C++新八股文

面向对象基础

C和C++的区别?

C++是在C语言基础上发展而来的多范式语言，包含更多特性和抽象机制。

主要区别：

特性	C	C++
编程范式	面向过程	面向过程 + 面向对象 + 泛型 + 函数式
封装	结构体（无访问控制）	类（public/private/protected）
继承	不支持	支持（单继承/多继承）
多态	函数指针模拟	虚函数、重载、模板
内存管理	malloc/free	new/delete + 智能指针
类型安全	弱类型	强类型（更严格）
标准库	基础库（stdio.h）	STL（容器/算法/迭代器）
引用	不支持	支持（左值/右值引用）
命名空间	不支持	namespace
异常处理	setjmp/longjmp	try/catch/throw
泛型编程	宏 + void*	模板（编译时类型安全）

C++的**核心扩展**：

1. 面向对象特性

- 封装：class/struct的访问控制
- 继承：代码复用和层次结构
- 多态：virtual函数、运行时绑定

2. 泛型编程

- 模板（template）：类型参数化
- STL：通用容器和算法

3. 现代C++特性（C++11后）

- 智能指针：自动内存管理
- 右值引用：移动语义
- lambda表达式
- 原子操作（atomic）
- 协程（C++20）

兼容性：

- C代码可以在C++中编译（大部分情况）
- C++提供 `extern "C"` 与C代码互操作

适用场景：

- 选C：嵌入式、系统编程、对性能和体积极致要求
- 选C++：大型应用、需要OOP/泛型/STL的场景

C++面向对象的三大特性

封装：隐藏实现细节，实现模块化。控制访问权限，private仅对自己和友元开放，protected开放给子类，public开放给所有对象。

继承：无需修改原有类的情况下实现对功能的扩展。存在三种继承，即private继承，protected继承和public继承，解决基类在子类中最高权限的问题(即基类中是public，子类中为private，则权限在子类中修改为private级别，也可以使用using去修改权限)，还可以做多继承和接口继承。

多态：一个接口多种形态，通过实现接口重用增加可扩展性。分为静态多态和动态多态。

- 静态多态：发生在编译期,主要包括函数重载和模板。
- 动态多态：发生在运行期,通过虚函数实现。基类中将成员函数声明为 `virtual`, 派生类重写该函数, 并通过基类指针或引用调用, 这样在运行时会根据对象的实际类型选择调用哪个版本。

C++多态的实现

C++ 多态分为静态多态和动态多态。

静态多态发生在编译期,主要包括函数重载和模板。

- 函数重载: 同一作用域下,函数名相同,参数列表不同,编译器在编译期根据实参类型/数量选择合适的重载版本。
- 函数模板: 使用类型参数编写一份通用代码,编译器在编译期根据实参类型对模板进行实例化,生成不同的函数实现。

动态多态发生在运行期,通过虚函数实现。基类中将成员函数声明为 `virtual`, 派生类重写该函数, 并通过基类指针或引用调用, 这样在运行时会根据对象的实际类型选择调用哪个版本。

其本质是**晚绑定**:

- 非虚函数在编译期就确定调用地址(早绑定);
- 虚函数通过虚函数表(vtable)实现, 对象中有一个虚表指针(vptr), 在构造时初始化, 运行时通过 vptr 在虚表中查找实际要调用的函数地址。

C++ 的构造函数能否定义为虚函数?

不能,语法上就不允许给构造函数加 `virtual`。

原因:

- 虚函数依赖虚函数表 (vtable) 和虚表指针 (vptr) 实现运行时多态。vptr 是在对象构造过程中由构造函数负责初始化的,也就是说,*<u>***对象还没完全构造好之前,就谈"根据动态类型做虚派发"在语义上说不通***</u>*。
- 构造顺序是自上而下: 先调用基类构造函数,再依次调用派生类构造函数。在执行基类构造函数时,对象只能被当作"基类子对象"来看待,此时即使有虚派发,也只会调用基类版本,达不到"根据派生类型选择构造函数"的效果。

因此 C++ 标准直接禁止"虚构造函数"。如果需要"多态创建对象",通常通过*<u>*工厂函数/虚 clone 接口*</u>*等模式来实现。

虚函数和纯虚函数的区别?

- 虚函数: 用 `virtual` 声明，可有默认实现，派生类可选重写，类可实例化。运行时通过基类指针/引用动态绑定到实际类型的函数。
- 纯虚函数: 声明后加 `=0`，无默认实现，强制派生类重写。包含纯虚函数的类是抽象类，不可实例化，用于定义接口规范。

虚析构函数的作用?

核心作用: 确保通过基类指针删除派生类对象时，能正确调用派生类析构函数，避免资源泄漏。

原理: 非虚析构时，`delete` 基类指针只调用基类析构函数 → 派生类资源无法释放。虚析构通过动态绑定，保证先调用派生类析构，再调用基类析构。

适用场景:

- 多态基类（通过基类指针管理派生类对象）
- 抽象接口类
- 派生类含动态资源

继承下的构造函数和析构函数执行顺序?

构造顺序（从上到下）:

1. 基类构造函数
2. 派生类成员对象构造函数（按声明顺序）
3. 派生类自身构造函数

析构顺序（与构造相反）:

1. 派生类自身析构函数
2. 派生类成员对象析构函数（按声明逆序）
3. 基类析构函数

原理: 确保派生类使用基类资源前，基类已完成初始化；销毁时先释放派生类资源，再释放基类资源。

多继承情况: 按继承列表顺序依次构造基类，析构时逆序。

虚函数表和虚指针的创建时机?

- **虚函数表**: 编译期生成。编译器检测到 `virtual` 关键字时为类生成虚表（函数指针数组）。派生类重写虚函数时更新表中地址。
- **虚指针**: 运行期对象构造时初始化。每个对象独立拥有 vptr，位于对象内存布局起始位置，指向所属类的虚表。

内存管理

讲一下循环引用如何发生的，以及如何解决?

对象相互引用时（如双向链表、图结构），会导致引用计数无法归零，资源无法释放。使用weak_ptr打破循环引用，因为它不增加引用计数。

shared_ptr是线程安全的吗？多线程中使用智能指针要注意什么?

