#### Audio多音频的播放与仲裁

1.负责apk的编写与pcm音频控制；

2.负责音频仲裁底层逻辑的编写；

3.在开发板中调试apk，修改bug；

#### Gallop项目优化

1.优化HU的代码，降低代码耦合，处理删除HU安装包前判断HU与TBox连接状态；

2.配合TSP新增三个通知状态并测试；

3.优化代码，修改部分代码在编译时产生的warning；

**第零节 常问问题**

**#00 什么是智能指针？智能指针有什么作用？分为哪几种？各自有什么样的特点？**

* 智能指针是一个RAII（资源获取就是初始化）类模型，用于动态分配内存，其设计思想是将基本类型指针封装为（模板）类对象指针，并在离开作用域时调用析构函数，使用delete删除指针所指向的内存空间。
* 智能指针的作用是，能够处理内存泄漏问题和空悬指针问题。
* 分为auto\_ptr、unique\_ptr、shared\_ptr和weak\_ptr四种，各自的特点：
* 对于auto\_ptr，实现独占式拥有的概念，同一时间只能有一个智能指针可以指向该对象；但auto\_ptr在C++11中被摒弃，其主要问题在于：
* 对象所有权的转移，比如在函数传参过程中，对象所有权不会返还，从而存在潜在的内存崩溃问题；
* 不能指向数组，也不能作为STL容器的成员。
* 对于unique\_ptr，实现独占式拥有的概念，同一时间只能有一个智能指针可以指向该对象，因为无法进行拷贝构造和拷贝赋值，但是可以进行移动构造和移动赋值；
* 对于shared\_ptr，实现共享式拥有的概念，即多个智能指针可以指向相同的对象，该对象及相关资源会在其所指对象不再使用之后，自动释放与对象相关的资源；
* 对于weak\_ptr，解决shared\_ptr相互引用时，两个指针的引用计数永远不会下降为0，从而导致死锁问题。而weak\_ptr是对对象的一种弱引用，可以绑定到shared\_ptr，但不会增加对象的引用计数。

**#01 const关键字有什么作用？**

* 修饰变量时，表示该变量的值在其生命周期内只读，不能被改变；
* 修饰指针：int \* const;
* 修饰指针所指向的对象：const int \*；
* 修饰引用所绑定的对象：const int &；
* 修饰函数的引用形参时，可以保护实参不被函数修改；
* 修饰非静态成员变量时，不能在类定义处初始化，必须通过构造函数初始化列表进行初始化；
* 修饰静态成员变量时，不能在类内部初始化，一般在类外部进行初始化；
* 修饰成员函数时，表示该函数不应修改费静态成员，但并不可靠，因为指针所指对象可能会被修改。

**#02 #define和const有什么区别？**

* 编译器处理方式不同：#define宏是在预处理阶段展开，不能对宏定义进行调试，而const常量是在编译阶段使用；
* 类型和安全检查不同：#define宏没有类型，不做任何类型检查，仅仅是代码展开，可能产生边际效应等错误，而const常量有具体类型，在编译阶段会执行类型检查；
* 存储方式不同：#define宏仅仅是代码展开，在多个地方进行字符串替换，不会分配内存，存储于程序的代码段中，而const常量会分配内存，但只维持一份拷贝，存储于程序的数据段中。
* 定义域不同：#define宏不受定义域限制，而const常量只在定义域内有效。

**#03 #define和inline有什么区别？**

* 宏是一种预处理功能；inline是C++的关键字；
* #define宏函数在预处理阶段展开，而inline内联函数在编译阶段展开；
* #define宏函数不检查参数类型，而inline内联函数检查参数类型，使用更加安全。

内联函数将函数体直接扩展到调用内联函数的地方，这样减少了参数压栈、跳转、返回等过程；

inline会向编译器提出内联请求，但是是否内联由编译器决定（当然可以通过设置编译器，强制使用内联）；

内联是一种优化，在某些情况下，即使没有显示的声明内联，比如定义在class内部的方法，编译器也可能将其作为内联函数；

inline函数不能过于复杂，最初C++限定不能有任何形式的循环，不能有过多的条件判断，不能对函数进行取地址操作等，但是现在编译器几乎没有任何限制，基本可以实现内联；

