C++（核心版）

MRL Liu

2022 年 07 月 28 日

一、语言特性

C 语言和 C++有什么区别？

（1）设计思想。**C 语言是面向过程的编程语言**，其**使用大量的函数来处理数据**，数据和函数一般是分离的，不存在类的概念；**C++语言是面向对象的编程语言**，其**将数据和操作数据的函数封装为类**，通过抽象出一个个的对象模型来解决问题。

（2）应用场景。**C 语言目前适合于代码体积小、效率高的场景**，例如嵌入式；**C++多用于大型的底层系统场景或者对性能要求高的应用**。

（3）语法。**C++是 C 的扩展**，可以支持 C 的语法，但是 C 不一定支持 C++的语法。例如 **C 采用 malloc 和 free 函数**动态申请和释放内存，而 **C++使用 new 和 delete 运算符**；C 语言中只有**局部**和**全局**两个作用域，而 C++中有**局部**、**全局**、**类**、**名称空间作用域**。C++支持**泛型编程**，C 不支持。

面向对象的三大特性是什么？C++如何实现这些特性？/谈谈对面向对象的理解

面向对象的三大特性分别是**封装**、**继承**和**多态**。

封装是将**数据**和操作数据的**方法**有机结合成**类**，**类**可以通过**访问权限**（public、protected、private）来对外隐藏实现细节，**封装提高了程序的安全性**。

继承可以让子类快速获取父类的属性和方法，并通过**继承权限控制**实现**父类**对**子类**的方法的开放程度，**继承提高了代码的复用性**。

多态是指调用对象的同一个函数，可能会执行不同的实现。**多态**分为**静态多态**和**动态多态**。**静态多态**指的是**函数重载**，同一个函数名，传入不同类型的参数会执行不同的函数，该选择可以在**编译期间**就确定；**动态多态**指的是**函数覆盖**，**动态多态**的基础是**继承**，子类需要**重写**父类的**虚函数**，当**父类指针或者引用**指向**父类对象**时调用**父类**中的虚函数，当父**类指针或者引用**指向**子类对象**时，调用**子类**中的虚函数，该选择必须在**运行期间**完成，**多态提高了代码的扩展性**。

C++和 Java 有什么区别？

	编译流程	语法规则	效率	跨平台
C++	**编译性语言**，其源代码经过 C++编译器编译和链接后生成可执行代码。	C++支持**面向对象**、**面向过程**和**泛型编程**，可以定义**全局变量**和**全局函数**。 C++**支持指针**，允许程序员动态分配内存	最高	不可以
	静态语言，变量定义必须有类型声明	**支持多继承**、**函数重载**、**运算符重载**等		
Java	**解释性语言**，其源代码经过 Java 编译器编译后需要由 Java 虚拟机（JVM）解释执行	Java 只支持**面向对象**，所有代码必须在类中实现，不存在全局变量或者全局函数。 Java **没有指针概念**，使用垃圾回收器实现内存的自动回收，对程序员来讲，Java 内存管理更加安全	较高	可以
	静态语言，变量定义必须有类型声明	Java **支持函数重载**，**不支持多继承**、**运算符重载**等		
Python	**解释性的脚本语言**，其源代码直接由 python 解释器生成并输出	Python 支持**面向对象**和**面向过程**，可以定义全局变量和全局函数。 Python **没有指针概念**，使用垃圾回收器实现内存的自动回收，对程序员来讲，Python 内存管理更加安全	较低	可以
	动态语言，变量定义无类型声明	Python **支持多继承**，不支持**函数重载**和**运算符重载**等		

二、编译过程

请简述一下 C++为什么要使用头文件？/头文件的作用是什么？

（1）**头文件用来保存程序的声明**。C++中的变量必须先声明后定义，重复性使用的声明必须保存在头文件中。

（2）**头文件可以提升程序安全性**。头文件只保存声明，使得代码的各类定义等核心代码可以被加密来保护知识产权。

（3）**头文件具备说明文档的作用**。头文件可以记录各类函数的使用方法而不必给出实现，减轻了程序员阅读程序的负担。

请简述以下 C++的（文件）编译过程？

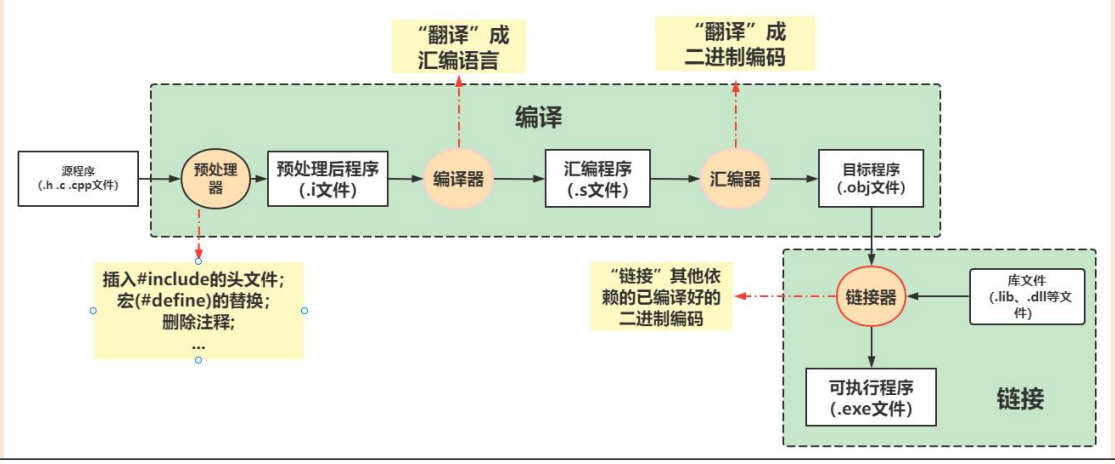
C++的编译过程分为以下 4 个阶段：

第一阶段：**预处阶段**。主要任务：**删除注释**、**头文件的代码拷贝**（**#include**）、**宏定义的文本替换**（**#define**）、**处理条件编译**（**#ifndef** 和**#endif** 等）。请注意，#开头的**预处理指令不属于 C++语句**，因为其没有以;结尾。

第二阶段：**编译阶段**。将上一阶段的文件“翻译”成**汇编语言**。

第三阶段：**汇编阶段**。把**汇编语言**“翻译”成**二进制语言**。

第四阶段：**链接阶段**。将多个**目标文件**（.obj）和**静态链接库**（.lib）链接为最终的**可执行文件**（.exe）



请简述一下静态链接和动态链接的区别。

静态链接发生在**程序**的**编译阶段**，将所有的目标文件（.obj 文件）和第三方**静态链接库**（.lib 文件，本质也是一组.obj 文件）**直接拷贝到最终的可执行程序文件**（.exe）中，。

动态链接发生在**程序**的**运行阶段**，在.exe 文件执行过程中，如果涉及**动态链接库**（.dll 文件）的代码才会导入对应.dll 文件中的代码，此时，.dll 文件是.exe 文件的一种**可执行程序组件**，其不可以单独执行，必须依赖于对应的可执行程序（.exe）执行。

