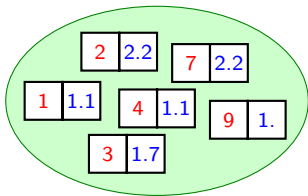


STL: map<>

👉 Ассоциативный контейнер *map* – контейнер для хранения пар (ключ, величина)

`std::map<int, double>`

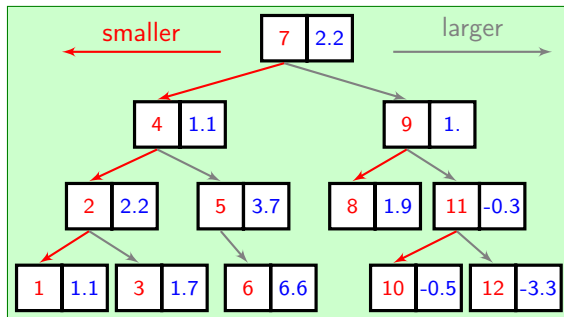


map: ассоциативный массив $m[key] = value$

- ключ (key) играет роль индекса массива
- величина (value) — 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: бинарный поиск
- 👉 для ключа нужна операция сравнения: например `оператор<()`
- 👉 значение ключа изменить нельзя, иначе сортировка нарушится

STL map: элементарные операции

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

Декларация map : параметры шаблона

<code>map<Tkey, Tel></code>	ключи сортируются с помощью <code>operator<()</code>
<code>map<Tkey, Tel, TOp></code>	задание типа сортировки <code>TOp()</code>

Создание и копирование

<code>map m</code>	пустой <code>map</code>
<code>map m(Op)</code>	пустой <code>map</code> с функцией сортировки <code>Op</code>
<code>map m1(m2)</code>	копирующий конструктор
<code>map m(beg, end)</code>	создаёт <code>map</code> с копией из <code>[beg, end)</code>
<code>map m {{k1, v1}, {k2, v2}, ...}</code>	инициализация списком (C++11)

Пример `map<int,double>`

```
typedef map<int,double> Mmap; // shorten the writing

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

for ( Mmap::iterator it = mm.begin(); it != mm.end(); ++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 с возможным созданием нового элемента:

- ✓ если ключ найден, возвращается ссылка на ассоциированное с ключом значение
- ✓ если ключа нет, **новый элемент создается конструктором по умолчанию**, после чего возвращается ссылка на него

Так делать не надо, иначе появятся новые элементы!

```
for ( int i = 1; i <= 7; i++ ) {  
    cout << " mm[" << i << "] = " << mm[i] << ", ";  
}  
cout << endl; // mm[1]= 0.75, mm[2]= 1, mm[3]= 0, mm[4]= 0,  
              // mm[5]= 0.2, mm[6]= 0, mm[7]= 0.6,
```

STL map: функция at()

(C++11)

at(key): аналог at(index) для vector

- ✓ возвращает ссылку на значение элемента с ключом `key`
- ✓ возбуждает исключение `out_of_range` если элемент с таким ключом отсутствует

Пример

```
try {  
    cerr << " mm[1]= " << mm.at(1) << endl;    // mm[1]= 0.75  
    cerr << " mm[100]= " << mm.at(100) << endl;  
} catch(out_of_range& e) {  
    cerr << " Exception: " << e.what() << endl;  
mm[100]= Exception: map::at  
}
```

Цикл for-range для map

```
cout << " mm= { ";  
for ( const auto& p : mm ) {  
    cout << "{" << p.first << ", " << p.second << "} ";  
}  
cout << "}" << endl;  
mm= { {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

👉 сравните с циклом по итераторам

```
for ( auto it = mm.begin(); it != mm.end(); ++it ) {  
    cout << it->first << ", " << it->second; // it указывает на пару  
}
```

STL: класс pair

- ☞ Контейнер `map` использует вспомогательный класс `pair` для хранения элементов (элемент `map` это объект класса `pair<Key, Value>`)
- ☞ Класс `pair<T1, T2>` содержит два элемента типа `T1` и `T2` соответственно

