1.函数

1.1 sizeof与strlen区别

1. sizeof是一个操作符，strlen是库函数

2. sizeof的参数可以是数据的类型，也可以变量，strlen只能以结尾为'\0'的字符串做参数

3. 编译器在编译时就计算出了sizeof的结果，而Strlen函数必须在运行时才能计算出来。sizeof计算的是数据类型占内存的大小，strlen计算的是字符串实际的长度

4. 数组做sizeof的参数不退化，传递给strlen就退化为指针了

空类的大小：一个空类对象的大小是1byte，这是被编译器安插进去的一个字节，这样就使得这个空类的两个实例得以在内存中配置独一无二的地址

1.2 strcpy sprintf memcpy

strcpy:主要实现字符串变量间的拷贝

sprintf:主要实现其他数据类型格式到字符串的转化

memcpy:主要是内存块间的拷贝

区别：

1. 操作对象不同：strcpy的两个操作对象均为字符串，sprintf的操作源对象可以使多种数据类型，目的操作对象是字符串，memcpy的两个对象就是两个任意可操作性的内存地址，并不限于何种数据类型

2. 执行效率不同，memcpy最高，Strcpy次之，sprintf的效率最低

1.3 new/delete 与 malloc/free

1. malloc与free标准库函数，new/delete是c++的运算符。他们都可以用于申请动态内存和释放内存

2. 对于非内部数据类型的对象而言，光用malloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free

3. 因此c++语言需要一个能完成动态内存分配和初始化工作的运算符new，和一个能完成清理与释放内存工作的运算符delete。

4. c++程序要经常调用c函数，而c程序只能用malloc/free管理动态内存

总结:

- new 是个操作符,和什么”+”,”-“,”=”…有一样的地位；malloc是个分配内存的函数,供你调用的.
- new是保留字,不需要头文件支持；malloc需要头文件库函数支持.
- new 建立的是一个对象；malloc分配的是一块内存.
- new建立的对象你可以把它当成一个普通的对象,用成员函数访问,不要直接访问它的地址空间；malloc分配的是一块内存区域,可以用指针访问,而且还可以在里面移动指针.

malloc()以及free()的机制

操作系统中有一个记录空闲内存地址的链表。 当操作系统收到程序的malloc申请时， 就会遍历该链表， 然后就寻找第一个空间大于所申请空间的堆结点， 然后就将该结点从空闲结点链表中删除， 并将该结点的空间分配给程序。

free:释放ptr指向的存储空间。 被释放的空间通常被送入可用存储区池， 以后可在调用malloc、 realloc以及realloc函数来再分配。

事实上， malloc()申请的时候实际上占用的内存要比申请的大。 因为超出的空间是用来记录对这块内存的管理信息。

1.4 ++i和i++的区别

理论上++i更快，实际与编译器优化有关，通常几乎无差别

```
//i++实现代码
int operator++(int)
{
    int temp = *this;
    ++*this;
    return temp;
}//返回一个int型的对象本身

//++i实现代码为
int& operator++()
{
    *this += 1;
    return *this;
}//返回一个int型的对象引用
```

2. 常见问题

2.1 数组名和指针的区别

指针是一个变量，有自己对应的存储空间，而数组名仅仅是一个符号，不是变量，因为没有自己对应的存储空间

1. 地址相同，大小不同

```
int arr[10];
int*p=arr;
cout<<arr<<endl;
cout<<p<<endl;
cout<<sizeof(arr)<<endl;//结果为40
cout<<sizeof(p)<<endl;//结果为4
```

1. 都可以用指针作为形参
2. 指针可以自加，数组名不可以
3. 作为参数的数组名的大小和指针的大小相同

2.2 指针和引用的区别

1. 引用必须被初始化，但是不分配存储空间，引用不占内存！ 指针不必在声明的时候初始化，在初始化的时候需要分配存储空间
2. 引用初始化后不能被改变，指针可以改变所指的对象
3. 不存在指向空值的引用，但是存在指向空值的指针

2.3 构造函数能否为虚函数

多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，**可以通过指向基类的指针**，来调用实现派生类中的方法

构造函数不能是虚函数：

虚函数主要是为了提供对多态的支持，也就是当处理一个对象，其动态类型和静态类型（编译时确定）不同时，可以提供对象的合适实际类型，而不是对象的静态类型。对于构造函数来讲，当创建一个对象时，静态类型和实际目标类型始终是一致的。