- 多线程代码操作的是同一个shared_ptr对象是**线程危险**的。
- 多线程代码操作的不是同一个shared_ptr对象，但不同的shared_ptr指向了相同的内存，此时是线程安全的。

何时用shared_ptr，何时用weak_ptr?

shared_ptr：需要共享资源所有权时使用。

weak_ptr：作为shared_ptr的辅助指针，用于两种场景：

- 打破循环引用
- 需要访问共享资源但不影响其生命周期

weak_ptr如何升级为shared_ptr?

weak_ptr 通过 lock() 方法安全升级为 shared_ptr，确保访问对象时其未被销毁。

```
std::weak_ptr<A> weak_a = ...;  // 从某处获取 weak_ptr

if (auto shared_a = weak_a.lock()) {  // 尝试升级

  shared_a->do_something();    // 对象存活，安全访问

} else {

  // 对象已销毁，避免悬垂指针

}
```

unique指针在编译期如何保证是真的unique?

unique_ptr实现真的unique是靠的以下几个机制：

- **禁用拷贝语义**(核心)：unique_ptr 内部将拷贝构造函数和拷贝赋值运算符声明为 = delete，直接禁止复制行为。
- 仅支持移动语义：unique_ptr 允许通过移动操作转移所有权，转移后原指针变为 nullptr。临时右值可隐式移动。

```
std::unique_ptr<int> p3 = std::unique_ptr<int>(new int(10)); // 合法
```

- 编译器的静态检查:类型系统强制约束,如当尝试拷贝时，编译器检查到调用了被删除的函数，直接报错。

new和malloc的区别?

- **性质**：new是C++操作符，malloc是C语言函数
- **初始化**：new调用构造函数初始化对象，malloc返回未初始化内存
- **语法**：new无需指定大小（如`new int`），malloc需要（如`malloc(sizeof(int))`）
- **返回类型**：new返回具体类型指针，malloc返回void*需强转
- **错误处理**：new抛出std::bad_alloc异常，malloc返回null
- **配对操作**：new配delete，malloc配free
- **实现**：**malloc**的实现核心是通过操作系统提供的系统调用管理堆内存。使用分配的内存块头部存储元数据，通过链表链接所有空闲块。**new**是C++运算符，其行为包含内存分配和对象构造两阶段，内存分配阶段调用全局operator new函数，默认实现内部调用malloc。对象构造阶段用placement new在已分配内存上调用构造函数。new直接返回响应的数据类型的指针。

C++堆和栈的区别?

- 堆：堆上的内存是**动态分配**的，程序在运行时可以根据需要分配和释放内存。堆的大小通常比栈大得多，因此可以用于存储较大的数据结构和对象。
- 栈：栈上的内存生命周期与函数调用相关。局部变量在函数被调用时自动分配内存，函数返回时自动释放内存。栈的大小相对较小，适用于存储较小的数据结构和对象。

如何让对象只能产生在堆/栈上?

对象产生在堆上：将对象的***析构函数设置为私有***的，因为在栈上分配对象的时候，编译器会自动调用对象的构造函数和析构函数，因此此时如果在栈上分配内存会编译报错，就将内存限制在了只能分配在堆上。

对象产生在栈上：把构造函数禁用，使其无法new对象在堆上。

C++指针悬挂问题是什么，如何解决?

指针悬挂：指针指向的内存已被释放或失效，但指针仍保留原地址，访问会引发未定义行为。

常见场景及解决方法：

- 指针释放后未置空：释放内存后立即置空指针（`p = nullptr`）
- 返回局部变量地址：避免返回局部对象指针/引用，改用动态分配或静态变量
- 容器内存重分配：容器扩容导致原有元素指针失效，避免长期持有容器元素指针
- 多指针共享同一内存：一个指针释放内存后，其他指针也需置空或使用智能指针

如果一个class的this指针被删除后强行访问会有什么影响?

未定义行为（UB），可能崩溃或读到乱码值。

核心原因:

- delete this后内存被释放回堆，但指针仍指向原地址（悬空指针）
- 访问已释放内存的结果不确定:
  - 内存未被覆写：可能读到原值（假象正常）
  - 内存被堆管理器覆写：读到随机值（乱码）
  - 内存被其他分配占用：可能触发访问违规（崩溃）
  - 调试模式下：可能被填充特殊值（0xDEADBEEF等）

关键字与语法

C++ 指针和引用的区别?

引用的本质：引用是对象的别名,从语义上看“不是一个独立对象”。
在大多数实现中,编译器会用一个隐藏指针来实现引用,因此它通常占用与指针相同的空间,但这属于实现细节,标准并不强制,一般不在代码中依赖 `sizeof(引用)` 的具体值。

指针与引用的区别:

- 指针本身是一个对象,有自己的地址,可以赋值、拷贝,也可以指向不同的对象,还能为 `nullptr`。
- 引用在定义时**必须初始化**,并且一旦绑定某个对象后就不能再改为引用其他对象,通常也不允许“空引用”。
- 使用指针需要显式解引用 `*p` 访问目标对象;引用则可以像普通变量一样直接使用。
- 语义上,指针适合表示“可选/可变的指向关系”(可以为空、可以重指向),引用适合表示“必须存在的别名”,常用于函数参数和返回值以避免拷贝。

override和final关键字的作用?

介绍一下static和const?

Const：指定语义约束，告诉编译器对象不能被改变，编译器强制检查。

- 可修饰：普通对象（局部/全局）、函数返回值/参数、指针本身/指针指向对象、类成员变量/函数

Static：声明静态成员变量、静态成员函数、静态局部变量、静态全局变量。

- **静态成员**：属于类而非对象，所有对象共享，无需对象即可调用
- **静态局部变量**：函数内声明，程序运行期间只初始化一次
- **静态全局变量**：仅在定义文件内可见，避免命名冲突

volatile关键字的作用和适用场景?

作用：防止编译器优化，强制每次从内存直接读写变量。不保证线程安全，原子操作推荐用atomic。

适用场景：

- 硬件寄存器访问：硬件寄存器的值可能被外部设备随时修改（如传感器、GPIO 状态）
- 中断服务程序（ISR）与主程序共享变量：中断可能异步修改变量（如标志位），主程序需感知最新值
- 多线程环境中的简单标志位：用于线程间通知（如退出标志），但不保证线程安全
- 防止空循环被优化

C++ inline内联的作用?

作用：将代码复制到调用处，消除函数调用开销，提高性能但会导致代码膨胀。

C++17新特性：允许多次定义，不同文件中同名inline函数可实现不同功能（类似static）。

explicit关键字是什么作用?

explicit 关键字在 C++ 中用于禁止**编译器进行隐式类型转换**，强制要求开发者显式调用构造函数或转换运算符，提高代码的安全性和可读性。

- 禁止隐式类型转换
- 禁止拷贝初始化
- 防止隐式转换链

什么是左值，什么是右值？如何使用

- 左值一般是指向一个指定内存的，具有名称的值，它通常拥有一个稳定的内存地址，并且有一段较长时间的声明周期。左值能取到地址，可多次使用。
- 右值通常是不指向稳定内存地址的匿名值，声明周期很短，通常是暂时的。基于此特性，可以用取地址符来判断，右值不能取到地址，通常为一次性使用的值。

前置++返回的是左值还是右值，后置++呢？字符串字面量呢?