静态链接会使最后生成的.exe 文件的**体积大幅提升**，**动态链接**不会；**动态链接库可以根据需要选择是否导入**，可以用于扩展程序的额外工程；动态链接库还可以进行特殊加密，保护核心代码。

请简述一下#include<>和#include""的区别。

#include<>用于包含 C++的**标准头文件**，其直接查找 C++的**默认头文件路径**。

#include""用于包含程序员**自定义**或者**第三方头文件**，其先从**自定义目录**找，找不到才找 C++的**默认头文件路径**，**如果用#include" " 包含标准头文件，不会报错，但会降低执行效率**。

三、内存模型

简述 C++的内存模型/说下你对 C++内存分配的了解。

C++程序在**运行时**会向**操作系统**申请一块内存空间，C++程序会将这块**内存空间**划分为如下 5 个区域：

内存分区	内存特点	变量类型	声明方式	生命周期	作用域	链接性
栈	系统自动维护	**局部变量** **函数参数** **局部常量**	在代码块中	自动存储持续性	所在代码块	**无链接性**
寄存器	系统自动维护	寄存器变量	在代码块中，使用关键字 register	自动存储持续性	所在代码块	无链接性
（**静态存储区**） 固定存储	系统自动维护	**局部静态变量**	在代码块中，使用 static 关键字	**静态存储持续性**	所在代码块	**无链接性**
		全局静态变量	不在任何函数内，使用 static 关键字		所在文件	**内部链接性**

		全局变量	不在任何函数内		所在文件	**外部链接性**
（**常量存储区**） 固定存储		**全局常量** **函数指针**				
堆	程序员手动维护	**动态变量**（new 分配的指针变量等）	使用 C++运算符 new 声明	动态存储持续性		
代码区	系统自动维护	函数体的二进制代码		**静态存储持续性**		**外部链接性**

C++中变量的作用域、生命周期、链接性、可见性是什么意思？

作用域描述了变量在文件的多大范围内可见，根据作用域，可以将变量分为全局变量、局部变量和形式参数。函数的作用域默认是全局的（因为 C++函数中不能定义函数）。

生命周期指的是变量从**被分配内存**到**最后释放内存**的一段区间，**静态变量**和**函数**都具备静态存储持续性：

.cpp 文件中的**全局静态变量**在程序运行之初就初始化。

函数中的**局部静态变量**在首次执行此函数时进行初始化。

类中的**静态成员变量**在程序运行之初、类实例化之前就已初始化。

链接性指的是**变量如何在不同源文件间共享**，**外部链接性**指的是**变量可以在其他文件中使用**，**内部链接性**指的是**变量只能在本文件中使用**，**无链接性**指的是**变量只能在当前代码块内使用**。**函数**的作用域默认是**外部链接性**。

变量/函数的声明和定义的区别是什么？

变量不**赋值**就是**声明**，一旦**赋值**就是**定义**；函数不定义**函数体**（带{}）就是**声明**，一旦定义**函数体**就是**定义**。

同一个**变量**或**函数**的**声明可以有多个**，但是**定义只能有一个**，且必须**先声明后定义**或者**声明的时候同时定义**。

声明表示该标识符**作用域**的开始，但是**不会分配内存空间**，定义**会分配内存空间**，表示了**生命周期**的开始。

C++的全局变量可不可以定义在头文件中？如何引用其他文件的全局变量？

C++的全局变量不可以在头文件中定义，但是可以在头文件中声明。头文件只能存放变量、函数和类的**声明**和**内联函数的定义**。全局变量如果在当前文件，直接引用即可；如果不在当前文件，则需要用 **extern** 关键字重新**声明**为**外部变量**。

C++中堆和栈的区别是什么？堆和栈中数据的生命周期？

C++中的堆和栈是 C++中数据存储的内存区域，它们有如下区别：

（1）**内存分配不同**。栈中存放**局部常量**、**局部变量**和**函数参数**，由系统自动分配和回收内存；堆中主要存放 new 操作符或者 malloc 等内存分配函数分配的变量，其内存由程序员手动分配和回收。

（2）**数据结构不同**。栈是一种先进后出的数据结构。堆是可以被看成是一棵树，如：堆排序。

（3）**缓存速度不同**。栈使用的缓存比堆的读写速度要快得多。

new 和 malloc 的区别是什么？

```
int* p=(int*)malloc(sizeof(int));
free(p)
```

	new/delete	**malloc()/free()**
本质	C++的运算符	C 的标准库函数
分配大小	自动计算	作为参数手动输入
主要操作	分配/释放内存时**自动调用对象的构造和析构函数，所以 new 用来动态分配对象指针**	分配/释放内存时不自动调用对象的构造和析构函数，所以 malloc 无法动态分配对象指针
安全性	**内存分配失败**时抛出 bad_alloc 异常	**内存分配失败**时返回 null
	进行类型识别检测，返回定义时具体类型指针，如果为 int 指针分配 float 变量时报错	**不进行类型识别检测**，返回 void 指针，需要手动进行类型转换，如果为 int 指针分配 float 大小的字节数不报错。

malloc 出来 20 字节内存，为什么 free 不需要传入 20 呢？不会产生内存泄露吗？

malloc 申请 20 字节的内存，实际上拿到的内存比 20 字节大，多出来的部分用来管理内存，这些管理信息中记录了该内存块的长度。free 释放内存时，需要传入一个 malloc 返回的指针，这个指针标识了申请的内存块的管理信息的所在区，free 内部读取管理信息就可以知道要如何释放了。不过 free 并不会立即将内存块释放掉，甚至上边的数据都不会改写，只会做个标记标识该内存块的所有权已经归还，但是为了安全起

见，free 后建议将指针设为 NULL 来防止释放后有被访问。

new[]和 delete[]为什么必须得配对使用？如果不配对使用呢？

new[]和 delete[]申请 C++内置数据类型（int,float 等）的内存时，可以不配对使用，如果申请的是自定义对象的内存时，必须配对使用。**这是因为 new[]在处理非内置数据类型时，其会在内存中多分配 4 个字节来存储数组的长度**。

如果 new[]处理 C++内置数据类型，delete[]知道 int 是内置类型，也知道其元素大小，只要按大小删除即可，不需要调用析构函数。

如果 new[]处理自定义对象，new[]会在内存起始部分分配 4 个字节作为数组个数。如果用 delete 删除，delete 不会访问前 4 个字节来获取元素大小，只知道要删除起始到结尾的一段内存，它将会只调用一次析构函数；**delete[]则会访问前 4 个字节来获取元素大小，调用多次析构函数**。

三个内存操作函数 memset、memcpy 和 strcpy 的区别

memset 是**内存重置函数**，来对一段内存空间全部设置为某个字符，例如:char a[100];memset(a, '', sizeof(a))。

memcpy 是**内存拷贝函数**，可以拷贝任何数据类型的对象，例如 memcpy(b,a,sizeof(b))。

strcpy 是**字符串拷贝函数**，遇到’\0′结束拷贝。

什么是内存泄露？一般什么情况下会出现内存泄漏？

内存泄露是指程序员在用 **new 运算符**或者 **malloc()**等**内存分配函数**时**没有如期回收内存**，结果导致一直占据该内存单元，这种内存管理失误就是**内存泄露**。内存泄露通常发生的情况有：

