C++ 基础面试题

更新 2024-07-25

1 请问拷贝构造与赋值的区别是什么?
拷贝构造函数用于创建一个新的对象，并使用另一个同类型的对象来初始化它；
赋值操作符 将一个对象的值 赋给另一个同类型的对象，并没有创建新对象。

2 请问拷贝赋值为什么要首先判断自赋值?
判断自赋值是为了避免内存泄漏和保证数据完整性，
对于包含动态资源的类，还可以确保正确释放和重新分配资源。

3 C++为啥要用extern "C"?
C++函数默认会有命名倾轧，用来支持函数重载等特性。
extern "C"后面跟的大括号里的内容表示会按照C语言链接规则来处理，以便与C语言库进行兼容。

4 一个类有几张虚函数表?
一般情况下，一个类只有一张虚函数表。虚函数表是针对类而言的，每个类都有自己的虚函数表。
如果一个类具有多个直接或间接基类，它可能会包含多张虚函数表，每张虚函数表对应一个基类。

5 一个对象有几个虚函数表指针?
一般情况下，每个对象通常只有一个虚表指针，指向其所属类的虚函数表。
如果一个类使用了多重继承或虚拟继承，那么对象可能会有多个虚表指针。

6 C++如何解决棱形继承问题?
在C++中，当出现棱形继承时，即一个派生类同时继承自两个或更多个基类，可能会导致数据冗余和函数调用的二义性问题。
为了解决这个问题，C++提供了虚继承（Virtual Inheritance）机制。

7 请问public继承中， 父类的private ，protected成员的区别?
在 public 继承中，父类的私有成员只能被父类内部访问，子类无法访问；而父类的保护成员可以被父类和子类访问，

8 请问类对象可以访问private proteced成员吗?
类对象不能直接访问类的私有成员和受保护成员，只有类的成员函数可以访问这些成员。

9 如何让一个类只能通过new来创建对象?
将析构函数设为 private 。

10 构造函数为什么不能设置为虚函数?
构造函数不能设置为虚函数的原因是因为在对象的构造过程中，虚函数机制尚未完全建立起来。

11 c++ 多态的实现原理是什么?

C++ 多态分为静态多台，动态多态 。静态多态 (Static Polymorphism) 也称为编译时多态或模板多态，是通过函数模板和重载来实现的。

动态多态是是通过虚函数 (virtual function) 和虚函数表 (virtual function table) 来实现的。

12 explicit 关键字的作用是什么?

explicit 关键字主要用于修饰单参数构造函数，它的作用是防止编译器进行隐式类型转换。

13 虚函数表存储在 c++ 内存模型的哪个区域?

虚函数表通常被存储在只读数据段 (.rodata) 中，这是 C++ 内存模型中的常量区。

14 将析构函数设置为 private 会怎样?

将类的析构函数声明为私有 (private)，这将阻止在栈上创建对象，只能使用堆上的动态内存分配来创建对象。

15 C++ 如何禁用拷贝构造?

1 将拷贝构造函数声明为私有 (private) 或删除 (deleted)。2 继承自不可拷贝的基类。

16 std:: list 底层实现原理是什么?

std::list 是 C++ 标准库中的一个容器，它实现了双向链表 (doubly linked list)。

17 std::map 底层是如何实现的?

std::map 的底层实现是基于红黑树 (Red-Black Tree)，这是一种平衡二叉查树。

18 std::unordered_map 底层是如何实现的?

std::unordered_map 的底层实现是哈希表 (Hash Table)。

19 std::sort 底层用了什么排序算法?

混合使用了插入排序 快速排序 堆排序等。

20 C++ 函数重写 与隐藏的区别是什么?

重写是在派生类中重新定义与基类中具有相同名称和参数列表的虚函数。需要使用 virtual 关键字声明基类函数为虚函数。

通过基类指针或引用调用虚函数时，实际执行的是派生类的版本。

隐藏发生在派生类中定义与基类中具有相同名称的非虚函数。不使用 virtual 关键字声明基类函数为虚函数。

当通过基类指针或引用调用函数时，实际执行的是基类的版本。

21 C++ 多态实现的条件是?

基类中的函数需要被声明为虚函数 (使用 virtual 关键字)。

派生类中的函数需要与基类中的虚函数具有相同的函数名、参数列表和返回类型。

使用基类的指针或引用来调用函数。

22 C++ 多态条件函数名，参数列表，都满足，只是函数返回值不同，请问会怎样?

如果派生类中的函数具有与基类中的虚函数相同的函数名和参数列表，但返回类型不同，基类没有 virtual 关键字，会导致隐藏 (hiding) 而不是多态。
如果基类有 virtual 关键字，函数名，参数列表相同返回值不同，会提示错误，协变情况会例外。

23 指针和引用的主要区别是什么?

指针是一个变量，引用是一个别名。
指针可以被初始化为一个有效的地址，也可以初始化为空指针 (即 nullptr)。
引用必须在声明时进行初始化，并且一旦绑定到某个对象，就不能重新绑定到其他对象。
通过指针可以访问原始对象，需要使用解引用操作符 *，例如 *ptr。
通过引用可以直接访问原始对象，无需使用解引用操作符。
指针可以为 nullptr，并且可以重新指向其他对象。
引用必须在声明时进行初始化，并且不能在后续使用过程中重新绑定到其他对象。
指针可能为空，需要进行空指针检查来确保指针有效性。
引用必须引用有效对象，无需担心空指针问题。

24 c++ 函数传递形参时，传递引用跟传递指针的区别?

传递引用比传递指针更简洁、安全，并且不需要空指针检查，但它也有一些限制，例如不能为空，无法重新绑定到其他对象等。
传递指针则更加灵活，可以为空，可以重新指向其他对象，但需要进行空指针检查。

25 请问以下输出结果是?

```
int a = 10;
int b = 20;
int & c = a;

c = b;
std::cout<<"a "<<a<<" b "<<b<<" c "<<c<<std::endl;
c = 15;
std::cout<<"a "<<a<<" b "<<b<<" c "<<c<<std::endl;

//a 20 b 20 c 20
//a 15 b 20 c 15
```

26 简述 c c++ 程序编译链接的步骤

C 和 C++ 程序的编译链接过程包括预处理、编译、汇编和链接。这个过程将源代码转换为可执行文件，并确保所有的符号引用都能正确地解析和访问。

1. 预处理 (Preprocessing)：预处理器根据源代码中以 # 开头的预处理指令，比如 #include、#define 等，对源代码进行处理。这些指令会被展开或替换，生成一个经过预处理的文件。
2. 编译 (Compilation)：编译器将预处理后的文件翻译成汇编语言或机器语言的目标文件 (.o 文件)。编译器会进行词法分析、语法分析、语义分析等步骤，并将代码转换成中间表示形式。

3. 汇编 (Assembly)：汇编器将编译生成的中间表示形式 (通常是汇编代码) 转化为机器可执行的目标文件。每条汇编语句对应着一条特定的机器指令。
4. 链接 (Linking)：链接器将目标文件与其他必要的库文件进行链接，生成最终的可执行文件。在链接的过程中，链接器会解析符号引用并将其与符号定义进行匹配，以确保所有的符号都能正确地解析和访问。

27 请问 *p ++ 与 (*p)++ 的区别是什么?