1. 虚函数对应一个vtable，这个vtable存储在对象的内存空间。如果构造函数是虚的，就要通过vtable来调用，可是对象还没有实例化，也就是内存空间还没有，怎么找vtable呢？所以不能是虚函数

2. 虚函数主要用于在信息不全的情况下，能使重写的函数得到对应的调用。构造函数本身就是要初始化实例，那使用虚函数也没有实际意义，所以构造函数没必要是虚函数

2.4 析构函数是虚函数

析构函数也可以是纯虚函数，但纯虚析构函数必须有定义体，因为析构函数的调用是在子类中隐含的

拓展：基类需要定义虚析构函数

规定：当derived class经由一个base class指针被删除而该base class的析构函数为non-virtual时，将发生未定义行为，通常将发生资源泄漏

解决办法：为多态基类声明一个virtual析构函数

virtual函数不能声明为内联

inline是编译期决定，意味着在执行前就将调用动作替换为被调用函数的本体

virtual是运行期决定的，意味着直到运行期菜决定调用哪个函数

这两者之间通常是冲突的

2.4 c语言编译全过程

编译的概念：编译程序读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令切换为功能等效的汇编代码，再由汇编程序转换为机器语言，并且按照操作系统对可执行文件格式的要求链接生成可执行程序。

编译的完整过程：c源程序 - 预编译处理(.c) - 编译优化程序(.s .asm) - 汇编程序(.obj .o .a .ko) - 链接程序(.exe .elf .axf)

2.4.1 编译预处理

读取C源程序，对其中的伪指令(#开头的指令)和特殊符号进行处理

伪指令主要包括以下四个方面：宏定义指令，条件编译指令，头文件包含指令，特殊符号

2.4.2 编译优化阶段

2.4.3 汇编过程

2.4.4 链接程序

2.5 返回引用的格式、好处、注意事项

格式：&

好处：在内存中不产生被返回值的副本（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生runtime error）

注意事项：

- 不能返回局部变量的引用。因为局部变量会在函数返回后被销毁，因此被返回的引用就成为了“无所指”的引用，程序会进入未知状态
- 不能返回函数内部new分配的内存的引用。如果被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由new分配）就无法释放，造成内存泄漏
- 可以返回类成员的引用，但最好是const

2.6 指针的表示

1. 一个整型数 int a
2. 一个指向整型数的指针 int *a
3. 一个指向指针的指针，它指向的指针是指向一个整型数 int **a
4. 一个有10个整型数的数组 int a[10]
5. 一个有10个指针的数组，该指针是指向一个整型数的 int *a[10]
6. 一个指向有10个整型数数组的指针 int (*a)[10]
7. 一个指向函数的指针，该函数有一个整型参数并返回一个整型数 int (*a)(int)
8. 一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整形 int (*a[10])(int)

2.7 拷贝构造函数的调用时机

1. 当类的一个对象去初始化该类的另一个对象时
2. 如果函数的形参是类的对象，调用函数进行形参和实参结合时
3. 如果函数的返回值是类对象，函数调用完成返回时

2.8 如何确保对象在抛出异常时也能被删除，什么是RAII

总的思想是RAII：设计一个class，令他的构造函数和析构函数分别获取和释放资源

有两个方法：1.利用“函数的局部对象无论函数以何种方式结束都会被析构”这一特性，将“一定要释放的资源”放进局部对象的析构函数

2.使用智能指针

2.9 在构造函数和析构函数中抛出异常会发生什么？什么是栈展开?

构造函数：

1. 构造函数中抛出异常，会导致析构函数不能被调用，但对象本身已申请到的内存资源会被系统释放（已申请到资源的内部成员变量会被系统依次逆序调用其析构函数）。
2. 因为析构函数不能被调用，所以可能会造成内存泄露或系统资源未被释放。
3. 构造函数中可以抛出异常，但必须保证在构造函数抛出异常之前，把系统资源释放掉，防止内存泄露。

