C++ 核心基础

语言特性与面试必备知识点

核心模块

- 左值右值与引用机制
- 移动语义与完美转发
- const、inline、constexpr 关键字
- 函数指针与类型转换
- 虚函数与多态机制
- 智能指针与 RAII
- 异常处理与安全性保证
- 现代 C++ 特性 (C++11/14/17)

面试八股 · 核心概念 · 深度解析

作者：Aweo

2025 年 10 月

Contents

1 基础部分

1.1 静态局部变量，全局变量，局部变量的特点，以及使用场景？

三者的特点可以从存储位置、生命周期、作用域和初始化方式四个维度分析：

1. 存储位置

- 静态局部变量：静态存储区（显示初始化的存储在.data 段，未初始化的存储在.bss 段）
- 全局变量：静态存储区（显示初始化的存储在.data 段，未初始化的存储在.bss 段）
- 局部变量：栈区

2. 生命周期

- 静态局部变量：程序运行期间持续存在（首次调用时初始化）
- 全局变量：程序运行期间持续存在（main 函数执行前初始化）
- 局部变量：函数调用期间存在（每次调用重新创建）

3. 作用域

- 静态局部变量：仅在定义它的函数/代码块内可见（因此同名变量之间不会冲突）
- 全局变量：从定义位置到文件结尾可见（可通过 extern 扩展到其他文件，如果添加 static 关键字，则只在本文件中可见）
- 局部变量：仅在定义它的函数/代码块内可见

4. 初始化

- 静态局部变量：默认零初始化，只初始化一次
- 全局变量：默认零初始化，编译期初始化
- 局部变量：需要显式初始化，每次调用重新初始化

使用场景：

- 静态局部变量：需要保持状态但不需要全局可见的场景（如计数器、统计函数执行次数、递归深度控制）
- 全局变量：需要跨多个函数/文件共享数据的场景（例如配置信息，缓冲区等）
- 局部变量：函数内部临时使用的数据存储

关于静态全局变量的补充：静态全局变量（static 修饰的全局变量）具有文件作用域，与普通全局变量的主要区别在于：

1. 链接属性不同：静态全局变量具有 internal linkage，普通全局变量具有 external linkage
2. 可见性不同：静态全局变量仅在定义它的编译单元（.cpp 文件）内可见
3. 使用建议：当需要文件内共享数据但避免与其他文件产生符号冲突时使用
4. 使用时与变量类型的顺序无关，例如 int static a; 和 static int a; 是等价的，但规范是 static int a;

1.2 static 关键字的作用

static 关键字主要用于控制变量和函数的生命周期、作用域以及访问权限。

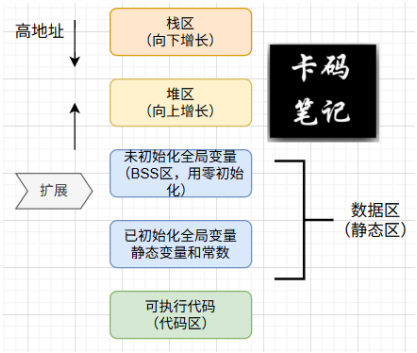

Figure 1: 内存四区示意图（图示提供：KamAcademy（卡码笔记））

2 指针和引用的区别

指针是变量，存储的值是内存地址，可以被修改，可以为空(nullptr)，可以进行加减运算（加减运算的单位是 sizeof(指针类型)）。可以有 const 修饰。此外，指针变量占用独立内存（32 位系统 4 字节，64 位系统 8 字节）

引用是别名，必须初始化且不能被修改，不能为空(nullptr)，不能进行加减运算。没有 const 修饰。它不占内存，由编译器在内部实现。

引用语法与值的属性（左值、右值）有关。

2.1 左值与右值的本质区别

根本区别：

- 左值**(lvalue)**：有身份（具名），有持久的内存地址，可以取地址，生命周期由作用域决定
- 右值**(rvalue)**：无身份（匿名），无持久地址，不可取地址，临时对象，表达式结束后销毁，但如果引用右值，则其生命周期会被延长

C++11 值分类体系：

```
1         表达式 (expression)
2        /                \
3    glvalue              rvalue
4    /     \              /     \
5  lvalue  xvalue     xvalue   prvalue
```

- **lvalue**（左值）：传统意义的左值，如变量名、前置++、解引用等
- **prvalue**（纯右值）：纯粹的临时值，如字面量、后置++、lambda 表达式等
- **xvalue**（将亡值）：即将被移动的值，如 `std::move(x)`、返回右值引用的函数调用

```cpp
int x = 10;         // x是左值，10是右值
int* p = &x;        // &x合法，x有地址
// int* p2 = &10;   // 错误！10是右值，无法取地址

int&& rref = std::move(x);  // std::move(x)是xvalue
```

2.2 引用语法详解

左值引用（**Lvalue Reference**）：

```cpp
int x = 42;
int& lref = x;           // ✓ 左值引用绑定左值
// int& lref2 = 42;      // ✗ 左值引用不能绑定右值
const int& cref = 42;    // ✓ const左值引用可以绑定右值（特例！）
```

右值引用（**Rvalue Reference C++11**）：

```cpp
int&& rref = 42;              // ✓ 右值引用绑定右值
int&& rref2 = std::move(x);   // ✓ std::move将左值转为右值
// int&& rref3 = x;           // ✗ 右值引用不能直接绑定左值
```

const 左值引用的特殊性：

```cpp
const std::string& ref = std::string("temp");
// 1. 创建临时string对象（右值）
// 2. const左值引用可以绑定右值（C++98特性，用于临时对象优化）
// 3. 临时对象生命周期延长到ref的作用域结束
```

生命周期延长示例：

```cpp
class Demo {
public:
    Demo() { std::cout << "构造\n"; }
    ~Demo() { std::cout << "析构\n"; }
};

Demo&& rref = Demo();  // 右值引用绑定临时对象
// 临时对象生命周期延长到rref作用域结束，可安全使用
// rref超出作用域时，临时对象才析构
```

2.3 为什么需要移动语义和完美转发？

移动语义（**Move Semantics**）的必要性：

问题背景：C++03 时代，所有对象传递都依赖拷贝，造成巨大性能开销。

```cpp
// C++03：深拷贝的性能问题
std::vector<int> create_large_vector() {
    std::vector<int> v(1000000);  // 100万元素
    // ... 填充数据
    return v;  // 返回时会深拷贝整个vector！
}

std::vector<int> data = create_large_vector();
// 1. 函数内创建vector（分配100万个int）
// 2. 返回时拷贝构造临时对象（再分配100万个int）
// 3. 用临时对象拷贝构造data（第三次分配100万个int）
// 4. 销毁临时对象（释放内存）
// 结果：三次内存分配，两次不必要的拷贝！
```

移动语义的解决方案：

```cpp
// C++11：移动构造，直接"偷"资源
std::vector<int> create_large_vector() {
    std::vector<int> v(1000000);
    return v;  // 返回右值，触发移动构造
}

std::vector<int> data = create_large_vector();
// 1. 函数内创建vector（分配100万个int）
// 2. 返回时移动构造临时对象（只转移指针，不拷贝数据）
// 3. 用临时对象移动构造data（只转移指针）
// 结果：一次内存分配，零拷贝！性能提升数十倍
```

核心价值：

1. 避免深拷贝：对于管理资源的类（如 `std::vector`、`std::string`），移动只转移指针，而非拷贝整个数据
2. 临时对象优化：临时对象本来就要销毁，直接“偷走”它的资源是安全且高效的
3. 显式资源转移：通过 `std::move` 明确表达“我不再需要这个对象”的语义

完美转发（**Perfect Forwarding**）的必要性：

问题背景：编写通用的包装函数时，无法保持参数的值类别（左值/右值）。

```cpp
// C++03：无法区分左值和右值
template<typename T>
void wrapper(T arg) {  // 按值传递：总是拷贝
```

```
    process(arg);      // 传递给process时总是左值
}

std::string s = "hello";
wrapper(s);                    // 拷贝一次
wrapper(std::string("temp"));  // 拷贝一次（本可移动！）
```

完美转发的解决方案：

```C++
// C++11：保持参数的值类别
template<typename T>
void wrapper(T&& arg) {              // 转发引用
    process(std::forward<T>(arg));   // 完美转发
}

std::string s = "hello";
wrapper(s);                    // 传递左值→调用process的左值版本（拷贝）
wrapper(std::string("temp"));  // 传递右值→调用process的右值版本（移动）
```

实际应用场景：

```C++
// 标准库中的emplace_back实现
template<typename T>
class vector {
public:
    template<typename... Args>
    void emplace_back(Args&&... args) {
        // 完美转发参数给T的构造函数
        new (ptr) T(std::forward<Args>(args)...);
    }
};

// 使用示例：
std::vector<std::string> vec;
std::string s = "hello";

vec.push_back(s);           // 拷贝s
vec.push_back(std::move(s)); // 移动s
vec.emplace_back("world");  // 完美转发"world"，原地构造，零拷贝！
```

核心价值：

1. 零开销抽象：包装函数不引入额外的拷贝/移动开销
2. 类型无关：适用于任意类型，无需为每种类型重载
3. 参数完整性：保持参数的所有属性（值类别、const 性等）

性能对比：

```C++
// 假设Big对象拷贝需要1ms，移动需要1μs
class Big {
    char data[1000000];
public:
    Big(const Big&) { /* 深拷贝：1ms */ }
    Big(Big&&) { /* 移动：1μs */ }
};

// 不使用移动语义和完美转发
```

```cpp
void old_style(Big b) { vec.push_back(b); }  // 两次拷贝：2ms
old_style(create_big());  // 总耗时：2ms

// 使用移动语义和完美转发
template<typename T>
void new_style(T&& b) { vec.push_back(std::forward<T>(b)); }
new_style(create_big());  // 两次移动：2μs，性能提升1000倍！
```

总结：

- 移动语义：解决“临时对象拷贝”的性能问题，让资源转移代替深拷贝
- 完美转发：解决“参数传递丢失值类别”的问题，让包装函数零开销
- 两者结合，实现了 C++的零开销抽象理念

2.4 引用折叠与完美转发原理

引用折叠规则（**Reference Collapsing**）：

在模板或 typedef 中，引用的引用会按以下规则折叠：

```cpp
// 规则：只有"右值引用的右值引用"折叠为右值引用，其余都折叠为左值引用
& + &   → &       // 左值引用 + 左值引用 = 左值引用
& + &&  → &       // 左值引用 + 右值引用 = 左值引用
&& + &  → &       // 右值引用 + 左值引用 = 左值引用
&& + && → &&      // 右值引用 + 右值引用 = 右值引用
```

