**IOStream: std::cin >> x >> y >> z == ((std::cin >> x) >> y) >> z**

Std::string: file <string> **(not <string.h> , not <cstring> !!)**

std::string s1(7, 'a');//aaaaaaa std::string s; // "" (empty string) Default-initialization of a std::string will produce **an empty string**, not indeterminate value and has no undefined behaviors!

字符串的连接: At least one operand of + should be std::string 例子如下 +具有左结合性

std::string hello{"hello"}; std::string s = hello + "world" + "C++";

**Yes!** + is **left-associated**. (hello + "world") is of type std::string .

**Use “+=”:** s1 += s2 appends s2 directly to the end of s1 , without copying s1 .

**“=”:** std::string s1{"Hello"}; std::string s2{"world"};

s2 = s1; // s2 is a copy of s1

s1 += 'a'; // s2 is still "Hello"

**std::cin >> s 的基本行为**

**跳过前导空白字符**：和 **std::cin >> int** 一样，**std::cin >> s** 会忽略输入中的前导空格、制表符（**\t**）、换行符（**\n**）等空白字符。

**读取到空白字符结束**：它会从第一个非空白字符开始读取，直到遇到下一个空白字符（空格、换行符等）时停止，**不会读取整行**。

**存储到目标变量**：最后将读取的连续非空白字符序列存入字符串 **s**。

**std::getline 的基本行为**

**读取起点**：从输入缓冲区的**当前字符**开始读取（可能是之前操作残留的字符，比如换行符）。

**停止条件**：遇到第一个换行符 **\n** 时停止。

**对 \n 的处理**：

**消耗 \n**：换行符会被从输入缓冲区中**移除**（即“消耗”）。

**不存储 \n**：换行符**不会**存入目标字符串 **s** 中。

**结果**：**s** 中存储的是 **\n** 之前的所有字符（可能为空字符串）。

先 **cin >> x** 再 **getline**，输入缓冲区可能残留换行符，导致 **getline** 读到空行.此时需要用 **cin.ignore()** 清除残留字符。

**Reference:** **Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object.**

**Therefore, references must be initialized.**

**References must be bound to *existing objects* ("lvalues")**

A reference is an alias. It is only an alternative name of another object, but the reference itself is **not an object**.

Therefore, there are no "references to references".

**Any use of a reference is actually using the object that it is bound to!**

Placing the ampersand near the referred type does not make a difference:

int& x = ival, y = ival, z = ival;

// Only `x` is a reference. `y` and `z` are of type `int`.

**Benefits of passing by reference-to- const**

1. Avoids copy.

2. Accepts temporaries and literals (*rvalues*).

3. The const qualification prevents accidental modifications to it.

**[Best practice]** Pass by reference-to- const if copy is not necessary and the parameter should not be modified.

nullptr is ***the*** null pointer value in C++. Do not use NULL .

**std::vector A "dynamic array".**

std::vector **is not a type itself**. It must be combined with some <T> to form a type.

std::vector v; // Error: missing template argument.

std::vector<std::vector<int>> vvi; // An empty vector of vector of `int`s. // "2-d" vector.

std::vector<int> v3(10); // A vector of ten `int`s, all initialized to 0.

std::vector<int> v4(10, 42); // A vector of ten `int`s, all initialized to 42.

**Append an element to the end of a std::vector**

v.push\_back(x)

**Remove the last element of a std::vector**

v.pop\_back()

**v.back() and v.front()**

Return the references to the last and the first elements, respectively.

It is a **reference**, through which we can modify the corresponding element.

v.front() = 42; ++v.back();

For v.back() , v.front() and v.pop\_back() , **the behavior is undefined** if v is empty. They do not perform any bounds checking.

**Access through subscripts**

v[i] returns the **reference** to the element indexed i .

i , where v.size() .

Subscript out of range is **undefined behavior**. v[i] performs no bounds checking.

In pursuit of efficiency, most operations on standard library containers do not

perform bounds checking.

A kind of "subscript" that has bounds checking: v.at(i) .

If i is out of range, *a std::out\_of\_range exception is thrown*

Default initialization of std::string and std::vector results in an empty string /container, not indeterminate values.

Copy of std::string and std::vector is done automatically, which performs member-wise copy.

Memory management is done automatically.

the return type of logical operators && , || , ! and comparison operators < ,<= , > , >= , == , != is bool , not int .

The type of string literals "hello" is const char [N+1] , not char [N+1] .

Recall that string literals are stored in **read-only memory**. Any attempt to modify them results in undefined behavior.

The type of character literals 'a' is char , not int .

const variables initialized with literals are compile-time constants. They can be used as the length of arrays.

int fun() declares a function accpeting no arguments. It is not accepting unknown arguments.

