

C++ note

STL容器算法

array

vector

特点：

常用操作和函数：

1. 声明
2. 添加元素
3. 访问元素
4. 获取大小
- 5 插入
6. 删除元素
7. 清空向量
8. 检查是否为空
9. 迭代遍历

二维vector

定义二维vector

访问元素

遍历二维vector

动态调整大小

注意事项

queue

1. 包含头文件
2. 创建队列
3. 入队 (push)
4. 出队 (pop)
5. 访问队首元素
6. 检查队列是否为空
7. 获取队列大小

map

核心特性

常见成员函数

1. 插入/修改
2. 访问
3. 删除
4. 容量
5. 迭代器

基本操作示例

与 unordered_map 对比

适用场景

例题

STL 容器（如 vector, string）相关用法：

1. pop_back()
 2. push_back()
 3. back()
 4. front()
 5. begin()
 6. getline()
1. begin() 和 end()

Range-based for loop

1. 基本语法
2. 执行流程

数据操作

sort

sort 接受的三个参数：

使用自定义比较函数

1. 使用自定义比较函数（降序排序）
2. 使用 lambda 表达式作为比较函数

排序对象（结构体、类）的示例

按结构体字段排序

排序数组

sort() 的其他特点

排序与并查集的例题

switch

基本语法:

主要组成部分:

示例代码:

atoi

语法

特点

示例

注意事项

用stoi 替代

并查集

一、核心操作实现

1. 初始化
2. Find操作（查找代表元素）
3. Union操作（合并集合）

二、优化方法

1. 按秩合并（Union by Rank）

三、时间复杂度

四、应用示例

题目：修改数组

数据类型与变量

数据类型大小

数据类型后缀

1. f 和 F
2. l 和 L
3. u 和 U
4. ll 和 LL
5. ul、UL、lu 和 LU
6. ull 和 ULL
7. ulll 和 ULLL

示例代码：

8. z (用于指针类型)

数据类型转换

1. 隐式转换

2. 显式转换

2.1 static_cast

2.2 dynamic_cast

2.3 const_cast

2.4 reinterpret_cast

字符 (char) 与 字符串 (string)

字符与字符串之间的转换

字符串操作函数：

1. substr

2. empty

3. size

4. erase

5. size() 或 length()

6. char *strncpy(char *s1, const char *s2, size_t n);

7. char *strncat(char *s1, const char *s2, size_t n);

8. int strcmp(const char *s1, const char *s2, size_t n);

9. size_t strlen(const char *s);

10. char *strtok(char *s1, const char *s2);

输入

1.cin

2.cin.get()

3.cin.getline()

4.getline()

5.cin.ignore()

枚举类型 (enum)

1. 基本的枚举类型

2. 指定枚举常量的值

3. 枚举底层类型

4. 枚举和数组的结合

举例：

5.enum class

特性

示例代码

enum class 的优势

引用 (Reference)

1. 引用的基本语法

2. 引用的特点：

3. 引用作为函数参数

4. 引用返回值

符号常量 (constant)

1. #define 宏常量

2. const 常量

自动存储类别和静态存储类别

1. 自动存储类别 (auto)

2. 静态存储类别 (static)

左值 (Lvalue) 和右值 (Rvalue)

位运算与表达式

位运算

1. 位与运算符 &

2. 位或运算符 |

3. 位异或运算符 ^

4. 取反运算符 ~

5. 左移运算符 <<

6. 右移运算符 >>

Lambda 表达式

Lambda 表达式的基本语法

1. 基本示例

2. 带参数的 Lambda

3. 捕获外部变量

3.1 按值捕获

3.2 按引用捕获

3.3 捕获所有变量

4. Lambda 的返回类型

4.1 自动推导返回类型

4.2 显式指定返回类型

5. Lambda 表达式的实际应用

ASCII码表

分区记忆

A. 控制字符 (0-31)

B. 数字字符 (48-57)

C. 大写字母 (65-90)

D. 小写字母 (97-122)

E. 标点符号和其他可打印字符 (32-47, 58-64, 91-96, 123-126)

关键值和偏移量

逻辑运算符的短路行为

应用示例：

指针与内存管理

指针

1. 指针的本质

2. 指针的声明和初始化

3. 指针的操作

4. 指针的用途

5. 指针的常见陷阱

6. 指针与数组的关系

7. 多级指针

8. &a++与 &+a

通用指针

1. void*指针的定义

2. void*的基本用法

3. 使用void*指针访问数据

4. 与void* 的类型转换

常量指针 (constant pointer) 和 指针常量 (pointer to constant)

1. 常量指针 (Constant Pointer)

示例：

2. 指针常量 (Pointer to Constant)

示例：

3. 同时使用常量指针和指针常量

. 和 ->

1. .的使用：

2. ->的使用：

函数与作用域

默认参数 (Default Arguments)

作用域解析运算符 ::

1. 访问全局变量或函数

2. 定义类外的类成员函数

3. 访问命名空间中的成员

4. 访问类的静态成员

5. 类的继承中的作用域

内联函数 (inline function)

内联函数的优点

内联函数的限制

使用场景

示例

函数重载 (Function Overloading)

1. 函数重载的规则

2. 函数重载的示例

3. 重载函数的选择

4. 重载的注意事项

函数模版 (Function Template)

1. 函数模板的定义

2. 如何使用函数模板

3. 模板参数的类型

4. 显式指定模板参数

5. 函数模板的特化 (Template Specialization)

6. 注意事项

函数调用堆栈

1. 代码区 (Text Segment)

示例：

2. 全局数据区 (Data Segment)

示例：

3. 堆区 (Heap)

示例：

4. 栈区 (Stack)

示例：

内存分区模型

1. 程序运行前

2. 程序运行后

3. new操作符

类和对象

接口分离

封装

意义：

struct和class区别：

对象的初始化和清理：

构造函数和析构函数

构造函数的分类及调用

拷贝构造函数调用时机

转换构造函数

拷贝构造函数和转换构造函数区别

构造函数调用规则

深拷贝与浅拷贝

初始化列表

类对象作为类成员

构造与析构的顺序

静态成员

对象模型 (类如何占用内存)

this指针

空指针访问成员函数

const修饰成员函数

友元

全局函数做友元

类做友元

成员函数做友元

运算符重载

加号运算符重载

左移运算符重载

递增运算符重载

赋值运算符重载

关系运算符重载

函数调用运算符重载

继承

基本语法

继承方式

对象模型

构造和析构顺序

继承同名（静态）成员处理方式

多继承语法

菱形继承

多态

基本概念

对象切片

纯虚函数和抽象类

虚析构和纯虚析构

类模版

链表

零散题目

题目1：构造/析构顺序与虚函数

题目2：继承中的对象切片

[题目3：成员初始化顺序](#)

[题目4：虚函数表与内存布局](#)

[题目5：静态绑定与默认参数](#)

[题目6：移动语义陷阱](#)

[题目7：类型转换运算符](#)

[题目8：模板类的静态成员](#)

[题目9：析构函数多态](#)

[题目10：const成员函数](#)

文件操作

[文本文件](#)

[写文件](#)

[读文件](#)

[二进制文件](#)

[写文件](#)

[读文件](#)

[文件和流（Files and Streams）](#)

[创建顺序文件](#)

[从顺序文件读取数据](#)

[代码示例（若没有文件，创建文件）](#)

[随机存取文件](#)

[综合示例](#)

异常处理

[处理流程](#)

[try 块：](#)

[throw 表达式：](#)

[catch 块：](#)

[终止模型（Termination Model）：](#)

[示例代码：](#)

[处理标准库抛出的异常](#)

[核心目标](#)

[示例：bad_alloc](#)

[代码：捕获 bad_alloc](#)

异常类型匹配规则

异常规范

栈展开

其他

生成随机数

格式化操作

1.left 和 right

2.setw(int width)

3.setFill(char fill)

4.fixed

5.showpoint

6.scientific

7.setprecision

8.设置bool

9.设置数值的基数

10.设置精度

特别注意

逗号表达式

1.逗号表达式结合性优先级最低

2.左边子表达式的计算和副作用均先于右边子表达式

3.整个表达式的值是最右边子表达式的值

STL容器算法

array

array 是 C++11 引入的一个模板类，它的大小是固定的，并且在编译时就确定。

```
1 #include <array>
2 #include <iostream>
3
4 int main() {
5     array<int, 5> arr = {1, 2, 3, 4, 5};
6
7     // 使用 at() 安全地访问数组元素
8     cout << "Element at index 2: " << arr.at(2) << endl;
9
10    // 使用 size() 获取数组大小
11    cout << "Size of array: " << arr.size() << endl;
12
13    // 使用范围 for 循环遍历数组
14    for (int i : arr) {
15        cout << i << " ";
16    }
17    return 0;
18 }
```

vector

特点：

1. 动态大小：vector 能根据需要动态调整大小。
2. 自动内存管理：vector 自动处理内存的分配和释放，不需要手动管理内存。
3. 随机访问：与数组类似，vector 也支持常量时间的随机访问操作。
4. 支持 STL 算法：vector 与许多标准算法（如sort）无缝兼容。

常用操作和函数:

1. 声明

```
1 #include <vector>
2 vector<int> vec;           // 创建一个存储 int 类型的空向量
3 vector<int> vec(10);      // 创建一个包含 10 个元素的向量，默认初始化为 0
4 vector<int> vec(10, 5);    // 包含 10 个元素，每个元素初始化为 5
```

2. 添加元素

- `push_back`: 在`vector` 的末尾添加一个元素。

```
1 vec.push_back(20); // 向量末尾添加 20
```

3. 访问元素

- `[]` 和`at()`: `[]` 提供类似数组的访问，`at()` 提供边界检查的访问方式。

```
1 int value = vec[0];        // 获取第一个元素
2 int value = vec.at(0);      // 获取第一个元素 (有边界检查，超出范围会抛异常)
```

4. 获取大小

- `size()`: 返回向量中当前元素的个数。

```
1 int size = vec.size(); // 获取向量大小
```

5 插入

- `insert()`: 在向量的某个位置插入元素。

```
1 vec.insert(vec.begin(), 10); // 在向量开头插入 10
```

6. 删除元素

- `pop_back()`: 删除`vector` 中最后一个元素。
- `erase()`: 删除指定位置的元素。

```
1 vec.pop_back();           // 删除最后一个元素  
2 vec.erase(vec.begin()); // 删除向量的第一个元素
```

7. 清空向量

- clear(): 删掉所有元素，size() 将变为 0。

```
1 vec.clear(); // 清空向量
```

8. 检查是否为空

- empty(): 检查向量是否为空。

```
1 bool isEmpty = vec.empty(); // 如果向量为空, 返回 true
```

9. 迭代遍历

```
1 for (int i = 0; i < vec.size(); ++i)  
2     cout << vec[i] << " "; // 使用索引遍历  
3  
4 for (auto it = vec.begin(); it != vec.end(); ++it)  
5     cout << *it << " "; // 使用迭代器遍历  
6  
7 for (int x : vec)  
8     cout << x << " "; // 使用范围 for 循环遍历
```

二维vector

定义二维vector

1. 指定行数和列数初始化

创建一个3行4列的二维 `vector`，初始值为0：

```
1 vector<vector<int>> mat(3, vector<int>(4, 0));  
2 vector<vector<vector<int>>> cube(2, vector<vector<int>>(3, vector<int>(4, 0  
    )));  
3 // 三维的定义, 可以发现多维就是迭代, 这是模版类的使用方法
```

2. 逐行动态添加

创建空二维 `vector`，然后逐行添加：

```
1 vector<vector<int>> mat;
2 mat.push_back({1, 2, 3});           // 添加一行 {1,2,3}
3 mat.push_back(vector<int>(4,5)); // 添加一行四个5
```

访问元素

- 使用双重下标访问元素（注意索引有效性）：

```
1 int value = mat[i][j]; // 访问第i行第j列
2 mat[i][j] = 10;        // 修改元素
```

遍历二维vector

- 使用范围for循环：

```
1 for (const auto& row : mat) {          // 遍历每一行
2     for (int elem : row) {              // 遍历行内元素
3         cout << elem << " ";
4     }
5     cout << endl;
6 }
```

- 使用下标遍历：

```
1 for (size_t i = 0; i < mat.size(); ++i) {
2     for (size_t j = 0; j < mat[i].size(); ++j) {
3         cout << mat[i][j] << " ";
4     }
5     cout << endl;
6 }
```

动态调整大小

- 调整行数：

```
1 mat.resize(5); // 将行数改为5，新增的行默认是空vector
```

- 调整某一行的大小：

```
1 mat[0].resize(5); // 将第一行的列数改为5
```

注意事项

1. 不规则二维结构：各行长度可以不同，适用于类似邻接表的场景。
2. 深拷贝问题：直接赋值会复制所有元素，大二维结构需谨慎操作。
3. 函数传参：建议使用 `const vector<vector<int>>&` 传递以避免拷贝。

queue

1. 包含头文件

```
1 #include <iostream>
2 #include <queue>
```

2. 创建队列

```
1 queue<int> q;
```

3. 入队 (push)

使用`push()`方法将元素添加到队列的末尾：

```
1 q.push(10);
2 q.push(20);
```

4. 出队 (pop)

使用pop()方法从队列的前面移除元素：

```
1 q.pop(); // 移除队首元素 (10)
```

5. 访问队首元素

使用front()方法访问队首元素，但不移除它：

```
1 int frontElement = q.front(); // frontElement 现在是20
```

6. 检查队列是否为空

使用empty()方法检查队列是否为空：

```
1 if (q.empty()) std::cout << "队列为空" << std::endl;
```

7. 获取队列大小

使用size()方法获取队列中元素的数量：

```
1 std::cout << "队列大小: " << q.size() << std::endl;
```

map

map 用于存储键值对 (key-value pairs)，并基于键自动排序元素。

核心特性

1. 有序性

- 元素按键的升序排列（默认使用 `<` 运算符比较键）。
- 支持自定义排序规则，通过提供比较函数或函数对象实现。

2. 底层实现

- 基于红黑树（自平衡二叉搜索树），保证插入、删除和查找操作的时间复杂度为 $O(\log n)$ 。

3. 唯一键

- 每个键在 `map` 中唯一，不可重复。若需允许重复键，应使用 `std::multimap`。

4. 键的类型要求

- 键必须支持严格弱序比较（默认通过 `<` 运算符），或提供自定义比较函数。

常见成员函数

1. 插入/修改

`insert`

- 功能：插入键值对。若键已存在，不覆盖原有值。
- 参数：可以是 `pair` 对象或使用 `make_pair`。
- 返回值：返回一个 `pair<iterator, bool>`，其中：
 - `iterator`：指向插入位置的迭代器。
 - `bool`：表示是否插入成功（`true` 表示插入新键，`false` 表示键已存在）。

```
1 std::map<int, std::string> m;
2 auto ret = m.insert(std::make_pair(1, "one"));
3 if (ret.second) {
4     std::cout << "Inserted: " << ret.first->second << std::endl;
5 }
```

`emplace`

- 功能：直接在容器内构造键值对，避免临时对象拷贝（比 `insert` 更高效）。
- 参数：构造键值对所需的参数。

```
1 auto ret = m.emplace(2, "two"); // 无需手动创建pair
2 if (ret.second) {
3     std::cout << "Emplaced: " << ret.first->second << std::endl;
4 }
```

`operator[]`

- 功能：通过键访问值。若键不存在，自动插入默认值（如 `std::string()`）。

- 注意：可能导致意外插入默认值，需谨慎使用。

```
1 m[3] = "three";           // 插入键3的值
2 std::cout << m[4];       // 键4不存在，自动插入空字符串!
```

2. 访问

at

- 功能：安全访问值。若键不存在，抛出 `std::out_of_range` 异常。
- 适用场景：需要严格检查键是否存在时使用。

```
1 try {
2     std::cout << m.at(5); // 若键5不存在，抛出异常
3 } catch (const std::out_of_range& e) {
4     std::cerr << "Key not found!" << std::endl;
5 }
```

operator[]

- 功能：通过键访问值。若键不存在，自动插入默认值（如 `int()`、`std::string()`）。
- 注意：可能意外修改容器内容，需确保键存在时使用。

find

- 功能：查找键，返回指向该键值对的迭代器。若键不存在，返回 `end()`。
- 适用场景：需要判断键是否存在并获取值。

```
1 auto it = m.find(2);
2 if (it != m.end()) {
3     std::cout << "Found: " << it->second << std::endl;
4 }
```

count

- 功能：返回键的数量（对 `std::map` 始终是 `0` 或 `1`）。
- 适用场景：仅需判断键是否存在，无需获取值。

```
1 if (m.count(3) > 0) {
2     std::cout << "Key 3 exists" << std::endl;
3 }
```

3. 删除

erase

- 功能：删除元素。支持通过键、迭代器或迭代器范围删除。
- 返回值：返回删除的元素数量（对 map 是 0 或 1）。

```
1 m.erase(1);                                // 删除键为1的元素 (返回1)
2 auto it = m.find(2);
3 if (it != m.end()) {
4     m.erase(it);                            // 通过迭代器删除 (无返回值)
5 }
```

4. 容量

size

- 功能：返回容器中元素的数量。

```
1 std::cout << "Size: " << m.size() << std::endl; // 输出元素总数
```

empty

- 功能：判断容器是否为空。

```
1 if (m.empty()) {
2     std::cout << "Map is empty" << std::endl;
3 }
```

5. 迭代器

begin / end

- 功能：返回指向容器第一个元素和尾后位置的迭代器（正向遍历）。

```
1 for (auto it = m.begin(); it != m.end(); ++it) {
2     std::cout << it->first << ":" << it->second << std::endl;
3 }
```

rbegin / rend

- 功能：返回反向迭代器，用于逆序遍历（从最后一个元素到第一个）。

```
1 for (auto rit = m.rbegin(); rit != m.rend(); ++rit) {
2     std::cout << rit->first << ":" << rit->second << std::endl; // 逆序输出
3 }
```

基本操作示例

```
1 #include <iostream>
2 #include <map>
3 #include <string>
4
5 int main() {
6     // 定义map, 键为int, 值为string
7     std::map<int, std::string> m;
8
9     // 插入元素
10    m.insert(std::make_pair(3, "three")); // 使用insert
11    m[1] = "one"; // 使用operator[]
12    m[5] = "five";
13    m[2] = "two";
14
15    // 遍历 (有序输出)
16    for (const auto& pair : m) {
17        std::cout << pair.first << ":" << pair.second << std::endl;
18    }
19    // 输出顺序: 1: one, 2: two, 3: three, 5: five
20
21    // 查找元素
22    auto it = m.find(3);
23    if (it != m.end()) {
24        std::cout << "Found: " << it->second << std::endl; // 输出: Found
25        d: three
26    }
27
28    // 删除元素
29    m.erase(2); // 删除键为2的元素
30
31    // 使用at访问 (键不存在时抛出异常)
32    try {
33        std::cout << m.at(4) << std::endl; // 抛出std::out_of_range
34    } catch (const std::out_of_range& e) {
35        std::cerr << "Key not found." << std::endl;
36    }
37
38    return 0;
}
```

与 `unordered_map` 对比

特性	<code>std::map</code>	<code>std::unordered_map</code>
底层结构	红黑树（有序）	哈希表（无序）
查找复杂度	$O(\log n)$	平均 $O(1)$, 最差 $O(n)$
插入/删除复杂度	$O(\log n)$	平均 $O(1)$, 最差 $O(n)$
内存占用	较高（树结构额外开销）	较低（哈希表可能需扩容）
键类型要求	支持比较（如 <code><</code> 运算符）	支持哈希函数和相等比较
遍历顺序	按键升序	无特定顺序

适用场景

- 需要有序键或范围查询（如遍历某区间内的键）。
- 对插入/删除/查找的稳定性要求较高（避免哈希冲突的最坏情况）。
- 键类型天然有序且无需自定义哈希函数时。

若无需顺序且追求更高性能，优先选择 `std::unordered_map`。

例题

旋转九宫格

<https://www.luogu.com.cn/problem/P10578>

题目：

给定一个 3×3 的九宫格，每个格子内分别含有一个数字，每个格子里的数字互不相同。每步我们可以选择任意一个 2×2 的区域将其顺时针旋转，例如：

例如

```

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

将其旋转右上角，可得：

1	1	5	2
2	4	6	3
3	7	8	9

问最少需要几步才能将给定的状态旋转为

1	1	2	3
2	4	5	6
3	7	8	9

代码：

```
1 #include <iostream>
2 #include <queue>
3 #include <unordered_map>
4 #include <string>
5 #include <vector>
6
7 using namespace std;
8
9 const string target = "123456789";
10 unordered_map<string, int> dist; // 步数
11
12 vector<string> generate_new_states(const string &s) {
13     vector<string> new_states;
14
15     // 左上区域: 0,1,3,4
16     string s1 = s;
17     s1[0] = s[1];
18     s1[1] = s[4];
19     s1[3] = s[0];
20     s1[4] = s[3];
21     new_states.push_back(s1);
22
23     // 右上区域: 1,2,4,5
24     string s2 = s;
25     s2[1] = s[2];
26     s2[2] = s[5];
27     s2[4] = s[1];
28     s2[5] = s[4];
29     new_states.push_back(s2);
23
30     // 左下区域: 3,4,6,7
31     string s3 = s;
32     s3[3] = s[4];
33     s3[4] = s[7];
34     s3[6] = s[3];
35     s3[7] = s[6];
36     new_states.push_back(s3);
37
38     // 右下区域: 4,5,7,8
39     string s4 = s;
40     s4[4] = s[5];
41     s4[5] = s[8];
42     s4[7] = s[4];
43     s4[8] = s[7];
44     new_states.push_back(s4);
45
46     return new_states;
47 }
```

```

48     }
49
50     void preprocess() {
51         queue<string> q;
52         q.push(target);
53         dist[target] = 0;
54
55         while (!q.empty()) {
56             string current = q.front();
57             q.pop();
58
59             int current_dist = dist[current];
60
61             vector<string> next_states = generate_new_states(current);
62
63             for (const string &next : next_states) {
64                 if (dist.find(next) == dist.end()) {
65                     dist[next] = current_dist + 1;
66                     q.push(next);
67                 }
68             }
69         }
70     }
71
72     int main() {
73         ios::sync_with_stdio(false);
74         cin.tie(nullptr);
75
76         preprocess();
77
78         int T;
79         cin >> T;
80         while (T--) {
81             string s;
82             for (int i = 0; i < 3; ++i) {
83                 int a, b, c;
84                 cin >> a >> b >> c;
85                 s += (char)('0' + a);
86                 s += (char)('0' + b);
87                 s += (char)('0' + c);
88             }
89             cout << dist[s] << '\n';
90         }
91
92         return 0;
93     }

```

STL 容器（如 vector, string ）相关用法：

1. pop_back()

pop_back() 用于删除字符串或容器的最后一个元素。

```
1 string str = "Hello!";
2 str.pop_back(); // 删除最后一个字符 str = "Hello"
```

2. push_back()

push_back() 用于在字符串或容器的末尾添加一个元素。

```
1 string str = "Hello";
2 str.push_back('!'); // 在末尾添加字符 '!' str = "Hello!"
```

3. back()

back() 方法返回字符串或容器最后一个元素的引用，但不删除它。

```
1 string str = "Hello!";
2 char lastChar = str.back(); // lastChar = '!'
```

4. front()

front() 方法返回字符串或容器第一个元素的引用，但不删除它。

```
1 string str = "Hello!";
2 char firstChar = str.front(); // firstChar = 'H'
```

5. begin()

`begin()` 方法返回一个迭代器，指向字符串或容器的第一个元素。

```
1 string str = "Hello";
2 auto it = str.begin(); // it 指向 'H'
3 cout << *it << endl; // 输出: H
```

6. `getline()`

`getline()` 用于从输入流中读取一整行字符串，[包括空格，直到遇到换行符。](#)

```
1 #include <iostream>
2 #include <string>
3 int main() {
4     string input;
5     cout << "Enter a line of text: ";
6     getline(cin, input); // 读取整行输入, 注意前面"cin,"
7     cout << "You entered: " << input << endl;
8     return 0;
9 }
```

1. `begin()` 和 `end()`

`begin()`,`end()` 用于获取容器中第一个元素的开始和容器中最后一个元素之后的位置，可以用于STL容器和传统数组。

```
1 vector<int> vec = {10, 20, 30, 40, 50};
2
3 // 使用 begin() 和 end() 手动遍历容器
4 for (auto it = vec.begin(); it != vec.end(); ++it) {
5     cout << *it << " "; // 因为返还的是位置, 所以需要用 * 解引用
6     // 输出: 10 20 30 40 50
7 }
8
```

Range-based `for` loop

适用于传统数组、stl容器和C++11之后的其他容器。

1. 基本语法