*p++ 表示先取出 p 指向的当前值，然后将指针 p 向后移动一个单位 (根据指针类型确定单位大小)，即将 p 指向下一个元素的地址。

因此，*p++ 表达式的值是取出的值，而 p 的值会自增一个单位。

(*p)++ 表示先取出 p 指向的当前值，然后将该值加 1，并将结果赋回到 p 指向的位置。因此，(*p)++ 表达式的值是取出的值加 1，

而 p 指向的位置的值也会加 1。

需要注意的是，这两个表达式中的 p 必须是指向可修改的内存单元的指针，否则执行过程中可能会出现未定义的行为。

同时，由于 ++ 运算符的优先级高于 * 运算符，因此在复合表达式中使用时需要注意加上括号，以明确优先级关系。

28 C++智能指针以及他们的区别。

在 C++ 中，智能指针是一种用于管理动态分配内存的指针类，它们可以自动处理资源的分配和释放，避免内存泄漏和悬空指针等问题。
C++ 标准库提供了四种主要的智能指针：std::unique_ptr、std::shared_ptr、std::weak_ptr 和 std::auto_ptr (已经被废弃)。

std::unique_ptr 用于管理独占所有权的指针，即同一时间只能有一个 std::unique_ptr 拥有指针指向的资源。不允许多个 std::unique_ptr 共享同一个资源，因此适用于单一所有权的情况。
std::shared_ptr 允许多个 std::shared_ptr 实例共享同一个资源，通过引用计数来跟踪资源的所有权。建新的 std::shared_ptr 引用计数会增加； std::shared_ptr 被销毁时，引用计数减少，直到为零时释放资源。
由于引用计数的存在，std::shared_ptr 可能会导致循环引用的问题，造成内存泄漏。
std::weak_ptr 是一种不控制对象生命周期的智能指针，用于辅助 shared_ptr，解决互相引用时发生的死锁问题，weak_ptr 不会使引用计数增加或减少。
std::auto_ptr 是 C++98/03 标准提供的智能指针，用于管理动态分配的内存。在现代 C++ 中已经被弃用。

29 智能指针什么时候会造成内存泄漏?

循环引用：如果存在两个或多个 std::shared_ptr 相互引用，形成循环引用，即使没有其他外部指针引用这些对象，它们也无法被销毁，从而导致内存泄漏。
另外使用智能指针的同时手动释放指针管理的资源也可能会造成内存泄漏.

30 为什么析构函数必须是虚函数，为什么 c++ 默认的析构函数不是虚函数?

析构函数必须是虚函数的主要原因是为了支持多态 (polymorphism) 和基类指针指向派生类对象时的正确析构行为。当基类指针指向派生类对象，
并使用基类指针删除对象时，如果基类的析构函数不是虚函数，就无法保证正确调用派生类的析构函数，从而可能导致资源泄漏或未定义行为。
C++ 默认的析构函数不是虚函数是为了提高性能和避免不必要的开销。在许多情况

下，类并不会被继承，因此将析构函数声明为虚函数可能会引入额外的空间开销和运行时开销.

31 什么是函数指针?
函数指针是指向函数的指针变量。在C语言中，函数名被解释为指向该函数的地址，因此可以通过函数指针来间接调用函数。

32 静态函数和虚函数的区别是?
静态函数是属于类而不是类的实例的函数。　　虚函数是用于实现运行时多态的机制，允许子类重新定义父类的函数。
静态函数在编译期间就确定，虚函数在运行时进行动态绑定，虚函数需要虚函数表来实现所以每次调用都会增加一次内存消耗。

33 析构函数的作用是?
析构函数是在对象生命周期结束时被调用的特殊成员函数，用于清理对象所占用的资源。在C++中，当对象超出作用域、被删除或者程序结束时，
会自动调用析构函数来释放对象所分配的资源，例如释放动态分配的内存、关闭文件等操作。析构函数的作用是确保对象在销毁时能够进行必要的清理工作，避免资源泄露和内存泄露问题。
析构函数不带任何参数，不带返回值，不能重载。如果没有定义析构函数，编译系统会自动生成一个缺省的析构函数。析构函数的调用顺序与构造时相反。

34 函数重载 重写 隐藏的区别?
函数重载是使用相同名称但参数列表不同的函数。函数重写是派生类中重新定义基类中的虚函数，实现多态性。
函数隐藏是子类中定义与基类同名函数但参数列表不同，或者同名 同参数列表但没基类没virtual 关键字，导致基类同名函数被隐藏。

35 strcpy和strlen的区别.
strcpy：字符串拷贝，会把字符串从头一直拷贝到'\0', 推荐使用strncpy. strlen:计算字符串的长度，返回从开始到'\0' 之间的字符个数，不包括'\0'.

36 C++ const修饰成员函数的目的是什么?
将成员函数声明为const的主要目的是为了告诉编译器和其他开发人员该成员函数不会修改对象的状态。

37 C++拷贝构造函数的形参能否进行值传递?
C++拷贝构造函数的形参通常使用引用传递（即const引用），而不是值传递。这是因为拷贝构造函数用于创建一个新对象，需要对已有对象进行复制和初始化，
如果使用值传递，会导致无限递归调用拷贝构造函数，最终死循环导致栈溢出。

38 C++ 类型转换是：static_cast, dynamic_cast, const_cast, reinterpret_cast 的区别
const_cast用于将const变量转为非const . static_cast用于各种隐式转换，比如非const转const，void*转指针等,
static_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

dynamic_cast用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。

向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。

reinterpret_cast几乎什么都可以转，具有较高的危险性，因为它可以忽略类型系统的限制，

将一个指针强制转换为另一个不相关的指针类型，甚至可能导致未定义行为。

39 那些函数不能声明为虚函数?

构造函数 静态成员函数 内联函数 友元函数

40 static关键字的应用.

1 全局静态变量：将全局变量声明为静态变量，使其在整个程序运行期间一直存在。全局静态变量位于静态存储区，

并且没有被初始化时，默认值为0。作用域仅限于声明它的文件内部，在其他文件中不可见。

2 局部静态变量：在函数内部将局部变量声明为静态变量，使其在函数执行结束后仍然存在。局部静态变量也位于静态存储区，

未被初始化时默认值为0。作用域仅限于声明它的函数内部，在函数执行结束后并不销毁，仍然保留其值。

3 静态函数：将函数声明为静态函数，使其只能在当前源文件中使用，不会与其他源文件中的同名函数引起冲突。

静态函数对于实现信息隐藏和封装提供了一种方式，因为它们不可见于其他源文件。

4 类的静态成员：类的静态成员是类的所有对象共享的成员，而不是某个对象的成员。通过在类中定义静态成员变量或静态成员函数，

可以在不创建对象的情况下使用它们。静态成员变量在内存中只有一份拷贝，静态成员函数不接收隐含的this指针。

41 strlen 和sizeof的区别

1.从功能定义上，strlen函数，用来求字符串的长度，返回的长度值不包含'\0'

sizeof函数是用来求指定变量或变量类型等所占用内存的大小；

2.sizeof是运算符，而strlen是C库函数strlen只能用char*做参数，且以'/0'结尾的；

对于静态数组处理：

```
char str[20]="0123456789";
strlen(str)=10;   //表示数组中字符串的长度
sizeof(str)=20;   //表示数组变量分配的长度
```

对于指针处理：

```
char *str="0123456789";
strlen(str)=10;     //表示字符串的长度
sizeof(str)=4;      //表示指针变量的所占内存大小
sizeof(*str)=1;     //表示'0'这个字符变量的所占内存大小
```

42 C++中 函数参数传递过程中，使用类对象的值传递而不传递引用会发生什么?

1) 会调拷贝构造函数来创建一个新的对象副本。这个新的对象与原始对象具有相同的值，但是它们是完全独立的对象。
2) 由于会创建新的对象副本，会导致额外的内存开销。如果类对象较大或复杂，可能会占用较多的内存空间。
3) 因为传递的是值的副本，所以对副本进行的修改不会影响原始对象。
4) 值传递通常比引用传递更耗时。特别是当类对象很大时，复制整个对象的开销会很高。