**const\_cast**

Cast away low-level constness **(DANGEROUS)**:

modifying a const object through a non- const access path (possibly formed by const\_cast ) results in **undefined behavior**!

const int cival = 42; int &ref = const\_cast<int &>(cival); // compiles, but dangerous ++ref; // undefined behavior (may crash)

**reinterpret\_cast**

Often used to perform conversion between different pointer types **(DANGEROUS)**:

int ival = 42; char \*pc = reinterpret\_cast<char \*>(&ival);

We must never forget that the actual object addressed by pc is an int , not a character! Any use of pc that assumes it's an ordinary character pointer **is likely to fail** at run time, e.g.:

std::string str(pc); // undefined behavior

**static\_cast<Type>(expr)**

* 用于**编译时已知的、相对安全的类型转换**。
* 不能移除 **const** 或 **volatile** 修饰符。
* 适用于：
  + 基本数据类型之间的转换（如 **int** → **double**）。
  + 父类指针/引用 → 子类指针/引用（向下转型，**不检查安全性**）。
  + 子类指针/引用 → 父类指针/引用（向上转型，安全）。
  + 非 **const** → **const**（但不能反过来）。

auto str = "hello"; // `const char \*`

Recall that the type of "hello" is **const char [6]** , not std::string . This is for compatibility with C.

When using auto , the array-to-pointer conversion ("decay") is performed automatically

**decltype**

decltype(expr) will deduce the type of the expression expr **without evaluating it**

decltype(fun(x, y)) only deduces the return type of fun without actually calling it. Therefore, **no output is produced**

Overloaded functions should be distinguished in the way they are called.

int fun(int); double fun(int); // Error: functions that differ only in // their return type cannot be overloaded.

void move\_cursor(Coord to);

void move\_cursor(int r, int c); // OK, differ in the number of arguments

The following are declaring **the same function**. They are not overloading.

void fun(int \*); void fun(int [10]);

The following are the same for an array argument:

void fun(int \*a); void fun(int (&a)[10]); int ival = 42; fun(&ival); // OK, calls fun(int \*) int arr[10]; fun(arr); // Error: ambiguous call

1. **精确匹配**（包括退化、顶层 **const** 忽略）  
   → **添加底层 const**  
   → **整型/浮点提升**  
   → **数值转换**  
   → **类类型转换**（优先级最低）。

The null pointer

NULL is a macro defined in standard library header files.

In C, it may be defined as (void \*)0 , 0 , (long)0 or other forms.

In C++, NULL cannot be (void \*)0 since the implicit conversion from void \* to other

pointer types is not allowed.

It is most likely to be an integer literal with value zero

Since C++11, a better null pointer is introduced: nullptr

nullptr has a unique type std::nullptr\_t (defined in <cstddef> ), which is neither void \* nor an integer.

fun(nullptr) will definitely match fun(int \*)

We can declare arr to be a **reference to array**:

void print(const int (&arr)[100]) {

for (auto x : arr) // OK. `arr` is an array.

std::cout << x << ' ';std::cout << '\n'; }

class：

private members: Only accessible to code inside the class and friends

public members: Accessible to all parts of the program.

 C++ 中，**class 和 struct 的唯一两个区别**就是：

1. **默认访问权限不同**

在 **class** 中，**默认的成员访问权限是 private**。

在 **struct** 中，**默认的成员访问权限是 public**。

如果有一组成员在开头没有访问修饰符

如果是类，它们是私有的。

如果是结构体，它们是公共的。

class Student { // private: 未声明

std::string name;

this：

class Student { // ... public: bool graduated(int year) const; }; Student s = someValue(); if (s.graduated(2023)) // ...

How many parameters does graduated have?

**显式参数（explicit parameter）**：**int year**（1 个）

**隐式参数（implicit parameter）**：**const Student\* this**

在 C++ 中，**const** 关键字放在成员函数的参数列表之后、函数体 **{ }** 之前，用于声明一个 **const 成员函数**。它的主要作用是保证该函数不会修改对象的状态（即不会修改任何非 **mutable** 成员变量）。

1. **const 成员函数不能修改非 mutable 成员变量**
   * 如果尝试修改，编译器会报错。
   * 但可以读取成员变量。
2. **const 成员函数不能调用非 const 成员函数**
   * 因为非 **const** 成员函数可能修改对象状态，违反 **const** 保证。
3. **const 对象只能调用 const 成员函数**
   * 如果尝试调用非 **const** 成员函数，编译器会报错。
4. **最佳实践**
   * **如果一个成员函数逻辑上不应该修改对象状态，就应该声明为 const**，这样它才能被 **const** 对象调用。

在 C++ 中，**const 成员函数** 的本质是修改了 **this** 指针的类型，使其指向一个 **const** 对象，从而保证该函数不会修改对象的成员变量（除非成员变量是 **mutable**）

