

# C++笔试题汇总

## ①链表反转

单向链表的反转是一个经常被问到的一个面试题，也是一个非常基础的问题。比如一个链表是这样的： 1->2->3->4->5 通过反转后成为 5->4->3->2->1。

最容易想到的方法遍历一遍链表，利用一个辅助指针，存储遍历过程中当前指针指向的下一个元素，然后将当前节点元素的指针反转后，利用已经存储的指针往后面继续遍历。源代码如下：

```
1. struct linka {  
2.     int data;  
3.     linka* next;  
4. };  
5. void reverse(linka*& head) {  
6.     if(head ==NULL)  
7.         return;  
8.     linka *pre, *cur, *ne;  
9.     pre=head;  
10.    cur=head->next;  
11.    while(cur)  
12.    {  
13.        ne = cur->next;  
14.        cur->next = pre;  
15.        pre = cur;  
16.        cur = ne;  
17.    }  
18.    head->next = NULL;  
19.    head = pre;  
20. }
```

还有一种利用递归的方法。这种方法的基本思想是在反转当前节点之前先调用递归函数反转后续节点。源代码如下。不过这个方法有一个缺点，就是在反转后的最后一个结点会形成一个环，所以必须将函数的返回的节点的 `next` 域置为 `NULL`。因为要改变 `head` 指针，所以我用了引用。算法的源代码如下：

```
1. linka* reverse(linka* p,linka*& head)  
2. {  
3.     if(p == NULL || p->next == NULL)
```

```
4. {
5.     head=p;
6.     return p;
7. }
8. else
9. {
10.    linka* tmp = reverse(p->next,head);
11.    tmp->next = p;
12.    return p;
13. }
14. }
```

## ②已知 **String** 类定义如下：

```
class String
{
public:
String(const char *str = NULL); // 通用构造函数
String(const String &another); // 拷贝构造函数
~String(); // 析构函数
String & operator =(const String &rhs); // 赋值函数
private:
char *m_data; // 用于保存字符串
};
```

尝试写出类的成员函数实现。

答案：

```
String::String(const char *str)
{
if ( str == NULL ) //strlen 在参数为 NULL 时会抛异常才会有这步判断
{
m_data = new char[1] ;
m_data[0] = '\0' ;
}
else
{
m_data = new char[strlen(str) + 1];
strcpy(m_data,str);
}
}
```

```
String::String(const String &another)
{
    m_data = new char[strlen(another.m_data) + 1];
    strcpy(m_data, another.m_data);
}
```

```
String& String::operator =(const String &rhs)
{
    if (this == &rhs)
        return *this;
    delete []m_data; //删除原来的数据，新开一块内存
    m_data = new char[strlen(rhs.m_data) + 1];
    strcpy(m_data, rhs.m_data);
    return *this;
}
```

```
String::~String()
{
    delete []m_data;
}
```

### ③网上流传的 C++ 笔试题汇总

#### 1. 求下面函数的返回值（微软）

```
int func(x)
{
    int countx = 0;
    while(x)
    {
        countx++;
        x = x & (x-1);
    }
    return countx;
}
```

假定  $x = 9999$ 。 答案: 8

思路: 将  $x$  转化为 2 进制, 看含有的 1 的个数。

#### 2. 什么是“引用”? 申明和使用“引用”要注意哪些问题?

答：引用就是某个目标变量的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行初始化。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

### 3. 将“引用”作为函数参数有哪些特点？

(1) 传递引用给函数与传递指针的效果是一样的。这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

(2) 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

(3) 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中同样要给形参分配存储单元，且需要重复使用“\*指针变量名”的形式进行运算，这很容易产生错误且程序的阅读性较差；另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

### 4. 在什么时候需要使用“常引用”？

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：const 类型标识符 &引用名=目标变量名；

#### 例 1

```
int a ;
const int &ra=a;
ra=1; //错误
a=1; //正确
```

#### 例 2

```
string foo( );
void bar(string & s);
```

那么下面的表达式将是非法的：

```
bar(foo( ));
```

```
bar("hello world");
```

原因在于 `foo()` 和 "hello world" 串都会产生一个临时对象，而在 C++ 中，这些临时对象都是 `const` 类型的。因此上面的表达式就是试图将一个 `const` 类型的对象转换为非 `const` 类型，这是非法的。

引用型参数应该在能被定义为 `const` 的情况下，尽量定义为 `const`。

## 5. 将“引用”作为函数返回值类型的格式、好处和需要遵守的规则？

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }

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

注意事项：

（1）不能返回局部变量的引用。这条可以参照 [Effective C++\[1\]](#) 的 Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了“无所指”的引用，程序会进入未知状态。

（2）不能返回函数内部 `new` 分配的内存的引用。这条可以参照 [Effective C++\[1\]](#) 的 Item 31。虽然不存在局部变量的被动销毁问题，但对于这种情况（返回函数内部 `new` 分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由 `new` 分配）就无法释放，造成 `memory leak`。

（3）可以返回类成员的引用，但最好是 `const`。这条原则可以参照 [Effective C++\[1\]](#) 的 Item 30。主要原因是当对象的属性是与某种业务规则（`business rule`）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（4）流操作符重载返回值申明为“引用”的作用：

流操作符 `<<` 和 `>>`，这两个操作符常常希望被连续使用，例如： `cout << "hello" << endl;` 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个 `<<` 操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用 `<<` 操作符。因此，返回一个流对象引用是唯一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是 C++ 语言中引入引用这个概念的原因吧。赋值操作符 `=`。这个操作符像流操作符一样，是可以连续使用的，例如： `x = j = 10;` 或者 `(x=10)=100;` 赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用

成了这个操作符的惟一返回值选择。

### 例 3

```
#i nclude <iostream.h>
int &put(int n);
int vals[10];
int error=-1;
void main()
{
    put(0)=10; //以 put(0)函数值作为左值, 等价于 vals[0]=10;
    put(9)=20; //以 put(9)函数值作为左值, 等价于 vals[9]=20;
    cout<<vals[0];
    cout<<vals[9];
}
int &put(int n)
{
    if (n>=0 && n<=9 ) return vals[n];
    else { cout<<"subscript error"; return error; }
}
```

(5) 在另外的一些操作符中, 却千万不能返回引用: `+-*/` 四则运算符。它们不能返回引用, **Effective C++[1]** 的 Item23 详细的讨论了这个问题。主要原因是这四个操作符没有 **side effect**, 因此, 它们必须构造一个对象作为返回值, 可选的方案包括: 返回一个对象、返回一个局部变量的引用, 返回一个 `new` 分配的对象的引用、返回一个静态对象引用。根据前面提到的引用作为返回值的三个规则, 第 2、3 两个方案都被否决了。静态对象的引用又因为  $((a+b) == (c+d))$  会永远为 `true` 而导致错误。所以可选的只剩下返回一个对象了。

## 6. “引用”与多态的关系?

引用是除指针外另一个可以产生多态效果的手段。这意味着, 一个基类的引用可以指向它的派生类实例。

### 例 4

```
Class A; Class B : Class A{...}; B b; A& ref = b;
```

## 7. “引用”与指针的区别是什么?

指针通过某个指针变量指向一个对象后, 对它所指向的变量间接操作。程序中使用指针, 程序的可读性差; 而引用本身就是目标变量的别名, 对引用的操作就是对目标变量的操作。此外, 就是上面提到的对函数传 `ref` 和 `pointer` 的区别。

## 8. 什么时候需要“引用”？

流操作符<<和>>、赋值操作符=的返回值、拷贝构造函数的参数、赋值操作符=的参数、其它情况都推荐使用引用。

以上 2-8 参考: <http://blog.csdn.net/wfwd/archive/2006/05/30/763551.aspx>

## 9. 结构与联合有和区别？

1. 结构和联合都是由多个不同的数据类型成员组成，但在任何同一时刻，联合中只存放了一个被选中的成员（所有成员共用一块地址空间），而结构的所有成员都存在（不同成员的存放地址不同）。
2. 对于联合的不同成员赋值，将会对其它成员重写，原来成员的值就不存在了，而对于结构的不同成员赋值是互不影响的。

## 10. 下面关于“联合”的题目的输出？

a)

```
#include <stdio.h>
union
{
    int i;
    char x[2];
}a;
```

```
void main()
```

```
{  
    a.x[0] = 10;  
    a.x[1] = 1;  
    printf("%d",a.i);  
}
```

答案: 266 (低位低地址, 高位高地址, 内存占用情况是 0x010A)

b)

```
main()
{
    union{ /*定义一个联合*/
        int i;
```

```

struct{ /*在联合中定义一个结构*/
char first;
char second;
}half;
}number;
number.i=0x4241; /*联合成员赋值*/
printf("%c%c\n", number.half.first, number.half.second);
number.half.first='a'; /*联合中结构成员赋值*/
number.half.second='b';
printf("%x\n", number.i);
getch();
}

```

答案: AB (0x41 对应'A', 是低位; 0x42 对应'B', 是高位)

6261 (number.i 和 number.half 共用一块地址空间)

**11.** 已知 **strcpy** 的函数原型: **char \*strcpy(char \*strDest, const char \*strSrc)** 其中 **strDest** 是目的字符串, **strSrc** 是源字符串。不调用 **C++/C** 的字符串库函数, 请编写函数 **strcpy**。

答案:

```

char *strcpy(char *strDest, const char *strSrc)
{
if ( strDest == NULL || strSrc == NULL)
return NULL ;
if ( strDest == strSrc)
return strDest ;
char *tempptr = strDest ;
while( (*strDest++ = *strSrc++) != '\0')
return tempptr ;
}

```

**12.** 已知 **String** 类定义如下:

```

class String
{
public:
String(const char *str = NULL); // 通用构造函数
String(const String &another); // 拷贝构造函数

```

```
~String(); // 析构函数
String & operator =(const String &rhs); // 赋值函数
private:
char *m_data; // 用于保存字符串
};
```

尝试写出类的成员函数实现。

答案：

```
String::String(const char *str)
{
if ( str == NULL ) //strlen 在参数为 NULL 时会抛异常才会有这步判断
{
m_data = new char[1] ;
m_data[0] = '\0' ;
}
else
{
m_data = new char[strlen(str) + 1];
strcpy(m_data,str);
}
}
```

```
String::String(const String &another)
{
m_data = new char[strlen(another.m_data) + 1];
strcpy(m_data,another.m_data);
}
```

```
String& String::operator =(const String &rhs)
{
if ( this == &rhs)
return *this ;
delete []m_data; //删除原来的数据，新开一块内存
m_data = new char[strlen(rhs.m_data) + 1];
strcpy(m_data,rhs.m_data);
return *this ;
}
```

```
String::~String()
```

```
{  
delete []m_data ;  
}
```

### 13. .h 头文件中的 **ifndef/define/endif** 的作用?

答: 防止该头文件被重复引用。

### 14. **#include<file.h>** 与 **#include "file.h"** 的区别?

答: 前者是从 Standard Library 的路径寻找和引用 file.h, 而后者是从当前工作路径搜寻并引用 file.h。

### 15. 在 C++ 程序中调用被 C 编译器编译后的函数, 为什么要加 **extern "C"**?

首先, 作为 **extern** 是 C/C++ 语言中表明函数和全局变量作用范围(可见性)的关键字, 该关键字告诉编译器, 其声明的函数和变量可以在本模块或其它模块中使用。

通常, 在模块的头文件中对本模块提供给其它模块引用的函数和全局变量以关键字 **extern** 声明。例如, 如果模块 B 欲引用该模块 A 中定义的全局变量和函数时只需包含模块 A 的头文件即可。这样, 模块 B 中调用模块 A 中的函数时, 在编译阶段, 模块 B 虽然找不到该函数, 但是并不会报错; 它会在连接阶段中从模块 A 编译生成的目标代码中找到此函数。

**extern "C"** 是连接申明(linkage declaration), 被 **extern "C"** 修饰的变量和函数是按照 C 语言方式编译和连接的, 来看看 C++ 中对类似 C 的函数是怎样编译的:

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++ 编译后在符号库中的名字与 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 编译器编译后在符号库中的名字为 **\_foo**, 而 C++ 编译器则会产生像 **\_foo\_int\_int** 之类的名字(不同的编译器可能生成的名字不同, 但是都采用了相同的机制, 生成的新名字称为“mangled name”)。

**\_foo\_int\_int** 这样的名字包含了函数名、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。例如, 在 C++ 中, 函数 **void foo( int x, int y )** 与 **void foo( int x, float y )** 编译生成的符号是不相同的, 后者为 **\_foo\_int\_float**。

同样地，C++中的变量除支持局部变量外，还支持类成员变量和全局变量。用户所编写程序的类成员变量可能与全局变量同名，我们以"."来区分。而本质上，编译器在进行编译时，与函数的处理相似，也为类中的变量取了一个独一无二的名字，这个名字与用户程序中同名的全局变量名字不同。

未加 `extern "C"` 声明时的连接方式

假设在 C++ 中，模块 A 的头文件如下：

```
// 模块 A 头文件 moduleA.h
#ifndef MODULE_A_H
#define MODULE_A_H
int foo( int x, int y );
#endif
```

在模块 B 中引用该函数：

```
// 模块 B 实现文件 moduleB.cpp
#include "moduleA.h"
foo(2,3);
```

实际上，在连接阶段，连接器会从模块 A 生成的目标文件 `moduleA.obj` 中寻找 `_foo_int_int` 这样的符号！

加 `extern "C"` 声明后的编译和连接方式

加 `extern "C"` 声明后，模块 A 的头文件变为：

```
// 模块 A 头文件 moduleA.h
#ifndef MODULE_A_H
#define MODULE_A_H
extern "C" int foo( int x, int y );
#endif
```

在模块 B 的实现文件中仍然调用 `foo( 2,3 )`，其结果是：

- (1) 模块 A 编译生成 foo 的目标代码时，没有对其名字进行特殊处理，采用了 C 语言的方式
- (2) 连接器在为模块 B 的目标代码寻找 `foo(2,3)` 调用时，寻找的是未经修改的符号名 `_foo`。

如果在模块 A 中函数声明了 `foo` 为 `extern "C"` 类型，而模块 B 中包含的是 `extern int foo( int x, int y )`，则模块 B 找不到模块 A 中的函数；反之亦然。

所以，可以用一句话概括 `extern "C"` 这个声明的真实目的（任何语言中的任何语法特性的诞生都不是随意而为的，来源于真实世界的需求驱动。我们在思考问题时，不能只停留在这个语言是怎么做的，还要问一问它为什么要这么做，动机是什么，这样我们可以更深入地理解许多问题）；实现 C++ 与 C 及其它语言的混合编程。

明白了 C++ 中 `extern "C"` 的设立动机，我们下面来具体分析 `extern "C"` 通常的使用技巧。

`extern "C"` 的惯用法

(1) 在 C++ 中引用 C 语言中的函数和变量，在包含 C 语言头文件（假设为 `cExample.h`）时，需进行下列处理

```
extern "C"  
{  
#include "cExample.h"  
}
```

而在 C 语言的头文件中，对其外部函数只能指定为 `extern` 类型，C 语言中不支持 `extern "C"` 声明，在 `.c` 文件中包含了 `extern "C"` 时会出现编译语法错误。

C++ 引用 C 函数例子工程中包含的三个文件的源代码如下：

```
/* c 语言头文件: cExample.h */  
#ifndef C_EXAMPLE_H  
#define C_EXAMPLE_H  
extern int add(int x,int y);  
#endif  
  
/* c 语言实现文件: cExample.c */  
#include "cExample.h"  
int add( int x, int y )  
{  
    return x + y;  
}  
  
// c++实现文件, 调用 add: cppFile.cpp  
extern "C"  
{  
#include "cExample.h"  
}  
int main(int argc, char* argv[])  
{
```

```
add(2,3);
return 0;
}
```

如果C++调用一个C语言编写的.DLL时,当包括.DLL的头文件或声明接口函数时,应加**extern "C" { }**。

(2) 在C中引用C++语言中的函数和变量时, C++的头文件需添加**extern "C"**, 但是在C语言中不能直接引用声明了**extern "C"**的该头文件, 应该仅将C文件中将C++中定义的**extern "C"**函数声明为**extern**类型。

C引用C++函数例子工程中包含的三个文件的源代码如下:

```
//C++头文件 cppExample.h
#ifndef CPP_EXAMPLE_H
#define CPP_EXAMPLE_H
extern "C" int add( int x, int y );
#endif

//C++实现文件 cppExample.cpp
#include "cppExample.h"
int add( int x, int y )
{
    return x + y;
}

/* C实现文件 cFile.c
/* 这样会编译出错: #include "cExample.h" */
extern int add( int x, int y );
int main( int argc, char* argv[] )
{
    add( 2, 3 );
    return 0;
}
```

15 题目的解答请参考《C++中**extern "C"**含义深层探索》注解:

**16. 关联、聚合(Aggregation)以及组合(Composition)的区别?**

涉及到 UML 中的一些概念：关联是表示两个类的一般性联系，比如“学生”和“老师”就是一种关联关系；聚合表示 **has-a** 的关系，是一种相对松散的关系，聚合类不需要对被聚合类负责，如下图所示，用空的菱形表示聚合关系：

从实现的角度讲，聚合可以表示为：

```
class A {...} class B { A* a; .....}
```

而组合表示 **contains-a** 的关系，关联性强于聚合：组合类与被组合类有相同的生命周期，组合类要对被组合类负责，采用实心的菱形表示组合关系：

实现的形式是：

```
class A{...} class B{ A a; ...}
```

参考文章：<http://blog.csdn.net/wfwd/archive/2006/05/30/763753.aspx>

<http://blog.csdn.net/wfwd/archive/2006/05/30/763760.aspx>

## 17. 面向对象的三个基本特征，并简单叙述之？

1. 封装：将客观事物抽象成类，每个类对自身的数据和方法实行 **protection(private, protected, public)**
2. 继承：广义的继承有三种实现形式：实现继承（指使用基类的属性和方法而无需额外编码的能力）、可视继承（子窗体使用父窗体的外观和实现代码）、接口继承（仅使用属性和方法，实现滞后到子类实现）。前两种（类继承）和后一种（对象组合=>接口继承以及纯虚函数）构成了功能复用的两种方式。
3. 多态：是将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：允许将子类类型的指针赋值给父类类型的指针。

## 18. 重载（**overload**）和重写（**overried**，有的书也叫做“覆盖”）的区别？

常考的题目。从定义上来说：

重载: 是指允许存在多个同名函数, 而这些函数的参数表不同 (或许参数个数不同, 或许参数类型不同, 或许两者都不同)。

重写: 是指子类重新定义父类虚函数的方法。

从实现原理上来说:

重载: 编译器根据函数不同的参数表, 对同名函数的名称做修饰, 然后这些同名函数就成了不同的函数 (至少对于编译器来说是这样的)。如, 有两个同名函数: `function func(p:integer):integer;` 和 `function func(p:string):integer;`。那么编译器做过修饰后的函数名称可能是这样的: `int_func`、`str_func`。对于这两个函数的调用, 在编译器间就已经确定了, 是静态的。也就是说, 它们的地址在编译期就绑定了 (早绑定), 因此, 重载和多态无关!

重写: 和多态真正相关。当子类重新定义了父类的虚函数后, 父类指针根据赋给它的不同的子类指针, 动态的调用属于子类的该函数, 这样的函数调用在编译期间是无法确定的 (调用的子类的虚函数的地址无法给出)。因此, 这样的函数地址是在运行期绑定的 (晚绑定)。

## 19. 多态的作用?

主要是两个: 1. 隐藏实现细节, 使得代码能够模块化; 扩展代码模块, 实现代码重用; 2. 接口重用: 为了类在继承和派生的时候, 保证使用家族中任一类的实例的某一属性时的正确调用。

## 20. Ado 与 Ado.net 的相同与不同?

除了“能够让应用程序处理存储于 DBMS 中的数据”这一基本相似点外, 两者没有太多共同之处。但是 Ado 使用 OLE DB 接口并基于微软的 COM 技术, 而 ADO.NET 拥有自己的 ADO.NET 接口并且基于微软的.NET 体系架构。众所周知.NET 体系不同于 COM 体系, ADO.NET 接口也就完全不同与 ADO 和 OLE DB 接口, 这也就是说 ADO.NET 和 ADO 是两种数据访问方式。ADO.net 提供对 XML 的支持。

## 21. New delete 与 malloc free 的联系与区别?

答案: 都是在堆(heap)上进行动态的内存操作。用 `malloc` 函数需要指定内存分配的字节数并且不能初始化对象, `new` 会自动调用对象的构造函数。`delete` 会调用对象的 `destructor`, 而 `free` 不会调用对象的 `destructor`。

## 22. #define DOUBLE(x) x+x , i = 5\*DOUBLE(5); i 是多少?

答案: `i` 为 30。

## 23. 有哪几种情况只能用 **initialization list** 而不能用 **assignment**?

答案: 当类中含有 `const`、`reference` 成员变量; 基类的构造函数都需要初始化表。

## 24. C++是不是类型安全的?

答案: 不是。两个不同类型的指针之间可以强制转换(用 `reinterpret cast`)。C#是类型安全的。

## 25. **main** 函数执行以前, 还会执行什么代码?

答案: 全局对象的构造函数会在 `main` 函数之前执行。

## 26. 描述内存分配方式以及它们的区别?

- 1) 从静态存储区域分配。内存在程序编译的时候就已经分配好, 这块内存在程序的整个运行期间都存在。例如全局变量, `static` 变量。
- 2) 在栈上创建。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。栈内存分配运算内置于处理器的指令集。
- 3) 从堆上分配, 亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存, 程序员自己负责在何时用 `free` 或 `delete` 释放内存。动态内存的生存期由程序员决定, 使用非常灵活, 但问题也最多。

## 27. **struct** 和 **class** 的区别

答案: `struct` 的成员默认是公有的, 而类的成员默认是私有的。`struct` 和 `class` 在其他方面是功能相当的。

从感情上讲, 大多数的开发者感到类和结构有很大的差别。感觉上结构仅仅象一堆缺乏封装和功能的开放的内存位, 而类就象活的并且可靠的社会成员, 它有智能服务, 有牢固的封装屏障和一个良好定义的接口。既然大多数人都这么认为, 那么只有在你的类有很少的方法并且有公有数据(这种事情在良好设计的系统中是存在的!)时, 你也许应该使用 `struct` 关键字, 否则, 你应该使用 `class` 关键字。

## 28. 当一个类 **A** 中没有生命任何成员变量与成员函数, 这时 **sizeof(A)** 的值是多少, 如果不是零, 请解释一下编译器为什么没有让它为零。 (Autodesk)

答案: 肯定不是零。举个反例, 如果是零的话, 声明一个 `class A[10]` 对象数组, 而每一个对象占用的空间是零, 这时就没办法区分 `A[0], A[1]...` 了。

## 29. 在 **8086** 汇编下, 逻辑地址和物理地址是怎样转换的? (Intel)

答案：通用寄存器给出的地址，是段内偏移地址，相应段寄存器地址\*10H+通用寄存器内地址，就得到了真正要访问的地址。

### 30. 比较 C++ 中的 4 种类型转换方式？

请参考：<http://blog.csdn.net/wfwd/archive/2006/05/30/763785.aspx>，重点是 static\_cast, dynamic\_cast 和 reinterpret\_cast 的区别和应用。

### 31. 分别写出 **BOOL, int, float, 指针** 类型的变量 a 与“零”的比较语句。

答案：

```
BOOL : if ( !a ) or if(a)
int : if ( a == 0 )
float : const EXPRESSION EXP = 0.000001
if ( a < EXP && a >-EXP )
pointer : if ( a != NULL) or if(a == NULL)
```

### 32. 请说出 **const** 与 **#define** 相比，有何优点？

答案：1) **const** 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意想不到的错误。

2) 有些集成化的调试工具可以对 **const** 常量进行调试，但是不能对宏常量进行调试。

### 33. 简述数组与指针的区别？

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。

(1) 修改内容上的差别

```
char a[] = "hello";
a[0] = 'X';
char *p = "world"; // 注意 p 指向常量字符串
p[0] = 'X'; // 编译器不能发现该错误，运行时错误
```

(2) 用运算符 **sizeof** 可以计算出数组的容量（字节数）。**sizeof(p)**, p 为指针得到的是一个指针变量的字节数，而不是 p 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
char a[] = "hello world";
char *p = a;
cout << sizeof(a) << endl; // 12 字节
cout << sizeof(p) << endl; // 4 字节
计算数组和指针的内存容量
void Func(char a[100])
{
    cout << sizeof(a) << endl; // 4 字节而不是 100 字节
```

}

### 34. 类成员函数的重载、覆盖和隐藏区别?

答案:

a. 成员函数被重载的特征:

- (1) 相同的范围 (在同一个类中);
- (2) 函数名字相同;
- (3) 参数不同;
- (4) **virtual** 关键字可有可无。

b. 覆盖是指派生类函数覆盖基类函数, 特征是:

- (1) 不同的范围 (分别位于派生类与基类);
- (2) 函数名字相同;
- (3) 参数相同;
- (4) 基类函数必须有 **virtual** 关键字。

c. “隐藏”是指派生类的函数屏蔽了与其同名的基类函数, 规则如下:

- (1) 如果派生类的函数与基类的函数同名, 但是参数不同。此时, 不论有无 **virtual** 关键字, 基类的函数将被隐藏 (注意别与重载混淆)。
- (2) 如果派生类的函数与基类的函数同名, 并且参数也相同, 但是基类函数没有 **virtual** 关键字。此时, 基类的函数被隐藏 (注意别与覆盖混淆)

### 35. There are two int variables: a and b, don't use "if", "? :", "switch" or other judgement statements, find out the biggest one of the two numbers.

答案:  $((a + b) + \text{abs}(a - b)) / 2$

### 36. 如何打印出当前源文件的文件名以及源文件的当前行号?

答案:

```
cout << __FILE__ ;  
cout << __LINE__ ;
```

**\_\_FILE\_\_** 和 **\_\_LINE\_\_** 是系统预定义宏, 这种宏并不是在某个文件中定义的, 而是由编译器定义的。

### 37. **main** 主函数执行完毕后, 是否可能会再执行一段代码, 给出说明?

答案: 可以, 可以用 **\_onexit** 注册一个函数, 它会在 **main** 之后执行 **int fn1(void)**, **fn2(void)**,

**fn3(void)**, **fn4 (void)**;

```
void main( void )
```

```
{
```

```
String str("zhanglin");
```

```
_onexit( fn1 );
```

```
_onexit( fn2 );
```

```
_onexit( fn3 );
```

```
_onexit( fn4 );
```

```
printf( "This is executed first.\n" );
```

```
}
```

```
int fn1()
{
printf( "next.\n" );
return 0;
}
int fn2()
{
printf( "executed " );
return 0;
}
int fn3()
{
printf( "is " );
return 0;
}
int fn4()
{
printf( "This " );
return 0;
}
```

The `_onexit` function is passed the address of a function (`func`) to be called when the program terminates normally. Successive calls to `_onexit` create a register of functions that are executed in LIFO (last-in-first-out) order. The functions passed to `_onexit` cannot take parameters.

### 38. 如何判断一段程序是由 C 编译程序还是由 C++ 编译程序编译的?

答案:

```
#ifdef __cplusplus
cout<<"c++";
#else
cout<<"c";
#endif
```

### 39. 文件中有一组整数, 要求排序后输出到另一个文件中

答案:

```
#include<iostream>

#include<fstream>

using namespace std;

void Order(vector<int>& data) //bubble sort
```

```

{
int count = data.size() ;
int tag = false ; // 设置是否需要继续冒泡的标志位
for ( int i = 0 ; i < count ; i++)
{
for ( int j = 0 ; j < count - i - 1 ; j++)
{
if ( data[j] > data[j+1])
{
tag = true ;
int temp = data[j] ;
data[j] = data[j+1] ;
data[j+1] = temp ;
}
}
if ( !tag )
break ;
}
}

```

```

void main( void )
{
vector<int>data;
ifstream in("c:\\data.txt");
if ( !in)
{
cout<<"file error!";
exit(1);
}
int temp;
while (!in.eof())
{
in>>temp;
data.push_back(temp);
}
in.close(); //关闭输入文件流
Order(data);
ofstream out("c:\\result.txt");
if ( !out)
{
cout<<"file error!";
exit(1);
}

```

```

for ( i = 0 ; i < data.size() ; i++)
out<<data[i]<<" ";
out.close(); //关闭输出文件流
}

```

#### 40. 链表题：一个链表的结点结构

```

struct Node
{
int data ;
Node *next ;
};
typedef struct Node Node ;

```

(1)已知链表的头结点 `head`,写一个函数把这个链表逆序 ( Intel)

```

Node * ReverseList(Node *head) //链表逆序
{
if ( head == NULL || head->next == NULL )
return head;
Node *p1 = head ;
Node *p2 = p1->next ;
Node *p3 = p2->next ;
p1->next = NULL ;
while ( p3 != NULL )
{
p2->next = p1 ;
p1 = p2 ;
p2 = p3 ;
p3 = p3->next ;
}
p2->next = p1 ;
head = p2 ;
return head ;
}

```

(2)已知两个链表 `head1` 和 `head2` 各自有序, 请把它们合并成一个链表依然有序。(保留所有结点, 即便大小相同)

```

Node * Merge(Node *head1 , Node *head2)
{
if ( head1 == NULL)
return head2 ;
if ( head2 == NULL)

```

```

return head1 ;
Node *head = NULL ;
Node *p1 = NULL;
Node *p2 = NULL;
if ( head1->data < head2->data )
{
head = head1 ;
p1 = head1->next;
p2 = head2 ;
}
else
{
head = head2 ;
p2 = head2->next ;
p1 = head1 ;
}
Node *pcurrent = head ;
while ( p1 != NULL && p2 != NULL)
{
if ( p1->data <= p2->data )
{
pcurrent->next = p1 ;
pcurrent = p1 ;
p1 = p1->next ;
}
else
{
pcurrent->next = p2 ;
pcurrent = p2 ;
p2 = p2->next ;
}
}
if ( p1 != NULL )
pcurrent->next = p1 ;
if ( p2 != NULL )
pcurrent->next = p2 ;
return head ;
}

(3)已知两个链表 head1 和 head2 各自有序, 请把它们合并成一个链表依然有序, 这次要求
用递归方法进行。 (Autodesk)
答案:

```

```

Node * MergeRecursive(Node *head1 , Node *head2)
{
if ( head1 == NULL )

```

```

return head2 ;
if ( head2 == NULL)
return head1 ;
Node *head = NULL ;
if ( head1->data < head2->data )
{
head = head1 ;
head->next = MergeRecursive(head1->next,head2);
}
else
{
head = head2 ;
head->next = MergeRecursive(head1,head2->next);
}
return head ;
}

```

#### 4.1. 分析一下这段程序的输出 (Autodesk)

```

class B
{
public:
B()
{
cout<<"default constructor"<<endl;
}
~B()
{
cout<<"destructed"<<endl;
}
B(int i):data(i) //B(int) works as a converter ( int -> instance of B)
{
cout<<"constructed by parameter " << data <<endl;
}
private:
int data;
};

```

```

B Play( B b)
{
return b ;
}

```

(1) results:

```
int main(int argc, char* argv[]) constructed by parameter 5
{ destructed B(5)形参析构
B t1 = Play(5); B t2 = Play(t1);    destructed t1 形参析构
return 0;                           destructed t2 注意顺序!
} destructed t1
```

(2) results:

```
int main(int argc, char* argv[]) constructed by parameter 5
{ destructed B(5)形参析构
B t1 = Play(5); B t2 = Play(10);    constructed by parameter 10
return 0;                           destructed B(10)形参析构
} destructed t2 注意顺序!
```

destructed t1

#### 42. 写一个函数找出一个整数数组中，第二大的数 (Microsoft)

答案:

```
const int MINNUMBER = -32767 ;
int find_sec_max( int data[ ] , int count)
{
int maxnumber = data[0] ;
int sec_max = MINNUMBER ;
for ( int i = 1 ; i < count ; i++)
{
if ( data[i] > maxnumber )
{
sec_max = maxnumber ;
maxnumber = data[i] ;
}
else
{
if ( data[i] > sec_max )
sec_max = data[i] ;
}
}
return sec_max ;
}
```

#### 43. 写一个在一个字符串(n)中寻找一个子串(m)第一个位置的函数。

KMP 算法效率最好，时间复杂度是  $O(n+m)$ 。

#### 44. 多重继承的内存分配问题：

比如有 class A : public class B, public class C {}

那么 A 的内存结构大致是怎么样的?

这个是 **compiler-dependent** 的, 不同的实现其细节可能不同。  
如果不考虑有虚函数、虚继承的话就相当简单; 否则的话, 相当复杂。  
可以参考《深入探索 C++ 对象模型》, 或者:  
<http://blog.csdn.net/wfwd/archive/2006/05/30/763797.aspx>

#### 45. 如何判断一个单链表是有环的? (注意不能用标志位, 最多只能用两个额外指针)

```
struct node { char val; node* next; }

bool check(const node* head) {} //return false : 无环; true: 有环
```

一种  $O(n)$  的办法就是 (搞两个指针, 一个每次递增一步, 一个每次递增两步, 如果有环的话两者必然重合, 反之亦然):

