**1．C++的构造函数，复制构造函数，和析构函数。什么是深复制和浅复制，构造函数和析构函数哪个能写成虚函数，为什么。**

拷贝构造函数，即复制构造函数，是将同一个类型的对象作为形参进行传递，默认的拷贝构造函数会产生浅拷贝，复制出的对象的值会与原本的对象完全相同，此时如果对象中不具有指针等类型将不会发生问题，但是如果具有指针，新对象的值和旧对象的指针将指向同一块内存区域，此时析构函数释放内存时两者的析构函数会释放同一片内存地址两次导致出现错误；所以在此情况下需要自定义拷贝构造函数，来实现不同的指针指向不同的地址吗，称为深拷贝。注意拷贝构造函数的参数必须是&引用，不然会出现无穷递归。

即使不进行构造函数和析构函数的创建，系统也会自动生成构造函数和清理内存用的析构函数。

系统默认的构造函数进行值的初始化，默认构造函数会根据类的成员进行生成，系统自带数据类型为随机值，自定义数据类型会去调用其构造函数。

构造函数可以重载，析构函数不可以。

析构函数本身不会释放空间，类在生命周期结束时只能释放在栈上分配的空间，fopen打开的文件以及new出来的在堆上的自行管理的空间等内容还是需要自身编写析构函数去释放。

构造函数不可以写为虚函数，构造对象时就必须知道其类型，而虚函数需要在运行过程中确定其类型，将构造函数声明为虚函数时，在构造函数执行时虚指针还没有被正确的初始化，仍然指向基类的虚函数表，所以无法正确执行子类的虚函数。这一段是网上查找到的答案。

我查看了很多资料，众说纷纭，我就开始思考为什么没有一个官方的统一的说法，最后在C++Primer上发现了原因，C++的底层从逻辑角度就已经不支持构造函数为虚函数，所以网络上为什么不能将虚函数声明为构造函数大家都是从“如果将构造函数声明为虚函数会发生什么”出发的猜测。

虚函数表在编译时生成在只读段，是类级别的，与对象的分配空间无关。

VPTR在基类的初始化在子类中被覆盖。

析构函数可以写为虚函数，以便实现适应子类对象内存的释放。

[https://blog.csdn.net/weixin\_69283129/article/details/127775958?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522a07092277ca367de0490e8a001b38473%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=a07092277ca367de0490e8a001b38473&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-127775958-null-null.142^v101^pc\_search\_result\_base6&utm\_term=%E6%8B%B7%E8%B4%9D%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&spm=1018.2226.3001.4187](https://blog.csdn.net/weixin_69283129/article/details/127775958?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522a07092277ca367de0490e8a001b38473%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=a07092277ca367de0490e8a001b38473&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-127775958-null-null.142%5ev101%5epc_search_result_base6&utm_term=%E6%8B%B7%E8%B4%9D%E6%9E%84%E9%80%A0%E5%87%BD%E6%95%B0&spm=1018.2226.3001.4187)

**2．C++数组，链表，二叉树的内存排列是什么样的，结构体占多大内存如何计算，类占用多大空间如何计算，空类的空间是多少，为什么。**

数组元素在内存中的分布有列优先和行优先两种，以行优先为例，一行内容在内存中按顺序排布，下一行接在这行内容后面。

链表是一个结构体，内部保存其指向的下一个节点的信息。

二叉树有两种内存结构，使用数组实现的顺序存储结构和使用链表实现的链式存储结构，

存储结构使用了特殊的计算关系来定义父节点与子节点的下标关系节点的父节点为（i-1

)/2，左子结点为2i+1。

链式存储结构则是在链表中定义左右子节点的指针。

类占用空间和结构体占用空间都需要进行内存对齐，由于从内存中取数据是按照块为单

行的，不进行对齐可能会导致一个数据元素需要取两次才能取出。

内存对齐的规则：1.所有元素的偏移量均为自身大小整数倍2.类的大小必须为对齐边界（

内存对齐指令决定，一般是最大数据元素）的整数倍不足则在最后补空间3.对于数组元素

其大小为其对应的数据类型的大小而非所有元素的实际大小。

空类大小为1字节，占位使用避免出现地址重叠。

<https://blog.csdn.net/luolaihua2018/article/details/115372273>

**3. 虚函数和虚表的原理是什么**

在 C++ 中，虚函数的实现原理基于两个关键概念： 虚函数表和虚函数指针。

虚函数表：每个包含虚函数的类都会生成一个虚函数表（Virtual Table），其中存储着该类中所有虚函数的地址。虚函数表是一个由指针构成的数组，每个指针指向一个虚函数的实现代码。

虚函数指针：在对象的内存布局中，编译器会添加一个额外的指针，称为虚函数指针或虚表指针（Virtual Table Pointer，简称 VTable 指针）。这个指针指向该对象对应的虚函数表，从而让程序能够动态地调用正确的虚函数。

