判断编译器和 C++ 标准版本

```
#include <iostream>

int main() {
#if defined(__clang__)
    std::cout << "Compiler: Clang\n";
#elif defined(__GNUC__)
    std::cout << "Compiler: GCC\n";
#elif defined(_MSC_VER)
    std::cout << "Compiler: MSVC\n";
#else
    std::cout << "Compiler: Unknown\n";
#endif

#if __cplusplus == 199711L
    std::cout << "Standard: C++98/03\n";
#elif __cplusplus == 201103L
    std::cout << "Standard: C++11\n";
#elif __cplusplus == 201402L
    std::cout << "Standard: C++14\n";
#elif __cplusplus == 201703L
    std::cout << "Standard: C++17\n";
#elif __cplusplus == 202002L
    std::cout << "Standard: C++20\n";
#elif __cplusplus == 202302L
    std::cout << "Standard: C++23\n";
#else
    std::cout << "Standard: Unknown (" << __cplusplus << ")\n";
#endif

    return 0;
}
```

头文件目录 C++11

```
// 容器（Containers）
#include <array>           // 固定大小数组容器
#include <deque>           // 双端队列容器
#include <forward_list>    // 单向链表容器
#include <list>            // 双向链表容器
#include <map>             // 有序键值对容器（红黑树）
#include <queue>           // 队列和优先队列容器
#include <set>             // 有序集合容器（红黑树）
#include <stack>           // 栈容器
#include <unordered_map>   // 无序键值对容器（哈希表）
#include <unordered_set>   // 无序集合容器（哈希表）
#include <vector>          // 动态数组容器
```

```cpp
// 算法与数值运算（Algorithms and Numeric Operations）
#include <algorithm>      // 提供通用算法，如排序、搜索、复制等
#include <bitset>         // 固定大小的位集合
#include <cmath>          // 数学函数，如 sin、cos、sqrt 等
#include <complex>        // 复数运算
#include <numeric>        // 数值算法，如累加、内积
#include <random>         // 随机数生成器
#include <valarray>       // 数值数组，优化数学运算

// 输入输出（Input/Output）
#include <fstream>        // 文件输入输出流
#include <iomanip>        // 格式化输入输出控制
#include <ios>            // 输入输出流基类
#include <iosfwd>         // 输入输出流前向声明
#include <iostream>       // 标准输入输出流（如 std::cin、std::cout）
#include <istream>        // 输入流支持
#include <ostream>        // 输出流支持
#include <sstream>        // 字符串流
#include <streambuf>      // 流缓冲区支持

// 并发与线程（Concurrency and Threads）
#include <atomic>         // 原子操作和内存同步
#include <condition_variable> // 线程同步的条件变量
#include <future>         // 异步操作，如 std::future 和 std::promise
#include <mutex>          // 互斥锁和线程同步
#include <thread>         // 线程支持

// 类型与内存管理（Types and Memory Management）
#include <cstddef>        // 基本类型定义，如 size_t、nullptr_t
#include <initializer_list> // 初始化列表支持
#include <limits>         // 数值类型的范围和限制
#include <memory>         // 智能指针和内存管理
#include <new>            // 动态内存分配（如 operator new）
#include <type_traits>    // 编译期类型检查和转换
#include <typeindex>      // 类型索引，用于类型标识

// C 兼容性（C Compatibility）
#include <cassert>        // 断言，用于调试
#include <cctype>         // 字符分类和转换函数
#include <cerrno>         // 错误码支持
#include <cfenv>          // 浮点环境访问，控制浮点异常和舍入模式
#include <cfloat>         // C 风格浮点数限制
#include <cinttypes>      // 整数类型格式化和扩展
#include <climits>        // C 风格整数限制
#include <csetjmp>        // 非局部跳转（C 风格）
#include <csignal>        // 信号处理（C 风格）
#include <cstdarg>        // 可变参数支持
#include <cstdio>         // C 风格输入输出
#include <cstdlib>        // C 风格通用工具，如内存分配、随机数
#include <cstring>        // C 风格字符串操作
#include <ctime>          // C 风格时间操作

// 本地化与字符串（Localization and Strings）
#include <clocale>        // C 风格本地化支持
#include <codecvt>        // 字符编码转换（C++17 部分废弃）
#include <cwchar>         // C 风格宽字符支持
#include <cwctype>        // C 风格宽字符分类
#include <locale>         // 本地化支持，如字符编码、货币格式
```

```
#include <string>          // 字符串操作

// 异常处理（Exception Handling）
#include <exception>       // 异常处理基类
#include <stdexcept>       // 标准异常类
#include <system_error>    // 系统错误码和异常

// 其他工具（Other Utilities）
#include <chrono>          // 时间和计时支持
#include <functional>      // 函数对象、绑定、lambda 支持
#include <iterator>        // 迭代器工具和基类
#include <ratio>           // 编译期有理数运算
#include <regex>           // 正则表达式支持
#include <scoped_allocator> // 作用域分配器
#include <tuple>           // 元组容器
#include <utility>         // 通用工具，如 std::pair、std::move
```

算法与数值运算（Algorithms and Numeric Operations）

`<algorithm>` 的核心功能与竞赛用途

1. 排序与查找（Sorting and Searching）

☑ **sort(first, last)** - 排序

```
vector<int> v{4, 2, 7, 1, 5};
// 对 v 升序排序
sort(v.begin(), v.end());
for (int x : v) cout << x << " ";  // 输出: 1 2 4 5 7
cout << endl;
```

☑ **stable_sort(first, last)** - 稳定排序

```
vector<pair<int, int>> v{{3, 1}, {1, 1}, {3, 2}};
// 对 v 按第一个元素升序排序，保持相等元素原顺序
stable_sort(v.begin(), v.end());
for (auto p : v) cout << "(" << p.first << "," << p.second << ") ";
// 输出: (1,1) (3,1) (3,2)
cout << endl;
```

☑ **lower_bound(first, last, value)** - 查找第一个不小于 value 的位置

```cpp
vector<int> v{1, 3, 3, 5, 7};
// 查找第一个不小于 3 的位置
auto it = lower_bound(v.begin(), v.end(), 3);
cout << *it << endl;  // 输出: 3
```

☑ **upper_bound(first, last, value)** - 查找第一个大于 value 的位置

```cpp
vector<int> v{1, 3, 3, 5, 7};
// 查找第一个大于 3 的位置
auto it = upper_bound(v.begin(), v.end(), 3);
cout << *it << endl;  // 输出: 5
```

☑ **binary_search(first, last, value)** - 判断是否存在某值

```cpp
vector<int> v{1, 3, 5, 7, 9};
// 判断是否存在 5
bool found = binary_search(v.begin(), v.end(), 5);
cout << boolalpha << found << endl;  // 输出: true
```

☑ **nth_element(first, nth, last)** - 使第 n 小元素归位（无序）

```cpp
vector<int> v{4, 2, 7, 1, 5};
// 使第 2 小元素归位，其前面是比它小的元素
nth_element(v.begin(), v.begin() + 2, v.end());
cout << v[2] << endl;  // 输出: 第 2 小元素（例如 4）
```

---

2. 最小/最大操作（Minimum/Maximum Operations）

☑ **min_element(first, last)** - 返回最小元素迭代器

```cpp
vector<int> v{4, 2, 7, 1, 5};
// 获取最小元素的迭代器
auto it = min_element(v.begin(), v.end());
cout << *it << endl;  // 输出: 1
```

☑ **max_element(first, last)** - 返回最大元素迭代器

```cpp
vector<int> v{4, 2, 7, 1, 5};
// 获取最大元素的迭代器
auto it = max_element(v.begin(), v.end());
cout << *it << endl;  // 输出: 7
```

☑ **minmax_element(first, last)** - 返回最小和最大元素迭代器对

```cpp
vector<int> v{4, 2, 7, 1, 5};
// 同时获取最小和最大元素的迭代器
auto p = minmax_element(v.begin(), v.end());
cout << *p.first << " " << *p.second << endl;  // 输出: 1 7
```

☑ **min(a, b) / max(a, b)** - 比较两个值

```cpp
int a = 5, b = 8;
// 比较两个值
cout << min(a, b) << " " << max(a, b) << endl;  // 输出: 5 8
```

☑ **minmax(a, b)** - 同时获取较小值和较大值

```cpp
int a = 5, b = 8;
// 同时获取 min 和 max
auto p = minmax(a, b);
cout << p.first << " " << p.second << endl;  // 输出: 5 8
```

3. 排列与组合（Permutation and Combination）

☑ **next_permutation(first, last)** - 生成下一个字典序排列

```cpp
vector<int> v{1, 2, 3};
// 生成下一个字典序排列
bool ok = next_permutation(v.begin(), v.end());
if (ok) {
    for (int x : v) cout << x << " ";  // 输出: 1 3 2
    cout << endl;
}
```

☑ **prev_permutation(first, last)** - 生成上一个字典序排列

```cpp
vector<int> v{3, 2, 1};
// 生成上一个字典序排列
bool ok = prev_permutation(v.begin(), v.end());
if (ok) {
    for (int x : v) cout << x << " ";  // 输出: 3 1 2
    cout << endl;
}
```

☑ **is_permutation(first1, last1, first2)** - 判断是否为排列关系

```cpp
vector<int> a{1, 2, 3};
vector<int> b{3, 1, 2};
// 检查 b 是否是 a 的排列
bool ok = is_permutation(a.begin(), a.end(), b.begin());
cout << boolalpha << ok << endl;  // 输出: true
```

4. 序列操作（Sequence Operations）

☑ **find(first, last, value)** - 查找等于指定值的元素

```
vector<int> v{1, 3, 5, 7, 9};
// 查找值为 5 的元素
auto it = find(v.begin(), v.end(), 5);
if (it != v.end()) cout << *it << endl;  // 输出: 5
```

☑ **find_if(first, last, pred)** - 查找满足条件的元素

```
vector<int> v{1, 3, 4, 6, 7};
// 查找第一个偶数
auto it = find_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
if (it != v.end()) cout << *it << endl;  // 输出: 4
```

☑ **find_if_not(first, last, pred)** - 查找不满足条件的元素

```
vector<int> v{2, 4, 6, 7, 8};
// 查找第一个不是偶数的元素
auto it = find_if_not(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
if (it != v.end()) cout << *it << endl;  // 输出: 7
```

☑ **count(first, last, value)** - 统计指定值的出现次数

```
vector<int> v{1, 2, 2, 3, 2};
// 统计 2 出现的次数
int cnt = count(v.begin(), v.end(), 2);
cout << cnt << endl;  // 输出: 3
```

☑ **count_if(first, last, pred)** - 统计满足条件的元素个数

```
vector<int> v{1, 3, 4, 6, 7};
// 统计偶数的个数
int cnt = count_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
cout << cnt << endl;  // 输出: 2
```

☑ **all_of(first, last, pred)** - 所有元素都满足条件

```
vector<int> v{2, 4, 6};
// 检查是否全部为偶数
bool res = all_of(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
cout << boolalpha << res << endl;  // 输出: true
```

☑ **any_of(first, last, pred)** - 存在元素满足条件

```cpp
vector<int> v{1, 3, 4, 5};
// 检查是否存在偶数
bool res = any_of(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
cout << boolalpha << res << endl;  // 输出: true
```

☑ **none_of(first, last, pred)** - 没有元素满足条件

```cpp
vector<int> v{1, 3, 5};
// 检查是否所有元素都不是偶数
bool res = none_of(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
cout << boolalpha << res << endl;  // 输出: true
```

☑ **unique(first, last)** - 移除连续重复元素（逻辑去重）

```cpp
vector<int> v{1, 1, 2, 2, 3, 3, 3};
// 去除连续重复元素
auto it = unique(v.begin(), v.end());
v.erase(it, v.end());
for (int x : v) cout << x << " ";  // 输出: 1 2 3
cout << endl;
```

☑ **reverse(first, last)** - 翻转元素顺序

```cpp
vector<int> v{1, 2, 3, 4};
// 翻转整个序列
reverse(v.begin(), v.end());
for (int x : v) cout << x << " ";  // 输出: 4 3 2 1
cout << endl;
```

☑ **rotate(first, middle, last)** - 左旋 middle 到前面

```cpp
vector<int> v{1, 2, 3, 4, 5};
// 将 v 旋转，使得 v[2] 成为新起点（左移2位）
rotate(v.begin(), v.begin() + 2, v.end());
for (int x : v) cout << x << " ";  // 输出: 3 4 5 1 2
cout << endl;
```

---

5. 堆操作（Heap Operations）

☑ **make_heap(first, last)** - 构建最大堆

```cpp
vector<int> v{3, 1, 4, 1, 5};
// 构建最大堆（默认使用 < 比较）
make_heap(v.begin(), v.end());
for (int x : v) cout << x << " ";  // 可能输出: 5 3 4 1 1（堆结构不唯一）
cout << endl;
```

☑ **push_heap(first, last)** - 插入元素并调整为堆

```
vector<int> v{4, 3, 1};
// 构建初始堆
make_heap(v.begin(), v.end());
// 添加新元素
v.push_back(5);
// 调整堆结构以包含新元素
push_heap(v.begin(), v.end());
for (int x : v) cout << x << " ";  // 输出: 5 4 1 3（可能有差异）
cout << endl;
```

☑ **pop_heap(first, last)** - 将堆顶移到末尾并调整堆

```
vector<int> v{5, 3, 4, 1};
// 假设 v 已为堆
pop_heap(v.begin(), v.end());  // 最大元素被移到末尾
int max_val = v.back();        // 最大值是最后一个元素
v.pop_back();                  // 移除最大元素
cout << max_val << endl;       // 输出: 5
```

☑ **sort_heap(first, last)** - 堆排序

```
vector<int> v{3, 1, 4, 1, 5};
make_heap(v.begin(), v.end());
// 对堆排序，得到升序序列
sort_heap(v.begin(), v.end());
for (int x : v) cout << x << " ";  // 输出: 1 1 3 4 5
cout << endl;
```

☑ **is_heap(first, last)** - 判断是否为堆

```
vector<int> v{5, 3, 4, 1};
// 检查 v 是否满足堆性质
bool res = is_heap(v.begin(), v.end());
cout << boolalpha << res << endl;  // 输出: true
```

☑ **is_heap_until(first, last)** - 查找堆范围终止点

```
vector<int> v{9, 7, 5, 1, 8};
// 找到最大前缀范围满足堆性质的位置
auto it = is_heap_until(v.begin(), v.end());
cout << distance(v.begin(), it) << endl;  // 输出: 4（前四个满足堆性质）
```

---

6. 集合操作（Set Operations）

☑ **set_union(first1, last1, first2, last2, result)** - 计算两个排序范围的并集

```
vector<int> a{1, 2, 4}, b{2, 3, 5}, result;
// 计算 a 和 b 的并集，结果存入 result（需先排序）
set_union(a.begin(), a.end(), b.begin(), b.end(), back_inserter(result));
for (int x : result) cout << x << " ";  // 输出: 1 2 3 4 5
cout << endl;
```

☑ **set_intersection(first1, last1, first2, last2, result)** - 计算两个排序范围的交集

```
vector<int> a{1, 2, 4}, b{2, 3, 4}, result;
// 计算 a 和 b 的交集，结果存入 result（需先排序）
set_intersection(a.begin(), a.end(), b.begin(), b.end(), back_inserter(result));
for (int x : result) cout << x << " ";  // 输出: 2 4
cout << endl;
```

☑ **set_difference(first1, last1, first2, last2, result)** - 计算 [first1, last1) 相对于 [first2, last2) 的差集

```
vector<int> a{1, 2, 3, 4}, b{2, 4}, result;
// 计算 a 相对于 b 的差集，结果存入 result（需先排序）
set_difference(a.begin(), a.end(), b.begin(), b.end(), back_inserter(result));
for (int x : result) cout << x << " ";  // 输出: 1 3
cout << endl;
```

☑ **set_symmetric_difference(first1, last1, first2, last2, result)** - 计算两个排序范围的对称差集

```
vector<int> a{1, 2, 3}, b{2, 3, 4}, result;
// 计算 a 和 b 的对称差集（即只出现在一个集合中的元素）
set_symmetric_difference(a.begin(), a.end(), b.begin(), b.end(),
back_inserter(result));
for (int x : result) cout << x << " ";  // 输出: 1 4
cout << endl;
```

☑ **merge(first1, last1, first2, last2, result)** - 合并两个排序范围，保持排序

```
vector<int> a{1, 3, 5}, b{2, 4, 6}, result;
// 合并 a 和 b 为一个有序序列（允许重复）
merge(a.begin(), a.end(), b.begin(), b.end(), back_inserter(result));
for (int x : result) cout << x << " ";  // 输出: 1 2 3 4 5 6
cout << endl;
```

☑ **includes(first1, last1, first2, last2)** - 判断一个排序范围是否包含另一个排序范围

```cpp
vector<int> a{1, 2, 3, 4, 5}, b{2, 4};
// 判断 b 是否是 a 的子集（需先排序）
bool res = includes(a.begin(), a.end(), b.begin(), b.end());
cout << boolalpha << res << endl;  // 输出: true
```

---

7. 其他实用算法

☑ **iota(first, last, value)** - 将范围填充为从 `value` 开始的递增序列

```cpp
vector<int> v(5);
// 使用 iota 填充 v 为 0, 1, 2, 3, 4
iota(v.begin(), v.end(), 0);
for (int x : v) cout << x << " ";  // 输出: 0 1 2 3 4
cout << endl;
```

☑ **swap(a, b)** - 交换两个元素的值

```cpp
int a = 3, b = 7;
// 交换 a 和 b 的值
swap(a, b);
cout << a << " " << b << endl;  // 输出: 7 3
```

☑ **swap_ranges(first1, last1, first2)** - 交换两个范围内的元素

```cpp
vector<int> v1{1, 2, 3};
vector<int> v2{4, 5, 6};
// 交换 v1 和 v2 中对应位置的元素
swap_ranges(v1.begin(), v1.end(), v2.begin());
for (int x : v1) cout << x << " ";  // 输出: 4 5 6
cout << endl;
for (int x : v2) cout << x << " ";  // 输出: 1 2 3
cout << endl;
```

☑ **transform(first, last, result, func)** - 对容器元素进行变换，结果存入新容器

```cpp
vector<int> v{1, 2, 3, 4};
vector<int> result;
// 将每个元素平方后插入到 result 中
transform(v.begin(), v.end(), back_inserter(result), [](int x) { return x * x;
});
for (int x : result) cout << x << " ";  // 输出: 1 4 9 16
cout << endl;
```