41 手写模仿C++ std::string，实现构造函数，析构函数 拷贝构造，拷贝赋值.

```
#include <iostream>
class MyString {
private:
    char *data; // 用于存储字符串
public:
    MyString(const char *str = nullptr) {
        if (str) {
            int len = strlen(str) + 1;
            data = new char[len];
            strncpy(data, str,len);
        } else {
            data = new char[1];
            *data = '\0';
        }
        std::cout<<"构造函数"<<std::endl;
    }

    MyString(const MyString &other) {
        int len = strlen(other.data) + 1;
        data = new char[len];
        strncpy(data, other.data,len);

        std::cout<<"拷贝构造"<<std::endl;
    }

    ~MyString() {
        delete[] data;
        std::cout<<"析构函数"<<std::endl;
    }

    MyString& operator=(const MyString &other) {
        std::cout<<"拷贝赋值"<<std::endl;
        if (this != &other) {
            delete[] data;
            int len = strlen(other.data) + 1;
            data = new char[len];
```

```
            strncpy(data, other.data,len);
        }
        return *this;
    }
    //因为需要访问MyString私有成员变量，所以要使用friend
    friend std::ostream& operator << (std::ostream& os, MyString& strObj)
    {
        os<<strObj.data;
        return os;
    }
};

int main()
{
    MyString str1("20240226");
    MyString str2 = str1;
    MyString str3;
    str3 = str1;
    std::cout<<"My string1:"<<str1<<std::endl;
    std::cout<<"My string2:"<<str2<<std::endl;
    std::cout<<"My string3:"<<str3<<std::endl;

    return 0;
}
```

42 什么是动态绑定，C++成员函数 什么是静态绑定?

1. 静态绑定 (Early Binding)：在静态绑定中，编译器在编译阶段就能够准确地确定要调用的函数。这种绑定方式适用于普通的成员函数、静态成员函数和全局函数。对于普通成员函数，即使通过指针或引用进行调用，编译器也会根据类型信息直接绑定到相应的函数地址，因此不需要在运行时再进行额外的查找和计算。

3. 动态绑定 (Late Binding)：动态绑定是针对虚函数的调用而言的，它是在运行时根据对象的实际类型来确定调用的函数。对于虚函数，通过基类指针或引用进行调用时，
编译器并不会直接确定要调用的函数地址，而是等到运行时再根据对象的实际类型来确定要调用的函数地址。

43 当delete一个对象 并设置指针为nullptr后，是否可以访问这个对象的成员函数?

可以通过野指针访问这个对象的成员函数，但不能通过野指针访问这个对象的虚函数。
普通的非虚函数，编译器在编译期间就已经确定了调用的函数地址，所以可以通过野指针访问。
虚函数的调用会被动态绑定 (dynamic binding)，需要在运行时根据对象的实际类型来确定调用的函数地址。
当指针为空时，无法获取到实际对象的信息，因此无法进行正确的动态绑定，这将导致未定义行为或者程序崩溃。

44 虚函数表是针对类还是针对对象的?
虚函数表是针对类而不是对象的，一个类的所有对象共享同一个虚函数表。因此无论一个类有多少个对象实例，它们都会共享相同的虚函数表。

45 纯虚函数和虚函数有什么区别?
虚函数可以有实现，派生类可以选择性地重写虚函数。纯虚函数没有实现，必须在派生类中实现，派生类才能被实例化。
虚函数和纯虚函数都用于实现多态性，允许在运行时根据对象的实际类型来调用适当的函数。

46 构造函数中可以调用虚函数吗?
从语法上讲，调用完全没有问题。但是，往往不能达到多态的效果。
当创建一个派生类对象时，派生类的构造函数会先调用基类的构造函数，在基类的构造函数执行时，此时的this指针指向的是基类对象，通过this->vptr调用的是基类的虚函数。
通常情况下，不建议在构造函数中调用虚函数，尤其是如果这个虚函数被派生类重写并且其行为依赖于派生类中的状态或数据。

47 析构函数中可以调用虚函数吗?
从语法上讲调用没有问题，但是在析构函数中调用虚函数，会存在一些问题。
当派生类对象生命周期结束时，会调用派生类的析构函数，如果在派生类的析构函数中调用虚函数，此时调用的是派生类自己的虚函数。
当进入到基类析构函数时，虚函数不起作用，调用虚函数同调用一般的成员函数一样。

48 什么是虚继承?
虚继承是为了解决多重继承出现菱形继承时出现的问题。例如：类B、C分别继承了类A。类D多重继承类B和C的时候，
类A中的数据就会在类D中存在多份。通过声明继承关系的时候加上virtual关键字可以实现虚继承。

49 说说C++和C相比最大的不同?
1) 类和对象： C++ 是一种面向对象的语言，引入了类和对象的概念，使得程序员可以更方便地组织数据和方法。
2) 函数重载和默认参数： C++ 支持函数重载，允许定义同名函数但参数列表不同的函数。此外，C++ 还支持默认参数，可以为函数参数指定默认值。
3) 异常处理： C++ 引入了异常处理机制，允许程序员编写可以在出现异常时执行特定操作的代码块。
4) 标准模板库 (STL)： 您提到了 STL，它是 C++ 标准库的一部分，提供了丰富的数据结构和算法，如向量、列表、映射、排序算法等，极大地扩展了 C++ 的功能和灵活性。
5) 运算符重载： C++ 允许对运算符进行重载，使得用户可以自定义类的行为，实现更加符合直觉的语义。
6) 类型安全的枚举类： C++ 中引入了枚举类 (enum class)，相比于 C 语言中的枚举类型，枚举类提供了更加严格的类型检查和作用域限制。
7) 内存分配和释放： 在 C++ 中，使用 new 和 delete 关键字进行动态内存的分配

和释放，而不是 C 语言中的 malloc 和 free 函数。
总体而言，C++ 在语言特性和功能上相较于 C 语言更加丰富和复杂，提供了更多的工具和机制来支持面向对象编程、泛型编程、异常处理等。

50 用c++实现一个 基于引用计数的智能指针，需要实现构造，析构，拷贝构造，=操作符重载，重载*-和>操作符。(快手音视频考题)

```
#include <iostream>

template <class T>
class SmartPointer {
private:
    T* ptr;
    int* refCount;

    void relaseCount(){

        (*refCount)--;
        if (*refCount == 0) {
            delete ptr;
            delete refCount;
        }
    }

public:
    SmartPointer() : ptr(nullptr), refCount(nullptr) {
        refCount = new int(0);
        (*refCount)++;
    }

    SmartPointer(T* p) : ptr(p), refCount(new int(1)) {
        if(p)
            *refCount = 1;
        else
            *refCount = 0;
    }

    SmartPointer(const SmartPointer<T>& sp) : ptr(sp.ptr),
refCount(sp.refCount) {
        (*refCount)++;
    }

    ~SmartPointer() {
        relaseCount();
        std::cout <<GetReferenceCout()<< std::endl;
    }
```

```
    SmartPointer<T>& operator=(const SmartPointer<T>& sp) {
        if (this != &sp) {
            relaseCount();

            ptr = sp.ptr;
            refCount = sp.refCount;
            (*refCount)++;
        }

        return *this;
    }

    T& operator*() {

        return *ptr;
    }

    T* operator->() {
        return ptr;
    }

    int GetReferenceCout(){
        return *refCount;
    }
};

int main() {


    SmartPointer<char> cp1(new char('a'));
    SmartPointer<char> cp2(cp1);
    SmartPointer<char> cp3;
    cp3 = cp2;
    cp3 = cp1;
    cp3 = cp3;

    std::cout <<cp1.GetReferenceCout()<< std::endl;
    std::cout <<cp2.GetReferenceCout()<< std::endl;
    std::cout <<cp3.GetReferenceCout()<< std::endl;
    std::cout << "Value pointed by cp1: " << *cp1 << std::endl;
    std::cout << "Value pointed by cp2: " << *cp2 << std::endl;

    std::cout <<"==============="<< std::endl;

    return 0;
}
```

