分析下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** test1()  {  **char** string[10];  **char**\* str1 = "0123456789";  **strcpy**( string, str1 );  } |

# **参考答案**

字符串str1需要11个字节才能存放下（包括末尾的’\0’），而string只有10个字节的空间，strcpy会导致数组越界；

分析下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **void** test2()  {  **char** string[10], str1[10];  **int** i;  **for**(i=0; i<10; i++)   {   str1  = 'a';   }  **strcpy**( string, str1 );  } |

# **参考答案**

首先，代码根本不能通过编译。因为数组名str1为 char \*const类型的右值类型，根本不能赋值。

再者，即使想对数组的第一个元素赋值，也要使用 \*str1 = 'a';

其次，对字符数组赋值后，使用库函数strcpy进行拷贝操作，strcpy会从源地址一直往后拷贝，直到遇到'\0'为止。所以拷贝的长度是不定的。如果一直没有遇到'\0'导致越界访问非法内存，程序就崩了。

完美修改方案为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** test2()  {  **char** string[10], str1[10];  **int** i;  **for**(i=0; i<9; i++)      {          str1[i]  = 'a';      }      str1[9] = '\0';  **strcpy**( string, str1 );  } |

指出下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** test3(**char**\* str1)  {  **if**(str1 == NULL){  **return** ;   }  **char** string[10];  **if**( **strlen**( str1 ) <= 10 )   {  **strcpy**( string, str1 );   }  } |

# **参考答案**

if(strlen(str1) <= 10)应改为if(strlen(str1) < 10)，因为strlen的结果未统计’\0’所占用的1个字节

写出完整版的strcpy函数

# **参考答案**

如果编写一个标准strcpy函数的总分值为10，下面给出几个不同得分的答案：  
2分

|  |  |
| --- | --- |
| 1  2  3  4 | **void** **strcpy**( **char** \*strDest, **char** \*strSrc )  {  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

4分

|  |  |
| --- | --- |
| 1  2  3  4  5 | **void** **strcpy**( **char** \*strDest, **const** **char** \*strSrc )  //将源字符串加const，表明其为输入参数，加2分  {  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

7分

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** **strcpy**(**char** \*strDest, **const** **char** \*strSrc)  {   //对源地址和目的地址加非0断言，加3分  **assert**( (strDest != NULL) && (strSrc != NULL) );  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  } |

10分  
//为了实现链式操作，将目的地址返回，加3分！

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **char** \* **strcpy**( **char** \*strDest, **const** **char** \*strSrc )  {  **assert**( (strDest != NULL) && (strSrc != NULL) );  **char** \*address = strDest;  **while**( (\*strDest++ = \* strSrc++) != ‘\0’ );  **return** address; |

|  |  |
| --- | --- |
| 1 | } |

检查下面代码有什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** GetMemory( **char** \*p )  {   p = (**char** \*) **malloc**( 100 );  }  **void** Test( **void** )  {  **char** \*str = NULL;   GetMemory( str );  **strcpy**( str, "hello world" );  **printf**( str );  } |

# **参考答案**

传入中GetMemory( char \*p )函数的形参为字符串指针，在函数内部修改形参并不能真正的改变传入形参的实参值，执行完

|  |  |
| --- | --- |
| 1  2 | **char** \*str = NULL;  GetMemory( str ); |

后的str仍然为NULL；

1：传入形参并不能真正改变形参的值，执行完之后为空；

2：在函数GetMemory中和Test中没有malloc对应的free，造成内存泄露

需要二级指针,才能改变

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **char** \*GetMemory( **void** )  {  **char** p[] = "hello world";  **return** p;  }  **void** Test( **void** )  {  **char** \*str = NULL;   str = GetMemory();  **printf**( str );  } |

# **参考答案**

|  |  |
| --- | --- |
| 1  2 | **char** p[] = "hello world";  **return** p; |

的p[]数组为函数内的局部自动变量，在函数返回后，内存已经被释放。这是许多程序员常犯的错误，其根源在于不理解变量的生存期。

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **void** GetMemory( **char** \*\*p, **int** num )  {   \*p = (**char** \*) **malloc**( num );  }  **void** Test( **void** )  {  **char** \*str = NULL;   GetMemory( &str, 100 );  **strcpy**( str, "hello" );  **printf**( str );  } |