- **前置**++：返回左值（直接对对象自增并返回该对象）
- **后置**++：返回右值（创建临时对象保存原值，对原对象自增后返回临时对象）
- **字符串字面量**：返回左值（存储在数据段，有固定内存地址可取址）

class和struct的区别?

- class的默认成员和继承都是**private**的，如果要存储一些内部使用的成员变量推荐使用class,因为内部的一些数据不希望被外部随意获取。
- struct默认是**public**的，如果是要给外部提供一些所需的数据可以使用struct。

引用在声明时是否可以不初始化?

不可以。引用必须在声明时立即绑定到一个有效对象，否则编译失败。

如果C++的引用在声明时不立刻绑定一个对象会有以下几种情况发生：

- 编译失败：主流编译器如GCC等规定引用在声明时**必须绑定**到一个已存在的有效对象，如果没有不会生成可执行文件。
- 运行时出现未定义行为：若通过某些方式绕过编译检查，会出现访问一块未知内存地址的情况，会出现崩溃、数据损坏等后果。
- 出现悬空引用：引用绑定到临时对象或已被销毁的对象，然后对象被销毁后会出现悬空的情况

C++深拷贝和浅拷贝的区别?

深拷贝和 浅拷贝是对象复制的两种核心机制，主要区别在于对**动态资源**（如堆内存）的处理方式。

- 浅拷贝：仅复制对象的成员变量值，多个对象共享同一块动态内存，易导致悬垂指针、双重释放的问题，性能开销小。
- 深拷贝：复制成员变量值，并为指针指向的资源分配新内存，复制内容。但是需手动实现拷贝构造函数和赋值运算符重载。优点是**资源独立，无共享风险。**

modern C++特性

C++11

智能指针有没有了解? 三种智能指针讲一下

智能指针是基于 RAII 封装裸指针的类对象,在构造时获取资源,在析构时自动释放资源,避免忘记 delete、异常路径泄露等问题。

- `shared_ptr` : 共享所有权,内部维护引用计数。每次拷贝计数 +1,销毁或 reset 计数 -1,为 0 时自动释放对象。计数本身是线程安全的,但多线程同时读写同一对象仍需加锁。
- `unique_ptr` : 独占所有权,禁止拷贝,只支持移动。同一时间只能有一个 `unique_ptr` 指向该对象,适合唯一拥有者场景,开销最小。
- `weak_ptr` : 不参与所有权,从 `shared_ptr` 构造,只做“旁观者”。不会增加引用计数,常用于解决 `shared_ptr` 之间的循环引用,需要访问时通过 `lock()` 临时升级为 `shared_ptr`。

Tips: `auto_ptr` 已在 C++11 中废弃, C++17 中移除。

shared_ptr引用计数的原理是什么？什么时候增加引用计数，什么时候减少引用计数?

核心原理：内部维护计数器跟踪指向资源的shared_ptr数量，计数为0时自动释放资源。

引用计数增加：创建新shared_ptr、拷贝构造、拷贝赋值时。

引用计数减少：对象析构（离开作用域）、reset()、重新赋值时。

右值引用是如何提高性能的?

右值引用通过**移动语义**和**完美转发**避免不必要的深拷贝，提升性能。

核心机制

1. 移动语义（Move Semantics）

- 允许"窃取"临时对象的资源（如堆内存、文件句柄），而非拷贝
- 通过移动构造函数和移动赋值运算符实现，将源对象资源的所有权转移到目标对象
- 源对象置为安全的"空状态"（如指针设为nullptr）

2. 完美转发（Perfect Forwarding）

- 使用 `std::forward<T>` 保持参数的值类别（左值/右值）不变
- 模板函数中配合万能引用 `T&&` 使用，避免额外的拷贝或移动

右值引用和左值引用的区别?

左值引用*：避免拷贝、修改原对象、延长临时对象生命周期（const T&）。

**右值引用*：窃取临时对象资源避免深拷贝、完美转发保持值类别。

移动语义如何使用?

1. 自动触发

- 使用临时对象（右值）时编译器自动调用移动构造/移动赋值

```
std::string s = "Hello";  // 普通构造
std::string s2 = std::string("World");  // 移动构造（临时对象）
```

2. 显式调用 std::move

- 将左值强制转换为右值引用，触发移动语义
- **注意**：移动后源对象处于有效但未指定状态，不应再使用

```
std::string s1 = "Hello";
std::string s2 = std::move(s1);  // s1资源转移给s2，s1变空
// s1不应再访问
```

3. 实现移动构造/赋值函数

```
class MyString {
    char* data;
public:
    // 移动构造函数
    MyString(MyString&& other) noexcept
        : data(other.data) {
        other.data = nullptr;  // 源对象置空
    }

    // 移动赋值运算符
```

```
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            delete[] data;  // 释放自身资源
            data = other.data;  // 窃取资源
            other.data = nullptr;
        }
        return *this;
    }
};
```

关键要点：

- **noexcept 声明**：保证异常安全，STL容器优先使用noexcept的移动操作
- **资源所有权转移**：源对象必须置为安全可析构的状态
- **常见场景**：容器操作、函数返回、智能指针转移

完美转发?

完美转发是 C++11 引入的核心特性，用于在函数模板中无损传递参数的原始值类别（左值/右值）和常量性，避免不必要的拷贝并正确触发移动语义。

- 通用引用：统一接收任意类型的参数
- 引用折叠规则：编译器根据传入参数的类型自动应用折叠规则
- forward语义还原：恢复参数的原始值类别。

C++类型推导的作用和用法?

auto:变量类型推导。会丢失引用和cv语义，用auto&保留。万能引用auto&&根据初始值推导左/右值引用。

decltype：推导表达式类型，保留所有信息（引用、cv限定符）。

```
int a = 10;
decltype(a) b = 20;     // b 为 int
decltype(a + 3.14) c;   // c 为 double[5]
```

C++类型推导为什么会有额外的开销?

