版本号 : V 2.0

> 一. 基础知识 二. STL **常用容器**,
>
> 三. 算法模板 四. Modern C++
>
> 五 Effective C++

一. Basic

1.0 如何判断你的代码能不能在规定时间内通过:

- 机器处理的数据量为 1e8
- 因此，一般1e4 范围内的数据可以 O(n2), 1e5范围内的数据O(nlogn)

1.1 概念辨析

命令	解释
fun(x++)	输入到函数里面的是 **x**
fun(++x)	输入到函数里面的是 **x+1**
if(! flag)	当 **flag == 0**的时候执行
if(1== a)	写出if(1=a)就会报错, 少打了=
int n = 1e5	少写几个零
(a, b)中有	b-a+1 个数字
vector<int>的中位数	a[**n/2**] 或者 1/2 * (a[n/2] +a[**n/2 -1**])
max({a, b, c,,,})	abc,,,,中最大的数

1.2 各种初始化

有三种初始化方式, `()`, `=`, `{}`

- `()` 初始化, 当你想用默认初始化时, Weight(), 会声明一个函数
- `{}`: 统一初始化, 从**概念上**可以用于**一切场合**, 表达一切意思的初始化, 有个新特性, 禁止**内建型别**之间进行**隐式窄化**型别转化

```
// ----------------------初始化------------------------
// 列表初始化
vector<vector<int> > map = {{1, 0}, {-1, 0}, {0, 1}, {0, -1} };

// 多维数组的列表初始化, 和上面一样
int map[4][2] = {1, 0, -1, 0, 0, 1, 0, -1};
```

```
// 默认值的初始化，全部是 0
int a[26] ={};
int a[26]{};

// 前三个是 1，2，3，后面的值都是 0
int a[26] ={1，2，3};

// 大括号 初始化
vector<int> v{1,3,4};
```

1.3 巧用 引用`ref`

```
// ----------------------巧用引用-------------------------
void mysort(vector<int> &nums){
    // 如果要对数组元素进行改变，可以使用引用.
    for(int &x: nums){
        if(x% 2) x=-x;
    }
    sort(nums.begin()，nums.end());
}
```

1.4 数组的中位数

- 长度为`len`，奇数: 正中间, 偶数中间偏右
- 使用下标时, a[l, r] $\frac{l+r}{2}$ **中间靠左**

1.6 类型上下界

缩写	类型
INT_MAX INT_MIN	int 类型最大最小
UINT_MAX	unsigned 最大最小
LONG_MIN	
LLONG_MAX	long long最大最小
ULONG_MAX	

1.5 模板 访问类内部成员

```
// 访问类内部的类型时，一般而言，需要实例化以后才可以访问类内部的对象
template<typename T>
class Myclass{
    typename T::subtype *ptr;
}
// 含义是指向类内部类型的指针
```

二. 进阶_容器使用

自然**有序**的容器, 可以直接使用 a.find(), 例如

- set multiset
- map /multimap
- unordered_set/unordered_multiset
- unordered_map/unordered_multimap

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
array	数组	随机读改 O (1)	无序	可重复	支持**随机**访问
vector	数组	随机读改、尾部插入、尾部删除 O (1) 头部插入、头部删除 O (n)	无序	可重复	支持随机访问
deque	双端队列	头尾插入、头尾删除 O (1)	无序	可重复	一个中央控制器 + 多个缓冲区，支持首尾快速增删，支持随机访问
forward_list	单向链表	插入、删除 O (1)	无序	可重复	不支持随机访问
list	双向链表	插入、删除 O (1)	无序	可重复	不支持随机访问

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
stack	deque / list	顶部插入、顶部删除 O (1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
queue	deque / list	尾部插入、头部删除 O (1)	无序	可重复	deque 或 list 封闭头端开口，不用 vector 的原因应该是容量大小有限制，扩容耗时
priority_queue	vector + max-heap	插入、删除 O (log2n)	有序	可重复	vector 容器 + heap 处理规则
set	红黑树	插入、删除、查找 O (log2n)	有序	不可重复	
multiset	红黑树	插入、删除、查找 O (log2n)	有序	可重复	
map	红黑树	插入、删除、查找 O (log2n)	有序	不可重复	
multimap	红黑树	插入、删除、查找 O (log2n)	有序	可重复	
unordered_set	哈希表	插入、删除、查找 O (1) 最差 O (n)	无序	不可重复	
unordered_multiset	哈希表	插入、删除、查找 O (1) 最差 O (n)	无序	可重复	
unordered_map	哈希表	插入、删除、查找 O (1) 最差 O (n)	无序	不可重复	

容器	底层数据结构	时间复杂度	有无序	可不可重复	其他
unordered_multimap	哈希表	插入、删除、查找 O (1) 最差 O (n)	无序	可重复	

1 迭代器 iterator

1.1 常用迭代器操作

```
// 迭代器在容器内部
vector<int> :: iterator it;
// 可以使用解引用操作符*，来访问迭代器指向的元素。
*it;

// 常用的操作有：
++ -- == !=

// 只有顺序容器才可以适应it + n, <=,>=

// 左闭右开的好处，判断相等
first == last 代表空。

// 访问元素
it-> mem;


// 对两个迭代器进行相减操作，会计算它们之间的距离，
auto distance = it2 - it1;

// 如果迭代器所指向的容器不支持随机访问，例如 list 或 set，则不能使用迭代器相减操作，可以考虑使用distance
// 位于 <iterator> 头文件中
#include<iterator>
typename std::iterator_traits<InputIt>::difference_type
    distance( InputIt first, InputIt last );

// 如果在调用 std::distance 函数时，第一个迭代器的位置在第二个迭代器的后面，则返回的结果为负数
auto d = std::distance(it1+3, it1);

// 在计算迭代器之间的距离时，要确保这两个迭代器都指向同一个容器中的元素或者它们都是指向同一块可寻址内存空间的合法指针。否则，std::distance() 函数的行为是未定义的。


// 找到最大值所在的元素的迭代器，注意，it1 <= it2
#include<algorithm>
max_element(it1, it2);
min_element(it1, it2);
```

获取迭代器的地址

```
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << &(*it) << " ";
}
```

2 序列式容器

2.1 string

```
//四种基本的初始化，顺序容器的初始化，默认，复制，范围，n个值
string s1;
string s1(s2);
string s3("sdf");
string s4(n, 'c');

// 输入 string
string s4;
cin >> s4;

// 从开始位置复制到最后
string s5(s1, pos);
string s6(s1, pos, len);

// operator+=
s1+= s2;
s1+= 'c';

// 清空
s.clear();

// 用迭代器内元素替换
s.assign(it1, it2);
s.assign(n, value);

// 也可以直接使用下标进行修改
s[i] = '0';


// // 也没有 iterator 的形式
// 查找的类型，开始位置
size_t find (char c, size_t pos = 0) const;
size_t find (const string& str, size_t pos = 0) const;
size_t find (const char* s, size_t pos = 0) const;
// 开始位置，结束位置
size_t find (const char* s, size_t pos, size_t n) const;


// 截取子串，只有使用下标的形式
```

```
// 若超出字符串长度，则默认从 pos 到字符串末尾提取所有字符。
string substr (size_t pos = 0, size_t len = npos) const;

// 临时将string对象转换为C风格字符串。不能直接用于修改字符串内容。
const char* string::c_str();
```

```
//---------------------插入删除--------------
// 插入字符串，字符串的子串
string& insert (size_t pos, const string& str);
string& insert (size_t pos, const string& str, size_t subpos, size_t sublen);
string& insert (size_t pos, const char* s);
string& insert (size_t pos, const char* s, size_t n);

// 插入char 类型必须要有数量参数
string& insert (size_t pos, size_t n, char c);
void insert (iterator p, size_t n, char c);

iterator insert (iterator p, char c);

// 删除，默认是全部删除，单点删除只支持迭代器
string& erase (size_t pos = 0, size_t len = npos);
iterator erase (iterator p);
iterator erase (iterator first, iterator last);
```

```
// ----------------string与 数值类型---------
// 在头文件 string 中
// string => int
int stoi (const string& str, size_t* idx = 0, int base = 10);

// string => double
double stod(const string& str, size_t* idx = 0);

// 转换为 string
// int/ double float => double
string s1 = to_string(123);
string s2 = to_string(4.5);

//字符处理<cctype>
#include <cctype>
// 判断一个字符是否为字母或数字
isalnum(char);
isalpha(char);

isdigit(char);
islower();
isupper();
```

```
toupper(c);
tolower(c);
// 判断是否是字母等
isalpha 字母母（包括大大写、小小写）

isalnum（字母母大大写小小写+数字）
isblank（space和\t）
isspace（space、\t、\r、\n）
```

输入输出

```
// 输入到 string
string str;
cout << "请输入一个字符串：";
cin >> str; // 用户输入 "Hello, World!"，则 str 的值为 "Hello,"

// 当cin遇到空白字符时就停止，最标准的方法是调用getline(cin,str)函数。
string line;
getline(cin, line); // 用户输入 "This is a sentence."，则 line 的值为 "This is a
sentence."

// 从文件输入
ifstream fin("file.txt");
string str;
fin >> str;
```

2.2 vector

the link with capacity

- 如果两个`vector`相比较, 返回第一个不相同元素的 `<` 比较结果,
- 内部以连续的方式存放, 当没有空间存放时, 会重新分配空间, 原来的迭代器会失效(因为**地址**发生了改变), **插入会使该容器所有的迭代器失效**

```
//初始化
vector<int> v1;
vector<int> v2(v1);      //复制v1
vector<int> v3(n, value);   //n个值为value的元素
vector<int> v3(n);          //n个初始元素的副本。
vector<int> v4(a+1, a+3) //使用数组进行初始化，不包括最后一个地址的元素。

//a的大小
a.size() ;
//判断是否为空
v.empty() ;

//在容器的最后添加一个值为t的数据，容器的size变大。
v.push_back(t);
//删除容器的末尾元素，仅仅删除，没有返回。
a.pop_back() ;
```

```
//第一个元素，最后一个元素的引用（值）。
v.back();
v.front();

//----------------------大小有关的操作---------------------
//清空
v.clear();
//调整大小。
v.resize(n, t);
v.resize(n);
//删除，返回删除元素的下一个位置，也是左闭右开。
v.erase(it);
v.erase(it1, it2);

//插入it 前面，代表 成为它 ，原来序号是3，插入的新值序号也是3
//同时返回新元素的迭代器
//第一个版本，插入一个新的值
v.insert(it, value);
//第二个版本，插入n个新的值
v.insert(it, n, value);
//插入it前面，从it1 到it2 的元素
v.insert(it , it1, it2);

//第一个元素的地址，要把vector 和数组区分开。
&v[0];
```

```
// ----------------------多维vector----------
// 多维vector 只能添加 vector<int>
// 若想定义A = [[0,1,2],[3,4]]，有两种方法。
vector<vector<int> > A;
vector<int> B = {0,1,2};
vector<int> C = {3, 4};
A.push_back(B);
A.push_back(C);


for(int i = 0; i < 2; ++i) {
     A.push_back(vector<int>());
}
A[0].push_back(0);

// vector<vector<int> >A中的vector元素的个数
len = A.size();
// vector<vector<int> >A中第i个vector元素的长度
len = A[i].size();
```

避免因为插入导致的迭代器失效

```
//一个容器中，读取完元素后，再插入一个值
auto first = a.begin(), last = a.end()
while(first != a.end()){
    first = a.insert(++first, 666);
    ++first;
}
```

关系运算符 ------- 容器的比较是基于**容器内元素**的比较

- 当长度相同且元素相等, 则相等
- 比较的结果 取决于 **第一个不相等**的元素
- 当 `vector<vector>` 进行比较时, 空的 vector 被放到前面

2.3 list

```
// 因为是双向链表，所以 it 可以++, --
list<int> l{1, 2, 3, 4};
// 多个元素的值
list<pair<int, int> > l;

// 获取元素的值
l.front();
l.back();

// 头尾均可插入删除
l.push_front();
l.pop_front();

l.push_back();
l.pop_back();

// 插入
l.insert(it, val);
l.insert(it, n, val); //n 个 value
l.insert(it, it_first, it_last);

// delete
l.erase(it);
l.erase(it_first, it_last);

// 拼接
// 整个l2 拼接到 it1的位置
l1.splice(it1, l2)
// 将 l2 的 l2_pos 指向元素（节点）切除，拼接到 l1 的 l1_pos 处（l1 和 l2 可相同）
l1.splice (iterator l1_pos, list<T,Allocator>& l2, iterator l2_pos );
l1.splice(head, l1, it_2);
//将容器lt6的指定迭代器区间内的数据拼接到容器lt5的开头
lt5.splice(lt5.begin(), lt6, lt6.begin(), lt6.end());

// 删除重复元素
l.unique();
```

```
// 删除满足条件的元素 bool fun()
lt.remove_if(fun);
```

2.4 deque

```
// 初始化
deque<int> de(10, 666);

de.push_back(1);
de.push_front(1);
```

2.5 stack

```
stack<int> st;

// 判断是否为空
st.empty();

// 入栈
st.push(1);

// 返回栈顶元素
st.top()

// 出栈
st.pop();
```

2.6 queue

```
queue<int> qu;

// 进入队列
qu.push(a);

// 返回队头元素
qu.front();

// 队头的元素 出队
qu.pop();
```

2.7 priority_queue

实质: **堆**

```
// template <class T, class Container = vector<T>, class Compare = less<typename
Container::value_type> >
// class priority_queue;
// 默认是按小于 (less) 的方式比较，这种比较方式创建出来的就是 大顶堆。
// 就当作 比较的时候 先输入 子节点，然后输入 父节点
priority_queue<pair<int, int>, vector<pair<int, int>>, compare> heap;

// 初始化方式，使用迭代器初始化
(it1, it2);

// 和 stack 很像
// 添加元素
heap.push();

// 获得堆顶元素
heap.top()

// 删除元素
heap.pop();
```

