C++面经

1. C++基础

1.1 语言基础

1.1.1 简述下C++语言的特点

参考回答

1. C++在C语言基础上引入了**面对对象**的机制，同时也**兼容C语言。**
2. C++有三大特性（1）封装。（2）继承。（3）多态；
3. C++语言编写出的程序结构清晰、易于扩充，程序**可读性好。**
4. C++生成的代码**质量高**，运行**效率高**，仅比汇编语言慢10%～20%；
5. C++更加安全，增加了const常量、引用、四类cast转换（static_cast、dynamic_cast、const_cast、reinterpret_cast）、智能指针、try—catch等等；
6. C++**可复用性**高，C++引入了**模板**的概念，后面在此基础上，实现了方便开发的标准模板库STL（Standard Template Library）。
7. 同时，C++是**不断在发展**的语言。C++后续版本更是发展了不少新特性，如C++11中引入了nullptr、auto变量、Lambda匿名函数、右值引用、智能指针。

1.1.2 说说C语言和C++的区别

参考回答

1. C语言是C++的子集，C++可以很好兼容C语言。但是C++又有很多**新特性**，如引用、智能指针、auto变量等。
2. C++是**面对对象**的编程语言；C语言是**面对过程**的编程语言。
3. C语言有一些不安全的语言特性，如指针使用的潜在危险、强制转换的不确定性、内存泄露等。而C++对此增加了不少新特性来**改善安全性**，如const常量、引用、cast转换、智能指针、try—catch等等；
4. C++**可复用性**高，C++引入了**模板**的概念，后面在此基础上，实现了方便开发的标准模板库STL。C++的STL库相对于C语言的函数库**更灵活、更通用。**

1.1.3 说说 C++中 struct 和 class 的区别

参考回答

1. struct 一般用于描述一个数据结构集合，而 class 是对一个对象数据的封装；

2. struct 中默认的访问控制权限是 public 的，而 class 中默认的访问控制权限是 private 的，例如：

```
struct A{
    int iNum;    // 默认访问控制权限是 public
}
class B{
    int iNum;    // 默认访问控制权限是 private
}
```

3. 在继承关系中，struct 默认是公有继承，而 class 是私有继承；
4. class 关键字可以用于定义模板参数，就像 typename，而 struct 不能用于定义模板参数，例如：

```
template<typename T, typename Y>    // 可以把typename 换成 class
int Func(const T& t, const Y& y) {
    //TODO
}
```

答案解析

1. C++ 中的 struct 是对 C 中的 struct 进行了扩充，它们在声明时的区别如下：

	C	C++
成员函数	不能有	可以
静态成员	不能有	可以
访问控制	默认public，不能修改	public/private/protected
继承关系	不可以继承	可从类或者其他结构体继承
初始化	不能直接初始化数据成员	可以

2. 使用时的区别：C 中使用结构体需要加上 struct 关键字，或者对结构体使用 typedef 取别名，而 C++ 中可以省略 struct 关键字直接使用，例如：

```
struct Student{
    int  iAgeNum;
    string strName;
}
typedef struct Student Student2;    //C中取别名

struct Student stu1;    // C 中正常使用
Student2 stu2;            // C 中通过取别名的使用
Student stu3;            // C++ 中使用
```

1.1.4 说说include头文件的顺序以及双引号""和尖括号<>的区别

参考回答

1. 区别：

（1）尖括号<>的头文件是**系统文件**，双引号""的头文件是**自定义文件**。

（2）编译器预处理阶段查找头文件的路径不一样。

2. 查找路径：

（1）使用尖括号<>的头文件的查找路径：编译器设置的头文件路径-->系统变量。

（2）使用双引号""的头文件的查找路径：当前头文件目录-->编译器设置的头文件路径-->系统变量。

1.1.5 说说C++结构体和C结构体的区别

参考回答

区别：

（1）C的结构体内不允许有函数存在，C++允许有内部成员函数，且允许该函数是虚函数。

（2）C的结构体对内部成员变量的访问权限只能是public，而C++允许public,protected,private三种。

（3）C语言的结构体是不可以继承的，C++的结构体是可以从其他的结构体或者类继承过来的。

（4）C 中使用结构体需要加上 struct 关键字，或者对结构体使用 typedef 取别名，而 C++ 中可以省略 struct 关键字直接使用。

答案解析

1. C++ 中的 struct 是对 C 中的 struct 进行了扩充，它们在声明时的区别如下：

	C	C++
成员函数	不能有	可以
静态成员	不能有	可以
访问控制	默认public，不能修改	public/private/protected
继承关系	不可以继承	可从类或者其他结构体继承
初始化	不能直接初始化数据成员	可以

2. 使用时的区别：C 中使用结构体需要加上 struct 关键字，或者对结构体使用 typedef 取别名，而 C++ 中可以省略 struct 关键字直接使用，例如：

```
struct Student{
    int  iAgeNum;
    string strName;
}
typedef struct Student Student2;     //C中取别名

struct Student stu1;     // C 中正常使用
Student2 stu2;             // C 中通过取别名的使用
Student stu3;             // C++ 中使用
```

1.1.6 导入C函数的关键字是什么，C++编译时和C有什么不同?

参考回答

1. **关键字：**在C++中，导入C函数的关键字是**extern**，表达形式为**extern “C”**， extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代

码按**C语言**的进行编译，而不是C++的。

2. **编译区别：**由于C++支持函数重载，因此编译器编译函数的过程中会将函数的**参数类型**也加到编译后的代码中，而不仅仅是**函数名**；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括**函数名**。

答案解析

```
//extern示例
//在C++程序里边声明该函数，会指示编译器这部分代码按C语言的进行编译
extern "C" int strcmp(const char *s1, const char *s2);

//在C++程序里边声明该函数
extern "C"{
    #include <string.h>//string.h里边包含了要调用的C函数的声明
}

//两种不同的语言，有着不同的编译规则，比如一个函数fun，可能C语言编译的时候为_fun，而C++则是__fun__
```

1.1.7 简述C++从代码到可执行二进制文件的过程

参考回答

C++和C语言类似，一个C++程序从源码到执行文件，有四个过程，**预编译、编译、汇编、链接。**

答案解析

1. 预编译：这个过程主要的处理操作如下：
   （1） 将所有的#define删除，并且展开所有的宏定义
   （2） 处理所有的条件预编译指令，如#if、#ifdef
   （3） 处理#include预编译指令，将被包含的文件插入到该预编译指令的位置。
   （4） 过滤所有的注释
   （5） 添加行号和文件名标识。
2. 编译：这个过程主要的处理操作如下：
   （1） 词法分析：将源代码的字符序列分割成一系列的记号。
   （2） 语法分析：对记号进行语法分析，产生语法树。
   （3） 语义分析：判断表达式是否有意义。
   （4） 代码优化：
   （5） 目标代码生成：生成汇编代码。
   （6） 目标代码优化：
3. 汇编：这个过程主要是将汇编代码转变成机器可以执行的指令。
4. 链接：将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。

   链接分为静态链接和动态链接。

   静态链接，是在链接的时候就已经把要调用的函数或者过程链接到了生成的可执行文件中，就算你在去把静态库删除也不会影响可执行程序的执行；生成的静态链接库，Windows下以.lib为后缀，Linux下以.a为后缀。

而动态链接，是在链接的时候没有把调用的函数代码链接进去，而是在执行的过程中，再去找要链接的函数，生成的可执行文件中没有函数代码，只包含函数的重定位信息，所以当你删除动态库时，可执行程序就不能运行。生成的动态链接库，Windows下以.dll为后缀，Linux下以.so为后缀。

1.1.8 说说 static关键字的作用

参考回答

1. **定义全局静态变量和局部静态变量**：在变量前面加上static关键字。初始化的静态变量会在数据段分配内存，未初始化的静态变量会在BSS段分配内存。直到程序结束，静态变量始终会维持前值。只不过全局静态变量和局部静态变量的作用域不一样；
2. **定义静态函数**：在函数返回类型前加上static关键字，函数即被定义为静态函数。静态函数只能在**本源文件**中使用；
3. 在变量类型前加上static关键字，变量即被定义为静态变量。**静态变量只能在本源文件中使用**；

```
//示例
static int a;
static void func();
```

4. 在c++中，**static关键字可以用于定义类中的静态成员变量**：使用静态数据成员，它既可以被当成全局变量那样去存储，但又被隐藏在类的内部。类中的static静态数据成员拥有一块单独的存储区，而不管创建了多少个该类的对象。所有这些对象的静态数据成员都**共享**这一块静态存储空间。
5. 在c++中，**static关键字可以用于定义类中的静态成员函数**：与静态成员变量类似，类里面同样可以定义静态成员函数。只需要在函数前加上关键字static即可。如静态成员函数也是类的一部分，而不是对象的一部分。所有这些对象的静态数据成员都**共享**这一块静态存储空间。

答案解析

当调用一个对象的非静态成员函数时，系统会把该对象的起始地址赋给成员函数的this指针。而静态成员函数不属于任何一个对象，因此C++规定静态成员函数没有this指针（划重点，面试题常考）。既然它没有指向某一对象，也就无法对一个对象中的非静态成员进行访问。

1.1.9 说说数组和指针的区别

参考回答

1. 概念：

   （1）数组：数组是用于储存多个相同类型数据的集合。 数组名是首元素的地址。

   （2）指针：指针相当于一个变量，但是它和不同变量不一样，它存放的是其它变量在**内存中的地址**。 指针名指向了内存的首地址。

2. 区别：

   （1）**赋值**：同类型指针变量可以相互赋值；数组不行，只能一个一个元素的赋值或拷贝

   （2）**存储方式**：

   数组：数组在内存中是连续存放的，开辟一块连续的内存空间。数组是根据数组的下进行访问的，数组的存储空间，不是在静态区就是在栈上。

   指针：指针很灵活，它可以指向任意类型的数据。指针的类型说明了它所指向地址空间的内存。由于指针本身就是一个变量，再加上它所存放的也是变量，所以指针的存储空间不能确定。

（3）**求sizeof**：

数组所占存储空间的内存大小：sizeof（数组名）/sizeof（数据类型）

在32位平台下，无论指针的类型是什么，sizeof（指针名）都是4，在64位平台下，无论指针的类型是什么，sizeof（指针名）都是8。

（4）**初始化**：

```
//数组
int a[5] = {0};
char b[]={"Hello"};//按字符串初始化，大小为6.
char c[]={'H','e','l','l','o','\0'};//按字符初始化
int* arr = new int[n];//创建一维数组

//指针
//指向对象的指针
int p=new int(0) ;
delete p;
//指向数组的指针
int p=new int[n];
delete[] p;
//指向类的指针：
class p=new class;
delete p;
//指针的指针（二级指针）
int **pp=new (int)[1];
pp[0]=new int[6];
delete[] pp[0];
```

（5）指针操作：

数组名的指针操作

```
int a[3][4];
int (*p)[4];  //该语句是定义一个数组指针，指向含4个元素的一维数组
p = a;        //将该二维数组的首地址赋给p，也就是a[0]或&a[0][0]
p++;          //该语句执行过后，也就是p=p+1；p跨过行a[0][]指向了行a[1][]
              //所以数组指针也称指向一维数组的指针，亦称行指针。
//访问数组中第i行j列的一个元素，有几种操作方式：
//*(p[i]+j)、*(*(p+i)+j)、(*(p+i))[j]、p[i][j]。其中，优先级：()>[]>*。
//这几种操作方式都是合法的。
```

指针变量的数据操作：

```
char *str = "hello,douya!";
str[2] = 'a';
*(str+2) = 'b';
//这两种操作方式都是合法的。
```

1.1.10 说说什么是函数指针，如何定义函数指针，有什么使用场景

参考回答

1. **概念：**函数指针就是**指向函数**的指针变量。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。

2. **定义**形式如下：

```
int func(int a);
int (*f)(int a);
f = &func;
```

1. 函数指针的**应用场景**：**回调**（callback）。我们调用别人提供的 API函数(Application Programming Interface,应用程序编程接口)，称为Call；如果别人的库里面调用我们的函数，就叫Callback。

答案解析

```
//以库函数qsort排序函数为例，它的原型如下：
void qsort(void *base,//void*类型，代表原始数组
           size_t nmemb, //第二个是size_t类型，代表数据数量
           size_t size, //第三个是size_t类型，代表单个数据占用空间大小
           int(*compar)(const void *,const void *)//第四个参数是函数指针
          );
//第四个参数告诉qsort，应该使用哪个函数来比较元素，即只要我们告诉qsort比较大小的规则，它就可以帮我们对任意数据类型的数组进行排序。在库函数qsort调用我们自定义的比较函数，这就是回调的应用。

//示例
int num[100];
int cmp_int(const void* _a , const void* _b){//参数格式固定
    int* a = (int*)_a;    //强制类型转换
    int* b = (int*)_b;
    return *a - *b;
}

qsort(num,100,sizeof(num[0]),cmp_int); //回调
```

1.1.11 说说静态变量什么时候初始化?

参考回答

对于C语言的全局和静态变量，初始化发生在任何代码执行之前，属于编译期初始化。

而C++标准规定：全局或静态对象当且仅当对象首次用到时才进行构造。

答案解析

1. **作用域**：C++里作用域可分为6种：全局，局部，类，语句，命名空间和文件作用域。

   静态全局变量 ： 全局作用域+文件作用域，所以无法在其他文件中使用。

   静态局部变量 ： 局部作用域，只被初始化一次，直到程序结束。

   类静态成员变量：类作用域。

2. **所在空间**：都在静态存储区。因为静态变量都在静态存储区，所以下次调用函数的时候还是能取到原来的值。
3. **生命周期**：静态全局变量、静态局部变量都在静态存储区，直到程序结束才会回收内存。类静态成员变量在静态存储区，当超出类作用域时回收内存。

1.1.12 nullptr调用成员函数可以吗？为什么？

参考回答

能。

原因：因为在**编译时对象**就绑定了**函数地址**，和指针空不空没关系。

答案解析

```
//给出实例
class animal{
public:
    void sleep(){ cout << "animal sleep" << endl; }
    void breathe(){ cout << "animal breathe haha" << endl; }
};
class fish :public animal{
public:
    void breathe(){ cout << "fish bubble" << endl; }
};
int main(){
    animal *pAn=nullptr;
    pAn->breathe();   // 输出：animal breathe haha
    fish *pFish = nullptr;
    pFish->breathe(); // 输出：fish bubble
    return 0;
}
```

原因：因为在**编译时对象**就绑定了**函数地址**，和指针空不空没关系。pAn->breathe();编译的时候，函数的地址就和指针pAn绑定了；调用breath(*this), this就等于pAn。由于函数中没有需要解引用this的地方，所以函数运行不会出错，但是若用到this，因为this=nullptr，运行出错。

1.1.13 说说什么是野指针，怎么产生的，如何避免？

参考回答

1. **概念：**野指针就是指针指向的位置是不可知的（随机的、不正确的、没有明确限制的）
2. **产生原因**：释放内存后指针不及时置空（野指针），依然指向了该内存，那么可能出现非法访问的错误。这些我们都要注意避免。
3. **避免办法：**

   （1）初始化置NULL

   （2）申请内存后判空

   （3）指针释放后置NULL

   （4）使用智能指针

答案解析

产生原因：释放内存后指针不及时置空（野指针），依然指向了该内存，那么可能出现非法访问的错误。这些我们都要注意避免。如：

```
char *p = (char *)malloc(sizeof(char)*100);
strcpy(p, "Douya");
free(p);//p所指向的内存被释放，但是p所指的地址仍然不变
...
if (p != NULL){//没有起到防错作用
    strcpy(p, "hello, Douya!");//出错
}
```

避免办法：

（1）初始化置NULL

（2）申请内存后判空

（3）指针释放后置NULL

```
int *p = NULL; //初始化置NULL
p = (int *)malloc(sizeof(int)*n); //申请n个int内存空间
assert(p != NULL); //判空，防错设计
p = (int *) realloc(p, 25);//重新分配内存，p 所指向的内存块会被释放并分配一个新的内存地址
free(p);
p = NULL; //释放后置空

int *p1 = NULL; //初始化置NULL
p1 = (int *)calloc(n, sizeof(int)); //申请n个int内存空间同时初始化为0
assert(p1 != NULL); //判空，防错设计
free(p1);
p1 = NULL; //释放后置空

int *p2 = NULL; //初始化置NULL
p2 = new int[n]; //申请n个int内存空间
assert(p2 != NULL); //判空，防错设计
delete []p2;
p2 = nullptr; //释放后置空
```

1.1.14 说说静态局部变量，全局变量，局部变量的特点，以及使用场景

参考回答

1. **首先从作用域考虑**：C++里作用域可分为6种：全局，局部，类，语句，命名空间和文件作用域。

   全局变量：全局作用域，可以通过extern作用于其他非定义的源文件。

   静态全局变量：全局作用域+文件作用域，所以无法在其他文件中使用。

   局部变量：局部作用域，比如函数的参数，函数内的局部变量等等。

   静态局部变量：局部作用域，只被初始化一次，直到程序结束。

2. **从所在空间考虑**：除了局部变量在栈上外，其他都在静态存储区。因为静态变量都在静态存储区，所以下次调用函数的时候还是能取到原来的值。

3. **生命周期**：局部变量在栈上，出了作用域就回收内存；而全局变量、静态全局变量、静态局部变量都在静态存储区，直到程序结束才会回收内存。

4. **使用场景**：从它们各自特点就可以看出各自的应用场景，不再赘述。

1.1.15 说说内联函数和宏函数的区别

参考回答

区别：

1. **宏定义不是函数**，但是使用起来像函数。预处理器用复制宏代码的方式代替函数的调用，省去了函数压栈退栈过程，提高了效率；**而内联函数本质上是一个函数**，内联函数一般用于函数体的代码比较简单的函数，不能包含复杂的控制语句，while、switch，并且内联函数本身不能直接调用自身。
2. **宏函数**是在预编译的时候把所有的宏名用宏体来替换，简单的说就是字符串替换；**而内联函数**则是在编译的时候进行代码插入，编译器会在每处调用内联函数的地方直接把内联函数的内容展开，这样可以省去函数的调用的开销，提高效率
3. **宏定义**是没有类型检查的，无论对还是错都是直接替换；**而内联函数**在编译的时候会进行类型的检查，内联函数满足函数的性质，比如有返回值、参数列表等

答案解析

```
//宏定义示例
#define MAX(a, b) ((a)>(b)?(a):(b))
MAX(a,"Hello");  //错误地比较int和字符串，没有参数类型检查

//内联函数示例
#include <stdio.h>
inline int add(int a, int b){
    return (a + b);
}
int main(void){
    int a;
    a = add(1, 2);
    printf("a+b=%d\n", a);
    return 0;
}
//以上a = add(1, 2);处在编译时将被展开为：a = (a + b);
```

1、使用时的一些注意事项：

- 使用宏定义一定要注意错误情况的出现，比如宏定义函数没有类型检查，可能传进来任意类型，从而带来错误，如举例。还有就是括号的使用，宏在定义时要小心处理宏参数，一般用括号括起来，否则容易出现二义性
- inline函数一般用于比较小的，频繁调用的函数，这样可以减少函数调用带来的开销。只需要在函数返回类型前加上关键字inline，即可将函数指定为inline函数。
- 同其它函数不同的是，最好将inline函数定义在头文件，而不仅仅是声明，因为编译器在处理inline函数时，需要在调用点内联展开该函数，所以仅需要函数声明是不够的。

2、内联函数使用的条件：

- 内联是以代码膨胀（复制）为代价，仅仅省去了函数调用的开销，从而提高函数的执行效率。如果执行函数体内代码的时间，相比于函数调用的开销较大，那么效率 的收获会很少。另一方面，每一处内联函数的调用都要复制代码，将使程序的总代码量增大，消耗更多的内存空间。以下情况不宜使用内联：
- （1）如果函数体内的代码比较长，使用内联将导致内存消耗代价较高。
- （2）如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大。
- 内联不是什么时候都能展开的，一个好的编译器将会根据函数的定义体，自动地取消不符合要求的内联。

1.1.16 说说运算符i++和++i的区别

参考回答

先看到实现代码：

```
#include <stdio.h>
int main(){
        int i = 2;
    int j = 2;
    j += i++; //先赋值后加
    printf("i= %d, j= %d\n",i, j); //i= 3, j= 4
    i = 2;
    j = 2;
    j += ++i; //先加后赋值
    printf("i= %d, j= %d",i, j); //i= 3, j= 5
}
```

1. **赋值顺序不同**：++ i 是先加后赋值；i ++ 是先赋值后加；++i和i++都是分两步完成的。
2. **效率不同**：后置++执行速度比前置的慢。
3. **i++ 不能作为左值，而++i 可以**：

```
int i = 0;
int *p1 = &（++i）; //正确
int *p2 = &（i++）; //错误
++i = 1; //正确
i++ = 1; //错误
```

4. 两者都不是原子操作。

1.1.17 说说new和malloc的区别，各自底层实现原理。

参考回答

1. new是操作符，而malloc是函数。
2. new在调用的时候先分配内存，在调用构造函数，释放的时候调用析构函数；而malloc没有构造函数和析构函数。
3. 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，不用指定内存的大小，返回指针不用强转。
4. new可以被重载；malloc不行
5. new分配内存更直接和安全。
6. new发生错误抛出异常，malloc返回null

答案解析

malloc底层实现：当开辟的空间小于 128K 时，调用 brk（）函数；当开辟的空间大于 128K 时，调用 mmap（）。malloc采用的是内存池的管理方式，以减少内存碎片。先申请大块内存作为堆区，然后将堆区分为多个内存块。当用户申请内存时，直接从堆区分配一块合适的空闲快。采用隐式链表将所有空闲块，每一个空闲块记录了一个未分配的、连续的内存地址。

new底层实现：关键字new在调用构造函数的时候实际上进行了如下的几个步骤：

1. 创建一个新的对象

2. 将构造函数的作用域赋值给这个新的对象（因此this指向了这个新的对象）
3. 执行构造函数中的代码（为这个新对象添加属性）
4. 返回新对象

1.1.18 说说const和define的区别。

参考回答

const用于定义常量；而define用于定义宏，而宏也可以用于定义常量。都用于常量定义时，它们的区别有：

1. const生效于编译的阶段；define生效于预处理阶段。
2. const定义的常量，在C语言中是存储在内存中、需要额外的内存空间的；define定义的常量，运行时是直接的操作数，并不会存放在内存中。
3. const定义的常量是带类型的；define定义的常量不带类型。因此define定义的常量不利于类型检查。

1.1.19 说说C++中函数指针和指针函数的区别。

参考回答

1. **定义不同**
   指针函数本质是一个函数，其返回值为指针。
   函数指针本质是一个指针，其指向一个函数。
2. **写法不同**

```
指针函数：int *fun(int x,int y);
函数指针：int (*fun)(int x,int y);
```

3. **用法不同**

   用法参考答案解析

答案解析

```
//指针函数示例
typedef struct _Data{
    int a;
    int b;
}Data;
//指针函数
Data* f(int a,int b){
    Data * data = new Data;
    //...
    return data;
}
int main(){
    //调用指针函数
    Data * myData = f(4,5);
    //Data * myData = static_cast<Data*>(f(4,5));
   //...
}
```

```
18
19  //函数指针示例
20  int add(int x,int y){
21      return x+y;
22  }
23  //函数指针
24  int (*fun)(int x,int y);
25  //赋值
26  fun = add;
27  //调用
28  cout << "(*fun)(1,2) = " << (*fun)(1,2) ;
29  //输出结果
30  //(*fun)(1,2) =  3
```

1.1.20 说说const int *a, int const *a, const int a, int *const a, const int *const a分别是什么，有什么特点。

参考回答

```
1  1. const int a;      //指的是a是一个常量，不允许修改。
2  2. const int *a;     //a指针所指向的内存里的值不变，即（*a）不变
3  3. int const *a;     //同const int *a;
4  4. int *const a;     //a指针所指向的内存地址不变，即a不变
5  5. const int *const a;    //都不变，即（*a）不变，a也不变
```

1.1.21 说说使用指针需要注意什么?

参考回答

1. 定义指针时，先初始化为NULL。
2. 用malloc或new申请内存之后，应该**立即检查**指针值是否为NULL。防止使用指针值为NULL的内存。
3. 不要忘记为数组和动态内存**赋初值**。防止将未被初始化的内存作为右值使用。
4. 避免数字或指针的下标**越界**，特别要当心发生“多1”或者“少1”操作
5. 动态内存的申请与释放必须配对，防止**内存泄漏**
6. 用free或delete释放了内存之后，立即将指针**设置为NULL**，防止“野指针”

答案解析

（1）初始化置NULL

（2）申请内存后判空

（3）指针释放后置NULL

```
1  int *p = NULL; //初始化置NULL
2  p = (int *)malloc(sizeof(int)*n); //申请n个int内存空间
3  assert(p != NULL); //判空，防错设计
4  p = (int *) realloc(p, 25);//重新分配内存，p 所指向的内存块会被释放并分配一个新的内存地址
5  free(p);
6  p = NULL; //释放后置空
7
8  int *p1 = NULL; //初始化置NULL
```

```
p1 = (int *)calloc(n, sizeof(int)); //申请n个int内存空间同时初始化为0
assert(p1 != NULL); //判空，防错设计
free(p1);
p1 = NULL; //释放后置空

int *p2 = NULL; //初始化置NULL
p2 = new int[n]; //申请n个int内存空间
assert(p2 != NULL); //判空，防错设计
delete []p2;
p2 = nullptr; //释放后置空
```

1.1.22 说说内联函数和函数的区别，内联函数的作用。

参考回答

1. 内联函数比普通函数多了关键字**inline**
2. 内联函数避免了函数调用的**开销**；普通函数有调用的开销
3. 普通函数在被调用的时候，需要**寻址（函数入口地址）**；内联函数不需要寻址。
4. 内联函数有一定的限制，内联函数体要求**代码简单**，不能包含复杂的结构控制语句；普通函数没有这个要求。

内联函数的作用：内联函数在调用时，是将调用表达式用内联函数体来替换。避免函数调用的开销。

答案解析

在使用内联函数时，应注意如下几点：

1. 在内联函数内不允许用循环语句和开关语句。
   如果内联函数有这些语句，则编译将该函数视同普通函数那样产生函数调用代码,递归函数是不能被用来做内联函数的。内联函数只适合于只有1～5行的小函数。对一个含有许多语句的大函数，函数调用和返回的开销相对来说微不足道，所以也没有必要用内联函数实现。
2. 内联函数的定义必须出现在内联函数第一次被调用之前。

1.1.23 简述C++有几种传值方式，之间的区别是什么?

参考回答

传参方式有这三种：**值传递、引用传递、指针传递**

1. 值传递：形参即使在函数体内值发生变化，也不会影响实参的值；
2. 引用传递：形参在函数体内值发生变化，会影响实参的值；
3. 指针传递：在指针指向没有发生改变的前提下，形参在函数体内值发生变化，会影响实参的值；

答案解析

值传递用于对象时，整个对象会拷贝一个副本，这样效率低；而引用传递用于对象时，不发生拷贝行为，只是绑定对象，更高效；指针传递同理，但不如引用传递安全。

代码示例

```
//代码示例
#include <iostream>
using namespace std;

```

```
void testfunc(int a, int *b, int &c){//形参a值发生了改变，但是没有影响实参i的值；但形参*b、c的值发生了改变，影响到了实参*j、k的值
    a += 1;
    (*b) += 1;
    c += 1;
    printf("a= %d, b= %d, c= %d\n",a,*b,c);//a= 2, b= 2, c= 2
}
int main(){
       int i = 1;
    int a = 1;
    int *j = &a;
    int k = 1;
    testfunc(i, j, k);
    printf("i= %d, j= %d, k= %d\n",i,*j,k);//i= 1, j= 2, k= 2
    return 0;
}
```

1.1.24 简述const（星号）和（星号）const的区别

参考回答

```
//const* 是指针常量，*const 是常量指针

int const *a;    //a指针所指向的内存里的值不变，即（*a）不变
int *const a;    //a指针所指向的内存地址不变，即a不变
```

答案解析

无。

1.2 C++内存

1.2.1 简述一下堆和栈的区别

参考回答

区别：

1. **堆栈空间分配不同。**栈由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等；堆一般由程序员分配释放。
2. **堆栈缓存方式不同。**栈使用的是一级缓存， 它们通常都是被调用时处于存储空间中，调用完毕立即释放；堆则是存放在二级缓存中，速度要慢些。
3. **堆栈数据结构不同。**堆类似数组结构；栈类似栈结构，先进后出。

1.2.2 简述C++的内存管理

参考回答

1. **内存分配方式**：

   在C++中，内存分成5个区，他们分别是堆、栈、自由存储区、全局/静态存储区和常量存储区。

栈，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。

堆，就是那些由new分配的内存块，一般一个new就要对应一个delete。

自由存储区，就是那些由malloc等分配的内存块，和堆是十分相似的，不过是用free来结束自己的生命。

全局/静态存储区，全局变量和静态变量被分配到同一块内存中

常量存储区，这是一块比较特殊的存储区，里面存放的是常量，不允许修改。

2. **常见的内存错误及其对策**：

   （1）内存分配未成功，却使用了它。

   （2）内存分配虽然成功，但是尚未初始化就引用它。

   （3）内存分配成功并且已经初始化，但操作越过了内存的边界。

   （4）忘记了释放内存，造成内存泄露。

   （5）释放了内存却继续使用它。

   对策：

   （1）定义指针时，先初始化为NULL。

   （2）用malloc或new申请内存之后，应该**立即检查**指针值是否为NULL。防止使用指针值为NULL的内存。

   （3）不要忘记为数组和动态内存**赋初值**。防止将未被初始化的内存作为右值使用。

   （4）避免数字或指针的下标**越界**，特别要当心发生“多1”或者“少1”操作

   （5）动态内存的申请与释放必须配对，防止**内存泄漏**

   （6）用free或delete释放了内存之后，立即将指针**设置为NULL**，防止“野指针”

   （7）使用智能指针。

3. **内存泄露及解决办法**：

   什么是内存泄露?

   简单地说就是申请了一块内存空间，使用完毕后没有释放掉。（1）new和malloc申请资源使用后，没有用delete和free释放；（2）子类继承父类时，父类析构函数不是虚函数。（3）Windows句柄资源使用后没有释放。

   怎么检测?

   第一：良好的编码习惯，使用了内存分配的函数，一旦使用完毕,要记得使用其相应的函数释放掉。

   第二：将分配的内存的指针以链表的形式自行管理，使用完毕之后从链表中删除，程序结束时可检查改链表。

   第三：使用智能指针。

   第四：一些常见的工具插件，如ccmalloc、Dmalloc、Leaky、Valgrind等等。

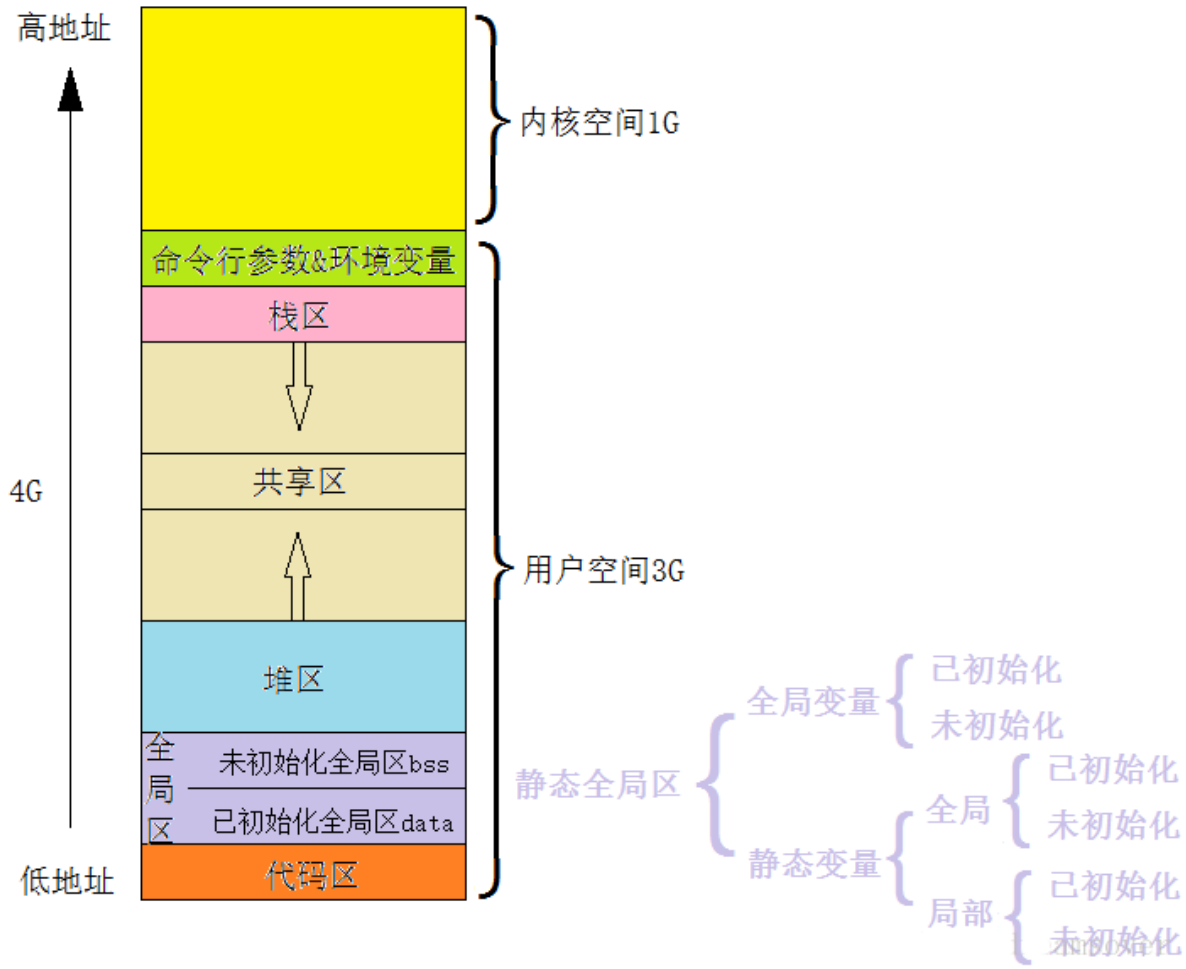
1.2.3 malloc和局部变量分配在堆还是栈?

参考回答

malloc是在**堆上分配内存**，需要程序员自己回收内存；局部变量是在**栈中分配内存**，超过作用域就自动回收。

1.2.4 程序有哪些section，分别的作用？程序启动的过程？怎么判断数据分配在栈上还是堆上？

参考回答



一个程序有哪些section：

如上图，**从低地址到高地址，一个程序由代码段、数据段、 BSS 段组成。**

1. **数据段：**存放程序中已初始化的全局变量和静态变量的一块内存区域。
2. **代码段：**存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。
3. **BSS** 段：存放程序中未初始化的全局变量和静态变量的一块内存区域。
4. 可执行程序在运行时又会多出两个区域：堆区和栈区。

   堆区：动态申请内存用。堆从低地址向高地址增长。

   栈区：存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。
5. 最后还有一个**文件映射区**，位于堆和栈之间。

程序启动的过程：

1. 操作系统首先创建相应的进程并分配私有的进程空间，然后操作系统的加载器负责把可执行文件的数据段和代码段映射到进程的虚拟内存空间中。
2. 加载器读入可执行程序的导入符号表，根据这些符号表可以查找出该可执行程序的所有依赖的动态链接库。
3. 加载器针对该程序的每一个动态链接库调用LoadLibrary
   （1）查找对应的动态库文件，加载器为该动态链接库确定一个合适的基地址。
   （2）加载器读取该动态链接库的导入符号表和导出符号表，比较应用程序要求的导入符号是否匹配该库的导出符号。

（3）针对该库的导入符号表，查找对应的依赖的动态链接库，如有跳转，则跳到3
（4）调用该动态链接库的初始化函数
4. 初始化应用程序的全局变量，对于全局对象自动调用构造函数。
5. 进入应用程序入口点函数开始执行。

怎么判断数据分配在栈上还是堆上：首先局部变量分配在栈上；而通过malloc和new申请的空间是在堆上。

1.2.5 初始化为0的全局变量在bss还是data

参考回答

BSS段通常是指用来存放程序中未初始化的或者初始化为0的全局变量和静态变量的一块内存区域。特点是可读写的，在程序执行之前BSS段会自动清0。

1.2.6 什么是内存泄露，内存泄露怎么检测?

参考回答

什么是内存泄露?

简单地说就是申请了一块内存空间，使用完毕后没有释放掉。（1）new和malloc申请资源使用后，没有用delete和free释放；（2）子类继承父类时，父类析构函数不是虚函数。（3）Windows句柄资源使用后没有释放。

怎么检测?

第一：良好的编码习惯，使用了内存分配的函数，一旦使用完毕,要记得使用其相应的函数释放掉。

第二：将分配的内存的指针以链表的形式自行管理，使用完毕之后从链表中删除，程序结束时可检查改链表。

第三：使用智能指针。

第四：一些常见的工具插件，如ccmalloc、Dmalloc、Leaky、Valgrind等等。

1.2.7 请简述一下atomoic内存顺序。

参考回答

有六个内存顺序选项可应用于对原子类型的操作：

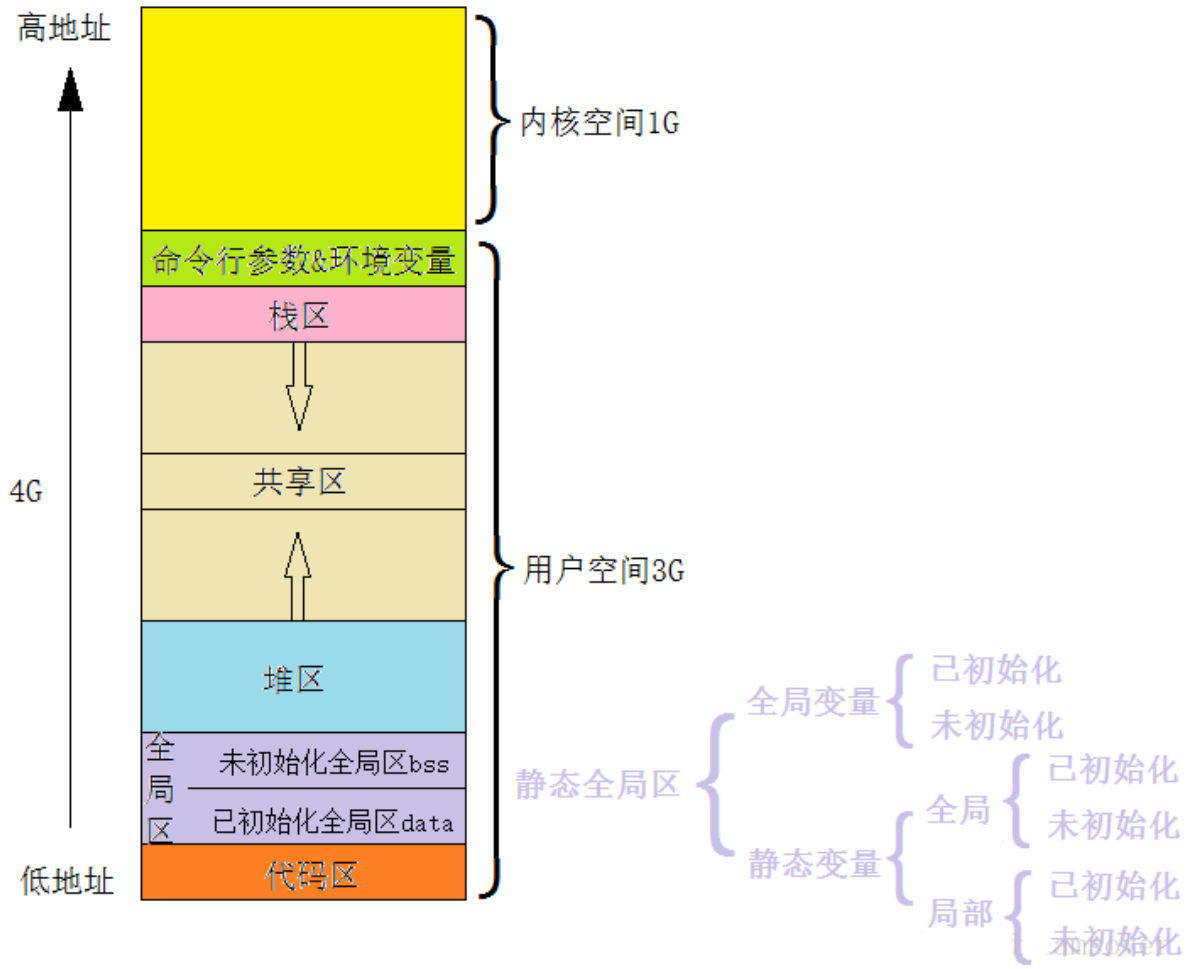
1. memory_order_relaxed：在原子类型上的操作以自由序列执行，没有任何同步关系，仅对此操作要求原子性。
2. memory_order_consume：memory_order_consume只会对其标识的对象保证该对象存储先行于那些需要加载该对象的操作。
3. memory_order_acquire：使用memory_order_acquire的原子操作，当前线程的读写操作都不能重排到此操作之前。
4. memory_order_release：使用memory_order_release的原子操作，当前线程的读写操作都不能重排到此操作之后。
5. memory_order_acq_rel：memory_order_acq_rel在此内存顺序的读-改-写操作既是获得加载又是释放操作。没有操作能够从此操作之后被重排到此操作之前，也没有操作能够从此操作之前被重排到此操作之后。

6. memory_order_seq_cst：memory_order_seq_cst比std::memory_order_acq_rel更为严格。memory_order_seq_cst不仅是一个"获取释放"内存顺序，它还会对所有拥有此标签的内存操作建立一个单独全序。

除非你为特定的操作指定一个顺序选项，否则内存顺序选项对于所有原子类型默认都是memory_order_seq_cst。

1.2.8 内存模型，堆栈，常量区。

参考回答



内存模型（内存布局）：

如上图，**从低地址到高地址，一个程序由代码段、数据段、 BSS 段组成。**

1. **数据段：**存放程序中已初始化的全局变量和静态变量的一块内存区域。
2. **代码段：**存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。
3. **BSS** 段：存放程序中未初始化的全局变量和静态变量的一块内存区域。
4. 可执行程序在运行时又会多出两个区域：堆区和栈区。

   堆区：动态申请内存用。堆从低地址向高地址增长。

   栈区：存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。
5. 最后还有一个**文件映射区**，位于堆和栈之间。

堆 heap： 由new分配的内存块，其释放由程序员控制（一个new对应一个delete）

栈 stack： 是那些编译器在需要时分配，在不需要时自动清除的存储区。存放局部变量、函数参数。

常量存储区：存放常量，不允许修改。

1.2.9 简述C++中内存对齐的使用场景

参考回答

内存对齐应用于三种数据类型中：**struct/class/union**

struct/class/union内存对齐原则有四个：

1. 数据成员对齐规则：结构(struct)或联合(union)的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小的整数倍开始。
2. 结构体作为成员:如果一个结构里有某些结构体成员,则结构体成员要从其内部"最宽基本类型成员"的整数倍地址开始存储。(struct a里存有struct b,b里有char,int ,double等元素,那b应该从8的整数倍开始存储)。
3. 收尾工作:结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的"最宽基本类型成员"的整数倍。不足的要补齐。(基本类型不包括struct/class/uinon)。
4. sizeof(union)，以结构里面size最大元素为union的size，因为在某一时刻，union只有一个成员真正存储于该地址。

答案解析

1. **什么是内存对齐?**

   那么什么是字节对齐？在C语言中，结构体是一种复合数据类型，其构成元素既可以是基本数据类型（如int、long、float等）的变量，也可以是一些复合数据类型（如数组、结构体、联合体等）的数据单元。在结构体中，**编译器为结构体的每个成员按其自然边界（alignment）分配空间。**各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构体的地址相同。

   为了使CPU能够对变量进行快速的访问，变量的起始地址应该具有某些特性，**即所谓的“对齐”，比如4字节的int型，其起始地址应该位于4字节的边界上，即起始地址能够被4整除**，也即“对齐”跟数据在内存中的位置有关。如果一个变量的内存地址正好位于它长度的整数倍，他就被称做自然对齐。

   比如在32位cpu下，假设一个整型变量的地址为0x00000004(为4的倍数)，那它就是自然对齐的，而如果其地址为0x00000002（非4的倍数）则是非对齐的。现代计算机中内存空间都是按照byte划分的，从理论上讲似乎对任何类型的变量的访问可以从任何地址开始，但实际情况是在访问特定类型变量的时候经常在特定的内存地址访问，这就需要各种类型数据按照一定的规则在空间上排列，而不是顺序的一个接一个的排放，这就是对齐。

2. **为什么要字节对齐?**

   需要字节对齐的根本原因在于CPU访问数据的效率问题。假设上面整型变量的地址不是自然对齐，比如为0x00000002，则CPU如果取它的值的话需要访问两次内存，第一次取从0x00000002-0x00000003的一个short，第二次取从0x00000004-0x00000005的一个short然后组合得到所要的数据，如果变量在0x00000003地址上的话则要访问三次内存，第一次为char，第二次为short，第三次为char，然后组合得到整型数据。

   而如果变量在自然对齐位置上，则只要一次就可以取出数据。一些系统对对齐要求非常严格，比如sparc系统，如果取未对齐的数据会发生错误，而在x86上就不会出现错误，只是效率下降。

   各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出这32bit，而如果存放在奇地址开始的地方，就需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该32bit数据。显然在读取效率上下降很多。

3. **字节对齐实例**

```c
union example {
    int a[5];
    char b;
    double c;
};
int result = sizeof(example);
/*
如果以最长20字节为准，内部double占8字节，这段内存的地址0x00000020并不是double的整数倍，只有当最小为0x00000024时可以满足整除double（8Byte）同时又可以容纳int a[5]的大小，所以正确的结果应该是result=24
*/

struct example {
    int a[5];
    char b;
    double c;
}test_struct;
int result = sizeof(test_struct);
/*
如果我们不考虑字节对齐，那么内存地址0x0021不是double（8Byte）的整数倍，所以需要字节对齐，那么此时满足是double（8Byte）的整数倍的最小整数是0x0024，说明此时char b对齐int扩充了三个字节。所以最后的结果是result=32
*/

struct example {
    char b;
    double c;
    int a;
}test_struct;
int result = sizeof(test_struct);
/*
字节对齐除了内存起始地址要是数据类型的整数倍以外，还要满足一个条件，那就是占用的内存空间大小需要是结构体中占用最大内存空间的类型的整数倍，所以20不是double（8Byte）的整数倍，我们还要扩充四个字节，最后的结果是result=24
*/
```

1.3 面向对象

1.3.1 简述一下什么是面向对象

参考回答

1. 面向对象是一种编程思想，把一切东西看成是一个个对象，比如人、耳机、鼠标、水杯等，他们各自都有属性，比如：耳机是白色的，鼠标是黑色的，水杯是圆柱形的等等，把这些对象拥有的属性变量和操作这些属性变量的函数打包成一个类来表示
2. 面向过程和面向对象的区别

   面向过程：根据业务逻辑从上到下写代码

   面向对象：将数据与函数绑定到一起，进行封装，这样能够更快速的开发程序，减少了重复代码的重写过程

1.3.2 简述一下面向对象的三大特征

参考回答

面向对象的三大特征是封装、继承、多态。

1. 封装：将数据和操作数据的方法进行有机结合，隐藏对象的属性和实现细节，仅对外公开接口来和对象进行 交互。封装本质上是一种管理：我们如何管理兵马俑呢？比如如果什么都不管，兵马俑就被随意破坏了。那么我们首先建了一座房子把兵马俑给封装起来。但是我们目的全封装起来，不让别人看。所以我们开放了售票通 道，可以买票突破封装在合理的监管机制下进去参观。类也是一样，不想给别人看到的，我们使用protected/private把成员封装起来。开放一些共有的成员函数对成员合理的访问。所以封装本质是一种管理。
2. 继承：可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。

