## ● 说一下static关键字的作用

### 参考回答：

1. 全局静态变量

在全局变量前加上关键字static，全局变量就定义成一个全局静态变量.

静态存储区，在整个程序运行期间一直存在。

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；

作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2.  局部静态变量

在局部变量之前加上关键字static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）；

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变；

3. 静态函数

在函数返回类型前加static，函数就定义为静态函数。函数的定义和声明在默认情况下都是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用static修饰，那么这个函数只可在本cpp内使用，不会同其他cpp中的同名函数引起冲突；

warning：不要再头文件中声明static的全局函数，不要在cpp内声明非static的全局函数，如果你要在多个cpp中复用该函数，就把它的声明提到头文件里去，否则cpp内部声明需加上static修饰；

4. 类的静态成员

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用

5. 类的静态函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);

## ● 说一下C++和C的区别

### 参考回答：

设计思想上：

C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有重载、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等

## ● 说一下C++中static关键字的作用

### 参考回答：

对于函数定义和代码块之外的变量声明，static修改标识符的链接属性，由默认的external变为internal，作用域和存储类型不改变，这些符号只能在声明它们的源文件中访问。

对于代码块内部的变量声明，static修改标识符的存储类型，由自动变量改为静态变量，作用域和链接属性不变。这种变量在程序执行之前就创建，在程序执行的整个周期都存在。

对于被static修饰的普通函数，其只能在定义它的源文件中使用，不能在其他源文件中被引用

对于被static修饰的类成员变量和成员函数，它们是属于类的，而不是某个对象，所有对象共享一个静态成员。静态成员通过<类名>::<静态成员>来使用。

## ● 请说一下static的作用

### 参考回答：

1. 全局静态变量

在全局变量前加上关键字static，全局变量就定义成一个全局静态变量。

静态存储区，在整个程序运行期间一直存在。

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）。

作用域：全局静态变量在声明他的文件之外是不可见的，准确地说是从定义之处开始，到文件结尾。

2.  局部静态变量

在局部变量之前加上关键字static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区。

初始化：未经初始化的全局静态变量会被自动初始化为0（自动对象的值是任意的，除非他被显式初始化）。

作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变。

3. 静态函数

在函数返回类型前加static，函数就定义为静态函数。函数的定义和声明在默认情况下都是extern的，但静态函数只是在声明他的文件当中可见，不能被其他文件所用。

函数的实现使用static修饰，那么这个函数只可在本cpp内使用，不会同其他cpp中的同名函数引起冲突。

warning：不要再头文件中声明static的全局函数，不要在cpp内声明非static的全局函数，如果你要在多个cpp中复用该函数，就把它的声明提到头文件里去，否则cpp内部声明需加上static修饰。

4. 类的静态成员

在类中，静态成员可以实现多个对象之间的数据共享，并且使用静态数据成员还不会破坏隐藏的原则，即保证了安全性。因此，静态成员是类的所有对象中共享的成员，而不是某个对象的成员。对多个对象来说，静态数据成员只存储一处，供所有对象共用。

5. 类的静态函数

静态成员函数和静态数据成员一样，它们都属于类的静态成员，它们都不是对象成员。因此，对静态成员的引用不需要用对象名。

在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员（这点非常重要）。如果静态成员函数中要引用非静态成员时，可通过对象来引用。从中可看出，调用静态成员函数使用如下格式：<类名>::<静态成员函数名>(<参数表>);

## ● 说一说c++中四种cast转换

### 参考回答：

C++中四种类型转换是：static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast

1、const\_cast

用于将const变量转为非const

2、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；

3、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。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret\_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换？

C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

## ● 请说一下C/C++ 中指针和引用的区别？

### 参考回答：

1.指针有自己的一块空间，而引用只是一个别名；

2.使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

3.指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象 的引用；

4.作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引 用的修改都会改变引用所指向的对象；

5.可以有const指针，但是没有const引用；

6.指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

7.指针可以有多级指针（\*\*p），而引用至于一级；

8.指针和引用使用++运算符的意义不一样；

9.如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

## ● 给定三角形ABC和一点P(x,y,z)，判断点P是否在ABC内，给出思路并手写代码

### 参考回答：

根据面积法，如果P在三角形ABC内，那么三角形ABP的面积+三角形BCP的面积+三角形ACP的面积应该等于三角形ABC的面积。算法如下：

#include <iostream>

#include <math.h>

using namespace std;

#define ABS\_FLOAT\_0 0.0001

struct point\_float

{

float x;

float y;

};

/\*\*

\* @brief 计算三角形面积

\*/

float GetTriangleSquar(const point\_float pt0, const point\_float pt1, const point\_float pt2)

{

point\_float AB,   BC;

AB.x = pt1.x - pt0.x;

AB.y = pt1.y - pt0.y;

BC.x = pt2.x - pt1.x;

BC.y = pt2.y - pt1.y;

return fabs((AB.x \* BC.y - AB.y \* BC.x)) / 2.0f;

}

/\*\*

\* @brief 判断给定一点是否在三角形内或边上

\*/

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | bool IsInTriangle(const point\_float A, const point\_float B, const point\_float C, const point\_float D)  {  float SABC, SADB, SBDC, SADC;  SABC = GetTriangleSquar(A, B, C);  SADB = GetTriangleSquar(A, D, B);  SBDC = GetTriangleSquar(B, D, C);  SADC = GetTriangleSquar(A, D, C);  float SumSuqar = SADB + SBDC + SADC;  if ((-ABS\_FLOAT\_0 < (SABC - SumSuqar)) && ((SABC - SumSuqar) < ABS\_FLOAT\_0))  {  return true;  }  else  {  return false;  }  } |

## ● 请你说一下你理解的c++中的smart pointer四个智能指针： shared\_ptr,unique\_ptr,weak\_ptr,auto\_ptr

### 参考回答：

C++里面的四个智能指针: auto\_ptr, shared\_ptr, weak\_ptr, unique\_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1. auto\_ptr（c++98的方案，cpp11已经抛弃）

采用所有权模式。

auto\_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));

auto\_ptr<string> p2;

p2 = p1; //auto\_ptr不会报错.

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto\_ptr的缺点是：存在潜在的内存崩溃问题！

2. unique\_ptr（替换auto\_ptr）

unique\_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。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式，还是上面那个例子

|  |  |
| --- | --- |
| 1  2  3 | unique\_ptr<string> p3 (new string ("auto"));   //#4  unique\_ptr<string> p4；                       //#5  p4 = p3;//此时会报错！！ |

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\_ptr比auto\_ptr更安全。

另外unique\_ptr还有更聪明的地方：当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做，比如：

unique\_ptr<string> pu1(new string ("hello world"));

unique\_ptr<string> pu2;

pu2 = pu1;                                      // #1 not allowed

unique\_ptr<string> pu3;

pu3 = unique\_ptr<string>(new string ("You"));   // #2 allowed

其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique\_ptr赋给另一个。例如：

unique\_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << \*ps2 << \*ps1 << endl;

3. shared\_ptr

shared\_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。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，资源会被释放。

shared\_ptr 是为了解决 auto\_ptr 在对象所有权上的局限性(auto\_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

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(即交换所拥有的对象)

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), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared\_ptr<int> sp(new int(1)); sp 与 sp.get()是等价的

4. weak\_ptr

weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 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, 它的构造和析构不会引起引用记数的增加或减少。weak\_ptr是用来解决shared\_ptr相互引用时的死锁问题,如果说两个shared\_ptr相互引用,那么这两个指针的引用计数永远不可能下降为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之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr。

class B;

class A

{

public:

shared\_ptr<B> pb\_;

~A()

{

cout<<"A delete\n";

}

};

class B

{

public:

shared\_ptr<A> pa\_;

~B()

{

cout<<"B delete\n";

}

};

void fun()

{

shared\_ptr<B> pb(new B());

shared\_ptr<A> pa(new A());

pb->pa\_ = pa;

pa->pb\_ = pb;

cout<<pb.use\_count()<<endl;

cout<<pa.use\_count()<<endl;

}

int main()

{

fun();

return 0;

}

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr pb\_; 改为weak\_ptr pb\_; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意的是我们不能通过weak\_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb\_->print(); 英文pb\_是一个weak\_ptr，应该先把它转化为shared\_ptr,如：shared\_ptr p = pa->pb\_.lock(); p->print();

