**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）。

**Dynamic\_cast**的主要用途是：

在继承层次结构中将**基类指针或引用**转换为**派生类指针或引用**。

检查转换是否安全。如果转换不合法，dynamic\_cast 会返回 nullptr（对于指针）或抛出 std::bad\_cast 异常（对于引用）

Derived\* derived\_ptr = dynamic\_cast<Derived\*>(base\_ptr);

Derived& derived\_ref = dynamic\_cast<Derived&>(base\_ref);

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

**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。

1. **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）。**

**Using：在类内使用using...=typename在类外调用时：**

**Here we use ClassName:: instead of object. , because such members belong to the class, not one single object**

**类型别名受访问控制 和成员变量、函数一样，可以是 public/private/protected。**

**class 默认 private 如果不写 public，外部代码无法访问 A::type。**

**struct 默认 public 适合直接暴露类型别名（如简单数据容器）。**

**STL 风格 STL 容器的类型别名（如 value\_type、size\_type）通常是 public。**

**静态数据成员（static data member）属于 类本身，而不是类的某个对象。它类似于全局变量，但作用域限制在类内，并受访问控制（public/private/protected）约束。访问时需通过 类名作用域（ClassName::member），而非对象名。**

**class A {**

**private:**

**static int private\_static; // 仅类内或友元可访问**

**public:**

**static int public\_static; // 任何地方可访问**

**};**

**声明与定义分离**

**类内声明：用 static 关键字。**

**类外定义：需在全局作用域中单独定义（分配内存）。**

**class A {**

**public:**

**static int count; // 声明**

**};**

**int A::count = 0; // 定义（初始化）**

**如果静态成员是 整型或枚举类型的 const，可以直接在类内初始化。**

**A static data member:**

**class A {**

**static int something;**

**// other members ...**

**};**

**There is only one A::something : it does not belong to any object of A . It belongs to the class A .**

**Like type alias members, we use ClassName:: instead of object. to access them这是一个语法糖:** **It can also be accessed by a.something (where a is an object of type A ), but a.something and b.something refer to the same variable.**

**If f is a function that returns an object of type A , f().something always accesses the same variable no matter what f() returns.**

**In the very first externally available C++ compiler (Cfront 1.0, 1985), f in the expression f().something is not even called! This bug has been fixed soon.早期过度优化**

**当你真的需要访问private变量时, 可以在class内的任意位置使用friend声明某个函数是自己的friend 从而让该函数获得private变量的访问权**

**class Student {**

**friend void print(const Student &); // The parameter name is not used in this declaration, so it is omitted.**

**std::string m\_name;。。。。}**

**你可以在类中的任意位置声明friend, 但习惯上我们把friend的声明位置放在每个类的最前面**

**考虑friend class: 一个对本class的private成员拥有全部访问权限的类:**

**class X {**

**friend class Y;**

**// ...};**

**静态成员函数是类的一部分，但 不依赖于类的实例，可以理解为 类作用域内的全局函数**

**不依赖对象实例：调用时无需创建对象，直接通过类名访问**

**无 this 指针：静态函数不能访问类的 非静态成员（变量或函数），因为它们没有 this 指针。只能访问：**

**他静态成员（变量或函数）。**

**全局变量或函数。**

**受访问控制约束：可以是 public、private 或 protected**

**It can also be called by a.fun(x, y) (where a is an object of type A ), but here a will not be bound to a this pointer, and fun has no way of accessing any non- static member of a .**

**A friend is not a member! You can put this friend delcaration anywhere in the class**

**body. The access modifiers have no effect on it.**

**The :: operator**

**class Widget { public: using gadgets\_list = std::vector<int>; static int special\_member; const gadgets\_list &get\_gadgets() const; // ... };**

**const Widget::gadgets\_list &Widget::get\_gadgets() const { return m\_gadgets; }**

**The members Widget::gadgets\_list and Widget::special\_member are accessed through ClassName:: . The name of the member function get\_gadgets is Widget::get\_gadgets**

**Incomplete Type 如果我们只知道一个类或结构体的名称, 其余一概不知, 那么这个类的类型就是incomplete type 我们能做的唯一一件事就是定义一个指向它的指针或者引用:**

**class Student; // We only have this declaration(incomplete).**

**void print(const Student &stu) { // OK. Declaring a reference to it is OK.**

**std::cout << stu.getName(); // Error. We don't know anything about its members.}**

**When the function body is executed, the object is *not yet* "dead".**

**You can still access its members.**

**~Student() { std::cout << m\_name << '\n'; }**

**After the function body is executed, all its data members are destroyed automatically, in reverse order in which they are declared.**

**For members of class type, their destructors are invoked automatically.**

**int &r = 42; // Error: Lvalue reference cannot be bound to rvalue.**

**int &&rr = 42; // Correct: `rr` is an rvalue reference.**

**const int &cr = 42; // Also correct:**

**// Lvalue reference-to-const can be bound to rvalue.**

**const int &&crr = 42; // Correct, but useless:**

**// Rvalue reference-to-const is seldom used.**

**int i = 42;**

**int &&rr2 = i; // Error: Rvalue reference cannot be bound to lvalue.**

**int &r2 = i \* 42; // Error: Lvalue reference cannot be bound to rvalue.**

**const int &cr2 = i \* 42; // Correct**

**int &&rr3 = i \* 42; // Correct**

**void fun(const std::string &); void fun(std::string &&);**

**fun(s1 + s2) matches fun(std::string &&) , because s1 + s2 is an rvalue.**

**fun(s) matches fun(const std::string &) , because s is an lvalue.**

**Note that if fun(std::string &&) does not exist, fun(s1 + s2) also matches**

**fun(const std::string &) .**

**class Dynarray { int \*m\_storage; std::size\_t m\_length;**

**public:**

**Dynarray(Dynarray &&other) noexcept // move constructor**

**: m\_storage(other.m\_storage), m\_length(other.m\_length) { other.m\_storage = nullptr; other.m\_length = 0; } };**

**1. *Steal* the resources of other , instead of making a copy.**

**2. Make sure other is in a valid state, so that it can be safely destroyed.**

**\* Take ownership of other 's resources!**

**Dynarray &operator=(Dynarray &&other) noexcept {**

**if (this != &other) {**

**delete[] m\_storage;**

**m\_storage = other.m\_storage;**

**m\_length = other.m\_length;**

**other.m\_storage = nullptr;**

**other.m\_length = 0; }**

**return \*this; } };**

**0. Avoid memory leaks!**

**1. *Steal* the resources from other .**

**2. Make sure other is in a valid state, so that it can be safely destroyed.**

**\* Self-assignment safe!**

**The Rule of Five: 以下的"拷贝控制成员", 你要么全部定义(定义五个), 要么一个也不定义**

* + **Copy Constructor**
  + **Copy Assignment Operator**
  + **Move Constructor**
  + **Move Assignment Operator**
  + **Destructor**

**Defined in <utility>**

**std::move 不会销毁对象**

**这个函数可以帮助你将左值转换成右值, 从而匹配T &&类型的参数->你可以直接"移动"它! 但这看起来很诡异: 你抢夺了一个"左值"的资源! 而左值的生命周期应该是持久的! 这就是问题所在, std::move相当于一个声明: 我不会再使用这个左值变量了 或者说: 我希望就地终止这个左值变量的生命周期, 并允许其它对象抢夺它的资源**

1. **比较以下两个constructor**

**Dynarray(Dynarray &&other) noexcept**

**: m\_storage(other.m\_storage), m\_length(other.m\_length),**

**m\_label(other.m\_label) { // Isn't this correct?**

**other.m\_storage = nullptr;**

**other.m\_length = 0;**

**}**

**Dynarray(Dynarray &&other) noexcept**

**: m\_storage(other.m\_storage), m\_length(other.m\_length),**

**m\_label(std::move(other.m\_label)) { // !!**

**other.m\_storage = nullptr;**

**other.m\_length = 0;**

**}**

**注意在构造器列表中m\_label的处理: 你应该使用std::move()而不是直接写other.m\_label 这是因为: 在构造器函数中, other是一个左值!(虽然他实际上是右值) Golden rule: Anything that has a name is an lvalue. 所以, 右值引用是一个左值! 因此你如果直接写other.m\_label, 这是一个拷贝操作!**

