C++ STL 总结-基于算法竞赛

本文介绍常用STL知识，注重应用，强调用法，不强调原理和繁杂的记忆。看过之后请多运用，多敲代码试。

> 费尽心思重新梳理了一下，注意了些美观性，修改了部分错误，添加了部分解释，编写过程非常难。

另外C++版本一定要对（可能要加编译参数 `-std=c++11` ），C++11即可，C++17或20更好。

> 使DEV支持C++20： https://blog.csdn.net/qq_50285142/article/details/122930647

> 实践才是检验真理的唯一标准!
>
> 本文章CSDN的版本： https://wyq666.blog.csdn.net/article/details/114026148 （CSDN更新可能不及时，毕竟多平台维护麻烦，最新版以本网站为主）

1 vector

1.1 介绍

`vector` 为可变长数组（动态数组），定义的 `vector` 数组可以随时添加数值和删除元素。

> 注意：**在局部区域中（比如局部函数里面）开vector数组，是在堆空间里面开的。**
>
> 在局部区域开数组是在栈空间开的，而栈空间比较小，如果开了非常长的数组就会发生爆栈。
>
> 故局部区域**不可以**开大长度数组，但是可以开大长度 `vector` 。

- 头文件：

```
#include <vector>
```

- 一维初始化：

```
vector<int> a; //定义了一个名为a的一维数组,数组存储int类型数据
vector<double> b;//定义了一个名为b的一维数组，数组存储double类型数据
vector<node> c;//定义了一个名为c的一维数组，数组存储结构体类型数据，node是结构体类型
```

指定**长度**和**初始值**的初始化

```
vector<int> v(n);// 定义一个长度为n的数组，初始值默认为0，下标范围[0, n - 1]
vector<int> v(n, 1);// v[0] 到 v[n - 1]所有的元素初始值均为1
//注意：指定数组长度之后（指定长度后的数组就相当于正常的数组了）
```

初始化中有多个元素

```
vector<int> a{1, 2, 3, 4, 5};//数组a中有五个元素，数组长度就为5
```

拷贝初始化

```
vector<int> a(n + 1, 0);
vector<int> b(a); // 两个数组中的类型必须相同,a和b都是长度为n+1，初始值都为0的数组
vector<int> c = a; // 也是拷贝初始化,c和a是完全一样的数组
```

- 二维初始化
  定义第一维固定长度为 `5`，第二维可变化的二维数组

```
vector<int> v[5];//定义可变长二维数组
//注意：行不可变（只有5行），而列可变,可以在指定行添加元素
//第一维固定长度为5，第二维长度可以改变
```

> `vector<int> v[5]` 可以这样理解：长度为5的v数组，数组中存储的是 `vector<int>` 数据类型，而该类型就是数组形式，故 `v` 为二维数组。其中每个数组元素均为空，因为没有指定长度，所以第二维可变长。可以进行下述操作：
>
> ```
> v[1].push_back(2);
> v[2].push_back(3);
> ```

行列均可变

```
//初始化二维均可变长数组
vector<vector<int>> v;//定义一个行和列均可变的二维数组
```

> 应用：可以在 `v` 数组里面装多个数组
>
> ```
> vector<int> t1{1, 2, 3, 4};
> vector<int> t2{2, 3, 4, 5};
> v.push_back(t1);
> v.push_back(t2);
> v.push_back({3, 4, 5, 6}) // {3, 4, 5, 6}可以作为vector的初始化,相当于一个无名vector
> ```

行列长度均固定 `n + 1` 行 `m + 1` 列初始值为0

```
vector<vector<int>> a(n + 1, vector<int>(m + 1, 0));
```

c++17或者c++20支持的形式（不常用），与上面相同的初始化

```
vector a(n + 1, vector(m + 1, 0));
```

1.2 方法函数

知道了如何定义初始化可变数组，下面就需要知道如何添加，删除，修改数据。

c指定为数组名称，含义中会注明算法复杂度。

代码	含义
`c.front()`	返回第一个数据$O(1)$
`c.back()`	返回数组中的最后一个数据 $O(1)$
`c.pop_back()`	删除最后一个数据$O(1)$
`c.push_back(element)`	在尾部加一个数据$O(1)$
`c.size()`	返回实际数据个数（unsigned类型）$O(1)$
`c.clear()`	清除元素个数$O(N)$，N为元素个数
`c.resize(n, v)`	改变数组大小为`n`，`n`个空间数值赋为`v`，如果没有默认赋值为`0`
`c.insert(it, x)`	向任意迭代器`it`插入一个元素`x`，$O(N)$
例：`c.insert(c.begin() + 2,-1)`	将`-1`插入`c[2]`的位置
`c.erase(first,last)`	删除`[first,last)`的所有元素，$O(N)$
`c.begin()`	返回首元素的迭代器（通俗来说就是地址）$O(1)$
`c.end()`	返回最后一个元素后一个位置的迭代器（地址）$O(1)$
`c.empty()`	判断是否为空，为空返回真，反之返回假 $O(1)$

注意：`end()`返回的是最后一个元素的后一个位置的地址，不是最后一个元素的地址，**所有STL容器均是如此**

排序

使用`sort`排序要：`sort(c.begin(), c.end());`

> `sort()`为STL函数，请参考本文最后面STL函数系列。

对所有元素进行排序，如果要对指定区间进行排序，可以对`sort()`里面的参数进行加减改动。

```
vector<int> a(n + 1);
sort(a.begin() + 1, a.end()); // 对[1, n]区间进行从小到大排序
```

1.3 访问

共三种方法：

- **下标法**： 和普通数组一样

注意：一维数组的下标是从`0`到`v.size()-1`，访问之外的数会出现越界错误

- **迭代器法**： 类似指针一样的访问，首先需要声明迭代器变量，和声明指针变量一样，可以根据代码进行理解（附有注释）。

```
vector<int> vi; //定义一个vi数组
vector<int>::iterator it = vi.begin();//声明一个迭代器指向vi的初始位置
```

- **使用auto**： 非常简便，但是会访问数组的所有元素（特别注意0位置元素也会访问到）

1.3.1 下标访问

直接和普通数组一样进行访问即可。

```
//添加元素
for(int i = 0; i < 5; i++)
    vi.push_back(i);

//下标访问
for(int i = 0; i < 5; i++)
    cout << vi[i] << " ";
cout << "\n";
```

1.3.2 迭代器访问

类似指针，迭代器就是充当指针的作用。

```
vector<int> vi{1, 2, 3, 4, 5};
//迭代器访问
vector<int>::iterator it;
// 相当于声明了一个迭代器类型的变量it
// 通俗来说就是声明了一个指针变量
```

- 方式一：

```
vector<int>::iterator it = vi.begin();
for(int i = 0; i < 5; i++)
    cout << *(it + i) << " ";
cout << "\n";
```

- 方式二

```
  vector<int>::iterator it;
  for(it = vi.begin(); it != vi.end();it ++)
      cout << *it << " ";
  //vi.end()指向尾元素地址的下一个地址

  // 或者
  auto it = vi.begin();
  while (it != vi.end()) {
      cout << *it << "\n";
      it++;
  }
```

1.3.3 智能指针

只能遍历完数组，如果要指定的内容进行遍历，需要另选方法。

`auto` 能够自动识别并获取类型。

```cpp
// 1. 输入
vector<int> a(n);
for (auto &x: a) {
    cin >> x; // 可以进行输入，注意加引用
}
// 2. 输出
vector<int> v;
v.push_back(12);
v.push_back(241);
for(auto val : v) {
    cout << val << " "; // 12 241
}

```

> `vector` 注意：
>
> - `vi[i]` 和 `*(vi.begin() + i)` 等价，与指针类似。
> - `vector` 和 `string` 的 `STL` 容器支持 `*(it + i)` 的元素访问，其它容器可能也可以支持这种方式访问，但用的不多，可自行尝试。

2 stack

2.1 介绍

栈为数据结构的一种，是STL中实现的一个先进后出，后进先出的容器。

```cpp
//头文件需要添加
#include<stack>

//声明
stack<int> s;
stack<string> s;
stack<node> s;//node是结构体类型
```