（1）**释放和分配的运算符不匹配**，例如用 new 分配指针数组，释放时没有使用 delete[]。

（2）含有**成员指针**的**类**没有在**析构函数**中释放成员指针。

（3）**没有将析构函数定义为虚函数**，如果父类析构函数不是虚函数，那么子类的析构函数无法被调用。

（4）**发生浅复制**，含有成员指针的类**没有自定义拷贝构造函数或者重载赋值运算符（=）**，复制后的对象指向同一块内存。

四、关键字辨析

const 关键字的作用是什么？什么时候使用 const 关键字？

const 关键字用来限定**某个变量是只读的，在编译期间生效。**

（1）**被 const 修饰的对象都是只读的，不可以被修改**。

（2）const **可以修饰几乎所有的变量类型**，但是**只能修饰类成员函数**。

（3）const 修饰的**类成员函数内部只能调用 const 成员函数**

（4）const 修饰的**类成员函数可以构成重载**，const 对象只能调用它的 const 成员函数，非 const 对象优先调用其非 const 函数。

使用场景如下：

（1）修饰**某个变量**作为常量使用。

（2）修饰**函数参数、函数返回值**，防止被函数无意间改动，减少 bug。

（3）修饰**类成员函数**，防止函数无意间改动，减少 bug。

constexpr 关键字的作用是什么？什么时候需要使用 constexpr 关键字？

constexpr 关键字**用来判断变量的值是否是一个常量表达式**（复杂系统中可能很难判断一个初始值是否是常量表达式），**在编译过程中如果不是则会报错**。它给编译器优化建议，即告知该类型可以在编译期间就计算出值，但编译器不一定采纳

constexpr 修饰**变量**，这个变量初始化后就具备了**常量的性质**，例如用来初始化数组长度；

constexpr 修饰**函数**，这个函数会被当做一个**常量表达式函数**，该函数只能有一条 return 语句来返回一个非空值。

volatile 关键字的作用是什么？什么时候需要使用 volatile 关键字？

volatile 关键字**用来声明某个变量的值是可写的，经常更改的，编译器不要对其进行优化**。C++编译器为了提升执行效率，经常会将一些变量的存储到寄存器中，这样很可能在关键时刻无法更改变量的值，**volatile 则不允许编**

译器自行将变量存储到寄存器中。volatile 的作用可以认为和 const 是相反的，volatile 一般用在：
（1）多任务程序中需要多个任务共享的一些标志变量。
（2）实时嵌入式程序中的中断处理程序中修改的供其他程序检测的变量。

static 关键字的作用是什么？什么时候使用 static 关键字？

static 关键字**修饰的变量是一种静态变量**，所有的静态变量都存储在 C++的全局数据区，其生命周期随程序终止而终止。使用场景如下：
（1）**静态全局变量和静态全局函数只能在本文件中访问**，不能在其它文件中访问，即便是 extern 外部声明也不可以（**限制数据的作用域，外部可见性缩小**）。
（2）**静态局部变量在首次执行时初始化，直至程序运行结束后才释放**（**生命周期延长**）。
（3）**类的静态成员变量被类的所有对象共享，其名义上属于该类，但实际可以只通过类名访问（共享内存）**。
（4）**类的静态成员函数只能调用静态成员变量，且函数内没有 this 指针**。

extern 关键字的作用是什么？什么时候使用 extern 关键字？

extern 关键字**用来修饰全局变量或者全局函数的声明，表示其定义在别的文件中，该变量是外部变量**。
extern 修饰的变量只能是全局的，且存储在静态存储区（全局区）。
extern 只能修饰声明，不能修饰定义。
extern 不能和 static 同时使用，因为 static 是会限制变量的作用域为本文件，extern 是扩展变量作用域到其他文件，两者的作用正好相反。
extern 和“c”连用时，表示编译器以 c 语言规则去编译函数名修饰声明。如 extern "C" void fun(int a, int b);
假如不同文件要使用同一个全局变量，在 A.cpp 定义全局变量 a，在 B.cpp 使用 extern 重新声明 a 即可是，或者将 extern 声明放入头文件。

inline 关键字的作用是什么？什么时候需要使用 inline 关键字？

inline 关键字用来显示声明一个函数为内联函数，类中定义的函数都默认为内联函数。
内联函数可以减少函数调用的开销，提高程序执行效率，是一种优化效率的手段。编译器处理内联函数时，不会单独进行函数调用，而是直接将整个函数体的代码插入调用语句处，就像整个函数体在调用处被重写了一遍一样，**这个过程发生在编译期间**。很显然，使用内联函数会使最终可执行程序的体积增加，因为内联函数是以空间换取时间。
内联函数适用于小而简单、执行很快的函数，如果一个函数非常庞大或者需要消耗大量时间，那么将其声明为内联函数虽然节省了函数调用的时间，但是却让程序体积增加了更多，这样程序执行速度很可能反而会下降。现代 C++编译器提供了内联函数的保护机制，**一般程序员声明的内联函数只是给编译器的建议**，具体编译器是否真的按照内联函数的方式处理可能由内部算法决定。

explicit/implicit 关键字的作用是什么？什么时候需要使用 explicit/implicit 关键字？

explicit 关键字修饰**类构造函数**，表明该构造函数**只可以显示调用，不能由编译器隐式调用**
implicit 关键字修饰**类构造函数**，表明该构造函数**可以由编译器隐式调用，该操作函数重载时可能存在歧义**。

你了解 C++11 的类型推导吗？请区分下 C++11 的 auto 和 decltype

auto 和 decltype 都是让编译器在编译期间就推导出变量的类型。两者的区别如下：
auto 在使用时必须进行显示初始化，让编译器能够将变量的类型设置为初始值的类型，不能推断数组、不能用作函数参数：

```
auto a = 12; auto pt = &a;
```

decltype 将变量的类型声明为表达式指定的类型，主要用于泛型编程，解决有些类型由模板参数来决定的问题。

```
int i = 4;
decltype(i) a; //推导结果为 int。a 的类型为 int。
```

typedef 的用法

typedef 是 C++中用来给复杂的**数据类型**或者**声明定义别名**，可以提升程序员的开发效率、程序的可读性和兼容性

i++和++i 的区别

i++是先使用 i 的值然后再让 i 自增 1，该表达式需要先返回一个旧值所以编译器会自动创建一个**临时对象**返回。
++i 直接执行 i 的自增，不会创建**临时对象**，所以++i 比 i++的执行效率更高。

do……while 和 while……do 有什么区别？

do...while 先执行一次循环再判断条件，while...do 先判断条件再选择是否执行循环。

五、数据类型

C++的不同变量占据多少字节？16 位机、32 位机、64 位机是什么含义？

C++的变量的内存大小和它运行时的系统位数有关。例如指针类型在 16 位机--2 字节、32 位机--4 字节、64 位机--8 字节。16 位、32 位、64 **位指的是计算机的 CPU 一次可以处理数据的字节数**。