   三种继承方式

继承方式	private继承	protected继承	public继承
基类的private成员	不可见	不可见	不可见
基类的protected成员	变为private成员	仍为protected成员	仍为protected成员
基类的public成员	变为private成员	变为protected成员	仍为public成员仍为public成员

3. 多态：用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。实现多态，有二种方式，重写，重载。

1.3.3 简述一下 C++ 的重载和重写，以及它们的区别

参考回答

1. 重写

   是指派生类中存在重新定义的函数。其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致。只有函数体不同（花括号内），派生类对象调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有virtual修饰。

   示例如下：

```
#include<bits/stdc++.h>

using namespace std;

class A
{
public:
    virtual    void fun()
    {
        cout << "A";
    }
};
class B :public A
{
public:
```

```
    virtual void fun()
    {
        cout << "B";
    }
};
int main(void)
{
    A* a = new B();
    a->fun();//输出B，A类中的fun在B类中重写
}
```

1. 重载

   我们在平时写代码中会用到几个函数但是他们的实现功能相同，但是有些细节却不同。例如：交换两个数的值其中包括（int, float,char,double)这些个类型。在C语言中我们是利用不同的函数名来加以区分。这样的代码不美观而且给程序猿也带来了很多的不便。于是在C++中人们提出了用一个函数名定义多个函数，也就是所谓的函数重载。函数重载是指同一可访问区内被声明的几个具有不同参数列（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。

```
#include<bits/stdc++.h>

using namespace std;

class A
{
    void fun() {};
    void fun(int i) {};
    void fun(int i, int j) {};
    void fun1(int i,int j){};
};
```

1.3.4 说说 C++ 的重载和重写是如何实现的

参考答案

1. C++利用命名倾轧（name mangling）技术，来改名函数名，区分参数不同的同名函数。命名倾轧是在编译阶段完成的。

   C++定义同名重载函数：

```
#include<iostream>
using namespace std;
int func(int a,double b)
{
    return ((a)+(b));
}
int func(double a,float b)
{
    return ((a)+(b));
}
int func(float a,int b)
{
    return ((a)+(b));
}
```

```
15 int main()
16 {
17     return 0;
18 }
```

由上图可得，d代表double，f代表float，i代表int，加上参数首字母以区分同名函数。

2. 在基类的函数前加上virtual关键字，在派生类中重写该函数，运行时将会根据对象的实际类型来调用相应的函数。如果对象类型是派生类，就调用派生类的函数；如果对象类型是基类，就调用基类的函数。

    1. 用virtual关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。
    2. 存在虚函数的类都有一个一维的虚函数表叫做虚表，类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
    3. 多态性是一个接口多种实现，是面向对象的核心，分为类的多态性和函数的多态性。
    4. 重写用虚函数来实现，结合动态绑定。
    5. 纯虚函数是虚函数再加上 = 0。
    6. 抽象类是指包括至少一个纯虚函数的类。

    纯虚函数：virtual void fun()=0。即抽象类必须在子类实现这个函数，即先有名称，没有内容，在派生类实现内容。

1.3.5 说说 C 语言如何实现 C++ 语言中的重载

参考答案

c语言中不允许有同名函数，因为编译时函数命名是一样的，不像c++会添加参数类型和返回类型作为函数编译后的名称，进而实现重载。如果要用c语言显现函数重载，可通过以下方式来实现：

1. 使用函数指针来实现，重载的函数不能使用同名称，只是类似的实现了函数重载功能
2. 重载函数使用可变参数，方式如打开文件open函数
3. gcc有内置函数，程序使用编译函数可以实现函数重载

示例如下：

```
1  #include<stdio.h>
2
3  void func_int(void * a)
4  {
5      printf("%d\n",*(int*)a);  //输出int类型，注意 void * 转化为int
6  }
7
8  void func_double(void * b)
9  {
10     printf("%.2f\n",*(double*)b);
11 }
12
13 typedef void (*ptr)(void *);  //typedef申明一个函数指针
14
15 void c_func(ptr p,void *param)
16 {
17      p(param);                 //调用对应函数
18 }
19
20 int main()
```

```
{
    int a = 23;
    double b = 23.23;
    c_func(func_int,&a);
    c_func(func_double,&b);
    return 0;
}
```

1.3.6 说说构造函数有几种，分别什么作用

参考答案

C++中的构造函数可以分为4类：默认构造函数、初始化构造函数、拷贝构造函数、移动构造函数。

1. 默认构造函数和初始化构造函数。 在定义类的对象的时候，完成对象的初始化工作。

```
class Student
{
public:
    //默认构造函数
    Student()
    {
       num=1001;
       age=18;
    }
    //初始化构造函数
    Student(int n,int a):num(n),age(a){}
private:
    int num;
    int age;
};
int main()
{
    //用默认构造函数初始化对象S1
    Student s1;
    //用初始化构造函数初始化对象S2
    Student s2(1002,18);
    return 0;
}
```

有了有参的构造了，编译器就不提供默认的构造函数。

2. 拷贝构造函数

```
#include "stdafx.h"
#include "iostream.h"

class Test
{
    int i;
    int *p;
public:
    Test(int ai,int value)
    {
        i = ai;
        p = new int(value);
```

```
    }
    ~Test()
    {
        delete p;
    }
    Test(const Test& t)
    {
        this->i = t.i;
        this->p = new int(*t.p);
    }
};
//复制构造函数用于复制本类的对象
int main(int argc, char* argv[])
{
    Test t1(1,2);
    Test t2(t1);//将对象t1复制给t2。注意复制和赋值的概念不同
    return 0;
}
```

赋值构造函数默认实现的是值拷贝（浅拷贝）。

3. 移动构造函数。用于将其他类型的变量，隐式转换为本类对象。下面的转换构造函数，将int类型的r转换为Student类型的对象，对象的age为r，num为1004.

```
Student(int r)
 {
    int num=1004;
    int age= r;

 }
```

1.3.7 只定义析构函数，会自动生成哪些构造函数

参考答案

只定义了析构函数，编译器将自动为我们生成拷贝构造函数和默认构造函数。

默认构造函数和初始化构造函数。 在定义类的对象的时候，完成对象的初始化工作。

```
class Student
{
public:
    //默认构造函数
    Student()
    {
       num=1001;
       age=18;
    }
    //初始化构造函数
    Student(int n,int a):num(n),age(a){}
private:
    int num;
    int age;
};
int main()
{
```

```
    //用默认构造函数初始化对象s1
    Student s1;
    //用初始化构造函数初始化对象s2
    Student s2(1002,18);
    return 0;
}
```

有了有参的构造了，编译器就不提供默认的构造函数。

拷贝构造函数

```
#include "stdafx.h"
#include "iostream.h"

class Test
{
    int i;
    int *p;
public:
    Test(int ai,int value)
    {
        i = ai;
        p = new int(value);
    }
    ~Test()
    {
        delete p;
    }
    Test(const Test& t)
    {
        this->i = t.i;
        this->p = new int(*t.p);
    }
};



//复制构造函数用于复制本类的对象



int main(int argc, char* argv[])
{
    Test t1(1,2);
    Test t2(t1);//将对象t1复制给t2。注意复制和赋值的概念不同。

    return 0;
}
```

赋值构造函数默认实现的是值拷贝（浅拷贝）。

答案解析

示例如下：

```
class HasPtr
{
public:
    HasPtr(const string& s = string()) :ps(new string(s)), i(0) {}
    ~HasPtr() { delete ps; }
private:
    string * ps;
    int i;
};
```

如果类外面有这样一个函数：

```
HasPtr f(HasPtr hp)
{
    HasPtr ret = hp;
    ///... 其他操作
    return ret;

}
```

当函数执行完了之后，将会调用hp和ret的析构函数，将hp和ret的成员ps给delete掉，但是由于ret和hp指向了同一个对象，因此该对象的ps成员被delete了两次，这样产生一个未定义的错误，所以说，如果一个类定义了析构函数，那么它要定义自己的拷贝构造函数和默认构造函数。

1.3.8 说说一个类，默认会生成哪些函数

参考答案

定义一个空类

```
class Empty
{
};
```

默认会生成以下几个函数

1. 无参的构造函数

   在定义类的对象的时候，完成对象的初始化工作。

```
Empty()
{
}
```

1. 拷贝构造函数

   拷贝构造函数用于复制本类的对象

```
Empty(const Empty& copy)
{
}
```

1. 赋值运算符

```
Empty& operator = (const Empty& copy)
{
}
```

1. 析构函数（非虚）

```
~Empty()
{
}
```

1.3.9 说说 C++ 类对象的初始化顺序，有多重继承情况下的顺序

参考答案

1. 创建派生类的对象，基类的构造函数优先被调用（也优先于派生类里的成员类）；
2. 如果类里面有成员类，成员类的构造函数优先被调用；(也优先于该类本身的构造函数)
3. 基类构造函数如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序而不是它们在成员初始化表中的顺序；
4. 成员类对象构造函数如果有多个成员类对象，则构造函数的调用顺序是对象在类中被声明的顺序而不是它们出现在成员初始化表中的顺序；
5. 派生类构造函数，作为一般规则派生类构造函数应该不能直接向一个基类数据成员赋值而是把值传递给适当的基类构造函数,否则两个类的实现变成紧耦合的（tightly coupled）将更加难于正确地修改或扩展基类的实现。（基类设计者的责任是提供一组适当的基类构造函数）
6. 综上可以得出，初始化顺序：

   父类构造函数–>成员类对象构造函数–>自身构造函数

   其中成员变量的初始化与声明顺序有关，构造函数的调用顺序是类派生列表中的顺序。

   析构顺序和构造顺序相反。

1.3.10 简述下向上转型和向下转型

1. 子类转换为父类：向上转型，使用dynamic_cast(expression)，这种转换相对来说比较安全不会有数据的丢失；
2. 父类转换为子类：向下转型，可以使用强制转换，这种转换时不安全的，会导致数据的丢失，原因是父类的指针或者引用的内存中可能不包含子类的成员的内存。

1.3.11 简述下深拷贝和浅拷贝，如何实现深拷贝

1. 浅拷贝：又称值拷贝，将源对象的值拷贝到目标对象中去，本质上来说源对象和目标对象共用一份实体，只是所引用的变量名不同，地址其实还是相同的。举个简单的例子，你的小名叫西西，大名叫冬冬，当别人叫你西西或者冬冬的时候你都会答应，这两个名字虽然不相同，但是都指的是你。
2. 深拷贝，拷贝的时候先开辟出和源对象大小一样的空间，然后将源对象里的内容拷贝到目标对象中去，这样两个指针就指向了不同的内存位置。并且里面的内容是一样的，这样不但达到了我们想要的目的，还不会出现问题，两个指针先后去调用析构函数，分别释放自己所指向的位置。即为每次增加一个指针，便申请一块新的内存，并让这个指针指向新的内存，深拷贝情况下，不会出现重复释放同一块内存的错误。

3. 深拷贝的实现：深拷贝的拷贝构造函数和赋值运算符的重载传统实现：

```
STRING( const STRING& s )
{
    //_str = s._str;
    _str = new char[strlen(s._str) + 1];
    strcpy_s( _str, strlen(s._str) + 1, s._str );
}
STRING& operator=(const STRING& s)
{
    if (this != &s)
    {
        //this->_str = s._str;
        delete[] _str;
        this->_str = new char[strlen(s._str) + 1];
        strcpy_s(this->_str, strlen(s._str) + 1, s._str);
    }
    return *this;
}
```

这里的拷贝构造函数我们很容易理解，先开辟出和源对象一样大的内存区域，然后将需要拷贝的数据复制到目标拷贝对象， 那么这里的赋值运算符的重载是怎么样做的呢?

这种方法解决了我们的指针悬挂问题，通过不断的开空间让不同的指针指向不同的内存，以防止同一块内存被释放两次的问题。

1.3.12 简述一下 C++ 中的多态

由于派生类重写基类方法，然后用基类引用指向派生类对象，调用方法时候会进行动态绑定，这就是多态。 多态分为静态多态和动态多态：

1. 静态多态：编译器在编译期间完成的，编译器会根据实参类型来推断该调用哪个函数，如果有对应的函数，就调用，没有则在编译时报错。

   比如一个简单的加法函数：

```
include<iostream>
using namespace std;

int Add(int a,int b)//1
{
    return a+b;
}

char Add(char a,char b)//2
{
    return a+b;
}

int main()
{
    cout<<Add(666,888)<<endl;//1
    cout<<Add('1','2');//2
    return 0;
```

```
}
```

显然，第一条语句会调用函数1，而第二条语句会调用函数2，这绝不是因为函数的声明顺序，不信你可以将顺序调过来试试。

1. 动态多态：其实要实现动态多态，需要几个条件——即动态绑定条件：
    1. 虚函数。基类中必须有虚函数，在派生类中必须重写虚函数。
    2. 通过基类类型的指针或引用来调用虚函数。

    说到这，得插播一条概念：重写——也就是基类中有一个虚函数，而在派生类中也要重写一个原型（返回值、名字、参数）都相同的虚函数。不过协变例外。协变是重写的特例，基类中返回值是基类类型的引用或指针，在派生类中，返回值为派生类类型的引用或指针。

```cpp
//协变测试函数
#include<iostream>
using namespace std;

class Base
{
public:
    virtual Base* FunTest()
    {
        cout << "victory" << endl;
        return this;
    }
};

class Derived :public Base
{
public:
    virtual Derived* FunTest()
    {
        cout << "yeah" << endl;
        return this;
    }
};

int main()
{
    Base b;
    Derived d;

    b.FunTest();
    d.FunTest();

    return 0;
}
```

1.3.13 说说为什么要虚析构，为什么不能虚构造

1. 虚析构：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。如果基类的析构函数不是虚函数，在特定情况下会导致派生来无法被析构。

1. 用派生类类型指针绑定派生类实例，析构的时候，不管基类析构函数是不是虚函数，都会正常析构
2. 用基类类型指针绑定派生类实例，析构的时候，如果基类析构函数不是虚函数，则只会析构基类，不会析构派生类对象，从而造成内存泄漏。为什么会出现这种现象呢，个人认为析构的时候如果没有虚函数的动态绑定功能，就只根据指针的类型来进行的，而不是根据指针绑定的对象来进行，所以只是调用了基类的析构函数；如果基类的析构函数是虚函数，则析构的时候就要根据指针绑定的对象来调用对应的析构函数了。

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。

2. 不能虚构造：
    1. 从存储空间角度：虚函数对应一个vtale,这个表的地址是存储在对象的内存空间的。如果将构造函数设置为虚函数，就需要到vtable 中调用，可是对象还没有实例化，没有内存空间分配，如何调用。（悖论）
    2. 从使用角度：虚函数主要用于在信息不全的情况下，能使重载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呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
    3. 从实现上看，vbtl 在构造函数调用后才建立，因而构造函数不可能成为虚函数。从实际含义上看，在调用构造函数时还不能确定对象的真实类型（因为子类会调父类的构造函数）；而且构造函数的作用是提供初始化，在对象生命期只执行一次，不是对象的动态行为，也没有太大的必要成为虚函数。

1.3.14 说说模板类是在什么时候实现的

1. 模板实例化：模板的实例化分为显示实例化和隐式实例化，前者是研发人员明确的告诉模板应该使用什么样的类型去生成具体的类或函数，后者是在编译的过程中由编译器来决定使用什么类型来实例化一个模板不管是显示实例化或隐式实例化，最终生成的类或函数完全是按照模板的定义来实现的
2. 模板具体化：当模板使用某种类型类型实例化后生成的类或函数不能满足需要时，可以考虑对模板进行具体化。具体化时可以修改原模板的定义，当使用该类型时，按照具体化后的定义实现，具体化相当于对某种类型进行特殊处理。
3. 代码示例：

```
#include <iostream>
using namespace std;

// #1 模板定义
template<class T>
struct TemplateStruct
{
    TemplateStruct()
    {
        cout << sizeof(T) << endl;
    }
};

// #2 模板显示实例化
template struct TemplateStruct<int>;
```

```cpp

// #3 模板具体化
template<> struct TemplateStruct<double>
{
    TemplateStruct() {
        cout << "--8--" << endl;
    }
};

int main()
{
    TemplateStruct<int> intStruct;
    TemplateStruct<double> doubleStruct;

    // #4 模板隐式实例化
    TemplateStruct<char> llStruct;
}
```

运行结果：

```
4
--8--
1
```

1.3.15 说说类继承时，派生类对不同关键字修饰的基类方法的访问权限

类中的成员可以分为三种类型，分别为public成员、protected成员、public成员。类中可以直接访问自己类的public、protected、private成员，但类对象只能访问自己类的public成员。

1. public继承：派生类可以访问基类的public、protected成员，不可以访问基类的private成员；派生类对象可以访问基类的public成员，不可以访问基类的protected、private成员。
2. protected继承：派生类可以访问基类的public、protected成员，不可以访问基类的private成员；派生类对象不可以访问基类的public、protected、private成员。
3. private继承：派生类可以访问基类的public、protected成员，不可以访问基类的private成员；派生类对象不可以访问基类的public、protected、private成员。

1.3.16 简述一下移动构造函数，什么库用到了这个函数?

C++11中新增了移动构造函数。与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用一个临时变量对对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作。

移动操作的概念对对象管理它们使用的存储空间很有用的，诸如对象使用new和delete分配内存的时候。在这类对象中，拷贝和移动是不同的操作：从A拷贝到B意味着，B分配了新内存，A的整个内容被拷贝到为B分配的新内存上。
而从A移动到B意味着分配给A的内存转移给了B，没有分配新的内存，它仅仅包含简单地拷贝指针。
看下面的例子：

```cpp
// 移动构造函数和赋值
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
public:
    Example6 (const string& str) : ptr(new string(str)) {}
    ~Example6 () {delete ptr;}
    // 移动构造函数，参数x不能是const Pointer&& x，
    // 因为要改变x的成员数据的值；
    // C++98不支持，C++0x（C++11）支持
    Example6 (Example6&& x) : ptr(x.ptr)
    {
        x.ptr = nullptr;
    }
    // move assignment
    Example6& operator= (Example6&& x)
    {
        delete ptr;
        ptr = x.ptr;
        x.ptr=nullptr;
        return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+(const Example6& rhs)
    {
        return Example6(content()+rhs.content());
    }
};
int main () {
    Example6 foo("Exam");            // 构造函数
    // Example6 bar = Example6("ple"); // 拷贝构造函数
    Example6 bar(move(foo));      // 移动构造函数
                                  // 调用move之后，foo变为一个右值引用变量，
                                  // 此时，foo所指向的字符串已经被"掏空"，
                                  // 所以此时不能再调用foo
    bar = bar+ bar;               // 移动赋值，在这儿"="号右边的加法操作，
                                  // 产生一个临时值，即一个右值
                                   // 所以此时调用移动赋值语句
    cout << "foo's content: " << foo.content() << '\n';
    return 0;
}
```

执行结果：

```
foo's content: Example
```

1.3.17 请你回答一下 C++ 类内可以定义引用数据成员吗?

c++类内可以定义引用成员变量，但要遵循以下三个规则：

1. 不能用默认构造函数初始化，必须提供构造函数来初始化引用成员变量。否则会造成引用未初始化错误。
2. 构造函数的形参也必须是引用类型。
3. 不能在构造函数里初始化，必须在初始化列表中进行初始化。

1.3.18 构造函数为什么不能被声明为虚函数?

1. 从存储空间角度：虚函数对应一个vtale,这个表的地址是存储在对象的内存空间的。如果将构造函数设置为虚函数，就需要到vtable 中调用，可是对象还没有实例化，没有内存空间分配，如何调用。（悖论）
2. 从使用角度：虚函数主要用于在信息不全的情况下，能使重载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呀。所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。
3. 从实现上看，vbtl 在构造函数调用后才建立，因而构造函数不可能成为虚函数。从实际含义上看，在调用构造函数时还不能确定对象的真实类型（因为子类会调父类的构造函数）；而且构造函数的作用是提供初始化，在对象生命期只执行一次，不是对象的动态行为，也没有太大的必要成为虚函数。

1.3.19 简述一下什么是常函数，有什么作用

类的成员函数后面加 const，表明这个函数不会对这个类对象的数据成员（准确地说是非静态数据成员）作任何改变。在设计类的时候，一个原则就是对于不改变数据成员的成员函数都要在后面加const，而对于改变数据成员的成员函数不能加 const。所以 const 关键字对成员函数的行为作了更明确的限定：有 const 修饰的成员函数（指 const 放在函数参数表的后面，而不是在函数前面或者参数表内），只能读取数据成员，不能改变数据成员；没有 const 修饰的成员函数，对数据成员则是可读可写的。除此之外，在类的成员函数后面加 const 还有什么好处呢？那就是常量（即 const）对象可以调用 const 成员函数，而不能调用非const修饰的函数。正如非const类型的数据可以给const类型的变量赋值一样，反之则不成立。

```
#include<iostream>
using namespace std;

class CStu
{
public:
    int a;
    CStu()
    {
        a = 12;
    }

    void Show() const
    {
        //a = 13; //常函数不能修改数据成员
        cout <<a << "I am show()" << endl;
```

```
    }
};

int main()
{
    CStu st;
    st.Show();
    system("pause");
    return 0;
}
```

1.3.20 说说什么是虚继承，解决什么问题，如何实现?

虚继承是解决C++多重继承问题的一种手段，从不同途径继承来的同一基类，会在子类中存在多份拷贝。这将存在两个问题：其一，浪费存储空间；第二，存在二义性问题，通常可以将派生类对象的地址赋值给基类对象，实现的具体方式是，将基类指针指向继承类（继承类有基类的拷贝）中的基类对象的地址，但是多重继承可能存在一个基类的多份拷贝，这就出现了二义性。虚继承可以解决多种继承前面提到的两个问题

```
#include<iostream>
using namespace std;
class A{
public:
    int _a;
};
class B :virtual public A
{
public:
    int _b;
};
class C :virtual public A
{
public:
    int _c;
};
class D :public B, public C
{
public:
    int _d;
};
//菱形继承和菱形虚继承的对象模型
int main()
{
    D d;
    d.B::_a = 1;
    d.C::_a = 2;
    d._b = 3;
    d._c = 4;
    d._d = 5;
    cout << sizeof(D) << endl;
    return 0;
}
```

分别从菱形继承和虚继承来分析：

菱形继承中A在B,C,D,中各有一份，虚继承中，A共享。

上面的虚继承表实际上是一个指针数组。B、C实际上是虚基表指针，指向虚基表。

虚基表：存放相对偏移量，用来找虚基类

1.3.21 简述一下虚函数和纯虚函数，以及实现原理

1. C++中的虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。如果调用非虚函数，则无论实际对象是什么类型，都执行基类类型所定义的函数。非虚函数总是在编译时根据调用该函数的对象，引用或指针的类型而确定。如果调用虚函数，则直到运行时才能确定调用哪个函数，运行的虚函数是引用所绑定或指针所指向的对象所属类型定义的版本。虚函数必须是基类的非静态成员函数。虚函数的作用是实现动态联编，也就是在程序的运行阶段动态地选择合适的成员函数，在定义了虚函数后，可以在基类的派生类中对虚函数重新定义，在派生类中重新定义的函数应与虚函数具有相同的形参个数和形参类型。以实现统一的接口，不同定义过程。如果在派生类中没有对虚函数重新定义，则它继承其基类的虚函数。

```
class Person{
    public:
        //虚函数
        virtual void GetName(){
            cout<<"PersonName:xiaosi"<<endl;
        };
};
class Student:public Person{
    public:
        void GetName(){
            cout<<"StudentName:xiaosi"<<endl;
        };
};
int main(){
    //指针
    Person *person = new Student();
    //基类调用子类的函数
    person->GetName();//StudentName:xiaosi
}
```

   虚函数（Virtual Function）是通过一张虚函数表（Virtual Table）来实现的。简称为V-Table。在这个表中，主是要一个类的虚函数的地址表，这张表解决了继承、覆盖的问题，保证其容真实反应实际的函数。这样，在有虚函数的类的实例中这个表被分配在了这个实例的内存中，所以，当我们用父类的指针来操作一个子类的时候，这张虚函数表就显得由为重要了，它就像一个地图一样，指明了实际所应该调用的函数。

2. 纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0” virtualvoid GetName() =0。在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。为了解决上述问题，将函数定义为纯虚函数，则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。将函数定义为纯虚函数能够说明，该函数为后代类型提供了可以覆盖的接口，但是这个类中的函数绝不会调用。声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例。必须在继承类中重新声明函数（不要后面的 = 0）

否则该派生类也不能实例化，而且它们在抽象类中往往没有定义。定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

```
//抽象类
class Person{
    public:
        //纯虚函数
        virtual void GetName()=0;
};
class Student:public Person{
    public:
        Student(){
        };
        void GetName(){
            cout<<"StudentName:xiaosi"<<endl;
        };
};
int main(){
    Student student;
}
```

1.3.22 说说纯虚函数能实例化吗，为什么？派生类要实现吗，为什么？

参考回答

1. 纯虚函数不可以实例化，但是可以用其派生类实例化，示例如下：

```
class Base
{
public:
    virtual void func() = 0;
};
```

```
#include<iostream>

using namespace std;

class Base
{
public:
    virtual void func() = 0;
};

class Derived :public Base
{
public:
    void func() override
    {
        cout << "哈哈" << endl;
    }
};

```

```
int main()
{
    Base *b = new Derived();
    b->func();

    return 0;
}
```

2. 虚函数的原理采用 vtable。类中含有纯虚函数时，其vtable 不完全，有个空位。

   即“纯虚函数在类的vftable表中对应的表项被赋值为0。也就是指向一个不存在的函数。由于编译器绝对不允许有调用一个不存在的函数的可能，所以该类不能生成对象。在它的派生类中，除非重写此函数，否则也不能生成对象。”

   所以纯虚函数不能实例化。

3. 纯虚函数是在基类中声明的虚函数，它要求任何派生类都要定义自己的实现方法，以实现多态性。
4. 定义纯虚函数是为了实现一个接口，用来规范派生类的行为，也即规范继承这个类的程序员必须实现这个函数。派生类仅仅只是继承函数的接口。纯虚函数的意义在于，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但基类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

1.3.23 说说C++中虚函数与纯虚函数的区别

参考回答

1. 虚函数和纯虚函数可以定义在同一个类中，含有纯虚函数的类被称为抽象类，而只含有虚函数的类不能被称为抽象类。
2. 虚函数可以被直接使用，也可以被子类重载以后，以多态的形式调用，而纯虚函数必须在子类中实现该函数才可以使用，因为纯虚函数在基类有声明而没有定义。
3. 虚函数和纯虚函数都可以在子类中被重载，以多态的形式被调用。
4. 虚函数和纯虚函数通常存在于抽象基类之中，被继承的子类重载，目的是提供一个统一的接口。
5. 虚函数的定义形式：`virtual{}`;纯虚函数的定义形式：`virtual { } = 0`;在虚函数和纯虚函数的定义中不能有static标识符，原因很简单，被static修饰的函数在编译时要求前期绑定,然而虚函数却是动态绑定，而且被两者修饰的函数生命周期也不一样。

答案解析

1. 我们举个虚函数的例子：

```
class A
{
public:
    virtual void foo()
    {
        cout<<"A::foo() is called"<<endl;
    }
};
class B:public A
{
public:
    void foo()
    {
        cout<<"B::foo() is called"<<endl;
    }
```

```
};
int main(void)
{
    A *a = new B();
    a->foo();    // 在这里，a虽然是指向A的指针，但是被调用的函数(foo)却是B的!
    return 0;
}
```

这个例子是虚函数的一个典型应用，通过这个例子，也许你就对虚函数有了一些概念。它虚就虚在所谓“推迟联编”或者“动态联编”上，一个类函数的调用并不是在编译时刻被确定的，而是在运行时刻被确定的。由于编写代码的时候并不能确定被调用的是基类的函数还是哪个派生类的函数，所以被成为“虚”函数。
虚函数只能借助于指针或者引用来达到多态的效果。

2. 纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”

   `virtual void funtion1()=0`

   为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。

   在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔雀等子类，但动物本身生成对象明显不合常理。

   为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;），则编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。
   声明了纯虚函数的类是一个抽象类。所以，用户不能创建类的实例，只能创建它的派生类的实例。

   纯虚函数最显著的特征是：它们必须在继承类中重新声明函数（不要后面的 = 0，否则该派生类也不能实例化），而且它们在抽象类中往往没有定义。

   定义纯虚函数的目的在于，使派生类仅仅只是继承函数的接口。

   纯虚函数的意义，让所有的类对象（主要是派生类对象）都可以执行纯虚函数的动作，但类无法为纯虚函数提供一个合理的缺省实现。所以类纯虚函数的声明就是在告诉子类的设计者，“你必须提供一个纯虚函数的实现，但我不知道你会怎样实现它”。

1.3.24 说说 C++ 中什么是菱形继承问题，如何解决

参考回答

1. 下面的图表可以用来解释菱形继承问题。

- 假设我们有类B和类C，它们都继承了相同的类A。另外我们还有类D，类D通过多重继承机制继承了类B和类C。因为上述图表的形状类似于菱形，因此这个问题被形象地称为菱形继承问题。现在，我们将上面的图表翻译成具体的代码：

```
/*
*Animal类对应于图表的类A*
*/

class Animal { /* ... */  }; // 基类
{
int weight;
```

```
8
9   public:
10
11  int getWeight() { return weight;};
12
13  };
14
15  class Tiger : public Animal { /* ... */ };
16
17  class Lion : public Animal { /* ... */  }
18
19  class Liger : public Tiger, public Lion { /* ... */ }
```

在上面的代码中，我们给出了一个具体的菱形继承问题例子。Animal类对应于最顶层类（图表中的A），Tiger和Lion分别对应于图表的B和C，Liger类（狮虎兽，即老虎和狮子的杂交种)对应于D。

现在，问题是如果我们有这种继承结构会出现什么样的问题。

看看下面的代码后再来回答问题吧。

```
1   int main( )
2   {
3   Liger lg ;
4   /*编译错误，下面的代码不会被任何C++编译器通过 */
5   int weight = lg.getWeight();
6   }
```

- 在我们的继承结构中，我们可以看出Tiger和Lion类都继承自Animal基类。所以问题是：因为Liger多重继承了Tiger和Lion类，因此Liger类会有两份Animal类的成员（数据和方法），Liger对象"lg"会包含Animal基类的两个子对象。

  所以，你会问Liger对象有两个Animal基类的子对象会出现什么问题？再看看上面的代码-调用"lg.getWeight()"将会导致一个编译错误。这是因为编译器并不知道是调用Tiger类的getWeight()还是调用Lion类的getWeight()。所以，调用getWeight方法是不明确的，因此不能通过编译。

1. 我们给出了菱形继承问题的解释，但是现在我们要给出一个菱形继承问题的解决方案。如果Lion类和Tiger类在分别继承Animal类时都用virtual来标注，对于每一个Liger对象，C++会保证只有一个Animal类的子对象会被创建。看看下面的代码:

```
1   class Tiger : virtual public Animal { /* ... */ };
2
3   class Lion : virtual public Animal { /* ... */ }
```

- 你可以看出唯一的变化就是我们在类Tiger和类Lion的声明中增加了"virtual"关键字。现在类Liger对象将会只有一个Animal子对象，下面的代码编译正常:

```
1   int main( )
2   {
3   Liger lg ;
4
5   /*既然我们已经在Tiger和Lion类的定义中声明了"virtual"关键字，于是下面的代码编译OK */
6
7   int weight = lg.getWeight();
8   }
```

1.3.25 请问构造函数中的能不能调用虚方法

参考回答

1. 不要在构造函数中调用虚方法，从语法上讲，调用完全没有问题，但是从效果上看，往往不能达到需要的目的。

   派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。

   同样，进入基类析构函数时，对象也是基类类型。

   所以，虚函数始终仅仅调用基类的虚函数（如果是基类调用虚函数），不能达到多态的效果，所以放在构造函数中是没有意义的，而且往往不能达到本来想要的效果。

1.3.26 1.3.26 请问拷贝构造函数的参数是什么传递方式，为什么

参考回答

1. 拷贝构造函数的参数必须使用引用传递
2. 如果拷贝构造函数中的参数不是一个引用，即形如CClass(const CClass c_class)，那么就相当于采用了传值的方式(pass-by-value)，而传值的方式会调用该类的拷贝构造函数，从而造成无穷递归地调用拷贝构造函数。因此拷贝构造函数的参数必须是一个引用。

   需要澄清的是，传指针其实也是传值，如果上面的拷贝构造函数写成CClass(const CClass* c_class)，也是不行的。事实上，只有传引用不是传值外，其他所有的传递方式都是传值。

1.3.27 说说类方法和数据的权限有哪几种

参考回答

1. 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。

关键字	权限
public	可以被任意实体访问
protected	只允许子类及本类的成员函数访问
private	只允许本类的成员函数访问

2. 下面介绍一个例子。

   父类：

```
class Person
{
public:
    Person(const string& name, int age) : m_name(name), m_age(age)
    {
    }

    void ShowInfo()
    {
        cout << "姓名: " << m_name << endl;
        cout << "年龄: " << m_age << endl;
```

```
    }

protected:
    string  m_name;     //姓名

private:
    int     m_age;      //年龄
};
```

子类：

```
class Teacher : public Person
{
public:
    Teacher(const string& name, int age, const string& title)
        : Person(name, age), m_title(title)
    {
    }

    void ShowTeacherInfo()
    {
        ShowInfo();                                 //正确，public属性子类可见
        cout << "姓名：" << m_name << endl;          //正确，protected属性子类可见
        cout << "年龄：" << m_age << endl;      //错误，private属性子类不可见

        cout << "职称：" << m_title << endl;    //正确，本类中可见自己的所有成员
    }

private:
    string  m_title;        //职称
};
```

调用方：

```
void test()
{
    Person person("张三", 22);
    person.ShowInfo();                  //public属性,对外部可见
    cout << person.m_name << endl;      //protected属性,对外部不可见
    cout << person.m_age << endl;       //private属性,对外部不可见
}
```

1.3.28 如何理解抽象类?

参考回答

1. 抽象类的定义如下：

   纯虚函数是在基类中声明的虚函数，它在基类中没有定义，但要求任何派生类都要定义自己的实现方法。在基类中实现纯虚函数的方法是在函数原型后加“=0”，有虚函数的类就叫做抽象类。

2. 抽象类有如下几个特点：

1）抽象类只能用作其他类的基类，不能建立抽象类对象。

2）抽象类不能用作参数类型、函数返回类型或显式转换的类型。

3）可以定义指向抽象类的指针和引用，此指针可以指向它的派生类，进而实现多态性。

1.3.29 什么是多态？除了虚函数，还有什么方式能实现多态？

参考回答

1. 多态是面向对象的重要特性之一，它是一种行为的封装，就是不同对象对同一行为会有不同的状态。(举例 : 学生和成人都去买票时,学生会打折,成人不会)
2. 多态是以封装和继承为基础的。在C++中多态分为静态多态（早绑定）和动态多态（晚绑定）两种，其中动态多态是通过虚函数实现，静态多态通过函数重载实现，代码如下：

```
class A
{
public:
    void do(int a);
    void do(int a, int b);
};
```

1.3.30 简述一下虚析构函数，什么作用

参考回答

1. 虚析构函数，是将基类的析构函数声明为virtual，举例如下：

```
class TimeKeeper
{
public:
    TimeKeeper() {}
    virtual ~TimeKeeper() {}
};
```

2. 虚析构函数的主要作用是防止内存泄露。

   定义一个基类的指针p，在delete p时，如果基类的析构函数是虚函数，这时只会看p所赋值的对象，如果p赋值的对象是派生类的对象，就会调用派生类的析构函数（毫无疑问，在这之前也会先调用基类的构造函数，在调用派生类的构造函数，然后调用派生类的析构函数，基类的析构函数，所谓先构造的后释放）；如果p赋值的对象是基类的对象，就会调用基类的析构函数，这样就不会造成内存泄露。

   如果基类的析构函数不是虚函数，在delete p时，调用析构函数时，只会看指针的数据类型，而不会去看赋值的对象，这样就会造成内存泄露。

答案解析

- 我们创建一个TimeKeeper基类和一些及其它的派生类作为不同的计时方法

```
class TimeKeeper
{
public:
    TimeKeeper() {}
    ~TimeKeeper() {}  //非virtual的
};

//都继承与TimeKeeper
class AtomicClock :public TimeKeeper{};
class WaterClock :public TimeKeeper {};
class WristWatch :public TimeKeeper {};
```

- 如果客户想要在程序中使用时间，不想操作时间如何计算等细节，这时候我们可以设计factory（工厂）函数，让函数返回指针指向一个计时对象。该函数返回一个基类指针，这个基类指针是指向于派生类对象的

```
TimeKeeper* getTimeKeeper()
{
    //返回一个指针，指向一个TimeKeeper派生类的动态分配对象
}
```

- 因为函数返回的对象存在于堆中，因此为了**在不使用时我们需要使用释放该对象（delete）**

```
TimeKeeper* ptk = getTimeKeeper();

delete ptk;
```

- 此处基类的析构函数是非virtual的，因此**通过一个基类指针删除派生类对象是错误的**
- **解决办法：** 将基类的析构函数改为virtual就正确了

```
class TimeKeeper
{
public:
    TimeKeeper() {}
    virtual ~TimeKeeper() {}
};
```

- 声明为virtual之后，通过**基类指针删除派生类对象就会释放整个对象（基类+派生类）**

1.3.31 说说什么是虚基类，可否被实例化?

参考回答

1. 在被继承的类前面加上virtual关键字，这时被继承的类称为虚基类，代码如下：

```
class A
class B1:public virtual A;
class B2:public virtual A;
class D:public B1,public B2;
```

2. 虚继承的类可以被实例化，举例如下：

```
class Animal {/* ... */ };

class Tiger : virtual public Animal { /* ... */ };

class Lion : virtual public Animal { /* ... */ }
```

```
int main( )
{
Liger lg ;

/*既然我们已经在Tiger和Lion类的定义中声明了"virtual"关键字，于是下面的代码编译OK */

int weight = lg.getWeight();
}
```

1.3.32 简述一下拷贝赋值和移动赋值?

参考回答

1. 拷贝赋值是通过拷贝构造函数来赋值，在创建对象时，使用同一类中之前创建的对象来初始化新创建的对象。
2. 移动赋值是通过移动构造函数来赋值，二者的主要区别在于

   1）拷贝构造函数的形参是一个左值引用，而移动构造函数的形参是一个右值引用；

   2）拷贝构造函数完成的是整个对象或变量的拷贝，而移动构造函数是生成一个指针指向源对象或变量的地址，接管源对象的内存，相对于大量数据的拷贝节省时间和内存空间。

1.3.33 仿函数了解吗？有什么作用

参考回答

1. 仿函数（functor）又称为函数对象（function object）是一个能行使函数功能的类。仿函数的语法几乎和我们普通的函数调用一样，不过作为仿函数的类，都必须重载operator()运算符，举个例子：

```
class Func{
     public:
         void operator() (const string& str) const {
             cout<<str<<endl;
         }
};
Func myFunc;
myFunc("helloworld!");
>>>helloworld!
```

1. 仿函数既能想普通函数一样传入给定数量的参数，还能存储或者处理更多我们需要的有用信息。我们可以举个例子：

   假设有一个`vector<string>`，你的任务是统计长度小于5的string的个数，如果使用`count_if`函数的话，你的代码可能长成这样：

```
bool LengthIsLessThanFive(const string& str) {
     return str.length()<5;
}
int res=count_if(vec.begin(), vec.end(), LengthIsLessThanFive);
```

其中`count_if`函数的第三个参数是一个函数指针，返回一个bool类型的值。一般的，如果需要将特定的阈值长度也传入的话，我们可能将函数写成这样：

```
bool LenthIsLessThan(const string& str, int len) {
    return str.length()<len;
}
```

这个函数看起来比前面一个版本更具有一般性，但是他不能满足`count_if`函数的参数要求：`count_if`要求的是unary function（仅带有一个参数）作为它的最后一个参数。如果我们使用仿函数，是不是就豁然开朗了呢：

```
class ShorterThan {
    public:
        explicit ShorterThan(int maxLength) : length(maxLength) {}
        bool operator() (const string& str) const {
            return str.length() < length;
        }
    private:
        const int length;
};
```

1.3.34 C++ 中哪些函数不能被声明为虚函数?

参考回答

常见的不不能声明为虚函数的有：普通函数（非成员函数），静态成员函数，内联成员函数，构造函数，友元函数。

1. 为什么C++不支持普通函数为虚函数?

   普通函数（非成员函数）只能被overload，不能被override，声明为虚函数也没有什么意思，因此编译器会在编译时绑定函数。

2. 为什么C++不支持构造函数为虚函数?

   这个原因很简单，主要是从语义上考虑，所以不支持。因为构造函数本来就是为了明确初始化对象成员才产生的，然而virtual function主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用virtual函数来完成你想完成的动作。（这不就是典型的悖论）

   构造函数用来创建一个新的对象,而虚函数的运行是建立在对象的基础上,在构造函数执行时,对象尚未形成,所以不能将构造函数定义为虚函数

3. 为什么C++不支持内联成员函数为虚函数?

   其实很简单，那内联函数就是为了在代码中直接展开，减少函数调用花费的代价，虚函数是为了在继承后对象能够准确的执行自己的动作，这是不可能统一的。（再说了，*inline*函数在编译时被展开，虚函数在运行时才能动态的绑定函数）

   内联函数是在编译时期展开,而虚函数的特性是运行时才动态联编,所以两者矛盾,不能定义内联函数为虚函数

4. 为什么C++不支持静态成员函数为虚函数?

   这也很简单，静态成员函数对于每个类来说只有一份代码，所有的对象都共享这一份代码，他也没有要动态绑定的必要性。

   静态成员函数属于一个类而非某一对象,没有this指针,它无法进行对象的判别

5. 为什么C++不支持友元函数为虚函数?

   因为C++不支持友元函数的继承，对于没有继承特性的函数没有虚函数的说法。

1.3.35 解释下 C++ 中类模板和模板类的区别

参考回答

1. 类模板是模板的定义，不是一个实实在在的类，定义中用到通用类型参数
2. 模板类是实实在在的类定义，是类模板的实例化。类定义中参数被实际类型所代替。

答案解析

1. 类模板的类型参数可以有一个或多个，每个类型前面都必须加class，如template <class T1,class T2>class someclass{...};在定义对象时分别代入实际的类型名，如 someclass<int,double> obj;
2. 和使用类一样，使用类模板时要注意其作用域，只能在其有效作用域内用它定义对象。
3. 模板可以有层次，一个类模板可以作为基类，派生出派生模板类。

1.3.36 虚函数表里存放的内容是什么时候写进去的?

参考回答

1. 虚函数表是一个存储虚函数地址的数组,以NULL结尾。虚表（vftable）在编译阶段生成，对象内存空间开辟以后，写入对象中的 vfptr，然后调用构造函数。即：虚表在构造函数之前写入
2. 除了在构造函数之前写入之外，我们还需要考虑到虚表的二次写入机制，通过此机制让每个对象的虚表指针都能准确的指向到自己类的虚表，为实现动多态提供支持。

1.4 STL

1.4.1 请说说 STL 的基本组成部分

参考回答

标准模板库（Standard Template Library,简称STL）简单说，就是一些常用数据结构和算法的模板的集合。

广义上讲，STL分为3类：Algorithm（算法）、Container（容器）和Iterator（迭代器），容器和算法通过迭代器可以进行无缝地连接。

详细的说，STL由6部分组成：容器(Container)、算法（Algorithm）、 迭代器（Iterator）、仿函数（Function object）、适配器（Adaptor）、空间配制器（Allocator）。

答案解析

标准模板库STL主要由6大组成部分：

1. 容器(Container)

是一种数据结构， 如list, vector, 和deques，以模板类的方法提供。为了访问容器中的数据，可以使用由容器类输出的迭代器。

2. 算法（Algorithm）

   是用来操作容器中的数据的模板函数。例如，STL用sort()来对一 个vector中的数据进行排序，用find()来搜索一个list中的对象， 函数本身与他们操作的数据的结构和类型无关，因此他们可以用于从简单数组到高度复杂容器的任何数据结构上。

3. 迭代器（Iterator）

   提供了访问容器中对象的方法。例如，可以使用一对迭代器指定list或vector中的一定范围的对象。迭代器就如同一个指针。事实上，C++ 的指针也是一种迭代器。 但是，迭代器也可以是那些定义了operator*()以及其他类似于指针的操作符方法的类对象;

4. 仿函数（Function object）

   仿函数又称之为函数对象， 其实就是重载了操作符的struct,没有什么特别的地方。

5. 适配器（Adaptor）

   简单的说就是一种接口类，专门用来修改现有类的接口，提供一中新的接口；或调用现有的函数来实现所需要的功能。主要包括3中适配器Container Adaptor、Iterator Adaptor、Function Adaptor。

6. 空间配制器（Allocator）

   为STL提供空间配置的系统。其中主要工作包括两部分：

   （1）对象的创建与销毁；

   （2）内存的获取与释放。

1.4.2 请说说 STL 中常见的容器，并介绍一下实现原理

参考回答

容器可以用于存放各种类型的数据（基本类型的变量，对象等）的数据结构，都是模板类，分为顺序容器、关联式容器、容器适配器三种类型，三种类型容器特性分别如下：

1. 顺序容器

   容器并非排序的，元素的插入位置同元素的值无关。包含vector、deque、list，具体实现原理如下：

   （1）vector 头文件

   动态数组。元素在内存连续存放。随机存取任何元素都能在常数时间完成。在尾端增删元素具有较佳的性能。

   （2）deque 头文件

   双向队列。元素在内存连续存放。随机存取任何元素都能在常数时间完成（仅次于vector）。在两端增删元素具有较佳的性能（大部分情况下是常数时间）。

   （3）list 头文件

   双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。不支持随机存取。

2. 关联式容器

   元素是排序的；插入任何元素，都按相应的排序规则来确定其位置；在查找时具有非常好的性能；通常以平衡二叉树的方式实现。包含set、multiset、map、multimap，具体实现原理如下：

   （1）set/multiset 头文件

   set 即集合。set中不允许相同元素，multiset中允许存在相同元素。

（2）map/multimap 头文件

map与set的不同在于map中存放的元素有且仅有两个成员变，一个名为first,另一个名为second,map根据first值对元素从小到大排序，并可快速地根据first来检索元素。

注意：map同multimap的不同在于是否允许相同first值的元素。

3. 容器适配器

封装了一些基本的容器，使之具备了新的函数功能，比如把deque封装一下变为一个具有stack功能的数据结构。这新得到的数据结构就叫适配器。包含stack,queue,priority_queue，具体实现原理如下：

（1）stack 头文件

栈是项的有限序列，并满足序列中被删除、检索和修改的项只能是最进插入序列的项（栈顶的项）。后进先出。

（2）queue 头文件

队列。插入只可以在尾部进行，删除、检索和修改只允许从头部进行。先进先出。

（3）priority_queue 头文件

优先级队列。内部维持某种有序，然后确保优先级最高的元素总是位于头部。最高优先级元素总是第一个出列。

1.4.3 说说 STL 中 map hashtable deque list 的实现原理

参考回答

map、hashtable、deque、list实现机理分别为红黑树、函数映射、双向队列、双向链表，他们的特性分别如下：

1. map实现原理

map内部实现了一个**红黑树**（红黑树是非严格平衡的二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树有自动排序的功能，因此map内部所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找、删除、添加等一系列的操作都相当于是对红黑树进行的操作。map中的元素是按照二叉树（又名二叉查找树、二叉排序树）存储的，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值。使用中序遍历可将键值按照从小到大遍历出来。

2. hashtable（也称散列表，直译作哈希表）实现原理

hashtable采用了**函数映射的思想**记录的存储位置与记录的关键字关联起来，从而能够很快速地进行查找。这决定了哈希表特殊的数据结构，它同数组、链表以及二叉排序树等相比较有很明显的区别，它能够快速定位到想要查找的记录，而不是与表中存在的记录的关键字进行比较来进行查找。

3. deque实现原理

deque内部实现的是一个**双向队列。**元素在内存连续存放。随机存取任何元素都在常数时间完成（仅次于vector）。所有适用于vector的操作都适用于deque。在两端增删元素具有较佳的性能（大部分情况下是常数时间）。

4. list实现原理

list内部实现的是一个**双向链表。**元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。不支持随机存取。无成员函数，给定一个下标i，访问第i个元素的内容，只能从头部挨个遍历到第i个元素。

1.4.4 请你来介绍一下 STL 的空间配置器（allocator）

参考回答

一般情况下,一个程序包括数据结构和相应的算法，而数据结构作为存储数据的组织形式，与内存空间有着密切的联系。在C++ STL中，空间配置器便是用来实现内存空间(一般是内存，也可以是硬盘等空间)分配的工具，他与容器联系紧密，每一种容器的空间分配都是通过空间分配器alloctor实现的。

答案解析

1. 两种C++类对象实例化方式的异同

   在c++中，创建类对象一般分为两种方式：一种是直接利用构造函数,直接构造类对象，如 Test test()；另一种是通过new来实例化一个类对象，如 Test *pTest = new Test；那么，这两种方式有什么异同点呢?