**#04 标准库中有哪些容器？分别有什么特点？**

标准库中的容器主要分为三类：顺序容器、关联容器、容器适配器。

* 顺序容器包括五种类型：
  + array<T, N>数组：固定大小数组，支持快速随机访问，但不能插入或删除元素；
  + vector<T>动态数组：支持快速随机访问，尾位插入和删除的速度很快；
  + deque<T>双向队列：支持快速随机访问，首尾位置插入和删除的速度很快；（可以看作是vector的增强版，与vector相比，可以快速地在首位插入和删除元素）
  + list<T>双向链表：只支持双向顺序访问，任何位置插入和删除的速度都很快；
  + forward\_list<T>单向链表：只支持单向顺序访问，任何位置插入和删除的速度都很快。
* 关联容器包含两种类型：
  + map容器：
    - map<K, T>关联数组：用于保存关键字-值对；
    - multimap<K, T>：关键字可重复出现的map；
    - unordered\_map<K, T>：用哈希函数组织的map；
    - unordered\_multimap<K, T>：关键词可重复出现的unordered\_map；
  + set容器：
    - set<T>：只保存关键字；
    - multiset<T>：关键字可重复出现的set；
    - unordered\_set<T>：用哈希函数组织的set；
    - unordered\_multiset<T>：关键词可重复出现的unordered\_set；
* 容器适配器包含三种类型：
  + stack<T>栈、queue<T>队列、priority\_queue<T>优先队列。

**#05 static关键字有什么作用？**

* 修饰局部变量时，使得该变量在静态存储区分配内存；只能在首次函数调用中进行首次初始化，之后的函数调用不再进行初始化；其生命周期与程序相同，但其作用域为局部作用域，并不能一直被访问；
* 修饰全局变量时，使得该变量在静态存储区分配内存；在声明该变量的整个文件中都是可见的，而在文件外是不可见的；
* 修饰函数时，在声明该函数的整个文件中都是可见的，而在文件外是不可见的，从而可以在多人协作时避免同名的函数冲突；
* 修饰成员变量时，所有的对象都只维持一份拷贝，可以实现不同对象间的数据共享；不需要实例化对象即可访问；不能在类内部初始化，一般在类外部初始化，并且初始化时不加static；
* 修饰成员函数时，该函数不接受this指针，只能访问类的静态成员；不需要实例化对象即可访问。

**#06 类模板**

* + 用于解决多个功能相同、数据类型不同的类需要重复定义的问题。
  + 在建立类时候使用template及任意类型标识符T，之后在建立类对象时，会指定实际的类型，这样才会是一个实际的对象。
  + 类模板是对一批仅数据成员类型不同的类的抽象，只要为这一批类创建一个类模板，即给出一套程序代码，就可以用来生成具体的类。

**第一节 语法基础**

**#00 C和C++有什么区别？**

* C++是面向对象的语言，而C是面向过程的语言；
* C++引入new/delete运算符，取代了C中的malloc/free库函数；
* C++引入引用的概念，而C中没有；
* C++引入类的概念，而C中没有；
* C++引入函数重载的特性，而C中没有。

**#01 C++11有哪些新特性？**

* Lambda表达式用于创建匿名函数：
* [capture](parameters)->return-type {body}
* 自动类型推导auto和decltype；
* 列表初始化；
* =default生成默认构造函数，=delete禁止使用拷贝构造函数；
* nullptr关键字，用于解决NULL的二义性；
* 基于RAII原则，引入shared\_ptr、unique\_ptr等智能指针；
* 右值引用，将引用绑定到右值，如临时对象或字面量；
* 引入线程库；
* 范围for循环。

**#02 struct和class有什么区别？**

* 成员的默认访问权限：
* struct的成员默认为public权限，class的成员默认为private权限；
* 默认继承权限：
* struct的继承按照public处理，class的继承按照private处理。

**#03 对于一个频繁使用的短小函数，应该使用什么来实现？有什么优缺点？**

应该使用inline内联函数，即编译器将inline内联函数内的代码替换到函数被调用的地方。

优点：

* 在内联函数被调用的地方进行代码展开，省去函数调用的时间，从而提高程序运行效率；
* 相比于宏函数，内联函数在代码展开时，编译器会进行语法安全检查或数据类型转换，使用更加安全；

缺点：

* 代码膨胀，产生更多的开销；
* 如果内联函数内代码块的执行时间比调用时间长得多，那么效率的提升并没有那么大；
* 如果修改内联函数，那么所有调用该函数的代码文件都需要重新编译；
* 内联声明只是建议，是否内联由编译器决定，所以实际并不可控。

**#04 #define和inline有什么区别？**

* 宏是一种预处理功能；inline是C++的关键字；
* #define宏函数在预处理阶段展开，而inline内联函数在编译阶段展开；
* #define宏函数不检查参数类型，而inline内联函数检查参数类型，使用更加安全。

内联函数将函数体直接扩展到调用内联函数的地方，这样减少了参数压栈、跳转、返回等过程；

inline会向编译器提出内联请求，但是是否内联由编译器决定（当然可以通过设置编译器，强制使用内联）；

内联是一种优化，在某些情况下，即使没有显示的声明内联，比如定义在class内部的方法，编译器也可能将其作为内联函数；

inline函数不能过于复杂，最初C++限定不能有任何形式的循环，不能有过多的条件判断，不能对函数进行取地址操作等，但是现在编译器几乎没有任何限制，基本可以实现内联；