转发引用（**Forwarding Reference/Universal Reference**）：

```cpp
template<typename T>
void func(T&& param);  // T&&在模板中是转发引用，不是普通右值引用！

int x = 10;
func(x);   // 传入左值：T推导为int&,  param类型为 int& && → int&（折叠）
func(10);  // 传入右值：T推导为int,   param类型为 int&&

// 详细推导过程：
// 1. 传入左值x时：
//    T = int&
//    T&& = int& && → int&（引用折叠）
//    结果：param是左值引用
//
// 2. 传入右值10时：
//    T = int
//    T&& = int&&
//    结果：param是右值引用
```

std::forward 的实现原理：

```cpp
// 简化版实现
template<typename T>
T&& forward(typename std::remove_reference<T>::type& param) {
    return static_cast<T&&>(param);
}

// 工作原理：
// 1. 如果T是int&（左值传入）：
//    返回类型：int& && → int&（引用折叠）
```

```
//     效果：转发为左值引用
//
// 2. 如果T是int（右值传入）：
//     返回类型：int&&
//     效果：转发为右值引用

// 使用示例：
template<typename T>
void wrapper(T&& arg) {
    process(std::forward<T>(arg));  // 保持arg的原始值类别
}

int x = 10;
wrapper(x);      // arg绑定左值，转发为左值
wrapper(10);     // arg绑定右值，转发为右值
```

std::move 的实现原理：

C++

```
// 简化版实现
template<typename T>
typename std::remove_reference<T>::type&& move(T&& param) {
    using ReturnType = typename std::remove_reference<T>::type&&;
    return static_cast<ReturnType>(param);
}

// 工作原理：无条件将任何参数转换为右值引用
int x = 10;
int&& rref = std::move(x);  // x是左值，std::move将其转为右值

// 注意：std::move不真正"移动"任何东西，只是类型转换！
// 真正的移动发生在移动构造函数或移动赋值运算符中
```

关键注意事项：

C++

```
template<typename T>
void func(T&& param) {
    // 重要：param虽然类型可能是右值引用，但param本身是左值（有名字！）
    process(param);                  // 总是调用process的左值版本
    process(std::forward<T>(param)); // 保持原始值类别
    process(std::move(param));       // 无条件转为右值
}

// 这是一个常见误区：
int&& rref = 42;
// rref的类型是"右值引用"，但rref本身是"左值"（因为有名字）
process(rref);  // 调用process(int&)，不是process(int&&)！
```

2.5 移动语义和完美转发的高级应用

1. 实现移动构造函数和移动赋值运算符：

C++

```
class MyString {
private:
    char* data;
    size_t length;

```

```
public:
    // 拷贝构造（深拷贝）
    MyString(const MyString& other)
        : length(other.length), data(new char[length]) {
        std::memcpy(data, other.data, length);
    }

    // 移动构造（转移资源）
    MyString(MyString&& other) noexcept
        : data(other.data), length(other.length) {
        other.data = nullptr;    // 将源对象置空
        other.length = 0;
    }

    // 移动赋值运算符
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            delete[] data;       // 释放当前资源
            data = other.data;   // 转移资源
            length = other.length;
            other.data = nullptr;// 将源对象置空
            other.length = 0;
        }
        return *this;
    }

    ~MyString() { delete[] data; }
};

// 使用示例：
MyString s1("hello");
MyString s2 = std::move(s1);  // 调用移动构造，s1的资源转移到s2
// 注意：s1现在处于有效但未定义的状态，不应再使用
```

2. 完美转发的实际应用：

```
// 工厂函数模板
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}

// emplace系列函数
template<typename T>
class vector {
public:
    template<typename... Args>
    void emplace_back(Args&&... args) {
        // 完美转发所有参数，直接在容器内存中构造对象
        new (end_ptr) T(std::forward<Args>(args)...);
        ++end_ptr;
    }
};

// 使用对比：
```

```
std::vector<std::pair<int, std::string>> vec;

// 方式1：临时对象 + 移动
vec.push_back(std::make_pair(1, "hello"));  // 1次构造 + 1次移动

// 方式2：完美转发，原地构造
vec.emplace_back(1, "hello");  // 1次构造，零拷贝零移动！
```

3. 函数返回值优化（RVO/NRVO）：

```cpp
// 返回值优化（RVO）
std::vector<int> create_vector() {
    return std::vector<int>(1000);  // RVO：直接在调用方的内存中构造
}

// 具名返回值优化（NRVO）
std::vector<int> create_vector_nrvo() {
    std::vector<int> v(1000);
    // ... 操作v
    return v;  // NRVO：编译器优化掉拷贝/移动
}

// 现代C++（C++17保证）：
auto vec = create_vector();  // 零拷贝，直接构造在vec的位置
```

4. type_traits 类型萃取：

```cpp
#include <type_traits>

template<typename T>
void smart_process(T&& param) {
    using RawType = typename std::remove_reference<T>::type;

    // 编译期类型判断
    if constexpr (std::is_lvalue_reference<T>::value) {
        std::cout << "收到左值引用，使用拷贝策略\n";
        // 拷贝相关操作
    } else if constexpr (std::is_rvalue_reference<T>::value) {
        std::cout << "收到右值引用，使用移动策略\n";
        // 移动相关操作
    }

    // 其他有用的traits
    static_assert(std::is_move_constructible<RawType>::value,
                  "T must be move constructible");
    static_assert(std::is_nothrow_move_constructible<RawType>::value,
                  "T's move constructor should be noexcept");
}
```

5. 条件移动（Conditional Move）：

```cpp
template<typename T>
class optional {
    alignas(T) char storage[sizeof(T)];
    bool has_value;

```

```
public:
    // 如果T支持移动，使用移动；否则拷贝
    template<typename U>
    void emplace(U&& value) {
        if constexpr (std::is_nothrow_move_constructible<T>::value) {
            new (storage) T(std::move(value));  // 移动
        } else {
            new (storage) T(value);  // 拷贝（更安全）
        }
        has_value = true;
    }
};
```

6. 性能最佳实践：

```
// ✓ 推荐：按值返回，依赖RVO/移动
std::vector<int> good_return() {
    std::vector<int> result(1000);
    return result;  // RVO或移动，性能优异
}

// ✗ 避免：返回const值（阻止移动）
const std::vector<int> bad_return() {
    return std::vector<int>(1000);  // const阻止移动！
}

// ✓ 推荐：移动构造函数标记noexcept
class MyClass {
public:
    MyClass(MyClass&&) noexcept;  // noexcept允许容器优化
};

// ✓ 推荐：sink参数按值传递+移动
class Widget {
    std::string name;
public:
    // 一个函数同时支持拷贝和移动
    void set_name(std::string n) {  // 按值传递
        name = std::move(n);  // 移动赋值
    }
};

Widget w;
std::string s = "hello";
w.set_name(s);           // s被拷贝到参数n，然后移动到name
w.set_name("world");     // 临时对象移动到参数n，再移动到name
```

总结与最佳实践：

1. 移动构造/赋值应标记 **noexcept**：允许容器等使用移动优化
2. 移动后的对象应处于有效状态：可以安全析构和赋新值
3. 完美转发用于模板：保持参数的所有属性不变
4. 优先使用 **emplace**：原地构造，避免临时对象
5. 理解“有名字的都是左值”：需要显式 `std::move` 或 `std::forward`
6. 依赖 **RVO**：按值返回局部对象，编译器会优化

3 const 关键字详解

const 是 C++中最常用的关键字之一，用于声明常量和保证对象不被修改，是实现封装性和类型安全的重要工具。

3.1 const 修饰变量

基本常量：

```cpp
const int MAX_SIZE = 100;      // 整型常量
const double PI = 3.14159;     // 浮点常量
const char* str = "hello";     // 指向常量字符的指针

// const位置的等价性（仅对简单类型）
const int x = 10;    // 等价于
int const y = 10;    // 两者完全相同
```

const 与指针：

```cpp
 // 1. 指向常量的指针（pointer to const）
 const int* p1;         // 不能通过p1修改所指向的值
 int const* p2;         // 同上，写法不同

 // 2. 常量指针（const pointer）
 int* const p3 = &x;    // p3本身不能改变指向，但可以修改所指向的值

 // 3. 指向常量的常量指针
 const int* const p4 = &x;  // 指向和值都不能改变

// 记忆技巧：从右向左读
// int* const p → p是const指针，指向int
// const int* p → p是指针，指向const int
```

const 引用：

```cpp
int x = 10;
const int& ref = x;    // 常量引用，不能通过ref修改x
// ref = 20;           // 错误！

// const引用的特殊能力：可以绑定临时对象（右值）
const int& temp_ref = 42;         // 合法！临时对象生命周期延长
const std::string& s = "hello";  // 合法！临时string对象绑定到引用
```

3.2 const 修饰函数参数

按值传递：

```cpp
void func1(const int x) {
    // x = 10;  // 错误！x是常量，不能修改
    int y = x;  // 可以读取
}
// 注意：按值传递时加const意义不大（函数内部修改的是副本）
// 通常不这样写，除非强调"不修改参数"的语义
```

按引用传递（重要！）：

```cpp
 // 避免拷贝，同时保证不修改原对象，一般作为一种语法习惯
 void process(const std::string& str) {
```

```
    std::cout << str;  // 可以读取
    // str += "!";     // 错误！不能修改
}

// 对比：
void bad_process(std::string str) {      // 按值传递：拷贝整个string，开销大
    std::cout << str;
}

void better_process(const std::string& str) {  // 常量引用：零拷贝，效率高
    std::cout << str;
}

// 使用场景：传递大对象时，优先使用const引用
void display(const std::vector<int>& vec);      // ✓ 推荐
void display(std::vector<int> vec);             // ✗ 低效，会拷贝
```

按指针传递：

```cpp
// 指向常量的指针：不能修改指向的内容
void print_array(const int* arr, size_t size) {
    for (size_t i = 0; i < size; ++i) {
        std::cout << arr[i];  // 可以读取
        // arr[i] = 0;        // 错误！不能修改
    }
}
```

3.3 const 修饰成员函数（重点）

基本语法：

```cpp
// const成员函数的完整语法顺序是：
// 返回类型 函数名(参数列表) const [异常说明] [->尾置返回类型];
class Point {
private:
    int x, y;

public:
    Point(int x, int y) : x(x), y(y) {}

    // const成员函数：承诺不修改对象的成员变量
    int getX() const { return x; }     // ✓ 只读取，不修改
    int getY() const { return y; }

    // ✗ 错误：const不能放在返回类型前面
    // const void func3() { }  // 这表示返回const void，不是const成员函数

    // ✗ 错误：const不能放在函数名前面
    // void const func4() { }  // 语法错误

    // ✗ 错误：const不能放在参数列表里面
    // void func5(const) { }   // 语法错误

    // 非const成员函数：可以修改成员变量
    void setX(int newX) { x = newX; }  // 不能声明为const
```

```
25
26
27      // const成员函数中不能修改成员变量
28      void invalidFunc() const {
29          // x = 10;  // 错误！const函数不能修改成员
30      }
31  };
```

const 对象只能调用 const 成员函数：

```C++
1  Point p1(1, 2);
2  p1.getX();    // ✓ 非const对象可以调用const函数
3  p1.setX(3);   // ✓ 非const对象可以调用非const函数
4
5  const Point p2(10, 20);
6  p2.getX();    // ✓ const对象可以调用const函数
7  // p2.setX(5); // ✗ 错误！const对象不能调用非const函数
```

const 重载：

```C++
1   class MyString {
2   private:
3       char* data;
4
5   public:
6       // 非const版本：返回可修改的引用
7       char& operator[](size_t index) {
8           return data[index];
9       }
10
11      // const版本：返回常量引用
12      const char& operator[](size_t index) const {
13          return data[index];
14      }
15  };
16
17  MyString s1("hello");
18  s1[0] = 'H';           // 调用非const版本，可以修改
19
20  const MyString s2("world");
21  char c = s2[0];        // 调用const版本，只能读取
22  // s2[0] = 'W';        // 错误！const版本返回const引用
```

const 成员函数的实现细节：

```C++
1   class Widget {
2   private:
3       int value;
4
5   public:
6       // 编译器实际上是这样理解const成员函数的：
7       void normalFunc() {
8           // 隐式参数：Widget* this
9           this->value = 10;  // 可以修改
10      }
11
```

```
    void constFunc() const {
        // 隐式参数：const Widget* this
        // this->value = 10;  // 错误！this是指向const的指针
        int x = this->value;  // 可以读取
    }
};
```