C++一般用**运算符 sizeof()**来获取变量占据的**字节数**，它的原理是：对于一些**基础变量**，在**编译期间**判断类型就可以获取字节数（可能有一个映射表）；对于一些**复杂变量**，例如不定长数组，在**运行期间**才能计算其字节数。

下表列出的是目前为主的 64 位系统，1 字节=8 位比特：

关键字	数据类型	内存大小
void	无类型	—（没有 void 对象，只有 void 指针）
bool	布尔型	1 个字节
char	字符型	**1** 个字节
wchar_t	宽字符型	2 或 4 个字节
int	整数型	**4** 个字节
float	浮点型	4 个字节
double	双浮点型	8 个字节
	指针	**8** 个字节
string	字符串	**字符数+1**(最后一个是空字符)
	对象	较复杂
struct	结构体	较复杂（和对象一致）
union	联合体	联合体的各个成员变量的**内存最大值**
enum	枚举	枚举中**单个成员的内存**（一般为 int，4 个字节）

C++中如何估计类对象的内存大小（重点关注红色字体部分）？

不考虑内存对齐的话，**C++中的类大小计算与成员函数和静态成员变量无关，影响类大小的有普通成员变量、虚函数**。所以普通的类的对象的内存大小是普通成员变量的内存之和+虚指针大小（如果有类中有虚函数的话，32 位机上是 4 个字节）。如果该类存在父类，则需要加上父类对象的大小，虚继承的父类认为其存在一个虚指针

特殊情况：**空类（无函数无变量）也可以实例化，其对象的大小为 1**，这是因为类的实例化必须在内存中分配一块地址，所以编译器会为空类分配一个字节。

C++类的内存对齐（重点关注红色字体部分）

C++在实际分配类的对象的内存时会存在一个内**存对齐原则**，并不是严格按照各个成员变量的内存大小顺序分配，内存对齐比较复杂，这里不详细介绍，只介绍内存对齐的意义：

内存对齐可以使 CPU 处理 C++程序的效率大幅度提升；部分硬件平台的 CPU 无法访问其地址空间的任意数据，所以必须使用内存对齐。

请详细介绍下 C++的 4 种强制类型转换的使用场景？

C 语言中，代码可以使用小括号来进行基本数据类型的强制类型转换，

上行转换是将**子类指针**转换为**父类指针**，一般是安全的

下行转换是将**父类指针**转换为**子类指针**，一般是不安全的

C++新增了 **4** 个关键字来**强制类型转换**，它们的使用场景和特点如下：

（1）静态转换（static_cast）：编译期间完成

基本数据类型之间的转换（如 int 转换为 char）；

将**任何类型的表达式**转换为 **void 类型**；

可以进行下行转换，但是不安全

（2）**常量转换（const_cast）：编译期间完成**

只能用于将 const 的**指针或引用**转换为**非 const** 的**指针或引用**，不能转换 const 的变量。
（3）**重解释转换（reinterpret_cast）：编译期间完成**
可以**在指针和引用里进行各种转换**，将**基本类型转换为指针**，也可以把指针转换为数组；但是这种转换是**简单的比特位的拷贝，平台移植性差，不安全**。
（4）动态转换（dynamic_cast）：运行期间完成
只能用于将**具有虚函数的父类指针**转换为**子类指针**的情况（下行转换），这种转换是安全的

六、指针和智能指针

指针和引用有什么区别？什么情况下用指针，什么情况下用引用？

1、定义不同。**指针是一种特殊的变量**，特殊在其 value 是另一个变量的地址，所有**指针的内存大小都是固定**的（例如在 X86 中为 4 个字节，在 X64 中为 8 个字节）；**引用是一个已存在的变量的别名**，引用的**内存大小由其指向的变量类型**决定，引用在定义时需要加&，使用时不能加&。
2、初始化要求不一样。**指针可以在任何时候初始化，引用只能在创建时初始化；指针可以为空，引用不可以为空**；
3、**引用本质上还是指针**，其底层的实现就是**指针**。

你了解 C++11 的左值和右值吗？谈论下左值引用和右值引用？

可以取地址的、有名字的就是**左值** （int a = 10; // 其中 a 就是左值）
不能取地址的、没有名字的就是**右值** （int a = 10; // 其中 10 就是右值）
1. **左值引用**就是对一个**左值**进行**引用**。左值引用使用&声明，即我们通常所说的引用就是左值引用。
2. **右值引用**就是对一个**右值**进行引用。右值引用使用&&声明，右值引用可以实现移动语义。

什么是空指针？nullptr 和 NULL 的区别是什么？

在 C++中 nullptr 和 NULL 都可以表示空指针，但是 nullptr 更安全。
因为 NULL 在**函数重载时存在二义性问题**，因为 **NULL 即可以表示 0 也可以表示空指针**，例如两个重载函数一个接收 int，一个接收 void*指针，那么 NULL 参数就会引起歧义，它是表示指针还是数字 0 就不确定。
nullptr 是 C++11 引入的空指针，它专指空指针，传参时传入 nullptr 可以避免歧义。

什么是裸指针和野指针？

裸指针就是 C++中通过 new 在堆上分配内存的**普通指针**，它**使用起来很不安全**，需要格外小心；
野指针就是**指针指向的对象是不确定的、随机的，**使用野指针，可能会访问到空指针、修改未知数据而报错。
造成野指针的原因可能有：
（1）指针**未初始化**或者**赋值不正确**；
（2）指针**指向一个局部变量**，离开该局部变量的作用域后就自动释放
（3）指针进行复杂的**拷贝、赋值、移动**操作，可能指向被早已释放的内存
（4）指针**释放后没有设为空**；

你了解深拷贝和浅拷贝吗？那移动语义和移动构造函数呢？

深拷贝和**浅拷贝**的**根本区别**在于**是否拷贝对象的实体**。例如**复制指针时，只复制指针变量**，不复制指针指向的对象就是**浅复制**，反之就是**深复制**。一般来讲，**浅复制**容易发生在**拷贝构造函数、赋值运算法重载**的实现中。**浅拷贝**是**不安全**的，因为它会产生**野指针，访问到空指针而报错**。

你了解 C++的移动语义和移动构造函数呢？

移动语义是 C++11 新引入的概念，**它转移的是对象所有权**。例如移动指针指向的内存，指针移动后，旧指针变为 NULL，新指针指向旧指针指向的内存，这个过程就是**将对象所有权从旧指针转移到了新指针**。
如何**让一个类支持移动语义**：**在类中重写移动构造函数**（注意和**拷贝构造函数**对比）：

```
A(A&& a){this->data_=a.data_;a.data_=nullptr;}//移动构造函数：
A(A& a){this->data_=a.data_;}//拷贝构造函数
```