C++的类型推导有额外开销的原因：

- 1.推导规则复杂：auto会忽略初始化表达式的顶层const、引用和数组退化。需编译器多步分析。decltype的值类别敏感，需根据表达式是变量、函数调用或带括号的左值，分别应用不同规则推导。
- 2.模板实例化负担：在模板中使用auto或decltype推导返回值时，可能触发多次模板实例化。
- 3.意外的值拷贝:若初始化表达式返回引用，但auto未显式声明引用，会进行值拷贝。

使用lambda表达式，捕获局部变量时的规则?

Lambada表达式的捕获规则主要有：值捕获，引用捕获，隐式捕获和显示捕获。

- 值捕获：使用值捕获时，lambda 表达式会复制外部作用域的局部变量，并在lambda 表达式内部使用它们的副本。这意味着捕获的变量在 lambda 表达式创建时就被复制，lambda 表达式内部的操作不会影响原始变量的值。
- 引用捕获：使用引用捕获时，lambda 表达式会获取外部作用域的局部变量的引用。这意味着lambda表达式内部对变量的操

作会影响到原始变量。

- 隐式捕获：通过在捕获列表中使用 = 或 & 符号，可以实现隐式捕获。使用= 捕获外部作用域的所有变量的副本，而使用 & 捕获所有变量的引用。
- 显式捕获：在捕获列表中，可以指定要捕获的特定变量，并且可以同时使用值捕获和引用捕获。

function,lambda,bind之间的关系?

关系总结：lambda和bind生成可调用对象，function包装它们提供统一接口。

各自**特点**：

- **lambda**：匿名函数对象，可捕获外部变量，直接内联开销小
- **bind**：绑定函数参数，生成新可调用对象，支持参数重排
- **function**：通用包装器，类型擦除，统一存储各种可调用对象

配合使用示例：

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

void print(int a, int b) {
    cout << a + b << endl;
}

int main() {
    // lambda -> function
    function<int(int, int)> f1 = [](int a, int b) { return a + b; };
    cout << f1(1, 2) << endl;  // 3

    // bind -> function
    function<void(int)> f2 = bind(print, placeholders::_1, 10);
    f2(5);  // 15

    // 直接使用lambda（无类型擦除，性能更好）
    auto f3 = [](int a, int b) { return a * b; };
    cout << f3(3, 4) << endl;  // 12

    return 0;
}
```

选择建议：

- 优先用lambda（简洁、高效）
- 需要统一类型存储时用function
- bind已被lambda替代，不推荐新代码使用

C++函数封装器为什么优于函数指针?

函数封装器的**优点**:

- 函数封装器兼容函数指针，lambda表达式和仿函数，代码更加简洁、清晰且利于拓展。
- 函数封装器类型安全，有严格的类型检查。
- 与现代C++特性的深度集成
- 面向对象支持

C++14

C++14的新特性?

C++14主要是对一些C++11的已有特性做了扩展。

- 支持更灵活的类型推导，C++14支持decltype(auto),这个auto仅仅作为占位符使用。
- constexpr支持更加广泛的语法和应用，如可以使用局部变量。
- 支持更加通用的lambda表达式，允许表达式内部使用auto参数，处理泛型类型更方便。
- 支持返回类型推导

```
auto add(int x,int y)//推导出来是int类型的返回值
{
    return x+y;
}
int main()
{
    int num1=1;
    int num2=10;
    int num3=add(num1,num2);
    cout<<num3;
    return 0;
}
```

C++17

C++17的新特性?

- **结构化绑定**：元组/结构体成员绑定到变量

```
auto [x, y] = std::make_pair(1, 2);  // x=1, y=2
```

- **if初始化**：if/switch语句中直接初始化变量

```
if (int a = getValue(); a > 0) {
    // 使用a
}
```

- **折叠表达式**：简化可变参数模板

```
template<typename... Args>
auto sum(Args... args) {
    return (args + ...);  // 展开为 arg1 + arg2 + arg3...
}
```

- **constexpr lambda**：编译时求值的lambda

```
constexpr auto add = [](int x, int y) { return x + y; };
constexpr int result = add(3, 4);  // 编译时计算
```

- **std::optional**：安全表示可能无值的对象

```
std::optional<int> divide(int a, int b) {
    if (b == 0) return std::nullopt;
    return a / b;
}

if (auto result = divide(10, 2)) {
    std::cout << *result;  // 5
}
```

- **std::variant**：类型安全的联合体
- **std::filesystem**：现代化文件系统API

模板与泛化

C++中的特化和偏特化是什么?

C++中的模板特化和偏特化是模板编程中用于针对特定类型或条件提供定制化实现的高级技术，旨在**优化性能、处理特殊逻辑或增强类型安全性。**

- 模板特化(全特化)：为模板的所有参数指定具体类型，完全覆盖通用模板的实现。
- 模板偏特化：仅对模板的部分参数进行特化，其余参数保持泛型。

```
/*模板全特化和偏特化的例子*/
#include <iostream>
using namespace std;

// 主模板
template <typename T>
class Vector {
private:
    T* data;
```

```
    size_t size;

public:
    Vector(size_t s) : size(s) {
        data = new T[size];
    }

    ~Vector() {
        delete[] data;
    }

    void info() {
        cout << "通用Vector" << endl;
    }
};

// 全特化：bool 类型
template <>
class Vector<bool> {
private:
    unsigned char* compressedData; // 使用位压缩存储 bool 数组

public:
    Vector(size_t size) {
        compressedData = new unsigned char[(size + 7) / 8];
    }

    ~Vector() {
        delete[] compressedData;
    }

    void info() {
        cout << "特化BoolVector" << endl;
    }
};

// 偏特化：所有指针类型的 Vector<T*>
template <typename T>
class Vector<T*> {
private:
    T** data;
    size_t size;

public:
    Vector(size_t s) : size(s) {
        data = new T*[size];
    }

    ~Vector() {
        delete[] data;
    }

    void info() {
        cout << "偏特化PointerVector" << endl;
    }
};
```

```
int main() {
    Vector<int> v1(10);       // 调用主模板
    v1.info();                // 输出: 通用Vector

    Vector<bool> v2(10);      // 调用全特化
    v2.info();                // 输出: 特化BoolVector

    Vector<int*> v3(10);      // 调用偏特化
    v3.info();                // 输出: 偏特化PointerVector

    return 0;
}
```

SFINAE是什么?

SFINAE（Substitution Failure Is Not An Error）：模板参数替换失败时，编译器不报错，而是忽略该模板候选，尝试其他重载。

应用场景:

1. 条件启用模板