```
1 for (declaration : container) {  
2     // code to process each element  
3 }
```

- `declaration`: 用于声明遍历容器中每个元素的类型。
- `container`: 你要遍历的容器, 例如数组、`vector`、`list` 等。

2. 执行流程

假设 `vec` 是一个包含整数的数组或容器, 假设 `vec` 包含 `{10, 20, 30, 40, 50}`, 则在 `for (int i : vec)` 中:

1. 循环开始时, `i` 会依次被赋值为容器 `arr` 中的每个元素。
2. 在每次循环中, `i` 会输出到控制台, 直到容器中的所有元素都被遍历一遍。

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> vec = {10, 20, 30, 40, 50};
7
8     // 使用range-based for 循环遍历并输出元素
9     for (int i : vec) {
10         cout << i << " "; // 输出每个元素并加一个空格
11     }
12     // 输出: 10 20 30 40 50
13
14     for (auto i : vec) { // 使用 auto 来自动推导元素的类型
15         cout << i << " ";
16     }
17
18     for (int& i : vec) { // 修改元素 (使用引用)
19         i *= 2; // 将每个元素的值翻倍
20     }
21
22     for (const int& i : vec) { // 不修改元素 (使用常量引用), 避免复制开销
23         cout << i << " "; // 输出每个元素, 不修改
24     }
25
26 }
```

数据操作

sort

位于 `<algorithm>` 头文件中，适用于多种容器类型，比如数组、`vector` 等。、

sort 接受的三个参数：

- 第一个参数是容器的起始迭代器（`nums.begin()`）。
- 第二个参数是容器的结束迭代器（`nums.end()`）。
- 第三个参数是一个函数（或可调用对象），用来定义如何比较容器中的元素。这个函数决定了元素排序的顺序。

使用自定义比较函数

`sort()` 默认按升序排序，但可以提供一个自定义的比较函数来改变排序的规则。

1. 使用自定义比较函数（降序排序）

```
1 // 自定义比较函数：降序排序
2 - bool compare(int a, int b) {
3     return a > b;
4     // 如果 a > b 返回 true，就将 a 排在前面
5     // 这里的 a, b 就相当于相邻的元素
6 }
7
8 - int main() {
9     vector<int> vec = {5, 2, 8, 1, 3};
10
11    // 使用自定义比较函数进行降序排序
12    sort(vec.begin(), vec.end(), compare);
13
14    return 0;
15 }
```

2. 使用 lambda 表达式作为比较函数

使用 C++11 引入的 lambda 表达式来快速定义一个排序规则，而无需单独编写一个函数：

```
1  vector<int> vec = {5, 2, 8, 1, 3};  
2  
3  // 使用 lambda 表达式进行降序排序  
4  sort(vec.begin(), vec.end(), [](int a, int b) {  
5      return a > b; // 降序  
6  });
```

排序对象（结构体、类）的示例

如果要排序的元素是自定义类型，比如结构体或类对象，可以根据某个字段进行排序。

按结构体字段排序

```
1  struct Person {  
2      string name;  
3      int age;  
4  };  
5  
6  int main() {  
7      vector<Person> people = {  
8          {"Alice", 30},  
9          {"Bob", 25},  
10         {"Charlie", 35}  
11     };  
12  
13     // 按照年龄进行升序排序  
14     sort(people.begin(), people.end(), [](const Person& a, const Person& b)  
15     ) {  
16         return a.age < b.age;  
17     });  
18  
19     // 输出排序后的结果  
20     for (const auto& person : people) {  
21         cout << person.name << ":" << person.age << endl;  
22     }  
23  
24     return 0;  
25 }
```

排序数组

你也可以对原生数组进行排序，只需传入数组的开始和结束指针：

```

1 int arr[] = {5, 2, 8, 1, 3};
2 int n = sizeof(arr) / sizeof(arr[0]); // 计算数组长度
3 //这里的sizeof返回的是总字节数，所以还要除一个单个元素的字节数
4
5 sort(arr, arr + n); //注意下标表示

```

当我们需要对两个（甚至更多）数组联动的时候：

```

1 struct node {
2     int x, y;
3 }
4
5 int main() {
6     int n;
7     node arr[20];
8     cin >> n;
9     for (int i = 1; i <= n; i++)
10    cin >> arr[i].x >> arr[i].y; //要习惯直接使用结构体
11
12    // 使用 lambda 表达式对 arr 数组进行排序
13    sort(arr, arr + n, [](node const& a, node const& b) {
14        if (a.x != b.x) {
15            return a.x < b.x; // 根据 x 排序
16        }
17        return a.y < b.y; // 如果 x 相同，再根据 y 排序
18    }); //这里就要很自然地理解a,b是相邻的两个元素
19
20    return 0;
21 }

```

sort() 的其他特点

- 稳定性：sort() 是不稳定的排序算法。可以使用 **stable_sort()** 作为稳定排序算法。
- 空容器：如果容器为空，sort() 不会进行任何操作，程序也不会报错。

排序与并查集的例题

<https://www.luogu.com.cn/problem/P10577>

题目描述

在一条数轴上有n个点，点i的初始位置为 p_i 。每个点会选择距离自己最近的另一个点作为目标，并开始向目标移动。

- 如果最近的点有两个（即左边和右边距离相等），则选择左边的那个。
- 每次移动只能向左或向右移动 1个单位。
- 当两个点之间的距离为1，且它们是彼此的目标时，靠右的点会跳到左边点的位置，左边的点保持不动，它们就完成了“集结”。
- 每个点会不断朝自己的目标移动，直到与目标完成集结后停止。

请你输出所有点最终的位置。

输入格式

第一行一个整数 n ，表示点的个数。

第二行 n 个整数 p_1, p_2, \dots, p_n ，表示每个点的初始位置。

输出格式

一行 n 个整数，表示每个点完成集结后的最终位置。

代码：

```

1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4 int const N = 1e5 + 5;
5 int fa[N];
6
7 struct stu {
8     int pos, id;
9 } s[N]; //等效 stu s[N];
10
11 int get(int x) {
12     if (fa[x] == x) return x;
13     return fa[x] = get(fa[x]);
14 }
15
16 bool cmp1(stu x, stu y) {
17     return x.pos < y.pos;
18 }
19 bool cmp2(stu x, stu y) {
20     return x.id < y.id;
21 }
22
23 int main() {
24     int n;
25     cin >> n;
26     for (int i = 1; i <= n; i++) {
27         cin >> s[i].pos;
28         s[i].id = i;
29     }
30     sort(s + 1, s + n + 1, cmp1);
31     fa[1] = 2;
32     fa[n] = n - 1; //边界优化
33     for (int i = 2; i <= n - 1; i++) {
34         int lp = s[i - 1].pos, p = s[i].pos, rp = s[i + 1].pos;
35         if (p - lp <= rp - p) fa[i] = i - 1; //判断左右哪一个更近
36         else fa[i] = i + 1;
37     }
38     for (int i = 1; i <= n - 1; i++) {
39         if (fa[i] == i + 1 && fa[i + 1] == i) {
40             fa[i] = i;
41             fa[i + 1] = i + 1;
42             int p = (s[i].pos + s[i + 1].pos) / 2;
43             s[i].pos = s[i + 1].pos = p;
44         } // 如果互为节点, 向中靠拢
45     }
46     for (int i = 1; i <= n; i++) {
47         if (fa[i] != i) {

```

```
48     int root = get(i);
49     s[i].pos = s[root].pos;
50 } // 更新节点位置
51 }
52 sort(s + 1, s + n + 1, cmp2); //结构体的作用，逆向回去
53 for (int i = 1; i <= n; i++) {
54     cout << s[i].pos << " ";
55 }
56 return 0;
57 }
```

switch

基本语法：

```
1 switch (expression) {  
2     case constant1:  
3         // 如果 expression == constant1, 执行这部分代码  
4         break; // break不可轻易省略; 如果break消失了, 就无条件继续下一个case  
5     case constant2:  
6         // 如果 expression == constant2, 执行这部分代码  
7         break;  
8     default:  
9         // 如果没有匹配到任何 case, 执行这部分代码  
10 }
```

主要组成部分：

1. **expression**: 通常是一个整型、字符型或枚举类型的值，不能是浮点型或字符串。
2. **case constant**: 每个 **case** 后面跟一个常量值，如果 **expression** 的值与某个 **case** 的常量值匹配，则执行该 **case** 下的代码。注意ASCII产生潜在的影响。
3. **break**: 用于退出 **switch** 语句。如果没有 **break** 语句，程序会继续无条件执行下一个 **case** 中的代码，即“穿透现象”，通常会导致错误，除非有特殊需求。
4. **default**: 如果没有任何 **case** 匹配 **expression** 的值，**default** 中的代码会被执行。**default** 可以放在任意位置，但通常放在最后。

示例代码：

```
1 int day = 3;  
2 switch (day) {  
3     case 1:  
4         cout << "Monday" << endl;  
5         break;  
6     case 2:  
7         cout << "Tuesday" << endl;  
8         break;  
9     case 3:  
10        cout << "Wednesday" << endl;  
11        break;  
12    default:  
13        cout << "Invalid day" << endl;  
14        break;  
15 }
```

atoi

`atoi` 是一个标准库函数，需要引用 `#include <cstdlib>`，用于将 C 风格的字符串（即以 `\0` 结尾的字符数组）转换为整数类型 `int`。

语法

```
1 int atoi(const char* str);
```

特点

1. 忽略前导空格：`atoi` 会忽略字符串开头的空格字符。
2. 处理符号：可以识别正负号（`+` 或 `-`）。
3. 无错误处理：`atoi` 并不会提供详细的错误信息。如果字符串不能转换为整数，它将返回 `0`，但不能区分字符串是否真的表示 `0`，还是无法转换的情况。

示例

```
1 #include <iostream>
2 #include <cstdlib> // 引入atoi
3
4 int main() {
5     const char* str1 = "12345";      // 正常的数字字符串
6     const char* str2 = "-678";       // 带负号的数字字符串
7     const char* str3 = "abc123";     // 无效的字符串
8     const char* str4 = "";          // 空字符串
9
10    std::cout << "str1: " << atoi(str1) << std::endl; // 输出 12345
11    std::cout << "str2: " << atoi(str2) << std::endl; // 输出 -678
12    std::cout << "str3: " << atoi(str3) << std::endl; // 输出 0
13    std::cout << "str4: " << atoi(str4) << std::endl; // 输出 0
14
15    return 0;
16 }
```

注意事项

- `atoi` 只适用于 C 风格字符串。如果你使用 `std::string`，可以先调用 `std::string::c_str()` 将其转换为 C 风格字符串，或者直接使用 `std::stoi`。
- `atoi` 不会检测溢出。如果输入值超出了 `int` 类型的表示范围，结果会是未定义的行为。

用`stoi` 替代

`stoi` (在 `<string>` 头文件中) 不仅支持字符串转换为整数，还能抛出异常 (例如 `invalid_argument` 或 `out_of_range`) 以便更好地处理错误。

```

1 #include <iostream>
2 #include <string>
3
4 int main() {
5     std::string str = "12345";
6     try {
7         int num = std::stoi(str);
8         std::cout << "Converted number: " << num << std::endl;
9     } catch (const std::invalid_argument& e) {
10        std::cout << "Invalid argument: " << e.what() << std::endl;
11    } catch (const std::out_of_range& e) {
12        std::cout << "Out of range: " << e.what() << std::endl;
13    }
14
15    return 0;
16 }
```

并查集

并查集是一种用于管理元素分组关系的数据结构，支持以下两种核心操作：

- **Find**: 查询元素所属的集合（通常返回集合的“代表元素”）。
- **Union**: 合并两个元素所在的集合。

常用于解决 **动态连通性问题**，例如：

- 网络中的节点连接状态判断
- 图的连通分量统计

一、核心操作实现

1. 初始化

每个元素初始指向自己，表示自己是独立的集合。

```
1 int parent[N]; // N为元素数量
2 void init() {
3     for (int i = 0; i < N; i++) parent[i] = i;
4 }
```

2. Find操作（查找代表元素）

查找元素所属集合的根节点（代表元素）。

```
1 int find(int x) {
2     if (x == fa[x]) // 如果x是根节点
3         return x;
4     return fa[x] = find(fa[x]); // 递归+路径压缩
5 }
```

3. Union操作（合并集合）

合并两个元素所在的集合。

```
1 void unionSet(int x, int y) {
2     x = findroot(x);
3     y = findroot(y);
4     fa[x] = y;
5 }
```

二、优化方法

1. 按秩合并（Union by Rank）

在 `Union` 时让小树合并到大树中，降低树的高度。

```

1 int rank[N]; // 记录树的深度
2 void init() {
3     for (int i = 0; i < N; i++) {
4         parent[i] = i;
5         rank[i] = 1; // 初始高度为1
6     }
7 }
8
9 void unionSet(int x, int y) {
10    int rootX = find(x), rootY = find(y);
11    if (rootX != rootY) {
12        if (rank[rootX] > rank[rootY]) {
13            parent[rootY] = rootX;
14        } else {
15            parent[rootX] = rootY;
16            if (rank[rootX] == rank[rootY]) rank[rootY]++;
17        }
18    }
19 }

```

三、时间复杂度

操作	平均时间复杂度	最坏时间复杂度
路径压缩	$O(\alpha(N))$	$O(\log N)$
路径压缩+按秩合并	$O(\alpha(N))$	$O(\alpha(N))$

其中 $\alpha(N)$ 是阿克曼函数的反函数，增长极其缓慢，可视为常数。

四、应用示例

题目：修改数组

<https://www.luogu.com.cn/problem/P8686>

```
1 #include <iostream>
2 #include <unordered_map>
3 using namespace std;
4
5 // 这里parent的含义就是下一个可用位置
6 unordered_map<int, int> parent;
7
8 - int find(int x) {
9     if (!parent.count(x)) parent[x] = x; // 动态初始化
10    if (parent[x] != x) parent[x] = find(parent[x]);
11    return parent[x];
12 }
13
14 - int main() {
15     int n, a;
16     cin >> n;
17 -     while (n--) {
18         cin >> a;
19         int root = find(a); // 找到可用位置
20         cout << root << " ";
21         parent[root] = root + 1; // 指向下一个可用位置
22     }
23     return 0;
24 }
```

数据类型与变量

数据类型大小

- **int**: 4字节, 表示范围为 -2^{31} 到 $2^{31}-1$ 。
- **unsigned int**: 4字节, 表示范围为 0 到 $2^{32}-1$ 。
- **short**: 2字节, 表示范围为 -2^{15} 到 $2^{15}-1$ 。
- **unsigned short**: 2字节, 表示范围为 0 到 2^{16} 。
- **long**: 4字节 (在某些系统上为8字节), 通常范围为 -2^{31} 到 $2^{31}-1$ 。
- **unsigned long**: 4字节 (在某些系统上为8字节), 通常范围为 0 到 2^{32} 。
- **long long**: 8字节, 范围为 -2^{63} 到 $2^{63}-1$ 。
- **unsigned long long**: 8字节, 范围为 0 到 $2^{64}-1$ 。
- **float**: 4字节, 表示单精度浮点数。
- **double**: 8字节, 表示双精度浮点数。
- **long double**: 通常为8、10或16字节, 具体取决于编译器和平台。
- **char**: 1字节, 表示范围为 -128 到 127 或 0 到 255 (取决于是否为有符号char)。

要确定在你的系统上的具体大小, 可以使用 **sizeof** 运算符来检查:

```
1 cout << "Size of int: " << sizeof(int) << " bytes" << endl;
```

注: $2^{31} \sim 2^{e10}$, $2^{63} \sim 2^{e19}$

数据类型后缀

1. `f` 和 `F`

表示 `float` 类型：

- `1.23f` 或 `1.23F` 是 `float` 类型字面量。
- 默认情况下，浮点数字面量是 `double` 类型，加上 `f` 后缀表示单精度浮点数。

2. `l` 和 `L`

表示 `long double` 类型：

- `1.23l` 或 `1.23L` 是 `long double` 类型字面量。
- 如果没有后缀，浮点数字面量默认是 `double` 类型，而加上 `l` 或 `L` 后缀表示扩展精度的浮点数。

3. `u` 和 `U`

表示 `unsigned` 类型：

- `10u` 或 `10U` 是 `unsigned int` 类型的字面量。
- 用 `u` 后缀来指定一个无符号整数。

4. `l` 和 `L`

表示 `long` 类型：

- `100L` 或 `100l` 是 `long` 类型的字面量。
- `l` 后缀可以与 `u` 或 `U` 组合，表示 `unsigned long` 类型（`100UL`）。

5. `ul`、`UL`、`lu` 和 `LU`

表示 `unsigned long` 类型：

- `100ul` 或 `100UL` 是 `unsigned long` 类型字面量。

6. `ll` 和 `LL`

表示 `long long` 类型:

- `100LL` 或 `100ll` 是 `long long` 类型的字面量。
- 用于表示比 `long` 类型更大的整数。

7. `ull` 和 `ULL`

表示 `unsigned long long` 类型:

- `100ULL` 或 `100ull` 是 `unsigned long long` 类型字面量。

示例代码:

```
1 int a = 100;           // 默认为 int 类型
2 unsigned int b = 100u; // 使用 u 后缀表示 unsigned int
3 long c = 100L;         // 使用 L 后缀表示 long 类型
4 unsigned long d = 100UL; // 使用 UL 后缀表示 unsigned long 类型
5 long long e = 100LL;   // 使用 LL 后缀表示 long long 类型
6 unsigned long long f = 100ULL; // 使用 ULL 后缀表示 unsigned long long 类型
7 float g = 1.23f;       // 使用 f 后缀表示 float 类型
8 double h = 1.23;       // 默认为 double 类型
9 long double i = 1.23L; // 使用 L 后缀表示 long double 类型
10
11 cout << "int: " << a << endl;
12 cout << "unsigned int: " << b << endl;
13 cout << "long: " << c << endl;
14 cout << "unsigned long: " << d << endl;
15 cout << "long long: " << e << endl;
16 cout << "unsigned long long: " << f << endl;
17 cout << "float: " << g << endl;
18 cout << "double: " << h << endl;
19 cout << "long double: " << i << endl;
```

8. `z` (用于指针类型)

在一些平台上, `z` 后缀被用于表示特定指针类型的字面量, 通常用于一些底层库中。

数据类型转换

1. 隐式转换

隐式转换是指编译器自动完成的类型转换，不需要显式指定。常见的隐式转换包括：

- 算术类型提升：如 `char`、`short` 等类型在算术运算中会自动提升为 `int` 类型。
- 整型到浮点型转换：当整数和浮点数一起运算时，整数会自动转换为浮点型。例如：`int` 转 `float`。如 `a=b*1.0`。
- 小范围到大范围类型转换：如 `int` 到 `double` 的转换，因为不会丢失数据。

示例：

```
1 int a = 10;
2 double b = a; // a 自动转换为 double 类型, b = 10.0
```

虽然隐式转换在大多数情况下很方便，但有时会产生精度丢失或编译警告。因此，复杂的转换通常通过显式转换来完成。

2. 显式转换

显式转换需要明确指定类型转换。C++ 中有四种类型的显式转换方式，分别是

`static_cast`、`dynamic_cast`、`const_cast` 和 `reinterpret_cast`，其中 `static_cast` 是最常用的。

2.1 `static_cast`

`static_cast` 用于大多数基本类型之间的转换，也是最安全的显式转换方式，主要用于：

- 基本数据类型之间的转换，例如 `int` 到 `float`、`double` 到 `int`。
- 指针的上行和下行转换，但通常只用于具有继承关系的类。

示例：

```
1 int a = 10;
2 double b = 2/static_cast<double>(a); // a 转换为 double 类型, b = 10.0
3 double c = 9.8;
4 int d = static_cast<int>(c); // c 转换为 int 类型, d = 9
```

2.2 `dynamic_cast`

`dynamic_cast` 主要用于具有继承关系的指针或引用之间的转换，并且通常用于将父类指针安全地转换为子类指针。`dynamic_cast` 需要类具有虚函数，以支持运行时类型检查（RTTI）。如果转换失败，会返回 `nullptr`（指针转换）或抛出异常（引用转换）。

示例：

```
1 class Base {
2     virtual void func() {} // 虚函数, 支持 RTTI
3 };
4
5 class Derived : public Base {};
6
7 Base* basePtr = new Derived();
8 Derived* derivedPtr = dynamic_cast<Derived*>(basePtr); // 成功转换
```

2.3 `const_cast`

`const_cast` 用于移除 `const` 或 `volatile` 属性，主要应用于需要对 `const` 对象进行修改的情况，但必须保证移除后不会违反常量性。

示例：

```
1 const int a = 10;
2 int* b = const_cast<int*>(&a); // 移除 const 属性
3 *b = 20; // 可能会导致未定义行为
```

2.4 `reinterpret_cast`

`reinterpret_cast` 是一种强制类型转换，可以用于任意指针类型之间的转换或整数和指针之间的转换，但很危险。通常用于底层操作，比如处理底层字节、设备地址等，不建议在普通代码中使用。

示例：

```
1 int a = 10;
2 int* p = &a;
3 char* pChar = reinterpret_cast<char*>(p); // 将 int* 转换为 char*
```

字符 (char) 与 字符串 (string)

字符与字符串之间的转换

- string --> 字符数组：

```
1 string str = "Hello";
2 const char* cstr = str.c_str(); // 转换为字符数组，用法为c_str()
```

- 字符数组 --> string：

```
1 char cstr[] = "Hello";
2 string str(cstr); // 从字符数组构造字符串，直接str()
```

字符串操作函数：

1. substr

用于提取子字符串。

```
1 std::string str = "Hello, World!";
2 std::string sub = str.substr(0, 5);
3 // 从索引0开始, 长度为5
4 // sub = "Hello"
```

2. empty

检查字符串是否为空。

```
1 if (str.empty()) std::cout << "String is empty." << std::endl;
```

3. size

返回字符串中的字符数（与length等效）。

```
1 string str = "Hello";
2 size_t len = str.size(); // len = 5
```

4. erase

删除字符串中的字符或子字符串。

```
1 string str = "Hello, World!";
2 str.erase(5, 7); // 从索引5开始删除7个字符
3 // str = "Hello!"
```

5. `size()` 或 `length()`

`size()` 和 `length()` 作用是相同，返回字符串中的字符数。

```
1 string str = "Hello, World!";
2 cout << "Length: " << str.size(); // 或者 str.length()
```

6. `char *strncpy(char *s1, const char *s2, size_t n);`

功能：将最多 `n` 个字符从 `s2` 复制到 `s1`。（如果省去`n`，则默认全部复制，这时候函数名为 `strcpy`）

用法:

- `strncpy` 会复制 `s2` 的最多 `n` 个字符到 `s1`，如果 `s2` 的长度小于 `n`，则会用 `\0` 填充 `s1`。
- 如果 `s2` 长度大于 `n`，则只会复制前 `n` 个字符。

注意事项:

- 如果 `s2` 的长度小于 `n`，需要注意 `s1` 会被填充 `\0`，因此有可能在目标字符串中留下多余的 `\0`。
- 如果 `s2` 长度超过 `n`，目标字符串不会被正确终止。

例子:

```
1 char s1[20];
2 char s2[] = "Hello, World!";
3 strncpy(s1, s2, 5); // 只复制前5个字符 "Hello"
```

7. `char *strncat(char *s1, const char *s2, size_t n);`

功能: 将最多 `n` 个字符从 `s2` 追加到 `s1` 的末尾。 (如果省去`n`, 则默认全部追加, 这时候函数名为`strcat`)

用法:

- `strncat` 会将 `s2` 中最多 `n` 个字符追加到 `s1`，并在末尾加上 `\0`。
- `s1` 必须有足够的空间来容纳追加的内容。

注意事项:

- 和 `strcat` 一样, 如果 `s1` 没有足够空间, 会导致缓冲区溢出。
- 不会追加超过 `n` 个字符。

例子:

```
1 char s1[20] = "Hello, ";
2 char s2[] = "World!";
3 strncat(s1, s2, 3); // s1 变成 "Hello, Wor"
```

8. `int strncmp(const char *s1, const char *s2, size_t n);`

功能: 比较 `s1` 和 `s2` 的前 `n` 个字符。“默认”`s1>s2`, 对了就是1, 错了就是-1, 一样就是0。
(如果省去`n`, 则默认逐个比较直至结束, 这时候函数名为`strcmp`)

注意事项:

- 如果比较到 `n` 个字符或遇到 `'\0'`，就停止比较。
- 需要确保不会超出 `s1` 和 `s2` 的长度。
- `'\0'` 可视为最小的。

例子：

```
1 char s1[] = "Hello";
2 char s2[] = "Hell";
3 int result = strncmp(s1, s2, 4); // result == 0, 前四个字符相同
```

9. `size_t strlen(const char *s);`

功能：计算字符串 `s` 的长度，不包括字符串的终止符 `\0`。

返回值：

- 返回 `s` 中字符的数量。

注意事项：

- `strlen` 遇到字符串结束标志 `\0` 时停止，计算的是字符数，不包括 `\0`。
- 如果传入的字符串没有正确终止，会导致未定义行为。

与 `sizeof` 的区别：

- `sizeof` 返回变量或类型的大小（以字节为单位），对于字符串常量，它返回的是字符串所占的内存大小，包括字符串结束符 `\0`。
- `strlen` 返回字符串的长度（不包括 `\0`），它只计算字符串中实际的字符数量。