2.8 bitset 位运算

与（AND）、或（OR）和异或（XOR）是逻辑运算符，在计算机编程中有一些特殊性：

- 与和或 亦或 运算都是**可结合的和可交换**的，即表达式中多个操作数的**顺序**不影响最终结果。

异或（XOR）的特殊性：**相同为假**, 不同为真

- 异或运算可以用于检测两个值是否不同，如果两个值不同，则结果为真。
- 在编程中，异或运算常用于交换两个值的变量，而无需引入第三个中间变量。

- `1`的特殊性: 二进制形式如 `000000001`, 只有末尾一个`1`
- 编程时，请注意运算符的**优先级。**例如 `==` 在某些语言中**优先级**更高, **位运算需要加括号**

```
// 将x左移 n位，补零
x << n;

// 判断 第 d 位是否为 1
// 1 的特殊性，只有最低位是 1
bit =  0b11001;
// 右移也是补 0
bit >>d &1 ;
```

```
// 按位与操作符的返回值是一个新的二进制数
// 每一位都是两个操作数相应位执行**逻辑操作**的结果

// 一个有 n 位数字的集合如何表示全集
(1<<n)-1  ;

// 遍历集合
for (int i = 0; i < n; i++) {
    if ((s >> i) & 1) { // i 在 s 中
        // 处理 i 的逻辑
    }
}

// 枚举 从空集到 全集
for (int s = 0; s < (1 << n); s++) {
    // 处理 s 的逻辑
}

// 设集合为 s，从大到小枚举 s 的所有非空子集 sub
// 暴力做法是从 s 出发，不断减一直到 0，但这样中途会遇到很多并不是s 的子集的情况。
for (int sub = s; sub; sub = (sub - 1) & s) {
    // 处理 sub 的逻辑
}
```

```
#include "biset"
// 和一般的容器不一样，<>中间是多少个二进制位数，5表示5个二二进位
// 默认在高位补零，下面就是 "00011"
bitset<5> b("11");

// 如果用较大的整数向较小的 bitset 对象赋值，也会发生同样的截断错误，不会产生警告或异常，并且会将高位丢弃
bitset<3> (16) ;


// 从整数创建
std::bitset<8> b(n);
// 高位置截断
string str = bs.to_string().substr(bs.to_string().find('1'));

bitset<5> b; 都为0
bitset<5> b(u); u为unsigned int，如果u = 1,则被初始化为10000
bitset<5> b(s); // s为字符串串，如"1101" -> "10110"
bitset<5> b(s, pos, n); // 从字符串串的s[pos]开始，n位长长度

// 使用下标，注意，重要区别
// b[0] 访问的是 b 的最右边的一位，即二进制数的最低位
// 访问 std::bitset 对象的元素时，如果所访问的元素的值为 0，则索引操作符返回 false；如果所访问的元素的值为 1，则索引操作符返回 true。

// 可以直接输出
```

```
cout << bitset<8>(15);

// b中二二进制位的个数
 b.size();
// 统计1的位数
int count = b.count();

// 0 1操作
//把b的下标为4处置1
b.set(4);

//所有位归零
b.reset();
b.reset(3); //b的下标3处归零

// 操作符, 必须位数相同才能操作
std::bitset<8> b3 = b1 & b2;   // 位与
std::bitset<8> b4 = b1 | b2;   // 位或
std::bitset<8> b5 = b1 ^ b2;   // 位异或
std::bitset<8> b6 = b1 << 2;   // 左移 2 位
std::bitset<8> b7 = b1 >> 2;   // 右移 2 位



// 所有位都为真
cout << endl << b.any(); //b中是否存在1的二二进制位
// 所有位都为假
cout << endl << b.none(); //b中不不存在1吗?
cout << endl << b.count(); //b中1的二二进制位的个数

cout << endl << b.test(2); //测试下标为2处是否二二进制位为1

b.flip(); //b的所有二二进制位逐位取反
unsigned long a = b.to_ulong(); //b转换为unsigned long类型
```

3 关联式容器

3.1 pair

```
typedef pair<int, int> mypair;

// map 中 key 类型是 const 类型
map<int, int> mp;
for(auto & node: mp){
    // 实际要写成
    pair<const int, int> node:
}

pair<int, int> p;
// 只有两个成员
```

```
p.first;
p.second;
```

3.2 map

map/multimap属于 关联式 容器，底层结构是用 二叉树(红黑树) 实现, 时间复杂度为O(logn)。

```
a.size(); //返回容器中元素的数目
a.empty(); //判断容器是否为空
swap(st); //交换两个集合容器

a.count(key); //统计key的元素个数

// 清除元素
void erase (iterator position);
size_type erase (const key_type& k);
void erase (iterator first, iterator last);

void printMap(map<int,int>&m){
    for (map<int, int>::iterator it = m.begin(); it != m.end(); it++){
        cout << "key = " << it->first << " value = " << it->second << endl;
    }
    cout << endl;
}
```

3.1 unordered_map

基于哈希表, 空间大, 时间复杂度不稳定,平均为常数级O(c), 取决于哈希函数,极端情况下为O(n)

- `unordered_map<pair<int, int>, int> mp` 会报错, 因为没有给 `pair` 做 Hash 函数
- `map` 里面是通过操作符 `<` 来比较大小，而 pair 是可以比较大小的。

```
// 注意：C++11才开始支持括号初始化
unordered_map<int, string> mp={{ 1, "张三" },{ 2, "李四" }};

// 使用[ ]进行单个插入，若已存在键值，则赋值修改，若无则 插入。
mp[2] = "李四";    //不会插入

// 使用键值 删除
size_type erase ( const key_type& k );

//    使用insert和pair插入, 麻烦
mp.insert(pair<int, string>(3, "王二"));

//遍历输出+迭代器的使用
auto iter = mp.begin();//auto自动识别为迭代器类型unordered_map<int,string>::iterator
while (iter!= myMap.end()){
    cout << iter->first << "," << iter->second << endl;
```

```
    ++iter;
}

unordered_map<Key,T>::iterator it;
it->first;               // same as (*it).first   (the key value)
it->second;              // same as (*it).second  (the mapped value)
```

3.3 set

```
// 定义一一个空集合s
set<int> s;

// 增
s.insert(1);

// 删除集合s中的1这个元素
s.erase(1);

// 查
s.count(1);

// s.find() 返回迭代器
// 根据 STL 前毕后开的特点 如果结果等于s.end()表示未找到
s.find(2) != s.end()) << endl;
```

3.4 multiset

```
multiset<int> set;
// 使用键值删除会全部删除
st.erase (40);      // [10 30 40 40 50 60]  ->  [10 30 50 60]
```

4 算法库 & 常用库

4.1 sort 函数

```
# include <algorithm>

// 默认，v从小小到大大排列，并且是前闭后开
sort(v.begin(), v.end());
```

```
//cmp函数返回的值是bool类型
bool cmp(int a, int b) {
    return a > b; // 从大大到小小排列列
}

// 有时候这种简单的if-else语句句我喜欢直接用用一一个C语言言里里里面面的三目目运算符表示～
bool cmp(stu a, stu b) {
    return a.score != b.score ? a.score > b.score : a.number < b.number;
}

// 也可以使用 lambda 表达式
sort(envelopes.begin(), envelopes.end(), [](const auto& e1, const auto& e2) {
    return e1[0] < e2[0] || (e1[0] == e2[0] && e1[1] > e2[1]);
});
```

4.2 max & min & max_element() & min_element

```
// max min 四 个版本
template <class T> const T& max (const T& a, const T& b);
template <class T, class Compare>  const T& max (const T& a, const T& b, Compare
comp);
template <class T> T max (initializer_list<T> il);
max({3, 4, 5});
template <class T, class Compare>  T max (initializer_list<T> il, Compare comp);
```

```
// 找最大最小值，返回的是迭代器
template <class ForwardIterator>  ForwardIterator max_element (ForwardIterator
first, ForwardIterator last);

template <class ForwardIterator, class Compare>  ForwardIterator max_element
(ForwardIterator first, ForwardIterator last, Compare comp);
```

4.5 lower_bound()

```
// 二分查找最低位
It lower_bound(ForwardIt first, ForwardIt last, const T& value);

// 重载形式
ForwardIt lower_bound(ForwardIt first, ForwardIt last, const T& value, Compare
comp);
```

4.3 累加accumulate

- 使用时注意初值
- `accumulate(nums2.begin(), nums2.end(), 0LL)` 中的 `0LL` 表示将初始值设为 long long 类型的 0 值，而 `accumulate(nums2.begin(), nums2.end(), 0)` 中的 `0` 表示将初始值设为 int 类型的 0 值。会有**溢出**风险

```
// 默认相加的操作，最后一个参数为初始值，必须有初始值
accumulate(it1, it2, init_value);

// 自定义运算
std::string str1 = "hello";
std::string str2 = "world";

int sum = std::accumulate(str1.begin(), str1.end(), 0,
   [=](int acc, char c) {return acc + c + str2[c - 'a'];});
```

```
template<class InputIt, class T>
T accumulate(InputIt first, InputIt last, T init) {
    for(; first != last; ++first) {
        init = init + *first;
    }
    return init;
}

//
template<class InputIt, class T, class BinaryOp>
T accumulate(InputIt first, InputIt last, T init, BinaryOp op) {
    for(; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}
```

unique

- 让区间内的元素**唯一**
- 使用之前先排序
- 只能移除相邻的重复元素，并将重复元素移动到区间的末尾，然后返回指向新的区间尾部

```
// 返回最后一个唯一元素
it unique(nums.begin(), nums.end());
```

4.2 常用算法_find fill for_each

```
//查找元素，find返回迭代器，如果没找到，返回it2，因此，检查返回值和it2 是否相等可以看出是否找到.
find(it1, it2 , value);

//将value的副本写入指定的范围，只对输入范围内部的元素进行写入操作
fill(it1, it2, value);

//对于每一个元素都执行的操作
for_each(it1, it2, func)

for_each(a.begin(), a.end(),[&](int x){cout<< x<< endl;});
```

4.4 copy()

```
// copy() 函数定义在 algorithm 头文件中。
template <typename InputIt, typename OutputIt>
OutputIt copy(InputIt first, InputIt last, OutputIt dest) {
    while (first != last) {
        *dest++ = *first++;
    }
    return dest;
}
```

5 流类

- 在 C++ 标准库中，`stream` 是一个**基类**，其定义了输入流和输出流的公共接口，派生出了 `istream` 和 `ostream` 两个类（以及它们的多个子类），分别用于处理输入和输出。以下是 `stream` 类的主要作用：
    - 提供输入和输出的抽象层：`stream` 类为所有输入和输出提供了一个抽象接口，使得使用者不需要考虑底层实现（如文件、终端、网络等），只需要通过标准化的接口来进行数据读写即可。
    - 支持数据类型和格式化的转换：`stream` 类可以对数据类型进行输入和输出，支持诸如二进制数据和文本数据之间的转换，还能够自动进行格式化输出，比如指定十六进制、八进制、科学计数法等方式。

- 支持输入和输出的控制：`stream` 类提供了一些控制输出行为的接口，例如设置输出精度、输出格式、对齐方式等，同时也提供了一些控制输入行为的接口，例如跳过空白字符、设置输入结束符等。
- 支持流状态检测和错误处理：`stream` 类提供了方法来检测和处理不良或错误的输入输出操作，例如检测文件读取是否到达文件结尾、是否输入的数据类型与期望类型不匹配等，还可以设置 `stream` 对象的错误状态并提供相应的错误处理方法。

5.1 输出整个容器

```
// cout:a global object of ostream
// typedef basic_ostream<char>ostream

template <typename T>
std::ostream & operator<<(std::ostream& os, const std::vector<T> & vec) {
    os << "[";
    for (size_t i = 0; i < vec.size(); ++i) {
        os << vec[i];
        if (i != vec.size() - 1) {
            os << ", ";
        }
    }
    os << "]";
    return os;
}


// 以下会有问题
template<typename T>
std::ostream& operator<<(std::ostream& os, T& c) {
    os << '[';
    for (auto it = c.begin(); it != c.end(); ++it) {
        os << (*it);
        if (std::next(it) != c.end()) {
            os << ',';
        }
    }
    os << ']';
    return os;
}


template<typename T>
std::ostream& operator<<(std::ostream& os, const T& c) {
    os << string("[");
    for (auto it = c.begin(); it != c.end(); ++it) {
        os << (*it);
        if (std::next(it) != c.end()) {
            os << ',';
        }
    }
   os << string("]");;
    return os;
}
```

ofstream

- open ()：打开一个文件，并指定打开模式；
- close ()：关闭当前打开的文件；
- write ()：从指定的缓冲区写入指定字节数的数据到文件中；
- put ()：往文件中写入一个字符；
- <<：重载了输出运算符，可以用于像 cout 一样方便地输出数据到文件中。

```
// ofstream,  写文件的输出流类
// <<: 重载了输出运算符，可以用于像 cout 一样方便地输出数据到文件中。
ofstream of("MyLog.txt");// Creates a new file for write,if the file didn't exist
of <<"Experience is the mother of wisdom"<<endl;
of<<234<<end1;
of<<2.3<<end1;

// 将输出点放在文件末尾
ofstream of("Mylog.txt", ofstream::app);

ofstream of("Mylog.txt", ofstream::in | ofstream::out);
// 将光标移至开始以后的 10个字符
of.seekp(10, ios::beg);

of.seekp(-5, ios::end);
of.seekp(-5, ios::cur);

ifstream inf("Mylog.txt");
int i;
// 读取一个字符
inf>> i;
inf.good(); // goodbit == 1;
inf.bad();  // badbit==1
inf.fail(); // failbit ==1, badbit == 1
inf.eof();  // eofbit ==1;

// 清除所有的 错误状态
inf.clear();
inf.clear(ios::badbit);

if(inf) ;// => if(!inf.fail())
if(inf>>i);

inf.exception(ios::badbit| ios::failbit);

inf.close(); // 关闭文件
```

三. 进阶_ 算法模板

1 数值处理

1.1 数值处理-----反转数字

```
for(int num : nums){
    int re =0;
    while(num){
        re = re*10 + num% 10;
        num/=10;
    }
    cout<< re<< endl;
}
```

1.2 数值处理------最大公因子

```
// 在algorithm中, 有__gcd()函数
int gcd(int a, int b){
    // 默认 b 为较小的
    int r;
    while (b) {
        r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

位运算

位运算符（也称为按位运算符）是用来操作操作数的二进制位的。

- 按位与（&）：将两个操作数的每一位进行比较，如果两个操作数在同一位上都是 1，则结果为 1，否则为 0。
- 按位或（|）：将两个操作数的每一位进行比较，如果两个操作数在同一位上都是 0，则结果为 0，否则为 1。
- 按位**异或**（^）：将两个操作数的每一位进行比较，如果两个操作数在同一位上**相同**，**则结果为 0**，否则为 1。
- 按位取反（~）：将操作数的每一位都取反（0 变成 1, 1 变成 0）。
- 左移（<<）：将左侧操作数的所有二进制位向左移动指定的位数，右侧的空位用 0 补充。
- 右移（>>）：将左侧操作数的所有二进制位向右移动指定的位数，左侧的空位用符号位（对于有符号类型）或 0（对于无符号类型）补充。

按位亦或

- 一个数和 0 做 XOR 运算等于**本身**：a^0 = a

- 一个数和其本身做 XOR 运算等于 0：a^a = 0
- XOR 运算满足**交换律和结合律**：a^b^a = (a^a)^b = 0^b = b

1.3 数值处理------求幂

分为奇数和偶数

1.4 数值处理------求模运算

自然数取余定义分为两种：

- 定义 1：如果 a 和 d 是两个自然数，d 非零，可以证明存在两个唯一的整数 q 和 r，满足 `a=qd+r且 0 ≤ r < d`（其中 q 为商，r 为余数）。
  定义 1 一般作为数学中的取余法则，即两个数取余，余数总是为**正数。**

```
// C++中 负数 % n 并不做处理
// 在 C++ 中，对于负数 a 和正整数 n 进行模运算（求余数），其结果的符号与被除数 a 的符号相同。具体地说，当 a <0 时，a % n 的结果为负数，当 a>= 0 时，a % n 的结果为非负数（即自然数）
// 在 C++ 中，对于除数为负数的情况，取模运算结果未定义。因此，应该避免除数为负数的情况。
cout<< (-1) %3<<endl; // -1

// 两数相加再取模
(m + n) % p = (m%p + n%p) %p

// 两数相乘再取模
(m * n) % p = (m%p) * (n%p) %p

// 两数相减再取模
(m - n) % p = ((m%p - n%p) + p) %p
```

1.5 数值处理-------模幂运算

```
// 先对 a^b % c
// 其中，b是数组 例如[1,2] 代表 12
```

字符串数值处理 数位DP

```
// 找到所有 小于该数值的字符串表示
unordered_map<int, int> mp;

function<int(int, bool )>func = [](int start, bool islimit)-> int{
    // wrong case
    if(start == len)// base case