☑ **copy_if(first, last, result, pred)** - 复制符合条件的元素到新容器

```cpp
vector<int> v2{1, 2, 3, 4, 5};
vector<int> result2;
// 复制所有大于 3 的元素到 result2 中
copy_if(v2.begin(), v2.end(), back_inserter(result2), [](int x) { return x > 3;
});
for (int x : result2) cout << x << " ";  // 输出: 4 5
cout << endl;
```

☑ **iter_swap(a, b)** - 交换两个迭代器指向的元素

```cpp
vector<int> v3{1, 2, 3, 4};
// 交换第一个元素和第三个元素的位置
iter_swap(v3.begin(), v3.begin() + 2);  // v3 = {3, 2, 1, 4}
for (int x : v3) cout << x << " ";  // 输出: 3 2 1 4
cout << endl;
```

`<bitset>` 的核心功能与竞赛用途

在算法竞赛中，`<bitset>` 是 C++ 标准库中提供位操作功能的头文件，定义了 `std::bitset<N>` 模板类，用于表示和操作固定大小的位序列。`<bitset>` 针对位运算进行了高度优化，适合处理二进制数据、状态压缩、集合运算等场景。相比手动使用整数或数组实现位操作，`<bitset>` 提供了简洁的接口和高效的底层实现，尤其在处理大规模位集合或需要快速位运算的题目中表现优异。在算法竞赛中，`<bitset>` 是高频工具，常用于动态规划、图算法、组合数学和位运算相关问题。

在 C++11 中，`<bitset>` 的功能保持稳定，未引入显著新特性，但其接口完备，性能优越。以下是 `<bitset>` 头文件中 `std::bitset<N>` 类的所有方法和相关功能的详细说明，按功能分类，涵盖：

- **功能**：方法或操作的定义和行为。
- **参数与返回值**：输入参数类型和返回值类型。
- **竞赛用途**：在算法竞赛中的典型应用场景。
- **复杂度**：操作的计算复杂度（通常为 O(N/64) 或 O(1)，依赖硬件实现）。
- **注意事项**：使用时的潜在问题或优化建议。

---

1. 构造与初始化（Construction and Initialization）

这些方法用于创建和初始化 `std::bitset<N>` 对象，竞赛中用于设置初始状态或表示集合。

- `bitset<N>()`
  - **功能**：默认构造，创建大小为 `N` 的位集，所有位初始化为 0。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`std::bitset<N>` 对象。
  - **竞赛用途**：初始化空集合或状态数组。
  - **复杂度**：O(N/64)（依赖底层整数数组初始化）。
  - **注意事项**：`N` 必须为非负整数常量，编译期确定。
  - **示例**：

```
std::bitset<4> b; // 0000
```

- `bitset<N>(unsigned long val)`
  - **功能**：从无符号整数 `val` 构造位集，低 `N` 位初始化为 `val` 的二进制表示。
  - **参数与返回值**：
    - 参数：`unsigned long val`（初始值）。
    - 返回：`std::bitset<N>` 对象。
  - **竞赛用途**：从整数快速构造位集，如初始化状态或掩码。
  - **复杂度**：O(1)（通常为单一整数赋值）。
  - **注意事项**：若 `val` 超出 `N` 位，高位被忽略；C++11 后支持 `unsigned long long`（见下）。
  - **示例**：

```
std::bitset<4> b(5); // 0101 (5 = 101 in binary)
```

- `bitset<N>(unsigned long long val)`（C++11 扩展）
  - **功能**：从 `unsigned long long` 构造位集，低 `N` 位初始化为 `val` 的二进制表示。
  - **参数与返回值**：
    - 参数：`unsigned long long val`（初始值）。
    - 返回：`std::bitset<N>` 对象。
  - **竞赛用途**：支持更大范围的整数初始化，适合大型位集。
  - **复杂度**：O(1) 或 O(N/64)（依赖 `N` 和实现）。
  - **注意事项**：C++11 新增，确保编译器支持。
  - **示例**：

```
std::bitset<64> b(1ULL << 63); // 1 followed by 63 zeros
```

- `bitset<N>(const std::string& str, size_t pos = 0, size_t n = std::string::npos)`
  - **功能**：从字符串 `str` 的子串（从 `pos` 开始，长度 `n`）构造位集，`0` 表示 0，`1` 表示 1。
  - **参数与返回值**：
    - 参数：`const std::string& str`（源字符串），`size_t pos`（起始位置），`size_t n`（子串长度）。
    - 返回：`std::bitset<N>` 对象。
  - **竞赛用途**：从输入的二进制字符串初始化位集，如解析测试用例。
  - **复杂度**：O(n)（n 为字符串长度）。
  - **注意事项**：
    - 字符串需只含 `0` 和 `1`，否则抛出 `std::invalid_argument`。
    - 低位对应字符串高索引（右到左）。
  - **示例**：

```
std::bitset<4> b("1100"); // 1100
```

---

2. 位操作（Bit Manipulation）

这些方法用于访问、修改和查询单个位或整个位集，竞赛中用于状态管理和集合操作。

- `operator[](size_t pos)`
  - **功能**：访问或修改第 `pos` 位的值，返回 `std::bitset<N>::reference`（可读写）。
  - **参数与返回值**：
    - 参数：`size_t pos`（位索引，0 到 N-1）。
    - 返回：`std::bitset<N>::reference`（代理对象，支持赋值和读取）。
  - **竞赛用途**：
    - **状态更新**：设置或查询集合中的元素。
    - **动态规划**：修改状态位。
  - **复杂度**：O(1)。
  - **注意事项**：`pos` 需在 `[0, N)` 范围内，否则未定义行为。
  - **示例**：

```
std::bitset<4> b("0000");
b[1] = 1; // 0010
bool bit = b[1]; // true
```

- `test(size_t pos) const`
  - **功能**：检查第 `pos` 位是否为 1。
  - **参数与返回值**：
    - 参数：`size_t pos`（位索引）。
    - 返回：`bool`（true 表示 1，false 表示 0）。
  - **竞赛用途**：查询集合中元素是否存在（如状态检查）。
  - **Complexity**：O(1)。
  - **注意事项**：`pos` 超界抛出 `std::out_of_range`。
  - **示例**：

```
std::bitset<4> b("1010");
bool is_set = b.test(1); // true
```

- `set()`, `set(size_t pos, bool value = true)`
  - **功能**：
    - `set()`：将所有位设为 1。
    - `set(pos, value)`：将第 `pos` 位设为 `value`（默认 1）。
  - **参数与返回值**：
    - 参数：`size_t pos`（位索引），`bool value`（设置值）。
    - 返回：`std::bitset<N>&`（自身引用）。
  - **竞赛用途**：
    - **全集初始化**：表示所有元素存在。
    - **单点更新**：添加或移除集合元素。
  - **复杂度**：O(N/64)（全设），O(1)（单设）。
  - **注意事项**：`pos` 超界抛出 `std::out_of_range`。
  - **示例**：

```
std::bitset<4> b;
b.set(); // 1111
b.set(1, 0); // 1101
```

- `reset()`, `reset(size_t pos)`
  - **功能**:
    - `reset()`：将所有位设为 0。
    - `reset(pos)`：将第 `pos` 位设为 0。
  - **参数与返回值**:
    - 参数：`size_t pos`（位索引）。
    - 返回：`std::bitset<N>&`（自身引用）。
  - **竞赛用途**：清空集合或移除元素。
  - **复杂度**：O(N/64)（全清），O(1)（单清）。
  - **注意事项**：`pos` 超界抛出 `std::out_of_range`。
  - **示例**:

```
std::bitset<4> b("1111");
b.reset(); // 0000
b.set(1); b.reset(1); // 0000
```

- `flip()`, `flip(size_t pos)`
  - **功能**:
    - `flip()`：反转所有位（0 变 1，1 变 0）。
    - `flip(pos)`：反转第 `pos` 位。
  - **参数与返回值**:
    - 参数：`size_t pos`（位索引）。
    - 返回：`std::bitset<N>&`（自身引用）。
  - **竞赛用途**:
    - **状态翻转**：切换集合元素或状态。
    - **补集**：计算集合补集。
  - **Complexity**：O(N/64)（全翻），O(1)（单翻）。
  - **注意事项**：`pos` 超界抛出 `std::out_of_range`。
  - **示例**:

```
std::bitset<4> b("1010");
b.flip(); // 0101
b.flip(1); // 0111
```

---

3. 位运算（Bitwise Operations）

这些操作支持位集间的逻辑运算，竞赛中用于集合操作或状态压缩。

- `operator&`, `operator|`, `operator^`, `operator~`
  - **功能**:
    - `&`：按位与，`result[i] = v1[i] & v2[i]`。
    - `|`：按位或，`result[i] = v1[i] | v2[i]`。

    - `^`：按位异或，`result[i] = v1[i] ^ v2[i]`。
    - `~`：按位取反，`result[i] = !v[i]`。
  - **参数与返回值**：
    - 参数：`const std::bitset<N>&`（操作数）。
    - 返回：`std::bitset<N>`（新位集）。
  - **竞赛用途**：
    - **集合运算**：交集（`&`）、并集（`|`）、对称差（`^`）、补集（`~`）。
    - **状态压缩**：合并或比较状态。
  - **复杂度**：O(N/64)（硬件按字长操作）。
  - **注意事项**：操作数需同为 `bitset<N>`。
  - **示例**：

```
std::bitset<4> b1("1100"), b2("1010");
auto intersect = b1 & b2; // 1000
auto union_set = b1 | b2; // 1110
auto complement = ~b1; // 0011
```

- `operator&=`, `operator|=`, `operator^=`
  - **功能**：原地执行按位与、或、异或，修改当前位集。
  - **参数与返回值**：
    - 参数：`const std::bitset<N>&`（操作数）。
    - 返回：`std::bitset<N>&`（自身引用）。
  - **竞赛用途**：高效更新集合或状态。
  - **复杂度**：O(N/64)。
  - **注意事项**：同位运算，注意操作数类型。
  - **示例**：

```
std::bitset<4> b("1100");
b &= std::bitset<4>("1010"); // 1000
```

- `operator<<`, `operator>>`, `operator<<=`, `operator>>=`
  - **功能**：
    - `<<`, `>>`：左移或右移指定位数，空位补 0，返回新位集。
    - `<<=`, `>>=`：原地移位。
  - **参数与返回值**：
    - 参数：`size_t n`（移位数）。
    - 返回：`std::bitset<N>`（新位集）或 `std::bitset<N>&`（自身引用）。
  - **竞赛用途**：
    - **位掩码**：生成特定模式。
    - **状态转移**：动态规划中移位表示状态变化。
  - **复杂度**：O(N/64)。
  - **注意事项**：移位超出范围补 0。
  - **示例**：

```cpp
std::bitset<4> b("1100");
auto shifted = b << 1; // 1000
b >>= 1; // 0110
```

---

4. 查询与统计（Query and Statistics）

这些方法用于查询位集的状态或统计位信息，竞赛中用于分析集合。

- `all() const`（C++11）
  - **功能**：检查是否所有位均为 1。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`bool`（true 表示全 1）。
  - **竞赛用途**：验证集合是否为全集。
  - **复杂度**：O(N/64)。
  - **注意事项**：C++11 新增，空位集返回 true。
  - **示例**：

```cpp
std::bitset<4> b("1111");
bool is_all = b.all(); // true
```

- `any() const`
  - **功能**：检查是否至少一位为 1。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`bool`（true 表示存在 1）。
  - **竞赛用途**：检查集合是否非空。
  - **复杂度**：O(N/64)（最坏情况）。
  - **注意事项**：空位集返回 false。
  - **示例**：

```cpp
std::bitset<4> b("0100");
bool has_one = b.any(); // true
```

- `none() const`
  - **功能**：检查是否所有位均为 0。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`bool`（true 表示全 0）。
  - **竞赛用途**：验证集合是否为空。
  - **复杂度**：O(N/64)（最坏情况）。
  - **注意事项**：空位集返回 true。
  - **示例**：

```cpp
std::bitset<4> b("0000");
bool is_empty = b.none(); // true
```

- `count() const`
  - **功能**：返回位集中 1 的个数。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`size_t`（1 的计数）。
  - **竞赛用途**：
    - **集合大小**：计算集合中元素个数。
    - **汉明距离**：比较两个位集的差异。
  - **复杂度**：O(N/64)（依赖硬件 popcount 指令）。
  - **注意事项**：高效实现依赖编译器优化。
  - **示例**：

```cpp
std::bitset<4> b("1010");
size_t ones = b.count(); // 2
```

---

5. 转换与输入/输出（Conversion and I/O）

这些方法用于位集与整数、字符串的转换，以及输入输出，竞赛中用于调试或格式化。

- `to_ulong() const`
  - **功能**：将位集转换为 `unsigned long`，位集表示其二进制值。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`unsigned long`。
  - **竞赛用途**：将位集转换为整数，如状态索引。
  - **复杂度**：O(1) 或 O(N/64)。
  - **注意事项**：若位集超出 `unsigned long` 范围，抛出 `std::overflow_error`。
  - **示例**：

```cpp
std::bitset<4> b("1010");
unsigned long val = b.to_ulong(); // 10
```

```cpp

```

- `to_ullong() const`（C++11）
  - **功能**：将位集转换为 `unsigned long long`。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`unsigned long long`。
  - **竞赛用途**：支持更大位集的整数转换。
  - **复杂度**：O(1) 或 O(N/64)。

- **注意事项**：C++11 新增，注意范围限制。
- **示例**：

```
std::bitset<64> b("1" + std::string(63, '0'));
unsigned long long val = b.to_ullong(); // 2^63
```

- `to_string() const`
  - **功能**：将位集转换为二进制字符串，`0` 和 `1` 表示位值。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`std::string`。
  - **竞赛用途**：调试或输出位集状态。
  - **复杂度**：O(N)。
  - **注意事项**：高位在字符串左侧，低位在右侧。
  - **示例**：

```
std::bitset<4> b("1010");
std::string s = b.to_string(); // "1010"
```

- `operator<<`, `operator>>`
  - **功能**：将位集输出到流（`0` 和 `1` 序列）或从流输入。
  - **参数与返回值**：
    - 参数：`std::ostream&` 或 `std::istream&`，`std::bitset<N>&`。
    - 返回：流引用。
  - **竞赛用途**：调试或格式化答案输出。
  - **复杂度**：O(N)。
  - **注意事项**：输入需为 `0` 和 `1` 序列。
  - **示例**：

```
std::bitset<4> b("1010");
std::cout << b; // 1010
```

竞赛中的典型应用场景

1. **状态压缩动态规划**：
   - 使用 `bitset` 表示子集状态（如物品选择、点覆盖）。
   - 示例：旅行商问题（TSP），用 `bitset` 表示已访问城市集合，`set`、`test` 和逻辑运算用于状态转移。
2. **子集枚举**：
   - 用 `bitset` 枚举集合的子集，`count` 计算子集大小，`to_ullong` 用于索引。
   - 示例：枚举集合的所有子集，检查是否满足某些条件。
3. **位运算优化**：
   - 用 `bitset` 替代手动位运算，处理大规模二进制数据（如 64 位以上）。
   - 示例：快速计算集合交集或并集，`b1 & b2` 比循环更快。
4. **图算法**：

    - 用 `bitset` 表示邻接矩阵的行，加速连通性检查或最短路径计算。
    - 示例：Floyd 算法中用 `bitset` 表示可达性。
5. **快速计数**：
    - 用 `count` 统计 1 的数量，快速计算集合大小。
    - 示例：统计子集中元素的个数。
6. **位掩码查询**：
    - 用 `test` 或 `operator[]` 检查某位状态，`any` / `none` 判断集合状态。
    - 示例：检查某个点是否在子集中。

竞赛使用技巧

- **高效性**：`bitset` 的位操作通常比手动位运算快，尤其是处理超过 64 位的情况，因为它利用了底层硬件优化。
- **固定大小**：`bitset<N>` 需要编译期常量 `N`，适合固定规模问题（如状态压缩 DP 中 `N ≤ 64`）。
- **内存优化**：`bitset<N>` 占用 `ceil(N/8)` 字节，适合存储密集布尔值，内存效率高于 `vector<bool>`。
- **子集枚举**：结合 `bitset` 和位运算（如 `subset & b == subset`）快速枚举子集，复杂度 O(2^n)。
- **调试**：用 `to_string()` 输出 `bitset` 状态，便于调试复杂位运算。
- **大位数支持**：`bitset` 支持任意位数（如 `bitset<1000>`），适合大规模位运算，而 `unsigned long long` 限于 64 位。

注意事项

- **位索引**：`bitset` 的索引从 0 开始（低位），右边是低位，左边是高位。
- **越界检查**：`operator[]` 不检查越界，建议用 `test(pos)` 确保安全。
- **性能权衡**：对于小规模位运算（如 `N ≤ 64`），`unsigned long long` 可能更简单且性能相当；大规模用 `bitset`。
- **竞赛限制**：确认题目输入规模，`bitset` 的 `N` 需足够大但不能过大以免超内存。

`<cmath>` 的核心函数与竞赛用途

在算法竞赛中，`<cmath>` 是 C++ 标准库中提供数学运算功能的头文件，包含一系列用于浮点数和整数的数学函数，涵盖基本运算、幂和指数、三角函数、双曲函数、取整、绝对值、特殊函数以及误差处理等。这些函数在解决几何、数值计算、概率、统计、物理模拟等问题时非常有用，尤其在需要高精度或复杂数学运算的场景。相比 `<algorithm>` 或 `<bitset>`，`<cmath>` 的使用频率稍低，但其提供的函数在特定题目中往往是关键。

`<cmath>` 中的函数主要针对浮点数（`double`、`float`、`long double`），C++11 还为部分函数（如 `abs`、`fmax`）添加了整数重载。以下是按功能分类的详细列表，每个函数包括：

- **功能**：函数的数学定义和行为。
- **参数与返回值**：输入参数类型和返回值类型。
- **竞赛用途**：在算法竞赛中的典型应用场景。
- **复杂度**：函数的计算复杂度（通常为 O(1)，但可能因实现而异）。
- **注意事项**：使用时的潜在问题或优化建议。