3.4 const 对象和 const 成员变量

const 对象：

```C++
const Point p(10, 20);  // 常量对象
// p.setX(5);           // 错误！不能调用非const成员函数
int x = p.getX();       // 正确！可以调用const成员函数
```

const 成员变量：

```C++
class Config {
private:
    const int MAX_CONNECTIONS;  // const成员变量
    const std::string APP_NAME;

public:
    // 必须在初始化列表中初始化const成员
    Config() : MAX_CONNECTIONS(100), APP_NAME("MyApp") {}

    // 错误的方式：
    // Config() {
    //     MAX_CONNECTIONS = 100;  // 错误！const成员不能赋值
    // }

    int getMax() const { return MAX_CONNECTIONS; }
};
```

静态 const 成员：

```C++
class Math {
public:
    static const double PI;           // 声明
    static constexpr int MAX = 100;   // C++11：可以直接初始化整型常量
};

// 类外定义（非constexpr的static const成员）
const double Math::PI = 3.14159;
```

3.5 mutable 关键字

mutable 允许在 const 成员函数中修改某些成员变量，通常用于缓存、统计等场景。

```C++
class Cache {
private:
    mutable int access_count;    // mutable成员
    mutable bool cache_valid;
    mutable std::string cached_data;
    std::string data;

public:
```

```
    // const函数，但可以修改mutable成员
    const std::string& getData() const {
        access_count++;  // ✓ 合法！access_count是mutable

        if (!cache_valid) {
            cached_data = computeData();  // ✓ 合法！
            cache_valid = true;
        }
        return cached_data;
    }

    std::string computeData() const {
        // data = "new";  // 错误！data不是mutable
        return data + "_processed";
    }
};

// 实际应用：延迟计算
class Person {
private:
    std::string first_name;
    std::string last_name;
    mutable std::string full_name;  // 缓存
    mutable bool full_name_cached;

public:
    const std::string& getFullName() const {
        if (!full_name_cached) {
            full_name = first_name + " " + last_name;  // 计算并缓存
            full_name_cached = true;
        }
        return full_name;
    }
};
```

3.6 const 修饰返回值

返回值为 **const**：

```
// 返回const值（现代C++中不推荐，会阻止移动）
const std::string getValue() {
    return std::string("hello");  // 阻止移动优化
}

// 返回const引用（常见于访问器）
class Container {
private:
    std::vector<int> data;

public:
    const std::vector<int>& getData() const {
        return data;  // 返回const引用，防止外部修改
    }
};
```

```cpp
// 返回const指针
class Manager {
private:
    Resource* resource;

public:
    const Resource* getResource() const {
        return resource;  // 返回指向const的指针
    }
};
```

3.7 const 与线程安全

```cpp
class ThreadSafeCounter {
private:
    mutable std::mutex mtx;  // mutable：在const函数中也需要加锁
    int count;

public:
    int getCount() const {
        std::lock_guard<std::mutex> lock(mtx);  // 可以修改mutable成员
        return count;
    }

    void increment() {
        std::lock_guard<std::mutex> lock(mtx);
        ++count;
    }
};
```

3.8 const 的最佳实践

1. 尽可能使用 const：

```cpp
// ✓ 推荐：能用const就用const
void process(const std::string& input);     // 参数不修改
int compute(const Data& data) const;        // 成员函数不修改对象

// ✗ 避免：不必要的非const
void process(std::string& input);           // 暗示会修改input
int compute(Data& data);                    // 暗示会修改data
```

2. const 正确性（const-correctness）：

```cpp
class Good {
public:
    int getValue() const { return value; }          // ✓ 访问器应该是const
    void setValue(int v) { value = v; }             // ✓ 修改器不是const

    void display() const {                          // ✓ 只读操作应该是const
        std::cout << value << std::endl;
    }

private:
    int value;
};
```

3. 返回 const 引用避免拷贝：

```cpp
class StringHolder {
private:
    std::string data;

public:
    // ✓ 推荐：返回const引用
    const std::string& getData() const { return data; }

    // ✗ 低效：返回值（会拷贝）
    std::string getData_bad() const { return data; }
};
```

4. 顶层 const vs 底层 const：

```cpp
// 顶层const：对象本身是常量
const int x = 10;
int* const p = &y;  // p本身是常量（顶层const）

// 底层const：指向的对象是常量
const int* q = &x;  // q指向常量（底层const）

// 赋值时：顶层const可以忽略，底层const必须匹配
int* p2 = p;        // ✗ 错误！底层const不匹配
const int* q2 = q;  // ✓ 正确
```

5. const 与 auto：

```cpp
const int x = 10;
auto y = x;        // y是int，丢失顶层const
const auto z = x;  // z是const int

const int& ref = x;
auto r1 = ref;     // r1是int，丢失引用和const
auto& r2 = ref;    // r2是const int&，保留引用和const
```

总结：

1. **const 成员函数**：不修改成员变量的函数都应声明为 const
2. **const 引用参数**：传递大对象时使用 const& 避免拷贝
3. **const 对象**：只能调用 const 成员函数
4. **mutable**：在 const 函数中修改缓存、统计信息等
5. **const 正确性**：整个程序保持 const 的一致性
6. **返回 const 引用**：避免不必要的拷贝，但注意生命周期

4 define、typedef、inline、const 的区别

这四个关键字在 C++中有不同的用途和特性，理解它们的区别对于写出高质量的代码很重要。

4.1 define vs typedef

基本对比：

特性	#define	typedef
处理阶段	预处理阶段（文本替换）	编译阶段（类型系统）
本质	宏定义，纯文本替换	类型别名，真正的类型
类型检查	无类型检查	有类型检查
作用域	全局，从定义到文件结尾	遵守 C++作用域规则
调试	难调试（宏展开后）	易调试（保留类型信息）
结尾分号	不需要	必须有

关键区别示例：

```cpp
// 1. 指针类型定义的陷阱
#define PINT int*
typedef int* PtrInt;

PINT p1, p2;      // 展开为：int* p1, p2;
                  // p1是int*，p2是int！(常见错误)

PtrInt p3, p4;    // p3和p4都是int*（正确）

// 2. 作用域
namespace MyNamespace {
    typedef int MyInt;       // 仅在命名空间内有效
    // #define MY_INT int    // 定义后全局有效，污染全局命名空间
}

void func() {
    typedef double LocalDouble;  // 局部作用域
    // #define LOCAL_DOUBLE double // 从这里到文件结尾都有效
}

// 3. 类型检查
typedef int Integer;
void process(Integer x) { }

#define INT int
void handle(INT y) { }

Integer a = 10;
process(a);      // 编译器知道a是Integer类型

INT b = 20;
handle(b);       // 编译器只看到int，不知道原始定义
```

typedef 的现代替代：using（C++11 推荐）：

```cpp
// C++11: using更清晰，尤其是模板别名
typedef std::vector<int> IntVec;          // 传统方式
```

```cpp
using IntVec = std::vector<int>;           // 现代方式，更易读

// 模板别名（typedef做不到）
template<typename T>
using Vec = std::vector<T>;

Vec<int> v1;      // 等价于 std::vector<int>
Vec<string> v2;   // 等价于 std::vector<string>

// 复杂类型别名
typedef void (*FuncPtr)(int, double);           // 传统：难读
using FuncPtr = void(*)(int, double);           // 现代：易读
```

4.2 define vs const

定义常量的对比：

特性	#define	const
类型	无类型	有明确类型
内存	不占用内存（文本替换）	占用内存（变量）
作用域	全局（预处理）	遵守作用域规则
调试	无法查看值	可以查看值
类型安全	无	有
可取地址	不可以	可以

实际对比：

C++

```cpp
// 1. 类型安全
#define PI 3.14159
const double PI_CONST = 3.14159;

double area1 = PI * r * r;          // PI被替换为字面量
double area2 = PI_CONST * r * r;    // 类型检查，更安全

// 2. 作用域
class Math {
public:
    // #define CLASS_PI 3.14  // 错误！#define不能有类作用域
    static const double CLASS_PI;   // ✓ 正确
};

// 3. 调试
#define MAX_SIZE 100
const int MAX_SIZE_CONST = 100;

// 调试时：
// MAX_SIZE：看到的是100（字面量）
// MAX_SIZE_CONST：看到的是变量名和值

// 4. 指针和引用
#define NUM 42
const int NUM_CONST = 42;

```

```
// const int* p1 = &NUM;         // 错误！宏没有地址
const int* p2 = &NUM_CONST;      // ✓ 正确

// 5. 类型转换
#define SIZE 100
const int SIZE_CONST = 100;

long long big = SIZE;             // 隐式转换，可能出错
long long big2 = SIZE_CONST;      // 编译器可以检查
```

现代 C++建议：

```
// ✗ 不推荐：使用宏定义常量
#define MAX_BUFFER 1024
#define PI 3.14159

// ✓ 推荐：使用const或constexpr
const int MAX_BUFFER = 1024;
constexpr double PI = 3.14159;

// ✓ 推荐：类内常量
class Config {
public:
    static constexpr int MAX_CONNECTIONS = 100;
    static const std::string APP_NAME;  // 非字面类型，类外定义
};
```

4.3 inline vs define（函数宏）

函数定义的对比：

特性	#define 宏	inline 函数
类型检查	无	有完整类型检查
参数求值	可能多次求值（危险）	只求值一次
作用域	全局	遵守作用域规则
调试	困难	容易
副作用	容易出错	安全
类型安全	无	有

宏的危险示例：

```
// 宏定义：危险！
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#define SQUARE(x) ((x) * (x))

int x = 5;
int result1 = MAX(x++, 10);  // x++ 可能被执行两次！
                             // 展开：((x++) > (10) ? (x++) : (10))

int result2 = SQUARE(x + 1); // 展开：((x + 1) * (x + 1))
                             // 看似正确，但如果是 x++ 就错了

// inline函数：安全
inline int max(int a, int b) {
```

```
    return a > b ? a : b;
}

inline int square(int x) {
    return x * x;
}

int y = 5;
int result3 = max(y++, 10);   // y++ 只执行一次
int result4 = square(y + 1);  // 安全，参数只求值一次
```