析构函数：

1. 不要在析构函数中抛出异常！虽然C++并不禁止析构函数抛出异常，但这样会导致程序过早结束或出现不明确的行为。
2. 如果某个操作可能会抛出异常，class应提供一个普通函数（而非析构函数），来执行该操作。目的是给客户一个处理错误的机会。
3. 如果析构函数中异常非抛不可，那就用try catch来将异常吞下，但这样方法并不好，我们提倡有错早些报出来。

堆栈展开：

抛出异常时，将暂停当前函数的执行，开始查找匹配的catch子句。首先检查throw本身是否在try块内部如果是，检查与该try相关的catch子句，看是否可以处理该异常。如果不能处理，就退出当前函数，并且释放当前函数的内存并销毁局部对象，继续到上层的调用函数中查找，直到找到一个可以处理该异常的catch。

3.STL

3.1vector 增减元素对vector的影响

1. 对于连续内存容器，如vector、deque等，增减元素均会使得当前之后的所有迭代器失效。因此，以删除元素为例：由于erase()总是指向被删除元素的下一个元素的有效迭代器，因此，可以利用该连续内存容器的成员erase()函数的返回值。

```
for(auto iter = myvec.begin(); iter != myvec.end()) //另外注意这里用 "!=" 而非 "<"
{
    if(delete iter)
        iter = myvec.erase(iter);
    else
        ++iter;
}
```

还有两种极端的情况是：

(1)、vector插入元素时位置过于靠前，导致需要后移的元素太多，因此vector增加元素建议使用push_back而非insert；

(2)、当增加元素后整个vector的大小超过了预设，这时会导致vector重新分分配内存，效率极低。因此习惯的编程方法为：在声明了一个vector后，立即调用reserve函数，令vector可以动态扩容。通常vector是按照之前大小的2倍来增长的。

1. 对于非连续内存容器，如set、map等。增减元素只会使得当前迭代器无效。仍以删除元素为例，由于删除元素后，erase()返回的迭代器将是无效的迭代器。因此，需要在调用erase()之前，就使得迭代器指向删除元素的下一个元素。常见的编程写法为：

```
for(auto iter = myset.begin(); iter != myset.end()) //另外注意这里用 "!=" 而非 "<"
{
    if(delete iter)
        myset.erase(iter++); //使用一个后置自增就OK了
    else
        ++iter;
}
```

3.2 排序算法的实现

sort()，在数据量大时，采用quicksort，分段递归排序；一旦分段后的数量小于某个门限值，改用insertion sort，避免quicksort深度递归带来的过大的额外负担，如果递归层次过深，还会改用heapsort

4.c和c++的区别

1. 标准：分别隶属于两个不同的标准委员会
2. 语言本身：

- c++是面向对象语言，c是面向过程语言
- 结构：c以结构体struct为核心结构；c++以类class为核心结构
- 多态：c可以以宏定义的方式“自定义”部分地支持多态；c++自身提供多态，并以模板template支持编译器多态，以虚函数virtual function支持运行期多态
- 头文件的调用：c++用< >代替 “” 代表系统头文件，且复用c的头文件时，去掉 “.h"在开头加上"C"
- 输出输出：c++输入和输出都是在流对象上的操作
- 封装：c中的封装由于struct的特性全部为公有封装，C++中的封装由于class的特性更加完善 安全
- 常见风格：c中常用宏定义来进行文本替换，不具有类型安全性；c++中常建议采用常量定义，具有类型安全性
- 常用语言/库特性：

数组：c采用内建数组，c++ vector

字符串：c风格字符串string(实际为字符串数组)，c++ string

内存分配：malloc/free new/delete

指针：C中原生指针，由于常出现程序员在申请后忘记释放造成资源泄漏的问题，c++98中加入第一代基于引用技术的智能指针auto_ptr，后来加入了shared_ptr weak_ptr unique_ptr

5.仅有c++才有的常用特性

- 语言(范式)特性：

1. 面向对象编程：C++中以关键字class和多态特性支持的一种编程范式；
2. 泛型编程：C++中以关键字template支持的一种编程范式；
3. 模板元编程 ：C++中以模板特化和模板递归调用机制支持的一种编程范式。