1. 基本运算（Basic Arithmetic）

这些函数处理基本的数值操作，常用于距离计算、误差处理或模运算。

- `abs(x)`
  - **功能**：返回 `x` 的绝对值。
  - **参数与返回值**：`x` 可为 `int`, `long`, `long long`, `float`, `double`, `long double`；返回同类型绝对值。
  - **竞赛用途**：计算曼哈顿距离、差值或误差（如比较浮点数）、处理负数场景。例如，在几何问题中计算两点间距离 `|x1 - x2|`。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 支持整数重载，避免使用 `std::fabs` 仅为浮点数的情况；注意整数溢出（如 `abs(INT_MIN)`）。
  - **示例**：`double d = std::abs(-3.14); // 返回 3.14`
- `fabs(x)`
  - **功能**：返回浮点数 `x` 的绝对值（专为浮点数设计）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：与 `abs` 类似，但明确用于浮点数，常见于需要高精度绝对值的场景，如误差计算。
  - **复杂度**：O(1)。
  - **注意事项**：优先使用 `abs`（C++11 更通用），除非题目明确要求 `fabs` 或兼容旧代码。
  - **示例**：`double f = std::fabs(-2.71); // 返回 2.71`
- `fmod(x, y)`
  - **功能**：返回 `x` 除以 `y` 的浮点余数，定义为 `x - n*y`，其中 `n` 是 `x/y` 的整数部分。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：处理浮点模运算，如周期性计算（角度归一化到 [0, 2π]）、浮点数循环问题。
  - **复杂度**：O(1)。
  - **注意事项**：`y` 不能为 0；结果符号与 `x` 相同；与 `remainder` 不同，`fmod` 不遵循 IEEE 标准。
  - **示例**：`double rem = std::fmod(10.5, 3.0); // 返回 1.5`
- `remainder(x, y)`
  - **功能**：返回 `x` 除以 `y` 的余数，遵循 IEEE 754 标准，`n` 是最接近 `x/y` 的整数（四舍五入）。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：需要标准余数的场景，如信号处理、数值分析或严格模运算。
  - **复杂度**：O(1)。
  - **注意事项**：`y` 不能为 0；结果可能为负；竞赛中较少见，但精度要求高时有用。
  - **示例**：`double rem = std::remainder(10.5, 3.0); // 可能返回 -1.5`
- `fmax(x, y)`, `fmin(x, y)`
  - **功能**：返回 `x` 和 `y` 的最大值或最小值，正确处理 NaN（返回非 NaN 值）。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`（C++11 也支持整数）；返回同类型。
  - **竞赛用途**：比较浮点数或整数，处理边界条件、优化问题或区间选择。例如，选择最大收益或最小成本。
  - **复杂度**：O(1)。
  - **注意事项**：比 `std::max`/`std::min`（`<algorithm>`）更适合浮点数，因其处理 NaN；整数场景用 `std::max` 更常见。
  - **示例**：`double m = std::fmax(3.14, 2.71); // 返回 3.14`

2. 幂和指数（Power and Exponential Functions）

这些函数处理幂、对数和指数运算，竞赛中常用于几何、概率或增长模型。

- `pow(x, y)`
  - **功能**：返回 `x` 的 `y` 次幂，`x^y`。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：计算幂次，如面积（`pow(r, 2)`）、体积、指数增长或组合计数。常用于公式计算。
  - **复杂度**：O(log y)（实现依赖）。
  - **注意事项**：对于整数幂，快速幂算法（`O(log y)`）通常比 `pow` 更快；注意 `x` 为负时 `y` 需为整数。
  - **示例**：`double p = std::pow(2.0, 3.0); // 返回 8.0`
- `sqrt(x)`
  - **功能**：返回 `x` 的平方根。
  - **参数与返回值**：`x` 为非负 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：计算欧几里得距离（如两点间距离 `sqrt((x1-x2)^2 + (y1-y2)^2)`）、几何问题（如圆面积）或二次方程解。
  - **复杂度**：O(1)（硬件优化）。
  - **注意事项**：`x` 必须 ≥ 0；对于大数值，使用 `hypot` 更安全以避免溢出。
  - **示例**：`double s = std::sqrt(16.0); // 返回 4.0`
- `cbrt(x)`
  - **功能**：返回 `x` 的立方根。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：三维几何（如立方体边长）、数值分析或方程求解。
  - **复杂度**：O(1)。
  - **注意事项**：支持负数输入（如 `cbrt(-8) = -2`）；竞赛中较少见。
  - **示例**：`double c = std::cbrt(27.0); // 返回 3.0`
- `hypot(x, y)`
  - **功能**：返回 `sqrt(x*x + y*y)`，避免中间溢出。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：计算二维欧几里得距离，广泛用于几何问题（如判断点到原点的距离）。
  - **复杂度**：O(1)。
  - **注意事项**：比手动 `sqrt(x*x + y*y)` 更安全，适合大数值；C++11 引入。
  - **示例**：`double d = std::hypot(3.0, 4.0); // 返回 5.0`
- `exp(x)`
  - **功能**：返回 `e` 的 `x` 次幂，`e^x`。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：概率计算（正态分布）、指数增长/衰减、模拟退火算法中的接受概率。
  - **复杂度**：O(1)。
  - **注意事项**：大 `x` 可能导致溢出；对于小 `x`，可用泰勒展开近似。
  - **示例**：`double e = std::exp(1.0); // 返回 e ≈ 2.71828`
- `exp2(x)`
  - **功能**：返回 2 的 `x` 次幂，`2^x`。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：二进制相关计算，如位运算优化或快速幂的浮点版本。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；整数场景用位运算（如 `1 << n`）更快。
  - **示例**：`double e2 = std::exp2(3.0); // 返回 8.0`

- `expm1(x)`
  - **功能**：返回 `e^x - 1`，对小 `x` 提供更高精度。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：高精度概率计算或数值分析，较少见。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；仅在高精度需求时使用。
  - **示例**：`double em = std::expm1(0.01); // 返回 ≈ 0.010050167`
- `log(x)`
  - **功能**：返回 `x` 的自然对数（底为 `e`）。
  - **参数与返回值**：`x` 为正 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：对数复杂度分析、熵计算、数值优化或概率密度函数。
  - **复杂度**：O(1)。
  - **注意事项**：`x` 必须 > 0；大 `x` 可能损失精度。
  - **示例**：`double l = std::log(2.71828); // 返回 ≈ 1.0`
- `log10(x)`
  - **功能**：返回 `x` 的以 10 为底的对数。
  - **参数与返回值**：`x` 为正 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：计算数字位数（如 `floor(log10(n)) + 1`）、科学计数法处理。
  - **复杂度**：O(1)。
  - **注意事项**：`x` 必须 > 0；竞赛中比 `log` 更常用，因十进制直观。
  - **示例**：`double l10 = std::log10(100.0); // 返回 2.0`
- `log2(x)`
  - **功能**：返回 `x` 的以 2 为底的对数。
  - **参数与返回值**：`x` 为正 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：计算二进制位数、树高、位运算优化。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；整数场景可用位运算（如 `__builtin_clz`）替代。
  - **示例**：`double l2 = std::log2(8.0); // 返回 3.0`
- `log1p(x)`
  - **功能**：返回 `ln(1 + x)`，对小 `x` 提供更高精度。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`（`x > -1`）；返回同类型。
  - **竞赛用途**：高精度对数计算，极少见。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；仅在高精度场景使用。
  - **示例**：`double lp = std::log1p(0.01); // 返回 ≈ 0.00995033`

3. 三角函数（Trigonometric Functions）

三角函数处理角度和周期性计算，广泛用于几何和物理模拟。

- `sin(x)`, `cos(x)`, `tan(x)`
  - **功能**：返回 `x`（弧度）的正弦、余弦、正切。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：几何问题（计算角度、向量旋转）、物理模拟（波形、周期运动）、圆形轨迹计算。
  - **复杂度**：O(1)（硬件优化）。
  - **注意事项**：输入为弧度，需转换角度（如 `rad = deg * PI / 180`）；注意 `tan` 在奇数倍 π/2 处未定义。
  - **示例**：`double s = std::sin(PI / 2); // 返回 ≈ 1.0`

- `asin(x)`, `acos(x)`, `atan(x)`
  - **功能**：返回反正弦、反余弦、反正切（结果为弧度）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`（`asin`, `acos` 要求 `|x| ≤ 1`）；返回同类型。
  - **竞赛用途**：计算角度，如三角形边长求角度、向量夹角、极坐标转换。
  - **复杂度**：O(1)。
  - **注意事项**：`asin`, `acos` 输入需在 [-1, 1]；`atan` 返回值在 [-π/2, π/2]。
  - **示例**：`double a = std::acos(0.0); // 返回 π/2`
- `atan2(y, x)`
  - **功能**：返回点 `(x, y)` 的极角（弧度），考虑象限。
  - **参数与返回值**：`x`, `y` 为 `float`, `double`, `long double`；返回同类型（[-π, π]）。
  - **竞赛用途**：计算向量角度、极坐标转换、机器人导航、几何排序（如极角排序）。
  - **复杂度**：O(1)。
  - **注意事项**：比 `atan(y/x)` 更可靠，因其处理所有象限；当 `x = y = 0` 时未定义。
  - **示例**：`double angle = std::atan2(1.0, 1.0); // 返回 π/4`

4. 双曲函数（Hyperbolic Functions）

双曲函数在竞赛中较少见，但可能出现在特定数学或物理问题中。

- `sinh(x)`, `cosh(x)`, `tanh(x)`
  - **功能**：返回 `x` 的双曲正弦、余弦、正切。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：双曲几何、物理模拟（如悬链线）、某些积分计算。
  - **复杂度**：O(1)。
  - **注意事项**：大 `x` 可能导致溢出；竞赛中极少见。
  - **示例**：`double sh = std::sinh(1.0); // 返回 ≈ 1.175`
- `asinh(x)`, `acosh(x)`, `atanh(x)`
  - **功能**：返回反双曲正弦、余弦、正切。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`（`acosh` 要求 `x ≥ 1`, `atanh` 要求 `|x| < 1`）；返回同类型。
  - **竞赛用途**：反双曲函数的逆运算，极少见，可能用于特定数学推导。
  - **复杂度**：O(1)。
  - **注意事项**：输入范围严格，需验证；竞赛中几乎不用。
  - **示例**：`double ah = std::asinh(1.175); // 返回 ≈ 1.0`

5. 取整与绝对值（Rounding and Absolute Value）

这些函数处理浮点数的取整或绝对值，常用于数值处理或网格划分。

- `ceil(x)`
  - **功能**：返回不小于 `x` 的最小整数。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：向上取整，如计算最小资源量、网格划分、区间覆盖。
  - **复杂度**：O(1)。
  - **注意事项**：返回值为浮点数，需显式转换为整数（如 `int(std::ceil(x))`）。
  - **示例**：`double c = std::ceil(3.14); // 返回 4.0`
- `floor(x)`
  - **功能**：返回不大于 `x` 的最大整数。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
  - **竞赛用途**：向下取整，如计算最大完整单位、数组索引、时间戳划分。

    - **复杂度**：O(1)。
    - **注意事项**：同 `ceil`，返回浮点数，需转换。
    - **示例**：`double f = std::floor(3.14); // 返回 3.0`
- `trunc(x)`
    - **功能**：返回 `x` 的整数部分（向零取整）。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
    - **竞赛用途**：截断小数部分，如提取整数部分或格式化输出。
    - **复杂度**：O(1)。
    - **注意事项**：与 `floor` 不同，负数向零取整（如 `trunc(-3.14) = -3.0`）。
    - **示例**：`double t = std::trunc(3.14); // 返回 3.0`
- `round(x)`
    - **功能**：返回 `x` 的四舍五入值（到最近整数）。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
    - **竞赛用途**：处理需要整数近似的情况，如统计、输出格式化、概率取整。
    - **复杂度**：O(1)。
    - **注意事项**：C++11 引入；对于 `.5`，向远离零的方向取整（如 `round(2.5) = 3.0`）。
    - **示例**：`double r = std::round(3.6); // 返回 4.0`
- `lround(x)`, `llround(x)`
    - **功能**：返回 `x` 的四舍五入值，分别为 `long` 和 `long long` 类型。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `long` 或 `long long`。
    - **竞赛用途**：直接获取整数结果，适合需要整数输出的场景。
    - **复杂度**：O(1)。
    - **注意事项**：C++11 引入；注意溢出（如大浮点数可能超出 `long` 范围）。
    - **示例**：`long lr = std::lround(3.6); // 返回 4`
- `nearbyint(x)`
    - **功能**：返回 `x` 的最近整数，按当前舍入模式（默认 FE_TONEAREST）。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
    - **竞赛用途**：需要特定舍入模式的场景，极少见。
    - **复杂度**：O(1)。
    - **注意事项**：依赖浮点环境设置，竞赛中几乎不用。
    - **示例**：`double ni = std::nearbyint(3.6); // 返回 4.0`
- `rint(x)`, `lrint(x)`, `llrint(x)`
    - **功能**：返回 `x` 的最近整数（按当前舍入模式），分别为浮点、`long`、`long long` 类型。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回相应类型。
    - **竞赛用途**：类似 `round`，但依赖舍入模式，极少见。
    - **复杂度**：O(1)。
    - **注意事项**：与 `nearbyint` 类似，竞赛中优先用 `round` 或 `lround`。
    - **示例**：`double ri = std::rint(3.6); // 返回 4.0`

6. 特殊函数（Special Functions）

这些函数处理高级数学函数，在竞赛中较少见，但在统计、概率或组合数学中有用。

- `erf(x)`, `erfc(x)`
    - **功能**：返回误差函数 `erf(x) = 2/√π ∫[0,x] e^(-t^2) dt` 和补误差函数 `erfc(x) = 1 - erf(x)`。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回同类型。
    - **竞赛用途**：正态分布的累积分布函数（CDF）计算，如概率统计问题。
    - **复杂度**：O(1)（数值逼近）。

- **注意事项**：C++11 引入；竞赛中极少见，仅在高级统计题目中出现。
- **示例**：`double e = std::erf(1.0); // 返回 ≈ 0.842701`

- `tgamma(x)`
  - **功能**：返回伽马函数 `Γ(x)`，对于正整数 `n`，`Γ(n) = (n-1)!`。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`（非负或非整数）；返回同类型。
  - **竞赛用途**：组合数学、阶乘推广、贝塔函数计算。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；`x` 不能为非正整数；大 `x` 可能溢出。
  - **示例**：`double g = std::tgamma(5.0); // 返回 24.0 (4!)`
- `lgamma(x)`
  - **功能**：返回伽马函数的自然对数 `ln|Γ(x)|`。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`（非负或非整数）；返回同类型。
  - **竞赛用途**：高精度组合计算，避免 `tgamma` 溢出。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；更适合大 `x` 场景。
  - **示例**：`double lg = std::lgamma(5.0); // 返回 ln(24.0)`

7. 浮点数分类与误差处理（Floating-Point Classification and Error Handling）

这些函数检查浮点数状态或处理误差，竞赛中用于精度控制或异常处理。

- `isfinite(x)`
  - **功能**：检查 `x` 是否为有限值（非无穷或 NaN）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `bool`。
  - **竞赛用途**：验证计算结果有效性，防止无穷大或 NaN 导致错误。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；常用于调试或边界检查。
  - **示例**：`bool fin = std::isfinite(1.0 / 0.0); // 返回 false`
- `isinf(x)`
  - **功能**：检查 `x` 是否为无穷大（正或负）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `bool`。
  - **竞赛用途**：检测溢出，如除以零或大数值运算。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；与 `isfinite` 互补。
  - **示例**：`bool inf = std::isinf(1.0 / 0.0); // 返回 true`
- `isnan(x)`
  - **功能**：检查 `x` 是否为 NaN（非数值）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `bool`。
  - **竞赛用途**：检测无效运算（如 `0.0 / 0.0` 或 `sqrt(-1)`）。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；NaN 不等于自身，需用 `isnan` 检查。
  - **示例**：`bool nan = std::isnan(0.0 / 0.0); // 返回 true`
- `isnormal(x)`
  - **功能**：检查 `x` 是否为正常值（非零、非无穷、非 NaN、非次正常）。
  - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `bool`。
  - **竞赛用途**：验证浮点数是否适合进一步计算，极少见。
  - **复杂度**：O(1)。
  - **注意事项**：C++11 引入；竞赛中几乎不用。

    - **示例**：`bool norm = std::isnormal(1.0); // 返回 true`
- `fpclassify(x)`
    - **功能**：返回 `x` 的浮点分类（`FP_INFINITE`, `FP_NAN`, `FP_ZERO`, `FP_SUBNORMAL`, `FP_NORMAL`）。
    - **参数与返回值**：`x` 为 `float`, `double`, `long double`；返回 `int`。
    - **竞赛用途**：详细检查浮点数状态，极少见。
    - **复杂度**：O(1)。
    - **注意事项**：C++11 引入；优先用 `isinf`, `isnan` 等单独函数。
    - **示例**：`int cls = std::fpclassify(1.0 / 0.0); // 返回 FP_INFINITE`

8. 数学常量（非标准，但常手动定义）

`<cmath>` 本身不直接提供常量，但竞赛中常使用以下常量（需手动定义或依赖编译器）：

- `M_PI`：π ≈ 3.141592653589793（非标准，部分编译器支持）。
- `M_E`：e ≈ 2.718281828459045（非标准）。
- **竞赛用途**：几何计算（圆面积、角度转换）、指数计算。
- **定义方法**：`const double PI = std::acos(-1.0); const double E = std::exp(1.0);`
- **注意事项**：C++11 不保证 `M_PI` 存在，建议用 `acos(-1.0)` 确保兼容。

竞赛中的典型应用场景

1. **几何问题**：
    - **距离计算**：`hypot`, `sqrt` 计算欧几里得距离；`abs` 计算曼哈顿距离。
    - **角度计算**：`sin`, `cos`, `atan2` 处理向量夹角、极角排序、三角形角度。
    - **网格划分**：`ceil`, `floor` 将浮点坐标映射到整数网格。
    - 示例：计算三角形面积（`0.5 * a * b * sin(C)`）或点到直线的距离。
2. **数值优化**：
    - **误差处理**：`abs`, `fmax`, `fmin` 比较结果；`isnan`, `isinf` 验证有效性。
    - **取整**：`round`, `lround` 处理近似值；`trunc` 提取整数部分。
    - 示例：最小化误差或最大化收益。
3. **概率与统计**：
    - **指数函数**：`exp`, `expm1` 计算概率密度或接受概率（如模拟退火）。
    - **对数**：`log`, `log10` 处理熵、概率或位数计算。
    - **误差函数**：`erf` 用于正态分布（极少见）。
    - 示例：计算正态分布的累积概率。
4. **物理模拟**：
    - **周期性**：`sin`, `cos` 模拟波形或圆周运动。
    - **增长/衰减**：`exp`, `pow` 模拟指数变化。
    - 示例：模拟弹簧运动或衰减过程。
5. **组合数学**：
    - **伽马函数**：`tgamma`, `lgamma` 处理阶乘推广或贝塔函数。
    - **对数**：`log` 用于高精度组合计数。
    - 示例：计算大阶乘的对数和。