2.2 方法函数

代码	含义
`s.push(ele)`	元素 `ele` 入栈，增加元素 $O(1)$
`s.pop()`	移除栈顶元素 $O(1)$
`s.top()`	取得栈顶元素（但不删除）$O(1)$
`s.empty()`	检测栈内是否为空，空为真 $O(1)$
`s.size()`	返回栈内元素的个数 $O(1)$

2.3 栈遍历

2.3.1 栈遍历

栈只能对栈顶元素进行操作，如果想要进行遍历，只能将栈中元素一个个取出来存在数组中

```
stack<int> st;
for (int i = 0; i < 10; ++i) st.push(i);
while (!st.empty()) {
    int tp = st.top(); // 栈顶元素
    st.pop();
}
```

2.3.2 数组模拟栈进行遍历

通过一个**数组**对栈进行模拟，一个存放下标的变量 `top` 模拟指向栈顶的指针。

> 一般来说单调栈和单调队列写法均可使用额外变量 `tt` 或 `hh` 来进行模拟

特点： 比 `STL` 的 `stack` 速度更快，遍历元素方便

```
int s[100]; // 栈 从左至右为栈底到栈顶
int tt = -1; // tt 代表栈顶指针,初始栈内无元素，tt为-1

for(int i = 0; i <= 5; ++i) {
    //入栈
    s[++tt] = i;
}
// 出栈
int top_element = s[tt--];

//入栈操作示意
//  0  1  2  3  4  5
//                tt
//出栈后示意
//  0  1  2  3  4
//             tt
```

3 queue

3.1 介绍

队列是一种先进先出的数据结构。

```
//头文件
#include<queue>
//定义初始化
queue<int> q;
```

3.2 方法函数

代码	含义
`q.front()`	返回队首元素 $O(1)$
`q.back()`	返回队尾元素 $O(1)$
`q.push(element)`	尾部添加一个元素 `element` 进队$O(1)$
`q.pop()`	删除第一个元素 出队 $O(1)$
`q.size()`	返回队列中元素个数，返回值类型 `unsigned int` $O(1)$
`q.empty()`	判断是否为空，队列为空，返回 `true` $O(1)$

3.3 队列模拟

使用 `q[]` 数组模拟队列

`hh` 表示队首元素的下标，初始值为 `0`

`tt` 表示队尾元素的下标，初始值为 `-1`，表示刚**开始队列为空**

> 一般来说单调栈和单调队列写法均可使用额外变量 `tt` 或 `hh` 来进行模拟

```
#include<bits/stdc++.h>
using namespace std;
const int N = 1e5 + 5;
int q[N];

int main() {
    int hh = 0,tt = -1;
//  入队
    q[++tt] = 1;
    q[++tt] = 2;
//  将所有元素出队
    while(hh <= tt) {
        int t = q[hh++];
        printf("%d ",t);
    }
    return 0;
 }
```

4 deque

4.1 介绍

首尾都可插入和删除的队列为双端队列。

```
//添加头文件
#include<deque>
//初始化定义
deque<int> dq;
```

4.2 方法函数

> 注意双端队列的常数比较大。

代码	含义
`push_back(x)/push_front(x)`	把 `x` 插入队尾后 / 队首 $O(1)$
`back()/front()`	返回队尾 / 队首元素 $O(1)$
`pop_back() / pop_front()`	删除队尾 / 队首元素 $O(1)$
`erase(iterator it)`	删除双端队列中的某一个元素
`erase(iterator first,iterator last)`	删除双端队列中 `[first,last)` 中的元素
`empty()`	判断deque是否空 $O(1)$
`size()`	返回deque的元素数量 $O(1)$
`clear()`	清空deque

4.3 注意点

deque可以进行排序

> 双端队列排序一般不用，感觉毫无用处，使用其他STL依然可以实现相同功能

```
//从小到大
sort(q.begin(), q.end())
//从大到小排序
sort(q.begin(), q.end(), greater<int>());//deque里面的类型需要是int型
sort(q.begin(), q.end(), greater());//高版本C++才可以用
```

5. priority_queue

5.1 介绍

优先队列是在正常队列的基础上加了优先级，保证每次的队首元素都是优先级最大的。

可以实现每次从优先队列中取出的元素都是队列中**优先级最大**的一个。

它的底层是通过**堆**来实现的。

```
//头文件
#include<queue>
//初始化定义
priority_queue<int> q;
```

5.2 函数方法

代码	含义
`q.top()`	访问队首元素 $O(1)$
`q.push()`	入队 $O(logN)$
`q.pop()`	堆顶（队首）元素出队 $O(logN)$
`q.size()`	队列元素个数 $O(1)$
`q.empty()`	是否为空 $O(1)$
注意没有 `clear()`！	不提供该方法
优先队列只能通过 `top()` 访问队首元素（优先级最高的元素）	

5.3 设置优先级

5.3.1 基本数据类型的优先级

```
priority_queue<int> pq; // 默认大根堆，即每次取出的元素是队列中的最大值
priority_queue<int, vector<int>, greater<int>> q; // 小根堆，每次取出的元素是队列中的最小值
```

参数解释：

- 第一个参数：就是优先队列中存储的数据类型
- 第二个参数：

  `vector<int>` 是用来承载底层数据结构堆的容器，若优先队列中存放的是 `double` 型数据，就要填 `vector< double >`

  总之存的是什么类型的数据，就相应的填写对应类型。同时也要改动第三个参数里面的对应类型。
- 第三个参数：

  `less<int>` 表示数字大的优先级大，堆顶为最大的数字

  `greater<int>` 表示数字小的优先级大，堆顶为最小的数字

  int代表的是数据类型，也要填优先队列中存储的数据类型

下面介绍基础数据类型优先级设置的写法：

1. 基础写法（非常常用）：

```
priority_queue<int> q1; // 默认大根堆，即每次取出的元素是队列中的最大值
priority_queue<int, vector<int>, less<int> > q2; // 大根堆，每次取出的元素是队列中的最大值，同第一行

priority_queue<int, vector<int>, greater<int> > q3; // 小根堆，每次取出的元素是队列中的最小值
```

2. 自定义排序（不常见，主要是写着麻烦）：

下面的代码比较长，基础类型优先级写着太麻烦，用第一种即可。

```
struct cmp1 {
    bool operator()(int x, int y) {
        return x > y;
    }
};
struct cmp2 {
    bool operator()(const int x, const int y) {
        return x < y;
    }
};
priority_queue<int, vector<int>, cmp1> q1; // 小根堆
priority_queue<int, vector<int>, cmp2> q2; // 大根堆
```

5.3.2 高级数据类型(结构体)优先级

> 即优先队列中存储结构体类型，必须要设置优先级，即结构体的比较运算（因为优先队列的堆中要比较大小，才能将对应最大或者最小元素移到堆顶）。

优先级设置可以定义在**结构体内**进行小于号重载，也可以定义在**结构体外**。

```
//要排序的结构体（存储在优先队列里面的）
struct Point {
    int x, y;
};
```

- **版本一：自定义全局比较规则**

```
//定义的比较结构体
//注意：cmp是个结构体
struct cmp {//自定义堆的排序规则
    bool operator()(const Point& a,const Point& b) {
        return a.x < b.x;
    }
};

//初始化定义，
priority_queue<Point, vector<Point>, cmp> q; // x大的在堆顶
```

- **版本二：直接在结构体里面写**

> 因为是在结构体内部自定义的规则，一旦需要比较结构体，自动调用结构体内部重载运算符规则。

结构体内部有两种方式：

方式一：

```
struct node {
    int x, y;
    friend bool operator < (Point a, Point b) {//为两个结构体参数，结构体调用一定要写上friend
        return a.x < b.x;//按x从小到大排，x大的在堆顶
    }
};
```

方式二：（推荐此种）

```
struct node {
    int x, y;
    bool operator < (const Point &a) const {//直接传入一个参数，不必要写friend
        return x < a.x;//按x升序排列，x大的在堆顶
    }
};
```