<https://zhuanlan.zhihu.com/p/629281871>

**4. 内存泄漏出现的时机和原因，如何避免**

内存泄露的原因当我们在程序中对原始指针(raw pointer)使用new操作符或者malloc函数的时候，实际上是在堆上为其分配内存，这个内存指的是RAM，而不是硬盘等永久存储。持续申请而不释放(或者少量释放)内存的应用程序，最终因内存耗尽导致OOM(out of memory)。

避免的方法：

在栈上分配空间，栈是使用类似与数据结构中出栈和入栈的方式由编译器进行管理，不会出现内存泄露，栈在内存空间上是连续的，不需要额外的开销来寻找可用的内存空间。

使用智能指针而非裸指针。

记得每次分配后进行释放。

使用STL中的数据结构而非C自带的结构，STL中的数据结构自身便带有相当优秀的内存管理。

使用RAII，也就是智能指针底层的实现思想，将资源的初始化放入对象中， 将资源的生命与对象的生命绑定，利用对象的析构函数对资源进行释放。

<https://blog.csdn.net/weixin_46645965/article/details/137088374>

**5. 指针的工作原理，函数的传值和传址。**

指针是一个独立的数据类型,不依托于其他的数据类型,所有指针,在程序位元（X64，X32）相同的情况下,占用的字节,永远是一样的，在X64的系统中为8字节，在X32的系统中为4字节.作为指针类型的变量,它只接受内存地址，本质是一个整数类型的变量。

指针本身的值是一个内存地址

指针本身被存放在一个内存空间中，其本身也有地址

\*运算符是去找指针指向的内存。

编译器会对指针的类型和权限进检查，只有符合的数据类型可以赋值。

Const char\*与char\* 不能互相赋值，const char\*指向的空间只能读，char\*可读可写，char\*的权限更高所以char\*可以赋值给const char\*而不能反过来。

[https://blog.csdn.net/qq\_42468226/article/details/117090516?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522d9f3cf4e81eda80462d66c84b1aef53d%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=d9f3cf4e81eda80462d66c84b1aef53d&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~baidu\_landing\_v2~default-1-117090516-null-null.142^v101^pc\_search\_result\_base6&utm\_term=C%2B%2B%E6%8C%87%E9%92%88%E7%9A%84%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86&spm=1018.2226.3001.4187](https://blog.csdn.net/qq_42468226/article/details/117090516?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522d9f3cf4e81eda80462d66c84b1aef53d%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=d9f3cf4e81eda80462d66c84b1aef53d&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-1-117090516-null-null.142%5ev101%5epc_search_result_base6&utm_term=C%2B%2B%E6%8C%87%E9%92%88%E7%9A%84%E5%B7%A5%E4%BD%9C%E5%8E%9F%E7%90%86&spm=1018.2226.3001.4187)

值传递是进行的复制传值，在传递的东西的数据类型是对象的时候由于创建了一个新的对象此时会执行类型的构造函数。

**6. new和delete使用解释一下，和malloc和free的区别**

New和delete后面直接加上数据类型来实现分配内存空间，如果要分配连续的空间则加上[],eg:test\* a=new test[5]分配五个连续的test大小的空间，new[]与delete[]对应使用，delete[] a，来删除五个test的空间。

New和delete本质是运算符，即operator new（），其底层操作的仍然是malloc和free，只是加上了其他功能的封装，malloc和free才是与os进行交互的部分，malloc和free是函数。

New和delete在创建和销毁内存空间时会去调用对象的构造函数和析构函数，而malloc和free不会，他们只是单纯的分配空间并返回一个指针，所以当使用malloc分配空间后，如果空间，直接使用free指针会导致内存的泄露。

New返回的指针直接就是对应类型，而malloc返回的是一个万能指针void\*,需要强制类型转换(C的转换模式即可)后才使用。

void\* raw = malloc(sizeof(Test)); // 分配原始的字节块，但并没有调用构造函数

Test\* a = new (raw) Test(); // 在 raw 所指向的内存上调用 Test 的构造函数

Malloc分配类的方法，raw是分配的原始空间的指针（），则test\* a=new（raw）test来进行使用malloc预先分配空间后的分配方式

Malloc无法分配空间时返回null，而new会报错。

<https://blog.csdn.net/ZWW_zhangww/article/details/142364552>

[C/C++中内存管理\_c++ malloc-CSDN博客](https://blog.csdn.net/ZWW_zhangww/article/details/142364552)

**7.** **C++内存区域划如何分说一下（栈，堆那些）**

static关键字：

static关键字的作用是将其修饰的内容存储地址修改为静态区，只会被初始化一次，再次遇到初始化语句会直接跳过。

静态区的所有变量都具有会被默认初始化为0的性质。

satic修饰局部变量的话其生命周期修改为整个文件的生命周期，但是作用域不会发生改变。相当于只是修改了存储的位置。

static修饰的内容无法跨文件进行引用，会将所修饰内容的链接属性修改为内部链接，本来全局变量/函数的链接属性应该是外部链接的，基于此特性，可以将全局变量声明为此属性以实现对其他文件的隐藏。

在引入面向对象的特性（C++）后，会带来基于类的新的性质。

静态成员为该类的所有对象所共用，所有对象操作的是同一个成员。

静态成员在类外进行定义（分配内存）

class A

{

private:

//声明

static int \_n;

static int \_k;

};

//定义

int A::\_n = 0;

int A::\_k = 0;

这些特性都是存储区带来的。

静态的成员函数由于没有默认设置的this指针所以不能访问除了静态成员变量之外的变量。

访问静态变量的方法：通过对象进行访问a.F，通过类进行访问A::F，通过匿名对象进行访问A().F，匿名对象相当于实例化了一个没有名字的对象，跟匿名函数相似都是简便使用。当将静态对象设置为static时，可以通过定义静态方法来进行访问该静态变量，静态方法的访问与静态变量相同。

[https://blog.csdn.net/weixin\_45031801/article/details/134215425?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522883b2657c698e28249708ae2aadabee8%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\_id=883b2657c698e28249708ae2aadabee8&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-1-134215425-null-null.142^v101^pc\_search\_result\_base6&utm\_term=C%2B%2BStatic%E3%80%81&spm=1018.2226.3001.4187](https://blog.csdn.net/weixin_45031801/article/details/134215425?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522883b2657c698e28249708ae2aadabee8%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request_id=883b2657c698e28249708ae2aadabee8&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~first_rank_ecpm_v1~rank_v31_ecpm-1-134215425-null-null.142%5ev101%5epc_search_result_base6&utm_term=C%2B%2BStatic%E3%80%81&spm=1018.2226.3001.4187)

内存划分：

在C++中，内存区域被分配为栈，堆，静态存储区和代码区，栈由计算机自动管理分配空间，不会出现内存泄露，堆是由程序员主动进行分配和回收，使用malloc和new进行分配，free和delete进行释放，会出现内存泄露，栈的数据读取速度快但是空间小，堆则相反，静态存储区中存放全局变量和静态变量以及常量（包括字符串常量“aaa”之类的）。代码区存放将原本代码转化的二进制语言

<https://baike.baidu.com/item/C%2FC%2B%2B%E5%86%85%E5%AD%98%E5%88%86%E5%8C%BA/18764208>

**8. 智能指针用过吗，有哪些，他们的区别和各自的优缺点**

auto\_ptr 使用此指针分配的空间将会在生命周期结束时自动释放。

用法：

Include <memory>

// 定义智能指针

auto\_ptr<Test> test(new Test);

//像普通指针一样使用

cout << "test->debug：" << test->getDebug() << endl;

cout << "(\*test).debug：" << (\*test).getDebug() << endl;

其底层的原理是与前面提到的将空间分配放入对象内部，利用对象自身的析构函数释放空间，其定义了管理指针的对象，可以将new 获得（直接或间接）的地址赋给这种对象。

该对象自身带有一些方法。

1. // 定义智能指针

auto\_ptr<Test> test(new Test);

Test \*tmp = test.get(); // 获取指针所托管的地址

cout << "tmp->debug：" << tmp->getDebug() << endl;

2. // 定义智能指针

auto\_ptr<Test> test(new Test);

Test \*tmp2 = test.release(); // 取消智能指针对动态内存的托管并将该地址进行返回给一个普通指针，注意此时并没有释放对应的内存空间，

delete tmp2; // 之前分配的内存需要自己手动释放

3. // 定义智能指针

auto\_ptr<Test> test(new Test);

test.reset(); // 释放掉智能指针托管的指针内存，并将其置NULL

test.reset(new Test()); // 释放掉智能指针托管的指针内存，并将参数指针取代之

reset函数会将参数的指针(不指定则为NULL)，与托管的指针比较，如果地址不一致，那么就会析构掉原来托管的指针，然后使用参数的指针替代之。然后智能指针就会托管参数的那个指针了。

现在auto\_ptr已经被放弃使用了因为其存在致命的缺陷

1.复制或者赋值会修改原本内存地址的所有权，使用的拷贝但是实际上实现的却不是拷贝的功能而是剥夺。

// auto\_ptr 被C++11抛弃的主要原因

auto\_ptr<string> p1(new string("I'm Li Ming!"));

auto\_ptr<string> p2(new string("I'm age 22."));

cout << "p1：" << p1.get() << endl;

cout << "p2：" << p2.get() << endl;

// p2赋值给p1后，首先p1会先将自己原先托管的指针释放掉，然后接收托管p2所托管的指针，

// 然后p2所托管的指针制NULL，也就是p1托管了p2托管的指针，而p2放弃了托管。

p1 = p2;

cout << "p1 = p2 赋值后：" << endl;

cout << "p1：" << p1.get() << endl;

cout << "p2：" << p2.get() << endl;

2.由于该机制的存在，在STL的容器vector中使用会有巨大风险，因为vector中会将该拷贝当做常规拷贝使用。

3.无法进行数组的内存管理

auto\_ptr在c++11中已经被弃用。

在C++11中取代auto\_ptr的内容是unique\_ptr，unique\_ptr相较于auto\_ptr只有三个区别，相关的函数及定义是相同的。

1. unique\_ptr仅允许右值赋值和右值拷贝，所有左值都要使用std::move转为右值后才能进行赋值或者拷贝，避免了原本的隐式所有权转移的缺陷。
2. 在vector中使用也是需要使用move来让程序员知道这样做的后果。
3. 允许对数组进行内存管理，会自动释放数组的完整内存。