# **参考答案**

1. 传入GetMemory的参数为字符串指针的指针，但是在GetMemory中执行申请内存及赋值语句

|  |  |
| --- | --- |
| 1 | \*p = (**char** \*) **malloc**( num ); |

后未判断内存是否申请成功，应加上：

|  |  |
| --- | --- |
| 1  2  3  4  5 | **if** ( \*p == NULL )  {   ...//进行申请内存失败处理  }  同时应考虑num>0； |

2. 未释放堆内存 动态分配的内存在程序结束之前没有释放，应该调用free, 把malloc生成的内存释放掉

3. printf(str) 改为 printf("%s",str),否则可使用格式化 字符串攻击

下面代码会出现什么问题？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **void** Test( **void** )  {  **char** \*str = (**char** \*) **malloc**( 100 );  **strcpy**( str, "hello" );  **free**( str );   ... //省略的其它语句  } |

# **参考答案**

在执行    
char \*str = (char \*) malloc(100);    
后未进行内存是否申请成功的判断；另外，在free(str)后未置str为空，导致可能变成一个“野”指针，应加上：    
str = NULL;    
试题6的Test函数中也未对malloc的内存进行释放。

看看下面的一段程序有什么错误?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | swap( **int**\* p1,**int**\* p2 )  {  **int** \*p;   \*p = \*p1;   \*p1 = \*p2;   \*p2 = \*p;  } |

# **参考答案**

1.需要一个返回值void

2在swap函数中，p是一个“野”指针，有可能指向系统区，导致程序运行的崩溃。在VC++中DEBUG运行时提示错误“Access Violation”。该程序应该改为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | **void** swap( **int**\* p1,**int**\* p2 )  {  **int** p;   p = \*p1;   \*p1 = \*p2;   \*p2 = p;  } |

分别给出BOOL，int，float，指针变量 与“零值”比较的 if 语句（假设变量名为var）

# **参考答案**

【解答】

BOOL型变量：if(!var)

int型变量： if(var==0)

float型变量：

const float EPSINON = 0.00001;

if ((x >= - EPSINON) && (x <= EPSINON)

指针变量：　　if(var==NULL)

【剖析】

考查对0值判断的“内功”，BOOL型变量的0判断完全可以写成if(var==0)，而int型变量也可以写成if(!var)，指针变量的判断也可以写成if(!var)，上述写法虽然程序都能正确运行，但是未能清晰地表达程序的意思。

一般的，如果想让if判断一个变量的“真”、“假”，应直接使用if(var)、if(!var)，表明其为“逻辑”判断；如果用if判断一个数值型变量(short、int、long等)，应该用if(var==0)，表明是与0进行“数值”上的比较；而判断指针则适宜用if(var==NULL)，这是一种很好的编程习惯。

浮点型变量并不精确，所以不可将float变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。如果写成if (x == 0.0)，则判为错，得0分。

以下为Windows NT下的32位C++程序，请计算sizeof的值

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **void** Func ( **char** str[100] )  {  **sizeof**( str ) = ?  }  **void** \*p = **malloc**( 100 );  **sizeof** ( p ) = ? |

# **参考答案**

sizeof( str ) = 4    
sizeof ( p ) = 4    
【剖析】    
Func ( char str[100] )函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。    
数组名的本质如下：    
（1）数组名指代一种数据结构，这种数据结构就是数组；    
例如：

|  |  |
| --- | --- |
| 1  2 | **char** str[10];  cout ＜＜ **sizeof**(str) ＜＜ endl; |

输出结果为10，str指代数据结构char[10]。    
（2）数组名可以转换为指向其指代实体的指针，而且是一个指针常量，不能作自增、自减等操作，不能被修改；    
char str[10];     
str++; //编译出错，提示str不是左值

（3）数组名作为函数形参时，沦为普通指针。

Windows NT 32位平台下，指针的长度（占用内存的大小）为4字节，故sizeof( str ) 、sizeof ( p ) 都为4。

写一个“标准”宏MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事？    
least = MIN(\*p++, b);