51 c++智能指针的原理：
智能指针是一种 C++ 中的智能内存管理工具，用于管理动态分配的内存，避免内存泄漏和悬空指针等问题。
智能指针的原理主要基于 RAII（资源获取即初始化）和引用计数两种机制。

52 智能指针支持的拷贝构造吗?
std::unique_ptr 是一种独占所有权的智能指针，不支持拷贝构造函数和赋值操作符。
std::unique_ptr 的设计初衷是确保资源的唯一所有权，通过禁止拷贝来避免多个指针共享同一资源的情况。

std::shared_ptr 支持拷贝构造函数和赋值操作符。当使用 std::shared_ptr 进行拷贝构造时，引用计数会增加，
多个 std::shared_ptr 实例可以共享同一块内存资源，并且在最后一个实例被销毁时才释放内存。

如果一块内存被 shared_ptr 和 weak_ptr 同时引用，当所有 shared_ptr 析构了之后，不管还有没有 weak_ptr 引用该内存，内存也会被释放。
所以 weak_ptr 不保证它指向的内存一定是有效的，在使用之前需要检查 weak_ptr 是否为空指针。

53 STL 里 set 和 map 是基于什么实现的？红黑树的特点?

在 C++ STL 中，set 和 map 数据结构都是基于红黑树（Red-Black Tree）实现的。
红黑树是一种自平衡的二叉查找树，它通过在每个节点上添加额外的颜色信息，
并遵循一定的规则来保持树的平衡性。红黑树的特点包括：
1) 节点是红色或者黑色。
2) 根节点是黑色的，所有叶子节点（NIL 节点）都是黑色的。
3) 如果节点是红色的，则其子节点必须是黑色的。
4) 从根节点到叶子节点的每条路径上，黑色节点的数量相同。
5) 没有连续两个红色节点，也就是不存在红色节点的父子关系。
与 AVL 树相比，红黑树的主要区别在于平衡性要求较为宽松，因此在插入和删除操作时，红黑树可能会通过旋转和颜色调整来维持相对平衡，
而不是像 AVL 树那样保持完全平衡。这使得红黑树在实际应用中更加高效，尤其适合于动态数据集合的维护。
在 C++ STL 中，set 使用红黑树来实现有序集合的功能，map 则使用红黑树来实现键-值对的存储和检索。红黑树的平衡性保证了这些容器在插入、
删除和查找操作上的较高效率，并且保持了元素的有序性。因此，红黑树作为 set 和 map 的底层数据结构，为这些容器提供了高效的实现方式。

54 C++ const 的作用是什么?

const 关键字用于声明常量或修饰变量、函数参数、函数返回值等，其作用包括：

1) 声明常量：通过在变量前添加 const 关键字，可以将该变量声明为一个常量，其数值在初始化后不能再被修改。例如：

const int MY_CONSTANT = 10;

2) 修饰变量：const 修饰的变量表示其数值不可被修改，这有助于提高代码的可读性和可维护性。例如：

const int x = 5;

3) 修饰指针：const 可以用来修饰指针，分为指向常量的指针和常量指针两种情况。

指向常量的指针：const int* ptr; // 指向常量的指针，指针指向的值不可修改

常量指针：int x = 5;　int* const ptr = &x; // 常量指针，指针本身不可修改

4) 修饰函数参数：在函数声明或定义中，使用 const 修饰参数表示函数不会修改该参数的值。

这可以避免意外地修改参数值，并使代码更加安全。例如：void func(const int x);

5) 修饰成员函数：在类中，const 修饰成员函数表示该函数不会修改类的成员变量。这样的成员函数称为常量成员函数，例如：

```
class MyClass
{
public:
    void foo() const; // 常量成员函数
};
```

55 C++ 中new与malloc的区别.

1) new 是 C++ 中的运算符，可以调用构造函数为分配的内存进行初始化，并返回所分配类型的指针。

因此，new 是类型安全的，能够确保分配的内存与其类型相匹配。

malloc 是 C 语言中的库函数，它只分配一块指定大小的内存块，并返回 void* 类型的指针。由于 malloc 不会调用构造函数，

所以在 C++ 中使用 malloc 分配内存时，无法自动调用对象的构造函数进行初始化。

2) 使用 new 分配内存时，编译器会根据所需类型的大小自动计算所需的内存空间，因此不需要显式指定分配的大小。

使用 malloc 需要显式指定要分配的内存大小。

3) new 返回所分配类型的指针，不需要进行类型转换。malloc 返回 void* 类型的指针，需要进行类型转换才能使用。

4) new 在分配内存后会调用对象的构造函数进行初始化。malloc 不会调用任何构造函数，只是简单地分配一段内存空间。

56 std::vector的特性

1 底层实现基于动态数组，根据需要动态增长缩小空间。

2 连续存储　内部使用连续的内存存储元素，可以通过索引快速访问元素。

3 随机访问　支持常量时间的随机访问。可以使用下标运算符 [] 或迭代器来访问元素。

4 支持在尾部快速插入和删除元素，但在中间或头部插入或删除元素时，需要移动后续元素。

57 std::list的特性

双向链表结构　std::list 内部使用双向链表结构存储元素

动态大小　std::list 允许根据需要动态增长或缩小

std::list 不支持常量时间的随机访问，因为元素在内存中不是连续存储的.
双向链表结构的特性，std::list 在任意位置进行插入和删除操作的效率都很高.
每个节点都需要额外的指针来指向前一个和后一个节点，存储元素会有一定的额外空间开销。
由于 std::list 的插入和删除操作效率高,适合用于需要频繁进行插入和删除操作的场景
总的来说，std::list 适用于需要频繁插入和删除操作的场景，但不适用于需要随机访问元素的场景。

58 std::map的特性
基于红黑树实现的关联容器。
内部元素按照键的顺序进行排序，因此元素是有序存储的。
查找、插入和删除操作的平均时间复杂度为 O(log n)。
不支持直接访问元素的索引操作，但可以使用迭代器遍历元素。
适用于有序键值对的存储和检索。

59 std::unordered_map的特性
基于哈希表实现的关联容器。
内部元素没有顺序，是无序存储的。
查找、插入和删除操作的平均时间复杂度为 O(1)，但最坏情况下可能会达到 O(n)。
不支持直接访问元素的索引操作，但可以使用迭代器遍历元素。
适用于无序键值对的存储和检索，通常比 std::map 在插入和查找等操作上更快。

60 std::map与std::multimap的比较
相同点：
1) 存储键值对：std::map 和 std::multimap 都用于存储键值对，并提供了对键值对的有序存储和访问能力。
2) 基于红黑树：std::map 和 std::multimap 在内部实现上都使用了红黑树 (Red-Black Tree)，以保持元素的有序性。
3) 迭代器支持：它们都提供了迭代器的支持，可以使用迭代器进行遍历、访问和修改容器中的元素。

不同点：
1) 唯一性：
std::map 中的键是唯一的，每个键只能对应一个值，如果插入具有相同键的元素，则会替换原有键对应的值。
而 std::multimap 允许多个元素具有相同的键，因此可以在 std::multimap 中存储重复的键值对。
2) 插入和查找：
std::map 中的键是唯一的，插入新元素时要进行键的比较和查找，以保持键的唯一性。这会导致插入和查找的时间复杂度是对数级别的。
而std::multimap 允许重复的键，因此插入新元素时只需按照键的顺序插入即可，插入的时间复杂度为常数级别。
3) 迭代器范围：
std::map，每个键只有一个对应的值，因此迭代器范围是唯一的。
std::multimap，由于允许重复的键，因此迭代器范围可以包含多个具有相同键的元素。