   我们知道，内存分配主要有三种方式：

   （1） 静态存储区分配：内存在程序编译的时候已经分配好，这块内存在程序的整个运行空间内都存在。如全局变量,静态变量等。

   （2） 栈空间分配：程序在运行期间，函数内的局部变量通过栈空间来分配存储（函数调用栈），当函数执行完毕返回时，相对应的栈空间被立即回收。主要是局部变量。
   （3）堆空间分配：程序在运行期间，通过在堆空间上为数据分配存储空间，通过malloc和new创建的对象都是从堆空间分配内存，这类空间需要程序员自己来管理，必须通过free()或者是delete()函数对堆空间进行释放，否则会造成内存溢出。

   那么，从**内存空间分配的角度**来对这两种方式的区别，就比较容易区分:

   （1）对于第一种方式来说，是直接通过调用Test类的构造函数来实例化Test类对象的,如果该实例化对象是一个局部变量，则其是在栈空间分配相应的存储空间。
   （2）对于第二种方式来说,就显得比较复杂。这里主要以new类对象来说明一下。new一个类对象,其实是执行了两步操作：首先,调用new在堆空间分配内存,然后调用类的构造函数构造对象的内容；同样，使用delete释放时，也是经历了两个步骤：首先调用类的析构函数释放类对象，然后调用delete释放堆空间。

2. C++ STL空间配置器实现

   很容易想象，为了实现空间配置器，完全可以利用new和delete函数并对其进行封装实现STL的空间配置器，的确可以这样。但是，为了最大化提升效率，SGI STL版本并没有简单的这样做，而是采取了一定的措施，实现了更加高效复杂的空间分配策略。由于以上的构造都分为两部分，所以，在SGI STL中，将对象的构造切分开来，分成空间配置和对象构造两部分。

   内存配置操作: 通过alloc::allocate()实现
   内存释放操作: 通过alloc::deallocate()实现
   对象构造操作: 通过::construct()实现
   对象释放操作: 通过::destroy()实现

   关于内存空间的配置与释放，SGI STL采用了两级配置器：一级配置器主要是考虑大块内存空间，利用malloc和free实现；二级配置器主要是考虑小块内存空间而设计的（为了最大化解决内存碎片问题，进而提升效率），采用链表free_list来维护内存池（memory pool），free_list通过union结构实现，空闲的内存块互相挂接在一块，内存块一旦被使用，则被从链表中剔除，易于维护。

1.4.5 STL 容器用过哪些，查找的时间复杂度是多少，为什么?

参考回答

STL中常用的容器有vector、deque、list、map、set、multimap、multiset、unordered_map、unordered_set等。容器底层实现方式及时间复杂度分别如下：

1. vector

   采用一维数组实现，元素在内存连续存放，不同操作的时间复杂度为：

   插入: O(N)

   查看: O(1)

   删除: O(N)

2. deque

   采用双向队列实现，元素在内存连续存放，不同操作的时间复杂度为：

   插入: O(N)

   查看: O(1)

   删除: O(N)

3. list

   采用双向链表实现，元素存放在堆中，不同操作的时间复杂度为：

   插入: O(1)

   查看: O(N)

   删除: O(1)

4. map、set、multimap、multiset

   上述四种容器采用红黑树实现，红黑树是平衡二叉树的一种。不同操作的时间复杂度近似为:

   插入: O(logN)

   查看: O(logN)

   删除: O(logN)

5. unordered_map、unordered_set、unordered_multimap、 unordered_multiset

   上述四种容器采用哈希表实现，不同操作的时间复杂度为：
   插入: O(1)，最坏情况O(N)

   查看: O(1)，最坏情况O(N)

   删除: O(1)，最坏情况O(N)

   注意：容器的时间复杂度取决于其底层实现方式。

1.4.6 迭代器用过吗？什么时候会失效?

参考回答

用过，常用容器迭代器失效情形如下。

1. 对于序列容器vector，deque来说，使用erase后，后边的每个元素的迭代器都会失效，后边每个元素都往前移动一位，erase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，不会影响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迭代器，因此上面两种方法都可以使用。

1.4.7 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. 迭代器的作用

   （1）用于指向顺序容器和关联容器中的元素

   （2）通过迭代器可以读取它指向的元素

   （3）通过非const迭代器还可以修改其指向的元素

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，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，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. 迭代器产生的原因

   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。

答案解析

1. 迭代器

   Iterator（迭代器）模式又称游标（Cursor）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. 迭代器示例：

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
    vector<int> v; //一个存放int元素的数组，一开始里面没有元素
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    v.push_back(4);
    vector<int>::const_iterator i; //常量迭代器
    for (i = v.begin(); i != v.end(); ++i) //v.begin()表示v第一个元素迭代器指针，++i指向下一个元素
        cout << *i << ","; //*i表示迭代器指向的元素
    cout << endl;

    vector<int>::reverse_iterator r; //反向迭代器
    for (r = v.rbegin(); r != v.rend(); r++)
        cout << *r << ",";
    cout << endl;
    vector<int>::iterator j; //非常量迭代器
```

```
    for (j = v.begin();j != v.end();j++)
        *j = 100;
    for (i = v.begin();i != v.end();i++)
        cout << *i << ",";
    return 0;
}

/*    运行结果:
          1,2,3,4,
          4,3,2,1,
          100,100,100,100,
*/
```

1.4.8 说说 STL 迭代器是怎么删除元素的

参考回答

这是主要考察迭代器失效的问题。

1. 对于序列容器vector，deque来说，使用erase后，后边的每个元素的迭代器都会失效，后边每个元素都往前移动一位，erase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，不会影响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迭代器，因此上面两种方法都可以使用。

答案解析

容器上迭代器分类如下表（详细实现过程请翻阅相关资料详细了解）：

容器	容器上的迭代器类别
vector	随机访问
deque	随机访问
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

1.4.9 说说 STL 中 resize 和 reserve 的区别

参考回答

1. 首先必须弄清楚两个概念：

（1）capacity：该值在容器初始化时赋值，指的是容器能够容纳的最大的元素的个数。还不能通过下标等访问，因为此时容器中还没有创建任何对象。

（2）size：指的是此时容器中实际的元素个数。可以通过下标访问0-(size-1)范围内的对象。

2. resize和reserve区别主要有以下几点：

（1）resize既分配了空间，也创建了对象；reserve表示容器预留空间，但并不是真正的创建对象，需要通过insert（）或push_back（）等创建对象。

（2）resize既修改capacity大小，也修改size大小；reserve只修改capacity大小，不修改size大小。

（3）两者的形参个数不一样。 resize带两个参数，一个表示容器大小，一个表示初始值（默认为0）；reserve只带一个参数，表示容器预留的大小。

答案解析

问题延伸：

resize 和 reserve 既有差别，也有共同点。两个接口的**共同点**是**它们都保证了vector的空间大小(capacity)最少达到它的参数所指定的大小。**下面就他们的细节进行分析。

为实现resize的语义，resize接口做了两个保证：

（1）保证区间[0, new_size)范围内数据有效，如果下标index在此区间内，vector[indext]是合法的；

（2）保证区间[0, new_size)范围以外数据无效，如果下标index在区间外，vector[indext]是非法的。

reserve只是保证vector的空间大小(capacity)最少达到它的参数所指定的大小n。在区间[0, n)范围内，如果下标是index，vector[index]这种访问有可能是合法的，也有可能是非法的，视具体情况而定。

以下是两个接口的源代码：

```
void resize(size_type new_size)

   {
           resize(new_size, T());
   }
  void resize(size_type new_size, const T& x)
   {
        if (new_size < size())
            erase(begin() + new_size, end()); // erase区间范围以外的数据，确保区间以外的数据无效
          else
            insert(end(), new_size - size(), x); // 填补区间范围内空缺的数据，确保区间内的数据有效
   }


#include<iostream>
#include<vector>
using namespace std;
int main()
{
    vector<int> a;
    cout<<"initial capacity:"<<a.capacity()<<endl;
    cout<<"initial size:"<<a.size()<<endl;

    /*resize改变capacity和size*/
    a.resize(20);
    cout<<"resize capacity:"<<a.capacity()<<endl;
```

```
    cout<<"resize size:"<<a.size()<<endl;


    vector<int> b;
     /*reserve改变capacity,不改变resize*/
    b.reserve(100);
    cout<<"reserve capacity:"<<b.capacity()<<endl;
    cout<<"reserve size:"<<b.size()<<endl;
return 0;
}

/*    运行结果:
          initial capacity:0
        initial size:0
        resize capacity:20
        resize size:20
        reserve capacity:100
        reserve size:0
*/
```

注意：如果n大于当前的vector的容量(是容量，并非vector的size)，将会引起自动内存分配。所以现有的pointer,references,iterators将会失效。而内存的重新配置会很耗时间。

1.4.10 说说 STL 容器动态链接可能产生的问题?

参考回答

1. 可能产生 的问题

   容器是一种动态分配内存空间的一个变量集合类型变量。在一般的程序函数里，局部容器，参数传递容器，参数传递容器的引用，参数传递容器指针都是可以正常运行的，而在动态链接库函数内部使用容器也是没有问题的，但是给动态库函数传递容器的对象本身，则会出现内存堆栈破坏的问题。

2. 产生问题的原因
   容器和动态链接库相互支持不够好，动态链接库函数中使用容器时，参数中只能传递容器的引用，并且要保证容器的大小不能超出初始大小，否则导致容器自动重新分配，就会出现内存堆栈破坏问题。

1.4.11 说说 map 和 unordered_map 的区别? 底层实现

参考回答

map和unordered_map的区别在于他们的**实现基理不同。**

1. map实现机理

   map内部实现了一个**红黑树**（红黑树是非严格平衡的二叉搜索树，而AVL是严格平衡二叉搜索树），红黑树有自动排序的功能，因此map内部所有元素都是有序的，红黑树的每一个节点都代表着map的一个元素。因此，对于map进行的查找、删除、添加等一系列的操作都相当于是对红黑树进行的操作。map中的元素是按照二叉树（又名二叉查找树、二叉排序树）存储的，特点就是左子树上所有节点的键值都小于根节点的键值，右子树所有节点的键值都大于根节点的键值。使用中序遍历可将键值按照从小到大遍历出来。

2. unordered_map实现机理

unordered_map内部实现了一个**哈希表**（也叫散列表），通过把关键码值映射到Hash表中一个位置来访问记录，查找时间复杂度可达O（1），其中在海量数据处理中有着广泛应用。因此，元素的排列顺序是无序的。

1.4.12 说说 vector 和 list 的区别，分别适用于什么场景?

参考回答

vector和list区别在于**底层实现机理不同**，因而特性和适用场景也有所不同。

vector：一维数组

特点：元素在内存连续存放，动态数组，在堆中分配内存，元素连续存放，有保留内存，如果减少大小后内存也不会释放。

优点：和数组类似开辟一段连续的空间，并且支持随机访问，所以它的查找效率高其时间复杂度O(1)。

缺点：由于开辟一段连续的空间，所以插入删除会需要对数据进行移动比较麻烦，时间复杂度O（n），另外当空间不足时还需要进行扩容。

list：双向链表

特点：元素在堆中存放，每个元素都是存放在一块内存中，它的内存空间可以是不连续的，通过指针来进行数据的访问。

优点：底层实现是循环双链表，当对大量数据进行插入删除时，其时间复杂度O(1)。

缺点：底层没有连续的空间，只能通过指针来访问，所以查找数据需要遍历其时间复杂度O（n），没有提供[]操作符的重载。

应用场景

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。

1.4.13 简述 vector 的实现原理

参考回答

vector底层实现原理为**一维数组**（元素在空间连续存放）。

1. 新增元素

   Vector通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素。插入新的数据分在最后插入push_back和通过迭代器在任何位置插入，这里说一下通过迭代器插入，通过迭代器与第一个元素的距离知道要插入的位置，即int index=iter-begin()。这个元素后面的所有元素都向后移动一个位置，在空出来的位置上存入新增的元素。

```
//新增元素
void insert(const_iterator iter,const T& t )
    {
        int index=iter-begin();
        if (index<size_)
       {
           if (size_==capacity_)
```

```
            {
                 int capa=calculateCapacity();
               newCapacity(capa);
            }
            memmove(buf+index+1,buf+index,(size_-index)*sizeof(T));
            buf[index]=t;
            size_++;
        }
    }
```

2. 删除元素

删除和新增差不多，也分两种，删除最后一个元素pop_back和通过迭代器删除任意一个元素erase(iter)。通过迭代器删除还是先找到要删除元素的位置，即int index=iter-begin();这个位置后面的每个元素都想前移动一个元素的位置。同时我们知道erase不释放内存只初始化成默认值。

删除全部元素clear：只是循环调用了erase，所以删除全部元素的时候，不释放内存。内存是在析构函数中释放的。

```
//删除元素
iterator erase(const_iterator iter)
        {
            int index=iter-begin();
            if (index<size_ && size_>0)
            {
                memmove(buf+index ,buf+index+1,(size_-index)*sizeof(T));
                buf[--size_]=T();
            }
            return iterator(iter);
        }
```

3. 迭代器iteraotr

迭代器iteraotr是STL的一个重要组成部分,通过iterator可以很方便的存储集合中的元素.STL为每个集合都写了一个迭代器, 迭代器其实是对一个指针的包装,实现一些常用的方法,如++,--,!=,==,*,->等,通过这些方法可以找到当前元素或是别的元素. vector是STL集合中比较特殊的一个,因为vector中的每个元素都是连续的,所以在自己实现vector的时候可以用指针代替。

```
//迭代器的实现
template<class _Category,
    class _Ty,
    class _Diff = ptrdiff_t,
    class _Pointer = _Ty *,
    class _Reference = _Ty&>
    struct iterator
    {    // base type for all iterator classes
           typedef _Category iterator_category;
           typedef _Ty value_type;
           typedef _Diff difference_type;
           typedef _Diff distance_type;    // retained
           typedef _Pointer pointer;
           typedef _Reference reference;
    };
```

4. vector实现源码

```
#ifndef _CVECTOR_H_
```

```
#define _CVECTOR_H_

namespace cth
{
    class NoCopy
    {
    public:
        inline NoCopy(){}
        NoCopy(const NoCopy&);
        NoCopy& operator=(const NoCopy&);
    };

    template<typename T>
    class viterator:public std::iterator<std::forward_iterator_tag,T>
    {
    public:
        viterator()
        {
            t=NULL;
        }
        viterator(T* t_)
        {
            t=t_;
        }
        viterator(const viterator& other)
        {
            t=other.t;
        }
        viterator& operator=(const viterator& other)
        {
            t=other.t;
            return *this;
        }

        viterator& operator++()
        {
            t++;
            return *this;
        }
        viterator operator++(int)
        {
            viterator iter=*this;
            t++;
            return iter;
        }
        viterator operator+(int count)
        {
            viterator iter=*this;
            iter.t+=count;
            return iter;
        }
        viterator& operator--()
        {
            t--;
            return *this;
        }
        viterator operator--(int)
        {
```

```
            viterator iter=*this;
            t--;
            return iter;
        }
        viterator operator-(int count)
        {
            viterator iter=*this;
            iter.t-=count;
            return iter;
        }

        int operator-(const viterator& other)
        {
            return t-other.t;
        }
        int operator-(const viterator& other)const
        {
            return t-other.t;
        }

        T& operator*()
        {
            return *t;
        }
        const T& operator*() const
        {
            return *t;
        }

        T* operator->()
        {
            return t;
        }
        const T* operator->() const
        {
            return t;
        }

        inline bool operator!=(const viterator& other)
        {
            return t!=other.t;
        }
        inline bool operator!=(const viterator& other)const
        {
            return t!=other.t;
        }

        inline bool operator==(const viterator& other)
        {
            return t==other.t;
        }
        inline bool operator==(const viterator& other)const
        {
            return t==other.t;
        }

        inline bool operator<(const viterator& other)
        {
```

```
            return t<other.t;
        }
        inline bool operator<(const viterator& other)const
        {
            return t<other.t;
        }

        inline bool operator<=(const viterator& other)
        {
            return t<=other.t;
        }
        inline bool operator<=(const viterator& other)const
        {
            return t<=other.t;
        }

        inline bool operator>(const viterator& other)
        {
            return t>other.t;
        }
        inline bool operator>(const viterator& other)const
        {
            return t>other.t;
        }

        inline bool operator>=(const viterator& other)
        {
            return t>=other.t;
        }
        inline bool operator>=(const viterator& other)const
        {
            return t>=other.t;
        }
    private:
        T* t;
    };
```

```
template<typename T>
class cvector:public NoCopy
{
public:
    typedef viterator<T> iterator;//viterator<T>就是对一个指针的包装，所以完全可以用T*代替viterator <T>
    typedef const viterator<T> const_iterator;

    //typedef T* iterator;
    //typedef const T* const_iterator;
    cvector()
    {
        initData(0);
    }
    cvector(int capa,const T& val=T())
    {
        initData(capa);
        newCapacity(capacity_);
        for (int i=0;i<size_;i++)
            buf[i]=val;
```

```
        }
        cvector(const_iterator first,const_iterator last)
        {
            initData(last-first);
            newCapacity(capacity_);
            iterator iter=iterator(first);
            int index=0;
            while(iter!=last)
                buf[index++]=*iter++;
        }

        ~cvector()
        {
            if (buf)
            {
                delete[] buf;
                buf=NULL;
            }
            size_=capacity_=0;
        }
        void clear()
        {
            if (buf)
                erase(begin(),end());
        }

        void push_back(const T& t)
        {
            if (size_==capacity_)
            {
                int capa=calculateCapacity();
                newCapacity(capa);
            }
            buf[size_++]=t;
        }
        void pop_back()
        {
            if (!empty())
                erase(end() - 1);
        }

        int insert(const_iterator iter,const T& t )
        {
            int index=iter-begin();
            if (index<=size_)
            {
                if (size_==capacity_)
                {
                    int capa=calculateCapacity();
                    newCapacity(capa);
                }
                memmove(buf+index+1,buf+index,(size_-index)*sizeof(T));
                buf[index]=t;
                size_++;
            }
            return index;
        }
        iterator erase(const_iterator iter)
```

```
        {
            int index=iter-begin();
            if (index<size_ && size_>0)
            {
                memmove(buf+index ,buf+index+1,(size_-index)*sizeof(T));
                buf[--size_]=T();
            }
            return iterator(iter);
        }

        iterator erase(const_iterator first,const_iterator last)
        {
            iterator first_=iterator(first);
            iterator last_=iterator(last);
            while(first_<=last_--)
                erase(first_);
            return iterator(first_);
        }

        T& front()
        {
            assert(size_>0);
            return buf[0];
        }
        T& back()
        {
            assert(size_>0);
            return buf[size_-1];
        }
        T& at(int index)
        {
            assert(size_>0);
            return buf[index];
        }
        T& operator[](int index)
        {
            assert(size_>0 && index>=0 && index<size_);
            return buf[index];
        }

        bool empty() const
        {
            return size_==0;
        }
        int size() const
        {
            return size_;
        }
        int capacity() const
        {
            return capacity_;
        }

        iterator begin()
        {
            return iterator(&buf[0]);
        }
        iterator end()
```

```
136         {
137             return iterator(&buf[size_]);
138         }
139 
140     private:
141         void newCapacity(int capa)
142         {
143             capacity_=capa;
144             T* newBuf=new T[capacity_];
145             if (buf)
146             {
147                 memcpy(newBuf,buf,size_*sizeof(T));
148                 delete [] buf;
149             }
150             buf=newBuf;
151         }
152 
153         inline int calculateCapacity()
154         {
155             return capacity_*3/2+1 ;
156         }
157 
158         inline void initData(int capa)
159         {
160             buf=NULL;
161             size_=capacity_=capa>0?capa:0;
162         }
163         int size_;
164         int capacity_ ;
165         T* buf;
166     };
167 
168 
169 
170     struct Point
171     {
172         Point(int x_=0,int y_=0):x(x_),y(y_){}
173         int x,y;
174     };
175 
176     bool operator<(const Point& p1,const Point& p2)
177     {
178         if(p1.x<p2.x)
179         {
180             return true;
181         }else if(p1.x>p2.x)
182         {
183             return false;
184         }
185         return p1.y<p2.y;
186     }
187 
188     void cvectorTest()
189     {
190         cvector<Point> vect;
191         for (int i=0;i<10;i++)
192         {
193             Point p(i,i);
```

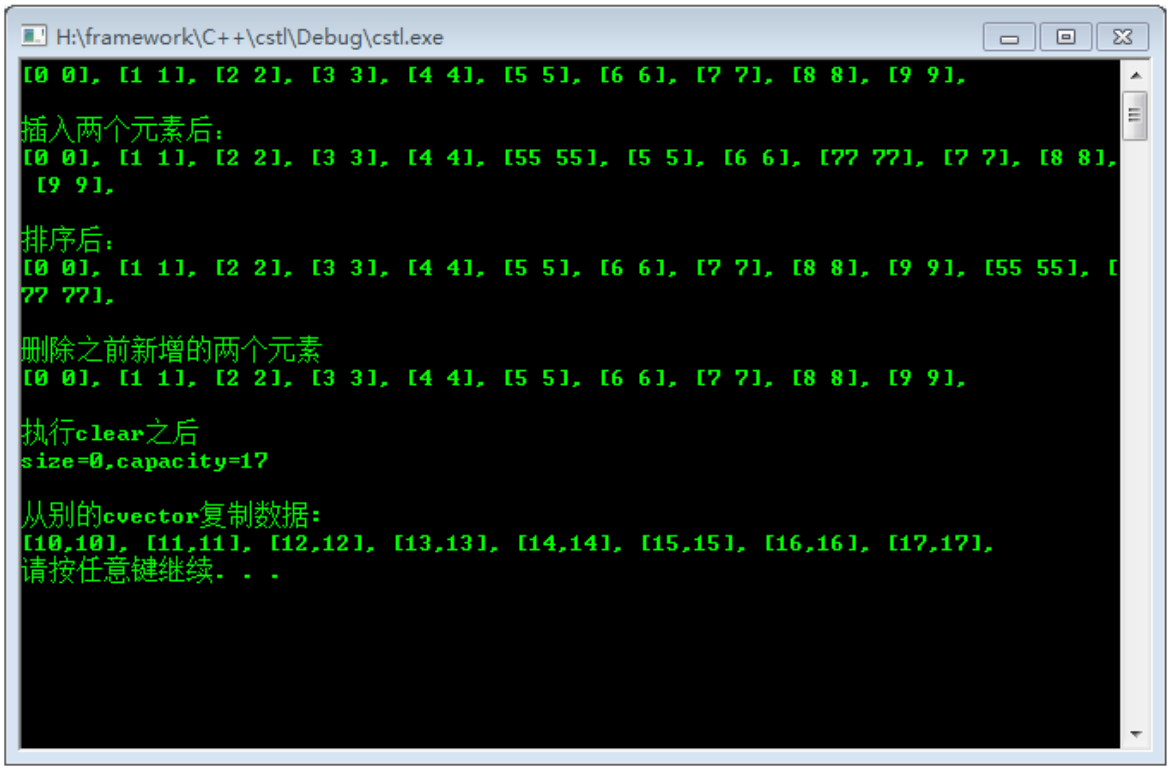
```
            vect.push_back(p);
        }

        cvector<Point>::iterator iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        iter=vect.begin()+5;
        vect.insert(iter,Point(55,55));
        iter=vect.end()-3;
        vect.insert(iter,Point(77,77));
        cout<<endl<<endl<<"插入两个元素后："<<endl;
        iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        std::sort(vect.begin(),vect.end());
        cout<<endl<<endl<<"排序后："<<endl;
        iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        vect.erase(vect.begin()+10);
        vect.erase(vect.begin()+10);
        cout<<endl<<endl<<"删除之前新增的两个元素"<<endl;
        iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        vect.clear();
        cout<<endl<<endl<<"执行clear之后"<<endl;
        cout<<"size="<<vect.size()<<",capacity="<<vect.capacity();

        cvector<Point> vect1;
        for (int i=10;i<20;i++)
        {
            Point p(i,i);
            vect1.push_back(p);
        }
        cout<<endl<<endl<<"从别的cvector复制数据:"<<endl;

        cvector<Point> vect2(vect1.begin(),vect1.end());
        vect2.pop_back();
        vect2.pop_back();
        for(int i=0;i<vect2.size();i++)
        {
            cout<<"["<<vect2[i].x<<","<<vect2[i].y<<"], ";
        }

        cout<<endl;
```

```
    }
}
```

```c++
//实例代码级运行结果
struct Point
    {
        Point(int x_=0,int y_=0):x(x_),y(y_){}
        int x,y;
    };

    bool operator<(const Point& p1,const Point& p2)
    {
        if(p1.x<p2.x)
        {
            return true;
        }else if(p1.x>p2.x)
        {
            return false;
        }
        return p1.y<p2.y;
    }

    void cvectorTest()
    {
        cvector<Point> vect;
        for (int i=0;i<10;i++)
        {
            Point p(i,i);
            vect.push_back(p);
        }

        cvector<Point>::iterator iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        iter=vect.begin()+5;
        vect.insert(iter,Point(55,55));
        iter=vect.end()-3;
        vect.insert(iter,Point(77,77));
        cout<<endl<<endl<<"插入两个元素后："<<endl;
        iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
        }
        std::sort(vect.begin(),vect.end());
        cout<<endl<<endl<<"排序后："<<endl;
        iter=vect.begin();
        while (iter!=vect.end())
        {
            cout<< "[" << iter->x << " " << iter->y <<"], ";
            ++iter;
```

```
            }
            vect.erase(vect.begin()+10);
            vect.erase(vect.begin()+10);
            cout<<endl<<endl<<"删除之前新增的两个元素"<<endl;
            iter=vect.begin();
            while (iter!=vect.end())
            {
                cout<< "[" << iter->x << " " << iter->y <<"], ";
                ++iter;
            }
            vect.clear();
            cout<<endl<<endl<<"执行clear之后"<<endl;
            cout<<"size="<<vect.size()<<",capacity="<<vect.capacity();

            cvector<Point> vect1;
            for (int i=10;i<20;i++)
            {
                Point p(i,i);
                vect1.push_back(p);
            }
            cout<<endl<<endl<<"从别的cvector复制数据:"<<endl;

            cvector<Point> vect2(vect1.begin(),vect1.end());
            vect2.pop_back();
            vect2.pop_back();
            for(int i=0;i<vect2.size();i++)
            {
                cout<<"["<<vect2[i].x<<","<<vect2[i].y<<"], ";
            }

            cout<<endl;
        }
```

代码运行结果如下：



1.4.14 1.4.14 简述 STL 中的 map 的实现原理

参考回答

map是关联式容器，它们的底层容器都是**红黑树。**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。

1. map的特性如下

   （1）map以RBTree作为底层容器；

   （2）所有元素都是键+值存在；

   （3）不允许键重复；

   （4）所有元素是通过键进行自动排序的；

   （5）map的键是不能修改的，但是其键对应的值是可以修改的。

1.4.15 1.4.15 C++ 的 vector 和 list中，如果删除末尾的元素，其指针和迭代器如何变化？若删除的是中间的元素呢?

参考回答

1. 迭代器和指针之间的区别

   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，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，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。

2. vector和list特性

   vector特性 动态数组。元素在内存连续存放。随机存取任何元素都在常数时间完成。在尾端增删元素具有较大的性能（大部分情况下是常数时间）。

   list特性 双向链表。元素在内存不连续存放。在任何位置增删元素都能在常数时间完成。不支持随机存取。

3. vector增删元素

   对于vector而言，删除某个元素以后，该元素后边的每个元素的迭代器都会失效，后边每个元素都往前移动一位，erase返回下一个有效的迭代器。

4. list增删元素

   对于list而言，删除某个元素，只有"指向被删除元素"的那个迭代器失效，其它迭代器不受任何影响。

1.4.16 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

1. set是一种关联式容器，其特性如下：

   （1）set以RBTree作为底层容器

   （2）所得元素的只有key没有value，value就是key

（3）不允许出现键值重复

（4）所有的元素都会被自动排序

（5）不能通过迭代器来改变set的值，因为set的值就是键，set的迭代器是const的

2. map和set一样是关联式容器，其特性如下：

（1）map以RBTree作为底层容器

（2）所有元素都是键+值存在

（3）不允许键重复

（4）所有元素是通过键进行自动排序的

（5）map的键是不能修改的，但是其键对应的值是可以修改的

综上所述，map和set**底层实现**都是红黑树；map和set的**区别**在于map的值不作为键，键和值是分开的。

1.4.17 hashtable 扩容和如何解决冲突

参考回答

1. 哈希表的扩容

（1）为什么要扩容

使用链地址法封装哈希表时, 填装因子(loaderFactor)会大于1，理论上这种封装的哈希表时可以无限插入数据的但是但是随着数据量的增多，哈希表中的每个元素会变得越来越长， 这是效率会大大降低。 因此，需要通过扩容来提高效率。

（2）如何扩容

Hashtable每次扩容，容量都为原来的2倍加1，而HashMap为原来的2倍。此时，需要将所有数据项都进行修改(需要重新调用哈希函数，来获取新的位置)。 哈希表扩容是一个比较耗时的过程，但是一劳永逸。

（3）什么情况下扩容

常见的情况是在填装因子(loaderFactor) > 0.75是进行扩容。

（4）扩容的代码实现（详见解析答案解析代码）

2. 如何解决哈希冲突

解决哈希冲突通常有开放地址法和链地址法两种方法，分别如下：

1.4.17.1 开放定址法

即当一个关键字和另一个关键字发生冲突时，使用某种探测技术在Hash表中形成一个探测序列，然后沿着这个探测序列依次查找下去，当碰到一个空的单元时，则插入其中。比较常用的探测方法有线性探测法，比如有一组关键字 `{12，13，25，23，38，34，6，84，91}`，Hash表长为14，Hash函数为 `address(key)=key%11` ，当插入12，13，25时可以直接插入，而当插入23时，地址1被占用了，因此沿着地址1依次往下探测(探测步长可以根据情况而定)，直到探测到地址4，发现为空，则将23插入其中。

1.4.17.2 链地址法

采用数组和链表相结合的办法，将Hash地址相同的记录存储在一张线性表中，而每张表的表头的序号即为计算得到的Hash地址。

答案解析

1. 哈希表的使用

   Hash表采用一个映射函数`f :key -> address`将关键字映射到该记录在表中的存储位置，从而在想要查找该记录时，可以直接根据关键字和映射关系计算出该记录在表中的存储位置，通常情况下，这种映射关系称作为Hash函数，而通过Hash函数和关键字计算出来的存储位置(注意这里的存储位置只是表中的存储位置，并不是实际的物理地址)称作为Hash地址。

2. 哈希函数的设计

   Hash函数设计的好坏直接影响到对Hash表的操作效率。通常有以下几种构造Hash函数的方法：

1.4.17.3 直接定址法

取关键字或者关键字的某个线性函数作为Hash地址，即`address(key) = a*key + b`。

1.4.17.4 平方取中法

对关键字进行平方计算，然后取结果的中间几位作为Hash地址，假如有以下关键字序列`{421，423，436}`，平方之后的结果为`{177241，178929，190096}`，那么可以取中间的两位数`{72，89，OO}`作为Hash地址。

1.4.17.5 折叠法

将关键字拆分成几个部分，然后将这几个部分组合在一起，以特定的方式进行转化形成Hash地址。例如假如知道某图书的SBN号为：`8903-241-23`，可以将`address(key)=89+03+24+12+3`作为Hash地址。

1.4.17.6 除留取余法

如果知道Hash表的最大长度为m，可以取不大于m的最大质数p，然后对关键字进行取余运算，`address(key)=key % p`。

在这里p的选取非常关键，p选择的好的话，能够最大程度地减少冲突，p一般取不大于m的最大质数。

3. 哈希表大小的确定

   Hash表大小的确定非常关键，如果Hash表的空间远远大于最后实际存储的记录个数，就会造成较大的空间浪费。如果选取小了的话，则容易造成冲突。在实际情况中，一般需要根据最终记录存储个数和关键字的分布特点来确定Hash表的大小。还有一种情况时可能事先不知道最终需要存储的记录个数，则需要动态维护Hash表的容量，此时可能需要重新计算Hash地址。

4. 冲突的解决

   如果产生了Hash冲突，就需要办法来解决，通常有如下两种方法：

1.4.17.7 开放定址法

即当一个关键字和另一个关键字发生冲突时，使用某种探测技术在Hash表中形成一个探测序列，然后沿着这个探测序列依次查找下去，当碰到一个空的单元时，则插入其中。比较常用的探测方法有线性探测法，比如有一组关键字`{12，13，25，23，38，34，6，84，91}`，Hash表长为14，Hash函数为`address(key)=key%11`，当插入12，13，25时可以直接插入，而当插入23时，地址1被占用了，因此沿着地址1依次往下探测(探测步长可以根据情况而定)，直到探测到地址4，发现为空，则将23插入其中。

1.4.17.8 链地址法

采用数组和链表相结合的办法，将Hash地址相同的记录存储在一张线性表中，而每张表的表头的序号即为计算得到的Hash地址。

5. 需注意的点

（1）Hashtable的默认容量为11，默认负载因子为0.75(HashMap默认容量为16，默认负载因子也是0.75)；

（2）Hashtable的容量可以为任意整数，最小值为1，而HashMap的容量始终为2的n次方；

（3）为避免扩容带来的性能问题，建议指定合理容量；

（4）跟HashMap一样，Hashtable内部也有一个静态类叫Entry，其实是个键值对对象，保存了键和值的引用；

（5）HashMap和Hashtable存储的是键值对对象，而不是单独的键或值。

6. 哈希表的构造函数

```
public Hashtable(int initialCapacity, float loadFactor) {//可指定初始容量和加载因子
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal Load: "+loadFactor);
        if (initialCapacity==0)
            initialCapacity = 1;//初始容量最小值为1
        this.loadFactor = loadFactor;
        table = new Entry[initialCapacity];//创建桶数组
        threshold = (int)Math.min(initialCapacity * loadFactor, MAX_ARRAY_SIZE + 1);//初始化容量阈值
        useAltHashing = sun.misc.VM.isBooted() &&
                (initialCapacity >= Holder.ALTERNATIVE_HASHING_THRESHOLD);
    }
    /**
     * Constructs a new, empty hashtable with the specified initial capacity
     * and default load factor (0.75).
     */
    public Hashtable(int initialCapacity) {
        this(initialCapacity, 0.75f);//默认负载因子为0.75
    }
    public Hashtable() {
        this(11, 0.75f);//默认容量为11，负载因子为0.75
```

```
    }
    /**
     * Constructs a new hashtable with the same mappings as the given
     * Map.  The hashtable is created with an initial capacity
sufficient to
     * hold the mappings in the given Map and a default load factor
(0.75).
     */
    public Hashtable(Map<? extends K, ? extends V> t) {
        this(Math.max(2*t.size(), 11), 0.75f);
        putAll(t);
    }
```

1.4.18 说说 push_back 和 emplace_back 的区别

参考回答

如果要将一个临时变量push到容器的末尾，push_back()需要先构造临时对象，再将这个对象拷贝到容器的末尾，而emplace_back()则直接在容器的末尾构造对象，这样就省去了拷贝的过程。

答案解析

参考代码：

```
#include <iostream>
#include <cstring>
#include <vector>
using namespace std;

class A {
public:
    A(int i){
        str = to_string(i);
        cout << "构造函数" << endl;
    }
    ~A(){}
    A(const A& other): str(other.str){
        cout << "拷贝构造" << endl;
    }

public:
    string str;
};

int main()
{
    vector<A> vec;
    vec.reserve(10);
    for(int i=0;i<10;i++){
        vec.push_back(A(i)); //调用了10次构造函数和10次拷贝构造函数，
//        vec.emplace_back(i);  //调用了10次构造函数一次拷贝构造函数都没有调用过
    }
```

1.4.19 STL 中 vector 与 list 具体是怎么实现的？常见操作的时间复杂度是多少？

参考回答

1. vector 一维数组（元素在内存连续存放）

   是动态数组，在堆中分配内存，元素连续存放，有保留内存，如果减少大小后，内存也不会释放；如果新增大小当前大小时才会重新分配内存。

   扩容方式： a. 倍放开辟三倍的内存

   b. 旧的数据开辟到新的内存

   c. 释放旧的内存

   d. 指向新内存

2. list 双向链表（元素存放在堆中）

   元素存放在堆中，每个元素都是放在一块内存中，它的内存空间可以是不连续的，通过指针来进行数据的访问，这个特点，使得它的随机存取变得非常没有效率，因此它没有提供[ ]操作符的重载。但是由于链表的特点，它可以很有效的支持任意地方的删除和插入操作。

   特点：a. 随机访问不方便

   b. 删除插入操作方便

3. 常见时间复杂度

   （1）vector插入、查找、删除时间复杂度分别为：O(n)、O(1)、O(n)；

   （2）list插入、查找、删除时间复杂度分别为：O(1)、O(n)、O(1)。

1.5 新特性

1.5.1 说说 C++11 的新特性有哪些

参考回答

C++新特性主要包括包含语法改进和标准库扩充两个方面，主要包括以下11点：

1. 语法的改进

   （1）统一的初始化方法

   （2）成员变量默认初始化

   （3）auto关键字 用于定义变量，编译器可以自动判断的类型（前提：定义一个变量时对其进行初始化）

   （4）decltype 求表达式的类型

   （5）智能指针 shared_ptr

   （6）空指针 nullptr（原来NULL）

   （7）基于范围的for循环

   （8）右值引用和move语义 让程序员有意识减少进行深拷贝操作

2. 标准库扩充（往STL里新加进一些模板类，比较好用）

   （9）无序容器（哈希表） 用法和功能同map一模一样，区别在于哈希表的效率更高

   （10）正则表达式 可以认为正则表达式实质上是一个字符串，该字符串描述了一种特定模式的字符串

（11）Lambda表达式

答案解析

1. 统一的初始化方法

   C++98/03 可以使用初始化列表（initializer list）进行初始化：

```
int i_arr[3] = { 1, 2, 3 };
long l_arr[] = { 1, 3, 2, 4 };
struct A
{
    int x;
    int y;
} a = { 1, 2 };
```

   但是这种初始化方式的**适用性非常狭窄**，只有上面提到的这两种数据类型可以使用初始化列表。在 C++11 中，初始化列表的适用性被大大增加了。它现在可以用于任何类型对象的初始化，实例如下：

```
class Foo
{
public:
    Foo(int) {}
private:
    Foo(const Foo &);
};
int main(void)
{
    Foo a1(123);
    Foo a2 = 123;  //error: 'Foo::Foo(const Foo &)' is private
    Foo a3 = { 123 };
    Foo a4 { 123 };
    int a5 = { 3 };
    int a6 { 3 };
    return 0;
}
```

   在上例中，a3、a4 使用了新的初始化方式来初始化对象，效果如同 a1 的直接初始化。a5、a6 则是基本数据类型的列表初始化方式。可以看到，它们的形式都是统一的。这里需要注意的是，a3 虽然使用了等于号，但它仍然是列表初始化，因此，私有的拷贝构造并不会影响到它。a4 和 a6 的写法，是 C++98/03 所不具备的。在 C++11 中，可以直接在变量名后面跟上初始化列表，来进行对象的初始化。

2. 成员变量默认初始化

   好处：构建一个类的对象不需要用构造函数初始化成员变量。

```
//程序实例
#include<iostream>
using namespace std;
class B
{
public:
    int m = 1234; //成员变量有一个初始值
    int n;
};
```

```
int main()
{
    B b;
    cout << b.m << endl;
    return 0;
}
```

3. auto关键字

用于定义变量，编译器可以自动判断的类型（前提：定义一个变量时对其进行初始化）。

```
//程序实例
#include <vector>
using namespace std;
int main(){
    vector< vector<int> > v;
    vector< vector<int> >::iterator i = v.begin();
    return 0;
}
```

可以看出来，定义迭代器 i 的时候，类型书写比较冗长，容易出错。然而有了 auto 类型推导，我们大可不必这样，只写一个 auto 即可。

4. decltype 求表达式的类型

decltype 是 C++11 新增的一个关键字，它和 auto 的功能一样，都用来在编译时期进行自动类型推导。

(1)为什么要有decltype

因为 auto 并不适用于所有的自动类型推导场景，在某些特殊情况下 auto 用起来非常不方便，甚至压根无法使用，所以 decltype 关键字也被引入到 C++11 中。

auto 和 decltype 关键字都可以自动推导出变量的类型，但它们的用法是有区别的：

```
auto varname = value;
decltype(exp) varname = value;
```

其中，varname 表示变量名，value 表示赋给变量的值，exp 表示一个表达式。

auto 根据"="右边的初始值 value 推导出变量的类型，而 decltype 根据 exp 表达式推导出变量的类型，跟"="右边的 value 没有关系。

另外，auto 要求变量必须初始化，而 decltype 不要求。这很容易理解，auto 是根据变量的初始值来推导出变量类型的，如果不初始化，变量的类型也就无法推导了。decltype 可以写成下面的形式：

```
decltype(exp) varname;
```

(2)代码示例

```
// decltype 用法举例
nt a = 0;
decltype(a) b = 1;  //b 被推导成了 int
decltype(10.8) x = 5.5;  //x 被推导成了 double
decltype(x + 100) y;  //y 被推导成了 double
```

5. 智能指针 shared_ptr

和 unique_ptr、weak_ptr 不同之处在于，多个 shared_ptr 智能指针可以共同使用同一块堆内存。并且，由于该类型智能指针在实现上采用的是引用计数机制，即便有一个 shared_ptr 指针放弃了堆内存的“使用权”（引用计数减 1），也不会影响其他指向同一堆内存的 shared_ptr 指针（只有引用计数为 0 时，堆内存才会被自动释放）。

```
#include <iostream>
#include <memory>
using namespace std;
int main()
{
    //构建 2 个智能指针
    std::shared_ptr<int> p1(new int(10));
    std::shared_ptr<int> p2(p1);
    //输出 p2 指向的数据
    cout << *p2 << endl;
    p1.reset();//引用计数减 1,p1为空指针
    if (p1) {
        cout << "p1 不为空" << endl;
    }
    else {
        cout << "p1 为空" << endl;
    }
    //以上操作，并不会影响 p2
    cout << *p2 << endl;
    //判断当前和 p2 同指向的智能指针有多少个
    cout << p2.use_count() << endl;
    return 0;
}

/*    程序运行结果：
            10
            p1 为空
            10
            1
*/
```

6. 空指针 nullptr（原来NULL）

nullptr 是 nullptr_t 类型的右值常量，专用于初始化空类型指针。nullptr_t 是 C++11 新增加的数据类型，可称为“指针空值类型”。也就是说，nullpter 仅是该类型的一个实例对象（已经定义好，可以直接使用），如果需要我们完全定义出多个同 nullptr 完全一样的实例对象。值得一提的是，nullptr 可以被隐式转换成任意的指针类型。例如：

```
int * a1 = nullptr;
char * a2 = nullptr;
double * a3 = nullptr;
```

显然，不同类型的指针变量都可以使用 nullptr 来初始化，编译器分别将 nullptr 隐式转换成 int、*char* 以及 double* 指针类型。另外，通过将指针初始化为 nullptr，可以很好地解决 NULL 遗留的问题，比如：

```
#include <iostream>
using namespace std;
void isnull(void *c){
    cout << "void*c" << endl;
```

```
}
void isnull(int n){
    cout << "int n" << endl;
}
int main() {
    isnull(NULL);
    isnull(nullptr);
    return 0;
}

/*    程序运行结果：
        int n
        void*c
*/
```

7. 基于范围的for循环

如果要用 for 循环语句遍历一个数组或者容器，只能套用如下结构：

```
for(表达式 1; 表达式 2; 表达式 3){
    //循环体
}
```

```
//程序实例
#include <iostream>
#include <vector>
#include <string.h>
using namespace std;
int main() {
    char arc[] = "www.123.com";
    int i;
    //for循环遍历普通数组
    for (i = 0; i < strlen(arc); i++) {
        cout << arc[i];
    }
    cout << endl;
    vector<char>myvector(arc,arc+3);
    vector<char>::iterator iter;
    //for循环遍历 vector 容器
    for (iter = myvector.begin(); iter != myvector.end(); ++iter) {
        cout << *iter;
    }
    return 0;
}
/*    程序运行结果：
        www.123.com
        www
*/
```

8. 右值引用和move语义

1. 右值引用

C++98/03 标准中就有引用，使用 "&" 表示。但此种引用方式有一个缺陷，即正常情况下只能操作 C++ 中的左值，无法对右值添加引用。举个例子：

```
int num = 10;
int &b = num; //正确
int &c = 10; //错误
```

如上所示，编译器允许我们为 num 左值建立一个引用，但不可以为 10 这个右值建立引用。因此，C++98/03 标准中的引用又称为左值引用。

注意，虽然 C++98/03 标准不支持为右值建立非常量左值引用，但允许使用常量左值引用操作右值。也就是说，常量左值引用既可以操作左值，也可以操作右值，例如：

```
int num = 10;
const int &b = num;
const int &c = 10;
```

我们知道，右值往往是没有名称的，因此要使用它只能借助引用的方式。这就产生一个问题，实际开发中我们可能需要对右值进行修改（实现移动语义时就需要），显然左值引用的方式是行不通的。

为此，C++11 标准新引入了另一种引用方式，称为右值引用，用 "&&" 表示。

需要注意的，和声明左值引用一样，右值引用也必须立即进行初始化操作，且只能使用右值进行初始化，比如：

```
int num = 10;
//int && a = num;  //右值引用不能初始化为左值
int && a = 10;
```

和常量左值引用不同的是，右值引用还可以对右值进行修改。例如：

```
int && a = 10;
a = 100;
cout << a << endl;
/*    程序运行结果：
        100
*/
```

另外值得一提的是，C++ 语法上是支持定义常量右值引用的，例如：

```
const int&& a = 10;//编译器不会报错
```

但这种定义出来的右值引用并无实际用处。一方面，右值引用主要用于移动语义和完美转发，其中前者需要有修改右值的权限；其次，常量右值引用的作用就是引用一个不可修改的右值，这项工作完全可以交给常量左值引用完成。

2. move语义

move 本意为 "移动"，但该函数并不能移动任何数据，它的功能很简单，就是将某个左值强制转化为右值。基于 move() 函数特殊的功能，其常用于实现移动语义。move() 函数的用法也很简单，其语法格式如下：

```
move( arg ) //其中，arg 表示指定的左值对象。该函数会返回 arg 对象的右值形式。
```

```
//程序实例
#include <iostream>
using namespace std;
```

```cpp
class first {
public:
    first() :num(new int(0)) {
        cout << "construct!" << endl;
    }
    //移动构造函数
    first(first &&d) :num(d.num) {
        d.num = NULL;
        cout << "first move construct!" << endl;
    }
public:    //这里应该是 private，使用 public 是为了更方便说明问题
    int *num;
};
class second {
public:
    second() :fir() {}
    //用 first 类的移动构造函数初始化 fir
    second(second && sec) :fir(move(sec.fir)) {
        cout << "second move construct" << endl;
    }
public:    //这里也应该是 private，使用 public 是为了更方便说明问题
    first fir;
};
int main() {
    second oth;
    second oth2 = move(oth);
    //cout << *oth.fir.num << endl;    //程序报运行时错误
    return 0;
}

/*    程序运行结果：
          construct!
        first move construct!
        second move construct
*/
```

9. 无序容器（哈希表）

用法和功能同map一模一样，区别在于哈希表的效率更高。

(1) 无序容器具有以下 2 个特点：

a. 无序容器内部存储的键值对是无序的，各键值对的存储位置取决于该键值对中的键，

b. 和关联式容器相比，无序容器擅长通过指定键查找对应的值（平均时间复杂度为 O(1)）；但对于使用迭代器遍历容器中存储的元素，无序容器的执行效率则不如关联式容器。

(2) 和关联式容器一样，无序容器只是一类容器的统称，其包含有 4 个具体容器，分别为 unordered_map、unordered_multimap、unordered_set 以及 unordered_multiset。功能如下表：

无序容器	功能
unordered_map	存储键值对 <key, value> 类型的元素，其中各个键值对键的值不允许重复，且该容器中存储的键值对是无序的。
unordered_multimap	和 unordered_map 唯一的区别在于，该容器允许存储多个键相同的键值对。
unordered_set	不再以键值对的形式存储数据，而是直接存储数据元素本身（当然也可以理解为，该容器存储的全部都是键 key 和值 value 相等的键值对，正因为它们相等，因此只存储 value 即可）。另外，该容器存储的元素不能重复，且容器内部存储的元素也是无序的。
unordered_multiset	和 unordered_set 唯一的区别在于，该容器允许存储值相同的元素。

(3) 程序实例（以 unordered_map 容器为例）

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main()
{
    //创建并初始化一个 unordered_map 容器，其存储的 <string,string> 类型的键值对
    std::unordered_map<std::string, std::string> my_uMap{
        {"教程1","www.123.com"},
        {"教程2","www.234.com"},
        {"教程3","www.345.com"} };
    //查找指定键对应的值，效率比关联式容器高
    string str = my_uMap.at("C语言教程");
    cout << "str = " << str << endl;
    //使用迭代器遍历哈希容器，效率不如关联式容器
    for (auto iter = my_uMap.begin(); iter != my_uMap.end(); ++iter)
    {
        //pair 类型键值对分为 2 部分
        cout << iter->first << " " << iter->second << endl;
    }
    return 0;
}

/*    程序运行结果：
          教程1 www.123.com
          教程2 www.234.com
          教程3 www.345.com
*/
```

10. 正则表达式

可以认为正则表达式实质上是一个字符串，该字符串描述了一种特定模式的字符串。常用符号的意义如下：

符号	意义
^	匹配行的开头
$	匹配行的结尾
.	匹配任意单个字符
[...]	匹配[]中的任意一个字符
(...)	设定分组
\	转义字符
\d	匹配数字[0-9]
\D	\d 取反
\w	匹配字母[a-z]，数字，下划线
\W	\w 取反
\s	匹配空格
\S	\s 取反
+	前面的元素重复1次或多次
*	前面的元素重复任意次
?	前面的元素重复0次或1次
{n}	前面的元素重复n次
{n,}	前面的元素重复至少n次
{n,m}	前面的元素重复至少n次，至多m次
|	逻辑或

11. Lambda匿名函数

所谓匿名函数，简单地理解就是没有名称的函数，又常被称为 lambda 函数或者 lambda 表达式。

（1）定义

lambda 匿名函数很简单，可以套用如下的语法格式：

[外部变量访问方式说明符] (参数) mutable noexcept/throw() -> 返回值类型
{
函数体;
};

其中各部分的含义分别为：

a. [外部变量方位方式说明符]

```
1 [ ] 方括号用于向编译器表明当前是一个 lambda 表达式，其不能被省略。在方括号内部，可以注明当前 lambda 函数的函数体中可以使用哪些“外部变量”。
```

所谓外部变量，指的是和当前 lambda 表达式位于同一作用域内的所有局部变量。

b. (参数)

1 和普通函数的定义一样，lambda 匿名函数也可以接收外部传递的多个参数。和普通函数不同的是，如果不需要传递参数，可以连同 () 小括号一起省略；

c. mutable

1 此关键字可以省略，如果使用则之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，对于以值传递方式引入的外部变量，不允许在 lambda 表达式内部修改它们的值（可以理解为这部分变量都是 const 常量）。而如果想修改它们，就必须使用 mutable 关键字。

注意:对于以值传递方式引入的外部变量，lambda 表达式修改的是拷贝的那一份，并不会修改真正的外部变量；

d. noexcept/throw()

1 可以省略，如果使用，在之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，lambda 函数的函数体中可以抛出任何类型的异常。而标注 noexcept 关键字，则表示函数体内不会抛出任何异常；使用 throw() 可以指定 lambda 函数内部可以抛出的异常类型。

e. -> 返回值类型

1 指明 lambda 匿名函数的返回值类型。值得一提的是，如果 lambda 函数体内只有一个 return 语句，或者该函数返回 void，则编译器可以自行推断出返回值类型，此情况下可以直接省略"-> 返回值类型"。

f. 函数体

1 和普通函数一样，lambda 匿名函数包含的内部代码都放置在函数体中。该函数体内除了可以使用指定传递进来的参数之外，还可以使用指定的外部变量以及全局范围内的所有全局变量。

（2）程序实例

```
#include <iostream>
#include <algorithm>
using namespace std;
int main()
{
    int num[4] = {4, 2, 3, 1};
    //对 a 数组中的元素进行排序
    sort(num, num+4, [=](int x, int y) -> bool{ return x < y; } );
    for(int n : num){
        cout << n << " ";
    }
    return 0;
}

/*    程序运行结果：
          1 2 3 4
*/
```

1.5.2 说说 C++ 中智能指针和指针的区别是什么?