```
bool check(const node* head)
{
    if(head==NULL) return false;
    node *low=head, *fast=head->next;
    while(fast!=NULL && fast->next!=NULL)
    {
        low=low->next;
        fast=fast->next->next;
        if(low==fast) return true;
    }
    return false;
}
```

经历了很多次笔试, 先把部分题目提供或者总结出来:

1. #include <aaa.h> 和 #include "ccc.h", 中 <> "" 的区别
2. sizeof 的应用, 特别是针对一个类, 一个结构的 sizeof, 还有结构的对齐方式 (就是一个 byte 按照四个 byte 处理)
3. const 应用
4. strcpy 的应用, 例如为什么这个函数有返回值, 自己再编一个这样的函数
5. 一些运算符的重载, 我被 ++class 和 class++ 难住了, 平时没注意
6. dll 的两种用法
7. sendmessage 与 postmessage 区别

1. 是不是一个父类写了一个 virtual 函数, 如果子类覆盖它的函数不加 virtual, 也能实现多态?

virtual 修饰符会被隐形继承的。private 也被集成, 只事派生类没有访问权限而已。virtual 可加可不加。子类的空间里有父类的所有变量 (static 除外)。同一个函数只存在一个实体 (inline)

除外)。子类覆盖它的函数不加 `virtual` ,也能实现多态。在子类的空间里,有父类的私有变量。私有变量不能直接访问。

---

2.输入一个字符串,将其逆序后输出。(使用 C++, 不建议用伪码)

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

void main()
{
    char a[50];memset(a,0,sizeof(a));
    int i=0,j;
    char t;
    cin.getline(a,50,'\n');
    for(i=0,j=strlen(a)-1;i<strlen(a)/2;i++,j--)
    {
        t=a[i];
        a[i]=a[j];
        a[j]=t;
    }
    cout<<a<<endl;
}

//第二种

string str;
cin>>str;
str.replace;
cout<<str;
```

---

3.请简单描述 Windows 内存管理的方法。

内存管理是操作系统中的重要部分,两三句话恐怕谁也说不清楚吧~~~  
我先说个大概,希望能够抛砖引玉吧

当程序运行时需要从内存中读出这段程序的代码。代码的位置必须在物理内存中才能被运行,由于现在的操作系统中有非常多的程序运行着,内存中不能够完全放下,所以引出了虚拟内存的概念。把哪些不常用的程序片断就放入虚拟内存,当需要用到它的时候在 `load` 入主存(物理内存)中。这个就是内存管理所要做的事。内存管理还有另外一件事需要做:计算程序片段在主存中的物理位置,以便 CPU 调度。

内存管理有块式管理，页式管理，段式和段页式管理。现在常用段页式管理

块式管理：把主存分为一大块、一大块的，当所需的程序片断不在主存时就分配一块主存空间，把程序片断 **load** 入主存，就算所需的程序片度只有几个字节也只能把这一块分配给它。这样会造成很大的浪费，平均浪费了 50% 的内存空间，但易于管理。

页式管理：把主存分为一页一页的，每一页的空间要比一块一块的空间小很多，显然这种方法的空间利用率要比块式管理高很多。

段式管理：把主存分为一段一段的，每一段的空间又要比一页一页的空间小很多，这种方法在空间利用率上又比页式管理高很多，但是也有另外一个缺点。一个程序片断可能会被分为几十段，这样很多时间就会被浪费在计算每一段的物理地址上(计算机最耗时间的大家都知道是 **I/O** 吧)。

段页式管理：结合了段式管理和页式管理的优点。把主存分为若干页，每一页又分为若干段。好处就很明显，不用我多说了吧。

各种内存管理都有它自己的方法来计算出程序片断在主存中的物理地址，其实都很相似。

这只是一个大概而已，不足以说明内存管理的皮毛。无论哪一本操作系统书上都有详细的讲解

---

4.

```
#include "stdafx.h"
#define SQR(X) X*X

int main(int argc, char* argv[])
{
    int a = 10;
    int k = 2;
    int m = 1;

    a /= SQR(k+m)/SQR(k+m);
    printf("%d\n",a);

    return 0;
}
```

这道题目的结果是什么啊？

**define** 只是定义而已，在编译时只是简单代换 **X\*X** 而已，并不经过算术法则的

```
a /= (k+m)*(k+m)/(k+m)*(k+m);
=>a /= (k+m)*1*(k+m);
=>a = a/9;
=>a = 1;
```

---

5.

const 符号常量;

(1) const char \*p

(2) char const \*p

(3) char \* const p

说明上面三种描述的区别;

如果 **const** 位于星号的左侧, 则 **const** 就是用来修饰指针所指向的变量, 即指针指向为常量;

如果 **const** 位于星号的右侧, **const** 就是修饰指针本身, 即指针本身是常量。

(1) const char \*p

一个指向 **char** 类型的 **const** 对象指针, **p** 不是常量, 我们可以修改 **p** 的值, 使其指向不同的 **char**, 但是不能改变它指向非 **char** 对象, 如:

```
const char *p;
char c1='a';
char c2='b';
p=&c1;//ok
p=&c2;//ok
*p=c1;//error
```

(2) char const \*p

(3) char \* const p

这两个好象是一样的, 此时 **\*p** 可以修改, 而 **p** 不能修改。

(4) const char \* const p

这种是地址及指向对象都不能修改。

---

6. 下面是 C 语言中两种 **if** 语句判断方式。请问哪种写法更好? 为什么?

```
int n;
if (n == 10) // 第一种判断方式
if (10 == n) // 第二种判断方式
```

如果少了个=号, 编译时就会报错, 减少了出错的可能行, 可以检测出是否少了=

---

7. 下面的代码有什么问题?

```
void DoSomeThing(...)
```

```
{
```

```
char* p;
```

```
...
```

```
p = malloc(1024); // 分配 1K 的空间
if (NULL == p)
    return;
...
p = realloc(p, 2048); // 空间不够, 重新分配到 2K
if (NULL == p)
    return;
...
}
```

A:

`p = malloc(1024);` 应该写成: `p = (char *) malloc(1024);`  
没有释放 `p` 的空间, 造成内存泄漏。

---

8.下面的代码有什么问题? 并请给出正确的写法。

```
void DoSomeThing(char* p)
{
    char str[16];
    int n;
    assert(NULL != p);
    sscanf(p, "%s%d", str, n);
    if (0 == strcmp(str, "something"))
    {
        ...
    }
}
```

A:

`sscanf(p, "%s%d", str, n);` 这句该写成: `sscanf(p, "%s%d", str, &n);`

---

9.下面代码有什么错误?

```
Void test1()
{
    char string[10];
    char *str1="0123456789";
    strcpy(string, str1);
}
```

数组越界

---

10.下面代码有什么问题?

```
Void test2()
{
    char string[10], str1[10];
    for(i=0; i<10;i++)
    {
        str1[i] ='a';
    }
    strcpy(string, str1);
}
```

数组越界

---

11.下面代码有什么问题?

```
Void test3(char* str1)
{
    char string[10];
    if(strlen(str1)<=10)
    {
        strcpy(string, str1);
    }
}
```

==数组越界

==`strcpy` 拷贝的结束标志是查找字符串中的\0 因此如果字符串中没有遇到\0 的话 会一直复制, 直到遇到\0,上面的 123 都因此产生越界的情况

建议使用 `strncpy` 和 `memcpy`

---

12.下面代码有什么问题?

```
#define MAX_SRM 256
```

```
DSN get_SRM_no()
{
    static int SRM_no; //是不是这里没赋初值?
    int I;
    for(I=0;I<MAX_SRM;I++,SRM_no++)
    {
        SRM_no %= MAX_SRM;
        if(MY_SRM.state==IDLE)
        {
            break;
        }
    }
}
```

```
    }
    if(I>=MAX_SRM)
        return (NULL_SRM);
    else
        return SRM_no;
}
```

系统会初始化 **static int** 变量为 0, 但该值会一直保存, 所谓的不可重入...

---

13. 写出运行结果:

```
{// test1
    char str[] = "world"; cout << sizeof(str) << ": ";
    char *p    = str;    cout << sizeof(p) << ": ";
    char i     = 10;    cout << sizeof(i) << ": ";
    void *pp   = malloc(10); cout << sizeof(p) << endl;
}
```

6: 4: 1: 4

---

14. 写出运行结果:

```
{// test2
    union V {
        struct X {
            unsigned char s1:2;
            unsigned char s2:3;
            unsigned char s3:3;
        } x;
        unsigned char c;
    } v;
    v.c = 100;
    printf("%d", v.x.s3);
}
```

3

---

15. 用 C++ 写个程序, 如何判断一个操作系统是 16 位还是 32 位的? 不能用 **sizeof()** 函数

A1:

16 位的系统下,

```
int i = 65536;
cout << i; // 输出 0;
int i = 65535;
cout << i; // 输出-1;
```

32 位的系统下，

```
int i = 65536;
cout << i; // 输出 65536;
int i = 65535;
cout << i; // 输出 65535;
```

A2:

```
int a = ~0;
if( a>65536 )
{
    cout<<"32 bit"<<endl;
}
else
{
    cout<<"16 bit"<<endl;
}
```

---

## 16. C 和 C++有什么不同？

从机制上：C 是面向过程的（但 C 也可以编写面向对象的程序）；C++是面向对象的，提供了类。  
但是，

C++编写面向对象的程序比 C 容易

从适用的方向：C 适合要求代码体积小的，效率高的场合，如嵌入式；C++适合更上层的，复杂的；Linux 核心大部分是 C 写的，因为它是系统软件，效率要求极高。

从名称上也可以看出，C++比 C 多了++，说明 C++是 C 的超集；那为什么不叫 C++而叫 C++呢？  
是因为 C++比 C 来说扩充的东西太多了，所以就在 C 后面放上两个++；于是就成了 C++

C 语言是结构化编程语言，C++是面向对象编程语言。

C++侧重于对象而不是过程，侧重于类的设计而不是逻辑的设计。

---

## 17. 在不用第三方参数的情况下，交换两个参数的值

```
#include <stdio.h>
```

```
void main()
{
    int i=60;
    int j=50;
    i=i+j;
    j=i-j;
    i=i-j;
    printf("i=%d\n",i);
    printf("j=%d\n",j);
}
```

方法二：

```
i^=j;
j^=i;
i^=j;
```

方法三：

```
// 用加减实现，而且不会溢出
a = a+b-(b=a)
```

---

18.有关位域的面试题（为什么输出的是一个奇怪的字符）

a.t = 'b';效果相当于 a.t= 'b' & 0xf;

'b' --> 01100010  
'b' & 0xf -->00000010  
所以输出 Ascii 码为 2 的特殊字符

char t:4;就是 4bit 的字符变量，同样  
unsigned short i:8;就是 8bit 的无符号短整形变量

---

19.int i=10, j=10, k=3; k\*=i+j; k最后的值是?

60

---

20.进程间通信的方式有?

进程间通信的方式有 共享内存， 管道， Socket， 消息队列， DDE 等

---

21.

```

struct A
{
char t:4;
char k:4;
unsigned short i:8;
unsigned long m;
}
sizeof(A)=? (不考虑边界对齐)

```

7

```

struct CELL          // Declare CELL bit field
{
    unsigned character : 8; // 00000000 ???????
    unsigned foreground : 3; // 00000??? 00000000
    unsigned intensity : 1; // 0000?000 00000000
    unsigned background : 3; // 0???0000 00000000
    unsigned blink : 1; // ?0000000 00000000
} screen[25][80]; // Array of bit fields

```

## 二、位结构

位结构是一种特殊的结构，在需按位访问一个字节或字的多个位时，位结构比按位运算符更加方便。

位结构定义的一般形式为：

```

struct 位结构名{
    数据类型 变量名: 整型常数;
    数据类型 变量名: 整型常数;
} 位结构变量;

```

其中：数据类型必须是 `int(unsigned 或 signed)`。整型常数必须是非负的整数，范围是 `0~15`，表示二进制位的个数，即表示有多少位。

变量名是选择项，可以不命名，这样规定是为了排列需要。

例如：下面定义了一个位结构。

```

struct{
    unsigned incon: 8; /*incon 占用低字节的 0~7 共 8 位*/
    unsigned txcolor: 4; /*txcolor 占用高字节的 0~3 位共 4 位*/
    unsigned bgcolor: 3; /*bgcolor 占用高字节的 4~6 位共 3 位*/
    unsigned blink: 1; /*blink 占用高字节的第 7 位*/
}ch;

```

位结构成员的访问与结构成员的访问相同。

例如：访问上例位结构中的 `bgcolor` 成员可写成：

`ch.bgcolor`

注意：

1. 位结构中的成员可以定义为 `unsigned`，也可定义为 `signed`，但当成员长度为 1 时，会被认为是 `unsigned` 类型。因为单个位不可能具有符号。
2. 位结构中的成员不能使用数组和指针，但位结构变量可以是数组和指针，

如果是指针，其成员访问方式同结构指针。

3. 位结构总长度(位数)，是各个位成员定义的位数之和，可以超过两个字节。

4. 位结构成员可以与其它结构成员一起使用。

例如：

```
struct info{  
    char name[8];  
    int age;  
    struct addr address;  
    float pay;  
    unsigned state: 1;  
    unsigned pay: 1;  
}workers;
```

上例的结构定义了关于一个工人的信息。其中有两个位结构成员，每个位结构成员只有一位，因此只占一个字节但保存了两个信息，该字节中第一位表示工人的状态，第二位表示工资是否已发放。由此可见使用位结构可以节省存储空间。

---

22. 下面的函数实现在一个固定的数上加上一个数，有什么错误，改正

```
int add_n(int n)  
{  
    static int i=100;  
    i+=n;  
    return i;  
}
```

答：

因为 **static** 使得 **i** 的值会保留上次的值。

去掉 **static** 就可了

---

23. 下面的代码有什么问题？

```
class A  
{  
public:  
    A() { p=this; }  
    ~A() { if(p!=NULL) { delete p; p=NULL; } }  
  
    A* p;  
};
```

答：

会引起无限递归

---

```
24.
union a {
    int a_int1;
    double a_double;
    int a_int2;
};
```

```
typedef struct
```

```
{  
    a a1;  
    char y;  
} b;
```

```
class c
```

```
{  
    double c_double;  
    b b1;  
    a a2;
```

```
};
```

输出 `cout<<sizeof(c)<<endl;` 的结果?

答:

VC6 环境下得出的结果是 32

另:

我(sun)在 VC6.0+win2k 下做过试验:

short - 2

int-4

float-4

double-8

指针-4

`sizeof(union)`, 以结构里面 size 最大的为 `union` 的 size

---

25. i 最后等于多少?

```
int i = 1;  
int j = i++;  
if((i>j++) && (i++ == j)) i+=j;
```

答:

i = 5

---

26.

```
unsigned short array[]={1,2,3,4,5,6,7};  
int i = 3;  
*(array + i) = ?
```

答:

4

---

27.

```
class A  
{  
    virtual void func1();  
    void func2();  
}  
Class B: class A  
{
```

```
    void func1(){cout << "fun1 in class B" << endl;}  
    virtual void func2(){cout << "fun2 in class B" << endl;}
```

}

A, A 中的 func1 和 B 中的 func2 都是虚函数.

B, A 中的 func1 和 B 中的 func2 都不是虚函数.

C, A 中的 func2 是虚函数., B 中的 func1 不是虚函数.

D, A 中的 func2 不是虚函数, B 中的 func1 是虚函数.

答:

A

---

28.

数据库: 抽出部门, 平均工资, 要求按部门的字符串顺序排序, 不能含有"human resource"部门,

employee 结构如下: employee\_id, employee\_name, depart\_id, depart\_name, wage

答:

```
select depart_name, avg(wage)  
from employee  
where depart_name <> 'human resource'  
group by depart_name  
order by depart_name
```

---

29.

给定如下 SQL 数据库: `Test(num INT(4))` 请用一条 SQL 语句返回 `num` 的最小值, 但不许使用统计功能, 如 `MIN`, `MAX` 等

答:

```
select top 1 num
from Test
order by num desc
```

---

30.

输出下面程序结果。

```
#include <iostream.h>

class A
{
public:
    virtual void print(void)
    {
        cout<<"A::print()"<<endl;
    }
};

class B:public A
{
public:
    virtual void print(void)
    {
        cout<<"B::print()"<<endl;
    }
};

class C:public B
{
public:
    virtual void print(void)
    {
        cout<<"C::print()"<<endl;
    }
};

void print(A a)
{
    a.print();
}

void main(void)
```

```

{
    A a, *pa,*pb,*pc;
    B b;
    C c;

    pa=&a;
    pb=&b;
    pc=&c;

    a.print();
    b.print();
    c.print();

    pa->print();
    pb->print();
    pc->print();

    print(a);
    print(b);
    print(c);
}

```

A:  
A::print()  
B::print()  
C::print()  
A::print()  
B::print()  
C::print()  
A::print()  
A::print()  
A::print()

---

### 31.

试编写函数判断计算机的字节存储顺序是开序(little endian)还是降序(bigendian)

答:

```

bool IsBigendian()
{
    unsigned short usData = 0x1122;
    unsigned char *pucData = (unsigned char*)&usData;

    return (*pucData == 0x22);
}

```

---

### 32. 简述 Critical Section 和 Mutex 的不同点

答：

对几种同步对象的总结

#### 1. Critical Section

- A. 速度快
- B. 不能用于不同进程
- C. 不能进行资源统计(每次只可以有一个线程对共享资源进行存取)

#### 2. Mutex

- A. 速度慢
- B. 可用于不同进程
- C. 不能进行资源统计

#### 3. Semaphore

- A. 速度慢
- B. 可用于不同进程
- C. 可进行资源统计(可以让一个或超过一个线程对共享资源进行存取)

#### 4. Event

- A. 速度慢
- B. 可用于不同进程
- C. 可进行资源统计

---

### 33. 一个数据库中有两个表：

一张表为 Customer, 含字段 ID, Name;

一张表为 Order, 含字段 ID, CustomerID (连向 Customer 中 ID 的外键), Revenue;

写出求每个 Customer 的 Revenue 总和的 SQL 语句。

建表

```
create table customer
(
  ID int primary key, Name char(10)
)
```

go

```
create table [order]
(
  ID int primary key, CustomerID int foreign key references customer(id), Revenue float
)
```

```
go

--查询
select Customer.ID, sum( isnul([Order].Revenue,0) )
from customer full join [order]
on( [order].customerid=customer.id )
group by customer.id
```

---

34.请指出下列程序中的错误并且修改

```
void GetMemory(char *p){
    p=(char *)malloc(100);
}
void Test(void){
    char *str=NULL;
    GetMemory=(str);
    strcpy(str,"hello world");
    printf(str);
}
```

A:错误--参数的值改变后，不会传回

GetMemory 并不能传递动态内存，Test 函数中的 str 一直都是 NULL。  
strcpy(str, "hello world"); 将使程序崩溃。

修改如下：

```
char *GetMemory(){
    char *p=(char *)malloc(100);
    return p;
}
void Test(void){
    char *str=NULL;
    str=GetMemory();
    strcpy(str,"hello world");
    printf(str);
}
```

方法二：void GetMemory2(char \*\*p) 变为二级指针。

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}
```

---

35.程序改错

```
class mm1
```

```

{
private:
    static unsigned int x;
public:
    mml(){ x++; }
    mml(static unsigned int &) {x++;}
    ~mml{ x--; }
public:
    virtual mon() {} = 0;
    static unsigned int mmc(){return x;}
    .....
};

class nnl:public mml
{
private:
    static unsigned int y;
public:
    nnl(){ x++; }
    nnl(static unsigned int &) {x++;}
    ~nnl{ x--; }
public:
    virtual mon() {};
    static unsigned int nnc(){return y;}
    .....
};

```

代码片断:

```

mml* pp = new nnl;
.....
delete pp;

```

A:

基类的析构函数应该为虚函数

```
virtual ~mml{ x--; }
```

---

36.101 100个硬币 100真、1假，真假区别在于重量。请用无砝码天平称两次给出真币重还是假币重的结论。

答:

101个先取出2堆,

33,33

第一次称,如果不相等,说明有一堆重或轻

那么把重的那堆拿下来,再放另外 35 个中的 33

如果相等,说明假的重,如果不相等,新放上去的还是重的话,说明假的轻(不可能新放上去的轻)

第一次称,如果相等的话,这 66 个肯定都是真的,从这 66 个中取出 35 个来,与剩下的没称过的 35 个比  
下面就不用说了

方法二:

第 3 题也可以拿 A(50),B(50) 比一下,一样的话拿剩下的一个和真的比一下。

如果不一样,就拿其中的一堆。比如 A(50) 再分成两堆 25 比一下,一样的话就在  
B(50) 中,不一样就在 A(50) 中,结合第一次的结果就知道了。

---

37. static 变量和 static 函数各有什么特点?

答:

static 变量: 在程序运行期内一直有效,如果定义在函数外,则在编译单元内可见,如果在函数内,在在  
定义的 block 内可见;

static 函数: 在编译单元内可见;

---

38. 用 C 写一个输入的整数,倒着输出整数的函数,要求用递归方法;

答:

```
void fun( int a )
{
    printf( "%d", a%10 );
    a /= 10;
    if( a <=0 )return;

    fun( a );
}
```

---

39. 写出程序结果:

```
void Func(char str[100])
{
    printf("%d\n", sizeof(str));
}
```

答:

4

分析:

指针长度

---

```
40.int id[sizeof(unsigned long)];
```

这个对吗？为什么??

答：

对

这个 `sizeof` 是编译时运算符，编译时就确定了  
可以看成和机器有关的常量。

1. 以下三条输出语句分别输出什么？

```
char str1[] = "abc";
char str2[] = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char* str5 = "abc";
const char* str6 = "abc";
cout << boolalpha << ( str1==str2 ) << endl; // 输出什么?
cout << boolalpha << ( str3==str4 ) << endl; // 输出什么?
cout << boolalpha << ( str5==str6 ) << endl; // 输出什么?
```

答：分别输出 `false, false, true`。`str1` 和 `str2` 都是字符数组，每个都有其自己的存储区，它们的值则是各存储区首地址，不等；`str3` 和 `str4` 同上，只是按 `const` 语义，它们所指向的数据区不能修改。`str5` 和 `str6` 并非数组而是字符指针，并不分配存储区，其后的“abc”以常量形式存于静态数据区，而它们自己仅是指向该区首地址的指针，相等。

2. 以下代码中的两个 `sizeof` 用法有问题吗？

```
void Uppercase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
    for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
        if( 'a'<=str[i] && str[i]<='z' )
            str[i] -= ('a'-'A');
    char str[] = "aBcDe";
    cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
    Uppercase( str );
    cout << str << endl;
```

答：函数内的 `sizeof` 有问题。根据语法，`sizeof` 如用于数组，只能测出静态数组的大小，无法检测动态分配的或外部数组大小。函数外的 `str` 是一个静态定义的数组，因此其大小为 6，函数内的 `str` 实际只是一个指向字符串的指针，没有任何额外的与数组相关的信息，因此 `sizeof` 作用于上只将其当指针看，一个指针为 4 个字节，因此返回 4。

3. 非 C++ 内建型别 A 和 B，在哪几种情况下 B 能隐式转化为 A？

答：

a. `class B : public A { .....` } // B 公有继承自 A，可以是间接继承的

- b. class B { operator A( ); } // B 实现了隐式转化为 A 的转化
- c. class A { A( const B& ); } // A 实现了 non-explicit 的参数为 B (可以有其他带默认值的参数) 构造函数
- d. A& operator= ( const A& ); // 赋值操作, 虽不是正宗的隐式类型转换, 但也可以勉强算一个

4. 以下代码有什么问题?

```
struct Test
{
    Test( int ) {}
    Test() {}
    void fun() {}
};

void main( void )
{
    Test a(1);
    a.fun();
    Test b();
    b.fun();
}
```

答: 变量 **b** 定义出错。按默认构造函数定义对象, 不需要加括号。

5. 以下代码有什么问题?

```
cout << (true?1:"1") << endl;
```

答: 三元表达式“? :”问号后面的两个操作数必须为同一类型。

6. 以下代码能够编译通过吗, 为什么?

```
unsigned int const size1 = 2;
char str1[ size1 ];
unsigned int temp = 0;
cin >> temp;
unsigned int const size2 = temp;
char str2[ size2 ];
```

答: **str2** 定义出错, **size2** 非编译器期间常量, 而数组定义要求长度必须为编译期常量。

7. 以下反向遍历 **array** 数组的方法有什么错误?

```
vector array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 3 );
for( vector::size_type i=array.size()-1; i>=0; --i ) // 反向遍历 array 数组
{
    cout << array[i] << endl;
}
```

答：首先数组定义有误，应加上类型参数：`vector<int> array`。其次 `vector::size_type` 被定义为 `unsigned int`，即无符号数，这样做为循环变量的 `i` 为 0 时再减 1 就会变成最大的整数，导致循环失去控制。

8. 以下代码中的输出语句输出 0 吗，为什么？

```
struct CLS
{
    int m_i;
    CLS( int i ) : m_i(i) {}
    CLS()
    {
        CLS(0);
    }
    CLS obj;
    cout << obj.m_i << endl;
```

答：不能。在默认构造函数内部再调用带参的构造函数属用户行为而非编译器行为，亦即仅执行函数调用而不会执行其后的初始化表达式。只有在生成对象时，初始化表达式才会随相应的构造函数一起调用。

9. C++中的空类，默认产生哪些类成员函数？

答：

```
class Empty
{
public:
    Empty(); // 缺省构造函数
    Empty( const Empty& ); // 拷贝构造函数
    ~Empty(); // 析构函数
    Empty& operator=( const Empty& ); // 赋值运算符
    Empty* operator&(); // 取址运算符
    const Empty* operator&() const; // 取址运算符 const
};
```

10. 以下两条输出语句分别输出什么？

```
float a = 1.0f;
cout << (int)a << endl;
cout << (int&)a << endl;
cout << boolalpha << ( (int)a == (int&)a ) << endl; // 输出什么？
float b = 0.0f;
cout << (int)b << endl;
cout << (int&)b << endl;
cout << boolalpha << ( (int)b == (int&)b ) << endl; // 输出什么？
```

答：分别输出 `false` 和 `true`。注意转换的应用。`(int)a` 实际上是以浮点数 `a` 为参数构造了一个整型数，该整数的值是 `1`，`(int&)a` 则是告诉编译器将 `a` 当作整数看（并没有做任何实质上的转换）。因为 `1` 以整数形式存放和以浮点形式存放其内存数据是不一样的，因此两者不等。对 `b` 的两种转换意义同上，但是 `0` 的整数形式和浮点形式其内存数据是一样的，因此在这种特殊情形下，两者相等（仅仅在数值意义上）。

注意，程序的输出会显示`(int&)a=1065353216`，这个值是怎么来的呢？前面已经说了，`1` 以浮点数形式存放在内存中，按 `ieee754` 规定，其内容为 `0x0000803F`（已考虑字节反序）。这也就是 `a` 这个变量所占据的内存单元的值。当`(int&)a` 出现时，它相当于告诉它的上下文：“把这块地址当做整数看待！不要管它原来是什么。”这样，内容 `0x0000803F` 按整数解释，其值正好就是 `1065353216`（十进制数）。

通过查看汇编代码可以证实“`(int)a` 相当于重新构造了一个值等于 `a` 的整型数”之说，而`(int&)` 的作用则仅仅是表达了一个类型信息，意义在于为 `cout<<` 及 `==` 选择正确的重载版本。

## 11. 以下代码有什么问题？

```
typedef vector<IntArray> IntArray;  
IntArray array; <BR>
```

前言: `string` 的角色

1 `string` 使用

1.1 充分使用 `string` 操作符

1.2 眼花缭乱的 `string find` 函数

1.3 `string insert, replace, erase` 2 `string` 和 C 风格字符串

3 `string` 和 `Character Traits`

4 `string` 建议

5 小结

6 附录前言: `string` 的角色

`C++` 语言是个十分优秀的语言，但优秀并不表示完美。还是有许多人不愿意使用 `C` 或者 `C++`，为什么？原因众多，其中之一就是 `C/C++` 的文本处理功能太麻烦，用起来很不方便。以前没有接触过其他语言时每当别人这么说，我总是不屑一顾，认为他们根本就没有领会 `C++` 的精华，或者不太懂 `C++`，现在我接触 `perl`, `php`, 和 `Shell` 脚本以后，开始理解了以前为什么有人说 `C++` 文本处理不方便了。

举例来说，如果文本格式是：用户名 电话号码，文件名 `name.txt`

Tom 23245332

Jenny 22231231

Heny 22183942

Tom 23245332

...

现在我们需要对用户名排序，且只输出不同的姓名。

那么在 `shell` 编程中，可以这样用：

```
awk '{print $1}' name.txt | sort | uniq
```

简单吧？

如果使用 C/C++ 就麻烦了，他需要做以下工作：

先打开文件，检测文件是否打开，如果失败，则退出。

声明一个足够大得二维字符数组或者一个字符指针数组

读入一行到字符空间

然后分析一行的结构，找到空格，存入字符数组中。

关闭文件

写一个排序函数，或者使用写一个比较函数，使用 `qsort` 排序

遍历数组，比较是否有相同的，如果有，则要删除，`copy...`

输出信息

你可以用 C++ 或者 C 语言去实现这个流程。如果一个人的主要工作就是处理这种类似的文本（例如做 apache 的日志统计和分析），你说他会喜欢 C/C++ 么？

当然，有了 STL，这些处理会得到很大的简化。我们可以使用 `fstream` 来代替麻烦的 `fopen` `fread` `fclose`，用 `vector` 来代替数组。最重要的是用 `string` 来代替 `char *` 数组，使用 `sort` 排序算法来排序，用 `unique` 函数来去重。听起来好像很不错。看看下面代码（例程 1）：

```
#include <string>
#include <iostream>
#include <algorithm>
#include <vector>
#include <fstream>
using namespace std;
int main(){
    ifstream in("name.txt");
    string strtmp;
    vector<string> vect;
    while(getline(in, strtmp, '\n'))
        vect.push_back(strtmp.substr(0, strtmp.find(' ')));
    sort(vect.begin(), vect.end());
    vector<string>::iterator it=unique(vect.begin(), vect.end());
    copy(vect.begin(), it, ostream_iterator<string>(cout, "\n"));
    return 0;
}
```

也还不错吧，至少会比想象得要简单得多！（代码里面没有对错误进行处理，只是为了说明问题，不要效仿）。

当然，在这个文本格式中，不用 `vector` 而使用 `map` 会更有扩充性，例如，还可通过人名找电话号码等等，但是使用了 `map` 就不那么好用 `sort` 了。你可以用 `map` 试一试。

这里 `string` 的作用不只是可以存储字符串，还可以提供字符串的比较，查找等。在 `sort` 和 `unique` 函数中就默认使用了 `less` 和 `equal_to` 函数，上面的一段代码，其实使用了 `string` 的以下功能：

存储功能，在 `getline()` 函数中

查找功能，在 `find()` 函数中

子串功能，在 `substr()` 函数中

`string operator <`，默认在 `sort()` 函数中调用

`string operator ==`，默认在 `unique()` 函数中调用

总之，有了 `string` 后，C++的字符文本处理功能总算得到了一定补充，加上配合 STL 其他容器使用，其在文本处理上的功能已经与 perl, shell, php 的距离缩小很多了。因此掌握 `string` 会让你的工作事半功倍。

## 1 `string` 使用

其实，`string` 并不是一个单独的容器，只是 `basic_string` 模板类的一个 `typedef` 而已，相对应的还有 `wstring`，你在 `string` 头文件中你会发现下面的代码：

```
extern "C++" {
    typedef basic_string <char> string;
    typedef basic_string <wchar_t> wstring;
} // extern "C++"
```

由于只是解释 `string` 的用法，如果没有特殊的说明，本文并不区分 `string` 和 `basic_string` 的区别。

`string` 其实相当于一个保存字符的序列容器，因此除了有字符串的一些常用操作以外，还有包含了所有的序列容器的操作。字符串的常用操作包括：增加、删除、修改、查找比较、链接、输入、输出等。详细函数列表参看附录。不要害怕这么多函数，其实有许多是序列容器带有的，平时不一定用的上。

如果你要想了解所有函数的详细用法，你需要查看 `basic_string`，或者下载 STL 编程手册。这里通过实例介绍一些常用函数。

### 1.1 充分使用 `string` 操作符

`string` 重载了许多操作符，包括 `+`, `+=`, `<`, `=`, `,`, `[]`, `<<`, `>>` 等，正式这些操作符，对字符串操作非常方便。先看看下面这个例子： `tt.cpp` (例程 2)

```
#include <string>
#include <iostream>
using namespace std;
int main(){
```

```

string strinfo="Please input your name:";
cout << strinfo ;
cin >> strinfo;
if( strinfo == "winter" )
cout << "you are winter!"<<endl;
else if( strinfo != "wende" )
cout << "you are not wende!"<<endl;
else if( strinfo < "winter" )
cout << "your name should be ahead of winter"<<endl;
else
cout << "your name should be after of winter"<<endl;
strinfo += " , Welcome to China!";
cout << strinfo<<endl;
cout <<"Your name is :"<<endl;
string strtmp = "How are you? " + strinfo;
for(int i = 0 ; i < strtmp.size(); i++)
cout<<strtmp[i];
return 0;
}

```

下面是程序的输出