综上所述，选择使用 std::map 还是 std::multimap 取决于你的需求和对键的唯一性的要求。
如果需要存储唯一的键值对，可以选择 std::map，如果允许并且需要存储重复的键值对，可以选择 std::multimap。

61 std::set的特性
基于红黑树实现的关联容器。
内部元素按照键的顺序进行排序，因此元素是有序存储的。
查找、插入和删除操作的平均时间复杂度为 O(log n)。
不支持直接访问元素的索引操作，但可以使用迭代器遍历元素。
适用于有序的唯一值的存储和检索。

62 std::unordered_set的特性
基于哈希表实现的关联容器。
内部元素没有顺序，是无序存储的。
查找、插入和删除操作的平均时间复杂度为 O(1)，但最坏情况下可能会达到 O(n)。
不支持直接访问元素的索引操作，但可以使用迭代器遍历元素。
适用于无序的唯一值的存储和检索，通常比 std::set 在插入和查找等操作上更快。

63 std::set与std::multiset的比较
相同：
1) 容器类型：它们都是关联容器，用于存储一组按照特定顺序排列的元素。
2) 元素的访问：它们提供了类似于容器的访问方式，可以使用迭代器进行遍历，还可以直接访问指定位置的元素。
3) 迭代器支持：它们都支持正向迭代器，可以用于遍历元素。
不同：
1) 元素唯一性：
std::set 中每个元素都是唯一的，容器中不允许出现重复的元素。
std::multiset 允许存储多个相同的元素，可以在容器中存储重复的元素。

2) 排序：
std::set 中的元素是按照升序进行排序的，默认使用 < 运算符进行比较。
std::multiset 也对元素进行排序，但与 std::set 不同的是，它允许存储相同的元素，因此可以有多个相同的元素按照顺序存储。

3) 插入操作：
向 std::set 插入重复的元素将被忽略，只有第一个插入的元素会被保留。
向 std::multiset 插入重复的元素会将所有的元素都保留。

4) 查找操作：
std::set 提供的查找函数返回的迭代器指向找到的元素或者尾后迭代器 (如果未找到)。
std::multiset 提供的查找函数返回的迭代器指向第一个匹配的元素。

5) 删除操作：

从 std::set 中删除一个元素将删除所有与之相等的元素。
从 std::multiset 中删除一个元素只会删除一个匹配的元素。

6) 如何选择set 和 multiset
如果需要保持元素的唯一性并进行快速查找，可以选择 std::set。
如果需要允许重复的元素，并按照顺序进行存储和查找，可以选择 std::multiset。

64 如何解决 std::vector 迭代器失效问题?
发生迭代器失效后，在使用之前，重新对迭代器进行赋值即可。让迭代器再次指向我们想要的位置.

65 C++ 类对象为空指针，是否可以访问类成员函数?

空指针可以访问成员函数，但存在以下限制：
不能调用虚函数
调用普通成员函数时，不能访问非静态成员变量
静态成员函数不受影响
在实际编程中，应避免使用空指针调用成员函数，尤其是虚函数和需要访问非静态成员变量的普通成员函数。
这样做可以提高程序的健壮性和可维护性。

66 在 C++ 中，哪些类成员变量必须使用成员初始化列表进行初始化?

const 成员变量: const 成员变量的值在声明后就必须确定，不能在构造函数中进行赋值。
引用成员变量: 引用成员变量必须指向一个有效的对象，也必须在构造函数中进行初始化。否则，编译器会报错。
非默认构造函数的参数类型没有默认构造函数的成员变量: 如果一个类成员变量的类型没有默认构造函数，
那么在构造函数中使用该类型作为参数时，也必须使用成员初始化列表对其进行显式地初始化。

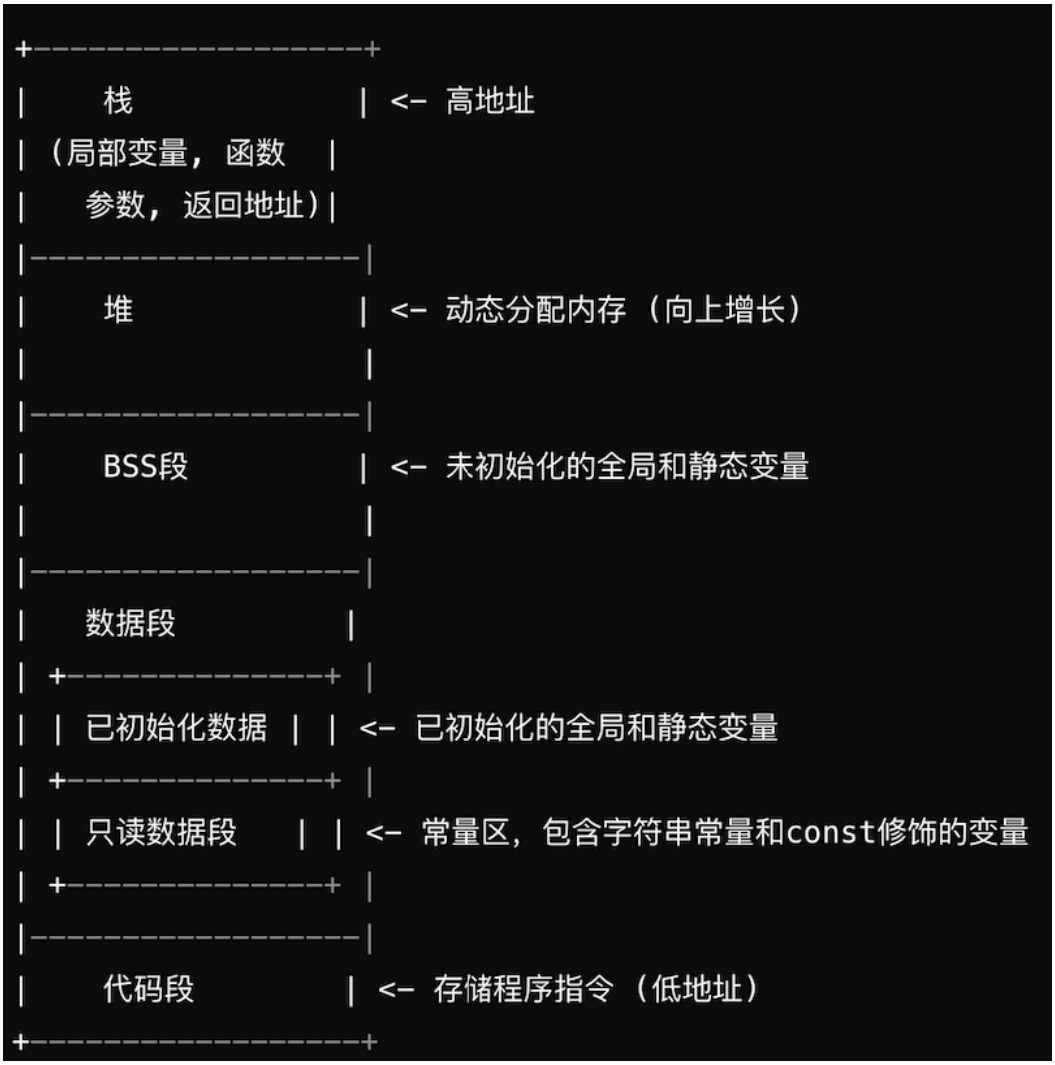
67 C++ 对象的内存分配和程序的内存布局有哪些?