竞赛使用技巧

- **精度控制**：浮点运算存在误差，竞赛中常用 `1e-9` 或 `1e-12` 作为阈值比较浮点数（如 `std::abs(a - b) < 1e-9`）。输出时用 `std::fixed` 和 `std::setprecision` 控制小数位。
- **避免溢出**：使用 `hypot` 替代 `sqrt(x*x + y*y)` 防止中间溢出；大数值用 `log` 或 `lgamma` 避免 `pow` 或 `tgamma` 溢出。
- **快速替代**：整数幂用快速幂算法替代 `pow`；整数位数用循环或位运算替代 `log10`；整数绝对值用条件语句替代 `abs`。
- **角度转换**：三角函数使用弧度，竞赛中常需手动转换（`deg = rad * 180 / PI`，`rad = deg * PI / 180`）。
- **编译器兼容**：`M_PI`，`M_E` 非标准，建议用 `std::acos(-1.0)` 和 `std::exp(1.0)` 定义常量。
- **浮点验证**：用 `isinf`，`isnan` 检查结果有效性，防止未定义行为（如除零）。
- **整数支持**：C++11 扩展了 `abs`，`fmax`，`fmin` 等支持整数，减少类型转换。
- **高精度需求**：对于小值运算，优先用 `expm1`，`log1p` 提高精度；大阶乘用 `lgamma` 避免溢出。

注意事项

- **输入范围**：严格检查函数输入，如 `sqrt(x)` 要求 `x ≥ 0`，`log(x)` 要求 `x > 0`，`asin(x)` 要求 `|x| ≤ 1`。
- **性能**：`<cmath>` 函数为 O(1)，但浮点运算比整数运算慢，竞赛中若可用整数运算优先考虑。
- **题目限制**：检查题目是否允许浮点误差，部分题目要求精确整数解，可能需避免 `<cmath>`。
- **浮点误差**：浮点比较需用误差阈值；多步计算可能累积误差，尽量简化运算。
- **C++11 特性**：C++11 引入了 `hypot`，`log2`，`round`，`erf` 等，确认比赛编译器支持（如 g++ -std=c++11）。

竞赛中的高频函数

以下是 `<cmath>` 中在算法竞赛中最常用的函数（基于题目类型）：

- **几何**：`hypot`，`sqrt`，`sin`，`cos`，`atan2`，`abs`.
- **数值处理**：`ceil`，`floor`，`round`，`fmax`，`fmin`，`log10`.
- **概率/优化**：`exp`，`log`，`erf`（少见）。
- **组合数学**：`tgamma`，`lgamma`（罕见）。

`<complex>` 的核心功能与竞赛用途

在算法竞赛中，`<complex>` 是 C++ 标准库中提供复数运算的头文件，定义了 `std::complex<T>` 模板类（`T` 通常为 `float`，`double`，或 `long double`），用于表示和操作复数。`<complex>` 在竞赛中的使用频率相对较低，远不如 `<algorithm>`、`<bitset>` 或 `<cmath>` 常见，但在涉及几何变换（如旋转、傅里叶变换）、多项式运算、信号处理或特定数学问题时非常有用。相比手动实现复数运算，`<complex>` 提供了高效、可靠的接口，且底层实现利用硬件优化，性能优异。

注意，`<complex>` 在 C++11 中功能稳定，未引入显著新特性，但其高效性和便捷性使其在特定题目中不可替代。

`std::complex<T>` 表示一个复数，形式为 `a + bi`，其中 `a` 是实部（real part），`b` 是虚部（imaginary part），`i` 是虚数单位（`i^2 = -1`）。`<complex>` 提供了构造、访问、运算、数学函数和规范计算等功能，适合处理复数相关的数学和几何问题。以下是按功能分类的详细列表，每个功能包括：

- **功能**：函数或操作的数学定义和行为。
- **参数与返回值**：输入参数类型和返回值类型。

- **竞赛用途**：在算法竞赛中的典型应用场景。
- **复杂度**：操作的计算复杂度（通常为 O(1) 或依赖底层 `<cmath>`）。
- **注意事项**：使用时的潜在问题或优化建议。

1. 构造与访问（Construction and Access）

这些功能用于创建和访问复数的实部和虚部，竞赛中常用于初始化或提取复数信息。

- `complex<T>(real, imag)`
  - **功能**：构造复数 `real + imag*i`。
  - **参数与返回值**：`real`, `imag` 为 `T`（`float`, `double`, `long double`）；返回 `std::complex<T>`。
  - **竞赛用途**：初始化复数表示二维点、向量或信号。例如，在几何问题中用复数表示平面上的点 `(x, y)`。
  - **复杂度**：O(1)。
  - **注意事项**：确保 `T` 为浮点类型，整数类型（如 `complex<int>`）不支持数学函数。
  - **示例**：`std::complex<double> z(3.0, 4.0); // 表示 3 + 4i`
- `real()`, `imag()`
  - **功能**：返回复数的实部或虚部。
  - **参数与返回值**：无参数；返回 `T` 类型值。
  - **竞赛用途**：提取复数的实部或虚部，如获取点的 x, y 坐标或信号的幅度。
  - **复杂度**：O(1)。
  - **注意事项**：`real()`, `imag()` 返回值，`z.real()` 是只读，修改需用 `std::real(z)`, `std::imag(z)` 或直接赋值。
  - **示例**：`double x = z.real(); // 返回 3.0`
- `std::real(z)`, `std::imag(z)`
  - **功能**：自由函数，返回复数 `z` 的实部或虚部。
  - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `T`。
  - **竞赛用途**：同 `real()`, `imag()`，但作为自由函数更灵活，适合泛型代码。
  - **复杂度**：O(1)。
  - **注意事项**：优先使用成员函数，除非需要函数式编程风格。
  - **示例**：`double y = std::imag(z); // 返回 4.0`

2. 基本运算（Basic Arithmetic Operations）

这些运算符支持复数的加减乘除，竞赛中常用于几何变换或多项式运算。

- `operator+`, `operator-`, `operator*`, `operator/`
  - **功能**：执行复数的加法、减法、乘法、除法。
    - 加法：`(a + bi) + (c + di) = (a+c) + (b+d)i`
    - 乘法：`(a + bi)(c + di) = (ac-bd) + (ad+bc)i`
    - 除法：`(a + bi)/(c + di) = ((ac+bd)/(c^2+d^2)) + ((bc-ad)/(c^2+d^2))i`
  - **参数与返回值**：操作数为 `std::complex<T>` 或标量 `T`；返回 `std::complex<T>`。
  - **竞赛用途**：
    - **几何**：复数乘法表示旋转（`z * std::complex<double>(cos θ, sin θ)` 旋转 θ 弧度）。
    - **多项式**：复数运算用于多项式根的计算或傅里叶变换。
    - **向量**：加减法表示平移，乘法表示缩放和旋转。
  - **复杂度**：O(1)（加减法）、O(1)（乘除法，含多次浮点运算）。
  - **注意事项**：除法需确保分母非零（`c^2 + d^2 ≠ 0`）；浮点误差可能累积，需控制精度。

    - **示例**：`auto w = z + std::complex<double>(1.0, 2.0); // (3+4i) + (1+2i) = 4+6i`
- `operator+=`, `operator-=`, `operator*=`, `operator/=`
    - **功能**：原地执行加、减、乘、除。
    - **参数与返回值**：操作数为 `std::complex<T>` 或标量 `T`；返回 `std::complex<T>&`。
    - **竞赛用途**：高效更新复数状态，如在循环中累积变换或计算多项式。
    - **复杂度**：O(1)。
    - **注意事项**：同基本运算，确保除法分母非零。
    - **示例**：`z *= std::complex<double>(0.0, 1.0); // 旋转 90 度`
- `operator==`, `operator!=`
    - **功能**：比较两个复数是否相等或不等。
    - **参数与返回值**：操作数为 `std::complex<T>`；返回 `bool`。
    - **竞赛用途**：验证复数相等，如检查几何变换结果或多项式根。
    - **复杂度**：O(1)。
    - **注意事项**：浮点比较可能受误差影响，建议用 `std::abs(z1 - z2) < eps` 替代。
    - **示例**：`bool eq = (z == std::complex<double>(3.0, 4.0));`

3. 数学函数（Mathematical Functions）

这些函数扩展了 `<cmath>` 的功能到复数域，竞赛中用于高级计算。

- `std::abs(z)`
    - **功能**：返回复数 `z` 的模，`sqrt(real(z)^2 + imag(z)^2)`。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `T`。
    - **竞赛用途**：计算复数的幅度，如几何中计算点到原点的距离或信号强度。
    - **复杂度**：O(1)（依赖 `std::sqrt`）。
    - **注意事项**：等价于 `std::hypot(real(z), imag(z))`，避免溢出。
    - **示例**：`double mag = std::abs(z); // 返回 5.0 for z = 3+4i`
- `std::arg(z)`
    - **功能**：返回复数 `z` 的辐角（主值），`atan2(imag(z), real(z))`。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `T`（弧度，[-π, π]）。
    - **竞赛用途**：计算向量角度，如极角排序、几何旋转或信号相位。
    - **复杂度**：O(1)（依赖 `std::atan2`）。
    - **注意事项**：当 `z = 0` 时未定义；返回弧度，需转换角度。
    - **示例**：`double theta = std::arg(z); // 返回 atan2(4, 3) ≈ 0.927`
- `std::norm(z)`
    - **功能**：返回复数 `z` 的模平方，`real(z)^2 + imag(z)^2`。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `T`。
    - **竞赛用途**：计算模平方，避免 `sqrt` 开销，如能量计算或快速比较距离。
    - **复杂度**：O(1)。
    - **注意事项**：比 `abs(z)^2` 更快，但可能溢出；不等价于线性代数中的范数。
    - **示例**：`double n = std::norm(z); // 返回 25.0 for z = 3+4i`
- `std::conj(z)`
    - **功能**：返回复数 `z` 的共轭，`real(z) - imag(z)*i`。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
    - **竞赛用途**：计算共轭，用于多项式运算、信号处理或几何反射。
    - **复杂度**：O(1)。
    - **注意事项**：简单操作，适合快速变换。
    - **示例**：`auto c = std::conj(z); // 返回 3-4i for z = 3+4i`
- `std::polar(rho, theta)`

    - **功能**：返回极坐标形式的复数，`rho * (cos(theta) + i*sin(theta))`。
    - **参数与返回值**：`rho`，`theta` 为 `T`（模和弧度）；返回 `std::complex<T>`。
    - **竞赛用途**：从极坐标构造复数，如生成旋转向量或周期信号。
    - **复杂度**：O(1)（依赖 `std::cos`，`std::sin`）。
    - **注意事项**：`theta` 为弧度；`rho` 通常非负。
    - **示例**：`auto p = std::polar(5.0, PI/4); // 返回 √2/2*(1+i)`
- `std::proj(z)`
    - **功能**：返回复数 `z` 在黎曼球面上的投影。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
    - **竞赛用途**：处理无穷复数，极少见，可能用于理论数学问题。
    - **复杂度**：O(1)。
    - **注意事项**：C++11 引入；竞赛中几乎不用。
    - **示例**：`auto pr = std::proj(z); // 通常返回 z 本身`
- `std::exp(z)`
    - **功能**：返回复数 `z` 的指数，`e^z = e^(real(z)) * (cos(imag(z)) + i*sin(imag(z)))`。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
    - **竞赛用途**：傅里叶变换、信号处理或复分析，计算复指数。
    - **复杂度**：O(1)（依赖 `std::exp`，`std::cos`，`std::sin`）。
    - **注意事项**：大实部可能导致溢出；竞赛中用于高级题目。
    - **示例**：`auto e = std::exp(std::complex<double>(0.0, PI)); // 返回 -1+0i`
- `std::log(z)`
    - **功能**：返回复数 `z` 的自然对数，`ln|z| + i*arg(z)`（主值）。
    - **参数与返回值**：`z` 为 `std::complex<T>`（非零）；返回 `std::complex<T>`。
    - **竞赛用途**：复分析、多项式对数计算、信号相位处理。
    - **复杂度**：O(1)（依赖 `std::log`，`std::arg`）。
    - **注意事项**：`z ≠ 0`；虚部在 [-π, π]。
    - **示例**：`auto l = std::log(z); // 返回 ln(5) + i*atan2(4,3)`
- `std::pow(z, w)`，`std::pow(z, n)`
    - **功能**：返回复数 `z` 的 `w` 次幂或标量 `n` 次幂，`e^(w*log(z))`。
    - **参数与返回值**：`z`，`w` 为 `std::complex<T>` 或 `n` 为 `T`；返回 `std::complex<T>`。
    - **竞赛用途**：复数幂运算，如多项式求幂或几何缩放。
    - **复杂度**：O(1)（依赖 `std::log`，`std::exp`）。
    - **注意事项**：`z ≠ 0`；整数幂 `pow(z, n)` 更高效。
    - **示例**：`auto p = std::pow(z, 2.0); // 返回 (3+4i)^2 = -7+24i`
- `std::sqrt(z)`
    - **功能**：返回复数 `z` 的平方根（主值，虚部非负）。
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
    - **竞赛用途**：复数平方根计算，如二次方程解或几何变换。
    - **复杂度**：O(1)。
    - **注意事项**：支持负数和复数，返回主值。
    - **示例**：`auto s = std::sqrt(std::complex<double>(-1.0, 0.0)); // 返回 0+1i`
- `std::sin(z)`，`std::cos(z)`，`std::tan(z)`
    - **功能**：返回复数 `z` 的正弦、余弦、正切。
        - `sin(z) = (e^(iz) - e^(-iz))/(2i)`
        - `cos(z) = (e^(iz) + e^(-iz))/2`
    - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。

- **竞赛用途**：信号处理、傅里叶变换、复分析中的周期函数。
- **复杂度**：O(1)（依赖 `std::exp`）。
- **注意事项**：支持复数输入，竞赛中较少见。
- **示例**：`auto s = std::sin(std::complex<double>(0.0, 1.0));`

- `std::asin(z)`, `acos(z)`, `atan(z)`
  - **功能**：返回复数 `z` 的反正弦、反余弦、反正切。
  - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
  - **竞赛用途**：复分析、反函数计算，极少见。
  - **复杂度**：O(1)（依赖 `std::log`）。
  - **注意事项**：支持任意复数，竞赛中几乎不用。
  - **示例**：`auto as = std::asin(std::complex<double>(1.0, 0.0));`
- `std::sinh(z)`, `cosh(z)`, `tanh(z)`
  - **功能**：返回复数 `z` 的双曲正弦、余弦、正切。
  - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
  - **竞赛用途**：复分析、双曲几何，极少见。
  - **Complexity**：O(1)（依赖 `std::exp`）。
  - **注意事项**：竞赛中几乎不用。
  - **示例**：`auto sh = std::sinh(std::complex<double>(1.0, 0.0));`
- `std::asinh(z)`, `acosh(z)`, `atanh(z)`
  - **功能**：返回复数 `z` 的反双曲正弦、余弦、正切。
  - **参数与返回值**：`z` 为 `std::complex<T>`；返回 `std::complex<T>`。
  - **竞赛用途**：反双曲函数计算，极少见。
  - **复杂度**：O(1)（依赖 `std::log`）。
  - **注意事项**：竞赛中几乎不用。
  - **示例**：`auto ash = std::asinh(std::complex<double>(1.0, 0.0));`

4. 输入/输出（Input/Output）

这些功能支持复数的流操作，竞赛中用于调试或格式化输出。

- `operator<<`, `operator>>`
  - **功能**：将复数输出为 `(real,imag)` 或从流输入。
  - **参数与返回值**：`std::ostream`, `std::istream` 和 `std::complex<T>`；返回流引用。
  - **竞赛用途**：调试复数计算结果或格式化输出答案。
  - **复杂度**：O(1)（依赖流操作）。
  - **注意事项**：输出格式固定为 `(real,imag)`，需包含 `<iostream>`。
  - **示例**：`std::cout << z; // 输出 (3,4)`

竞赛中的典型应用场景

1. **几何问题**：
   - **点和向量**：用 `std::complex<double>` 表示二维点 `(x, y)`，加减法表示平移，乘法表示旋转和缩放。
   - **旋转**：`z * std::complex<double>(cos θ, sin θ)` 旋转 θ 弧度，`z * std::polar(1.0, θ)` 更简洁。
   - **距离和角度**：`std::abs(z)` 计算到原点的距离，`std::arg(z)` 计算极角。
   - 示例：计算多边形面积、点集凸包、极角排序。
2. **傅里叶变换**：
   - **信号处理**：`std::exp(2πi*k/N)` 生成旋转因子，`std::sin`, `std::cos` 用于复数变换。

    - 示例：快速傅里叶变换（FFT）实现多项式乘法。
3. **多项式运算**：
    - **根和幂**：`std::sqrt`, `std::pow` 计算复数根或幂。
    - 示例：求解二次方程的复数根或多项式估值。
4. **信号处理**：
    - **相位和幅度**：`std::arg` 计算相位，`std::abs` 计算幅度。
    - 示例：分析周期信号或复数域滤波。
5. **复分析**：
    - **复函数**：`std::exp`, `std::log` 用于复数域分析。
    - 示例：计算复对数或复指数（高级数学题目）。

示例代码

以下是一个示例程序，展示 `<complex>` 在算法竞赛中的常见用法，涵盖几何（点旋转、距离、角度）、多项式（复数幂）、傅里叶变换因子生成和复数运算。代码模拟一个竞赛场景，计算点旋转后的坐标、两点距离、多边形面积和 FFT 旋转因子。

```
#include <complex>
#include <iostream>
#include <iomanip>
#include <vector>

int main() {
    // 定义常量
    const double PI = std::acos(-1.0);

    // 1. 几何：点表示和旋转
    using Complex = std::complex<double>;
    Complex p1(3.0, 4.0), p2(1.0, 1.0); // 点 (3,4) 和 (1,1)
    std::cout << "Point p1: " << p1 << '\n';
    std::cout << "Point p2: " << p2 << '\n';

    // 计算两点距离
    double distance = std::abs(p1 - p2);
    std::cout << "Distance between p1 and p2: " << std::fixed <<
std::setprecision(6)
              << distance << '\n';

    // 旋转 p1 90 度（乘以 i）
    Complex rotated = p1 * Complex(0.0, 1.0);
    std::cout << "p1 rotated 90°: " << rotated << '\n';

    // 计算 p1 的极角（度）
    double angle = std::arg(p1) * 180.0 / PI;
    std::cout << "Angle of p1 (degrees): " << angle << '\n';

    // 2. 多边形面积（用复数表示顶点）
    std::vector<Complex> polygon = {Complex(0, 0), Complex(3, 0), Complex(0,
4)};
    double area = 0.0;
    for (size_t i = 0; i < polygon.size(); ++i) {
        Complex z1 = polygon[i], z2 = polygon[(i + 1) % polygon.size()];
        area += std::imag(std::conj(z1) * z2); // 叉积公式
    }
    area = std::abs(area) / 2.0;
```

```
    std::cout << "Polygon area: " << area << '\n';

    // 3. 多项式：计算复数平方
    Complex z = Complex(3.0, 4.0);
    Complex z_squared = std::pow(z, 2.0);
    std::cout << "Square of " << z << ": " << z_squared << '\n';

    // 4. 傅里叶变换：生成旋转因子
    int N = 4; // FFT 点数
    std::vector<Complex> roots(N);
    for (int k = 0; k < N; ++k) {
        roots[k] = std::exp(Complex(0.0, 2.0 * PI * k / N)); // e^(2πi*k/N)
    }
    std::cout << "FFT roots of unity:\n";
    for (const auto& root : roots) {
        std::cout << root << '\n';
    }

    // 5. 复数运算验证
    Complex sum = p1 + p2;
    Complex prod = p1 * p2;
    std::cout << "Sum: " << sum << '\n';
    std::cout << "Product: " << prod << '\n';

    // 6. 模和共轭
    double norm = std::norm(z);
    Complex conj = std::conj(z);
    std::cout << "Norm of " << z << ": " << norm << '\n';
    std::cout << "Conjugate of " << z << ": " << conj << '\n';

    return 0;
}
```