    // 当前的唯一状态表示，要有 start等信息
    int cur;
```

```
    if(!islimit && mp.count(cur)){
        return mp[cur];
    }

    int up  = islimit? s[start]-'0': 9;

    int res = 0;
    for(int i = 0; i<= up; ++i){
        // 只有当前位置有限制，且等于 up时，才会 有限制
        res += func(start, islimit && i == up)
    }
    mp[cur] = res;
    return res;
}
```

排序技巧

```
// 添加元素所对应的坐标，然后进行自己的排序
vector<vector<int> > nums;

for(int i = 0; i < m; i++)
   nums[i].push_back(i);
sort(nums.begin(), nums.end(), cmp);


// 两个 vector 排序
// 用来比较两个序列中从前往后逐个元素进行比较，直到出现不同为止。
// 如果全部元素都相等，则比较两个序列长度，长度较短的序列小于长度较长的序列。
vector<int> v1 = {1, 2, 3,};
vector<int> v2 = {1, 2, 3, 5};
// v1 <v2

// vector<pair<>> 排序
// 默认按照 pair::first的大小
```

2 回溯 = (先根)DFS + 剪枝

```
res = []
# can是选择列表
def backtrack(res, track, can):
    if 满足结束条件:
        res.add(track)
        return

    # 时间复杂度O(n^n)
    for i in can:
        if(满足剪枝条件):
```

```
            #判断是否需要剪枝，也就是将不符合题意的循环删除
            continue

        #在递归之前做出选择，在递归之后撤销刚才的选择
        track.push_back(i)
        backtrack(res, track, can)
        track.pop_back(i)
```

BFS

- 重点在如何表示状态, 以及状态的转变
- 状态的组成必须**唯一**, 不能有状态是相同的
- 状态的组成必须**完备**
-

```
// encode the state

typedef <> State
queue<State> qu;

// 防止走回头路，可以使用 unordered_set / set/ vector / 数组
unordered_set<State> visit;
qu.push(start);
visit.insert(start);

int step = 0;
while(qu.size()){
    int len = qu.size();
    for(int i =0; i<len; ++i){
        State cur = qu.front();
        qu.pop();

        // 处理，进行每一步的操作
        if(finished){
            return step;
        }

        // k 是一共有多少种下一步的操作
        for(int j =0; j<k; ++j){
            State next ;
            qu.push(next);
            visit.insert(next);
        }
    }
    step++;
}
```

3 单调栈

```
// ------------------从后向前计算------------------
// 找到该元素 右边 第一个 比自己大的元素
len = nums.size();
vector<int> ans(len, -1);
stack<int> st;
//从后向前遍历, 小(等于) 当前元素的出去
for(int i =len - 1; i> -1; --i){
    while(!st.empty() &&  nums[i] >= st.top()){
        st.pop();
    }
    ans[i] = !st.empty()? st.top() : -1;
    st.push(nums[i]);
}


// ------------------从前往后计算-----------------
// 找到该元素 右边 第一个 比自己大的元素
stack<int> st; // 递增栈(从栈头到栈底的顺序)
vector<int> result(nums.size(), 0);
for (int i = 0; i < nums.size(); i++) {
    while (st.size() && nums[i] > nums[st.top()]) { // 注意栈不能为空
        // 当前元素 是 已遍历元素的 解
        result[st.top()] = i;
        st.pop();
    }
    st.push(i);
}
```

双指针 和 KMP

滑动窗口的**右端点**一定会到达**答案的右端点**，这时候左端点就会收缩到答案的左端点了

KMP

`KMP算法之 next 数组 (前缀表)

- 前缀表会告诉你下一步匹配中，模式串应该**跳到哪个位置。**
- 实质是: 记录下标i 之前（包括i）的字符串中，有多大长度的相同**前缀后缀**, 类似于`abcabc`这种
- **前缀**是指 不包含 最后一个字符的所有以第一个字符开头的连续子串；**后缀**是指 不包含 第一个字符的所有以最后一个字符结尾的连续子串。

- next[i] 的值**含义**为, 匹配的最长的前缀 实际能匹配到的**下标**
- next [i] == -1 时, 要么代表**无法匹配**(i == 0), 要么代表 **没有相同的前后缀**
- 如何快速求解 next [i], 直接用上一步求解的 next[i-1], `j=next[i-1]`, 如果`p[j+1]== p[i]`, `next[i] = j+1` 否则, `j=next[j]`, 什么意思呢, 再看看

```
    void getnext(vector<int> &next, const string &t){
        // next[i] 的值含义为, 实际能匹配到的下标
        // next [i] == -1 时, 要么代表无法匹配(i == 0), 要么代表 没有相同的前后缀
        int len = t.size();
        int j =-1;
        for(int i = 1; i<len; ++i){
            // 每次开始循环, j 代表了 p[i-1] 最多匹配到的下标
            while(j>-1 && t[i] != t[j+1] ){
                j = next[j];
            }
            if(t[i] == t[j+1]){
                ++j;
            }
            next[i] = j;
        }
    }
```

图论

4.1 图的遍历(使用邻接矩阵)

```
// 图遍历框架
// 图的遍历和一般 回溯 框架还是有所不同的, 因为传到traverse 函数中的点还没有进行过遍历, 所以衍生出了两种方法:
// - 丢进函数之前 对其进行处理
// - 在for循环之前进行处理, for循环中不进行处理( 更一般 )

void traverse(vector<vector<int> > &graph, int s) {
    if (visited[s]) return;
    // 经过节点 s
    visited[s] = true;
    for (TreeNode neighbor : graph.neighbors(s))
        traverse(neighbor);
    // 离开节点 s
    visited[s] = false;
}
```

图遍历过程中, 有些节点被访问了两次(遍历时, visit[i] == 1)的情况:

- 图中有环
- 某节点 的 入度 大于等于 2

要 注意 区分这两种情况

4.2 二分图

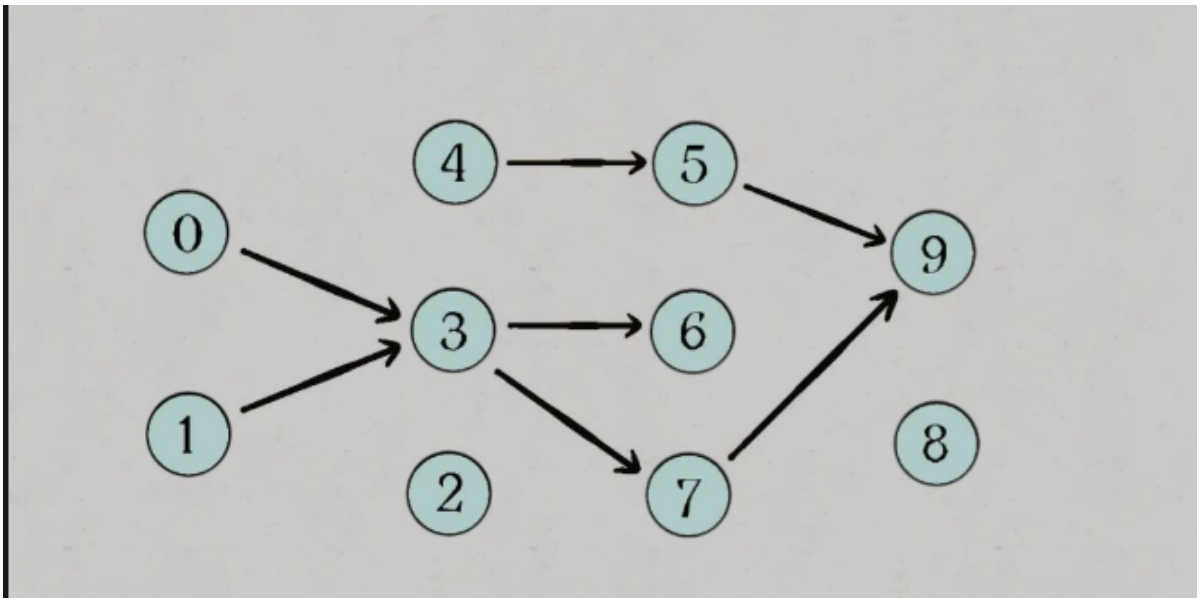
> 定义
>
> 二分图的顶点集可分割为两个互不相交的子集，图中每条边依附的两个顶点都分属于这两个子集，且两个子集内的顶点不相邻。
>
> 二分图**不是 "回溯"**

```
//遍历一遍图，并且一边遍历一遍染色，看看能不能用两种颜色给所有节点染色，且相邻节点的颜色都 不相同。

bool isBipartite(vector<vector<int>>& graph) {
    // the number of nodes
    int len = graph.size();

    vector<int> visit(len, 0);
    vector<int> color(len, 0);
    int res = 1;
    // 防止有单独的子图
    for(int i=0; i<len; ++i){
        if(visit[i] == 0)
            //深度遍历, 进行染色, start染c, 与start相连的点染 -c
            dfs(graph, i, visit, color, res, 1);
    }

    return res;
}

void dfs(vector<vector<int> > &g, int start, vector<int> &visit, vector<int>
&color, int &res , int c){
    if(!res) return;

    // if(visit[v] == 1) return;
    // 没有遍历过(染色)
    if(visit[start] == 0){
        visit[start] = 1;
        color[start] = c;
        cout<< "node "<< start<< "=\t "<< c<< endl;
        for(int nextnode : g[start]){
            dfs(g, nextnode, visit, color, res, -c);
        }

    }
    else if(color[start] == c) return;
    else {
        res = 0;
        //cout<< "node"<< start<< "\t is unsuccessful"<< endl;
    }
}
```

4.3 判断是否有环

```
bool hascycle(vector<vector<int>>& g) {
    // 检测是不是拓扑序列  也就是 检查 是否有环
    // 使用回溯法, 如果访问的节点在path中, 那么就是有环
    // 为什么不使用 visited, 因为visited是全局的,
    int len = g.size();

    int res = 1;

    vector<int> visited(n, 0), path(n, 0);
    for(int i = 0; i<n; ++i){
        dfs(res, g, visited, path, i);
        if(!res) return false;
    }

    return true;
}

void dfs(int &res, vector<vector<int> > &g, vector<int> &visited, vector<int>
path, int start){
    if(!res) return;

    // [----------------------bug is here--------------------]
    // 注意这两者的顺序, 先判断是否 在路径上, 因为在路径上一定浏览过,
    if(path[start] ==1 ) res = 0;
    if(visited[start]) return;
    path[start] =1;
    visited[start] = 1;
    for(int next : g[start]){
        dfs(res, g, visited, path, next);
    }
    path[start] = 0;
}
```

4.4 拓扑排序




```
vector<int> findOrder(vector<vector<int> > g ) {
    // 返回 拓扑排序
    // 结论 将后序遍历的结果进行反转（逆后序遍历顺序），就是拓扑排序的结果。
    int len = g.size();

    // 假设我们当前搜索到了节点 u，如果它的所有相邻节点都已经搜索完成，那么这些节点都已经在栈中了，此时我们就可以把 u 入栈
    vector<int> res, trace, path(numCourses, 0), visited(numCourses, 0);
    int flag = 1;

    for(int i =0; i< numCourses; ++i){
        // start 是要处理的点
        // cout<< start<<
        if(!visited[i]) dfs(flag, trace, g, visited, path, i);
        if(!flag) return {};
    }
    // 最后反转一下
    reverse(trace.begin(), trace.end());
    return trace;


}

void dfs(int &flag, vector<int> &trace, vector<vector<int> > &g, vector<int> &visited, vector<int> &path, int start){
    if(!flag) {return;}

    if(path[start]) {
        flag = 0;
        return;
    }
    if(visited[start]) return;

    path[start]=1;
    visited[start] = 1;
```

```
    for(int next: g[start]){
        dfs(flag, trace, g, visited, path, next);
    }
    // 到了尽头才添加，后序遍历
    trace.push_back(start);
    path[start] =0;

}
```

5.1 双指针------二分查找

```
int searchInsert(vector<int>& nums, int target) {
    int left =0, right = nums.size()-1;

    while(left <= right){
        int mid = left+ (right - left )/2;
        // 相等的时候向左收缩边界，也就是修改 r
        if(nums[mid] >= target) right=mid-1;
        else if(nums[mid] < target) left =mid+1;
    }
    return left;
}
```

5 双指针------滑动窗口

O(n) 时间内解决 **子串, 子数组**问题

- 窗口其实就是 **[left, right)**, 窗口大小是 **right - left**
- right 向右寻找可行解，left 向右寻找最优解
- right 表示**待处理**的节点
- 等价于枚举**左端点**

```
#-----------------V1----------------
while right < s.size():
    # 处理s[right]
    ++right;

    while( shirink && l<r ):
        # 满足 题目 条件的在这里更新
        updata res
        # 处理left
        ++left

#-----------------V2----------------
while right < s.size():
    # 处理s[right]
    ++right;

    while(!shirink  && l<r ):
        # 处理left
```

```
        ++left
    #不满足条件的在这里更新
    updata res
```

高级数据结构

前缀和 与 差分

- 我们可以通过如下方式构造其前缀和数组 s：`s[1] = a[1], s[i] = s[i-1] + a[i] (2 ≤ i ≤ n)`
- 我们可以通过如下方式构造其差分数组 d：`d[1] = a[1], d[i] = a[i] - a[i-1] (2 ≤ i ≤ n)`，
- 原始数组 `a[i]=` $\sum(d_i)$, 所以差分是前缀和的逆运算
- 如果想对原数组 `[l, r]` 内的元素加 `c`, 只需要对差分数组以下操作
  - `d[l]+=c` 以及 `d[r+1] -= c`
  - 开辟数组的时候, 要多开辟一位

线段树

- **区间更新，区间查询**，只能用线段树，前缀和做不到
- 通过空间换取时间, 尤其是**多次修改**的时候, 实质是利用 `未改变` 区间的结果
- 统一记录 p 所对应区间的修改
- 根节点为使用下标 `1`, 编号为k的节点的左儿子编号为2k，右儿子编号为2k+1，父节点编号为 `floor(k/2)`，用位运算优化一下，以上的节点编号就变成了 `k<<1`, `k<<1|1`
- **动态开点**权值线段树的合并代替数组的合并

```
class SegTree{
    int len;
    vector<long long> tree, tag;
    vector<int>  nums;
public:
    // 初始化
    SegTree(vector<int> &nums_){
        len = nums_.size();
        // 创建线段树, 区间长度为nums 的四倍
        tree = vector<long long>(len * 4, 0);
        tag = vector<long long>(len * 4, 0);
        nums=nums_;
        // 创建树的具体工作交给 build();
        build(1, 0, len-1);
    }

    //---------------------------------//
    // 线段树的 function, 求和, 最值,,,
```

```
void push_up(long long p){
    tree[p] = tree[p<<1]+ tree[p<<1 |1];
}

// 向线段树中填写值
void build(long long p , int start , int end ){
    // base case
    if(start == end){
        tree[p] = nums[start];
        return ;
    }

    long long  mid = (start + end)/2;
    build(p<<1, start, mid );
    build(p<<1|1, mid+1, end);

    push_up(p);

}

// 区间查询
long long  query(int l, int r, long long p , int start , int end ){
    // 查询区间为原始数组的[l, r]
    if(l<= start && r>= end){
        return tree[p];
    }
    long long mid = (start + end)/2;
    long long res = 0;
    if(l<= mid){
        res +=query(l, r, p *2, start, mid);
    }
    if(r>mid){
        res += query(l, r, p*2 +1, mid+1, end);
    }
    return res;
}