优先队列的定义

```
priority_queue<Point> q;
```

注意： 优先队列自定义排序规则和 `sort()` 函数定义 `cmp` 函数很相似，但是最后返回的情况是**相反**的。即相同的符号，最后定义的排列顺序是完全相反的。
所以只需要记住 `sort` 的排序规则和优先队列的排序规则是相反的就可以了。

> 当理解了堆的原理就会发现，堆调整时比较顺序是孩子和父亲节点进行比较，如果是 `>` ，那么孩子节点要大于父亲节点，堆顶自然是最小值。

5.4 存储特殊类型的优先级

5.4.1 存储pair类型

- 排序规则：
  默认先对 `pair` 的 `first` 进行降序排序，然后再对 `second` 降序排序
  对 `first` 先排序，大的排在前面，如果 `first` 元素相同，再对 `second` 元素排序，保持大的在前面。

> `pair` 请参考下文

```
#include<bits/stdc++.h>
using namespace std;
int main() {
    priority_queue<pair<int, int> >q;
    q.push({7, 8});
    q.push({7, 9});
    q.push(make_pair(8, 7));
    while(!q.empty()) {
        cout << q.top().first << " " << q.top().second << "\n";
        q.pop();
    }
    return 0;
}
```

> 结果：
> 8 7
> 7 9
> 7 8

6. map

6.1 介绍

映射类似于函数的对应关系，每个 `x` 对应一个 `y`，而 `map` 是每个键对应一个值。会python的朋友学习后就会知道这和python的字典非常类似。

> 比如说：学习 对应 看书，学习 是键，看书 是值。
> 学习->看书
> 玩耍 对应 打游戏，玩耍 是键，打游戏 是值。
> 玩耍->打游戏

```
//头文件
#include<map>
//初始化定义
map<string, string> mp;
map<string, int> mp;
map<int, node> mp;//node是结构体类型
```

> map特性：map会按照键的顺序从小到大自动排序，键的类型必须可以比较大小

6.2 函数方法

6.2.1 函数方法

代码	含义
`mp.find(key)`	返回键为key的映射的迭代器 $O(logN)$ 注意：用find函数来定位数据出现位置，它返回一个迭代器。当数据存在时，返回数据所在位置的迭代器，数据不存在时，返回$mp.end()$
`mp.erase(it)`	删除迭代器对应的键和值$O(1)$
`mp.erase(key)`	根据映射的键删除键和值 $O(logN)$
`mp.erase(first,last)`	删除左闭右开区间迭代器对应的键和值 $O(last-first)$
`mp.size()`	返回映射的对数$O(1)$
`mp.clear()`	清空map中的所有元素$O(N)$
`mp.insert()`	插入元素，插入时要构造键值对
`mp.empty()`	如果map为空，返回true，否则返回false
`mp.begin()`	返回指向map第一个元素的迭代器（地址）
`mp.end()`	返回指向map尾部的迭代器（最后一个元素的**下一个**地址）
`mp.rbegin()`	返回指向map最后一个元素的迭代器（地址）
`mp.rend()`	返回指向map第一个元素前面(上一个）的逆向迭代器（地址）
`mp.count(key)`	查看元素是否存在，因为map中键是唯一的，所以存在返回1，不存在返回0
`mp.lower_bound()`	返回一个迭代器，指向键值>= **key**的第一个元素
`mp.upper_bound()`	返回一个迭代器，指向键值> key的第一个元素

6.2.2 注意点

下面说明部分函数方法的注意点

> 注意：
> 查找元素是否存在时，可以使用
> ① `mp.find()` ② `mp.count()` ③ `mp[key]`
> 但是第三种情况，如果不存在对应的 `key` 时，会自动创建一个键值对（产生一个额外的键值对空间）
> 所以为了不增加额外的空间负担，最好使用前两种方法

6.2.3 迭代器进行正反向遍历

- `mp.begin()` 和 `mp.end()` 用法：

用于正向遍历map

```
map<int,int> mp;
mp[1] = 2;
mp[2] = 3;
mp[3] = 4;
auto it = mp.begin();
while(it != mp.end()) {
    cout << it->first << " " << it->second << "\n";
    it ++;
}
```

结果:

```
1 2
2 3
3 4
```

- `mp.rbegin()` 和 `mp.rend()`

用于逆向遍历map

```
map<int,int> mp;
mp[1] = 2;
mp[2] = 3;
mp[3] = 4;
auto it = mp.rbegin();
while(it != mp.rend()) {
    cout << it->first << " " << it->second << "\n";
    it ++;
}
```

结果:

```
3 4
2 3
1 2
```

6.2.4 二分查找

二分查找 `lower_bound()` `upper_bound()`

> map的二分查找以第一个元素（即键为准），对**键**进行二分查找
> 返回值为map迭代器类型

```
#include<bits/stdc++.h>
using namespace std;

int main() {
    map<int, int> m{{1, 2}, {2, 2}, {1, 2}, {8, 2}, {6, 2}};//有序
    map<int, int>::iterator it1 = m.lower_bound(2);
    cout << it1->first << "\n";//it1->first=2
    map<int, int>::iterator it2 = m.upper_bound(2);
    cout << it2->first << "\n";//it2->first=6
    return 0;
}

```

6.3 添加元素

```
//先声明
map<string, string> mp;
```

- **方式一：**

```
mp["学习"] = "看书";
mp["玩耍"] = "打游戏";
```

- **方式二：插入元素构造键值对**

```
mp.insert(make_pair("vegetable","蔬菜"));
```

- **方式三：**

```
mp.insert(pair<string,string>("fruit","水果"));
```

- **方式四：**

```
mp.insert({"hahaha","wawawa"});
```

6.4 访问元素

6.4.1 下标访问

(大部分情况用于访问单个元素)

```
mp["菜哇菜"] = "强哇强";
cout << mp["菜哇菜"] << "\n";//只是简写的一个例子，程序并不完整
```

6.4.2 遍历访问

- 方式一：迭代器访问

```
map<string,string>::iterator it;
for(it = mp.begin(); it != mp.end(); it++) {
    //      键                  值
    // it是结构体指针访问所以要用 → 访问
    cout << it→first << " " << it→second << "\n";
    //*it是结构体变量 访问要用 . 访问
    //cout<<(*it).first<<" "<<(*it).second;
}
```

- 方式二：智能指针访问

```
for(auto i : mp)
cout << i.first << " " << i.second << endl;//键，值
```

- 方式三：对指定单个元素访问

```
map<char,int>::iterator it = mp.find('a');
cout << it → first << " " <<  it→second << "\n";
```

- 方式四：c++17特性才具有

```
for(auto [x, y] : mp)
    cout << x << " " << y << "\n";
//x,y对应键和值
```

6.5 与unordered_map的比较

这里就不单开一个大目录讲unordered_map了，直接在map里面讲了。

6.5.1 内部实现原理

map：内部用**红黑树**实现，具有**自动排序**（按键从小到大）功能。

unordered_map：内部用**哈希表**实现，内部元素无序杂乱。

6.5.2 效率比较

map：

- 优点：内部用红黑树实现，内部元素具有有序性，查询删除等操作复杂度为$O(logN)$
- 缺点：占用空间，红黑树里每个节点需要保存父子节点和红黑性质等信息，空间占用较大。

unordered_map：

- 优点：内部用哈希表实现，查找速度非常快（适用于大量的查询操作）。
- 缺点：建立哈希表比较耗时。

> 两者方法函数基本一样，差别不大。
>
> 注意：
>
> - 随着内部元素越来越多，两种容器的插入删除查询操作的时间都会逐渐变大，效率逐渐变低。
> - 使用 `[]` 查找元素时，如果元素不存在，两种容器**都是**创建一个空的元素；如果存在，会正常索引对应的值。所以如果查询过多的不存在的元素值，容器内部会创建大量的空的键值对，后续查询创建删除效率会**大大降低**。
> - 查询容器内部元素的最优方法是：先判断存在与否，再索引对应值（适用于这两种容器）
>
> ```
> // 以 map 为例
> map<int, int> mp;
> int x = 999999999;
> if(mp.count(x)) // 此处判断是否存在x这个键
>     cout << mp[x] << "\n";    // 只有存在才会索引对应的值，避免不存在x时多余空元素的创建
> ```