1. 代码段 (Text Segment)
代码段包含程序的机器指令，通常是只读的，以防止意外修改。这部分内存存储了程序的可执行代码。
2. 数据段 (Data Segment)
数据段用于存储全局变量和静态变量。数据段又可以进一步分为以下两个子区域：
　已初始化数据段 (Initialized Data Segment)：存储已初始化的全局变量和静态变量。这些变量在程序开始运行时即被初始化。
　未初始化数据段 (BSS Segment)：存储未初始化的全局变量和静态变量。在程序开始运行时，这些变量会被自动初始化为零。
3. 栈 (Stack)
栈用于存储局部变量、函数参数和返回地址。栈是 LIFO (Last In First Out) 结构，由编译器自动管理，具有以下特点：

栈上的内存分配和释放速度非常快。
局部变量在函数调用时分配，在函数返回时自动释放。
栈空间通常较小，过深的递归或过大的局部数组可能导致栈溢出。

4. 堆（Heap）

堆用于动态分配内存，程序员需要显式地分配和释放堆内存。堆内存的特点包括：
堆内存可以在运行时按需分配，因此灵活性较高。
堆内存的分配和释放速度较慢，需要显式管理（例如使用 new 和 delete）。
不当的内存管理可能导致内存泄漏或悬挂指针。



68 虚函数表指针存储在什么区?

虚函数表指针存储在什么区域看对象在堆上创建还是栈上创建。

69 std::vector 动态扩容机制。

每当需要插入元素且当前容量已满时，std::vector 会进行扩容。通常采用倍增策略，即新容量通常是当前容量的两倍。
这种策略确保扩容的频率较低，从而使插入操作的平均时间复杂度为 O(1)。
当需要扩容时，std::vector 会分配一块新的、更大的内存空间，通常是当前容量的两倍。

提供了 size 和 capacity 两个方法来分别获取当前 vector 的元素数量和当前容量。
当 std::vector 的 push_back 操作导致元素数量超过当前 capacity 时，std::vector 会自动扩容以容纳更多的元素。

扩容过程通常涉及以下几个步骤：
1. 计算新容量： 通常新容量是当前容量的两倍，但具体策略可能因实现而异。
2. 分配新内存： 分配一块能够容纳新容量元素的内存。
3. 复制元素： 将旧内存中的元素复制到新内存中。
4. 释放旧内存： 释放旧的内存空间。
5. 更新指针和容量信息： 更新 vector 的内部指针和容量信息以指向新内存。

70 std::vector 频繁 pushback 会造成什么影响，如何解决

频繁的 push_back 操作可能会对性能产生负面影响，特别是在 std::vector 需要频繁扩容的情况下。
每次扩容时，std::vector 会分配新的内存并将现有元素复制到新位置，会引起性能开销内存碎片问题。

解决频繁扩容问题可以采用
1 预先分配容量：如果知道要插入的大致元素数量，可以使用 reserve 方法预先分配足够的内存，从而减少扩容次数
2 使用 resize 方法：如果知道最终需要的大小，可以一次性调整 vector 的大小，然后使用下标访问元素。这可以避免多次扩容：

71 std::vector emplace_back 和 push_back 的区别

push_back：需要将一个已经构造好的对象插入到 vector 末尾。如果插入的对象是临时对象或需要从其他地方移动或复制，则会有额外的构造和移动/复制开销。
emplace_back：直接在 vector 的末尾原地构造对象，避免了额外的构造和移动/复制开销。

emplace_back 的优点
减少不必要的构造和移动/复制：当对象的构造、复制或移动开销较大时，emplace_back 可以显著提高性能。
更直观：对于复杂对象，可以直接传递构造参数，代码更简洁明了。

emplace_back 不能直接解决频繁扩容的问题
尽管 emplace_back 可以减少对象的构造和移动/复制开销，但它不能直接解决 std::vector 扩容带来的性能问题。
频繁的 emplace_back 操作仍然可能导致 std::vector 频繁扩容。
结合 reserve 使用: 为了减少扩容次数，最好的方法是结合 reserve 使用 emplace_back：

72 c++ 函数指针 回调函数考察: C++ 普通成员函数为啥不能作为回调函数?

回调机制通常要求回调函数是一个 普通的全局函数 或 静态成员函数，具有固定的签名，
不包含 this 指针。因此，普通成员函数不能直接用作回调函数。

函数签名问题： 对于一个c++类的成员函数
```
class MyClass {
 public:
       void memberFunction(int value);
};
```
成员函数 memberFunction，其实际签名如下：

```
void memberFunction(MyClass* this, int value);
```

这意味着它需要一个 MyClass 类型的对象实例来调用。因此，成员函数不能直接用作回调函数，
因为回调函数通常要求的签名是：
typedef void (*Callback)(int value); 这种签名不包含 this 指针。

静态成员函数：由于静态成员函数没有 this 指针，可以直接用作回调函数。
全局函数：全局函数也没有 this 指针，可以直接用作回调函数。
绑定器或 Lambda 表达式：可以使用 std::bind 或 lambda 表达式将普通成员函数包装为符合回调签名的函数。

73 函数返回值是否可以作为重载依据，为什么?

不可以 函数重载是通过参数列表来区分不同的重载函数，而不是通过返回类型。
这是因为在函数调用时，编译器主要通过函数名和参数列表来查找和匹配适当的函数，而返回类型在调用时并不是一个直接的匹配条件。

74 const 修饰成员函数的作用是什么?

在C++中，const修饰成员函数的目的是确保该成员函数不会修改类的成员变量，
也不能调用其他非const成员函数。这样可以提供更高的安全性和代码的可维护性。

75 c++ 如何实现const成员函数修改成员变量?

const 成员函数是不允许修改对象的成员变量的。但是，通过使用 mutable 关键字修饰成员变量，
静态成员变量，属于类,属于类的每个对象共享,存储在静态区，const 成员函数可以访问和修改静态成员变量。

76 #pragma once 和 #ifndef 的区别
#pragma once 和 #ifndef 是两种用于防止头文件重复包含的预处理指令，它们的功能是类似的，但实现方式和一些细节有所不同。
#pragma once 是一种非标准但被大多数现代编译器支持的预处理指令。
#ifndef 是一种标准的预处理指令，适用于所有符合C++标准的编译器。

#pragma once 在某些编译器中可以通过更高效的方式实现，可能比 #ifndef 更快，因为编译器可以更轻松地识别文件是否已被包含。
#ifndef 需要预处理器解析宏定义，这可能稍微增加编译时间，尤其是在大型项目中。
如果目标编译器支持 #pragma once，使用它可以简化代码和减少错误。而如果需要保证代码的最大兼容性，#ifndef 是更稳妥的选择。

77 函数传参什么时候传引用，什么时候传指针?

传引用：
需要修改参数：如果函数需要修改传入的参数值，可以使用传引用。
避免拷贝大对象：传递大的对象时，通过引用传递可以避免对象的拷贝，提升性能。
语法简洁：引用传递语法上比指针更简洁，不需要解引用操作。
常量引用：如果函数不需要修改参数，但仍然希望避免对象的拷贝，可以使用 const 引用。

传指针：
需要修改参数：和传引用一样，可以通过指针修改参数的值。
可能传递空值：如果需要传递一个可能为空的参数，可以使用指针来实现。这在某些情况下非常有用，例如表示可选参数。
动态分配对象：传递动态分配的对象时，可以使用指针。
数组传递：C 语言类型数组在 C++ 中通过指针传递。在现代 C++ 中，更推荐使用标准库容器 (如 std::vector) 来管理数组，而不是使用原始指针。

78 c++栈与堆的区别

栈 (Stack) 特点
自动管理：栈内存由编译器自动管理，程序员不需要手动分配和释放。
生命周期：栈上的变量在其作用域结束时自动销毁。
速度：栈内存分配和释放非常快，因为只需移动栈指针。
大小有限：栈的大小通常有限，适合存储小的、生命周期短的变量。
内存布局：栈内存是连续的，便于快速访问。

堆 (Heap) 特点
手动管理：堆内存由程序员手动管理，使用 new 进行分配，使用 delete 进行释放。
生命周期：堆上的变量在程序员显式释放前不会自动销毁。
速度：堆内存分配和释放相对较慢，因为涉及到复杂的内存管理机制。
大小灵活：堆的大小通常较大，适合存储大的、生命周期较长的变量。
内存布局：堆内存是非连续的，内存碎片化可能导致效率降低。