参考回答

1. 智能指针

   如果在程序中使用new从堆（自由存储区）分配内存，等到不需要时，应使用delete将其释放。C++引用了智能指针auto_ptr，以帮助自动完成这个过程。随后的编程体验（尤其是使用STL）表明，需要有更精致的机制。基于程序员的编程体验和BOOST库提供的解决方案，C++11摒弃了auto_ptr，并新增了三种智能指针：unique_ptr、shared_ptr和weak_ptr。所有新增的智能指针都能与STL容器和移动语义协同工作。

2. 指针

   C 语言规定所有变量在使用前必须先定义，指定其类型，并按此分配内存单元。指针变量不同于整型变量和其他类型的变量，它是专门用来存放地址的，所以必须将它定义为“指针类型”。

3. 智能指针和普通指针的区别

智能指针和普通指针的区别在于智能指针实际上是对普通指针加了一层封装机制，区别是它负责自动释放所指的对象，这样的一层封装机制的目的是为了使得智能指针可以方便的管理一个对象的生命期。

答案解析

无

1.5.3 说说 C++中的智能指针有哪些？分别解决的问题以及区别?

参考回答

1. C++中的智能指针有4种，分别为：**shared_ptr、unique_ptr、weak_ptr、auto_ptr**，其中auto_ptr被C++11弃用。

2. 使用智能指针的原因

   申请的空间（即new出来的空间），在使用结束时，需要delete掉，否则会形成内存碎片。在程序运行期间，new出来的对象，在析构函数中delete掉，但是这种方法不能解决所有问题，因为有时候new发生在某个全局函数里面，该方法会给程序员造成精神负担。**此时，智能指针就派上了用场。**使用智能指针可以很大程度上避免这个问题，因为智能指针就是一个类，当超出了类的作用域时，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，避免了手动释放内存空间。

3. 四种指针分别解决的问题以及各自特性如下：

   （1）auto_ptr（C++98的方案，C++11已经弃用）

   采用所有权模式。

   ```
   auto_ptr<string> p1(new string("I reigned loney 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时的情形特别有用。

采用所有权模式，和上面例子一样。

```
auto_ptr<string> p3(new string("I reigned loney as a cloud."));
auto_ptr<string> p4;
p4=p3; //此时不会报错
```

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_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 。

注意：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（非常好使）

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 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：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();

答案解析

无

1.5.4 简述 C++ 右值引用与转移语义

参考回答

1. 右值引用

   一般来说，不能取地址的表达式，就是右值引用，能取地址的，就是左值。

```
class A { };
A & r = A(); //error,A()是无名变量，是右值
A && r = A(); //ok,r是右值引用
```

1. 转移语义

   move 本意为 "移动"，但该函数并不能移动任何数据，它的功能很简单，就是将某个左值强制转化为右值。基于 move() 函数特殊的功能，其常用于实现移动语义。

答案解析

1. 右值引用

C++98/03 标准中就有引用，使用 "&" 表示。但此种引用方式有一个缺陷，即正常情况下只能操作 C++ 中的左值，无法对右值添加引用。举个例子：

```
int num = 10;
int &b = num; //正确
int &c = 10; //错误
```

如上所示，编译器允许我们为 num 左值建立一个引用，但不可以为 10 这个右值建立引用。因此，C++98/03 标准中的引用又称为左值引用。

注意：虽然 C++98/03 标准不支持为右值建立非常量左值引用，但允许使用常量左值引用操作右值。也就是说，常量左值引用既可以操作左值，也可以操作右值，例如：

```
int num = 10;
const int &b = num;
const int &c = 10;
```

我们知道，右值往往是没有名称的，因此要使用它只能借助引用的方式。这就产生一个问题，实际开发中我们可能需要对右值进行修改（实现移动语义时就需要），显然左值引用的方式是行不通的。

为此，C++11 标准新引入了另一种引用方式，称为右值引用，用 "&&" 表示。

注意：和声明左值引用一样，右值引用也必须立即进行初始化操作，且只能使用右值进行初始化，比如：

```
int num = 10;
//int && a = num;  //右值引用不能初始化为左值
int && a = 10;
```

和常量左值引用不同的是，右值引用还可以对右值进行修改。例如：

```
int && a = 10;
a = 100;
cout << a << endl;
/*    程序运行结果：
        100
*/
```

另外值得一提的是，C++ 语法上是支持定义常量右值引用的，例如：

```
const int&& a = 10;//编译器不会报错
```

但这种定义出来的右值引用并无实际用处。一方面，右值引用主要用于移动语义和完美转发，其中前者需要有修改右值的权限；其次，常量右值引用的作用就是引用一个不可修改的右值，这项工作完全可以交给常量左值引用完成。

1. move语义

```
//程序实例
#include <iostream>
using namespace std;
class first {
public:
    first() :num(new int(0)) {
        cout << "construct!" << endl;
    }
    //移动构造函数
    first(first &&d) :num(d.num) {
        d.num = NULL;
        cout << "first move construct!" << endl;
    }
public:    //这里应该是 private，使用 public 是为了更方便说明问题
    int *num;
};
class second {
public:
    second() :fir() {}
    //用 first 类的移动构造函数初始化 fir
    second(second && sec) :fir(move(sec.fir)) {
        cout << "second move construct" << endl;
    }
public:    //这里也应该是 private，使用 public 是为了更方便说明问题
    first fir;
};
int main() {
    second oth;
    second oth2 = move(oth);
    //cout << *oth.fir.num << endl;    //程序报运行时错误
    return 0;
}

/*    程序运行结果：
          construct!
        first move construct!
        second move construct
*/
```

1.5.5 简述 C++ 中智能指针的特点

参考回答

1. C++中的智能指针有4种，分别为：**shared_ptr、unique_ptr、weak_ptr、auto_ptr**，其中auto_ptr被C++11弃用。
2. **为什么要使用智能指针**：智能指针的作用是管理一个指针，因为存在申请的空间在函数结束时忘记释放，造成内存泄漏的情况。使用智能指针可以很大程度上避免这个问题，因为智能指针就是一个类，当超出了类的作用域时，类会自动调用析构函数，自动释放资源。
3. 四种指针各自特性

   （1）auto_ptr

   auto指针存在的问题是，两个智能指针同时指向一块内存，就会两次释放同一块资源，自然报错。

   （2）unique_ptr

   unique指针规定一个智能指针独占一块内存资源。当两个智能指针同时指向一块内存，编译报错。

   实现原理：将拷贝构造函数和赋值拷贝构造函数申明为private或delete。不允许拷贝构造函数和赋值操作符，但是支持移动构造函数，通过std:move把一个对象指针变成右值之后可以移动给另一个unique_ptr

   （3）shared_ptr

   共享指针可以实现多个智能指针指向相同对象，该对象和其相关资源会在引用为0时被销毁释放。

   实现原理：有一个引用计数的指针类型变量，专门用于引用计数，使用拷贝构造函数和赋值拷贝构造函数时，引用计数加1，当引用计数为0时，释放资源。

注意：weak_ptr、shared_ptr存在一个问题，当两个shared_ptr指针相互引用时，那么这两个指针的引用计数不会下降为0，资源得不到释放。因此引入weak_ptr，weak_ptr是弱引用，weak_ptr的构造和析构不会引起引用计数的增加或减少。

答案解析

无

1.5.6 weak_ptr 能不能知道对象计数为 0，为什么?

参考回答

不能。

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。进行该对象管理的是那个引用的shared_ptr。weak_ptr只是提供了对管理 对象的一个访问手段。weak_ptr设计的目的只是为了配合shared_ptr而引入的一种智能指针，配合shared_ptr工作，它只可以从一个shared_ptr或者另一个weak_ptr对象构造，**它的构造和析构不会引起计数的增加或减少。**

答案解析
无

1.5.7 weak_ptr 如何解决 shared_ptr 的循环引用问题?

参考回答

为了解决循环引用导致的内存泄漏，引入了弱指针weak_ptr，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，但是不会指向引用计数的共享内存，但是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。

答案解析

参见1.2.10

1.5.8 share_ptr 怎么知道跟它共享对象的指针释放了

参考回答

多个shared_ptr对象可以同时托管一个指针，系统会维护一个托管计数。当无shared_ptr托管该指针时，delete该指针。

答案解析
无

1.5.9 说说智能指针及其实现，shared_ptr 线程安全性，原理

参考回答

1. 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。
2. 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。
3. 四种指针分别解决的问题以及各自特性如下：

   （1）auto_ptr（C++98的方案，C++11已经弃用）

   ```
   //程序实例
   auto_ptr<string> p1(new string("I reigned loney as a cloud."));
   auto_ptr<string> p2;
   p2=p1; //auto_ptr不会报错
   ```

   采用所有权模式。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时的情形特别有用。**

```
//程序实例
auto_ptr<string> p3(new string("I reigned loney as a cloud."));
auto_ptr<string> p4;
p4=p3; //此时不会报错
```

采用所有权模式，和上面例子一样。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，比如以下代码：

```
//程序实例
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 。
注意：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：

```
//程序实例
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

（3）shared_ptr（非常好使）

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。其成员函数如下：

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 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。

```
//程序实例
    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：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();

1. **线程安全性**

多线程环境下，调用不同shared_ptr实例的成员函数是不需要额外的同步手段的，即使这些shared_ptr拥有的是同样的对象。但是如果多线程访问（有写操作）同一个shared_ptr，则需要同步，否则就会有race condition 发生。也可以使用 shared_ptr overloads of atomic functions来防止race condition的发生。

多个线程同时读同一个shared_ptr对象是线程安全的，但是如果是多个线程对同一个shared_ptr对象进行读和写，则需要加锁。

多线程读写shared_ptr所指向的同一个对象，不管是相同的shared_ptr对象，还是不同的shared_ptr对象，也需要加锁保护。例子如下：

```
//程序实例
shared_ptr<long> global_instance = make_shared<long>(0);
std::mutex g_i_mutex;

void thread_fcn()
{
    //std::lock_guard<std::mutex> lock(g_i_mutex);

    //shared_ptr<long> local = global_instance;

    for(int i = 0; i < 100000000; i++)
    {
        *global_instance = *global_instance + 1;
        //*local = *local + 1;
    }
}

int main(int argc, char** argv)
{
    thread thread1(thread_fcn);
    thread thread2(thread_fcn);

    thread1.join();
    thread2.join();

    cout << "*global_instance is " << *global_instance << endl;

    return 0;
}
```

在线程函数thread_fcn的for循环中，2个线程同时对global_instance进行加1的操作。这就是典型的非线程安全的场景，最后的结果是未定的，运行结果如下：

*global_instance is 197240539

如果使用的是每个线程的局部shared_ptr对象local，因为这些local指向相同的对象，因此结果也是未定的，运行结果如下： *global_instance is 160285803

因此，这种情况下必须加锁，将thread_fcn中的第一行代码的注释去掉之后，不管是使用global_instance，还是使用local，得到的结果都是：

*global_instance is 200000000

答案解析

无

1.5.10 请你回答一下智能指针有没有内存泄露的情况

参考回答

智能指针有内存泄露的情况发生。

1. 智能指针发生内存泄露的情况

   当两个对象同时使用一个shared_ptr成员变量指向对方，会造成循环引用，使引用计数失效，从而导致内存泄露。

2. 智能指针的内存泄漏如何解决?

> 为了解决循环引用导致的内存泄漏，引入了弱指针weak_ptr，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，但是不会指向引用计数的共享内存，但是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。

答案解析

```
//程序实例
#include <memory>
#include <iostream>
using namespace std;

class Child;
class Parent{
private:
    std::shared_ptr<Child> ChildPtr;
public:
    void setChild(std::shared_ptr<Child> child) {
        this->ChildPtr = child;
    }

    void doSomething() {
        if (this->ChildPtr.use_count()) {

        }
    }

    ~Parent() {

    }
};

class Child{
private:
    std::shared_ptr<Parent> ParentPtr;
public:
    void setPartent(std::shared_ptr<Parent> parent) {
        this->ParentPtr = parent;
    }
    void doSomething() {
        if (this->ParentPtr.use_count()) {

        }
    }
    ~Child() {
```

```
    }
};

int main() {
    std::weak_ptr<Parent> wpp;
    std::weak_ptr<Child> wpc;

    {
        std::shared_ptr<Parent>    p(new Parent);
        std::shared_ptr<Child> c(new Child);
        p->setChild(c);
        c->setPartent(p);
        wpp = p;
        wpc = c;
        std::cout << p.use_count() << std::endl;
        std::cout << c.use_count() << std::endl;
    }
    std::cout << wpp.use_count() << std::endl;
    std::cout << wpc.use_count() << std::endl;
    return 0;
}
/*    程序运行结果:
        2
        2
        1
        1
*/
```

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。

1.5.11 简述一下 C++11 中四种类型转换

参考回答

C++中四种类型转换分别为**const_cast、static_cast、dynamic_cast、reinterpret_cast**，四种转换功能分别如下：

1. const_cast

将const变量转为非const

1. static_cast

   最常用，可以用于各种隐式转换，比如非const转const，static_cast可以用于类向上转换，但向下转换能成功但是不安全。

2. dynamic_cast

   只能用于含有虚函数的类转换，用于类向上和向下转换

   向上转换：指子类向基类转换。

   向下转换：指基类向子类转换。

   这两种转换，子类包含父类，当父类转换成子类时可能出现非法内存访问的问题。

dynamic_cast通过判断变量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。dynamic_cast可以做类之间上下转换，转换的时候会进行类型检查，类型相等成功转换，类型不等转换失败。运用RTTI技术，RTTI是"Runtime Type Information"的缩写，意思是运行时类型信息，它提供了运行时确定对象类型的方法。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，dynamic_cast和typeid都会去查询type_info。

3. reinterpret_cast

reinterpret_cast可以做任何类型的转换，不过不对转换结果保证，容易出问题。

注意：为什么不用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，但是转换不够明确，不能进行错误检查，容易出错。

答案解析

无

1.5.12 简述一下 C++ 11 中 auto 的具体用法

参考回答

auto用于定义变量，编译器可以自动判断变量的类型。auto主要有以下几种用法：

1. auto的基本使用方法

（1）基本使用语法如下

```
auto name = value; //name 是变量的名字，value 是变量的初始值
```

注意：auto 仅仅是一个占位符，在编译器期间它会被真正的类型所替代。或者说，C++ 中的变量必须是有明确类型的，只是这个类型是由编译器自己推导出来的。

（2）程序实例如下

```
auto n = 10;
auto f = 12.8;
auto p = &n;
auto url = "www.123.com";
```

a. 第 1 行中，10 是一个整数，默认是 int 类型，所以推导出变量 n 的类型是 int。

b. 第 2 行中，12.8 是一个小数，默认是 double 类型，所以推导出变量 f 的类型是 double。

c. 第 3 行中，&n 的结果是一个 int* 类型的指针，所以推导出变量 f 的类型是 int*。

d. 第 4 行中，由双引号`""`包围起来的字符串是 const char* 类型，所以推导出变量 url 的类型是 const char*，也即一个常量指针。

1. auto和 const 的结合使用

（1） auto 与 const 结合的用法

a. 当类型不为引用时，auto 的推导结果将不保留表达式的 const 属性；

b. 当类型为引用时，auto 的推导结果将保留表达式的 const 属性。

（2）程序实例如下

```
int  x = 0;
const  auto n = x;  //n 为 const int ，auto 被推导为 int
auto f = n;       //f 为 const int，auto 被推导为 int（const 属性被抛弃）
const auto &r1 = x;  //r1 为 const int& 类型，auto 被推导为 int
auto &r2 = r1;  //r1 为 const int& 类型，auto 被推导为 const int 类型
```

a. 第 2 行代码中，n 为 const int，auto 被推导为 int。

b. 第 3 行代码中，n 为 const int 类型，但是 auto 却被推导为 int 类型，这说明当`=`右边的表达式带有 const 属性时，auto 不会 使用 const 属性，而是直接推导出 non-const 类型。

c. 第 4 行代码中，auto 被推导为 int 类型，这个很容易理解，不再赘述。

d. 第 5 行代码中，r1 是 const int & 类型，auto 也被推导为 const int 类型，这说明当 const 和引用结合时，auto 的推导将保留 表达式的 const 类型。

1. 使用auto定义迭代器

   在使用 stl 容器的时候，需要使用迭代器来遍历容器里面的元素；不同容器的迭代器有不同的类型，在定义迭代器时必须指明。而迭代器的类型有时候比较复杂，请看下面的例子：

```
#include <vector>
using namespace std;
int main(){
    vector< vector<int> > v;
    //vector< vector<int> >::iterator i = v.begin();
    auto i = v.begin();  //使用 auto 代替具体的类型,该句比上一句简洁,根据表达式 v.begin() 的类型（begin() 函数的返回值类型）来推导出变量i的类型
    return 0;
}
```

1. 用于泛型编程

   auto 的另一个应用就是当我们不知道变量是什么类型，或者不希望指明具体类型的时候，比如泛型编程中。请看下面例子：

```
#include <iostream>
using namespace std;
class A{
public:
    static int get(void){
        return 100;
    }
};
class B{
public:
    static const char* get(void){
        return "www.123.com";
    }
};
template <typename T>
void func(void){
    auto val = T::get();
    cout << val << endl;
}
int main(void){
    func<A>();
    func<B>();
```

```
    return 0;
}

/*          运行结果：
              100
              www.123.com
*/
```

本例中的模板函数 func() 会调用所有类的静态函数 get()，并对它的返回值做统一处理，但是 get() 的返回值类型并不一样，而且不能自动转换。这种要求在以前的 C++ 版本中实现起来非常的麻烦，需要额外增加一个模板参数，并在调用时手动给该模板参数赋值，用以指明变量 val 的类型。但是有了 auto 类型自动推导，编译器就根据 get() 的返回值自己推导出 val 变量的类型，就不用再增加一个模板参数了。

答案解析

无

1.5.13 简述一下 C++11 中的可变参数模板新特性

参考回答

可变参数模板(variadic template)使得编程者能够创建这样的模板函数和模板类，即可接受可变数量的参数。例如要编写一个函数，它可接受任意数量的参数，参数的类型只需是cout能显示的即可，并将参数显示为用逗号分隔的列表。

```
int n = 14;
double x = 2.71828;
std::string mr = "Mr.String objects!";
show_list(n, x);
show_list(x*x, '!', 7, mr); //这里的目标是定义show_list()

/*     运行结果：
          14, 2.71828
          7.38905, !, 7, Mr.String objects!
*/
```

要创建可变参数模板，需要理解几个要点：

（1）模板参数包（parameter pack）；

（2）函数参数包；

（3）展开（unpack）参数包；

（4）递归。

答案解析

无

1.5.14 简述一下 C++11 中 Lambda 新特性

参考回答

1. 定义

lambda 匿名函数很简单，可以套用如下的**语法格式**：

[外部变量访问方式说明符] (参数) mutable noexcept/throw() -> 返回值类型
{
函数体;
};

其中各部分的含义分别为：

a. **[外部变量方位方式说明符]**

1 [ ] 方括号用于向编译器表明当前是一个 lambda 表达式，其不能被省略。在方括号内部，可以注明当前 lambda 函数的函数体中可以使用哪些“外部变量”。

所谓外部变量，指的是和当前 lambda 表达式位于同一作用域内的所有局部变量。

b. **(参数)**

1 和普通函数的定义一样，lambda 匿名函数也可以接收外部传递的多个参数。和普通函数不同的是，如果不需要传递参数，可以连同 () 小括号一起省略；

c. **mutable**

1 此关键字可以省略，如果使用则之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，对于以值传递方式引入的外部变量，不允许在 lambda 表达式内部修改它们的值（可以理解为这部分变量都是 const 常量）。而如果想修改它们，就必须使用 mutable 关键字。

注意:对于以值传递方式引入的外部变量，lambda 表达式修改的是拷贝的那一份，并不会修改真正的外部变量；

d. **noexcept/throw()**

1 可以省略，如果使用，在之前的 () 小括号将不能省略（参数个数可以为 0）。默认情况下，lambda 函数的函数体中可以抛出任何类型的异常。而标注 noexcept 关键字，则表示函数体内不会抛出任何异常；使用 throw() 可以指定 lambda 函数内部可以抛出的异常类型。

e. **-> 返回值类型**

1 指明 lambda 匿名函数的返回值类型。值得一提的是，如果 lambda 函数体内只有一个 return 语句，或者该函数返回 void，则编译器可以自行推断出返回值类型，此情况下可以直接省略"-> 返回值类型"。

f. **函数体**

1 和普通函数一样，lambda 匿名函数包含的内部代码都放置在函数体中。该函数体内除了可以使用指定传递进来的参数之外，还可以使用指定的外部变量以及全局范围内的所有全局变量。

2. 程序实例

```
1 #include <iostream>
2 #include <algorithm>
3 using namespace std;
4 int main()
5 {
6     int num[4] = {4, 2, 3, 1};
```

```
    //对 a 数组中的元素进行排序
    sort(num, num+4, [=](int x, int y) -> bool{ return x < y; } );
    for(int n : num){
        cout << n << " ";
    }
    return 0;
}

/*    程序运行结果：
        1 2 3 4
*/
```

2. C++操作系统

2.1 Linux中查看进程运行状态的指令、查看内存使用情况的指令、tar解压文件的参数。

参考回答

1. **查看进程运行状态的指令**：ps命令。“**ps -aux | grep PID**”，用来查看某PID进程状态
2. **查看内存使用情况的指令**：free命令。“**free -m**”，命令查看内存使用情况。
3. **tar解压文件的参数**：

```
五个命令中必选一个
      -c：建立压缩档案
      -x：解压
      -t：查看内容
      -r：向压缩归档文件末尾追加文件
      -u：更新原压缩包中的文件
这几个参数是可选的
      -z：有gzip属性的
      -j：有bz2属性的
      -Z：有compress属性的
      -v：显示所有过程
      -O：将文件解开到标准输出
```

答案解析

```
//ps使用示例
//显示当前所有进程
ps -A
//与grep联用查找某进程
ps -aux | grep apache

//查看进程运行状态、查看内存使用情况的指令均可使用top指令。
top
```

2.2 文件权限怎么修改

参考回答

Linux文件的基本权限就有九个，分别是owner/group/others三种身份各有自己的read/write/execute权限

修改权限指令：**chmod**

答案解析

举例：文件的权限字符为 -rwxrwxrwx 时，这九个权限是三个三个一组。其中，我们可以使用数字来代表各个权限。

各权限的分数对照如下：

r	w	x
4	2	1

每种身份(owner/group/others)各自的三个权限(r/w/x)分数是需要累加的，

例如当权限为： [-rwxrwx---]，则分数是：

owner = rwx = 4+2+1 = 7

group = rwx = 4+2+1 = 7

others= --- = 0+0+0 = 0

所以我们设定权限的变更时，该文件的权限数字就是770！变更权限的指令chmod的语法是这样的：

```
[root@www ~]# chmod [-R] xyz 文件或目录
选项与参数：
xyz : 就是刚刚提到的数字类型的权限属性，为 rwx 属性数值的相加。
-R : 进行递归(recursive)的持续变更，亦即连同次目录下的所有文件都会变更

# chmod 770 douya.c //即修改douya.c文件的权限为770
```

2.3 说说常用的Linux命令

参考回答

1. cd命令：用于切换当前目录
2. ls命令：查看当前文件与目录
3. grep命令：该命令常用于分析一行的信息，若当中有我们所需要的信息，就将该行显示出来，该命令通常与管道命令一起使用，用于对一些命令的输出进行筛选加工。
4. cp命令：复制命令
5. mv命令：移动文件或文件夹命令
6. rm命令：删除文件或文件夹命令
7. ps命令：查看进程情况
8. kill命令：向进程发送终止信号
9. tar命令：对文件进行打包，调用gzip或bzip对文件进行压缩或解压
10. cat命令：查看文件内容，与less、more功能相似
11. top命令：可以查看操作系统的信息，如进程、CPU占用率、内存信息等

12. pwd命令：命令用于显示工作目录。

2.4 说说如何以root权限运行某个程序。

参考回答

```
sudo chown root app（文件名）
sudo chmod u+s app（文件名）
```

输入上面两条指令后即可

2.5 说说软链接和硬链接的区别。

参考回答

1. **定义不同**

   软链接又叫符号链接，这个文件包含了另一个文件的路径名。可以是任意文件或目录，可以链接不同文件系统的文件。

   硬链接就是一个文件的一个或多个文件名。把文件名和计算机文件系统使用的节点号链接起来。因此我们可以用多个文件名与同一个文件进行链接，这些文件名可以在同一目录或不同目录。

2. **限制不同**

   硬链接只能对已存在的文件进行创建，不能交叉文件系统进行硬链接的创建；

   软链接可对不存在的文件或目录创建软链接；可交叉文件系统；

3. **创建方式不同**

   硬链接不能对目录进行创建，只可对文件创建；

   软链接可对文件或目录创建；

4. **影响不同**

   删除一个硬链接文件并不影响其他有相同 inode 号的文件。

   删除软链接并不影响被指向的文件，但若被指向的原文件被删除，则相关软连接被称为死链接（即 dangling link，若被指向路径文件被重新创建，死链接可恢复为正常的软链接）。

2.6 说说静态库和动态库怎么制作及如何使用，区别是什么。

参考回答

静态库的制作：

```
gcc  hello.c  -c //这样就生成hello.o目标文件

ar rcs libhello.a  hello.o//生成libhello.a静态库
```

静态库的使用：

```
gcc main.c -lhello -o staticLibrary//main.c和hello静态库链接，生成staticLibrary执行文件
/*
main.c:    是指main主函数
-lhello:   是我们生成的.a 文件砍头去尾（lib不要 .a也不要）前面加-l
-L:        是指告诉gcc编译器先从-L指定的路径去找静态库，默认是从/usr/lib/ 或者 /usr/local/lib/ 去找。
./:        是指当前路径的意思
staticLibrary:  是最后想生成的文件名（这里可随意起名字）
*/
```

动态库的制作：

```
gcc -shared -fpic hello.c -o libhello.so
-shared 指定生成动态库
-fpic ：fPIC选项作用于编译阶段，在生成目标文件时就得使用该选项，以生成位置无关的代码。
```

动态库的使用：

```
gcc main.c -lhello -L ./ -o dynamicDepot
/*
main.c:    是指main主函数
-lhello:   是我们生成的.so 文件砍头去尾（lib不要 .so也不要）前面加-l
-L:        是指告诉gcc编译器先从-L指定的路径去找静态库，默认是从/usr/lib/ 或者 /usr/local/lib/ 去找。
./:        是指当前路径的意思
dynamicDepot:  是最后想生成的文件名（这里可随意起名字）
*/
```

区别：

1. 静态库代码装载的速度快，执行速度略比动态库快。
2. 动态库更加节省内存，可执行文件体积比静态库小很多。
3. 静态库是在编译时加载，动态库是在运行时加载。
4. 生成的静态链接库，Windows下以.lib为后缀，Linux下以.a为后缀。生成的动态链接库，Windows下以.dll为后缀，Linux下以.so为后缀。

2.7 简述GDB常见的调试命令，什么是条件断点，多进程下如何调试。

参考回答

GDB调试：gdb调试的是可执行文件，在gcc编译时加入 -g ，告诉gcc在编译时加入调试信息，这样gdb才能调试这个被编译的文件 gcc -g tesst.c -o test

GDB命令格式：

1. quit：退出gdb，结束调试
2. list：查看程序源代码

   list 5，10：显示5到10行的代码

   list test.c:5, 10: 显示源文件5到10行的代码，在调试多个文件时使用

   list get_sum: 显示get_sum函数周围的代码

list test,c get_sum: 显示源文件get_sum函数周围的代码，在调试多个文件时使用

3. reverse-search：字符串用来从当前行向前查找第一个匹配的字符串
4. run：程序开始执行
5. help list/all：查看帮助信息
6. break：设置断点

   break 7：在第七行设置断点

   break get_sum：以函数名设置断点

   break 行号或者函数名 if 条件：以条件表达式设置断点
7. watch 条件表达式：条件表达式发生改变时程序就会停下来
8. next：继续执行下一条语句 ， 会把函数当作一条语句执行
9. step：继续执行下一条语句，会跟踪进入函数，一次一条的执行函数内的代码

条件断点：break if 条件 以条件表达式设置断点

多进程下如何调试：用set follow-fork-mode child 调试子进程

或者set follow-fork-mode parent 调试父进程

2.8 说说什么是大端小端，如何判断大端小端?

参考回答

小端模式：**低**的有效字节存储在**低的**存储器地址。小端一般为主机字节序；常用的X86结构是小端模式。很多的ARM，DSP都为小端模式。

大端模式：**高**的有效字节存储在**低的**存储器地址。大端为网络字节序；KEIL C51则为大端模式。

有些ARM处理器还可以由硬件来选择是大端模式还是小端模式。

如何判断：我们可以根据**联合体**来判断系统是大端还是小端。因为联合体变量总是从**低地址**存储。

```
int fun1(){
    union test{
        char c;
        int i;
    };
    test t; t.i = 1;
    //如果是大端，则t.c为0x00，则t.c != 1，反之是小端
    return (t.c == 1);
}
```

答案解析

1. **在进行网络通信时是否需要进行字节序转换?**

   相同字节序的平台在进行网络通信时可以不进行字节序转换，但是跨平台进行网络数据通信时必须进行字节序转换。

   原因如下：网络协议规定接收到得第一个字节是高字节，存放到低地址，所以发送时会首先去低地址取数据的高字节。小端模式的多字节数据在存放时，低地址存放的是低字节，而被发送方网络协议函数发送时会首先去低地址取数据（想要取高字节，真正取得是低字节），接收方网络协议函数接收时会将接收到的第一个字节存放到低地址（想要接收高字节，真正接收的是低字节），所以最后双方都正确的收发了数据。而相同平台进行通信时，如果双方都进行转换最后虽然能够正确收发

数据，但是所做的转换是没有意义的，造成资源的浪费。而不同平台进行通信时必须进行转换，不转换会造成错误的收发数据，字节序转换函数会根据当前平台的存储模式做出相应正确的转换，如果当前平台是大端，则直接返回不进行转换，如果当前平台是小端，会将接收到得网络字节序进行转换。

2. **网络字节序**

   网络上传输的数据都是字节流,对于一个多字节数值,在进行网络传输的时候,先传递哪个字节?也就是说,当接收端收到第一个字节的时候,它将这个字节作为高位字节还是低位字节处理,是一个比较有意义的问题; UDP/TCP/IP协议规定:把接收到的第一个字节当作高位字节看待,这就要求发送端发送的第一个字节是高位字节;而在发送端发送数据时,发送的第一个字节是该数值在内存中的起始地址处对应的那个字节,也就是说,该数值在内存中的起始地址处对应的那个字节就是要发送的第一个高位字节(即:高位字节存放在低地址处);由此可见,多字节数值在发送之前,在内存中因该是以大端法存放的; 所以说,网络字节序是大端字节序; 比如,我们经过网络发送整型数值0x12345678时,在80X86平台中,它是以小端发存放的,在发送之前需要使用系统提供的字节序转换函数htonl()将其转换成大端法存放的数值;

2.9 说说进程调度算法有哪些?

参考回答

1. 先来先服务调度算法
2. 短作业(进程)优先调度算法
3. 高优先级优先调度算法
4. 时间片轮转法
5. 多级反馈队列调度算法

答案解析

1. 先来先服务调度算法：每次调度都是从后备作业（进程）队列中选择一个或多个最先进入该队列的作业（进程），将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。
2. 短作业(进程)优先调度算法：短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业（进程），将它们调入内存运行。
3. 高优先级优先调度算法：当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程
4. 时间片轮转法：每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。
5. 多级反馈队列调度算法：综合前面多种调度算法。

在这些调度算法中，有抢占式和非抢占式的区别。

1. 非抢占式优先权算法
   在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。
2. 抢占式优先权调度算法
   在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执

行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

区别：

非抢占式（Nonpreemptive）：让进程运行直到结束或阻塞的调度方式，容易实现，适合专用系统，不适合通用系统。
抢占式（Preemptive）：允许将逻辑上可继续运行的在运行过程暂停的调度方式可防止单一进程长时间独占，CPU系统开销大（降低途径：硬件实现进程切换，或扩充主存以贮存大部分程序）

2.10 简述操作系统如何申请以及管理内存的?

参考回答

操作系统如何管理内存：

1. **物理内存**：物理内存有四个层次，分别是寄存器、高速缓存、主存、磁盘。

   寄存器：速度最快、量少、价格贵。

   高速缓存：次之。

   主存：再次之。

   磁盘：速度最慢、量多、价格便宜。

   操作系统会对物理内存进行管理，有一个部分称为**内存管理器(memory manager)**，它的主要工作是有效的管理内存，记录哪些内存是正在使用的，在进程需要时分配内存以及在进程完成时回收内存。

2. **虚拟内存**：操作系统为每一个进程分配一个独立的地址空间，但是虚拟内存。虚拟内存与物理内存存在映射关系，通过页表寻址完成虚拟地址和物理地址的转换。

操作系统如何申请内存：

从操作系统角度来看，进程分配内存有两种方式，分别由两个系统调用完成：*brk*和*mmap*

2.11 简述Linux系统态与用户态，什么时候会进入系统态?

参考回答

1. **内核态与用户态**：**内核态**（系统态）与**用户态**是操作系统的两种运行级别。内核态拥有最高权限，可以访问所有系统指令；用户态则只能访问一部分指令。
2. **什么时候进入内核态**：共有三种方式：a、**系统调用。**b、**异常。**c、**设备中断。**其中，系统调用是主动的，另外两种是被动的。
3. **为什么区分内核态与用户态**：在CPU的所有指令中，有一些指令是非常危险的，如果错用，将导致整个系统崩溃。比如：清内存、设置时钟等。所以区分内核态与用户态主要是出于安全的考虑。

2.12 简述LRU算法及其实现方式。

参考回答

1. **LRU算法**：LRU算法用于缓存淘汰。思路是将缓存中最近最少使用的对象删除掉
2. **实现方式**：利用**链表和hashmap。**

   当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。

   在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。

答案解析

给出C++实现的代码

```
class LRUCache {
    list<pair<int, int>> cache;//创建双向链表
    unordered_map<int, list<pair<int, int>>::iterator> map;//创建哈希表
    int cap;
public:
    LRUCache(int capacity) {
        cap = capacity;
    }

    int get(int key) {
        if (map.count(key) > 0){
            auto temp = *map[key];
            cache.erase(map[key]);
            map.erase(key);
            cache.push_front(temp);
            map[key] = cache.begin();//映射头部
            return temp.second;
        }
        return -1;
    }

    void put(int key, int value) {
        if (map.count(key) > 0){
            cache.erase(map[key]);
            map.erase(key);
        }
        else if (cap == cache.size()){
            auto temp = cache.back();
            map.erase(temp.first);
            cache.pop_back();
        }
        cache.push_front(pair<int, int>(key, value));
        map[key] = cache.begin();//映射头部
    }
};

/**
* Your LRUCache object will be instantiated and called as such:
* LRUCache* obj = new LRUCache(capacity);
```

```
40 * int param_1 = obj->get(key);
41 * obj->put(key,value);
42 */
```

2.13 一个线程占多大内存?

参考回答

一个linux的线程大概占8M内存。

答案解析

linux的栈是通过缺页来分配内存的，不是所有栈地址空间都分配了内存。因此，8M是最大消耗，实际的内存消耗只会略大于实际需要的内存(内部损耗，每个在4k以内)。

2.14 什么是页表，为什么要有?

参考回答

页表是虚拟内存的概念。**操作系统虚拟内存到物理内存的映射表，就被称为页表。**

原因：不可能每一个虚拟内存的 Byte 都对应到物理内存的地址。这张表将大得真正的物理地址也放不下，于是操作系统引入了页（Page）的概念。进行分页，这样可以减小虚拟内存页对应物理内存页的映射表大小。

答案解析

如果将每一个虚拟内存的 Byte 都对应到物理内存的地址，每个条目最少需要 8字节（32位虚拟地址->32位物理地址），在 4G 内存的情况下，就需要 32GB 的空间来存放对照表，那么这张表就大得真正的物理地址也放不下了，于是操作系统引入了页（Page）的概念。

在系统启动时，操作系统将整个物理内存以 4K 为单位，划分为各个页。之后进行内存分配时，都以页为单位，那么虚拟内存页对应物理内存页的映射表就大大减小了，4G 内存，只需要 8M 的映射表即可，一些进程没有使用到的虚拟内存，也并不需要保存映射关系，而且Linux 还为大内存设计了多级页表，可以进一页减少了内存消耗。

2.15 简述操作系统中的缺页中断。

参考回答

1. **缺页异常**：malloc和mmap函数在分配内存时只是建立了进程虚拟地址空间，并没有分配虚拟内存对应的物理内存。当进程访问这些没有建立映射关系的虚拟内存时，处理器自动触发一个**缺页异常，引发缺页中断。**
2. **缺页中断**：缺页异常后将产生一个缺页中断，此时操作系统会根据页表中的**外存地址**在外存中找到所缺的一页，将其调入**内存。**

答案解析

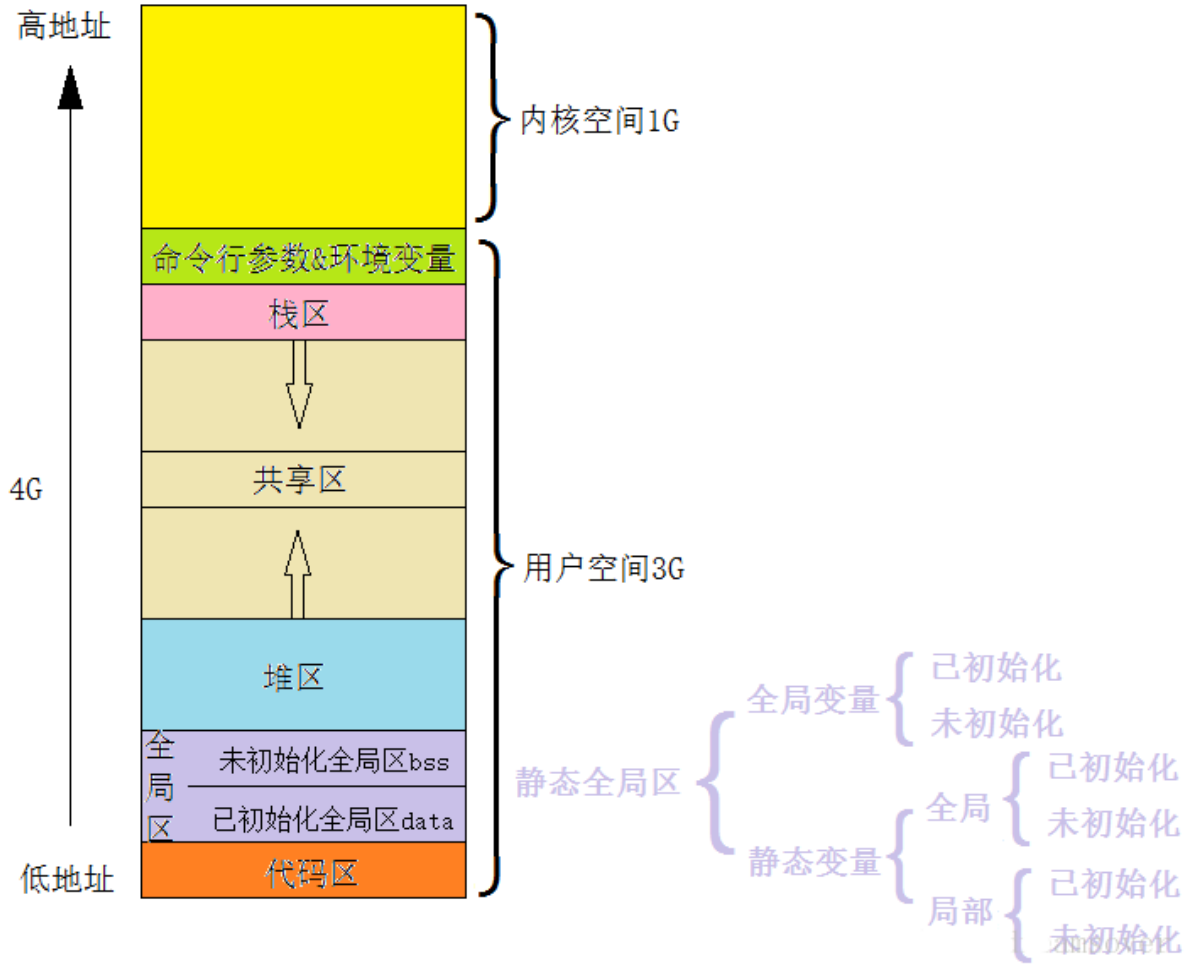
两者区别。

缺页中断与一般中断一样，需要经历四个步骤：保护CPU现场、分析中断原因、转入缺页中断处理程序、恢复CPU现场，继续执行。
缺页中断与一般中断区别：
（1）在指令执行期间产生和处理缺页中断信号
（2）一条指令在执行期间，可能产生多次缺页中断
（3）缺页中断返回的是执行产生中断的一条指令，而一般中断返回的是执行下一条指令。

2.16 说说虚拟内存分布，什么时候会由用户态陷入内核态?

参考回答

1. **虚拟内存分布**：



用户空间：

（1）**代码段.text**：存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

（2）**数据段.data**：存放程序中已初始化的全局变量和静态变量的一块内存区域。

（3）**BSS 段.bss**：存放程序中未初始化的全局变量和静态变量的一块内存区域。

（4）可执行程序在运行时又会多出两个区域：**堆区和栈区**。

堆区：动态申请内存用。堆从低地址向高地址增长。

栈区：存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。

（5）最后还有一个**文件映射区**，位于堆和栈之间。

内核空间：DMA区、常规区、高位区。

1. **什么时候进入内核态**：共有三种方式：a、**系统调用。**b、**异常。**c、**设备中断。**其中，系统调用是主动的，另外两种是被动的。

2.17 简述一下虚拟内存和物理内存，为什么要用虚拟内存，好处是什么?