# **参考答案**

解答：

|  |  |
| --- | --- |
| 1 | #define MIN(A,B) ((A) <= (B) ? (A) : (B)) |

MIN(\*p++, b)会产生宏的副作用    
剖析：    
这个面试题主要考查面试者对宏定义的使用，宏定义可以实现类似于函数的功能，但是它终归不是函数，而宏定义中括弧中的“参数”也不是真的参数，在宏展开的时候对“参数”进行的是一对一的替换。    
程序员对宏定义的使用要非常小心，特别要注意两个问题：    
（1）谨慎地将宏定义中的“参数”和整个宏用用括弧括起来。所以，严格地讲，下述解答：

|  |  |
| --- | --- |
| 1  2 | #define MIN(A,B) (A) <= (B) ? (A) : (B)  #define MIN(A,B) (A <= B ? A : B ) |

都应判0分；    
（2）防止宏的副作用。    
宏定义#define MIN(A,B) ((A) <= (B) ? (A) : (B))对MIN(\*p++, b)的作用结果是：

|  |  |
| --- | --- |
| 1 | ((\*p++) <= (b) ? (\*p++) : (b)) |

这个表达式会产生副作用，指针p会作2次++自增操作。

除此之外，另一个应该判0分的解答是：

|  |  |
| --- | --- |
| 1 | #define MIN(A,B) ((A) <= (B) ? (A) : (B)); |

这个解答在宏定义的后面加“;”，显示编写者对宏的概念模糊不清，只能被无情地判0分并被面试官淘汰。

为什么标准头文件都有类似以下的结构？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #ifndef \_\_INCvxWorksh  #define \_\_INCvxWorksh  #ifdef \_\_cplusplus  **extern** "C" {  #endif  /\*...\*/  #ifdef \_\_cplusplus  }  #endif  #endif /\* \_\_INCvxWorksh \*/ |

# **参考答案**

头文件中的编译宏

|  |  |
| --- | --- |
| 1  2  3 | #ifndef　\_\_INCvxWorksh  #define　\_\_INCvxWorksh  #endif |

的作用是防止被重复引用。  
作为一种面向对象的语言，C++支持函数重载，而过程式语言C则不支持。函数被C++编译后在symbol库中的名字与C语言的不同。例如，假设某个函数的原型为：  
void foo(int x, int y);  
该函数被C编译器编译后在symbol库中的名字为\_foo，而C++编译器则会产生像\_foo\_int\_int之类的名字。\_foo\_int\_int这样的名字包含了函数名和函数参数数量及类型信息，C++就是靠这种机制来实现函数重载的。  
为了实现C和C++的混合编程，C++提供了C连接交换指定符号extern "C"来解决名字匹配问题，函数声明前加上extern "C"后，则编译器就会按照C语言的方式将该函数编译为\_foo，这样C语言中就可以调用C++的函数了

编写一个函数，作用是把一个char组成的字符串循环右移n个。比如原来是“abcdefghi”如果n=2，移位后应该是“hiabcdefg” 函数头是这样的：  
//pStr是指向以'\0'结尾的字符串的指针  
//steps是要求移动的n

|  |  |
| --- | --- |
| 1  2  3  4 | **void** LoopMove ( **char** \* pStr, **int** steps )  {   //请填充...  } |

# **参考答案**

void LoopMove(char \*str, int steps)

{

int len = strlen(str);

char tmp[MAXSIZE];

strcpy(tmp, str+len-steps);

strcpy(tmp+steps, str);

\*(tmp+len)  = '/0';

strcpy(str, tmp);

}

或:

void LoopMove(char \*str, int steps)

{

int len = strlen(str);

char tmp[MAXSIZE];

memcpy(tmp, str+len-steps, steps);

memcpy(str+steps, str, len-steps);

memcpy(str, tmp, steps);

}

编写类String的构造函数、析构函数和赋值函数，已知类String的原型为：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **class** String  {  **public**:   String(**const** **char** \*str = NULL); // 普通构造函数   String(**const** String &other); // 拷贝构造函数   ~ String(**void**); // 析构函数   String & operator =(**const** String &other); // 赋值函数  **private**:  **char** \*m\_data; // 用于保存字符串  }; |

