#define预处理阶段 **const**运行时常量 **constexpr**编译时常量，必须能在编译期求值Compile-time evaluated in compilation
#pragma once 防止头文件被包含多次 Prevent header files from being included multiple times

```
int arr[3] = {1,2,3};// 栈数组
int* heapArr = new int[3]; // 堆数组
heapArr[0] = 5; delete[] heapArr;// 释放内存
```

```
void addByValue(int x){ x += 10; }// 不改变原变量
void addByRef(int& x){ x += 10; }// 改变原变量
int n = 5;
addByValue(n);// n = 5 addByRef(n);// n = 15
```

struct与**class**基本相同，默认访问级别不同。struct默认公class默认私。struct用于数据集合，class用于封装逻辑

特性	解释	示例关键点
封装**Encapsulation**	把数据和函数封成一体	`private` 数据，`public` getter/setter
抽象**Abstraction**	只暴露必要信息	`public` 接口隐藏实现
继承**Inheritance**	子类获得父类属性/行为	`class Dog : public Animal`
多态**Polymorphism**	父类指针可调用子类重写函数	`virtual void speak()`

成员初始化列表语法比在构造体内赋值更高效。成员变量会按照声明顺序初始化。构造函数参数可以定义默认值func(x = n)
拷贝构造在用一个对象创建另一个对象"时自动调用，析构函数销毁对象时自动执行释放资源。若有virtual，析构也virtual防止内存泄露
const - 允许复制可变对象和不可变对象 **&**防止出现无限的内部复制循环

```
#include <iostream> using namespace std; class Complex {private:double r, i;
public:Complex(double real = 0, double imag = 0) : r(real), i(imag) {}
Complex(const Complex& c) : r(c.r), i(c.i) {} ~Complex() { cout << "销毁" << endl; }};
int main(){Complex c1(2,3);Complex c2 = c1;return 0; //(c2)销毁 (c1)销毁}
```

继承:子类继承父类的实现 `class Car : public Vehicle,` 子类成员访问性取其与父类成员最小值，但仅改变通过子类访问的成员，不改变父类原本访问性。| **override**:标记函数重写了父类的virtual函数
多态:同父类虚函数不同对象有不同实现，父类要阻止重写可以不写virtual(重载，模板属于编译时多态，不需要virtual)

```
#include <iostream> using namespace std; class Animal {
public: virtual void speak() { cout << "???" << endl; } void speak1() { cout << "???" << endl; }};
class Dog : public Animal {public:void speak() override { cout << "Woof" << endl; }
   void speak1() { cout << "Woof" << endl; }};
int main() { Animal* a = new Dog(); a->speak(); // 输出 "Woof" a->speak1(); // 输出 "???}
```

前向声明**Forward Declaration**:类A需要用到B，但B还未定义，需要提前声明B存在

```
// --- Header1.hpp ---#pragma once
class B;// 前向声明 class A {public: void setB(B* b);  // 这里只需要知道 B 存在 private: B* bPtr;};
// --- Header2.hpp ---#pragma once #include "Header1.hpp"class B {private:int data;};
```

构造函数与析构函数的调用顺序:创建:1.父类构造 2.子类构造 销毁:反过来后进先出
友元**friend**让非成员函数或其他类访问类的私有成员
运算符重载:

```
class Complex {private:double real, imag;
public:Complex(double r=0, double i=0) : real(r), imag(i) {}
   friend Complex operator+(const Complex &a, const Complex &b);};
Complex operator+(const Complex &a, const Complex &b) {
return Complex(a.real + b.real, a.imag + b.imag);}
```

抽象类:有纯虚函数的类。不能被实例化，必须被继承，子类必须重写纯虚函数(Pure Virtual Function，virtual … = 0;)
对象切片**Object Slicing**:当一个子类对象被赋值或拷贝给一个父类对象(不是指针或引用)时，子类中独有的部分会被"切掉"，只保留父类那部分，因为构造函数、拷贝构造函数不能声明为虚函数，也就意味着不支持多态

```
class Derived : public Base
Base b = d; (等价于Base b(d); )，实际调用不支持多态的Base拷贝函数，子类信息丢失，拷贝出的是父类对象
```

解决:不要值拷贝父类对象，引用或指针指向父类对象

```
Derived d; Base& ref = d;      // 父类引用绑定子类对象 Base* ptr = &d;     // 父类指针指向子类对象
ref.show();// Derived::show() ptr->show();// Derived::show()
```

并在父类禁用拷贝和赋值运算符

```
Base(const Base&) = delete;// 禁用拷贝构造 Base& operator=(const Base&) = delete; // 禁用赋值
```

任何类型都可以被throw。如果自定义异常，类需要继承 `std::exception`

```
#include <exception>#include <string>#include <iostream>
class MyError : public std::exception{
public: const char *what() const noexcept override{return "自定义异常类";}};
int main(){try{throw 404; } catch(int e){int err = e;}
   try{throw MyError();} catch(MyError e){std::string err = e.what();
   } catch (...){ std::cout << "其它类型错误";}}
```

SOLID	含义	解释
Single Responsibility Principle单一职责	一个类只负责一项功能	"一个原因导致修改"
Open/Closed Principle开闭	对扩展开放，对修改关闭	继承 + 虚函数。不要修改旧代码添加新功能，而是继承多态来扩展
Liskov Substitution Principle里氏替换	子类可替换父类使用	不破坏父类语义:火车不能飞，就不应该继承飞行器
Interface Segregation Principle接口隔离	不强迫实现不需要的接口	小而专的接口IMachine{p(),s()}×，IPrinter{p()} IScanner{s()}√
Dependency Inversion Principle依赖倒置	依赖抽象而非具体实现	高层类不依赖具体类:插座不能依赖电视，用插座和电视依赖电器