参考回答

1. **物理内存**：物理内存有四个层次，分别是寄存器、高速缓存、主存、磁盘。

   寄存器：速度最快、量少、价格贵。

   高速缓存：次之。

   主存：再次之。

   磁盘：速度最慢、量多、价格便宜。

   操作系统会对物理内存进行管理，有一个部分称为**内存管理器(memory manager)**，它的主要工作是有效的管理内存，记录哪些内存是正在使用的，在进程需要时分配内存以及在进程完成时回收内存。

2. **虚拟内存**：操作系统为每一个进程分配一个独立的地址空间，但是虚拟内存。虚拟内存与物理内存存在映射关系，通过页表寻址完成虚拟地址和物理地址的转换。

3. **为什么要用虚拟内存**：因为早期的内存分配方法存在以下问题：

   （1）进程地址空间不隔离。会导致数据被随意修改。

   （2）内存使用效率低。

   （3）程序运行的地址不确定。操作系统随机为进程分配内存空间，所以程序运行的地址是不确定的。

4. **使用虚拟内存的好处**：

   （1）扩大地址空间。每个进程独占一个4G空间，虽然真实物理内存没那么多。

   （2）内存保护：防止不同进程对物理内存的争夺和践踏，可以对特定内存地址提供写保护，防止恶意篡改。

   （3）可以实现内存共享，方便进程通信。

   （4）可以避免内存碎片，虽然物理内存可能不连续，但映射到虚拟内存上可以连续。

5. **使用虚拟内存的缺点**：

   （1）虚拟内存需要额外构建数据结构，占用空间。

   （2）虚拟地址到物理地址的转换，增加了执行时间。

   （3）页面换入换出耗时。

   （4）一页如果只有一部分数据，浪费内存。

2.18 虚拟地址到物理地址怎么映射的?

参考回答

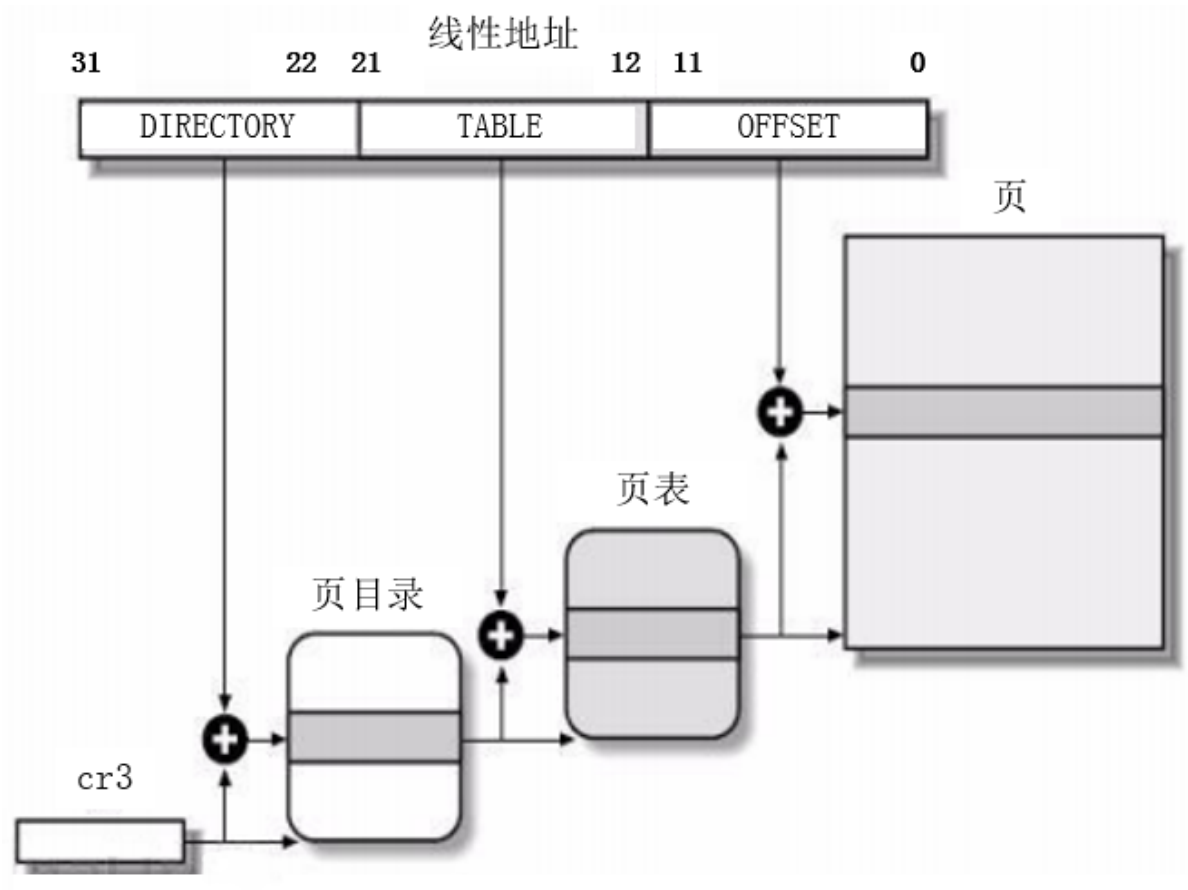
操作系统为每一个进程维护了一个从虚拟地址到物理地址的映射关系的数据结构，叫页表。页表中的每一项都记录了这个页的基地址。

三级页表转换方法：（两步）

1. 逻辑地址转线性地址：段起始地址+段内偏移地址=线性地址
2. 线性地址转物理地址：

   （1）每一个32位的线性地址被划分为三部分：页目录索引（DIRECTORY，10位）、页表索引（TABLE，10位）、页内偏移（OFFSET，12位）
   （2）从**cr3**中取出进程的页目录地址（操作系统调用进程时，这个地址被装入寄存器中）

```
页目录地址 + 页目录索引 = 页表地址
页表地址 + 页表索引 = 页地址
页地址 + 页内偏移 = 物理地址
```



按照以上两步法，就完成了一个三级页表从虚拟地址到物理地址的转换。

2.19 说说堆栈溢出是什么，会怎么样?

参考回答

堆栈溢出就是不顾堆栈中分配的局部数据块大小，向该数据块写入了过多的数据，导致数据越界。常指调用堆栈溢出，本质上一种数据结构的满溢情况。堆栈溢出可以理解为两个方面：**堆溢出和栈溢出。**

1. 堆溢出：比如不断的new 一个对象，一直创建新的对象，而不进行释放，最终导致内存不足。将会报错：OutOfMemory Error。
2. 栈溢出：一次函数调用中，栈中将被依次压入：参数，返回地址等，而方法如果递归比较深或进去死循环，就会导致栈溢出。将会报错：StackOverflow Error。

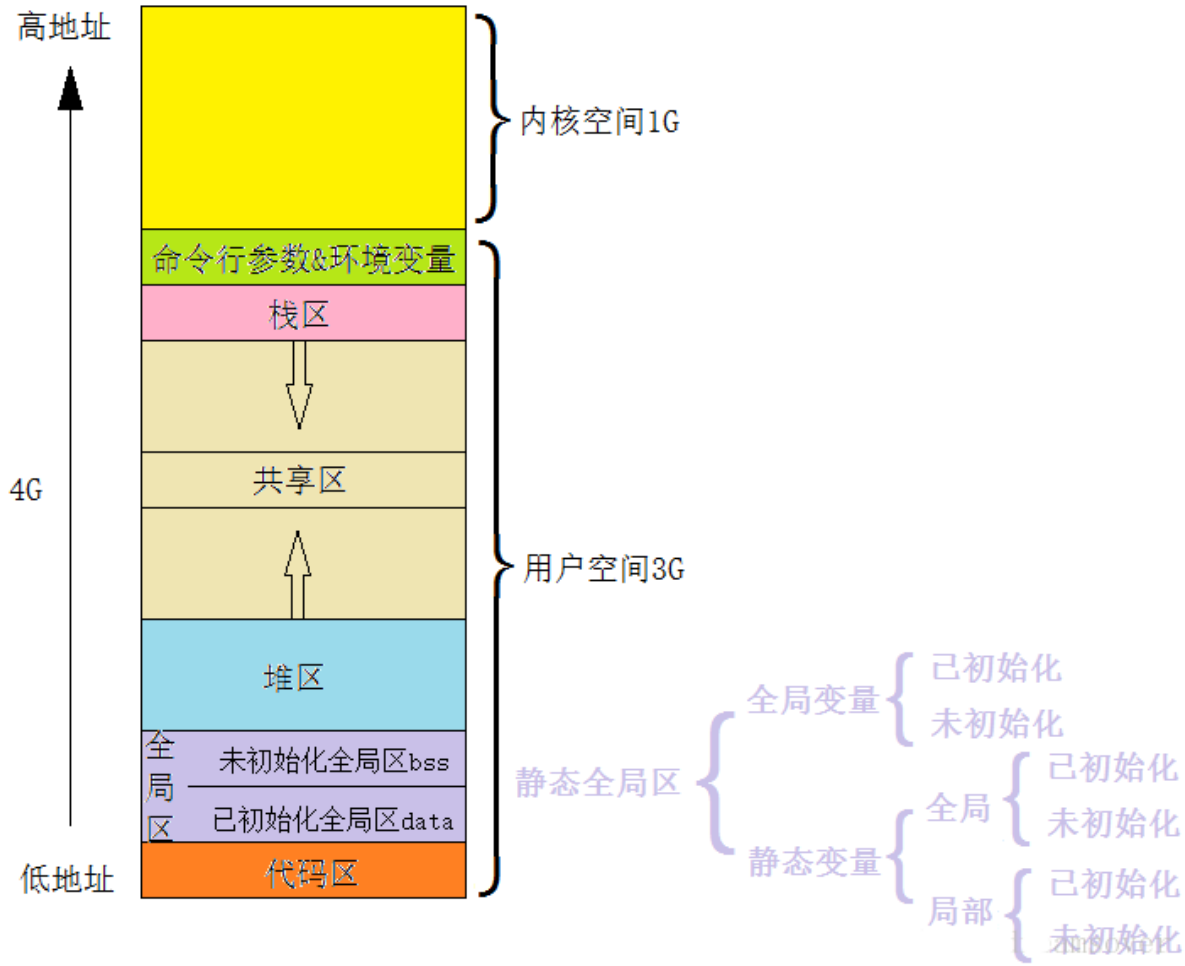
2.20 简述操作系统中malloc的实现原理

参考回答

malloc底层实现：当开辟的空间小于 128K 时，调用 brk（）函数；当开辟的空间大于 128K 时，调用 mmap（）。malloc采用的是内存池的管理方式，以减少内存碎片。先申请大块内存作为堆区，然后将堆区分为多个内存块。当用户申请内存时，直接从堆区分配一块合适的空闲快。采用隐式链表将所有空闲块，每一个空闲块记录了一个未分配的、连续的内存地址。

2.21 说说进程空间从高位到低位都有些什么?

参考回答



如上图，**从高地址到低地址，一个程序由命令行参数和环境变量、栈、文件映射区、堆、BSS段、数据段、代码段组成。**

1. **命令行参数和环境变量**
2. **栈区：**存储局部变量、函数参数值。栈从高地址向低地址增长。是一块连续的空间。
3. **文件映射区**，位于堆和栈之间。
4. **堆区：**动态申请内存用。堆从低地址向高地址增长。
5. **BSS 段**：存放程序中未初始化的全局变量和静态变量的一块内存区域。

6. **数据段**：存放程序中已初始化的全局变量和静态变量的一块内存区域。
7. **代码段：**存放程序执行代码的一块内存区域。只读，代码段的头部还会包含一些只读的常数变量。

2.22 32位系统能访问4GB以上的内存吗?

参考回答

正常情况下是不可以的。原因是计算机使用二进制，每位数只有0或1两个状态，32位正好是2的32次方，正好是4G，所以大于4G就没办法表示了，而在32位的系统中，因其它原因还需要占用一部分空间，所以内存只能识别3G多。要使用4G以上就只能换64位的操作系统了。

但是使用**PAE技术**就可以实现 32位系统能访问4GB以上的内存。

答案解析

Physical Address Extension（PAE）技术最初是为了弥补32位地址在PC服务器应用上的不足而推出的。我们知道，传统的IA32架构只有32位地址总线，只能让系统容纳不超过4GB的内存，这么大的内存，对于普通的桌面应用应该说是足够用了。可是，对于服务器应用来说，还是显得不足，因为服务器上可能承载了很多同时运行的应用。PAE技术将地址扩展到了36位，这样，系统就能够容纳2^36=64GB的内存。

2.23 请你说说并发和并行

参考回答

1. **并发**：对于单个CPU，在一个时刻只有一个进程在运行，但是线程的切换时间则减少到纳秒数量级，多个任务不停来回快速切换。
2. **并行**：对于多个CPU，多个进程同时运行。
3. **区别。**通俗来讲，它们虽然都说是"多个进程同时运行"，但是它们的"同时"不是一个概念。并行的"同时"是同一时刻可以多个任务在运行(处于running)，并发的"同时"是经过不同线程快速切换，使得看上去多个任务同时都在运行的现象。

2.24 说说进程、线程、协程是什么，区别是什么?

参考回答

1. **进程**：程序是指令、数据及其组织形式的描述，而进程则是程序的运行实例，包括程序计数器、寄存器和变量的当前值。
2. **线程**：微进程，一个进程里更小粒度的执行单元。一个进程里包含多个线程并发执行任务。
3. **协程**：协程是微线程，在子程序内部执行，可在子程序内部中断，转而执行别的子程序，在适当的时候再返回来接着执行。

区别：

1. **线程与进程的区别**：

   （1）一个线程从属于一个进程；一个进程可以包含多个线程。

   （2）一个线程挂掉，对应的进程挂掉；一个进程挂掉，不会影响其他进程。

   （3）进程是系统资源调度的最小单位；线程CPU调度的最小单位。

   （4）进程系统开销显著大于线程开销；线程需要的系统资源更少。

（5）进程在执行时拥有独立的内存单元，多个线程共享进程的内存，如代码段、数据段、扩展段；但每个线程拥有自己的栈段和寄存器组。

（6）进程切换时需要刷新TLB并获取新的地址空间，然后切换硬件上下文和内核栈，线程切换时只需要切换硬件上下文和内核栈。

（7）通信方式不一样。

（8）进程适应于多核、多机分布；线程适用于多核

2. **线程与协程的区别：**

（1）协程执行效率极高。协程直接操作栈基本没有内核切换的开销，所以上下文的切换非常快，切换开销比线程更小。

（2）协程不需要多线程的锁机制，因为多个协程从属于一个线程，不存在同时写变量冲突，效率比线程高。

（3）一个线程可以有多个协程。

2.25 请你说说Linux的fork的作用

参考回答

fork函数用来创建一个子进程。对于父进程，fork()函数返回新创建的子进程的PID。对于子进程，fork()函数调用成功会返回0。如果创建出错，fork()函数返回-1。

答案解析

fork()函数，其原型如下：

```
#include <unistd.h>
pid_t fork(void);
```

fork()函数不需要参数，返回值是一个进程标识符PID。返回值有以下三种情况：

（1） 对于父进程，fork()函数返回新创建的子进程的PID。
（2） 对于子进程，fork()函数调用成功会返回0。
（3） 如果创建出错，fork()函数返回-1。

fork()函数创建一个新进程后，会为这个新进程分配进程空间，将父进程的进程空间中的内容复制到子进程的进程空间中，包括父进程的数据段和堆栈段，并且和父进程共享代码段。这时候，子进程和父进程一模一样，都接受系统的调度。因为两个进程都停留在fork()函数中，最后fork()函数会返回两次，一次在父进程中返回，一次在子进程中返回，两次返回的值不一样，如上面的三种情况。

2.26 请你说说什么是孤儿进程，什么是僵尸进程，如何解决僵尸进程

参考回答

1. **孤儿进程**：是指一个父进程退出后，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。孤儿进程将被init进程（进程号为1）所收养，并且由init进程对它们完整状态收集工作。
2. **僵尸进程**：是指一个进程使用fork函数创建子进程，如果子进程退出，而父进程并没有调用wait()或者waitpid()系统调用取得子进程的终止状态，那么子进程的进程描述符仍然保存在系统中，占用系统资源，这种进程称为僵尸进程。
3. **如何解决僵尸进程**：

（1）一般，为了防止产生僵尸进程，在fork子进程之后我们都要及时使用**wait系统调用**；同时，当子进程退出的时候，内核都会给父进程一个SIGCHLD信号，所以我们可以建立一个捕获SIGCHLD信号的信号处理函数，在函数体中调用wait（或waitpid），就可以清理退出的子进程以达到防止僵尸进程的目的。

（2）**使用kill命令。**

打开终端并输入下面命令:

```
    ps aux | grep Z
```

会列出进程表中所有僵尸进程的详细内容。

然后输入命令：

```
    kill -s SIGCHLD pid(父进程pid)
```

2.27 请你说说什么是守护进程，如何实现?

参考回答

1. **守护进程**：守护进程是运行在后台的一种生存期长的特殊进程。它独立于控制终端，处理一些系统级别任务。
2. **如何实现**：

   （1）创建子进程，终止父进程。方法是调用fork() 产生一个子进程，然后使父进程退出。

   （2）调用setsid() 创建一个新会话。

   （3）将当前目录更改为根目录。使用fork() 创建的子进程也继承了父进程的当前工作目录。

   （4）重设文件权限掩码。文件权限掩码是指屏蔽掉文件权限中的对应位。

   （5）关闭不再需要的文件描述符。子进程从父进程继承打开的文件描述符。

答案解析

实现代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/stat.h>

#define MAXFILE 65535

int main(){
    //第一步:创建进程
    int pid = fork();
    if (pid > 0)
        exit(0);//结束父进程
    else if (pid < 0){
        printf("fork error!\n");
        exit(1);//fork失败，退出
```

```
    }
    //第二步:子进程成为新的会话组长和进程组长,并与控制终端分离
       setsid();
    //第三步:改变工作目录到
    chdir("/");
    //第四步:重设文件创建掩模
    umask(0);
    //第五步:关闭打开的文件描述符
    for (int i=0; i<MAXFILE; ++i)
        close(i);
        sleep(2);
    }
    return 0;
}
```

2.28 说说进程通信的方式有哪些?

参考回答

进程间通信主要包括**管道**、**系统IPC**（包括消息队列、信号量、信号、共享内存）、**套接字socket**。

1. **管道**：包括无名管道和命名管道，无名管道半双工，只能用于具有亲缘关系的进程直接的通信（父子进程或者兄弟进程），可以看作一种特殊的文件；命名管道可以允许无亲缘关系进程间的通信。
2. **系统IPC**

   消息队列：消息的链接表，放在内核中。消息队列独立于发送与接收进程，进程终止时，消息队列及其内容并不会被删除；消息队列可以实现消息的随机查询，可以按照消息的类型读取。

   信号量semaphore：是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步。

   信号：用于通知接收进程某个事件的发生。

   内存共享：使多个进程访问同一块内存空间。
3. **套接字socket**：用于不同主机直接的通信。

2.29 说说进程同步的方式?

参考回答

1. **信号量semaphore**：是一个计数器，可以用来控制多个进程对共享资源的访问。信号量用于实现进程间的互斥与同步。P操作(递减操作)可以用于阻塞一个进程，V操作(增加操作)可以用于解除阻塞一个进程。
2. **管道**：一个进程通过调用管程的一个过程进入管程。在任何时候，只能有一个进程在管程中执行，调用管程的任何其他进程都被阻塞，以等待管程可用。
3. **消息队列**：消息的链接表，放在内核中。消息队列独立于发送与接收进程，进程终止时，消息队列及其内容并不会被删除；消息队列可以实现消息的随机查询，可以按照消息的类型读取。

2.30 说说Linux进程调度算法及策略有哪些?

参考回答

1. 先来先服务调度算法
2. 短作业(进程)优先调度算法
3. 高优先级优先调度算法
4. 时间片轮转法
5. 多级反馈队列调度算法

答案解析

1. 先来先服务调度算法：每次调度都是从后备作业（进程）队列中选择一个或多个最先进入该队列的作业（进程），将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。
2. 短作业(进程)优先调度算法：短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业（进程），将它们调入内存运行。
3. 高优先级优先调度算法：当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程
4. 时间片轮转法：每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。
5. 多级反馈队列调度算法：综合前面多种调度算法。

在这些调度算法中，有抢占式和非抢占式的区别。

1. 非抢占式优先权算法
   在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。
2. 抢占式优先权调度算法
   在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

区别：

非抢占式（Nonpreemptive）：让进程运行直到结束或阻塞的调度方式，容易实现，适合专用系统，不适合通用系统。
抢占式（Preemptive）：允许将逻辑上可继续运行的在运行过程暂停的调度方式可防止单一进程长时间独占，CPU系统开销大（降低途径：硬件实现进程切换，或扩充主存以贮存大部分程序）

2.31 说说进程有多少种状态?

参考回答

进程有五种状态：**创建、就绪、执行、阻塞、终止。**一个进程创建后，被放入队列处于就绪状态，等待操作系统调度执行，执行过程中可能切换到阻塞状态（并发），任务完成后，进程销毁终止。

答案解析

创建状态
一个应用程序从系统上启动，首先就是进入**创建状态**，需要获取系统资源创建进程管理块（PCB：Process Control Block）完成资源分配。

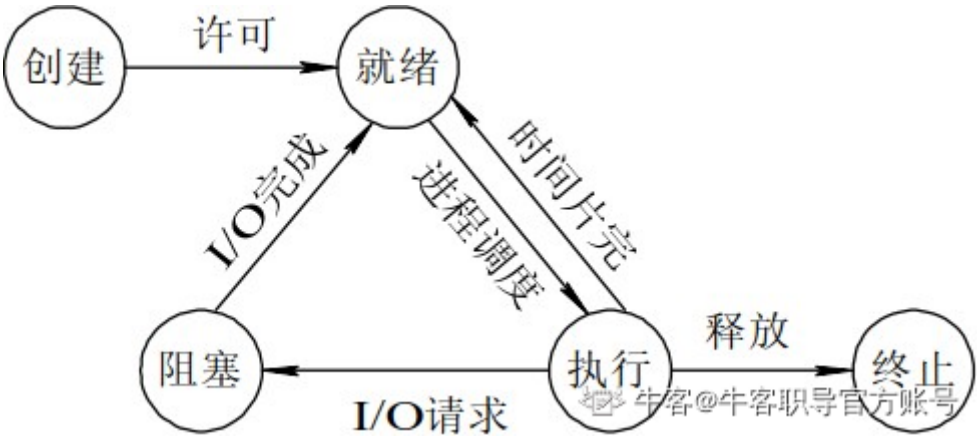
就绪状态
在**创建状态**完成之后，进程已经准备好，处于**就绪状态**，但是还未获得处理器资源，无法运行。

运行状态
获取处理器资源，被系统调度，**当具有时间片**开始进入**运行状态。**如果进程的时间片用完了就进入**就绪状态。**

阻塞状态
在**运行状态**期间，如果进行了阻塞的操作，如耗时的I/O操作，此时进程暂时无法操作就进入到了**阻塞状态**，在这些操作完成后就进入**就绪状态。**等待再次获取处理器资源，被系统调度，**当具有时间片**就进入**运行状态。**

终止状态
进程结束或者被系统终止，进入**终止状态**

相互转换如图:



2.32 进程通信中的管道实现原理是什么?

参考回答

操作系统在内核中开辟一块**缓冲区**（称为**管道**）用于通信。**管道**是一种两个进程间进行**单向通信**的机制。因为这种单向性，管道又称为半双工管道，所以其使用是有一定的局限性的。半双工是指数据只能由一个进程流向另一个进程（一个管道负责读，一个管道负责写）；如果是全双工通信，需要建立两个管道。管道分为无名管道和命名管道，无名管道只能用于具有亲缘关系的进程直接的通信（父子进程或者兄弟进程），可以看作一种特殊的文件，**管道本质是一种文件**；命名管道可以允许无亲缘关系进程间的通信。

管道原型如下:

```
#include <unistd.h>
int pipe(int fd[2]);
```

pipe()函数创建的管道处于一个进程中间，因此一个进程在由 pipe()创建管道后，一般再使用fork() 建立一个子进程，然后通过管道实现父子进程间的通信。管道两端可分别用描述字fd[0]以及fd[1]来描述。注意管道的两端的任务是固定的，即一端只能用于读，由描述字fd[0]表示，称其为管道读端；另 一端则只能用于写，由描述字fd[1]来表示，称其为管道写端。如果试图从管道写端读取数据，或者向管道读端写

入数据都将发生错误。一般文件的 I/O 函数都可以用于管道，如close()、read()、write()等。

具体步骤如下：

1. 父进程调用pipe开辟管道,得到两个文件描述符指向管道的两端。
2. 父进程调用fork创建子进程,那么子进程也有两个文件描述符指向同一管道。
3. 父进程关闭管道读端,子进程关闭管道写端。父进程可以往管道里写,子进程可以从管道里读,管道是用环形队列实现的,数据从写端流入从读端流出,这样就实现了进程间通信。

答案解析

给出实现的代码，实现父子进程间的管道通信

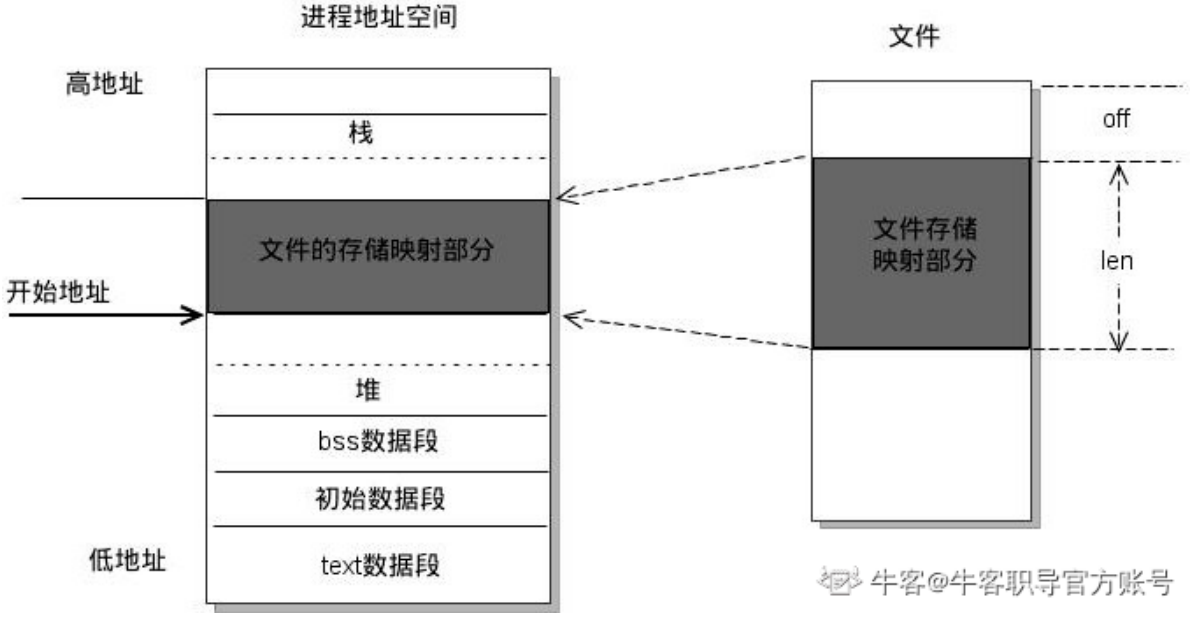
```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define INPUT  0
#define OUTPUT 1

int main(){
    //创建管道
    int fd[2];
    pipe(fd);
    //创建子进程
    pid_t pid = fork();
    if (pid < 0){
        printf("fork error!\n");
        exit(-1);
    }
    else if (pid == 0){//执行子进程
        printf("Child process is starting...\n");
        //子进程向父进程写数据，关闭管道的读端
        close(fd[INPUT]);
        write(fd[OUTPUT], "hello douya!", strlen("hello douya!"));
        exit(0);
    }
    else{//执行父进程
        printf ("Parent process is starting......\n");
        //父进程从管道读取子进程写的数据 ，关闭管道的写端
        close(fd[OUTPUT]);
        char buf[255];
        int output = read(fd[INPUT], buf, sizeof(buf));
        printf("%d bytes of data from child process: %s\n", output, buf);
    }
    return 0;
}
```

2.33 简述mmap的原理和使用场景

参考回答

原理：**mmap是一种内存映射文件的方法**，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read, write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空

间，从而可以实现不同进程间的文件共享。如下图：



使用场景：

1. 对同一块区域频繁读写操作；
2. 可用于实现用户空间和内核空间的高效交互
3. 可提供进程间共享内存及相互通信
4. 可实现高效的大规模数据传输。

2.34 互斥量能不能在进程中使用?

参考回答

能。

不同的进程之间，存在资源竞争或并发使用的问题，所以需要**互斥量**。

进程中也需要**互斥量**，因为一个进程中可以包含多个线程，线程与线程之间需要通过互斥的手段进行同步，避免导致共享数据修改引起冲突。可以使用**互斥锁**，属于互斥量的一种。

2.35 协程是轻量级线程，轻量级表现在哪里?

参考回答

1. **协程调用跟切换比线程效率高**：协程执行效率极高。协程不需要多线程的锁机制，可以不加锁的访问全局变量，所以上下文的切换非常快。
2. **协程占用内存少**：执行协程只需要极少的栈内存（大概是4～5KB），而默认情况下，线程栈的大小为1MB。
3. **切换开销更少**：协程直接操作栈基本没有内核切换的开销，所以切换开销比线程少。

2.36 说说常见信号有哪些，表示什么含义?

参考回答

编号为1 ~ 31的信号为传统UNIX支持的信号，是不可靠信号(非实时的)。不可靠信号和可靠信号的区别在于前者不支持排队，可能会造成信号丢失，而后者不会。编号为1 ~ 31的信号如下：

信号代号	信号名称	说 明
1	SIGHUP	该信号让进程立即关闭.然后重新读取配置文件之后重启
2	SIGINT	程序中止信号，用于中止前台进程。相当于输出 Ctrl+C 快捷键
3	SIGQUIT	和SIGINT类似, 但由QUIT字符(通常是Ctrl-/)来控制. 进程在因收到SIGQUIT退出时会产生core文件, 在这个意义上类似于一个程序错误信号。
4	SIGILL	执行了非法指令. 通常是因为可执行文件本身出现错误, 或者试图执行数据段. 堆栈溢出时也有可能产生这个信号。
5	SIGTRAP	由断点指令或其它trap指令产生. 由debugger使用。
6	SIGABRT	调用abort函数生成的信号。
7	SIGBUS	非法地址, 包括内存地址对齐(alignment)出错。
8	SIGFPE	在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术运算错误
9	SIGKILL	用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。般用于强制中止进程
10	SIGUSR1	留给用户使用
11	SIGSEGV	试图访问未分配给自己的内存, 或试图往没有写权限的内存地址写数据.
12	SIGUSR2	留给用户使用
13	SIGPIPE	管道破裂。这个信号通常在进程间通信产生，比如采用FIFO(管道)通信的两个进程，读管道没打开或者意外终止就往管道写，写进程会收到SIGPIPE信号。
14	SIGALRM	时钟定时信号，计算的是实际的时间或时钟时间。alarm 函数使用该信号
15	SIGTERM	正常结束进程的信号，kill 命令的默认信号。如果进程已经发生了问题，那么这 个信号是无法正常中止进程的，这时我们才会尝试 SIGKILL 信号，也就是信号 9
17	SIGCHLD	子进程结束时, 父进程会收到这个信号。
18	SIGCONT	该信号可以让暂停的进程恢复执行。本信号不能被阻断
19	SIGSTOP	该信号可以暂停前台进程，相当于输入 Ctrl+Z 快捷键。本信号不能被阻断
20	SIGTSTP	停止进程的运行, 但该信号可以被处理和忽略. 用户键入SUSP字符时(通常是Ctrl-Z)发出这个信号
21	SIGTTIN	当后台作业要从用户终端读数据时, 该作业中的所有进程会收到SIGTTIN信号. 缺省时这些进程会停止执行.
22	SIGTTOU	类似于SIGTTIN, 但在写终端(或修改终端模式)时收到.
23	SIGURG	有"紧急"数据或out-of-band数据到达socket时产生.

信号代号	信号名称	说 明
24	SIGXCPU	超过CPU时间资源限制. 这个限制可以由getrlimit/setrlimit来读取/改变。
25	SIGXFSZ	当进程企图扩大文件以至于超过文件大小资源限制。
26	SIGVTALRM	虚拟时钟信号. 类似于SIGALRM, 但是计算的是该进程占用的CPU时间.
27	SIGPROF	类似于SIGALRM/SIGVTALRM, 但包括该进程用的CPU时间以及系统调用的时间.
28	SIGWINCH	窗口大小改变时发出.
29	SIGIO	文件描述符准备就绪, 可以开始进行输入/输出操作.
30	SIGPWR	Power failure
31	SIGSYS	非法的系统调用。

而常见信号如下:

信号代号	信号名称	说 明
1	SIGHUP	该信号让进程立即关闭.然后重新读取配置文件之后重启
2	SIGINT	程序中止信号，用于中止前台进程。相当于输出 Ctrl+C 快捷键
8	SIGFPE	在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其他所有的算术运算错误
9	SIGKILL	用来立即结束程序的运行。本信号不能被阻塞、处理和忽略。般用于强制中止进程
14	SIGALRM	时钟定时信号，计算的是实际的时间或时钟时间。alarm 函数使用该信号
15	SIGTERM	正常结束进程的信号，kill 命令的默认信号。如果进程已经发生了问题，那么这 个信号是无法正常中止进程的，这时我们才会尝试 SIGKILL 信号，也就是信号 9
17	SIGCHLD	子进程结束时, 父进程会收到这个信号。
18	SIGCONT	该信号可以让暂停的进程恢复执行。本信号不能被阻断
19	SIGSTOP	该信号可以暂停前台进程，相当于输入 Ctrl+Z 快捷键。本信号不能被阻断

其中最重要的就是 "1"、"9"、"15"、"17"这几个信号。

2.37 说说线程间通信的方式有哪些?

参考回答

线程间的通信方式包括**临界区、互斥量、信号量、条件变量、读写锁**:

1. 临界区：每个线程中访问临界资源的那段代码称为临界区（Critical Section）（临界资源是一次仅允许一个线程使用的共享资源）。每次只准许一个线程进入临界区，进入后不允许其他线程进入。不论是硬件临界资源，还是软件临界资源，多个线程必须互斥地对它进行访问。
2. 互斥量：采用互斥对象机制，只有拥有互斥对象的线程才可以访问。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。
3. 信号量：计数器，允许多个线程同时访问同一个资源。
4. 条件变量：通过条件变量通知操作的方式来保持多线程同步。
5. 读写锁：读写锁与互斥量类似。但互斥量要么是锁住状态，要么就是不加锁状态。读写锁一次只允许一个线程写，但允许一次多个线程读，这样效率就比互斥锁要高。

2.38 说说线程同步方式有哪些?

参考回答

线程间的同步方式包括**互斥锁、信号量、条件变量、读写锁**:

1. **互斥锁**：采用互斥对象机制，只有拥有互斥对象的线程才可以访问。因为互斥对象只有一个，所以可以保证公共资源不会被多个线程同时访问。
2. **信号量**：计数器，允许多个线程同时访问同一个资源。
3. **条件变量**：通过条件变量通知操作的方式来保持多线程同步。
4. **读写锁**：读写锁与互斥量类似。但互斥量要么是锁住状态，要么就是不加锁状态。读写锁一次只允许一个线程写，但允许一次多个线程读，这样效率就比互斥锁要高。

2.39 说说什么是死锁，产生的条件，如何解决?

参考回答

1. **死锁**: 是指多个进程在执行过程中，因争夺资源而造成了互相等待。此时系统产生了死锁。比如两只羊过独木桥，若两只羊互不相让，争着过桥，就产生死锁。
2. **产生的条件**：死锁发生有**四个必要条件**:
   （1）**互斥条件**：进程对所分配到的资源不允许其他进程访问，若其他进程访问，只能等待，直到进程使用完成后释放该资源；

   （2）**请求保持条件**：进程获得一定资源后，又对其他资源发出请求，但该资源被其他进程占有，此时请求阻塞，而且该进程不会释放自己已经占有的资源；

   （3）**不可剥夺条件**：进程已获得的资源，只能自己释放，不可剥夺；

   （4）**环路等待条件**：若干进程之间形成一种头尾相接的循环等待资源关系。
3. **如何解决**:

   （1）资源一次性分配，从而解决请求保持的问题

   （2）可剥夺资源：当进程新的资源未得到满足时，释放已有的资源；

   （3）资源有序分配：资源按序号递增，进程请求按递增请求，释放则相反。

答案解析

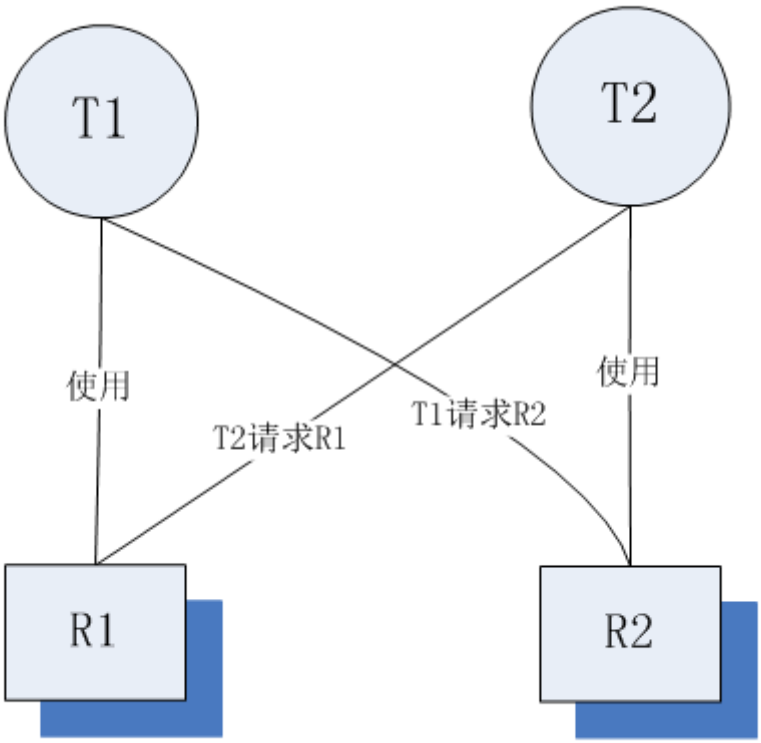
举个例子，比如：如果此时有两个线程T1和T2，它们分别占有R1和R2资源

此时，T1请求R2资源的同时，T2请求R1资源。

这个时候T2说：你把R1给我，我就给你R2

T1说：不行，你要先给我R2，我才能给你R1

那么就这样，死锁产生了。如下图：



2.40 有了进程，为什么还要有线程?

参考回答

1. **原因**

   进程在早期的多任务操作系统中是基本的**执行单元。**每次进程切换，都要先保存进程资源然后再恢复，这称为上下文切换。**但是进程频繁切换将引起额外开销，从而严重影响系统的性能。**为了减少进程切换的开销，人们把两个任务放到一个进程中，每个任务用一个更小**粒度**的执行单元来实现并发执行，这就是**线程。**

2. **线程与进程对比**

   （1）**进程间的信息难以共享。**由于除去只读代码段外，父子进程并未共享内存，因此必须采用一些进程间通信方式，在进程间进行信息交换。

   但**多个线程共享**进程的内存，如代码段、数据段、扩展段，线程间进行信息交换十分方便。

   （2）调用 fork() 来创建进程的代价相对较高，即便利用写时复制技术，仍然需要复制诸如内存页表和文件描述符表之类的多种进程属性，这意味着 fork() 调用在时间上的开销依然不菲。

   但创建线程比创建进程通常要快 10 倍甚至更多。线程间是共享虚拟地址空间的，无需采用写时复制来复制内存，也无需复制页表。

2.41 单核机器上写多线程程序，是否要考虑加锁，为什么?

参考回答

在单核机器上写多线程程序，仍然需要线程锁。

原因：因为线程锁通常用来实现线程的同步和通信。在单核机器上的多线程程序，仍然存在线程同步的问题。因为在抢占式操作系统中，通常为每个线程分配一个时间片，当某个线程时间片耗尽时，操作系统会将其挂起，然后运行另一个线程。如果这两个线程共享某些数据，**不使用线程锁的前提下，可能会导致共享数据修改引起冲突。**

2.42 说说多线程和多进程的不同?

参考回答

（1）一个线程从属于一个进程；一个进程可以包含多个线程。

（2）一个线程挂掉，对应的进程挂掉，多线程也挂掉；一个进程挂掉，不会影响其他进程，多进程稳定。

（3）进程系统开销显著大于线程开销；线程需要的系统资源更少。

（4）多个进程在执行时拥有各自独立的内存单元，多个线程共享进程的内存，如代码段、数据段、扩展段；但每个线程拥有自己的栈段和寄存器组。

（5）多进程切换时需要刷新TLB并获取新的地址空间，然后切换硬件上下文和内核栈；多线程切换时只需要切换硬件上下文和内核栈。

（6）通信方式不一样。

（7）多进程适应于多核、多机分布；多线程适用于多核

2.43 简述互斥锁的机制，互斥锁与读写的区别?

参考回答

1. **互斥锁机制**：mutex，用于保证在任何时刻，都只能有一个线程访问该对象。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒。
2. **互斥锁和读写锁**：

   （1） 读写锁区分读者和写者，而互斥锁不区分

   （2）互斥锁同一时间只允许一个线程访问该对象，无论读写；读写锁同一时间内只允许一个写者，但是允许多个读者同时读对象。

答案解析

原理详解：

互斥锁其实就是一个bool型变量，为true时表示锁可获取，为false时表示已上锁。这里说的是**互斥锁**，其实是泛指linux中所有的锁机制。

我们采用互斥锁保护临界区，从而防止竞争条件。也就是说，一个线程在进入临界区时应得到锁；它在退出临界区时释放锁。函数 acquire() 获取锁，而函数 release() 释放锁，如图：

每个互斥锁有一个布尔变量 available，它的值表示锁是否可用。如果锁是可用的，那么调用 acquire() 会成功，并且锁不再可用。当一个线程试图获取不可用的锁时，它会阻塞，直到锁被释放。

按如下定义 acquire()：

```
acquire() {
    while (!available);
    /* busy wait */
    available = false;
}
```

按如下定义release()：

```
release() {
    available = true;
}
```

2.44 说说什么是信号量，有什么作用?

参考回答

1. **概念**：信号量本质上是一个计数器，用于多进程对共享数据对象的读取，它主要是用来保护共享资源（信号量也属于临界资源），使得资源在一个时刻只有一个进程独享。
2. **原理**：由于信号量只能进行两种操作等待和发送信号，即P(sv)和V(sv)，具体的行为如下：

   （1）P(sv)操作：如果sv的值大于零，就给它减1；如果它的值为零，就挂起该进程的执行（信号量的值为正，进程获得该资源的使用权，进程将信号量减1，表示它使用了一个资源单位）。

   （2）V(sv)操作：如果有其他进程因等待sv而被挂起，就让它恢复运行，如果没有进程因等待sv而挂起，就给它加1（若此时信号量的值为0，则进程进入挂起状态，直到信号量的值大于0，若进程被唤醒则返回至第一步）。
3. **作用**：用于多进程对共享数据对象的读取，它主要是用来保护共享资源（信号量也属于临界资源），使得资源在一个时刻只有一个进程独享。

2.45 进程、线程的中断切换的过程是怎样的?

参考回答

上下文切换指的是内核（操作系统的核心）在CPU上对进程或者线程进行切换。

1. **进程上下文切换**

   （1）保护被中断进程的处理器现场信息

   （2）修改被中断进程的进程控制块有关信息，如进程状态等

   （3）把被中断进程的进程控制块加入有关队列

   （4）选择下一个占有处理器运行的进程

   （5）根据被选中进程设置操作系统用到的地址转换和存储保护信息

   切换页目录以使用新的地址空间

   切换内核栈和硬件上下文（包括分配的内存，数据段，堆栈段等）

   （6）根据被选中进程恢复处理器现场

2. **线程上下文切换**

   （1）保护被中断线程的处理器现场信息

   （2）修改被中断线程的线程控制块有关信息，如线程状态等

   （3）把被中断线程的线程控制块加入有关队列

   （4）选择下一个占有处理器运行的线程

   （5）根据被选中线程设置操作系统用到的存储保护信息

   切换内核栈和硬件上下文（切换堆栈，以及各寄存器）

   （6）根据被选中线程恢复处理器现场

2.46 简述自旋锁和互斥锁的使用场景

参考回答

1. **互斥锁**用于临界区持锁时间比较长的操作，比如下面这些情况都可以考虑

   （1）临界区有IO操作

   （2）临界区代码复杂或者循环量大

   （3）临界区竞争非常激烈

   （4）单核处理器

1. **自旋锁就**主要用在临界区持锁时间非常短且CPU资源不紧张的情况下。

2.47 请你说说线程有哪些状态，相互之间怎么转换?

参考回答

类似进程，有以下五种状态：

1. **新建状态(New)**
2. **就绪状态(Runnable)**

3. **运行状态(Running)**
4. **阻塞状态(Blocked)**
5. **死亡状态(Dead)**

转换方式如下：

创建状态
一个应用程序从系统上启动，首先就是进入**创建状态**，获取系统资源。

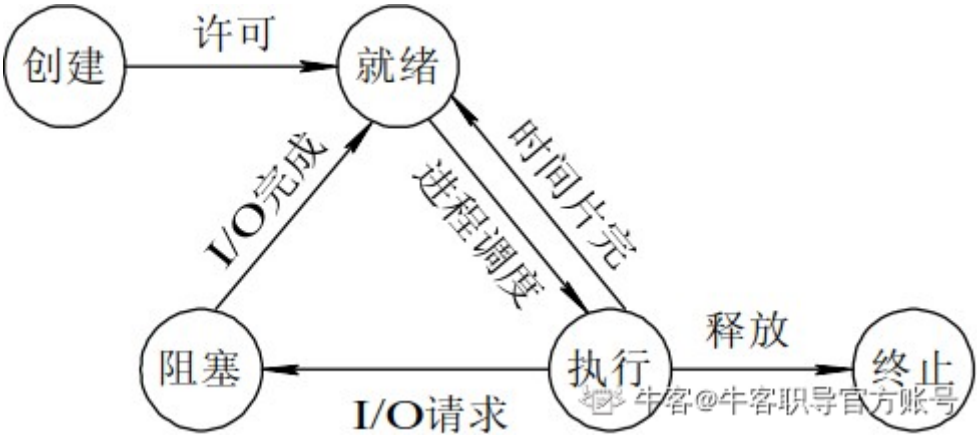
就绪状态
在**创建状态**完成之后，线程已经准备好，处于**就绪状态**，但是还未获得处理器资源，无法运行。

运行状态
获取处理器资源，被系统调度，当具有时间片开始进入**运行状态。**如果线程的时间片用完了就进入**就绪状态。**

阻塞状态
在**运行状态**期间，如果进行了阻塞的操作，如耗时的I/O操作，此时线程暂时无法操作就进入到了**阻塞状态**，在这些操作完成后就进入**就绪状态。**等待再次获取处理器资源，被系统调度，当具有时间片就进入**运行状态。**

终止状态
线程结束或者被系统终止，进入**终止状态**

相互转换如图：



2.48 多线程和单线程有什么区别，多线程编程要注意什么，多线程加锁需要注意什么?

参考回答

1. **区别**：

   （1）多线程从属于一个进程，单线程也从属于一个进程；一个线程挂掉都会导致从属的进程挂掉。

   （2）一个进程里有多个线程，可以并发执行多个任务；一个进程里只有一个线程，就只能执行一个任务。

   （3）多线程并发执行多任务，需要切换内核栈与硬件上下文，有切换的开销；单线程不需要切换，没有切换的开销。

   （4）多线程并发执行多任务，需要考虑同步的问题；单线程不需要考虑同步的问题。

2. 多线程编程需要考虑**同步**的问题。线程间的同步方式包括**互斥锁、信号量、条件变量、读写锁。**
3. 多线程加锁，主要需要注意**死锁**的问题。破坏死锁的必要条件从而避免死锁。

答案解析

1. **死锁**: 是指多个进程在执行过程中，因争夺资源而造成了互相等待。此时系统产生了死锁。比如两只羊过独木桥，若两只羊互不相让，争着过桥，就产生死锁。
2. **产生的条件**：死锁发生有**四个必要条件**：
   （1）**互斥条件**：进程对所分配到的资源不允许其他进程访问，若其他进程访问，只能等待，直到进程使用完成后释放该资源；
   （2）**请求保持条件**：进程获得一定资源后，又对其他资源发出请求，但该资源被其他进程占有，此时请求阻塞，而且该进程不会释放自己已经占有的资源；
   （3）**不可剥夺条件**：进程已获得的资源，只能自己释放，不可剥夺；
   （4）**环路等待条件**：若干进程之间形成一种头尾相接的循环等待资源关系。
3. **如何解决**：
   （1）资源一次性分配，从而解决请求保持的问题
   （2）可剥夺资源：当进程新的资源未得到满足时，释放已有的资源；
   （3）资源有序分配：资源按序号递增，进程请求按递增请求，释放则相反。

答案解析

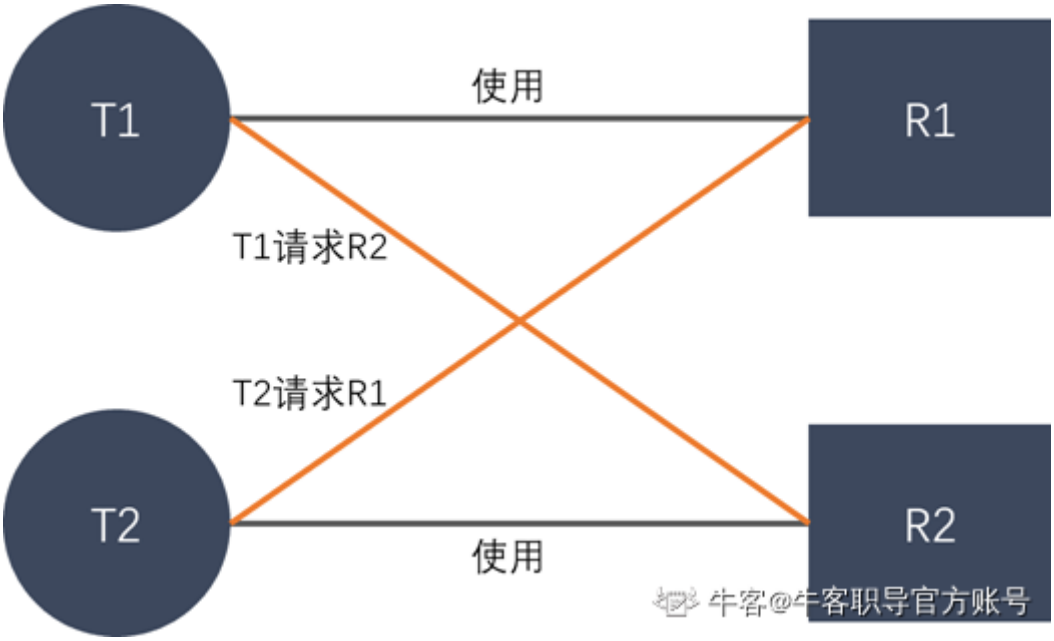
举个例子，比如：如果此时有两个线程T1和T2，它们分别占有R1和R2资源

此时，T1请求R2资源的同时，T2请求R1资源。

这个时候T2说：你把R1给我，我就给你R2

T1说：不行，你要先给我R2，我才能给你R1

那么就这样，死锁产生了。如下图：



2.49 说说sleep和wait的区别?

参考回答

1. **sleep**

   sleep是一个延时函数，让进程或线程进入休眠。休眠完毕后继续运行。

   在linux下面，sleep函数的参数是秒，而windows下面sleep的函数参数是毫秒。

windows下面sleep的函数参数是毫秒。

例如：