竞赛使用技巧

- **几何优化**：用 `std::complex<double>` 替代手动 `(x, y)` 坐标，简化旋转（`z * std::polar(1.0, θ)`）和距离（`std::abs(z)`）计算。
- **精度控制**：复数运算涉及浮点计算，比较时用误差阈值（如 `std::abs(z1 - z2) < 1e-9`）；输出用 `std::fixed` 和 `std::setprecision`。
- **旋转因子**：傅里叶变换中，`std::exp(Complex(0, 2*PI*k/N))` 生成单位根，效率高于手动 `cos` 和 `sin`。
- **模平方**：用 `std::norm(z)` 替代 `std::abs(z)^2`，避免 `sqrt` 开销，适合快速比较距离。
- **类型选择**：优先用 `std::complex<double>`，`float` 精度较低，`long double` 性能稍慢。
- **调试**：用 `operator<<` 输出复数，快速验证结果；`std::to_string(std::abs(z))` 可用于格式化模。

注意事项

- **浮点误差**：复数运算累积浮点误差，需用误差阈值比较（如 `1e-9`）；多步运算可能放大误差。
- **输入范围**：除法和 `log`，`pow` 要求分母或底非零；`sqrt`，`sin` 等支持任意复数。
- **性能**：`<complex>` 运算为 O(1)，但涉及多次浮点运算，整数场景可能用手动 `(x, y)` 更快。
- **竞赛限制**：确认题目是否允许浮点误差，部分题目要求精确解，可能需避免复数。
- **C++11 兼容**：`<complex>` 在 C++11 中功能完备，比赛编译器通常支持（如 g++ -std=c++11）。
- **头文件**：`<complex>` 自动包含 `<cmath>` 的相关函数（如 `sin`，`cos`），无需重复包含。

竞赛中的高频功能

以下是 `<complex>` 在算法竞赛中最常用的功能（基于题目类型）：

- **几何**：`std::complex`, `operator*`, `std::abs`, `std::arg`, `std::polar`.
- **傅里叶变换**：`std::exp`, `std::sin`, `std::cos`.
- **多项式**：`std::pow`, `std::sqrt`.
- **调试**：`operator<<`, `std::norm`, `std::conj`.

`<numeric>` 的核心功能与竞赛用途

在算法竞赛中，`<numeric>` 是 C++ 标准库中提供数值计算的头文件，包含一系列针对序列（如数组、向量）的算法。这些算法在处理数值序列、前缀和、差分、累积运算以及序列初始化等场景中非常实用。尽管 `<numeric>` 的使用频率不如 `<algorithm>` 或 `<vector>`，但其提供的功能简洁高效，特别适合快速实现数学相关的操作。相比手动实现，`<numeric>` 算法经过优化，代码简洁且性能可靠。

在 C++11 中，`<numeric>` 的功能稳定，新增了 `std::iota`，增强了序列初始化的便捷性。以下是 `<numeric>` 头文件中所有算法的详细说明，按功能分类，涵盖：

- **功能**：算法的数学定义和行为。
- **参数与返回值**：输入参数类型和返回值类型。
- **竞赛用途**：在算法竞赛中的典型应用场景。
- **复杂度**：算法的计算复杂度。
- **注意事项**：使用时的潜在问题或优化建议。

---

1. 累积运算（Accumulation Operations）

这些算法用于计算序列的累积结果，竞赛中常用于求和、求积或自定义操作。

- **accumulate(first, last, init)** - 求累积值
  - **功能**：计算 `[first, last)` 范围内元素的累积结果，从初始值 `init` 开始，应用加法或自定义二元操作。
    - 默认：`init + v[0] + v[1] + ... + v[n-1]`
    - 自定义：`op(op(...op(init, v[0]), v[1]), ..., v[n-1])`
  - **参数与返回值**：
    - 参数：`InputIt first, last`（输入迭代器），`T init`（初始值），可选 `BinaryOperation op`（二元操作）。
    - 返回：`T` 类型累积结果。
  - **竞赛用途**：
    - **求和**：计算数组或向量的总和（如总分、总距离）。
    - **求积**：结合 `std::multiplies` 计算乘积（如阶乘、概率）。
    - **自定义操作**：统计满足条件的元素、计算最大公约数（GCD）或最小公倍数（LCM）。
  - **复杂度**：O(n)，其中 n 为范围长度。
  - **注意事项**：
    - 确保 `init` 类型与操作结果兼容（如用 `long long` 避免溢出）。
    - 自定义操作需满足结合律。
    - 默认加法可能溢出，建议显式指定 `op` 或检查输入范围。
  - **示例**：

```cpp
vector<int> v{1, 2, 3, 4};
// 求和（默认加法）
long long sum = accumulate(v.begin(), v.end(), 0LL);  // 输出: 10

// 求积（自定义操作）
long long product = accumulate(v.begin(), v.end(), 1LL, multiplies<long long>());  // 输出: 24
```

- **inner_product(first1, last1, first2, init)** - 向量内积
  - **功能**：计算两个范围 `[first1, last1)` 和 `[first2, first2 + (last1 - first1))` 的内积，或应用自定义二元操作。
    - 默认：`init + (v1[0]*v2[0] + v1[1]*v2[1] + ... + v1[n-1]*v2[n-1])`
    - 自定义：`op1(init, op2(v1[0], v2[0]), op2(v1[1], v2[1]), ...)`
  - **参数与返回值**：
    - 参数：`InputIt1 first1, last1`（第一范围迭代器），`InputIt2 first2`（第二范围起点），`T init`（初始值），可选 `BinaryOperation1 op1`（累积操作）、`BinaryOperation2 op2`（元素操作）。
    - 返回：`T` 类型内积结果。
  - **竞赛用途**：
    - **向量内积**：计算点积（如几何中向量夹角、余弦定理）。
    - **加权和**：结合自定义操作计算加权累积（如多项式估值、评分系统）。
    - **矩阵运算**：简化矩阵点积或相关计算。
  - **复杂度**：O(n)，其中 n 为第一范围长度。
  - **注意事项**：
    - 两个范围长度必须匹配，否则行为未定义。
    - 默认乘法可能溢出，建议用 `long long` 或检查范围。
    - 自定义操作需保证结合律和类型兼容性。
  - **示例**：

```cpp
vector<int> v1{1, 2, 3}, v2{4, 5, 6};
// 内积: 1×4 + 2×5 + 3×6 = 32
long long dot = inner_product(v1.begin(), v1.end(), v2.begin(), 0LL);
cout << dot << endl;  // 输出: 32
```

---

2. 差分与前缀和（Difference and Partial Sum Operations）

这些算法用于计算序列的相邻差或前缀累积，竞赛中常用于差分数组或前缀和处理。

- `std::adjacent_difference`
  - **功能**：计算 `[first, last)` 范围内相邻元素的差（或自定义操作），结果存储到 `[d_first, d_first + (last - first))`。
    - 默认：`result[0] = v[0]`，`result[i] = v[i] - v[i-1]`（$i \geq 1$）。
    - 自定义：`result[0] = v[0]`，`result[i] = op(v[i], v[i-1])`.
  - **参数与返回值**：
    - 参数：`InputIt first, last`（输入迭代器），`OutputIt d_first`（输出迭代器），可选 `BinaryOperation op`（二元操作）。
    - 返回：`OutputIt` 指向输出范围末尾。
  - **竞赛用途**：

        - **差分数组**：构造差分序列，用于区间更新（如区间加减）。
        - **序列分析**：计算相邻元素变化（如速度、增量）。
        - **信号处理**：提取序列的离散导数或变化趋势。
    - **复杂度**：O(n)，其中 n 为范围长度。
    - **注意事项**：
        - 输出范围需至少与输入范围等大。
        - 默认减法可能导致负数，需确保类型支持。
        - 自定义操作需明确输入顺序（`op(v[i], v[i-1])`）。
    - **示例**：

```
vector<int> v = {1, 4, 6, 8};
vector<int> result(v.size());
std::adjacent_difference(v.begin(), v.end(), result.begin()); // {1, 3,
2, 2}
```

- `std::partial_sum`
    - **功能**：计算 `[first, last)` 范围内元素的前缀和（或自定义操作），结果存储到 `[d_first, d_first + (last - first))`。
        - 默认：`result[0] = v[0]`，`result[i] = v[0] + ... + v[i]`（$i \geq 1$）。
        - 自定义：`result[i] = op(...op(v[0], v[1]), ..., v[i])`.
    - **参数与返回值**：
        - 参数：`InputIt first, last`（输入迭代器），`OutputIt d_first`（输出迭代器），可选 `BinaryOperation op`（二元操作）。
        - 返回：`OutputIt` 指向输出范围末尾。
    - **竞赛用途**：
        - **前缀和**：快速计算区间和（如数组区间查询）。
        - **累积运算**：生成前缀积、最大值序列等。
        - **动态规划**：简化状态转移（如前缀和优化 DP）。
    - **复杂度**：O(n)，其中 n 为范围长度。
    - **注意事项**：
        - 输出范围需足够大。
        - 默认加法可能溢出，建议用 `long long`。
        - 自定义操作需满足结合律。
    - **示例**：

```
vector<int> v = {1, 2, 3, 4};
vector<int> result(v.size());
std::partial_sum(v.begin(), v.end(), result.begin()); // {1, 3, 6, 10}
```

---

3. 序列填充（Sequence Generation）

这些算法用于生成特定序列，竞赛中常用于初始化。

- `std::iota`
    - **功能**：填充 `[first, last)` 范围，从初始值 `value` 开始，依次递增（`value, value+1, value+2, ...`）。
    - **参数与返回值**：

        - 参数：`ForwardIt first, last`（前向迭代器），`T value`（初始值）。
        - 返回：无（`void`）。
    - **竞赛用途**：
        - **初始化序列**：生成连续整数序列（如索引、ID）。
        - **排列生成**：配合 `std::shuffle` 或 `std::next_permutation` 生成排列。
        - **图算法**：初始化节点编号或权重。
    - **复杂度**：O(n)，其中 n 为范围长度。
    - **注意事项**：
        - 确保 `T` 支持 `++` 操作。
        - 范围需可写，检查迭代器有效性。
        - 可能溢出，注意 `value` 和范围大小。
    - **示例**：

```
vector<int> v(5);
std::iota(v.begin(), v.end(), 1); // {1, 2, 3, 4, 5}
```

竞赛中的典型应用场景

1. **前缀和与区间查询**：
    - **问题**：快速计算数组区间 `[l, r]` 的和。
    - **方法**：用 `std::partial_sum` 预计算前缀和数组，区间和为 `prefix[r] - prefix[l-1]`。
    - **示例**：动态规划优化、区间统计（如求子数组和）。
2. **差分数组**：
    - **问题**：高效处理区间更新（如将 `[l, r]` 所有元素加 d）。
    - **方法**：用 `std::adjacent_difference` 构造差分数组，更新差分后用 `std::partial_sum` 恢复。
    - **示例**：区间修改、序列维护（如区间加减操作）。
3. **累积运算**：
    - **问题**：计算序列的和、积或自定义统计。
    - **方法**：用 `std::accumulate` 求和或积，结合 `std::multiplies` 或 lambda 实现复杂操作。
    - **示例**：统计总和、计算多项式值、GCD 序列。
4. **序列初始化**：
    - **问题**：快速生成连续整数序列。
    - **方法**：用 `std::iota` 初始化数组或向量，配合其他算法生成排列或测试数据。
    - **示例**：图节点编号、排列生成、测试用例构造。
5. **向量运算**：
    - **问题**：计算向量点积或加权和。
    - **方法**：用 `std::inner_product` 高效计算内积，简化几何或矩阵运算。
    - **示例**：向量夹角、加权评分、矩阵点积。

示例代码

以下是一个示例程序，展示 `<numeric>` 在算法 competitions 中的常见用法，涵盖前缀和、差分、累积运算、序列初始化和内积计算。代码模拟一个竞赛场景，处理数组的区间和查询、区间更新、统计乘积、生成序列和计算点积。

```cpp
#include <numeric>
#include <vector>
#include <iostream>
#include <functional>

int main() {
    // 1. 累积运算：求和与求积
    std::vector<int> v = {1, 2, 3, 4};
    long long sum = std::accumulate(v.begin(), v.end(), 0LL); // 10
    long long product = std::accumulate(v.begin(), v.end(), 1LL,
std::multiplies<long long>()); // 24
    std::cout << "Sum: " << sum << "\nProduct: " << product << '\n';

    // 2. 前缀和：计算区间和
    std::vector<long long> prefix(v.size());
    std::partial_sum(v.begin(), v.end(), prefix.begin()); // {1, 3, 6, 10}
    int l = 1, r = 3; // 区间 [1, 3]
    long long range_sum = prefix[r] - (l > 0 ? prefix[l - 1] : 0); // 6 + 4 = 10
    std::cout << "Range sum [" << l << ", " << r << "]: " << range_sum << '\n';

    // 3. 差分：区间更新
    std::vector<int> arr = {1, 2, 3, 4};
    std::vector<int> diff(arr.size());
    std::adjacent_difference(arr.begin(), arr.end(), diff.begin()); // {1, 1, 1,
1}
    // 区间 [1, 2] 所有元素加 5
    diff[1] += 5;
    diff[3] -= 5; // 差分更新
    std::vector<int> updated(arr.size());
    std::partial_sum(diff.begin(), diff.end(), updated.begin()); // {1, 7, 8, 4}
    std::cout << "After range update [1, 2] +5: ";
    for (int x : updated) std::cout << x << ' ';
    std::cout << '\n';

    // 4. 序列初始化
    std::vector<int> seq(5);
    std::iota(seq.begin(), seq.end(), 0); // {0, 1, 2, 3, 4}
    std::cout << "Sequence: ";
    for (int x : seq) std::cout << x << ' ';
    std::cout << '\n';

    // 5. 内积：向量点积
    std::vector<int> v1 = {1, 2, 3}, v2 = {4, 5, 6};
    long long dot = std::inner_product(v1.begin(), v1.end(), v2.begin(), 0LL);
// 32
    std::cout << "Dot product: " << dot << '\n';

    return 0;
}
```

竞赛使用技巧

- **前缀和优化**：用 `std::partial_sum` 预计算前缀和，O(n) 预处理后支持 O(1) 区间查询，适合区间和查询题目。
- **差分技巧**：结合 `std::adjacent_difference` 和 `std::partial_sum` 实现区间更新，O(n) 恢复数组，适合频繁修改的场景。
- **类型安全**：在 `std::accumulate` 和 `std::inner_product` 中使用 `long long` 作为初始值类型，防止整数溢出。
- **自定义操作**：利用 lambda 或 `<functional>`（如 `std::multiplies`, `std::plus`）扩展 `std::accumulate`、`std::inner_product` 的功能，适应复杂运算。
- **快速初始化**：用 `std::iota` 生成连续序列，配合 `std::vector` 动态分配，简化排列或索引生成。
- **调试**：输出中间结果（如前缀和、差分数组）验证正确性，使用 `std::cout` 检查算法行为。

注意事项

- **溢出风险**：`std::accumulate` 和 `std::partial_sum` 的加法/乘法可能溢出，建议使用 `long long` 或检查输入范围。
- **迭代器有效性**：确保输入/输出迭代器范围合法，输出范围需足够大，避免越界。
- **操作顺序**：`std::adjacent_difference` 的 `op(v[i], v[i-1])` 和 `std::inner_product` 的 `op2(v1[i], v2[i])` 需注意参数顺序，确保操作符合预期。
- **C++11 兼容**：`std::iota` 为 C++11 新增，比赛编译器通常支持（如 g++ -std=c++11）。
- **头文件**：`<numeric>` 包含所有上述算法，无需额外包含 `<functional>`（除非使用 `std::multiplies` 等）。
- **竞赛限制**：确认题目是否允许 STL，部分题目可能限制库函数使用，需手动实现类似功能。

竞赛中的高频功能

以下是 `<numeric>` 在算法竞赛中最常用的功能（基于题目类型）：

- **前缀和/区间查询**：`std::partial_sum`, `std::accumulate`.
- **差分/区间更新**：`std::adjacent_difference`, `std::partial_sum`.
- **累积运算**：`std::accumulate`, `std::inner_product`.
- **序列初始化**：`std::iota`.