4. C++中以对象和类型作为整个程序的核心，在对象方面，时刻注意对象创建和析构的成本，例如有一个很常用的(具名)返回值优化((N)RVO);在类型方面，有运行时类型信息(RTTI)等技术作为C++类型技术的支撑。

5. 函数重载：C++允许拥有不同变量但具有相同函数名的函数(函数重载的编译器实现方式、函数重载和(主)模板特化的区别都曾考过)。

6. 异常：以catch、throw、try等关键字支持的一种机制。

7. 名字空间：namespace，可以避免和减少命名冲突且让代码具有更强的可读性。

8. 谓词用法：通常以bool函数或仿函数(functor)或lambda函数的形式，出现在STL的大多数算法的第三个元素。

- 常见关键字(操作符)特性:

1. auto：在C中，auto代表自动类型通常都可省略；而在C++11新标准中，则起到一种“动态类型”的作用——通常在自动类型推导和decltype搭配使用。

2. 空指针：在C中常以NULL代表空指针，在C++中根据新标准用nullptr来代表空指针。

3. &: 在C中仅代表取某个左值(lvalue)的地址，在C++中还可以表示引用(别名)。

4. &&：在C中仅能表示逻辑与，在C++中还可以表示右值引用。

5. []：在C中仅能表示下标操作符，在C++中还可以表示lambda函数的捕捉列表。

6. {}：在C中仅能用于数组的初始化，在C++中由于引入了初始化列表(initializer_list)，可用于任何类型、容器等的初始化。

7. 常量定义：C中常以define来定义常量，C++中用const来定义运行期常量，用constexpr来定义编译器常量。

- 常用新特性:

1. 右值引用和move语义(太多内容，建议自查)。

2. 基于范围的for循环(与python中的写法类似，常用于容器)。

3. 基于auto——decltype的自动类型推导。

4. lambda函数(一种局部、匿名函数，高效方便地出现在需要局部、匿名语义的地方)。

5. 标准规范后的多线程库。

6.c++转换机制(static_cast dynamic_cast reinpreter_cast const_cast)

1.static_cast:

用法：static_cast<type-id>(expression)

该运算符把expression转换为type-id类型，但没有运行时类型检查来保证转换的安全性。主要有如下几种用法：

- 用于类层次结构中基类和子类之间指针或引用的转换：进行上行转换（把子类的指针或引用转换成基类的）是安全的；下行转换（把基类指针或引用转换成子类表示）时，由于没有动态类型检查，是不安全的
- 用于基本数据类型之间的转换，如把int转换成char，把int转换成enum，这种转换的安全性也要开发人员来保证
- 把空指针转换成目标类型的空指针
- 把任何类型的表达式转换成void类型

2.dynamic_cast

用法：dynamic_cast < type-id > (expression)

该运算符把expression转换成type-id类型的对象。Type-id必须是类的指针、类的引用或者void *；

如果type-id是类指针类型，那么expression也必须是一个指针，如果type-id是一个引用，那么expression也必须是一个引用。

主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换。

在类层次间进行上行转换时，和static_cast效果一样；下行转换时，dynamic具有类型检查的功能，更安全

3.reinpreter_cast:

用法：reinpreter_cast<type-id> (expression)

type-id必须是一个指针、引用、算术类型、函数指针或者成员指针。

他可以把一个指针转换成一个整数，也可以把整数转换成一个指针（先把一个指针转换成一个整数，再把该整数转换成原类型的指针，还可以得到原先的指

针值）

4. const_cast

用法：const_cast (expression)

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外，type_id和expression的类型是一样的。

常量指针被转化成非常量指针，并且仍然指向原来的对象；常量引用被转换成非常量引用，并且仍然指向原来的对象；常量对象被转换成非常量对象。

7.深拷贝和浅拷贝

浅拷贝：如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会根据需要生成一个默认的拷贝构造函数，完成对象之间的位拷贝，default memberwise copy即为浅拷贝

深拷贝：在某些情况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，A=B。这时如果B中有一个成员变量指针已经申请了内存，那么A中的成员变量也会指向同一块内存。如果此时B中执行析构函数释放掉指向那一块堆的指针，这时A的指针将成为悬挂指针。因此这种情况下不能简单地复制指针，而应该复制“资源”，再重新开辟一块同样大小的内存空间。

8.动态绑定和静态绑定

1. 对象的静态类型：对象在声明时采用的类型。是在编译器确定的
2. 对象的动态类型：目前所指对象的类型。是在运行期决定的，对象的动态类型可以更改，但是静态类型无法更改
3. 静态绑定：绑定的是对象的静态类型，发生在编译期
4. 动态绑定：绑定的是动态类型，发生在运行期