**#05 const关键字有什么作用？**

* 修饰变量时，表示该变量的值在其生命周期内只读，不能被改变；
* 修饰指针：int \* const;
* 修饰指针所指向的对象：const int \*；
* 修饰引用所绑定的对象：const int &；
* 修饰函数的引用形参时，可以保护实参不被函数修改；
* 修饰非静态成员变量时，不能在类定义处初始化，必须通过构造函数初始化列表进行初始化；
* 修饰静态成员变量时，不能在类内部初始化，一般在类外部进行初始化；
* 修饰成员函数时，表示该函数不应修改非静态成员，但并不可靠，因为指针所指对象可能会被修改。

**#06 #define和const有什么区别？**

* 编译器处理方式不同：#define宏是在预处理阶段展开，不能对宏定义进行调试，而const常量是在编译阶段使用；
* 类型和安全检查不同：#define宏没有类型，不做任何类型检查，仅仅是代码展开，可能产生边际效应等错误，而const常量有具体类型，在编译阶段会执行类型检查；
* 存储方式不同：#define宏仅仅是代码展开，在多个地方进行字符串替换，不会分配内存，存储于程序的代码段中，而const常量会分配内存，但只维持一份拷贝，存储于程序的数据段中。
* 定义域不同：#define宏不受定义域限制，而const常量只在定义域内有效。

**#07 explicit关键字有什么作用？**

* 首先，可以用单个实参来调用的构造函数都定义了从形参类型到实参类型的隐式转换，这种转换往往都是非预期的，所以使用explicit关键字对构造函数进行修饰，从而避免由构造函数定义的隐式转换。

**#08 extern关键字有什么作用？**

* 当extern修饰变量或函数时，表示变量或函数的定义在其他文件中，提示编译器在其他模块中寻找其定义；
* 当extern C时，提示编译器在编译函数时按照C的规则去翻译相应的函数名，如果按照C++的规则，函数名会被翻译得变得面目全非，因为C++支持函数的重载。

同义：C++支持函数重载，C语言不支持重载，函数被C++编译器编译后在库中的名字与C语言不同，假定某个函数原型为：void foo(int x, int y);

在被C编译器编译后在库中的名字为\_foo，而C++编译器则会产生像：\_foo\_int\_int之类的名字。 **为了解决此类名字匹配的问题，C++提供了C链接交换指定符号 extern “C”；**

**#09 static关键字有什么作用？**

* 修饰局部变量时，使得该变量在静态存储区分配内存；只能在首次函数调用中进行首次初始化，之后的函数调用不再进行初始化；其生命周期与程序相同，但其作用域为局部作用域，并不能一直被访问；
* 修饰全局变量时，使得该变量在静态存储区分配内存；在声明该变量的整个文件中都是可见的，而在文件外是不可见的；
* 修饰函数时，在声明该函数的整个文件中都是可见的，而在文件外是不可见的，从而可以在多人协作时避免同名的函数冲突；
* 修饰成员变量时，所有的对象都只维持一份拷贝，可以实现不同对象间的数据共享；不需要实例化对象即可访问；不能在类内部初始化，一般在类外部初始化，并且初始化时不加static；
* 修饰成员函数时，该函数不接受this指针，只能访问类的静态成员；不需要实例化对象即可访问。

**#10 volatile关键字有什么作用？可以同时是const和volatile吗？指针可以是volatile吗？**

* 访问寄存器比访问内存要快，所以CPU会优先访问该数据在寄存器中的存储结果，但内存中的数据可能已经被意想不到地改变，而寄存器中还保留着原来的结果，所以为了避免这种情况，可以将变量声明为volatile，不进行编译优化，使得CPU每次都从内存中读取数据。
* 例子是并行设备的硬件寄存器、中断服务子程序会访问到的非自动变量、多线程应用中被几个任务共享的变量。
* 一个变量可以同时是const和volatile，一个例子是只读状态寄存器，是const表示程序不应该试图修改变量的值，是volatile表示变量的值可能被操作系统、硬件、其他线程等改变。
* 指针可以是volatile，一个例子是中断服务子程序修改一个指向buffer的指针。
* 补充：下面这段代码有什么错误？
* int square(volatile int \*ptr) {
* return (\*ptr \* \*ptr);
* }
* 因为\*ptr的值可能被意想不到地改变，所以两次解引用得到的值不一定相同，因此应该如下：
* int square(volatile int \*ptr) {
* int a = \*ptr;
* return a \* a;
* }

**#11 sizeof和strlen之间有什么区别？**

* sizeof属于运算符，不是库函数，其结果在编译时期计算得到，因此不能用来得到动态分配的内存大小，而strlen属于库函数，其结果在运行期间计算得到；
* sizeof参数可以是任何数据或数据类型，而strlen的参数只能是字符指针，且该指针指向结尾为\0的字符串。