//-----------------------------------//
// 给p 打上标记, 并且更新 该树的值
void addtag(int t, long long p, int start, int end ){
    tag[p] = (1-tag[p]);
    tree[p] = end - start + 1 - tree[p];
}

// 将tag 分裂为两个
void downtag(long long p , int start, int end){
    if(tag[p] ){
        long long m = (end- start)/2 + start;
        addtag(tag[p], p<<1, start, m);
        addtag(tag[p], p<<1 | 1, m+1, end);
        tag[p] = 0;
    }
}

// 带有标记的区间修改(更新), 需要将改变向上传递
```

```
    void update(int l, int r, int t, long long p, int start, int end ){
        if(l<= start && r>= end){
            addtag(t, p, start, end);
            return ;
        }

        // cout<< "branch here "<< p << endl;
        downtag(p, start, end);
        long long  m = (end- start)/2 + start;
        if(m>= l){
           update(l, r, t, p<<1, start, m);
        }
        if(m<r){
            update(l, r, t, p<<1|1, m+1, end);
        }
        push_up(p);
    }

    friend ostream& operator<<(ostream &os, SegTree &seg){
        os<< '[';
        for(long long  i = 1; i< (long long )seg.len * 4; ++i){
            os<< seg.tree[i]<< '\t';
        }
        return os;
    }

};
```

```
// 存储该节点所代表的区间范围、左右儿子的指针以及 所求的结果
struct Node {
    int l, r;
    long long sum, maxval;
    Node *left, *right;
    Node(int l = 0, int r = 0) : l(l), r(r), sum(0), maxval(0), left(nullptr),
right(nullptr) {}
    ~Node() {
        delete left;
        delete right;
    }
    void pushup() {
        sum = (left ? left->sum : 0) + (right ? right->sum : 0);
        maxval = max(left ? left->maxval : 0, right ? right->maxval : 0);
    }
};

class SegmentTree {
public:
    SegmentTree(int n) : size(n), root(new Node(1, n)) {}
    ~SegmentTree() {
        delete root;
    }
    void update(int pos, int val) {
        update(root, pos, val);
```

```
    }
    long long query(int ql, int qr) {
        return query(root, ql, qr);
    }
private:
    int size;
    Node* root;
    void update(Node* node, int pos, int val) {
        if (node->l == node->r) {
            node->sum += val;
            node->maxval += val;
            return;
        }
        int mid = (node->l + node->r) / 2;
        if (pos <= mid) {
            if (!node->left) node->left = new Node(node->l, mid);
            update(node->left, pos, val);
        } else {
            if (!node->right) node->right = new Node(mid + 1, node->r);
            update(node->right, pos, val);
        }
        node->pushup();
    }
    long long query(Node* node, int ql, int qr) {
        if (ql <= node->l && qr >= node->r) return node->sum;
        int mid = (node->l + node->r) / 2;
        long long res = 0;
        if (ql <= mid && node->left) res += query(node->left, ql, qr);
        if (qr > mid && node->right) res += query(node->right, ql, qr);
        return res;
    }
};
```

sort（十大基础排序)

概论

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	O(n^2)	O3	Om3	OG)	In-place	稳定
选择排序	O(n^2)	Om	Om3	OG)	In-place	不稳定
插入排序	O(n^2)		Om	O()	In-place	稳定

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
希尔排序	O(n log n)	OmlognOmlogn)	OG)	In-place	不稳定	
归并排序	O(n log n)	Omlogn	Om logn)	0	Out place	稳定
快速排序	O(n log n)	Om logn)		Olon n	In-place	不稳定
推排序	O(n log n)	Omlogn)	Om logn)	OC)	In-place	不稳定
计数排序	O(n+k)	Om+)	O+z	O元)	Out-place	稳定
桶排序	O(n+k)	Om+)	Om	Om+z)	Out-place	稳定
基数排序	O(n* k)	O(n* k)	Omxz)	Om+Z)	Out-place	稳定

关于时间复杂度:

1. 1.

   平方阶 (O (n2)) 排序

   各类简单排序：直接插入、直接选择和冒泡排序。

2. 2.

   线性对数阶 (O (nlog2n)) 排序

   快速排序、堆排序和归并排序；

3. 3.

   O (n1+§)) 排序，§ 是介于 0 和 1 之间的常数。

   希尔排序

4. 4.

   线性阶 (O (n)) 排序

   基数排序，此外还有桶、箱排序。

插入排序------打扑克

- 将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。
- 从头到尾依次扫描未排序序列，将扫描到的每个元素**插入有序序列的适当位置。**（如果待插入的元素与有序序列中的某个元素相等，则将待插入元素插入到相等元素的后面。

```
int lowwer_bound(vector<int> &nums, int end, int target){
    int l = 0;
    int r = end;
    while(l<=r){
        int mid = (l+r)/2;
        if(a[mid]>= target){
            r = mid-1;
        }
        else{
            l = mid+1;
        }
    }
    return l;
}


void insert_sort(vector<int> &nums){
    int len = nums.size();
    if(len<2){
        return;
    }

    for(int i =1; i<len; ++i){
        int target = nums[i];
        int index = lowwer_bound(nums, i-1, nums[i]);
        for(int j =i; j> index; --j){
            // 将 index 位置空出来
            nums[j] = nums[j-1];
        }
        nums[index] = target;
    }
}
```

希尔排序------高级的插入排序

- 先将整个待排序的记录序列**分割成为若干子序列**分别进行直接插入排序，将距离为`gap`的所有元素分到**一组**，`gap`是 子数组 **数量**
- 待整个序列中的记录 “基本有序” 时，再对**全体记录**进行**依次直接插入**排序。
- 四层循环

先将初始步长设为 n/2，然后对每个步长内的元素进行插入排序操作，再将步长缩小一半，重复上述操作，直至**步长为 1** 时完成排序。

```
void ShellSort(int nums[],int len){
```

```
    // 步长调整 log(n)次，初始步长gap为len/2
    for(int gap=len/2; gap> 0; gap=gap/2){
        // 子数组数量为 gap
        for(int i=0; i<gap; i++){
             // 遍历子数组，进行直接插入排序
            for(int j=i+gap; j<len; j+=gap){
                int temp=nums[j];
                int m=j-gap;
                for(m=j-gap; m>= 0 && nums[m]>temp; m-= gap){//nums[m]小于 temp 时循环结束，结束后应将 temp 赋值给a[m+gap]
                    nums[m+ gap]=nums[m];   //只要nums[m]比temp大，就将此数后移一位（此处是gap位）
                }
                nums[m+ gap]=temp;      //将 temp 赋值给a[m+gap]
            }
        }
    }
}
```

```
void shell_sort(vector<int> &nums) {
    int len = nums.size();
    // 初始步长为数组长度的一半
    int gap = len / 2;
    while(gap > 0) {
        // 直接插入排序，首元素默认有序，从 第二个元素开始遍历
        for(int i = gap; i < len; i++) {
            int temp = nums[i];
            int j = i;
            while(j >= gap && nums[j - gap] > temp) {
                arr[j] = nums[j - gap];
                j -= gap;
            }
            nums[j] = temp;
        }
        gap /= 2; // 步长递减
    }
}
```

冒泡排序

- 比较相邻的元素。如果第一个比第二个大，就交换他们两个。
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，**最后的元素**会是最大的数。
- 针对所有的元素重复以上的步骤，除了最后一个。
- 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

```
void bubblesort(vector<int> nums){
    int len = nums.size();

    for(int i = 0; i< len-1; ++i){
        // 有多少元素 已经 是有序的
        for(int j=0; j<len-1 -i; ++j){
            // 因为要 比较 nums[i] 与nums[i+1], 所以只能到 len-2 -i
            if(nums[j]> numd[j-1]){
                nums[j]+= nums[j+1];
                nums[j+1] = nums[j]-nums[j+1];
                nums[j] -= nums[j+1];
            }
        }
    }
}
```

选择排序

- 首先在未排序序列中找到最小（大）元素，存放到排序序列的**起始位置**
- 再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。
- 重复第二步，直到所有元素均排序完毕。

```
void selectionsort(vector<int> nums){
    int len = nums.size();

    for(int i = 0; i<len-1; ++i){
        int maxi = i;
        for(int j = i+1; j<len; ++j){
            if(nums[maxi]> nums[j]){
                maxi = j;
            }
        }
        int temp =nums[i];
        nums[i] = nums[maxi];
        nums[maxi] = temp;
    }
}
```

归并排序(unfinish)

- 将待排序数组分成两个子数组，分别对它们进行递归排序，直到每个子数组只剩下一个元素。
- 然后，将两个排好序的子数组合并成一个大的有序数组。
- 重复上述过程，不断地二分子数组、排序、合并，直到整个数组排序完成。

```
void merge(vector<int>& nums, int l, int mid, int r) {
    // 归并排序的合并操作，将两个有序的子序列合并成一个有序序列
    int index = 0; // 辅助数组的索引
```

```
    int ptrL = l; // 左子序列的指针
    int ptrR = mid; // 右子序列的指针
    static vector<int>tempary; // 静态辅助数组，用于暂存排序结果

    if (nums.size() > tempary.size()) { // 根据排序数组长度动态调整辅助数组大小
        tempary.resize(nums.size());
    }

    while (ptrL != mid && ptrR != r) { // 循环遍历左右子序列，比较并合并元素
        if (nums[ptrL] < nums[ptrR]) {
            tempary[index++] = nums[ptrL++];
        } else {
            tempary[index++] = nums[ptrR++];
        }
    }
    while (ptrL != mid) { // 将剩余的左子序列元素合并至结果数组
        tempary[index++] = nums[ptrL++];
    }
    while (ptrR != r) { // 将剩余的右子序列元素合并至结果数组
        tempary[index++] = nums[ptrR++];
    }
    copy(tempary.begin(), tempary.begin() + index, nums.begin() + l); // 将暂存的排序结果拷贝回原始数组
}

void mergeSort(vector<int>& nums, int l, int r) {
    if (r - l <= 1) { // 区间长度小于等于1时结束递归
        return;
    }
    int mid = (l + r) / 2; // 将排序区间二分
    mergeSort(nums, l, mid); // 递归地对左子序列进行排序
    mergeSort(nums, mid, r); // 递归地对右子序列进行排序
    merge(nums, l, mid, r); // 将排序好的左右子序列合并成一个有序序列
}
```

快速排序

- 从数列中挑出一个元素，称为 “基准”（pivot）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；该操作实现了找到 ''基准' 的位置.
    - 挖坑法, 将基准所在的位置作为坑
    - 从右边 找到第 一个 **小于** pivot 的元素, 左边的坑被填满, 坑出现在了右边
    - 从左边 找到第 一个 **大于** pivot 的元素 , 坑又出现在了左边
- **递归**地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序；

```
void quicksort(vector<int> &nums, int start, int end){
    // 排序范围是 [start, end]
```

```
    if(start >= end){
        return ;
    }

    // 否则选择 **第一个元素** 做为 **基准**
    int index = start;
    int pivot = nums[index];
    int l = start, r=end;
    while(l< r){
        // 从右边 找到第一个 小于 pivot 的元素
        while(l<r && nums[j] >= pivot){
            --j;
        }
        nums[i] = nums[j];

        // 从左边 找到一个 大于pivot 的元素
        while(l< r && nums[i]<= pivot){
            ++i;
        }
        nums[j] = nums[i];
    }
    nums[l] = pivot;

    // 递归
    quicksort(nums, start, l-1);
    quicksort(nums, l+1, end);

}


// {5, 3, 8, 4, 2}
// 第一次循环
// l = 0, r = 4 <=> nums[l] = nums[j] <=> {2, 3, 8, 4, 2}
// l = 2, r = 4 <=> nums[j] = nums[i] <=> {2, 3, 8, 4, 8}
// 第二次循环
// l = 2, r = 3 <=> nums[l] = nums[j] <=> {2, 3, 4, 4, 8}
// l = 3, r = 3 <=> nums[j] = nums[i] <=> {2, 3, 4, 4, 8}
// 循环结束
```

堆排序

堆的**操作**

- 插入元素
    - 插入到堆末尾, 然后开始调整
- 删除元素, 也就是 将堆顶元素输出

算法步骤

- 建成初始堆, (从后往前, 从下向上)
  - 对第 **n/2 个结点为根的子树** 不断 **向上** 进行调整,
  - **向前**依次对各结点 (n/2 - 1 ~ 1) 为根的**子树**进行
- 把**堆首**（最大值）和**堆尾**互换；
- 把堆的尺寸缩小 1，并调用 shift_down (0)，目的是把新的数组顶端数据调整到相应位置；
- 重复步骤 2，直到堆的尺寸为 1。

```
void heapify(vector<int>& arr, int len, int i) {
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    int largest = i;

    if (left < len && arr[left] > arr[largest]) {
        largest = left;
    }

    if (right < len && arr[right] > arr[largest]) {
        largest = right;
    }

    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, len, largest);
    }
}

void buildMaxHeap(vector<int>& arr) {
    // 建堆
    int len = arr.size();
    for (int i = len / 2; i >= 0; i--) {
        heapify(arr, len, i);
    }
}

void heapSort(vector<int>& arr) {
    buildMaxHeap(arr);

    for (int i = arr.size() - 1; i > 0; i--) {
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}

int main() {
    vector<int> arr = {5, 2, 9, 1, 5, 6};
    heapSort(arr);
    for (int x : arr) {
        cout << x << " ";
    }
    cout << endl;

    return 0;
}
```

计数排序---线性时间复杂度

要求 : 要求输入的数据必须是有确定范围的**整数**

步骤:

- 找出待排序数组中的**最大值 max 和最小值 min**;
- 创建一个计数数组 count，大小为 max-min+1，将 count 数组的元素全部初始化为 0;
- 遍历待排序数组，统计每个元素 出现的**次数**，即在 count [i-min] 中增加 1;
- 遍历 count 数组，将每个元素设置为前面所有元素之和，即 count [i-min] 表示小于等于 i 的元素个数;
- 反向遍历待排序数组，根据 count 数组中的信息将每个元素放到输出数组中的合适位置;

```
void countingSort(vector<int>& nums) {
    int max_val = *max_element(nums.begin(), nums.end()); // 获取最大值
    vector<int> count(max_val + 1, 0); // 初始化计数数组
    vector<int> result(nums.size()); // 初始化结果数组

    for (int i = 0; i < nums.size(); i++) {
        count[nums[i]]++; // 统计元素出现次数
    }
    for (int i = 1; i < count.size(); i++) {
        count[i] += count[i - 1]; // 统计不大于i的元素个数
    }
    for (int i = nums.size() - 1; i >= 0; i--) {
        // int index = count[nums[i]] -1;
        result[count[nums[i]] - 1] = nums[i]; // 根据元素出现次数和位置填充结果数组
        count[nums[i]]--;
    }
    nums = result; // 将排序后的结果赋值给原数组
}
```

桶排序

- 确定桶的个数和桶的范围：将待排序的元素分到几个桶中，桶的范围可以根据待排序元素的大小范围来确定。
- 将元素放入桶中：遍历待排序的元素，根据元素的大小，将元素放入对应的桶中。
- 对每个**桶内的元素进行排序**：对每个桶内的元素进行排序，可以使用插入排序等排序算法。
- 合并桶：将每个桶内排好序的元素按顺序合并起来，形成有序的序列。

```
void bucketSort(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return;
```