```
template <typename T>
typename std::enable_if<std::is_integral<T>::value, void>::type
process(T val) { /* 处理整型 */ }

template <typename T>
typename std::enable_if<std::is_floating_point<T>::value, void>::type
process(T val) { /* 处理浮点型 */ }
```

2. 检测类型成员

```
template <typename, typename = void>
struct has_serialize : std::false_type {};

template <typename T>
struct has_serialize<T, std::void_t<decltype(std::declval<T>().serialize())>>
    : std::true_type {};
```

3. 重载决策

```
void process(double val);  // 普通函数优先
template <typename T>
void process(T val);       // 模板备选
```

STL容器

C++的栈容器的内部是什么样的，内存是否连续?

C++的栈stack不是独立容器，是基于其他序列容器的适配器。默认使用deque实现。

底层容器内存特性:

- deque：分段连续，由多个固定大小内存块组成，通过中控器管理，逻辑连续但物理不连续
- vector：完全连续，单一大块内存
- list：非连续，双向链表节点分散存储

vector与普通数组的区别？vector扩容如何影响复杂度?

主要区别:

特性	vector	数组
大小	动态，可自动扩容	静态，编译时确定
内存管理	自动管理堆内存	栈上分配或手动管理堆
边界检查	`at()`有边界检查	无边界检查
功能接口	丰富（push_back/size/clear等）	仅基础操作
传递方式	值语义，可拷贝/移动	退化为指针

vector扩容机制与复杂度影响:

1. 扩容过程:

- 容量不足时，分配更大内存（通常是1.5倍或2倍）
- 将原有元素**移动或拷贝**到新内存
- 释放旧内存

2. 复杂度分析:

```
vector<int> vec;
for (int i = 0; i < n; ++i) {
    vec.push_back(i);  // 平摊O(1)，最坏O(n)
}
```

- **单次push_back**:
  - 无需扩容：O(1)
  - 需要扩容：O(n)（拷贝n个元素）
- **摘销分析**：虽然单次扩容是O(n)，但因为扩容频率低（每次增加2倍），平摊O(1)

3. 性能优化建议:

```
// 预先分配空间，避免多次扩容
vec.reserve(1000);  // 预留容量

// 或直接指定初始大小
vector<int> vec(1000);  // 初始化1000个元素
```

4. 频繁扩容的影响：

- 内存分配/释放开销
- 元素拷贝/移动开销
- 迭代器失效（扩容后所有迭代器、指针、引用均失效）

vector的reserve和resize的区别是什么?

vector 的 resize 和 reserve 是两个用于管理容器大小和内存分配的成员函数。它们的区别如下：

- reserve:预留至少 n 个元素的内存空间，但**不创建任何元素**。如果不存在扩容的情况，内存不变化。目的就是为了提前开辟内存空间**提高性能**，避免内存碎片。
- resize：直接改变元素数量，**可能增删元素**。不存在重分配内存的情况。更改容器的大小并调整元素数量，会分配/释放内存和复制/删除元素。

deque和vector的区别？内存布局有啥区别?

deque的分段存储:

- 由多个固定大小内存块（通常512字节）组成
- 中控器（指针数组）管理这些块
- 每个块内部连续，块之间不连续

vector的连续存储：

- 所有元素在一块连续内存中
- 支持直接传递首地址给C API

适用场景：

deque：

- 需要高频头尾操作（队列/双端队列）
- 不需要内存严格连续
- 避免频繁扩容的拷贝开销

vector：

- 只在尾部操作
- 需要与C API交互（传递首地址）
- 需要最优随机访问性能
- 内存局部性要求高

deque,list,set,multiset,map的对比?

- deque:双向队列实现，存储空间连续，支持随机访问，性能比vector低。适合头尾部频繁操作且需要随机访问的场景。
- list:由双向链表实现，存储空间不连续，只能通过迭代器访问，插入删除效率高，但是每个位置都需要分配额外空间存储前驱元素和后继元素。适用于在**任意位置频繁插入/删除**的场景。
- set:红黑树实现，存储空间不连续，只能通过迭代器访问，适用于有序集合且**元素不重复**的场景。
- multiset:红黑树实现，存储空间不连续，只能通过迭代器访问，默认使用less仿函数进行排序，也可以自定义，适用于有序集合且**元素重复**的场景。
- map:由**红黑树**实现,红黑树是一种平衡二叉搜索树,存储空间**不连续**; 存储的元素是键值对元素。只能通过迭代器进行访问。默认使用less仿函数进行排序，map映射容器不允许重复的键。

emplace_back和push_back的区别?

底层机制对比:

- push_back：先构造临时对象 → 拷贝/移动到容器 → 销毁临时对象
- emplace_back：直接在容器内存中原地构造，无中间临时对象

适用场景:

使用emplace_back：

- 构造复杂对象（多个参数）
- 性能关键场景
- 确保参数类型安全

使用push_back：

- 已有对象实例
- 需要初始化列表：`vec.push_back({1, 2, 3})`
- 代码可读性更重要

C++空的class，会主动提供哪些函数?

C++创建了一个空的class，会在特定的情况下提供一些函数。

- 缺省构造函数：声明无参对象时会提供。
- 缺省拷贝构造函数：对象拷贝初始化时候会触发
- 缺省析构函数：对象声明周期结束时会触发
- 缺省赋值运算符：对象复制操作时会触发
- 移动构造函数：对象通过右值初始化触发(C++11)
- 移动赋值运算符：对象通过右值赋值触发(C++11)

多线程与并发

对锁有没有了解,介绍一下?

常见 C++ 锁类型:

- `std::mutex` : 最基本的互斥锁,不可重入。
- `std::recursive_mutex` : 可重入互斥锁,同一线程可多次加锁。
- `std::timed_mutex` : 带超时的互斥锁,支持 `try_lock_for/try_lock_until`。
- `std::recursive_timed_mutex`: 可重入 + 支持超时。
- `std::shared_mutex` : 读写锁,支持多读单写,适合读多写少场景。
- `std::shared_timed_mutex` : 在 `shared_mutex` 基础上增加超时功能。

进程同步的技术有哪些?

1. **互斥锁**：同一时刻只有一个进程访问临界区

```
std::mutex mtx;
std::lock_guard<std::mutex> lock(mtx);  // RAII自动加锁/解锁
```

2. **信号量**：计数器控制多进程访问权限

```
std::counting_semaphore<10> sem(3);  // 允许3个并发
sem.acquire();  // 获取
sem.release();  // 释放
```

3. **条件变量**：等待特定条件成立

```
std::condition_variable cv;
cv.wait(lock, []{ return data_ready; });  // 等待
cv.notify_one();  // 唤醒
```

4. **屏障**：多进程同步点等待