## ● 怎么判断一个数是二的倍数，怎么求一个数中有几个1，说一下你的思路并手写代码

### 参考回答：

1、判断一个数是不是二的倍数，即判断该数二进制末位是不是0：

a % 2 == 0 或者a & 0x0001 == 0。

2、求一个数中1的位数，可以直接逐位除十取余判断：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | int fun(long x)  {  int \_count = 0;  while(x)  {  if(x % 10 == 1)  ++\_count;  x /= 10;  }  return \_count;  }  int main()  {  cout << fun(123321) << endl;  return 0;  } |

## ● 请回答一下数组和指针的区别

### 参考回答：

指针和数组的主要区别如下：

|  |  |
| --- | --- |
| 指针 | 数组 |
| 保存数据的地址 | | 保存数据 |
| 间接访问数据，首先获得指针的内容，然后将其作为地址，从该地址中提取数据 | | 直接访问数据， |
| 通常用于动态的数据结构 | | 通常用于固定数目且数据类型相同的元素 |
| 通过Malloc分配内存，free释放内存 | | 隐式的分配和删除 |
| 通常指向匿名数据，操作匿名函数 | | 自身即为数据名 |

## ● 请你回答一下野指针是什么？

### 参考回答：

野指针就是指向一个已删除的对象或者未申请访问受限内存区域的指针

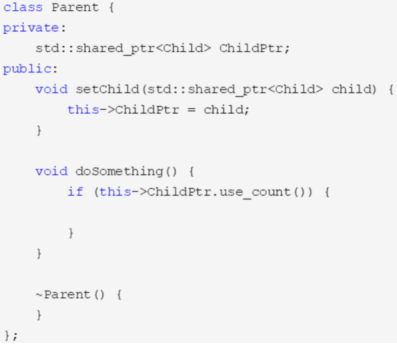
## ● 请你介绍一下C++中的智能指针

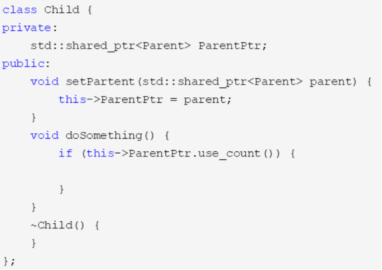
### 参考回答：

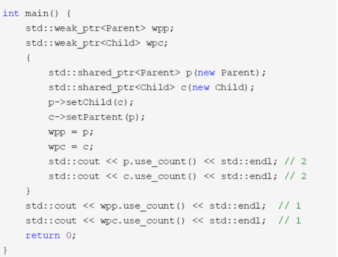
智能指针主要用于管理在堆上分配的内存，它将普通的指针封装为一个栈对象。当栈对象的生存周期结束后，会在析构函数中释放掉申请的内存，从而防止内存泄漏。C++ 11中最常用的智能指针类型为shared\_ptr,它采用引用计数的方法，记录当前内存资源被多少个智能指针引用。该引用计数的内存在堆上分配。当新增一个时引用计数加1，当过期时引用计数减一。只有引用计数为0时，智能指针才会自动释放引用的内存资源。对shared\_ptr进行初始化时不能将一个普通指针直接赋值给智能指针，因为一个是指针，一个是类。可以通过make\_shared函数或者通过构造函数传入普通指针。并可以通过get函数获得普通指针。

## ● 请你回答一下智能指针有没有内存泄露的情况

### 参考回答：

当两个对象相互使用一个shared\_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄漏。例如：





上述代码中，parent有一个shared\_ptr类型的成员指向孩子，而child也有一个shared\_ptr类型的成员指向父亲。然后在创建孩子和父亲对象时也使用了智能指针c和p，随后将c和p分别又赋值给child的智能指针成员parent和parent的智能指针成员child。从而形成了一个循环引用：

## ● 请你来说一下智能指针的内存泄漏如何解决

### 参考回答：

为了解决循环引用导致的内存泄漏，引入了weak\_ptr弱指针，weak\_ptr的构造函数不会修改引用计数的值，从而不会对对象的内存进行管理，其类似一个普通指针，但不指向引用计数的共享内存，但是其可以检测到所管理的对象是否已经被释放，从而避免非法访问。

## ● 请你理解的c++中的引用和指针

### 参考回答：

定义：

1、引用：

C++是C语言的继承，它可进行过程化程序设计，又可以进行以抽象数据类型为特点的基于对象的程序设计，还可以进行以继承和多态为特点的面向对象的程序设计。引用就是C++对C语言的重要扩充。引用就是某一变量的一个别名，对引用的操作与对变量直接操作完全一样。引用的声明方法：类型标识符 &引用名=目标变量名；引用引入了对象的一个同义词。定义引用的表示方法与定义[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88/2878304" \t "_blank)相似，只是用&代替了\*。

2、指针：

指针利用地址，它的值直接指向存在电脑存储器中另一个地方的值。由于通过地址能找到所需的变量单元，可以说，地址指向该变量单元。因此，将地址形象化的称为“指针”。意思是通过它能找到以它为地址的内存单元。

区别：

1、指针有自己的一块空间，而引用只是一个别名；

2、使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

3、指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用；

4、作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象；

5、可以有const指针，但是没有const引用；

6、指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；

7、指针可以有多级指针（\*\*p），而引用至于一级；

8、指针和引用使用++运算符的意义不一样；

9、如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

**● 请你来说一下C++中的智能指针**

**参考回答：**

C++里面的四个智能指针: auto\_ptr, shared\_ptr, weak\_ptr, unique\_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

为什么要使用智能指针：

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

1、auto\_ptr（c++98的方案，cpp11已经抛弃）

采用所有权模式。

auto\_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));

auto\_ptr<string> p2;

p2 = p1; //auto\_ptr不会报错.

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto\_ptr的缺点是：存在潜在的内存崩溃问题！

2、unique\_ptr（替换auto\_ptr）

unique\_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。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式，还是上面那个例子

unique\_ptr<string> p3 (new string ("auto"));           //#4

unique\_ptr<string> p4；                           //#5

p4 = p3;//此时会报错！！

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\_ptr比auto\_ptr更安全。

另外unique\_ptr还有更聪明的地方：当程序试图将一个 unique\_ptr 赋值给另一个时，如果源 unique\_ptr 是个临时右值，编译器允许这么做；如果源 unique\_ptr 将存在一段时间，编译器将禁止这么做，比如：

unique\_ptr<string> pu1(new string ("hello world"));

unique\_ptr<string> pu2;

pu2 = pu1;                                          // #1 not allowed

unique\_ptr<string> pu3;

pu3 = unique\_ptr<string>(new string ("You"));           // #2 allowed

其中#1留下悬挂的unique\_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique\_ptr，因为它调用 unique\_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique\_ptr 优于允许两种赋值的auto\_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique\_ptr赋给另一个。例如：

unique\_ptr<string> ps1, ps2;

ps1 = demo("hello");

ps2 = move(ps1);

ps1 = demo("alexia");

cout << \*ps2 << \*ps1 << endl;

3、shared\_ptr

shared\_ptr实现共享式拥有概念。多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。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，资源会被释放。

shared\_ptr 是为了解决 auto\_ptr 在对象所有权上的局限性(auto\_ptr 是独占的), 在使用引用计数的机制上提供了可以共享所有权的智能指针。

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(即交换所拥有的对象)

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), 由于已经重载了()方法, 因此和直接使用对象是一样的.如 shared\_ptr<int> sp(new int(1)); sp 与 sp.get()是等价的

4、weak\_ptr

weak\_ptr 是一种不控制对象生命周期的智能指针, 它指向一个 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, 它的构造和析构不会引起引用记数的增加或减少。weak\_ptr是用来解决shared\_ptr相互引用时的死锁问题,如果说两个shared\_ptr相互引用,那么这两个指针的引用计数永远不可能下降为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之间可以相互转化，shared\_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared\_ptr。

class B;

class A

{

public:

shared\_ptr<B> pb\_;

~A()

{

cout<<"A delete\n";

}

};

class B

{

public:

shared\_ptr<A> pa\_;

~B()

{

cout<<"B delete\n";