```
    // 找到待排序数组的最大值和最小值
    int min_val = nums[0], max_val = nums[0];
    for (int num : nums) {
        min_val = min(min_val, num);
        max_val = max(max_val, num);
    }

    // 计算桶的数量
    int bucket_size = max(1, (max_val - min_val) / n + 1);
    int bucket_count = (max_val - min_val) / bucket_size + 1;

    // 初始化桶
    vector<vector<int>> buckets(bucket_count);
    for (int num : nums) {
        int bucket_idx = (num - min_val) / bucket_size;
        buckets[bucket_idx].push_back(num);
    }

    // 对每个桶中的数进行排序
    for (auto& bucket : buckets) {
        sort(bucket.begin(), bucket.end());
    }

    // 把桶中的数合并到结果数组中
    int i = 0;
    for (auto& bucket : buckets) {
        for (int num : bucket) {
            nums[i++] = num;
        }
    }
}
```

基数排序

- 取得数组中的最大数，并取得**位数**。
- 从个位(**最低位**)开始，按照每个位数的值分别将数据放入相应的桶中。
- 将各个桶中的数据按照从小到大的顺序**依次取出**，得到新的数组。
- 重复步骤 2 和 3，直到所有的位数都被处理完毕。

```
// 计算位数
int countDigit(int n){
    int count = 0;
    while(n){
        n /= 10;
        ++count;
    }
    return count;
}

void radixSort(vector<int>& nums){
```

```
    int maxDigits = 0;  // 最大位数
    for(int num: nums){
        maxDigits = max(maxDigits, countDigit(num));
    }

    // 位数
    int exp = 1;
    vector<int> bucket(10);

    for(int i = 0; i < maxDigits; ++i){
        // 清空桶
        fill(bucket.begin(), bucket.end(), 0);

        // 分桶
        for(int num: nums){
            int digit = (num / exp) % 10;
            ++bucket[digit];
        }

        // 累加桶
        for(int j = 1; j < 10; ++j){
            bucket[j] += bucket[j - 1];
        }

        // 按照桶的顺序收集元素
        vector<int> temp(nums.size());
        for(int k = nums.size() - 1; k >= 0; --k){
            int digit = (nums[k] / exp) % 10;
            temp[--bucket[digit]] = nums[k];
        }

        // 更新数组
        nums = temp;

        // 下一位
        exp *= 10;
    }
}
```

树的递归法遍历

```
void traverse(TreeNode *root. res) :
    if (root == null) :
        return;

    # 前序操作
    traverse(root.left)
    # 中序操作
    traverse(root.right)
    # 后序操作
```

树的 非递归 遍历

```
//--------------------先序遍历----------------------
vector<int> preorderTraversal(TreeNode* root) {
    //栈的作用是用来记录哪些节点没有进行处理
    stack<TreeNode*> st;
    vector<int> result;
    if (root == NULL) return result;
    st.push(root);//入栈
    while (!st.empty()) {
        //把节点取出来，对值操作，再把左右孩子放进栈
        TreeNode* node = st.top();                          // 中
        st.pop();
        result.push_back(node->val);
        if (node->right) st.push(node->right);          // 右（空节点不入栈）
        if (node->left) st.push(node->left);            // 左（空节点不入栈）
    }
    return result;
}


//----------------------中序遍历-------------------------
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> st;
    TreeNode* cur = root;
    while (cur != NULL || !st.empty()) {
        if (cur != NULL) { // 指针来访问节点，访问到最底层
            st.push(cur); // 将访问的节点放进栈
            cur = cur->left;                // 左
        } else {// cur == nullptr
            // 左边处理完了，要处理中间了
            // 从栈里弹出的数据，就是要处理的数据（放进result数组里的数据）
            cur = st.top();
            st.pop();
            result.push_back(cur->val);     // 中
            cur = cur->right;               // 右
        }
    }
    return result;
}
```

```cpp
//------------------后序遍历------------------
//先序遍历是中左右--(调整代码左右循序)-->中右左--(反转result数组)->左右中(也就是后序)
vector<int> postorderTraversal(TreeNode* root) {
    stack<TreeNode*> st;
    vector<int> result;
    if (root == NULL) return result;
    st.push(root);
    while (!st.empty()) {
        TreeNode* node = st.top();
        st.pop();
        result.push_back(node->val);
        if (node->left) st.push(node->left); // 相对于前序遍历，这更改一下入栈顺序 （空节点不入栈）
        if (node->right) st.push(node->right); // 空节点不入栈
    }
    reverse(result.begin(), result.end()); // 将结果反转之后就是左右中的顺序了
    return result;
}
```

四. Modern C++

1 简介

C++11 是c++划时代的一个版本, C++ 11 及其后续版本统称为 "**Modern C++**"

- C++借助**型别推导机制**来判定哪里应该用什么类型, 型别一旦判定, 就固定下来
- 右值引用, 尽可能**复用**已经在内存中已经存在的值

对于**新的语言特性**, 掌握本身并不困难, 而是考虑到它与众多**已存在的语言特性**之发生的相互作用就不容易了

C++14 中, 有函数返回值性别推导

1.0 C++的优势

C++的**高抽象**层次，又兼具**高性能**，是其他语言所无法替代的，但在各种活跃编程语言中，C++门槛依然很高，尤其C++的内存问题（内存泄露，内存溢出，内存宕机，堆栈破坏等问题），需要理解C++标准对象模型，C++标准库，标准C库，操作系统等内存设计，才能更加深入理解C++内存管理，这是跨越C++三座大山之一，我们必须拿下它。

1.1 c++11 新特性

C++11 是 C++ 编程语言的一个新标准，于 2011 年正式发布。它引入了许多新的特性和语言改进，使得 C++ 更加现代化、安全和高效。以下是 C++11 中一些重要的新特性：

1. **右值引用和移动语义**：引入了右值引用和移动构造函数，允许通过移动而不是拷贝来传递对象的值。这可以显著提高程序的性能和效率。
2. **Lambda** 表达式：引入了 Lambda 表达式，使得 C++ 中的函数式编程更加方便和灵活。
3. 自动类型推导：引入了 auto 关键字，允许编译器自动推导变量类型。
4. 列表初始化：引入了统一的初始化语法，允许使用花括号来初始化数组、容器和结构体等对象。
5. 空指针常量：引入了空指针常量 nullptr，取代了旧的 NULL 宏。
6. 智能指针：引入了 shared_ptr 和 unique_ptr 等智能指针类，使得内存管理更加安全和方便。
7. 线程支持：引入了对多线程编程的支持，包括 thread、mutex、condition_variable 等标准库组件。
8. **foreach 循环**：引入了 foreach 循环语法，使得遍历容器和数组等对象更加方便。
9. constexpr 函数：引入了 constexpr 关键字，允许编译时求值的函数。
10. 变长模板参数：引入了变长模板参数语法，允许在模板中使用可变数量的参数。

除了以上列出的新特性，C++11 还包括了许多其他的语言改进和标准库增强，如 noexcept 关键字、nullptr、long long、std::to_string ()、std::thread_local 等等。这些新特性和改进使得 C++ 更加现代化、安全和高效，有助于提高程序员的生产力和代码质量。

1.2 c++ 14 的新特性

1. 二进制字面量：可以用0b或0B前缀来表示二进制数字，如0b1101表示十进制的13。
2. 生成通用编译时常量表达式的constexpr函数：constexpr函数可以在编译时执行，其返回值可以在常量表达式中使用。
3. **初始化列表和构造函数**使用通用类型列表：构造函数可以使用initializer_list来初始化成员变量。
4. lambda表达式的返回值类型可以自动推导：可以省略lambda表达式中return语句的返回类型。
5. 通用的返回值类型推导：函数的返回类型可以使用auto关键字代替明确的类型声明。
6. 静态断言：可以使用static_assert在编译期间进行断言验证。
7. 变量模板：可以使用template定义一个变量并初始化。
8. 通用lambda表达式：lambda表达式可以使用auto作为参数类型。
9. 字符串字面量的变长编码：可以使用UTF-8编码的字符字符串字面量。

10. 声明变量时可以使用auto关键字来自动推导类型：可以根据变量的初始化值自动推导变量类型。

C++14 是 C++ 编程语言的一个标准，于 2014 年发布，引入了一些新的语言特性和库增强。以下是 C++14 的一些主要特性，并且每个特性都附有一个简单的示例：

1. 二进制字面量：允许使用二进制表示整数，以前只能使用十进制、八进制和十六进制。

```
// 二进制字面量示例
int binaryNum = 0b101010;  // 二进制数 101010，等同于十进制的 42
```

2. 返回类型自动推断：允许在函数声明时使用 `auto` 自动推断返回类型。

```
// 返回类型自动推断示例
auto add(int a, int b) {  // 返回类型自动推断为 int
    return a + b;
}
```

3. **泛型 lambda 表达式**：允许使用泛型参数的 lambda 表达式，不需要显式指定参数类型。

```
// 泛型 lambda 表达式示例
auto sum = [](auto a, auto b) {  // 泛型 lambda 表达式，可以接受任意类型的参数
    return a + b;
};
int result = sum(3, 4);  // result = 7
float f_result = sum(1.5, 2.5);  // f_result = 4.0
```

4. 初始化列表的返回值推断：允许在函数中使用初始化列表作为返回值，并自动推断返回类型。

```
// 初始化列表的返回值推断示例
auto getNumbers() {  // 返回类型自动推断为 std::initializer_list<int>
    return {1, 2, 3, 4, 5};
}
auto numbers = getNumbers();  // numbers 类型为 std::initializer_list<int>
```

5. 更强大的 constexpr：允许在 constexpr 函数中使用更多的语言特性，包括控制流语句（如 if 和 switch）和局部变量定义。

```
// 更强大的 constexpr 示例
constexpr int factorial(int n) {
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
constexpr int result = factorial(5);  // result = 120
```

6. 编译时字符串字面量操作：允许在编译时对字符串字面量进行操作，如拼接、截取等。

```
// 编译时字符串字面量操作示例
constexpr const char* hello = "Hello";
constexpr const char* world = "World";
constexpr const char* helloWorld = hello " " world;  // 在编译时拼接字符串
```

7. lambda 表达式的泛型参数列表初始化：允许在 lambda 表达式中使用初始化列表初始化泛型参数。

```
// lambda 表达式的泛型参数列表初始化示例
int x = 2;
auto printValues = [x](auto... args) {  // 使用初始化列表初始化泛型参数
```

1.3 c++17 新特性

C++17 是 C++ 编程语言的另一个新标准，于 2017 年正式发布。相比于 C++11 和 C++14，C++17 引入了更多的新特性和语言改进，以下是其中的一些：

1. 结构化绑定：引入了结构化绑定语法，允许通过一行代码将结构体或元组中的成员绑定到变量上。
2. if constexpr：引入了 if constexpr 语法，允许在编译时选择不同的代码路径，从而提高代码的效率和可读性。
3. constexpr if：引入了 constexpr if 语法，允许在编译时选择不同的代码路径，使得模板编程更加灵活和高效。
4. inline 变量：引入了 inline 变量语法，允许在头文件中定义和初始化全局变量，从而避免了多个编译单元中的定义冲突。
5. 折叠表达式：引入了折叠表达式语法，允许通过简洁的语法实现对参数包的操作，如求和、取最大值等。
6. std::optional：引入了 std::optional 类型，允许表示可能为空的值，避免了裸指针的使用。
7. std::string_view：引入了 std::string_view 类型，允许对字符串进行视图操作，避免了字符串的不必要拷贝和内存分配。
8. 新的文件系统库：引入了新的文件系统库，允许对文件系统进行更加简便和安全的操作。
9. 并行算法：引入了并行算法库，允许利用多核处理器并行地执行算法，提高程序的性能和效率。

除了以上列出的新特性，C++17 还包括了许多其他的语言改进和标准库增强，如 constexpr lambda、constexpr std::tuple、std::invoke、std::clamp 等等。这些新特性和改进使得 C++ 更加现代化、安全和高效，有助于提高程序员的生产力和代码质量。

2 C++ 11

2.1 foreach循环

使用了`迭代器`(iterator) 来遍历容器中的元素。具体实现机制如下：

1. 首先，编译器会调用容器的 begin () 成员函数，获取容器的起始位置迭代器；
2. 然后，每次循环都将迭代器作为条件，检查是否到达容器的末尾 (end ())；如果没有到达末尾，则执行循环体内的语句；
3. 在循环体中，auto& nums 表示当前迭代器指向的元素的引用，也就是可以直接操作和修改当前元素；
4. 最后，编译器会自动将迭代器指向容器的下一个元素，然后继续执行循环。

需要注意的是，**只有实现了迭代器的容器**才能使用范围 for 循环。对于普通的数组，也可以使用这种语法，但实际上编译器会将其转换为传统的 for 循环来处理。

2.2 Lambda表达式

- `lambda` 表达式的类型是一个闭包 (closure)又或者说, **函数对象**(仿函数)，它可以被转换成一个**函数指针**或者一个 std::function 对象。在 C++ 中，所有的 lambda 表达式都产生一个**匿名类**类型并实例化该类型的一个对象。
  - functon<>会使用比函数对象更多的内存
  - 函数对象运行更快

```
[传入形参的方式](函数形参)修饰符->返回值类型{函数体};
for_each(a.begin(), a.end(),[&](int x){cout<< x<< endl;});

auto sum = [](int a, int b) -> int {
    return a + b;};
```

传入形参的方式

[] 标识一个Lambda表达式的开始，这一部分是**不可以忽略**的

- 空：代表**不捕获**Lambda表达式外的变量；
- &：代表以**引用**传递的方式捕获Lambda表达式外的变量；
- =：代表以值传递的方式捕获Lambda表达式外的变量，即以const引用的方式传值；
- `变量名`：按照变量名捕获
- this：表示Lambda表达式可以使用Lambda表达式所在类的成员变量；

```
int a = 10, b = 20;

// 所有变量以值的形式传入
auto lambda1 = [=](){ return a; };

// 都以引用的方式
auto lambda2 = [&](){ return a; };

// 按照变量名捕获
auto lambda3 = [a](){ return a; };
```

函数形参

这一部分可以被省略（如果函数无参数）

修饰符

这一部分是可以省略的，常见的修饰符有两个，一个是mutable，另一个是exception

- mutable：当函数参数以值引用传递方式传递时，在函数体内是不可以修改该函数参数的值的，我们可以使用mutable修饰符，使得该函数参数可以在函数体内改变
- exception：exception 声明用于指定函数抛出的异常，如抛出整数类型的异常，可以使用throw(int)

返回类型

这一部分也是可以省略的，Lambda表达式会自动推断返回值类型，但是返回类型不统一会报错；

函数类型

标识函数的实现，这一部分可以为空，但是不能省略。

2.3 auto

- 必须初始化
- `C14` 支持 `lambda` 表达式的形参也可以使用 `auto`

```
auto f = [](const &a, const&b){// 应用于类似指针之物
    return *a< *p;
}
```

2.4 function (函数包装器)

C++11 标准库中引入的一个模板类, 可以**指向**

- 包括函数，lambda 表达式，函数指针, 其他函数对象，
- 指向**成员函数**和指向**数据成员**的指针。

```
#include <functional>

int add(int a, int b){
    return a+b;
}

template< class T>
T add(T a, T b){
    return a+b;
}

int main(){
    // 一般情况
    std::function<int (int, int)> f1 = add;
    cout<< add(3, 4);

    // 当使用模板时
    std::function<int(int, int)> f2 = add<int>;