```

-bash-2.05b$ make tt
c++ -O -pipe -march=pentiumpro tt.cpp -o tt
-bash-2.05b$ ./tt
Please input your name:Hero
you are not wende!
Hero , Welcome to China!
How are you? Hero , Welcome to China!

```

有了这些操作符，在 **STL** 中仿函数都可以直接使用 **string** 作为参数，例如 **less**, **great**, **equal\_to** 等，因此在把 **string** 作为参数传递的时候，它的使用和 **int** 或者 **float** 等已经没有什么区别了。例如，你可以使用：

```

map<string, int> mymap;
//以上默认使用了 less<string>

```

有了 **operator +** 以后，你可以直接连加，例如：

```

string strinfo="Winter";
string strlast="Hello " + strinfo + "!";
//你还可以这样：
string strtest="Hello " + strinfo + " Welcome" + " to China" + "!";

```

看见其中的特点了吗？只要你的等式里面有一个 `string` 对象，你就可以一直连续“+”，但有一点需要保证的是，在开始的两项中，必须有一项是 `string` 对象。其原理很简单：

系统遇到“+”号，发现有一项是 `string` 对象。  
系统把另一项转化为一个临时 `string` 对象。  
执行 `operator +` 操作，返回新的临时 `string` 对象。  
如果又发现“+”号，继续第一步操作。

由于这个等式是由左到右开始检测执行，如果开始两项都是 `const char*`，程序自己并没有定义两个 `const char*` 的加法，编译的时候肯定就有问题了。

有了操作符以后，`assign()`, `append()`, `compare()`, `at()` 等函数，除非有一些特殊的需求时，一般是用不上。当然 `at()` 函数还有一个功能，那就是检查下标是否合法，如果是使用：

```
string str="winter";
//下面一行有可能会引起程序中断错误
str[100]='!';
//下面会抛出异常:throws: out_of_range
cout<<str.at(100)<<endl;
```

了解了吗？如果你希望效率高，还是使用`[]`来访问，如果你希望稳定性好，最好使用 `at()` 来访问。

## 1.2 眼花缭乱的 `string` `find` 函数

由于查找是使用最为频繁的功能之一，`string` 提供了非常丰富的查找函数。其列表如下：

函数名 描述  
`find` 查找 `rfind` 反向查找 `find_first_of` 查找包含子串中的任何字符，返回第一个位置  
`find_first_not_of` 查找不包含子串中的任何字符，返回第一个位置 `find_last_of` 查找包含子串中的任何字符，返回最后一个位置 `find_last_not_of` 查找不包含子串中的任何字符，返回最后一个位置 以上函数都是被重载了 4 次，以下是以 `find_first_of` 函数为例说明他们的参数，其他函数和其参数一样，也就是说总共有 24 个函数：

```
size_type find_first_of(const basic_string& s, size_type pos = 0)
size_type find_first_of(const charT* s, size_type pos, size_type n)
size_type find_first_of(const charT* s, size_type pos = 0)
size_type find_first_of(charT c, size_type pos = 0)
```

所有的查找函数都返回一个 `size_type` 类型，这个返回值一般都是所找到字符串的位置，如果没有找到，则返回 `string::npos`。有一点需要特别注意，所有和 `string::npos` 的比较一定要用 `string::size_type` 来使用，不要直接使用 `int` 或者 `unsigned int` 等类型。其实 `string::npos` 表示的是 -1，看看头文件：

```
template <class _CharT, class _Traits, class _Alloc>
const basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>::npos
= basic_string<_CharT,_Traits,_Alloc>::size_type) -1;
```

`find` 和 `find` 都还比较容易理解，一个是正向匹配，一个是逆向匹配，后面的参数 `pos` 都是用来指定起始查找位置。对于 `find_first_of` 和 `find_last_of` 就不是那么好理解。

`find_first_of` 是给定一个要查找的字符集，找到这个字符集中任何一个字符所在字符串中第一个位置。或许看一个例子更容易明白。

有这样一个需求：过滤一行开头和结尾的所有非英文字符。看看用 `string` 如何实现：

```
#i nclude <string>
#i nclude <iostream>
using namespace std;
int main(){
    string strinfo="  /*---Hello Word!.....-----";
    string strset="ABCDEFIGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
    int first = strinfo.find_first_of(strset);
    if(first == string::npos) {
        cout<<"not find any characters"<<endl;
        return -1;
    }
    int last = strinfo.find_last_of(strset);
    if(last == string::npos) {
        cout<<"not find any characters"<<endl;
        return -1;
    }
    cout << strinfo.substr(first, last - first + 1)<<endl;
    return 0;
}
```

这里把所有的英文字母大小写作为为了需要查找的字符集，先查找第一个英文字母的位置，然后查找最后一个英文字母的位置，然后用 `substr` 来的到中间的一部分，用于输出结果。下面就是其结果：

Hello Word

前面的符号和后面的符号都没有了。像这种用法可以用来查找分隔符，从而把一个连续的字符串分割成为几部分，达到 `shell` 命令中的 `awk` 的用法。特别是当分隔符有多个的时候，可以一次指定。例如有这样 的需求：

```
张三|3456123, 湖南
李四,4564234| 湖北
王小二, 4433253|北京
...
```

我们需要以 " | " 为分隔符, 同时又要过滤空格, 把每行分成相应的字段。可以作为你一个家庭作业来试试, 要求代码简洁。

### 1.3 string insert, replace, erase

了解了 `string` 的操作符, 查找函数和 `substr`, 其实就已经了解了 `string` 的 80% 的操作了。`insert` 函数, `replace` 函数和 `erase` 函数在使用起来相对简单。下面以一个例子来说明其应用。

`string` 只是提供了按照位置和区间的 `replace` 函数, 而不能用一个 `string` 字串来替换指定 `string` 中的另一个字串。这里写一个函数来实现这个功能:

```
void string_replace(string & strBig, const string & strsrc, const string & strdst) {  
    string::size_type pos=0;  
    string::size_type srclen=strsrc.size();  
    string::size_type dstlen=strdst.size();  
    while( (pos=strBig.find(strsrc, pos)) != string::npos){  
        strBig.replace(pos, srclen, strdst);  
        pos += dstlen;  
    }  
}
```

}看看如何调用:

```
#i nclude <string>  
#i nclude <iostream>  
using namespace std;  
int main() {  
    string strinfo="This is Winter, Winter is a programmer. Do you know Winter?";  
    cout<<"Orign string is :\n"<<strinfo<<endl;  
    string_replace(strinfo, "Winter", "wende");  
    cout<<"After replace Winter with wende, the string is :\n"<<strinfo<<endl;  
    return 0;  
}
```

}其输出结果:

Orign string is :

This is Winter, Winter is a programmer. Do you know Winter?

After replace Winter with wende, the string is :

This is wende, wende is a programmer. Do you know wende? 如果不用 `replace` 函数, 则可以使用 `erase` 和 `insert` 来替换, 也能实现 `string_replace` 函数的功能:

```
void string_replace(string & strBig, const string & strsrc, const string & strdst) {  
    string::size_type pos=0;  
    string::size_type srclen=strsrc.size();  
    string::size_type dstlen=strdst.size();  
    while( (pos=strBig.find(strsrc, pos)) != string::npos){  
        strBig.erase(pos, srclen);  
        strBig.insert(pos, strdst);  
        pos += dstlen;  
    }  
}
```

}当然, 这种方法没有使用 `replace` 来得直接。

## 2 string 和 C 风格字符串

现在看了这么多例子, 发现 `const char*` 可以和 `string` 直接转换, 例如我们在上面的例子中, 使用

string\_replace(strinfo, "Winter", "wende"); 来代用  
void string\_replace(string & strBig, const string & strsrc, const string & strdst) 在 C 语言中只有 `char*` 和 `const char*`, 为了使用起来方便, `string` 提供了三个函数满足其要求:

`const charT* c_str() const`  
`const charT* data() const`

`size_type copy(charT* buf, size_type n, size_type pos = 0) const` 其中:  
`c_str` 直接返回一个以`\0`结尾的字符串。

`data` 直接以数组方式返回 `string` 的内容, 其大小为 `size()` 的返回值, 结尾并没有`\0`字符。

`copy` 把 `string` 的内容拷贝到 `buf` 空间中。

你或许会问, `c_str()` 的功能包含 `data()`, 那还需要 `data()` 函数干什么? 看看源码:

```
const charT* c_str () const
{ if (length () == 0) return ""; terminate (); return data (); }原来 c_str() 的流程是: 先调用 terminate(), 然后在返回 data()。因此如果你对效率要求比较高, 而且你的处理又不一定需要以\0的方式结束, 你最好选择 data()。但是对于一般的 C 函数中, 需要以 const char* 为输入参数, 你就要使用 c_str() 函数。
```

对于 `c_str()` `data()` 函数, 返回的数组都是由 `string` 本身拥有, 千万不可修改其内容。其原因是许多 `string` 实现的时候采用了引用机制, 也就是说, 有可能几个 `string` 使用同一个字符存储空间。而且你不能使用 `sizeof(string)` 来查看其大小。详细的解释和实现查看 **Effective STL** 的条款 15: 小心 `string` 实现的多样性。

另外在你的程序中, 只在需要时才使用 `c_str()` 或者 `data()` 得到字符串, 每调用一次, 下次再使用就会失效, 如:

```
string strinfo("this is Winter");
...
//最好的方式是:
foo(strinfo.c_str());
//也可以这么用:
const char* pstr=strinfo.c_str();
foo(pstr);
//不要再使用了 pstr 了, 下面的操作已经使 pstr 无效了。
strinfo += " Hello!";
foo(pstr); //错误! 会遇到什么错误? 当你幸运的时候 pstr 可能只是指向"this is Winter Hello!"的字符串, 如果不幸运, 就会导致程序出现其他问题, 总会有一些不可预见的错误。总之不会是你预期的那个结果。
```

### 3 string 和 Character Traits

了解了 `string` 的用法, 该详细看看 `string` 的真相了。前面提到 `string` 只是 `basic_string` 的一个 `typedef`, 看看 `basic_string` 的参数:

```
template <class charT, class traits = char_traits<charT>,
class Allocator = allocator<charT> >
class basic_string
{
```

`char_traits` 不仅是在 `basic_string` 中有用, 在 `basic_istream` 和 `basic_ostream` 中也需要用到。

就像 **Steve Donovan** 在过度使用 C++ 模板中提到的，这些确实有些过头了，要不是系统自己定义了相关的一些属性，而且用了个 **typedef**，否则还真不知道如何使用。

但复杂总有复杂道理。有了 **char\_traits**，你可以定义自己的字符串类型。当然，有了 **char\_traits < char >** 和 **char\_traits < wchar\_t >** 你的需求使用已经足够了，为了更好的理解 **string**，咱们来看看 **char\_traits** 都有哪些要求。

如果你希望使用你自己定义的字符，你必须定义包含下列成员的结构： 表达式 描述

**char\_type** 字符类型

**int\_type** int 类型

**pos\_type** 位置类型

**off\_type** 表示位置之间距离的类型

**state\_type** 表示状态的类型

**assign(c1,c2)** 把字符 c2 赋值给 c1

**eq(c1,c2)** 判断 c1,c2 是否相等

**lt(c1,c2)** 判断 c1 是否小于 c2

**length(str)** 判断 str 的长度

**compare(s1,s2,n)** 比较 s1 和 s2 的前 n 个字符

**copy(s1,s2, n)** 把 s2 的前 n 个字符拷贝到 s1 中

**move(s1,s2, n)** 把 s2 中的前 n 个字符移动到 s1 中

**assign(s,n,c)** 把 s 中的前 n 个字符赋值为 c

**find(s,n,c)** 在 s 的前 n 个字符内查找 c

**eof()** 返回 end-of-file

**to\_int\_type(c)** 将 c 转换成 int\_type

**to\_char\_type(i)** 将 i 转换成 char\_type

**not\_eof(i)** 判断 i 是否为 EOF

**eq\_int\_type(i1,i2)** 判断 i1 和 i2 是否相等

想看看实际的例子，你可以看看 **sgi STL** 的 **char\_traits** 结构源码。

现在默认的 **string** 版本中，并不支持忽略大小写的比较函数和查找函数，如果你想练练手，你可以试试改写一个 **char\_traits**，然后生成一个 **case\_string** 类，也可以在 **string** 上做继承，然后派生一个新的类，例如： **ext\_string**，提供一些常用的功能，例如：

定义分隔符。给定分隔符，把 **string** 分为几个字段。

提供替换功能。例如，用 **winter**，替换字符串中的 **wende**

大小写处理。例如，忽略大小写比较，转换等

整形转换。例如把"123"字符串转换为 123 数字。

这些都是常用的功能，如果你有兴趣可以试试。其实有人已经实现了，看看 **Extended STL string**。如果你想偷懒，下载一个头文件就可以用，有了它确实方便了很多。要是有人能提供一个支持正则表达式的 **string**，我会非常乐意用。

#### 4 string 建议

使用 **string** 的方便性就不用再说了，这里要重点强调的是 **string** 的安全性。

**string** 并不是万能的，如果你在一个大工程中需要频繁处理字符串，而且有可能是多线程，那么你一定要慎重(当然，在多线程下你使用任何 **STL** 容器都要慎重)。

**string** 的实现和效率并不一定是你想象的那样，如果你对大量的字符串操作，而且特别关心其效率，那么你有两个选择，首先，你可以看看你使用的 **STL** 版本中 **string** 实现的源码；另一选择是你自己写一个只提供你需要的功能的类。

**string** 的 **c\_str()** 函数是用来得到 **C** 语言风格的字符串，其返回的指针不能修改其空间。而且在下一次使用时重新调用获得新的指针。

**string** 的 **data()** 函数返回的字符串指针不会以"\0"结束，千万不可忽视。

尽量去使用操作符，这样可以让程序更加易懂（特别是那些脚本程序员也可以看懂）

## 5 小结

难怪有人说：

**string** 使用方便功能强，我们一直用它！

## 6 附录

**string** 函数列表 函数名 描述

**begin** 得到指向字符串开头的 **Iterator**

**end** 得到指向字符串结尾的 **Iterator**

**rbegin** 得到指向反向字符串开头的 **Iterator**

**rend** 得到指向反向字符串结尾的 **Iterator**

**size** 得到字符串的大小

**length** 和 **size** 函数功能相同

**max\_size** 字符串可能的最大大小

**capacity** 在不重新分配内存的情况下，字符串可能的大小

**empty** 判断是否为空

**operator[]** 取第几个元素，相当于数组

**c\_str** 取得 **C** 风格的 **const char\*** 字符串

**data** 取得字符串内容地址

**operator=** 赋值操作符

**reserve** 预留空间

**swap** 交换函数

**insert** 插入字符

**append** 追加字符

**push\_back** 追加字符

**operator+=** **+=** 操作符

**erase** 删除字符串

**clear** 清空字符容器中所有内容

**resize** 重新分配空间

**assign** 和赋值操作符一样

**replace** 替代

**copy** 字符串到空间

**find** 查找

**rfind** 反向查找

**find\_first\_of** 查找包含子串中的任何字符，返回第一个位置

**find\_first\_not\_of** 查找不包含子串中的任何字符，返回第一个位置

**find\_last\_of** 查找包含子串中的任何字符，返回最后一个位置

**find\_last\_not\_of** 查找不包含子串中的任何字符，返回最后一个位置

```
substr 得到字符串
compare 比较字符串
operator+ 字符串链接
operator== 判断是否相等
operator!= 判断是否不等于
operator< 判断是否小于
operator>> 从输入流中读入字符串
operator<< 字符串写入输出流
getline 从输入流中读入一行
```

## c/c++面试题

已经 n 次倒在 c 语言面试的问题上，总结了一下，是由于基础知识不扎实。痛定思痛，决定好好努力！

### 1.引言

本文的写作目的并不在于提供 C/C++ 程序员求职面试指导，而旨在从技术上分析面试题的内涵。文中的大多数面试题来自各大论坛，部分试题解答也参考了网友的意见。

许多面试题看似简单，却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 strcpy 函数都可看出面试者在技术上究竟达到了怎样的程度，我们能真正写好一个 strcpy 函数吗？我们都觉得自己能，可是我们写出的 strcpy 很可能只能拿到 10 分中的 2 分。读者可从本文看到 strcpy 函数从 2 分到 10 分解答的例子，看看自己属于什么样的层次。此外，还有一些面试题考查面试者敏捷的思维能力。

分析这些面试题，本身包含很强的趣味性；而作为一名研发人员，通过对这些面试题的深入剖析则可进一步增强自身的内功。

### 2.找错题

试题 1:

```
void test1()
{
    char string[10];
    char* str1 = "0123456789";
    strcpy( string, str1 );
}
```

试题 2:

```
void test2()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

试题 3:

```

void test3(char* str1)
{
    char string[10];
    if( strlen( str1 ) <= 10 )
    {
        strcpy( string, str1 );
    }
}

```

解答：

试题 1 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；

对试题 2，如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string, str1) 调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；

对试题 3, 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。

剖析：

考查对基本功的掌握：

(1)字符串以'0'结尾；

(2)对数组越界把握的敏感度；

(3)库函数 strcpy 的工作方式，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

```

void strcpy( char *strDest, char *strSrc )
{
    while( (*strDest++ = * strSrc++) != '0' );
}

```

4 分

```

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

```

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！

```
char * strcpy( char *strDest, const char *strSrc )
```

```

{
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
    while( (*strDest++ = * strSrc++) != '0' );
    return address;
}

```

从 2 分到 10 分的几个答案我们可以清楚的看到，小小的 strcpy 竟然暗藏着这么多玄机，真不是盖的！需要多么扎实的基本功才能写一个完美的 strcpy 啊！

(4)对 strlen 的掌握，它没有包括字符串末尾的'0'。

读者看了不同分值的 strcpy 版本，应该也可以写出一个 10 分的 strlen 函数了，完美的版本为：

```
int strlen( const char *str ) //输入参数 const
```

```

{
    assert( str != NULL ); //断言字符串地址非 0
    int len;
    while( (*str++) != '0' )
    {
        len++;
    }
    return len;
}

```

试题 4：

```
void GetMemory( char *p )
{
    p = (char *) malloc( 100 );
}
```

```
void Test( void )
```

```
{
    char *str = NULL;
    GetMemory( str );
    strcpy( str, "hello world" );
    printf( str );
}
```

试题 5：

```
char *GetMemory( void )
{
    char p[] = "hello world";
    return p;
}
```

```
void Test( void )
```

```
{
```

```

char *str = NULL;
str = GetMemory();
printf( str );
}

试题 6:
void GetMemory( char **p, int num )
{
    *p = (char *) malloc( num );
}

void Test( void )
{
    char *str = NULL;
    GetMemory( &str, 100 );
    strcpy( str, "hello" );
    printf( str );
}

试题 7:
void Test( void )
{
    char *str = (char *) malloc( 100 );
    strcpy( str, "hello" );
    free( str );
    ... //省略的其它语句
}

```

解答：

试题 4 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值，执行完

```

char *str = NULL;
GetMemory( str );

```

后的 str 仍然为 NULL;

试题 5 中

```

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。

试题 6 的 GetMemory 避免了试题 4 的问题，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

```
*p = (char *) malloc( num );
```

后未判断内存是否申请成功，应加上：

```

if( *p == NULL )
{
    ...//进行申请内存失败处理
}

```

试题 7 存在与试题 6同样的问题，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。

剖析：

试题 4~7 考查面试者对内存操作的理解程度，基本功扎实的面试者一般都能正确的回答其中 50~60 的错误。但是要完全解答正确，却也绝非易事。

对内存操作的考查主要集中在：

- (1) 指针的理解；
- (2) 变量的生存期及作用范围；
- (3) 良好的动态内存申请和释放习惯。

再看看下面的一段程序有什么错误：

```
swap( int* p1,int* p2 )  
{  
    int *p;  
    *p = *p1;  
    *p1 = *p2;  
    *p2 = *p;  
}
```

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：

```
swap( int* p1,int* p2 )  
{  
    int p;  
    p = *p1;  
    *p1 = *p2;  
    *p2 = p;  
}
```

+++++

### 3. 内功题

试题 1：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）

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。

试题 2: 以下为 Windows NT 下的 32 位 C++ 程序, 请计算 sizeof 的值

```
void Func ( char str[100] )
```

```
{
```

```
    sizeof( str ) = ?
```

```
}
```

```
void *p = malloc( 100 );
```

```
sizeof( p ) = ?
```

解答:

```
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:

```
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。

试题 3: 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);
```

解答:

```
#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”和整个宏用括弧括起来。所以，严格地讲，下述解答：

```
#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：

```
((*p++) <= (b) ? (*p++) : (*p++))
```

这个表达式会产生副作用，指针 p 会作三次++自增操作。

除此之外，另一个应该判 0 分的解答是：

```
#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。

试题 4：为什么标准头文件都有类似以下的结构？

```
#ifndef __INCvxWorksh
```

```
#define __INCvxWorksh
```

```
#ifdef __cplusplus
```

```
extern "C" {
```

```
#endif
```

解答：

头文件中的编译宏

```
#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++ 就是考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。

```
+++++
```

试题 5：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

```
void LoopMove ( char * pStr, int steps )
```

```
{
```

//请填充...

```
}
```

解答：

正确解答 1：

```
void LoopMove ( char *pStr, int steps )
```

```
{
```

```
    int n = strlen( pStr ) - steps;
```

```
    char tmp[MAX_LEN];
```

```
    strcpy ( tmp, pStr + n );
```

```
    strcpy ( tmp + steps, pStr);
```

```
    *( tmp + strlen ( pStr ) ) = '0';
```

```
    strcpy( pStr, tmp );
```

```
}
```

正确解答 2：

```
void LoopMove ( char *pStr, int steps )
```

```
{
```

```
    int n = strlen( pStr ) - steps;
```

```
    char tmp[MAX_LEN];
```

```
    memcpy( tmp, pStr + n, steps );
```

```
    memcpy(pStr + steps, pStr, n );
```

```
    memcpy(pStr, tmp, steps );
```

```
}
```

剖析：

这个试题主要考查面试者对标准库函数的熟练程度，在需要的时候引用库函数可以很大程度上简化程序编写的工作量。

最频繁被使用的库函数包括：

(1) strcpy

(2) memcpy

(3) memset

试题 6：已知 WAV 文件格式如下表，打开一个 WAV 文件，以适当的数据结构组织 WAV 文件头并解析 WAV 格式的各项信息。

WAVE 文件格式说明表

1 #i include “filename.h”和#i include 的区别？

答：对于#i include 编译器从标准库开始搜索 filename.h

对于`#include "filename.h"`编译器从用户工作路径开始搜索 `filename.h`

## 2 头文件的作用是什么？

答：一、通过头文件来调用库功能。在很多场合，源代码不便（或不准）向用户公布，只要向用户提供头文件和二进制的库即可。用户只需

要按照头文件中的接口声明来调用库功能，而不必关心接口怎么实现的。编译器会从库中提取相应的代码。

二、头文件能加强类型安全检查。如果某个接口被实现或被使用时，其方式与头文件中的声明不一致，编译器就会指出错误，这一简单的规

则能大大减轻程序员调试、改错的负担。

## 3 C++函数中值的传递方式有哪几种？

答：C++函数的三种传递方式为：值传递、指针传递和引用传递。

## 4 内存的分配方式的分配方式有几种？

答：一、从静态存储区域分配。内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量。

二、在栈上创建。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。栈内

存分配运

算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

三、从堆上分配，亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存，程序员自己负责在何时用 `free`

或 `delete` 释放

内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

## 5 实现双向链表删除一个节点 P，在节点 P 后插入一个节点，写出这两个函数；

答：双向链表删除一个节点 P

```
template void list::delnode(int p)
{
int k=1;
listnode *ptr,*t;
ptr=first;
while(ptr->next!=NULL&&k!=p)
{
ptr=ptr->next;
k++;
}
t=ptr->next;
cout<<"你已经将数据项 "<data<<"删除"<
ptr->next=ptr->next->next;
length--;
delete t;
}
```

在节点 P 后插入一个节点：

```
template bool list::insert(type t,int p)
{
```

```

listnode *ptr;
ptr=first;
int k=1;
while(ptr!=NULL&&k{
ptr=ptr->next;
k++;
}
if(ptr==NULL&&k!=p)
return false;
else
{
listnode *tp;
tp=new listnode;
tp->data=t;
tp->next=ptr->next;
ptr->next=tp;
length++;
return true;
}
}

```

6 写一个函数，将其中的\t 都转换成 4 个空格。

```

void change(char* pstr)
{
while(*pstr++ != '\0')
{
if (*pstr == '\t')

}
}

```

7 Windows 程序的入口是哪里？写出 Windows 消息机制的流程。

答：Winmain()，它定义窗口界面和消息循环。设置全局资源->登记实例->调入全局资源->初始化应用->消息循环(解释消息,分发消息)

8 如何定义和实现一个类的成员函数为回调函数？

1). 不使用成员函数，直接使用普通 C 函数，为了实现在 C 函数中可以访问类的成员变量，可以使友元操作符(**friend**)，在 C++ 中将该 C 函数说明为类的友元即可。这种处理机制与普通的 C 编程中使用回调函数一样。

2). 使用静态成员函数，静态成员函数不使用 **this** 指针作为隐含参数，这样就可以作为回调函数了。

9 C++ 里面是不是所有的动作都是 main() 引起的？如果不是，请举例。

答：在运行 c++ 程序时，通常从 main() 函数开始执行。因此如果没有 main()，程序将不完整，编译器将指出未定义 main() 函数。

例外情况：如，在 windows 编程中，可以编写一个动态连接库 (dll) 模块，这是其他 windows 程序可以使用的代码。由于

DLL 模块不是独立的程序，因此不需要 main()。用于专用环境的程序--如机器人中的控制器芯片--

可能不需要 `main()`. 但常规的

独立程序都需要 `main()`.

10 C++ 里面如何声明 `const void f(void)` 函数为 C 程序中的库函数?

11 下列哪两个是等同的

int b;

A `const int* a = &b;`

B `const* int a = &b;`

C `const int* const a = &b;`

D `int const* const a = &b;`

12 内联函数在编译时是否做参数类型检查

13 三个 `float:a,b,c`

问值

$(a+b)+c == (b+a)+c$

$(a+b)+c == (a+c)+b$

14 把一个链表反向填空

`void reverse(test* head)`

{

`test* pe = head;`

`test* ps = head->next;`

`while(ps)`

{

`pe->next = ps->next;`

`ps->next = head;`

`head = ps;`

`ps = pe->next;`

}

}

15 设计一个重采样系统, 说明如何 anti-alias

16 某个程序在一个嵌入式系统(200M 的 CPU, 50M 的 SDRAM)中已经最优化了, 换到另一个系统(300M 的 CPU, 50M 的 SDRAM)中运行, 还需要优化吗?

17. 下面哪种排序法对 12354 最快

a. quick sort

b. bubble sort

c. merge sort

18. 哪种结构, 平均来讲, 获取一个值最快

a. binary tree

b. hash table

c. stack

19 请问 C++ 的类和 C 里面的 struct 有什么区别?

答: C++ 的类的成员默认情况下是私有的, C 的 struct 的成员默认情况下是公共的.

20 请讲一讲析构函数和虚函数的用法和作用?

答: 析构函数的作用是当对象生命期结束时释放对象所占用的资源。 析构函数用法: 析构函数是特殊的类成员函数

它的名字和类名相同, 没有返回值, 没有参数不能随意调用也没有重载。只是在类对象生命期结束时有系统自动调用。

虚函数用在继承中, 当在派生类中需要重新定义基类的函数时需要在基类中将该函数声明为虚函数, 作用为使程序支持动态联编。

21 全局变量和局部变量有什么区别? 是怎么实现的? 操作系统和编译器是怎么知道的?

答: 一些变量整个程序中都是可见的, 它们称为全局变量, 一些变量在函数内部定义且只在函数中可知, 则称为局部变量。

全局变量由编译器建立且存放在内存的全局数据区, 局部变量存放在栈区

22 一些寄存器的题目, 主要是寻址和内存管理等一些知识。

23 8086 是多少尉的系统? 在数据总线上是怎么实现的?