例子：

```
1 char s[] = "Hello";
2 printf("%lu\n", sizeof(s)); // 6 (包括 '\0')
3 printf("%lu\n", strlen(s)); // 5 (不包括 '\0')
```

10. `char *strtok(char *s1, const char *s2);`

功能：将字符串 `s1` 分割成一系列的子字符串，分隔符为 `s2`。

用法：

- 第一次调用时，传入待分割的字符串 `s1`，后续调用时，传入 `NULL` 来继续分割上次剩余的字符串。
- 每次调用会返回指向下一个子字符串的指针。

注意事项:

- `strtok` 会修改原始字符串，将分隔符替换为 `\0`。
- 不适合多线程程序，因为它会使用静态内部状态。
- 假如vs表示不安全，可以添加 `#define _CRT_SECURE_NO_WARNINGS` 在开头（宏不用;）

例子：

```
1 char s[] = "Hello, World!";
2 char *token = strtok(s, ", ");
3 token = strtok(NULL, ", ");
```

输入

1.cin

```
1 char a[50];
2 cin >> a;
3 cout << a;
4 //直接输入字符串，遇到空格停止
5 //输入：12 23
6 //输出：12
```

2.cin.get()

```
1 char ch;
2 cin.get(ch); // 读取一个字符，包括空格、换行符等所有字符
3 cout << "You entered: " << ch << endl;
4 //若 int ch;
5 //输入: 1
6 //输出: 49
7
8 char str[100];
9 cout << "Enter a line of text: ";
10 cin.get(str, 100); // 读取一行最多100个字符
11 cout << "You entered: " << str << endl;
12 //到换行符为止，数量不够也就算了
13 //如果输入超过了数量，只会读取前 n-1 个字符，并在结尾加上'\0'表示字符串的结束
```

3. `cin.getline()`

`cin.getline()` 用于读取一行字符，包括空格，但不会读取换行符。它会将换行符替换为字符串结束符 '`\0`'。`cin.getline()` 可以防止缓冲区溢出，通过指定最大字符数来限制读取的长度。

```
1 char str[100];
2
3 cout << "Enter a line of text: ";
4 cin.getline(str, 100); // 读取一行文本（最多n-1个字符）
5
6 cout << "You entered: " << str << endl;
```

4. `getline()`

`getline(cin,string)` 用于从输入流中读取一整行字符串，包括空格，直到遇到换行符。

```
1 string input;
2 cout << "Enter a line of text: ";
3 getline(cin, input); // 读取整行输入，注意前面"cin,"
4 cout << "You entered: " << input << endl;
5 return 0;
```

5. `cin.ignore()`

```
1 cin.ignore(int n = 1, int delim = '\n')
```

- `n` (可选，默认为 1)：表示要忽略的字符数。

- **delim** (可选, 默认为 `'\n'`) : 指定忽略直到遇到的字符 (包括该字符) 。

便捷写法:

```
1 cin.ignore(10,)
```

作用举例:

```
1 int num;
2 char ch;
3
4 cout << "Enter an integer: ";
5 cin >> num;
6
7 cin.ignore(); // 默认忽略输入中的一个字符
8 //假如没有ignore(), 输入num时携带'\n'残留, 在下一次cin.get(ch)时进入ch
9
10 cout << "Enter a character: ";
11 cin.get(ch); // 读取一个字符, 包括换行符
12
13 cout << "You entered: " << num << " and " << ch << endl;
```

枚举类型 (**enum**)

1. 基本的枚举类型

一个最简单的枚举类型定义如下:

```
1 enum Color {Red,Green,Blue};
2 Color color = Green; //通过枚举类型来声明变量
```

这段代码定义了一个名为 `Color` 的枚举类型，并为它定义了三个枚举常量：`Red`、`Green` 和 `Blue`。这些常量会自动赋予一个整数值，从 0 开始（依次递增）。即：

- `Red = 0`
- `Green = 1`
- `Blue = 2`

2. 指定枚举常量的值

```
1 enum Color {Red = 10, Green = 20, Blue = 30}; // 显式地指定枚举常量的值
```

3. 枚举底层类型

默认情况下，枚举常量的底层类型是 `int`。但可以指定枚举的底层类型。

```
1 enum Color : unsigned char {Red, Green, Blue};
```

在这个例子中，`Color` 枚举的底层类型是 `unsigned char`，而不是默认的 `int`。

4. 枚举和数组的结合

可以用枚举作为数组的索引：

```
1 enum Color {Red, Green, Blue, ColorCount};  
2  
3 int colorValues[ColorCount] = {255, 0, 0}; // 数组大小等于枚举常量数量
```

举例：

```

1 #include<iostream>
2 using namespace std;
3 int main() {
4     enum Test { AA = 2, BB, CC = 3, DD = 1, EE };
5     Test haha=BB;
6     //这里BB顺着AA递增，即为3；同理，EE为2；
7
8     if(haha==3) cout<<"win!"<<endl; //3换成BB结果不变
9     else cout<<"lose!"<<endl;
10    //输出：win!
11
12    int my=AA;
13    cout<<my<<endl;
14    //输出：2
15
16    return 0;
17 }

```

5. enum class

C++11 引入的一个强类型枚举，相对于传统的 `enum` 提供了更强的类型安全和作用域控制 `enum class` 可以避免名称冲突，并且不自动将枚举值转换为整数。

特性

1. 作用域控制：

- 在 `enum class` 中，枚举值不会自动放入外部作用域，而是需要通过 `EnumName::Value1` 这种方式来访问。
- 这样可以避免与其他变量或枚举值产生命名冲突。

2. 强类型：

- `enum class` 的值不能隐式转换为整数类型。也就是说，`EnumName::Value1` 不能直接赋值给一个整数，除非你显式进行类型转换。
- 这增强了类型安全，避免了可能的错误。

3. 显式指定底层类型：

- 可以通过 `: underlying_type` 来指定 `enum class` 的底层类型（默认是 `int`）。

示例代码

```
1 #include <iostream>
2
3 enum class Color {Red,Green,Blue};
4
5 int main() {
6     Color color = Color::Red;
7
8     // 编译错误: 不能将 Color 直接赋给 int
9     // int color_value = color;
10
11    // 显式转换为整数
12    int color_value = static_cast<int>(color);
13
14    std::cout << "Color as integer: " << color_value << std::endl;
15    return 0;
16 }
```

enum class 的优势

- **避免名称冲突:** 传统的 `enum` 会将所有枚举值放在全局作用域，而 `enum class` 会将它们封装在指定的枚举类型作用域中。
- **类型安全:** 由于 `enum class` 是强类型的，它不能隐式转换为其他类型，因此更加安全。
- **可选的底层类型:** 可以自定义底层类型来节省内存，尤其在处理较大枚举时。

引用 (Reference)

引用是 C++ 中的一种数据类型，它是某个变量的别名。引用变量与原始变量共享相同的内存地址，因此修改引用变量的值，也会修改原始变量的值。注：引用变量不能作为数组大小。

1. 引用的基本语法

声明一个引用变量时，在类型和变量名之间使用 `&` 符号。例如：

```
1 int a = 10;
2 int& ref = a; // ref 是 a 的引用
```

2. 引用的特点：

- 必须初始化：引用变量必须在声明时被初始化。
- 不可更改指向：引用一旦绑定，就不能指向其他变量。引用永远指向它最初绑定的变量。

```
1 int x = 5;
2 int y = 10;
3 int& ref = x; // ref现在是x的别名，对ref的修改将直接影响x
4 ref = y;
5 // 修改ref，由于ref就是x的别名，相当于x也发生了变化
6 // 即此时 x=10, y=10, ref=10
```

3. 引用作为函数参数

引用允许函数直接修改调用者的参数，而无需复制数据。

```
1 void increment(int& n) {
2     n++; // 修改传入的变量
3 }
4
5 int main() {
6     int a = 5;
7     increment(a); // 传入 a 的引用
8     cout << a; // 输出 6
9 }
```

4. 引用返回值

引用也可以作为函数的返回值，允许修改调用者传入的变量。

```
1 int& getElement(int arr[], int index) { // 关键是函数名前面有 &
2     return arr[index]; // 返回数组元素的引用
3 }
4
5 int main() {
6     int arr[] = {1, 2, 3};
7     getElement(arr, 1) = 10; // 修改数组元素
8     cout << arr[1]; // 输出 10
9 }
```

- `getElement` 返回数组元素的引用，可以直接修改数组中的数据。

符号常量 (`constant`)

符号产量是指在程序执行过程中其值不会改变的量。

1. `#define` 宏常量

`#define` 用于定义符号常量，它在预处理阶段进行替换。宏常量通常没有类型，且没有作用域限制。

```
1 #define PI 3.14159
```

在代码中，`PI` 会被替换为 `3.14159`，这在编译阶段完成。注意，`#define` 是一个预处理指令，不会进行类型检查，因此不能直接用在复杂的计算中，容易出现错误。

2. `const` 常量

`const` 关键字可以用来定义类型安全的常量。通过 `const` 声明的常量具有指定类型，并且只能赋值一次。

```
1 const double PI = 3.14159;
```

这种方式与 `#define` 不同，`const` 常量有明确的类型，并且可以在代码中像变量一样使用。`const` 常量的作用域是局部或全局，取决于它的位置。

自动存储类别和静态存储类别

1. 自动存储类别 (`auto`)

- **生命周期：**变量的生命周期从其定义的作用域开始，到作用域结束时自动销毁。
- **存储位置：**通常存储在栈区。
- **初始化：**每次进入作用域时，变量会被重新初始化。
- **特点：**大多数局部变量默认具有自动存储类别，内存管理由编译器自动处理。

```
1 void func() {  
2     int x = 10; // 自动存储类别  
3 } // x 在函数结束时销毁
```

2. 静态存储类别 (`static`)

- **生命周期：**变量的生命周期从程序启动到程序结束。即使在作用域之外，也会一直存在。
- **存储位置：**通常存储在静态存储区。

- **初始化**: 变量只会在第一次调用时初始化一次，之后保持其值。数值变量默认初始化为0。
- **特点**: 通过 `static` 关键字声明的变量或全局变量具有静态存储类别，生命周期贯穿程序的整个运行过程。

示例：

```
1 void func() {  
2     static int x = 10; // 静态存储类别  
3     x++;  
4     std::cout << x << std::endl; // 保持上次值  
5 }
```

左值 (Lvalue) 和右值 (Rvalue)

值类别定义了表达式的结果是可以被修改的还是临时的。左值和右值描述了对象在内存中的存在方式和生命周期，并决定了它们如何与引用、赋值等操作交互。

左值 (Lvalue) :

表示一个持久的对象，它有一个明确的内存地址。可以出现在赋值语句的左边，也就是可以被修改的对象。

右值 (Rvalue) :

表示一个临时的、没有名称的对象，它没有明确的内存地址，通常用于计算表达式的结果、返回临时对象等场景。右值不能出现在赋值语句的左边。

位运算与表达式

位运算

1. 位与运算符 &

功能：对两个二进制数的每一位进行与运算，只有当两位都是1时结果才为1。

示例：

```
1 int a = 5; // 二进制 0101
2 int b = 3; // 二进制 0011
3 int result = a & b; // 结果为 1, 二进制 0001
```

2. 位或运算符 |

功能：对两个二进制数的每一位进行或运算，只要有一位为1，结果就为1。

示例：

```
1 int a = 5; // 二进制 0101
2 int b = 3; // 二进制 0011
3 int result = a | b; // 结果为 7, 二进制 0111
```

3. 位异或运算符 ^

功能：对两个二进制数的每一位进行异或运算，当两位不同（一个为1一个为0）时结果为1。

示例：

```
1 int a = 5; // 二进制 0101
2 int b = 3; // 二进制 0011
3 int result = a ^ b; // 结果为 6, 二进制 0110
```

4. 取反运算符 ~

功能：对一个二进制数的每一位进行取反运算，0变为1，1变为0。

示例：

```
1 int a = 5; // 二进制 0101
2 int result = ~a; // 结果为 -6, 二进制 1010 (补码表示)
```

5. 左移运算符 <<

功能：将一个数的二进制位向左移动指定的位数，左移一位相当于乘以2。

示例：

```
1 int a = 5; // 二进制 0101
2 int result = a << 1; // 结果为 10, 二进制 1010
```

6. 右移运算符 >>

功能：将一个数的二进制位向右移动指定的位数，右移一位相当于整除2。

示例：

```
1 int a = 5; // 二进制 0101
2 int result = a >> 1; // 结果为 2, 二进制 0010
```

Lambda 表达式

Lambda 表达式是一种简洁的方式来定义匿名函数对象（函数指针）。Lambda 表达式允许你在代码中内联定义和使用函数，而无需显式定义一个命名函数。这使得代码更加简洁、灵活，尤其适用于那些需要将短小的函数传递给其他函数的场景，即一种简化函数。

Lambda 表达式的基本语法

```
1 [捕获列表] (参数列表) -> 返回类型 {函数体}
```

- **捕获列表 (Capture List)**：指定 Lambda 可以使用外部作用域中的哪些变量。可以按值捕获、按引用捕获，或者不捕获任何外部变量。
- **参数列表 (Parameter List)**：指定 Lambda 接受的参数（和普通函数一样）。如果没有参数，可以省略。例如：(node const& a, node const& b)
- **返回类型 (Return Type)**：指定 Lambda 返回值的类型。如果 Lambda 不返回任何值，可以省略。
- **函数体 (Body)**：Lambda 的实际代码，表示要执行的操作。

1. 基本示例

```

1 int main() {
2     // 定义一个没有参数的 Lambda, 返回常数 10
3     auto lambda = []() { return 10; };
4
5     cout << "Lambda result: " << lambda() << endl;
6     //输出: Lambda result: 10
7     return 0;
8 }
```

2. 带参数的 Lambda

Lambda 表达式也可以接受参数，就像普通函数一样：

```

1 // 定义一个带参数的 Lambda
2 auto sum = [](int a, int b) { return a + b; };
3
4 cout << "Sum: " << sum(5, 10) << endl;
5 //输出: Sum: 15
```

3. 捕获外部变量

3.1 按值捕获

```
1 int x = 10;
2 int y = 20;
3
4 // 按值捕获变量 x 和 y
5 auto sum = [x, y]() { return x + y; };
6
7 cout << "Captured sum: " << sum() << endl; // x 和 y 按值捕获
8 //输出: Captured sum: 30
9
10 // 修改原始变量 x 和 y
11 x = 30;
12 y = 40;
13
14 // 再次调用 Lambda
15 cout << "Captured sum after modification: " << sum() << endl;
16 // 依然输出: Captured sum after modification:30
```

在这个例子中，`x` 和 `y` 被按值捕获，意味着它们的拷贝被传递给 Lambda，即使 `x` 或 `y` 后续发生变化，Lambda 中的值也不会改变。

3.2 按引用捕获

```
1 int x = 10;
2 int y = 20;
3
4 // 按引用捕获变量 x 和 y
5 auto sum = [&x, &y]() { return x + y; };
6 //即 & 的区别
7
8 // 修改外部变量
9 x = 30;
10 y = 40;
11
12 // x 和 y 按引用捕获
13 cout << "Captured sum after modification: " << sum() << endl;
14 //输出: Captured sum after modification: 70
```

在这个例子中，`x` 和 `y` 是按引用捕获的，因此 Lambda 表达式会直接操作外部变量的原始值，而不是它们的副本。

3.3 捕获所有变量

- 按值捕获所有变量：`[=]`

- 按引用捕获所有变量: [&]

- 这样是会捕获无关变量的。

```
1 int x = 10, y = 20;
2
3 // 按值捕获所有变量
4 auto sum_by_value = [=]() { return x + y; };
5
6 // 按引用捕获所有变量
7 auto sum_by_reference = [&]() { return x + y; };
8
9 x = 50;
10 y = 60;
11
12 // 捕获时按值
13 cout << "Sum by value: " << sum_by_value() << endl;
14 //输出: Sum by value: 30
15
16 // 捕获时按引用
17 cout << "Sum by reference: " << sum_by_reference() << endl;
18 //输出: Sum by reference: 110
```

4. Lambda 的返回类型

C++11 中，如果 Lambda 的返回类型可以推导，则可以省略返回类型，C++ 会自动推导。你也可以显式指定返回类型。

4.1 自动推导返回类型

```
1 // 返回类型自动推导为 int
2 auto multiply = [](int a, int b) { return a * b; };
3
4 cout << "Product: " << multiply(5, 6) << endl;
```

4.2 显式指定返回类型

```
1 // 显式指定返回类型为 long
2 auto multiply = [](int a, int b) -> long { return a * b; };
3
4 cout << "Product: " << multiply(5, 6) << endl;
```

5. Lambda 表达式的实际应用

Lambda 表达式常用于以下几种场景：

1. **作为回调函数**: 很多 STL 算法 (如 `std::sort`, `std::for_each`) 都可以接受 Lambda 表达式作为回调函数。 (可参见章节 sort)
2. **短小的函数**: 当你需要一个简单的函数, 不想为其单独定义一个命名函数时, 可以使用 Lambda 表达式。
3. **多线程编程**: 在并发编程中, Lambda 表达式可以非常方便地作为线程的任务传递给 `std::thread` 。

ASCII码表

分区记忆

A. 控制字符 (0–31)

- 0: `NUL` (空字符)
- 9: `TAB` (制表符)

- 10: LF (换行)
- 13: CR (回车)
- 27: ESC (转义符)
- 32: SPACE (空格)

B. 数字字符 (48–57)

C. 大写字母 (65–90)

D. 小写字母 (97–122)

E. 标点符号和其他可打印字符 (32–47, 58–64, 91–96, 123–126)

- 标点符号:
 - '!' = 33, '"' = 34, '#' = 35, ...
 - '(' = 40, ')' = 41, '*' = 42, '+' = 43, ...
 - ';' = 59, ':' = 58, '<' = 60, '>' = 62, '?' = 63, ...

关键值和偏移量

- '0' = 48, '9' = 57, 'A' = 65, 'a' = 97
- 大写字母和小写字母之间的 ASCII 值有固定差距：大写字母的 ASCII 值比小写字母小 32。

高四位		ASCII表																							
		ASCII控制字符						ASCII打印字符																	
		0000			0001			0010		0011		0100		0101		0110		0111							
		0		1		2		3		4		5		6		7									
低四位	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	Ctrl	代码	转义字符	字符解释	十进制	字符	十进制	字符	十进制	字符	十进制	字符	Ctrl				
0000	0	0		^@	NUL	\0	空字符	16	▶	^P	DLE		数据链路转义	32		48	0	64	@	80	P	96	`	112	p
0001	1	1	☺	^A	SOH		标题开始	17	◀	^Q	DC1		设备控制 1	33	!	49	1	65	A	81	Q	97	a	113	q
0010	2	2	☻	^B	STX		正文开始	18	↑	^R	DC2		设备控制 2	34	"	50	2	66	B	82	R	98	b	114	r
0011	3	3	♥	^C	ETX		正文结束	19	!!	^S	DC3		设备控制 3	35	#	51	3	67	C	83	S	99	c	115	s
0100	4	4	◆	^D	EOT		传输结束	20	¶	^T	DC4		设备控制 4	36	\$	52	4	68	D	84	T	100	d	116	t
0101	5	5	♣	^E	ENQ		查询	21	§	^U	NAK		否定应答	37	%	53	5	69	E	85	U	101	e	117	u
0110	6	6	♠	^F	ACK		肯定应答	22	—	^V	SYN		同步空闲	38	&	54	6	70	F	86	V	102	f	118	v
0111	7	7	•	^G	BEL	\a	响铃	23	↑	^W	ETB		传输块结束	39	'	55	7	71	G	87	W	103	g	119	w
1000	8	8	█	^H	BS	\b	退格	24	↑	^X	CAN		取消	40	(56	8	72	H	88	X	104	h	120	x
1001	9	9	○	^I	HT	\t	横向制表	25	↓	^Y	EM		介质结束	41)	57	9	73	I	89	Y	105	i	121	y
1010	A	10	◎	^J	LF	\n	换行	26	→	^Z	SUB		替代	42	*	58	:	74	J	90	Z	106	j	122	z
1011	B	11	♂	^K	VT	\v	纵向制表	27	←	^[\	ESC	\e	溢出	43	+	59	;	75	K	91	[107	k	123	{
1100	C	12	♀	^L	FF	\f	换页	28	_	^`	FS		文件分隔符	44	,	60	<	76	L	92	\	108	l	124	
1101	D	13	♪	^M	CR	\r	回车	29	↔	^]	GS		组分隔符	45	-	61	=	77	M	93]	109	m	125	}
1110	E	14	🎵	^N	SO		移出	30	▲	^^	RS		记录分隔符	46	.	62	>	78	N	94	^	110	n	126	~
1111	F	15	⌚	^O	SI		移入	31	▼	^-	US		单元分隔符	47	/	63	?	79	O	95	_	111	o	127	^K

注：表中的ASCII字符可以用“Alt + 小键盘上的数字键”方法输入。

2013/08/08

^Backspace
代码：DEL

逻辑运算符的短路行为

1. 逻辑与 (`&&`) :

- 如果第一个操作数为 `false`，则整个表达式的结果为 `false`，因此第二个操作数不会被计算（因为结果已经确定）。

2. 逻辑或 (`||`) :

- 如果第一个操作数为 `true`，则整个表达式的结果为 `true`，因此第二个操作数不会被计算（因为结果已经确定）。

3. 逻辑非 (`!`) :

- `!` 运算符没有短路行为，因为它只作用于单个操作数。

应用示例：

```
1 int a = 1, b = 1;
2 if (a == 2 && ++b == 2) cout << "true" ; //a==2返回false, 判断结束, 轮不到++
b
3 else cout << "false" ;
4 cout << a << " " << b << endl;
5 //输出 false 1 1
6
7 a = 1, b = 1;
8 if (a == 1 || ++b == 10) cout << "true" ; //a==1返回true, 判断结束, 轮不到++
b
9 else cout << "false" ;
10 cout << a << " " << b << endl;
11 //输出 true 1 1
12
13 a = 1;
14 if (a++ == 2) cout << "true" ; //这里a先判断 ==2, 再++
15 else cout << "false" ;
16 cout << a << endl;
17 //输出 false 2
18
19 a = 1;
20 if (++a == 2) cout << "true" ; //这里a先++, 再判断 ==2
21 else cout << "false" ;
22 cout << a << endl;
23 //输出 true 2
```

指针与内存管理

指针

1. 指针的本质

指针存储的是另一个变量的内存地址，通过指针可以访问和操作该位置的数据。

- **指针类型：**指针有类型，指针的类型决定了它指向的变量的类型。例如，`int*` 类型的指针指向的是一个整数类型的变量，`char*` 类型的指针指向的是一个字符类型的变量。

2. 指针的声明和初始化

指针的声明方式为： 数据类型`*` 指针名称；```*` 表示这是一个指针变量。

```
1 int* ptr; // 指向整数类型的指针
```

在声明指针时，通常需要将它初始化为一个有效的内存地址。如果没有初始化，指针会指向一个不确定的地址，可能导致程序出错。

```
1 int x = 10;
2 int* ptr = &x; // &x表示取变量x的内存地址
```

此时，`ptr` 就是一个指向 `x` 的指针，`*ptr` 则可以访问到 `x` 的值。

3. 指针的操作

- 取地址操作符（&）：获取变量的内存地址。

```
1 int x = 10;
2 int* ptr = &x; // ptr保存的是x的地址
```

- 解引用操作符（*）：通过指针访问指针所指向的内存位置的值。

```
1 int x = 10;
2 int* ptr = &x;
3 cout << *ptr; // 输出10, *ptr是解引用操作，得到ptr指向的值
```

4. 指针的用途

- 动态内存分配：在C++中，通过指针可以动态地申请和释放内存。常用的操作包括 `new` 和 `delete`。

```
1 int* ptr = new int; // 动态分配内存
2 *ptr = 20;           // 给指针指向的内存赋值
3 cout << *ptr;       // 输出20
4 delete ptr;         // 释放动态分配的内存
```

- 函数参数传递：指针可以作为函数参数传递，允许函数修改调用者提供的变量的值。

```
1 void modify(int* ptr) {
2     *ptr = 20; // 修改指针指向的值
3 }
4
5 int x = 10;
6 modify(&x); // 传递x的地址
7 cout << x; // 输出20
```

- 实现复杂数据结构：指针是实现链表、树等复杂数据结构的基础。在这些数据结构中，节点之间通过指针连接。

5. 指针的常见陷阱

- 野指针：如果指针没有被初始化，或者指向一个已经释放的内存地址，它就会成为“野指针”，访问这种指针会导致未定义行为。

```
1 int* ptr; // ptr没有初始化，指向一个未知地址
2 cout << *ptr; // 这是危险的，可能导致崩溃
```

- 空指针（Null Pointer）：空指针是一个不指向任何有效内存地址的指针。在使用指针之前，最好检查它是否为空。

```
1 int* ptr = nullptr; // 空指针
2 if (ptr != nullptr) {
3     // 使用ptr之前先检查
4 }
```