对于上面auto\_ptr和unique\_ptr都无法解决的指针共享的问题，在C++11中引入了新的智能指针shared\_ptr,该指针允许多个托管同一个地址也就是说可以可以左值赋值和左值拷贝，使用的函数之类的内容与上面的unique\_ptr相同。

其维护了一个值来显示有多少个shared\_ptr引用了该指针，在一个shared\_ptr取消托管后该值减一，增加一个shared\_ptr托管后该值加一。

shared\_ptrr<int> up1(new int(10));

up1 = nullptr ; // int(10) 的引用计数减1,计数归零内存释放

// 或

up1 = NULL; // 作用同上

除了常规的初始化外，shared\_ptr还有自己的效率更高的初始化方法

shared\_ptr<int> up3 = make\_shared<int>(2); // 多个参数以逗号','隔开，最多接受十个

shared\_ptr<string> up4 = make\_shared<string>("字符串");

shared\_ptr<Person> up5 = make\_shared<Person>(9);

注意类互相使用shared\_ptr引用对方的对象会导致出现循环，出现内存无法被释放的现象。

使用shared\_ptr托管的对象的生命周期不再与作用域关联而是只与shared\_ptr的引用计数相关，只有所有的shared\_ptr被销毁时对象才会被销毁。

weak\_ptr 设计的目的是为配合 shared\_ptr 而引入的一种智能指针来协助 shared\_ptr 工作, 它只可以从一个 shared\_ptr 或另一个 weak\_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少。 同时weak\_ptr 没有重载\*和->，也就无法通过常规的方法获取指针指向对象的数据，但可以使用 lock 获得一个可用的 shared\_ptr 对象。同样可以使用.use\_count来获取被托管的共享指针数量。

shared\_ptr<Boy> spBoy(new Boy());

shared\_ptr<Girl> spGirl(new Girl());

// 弱指针的使用

weak\_ptr<Girl> wpGirl\_1; // 定义空的弱指针

weak\_ptr<Girl> wpGirl\_2(spGirl); // 使用共享指针构造

wpGirl\_1 = spGirl; // 允许共享指针赋值给弱指针

// 在必要的使用可以转换成共享指针

shared\_ptr<Girl> sp\_girl;

sp\_girl = wpGirl\_1.lock();

将原本在类中相互引用的共享指针替换为weak\_ptr，在需要使用时再.lock进行转化则可以解决循环的问题

Weak\_ptr具有的expired函数可以得到当前weak\_ptr所代表的shared\_ptr指针use\_count是否为0，

[C++ 智能指针 - 全部用法详解-CSDN博客](https://blog.csdn.net/cpp_learner/article/details/118912592?ops_request_misc=%257B%2522request%255Fid%2522%253A%25228fb3a511c05f24323568f4dfa1cb6cec%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=8fb3a511c05f24323568f4dfa1cb6cec&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-118912592-null-null.142%5ev101%5epc_search_result_base6&utm_term=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88&spm=1018.2226.3001.4187)

[https://blog.csdn.net/cpp\_learner/article/details/118912592?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%25228fb3a511c05f24323568f4dfa1cb6cec%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=8fb3a511c05f24323568f4dfa1cb6cec&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-118912592-null-null.142^v101^pc\_search\_result\_base6&utm\_term=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88&spm=1018.2226.3001.4187](https://blog.csdn.net/cpp_learner/article/details/118912592?ops_request_misc=%257B%2522request%255Fid%2522%253A%25228fb3a511c05f24323568f4dfa1cb6cec%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=8fb3a511c05f24323568f4dfa1cb6cec&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-118912592-null-null.142%5ev101%5epc_search_result_base6&utm_term=%E6%99%BA%E8%83%BD%E6%8C%87%E9%92%88&spm=1018.2226.3001.4187)

[c++中引用和指针的详细区分\_c++ 引用和指针的区别-CSDN博客](https://blog.csdn.net/M3255094930/article/details/142959785)

<https://blog.csdn.net/M3255094930/article/details/142959785>

[一文读懂C++右值引用和std::move - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/335994370)

https://zhuanlan.zhihu.com/p/335994370

**9.auto关键字知道吗，如果全部都用auto声明变量行不行**

#include <iostream>

int main() {

int a = 10;

int& ref = a;

int\* ptr = &a;

auto x = ref; // x被推导为int，而不是int&（因为会拷贝ref的值）

auto& y = ref; // y被推导为int&，保持了引用特性

Auto\* p = ptr; // p被推导为int\*，保持了指针特性

y = 20; // y是引用，改变y会改变a

std::cout << a << " " << \*p << std::endl; // 输出 20 20

return 0;