# **参考答案**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | //普通构造函数  String::String(**const** **char** \*str)  {  **if**(str==NULL)   {   m\_data = **new** **char**[1]; // 得分点：对空字符串自动申请存放结束标志'\0'的空   //加分点：对m\_data加NULL 判断   \*m\_data = '\0';   }  **else**   {  **int** length = **strlen**(str);   m\_data = **new** **char**[length+1];  **strcpy**(m\_data, str);   }  }  // String的析构函数  String::~String(**void**)  {  **delete** [] m\_data; // 或delete m\_data;  }  //拷贝构造函数  String::String(**const** String &other) 　　　// 得分点：输入参数为const型  {  **int** length = **strlen**(other.m\_data);   m\_data = **new** **char**[length+1];  **strcpy**(m\_data, other.m\_data);  }  //赋值函数  String & String::operator =(**const** String &other) // 得分点：输入参数为const型  {  **if**(**this** == &other) 　　//得分点：检查自赋值  **return** \***this**;  **delete** [] m\_data; 　　　　//得分点：释放原有的内存资源  **int** length = **strlen**( other.m\_data );   m\_data = **new** **char**[length+1];  **strcpy**( m\_data, other.m\_data );  **return** \***this**; 　　　　　　　　//得分点：返回本对象的引用  } |

剖析  
能够准确无误地编写出String类的构造函数、拷贝构造函数、赋值函数和析构函数的面试者至少已经具备了C++基本功的60%以上！  
在这个类中包括了指针类成员变量m\_data，当类中包括指针类成员变量时，一定要重载其拷贝构造函数、赋值函数和析构函数，这既是对C++程序员的基本要求，也是《Effective　C++》中特别强调的条款。  
仔细学习这个类，特别注意加注释的得分点和加分点的意义，这样就具备了60%以上的C++基本功！

请说出static和const关键字尽可能多的作用

# **参考答案**

【解答】  
static关键字至少有下列n个作用：    
（1）函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；    
（2）在模块内的static全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；    
（3）在模块内的static函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；    
（4）在类中的static成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；    
（5）在类中的static成员函数属于整个类所拥有，这个函数不接收this指针，因而只能访问类的static成员变量。     
const关键字至少有下列n个作用：    
（1）欲阻止一个变量被改变，可以使用const关键字。在定义该const变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；    
（2）对指针来说，可以指定指针本身为const，也可以指定指针所指的数据为const，或二者同时指定为const；    
（3）在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；    
（4）对于类的成员函数，若指定其为const类型，则表明其是一个常函数，不能修改类的 成员变量；    
（5）对于类的成员函数，有时候必须指定其返回值为const类型，以使得其返回值不为“左值”。例如：    
const classA operator\*(const classA& a1,const classA& a2);    
operator\*的返回结果必须是一个const对象。如果不是，这样的变态代码也不会编译出错：    
classA a, b, c;    
(a \* b) = c; // 对a\*b的结果赋值    
操作(a \* b) = c显然不符合编程者的初衷，也没有任何意义。

写一个函数返回1+2+3+…+n的值（假定结果不会超过长整型变量的范围）

# **参考答案**

【解答】

|  |  |
| --- | --- |
| 1  2  3  4 | **int** Sum( **int** n )  {  **return** ( (**long**)1 + n) \* n / 2;　　//或return (1l + n) \* n / 2;  } |

【剖析】  
对于这个题，只能说，也许最简单的答案就是最好的答案。下面的解答，或者基于下面的解答思路去优化，不管怎么“折腾”，其效率也不可能与直接return ( 1 l + n ) \* n / 2相比！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **int** Sum( **int** n )  {  **long** sum = 0;  **for**( **int** i=1; i<=n; i++ )   {   sum += i;   }  **return** sum;  } |

说一下static关键字的作用

参考答案

参考回答:

1. 全局静态变量

在全局变量前加上关键字static，全局变量就定义成一个全局静态变量.