完整对比示例：

C++

```
// 1. 宏：文本替换
#define ADD(x, y) ((x) + (y))

int a = ADD(1, 2);            // 展开为：((1) + (2))
double b = ADD(1.5, 2.5);     // 展开为：((1.5) + (2.5))
                              // 没有类型检查！

// 2. inline：真正的函数
inline int add(int x, int y) {
    return x + y;
}

inline double add(double x, double y) {  // 可以重载
    return x + y;
}

int c = add(1, 2);            // 调用 int 版本
double d = add(1.5, 2.5);     // 调用 double 版本
// add("hello", "world");     // 错误！类型检查

// 3. 宏的副作用问题
#define MIN(a, b) ((a) < (b) ? (a) : (b))

int x = 5, y = 10;
int result = MIN(x++, y++);  // x和y的增量次数不确定！
// 如果 x < y：x增加2次，y增加1次
// 如果 x >= y：x增加1次，y增加2次

// 4. inline函数：安全
inline int min(int a, int b) {
    return a < b ? a : b;
}

int x2 = 5, y2 = 10;
int result2 = min(x2++, y2++);  // x2和y2各增加1次，行为明确
```

inline 的注意事项：

C++

```
// inline只是建议，编译器可以忽略
inline void complex_function() {
    // 很长的函数体
    // 编译器可能不会内联
```

```
}

// 现代C++：编译器很聪明
// 即使不写inline，编译器也可能内联短函数
int simple_add(int a, int b) {
    return a + b;  // 编译器可能自动内联
}

// constexpr隐含inline
constexpr int factorial(int n) {  // 自动inline
    return n <= 1 ? 1 : n * factorial(n - 1);
}
```

4.4 综合对比总结

1. 定义常量：

```cpp
// ✗ 不推荐
#define MAX_SIZE 100

// ✓ 推荐
const int MAX_SIZE = 100;           // C++98
constexpr int MAX_SIZE = 100;       // C++11，编译期常量
```

2. 定义类型别名：

```cpp
// ✗ 不推荐（但有时不可避免）
#define IntPtr int*

// ✓ 推荐
typedef int* IntPtr;                // C++98
using IntPtr = int*;                // C++11，更清晰
```

3. 定义“函数”：

```cpp
// ✗ 不推荐
#define MAX(a, b) ((a) > (b) ? (a) : (b))

// ✓ 推荐
inline int max(int a, int b) { return a > b ? a : b; }
// 或
constexpr int max(int a, int b) { return a > b ? a : b; }  // C++11
// 或
template<typename T>
constexpr T max(T a, T b) { return a > b ? a : b; }  // 泛型版本
```

4. 何时仍需使用宏：

```cpp
// 条件编译
#ifdef DEBUG
    #define LOG(msg) std::cout << msg << std::endl
#else
    #define LOG(msg)  // 空宏
#endif

// 文件和行号（编译器内置宏）
#define ASSERT(cond) \
```

```
    if (!(cond)) { \
        std::cerr << "Assertion failed: " << #cond \
                  << " at " << __FILE__ << ":" << __LINE__ << std::endl; \
    }

// 字符串化
#define STRINGIFY(x) #x
```

最佳实践：

1. 常量：优先使用 const 或 constexpr，避免 #define
2. 类型别名：使用 using（C++11）或 typedef，避免 #define
3. 函数：使用 inline 或 constexpr 函数，避免宏
4. 宏：仅在必要时使用（条件编译、特殊操作）
5. 调试：const、typedef、inline 都保留类型信息，更易调试

4.5 size_t 类型

size_t 类型是 C++标准库中定义的一个无符号整数类型，用于表示内存块的大小或数组索引等。它的定义通常是：

```C++
typedef unsigned long long size_t;
```

在 32 位系统上，size_t 通常是 4 字节，在 64 位系统上通常是 8 字节。使用 size_t 而不是 int 或 unsigned int 主要有两个原因：可移植性 和 语义明确。当你的代码中使用 size_t 时，阅读代码的人会立刻明白：

- 这个变量代表一个大小、数量或索引。
- 这个值永远不可能是负数。

4.6 void* 类型

void* 类型是一种通用的指针类型，可以指向任何类型的数据。它没有类型信息，因此不能直接进行解引用操作。需要先进行类型转换才能使用。

```C++
void* p = malloc(10);
int* pi = static_cast<int*>(p);
```

C 语言的 malloc，其原型就是 void* malloc(size_t size);。malloc 也不知道你要用这块内存做什么，所以它也返回一个通用的 void*

4.7 new 和 malloc 的区别

首先 new/delete 是 cpp 的运算符，它可以重载，而 malloc/free 是 c/cpp 的标准库函数，不可以重载。new 无需指定内存块的大小，编译器会自动计算，而 malloc 需要指定内存块的大小。new 返回的是对象的指针，而 malloc 返回的是 void*，需要类型转换。new 会调用对象的构造函数，而 malloc 不会。new 操作符从自由存储区上为对象动态分配内存空间，而 malloc 函数从堆上动态分配内存 new 有类型安全检查，而 malloc 没有类型安全检查。

4.8 constexpr 和 const 的区别

constexpr 只能定义编译期常量，而 const 可以定义编译期常量，也可以定义运行期常量。

```C++
// 基础示例
const int c1 = 42;          // 运行时常量（可能编译期计算）
constexpr int ce = 42;      // 编译期常量

int runtime_val = rand();
const int c2 = runtime_val; // 合法（运行时初始化）
constexpr int ce2 = runtime_val; // 错误：需要编译期可知值
```

特性	const	constexpr
求值时机	运行时常量（可能编译期优化）	必须编译期确定值
初始化要求	可运行时初始化	必须编译期常量表达式
适用对象	变量/成员函数	变量/函数/构造函数
类型限制	无	必须是字面类型（LiteralType）
数组大小声明	C99 变长数组可能允许	始终合法
模板元编程	有限支持	核心工具（编译期计算）

函数应用示例：

```cpp
// constexpr函数（C++11要求单return语句，C++14放宽）
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n-1);
}

constexpr int fact5 = factorial(5);  // 编译期计算120
int dynamic_val = factorial(runtime_val); // 运行时计算
```

类与对象示例：

```cpp
class Point {
public:
    constexpr Point(double x, double y) : x(x), y(y) {}
    constexpr double getX() const { return x; }
private:
    double x, y;
};

constexpr Point origin(0, 0);  // 编译期构造对象
constexpr double x = origin.getX(); // 编译期调用成员函数
```

工程实践建议：

1. 优先使用 **constexpr** 的场景：

```cpp
// 数学常量
constexpr double PI = 3.141592653589793;

// 模板元编程
template <size_t N>
struct ArrayWrapper {
    int data[N];
};

// 替代宏定义的常量
constexpr int MAX_BUFFER_SIZE = 1024;
```

2. **C++17** 起可组合使用：

```cpp
constexpr const char* LOG_PREFIX = "[DEBUG]"; // 编译期常量指针
```

特殊注意：

- constexpr 变量隐式包含 const 属性
- const 指针的常量性：

```cpp
const int* p1;        // 指向常量的指针
int const* p2;        // 同p1
int* const p3 = &var; // 常量指针（需初始化）
constexpr int* p4 = &var; // 编译期常量指针（C++17起）
```

编译期检测机制：

```cpp
// 通过模板参数验证编译期常量
template <int N>
struct MustBeCompileTimeConstant {};

MustBeCompileTimeConstant<ce> valid;   // OK
MustBeCompileTimeConstant<c1> invalid; // 错误（除非c1是constexpr）
```

通过合理使用 constexpr，可以提升代码性能（编译期计算），增强类型安全性，并支持更复杂的模板元编程场景。而 const 主要用于运行时的只读保证和常量语义表达。

4.9 volatile

指令关键字，确保本条指令不会因编译器的优化而省略，且要求每次直接读值，保证对特殊地址的稳定访问。例如：空循环不会被编译器优化。

4.10 前置++与后置++

前置++和后置++操作符有本质区别，主要体现在返回值和性能上。

```cpp
class CustomInt {
public:
    // 前置++（返回引用）
    CustomInt& operator++() {
        ++value;
        return *this;
    }

    // 后置++（返回旧值副本）
    CustomInt operator++(int) { // int参数用于区分重载，在调用时会传入0，因此i++++是不合法的。
        CustomInt temp = *this; // 构造临时对象，产生额外开销
        ++value;
        return temp;
    }

private:
    int value = 0;
};
```

5 函数指针

5.1 为什么需要函数指针？

核心问题：如何在运行时动态决定调用哪个函数？

传统方式下，函数调用是静态的：

```cpp
// 静态调用：编译时就确定调用哪个函数
void processAdd(int a, int b) { std::cout << a + b; }
void processSub(int a, int b) { std::cout << a - b; }

int main() {
    processAdd(5, 3);  // 固定调用加法
    processSub(5, 3);  // 固定调用减法
}
```

问题：如果要根据用户输入或运行时条件选择不同的操作，怎么办？

不好的方案：使用大量 if-else 或 switch

```cpp
void process(int a, int b, int op) {
    if (op == 1) processAdd(a, b);
    else if (op == 2) processSub(a, b);
    else if (op == 3) processMul(a, b);
    // ... 更多操作，代码臃肿
}
```

更好的方案：使用函数指针

```cpp
// 函数指针允许在运行时选择要调用的函数
typedef void (*Operation)(int, int);

void process(int a, int b, Operation op) {
    op(a, b);  // 动态调用，简洁高效
}

// 使用
Operation ops[] = {processAdd, processSub, processMul};
process(5, 3, ops[userChoice]);  // 根据用户选择动态调用
```

函数指针的本质：函数在内存中也有地址，函数指针就是存储这个地址的变量。

5.2 基本语法

1. 函数指针的声明：

```cpp
// 返回类型 (*指针名)(参数类型列表)

// 示例：指向接受两个int、返回void的函数的指针
void (*funcPtr)(int, int);

// 对比：普通函数声明
void func(int, int);

// 记忆技巧：在函数名外面加(*)，func变成(*funcPtr)
```

2. 使用 **typedef** 简化：

```cpp
// 传统方式（难读）
```

```
void (*ptr1)(int, int);

// typedef方式（推荐）
typedef void (*FuncPtr)(int, int);
FuncPtr ptr2;  // 清晰易读

// C++11 using方式（更推荐）
using FuncPtr = void(*)(int, int);
FuncPtr ptr3;
```

3. 函数指针的赋值和调用：

```
void myFunc(int x, int y) {
    std::cout << x + y << std::endl;
}

// 赋值
FuncPtr ptr = myFunc;       // 方式1：隐式转换
FuncPtr ptr2 = &myFunc;     // 方式2：显式取地址（推荐，更清晰）

// 调用
ptr(3, 5);       // 方式1：直接调用
(*ptr)(3, 5);    // 方式2：解引用后调用（更明确）
```