}

解释：auto会自动推导为变量的实际类型，若需要保持引用或指针的功能，必须显式使用&或\*。

当在同一行声明多个变量时，这些变量必须是相同的类型，否则编译器将会报错，因为编译器实际只对第一个类型进行推导，然后用推导出来的类型定义其他变量。

错误示例：

void TestAuto()

{

auto a = 1, b = 2;

auto c = 3, d = 4.0; // 该行代码会编译失败，因为c和d的初始化表达式类型不同

}

在C++11引入了由程序自身判定范围的遍历for

void TestFor()

{

int array[] = { 1, 2, 3, 4, 5 };

for (auto& e : array)// 相当于foreach (var a in list)

{

e \*= 2;

}

for (auto e : array)

{

cout << e << " ";

}

}

Auto的合理使用可以使代码更加简洁，但是对于auto的滥用会导致会导致程序的可读性下降，无法在变量定义的位置直接看到变量的具体类型需要通过上下文进行查看。

Auto不能在没有初始化的情况下使用，也不能在类的成员中使用。

[【C++】关键字auto详解\_c++ auto-CSDN博客](https://blog.csdn.net/Huangcancan666/article/details/142318346)

<https://blog.csdn.net/Huangcancan666/article/details/142318346>

**10. lambda表达式会用吗**

auto fun = [a,b] (int c,int b) mutable throw() ->int {函数体}

fun(a,b)

[]:是捕获列表，a，b是直接上文的变量。

():参数列表，该lambda所需要的参数，就相当于普通函数的参数列表

mutable:所有lambda表达式都是默认为const的，加上此参数后会取消其常量判定

throw():显式声明该表达式不会抛出异常只是语法上的告诉其他人不需要进行异常检查

->int:返回值的类型，如果返回值的类型是确定的那么可以省略，由编译器自身推导。

一个仅包含必须部分的最简单的lambda表达式：[]{}

[C++ Lambda表达式详解\_c++ lamda-CSDN博客](https://blog.csdn.net/qq_37085158/article/details/124626913?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522476511bf4571adde486aae5041f25cdc%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=476511bf4571adde486aae5041f25cdc&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-124626913-null-null.142%5ev101%5epc_search_result_base6&utm_term=c%2B%2Blambda%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1018.2226.3001.4187)

[https://blog.csdn.net/qq\_37085158/article/details/124626913?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522476511bf4571adde486aae5041f25cdc%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request\_id=476511bf4571adde486aae5041f25cdc&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~top\_positive~default-1-124626913-null-null.142^v101^pc\_search\_result\_base6&utm\_term=c%2B%2Blambda%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1018.2226.3001.4187](https://blog.csdn.net/qq_37085158/article/details/124626913?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522476511bf4571adde486aae5041f25cdc%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=476511bf4571adde486aae5041f25cdc&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-124626913-null-null.142%5ev101%5epc_search_result_base6&utm_term=c%2B%2Blambda%E8%A1%A8%E8%BE%BE%E5%BC%8F&spm=1018.2226.3001.4187)

**11.** **override关键字必须吗**

Override关键字作用于显式声明派生类的函数是重载至父类，不加也可以，都是一样的重载效果。

但是在不加的情况下，只有派生类的参数列表和返回值与父类的函数一样才会被视为重载，否则会被视为两个独立的函数。

而在增加了override关键字后会被视为重载函数，其如果没有在父类中找到对应的虚函数会报错。

Override关键字只是语法层面的规范，并不是功能的限制，只是用于避免父类函数修改后而派生类忘了修改的类似的错误。

虚函数子类和基类之间是函数的重写而不是函数的重载，要求函数的参数列表以及函数名完全相同。

**12.右值引用说一下，可能会实际应用**

右值引用:int&&

左值引用:int&

左值引用相当于是对资源取别名，最终与原本的变量指向的是同一个资源，而右值引用是剥夺资源原本的所有权，让原本右值可以被保存在一个地址中。右值由于没有地址，所以其的生命周期只在产生的一行，即将被销毁的对象被称为将亡值而完全没有地址的临时值称为纯右值。

引用的本质作用是为了在要使用数据时减少拷贝的次数，左值引用在作为函数参数时使用很方便可以减少资源消耗。而在作为函数返回值时，左值引用要求返回的值必须是全局的，不然在函数结束时资源会被回收，但是仅为了解决这个问题就定义全局变量显然不合理，于是出现了右值引用，右值引用可以将原本的将亡值进行引用并剥夺其所有权让其变为左值。

右值引用只能引用右值，而const左值引用还可以引用右值。

可以对左值使用move来使其可以被右值引用以实现语义转移。

移动构造/赋值的赋值对象是右值。

右值引用的用处是移动语义(剥夺资源所有权)和实现完美转发

[【C++】深入剖析C++11中右值引用和左值引用-CSDN博客](https://blog.csdn.net/Dawn_Lillian/article/details/138417640)