Члены класса `pair<T1, T2>`

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

Функции в классе pair

- ❶ Глобальная функция 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

📌 для сравнения между элементами пары используется только `operator<()`

STL map: задание функции сравнения

Функцию сортировки можно определить:

❶ с помощью `operator<()` для типа `Key`

👉 `map<Key, Elem>` – дополнительно ничего указывать не надо

❷ внешняя функция сравнения `bool FunCmp(TKey k1, TKey k2)`

👉 `map<Key, Elem, PointerFunCmp>` – указывается тип указателя на ф-ю

❸ функтор: указывается имя класса `Compare`

👉 `map<Key, Elem, Compare>`

а в классе должен быть: `bool operator()(TKey k1, TKey k2)`

👉 в шаблонах всегда указывается тип сравнения, а сами функции указывают при вызове конструктора:

```
map<Key, Elem, PointerFunCmp> mymap(FunCmp);
```

STL: объекты-функции (functors)

👉 STL широко использует объекты-функции – объекты имеющие свойства функций, так как для них определён `operator()`

Пример: `map` ("имя месяца", число дней)

объект-функцию для сравнение двух строк:

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

определение `map` с функции сравнения ключей `LtStr`:

```
typedef map<const char*, int, LtStr> ManthMap;
```

👉 более «просто» использовать `string` для ключа: `map<string, int>`


STL map: функции сравнения

1 внешняя функция

```
bool GtStr(const char* s1, const char* s2) {  
    return strcmp(s1, s2) > 0;  
}  
  
typedef bool (*FUNCPTR)(const char*, const char*); // указатель на ф-ю  
map<const char*, int, FUNCPTR> months(GtStr);
```

2 лямбда функция

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

 обратите внимание на оператор `decltype`

STL map: функции вставки и удаления

<code>m.insert(elem)</code>	вставляет элемент и возвращает пару: позицию нового элемента и код выполнения
<code>m.insert(elem,pos)</code>	то же самое! <code>pos</code> используется только как «намёк» (hint) где может находиться <code>elem</code>
<code>m.insert(beg,end)</code>	вставляет элементы из диапазона <code>[beg,end)</code>
<code>m.erase(elem)</code>	удаляет все элементы со значением <code>elem</code> и возвращает число удалённых элементов
<code>m.erase(pos)</code>	удаляет элемент на позиции <code>pos</code>
<code>m.erase(beg,end)</code>	удаляет элементы в диапазоне <code>[beg,end)</code>
<code>m.clear()</code>	удаляет все элементы

STL map: вставка

👉 В отличии от `vector` или `list` работаем с парой: `pair<Key, Value>`

❶ функция `make_pair()`

```
mm.insert(make_pair(11,2.78));  
cout << mm[11] << endl; // 2.78
```

❷ С использованием `map::value_type()`

```
std::pair<Mmap::iterator,bool> pp=mm.insert(Mmap::value_type(10,3.14));  
auto it = pp.first;  
cout<<" mm[" << it->first << "]= "<<it->second<<endl; // mm[10]= 3.14  
cout << pp.second << endl; // 1 - успех
```

👉 обратите внимание на возвращаемое значение `map::insert()`

STL map: удаление

удаляем из контейнера элементы с нулевым значением

```
mm = {{1,1}, {6,1}, {7,1}, {5,0}, {8,0}, {4,0}, {2,0}};
for(auto it = mm.begin(); it != mm.end(); ) {
    if( it->second == 0 ) {
//          mm.erase(it++);      // C++03
        it = mm.erase(it); // C++11
    } else {
        ++it;
    }
}
cout << " fin: mm= " << mm << endl; // fin: mm= { {1,1} {6,1} {7,1} }
```

☞ C++03: `mm.erase(it)` ничего не возвращает, поэтому приходится использовать `mm.erase(it++)`: увеличение итератора происходит до вызова функции, а удаляется старое значение `it`