1. **const 成员函数** 的 **this** 指针是 **const X\***，因此：

通过 **this->member** 访问的任何成员变量都被视为 **const** 的（不可修改）。

如果尝试修改，编译器会报错。

1. **非 const 成员函数** 的 **this** 指针是 **X\***，因此：

可以自由修改成员变量

**返回局部变量的引用或指针**：这是危险的行为，因为局部变量在函数结束后会被销毁，返回的引用或指针会指向无效的内存。

int& badReference() {

int x = 10;

return x; // 错误：返回局部变量的引用

}

**返回静态变量或全局变量的引用或指针**：这是安全的，因为这些变量的生命周期不会在函数结束后结束。

int\* getPointer() {

static int x = 10;

return &x; // 返回 x 的指针

}

**返回值类型**：适合小型数据或需要独立副本的情况。

**返回引用类型**：适合大型对象或需要直接修改函数内部对象的情况。

**返回常量引用类型**：适合大型对象且不希望被修改的情况。

**返回指针类型**：适合动态对象或可能为空的对象。

返回引用的主要原因是为了支持**链式操作**（Chaining）和**避免不必要的复制**。

**（1）支持链式操作**

链式操作是指将多个操作连续写在一起，例如：

a = b = c; // 赋值运算符的链式操作

a += b += c; // 复合赋值运算符的链式操作

std::cout << a << b << c; // 流插入运算符的链式操作

为了实现这种链式操作，运算符重载函数需要返回一个引用，以便后续操作可以继续作用于同一个对象。

**（2）避免不必要的复制**

如果不返回引用，而是返回值类型，会导致以下问题：

**（1）链式操作失效**

class MyClass {

public:

MyClass operator=(const MyClass& other) {

// 实现赋值逻辑

return \*this; // 返回一个临时对象

}

};

MyClass a, b, c;

a = b = c; // 链式操作失效，因为 b = c 返回的是临时对象，而不是 b 本身

**（2）性能开销**

构造器

构造函数没有返回类型（连 **void** 也没有）。  
构造函数体内可以包含 **return;** 语句，但不能返回任何值。  
这个构造函数的函数体是空的：**{}**

**初始化列表以 : 开头**，后跟成员变量的初始化语句用 **,** 分隔。

**初始化方式必须是 (args) 或 {args}，不能使用 = args**（C++11 后 **=** 在某些情况下可用，但不推荐）。

**初始化顺序由成员变量的声明顺序决定，而非初始化列表中的顺序**。

**所有成员变量都会在进入构造函数体之前初始化**：

如果某个成员未在初始化列表中初始化，则：

如果是基本类型（**int**, **double** 等），其值是**未定义的（undefined）**。

如果是类类型（如 **std::string**），会调用**默认构造函数**。

必须使用初始化列表的情况：

初始化 const 成员（必须在初始化时赋值）。

初始化引用成员（必须在初始化时绑定）。

初始化没有默认构造函数的类成员。

在 C++ 中，**数据成员的初始化顺序严格按照它们在类中的声明顺序执行**，而不是按照构造函数初始化列表中的顺序。

**关键规则**

**初始化顺序 = 类定义中的声明顺序**（从上到下）。

**初始化列表的顺序不影响实际初始化顺序**，但如果二者不一致，编译器可能会发出警告（**-Wreorder**）。

**如果某个成员初始化依赖另一个成员，必须确保被依赖的成员先声明**，否则会导致 **未定义行为（UB）**。

场景 必须用初始化列表？ 示例

const 成员 ✅ 必须 : x(val)

引用成员 ✅ 必须 : ref(val)

无默认构造的类成员 ✅ 必须 : obj(val)

基本类型（int 等） ❌ 可选（但推荐） : x(val)

有默认构造的类成员 ❌ 可选（但推荐） : str(s)

class Point2d {

double x, y;

public: Point2d() : x(0), y(0) {} // default constructor Point2d(double x\_, double y\_) : x(x\_), y(y\_) {} };

|  |  |
| --- | --- |
| Point2d p1; | // calls default ctor, (0, 0) |
| Point2d p2(3, 4); // calls Point2d(double, double), (3, 4) |
| Point2d p3(); |