<https://blog.csdn.net/Dawn_Lillian/article/details/138417640>

实际上右值引用就是剥夺原本资源所有权的左值引用

**13. STL的六大部件和联系**

STL的六大部件包括：容器，迭代器，算法，适配器，仿函数，分配器

仿函数是对算法的扩充，可以作为其参数。其是重载了调用运算符的类，可以像函数一样使用。

算法通过迭代器对容器进行修改。

适配器是基于底层容器实现特定数据结构，复用容器的功能。

容器依赖分配器实现对内存的管理。

**14.** **STL容器知道哪些，都解释一下，原理是什么，实现一下链表**

Vector，可以在末尾增加和删除元素，其底层是动态数组，支持随机存储。在增加内存空间时，如果当前内存空间不足，会将整体全部移动至新的内存空间并释放原本的内存空间。

Deque，双端队列，可以在头部和尾部进行删除/插入操作。底层为一个管理数组加多个被管理数组，管理数组包含指向被管理数组的指针，被管理数组包含实际的数据，vector要求数据在内存上连续分布而deque允许数据整体呈现离散而在局部进行连续。在增加空间时直接在一个空闲的地方创建即可。

List，本质就是一个双向链表，跟常见的链表的实现相同数据加指向前和后的指针。

Set，底层维护一个红黑树（特殊的平衡树），其会高效自动排序并将元素去重，可以基于此特性来实现排序和去重功能。

Multiset，不会对插入的元素进行检查所以相当于可以插入重复数据的Set。

Map，底层同样维护一个红黑树，与set的所有节点都是一个数据相比，map像是dictionary，其每一个节点都由键值和数据构成，键值充当索引的作用，键值不允许重复。

Multimap，键值可以重复的map，对相同键值的元素的访问方式如下。

auto range = mm.equal\_range(1);

for (auto it = range.first; it != range.second; ++it) {

std::cout << it->first << " => " << it->second << std::endl;

}

容器介绍：

[C++ STL之容器介绍（vector、list、set、map）\_stl容器-CSDN博客](https://blog.csdn.net/new9232/article/details/144159335)

https://blog.csdn.net/new9232/article/details/144159335

容器的原理：

[C++ STL 六组件介绍 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/664872204)

<https://zhuanlan.zhihu.com/p/664872204>

**15.** **hashtable的作用与原理(哈希表，散列表)**

Hashtable的底层实现逻辑是使用了动态数组，通过哈希函数/散列函数来将数值与存放的位置的下标相对应。可以将查找的时间复杂度降低为O(1)。

哈希冲突的解决方法：开放地址法，如果出现了冲突那么按照一定规则将该键值对放入其他位置，查找时逐次比较键来判断是否相同。链表法，在键值对中增加一个指针，在出现冲突时将该指针指向后来的键值对。

哈希函数的确定方法有很多种，使用线性函数的叫做直接定址法。可以有多种方案，选择最适合项目的即可。

[来吧！一文彻底搞定哈希表！ - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/95156642)

<https://zhuanlan.zhihu.com/p/95156642>

**16.操作符重载**

操作符重载：返回值 opertor 运算符（运算值，运算值）

其本质是函数的重载，作为全局运算重载时，参与运算的所有值都被作为参数进行传递。相当于是参数不同的重载。

作为成员函数进行重载时，会调用操作符左边的对象的重载函数，默认隐式将this作为第一个参数，另一个对象作为另一个参数。

如果全局函数需要操作对象的数据那么需要将全局函数声明为类的友元。

使用操作符重载的点：new重载以实现自定义的空间分配；智能指针重载以实现资源的剥夺；无关类型自定义转换

**17. 前置++与后置++的区别，操作符重载的角度分析源码实现**

前置++：

以成员函数形式进行重载Int operator++(){}

以友元函数形式进行重载Firend Int operator++(int& a){}

后置++：

以成员函数形式进行重载Int operator++(int){}

以友元函数形式进行重载Firend int operator++(int& a,int){}

依靠参数中单独的int来确定是前置重载还是后置的重载

前置++是直接修改对原本的对象然后返回原本的对象。

后置++是显拷贝一个原本对象的副本然后修改对象再返回副本。其底层的对对象的修改常常使用前置++来实现。自定义类型后置++计算的过程中有一个拷贝所以性能比较低，需要执行拷贝对象的构造函数之类的函数，内置类型会有编译器优化前置++和后置++效率相似。

**18. 函数模板（泛化和特化的问题),什么是全特化和偏特化**

模板有类和函数两种定义的语法如下：

函数的模板：

template <typename T>

T multiply(T a, T b) {

return a \* b;

}

int main() {

auto result = multiply(3, 4); //隐式推导出 T = int

int result1 = multiply<int>(3,5);//显式生成模板实例

cout << result << endl;

return 0;

}

类的模板：

template <typename T>

class Box {

private:

T value;

public:

Box(T val) : value(val) {}

T getValue() { return value; }

};

int main() {

Box<int> intBox(5);

Box<double> doubleBox(3.14);

cout << intBox.getValue() << " " << doubleBox.getValue() << endl;

return 0;

}

类的部分特化：

template <typename T, typename U>

class Pair {

T first;

U second;

};

template <typename U>

class Pair<int, U> {

int first;

U second;

public:

Pair(int f, U s) : first(f), second(s) {}

};