`<valarray>` 的核心功能与竞赛用途

在算法竞赛中，`<valarray>` 是 C++ 标准库中提供高效数值数组操作的头文件，设计用于优化数学运算，特别适合处理大规模数值数据（如向量、矩阵）的元素级计算。相比标准容器如 `std::vector`，`<valarray>` 针对数值运算进行了底层优化，支持数组级的数学操作（如加法、乘法、三角函数等），并允许高效的切片和子数组操作。然而，在算法竞赛中，`<valarray>` 的使用频率较低，因为其功能较为专业化，且 `std::vector` 配合 `<algorithm>` 和 `<numeric>` 通常足以应对大多数场景。尽管如此，在涉及密集数值计算（如矩阵运算、信号处理、科学计算）或需要高性能数组操作的题目中，`<valarray>` 提供了简洁且高效的解决方案。

在 C++11 中，`<valarray>` 的功能保持稳定，未引入显著新特性，但其接口和优化使其在特定场景下仍具竞争力。以下是 `<valarray>` 头文件中核心功能和类的详细说明，按功能分类，涵盖：

- **功能**：类或方法的数学定义和行为。

- **参数与返回值**：输入参数类型和返回值类型。
- **竞赛用途**：在算法竞赛中的典型应用场景。
- **复杂度**：操作的计算复杂度（通常为 O(n) 或依赖底层实现）。
- **注意事项**：使用时的潜在问题或优化建议。

---

1. 核心类与构造（Core Class and Construction）

`<valarray>` 提供了一个核心模板类 `std::valarray<T>`，用于表示动态大小的数值数组，支持高效的数学运算。

- `std::valarray<T>`
  - **功能**：表示一个数值数组，元素类型为 `T`（通常为 `float`, `double`, `int` 等算术类型），支持数组级运算和切片操作。
  - **构造方式**：
    - 默认构造：`valarray<T>()`（空数组）。
    - 填充构造：`valarray<T>(value, size)`（`size` 个元素，初始化为 `value`）。
    - 大小构造：`valarray<T>(size)`（`size` 个元素，默认初始化）。
    - 数组构造：`valarray<T>(const T* arr, size)`（从指针 `arr` 复制 `size` 个元素）。
    - 初始化列表构造（C++11）：`valarray<T>(std::initializer_list<T>)`。
  - **参数与返回值**：
    - 参数：`T value`（填充值），`size_t size`（数组大小），`const T* arr`（源数组），`std::initializer_list<T>`（初始化列表）。
    - 返回：`std::valarray<T>` 对象。
  - **竞赛用途**：
    - **数值数组**：表示向量或矩阵，进行元素级运算。
    - **快速初始化**：快速构造大型数值数组，适合模拟或测试。
    - **信号处理**：存储信号数据，支持批量数学运算。
  - **复杂度**：O(n)（n 为数组大小，构造和复制时）。
  - **注意事项**：
    - `T` 需为算术类型（如 `int`, `double`），非算术类型可能不支持某些操作。
    - 内存动态分配，需注意性能开销。
    - 初始化列表构造在 C++11 中支持，比赛编译器需兼容。
  - **示例**：

```
std::valarray<double> v1(5.0, 4); // {5.0, 5.0, 5.0, 5.0}
std::valarray<double> v2 = {1.0, 2.0, 3.0}; // {1.0, 2.0, 3.0}
```

---

2. 基本操作（Basic Operations）

这些操作支持数组的访问、修改和基本管理，竞赛中用于数据处理和调试。

- `operator[]`
  - **功能**：访问或修改指定索引的元素，或对整个数组进行切片操作。
    - 单元素：`v[i]` 返回第 `i` 个元素（可读写）。
    - 切片：`v[slice]`、`v[gslice]` 等返回子数组（见切片操作）。
  - **参数与返回值**：

    - 参数：`size_t i`（索引），`std::slice` 或 `std::gslice`（切片对象）。
    - 返回：`T&`（单元素），`std::valarray<T>`（切片结果）。
  - **竞赛用途**：
    - **元素访问**：读取或修改数组元素，如更新向量分量。
    - **子数组**：通过切片提取部分数据，简化矩阵操作。
  - **复杂度**：O(1)（单元素访问），O(n)（切片操作，n 为子数组大小）。
  - **注意事项**：
    - 索引需在 `[0, size())` 范围内，否则未定义行为。
    - 切片操作需配合 `std::slice` 或 `std::gslice`。
  - **示例**：

```
std::valarray<int> v = {1, 2, 3, 4};
v[1] = 5; // v = {1, 5, 3, 4}
```

- `size()`
  - **功能**：返回数组的元素个数。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`size_t`（数组大小）。
  - **竞赛用途**：检查数组长度，循环遍历或分配输出空间。
  - **复杂度**：O(1)。
  - **注意事项**：空数组返回 0，需检查大小。
  - **示例**：

```
std::valarray<int> v(3);
std::cout << v.size(); // 3
```

- `resize(size_t n, T c = T())`
  - **功能**：调整数组大小为 `n`，新元素初始化为 `c`（默认 `T()`）。
  - **参数与返回值**：
    - 参数：`size_t n`（新大小），`T c`（填充值）。
    - 返回：无（`void`）。
  - **竞赛用途**：动态调整数组大小，适应输入数据。
  - **复杂度**：O(n)（n 为新大小）。
  - **注意事项**：
    - 原数据丢失，需备份重要数据。
    - 内存重新分配，性能开销较大。
  - **示例**：

```
std::valarray<int> v = {1, 2};
v.resize(4, 0); // v = {0, 0, 0, 0}
```

---

3. 数组级运算（Array-Wide Arithmetic Operations）

`<valarray>` 支持数组级的算术运算，自动对每个元素应用操作，竞赛中用于高效的向量或矩阵计算。

- `operator+`, `operator-`, `operator*`, `operator/`, `operator%`
  - **功能**：对两个 `valarray` 或 `valarray` 与标量执行元素级运算。
    - 数组与数组：`result[i] = v1[i] op v2[i]`（如 `v1 + v2`）。
    - 数组与标量：`result[i] = v[i] op c` 或 `result[i] = c op v[i]`。
  - **参数与返回值**：
    - 参数：`const std::valarray<T>&`（数组），`T`（标量）。
    - 返回：`std::valarray<T>`（新数组，存储结果）。
  - **竞赛用途**：
    - **向量运算**：批量加减乘除，如向量加法、缩放。
    - **矩阵运算**：元素级操作，简化矩阵计算。
    - **信号处理**：对信号数组进行批量变换。
  - **复杂度**：O(n)（n 为数组大小）。
  - **注意事项**：
    - 数组与数组操作需大小相同。
    - 除法和模运算需确保除数非零。
    - 返回新数组，需注意内存开销。
  - **示例**：

```
std::valarray<double> v1 = {1.0, 2.0, 3.0};
std::valarray<double> v2 = {4.0, 5.0, 6.0};
auto sum = v1 + v2; // {5.0, 7.0, 9.0}
auto scaled = v1 * 2.0; // {2.0, 4.0, 6.0}
```

- `operator+=`, `operator-=`, `operator*=`, `operator/=`, `operator%=`
  - **功能**：原地执行元素级运算，修改当前数组。
  - **参数与返回值**：
    - 参数：`const std::valarray<T>&`（数组），`T`（标量）。
    - 返回：`std::valarray<T>&`（当前数组引用）。
  - **竞赛用途**：高效更新数组，如批量缩放或平移。
  - **复杂度**：O(n)（n 为数组大小）。
  - **注意事项**：同基本运算，注意除数非零和数组大小匹配。
  - **示例**：

```
std::valarray<double> v = {1.0, 2.0, 3.0};
v += 2.0; // {3.0, 4.0, 5.0}
```

```
```

- **一元运算**：`operator+`, `operator-`, `operator~`, `operator!`
  - **功能**：对数组元素应用一元运算（正、负、位反、逻辑非）。
  - **参数与返回值**：
    - 参数：无（应用于自身）。

    - 返回：`std::valarray<T>`（新数组）。
  - **竞赛用途**：快速变换数组，如取负或位操作。
  - **复杂度**：O(n)。
  - **注意事项**：`~` 和 `!` 仅对整数或布尔类型有意义。
  - **示例**：

```
std::valarray<int> v = {1, -2, 3};
auto neg = -v; // {-1, 2, -3}
```

---

4. 数学函数（Mathematical Functions）

`<valarray>` 提供了一系列数学函数，自动应用于数组的每个元素，竞赛中用于批量数值计算。

- `abs`, `acos`, `asin`, `atan`, `atan2`, `cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`, `exp`, `log`, `log10`, `pow`, `sqrt`
  - **功能**：对数组每个元素应用对应的数学函数（如绝对值、三角函数、指数、对数、幂、平方根）。
    - 示例：`cos(v)[i] = cos(v[i])`。
    - `atan2(v1, v2)[i] = atan2(v1[i], v2[i])`（双参数）。
    - `pow(v, c)[i] = pow(v[i], c)`（标量指数）。
  - **参数与返回值**：
    - 参数：`const std::valarray<T>&`（输入数组），`T` 或 `const std::valarray<T>&`（标量或数组参数）。
    - 返回：`std::valarray<T>`（结果数组）。
  - **竞赛用途**：
    - **信号处理**：应用三角函数或指数函数处理信号。
    - **几何计算**：计算角度、距离或变换。
    - **科学计算**：批量计算复杂函数值。
  - **复杂度**：O(n)（n 为数组大小，依赖底层 `<cmath>`）。
  - **注意事项**：
    - 输入需在函数定义域内（如 `log` 要求正数）。
    - 浮点运算可能引入误差，需控制精度。
    - 部分函数（如 `atan2`, `pow`）支持数组与数组操作。
  - **示例**：

```
std::valarray<double> v = {0.0, 1.57, 3.14};
auto cos_v = std::cos(v); // {cos(0), cos(1.57), cos(3.14)}
```

---

5. 切片与子数组（Slicing and Subarray Operations）

`<valarray>` 支持通过 `std::slice` 和 `std::gslice` 进行高效切片操作，竞赛中用于矩阵或数组子集处理。

- `std::slice` **和** `operator[slice]`
  - **功能**：从数组中提取子数组，指定起始位置、长度和步长。

    - `slice(start, length, stride)`：从索引 `start` 开始，提取 `length` 个元素，步长为 `stride`。
  - **参数与返回值**：
    - 参数：`size_t start`（起始索引），`size_t length`（元素个数），`size_t stride`（步长）。
    - 返回：`std::valarray<T>`（子数组）。
  - **竞赛用途**：
    - **矩阵操作**：提取矩阵的行、列或子块。
    - **信号处理**：处理周期性数据或采样。
  - **复杂度**：O(n)（n 为子数组大小）。
  - **注意事项**：
    - 索引需合法，超出范围未定义。
    - 切片返回新数组，修改切片不影响原数组。
  - **示例**：

```
std::valarray<int> v = {1, 2, 3, 4, 5};
std::slice s(1, 3, 2); // 索引 1, 3（步长 2）
auto sub = v[s]; // {2, 4}
```

- `std::gslice`
  - **功能**：广义切片，支持多维索引提取（如矩阵子块）。
  - **参数与返回值**：
    - 参数：`size_t start`（起始索引），`const std::valarray<size_t>& length`（各维度长度），`const std::valarray<size_t>& stride`（各维度步长）。
    - 返回：`std::valarray<T>`（子数组）。
  - **竞赛用途**：处理多维数组或复杂矩阵操作。
  - **复杂度**：O(n)（n 为子数组大小）。
  - **注意事项**：实现复杂，竞赛中较少使用。
  - **示例**：

```
std::valarray<int> v = {1, 2, 3, 4, 5, 6};
std::valarray<size_t> len = {2}, str = {2};
std::gslice gs(0, len, str);
auto sub = v[gs]; // 复杂索引提取
```

---

6. 统计与聚合（Statistical and Aggregation Operations）

这些方法用于计算数组的统计信息，竞赛中用于快速分析数据。

- `sum()`
  - **功能**：计算数组所有元素的和。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`T`（和）。
  - **竞赛用途**：快速计算向量或矩阵的元素总和。
  - **复杂度**：O(n)。

- **注意事项**：可能溢出，建议检查类型。
- **示例**：

```
std::valarray<int> v = {1, 2, 3};
int s = v.sum(); // 6
```

```

```

- `min()`, `max()`
  - **功能**：返回数组的最小值或最大值。
  - **参数与返回值**：
    - 参数：无。
    - 返回：`T`（最小/最大值）。
  - **竞赛用途**：统计数组范围，检查边界条件。
  - **复杂度**：O(n)。
  - **注意事项**：空数组行为未定义。
  - **示例**：

```
std::valarray<int> v = {1, 2, 3};
int m = v.min(); // 1
```

竞赛中的典型应用场景

1. **向量与矩阵运算**：
   - **问题**：对向量或矩阵进行元素级加减乘除。
   - **方法**：用 `operator+`, `operator*` 等实现批量运算，简化代码。
   - **示例**：向量加法、矩阵缩放。
2. **信号处理**：
   - **问题**：对信号数组应用数学函数（如 `sin`, `exp`）。
   - **方法**：用 `std::sin`, `std::exp` 等批量处理数组。
   - **示例**：傅里叶变换预处理、信号滤波。
3. **矩阵切片**：
   - **问题**：提取矩阵的行、列或子块。
   - **方法**：用 `std::slice` 或 `std::gslice` 提取子数组。
   - **示例**：矩阵分解、子矩阵运算。
4. **统计分析**：
   - **问题**：快速计算数组的和、最值。
   - **方法**：用 `sum()`, `min()`, `max()` 提取统计信息。
   - **示例**：数据范围检查、总和计算。
5. **数值模拟**：
   - **问题**：模拟大规模数值数据。
   - **方法**：用 `valarray` 构造和操作大型数组。
   - **示例**：物理模拟、测试用例生成。

示例代码

以下是一个示例程序，展示 `<valarray>` 在算法竞赛中的常见用法，涵盖数组构造、算术运算、数学函数、切片和统计操作。代码模拟一个竞赛场景，处理向量的加法、缩放、数学变换、子数组提取和统计分析。

```
#include <valarray>
#include <iostream>
#include <iomanip>

int main() {
    // 1. 构造与初始化
    std::valarray<double> v1 = {1.0, 2.0, 3.0, 4.0}; // 初始化列表
    std::valarray<double> v2(2.0, 4); // 填充构造 {2.0, 2.0, 2.0, 2.0}
    std::cout << "v1: ";
    for (auto x : v1) std::cout << x << ' ';
    std::cout << '\n';

    // 2. 数组级运算
    auto sum = v1 + v2; // {3.0, 4.0, 5.0, 6.0}
    auto scaled = v1 * 2.0; // {2.0, 4.0, 6.0, 8.0}
    std::cout << "v1 + v2: ";
    for (auto x : sum) std::cout << x << ' ';
    std::cout << "\nv1 * 2: ";
    for (auto x : scaled) std::cout << x << ' ';
    std::cout << '\n';

    // 3. 数学函数
    auto cos_v1 = std::cos(v1); // {cos(1), cos(2), cos(3), cos(4)}
    std::cout << "cos(v1): ";
    for (auto x : cos_v1) std::cout << std::fixed << std::setprecision(4) << x
<< ' ';
    std::cout << '\n';

    // 4. 切片操作
    std::valarray<int> v3 = {1, 2, 3, 4, 5, 6};
    std::slice s(1, 3, 2); // 索引 1, 3, 5
    auto sub = v3[s]; // {2, 4, 6}
    std::cout << "Slice of v3: ";
    for (auto x : sub) std::cout << x << ' ';
    std::cout << '\n';

    // 5. 统计操作
    std::cout << "Sum of v1: " << v1.sum() << '\n';
    std::cout << "Min of v1: " << v1.min() << '\n';
    std::cout << "Max of v1: " << v1.max() << '\n';

    return 0;
}
```

竞赛使用技巧

- **高效运算**：用数组级运算（如 `v1 + v2`）替代循环，简化代码并优化性能。
- **数学函数**：批量应用 `std::sin`, `std::exp` 等，适合信号处理或几何计算。
- **切片优化**：用 `std::slice` 提取子数组，减少手动循环开销。
- **类型选择**：优先用 `double` 或 `float` 作为 `T`，整数类型可能不支持所有数学函数。
- **调试**：遍历 `valarray` 输出元素，验证运算结果；用 `sum()`, `min()` 检查数据。
- **替代方案**：若 `<valarray>` 不熟悉，可用 `std::vector` 配合 `<algorithm>` 和 `<numeric>` 实现类似功能。

注意事项

- **使用场景**：`<valarray>` 适合密集数值计算，普通数组操作用 `std::vector` 更通用。
- **性能**：`<valarray>` 底层优化依赖编译器，实际性能因实现而异。
- **溢出与误差**：算术和数学函数可能溢出或引入浮点误差，需检查输入范围和精度。
- **切片复杂性**：`std::slice` 和 `std::gslice` 使用复杂，需熟悉接口。
- **C++11 兼容**：`<valarray>` 在 C++11 中支持初始化列表，比赛编译器通常支持（如 g++ -std=c++11）。
- **竞赛限制**：确认题目是否允许 `<valarray>`，部分比赛可能限制非核心 STL 使用。
- **头文件**：`<valarray>` 自动包含 `<cmath>` 的数学函数，无需额外包含。

竞赛中的高频功能

以下是 `<valarray>` 在算法竞赛中最常用的功能（基于题目类型）：

- **向量/矩阵运算**：`operator+`, `operator*`, `operator+=`.
- **数学变换**：`std::cos`, `std::exp`, `std::pow`.
- **切片操作**：`operator[slice]`.
- **统计分析**：`sum()`, `min()`, `max()`.