另外：

> 还有一种映射：`multimap`
>
> 键可以重复，即一个键对应多个值，如要了解，可以自行搜索。

7 set

7.1 介绍

set容器中的元素不会重复，当插入集合中已有的元素时，并不会插入进去，而且set容器里的元素自动从小到大排序。

即：set里面的元素**不重复 且有序**

```
//头文件
#include<set>
//初始化定义
set<int> s;
```

7.2 函数方法

代码	含义
`s.begin()`	返回set容器的第一个元素的地址（迭代器）$O(1)$
`s.end()`	返回set容器的最后一个元素的下一个地址（迭代器）$O(1)$
`s.rbegin()`	返回逆序迭代器，指向容器元素最后一个位置$O(1)$
`s.rend()`	返回逆序迭代器，指向容器第一个元素前面的位置$O(1)$
`s.clear()`	删除set容器中的所有的元素,返回unsigned int类型$O(N)$
`s.empty()`	判断set容器是否为空$O(1)$
`s.insert()`	插入一个元素
`s.size()`	返回当前set容器中的元素个数$O(1)$
`erase(iterator)`	删除定位器iterator指向的值
`erase(first,second)`	删除定位器first和second之间的值
`erase(key_value)`	删除键值key_value的值
查找	
`s.find(element)`	查找set中的某一元素，有则返回该元素对应的迭代器，无则返回结束迭代器
`s.count(element)`	查找set中的元素出现的个数，由于set中元素唯一，此函数相当于查询element是否出现
`s.lower_bound(k)`	返回大于等于k的第一个元素的迭代器$O(logN)$
`s.upper_bound(k)`	返回大于k的第一个元素的迭代器$O(logN)$

7.3 访问

- **迭代器访问**

```
for(set<int>::iterator it = s.begin(); it != s.end(); it++)
    cout << *it << " ";
```

- **智能指针**

```
for(auto i : s)
    cout << i << endl;
```

- **访问最后一个元素**

```
//第一种
cout << *s.rbegin() << endl;
```

```
 //第二种
set<int>::iterator iter = s.end();
iter--;
cout << (*iter) << endl; //打印2;
```

```
//第三种
cout << *(--s.end()) << endl;
```

7.4 重载<运算符

- **基础数据类型**

方式一：改变set排序规则，set中默认使用less比较器，即从小到大排序。（常用）

```
set<int> s1; // 默认从小到大排序
set<int, greater<int> > s2; // 从大到小排序
```

方式二：重载运算符。（很麻烦，不太常用，没必要）

```
//重载 < 运算符
struct cmp {
    bool operator () (const int& u, const int& v) const {
       // return + 返回条件
       return u > v;
    }
};
set<int, cmp> s;

for(int i = 1; i <= 10; i++)
    s.insert(i);
for(auto i : s)
    cout << i << " ";
// 10 9 8 7 6 5 4 3 2 1
```

方式三：初始化时使用匿名函数定义比较规则

```
set<int, function<bool(int, int)>> s([&](int i, int j){
    return i > j; // 从大到小
});
for(int i = 1; i <= 10; i++)
    s.insert(i);
for(auto x : s)
    cout << x << " ";
```

- **高级数据类型（结构体）**

直接重载结构体运算符即可，让结构体可以比较。

```cpp
struct Point {
    int x, y;
    bool operator < (const Point &p) const {
        // 按照点的横坐标从小到大排序,如果横坐标相同,纵坐标从小到大
        if(x == p.x)
            return y < p.y;
        return x < p.x;
    }
};

set<Point> s;
for(int i = 1; i <= 5; i++) {
    int x, y;
    cin >> x >> y;
    s.insert({x, y});
}
/* 输入
5 4
5 2
3 7
3 5
4 8
*/

for(auto i : s)
    cout << i.x << " " << i.y << "\n";
/* 输出
3 5
3 7
4 8
5 2
5 4
*/
```

7.5 其它set

`multiset`:元素可以重复，且元素有序

`unordered_set` ： 元素无序且只能出现一次

`unordered_multiset` ： 元素无序可以出现多次

8 pair

8.1 介绍

pair只含有两个元素，可以看作是只有两个元素的结构体。

应用:

- 代替二元结构体
- 作为map键值对进行插入（代码如下）

```
map<string,int>mp;
mp.insert(pair<string,int>("xingmaqi",1));
// mp.insert(make_pair("xingmaqi", 1));
// mp.insert({"xingmaqi", 1});
```

```
//头文件
#include<utility>

//1.初始化定义
pair<string,int> p("wangyaqi",1);//带初始值的
pair<string,int> p;//不带初始值的

//2.赋值
p = {"wang", 18};
p = make_pair("wang", 18);
p = pair<string, int>("wang", 18);
```

8.2 访问

```
//定义结构体数组
pair<int,int> p[20];
for(int i = 0; i < 20; i++) {
    //和结构体类似，first代表第一个元素，second代表第二个元素
    cout << p[i].first << " " << p[i].second;
}
```

9 string

9.1 介绍

string是一个字符串类，和 `char` 型字符串类似。

可以把string理解为一个字符串类型，像int一样可以定义

9.2 初始化及定义

```
//头文件
#include<string>

//1.
string str1; //生成空字符串

//2.
string str2("123456789"); //生成"1234456789"的复制品

//3.
string str3("12345", 0, 3);//结果为"123" ，从0位置开始，长度为3

//4.
string str4("123456", 5); //结果为"12345" ，长度为5
```

```
15
16    //5.
17    string str5(5, '2'); //结果为"22222" ,构造5个字符'2'连接而成的字符串
18
19    //6.
20    string str6(str2, 2); //结果为"3456789"，截取第三个元素（2对应第三位）到最后
21
```

简单使用

- 访问单个字符：

```
#include<iostream>
#include<string>
using namespace std;
int main() {
    string s = "xing ma qi!!!";
    for(int i = 0; i < s.size(); i++)
        cout << s[i] << " ";
    return 0;
}
```

- `string` 数组使用：

```
#include<iostream>
#include<string>
using namespace std;
int main() {
    string s[10];
    for(int i = 1; i < 10; i++) {
        s[i] = "loading ...   " ;
        cout << s[i] << i << "\n";
    }
    return 0;
}
```

结果：

```
loading ...   1
loading ...   2
loading ...   3
loading ...   4
loading ...   5
loading ...   6
loading ...   7
loading ...   8
loading ...   9
```

9.3 string 特性

- 支持**比较**运算符
  string字符串支持常见的比较操作符 `（>, ≥ ,<, ≤ , ═ , ≠）` ，支持 `string` 与 `C-string` 的比较（如 `str < "hello"` ）。
  在使用 `>, ≥ ,<, ≤` 这些操作符的时候是根据“当前字符特性”将字符按 `字典顺序` 进行逐一得 比较。字典排序靠前的字符小， 比较的顺序是从前向后比较，遇到不相等的字符就按这个位置上的两个字符的比较结果确定两个字符串的大小（前面减后面）。
  同时， `string ("aaaa") <string(aaaaa)` 。

- 支持 `+` **运算**符，代表拼接字符串
  string字符串可以拼接，通过"+"运算符进行拼接。

```
string s1 = "123";
string s2 = "456";
string s = s1 + s2;
cout << s;   //123456
```

9.4 读入详解

读入字符串，遇空格，回车结束

```
string s;
cin >> s;
```

读入一行字符串（包括空格），遇回车结束

```
string s;
getline(cin, s);
```

注意: `getline(cin, s)` 会获取前一个输入的换行符，需要在前面添加读取换行符的语句。如： `getchar()` 或 `cin.get()`

错误读取：

```
int n;
string s;
cin >> n;
getline(cin, s); //此时读取相当于读取了前一个回车字符
```

正确读取：

```
int n;
string s;
cin >> n;
getchar(); //cin.get()
getline(cin, s);//可正确读入下一行的输入
```