P3是返回point2d的函数声明不是默认构造

Point2d p4{}; // 明确调用默认构造函数

Point2d p5{1,2}; // 调用带参构造函数

auto ptr = new Point2d; // 动态分配对象调用默认构造函数

std::vector 的关键设计在于将内存分配（allocation）和对象构造（construction）分离：

声明时（如 std::vector<Student> students;）：

仅创建一个空的 vector 对象（通常包含 3 个指针：起始、结束、容量末尾）。不会分配内存，也不会构造任何 Student 对象。此时 students.size() == 0，students.capacity() == 0。

reserve(n) 时：

仅分配足够存储 n 个 Student 的内存（堆内存）。仍然不会构造任何对象（内存中的值是未初始化的）。

2. push\_back 和 emplace\_back 的机制

当向 vector 添加元素时，才会在已分配的内存中构造对象：

方法 行为

push\_back(value) 1. 在尾部直接构造对象的拷贝（调用 Student 的拷贝构造函数）。

2. 若空间不足，先扩容（重新分配内存+迁移现有元素）。

emplace\_back(args) 1. 直接在尾部原地构造对象（调用 Student(args...) 的构造函数）。

避免临时对象的拷贝，效率更高

**局部非静态对象的生命周期**

在 C++ 中，局部非静态对象的生命周期从它被初始化时开始，到控制流离开其作用域时结束。

RAII 是 C++ 中管理资源的核心原则：

**资源获取即初始化**：资源的获取与对象的初始化绑定在一起。

**资源释放即销毁**：资源的释放与对象的销毁绑定在一起。

{

std::string s = "Hello, World!"; // 资源（内存）在初始化时分配

} // 资源在 s 销毁时释放

**局部对象的销毁顺序**

当控制流离开作用域时，局部对象会按照**后进先出**（LIFO）的顺序销毁。

{

std::string s1 = "First";

std::string s2 = "Second";

// s2 先销毁，然后 s1 销毁

}

new 会调用对象的构造函数。

delete 会调用对象的析构函数。

malloc 仅分配内存，不会调用构造函数。

free 仅释放内存，不会调用析构函数。

**析构函数不能 delete[]\*this因为会导致无限递归和未定义行为。**

析构函数只负责清理资源，不负责决定对象本身的生存期。

对象的销毁应由创建它的代码（如 **delete** 或作用域结束）管理，而不是由析构函数自身触发。

class Dynarray {

std::size\_t m\_size;

int \*m\_storage;

public:

~Student() {

delete[] m\_storage;//注意就算size为0也要delete因为new[0]也可能分配内存！

delete this; // 错误！

// 下⾯的赋值毫⽆意义。

m\_size = 0;

m\_storage = nullptr;

}

};

Dynarray a(3);

a.at(0) = 2; a.at(1) = 3; a.at(2) = 5;

Dynarray b = a; // 拷贝初始化（默认是浅拷贝）

print(b); // 输出：2 3 5

a.at(0) = 70; // 修改 `a` 会影响 `b`！

print(b); // 输出：70 3 5（但期望 `b` 仍然是 2 3 5）

**问题原因**：

C++ 默认的 **拷贝构造函数** 执行的是 **浅拷贝**（**shallow copy**），即：

**b.m\_storage** 直接复制 **a.m\_storage** 的指针值，导致 **a** 和 **b** 指向 **同一块内存**。

修改 **a** 的数据会影响 **b**，因为它们共享同一个数组。

如果 **a** 或 **b** 被析构，另一个对象的 **m\_storage** 会变成 **悬空指针**（**dangling pointer**），可能导致 **双重释放**（**double-free**）或内存泄漏

**Type b = a;** 是 **初始化**（调用拷贝构造函数）。

**b = a;**（假设 **b** 已存在）是 **赋值**（调用 **operator=**）。

**你绝对不应该在析构函数中手动调用成员变量的析构函数!**

**拷贝赋值运算符的基本要求**

**返回类型：Dynarray&（返回当前对象的引用，以支持链式赋值 a = b = c）。**

**参数：const Dynarray&（常量引用，避免不必要的拷贝，同时承诺不修改 other）。**

**必须正确处理自赋值（a = a否则可能导致资源泄漏或崩溃。**

**如果类 没有自定义 operator=，编译器会生成一个 默认的拷贝赋值运算符，其行为是：浅拷贝（逐成员赋值）：**

**基本类型直接复制值。**

**指针类型仅复制指针值（导致共享内存、双重释放等问题）。**

**Among the copy constructor, the copy-assignment operator and the destructor:**

**If a class needs a user-provided version of one of them, usually, it needs a user provided version of each of them.**

**如果类 需要自定义析构函数（管理资源），通常也需要 自定义拷贝构造和拷贝赋值运算符。**

**否则默认生成的拷贝操作可能是 不安全的浅拷贝。**

**C++11 的废弃行为：**

**仅声明析构函数时，隐式生成拷贝操作的行为已被废弃（编译器可能警告）。**

**应显式定义所有三个函数（或禁用拷贝）。**

**使用 = default 显式保留默认行为。**

**使用 = delete 显式禁用拷贝（如单例类、不可复制的资源）。**

**需要移动语义时，扩展为 Rule of Five（增加移动构造和移动赋值）。**