24 多态。overload 和 override 的区别。

答: 重载在相同范围(同一个类中), 函数名字相同, 参数不同, `virtual` 关键字可有可无。

覆盖是指派生类函数覆盖基类函数, 不同的范围, 函数名字相同, 参数相同, 基类函数必须有 `virtual` 关键字。

<>

25. 完成下列程序

```
*  
*.*.  
*..*..*..  
*...*...*...*..  
*....*....*....*....  
*....*....*....*....*....  
*....*....*....*....*....*....  
#i nclude  
using namespace std;  
const int n = 8;  
main()  
{  
int i;  
int j;  
int k;  
for(i = n; i >= 1; i--)  
{  
for(j = 0; j < n-i+1; j++)  
{  
cout<<"*";  
for(k=1; k < n-i+1; k++)  
{  
cout<<".";
```

```
    }
}
cout< }
system("pause");
}
```

26 完成程序，实现对数组的降序排序

```
#i nclude
using namespace std;
void sort(int* arr, int n);
int main()
{
int array[]={45,56,76,234,1,34,23,2,3};
sort(array, 9);
for(int i = 0; i <= 8; i++)//曾经在这儿出界
cout<}
void sort(int* arr, int n)
{
int temp;
for(int i = 1; i < 9; i++)
{
for(int k = 0; k < 9 - i; k++)//曾经在这儿出界
{
if(arr[k] < arr[k + 1])
{
temp = arr[k];
arr[k] = arr[k + 1];
arr[k + 1] = temp;
}
}
}
}
```

27 费波那其数列，1，1，2，3，5.....编写程序求第十项。可以用递归，也可以用其他方法，但要说明你选择的理由。

非递归

```
#i nclude
using namespace std;
int Pheponatch(int n);
main()
{
int Ph = Pheponatch(10);
cout< system("pause");
}
```

```

int Pheponatch(int n)
{
    int elem;
    int n1 = 1;
    int n2 = 1;
    if(n == 1 || n ==2)
        return 1;
    else
    {
        for(int i = 3; i <= n; i++)
        {
            elem = n1 + n2;
            n1 = n2;
            n2 = elem;
        }
        return elem;
    }
}
递归
#include
using namespace std;
int Pheponatch(int n);
main()
{
    int n;
    cin>>n;
    int ph = Pheponatch(n);
    cout<< system("pause");
}
int Pheponatch(int n)
{
    if(n <= 0)
        exit(-1);
    else
        if(n == 1 || n ==2)
            return 1;
        else
            return Pheponatch(n - 1) + Pheponatch(n - 2);
}
28 下列程序运行时会崩溃, 请找出错误并改正, 并且说明原因。
#include
#include

typedef struct{

```

```

TNode* left;
TNode* right;
int value;
} TNode;

TNode* root=NULL;

void append(int N);

int main()
{
append(63);
append(45);
append(32);
append(77);
append(96);
append(21);
append(17); // Again, 数字任意给出
}

void append(int N)
{
TNode* NewNode=(TNode *)malloc(sizeof(TNode));
NewNode->value=N;

if(root==NULL)
{
root=NewNode;
return;
}
else
{
TNode* temp;
temp=root;
while((N>=temp.value && temp.left!=NULL) || (N
))
{
while(N>=temp.value && temp.left!=NULL)
temp=temp.left;
while(N
temp=temp.right;
}
if(N>=temp.value)
temp.left=NewNode;
}

```

```
else
temp.right=NewNode;
return;
}
}
```

29. A class B network on the internet has a subnet mask of 255.255.240.0, what is the maximum number of hosts per subnet .  
a. 240 b. 255 c. 4094 d. 65534
30. What is the difference: between  $O(\log n)$  and  $O(\log n^2)$ , where both logarithms have base 2 .  
a.  $O(\log n^2)$  is bigger b.  $O(\log n)$  is bigger  
c. no difference
31. For a class what would happen if we call a class's constructor from with the same class's constructor .  
a. compilation error b. linking error  
c. stack overflow d. none of the above
32. "new" in c++ is a: .  
a. library function like malloc in c  
b. key word c. operator  
d. none of the above
33. Which of the following information is not contained in an inode .  
a. file owner b. file size  
c. file name d. disk address
34. What's the number of comparisons in the worst case to merge two sorted lists containing  $n$  elements each .  
a.  $2n$  b.  $2n-1$  c.  $2n+1$  d.  $2n-2$
35. Time complexity of  $n$  algorithm  $T(n)$ , where  $n$  is the input size ,is  $T(n)=T(n-1)+1/n$  if  $n>1$  otherwise 1 the order of this algorithm is .  
a.  $\log(n)$  b.  $n$  c.  $n^2$  d.  $n^n$
36. The number of 1's in the binary representation of  $3*4096+15*256+5*16+3$  are .  
a. 8 b. 9 c. 10 d. 12
37. 设计函数 `int atoi(char *s)`。
38. `int i=(j=4,k=8,l=16,m=32); printf("%d", i);` 输出是多少?
39. 解释局部变量、全局变量和静态变量的含义。
40. 解释堆和栈的区别。
- 栈区 (stack) — 由编译器自动分配释放，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。
- 堆:一般由程序员分配释放，若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
41. 论述含参数的宏与函数的优缺点。

42. 以下三条输出语句分别输出什么? [C 易]

```
char str1[] = "abc";
char str2[] = "abc";
const char str3[] = "abc";
const char str4[] = "abc";
const char* str5 = "abc";
const char* str6 = "abc";
cout << boolalpha << ( str1==str2 ) << endl; // 输出什么?
cout << boolalpha << ( str3==str4 ) << endl; // 输出什么?
cout << boolalpha << ( str5==str6 ) << endl; // 输出什么?
```

43. 非 C++ 内建型别 A 和 B, 在哪几种情况下 B 能隐式转化为 A? [C++ 中等]

答:

- a. class B : public A { .....} // B 公有继承自 A, 可以是间接继承的
- b. class B { operator A(); } // B 实现了隐式转化为 A 的转化
- c. class A { A( const B& ); } // A 实现了 non-explicit 的参数为 B (可以有其他带默认值的参数) 构造函数
- d. A& operator= ( const A& ); // 赋值操作, 虽不是正宗的隐式类型转换, 但也可以勉强算一个

44. 以下代码中的两个 sizeof 用法有问题吗? [C 易]

```
void UpperCase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
    for( size_t i=0; i < str.length(); i++ )
        if( 'a'<=str[i] && str[i]<='z' )
            str[i] -= ('a'-'A');
    cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
    UpperCase( str );
    cout << str << endl;
}
```

45. 以下代码有什么问题? [C 难]

```
void char2Hex( char c ) // 将字符以 16 进制表示
{
    char ch = c/0x10 + '0'; if( ch > '9' ) ch += ('A'-'9'-1);
    char cl = c%0x10 + '0'; if( cl > '9' ) cl += ('A'-'9'-1);
    cout << ch << cl << ' ';
}
char str[] = "I love 中国";
for( size_t i=0; i < str.length(); i++ )
    cout << char2Hex( str[i] );
cout << endl;
```

46. 以下代码有什么问题? [C++ 易]

```
struct Test
{
    Test( int ) {}
    Test() {}
    void fun() {}
}
```

```
};

void main( void )
{
    Test a(1);
    a.fun();
    Test b();
    b.fun();
}

*** Test b(); // 定义了一个函数
```

47. 以下代码有什么问题? [C++易]

```
cout << (true?1:"1") << endl;
```

8. 以下代码能够编译通过吗, 为什么? [C++易]

```
unsigned int const size1 = 2;
char str1[ size1 ];
unsigned int temp = 0;
cin >> temp;
unsigned int const size2 = temp;
char str2[ size2 ];
```

48. 以下代码中的输出语句输出 0 吗, 为什么? [C++易]

```
struct CLS
{
    int m_i;
    CLS( int i ) : m_i(i) {}
    CLS()
    {
        CLS(0);
    }
};

CLS obj;
cout << obj.m_i << endl;
```

49. C++中的空类, 默认产生哪些类成员函数? [C++易]

答:

```
class Empty
{
public:
    Empty(); // 缺省构造函数
    Empty( const Empty& ); // 拷贝构造函数
    ~Empty(); // 析构函数
    Empty& operator=( const Empty& ); // 赋值运算符
    Empty* operator&(); // 取址运算符
    const Empty* operator&() const; // 取址运算符 const
};
```

50. 以下两条输出语句分别输出什么? [C++难]

```
float a = 1.0f;
```

```
cout << (int)a << endl;
cout << (int&)a << endl;
cout << boolalpha << ( (int)a == (int&)a ) << endl; // 输出什么?
float b = 0.0f;
cout << (int)b << endl;
cout << (int&)b << endl;
cout << boolalpha << ( (int)b == (int&)b ) << endl; // 输出什么?
```

51. 以下反向遍历 `array` 数组的方法有什么错误? [STL 易]

```
vector array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 3 );
for( vector::size_type i=array.size()-1; i>=0; --i ) // 反向遍历 array 数组
{
    cout << array[i] << endl;
}
```

52. 以下代码有什么问题? [STL 易]

```
typedef vector IntArray;
IntArray array;
array.push_back( 1 );
array.push_back( 2 );
array.push_back( 2 );
array.push_back( 3 );
// 删除 array 数组中所有的 2
for( IntArray::iterator itor=array.begin(); itor!=array.end(); ++itor )
{
    if( 2 == *itor ) array.erase( itor );
}
```

53. 写一个函数, 完成内存之间的拷贝。[考虑问题是否全面]

答:

```
void* mymemcpy( void *dest, const void *src, size_t count )
{
    char* pdest = static_cast( dest );
    const char* psrc = static_cast( src );
    if( pdest>psrc && pdest {
        for( size_t i=count-1; i!= -1; --i )
            pdest[i] = psrc[i];
    }
    else
    {
        for( size_t i=0; i < pdest[i] = psrc[i];
    }
    return dest;
}
```

```
int main( void )
{
char str[] = "0123456789";
mymemcpy( str+1, str+0, 9 );
cout << str << endl;
system( "Pause" );
return 0;
}
```

#### 54 线程与进程的区别

进程：（在批处理系统中）是资源分配的最小单位

线程：最独立运行的最小单位。

一个进程中可以一个或多个线程。当系统的资源分配给进程，线程从所属的进程中得到保证能运行的很少的资源，线程完成后把资源“还”给进程，只有当进程中的线程全都完成后，进程才把占有的系统的资源释放（进程挂起除外）。

Top

55：请你分别划划 OSI 的七层网络结构图，和 TCP/IP 的五层结构图？

56：请你详细的解释一下 IP 协议的定义，在哪个层上面，主要有什么作用？ TCP 与 UDP 呢？

IP 协议是网络层的协议，它实现了 Internet 中自动路由的功能，即寻径的功能，TCP 协议是一个传输性的协议它向下屏蔽了 IP 协议不可靠传输的特性，向上提供一个可靠的点到点的传输，UDP 提供的是一种无连接的服务，主要考虑到很多应用不需要可靠的连接，但需要快速的传输

57：请问交换机和路由器分别的实现原理是什么？分别在哪个层次上面实现的？

交换机用在局域网中，交换机通过纪录局域网内各节点机器的 MAC 地址（物理地址）就可以实现传递报文，无需看报文中的 IP 地址。路由器识别不同网络的方法是通过识别不同网络的网络 ID 号（IP 地址的高端部分）进行的，所以为了保证路由成功，每个网络都必须有一个唯一的网络编号。路由器通过察看报文中 IP 地址，来决定路径，向那个子网（下一跳）路由，也就是说交换机工作在数据链路层看 MAC 地址，路由器工作在网际层看 IP 地址

但是由于现在网络设备的发展，很多设备既有交换机的功能又有路由器的功能（交换机路由器）使得两者界限越来越模糊。

58：请问 C++ 的类和 C 里面的 struct 有什么区别？

59：请讲一讲析构函数和虚函数的用法和作用？

60：全局变量和局部变量有什么区别？它们怎么实现的？操作系统和编译器是怎么知道的？

全局变量是整个程序都可访问的变量，谁都可以访问，生存期在整个程序从运行到结束（在程序结束时所占内存释放），而局部变量存在于模块（子程序，函数）中，只有所在模块可以访问，其他模块不可直接访问，模块结束（函数调用完毕），局部变量消失，所占据的内存释放。

全局变量分配在全局数据段并且在程序开始运行的时候被加载，局部变量则分配在堆栈里面。

61：一些寄存器的题目，主要是寻址和内存管理等一些知识。

2、交换机用在局域网中，交换机通过纪录局域网内各节点机器的 **MAC 地址(物理地址)**就可以实现传递报文,无需看报文中的 **IP 地址**。路由器识别不同网络的方法是通过识别不同网络的网络 ID 号(**IP 地址的高端部分**)进行的，所以为了保证路由成功，每个网络都必须有一个唯一的网络编号。路由器通过察看报文中 **IP 地址**，来决定路径，向那个子网(下一跳)路由，也就是说交换机工作在数据链路层看 **MAC 地址**，路由器工作在网际层看 **IP 地址**

但是由于现在网络设备的发展，很多设备既有交换机的功能又有路由器的功能(交换机路由器)使得两者界限越来越模糊。

3、**IP** 协议是网络层的协议，它实现了 **Internet** 中自动路由的功能，即寻径的功能，**TCP** 协议是一个传输性的协议它向下屏蔽了 **IP** 协议不可靠传输的特性，向上提供一个可靠的点到点的传输，**UDP** 提供的是一种无连接的服务，主要考虑到很多应用不需要可靠的连接，但需要快速的传输

4、

Test b();//定义了一个函数

62:8086 是多少位的系统？在数据总线上是怎么实现的？

<>

63.怎样用最快的方法判断链表是否有环？

64.c++中引用和指针有什么不同？指针加上什么限制等于引用？

答：**1** 引用被创建的时候必须被初始化，而指针不是必需的。**2** 引用在创建后就不能改变引用的关系，而指针在初始化后可以随时指向

其它的变量或对象。**3** 没有 **NULL** 引用，引用必须与合法的存储单元关联，而指针可以是 **NULL**。

65.做的项目，遇到的困难，怎样解决？

69.操作符重载

```
class CMyObject:public CObject
{
Public:
CMyObject();
CMyObject &operator=(const CMyObject &my);
private:
CString strName;
int nId;
};
```

请重载赋值操作符

70.链表

```
Struct structList
{
int value;
structList *pHead;
}
Struct LinkedList *pMyList;
```

请编写删除链表的头、尾和第 **n** 个节点的程序

71.用 **Socket API** 制作一个聊天程序，通讯协议使用 **tcp/ip**。要求有简单界面即可，支持多人聊天。

72.如果有过工作经验,请说明在先前公司的工作以及离职原因(如无,请说明毕业后的个人展望)

\*\*\*\*\*

\*\*\*\*\*

73 对于 C++ 中类(class) 与结构(struct)的描述正确的为:

A,类中的成员默认是 private 的,当是可以声明为 public,private 和 protected,结构中定义的成员默认的都是 public;

B,结构中不允许定义成员函数,当是类中可以定义成员函数;

C,结构实例使用 malloc() 动态创建,类对象使用 new 操作符动态分配内存;

D,结构和类对象都必须使用 new 创建;

E,结构中不可以定义虚函数,当是类中可以定义虚函数.

F,结构不可以存在继承关系,当是类可以存在继承关系.

答:A,D,F

74,两个互相独立的类:ClassA 和 ClassB,都各自定义了非静态的公有成员函数 PublicFunc() 和非静态的私有成员函数 PrivateFunc();

现在要在 ClassA 中增加定义一个成员函数 ClassA::AdditionalPunction(ClassA a,ClassB b);  
则可以在 AdditionalPunction(ClassA x,ClassB y) 的实现部分(函数功能体内部)  
出现的合法的表达是最全的是:

A,x.PrivateFunc();x.PublicFunc();y.PrivateFunc();y.PublicFunc();

B,x.PrivateFunc();x.PublicFunc();y.PublicFunc();

C,x.PrivateFunc();y.PrivateFunc();y.PublicFunc();

D,x.PublicFunc();y.PublicFunc();

答:B

75,C++程序下列说法正确的有:

A,对调用的虚函数和模板类都进行迟后编译.

B,基类与子类中函数如果要构成虚函数,除了要求在基类中用 virtual 声名,而且必须名字相同且参数类型相同返回类型相同

C,重载的类成员函数都必须要:或者返回类型不同,或者参数数目不同,或者参数序列的类型不同.

D,静态成员函数和内联函数不能是虚函数,友员函数和构造函数也不能是虚函数,但是析构函数可以是虚函数.

答:A

\*\*\*\*\*

\*\*\*\*\*

76,C++中的类与结构的区别?

77,构造函数和析构函数是否可以被重载,为什么?

答:构造函数可以被重载,析构函数不可以被重载.因为构造函数可以有多个且可以带参数,而析构函数只能有一个,且不能带参数.

78,一个类的构造函数和析构函数什么时候被调用,是否需要手工调用?

答:构造函数在创建类对象的时候被自动调用,析构函数在类对象生命期结束时,由系统自动调用.

1 #include "filename.h"和#include 的区别?

答: `#include "filename.h"`表明该文件是用户提供的头文件, 查找该文件时从当前文件目录开始  
`#include` 表明这个文件是一个工程或标准头文件, 查找过程会检查预定义的目录。

## 2 头文件的作用是什么?

答: 一、通过头文件来调用库功能。在很多场合, 源代码不便(或不准)向用户公布, 只要向用户提供头文件和二进制的库即可。用户只需要按照头文件中的接口声明来调用库功能, 而不必关心接口怎么实现的。编译器会从库中提取相应的代码。

二、头文件能加强类型安全检查。如果某个接口被实现或被使用时, 其方式与头文件中的声明不一致, 编译器就会指出错误, 这一简单的规则能大大减轻程序员调试、改错的负担。

## 3 C++函数中值的传递方式有哪几种?

答: C++函数的三种传递方式为: 值传递、指针传递和引用传递。

## 4 内存的分配方式的分配方式有几种?

答: 一、从静态存储区域分配。内存在程序编译的时候就已经分配好, 这块内存程序的整个运行期间都存在。例如全局变量。

二、在栈上创建。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。栈内存分配运算内置于处理器的指令集中, 效率很高, 但是分配的内存容量有限。

三、从堆上分配, 亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存, 程序员自己负责在何时用 `free` 或 `delete` 释放内存。动态内存的生存期由我们决定, 使用非常灵活, 但问题也最多。

## 5 实现双向链表删除一个节点 P, 在节点 P 后插入一个节点, 写出这两个函数;

答: 双向链表删除一个节点 P

```
template void list::delnode(int p)
{
int k=1;
listnode *ptr,*t;
ptr=first;
while(ptr->next!=NULL&&k!=p)
{
ptr=ptr->next;
k++;
}
t=ptr->next;
cout<<"你已经将数据项 "<<data<<"删除"<
ptr->next=ptr->next->next;
length--;
delete t;
}
```

在节点 P 后插入一个节点:

```
template bool list::insert(type t,int p)
{
listnode *ptr;
ptr=first;
int k=1;
while(ptr!=NULL&&k{
```

```
ptr=ptr->next;
k++;
}
if(ptr==NULL&&k!=p)
return false;
else
{
listnode *tp;
tp=new listnode;
tp->data=t;
tp->next=ptr->next;
ptr->next=tp;
length++;
return true;
}
}
```

//上海贝尔的面试题 43 分即可进入复试

一、请填写 **BOOL** , **float** , 指针变量 与“零值”比较的 **if** 语句。 (10 分)

提示: 这里“零值”可以是 0, 0.0 , FALSE 或者“空指针”。例如 **int** 变量 n 与“零值”比较的 **if** 语句为:

```
if ( n == 0 )
if ( n != 0 )
```

以此类推。

请写出 **BOOL** flag 与“零值”比较的 **if** 语句:

请写出 **float** x 与“零值”比较的 **if** 语句:

请写出 **char** \*p 与“零值”比较的 **if** 语句:

二、以下为 Windows NT 下的 32 位 C++ 程序, 请计算 **sizeof** 的值 (10 分)

```
char str[] = "Hello" ;
```

```
char *p = str ;
```

```
int n = 10;
```

请计算

```
sizeof (str) =
```

```
sizeof ( p ) =
```

```
sizeof ( n ) =void Func ( char str[100])
```

```
{
```

请计算

```
sizeof( str ) =
```

```
}
```

```
void *p = malloc( 100 );
```

请计算

```
sizeof ( p ) =
```

、简答题 (25 分)

- 1、头文件中的 `ifndef/define/endif` 干什么用?
- 2、`#include` 和 `#i include "filename.h"` 有什么区别?
- 3、`const` 有什么用途? (请至少说明两种)
- 4、在 C++ 程序中调用被 C 编译器编译后的函数, 为什么要加 `extern "C"` 声明?
- 5、请简述以下两个 `for` 循环的优缺点

```
// 第一个
for (i=0; i<
if (condition)
DoSomething();
else
DoOtherthing();
}// 第二个
if (condition)
{
for (i=0; i DoSomething();
}
else
{
for (i=0; i DoOtherthing();
```

优点: N 次中, 每次都要对 condition 进行判断

缺点: 优点: 一次判断 condition 后, 对 something 或 Otherthing 执行 N 次

缺点:

#### 四、有关内存的思考题 (20 分)

```
void GetMemory(char *p)
{
p = (char *)malloc(100);
}
void Test(void)
{
char *str = NULL;
GetMemory(str);
strcpy(str, "hello world");
printf(str);
}
```

请问运行 `Test` 函数会有什么样的结果?

```
char *GetMemory(void)
{
char p[] = "hello world";
return p;
}
void Test(void)
{
char *str = NULL;
str = GetMemory();
```

```
printf(str);
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

```
Void GetMemory2(char **p, int num)
```

```
{
```

```
*p = (char *)malloc(num);
```

```
}
```

```
void Test(void)
```

```
{
```

```
char *str = NULL;
```

```
GetMemory(&str, 100);
```

```
strcpy(str, "hello");
```

```
printf(str);
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

```
void Test(void)
```

```
{
```

```
char *str = (char *) malloc(100);
```

```
strcpy(str, "hello");
```

```
free(str);
```

```
if(str != NULL)
```

```
{
```

```
strcpy(str, "world");
```

```
printf(str);
```

```
}
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

## 五、编写 **strcpy** 函数 (10 分)

已知 **strcpy** 函数的原型是

```
char *strcpy(char *strDest, const char *strSrc);
```

其中 **strDest** 是目的字符串，**strSrc** 是源字符串。

(1) 不调用 C++/C 的字符串库函数，请编写函数 **strcpy**

```
char *strcpy(char *strDest, const char *strSrc){
```

```
int n=0;
```

```
while(strSrc[n]!=NULL){
```

```
    n++;
```

```
}
```

```
    *strDest=new char[n];
```

```
    for(int i=0;i<n;i++) strDest[i]=strSrc[i];
```

```
    i++;
```

```
}
```

```
return *strDest;
```

}

(2) `strcpy` 能把 `strSrc` 的内容复制到 `strDest`, 为什么还要 `char *` 类型的返回值?

因为该函数的还可以把复制的字符串首地址指针给其他的指针, 而且这种需要也是有用的。

六、编写类 `String` 的构造函数、析构函数和赋值函数 (25 分)

已知类 `String` 的原型为:

```
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
};
```

请编写 `String` 的上述 4 个函数。

//答案一并给出

一、请填写 `BOOL`, `float`, 指针变量 与“零值”比较的 `if` 语句。 (10 分)

请写出 `BOOL flag` 与“零值”比较的 `if` 语句。 (3 分)

标准答案:

```
if ( flag )
if ( !flag )如下写法均属不良风格, 不得分。
if (flag == TRUE)
if (flag == 1 )
if (flag == FALSE)
if (flag == 0)
```

请写出 `float x` 与“零值”比较的 `if` 语句。 (4 分)

标准答案示例:

```
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON))
```

不可将浮点变量用“==”或“!=”与数字比较, 应该设法转化成“>=”或“<=”此类形式。

如下是错误的写法, 不得分。

```
if (x == 0.0)
if (x != 0.0)
```

请写出 `char *p` 与“零值”比较的 `if` 语句。 (3 分)

标准答案:

```
if (p == NULL)
if (p != NULL)如下写法均属不良风格, 不得分。
if (p == 0)
if (p != 0)
if (p)
if (!)
```

二、以下为 Windows NT 下的 32 位 C++ 程序, 请计算 `sizeof` 的值 (10 分)

```
char str[] = "Hello" ;
char *p = str ;
int n = 10;
请计算
sizeof (str) = 6 (2 分)
sizeof (p) = 4 (2 分)
sizeof (n) = 4 (2 分) void Func ( char str[100])
{
请计算
sizeof( str ) = 4 (2 分)
}
void *p = malloc( 100 );
请计算
sizeof (p) = 4 (2 分)
```

### 三、简答题 (25 分)

1、头文件中的 `ifndef/define/endif` 干什么用? (5 分)

答: 防止该头文件被重复引用。

2、`#include` 和 `#i include "filename.h"` 有什么区别? (5 分)

答: 对于`#include`，编译器从标准库路径开始搜索 `filename.h`

对于`#i include "filename.h"`，编译器从用户的工作路径开始搜索 `filename.h`

3、`const` 有什么用途? (请至少说明两种) (5 分)

答: (1) 可以定义 `const` 常量

(2) `const` 可以修饰函数的参数、返回值，甚至函数的定义体。被 `const` 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

4、在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 `extern "C"`? (5 分)

答: C++ 语言支持函数重载，C 语言不支持函数重载。函数被 C++ 编译后在库中的名字与 C 语言的不同。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。

C++ 提供了 C 连接交换指定符号 `extern "C"` 来解决名字匹配问题。

5、请简述以下两个 `for` 循环的优缺点 (5 分)

```
for (i=0; i{
if (condition)
DoSomething();
else
DoOtherthing();
}if (condition)
{
for (i=0; i DoSomething();
}
else
{
for (i=0; i DoOtherthing();
```

}

优点：程序简洁

缺点：多执行了  $N-1$  次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能对循环进行优化处理，降低了效率。

优点：循环的效率高

缺点：程序不简洁

#### 四、有关内存的思考题（每小题 5 分，共 20 分）

```
void GetMemory(char *p)
```

```
{
```

```
    p = (char *)malloc(100);
```

```
}
```

```
void Test(void)
```

```
{
```

```
    char *str = NULL;
```

```
    GetMemory(str);
```

```
    strcpy(str, "hello world");
```

```
    printf(str);
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

答：程序崩溃。

因为 **GetMemory** 并不能传递动态内存，

**Test** 函数中的 **str** 一直都是 **NULL**。

**strcpy(str, "hello world");** 将使程序崩溃。

```
char *GetMemory(void)
```

```
{
```

```
    char p[] = "hello world";
```

```
    return p;
```

```
}
```

```
void Test(void)
```

```
{
```

```
    char *str = NULL;
```

```
    str = GetMemory();
```

```
    printf(str);
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

答：可能是乱码。

因为 **GetMemory** 返回的是指向“栈内存”的指针，该指针的地址不是 **NULL**，但其原现的内容已经被清除，新内容不可知。

```
void GetMemory2(char **p, int num)
```

```
{
```

```
*p = (char *)malloc(num);
```

```
}
```

```
void Test(void)
```

```
{
```

```
char *str = NULL;
GetMemory(&str, 100);
strcpy(str, "hello");
printf(str);
}
```

请问运行 **Test** 函数会有什么样的结果?

答:

- (1) 能够输出 **hello**
- (2) 内存泄漏

```
void Test(void)
{
char *str = (char *) malloc(100);
strcpy(str, "hello");
free(str);
if(str != NULL)
{
strcpy(str, "world");
printf(str);
}
}
```

请问运行 **Test** 函数会有什么样的结果?

答: 篡改动态内存区的内容, 后果难以预料, 非常危险。

因为 **free(str);** 之后, **str** 成为野指针,

**if(str != NULL)** 语句不起作用。

27 费波那其数列, 1, 1, 2, 3, 5.....编写程序求第十项。可以用递归, 也可以用其他方法, 但要说明你选择的理由。

---

```
#i nclude
#i nclude
int Pheponatch(int);
int Pheponatch2(int);
int main()
{
printf("The 10th is %d",Pheponatch2(20));
system("pause");
return 0;
}
//递归算法
int Pheponatch(int N)
{
if( N == 1 || N == 2)
{
return 1;
}
else
{
return Pheponatch(N-1) + Pheponatch(N-2);
}
}
```

```

}
else
return Pheponatch( N -1 ) + Pheponatch( N -2 );
}
//非递归算法
int Pheponatch2(int N)
{
int x = 1, y = 1, temp;
int i = 2;
while(true)
{
temp = y;
y = x + y;
x = temp;
i++;
if( i == N )
break;
}
return y;
}

```

### 25. 完成下列程序

```

*
*.*.
*.*.*..
*...*...*...*...
*....*....*....*...
*.....*.....*.....*...
*.....*.....*.....*.....*...
*.....*.....*.....*.....*.....*...
#include
#define N 8
int main()
{
int i;
int j;
int k;
-----
|||  

|||  

|||
-----
return 0;
}

```

```

#include
#include
#define N 8
int main()
{
    int i;
    int j;
    int k;
    for(i=N; i>=1; i--)
    {
        for(j=0; j < i; j++)
        {
            cout<<"*";
        }
        cout<<"\n";
    }
    return 0;
}

```

"28 下列程序运行时会崩溃, 请找出错误并改正, 并且说明原因。"

```

// void append(int N) ;
//指针没有初始化:
//NewNode->left=NULL;
//NewNode->right=NULL;
#include
#include

typedef struct TNode{
    TNode* left;
    TNode* right;
    int value;
} TNode;

TNode* root=NULL;

void append(int N);

int main()
{
    append(63);
    append(45);
    append(32);
    append(77);
}

```

```

append(96);
append(21);
append(17); // Again, 数字任意给出
return 0;
}

void append(int N)
{
TNode* NewNode=(TNode *)malloc(sizeof(TNode));
NewNode->value=N;
NewNode->left=NULL;
NewNode->right=NULL;
if(root==NULL)
{
root=NewNode;
return;
}
else
{
TNode* temp;
temp=root;
while((N>=temp->value && temp->left!=NULL) || (Nvalue && temp->right!=NULL))
{
while(N>=temp->value && temp->left!=NULL)
temp=temp->left;
while(Nvalue && temp->right!=NULL)
temp=temp->right;
}
if(N>=temp->value)
temp->left=NewNode;
else
temp->right=NewNode;
return;
}
}

```

算法:

1. 什么是 NPC, NP-Hard?
2. 起泡排序的时间复杂度是多少?  
说出至少一个比它更快的算法;  
排序的极限时间复杂度是多少?
3. 有一个链表, 如何判断它是一个循环链表?  
如果链表是单向的呢?  
如果出现循环的点可能在任意位置呢?

如果缓存空间是有限的，比如是一个常数呢？

如果只能使用 2 个缓存呢？

4. 有一个文件，保存了若干个整数，如何以平均的概率随机得到其中的一个整数？

如果整数的个数是未知的呢？

如果整数是以字符串形式存放，如：（即如何得到随机的一个字符串）

123

-456

...

如果只允许遍历文件一次呢？

5. 用两组数据，都在内存中，对它们排序分别需要 1 和 2 分钟；那么使用两个线程一起排序，大概需要多少时间？

C/C++:

1. C 与 C++ 的异同，优劣；
2. C, C++, VC, BC, TC 的区别；
3. C++ 中 `try...catch` 关键字的用法与优点；
4. 枚举的用法，以及它与宏的区别；
5. `const` 的用法，以及声明 `const` 变量与宏的区别；

`const` 的用法有四种：

区别：`const` 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查，而对后者只能进行字符替换，没有类型

安全检查。而且字符替换可能会带来意想不到的边界效应。

有些集成化工具可以对 `const` 常量进行调试，但不能对宏量进行调试。

6. C++ 中引用与指针的区别：

答：1 引用实际上是所引用的对象或变量的别名，而指针是包含所指向对象或变量的地址的变量。

2 引用在定义时必须初始化，而指针在定义时不初始化。

3 不可以有空 `NULL` 的引用，而可以有指向 `NULL` 的指针。

4 引用在初始化后不可以改变引用关系，而指针可以随时指向其他对象（非 `const` 指针）。

7. C++ 中 `virtual` 与 `inline` 的含义分别是什么？

答：在基类成员函数的声明前加上 `virtual` 关键字，意味着将该成员函数声明为虚函数。

`inline` 与函数的定义体放在一起，使该函数称为内联。`inline` 是一种用于实现的关键字，而不是用于声明的关键字。

虚函数的特点：如果希望派生类能够重新定义基类的方法，则在基类中将该方法定义为虚方法，这样可以启用动态联编。

内联函数的特点：使用内联函数的目的是为了提高函数的运行效率。内联函数体的代码不能过长，因为内联函数省去调用函数

的时间是以代码膨胀为代价的。内联函数不能包含循环语句，因为执行循环语句要比调用函数的开销大。

一个函数能否即是虚函数又是内联函数？

8. 以下关键字的含义与用法：

`extern, extern "C", static, explicit, register, #undef, #ifndef`

9. 什么是函数重载与覆盖？

为什么 C 不支持函数重载？

为什么 C++ 能支持函数重载？

10. VC 中，编译工具条内的 `Debug` 与 `Release` 选项是什么含义？

11. 编写 `my_memcpy` 函数, 实现与库函数 `memcpy` 类似的功能, 不能使用任何库函数;

```
void* mymemcpy(void* pvTo, const char* pvFrom, size_t size)
{
    assert((dest != NULL) && (src != NULL));
    byte* psTo = (byte*)pvTo;
    byte* psFrom = (byte*)pvFrom;
    while (size-- > 0)
    {
        *psTo++ = *psFrom++;
    }
    return pvTo;
}
```

12. 编写 `my_strcpy` 函数, 实现与库函数 `strcpy` 类似的功能, 不能使用任何库函数;

答: `char* my_strcpy(char* strdest, const char* strsrc)`

```
{
    assert(strdest != NULL) && (strsrc != NULL)
    char* address = strdest;
    while((*strdest++ = *strsrc++) != NULL)
        return address;
}
```

13. 编写 `gbk_strlen` 函数, 计算含有汉字的字符串的长度, 汉字作为一个字符处理;

已知: 汉字编码为双字节, 其中首字节<0, 尾字节在 0~63 以外; (如果一个字节是-128~127)

14. 函数 `assert` 的用法?

答: 断言 `assert` 是仅在 `debug` 版本起作用的宏, 用于检查“不应该”发生的情况。程序员可以把 `assert` 看成一个

在任何系统状态下都可以安全使用的无害测试手段。

15. 为什么在头文件的最前面都会看到这样的代码:

```
#ifndef _STDIO_H_
#define _STDIO_H_
```

16. 为什么数组名作为参数, 会改变数组的内容, 而其它类型如 `int` 却不会改变变量的值?

答: 当数组名作为参数时, 传递的实际上是地址。而其他类型如 `int` 作为参数时, 由于函数参数值实质上是实参的一份拷贝, 被调

函数内部对形参的改变并不影响实参的值。

1. 实现双向链表删除一个节点 `P`, 在节点 `P` 后插入一个节点, 写出这两个函数。

2. 写一个函数, 将其中的\t 都转换成 4 个空格。

3. Windows 程序的入口是哪里? 写出 Windows 消息机制的流程。

4. 如何定义和实现一个类的成员函数为回调函数?

5. C++里面是不是所有的动作都是 `main()` 引起的? 如果不是, 请举例。

6. C++里面如何声明 `const void f(void)` 函数为 C 程序中的库函数?

7. 下列哪两个是等同的

int b;

A `const int* a = &b;`

B `const* int a = &b;`

C `const int* const a = &b;`

```

D int const* const a = &b;
8. 内联函数在编译时是否做参数类型检查?
void g(base & b){
    b.play;
}
void main(){
    son s;
    g(s);
    return;
}
3、WinMain
while( (bRet = GetMessage( &msg, NULL, 0, 0 )) != 0)
{
if (bRet == -1)
{
// handle the error and possibly exit
}
else
{
TranslateMessage(&msg);
DispatchMessage(&msg);
}
}

```

MSRA Interview Written Exam (December 2003, Time: 2.5 Hours)

1 写出下列算法的时间复杂度。

- (1)冒泡排序;
- (2)选择排序;
- (3)插入排序;
- (4)快速排序;
- (5)堆排序;
- (6)归并排序;

2 写出下列程序在 X86 上的运行结果。

```

struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
}test
void main(void)
{
    int i;
    test.a=2;

```

```
test.b=3;
test.c=0;
i=*((short *)&test);
printf("%d\n",i);
}
```

3 写出下列程序的运行结果。

```
unsigned int i=3;
cout<
```

4 写出下列程序所有可能的运行结果。

```
int a;
int b;
int c;
void F1()
{
b=a*2;
a=b;
}
void F2()
{
c=a+1;
a=c;
}
main()
{
a=5;
//Start F1,F2 in parallel
F1(); F2();
printf("a=%d\n",a);
}
```

5 考察了一个 CharPrev() 函数的作用。

6 对 16 Bits colors 的处理, 要求:

- (1) Byte 转换为 RGB 时, 保留高 5、6bits;
- (2) RGB 转换为 Byte 时, 第 2、3 位置零。

7 一个链表的操作, 注意代码的健壮和安全性。要求:

- (1) 增加一个元素;
- (2) 获得头元素;
- (3) 弹出头元素 (获得值并删除)。

8 一个给定的数值由左边开始升位到右边第 N 位, 如

$0010 \ll 1 == 0100$

或者

$0001\ 0011 \ll 4 == 0011\ 0000$

请用 C 或者 C++ 或者其他 X86 上能运行的程序实现。

附加题 (只有在完成以上题目后, 才获准回答)

In C++, what does "explicit" mean? what does "protected" mean?

## 2、解释 1NF、2NF、3NF、BCNF

第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系；简而言之，第一范式就是无重复的列。

第二范式（2NF）是在第一范式（1NF）的基础上建立起来的，即满足第二范式（2NF）必须先满足第一范式（1NF）。第二范式（2NF）要求数据库表中的每个实例或行必须可以被唯一地区分。为实现区分通常需要为表加上一个列，以存储各个实例的唯一标识。

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。

在第三范式的基础上，数据库表中如果不存在任何字段对任一候选关键字段的传递函数依赖则符合 BCNF。

## 3、如何存储稀疏矩阵

好几种吧，还要看矩阵的类型

## 4、快排序在最好，最坏，平均情况下的时间复杂度与辅助空间复杂度

最好复杂度  $O(n \log 2n)$  最坏  $O(n^2)$  平均  $O(n \log 2n)$

空间：最好  $\log 2n$  最坏  $n$  平均  $\log 2^n$

## 1、不用任何变量交换 a, b 两个变量

（看过...那个加减法吧）

## 2、用递归求最大公约数