> `cin` 与 `cin.getline()` 混用
>
> cin输入完后，回车，cin遇到回车结束输入，但回车还在输入流中，cin并不会清除，导致 `getline()` 读取回车，结束。
> 需要在cin后面加 `cin.ignore()` ；主动删除输入流中的换行符。（不常用）

cin和cout解锁

代码（写在main函数开头）：

```
ios::sync_with_stdio(false);
cin.tie(0),cout.tie(0);
```

> 为什么要进行 `cin` 和 `cout` 的解锁，原因是：
>
> 在一些题目中，读入的**数据量很大**，往往超过了1e5（10^5）的数据量,而 `cin` 和 `cout` 的读入输出的速度**很慢**（是因为 `cin` 和 `cout` 为了兼容C语言的读入输出在性能上做了妥协），远不如 `scanf` 和 `printf` 的速度，具体原因可以搜索相关的博客进行了解。
>
> **所以**对 `cin` 和 `cout` 进行解锁使 `cin` 和 `cout` 的速度几乎接近 `scanf` 和 `printf` ，避免输入输出超时。

注意： `cin cout` 解锁使用时，不能与 `scanf,getchar, printf,cin.getline()` 混用，一定要注意，会出错。

> **string与C语言字符串（C-string）的区别**
>
> - string
>   是C++的一个类，专门实现字符串的相关操作。具有丰富的操作方法，数据类型为 `string` ，字符串结尾没有 `\0` 字符
> - C-string
>   C语言中的字符串，用char数组实现，类型为 `const char *` ,字符串结尾以 `\0` 结尾

一般来说string向char数组转换会出现一些问题，所以为了能够实现转换，string有一个方法 `c_str()` 实现string向char数组的转换。

```
1    string s = "xing ma qi";
2    char s2[] = s.c_str();
```

9.5 函数方法

- **获取字符串长度**

代码	含义
`s.size()` 和 `s.length()`	返回string对象的字符个数，他们执行效果相同。
`s.max_size()`	返回string对象最多包含的字符数，超出会抛出length_error异常
`s.capacity()`	重新分配内存之前，string对象能包含的最大字符数

- **插入**

代码	含义
`s.push_back()`	在末尾插入
例： `s.push_back('a')`	末尾插入一个字符a
`s.insert(pos,element)`	在pos位置插入element
例： `s.insert(s.begin(),'1')`	在第一个位置插入1字符
`s.append(str)`	在s字符串结尾添加str字符串
例： `s.append("abc")`	在s字符串末尾添加字符串“abc”

- **删除**

代码	含义
`erase(iterator p)`	删除字符串中p所指的字符
`erase(iterator first, iterator last)`	删除字符串中迭代器区间 `[first,last)` 上所有字符
`erase(pos, len)`	删除字符串中从索引位置pos开始的len个字符
`clear()`	删除字符串中所有字符

- **字符替换**

代码	含义
`s.replace(pos,n,str)`	把当前字符串从索引pos开始的n个字符替换为str
`s.replace(pos,n,n1,c)`	把当前字符串从索引pos开始的n个字符替换为n1个字符c
`s.replace(it1,it2,str)`	把当前字符串 `[it1,it2)` 区间替换为str **it1 ,it2为迭代器哦**

- **大小写转换**

法一：

代码	含义
`tolower(s[i])`	转换为小写
`toupper(s[i])`	转换为大写

法二：

通过stl的 `transform` 算法配合 `tolower` 和 `toupper` 实现。
有4个参数，前2个指定要转换的容器的起止范围，第3个参数是结果存放容器的起始位置，第4个参数是一元运算。

```
1   string s;
2   transform(s.begin(),s.end(),s.begin(),::tolower);//转换小写
3   transform(s.begin(),s.end(),s.begin(),::toupper);//转换大写
```

- **分割**

代码	含义
`s.substr(pos,n)`	截取从pos索引开始的n个字符

- **查找**

代码	含义
`s.find (str, pos)`	在当前字符串的pos索引位置（默认为0）开始，查找子串str，返回找到的位置索引，-1表示查找不到子串
`s.find (c, pos)`	在当前字符串的pos索引位置（默认为0）开始，查找字符c，返回找到的位置索引，-1表示查找不到字符
`s.rfind (str, pos)`	在当前字符串的pos索引位置开始，反向查找子串s，返回找到的位置索引，-1表示查找不到子串
`s.rfind (c,pos)`	在当前字符串的pos索引位置开始，反向查找字符c，返回找到的位置索引，-1表示查找不到字符
`s.find_first_of (str, pos)`	在当前字符串的pos索引位置（默认为0）开始，查找子串s的字符，返回找到的位置索引，-1表示查找不到字符
`s.find_first_not_of (str,pos)`	在当前字符串的pos索引位置（默认为0）开始，查找第一个不位于子串s的字符，返回找到的位置索引，-1表示查找不到字符
`s.find_last_of(str, pos)`	在当前字符串的pos索引位置开始，查找最后一个位于子串s的字符，返回找到的位置索引，-1表示查找不到字符
`s.find_last_not_of ( str, pos)`	在当前字符串的pos索引位置开始，查找最后一个不位于子串s的字符，返回找到的位置索引，-1表示查找不到子串

```cpp
#include<string>
#include<iostream>
int main() {
    string s("dog bird chicken bird cat");
//字符串查找———找到后返回首字母在字符串中的下标
// 1. 查找一个字符串
    cout << s.find("chicken") << endl;// 结果是：9

// 2. 从下标为6开始找字符'i'，返回找到的第一个i的下标
    cout << s.find('i',6) << endl;// 结果是：11

// 3. 从字符串的末尾开始查找字符串，返回的还是首字母在字符串中的下标
    cout << s.rfind("chicken") << endl;// 结果是：9

// 4. 从字符串的末尾开始查找字符
    cout << s.rfind('i') << endl;// 结果是：18因为是从末尾开始查找，所以返回第一次找到的字符

// 5. 在该字符串中查找第一个属于字符串s的字符
    cout << s.find_first_of("13br98") << endl;// 结果是：4——b

// 6. 在该字符串中查找第一个不属于字符串s的字符————先匹配dog，然后bird匹配不到，所以打印4
    cout << s.find_first_not_of("hello dog 2006") << endl; // 结果是：4
    cout << s.find_first_not_of("dog bird 2006") << endl;  // 结果是：9

// 7. 在该字符串最后中查找第一个属于字符串s的字符
    cout << s.find_last_of("13r98") << endl;// 结果是：19

// 8. 在该字符串最后中查找第一个不属于字符串s的字符———先匹配t--a——c，然后空格匹配不到，所以打印21
    cout << s.find_last_not_of("teac") << endl;// 结果是：21
}
```

- **排序**

```
sort(s.begin(),s.end());  //按ASCII码排序
```

10 bitset

10.1 介绍

bitset 在 bitset 头文件中，它类似数组，并且每一个元素只能是 0 或 1，每个元素只用 1 bit空间

```
//头文件
#include<bitset>
```

10.2 初始化定义

初始化方法

代码	含义
`bitset<n> a`	a有n位，每位都为0
`bitset<n> a(b)`	a是unsigned long型u的一个副本
`bitset<n> a(s)`	a是string对象s中含有的位串的副本
`bitset<n> a(s, pos, n)`	a是s中从位置pos开始的n个位的副本

> 注意：`n` 必须为常量表达式

演示代码：

```
#include<bits/stdc++.h>
using namespace std;
int main() {
    bitset<4> bitset1;      //无参构造，长度为 4，默认每一位为0

    bitset<9> bitset2(12);  //长度为9，二进制保存，前面用0补充

    string s = "100101";
    bitset<10> bitset3(s);    //长度为10，前面用0补充

    char s2[] = "10101";
    bitset<13> bitset4(s2);    //长度为13，前面用0补充

    cout << bitset1 << endl;    //0000
    cout << bitset2 << endl;    //000001100
    cout << bitset3 << endl;    //0000100101
    cout << bitset4 << endl;  //0000000010101
    return 0;
}
```