**Move and Copy Elision 考虑如下代码:**

**std::string foo(const std::string &a, const std::string &b) {**

**return a + b; // a temporary**

**}**

**std::string s = foo(a, b);**

**s的赋值操作时copy还是move?**

**两者都不是! s将会被直接以a+b初始化, 如下:**

**std::string s(a + b);**

**在C++17后这被称作copy elison**

**NRVO 的触发条件**

**当函数返回一个 局部对象（具名变量） 时，编译器可能会应用 NRVO：**

**std::string createString() {**

**std::string s = "Hello"; // 局部具名对象**

**return s; // 可能触发 NRVO**

**}**

**Dynarray concat(const Dynarray &a, const Dynarray &b) {**

**Dynarray result(a.size() + b.size());**

**// ...**

**return result;**

**}**

**Dynarray a = concat(b, c); // Initialization**

**经过编译器的NRVO优化后, 代码会被修改为:**

**// Pseudo C++ code.**

**void concat(Dynarray &result, const Dynarray &a, const Dynarray &b) {**

**// Pseudo C++ code. For demonstration only.**

**// 以下语句在函数头第一行发生!**

**result.Dynarray::Dynarray(a.size() + b.size()); // construct in-place**

**// ...**

**}**

**Dynarray a@; // Uninitialized.**

**concat(a@, b, c);**

**相当于直接给concat函数加了一个参数: 传入未经初始化的最终变量a, 所有的操作都会在a中进行 完全规避了任何可能发生的move和copy, 相当于完全不用中间变量!**

**一个比较drama的用法 值传递的妙用：class Dynarray {**

**public:**

**Dynarray &operator=(Dynarray other) noexcept {**

**swap(other);**

**return \*this;**

**}**

**};**

**在传参时, 如果传入右值, other会被移动初始化, 如果传入左值, 那么会被拷贝初始化 因此, 对于传入右值或者左值的情况都能胜任 这既是一个拷贝赋值运算符, 又是一个移动赋值运算符!**

**值语义（深拷贝） 复制所有资源 安全、独立 成本高 std::string, std::vector**

**指针语义（浅拷贝） 共享资源 + 引用计数 高效、共享数据 需处理竞态条件 std::shared\_ptr**

**禁用拷贝 = delete 拷贝操作 强制独占所有权 灵活性降低 std::unique\_ptr**

**所有智能指针均支持以下指针式操作：**

**std::shared\_ptr<Widget> sp(new Widget);// 解引用**

**Widget& ref = \*sp; // 获取对象引用**

**sp->doSomething(); // 调用成员函数**

**// 布尔上下文检查**

**if (sp) { // 等价于 sp != nullptr**

**std::cout << "sp points to a valid object";**

**}**

**std::unique\_ptr：独占所有权**

**std::unique\_ptr<Widget> up1(new Widget);**

// std::unique\_ptr<Widget> up2 = up1; // 错误！不可复制

**std::unique\_ptr<Widget> up2 = std::move(up1); // 正确：所有权转移**

**std::shared\_ptr：共享所有权**

**引用计数机制：**

**auto sp1 = std::make\_shared<Widget>(); // 引用计数=1**

**auto sp2 = sp1; // 引用计数=2**

**循环引用问题（需配合 weak\_ptr）：**

**struct Node {**

**std::shared\_ptr<Node> next;**

**// std::weak\_ptr<Node> next; // 正确解法**

**};**

**auto node1 = std::make\_shared<Node>();**

**auto node2 = std::make\_shared<Node>();**

**node1->next = node2; // 循环引用导致内存泄漏！**

**class WindowPtr {**

**Window \*ptr;**

**public:**

**// 此处的 = nullptr的意思是默认参数为nullptr, 并不是赋值语句!**

**// 即你不传入参数时, 默认p为nullptr**

**WindowPtr(Window \*p = nullptr) : ptr(p) {}**

**// 利用destructor做自动垃圾回收, 生命周期结束时, 自动释放内存**

**~WindowPtr() { delete ptr; }**

**// 不允许使用另一个已经存在的unique\_ptr来初始化新的unique\_ptr!(否则指向了同一个对象)**

**WindowPtr(const WindowPtr &) = delete;**

**// 不允许使用赋值语句复制另一个unique\_ptr**

**WindowPtr &operator=(const WindowPtr &) = delete;**

**// Move COnstructor: 移动语义: 你可以移动右值类型的unique\_ptr, 表现为资源抢占, 原来的unique\_ptr将失效**

**WindowPtr(WindowPtr &&other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }**

**// Move Assignment Operator:**

**WindowPtr &operator=(WindowPtr &&other) noexcept {**

**if (&other != this) {**

**delete ptr; ptr = other.ptr; other.ptr = nullptr;**

**}**

**return \*this;**

**}**

**};**

**创建std::unique\_ptr的两种方式: constructor与make\_unique完美转发**

**std::unique\_ptr<Student> p(new Student("Bob", 2020123123));//传统方式 但如果student类创建失败就会内存泄漏**

**std::unique\_ptr<Student> p1 = std::make\_unique<Student>("Bob", 2020123123);**

**auto p2 = std::make\_unique<Student>("Alice", 2020321321);**

**如果你在初始化某个unique\_ptr时不传入任何参数, 那么它将会是nullptr空指针 std::unique\_ptr<T> up;**

**unique\_ptr的Move Only语法:**

**auto p = std::make\_unique<std::string>(5, 'c');**

**std::cout << \*p << std::endl; // Prints "ccccc".**

**auto q = p; // Error. Copy is not allowed.**

**auto r = std::move(p); // !!!Correct.!!!**

**// Now the ownership of this string has been transferred to `r`.**

**std::cout << \*r << std::endl; // Prints "ccccc".**

**if (!p) // true**

**std::cout << "p is \"null\" now." << std::endl;**

**注意! 在move完成后, 原来的unique\_ptr会变成nullptr! 这是因为move操作会销毁原来的对象**

**std::unique\_ptr<int> up(new int[10]); // 危险操作！**

**1. 实际发生的行为**

1. **内存分配：**

**new int[10] 动态分配了一个包含 10 个 int 的数组，返回指向数组首元素的指针（int\*）。**

1. **所有权接管：**

**std::unique\_ptr<int> 接管了这个指针的所有权。**

1. **析构时的灾难：**

**当 up 离开作用域时，其默认析构器会调用 delete（而非 delete[]）释放内存。**

**2. 为什么这是未定义行为（UB）？**

**new[] 必须配对 delete[]：  
数组分配会额外存储元素数量（编译器实现相关），delete 无法正确识别该信息，导致：**

**内存泄漏（部分编译器）。**

**堆损坏或程序崩溃（多数现代编译器）。**

**如果我需要一个指向数组的unique\_ptr怎么办? 要知道, unique\_ptr内部是通过delete来释放资源的 而释放数组需要使用delete[] ! 解决方案: 使用std::unique\_ptr<T[]>, 它内部使用delete[]来销毁对象 但使用这种方案生成的unique\_ptr无法支持operatoe->, operator\*的操作! (忘掉C语言的"数组名就是首地址"吧!)**