```
std::barrier sync_point(5);  // 等5个进程
sync_point.arrive_and_wait();  // 同步
```

5. **原子操作**：不可分割的变量操作

```
std::atomic<int> counter(0);
counter.fetch_add(1, std::memory_order_relaxed);
```

6. **读写锁**：多读单写

```
std::shared_mutex rw_mutex;
std::shared_lock lock(rw_mutex);  // 读锁（并发）
std::unique_lock lock(rw_mutex);  // 写锁（独占）
```

C++中的原子变量?

原子变量（std::atomic）是C++11引入的无锁线程安全工具，保证操作不可分割，避免数据竞争。

核心特性：

- **原子性**：操作要么未开始，要么已完成，无中间状态
- **内存序**：控制编译器/CPU乱序执行，保证可见性

基本用法：

```
#include <atomic>
#include <thread>

std::atomic<int> counter(0);

void increment() {
    for (int i = 0; i < 1000; ++i) {
        counter.fetch_add(1);  // 原子自增
    }
}

int main() {
    std::thread t1(increment);
    std::thread t2(increment);
    t1.join();
    t2.join();
    std::cout << counter.load() << std::endl;  // 2000
}
```

内存序参数：

内存序	使用场景
memory_order_relaxed	无同步要求，仅保证原子性
memory_order_acquire	读操作，后续读写不可重排到此之前
memory_order_release	写操作，之前读写不可重排到此之后
memory_order_seq_cst	顺序一致（默认），最强同步

底层实现：

- **x86**：lock 前缀指令（lock add），锁定总线/缓存行
- **ARM**：LL/SC指令对（LDREX/STREX），加载链接/存储条件

适用场景：

- 计数器、标志位
- 无锁队列/栈
- 双检锁定单例

注意：复杂场景（多变量同步）仍需互斥锁。

协程是什么?

C++协程（Coroutine）是C++20引入的一种轻量级并发编程机制，它允许函数在执行过程中暂停（挂起）并在稍后恢复，而无需依赖操作系统线程调度，从而简化异步编程、提高资源利用率。

协程是一种特殊函数，可在执行中主动挂起，保存当前状态（局部变量、执行位置等），后续通过协程句柄恢复执行。
C++20采用**无栈协程模型**，挂起时将上下文（局部变量、寄存器状态）存储在堆上

编译与链接

介绍一下RVO?

RVO（Return Value Optimization）是编译器优化技术，在返回对象时直接在调用者的内存位置构造，消除拷贝/移动操作。

RVO分为两种形式：

1. 纯RVO

返回无名临时对象。

```
MyClass createObject() {
    return MyClass();  // 直接返回临时对象，触发RVO
}
```

2. NRVO

返回具名局部对象。

```
MyClass createObject() {
    MyClass obj;  // 局部对象
    // ... 一些操作
    return obj;  // 返回具名对象，可能NRVO
}
```

C++编译链接的过程?

C++程序从源代码到可执行程序的四个阶段：

1. 预处理（Preprocessing）

- 处理 `#include`：展开头文件内容
- 处理 `#define`：宏替换
- 处理 `#if/#ifdef`：条件编译
- 删除注释
- 输出：`.i` 或 `.ii` 文件（纯文本）

```
g++ -E main.cpp -o main.i  # 只预处理
```

2. 编译（Compilation）

- 词法分析、语法分析、语义分析
- 中间代码优化
- 生成特定平台的汇编代码
- 输出：`.s` 文件（汇编代码）

```
g++ -S main.i -o main.s  # 生成汇编
```

3. 汇编（Assembly）

- 将汇编代码转换为机器码（二进制）
- 生成目标文件，包含：
  - 代码段（.text）
  - 数据段（.data/.bss）
  - 符号表
  - 重定位表
- 输出：`.o` 或 `.obj` 文件

```
g++ -c main.s -o main.o  # 生成目标文件
```

4. 链接（Linking）

- **符号解析**：将函数调用与定义关联
- **地址重定位**：确定每个符号的最终内存地址
- **合并段**：将多个.o文件的同类段合并
- **链接库文件**：静态库（.a/.lib）或动态库（.so/.dll）
- 输出：可执行文件（ELF/PE格式）

```
g++ main.o -o main  # 生成可执行文件
```

完整命令：

```
g++ main.cpp -o main  # 一步完成所有阶段
```

关键点：

- 预处理和编译针对单个文件
- 链接阶段处理多个目标文件
- 头文件不参与编译，只在预处理时展开

C和C++编译出来的文件有什么区别?

C和C++编译生成的可执行文件（如ELF、PE格式）在格式层面是兼容的（均可由操作系统加载执行），但因其语言特性差异，二进制内容存在显著区别。

- 名字修饰（Name Mangling）：C++编译器将函数名、参数类型/数量/顺序编码为唯一符号，C语言仅用函数名标识符号。
- 异常处理与运行时类型信息(RTTI)：编译器在二进制中插入异常处理框架（如try/catch的栈回退逻辑）和RTTI数据结构（用于dynamic_cast和typeid），以支持面向对象特性。C语言没有。
- 函数调用约定与对象模型：C++成员函数调用隐含传递this指针（通常通过寄存器或栈），而C函数无此机制。C++在main()前/后插入全局/静态对象的构造/析构代码，而C程序仅按代码顺序执行。

C++的重载和C语言的区别在哪里，具体是如何实现?

- C++：通过名字修饰（Name Mangling）实现。编译阶段将函数名+参数类型/数量编码为唯一符号（如`foo(int)`→`_Z3fooi`），不同参数列表生成不同符号。重载决议在编译阶段完成，编译器根据调用时的实参类型选择最匹配的版本。
- C语言：编译阶段仅用函数名生成符号，同名函数导致符号重复定义，因此不支持重载。

c++重载和重写的区别?

重载是**编译时多态**，参数列表必须不同，作用域是**同一作用域**。

```
class Calculator {
public:
    int add(int a, int b) { return a + b; }          // 重载：参数类型不同
    double add(double a, double b) { return a + b; } // 重载：参数类型不同
    void print(int x) { cout << "Int: " << x; }      // 重载：参数数量不同
    void print(int x, string s) { cout<< s << ": " << x; }
};
```

重写是**运行时多态**,通过虚表（vtable）在运行时决定调用哪个函数。参数列表、返回类型必须与基类虚函数完全一致，可以使用`override`显式标记。