如果**调用一个对象的移动语义**：通过 **move 函数**调用 A c=std::move(a);// 调用移动构造函数。

可见，**只有定义了移动构造函数的类的对象才支持移动操作**，C++中的 **STL 基本都支持移动语义**。

你了解智能指针吗？请简述下 C++中的智能指针。

裸指针的使用很不安全，很容易变成**野指针**，所以 C++引入了**智能指针**的概念。

智能指针（pointer-like classes，像指针的类）和**裸指针**的**根本区别**在于，**它会自动释放自己指向的对象**。**智能指针**本质是一个 **C++类模板**，封装了 C++的普通类指针，重写了**析构函数**、**拷贝构造函数**、**移动构造函数**，重载了*、->、=操作符函数。

C++中一共设计 **4 种智能指针**，C++98 提供了 **auto_ptr**；C++11 舍弃了 C++98 的 auto_ptr，在头文件<memory>提供了 3 个新的智能指针类型，这些智能指针都是在**裸指针**的基础上封装而来的，对比如下：

	对象所有权	执行效率	应用频率	安全性	特点
裸指针	—	最高	最高	较低	手动 delete、复制、拷贝、处理异常等
auto_ptr<T>	**专属所有权**	较高	最高	较高	**只支持移动**，不支持拷贝和赋值;
unique_ptr<T>	**专属所有权**	较高	最高	较高	**只支持移动**，不支持拷贝和赋值;
shared_ptr<T>	**共享所有权**	较低	较低	较高	支持拷贝、赋值和移动；**循环引用**时出错
weak_ptr<T>	**临时所有权**	较低	最低	较高	支持拷贝、赋值和移动；弥补**循环引用**

unique_ptr 具有**专属所有权，**该指针**只支持移动**，不支持拷贝和赋值，其内部实现了**移动构造函数**，执行性能和裸指针接近。unique_ptr 作为类成员时，不需要在析构函数中 delete；unique_ptr **在编译期间就会进行安全检查**，所以 **unique_ptr 和裸指针性能接近，还更加安全**，所以经常被使用。

shared_ptr 具有共享所有权，该指针**支持移动、赋值和拷贝**，内部使用一个**原子级别**的**引用计数**来维护指向的对象，在发生拷贝或者赋值时，**引用计数+1**；在销毁对象时，**当引用计数为 0 时才会真正销毁**，所以 shared_ptr 可以保证同一个对象被安全地多次引用，**适合用于并发场景，可以确保多个线程同时都能读到对象，但是修改指向的对象时仍然需要加锁，其执行效率较低**。

weak_ptr 具有**临时所有权**，它专门用于解决 shared_ptr 的**循环引用**问题：父类持有指向子类的 shared_ptr，子类持有指向父类的 shared_ptr 时，**两个对象互相依赖无法真正销毁**。只要将其中一个改为 weak_ptr 就可以解决这个问题。weak_ptr 可以认为是 shared_ptr 的补充，经常需要和 shared_ptr 一起使用。

智能指针是否存在内存泄漏的情况？

智能指针依然可能会发生**内存泄漏**的情况。当两个类对象中各自有一个 **shared_ptr** 指向对方时，会造成**循环引用，两个对象互相依赖无法真正销毁**，从而导致内存泄露。**解决办法是使用弱指针** weak_ptr，weak_ptr 的**构造函数**不会修改**引用计数**的值，可以确保两个对象被真正销毁。

七、类的封装

C++和 C 中的 struct 有什么区别？

（1）C 中的 struct 内部**只能有变量，不可以有函数**；C++中的 struct 内部**可以有变量和函数，且函数可以是虚函数**。

（2）C 中的 struct 的**所有变量默认都是 public，不可更改**；C++的 **struct 的所有成员默认都是 public，可以更改**。

（3）C 中的 struct 的**不可以直接初始化变量**，C++可以

（4）C 中的 struct **不可以继承**，C++可以。

（5）C 中的 struct **在使用时必须添加 struct 关键字**（用 typedef 起别名除外），C++中的 struct 使用时不需要添加 struct 关键字。

C++的 struct 和 class 的区别？

（1）struct 的**默认访问权限是 public**，class 的**默认访问权限是** private。

（2）struct 的**默认按照 public 继承**；class **默认按照 private 继承**。

（3）struct **不可以用于定义模板参数**；class 可以用于定义模板参数：template<class t=””>

（4）struct **一般用于描述一个数据结构的集合**，内部一般是各种变量和构造函数；class 是一个对象数据的封装，内部是变量和各种方法。

C++类的访问权限控制/访问级别。

C++成员的默认访问权限是 private

访问权限	类外（实例化对象）	类内成员	子类成员	友元函数	友元类
public	可访问	可访问	可访问	可访问	可访问
protected	**不可访问**	可访问	可访问	可访问	可访问
private	**不可访问**	可访问	**不可访问**	可访问	可访问

C++类的构造函数和析构函数。

构造函数是 C++类中**创建对象**时**自动调用**的**特殊函数**，可以自定义**函数参数**，但无**返回值**，可以**重载**。

析构函数是 C++类中**销毁对象**时**自动调用**的**特殊函数**，无**函数参数**，无**返回值**，不可以**重载**。

C++类会提供默认的构造函数和析构函数，但是**构造函数一般要自定义**，

如果**类中有指针成员变量时，析构函数必须自定义，且建议声明为虚函数**。

构造函数和析构函数的对比如下：

	形式	访问权限	参数	返回值	重载方式	虚函数
构造函数	类名(参数列表){};	一般为 public，**单例模式**中设为 private	自定义	无	可以重载，**一般有多个构造函数**	不能声明为虚函数
析构函数	~类名(){};	public	无	无	不可以重载，**只能有一个析构函数**	推荐声明为虚函数

构造函数中有一种特殊的**初始化列表**，用其来初始化类的成员变量**执行效率会更高**：

```
// 构造函数(2 种写法实现的功能一致，但是第一种写法效率更高)
complex(double r=0,double i=0):re(r),im(i){}
complex(double r=0,double i=0){re=r;im=i;}
```

简述一下 C++的 RAII 机制？

RAII 是 Resource Acquisition Is Initialization（“**资源获取就是初始化**”）的简称，是 C++的管理资源、避免泄漏的方法，实际上我们已经在不自觉的使用。**RAII 的做法是使用一个对象，**将**对资源的申请放在它的构造函数**中，将**对资源的释放放在它的析构函数**中，这样**在对象生命周期内保持对资源的访问**。这里的资源不单单指使用 new 或 delete 分配和释放内存资源，也可以指网络套接字、互斥锁、文件句柄等。这样当执行某些操作时就会自动执行资源的申请和释放操作。

假如我们使用 RAII 思想来计算函数执行用时，可以创建一个 timer 类，求构造函数记录此刻运行时间，析构函数记录此时已运行时间并且输出。然后让 timer 和函数 fun 放在一个代码块中执行即可：{timer t;fun()}。

ScopeExit 就是基于 RAII 机制实现的一个资源管理类，当退出它的作用域时自动执行资源的销毁操作。

请简单介绍下 C++类中的 this 指针。

this 指针是 C++中类中隐藏的一个 const 的指向当前对象的特殊指针。

this 指针的作用是在类普通成员函数中标识当前调用对象。例如 stu.show()；对成员函数 show()来讲，其内部可以访问 this 指针，该指针指向当前调用它的对象 stu，可以用来调用该类的其他普通成员函数和成员变量。

this 指针只能在类普通成员函数内部使用，不能在类的静态成员函数中使用，因为静态成员函数可以直接调用，没有调用对象。

请介绍下 C++中的友元类和友元函数

类 A 中如果添加一个用 friend 修饰的全局函数的声明，则该函数被视为类 A 的友元函数，友元函数本身不属于类 A，**友元函数可以访问类 A 的所有权限的成员**。