类的完全特化：

template<typename T>

class MyClass {

public:

void show() { std::cout << "Generic\n"; }

};

// 全特化

template<>//此关键字表明该类仍然是一个模板，不能将全特化直接写成普通类是因为会导致重定义

class MyClass<int> {

public:

void show() { std::cout << "Specialized for int\n"; }

};在c++中函数没有部分特化（因为函数的部分特化实际上就是函数重载），且不建议使用完全特化，特化实际上是定义了一个新的函数/类，函数的完全特化是定义了实际的参数，那么不如直接定义一个单独的函数。模板的调用顺序是优先查看完全特化是否符合然后查看部分特化是否符合，如果均不符合则使用通用模板来调用。特化的核心优势是调用的统一性，特殊的处理可以与泛化的处理使用同样的调用接口。

函数不建议使用全特化是因为函数可以重载，类建议使用是因为类不能重载。

[《深入理解C++模板：从泛型函数到特化/偏特化与分离编译》\_模板泛化特化-CSDN博客](https://blog.csdn.net/Dreaming_TI/article/details/144088334)

<https://blog.csdn.net/Dreaming_TI/article/details/144088334>

**19. STL中常见的算法用过哪些，原理是什么（随机数，查找等）**

遍历：

For\_each(A.begin(),A.end(),func()/func)

//函数名或者仿函数，会将容器中的每一个值作为函数的参数传递

Transform(A.begin(),A.end(),B.begin(),func/func())

//函数会遍历A中的每一个值并将值作为func的参数，然后将返回值插入B中

查找：

Find(A.begin(),A.end(),value)

//value表示需要被查找的元素

//返回查找到的元素的迭代器，如果没有找到就返回end

//底层就是逐个遍历迭代器然后判断是否相等

Find\_if(A.begin(),A.end(),func/func())

//将遍历后的每一个值作为参数传递给函数/仿函数，函数返回bool值以判定是否成功找到

//返回目标值的迭代器

Adjacent\_find(A.begin(),A.end())

//查找相邻的重复元素并返回第一个相邻重复元素的迭代器eg：1 2 3 4 5 5 6

//会返回第一个5的迭代器

//同样是逐个遍历，然后判断元素与后一个元素是否相等

binary\_search(A.begin(),A.begin(),value)

//二分查找，严格要求容器为递增的，查找的效率高于Find

//底层使用STL中的lower\_bound（二分查找算法。）

Count(A.begin(),A.end(),value)

//其底层仍然是逐个遍历然后比较，注意如果是自己定义的数据类型需要重载==运算符。

Count\_if(A.begin(),A.end(),func/func())

//将遍历的值作为参数进行传递

//返回的值为true则计数增加

Sort(A.begin(),A.end(),func/func())

//将值和后一个值作为参数传递，使用的方法仍然是比较两个值的大小，返回true则值不修改，返回为false则调换值的位置。

Random\_shuffle(A.begin(),A.end())

//将范围内的值随机打乱

Merge(A.begin(),A.end(),B.begin(),B.end(),C.begin())

//将AB元素插入C中，由于底层使用了归并排序所以要求必须AB均有序。

//普通的插入使用insert(A.begin(),B.begin(),B.end())

Reverse(A.begin(),A.end())

//翻转容器内的元素

//有两个双向指针，分别指向头尾，交换指针指向的元素然后移动。

[STL- 常用算法（遍历，查找，排序）\_遍历stl-CSDN博客](https://blog.csdn.net/weixin_62848630/article/details/125960096)

<https://blog.csdn.net/weixin_62848630/article/details/125960096>

**面试真题部分**

**1.右值引用的应用，优点，lambda表达式**

最核心的应用是实现移动语义，直接变更资源的所有权，而不需要进行拷贝赋值降低了性能开销。一个具体的例子就是右值引用可以在接受函数返回值时使用，直接获取返回值的所有权，以减少复制对象的开销。实际上如果启用了编译器优化并不会带来性能优化因为编译器会进行优化直接将对象构造在调用者的栈上。

[]()metabl throw() ->int {}

**2.空类的占用的内存，结构体的占用的字节数，虚函数和虚表**

空类的大小是一个字节，用于占位，避免地址重叠。

常见类型的大小（字节）

Short:2

Int:4

Long:8（64位）或4（32位）由于在C++中没有严格规定所以不建议使用

Long long:8

Float:4

Double:8

Char:1

Bool:1

注意内存对齐

如果类中有虚函数那么会建立一个虚函数表，保存了类中所有虚函数的地址，继承父类的虚表，如果子类有覆写那么直接覆盖虚表中的对应条目，新增则在虚表的末尾增加。调用虚函数时通过对象的虚指针找到虚函数的地址。

**3.虚析构函数的作用**

如果不将析构函数声明为虚函数，那么在使用父类指针引用子类时会使用父类的析构函数来释放空间不能完全释放子类的内存空间造成内存泄露。

自己通过new在堆上分配的空间是独立于对象之外的，所以不受对象销毁的影响。

**4.传值和传址，深拷贝和浅拷贝**

传值的本质是进行了一次拷贝赋值，传址则是将内存空间的地址进行传递并没有开辟新的内存空间，浅拷贝是默认的拷贝方式，深拷贝和浅拷贝在类的成员没有指针时效果相同，都是开辟复制原本的值，但是如果存在指针，使用浅拷贝会使新对象和原对象的指针成员指向同一个内存地址，此时就需要使用来深拷贝来避免。