- 内存泄漏：在使用动态内存分配时，如果忘记释放内存，可能会导致内存泄漏。要确保使用 `delete` 释放 `new` 分配的内存。

```
1 int* ptr = new int(10);
2 // 使用完ptr后要释放内存
3 delete ptr;
```

- 指针算术：指针不仅可以存储内存地址，还可以进行加减运算。指针算术允许你通过指针操作数组元素等。

```
1 int arr[] = {10, 20, 30};
2 int* ptr = arr; // ptr指向arr[0]
3 cout << *(ptr + 1); // 输出20, ptr + 1指向arr[1]
```

6. 指针与数组的关系

数组名本身就是一个常量指针，指向数组的第一个元素。因此，你可以用指针访问数组元素，或者通过数组名来实现指针的功能。

```
1 int arr[] = {10, 20, 30};
2 int* ptr = arr; // ptr指向arr[0]
3 cout << *ptr; // 输出10
4 cout << *(ptr + 1); // 输出20
```

7. 多级指针

指针还可以指向其他指针。多级指针的声明方式为 `数据类型** 指针名称`。

```
1 int x = 10;
2 int* ptr = &x;
3 int** ptr2 = &ptr;
4 int*** ptr3 = &ptr2;
5 cout << ***ptr3; // 输出10, ***ptr3解引用三次，获取ptr指向的值
```

8. &a++ 与 &++a

a++ 是后置递增，它先返回 a 当前的值（此时即右值），然后递增 a。因此，&a++ 会导致编译错误，因为 a++ 是一个表达式，而不能直接取它的地址。

++a 是前置递增，它先递增 a 的值，然后返回递增后的值（递增结束后，此时为左值），所以可以通过 & 运算符取 a 的地址。

通用指针

1. void *指针的定义

void* 是一种特殊的指针类型，可以指向任何类型的对象或数据，但是它不直接关联任何具体类型。因此，void* 被称为“通用指针”或“空指针”。

```
1 void* ptr; // 声明一个 void 指针
```

2. void* 的基本用法

- 指向任何类型的数据

void* 指针能够指向任意类型的数据，可以通过将任意类型的指针转换为 void*，或者将 void* 转换回具体类型的指针。

```
1 int a = 10;
2 float b = 3.14f;
3 char c = 'A';
4
5 void* ptr; // 声明一个通用指针
6
7 ptr = &a;    // ptr 指向 int 类型变量
8 ptr = &b;    // ptr 现在指向 float 类型变量
9 ptr = &c;    // ptr 现在指向 char 类型变量
```

3. 使用 void* 指针访问数据

`void*` 本身不能直接进行解引用，因为它没有类型信息。因此，在使用 `void*` 时，必须先将它转换为正确的类型指针（即通过类型转换）。

```
1 int a = 10;
2 void* ptr = &a;
3
4 // 将 void* 转换回 int* 来解引用
5 int* int_ptr = static_cast<int*>(ptr);
6 cout << *int_ptr << endl; // 输出 10
```

4. 与 `void*` 的类型转换

- 将其他指针转换为 `void*`：任何类型的指针都可以隐式转换为 `void*`，因为 `void*` 是一个通用指针类型。

```
1 int x = 10;
2 float y = 5.5f;
3
4 void* ptr1 = &x; // int* 隐式转换为 void*
5 void* ptr2 = &y; // float* 隐式转换为 void*
```

- 将 `void` 转换为具体类型的指针：从 `void*` 转换回具体类型时，需要显式转换（使用 `static_cast` 或 `reinterpret_cast`）。因为 `void*` 不包含类型信息，编译器不知道它实际指向什么类型的数据。

```
1 void* ptr = &x;
2 int* int_ptr = static_cast<int*>(ptr); // 将 void* 转换为 int*
```

常量指针 (constant pointer) 和 指针常量 (pointer to constant)

1. 常量指针 (Constant Pointer)

常量指针是指指针本身的值是常量，不能改变。也就是说，常量指针不能再指向其他位置，但它可以修改所指向地址的内容。

示例：

```
1 int a = 10;
2 int b = 20;
3 int* const ptr = &a; // ptr 是一个常量指针，指向 int 类型
4
5 *ptr = 30; // 可以修改 ptr 指向的值
6 // ptr = &b; // 错误：常量指针 ptr 不能修改指向的地址
```

2. 指针常量 (Pointer to Constant)

指针常量是指指针指向的数据是常量，不能修改，但指针本身可以改变，指向其他的内存地址。

示例：

```
1 int a = 10;
2 int b = 20;
3 const int* ptr = &a; // ptr 是一个指向常量 int 的指针
4
5 // *ptr = 30; // 错误：无法通过 ptr 修改所指向的数据
6 ptr = &b; // 可以修改 ptr 指向的地址
```

3. 同时使用常量指针和指针常量

可以同时使用常量指针和指针常量，这样既限制指针不能指向其他地址，也限制指针指向的数据不能修改。定义方式如下：

```
1 const int* const ptr = &a; // ptr 是一个常量指针，指向常量数据
```

• 和 ->

`.` 和 `->` 都用于访问对象的成员（属性或方法）。`.` 用于通过对象直接访问其成员；`->` 用于通过指针访问对象的成员。

1. `.` 的使用：

`dot` 操作符 (`.`) 直接应用于对象（不是指针）。

```
1 class MyClass {  
2     public:  
3         int x;  
4     };  
5  
6 int main() {  
7     MyClass obj;  
8     obj.x = 10;  
9  
10    // 使用 . 直接访问对象的成员  
11    cout << obj.x << endl; // 输出: 10  
12    return 0;  
13 }
```

2. `->` 的使用：

`arrow` 操作符 (`->`) 用于通过指针访问对象的成员。指针指向对象时，我们使用 `->` 来访问该对象的成员。

```
1 -> class MyClass {
2     public:
3         int x;
4 ->     void print() {
5             cout << "x = " << x << endl;
6     }
7 };
8
9 -> int main() {
10    MyClass obj;
11    obj.x = 10;
12
13    // 使用指针来访问对象
14    MyClass* ptr = &obj;
15
16    // 使用 -> 访问成员
17    cout << ptr->x << endl;    // 输出: 10
18    ptr->print();    // 调用成员函数 print()
19    return 0;
20 }
```

函数与作用域

默认参数 (Default Arguments)

默认参数是指在函数定义时为参数指定一个默认值，当调用该函数时，如果没有提供相应的参数值，则使用这个默认值。

```
1 // 函数定义，参数a和b有默认值
2 // 默认实参设置之后，其右侧的变量必须也要设置。
3 // 即(int a=1,int b) 是不正确的
4 // 注意点二：不要重复设置默认参数
5 - void printNumbers(int a = 1, int b = 2) {
6     cout << "a: " << a << ", b: " << b << endl;
7 }
8
9 - int main() {
10    printNumbers();           // 使用默认值，输出：a: 1, b: 2
11    printNumbers(10);         // 使用a=10, b使用默认值，输出：a: 10, b: 2
12    printNumbers(10, 20);      // 使用a=10, b=20, 输出：a: 10, b: 20
13    return 0;
14 }
```

作用域解析运算符 ::

:: 用于指定某个标识符（如变量、函数、类等）属于哪个作用域或命名空间。

1. 访问全局变量或函数

当局部作用域和全局作用域中有同名的变量或函数时，可以使用 :: 来访问全局作用域中的变量或函数。

例子：

```
1 int x = 10; // 全局变量
2
3 - int main() {
4     int x = 5; // 局部变量
5     cout << "Local x: " << x << endl;
6     cout << "Global x: " << ::x << endl; // 使用 :: 访问全局变量 x
7     //输出： Local x: 5
8     //          Global x: 10
9     return 0;
10 }
```

2. 定义类外的类成员函数

在类的外部定义成员函数时，需要使用 `::` 来指定这个函数是哪个类的成员函数。

例子：

```
1 #include <iostream>
2 using namespace std;
3
4 - class MyClass {
5     public:
6         void printMessage();
7 };
8
9 - void MyClass::printMessage() {
10     cout << "Hello from MyClass!" << endl;
11 }
12
13 - int main() {
14     MyClass obj;
15     obj.printMessage(); // 调用成员函数
16     return 0;
17 }
```

3. 访问命名空间中的成员

`::` 也用于访问命名空间中的元素。当函数、变量或类定义在特定的命名空间中时，使用 `::` 来访问它们。

例子：

```
1 -> namespace MyNamespace {
2     int x = 100;
3 }
4
5 -> int main() {
6     cout << "Value of x in MyNamespace: " << MyNamespace::x << endl;
7     return 0;
8 }
```

4. 访问类的静态成员

对于类的静态成员（例如静态变量或静态函数），可以使用 `::` 来访问。

例子：

```
1 -> class MyClass {
2     public:
3         static int count;           // 静态成员变量
4 ->     static void showCount() { // 静态成员函数
5         cout << "Count: " << count << endl;
6     }
7 };
8
9     int MyClass::count = 0;    // 静态成员变量的初始化
10
11 -> int main() {
12     MyClass::count = 10;      // 访问静态成员变量
13     MyClass::showCount();   // 访问静态成员函数
14     return 0;
15 }
```

5. 类的继承中的作用域

如果派生类重写了基类的函数或成员，但你希望在派生类中调用基类的版本，可以使用 `:::` 来指定基类的成员。

例子：

```
1 - class Base {
2   public:
3 -     void show() {
4       cout << "Base class show()" << endl;
5     }
6   };
7
8 - class Derived : public Base {
9   public:
10 -    void show() {
11      cout << "Derived class show()" << endl;
12      Base::show(); // 使用 Base:: 调用基类的 show()
13    }
14  };
15
16 - int main() {
17   Derived obj;
18   obj.show();
19   return 0;
20 }
21
22 //输出: Derived class show()
23 //       Base class show()
```

内联函数 (inline function)

内联函数是一种通过在编译时将函数的代码直接插入调用处来提高程序执行效率的机制。内联函数的调用不经过常规的函数调用过程，而是直接替换为函数体的代码，这样可以避免函数调用的开销。

销，如栈操作和跳转。

内联函数的优点

1. 提高性能
2. 减少栈空间的使用
3. 代码简洁

内联函数的限制

1. 不能内联太复杂的函数
2. 增加代码体积
3. 编译器的优化：最终是否内联函数还是由编译器决定。即使你标记了 `inline`，编译器可能根据优化策略选择不内联。

使用场景

- 简单的访问器和设置器 (`getters and setters`)：比如返回类成员值的函数。
- 数学运算或小函数：对于计算密集型的小函数，如加法、乘法等。

示例

```
1 - inline int square(int x) { //内联函数的使用就是在函数开头加上 inline
2     return x * x;
3 }
4
5 - int main() {
6     int result = square(5); // 编译器会将 square(5) 替换为 5 * 5
7     cout << "The square of 5 is: " << result << endl;
8     return 0;
9 }
```

函数重载（Function Overloading）

函数重载是指在同一个作用域中，允许定义多个同名的函数，只要它们的参数列表不同（即参数的个数、类型或顺序不同）。函数的返回类型不影响重载，因为编译器是通过参数来区分不同的函数调用。

1. 函数重载的规则

- **函数名相同：**所有重载函数的名字必须相同。
- **参数列表不同：**参数的个数、类型或顺序必须不同。
- **返回类型无关：**仅仅改变函数的返回类型不能构成函数重载。

2. 函数重载的示例

```

1 void print(int i) {
2     cout << "Printing integer: " << i << endl;
3 }
4
5 // 打印浮点数
6 void print(double d) {
7     cout << "Printing double: " << d << endl;
8 }
9
10 // 打印字符串
11 void print(string s) {
12     cout << "Printing string: " << s << endl;
13 }
14
15 int main() {
16     print(42);          // 调用 print(int)
17     print(3.14);        // 调用 print(double)
18     print("Hello");    // 调用 print(string)
19     return 0;
20 }
```

3. 重载函数的选择

编译器会根据传递给函数的参数类型来决定调用哪个重载版本。如果传递的参数类型与某个重载版本完全匹配，编译器就会选择该版本。如果没有完全匹配，编译器会尝试将参数转换为可接受的类型，若仍然没有匹配项，则会报错。

4. 重载的注意事项

- 参数个数差异：参数个数不同是最常见的重载形式。

```

1 void func(int a) {
2     cout << "Function with one integer: " << a << endl;
3 }
4
5 void func(int a, int b) {
6     cout << "Function with two integers: " << a << ", " << b << endl;
7 }
```

- 参数类型差异：可以通过参数类型的不同来区分重载函数。

```
1 ~ void func(double d) {  
2     cout << "Function with a double: " << d << endl;  
3 }  
4  
5 ~ void func(int i) {  
6     cout << "Function with an integer: " << i << endl;  
7 }
```

- 参数顺序差异：当参数（类型）顺序不同，编译器也能够区分。

```
1 ~ void func(int a, double b) {  
2     cout << "Function with an int and a double: " << a << ", " << b << endl  
3 ;  
4 }  
5 ~ void func(double b, int a) {  
6     cout << "Function with a double and an int: " << b << ", " << a << endl  
7 ;
```

- 默认参数与重载：如果在函数重载中使用了默认参数，可能会导致重载不明确的问题。例如：

```
1 ~ void func(int a, int b = 10) {  
2     cout << "a: " << a << ", b: " << b << endl;  
3 }  
4  
5 ~ void func(double a) {  
6     cout << "a: " << a << endl;  
7 }
```

调用 `func(5)` 时，编译器就会感到困惑，因为它不确定是调用 `func(int, int)` 还是 `func(double)`，因此会报错。

函数模版 (Function Template)

函数模板允许编写一个函数的模板，而不需要指定具体的数据类型。函数模板可以用于创建一个函数，该函数可以处理多种数据类型，避免了为每个数据类型编写重复的代码。

1. 函数模板的定义

函数模板通过使用 `template` 关键字来定义，通常在函数名之前声明一个模板参数。

```
1 template <typename T>
2 T add(T a, T b) {
3     return a + b;
4 }
```

在这个例子中，`T` 是一个模板参数，表示一个类型占位符，`T` 可以是任何数据类型，`T` 可以在 C++ 标识符的命名规则下任意发挥，如取名：Judy。

`add` 函数的返回类型和参数类型都是 `T`，这样就可以在调用时传递不同的数据类型。

2. 如何使用函数模板

当你调用一个函数模板时，编译器会自动根据传递给模板的实参来推导出类型。例如：

```
1 template <typename judy>
2 judy add(judy a, judy b) {
3     return a + b;
4 }
5
6 int main() {
7     cout << add(10, 20) << endl;           // T推导为 int
8     cout << add(3.14, 1.59) << endl;         // T推导为 double
9     cout << add(1.5f, 2.5f) << endl;         // T推导为 float
10    //输出: 30
11    //    4.73
12    //    4
13    return 0;
14 }
```

3. 模板参数的类型

你可以定义多个模板参数，允许函数接受多种类型的参数。例如：

```
1 template <typename judy, typename fox>
2 auto add(fox a, judy b) { //注意, 如果返回不明确, 函数返回值就写 auto
3     return a + b;
4 }
```

在这个例子中, `T1` 和 `T2` 可以是不同的数据类型, 返回类型由 `auto` 推导。这里的 `auto` 让编译器自动推导返回值的类型。

```
1 cout << add(10, 3.14) << endl; // T1是int, T2是double
```

4. 显式指定模板参数

虽然编译器可以根据参数自动推导模板参数的类型, 但你也可以显式指定模板参数 (是指定, 即告诉编译器类型是什么, 而不是强制转化输出) 。

```
1 cout << add<int,double>(10, 2.5) << endl;
```

5. 函数模板的特化 (Template Specialization)

函数模板可以针对特定类型进行特化。这意味着可以为某些数据类型提供一个特定的实现, 而不是使用通用模板。(如果差别比较大, 那我们为什么不用单纯的函数重载呢?)

```

1 // 普通模板 (用于处理任意类型的单个参数)
2 template <typename judy>
3 - void print(judy value) {
4     cout << "Template version: " << value << endl;
5 }
6
7 // 完全特化 (针对 int 类型)
8 template <>
9 - void print(int value) {
10    cout << "Specialized version for int: " << value << endl;
11 }
12
13 // 完全特化 (针对 double 类型)
14 template <>
15 - void print(double value) {
16    cout << "Specialized version for double: " << value << endl;
17 }
18
19 // 偏特化 (针对第一个参数为 int 类型的情况)
20 template <typename judy>
21 - void print(int t, judy u) {
22    cout << "Specialized version for int and second type: " << t << ", " <<
23    u << endl;
24 }
25
26 - int main() {
27     print(42);           // 使用完全特化, 输出: Specialized version for int:
28     print(3.14);         // 使用完全特化, 输出: Specialized version for doubl
29     print(42, "hello"); // 使用偏特化, 输出: Specialized version for int an
30     return 0;
31 }

```

6. 注意事项

- **函数模板和普通函数的重载:** 如果函数模板与普通函数重载存在冲突, 编译器可能会产生错误。例如, 如果你有一个普通函数与模板函数名字相同, 且参数类型不完全匹配, 编译器可能无法决定调用哪个函数。

```
1 - void print(int x) {
2     cout << "int: " << x << endl;
3 }
4
5 template <typename T>
6 - void print(T x) {
7     cout << "Generic: " << x << endl;
8 }
9
10 - int main() {
11     print(5); // 会调用 void print(int) 而不是模板函数
12 }
```

- 模板实例化：模板只有在你调用时才会被实例化，编译器会根据实际传入的类型生成特定的函数代码。

函数调用堆栈

1. 代码区 (Text Segment)

- 用途：存储程序的机器码，即程序的可执行指令。
- 特点：只读、不可修改，多个进程共享，执行效率高。

示例：

```
1 ~ void foo() {
2     int a = 5;
3     cout << a << endl;
4 }
5
6 ~ int main() {
7     foo(); // 这里 foo() 的机器代码存储在代码区
8     return 0;
9 }
```

2. 全局数据区 (Data Segment)

- 用途：存储全局变量和静态变量。
- 特点：全局变量和静态变量的生命周期是程序运行期间，分为已初始化区和未初始化区（BSS 区）。

示例：

```
1 int globalVar = 10; // 存储在全局数据区（已初始化区）
2
3 ~ void foo() {
4     static int staticVar = 20; // 存储在全局数据区（未初始化区）
5 }
```

3. 堆区 (Heap)

- 用途：用于动态分配内存，通常通过 `new` 或 `malloc()` 分配。
- 特点：程序员手动管理内存，分配大小灵活，但容易导致内存泄漏和碎片化。

示例：

```
1 int* ptr = new int(10); // 在堆区分配内存
2 cout << *ptr << endl; // 输出堆区中的值
3 delete ptr; // 释放堆区内存
```

在这个例子中，`new` 操作符为变量 `ptr` 分配了堆区内存，`delete` 操作符用于释放该内存。

4. 栈区 (Stack)

- 用途：存储局部变量和函数调用的临时信息（如返回地址）。
- 特点：由操作系统自动管理，内存分配和释放迅速，但空间有限，递归调用过多可能导致栈溢出。

示例：

```
1 ▼ void foo() {  
2     int localVar = 10; // 存储在栈区  
3     cout << localVar << endl;  
4 }  
5  
6 ▼ int main() {  
7     foo(); // 在调用 foo() 时, foo 的栈帧会被压入栈中  
8     return 0;  
9 }
```

在这个例子中，`localVar` 存储在栈区。当 `foo()` 被调用时，它的栈帧会被压入栈中，执行完后栈帧会被弹出。

内存分区模型

内存大方向划分为4个区域：

- 代码区：存放函数体的二进制代码，由操作系统进行管理的
- 全局区：存放全局变量和静态变量以及常量
- 栈区：由编译器自动分配释放，存放函数的参数值、局部变量等
- 堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收

1. 程序运行前

在程序编译后，生成了exe可执行程序，未执行该程序前分为两个区域

代码区：

存放 CPU 执行的机器指令。代码区是共享的，共享的目的是对于频繁被执行的程序，只需要在内存中有一份代码即可。代码区是只读的，使其只读的原因是防止程序意外地修改了它的指令。

全局区：

全局变量和静态变量存放在此。全局区还包含了常量区，字符串常量和其他常量也存放在此。该区域的数据在程序结束后由操作系统释放。

示例：

```

1 //全局变量
2 int g_a = 10;
3 int g_b = 10;
4
5 //全局常量
6 const int c_g_a = 10;
7 const int c_g_b = 10;
8
9 - int main() {
10
11     //局部变量
12     int a = 10;
13     int b = 10;
14
15     //打印地址
16     cout << "局部变量a地址为: " << (int)&a << endl;
17     cout << "局部变量b地址为: " << (int)&b << endl;
18
19     cout << "全局变量g_a地址为: " << (int)&g_a << endl;
20     cout << "全局变量g_b地址为: " << (int)&g_b << endl;
21
22     //静态变量
23     static int s_a = 10;
24     static int s_b = 10;
25
26     cout << "静态变量s_a地址为: " << (int)&s_a << endl;
27     cout << "静态变量s_b地址为: " << (int)&s_b << endl;
28
29     cout << "字符串常量地址为: " << (int)&"hello world" << endl;
30     cout << "字符串常量地址为: " << (int)&"hello world1" << endl;
31
32     cout << "全局常量c_g_a地址为: " << (int)&c_g_a << endl;
33     cout << "全局常量c_g_b地址为: " << (int)&c_g_b << endl;
34
35     const int c_l_a = 10;
36     const int c_l_b = 10;
37     cout << "局部常量c_l_a地址为: " << (int)&c_l_a << endl;
38     cout << "局部常量c_l_b地址为: " << (int)&c_l_b << endl;
39
40     return 0;
41 }
```

打印结果：

 图片加载失败 局部变量a地址为： 0x6ffe4c

局部变量b地址为: 0x6ffe48
全局变量g_a地址为: 0x472010
全局变量g_b地址为: 0x472014
静态变量s_a地址为: 0x472018
静态变量s_b地址为: 0x47201c
字符串常量地址为: 0x48808e
字符串常量地址为: 0x48809a
全局常量c_g_a地址为: 0x488104
全局常量c_g_b地址为: 0x488108
局部常量c_l_a地址为: 0x6ffe44
局部常量c_l_b地址为: 0x6ffe40

2. 程序运行后

栈区:

由编译器自动分配释放, 存放函数的参数值, 局部变量等

注意事项:

不要返回局部变量的地址, 栈区开辟的数据由编译器自动释放

示例:

```
1 int * func(){
2     int a = 10;
3     return &a;
4 }
5
6 int main() {
7
8     int *p = func();
9
10    cout << *p << endl;
11    cout << *p << endl;
12    return 0;
13 }
```

堆区：

由程序员分配释放,若程序员不释放,程序结束时由操作系统回收。在C++中主要利用new在堆区开辟内存。

示例：

```
1 int* func(){
2     int* a = new int(10);
3     return a;
4 }
5
6 int main() {
7
8     int *p = func();
9
10    cout << *p << endl;
11    cout << *p << endl;
12    return 0;
13 }
```

总结：

堆区数据由程序员管理开辟和释放

堆区数据利用new关键字进行开辟内存

3. new操作符

利用new操作符在堆区开辟数据。堆区开辟的数据，由程序员手动开辟，手动释放，释放利用操作符 delete。利用new创建的数据，会返回该数据对应的类型的指针。

语法： new 数据类型

示例1： 基本语法

```
1 - int* func(){
2     int* a = new int(10);
3     return a;
4 }
5
6 - int main() {
7     int *p = func();
8     cout << *p << endl;
9
10    //利用delete释放堆区数据
11    delete p;
12    //cout << *p << endl; //报错，释放的空间不可访问
13    return 0;
14 }
```

示例2：开辟数组

```
1 //堆区开辟数组
2 - int main() {
3     int* arr = new int[10];
4
5 -     for (int i = 0; i < 10; i++) {
6         arr[i] = i + 100;
7     }
8
9 -     for (int i = 0; i < 10; i++) {
10        cout << arr[i] << endl;
11    }
12    //释放数组 delete 后加 []
13    delete[] arr;
14    return 0;
15 }
16
```

类和对象

接口分离

源文件包含客户端，例子为main.cpp

```
1 #include <iostream>
2 #include <string>
3 #include "Employee.h" //调用头文件
4 using namespace std;
5 int main() {
6     Employee employee1("Bob", 34500); //这里的Employee就相当于一种结构体
7
8     cout << "Employee 1: " << employee1.getName() << endl; //操作对象.成员函数()
9     cout << "Yearly Salary: " << employee1.getSalary() << endl;
10
11    cout << "Increasing employee salaries by 10%" << endl;
12    employee1.setSalary(employee1.getSalary()*1.1);
13
14    return 0;
15 }
```

头文件包含接口，实现

接口Employee.h

```
1 class Employee {
2 private:
3     std::string Name; //避免污染不用using, 记得要std::
4     int salary;
5 public:
6     Employee(std::string, int); //在接口的函数调用中, 只用指明各个参数的类型, 不用具体的参数名
7     void setName(const std::string); //const在内部说明该参数值在函数内部不能被修改
8     std::string getName() const; //const在外部说明函数返回的值是常量, 不能被修改
9     void setSalary(int );
10    int getSalary() const;
11}; // class 大括号结尾是有'；'的
```