```
#include <iostream>
using namespace std;
// 求两个整型数的最大公约数
int gcd( int a, int b )
{
    int temp = 2;           // 公因子，从 2 开始递增直到两个数中最小的一个
    while (temp <= (a < b ? a : b))
    {
        if (a % temp == 0 && b % temp == 0)
        {
            // 都能被整除时递归
            return temp * gcd( a / temp, b / temp );
        }
        else
        {
            // 有一个不能被整除则公因子加一
            temp++;
        }
    }
    return 1;           // 最大公因子为 1
}
```

## 3、举一个动态的例子

（动态...是不是必须要有指针才行啊？）

## 4、二叉平衡树

（...）

## 5、UNIX 进程包括那三个部分:...

（程序,数据和进程控制块 PCB）

6、new 动态分配失败会抛出什么异常,C++中提供了那两个标准函数来设定异常处理 HANLDER

(不知道)

7、EJB 都有那些 Beans?区别

(....)

8、asp 和 asp.net 的区别

(除了编译和解释外...)

9、JAVA 中的 interface 和 abstract class 区别

(....)

10、logic thinking:如何证明一个电冰箱是否是好的。

(....)

内联的定义，什么情况下选择内联。内连接与外连接的区别。

...

=====

1、 $100=99999$

$9 \times 9 + 9 + 9 + 9 / 9$

2、根据 pseudo code 判断下列哪个 x 打印"Text 2"

If ( $x > 4$ ) then print "Text 1"

Else if ( $x > 9$ ) then print "Text 2"

Else print "Text 3"

(1) less than 0,(2) less than 4,(3) between 4 and 9,(4)  $> 9$ , (5) none

5 吧

3、填 bug report:a browser based software crashes when you type "-1" in a input field called ABC on second html page after loggong on on WindowXP platfor

m. And this happens every time you try typing "-1". You are not sure if any database servers are being used or not.

Severity:	Priority:
<hr/>	
Short Description	
<hr/>	<hr/>
Reproducible?	
<hr/>	<hr/>
Recreate steps:	
<hr/>	<hr/>
Attachment:	
<hr/>	
platform tested:	
<hr/>	
database server:	
<hr/>	
browser :	
<hr/>	

---

4、添加注释

```
//  
//  
//  
//  
private static final int SHORT_COLUMN_WIDTH=5;  
private static final int INT_COLUMN_WIDTH=10;  
private static final int LONG_COLUMN_WIDTH=19;  
private static final int DOUBLE_COLUMN_WIDTH=23;  
.....  
int decimalDigits=DataTypeInfo.getScale();  
int widthOfNumber=DataTypeInfo.getPrecision();  
int dataType=DataTypeInfo.numberValue;  
//  
//  
//  
if (decimalDigits==0){  
    if (widthOfNumber<=SHORT_COLUMN_WIDTH) {dataType= dataTypeInfo.signedInt16Value;}  
    else if (widthOfNumber<=INT_COLUMN_WIDTH) {dataType= dataTypeInfo.signedInt32Value;}  
    else if (widthOfNumber<=LONG_COLUMN_WIDTH) {dataType= dataTypeInfo.signedInt64Value;}  
    else if (widthOfNumber<=DOUBLE_COLUMN_WIDTH) {dataType= dataTypeInfo.numberValue;}  
    else {dataType= dataTypeInfo.numberValue;}
```

5、设计一个算法判断一个字符串是否是回文，并写出代码。

"A man a plan a canal panama"是回文。

6、定义函数，给出三个参数，从字符串 `inputString` 中的下标 `indexOfChar` 开始返回 `noOfChar` 个字符。要求找出尽可能多的错误情形

7、编一段代码，求两个 `int` 的最大公约数

8、给出一段 `c++` 代码（关于构建器和虚析构器的调用问题），要求（1）其输出，（2）说明 `virtual destructor` 的作用（role）

9、英文阅读理解。

写一个程序，要求功能：求出用 1, 2, 5 这三个数不同个数组合的和为 100 的组合个数。

如：100 个 1 是一个组合，5 个 1 加 19 个 5 是一个组合。.... 请用 `C++` 语言写。

答案：最容易想到的算法是：

设 `x` 是 1 的个数，`y` 是 2 的个数，`z` 是 5 的个数，`number` 是组合数。注意到  $0 \leq x \leq 100$ ,  $0 \leq y \leq 50$ ,  $0 \leq z \leq 20$ ，所以可以编程为：

```
number=0;  
for (x=0; x<=100; x++)  
for (y=0; y<=50; y++)  
for (z=0; z<=20; z++)
```

```

if ((x+2*y+5*z)==100)
number++;
cout<<number<<endl;

```

上面这个程序一共要循环  $100*50*20$  次，效率实在是太低了事实上，这个题目是一道明显的数学问题，而不是单纯的编程问题。我的解法如下：

因为  $x+2y+5z=100$

所以  $x+2y=100-5z$ ，且  $z \leq 20$   $x \leq 100$   $y \leq 50$

所以  $(x+2y) \leq 100$ ，且  $(x+5z)$  是偶数

对  $z$  作循环，求  $x$  的可能值如下：

$z=0, x=100, 98, 96, \dots, 0$

$z=1, x=95, 93, \dots, 1$

$z=2, x=90, 88, \dots, 0$

$z=3, x=85, 83, \dots, 1$

$z=4, x=80, 78, \dots, 0$

.....

$z=19, x=5, 3, 1$

$z=20, x=0$

因此，组合总数为 100 以内的偶数+95 以内的奇数+90 以内的偶数+...+5 以内的奇数+1，

即为：

$(51+48)+(46+43)+(41+38)+(36+33)+(31+28)+(26+23)+(21+18)+(16+13)+(11+8)+(6+3)+1$

某个偶数  $m$  以内的偶数个数（包括 0）可以表示为  $m/2+1=(m+2)/2$

某个奇数  $m$  以内的奇数个数也可以表示为  $(m+2)/2$

所以，求总的组合次数可以编程为：

```

number=0;
for (int m=0;m<=100;m+=5)
{
    number+=(m+2)/2;
}
cout<<number<<endl;

```

这个程序，只需要循环 21 次，两个变量，就可以得到答案，比上面的那个程序高效了许多倍——只是因为作了一些简单的数学分析。

这再一次证明了：计算机程序=数据结构+算法，而且算法是程序的灵魂，对任何工程问题，当用软件来实现时，必须选取满足当前的资源限制，用户需求限制，开发时间限制等种种限制条件下的最优算法。而绝不能一拿到手，就立刻用最容易想到的算法编出一个程序了事——这不是一个专业的研发人员的行为。

那么，那种最容易想到的算法就完全没用吗？不，这种算法正好可以用来验证新算法的正确性，在调试阶段，这非常有用。在很多大公司，例如微软，都采用了这种方法：在调试阶段，对一些重要的需要好的算法来实现的程序，而这种好的算法又比较复杂时，同时用容易想到的算法来验证这段程序，如果两种算法得出的结果不一致（而最容易想到的算法保证是正确的），那么说明优化的算法出了问题，需要修改。

可以举例表示为：

```
#ifdef DEBUG
int simple();
#endif if
int optimize();
.....
in a function:
{
result=optimize();
ASSERT(result==simple());
}
```

这样，在调试阶段，如果简单算法和优化算法的结果不一致，就会打出断言。同时，在程序的发布版本，却不会包含笨重的 simple() 函数。——任何大型工程软件都需要预先设计良好的调试手段，而这里提到的就是一种有用的方法。

一个学生的信息是：姓名，学号，性别，年龄等信息，用一个链表，把这些学生信息连在一起，给出一个 age，在些链表中删除学生年龄等

于 age 的学生信息。

```
#include "stdio.h"
#include "conio.h"

struct stu{
char name[20];
char sex;
int no;
int age;
struct stu * next;
}*linklist;
struct stu *creatlist(int n)
{
int i;
//h 为头结点， p 为前一结点， s 为当前结点
struct stu *h,*p,*s;
h = (struct stu *)malloc(sizeof(struct stu));
h->next = NULL;
p=h;
```

```

for(i=0;i<n;i++)
{
    s = (struct stu *)malloc(sizeof(struct stu));
    p->next = s;
    printf("Please input the information of the student: name sex no age \n");
    scanf("%s %c %d %d", s->name, &s->sex, &s->no, &s->age);
    s->next = NULL;
    p = s;
}
printf("Create successful!");
return(h);
}

void deletelist(struct stu *s, int a)
{
    struct stu *p;
    while(s->age!=a)
    {
        p = s;
        s = s->next;
    }
    if(s==NULL)
        printf("The record is not exist.");
    else
    {
        p->next = s->next;
        printf("Delete successful!");
    }
}

void display(struct stu *s)
{
    s = s->next;
    while(s!=NULL)
    {
        printf("%s %c %d %d\n", s->name, s->sex, s->no, s->age);
        s = s->next;
    }
}

int main()
{
    struct stu *s;
    int n, age;
    printf("Please input the length of seqlist:\n");
    scanf("%d", &n);
    s = creatlist(n);
}

```

```

display(s);
printf("Please input the age:\n");
scanf("%d", &age);
deletelist(s, age);
display(s);
return 0;
}

```

2、实现一个函数，把一个字符串中的字符从小写转为大写。

```

#include "stdio.h"
#include "conio.h"

void uppers(char *s, char *us)
{
    for(;*s!='\0' ;s++,us++)
    {
        if(*s>='a' &&*s<='z')
            *us = *s-32;
        else
            *us = *s;
    }
    *us = '\0';
}

int main()
{
    char *s, *us;
    char ss[20];
    printf("Please input a string:\n");
    scanf("%s", ss);
    s = ss;
    uppers(s, us);
    printf("The result is:\n%s\n", us);
    getch();
}

```

随机输入一个数，判断它是不是对称数（回文数）（如3, 121, 12321, 45254）。不能用字符串库函数

```

*****
1.
函数名称: Symmetry
功能: 判断一个数时候为回文数(121, 35653)
输入: 长整型的数
输出: 若为回文数返回值为1 esle 0
*****

```

```

unsigned char Symmetry (long n)
{
    long i, temp;
    i=n; temp=0;
    while(i) //不用出现长度问题,将数按高低位掉换
    {
        temp=temp*10+i%10;
        i/=10;
    }
    return(temp==n);
}

方法一
/* -----
功能:
判断字符串是否为回文数字
实现:
先将字符串转换为正整数,再将正整数逆序组合为新的正整数,两数相同则为回文数字
输入:
char *s: 待判断的字符串
输出:
无
返回:
0: 正确; 1: 待判断的字符串为空; 2: 待判断的字符串不为数字;
3: 字符串不为回文数字; 4: 待判断的字符串溢出
----- */
unsigned IsSymmetry(char *s)
{
    char *p = s;
    long nNumber = 0;
    long n = 0;
    long nTemp = 0;

    /*判断输入是否为空*/
    if (*s == '\0')
        return 1;

    /*将字符串转换为正整数*/
    while (*p != '\0')
    {
        /*判断字符是否为数字*/
        if (*p<'0' || *p>'9')
            return 2;

```

```

/*判断正整数是否溢出*/
if ((*p-\'0\') > (4294967295-(nNumber*10)))
return 4;

nNumber = (*p-\'0\') + (nNumber * 10);

p++;
}

/*将数字逆序组合, 直接抄楼上高手的代码, 莫怪, 呵呵*/
n = nNumber;
while(n)
{
/*判断正整数是否溢出*/
if ((n%10) > (4294967295-(nTemp*10)))
return 3;

nTemp = nTemp*10 + n%10;
n /= 10;
}

/*比较逆序数和原序数是否相等*/
if (nNumber != nTemp)
return 3;

return 0;
}

```

方法二

---

功能:  
判断字符串是否为回文数字  
实现:  
先得到字符串的长度, 再依次比较字符串的对应位字符是否相同  
输入:  
char \*s: 待判断的字符串  
输出:  
无  
返回:  
0: 正确; 1: 待判断的字符串为空; 2: 待判断的字符串不为数字;  
3: 字符串不为回文数字

---

unsigned IsSymmetry\_2(char \*s)
{
char \*p = s;

```

int nLen = 0;
int i = 0;

/*判断输入是否为空*/
if (*s == '\0')
return 1;

/*得到字符串长度*/
while (*p != '\0')
{
/*判断字符是否为数字*/
if (*p<'0' || *p>'9')
return 2;

nLen++;
p++;
}

/*长度不为奇数, 不为回文数字*/
if (nLen%2 == 0)
return 4;

/*长度为 1, 即为回文数字*/
if (nLen == 1)
return 0;

/*依次比较对应字符是否相同*/
p = s;
i = nLen/2 - 1;
while (i)
{
if (*(p+i) != *(p+nLen-i-1))
return 3;

i--;
}

return 0;
}

```

求  $2^{2000}$  的所有素数, 有足够的内存, 要求尽量快

答案:

```
int findvalue[2000]={2};
```

```

static int find=1;
bool adjust(int value)
{
    assert(value>=2);
    if(value==2) return true;
    for(int i=0;i<=find;i++)
    {
        if(value%findvalue[i]==0)
            return false;
    }
    findvalue[find]++;
    return true;
}

```

1.引言 本文的写作目的并不在于提供 C/C++ 程序员求职面试指导，而旨在从技术上分析面试题的内涵。文中的大多数面试题来自各大论坛，部分试题解答也参考了网友的意见。许多面试题看似简单，却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 `strcpy` 函数都可看出面试者在技术上究竟达到了怎样的程度，我们能真正写好一个 `strcpy` 函数吗？我们都觉得自己能，可是我们写出的 `strcpy` 很可能只能拿到 10 分中的 2 分。读者可从本文看到 `strcpy` 函数从 2 分到 10 分解答的例子，看看自己属于什么样的层次。此外，还有一些面试题考查面试者敏捷的思维能力。 分析这些面试题，本身包含很强的趣味性；而作为一名研发人员，通过对这些面试题的深入剖析则可进一步增强自身的内功。

2. 找错题 试题 1： `void test1(){ char string[10]; char* str1 = "0123456789"; strcpy( string, str1 );}` 试题 2： `void test2(){ char string[10], str1[10]; int i; for(i=0; i<10; i++) { str1[i] = 'a'; } strcpy( string, str1 );}` 试题 3： `void test3(char* str1){ char string[10]; if( strlen( str1 ) <= 10 ) { strcpy( string, str1 ); } }` 解答：

试题 1 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； 对试题 2，如果面试者指出字符数组 `str1` 不能在数组内结束可以给 3 分；如果面试者指出 `strcpy(string, str1)` 调用使得从 `str1` 内存起复制到 `string` 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 `strcpy` 工作方式的给 10 分； 对试题 3，`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。 剖析： 考查对基本功的掌握：

(1) 字符串以'\0'结尾； (2) 对数组越界把握的敏感度； (3) 库函数 `strcpy` 的工作方式，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 `void strcpy( char *strDest, char *strSrc ){ while( (*strDest++ = * strSrc++) != '\0' );}` 4 分 `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 `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 `! char * strcpy( char *strDest, const char *strSrc ) { assert( (strDest != NULL) && (strSrc != NULL) ); char *address = strDest; while( (*strDest++ = * strSrc++) != '\0' ); return address;}` 从 2 分到 10 分的几个答案我们可以清楚的看到，小小的 `strcpy` 竟然暗藏着这么多玄机，真不是盖的！需要多么扎实的基本功才能写一个完美的 `strcpy` 啊！ (4) 对 `strlen` 的掌握，它没有包括字符串末

尾的'\0'. 读者看了不同分值的 `strcpy` 版本, 应该也可以写出一个 10 分的 `strlen` 函数了, 完美的版本为: `int strlen( const char *str ) //输入参数 const{ assert( str != NULL ); //断言字符串地址非 0 int len; while( (*str++) != '\0' ) { len++; } return len;}`

试题 4: `void GetMemory( char *p ) { p = (char *) malloc( 100 ); } void Test( void ) { char *str = NULL; GetMemory( str ); strcpy( str, "hello world" ); printf( str ); } 试题 5: char *GetMemory( void ) { char p[] = "hello world"; return p; } void Test( void ) { char *str = NULL; str = GetMemory(); printf( str ); }` 试题 6: `void GetMemory( char **p, int num ) { *p = (char *) malloc( num ); } void Test( void ) { char *str = NULL; GetMemory( &str, 100 ); strcpy( str, "hello" ); printf( str ); }` 试题 7: `void Test( void ) { char *str = (char *) malloc( 100 ); strcpy( str, "hello" ); free( str ); ... //省略的其它语句 }` 解答: 试题 4 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值, 执行完 `char *str = NULL; GetMemory( str );` 后的 `str` 仍然为 `NULL`; 试题 5 中 `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。试题 6 的 `GetMemory` 避免了试题 4 的问题, 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 `*p = (char *) malloc( num );` 后未判断内存是否申请成功, 应加上: `if ( *p == NULL ) { ... //进行申请内存失败处理 }` 试题 7 存在与试题 6 同样的问题, 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。剖析: 试题 4~7 考查面试者对内存操作的理解程度, 基本功扎实的面试者一般都能正确的回答其中 50~60 的错误, 但是要完全解答正确, 却也绝非易事。对内存操作的考查主要集中在: (1) 指针的理解; (2) 变量的生存期及作用范围; (3) 良好的动态内存申请和释放习惯。再看看下面的一段程序有什么错误: `swap( int* p1, int* p2 ) { int *p; *p = *p1; *p1 = *p2; *p2 = *p; }` 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: `swap( int* p1, int* p2 ) { int p; p = *p1; *p1 = *p2; *p2 = p; }`

## 1.引言

本文的写作目的并不在于提供 C/C++ 程序员求职面试指导, 而旨在从技术上分析面试题的内涵。文中的大多数面试题来自各大论坛, 部分试题解答也参考了网友的意见。

许多面试题看似简单, 却需要深厚的基本功才能给出完美的解答。企业要求面试者写一个最简单的 `strcpy` 函数都可看出面试者在技术上究竟达到了怎样的程度, 我们能真正写好一个 `strcpy` 函数吗? 我们都觉得自己能, 可是我们写出的 `strcpy` 很可能只能拿到 10 分中的 2 分。读者可从本文看到 `strcpy` 函数从 2 分到 10 分解答的例子, 看看自己属于什么样的层次。此外, 还有一些面试题考查面试者敏捷的思维能力。

分析这些面试题，本身包含很强的趣味性；而作为一名研发人员，通过对这些面试题的深入剖析则可进一步增强自身的内功。

## 2. 找错题

试题 1：

```
void test1()
{
    char string[10];
    char* str1 = "0123456789";
    strcpy( string, str1 );
}
```

试题 2：

```
void test2()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}
```

试题 3：

```
void test3(char* str1)
{
    char string[10];
    if( strlen( str1 ) <= 10 )
    {
```

```
    strcpy( string, str1 );
}
}
```

解答：

试题 1 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；

对试题 2，如果面试者指出字符数组 str1 不能在数组内结束可以给 3 分；如果面试者指出 strcpy(string, str1) 调用使得从 str1 内存起复制到 string 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 strcpy 工作方式的给 10 分；

对试题 3, 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。

剖析：

考查对基本功的掌握：

(1)字符串以'\0'结尾；

(2)对数组越界把握的敏感度；

(3)库函数 strcpy 的工作方式，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

```
void strcpy( char *strDest, char *strSrc )
{
    while( (*strDest++ = * strSrc++) != '\0' );
}
```

4 分

```
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

```
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!

char * strcpy( char *strDest, const char *strSrc )
{
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
    while( (*strDest++ = * strSrc++) != '\0' );
    return address;
}
```

从 2 分到 10 分的几个答案我们可以清楚的看到, 小小的 `strcpy` 竟然暗藏着这么多玄机, 真不是盖的! 需要多么扎实的基本功才能写一个完美的 `strcpy` 啊!

(4)对 `strlen` 的掌握, 它没有包括字符串末尾的'\0'。

读者看了不同分值的 `strcpy` 版本，应该也可以写出一个 10 分的 `strlen` 函数了，完美的版本为： `int strlen( const char *str ) //输入参数 const`

```
{  
    assert( str != NULL ); //断言字符串地址非 0  
    int len;  
    while( (*str++) != '\0' )  
    {  
        len++;  
    }  
    return len;  
}
```

试题 4：

```
void GetMemory( char *p )  
{  
    p = (char *) malloc( 100 );  
}  
  
void Test( void )  
{  
    char *str = NULL;  
    GetMemory( str );  
    strcpy( str, "hello world" );  
    printf( str );  
}
```

试题 5：

```
char *GetMemory( void )  
{  
    char p[] = "hello world";  
    return p;
```

```
}
```

  

```
void Test( void )
```

```
{
```

```
    char *str = NULL;
```

```
    str = GetMemory();
```

```
    printf( str );
```

```
}
```

试题 6:

```
void GetMemory( char **p, int num )
```

```
{
```

```
    *p = (char *) malloc( num );
```

```
}
```

  

```
void Test( void )
```

```
{
```

```
    char *str = NULL;
```

```
    GetMemory( &str, 100 );
```

```
    strcpy( str, "hello" );
```

```
    printf( str );
```

```
}
```

试题 7:

```
void Test( void )
```

```
{
```

```
    char *str = (char *) malloc( 100 );
```

```
    strcpy( str, "hello" );
```

```
    free( str );
```

```
    ... //省略的其它语句
```

```
}
```

解答：

试题 4 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值，执行完

```
char *str = NULL;  
GetMemory( str );
```

后的 `str` 仍然为 `NULL`；

试题 5 中

```
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。

试题 6 的 `GetMemory` 避免了试题 4 的问题，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

```
*p = (char *) malloc( num );
```

后未判断内存是否申请成功，应加上：

```
if( *p == NULL )  
{  
    ...//进行申请内存失败处理  
}
```

试题 7 存在与试题 6 同样的问题，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。

剖析：

试题 4~7 考查面试者对内存操作的理解程度，基本功扎实的面试者一般都能正确的回答其中 50~60 的错误。但是要完全解答正确，却也绝非易事。

对内存操作的考查主要集中在：

- (1) 指针的理解；
- (2) 变量的生存期及作用范围；
- (3) 良好的动态内存申请和释放习惯。

再看看下面的一段程序有什么错误：

```
swap( int* p1,int* p2 )  
{  
    int *p;  
    *p = *p1;  
    *p1 = *p2;  
    *p2 = *p;  
}
```

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：

```
swap( int* p1,int* p2 )
```

```
{  
    int p;  
    p = *p1;  
    *p1 = *p2;  
    *p2 = p;  
}
```

### 3. 内功题

试题 1：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**）

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。

试题 2：以下为 Windows NT 下的 32 位 C++ 程序，请计算 sizeof 的值

```
void Func ( char str[100] )
{
    sizeof( str ) = ?
}

void *p = malloc( 100 );
sizeof ( p ) = ?
```

解答：

```
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：

```
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。

试题 3：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);
```

解答：

```
#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”和整个宏用括弧括起来。所以，严格地讲，下述解答：

```
#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：

```
(*p++) <= (b) ? (*p++) : (*p++)
```

这个表达式会产生副作用，指针 p 会作三次++自增操作。

除此之外，另一个应该判 0 分的解答是：

```
#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。

试题 4：为什么标准头文件都有类似以下的结构？

```
#ifndef __INCvxWorksh  
#define __INCvxWorksh  
#ifdef __cplusplus  
  
extern "C" {  
#endif
```

```
/*...*/  
#ifdef __cplusplus  
}  
  
#endif  
#endif /* __INCvxWorksh */
```

解答：

头文件中的编译宏

```
#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++就是考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。

试题5：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“abcdefghijklm”如果n=2，移位后应该是“mabcdefghijkl”

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
```

```
void LoopMove ( char * pStr, int steps )  
{  
    //请填充...  
}
```

解答：

正确解答 1：

```
void LoopMove ( char *pStr, int steps )  
{  
    int n = strlen( pStr ) - steps;  
    char tmp[MAX_LEN];  
    strcpy ( tmp, pStr + n );  
    strcpy ( tmp + steps, pStr );  
    *( tmp + strlen ( pStr ) ) = '\0';  
    strcpy( pStr, tmp );  
}
```

正确解答 2：

```
void LoopMove ( char *pStr, int steps )  
{  
    int n = strlen( pStr ) - steps;  
    char tmp[MAX_LEN];  
    memcpy( tmp, pStr + n, steps );  
    memcpy(pStr + steps, pStr, n );  
    memcpy(pStr, tmp, steps );  
}
```

剖析：

这个试题主要考查面试者对标准库函数的熟练程度，在需要的时候引用库函数可以很大程度上简化程序编写的工作量。

最频繁被使用的库函数包括：

(1) `strcpy`

(2) `memcpy`

(3) `memset`

试题 6：已知 **WAV** 文件格式如下表，打开一个 **WAV** 文件，以适当的数据结构组织 **WAV** 文件头并解析 **WAV** 格式的各项信息。

**WAVE** 文件格式说明表

	偏移地址	字节数	数据类型	内 容
文件头	00H	4	Char	"RIFF"标志
	04H	4	int32	文件长度
	08H	4	Char	"WAVE"标志
	0CH	4	Char	"fmt"标志
	10H	4		过渡字节（不定）
	14H	2	int16	格式类别
	16H	2	int16	通道数
	18H	2	int16	采样率（每秒样本数），表示每个通道的播放速度
	1CH	4	int32	波形音频数据传送速率
	20H	2	int16	数据块的调整数（按字节算的）
	22H	2		每样本的数据位数

24H	4	Char	数据标记符 " data "
28H	4	int32	语音数据的长度

解答：

将 WAV 文件格式定义为结构体 WAVEFORMAT：

```
typedef struct tagWaveFormat
{
    char cRiffFlag[4];
    UIN32 nFileLen;
    char cWaveFlag[4];
    char cFmtFlag[4];
    char cTransition[4];
    UIN16 nFormatTag ;
    UIN16 nChannels;
    UIN16 nSamplesPerSec;
    UIN32 nAvgBytesperSec;
    UIN16 nBlockAlign;
    UIN16 nBitNumPerSample;
    char cDataFlag[4];
    UIN16 nAudioLength;

} WAVEFORMAT;
```

假设 WAV 文件内容读出后存放在指针 **buffer** 开始的内存单元内，则分析文件格式的代码很简单，为：

```
WAVEFORMAT waveFormat;
memcpy( &waveFormat, buffer,sizeof( WAVEFORMAT ) );
```

直接通过访问 **waveFormat** 的成员，就可以获得特定 WAV 文件的各项格式信息。

剖析：

试题 6 考查面试者组织数据结构的能力，有经验的程序设计者将属于一个整体的数据成员组织为一个结构体，利用指针类型转换，可以将 `memcpy`、`memset` 等函数直接用于结构体地址，进行结构体的整体操作。透过这个题可以看出面试者的程序设计经验是否丰富。

试题 7：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：

```
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
};
```

解答：

```
//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]; // 若能加 NULL 判断则更好
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];          //加分点: 对 m_data 加 NULL 判断
    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];  //加分点: 对 m_data 加 NULL 判断
    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！

试题 8：请说出 **static** 和 **const** 关键字尽可能多的作用

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。

剖析：

惊讶吗？小小的 `static` 和 `const` 居然有这么多功能，我们能回答几个？如果只能回答 1~2 个，那还真得闭关再好好修炼修炼。

这个题可以考查面试者对程序设计知识的掌握程度是初级、中级还是比较深入，没有一定的知识广度和深度，不可能对这个问题给出全面的解答。大多数人只能回答出 **static** 和 **const** 关键字的部分功能。

#### 4. 技巧题

试题 1：请写一个 C 函数，若处理器是 **Big\_endian** 的，则返回 0；若是 **Little\_endian** 的，则返回 1

解答：

```
int checkCPU()
{
{
    union w
    {
        int a;
        char b;
    } c;
    c.a = 1;
    return (c.b == 1);
}
```

剖析：

嵌入式系统开发者应该对 **Little-endian** 和 **Big-endian** 模式非常了解。采用 **Little-endian** 模式的 CPU 对操作数的存放方式是从低字节到高字节，而 **Big-endian** 模式对操作数的存放方式是从高字节到低字节。例如，16bit 宽的数 0x1234 在 **Little-endian** 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	存放内容
0x4000	0x34

0x4001	0x12
--------	------

而在 **Big-endian** 模式 CPU 内存中的存放方式则为：

内存地址	存放内容
0x4000	0x12
0x4001	0x34

32bit 宽的数 0x12345678 在 **Little-endian** 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	存放内容
0x4000	0x78
0x4001	0x56
0x4002	0x34
0x4003	0x12

而在 **Big-endian** 模式 CPU 内存中的存放方式则为：

内存地址	存放内容
0x4000	0x12
0x4001	0x34
0x4002	0x56
0x4003	0x78

联合体 **union** 的存放顺序是所有成员都从低地址开始存放，面试者的解答利用该特性，轻松地获得了 CPU 对内存采用 **Little-endian** 还是 **Big-endian** 模式读写。如果谁能当场给出这个解答，那简直就是一个天才的程序员。

试题 2：写一个函数返回  $1+2+3+\dots+n$  的值（假定结果不会超过长整型变量的范围）

解答：

```
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 ( 1l + n ) * n / 2` 相比！

```
int Sum( int n )
{
    long sum = 0;
    for( int i=1; i<=n; i++ )
    {
        sum += i;
    }
    return sum;
}
```

所以程序员们需要敏感地将数学等知识用在程序设计中。

Trackback: <http://tb.blog.csdn.net/TrackBack.aspx?PostId=1238902>

[\[收藏到我的网摘\]](#) zhaoyawei 发表于 2006 年 09 月 18 日 17:52:00

相关文章：

- [声明函数指针并实现回调 2001-02-10 jeffreyren](#)

- [函数指针 2006-04-22 kingswood](#)
- [声明与函数、函数指针 2006-03-14 norbe](#)
- [c++语言程序设计----学习笔记\(3\) 2005-02-07 youki1234](#)
- [函数指针 point of function 2004-09-08 leisureful](#)

特别推荐：

- [linux,C++人才哪去了.](#)  
趋势科技—网络安全软件及服务领域的全球领导者 趋势中国研发中心诚聘英才 [c](#)
- [浙江支付宝,中国的硅谷,开发者的天堂](#)  
支付宝招聘英才 员工将拥有具竞争力的薪金及众多的培训机会 [c](#)
- [青牛软件招聘软件工程师](#)  
青牛（北京）技术有限公司（简称青牛软件）是中国领 [c](#)
- [上海日资软件企业招聘](#)  
上海交大海外 AIC 学院为你打通高薪之路--25 万年 [c](#)
- [主机完全 DIY,域名免费试用](#)  
时代互联 100M 主机 216 元/年 [c](#)

# Crazyazreal 发表于 2006-09-21 00:46:00 IP: 218.13.199.\*

看完！发现自己能完成 50%，嘿``继续努力！楼主多发这类文章啊！支持！

# 程先 发表于 2006-09-23 13:22:00 IP: 218.65.61.\*

//为了实现链式操作，将目的地址返回，加 3 分！

```
char * strcpy( char *strDest, const char *strSrc )
{
    assert( (strDest != NULL) && (strSrc != NULL) );
    char *address = strDest;
    while( (*strDest++ = * strSrc++) != '\0' );
    return address;
}
```

这个给 10 分有失偏颇，主管定的？后面就没继续看了，扯的！

# lz 发表于 2006-09-25 11:39:00 IP: 60.208.111.\*

应该是主管定的，因为，也完全不这样折腾的哈。

可以 `return`，也可以就那么调用一哈嘛……

# sophisticated 发表于 2006-09-28 09:34:00 IP: 221.250.216.\*

```
void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    strcpy ( tmp, pStr + n );
    strcpy ( tmp + steps, pStr );
    *( tmp + strlen ( pStr ) ) = '\0';
    strcpy( pStr, tmp );
}
```

---

steps > strlen( pStr ) 时会有什么后果？

# sophisticated 发表于 2006-09-28 09:48:00 IP: 221.250.216.\*

```
void LoopMove ( char *pStr, int steps )
{
    int n = strlen( pStr ) - steps;
    char tmp[MAX_LEN];
    strcpy ( tmp, pStr + n );
    strcpy ( tmp + steps, pStr );
    *( tmp + strlen ( pStr ) ) = '\0';
    strcpy( pStr, tmp );
```

1. 下面这段代码的输出是多少(在 32 位机上).