输入输出（Input/Output）

`<iomanip>` 的核心功能与竞赛用途

`fixed`, `setprecision`

- **功能**：控制浮点数输出格式和精度。
  - `fixed`：强制以定点表示法（小数形式）输出浮点数。
- `setprecision(n)`：设置输出浮点数的精度为 n 位（小数点后有效数字）。
- **参数与返回值**：
  - 参数：
    - `fixed`：无参数。
    - `setprecision(n)`：整数 n，表示精度。
  - 返回：流操纵符，应用于 `cout` 等输出流。
- **复杂度**：O(1)（仅影响输出格式，不涉及计算）。

- **注意事项**：
  - `fixed` 与 `setprecision` 通常结合使用。
  - 精度设置对后续所有浮点输出生效，直到被重置。
  - 过高的精度可能导致输出冗长，注意题目要求的位数。
  - 不影响计算精度，仅控制输出。
  - 重置格式：若需取消 fixed 或更改精度，可使用 `cout.unsetf(ios::fixed)` 或再次调用 `setprecision`。
- **示例**：

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

int main() {
    double x = 123.456789;
    cout << fixed << setprecision(2) << x << endl; // 123.46
    cout << setprecision(4) << x << endl; // 123.4568
    return 0;
}
```

`<fstream>`的核心功能与竞赛用途

```
#include <iostream>       // 标准输入输出头文件
#include <string>         // 字符串类支持
#include <fstream>        // 文件输入输出头文件
using namespace std;      // 使用标准命名空间，简化代码书写

int main()
{
    ifstream input("string.in");      // 创建输入文件流，打开文件 string.in
    ofstream output("string.out");    // 创建输出文件流，准备写入文件 string.out

    if (!input || !output) {          // 检查文件是否成功打开
        cerr << "文件打开失败。" << endl;  // 输出错误信息
        return 1;                     // 返回非零值，表示程序异常退出
    }

    char c;               // 用于读取一个字符（预期是数字字符）
    string a, b;          // 定义两个字符串，用于存储输入的两行内容

    input >> c;           // 从文件读取一个字符（如 '2'）
    input.get();          // 读取并丢弃下一个字符（通常是换行符）
    getline(input, a);    // 读取第二行字符串到 a
    getline(input, b);    // 读取第三行字符串到 b
    input.close();        // 关闭输入文件

    int p = c - '0';      // 将字符数字转换为整数，比如 '3' → 3
    a.erase(p);           // 从位置 p 开始删除字符串 a 的后续部分
    a += b;               // 将字符串 b 拼接到 a 的后面

    // 在主函数中直接反转字符串 a
```

```
    for (int i = 0, j = a.size() - 1; i < j; ++i, --j) {
        swap(a[i], a[j]);  // 交换第 i 和 j 个字符，实现就地反转
    }

    output << a << endl;  // 将最终结果写入输出文件，并换行
    output.close();       // 关闭输出文件

    cout << "DONE" << endl; // 向控制台输出提示信息，表示程序结束
    return 0;               // 程序正常结束，返回 0
}
```

`ifstream`

- **功能**：输入文件流，用于从文件中读取数据。继承自 `istream`，支持格式化输入（`>>`）和非格式化输入（`read`、`getline`），适用于文本或二进制文件。
- **参数与返回值**：
  - **构造函数**：
    - `ifstream(filename, mode = ios::in)`：
      - `filename`：文件路径（字符串或 C 风格字符串）。
      - `mode`：打开模式（默认为 `ios::in`，可结合 `ios::binary` 用于二进制模式）。
  - **主要方法**：
    - `open(filename, mode)`：打开文件。
    - `is_open()`：返回 `bool`，检查文件是否成功打开。
    - `close()`：关闭文件。
    - `eof()`：返回 `bool`，检查是否到达文件末尾。
    - `read(char* buffer, streamsize n)`：读取 n 个字节到缓冲区（二进制）。
    - `getline`：读取一行到字符串。
    - `>>`：格式化输入，支持内置类型。
  - **返回值**：如 `is_open()` 返回 `bool`；`>>` 返回流对象以支持链式操作。
- **示例**：

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

int main() {
    ifstream in("input.txt");
    if (in.is_open()) {
        int x;
        in >> x; // 读取整数
        cout << "读取: " << x << endl;
        in.close();
    } else {
        cerr << "无法打开输入文件" << endl;
    }
    return 0;
}
```

`ofstream`

- **功能**：输出文件流，用于向文件中写入数据。继承自 `ostream`，支持格式化输出（`<<`）和非格式化输出（`write`），适用于文本或二进制文件。
- **参数与返回值**：
  - **构造函数**：
    - `ofstream(filename, mode = ios::out)`：
      - `filename`：文件路径（字符串或 C 风格字符串）。
      - `mode`：打开模式（默认为 `ios::out`，可结合 `ios::binary` 或 `ios::app` 用于追加模式）。
  - **主要方法**：
    - `open(filename, mode)`：打开文件。
    - `is_open()`：返回 `bool`，检查文件是否成功打开。
    - `close()`：关闭文件。
    - `write(const char* buffer, streamsize n)`：写入 n 个字节（二进制）。
    - `<<`：格式化输出，支持内置类型。
  - **返回值**：如 `is_open()` 返回 `bool`；`<<` 返回流对象以支持链式操作。
- **示例**：

```
#include <fstream>
#include <iomanip>
using namespace std;

int main() {
    ofstream out("output.txt");
    if (out.is_open()) {
        out << fixed << setprecision(2) << 123.456 << endl;
        out.close();
    } else {
        cerr << "无法打开输出文件" << endl;
    }
    return 0;
}
```

`fstream`

- **功能**：双向文件流，支持文件的读写操作。继承自 `iostream`（结合 `istream` 和 `ostream`），适用于需要同时读写同一文件的场景，支持文本或二进制模式。
- **参数与返回值**：
  - **构造函数**：
    - `fstream(filename, mode = ios::in | ios::out)`：
      - `filename`：文件路径（字符串或 C 风格字符串）。
      - `mode`：打开模式（默认 `ios::in | ios::out`，可结合 `ios::binary`、`ios::app` 等）。
  - **主要方法**：
    - `open(filename, mode)`：打开文件。
    - `is_open()`：返回 `bool`，检查文件是否成功打开。

        - `close()`：关闭文件。
        - `eof()`：返回 `bool`，检查是否到达文件末尾。
        - `read(char* buffer, streamsize n)`：读取 n 个字节。
        - `write(const char* buffer, streamsize n)`：写入 n 个字节。
        - `seekg` / `seekp`：设置读/写指针位置。
        - `>>` / `<<`：格式化读写。
    - **返回值**：如 `is_open()` 返回 `bool`；`>>` / `<<` 返回流对象以支持链式操作。
    - **示例**：

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

int main() {
    fstream file("data.txt", ios::in | ios::out | ios::trunc);
    if (file.is_open()) {
        // 写入
        file << 123.456 << endl;
        file.seekg(0); // 移动读指针到开头
        // 读取
        double x;
        file >> x;
        cout << "读取: " << x << endl;
        file.close();
    } else {
        cerr << "无法打开文件" << endl;
    }
    return 0;
}
```

`<istream>`的核心功能与竞赛用途

- `getline`
    - **功能**：
        - 从输入流（如 `cin` 或 `ifstream`）读取一行文本，直到遇到换行符（`\n`）或流结束。
- 行为：将读取的字符存储到 `string` 中，丢弃换行符，支持指定自定义分隔符。
    - **参数与返回值**：
        - `getline(istream& is, string& str)`：
            - 参数：
                - `istream& is`：输入流（如 `cin`）。
                - `string& str`：存储读取的字符串。
            - 返回值：`istream&`（输入流引用，支持链式操作）。
        - `getline(istream& is, string& str, char delim)`：
            - 参数：
                - `istream& is`：输入流。
                - `string& str`：存储读取的字符串。
            - `char delim`：自定义分隔符（默认 `\n`）。

    - 返回值：`istream&`。
  - 状态：读取失败（流结束或错误）时，设置流的 `fail` 标志。
  - **竞赛用途**：
    - **复杂输入**：读取包含空格的字符串（如句子、表达式）。
  - **多行输入**：处理每行格式不同的输入数据。
    - **字符串处理**：结合 `<sstream>` 解析行内数据（如空格分隔的数字）。
  - **文件输入**：从 `<fstream>` 的 `ifstream` 读取文件行。
  - **复杂度**：
  - O(n)，n 为读取的字符数（线性扫描直到分隔符或流结束）。
  - **注意事项**：
    - **换行符处理**：读取 `cin >> x` 后可能残留换行符，需用 `cin.ignore()` 清除。
  - **空行**：`getline` 可读取空行，需检查 `str.empty()`。
    - **性能**：适合小规模输入，大规模字符串输入可能较慢，考虑字符数组或自定义解析。
  - **依赖性**：需要 `<string>` 支持 `string`，通常与 `<iostream>` 或 `<fstream>` 配合。
    - **错误检查**：检查流状态（如 `if (getline(cin, str))`）以处理输入结束或错误。
  - **示例**：

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

int main() {
    string line;
    getline(cin, line); // 读取整行（如 "hello world"）
    cout << "输入: " << line << '\n';
    return 0;
}
```

`<sstream>` 的核心功能与竞赛用途

`stringstream`

- **功能**：双向字符串流，支持将字符串作为输入/输出流进行读写操作。继承自 `iostream`（结合 `istream` 和 `ostream`），可进行格式化输入（`>>`）和输出（`<<`），用于字符串的解析和构建。
- **参数与返回值**：
  - **构造函数**：
    - `stringstream()`：创建空字符串流。
    - `stringstream(string s)`：以字符串 `s` 初始化流。
    - `stringstream(mode)`：指定模式（如 `ios::in | ios::out`，默认支持读写）。
  - **主要方法**：
    - `str()`：获取流的底层字符串（返回 `string`）。
    - `str(string s)`：设置流的底层字符串。
    - `clear()`：清除流状态（不清除内容）。
    - `>>`：从流中读取数据（格式化输入）。
    - `<<`：向流中写入数据（格式化输出）。

- **返回值**：如 `str()` 返回 `string`；`>>`/`<<` 返回流对象以支持链式操作。
- **竞赛用途**：
  - **字符串解析**：将字符串拆分为数字、单词等（如解析空格分隔的输入）。
  - **格式化输出**：将多种类型数据（数字、字符串）拼接为特定格式的字符串。
  - **类型转换**：在字符串和数字（如 `int`、`double`）之间转换。
  - **临时缓冲**：作为中间容器，处理复杂输入输出转换，替代直接操作 `string`。
  - **调试**：快速构建或解析测试数据。
- **复杂度**：
  - 构造/设置字符串：O(n)，n 为字符串长度。
  - 读写操作：O(n)，取决于数据量和格式化复杂度。
- **注意事项**：
  - 使用 `clear()` 清除状态（如 `eof` 或 `fail`），但内容需用 `str("")` 清空。
  - **在进行多次转换的时候，必须调用stringstream的成员函数.clear()**
  - 流状态需检查（如 `fail()`），尤其在解析失败时。
  - 性能略低于直接字符串操作，适合中小规模数据处理。
  - **由于stringstream构造函数会特别消耗内存，似乎不打算主动释放内存(或许是为了提高效率)，但如果你要在程序中用同一个流，反复读写大量的数据，将会造成大量的内存消耗，因些这时候，需要适时地清除一下缓冲 (用 stream.str(""))。**

    另外不要企图用 stream.str().resize(0)，或 stream.str().clear() 来清除缓冲，使用它们似乎可以让stringstream的内存消耗不要增长得那么快，但仍然不能达到清除stringstream缓冲的效果，内存的消耗还在缓慢的增长！，至于stream.flush()，则根本就起不到任何作用。
- 竞赛中常用于处理动态输入格式或生成格式化输出。
- **示例**：

```cpp
#include <sstream>
#include <iostream>
using namespace std;

int main() {
    // 解析字符串
    stringstream ss("123 45.67 hello");
    int a;
    double b;
    string c;
    ss >> a >> b >> c;
    cout << "解析: " << a << ", " << b << ", " << c << endl;

    // 构建字符串
    stringstream out;
    out << a << " + " << b << " = " << (a + b);
    cout << "构建: " << out.str() << endl;

    stringstream ss;
    int first = 0, second = 0;

    ss << "456";  // 插入字符串
    ss >> first;  //转换成int
    cout << first << endl;
```

```
    ss.clear();  //在进行多次转换前， 必须清除ss
    ss << true;
    ss >> second;
    cout << second << endl;
    return 0;
}
```

在类型转换中使用模板

你可以轻松地定义函数模板来将一个任意的类型转换到特定的目标类型。例如，需要将各种数字值，如int、long、double等等转换成字符串，要使用以一个string类型和一个任意值*t*为参数的to_string()函数。to_string()函数将*t*转换为字符串并写入result中。使用str()成员函数来获取流内部缓冲的一份拷贝。

```
template<class T>

void to_string(string &result, const T &t)
{

    ostringstream oss;  //创建一个流
    oss << t;  //把值传递入流中
    result = oss.str();  //获取转换后的字符并将其写入result
}

//这样，你就可以轻松地将多种数值转换成字符串了
to_string(s1, 10.5);  //double到string
to_string(s2, 123);  //int到string
to_string(s3, true);  //bool到string

//可以更进一步定义一个通用的转换模板，用于任意类型之间的转换。函数模板convert()含有两个模板参数out_type和in_value，功能是将in_value值转换成out_type类型：

template<class out_type, class in_value>
out_type convert(const in_value & t)
{
    stringstream stream;

    stream << t;  //向流中传值
    out_type result;  //这里存储转换结果
    stream >> result;  //向result中写入值

    return result;
}
```

`istringstream`

- **功能**：输入字符串流，仅支持从字符串读取数据。继承自 `istream`，用于格式化输入（`>>`）或非格式化输入（如 `getline`），适合解析已有字符串内容。
- **参数与返回值**：
  - **构造函数**：
    - `istringstream()`：创建空输入流。

        - `istringstream(string s)`：以字符串 `s` 初始化流。
        - `istringstream(mode)`：指定模式（通常为 `ios::in`）。
    - **主要方法**：
        - `str()`：获取流的底层字符串（返回 `string`）。
        - `str(string s)`：设置流的底层字符串。
        - `clear()`：清除流状态。
        - `>>`：格式化读取数据。
        - `getline`：读取一行（需结合 `<string>`）。
    - **返回值**：如 `str()` 返回 `string`；`>>` 返回流对象以支持链式操作。

`ostringstream`

- **功能**：输出字符串流，仅支持向字符串写入数据。继承自 `ostream`，用于格式化输出（`<<`），适合构建字符串内容。
- **参数与返回值**：
    - **构造函数**：
        - `ostringstream()`：创建空输出流。
        - `ostringstream(string s)`：以字符串 `s` 初始化流（通常用于追加）。
        - `ostringstream(mode)`：指定模式（通常为 `ios::out`）。
    - **主要方法**：
        - `str()`：获取流的底层字符串（返回 `string`）。
        - `str(string s)`：设置流的底层字符串。
        - `clear()`：清除流状态。
        - `<<`：格式化写入数据。
    - **返回值**：如 `str()` 返回 `string`；`<<` 返回流对象以支持链式操作。

优化 I/O 性能：

`std::ios::sync_with_stdio(false)` 和 `std::cin.tie(nullptr)`

- **功能**：
    - `std::ios::sync_with_stdio(false)`：禁用 C++ 流（`std::cin`, `std::cout`）与 C 风格 I/O（`stdio.h` 的 `scanf`, `printf`）的同步，减少流操作的开销。
    - `std::cin.tie(nullptr)`：解除 `std::cin` 和 `std::cout` 的绑定，确保输入前不自动刷新输出缓冲区，提升性能。
    - 行为：优化标准输入输出的性能，减少不必要的同步和缓冲区刷新，尤其适用于大规模 I/O。
- **参数与返回值**：
    - `std::ios::sync_with_stdio(bool sync)`：
        - 参数：`bool sync`（true 启用同步，false 禁用）。
        - 返回值：`bool`（之前的同步状态）。
    - `std::cin.tie(std::ostream* tie)`：
        - 参数：`std::ostream* tie`（绑定的输出流，`nullptr` 表示解除绑定）。
        - 返回值：`std::ostream*`（之前的绑定流）。
    - 使用方式：在程序开始时调用，通常为 `std::ios::sync_with_stdio(false); std::cin.tie(nullptr);`。

- **竞赛用途**：
  - **大规模输入输出**：加速处理大数据量输入（如 10^6 个整数）或输出（如长数组）。
  - **时间限制严格**：避免 I/O 成为瓶颈，确保程序在时间限制内完成。
  - **标准 I/O 优化**：使 `std::cin`/`std::cout` 的性能接近 C 风格 I/O（如 `scanf`/`printf`）。
- **复杂度**：
  - 设置操作：O(1)（仅修改流状态）。
  - 对 I/O 影响：显著降低每次 I/O 操作的常数开销（实际提升因实现和数据量而异）。
- **注意事项**：
  - **调用时机**：必须在任何 I/O 操作（如 `std::cin >> x` 或 `std::cout << x`）之前调用，否则行为未定义。
  - **C 风格 I/O 冲突**：禁用同步后，不能混用 `scanf`/`printf` 和 `std::cin`/`std::cout`，否则可能导致输出顺序错误。
  - **输出刷新**：解除 `tie` 后，需手动调用 `std::cout << std::flush` 或 `std::endl` 确保输出刷新（`std::endl` 包含换行和刷新）。
  - **适用范围**：仅对标准 I/O（`std::cin`, `std::cout`）有效，对文件流或字符串流无影响。
  - **调试**：优化可能影响调试输出的实时性，建议调试时暂时禁用。

其他

`<limits>`的实用功能

概念：
基本格式：std::numeric_limits::max()——其中max()可替换为min()及lowest()；T表示数据类型；

std::numeric_limits包含了以下的常用函数：

- **max()** : 返回指定类型最大有限值
- **min()**：返回指定类型的最小的有限值
- **lowest()**：返回指定类型的最低有限值
- **epsilon()**: 返回的是表示1和大于1的最小浮点数之间的差的值。这对于浮点数的精度控制非常有用。

```
#include <iostream>
#include <limits>
int main()
{
    std::cout << "-----int-----" << std::endl;
    std::cout << "max = " << std::numeric_limits<int>::max() << std::endl;
    std::cout << "min = " << std::numeric_limits<int>::min() << std::endl;
    std::cout << "lowest = " << std::numeric_limits<int>::lowest() << std::endl;
    std::cout << "epsilon = " << std::numeric_limits<int>::epsilon() <<
std::endl;
    return 0;
}
```

```
-----short-----
max = 32767
min = -32768
lowest = -32768
```

```
epsilon = 0
-----int-----
max = 2147483647
min = -2147483648
lowest = -2147483648
epsilon = 0
-----unsigned int-----
max = 4294967295
min = 0
lowest = 0
epsilon = 0
-----float-----
max = 3.40282e+38
min = 1.17549e-38
lowest = -3.40282e+38
epsilon = 1.19209e-07
-----double-----
max = 1.79769e+308
min = 2.22507e-308
lowest = -1.79769e+308
epsilon = 2.22045e-16
-----long-----
max = 2147483647
min = -2147483648
lowest = -2147483648
epsilon = 0
-----long long-----
max = 9223372036854775807
min = -9223372036854775808
lowest = -9223372036854775808
epsilon = 0
```

`<locale>`的实用功能

- `toupper(c, loc)` 将字符 c 转换为大写，基于`loc`指定的本地化规则。
- `tolower(c, loc)` 将字符 c 转换为小写。
- 如果不指定 `loc`，可以使用默认的全局本地化环境（`locale()`）。

`<utility>`的实用功能

pair：存储两个值的容器

基本用法

```
#include <utility>
#include <iostream>
int main() {
    // 创建 pair
    std::pair<int, std::string> student(1, "Alice");

    // 访问元素
    std::cout << "ID: " << student.first << ", Name: " << student.second <<
"\n";
```

```
    // 结构化绑定（C++17）
    auto [id, name] = student;
    std::cout << "ID: " << id << ", Name: " << name << "\n";

    return 0;
}
```

==========
输出：
ID: 1, Name: Alice
ID: 1, Name: Alice
==========
用途：存储键值对（如 map 的元素）、函数多返回值。

工厂函数 std::make_pair

```
auto point = std::make_pair(3.14, 2.71);  // 自动推导类型
```

move：转换为右值引用（启用移动语义）

移动资源（避免拷贝）

```
#include <utility>
#include <vector>

int main() {
    std::vector<int> src = {1, 2, 3};
    std::vector<int> dest = std::move(src);  // 移动构造

    std::cout << "src size: " << src.size() << "\n";  // 0（资源已转移）
    std::cout << "dest size: " << dest.size() << "\n"; // 3
    return 0;
}
```