**#12 assert有什么用处？**

* assert是一种仅在debug版本中使用的宏函数，用于检查不该发生的情况，可以看作是一种在任何系统状态下都可以安全使用的无害测试手段。
* 另外，可以通过#define NDEBUG来关闭assert（需要在<cassert>头文件之前）。

**# 13 auto关键字**

* 1. auto的作用是让编译器自动推断变量的类型，而不需要显式指定类型。这种隐式类型的推导发生在编译期。
* 2. auto并不能代表实际的类型声明，只是一个类型声明的“占位符”
* 3. auto声明的变量必须马上初始化，以让编译器推断出它的实际类型。

**#14 变量的声明和定义有什么区别？**

* 声明仅仅是把变量类型等信息提供给编译器，并不为其分配内存空间，而定义需要为变量分配内存空间；
* 变量可以在多处声明，如外部变量extern，但只能在一处定义。

**#15 指针和引用有什么区别？**

* 指针是一种对象，用来存放某个对象的地址，占用内存空间，而引用是一种别名，不占用内存空间；
* 指针可以声明为空，之后进行初始化，普通指针可以随时更换所指对象，而引用必须在声明的时候初始化，而且初始化后不可改变；
* 引用是安全的，而指针不是（引用比指针多了类型检查）；
* 引用具有更好的可读性和实用性；
* 指针包含指向常量的指针和常量指针，而引用不包含常量引用，但包含对常量的引用。

**#16 [识别指针、函数和数组] 以下声明语句分别代表什么意思？**

* void \*(\*(\*fp1)(int))[10];
* fp1是一个指针，指向一个函数，这个函数的参数为整型，返回一个指针，这个指针指向一个数组，数组中有10个元素，每个元素都是一个void \*指针。
* float (\*(\*fp2)(int, int, int))(int);
* fp2是一个指针，指向一个函数，这个函数的参数是三个整型，返回一个指针，这个指针指向一个函数，这个函数的参数是整型，返回是浮点型；
* int (\*(\*fp3)())[10]();
* fp3是一个指针，指向一个函数，这个函数返回一个指针，这个指针指向一个数组，数组中有10个元素，每个元素都是一个指针，指向一个函数，这个函数的参数为空，返回整型。

**#17 常量指针与指针常量**

* char greeting[] = "Hello";
* const char\* p2 = greeting; // 常量指针即常指针，指针指向的地址可以改变，但是所存的内容不能变
* char const\* p2 = greeting; // 与const char\* p2 等价
* char\* const p3 = greeting; // 指针常量，指针是一个常量，即指针指向的地址不能改变，但是指针所存的内容可以改变
* const char\* const p4 = greeting; // 指向常量的常指针，指针和指针所存的内容都不能改变，本质是一个常量

**#18 常对象、常指针、常引用**

* A b; // 普通对象，可以调用全部成员函数、更新常成员变量
* const A a; // 常对象，只能调用常成员函数
* const A \*p = &a; // 常指针
* const A &q = a; // 常引用

**第二节 内存管理**

**#00 常用的数据类型各自占用多大的内存空间？**

| **数据类型** | **32位编译器** | **64位编译器** |
| --- | --- | --- |
| **bool** | **1** | 1 |
| **char** | **1** | 1 |
| **short (int)** | **2** | 2 |
| **int** | **4** | 4 |
| **unsigned int** | **4** | 4 |
| **long** | **4** | **8** |
| **long long** | **8** | 8 |
| **float** | **4** | 4 |
| **double** | **8** | 8 |
| **pointer** | **4** | **8** |

**#01 new/delete和malloc/free之间有什么关系？**

* 相同点：
  + 对于内部数据类型来说，没有构造与析构的过程，所以两者是等价的，都可以用于申请动态内存和释放内存；
* 不同点：
  + new/delete可以调用对象的构造函数和析构函数，属于运算符，在编译器权限之内；
  + malloc/free仅用于内存分配和释放，属于库函数，不在编译器权限之内；
  + new是类型安全的，而malloc返回的数据类型是void \*，所以要显式地进行类型转换；
  + new可以自动计算所需字节数，而malloc需要手动计算；
  + new申请内存失败时抛出bad\_malloc异常，而malloc返回空指针。

**#02 delete与delete []有什么区别？**

* 对于简单类型来说，使用new分配后，不管是数组数组还是非数组形式，两种方式都可以释放内存：
* int \*a = new int(1);
* delete a;
* int \*b = new int(2);
* delete [] b;
* int \*c = new int[11];
* delete c;
* int \*d = new int[12];

delete [] d;