9.程序内存分配

1. 栈区stack:由编译器自动分配释放，存放函数的参数值，局部变量的值等，其操作方式类似于数据结构中的栈
2. 堆区heap:一般由程序员分配释放，若程序员不释放，程序结束时可能由OS回收。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表

3. 全局区（静态区）static:全局变量和静态变量的存储是放在一块的，初始化的全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域，程序结束后由系统释放

4. 文字常量区：常量字符串就是放在这里的，程序结束后由系统释放

5. 程序代码区：存放函数体的二进制代码

10.堆和栈的区别

1. 栈：由系统自动分配，只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出

2. 堆：需要程序员自己申请，并指明大小。分配内存时，首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于申请空间的堆节点，然后将该节点从空闲节点链表中删除，并将该节点的空间分配给程序。另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样代码中的delete语句才能正确释放内存空间。另外，由于找到的堆得节点大小不一定正好等于申请的大小，系统会自动地将多余的那部分重新放入空闲链表中。

3. 栈是向低地址扩展的数据结构，是一块连续的内存的区域。意思是栈顶的地址和栈的最大容量是系统预先订好的，windows下，栈的大小是2M

4. 堆是向高地址扩展的数据结构，是不连续的内存区域。由于系统是用链表来存储空闲地址的，而链表的遍历方向是低地址向高地址。堆得大小受限于计算机系统中有效的虚拟内存。堆获得的空间比较灵活，也比较大。

11.栈溢出的原因

1. 函数调用层次过深，每调用一次，函数的参数局部变量等信息就压一次栈

2. 函数变量体积太大

12.c++11的新特性

在面试中，经常被问的一个问题就是：你了解C++11哪些新特性？一般而言，回答以下四个方面就够了：

- “语法糖”：nullptr, auto自动类型推导，范围for循环，初始化列表, lambda表达式等
- 右值引用和移动语义
- 智能指针
- C++11多线程编程：thread库及其相配套的同步原语mutex, lock_guard, condition_variable, 以及异步std::furture

1. “语法糖”

这部分内容一般是一句话带过的，但是有时候也需要说一些，比较重重要的就是auto和lambda。

auto自动类型推导

C语言也有auto关键字，但是其含义只是与static变量做一个区分，一个变量不指定的话默认就是auto。。因为很少有人去用这个东西，所以在C++11中就把原有的auto功能给废弃掉了，而变成了现在的自动类型推导关键字。用法很简单不多赘述，比如写一个auto a = 3, 编译器就会自动推导a的类型为int. 在遍历某些STL容器的时候，不用去声明那些迭代器的类型，也不用去使用typedef就能很简洁的实现遍历了。

auto的使用有以下两点必须注意：

- auto声明的变量必须要初始化，否则编译器不能判断变量的类型。
- auto不能被声明为返回值，auto不能作为形参，auto不能被修饰为模板参数

关于效率：auto实际上实在编译时对变量进行了类型推导，所以不会对程序的运行效率造成不良影响。另外，auto并不会影响编译速度，因为编译时本来也要右侧推导然后判断与左侧是否匹配。

关于具体的推导规则，可以参考这里

lambda表达式

lambda表达式是匿名函数，可以认为是一个可执行体functor，语法规则如下：
[捕获区](参数区){代码区};

如
```
auto add = [](int a, int b) {return a + b};
```
就我的理解而言，捕获的意思即为将一些变量展开使得为lambda内部可见，具体方式有如下几种

- **[a,&b] 其中 *a* 以复制捕获而 *b* 以引用捕获。**
- **[this] 以引用捕获当前对象（ `*this` ）**
- **[&] 以引用捕获所有用于 lambda 体内的自动变量，并以引用捕获当前对象，若存在**
- **[=] 以复制捕获所有用于 lambda 体内的自动变量，并以引用捕获当前对象，若存在**
- **[] 不捕获，大部分情况下不捕获就可以了**