实现employee.cpp

```

1 #include<string>
2 #include<iostream>
3 #include"Employee.h" //在实现中也要调用头文件
4 //这里就不用再提示 class了，我们直接写函数，包括构造函数
5 Employee::Employee(std::string name, int sal) { //这是构造函数，作用是在创建
6     Name = name; //这里的函数都要有Employee
7     if (sal < 0) sal = 0;
8     salary = sal / 12;
9 }
10
11 void Employee::setName(const std::string name) {
12     Name = name;
13 }
14
15 std::string Employee::getName() const {
16     return Name;
17 }
18
19 void Employee::setSalary(int sal) {
20     if (sal < 0) sal = 0;
21     salary = sal / 12;
22 }
23
24 int Employee::getSalary() const {
25     return salary * 12;
26 }

```

封装

C++面向对象的三大特性为：封装、继承、多态

C++认为万事万物都皆为对象，对象上有其属性和行为

意义：

- 将属性和行为作为一个整体

- 将属性和行为加以权限控制

struct和class区别：

在C++中 struct和class唯一的区别就在于 默认的访问权限不同：

- struct 默认权限为公共
- class 默认权限为私有

```
1 -> class C1{  
2     int m_A; //默认是私有权限  
3 };  
4  
5 -> struct C2{  
6     int m_A; //默认是公共权限  
7 };  
8  
9 -> int main() {  
10  
11     C1 c1;  
12     c1.m_A = 10; //错误，访问权限是私有  
13  
14     C2 c2;  
15     c2.m_A = 10; //正确，访问权限是公共  
16  
17     return 0;  
18 }
```

对象的初始化和清理：

构造函数和析构函数

构造函数：主要作用在于创建对象时为对象的成员属性赋值，构造函数由编译器自动调用。

构造函数语法： `类名(){}`

1. 构造函数，没有返回值也不写void
2. 函数名称与类名相同
3. 构造函数可以有参数，因此可以发生重载
4. 程序在调用对象时候会自动调用构造，无须手动调用,而且只会调用一次

析构函数：主要作用在于对象销毁前系统自动调用。

析构函数语法： `~类名(){}`

1. 析构函数，没有返回值也不写void
2. 函数名称与类名相同,在名称前加上符号 ~
3. 析构函数不可以有参数，因此不可以发生重载
4. 程序在对象销毁前会自动调用析构，无须手动调用,而且只会调用一次

构造函数的分类及调用

两种分类方式：

按参数分为： 有参构造和无参构造

按类型分为： 普通构造和拷贝构造

三种调用方式：括号法，显示法，隐式转换法

示例：

```

1  class Person {
2      public:
3          //无参（默认）构造函数
4          Person() {
5              cout << "无参构造函数!" << endl;
6          }
7          //有参构造函数
8          Person(int a) {
9              age = a;
10             cout << "有参构造函数!" << endl;
11         }
12         //拷贝构造函数
13         Person(Person const& p) {
14             age = p.age;
15             cout << "拷贝构造函数!" << endl;
16         }
17         //析构函数
18         ~Person() {
19             cout << "析构函数!" << endl;
20         }
21
22     public:
23         int age;
24     };
25
26     //2、构造函数的调用
27     //调用无参构造函数
28     void test01() {
29         Person p; //调用无参构造函数
30     }
31
32     //调用有参的构造函数
33     void test02() {
34         //2.1 括号法，常用
35         Person p1(10);
36         //注意1：调用无参构造函数不能加括号，如果加了编译器认为这是一个函数声明
37         //Person p2();
38
39         //2.2 显式法
40         Person p2 = Person(10);
41         Person p3 = Person(p2);
42         //Person(10)单独写就是匿名对象 当前行结束之后，马上析构
43
44         //2.3 隐式转换法
45         Person p4 = 10; // Person p4 = Person(10);
46         Person p5 = p4; // Person p5 = Person(p4);
47

```

```
48     //注意2：不能利用 拷贝构造函数 初始化匿名对象 编译器认为是对象声明
49     //Person p5(p4);
50 }
51
52 int main() {
53     test01();
54     //test02();
55     return 0;
56 }
```

拷贝构造函数调用时机

通常有三种情况：

- 使用一个已经创建完毕的对象来初始化一个新对象
- 值传递的方式给函数参数传值
- 以值方式返回局部对象

```

1  class Person {
2      public:
3          Person() {
4              cout << "无参构造函数!" << endl;
5              mAge = 0;
6          }
7          Person(int age) {
8              cout << "有参构造函数!" << endl;
9              mAge = age;
10         }
11         Person(const Person& p) {
12             cout << "拷贝构造函数!" << endl;
13             mAge = p.mAge;
14         }
15     //析构函数在释放内存之前调用
16     ~Person() {
17         cout << "析构函数!" << endl;
18     }
19     private:
20         int mAge;
21     };
22
23     //1. 使用一个已经创建完毕的对象来初始化一个新对象
24     void test01() {
25         Person man(100); //p对象已经创建完毕
26         Person newman(man); //调用拷贝构造函数
27         Person newman2 = man; //拷贝构造
28
29         //Person newman3;
30         //newman3 = man; //不是调用拷贝构造函数，赋值操作，即调用运算重载符
31     }
32
33     //2. 值传递的方式给函数参数传值
34     //相当于Person p1 = p;
35     void doWork(Person p1) {}
36     void test02() {
37         Person p; //无参构造函数
38         doWork(p);
39     }
40
41     //3. 以值方式返回局部对象
42     Person doWork2() {
43         Person p1;
44         cout << (int *)&p1 << endl; //这里就是指针，即输出地址
45         return p1;
46     }
47     void test03() {

```

```
48     Person p = doWork2();
49     cout << (int *)&p << endl;
50 }
51
52
53 int main() {
54     //test01();
55     //test02();
56     test03();
57     return 0;
58 }
```

转换构造函数

定义：

转换构造函数是能通过不同类型参数构造对象的构造函数。

其标准形式为：

```
1 ClassName(T arg); // T 表示其他类型 (如 int、double)
```

核心作用：

1. 实现从其他类型到当前类的**隐式类型转换**
2. 提供灵活的对象构造方式

注意事项：

1. 隐式类型转换

- 默认允许从参数类型隐式转换为当前类对象。（隐式就是直接一个=；而显式就是要从新定义一个类型+ ()）
- 例如： `MagicScroll s = "Heal";` 等价于 `MagicScroll s("Heal");`

- 风险：可能导致意外的隐式转换（如 `MagicScroll s = 3.14;` 若存在 `double` 参数的构造函数）。

2. `explicit` 关键字

- 禁止隐式转换，强制要求显式调用构造函数。

```
1 explicit MagicScroll(int level); // 必须显式调用 MagicScroll(5)
```

- 若未加 `explicit`，`MagicScroll s = 5` 是合法的；加了 `explicit` 后必须写成 `MagicScroll s(5)`。

3. 多参数转换构造

- 如果构造函数有多个参数，但除第一个外都有默认值，仍可视为转换构造函数。

```
1 MagicScroll(int power, std::string type = "Attack");
```

示例：

```
1 public:
2 // 转换构造函数: int → Date
3 Date(int y) : year(y) {}
4 ;
5
6 // 隐式转换示例
7 void printDate(Date d) {
8     /* ... */
9 }
10
11 int main() {
12     printDate(2023); // 隐式调用 Date(2023)
13 }
```

拷贝构造函数和转换构造函数区别

特性	拷贝构造函数	转换构造函数
----	--------	--------

参数类型	同类对象的引用	其他类型参数
调用场景	同类型对象初始化	不同类型→当前类转换
隐式行为	总是允许	默认允许（可禁用）

示例

```

1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5
6 - class MagicScroll {
7     string spellContent;
8     bool isAncientCopy; // 标识是否是古老复刻版
9     //没有访问控制修饰符，默认private，而结构体默认public
10
11 public:
12     /* 基础转换构造函数（支持隐式转换）*/
13 -     MagicScroll(string const& spell) : spellContent("现代魔咒: " + spell), is
14     AncientCopy(false) {
15         cout << "✨ 转换构造新卷轴: " << spellContent << endl;
16     }
17 -     MagicScroll(char const* spell) : spellContent("现代魔咒: " + string(spell
18     )), isAncientCopy(false) {
19         cout << "✨ 转换构造新卷轴: " << spellContent << endl;
20     }
21
22     /* 高级转换构造函数（显式调用）*/
23 -     explicit MagicScroll(int starLevel) {
24         ostringstream oss;
25         oss << "星际秘法_" << starLevel << "级";
26         spellContent = oss.str();
27         isAncientCopy = (starLevel > 10); // 超过10级的视为古老卷轴
28         cout << "🌌 星际转换构造: " << spellContent << endl;
29     }
30
31     /* 标准拷贝构造函数 */
32 -     MagicScroll(MagicScroll const& other)
33         : spellContent(other.isAncientCopy ? "上古复刻: " + other.spellContent
34         : // 添加古老标识
35             "普通复制: " + other.spellContent),
36             // 普通复制标识
37             isAncientCopy(false) {
38             // 复制品不再标记为古老
39             cout << "⌚ 复制生成: " << spellContent << endl;
40         }
41
42     /* 古老卷轴专用拷贝构造（演示重载拷贝构造）*/
43 -     MagicScroll(MagicScroll const& other, bool forceAncient)
44         : spellContent("[禁术]" + other.spellContent), isAncientCopy(true) {
45             cout << "🔮 秘法复刻: " << spellContent << endl;
46         }

```

```

43
44     void reveal() const {
45         cout << "卷轴解密 ▶ " << spellContent << (isAncientCopy ? "(上古)" : "
46         (现代)") << endl;
47     }
48
49     // 制作古老副本的工厂方法
50     static MagicScroll MakeAncientCopy(MagicScroll const& original) {
51         return MagicScroll(original, true); // 调用专用拷贝构造
52     }
53 }
54
55     void enchantScroll(MagicScroll sc) {
56         cout << "施法效果: ";
57         sc.reveal();
58     }
59
60     int main() {
61         cout << "===== 转换构造演示 =====\n";
62         MagicScroll modernScroll = "火焰冲击"; // 隐式转换构造
63         MagicScroll starScroll = MagicScroll(7); // 显式星际转换
64
65         cout << "\n===== 普通拷贝构造 =====\n";
66         MagicScroll copiedScroll = modernScroll; // 调用标准拷贝构造
67         enchantScroll(starScroll); // 传参时拷贝构造
68
69         cout << "\n===== 古老复制流程 =====\n";
70         MagicScroll ancientCopy = MagicScroll::MakeAncientCopy(modernScroll);
71
72         cout << "\n===== 混合转换案例 =====\n";
73         enchantScroll("寒冰箭"); // 隐式转换+拷贝构
74         造
75         MagicScroll hybridScroll = static_cast<MagicScroll>(9); // 显式数字转换
76
77         cout << "\n===== 最终卷轴状态 =====\n";
78         modernScroll.reveal();
79         copiedScroll.reveal();
80         ancientCopy.reveal();
81         hybridScroll.reveal();
82     }

```

构造函数调用规则

默认情况下，c++编译器至少给一个类添加3个函数

1. 默认构造函数(无参， 函数体为空)
2. 默认析构函数(无参， 函数体为空)
3. 默认拷贝构造函数， 对属性进行值拷贝

构造函数调用规则如下：

- 如果用户定义有参构造函数，c++不在提供默认无参构造，但是会提供默认拷贝构造
- 如果用户定义拷贝构造函数，c++不会再提供其他构造函数

```

1 -> class Person {
2     public:
3         //无参（默认）构造函数
4     Person() {
5         cout << "无参构造函数!" << endl;
6     }
7     //有参构造函数
8     Person(int a) {
9         age = a;
10        cout << "有参构造函数!" << endl;
11    }
12    //拷贝构造函数
13    Person(const Person& p) {
14        age = p.age;
15        cout << "拷贝构造函数!" << endl;
16    }
17    //析构函数
18    ~Person() {
19        cout << "析构函数!" << endl;
20    }
21    public:
22        int age;
23    };
24
25 -> void test01() {
26     Person p1(18);
27     //如果不写拷贝构造，编译器会自动添加拷贝构造，并且做浅拷贝操作
28     Person p2(p1);
29
30     cout << "p2的年龄为: " << p2.age << endl;
31 }
32
33 -> void test02() {
34     //如果用户提供有参构造，编译器不会提供默认构造，会提供拷贝构造
35     Person p1; //此时如果用户自己没有提供默认构造，会出错
36     Person p2(10); //用户提供的有参
37     Person p3(p2); //此时如果用户没有提供拷贝构造，编译器会提供
38
39     //如果用户提供拷贝构造，编译器不会提供其他构造函数
40     Person p4; //此时如果用户自己没有提供默认构造，会出错
41     Person p5(10); //此时如果用户自己没有提供有参，会出错
42     Person p6(p5); //用户自己提供拷贝构造
43 }
44
45 -> int main() {
46     test01();
47     return 0;

```

深拷贝与浅拷贝

浅拷贝：简单的赋值拷贝操作

深拷贝：在堆区重新申请空间，进行拷贝操作

```

1  class Person {
2      public:
3      //无参（默认）构造函数
4      Person() {
5          cout << "无参构造函数!" << endl;
6      }
7      //有参构造函数
8      Person(int age ,int height) {
9          cout << "有参构造函数!" << endl;
10         m_age = age;
11         m_height = new int(height);
12     }
13     //拷贝构造函数 自己实现拷贝构造函数 解决拷贝带来的问题
14     Person(const Person& p) {
15         cout << "拷贝构造函数!" << endl;
16         //如果不利用深拷贝在堆区创建新内存，会导致浅拷贝带来的重复释放堆区问题
17         m_age = p.m_age;
18         m_height = new int(*p.m_height);
19         //这里重新开辟内存空间，可以用深拷贝解决浅拷贝的问题
20     }
21
22     //析构函数
23     ~Person() {
24         cout << "析构函数!" << endl;
25         if (m_height != NULL) {
26             delete m_height; // 堆区手动开辟的数据记得要释放
27             //m_height = NULL;
28             // 防止野指针的出现，规范操作。
29             // 但是！注意浅拷贝带来的重复释放内存问题。
30         }
31     }
32     public:
33     int m_age;
34     int* m_height;
35 };
36
37 void test01() {
38     Person p1(18, 180);
39     Person p2(p1);
40
41     cout << "p1的年龄: " << p1.m_age << " 身高: " << *p1.m_height << endl;
42     cout << "p2的年龄: " << p2.m_age << " 身高: " << *p2.m_height << endl;
43 }
44
45 int main() {
46     test01();
47     return 0;

```

初始化列表

作用：

C++提供了初始化列表语法，用来初始化属性

语法：

构造函数()：属性1(值1), 属性2 (值2) ... {}

```
1 class Person {
2     public:
3
4     ////传统方式初始化
5     //Person(int a, int b, int c) {
6     //    m_A = a;
7     //    m_B = b;
8     //    m_C = c;
9     //}
10
11    //初始化列表方式初始化
12    Person(int a, int b, int c) :m_A(a), m_B(b), m_C(c) {
13
14    }
15    void PrintPerson() {
16        cout << "mA:" << m_A << endl;
17        cout << "mB:" << m_B << endl;
18        cout << "mC:" << m_C << endl;
19    }
20    private:
21        int m_A;
22        int m_B;
23        int m_C;
24    };
25
26    int main() {
27        Person p(1, 2, 3);
28        p.PrintPerson();
29        return 0;
30    }
}
```

类对象作为类成员

对象成员：类中的成员可以是另一个类的对象。

示例：

```
1 class A {}
2 - class B {
3     A a;
4 }
```

构造与析构的顺序

全局对象:

构造: 在任何函数(含main)执行前

析构: 在程序结束时

局部对象:

自动变量

构造: 对象定义时

析构: 块结束时

静态变量

构造: 首次定义时

析构: 程序结束时

规则:

1. 多个全局和静态局部对象(均为静态存储类别, 程序结束时析构)析构顺序恰好与构造顺序相反。
2. 先调用对象成员的构造, 再调用本类构造。
3. 调用exit函数退出程序执行时, 不调用剩余自动对象的析构函数。
4. 调用abort函数退出程序执行时, 不调用任何剩余对象的析构函数。

示例:

```

1 -> class Phone {
2     public:
3 ->         Phone(string name) {
4             m_PhoneName = name;
5             cout << "Phone构造" << endl;
6         }
7
8 ->         ~Phone() {
9             cout << "Phone析构" << endl;
10            }
11            string m_PhoneName;
12        };
13
14 -> class Person {
15     public:
16     //初始化列表可以告诉编译器调用哪一个构造函数
17 ->         Person(string name, string pName) :m_Name(name), m_Phone(pName) {
18             cout << "Person构造" << endl;
19         }
20 ->         ~Person() {
21             cout << "Person析构" << endl;
22         }
23 ->         void playGame() {
24             cout << m_Name << " 使用" << m_Phone.m_PhoneName << " 牌手机!" 
25             << endl;
26         }
27         string m_Name;
28         Phone m_Phone;
29     //这里就是先调用了对象成员的构造，在调用本类构造
30    };
31 -> void test01() {
32     //当类中成员是其他类对象时，我们称该成员为 对象成员
33     //构造的顺序是：先调用对象成员的构造，再调用本类构造
34     //析构顺序与构造相反
35     Person p("张三", "苹果X");
36     p.playGame();
37
38 }
39
40
41 -> int main() {
42     test01();
43     //输出： Phone构造
44     //      Person构造
45     //      张三 使用苹果X 牌手机！
46     //      Person析构

```

```
47      //      Phone析构  
48      return 0;  
49 }
```

静态成员

静态成员：在成员变量和成员函数前加上关键字 **static**。

静态成员分为：静态成员变量和静态成员函数。

静态成员变量：

- 所有对象共享同一份数据（就相当于全局的）
- 在编译阶段分配内存（在程序运行前就分配内存了）
- 类内声明，**类外初始化**
- 静态成员变量也是有访问权限的
- 有两种访问方式
- 属于类本身而不是实例（即存在第二种通过类名的访问方式）

静态成员函数：

- 所有对象共享同一个函数
- 静态成员函数只能访问静态成员变量
- 和静态成员函数一样，有两种访问方式
- 静态成员函数也是有访问权限的

示例1：静态成员变量

```
1 -> class Person {
2     public:
3         static int m_A; //静态成员变量
4     private:
5         static int m_B; //静态成员变量也是有访问权限的
6 };
7 int Person::m_A = 10; //先在类外初始化
8 int Person::m_B = 10;
9
10-> void test01() {
11     //静态成员变量两种访问方式
12
13     //1、通过对象
14     Person p1;
15     p1.m_A = 100;
16     cout << "p1.m_A = " << p1.m_A << endl;
17
18     Person p2;
19     p2.m_A = 200;
20     cout << "p1.m_A = " << p1.m_A << endl; //共享同一份数据
21     cout << "p2.m_A = " << p2.m_A << endl; //所以输出都为200
22
23     //2、通过类名
24     cout << "m_A = " << Person::m_A << endl;
25     //cout << "m_B = " << Person::m_B << endl; //私有权限访问不到
26 }
27
28-> int main() {
29     test01();
30     return 0;
31 }
```

示例2：静态成员函数

```
1 -> class Person {
2     public:
3 ->     static void func() {
4         //前面有了static，就是静态成员函数
5         cout << "func调用" << endl;
6         m_A = 100;
7         //m_B = 100;
8         //错误，不可以访问非静态成员变量
9     }
10
11     static int m_A; //静态成员变量
12     int m_B; //非静态成员变量
13 private:
14 //静态成员函数也是有访问权限的
15 ->     static void func2() {
16         cout << "func2调用" << endl;
17     }
18 };
19 int Person::m_A = 10;
20
21
22 -> void test01() {
23     //静态成员变量两种访问方式
24
25     //1、通过对象
26     Person p1;
27     p1.func();
28
29     //2、通过类名
30     Person::func();
31
32     //私有权限访问不到
33     //Person::func2();
34 }
35
36 -> int main() {
37     test01();
38     return 0;
39 }
```

对象模型（类如何占用内存）

类内的成员变量和成员函数分开存储，只有非静态成员变量才属于类的对象上。

```
1  class Person {
2      public:
3          Person() {
4              mA = 0;
5          }
6          //非静态成员变量占对象空间, mA占4字节
7          int mA;
8
9          //静态成员变量不占对象空间
10         static int mB;
11
12         //函数也不占对象空间, 所有函数共享一个函数实例
13         void func() {
14             cout << "mA:" << this->mA << endl;
15         }
16         //静态成员函数也不占对象空间
17         static void sfunc() {
18     }
19 };
20
21 //class Person{}
22 //空对象占用内存空间为: 1
23 //C++编译器会给每一个空对象分配一个字节空间, 有一个独一无二的内存地址, 为了区分空对象占
24 //内存的位置。
25 int main() {
26     cout << sizeof(Person) << endl;
27     return 0;
28 }
```

this指针

每一个非静态成员函数只会诞生一份函数实例，也就是说多个同类型的对象会共用一块代码。

那么问题是：这一块代码是如何区分那个对象调用自己的呢？

this指针指向被调用的成员函数所属的对象，隐含每一个非静态成员函数内的一种指针，不需要定义，直接使用即可。

用途：

- 当形参和成员变量同名时，可用this指针来区分，解决名称冲突。
- 在类的非静态成员函数中返回对象本身，可使用return *this。

```
1 -> class Person {
2     public:
3 ->         Person(int age) {
4             //1、当形参和成员变量同名时，可用this指针来区分
5             this->age = age;
6         }
7 ->         Person& PersonAddPerson(Person p) {
8             this->age += p.age;
9             //返回对象本身
10            return *this;
11        }
12        int age;
13    };
14
15 -> void test01() {
16     Person p1(10);
17     cout << "p1.age = " << p1.age << endl;
18     //这里就输出10
19
20     Person p2(10);
21     p2.PersonAddPerson(p1).PersonAddPerson(p1).PersonAddPerson(p1);
22     cout << "p2.age = " << p2.age << endl;
23     //这里就输出40
24 }
25
26 -> int main() {
27     test01();
28     return 0;
29 }
```

空指针访问成员函数

空指针也是可以调用成员函数的，但是也要注意有没有用到this指针

如果用到this指针，需要加以判断保证代码的健壮性

示例：

```
1 //空指针访问成员函数
2 class Person {
3 public:
4     void ShowClassName() {
5         cout << "我是Person类!" << endl;
6     }
7
8     void ShowPerson() {
9         if (this == NULL) {
10             return;
11         }
12         cout << mAge << endl;
13     }
14 public:
15     int mAge;
16 };
17
18 void test01() {
19     Person *p = NULL;
20     p->ShowClassName(); //空指针，可以调用成员函数
21     p->ShowPerson(); //但是如果成员函数中用到了this指针，就不可以了
22 }
23
24 int main() {
25     test01();
26     return 0;
27 }
```

const修饰成员函数

常函数：

- 成员函数后加const后我们称为这个函数为**常函数**
- 常函数内**不可以修改成员属性**
- 成员属性声明时加关键字mutable后，在常函数中依然可以修改

常对象：

- 声明对象前加const称该对象为常对象
- 常对象只能调用常函数

示例：

```

1  class Person {
2  public:
3  Person() {
4      m_A = 0;
5      m_B = 0;
6  }
7
8 //this指针的本质是一个指针常量，指针的指向不可修改
9 //如果想让指针指向的值也不可以修改，需要声明常函数
10 void ShowPerson() const {
11     //const Type* const pointer;
12     //this = NULL; //不能修改指针的指向 Person* const this;
13     //this->mA = 100; //但是this指针指向的对象的数据是可以修改的
14
15     //const修饰成员函数，表示指针指向的内存空间的数据不能修改，除了mutable修饰的
16     //变量
16     this->m_B = 100;
17 }
18
19 void MyFunc() const {
20     //mA = 10000;
21 }
22 public:
23     int m_A;
24     mutable int m_B; //可修改 可变的
25 };
26
27 //const修饰对象 常对象
28 void test01() {
29     const Person person; //常量对象
30     cout << person.m_A << endl;
31     //person.mA = 100; //常对象不能修改成员变量的值，但是可以访问
32     person.m_B = 100; //但是常对象可以修改mutable修饰成员变量
33
34     //常对象访问成员函数
35     person.MyFunc(); //常对象不能调用const的函数
36 }
37
38 int main() {
39     test01();
40     return 0;
41 }

```

友元

有些私有属性也想让类外特殊的一些函数或者类进行访问，就需要用到友元的技术。友元的目的就是让一个函数或者类访问另一个类中私有成员。友元的关键字为 `friend`。