里氏替换LSP:

```
//统一接口：任何家电都必须能"通电运行"
class IAppliance {public:virtual void powerOn() = 0;virtual ~IAppliance() = default;};
//不同家电实现接口：class Television : public IAppliance {void powerOn() override {}};
class Fridge : public IAppliance {void powerOn() override {}};
//高层模块：插座：class WallSocket {IAppliance* device;
public: WallSocket(IAppliance* d) : device(d) {} void plugIn() {device->powerOn();}};
```

```
int main() {Television tv; Fridge fridge;
WallSocket socket1(&tv); WallSocket socket2(&fridge);socket1.plugIn(); socket2.plugIn();}
```
Law of Demeter（迪米特法则）：类应避免对其它类内部的了解，避免链式依赖
```
car.getEngine().getCylinder().ignite();// ❌ car.start(); // ✅
```
auto关键字：编译器自行推断类型，意味着必须初始化，可以作为方法返回值，不可以作为方法参数值
```
auto add(int x, int y) { return x + y;}
```
Ranged For：相当于Java的for-each
```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
for(int i = 0; i < v.size(); i ++){std::cout << v[i] << ' ';}
for(const int &i: v){std::cout << i << ' ';} //等同于前者
```

STL类型	示例	特点
顺序容器	array, vector, deque, list, forward_list	元素按插入顺序排列
关联容器	set, multiset, map, multimap	自动排序，基于树
无序关联容器	unordered_map, unordered_set	基于哈希表，平均 O(1) 查找

Vector动态数组
```
for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {cout << *it << endl;}
std::vector<int> v = {1, 2, 3};v.push_back(4);// {1,2,3,4}v.pop_back();// {1,2,3}
v.insert(v.begin() + 1, 99);// {1,99,2,3}v.erase(v.begin() + 2);// {1,99,3}v.clear();// {}
```
Map存放Key(唯一)，T，元素按照Key生序排序
```
map<string, int> scores = { {"Alice", 90}, {"Bob", 85}, {"Charlie", 95}};
scores["David"] = 88;  //若不存在则插入，若存在则修改 scores.insert({"Eve", 92}); scores["Alice"];
for (auto &p : scores) cout << p.first << " => " << p.second << endl;
```
Set:元素本身就是Key，因此不允许重复，存T需要T实现"<"操作
严格弱序：自定义比较函数需满足：1.自反性(Irreflexive)：a < a =False 2.反对称性(Antisymmetric)：如果 a < b，则 !(b < a) 3.传递性(Transitive)：如果 a < b 且 b < c，则 a < c | **typeof**为 复杂类型起别名
迭代器**Iterators**：用来遍历容器的对象，对于迭代器对象it使用*it访问当前元素
begin() end()指向首尾，rbegin() rend()返回最后一个元素、第一个元素之前(反向迭代器++相当于反方向)cbegin() cend()只读迭代器
容器结构发生变化时需要重新获取迭代器，否则未定义
STL算法Algorithms：只读find, count, for_each, all_of, any_of，写入copy， remove， replace， transform，排序sort, reverse, unique, partition | remove只把选中元素移到后面并返回一个指针，v.begin()到这个指针之间是保留元素，这个指针到v.end()是排除的元素，彻底删除用v.erase(remove(首指针,尾指针,目标))
Static类的静态成员：每个类只存在一份副本，所有对象共享，生命周期贯穿整个程序，使用类::成员访问。
```
class X {static int m = 5; //ERROR
static int n; //OK constexpr static intarr[] = { 1, 2, 3 }; // OK
constexpr static std::complex<double> n = {1,2}; // OK
constexpr static intk; // Error};
int X::n = 5; //OK virtual static void staticFunction() //ERROR
static void staticFunction() const //ERROR
```
复制省略**Copy Elision** is a 编译优化compiler optimization technique that avoids unnecessary copying of objects
```
class Number {public:
   Number(int n){ cout << "ctor "; }
   Number(const Number& n2){ cout << "copy "; }};
Number create(int x){
   return Number(x);}
int main(){ Number n = create(5); // 输出ctor，未调用拷贝函数}
```
返回值优化**Return Value OptimizationRVO** is a subset of copy elision. Avoids unnecessary copying of objects returned from a function
```
Number create(int x){
   Number temp(x);
   return temp; // 即使Create创建了一个temp，优化后也不会创建temp}
```
右左法则**Right-Left Rule**用来读懂复杂声明：从变量名开始，先向右看，再向左看，按优先级依次读，直到整个声明解释完。
规则顺序从变量名开始。
1. 向右看：如果右边有 [] 或 ()，先读它。**2.** 向左看：如果左边有 *，再读它。**3.** 当遇到括号 () 包裹的部分时，把它当作一个整体，跳出再继续应用规则。4.最终把所有部分组合起来，读成一句话。
```
int (**var[])(); "1.var 2([]).is an array 3.(*)of pointers 4.(*)to pointers 5.(())to a function 6.(int)that returns int"
```

```
int * (* (*fp1) (int) ) [10];
```

Start from the variable name (fp1)
Nothing to right but) so go left to find * (is a pointer)
Jump out of parentheses and encounter (int) (to a function that takes an int parameter)
Go left, find *(and returns a pointer)
Jump put of parentheses, go right and hit [10] (to an array of 10)
Go left find * (pointers to)
Go left again, find int (ints)

```
int *( *( *arr[5]) ()) ();
```

Start from the variable name (arr)
Go right, find array subscript (is an array of 5)
Go left, find * (pointers)
Jump out of parentheses, go right to find () (to functions)
Go left, encounter * (that return pointers)
Jump out, go right, find () (to functions)
Go left, find * (that return pointers)
Continue left, find int (to ints).