一般使用场景：sort等自定义比较函数、用thread起简单的线程。

2. 右值引用与移动语义

右值引用是C++11新特性，它实现了转移语义和完美转发，主要目的有两个方面

- 消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率
- 能够更简洁明确地定义泛型函数

C++中的变量要么是左值、要么是右值。通俗的左值定义指的是非临时变量，而左值指的是临时对象。左值引用的符号是一个&，右值引用是两个&&

移动语义

转移语义可以将资源(堆、系统对象等)从一个对象转移到另一个对象，这样可以减少不必要的临时对象的创建、拷贝及销毁。移动语义与拷贝语义是相对的，可以类比文件的剪切和拷贝。在现有的C++机制中，自定义的类要实现转移语义，需要定义移动构造函数，还可以定义转移赋值操作符。

以string类的移动构造函数为例

```
MyString(MyString&& str) {
    std::cout << "Move Ctor source from " << str._data << endl;
    _len = str._len;
    _data = str._data;
    str._len = 0;
    str._data = NULL;
}
```

和拷贝构造函数类似，有几点需要注意：

1. 参数(右值)的符号必须是&&
2. 参数(右值)不可以是常量，因为我们需要修改右值
3. 参数(右值)的资源链接和标记必须修改，否则，右值的析构函数就会释放资源。转移到新对象的资源也就无效了。

标准库函数std::move --- 将左值变成一个右值

编译器只对右值引用才能调用移动构造函数，那么如果已知一个命名对象不再被使用，此时仍然想调用它的移动构造函数，也就是**把一个左值引用当做右值引用来使用**，该怎么做呢？用std::move，这个函数以非常简单的方式将左值引用转换为右值引用。

完美转发 Perfect Forwarding

完美转发使用这样的场景：需要将一组参数原封不动地传递给另一个函数。原封不动不仅仅是参数的值不变，在C++中还有以下的两组属性：

- 左值/右值
- const / non-const

完美转发就是在参数传递过程中，所有这些属性和参数值都不能改变。在泛型函数中，这样的需求十分普遍。

为了保证这些属性，泛型函数需要重载各种版本，左值右值不同版本，还要分别对应不同的const关系，但是如果只定义一个右值引用参数的函数版本，这个问题就迎刃而解了，原因在于：

C++11对T&&的类型推导： 右值实参为右值引用，左值实参仍然为左值

3.智能指针

核心思想：为防止内存泄露等问题，用一个对象来管理野指针，使得在该对象构造时获得该指针管理权，析构时自动释放(delete).

基于此思想C++98提供了第一个智能指针：`auto_ptr`

`auto_ptr`基于所有权转移的语义，即将一个就的auto_ptr赋值给另外一个新的auto_ptr时，旧的那一个就不再拥有该指针的控制权（内部指针被赋值为null），那么这就会带来一些根本性的破绽：

- 函数参数传递时，会有隐式的赋值，那么原来的auto_ptr自动失去了控制权
- 自我赋值时，会将自己内部指针赋值为null，造成bug

因为auto_ptr的各种bug，C++11标准基本废弃了这种类型的智能指针，转而带来了三种全新的智能指针：

- **shared_ptr，基于引用计数的智能指针，会统计当前有多少个对象同时拥有该内部指针；当引用计数降为0时，自动释放**
- **weak_ptr，基于引用计数的智能指针在面对循环引用的问题将无能为力，因此C++11还引入weak_ptr与之配套使用，weak_ptr只引用，不计数**
- **unique_ptr: 遵循独占语义的智能指针，在任何时间点，资源智能唯一地被一个unique_ptr所占有，当其离开作用域时自动析构。资源所有权的转移只能通过`std::move()`而不能通过赋值**

> 展现知识广度：Java等语言的中垃圾回收机制
> 垃圾收集器将内存视为一张有向可达图，该图的节点被分成一组根节点和一组堆节点。每个堆节点对应一个内存分配块，当存在一条从任意根节点出发到达某堆节点p的有向路径时，我们就说节点p是可达的。在任意时刻，不可达节点属于垃圾。垃圾收集器通过维护这一张图，并通过定期地释放不可达节点并将它们返回给空闲链表，来定期地回收它们。
> *所以，聊到这里还可以引申malloc的分配机制、伙伴系统、虚拟内存等等概念*