* 对于自定义类型来说，就需要对于单个对象使用delete，对于对象数组使用delete []，逐个调用数组中对象的析构函数，从而释放所有内存；

如果反过来使用，即对于单个对象使用delete []，对于对象数组使用delete，其行为是未定义的；

* 所以，最恰当的方式就是如果用了new，就用delete；如果用了new []，就用delete []。

**#03 如果在申请动态内存时找不到足够大的内存块，即malloc和new返回空指针，那么应该如何处理这种情况？**

* 对于malloc来说，需要判断其是否返回空指针，如果是则马上用return语句终止该函数或者exit终止该程序；
* 对于new来说，默认抛出异常，所以可以使用try...catch...代码块的方式：
* try {
* int \*ptr = new int[10000000];
* } catch(bad\_alloc &memExp) {
* cerr << memExp.what() << endl;
* }
* 还可以使用set\_new\_handler函数的方式：
* void no\_more\_memory() {
* cerr << "Unable to satisfy request for memory" << endl;
* abort();
* }
* int main() {
* set\_new\_handler(no\_more\_memory);
* int \*ptr = new int[10000000];
* }
* 在这种方式里，如果new不能满足内存分配请求，no\_more\_memory会被反复调用，所以new\_handler函数必须完成以下事情：
  + 让更多内存可被使用：可以在程序一开始执行就分配一大块内存，之后当new\_handler第一次被调用，就将这些内存释放还给程序使用；
  + 使用另一个new\_handler；
  + 卸除new\_handler：返回空指针，这样new就会抛出异常；
  + 直接抛出bad\_alloc异常；
  + 调用abort或exit。

**#04 内存泄漏的场景有哪些？如何判断和定位内存泄漏？如何避免内存泄漏？**

**内存泄漏的场景：**

* 1. **malloc和free未成对出现；new/new []和delete/delete []未成对出现**；
  + 在堆中创建对象分配内存，但未显式释放内存；比如，通过局部分配的内存，未在调用者函数体内释放：
  + char\* getMemory() {
  + char \*p = (char \*)malloc(30);
  + return p;
  + }
  + int main() {
  + char \*p = getMemory();
  + return 0;
  + }
  + 在构造函数中动态分配内存，但未在析构函数中正确释放内存；
* 2. **未定义拷贝构造函数或未重载赋值运算符，从而造成两次释放相同内存的做法**；比如，类中包含指针成员变量，在未定义拷贝构造函数或未重载赋值运算符的情况下，编译器会调用默认的拷贝构造函数或赋值运算符，以逐个成员拷贝的方式来复制指针成员变量，使得两个对象包含指向同一内存空间的指针，那么在释放第一个对象时，析构函数释放该指针指向的内存空间，在释放第二个对象时，析构函数就会释放同一内存空间，这样的行为是错误的；
* 3. **没有将基类的析构函数定义为虚函数**。

**判断和定位内存泄漏的方法：**

在Linux系统下，可以使用valgrind、mtrace等内存泄漏检测工具。

**避免内存泄露的方法：**

1 使用的时候记得指针的长度；

* 2 malloc与new记得free或delete；
* 3 对指针赋值的时候应该注意被赋值指针需不需要释放；
* 4 动态分配内存的指针最好不要再次赋值；
* 5 在C++中应该优先考虑使用智能指针；

**#05 内存的分配方式有几种？**

* 在栈上分配：在执行函数时，局部变量的内存都可以在栈上分配，函数结束时会自动释放；栈内存的分配运算内置于处理器的指令集中，效率很高，但分配的内存容量有限；
* 从堆上分配：由new分配/delete释放的内存块，也称为动态内存分配，程序员自行申请和释放内存，使用灵活；
* 从自由存储区分配：由malloc分配/free释放的内存块，与堆类似；
* 从常量存储区分配：特殊的存储区，存放的是常量，不可修改；
* 从全局/静态存储区分配：编译期间分配内存，整个程序运行期间都存在，如全局变量、静态变量等。

**#06 堆和栈有什么区别？**

* 分配和管理方式不同：
  + 堆是动态分配的，其空间的分配和释放都由程序员控制；
  + 栈是由编译器自动管理的，其分配方式有两种：静态分配由编译器完成，比如局部变量的分配；动态分配由alloca()函数进行分配，但是会由编译器释放；
* 产生碎片不同：
  + 对堆来说，频繁使用new/delete或者malloc/free会造成内存空间的不连续，产生大量碎片，是程序效率降低；
  + 对栈来说，不存在碎片问题，因为栈具有先进后出的特性；
* 生长方向不同：
  + 堆是向着内存地址增加的方向增长的，从内存的低地址向高地址方向增长；
  + 栈是向着内存地址减小的方向增长的，从内存的高地址向低地址方向增长；
* 申请大小限制不同：
  + 栈顶和栈底是预设好的，大小固定；
  + 堆是不连续的内存区域，其大小可以灵活调整。