**允许多个指针指向同一个变量, 或者说, 允许"被复制copy"而不是move only 一个自然的问题是: 我们什么时候释放内存? 现在有很多个不同生命周期的指针指向同一块内存 解决方案: 引用计数!**

1. **例子:**

**class WindowPtr {**

**WindowWithCounter \*ptr;**

**public:**

**// Copy (Assignment) Constructor**

**WindowPtr(const WindowPtr &other) : ptr(other.ptr) { ++ptr->refCount; }**

**WindowPtr &operator=(const WindowPtr &other) {**

**// 先加上对方引用计数: 保证self-assign的安全性! ptrA = ptrA不会导致ptrA被delete**

**++other.ptr->refCount;**

**// ptrA = ptrB, 那么ptrA所指向的对象就少了一个指向它的指针, 因此要引用计数减一**

**// 或者说, 原来指向A的指针现在指向了B, 自然A的引用计数要减一**

**if (--ptr->refCount == 0)**

**delete ptr;**

**ptr = other.ptr;**

**// 运算符重载必须显式地返回\*this!!!**

**// \*this是对this指针的解引用, 类型为WindowPtr**

**// 但是函数返回的是WindownPtr &, 这会导致编译器做一个"类型转换"**

**// 实际上返回的是引用**

**return \*this;**

**}**

**//Move (Assignment) Constructor**

**// 由于是移动, 是"抢夺"式的赋值, 所以根本不需要更新引用计数!**

**// 比如WindowPtr ptrA(std::move(ptrB)), 原先ptrB指向B, 现在ptrA指向B而ptrB为nullptr**

**// 从对象B的视角来看, 没有引用计数的变化!**

**// 而ptrA在这个表达式之前不存在, 从未被初始化! 因此也不需要更新ptrA指向对象的引用计数(ptrA不指向任何一个对象)**

**WindowPtr(WindowPtr &&other) noexcept : ptr(other.ptr) { other.ptr = nullptr; }**

**WindowPtr &operator=(WindowPtr &&other) noexcept {**

**if (this != &other) {**

**// ptrA = ptrB**

**// 你需要更新A对象的引用计数! 因为此时ptrA指向了其他对象**

**if (--ptr->refCount == 0)**

**delete ptr;**

**ptr = other.ptr; other.ptr = nullptr;**

**}**

**return \*this;**

**}**

**~WindowPtr() {**

**if (--ptr->refCount == 0)**

**delete ptr;**

**}**

**};**

**[Best practice] 永远优先使用make\_shared来创建shared\_ptr**

1. **你可以使用use\_count来查看当前指针指向对象的引用计数:**

**auto sp = std::make\_shared<std::string>(10, 'c');**

**{**

**auto sp2 = sp;**

**std::cout << sp.use\_count() << std::endl; // 2**

**} // `sp2` is destroyed, but the managed object is not destroyed.**

**std::cout << sp.use\_count() << std::endl; // 1**

**shared\_ptr不是零成本抽象!因为你需要维护一个引用计数器!**

**完美转发: make\_shared和make\_unique会把传入的参数忠实的完整地完美地传递给对应类型的构造函数! 传入右值就会触发对应类型构造函数的移动语义, 完全保留传入参数的类型与左右值状态, 这被称作完美转发**

**auto sp = std::make\_shared<std::string>(10, 'c'); // "cccccccccc"**

**// 传入右值, 意为说明"你可以窃取sp的资源", 因此触发了std::string的Move Constructor, 将sp指向的资源抢夺走了!**

**// 因此sp指向的对象会被"归零"**

**auto sp3 = std::make\_shared<std::string>(std::move(\*sp));**

**std::cout << \*sp << std::endl; // empty string**

**// 这确实证明了我们成功触发了Move Constructor!**

**运算符重载的一些规则:**

**各个运算对象从左向右依次作为这个函数的参数。**

**如果是成员函数，则最左侧的运算对象绑定到 \*this 。**

**运算符重载有两种方式: 作为某个对象类内的成员函数, 以及非成员函数**

**a[i] //a.operator[](i)**

**a = b //a.operator=(b)**

**\*a //a.operator\*()**

**f(arg1, arg2, arg3, ...) //f.operator()(arg1, arg2, arg3, ...)**

**a == b //operator==(a, b)**

**a + b //operator+(a, b)**

**但有些运算符不建议重载: 比如&&和||, 你无法保证重载后的运算符还能保持"短路"的特性!**

**运算符重载的规则: 至少一个操作数需要时你定义的类; 你不能发明或者新定义一个新的运算符**

**int operator+(int, int); // Error.**

**MyInt operator-(int, int); // Still error.**

**double operator\*\*(double x, double exp); // Error.**

**运算符重载不会改变优先级, 以及evaluation order**

**std::cout << a + b; // Equivalent to `std::cout << (a + b)`.**

**Example: 有理数Rationale**

**class Rational {**

**friend Rational operator-(const Rational &); // Unary `operator-` as in `-x`.**

**int m\_num; // numerator**

**unsigned m\_denom; // denominator IMPORTANT: 注意这里的unsigned!!!**

**public:**

**Rational &operator+=(const Rational &rhs) {**

**m\_num = m\_num \* static\_cast<int>(rhs.m\_denom) // Be careful with `unsigned`! IMPORTANT!!**

**+ static\_cast<int>(m\_denom) \* rhs.m\_num; // 注意我们在进行int与uint的混合计算! 必须进行显式类型转换!**

**m\_denom \*= rhs.m\_denom;**

**simplify();**

**return \*this; // `x += y` should return a reference to `x`.**

**// 这里的(x += y)表达式的返回值应该是x!**

**Rational operator-(const Rational &x) {**

**return {-x.m\_num, x.m\_denom};**

**// The above is equivalent to `return Rational(-x.m\_num, x.m\_denom);`.**

**// IMPORTANT! 相当于直接返回了一个构造器!**

**什么时候用成员函数类型的运算符重载, 什么时候不用成员函数? 成员函数运算符重载总是比对应的非成员函数版本少一个显式参数（因为左操作数或唯一操作数通过 this 隐式传递了）!**

**当你需要对称性, 即左操作数不是你定义的类对象时(比如int + MyClass), 使用成员函数运算符重载就无法满足你的要求 阿贝尔群上(有交换律/对称的)的运算符就需要使用非成员函数来重载!**

**如果用成员函数来重载rational&型错误如下：Rational r = some\_value();**

**auto s = r + 0; // OK, `r.operator+(0)`, effectively `r.operator+(Rational(0))` auto t = 0 + r; // Error! `0.operator+(r)` ???**

**继续完善Rationale类: 巧妙运用+=运算符! 注意是否是成员函数! 注意到二元运算符一般是不适用成员函数来重载**

**class Rational {**

**public:**

**Rational &operator-=(const Rational &rhs) {**

**// Makes use of `operator+=` and the unary `operator-`.**

**return \*this += -rhs;**

**}**

**};**

**Rational operator+(const Rational &lhs, const Rational &rhs) {**

**return Rational(lhs) += rhs; // Makes use of `operator+=`.**

**}**

**Rational operator-(const Rational &lhs, const Rational &rhs) {**

**return Rational(lhs) -= rhs; // Makes use of `operator-=`.**

**}**

**// 假设我们已经实现了operator<**

**friend bool operator<(const Rational &lhs, const Rational &rhs) {**

**return lhs.m\_num \* static\_cast<int>(rhs.m\_denom) < rhs.m\_num \* static\_cast<int>(lhs.m\_denom);}**

**bool operator>(const Rational &lhs, const Rational &rhs) {**

**return rhs < lhs;**

**}**

**bool operator<=(const Rational &lhs, const Rational &rhs) {**

**return !(lhs > rhs);**

**}**

**bool operator>=(const Rational &lhs, const Rational &rhs) {**

**return !(lhs < rhs);**

**}**

**bool operator!=(const Rational &lhs, const Rational &rhs) {**

**return !(lhs == rhs);**

**}**

**[Best Practice] Avoid Repetition: 只实现一两个核心的运算符重载, 其他的重载函数依赖这个"元函数"来实现!**

**注意++x和x++: 自增自减运算符的重载并不一样! ++x --> x.operator++(void) x++ --> x.operator++(0)**

**x++的重载函数会有一个int类型的dummy parameter用以区分++x和x++!**

**class Rational {**

**public:**

**// 想想看: 为什么++x的返回值是x本身的引用, 而x++返回的是一个值?**

**// IMPORTANT:**

**Rational &operator++() { ++m\_num; simplify(); return \*this; }**

**Rational operator++(int) { // This `int` parameter is not used.**

**//The postfix version is almost always defined like this.**

**auto tmp = \*this;**

**++\*this; // Makes use of the prefix version.**

**return tmp;**

**重载IO运算符: 必须使用 非成员函数来重载! 因为其参数中有输入输出流! 而你定义的类对象的成员函数无法接受输入输出流作为参数! std::istream 和 std::ostream 都不能拷⻉，必须按引⽤传递，⽽且不能是常量引⽤。**

**std::ostream &****operator<<(std::ostream &, const Rational &);**

**std::istream &operator>>(std::istream &, Rational &);**

**// 错误! 不能使用常量引用!**

**std::istream &operator>>(const std::istream &, Rational &); // Error!**

**有关Rationale的输出输出流重载:**

**std::istream &operator>>(std::istream &is, Rational &r) {**

**int x, y; is >> x >> y; // 这句话的意思是从is中读取两个int类型的变量**

**这是重载输入输出流的核心!**

**if (!is) { // Pay attention to input failures!**

**// 注意! input failure对应的istream为0!**

**x = 0;**

**y = 1;**

**}**

**if (y < 0) { y = -y; x = -x; }**

**r = Rational(x, y);**

**return is;**

**使用[]下标运算符重载: 用于Dynarray**

**class Dynarray {**

**public:**

**int &operator[](std::size\_t n) {**

**return m\_storage[n];**

**}**

**const int &operator[](std::size\_t n) const {**

**return m\_storage[n];**

**为什么需要两种类型的(const和非const)的[]下标运算符重载?**

**因为const类型的Dynarray对象只能调用const类型的成员函数! 所以你必须写这个重载!**

**重载解引用\*运算符: 用于WindowPtr**

**class WindowPtr {**

**WindowWithCounter \*m\_ptr;**

**public:**

**Window &operator\*() const {**

**return m\_ptr->theWindow;**

**为什么operator\*函数需要是const的? 因为你需要保证const的Window对象可以调用! 记住! const的对象\_\_不能调用\_\_非const的成员函数! 然而以上的代码仍然有一个问题: 你可以尝试修改返回的对象! 而这不是const函数所承诺的内容: 保证不会修改原对象**