4. 复杂示例：

```
// 返回int、接受两个double的函数指针
int (*funcPtr1)(double, double);

// 返回指针的函数指针
int* (*funcPtr2)(int);  // 返回int*的函数指针

// 函数指针数组
void (*funcArray[10])(int);  // 10个函数指针的数组

// 指向函数指针的指针
void (**ptrPtr)(int);  // 指向函数指针的指针
```

5.3 实际应用场景

1. 回调函数（**Callback**）：

最常见的应用，允许将自定义行为传递给库函数。

```
// 标准库qsort的回调
int compare_int(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int arr[] = {5, 2, 8, 1, 9};
qsort(arr, 5, sizeof(int), compare_int);  // 传递比较函数

// 自定义排序策略
int compare_desc(const void* a, const void* b) {
    return (*(int*)b - *(int*)a);  // 降序
}
```

```
13
14 qsort(arr, 5, sizeof(int), compare_desc);  // 不同的排序行为
```

2. 状态机（**State Machine**）：

使用函数指针数组实现状态转换。

```cpp
enum State { IDLE, RUNNING, PAUSED, STOPPED };

class StateMachine {
private:
    State currentState;

    // 状态处理函数
    void handleIdle() { std::cout << "Idle state\n"; }
    void handleRunning() { std::cout << "Running state\n"; }
    void handlePaused() { std::cout << "Paused state\n"; }
    void handleStopped() { std::cout << "Stopped state\n"; }

    // 函数指针数组（状态表）
    typedef void (StateMachine::*StateHandler)();
    StateHandler stateHandlers[4] = {
        &StateMachine::handleIdle,
        &StateMachine::handleRunning,
        &StateMachine::handlePaused,
        &StateMachine::handleStopped
    };

public:
    void update() {
        (this->*stateHandlers[currentState])();  // 根据状态调用对应处理函数
    }
};
```

3. 命令模式/策略模式：

实现可插拔的算法或操作。

```cpp
// 不同的日志策略
void logToConsole(const std::string& msg) {
    std::cout << msg << std::endl;
}

void logToFile(const std::string& msg) {
    std::ofstream file("log.txt", std::ios::app);
    file << msg << std::endl;
}

void logToNetwork(const std::string& msg) {
    // 发送到远程服务器
}

class Logger {
private:
    using LogFunc = void(*)(const std::string&);
    LogFunc logStrategy;

```

```
public:
    void setStrategy(LogFunc func) {
        logStrategy = func;
    }

    void log(const std::string& msg) {
        if (logStrategy) {
            logStrategy(msg);  // 使用当前策略
        }
    }
};

// 使用
Logger logger;
logger.setStrategy(logToConsole);  // 运行时切换策略
logger.log("Console message");

logger.setStrategy(logToFile);
logger.log("File message");
```

4. 事件处理系统：

GUI 编程中常见的模式。

C++

```
class Button {
private:
    using ClickHandler = void(*)(void);
    ClickHandler onClick;

public:
    void setOnClick(ClickHandler handler) {
        onClick = handler;
    }

    void click() {
        if (onClick) {
            onClick();  // 触发回调
        }
    }
};

// 事件处理函数
void onSaveClick() {
    std::cout << "Save button clicked\n";
}

void onCancelClick() {
    std::cout << "Cancel button clicked\n";
}

// 使用
Button saveBtn, cancelBtn;
saveBtn.setOnClick(onSaveClick);
cancelBtn.setOnClick(onCancelClick);
```

```
saveBtn.click();    // 输出：Save button clicked
cancelBtn.click();  // 输出：Cancel button clicked
```

5. 函数映射表（Jump Table）：

替代冗长的 switch-case。

```cpp
// 计算器示例
double add(double a, double b) { return a + b; }
double subtract(double a, double b) { return a - b; }
double multiply(double a, double b) { return a * b; }
double divide(double a, double b) { return a / b; }

class Calculator {
private:
    using BinaryOp = double(*)(double, double);
    std::map<char, BinaryOp> operations;

public:
    Calculator() {
        operations['+'] = add;
        operations['-'] = subtract;
        operations['*'] = multiply;
        operations['/'] = divide;
    }

    double calculate(char op, double a, double b) {
        auto it = operations.find(op);
        if (it != operations.end()) {
            return it->second(a, b);  // 调用对应函数
        }
        throw std::invalid_argument("Unknown operator");
    }
};

// 使用
Calculator calc;
std::cout << calc.calculate('+', 5, 3);  // 8
std::cout << calc.calculate('*', 5, 3);  // 15
```

6. 动态加载库（DLL/SO）：

运行时加载外部函数。

```cpp
#include <dlfcn.h>  // Linux

// 加载动态库
void* handle = dlopen("libmath.so", RTLD_LAZY);

// 获取函数指针
typedef int (*MathFunc)(int, int);
MathFunc add = (MathFunc)dlsym(handle, "add_function");

// 调用动态加载的函数
int result = add(5, 3);

dlclose(handle);
```

5.4 成员函数指针

成员函数指针语法更复杂，因为需要通过对象调用。

```cpp
class Math {
public:
    int add(int a, int b) { return a + b; }
    int multiply(int a, int b) { return a * b; }
};

// 成员函数指针类型
using MemberFunc = int (Math::*)(int, int);

MemberFunc ptr = &Math::add;  // 必须使用&

Math obj;
int result = (obj.*ptr)(3, 5);  // 通过对象调用：obj.*ptr

Math* pObj = &obj;
result = (pObj->*ptr)(3, 5);    // 通过指针调用：pObj->*ptr
```

5.5 现代 C++的替代方案

虽然函数指针功能强大，但现代 C++提供了更好的选择：

1. std::function（C++11，推荐）：

```cpp
#include <functional>

std::function<int(int, int)> func;

func = add;                    // 可以存储普通函数
func = [](int a, int b) { return a * b; };  // 可以存储lambda
func = std::bind(&Math::add, &obj, std::placeholders::_1, std::placeholders::_2);  // 可以存储成员函数

int result = func(3, 5);
```

2. Lambda 表达式（C++11，更推荐）：

```cpp
// 函数指针方式
void process(int (*func)(int)) {
    std::cout << func(5);
}

// Lambda方式（更简洁）
auto lambda = [](int x) { return x * 2; };
process(lambda);  // lambda可以隐式转换为函数指针（无捕获时）

// 或直接传入
process([](int x) { return x * 2; });
```

3. 模板（编译期多态）：

```cpp
// 函数指针方式（运行时）
void execute(void (*func)()) {
    func();
}
```

```
5
6   // 模板方式（编译期，更高效）
7   template<typename Func>
8   void execute(Func func) {
9       func();
10  }
```

5.6 函数指针 vs 指针函数

容易混淆的两个概念：

```
1   // 函数指针：指向函数的指针                                    C++
2   int (*funcPtr)(int, int);      // funcPtr是指针，指向函数
3   funcPtr = &add;
4   int result = funcPtr(3, 5);    // 通过指针调用函数
5
6   // 指针函数：返回指针的函数
7   int* pointerFunc(int x) {      // pointerFunc是函数，返回int*
8       static int result = x * 2;
9       return &result;
10  }
11  int* ptr = pointerFunc(5);     // 调用函数，得到指针
```

记忆技巧：看*的位置

- `int (*ptr)()` → *在函数名位置，是指针，指向函数 = 函数指针
- `int* func()` → *在返回类型，是函数，返回指针 = 指针函数

5.7 函数指针的优缺点

优点：

1. 运行时灵活性：可以动态改变函数行为
2. 解耦：调用者不需要知道具体实现
3. 回调机制：实现异步操作、事件处理
4. 策略模式：轻松切换算法

缺点：

1. 语法复杂：声明和使用都不直观
2. 类型不安全：容易出错
3. 调试困难：间接调用难以追踪
4. 性能开销：无法内联优化（相比直接调用）

最佳实践：

- 简单场景：使用 lambda 表达式
- 需要存储/传递：使用 `std::function`
- 性能关键：使用模板
- C 接口/兼容性：使用函数指针
- 复杂逻辑：考虑设计模式（策略、命令等）

6 强制类型转换

关键字：static_cast、dynamic_cast、reinterpret_cast 和 const_cast

- static_cast

没有运行时类型检查来保证转换的安全性 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的 进行下行转换（把基类的指针或引用转换为派生类表示），由于没有动态类型检查，所以是不安全的。

```cpp
// 基本类型转换
double d = 3.14;
int i = static_cast<int>(d);

// 类层次向上转换
Base* pb = static_cast<Base*>(&derived);

// 显式构造函数调用
void* p = malloc(sizeof(int));
int* pi = static_cast<int*>(p);

// 枚举与整型转换
enum Color { RED, GREEN };
int color_code = static_cast<int>(RED);
```

- dynamic_cast

在进行下行转换时，dynamic_cast 具有类型检查（信息在虚函数中）的功能，比 static_cast 更安全。转换后必须是类的指针、引用或者 void*，基类要有虚函数，可以交叉转换。dynamic 本身只能用于存在虚函数的父子关系的强制类型转换；对于指针，转换失败则返回 nullptr，对于引用，转换失败会抛出异常。

```cpp
// 安全向下转型
Base* pb = &derived;
Derived* pd = dynamic_cast<Derived*>(pb); // 成功

Base* pbase = new Base;
Derived* pderived = dynamic_cast<Derived*>(pbase); // 返回nullptr

// 引用类型转换（失败时抛出std::bad_cast）
try {
    Derived& rd = dynamic_cast<Derived&>(base);
} catch (const std::bad_cast& e) {
    // 处理转换失败
}
```

- reinterpret_cast

可以将整型转换为指针，也可以把指针转换为数组；可以在指针和引用里进行肆无忌惮的转换，平台移植性比价差。不进行任何类型检查，高度依赖具体实现。

```cpp
// 指针与整数互转
intptr_t addr = reinterpret_cast<intptr_t>(&i);

// 不同类型指针转换
Unrelated* up = reinterpret_cast<Unrelated*>(&derived);

// 函数指针转换
```

```cpp
using FuncPtr = void(*)();
FuncPtr func = reinterpret_cast<FuncPtr>(&dummy);

// 结构体二进制处理
struct Packet { char data[128]; };
Packet pkt;
int32_t* pnum = reinterpret_cast<int32_t*>(pkt.data + 4);
```

- const_cast

不能改变基础类型，常量指针转换为非常量指针，并且仍然指向原来的对象。常量引用被转换为非常量引用，并且仍然指向原来的对象。去掉类型的 const 或 volatile 属性。

```cpp
// 常量指针转换为非常量指针
const int* p = &i;
int* p2 = const_cast<int*>(p);

// 常量引用转换为非常量引用
const int& r = i;
int& r2 = const_cast<int&>(r);

// 修改mutable成员
class C {
    mutable int counter;
public:
    void inc() const {
        const_cast<C*>(this)->counter++;
    }
};
```