    // lambda 表达式
    auto f = [](const int a, const int b) { return a + b; };
```

```
    std::function<int(int, int)>f3 = f;

    // 静态成员函数

    return 0;
}
```

2.4.2 function 与 auto 与 递归

```
// 会出现错误，因为使用auto 变量的类型由编译器根据变量的初始化表达式推导得出。
// 函数没结束，就没有推导出类型，因此不能用 auto
// 使用 int也会报错
// `fun declared with deduced type 'auto' cannot appear in its own initializer`
// 也就是说，当该函数是递归时，不能使用auto声明，而应该确切的声明
auto fun1= [](int x, int y){
    return x+y;
}

auto fun2 = [](int x) -> int {
    if(x == 1){
        return 1;
    }
    auto y = fun2(x - 1);
    return x * y;
};

function<int(int)> fun3 = [](int x) -> int {
    if(x == 1)return 1;
    auto y = fun(x - 1);
    return x * y;
};
```

2.5 返回类型后置

- `返回类型后置语法` 是一种 C++11 引入的特殊语法，用于在函数声明中将返回类型放到参数列表之后。

```
auto functionName(Args... args) -> ReturnType;

// 实例
template <typename Container>
auto begin(Container& c) -> decltype(c.begin()) {
  return c.begin();
}

template <typename Container>
auto end(Container& c) -> decltype(c.end()) {
  return c.end();
}
```

2.6 using

- `using` 是一个用于类型别名（type alias）或模板别名（template alias）定义的关键字
- 解决了`typedef`无法定义模板别名

```
using new_name = old_type;

using int_ptr = int*;
int_ptr p = new int(10);


//  声明模板的别名
template <typename T>
using vec = std::vector<T>;
vec<int> my_vec = {1, 2, 3}; // 使用别名 vec 定义 vector 变量
```

2.7 右值和左值

- 左值是可以被**地址**访问的、有持久标识的对象或变量，右值则是不可被直接访问的、**临时生成的对象或值。**通常来说，左值表示一个具有地址标识的实体，而右值表示该实体的值。
- **左值**: 是指放在`=`**左边**可以被赋值的值（左值必须要在**内存**中有实体）
- **右值**: 右边赋给其他变量的值（右值可以在内存也可以在 CPU 寄存器）
  - 右值**不可以直接修改**
  - 右值可以赋值给左值
  - 右值**不可以直接取地址**

C++11 将右值分为**纯右值**和**将亡值。**

- 纯右值：非引用返回的临时变量；运算表达式产生的结果；字面常量.
- 将亡值：其实就是中间变量的过渡，过度后就消亡，分为两种
  - 函数的临时返回值 int a = f (3);f (3) 的返回值就是右值，副本拷贝给 a，然后消亡

- 表达式像 (x+y)，其中 (x+y) 就是右值

```
// 左值引用
// 类型名 & 引用名 = 左值表达式;
int a = 10;
int &ra = a;
```

2.8 右值引用

- 右值引用的引入主要是为了支持**移动语义**（Move Semantics）和**完美转发**（Perfect Forwarding）。
- 移动语义：当使用右值引用时，可以利用右值对象的数据来初始化另一个对象，而不需要进行深拷贝操作。这样可以避免不必要的内存分配和释放，提高程序效率和性能。
- 完美转发：右值引用还可以用于实现完美转发的功能，即将一个函数参数以原有类型和值的方式转发给另一个函数，从而避免了额外的对象创建和销毁操作。
- 消除临时对象：通过使用右值引用，可以避免创建不必要的临时对象，减少程序开销。

`返回值`没有意义的复制构造, 虽然可以通过**Return-value optimization** (RVO)来进行优化, 但是语言本身没有优化的空间

```
std::string foo() {
    return string(10, 'c');
}
std::string t = foo();
```

```
// O(n)
for (auto i = 0; i != n; ++i)
    s += some_char(i);

// O(n^2), 会对原来的s 做一次拷贝
for (auto i = 0; i != n; ++i)
    s = s + some_char(i);
```

```
std::vector&&<int> getVector() {
  std::vector<int> v;
  // 添加元素到 v 中
  return std::move(v); // 使用 std::move 将 v 转换为右值引用
}

// 使用右值引用来初始化一个新的 vector 对象，避免不必要的拷贝操作
std::vector<int> vec = getVector();
```

2.9 万能引用

- 既可以是`右值引用`,也可以是`左值引用`,二者居其一
- 只有涉及到`类型推导`时

```
// 万能引用
template <typename T> void f(T&& para);

// 表现为左值
int a;
f(a);

// 表现为右值
f(move(a))

// 万能引用
auto && var2 = var1;
```

```
// 例外，性别推导在 推导类的时候就已经推导出来了
// 因此，不是万能引用
template<class T>{
public:
    void push_back(T&& x);
}
// 型别独立于 T时，才会是 万能引用
```

引用折叠

- 在 C++ 中，**“引用的引用”是非法的。**像 auto& &rx = x;（注意两个 & 之间有空格）这种直接定义引用的引用是不合法的，但是编译器在通过类型别名或模板参数推导等语境中，会间接定义出 “引用的引用”，这时引用会形成 “折叠”。

- 引用折叠**会发生在模板实例化、auto 类型推导、创建和运用 typedef 和别名声明、以及 decltype 语境中。**
- `& &` `& &&` and `&& &` collapse to &
- `&& &&` collapses to `&&`

```
template <typename T>
class A {
public:
    using ref_type = T&&;
};

// 使用引用折叠简化类型表达式
A<int&>::ref_type x = 1;      // 折叠成 int&
A<int&&>::ref_type y = 2;     // 折叠成 int&&
```

类型转换

- `static_cast()` 使用最多, 低风险的类型转换
- `reinterpret_cast` 任何类型
- `const_cast` 去掉const
- `dynamic_cast` 继承关系间，指针和引用的互相转换

static_cast

- 通常用于转换**数值**数据类型（如 float -> int）
- 用于非多态类型的转换
- 不执行运行时类型检查（转换安全性不如 dynamic_cast）
- 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）

```
double d = 3.14159;
int i = static_cast<int>(d); // i= 3
```

dynamic_cast

- 在运行时将一个指向**基类**的指针或引用转换为指向**派生类**的指针或引用。相比于 `static_cast`，`dynamic_cast` 的作用更加动态和安全，可以**避免不安全**的类型转换。

```
class Animal {
public:
    virtual ~Animal() {}
};

class Cat : public Animal {
public:
    void meow() { cout << "Meow!" << endl; }
};

class Dog : public Animal {
public:
    void bark() { cout << "Woof!" << endl; }
};


Animal *a1 = new Cat();
Animal *a2 = new Dog();

Cat *c1 = dynamic_cast<Cat*>(a1);
```

const_cast

2 智能指针

普通指针的缺点

- 没有指出是**单个对象**还是一个**数组**

共有三种智能指针, 都被定义在 **memory** 头文件中

- shared_ptr<T>
- unique_ptr<T>
- weak_ptr<T>

和普通指针的区别

- 重载了操作符 *，-> 可以象使用指针那样使用它们
- 只对数组版本的 重载了数组操作符 []，可以通过它访问指定位置的数组元素

- **没有重载** +，-，++，–，+=，-= 等算术运算操作符，如果访问其它位置的元素，只能通过裸指针

2.1 shared_ptr<T>

```
// 创建
shared_ptr<string> sp;

// 创建与构造
make_shared<T> (args);
shared_ptr<int> p1 = make_shared<int> (42);

// 返回p中所保存的 指针
sp.get();

// 是否独占
p.unique();

// 返回和 p 共享对象的智能指针
p.use_count();
```

2.2 shared_ptr && 工厂模式 创建对象

```
shared_ptr<Foo> factory(T arg ){
    // 处理 arg
    return make_shared<Foo> (arg);
}
```

2.3 unique_ptr<T>

- `unique_ptr` 实现的是**专属所有权**语义，用于独占它所指向的资源对象的场合。某个时刻只能有一个 unique_ptr 指向一个动态分配的资源对象，也就是这个资源不会被多个 unique_ptr 对象同时占有，它所管理的资源只能在 unique_ptr 对象之间进行移动，不能拷贝，所以它只提供了移动语义。
- 某一个时刻, 只能有一个 unique_ptr 指向 一个 资源 对象, 同时, 默认情况下, 有着和裸指针相同的大小
- 没有拷贝语义, 它的**拷贝构造和拷贝赋值函数**都是 `=delete` 的
- 如果使用裸指针的话，需要在函数退出的所有路径处，包括发生异常时，都要加上**释放内存**的语句，否则就会产生内存泄露。
- 智能指针之 unique_ptr

```
// 创建, 删除器缺省为delete
unique_ptr<int> up(new int(8));
```

```
// 指向一个10个元素的int型数组，删除器缺省为 `delete[]`
unique_ptr<int[]> ar_up(new int[10]);
// unique_ptr<int[10]> ar_up(new int[10]); // 无法编译

// 可以不拥有资源对象
unique_ptr<int> up_empty;
// 空对象可以在需要的时候使用 reset () 为它分配指针
up_empty.reset(new int(42));

// 不能直接把裸指针赋值给 unique_ptr 对象，它没有提供这样的隐式转换
// 但是可以将nullptr 赋值给它
unique_ptr<int> up_1(nullptr);

// release 返回的指针通常被用来 初始化 或者 赋值
// 管理指针的 责任 从一个智能指针 转移给另一个
p2.reset(p1.release());
// 以下是错误的
// p1.release();

// 不支持 拷贝或者赋值操作
// 以下初始化方式是错误的
unique_ptr<int> p2(p1);

// // 使用 std::move() 转移所有权
up_empty= std::move(up);
```

移动的实质------（所有权转移）

auto_ptr是在C++ 98中引入的，在**C++ 17中被移除掉。**它的引入是为了管理动态分配的内存，它的移除是因为本身有**严重的缺陷**，并且已经有了很好的替代者（unique_ptr）。

```
// 可以使用swap 来交换 两个 unique_ptr的内容
unique_ptr<string> ps1(new string("good luck"));
unique_ptr<string> ps2(new string("good luck2"));
ps1.swap(ps2);
std::swap(ps1, ps2);//也可使用标准库的swap
```

2.4 unique_ptr 作为 类内数据成员

- 一个类中，如果数据成员是指针类型的，而这个成员是这个类的对象独有的，也可以考虑使用 `unique_ptr`。
- 如果对象中只有一个指针成员，也可以不使用, 自己在析构函数中 delete 就好
- 如果有多个成员, 那么最好使用 `unique_ptr`
- 因为对资源的独占性, 所以需自定义拷贝构造函数和拷贝赋值运算符

```
class D {
    unique_ptr<A> a;
    unique_ptr<B> b;
public:
    D() : a(new A), b(new B) {
    }

    ~D() = default; // 析构函数使用缺省方式就可以了，会自动析构a和b对象
};
```

问题:

```
class C {
    A *a;
    B *b;
public:
    C(): a(new A), b(new B) { // 构造函数分配资源
    }

    ~C() { // 需要提供析构函数来释放a和b
        delete a;
        delete b;
    }
};
```

- 当构造一个对象时，a已经初始化成功了，当初始化b时抛出了**异常**，此时不会调用析构函数（C++语义规定，没有构造成功的对象不会调用它的析构函数），a 所分配的内存就不会释放，造成**内存泄漏**。

```
#include <memory>
#include <iostream>
using namespace std;

struct Node{
    int data;
    std::unique_ptr<Node> next;

    ~Node()
    {
        cout<<"dtor: "<<data<<endl;
    }
};

struct List{
    std::unique_ptr<Node> head;

    void push(int data)
    {
        head = std::unique_ptr<Node>(new Node{data, std::move(head)});
    }
```

```
    //解开析构函数的递归调用，防止栈溢出
    ~List()
    {
        while (head)
            head = std::move(head->next);
    }
};

int main()
{
    List wall;
    for (int beer = 0; beer != 1000000; ++beer)
    {
        wall.push(beer);
    }
}
```

2.5 unique_ptr && 函数参数

```
void fun(unique_ptr<T> up);
```

- 传入一个**纯右值对象**，要么对左值使用 `move()` 转为将亡值。
- 调用完成后意味着资源对象被转移了，从调用者的实参**转移到函数内部**的 unique_ptr 参数中。函数调用完成之后资源对象被删除, (资源接管)
- 原参数对象就成为了一个空对象，它接下来的命运要么在离开作用域后被销毁，要么接收一个新的资源对象

```
void fun(const unique_ptr<T> &up);
```

- 只能以只读的形式访问up参数，不能进行移动操作，也不能调用reset()、release()等函数
- 不会对实参对象的**生命周期**造成影响，当函数调用完成之后，原参数对象仍然占有资源对象。

```
void fun(unique_ptr<T> &&up);
```

- 可以进行**资源转移**(`std::move`)，或者调用reset()、release()等函数**释放资源**
- 尽量不要使用，稍有不慎，就会造成错误。

```
void fun(unique_ptr<T> &up);
```

- 和右值引用形式的参数类似，也有类似的不安全问题，使用时要注意资源是否被转移了。
- 尽量不要使用，稍有不慎，就会造成错误。

2.6 unique_ptr && 函数返回值

```
// 虽然不支持拷贝操作，但却有一个例外
// 返回值为 unique_ptr 类型的函数，可以直接赋值
unique_ptr<T> get_resource();
auto up = get_resource();
```

- 资源就从函数内部转移到智能指针对象中了，释放资源也就完全交给智能指针来负责了

unique_ptr指针

```
std::unique_ptr<object>* func() {
    std::unique_ptr<object>* ptr = new std::unique_ptr<object>(new object());
    return ptr;
}
std::unique_ptr<object>* ptr = func();
```

对象移动

- 新标准的一个最主要的特性是可以**移动**而非拷贝对象的能力(`将一个对象的值移动到另一个对象中`). 很多情况下都会发生对象**拷贝**. 在其中某些情况下, 对象拷贝后就立即被**销毁**了. 在这些情况下, 移动而非拷贝对象会大幅度提升性能.
- 移动语义的主要好处是可以**避免**对象的**不必要复制**
- 移动语义的实现可能会**改变源对象的值**。在移动之后，源对象通常会变为未定义值，因此在移动后应该避免继续使用源对象。
- 右值引用有一个重要的性质------只能绑定到一个**将要销毁**的对象. 因此,我们可以自由地将一个右值引用的资源 "**移动**" 到另一个对象中.

移动语义

- 因为`unique_ptr`的独占性, 它和其它智能指针只能通过 **move 操作**来转移内部资源，比如可以转移给另一个 unique_ptr 对象，或者 shared_ptr 对象。
- 移动语义和临时对象是有一定关系的。

  当使用移动语义时，通常会用到右值引用。右值引用是一种新的引用类型，它可以绑定到一个右值临时对象上，并且右值引用所绑定的对象可以被移动。

  当一个右值被绑定到右值引用上后，它就具有了对象的身份，可以在程序中被使用。如果使用复制构造函数对右值进行传递，则会复制一个全新的对象，而右值原始对象则会被销毁。但是如果使用移动构造函数进行传递，则会利用右值引用将临时对象的内存资源“移动”到新的对象中，而不需要进行复制构造。

  因此，在使用移动语义时，右值临时对象通常会被用来表示需要在函数调用或表达式计算过程中临时产生的中间值或者暂时不再需要使用的对象。这些临时对象在计算完成后将被销毁，因此可以通过移动语义来有效地利用对象的内存资源，提高程序效率。
- 虽然不能直接将一个右值引用绑定到一个左值上,但可以用move获得一个绑定到左值上的右值引用. 由于move本质上可以接受任何类型的实参,因此我们不会惊讶于它是一个函数模板