**#07 C++内存区域**

* 栈Stack，就是那些由编译器在需要的时候分配，在不需要的时候自动清除的变量的储存区。里面的变量通常是局部变量、函数参数等；
* 堆Heap，就是那些由new分配的内存块，它们的释放编译器不管，又我们的应用程序去控制，一般一个new需要对应一个delete。如果没有释放掉，那么程序结束后，操作系统会自动回收；
* 自由储存区，就是由malloc分配内存块，他和堆是十分相似的，不过它用free来释放；
* 全局/静态储存区，全局变量和静态变量被分配到同一块内存中，在以前的C语言中，全局变量又分为初始化和未初始化的，在C++中没有这个区分了，它们共同占用一块内存区；
* 常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量，不允许修改；

**#08 静态内存分配和动态内存分配有什么区别？**

* 静态内存分配是在编译时期完成的，不占用CPU资源；动态内存分配是在运行时期完成的，分配和释放需要占用CPU资源；
* 静态内存分配是在栈上分配的；动态内存分配是在堆上分配的；
* 静态内存分配不需要指针或引用类型的支持；动态内存分配需要；
* 静态内存分配是按计划分配的，在编译前确定内存块的大小；动态内存分配是按需要分配的；
* 静态内存分配是把内存的控制权交给了编译器；动态内存分配是把内存的控制权给了程序员；
* 静态内存分配的运行效率比动态内存分配高，动态内存分配不当可能造成内存泄漏。

**#09 如何构造一个类，使得只能在堆上或只能在栈上分配内存？**

* 只能在堆上分配内存：将析构函数声明为private；
* 只能在栈上生成对象：将new和delete重载为private。

**#10 浅拷贝和深拷贝有什么区别？**

* 浅拷贝只复制指向某个对象的指针，而不复制对象本身，新旧对象还是共享一块内存；而深拷贝会创造一个相同的对象，新对象与原对象不共享内存，修改新对象不会影响原对象。

**#11 字节对齐的原则是什么？**

* 从偏移为0的位置开始存储；
* 如果没有定义#pragma pack(n)：
  + sizeof的最终结果必然是结构内部最大成员的整数倍，不够补齐；
  + 结构内部各个成员的首地址必然是自身大小的整数倍；
* 如果定义了#pragma pack(n)：
  + sizeof的最终结果必然必然是min[n,结构内部最大成员]的整数倍，不够补齐；
  + 结构内部各个成员的首地址必然是min[n,自身大小]的整数倍。

**第三节 面向对象**

**#00 面向对象的三大特征是哪些？各自有什么样的特点？**

* 封装：将客观事物封装成抽象的类，而类可以把自己的数据和方法暴露给可信的类或者对象，对不可信的类或对象则进行信息隐藏。
* 继承：可以使用现有类的所有功能，并且无需重新编写原来的类即可对功能进行拓展；
* 多态：一个类实例的相同方法在不同情形下有不同的表现形式，使不同内部结构的对象可以共享相同的外部接口。

**#01 多态的实现有哪几种？**

* 多态分为静态多态和动态多态。其中，静态多态是通过重载和模板技术实现的，在编译期间确定；动态多态是通过虚函数和继承关系实现的，执行动态绑定，在运行期间确定。

**#02 动态多态有什么作用？有哪些必要条件？**

动态多态的作用：

* 隐藏实现细节，使代码模块化，提高代码的可复用性；
* 接口重用，使派生类的功能可以被基类的指针/引用所调用，即向后兼容，提高代码的可扩充性和可维护性。

动态多态的必要条件：

* 需要有继承；
* 需要有虚函数覆盖；
* 需要有基类指针/引用指向子类对象。

**#03 动态绑定是如何实现的？**

* 当编译器发现类中有虚函数时，会创建一张虚函数表，把虚函数的函数入口地址放到虚函数表中，并且在对象中增加一个指针vptr，用于指向类的虚函数表。当派生类覆盖基类的虚函数时，会将虚函数表中对应的指针进行替换，从而调用派生类中覆盖后的虚函数，从而实现动态绑定。

**#04 纯虚函数有什么作用？如何实现？**

* 定义纯虚函数是为了实现一个接口，起到规范的作用，想要继承这个类就必须覆盖该函数。
* 实现方式是在虚函数声明的结尾加上= 0即可。

**#05 虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的？**

* 虚函数表是针对类的，类的所有对象共享这个类的虚函数表，因为每个对象内部都保存一个指向该类虚函数表的指针vptr，每个对象的vptr的存放地址都不同，但都指向同一虚函数表。