```
class Animal {
public:
    virtual void sound() { cout << "Animal sound" << endl; } // 基类虚函数
};

class Dog : public Animal {
public:
    void sound() override { cout << "Dog barks" << endl; }   // 重写基类虚函数
};

int main() {
    Animal* animal = new Dog();
    animal->sound(); // 输出 "Dog barks"（运行时多态）
    delete animal;
}
```

动态库和静态库的区别?

- 静态库：通过编译器生成目标文件，再用归档工具打包成.a或.lib文件。编译时**直接嵌入**库代码，符号在**链接阶段**解析完成。适合嵌入式系统或离线环境。程序启动时**自动加载**。牺牲空间换取独立性和启动速度，适合封闭环境或资源隔离需求。
- 动态库：编译时添加-fPIC（位置无关码）和-shared选项，生成.so或.dll文件。运行时通过**动态加载器解析**符号地址。运行时通过API**手动加载**。牺牲部署复杂度换取灵活性和资源共享，适合模块化系统或高频更新场景。

C++如何搜索链接到so动态库中的符号的?

C++链接到动态库的过程：

1. 动态库加载与初始化
    - 操作系统通过 mmap 将库文件映射到进程地址空间
    - 动态链接器（ld-linux.so）解析库的依赖关系，递归加载所有依赖库
    - 执行库的初始化函数（.init段和**attribute**((constructor))标记的函数）
2. 符号查找顺序（按优先级）
    - LD_PRELOAD 指定的库（可用于hook系统函数）
    - 主程序符号表
    - DT_NEEDED 指定的依赖库（按广度优先顺序）
    - 全局符号表（RTLD_GLOBAL加载的库）
3. 符号绑定机制
    - 立即绑定（RTLD_NOW）：加载时解析所有符号，启动慢但运行时无开销
    - 延迟绑定（RTLD_LAZY）：首次调用时解析，通过PLT/GOT实现
4. PLT/GOT工作原理

```
首次调用：call PLT[n] -> GOT[n](指向PLT resolver) -> 解析符号 -> 更新GOT[n]为真实地址
后续调用：call PLT[n] -> GOT[n](直接跳转真实地址)
```

A.cpp 调用 B.cpp 中的方法，发生了什么？（静态链接）

1. 编译阶段
    - A.cpp → A.o，B.cpp → B.o
    - A.o中对B的函数调用生成未解析符号（UND）
    - B.o中函数定义生成导出符号
2. 链接阶段
    - 符号收集：合并A.o和B.o的符号表为全局符号表
    - 符号决议：查找A.o中未解析符号在全局符号表中的定义
    - 地址重定位：将A.o中的调用地址修正为B中函数的实际地址
    - 段合并：合并.text、.data、.bss等段，生成可执行文件

A.dll 调用 B.dll 发生了什么？（动态链接）

1. 编译时
   - 只记录B.dll的导入信息（库名、符号名），不嵌入代码
   - 生成导入表（.idata段）和PLT/GOT表项
2. 加载时（程序启动）
   - 加载器解析可执行文件的依赖列表
   - 通过mmap将B.dll映射到进程地址空间
   - 动态链接器解析符号，填充GOT表
3. 运行时（调用瞬间）
   - 通过PLT跳转到GOT中存储的实际地址
   - 延迟绑定时，首次调用触发符号解析
4. 与静态链接的关键区别
   - 静态：符号在编译时解析，地址固定
   - 动态：符号在运行时解析，地址通过间接跳转

内存布局与对齐

进程的虚拟内存布局?

进程的虚拟地址空间布局（从低地址到高地址）：

```
低地址
┌──────────────────┐
│   .text（代码段）    │  可执行机器指令，只读
├──────────────────┤
│  .rodata（只读数据）  │  字符串常量、const全局变量
├──────────────────┤
│   .data（数据段）    │  已初始化的全局/静态变量
├──────────────────┤
│   .bss           │  未初始化的全局/静态变量（不占磁盘空间）
├──────────────────┤
│      堆 ↑         │  动态分配（new/malloc），向高地址增长
│                  │
│  内存映射区         │  动态库、共享内存、mmap文件
│                  │
│      栈 ↓         │  函数调用栈帧，向低地址增长
├──────────────────┤
│   内核空间          │  用户态不可直接访问
└──────────────────┘
高地址
```

各段说明：

- text：程序编译后的机器指令，只读可执行

- rodata：只读数据，如字符串字面量 `"hello"`
- data：已初始化的全局变量和静态变量
- bss：未初始化的全局/静态变量，运行时初始化为0
- 堆：new/malloc分配，需手动释放或由智能指针管理
- 内存映射区：动态库加载、mmap文件映射、共享内存
- 栈：函数调用时自动分配，返回时自动释放，存储局部变量、参数、返回地址
- 内核空间：操作系统内核代码和数据，用户态通过系统调用访问

C++的内存对齐?

C++中的内存对齐（Memory Alignment）是一种由编译器自动实施的内存优化机制，其核心目的是**提升CPU访问内存的效率**，并确保数据在特定硬件架构上能够安全访问。经过内存对齐之后，CPU 的**内存访问速度提升。**
目的：

- **性能优化**：CPU访问对齐的内存（如4字节数据起始地址为4的倍数）通常只需一次内存操作；若未对齐，则可能触发多次访问或硬件异常，尤其在RISC架构。
- 硬件兼容性：**有利于平台移植。**某些处理器（如SPARC、早期ARM）要求数据严格对齐，否则抛出硬件异常。在ARM64上，normal memory非对齐访问不会有问题。但是device memory非对齐访问会报bus error

内存对齐一般是按从大到小的顺序去对齐的。这样可以减少内存对齐所填充的字节数，优化空间。C++的类的虚指针也会参与内存对齐。

内存对齐的规则：(简单来说就是结构体里最大的和编译器的默认对齐系数选较小值，然后如果要节约空间可以把最大的类型放最前面，节约空间)

- 结构体第一个成员的偏移量（offset）为 0，以后每个成员相对于结构体首地址的offset 都是该成员大小与有效对齐值中较小那个的整数倍，如有需要编译器会在成员之间加上填充字节。
- 结构体的总大小为有效对齐值的整数倍，如有需要编译器会在最末一个成员之后加上填充字节。

全局变量存储在哪里?

全局变量存储在静态存储区，和static静态变量一样。

- .data段：已初始化为非0值的全局变量
- .bss段：未初始化或初始化为0的全局变量（不占磁盘空间，运行时统一置0）
- .rodata段：const全局变量

C++变量名是否占用内存空间?

C++变量名是**编译期概念**，编译后被替换为内存地址或寄存器偏移，运行时不存在。

符号表情况：

- 调试符号表（.debug）：编译时生成，用于gdb调试，可通过strip去除，不影响运行
- 动态符号表（.dynsym）：动态库导出符号，运行时加载，占用内存
- 静态符号表（.symtab）：链接时使用，不加载到内存

栈何时会溢出?

栈空间有限（Linux默认8MB），以下情况会导致栈溢出：

- 递归调用过深：每层递归压入栈帧，无终止条件或层数过多
- 局部变量过大：栈上分配大数组（如`int arr[1000000]`）
- 函数调用嵌套过多：每次调用都会压入返回地址、参数、局部变量

栈的速度为什么比堆要快?

原因：

- 分配方式：栈分配只需移动栈指针（一条指令）；堆分配需调用malloc/new，涉及空闲链表查找、内存碎片处理
- 内存连续性：栈内存连续，CPU缓存友好；堆内存分散，缓存命中率低
- 管理开销：栈自动回收（函数返回时）；堆需手动释放或通过智能指针管理
- 硬件优化：栈操作简单频繁，CPU有专用栈指令（push/pop）；堆操作复杂，难以优化

多继承把子类指针转为父类指针和单继承的区别在哪里?

核心区别在于指针是否需要偏移。

单继承：

- 内存布局：父类在前，子类新增在后
- 转换时地址不变，无需偏移
- 共享一个虚表指针