}

};

void fun()

{

shared\_ptr<B> pb(new B());

shared\_ptr<A> pa(new A());

pb->pa\_ = pa;

pa->pb\_ = pb;

cout<<pb.use\_count()<<endl;

cout<<pa.use\_count()<<endl;

}

int main()

{

fun();

return 0;

}

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak\_ptr就可以了，我们把类A里面的shared\_ptr pb\_; 改为weak\_ptr pb\_; 运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意的是我们不能通过weak\_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb\_->print(); 英文pb\_是一个weak\_ptr，应该先把它转化为shared\_ptr,如：shared\_ptr p = pa->pb\_.lock(); p->print();

**● 请你回答一下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 考点:虚函数 析构函数**

**参考回答：**

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数，而是只有当需要当作父类时，设置为虚函数。

**● 请你来说一下函数指针**

**参考回答：**

1、定义

函数指针是指向函数的指针变量。

函数指针本身首先是一个指针变量，该指针变量指向一个具体的函数。这正如用指针变量可指向整型变量、字符型、数组一样，这里是指向函数。

C在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。有了指向函数的指针变量后，可用该指针变量调用函数，就如同用指针变量可引用其他类型变量一样，在这些概念上是大体一致的。

2、用途：

调用函数和做函数的参数，比如回调函数。

3、示例：

char \* fun(char \* p)  {…}       // 函数fun

char \* (\*pf)(char \* p);             // 函数指针pf

pf = fun;                        // 函数指针pf指向函数fun

pf(p);                        // 通过函数指针pf调用函数fun

**● 请你来说一下fork函数**

**参考回答：**

Fork：创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。

最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。

在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

**● 请你来说一下C++中析构函数的作用**

**参考回答：**

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

**● 请你来说一下静态函数和虚函数的区别**

**参考回答：**

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

**● 请你来说一说重载和覆盖**

**参考回答：**

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中  
重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

**● 请你来说一说static关键字**

**参考回答：**

1.加了static关键字的全局变量只能在本文件中使用。例如在a.c中定义了static int a=10;那么在b.c中用extern int a是拿不到a的值得，a的作用域只在a.c中。 2.static定义的静态局部变量分配在数据段上，普通的局部变量分配在栈上，会因为函数栈帧的释放而被释放掉。  
3. 对一个类中成员变量和成员函数来说，加了static关键字，则此变量/函数就没有了this指针了，必须通过类名才能访问

**● 请你说一说strcpy和strlen**

**参考回答：**

strcpy是字符串拷贝函数，原型：

char \*strcpy(char\* dest, const char \*src);

从src逐字节拷贝到dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是strncpy函数。  
strlen函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个数。

**● 请你说一说你理解的虚函数和多态**

**参考回答：**

多态的实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定；动态多态是用虚函数机制实现的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。  
虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

**● 请你来回答一下++i和i++的区别**

**参考回答：**

++i先自增1，再返回，i++先返回i,再自增1

**● 请你来说一说++i和i++的实现**

**参考回答：**

1. ++i 实现：

int&  int::operator++（）

{

\*this +=1；

return \*this；

}

2.  i++ 实现：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | const int  int::operator（int）  {  int oldValue = \*this；  ++（\*this）；  return oldValue；  } |

**● 请你来写个函数在main函数执行前先运行**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4 | \_\_attribute((constructor))void before()  {      printf("before main\n");  } |

**● 有段代码写成了下边这样，如果在只修改一个字符的前提下，使代码输出20个hello? for(int i = 0; i < 20; i--) cout << "hello" << endl;**

**参考回答：**

|  |  |
| --- | --- |
| 1  2 | for(int i = 0; i + 20; i--)  cout << "hello" << endl; |

**● 请你来说一下智能指针shared\_ptr的实现**

参考回答:

**核心要理解引用计数，什么时候销毁底层指针，还有赋值，拷贝构造时候的引用计数的变化，析构的时候要判断底层指针的引用计数为0了才能真正释放底层指针的内存**

**template <typename T>**

**class SmartPtr**

**{**

**private:**

**T \*ptr;    //底层真实的指针**

**int \*use\_count;//保存当前对象被多少指针引用计数**

**public:**

**SmartPtr(T \*p); //SmartPtr<int>p(new int(2));**

**SmartPtr(const SmartPtr<T>&orig);//SmartPtr<int>q(p);**

**SmartPtr<T>&operator=(const SmartPtr<T> &rhs);//q=p**

**~SmartPtr();**

**T operator\*();  //为了能把智能指针当成普通指针操作定义解引用操作**

**T\*operator->();  //定义取成员操作**

**T\* operator+(int i);//定义指针加一个常数**

**int operator-(SmartPtr<T>&t1,SmartPtr<T>&t2);//定义两个指针相减**

**void getcount() { return \*use\_count } }; template <typename T> int SmartPtr<T>::operator-(SmartPtr<T> &t1, SmartPtr<T> &t2) { return t1.ptr-t2.ptr; } template <typename T> SmartPtr<T>::SmartPtr(T \*p) { ptr=p; try { use\_count=new int(1); }catch (...) {**

**delete ptr;    //申请失败释放真实指针和引用计数的内存**

**ptr= nullptr; delete use\_count; use\_count= nullptr;** **}** **} template <typename T>**

**SmartPtr<T>::SmartPtr(const SmartPtr<T> &orig) //复制构造函数**

**{**

**use\_count=orig.use\_count;//引用计数保存在一块内存，所有的SmarPtr对象的引用计数都指向这里**

**this->ptr=orig.ptr;**

**++(\*use\_count);//当前对象的引用计数加1**

**}** **template <typename T> SmartPtr<T>& SmartPtr<T>::operator=(const SmartPtr<T> &rhs) {**

**//重载=运算符，例如SmartPtr<int>p,q; p=q;这个语句中，首先给q指向的对象的引用计数加1，因为p重新指向了q所指的对象，所以p需要先给原来的对象的引用计数减1，如果减一后为0，先释放掉p原来指向的内存，然后讲q指向的对象的引用计数加1后赋值给p**

**++\*(rhs.use\_count); if((--\*(use\_count))==0) {** **delete ptr; ptr= nullptr; delete use\_count; use\_count= nullptr; } ptr=rhs.ptr; \*use\_count=\*(rhs.use\_count); return \*this;** **} template <typename T> SmartPtr<T>::~SmartPtr() { getcount();**

**if(--(\*use\_count)==0)  //SmartPtr的对象会在其生命周期结束的时候调用其析构函数，在析构函数中检测当前对象的引用计数是不是只有正在结束生命周期的这个SmartPtr引用，如果是，就释放掉，如果不是，就还有其他的SmartPtr引用当前对象，就等待其他的SmartPtr对象在其生命周期结束的时候调用析构函数释放掉**

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | {  getcount();  delete ptr;  ptr= nullptr;  delete use\_count;  use\_count=nullptr;  }  }  template <typename T>  T SmartPtr<T>::operator\*()  {  return \*ptr;  }  template <typename T>  T\*  SmartPtr<T>::operator->()  {  return ptr;  }  template <typename T>  T\* SmartPtr<T>::operator+(int i)  {  T \*temp=ptr+i;  return temp;  }  } |

**● 以下四行代码的区别是什么？ const char \* arr = "123"; char \* brr = "123"; const char crr[] = "123"; char drr[] = "123";**

**参考回答：**

const char \* arr = "123";

//字符串123保存在常量区，const本来是修饰arr指向的值不能通过arr去修改，但是字符串“123”在常量区，本来就不能改变，所以加不加const效果都一样

char \* brr = "123";

//字符串123保存在常量区，这个arr指针指向的是同一个位置，同样不能通过brr去修改"123"的值

const char crr[] = "123";

//这里123本来是在栈上的，但是编译器可能会做某些优化，将其放到常量区

char drr[] = "123";

//字符串123保存在栈区，可以通过drr去修改

**● 请你来说一下C++里是怎么定义常量的？常量存放在内存的哪个位置？**

**参考回答：**

常量在C++里的定义就是一个top-level const加上对象类型，常量定义必须初始化。对于局部对象，常量存放在栈区，对于全局对象，常量存放在全局/静态存储区。对于字面值常量，常量存放在常量存储区。