关键点：
move 将对象标记为“可移动”，调用移动构造函数/赋值运算符。
被移动后的对象处于有效但未定义状态（如 src 为空）。

forward：完美转发（保持值类别）

保持左值/右值属性

```
#include <utility>
#include <iostream>

// 定义一个函数模板，接受任意类型 T 的右值引用（注意这里是“万能引用”）
template<typename T>
void wrapper(T&& arg) {
    // 使用 std::forward 保留参数的值类别（左值/右值）进行传递
    process(std::forward<T>(arg));  // 完美转发
}
```

```
// 接收左值引用的版本
void process(int& x) { std::cout << "Lvalue: " << x << "\n"; }

// 接收右值引用的版本
void process(int&& x) { std::cout << "Rvalue: " << x << "\n"; }

int main() {
    int a = 10;
    wrapper(a);   // a 是左值 → T 推导为 int& → wrapper<int&>(int&) → forward 保留为左值
    wrapper(20);  // 20 是右值 → T 推导为 int  → wrapper<int>(int&&) → forward 保留为右值

    return 0;
}

============================================================
Lvalue: 10
Rvalue: 20
============================================================
```

用途：在模板函数中保持参数原始类型（左值/右值）。

swap：交换两个对象的值

高效交换

```
#include <utility>
#include <vector>

int main() {
    std::vector<int> vec1 = {1, 2, 3};
    std::vector<int> vec2 = {4, 5};

    std::swap(vec1, vec2);  // 交换内容

    // vec1 = {4, 5}, vec2 = {1, 2, 3}
    return 0;
}
```

底层优化：对标准库容器（如 vector）通常为 O(1) 时间复杂度（交换指针）

exchange：赋值并返回旧值

原子性操作模拟

```
#include <utility>
#include <iostream>

int main() {
    int x = 10;
    int old_x = std::exchange(x, 20);  // x=20，返回旧值10
```

```
    std::cout << "old_x: " << old_x << ", x: " << x << "\n";
    return 0;
}

============================================================
输出：
old_x: 10, x: 20
用途：状态机更新、线程安全编程。
============================================================
```

`<functional>` 实用功能

C++11 标准库中的 `<functional>` 头文件提供了一组工具，用于支持函数式编程风格，包括函数对象、函数包装器、函数适配器、引用包装器、成员函数适配器等功能。这些工具在泛型编程、回调机制、算法调用、标准容器等场景中具有广泛用途。

在 C++ 中，`<functional>` 头文件中的 std::function 是一个通用的多态函数包装器，用于封装任意可调用对象（callable objects），包括普通函数、Lambda 表达式、函数指针、成员函数指针以及具有 operator() 的函数对象（即仿函数）。它提供了一种类型安全的方式来存储、传递和调用这些可调用对象，广泛应用于需要动态函数调用的场景。在算法竞赛中，std::function 虽然不是最核心的工具，但在需要动态传递函数或实现复杂逻辑时非常有用。

定义：
std::function 是一个模板类，声明在 头文件中，定义如下：

```
template<class R, class... Args>
class function<R(Args...)>;
```

- R：函数返回类型。
- Args...：函数参数类型列表。
- R(Args...)：表示函数签名。

功能：

- 包装任意符合指定函数签名的可调用对象。
- 提供统一的调用接口，隐藏底层实现细节。
- 支持类型擦除（type erasure），允许不同类型的可调用对象存储在同一 std::function 对象中。

支持的可调用对象：

1. **普通函数**：全局函数或静态函数。
2. **函数指针**：指向普通函数的指针。
3. **Lambda 表达式**：C++11 引入的匿名函数。
4. **仿函数（函数对象）**：定义了 operator() 的类或结构体。
5. **成员函数指针**：类成员函数（需绑定对象）。
6. **绑定表达式**：通过 std::bind 或 Lambda 创建的绑定对象。

`std::function` 的核心是通过统一的接口存储和调用不同类型的可调用对象。以下是基本用法示例：

示例 1：包装普通函数

```cpp
#include <functional>
#include <iostream>

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

int main() {
    std::function<int(int, int)> func = add; // 包装普通函数
    std::cout << func(3, 4) << "\n"; // 输出: 7
}
```

示例 2：包装 Lambda 表达式

```cpp
#include <functional>
#include <iostream>

int main() {
    std::function<int(int, int)> func = [](int a, int b) { return a + b; };
    std::cout << func(3, 4) << "\n"; // 输出: 7
}
```

示例 3：包装仿函数

```cpp
#include <functional>
#include <iostream>

struct Multiplier {
    int operator()(int a, int b) const { return a * b; }
};

int main() {
    std::function<int(int, int)> func = Multiplier{};
    std::cout << func(3, 4) << "\n"; // 输出: 12
}
```

示例 4：包装成员函数

成员函数需要绑定对象（通过 `std::bind` 或 Lambda）。

```cpp
#include <functional>
#include <iostream>
using namespace std;
struct Calculator {
    int add(int a, int b, int c) const { return a + b + c; }
};

int main() {
    Calculator calc;
    // placeholders 是命名空间，里面放了一些预定义的占位符
    // func 是&Calculator::add函数经过bind绑定适配之后得到的结果，func是可调用的对象
    // 将 c 绑定为100
    //注意：在绑定时bind参数的顺序和被绑定函数的顺序是一一对应的
    function<int(int, int)> func = bind(&Calculator::add, &calc,
placeholders::_1,placeholders::_2,100);
    // 调用时：参数传递的位置和占位_1,_2依次对应，不是和顺序对应
    cout << func(3, 4) << "\n"; // 输出: 107
```

```
    // 使用 Lambda 更简洁
    function<int(int, int)> func2 = [&calc](int a, int b) { return calc.add(a,
b, 100); };
    cout << func2(3, 4) << "\n"; // 输出: 107
}
```

<iterator>的实用功能

遍历

```
// std::next - 向后移动迭代器，不修改原始迭代器
vector<int> v{1, 2, 3, 4};
auto it = next(v.begin(), 2); // 指向 3
cout << *it << endl; // 输出 3

// std::prev - 向前移动迭代器，不修改原始迭代器
vector<int> v{1, 2, 3, 4};
auto it = prev(v.end(), 2); // 指向 3
cout << *it << endl; // 输出 3

// std::advance - 原地移动迭代器，会修改原迭代器
vector<int> v{1, 2, 3, 4};
auto it = v.begin();
advance(it, 2); // it 现在指向 3
cout << *it << endl; // 输出 3

// std::distance - 计算两个迭代器之间的距离（元素个数）
vector<int> v{1, 2, 3, 4};
auto it = next(v.begin(), 3); // 指向 4
int d = distance(v.begin(), it); // d = 3
cout << d << endl; // 输出 3

// std::iter_swap - 交换两个迭代器指向的元素值
vector<int> v{1, 2, 3, 4};
iter_swap(v.begin(), v.begin() + 2); // 交换 1 和 3
// v 变为 {3, 2, 1, 4}
```

输入输出：`istream_iterator` 和 `ostream_iterator` 简化批量操作。

```
// 从标准输入读取一串整数到 vector 中（输入以非整数结束，如 Ctrl+D）
vector<int> v(istream_iterator<int>(cin), istream_iterator<int>());

// 使用 ostream_iterator 将 vector 内容写入标准输出，以空格分隔
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
```

插入

```
// back_inserter - 从容器末尾插入元素
vector<int> v{1, 2};        // 原始容器
vector<int> src{3, 4};      // 要插入的内容

// 将 src 的元素复制到 v 尾部，自动调用 v.push_back()
copy(src.begin(), src.end(), back_inserter(v));
```

```cpp
// front_inserter – 从容器头部插入元素（需要支持 push_front）
list<int> lst{1, 2};        // list 支持头插
vector<int> src{3, 4};      // 要插入的内容

// 将 src 的元素按顺序插入 lst 头部（即先插 3 再插 4 → 最终顺序 4, 3, 1, 2）
copy(src.begin(), src.end(), front_inserter(lst));

// inserter – 向指定位置插入（支持 insert(pos, val)）

set<int> s{1, 2};           // 有序集合，插入元素后会自动排序
vector<int> src{3, 4};      // 要插入的内容

// inserter(s, s.begin()) 创建一个插入迭代器，插入到集合中（位置对 set 无影响）
copy(src.begin(), src.end(), inserter(s, s.begin()));
// 结果：1 2 3 4（自动排序）
```

`<regex>`的核心功能与竞赛用途

我来详细讲解 **C++11 中的正则表达式（regex）**，从基础概念到实际应用，尽量清晰、全面，并附上代码示例。C++11 引入了 `<regex>` 头文件，提供了强大的正则表达式支持，用于字符串匹配、搜索、替换等操作。以下内容将逐步展开：

---

1. **什么是正则表达式?**

正则表达式（Regular Expression，简称 regex）是一种描述字符串模式的工具，用于匹配、搜索或替换符合特定规则的字符串。例如，验证邮箱格式、提取电话号码、替换文本中的特定模式等。

在 C++11 中，`<regex>` 提供了标准化的正则表达式支持，基于 ECMAScript（类似 JavaScript 的正则语法）作为默认语法。

---

2. **C++11 `<regex>` 核心组件**

`<regex>` 库包含以下核心类和函数：

2.1 **核心类**

- `std::regex`：表示一个正则表达式对象，存储正则表达式模式。
- `std::smatch`：存储匹配结果（用于 `std::string` 的匹配）。
- `std::cmatch`：存储匹配结果（用于 C 风格字符串 `const char*`）。
- `std::regex_iterator`：用于迭代字符串中的所有匹配。
- `std::regex_token_iterator`：用于迭代匹配的子字符串或非匹配部分。

2.2 **主要函数**

- `std::regex_match`：检查整个字符串是否完全匹配正则表达式。
- `std::regex_search`：在字符串中搜索第一个匹配的子字符串。
- `std::regex_replace`：替换字符串中匹配正则表达式的部分。

2.3 **正则表达式语法选项**

C++11 支持多种正则表达式语法，通过 `std::regex::flag_type` 指定：

- `std::regex::ECMAScript`（默认）：类似 JavaScript 的语法。
- `std::regex::basic`：POSIX 基本正则表达式。
- `std::regex::extended`：POSIX 扩展正则表达式。
- `std::regex::awk`：AWK 风格。
- `std::regex::grep`：Grep 风格。
- `std::regex::egrep`：Egrep 风格。

通常使用默认的 `ECMAScript`，因为它功能强大且语法灵活。

---

3. **正则表达式基本语法（ECMAScript）**

以下是 ECMAScript 语法中常用的正则表达式元素：

3.1 **字符匹配**

- **普通字符**：如 `a`, `1`, `@` 直接匹配自身。
- **点号 (`.`)**：匹配任意单个字符（除换行符 `\n` 外）。
- **转义字符**：如 `\.` 匹配点号本身，`\\` 匹配反斜杠。

3.2 **字符集**

- `[abc]`：匹配字符集中的任意一个字符（如 `a`, `b`, 或 `c`）。
- `[^abc]`：匹配不在字符集中的任意字符。
- `[a-z]`：匹配小写字母范围。
- `[0-9]`：匹配数字范围。

3.3 **预定义字符类**

- `\d`：匹配任意数字（等价于 `[0-9]`）。
- `\w`：匹配任意单词字符（字母、数字、下划线，等价于 `[a-zA-Z0-9_]`）。
- `\s`：匹配任意空白字符（空格、制表符、换行等）。
- `\D`, `\W`, `\S`：分别表示非数字、非单词字符、非空白字符。

3.4 **量词**

- `*`：匹配 0 次或多次。
- `+`：匹配 1 次或多次。
- `?`：匹配 0 次或 1 次。
- `{n}`：匹配恰好 n 次。
- `{n,}`：匹配至少 n 次。
- `{n,m}`：匹配 n 到 m 次。

3.5 **分组与捕获**

- `(...)`：定义一个捕获组，用于提取匹配的部分。
- `(?:...)`：非捕获组，仅用于分组，不保存匹配结果。
- `|`：逻辑或，如 `cat|dog` 匹配 `cat` 或 `dog`。

3.6 **锚点**

- `^`：匹配字符串开始。
- `$`：匹配字符串结束。
- `\b`：匹配单词边界。

3.7 **修饰符**

C++11 支持的修饰符（通过 `std::regex` 的标志）：

- `std::regex::icase`：忽略大小写。
- `std::regex::multiline`：多行模式，`^` 和 `$` 匹配每行开头和结尾。

---

4. **C++11 正则表达式使用步骤**

使用 C++11 的 `<regex>` 通常遵循以下步骤：

1. 包含 `<regex>` 头文件。
2. 创建 `std::regex` 对象，指定正则表达式模式。
3. 使用 `std::regex_match`、`std::regex_search` 或 `std::regex_replace` 处理字符串。
4. 通过 `std::smatch` 或 `std::cmatch` 获取匹配结果。

---

5. **详细代码示例**

以下是几个常见场景的代码示例，展示如何使用 C++11 正则表达式。

5.1 **完整字符串匹配（`std::regex_match`）**

检查字符串是否完全匹配正则表达式。

```
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string str = "12345";
    std::regex pattern("\\d{5}"); // 匹配 5 个数字

    if (std::regex_match(str, pattern)) {
        std::cout << "String is exactly 5 digits.\n";
    } else {
        std::cout << "String does not match.\n";
    }

    return 0;
}
```

输出：

```
String is exactly 5 digits.
```

解释：

- `\\d{5}` 表示 5 个连续数字。
- `std::regex_match` 要求整个字符串完全匹配模式。

5.2 **搜索子字符串（`std::regex_search`）**

在字符串中查找第一个匹配的子字符串，并提取捕获组。

```
#include <iostream>
#include <regex>
#include <string>

int main() {
```

```
    std::string str = "My email is user123@example.com.";
    std::regex pattern("(\\w+)@([\\w\\.]+)"); // 匹配邮箱格式
    std::smatch match;

    if (std::regex_search(str, match, pattern)) {
        std::cout << "Found email: " << match[0] << "\n"; // 完整匹配
        std::cout << "Username: " << match[1] << "\n";   // 第一个捕获组
        std::cout << "Domain: " << match[2] << "\n";     // 第二个捕获组
    } else {
        std::cout << "No email found.\n";
    }

    return 0;
}
```

输出:

```
Found email: user123@example.com
Username: user123
Domain: example.com
```

解释:

- `(\\w+)@([\\w\\.]+)` 匹配邮箱格式，`(\\w+)` 捕获用户名，`([\\w\\.]+)` 捕获域名。
- `std::smatch` 存储匹配结果，`match[0]` 是完整匹配，`match[1]` 和 `match[2]` 是捕获组。

5.3 **替换匹配内容（`std::regex_replace`）**

将字符串中匹配的部分替换为指定内容。

```
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string str = "My phone is 123-456-7890 and 987-654-3210.";
    std::regex pattern("\\d{3}-\\d{3}-\\d{4}"); // 匹配电话号码格式
    std::string result = std::regex_replace(str, pattern, "XXX-XXX-XXXX");

    std::cout << "After replacement: " << result << "\n";

    return 0;
}
```

输出:

```
After replacement: My phone is XXX-XXX-XXXX and XXX-XXX-XXXX.
```

解释:

- `\\d{3}-\\d{3}-\\d{4}` 匹配类似 `123-456-7890` 的电话号码。
- `std::regex_replace` 将匹配的电话号码替换为 `XXX-XXX-XXXX`。

5.4 **迭代所有匹配（`std::regex_iterator`）**

查找字符串中所有匹配的子字符串。

```cpp
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string str = "Dates: 2023-10-01, 2024-05-04, 2025-12-31";
    std::regex pattern("\\d{4}-\\d{2}-\\d{2}"); // 匹配日期格式
    std::sregex_iterator it(str.begin(), str.end(), pattern);
    std::sregex_iterator end;

    while (it != end) {
        std::cout << "Found date: " << it->str() << "\n";
        ++it;
    }

    return 0;
}
```

输出:

```
Found date: 2023-10-01
Found date: 2024-05-04
Found date: 2025-12-31
```

解释:

- `std::sregex_iterator` 遍历字符串中的所有匹配。
- `it->str()` 返回当前匹配的字符串。

5.5 忽略大小写匹配

使用 `std::regex::icase` 忽略大小写。

```cpp
#include <iostream>
#include <regex>
#include <string>

int main() {
    std::string str = "Hello WORLD";
    std::regex pattern("hello world", std::regex::icase); // 忽略大小写

    if (std::regex_match(str, pattern)) {
        std::cout << "Match found (case-insensitive).\n";
    } else {
        std::cout << "No match.\n";
    }

    return 0;
}
```

输出:

```
Match found (case-insensitive).
```

解释:

- `std::regex::icase` 使正则表达式匹配时忽略大小写。

---

6. **异常处理**

正则表达式操作可能抛出异常，需妥善处理：

- `std::regex_error`：当正则表达式语法错误或资源不足时抛出。

示例：

```
#include <iostream>
#include <regex>
#include <string>

int main() {
    try {
        std::regex pattern("[a-z+"); // 错误的正则表达式（括号未闭合）
        std::cout << "This won't be printed.\n";
    } catch (const std::regex_error& e) {
        std::cout << "Regex error: " << e.what() << "\n";
    }

    return 0;
}
```

输出：

```
Regex error: The expression contained mismatched [ and ].
```

建议：总是用 `try-catch` 包装正则表达式操作，尤其是用户输入的模式。

---

7. **性能注意事项**

- **预编译正则表达式**：`std::regex` 对象的构造开销较大，尽量复用已编译的 `std::regex` 对象。
- **避免复杂模式**：过于复杂的正则表达式可能导致性能问题，尤其是回溯较多时。
- **选择合适的函数**：如果只需要检查是否匹配，使用 `std::regex_match` 或 `std::regex_search` 而不是迭代器。

---

8. **常见应用场景**

- **验证输入**：如邮箱、电话号码、邮编等。
- **文本解析**：从日志中提取时间戳、IP 地址等。
- **文本替换**：批量替换文件中的特定模式。
- **数据清洗**：去除不符合格式的字符串。

---

9. **局限性与替代方案**

- **性能**：C++11 的 `<regex>` 在某些场景下性能不如 Boost.Regex 或 PCRE 库。
- **功能**：不支持某些高级特性（如环视断言），需要时可考虑 Boost.Regex。
- **跨平台一致性**：不同编译器对 `<regex>` 的实现可能有细微差异，建议测试。

---

10. **总结**

C++11 的 `<regex>` 库提供了一种标准化的方式来处理正则表达式，适用于大多数字符串处理需求。通过 `std::regex_match`、`std::regex_search` 和 `std::regex_replace`，结合捕获组和迭代器，可以实现复杂的字符串操作。关键点：

- 熟悉 ECMAScript 语法。
- 合理使用捕获组和标志（如 `icase`）。
- 注意异常处理和性能优化。