79 constexpr 的作用
constexpr 是 C++11 引入的一个强大特性，使得可以在编译时求值常量表达式，从而提高程序的性能和安全性。
通过在变量、函数、构造函数和模板中使用 constexpr，可以显著增强编译时常量表达式的能力。

80 constexpr 和 const 区别

使用 const 来声明不可变的值，可以在运行时确定。
使用 constexpr 来声明编译时常量，可以在编译时确定，并确保表达式在编译时求值。

81 c++ this指针的作用

访问对象的成员：this指针用于访问对象的成员变量和成员函数 。返回对象自身

82 c++委托构造函数是什么

委托构造函数（Delegating Constructors）是C++11引入的一项特性，允许一个构造函数调用同一类中的另一个构造函数。
这对于避免重复代码和简化构造函数的实现非常有用。

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

class Person {
private:
    string name;
    int age;
public:
    // 基本构造函数
    Person(const string& name, int age) : name(name), age(age) {
        cout << "Person(const string&, int) called" << endl;
    }

    // 委托构造函数，只初始化name，age默认设为0
    Person(const string& name) : Person(name, 0) {
        cout << "Person(const string&) called" << endl;
    }

};
```

83 std::vector resize reverse的区别是什么
resize函数用于改变std::vector的大小。它可以增加或减少向量的大小，并初始化新元素。

```
std::vector<int> vec = {1, 2, 3};
vec.resize(5); // vec: {1, 2, 3, 0, 0}
vec.resize(7, 10); // vec: {1, 2, 3, 0, 0, 10, 10}
vec.resize(2); // vec: {1, 2}
```

reserve函数用于预留std::vector的存储空间，以避免在向量增长时多次分配内存。它仅影响容量（capacity），不会改变当前元素的数量。不改变大小：reserve只影响

容量，
不改变向量的大小 (size)，即不添加或删除任何元素。

```
std::vector<int> vec = {1, 2, 3};
vec.reserve(10); // 预留空间，但 vec 的大小仍为 3
// vec: {1, 2, 3}, capacity: >= 10
```

84 std::vector std::array 的区别是什么

std::vector 是动态数组，可以动态调整容器大小，而 std::array 是静态数组，其大小在编译时就已经确定，不能改变。
std::vector 依赖动态分配 / 释放内存，这是比较耗时的，而 std::array 的内存大小固定，编译时直接分配一段栈上 / 静态存储区内存即可，速度远快于 std::vector。
std::vector 支持动态扩容，可以在运行时动态增加或减少容器大小，而 std::array 的大小在编译时就已经确定，不能改变。
std::vector 可以使用迭代器访问元素，而 std::array 可以使用下标运算符访问元素。
std::vector 的元素在内存中不一定是连续的，而 std::array 的元素在内存中是连续的。
因此，如果程序需要频繁的创建销毁容器，大小固定 (而且不会爆栈).则选用 std::array 较为合适如果创建销毁不是特别频繁，且容器大小不确定，
可以选用 std::vector。在正确使用下的 std::vector 和 std::array 它们的性能差距在大部分情况下不会构成瓶颈。

85 sizeof(std::string) 在不同的架构 x86 x86_64 大小是多少？

std::string 的大小在不同的体系结构 (x86 和 x86_64) 和不同的编译器实现之间可能有所不同。
x86 架构中，由于指针的大小是 4 字节 (32 位)，因此 std::string 的大小通常为 12 或 16 字节。不同的标准库实现可能略有差异。
在 x86_64 架构中，由于指针的大小是 8 字节 (64 位)，因此 std::string 的大小通常为 24 或 32 字节。

86 C++ 字符串如何转化为数字

使用 std::stoi、std::stof、std::stoll、std::stod

87 C++ 中什么是 RAII?

RAII (Resource Acquisition Is Initialization) 是 C++ 编程中的一种重要惯用法，它通过将资源的获取与对象的生命周期绑定在一起，从而实现资源管理的自动化。
RAII 在许多情况下被用来管理动态内存、文件句柄、网络连接、互斥锁等资源。以下是对 RAII 的详细解释和示例：
RAII 的核心理念：
资源获取即初始化：在对象构造时获取资源，并在对象析构时释放资源。

资源绑定到对象生命周期：资源的生命周期与对象的生命周期绑定，当对象超出作用域或被销毁时，资源也会被自动释放。

88 什么是函数指针？ 什么是指针函数?

函数指针：一个指向函数的指针。用于调用函数或者将函数作为参数传递。
例子：int (*funcPtr)(int, int) = add;
指针函数：一个返回指针的函数。用于返回某种类型的数据指针。
例子：int* getArray() { ... }

89 构造函数，析构函数中是否可以抛出异常?

构造函数：构造函数中尽量不要抛出异常，能避免的就避免，如果必须，要考虑不要内存泄露!
析构函数：不应该抛出异常。

90 C++内联函数是如何实现的，优缺点是啥

内联函数可以通过在函数定义前添加 inline 关键字来声明
内联函数适用于频繁调用的小函数 缺点是 可是它的展开视编译器而定，当代码太长的时候展开内联函数反而会浪费时间
内联函数的优点：减少函数调用开销，有利于优化，有助于小函数
内联函数的缺点：代码膨胀，编译时间增加，调试困难，不适合递归：

91 常量指针和指针常量的区别是?

常量指针 (Pointer to Constant)：
定义：const int* ptr; 或 int const* ptr;
含义：指向常量数据的指针，通过该指针不能修改所指向的数据，但可以修改指针本身。
适用场景：当需要通过指针访问数据但不希望数据被修改时使用。

指针常量 (Constant Pointer)：
定义：int* const ptr;
含义：指针本身是常量，一旦初始化，不能改变其指向的地址，但可以通过该指针修改所指向的数据。
适用场景：当需要一个固定指向特定数据的指针，并且希望通过该指针修改数据时使用。

92 C++深拷贝 浅拷贝的区别

浅拷贝是对象的逐位复制。复制出的新对象中的成员变量 (包括指针) 都指向原对象中相同的内存位置。
这意味着如果两个对象共享动态分配的资源 (如堆内存)，对一个对象的修改可能会影响到另一个对象。

深拷贝是对象的完整复制，包括其指向的动态分配的内存。这意味着每个对象拥有自己独立的内存副本，
对一个对象的修改不会影响到另一个对象。

93 什么是左值?

左值 (lvalue)：有名字并且可以取地址的对象。例如，变量 a 是一个左值，可以通过它的名字访问或修改它的值。
int a = 10;

94 什么是右值?
右值引用 (rvalue reference) 是C++11引入的一种引用类型，用于高效地操作临时对象和实现移动语义。
右值引用使得可以“窃取”或“移动”资源，而不是进行昂贵的拷贝操作，
从而提升程序的性能。右值引用使用 && 符号来表示。

右值 (rvalue)：没有名字、临时的值。右值通常是表达式求值后得到的结果，例如字面量 10 或表达式 a + b。
int b = a + 5; // a + 5 是一个右值

95 一个 unique_ptr 怎么赋值给另一个 unique_ptr 对象

一个std::unique_ptr不能直接赋值给另一个std::unique_ptr，因为std::unique_ptr拥有的指针是独占的。
如果你想要转移所有权，你应该使用std::move来转移所有权。
这样做可以让原始指针的所有权转移到新的unique_ptr，而原始unique_ptr将被置为空。

```
// 创建一个 unique_ptr 管理一个动态分配的整数
std::unique_ptr<int> ptr1 = std::make_unique<int>(42);

// 输出 ptr1 所管理的对象的值
std::cout << "ptr1 value: " << *ptr1 << std::endl;