**● 请你来回答一下const修饰成员函数的目的是什么？**

**参考回答：**

const修饰的成员函数表明函数调用不会对对象做出任何更改，事实上，如果确认不会对对象做更改，就应该为函数加上const限定，这样无论const对象还是普通对象都可以调用该函数。

**如果同时定义了两个函数，一个带const，一个不带，会有问题吗？**

**参考回答：**

不会，这相当于函数的重载。

**● 请你来说一说隐式类型转换**

**参考回答：**

首先，对于内置类型，低精度的变量给高精度变量赋值会发生隐式类型转换，其次，对于只存在单个参数的构造函数的对象构造来说，函数调用可以直接使用该参数传入，编译器会自动调用其构造函数生成临时对象。

**● 说说你了解的类型转换**

**参考回答：**

reinterpret\_cast：可以用于任意类型的指针之间的转换，对转换的结果不做任何保证  
dynamic\_cast：这种其实也是不被推荐使用的，更多使用static\_cast，dynamic本身只能用于存在虚函数的父子关系的强制类型转换，对于指针，转换失败则返回nullptr，对于引用，转换失败会抛出异常  
const\_cast：对于未定义const版本的成员函数，我们通常需要使用const\_cast来去除const引用对象的const，完成函数调用。另外一种使用方式，结合static\_cast，可以在非const版本的成员函数内添加const，调用完const版本的成员函数后，再使用const\_cast去除const限定。  
static\_cast：完成基础数据类型；同一个继承体系中类型的转换；任意类型与空指针类型void\* 之间的转换。

**● 请你来说一说C++函数栈空间的最大值**

**参考回答：**

默认是1M，不过可以调整

**● 请你来说一说extern“C”**

参考回答:

C++调用C函数需要extern C，因为C语言没有函数重载。

**● 请你回答一下new/delete与malloc/free的区别是什么**

**参考回答：**

首先，new/delete是C++的关键字，而malloc/free是C语言的库函数，后者使用必须指明申请内存空间的大小，对于类类型的对象，后者不会调用构造函数和析构函数

**● 请你说说你了解的RTTI**

**参考回答：**

运行时类型检查，在C++层面主要体现在dynamic\_cast和typeid,VS中虚函数表的-1位置存放了指向type\_info的指针。对于存在虚函数的类型，typeid和dynamic\_cast都会去查询type\_info

**● 请你说说虚函数表具体是怎样实现运行时多态的?**

**参考回答：**

子类若重写父类虚函数，虚函数表中，该函数的地址会被替换，对于存在虚函数的类的对象，在VS中，对象的对象模型的头部存放指向虚函数表的指针，通过该机制实现多态。

**● 请你说说C语言是怎么进行函数调用的？**

**参考回答：**

每一个函数调用都会分配函数栈，在栈内进行函数执行过程。调用前，先把返回地址压栈，然后把当前函数的esp指针压栈。

**● 请你说说C语言参数压栈顺序？**

**参考回答：**

从右到左

**● 请你说说C++如何处理返回值？**

**参考回答：**

生成一个临时变量，把它的引用作为函数参数传入函数内。

**● 请你回答一下C++中拷贝赋值函数的形参能否进行值传递？**

**参考回答：**

不能。如果是这种情况下，调用拷贝构造函数的时候，首先要将实参传递给形参，这个传递的时候又要调用拷贝构造函数。。如此循环，无法完成拷贝，栈也会满。

**● 请你回答一下malloc与new区别**

**参考回答：**

malloc需要给定申请内存的大小，返回的指针需要强转。  
new会调用构造函数，不用指定内存大小，返回的指针不用强转。

**● 请你说一说select**

**参考回答：**

select在使用前，先将需要监控的描述符对应的bit位置1，然后将其传给select,当有任何一个事件发生时，select将会返回所有的描述符，需要在应用程序自己遍历去检查哪个描述符上有事件发生，效率很低，并且其不断在内核态和用户态进行描述符的拷贝，开销很大

**● 请你说说fork,wait,exec函数**

**参考回答：**

父进程产生子进程使用fork拷贝出来一个父进程的副本，此时只拷贝了父进程的页表，两个进程都读同一块内存，当有进程写的时候使用写实拷贝机制分配内存，exec函数可以加载一个elf文件去替换父进程，从此父进程和子进程就可以运行不同的程序了。fork从父进程返回子进程的pid，从子进程返回0.调用了wait的父进程将会发生阻塞，直到有子进程状态改变,执行成功返回0，错误返回-1。exec执行成功则子进程从新的程序开始运行，无返回值，执行失败返回-1

**● 请你回答一下静态函数和虚函数的区别**

**参考回答：**

静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。虚函数因为用了虚函数表机制，调用的时候会增加一次内存开销

**● 请你说一说重载和覆盖**

**参考回答：**

重载：两个函数名相同，但是参数列表不同（个数，类型），返回值类型没有要求，在同一作用域中  
重写：子类继承了父类，父类中的函数是虚函数，在子类中重新定义了这个虚函数，这种情况是重写

**● 请你说一说static关键字**

**参考回答：**

1.加了static关键字的全局变量只能在本文件中使用。例如在a.c中定义了static int a=10;那么在b.c中用extern int a是拿不到a的值得，a的作用域只在a.c中。  
2.static定义的静态局部变量分配在数据段上，普通的局部变量分配在栈上，会因为函数栈帧的释放而被释放掉。  
3. 对一个类中成员变量和成员函数来说，加了static关键字，则此变量/函数就没有了this指针了，必须通过类名才能访问

**● 请你说一说strcpy和strlen**

**参考回答：**

strcpy是字符串拷贝函数，原型：

char \*strcpy(char\* dest, const char \*src);

从src逐字节拷贝到dest，直到遇到'\0'结束，因为没有指定长度，可能会导致拷贝越界，造成缓冲区溢出漏洞,安全版本是strncpy函数。  
strlen函数是计算字符串长度的函数，返回从开始到'\0'之间的字符个

**请你来说一下map和set有什么区别，分别又是怎么实现的？**

**参考回答：**

map和set都是C++的关联容器，其底层实现都是红黑树（RB-Tree）。由于 map 和set所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 和set的操作行为，都只是转调 RB-tree 的操作行为。

map和set区别在于：

（1）map中的元素是key-value（关键字—值）对：关键字起到索引的作用，值则表示与索引相关联的数据；Set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字。

（2）set的迭代器是const的，不允许修改元素的值；map允许修改value，但不允许修改key。其原因是因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值；而map的迭代器则不允许修改key值，允许修改value值。

（3）map支持下标操作，set不支持下标操作。map可以用key做下标，map的下标运算符[ ]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped\_type类型默认值的元素至map中，因此下标运算符[ ]在map应用中需要慎用，const\_map不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped\_type类型没有默认值也不应该使用。如果find能解决需要，尽可能用find。

**● 请你来介绍一下STL的allocaotr**

**参考回答：**

STL的分配器用于封装STL容器在内存管理上的底层细节。在C++中，其内存配置和释放如下：

new运算分两个阶段：(1)调用::operator new配置内存;(2)调用对象构造函数构造对象内容

delete运算分两个阶段：(1)调用对象希构函数；(2)掉员工::operator delete释放内存

为了精密分工，STL allocator将两个阶段操作区分开来：内存配置有alloc::allocate()负责，内存释放由alloc::deallocate()负责；对象构造由::construct()负责，对象析构由::destroy()负责。

同时为了提升内存管理的效率，减少申请小内存造成的内存碎片问题，SGI STL采用了两级配置器，当分配的空间大小超过128B时，会使用第一级空间配置器；当分配的空间大小小于128B时，将使用第二级空间配置器。第一级空间配置器直接使用malloc()、realloc()、free()函数进行内存空间的分配和释放，而第二级空间配置器采用了内存池技术，通过空闲链表来管理内存。

**● 请你来说一说STL迭代器删除元素**

**参考回答：**

这个主要考察的是迭代器失效的问题。1.对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；2.对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator，因此上面两种正确的方法都可以使用。