**#06 为什么基类的构造函数不能定义为虚函数？**

* 虚函数的调用依赖于虚函数表，而指向虚函数表的指针vptr需要在构造函数中进行初始化，所以无法调用定义为虚函数的构造函数。

**#07 为什么基类的析构函数需要定义为虚函数？**

* 为了实现动态绑定，基类指针指向派生类对象，如果析构函数不是虚函数，那么在对象销毁时，就会调用基类的析构函数，只能销毁派生类对象中的部分数据，所以必须将析构函数定义为虚函数，从而在对象销毁时，调用派生类的析构函数，从而销毁派生类对象中的所有数据。

**#08 构造函数和析构函数能抛出异常吗？**

* 从语法的角度来说，构造函数可以抛出异常，但从逻辑和风险控制的角度来说，尽量不要抛出异常，否则可能导致内存泄漏。
* 析构函数不可以抛出异常，如果析构函数抛出异常，则异常点之后的程序，比如释放内存等操作，就不会被执行，从而造成内存泄露的问题；而且当异常发生时，C++通常会调用对象的析构函数来释放资源，如果此时析构函数也抛出异常，即前一个异常未处理又出现了新的异常，从而造成程序崩溃的问题。

**#09 如何让一个类不能实例化？**

将类定义为抽象类（也就是存在纯虚函数）或者将构造函数声明为private。

**#10 多继承存在什么问题？如何消除多继承中的二义性？**

1. 增加程序的复杂度，使得程序的编写和维护比较困难，容易出错；
2. 在继承时，基类之间或基类与派生类之间发生成员同名时，将出现对成员访问的不确定性，即同名二义性；

消除同名二义性的方法：

* + 利用作用域运算符::，用于限定派生类使用的是哪个基类的成员；
  + 在派生类中定义同名成员，覆盖基类中的相关成员；

1. 当派生类从多个基类派生，而这些基类又从同一个基类派生，则在访问此共同基类的成员时，将产生另一种不确定性，即路径二义性；

消除路径二义性的方法：

* + 消除同名二义性的两种方法都可以；
  + 使用虚继承，使得不同路径继承来的同名成员在内存中只有一份拷贝。

**#11 如果类A是一个空类，那么sizeof(A)的值为多少？为什么？**

* sizeof(A)的值为1，因为编译器需要区分这个空类的不同实例，分配一个字节，可以使这个空类的不同实例拥有独一无二的地址。

**第四节 高级特性**

**#00 什么是智能指针？智能指针有什么作用？分为哪几种？各自有什么样的特点？**

* 智能指针是一个RAII（资源获取就是初始化）类模型，用于动态分配内存，其设计思想是将基本类型指针封装为（模板）类对象指针，并在离开作用域时调用析构函数，使用delete删除指针所指向的内存空间。
* 智能指针的作用是，能够处理内存泄漏问题和空悬指针问题。
* 分为auto\_ptr、unique\_ptr、shared\_ptr和weak\_ptr四种，各自的特点：
* 对于auto\_ptr，实现独占式拥有的概念，同一时间只能有一个智能指针可以指向该对象；但auto\_ptr在C++11中被摒弃，其主要问题在于：
* 对象所有权的转移，比如在函数传参过程中，对象所有权不会返还，从而存在潜在的内存崩溃问题；
* 不能指向数组，也不能作为STL容器的成员。
* 对于unique\_ptr，实现独占式拥有的概念，同一时间只能有一个智能指针可以指向该对象，因为无法进行拷贝构造和拷贝赋值，但是可以进行移动构造和移动赋值；
* 对于shared\_ptr，实现共享式拥有的概念，即多个智能指针可以指向相同的对象，该对象及相关资源会在其所指对象不再使用之后，自动释放与对象相关的资源；
* 对于weak\_ptr，解决shared\_ptr相互引用时，两个指针的引用计数永远不会下降为0，从而导致死锁问题。而weak\_ptr是对对象的一种弱引用，可以绑定到shared\_ptr，但不会增加对象的引用计数。

**#01 shared\_ptr是如何实现的？**

* 构造函数中计数初始化为1；
* 拷贝构造函数中计数值加1；
* 赋值运算符中，左边的对象引用计数减1，右边的对象引用计数加1；
* 析构函数中引用计数减1；
* 在赋值运算符和析构函数中，如果减1后为0，则调用delete释放对象。

**#02 类型转换分为哪几种？各自有什么样的特点？**

* static\_cast：用于基本数据类型之间的转换、子类向父类的安全转换、void\*和其他类型指针之间的转换；
* const\_cast：用于去除const或volatile属性；
* dynamic\_cast：用于子类和父类之间的安全转换，可以实现向上向下转换，因为编译器默认向上转换总是安全的，而向下转换时，dynamic\_cast具有类型检查的功能；
* dynamic\_cast转换失败时，对于指针会返回目标类型的nullptr，对于引用会返回bad\_cast异常；
* reinterpret\_cast：用于不同类型指针之间、不同类型引用之间、指针和能容纳指针的整数类型之间的转换。

