sec12, begin(sec12)) ); // result
prt_set("s1 A s2=",sec12); // s1 A s2={ 1 3 5 }

// remove in the range [first,last)
s1.erase( begin(s1), s1.find(3) );
prt_set("",s1); // { 3 4 5 }
```

☞ В библиотеке алгоритмов имеются функции для вычисления объединения, пересечения, разности и симметричной разности множеств



# STL: multiset<> и multimap<>

заголовочные файлы <set> и <map> соответственно

- Неформальное определение: то же, что `set` и `map`, но могут существовать элементы с одинаковыми ключами
- Функции такие же как у `set` и `map`, но с поправками, что ключ не единственный

## Пример для multiset<>

```
// export elements from the 's2' container
std::multiset<int> ms( begin(s2), end(s2) );
prt_set("",ms); // { 1 3 5 7 9 } the same print function

ms.insert(3); // a key can always be inserted into multiset
prt_set("",ms); // { 1 3 3 5 7 9 }
ms.insert( begin(s1), end(s1) ); // insert from 's1' set
prt_set("",ms); // { 1 3 3 3 4 5 5 7 9 }
```

### ... продолжение

```
pri_set("",ms); // { 1 3 3 3 4 5 5 7 9 }
int tag = 3; // element to tests

// Number of keys equal to 'tag'
cout << ms.count(tag) << endl; // 3

// Find All Occurrences of 'tag' in multiset
auto it = ms.equal_range(tag); // return iterators to the range
                                // that contains 'tag'
for( auto i = it.first; i != it.second; ++i ) {
    cout << distance(begin(ms),i) << " ";
}
cout << endl; // 1 2 3
```

☞ функция `equal_range(key)` возвращает пару итераторов каждый из которых можно получить с помощью `lower_bound(key)` и `upper_bound(key)` функций

# Неупорядоченные ассоциативные контейнеры

## Unordered associative containers:

- `std::unordered_map`
- `std::unordered_multimap`
- `std::unordered_set`
- `std::unordered_multiset`

- ✓ Технически такие контейнеры реализуют хешированные структуры данных, в которых можно быстро осуществлять поиск со сложностью  $O(1)$  в среднем
- ✓ Функционально аналогичны ассоциативным контейнерам с сортировкой

```
#include <unordered_map> // header

std::unordered_map<std::string,int> mths;

mths = { {"january",31}, {"february",28}, {"march",31},
         {"april",30},   {"may",31},      {"june",30},
         {"july",31},    {"august",31},    {"september",30},
         {"october",31}, {"november",30}, {"december",31} };

// Порядок хранения зависит от «хеш»-функции...
for(const auto& m: mths) {
    cout << m.first << " has " << m.second << " days\n";
}

// october has 31 days
// september has 30 days
// august has 31 days

// ... но поиск работает как ожидается
if( auto it = mths.find("april"); it != mths.end() ) {
    cout << it->first << " has " << it->second << " days\n";
} // april has 30 days
```

# Перемещение узлов в ассоциативных контейнерах

## Splicing for maps and sets (C++17)

- 👉 Прямое перемещение узлов (nodes) из одного контейнера в другой без накладных расходов на копирование и выделение памяти

### Примера с map<> ...

```
std::map<int,std::string> s { {1,"a"},{2,"b"},{3,"c"} };
std::map<int,std::string> t { {11,"x"},{12,"y"},{13,"z"} };
PrtMap("s=",s); // s={ {1,a} {2,b} {3,c} }
PrtMap("t=",t); // t={ {11,x} {12,y} {13,z} }

// extract from 's' and insert into 't'
t.insert(s.extract(2)); // by key
PrtMap("",s); // { {1,a} {3,c} }
PrtMap("",t); // { {2,b} {11,x} {12,y} {13,z} }
```

### ... пример на перемещение узлов

```
// change key without copying
auto node = t.extract(13); // extract node
node.key() = 5;           // change key
t.insert(std::move(node)); // move back
PrtMap("",t); // { {2,b} {5,z} {11,x} {12,y} }

// moving nodes from source to target without copying
t.merge(s);
PrtMap("",s); // { }
PrtMap("",t); // { {1,a} {2,b} {3,c} {5,z} {11,x} {12,y} }
```

## STL: tuple<> (C++11)

```
std::tuple<T1,T2,...,Tn>
```

- Обобщает `std::pair<>` на случай трех или более переменных
- Используются вариативные шаблоны (*variadic template*)

```
#include <tuple> // header for tuple

// constructors
std::tuple<int,bool,std::string> tu(1,true,"tu");
// type deduction in C++17:
std::tuple tu2 {2,1.1,2.2}; // <int,double,double>

// get access to elements of tuple: get<idx>(tuple)
cout << std::boolalpha << " tu= "
    << std::get<0>(tu) << " "
    << std::get<1>(tu) << " "
    << std::get<2>(tu) << endl; // tu= 1 true tu
```

### ... продолжение

```
// get<idx>(tuple) return reference
std::get<2>(tu) = "TUPLE"; // assign value to 2-nd element
cout << std::get<std::string>(tu) << endl; // get by type: TUPLE

// in std::get<idx>() idx is the compile-time constant
for(size_t i=0; i < 3; ++i) cout << std::get<i>(tu);
                        // ERROR: i is no compile-time value
cout << std::get<5>(tu); // ERROR: tu has only three elements
cout << std::get<double>(tu2) << endl; // ERROR: ambiguous
```

### ● «замена» цикла: утилита std::apply(Function, std::tuple)

```
// print std::tuple
// 1. using generic lambda with folding expression as Function
auto FoldPrint = [](auto&& ... args) {
    ((cout << std::forward<decltype(args)>(args) << " "), ...) << '\n';
};
// 2. std::apply(): call Function with elements of tuple as arguments
std::apply( FoldPrint, tu ); // 1 true TUPLE
```



● **число элементов:** `std::tuple_size<>`

```
int a[std::tuple_size_v<decltype(tu)>]; // compile-time  
cout << std::tuple_size_v<decltype(tu)> << endl; // run-time 3
```

● **упаковка и распаковка:** `std::make_tuple()`, `std::tie()`

```
auto tu3 = std::make_tuple(3,false,"tu3"); // packing values into tuple
```

```
int i; bool b; std::string s;  
std::tie(i,b,s) = tu3; // unpacking tuple into variables  
cout<<"i="<<i<<" b="<<b<<" s="<<s<< endl; // i=3 b=false s=tu3
```

```
const auto& [j,c,t] = tu3; // binding declaration C++17  
cout<<"j="<<j<<" c="<<c<<" t="<<t<< endl; // j=3 c=false t=tu3
```

## ● функция возвращающая tuple<>

```
std::tuple<double,bool> mysqrt(double x) { // (result,ok_flag)
    return (x >= 0)
        ? std::make_tuple(std::sqrt(x),true)
        : std::make_tuple(0.,false);
}

for ( auto x : {-1.1, 4.1} ) { // testing the function
    auto [res,ok] = mysqrt(x);
    if( ok ) {
        cout << "mysqrt("<< x <<" )=" << res << endl;
    } else {
        cout << "Negative x="<< x <<" in mysqrt()" << endl;
    }
}

// Negative x=-1.1 in mysqrt()
// mysqrt(4.1)=2.02485
```