

特殊设置

CMake 设置

启用 **C++17** 标准，并开启编译选项：

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -lm -static ")
```

- -Wall ： 开启所有常见警告
- -lm ： 链接数学库
- -static ： 生成静态可执行文件，避免依赖运行环境

手动扩展栈空间

```
int main() {
    int size = 256 << 20; // 申请 256MB 栈空间
    __asm__ ("movq %0, %%rsp\n" :: "r"((char*)malloc(size) + size));
    // code
    exit(0); // 一定要这样写
}
```

pbds 库

```
#include <bits/stdc++.h>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
namespace pbds = __gnu_pbds;
template<typename T>
using rb_tree = pbds::tree<
    T,
    pbds::null_type,
    std::less<T>,
    pbds::rb_tree_tag,
    pbds::tree_order_statistics_node_update
>;
int main() {
    rb_tree<int> s;
    // 插入元素
    s.insert(5);
    // 删除元素
    s.erase(5);
    // 表示有多少元素 < 7 => 即 7 的排名
    int rank7 = s.order_of_key(7);
    // 找某个元素的排名（如果不存在则返回它应该插入的位置）
    int rank = s.order_of_key(10);
    // 取排名为 0 的元素（从 0 开始计）
    auto it = s.find_by_order(0);
    // 遍历：按排序后的顺序
    for (auto x : s) std::cout << " " << x;
    // 如果需要判断是否存在，可以检查：s.find(x) != s.end()
    return 0;
}
```

Python 快速读入

使用 C 层面的 I/O，从标准输入**快速读取一整行**：

```
import sys
def input():
    return sys.stdin.readline().strip()
```

生成随机数

简单版

```
int main() {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<long long> dis(0, 20000000000ULL);
    for (int i = 0; i < 20; ++i) {
        std::cout << dis(gen) << '\n';
    }
    return 0;
}
```

升级版

```
int main() {
    std::random_device rd;
    // 用多个 rd() 值通过 seed_seq 提供更好的种子熵
    std::seed_seq seq{rd(), rd(), rd(), rd(), rd(), rd(), rd(), rd()};
    std::mt19937_64 gen(seq); // 64-bit generator
    std::uniform_int_distribution<uint64_t> dis(0, 20000000000ULL);
    for (int i = 0; i < 20; ++i) {
        std::cout << dis(gen) << '\n';
    }
    return 0;
}
```

手写 hash 函数

```
struct SplitMix64 {
    static uint64_t mix(uint64_t x) {
        x += 0x9e3779b97f4a7c15ULL;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9ULL;
        x = (x ^ (x >> 27)) * 0x94d049bb133111ebULL;
        x = x ^ (x >> 31);
        return x;
    }
};

struct CustomHash {
    size_t operator()(uint64_t x) const {
        static const uint64_t FIXED_RANDOM =
            std::chrono::steady_clock::now().time_since_epoch().count();
        return (size_t)SplitMix64::mix(x + FIXED_RANDOM);
    }
};

struct pair_hash {
    template<class T1, class T2>
    size_t operator()(const std::pair<T1, T2> &p) const {
        size_t h1 = std::hash<T1>{}(p.first);
        size_t h2 = std::hash<T2>{}(p.second);
        return h1 ^ (h2 + 0x9e3779b97f4a7c15ULL + (h1 << 6) + (h1 >> 2));
    }
};

struct array_hash {
    template<class T, size_t N>
    size_t operator()(const std::array<T, N> &arr) const {
        size_t seed = 0;
        for (const auto &elem : arr) {
            seed ^= std::hash<T>{}(elem) +
                0x9e3779b97f4a7c15ULL + (seed << 6) + (seed >> 2);
        }
        return seed;
    }
};
```

Python

高精度

使用方法

```
from decimal import Decimal, localcontext, ROUND_HALF_UP, getcontext

# 设置全局精度默认为28位
getcontext().prec = 28

# 定义小数
a = Decimal(1) # 也可以用字符串
b = Decimal("7")

# 计算结果为28位小数
result = a / b
print(result)

# 截断并保留3位小数
result1 = result.quantize(Decimal('0.001'), rounding=ROUND_HALF_UP)
print(result1)

# 截断并保留 m 位小数
m = 4
precision_format = '0.' + '0' * m
result2 = result.quantize(Decimal(precision_format), rounding=ROUND_HALF_UP)
print(result2)