10.3 特性

`bitset` 可以进行**位操作**

```
bitset<4> foo (string("1001"));
bitset<4> bar (string("0011"));

cout << (foo ^= bar) << endl;// 1010 (foo对bar按位异或后赋值给foo)

cout << (foo &= bar) << endl;// 0001 (按位与后赋值给foo)

cout << (foo |= bar) << endl;// 1011 (按位或后赋值给foo)

cout << (foo<<=2) << endl;// 0100 (左移2位，低位补0，有自身赋值)

cout << (foo>>=1) << endl;// 0100 (右移1位，高位补0，有自身赋值)

cout << (~bar) << endl;// 1100 (按位取反)

cout << (bar<<1) << endl;// 0110 (左移，不赋值)

cout << (bar>>1) << endl;// 0001 (右移，不赋值)

cout << (foo==bar) << endl;// false (1001==0011为false)

cout << (foo!=bar) << endl;// true  (1001!=0011为true)

cout << (foo&bar) << endl;// 0001 (按位与，不赋值)

cout << (foo|bar) << endl;// 1011 (按位或，不赋值)

cout << (foo^bar) << endl;// 1010 (按位异或，不赋值)
```

访问

```
//可以通过 [] 访问元素(类似数组)，注意最低位下标为0，类似于数的二进制表示，如下：
bitset<4> f("1011");
for (int i = 0; i < 4; ++i) {
    cout << f[i];
} // 输出1101
```

10.4 方法函数

代码	含义
`b.any()`	b中是否存在置为1的二进制位，有 返回true
`b.none()`	b中是否没有1，没有 返回true
`b.count()`	b中为1的个数
`b.size()`	b中二进制位的个数
`b.test(pos)`	测试b在pos位置是否为1，是 返回true
`b[pos]`	返回b在pos处的二进制位
`b.set()`	把b中所有位都置为1
`b.set(pos)`	把b中pos位置置为1
`b.reset()`	把b中所有位都置为0
`b.reset(pos)`	把b中pos位置置为0
`b.flip()`	把b中所有二进制位取反
`b.flip(pos)`	把b中pos位置取反
`b.to_ulong()`	用b中同样的二进制位返回一个unsigned long值

10.5 bitset优化

一般会使用bitset来优化时间复杂度，大概时间复杂度会除64或32，例如没有优化的时间复杂度为 $O(NM)$ ，使用bitset优化后复杂度可能就为 $O(\frac{NM}{64})$

bitset还有开动态空间的技巧，bitset常用在 `01背包` 优化等算法中

```
// 动态长度bitset实现
const int N = 1e6 + 5; // 开空间的上限，一般为数据范围附近的值
template <int len = 1>
void bitset_(int sz) { // sz即为想要开的大小
    if (len < sz) { bitset_<min(len * 2, N)>(sz); return; }
    bitset<len + 1> dp;
    // 具体算法的实现
}
```

11 array

11.1 介绍

头文件

```
#include<array>
```

`array`是C++11新增的容器，效率与普通数据相差无几，比`vector`效率要高，自身添加了一些成员函数。

和其它容器不同，array 容器的大小是**固定**的，无法动态的扩展或收缩，**只允许访问或者替换存储的元素。**

注意：

`array`的使用要在`std`命名空间里

11.2 声明与初始化

基础数据类型

声明一个大小为100的`int`型数组，元素的值不确定

```
array<int, 100> a;
```

声明一个大小为100的`int`型数组，初始值均为`0`(初始值与默认元素类型等效)

```
array<int, 100> a{};
```

声明一个大小为100的`int`型数组，初始化部分值，其余全部为`0`

```
array<int, 100> a{1, 2, 3};
```

或者可以用等号

```
array<int, 100> a = {1, 2, 3};
```

高级数据类型

不同于数组的是对元素类型不做要求，可以套结构体

```
array<string, 2> s = {"ha", string("haha")};
array<node, 2> a;
```

11.3 存取元素

- 修改元素

```
array<int, 4> a = {1, 2, 3, 4};
a[0] = 4;
```

- 访问元素

下标访问

```
array<int, 4> a = {1, 2, 3, 4};
for(int i = 0; i < 4; i++)
    cout << a[i] << " \n"[i == 3];
```

利用`auto`访问

```
for(auto i : a)
    cout << i << " ";
```

迭代器访问

```
auto it = a.begin();
for(; it != a.end(); it++)
    cout << *it << " ";
```

`at()` 函数访问

下标为 `1` 的元素加上下标为 `2` 的元素，答案为 `5`

```
array<int, 4> a = {1, 2, 3, 4};
int res = a.at(1) + a.at(2);
cout << res << "\n";
```

`get` 方法访问

将 `a` 数组下标为 `1` 位置处的值改为 `x`

注意：获取的下标只能写数字，不能填变量

```
get<1>(a) = x;
```

11.4 成员函数

成员函数	功能
`begin()`	返回容器中第一个元素的访问迭代器（地址）
`end()`	返回容器最后一个元素之后一个位置的访问迭代器（地址）
`rbegin()`	返回最后一个元素的访问迭代器（地址）
`rend()`	返回第一个元素之前一个位置的访问迭代器（地址）
`size()`	返回容器中元素的数量，其值等于初始化 array 类的第二个模板参数 `N`
`max_size()`	返回容器可容纳元素的最大数量，其值始终等于初始化 array 类的第二个模板参数 N
`empty()`	判断容器是否为空
`at(n)`	返回容器中 n 位置处元素的引用，函数会自动检查 n 是否在有效的范围内，如果不是则抛出 out_of_range 异常
`front()`	返回容器中第一个元素的直接引用，函数不适用于空的 array 容器
`back()`	返回容器中最后一个元素的直接引用，函数不适用于空的 array 容器。
`data()`	返回一个指向容器首个元素的指针。利用该指针，可实现复制容器中所有元素等类似功能
`fill(x)`	将 `x` 这个值赋值给容器中的每个元素,相当于初始化
`array1.swap(array2)`	交换 array1 和 array2 容器中的所有元素，但前提是它们具有相同的长度和类型

11.5 部分用法示例

`data()`

指向底层元素存储的指针。对于非空容器，返回的指针与首元素地址比较相等。

`at()`

下标为 `1` 的元素加上下标为 `2` 的元素，答案为 `5`

```
array<int, 4> a = {1, 2, 3, 4};
int res = a.at(1) + a.at(2);
cout << res << "\n";
```

`fill()`

array的 `fill()` 函数，将 `a` 数组全部元素值变为 `x`

```
a.fill(x);
```

另外还有其它的 `fill()` 函数:将 `a` 数组$[begin, end)$全部值变为 `x`

```
fill(a.begin(), a.end(), x);
```

get方法获取元素值

将 `a` 数组下标为 `1` 位置处的值改为 `x`

注意:获取的下标只能写数字，不能填变量

```
get<1>(a) = x;
```

排序

```
sort(a.begin(), a.end());
```

12 tuple

12.1 介绍

tuple模板是pair的泛化，可以封装不同类型任意数量的对象。

可以把tuple理解为pair的扩展，tuple可以声明二元组，也可以声明三元组。

tuple可以等价为**结构体**使用

头文件

```
#include <tuple>
```

12.2 声明初始化

声明一个空的 `tuple` 三元组

```
tuple<int, int, string> t1;
```

赋值

```
t1 = make_tuple(1, 1, "hahaha");
```

创建的同时初始化

```
tuple<int, int, int, int> t2(1, 2, 3, 4);
```

可以使用pair对象构造tuple对象，但tuple对象必须是两个元素

```
auto p = make_pair("wang", 1);
tuple<string, int> t3 {p}; //将pair对象赋给tuple对象
```

12.3 元素操作

获取tuple对象 `t` 的第一个元素

```
int first = get<0>(t);
```

修改tuple对象 `t` 的第一个元素

```
get<0>(t) = 1;
```

12.4 函数操作

- 获取元素个数

```
tuple<int, int, int> t(1, 2, 3);
cout << tuple_size<decltype(t)>::value << "\n"; // 3
```