静态存储区，在整个程序运行期间一直存在。

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

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

2. 局部静态变量

在局部变量之前加上关键字static，局部变量就成为一个局部静态变量。

内存中的位置：静态存储区

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

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

3. 静态函数

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

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

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

4. 类的静态成员

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

5. 类的静态函数

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

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

说一下C++和C的区别

# **参考答案**

参考回答:

设计思想上：

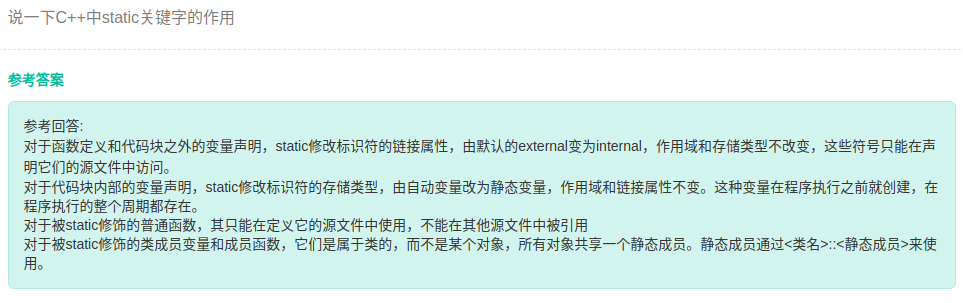
C++是面向对象的语言，而C是面向过程的结构化编程语言

语法上：

C++具有封装、继承和多态三种特性

C++相比C，增加多许多类型安全的功能，比如强制类型转换、

C++支持范式编程，比如模板类、函数模板等



说一说c++中四种cast转换

参考答案

参考回答:

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

1、const\_cast

用于将const变量转为非const

2、static\_cast

用于各种隐式转换，比如非const转const，void\*转指针等, static\_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic\_cast

用于动态类型转换。只能用于含有虚函数的类，用于类层次间的向上和向下转化。只能转指针或引用。向下转化时，如果是非法的对于指针返回NULL，对于引用抛异常。要深入了解内部转换的原理。

向上转换：指的是子类向基类的转换

向下转换：指的是基类向子类的转换

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

4、reinterpret\_cast

几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换？

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

请说一下C/C++ 中指针和引用的区别？

参考回答:

1.指针有自己的一块空间，而引用只是一个别名；

2.使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；

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

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

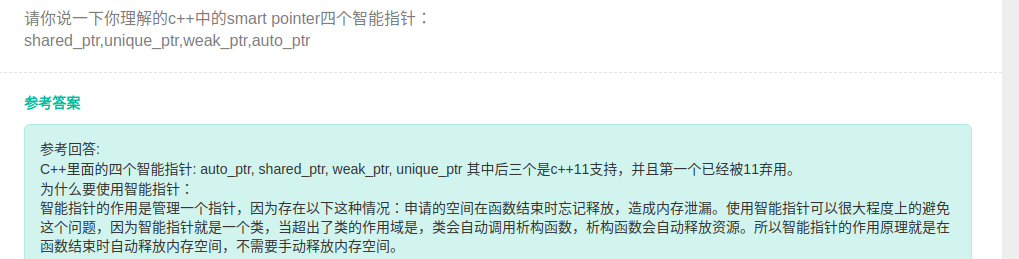
5.可以有const指针，但是没有const引用；

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

7.指针可以有多级指针（\*\*p），而引用至于一级；

8.指针和引用使用++运算符的意义不一样；

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



怎么判断一个数是二的倍数，怎么求一个数中有几个1，说一下你的思路并手写代码

参考答案

参考回答:

1、判断一个数是不是二的倍数，即判断该数二进制末位是不是0：

a % 2 == 0 或者a & 0x0001 == 0。

2、求一个数中1的位数，可以直接逐位除十取余判断：

int fun(long x)

{

int \_count = 0;

while(x)

{

if(x % 10 == 1)

++\_count;

x /= 10;

}

return \_count;

}

int main()

{

cout << fun(123321) << endl;

return 0;

}



请你回答一下野指针是什么？

# **参考答案**

参考回答:

野指针就是指向一个已删除的对象或者未申请访问受限内存区域的指针

请你介绍一下C++中的智能指针

# **参考答案**

参考回答:

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

请你回答一下智能指针有没有内存泄露的情况

# **参考答案**

参考回答:

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

请你来说一下智能指针的内存泄漏如何解决

# **参考答案**

参考回答:

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

请你理解的c++中的引用和指针

# **参考答案**

参考回答:

定义：

1、引用：

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

2、指针：

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

区别：

1. 指针有自己的一块空间，而引用只是一个别名；2、使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；3、指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用；4、作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象；5、可以有const指针，但是没有const引用；6、指针在使用中可以指向其它对象，但是引用只能是一个对象的引用，不能 被改变；7、指针可以有多级指针（\*\*p），而引用至于一级；8、指针和引用使用++运算符的意义不一样；9、如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