**● 请你说一说STL中MAP数据存放形式**

**参考回答：**

红黑树。unordered map底层结构是哈希表

**● 请你讲讲STL有什么基本组成**

**参考回答：**

STL主要由：以下几部分组成：  
容器迭代器仿函数算法分配器配接器  
他们之间的关系：分配器给容器分配存储空间，算法通过迭代器获取容器中的内容，仿函数可以协助算法完成各种操作，配接器用来套接适配仿函数

**● 请你说说STL中map与unordered\_map**

**参考回答：**

1、Map映射，map 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。不允许键值重复。

底层实现：红黑树

适用场景：有序键值对不重复映射

2、Multimap

多重映射。multimap 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。允许键值重复。

底层实现：红黑树

适用场景：有序键值对可重复映射

**● 请你说一说vector和list的区别，应用，越详细越好**

**参考回答：**

1、概念：

1）Vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector 增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效。

性能：

访问：O(1)

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除。

2、List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点。

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

2、区别：

1）vector底层实现是数组；list是双向 链表。

2）vector支持随机访问，list不支持。

3）vector是顺序内存，list不是。

4）vector在中间节点进行插入删除会导致内存拷贝，list不会。

5）vector一次性分配好内存，不够时才进行2倍扩容；list每次插入新节点都会进行内存申请。

6）vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好。

3、应用

vector拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用vector。

list拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用list。

**● 请你来说一下STL中迭代器的作用，有指针为何还要迭代器**

**参考回答：**

1、迭代器

Iterator（迭代器）模式又称Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由iterator提供的方法）访问聚合对象中的各个元素。

由于Iterator模式的以上特性：与聚合对象耦合，在一定程度上限制了它的广泛运用，一般仅用于底层聚合支持类，如STL的list、vector、stack等容器类及ostream\_iterator等扩展iterator。

2、迭代器和指针的区别

迭代器不是指针，是类模板，表现的像指针。他只是模拟了指针的一些功能，通过重载了指针的一些操作符，->、\*、++、--等。迭代器封装了指针，是一个“可遍历STL（ Standard Template Library）容器内全部或部分元素”的对象， 本质是封装了原生指针，是指针概念的一种提升（lift），提供了比指针更高级的行为，相当于一种智能指针，他可以根据不同类型的数据结构来实现不同的++，--等操作。

迭代器返回的是对象引用而不是对象的值，所以cout只能输出迭代器使用\*取值后的值而不能直接输出其自身。

3、迭代器产生原因

Iterator类的访问方式就是把不同集合类的访问逻辑抽象出来，使得不用暴露集合内部的结构而达到循环遍历集合的效果。

**● 请你说一说epoll原理**

**参考回答：**

调用顺序：

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

int epoll\_wait(int epfd, struct epoll\_event \*events,int maxevents, int timeout);

首先创建一个epoll对象，然后使用epoll\_ctl对这个对象进行操作，把需要监控的描述添加进去，这些描述如将会以epoll\_event结构体的形式组成一颗红黑树，接着阻塞在epoll\_wait，进入大循环，当某个fd上有事件发生时，内核将会把其对应的结构体放入到一个链表中，返回有事件发生的链表。

**● 请你说一说STL迭代器是怎么删除元素的呢**

**参考回答：**

这个主要考察的是迭代器失效的问题。

1.对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；

2.对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

3.对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator，因此上面两种正确的方法都可以使用。

**● 请你说一说STL中MAP数据存放形式**

**参考回答：**

红黑树。unordered map底层结构是哈希表

**● n个整数的无序数组，找到每个元素后面比它大的第一个数，要求时间复杂度为O(N)**

**参考回答：**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | vector<int> findMax(vector<int>num)  {  if(num.size()==0)return num;  vector<int>res(num.size());  int i=0;  stack<int>s;  while(i<num.size())  {  if(s.empty()||num[s.top()]>=num[i])  {  s.push(i++);  }  else  {  res[s.top()]=num[i];  s.pop();  }  }  while(!s.empty())  {  res[s.top()]=INT\_MAX;  s.pop();  }  for(int i=0; i<res.size(); i++)  cout<<res[i]<<endl;  return res;  } |

**● 请你回答一下STL里resize和reserve的区别**

**参考回答：**

resize()：改变当前容器内含有元素的数量(size())，eg: vector<int>v; v.resize(len);v的size变为len,如果原来v的size小于len，那么容器新增（len-size）个元素，元素的值为默认为0.当v.push\_back(3);之后，则是3是放在了v的末尾，即下标为len，此时容器是size为len+1；  
reserve()：改变当前容器的最大容量（capacity）,它不会生成元素，只是确定这个容器允许放入多少对象，如果reserve(len)的值大于当前的capacity()，那么会重新分配一块能存len个对象的空间，然后把之前v.size()个对象通过copy construtor复制过来，销毁之前的内存；  
测试代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include <iostream>  #include <vector>  using namespace std;  int main() {      vector<int> a;      a.reserve(100);      a.resize(50);      cout<<a.size()<<"  "<<a.capacity()<<endl;          //50  100      a.resize(150);      cout<<a.size()<<"  "<<a.capacity()<<endl;          //150  200      a.reserve(50);      cout<<a.size()<<"  "<<a.capacity()<<endl;          //150  200      a.resize(50);      cout<<a.size()<<"  "<<a.capacity()<<endl;          //50  200  } |

**● 请你说一说stl里面set和map怎么实现的**

**参考回答：**

集合，所有元素都会根据元素的值自动被排序，且不允许重复。

底层实现：红黑树

set 底层是通过红黑树（RB-tree）来实现的，由于红黑树是一种平衡二叉搜索树，自动排序的效果很不错，所以标准的 STL 的 set 即以 RB-Tree 为底层机制。又由于 set 所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 set 操作行为，都只有转调用 RB-tree 的操作行为而已。

适用场景：有序不重复集合

2、map

映射。map 的所有元素都是 pair，同时拥有实值（value）和键值（key）。pair 的第一元素被视为键值，第二元素被视为实值。所有元素都会根据元素的键值自动被排序。不允许键值重复。

底层：红黑树

适用场景：有序键值对不重复映射

**● 请你来说一下C++中类成员的访问权限**

**参考回答：**

参考回答：C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员

**● 请你来说一下C++中struct和class的区别**

**参考回答：**

在C++中，可以用struct和class定义类，都可以继承。区别在于：structural的默认继承权限和默认访问权限是public，而class的默认继承权限和默认访问权限是private。

另外，class还可以定义模板类形参，比如template <class T, int i>。

**● 请你回答一下C++类内可以定义引用数据成员吗？**

**参考回答：**

可以，必须通过成员函数初始化列表初始化。

**● 请你回答一下什么是右值引用，跟左值又有什么区别？**

**参考回答：**

右值引用是C++11中引入的新特性 , 它实现了转移语义和精确传递。它的主要目的有两个方面：

1. 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。

2. 能够更简洁明确地定义泛型函数。

左值和右值的概念：

左值：能对表达式取地址、或具名对象/变量。一般指表达式结束后依然存在的持久对象。

右值：不能对表达式取地址，或匿名对象。一般指表达式结束就不再存在的临时对象。

右值引用和左值引用的区别：

1. 左值可以寻址，而右值不可以。

2. 左值可以被赋值，右值不可以被赋值，可以用来给左值赋值。

3. 左值可变,右值不可变（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）。

**请你来说一下一个C++源文件从文本到可执行文件经历的过程？**

**参考回答：**

对于C++源文件，从文本到可执行文件一般需要四个过程：

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

**● 请你来回答一下include头文件的顺序以及双引号””和尖括号<>的区别？**

**参考回答：**

Include头文件的顺序：对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h。那么要在a.c文件中引用b.h文件，并且要先引用b.h，后引用a.h,否则汇报变量类型未声明错误。

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

对于使用尖括号包含的头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS\_INCLUDE\_PATH/C\_INCLUDE\_PATH指定的头文件路径

**● 请你回答一下malloc的原理，另外brk系统调用和mmap系统调用的作用分别是什么？**

**参考回答：**

Malloc函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。Malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同时malloc采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

当进行内存分配时，Malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

Malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请。其中当申请内存小于128K时，会使用系统函数brk在堆区中分配；而当申请内存大于128K时，会使用系统函数mmap在映射区分配。