```
#include <windows.h>// 首先应该先导入头文件
Sleep (500) ; //注意第一个字母是大写。
//就是到这里停半秒，然后继续向下执行。
```

在 Linux C语言中 sleep的单位是秒

例如：

```
#include <unistd.h>// 首先应该先导入头文件
sleep(5); //停5秒
//就是到这里停5秒，然后继续向下执行。
```

2. **wait**

wait是父进程回收子进程PCB资源的一个系统调用。进程一旦调用了wait函数，就立即阻塞自己本身，然后由wait函数自动分析当前进程的某个子进程是否已经退出，当找到一个已经变成僵尸的子进程，wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞，直到有一个出现为止。函数原型如下：

```
#include<sys/types.h>
#include<sys/wait.h>

pid_t wait(int* status);
```

子进程的结束状态值会由参数status返回，而子进程的进程识别码也会一起返回。如果不需要结束状态值，则参数status可以设成 NULL。

3. **区别**：

（1）sleep是一个延时函数，让进程或线程进入休眠。休眠完毕后继续运行。

（2）wait是父进程回收子进程PCB（Process Control Block）资源的一个系统调用。

2.50 说说线程池的设计思路，线程池中线程的数量由什么确定?

参考回答

1. **设计思路**：

实现线程池有以下几个步骤：

（1）设置一个生产者消费者队列，作为临界资源。

（2）初始化n个线程，并让其运行起来，加锁去队列里取任务运行

（3）当任务队列为空时，所有线程阻塞。

（4）当生产者队列来了一个任务后，先对队列加锁，把任务挂到队列上，然后使用条件变量去通知阻塞中的一个线程来处理。

2. **线程池中线程数量**：

线程数量和哪些因素有关：CPU，IO、并行、并发

```
如果是CPU密集型应用，则线程池大小设置为：CPU数目+1
如果是IO密集型应用，则线程池大小设置为：2*CPU数目+1
最佳线程数目 = （线程等待时间与线程CPU时间之比 + 1）* CPU数目
```

所以线程等待时间所占比例越高，需要越多线程。线程CPU时间所占比例越高，需要越少线程。

答案解析

1. **为什么要创建线程池**：

   创建线程和销毁线程的花销是比较大的，这些时间有可能比处理业务的时间还要长。这样频繁的创建线程和销毁线程，再加上业务工作线程，消耗系统资源的时间，可能导致系统资源不足。**同时线程池也是为了提升系统效率。**

2. **线程池的核心线程与普通线程：**

   任务队列可以存放100个任务，此时为空，线程池里有10个核心线程，若突然来了10个任务，那么刚好10个核心线程直接处理；若又来了90个任务，此时核心线程来不及处理，那么有80个任务先入队列，再创建核心线程处理任务；若又来了120个任务，此时任务队列已满，不得已，就得创建20个普通线程来处理多余的任务。
   以上是线程池的工作流程。

2.51 进程和线程相比，为什么慢?

参考回答

1. 进程系统开销显著大于线程开销；线程需要的系统资源更少。
2. 进程切换开销比线程大。多进程切换时需要刷新TLB并获取新的地址空间，然后切换硬件上下文和内核栈；多线程切换时只需要切换硬件上下文和内核栈。
3. 进程通信比线程通信开销大。进程通信需要借助管道、队列、共享内存，需要额外申请空间，通信繁琐；而线程共享进程的内存，如代码段、数据段、扩展段，通信快捷简单，同步开销更小。

2.52 简述Linux零拷贝的原理?

参考回答

1. **什么是零拷贝**：

   所谓「零拷贝」描述的是计算机操作系统当中，CPU不执行将数据从一个内存区域，拷贝到另外一个内存区域的任务。通过网络传输文件时，这样通常可以节省 CPU 周期和内存带宽。

2. **零拷贝的好处**：

   （1）节省了 CPU 周期，空出的 CPU 可以完成更多其他的任务

   （2）减少了内存区域之间数据拷贝，节省内存带宽

   （3）减少用户态和内核态之间数据拷贝，提升数据传输效率

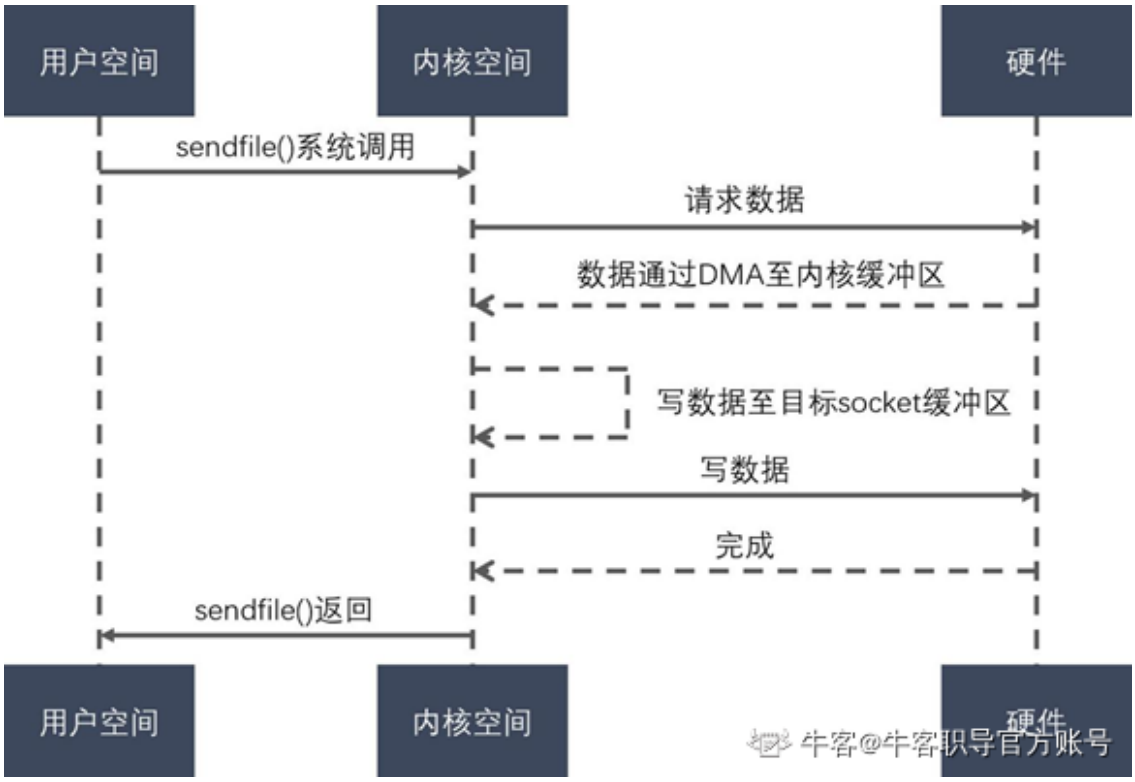
   （4）应用零拷贝技术，减少用户态和内核态之间的上下文切换

3. **零拷贝原理**：

   在传统 IO 中，用户态空间与内核态空间之间的复制是完全不必要的，因为用户态空间仅仅起到了一种数据转存媒介的作用，除此之外没有做任何事情。

   （1）Linux 提供了 sendfile() 用来减少我们的数据拷贝和上下文切换次数。

过程如图：



a. 发起 sendfile() 系统调用，操作系统由用户态空间切换到内核态空间（第一次上下文切换）

b. 通过 DMA 引擎将数据从磁盘拷贝到内核态空间的输入的 socket 缓冲区中（第一次拷贝）

c. 将数据从内核空间拷贝到与之关联的 socket 缓冲区（第二次拷贝）

d. 将 socket 缓冲区的数据拷贝到协议引擎中（第三次拷贝）

e. sendfile() 系统调用结束，操作系统由用户态空间切换到内核态空间（第二次上下文切换）

根据以上过程，一共有 2 次的上下文切换，3 次的 I/O 拷贝。我们看到从用户空间到内核空间并没有出现数据拷贝，**从操作系统角度来看，这个就是零拷贝。**内核空间出现了复制的原因: 通常的硬件在通过DMA访问时期望的是连续的内存空间。

（2）**mmap 数据零拷贝原理**

如果需要对数据做操作，Linux 提供了mmap 零拷贝来实现。

2.53 简述epoll和select的区别，epoll为什么高效?

参考回答

1. **区别**：

   （1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大；而epoll保证了每个fd在整个过程中只会拷贝一次。

   （2）每次调用select都需要在内核遍历传递进来的所有fd；而epoll只需要轮询一次fd集合，同时查看就绪链表中有没有就绪的fd就可以了。

   （3）select支持的文件描述符数量太小了，默认是1024；而epoll没有这个限制，它所支持的fd上限是最大可以打开文件的数目，这个数字一般远大于2048。

2. **epoll为什么高效**：

   （1）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把当前进程往设备等待队列中挂一次，而epoll只要一次拷贝，而且把当前进程往等待队列上挂也只挂一次，这也能节省不少的开销。

2.54 说说多路IO复用技术有哪些，区别是什么?

参考回答

1. **select，poll，epoll**都是IO多路复用的机制，I/O多路复用就是通过一种机制，可以监视多个文件描述符，一旦某个文件描述符就绪（一般是读就绪或者写就绪），能够通知应用程序进行相应的读写操作。
2. **区别**：

   （1）poll与select不同，通过一个pollfd数组向内核传递需要关注的事件，故没有描述符个数的限制，pollfd中的events字段和revents分别用于标示关注的事件和发生的事件，故pollfd数组只需要被初始化一次。

   （2）select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。

   （3）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把当前进程往设备等待队列中挂一次，而epoll只要一次拷贝，而且把当前进程往等待队列上挂也只挂一次，这也能节省不少的开销。

2.55 简述socket中select，epoll的使用场景和区别，epoll水平触发与边缘触发的区别?

参考回答

1. **select，epoll的使用场景**：都是IO多路复用的机制，应用于高并发的网络编程的场景。I/O多路复用就是通过一种机制，可以监视多个文件描述符，一旦某个文件描述符就绪（一般是读就绪或者写就绪），能够通知应用程序进行相应的读写操作。
2. **select，epoll的区别**：

   （1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大；而epoll保证了每个fd在整个过程中只会拷贝一次。

   （2）每次调用select都需要在内核遍历传递进来的所有fd；而epoll只需要轮询一次fd集合，同时查看就绪链表中有没有就绪的fd就可以了。

   （3）select支持的文件描述符数量太小了，默认是1024；而epoll没有这个限制，它所支持的fd上限是最大可以打开文件的数目，这个数字一般远大于2048。
3. **epoll水平触发与边缘触发的区别**

   LT模式（水平触发）下，只要这个fd还有数据可读，每次 epoll_wait都会返回它的事件，提醒用户程序去操作；

   而在ET（边缘触发）模式中，它只会提示一次，直到下次再有数据流入之前都不会再提示了，无论fd中是否还有数据可读。

2.56 说说Reactor、Proactor模式。

参考回答

在高性能的I/O设计中，有两个比较著名的模式Reactor和Proactor模式，其中**Reactor模式用于同步I/O**，而**Proactor运用于异步I/O**操作。

1. **Reactor模式**：Reactor模式应用于同步I/O的场景。Reactor中读操作的具体步骤如下：

   读取操作：

   （1）应用程序注册读就需事件和相关联的事件处理器

   （2）事件分离器等待事件的发生

   （3）当发生读就需事件的时候，事件分离器调用第一步注册的事件处理器

   （4）事件处理器首先执行实际的读取操作，然后根据读取到的内容进行进一步的处理

2. **Proactor模式**：Proactor模式应用于异步I/O的场景。Proactor中读操作的具体步骤如下：

   （1）应用程序初始化一个异步读取操作，然后注册相应的事件处理器，此时事件处理器不关注读取就绪事件，而是关注读取完成事件，这是区别于Reactor的关键。

   （2）事件分离器等待读取操作完成事件

   （3）在事件分离器等待读取操作完成的时候，操作系统调用内核线程完成读取操作，并将读取的内容放入用户传递过来的缓存区中。这也是区别于Reactor的一点，Proactor中，应用程序需要传递缓存区。

   （4）事件分离器捕获到读取完成事件后，激活应用程序注册的事件处理器，事件处理器直接从缓存区读取数据，而不需要进行实际的读取操作。

3. **区别**：从上面可以看出，Reactor中需要**应用程序自己读取或者写入数据**，而Proactor模式中，应用程序不需要用户再自己接收数据，直接使用就可以了，操作系统会将数据从**内核拷贝到用户区。**

答案解析

IO模型的类型。
（1）阻塞IO：调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的检查这个函数有没有返回，必须等这个函数返回后才能进行下一步动作。

（2）非阻塞IO：非阻塞等待，每隔一段时间就去检查IO事件是否就绪。没有就绪就可以做其他事情。

（3）信号驱动IO：Linux用套接口进行信号驱动IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO事件就绪，进程收到SIGIO信号，然后处理IO事件。

（4）IO多路复用：Linux用select/poll函数实现IO复用模型，这两个函数也会使进程阻塞，但是和阻塞IO所不同的是这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检查。知道有数据可读或可写时，才真正调用IO操作函数。

（5）异步IO：Linux中，可以调用aio_read函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。用户可以直接去使用数据。

前四种模型--阻塞IO、非阻塞IO、多路复用IO和信号驱动IO都属于**同步模式**，因为其中真正的IO操作(函数)都将会阻塞进程，只有**异步IO模型**真正实现了IO操作的异步性。

2.57 简述同步与异步的区别，阻塞与非阻塞的区别?

参考回答

1. **同步与异步的区别**：

   同步：是所有的操作都做完，才返回给用户结果。即**写完数据库**之后，**再响应用户**，用户体验不好。

   异步：不用等所有操作都做完，就响应用户请求。即**先响应用户请求**，然后**慢慢去写数据库**，用户体验较好。

2. **阻塞与非阻塞的区别**：

   阻塞：调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的检查这个函数有没有返回，必须等这个函数返回后才能进行下一步动作。

   非阻塞：非阻塞等待，每隔一段时间就去检查IO事件是否就绪。没有就绪就可以做其他事情。

2.58 BIO、NIO有什么区别?

参考回答

BIO（Blocking I/O）：**阻塞IO。**调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的检查这个函数有没有返回，必须等这个函数返回后才能进行下一步动作。

NIO（New I/O）：**同时支持阻塞与非阻塞模式**，NIO的做法是叫一个线程不断的轮询每个IO的状态，看看是否有IO的状态发生了改变，从而进行下一步的操作。

2.59 2.59 请介绍一下5种IO模型

参考回答

1. 阻塞IO：调用者调用了某个函数，等待这个函数返回，期间什么也不做，不停的检查这个函数有没有返回，必须等这个函数返回后才能进行下一步动作。
2. 非阻塞IO：非阻塞等待，每隔一段时间就去检查IO事件是否就绪。没有就绪就可以做其他事情。
3. 信号驱动IO：Linux用套接口进行信号驱动IO，安装一个信号处理函数，进程继续运行并不阻塞，当IO事件就绪，进程收到SIGIO信号，然后处理IO事件。
4. IO多路复用：Linux用select/poll函数实现IO复用模型，这两个函数也会使进程阻塞，但是和阻塞IO所不同的是这两个函数可以同时阻塞多个IO操作。而且可以同时对多个读操作、写操作的IO函数进行检查。知道有数据可读或可写时，才真正调用IO操作函数。
5. 异步IO：Linux中，可以调用aio_read函数告诉内核描述字缓冲区指针和缓冲区的大小、文件偏移及通知的方式，然后立即返回，当内核将数据拷贝到缓冲区后，再通知应用程序。用户可以直接去使用数据。

答案解析

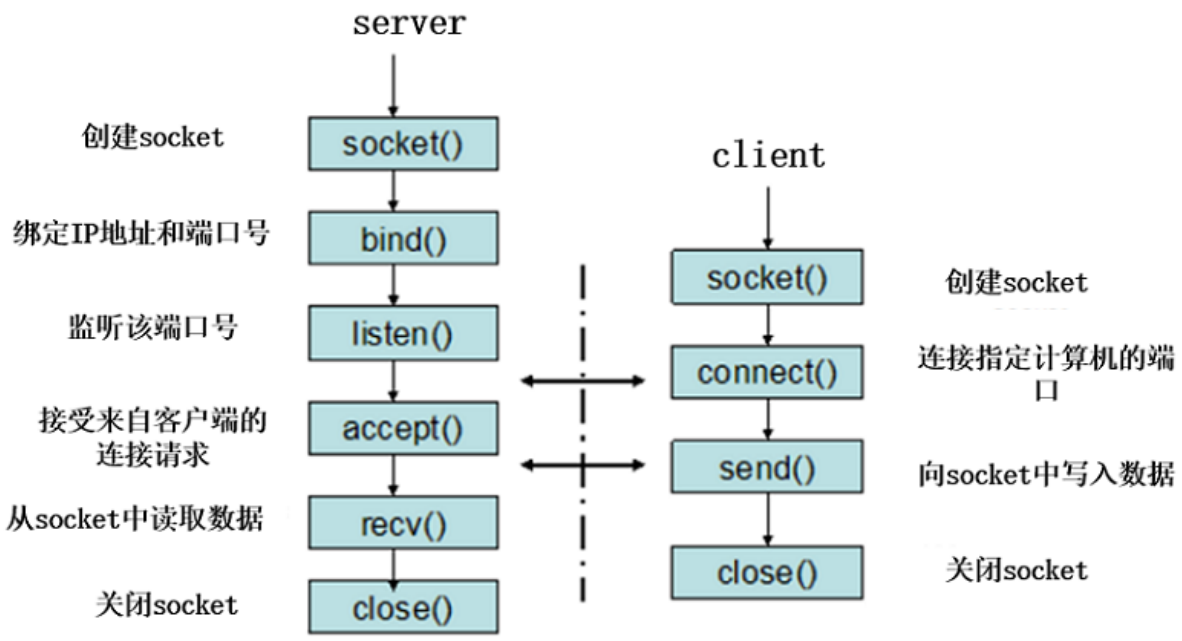
前四种模型--阻塞IO、非阻塞IO、多路复用IO和信号驱动IO都属于**同步模式**，因为其中真正的IO操作(函数)都将会阻塞进程，只有**异步IO模型**真正实现了IO操作的异步性。

异步和同步的区别就在于，异步是内核将数据拷贝到用户区，不需要用户再自己接收数据，直接使用就可以了，而同步是内核通知用户数据到了，然后用户自己调用相应函数去接收数据。

2.60 请说一下socket网络编程中客户端和服务端用到哪些函数?

参考回答

1. **服务器端函数**:
   （1）socket创建一个套接字
   （2）bind绑定ip和port
   （3）listen使套接字变为可以被动链接
   （4）accept等待客户端的链接
   （5）write/read接收发送数据
   （6）close关闭连接
2. **客户端函数**:
   （1）创建一个socket，用函数socket()
   （2）bind绑定ip和port
   （3）连接服务器，用函数connect()
   （4）收发数据，用函数send()和recv()，或read()和write()
   （5）close关闭连接、



2.61 简述网络七层参考模型，每一层的作用?

参考回答

OSI七层模型	功能	对应的网络协议	TCP/IP四层概念模型
应用层	文件传输，文件管理，电子邮件的信息处理	HTTP、TFTP, FTP, NFS, WAIS、SMTP	应用层
表示层	确保一个系统的应用层发送的消息可以被另一个系统的应用层读取，编码转换，数据解析，管理数据的解密和加密。	Telnet, Rlogin, SNMP, Gopher	应用层
会话层	负责在网络中的两节点建立，维持和终止通信。	SMTP, DNS	应用层
传输层	定义一些传输数据的协议和端口。	TCP, UDP	传输层
网络层	控制子网的运行，如逻辑编址，分组传输，路由选择	IP, ICMP, ARP, RARP, AKP, UUCP	网络层
数据链路层	主要是对物理层传输的比特流包装，检测保证数据传输的可靠性，将物理层接收的数据进行MAC（媒体访问控制）地址的封装和解封装	FDDI, Ethernet, Arpanet, PDN, SLIP, PPP，STP。HDLC,SDLC,帧中继	数据链路层
物理层	定义物理设备的标准，主要对物理连接方式，电气特性，机械特性等制定统一标准。	IEEE 802.1A, IEEE 802.2到IEEE 802.	数据链路层

3. C++计算机网络

3.1 简述静态路由和动态路由

参考回答

1. 静态路由是由系统管理员设计与构建的路由表规定的路由。适用于网关数量有限的场合，且网络拓朴结构不经常变化的网络。其缺点是不能动态地适用网络状况的变化，当网络状况变化后必须由网络管理员修改路由表。
2. 动态路由是由路由选择协议而动态构建的，路由协议之间通过交换各自所拥有的路由信息实时更新路由表的内容。动态路由可以自动学习网络的拓朴结构，并更新路由表。其缺点是路由广播更新信息将占据大量的网络带宽。

3.2 说说有哪些路由协议，都是如何更新的

参考回答

1. 路由可分为静态&动态路由。静态路由由管理员手动维护；动态路由由路由协议自动维护。

   路由选择算法的必要步骤：

   1）向其它路由器传递路由信息；

   2）接收其它路由器的路由信息；

   3）根据收到的路由信息计算出到每个目的网络的最优路径，并由此生成路由选择表；

   4）根据网络拓扑的变化及时的做出反应，调整路由生成新的路由选择表，同时把拓扑变化以路由信息的形式向其它路由器宣告。

   两种主要算法：距离向量法（Distance Vector Routing）和链路状态算法（Link-State Routing）。

   由此可分为距离矢量（如：RIP、IGRP、EIGRP）&链路状态路由协议（如：OSPF、IS-IS）。 路由协议是路由器之间实现路由信息共享的一种机制，它允许路由器之间相互交换和维护各 自的路由表。当一台路由器的路由表由于某种原因发生变化时，它需要及时地将这一变化通 知与之相连接的其他路由器，以保证数据的正确传递。路由协议不承担网络上终端用户之间 的数据传输任务。

2. 1）RIP 路由协议：RIP 协议最初是为 Xerox 网络系统的 Xerox parc 通用协议而设计的，是 Internet 中常用的 路由协议。RIP 采用距离向量算法，即路由器根据距离选择路由，所以也称为距离向量协议。 路由器收集所有可到达目的地的不同路径，并且保存有关到达每个目的地的最少站点数的路 径信息，除到达目的地的最佳路径外，任何其它信息均予以丢弃。同时路由器也把所收集的路由信息用 RIP 协议通知相邻的其它路由器。这样，正确的路由信息逐渐扩散到了全网。RIP 使用非常广泛，它简单、可靠，便于配置。但是 RIP 只适用于小型的同构网络，因 为它允许的最大站点数为 15，任何超过 15 个站点的目的地均被标记为不可达。而且 RIP 每 隔 30s 一次的路由信息广播也是造成网络的广播风暴的重要原因之一。

   2）OSPF 路由协议：0SPF 是一种基于链路状态的路由协议，需要每个路由器向其同一管理域的所有其它路 由器发送链路状态广播信息。在 OSPF 的链路状态广播中包括所有接口信息、所有的量度和 其它一些变量。利用 0SPF 的路由器首先必须收集有关的链路状态信息，并根据一定的算法 计算出到每个节点的最短路径。而基于距离向量的路由协议仅向其邻接路由器发送有关路由 更新信息。与 RIP 不同，OSPF 将一个自治域再划分为区，相应地即有两种类型的路由选择方式： 当源和目的地在同一区时，采用区内路由选择；当源和目的地在不同区时，则采用区间路由 选择。这就大大减少了网络开销，并增加了网络的稳定性。当一个区内的路由器出了故障时 并不影响自治域内其它区路由器的正常工作，这也给网络的管理、维护带来方便。

   3）BGP 和 BGP4 路由协议：BGP 是为 TCP / IP 互联网设计的外部网关协议，用于多个自治域之间。它既不是基于纯 粹的链路状态算法，也不是基于纯粹的距离向量算法。它的主要功能是与其它自治域的 BGP 交换网络可达信息。各个自治域可以运行不同的内部网关协议。BGP 更新信息包括网络号 / 自治域路径的成对信息。自治域路径包括到达某个特定网络须经过的自治域串，这些更新信 息通过 TCP 传送出去，以保证传输的可靠性。为了满足 Internet 日益扩大的需要，BGP 还在不断地发展。在最新的 BGP4 中，还可以 将相似路由合并为一条路由。

   4）IGRP 和 EIGRP 协议：EIGRP 和早期的 IGRP 协议都是由 Cisco 发明，是基于距离向量算法的动态路由协议。 EIGRP(Enhanced Interior Gateway Routing Protocol)是增强版的 IGRP 协议。它属于动态内部网 关路由协议，仍然使用矢量 - 距离算法。但它的实现比 IGRP 已经有很大改进，其收敛特性 和操作效率比 IGRP 有显著的提高。它的收敛特性是基于 DUAL (Distributed Update Algorithm) 算法的。DUAL 算法使得路径 在路由计算中根本不可能形成环路。它的收敛时间可以与已存在的其他任何路由协议相匹敌

Enhanced IGRP 与其它路由选择协议之间主要区别包括：收敛宽速（Fast Convergence）、 支持变长子网掩模（Subnet Mask）、局部更新和多网络层协议。执行 Enhanced IGRP 的路由 器存储了所有其相邻路由表，以便于它能快速利用各种选择路径（Alternate Routes）。如果没有合适路径，Enhanced IGRP 查询其邻居以获取所需路径。直到找到合适路径，EnhancedIGRP 查询才会终止，否则一直持续下去。

EIGRP 不作周期性更新。取而代之，当路径度量标准改变时，Enhanced IGRP 只发送局 部更新（Partial Updates）信息。局部更新信息的传输自动受到限制，从而使得只有那些需 要信息的路由器才会更新。基于以上这两种性能，因此 Enhanced IGRP 损耗的带宽比 IGRP 少得多。

3.3 3.3 简述域名解析过程，本机如何干预域名解析

参考回答

1. （1）在浏览器中输入`www.qq.com`域名，操作系统会先检查自己本地的hosts文件是否有这个网址映射关系，如果有，就先调用这个IP地址映射，完成域名解析。

   （2）如果hosts里没有这个域名的映射，则查找本地DNS解析器缓存，是否有这个网址映射关系，如果有，直接返回，完成域名解析。

   （3）如果hosts与本地DNS解析器缓存都没有相应的网址映射关系，首先会找TCP/IP参数中设置的首选DNS服务器，在此我们叫它本地DNS服务器，此服务器收到查询时，如果要查询的域名，包含在本地配置区域资源中，则返回解析结果给客户机，完成域名解析，此解析具有权威性。

   （4）如果要查询的域名，不由本地DNS服务器区域解析，但该服务器已缓存了此网址映射关系，则调用这个IP地址映射，完成域名解析，此解析不具有权威性。

   （5）如果本地DNS服务器本地区域文件与缓存解析都失效，则根据本地DNS服务器的设置（是否设置转发器）进行查询，如果未用转发模式，本地DNS就把请求发至13台根DNS，根DNS服务器收到请求后会判断这个域名(.com)是谁来授权管理，并会返回一个负责该顶级域名服务器的一个IP。本地DNS服务器收到IP信息后，将会联系负责.com域的这台服务器。这台负责.com域的服务器收到请求后，如果自己无法解析，它就会找一个管理.com域的下一级DNS服务器地址(qq.com)给本地DNS服务器。当本地DNS服务器收到这个地址后，就会找qq.com域服务器，重复上面的动作，进行查询，直至找到`www.qq.com`主机。

   （6）如果用的是转发模式，此DNS服务器就会把请求转发至上一级DNS服务器，由上一级服务器进行解析，上一级服务器如果不能解析，或找根DNS或把转请求转至上上级，以此循环。不管是本地DNS服务器用是是转发，还是根提示，最后都是把结果返回给本地DNS服务器，由此DNS服务器再返回给客户机。

   从客户端到本地DNS服务器是属于递归查询，而DNS服务器之间就是的交互查询就是迭代查询。

2. 通过修改本机host来干预域名解析，例如：
   在/etc/hosts文件中添加一句话

   ```
   192.168.188.1 www.baidu.com
   ```

   保存文件后再ping一下www.baidu.com就会连接到192.168.188.1了

   每一行为一条记录，分成两部分，第一部分是IP，第二部分是域名。

   - 一个IP后面可以跟多个域名，可以是几十个甚至上百个
   - 每一行只能有一个IP，也就是说一个域名不能对应多个IP
   - 如果有多行中出现相同的域名（对应的ip不一样），会按最前面的记录来解析

3.4 简述 DNS 查询服务器的基本流程是什么？DNS 劫持是什么？

参考回答

1. 打开浏览器，输入一个域名。比如输入www.163.com，这时，你使用的电脑会发出一个DNS请求到本地DNS服务器。本地DNS服务器一般都是你的网络接入服务器商提供，比如中国电信，中国移动。

   DNS请求到达本地DNS服务器之后，本地DNS服务器会首先查询它的缓存记录，如果缓存中有此条记录，就可以直接返回结果。如果没有，本地DNS服务器还要向DNS根服务器进行查询。

   根DNS服务器没有记录具体的域名和IP地址的对应关系，而是告诉本地DNS服务器，你可以到域服务器上去继续查询，并给出域服务器的地址。

   本地DNS服务器继续向域服务器发出请求，在这个例子中，请求的对象是.com域服务器。.com域服务器收到请求之后，也不会直接返回域名和IP地址的对应关系，而是告诉本地DNS服务器，你的域名的解析服务器的地址。

   最后，本地DNS服务器向域名的解析服务器发出请求，这时就能收到一个域名和IP地址对应关系，本地DNS服务器不仅要把IP地址返回给用户电脑，还要把这个对应关系保存在缓存中，以备下次别的用户查询时，可以直接返回结果，加快网络访问。

2. DNS劫持就是通过劫持了DNS服务器，通过某些手段取得某域名的解析记录控制权，进而修改此域名的解析结果，导致对该域名的访问由原IP地址转入到修改后的指定IP，其结果就是对特定的网址不能访问或访问的是假网址，从而实现窃取资料或者破坏原有正常服务的目的。DNS劫持通过篡改DNS服务器上的数据返回给用户一个错误的查询结果来实现的。

   DNS劫持症状：在某些地区的用户在成功连接宽带后，首次打开任何页面都指向ISP提供的“电信互联星空”、“网通黄页广告”等内容页面。还有就是曾经出现过用户访问Google域名的时候出现了百度的网站。这些都属于DNS劫持。

3.5 简述网关的作用是什么，同一网段的主机如何通信

参考回答

1. 网关即网络中的关卡，我们的互联网是一个一个的局域网、城域网、等连接起来的，在连接点上就是一个一个网络的关卡，即我们的网关，他是保证网络互连的，翻译和转换，使得不同的网络体系能够进行。
2. 网内通信，即通信双方都位处同一网段中，数据传输无需经过路由器(或三层交换机)，即可由本网段自主完成。

   假设发送主机的ARP表中并无目的主机对应的表项，则发送主机会以目的主机IP地址为内容，广播ARP请求以期获知目的主机MAC地址，并通过交换机(除到达端口之外的所有端口发送，即洪泛(Flooding))向全网段主机转发，而只有目的主机接收到此ARP请求后会将自己的MAC地址和IP地址装入ARP应答后将其回复给发送主机，发送主机接收到此ARP应答后，从中提取目的主机的MAC地址，并在其ARP表中建立目的主机的对应表项(IP地址到MAC地址的映射)，之后即可向目的主机发送数据，将待发送数据封装成帧，并通过二层设备(如交换机)转发至本网段内的目的主机，自此完成通信。

3.6 简述CSRF攻击的思想以及解决方法

参考回答

1. CSRF全称叫做，跨站请求伪造。就是黑客可以伪造用户的身份去做一些操作，进而满足自身目的。

   要完成一次CSRF攻击，受害者必须依次完成两个步骤：

   1）登录受信任网站A，并在本地生成Cookie。

   2）在不登出A的情况下，访问危险网站B。

   此时，黑客就可以获取你的cookie达成不可告人的目的了。

2. CSRF 攻击是一种请求伪造的攻击方式，它利用的是服务器不能识别用户的类型从而盗取用户的信息来攻击。因此要防御该种攻击，因为从服务器端着手，增强服务器的识别能力，设计良好的防御机制。主要有以下几种方式：

   1）请求头中的Referer验证（不推荐）

   HTTP的头部有一个`Referer`信息的字段，它记录着该次HTTP请求的来源地址（即它从哪里来的）,既然CSRF攻击是伪造请求是从服务器发送过来的，那么我们就禁止跨域访问，在服务器端增加验证，过滤掉那些不是从本服务器发出的请求，这样可以在一定程度上避免CSRF攻击。 但是这也有缺点，比如如果是从搜索引擎所搜结果调整过来，请求也会被认为是跨域请求。

   2）请求令牌验证（token验证）

   token验证是一种比较广泛使用的防止`CSRF攻击`的手段，当用户通过正常渠道访问服务器时，服务器会生成一个随机的字符串保存在session中，并作为令牌（token）返回给客户端，以隐藏的形式保存在客户端中，客户端每次请求都会带着这个token，服务器根据该token判断该请求是否合法

3.7 说说 MAC地址和IP地址分别有什么作用

参考回答

1. IP地址是IP协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。而MAC地址，指的是物理地址，用来定义网络设备的位置。
2. IP地址的分配是**根据网络的拓扑结构**，而不是根据谁制造了网络设置。若将高效的路由选择方案建立在设备制造商的基础上而不是网络所处的拓朴位置基础上，这种方案是不可行的。
3. 当存在一个附加层的地址寻址时，设备更易于移动和维修。例如，如果一个以太网卡坏了，可以被更换，而无须取得一个新的IP地址。如果一个IP主机从一个网络移到另一个网络，可以给它一个新的IP地址，而无须换一个新的网卡。
4. 无论是局域网，还是广域网中的计算机之间的通信，最终都表现为将数据包从某种形式的链路上的初始节点出发，从一个节点传递到另一个节点，最终传送到目的节点。数据包在这些节点之间的移动都是由**ARP**（Address Resolution Protocol：地址解析协议）负责**将IP地址映射到MAC地址上来完成的。**

3.8 简述 TCP 三次握手和四次挥手的过程

参考回答

3.8.1 三次握手

1）第一次握手：建立连接时，客户端向服务器发送SYN包（seq=x），请求建立连接，等待确认

2）第二次握手：服务端收到客户端的SYN包，回一个ACK包（ACK=x+1）确认收到，同时发送一个SYN包（seq=y）给客户端

3）第三次握手：客户端收到SYN+ACK包，再回一个ACK包（ACK=y+1）告诉服务端已经收到

4）三次握手完成，成功建立连接，开始传输数据

3.8.2 四次挥手1）客户端发送FIN包（FIN=1）给服务端，告诉它自己的数据已经发送完毕，请求终止连接，此时客户端不发送数据，但还能接收数据

2）服务端收到FIN包，回一个ACK包给客户端告诉它已经收到包了，此时还没有断开socket连接，而是等待剩下的数据传输完毕

3）服务端等待数据传输完毕后，向客户端发送FIN包，表明可以断开连接

4）客户端收到后，回一个ACK包表明确认收到，等待一段时间，确保服务端不再有数据发过来，然后彻底断开连接

3.9 说说 TCP 2次握手行不行？为什么要3次

参考回答

1. 为了实现可靠数据传输， TCP 协议的通信双方， 都必须维护一个序列号， 以标识发送出去的数据包中， 哪些是已经被对方收到的。 三次握手的过程即是通信双方相互告知序列号起始值， 并确认对方已经收到了序列号起始值的必经步骤
2. 如果只是两次握手， 至多只有连接发起方的起始序列号能被确认， 另一方选择的序列号则得不到确认

3.10 简述 TCP 和 UDP 的区别，它们的头部结构是什么样的

参考回答

1. TCP协议是有连接的，有连接的意思是开始传输实际数据之前TCP的客户端和服务器端必须通过三次握手建立连接，会话结束之后也要结束连接。而UDP是无连接的

   TCP协议保证数据按序发送，按序到达，提供超时重传来保证可靠性，但是UDP不保证按序到达，甚至不保证到达，只是努力交付，即便是按序发送的序列，也不保证按序送到。

   TCP协议所需资源多，TCP首部需20个字节（不算可选项），UDP首部字段只需8个字节。

   TCP有流量控制和拥塞控制，UDP没有，网络拥堵不会影响发送端的发送速率

   TCP是一对一的连接，而UDP则可以支持一对一，多对多，一对多的通信。

   TCP面向的是字节流的服务，UDP面向的是报文的服务。

2. TCP头部结构如下：

```cpp
/*TCP头定义，共20个字节*/
typedef struct _TCP_HEADER
{
 short m_sSourPort;                    // 源端口号16bit
 short m_sDestPort;                    // 目的端口号16bit
 unsigned int m_uiSequNum;             // 序列号32bit
 unsigned int m_uiAcknowledgeNum;  // 确认号32bit
 short m_sHeaderLenAndFlag;            // 前4位：TCP头长度；中6位：保留；后6位：标志位
 short m_sWindowSize;                  // 窗口大小16bit
 short m_sCheckSum;                    // 检验和16bit
 short m_surgentPointer;                 // 紧急数据偏移量16bit
}__attribute__((packed))TCP_HEADER, *PTCP_HEADER;
```

```cpp
/*TCP头中的选项定义
kind(8bit)+Length(8bit，整个选项的长度，包含前两部分)+内容(如果有的话)
KIND = 1表示 无操作NOP，无后面的部分
  2表示 maximum segment    后面的LENGTH就是maximum segment选项的长度（以byte为单位，1+1+内容部分长度）
  3表示 windows scale      后面的LENGTH就是 windows scale选项的长度（以byte为单位，1+1+内容部分长度）
  4表示 SACK permitted     LENGTH为2，没有内容部分
  5表示这是一个SACK包      LENGTH为2，没有内容部分
  8表示时间戳，LENGTH为10，含8个字节的时间戳
*/

typedef struct _TCP_OPTIONS
{
char m_ckind;
char m_cLength;
char m_cContext[32];
}__attribute__((packed))TCP_OPTIONS, *PTCP_OPTIONS;
```

UDP头部结构如下：

```cpp
/*UDP头定义，共8个字节*/

typedef struct _UDP_HEADER
{
 unsigned short m_usSourPort;          // 源端口号16bit
 unsigned short m_usDestPort;          // 目的端口号16bit
 unsigned short m_usLength;            // 数据包长度16bit
 unsigned short m_usCheckSum;        // 校验和16bit
}__attribute__((packed))UDP_HEADER, *PUDP_HEADER;

```

3.11 简述 TCP 连接 和 关闭的具体步骤

参考回答

1. TCP通过三次握手建立链接

```
1）第一次握手：建立连接时，客户端向服务器发送SYN包（seq=x），请求建立连接，等待确认

2）第二次握手：服务端收到客户端的SYN包，回一个ACK包（ACK=x+1）确认收到，同时发送一个SYN包（seq=y）给客户端

3）第三次握手：客户端收到SYN+ACK包，再回一个ACK包（ACK=y+1）告诉服务端已经收到

4）三次握手完成，成功建立连接，开始传输数据
```

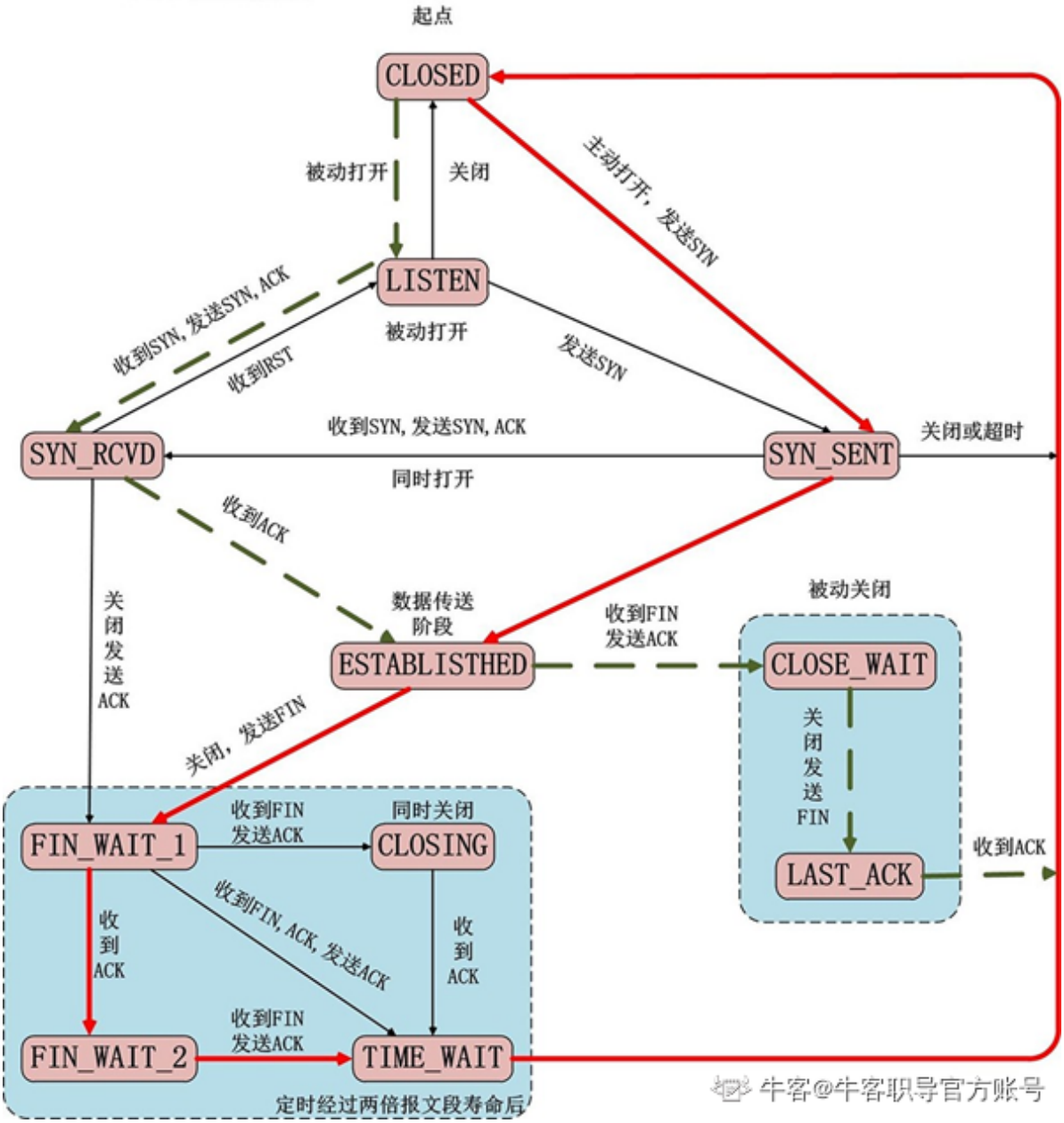
1. 通过4次挥手关闭链接