友元的三种实现

- 全局函数做友元
- 类做友元
- 成员函数做友元

全局函数做友元

```
1  class Building {
2      //告诉编译器 goodGay全局函数 是 Building类的好朋友，可以访问类中的私有内容
3      friend void goodGay(Building * building){
4          cout << "好基友正在访问: " << building->m_SittingRoom << endl;
5          cout << "好基友正在访问: " << building->m_BedRoom << endl;
6      }
7      public:
8      Building() {
9          this->m_SittingRoom = "客厅";
10         this->m_BedRoom = "卧室";
11     }
12     public:
13     string m_SittingRoom; //客厅
14     private:
15     string m_BedRoom; //卧室
16 };
17
18 void test01() {
19     Building b;
20     goodGay(&b);
21 }
22
23 int main() {
24     test01();
25     return 0;
26 }
```

类做友元

```
1 -> class Building {
2     //告诉编译器 goodGay类是Building类的好朋友，可以访问到Building类中私有内容
3         friend class goodGay;
4     public:
5         Building() {
6             this->m_SittingRoom = "客厅";
7             this->m_BedRoom = "卧室";
8         }
9         string m_SittingRoom; //客厅
10    private:
11        string m_BedRoom;//卧室
12    };
13
14 -> class goodGay {
15     public:
16         goodGay() {
17             building = new Building;
18         }
19         void visit() {
20             cout << "好基友正在访问" << building->m_SittingRoom << endl;
21             cout << "好基友正在访问" << building->m_BedRoom << endl;
22         }
23     private:
24         Building *building;
25     };
26
27 -> void test01() {
28     goodGay gg;
29     gg.visit();
30 }
```

成员函数做友元

```

1 // 前向声明
2 class Building;
3
4 // 提前声明goodGay类, 但成员函数稍后定义
5 - class goodGay {
6     public:
7         goodGay();
8         void visit();    // 声明为友元的函数
9         void visit2();  // 普通成员函数
10    private:
11        Building* building;
12    };
13
14 // Building类的完整定义
15 - class Building {
16     // 声明goodGay的visit函数为友元
17     friend void goodGay::visit();
18     public:
19         Building() : m_SittingRoom("客厅"), m_BedRoom("卧室") {}
20         string m_SittingRoom;
21     private:
22         string m_BedRoom;
23    };
24
25 // goodGay成员函数的实现 (此时Building已完整定义)
26 goodGay::goodGay() : building(new Building) {}
27
28 - void goodGay::visit() {
29     cout << "好基友正在访问" << building->m_SittingRoom << endl;
30     cout << "好基友正在访问" << building->m_BedRoom << endl; // 友元可访问私
31     有成员
32 }
33
34 - void goodGay::visit2() {
35     cout << "普通访问: " << building->m_SittingRoom << endl;
36     // cout << building->m_BedRoom << endl; // 编译错误, 非友元无法访问
37 }
38
39 - void test01() {
40     goodGay gg;
41     gg.visit();
42     gg.visit2();
43 }
44
45 - int main() {
46     test01();
47     return 0;

```

运算符重载

概念：对已有的运算符重新进行定义，赋予其另一种功能，以适应不同的数据类型

加号运算符重载

作用：实现两个自定义数据类型相加的运算

注意：

- 对于内置的数据类型的表达式的的运算符是不可能改变的
- 不要滥用运算符重载

```

1  -> class Person {
2      public:
3          Person() {};
4      -> Person(int a, int b) {
5              this->m_A = a;
6              this->m_B = b;
7          }
8      //成员函数实现 + 号运算符重载
9      -> Person operator+(const Person& p) {
10         Person temp;
11         temp.m_A = this->m_A + p.m_A;
12         temp.m_B = this->m_B + p.m_B;
13         return temp;
14     }
15     public:
16         int m_A;
17         int m_B;
18     };
19
20 //全局函数实现 + 号运算符重载
21 //Person operator+(const Person& p1, const Person& p2) {
22 //    Person temp(0, 0);
23 //    temp.m_A = p1.m_A + p2.m_A;
24 //    temp.m_B = p1.m_B + p2.m_B;
25 //    return temp;
26 //}
27
28 //运算符重载 可以发生函数重载
29 -> Person operator+(const Person& p2, int val) {
30     Person temp;
31     temp.m_A = p2.m_A + val;
32     temp.m_B = p2.m_B + val;
33     return temp;
34 }
35
36 -> void test() {
37     Person p1(10, 10);
38     Person p2(20, 20);
39
40     //成员函数方式
41     Person p3 = p2 + p1; //相当于 p2.operator+(p1)
42     cout << "mA:" << p3.m_A << " mB:" << p3.m_B << endl;
43
44     Person p4 = p3 + 10; //相当于 operator+(p3, 10)
45     cout << "mA:" << p4.m_A << " mB:" << p4.m_B << endl;
46
47 }

```

```
48
49     int main() {
50         test();
51         return 0;
52     }
```

左移运算符重载

作用：可以输出自定义数据类型

```

1 -> class Person {
2     friend ostream& operator<<(ostream& out, Person& p);
3     public:
4     ->     Person(int a, int b) {
5             this->m_A = a;
6             this->m_B = b;
7         }
8
9     //成员函数 实现不了 p << cout 不是我们想要的效果
10    //void operator<<(Person& p){
11    //}
12
13    private:
14        int m_A;
15        int m_B;
16    };
17
18    //全局函数实现左移重载
19    //ostream对象只能有一个
20    -> ostream& operator<<(ostream& out, Person& p) {
21        out << "a:" << p.m_A << " b:" << p.m_B;
22        return out;
23    }
24
25    -> void test() {
26        Person p1(10, 20);
27        cout << p1 << "hello world" << endl; //链式编程
28    }
29
30    -> int main() {
31        test();
32        return 0;
33    }

```

| 总结：重载左移运算符配合友元可以实现输出自定义数据类型

递增运算符重载

作用：通过重载递增运算符，实现自己的整型数据

总结：前置递增返回引用，后置递增返回值

```

1  class MyInteger {
2      friend ostream& operator<<(ostream& out, MyInteger myint);
3  public:
4      MyInteger() {
5          m_Num = 0;
6      }
7      //前置++
8      MyInteger& operator++() {
9          //先++
10         m_Num++;
11         //再返回
12         return *this;
13     }
14
15     //后置++
16     MyInteger operator++(int) {
17         //先返回
18         MyInteger temp = *this; //记录当前本身的值，然后让本身的值加1，但是返回的是以前的值，达到先返回后++;
19         m_Num++;
20         return temp;
21     }
22
23 private:
24     int m_Num;
25 };
26
27     ostream& operator<<(ostream& out, MyInteger myint) {
28         out << myint.m_Num;
29         return out;
30     }
31
32     //前置++ 先++ 再返回
33     void test01() {
34         MyInteger myInt;
35         cout << ++myInt << endl;
36         cout << myInt << endl;
37     }
38
39     //后置++ 先返回 再++
40     void test02() {
41         MyInteger myInt;
42         cout << myInt++ << endl;
43         cout << myInt << endl;
44     }
45
46     int main() {

```

```
47     test01();  
48     //test02();  
49     return 0;  
50 }
```

赋值运算符重载

c++编译器至少给一个类添加4个函数

1. 默认构造函数(无参, 函数体为空)
2. 默认析构函数(无参, 函数体为空)
3. 默认拷贝构造函数, 对属性进行值拷贝
4. 赋值运算符 operator=, 对属性进行值拷贝

如果类中有属性指向堆区, 做赋值操作时也会出现深浅拷贝问题

```
1 -> class Person {
2     public:
3 ->         Person(int age) {
4             //将年龄数据开辟到堆区
5             m_Age = new int(age);
6         }
7
8     //重载赋值运算符
9 ->     Person& operator=(Person &p) {
10    if (m_Age != NULL) {
11        delete m_Age;
12        m_Age = NULL;
13    }
14    //编译器提供的代码是浅拷贝
15    //m_Age = p.m_Age;
16
17    //提供深拷贝 解决浅拷贝的问题
18    m_Age = new int(*p.m_Age);
19
20    //返回自身
21    return *this;
22 }
23
24
25 -> ~Person() {
26    if (m_Age != NULL) {
27        delete m_Age;
28        m_Age = NULL;
29    }
30 }
31
32 //年龄的指针
33     int *m_Age;
34
35 };
36
37
38 -> void test01() {
39     Person p1(18);
40     Person p2(20);
41     Person p3(30);
42     p3 = p2 = p1; //赋值操作
43     cout << "p1的年龄为: " << *p1.m_Age << endl;
44     cout << "p2的年龄为: " << *p2.m_Age << endl;
45     cout << "p3的年龄为: " << *p3.m_Age << endl;
46 }
47
```

```
48 int main() {
49     test01();
50     //int a = 10;
51     //int b = 20;
52     //int c = 30;
53
54     //c = b = a;
55     //cout << "a = " << a << endl;
56     //cout << "b = " << b << endl;
57     //cout << "c = " << c << endl;
58     return 0;
59 }
```

关系运算符重载

作用：重载关系运算符，可以让两个自定义类型对象进行对比操作

示例：

```
1 -> class Person {
2     public:
3 ->         Person(string name, int age) {
4             this->m_Name = name;
5             this->m_Age = age;
6         };
7
8 ->         bool operator==(Person & p) {
9             if (this->m_Name == p.m_Name && this->m_Age == p.m_Age) {
10                return true;
11            } else {
12                return false;
13            }
14        }
15
16 ->         bool operator!=(Person & p) {
17             if (this->m_Name == p.m_Name && this->m_Age == p.m_Age) {
18                 return false;
19             } else {
20                 return true;
21             }
22         }
23
24         string m_Name;
25         int m_Age;
26     };
27
28 -> void test01() {
29     //int a = 0;
30     //int b = 0;
31
32     Person a("孙悟空", 18);
33     Person b("孙悟空", 18);
34
35 ->     if (a == b) {
36         cout << "a和b相等" << endl;
37     } else {
38         cout << "a和b不相等" << endl;
39     }
40
41 ->     if (a != b) {
42         cout << "a和b不相等" << endl;
43     } else {
44         cout << "a和b相等" << endl;
45     }
46 }
47 }
```

```
48
49  int main() {
50
51      test01();
52
53      system("pause");
54
55      return 0;
56 }
```

函数调用运算符重载

- 函数调用运算符 () 也可以重载
- 由于重载后使用的方式非常像函数的调用，因此称为仿函数
- 仿函数没有固定写法，非常灵活

示例：

```
1 -> class MyPrint {
2     public:
3 ->         void operator()(string text) {
4             cout << text << endl;
5         }
6
7     };
8 -> void test01() {
9     //重载的 () 操作符 也称为仿函数
10    MyPrint myFunc;
11    myFunc("hello world");
12 }
13
14
15 -> class MyAdd {
16     public:
17 ->         int operator()(int v1, int v2) {
18             return v1 + v2;
19         }
20     };
21
22 -> void test02() {
23     MyAdd add;
24     int ret = add(10, 10);
25     cout << "ret = " << ret << endl;
26
27     //匿名对象调用
28     cout << "MyAdd()(100,100) = " << MyAdd()(100, 100) << endl;
29 }
30
31 -> int main() {
32
33     test01();
34     test02();
35
36     system("pause");
37
38     return 0;
39 }
```

继承

基本语法

继承意义：可以减少重复的代码。

class A : public B;

A 类称为子类 或 派生类

B 类称为父类 或 基类

派生类中包含两大部分成员：

1.从基类继承过来的 2.自己增加的成员。

普通实现：

```
1 //Java页面
2 -> class Java {
3     public:
4         void header() {
5             cout << "首页、公开课、登录、注册... (公共头部)" << endl;
6         }
7         void footer() {
8             cout << "帮助中心、交流合作、站内地图... (公共底部)" << endl;
9         }
10        void left() {
11            cout << "Java,Python,C++... (公共分类列表)" << endl;
12        }
13        void content() {
14            cout << "JAVA学科视频" << endl;
15        }
16    };
17
18 //Python页面
19 -> class Python {
20     public:
21         void header() {
22             cout << "首页、公开课、登录、注册... (公共头部)" << endl;
23         }
24         void footer() {
25             cout << "帮助中心、交流合作、站内地图... (公共底部)" << endl;
26         }
27         void left() {
28             cout << "Java,Python,C++... (公共分类列表)" << endl;
29         }
30         void content() {
31             cout << "Python学科视频" << endl;
32         }
33    };
34
35 //C++页面
36 -> class CPP {
37     public:
38         void header() {
39             cout << "首页、公开课、登录、注册... (公共头部)" << endl;
40         }
41         void footer() {
42             cout << "帮助中心、交流合作、站内地图... (公共底部)" << endl;
43         }
44         void left() {
45             cout << "Java,Python,C++... (公共分类列表)" << endl;
46         }
47         void content() {
```

```
48             cout << "C++学科视频" << endl;
49         }
50     };
51
52     void test01() {
53         //Java页面
54         cout << "Java下载视频页面如下: " << endl;
55         Java ja;
56         ja.header();
57         ja.footer();
58         ja.left();
59         ja.content();
60         cout << "-----" << endl;
61
62         //Python页面
63         cout << "Python下载视频页面如下: " << endl;
64         Python py;
65         py.header();
66         py.footer();
67         py.left();
68         py.content();
69         cout << "-----" << endl;
70
71         //C++页面
72         cout << "C++下载视频页面如下: " << endl;
73         CPP cp;
74         cp.header();
75         cp.footer();
76         cp.left();
77         cp.content();
78
79     }
80
81     int main() {
82         test01();
83         return 0;
84     }
}
```

继承实现：

```
1 //公共页面
2 -> class BasePage {
3     public:
4         void header() {
5             cout << "首页、公开课、登录、注册... (公共头部)" << endl;
6         }
7
8         void footer() {
9             cout << "帮助中心、交流合作、站内地图... (公共底部)" << endl;
10        }
11        void left() {
12            cout << "Java,Python,C++... (公共分类列表)" << endl;
13        }
14    };
15
16
17 //Java页面
18 -> class Java : public BasePage {
19     public:
20         void content() {
21             cout << "JAVA学科视频" << endl;
22         }
23    };
24 //Python页面
25 -> class Python : public BasePage {
26     public:
27         void content() {
28             cout << "Python学科视频" << endl;
29         }
30    };
31 //C++页面
32 -> class CPP : public BasePage {
33     public:
34         void content() {
35             cout << "C++学科视频" << endl;
36         }
37    };
38
39 -> void test01() {
40     //Java页面
41     cout << "Java下载视频页面如下：" << endl;
42     Java ja;
43     ja.header();
44     ja.footer();
45     ja.left();
46     ja.content();
47     cout << "-----" << endl;
```

```
48
49     //Python页面
50     cout << "Python下载视频页面如下: " << endl;
51     Python py;
52     py.header();
53     py.footer();
54     py.left();
55     py.content();
56     cout << "-----" << endl;
57
58     //C++页面
59     cout << "C++下载视频页面如下: " << endl;
60     CPP cp;
61     cp.header();
62     cp.footer();
63     cp.left();
64     cp.content();
65 }
66
67 int main() {
68     test01();
69     return 0;
70 }
```

继承方式

继承的语法: `class 子类 : 继承方式 父类`

方式:

- 公共继承
- 保护继承
- 私有继承

三个控制修饰访问符:

- `public` : 无限制，任何代码均可访问。
- `private` : 仅类内部和友元可访问，派生类不可访问。
- `protected` : 介于两者之间，允许派生类访问，但阻止外部直接访问。

继承方式	基类(A)	派生类(B)
<code>public</code>	<code>public</code> 成员 → <code>public</code> 成员 <code>protected</code> 成员 → <code>protected</code> 成员 <code>private</code> 成员 → 不可见	
<code>private</code>	<code>public</code> 成员 → <code>private</code> 成员 <code>protected</code> 成员 → <code>private</code> 成员 <code>private</code> 成员 → 不可见	
<code>protected</code>	<code>public</code> 成员 → <code>protected</code> 成员 <code>protected</code> 成员 → <code>protected</code> 成员 <code>private</code> 成员 → 不可见	

总结：

基类的`private`成员在派生类中都不可见；

继承方式`public`不改变成员类型，`private`和`protected`都会使成员变成继承方式。

示例：

```
1 -> class Base1 {
2     public:
3         int m_A;
4     protected:
5         int m_B;
6     private:
7         int m_C;
8 };
9
10 //公共继承
11 -> class Son1 :public Base1 {
12     public:
13     void func() {
14         m_A; //可访问 public权限
15         m_B; //可访问 protected权限
16         //m_C; //不可访问
17     }
18 };
19
20 -> void myClass() {
21     Son1 s1;
22     s1.m_A; //其他类只能访问到公共权限
23 }
24
25 //保护继承
26 -> class Base2 {
27     public:
28         int m_A;
29     protected:
30         int m_B;
31     private:
32         int m_C;
33 };
34 -> class Son2:protected Base2 {
35     public:
36     void func() {
37         m_A; //可访问 protected权限
38         m_B; //可访问 protected权限
39         //m_C; //不可访问
40     }
41 };
42 -> void myClass2() {
43     Son2 s;
44     //s.m_A; //不可访问
45 }
46
47 //私有继承
```

```
48  class Base3 {
49      public:
50          int m_A;
51      protected:
52          int m_B;
53      private:
54          int m_C;
55  };
56  class Son3:private Base3 {
57  public:
58      void func() {
59          m_A; //可访问 private权限
60          m_B; //可访问 private权限
61          //m_C; //不可访问
62      }
63  };
64  class GrandSon3 :public Son3 {
65  public:
66      void func() {
67          //Son3是私有继承, 所以继承Son3的属性在GrandSon3中都无法访问到
68          //m_A;
69          //m_B;
70          //m_C;
71      }
72  };
```

对象模型

问题：从父类继承过来的成员，哪些属于子类对象中？

父类中私有成员private也是被子类继承下去了，只是由编译器给隐藏后访问不到

示例：

```
1 -> class Base {
2     public:
3         int m_A;
4     protected:
5         int m_B;
6     private:
7         int m_C; //私有成员只是被隐藏了，但是还是会继承下去
8 };
9
10 //公共继承
11 -> class Son :public Base {
12     public:
13         int m_D;
14 };
15
16 -> void test01() {
17     cout << "sizeof Son = " << sizeof(Son) << endl;
18     // 输出: 16
19 }
20
21 -> int main() {
22     test01();
23     return 0;
24 }
```

构造和析构顺序

子类继承父类后，当创建子类对象，也会调用父类的构造函数。

顺序：先调用父类构造函数，再调用子类构造函数，析构顺序与构造相反

示例：

```
1 -> class Base {
2     public:
3 ->         Base() {
4             cout << "Base构造函数!" << endl;
5         }
6 ->         ~Base() {
7             cout << "Base析构函数!" << endl;
8         }
9     };
10
11 -> class Son : public Base {
12     public:
13 ->         Son() {
14             cout << "Son构造函数!" << endl;
15         }
16 ->         ~Son() {
17             cout << "Son析构函数!" << endl;
18         }
19
20     };
21
22
23 -> void test01() {
24     Son s;
25 }
26
27 -> int main() {
28     test01();
29     //输出: Base构造函数!
30     //      Son构造函数!
31     //      Son析构函数!
32     //      Base析构函数!
33     return 0;
34 }
```

继承同名（静态）成员处理方式

问题：当子类与父类出现同名的（静态）成员，如何通过子类对象，访问到子类或父类中同名的数据呢？

- 访问子类同名（静态）成员 直接访问即可
- 访问父类同名（静态）成员 需要加作用域

注意：小心不同函数定义带来的运行错误。

示例：

```

1 #include<iostream>
2 using namespace std;
3 class Base {
4     public:
5         Base():m_A(100){}
6     void func() {
7         cout << "Base - func()调用" << endl;
8     }
9     void func(int a) {
10        cout << "Base - func(int a)调用" << endl;
11    }
12    public:
13        int m_A;
14    };
15
16
17 class Son : public Base {
18     public:
19         Son():m_A(100){}
20 //当子类与父类拥有同名的成员函数，子类会隐藏父类中所有版本的同名成员函数
21 //如果想访问父类中被隐藏的同名成员函数，需要加父类的作用域
22     void func() {
23         cout << "Son - func()调用" << endl;
24     }
25     public:
26         int m_A;
27    };
28
29 void test01() {
30     Son s;
31     cout << "Son下的m_A = " << s.m_A << endl;
32 //输出: Son下的m_A = 200
33     cout << "Base下的m_A = " << s.Base::m_A << endl; //这里就是调用的区别
34 //输出: Base下的m_A = 100
35     cout << "再次回到Son下的m_A = "<<s.m_A << endl;
36 //输出: 再次回到Son下的m_A = 200
37 //解释: 子类和父类的同名成员变量是独立的，不会相互影响，因为子类和父类属于不同的类作用域
38     s.func();
39 //输出: Son - func()调用
40     s.Base::func();
41 //输出: Base - func()调用
42     s.Base::func(10);
43 //输出: Base - func(int a)调用
44    }
45 int main() {
46     test01();

```

```
47     return 0;  
48 }
```

多继承语法

允许一个类继承多个类，即有多个父类。

语法： class 子类 : 继承方式 父类1 , 继承方式 父类2...

多继承可能会引发父类中有同名成员出现，需要加作用域区分

示例：

```

1 #include<iostream>
2 using namespace std;
3 - class Base1 {
4     public:
5         Base1():m_A(100) {}
6     public:
7         int m_A;
8 };
9
10 - class Base2 {
11     public:
12         Base2():m_A(200) {}
13         //开始是m_B 不会出问题，但是改为mA就会出现不明确
14     public:
15         int m_A;
16 };
17
18 //语法: class 子类: 继承方式 父类1 , 继承方式 父类2
19 - class Son : public Base2, public Base1 {
20     public:
21         Son():m_C(300),m_D(400) {}
22     public:
23         int m_C;
24         int m_D;
25 };
26
27
28 //多继承容易产生成员同名的情况
29 //通过使用类名作用域可以区分调用哪一个基类的成员
30 - void test01() {
31     Son s;
32     cout << "sizeof Son = " << sizeof(s) << endl;//输出: 16
33     cout << s.Base1::m_A << endl;
34     cout << s.Base2::m_A << endl;
35 }
36
37 - int main() {
38     test01();
39     return 0;
40 }

```

菱形继承

菱形继承概念：

两个派生类继承同一个基类

又有某个类同时继承者两个派生类

这种继承被称为菱形继承，或者钻石继承

菱形继承问题：

1. 羊继承了动物的数据，驼同样继承了动物的数据，当草泥马使用数据时，就会产生二义性。
2. 草泥马继承自动物的数据继承了两份，其实我们应该清楚，这份数据我们只需要一份就可以。

示例：

```
1 #include<iostream>
2 using namespace std;
3 class Animal {
4     public:
5         int m_Age;
6 };
7
8 //继承前加virtual关键字后，变为虚继承
9 //此时公共的父类Animal称为虚基类
10 class Sheep : virtual public Animal {};
11 class Tuo : virtual public Animal {};
12 class SheepTuo : public Sheep, public Tuo {};
13
14 void test01() {
15     SheepTuo st;
16     st.Sheep::m_Age = 100;
17     st.Tuo::m_Age = 200;
18
19     cout << "st.Sheep::m_Age = " << st.Sheep::m_Age << endl;
20     cout << "st.Tuo::m_Age = " << st.Tuo::m_Age << endl;
21     cout << "st.m_Age = " << st.m_Age << endl;
22 }
23
24 int main() {
25     test01();
26     return 0;
27 }
```

总结：

- 菱形继承带来的主要问题是子类继承两份相同的数据，导致资源浪费以及毫无意义
- 利用虚继承可以解决菱形继承问题

多态

基本概念

多态分为两类

- **静态多态**: 函数重载 和 运算符重载属于静态多态，复用函数名
- **动态多态**: 派生类和虚函数实现运行时多态

静态多态和动态多态区别：

- 静态多态的**函数地址早绑定** – 编译阶段确定函数地址
- 动态多态的**函数地址晚绑定** – 运行阶段确定函数地址

多态使用条件

- 父类指针或引用指向子类对象

覆盖 (override) : 函数返回值类型 函数名 参数列表 一致称为覆盖

```
1 -> class Animal {
2     public:
3     //Speak函数就是虚函数
4     //函数前面加上virtual关键字，变成虚函数，那么编译器在编译的时候就不能确定函数调用了。
5 ->     virtual void speak() {
6         cout << "动物在说话" << endl;
7     }
8 };
9
10-> class Cat :public Animal {
11     public:
12     void speak() {
13         cout << "小猫在说话" << endl;
14     }
15 };
16-> class Dog :public Animal {
17     public:
18     void speak() {
19         cout << "小狗在说话" << endl;
20     }
21 };
22 //我们希望传入什么对象，那么就调用什么对象的函数
23 //如果函数地址在编译阶段就能确定，那么静态联编
24 //如果函数地址在运行阶段才能确定，就是动态联编
25
26-> void DoSpeak(Animal & animal) {
27     animal.speak();
28 }
29 //
30 //多态满足条件：
31 //1、有继承关系
32 //2、子类重写父类中的虚函数
33 //多态使用：
34 //父类指针或引用指向子类对象
35
36-> void test01() {
37     Cat cat;
38     DoSpeak(cat);
39     Dog dog;
40     DoSpeak(dog);
41 }
42
43-> int main() {
44     test01();
45     return 0;
46 }
```