**● 请你说一说C++的内存管理是怎样的？**

**参考回答：**

在C++中，虚拟内存分为代码段、数据段、BSS段、堆区、文件映射区以及栈区六部分。

代码段:包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

数据段：存储程序中已初始化的全局变量和静态变量

bss 段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量。

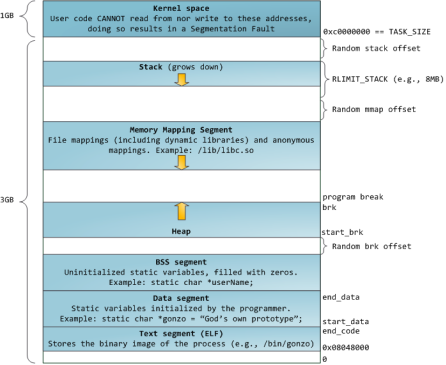
堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存。

映射区:存储动态链接库以及调用mmap函数进行的文件映射

栈：使用栈空间存储函数的返回地址、参数、局部变量、返回值

**● 请你来说一下C++/C的内存分配**

**参考回答：**



32bitCPU可寻址4G线性空间，每个进程都有各自独立的4G逻辑地址，其中0~3G是用户态空间，3~4G是内核空间，不同进程相同的逻辑地址会映射到不同的物理地址中。其逻辑地址其划分如下：

各个段说明如下：

3G用户空间和1G内核空间

静态区域：

text segment(代码段):包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

data segment(数据段)：存储程序中已初始化的全局变量和静态变量

bss segment：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量，对于未初始化的全局变量和静态变量，程序运行main之前时会统一清零。即未初始化的全局变量编译器会初始化为0

动态区域：

heap（堆）： 当进程未调用malloc时是没有堆段的，只有调用malloc时采用分配一个堆，并且在程序运行过程中可以动态增加堆大小(移动break指针)，从低地址向高地址增长。分配小内存时使用该区域。  堆的起始地址由mm\_struct 结构体中的start\_brk标识，结束地址由brk标识。

memory mapping segment(映射区):存储动态链接库等文件映射、申请大内存（malloc时调用mmap函数）

stack（栈）：使用栈空间存储函数的返回地址、参数、局部变量、返回值，从高地址向低地址增长。在创建进程时会有一个最大栈大小，Linux可以通过ulimit命令指定。

**● 请你回答一下如何判断内存泄漏？**

**参考回答：**

内存泄漏通常是由于调用了malloc/new等内存申请的操作，但是缺少了对应的free/delete。为了判断内存是否泄露，我们一方面可以使用linux环境下的内存泄漏检查工具Valgrind,另一方面我们在写代码时可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否泄露。

**● 请你来说一下什么时候会发生段错误**

**参考回答：**

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

使用野指针

试图修改字符串常量的内容

**● 请你来回答一下什么是memory leak，也就是内存泄漏**

**参考回答：**

内存泄漏(memory leak)是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的分类：

1. 堆内存泄漏 （Heap leak）。对内存指的是程序运行中根据需要分配通过malloc,realloc new等从堆中分配的一块内存，再是完成后必须通过调用对应的 free或者delete 删掉。如果程序的设计的错误导致这部分内存没有被释放，那么此后这块内存将不会被使用，就会产生Heap Leak.

2. 系统资源泄露（Resource Leak）。主要指程序使用系统分配的资源比如 Bitmap,handle ,SOCKET等没有使用相应的函数释放掉，导致系统资源的浪费，严重可导致系统效能降低，系统运行不稳定。

3. 没有将基类的析构函数定义为虚函数。当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露。

**● 请你来回答一下new和malloc的区别**

**参考回答：**

1、new分配内存按照数据类型进行分配，malloc分配内存按照指定的大小分配；

2、new返回的是指定对象的指针，而malloc返回的是void\*，因此malloc的返回值一般都需要进行类型转化。

3、new不仅分配一段内存，而且会调用构造函数，malloc不会。

4、new分配的内存要用delete销毁，malloc要用free来销毁；delete销毁的时候会调用对象的析构函数，而free则不会。

5、new是一个操作符可以重载，malloc是一个库函数。

6、malloc分配的内存不够的时候，可以用realloc扩容。扩容的原理？new没用这样操作。

7、new如果分配失败了会抛出bad\_malloc的异常，而malloc失败了会返回NULL。

8、申请数组时： new[]一次分配所有内存，多次调用构造函数，搭配使用delete[]，delete[]多次调用析构函数，销毁数组中的每个对象。而malloc则只能sizeof(int) \* n。

**● 请你来说一下共享内存相关api**

**参考回答：**

Linux允许不同进程访问同一个逻辑内存，提供了一组API，头文件在sys/shm.h中。

1）新建共享内存shmget

int shmget(key\_t key,size\_t size,int shmflg);

key：共享内存键值，可以理解为共享内存的唯一性标记。

size：共享内存大小

shmflag：创建进程和其他进程的读写权限标识。

返回值：相应的共享内存标识符，失败返回-1

2）连接共享内存到当前进程的地址空间shmat

void \*shmat(int shm\_id,const void \*shm\_addr,int shmflg);

shm\_id：共享内存标识符

shm\_addr：指定共享内存连接到当前进程的地址，通常为0，表示由系统来选择。

shmflg：标志位

返回值：指向共享内存第一个字节的指针，失败返回-1

3）当前进程分离共享内存shmdt

int shmdt(const void \*shmaddr);

4）控制共享内存shmctl

和信号量的semctl函数类似，控制共享内存

int shmctl(int shm\_id,int command,struct shmid\_ds \*buf);

shm\_id：共享内存标识符

command: 有三个值

IPC\_STAT:获取共享内存的状态，把共享内存的shmid\_ds结构复制到buf中。

IPC\_SET:设置共享内存的状态，把buf复制到共享内存的shmid\_ds结构。

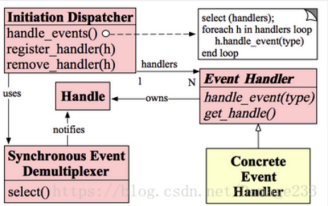
IPC\_RMID:删除共享内存

buf：共享内存管理结构体。

**● 请你来说一下reactor模型组成**

**参考回答：**

reactor模型要求主线程只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程，除此之外，主线程不做任何其他实质性的工作，读写数据、接受新的连接以及处理客户请求均在工作线程中完成。其模型组成如下：



1）Handle：即操作系统中的句柄，是对资源在操作系统层面上的一种抽象，它可以是打开的文件、一个连接(Socket)、Timer等。由于Reactor模式一般使用在网络编程中，因而这里一般指Socket Handle，即一个网络连接。

2）Synchronous Event Demultiplexer（同步事件复用器）：阻塞等待一系列的Handle中的事件到来，如果阻塞等待返回，即表示在返回的Handle中可以不阻塞的执行返回的事件类型。这个模块一般使用操作系统的select来实现。

3）Initiation Dispatcher：用于管理Event Handler，即EventHandler的容器，用以注册、移除EventHandler等；另外，它还作为Reactor模式的入口调用Synchronous Event Demultiplexer的select方法以阻塞等待事件返回，当阻塞等待返回时，根据事件发生的Handle将其分发给对应的Event Handler处理，即回调EventHandler中的handle\_event()方法。