```
1）客户端发送FIN包（FIN=1）给服务端，告诉它自己的数据已经发送完毕，请求终止连接，此时客户端不发送数据，但还能接收数据

2）服务端收到FIN包，回一个ACK包给客户端告诉它已经收到包了，此时还没有断开socket连接，而是等待剩下的数据传输完毕

3）服务端等待数据传输完毕后，向客户端发送FIN包，表明可以断开连接

4）客户端收到后，回一个ACK包表明确认收到，等待一段时间，确保服务端不再有数据发过来，然后彻底断开连接
```

3.12 简述 TCP 连接 和 关闭的状态转移

参考回答

状态转换如图所示：



上半部分是TCP三路握手过程的状态变迁，下半部分是TCP四次挥手过程的状态变迁。

1. **CLOSED**：起始点，在超时或者连接关闭时候进入此状态，这并不是一个真正的状态，而是这个状态图的假想起点和终点。
2. **LISTEN**：服务器端等待连接的状态。服务器经过 socket，bind，listen 函数之后进入此状态，开始监听客户端发过来的连接请求。此称为应用程序被动打开（等到客户端连接请求）。
3. **SYN_SENT**：第一次握手发生阶段，客户端发起连接。客户端调用 connect，发送 SYN 给服务器端，然后进入 SYN_SENT 状态，等待服务器端确认（三次握手中的第二个报文）。如果服务器端不能连接，则直接进入CLOSED状态。
4. **SYN_RCVD**：第二次握手发生阶段，跟 3 对应，这里是服务器端接收到了客户端的 SYN，此时服务器由 LISTEN 进入 SYN_RCVD状态，同时服务器端回应一个 ACK，然后再发送一个 SYN 即 SYN+ACK 给客户端。状态图中还描绘了这样一种情况，当客户端在发送 SYN 的同时也收到服务器端的 SYN请求，即两个同时发起连接请求，那么客户端就会从 SYN_SENT 转换到 SYN_REVD 状态。
5. **ESTABLISHED**：第三次握手发生阶段，客户端接收到服务器端的 ACK 包（ACK，SYN）之后，也会发送一个 ACK 确认包，客户端进入 ESTABLISHED 状态，表明客户端这边已经准备好，但TCP 需要两端都准备好才可以进行数据传输。服务器端收到客户端的 ACK 之后会从 SYN_RCVD 状态转移到 ESTABLISHED 状态，表明服务器端也准备好进行数据传输了。这样客户端和服务器端都是

ESTABLISHED 状态，就可以进行后面的数据传输了。所以 ESTABLISHED 也可以说是一个数据传送状态。

下面看看TCP四次挥手过程的状态变迁。

1. **FIN_WAIT_1**：第一次挥手。主动关闭的一方（执行主动关闭的一方既可以是客户端，也可以是服务器端，这里以客户端执行主动关闭为例），终止连接时，发送 FIN 给对方，然后等待对方返回 ACK 。调用 close() 第一次挥手就进入此状态。
2. **CLOSE_WAIT**：接收到FIN 之后，被动关闭的一方进入此状态。具体动作是接收到 FIN，同时发送 ACK。之所以叫 CLOSE_WAIT 可以理解为被动关闭的一方此时正在等待上层应用程序发出关闭连接指令。TCP关闭是全双工过程，这里客户端执行了主动关闭，被动方服务器端接收到FIN 后也需要调用 close 关闭，这个 CLOSE_WAIT 就是处于这个状态，等待发送 FIN，发送了FIN 则进入 LAST_ACK 状态。
3. **FIN_WAIT_2**：主动端（这里是客户端）先执行主动关闭发送FIN，然后接收到被动方返回的 ACK 后进入此状态。
4. **LAST_ACK**：被动方（服务器端）发起关闭请求，由状态2 进入此状态，具体动作是发送 FIN给对方，同时在接收到ACK 时进入CLOSED状态。
5. **CLOSING**：两边同时发起关闭请求时（即主动方发送FIN，等待被动方返回ACK，同时被动方也发送了FIN，主动方接收到了FIN之后，发送ACK给被动方），主动方会由FIN_WAIT_1 进入此状态，等待被动方返回ACK。
6. **TIME_WAIT**：从状态变迁图会看到，四次挥手操作最后都会经过这样一个状态然后进入CLOSED状态。

答案解析

状态	描述
CLOSED	阻塞或关闭状态，表示主机当前没有正在传输或者建立的链接
LISTEN	监听状态，表示服务器做好准备，等待建立传输链接
SYN RECV	收到第一次的传输请求，还未进行确认
SYN SENT	发送完第一个SYN报文，等待收到确认
ESTABLISHED	链接正常建立之后进入数据传输阶段
FIN WAIT1	主动发送第一个FIN报文之后进入该状态
FIN WAIT2	已经收到第一个FIN的确认信号，等待对方发送关闭请求
TIMED WAIT	完成双向链接关闭，等待分组消失
CLOSING	双方同时关闭请求，等待对方确认时
CLOSE WAIT	收到对方的关闭请求并进行确认进入该状态
LAST ACK	等待最后一次确认关闭的报文

3.13 简述 TCP 慢启动

参考回答

1. **慢启动**（Slow Start），是传输控制协议（TCP）使用的一种阻塞控制机制。慢启动也叫做指数增长期。慢启动是指每次TCP接收窗口收到确认时都会增长。增加的大小就是已确认段的数目。这种情况一直保持到要么没有收到一些段，要么窗口大小到达预先定义的阈值。如果发生丢失事件，

TCP就认为这是网络阻塞，就会采取措施减轻网络拥挤。一旦发生丢失事件或者到达阈值，TCP就会进入线性增长阶段。这时，每经过一个RTT窗口增长一个段。

3.14 说说 TCP 如何保证有序

参考回答

- 主机每次发送数据时，TCP就给每个数据包分配一个序列号并且在一个特定的时间内等待接收主机对分配的这个序列号进行确认，如果发送主机在一个特定时间内没有收到接收主机的确认，则发送主机会重传此数据包。接收主机利用序列号对接收的数据进行确认，以便检测对方发送的数据是否有丢失或者乱序等，接收主机一旦收到已经顺序化的数据，它就将这些数据按正确的顺序重组成数据流并传递到高层进行处理。
- 具体步骤如下：

  （1）为了保证数据包的可靠传递，发送方必须把已发送的数据包保留在缓冲区；

  （2）并为每个已发送的数据包启动一个超时定时器；

  （3）如在定时器超时之前收到了对方发来的应答信息（可能是对本包的应答，也可以是对本包后续包的应答），则释放该数据包占用的缓冲区；

  （4）否则，重传该数据包，直到收到应答或重传次数超过规定的最大次数为止。

  （5）接收方收到数据包后，先进行CRC校验，如果正确则把数据交给上层协议，然后给发送方发送一个累计应答包，表明该数据已收到，如果接收方正好也有数据要发给发送方，应答包也可方在数据包中捎带过去。

3.15 说说 TCP 常见的拥塞控制算法有哪些

参考回答

1. TCP Tahoe/Reno

   最初的实现，包括慢启动、拥塞避免两个部分。基于重传超时（retransmission timeout/RTO）和重复确认为条件判断是否发生了丢包。两者的区别在于：Tahoe算法下如果收到三次重复确认，就进入快重传立即重发丢失的数据包，同时将慢启动阈值设置为当前拥塞窗口的一半，将拥塞窗口设置为1MSS，进入慢启动状态；而Reno算法如果收到三次重复确认，就进入快重传，但不进入慢启动状态，而是直接将拥塞窗口减半，进入拥塞控制阶段，这称为“快恢复”。

   而Tahoe和Reno算法在出现RTO时的措施一致，都是将拥塞窗口降为1个MSS，然后进入慢启动阶段。

2. TCP BBR（Bottleneck Bandwidth and Round-trip propagation time）

   BBR是由Google设计，于2016年发布的拥塞算法。以往大部分拥塞算法是基于丢包来作为降低传输速率的信号，而BBR则基于模型主动探测。该算法使用网络最近出站数据分组当时的最大带宽和往返时间来建立网络的显式模型。数据包传输的每个累积或选择性确认用于生成记录在数据包传输过程和确认返回期间的时间内所传送数据量的采样率。该算法认为随着网络接口控制器逐渐进入千兆速度时，分组丢失不应该被认为是识别拥塞的主要决定因素，所以基于模型的拥塞控制算法能有更高的吞吐量和更低的延迟，可以用BBR来替代其他流行的拥塞算法，例如CUBIC。

3.16 简述 TCP 超时重传

参考回答

TCP可靠性中最重要的一个机制是处理数据超时和重传。TCP协议要求在发送端每发送一个报文段，就启动一个定时器并等待确认信息；接收端成功接收新数据后返回确认信息。若在定时器超时前数据未能被确认，TCP就认为报文段中的数据已丢失或损坏，需要对报文段中的数据重新组织和重传。

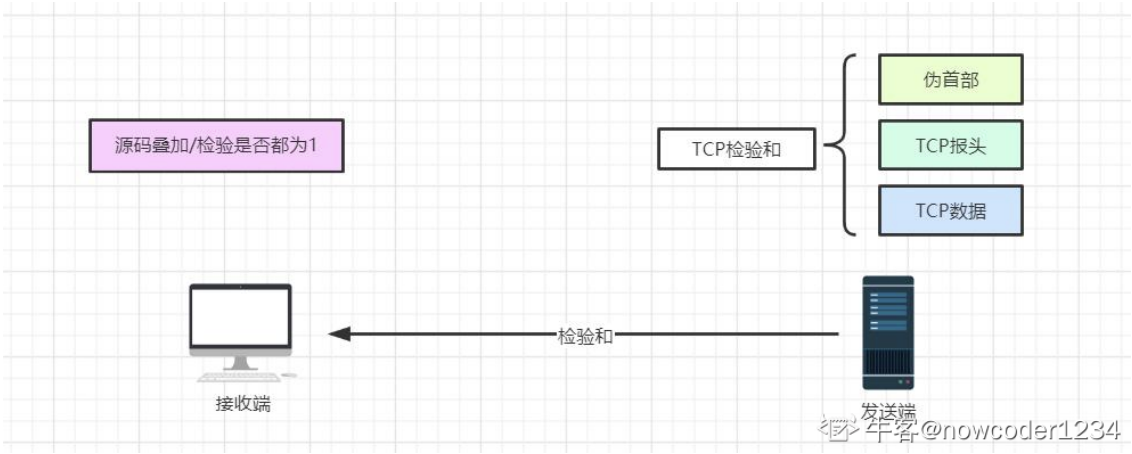
3.17 说说 TCP 可靠性保证

参考回答

TCP主要提供了检验和、序列号/确认应答、超时重传、最大消息长度、滑动窗口控制等方法实现了可靠性传输。

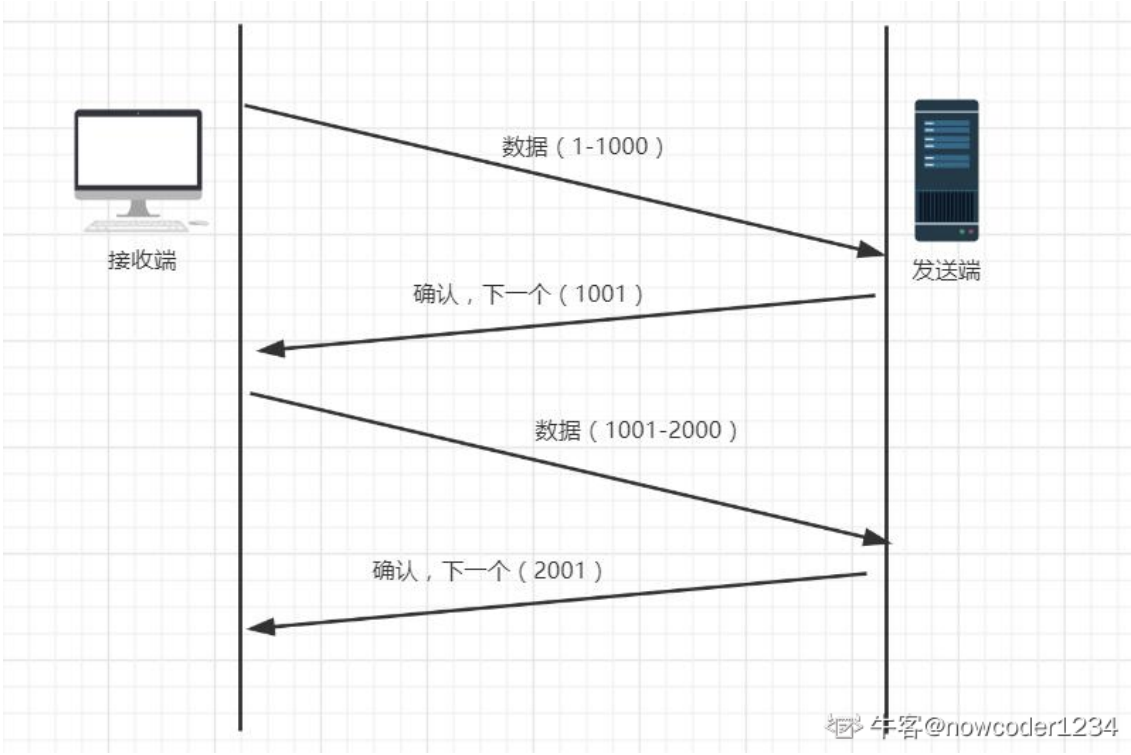
3.17.1 检验和

- 通过检验和的方式，接收端可以检测出来数据是否有差错和异常，假如有差错就会直接丢弃TCP段，重新发送。TCP在计算检验和时，会在TCP首部加上一个12字节的伪首部。检验和总共计算3部分：TCP首部、TCP数据、TCP伪首部



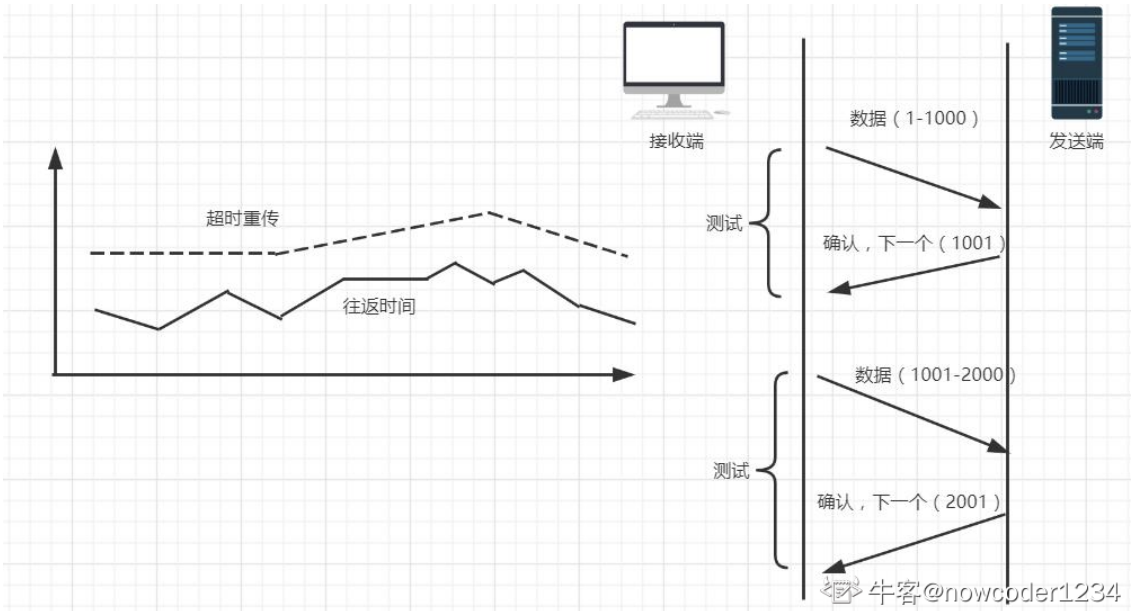
3.17.2 序列号/确认应答

- 这个机制类似于问答的形式。比如在课堂上老师会问你“明白了吗？”，假如你没有隔一段时间没有回应或者你说不明白，那么老师就会重新讲一遍。其实计算机的确认应答机制也是一样的，发送端发送信息给接收端，接收端会回应一个包，这个包就是应答包。
- 上述过程中，只要发送端有一个包传输，接收端没有回应确认包（ACK包），都会重发。或者接收端的应答包，发送端没有收到也会重发数据。这就可以保证数据的完整性。



3.17.3 超时重传

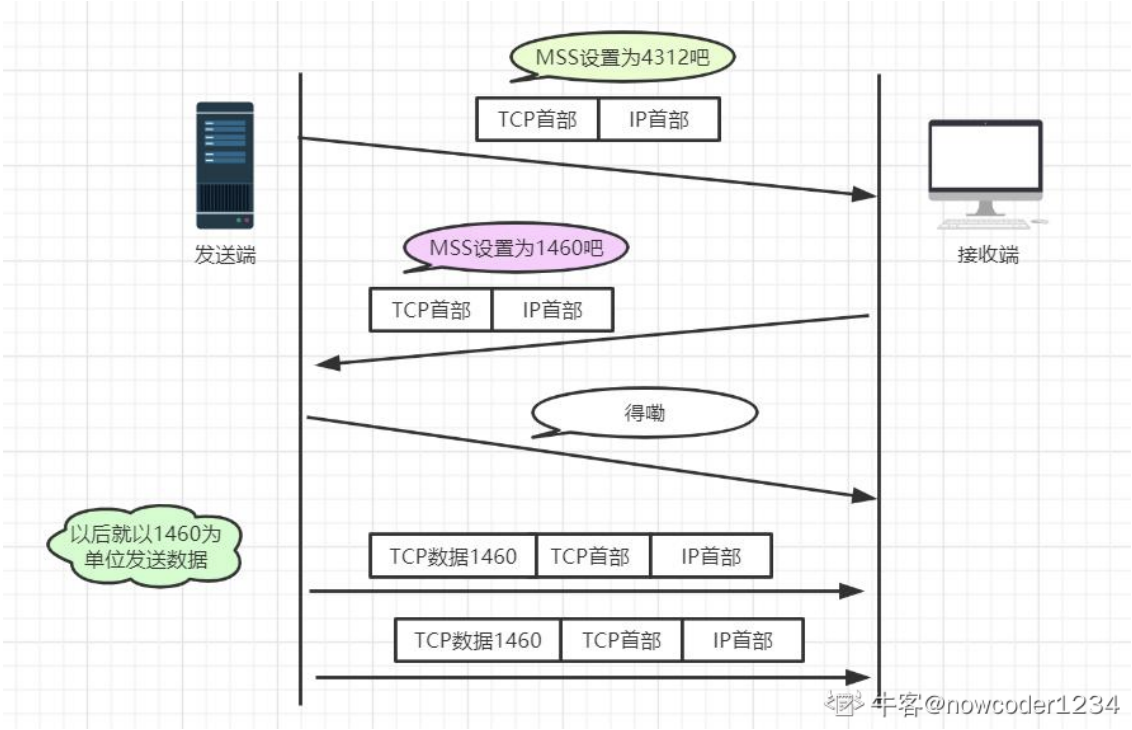
- 超时重传是指发送出去的数据包到接收到确认包之间的时间，如果超过了这个时间会被认为是丢包了，需要重传。那么我们该如何确认这个时间值呢?
- 我们知道，一来一回的时间总是差不多的，都会有一个类似于平均值的概念。比如发送一个包到接收端收到这个包一共是0.5s，然后接收端回发一个确认包给发送端也要0.5s，这样的两个时间就是RTT（往返时间）。然后可能由于网络原因的问题，时间会有偏差，称为抖动（方差）。
- 从上面的介绍来看，超时重传的时间大概是比往返时间+抖动值还要稍大的时间。



- 但是在重发的过程中，假如一个包经过多次的重发也没有收到对端的确认包，那么就会认为接收端异常，强制关闭连接。并且通知应用通信异常强行终止。

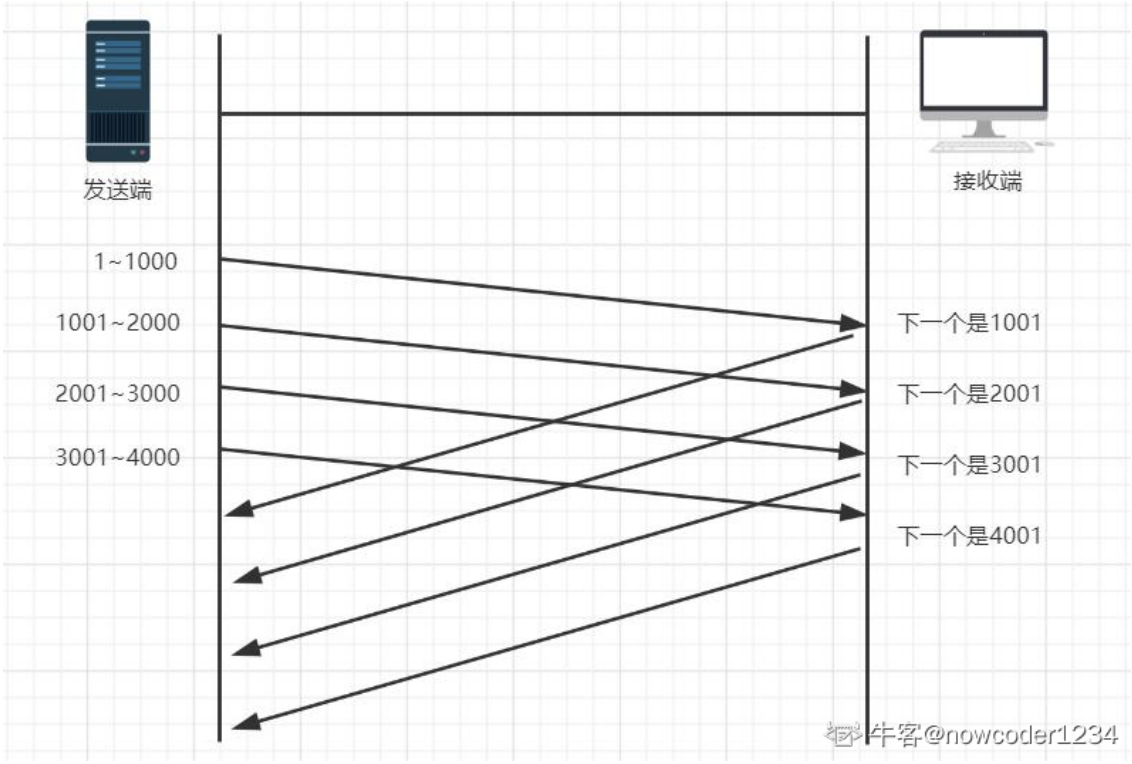
3.17.4 最大消息长度

- 在建立TCP连接的时候，双方约定一个最大的长度（MSS）作为发送的单位，重传的时候也是以这个单位来进行重传。理想的情况下是该长度的数据刚好不被网络层分块。



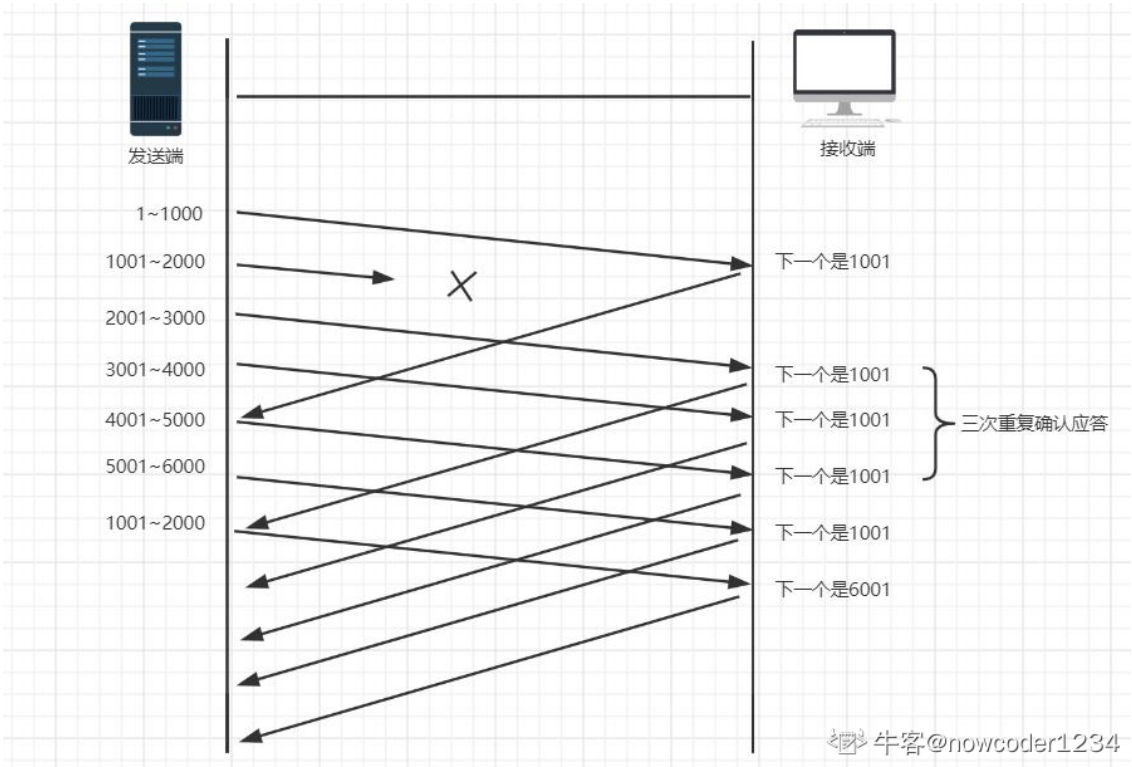
3.17.5 滑动窗口控制

- 我们上面提到的超时重传的机制存在效率低下的问题，发送一个包到发送下一个包要经过一段时间才可以。所以我们就想着能不能不用等待确认包就发送下一个数据包呢？这就提出了一个滑动窗口的概念。



- 窗口的大小就是在无需等待确认包的情况下，发送端还能发送的最大数据量。这个机制的实现就是使用了大量的缓冲区，通过对多个段进行确认应答的功能。通过下一次的确认包可以判断接收端是否已经接收到了数据，如果已经接收了就从缓冲区里面删除数据。

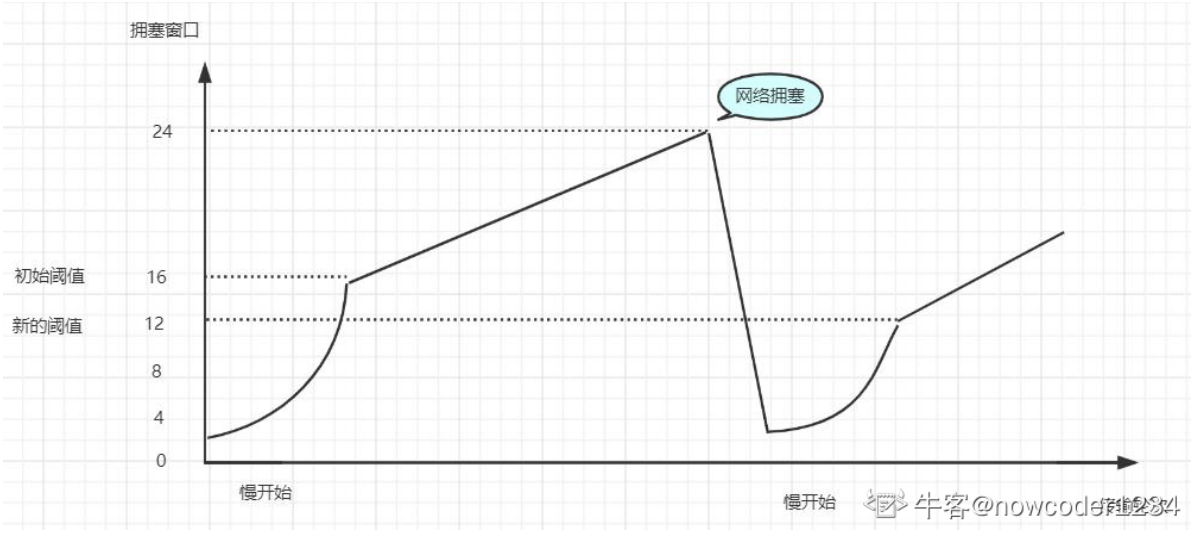
- 在窗口之外的数据就是还未发送的和对端已经收到的数据。那么发送端是怎么样判断接收端有没有接收到数据呢？或者怎么知道需要重发的数据有哪些呢？通过下面这个图就知道了。



- 如上图，接收端在没有收到自己所期望的序列号数据之前，会对之前的数据进行重复确认。发送端在收到某个应答包之后，又连续3次收到同样的应答包，则数据已经丢失了，需要重发。

3.17.6 拥塞控制

- 窗口控制解决了 两台主机之间因传送速率而可能引起的丢包问题，在一方面保证了TCP数据传送的可靠性。然而如果网络非常拥堵，此时再发送数据就会加重网络负担，那么发送的数据段很可能超过了最大生存时间也没有到达接收方，就会产生丢包问题。为此TCP引入慢启动机制，先发出少量数据，就像探路一样，先摸清当前的网络拥堵状态后，再决定按照多大的速度传送数据。
- 发送开始时定义拥塞窗口大小为1；每次收到一个ACK应答，拥塞窗口加1；而在每次发送数据时，发送窗口取拥塞窗口与接送段接收窗口最小者。
- 慢启动：在启动初期以指数增长方式增长；设置一个慢启动的阈值，当以指数增长达到阈值时就停止指数增长，按照线性增长方式增加至拥塞窗口；线性增长达到网络拥塞时立即把拥塞窗口置回1，进行新一轮的“慢启动”，同时新一轮的阈值变为原来的一半。



3.18 简述 TCP 滑动窗口以及重传机制

参考回答

1. 滑动窗口协议是传输层进行流控的一种措施，接收方通过通告发送方自己的窗口大小，从而控制发送方的发送速度，从而达到防止发送方发送速度过快而导致自己被淹没的目的。

   TCP的滑动窗口解决了端到端的流量控制问题，允许接受方对传输进行限制，直到它拥有足够的缓冲空间来容纳更多的数据。

2. TCP在发送数据时会设置一个计时器，若到计时器超时仍未收到数据确认信息，则会引发相应的超时或基于计时器的重传操作，计时器超时称为**重传超时（RTO）**。另一种方式的重传称为快速重传，通常发生在没有延时的情况下。若TCP累积确认无法返回新的ACK，或者当ACK包含的选择确认信息（SACK）表明出现失序报文时，快速重传会推断出现丢包，需要重传。

3.19 说说滑动窗口过小怎么办

参考回答

1. 我们可以假设窗口的大小是1，也是就每次只能发送一个数据，并且发送方只有接受方对这个数据进行确认了以后才能发送下一个数据。如果说窗口过小，那么当传输比较大的数据的时候需要不停的对数据进行确认，这个时候就会造成很大的延迟。

3.20 说说如果三次握手时候每次握手信息对方没收到会怎么样，分情况介绍

参考回答

1. 如果第一次握手消息丢失，那么请求方不会得到ack消息，超时后进行重传
2. 如果第二次握手消息丢失，那么请求方不会得到ack消息，超时后进行重传
3. 如果第三次握手消息丢失，那么Server 端该TCP连接的状态为SYN_RECV,并且会根据 TCP的超时重传机制，会等待3秒、6秒、12秒后重新发送SYN+ACK包，以便Client重新发送ACK包。而Server重发SYN+ACK包的次数，可以设置/proc/sys/net/ipv4/tcp_synack_retries修改，默认值为5.如果重发指定次数之后，仍然未收到 client 的ACK应答，那么一段时间后，Server自动关闭这个连接。

   client 一般是通过 connect() 函数来连接服务器的，而connect()是在 TCP的三次握手的第二次握手完成后就成功返回值。也就是说 client 在接收到 SYN+ACK包，它的TCP连接状态就为 established（已连接），表示该连接已经建立。那么如果 第三次握手中的ACK包丢失的情况下，Client 向server端发送数据，Server端将以 RST包响应，方能感知到Server的错误。

3.21 简述 TCP 的 TIME_WAIT，为什么需要有这个状态

参考回答

1. TIME_WAIT状态也成为2MSL等待状态。每个具体TCP实现必须选择一个报文段最大生存时间MSL（Maximum Segment Lifetime），它是任何报文段被丢弃前在网络内的最长时间。这个时间是有限的，因为TCP报文段以IP数据报在网络内传输，而IP数据报则有限制其生存时间的TTL字段。

   对一个具体实现所给定的MSL值，处理的原则是：当TCP执行一个主动关闭，并发回最后一个ACK，该连接必须在TIME_WAIT状态停留的时间为2倍的MSL。这样可让TCP再次发送最后的ACK以防这个ACK丢失（另一端超时并重发最后的FIN）。

这种2MSL等待的另一个结果是这个TCP连接在2MSL等待期间，定义这个连接的插口（客户的IP地址和端口号，服务器的IP地址和端口号）不能再被使用。这个连接只能在2MSL结束后才能再被使用。

2. 理论上，四个报文都发送完毕，就可以直接进入CLOSE状态了，但是可能网络是不可靠的，有可能最后一个ACK丢失。所以TIME_WAIT状态就是用来重发可能丢失的ACK报文。

3.22 简述什么是 MSL，为什么客户端连接要等待2MSL的时间才能完全关闭

参考回答

1. MSL是Maximum Segment Lifetime的英文缩写，可译为“最长报文段寿命”，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。
2. 为了保证客户端发送的最后一个ACK报文段能够到达服务器。因为这个ACK有可能丢失，从而导致处在LAST-ACK状态的服务器收不到对FIN-ACK的确认报文。服务器会超时重传这个FIN-ACK，接着客户端再重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待2MSL，而是在发送完ACK之后直接释放关闭，一但这个ACK丢失的话，服务器就无法正常的进入关闭连接状态。

- 两个理由：
    - 保证客户端发送的最后一个ACK报文段能够到达服务端。

      这个ACK报文段有可能丢失，使得处于LAST-ACK状态的B收不到对已发送的FIN+ACK报文段的确认，服务端超时重传FIN+ACK报文段，而客户端能在2MSL时间内收到这个重传的FIN+ACK报文段，接着客户端重传一次确认，重新启动2MSL计时器，最后客户端和服务端都进入到CLOSED状态，若客户端在TIME-WAIT状态不等待一段时间，而是发送完ACK报文段后立即释放连接，则无法收到服务端重传的FIN+ACK报文段，所以不会再发送一次确认报文段，则服务端无法正常进入到CLOSED状态。
    - 防止“已失效的连接请求报文段”出现在本连接中。

      客户端在发送完最后一个ACK报文段后，再经过2MSL，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失，使下一个新的连接中不会出现这种旧的连接请求报文段。

3.23 说说什么是 SYN flood，如何防止这类攻击?

参考回答

1. SYN Flood是当前最流行的DoS（拒绝服务攻击）与DDoS(分布式拒绝服务攻击)的方式之一，这是一种利用TCP协议缺陷，发送大量伪造的TCP连接请求，使被攻击方资源耗尽（CPU满负荷或内存不足)的攻击方式.
2. 有以下三种方法预防或响应网络上的DDoS攻击：

   (1)从互联网服务提供商(ISP)购买服务。

   许多互联网服务提供商(ISP)提供DDoS缓解服务，但是当企业网络受到攻击时，企业需要向互联网服务提供商(ISP)报告事件以开始缓解。这种策略称为“清洁管道”，在互联网服务提供商(ISP)收取服务费用时很受欢迎，但在缓解措施开始之前，通常会导致30到60分钟的网络延迟。

   (2)保留在内部并自己解决。

企业可以使用入侵防御系统/防火墙技术和专用于防御DDoS攻击的专用硬件来实现内部预防和响应DDoS攻击。不幸的是，受影响的流量已经在网络上消耗了宝贵的带宽。这使得该方法最适合在托管设施中配备设备的企业，在这些企业中，流量是通过交叉连接到达互联网服务提供商(ISP)，从而保护流向企业其他部门的下游带宽。

(3)使用内容分发网络(CDN)。

由于IT团队可以将基础设施置于内容分发网络(CDN)后面，因此这种方法可以最大程度地减少对企业网络基础设施的攻击。这些网络庞大而多样，如果组织订阅DNS和DDoS缓解措施，则它们可以保护电子商务站点以及企业本身。

3.24 说说什么是 TCP 粘包和拆包?

参考回答

1. TCP是个"流"协议，所谓流，就是没有界限的一串数据。大家可以想想河里的流水，是连成一片的，其间并没有分界线。TCP底层并不了解上层业务数据的具体含义，它会根据TCP缓冲区的实际情况进行包的划分，所以在业务上认为，一个完整的包可能会被TCP拆分成多个包进行发送，也有可能把多个小的包封装成一个大的数据包发送，这就是所谓的TCP粘包和拆包问题。

答案解析

假设客户端分别发送了两个数据包D1和D2给服务端，由于服务端一次读取到的字节数是不确定的，故可能存在以下4种情况。

（1）服务端分两次读取到了两个独立的数据包，分别是D1和D2，没有粘包和拆包；

（2）服务端一次接收到了两个数据包，D1和D2粘合在一起，被称为TCP粘包；

（3）服务端分两次读取到了两个数据包，第一次读取到了完整的D1包和D2包的部分内容，第二次读取到了D2包的剩余内容，这被称为TCP拆包；

（4）服务端分两次读取到了两个数据包，第一次读取到了D1包的部分内容D1_1，第二次读取到了D1包的剩余内容D1_2和D2包的整包。

如果此时服务端TCP接收滑窗非常小，而数据包D1和D2比较大，很有可能会发生第五种可能，即服务端分多次才能将D1和D2包接收完全，期间发生多次拆包。

3.25 说说 TCP 与 UDP 在网络协议中的哪一层，他们之间有什么区别?

参考回答

TCP和UDP协议都是**传输层**协议。二者的区别主要有：

1. 基于连接vs无连接

- TCP是面向连接的协议。
- UDP是无连接的协议。UDP更加适合消息的多播发布，从单个点向多个点传输消息。

1. 可靠性

- TCP提供交付保证，传输过程中丢失，将会重发。
- UDP是不可靠的，不提供任何交付保证。（网游和视频的丢包情况）

1. 有序性

- TCP保证了消息的有序性，即使到达客户端顺序不同，TCP也会排序。
- UDP不提供有序性保证。

1. 数据边界

- TCP不保存数据边界。
    - 虽然TCP也将在收集所有字节之后生成一个完整的消息，但是这些信息在传给传输给接受端之前将储存在TCP缓冲区，以确保更好的使用网络带宽。
- UDP保证。
    - 在UDP中，数据包单独发送的，只有当他们到达时，才会再次集成。包有明确的界限来哪些包已经收到，这意味着在消息发送后，在接收器接口将会有一个读操作，来生成一个完整的消息。

1. 速度

- TCP速度慢
- UDP速度快。应用在在线视频媒体，电视广播和多人在线游戏。

1. 发送消耗

- TCP是重量级。
- UDP是轻量级。
    - 因为UDP传输的信息中不承担任何间接创造连接，保证交货或秩序的的信息。
    - 这也反映在用于报头大小。

1. 报头大小

- TCP头大。
    - 一个TCP数据包报头的大小是20字节。
    - TCP报头中包含序列号，ACK号，数据偏移量，保留，控制位，窗口，紧急指针，可选项，填充项，校验位，源端口和目的端口。
- UDP头小。
    - UDP数据报报头是8个字节。
    - 而UDP报头只包含长度，源端口号，目的端口，和校验和。

1. 拥塞或流控制

- TCP有流量控制。
    - 在任何用户数据可以被发送之前，TCP需要三数据包来设置一个套接字连接。TCP处理的可靠性和拥塞控制。
- UDP不能进行流量控制。

1. 应用

- 由于TCP提供可靠交付和有序性的保证，它是最适合需要高可靠并且对传输时间要求不高的应用。
- UDP是更适合的应用程序需要快速，高效的传输的应用，如游戏。
- UDP是无状态的性质，在服务器端需要对大量客户端产生的少量请求进行应答的应用中是非常有用的。
- 在实践中，TCP被用于金融领域，如FIX协议是一种基于TCP的协议，而UDP是大量使用在游戏和娱乐场所。

10.上层使用的协议

- 基于TCP协议的：Telnet，FTP以及SMTP协议。
- 基于UDP协议的：DHCP、DNS、SNMP、TFTP、BOOTP。

3.26 说说从系统层面上，UDP 如何保证尽量可靠?

参考回答

1. UDP仅提供了最基本的数据传输功能，至于传输时连接的建立和断开、传输可靠性的保证这些UDP统统不关心，而是把这些问题抛给了UDP上层的应用层程序去处理，自己仅提供传输层协议的最基本功能。
2. 最简单的方式是在应用层模仿传输层TCP的可靠性传输。下面不考虑拥塞处理，可靠UDP的简单设计。
    - 添加seq/ack机制，确保数据发送到对端
    - 添加发送和接收缓冲区，主要是用户超时重传。
    - 添加超时重传机制。

3.27 说一说 TCP 的 keepalive，以及和 HTTP 的 keepalive 的区别?

参考回答

1. **HTTP Keep-Alive**

   在http早期，每个http请求都要求打开一个tpc socket连接，并且使用一次之后就断开这个tcp连接。使用keep-alive可以改善这种状态，即在一次TCP连接中可以持续发送多份数据而不会断开连接。通过使用keep-alive机制，可以减少tcp连接建立次数，也意味着可以减少TIME_WAIT状态连接，以此提高性能和提高httpd服务器的吞吐率(更少的tcp连接意味着更少的系统内核调用,socket的accept()和close()调用)。但是，keep-alive并不是免费的午餐,长时间的tcp连接容易导致系统资源无效占用。配置不当的keep-alive，有时比重复利用连接带来的损失还更大。所以，正确地设置keep-alive timeout时间非常重要。

2. **TCP KEEPALIVE**

   链接建立之后，如果应用程序或者上层协议一直不发送数据，或者隔很长时间才发送一次数据，当链接很久没有数据报文传输时如何去确定对方还在线，到底是掉线了还是确实没有数据传输，链接还需不需要保持，这种情况在TCP协议设计中是需要考虑到的。TCP协议通过一种巧妙的方式去解决这个问题，当超过一段时间之后，TCP自动发送一个数据为空的报文给对方，如果对方回应了这个报文，说明对方还在线，链接可以继续保持，如果对方没有报文返回，并且重试了多次之后则认为链接丢失，没有必要保持链接。

3. TCP的keepalive机制和HTTP的keep-alive机制是说的完全不同的两个东西，tcp的keepalive是在ESTABLISH状态的时候，双方如何检测连接的可用行。而http的keep-alive说的是如何避免进行重复的TCP三次握手和四次挥手的环节。

3.28 简述 TCP 协议的延迟 ACK 和累计应答

参考回答

1. 延迟应答指的是：TCP在接收到对端的报文后，并不会立即发送ack，而是等待一段时间发送ack，以便将ack和要发送的数据一块发送。当然ack不能无限延长，否则对端会认为包超时而造成报文重传。linux采用动态调节算法来确定延时的时间。
2. 累计应答指的是：为了保证**顺序性**，每一个包都有一个**ID**（序号），在建立连接的时候，会商定起始的ID是多少，然后按照ID一个个发送。而为了保证不丢包，对应发送的包都要进行应答，但不是一个个应答，而是会**应答某个之前的ID**，该模式称为**累计应答**

3.29 说说 TCP 如何加速一个大文件的传输

参考回答

1. 建连优化：TCP 在建立连接时，如果丢包，会进入重试，重试时间是 1s、2s、4s、8s 的指数递增间隔，缩短定时器可以让 TCP 在丢包环境建连时间更快，非常适用于高并发短连接的业务场景。
2. 平滑发包：在 RTT 内均匀发包，规避微分时间内的流量突发，尽量避免瞬间拥塞
3. 丢包预判：有些网络的丢包是有规律性的，例如每隔一段时间出现一次丢包，例如每次丢包都连续丢几个等，如果程序能自动发现这个规律（有些不明显），就可以针对性提前多发数据，减少重传时间、提高有效发包率。
4. RTO 探测：若始终收不到 ACK 报文，则需要触发 RTO 定时器。RTO 定时器一般都时间非常长，会浪费很多等待时间，而且一旦 RTO，CWND 就会骤降（标准 TCP），因此利用 Probe 提前与 RTO 去试探，可以规避由于 ACK 报文丢失而导致的速度下降问题。
5. 带宽评估：通过单位时间内收到的 ACK 或 SACK 信息可以得知客户端有效接收速率，通过这个速率可以更合理的控制发包速度。
6. 带宽争抢：有些场景（例如合租）是大家互相挤占带宽的，假如你和室友各 1Mbps 的速度看电影，会把 2Mbps 出口占满，而如果一共有 3 个人看，则每人只能分到 1/3。若此时你的流量流量达到 2Mbps，而他俩还都是 1Mbps，则你至少仍可以分到 2/(2+1+1) * 2Mbps = 1Mbps 的 50% 的带宽，甚至更多，代价就是服务器侧的出口流量加大，增加成本。（TCP 优化的本质就是用带宽换用户体验感）

3.30 服务器怎么判断客户端断开了连接

参考回答

1. 检测连接是否丢失的方法大致有两种：**keepalive**和**heart-beat**
2. （tcp内部机制）采用keepalive，它会先要求此连接一定时间没有活动（一般是几个小时），然后发出数据段，经过多次尝试后（每次尝试之间也有时间间隔），如果仍没有响应，则判断连接中断。可想而知，整个**周期需要很长**的时间。
3. （应用层实现）一个简单的heart-beat实现一般测试连接是否中断采用的时间间隔都比较短，可以**很快的决定连接是否中断**。并且，由于是在应用层实现，因为可以自行决定当判断连接中断后应该采取的行为，而keepalive在判断连接失败后只会将连接丢弃。

3.31 说说端到端，点到点的区别

参考回答

1. 端到端通信是针对传输层来说的，传输层为网络中的主机提供端到端的通信。因为无论tcp还是udp协议，都要负责把上层交付的数据从发送端传输到接收端，不论其中间跨越多少节点。只不过tcp比较可靠而udp不可靠而已。所以称之为端到端，也就是从发送端到接收端。

   它是一个网络连接，指的是在数据传输之前，在发送端与接收端之间（忽略中间有多少设备）为数据的传输建立一条链路，链路建立以后，发送端就可以发送数据，知道数据发送完毕，接收端确认接收成功。 也就是说在数据传输之前，先为数据的传输开辟一条通道，然后在进行传输。从发送端发出数据到接收端接收完毕，结束。

   端到端通信建立在点到点通信的基础之上，它是由一段段的点到点通信信道构成的，是比点到点通信更高一级的通信方式，完成应用程序(进程)之间的通信。

   端到端的优点：

链路建立之后，发送端知道接收端一定能收到，而且经过中间交换设备时不需要进行存储转发，因此传输延迟小。

端到端传输的缺点：

（1）直到接收端收到数据为止，发送端的设备一直要参与传输。如果整个传输的延迟很长，那么对发送端的设备造成很大的浪费。

（2）如果接收设备关机或故障，那么端到端传输不可能实现。

2. 点到点通信是针对数据链路层或网络层来说的，因为数据链路层只负责直接相连的两个节点之间的通信，一个节点的数据链路层接受ip层数据并封装之后，就把数据帧从链路上发送到与其相邻的下一个节点。 点对点是基于MAC地址和或者IP地址，是指一个设备发数据给与该这边直接连接的其他设备，这台设备又在合适的时候将数据传递给与它相连的下一个设备，通过一台一台直接相连的设备把数据传递到接收端。

   直接相连的节点对等实体的通信叫点到点通信。它只提供一台机器到另一台机器之间的通信，不会涉及到程序或进程的概念。同时点到点通信并不能保证数据传输的可靠性，也不能说明源主机与目的主机之间是哪两个进程在通信。

   由物理层、数据链路层和网络层组成的通信子网为网络环境中的主机提供点到点的服务

   点到点的优点：

   （1）发送端设备送出数据后，它的任务已经完成，不需要参与整个传输过程，这样不会浪费发送端设备的资源。

   （2）即使接收端设备关机或故障，点到点传输也可以采用存储转发技术进行缓冲。

   点到点的缺点：

   点到点传输的缺点是发送端发出数据后，不知道接收端能否收到或何时能收到数据。

   在一个网络系统的不同分层中，可能用到端到端传输，也可能用到点到点传输。如Internet网，IP及以下各层采用点到点传输，4层以上采用端到端传输。

3.32 说说浏览器从输入 URL 到展现页面的全过程

参考回答

- 1、输入地址
- 2、浏览器查找域名的 IP 地址
- 3、浏览器向 web 服务器发送一个 HTTP 请求
- 4、服务器的永久重定向响应
- 6、服务器处理请求
- 7、服务器返回一个 HTTP 响应
- 8、浏览器显示 HTML
- 9、浏览器发送请求获取嵌入在 HTML 中的资源（如图片、音频、视频、CSS、JS等等）

3.33 简述 HTTP 和 HTTPS 的区别?

参考回答

1. HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

   HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

2. HTTP与HTTPS的区别

- https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。
- http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
- http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

3.34 说说 HTTP 中的 referer 头的作用

参考回答

1. HTTP Referer是header的一部分，当浏览器向web服务器发送请求的时候，一般会带上Referer，告诉服务器该网页是从哪个页面链接过来的，服务器因此可以获得一些信息用于处理。
2. 防盗链。假如在www.google.com里有一个`www.baidu.com`链接，那么点击进入这个`www.baidu.com`，它的header信息里就有：Referer= http://www.google.com

   只允许我本身的网站访问本身的图片服务器，假如域是`www.google.com`，那么图片服务器每次取到Referer来判断一下域名是不是`www.google.com`，如果是就继续访问，不是就拦截。

   将这个http请求发给服务器后，如果服务器要求必须是某个地址或者某几个地址才能访问，而你发送的referer不符合他的要求，就会拦截或者跳转到他要求的地址，然后再通过这个地址进行访问。
3. 防止恶意请求

   比如静态请求是`*.html`结尾的，动态请求是`*.shtml`，那么由此可以这么用，所有的`*.shtml`请求，必须Referer为我自己的网站。
4. 空Referer

   定义：Referer头部的内容为空，或者，一个HTTP请求中根本不包含Referer头部（一个请求并不是由链接触发产生的）

   直接在浏览器的地址栏中输入一个资源的URL地址，那么这种请求是不会包含Referer字段的，因为这是一个“凭空产生”的HTTP请求，并不是从一个地方链接过去的。

   那么在防盗链设置中，允许空Referer和不允许空Referer有什么区别?

   允许Referer为空，意味着你允许比如浏览器直接访问。
5. 防御CSRF

   比对HTTP 请求的来源地址，如果Referer中的地址是安全可信任的地址，那么就放行

3.35 说说 HTTP 的方法有哪些

参考回答

- GET： 用于请求访问已经被URI（统一资源标识符）识别的资源，可以通过URL传参给服务器
- POST：用于传输信息给服务器，主要功能与GET方法类似，但一般推荐使用POST方式。
- PUT： 传输文件，报文主体中包含文件内容，保存到对应URI位置。
- HEAD： 获得报文首部，与GET方法类似，只是不返回报文主体，一般用于验证URI是否有效。
- DELETE：删除文件，与PUT方法相反，删除对应URI位置的文件。
- OPTIONS：查询相应URI支持的HTTP方法。

3.36 简述 HTTP 1.0，1.1，2.0 的主要区别

参考回答

http/1.0 :

1. 默认不支持长连接，需要设置keep-alive参数指定
2. 强缓存expired、协商缓存last-modified\if-modified-since 有一定的缺陷

http 1.1 :

1. 默认长连接(keep-alive)，http请求可以复用Tcp连接，但是同一时间只能对应一个http请求(http请求在一个Tcp中是串行的)
2. 增加了强缓存cache-control、协商缓存etag\if-none-match 是对http/1 缓存的优化

http/2.0 :

1. 多路复用，一个Tcp中多个http请求是并行的 (雪碧图、多域名散列等优化手段http/2中将变得多余)
2. 二进制格式编码传输
3. 使用HPACK算法做header压缩
4. 服务端推送

3.37 说说 HTTP 常见的响应状态码及其含义

参考回答

- **200** : 从状态码发出的请求被服务器正常处理。
- **204** : 服务器接收的请求已成功处理，但在返回的响应报文中不含实体的主体部分【即没有内容】。
- **206** : 部分的内容（如：客户端进行了范围请求，但是服务器成功执行了这部分的干请求）。
- **301** : 跳转，代表永久性重定向（请求的资源已被分配了新的URI，以后已使用资源，现在设置了URI）。
- **302** : 临时性重定向（请求的资源已经分配了新的URI，希望用户本次能够使用新的URI来进行访问）。
- **303** : 由于请求对应的资源存在的另一个URI（因使用get方法，定向获取请求的资源）。
- **304** : 客户端发送附带条件的请求时，服务器端允许请求访问资源，但因发生请求未满足条件的情况后，直接返回了 304。
- **307** : 临时重定向【该状态码与302有着相同的含义】。
- **400** : 请求报文中存在语法错误（当错误方式时，需修改请求的内容后，再次发送请求）。
- **401** : 发送的请求需要有通过HTTP认证的认证信息。
- **403** : 对请求资源的访问被服务器拒绝了。
- **404** : 服务器上无法找到请求的资源。
- **500** : 服务器端在执行请求时发生了错误。
- **503** : 服务器暂时处于超负载或者是正在进行停机维护，现在无法处理请求。

答案解析

- 1XX : 信息类状态码（表示接收请求状态处理）
- 2XX : 成功状态码（表示请求正常处理完毕）
- 3XX : 重定向（表示需要进行附加操作，已完成请求）
- 4XX : 客户端错误（表示服务器无法处理请求）
- 5XX : 服务器错误状态码（表示服务器处理请求的时候出错）

3.38 说说 GET请求和 POST 请求的区别

参考回答

1. GET请求在URL中传送的参数是有长度限制的，而POST没有。
2. GET比POST更不安全，因为参数直接暴露在URL上，所以不能用来传递敏感信息。
3. GET参数通过URL传递，POST放在Request body中。
4. GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。
5. GET请求只能进行url编码，而POST支持多种编码方式。
6. GET请求会被浏览器主动cache，而POST不会，除非手动设置。
7. GET产生的URL地址可以被Bookmark，而POST不可以。
8. GET在浏览器回退时是无害的，而POST会再次提交请求。

3.39 说说 Cookie 和 Session 的关系和区别是什么

参考回答

1. Cookie与Session都是会话的一种方式。它们的典型使用场景比如“购物车”，当你点击下单按钮时，服务端并不清楚具体用户的具体操作，为了标识并跟踪该用户，了解购物车中有几样物品，服务端通过为该用户创建Cookie/Session来获取这些信息。
2. cookie数据存放在客户的浏览器上，session数据放在服务器上。
3. cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗 考虑到安全应当使用session。
4. session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能 考虑到减轻服务器性能方面，应当使用COOKIE。
5. 单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

3.40 简述 HTTPS 的加密与认证过程

参考回答

1. 客户端在浏览器中输入一个https网址，然后连接到server的443端口 采用https协议的server必须有一套数字证书（一套公钥和密钥） 首先server将证书（公钥）传送到客户端 客户端解析证书，验证成功，则生成一个随机数（私钥），并用证书将该随机数加密后传回server server用密钥解密后，获得这个随机值，然后将要传输的信息和私钥通过某种算法混合在一起（加密）传到客户端 客户端用之前的生成的随机数（私钥）解密服务器端传来的信息

2. 首先浏览器会从内置的证书列表中索引，找到服务器下发证书对应的机构，如果没有找到，此时就会提示用户该证书是不是由权威机构颁发，是不可信任的。如果查到了对应的机构，则取出该机构颁发的公钥。

   用机构的证书公钥解密得到证书的内容和证书签名，内容包括网站的网址、网站的公钥、证书的有效期等。浏览器会先验证证书签名的合法性。签名通过后，浏览器验证证书记录的网址是否和当前网址是一致的，不一致会提示用户。如果网址一致会检查证书有效期，证书过期了也会提示用户。这些都通过认证时，浏览器就可以安全使用证书中的网站公钥了。

4. C++设计模式

4.1 说说什么是单例设计模式，如何实现

参考回答

1. 单例模式定义

   保证一个类仅有一个实例，并提供一个访问它的全局访问点，该实例被所有程序模块共享。

   那么我们就必须保证：

   （1）该类不能被复制。

   （2）该类不能被公开的创造。

   那么对于C++来说，它的构造函数，拷贝构造函数和赋值函数都不能被公开调用。

2. 单例模式**实现方式**

   单例模式通常有两种模式，分别为**懒汉式单例**和**饿汉式单例**。两种模式实现方式分别如下：

   （1）懒汉式设计模式实现方式（2种）

   a. 静态指针 + 用到时初始化

   b. 局部静态变量

   （2）饿汉式设计模式（2种）

   a. 直接定义静态对象

   b. 静态指针 + 类外初始化时new空间实现

答案解析

1. 懒汉模式

   懒汉模式的特点是延迟加载，比如配置文件，采用懒汉式的方法，配置文件的实例直到用到的时候才会加载，不到万不得已就不会去实例化类，也就是说在第一次用到类实例的时候才会去实例化。以下是懒汉模式实现方式C++代码：

   （1）**懒汉模式实现一：静态指针 + 用到时初始化**