**#03 RTTI是什么？其原理是什么？**

* RTTI即运行时类型识别，其功能由两个运算符实现：
* typeid运算符，用于返回表达式的类型，可以通过基类的指针获取派生类的数据类型；
* dynamic\_cast运算符，具有类型检查的功能，用于将基类的指针或引用安全地转换成派生类的指针或引用。

**#04 右值引用有什么作用？**

* 右值引用的主要目的是为了实现转移语义和完美转发，消除两个对象交互时不必要的对象拷贝，也能够更加简洁明确地定义泛型函数。

**#05 标准库中有哪些容器？分别有什么特点？**

标准库中的容器主要分为三类：顺序容器、关联容器、容器适配器。

* 顺序容器包括五种类型：
  + array<T, N>数组：固定大小数组，支持快速随机访问，但不能插入或删除元素；
  + vector<T>动态数组：支持快速随机访问，尾位插入和删除的速度很快；
  + deque<T>双向队列：支持快速随机访问，首尾位置插入和删除的速度很快；（可以看作是vector的增强版，与vector相比，可以快速地在首位插入和删除元素）
  + list<T>双向链表：只支持双向顺序访问，任何位置插入和删除的速度都很快；
  + forward\_list<T>单向链表：只支持单向顺序访问，任何位置插入和删除的速度都很快。
* 关联容器包含两种类型：
  + map容器：
    - map<K, T>关联数组：用于保存关键字-值对；
    - multimap<K, T>：关键字可重复出现的map；
    - unordered\_map<K, T>：用哈希函数组织的map；
    - unordered\_multimap<K, T>：关键词可重复出现的unordered\_map；
  + set容器：
    - set<T>：只保存关键字；
    - multiset<T>：关键字可重复出现的set；
    - unordered\_set<T>：用哈希函数组织的set；
    - unordered\_multiset<T>：关键词可重复出现的unordered\_set；
* 容器适配器包含三种类型：
  + stack<T>栈、queue<T>队列、priority\_queue<T>优先队列。

**#06 vector的reserve()和resize()方法之间有什么区别？**

* 首先，vector的容量capacity()是指在不分配更多内存的情况下可以保存的最多元素个数，而vector的大小size()是指实际包含的元素个数；
* 其次，vector的reserve(n)方法只改变vector的容量，如果当前容量小于n，则重新分配内存空间，调整容量为n；如果当前容量大于等于n，则无操作；
* 最后，vector的resize(n)方法改变vector的大小，如果当前容量小于n，则调整容量为n，同时将其全部元素填充为初始值；如果当前容量大于等于n，则不调整容量，只将其前n个元素填充为初始值。

**#07 静态链接和动态链接有什么区别？**

* 静态链接是在编译链接时直接将需要的执行代码拷贝到调用处；
* 优点在于程序在发布时不需要依赖库，可以独立执行，缺点在于程序的体积会相对较大，而且如果静态库更新之后，所有可执行文件需要重新链接；
* 动态链接是在编译时不直接拷贝执行代码，而是通过记录一系列符号和参数，在程序运行或加载时将这些信息传递给操作系统，操作系统负责将需要的动态库加载到内存中，然后程序在运行到指定代码时，在共享执行内存中寻找已经加载的动态库可执行代码，实现运行时链接；
* 优点在于多个程序可以共享同一个动态库，节省资源；
* 缺点在于由于运行时加载，可能影响程序的前期执行性能。

**#08 悬挂指针与野指针有什么区别？**

* 悬挂指针：当指针所指向的对象被释放，但是该指针没有任何改变，以至于其仍然指向已经被回收的内存地址，这种情况下该指针被称为悬挂指针；
* 野指针：未初始化的指针被称为野指针。

**#09 拷贝构造函数和赋值运算符重载之间有什么区别？**

* 拷贝构造函数用于构造新的对象；
* Student s;
* Student s1 = s; // 隐式调用拷贝构造函数
* Student s2(s); // 显式调用拷贝构造函数
* 赋值运算符重载用于将源对象的内容拷贝到目标对象中，而且若源对象中包含未释放的内存需要先将其释放；
* Student s;
* Student s1;
* s1 = s; // 使用赋值运算符
* 一般情况下，类中包含指针变量时需要重载拷贝构造函数、赋值运算符和析构函数。

**#10 覆盖和重载之间有什么区别？**

* 覆盖是指派生类中重新定义的函数，其函数名、参数列表、返回类型与父类完全相同，只是函数体存在区别；覆盖只发生在类的成员函数中；
* 重载是指两个函数具有相同的函数名，不同的参数列表，不关心返回值；当调用函数时，根据传递的参数列表来判断调用哪个函数；重载可以是类的成员函数，也可以是普通函数。