4）Event Handler：定义事件处理方法：handle\_event()，以供InitiationDispatcher回调使用。

5）Concrete Event Handler：事件EventHandler接口，实现特定事件处理逻辑。

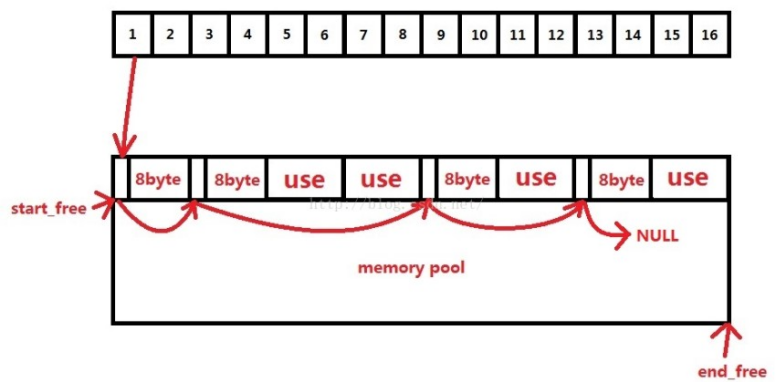
**● 请自己设计一下如何采用单线程的方式处理高并发**

**参考回答：**

在单线程模型中，可以采用I/O复用来提高单线程处理多个请求的能力，然后再采用事件驱动模型，基于异步回调来处理事件来

**● 请你说一说C++ STL 的内存优化**

**参考回答：**

1）二级配置器结构  
STL内存管理使用二级内存配置器。  
1、第一级配置器  
第一级配置器以malloc()，free()，realloc()等C函数执行实际的内存配置、释放、重新配置等操作，并且能在内存需求不被满足的时候，调用一个指定的函数。  
一级空间配置器分配的是大于128字节的空间  
如果分配不成功，调用句柄释放一部分内存  
如果还不能分配成功，抛出异常  
2、第二级配置器  
在STL的第二级配置器中多了一些机制，避免太多小区块造成的内存碎片，小额区块带来的不仅是内存碎片，配置时还有额外的负担。区块越小，额外负担所占比例就越大。  
3、分配原则  
如果要分配的区块大于128bytes，则移交给第一级配置器处理。  
如果要分配的区块小于128bytes，则以内存池管理（memory pool），又称之次层配置（sub-allocation）：每次配置一大块内存，并维护对应的16个空闲链表（free-list）。下次若有相同大小的内存需求，则直接从free-list中取。如果有小额区块被释放，则由配置器回收到free-list中。  
当用户申请的空间小于128字节时，将字节数扩展到8的倍数，然后在自由链表中查找对应大小的子链表  
如果在自由链表查找不到或者块数不够，则向内存池进行申请，一般一次申请20块  
如果内存池空间足够，则取出内存  
如果不够分配20块，则分配最多的块数给自由链表，并且更新每次申请的块数  
如果一块都无法提供，则把剩余的内存挂到自由链表，然后向系统heap申请空间，如果申请失败，则看看自由链表还有没有可用的块，如果也没有，则最后调用一级空间配置器  
2）二级内存池  
二级内存池采用了16个空闲链表，这里的16个空闲链表分别管理大小为8、16、24......120、128的数据块。这里空闲链表节点的设计十分巧妙，这里用了一个联合体既可以表示下一个空闲数据块（存在于空闲链表中）的地址，也可以表示已经被用户使用的数据块（不存在空闲链表中）的地址。  
  
  
1、空间配置函数allocate  
首先先要检查申请空间的大小，如果大于128字节就调用第一级配置器，小于128字节就检查对应的空闲链表，如果该空闲链表中有可用数据块，则直接拿来用（拿取空闲链表中的第一个可用数据块，然后把该空闲链表的地址设置为该数据块指向的下一个地址），如果没有可用数据块，则调用refill重新填充空间。  
2、空间释放函数deallocate  
首先先要检查释放数据块的大小，如果大于128字节就调用第一级配置器，小于128字节则根据数据块的大小来判断回收后的空间会被插入到哪个空闲链表。  
3、重新填充空闲链表refill  
在用allocate配置空间时，如果空闲链表中没有可用数据块，就会调用refill来重新填充空间，新的空间取自内存池。缺省取20个数据块，如果内存池空间不足，那么能取多少个节点就取多少个。  
从内存池取空间给空闲链表用是chunk\_alloc的工作，首先根据end\_free-start\_free来判断内存池中的剩余空间是否足以调出nobjs个大小为size的数据块出去，如果内存连一个数据块的空间都无法供应，需要用malloc取堆中申请内存。  
假如山穷水尽，整个系统的堆空间都不够用了，malloc失败，那么chunk\_alloc会从空闲链表中找是否有大的数据块，然后将该数据块的空间分给内存池（这个数据块会从链表中去除）。  
3、总结：  
1. 使用allocate向内存池请求size大小的内存空间，如果需要请求的内存大小大于128bytes，直接使用malloc。  
2. 如果需要的内存大小小于128bytes，allocate根据size找到最适合的自由链表。  
a. 如果链表不为空，返回第一个node，链表头改为第二个node。  
b. 如果链表为空，使用blockAlloc请求分配node。  
x. 如果内存池中有大于一个node的空间，分配竟可能多的node(但是最多20个)，将一个node返回，其他的node添加到链表中。  
y. 如果内存池只有一个node的空间，直接返回给用户。  
z. 若果如果连一个node都没有，再次向操作系统请求分配内存。  
①分配成功，再次进行b过程。  
②分配失败，循环各个自由链表，寻找空间。  
I. 找到空间，再次进行过程b。  
II. 找不到空间，抛出异常。  
3. 用户调用deallocate释放内存空间，如果要求释放的内存空间大于128bytes，直接调用free。  
4. 否则按照其大小找到合适的自由链表，并将其插入。

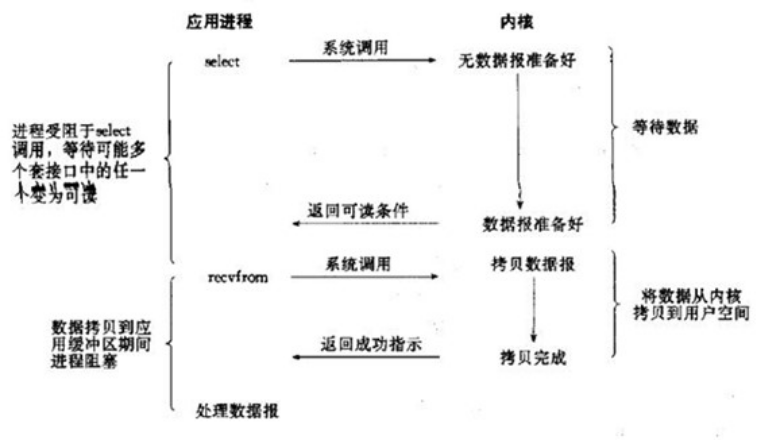
**● 请你说说select，epoll的区别，原理，性能，限制都说一说**

**参考回答：**

1）IO多路复用

IO复用模型在阻塞IO模型上多了一个select函数，select函数有一个参数是文件描述符集合，意思就是对这些的文件描述符进行循环监听，当某个文件描述符就绪的时候，就对这个文件描述符进行处理。

这种IO模型是属于阻塞的IO。但是由于它可以对多个文件描述符进行阻塞监听，所以它的效率比阻塞IO模型高效。



IO多路复用就是我们说的select，poll，epoll。select/epoll的好处就在于单个process就可以同时处理多个网络连接的IO。它的基本原理就是select，poll，epoll这个function会不断的轮询所负责的所有socket，当某个socket有数据到达了，就通知用户进程。

当用户进程调用了select，那么整个进程会被block，而同时，kernel会“监视”所有select负责的socket，当任何一个socket中的数据准备好了，select就会返回。这个时候用户进程再调用read操作，将数据从kernel拷贝到用户进程。

所以，I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，select()函数就可以返回。

I/O多路复用和阻塞I/O其实并没有太大的不同，事实上，还更差一些。因为这里需要使用两个system call (select 和 recvfrom)，而blocking IO只调用了一个system call (recvfrom)。但是，用select的优势在于它可以同时处理多个connection。

所以，如果处理的连接数不是很高的话，使用select/epoll的web server不一定比使用multi-threading + blocking IO的web server性能更好，可能延迟还更大。select/epoll的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在IO multiplexing Model中，实际中，对于每一个socket，一般都设置成为non-blocking，但是，如上图所示，整个用户的process其实是一直被block的。只不过process是被select这个函数block，而不是被socket IO给block。

2、select

select：是最初解决IO阻塞问题的方法。用结构体fd\_set来告诉内核监听多个文件描述符，该结构体被称为描述符集。由数组来维持哪些描述符被置位了。对结构体的操作封装在三个宏定义中。通过轮寻来查找是否有描述符要被处理。

存在的问题：

1. 内置数组的形式使得select的最大文件数受限与FD\_SIZE；

2. 每次调用select前都要重新初始化描述符集，将fd从用户态拷贝到内核态，每次调用select后，都需要将fd从内核态拷贝到用户态；

3. 轮寻排查当文件描述符个数很多时，效率很低；

3、poll

poll：通过一个可变长度的数组解决了select文件描述符受限的问题。数组中元素是结构体，该结构体保存描述符的信息，每增加一个文件描述符就向数组中加入一个结构体，结构体只需要拷贝一次到内核态。poll解决了select重复初始化的问题。轮寻排查的问题未解决。