类 A 中如果添加一个用 friend 修饰的类 B 的声明，则类 B 被视为类 A 的友元类，**友元类 B 的所有成员函数可以访问类 A 的所有权限的成员**。

友元关系是单向性的，不能被继承，也可以被传递。友元函数和友元类可以提高程序运行效率（让编译器放弃类型检查和安全性检查），**但是它破坏了类的封装性和安全性**，一般推荐不使用或者少用。

请简单介绍下 C++中的临时对象？什么时候会产生临时对象？

C++中的**临时对象**，也叫**无名对象**，是一种不需要声明对象名称、由编译器自动隐式创建的对象，其通过“类型名()”的形式来创建，常见的有以下几种情况：

（1）**按值传参**。在调用函数 F(A a)时需要传入一个对象 a，这时编译器会自动创建一个和 a 一样的临时对象去传入函数中。

（2）**按值返回**。return A();这种情况下编译器会创建一个 A()的临时对象，然后将临时对象返回，原有

的对象销毁。

（3）**类型转换**。假设类 A 有一个可输入一个 int 的构造函数，那么在执行语句 A a=10;时，10 会被编译器隐式调用构造函数构建一个 A 的临时对象赋给 a。

由于**临时对象也需要调用构造函数和析构函数**，所以也会增加程序的开销，**在实际编程中要尽可能避免临时对象的产生，例如多使用按引用传递和按引用返回**。

你了解 C++中的可调用对象？/bind、function 和 lambda 表达式吗？

可调用对象就是可以让一个**函数调用**作为**其他函数**的**参数**。C++中的**可调用对象**有以下几种情况：

(1)**普通函数**。bool result= cmp(a, b); //cmp 是一个返回值为 bool 的普通函数

(2)**函数指针**。

```
bool (*p)(const int &a, const int &b);//创建一个函数指针
p = cmp;//p 指向了一个函数调用
```

(3)**仿函数**（重载了()运算符的类对象）

```
class MyPlus{
  int operator()(const int &a , const int &b) const{ return a + b; }
};
MyPlus a;//定义一个仿函数
a(1,2);//调用仿函数
```

(4)**匿名函数**（lambda 表达式）。lambda 表达式定义了一个匿名函数，可以快速插入程序中，不需要单独定义一个函数体，让程序更加灵活简介，例如：

```
auto f = [](int a) -> int { return a + 1; };//定义一个 lambda 表达，-> int 表示返回一个 int
auto f = [](int a) { return a + 1; };//定义一个 lambda 表达，省略返回值
std::cout << f(1) << std::endl; // 输出: 2
```

(5)**std::function 对象**。function 可以认为就是 C++11 中的可调用对象的包装器。

```
std::function<int(int, int)> Func;//声明一个返回值为 int，参数为两个 int 的可调用对象类型
```

std::bind()用来将参数和可调用对象一起绑定，绑定结果可以使用 function 保存。例如：

```
std::function<void()> f_display= std::bind(print_num, 31337);//print_num 是函数名，31337 是为它输入的参数。
```

八、类的继承

请简单介绍下 C++类的继承机制？

继承可以让子类快速获取父类的属性和方法，常见的继承有 3 种形式：

实现继承：父类提供属性和方法，子类不需要添加额外代码（对应 C++的**普通函数**）。

可视继承：父类提供属性和方法，子类可以重写父类的方法（对应 C++的**虚函数**）

接口继承：父类提供属性和方法名，子类必须实现父类的方法（对应 C++的**纯虚函数**）

C++中按照**继承权限**可以分为：

继承方式	公有成员	保护成员	私有成员
public	公有成员->公有成员	保护成员->保护成员	父类私有成员，子类不可访问
protected	公有成员->保护成员	保护成员->保护成员	父类私有成员，子类不可访问
private	公有成员->私有成员	保护成员->私有成员	父类私有成员，子类不可访问

C++一般最常用的是**公有继承**（public），显然**保护继承和私有继承会让子类变成一个外界不可访问的黑盒**。

C++的继承中，子类会继承父类的**所有成员变量**和**部分成员函数**，**父类的成员变量会在子类中复制一份，子类会继承父类成员函数的调用权，而不会复制函数代码**。父类的如下方法无法被子类继承：

（1）**父类的构造函数、析构函数和拷贝构造函数**。

（2）**父类的重载运算符**。

（3）**父类的友元函数**

请介绍下 C++中单继承、多继承和虚继承的区别。

C++允许**单继承**、**多继承**和**虚继承**。多继承往往会使代码和问题变得复杂，所以**在实际的开发中，不推荐使用多继承和虚继承**，java 等语言不允许多继承。

单继承是子类只有一个父类；

多继承是子类有多个父类，**多继承存在一种特殊情况：菱形继承**，容易存在数据冗余和二义性问题。**虚继承是为了解决多继承时的命名冲突和冗余数据问题**，虚继承允许派生类中只保留一份间接基类的成员。虚基类不论被继承多少次，其子类中都只包含一份虚基类的成员。C++标准库中的 iostream 就是虚继承的实际应用案例。

九、类的多态

请简单介绍下 C++类的多态机制、谈一谈重载、重写和隐藏？

多态是指调用对象的同一个函数，可能会执行不同的实现。**多态**分为**静态多态**和**动态多态**。

静态多态（也叫早期绑定），指的是发生在**编译期间**的**函数重载**，在同一作用域中，同名函数的参数列表不同时构成函数重载，与返回值无关；C++中的重载分为**函数重载**和**运算符重载** 2 种形式。

动态多态（也叫后期绑定），指的是发生在**运行期间**的**函数覆盖**，**动态多态**的基础是**继承**，子类需要**重写**父类的**虚函数。**在不同的作用域中，子类的成员函数就会覆盖父类中的同名成员函数，当**父类指针或者引用**指向**父类对象**时调用**父类**中的虚函数，当父**类指针或者引用**指向**子类对象**时，调用**子类**中的虚函数。

隐藏是指在不同作用域中，同名函数（不要求返回值和参数列表相同）构成隐藏，例如子类函数隐藏父类函数，类成员函数隐藏类外的全局函数。

重载、重写和隐藏的对比如下：

	作用域	虚函数	函数名	形参列表	返回值类型	相关术语	底层实现
重载	相同	无关	相同	不同	无关	**静态多态**、函数重载、**运算符重载**	命名倾轧技术，编译时将**同一作用域**的同名函数按照某种规则生成不同的函数名
重写	不同	有关	相同	相同	相同	**动态多态**、子类重写父类的**虚函数**	每个有虚函数的对象都有一个**虚指针**，指向一个记录该类所有虚函数的**虚函数表**
隐藏	不同	无关	相同	无关	无关	**子类**与**父类**、**类成员**与**类外**	