- 获取对应元素的值

通过 `get<n>(obj)` 方法获取, `n` 必须为数字不能是变量

```
tuple<int, int, int> t(1, 2, 3);
cout << get<0>(t) << '\n'; // 1
cout << get<1>(t) << '\n'; // 2
cout << get<2>(t) << '\n'; // 3
```

- 通过 `tie` 解包 获取元素值

`tie` 可以让tuple变量中的三个值依次赋到tie中的三个变量中

```
int one, three;
string two;
tuple<int, string, int> t(1, "hahaha", 3);
tie(one, two, three) = t;
cout << one << two << three << "\n"; // 1hahaha3
```

STL函数

accumulate

```
accumulate(beg, end, init)
```

复杂度： $O(N)$

> 作用：对一个序列的元素求和

`init` 为对序列元素求和的**初始值**

返回值类型：与 `init` 相同

- **基础累加求和：**

```
int a[]={1, 3, 5, 9, 10};

//对[0,2]区间求和，初始值为0，结果为0 + 1 + 3 + 5 = 9
int res1 = accumulate(a, a + 3, 0);

//对[0,3]区间求和，初始值为5，结果为5 + 1 + 3 + 5 + 9 = 23
int res2 = accumulate(a, a + 4, 5);
```

- **自定义二元对象求和：**

使用lambda表达式

```
typedef long long ll;
struct node {
    ll num;
} st[10];

for(int i = 1; i ⩽ n; i++)
    st[i].num = i + 10000000000;
//返回值类型与init一致，同时注意参数类型（a）也要一样
//初始值为1，累加1+10000000001+10000000002+10000000003=30000000007
ll res = accumulate(st + 1, st + 4, 1ll, [](ll a,node b) {
    return a + b.num;
});
```

atoi

```
atoi(const char *)
```

> 将字符串转换为 `int` 类型

注意参数为 `char` 型数组，如果需要将string类型转换为int类型，可以使用 `stoi` 函数（参考下文），或者将 `string` 类型转换为 `const char *` 类型。

关于输出数字的范围：
`atoi` **不做**范围检查，如果超出上界，输出上界，超出下界，输出下界。
`stoi` **会做**范围检查，默认必须在 `int` 范围内，如果超出范围，会出现RE（Runtime Error）错误。

```
string s = "1234";
int a = atoi(s.c_str());
cout << a << "\n"; // 1234
```

或者

```
char s[] = "1234";
int a = atoi(s);
cout << a << "\n";
```

fill

```
fill(beg, end, num)
```

复杂度： $O(N)$

> 对一个序列进行初始化赋值

```
//对a数组的所有元素赋1
int a[5];
fill(a, a + 5, 1);
for(int i = 0; i < 5; i++)
    cout << a[i] << " ";
//1 1 1 1 1
```

注意区分memset：

`memset()` 是按**字节**进行赋值，对于初始化赋 `0` 或 `-1` 有比较好的效果.

如果赋某个特定的数会**出错**，赋值特定的数建议使用 `fill()`

is_sorted

```
is_sorted(beg, end)
```

复杂度： $O(N)$

> 判断序列是否有序（升序），返回 `bool` 值

```
//如果序列有序，输出YES
if(is_sorted(a, a + n))
    cout << "YES\n";
```

iota

```
iota(beg, end)
```

> 让序列递增赋值

```
vector<int> a(10);
iota(a.begin(), a.end(), 0);
for(auto i : a)
    cout << i << " ";
// 0 1 2 3 4 5 6 7 8 9
```

lower_bound + upper_bound

复杂度： $O(logN)$

> 作用：二分查找

```
//在a数组中查找第一个大于等于x的元素，返回该元素的地址
lower_bound(a, a + n, x);
//在a数组中查找第一个大于x的元素，返回该元素的地址
upper_bound(a, a + n, x);

//如果未找到，返回尾地址的下一个位置的地址
```

max_element+min_element

复杂度： $O(N)$

> 找最大最小值

```
//函数都是返回地址，需要加*取值
int mx = *max_element(a, a + n);
int mn = *min_element(a, a + n);
```

max+min

复杂度： $O(1)$

> 找多个元素的最大值和最小值

```
//找a，b的最大值和最小值
mx = max(a, b);
mn = min(a, b);
```

```
//找到a,b,c,d的最大值和最小值
mx = max({a, b, c, d});
mn = min({a, b, c, d});
```

minmax

```
minmax(a, b)
```

复杂度： $O(1)$

> 返回一个 `pair` 类型，第一个元素是 `min(a, b)` ， 第二个元素是 `max(a, b)`

```
pair<int, int> t = minmax(4, 2);
// t.first = 2, t.second = 4
```

minmax_element

```
minmax_element(beg, end)
```

复杂度： $O(N)$

> 返回序列中的最小和最大值组成pair的对应的地址，返回类型为 `pair<vector<int>::iterator, vector<int>::iterator>`

```
int n = 10;
vector<int> a(n);
iota(a.begin(), a.end(), 1);
auto t = minmax_element(a.begin(), a.end()); // 返回的是最小值和最大值对应的地址
// *t.first = 1, *t.second = 10 输出对应最小最大值时需要使用指针
```

nth_element

```
nth_element(beg, nth, end)
```

复杂度： 平均$O(N)$

> 寻找第序列第n小的值

`nth` 为一个迭代器，指向序列中的一个元素。第n小的值恰好在 `nth` 位置上。

执行 `nth_element()` 之后，序列中的元素会围绕nth进行划分： **nth之前的元素都小于等于它，而之后的元素都大于等于它**

实例：求序列中的第3小的元素

```
nth_element(a, a + 2, a + n);
cout << a[2] << '\n';
```

next_permutation

```
next_permutation(beg, end)
```

复杂度： $O(N)$

> 求序列的下一个排列，下一个排列是字典序大一号的排列

返回`true`或`false`

- `next_permutation(beg, end)`

  如果是最后一个排列，返回`false`,否则求出下一个序列后，返回`true`

```
//对a序列进行重排
next_permutation(a, a + n);
```

应用：求所有的排列

输出`a`的所有排列

```
// 数组a不一定是最小字典序序列，一定注意将它排序
sort(a, a + n);
do {
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
} while(next_permutation(a, a + n));
```

- `prev_permutation(beg, end)`

> 求出前一个排列，如果序列为最小的排列，将其重排为最大的排列，返回false

partial_sort

```
partial_sort(beg, mid, end)
```

复杂度： 大概$O(NlogM)$ `M`为距离

> 部分排序,排序mid-beg个元素，mid为要排序区间元素的尾后的一个位置
>
> 从beg到mid**前**的元素都排好序

对a数组前5个元素排序按从小到大排序

```
int a[] = {1,2,5,4,7,9,8,10,6,3};
partial_sort(a, a + 5, a + 10);
for(int i = 0; i < 10; i++)
    cout << a[i] << ' ';
//1 2 3 4 5 9 8 10 7 6
//前五个元素都有序
```