静态类型 vs. 动态类型

- 静态类型：指针/引用的声明类型（如 `A*`），相当于“标签”，非虚函数只看“标签”。
- 动态类型：实际指向的对象类型（如 `B`），相当于“实际内容”，虚函数看“实际内容”。
- 非虚函数由静态绑定，虚函数由动态绑定。

示例：

```

1 #include <iostream>
2 using namespace std;
3 class A {
4 public:
5     // 非虚函数: 静态绑定, 调用路径由指针/引用的声明类型决定
6     void testfuc() {
7         cout << "A::testfuc ";    // [Step1] 静态类型为A, 直接调用A的testfuc
8         func();                // [Step2] 非虚函数func, 继续静态绑定到A::func
9     }
10
11    // 非虚函数: 派生类B中的func会隐藏此函数, 但通过A类型调用时仍选此版本
12    void func() {
13        cout << "A::func ";      // [Step3] 静态绑定到A::func
14        vfunc();                // [Step4] 虚函数vfunc, 动态绑定到实际对象类型
15    }
16
17    // 虚函数: 动态绑定, 实际调用由对象类型决定
18    virtual void vfunc() {
19        cout << "A::vf" << endl;
20    }
21 };
22
23 class B : public A {
24 public:
25     // 非虚函数: 隐藏A::func (非重写), 仅通过B类型调用时生效
26     void func() {
27         cout << "B::func" << endl;
28     }
29
30     // 虚函数重写: 覆盖A::vfunc, 通过基类指针调用时动态绑定到此版本
31     virtual void vfunc() override {
32         cout << "B::vf" << endl;
33     }
34 };
35
36 int main() {
37     A a;
38     a.func();cout<<endl;
39     // 输出: A::func A::vf
40
41     B b;
42     b.func();
43     // 输出: B::func
44     b.testfuc();cout<<endl; // 从A继承的非虚函数testfuc → A::testfuc → A::fu
nc → 动态绑定B::vfunc
45     // 输出: A::testfuc A::func B::vf
46

```

```
47 A* p = &b;
48 p->vfunc(); // 虚函数 → 动态绑定B::vfunc
49 // 输出: B::vf
50
51 p->testfuc(); // 非虚函数A::testfuc → A::func → 动态绑定B::vfunc
52 // 输出: A::testfuc A::func B::vf
53
54 p->func(); // 非虚函数 → 静态绑定A::func → 动态绑定B::vfunc
55 // 输出: A::func B::vf
56 return 0;
57 }
```

总结:

非虚函数看“标签”，回基类；

虚函数看“实际内容”，在派生类。

习题2：

```
1 #include <iostream>
2 using namespace std;
3
4 class Base {
5 public:
6 void foo() {
7     cout << "Base::foo()\n";
8     bar();
9     baz();
10 }
11
12 virtual void bar() {
13     cout << "Base::bar()\n";
14 }
15
16 void baz() {
17     cout << "Base::baz()\n";
18 }
19 };
20
21 class Derived : public Base {
22 public:
23 void foo() {
24     cout << "Derived::foo()\n";
25     bar();
26     Base::baz();
27 }
28
29 virtual void bar() override {
30     cout << "Derived::bar()\n";
31 }
32
33 void baz() {
34     cout << "Derived::baz()\n";
35 }
36 };
37
38 class DeepDerived : public Derived {
39 public:
40 void foo() {
41     cout << "DeepDerived::foo()\n";
42     bar();
43     Derived::bar();
44 }
45
46 void bar() override {
47     cout << "DeepDerived::bar()\n";
```

```
48     }
49     virtual void baz() {
50         cout << "DeepDerived::baz()\n";
51     }
52 };
53
54 int main() {
55     DeepDerived dd;
56     Base* p1= &dd;
57     Derived* p2 = &dd;
58
59     cout << "==== Test 1 ===" << endl;
60     p1->foo();
61
62     cout << "\n==== Test 2 ===" << endl;
63     p2->foo();
64
65     cout << "\n==== Test 3 ===" << endl;
66     dd.foo();
67
68     cout << "\n==== Test 4 ===" << endl;
69     p1->baz();
70     p2->baz();
71     dd.baz();
72
73     return 0;
74 }
```

输出：

==== Test 1 ===

Base::foo()

DeepDerived::bar()

Base::baz()

==== Test 2 ===

Derived::foo()

DeepDerived::bar()

Base::baz()

==== Test 3 ====

DeepDerived::foo()

DeepDerived::bar()

Derived::bar()

==== Test 4 ====

Base::baz()

Derived::baz()

DeepDerived::baz()

总结的总结：

1. 明确对象的“标签”和“实际内容”，判断对象是不是纯粹的（标签与实际一致）
2. 回到“标签”的基类看看这个函数是否为（非）虚函数，虚函数用实际内容对应的域，非虚函数用标签对应的域。

对象切片

对象切片发生在将派生类对象赋值给基类对象时，导致派生类特有的成员数据丢失。这是因为基类对象仅能存储基类部分的成员，而派生类新增的成员会被“切掉”。

示例：

```

1 -> class Base {
2     public:
3         int a;
4     };
5
6 -> class Derived : public Base {
7     public:
8         int b; // 派生类新增成员
9     };
10
11     Derived d;
12     Base b = d; // 对象切片: b 仅保留 Base::a, Derived::b 被丢弃

```

关键点：

- **原因：**直接对对象进行值传递或赋值时，编译器仅复制基类部分。
- **解决方法：**使用基类的指针或引用，保持多态性：

```

1     Base& ref = d;      // 通过引用访问，避免切片
2     Base* ptr = &d;      // 通过指针访问，避免切片

```

纯虚函数和抽象类

在多态中，通常父类中虚函数的实现是毫无意义的，主要都是调用子类重写的内容，因此可以将虚函数改为纯虚函数。

语法： `virtual 返回值类型 函数名 (参数列表) = 0 ;`

当类中有了纯虚函数，这个类也称为抽象类。

抽象类特点：

- 无法实例化对象
- 子类必须覆盖抽象类中的纯虚函数，否则也属于抽象类

示例：

```

1 -> class Base {
2     public:
3         virtual void func() = 0;
4     };
5
6 -> class Son :public Base {
7     public:
8     virtual void func() {
9         cout << "func调用" << endl;
10    };
11 };
12
13 -> void test01() {
14     Base * base = NULL;
15     //base = new Base; // 错误, 抽象类无法实例化对象
16     base = new Son;
17     base->func();
18     delete base;//记得销毁
19 }
20
21 -> int main() {
22     test01();
23     return 0;
24 }

```

虚析构和纯虚析构

多态使用时, 如果子类中有属性开辟到堆区, 那么父类指针在释放时无法调用到子类的析构代码。

解决方式: 将父类中的析构函数改为**虚析构**或者**纯虚析构**

虚析构和纯虚析构共性:

- 可以解决父类指针释放子类对象
- 都需要有具体的函数实现

虚析构和纯虚析构区别:

- 如果是纯虚析构, 该类属于抽象类, 无法实例化对象

虚析构语法：

```
virtual ~类名(){};
```

纯虚析构语法：

```
virtual ~类名() = 0;
```

```
类名::~类名(){};
```

示例：

```

1 #include <iostream>
2 using namespace std;
3 class Animal {
4 public:
5     Animal() {
6         cout << "Animal 构造函数调用! " << endl;
7     }
8     virtual void Speak() = 0;
9     //析构函数加上virtual关键字，变成虚析构函数
10    //virtual ~Animal()
11    //{
12    //    cout << "Animal虚析构函数调用! " << endl;
13    //}
14    virtual ~Animal() = 0;
15 };
16
17 ~Animal() {
18     cout << "Animal 纯虚析构函数调用! " << endl;
19 }
20
21 //和包含普通纯虚函数的类一样，包含了纯虚析构函数的类也是一个抽象类。不能够被实例化。
22 class Cat : public Animal {
23 public:
24     Cat(string name) {
25         cout << "Cat构造函数调用! " << endl;
26         m_Name = new string(name);
27     }
28     virtual void Speak() {
29         cout << *m_Name << "小猫在说话!" << endl;
30     }
31 ~Cat() {
32     cout << "Cat析构函数调用!" << endl;
33     if (this->m_Name != NULL) {
34         delete m_Name;
35         m_Name = NULL;
36     }
37 }
38 public:
39     string *m_Name;
40 };
41
42 void test01() {
43     Animal *animal = new Cat("Tom");
44     animal->Speak();
45     //通过父类指针去释放，会导致子类对象可能清理不干净，造成内存泄漏
46     //怎么解决？给基类增加一个虚析构函数
47     //虚析构函数就是用来解决通过父类指针释放子类对象

```

```
48     delete animal;
49 }
50
51 int main() {
52     test01();
53     return 0;
54 }
```

总结：

1. 虚析构或纯虚析构就是用来解决通过父类指针释放子类对象
2. 如果子类中没有堆区数据，可以不写为虚析构或纯虚析构
3. 拥有纯虚析构函数的类也属于抽象类

类模版

代码示例：

```
1 #include <iostream>
2 #include <stdexcept>
3 #include <vector>
4 using namespace std;
5
6 template <typename T>
7 class Stack;
8
9 // 模版化的友元函数必须前向声明
10 template <typename T>
11 ostream& operator>>(ostream& os, Stack<T>& stack);
12
13 template <typename T, int MaxSize = 0>
14 class myStack {
15 private:
16     std::vector<T> elements;
17
18 public:
19     myStack() {
20         ++instance_count;
21     }
22
23     void push(T const& element);
24
25     T top() const {
26         if (empty()) {
27             throw std::out_of_range("Stack<T>::top(): empty stack");
28         }
29         return elements.back();
30     }
31
32     void pop() {
33         if (empty()) {
34             throw std::out_of_range("Stack<T>::pop(): empty stack");
35         }
36         elements.pop_back();
37     }
38
39     bool empty() const {
40         return elements.empty();
41     }
42
43     friend ostream& operator>> (ostream& os, myStack<T>& stack){
44         for (const auto& elem : stack.elements) {
45             os << elem << " ";
46         }
47         return os;
48     }
49 }
```

```

48     }
49
50     // 静态成员, 每个模板特化类拥有独立的静态成员。
51     static int instance_count;
52 };
53
54 // 外部成员函数定义, 注意形式 类名<T>::
55 template <typename T, int MaxSize>
56 void myStack<T, MaxSize>::push(T const& element) {
57     if constexpr (MaxSize > 0) {
58         if (elements.size() >= MaxSize) {
59             throw std::out_of_range("Stack is full");
60         }
61     }
62     elements.push_back(element);
63 }
64
65 // 初始化静态成员
66 template <typename T, int MaxSize>
67 int myStack<T, MaxSize>::instance_count = 0;
68
69 int main() {
70     myStack<int> intStack;    // 等价于 myStack<int, 0>, 使用了默认的
71     myStack<int> intStack2;  // 同种类还是一个静态成员
72     intStack.push(42);
73     intStack.push(7);
74
75     myStack<std::string, 3> strStack; // 最大容量3
76     try {
77         strStack.push("Hello");
78         strStack.push("Template");
79         strStack.push("World");
80         strStack.push("Extra"); // 这里会抛出异常
81     } catch (std::exception const& e) {
82         std::cout << "Exception: " << e.what() << std::endl;
83     }
84
85     myStack<double> anotherStack;
86     std::cout << myStack<double>::instance_count << " " << myStack<double>::
87     instance_count << std::endl;
88     // 输出: 2 1
89
90     return 0;
91 }
```

链表

```
1 #include <iostream>
2 using namespace std;
3
4 // 链表节点模板
5 template<typename T>
6 struct ListNode {
7     T data;
8     ListNode<T>* next;
9     ListNode(const T& val) : data(val), next(nullptr) {}
10 };
11
12 // 链表模板类
13 template<typename T>
14 class LinkedList {
15 private:
16     ListNode<T>* head;
17     ListNode<T>* tail;
18
19 public:
20     // 构造函数
21     LinkedList() : head(nullptr), tail(nullptr) {}
22
23     // 析构函数
24     ~LinkedList() {
25         ListNode<T>* current = head;
26         while (current) {
27             ListNode<T>* next = current->next;
28             delete current;
29             current = next;
30         }
31     }
32
33     // 尾部插入
34     void push_back(const T& value) {
35         ListNode<T>* newNode = new ListNode<T>(value);
36         if (!head) {
37             head = tail = newNode;
38         } else {
39             tail->next = newNode;
40             tail = newNode;
41         }
42     }
43
44     // 打印链表
45     void print() const {
46         ListNode<T>* current = head;
47         while (current) {
```

```

48         cout << current->data << " ";
49         current = current->next;
50     }
51     cout << endl;
52 }
53
54 // 获取头节点 (用于合并操作)
55 ListNode<T>* get_head() const { return head; }
56 };
57
58 // 合并两个有序链表 (非成员函数)
59 template<typename T>
60 void merge_sorted_lists(const LinkedList<T>& l1,
61                         const LinkedList<T>& l2,
62                         LinkedList<T>& result) {
63     ListNode<T>* p1 = l1.get_head();
64     ListNode<T>* p2 = l2.get_head();
65
66     while (p1 && p2) {
67         if (p1->data <= p2->data) {
68             result.push_back(p1->data);
69             p1 = p1->next;
70         } else {
71             result.push_back(p2->data);
72             p2 = p2->next;
73         }
74     }
75
76     // 添加剩余元素
77     while (p1) {
78         result.push_back(p1->data);
79         p1 = p1->next;
80     }
81     while (p2) {
82         result.push_back(p2->data);
83         p2 = p2->next;
84     }
85 }
86
87 // 使用示例
88 int main() {
89     LinkedList<int> list1, list2, merged;
90
91     // 创建有序链表
92     list1.push_back(1);
93     list1.push_back(3);
94     list1.push_back(5);
95 }
```

```
96     list2.push_back(2);
97     list2.push_back(4);
98     list2.push_back(6);
99
100    // 合并链表
101    merge_sorted_lists(list1, list2, merged);
102
103    cout << "Merged list: ";
104    merged.print(); // 输出: 1 2 3 4 5 6
105
106    return 0;
107 }
```

零散题目

题目1：构造/析构顺序与虚函数

```

1 -> class Base {
2     public:
3         Base() { callVirtual(); }
4         virtual ~Base() { cout << "~Base "; }
5         virtual void callVirtual() { cout << "Base "; }
6     };
7
8 -> class Derived : public Base {
9     public:
10        Derived() { callVirtual(); }
11        ~Derived() { cout << "~Derived "; }
12        void callVirtual() override { cout << "Derived "; }
13    };
14
15 Derived d; // 输出什么?

```

答案与考察点

输出: Base Derived ~Derived ~Base

隐蔽点: 构造函数中虚函数机制未生效 (此时对象类型仍视为 Base)

题目2: 继承中的对象切片

```

1 -> class Animal {
2     public:
3         virtual void speak() { cout << "Animal "; }
4     };
5
6 -> class Cat : public Animal {
7     public:
8         void speak() override { cout << "Cat "; }
9         void groom() { cout << "Grooming "; }
10    };
11
12 -> void process(Animal a) {
13     a.speak();
14 }
15
16 Cat c;
17 process(c); // 输出什么? 能否调用c.groom()?

```

答案与考察点

输出: Animal

隐蔽点: 值传递导致对象切片, 丢失派生类信息; groom()不可访问

题目3: 成员初始化顺序

```
1 class Clock {  
2     int hours = initHours();  
3     int minutes = 30;  
4 public:  
5     Clock() : minutes(0) {}  
6     int initHours() { return minutes + 1; }  
7 };
```

Clock 对象构造后, hours和minutes的值分别是多少?

答案与考察点

hours=1, minutes=0

隐蔽点: 初始化顺序按声明顺序 (先hours后minutes), 覆盖初始化列表

题目4: 虚函数表与内存布局

```
1 class Empty {};  
2 class WithVirtual { virtual void f() {} };
```

sizeof(Empty) 和 sizeof(WithVirtual) 的值为? 若继承多个含虚函数的类, 大小如何变化?

答案与考察点

1 (空类)、8 (64位指针)

隐蔽点: 虚函数表指针的存在; 多重继承可能增加多个vptr

题目5: 静态绑定与默认参数

```

1 -> class Shape {
2     public:
3 ->     virtual void draw(int thickness=1) {
4         cout << "Shape:" << thickness << " ";
5     }
6 };
7
8 -> class Circle : public Shape {
9     public:
10    void draw(int thickness=5) override {
11        cout << "Circle:" << thickness << " ";
12    }
13 };
14
15 Shape* p = new Circle();
16 p->draw(); // 输出什么?

```

答案与考察点

Circle:1

隐蔽点：默认参数静态绑定（根据 `Shape` 类型），函数动态绑定

题目6：移动语义陷阱

```

1 -> class Buffer {
2     int* data;
3     public:
4     Buffer() : data(new int[1024]) {}
5 ->     Buffer(Buffer&& other) : data(other.data) {
6         other.data = nullptr;
7     }
8     ~Buffer() { delete[] data; }
9 };
10
11 -> Buffer createBuffer() {
12     Buffer local;
13     return local;
14 }
15
16 Buffer b = createBuffer(); // 调用几次构造函数?

```

答案与考察点

1次 (RVO优化)

隐蔽点：移动构造可能被编译器优化完全跳过

题目7：类型转换运算符

```
1 class Meter {
2     double value;
3 public:
4     explicit operator int() { return value; }
5     operator double() { return value; }
6 };
7
8 Meter m{2.5};
9 double d = m;
10 int i = m;    // 哪行编译错误?
```

答案与考察点

`int i = m` 错误

隐蔽点：`explicit`转换必须显式调用（如 `static_cast<int>(m)`）

题目8：模板类的静态成员

```
1 template<typename T>
2 class Counter {
3 public:
4     static int count;
5     Counter() { count++; }
6 };
7 template<> int Counter<int>::count = 0;
8
9 Counter<int> a, b;
10 Counter<double> c;
```

`Counter<int>::count` 和 `Counter<double>::count` 的值分别为？

答案与考察点

2 和 1

隐蔽点：模板特化导致不同模板参数的静态成员独立存在

题目9：析构函数多态

```
1 -> class Base {
2     public:
3         ~Base() { cout << "Base "; }
4     };
5
6 -> class Derived : public Base {
7     public:
8         ~Derived() { cout << "Derived "; }
9     };
10
11 Base* p = new Derived();
12 delete p; // 输出什么？如何修正？
```

答案与考察点

输出 `Base` (资源泄漏)

隐蔽点：基类析构函数未声明为virtual导致派生类析构未调用

题目10：const成员函数

```
1 -> class Data {
2     mutable int accessCount = 0;
3     int value = 42;
4     public:
5 ->     void modify() const {
6         accessCount++;
7         value = 0;      // 是否合法？
8     }
9 };
```

答案与考察点

`accessCount++` 合法 (`mutable`修饰) , `value=0` 不合法

隐蔽点：const成员函数对`mutable`成员的可修改性

文件操作

程序运行时产生的数据都属于临时数据，程序一旦运行结束都会被释放，通过文件可以将数据持久化，对文件操作需要包含头文件 `< fstream >`。

文件类型分为两种：

1. 文本文件 – 文件以文本的ASCII码形式存储在计算机中
2. 二进制文件 – 文件以文本的二进制形式存储在计算机中，用户一般不能直接读懂它们

操作文件的三大类：

1. ofstream：写操作
2. ifstream：读操作
3. fstream：读写操作

文本文件

写文件

步骤：

1. 包含头文件 `#include <fstream>`
2. 创建流对象 `ofstream ofs;`

3. 打开文件ofs.open("文件路径",打开方式);
4. 写数据ofs << "写入的数据";
5. 关闭文件ofs.close();

文件打开方式：

打开方式	解释
ios::in	为读文件而打开文件
ios::out	为写文件而打开文件
ios::ate	初始位置：文件尾
ios::app	追加方式写文件
ios::trunc	如果文件存在先删除，再创建
ios::binary	二进制方式

注意：文件打开方式可以配合使用，利用|操作符

例如：用二进制方式写文件 `ios::binary | ios:: out`

示例：

```

1 #include <iostream>
2
3 void test01(){
4     ofstream ofs;
5     ofs.open("test.txt", ios::out);
6
7     ofs << "姓名: 张三" << endl;
8     ofs << "性别: 男" << endl;
9     ofs << "年龄: 18" << endl;
10
11    ofs.close();
12 }
13
14 int main() {
15     test01();
16     return 0;
17 }
```

读文件

读文件与写文件步骤相似，但是读取方式相对于比较多

步骤：

1. 包含头文件 #include <fstream>
2. 创建流对象 ifstream ifs;
3. 打开文件并判断文件是否打开成功ifs.open("文件路径",打开方式);
4. 读数据四种方式读取
5. 关闭文件ifs.close();

示例：

```

1 #include <fstream>
2 #include <string>
3 void test01(){
4     ifstream ifs;
5     ifs.open("test.txt", ios::in);
6
7     if (!ifs.is_open()){
8         cout << "文件打开失败" << endl;
9         return;
10    }
11
12    //第一种方式
13    char buf[1024] = {  };
14    while (ifs >> buf){
15        cout << buf << endl;
16    }
17
18    //第二种
19    char buf[1024] = { 0 };
20    while (ifs.getline(buf,sizeof(buf))){
21        cout << buf << endl;
22    }
23
24    //第三种
25    string buf;
26    while (getline(ifs, buf)){
27        cout << buf << endl;
28    }
29
30    //第四种
31    char c;
32    while ((c = ifs.get()) != EOF){
33        cout << c;
34    }
35    ifs.close();
36 }
37
38 int main() {
39     test01();
40     return 0;
41 }

```

总结：

- 读文件可以利用 ifstream，或者fstream类
- 利用is_open函数可以判断文件是否打开成功

- close 关闭文件

二进制文件

以二进制的方式对文件进行读写操作， 打开方式要指定为 `ios::binary`。

写文件

二进制方式写文件主要利用流对象调用成员函数`write`。

函数原型 : `ostream& write(const char * buffer, int len);`

参数解释：字符指针buffer指向内存中一段存储空间。len是读写的字节数。

示例：

```
1 #include <fstream>
2 #include <string>
3 using namespace std;
4
5 - class Person{
6     public:
7         char m_Name[64];
8         int m_Age;
9     };
10
11 //二进制文件 写文件
12 - void test01(){
13     //1、包含头文件
14
15     //2、创建输出流对象
16     ofstream ofs("person.txt", ios::out | ios::binary);
17
18     //3、打开文件
19     //ofs.open("person.txt", ios::out | ios::binary);
20
21     Person p = {"张三" , 18};
22
23     //4、写文件
24     ofs.write((const char *)&p, sizeof(p));
25
26     //5、关闭文件
27     ofs.close();
28 }
29
30 - int main() {
31     test01();
32     return 0;
33 }
```

读文件

二进制方式读文件主要利用流对象调用成员函数**read**。

函数原型: `istream& read(char *buffer, int len);`

参数解释：字符指针buffer指向内存中一段存储空间。len是读写的字节数。

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Person{
6     public:
7         char m_Name[64];
8         int m_Age;
9     };
10
11 void test01(){
12     ifstream ifs("person.txt", ios::in | ios::binary);
13     if (!ifs.is_open()){
14         cout << "文件打开失败" << endl;
15     }
16
17     Person p;
18     ifs.read((char *)&p, sizeof(p));
19
20     cout << "姓名: " << p.m_Name << " 年龄: " << p.m_Age << endl;
21 }
22
23 int main() {
24     test01();
25     return 0;
26 }
```

文件和流 (Files and Streams)

流 (Stream) : 字节序列，用于数据传输。

- 输入流 (Input Stream) : 数据从外部设备流向内存 (如键盘、文件读取)。

- 输出流（Output Stream）：数据从内存流向外部设备（如屏幕、文件写入）。

C++文件流类：

- 头文件：`#include <fstream>`。
- 类模板特化：
 - `ifstream`：文件输入流（读文件）。
 - `ofstream`：文件输出流（写文件）。
 - `fstream`：文件输入输出流（支持读写）。

流继承关系：

`ios` → `istream/ostream` → `iostream`

→ `ifstream/ofstream` → `fstream`

标准流对象：

`cin`（输入）、`cout`（输出）、`cerr`（错误输出）。

创建顺序文件

步骤

1. 定义文件流对象：

```
1  ofstream outClientFile;
```

2. 打开文件：

方式一：构造函数直接打开。

```
1  ofstream outClientFile("clients.dat", ios::out);
```

方式二：先定义对象，后调用 `open` 函数。

```
1  outClientFile.open("clients.dat", ios::out);
```

3. 文件模式（Mode）：

`ios::out`：默认模式，若文件存在则清空内容，若不存在则创建。

`ios::app` : 在文件末尾追加数据。

4. 写入数据 :

```
1 outClientFile << account << ' ' << name << ' ' << balance << endl;
```

需用空格分隔数据（便于后续读取）。

5. 关闭文件 :