☞ C++11: `mm.erase(it)` возвращает позицию следующего элемента

STL map: функции 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 >= 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 > k</code>)
<code>m.equal_range(k)</code>	возвращает пару <code>lower_bound(k)</code> , <code>upper_bound(k)</code>
<code>m.count(k)</code>	число элементов с ключом <code>k</code>

Пример: цикл с проверкой существования ключа

```
Mmap m2 = {{1,1}, {5,5}, {7,7}};  
for(int i = 1; i <= 7; i++) {  
    auto it = m2.find(i);  
    if( it != m2.end() ) { // проверка, что что-то найдено  
        cout << " m2[" << it->first << "]= " << it->second << ", ";  
    }  
}  
cout << endl; // m2[1]= 1,  m2[5]= 5,  m2[7]= 7,
```

STL: set<>

Неформальное определение

👉 *Ассоциативный контейнер set:*

то же что и `map`, но хранятся только ключи

👉 *Почти все функции из `map` есть и в `set` (нет `operator[]` и `at()`)*

Пример: set<>

```
#include <set>           // header for set!
#include <algorithm>
#include <iterator>
using namespace std;
// удобная печать для коротких set<int>
ostream& operator << (ostream& out, const set<int> & S) {
    for( const auto& s:S ) { out << s << " "; }
    return out;
}
```

... продолжение set<>

```
set<int> s1 = {1,2,3,4,5};
set<int> s2 = {5,3,7,9,1};
cout << " s1= " << s1 << ", s2= " << s2 << endl;
    s1= 1 2 3 4 5, s2= 1 3 5 7 9

auto ret = s2.insert(7); // try to insert existing key
if ( !ret.second )
    cout << " 7 already exist in s2" << endl; // 7 already exist in s2

set<int> intersect12;
set_intersection( s1.begin(),s1.end(),           // 1-st
                  s2.begin(),s2.end(),           // 2-nd
                  inserter(intersect12, intersect12.begin())); // result
cout << " intersect12= " << intersect12 << endl; // intersect12= 1 3 5

s1.erase(s1.begin(),s1.find(3)); // erase in range [...]
cout << " s1= " << s1 << endl;    // s1= 3 4 5
```

STL: multiset<> и multimap<>

Неформальное определение

- То же, что `set` и `map`, но могут существовать элементы с одинаковыми ключами
- Функции такие же как у `set` и `map` соответственно

Пример: multiset<>

```
multiset<int> u12(s1.begin(),s1.end());  
u12.insert(s2.begin(),s2.end());  
cout << " u12= " << u12 << endl; // u12= 1 3 3 4 5 5 7 9  
auto ipos = u12.find(3);  
cout << " 3 has position: "  
    << distance(u12.begin(),ipos)+1 << endl; // 3 has position: 2
```

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

- Расширяет концепцию `pair` на случай трех или более переменных
- Используются шаблоны с переменным числом аргументов (*variadic templates*)

Пример: `tuple<>`

```
#include <tuple>      // header for tuple!
using namespace std;
...
tuple<int,double,string> tu(1,2.2,"tu");
// get access to elements of tuple:
get<2>(tu) = "TU"; // assign TU to 2-nd element
cout<<"tu= "<<get<0>(tu)<<" "<<get<1>(tu)<<" "<< get<2>(tu)<<endl;
tu= 1 2.2 TU
```

👉 в `get<idx>`, `idx` – константа времени компиляции

```
int i = 1;
cout << get<0>(tu); // 1
cout << get<i>(tu); // ERROR: i is no compile-time value
cout << get<5>(tu); // ERROR: tu has only three elements
```

Упаковка/распаковка tuple: `make_tuple` и `tie`

```
// packing values into tuple
auto tu2 = make_tuple(2,3.3,"tu2");

int a = 0;
double b = 0;
string s;
tie(a,b,s) = tu2; // unpacking tuple into variables
cout << " a= " << a << " b= " << b << " s= " << s << endl;
a= 2 b= 3.3 s= tu2
```