```
Derived对象内存布局：
┌──────────────┐ <- Derived* 和 Base* 指向同一地址
│  Base部分    │
├──────────────┤
│ Derived新增  │
└──────────────┘
```

多继承：

- 内存布局：按继承顺序排列多个父类
- 转换到第一个父类：地址不变
- 转换到后续父类：编译器自动添加偏移量
- 每个含虚函数的父类有独立虚表指针

```
Derived对象内存布局：
┌──────────────┐ <- Derived* 和 Base1* 指向此处
│  Base1部分   │
├──────────────┤ <- Base2* 指向此处（需偏移）
│  Base2部分   │
├──────────────┤
│ Derived新增  │
└──────────────┘
```

```
class Base1 { int a; };
class Base2 { int b; };
class Derived : public Base1, public Base2 { int c; };

Derived d;
Base1* p1 = &d;  // 地址不变
Base2* p2 = &d;  // 编译器自动偏移：p2 = (char*)&d + sizeof(Base1)
```

有的类把析构函数声明为虚函数，什么场景下会用到?

核心场景：通过基类指针删除派生类对象时，需要虚析构确保派生类析构函数被调用。

```
class Base {
public:
    virtual ~Base() {}  // 虚析构
};
class Derived : public Base {
    int* data;
public:
    Derived() { data = new int[100]; }
    ~Derived() { delete[] data; }  // 释放资源
};

Base* p = new Derived();
delete p;  // 虚析构：Derived::~Derived() → Base::~Base()
           // 非虚析构：只调用Base::~Base()，Derived的data泄漏
```

使用原则：

- 多态基类必须声明虚析构
- 纯虚析构需提供定义：`virtual ~Base() = 0; Base::~Base() {}`
- 不作为基类的类不需要虚析构（避免虚表开销）

C++如何解决头文件重复包含的问题?

两种方法：

```
// 方法 1：#pragma once
#pragma once
class MyClass { };

// 方法 2：#ifndef 宏守卫
#ifndef MY_HEADER_H
#define MY_HEADER_H
class MyClass { };
#endif
```

对比：

- pragma once：编译器记住文件路径，后续直接跳过，性能更优，但非标准（主流编译器均支持）
- ifndef：每次都要打开文件并解析到endif，性能稍差，但是标准方法

其他

C++异常的开销是什么?

C++异常处理的开销:

- 栈展开：异常抛出时逆向遍历调用栈，释放局部对象并查找匹配的catch块
- 异常对象管理：异常对象通常在堆上创建，涉及内存分配和拷贝/移动构造
- 类型匹配：运行时解析异常处理表，线性匹配catch块类型，调用栈越深耗时越长
- 隐性开销：即使未使用try/catch，编译器仍生成异常处理元数据，代码体积增加5%-10%

性能特点:

- 正常路径（无异常）：几乎零开销（zero-cost exception）
- 异常路径：开销较大，不适合高频场景

C++中Static全局变量，全局变量，和extern变量的区别

核心区别在于**链接属性**:

类型	链接属性	作用域	存储位置	说明
全局变量	外部链接	整个程序	静态存储区	其他文件可通过extern访问
static全局变量	内部链接	仅本文件	静态存储区	不同文件可同名，互不干扰
extern声明	-	-	不分配空间	声明外部变量，不是定义

```
// a.cpp
int g_var = 1;           // 全局变量，外部链接
static int s_var = 2;    // static全局变量，仅a.cpp可见

// b.cpp
extern int g_var;        // 声明a.cpp中的g_var，可访问
extern int s_var;        // 链接错误，s_var是内部链接
```

内联函数和宏定义的区别?

特性	内联函数	宏定义
处理时机	编译阶段	预处理阶段
展开方式	编译器决定是否展开	无条件文本替换
类型检查	有	无
参数求值	一次	每次使用都求值（可能有副作用）
调试	可调试	无法调试
作用域	遵循C++作用域规则	无作用域概念

```
// 宏的副作用问题
#define SQUARE(x) ((x) * (x))
int a = 5;
int b = SQUARE(a++);  // 展开为 ((a++) * (a++))，未定义行为

// 内联函数无此问题
inline int square(int x) { return x * x; }
int c = square(a++);  // a只求值一次
```

不相关的进程间能否使用管道实现通信?

可以，通过命名管道（Named Pipe/FIFO）实现。

匿名管道 vs 命名管道：

- 匿名管道（pipe）：仅限父子/兄弟进程，通过fork继承文件描述符
- 命名管道（FIFO）：任意进程，通过文件路径访问

命名管道使用：

```
// 创建
mkfifo("./myfifo", 0666);

// 写进程
int fd = open("./myfifo", O_WRONLY);
write(fd, data, len);

// 读进程
int fd = open("./myfifo", O_RDONLY);
read(fd, buf, len);
```

FIFO文件不存储实际数据，仅作为内核管道缓冲区的**访问入口**。

C++什么时候生成默认拷贝构造函数?

默认拷贝构造函数（执行浅拷贝）在以下情况下会被编译器自动生成：

- 类成员包含有拷贝构造函数的类对象
- 类继承自有拷贝构造函数的基类
- 类包含虚函数
- 类存在虚继承