4、epoll

epoll：轮寻排查所有文件描述符的效率不高，使服务器并发能力受限。因此，epoll采用只返回状态发生变化的文件描述符，便解决了轮寻的瓶颈。

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式

1. LT模式

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的。

2. ET模式

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

3、LT模式与ET模式的区别如下：

LT模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll\_wait时，会再次响应应用程序并通知此事件。  
ET模式：当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。

**● 请你说说C++如何处理内存泄漏？**

**参考回答：**

使用varglind，mtrace检测

C++11 最常用的新特性如下：

auto关键字：编译器可以根据初始值自动推导出类型。但是不能用于函数传参以及数组类型的推导

nullptr关键字：nullptr是一种特殊类型的字面值，它可以被转换成任意其它的指针类型；而NULL一般被宏定义为0，在遇到重载时可能会出现问题。

智能指针：C++11新增了std::shared\_ptr、std::weak\_ptr等类型的智能指针，用于解决内存管理的问题。

初始化列表：使用初始化列表来对类进行初始化

右值引用：基于右值引用可以实现移动语义和完美转发，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

atomic原子操作用于多线程资源互斥操作

新增STL容器array以及tuple

**● 请你详细介绍一下C++11中的可变参数模板、右值引用和lambda这几个新特性。**

**参考回答：**

可变参数模板：

C++11的可变参数模板，对参数进行了高度泛化，可以表示任意数目、任意类型的参数，其语法为：在class或typename后面带上省略号”。

例如：

Template<class ... T>

void func(T ... args)

{

cout<<”num is”<<sizeof ...(args)<<endl;

}

func();//args不含任何参数

func(1);//args包含一个int类型的实参

func(1,2.0)//args包含一个int一个double类型的实参

其中T叫做模板参数包，args叫做函数参数包

省略号作用如下：

1）声明一个包含0到任意个模板参数的参数包

2）在模板定义得右边，可以将参数包展成一个个独立的参数

C++11可以使用递归函数的方式展开参数包，获得可变参数的每个值。通过递归函数展开参数包，需要提供一个参数包展开的函数和一个递归终止函数。例如：

#include using namespace std;

// 最终递归函数

void print()

{

cout << "empty" << endl;

}

// 展开函数

template void print(T head, Args... args)

{

cout << head << ","; print(args...);

}

int main()

{

print(1, 2, 3, 4); return 0;

}

参数包Args ...在展开的过程中递归调用自己，没调用一次参数包中的参数就会少一个，直到所有参数都展开为止。当没有参数时就会调用非模板函数printf终止递归过程。

右值引用:

C++中，左值通常指可以取地址，有名字的值就是左值，而不能取地址，没有名字的就是右值。而在指C++11中，右值是由两个概念构成，将亡值和纯右值。纯右值是用于识别临时变量和一些不跟对象关联的值，比如1+3产生的临时变量值，2、true等，而将亡值通常是指具有转移语义的对象，比如返回右值引用T&&的函数返回值等。

C++11中，右值引用就是对一个右值进行引用的类型。由于右值通常不具有名字，所以我们一般只能通过右值表达式获得其引用，比如：

T && a=ReturnRvale();

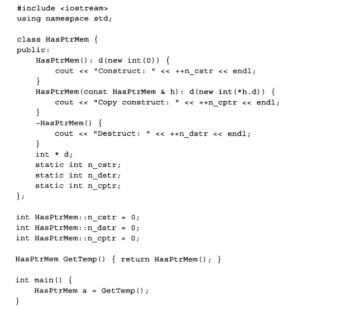
假设ReturnRvalue()函数返回一个右值，那么上述语句声明了一个名为a的右值引用，其值等于ReturnRvalue函数返回的临时变量的值。

基于右值引用可以实现转移语义和完美转发新特性。

移动语义：

对于一个包含指针成员变量的类，由于编译器默认的拷贝构造函数都是浅拷贝，所有我们一般需要通过实现深拷贝的拷贝构造函数，为指针成员分配新的内存并进行内容拷贝，从而避免悬挂指针的问题。

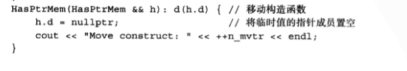
但是如下列代码所示：



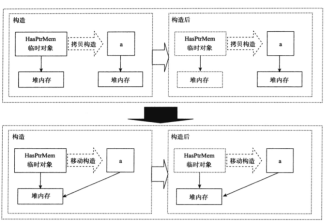
当类HasPtrMem包含一个成员函数GetTemp,其返回值类型是HasPtrMem,如果我们定义了深拷贝的拷贝构造函数，那么在调用该函数时需要调用两次拷贝构造函数。第一次是生成GetTemp函数返回时的临时变量，第二次是将该返回值赋值给main函数中的变量a。与此对应需要调用三次析构函数来释放内存。

而在上述过程中，使用临时变量构造a时会调用拷贝构造函数分配对内存，而临时对象在语句结束后会释放它所使用的堆内存。这样重复申请和释放内存，在申请内存较大时会严重影响性能。因此C++使用移动构造函数，从而保证使用临时对象构造a时不分配内存，从而提高性能。

如下列代码所示，移动构造函数接收一个右值引用作为参数，使用右值引用的参数初始化其指针成员变量。



其原理就是使用在构造对象a时，使用h.d来初始化a，然后将临时对象h的成员变量d指向nullptr，从而保证临时变量析构时不会释放对内存。



完美转发：

完美转发是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另一个函数，即传入转发函数的是左值对象，目标函数就能获得左值对象，转发函数是右值对象，目标函数就能获得右值对象，而不产生额外的开销。

因此转发函数和目标函数参数一般采用引用类型，从而避免拷贝的开销。其次，由于目标函数可能需要能够既接受左值引用，又接受右值引用，所以考虑转发也需要兼容这两种类型。

C++11采用引用折叠的规则，结合新的模板推导规则实现完美转发。其引用折叠规则如下：



因此，我们将转发函数和目标函数的参数都设置为右值引用类型，

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468606591_DA3E71352BDD7421BBDD73F6A044E82E

当传入一个X类型的左值引用时，转发函数将被实例为：

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468630496_031C3EF295357260760C1CC0ADAB487E

经过引用折叠，变为：

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468653093_6E5FA6DE998857AEDC9C15075577E55B

当传入一个X类型的右值引用时，转发函数将被实例为：

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468675273_A23543D482224F954EE8497E93B1C1E9

经过引用折叠，变为：

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468702620_792FD016CA5B45DEFFD7CC690F0DDD3A

除此之外，还可以使用forward()函数来完成左值引用到右值引用的转换：

https://uploadfiles.nowcoder.com/images/20190313/311436_1552468733859_0F098D9684B8C76836890D92B3893DC0

Lambda表达式:

Lambda表达式定义一个匿名函数，并且可以捕获一定范围内的变量，其定义如下：

[capture](params)mutable->return-type{statement}

其中，

[capture]：捕获列表，捕获上下文变量以供lambda使用。同时[]是lambda寅初复，编译器根据该符号来判断接下来代码是否是lambda函数。

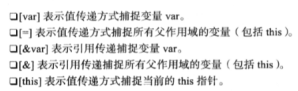
(Params)：参数列表，与普通函数的参数列表一致，如果不需要传递参数，则可以连通括号一起省略。

mutable是修饰符，默认情况下lambda函数总是一个const函数，Mutable可以取消其常量性。在使用该修饰符时，参数列表不可省略。

->return-type:返回类型是返回值类型

{statement}:函数体，内容与普通函数一样，除了可以使用参数之外，还可以使用所捕获的变量。

Lambda表达式与普通函数最大的区别就是其可以通过捕获列表访问一些上下文中的数据。其形式如下:



Lambda的类型被定义为“闭包”的类，其通常用于STL库中，在某些场景下可用于简化仿函数的使用，同时Lambda作为局部函数，也会提高复杂代码的开发加速，轻松在函数内重用代码，无须费心设计接口。