```
//代码实例（线程不安全）
template<typename T>
class Singleton
{
public:
static T& getInstance()
{
    if (!value_)
    {
        value_ = new T();
    }
    return *value_;
}
private:
    Singleton();
    ~Singleton();
    static T* value_;
};
template<typename T>
T* Singleton<T>::value_ = NULL;
```

在单线程中，这样的写法是可以正确使用的，但是在多线程中就不行了，该方法是线程不安全的。
a. 假如线程A和线程B, 这两个线程要访问getInstance函数，线程A进入getInstance函数，并检测if条件，由于是第一次进入，value为空，if条件成立，准备创建对象实例。
b. 但是，线程A有可能被OS的调度器中断而挂起睡眠，而将控制权交给线程B。
c. 线程B同样来到if条件，发现value还是为NULL，因为线程A还没来得及构造它就已经被中断了。此时假设线程B完成了对象的创建，并顺利的返回。
d. 之后线程A被唤醒，继续执行new再次创建对象，这样一来，两个线程就构建两个对象实例，这就破坏了唯一性。
另外，还存在内存泄漏的问题，new出来的东西始终没有释放，下面是一种饿汉式的一种改进。

```
//代码实例（线程安全）
emplate<typename T>
class Singleton
{
public:
static T& getInstance()
{
    if (!value_)
    {
        value_ = new T();
    }
    return *value_;
}
private:
     class CGarbo
    {
    public:
        ~CGarbo()
        {
            if(Singleton::value_)
                delete Singleton::value_;
        }
    };
    static CGarbo Garbo;
    Singleton();
    ~Singleton();
    static T* value_;
};
template<typename T>
T* Singleton<T>::value_ = NULL;
```

在程序运行结束时，系统会调用Singleton的静态成员Garbo的析构函数，该析构函数会删除单例的唯一实例。使用这种方法释放单例对象有以下特征：

a. 在单例类内部定义专有的嵌套类；

```
b. 在单例类内定义私有的专门用于释放的静态成员；
c. 利用程序在结束时析构全局变量的特性，选择最终的释放时机。
```

（2）懒汉模式实现二：局部静态变量

```
//代码实例（线程不安全）
template<typename T>
class Singleton
{
```

```
public:
static T& getInstance()
{
    static T instance;
    return instance;
}

private:
    Singleton(){};
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
};
```

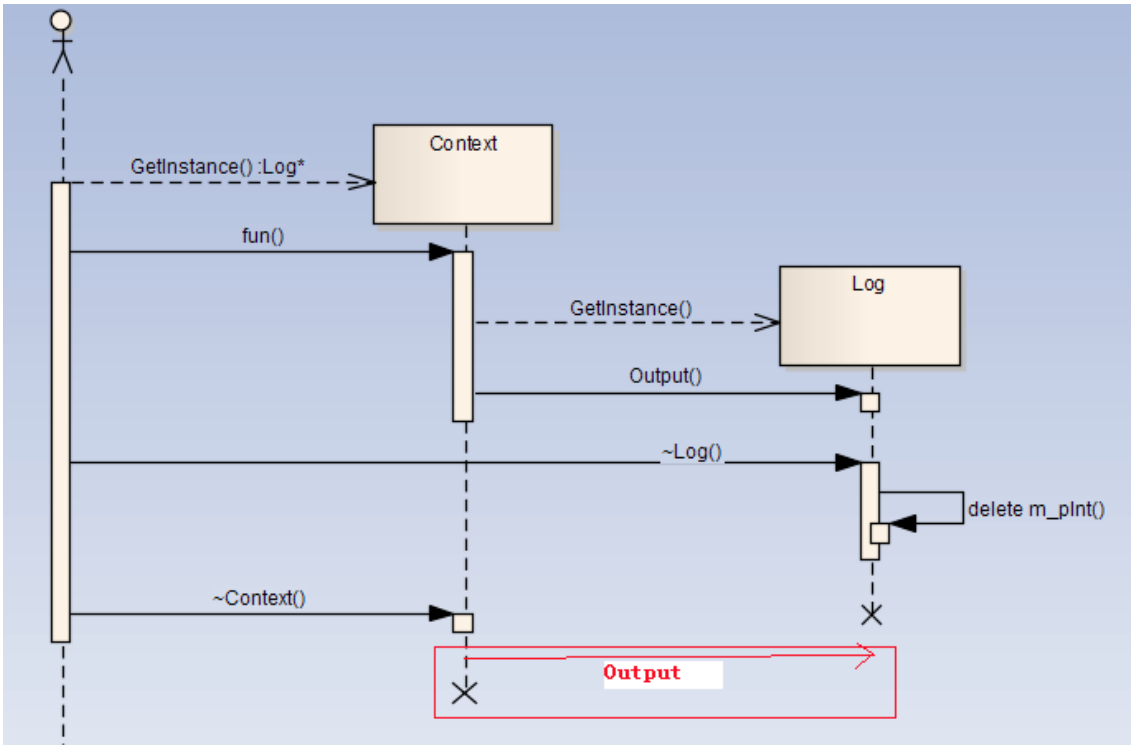
同样，静态局部变量的实现方式也是线程不安全的。如果存在多个单例对象的析构顺序有依赖时，可能会出现程序崩溃的危险。
对于局部静态对象的也是一样的。因为 static T instance；语句不是一个原子操作，在第一次被调用时会调用Singleton的构造函数，而如果构造函数里如果有多条初始化语句，则初始化动作可以分解为多步操作，就存在多线程竞争的问题。
为什么存在多个单例对象的析构顺序有依赖时，可能会出现程序崩溃的危险?

原因：由于静态成员是在第一次调用函数GetInstance时进行初始化，调用构造函数的，因此构造函数的调用顺序时可以唯一确定了。对于析构函数，我们只知道其调用顺序和构造函数的调用顺序相反，但是如果几个Singleton类的析构函数之间也有依赖关系，而且出现类似单例实例A的析构函数中使用了单例实例B，但是程序析构时是先调用实例B的析构函数，此时在A析构函数中使用B时就可能会崩溃。

```
//代码实例（线程安全）
#include <string>
#include <iostream>
using namespace std;
class Log
{
public:
    static Log* GetInstance()
    {
        static Log oLog;
        return &oLog;
    }

    void Output(string strLog)
    {
        cout<<strLog<<(*m_pInt)<<endl;
    }
private:
    Log():m_pInt(new int(3))
    {
    }
    ~Log()
    {cout<<"~Log"<<endl;
        delete m_pInt;
        m_pInt = NULL;
    }
    int* m_pInt;
};

class Context
```

```
{
public:
    static Context* GetInstance()
    {
        static Context oContext;
        return &oContext;
    }
    ~Context()
    {
        Log::GetInstance()->Output(__FUNCTION__);
    }

    void fun()
    {
        Log::GetInstance()->Output(__FUNCTION__);
    }
private:
    Context(){}
    Context(const Context& context);
};

int main(int argc, char* argv[])
{
    Context::GetInstance()->fun();
    return 0;
}
```

在这个反例中有两个Singleton: Log和Context，Context的fun和析构函数会调用Log来输出一些信息，结果程序Crash掉了，该程序的运行的序列图如下（其中画红框的部分是出问题的部分）：



解决方案：对于析构的顺序，我们可以用一个容器来管理它，根据单例之间的依赖关系释放实例，对所有的实例的析构顺序进行排序，之后调用各个单例实例的析构方法，如果出现了循环依赖关系，就给出异常，并输出循环依赖环。

2. 饿汉模式

单例类定义的时候就进行实例化。因为main函数执行之前，全局作用域的类成员静态变量m_Instance已经初始化，故**没有多线程的问题。**

（1）饿汉模式实现一：直接定义静态对象

```
//代码实例（线程安全）
//.h文件
class Singleton
{
public:
  static Singleton& GetInstance();
private:
  Singleton(){}
  Singleton(const Singleton&);
  Singleton& operator= (const Singleton&);
private:
  static Singleton m_Instance;
};
//CPP文件
Singleton Singleton::m_Instance;//类外定义-不要忘记写
Singleton& Singleton::GetInstance()
{
   return m_Instance;
}
//函数调用
Singleton& instance = Singleton::GetInstance();
```

优点：

实现简单，多线程安全。

缺点：

a. 如果存在多个单例对象且这几个单例对象相互依赖，可能会出现程序崩溃的危险。原因:对编译器来说，静态成员变量的初始化顺序和析构顺序是一个未定义的行为;具体分析在懒汉模式中也讲到了。

b. 在程序开始时，就创建类的实例，如果Singleton对象产生很昂贵，而本身有很少使用，这种方式单从资源利用效率的角度来讲，比懒汉式单例类稍差些。但从反应时间角度来讲，则比懒汉式单例类稍好些。

使用条件：

a. 当肯定不会有构造和析构依赖关系的情况。

b. 想避免频繁加锁时的性能消耗

（2）饿汉模式实现二：静态指针 + 类外初始化时new空间实现

```
//代码实例（线程安全）
class Singleton
{
protected:
    Singleton(){}
private:
    static Singleton* p;
public:
    static Singleton* initance();
};
Singleton* Singleton::p = new Singleton;
Singleton* singleton::initance()
```

```
{
    return p;
}
```

4.2 简述一下单例设计模式的懒汉式和饿汉式，如何保证线程安全

参考回答

1. 懒汉式设计模式

   懒汉模式的特点是延迟加载，比如配置文件，采用懒汉式的方法，配置文件的实例直到用到的时候才会加载，不到万不得已就不会去实例化类，也就是说在第一次用到类实例的时候才会去实例化。

2. 饿汉模式

   单例类定义的时候就进行实例化。因为main函数执行之前，全局作用域的类成员静态变量m_Instance已经初始化，故**没有多线程的问题**。

答案解析

懒汉设计模式两种实现方式线程不安全问题的解决：

（1）懒汉模式实现一：静态指针 + 用到时初始化

```
//代码实例（线程不安全）
template<typename T>
class Singleton
{
public:
static T& getInstance()
{
    if (!value_)
    {
        value_ = new T();
    }
    return *value_;
}
private:
    Singleton();
    ~Singleton();
    static T* value_;
};
template<typename T>
T* Singleton<T>::value_ = NULL;
```

在单线程中，这样的写法是可以正确使用的，但是在多线程中就不行了，该方法是**线程不安全**的。
a. 假如线程A和线程B, 这两个线程要访问getInstance函数，线程A进入getInstance函数，并检测if条件，由于是第一次进入，value为空，if条件成立，准备创建对象实例。
b. 但是，线程A有可能被OS的调度器中断而挂起睡眠，而将控制权交给线程B。
c. 线程B同样来到if条件，发现value还是为NULL，因为线程A还没来得及构造它就已经被中断了。此时假设线程B完成了对象的创建，并顺利的返回。
d. 之后线程A被唤醒，继续执行new再次创建对象，这样一来，两个线程就构建两个对象实例，这就破坏了唯一性。
另外，还存在内存泄漏的问题，new出来的东西始终没有释放，下面是一种饿汉式的一种线程安全的改进。

```
//代码实例（线程安全）
emplate<typename T>
class Singleton
{
public:
static T& getInstance()
{
    if (!value_)
    {
        value_ = new T();
    }
    return *value_;
}
private:
     class CGarbo
    {
    public:
        ~CGarbo()
        {
            if(Singleton::value_)
                delete Singleton::value_;
        }
    };
    static CGarbo Garbo;
    Singleton();
    ~Singleton();
    static T* value_;
};
template<typename T>
T* Singleton<T>::value_ = NULL;
```

在程序运行结束时，系统会调用Singleton的静态成员Garbo的析构函数，该析构函数会删除单例的唯一实例。使用这种方法释放单例对象有以下特征：

a. 在单例类内部定义专有的嵌套类；
b. 在单例类内定义私有的专门用于释放的静态成员；
c. 利用程序在结束时析构全局变量的特性，选择最终的释放时机。

（2）懒汉模式实现二：局部静态变量

```
//代码实例（线程不安全）
template<typename T>
class Singleton
{
public:
static T& getInstance()
{
    static T instance;
    return instance;
}

private:
    Singleton(){};
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
};
```

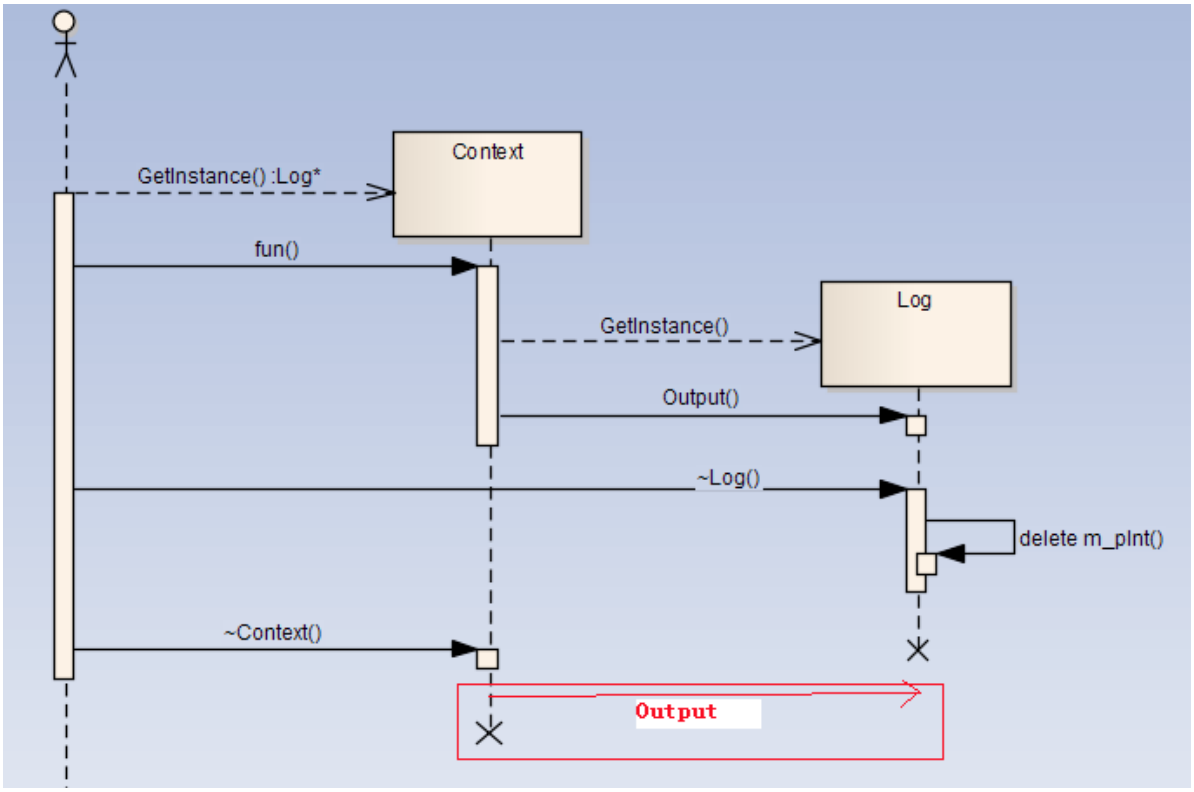
同样，静态局部变量的实现方式也是**线程不安全**的。如果存在多个单例对象的析构顺序有依赖时，可能会出现程序崩溃的危险。
对于局部静态对象的也是一样的。因为 static T instance；语句不是一个原子操作，在第一次被调用时会调用Singleton的构造函数，而如果构造函数里如果有多条初始化语句，则初始化动作可以分解为多步操作，就存在多线程竞争的问题。
为什么存在多个单例对象的析构顺序有依赖时，可能会出现程序崩溃的危险?

原因：由于静态成员是在第一次调用函数GetInstance时进行初始化，调用构造函数的，因此构造函数的调用顺序时可以唯一确定了。对于析构函数，我们只知道其调用顺序和构造函数的调用顺序相反，但是如果几个Singleton类的析构函数之间也有依赖关系，而且出现类似单例实例A的析构函数中使用了单例实例B，但是程序析构时是先调用实例B的析构函数，此时在A析构函数中使用B时就可能会崩溃。

```
//代码实例（线程安全）
#include <string>
#include <iostream>
using namespace std;
class Log
{
public:
    static Log* GetInstance()
    {
        static Log oLog;
        return &oLog;
    }

    void Output(string strLog)
    {
        cout<<strLog<<(*m_pInt)<<endl;
    }
private:
    Log():m_pInt(new int(3))
    {
    }
    ~Log()
    {cout<<"~Log"<<endl;
        delete m_pInt;
        m_pInt = NULL;
    }
    int* m_pInt;
};

class Context
{
public:
    static Context* GetInstance()
    {
        static Context oContext;
        return &oContext;
    }
    ~Context()
    {
        Log::GetInstance()->Output(__FUNCTION__);
    }

    void fun()
    {
        Log::GetInstance()->Output(__FUNCTION__);
```

```
    }
private:
    Context(){}
    Context(const Context& context);
};

int main(int argc, char* argv[])
{
    Context::GetInstance()->fun();
    return 0;
}
```

在这个反例中有两个Singleton: Log和Context，Context的fun和析构函数会调用Log来输出一些信息，结果程序Crash掉了，该程序的运行的序列图如下（其中画红框的部分是出问题的部分）：



解决方案：对于析构的顺序，我们可以用一个容器来管理它，根据单例之间的依赖关系释放实例，对所有的实例的析构顺序进行排序，之后调用各个单例实例的析构方法，如果出现了循环依赖关系，就给出异常，并输出循环依赖环。

4.3 请说说工厂设计模式，如何实现，以及它的优点

参考回答

1. 工厂设计模式的定义

   定义一个创建对象的接口，让子类决定实例化哪个类，而对象的创建统一交由工厂去生产，有良好的封装性，既做到了解耦，也保证了最少知识原则。

2. 工厂设计模式分类

   工厂模式属于创建型模式，大致可以分为三类，**简单工厂模式、工厂方法模式、抽象工厂模式。**听上去差不多，都是工厂模式。下面一个个介绍：

   （1）简单工厂模式

它的主要特点是需要在工厂类中做判断，从而创造相应的产品。当增加新的产品时，就需要修改工厂类。

举例：有一家生产处理器核的厂家，它只有一个工厂，能够生产两种型号的处理器核。客户需要什么样的处理器核，一定要显示地告诉生产工厂。下面给出一种实现方案：

```
//程序实例（简单工厂模式）
enum CTYPE {COREA, COREB};
class SingleCore
{
public:
    virtual void Show() = 0;
};
//单核A
class SingleCoreA: public SingleCore
{
public:
    void Show() { cout<<"SingleCore A"<<endl; }
};
//单核B
class SingleCoreB: public SingleCore
{
public:
    void Show() { cout<<"SingleCore B"<<endl; }
};
//唯一的工厂，可以生产两种型号的处理器核，在内部判断
class Factory
{
public:
    SingleCore* CreateSingleCore(enum CTYPE ctype)
    {
        if(ctype == COREA) //工厂内部判断
            return new SingleCoreA(); //生产核A
        else if(ctype == COREB)
            return new SingleCoreB(); //生产核B
        else
            return NULL;
    }
};
```

优点： 简单工厂模式可以根据需求，动态生成使用者所需类的对象，而使用者不用去知道怎么创建对象，使得各个模块各司其职，降低了系统的耦合性。

缺点：就是要增加新的核类型时，就需要修改工厂类。这就违反了开放封闭原则：软件实体（类、模块、函数）可以扩展，但是不可修改。

（2）工厂方法模式

所谓工厂方法模式，是指定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。

举例：这家生产处理器核的产家赚了不少钱，于是决定再开设一个工厂专门用来生产B型号的单核，而原来的工厂专门用来生产A型号的单核。这时，客户要做的是找好工厂，比如要A型号的核，就找A工厂要；否则找B工厂要，不再需要告诉工厂具体要什么型号的处理器核了。下面给出一个实现方案：

```
//程序实例（工厂方法模式）
class SingleCore
```

```
{
public:
    virtual void Show() = 0;
};
//单核A
class SingleCoreA: public SingleCore
{
public:
    void Show() { cout<<"SingleCore A"<<endl; }
};
//单核B
class SingleCoreB: public SingleCore
{
public:
    void Show() { cout<<"SingleCore B"<<endl; }
};
class Factory
{
public:
    virtual SingleCore* CreateSingleCore() = 0;
};
//生产A核的工厂
class FactoryA: public Factory
{
public:
    SingleCoreA* CreateSingleCore() { return new SingleCoreA; }
};
//生产B核的工厂
class FactoryB: public Factory
{
public:
    SingleCoreB* CreateSingleCore() { return new SingleCoreB; }
};
```

优点： **扩展性好，符合了开闭原则**，新增一种产品时，只需增加改对应的产品类和对应的工厂子类即可。

缺点：每增加一种产品，就需要增加一个对象的工厂。如果这家公司发展迅速，推出了很多新的处理器核，那么就要开设相应的新工厂。在C++实现中，就是要定义一个个的工厂类。显然，相比简单工厂模式，工厂方法模式需要更多的类定义。

（3）抽象工厂模式

举例：这家公司的技术不断进步，不仅可以生产单核处理器，也能生产多核处理器。现在简单工厂模式和工厂方法模式都鞭长莫及。抽象工厂模式登场了。它的定义为提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。具体这样应用，这家公司还是开设两个工厂，一个专门用来生产A型号的单核多核处理器，而另一个工厂专门用来生产B型号的单核多核处理器，下面给出实现的代码：

```
//程序实例（抽象工厂模式）
//单核
class SingleCore
{
public:
    virtual void Show() = 0;
};
class SingleCoreA: public SingleCore
{
```

```
public:
    void Show() { cout<<"Single Core A"<<endl; }
};
class SingleCoreB :public SingleCore
{
public:
    void Show() { cout<<"Single Core B"<<endl; }
};
//多核
class MultiCore
{
public:
    virtual void Show() = 0;
};
class MultiCoreA : public MultiCore
{
public:
    void Show() { cout<<"Multi Core A"<<endl; }

};
class MultiCoreB : public MultiCore
{
public:
    void Show() { cout<<"Multi Core B"<<endl; }
};
//工厂
class CoreFactory
{
public:
    virtual SingleCore* CreateSingleCore() = 0;
    virtual MultiCore* CreateMultiCore() = 0;
};
//工厂A，专门用来生产A型号的处理器
class FactoryA :public CoreFactory
{
public:
    SingleCore* CreateSingleCore() { return new SingleCoreA(); }
    MultiCore* CreateMultiCore() { return new MultiCoreA(); }
};
//工厂B，专门用来生产B型号的处理器
class FactoryB : public CoreFactory
{
public:
    SingleCore* CreateSingleCore() { return new SingleCoreB(); }
    MultiCore* CreateMultiCore() { return new MultiCoreB(); }
};
```

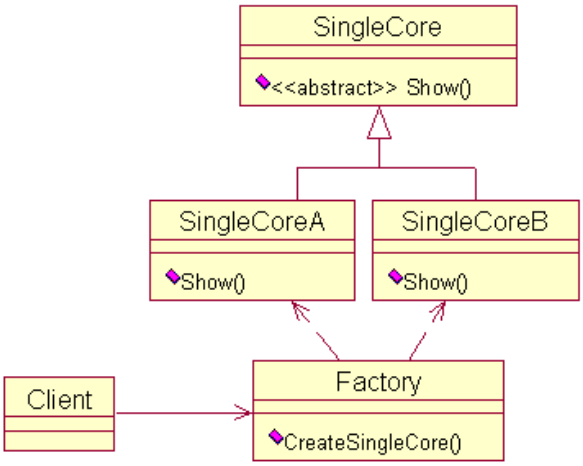
优点： 工厂抽象类创建了多个类型的产品，当有需求时，可以创建相关产品子类和子工厂类来获取。

缺点： 扩展新种类产品时困难。抽象工厂模式需要我们在工厂抽象类中提前确定了可能需要的产品种类，以满足不同型号的多种产品的需求。但是如果我们需要的产品种类并没有在工厂抽象类中提前确定，那我们就需要去修改工厂抽象类了，而一旦修改了工厂抽象类，那么所有的工厂子类也需要修改，这样显然扩展不方便。

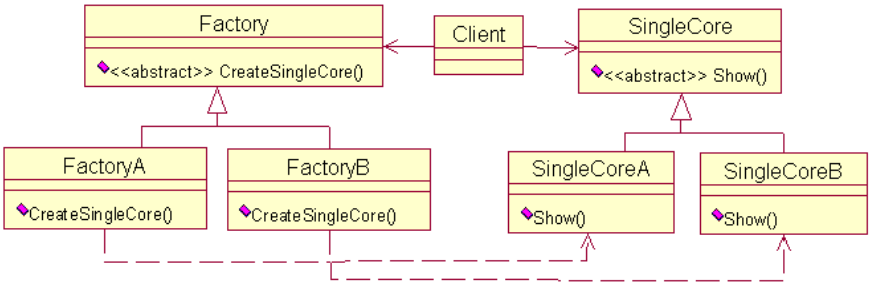
答案解析

三种工厂模式的UML图如下：

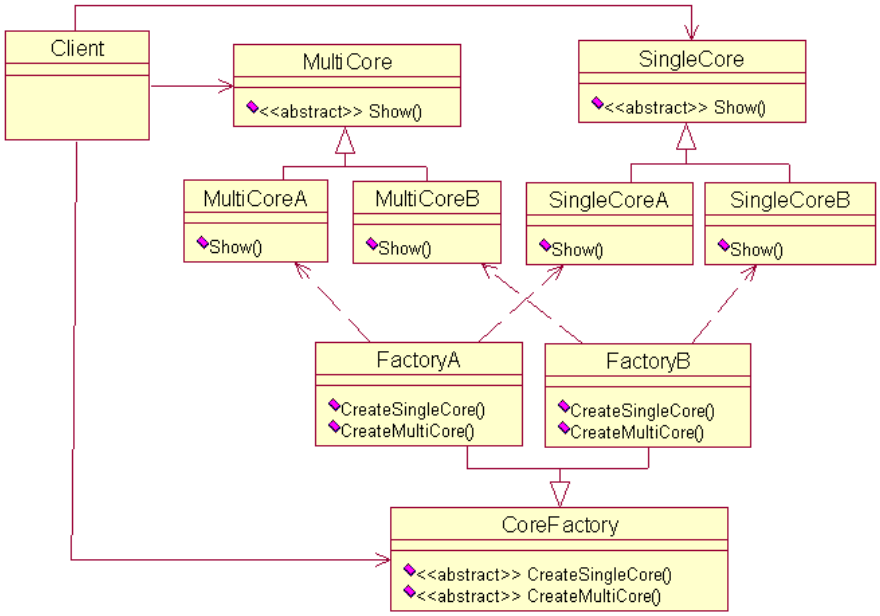
1. 简单工厂模式UML



2. 工厂方法的UML图



3. 抽象工厂模式的UML图



4.4 请说说装饰器计模式，以及它的优缺点

参考回答

1. 装饰器计模式的定义

   指在不改变现有对象结构的情况下，动态地给该对象增加一些职责（即增加其额外功能）的模式，它属于对象结构型模式。

2. 优点

（1）装饰器是继承的有力补充，比继承灵活，在不改变原有对象的情况下，动态的给一个对象扩展功能，即插即用；

（2）通过使用不用装饰类及这些装饰类的排列组合，可以实现不同效果；

（3）装饰器模式完全遵守开闭原则。

3. 缺点

装饰模式会增加许多子类，过度使用会增加程序得复杂性。

4. 装饰模式的结构与实现

通常情况下，扩展一个类的功能会使用继承方式来实现。但继承具有静态特征，耦合度高，并且随着扩展功能的增多，子类会很膨胀。如果使用组合关系来创建一个包装对象（即装饰对象）来包裹真实对象，并在保持真实对象的类结构不变的前提下，为其提供额外的功能，这就是装饰模式的目标。下面来分析其基本结构和实现方法。

装饰模式主要包含以下角色：

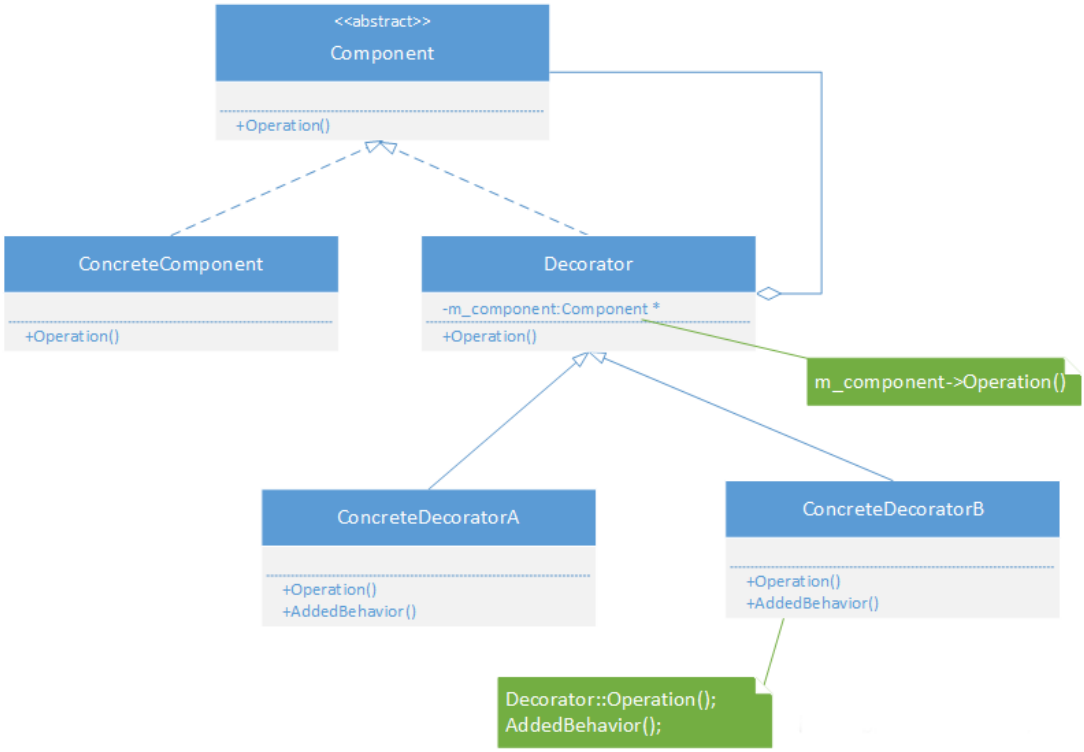
（1）抽象构件（Component）角色：定义一个抽象接口以规范准备接收附加责任的对象。

（2）具体构件（ConcreteComponent）角色：实现抽象构件，通过装饰角色为其添加一些职责。

（3）抽象装饰（Decorator）角色：继承抽象构件，并包含具体构件的实例，可以通过其子类扩展具体构件的功能。

（4）具体装饰（ConcreteDecorator）角色：实现抽象装饰的相关方法，并给具体构件对象添加附加的责任。

装饰模式的结构图如下图所示：



装饰模式结构图

装饰模式的实现代码如下：

```
#include <string>
#include <iostream>

//基础组件接口定义了可以被装饰器修改的操作
```

```cpp
class Component {
public:
  virtual ~Component() {}
  virtual std::string Operation() const = 0;
};

//具体组件提供了操作的默认实现。这些类在程序中可能会有几个变体
class ConcreteComponent : public Component {
public:
  std::string Operation() const override {
    return "ConcreteComponent";
  }
};

//装饰器基类和其他组件遵循相同的接口。这个类的主要目的是为所有的具体装饰器定义封装接口。
//封装的默认实现代码中可能会包含一个保存被封装组件的成员变量，并且负责对齐进行初始化
class Decorator : public Component {

protected:
  Component* component_;

public:
  Decorator(Component* component) : component_(component) {
  }

  //装饰器会将所有的工作分派给被封装的组件
  std::string Operation() const override {
    return this->component_->Operation();
  }
};

//具体装饰器必须在被封装对象上调用方法，不过也可以自行在结果中添加一些内容。
class ConcreteDecoratorA : public Decorator {

//装饰器可以调用父类的是实现，来替代直接调用组件方法。
public:
  ConcreteDecoratorA(Component* component) : Decorator(component) {
  }
  std::string Operation() const override {
    return "ConcreteDecoratorA(" + Decorator::Operation() + ")";
  }
};

//装饰器可以在调用封装的组件对象的方法前后执行自己的方法
class ConcreteDecoratorB : public Decorator {
public:
  ConcreteDecoratorB(Component* component) : Decorator(component) {
  }

  std::string Operation() const override {
    return "ConcreteDecoratorB(" + Decorator::Operation() + ")";
  }
};

//客户端代码可以使用组件接口来操作所有的具体对象。这种方式可以使客户端和具体的实现类脱耦
void ClientCode(Component* component) {
  // ...
  std::cout << "RESULT: " << component->Operation();
```

```
  // ...
}

int main() {

  Component* simple = new ConcreteComponent;
  std::cout << "Client: I've got a simple component:\n";
  ClientCode(simple);
  std::cout << "\n\n";

  Component* decorator1 = new ConcreteDecoratorA(simple);
  Component* decorator2 = new ConcreteDecoratorB(decorator1);
  std::cout << "Client: Now I've got a decorated component:\n";
  ClientCode(decorator2);
  std::cout << "\n";

  delete simple;
  delete decorator1;
  delete decorator2;

  return 0;
}
```

4.5 请说说观察者设计模式，如何实现

参考回答

1. 观察者设计模式的定义

   指多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。这种模式有时又称作发布-订阅模式、模型-视图模式，它是对象行为型模式。

2. 优点

   （1）降低了目标与观察者之间的耦合关系，两者之间是抽象耦合关系。符合依赖倒置原则。

   （2）目标与观察者之间建立了一套触发机制。

3. 缺点

   （1）目标与观察者之间的依赖关系并没有完全解除，而且有可能出现循环引用。

   （2）当观察者对象很多时，通知的发布会花费很多时间，影响程序的效率。

4. 观察者设计模式的结构与实现

   观察者模式的主要角色如下：

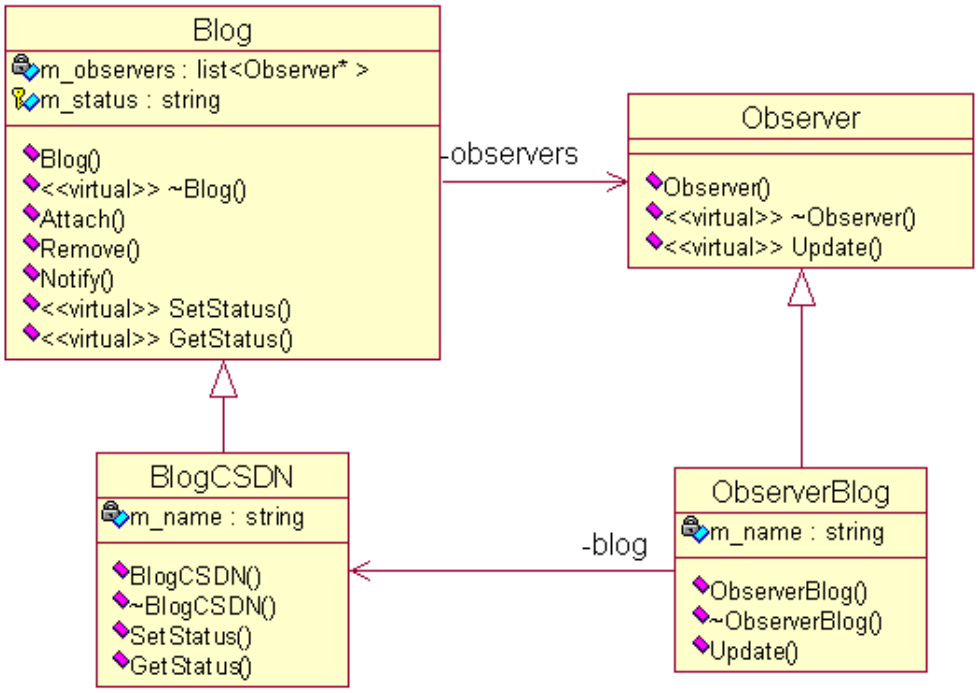
   （1）抽象主题（Subject）角色：也叫抽象目标类，它提供了一个用于保存观察者对象的聚集类和增加、删除观察者对象的方法，以及通知所有观察者的抽象方法。

   （2）具体主题（Concrete Subject）角色：也叫具体目标类，它实现抽象目标中的通知方法，当具体主题的内部状态发生改变时，通知所有注册过的观察者对象。

   （3）抽象观察者（Observer）角色：它是一个抽象类或接口，它包含了一个更新自己的抽象方法，当接到具体主题的更改通知时被调用。

   （4）具体观察者（Concrete Observer）角色：实现抽象观察者中定义的抽象方法，以便在得到目标的更改通知时更新自身的状态。

可以举个博客订阅的例子，当博主发表新文章的时候，即博主状态发生了改变，那些订阅的读者就会收到通知，然后进行相应的动作，比如去看文章，或者收藏起来。博主与读者之间存在种一对多的依赖关系。下面给出相应的UML图设计:



观察者模式的结构图

可以看到博客类中有一个观察者链表（即订阅者），当博客的状态发生变化时，通过Notify成员函数通知所有的观察者，告诉他们博客的状态更新了。而观察者通过Update成员函数获取博客的状态信息。代码实现不难，下面给出C++的一种实现。

```
//观察者
class Observer
{
public:
    Observer() {}
    virtual ~Observer() {}
    virtual void Update() {}
};
//博客
class Blog
{
public:
    Blog() {}
    virtual ~Blog() {}
    void Attach(Observer *observer) { m_observers.push_back(observer); }
//添加观察者
    void Remove(Observer *observer) { m_observers.remove(observer); }
 //移除观察者
    void Notify() //通知观察者
    {
        list<Observer*>::iterator iter = m_observers.begin();
        for(; iter != m_observers.end(); iter++)
            (*iter)->Update();
    }
    virtual void SetStatus(string s) { m_status = s; } //设置状态
    virtual string GetStatus() { return m_status; }    //获得状态
private:
```

```
    list<Observer* > m_observers; //观察者链表
protected:
    string m_status; //状态
};
```

以上是观察者和博客的基类，定义了通用接口。博客类主要完成观察者的添加、移除、通知操作，设置和获得状态仅仅是一个默认实现。下面给出它们相应的子类实现。

```
//具体博客类
class BlogCSDN : public Blog
{
private:
    string m_name; //博主名称
public:
    BlogCSDN(string name): m_name(name) {}
    ~BlogCSDN() {}
    void SetStatus(string s) { m_status = "CSDN通知 : " + m_name + s; } //具体设置状态信息
    string GetStatus() { return m_status; }
};
//具体观察者
class ObserverBlog : public Observer
{
private:
    string m_name;  //观察者名称
    Blog *m_blog;   //观察的博客，当然以链表形式更好，就可以观察多个博客
public:
    ObserverBlog(string name,Blog *blog): m_name(name), m_blog(blog) {}
    ~ObserverBlog() {}
    void Update()  //获得更新状态
    {
        string status = m_blog->GetStatus();
        cout<<m_name<<"-------"<<status<<endl;
    }
};
//测试案例
int main()
{
    Blog *blog = new BlogCSDN("wuzhekai1985");
    Observer *observer1 = new ObserverBlog("tutupig", blog);
    blog->Attach(observer1);
    blog->SetStatus("发表设计模式C++实现（15）--观察者模式");
    blog->Notify();
    delete blog; delete observer1;
    return 0;
}
```