请简单介绍下你对 C++中运算符重载的理解。

C++中的重载分为**函数重载**和**运算符重载**，**函数重载**容易理解，**运算符重载**则是 C++特有的机制。C++允许重载大部分 C++内置的运算符，**运算符重载函数**可以声明为**类的成员函数**或者**全局函数**。

假设要为类 A 重载+运算符，使得它的对象可以进行加法运算，有 2 种实现方式：

（1）声明为类 A 的非成员函数（**全局函数**）（传递 2 个参数）：

```
A operator+(const A&,const A&);
```

（2）声明为**类 A 的成员函数**（只传递 1 个参数即可，另一个是隐藏的 this 指针）：

```
A operator+(const A&);
```

请简单介绍下你对虚函数原理的理解。

虚函数是用 virtual 关键字声明的**成员函数**，它将允许**子类**覆盖该函数，是 C++中类的**动态多态**的实现。

动态多态（实现条件）要求：

（1）调用函数的对象必须是**指针**或**引用**;

（2）被调用的函数必须是**虚函数**且子类完成了虚函数的重写。

动态多态（内部原理）：为**每个具有虚函数的类对象**在内存中会添加一个隐藏的**虚指针，虚指针**指向它所属类的**虚函数表**，虚函数表像一个数组，存放了**当前对象和其父类的每个虚函数的地址**。如果子类重写了父类的虚函数，则子类的虚函数表中保存的是**子类的虚函数地址**；如果子类没有重写父类的虚函数，则**子类保存的是父类的虚函数地址**。

动态多态（最终目的）：**父类指针**指向**子类对象**时，可以调用对应子类的**同名重写函数**，让父类看起来有多种“执行形态”。

例如，父类 A 有两个子类 B 和 C，A 中有一个虚函数 method()，且 B 和 C 中也都重写了 method()。

```
A *a;//定义父类指针
a=new B();//父类指针指向子类对象
a->method();//执行 B 的 method()
a=new C();
a->method();//执行 C 的 method()
```

类中的所有成员函数都可以设为虚函数吗？

C++类中有 3 类函数不能声明为**虚函数**：

（1）**内联函数**。内联函数是在**编译期**用函数体来替换函数的调用指令，而虚函数要求**运行时**根据对象类型才能知道调用哪个虚函数。

（2）**构造函数**。类的构造函数不能声明为虚函数，即使声明了也会调用父类的函数，因为子类还没构造好，不会产生多态（子类构造好才有多态）；相反，**类的析构函数被推荐声明为虚函数**，因为这样子类就可以覆盖父类的析构函数，如果不能覆盖父类析构函数的话。

（3）**静态函数。静态成员函数中不存在 this 指针**，无法知道自身的调用对象是谁，虚函数必须知道自己是被谁调用，所以矛盾。

为什么要将析构函数设为虚函数？

防止内存泄露。如果父类中的析构函数没有声明为虚函数，父类指针指向子类对象时，则当**父类指针释放时不会调用子类对象的析构函数**，而是调用父类的析构函数，如果子类析构函数需要做某些释放资源的操作，就可能会造成内存泄露。

请问 C++中的纯虚函数的作用是什么？什么场景需要用到纯虚函数？

纯虚函数是在虚函数的声明中加“=0”，**含有一个即以上的纯虚函数的类就是抽象类，抽象类无法实例化对象**，只能作为 C++的接口使用。**抽象类的子类必须实现其父类的所有纯虚函数**。

纯虚函数的作用就是将类的声明和实现分离，父类负责声明，子类负责实现，为其他程序提供一个适当的、通用的、标准化接口。当不知道某个方法如何实现或者有多种实现方式时，可以声明为纯虚函数。

十、泛型编程

请简单介绍下 C++的 STL？STL 的六大组件是什么？

STL 是 C++中的标准模板库，STL 的设计时将数据和操作分离（与类的封装相反），主要有六大组件：

（1）**容器**（Containers）本质是 C++的**类模板**，提供**各种数据结构**来存储数据。

（2）**分配器**（Allocators）本质是 C++的**类模板**，为容器的**内存分配**提供支持。

（3）**算法**（Algorithms）本质是 C++的**函数模板**，封装了对于容器内数据的**常见算法操作**。

（4）**迭代器**（Iterators）本质是 C++的**类模板**，一种泛化的**智能指针**，**每个容器都有专属的迭代器**。

（5）**仿函数**（Functors）本质是 C++的**类模板**，重载了 operator()函数，作用**类似于函数**。

（6）**适配器**（Adapters）本质是 C++的**类模板**，可以对容器、迭代器、仿函数进行**类型转换**。

六大组件的关系如下：**容器**通过**分配器**获得**内存空间**，**算法**通过**迭代器**操作**容器**中的**数据**，**仿函数**协助完成**算法设计**（传入函数参数），**适配器**可以修饰**仿函数**或者**容器**来适配不同的算法。

C++的 string 内部是怎么实现的，它的常用 API 有哪些？

string 不属于 C++的 STL 库，但是使用频率也非常高。**string 本质上是一个保存字符的序列容器，其末尾字符是空字符'0'**。

说一说 STL 中有哪些常见的容器？

STL 中的容器可以分为**序列式容器**、**关联式容器**和**容器适配器**三种类型，

（1）**序列式容器**：元素的插入位置同元素的值无关，元素是无序的

容器名	中文名	元素存储方式	底层数据结构	查询效率	增删效率
array	固定数组	连续存放	一维数组		
vector	动态数组		一维数组	**O(1)**	O(n)
deque	双向队列		分段连续空间	**O(1)**	O(n)
list	双向链表	不连续存放	堆	O(n)	**O(1)**
forward_list	单向链表				

（2）**关联式容器**：元素的插入位置同元素的值有关，元素是排序的

容器名	中文名	是否允许 Key 重复	是否排序	底层数据结构	查询效率	增删效率
set	集合	否	**有序**	**红黑树**	**O(logN)**	**O(logN)**
map	映射 (first,second)					
multiset	集合	是				
multimap	映射 (first,second)					
unordered_set	哈希集合	否	**无序**	**哈希表**	**O(1)**	**O(1)**
unordered_map	哈希映射 (first,second)					
unordered_multiset	哈希集合	是				
unordered_multimap	哈希映射 (first,second)					

（3）**容器适配器**：在序列式容器上封装得来

容器名	中文名	存取方式	底层数据结构	插入位置	取出位置
stack	栈	先进后出	一维数组	栈顶	栈顶
queue	队列	先进先出		队头	队尾
priority_queue	优先级队列	按优先级出（默认最大先出）	堆	队头	队尾

C++直接使用数组好还是使用 STL 中的 array？它是怎么实现的？

array 是 STL 中的序列式容器，它是一段无法扩容的固定长度的顺序存储的数组结构。它是在普通数组的基础上，添加一些常见的成员函数，**它比普通数组更安全，并且效率接近**。由于 array 是 STL 库中的，所以**它可以直接使用其算法库的函数**（如 sort 等），它使用 at()的成员函数来获取数组内的元素，**不用担心**