```
1 outClientFile.close();
```

析构函数自动关闭文件，但建议显式调用 `close()`。

从顺序文件读取数据

步骤：

1. 定义文件流对象 :

```
1 ifstream inClientFile;
```

2. 打开文件 :

```
1 ifstream inClientFile("clients.dat", ios::in);
```

3. 读取数据 :

```
1 inClientFile >> account >> name >> balance;
```

需按写入时的格式读取（空格分隔）。

4. 判断文件结束（使用 `eof()` 函数）:

```
1 while (!inClientFile.eof()) { // 处理数据}
```

注意事项：文件末尾需有空行，否则可能忽略最后一行数据。

5. 重新定位文件指针 :

`seekg()` : 移动输入流的“get指针”。

`seekp()` : 移动输出流的“put指针”。

`tellg()` : 获取当前“get指针”位置。

`tellg()` : 获取当前“put指针”位置。

```
1  inClientFile.seekg(0); // 回到文件开头  
2  inClientFile.clear(); // 清除EOF标志
```

6. 文件指针偏移方式 :

`ios::beg` : 从文件开头偏移。

`ios::cur` : 从当前位置偏移。

`ios::end` : 从文件末尾偏移。

代码示例 (若没有文件, 创建文件)

```

1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     string filename;
7     cout << "输入文件名 (如: file.dat): ";
8     cin >> filename;
9
10    // 尝试以读写模式打开文件, 若不存在则创建
11    fstream file(filename, ios::in | ios::out | ios::binary);
12    if (!file) {
13        ofstream createFile(filename, ios::binary);
14        if (!createFile) {
15            cerr << "无法创建文件!" << endl;
16            return 1;
17        }
18        createFile.close();
19        file.open(filename, ios::in | ios::out | ios::binary);
20    }
21
22    // 连续写入三个字符, 指针自动移动
23    char data[] = {'a', 'b', 'c'};
24    file.write(data, sizeof(data)); // 写入后指针在位置3
25    file.seekp(0); // 确保写入后指针重置
26    file.seekg(0); // 重置指针到开头以便读取
27
28    // 阶段1: 读取并修改数据
29    cout << "----- 修改阶段 -----" << endl;
30    char ch;
31    while (file.read(&ch, sizeof(ch))) {
32        // 显示读取信息
33        cout << "读取字符: " << ch << " | 指针位置: " << file.tellg() << endl;
34
35        // 修改数据
36        ch += 1;
37        file.seekp(-1, ios::cur); // 回退写指针
38        file.write(&ch, 1); // 写入修改后的字符
39        file.seekg(file.tellp()); // 同步读指针到当前写位置
40    }
41
42    // 阶段2: 重新读取验证
43    cout << "\n----- 验证阶段 -----" << endl;
44    file.clear(); // 清除eof状态
45    file.seekg(0); // 重置到文件开头
46
47    while (file.read(&ch, 1)) {

```

```
48     cout << "最终字符: " << ch << " | 指针位置: " << file.tellg() << endl;
49 }
50
51     file.close();
52
53 }
```

随机存取文件

特点：

1. 支持直接访问文件中任意记录（无需顺序读取）。
2. 需固定记录大小（二进制模式写入）。

实现步骤：

1. 定义数据结构：

```
1 class ClientData {
2 public:
3     int accountNumber;
4     char name[15]; // 固定长度字段
5     double balance;
6 };
```

2. 写入二进制文件：

```
1 ofstream outCredit("credit.dat", ios::out | ios::binary);
2 ClientData client = {100, "yang", 99.9};
3 outCredit.write(reinterpret_cast<const char*>(&client), sizeof(ClientData))
;
```

3. 读取二进制文件：

```
1 ifstream inCredit("credit.dat", ios::in | ios::binary);
2 ClientData client;
3 inCredit.read(reinterpret_cast<char*>(&client), sizeof(ClientData));
```

4. 定位特定记录：

读取第 N 条记录：

```
1    cpp1inCredit.seekg(sizeof(ClientData) * (N - 1));
```

写入第 N 条记录：

```
1    fstream file("credit.dat", ios::in | ios::out | ios::binary);
2    file.seekp(sizeof(ClientData) * (N - 1));
```

综合示例

```

1 #include "filehandler.h"
2 #include <fstream>
3 #include <iostream>
4 #include <iomanip>
5 #include <string>
6
7 - static int findRecordPosition(std::fstream& file, int targetId) {
8     file.clear();
9     file.seekg(0);
10    Student s ;
11    int position = 0;
12 -   while (file.read(reinterpret_cast<char*>(&s), sizeof(Student))) {
13 -       if (s.getId() == targetId) {
14           return position;
15       }
16       position++;
17   }
18   return -1;
19 }
20
21 - static bool idExists(std::fstream& file, int id) {
22     file.clear();
23     file.seekg(0);
24
25     Student s ;
26 -   while (file.read(reinterpret_cast<char*>(&s), sizeof(Student))) {
27 -       if (s.getId() == id && s.getId() != 0) {
28           return true;
29       }
30   }
31   return false;
32 }
33
34 - void displayRecords(std::fstream& file) {
35     file.clear();
36     file.seekg(0);
37     Student s ;
38     int count = 0;
39     double total = 0.0;
40
41 -   while (file.read(reinterpret_cast<char*>(&s), sizeof(Student))) {
42 -       if (s.getId() != 0) {
43           std::cout << "ID: " << s.getId() << "\n"
44               << "姓名: " << s.getFirstName() << " " << s.getLastName()
45               << "\n"
46               << "成绩: " << std::fixed << std::setprecision(1) << s.getGrade() << "\n\n";

```

```

46             total += s.getGrade();
47         count++;
48     }
49 }
50
51 if (count > 0) {
52     std::cout << "平均成绩: " << (total / count) << "\n";
53 }
54 else {
55     std::cout << "暂无记录.\n";
56 }
57 }
58
59 void addRecord(std::fstream& file) {
60     file.clear();
61     file.seekg(0);
62     int id;
63     std::string fname, lname;
64     double grade;
65
66     std::cout << "输入学生ID: ";
67     std::cin >> id;
68     std::cin.ignore();
69
70     std::cout << "输入姓氏: ";
71     std::getline(std::cin, fname);
72     std::cout << "输入名字: ";
73     std::getline(std::cin, lname);
74     std::cout << "输入成绩: ";
75     std::cin >> grade;
76
77     Student temp=Student(0," "," ",0);
78     int position = -1;
79     file.seekg(0);
80     while (file.read(reinterpret_cast<char*>(&temp), sizeof(Student))) {
81         if (temp.getId() == 0) {
82             position = static_cast<int>(file.tellg()) / sizeof(Student) -
83             1;
84             break;
85         }
86     }
87     Student newStudent(id, fname, lname, grade);
88     if (position != -1) {
89         file.seekp(position * sizeof(Student));
90     }
91 else {
92     file.clear();

```

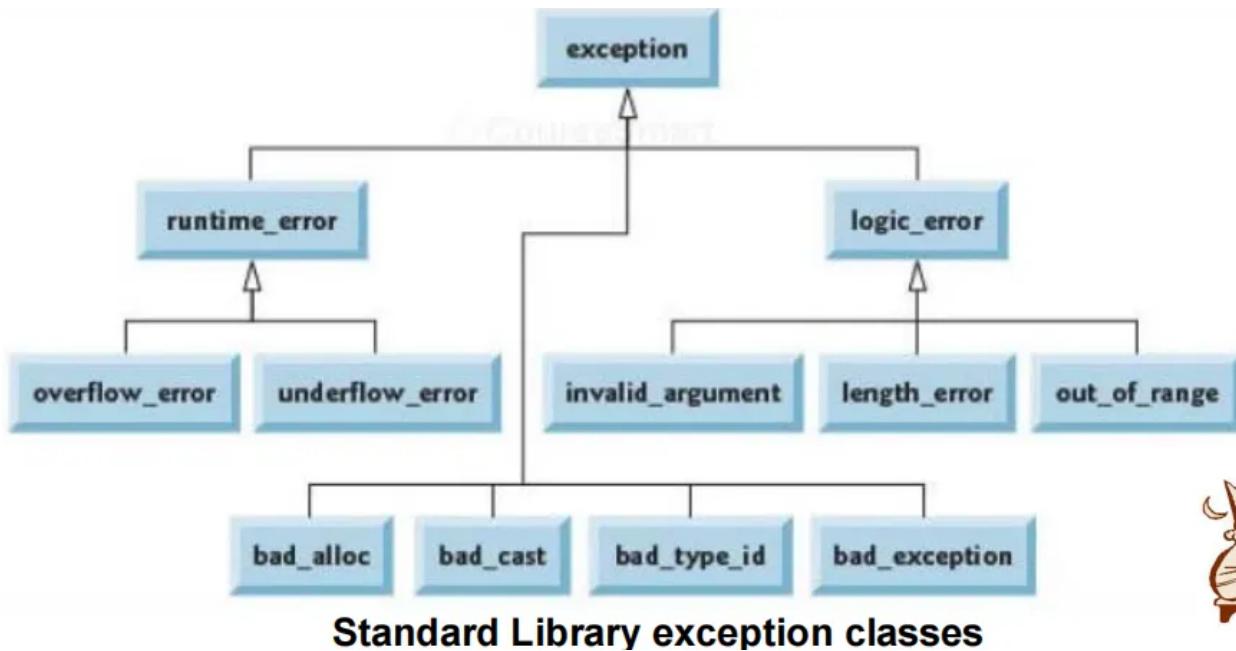
```

93         file.seekp(0, std::ios::end);
94     }
95
96     file.write(reinterpret_cast<const char*>(&newStudent), sizeof(Student));
97     std::cout << "输入完毕.\n";
98 }
99
100 void deleteRecord(std::fstream& file) {
101     file.clear();
102     file.seekg(0);
103     int targetId;
104     std::cout << "输入要删除的ID: ";
105     std::cin >> targetId;
106
107     int pos = findRecordPosition(file, targetId);
108     if (pos == -1) {
109         std::cout << "无法找到该ID.\n";
110         return;
111     }
112
113     Student blank;
114     file.seekp(pos * sizeof(Student));
115     file.write(reinterpret_cast<const char*>(&blank), sizeof(Student));
116     std::cout << "记录已删除.\n";
117 }
118
119 void modifyRecord(std::fstream& file) {
120     file.clear();
121     file.seekg(0);
122     int targetId;
123     std::cout << "输入需要修改的ID: ";
124     std::cin >> targetId;
125
126     int pos = findRecordPosition(file, targetId);
127     if (pos == -1) {
128         std::cout << "无法找到该ID.\n";
129         return;
130     }
131
132     Student s ;
133     file.seekg(pos * sizeof(Student));
134     file.read(reinterpret_cast<char*>(&s), sizeof(Student));
135
136     int newId;
137     std::string fname, lname;
138     float grade;
139

```

```
140     std::cout << "输入新ID ( " << s.getId() << "): ";
141     std::cin >> newId;
142     if (newId != targetId) {
143         std::cout << "错误: 该ID已存在! \n";
144         return;
145     }
146
147     std::cin.ignore();
148     std::cout << "输入姓氏 ( " << s.getFirstName() << "): ";
149     std::getline(std::cin, fname);
150     std::cout << "输入名称 ( " << s.getLastName() << "): ";
151     std::getline(std::cin, lname);
152     std::cout << "输入成绩 ( " << s.getGrade() << "): ";
153     std::cin >> grade;
154
155     s.setId(newId);
156     s.setFirstName(fname);
157     s.setLastName(lname);
158     s.setGrade(grade);
159
160     file.seekp(pos * sizeof(Student));
161     file.write(reinterpret_cast<const char*>(&s), sizeof(Student));
162     std::cout << "记录已更新\n";
163 }
164
165 }
```

异常处理



处理流程

try 块：

包裹可能抛出异常的代码。若 try 块内未抛出异常，程序继续执行后续代码。

throw 表达式：

抛出异常时，程序立即终止当前函数的执行，并沿调用链向上查找匹配的 **catch** 块。

catch 块：

捕获并处理特定类型的异常。支持多态匹配（如 `catch (const exception& e)` 可捕获所有标准异常派生类）。

终止模型 (Termination Model) :

异常未被捕获时，程序调用 `terminate()` 终止（默认调用 `abort()`）。

流程：

- 抛出异常 → 终止当前函数 → 栈展开 → 查找匹配的 **catch** → 执行处理代码 → 程序继续执行后续代码。

注意： 未匹配的异常会导致程序崩溃，需确保关键路径有兜底处理。

示例代码：

C++ |

```
1 #include <iostream>
2 #include <stdexcept>
3 using namespace std;
4
5 // 自定义除零异常类
6 class DivideByZeroException : public runtime_error {
7 public:
8     DivideByZeroException() : runtime_error("Attempted to divide by zero")
9     {}
10 }
11 // 计算商
12 double quotient(int numerator, int denominator) {
13     if (denominator == 0) {
14         throw DivideByZeroException(); // 抛出异常
15     }
16     return static_cast<double>(numerator) / denominator;
17 }
18
19 int main() {
20     int a = 10, b = 0;
21     try {
22         double result = quotient(a, b); // 可能抛出异常
23         cout << "Result: " << result << endl;
24     } catch (const DivideByZeroException& e) {
25         cerr << "DivideByZeroException: " << e.what() << endl; // 捕获并处
理
26     } catch (const exception& e) {
27         cerr << "Standard exception: " << e.what() << endl; // 兜底捕获
28     } catch (...) {
29         cerr << "Unknown exception occurred." << endl; // 捕获所有未匹配异常
30     }
31     return 0;
32 }
```

处理标准库抛出的异常

核心目标

捕获并处理C++标准库抛出的异常（如 `bad_alloc`、`out_of_range` 等），避免程序因未处理异常而崩溃。

- **关键操作**: 使用 `try`、`catch` 和 `throw` 分离正常逻辑与错误处理。

示例: `bad_alloc`

- **场景**: 使用 `new` 动态分配内存时, 若内存不足, 会抛出 `bad_alloc` 异常。
- **传统问题**: 未捕获异常会导致程序直接调用 `abort()`, 无法释放资源。
- **解决方案**: 通过 `try-catch` 捕获异常, 执行清理操作或优雅退出。

代码: 捕获 `bad_alloc`

```

1  #include <iostream>
2  #include <new> // for bad_alloc
3  using namespace std;
4
5  class Test {
6  public:
7      Test() { cout << "Constructor called." << endl; }
8      ~Test() { cout << "Destructor ok." << endl; }
9  };
10
11 int main() {
12     Test t;
13     double* ptr[50];
14
15     try {
16         for (int i = 0; i < 50; i++) {
17             ptr[i] = new double[50000000]; // 可能抛出 bad_alloc
18             cout << "Allocated 50,000,000 doubles in ptr[" << i << "]"
19             << endl;
20         }
21     } catch (bad_alloc& e) {
22         cerr << "Exception occurred: " << e.what() << endl;
23     }
24     cout << "Exception handled." << endl;
25     return 0;
26 }
```

输出示例:

```
1 Constructor called.  
2 Allocated 50,000,000 doubles in ptr[0]  
3 Allocated 50,000,000 doubles in ptr[1]  
4 ...  
5 Exception occurred: bad allocation  
6 Destructor ok.  
7 Exception handled.
```

异常类型匹配规则

- **is-a 关系：** `catch` 块按类型匹配异常。若异常类型是 `catch` 声明类型的派生类，则匹配成功。

```
1 catch (const std::exception& e) { /* 可捕获所有标准库异常 */ }
```

- **捕获所有异常：** 使用 `catch (...)` 捕获未知异常。

```
1 ~ try {  
2     // 可能抛出任何异常  
3 } catch (...) {  
4     cerr << "Unknown exception caught!" << endl;  
5 }
```

异常规范

- **异常说明 (Exception Specification) :** 指定函数可能抛出的异常类型。

```
1 void func() throw(std::runtime_error); // 仅允许抛出 runtime_error  
2 void func() throw(); // 不允许抛出任何异常
```

- **C++11 替代方案：** 使用 `noexcept` 指定函数不抛出异常。

```
1 void func() noexcept; // 等效于 throw()
```

栈展开

```

1 #include <iostream>
2 using namespace std;
3
4 class obj{
5     int id;
6 public:
7     obj(int n){
8         id=n;
9         cout<<"ctor"<<n<<endl;
10    }
11 ~obj(){
12     cout<<"dtor"<<id<<endl;
13    }
14 };
15
16 void f2(){
17     obj o(2);
18     double a=0;
19 try{
20     throw a;
21 }
22 catch(double e){
23     cout<<"OK2! "<<e<<endl;
24     throw;
25 }
26 cout<<"end2"<<endl;
27 }
28
29 void f1(){
30     obj o(1);
31 try{
32     f2();
33 }
34 catch(char e){
35     cout<<"OK1! "<<e<<endl;
36 }
37 cout<<"end1"<<endl;
38 }
39 int main(){
40 try{
41     obj o(0);
42     f1();
43 }
44 catch(double e){
45     cout<<"OK0!"<<e<<endl;
46 }
47 cout<<"end0"<<endl;

```

```
48     return 0;  
49 }
```

假如删去24行throw;

输出: ctor0

ctor1

ctor2

OK2! 0

dtor2

dtor1

dtor0

OK0! 0

end0

输出: ctor0

ctor1

ctor2

OK2! 0

end2

dtor2

end1

dtor1

dtor0

end0

为什么 dtor2 在 OK0! 之前?

- 当 f2 中的 throw; 重新抛出异常时, 程序会立即退出 f2 函数作用域
- 在退出 f2 时, 其局部对象 o(2) 必须被析构 → 输出 dtor2
- 只有完成当前作用域的清理后, 异常才会继续向上传播到 main

技术要点总结

1. 栈展开的触发时机: 每次执行 throw (包括重新抛出) 都会立即触发当前作用域的析构
2. 析构顺序原则: 后构造的对象先析构 (LIFO/FILO 后出先进/先进后出)
3. 重新抛出的特殊性: throw; 语句会保留原始异常对象, 但会触发当前函数作用域的清理

其他

生成随机数

```
1 #include <iostream>
2 #include <ctime>      // 时间
3 #include <random>     // 引入标准库中的随机数相关头文件
4 using namespace std;
5
6 int main() {
7     // 使用random_device来生成随机种子
8     random_device rd;
9     //这时候rd已经可以拿来用了, 如: cout<<rd();
10
11    default_random_engine generator(rd());
12    //或者 default_random_engine generator(static_cast<unsigned int>(time
13    //均匀实数分布, 用于生成范围内的浮点数, 可以用float, double 或者仅 <>
14    uniform_real_distribution<float> distribution1(0.0, 1.0);
15
16    //均匀整数分布, 用于生成范围内的整数, 可以用int ,ll 或者仅 <>
17    uniform_int_distribution<int> distribution2(0,10);
18
19    //正态分布, 用于生成符合高斯分布的浮点数。
20    normal_distribution<float> distribution3(0,10);
21
22    //伯努利分布, 用于生成0或1的随机布尔值。
23    bernoulli_distribution distribution4;
24
25    // 生成并打印10个随机数
26    for (int i = 0; i < 10; ++i)
27        cout << distribution1(generator) << " ";
28        //11行定义的generator就会在这里使用
29
30    return 0;
31 }
32 }
```

格式化操作

除了特定指出的，这里的操作都要调用，这个头文件提供格式化相关操作。

1. `left` 和 `right`

这两个操作符用于设置输出流的对齐方式。默认情况下，`iostream` 是右对齐的。特点：都是粘性设置。

- `left` : 设置输出流为左对齐。
- `right` : 设置输出流为右对齐（默认）。

2. `setw(int width)`

这个操作符用于设置输出的宽度。如果输出的字段宽度小于指定的宽度，`setw` 会在字段前填充空格（默认），或者根据对齐方式填充。如果输出字段宽度大于指定的宽度，`setw` 失效，无事发生。

3. `setfill(char fill)`

这个操作符用于设置填充字符。默认的填充字符是空格。

```
1 int a = 5;
2 int b = 10;
3
4 // 左对齐
5 cout << left;
6 cout << "左对齐: " << setw(10) << a << endl;
7 cout << "左对齐: " << setw(10) << b << endl;
8 //输出 左对齐:    5
9 //      左对齐:    10
10
11 // 右对齐
12 cout << right;
13 cout << "右对齐: " << setw(10) << a << endl;
14 cout << "右对齐: " << setw(10) << b << endl;
15 //输出 右对齐:    5
16 //      右对齐:    10
17
18 cout << setw(10) << setfill('*') << "" << endl;
19 //输出 *****
```

4. `fixed`

`fixed` 操纵算子用于设置浮点数的输出格式为固定小数点表示法。当使用 `fixed` 时，浮点数将被格式化为小数形式，即使这个数可以表示为一个整数或者科学记数法形式。当使用 `std::fixed` 时，通常配合使用 `setprecision` 来指定小数点后的位数，多去少补。

```
1 double pi = 3.141592653589793;
2 // 使用 fixed 和 setprecision 来设置输出格式
3 cout << fixed << setprecision(2) << pi << endl;
4 // 输出: 3.14
5
6 double pi = 3.1;
7 cout << fixed << setprecision(2) << pi << endl;
8 // 输出: 3.10
```

5. `showpoint`

`showpoint` 输出强制显示小数点后面的零，即使数字的值是整数。

```
1 double number = 12.0;
2
3 cout << "Without showpoint: " << number << endl;
4 //输出 Without showpoint: 12      系统自动省略了小数点和零
5
6 cout << "With showpoint: " << showpoint << number << endl;
7 //输出 With showpoint: 12.0000
8
9 cout << fixed << showpoint << 12.0 << endl;
10 // 输出: 12.00000
11
12 cout << scientific << showpoint << 12.0 << endl;
13 // 输出: 1.20000e+01
```

6. scientific

`scientific` 操纵算子用于设置浮点数的输出格式为科学记数法。使用 `scientific` 时，也可以配合使用 `setprecision` 来指定有效数字的位数，这通常包括小数点后的位数。

```
1 double largeNumber = 12345678901234567890;
2 // 使用 scientific 和 setprecision 来设置输出格式
3 cout << scientific << setprecision(3) << largeNumber << endl;
4 // 输出: 1.235e+19
```

- 当使用 `fixed` 或 `scientific` 时，它们会影响后续所有浮点数的输出，直到被改变或重置。
- `setprecision` 指定的是有效数字的位数，而不是小数点后的位数。这意味着如果数字小于 1，指定的精度也包括小数点前的数字。

7.setprecision

`setprecision` 用于设置浮点数输出时的有效数字的总数，包括小数点前的数字和小数点后的数字。若与`fixed`或`scientific`同时使用，则改为表示小数点后精度。

8. 设置bool

这里的可以不使用

- `boolalpha`：输出布尔值时使用 `true` 和 `false` 而不是 `1` 和 `0`。

- `noboolalpha` : 输出布尔值时使用 `1` 和 `0`。

9.设置数值的基数

这里的可以不使用

- `dec` : 设置数值以十进制形式输出。
- `hex` : 设置数值以十六进制形式输出。
- `oct` : 设置数值以八进制形式输出。

10.设置精度

- `fixed` : 设置浮点数以固定小数点形式输出。
- `setprecision(int precision)` : 设置浮点数输出时的精度（小数点后的位数）。
- `scientific` : 设置浮点数以科学记数法形式输出。

特别注意

- 黏性设置: `left`, `right`, `fixed`, `scientific`, `showpoint`, `boolalpha`, `noboolalpha`, `dec`, `hex`, `oct`, `setfill`
- 一次性设置: `setw`, `setprecision` (如果没有与 `fixed` 或 `scientific` 配合使用, 则是一次性设置)

逗号表达式

1.逗号表达式结合性优先级最低

```
1 int value = 0;
2 cout << value, ++value << value;
3 //输出: 0
4 //由于优先级最低, 可以直接将 , 改为 ; 进行理解
5 //即 cout << value; ++value << value;
6 //而后面的++value<<value就是一种位运算, 但是没赋值
```

2. 左边子表达式的计算和副作用均先于右边子表达式

```
1 #include <iostream>
2 int value = 1;
3
4 void f() {
5     value += 1;
6     std::cout << value;
7 }
8
9 void g() {
10    value *= 10;
11    std::cout << value;
12 }
13
14 int main() {
15     f(), g();
16     //输出: 220 注意 没有空格
17 }
```

3. 整个表达式的值是最右边子表达式的值

```
1 #include <iostream>
2
3 int function() {
4     int value = 0;
5     return value++, value;
6 }
7
8 int main() {
9     std::cout << function();
10    //输出: 1
11 }
```

部分总结

```
1 int x,y,z;
2 x=y=1;      //y先赋值1, 然后y的值赋值给x, 本质来讲就是=从右往左的顺序
3 z=x++,y++,++y;    //由于x++操作在后面, 所以先是z=x, 然后x++,y++,++y, 这里可以把逗号当做; 来理解
4 cout<<x<<" " <<y<<" " <<z;    //输出: 2 3 1
5 // 若z=(x++,y++,++y); 则输出: 2 3 2
```