**重载operator-> 用于WindowPtr**

**class WindowPtr {**

**public:**

**Window \*operator->() const {**

**return std::addressof(operator\*());**

**其实std::addressof(x)的作用和&x几乎无异, 但关键点是, 程序员很可能自己重载了&运算符!因此你最好不要用&!**

**重载类型转换运算符:**

**class YourClass {**

**public:**

**// 语法: operator TargetType() [const] [explicit] [noexcept] { ... }**

**operator TargetType() const {**

**// ... 实现转换逻辑 ...**

**// 必须返回一个 TargetType 类型的值，**

**// 或者可以隐式转换为 TargetType 的值。**

**TargetType result = /\* ... \*/;**

**return result;**

**}**

**// ... 其他成员 ...**

**};**

**注意以下几点:**

**类型转换运算符只能定义为成员函数**

**语法一定是operator起手, operator是一个关键字! 而不是返回值类型**

**必须加上const修饰!**

**如果你不期望发生如下的隐式类型转换:**

**class Rational {**

**public:**

**// conversion from `Rational` to `double`.**

**operator double() const { return 1.0 \* m\_num / m\_denom; }**

**};**

**Rational r(3, 4);**

**double dval = r; // 0.75 IMPORTANT! 此处发生了隐式类型转换!**

**void bar(const std::vector<int> &vec) { /\* ...\*/ }**

**bar(1000); // ??? Too weird!**

**std::vector<int> v2 = 1000; // No! This should never happen. Too weird!**

**你就应该在类型转换重载函数的前面使用explicit关键字: 这样就保证了你只能使用TypeName(ClassType)的方式来实现类型转换, 而不会发生任何形式的隐式类型转换**

**class Rational {**

**public:**

**explicit operator double() const { return 1.0 \* m\_num / m\_denom; }**

**};**

**Rational r(3, 4);**

**double d = r; // Error.**

**void foo(double x) { /\* ... \*/ }**

**foo(r); // Error.**

**foo(double(r)); // OK.**

**foo(static\_cast<double>(r)); // OK.**

**但有一个例外: bool类型的类型转换, 即使你加了explicit, 在以下场景中编译仍然通过:**

**if (expr) , while (expr) , for (...; expr; ...) , do ... while (expr)**

**as the operand of ! , && , ||**

**as the first operand of ?: : expr ?something: something\_else**

**Basic operations, supported by almost all kinds of iterators:**

**\*it : returns a reference to the element that it refers to.**

**it->mem : equivalent to (\*it).mem .**

**++it , it++ : moves it one step forward, so that it refers to the "next" element.**

**++it returns a reference to it , while it++ returns a copy of it before**

**incrementation.**

**it1 == it2 : checks whether it1 and it2 refer to the same position in the**

**container.**

**it1 != it2 : equivalent to !(it1 == it2) .**

**These are supported by the iterators of all sequence containers, as well as std::string .**

**The range-based for loop**

**for (@declaration : container) @loop\_body**

**is equivalent to**

**{ auto b = std::begin(container); auto e = std::end(container); for (; b != e; ++b) { @declaration = \*b;**

**@loop\_body }}**

**Like pointers, an iterator can be dereferenced ( \*it ) only when it refers to an existing element. ("dereferenceable")**

**\*v.end() is undefined behavior.**

**++it is undefined behavior if it is not dereferenceable. In other words, moving**

**an iterator out of the range [begin, off\_the\_end] is undefined behavior.**

**迭代器失效：**

**std::vector**

**操作 失效范围**

**push\_back(x) 所有迭代器（若触发扩容）；否则仅 end()。**

**pop\_back() 指向末尾元素的迭代器（如 end() - 1）。**

**insert(pos, x) pos 及之后的迭代器（若触发扩容则全部失效）。**

**erase(pos) pos 及之后的迭代器。**

**clear() 全部迭代器。**

**(2) std::deque**

**操作 失效范围**

**push\_back(x)/push\_front(x) 通常无影响（除非重新分配内部块）。**

**insert(pos, x) 所有迭代器（中间插入可能触发重新分配）。**

**erase(pos) 所有迭代器（若删除中间元素）。**

**(3) std::list/std::forward\_list**

**插入操作：不会失效任何迭代器（除被删除元素的迭代器）。**

**删除操作：仅失效被删除元素的迭代器。**

**(4) 关联容器（std::set, std::map, etc.）**

**插入/删除：仅失效被操作元素的迭代器（除非节点被重组，如 unordered\_map 的 rehash）。**

**前向迭代器（ForwardIterator）：支持 \*it、it->mem、++it、it++、it1 == it2、it1 != it2。**

**双向迭代器（BidirectionalIterator）：前向迭代器，可以双向移动。**

**支持 --it 和 it--。**

**随机访问迭代器（RandomAccessIterator）：双向迭代器，可以在常数时间内移动到任何元素。**

**支持 it + n、n + it、it - n、it += n、it -= n（n 为整数）。**

**支持 it[n]，等价于 \*(it + n)。**

**支持 it1 - it2，返回两个迭代器之间的距离。**

**支持 <、<=、>、>=。**

**Which category is the built-in pointer in? – RandomAccessIterator**

**\_n结尾的algorithms接受的参数为一个begin iterator和一个正整数n, 表示遍历begin + n 正常不带\_n结尾的algorithm接受的参数是两个iterator! 分别是begin与end**

**使用algorithm重写Dynarray类**

**Dynarray::Dynarray(const int \*begin, const int \*end)**

**: m\_storage{new int[end - begin]}, m\_length(end - begin) {**

**std::copy(begin, end, m\_storage);**

**}**

**Dynarray::Dynarray(const Dynarray &other)**

**: m\_storage{new int[other.size()]}, m\_length{other.size()} {**

**std::copy\_n(other.m\_storage, other.size(), m\_storage);**

**}**

**Dynarray::Dynarray(std::size\_t n, int x = 0)**

**: m\_storage{new int[n]}, m\_length{n} {**

**std::fill\_n(m\_storage, m\_length, x)**

**Algorithms的返回值一般是一个iterator, 用于表示容器内的某个position "Not found" / "No such element"一般是返回c.end()这个iterator**

**std::vector<int> v = someValues();**

**auto pos = std::find(v.begin(), v.end(), 42);**

**assert(\*pos == 42);**

**auto maxPos = std::max\_element(v.begin(), v.end());**

**// pos和maxPos都是Iterataor**

**如果你在if中检查的临时变量在if-body内仍然要用, 这很常见 你可以试试看C++17添加的if新语法: if(init\_expr; condition)**

**if (auto pos = std::find(v.begin(), v.end(), something); pos != v.end())**

**std::cout << \*pos << '\n';**

**init\_expr就像for循环的第一个子句 而init\_expr定义的变量的作用域就是整个if-body**

**Algorithm可能对输入参数和容器有一些要求:**

**对与输入的Iterator类型有要求, 比如std::sort()要求必须输入RandomAccessIterator**

**对于容器内存储的内容有一定要求**

**需要进行容器元素比较的Algorithm一般都需要你实现类对象的operator<和operator==运算符(你需要重载!) 注意! 任何用到比较的Algorithm都只需要用到operator<和operator==, 你不需要重载其它的operator>, 等等**

**Algorithm承诺绝对不会修改容器的长度, 从来不会主动删除, 插入新的元素, 只会修改/替换原有元素**

**std::vector<int> a = someValues();**

**std::vector<int> b(a.size());**

**std::vector<int> c{};**

**std::copy(a.begin(), a.end(), b.begin()); // OK**

**std::copy(a.begin(), a.end(), c.begin()); // Undefined behavior!**

**// c的长度为0, std::copy不会修改c!**

**std::unique(begin, end) unique用于消除容器内重复的元素, 需要你实现operator==和operator< 注意! unique对容器内的元素有要求: 必须已经被排序为升序排列 但std的algorithm承诺不会修改容器长度, 那么怎么做到去除重复元素呢? 答案是unique会将所有的重复元素移动到容器末尾, 并返回重复元素开始的pos iterator**