7 struct 和 Class 的区别

• 相同点：

如果结构体没有定义任何构造函数，编译器会生成默认的无参数构造函数。如果类没有定义任何构造函数，编译器也会生成默认的无参数构造函数。

• 不同点：

通常，struct 用于表示一组相关的数据，而 class 用于表示一个封装了数据和操作的对象,在实际使用中，可以根据具体的需求选择使用 struct 或 class。如果只是用来组织一些数据，而不涉及复杂的封装和继承关系，struct 可能更直观；如果需要进行封装、继承等面向对象编程的特性，可以选择使用 class。
struct 结构体中的成员默认是公有的（public）。类中的成员默认是私有的（private）。 struct 继承时默认使用公有继承。class 继承时默认使用私有继承

8 C++中的 nullptr 与 NULL 的区别

nullptr 为 C++11 引入的关键字，表示一种特殊的空指针类型，具体为 std::nullptr_t 线程安全类型，这种类型可以隐式转换为任意的指针类型，但不能转换为整数类型。 NULL 是一个宏定义，通常定义为 0 或 `(void*)0`，它的本质还是一个整数常量，可以隐式的转换为指针类型，但可能引发分歧。

9 extern 关键字

extern 关键字用于声明一个变量或函数，不分配内存。它的作用是告诉编译器这个变量或函数在其他文件中定义，。

```cpp
// 文件A.cpp
int x = 10; // 定义全局变量x（分配内存）
// 文件B.cpp
extern int x; // 声明x，链接到A.cpp中的定义
void func() {
    x = 20; // 使用A.cpp中定义的x
}
```

函数声明默认是带 extern 的。

```cpp
// 文件A.h
extern void func(); // 等价于 void func();
// 文件B.cpp
void func() { /* 实现 */ } // 定义函数
```

C++ 与 C 混合编程：

```cpp
extern "C" { /* C函数声明 */ } // 告诉C++编译器按C语言规则链接
```

模板与 extern（C++11+） 显式实例化声明：

```cpp
// 声明：告知编译器某个模板实例已在其他文件中定义
extern template class std::vector<int>;
// 定义（另一个文件中）
template class std::vector<int>;
```

作用： 减少编译时间（避免重复实例化），常用于大型项目。

10 size of

sizeof 是 C++ 中用于获取类型或表达式大小的编译时操作符，其核心价值在于： 内存管理：精确计算数据结构大小，避免内存泄漏 跨平台兼容：处理不同系统的类型大小差异 模板元编程：在编译期进行类型大小的条件判断 性能优化：通过对齐优化减少内存碎片

```cpp
int arr[10];
void func(int arr[]) {
    cout << sizeof(arr) << endl;
}
int main() {
    cout << sizeof(arr) << endl; // 40
    func(arr);  // 8 (64 bit system)
    return 0;
}
```

11 其他重要关键字

11.1 virtual - 虚函数

为什么需要虚函数？

实现运行时多态（动态绑定），让基类指针可以调用派生类的函数。

```cpp
// 没有virtual：静态绑定
class Animal {
public:
    void speak() { std::cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void speak() { std::cout << "Woof!\n"; }  // 隐藏基类函数，不是重写
};

Animal* animal = new Dog();
animal->speak();  // 输出：Animal speaks（调用基类版本）
delete animal;

// 使用virtual：动态绑定
class Animal {
public:
    virtual void speak() { std::cout << "Animal speaks\n"; }
    virtual ~Animal() {}  // 虚析构函数（重要！）
};

class Dog : public Animal {
public:
    void speak() override { std::cout << "Woof!\n"; }  // 重写
};

Animal* animal = new Dog();
animal->speak();  // 输出：Woof!（调用派生类版本）
delete animal;    // 正确调用Dog的析构函数
```

虚函数表（**vtable**）原理：

```cpp
class Base {
    int data;
public:
    virtual void func1() {}
    virtual void func2() {}
    void nonVirtual() {}
};

// 内存布局：
// [vptr指向虚函数表] [data成员]
// 虚函数表：[&Base::func1, &Base::func2]

class Derived : public Base {
public:
    void func1() override {}  // 重写
```

```
    virtual void func3() {}   // 新增虚函数
};

// Derived的虚函数表：[&Derived::func1, &Base::func2, &Derived::func3]
```

纯虚函数（抽象类）：

```C++
class Shape {
public:
    virtual double area() const = 0;  // 纯虚函数
    virtual ~Shape() = default;
};

// Shape s;  // 错误！不能实例化抽象类

class Circle : public Shape {
    double radius;
public:
    double area() const override {
        return 3.14 * radius * radius;
    }
};

Circle c;  // ✓ 正确，Circle实现了所有纯虚函数
```

虚析构函数的重要性：

```C++
class Base {
public:
    ~Base() { std::cout << "~Base\n"; }  // 非虚析构
};

class Derived : public Base {
    int* data;
public:
    Derived() : data(new int[100]) {}
    ~Derived() {
        delete[] data;  // 释放资源
        std::cout << "~Derived\n";
    }
};

Base* ptr = new Derived();
delete ptr;  // 只调用~Base()，内存泄漏！

// 正确做法：
class Base {
public:
    virtual ~Base() { std::cout << "~Base\n"; }  // 虚析构
};

delete ptr;  // 先调用~Derived()，再调用~Base()
```

虚函数的注意事项：

```C++
class Base {
```

```
public:
    virtual void func(int x) {}
};

class Derived : public Base {
public:
    // void func(double x) {}  // 不是重写，是重载（参数不同）
    void func(int x) override {}  // 正确重写，使用override确保
};
```

11.2 explicit - 显式构造函数

为什么需要 **explicit**？

防止隐式类型转换，避免意外的对象构造。

C++

```
// 没有explicit：允许隐式转换
class String {
    char* data;
public:
    String(int size) : data(new char[size]) {}  // 可以隐式转换
};

void process(String s) {}

process(10);  // 隐式转换：10 → String(10)，可能不是预期行为

// 使用explicit：禁止隐式转换
class String {
    char* data;
public:
    explicit String(int size) : data(new char[size]) {}
};

// process(10);  // 错误！不能隐式转换
process(String(10));  // ✓ 显式构造，意图明确
```

实际应用场景：

C++

```
class Vector {
    double x, y, z;
public:
    // 单参数构造函数应该用explicit
    explicit Vector(double val) : x(val), y(val), z(val) {}

    // 多参数构造函数不需要explicit（不会隐式转换）
    Vector(double x, double y, double z) : x(x), y(y), z(z) {}
};

void move(Vector v) {}

// move(5.0);  // 错误！explicit阻止隐式转换
move(Vector(5.0));  // ✓ 显式构造
move(Vector(1, 2, 3));  // ✓ 多参数构造
```

explicit 与拷贝/移动构造：

```cpp
class MyClass {
public:
    explicit MyClass(int x) {}

    // 拷贝构造通常不应该explicit
    MyClass(const MyClass& other) {}  // 允许：MyClass b = a;

    // 转换构造应该explicit
    explicit MyClass(const std::string& s) {}
};

MyClass a(10);
MyClass b = a;  // ✓ 拷贝构造，允许
// MyClass c = "hello";  // 错误！explicit阻止
MyClass d("hello");  // ✓ 显式构造
```

11.3 delete/default (C++11)

= delete：禁用函数

```cpp
class NonCopyable {
public:
    NonCopyable() = default;

    // 禁用拷贝构造和拷贝赋值
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    // 允许移动
    NonCopyable(NonCopyable&&) = default;
    NonCopyable& operator=(NonCopyable&&) = default;
};

NonCopyable obj1;
// NonCopyable obj2 = obj1;  // 错误！拷贝被禁用
NonCopyable obj3 = std::move(obj1);  // ✓ 移动允许
```

禁用特定重载：

```cpp
class SmartPtr {
public:
    SmartPtr(int* p) {}

    // 禁止接受void*（防止类型不安全）
    SmartPtr(void*) = delete;

    // 禁止bool转换（防止意外）
    SmartPtr(bool) = delete;
};

int* p = new int(42);
SmartPtr sp1(p);  // ✓
// SmartPtr sp2(nullptr);  // 错误！void*被禁用
```

= default：显式要求默认实现

```cpp
class MyClass {
```

```
public:
    MyClass() = default;  // 显式要求默认构造函数
    ~MyClass() = default;

    // 即使声明了其他构造函数，也保留默认构造
    MyClass(int x) : value(x) {}

    // 显式要求编译器生成
    MyClass(const MyClass&) = default;
    MyClass(MyClass&&) = default;
    MyClass& operator=(const MyClass&) = default;
    MyClass& operator=(MyClass&&) = default;

private:
    int value = 0;
};
```

Rule of Five/Zero：

```
// Rule of Zero：不管理资源，全部用default
class Simple {
    std::string name;
    std::vector<int> data;
public:
    // 编译器自动生成所有特殊成员函数，完美工作
};

// Rule of Five：管理资源，全部显式定义或delete
class ResourceOwner {
    int* data;
public:
    ResourceOwner() : data(new int[100]) {}
    ~ResourceOwner() { delete[] data; }

    // 必须定义或delete所有五个
    ResourceOwner(const ResourceOwner&);
    ResourceOwner& operator=(const ResourceOwner&);
    ResourceOwner(ResourceOwner&&) noexcept;
    ResourceOwner& operator=(ResourceOwner&&) noexcept;
};
```

11.4 decltype 和 auto (C++11 类型推导)

auto：自动类型推导

```
// 基本用法
auto x = 42;              // int
auto y = 3.14;            // double
auto s = std::string("hello");  // std::string

// 与指针和引用
int value = 10;
auto* ptr = &value;       // int*
auto& ref = value;        // int&

const int cx = 10;
```

```cpp
auto a = cx;              // int（丢失顶层const）
const auto b = cx;        // const int
auto& c = cx;             // const int&（保留底层const）

// 迭代器简化
std::vector<int> vec = {1, 2, 3};
// std::vector<int>::iterator it = vec.begin();  // 繁琐
auto it = vec.begin();    // 简洁

// 与范围for循环
for (auto& elem : vec) {  // 引用，可修改
    elem *= 2;
}

for (const auto& elem : vec) {  // const引用，只读
    std::cout << elem;
}
```

decltype：获取表达式类型

```cpp
int x = 10;
decltype(x) y = 20;       // y的类型是int

const int& ref = x;
decltype(ref) ref2 = y;   // ref2的类型是const int&

// decltype vs auto
auto a = ref;             // a是int（丢失引用和const）
decltype(ref) b = ref;    // b是const int&（保留所有）

// 函数返回类型推导
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) {  // C++11尾置返回类型
    return t + u;
}

// C++14简化
template<typename T, typename U>
auto add(T t, U u) {
    return t + u;  // 自动推导返回类型
}
```

decltype 的特殊规则：

```cpp
int x = 10;
decltype(x) y;      // y是int
decltype((x)) z = x;  // z是int&（加括号变成引用）

int arr[5];
decltype(arr) arr2;  // arr2是int[5]
auto arr3 = arr;     // arr3是int*（数组退化）
```

实际应用：

```cpp
// 泛型编程
```

```cpp
template<typename Container>
void process(Container& c) {
    // 自动推导元素类型
    for (auto& elem : c) {
        // ...
    }

    // 获取迭代器类型
    using Iterator = decltype(c.begin());
}

// 完美转发的返回类型
template<typename Func, typename... Args>
auto call_function(Func f, Args&&... args)
    -> decltype(f(std::forward<Args>(args)...)) {
    return f(std::forward<Args>(args)...);
}
```