数组越界的问题。

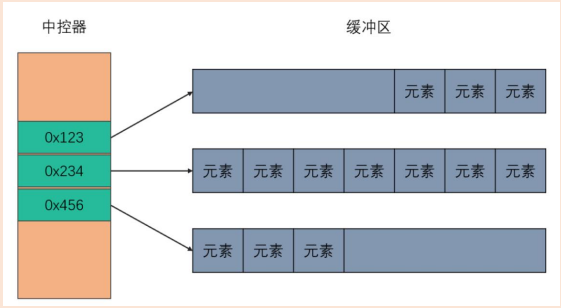
C++的 vector 是怎么实现的？扩容机制？resize、reserve、clear 是怎么实现的？

vector 是 STL 中的序列式容器，它是一段可以单向扩容的顺序存储的数组结构。它内存**存在三个迭代器：first**（指向起始字节位置）、**last**（指向最后一个存储的元素）和 **end**（指向末尾字节位置）。size()获取其容器中存储的元素个数，capacity()获取其容器此时的容量。**vector 每次插入新节点，不一定分配内存，只有超过容量时才会申请新内存。当 size=capacity 时，vector 就会自动扩容一倍**。reserve()是主动设置的 vector 容量，resize()是主动调整的 vector 的元素个数，不足补足，超过删除。clear()把 size 设置为 0，capacity 不变。**vector 的扩容是新分配一段连续空间，然后将原来空间上的元素拷贝到新空间上并弃用原有空间，vector 的扩容是非常耗时的**。

C++的 deque 的内部是怎么实现的？它和 queue 和 stack 的关系是什么？

deque 是 STL 中的序列式容器，它是一段可以双向扩容的顺序存储的双向队列结构。由于其双端都可以添加或者删除元素，所以其本质已经不满足队列的“先进先出”的性质。**deque 运行效率不如 vector**。

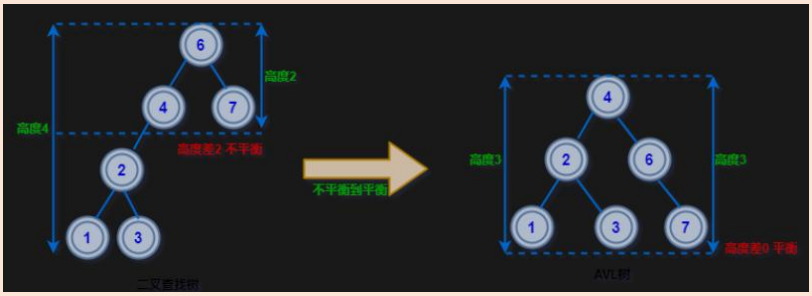
【deque 的内部原理】deque 使用的是**分段的连续存储空间**，能提供 **O(1)的查询效率**，当向 deque 中添加元素时，如果 deque 的空间快被占满，就**重新分配一段连续内存空间串接在 deque 的头端或者尾端**，然后**将新添加的元素放在新的内存空间上，不需要拷贝原有元素**（注意和 vector 的扩容对比），**避开了原来元素拷贝到新空间的损耗，代价是需要复杂的迭代器架构**。



stack 和 queue 是 deque 的容器适配器，即在 deque 的基础上封装而来：

请你说说红黑树的特性，为什么要有红黑树

高度平衡的二叉搜索树的查找效率也可以接近 O(logn)，但是其每次插入新元素时都可能会破坏平衡性（左右子树的高度差越来越大），进而需要通过左旋和右旋来进行调整，使之再次成为一颗符合要求的平衡树。显然在那种插入、删除很频繁的场景中，性能大打折扣：



为了解决这个问题，于是有了红黑树：

1、具有二叉查找树的特点；

2、每个节点都是**红色**或者**黑色**的一种；

3、**根节点是黑色，存储数据；叶节点是黑色，不存数据**

4、**任何相邻的节点都不能同时为红色，红色节点必须被黑色节点隔开**；

5、**从任意节点到每个叶节点的路径都包含相同数量的黑色节点**。

这些约束确保了红黑树的关键特性：**从根节点到叶节点的最长可能路径不多于最短可能路径的两倍。节点的路径长度决定着对节点的查询效率**，这样确保了最坏情况下的查找、插入、删除操作的时间复杂度不超过 O(log n)。

请你说说 unordered_map 的原理，为什么要有红黑树

unordered_map 底层数据结构采用**哈希表**，每次需要插入一个新元素时，首先调用**哈希函数**产生一个**哈**

希值，这个**哈希值**表明应该插入几号桶上，每个桶对应一个链表，新插入的元素就加入链表中。

哈希表极高的查询效率就是由其**哈希函数**来保证的，但是数据量非常大时，**不同的 key** 可能从哈希函数中**获得相同的哈希值**，这时就产生了**哈希冲突**。影响哈希冲突的影响因素：**数据总量**、**哈希表长**、**哈希函数**等。处理哈希冲突的解决方法有：

（1）**链式地址法**：每个哈希值对应一个链表，链表可以动态分配内存

（2）**直接定址法**：H=aHash(key)+b，只要空间足够大就不会发生冲突，空间效率很低

（3）**开放地址法**：H=(Hash(key)+d) mod m 其中 d 为一个增量序列，m 为哈希表长

C++的 heap 和 priority_queue 有什么区别？

heap 不属于 STL 的容器，它是 priority_queue(优先级队列)的幕后助手。**heap 建立在完全二叉树上**，分为**大根堆**和**小根堆**，它的特点是以任何顺序加入容器的元素，取出时一定从优先权最高的元素开始取。priority_queue 也不属于 STL 的**容器**，它是**容器适配器**。

你了解 pair 容器吗？它经常怎么使用？

Pair 是 STL 提供的用于关联式容器保存数据的类模板，它可以保存 2 个相关的值，**通过它的两个公有的数据成员 first 或者 second 来访问**，可以通过 **make_pair()函数**来**创建一个 pair 对象**。

它常见的用法是遍历关联式容器 map 和 unordered_map

请你说说迭代器失效原因，有哪些情况？怎么解决？

STL 中某些容器调用了某些成员方法后会导致迭代器失效。

1、**vector 容器**，如果**调用 reserve()来增加容器容量**，之前创建好的任何迭代器（例如开始迭代器和结束迭代器）都可能会失效，这是因为为了增加容器的容量，vector 容器的元素可能已经被复制或移到了新的内存地址。

2、对于**序列式容器**（例如 vector、deque），**调用 erase(iter)删除当前的迭代器，iter 之后的所有的迭代器都会失效**。这是因为序列式容器是**连续存储的一段空间**，所以当对其进行 erase 操作时，其后的每一个元素都会向前移一个位置。**解决办法：iter=erase（iter）//返回下一个有效的迭代器**。

3、对于**关联式容器**（例如 map、set），**调用 erase(iter)删除当前的迭代器，会使当前的迭代器 iter 失效**。这是因为关联式容器**使用了红黑树**来实现，插入、删除一个节点不会对其他元素造成影响。erase 迭代器只是被删元素的迭代器失效。**解决办法：采用 erase(iter++) //无返回值，自增**。