# 插入临时语块 不影响语块外的
with localcontext() as ctx:
    ctx.prec = 5 # 临时设置精度为5
    result3 = a / b
    print(result3)
print(result) # 结果仍为28位精度
```

舍入模式

- **ROUND_DOWN**: 直接截断，不进行四舍五入。

例如: 1.2345 保留两位小数 → 1.23

- **ROUND_UP**: 向远离零的方向舍入，任何非零部分都会进位。

例如: 1.2345 保留两位小数 → 1.24

- **ROUND_CEILING**: 向正无穷大方向舍入。

例如:

- 1.2345 保留两位小数 → 1.24
- -1.2345 保留两位小数 → -1.23

- **ROUND_FLOOR**: 向负无穷大方向舍入。

例如:

- 1.2345 保留两位小数 → 1.23
- -1.2345 保留两位小数 → -1.24

- **ROUND_HALF_UP**: 四舍五入，五进位。

例如: 1.235 保留两位小数 → 1.24

- **ROUND_HALF_DOWN**: 四舍五入，五舍去。

例如: 1.235 保留两位小数 → 1.23

- **ROUND_HALF_EVEN**: 遇到 5 时舍入到最接近的偶数，也叫“银行家舍入法”。

例如:

- 1.235 保留两位小数 → 1.24
- 1.225 保留两位小数 → 1.22

- **ROUND_05UP**: 只在舍去部分中遇到 0 或 5 时进位，其余情况舍去。

例如:

- 1.150 保留两位小数 → 1.16
- 1.149 保留两位小数 → 1.14

进制转换

```
# 定义十进制数
decimal_number = 10

# 这种方法本质是表达常量,这些变量的存储还是以10 进制
binary_number = 0b1010 # 二进制 1010
octal_number = 0o12    # 八进制 12
hex_number = 0xA       # 十六进制 A

# 这种方法生成的都是 str
print("二进制表示:", bin(decimal_number))    # 二进制表示 输出: 0b1010
print("八进制表示:", oct(decimal_number))    # 八进制表示 输出: 0o12
print("十六进制表示:", hex(decimal_number))  # 十六进制表示 输出: 0xa

# 如果不想要前两个字符,可以用 str 切片
print("二进制表示:", bin(decimal_number)[2:]) # 二进制表示输出: 1010

# 进行转换
# 将二进制、八进制和十六进制转换为十进制
binary_to_decimal = int('1010', 2) # 从二进制字符串转换
octal_to_decimal = int('12', 8)    # 从八进制字符串转换
hex_to_decimal = int('A', 16)      # 从十六进制字符串转换
# 可以将2-36 进制直接转换

print("二进制 '1010' 转换为十进制:", binary_to_decimal) # 输出: 10
print("八进制 '12' 转换为十进制:", octal_to_decimal)    # 输出: 10
print("十六进制 'A' 转换为十进制:", hex_to_decimal)     # 输出: 10
```

set

set 创建

1. 使用大括号创建集合

```
s = {1, 2, 3, 4}
print(s) # 输出: {1, 2, 3, 4}
```

2. 使用 set() 函数创建集合

使用 set() 函数从列表创建集合

```
lst = [1, 2, 3, 4]
s = set(lst)
print(s) # 输出: {1, 2, 3, 4}
```

使用 set() 函数从元组创建集合

```
tup = (1, 2, 3, 4)
s = set(tup)
print(s) # 输出: {1, 2, 3, 4}
```

使用 set() 函数从字符串创建集合 (去重)

```
s = set("hello")
print(s) # 输出: {'h', 'e', 'l', 'o'}, 重复的字母 'l' 只会出现一次
```

3. 创建空集合注意: 不能使用 {} 创建空集合, 因为 {} 是空字典的语法

```
empty_set = set()
print(empty_set) # 输出: set()
```

4. 使用集合推导式

```
s = {x * x for x in range(5)}
print(s) # 输出: {0, 1, 4, 9, 16}
```

带有条件的集合推导式

```
s = {x for x in range(10) if x % 2 == 0}
print(s) # 输出: {0, 2, 4, 6, 8}
```


set 操作

```
# 添加元素
s.add(5) # 向集合中添加元素 5

# 删除元素
s.remove(1) # 删除元素 1, 如果元素不存在会抛出 KeyError
s.discard(2) # 删除元素 2, 若元素不存在不会抛出异常

# 检查元素是否存在
if 3 in s:
    print("3 存在于集合中")

# 集合运算
a = {1, 2, 3}
b = {3, 4, 5}

# 并集
union = a | b # {1, 2, 3, 4, 5}
print("a 和 b 的并集:", union)

# 交集
intersection = a & b # {3}
print("a 和 b 的交集:", intersection)

# 差集
difference = a - b # {1, 2}
print("a 和 b 的差集:", difference)

# 对称差集
symmetric_difference = a ^ b # {1, 2, 4, 5}
print("a 和 b 的对称差集:", symmetric_difference)
```

set 遍历

方法 1: 使用 for 循环遍历集合

```
s = {1, 2, 3, 4, 5}
for elem in s:
    print(elem)
```

方法 2: 使用 enumerate() 函数遍历集合并获取索引

```
for index, elem in enumerate(s):
    print("Index:", index, "Element:", elem)
```

如果需要按顺序遍历,可以使用 sorted() 函数将集合转换为有序列表

```
for elem in sorted(s):
    print(elem)
```

dict

dict 创建

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