11.5 friend - 友元

为什么需要友元？

允许外部函数或类访问私有成员，打破封装用于特殊场景。

友元函数：

C++

```cpp
class Complex {
private:
    double real, imag;

public:
    Complex(double r, double i) : real(r), imag(i) {}

    // 友元函数：可以访问私有成员
    friend Complex operator+(const Complex& a, const Complex& b);
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
};

// 友元函数实现（不是成员函数）
Complex operator+(const Complex& a, const Complex& b) {
    return Complex(a.real + b.real, a.imag + b.imag);  // 访问私有成员
}

std::ostream& operator<<(std::ostream& os, const Complex& c) {
    return os << c.real << " + " << c.imag << "i";
}

Complex c1(1, 2), c2(3, 4);
Complex c3 = c1 + c2;  // 使用友元运算符
std::cout << c3;       // 使用友元输出运算符
```

友元类：

C++

```cpp
class Engine;  // 前向声明

class Car {
private:
```

```
    int speed;
    Engine* engine;

public:
    // Engine是Car的友元类
    friend class Engine;
};

class Engine {
public:
    void boost(Car& car) {
        car.speed += 20;  // 可以访问Car的私有成员
    }
};
```

友元成员函数：

```C++
class Display;

class Data {
private:
    int value;

public:
    Data(int v) : value(v) {}

    // 只让Display的特定成员函数成为友元
    friend void Display::show(const Data& d);
};

class Display {
public:
    void show(const Data& d) {
        std::cout << d.value;  // 可以访问Data::value
    }
};
```

友元的注意事项：

```C++
// 1. 友元不具有传递性
class A {
    friend class B;
    int data;
};

class B {
    friend class C;
    void accessA(A& a) { a.data = 10; }  // ✓ B是A的友元
};

class C {
    void accessA(A& a) {
        // a.data = 20;  // ✗ C不是A的友元
    }
};

```

```cpp
// 2. 友元不能被继承
class Base {
    friend class Friend;
    int data;
};

class Derived : public Base {
    // Friend不能访问Derived新增的私有成员
};

// 3. 友元破坏封装，谨慎使用
// 仅在必要时使用：运算符重载、工厂模式、测试类等
```

11.6 override 和 final (C++11)

override：明确标记重写

```cpp
class Base {
public:
    virtual void func1() {}
    virtual void func2(int x) {}
    virtual void func3() const {}
};

class Derived : public Base {
public:
    void func1() override {}  // ✓ 正确重写

    // void func2(double x) override {}  // ✗ 编译错误！参数不匹配
    void func2(int x) override {}  // ✓ 正确重写

    // void func3() override {}  // ✗ 编译错误！缺少const
    void func3() const override {}  // ✓ 正确重写
};

// 没有override的问题：
class Problem : public Base {
    void func1() {}  // 实际是重写，但如果拼写错误编译器不会报错
    void fucn1() {}  // 拼写错误！编译器认为这是新函数
};
```

final：禁止重写/继承

```cpp
// 1. 禁止类被继承
class FinalClass final {
public:
    virtual void func() {}
};

// class Derived : public FinalClass {};  // ✗ 错误！

// 2. 禁止虚函数被重写
class Base {
public:
    virtual void canOverride() {}
    virtual void cannotOverride() final {}
```

```
};

class Derived : public Base {
public:
    void canOverride() override {}  // ✓ 允许
    // void cannotOverride() override {}  // ✗ 错误！final禁止重写
};
```

11.7 noexcept (C++11)

为什么需要 **noexcept**？

```C++
// 1. 性能优化：编译器可以做更多优化
void fast_function() noexcept {
    // 编译器知道不会抛异常，可以优化
}

// 2. 移动语义的关键
class MyVector {
public:
    // 没有noexcept：vector扩容时会拷贝（安全但慢）
    MyVector(MyVector&& other) { /*...*/ }

    // 有noexcept：vector扩容时会移动（快）
    MyVector(MyVector&& other) noexcept { /*...*/ }
};

std::vector<MyVector> vec;
vec.push_back(MyVector());  // 触发扩容
// 如果移动构造有noexcept：使用移动
// 如果移动构造没有noexcept：使用拷贝（异常安全）
```

条件 **noexcept**：

```C++
template<typename T>
class Container {
public:
    // 根据T的属性决定是否noexcept
    Container(Container&& other)
        noexcept(std::is_nothrow_move_constructible<T>::value) {
        // ...
    }
};
```

检测是否 **noexcept**：

```C++
void may_throw() {}
void no_throw() noexcept {}

static_assert(noexcept(no_throw()), "should be noexcept");
// static_assert(noexcept(may_throw()), "");  // 失败

// 实际应用
template<typename T>
void smart_swap(T& a, T& b) noexcept(noexcept(std::swap(a, b))) {
    std::swap(a, b);
}
```

11.8 this 指针

this 的本质：

```cpp
class MyClass {
    int value;
public:
    void setValue(int value) {
        this->value = value;  // 区分成员和参数
    }

    MyClass& returnSelf() {
        return *this;  // 返回自身引用，支持链式调用
    }

    // 编译器实际看到的：
    // void setValue(MyClass* const this, int value) {
    //     this->value = value;
    // }
};
```

链式调用：

```cpp
class Builder {
    std::string name;
    int age;

public:
    Builder& setName(const std::string& n) {
        name = n;
        return *this;  // 返回自身
    }

    Builder& setAge(int a) {
        age = a;
        return *this;
    }

    void build() { /*...*/ }
};

Builder b;
b.setName("Alice").setAge(25).build();  // 链式调用
```

this 的类型：

```cpp
class MyClass {
    void normalFunc() {
        // this的类型：MyClass* const
        // 可以修改成员
    }

    void constFunc() const {
        // this的类型：const MyClass* const
        // 不能修改成员
    }
};
```

返回 this 进行比较：

```cpp
class MyClass {
    int value;
public:
    bool operator==(const MyClass& other) const {
        return this->value == other.value;
    }

    MyClass& operator=(const MyClass& other) {
        if (this != &other) {  // 自赋值检查
            value = other.value;
        }
        return *this;
    }
};
```

11.9 using 声明和 using 指示

using 声明：

```cpp
namespace MyLib {
    void func() {}
    int value = 42;
}

// 引入特定名称
using MyLib::func;
func();  // 可以直接使用

// using MyLib::value;
// int value = 10;  // 错误！命名冲突
```

using 指示（避免使用）：

```cpp
// ✗ 不推荐：污染命名空间
using namespace std;
vector<int> vec;  // 可能与其他库冲突

// ✓ 推荐：限定作用域
void func() {
    using namespace std;  // 仅在函数内有效
    vector<int> vec;
}

// ✓ 最推荐：显式指定
std::vector<int> vec;
```

using 类型别名（C++11）：

```cpp
// 替代typedef，更清晰
using IntPtr = int*;
using FuncPtr = void(*)(int, int);

// 模板别名（typedef做不到）
template<typename T>
using Vec = std::vector<T>;
```

```
8
9 Vec<int> v1;  // 等价于std::vector<int>
```

11.10 namespace - 命名空间

基本用法：

```C++
1  namespace MyLib {
2      class MyClass {};
3      void func() {}
4
5      namespace Internal {  // 嵌套命名空间
6          void helper() {}
7      }
8  }
9
10 MyLib::MyClass obj;
11 MyLib::func();
12 MyLib::Internal::helper();
13
14 // C++17简化嵌套
15 namespace MyLib::Internal {
16     void helper() {}
17 }
```

匿名命名空间：

```C++
1 // 替代static全局变量（更现代）
2 namespace {
3     int internal_var = 42;  // 仅在本文件可见
4     void internal_func() {}
5 }
```

别名：

```C++
1 namespace VeryLongNamespaceName {
2     void func() {}
3 }
4
5 namespace Short = VeryLongNamespaceName;
6 Short::func();
```

12 异常处理

12.1 为什么需要异常处理？

传统错误处理的问题：

```cpp
// 方式1：返回错误码（繁琐且容易被忽略）
int openFile(const char* filename) {
    // 返回0表示成功，-1表示失败
    if (/* 文件不存在 */) return -1;
    if (/* 权限不足 */) return -2;
    if (/* 内存不足 */) return -3;
    return 0;
}

int result = openFile("data.txt");
if (result != 0) {
    // 处理错误
}

// 方式2：全局错误变量（线程不安全）
int errno_global;
void processData() {
    if (/* 错误 */) {
        errno_global = ERROR_CODE;
        return;
    }
}

// 方式3：输出参数（不直观）
bool readData(char* buffer, int* errorCode);
```

问题：

1. 错误码容易被忽略
2. 错误处理代码与正常逻辑混在一起
3. 无法跨多层函数传递错误
4. 构造函数无法返回错误码

异常处理的优势：

```cpp
class File {
public:
    File(const std::string& filename) {
        if (!open(filename)) {
            throw std::runtime_error("Cannot open file");  // 构造失败直接抛异常
        }
    }
};

try {
    File f("data.txt");  // 失败会自动传播
    processFile(f);
    saveFile(f);
} catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
```

12.2 基本语法

try-catch-throw：

```cpp
#include <exception>
#include <stdexcept>

// 1. 抛出异常
void divide(int a, int b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero");
    }
    std::cout << a / b << std::endl;
}

// 2. 捕获异常
void process() {
    try {
        divide(10, 0);  // 抛出异常
        std::cout << "This won't execute\n";  // 不会执行
    } catch (const std::invalid_argument& e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }
    std::cout << "Continue execution\n";  // 继续执行
}

// 3. 多个catch块
try {
    // 可能抛出多种异常的代码
} catch (const std::invalid_argument& e) {
    // 处理invalid_argument
} catch (const std::runtime_error& e) {
    // 处理runtime_error
} catch (const std::exception& e) {
    // 处理其他std::exception
} catch (...) {
    // 捕获所有异常（包括非标准异常）
    std::cerr << "Unknown exception\n";
}
```

异常传播：

```cpp
void funcA() {
    throw std::runtime_error("Error in A");
}

void funcB() {
    funcA();  // 不捕获，继续传播
}

void funcC() {
    try {
        funcB();  // 在这里捕获
    } catch (const std::exception& e) {
        std::cout << "Caught in C: " << e.what() << std::endl;
    }
```

```
}
```

重新抛出异常：

```cpp
void process() {
    try {
        // 某些操作
    } catch (std::exception& e) {
        std::cerr << "Logging: " << e.what() << std::endl;
        throw;  // 重新抛出当前异常（保持原始类型）
    }
}

// 或者抛出新异常
try {
    // 操作
} catch (const std::exception& e) {
    throw std::runtime_error("New error: " + std::string(e.what()));
}
```