```
char *p;
char *q[20];
char *m[20][20];
int (*n)[10];
```

```
struct MyStruct

{
    char dda;
    double dda1;
    int type ;
};

MyStruct k;

printf("%d %d %d %d",sizeof(p),sizeof(q),sizeof(m),sizeof(n),sizeof(k));
```

2.

(1)

```
char a[2][2][3]={{1,6,3},{5,4,15},{3,5,33},{23,12,7}} ;
for(int i=0;i<12;i++)
printf("%d ",_____);
```

在空格处填上合适的语句，顺序打印出 a 中的数字

(2)

```
char **p, a[16][8];
```

问： p=a 是否会导致程序在以后出现问题？为什么？

3.用递归方式,非递归方式写函数将一个字符串反转.

函数原型如下:char \*reverse(char \*str);

4.strcpy 函数和 memcpy 函数有什么区别？它们各自使用时应该注意什么问题？

5.写一个函数将一个链表逆序.

一个单链表，不知道长度，写一个函数快速找到中间节点的位置.

写一个函数找出一个单向链表的倒数第 **n** 个节点的指针.(把能想到的最好算法写出).

**6.用递归算法判断数组 **a[N]**是否为一个递增数组。**

**7.**

有一个文件(名为 **a.txt**)如下,每行有 **4** 项,第一项是他们的名次,写一个 **c** 程序,将五个人的名字打印出来.并按名次排序后将 **5** 行数据仍然保存到 **a.txt** 中.使文件按名次排列每行.

**2,07010188,0711,李镇豪,**  
**1,07010154,0421,陈亦良,**  
**3,07010194,0312,凌瑞松,**  
**4,07010209,0351,罗安祥,**  
**5,07010237,0961,黄世传,**

**8.写一个函数,判断一个 **unsigned char** 字符有几位是 **1**.**

写一个函数判断计算机的字节存储顺序是升序(**little-endian**)还是降序(**big-endian**).

**9.微软的笔试题.**

**Implement a string class in C++ with basic functionality like comparison, concatenation, input and output. Please also provide some test cases and using scenarios (sample code of using this class).**

**Please do not use MFC, STL and other libraries in your implementation.**

**10.有个数组 **a[100]**存放了 **100** 个数,这 **100** 个数取自 **1-99**,且只有两个相同的数,剩下的 **98** 个数不同,写一个搜索算法找出相同的那个数的值.(注意空间效率时间效率尽可能要低).**

这十道题还是能够看出自己的水平如何的.如果你能不假思索地做出这 **10** 道题,估计去国外大公司是没有问题了,呵呵.

答案我在整理中,以后陆续发布.....

下面有些题也不错,可以参考.

1. 下面的代码输出是什么, 为什么?

```
void foo(void)
{
    unsigned int a = 6;
    int b = -20;
    (a+b>6)?puts(">6"):puts("<=6");//puts 为打印函数
}
```

输出 >6.

就是考察隐式转换. int 型变量转化成 unsigned int, b 成了正数.

2. b)运行下面的函数会有什么结果? 为什么?

```
void foo(void)
{
    char string[10],str1[10];
    int i;
    for(i=0;i<10;i++)
    {
        str1[i] = 'a';
    }
    strcpy(string, str1);
    printf("%s",string);
}
```

首先搞清 strcpy 函数的实现方法,

```
char * strcpy(char * strDest,const char * strSrc)
{
    if ((strDest == NULL) || (strSrc == NULL))
        throw "Invalid argument(s)";
    char * strDestCopy = strDest;
    while ((*strDest++ = *strSrc++) != '\0');
    return strDestCopy;
}
```

由于 `str1` 末尾没有'\0'结束标志，所以 `strcpy` 不知道拷贝到何时结束。  
`printf` 函数，对于输出 `char*` 类型，顺序打印字符串中的字符直到遇到空字符 (' \ 0 ') 或已打印了由精度指定的字符数为止。

下面是微软的两道笔试题....

3. Implement a string class in C++ with basic functionality like comparison, concatenation, input and output. Please also provide some test cases and using scenarios (sample code of using this class).

Please do not use MFC, STL and other libraries in your implementation.

我的实现方案如下,这道题真地对 c++的主要特性都进行了较好地考察.

String.h:

```
#ifndef STRING_H
#define STRING_H

#include <iostream>
using namespace std;

class String{
public:
    String();
    String(int n,char c);
    String(const char* source);
    String(const String& s);
    //String& operator=(char* s);
    String& operator=(const String& s);
    ~String();

    char& operator[](int i){return a[i];}
    const char& operator[](int i) const {return a[i];}//对常量的索引.
    String& operator+=(const String& s);
    int length();
```

```
friend istream& operator>>(istream& is, String& s);//搞清为什么将>>设置为友元函数的原因.
```

```
//friend bool operator< (const String& left,const String& right);  
friend bool operator> (const String& left, const String& right);//下面三个运算符都没必要  
设成友元函数,这里是为了简单.  
friend bool operator==(const String& left, const String& right);  
friend bool operator!=(const String& left, const String& right);  
private:  
char* a;  
int size;  
};  
  
#endif
```

String.cpp:

```
#include "String.h"  
#include <cstring>  
#include <cstdlib>  
  
String::String(){  
a = new char[1];  
a[0] = '\0';  
size = 0;  
}  
  
String::String(int n,char c){  
a = new char[n + 1];  
memset(a,c,n);  
a[n] = '\0';  
size = n;  
}
```

```
String::String(const char* source){  
    if(source == NULL){  
        a = new char[1];  
        a[0] = '\0';  
        size = 0;  
    }  
    else  
    {    size = strlen(source);  
        a = new char[size + 1];  
        strcpy(a,source);  
    }  
}
```

```
String::String(const String& s){  
    size = strlen(s.a);//可以访问私有变量.  
    a = new char[size + 1];  
    //if(a == NULL)  
    strcpy(a,s.a);  
}
```

```
String& String::operator=(const String& s){  
    if(this == &s)  
        return *this;  
    else  
    {  
        delete[] a;  
        size = strlen(s.a);  
        a = new char[size + 1];  
        strcpy(a,s.a);  
        return *this;  
    }  
}  
String::~String(){
```

```

    delete[] a;//
}

String& String::operator+=(const String& s){
    int j = strlen(a);
    int size = j + strlen(s.a);
    char* tmp = new char[size+1];
    strcpy(tmp,a);
    strcpy(tmp+j,s.a);
    delete[] a;
    a = tmp;

    return *this;
}

```

```

int String::length(){
    return strlen(a);
}

```

main.cpp:

```

#include <iostream>
#include "String.h"

using namespace std;

bool operator==(const String& left, const String& right)
{
    int a = strcmp(left.a,right.a);
    if(a == 0)
        return true;
    else
        return false;
}

bool operator!=(const String& left, const String& right)
{

```

```
    return !(left == right);
}

ostream& operator<<(ostream& os, String& s){
    int length = s.length();
    for(int i = 0; i < length; i++)
        //os << s.a[i]; 这么不行,私有变量.
        os << s[i];
    return os;
}
```

```
String operator+(const String& a, const String& b){
    String temp;
    temp = a;
    temp += b;
    return temp;
}
```

```
bool operator<(const String& left, const String& right){

    int j = 0;
    while((left[j] != '\0') && (right[j] != '\0')){
        if(left[j] < right[j])
            return true;
        else
        {
            if(left[j] == right[j]){
                j++;
                continue;
            }
            else
                return false;
        }
    }
}
```

```

    }

    if((left[j] == '\0') && (right[j] != '\0'))
        return true;
    else
        return false;
    }

bool operator>(const String& left, const String& right)
{
    int a = strcmp(left.a,right.a);

    if(a > 0)
        return true;
    else
        return false;
}

istream& operator>>(istream& is, String& s){

    delete[] s.a;
    s.a = new char[20];
    int m = 20;
    char c;
    int i = 0;
    while (is.get(c) && isspace(c));
    if (is) {
        do {s.a[i] = c;
            i++;
            /*if(i >= 20){
                cout << "Input too much characters!" << endl;
                exit(-1);
            }*/
        if(i == m - 1 ){
            s.a[i] = '\0';
            char* b = new char[m];
            strcpy(b,s.a);
            m = m * 2;
        }
    }
}

```

```

s.a = new char[m];
strcpy(s.a,b);
delete[] b;
}
}

while (is.get(c) && !isspace(c));
//如果读到空白,将其放回.

if (is)
is.unget();
}

s.size = i;
s.a[i] = '\0';
return is;
}

```

```

int main(){
String a = "abcd";
String b = "www";
//String c(6,b);这么写不对.
String c(6,'l');
String d;
String e = a;//abcd
String f;
cin >> f;//需要输入...
String g;
g = a + b;//abcdwww

if(a < b)
cout << "a < b" << endl;
else
cout << "a >= b" << endl;
if(e == a)
cout << "e == a" << endl;
else

```

```
cout << "e != a" << endl;

b += a;

cout << a << endl;
cout << b << endl;
cout << c << endl;
cout << d << endl;
cout << e << endl;
cout << f << endl;
cout << g << endl;
cout << g[0] << endl;
return 0;
}
```

4. Implement a single-direction linked list sorting algorithm. Please first define the data structure of linked list and then implement the sorting algorithm.

5. 编写一个函数，返回两个字符串的最大公串！例如，“adbccadebbca”和“edabccadece”，返回“ccade”

### 联想笔试题

1. 设计函数 `int atoi(char *s)`。

`int atoi(const char *nptr);`

函数说明

`atoi()`会扫描参数 `nptr` 字符串，跳过前面的空格字符，直到遇上数字或正负符号才开始做转换，而再遇到非数字或字符串结束时('0')才结束转换，并将结果返回。

返回值 返回转换后的整型数。

```

#include <stdio.h>
#include <ctype.h>

int myAtoi(const char* s){
    int result = 0;
    int flag = 1;
    int i = 0;
    while(isspace(s[i]))
        i++;
    if(s[i] == '-'){
        flag = -1;
        i++;
    }
    if(s[i] == '+')
        i++;
    while(s[i] != '\0'){
        if((s[i] > '9') || (s[i] < '0'))
            break;
        int j = s[i] - '0';
        result = 10 * result + j;
        i++;
    }
    result = result * flag;
    return result;
}

int main(){
    char* a = " -1234def";
    char* b = "+1234";
    int i = myAtoi(a);
    int j = myAtoi(b);
    printf("%d \n",i);
    printf("%d",j);
    return 0;
}

```

2. `int i=(j=4,k=8,l=16,m=32); printf("%d", i);` 输出是多少?
3. 解释局部变量、全局变量和静态变量的含义。
4. 解释堆和栈的区别。
5. 论述含参数的宏与函数的优缺点。

#### 普天 C++ 笔试题

1. 实现双向链表删除一个节点 **P**，在节点 **P** 后插入一个节点，写出这两个函数。
2. 写一个函数，将其中的 `\t` 都转换成 4 个空格。
3. **Windows** 程序的入口是哪里？写出 **Windows** 消息机制的流程。
4. 如何定义和实现一个类的成员函数为回调函数？
5. **C++** 里面是不是所有的动作都是 `main()` 引起的？如果不是，请举例。
6. **C++** 里面如何声明 `const void f(void)` 函数为 **C** 程序中的库函数？
7. 下列哪两个是等同的

**A** `int b;`  
**B** `const int* a = &b;`  
**C** `const int* const a = &b;`  
**D** `int const* const a = &b;`

8. 内联函数在编译时是否做参数类型检查？

```
void g(base & b){  
    b.play;  
}  
void main(){  
    son s;  
    g(s);  
    return;  
}
```

#### 华为笔试题

1. 请你分别画出 **OSI** 的七层网络结构图和 **TCP/IP** 的五层结构图。
2. 请你详细地解释一下 **IP** 协议的定义，在哪个层上面？主要有什么作用？**TCP** 与 **UDP**

呢？

3. 请问交换机和路由器各自的实现原理是什么？分别在哪个层次上面实现的？
4. 请问 C++ 的类和 C 里面的 struct 有什么区别？
5. 请讲一讲析构函数和虚函数的用法和作用。
6. 全局变量和局部变量有什么区别？是怎么实现的？操作系统和编译器是怎么知道的？
7. 8086 是多少位的系统？在数据总线上是怎么实现的？

### Sony 笔试题

1. 完成下列程序

```
*  
**.  
*.*.*..  
*...*...*...  
*...*...*...*...  
*...*...*...*...*...  
*...*...*...*...*...*...  
*...*...*...*...*...*...*...  
-----  
||  
||  
||  
-----
```

```
#include <stdio.h>
```

```
#define N 8
```

```
int main()
```

```
{
```

```
    int i;
```

```
    int j;
```

```
    int k;
```

```
-----  
||  
||  
||  
-----
```

```
    return 0;
```

```
}
```

2. 完成程序，实现对数组的降序排序

```
#include <stdio.h>
```

```
void sort( );
```

```

int main()
{
    int array[]={45, 56, 76, 234, 1, 34, 23, 2, 3}; //数字任//意给出
    sort( );
    return 0;
}

void sort( )
{
    -----
    ||
    ||
    -----
}

```

3. 费波那其数列，1，1，2，3，5.....编写程序求第十项。可以用递归，也可以用其他方法，但要说明你选择的理由。

```

#include <stdio.h>
int Pheponatch(int);
int main()
{
    printf("The 10th is %d",Pheponatch(10));
    return 0;
}
int Pheponatch(int N)
{
    -----
    ||
    ||
    -----
}

```

4. 下列程序运行时会崩溃，请找出错误并改正，并且说明原因。

```

#include <stdio.h>
#include <malloc.h>
typedef struct{
    TNode* left;

```

```

TNode* right;
int value;
} TNode;
TNode* root=NULL;
void append(int N);
int main()
{
    append(63);
    append(45);
    append(32);
    append(77);
    append(96);
    append(21);
    append(17); // Again, 数字任意给出
}
void append(int N)
{
    TNode* NewNode=(TNode *)malloc(sizeof(TNode));
    NewNode->value=N;

    if(root==NULL)
    {
        root=NewNode;
        return;
    }
    else
    {
        TNode* temp;
        temp=root;
        while((N>=temp.value && temp.left!=NULL) || (N<temp. value && temp.
right!=NULL
))
        {
            while(N>=temp.value && temp.left!=NULL)

```

```

temp=temp.left;
while(N<temp.value && temp.right!=NULL)
temp=temp.right;
}
if(N>=temp.value)
temp.left=NewNode;
else
temp.right=NewNode;
return;
}
}

```

MSRA Interview Written Exam (December 2003, Time: 2.5 Hours)

1 写出下列算法的时间复杂度。

- (1)冒泡排序;
- (2)选择排序;
- (3)插入排序;
- (4)快速排序;
- (5)堆排序;
- (6)归并排序;

2 写出下列程序在 X86 上的运行结果。

```

struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
}test

```

```

void main(void)
{
    int i;
    test.a=2;
}

```

```
test.b=3;  
test.c=0;  
  
i=*((short *)&test);  
printf("%d\n",i);  
}
```

3 写出下列程序的运行结果。

```
unsigned int i=3;  
cout<<i * -1;
```

4 写出下列程序所有可能的运行结果。

```
int a;  
int b;  
int c;  
  
void F1()  
{  
    b=a*2;  
    a=b;  
}  
  
void F2()  
{  
    c=a+1;  
    a=c;  
}  
  
main()  
{  
    a=5;  
    //Start F1,F2 in parallel  
    F1(); F2();  
    printf("a=%d\n",a);  
}
```

5 考察了一个 CharPrev() 函数的作用。

6 对 16 Bits colors 的处理，要求：

- (1) Byte 转换为 RGB 时，保留高 5、6bits；
- (2) RGB 转换为 Byte 时，第 2、3 位置零。

7 一个链表的操作，注意代码的健壮和安全性。要求：

- (1) 增加一个元素；
- (2) 获得头元素；
- (3) 弹出头元素（获得值并删除）。

8 一个给定的数值由左边开始升位到右边第 N 位，如

0010<<1 == 0100

或者

0001 0011<<4 == 0011 0000

请用 C 或者 C++ 或者其他 X86 上能运行的程序实现。

附加题（只有在完成以上题目后，才获准回答）

In C++, what does "explicit" mean? what does "protected" mean?

1. 在 C++ 中有没有纯虚构造函数？
2. 在 C++ 的一个类中声明一个 static 成员变量有没有用？
3. 在 C++ 的一个类中声明一个静态成员函数有没有用？
4. 如何实现一个非阻塞的 socket？
5. setsockopt, ioctl 都可以对 socket 的属性进行设置，他们有什么不同？
6. 解释一下进程和线程的区别？
7. 解释一下多播（组播）和广播的含义？
8. 多播采用的协议是什么？
9. 在 C++ 中纯虚析构函数的作用是什么？请举例说明。
10. 编程，请实现一个 C 语言中类似 atoi 的函数功能（输入可能包含非数字和空格）

1. 分析下面的程序：

```
void GetMemory(char **p,int num)
{
    *p=(char *)malloc(num);
```

```
}

int main()
{
    char *str=NULL;

    GetMemory(&str,100);

    strcpy(str,"hello");

    free(str);

    if(str!=NULL)
    {
        strcpy(str,"world");
    }

    printf("\n str is %s",str);
    getchar();
}
```

问输出结果是什么？

答案：输出 str is world。

free 只是释放的 str 指向的内存空间,它本身的值还是存在的.

所以 free 之后，有一个好的习惯就是将 str=NULL.

此时 str 指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出 world 来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。

当你 malloc 一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的，只不过。。。。。。。楼上都说过了，最好别这么干。

## 2. 运行的结果为什么等于 15

```
#include "stdio.h"
#include "string.h"

void main()
{
    char aa[10];
    printf("%d",strlen(aa));
}
```

答案: `sizeof()`和初不初始化, 没有关系; `strlen()`和初始化有关。

### 3. 给定结构 struct A

```
{
    char t:4;
    char k:4;
    unsigned short i:8;
    unsigned long m;
};问 sizeof(A) = ?
```

答案: 给定结构

```
struct A
{
    char t:4; //4 位
    char k:4; //4 位
    unsigned short i:8; //8 位 这里要偏移 2 字节保证 4 字节对齐
    unsigned long m; //4 个字节
}; // 共 8 字节
```

### 4. 分析一下

```
#include<iostream.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
```

```

#include <memory.h>
typedef struct AA
{
    int b1:5;
    int b2:2;
}AA;
void main()
{
    AA aa;
    char cc[100];
    strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
    memcpy(&aa,cc,sizeof(AA));
    cout << aa.b1 << endl;
    cout << aa.b2 << endl;
}

```

答案: -16 和 1

首先 sizeof(AA) 的大小为 4, b1 和 b2 分别占 5bit 和 2bit.

经过 strcpy 和 memcpy 后, aa 的 4 个字节所存放的值是:

0,1,2,3 的 ASC 码, 即 00110000,00110001,00110010,00110011

所以, 最后一步: 显示的是这 4 个字节的前 5 位, 和之后的 2 位

分别为: 10000 和 01

因为 int 是有正负之分 所以是-16 和 1

5. 求函数返回值, 输入 x=9999;

```

int func ( x )
{
    int countx = 0;
    while ( x )
    {
        countx++;
        x = x&(x-1);
    }
    return countx;
}

```

}

结果呢？

答案：知道了这是统计 9999 的二进制数值中有多少个 1 的函数，且有

$$9999 = 9 \times 1024 + 512 + 256 + 15$$

$9 \times 1024$  中含有 1 的个数为 2；

512 中含有 1 的个数为 1；

256 中含有 1 的个数为 1；

15 中含有 1 的个数为 4；

故共有 1 的个数为 8，结果为 8。

$1000 - 1 = 0111$ ，正好是原数取反。这就是原理。

用这种方法来求 1 的个数是很效率很高的。

不必去一个一个地移位。循环次数最少。

6. int a,b,c 请写函数实现  $C=a+b$ ，不可以改变数据类型，如将 c 改为 long int，关键是如何处理溢出问题

答案：bool add (int a, int b,int \*c)

```
{  
    *c=a+b;  
    return (a>0 && b>0 &&(*c<a || *c<b) || (a<0 && b<0 &&(*c>a || *c>b)));  
}
```

7. 分析：

```
struct bit  
{  int a:3;  
    int b:2;  
    int c:3;  
};  
int main()  
{  
    bit s;  
    char *c=(char*)&s;  
    cout<<sizeof(bit)<<endl;
```

```
*c=0x99;  
cout << s.a << endl << s.b << endl << s.c << endl;  
int a=-1;  
printf("%x",a);  
return 0;  
}
```

输出为什么是?

答案: 4

1  
-1  
-4  
ffffffff

因为 0x99 在内存中表示为 100 11 001 , a = 001, b = 11, c = 100 (在 vc 环境中, 一般是由右到左进行分配的)

当 c 为有符号数时, c = 100, 最高 1 为表示 c 为负数, 负数在计算机用补码表示, 所以 c = -4; 同理  
b = -1;

当 c 为有符号数时, c = 100, 即 c = 4, 同理 b = 3

8. 改错:

```
#include <stdio.h>
```

```
int main(void) {  
  
    int **p;  
    int arr[100];  
  
    p = &arr;  
  
    return 0;  
}
```

答案: 搞错了, 是指针类型不同,

```
int **p; //二级指针  
&arr; //得到的是指向第一维为 100 的数组的指针
```

应该这样写#include <stdio.h>

```
int main(void) {
    int **p, *q;
    int arr[100];
    q = arr;
    p = &q;
    return 0;
}
```

9。下面这个程序执行后会有什么错误或者效果:

```
#define MAX 255
int main()
{
    unsigned char A[MAX],i; //i 被定义为 unsigned char
    for (i=0;i<=MAX;i++)
        A[i]=i;
}
```

答案: 死循环加数组越界访问 (C/C++不进行数组越界检查)

MAX=255

数组 A 的下标范围为:0..MAX-1,这是其一..

其二.当 i 循环到 255 时,循环内执行:

A[255]=255;

这句本身没有问题..但是返回 for (i=0;i<=MAX;i++)语句时,

由于 unsigned char 的取值范围在(0..255),i++以后 i 又为 0 了..无限循环下去.

11。 struct name1{  
 char str;  
 short x;  
 int num;  
}

```
struct name2{
    char str;
    int num;
```

```

short x;
}

sizeof(struct name1)=? ?, sizeof(struct name2)=? ?

```

答案: sizeof(struct name1)=8, sizeof(struct name2)=12

在第二个结构中, 为保证 num 按四个字节对齐, char 后必须留出 3 字节的空间; 同时为保证整个结构的自然对齐 (这里是 4 字节对齐), 在 x 后还要补齐 2 个字节, 这样就是 12 字节。

1.写出判断 ABCD 四个表达式的是否正确, 若正确, 写出经过表达式中 a 的值(3分)

int a = 4;  
 (A) a += (a++); (B) a += (++a); (C) (a++) += a; (D) (++a) += (a++);  
 a = ?

答: C 错误, 左侧不是一个有效变量, 不能赋值, 可改为 (++a) += a;  
 改后答案依次为 9,10,10,11

2.某 32 位系统下, C++程序, 请计算 sizeof 的值(5分).

```

char str[] = "www.ibegroup.com"
char *p = str;
int n = 10;

```

请计算

sizeof(str) = ? (1) //数组分配的内存是连续的  
 sizeof(p) = ? (2) //指针内存不连续, 指针的大小为 4  
 sizeof(n) = ? (3) /整形

```
void Foo ( char str[100] ) {
```

请计算

sizeof(str) = ? (4) //数组作为参数传递时, 传递的是数组的首地址, 即指向第一个元素的指针

}

```
void *p = malloc( 100 );
```

请计算

sizeof(p) = ? (5)

答: (1) 17 (2) 4 (3) 4 (4) 4 (5) 4

4. 回答下面的问题(6分)

(1).

```

Void GetMemory(char **p, int num){
  *p = (char *)malloc(num);
}
void Test(void){

```

```
char *str = NULL;
GetMemory(&str, 100);
strcpy(str, "hello");
printf(str);
}
```

请问运行 Test 函数会有什么样的结果？

答：输出“hello”

(2).

```
void Test(void){
char *str = (char *) malloc(100);
strcpy(str, "hello");
free(str);
if(str != NULL){
strcpy(str, "world");
printf(str);
}
}
```

请问运行 Test 函数会有什么样的结果？

答：输出“world”，因为 free(str)后并未改变 str 所指的内存内容。

(3).

```
char *GetMemory(void){
char p[] = "hello world";
return p;
}
void Test(void){
char *str = NULL;
str = GetMemory();
printf(str);
}
```

请问运行 Test 函数会有什么样的结果？

答：无效的指针，输出不确定

## 5. 编写 strcat 函数(6 分)

已知 strcat 函数的原型是 char \*strcat (char \*strDest, const char \*strSrc);  
其中 strDest 是目的字符串，strSrc 是源字符串。

(1) 不调用 C++/C 的字符串库函数，请编写函数 strcat

答：

VC 源码：

```
char * __cdecl strcat (char * dst, const char * src)
{
char * cp = dst;
```

```
while( *cp )
cp++; /* find end of dst */
while( *cp++ = *src++ ); /* Copy src to end of dst */
return( dst ); /* return dst */
}
```

(2) `strcat` 能把 `strSrc` 的内容连接到 `strDest`, 为什么还要 `char *` 类型的返回值?

答: 方便赋值给其他变量

6. MFC 中 `CString` 是类型安全类么?

答: 不是, 其它数据类型转换到 `CString` 可以使用 `CString` 的成员函数 `Format` 来转换

7. C++ 中为什么用模板类。

答: (1) 可用来创建动态增长和减小的数据结构

(2) 它是类型无关的, 因此具有很高的可复用性。

(3) 它在编译时而不是运行时检查数据类型, 保证了类型安全

(4) 它是平台无关的, 可移植性

(5) 可用于基本数据类型

8. `CSingleLock` 是干什么的。

答: 同步多个线程对一个数据类的同时访问

9. `NEWTEXTMETRIC` 是什么。

答: 物理字体结构, 用来设置字体的高宽大小

10. 程序什么时候应该使用线程, 什么时候单线程效率高。

答: 1. 耗时的操作使用线程, 提高应用程序响应

2. 并行操作时使用线程, 如 C/S 架构的服务器端并发线程响应用户的请求。

3. 多 CPU 系统中, 使用线程提高 CPU 利用率

4. 改善程序结构。一个既长又复杂的进程可以考虑分为多个线程, 成为几个独立或半独立的运行部分, 这样的程序会利于理解和修改。

其他情况都使用单线程。

11. Windows 是内核级线程么。

答: 见下一题

12. Linux 有内核级线程么。

答: 线程通常被定义为一个进程中代码的不同执行路线。从实现方式上划分, 线程有两种类型: “用户级线程”和“内核级线程”。用户线程指不需要内核支持而在用户程序中实现的线程, 其不依赖于操作系统核心, 应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。这种线程甚至在象 DOS 这样的操作系统中也可实现, 但线程的调度需要用户程序完成, 这有些类似 Windows 3.x 的协作式多任务。另外一种则需要内核的参与, 由内核完成线程的调度。其依赖于操作系统核心, 由内核的内部需求进行创建和撤销, 这两

种模型各有其好处和缺点。用户线程不需要额外的内核开支，并且用户态线程的实现方式可以被定制或修改以适应特殊应用的要求，但是当一个线程因 I/O 而处于等待状态时，整个进程就会被调度程序切换为等待状态，其他线程得不到运行的机会；而内核线程则没有各个限制，有利于发挥多处理器的并发优势，但却占用了更多的系统开支。

**Windows NT 和 OS/2 支持内核线程。Linux 支持内核级的多线程**

**13.C++中什么数据分配在栈或堆中，New 分配数据是在近堆还是远堆中？**

答：栈：存放局部变量，函数调用参数，函数返回值，函数返回地址。由系统管理

堆：程序运行时动态申请，new 和 malloc 申请的内存就在堆上

近堆还是远堆不是很清楚。

**14.使用线程是如何防止出现大的波峰。**

答：意思是防止同时产生大量的线程，方法是使用线程池，线程池具有可以同时提高调度效率和限制资源使用的好处，线程池中的线程达到最大数时，其他线程就会排队等候。

**15 函数模板与类模板有什么区别？**

答：函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

**16 一般数据库若出现日志满了，会出现什么情况，是否还能使用？**

答：只能执行查询等读操作，不能执行更改，备份等写操作，原因是任何写操作都要记录日志。也就是说基本上处于不能使用状态。

**17 SQL Server 是否支持行级锁，有什么好处？**

答：支持，设立封锁机制主要是为了对并发操作进行控制，对干扰进行封锁，保证数据的一致性和准确性，行级封锁确保在用户取得被更新的行到该行进行更新这段时间内不被其它用户所修改。因而行级锁即可保证数据的一致性又能提高数据操作的并发性。

**18 如果数据库满了会出现什么情况，是否还能使用？**

答：见 16

**19 关于内存对齐的问题以及 sizeof() 的输出**

答：编译器自动对齐的原因：为了提高程序的性能，数据结构（尤其是栈）应该尽可能地在自然边界上对齐。原因在于，为了访问未对齐的内存，处理器需要作两次内存访问；然而，对齐的内存访问仅需要一次访问。

**20 int i=10, j=10, k=3; k\*=i+j; k 最后的值是？**

答：60，此题考察优先级，实际写成：k=(i+j);，赋值运算符优先级最低

**21.对数据库的一张表进行操作,同时要对另一张表进行操作,如何实现？**

答：将操作多个表的操作放入到事务中进行处理

**22.TCP/IP 建立连接的过程?(3-way shake)**

答：在 TCP/IP 协议中，TCP 协议提供可靠的连接服务，采用三次握手建立一个连接。

第一次握手: 建立连接时, 客户端发送 syn 包( $syn=j$ )到服务器, 并进入 SYN\_SEND 状态, 等待服务器确认;

第二次握手: 服务器收到 syn 包, 必须确认客户的 SYN( $ack=j+1$ ), 同时自己也发送一个 SYN 包 ( $syn=k$ ), 即 SYN+ACK 包, 此时服务器进入 SYN\_RECV 状态;

第三次握手: 客户端收到服务器的 SYN+ACK 包, 向服务器发送确认包 ACK( $ack=k+1$ ), 此包发送完毕, 客户端和服务器进入 ESTABLISHED 状态, 完成三次握手。

23. ICMP 是什么协议, 处于哪一层?

答: Internet 控制报文协议, 处于网络层 (IP 层)

24. 触发器怎么工作的?

答: 触发器主要是通过事件进行触发而被执行的, 当对某一表进行诸如 UPDATE, INSERT, DELETE 这些操作时, 数据库就会自动执行触发器所定义的 SQL 语句, 从而确保对数据的处理必须符合由这些 SQL 语句所定义的规则。

25. winsock 建立连接的主要实现步骤?

答: 服务器端: socket() 建立套接字, 绑定 (bind) 并监听 (listen), 用 accept () 等待客户端连接。

客户端: socket() 建立套接字, 连接 (connect) 服务器, 连接上后使用 send() 和 recv (), 在套接字上写读数据, 直至数据交换完毕, closesocket() 关闭套接字。

服务器端: accept () 发现有客户端连接, 建立一个新的套接字, 自身重新开始等待连接。该新产生的套接字使用 send() 和 recv () 写读数据, 直至数据交换完毕, closesocket() 关闭套接字。

26. 动态连接库的两种方式?

答: 调用一个 DLL 中的函数有两种方法:

1. 载入时动态链接 (load-time dynamic linking), 模块非常明确调用某个导出函数, 使得他们就像本地函数一样。这需要链接时链接那些函数所在 DLL 的导入库, 导入库向系统提供了载入 DLL 时所需的信息及 DLL 函数定位。

2. 运行时动态链接 (run-time dynamic linking), 运行时可以通过 LoadLibrary 或 LoadLibraryEx 函数载入 DLL。DLL 载入后, 模块可以通过调用 GetProcAddress 获取 DLL 函数的出口地址, 然后就可以通过返回的函数指针调用 DLL 函数了。如此即可避免导入库文件了。

27. IP 组播有那些好处?答: Internet 上产生的许多新的应用, 特别是高带宽的多媒体应用, 带来了带宽的急剧消耗和网络拥挤问题。组播是一种允许一个或多个发送者 (组播源) 发送单一的数据包到多个接收者 (一次的, 同时的) 的网络技术。组播可以大大的节省网络带宽, 因为无论有多少个目标地址, 在整个网络的任何一条链路上只传送单一的数据包。所以说组播技术的核心就是针对如何节约网络资源的前提下保证服务质量。

## 一、 #include "filename.h" 和 #include <filename.h> 的区别

#include "filename.h" 是指编译器将从当前工作目录上开始查找此文件

#include <filename.h> 是指编译器将从标准库目录中开始查找此文件

## 二、头文件的作用

加强安全检测

通过头文件可能方便地调用库功能，而不必关心其实现方式

## 三、\*, &修饰符的位置

对于\*和&修饰符，为了避免误解，最好将修饰符紧靠变量名

## 四、if语句

不要将布尔变量与任何值进行比较，那会很容易出错的。

整形变量必须要有类型相同的值进行比较

浮点变量最好少比点，就算要比也要有值进行限制

指针变量要和 NULL 进行比较，不要和布尔型和整形比较

## 五、const 和#define 的比较

const 有数据类型，#define 没有数据类型

个别编译器中 const 可以进行调试，#define 不可以进行调试

在类中定义常量有两种方式

1、在类中声明常量，但不赋值，在构造函数初始化表中进行赋值；

2、用枚举代替 const 常量。

## 六、C++函数中值的传递方式

有三种方式：值传递(Pass by value)、指针传递(Pass by pointer)、引用传递(Pass by reference)