请你回答一下为什么析构函数必须是虚函数？为什么C++默认的析构函数不是虚函数

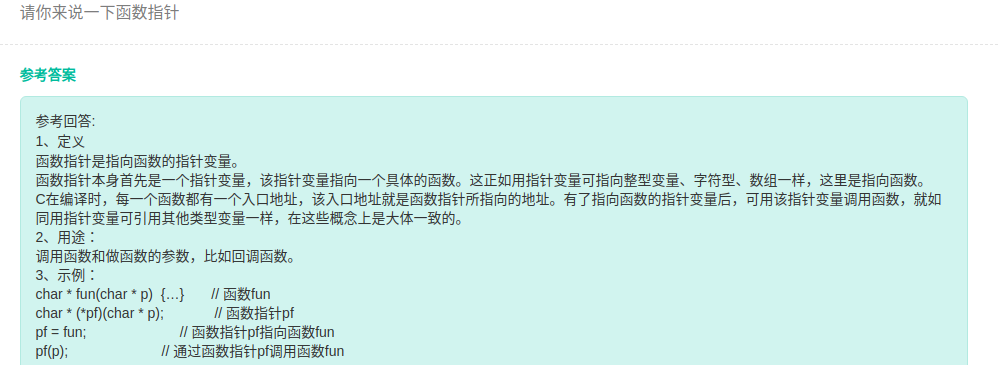
考点:虚函数 析构函数

# **参考答案**

参考回答:

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

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



请你来说一下fork函数

# **参考答案**

参考回答:

Fork：创建一个和当前进程映像一样的进程可以通过fork( )系统调用：

#include <sys/types.h>

#include <unistd.h>

pid\_t fork(void);

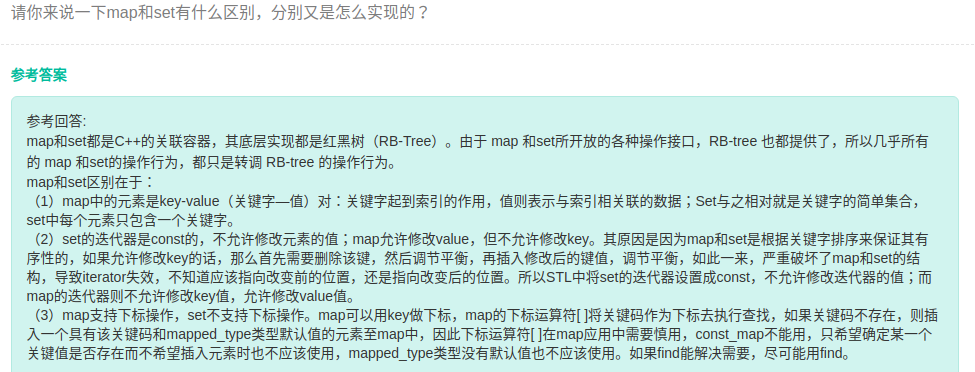
成功调用fork( )会创建一个新的进程，它几乎与调用fork( )的进程一模一样，这两个进程都会继续运行。在子进程中，成功的fork( )调用会返回0。在父进程中fork( )返回子进程的pid。如果出现错误，fork( )返回一个负值。 最常见的fork( )用法是创建一个新的进程，然后使用exec( )载入二进制映像，替换当前进程的映像。这种情况下，派生（fork）了新的进程，而这个子进程会执行一个新的二进制可执行文件的映像。这种“派生加执行”的方式是很常见的。 在早期的Unix系统中，创建进程比较原始。当调用fork时，内核会把所有的内部数据结构复制一份，复制进程的页表项，然后把父进程的地址空间中的内容逐页的复制到子进程的地址空间中。但从内核角度来说，逐页的复制方式是十分耗时的。现代的Unix系统采取了更多的优化，例如Linux，采用了写时复制的方法，而不是对父进程空间进程整体复制。

请你来说一下C++中析构函数的作用

# **参考答案**

参考回答:

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。 析构函数名也应与类名相同，只是在函数名前面加一个位取反符~，例如~stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。 如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。 如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。 类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。



41>