```
// 函数签名
extern void foo(unique_ptr<int> up);

// 只有如下形式
unique_ptr<int> up(new int (5));
foo(move(up));

// 或者写成一句话
foo(unique_ptr<int>(new int (5)));

// ----------------------------实现--------------------------------
template<typename T>
typename remove_reference<T>::type&& move(T&& arg)
noexcept {
    return static_cast<typename remove_reference<T>::type&&>(arg);
}

// remove_reference 去除掉 引用
// 头文件：<type_traits>，
// 其定义如下：
template <class T> struct remove_reference;
template <class T> struct remove_reference<T&> {
typedef T type;
};
template <class T> struct remove_reference<T&&> {
typedef T type;
};
```

std::forward() 完美转发

- 保持类型信息, 右值引用类型是独立于值的，一个 **右值引用作为函数参数的形参**时，在函数内部转发该参数给内部其他函数时，它就变成一个**左值**（因为没有实名的右值被编译器视为左值），并不是原来的类型了。
- 当模板函数模板参数是左值引用类型时，std::forward() 返回参数的本身，即参数的类型为 `typename std::remove_reference<T>::type&` 。
- 当模板函数模板参数是右值引用类型时，std::forward() 会进行类型转换并返回相应的右值引用类型，即参数的类型为 `typename std::remove_reference<T>::type&&` 。

```
template <typename T>
T&& forward(typename std::remove_reference<T>::type& arg) noexcept {
    return static_cast<T&&>(arg);
}

template <typename T>
T&& forward(typename std::remove_reference<T>::type&& arg) noexcept {
    static_assert(!std::is_lvalue_reference<T>::value, "template argument
substituting T is an lvalue reference type");
    return static_cast<T&&>(arg);
}
```

Model	Usage
gpt-3.5-turbo	$0.002 / 1K tokens

Effective C++ (3nd)

1 C++ 基础知识

01_将c++看成一个语言联邦

C++由几个重要的**次语言**构成 (我的理解是彼此独立)

- C语言: 区块, 语句, 预处理器, 数组, 指针等
- 类: 封装, 继承多态, 动态绑定
- 模板: 泛型编程
- STL: 模板库, 容器, 算法, 迭代器

02_使用const enum inline 来替换 define

- 对于单纯常量, 最好以const 对象或enums 替换 #defines. #define 不能定义类的常量，因为被 #define 定义的常量可以被**全局访问**，它不能提供任何**封装性。**

```
#define ASPECT_RATIO 1.653
// 改写成
const double AspectRatio 1.653;

class Gameplayer{
private:
    // static 确保此常量至多只有一份实体
    static const in NumTurns = 5;
    int scores[NumsTurns];
};
```

- 对于形似函数的宏(macros), 最好改用inline函数替换 #defines.

```
// 这是一个三目运算符，如果 a > b，则返回 a，否则返回 b
// 宏中的每一个实参都必须加上小括号
#define MY_MAX(a, b) f((a) > (b) ? (a) : (b))

int a = 5, b = 0;

// 被累加了两次
// ++a => a = 6 => 6 > b = 0 => return ++a;
MY_COMPARE(++a, b);

// 被累加了一次
MY_MAX(++a, b + 10);//2
```

03_尽可能使用 const

1、const 修饰 变量，指针，函数，函数返回值等，可以使程序减少错误，或者更容易检测错误：

const 右靠近原则:

```
// 指针常量：   靠近 p ,也就是指针地址是不变的
// 指针**地址不可变**，指针指向值可变
int* const p;


// 常量指针：靠近 int 也就是说
// 指针指向值不可变，指针地址可变
const int* p;

// 常量指针常量：
// 都不可变
const int* const p;
```

const 修饰迭代器:

- iterator 相当于 T* const // 指针常量, 指针**地址不可变**，指针指向值可变
- const_iterator 相当于 const T* // 常量指针

```
class Rational{};
// 为什么返回一个const 对象
// if (a * b=c) , 该行为是没有意义的, const 可以避免这种情况的发生
const Rational operator*(const Rational &l, const Rational &b);
```

const 成员函数

- 将 const 实施于成员函数的目的, 是为了确认该成员函数可作用于**const对象**, 两个成员函数如果只是常量性(constness)不同, 可以被重载

```
const char& operator[](int pos) const{
    return text[pos];
}

// 调用 const 实现
char& operator[](int pos){
    // const_cast 移除第一次转型添加的 const
    return const_cast<char&>(
                static_cast<const char&>(*this)[pos]);
    // static_cast 把 char& 类型数据转换为 char &
}
```

04_确定对象被使用前已经初始化

- 为**内置类型**进行手工初始化,因为C++不保证初始化它们.
- 构造函数最好使用成员初值列(member initialization list), 而不要在构造函数本体内使用赋值操作(assignment).初值列列出的成员变量,其排列次序应该和它们在class中的声明次序相同.
- 为免除"跨编译单元之初始化次序"问题,请以 local static 对象替换non-local static对象.

```
// 内置数据类型
int a =0;
char *b = "asdqwe";

// 自定义类型
// 构造函数 1. 设初值, 2.赋新值
// 列表初始化 直接 复制构造
```

C++有着十分固定的"成员初始化次序".

- base classes更早于其derived classes被初始化
- class 的成员变量总是以其**声明**次序初始化

如果**某编译单元**内的某个non-local static对象的初始化动作使用了**另一编译单元内**的某个non-local static对象, 它所用到的这个对象可能**尚未被初始化**, 因为.C++ 对 “定义于不同编译单元内的 non-local static 对象” 的初始化次序并无明确定义

- static 对象, 其寿命从被构造出来直到**程序结束**为止 (无论在哪里构造, 都只会在程序结束时才被析构)
- **函数内**的 static 对象称为 local static 对象. 其它 static 对象称为 non-local static 对象, 例如 **namespace** , class 以及 file 作用域中被声明
- 对于 local static 对象，在其所属的函数被调用之前，该对象并不存在，即只有在**第一次调用**对应函数时，local static对象才被构造出来。

解决方案:

- 将每个non-local static对象搬到自己的**专属函数**内(该对象在此函数内被声明为static). 这些函数**返回一个reference** 指向它所含的对象.然后用户调用这些函数,而不直接指涉这些对象.换句话说,non-local static对象被local static对象替换了.Design Patterns迷哥迷姊们想必认出来了,这是Singleton模式的一个常见实现手法.

2 构造/ 析构/ 复制运算

05_了解c++背后编写并且调用哪些函数

- 编译器可以暗自为 class 创建 default构造函数、copy构造函数、copy assignment操作符, 以及析构函数.
- 当 对象 被创建以后使用 operator=, 调用的拷贝运算符函数, 但是当创建时使用 Myclass p2 = p1; 调用的是 **拷贝构造函数**
- 若成员变量中有**引用**，或者被 **const** 修饰等等，拷贝运算符不可被调用。

```
class person{
    string& m_name_;
    const int m_age_;
};

// main 函数
person p1("张三", 18);
person p2("李四", 20);
p1 = p2;//error!
```

06_若不想使用编译器自动生成的函数,就该明确拒绝

- 对于每一个独一无二的对象, 将权限设为 private : 可以防止调用拷贝构造函数, 但是 **成员函数** 和**友元函数**可以调用
- 解决方案: 建立一个父类，在**父类中定义 private** 拷贝函数，子类 (person 等等) 继承父类。因为子类不可以调用父类的 private 函数:

```
class uncopyable{
private:
    uncopyable(const uncopyable&);
    uncopyable operator=(const uncopyable&);
};
```

07_基类 声明 virtual 析构函数

- 多态把父类当作一个**接口**，用以处理子类对象：利用父类指针，指向一个在堆区开辟的子类对象。
- 当 父类指针删除时, 会有问题, 子类中的某些成分可能没有被销毁, 也就是 `局部销毁` 问题
- 解决方案: 基类声明 virtual 析构函数, 则在子类对象被销毁时，子类特有的成员会首先被销毁，然后会调用父类的析构函数进行销毁。
- 因为 virtual 函数会增加内存开销, 只有当class内含至少一个virtual函数,才为它声明virtual析构函数.
- 反之, 如果一个类的设计目的不是作为基类, 或者不是为了具备多态性, 就不应该 声明 virtual 析构函数

```
class Base{
public:
    Base();
    ......
    virtual ~Base();
};
```

08_别让异常逃离析构函数

- C+并不禁止析构函数吐出异常, 但它不鼓励你这样做.

09_决不在构造和析构过程中调用virtual 函数

- 在类的操作中，父类比子类先**构造**，子类也比父类先**析构**, (剥洋葱要从外面剥.), 所以在构造父类的时候，子类 的那一部分还未构造，在析构父类的时候，子类部分已经被销毁。

10_令 `operator=`返回一个 reference to *this

```
int x,y,z;
X=y=z=15;//赋值连锁形式
// 赋值采用右结合律,所以上述连锁赋值被解析为:
x=(y=(z=15));
```

11_在 operator= 中处理自我赋值

- 可能会为了**逻辑**有个空操作, `a[i] = a[j], *p = *q`
- 当涉及指针时, 回收空间, 会出现问题

```
class Bitmap{};
class Myclass{
    Bitmap *p;
public:
    Myclass& operator=(const Myclass &r){
        delete p;
        p = new Bitmap(*r.p);
        return *this;
    }
};
```

解决方案:

- 证同测试

```
Myclass& Myclass:: operator=(const Myclass &r){
    if(&r == this) {
        return *this;
    }
    delete p;
    // 不具备 "异常安全性"
    // 无论是 拷贝构造函数抛出异常, 还是 分配内存不足
    // 指针 都会指向一个 已经被删除的 Bitmap
    p = new Bitmap(*r.p);
    return *this;
}
```

- 复制之前删除

```
Myclass& Myclass:: operator=(const Myclass &r){
    Bitmap *rem = p;
    // 深拷贝, 又在其他的地方复制
    p = new Bitmap(*r.p);
    delete rem;
    return *this;
}
```

12_复制对象时勿忘每一个成分

- 因为面向对象有着封装性, 也就是, 父类变量通常存储在 private 里，子类不能访问父类 private 对象，让派生类的**拷贝构造**函数调用相应的基类的函数
- 确保:
  - 复制所有local成员变量
  - 调用所有 **基类** 内的适当的copying函数.
- 当你发现, 拷贝构造函数和 操作符有相同的行为时，不要
  - 令copy构造函数调用copy assignment:操作符------试图构造一个已经存在的对象
  - 令copy assignment操作符调用copy构造函数是不合理的------对一个尚未初始化的对象身上做

  - 个人认为, operator= 操作时, 已经**分配好了内存**, 但是copy 构造函数一开始并没有分配内存

```
class Animal{
public:
    int m_Age;
    Animal& operator=();
};

class Sheep :  public Animal {
public:
    Sheep& operator=(Sheep &r){
        // 复制行为
        Animal::operator(r);
    }
};
```

3 资源管理

13_以对象管理资源

- 在功能函数中, 一个过早的return语句可能会导致内存泄漏问题
- 以**专门的类**来管理资源的释放, 也就是**资源管理类**, 因为对象在离开作用域时会自动调用析构函数.

```
//------------资源管理类-------------
template<class T>
class Manage{
private:
    T *p;
public:
    ~Manage(){
        delete p;
    }
}
```

```
class Animal{};
class Sheep : public Animal {};

void fun(){
    // 以下语句是不对的
    Animal *p = animaFactory();
    // 修改为
    std::auto_ptr<Animal> p (animaFactory());
    return;
    delete p;
}
```

14_在资源管理类中小心 copying 行为(unfiinished)

15_在资源管理类中提供对原始资源(资源指针)的访问

- 管理类存放的是资源的指针，我们只有一个资源对象/ 指针
- 我们无法从管理类直接得到一个资源对象, 最好用`显式`转化或者`隐式转换`（自动类型转换）来得到一个资源对象：

```
template<class T>
class Manage{
private:
    T *p;
public:
    T* get(){
        // 显示的转化
        return p;
    }
    ~Manage(){
        delete p;
    }
    oparator T () const{
        // 隐式转化
        return p;
    }
}

// 函数签名
void fun(Sheep&);
// 创建资源管理类
Manage m (Animalfactory());
// 使用显示转化
fun(m.get());
// 使用隐式转化
fun(m);    <=>   fun(Animal(m))
```

16_成对使用new和 delete 时要采用相同的形式

- `new` 和 `delete` 涉及到构造析构的过程, 因此要成对使用, 数组的使用数组, 单个的对应单个
- 如果使用了 `new[]`, 而使用了 `delete`, 会之调用一次析构函数, 可能会出现 `内存泄漏`

17_以独立语句将 newed 对象 置入指针中

- 以独立语句将newed对象存储于(置入)智能指针内.如果不这样做,一旦异常被抛出,有可能导致难以察觉的资源泄漏.

```
fun1(shared_ptr<Complex>(new complex), fun2());
```

在调用 `fun1` 之前, 需要做 以下三件事

- 调用 `fun2`
- 执行 `new complex`
- 调用 `shared_ptr` 构造函数

C++中, 不会确定三者的关系, 但是如果

- 执行 `new complex`
- 调用 `fun2`
- 调用 `shared_ptr` 构造函数

会出现问题, 如果 `fun2` 调用异常, 返回的指针会遗失

4 函数设计

18_让接口容易被正确使用, 不易被误用

- 许多客户端错误可以因为导入新类型而获得预防.

```
class Date{
public:
    Date(int month, int day, int year);
};
// 修改为
class Date{
public:
    Date(Month month, Day day, Year year);
};
```

19_设计 class 犹如设计 type

身为 c++ 程序员, 你的许多时间主要用来扩张你的类型系统, 因此, 我们不只是 类的设计者, 还是 **type**的设计者

- 新对象应该如何被创建和销毁
- 对象的初始化和对象的复制该有什么样的差别
- 新的 type 对象如果被 值传递 意味着什么
- 什么是 type 的合法值
- 新的 type 需要什么样的转换

20_宁以 pass-by-reference-to-const 替换 pass-by-value

- **用 const 引用传递替换值传递**
- 函数传参时, C++默认 以 by value 方式传递对象.使用 const ref 不用复制, 节省内存和时间
- 值传递容易造成切割问题,
- 以上规则并不适用于**内置类型**, 以及STL的**迭代器**和**函数对象**. 对它们而言, pass-by-value往往比较适当.