也可以添加自定义排序规则：

`partial_sort(beg,mid,end,cmp)`

对a的前五个元素都是降序排列

```
int a[] = {1,2,5,4,7,9,8,10,6,3};
partial_sort(a, a + 5, a + 10, greater<int>());
for(int i = 0; i < 10; i++)
    cout << a[i] << ' ';
//10 9 8 7 6 1 2 4 5 3
//前五个元素降序有序
```

random_shuffle

复杂度： $O(N)$

> 1. 随机打乱序列的顺序
> 2. `random_shuffle` 在 `C++14` 中被弃用，在 `C++17` 中被废除，C++11之后应尽量使用 `shuffle` 来代替。

```
vector<int> b(n);
iota(b.begin(), b.end(), 1);// 序列b递增赋值 1, 2, 3, 4, ...
// 对a数组随机重排
random_shuffle(a, a + n);
// C++11之后尽量使用shuffle
shuffle(b.begin(), b.end());
```

reverse

```
reverse(beg,end)
```

复杂度： $O(N)$

> 对序列进行翻转

```
string s = "abcde";
reverse(s.begin(), s.end());//对s进行翻转
cout << s << '\n';//edcba

//对a数组进行翻转
int a[] = {1, 2, 3, 4};
reverse(a, a + 4);
cout << a[0] << a[1] << a[2] << a[3];//4321
```

set_union, set_intersection,set_difference

复杂度：$O(N+M)$

> 求两个集合的并集，交集，差集。手动实现双指针就可以搞定，嫌麻烦可以使用该函数。

注意：两个集合必须为有序集合，所以下面演示代码使用了排序。vector容器可以替换成set容器，因为set自动会对元素进行排序。

函数的参数有五个，前两个为第一个容器的首尾迭代器，第三四个为第二个容器的首尾迭代器，最后一个为插入位置，即将结果插入到哪个地址之后。

```
vector<int> a = {4, 5, 2, 1, 8}, b = {5, 3, 8, 9, 2};
sort(a.begin(), a.end()); // 1 2 4 5 8
sort(b.begin(), b.end()); // 2 3 5 8 9
vector<int> c, d, e;
// a并b：1 2 3 4 5 8 9
set_union(a.begin(), a.end(), b.begin(), b.end(), inserter(c, c.begin()));
// a交b：2 5 8
set_intersection(a.begin(), a.end(), b.begin(), b.end(), inserter(d, d.begin()));
// a差b： 1 4
set_difference(a.begin(), a.end(), b.begin(), b.end(), inserter(e, e.begin()));
```

sort

复杂度： $O(NlogN)$

> 作用：对一个序列进行排序

```
//原型：
sort(beg, end);
sort(beg, end, cmp);
```

几种排序的常见操作：

- 操作一：对数组正常升序排序

```
int a[N]; // 普通数组定义
// 对 a 数组的[1, n]位置进行从小到大排序
sort(a + 1, a + 1 + n);

vector<int> b(n + 1); // vector数组定义
sort(b.begin() + 1, b.end());
```

- 操作二：使用第三个参数，进行降序排序

```
//对a数组的[0, n-1]位置从大到小排序
sort(a, a + n, greater<int>());
//对a数组的[0, n-1]位置从小到大排序
sort(a, a + n, less<int>());

vector<int> b(n + 1);
sort(b.begin() + 1, b.end()); // 升序
sort(b.begin() + 1, b.end(), greater<int>()); // 降序
```

- 操作三：另外一种降序排序方法，针对 `vector`

```
vector<int> a(n);
sort(a.rbegin(), a.rend()); // 使用反向迭代器进行降序排序
```

- 操作四：自定义排序规则

```cpp
// 1. 使用函数自定义排序，定义比较函数
bool cmp(node a, node b) {
    //按结构体里面的x值降序排列
    return a.x > b.x;
}
sort(node, node + n, cmp); // 只能接受 以函数为形式的自定义排序规则，无法接受以结构体为形式的自定义排序规则

// 2. 或者使用匿名函数自定义排序规则
sort(node, node + n, [](node a, node b) {
    return a.x > b.x;
});
```

stable_sort

复杂度： $O(NlogN)$

> 功能和 `sort()` 基本一样
>
> 区别在于 `stable_sort()` 能够保证相等元素的相对位置，排序时不会改变相等元素的相对位置

使用用法和 `sort()` 一样，见上

stoi

```cpp
stoi(const string*)
```

> 将对应string类型字符串转换为数字，记忆： `s → t` 分别对应两个数据类型的某个字母

注意参数为 `string` 字符串类型。

关于输出数字的范围：
`stoi` **会做**范围检查，默认必须在 `int` 范围内，如果超出范围，会出现RE（Runtime Error）错误。
`atoi` **不做**范围检查，如果超出上界，输出上界，超出下界，输出下界。

```cpp
string s = "1234";
int a = atoi(s);
cout << a << "\n"; // 1234
```

transform

复杂度： $O(N)$

> 作用：使用给定操作，将结果写到dest中

```cpp
//原型：
transform(beg, end, dest, unaryOp);
```

一般不怎么使用，徒增记忆负担，不如手动实现。

```
//将序列开始地址beg到结束地址end大小写转换，把结果存到起始地址为dest的序列中
transform(beg, end, dest, ::tolower);
transform(beg, end, dest, ::toupper);
```

to_string

> 将数字转化为字符串，支持小数（double）

```
int a = 12345678;
cout << to_string(a) << '\n';
```

unique

```
unique(beg, end)
```

复杂度： $O(N)$

> 消除重复元素，返回消除完重复元素的下一个位置的地址
>
> 如：`a[] = {1, 3, 2, 3, 6}`；
>
> `unique` 之后 `a` 数组为 `{1, 2, 3, 6, 3}` 前面为无重复元素的数组，后面则是重复元素移到后面，返回 `a[4]` 位置的地址（不重复元素的尾后地址）

消除重复元素一般需要原序列是**有序序列**

应用：离散化

- 方法一：利用数组离散化

```
for(int i = 0; i < n; i++) {
    cin >> a[i];
    b[i] = a[i];//将a数组复制到b数组
}
// 排序后 b：{1, 2, 3, 3, 6}
sort(b, b + n);//对b数组排序
// 消除重复元素b：{1, 2, 3, 6, 3} 返回的地址为最后一个元素3的地址
int len = unique(b, b + n) - b;//消除 b 的重复元素，并获取长度
for(int i = 0; i < n; i++) {
    //因为b有序，查找到的下标就是对应的 相对大小（离散化后的值）
    int pos = lower_bound(b, b + len, a[i]) - b;//在b数组中二分查找第一个大于等于a[i]的下标
    a[i] = pos; // 离散化赋值
}
```

- 方法二：利用 `vector` 进行离散化

```
vector<int> a(n);
for (int i = 0; i < n; ++i) {
    cin >> a[i];
}
vector<int> b = a;
sort(b.begin(), b.end());
b.erase(unique(b.begin(), b.end()), b.end());
for (int i = 0; i < n; ++i) {
    a[i] = lower_bound(b.begin(), b.end(), a[i]) - b.begin() + 1; // 离散后的数据从1开始
}
```

__gcd

```
__gcd(a, b)
```

> 求a和b的最大公约数

`__gcd(12,15) = 3`

`__gcd(21,0) = 21`

__lg

```
__lg(a)
```

> 1. 求一个数二进制下最高位位于第几位（从**第0位**开始）（或二进制数下有几位）
> 2. `__lg(x)` 相当于返回$\lfloor log_2 x \rfloor$
> 3. 复杂度$O(1)$

`__lg(8) = 3`

`__lg(15) = 3`

_builtin 内置位运算函数

> 需要注意：内置函数有相应的 `unsigned lnt` 和 `unsigned long long` 版本，`unsigned long long` 只需要在函数名后面加上 `ll` 就可以了，比如 `__builtin_clzll(x)` ，默认是32位 `unsigned int`
>
> 很多题目和 `long long` 数据类型有关，如有需要注意添加 `ll`

__builtin_ffs

```
__builtin_ffs(x)
```

> 二进制中对应最后一位 `1` 的位数，比如 `4` 会返回 `3` （100）

__builtin_popcount

```
__builtin_popcount(x)
```

> `x` 中 `1` 的个数

__builtin_ctz

```
__builtin_ctz(x)
```

> `x` 末尾 `0` 的个数（`count tail zero`）

__builtin_clz

```
__builtin_clz(x)
```

> `x` 前导 `0` 的个数（`count leading zero`）

```
cout << __builtin_clz(32); // 26
//因为共有6位,默认数据范围为32位，32 - 6 = 26
```

__builtin_parity

```
__builtin_parity(x)
```

> `x` 中1的个数的奇偶性， 奇数输出 `1`，偶数输出 `0`

> 可参考链接：
>
> 1. **C++语法糖** https://www.luogu.com.cn/blog/AccRobin/grammar-candies

可能有些人需要PDF文件，公众号【行码棋】回复 【STL】 获取（三个大写英文字母即可），抱歉😭

> 2023.03.28 已更新PDF文件（除去了水印，内容进行了部分排版调整和更新）