**std::vector v{1, 1, 2, 2, 2, 3, 5};**

**auto pos = std::unique(v.begin(), v.end());**

**// Now [v.begin(), pos) contains {1, 2, 3, 5}.**

**// [pos, v.end()) has the values {1, 2, 2}, but the exact order is not known.**

**v.erase(pos, v.end()); // Typical use with the container's `erase` operation**

**// Now v becomes {1, 2, 3, 5}.**

**Predicate std的Algorithm支持在最后传入一个callable对象, 返回值为Bool, 用于进行比较或搜寻**

**bool less\_than\_10(int x) {return x < 10;}**

**std::vector<int> v = someValues();**

**auto pos = std::find\_if(v.begin(), v.end(), less\_than\_10);**

**也可以用predicate来实现遍历并打印容器中每个值**

**void print\_int(int x) { std::cout << x << ' '; }**

**std::for\_each(v.begin(), v.end(), print\_int);**

**如果这个Predicate依赖某些运行时定义的参数, 怎么办? 按如下定义!**

**struct LessThan {**

**int k\_; // No constructor is needed, and k\_ is public.**

**//重载operator()使其成为callable对象, 让std::find\_if能够调用**

**bool operator()(int x) const { return x < k\_; }};**

**// 在此处用k初始化LessThan对象**

**auto pos = std::find\_if(v.begin(), v.end(), LessThan{k}); // {} instead of ()**

**fo(arg1, arg2, ...) is equivalent to fo.operator()(arg1, arg2, ...) . Any number of arguments is allowed.**

**再来一个例子:**

**struct AbsCmp {**

**bool operator()(int a, int b) const {**

**return std::abs(a) < std::abs(b);}};**

**std::sort(v.begin(), v.end(), AbsCmp{});**

**Lambda匿名函数 语法:**[**capture\_list**](https://github.com/kiturolin/CS100/blob/main/notes/params)**-> return\_type { function\_body } capture\_list即为function\_body需要用到的, 运行时决定的某些常数 eg.**

**std::sort(v.begin(), v.end(),**

**[](int a, int b) -> bool { return std::abs(a) < std::abs(b); });**

**std::sort(v.begin(), v.end(),**

**[](int a, int b) -> bool { return a > b; });**

**auto pos = std::find\_if(v.begin(), v.end(),**

**[k](int x) -> bool { return x < k; });**

**有了CTAD, 我们也可以省略返回值类型, 编译器会帮我们自动推断 非常方便快捷的定义函数!**

**int k = 42;**

**auto f = [k](int x) -> bool { return x < k; };//k是需要捕获的外部变量**

**bool b1 = f(10); // true**

**bool b2 = f(100); // false**

**[k] 按值捕获 k（副本） [k] { return k + 1; }**

**[&k] 按引用捕获 k（可修改原变量） [&k] { k \*= 2; }**

**[=] 按值捕获所有外部变量 [=] { return x + y; }**

**[&] 按引用捕获所有外部变量 [&] { ++x; ++y; }**

**[this] 捕获当前类的 this 指针 [this] { return member; }**

**To capture more variables:**

**auto pos = std::find\_if(v.begin(), v.end(), [lower, upper](int x) { return lower <= x && x <= upper;});**

**To capture by reference (so that copy is avoided)**

**std::string str = someString(); std::vector<std::string> wordList;**

**// finds the first string that is lexicographically greater than `str`, // but shorter than `str`.**

**auto pos = std::find\_if(wordList.begin(), wordList.end(), [&str](const std::string &s) { return s > str && s.size() < str.size();});**

**Here &str indicates that str is captured by referece.**

**继承方式 基类 public 成员在子类变为 基类 protected 成员在子类变为 基类 private 成员**

**public public protected 不可访问**

**protected protected protected 不可访问**

**private private private 不可访问**

**子类对基类成员的访问权限**

**public 和 protected 成员：子类可直接访问。**

**private 成员：子类不可直接访问（需通过基类公有/保护接口）。**

**构造与析构顺序**

**构造顺序：基类 → 子类成员变量 → 子类构造函数。**

**析构顺序：子类析构函数 → 子类成员变量 → 基类析构函数。**

**函数重写（Override）与多态**

**虚函数（virtual）：允许子类重写基类方法，实现运行时多态。**

**override 关键字（C++11）：显式标记重写，增强可读性并防止错误。**

**多重继承：一个子类继承多个基类（可能引发命名冲突）。**

**菱形继承问题：使用虚继承（virtual）解决重复基类成员问题。**

**class A { public: int data; };**

**class B : virtual public A {}; // 虚继承**

**class C : virtual public A {};**

**class D : public B, public C {}; // 仅一份 A 的成员**

**int main() {**

**D d;**

**d.data = 42; // 合法：无二义性**

**class DiscountedItem : public Item {**

**int m\_minQuantity = 0;**

**double m\_discount = 1.0;**

**public:**

**DiscountedItem(const std::string &name, double price,**

**int minQ, double disc)**

**: Item(name, price), m\_minQuantity(minQ), m\_discount(disc) {} };**

**不允许这样写：**

**DiscountedItem(const std::string &name, double price,**

**int minQ, double disc)**

**: m\_name(name), m\_price(price), m\_minQuantity(minQ), m\_discount(disc) {}**

**在派生类自己的数据成员初始化之前，基类子对象 必须 通过调用其构造函数来初始化。**

**如果没有显式调用基类的构造函数会发生什么？**

**基类的默认构造函数会被调用。**

**如果基类不可默认构造，则报错。**

**这个构造函数做了什么？**

**[DiscountedItem() = default;]**

**调用 Item::Item() 默认初始化基类子对象，然后初始化 m\_minQuantity 和 m\_discount。**

**class Item {**

**public:**

**// 由于 `Item` 有用户声明的构造函数，它没有默认构造函数。**

**Item(const std::string &name, double p) : m\_name(name), m\_price(p) {}**

**class DiscountedItem : public Item {**

**public:**

**DiscountedItem(const std::string &name, double p, int ma, double disc)**

**// 在进入函数体之前，会调用 `Item::Item()` --> 错误！ };**

**基类 Item 的情况：**

**Item 声明了一个带参数的构造函数 Item(const std::string &name, double p)，因此编译器不会自动生成默认构造函数（即 Item() 不存在）。**

**如果派生类不显式调用基类的构造函数，编译器会尝试调用基类的默认构造函数（但这里没有）。**

**派生类 DiscountedItem 的问题：**

**在 DiscountedItem 的构造函数中，没有通过初始化列表显式调用 Item 的构造函数（如 Item(name, p)）。**

**编译器会隐式尝试调用 Item 的默认构造函数（但 Item 没有默认构造函数），因此报错。**

**如果 D 是 B 的子类：**

**B\* 可以指向 D 对象，**

**B& 可以绑定到 D 对象。**

**DiscountedItem di = someValue();**

**Item &ir = di; // 正确**

**Item \*ip = &di; // 正确**

**原因：is-a 关系！D 是一种 B。**

**但在这样的引用或指针上，只能访问 B 的成员。**

**void printItemName(const Item &item) {**

**std::cout << "Name: " << item.getName() << std::endl;**

**}**

**DiscountedItem di("A", 10, 2, 0.8);**

**Item i("B", 15);**

**printItemName(i); // "Name: B"**

**printItemName(di); // "Name: A"**

**const Item &item 可以绑定到 Item 或 DiscountedItem。**

**表达式的静态类型：编译时已知的类型。**

**表达式的动态类型：表达式所代表的对象的实际类型。运行时才可知。**

**表达式 item 的静态类型是 const Item，但其动态类型直到运行时才可知。（可能是 const Item 或 const DiscountedItem。）**

**class Item {**

**public:**

**virtual double netPrice(int cnt) const {**

**return m\_price \* cnt; } // 其他成员 };**

**class DiscountedItem : public Item {**

**public:**

**double netPrice(int cnt) const override {**

**return cnt < m\_minQuantity ? cnt \* m\_price : cnt \* m\_price \* m\_discount; } // 其他成员 };**

**注意：auto 不能用于推导 virtual 函数的返回类型。**

**class DiscountedItem : public Item {**

**virtual double netPrice(int cnt) const override; // 正确，显式 virtual };**

**class DiscountedItem : public Item {**

**double netPrice(int cnt) const; // 也正确，但不推荐 };**

**override 关键字让编译器检查并报告函数是否真正覆盖。**

**当释放内存时需要动态确定删除的类型就需要虚析构函数**

**智能指针通过包装原始指针实现，因此也可用于动态绑定。**

**std::vector<std::shared\_ptr<Item>> myItems;**

**for (auto i = 0; i != n; ++i) {**

**if (someCondition) { myItems.push\_back(std::make\_shared<Item>(someParams));**

**} else { myItems.push\_back(std::make\_shared<DiscountedItem>(someParams)); } }**

**std::unique\_ptr<Derived> 可以隐式转换为 std::unique\_ptr<Base>。  
std::shared\_ptr<Derived> 可以隐式转换为 std::shared\_ptr<Base>。**