```
my_dict = dict(name="Alice", age=25, city="New York")
```

```
my_dict = dict([("name", "Alice"), ("age", 25), ("city", "New York")])
```

```
my_dict = {x: x * x for x in range(6)}
```

dict 操作

```
print("访问 name:", my_dict["name"])      # 直接访问, 键不存在会抛异常
print("使用 get:", my_dict.get("name"))    # 键不存在返回 None
print("get 默认值:", my_dict.get("addr", 0)) # 键不存在返回默认值
```

```
if "age" in my_dict:
    print("'age' 存在于字典中")
```

```
my_dict["age"] = 26          # 更新/添加
my_dict.update({"age": 27, "address": "123 Main St"}) # 批量更新
```

```
val = my_dict.pop("age", 0) # 删除并返回, 若不存在返回默认值
del my_dict["city"]         # del 删除
my_dict.clear()             # 清空
```

集合运算

```
a = {"x": 1, "y": 2}
b = {"y": 99, "z": 3}
```

并集 (右边覆盖左边的值)

```
print("a | b:", a | b)    # {'x': 1, 'y': 99, 'z': 3}
```

交集 (只保留两边都有的键, 值取左边)

```
print("a & b:", a & b)    # {'y': 2}
```

差集 (保留左边独有的键)

```
diff = {k: a[k] for k in a.keys() - b.keys()}
print("a - b:", diff)    # {'x': 1}
```

对称差集 (只保留不相同的键)

```
sym_diff = {k: (a.get(k) or b.get(k)) for k in a.keys() ^ b.keys()}
print("a ^ b:", sym_diff) # {'x': 1, 'z': 3}
```

dict 遍历

```
my_dict = {"name": "Alice", "age": 25, "city": "New York"}
```

```
# 遍历字典的键
```

```
for key in my_dict:  
    print(key)
```

```
# 遍历字典的值
```

```
for value in my_dict.values():  
    print(value)
```

```
# 遍历字典的键值对
```

```
for key, value in my_dict.items():  
    print(key, ":", value)
```

```
num_dict = {1: 4, 2: 3, 3: 2, 4: 1}
```

```
# 按键排序, 逆序排序使用 reverse=True
```

```
for key, value in sorted(num_dict.items(), reverse=False):  
    print(key, ":", value)
```

```
# 按值排序, 逆序排序使用 reverse=True
```

```
for key, value in sorted(num_dict.items(), key=lambda item: item[1], reverse=False):  
    print(key, ":", value)
```

deque

deque 创建

```
from collections import deque

# 空 deque
dq = deque()

# 从已有 list 构造
dq1 = deque([1, 2, 3, 4])

# 从已有 tuple 构造
dq2 = deque((10, 20, 30))

# 限定长度的 deque (超过长度会自动丢弃旧元素)
dq3 = deque([1, 2, 3, 4], maxlen=3)
# deque([2, 3, 4], maxlen=3)
```

deque 操作

```
dq = deque([1, 2, 3])

# 插入
dq.append(4)      # 尾部插入
dq.appendleft(0)  # 头部插入

# 访问 (支持随机下标访问和修改)
print(dq[0])      # 头元素
print(dq[-1])     # 尾元素
dq[1] = 99        # 修改下标为 1 的元素

# 删除
dq.pop()          # 删除尾部
dq.popleft()      # 删除头部
```

deque 遍历

```
dq = deque([10, 20, 30])
```

```
# 直接遍历
```

```
for x in dq:  
    print(x)
```

defaultdict

defaultdict 创建

```
from collections import defaultdict
```

```
# 默认值为 int (初始为 0)
```

```
dd1 = defaultdict(int)
```

```
# 默认值为 list (自动生成空列表)
```

```
dd2 = defaultdict(list)
```

```
# 默认值为 set (自动生成空集合)
```

```
dd3 = defaultdict(set)
```

```
# 默认值为 lambda (自定义默认值)
```

```
dd4 = defaultdict(lambda: None)
```

defaultdict 操作

```
dd = defaultdict(list)

# 添加/更新元素
dd["a"].append(1)
dd["a"].append(2)
dd["b"].append(3)

# 访问元素
print(dd["a"])    # [1, 2]
print(dd["b"])    # [3]
print(dd["c"])    # [] 自动生成空列表

# 删除元素
val = dd.pop("a") # 删除并返回

# 检查键是否存在
if "b" in dd:
    print("'b' 存在于 defaultdict")

# 批量更新
dd.update({"b": [4, 5], "c": [6]})
print("更新后:", dd)
```

defaultdict 遍历

```
dd = defaultdict(list, {"b": [4, 5], "c": [6]})
```

```
# 遍历键
```

```
for key in dd:  
    print("key:", key)
```

```
# 遍历值
```

```
for val in dd.values():  
    print("val:", val)
```

```
# 遍历键值对
```

```
for key, val in dd.items():  
    print("key:", key, "val:", val)
```

```
# 遍历时修改元素
```

```
for key in dd:  
    dd[key].append(0)  
print("修改后的 dd:", dd)
```

```
# 排序遍历
```

```
for key, val in sorted(dd.items()):  
    print(key, ":", val)
```