// 将 ptr1 的所有权转移给 ptr2
std::unique_ptr<int> ptr2 = std::move(ptr1);

 // 此时 ptr1 不再管理任何对象
 if (ptr1 == nullptr) {
        std::cout << "ptr1 is null" << std::endl;
}

 // ptr2 管理原来由 ptr1 管理的对象
std::cout << "ptr2 value: " << *ptr2 << std::endl;
```

96 什么是虚拟内存

虚拟内存 (Virtual Memory) 是计算机系统的一项重要技术，
它允许计算机使用硬盘 (或其他存储设备) 来扩展其物理内存 (RAM)，
从而为运行程序提供更大的地址空间。虚拟内存技术的核心思想是通过将物理内存和硬盘结合起来，
创建一个统一的虚拟地址空间，使得程序可以访问比实际物理内存大得多的内存。

97 面向对象三大特性：

面向对象的三大特性是：封装 (Encapsulation)、继承 (Inheritance) 和多态 (Polymorphism)。
封装：
数据保护：通过访问控制 (如私有、保护和公共访问修饰符)，可以保护对象的数据不被随意修改。
模块化：封装使得每个类成为一个独立的模块，代码更易于理解和维护。
隐藏复杂性：对外部隐藏实现细节，简化接口，减少外部依赖。

继承：
代码重用：子类继承父类的属性和方法，减少重复代码。
扩展性：子类可以在父类的基础上添加新的属性和方法，便于功能扩展。
层次结构：通过继承，可以建立类的层次结构，反映现实世界的分类关系。

多态度
接口统一：通过统一的接口，调用不同的实现，简化代码。
灵活性：可以在运行时选择合适的实现，提高代码的灵活性和可扩展性。
可维护性：通过多态，可以减少条件语句，使代码更易维护。

98 c++ lambda 表达式 (匿名函数) 的具体应用和使用场景

C++ 中的 lambda 表达式 (匿名函数) 是在 C++11 标准中引入的，它们可以在需要定义一个临时函数对象的地方使用，
提供了一种更简洁的语法来编写内联函数。lambda 表达式在 C++ 中提供了一种简洁的方式来定义临时函数对象，
尤其适用于 STL 算法、事件处理、回调函数、并发编程和自定义比较器等场景。通过使用 lambda 表达式，代码变得更加简洁和可读。

99 C 和 C++ static 的区别

在 C 中，static 关键字有两种主要用途：
静态全局变量和函数：
文件作用域：如果在文件顶部 (即全局范围内) 声明一个变量或函数，并使用 static 关键字，那么这个变量或函数的作用域将被限制在该文件内。
这意味着其他文件不能访问该变量或函数。

静态局部变量：

块作用域：如果在函数内声明一个变量并使用 static 关键字，这个变量的存储持续时间将是整个程序运行期间 (全局的)，但它的作用域仅限于该函数。
这意味着该变量在函数调用之间保持其值不变。

```
void func() {
    static int x = 0;
    x++;
    printf("%d\n", x);
}

int main() {
    func(); // 输出 1
    func(); // 输出 2
    func(); // 输出 3
    return 0;
}
```

在 C++ 中，static 关键字有类似的用途，但也有一些扩展和增强的功能：

静态全局变量和函数：与 C 中的用法相同
静态局部变量：与 C 中的用法相同

静态成员变量和成员函数：
静态成员变量：在类中，静态成员变量属于类本身，而不是类的实例。所有实例共享同一个静态成员变量。
静态成员变量需要在类定义外进行初始化。
静态成员函数：静态成员函数属于类本身，而不是类的实例。它们只能访问静态成员变量和静态成员函数，
不能访问非静态成员变量和非静态成员函数。

100 inline 函数工作原理

inline 函数是一种提示编译器在调用该函数时将其代码内联到调用点处的函数，以避免函数调用的开销。
它的主要目的是通过减少函数调用的开销 (如参数传递、返回地址保存等) 来提高程序的运行效率。
尽管它只是一个建议，编译器可以根据具体情况决定是否内联化函数。

inline 函数的工作原理
当一个函数被标记为 inline 后，编译器在处理该函数时，会尝试将函数的代码直接嵌入到每个调用点。
这意味着，调用 inline 函数时，不需要执行标准的函数调用流程 (如跳转到函数地址，执行函数代码，返回原地址)，
而是直接在调用点展开函数体。

101 宏定义 (define) 和内联函数 (inline) 的区别

宏定义只是编译前文本替换，无类型检查
宏定义是预处理器指令，在编译前由预处理器进行文本替换。宏可以用来定义常量、简化代码片段以及定义函数式的宏。
文本替换：宏是纯粹的文本替换，不进行类型检查。
无类型检查：由于宏在预处理阶段展开，没有类型检查，可能会导致类型相关的错误。
调试困难：宏的展开发生在编译前，调试时无法直接看到宏的展开情况，增加了调试的难度。
作用域：宏没有作用域的概念，一旦定义，在整个编译单元内有效，可能导致命名冲突。

内联函数 (inline) 减少函数调用开销，编译期间处理 类型检查
内联函数是编译器的一个优化建议，提示编译器在函数调用处直接展开函数体，以减少函数调用的开销。
内联函数在编译阶段处理，具有函数的所有特性，包括类型检查和作用域。
类型检查：内联函数在编译阶段进行类型检查，确保类型安全。
作用域和命名空间：内联函数遵循 C++ 的作用域和命名空间规则，减少命名冲突的风险。
调试友好：内联函数在调试时可以看到函数的调用和执行情况，调试更加友好。
编译器优化：inline 只是对编译器的建议，编译器会根据具体情况决定是否展开函数体。如果函数体较大或复杂，编译器可能不会内联展开。

102 class 和 struct 的异同

struct 的成员默认是 public 的， struct 的继承默认是 public 的
class 的成员默认是 private 的，class 的继承默认是 private 的

struct 常用于表示纯粹的数据结构，不包含复杂的业务逻辑。它通常用于表示数据的聚合，而没有太多的方法。
class 通常用于表示具有封装、继承和多态等特性的复杂对象。它不仅仅包含数据，还包含对数据的操作方法和实现细节。

103 struct 和 union 的区别

struct 中的每个成员都有自己的内存位置。成员按声明的顺序依次存储，每个成员都有独立的存储空间。
结构体的大小是所有成员大小之和，加上可能的填充字节 (用于对齐)。

union 中的所有成员共享同一块内存。也就是说，union 中的所有成员的起始地址相同，内存大小为最大成员的大小。
联合体中的所有成员共享同一块内存，因此同一时间只能访问一个成员，访问其他成员会导致数据覆盖。
联合体的大小是其最大成员的大小，而不是所有成员大小的总和。

104 什么是模板特化?为什么特化?

当你需要为特定类型提供特定的实现时，你可以使用模板特化。模板特化分为两种类型：完全特化和偏特化。

105 泛型编程如何实现?

泛型编程 (Generic Programming) 是一种编程范式，旨在通过抽象化来编写与类型无关的代码，从而提高代码的复用性和灵活性。
在 C++ 中，泛型编程主要通过模板机制来实现。通过合理使用模板、类型萃取、模板特化和模板元编程，你可以编写高效、可维护和通用的代码。

106 虚函数 纯虚函数的区别

虚函数可以有实现，也可以在派生类中重写。
纯虚函数在基类中没有实现，必须在派生类中实现。

包含虚函数的类可以实例化。
包含纯虚函数的类是抽象类，不能实例化。

虚函数用于允许或强制派生类提供自己的实现，同时允许基类提供默认实现。
纯虚函数用于定义接口，使得基类不能实例化，只能作为接口被派生类实现。

虚函数使用关键字 virtual。
纯虚函数使用 virtual 并将函数设置为= 0。