**动态绑定仅发生在基类的引用或指针上。**

**DiscountedItem di("A", 10, 2, 0.8);**

**Item i = di; // 会发生什么？**

**auto x = i.netPrice(3); // 调用哪个 netPrice？**

**Item i = di; 调用 Item 的拷贝构造函数**

**但 Item 的拷贝构造函数仅处理基类部分。**

**因此 DiscountedItem 自己的成员被忽略，或被“切片”掉。**

**i.netPrice(3) 调用 Item::netPrice。**

**如果一个类有（声明或继承）至少一个虚函数，则称其为 多态类。**

**无论是虚普通成员函数还是虚析构函数都可以。**

**如果一个类是多态的，所有从它派生的类也是多态的。**

**无法“拒绝”继承任何成员函数，因此虚成员函数必须被继承。**

**如果基类的析构函数是虚的，派生类的析构函数也必须是虚的。**

**向下转型：仅适用于多态类**

**dynamic\_cast<Target>(expr)。**

**Base \*bp = new Derived();**

**Derived \*dp = dynamic\_cast<Derived \*>(bp);**

**Derived &dr = dynamic\_cast<Derived &>(\*bp);**

**Target 必须是引用或指针类型。**

**dynamic\_cast 会执行运行时类型识别（RTTI）来检查表达式的动态类型。**

**如果动态类型是 Derived 或其派生类（直接或间接），向下转型成功。**

**否则，向下转型失败。如果 Target 是指针，返回空指针；如果是引用，抛出 std::bad\_cast 异常。**

**如果向下转型保证成功，可以使用 static\_cast：**

**auto dp = static\_cast<Derived \*>(bp); //比 dynamic\_cast 快但不进行检查。如果动态类型不是Derived则行为未定义。**

**纯虚函数**

**如果一个虚函数在基类中没有合理的定义，应该通过写作 =θ=*θ* 将其声明为纯虚函数。**

**class Shape {**

**public:**

**virtual void draw(ScreenHandle &) const = \(\theta\);**

**virtual double area() const = \(\theta\);**

**virtual double perimeter() const = \(\theta\);**

**virtual ~Shape() = default; };**

**任何包含纯虚函数的类都是抽象类。纯虚函数（通常）不能被调用11，且抽象类不能被实例化。**

**Shape shape; // 错误。**

**Shape \*p = new Shape; // 错误。**

**auto sp = std::make\_shared<Shape>(); // 错误。**

**std::shared\_ptr<Shape> sp2 = std::make\_shared<Rectangle>(p1, p2); // 正确。**

**非纯虚函数必须被定义。否则，编译器将无法生成必要的运行时信息（虚表），从而导致错误。**

**class X {**

**virtual void foo(); // 声明，未定义**

**// 即使 `foo` 未被使用，也会导致错误。 };**

**让接口健壮，不易出错。**

**class Shape {**

**public:**

**virtual double area() const {**

**throw std::logic\_error("area() called on Shape!");**

**//而非隐式的return0不返回任何信息错误会无声地发生！} };**

**如果意外调用了 Shape::area，会抛出异常。**

**然而，好的设计应该让错误在编译时就被捕获。**

**[最佳实践] 如果错误能在编译时捕获，就不要留到运行时。**

**运行时多态体现在继承类虚函数覆盖 编译多态体现在模板**

**纯虚函数仅指定接口继承。父类提供接口但不提供实现实现在子类。**

**简单（非纯）虚函数指定接口继承 + 默认实现。子类可以使用默认实现且默认实现可以被重写。**

**非虚函数指定接口继承 + 强制实现。**

**注意：在公有继承中，*接口总是被继承*。**

**纯虚函数是通过 =θ=*θ* 声明的虚函数。**

**调用纯虚函数是不允许的。**

**纯虚函数定义接口并强制派生类重写它。**

**包含纯虚函数的类是抽象类。不能创建抽象类的对象。**

**抽象类通常用于表示抽象、通用的概念。**

**在C++中，抽象类（不能被实例化的类）通常通过包含至少一个纯虚函数（用= 0声明）来实现。但有时我们希望一个类成为抽象类，而它并没有其他需要声明为纯虚的函数。这时可以通过以下方式实现：将析构函数声明为纯虚函数，并为其提供定义：**

**struct Foo {**

**virtual ~Foo() = 0;**

**};**

**Foo::~Foo() = default; // 在类外提供定义。**

**容器适配器：std::stack、std::queue、std::priority\_queue**

**分别表示栈、队列和优先队列数据结构。**

**它们本身不是容器，而是基于某些底层容器，并提供相应数据结构的接口。**

**序列容器**

**std::vector<T>：动态连续数组（我们非常熟悉）**

**| v[0] | v[1] | v[2] | v[3] | ... |**

**std::deque<T>：双端队列（通常发音为“deck”）**

**std::deque<T> 支持在开头和结尾快速插入和删除。（push\_front、pop\_front、push\_back、pop\_back）**

**| ... | d[0] | d[1] | d[2] | d[3] | ... |**

**std::array<T, N>：与 T[N] 相同，但它是一个容器**

**永远不会退化为 T\*。提供容器接口：.at(i)、.front()、.back()、.size() 等，以及迭代器。**

**std::list<T>：双向链表**

**std::list<T> 支持在容器中任意位置快速插入和删除，**

**但不支持快速随机访问（即没有 operator[]）。支持双向遍历。无法直接计算地址：由于内存不连续，无法通过基地址 + 偏移量（O(1)）直接访问第 i 个元素。任意位置插入/删除：链表只需修改相邻节点的指针，时间复杂度为 O(1)（前提是已定位到插入/删除位置）。**

**std::forward\_list<T>：单向链表**

**旨在节省时间和空间（与 std::list 相比）。仅支持向前遍历。**

**STL容器具有一致的接口**

**c.at(i)、c[i]：访问索引为 i 的元素。at 会进行边界检查，如果 i 超出有效范围，则抛出 std::out\_of\_range。**

**c.front()、c.back()：访问首/尾元素。**

**大小和容量：c.size()和 c.empty()**

**c.resize(n)、c.resize(n,x)：调整容器大小，使其恰好包含 n 个元素。如果 n>c.size()，将追加 n−c.size()个元素。**

**c.resize(n)：追加的元素是值初始化的。**

**c.resize(n,x)：追加的元素是 *x* 的副本。**

**c.capacity()、c.reserve(n)、c.shrinktofit()：仅适用于 string 和 vector。**

**c.capacity()返回容量（当前存储中可以存储的元素数量）。**

**c.reserve(n)：预留至少 n*n* 个元素的空间。**

**c.shrinktofit()：请求移除未使用的容量使 c.capacity()=c.size()。**

**修改器：**

**c.push\_back(x)、c.emplace\_back(args...)、c.pop\_back()：在容器末尾插入/删除元素。**

**c.push\_front(x)、c.emplace\_front(args...)、c.pop\_front()：在容器开头插入/删除元素。**

**c.clear()：移除容器中的所有元素。**

**c.insert(...)、c.emplace(...)、c.erase(...)：在指定位置插入/删除元素。**

**警告：对于需要维护连续存储的容器（string、vector、deque），在中间插入和删除可能非常慢（O(n)*O*(*n*)）。**

**某些成员函数在某些容器上不受支持，具体取决于底层数据结构。例如：**

**任何修改容器长度的操作对 array 都不允许。**

**push\_front、emplace\_front 和 pop\_front 在 string、vector 和 array 上不受支持。**

**forward\_list 不支持 size，以节省时间和空间。**

**operator[] 和 at 在链表上不受支持。**

**auto 来帮忙！  
c.begin() 返回指向容器 c 第一个元素的迭代器。  
c.end() 返回指向容器 c 最后一个元素之后的迭代器**

**Container c(b, e)，其中 [b, e) 是一个迭代器范围。**

**从迭代器范围 [b, e) 复制元素。**

**Container c(n, x)，其中 n 是非负整数，x 是一个值。**

**用 n 个 x 的副本初始化容器。**

**Container c(n)，其中 n 是非负整数。**

**用 n 个元素初始化容器。所有元素都是值初始化的。**

**string 不支持此方式，因为拥有 n 个值初始化的 char（所有都是 '\0'）毫无意义。**

**动机：set表示一个“集合”：**

**快速插入、查找和删除元素。顺序不重要。**

**序列容器无法满足：**

**查找元素的时间复杂度为 O(n)*O*(*n*)。**

**快速插入/删除仅在某些容器的特定位置支持。**

**元素的顺序被保留，但这并不重要。**

**定义在 <set> 中。**

**std::set<T> 是一个元素类型为 T 的集合。需要支持 operator<(const T, const T)，因为它通常用红黑树实现。**

**std::set<T, Cmp> 也可用。x < y 会被替换为 cmp(x, y)，其中 cmp 是类型为 Cmp 的函数对象。**

**std::set<int> s1; // 一个空的 int 集合  
std::set<std::string> s2{"hello", "world"}; // 一个字符串集合，初始化为两个元素  
struct Student { std::string name; int id; };  
std::set<Student> s3; // Student 没有可用的运算符。仅此一行不会报错，但无法插入元素。  
s3.insert(Student{"Alice", 42}); // 错误：无可用运算符。**