```
void fun(char c) //pass by value
void fun(char *str) //pass by pointer
void fun(char &str) //pass by reference
```

如果输入参数是以值传递的话，最好使用引用传递代替，因为引用传递省去了临时对象的构造和析构

函数的类型不能省略，就算没有也要加个 void

## 七、函数体中的指针或引用常量不能被返回

```
Char *func(void)
{
    char str[]="Hello Word";
    //这是不能被返回的，因为 str 是个指定变量，不是一般的值，函数结束后会被
    //注销掉
    return str;
}
```

函数体内的指针变量并不会随着函数的消亡而自动释放

## 八、一个内存拷贝函数的实现体

```
void *memcpy(void *pvTo,const void *pvFrom,size_t size)
{
    assert((pvTo!=NULL)&&(pvFrom!=NULL));
    byte *pbTo=(byte*)pvTo;      //防止地址被改变
    byte *pbFrom=(byte*)pvFrom;
    while (size-- >0)
        pbTo++ = pbFrom++;
    return pvTo;
}
```

## 九、内存的分配方式

分配方式有三种，请记住，说不定那天去面试的时候就会有人问你这问题

- 1、静态存储区，是在程序编译时就已经分配好的，在整个运行期间都存在，如全局变量、常量。
- 2、栈上分配，函数内的局部变量就是从这分配的，但分配的内存容易有限。
- 3、堆上分配，也称动态分配，如我们用 new,malloc 分配内存，用 delete,free 来释放的内存。

## 十、内存分配的注意事项

用 new 或 malloc 分配内存时，必须要对此指针赋初值。

用 delete 或 free 释放内存后，必须要将指针指向 NULL

不能修改指向常量的指针数据

## 十一、内容复制与比较

```
//数组 .....
char a[]="Hello Word!";
char b[10];
strcpy(b,a);
if (strcmp(a,b)==0)
{}
//指针 .....
char a[]="Hello Word!";
char *p;
p=new char[strlen(a)+1];
```

```
strcpy(p,a);
if (strcmp(p,a)==0)
{}
```

## 十二、**sizeof** 的问题

记住一点，C++无法知道指针所指对象的大小，指针的大小永远为4字节

```
char a[]="Hello World!"
char *p=a;
cout<<sizeof(a)<<endl; //12字节
cout<<sizeof(p)<<endl; //4字节
```

而且，在函数中，数组参数退化为指针，所以下面的内容永远输出为4

```
void fun(char a[1000])
{
cout<<sizeof(a)<<endl; //输出4而不是1000
}
```

## 十三、关于指针

- 1、指针创建时必须被初始化
- 2、指针在 free 或 delete 后必须置为 NULL
- 3、指针的长度都为4字节
- 4、释放内存时，如果是数组指针，必须要释放掉所有的内存，如

```
char *p=new char[100];
strcpy(p,"Hello World");
delete []p; //注意前面的〔〕号
p=NULL;
```

- 5、数组指针的内容不能超过数组指针的最大容量。

如：

```
char *p=new char[5];
strcpy(p,"Hello World"); //报错 目标容量不够大
delete []p; //注意前面的〔〕号
p=NULL;
```

## 十四、关于 **malloc/free** 和 **new /delete**

- **malloc/free** 是 C/C++的内存分配符，**new /delete** 是 C++的内存分配符。

- 注意： malloc/free 是库函数， new/delete 是运算符
- malloc/free 不能执行构造函数与析构函数，而 new/delete 可以
- new/delete 不能在 C 上运行，所以 malloc/free 不能被淘汰
- 两者都必须要成对使用
- C++ 中可以使用 `_set_new_hander` 函数来定义内存分配异常的处理

## 十五、C++的特性

C++新增加有重载(overload)，内联 (inline)，Const，Virtual 四种机制

重载和内联：即可用于全局函数，也可用于类的成员函数；

Const 和 Virtual：只可用于类的成员函数；

重载：在同一类中，函数名相同的函数。由不同的参数决定调用那个函数。函数可要不可要 Virtual 关键字。和全局函数同名的函数不叫重载。如果在类中调用同名的全局函数，必须用全局引用符号::引用。

覆盖是指派生类函数覆盖基类函数

函数名相同；

参数相同；

基类函数必须有 Virtual 关键字；

不同的范围(派生类和基类)。

隐藏是指派生类屏蔽了基类的同名函数相同

1、函数名相同，但参数不同，此时不论基类有无 Virtual 关键字，基类函数将被隐藏。

2、函数名相同，参数也相同，但基类无 Virtual 关键字(有就是覆盖)，基类函数将被隐藏。

内联：inline 关键字必须与定义体放在一起，而不是单单放在声明中。

Const：const 是 constant 的缩写，“恒定不变”的意思。被 const 修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

1、参数做输入用的指针型参数，加上 const 可防止被意外改动。

2、按值引用的用户类型做输入参数时，最好将按值传递的改为引用传递，并加上 const 关键字，目的是为了提高效率。数据类型为内部类型的就没必要做这件事情；如：

将 `void Func(A a)` 改为 `void Func(const A &a)`。

而 `void func(int a)` 就没必要改成 `void func(const int &a)`；

3、给返回值为指针类型的函数加上 `const`，会使函数返回值不能被修改，赋给的变量也只能是 `const` 型变量。如：函数 `const char*GetString(void);` `char*str=GetString()` 将会出错。而 `const char *str=GetString()` 将是正确的。

4、`Const` 成员函数是指此函数体内只能调用 `Const` 成员变量，提高 程序 的健壮性。如声明函数 `int GetCount(void) const;` 此函数体内就只能调用 `Const` 成员变量。

Virtual：虚函数：派生类可以覆盖掉的函数，纯虚函数：只是个空函数，没有函数实现体；

## 十六、`extern "C"`有什么作用？

`Extern "C"` 是由 C++ 提供的一个连接交换指定符号，用于告诉 C++ 这段代码是 C 函数。这是因为 C++ 编译后库中函数名会变得很长，与 C 生成的不一致，造成 C++ 不能直接调用 C 函数，加上 `extern "c"` 后，C++ 就能直接调用 C 函数了。

`Extern "C"` 主要使用正规 DLL 函数的引用和导出 和 在 C++ 包含 C 函数或 C 头文件 时使用。使用时在前面加上 `extern "c"` 关键字即可。

## 十七、构造函数与析构函数

派生类的构造函数应在初始化表里调用基类的构造函数；

派生类和基类的析构函数应加 `Virtual` 关键字。

不要小看构造函数和析构函数，其实编起来还是不容易。

```
#include <iostream.h>

class Base
{
public:
    virtual ~Base() { cout<< "~Base" << endl; }

class Derived : public Base
{
public:
    virtual ~Derived() { cout<< "~Derived" << endl; }

void main(void)
{
```

```
    Base * pB = new Derived; // upcast
    delete pB;
}
```

输出结果为：

```
~Derived
~Base
```

如果析构函数不为虚，那么输出结果为

```
~Base
```

1. 已知 `strcpy` 函数的原型是：

```
char *strcpy(char *strDest, const char *strSrc);
```

其中 `strDest` 是目的字符串，`strSrc` 是源字符串。不调用 C++/C 的字符串库函数，请编写函数 `strcpy`

答案：

```
char *strcpy(char *strDest, const char *strSrc)
{
    if ( strDest == NULL || strSrc == NULL)
        return NULL ;
    if ( strDest == strSrc)
        return strDest ;
    char *tempPtr = strDest ;
    while( (*strDest++ = *strSrc++) != '\0')
    ;
    return tempPtr ;
}
```

2. 已知类 `String` 的原型为：

```
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  
};
```

请编写 **String** 的上述 4 个函数。

答案:

```
String::String(const char *str)  
{  
if ( str == NULL ) //strlen 在参数为 NULL 时会抛异常才会有这步判断  
{  
m_data = new char[1] ;  
m_data[0] = " " ;  
}  
else  
{  
m_data = new char[strlen(str) + 1];  
strcpy(m_data,str);  
}  
}  
String::String(const String &other)  
{  
m_data = new char[strlen(other.m_data) + 1];  
strcpy(m_data,other.m_data);  
}  
String & String::operator =(const String &other)  
{  
if ( this == &other)  
return *this ;  
delete []m_data;  
m_data = new char[strlen(other.m_data) + 1];  
strcpy(m_data,other.m_data);  
return *this ;  
}  
String::~ String(void)  
{
```

```
delete []m_data ;  
}
```

### 3.简答

#### 3.1 头文件中的 `ifndef/define/endif` 干什么用?

答: 防止该头文件被重复引用。

#### 3.2 `#include` 和 `#i include "filename.h"` 有什么区别?

答: 对于 `#include` , 编译器从标准库路径开始搜索 `filename.h`

对于 `#i include "filename.h"` , 编译器从用户的工作路径开始搜索 `filename.h`

#### 4. `static` 有什么用途? (请至少说明两种)

1.限制变量的作用域

2.设置变量的存储域

#### 7. 引用与指针有什么区别?

1) 引用必须被初始化, 指针不必。

2) 引用初始化以后不能被改变, 指针可以改变所指的对象。

2) 不存在指向空值的引用, 但是存在指向空值的指针。

#### 8. 描述实时系统的基本特性

在特定时间内完成特定的任务, 实时性与可靠性

#### 9. 全局变量和局部变量在内存中是否有区别? 如果有, 是什么区别?

全局变量储存在静态数据库, 局部变量在堆栈

#### 10. 什么是平衡二叉树?

左右子树都是平衡二叉树 且左右子树的深度差值的绝对值不大于 1

#### 11. 堆栈溢出一般是由什么原因导致的?

没有回收垃圾资源

#### 12. 什么函数不能声明为虚函数?

`constructor`

#### 13. 冒泡排序算法的时间复杂度是什么?

$O(n^2)$

#### 14. 写出 `float x` 与“零值”比较的 `if` 语句。

```
if(x>0.000001&&x<-0.000001)
```

#### 16. Internet 采用哪种网络协议? 该协议的主要层次结构?

`tcp/ip` 应用层/传输层/网络层/数据链路层/物理层

17. Internet 物理地址和 IP 地址转换采用什么协议?

ARP (Address Resolution Protocol) (地址解析协议)

18. IP 地址的编码分为哪俩部分?

IP 地址由两部分组成, 网络号和主机号。不过是要和“子网掩码”按位与上之后才能区分哪些是网络位哪些是主机位。

2. 用户输入 M, N 值, 从 1 至 N 开始顺序循环数数, 每数到 M 输出该数值, 直至全部输出。

写出 C 程序。

循环链表, 用取余操作做

3. 不能做 switch() 的参数类型是:

switch 的参数不能为实型。

## 華為

1、局部变量能否和全局变量重名?

答: 能, 局部会屏蔽全局。要用全局变量, 需要使用"::"

局部变量可以与全局变量同名, 在函数内引用这个变量时, 会用到同名的局部变量, 而不会用到全局变量。对于有些编译器而言, 在同一个函数内可以定义多个同名的局部变量, 比如在两个循环体内都定义一个同名的局部变量, 而那个局部变量的作用域就在那个循环体内

2、如何引用一个已经定义过的全局变量?

答: **extern**

可以用引用头文件的方式, 也可以用 **extern** 关键字, 如果用引用头文件方式来引用某个在头文件中声明的全局变量, 假定你将那个变量写错了, 那么在编译期间会报错, 如果你用 **extern** 方式引用时, 假定你犯了同样的错误, 那么在编译期间不会报错, 而在连接期间报错

3、全局变量可不可以定义在可被多个 C 文件包含的头文件中? 为什么?

答: 可以, 在不同的 C 文件中以 **static** 形式来声明同名全局变量。

可以在不同的 C 文件中声明同名的全局变量, 前提是其中只能有一个 C 文件中对此变量赋初值, 此时连接不会出错

4、语句 for( ; 1 ; ) 有什么问题? 它是什么意思?

答: 和 **while(1)** 相同。

5、do.....while 和 while.....do 有什么区别?

答: 前一个循环一遍再判断, 后一个判断以后再循环

6、请写出下列代码的输出内容

```
#include<stdio.h>
```

```
main()
{
int a,b,c,d;
a=10;
b=a++;
c=++a;
d=10*a++;
printf("b, c, d: %d, %d, %d", b, c, d) ;
return 0;
}
```

答: 10, 12, 120

1、**static** 全局变量与普通的全局变量有什么区别？**static** 局部变量和普通局部变量有什么区别？**static** 函数与普通函数有什么区别？

全局变量(外部变量)的说明之前再冠以 **static** 就构成了静态的全局变量。全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。这两者在存储方式上并无不同。这两者的区别虽在于非静态全局变量的作用域是整个源程序，当一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域局限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其它源文件中引起错误。

从以上分析可以看出，把局部变量改变为静态变量后是改变了它的存储方式即改变了它的生存期。把全局变量改变为静态变量后是改变了它的作用域，限制了它的使用范围。

**static** 函数与普通函数作用域不同。仅在本文件。只在当前源文件中使用的函数应该说明为内部函数(**static**)，内部函数应该在当前源文件中说明和定义。对于可在当前源文件以外使用的函数，应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件

**static** 全局变量与普通的全局变量有什么区别：**static** 全局变量只初使化一次，防止在其他文件单元中被引用；

**static** 局部变量和普通局部变量有什么区别：**static** 局部变量只被初始化一次，下一次依据上一次结果值；

**static** 函数与普通函数有什么区别：**static** 函数在内存中只有一份，普通函数在每个被调用中维持一份拷贝

2、程序的局部变量存在于（堆栈）中，全局变量存在于（静态区）中，动态申请数据存

在于（堆）中。

3、设有以下说明和定义：

```
typedef union {long i; int k[5]; char c;} DATE;
struct data { int cat; DATE cow; double dog;} too;
DATE max;
```

则语句 `printf("%d",sizeof(struct date)+sizeof(max));` 的执行结果是： 52

答：`DATE` 是一个 `union`，变量公用空间。里面最大的变量类型是 `int[5]`，占用 20 个字节。所以它的大小是 20

`data` 是一个 `struct`，每个变量分开占用空间。依次为 `int4 + DATE20 + double8 = 32`。

所以结果是  $20 + 32 = 52$ 。

当然...在某些 16 位编辑器下，`int` 可能是 2 字节，那么结果是 `int2 + DATE10 + double8 = 20`

4、队列和栈有什么区别？

队列先进先出，栈后进先出

5、写出下列代码的输出内容

```
#include<stdio.h>
int inc(int a)
{
    return(++a);
}
int multi(int*a,int*b,int*c)
{
    return(*c=(*a)**b);
}
typedef int(FUNC1)(int in);
typedef int(FUNC2) (int*,int*,int*);

void show(FUNC2 fun,int arg1, int*arg2)
{
    INCp=&inc;
    int temp =p(arg1);
    fun(&temp,&arg1, arg2);
    printf("%d\n",*arg2);
}
```

```
main()
```

```
{  
int a;  
show(multi,10,&a);  
return 0;  
}
```

答: 110

7、请找出下面代码中的所以错误

说明: 以下代码是把一个字符串倒序, 如“abcd”倒序后变为“dcba”

```
1、#include"string.h"  
2、main()  
3、{  
4、  char*src="hello,world";  
5、  char* dest=NULL;  
6、  int len=strlen(src);  
7、  dest=(char*)malloc(len);  
8、  char* d=dest;  
9、  char* s=src[len];  
10、 while(len--!=0)  
11、   d++=s--;  
12、   printf("%s",dest);  
13、   return 0;  
14、 }
```

答:

方法 1:

```
int main(){  
char* src = "hello,world";  
int len = strlen(src);  
char* dest = (char*)malloc(len+1);//要为\0 分配一个空间  
char* d = dest;  
char* s = &src[len-1];//指向最后一个字符  
while( len-- != 0 )  
*d++=*s--;
```

```
*d = 0;//尾部要加\0
printf("%s\n",dest);
free(dest);// 使用完，应当释放空间，以免造成内存泄露
return 0;
}
```

方法 2：

```
#include <stdio.h>
#include <string.h>
main()
{
char str[]="hello,world";
int len=strlen(str);
char t;
for(int i=0; i<len/2; i++)
{
t=str[i];
str[i]=str[len-i-1]; str[len-i-1]=t;
}
printf("%s",str);
return 0;
}
```

1.-1,2,7,28,,126 请问 28 和 126 中间那个数是什么？为什么？

第一题的答案应该是  $4^3-1=63$

规律是  $n^3-1$ (当  $n$  为偶数 0, 2, 4)

$n^3+1$ (当  $n$  为奇数 1, 3, 5)

答案: 63

2.用两个栈实现一个队列的功能？要求给出算法和思路！

设 2 个栈为 A,B，一开始均为空。

入队：

将新元素 push 入栈 A；

出队：

(1)判断栈 B 是否为空；

- (2)如果不为空，则将栈 A 中所有元素依次 **pop** 出并 **push** 到栈 B;
- (3)将栈 B 的栈顶元素 **pop** 出;

这样实现的队列入队和出队的平摊复杂度都还是  $O(1)$ ，比上面的几种方法要好。3.在 c 语言库函数中将一个字符转换成整型的函数是 **atool()** 吗，这个函数的原型是什么？

函数名: **atol**

功 能: 把字符串转换成长整型数

用 法: **long atol(const char \*nptr);**

程序例:

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    long l;
    char *str = "98765432";
    l = atol(lstr);
    printf("string = %s integer = %ld\n", str, l);
    return(0);
}
```

2.对于一个频繁使用的短小函数,在 C 语言中应用什么实现,在 C++中应用什么实现?

c 用宏定义, c++用 **inline**

3.直接链接两个信令点的一组链路称作什么?

**PPP** 点到点连接

4.接入网用的是什么接口?

5.**voip** 都用了那些协议?

6.软件测试都有那些种类?

黑盒: 针对系统功能的测试 白盒: 测试函数功能, 各函数接口

7.确定模块的功能和模块的接口是在软件设计的那个阶段完成的?

概要设计阶段

8.**enum string**

{

```

x1,
x2,
x3=10,
x4,
x5,
}x;

问 x= 0x801005, 0x8010f4 ;

9.unsigned char *p1;
unsigned long *p2;
p1=(unsigned char *)0x801000;
p2=(unsigned long *)0x810000;
请问 p1+5= ;
p2+5= ;

```

### 三.选择题:

- 1.Ethternet 链接到 Internet 用到以下那个协议?  
A.HDLC;B.ARP;C.UDP;D.TCP;E.ID
- 2.属于网络层协议的是:  
A.TCP;B.IP;C.ICMP;D.X.25
- 3.Windows 消息调度机制是:  
A.指令队列;B.指令堆栈;C.消息队列;D.消息堆栈;
- 4.unsigned short hash(unsigned short key)

```

{
    return (key>>)%256
}

```

请问 hash(16),hash(256)的值分别是:

- A.1.16;B.8.32;C.4.16;D.1.32

### 四.找错题:

- 1.请问下面程序有什么错误?

```

int a[60][250][1000],i,j,k;
for(k=0;k<=1000;k++)
    for(j=0;j<250;j++)
        for(i=0;i<60;i++)
            a[i][j][k]=0;

```

把循环语句内外换一下

```

2.#define Max_CB 500
void LmiQueryCSmd(Struct MSgCB * pmsg)
{
    unsigned char ucCmdNum;
    .....

    for(ucCmdNum=0;ucCmdNum<Max_CB;ucCmdNum++)
    {
        ....;
    }
}

```

死循环

3.以下是求一个数的平方的程序,请找出错误:

```

#define SQUARE(a)((a)*(a))
int a=5;
int b;
b=SQUARE(a++);

```

4.typedef unsigned char BYTE

```

int example_fun(BYTE gt_len; BYTE *gt_code)
{
    BYTE *gt_buf;
    gt_buf=(BYTE *)MALLOC(Max_GT_Length);
    .....
    if(gt_len>Max_GT_Length)
    {
        return GT_Length_ERROR;
    }
    .....
}

```

五.问答题:

1.IP Phone 的原理是什么?

IPV6

2.TCP/IP 通信建立的过程怎样, 端口有什么作用?

三次握手, 确定是哪个应用程序使用该协议

3.1 号信令和 7 号信令有什么区别, 我国某前广泛使用的是那一种?

#### 4. 列举 5 种以上的电话新业务？

微软亚洲技术中心的面试题！！！

##### 1. 进程和线程的差别。

线程是指进程内的一个执行单元,也是进程内的可调度实体.

与进程的区别:

(1)调度: 线程作为调度和分配的基本单位, 进程作为拥有资源的基本单位

(2)并发性: 不仅进程之间可以并发执行, 同一个进程的多个线程之间也可并发执行

(3)拥有资源: 进程是拥有资源的一个独立单位, 线程不拥有系统资源, 但可以访问隶属于进程的资源.

(4)系统开销: 在创建或撤消进程时, 由于系统都要为之分配和回收资源, 导致系统的开销明显大于创建或撤消线程时的开销。

##### 2. 测试方法

人工测试: 个人复查、抽查和会审

机器测试: 黑盒测试和白盒测试

##### 2. Heap 与 stack 的差别。

Heap 是堆, stack 是栈。

Stack 的空间由操作系统自动分配/释放, Heap 上的空间手动分配/释放。

Stack 空间有限, Heap 是很大的自由存储区

C 中的 malloc 函数分配的内存空间即在堆上,C++中对应的是 new 操作符。

程序在编译期对变量和函数分配内存都在栈上进行,且程序运行过程中函数调用时参数的传递也在栈上进行

##### 3. Windows 下的内存是如何管理的?

##### 4. 介绍 .Net 和 .Net 的安全性。

##### 5. 客户端如何访问 .Net 组件实现 Web Service?

##### 6. C/C++ 编译器中虚表是如何完成的?

##### 7. 谈谈 COM 的线程模型。然后讨论进程内/外组件的差别。

##### 8. 谈谈 IA32 下的分页机制

小页(4K)两级分页模式, 大页(4M)一级

##### 9. 给两个变量, 如何找出一个带环单链表中是什么地方出现环的?

一个递增一, 一个递增二, 他们指向同一个接点时就是环出现的地方

##### 10. 在 IA32 中一共有多少种办法从用户态跳到内核态?

通过调用门, 从 ring3 到 ring0, 中断从 ring3 到 ring0, 进入 vm86 等等

11. 如果只想让程序有一个实例运行，不能运行两个。像 **winamp** 一样，只能开一个窗口，怎样实现？

用内存映射或全局原子（互斥变量）、查找窗口句柄...

**FindWindow**, 互斥，写标志到文件或注册表,共享内存。.

12. 如何截取键盘的响应，让所有的'a'变成'b'？

键盘钩子 **SetWindowsHookEx**

13. Apartment 在 COM 中有什么用？为什么要引入？

14. 存储过程是什么？有什么用？有什么优点？

我的理解就是一堆 **sql** 的集合，可以建立非常复杂的查询，编译运行，所以运行一次后，以后再运行速度比单独执行 **SQL** 快很多

15. Template 有什么特点？什么时候用？

16. 谈谈 Windows DNA 结构的特点和优点。

网络编程中设计并发服务器，使用多进程 与 多线程，请问有什么区别？

1，进程：子进程是父进程的复制品。子进程获得父进程数据空间、堆和栈的复制品。

2，线程：相对与进程而言，线程是一个更加接近与执行体的概念，它可以与同进程的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。

两者都可以提高程序的并发度，提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源管理和保护；而进程正相反。同时，线程适合于在 **SMP** 机器上运行，而进程则可以跨机器迁移。

思科

1. 用宏定义写出 **swap** (x, y)

```
#define swap(x, y)\n    x = x + y;\n    y = x - y;\n    x = x - y;
```

2. 数组 **a[N]**，存放了 1 至 **N-1** 个数，其中某个数重复一次。写一个函数，找出被重复的数字。时间复杂度必须为 **o (N)** 函数原型：

```
int do_dup(int a[],int N)
```

3 一语句实现 x 是否为 2 的若干次幂的判断

```
int i = 512;
```

```
cout << boolalpha << ((i & (i - 1)) ? false : true) << endl;
```

4. **unsigned int intvert(unsigned int x,int p,int n)** 实现对 x 的进行转换, p 为起始转化位, n 为需要转换的长度, 假设起始点在右边. 如 x=0b0001 0001, p=4, n=3 转换后 x=0b0110 0001

```
unsigned int invert(unsigned int x,int p,int n){  
    unsigned int _t = 0;  
    unsigned int _a = 1;  
    for(int i = 0; i < n; ++i){  
        _t |= _a;  
        _a = _a << 1;  
    }  
    _t = _t << p;  
    x ^= _t;  
    return x;  
}
```

慧通：

什么是预编译

何时需要预编译：

- 1、总是使用不经常改动的大型代码体。
- 2、程序由多个模块组成，所有模块都使用一组标准的包含文件和相同的编译选项。在这种情况下，可以将所有包含文件预编译为一个预编译头。

```
char * const p;  
char const * p  
const char *p
```

上述三个有什么区别？

```
char * const p; //常量指针， p 的值不可以修改  
char const * p; //指向常量的指针， 指向的常量值不可以改  
const char *p; //和 char const *p
```

```
char str1[] = "abc";  
char str2[] = "abc";  
  
const char str3[] = "abc";  
const char str4[] = "abc";  
  
const char *str5 = "abc";  
const char *str6 = "abc";
```

```

char *str7 = "abc";
char *str8 = "abc";

cout << ( str1 == str2 ) << endl;
cout << ( str3 == str4 ) << endl;
cout << ( str5 == str6 ) << endl;
cout << ( str7 == str8 ) << endl;

```

结果是: 0 0 1 1

解答: str1,str2,str3,str4 是数组变量, 它们有各自的内存空间;

而 str5,str6,str7,str8 是指针, 它们指向相同的常量区域。

12. 以下代码中的两个 `sizeof` 用法有问题吗? [C 易]

```

void Uppercase( char str[] ) // 将 str 中的小写字母转换成大写字母
{
    for( size_t i=0; i<sizeof(str)/sizeof(str[0]); ++i )
        if( 'a'<=str[i] && str[i]<='z' )
            str[i] -= ('a'-'A');

    char str[] = "aBcDe";
    cout << "str 字符长度为: " << sizeof(str)/sizeof(str[0]) << endl;
    Uppercase( str );
    cout << str << endl;
}

```

答: 函数内的 `sizeof` 有问题。根据语法, `sizeof` 如用于数组, 只能测出静态数组的大小, 无法检测动态分配的或外部数组大小。函数外的 `str` 是一个静态定义的数组, 因此其大小为 6, 函数内的 `str` 实际只是一个指向字符串的指针, 没有任何额外的与数组相关的信息, 因此 `sizeof` 作用于上只将其当指针看, 一个指针为 4 个字节, 因此返回 4。

一个 32 位的机器, 该机器的指针是多少位

指针是多少位只要看地址总线的位数就行了。80386 以后的机子都是 32 的数据总线。所以

指针的位数就是 4 个字节了。

```
main()
{
    int a[5]={1,2,3,4,5};
    int *ptr=(int *)(&a+1);
```

```
    printf("%d,%d",*(a+1),*(ptr-1));
```

}

输出: 2,5

$*(a+1)$  就是  $a[1]$ ,  $*(ptr-1)$  就是  $a[4]$ , 执行结果是 2, 5

$\&a+1$  不是首地址+1, 系统会认为加一个  $a$  数组的偏移, 是偏移了一个数组的大小 (本例是 5 个 int)

```
int *ptr=(int *)(&a+1);
```

则  $ptr$  实际是  $\&(a[5])$ , 也就是  $a+5$

原因如下:

$\&a$  是数组指针, 其类型为  $int (*)[5]$ ;

而指针加 1 要根据指针类型加上一定的值,

不同类型的指针+1 之后增加的大小不同

$a$  是长度为 5 的 int 数组指针, 所以要加  $5 * sizeof(int)$

所以  $ptr$  实际是  $a[5]$

但是  $ptr$  与  $(\&a+1)$  类型是不一样的(这点很重要)

所以  $ptr-1$  只会减去  $sizeof(int*)$

$a$ ,  $\&a$  的地址是一样的, 但意思不一样,  $a$  是数组首地址, 也就是  $a[0]$  的地址,  $\&a$  是对象 (数组) 首地址,  $a+1$  是数组下一元素的地址, 即  $a[1]$ ,  $\&a+1$  是下一个对象的地址, 即  $a[5]$ .

1. 请问以下代码有什么问题:

```
int main()
{
    char a;
    char *str=&a;
    strcpy(str,"hello");
    printf(str);
    return 0;
```

}

没有为 **str** 分配内存空间，将会发生异常

问题出在将一个字符串复制进一个字符变量指针所指地址。虽然可以正确输出结果，但因为越界进行内存读写而导致程序崩溃。

```
char* s="AAA";
```

```
printf("%s",s);
```

```
s[0]='B';
```

```
printf("%s",s);
```

有什么错？

"AAA"是字符串常量。**s** 是指针，指向这个字符串常量，所以声明 **s** 的时候就有问题。

```
cosnt char* s="AAA";
```

然后又因为是常量，所以对 **s[0]**的赋值操作是不合法的。

1、写一个“标准”宏，这个宏输入两个参数并返回较小的一个。

```
.#define Min(X, Y) ((X)>(Y)?(Y):(X))//结尾没有;
```

2、嵌入式系统中经常要用到无限循环，你怎么用 C 编写死循环。

```
while(1){}或者 for(;;)
```

3、关键字 **static** 的作用是什么？

定义静态变量

4、关键字 **const** 有什么含意？

表示常量不可以修改的变量。

5、关键字 **volatile** 有什么含意？并举出三个不同的例子？

提示编译器对象的值可能在编译器未监测到的情况下改变。

**int (\*s[10])(int)** 表示的是什么啊

**int (\*s[10])(int)** 函数指针数组，每个指针指向一个 **int func(int param)**的函数。

1. 有以下表达式：

```
int a=248; b=4;int const c=21;const int *d=&a;
```

```
int *const e=&b;int const *f const =&a;
```

请问下列表达式哪些会被编译器禁止？为什么？

```
*c=32;d=&b;*d=43;e=&a;f=0x321f;
```

\*c 这是个什么东东，禁止

\*d 说了是 const， 禁止

e = &a 说了是 const 禁止

const \*f const =&a; 禁止

2. 交换两个变量的值，不使用第三个变量。即 a=3,b=5,交换之后 a=5,b=3;

有两种解法，一种用算术算法，一种用^(异或)

```
a = a + b;
```

```
b = a - b;
```

```
a = a - b;
```

or

```
a = a^b;// 只能对 int,char..
```

```
b = a^b;
```

```
a = a^b;
```

or

```
a ^= b ^= a;
```

3. c 和 c++ 中的 struct 有什么不同？

c 和 c++ 中 struct 的主要区别是 c 中的 struct 不可以含有成员函数，而 c++ 中的 struct 可以。

c++ 中 struct 和 class 的主要区别在于默认的存取权限不同，struct 默认为 public，而 class 默认为 private

4. #include <stdio.h>

```
#include <stdlib.h>
void getmemory(char *p)
{
    p=(char *) malloc(100);
    strcpy(p,"hello world");
}
int main( )
{
    char *str=NULL;
    getmemory(str);
    printf("%s/n",str);
    free(str);
    return 0;
```

}

程序崩溃，`getmemory` 中的 `malloc` 不能返回动态内存，`free()` 对 `str` 操作很危险

5. `char szstr[10];`

`strcpy(szstr,"0123456789");`

产生什么结果？为什么？

长度不一样，会造成非法的 OS

6. 列举几种进程的同步机制，并比较其优缺点。

原子操作

信号量机制

自旋锁

管程，会合，分布式系统

7. 进程之间通信的途径

共享存储系统

消息传递系统

管道：以文件系统为基础

11. 进程死锁的原因

资源竞争及进程推进顺序非法

12. 死锁的 4 个必要条件

互斥、请求保持、不可剥夺、环路

13. 死锁的处理

鸵鸟策略、预防策略、避免策略、检测与解除死锁

15. 操作系统中进程调度策略有哪几种？

FCFS(先来先服务)，优先级，时间片轮转，多级反馈

8. 类的静态成员和非静态成员有何区别？

类的静态成员每个类只有一个，非静态成员每个对象一个

9. 纯虚函数如何定义？使用时应注意什么？

`virtual void f()=0;`

是接口，子类必须要实现

10. 数组和链表的区别

数组：数据顺序存储，固定大小

链表：数据可以随机存储，大小可动态改变

12. ISO 的七层模型是什么？tcp/udp 是属于哪一层？tcp/udp 有何优缺点？

应用层  
表示层  
会话层  
运输层  
网络层  
物理链路层  
物理层

**tcp /udp 属于运输层**

**TCP** 服务提供了数据流传输、可靠性、有效流控制、全双工操作和多路复用技术等。  
与 **TCP** 不同， **UDP** 并不提供对 **IP** 协议的可靠机制、流控制以及错误恢复功能等。由于 **UDP** 比较简单， **UDP** 头包含很少的字节，比 **TCP** 负载消耗少。  
**tcp**: 提供稳定的传输服务，有流量控制，缺点是包头大，冗余性不好  
**udp**: 不提供稳定的服务，包头小，开销小

1: `(void *)ptr` 和 `(*(void**))ptr` 的结果是否相同？其中 `ptr` 为同一个指针

`(void *)ptr` 和 `(*(void**))ptr` 值是相同的

2: `int main()`

```
{  
    int x=3;  
    printf("%d",x);  
    return 1;  
  
}
```

问函数既然不会被其它函数调用，为什么要返回 1？

`main` 中，`c` 标准认为 0 表示成功，非 0 表示错误。具体的值是某中具体出错信息

1，要对绝对地址 `0x100000` 赋值，我们可以用

`(unsigned int*)0x100000 = 1234;`

那么要是想让程序跳转到绝对地址是 `0x100000` 去执行，应该怎么做？

`*((void (*)())0x100000)();`

首先要将 `0x100000` 强制转换成函数指针,即:

`(void (*)())0x100000`

然后再调用它：

```
*((void (*)())0x100000)();
```

用 `typedef` 可以看得更直观些：

```
typedef void(*)() voidFuncPtr;
```

```
*((voidFuncPtr)0x100000)();
```

2, 已知一个数组 `table`, 用一个宏定义, 求出数据的元素个数

```
#define NTBL
```

```
#define NTBL (sizeof(table)/sizeof(table[0]))
```

面试题：线程与进程的区别和联系？线程是否具有相同的堆栈？`DLL` 是否有独立的堆栈？

进程是死的，只是一些资源的集合，真正的程序执行都是线程来完成的，程序启动的时候操作系统就帮你创建了一个主线程。

每个线程有自己的堆栈。

`DLL` 中有没有独立的堆栈，这个问题不好回答，或者说这个问题本身是否有问题。因为 `DLL` 中的代码是被某些线程所执行，只有线程拥有堆栈，如果 `DLL` 中的代码是 `EXE` 中的线程所调用，那么这个时候是不是说这个 `DLL` 没有自己独立的堆栈？如果 `DLL` 中的代码是由 `DLL` 自己创建的线程所执行，那么是不是说 `DLL` 有独立的堆栈？

以上讲的是堆栈，如果对于堆来说，每个 `DLL` 有自己的堆，所以如果是从 `DLL` 中动态分配的内存，最好是从 `DLL` 中删除，如果你从 `DLL` 中分配内存，然后在 `EXE` 中，或者另外一个 `DLL` 中删除，很有可能导致程序崩溃

```
unsigned short A = 10;
```

```
printf("~A = %u\n", ~A);
```

```
char c=128;
```

```
printf("c=%d\n",c);
```

输出多少？并分析过程

第一题，`~A = 0xffffffff5`, `int` 值为 `-11`，但输出的是 `uint`。所以输出 `4294967285`

第二题，`c=0x10`, 输出的是 `int`, 最高位为 `1`, 是负数，所以它的值就是 `0x00` 的补码就是 `128`，所以输出 `-128`。

这两道题都是在考察二进制向 int 或 uint 转换时的最高位处理。

分析下面的程序：

```
void GetMemory(char **p,int num)
{
    *p=(char *)malloc(num);

}

int main()
{
    char *str=NULL;

    GetMemory(&str,100);

    strcpy(str,"hello");

    free(str);

    if(str!=NULL)
    {
        strcpy(str,"world");
    }

    printf("\n str is %s",str);
    getchar();
}
```

问输出结果是什么？希望大家能说说原因，先谢谢了

输出 str is world。

free 只是释放的 str 指向的内存空间,它本身的值还是存在的.