12.3 标准异常类层次结构

```cpp
// 标准异常继承体系
std::exception
├── std::bad_alloc            // new失败
├── std::bad_cast             // dynamic_cast失败
├── std::bad_typeid           // typeid失败
├── std::bad_exception        // 意外异常
├── std::logic_error          // 逻辑错误（程序错误）
│   ├── std::invalid_argument    // 无效参数
│   ├── std::domain_error        // 数学域错误
│   ├── std::length_error        // 长度错误
│   ├── std::out_of_range        // 越界
│   └── std::future_error        // future错误
└── std::runtime_error        // 运行时错误（外部因素）
    ├── std::range_error         // 范围错误
    ├── std::overflow_error      // 溢出错误
    ├── std::underflow_error     // 下溢错误
    └── std::system_error        // 系统错误
```

使用标准异常：

```cpp
#include <stdexcept>

void validateAge(int age) {
    if (age < 0 || age > 150) {
        throw std::out_of_range("Age out of valid range");
    }
}

void allocateMemory(size_t size) {
    if (size > MAX_SIZE) {
        throw std::length_error("Size too large");
    }
}
```

```
14
15 std::vector<int> vec = {1, 2, 3};
16 try {
17     int val = vec.at(10);  // 抛出std::out_of_range
18 } catch (const std::out_of_range& e) {
19     std::cerr << e.what() << std::endl;
20 }
```

自定义异常类：

```cpp
1  // 继承std::exception
2  class MyException : public std::exception {
3  private:
4      std::string message;
5
6  public:
7      explicit MyException(const std::string& msg) : message(msg) {}
8
9      const char* what() const noexcept override {
10         return message.c_str();
11     }
12 };
13
14 // 更好的方式：继承适当的标准异常
15 class FileNotFoundException : public std::runtime_error {
16 public:
17     explicit FileNotFoundException(const std::string& filename)
18         : std::runtime_error("File not found: " + filename) {}
19 };
20
21 // 使用
22 try {
23     throw FileNotFoundException("data.txt");
24 } catch (const std::runtime_error& e) {
25     std::cerr << e.what() << std::endl;
26 }
```

12.4 异常安全性保证

三个级别的异常安全：

```cpp
1  // 1. 基本保证（Basic Guarantee）
2  // 异常发生后，程序仍处于有效状态，无资源泄漏，但对象状态可能改变
3  class Basic {
4      std::vector<int> data;
5  public:
6      void append(int value) {
7          data.push_back(value);  // 可能抛异常
8          // 如果抛异常，data保持原状，但可能容量已改变
9      }
10 };
11
12 // 2. 强保证（Strong Guarantee）
13 // 异常发生后，程序状态回滚到操作前（commit-or-rollback）
14 class Strong {
15     std::vector<int> data;
```

```
public:
    void append(int value) {
        std::vector<int> temp = data;  // 复制
        temp.push_back(value);  // 在副本上操作
        data = std::move(temp);  // 不抛异常的操作
        // 要么完全成功，要么data完全不变
    }
};

// 3. 不抛异常保证（No-throw Guarantee）
// 承诺不抛出异常
class NoThrow {
    int* data;
public:
    void swap(NoThrow& other) noexcept {
        std::swap(data, other.data);  // 不会抛异常
    }

    ~NoThrow() noexcept {
        delete data;  // 析构函数不应抛异常
    }
};
```

Copy-and-Swap 惯用法：

```
class Widget {
    int* data;
    size_t size;

public:
    // 拷贝赋值运算符（强异常安全）
    Widget& operator=(const Widget& other) {
        Widget temp(other);  // 拷贝构造（可能抛异常）
        swap(temp);          // 交换（不抛异常）
        return *this;
    }  // temp析构，释放旧资源

    void swap(Widget& other) noexcept {
        std::swap(data, other.data);
        std::swap(size, other.size);
    }
};
```

12.5 RAII 与异常安全

RAII（Resource Acquisition Is Initialization）：

资源获取即初始化，利用对象生命周期管理资源。

```
// 手动管理资源（危险）
void bad_example() {
    int* ptr = new int[100];
    processData(ptr);  // 如果抛异常，内存泄漏！
    delete[] ptr;
}
```

```cpp
// RAII方式（安全）
void good_example() {
    std::unique_ptr<int[]> ptr(new int[100]);
    processData(ptr.get());  // 即使抛异常，ptr也会自动释放
}  // 自动释放资源

// 自定义RAII类
class FileHandle {
    FILE* file;
public:
    FileHandle(const char* filename) : file(fopen(filename, "r")) {
        if (!file) {
            throw std::runtime_error("Cannot open file");
        }
    }

    ~FileHandle() {
        if (file) {
            fclose(file);  // 自动关闭文件
        }
    }

    // 禁用拷贝
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    FILE* get() { return file; }
};

// 使用
void processFile() {
    FileHandle fh("data.txt");  // 自动打开
    // 使用文件
    // 即使抛异常，也会自动关闭
}  // 自动关闭
```

标准库 **RAII** 工具：

```cpp
// 1. 智能指针
{
    std::unique_ptr<int> p1(new int(42));
    std::shared_ptr<int> p2 = std::make_shared<int>(42);
    // 自动释放
}

// 2. 容器
{
    std::vector<int> vec(1000);
    // 自动释放内存
}

// 3. 互斥锁
{
    std::mutex mtx;
```

```
    std::lock_guard<std::mutex> lock(mtx);  // 自动加锁
    // 临界区
}  // 自动解锁，即使抛异常

// 4. 文件流
{
    std::ifstream file("data.txt");  // 自动打开
    // 读取数据
}  // 自动关闭
```

12.6 函数 try 块

普通函数的 try 块：

```cpp
void func() try {
    // 函数体
} catch (const std::exception& e) {
    // 异常处理
}
```

构造函数初始化列表的异常：

```cpp
class MyClass {
    std::string name;
    std::vector<int> data;

public:
    // 处理初始化列表中的异常
    MyClass(const std::string& n, size_t size)
    try : name(n), data(size) {  // 初始化列表可能抛异常
        // 构造函数体
    } catch (const std::bad_alloc& e) {
        std::cerr << "Memory allocation failed\n";
        throw;  // 必须重新抛出或抛出新异常
    }

    // 注意：catch块中必须抛出异常，因为对象构造失败
};
```

12.7 noexcept 说明符

基本用法：

```cpp
// 承诺不抛异常
void safe_function() noexcept {
    // 如果抛异常，调用std::terminate()
}

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

// 检测是否noexcept
```

```
15 void func1() {}
16 void func2() noexcept {}
17
18 static_assert(!noexcept(func1()), "func1 may throw");
19 static_assert(noexcept(func2()), "func2 is noexcept");
```

何时使用 **noexcept**：

```C++
1  class MyClass {
2  public:
3      // 1. 析构函数（默认就是noexcept）
4      ~MyClass() noexcept {
5          // 析构函数不应抛异常
6      }
7
8      // 2. 移动操作（重要！）
9      MyClass(MyClass&&) noexcept;
10     MyClass& operator=(MyClass&&) noexcept;
11
12     // 3. swap函数
13     void swap(MyClass& other) noexcept {
14         std::swap(data, other.data);
15     }
16
17     // 4. 简单的getter
18     int getValue() const noexcept { return value; }
19
20 private:
21     int value;
22     int* data;
23 };
```

noexcept 与性能：

```C++
1  class Widget {
2  public:
3      Widget(Widget&&) noexcept;  // 有noexcept
4  };
5
6  std::vector<Widget> vec;
7  vec.push_back(Widget());  // 触发扩容
8
9  // 如果移动构造是noexcept：
10 //    - vector使用移动（快）
11 // 如果移动构造不是noexcept：
12 //    - vector使用拷贝（慢，但异常安全）
```

12.8 异常处理的性能考虑

零开销原则：

```C++
1  // 现代C++异常实现：
2  // - 不抛异常时：几乎零开销
3  // - 抛异常时：有显著开销
4
5  void normal_path() noexcept {
```

```
    // 正常执行路径，无异常开销
}

void exceptional_path() {
    throw std::exception();  // 抛异常时开销大
}

// 因此：异常应用于真正的"异常"情况，不用于正常控制流
```

不要用异常做控制流：

```C++
// ✗ 错误：用异常控制循环
try {
    for (int i = 0; ; ++i) {
        if (i >= vec.size()) throw std::out_of_range("");
        process(vec[i]);
    }
} catch (const std::out_of_range&) {
    // 正常退出
}

// ✓ 正确：用正常控制流
for (size_t i = 0; i < vec.size(); ++i) {
    process(vec[i]);
}
```

12.9 异常处理最佳实践

1. 按 **const** 引用捕获：

```C++
// ✗ 错误：按值捕获（对象切片）
try {
    throw DerivedEx();
} catch (BaseEx e) {  // 切片！丢失派生类信息
    // ...
}

// ✓ 正确：按const引用捕获
try {
    throw DerivedEx();
} catch (const BaseEx& e) {  // 保持多态
    // ...
}
```

2. 从特殊到一般的顺序捕获：

```C++
try {
    // ...
} catch (const std::bad_alloc& e) {      // 最特殊
    // 内存分配失败
} catch (const std::runtime_error& e) {  // 较一般
    // 运行时错误
} catch (const std::exception& e) {      // 最一般
    // 其他标准异常
} catch (...) {                          // 捕获所有
    // 未知异常
}
```

```
}
```

3. 不要在析构函数中抛异常：

```C++
class Bad {
public:
    ~Bad() {
        // throw std::exception();  // 绝对不要！
        // 如果析构时已有异常在传播，会调用std::terminate()
    }
};

// 正确做法：捕获并处理
class Good {
public:
    ~Good() noexcept {
        try {
            cleanup();  // 可能抛异常
        } catch (...) {
            // 记录错误，但不重新抛出
            std::cerr << "Cleanup failed\n";
        }
    }
};
```

4. 异常安全的代码结构：

```C++
class TransactionManager {
    Database& db;

public:
    void executeTransaction() {
        db.beginTransaction();  // 开始事务

        try {
            db.insertRecord(data1);
            db.updateRecord(data2);
            db.deleteRecord(data3);
            db.commit();  // 提交
        } catch (...) {
            db.rollback();  // 回滚
            throw;  // 重新抛出
        }
    }
};
```

5. 自定义异常应继承 **std::exception**：

```C++
// ✓ 推荐
class MyException : public std::runtime_error {
public:
    explicit MyException(const std::string& msg)
        : std::runtime_error(msg) {}
};

// ✗ 不推荐：抛出基本类型
```

```
    // throw "Error";  // C字符串
    // throw 42;       // 整数
```

6. 使用智能指针避免资源泄漏：

```C++
void risky_function() {
    std::unique_ptr<Resource> res(new Resource());

    process(res.get());  // 可能抛异常

    // 即使抛异常，res也会自动释放
}
```

7. 记录和重新抛出：

```C++
void log_and_rethrow() {
    try {
        dangerous_operation();
    } catch (const std::exception& e) {
        // 记录错误
        logger.error("Operation failed: {}", e.what());
        // 添加上下文后重新抛出
        throw std::runtime_error(
            std::string("In log_and_rethrow: ") + e.what()
        );
    }
}
```