```
class Base{
public:
    std::string name()const;
    virtual void display()const;
};

class Drived: public Base{
public:
    virtual void display()const;
};

void func(Base a){
    std:cout << a.name ();
    a.display();
}

// d 会被 构造为一个 Base 对象, 所有的 特化信息都被切除
// 调用的是 Base::display(), 而不是 Drived:: display()
Drived d;
func(d);
```

21_必须返回对象时, 不要妄想返回引用 (operator=)

- 一个 "必须返回新对象" 的函数的正确写法是: 就让那个函数返回一个新对象.
- 所谓reference只是个名称, 代表某个既有对象.任何时候看到一个reference声明式, 你都应该立刻问自己,它的另一个名称是什么?因为它一定只是某物的**另一个名称**.
- 新对象的途径有两个: 在 `stack` 空间或在 `heap` 空间创建

```
class Rational{
public:
    Rational(int numerator 0, int denominator 1);
    int numerator()const;
    int denominator ()const;
private:
};

// 错误, 返回local 对象
const Rational& operator*(const Rational &lhs, const Rational &rhs){
    Rational result(Ihs.n rhs.n, lhs.d rhs.d);
    return result;
}

// new 和 delete 应该成对使用
// 谁应该对着 delete 这个对象呢
// w = x* y* z;
// 会有 内存泄露 问题
const Rational& operator*(const Rational &lhs, const Rational &rhs){
    Rational *result = Rational(Ihs.n rhs.n, lhs.d rhs.d);
    return *result;
}

// 一个"必须返回新对象"的函数的正确写法是:
//
inline const Rational operator *(const Rational &lhs,const Rational &rhs){
    return Rational(1hs.n rhs.n,Ihs.d rhs.d);
}
```

22_将成员变量声明为 private

- **成员变量应该是private**, 从封装的角度看, 其实只有两种访问权限:private(提供封装)和其他(不提供封装)
- 如果在 public 接口内的每样东西都是**函数**, 那么客户不需要在打算访问class 成员时, 迷惑的试着记住是否应该使用 **括号**
- 实现**权限控制**, 如果你以函数取得或设定其值,你就可以实现出 **"不准访问"、"只读访问"以及"读写访问"**.见鬼了,你甚至可以实现"惟写访问",如果你想要的话:
- 可以在不改变功能的前提下, 改变其实现, (将成员变量隐藏在函数接口的背后,可以为"所有可能的实现"提供弹性.)

```
class AccessLevels{
public:
    int getReadOnlu()const{return readOnly;}
    void setReadWrite(int val){readWrite = val;}
    int getReadWrite()const{return readWrite;}
    void setWriteOnly(int val){ rwiteOnly = val;}
private:
    int noAccess;
    int readOnly;
    int readWrite;
    int writeOnly;

}
```

23_ 宁以 non-member, non-friend 替换 member 函数

- 对类变量的操作 **只能** 通过类**成员函数**实现，那么如果一个成员函数内部实现是调用其他的成员函数，则一个**非成员**函数也可以做到这样的效果：

```
class Myclass{
public:
    void func1();
    void func2();

    void func3(){
        func1();
        func2();
    }
};

// 替换为
void ues_out(const Myclass& p){
    p.func1();
    p.func2();
}
```

24_如果所有参数都需要类型转换, 请为此采用 non-member 函数（operator*)

- 想支持算术运算诸如加法、乘法等等, 你知道有理数相乘和Rational class有关,因此很自然地似乎该在Rational class内为有理数实现operator*
- 想要支持混合式算术运算, 使 operator* 成为一个non-member函数
- operator*可以完全藉由Rational的public接口完成, 无论何时如果你可以避免friend函数就该避免,

```
class Rational{
public:
    Rational(int numerator 0, int denominator 1);
```

```
    int numerator()const;
    int denominator ()const;
private:
};

Rational oneHalf(1, 2);
result=  oneHalf * 2;//很好
result=  2 * oneHalf; //错误!

// 以对应的函数形式重写上述两个式子, 就会发现, 问题在于, int 没有相应的 class
// 发生了隐式类型转换, 函数需要的是Rational, 用提供的2(int) 调用构造函数, 生成了一个 适当的
Rational对象
result= oneHalf.operator*(2);//很好
result =2.operator*(oneHalf); //错误!

// ----------------解决方案----------
class Rational{};
const Rational operator*(const Rational &l, const Rational &b);

Rational oneHalf(1, 2);
result=  oneHalf * 2; //很好
result=  2 * oneHalf; //通过了
```

25_考虑写一个不抛出异常的 swap 函数

问题:

- std::swap 函数是通过**拷贝**完成, 当类内**变量个数很多**或者有 `unique_tpr` 这样的**禁止复制的成员**时, 就要考虑写自己的 swap函数
- "以指针指向一个对象,内含真正数据"那种类型, 只需要交换 其**指针内容**就可以

解决:

- 当 std:swap 对你的类型效率不高时, 提供一个swap 成员函数,并确定这个函数**不抛出异常**.
- 如果你提供一个member swap,也 该提供一个non-member swap 用来调用前者.对于classes(而非templates),也请特化std:swap.
- 调用swap时应针对std:swap **使用 using声明式**, 然后调用swap并且不带任何"命名空间资格修饰".
- 为"用户定义类型"进行std templates全特化是好的,但千万**不要尝试**在std内**加入**某些对std而言全新的东西.

```
// std::swap()
template<typename T>
void swap(T &a,T &b){
    T temp(a);
    a =b;
    b = temp;
}

class Myclass{...};
```

```
// ----------------错误实现 1----------
void my_swap(Myclass& p1, Myclass& p2){
    // 因为 ptr 是private, 无法访问
    swap(p1.ptr, p2.ptr);
}

// ----------------修改实现-----------
// 在 类内部 自己实现 swap, 然后调用
class Myclass{
public:
    void my_swap(Myclass& p){
        swap(this->ptr, p.ptr);
    }
};

// 特化 std::swap()
namespace std{
    template<>
    void swap<Myclass> (Myclass& p1, Myclass& p2){
        p1.my_swap(p2);
    }
}//namespace std
```

问题

- 上述实现只能用于 class 而非是 template calss,
- C++只允许对class templates 偏特化, 在function templates身上偏特化是行不通的.
- 偏特化的含义是: 在 template 中, 当某个类型参数固定下来, 有很漂亮的实现方法.
- 一般而言,重载 function templates 没有问题,但std是个特殊的命名空间, 其管理规则也比较特殊.客户可以**全特化std内的templates**, 但不可以**添加**新的templates(或classes或functions或其他任何东西)到std里头. std的内容完全由C+标准委员会决定,标准委员会禁止我们膨胀那些已经声明好的东西.

```
template<typename T> class WidgetImpl {};
template<typename T> class Widget{};

// -----------不合法!-------
namespace std {
    template <typename T>
    void swap< Widget<T> >( Widget<T>&a, Widget<T>&b){
        a.swap(b);
    }
}//namespace

// -----------不合法!-------
// 不能 添加到 std 中
namespace std {
    template<typename T>
    void swap(Widget<T> &a, Widget<T>&b){
        a.swap(b);
    }
}

// 声明一个 non-member 版本的 函数
```

```
namespace mynamespace {
    template<typename T> class WidgetImpl {};
    template<typename T> class Widget{};

    template<typename T>
    void swap(Widget<T> &a, Widget<T>&b){
        a.swap(b);
    }
}
```

在调用 `swap(a, b)` 中发生了什么

- 如果T是 **Myclass** 并位于命名空间 **mynamespace** 内,编译器会使用"实参取决之查找规则"(argument-dependent lookup)找出 **mynamespace** 的swap.
- 如果**没有**T **专属**之swap存在,编译器就使用std内的swap,这得感谢using声明式让std:swap在函数内曝光.

```
template<typename T>void doSomething(T &obj1, T &obj2){
    using std:swap;//令std::swap在此函数内可用
    swap(obj1, obj2); //为T型对象调用最佳swap版本
}
```

Effective Modern C++

01_型别推导

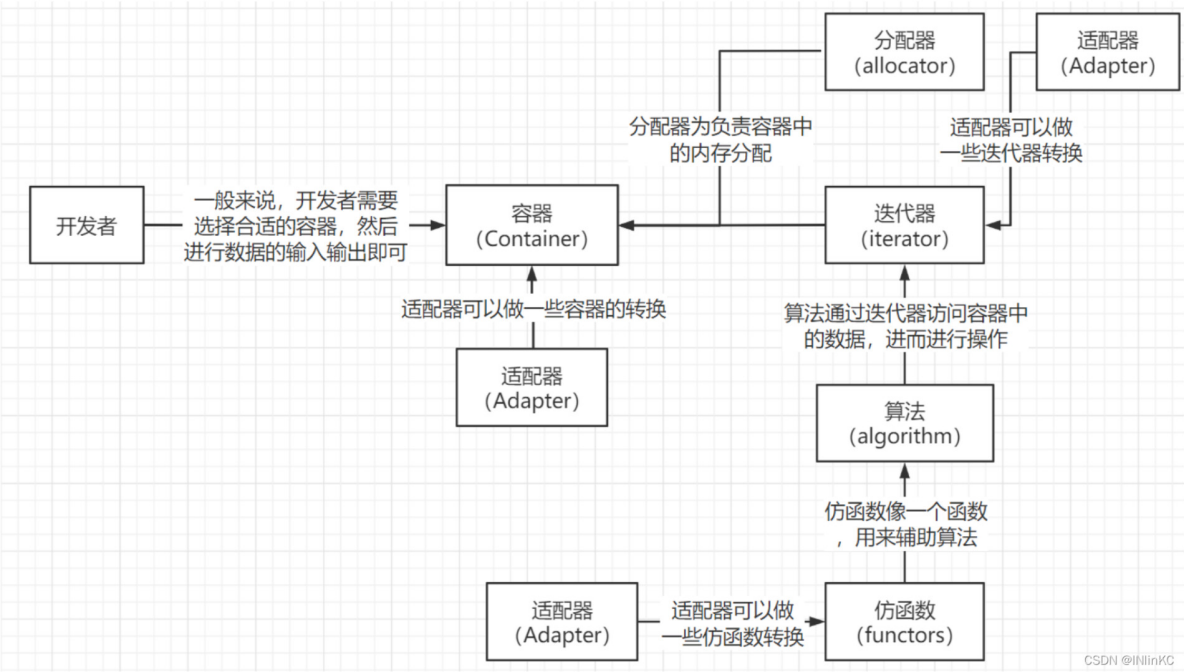
```
// paratype 通常包含了一些修饰词, 例如, const / & / 指针
template<typename T>
void fun(paratype T);
// 调用形式
fun(expr);

// 当 人们向引用型别的形参传入 const 时, 他们期望对象保持不可修改的属性

// 当 paratype 是指针或者引用, 但不是个万能引用
// 1. 若expr 有引用, 先将引用部分忽略
```

附录

Ascii表

ASCII 值	16 进制	控制字符	ASCII 值	16 进制	控制字符	ASCII 值	16 进制	控制字符	ASCII	16 进制	控制字符
0	00H	NUT	32	20H	(space)	64	40H	@	96	60H	、
1	01H	SOH	33	21H	!	65	41H	A	97	61H	a
2	02H	STX	34	22H	"	66	42H	B	98	62H	b
3	03H	ETX	35	23H	#	67	43H	C	99	63H	c
4	04H	EOT	36	24H	$	68	44H	D	100	64H	d
5	05H	ENO	37	25H	%	69	45H	E	101	65H	e
6	06H	ACK	38	26H	&	70	46H	F	102	66H	f
7	07H	BEL	39	27H	、	71	47H	G	103	67H	g
8	08H	BS	40	28H	(	72	48H	H	104	68H	h
9	09H	HT	41	29H	)	73	49H	I	105	69H	i
10	0AH	LF	42	2AH	*	74	4AH	J	106	6AH	j
11	0BH	VT	43	2BH	+	75	4BH	K	107	6BH	k
12	0CH	FF	44	2CH	、	76	4CH	L	108	6CH	l
13	0DH	CR	45	2DH	-	77	4DH	M	109	6DH	m
14	0EH	SO	46	2EH	.	78	4EH	N	110	6EH	n
15	0FH	SI	47	2FH	/	79	4FH	O	111	6FH	o
16	10H	DLE	48	30H	0	80	50H	P	112	70H	p
17	11H	DCI	49	31H	1	81	51H	Q	113	71H	q
18	12H	DC2	50	32H	2	82	52H	R	114	72H	r
19	13H	DC3	51	33H	3	83	53H	X	115	73H	s
20	14H	DC4	52	34H	4	84	54H	T	116	74H	t
21	15H	NAK	53	35H	5	85	55H	U	117	75H	u
22	16H	SYN	54	36H	6	86	56H	V	118	76H	v
23	17H	TB	55	37H	7	87	57H	W	119	77H	w
24	18H	CAN	56	38H	8	88	58H	X	120	78H	x
25	19H	EM	57	39H	9	89	59H	Y	121	79H	v
26	1AH	SUB	58	3AH	:	90	5AH	Z	122	7AH	z
27	1BH	ESC	59	3BH	;	91	5BH	[	123	7BH	{
28	1CH	FS	60	3CH	<	92	5CH	/	124	7CH	|
29	1DH	GS	61	3DH	=	93	5DH	]	125	7DH	}
30	1EH	RS	62	3EH	>	94	5EH	^	126	7EH	~
31	1FH	US	63	3FH	?	95	5FH	—	127	7FH	DEL

1 STL 介绍



> - 容器只定义了少部分操作，大部分操作由算法库完成。
> - 当容器套容器时，使用空格分隔开相邻的>符号，表示这是两个分开的符号。
> - capcity()表示的是不重新分配空间可以存储的元素数目，其中有没有使用的空间。
> - 我们要明白，指针和迭代器并不一样，区别具体表现在
>
> 1、迭代器不是指针，它只是表现得像指针的类模板，模仿了指针的很多操作而已。指针是狭义上的迭代器，迭代器是抽象的指针
>
> 2、迭代器返回的是对象的引用，而不是对象的值。所谓引用，就是把它和要引用的对象绑定在一起。所以对于cout操作，只能输出*迭代器，而不能直接输出迭代器

2.5 IO库(使用了继承关系的库)

作用

- 对控制窗口的IO
- 读写已经命名的文件
- 格式化内存中的数据

限制:

IO对象不可以**复制或者赋值**

2.5fstream

模式标志	描述
ios::in	读方式打开文件
ios:out	写方式打开文件
ios::trunc	如果此文件已经存在, 就会打开文件之前把文件长度截断为0
ios::app	尾部最加方式(在尾部写入)

模式标志	描述
ios::ate	文件打开后, 定位到文件尾
ios::binary	二进制方式(默认是文本方式)

```
#include<fstream>
ifstream fin;
ofstream fout;
//写文件
fout.open(filepath, moshi);
fout << 'a' << endl;
fout.close();

//读文件
fin.open(filepath);
string s;
fin >> s;
if(fin.eof()) break;
```

```
#include <fstream>
#include <string>
void test01()
{
    ifstream ifs;
    ifs.open("test.txt", ios::in);

    if (!ifs.is_open())
    {
        cout << "文件打开失败" << endl;
        return;
    }

    //第一种方式
    //char buf[1024] = { 0 };
    //while (ifs >> buf)
    //{
    //  cout << buf << endl;
    //}

    //第二种
    //char buf[1024] = { 0 };
    //while (ifs.getline(buf,sizeof(buf)))
    //{
    //  cout << buf << endl;
    //}

    //第三种
    //string buf;
    //while (getline(ifs, buf))
    //{
    //  cout << buf << endl;
    //}
```

```
    char c;
    while ((c = ifs.get()) != EOF)
    {
        cout << c;
    }

    ifs.close();


}

int main() {
    test01();
    system("pause");
    return 0;
}
```

2.5fstream

模式标志	描述
ios::in	读方式打开文件
ios:out	写方式打开文件
ios::trunc	如果此文件已经存在, 就会打开文件之前把文件长度截断为0
ios::app	尾部最加方式(在尾部写入)
ios::ate	文件打开后, 定位到文件尾
ios::binary	二进制方式(默认是文本方式)

```
#include<fstream>
ifstream fin;
ofstream fout;
//写文件
fout.open(filepath, moshi);
fout << 'a' << endl;
fout.close();

//读文件
fin.open(filepath);
string s;
fin >> s;
```

```
if(fin.eof()) break;
```