这里给出一个shared_ptr的简单实现：

```
class Counter {
    friend class SmartPointPro;
public:
    Counter(){
        ptr = NULL;
        cnt = 0;
    }
    Counter(Object* p){
        ptr = p;
        cnt = 1;
    }
    ~Counter(){
        delete ptr;
    }

private:
```

```
    Object* ptr;
    int cnt;
};

class SmartPointPro {
public:
    SmartPointerPro(Object* p){
        ptr_counter = new Counter(p);
    }
    SmartPointerPro(const SmartPointerPro &sp){
        ptr_counter = sp.ptr_counter;
        ++ptr_counter->cnt;
    }
    SmartPointerPro& operator=(const SmartPointerPro &sp){
        ++sp.ptr_counter->cnt;
        --ptr_counter.cnt;
        if(ptr_counter.cnt == 0)
            delete ptr_counter;
        ptr_counter = sp.ptr_counter;
    }
    ~SmartPointerPro(){
        --ptr_counter->cnt;
        if(ptr_counter.cnt == 0)
            delete ptr_counter;
    }
private:
    Counter *ptr_counter;
};
```

需要记住的事，在以下三种情况下会引起引用计数的变更：

1. 调用构造函数时： SmartPointer p(new Object());
2. 赋值构造函数时： SmartPointer p(const SmartPointer &p);
3. 赋值时：SmartPointer p1(new Object()); SmartPointer p2 = p1;

4.C++11多线程编程

线程 `#include <thread>`

std::thread可以和普通函数和lambda表达式搭配使用。它还允许向线程执行函数传递任意多参数。

```
#include <thread>
void func() {
 // do some work here
}
int main() {
   std::thread thr(func);
```

```
    t.join();
    return 0;
}
```

上面就是一个最简单的使用std::thread的例子，函数func()在新起的线程中执行。调用join()函数是为了阻塞主线程，直到这个新起的线程执行完毕。线程函数的返回值都会被忽略，但线程函数可以接受任意数目的输入参数。

std::thread的其他成员函数

- **joinable(): 判断线程对象是否可以join,当线程对象被析构的时候如果对象``joinable()==true会导致std::terminate`被调用。**
- **join(): 阻塞当前进程(通常是主线程)，等待创建的新线程执行完毕被操作系统回收。**
- **detach(): 将线程分离，从此线程对象受操作系统管辖。**

线程管理函数

除了std::thread的成员函数外，在std::this_thread命名空间也定义了一系列函数用于管理当前线程。

函数名	作用
get_id	返回当前线程的id
yield	告知调度器运行其他线程，可用于当前处于繁忙的等待状态。相当于主动让出剩下的执行时间，具体的调度算法取决于实现
sleep_for	指定的一段时间内停止当前线程的执行
sleep_until	停止当前线程的执行直到指定的时间点

至于mutex, condition_variable等同步原语以及future关键字的使用这里不做详细介绍，如果用过自然可以说出，没有用过的话这部分内容也不应该和面试官讨论。

13.智能指针

1. 原理：智能指针是一个类，这个类的构造函数中传入一个普通指针，析构函数中释放传入的指针。智能指针的类都是栈上的对象，所以当函数（或程序）结束时会自动释放。

2. 常用智能指针：

- auto_ptr（c++98的方案，c++11已经抛弃），存在潜在的内存崩溃问题

- unique_str：替换auto_ptr，实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。他对于避免资源泄漏特别有用。

```
unique_ptr<string> p3(new string("auto"));
unique_ptr<string> p4;
p4=p3;//p4剥夺了p3的所有权，当程序运行时访问p3，会报错，但auto_ptr不会报错
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此unique_str比auto_ptr更安全

- shared_ptr：实现共享式拥有概念。多个智能指可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁时”释放。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr，unique_ptr，weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源释放。shared_ptr是为了解决auto_ptr在对象所有权上的局限性，在使用引用计数的机制上提供了可以共享所有权的指针。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权（use_count为1）

swap 交换两个shared_ptr对象

reset放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少

get 返回内部对象（指针）

- weak_ptr: 一种不控制对象生命周期的智能指针，指向一个shared_ptr管理的对象，进行该对象的内存管理的是那个强引用的shared_ptr。weak_ptr只是提供了对管理对象的一个访问手段。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，他的构造和析构不会引起引用计数的增加或减少。**weak_ptr解决shared_ptr相互引用的死锁问题**，如果两个shared相互引用，那么这两

个指针的引用计数永远不可能降为0，资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr可以相互转化。