**struct Student { std::string name; int id; }; struct CmpStudentByName { bool operator()(const Student &a, const Student &b) const { return a.name < b.name; } }; std::set<Student, CmpStudentByName> students; // OK students.insert(Student{"Alice", 42}); // OK这里的结构体是一个callable的对象可以作为比较器**

**std::set s1{a, b, c, ...}; // 元素类型根据列表推导  
std::set s2(begin, end); // 元素类型根据 begin 和 end 指向的元素类型推导**

**此外，std::set 是可复制构造、可复制赋值、可移动构造和可移动赋值的，就像我们学过的序列容器一样。**

**std::set 不包含重复元素。这些构造函数会忽略重复元素。**

**插入：insert 和 emplace。重复元素不会被插入。**

**s.insert(x)、s.insert([a, b, ...])、s.insert(begin, end)。**

**std::set s{3, 2, 5, 5, 1}; // {1, 2, 3, 5}。重复的 5 被移除。  
std::cout << s.size() << std::endl; // 4  
s.insert(42); // {1, 2, 3, 5, 42}  
s.insert(42); // 未插入任何内容。（无错误。）  
int a[]{10, 20, 30};  
s.insert(a, a + 3); // 迭代器范围。s 现在包含 {1, 2, 3, 5, 10, 20, 30, 42}。  
s.insert({11, 12}); // {1, 2, 3, 5, 10, 11, 12, 20, 30, 42}。**

**s.emplace(args...)。将参数 args... 转发给元素类型的构造函数，就地构造元素。**

**std::setstd::string s;  
s.emplace(10, 'c'); // 插入字符串 "cccccccccc"**

**s.insert(x) 和 s.emplace(args...) 返回 std::pair<iterator, bool>：**

**成功时，.first 是指向插入元素的迭代器，.second 是 true。**

**失败时，.first 是指向阻止插入的元素的迭代器，.second 是 false。**

**元素按升序排列：以下断言始终成立（如果 tmp 和 iter 都可解引用）。**

**auto tmp = iter;  
++iter;  
assert(\*tmp < \*iter);**

**s.erase(x)、s.erase(pos)、s.erase(begin, end)，其中 pos 是指向 s 中某个元素的迭代器，[begin, end) 是 s 中的迭代器范围。**

**s.erase(x) 移除与 x 等价的元素（如果存在）。**

**返回 0 或 1，表示移除的元素数量。**

**std::set<int> s{5, 5, 7, 3, 20, 12, 42};  
std::cout << s.erase(42) << std::endl; // 4被移除。输出：1 s 现在是 {3, 5, 7, 12, 20}。  
s.erase(++++s.begin()); // 7 被移除。**

**s.find(x) 返回指向与 x 等价的元素的迭代器（如果找到），否则返回 s.end()（如果未找到）。**

**std::set<int> s = someValues();  
if (s.find(x) != s.end()) // x 被找到  
// ...**

**元素自动排序。不支持像 v[i] 这样的快速随机访问。**

**map表示一个映射：f:S→T*f*:*S*→*T*。**

**std::map<std::string, int> counter; // 将每个字符串映射到一个整数  
std::string word;  
while (std::cin >> word)  
++counter[word]; // !!**

**现在对于任何字符串 str，counter[str] 是一个整数，表示 str 出现的次数。**

**如果忽略 Value，它就是一个 std::set<Key>。**

**不允许重复键。需要 operator<(const Key, const Key)。**

**元素按键的升序存储。键不能直接修改。**

**std::map<Key, Value> 的元素类型是 std::pair<const Key, Value>。**

**\*iter 返回 std::pair<const Key, Value> &。**

**std::map<Key, Value> m{{key1, value1}, {key2, value2}, ...};**

**std::map<Key, Value> m(begin, end)，但元素应为 pair：**

**std::vector<std::pair<int, int>> v{{1, 2}, {3, 4}};  
std::map<int, int> m(v.begin(), v.end());**

**m.insert({key, value})**

**m.insert({{key1, value1}, {key2, value2}, ...})**

**m.insert(begin, end)**

**使用基于范围的 for 循环：**

**for (const auto &[word, occ] : counter)**

**std::cout << word << " 出现了 " << occ << " 次。\n";**

**m[key] 查找键为 key 的键值对。**

**如果该键不存在，则插入 {key, Value{}} - 值是值初始化的。然后，返回对该值的引用。**

**std::map<std::string, int> counter;  
std::string word;  
while (std::cin >> word)  
++counter[word]; // 如果 word 不在 counter 中，  
// 会先插入一个 {word, 0}。**

**命名空间定义末尾无需分号 ;。**

**扩展：可以重新打开命名空间以添加更多声明。**

**namespace namespace\_name { int a; // Variable declaration void add() { // Function declaration // ... }c lass ClassName { // Class declaration // ... }; }**

**运算符 :: 用于显式指定标识符的作用域，确保在多个作用域定义同名标识符时的清晰性。**

**访问全局变量：当局部变量遮蔽全局变量时。**

**int x = 10; // 全局变量  
int main() {int x = 20; // 局部变量  
std::cout << ::x; // 输出全局变量 'x' (10)}**

**using 声明将另一个命名空间中的特定名称引入当前作用域，使代码更简洁而不失清晰性。**

**在命名空间作用域中**

**void f();namespace A {void g();}**

**namespace X {  
using ::f; // 全局 'f' 可作为 X::f 访问。  
using A::g; // A::g 可作为 X::g 访问。}**

**using 指令将指定命名空间中的所有标识符导入当前作用域，使其可以直接访问。与 using 声明不同，using 指令不会将名称添加到声明区域，因此不会阻止标识符的重新声明。重要提示：避免在头文件或全局作用域中使用 using 指令，以免引发名称冲突。**

**namespace A {  
namespace B {int k = 9; // A::B::k}  
using namespace B;  
int k = 10; // A::k  
int l = k; // 错误  
}在命名空间 A 中引用 k 会产生歧义，因为 A::k 和 A::B::k 都可见。**

**namespace B {int k = 9; // B::k}**

**namespace A {using namespace B;  
int k = 10; // A::k  
int l = k; // 正确  
}**

**• 当前作用域中引入的名称总是优先于外部作用域的名称。  
• 在此例中，A::k 隐藏了 B::k。**

**使用 using std::cin; 仅将 std::cin 引入当前作用域，可直接用 cin 引用。**

**使用 using namespace std; 将所有 std 中的标识符导入当前作用域，但会增加名称冲突的风险。**

**连接运算符 (operator+) 为 std::string 重载。至少一个操作数必须是 std::string 类型才能调用适当的重载。注意：字符串字面量（类型为 const char[N]）不能直接用 + 连接另一个字面量，因为两个操作数必须转换为或本身就是 std::string。例如，以下代码会导致编译错误：**

**std::string str = "Hello " + "World"; // 错误：无匹配的 operator+**

**std::vector 的动态增长可能重新分配内部存储，这会使所有指向其元素的指针、引用和迭代器失效。**

**for (int i : vec)  
if (i % 2 == 0)  
vec.push\_back(i + 1);**

**此循环不安全，因为依赖的迭代器可能在 push\_back 时因向量调整大小而失效。**

**切勿在使用基于范围的 for 循环迭代容器时修改其大小。**

**实现 rstrip**

**std::string rstrip(std::string str) {  
while (!str.empty() && std::isspace(str.back()))  
str.pop\_back();  
return str;}**

**for (char c : str)c = std::tolower(c);**

**此方法无效，因为 c 是 str 中每个字符的副本。修改 c 不会影响原字符串。要修改原始元素，必须在循环中使用引用。**

**the following are not allowed in C++:**

**Arrays of references Pointers to references**

**References to references**

**int& a[3]; // error: array of references**

**int&\* p; // error: pointer to reference**

**int& &r; // error: reference to reference**

**void print\_array(const int (&array)[10]) {//只读：const 限定**

**int a[10], ival = 42, b[20];  
print\_array(a); // 正确：a 是 int[10]  
print\_array(&ival); // 错误：&ival 不是数组  
print\_array(b); // 错误：b 是 int[20]**