**5.前置++和后置++的本质区别**

前置++直接进行自增然后返回自增后的值，后置++先将原本的值拷贝复制一份然后再进行自增，最后返回拷贝复制的副本。由于过程中存在拷贝复制，后置++的效率低于前置++。

**6.** **map，set的原理，别的容器会那些**

map和set的底层原理都是红黑搜索树，只是map的节点由键和值构成。别的容器还会vector，array，list。

**7.C++内存空间划分，值类型和引用类型，C#的垃圾回收原理**

C++中内存被划分为栈，堆，静态存储区和代码区。值类型直接在栈上分配空间，而引用类型虽然保存地址的变量仍然在栈，但是实际的数据是在堆上分配空间，同时在进行赋值时，值类型往往是拷贝赋值，而引用类型会指向同一个数据。

C#的垃圾回收由三个阶段构成：

标记阶段：从程序的根出发遍历遍其在heap上动态分配的所有对象并标记所有被引用的对象。

清除阶段：清除所有没有被标记的对象。

压缩阶段：移动内存中仍然存活的对象使他们在内存上连续，以减少内存碎片化的问题。但是在此过程中大对象不会被移动，大对象移动的开销高于整理该部分内存带来的性能提升，大对象会被单独放置于一个地方。

**8. new和delete本质是什么，和malloc和free的区别**

new和delete的本质是对malloc和free的封装。malloc和free是C++中直接与系统指令打交道的命令，其较高的底层性也就决定了没有其他功能，而new，delete补充了他们功能的缺少，比如在创建和销毁对象时会调用对象的构造和析构函数，可以进行数组的空间分配。

**9. 内存泄漏如何造成的，如何避免（可以使用智能指针）**

内存泄露来源于没有回收在内存空间申请的资源。最简单也最基础的方法就是每次申请空间都记得回收，如果所需要的内存不大可以在栈上申请，将内存管理交给计算机，或者使用智能指针托管内存空间，也可以采用将所有对空间的分配放到类中，让类的析构函数自动进行空间的回收。

**10. 智能指针有哪些**

最初的智能指针是auto\_ptr，基本实现了内存托管的功能，但是其存在致命的缺陷，可以通过赋值来剥夺资源的所有权，该特性使得auto\_ptr无法在容器中使用，随后C++11中推出了unique\_ptr，该智能指针只允许右值赋值和拷贝，如果想要托管左值需要使用Move，但是上面的两种指针都没有解决共享资源的问题，所以C++中引入了shared\_ptr，此指针实现了对资源所有权的共享，维护了一个值来保存该资源被多少个ptr引用。但是这样如果两个对象分别引用了彼此那么就会构成循环引用资源不会被成功释放，所以又引入了weak\_ptr，来补充，weak\_ptr相较于shared\_ptr不会导致计数的增加或减少。

**11.** **动态链接库和静态链接库的区别，exe是静态链接还是动态链接**

动态链接会在编译时记录库的符号引用，在运行时由操作系统动态库的内容加载到内存中，而静态链接在编译时将代码或数据直接拷贝复制到可执行文件中生成独立的可执行文件。动态链接和静态链接都会生成.exe但是静态链接仅会生成一个独立的.exe而动态链接会依赖于动态链接库文件。

[静态链接库和动态链接库的区别 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/712829798)

<https://zhuanlan.zhihu.com/p/712829798>

**12.指针和引用的区别。**

指针是创建了一个新的变量保存原本的数据的地址，而引用只是对原本的变量取了一个别名。

**13. 宏定义，常见的宏指令，define，ifdef那些。**

宏是C/C++的一个预处理指令，本质上是编译开始前进行的简单文本替换。

#include 包含头文件，会将头文件的内容插入当前文件。<>是去编译器内置的路径和环境变量路径中找文件适用于标准库头文件。””是从父文件目录向上递归查找，适用于自定义或者第三方的库。

#define A B 会将代码中的所有B替换为A

#define MUL(x,y) x\*y

int ret=MUL(2,3);=>int ret=3\*2;

int ret=MUL(2+3,4);=>int ret=2+3\*4

#define STRING(x) #x//将内容替换为字符串

const char \*str=STRING (test);=>const char\* str=”test”

#define VAR(x) ADD\_##X//替换并拼接

VAR(1);=>ADD\_1

typedef 进行类型的替换,与define相比增加了类型检查的限制。不是宏指令

typedef int a

a b=>int a

#ifdef A

程序段1

#else

程序段2

#endif

如果A被预定义过了那么执行程序段1

在编译阶段使用的条件语句，防止头文件被重复引用。

**14.null，nullptr的区别**

null是一个宏定义，本质是整数类型0，而nullptr是一个单独的类型，避免了歧义。