**int& a,b a是引用 b是int**

**默认初始化：内存未初始化。**

**值初始化：确保内存初始化为零**

**int \*pil = new int; // 默认初始化（值不确定）。  
int \*pil2 = new int(); // 值初始化（内置类型为零）。**

**内置类型（如 int、float、指针等）：**

**全局作用域或命名空间作用域的静态变量会被零初始化（默认值为 0/nullptr）。**

**static int a; // 初始化为 0**

**static char\* ptr; // 初始化为 nullptr**

**函数内的 static 局部变量如果没有显式初始化，也会被零初始化：**

**void func() {**

**static int b; // 初始化为 0**

**}**

**类类型：**

**调用默认构造函数。如果没有默认构造函数，则编译错误。**

**static std::string s; // 调用默认构造函数，初始化为空字符串**

**struct Data {**

**Data(size\_t s) { printf("默认构造\n"); };**

**Data(const Data&) { printf("拷贝构造\n"); }**

**Data(Data&&) { printf("移动构造\n"); }**

**size\_t s; int\* b; };**

**const Data getData(size\_t s) { return Data(s); }**

**auto d1 = Data(42);//自c++17起，强制拷贝省略，即没有拷贝省略时，先data(42)default，再”=”拷贝构造，而强制拷贝省略时，d1 直接再data(42)处默认构造**

**auto d2 = std::move(d1);//直接在d1移动处构造**

**auto d3 = getData(42);//默认构造，d3直接绑定到return处**

**auto d4 = std::move(getData(42));//默认构造，再移动构造**

**struct Data { Data(int x);int x\_; };**

**void foo(Data&& x) { x = 42; }**

**int main() { foo(Data(73)); return 0; }**

**实体在不同上下文中可能有不同的值类别：**

**函数调用期间（foo(Data(73))）：**

**调用构造函数，创建未命名的临时对象 Data(73)。**

**Data(73) 绑定到右值引用 x。**

**在 foo 的作用域内，x 是一个有名称的实体，因此是左值。**

**有标识（glvalue） 无标识**

**不可移动 左值 —**

**可移动（右值）xvalue prvalue**

**左值（定位值）：指定一个对象（特定内存位置）**

**x, arr[0], \*ptr**

**纯右值（prvalue）：表示无标识的值：42, a + b, func()**

**将亡值（xvalue）：表示生命周期即将结束的对象**

**std::move(a), static\_cast<int&&>(a)**

**为什么 ++i = 42 合法，而 i++ = 42 不合法？**

**++i 返回引用：++i 返回的是 i 的引用是一个左值可以赋值。**

**i++ 返回值：i++ 返回的是 i 的当前值是一个右值不能赋值。**

**前置递增运算符（++i）返回左值。**

**后置递增运算符（i++）返回右值。**

**标准引用**

**前置递增运算符：返回 \*this 的引用。**

**后置递增运算符：返回 \*this 的副本。**

**左值——表示对象的表达式**

**对象访问：指针解引用：\*ptr；数组下标：arr[i]**

**产生左值的运算符：前置递增/递减：++i, --j（因此 ++i = 42 是合法的）；赋值：a = b 返回对 a 的左值引用**

**命名实体：变量、函数、静态数据；静态内存中的字符串字面量（如 "hello"）**

**右值——表示值的表达式**

**临时对象——函数调用的结果std::string fun();**

**std::string s = fun(); // fun() 返回临时右值**

**字面常量——整数、浮点数、字符字面量**

**算术/逻辑表达式——a + b, x \* y, cond1 && cond2**

**函数式转换——Type(args…) 构造临时对象**

**std::string temp = std::string(10, 'c');**

**int n = int(x); 注意： 内置转换和算术运算总是产生右值。**

**解引用操作符 \* 的返回值**

**如果解引用的指针是普通指针（如 T\*），则 \*ptr 返回的是左值引用（T&）。如果解引用的指针是右值引用指针（如 T\*&&），则 \*ptr 返回的是右值引用（T&&）**

**++ival 返回 ival 的引用 → 左值。**

**ival++ 返回原始值 → 右值。**

**\*&ptr 返回 ptr 的引用 → 左值。**

**&\*ptr 返回 ptr 存储的地址；结果是值而非 ptr 本身 →右值。**

**ptr[ival] 和 \*(ptr + ival) 等价；均返回 ptr[ival] 的引用→ 左值**

**成员函数：this 和 const**

**若 ptr 类型为 const T\* 且 T 包含类型为 U 的数据成员 m，则 ptr->m 类型为 const U。const 具有传递性。const 对象只能调用 const 成员函数；不能调用非 const 函数。**

**class Rational {public:bool operator==(const Rational&) const; };**

**bool operator!=(const Rational&, const Rational&);**

**非成员运算符接受两个 const 引用。**

**成员运算符接受一个 const 参数，且自身为 const**

**I/O 运算符必须为非成员函数，因为无法修改 std::istream 或 std::ostream：**

**std::istream（如 std::cin）和 std::ostream（如 std::cout）是 C++ 标准库中的类，它们的对象（如 std::cin 和 std::cout）是全局对象，无法被修改。如果 I/O 运算符是成员函数，那么它们的左操作数必须是调用该运算符的对象。例如：**

**class Rational {**

**public: std::istream& operator>>(std::istream&); // 成员函数};**

**这样使用时：**

**Rational r;r >> std::cin; // 语法不符合习惯**

**（1）流对象不可拷贝**

**std::istream 和 std::ostream 是不可拷贝的类。如果通过值传递流对象，会导致编译错误。因此，必须通过引用传递流对象。**

**（2）流对象需要被修改**

**I/O 操作会修改流对象的状态（如读取或写入数据）。如果流对象是 const 的，则无法修改其状态。因此，流对象必须通过非 const 引用传递。**

**（3）链式调用**

**精确匹配**

**左值到右值转换：void f(int); int x = 42; f(x);**

**数组到指针转换：void f(int \*a); int ar[10]; f(ar);**

**函数到指针转换：void f(int, fp); int g(int); f(5, g);**

**限定转换：void g(const int \*); int \*p = &a; g(p);**

**重载解析：提升：整型提升、浮点提升。**

**char → int；float → double**

**enum → int/short/unsigned int/... bool → int**

**重载解析：转换：整型转换、浮点转换、指针转换、布尔转换等。**

**unsigned int → short char\* → void\***

**double → float int → bool**

**多参数重载解析**

**若函数 F1 在某个参数上的转换优于 F2，且其他参数不劣于 F2，则 F1 胜出。**

**int fun(int, int, int); // F1**

**int fun(double, double, double); // F2**

**fun(5, 5, 2.0); // 歧义**

**int fun(int, int, double); // F1**

**int fun(int, double, double); // F2**

**fun(5, 5, 5); // 调用 F1**

**F1 在第二个参数上更优，其他参数不劣于 F2。**

Dynarray &operator=(Dynarray other) noexcept {

swap(\*this, other);

return \*this;

} 值传递同时满足左值右值，直接走形参的构造

参数为左值时调用拷贝构造，为右值时调用移动构造。

因此，此赋值运算符同时支持拷贝和移动赋值。

**仿函数（operator()）**是定义了函数调用运算符的对象，属于**可调用对象**之一（另一类是 lambda 表达式）。

struct Adder {int base; int operator()(int x) const { return base + x; } };

Adder add5(5);

int result = add5(10); // 调用 add5.operator()(10) → 15

**运算符 ->**对内置类型，**E1->E2** 等价于 **(\*E1).E2**。

用户定义的 **operator->** 需返回原始指针或链式调用 **operator->**

**结构化绑定 (Structured binding)**

结构化绑定的声明可以带有 **const**、引用等，也可以用于基于范围的 for 循环：

enum class Gender { Male, Female };

struct PersonInfo { std::string name; Gender gender; int birthYear; };

void foo(const std::vector<PersonInfo> &persons) {

for (const auto &[n, g, by] : persons) { // ... } }

**（1）for 循环的语义**for (const auto &[n, g, by] : persons) 是一个范围 for 循环，用于遍历 persons 中的每个元素。

**（2）结构化绑定**const auto &[n, g, by] 是结构化绑定语法，用于将 PersonInfo 对象的成员解包到变量 n、g 和 by 中。

n 绑定到 PersonInfo::name。g 绑定到 PersonInfo::gender。

by 绑定到 PersonInfo::birthYear。

**（3）const auto &**const 表示绑定后的变量是只读的，不能修改 PersonInfo 对象的成员。auto & 表示绑定后的变量是引用类型，避免拷贝。

比较元素的算法通常默认使用 **<**，但也提供接受二元谓词 **cmp** 的版本，用 **cmp(a, b)** 替代 **a < b**。带 **\_if** 后缀的函数接受一元谓词 **cond**，仅处理 **cond(element)** 为真的元素

C语言：int main(void){ int a = 1; int b = (a++, a + 100, a - 10); }运用“，“运算符依次执行最后b=8

**C语言中合法指针转换：**

**void\* 与其他指针类型的相互转换。**

**char\* 与其他指针类型的相互转换。**

**相同类型的指针转换。**

**可能导致未定义行为的指针转换：**

**不兼容类型的指针转换。对齐问题。**

**char 类型的 ++ 操作：**

**char 是整数类型，++ 的行为与 int 类型相同。**

**++ 操作可能导致溢出，行为是实现定义的。**