所以 free 之后，有一个好的习惯就是将 str=NULL.

此时 str 指向空间的内存已被回收,如果输出语句之前还存在分配空间的操作的话,这段存储空间是可能被重新分配给其他变量的,

尽管这段程序确实是存在大大的问题（上面各位已经说得很清楚了），但是通常会打印出 world 来。

这是因为，进程中的内存管理一般不是由操作系统完成的，而是由库函数自己完成的。当你 `malloc` 一块内存的时候，管理库向操作系统申请一块空间（可能会比你申请的大一些），然后在这块空间中记录一些管理信息（一般是在你申请的内存前面一点），并将可用内存的地址返回。但是释放内存的时候，管理库通常都不会将内存还给操作系统，因此你是可以继续访问这块地址的，只不过。。。。。。。楼上都说过了，最好别这么干。

`char a[10],strlen(a)`为什么等于 15? 运行的结果

```
#include "stdio.h"  
#include "string.h"  
  
void main()  
{  
  
char aa[10];  
printf("%d",strlen(aa));  
}
```

`sizeof()`和初不初始化，没有关系；  
`strlen()`和初始化有关。

```
char (*str)[20];/*str 是一个数组指针，即指向数组的指针. */  
char *str[20];/*str 是一个指针数组，其元素为指针型数据. */  
  
long a=0x801010;  
a+5=?  
0x801010 用二进制表示为：“1000 0000 0001 0000 0001 0000”，十进制的值为 8392720,  
再加上 5 就是 8392725 罗
```

1)给定结构 `struct A`

```
{  
char t:4;  
char k:4;
```

```

unsigned short i:8;
unsigned long m;
};问 sizeof(A) = ?
给定结构 struct A
{
    char t:4; 4 位
    char k:4; 4 位
    unsigned short i:8; 8 位
    unsigned long m; // 偏移 2 字节保证 4 字节对齐
}; // 共 8 字节

```

2)下面的函数实现在一个数上加一个数，有什么错误？请改正。

```

int add_n ( int n )
{
    static int i = 100;
    i += n;
    return i;
}

```

当你第二次调用时得不到正确的结果，难道你写个函数就是为了调用一次？问题就出在 **static** 上？

```

// 帮忙分析一下
#include<iostream.h>
#include <string.h>
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
typedef struct AA
{
    int b1:5;
    int b2:2;
}AA;
void main()

```

```

{
    AA aa;
    char cc[100];
    strcpy(cc,"0123456789abcdefghijklmnopqrstuvwxyz");
    memcpy(&aa,cc,sizeof(AA));
    cout << aa.b1 << endl;
    cout << aa.b2 << endl;
}

```

答案是 -16 和 1

首先 `sizeof(AA)` 的大小为 4, `b1` 和 `b2` 分别占 5bit 和 2bit.

经过 `strcpy` 和 `memcpy` 后, `aa` 的 4 个字节所存放的值是:

0,1,2,3 的 ASC 码, 即 00110000,00110001,00110010,00110011

所以, 最后一步: 显示的是这 4 个字节的前 5 位, 和之后的 2 位

分别为: 10000, 和 01

因为 `int` 是有正负之分 所以: 答案是-16 和 1

求函数返回值, 输入 `x=9999`;

```

int func ( x )
{
    int countx = 0;
    while ( x )
    {
        countx++;
        x = x&(x-1);
    }
    return countx;
}

```

结果呢?

知道了这是统计 9999 的二进制数值中有多少个 1 的函数, 且有

$$9999 = 9 \times 1024 + 512 + 256 + 15$$

$9 \times 1024$  中含有 1 的个数为 2;

512 中含有 1 的个数为 1;

256 中含有 1 的个数为 1;

15 中含有 1 的个数为 4;

故共有 1 的个数为 8, 结果为 8。

$1000 - 1 = 0111$ , 正好是原数取反。这就是原理。

用这种方法来求 1 的个数是很效率很高的。

不必去一个一个地移位。循环次数最少。

int a,b,c 请写函数实现  $C=a+b$ , 不可以改变数据类型, 如将 c 改为 long int, 关键是如何处理溢出问题

```
bool add (int a, int b,int *c)
{
    *c=a+b;
    return (a>0 && b>0 &&(*c<a || *c<b) || (a<0 && b<0 &&(*c>a || *c>b)));
}
```

分析:

```
struct bit
{
    int a:3;
    int b:2;
    int c:3;
};

int main()
{
    bit s;
    char *c=(char*)&s;
    cout<<sizeof(bit)<<endl;
    *c=0x99;
    cout << s.a <<endl <<s.b<<endl<<s.c<<endl;
    int a=-1;
    printf("%x",a);
    return 0;
}
```

输出为什么是

```
1
-1
-4
fffffff
```

因为 0x99 在内存中表示为 100 11 001 , a = 001, b = 11, c = 100

当 c 为有符号数时, c = 100, 最高 1 为表示 c 为负数, 负数在计算机用补码表示, 所以 c = -4;  
同理

b = -1;

当 c 为有符号数时, c = 100, 即 c = 4, 同理 b = 3

位域 :

有些信息在存储时, 并不需要占用一个完整的字节, 而只需占几个或一个二进制位。例如在存放一个开关量时, 只有 0 和 1 两种状态, 用一位二进位即可。为了节省存储空间, 并使处理简便, C 语言又提供了一种数据结构, 称为“位域”或“位段”。所谓“位域”是把一个字节中的二进位划分为几个不同的区域, 并说明每个区域的位数。每个域有一个域名, 允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

一、位域的定义和位域变量的说明位域定义与结构定义相仿, 其形式为:

**struct** 位域结构名

{ 位域列表 };

其中位域列表的形式为: 类型说明符 位域名: 位域长度

例如:

```
struct bs
{
    int a:8;
    int b:2;
    int c:6;
};
```

位域变量的说明与结构变量说明的方式相同。可采用先定义后说明, 同时定义说明或者直接说明这三种方式。例如:

```
struct bs
{
    int a:8;
    int b:2;
    int c:6;
}data;
```

说明 **data** 为 **bs** 变量，共占两个字节。其中位域 **a** 占 8 位，位域 **b** 占 2 位，位域 **c** 占 6 位。对于位域的定义尚有以下几点说明：

1. 一个位域必须存储在同一个字节中，不能跨两个字节。如一个字节所剩空间不够存放另一位域时，应从下一单元起存放该位域。也可以有意使某位域从下一单元开始。例如：

```
struct bs
{
    unsigned a:4
    unsigned :0 /*空域*/
    unsigned b:4 /*从下一单元开始存放*/
    unsigned c:4
}
```

在这个位域定义中，**a** 占第一字节的 4 位，后 4 位填 0 表示不使用，**b** 从第二字节开始，占用 4 位，**c** 占用 4 位。

2. 由于位域不允许跨两个字节，因此位域的长度不能大于一个字节的长度，也就是说不能超过 8 位二进位。
3. 位域可以无位域名，这时它只用来作填充或调整位置。无名的位域是不能使用的。例如：

```
struct k
{
    int a:1
    int :2 /*该 2 位不能使用*/
    int b:3
    int c:2
};
```

从以上分析可以看出，位域在本质上就是一种结构类型，不过其成员是按二进位分配的。

二、位域的使用 位域的使用和结构成员的使用相同，其一般形式为：位域变量名•位域名 位域允许用各种格式输出。

```
main(){
```

```
struct bs
{
    unsigned a:1;
    unsigned b:3;
    unsigned c:4;
} bit,*pbit;
bit.a=1;
bit.b=7;
bit.c=15;
pri
```

改错：

```
#include <stdio.h>
```

```
int main(void) {
```

```
    int **p;
    int arr[100];
```

```
    p = &arr;
```

```
    return 0;
}
```

解答：

搞错了,是指针类型不同,

```
int **p; //二级指针
&arr; //得到的是指向第一维为 100 的数组的指针
```

```
#include <stdio.h>
```

```
int main(void) {
```

```
    int **p, *q;
```

```
    int arr[100];
```

```
    q = arr;
```

```
    p = &q;
```

```
    return 0;
}
```

```
}
```

下面这个程序执行后会有什么错误或者效果:

```
#define MAX 255
int main()
{
    unsigned char A[MAX],i;/i 被定义为 unsigned char
    for (i=0;i<=MAX;i++)
        A[i]=i;
}
```

解答: 死循环加数组越界访问 (C/C++不进行数组越界检查)

MAX=255

数组 A 的下标范围为:0..MAX-1,这是其一..

其二.当 i 循环到 255 时,循环内执行:

```
A[255]=255;
```

这句本身没有问题..但是返回 for (i=0;i<=MAX;i++)语句时,

由于 unsigned char 的取值范围在(0..255),i++以后 i 又为 0 了..无限循环下去.

```
struct name1{
    char str;
    short x;
    int num;
}
```

```
struct name2{
    char str;
    int num;
    short x;
}
```

sizeof(struct name1)=8,sizeof(struct name2)=12

在第二个结构中, 为保证 num 按四个字节对齐, char 后必须留出 3 字节的空间; 同时为保证整个结构的自然对齐 (这里是 4 字节对齐), 在 x 后还要补齐 2 个字节, 这样就是 12 字节。

intel:

A.c 和 B.c 两个 c 文件中使用了两个相同名字的 static 变量,编译的时候会不会有问题?这两个 static 变量会保存到哪里 (栈还是堆或者其他) ?

static 的全局变量, 表明这个变量仅在本模块中有意义, 不会影响其他模块。

他们都放在数据区, 但是编译器对他们的命名是不同的。

如果要使变量在其他模块也有意义的话, 需要使用 **extern** 关键字。

```
struct s1
{
    int i: 8;
    int j: 4;
    int a: 3;
    double b;
};
```

```
struct s2
{
    int i: 8;
    int j: 4;
    double b;
    int a:3;
};
```

```
printf("sizeof(s1)= %d\n", sizeof(s1));
```

```
printf("sizeof(s2)= %d\n", sizeof(s2));
```

result: 16, 24

第一个 struct s1

```
{  
    int i: 8;  
    int j: 4;  
    int a: 3;  
    double b;  
};
```

理论上是这样的, 首先是 i 在相对 0 的位置, 占 8 位一个字节, 然后, j 就在相对一个字节

的位置，由于一个位置的字节数是 4 位的倍数，因此不用对齐，就放在那里了，然后是 **a** 要在 3 位的倍数关系的位置上，因此要移一位，在 15 位的位置上放下，目前总共是 18 位，折算过来是 2 字节 2 位的样子，由于 **double** 是 8 字节的，因此要在相对 0 要是 8 个字节的位置上放下，因此从 18 位开始到 8 个字节之间的位置被忽略，直接放在 8 字节的位置了，因此，总共是 16 字节。

第二个最后会对照是不是结构体内最大数据的倍数，不是的话，会补成是最大数据的倍数

上面是基本问题，接下来是编程问题：

本人很弱，这几个题也搞不定，特来求救：

1) 读文件 **file1.txt** 的内容（例如）：

12

34

56

输出到 **file2.txt**:

56

34

12

（逆序）

2) 输出和为一个给定整数的所有组合

例如 **n=5**

**5=1+4; 5=2+3** (相加的数不能重复)

则输出

1, 4; 2, 3。

望高手赐教！！

第一题,注意可增长数组的应用.

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main(void)
```

```
{
```

```
    int MAX = 10;
```

```
    int *a = (int *)malloc(MAX * sizeof(int));
```

```

int *b;

FILE *fp1;
FILE *fp2;

fp1 = fopen("a.txt","r");
if(fp1 == NULL)
{printf("error1");
 exit(-1);
}

fp2 = fopen("b.txt","w");
if(fp2 == NULL)
{printf("error2");
 exit(-1);
}

int i = 0;
int j = 0;

while(fscanf(fp1,"%d",&a[i]) != EOF)
{
i++;
j++;
if(i >= MAX)
{
MAX = 2 * MAX;
b = (int*)realloc(a,MAX * sizeof(int));
if(b == NULL)
{
printf("error3");
exit(-1);
}
a = b;
}
}

```

```
    }  
    }  
  
    for(--j >= 0;)  
        fprintf(fp2, "%d\n", a[j]);  
  
    fclose(fp1);  
    fclose(fp2);  
  
    return 0;  

```

```
}
```

第二题.

```
#include <stdio.h>  
  
int main(void)  
{  
    unsigned long int i,j,k;  
  
    printf("please input the number\n");  
    scanf("%d",&i);  
    if( i % 2 == 0)  
        j = i / 2;  
    else  
        j = i / 2 + 1;  
  
    printf("The result is \n");  
    for(k = 0; k < j; k++)  
        printf("%d = %d + %d\n",i,k,i - k);  
    return 0;  
}
```

```

#include <stdio.h>
void main()
{
    unsigned long int a,i=1;
    scanf("%d",&a);
    if(a%2==0)
    {
        for(i=1;i<a/2;i++)
            printf("%d",a,a-i);
    }
    else
        for(i=1;i<=a/2;i++)
            printf(" %d, %d",i,a-i);
}

```

兄弟,这样的题目若是做不出来实在是有些不应该,给你一个递规反向输出字符串的例子,可谓反序的经典例程.

```

void inverse(char *p)
{
    if( *p == '\0' )
        return;
    inverse( p+1 );
    printf( "%c", *p );
}

int main(int argc, char *argv[])
{
    inverse("abc\0");
    return 0;
}

```

借鉴了楼上的“递规反向输出”

```

#include <stdio.h>
void test(FILE *fread, FILE *fwrite)
{
    char buf[1024] = {0};
    if (!fgets(buf, sizeof(buf), fread))
        return;
    test( fread, fwrite );
    fputs(buf, fwrite);
}
int main(int argc, char *argv[])
{
    FILE *fr = NULL;
    FILE *fw = NULL;
    fr = fopen("data", "rb");
    fw = fopen("dataout", "wb");
    test(fr, fw);
    fclose(fr);
    fclose(fw);
    return 0;
}

```

在对齐为 4 的情况下

```

struct BBB
{
    long num;
    char *name;
    short int data;
    char ha;
    short ba[5];
}*p;
p=0x1000000;
p+0x200=____;
(Ulong)p+0x200=____;
(char*)p+0x200=____;

```

希望各位达人给出答案和原因，谢谢拉

解答：假设在 32 位 CPU 上，

`sizeof(long) = 4 bytes`

`sizeof(char *) = 4 bytes`

`sizeof(short int) = sizeof(short) = 2 bytes`

`sizeof(char) = 1 bytes`

由于是 4 字节对齐，

`sizeof(struct BBB) = sizeof(*p)`

`= 4 + 4 + 2 + 1 + 1/*补齐*/ + 2*5 + 2/*补齐*/ = 24 bytes (经 Dev-C++验证)`

`p=0x1000000;`

`p+0x200=____;`

`= 0x1000000 + 0x200*24`

`(Ulong)p+0x200=____;`

`= 0x1000000 + 0x200`

`(char*)p+0x200=____;`

`= 0x1000000 + 0x200*4`

你可以参考一下指针运算的细节

写一段程序，找出数组中第  $k$  大小的数，输出数所在的位置。例如{2, 4, 3, 4, 7}中，第一大的数是 7，位置在 4。第二大、第三大的数都是 4，位置在 1、3 随便输出哪一个均可。

函数接口为： `int find_orderk(const int* narry,const int n,const int k)`

要求算法复杂度不能是  $O(n^2)$

谢谢！

可以先用快速排序进行排序，其中用另外一个进行地址查找

代码如下，在 VC++6.0 运行通过。给分吧^-^

`//快速排序`

```

#include<iostream>

using namespace std;

int Partition (int*L,int low,int high)
{
    int temp = L[low];
    int pt = L[low];

    while (low < high)
    {
        while (low < high && L[high] >= pt)
            --high;
        L[low] = L[high];
        while (low < high && L[low] <= pt)
            ++low;
        L[low] = temp;
    }
    L[low] = temp;

    return low;
}

void QSort (int*L,int low,int high)
{
    if (low < high)
    {
        int pl = Partition (L,low,high);

        QSort (L,low,pl - 1);
        QSort (L,pl + 1,high);
    }
}

```

```

intmain ()
{
intnarry[100],addr[100];
intsum = 1,t;

cout << "Input number:" << endl;
cin >> t;

while (t != -1)
{
narry[sum] = t;
addr[sum - 1] = t;
sum++;

cin >> t;
}

sum -= 1;
QSort (narry,1,sum);

for (int i = 1; i <= sum;i++)
cout << narry[i] << '\t';
cout << endl;

intk;
cout << "Please input place you want:" << endl;
cin >> k;

intaa = 1;
intkk = 0;
for (++)
{
if (aa == k)
break;
}

```

```

if (narry[kk] != narry[kk + 1])
{
    aa += 1;
    kk++;
}

}

cout << "The NO." << k << "number is:" << narry[sum - kk] << endl;
cout << "And it's place is:" ;
for (i = 0;i < sum;i++)
{
    if (addr[i] == narry[sum - kk])
        cout << i << '\t';
}

return0;
}

```

### 1、找错

```

Void test1()
{
    char string[10];
    char* str1="0123456789";
    strcpy(string, str1);// 溢出，应该包括一个存放'\0'的字符 string[11]
}

```

```

Void test2()
{
    char string[10], str1[10];
    for(l=0; l<10;l++)
    {

```

```

str1[i] ='a';
}
strcpy(string, str1);// l, i 没有声明。
}

Void test3(char* str1)
{
char string[10];
if(strlen(str1)<=10)// 改成<10,字符溢出, 将 strlen 改为 sizeof 也可以
{
strcpy(string, str1);
}
}

```

2.

```

void g(int**);
int main()
{
int line[10],i;
int *p=line; //p 是地址的地址
for (i=0;i<10;i++)
{
*p=i;
g(&p);//数组对应的值加 1
}
for(i=0;i<10;i++)
printf("%d\n",line[i]);
return 0;
}

```

```

void g(int**p)
{
(**p)++;
(*p)++;// 无效
}

```

}

输出:

1

2

3

4

5

6

7

8

9

10

3. 写出程序运行结果

```
int sum(int a)
```

```
{
```

```
auto int c=0;
```

```
static int b=3;
```

```
c+=1;
```

```
b+=2;
```

```
return(a+b+c);
```

```
}
```

```
void main()
```

```
{
```

```
int l;
```

```
int a=2;
```

```
for(l=0;l<5;l++)
```

```
{
```

```
printf("%d, ", sum(a));
```

```
}
```

```
}
```

// static 会保存上次结果, 记住这一点, 剩下的自己写

输出: 8,10,12,14,16,

4.

```
int func(int a)
```

```
{
```

```
int b;
```

```
switch(a)
```

```
{
```

```
case 1: 30;
```

```
case 2: 20;
```

```
case 3: 16;
```

```
default: 0
```

```
}
```

```
return b;
```

```
}
```

则  $func(1)=?$

// b 定义后就没有赋值。

5:

```
int a[3];
```

```
a[0]=0; a[1]=1; a[2]=2;
```

```
int *p, *q;
```

```
p=a;
```

```
q=&a[2];
```

则  $a[q-p]=a[2]$

解释：指针一次移动一个 int 但计数为 1

今天早上的面试题 9 道，比较难，向牛人请教，国内的一牛公司，坐落在北京北四环某大厦：

1、线形表 **a**、**b** 为两个有序升序的线形表，编写一程序，使两个有序线形表合并成一个有序升序线形表 **h**；

答案在 请化大学 严锐敏《数据结构第二版》第二章例题，数据结构当中，这个叫做：两路归并排序

```

Linklist *unio(Linklist *p,Linklist *q){
linklist *R,*pa,*qa,*ra;
pa=p;
qa=q;
R=ra=p;
while(pa->next!=NULL&&qa->next!=NULL){
if(pa->data>qa->data){
ra->next=qa;
qa=qa->next;
}
else{
ra->next=pa;
pa=pa->next;
}
}
if(pa->next!=NULL)
ra->next=pa;
if(qa->next!=NULL)
ra->next==qa;
return R;
}

```

2、运用四色定理，为 N 个区域举行配色，颜色为 1、2、3、4 四种，另有数组 **adj[N][N]**，如 **adj[i][j]=1** 则表示 i 区域与 j 区域相邻，数组 **color[N]**，如 **color[i]=1**，表示 i 区域的颜色为 1 号颜色。

#### 四色填充

3、用递归算法判断数组 **a[N]** 是否为一个递增数组。

递归的方法，记录当前最大的，并且判断当前的是否比这个还大，大则继续，否则返回 **false** 结束：

```

bool fun( int a[], int n )
{
if( n==1 )
return true;
if( n==2 )
return a[n-1] >= a[n-2];

```

```
return fun( a,n-1 ) && ( a[n-1] >= a[n-2] );
}
```

4、编写算法，从 10 亿个浮点数当中，选出其中最大的 10000 个。

用外部排序，在《数据结构》书上有

《计算方法导论》在找到第  $n$  大的数的算法上加工

5、编写一 unix 程序，防止僵尸进程的出现。

Top

free131(白日?做梦!)() 信誉: 100 2006-4-17 10:17:34 得分: 0

同学的 4 道面试题，应聘的职位是搜索引擎工程师，后两道超级难，（希望大家多给一些算发）

1.给两个数组和他们的大小，还有一动态开辟的内存，求交集，把交集放到动态内存 `dongtai`，并且返回交集个数

```
long jiaoji(long* a[],long b[],long* alength,long blength,long* dongtai[])
```

2.单连表的建立，把'a'--'z'26 个字母插入到连表中，并且倒叙，还要打印！

方法 1：

```
typedef struct val
```

```
{    int date_1;
```

```
    struct val *next;
```

```
}*p;
```

```
void main(void)
```

```
{    char c;
```

```
    for(c=122;c>=97;c--)
```

```
    { p.date=c;
```

```
        p=p->next;
```

```
}
```

```

p->next=NULL;
}
}

方法 2:

node *p = NULL;
node *q = NULL;

node *head = (node*)malloc(sizeof(node));
head->data = ' ';head->next=NULL;

node *first = (node*)malloc(sizeof(node));
first->data = 'a';first->next=NULL;head->next = first;
p = first;

int length = 'z' - 'b';
int i=0;
while ( i<=length )
{
    node *temp = (node*)malloc(sizeof(node));
    temp->data = 'b'+i;temp->next=NULL;q=temp;

    head->next = temp; temp->next=p;p=q;
    i++;
}

print(head);

```

### 3. 可怕的题目终于来了

象搜索的输入信息是一个字符串，统计 300 万输入信息中的最热门的前十条，我们每次输入的一个字符串为不超过 255byte, 内存使用只有 1G,

请描述思想，写出算发（c 语言），空间和时间复杂度，

4. 国内的一些贴吧，如 **baidu**, 有几十万个主题，假设每一个主题都有上亿的跟帖子，怎么样设计这个系统速度最好，请描述思想，写出算发（c 语言），空间和时间复杂度，

```

#include  string.h
main(void)
{   char  *src="hello,world";
    char  *dest=NULL;
    dest=(char  *)malloc(strlen(src));
    int  len=strlen(str);
    char  *d=dest;
    char  *s=src[len];
    while(len--!=0)
        d++=s--;
    printf("%s",dest);
}
找出错误！！
#include  "string.h"
#include "stdio.h"
#include "malloc.h"
main(void)
{
    char  *src="hello,world";
    char  *dest=NULL;
    dest=(char  *)malloc(sizeof(char)*(strlen(src)+1));
    int  len=strlen(src);
    char  *d=dest;
    char  *s=src+len-1;
    while(len--!=0)
        *d++=*s--;
    *d='\0';
    printf("%s",dest);
}

```

1. 简述一个 Linux 驱动程序的主要流程与功能。

2. 请列举一个软件中时间换空间或者空间换时间的例子。

```
void swap(int a,int b)
```

```
{
```

```
int c; c=a;a=b;b=a;
```

```
}
```

--->空优

```
void swap(int a,int b)
```

```
{
```

```
a=a+b;b=a-b;a=a-b;
```

```
}
```

6. 请问一下程序将输出什么结果？

```
char *RetMemory(void)
```

```
{
```

```
char p[] = "hellow world";
```

```
return p;
```

```
}
```

```
void Test(void)
```

```
{
```

```
char *str = NULL;
```

```
str = RetMemory();
```

```
printf(str);
```

```
}
```

RetMemory 执行完毕， p 资源被回收，指向未知地址。返回地址， str 的内容应是不可预测的，打印的应该是 str 的地址

写一个函数,它的原形是 `int continuumax(char *outputstr,char *inputstr)`

功能:

在字符串中找出连续最长的数字串，并把这个串的长度返回，并把这个最长数字串付给其中一个函数参数 `outputstr` 所指内存。例如: "abcd12345ed125ss123456789"的首地址传给 `inputstr` 后，函数将返回

9, `outputstr` 所指的值为 123456789

```
int continuumax(char *outputstr, char *inputstr)
```

```
{
```

```

char *in = inputstr, *out = outputstr, *temp, *final;
int count = 0, maxlen = 0;

while( *in != '\0' )
{
    if( *in > 47 && *in < 58 )
    {
        for(temp = in; *in > 47 && *in < 58 ; in++)
            count++;
    }
    else
        in++;
}

if( maxlen < count )
{
    maxlen = count;
    count = 0;
    final = temp;
}
}

for(int i = 0; i < maxlen; i++)
{
    *out = *final;
    out++;
    final++;
}
*out = '\0';
return maxlen;
}

```

不用库函数,用 C 语言实现将一整型数字转化为字符串

方法 1:

```

int getlen(char *s){
    int n;

```

```

for(n = 0; *s != '\0'; s++)
    n++;
return n;
}

void reverse(char s[])
{
    int c,i,j;
    for(i = 0,j = getlen(s) - 1; i < j; i++,j--){
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}

void itoa(int n,char s[])
{
    int i,sign;
    if((sign = n) < 0)
        n = -n;
    i = 0;
    do{/*以反序生成数字*/
        s[i++] = n%10 + '0';/*get next number*/
    }while((n /= 10) > 0);/*delete the number*/

    if(sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}

方法 2:

#include <iostream>
using namespace std;

void itochar(int num);

```

```

void itochar(int num)
{
int i = 0;
int j ;
char stra[10];
char strb[10];
while ( num )
{
stra[i++]=num%10+48;
num=num/10;
}
stra[i] = '\0';
for( j=0; j < i; j++)
{
strb[j] = stra[i-j-1];
}
strb[j] = '\0';
cout<<strb<<endl;

}
int main()
{
int num;
cin>>num;
itochar(num);
return 0;
}

```

前几天面试，有一题想不明白，请教大家！

```

typedef struct
{
int a:2;
int b:2;

```

```
int c:1;  
}test;  
  
test t;  
t.a = 1;  
t.b = 3;  
t.c = 1;  
  
printf("%d",t.a);  
printf("%d",t.b);  
printf("%d",t.c);
```

谢谢!

t.a 为 01,输出就是 1

t.b 为 11, 输出就是-1

t.c 为 1, 输出也是-1

3 个都是有符号数 int 嘛。

这是位扩展问题

01

11

1

编译器进行符号扩展

求组合数: 求 n 个数 (1...n) 中 k 个数的组合....

如: combination(5,3)

要求输出: 543, 542, 541, 532, 531, 521, 432, 431, 421, 321,

```
#include<stdio.h>
```

```
int pop(int *);  
int push(int );  
void combination(int ,int );  
  
int stack[3]={0};
```

```

top=-1;

int main()
{
int n,m;
printf("Input two numbers:\n");
while( (2!=scanf("%d%c%d",&n,&m)) )
{
fflush(stdin);
printf("Input error! Again:\n");
}
combination(n,m);
printf("\n");
}

void combination(int m,int n)
{
int temp=m;
push(temp);
while(1)
{
if(1==temp)
{
if(pop(&temp)&&stack[0]==n) //当栈底元素弹出&&为可能取的最小值, 循环退出
break;
}
else if( push(--temp))
{
printf("%d%d%d ",stack[0],stack[1],stack[2]);//§&auml;`¤@?
pop(&temp);
}
}
}

int push(int i)
{

```

```

stack[++top]=i;
if(top<2)
return 0;
else
return 1;
}
int pop(int *i)
{
*i=stack[top--];
if(top>=0)
return 0;
else
return 1;
}

```

1、用指针的方法，将字符串“ABCD1234efgh”前后对调显示

```

#include <stdio.h>
#include <string.h>
#include <dos.h>
int main()
{
    char str[] = "ABCD1234efgh";
    int length = strlen(str);
    char * p1 = str;
    char * p2 = str + length - 1;
    while(p1 < p2)
    {
        char c = *p1;
        *p1 = *p2;
        *p2 = c;
        ++p1;
        --p2;
    }
    printf("str now is %s\n",str);
}

```

```
    system("pause");
    return 0;
}
```

2、有一分数序列: 1/2,1/4,1/6,1/8....., 用函数调用的方法, 求此数列前 20 项的和

```
#include <stdio.h>
double getValue()
{
    double result = 0;
    int i = 2;
    while(i < 42)
    {
        result += 1.0 / i;//一定要使用 1.0 做除数, 不能用 1, 否则结果将自动转化成整数,
        即 0.000000
        i += 2;
    }
    return result;
}
int main()
{
    printf("result is %f\n", getValue());
    system("pause");
    return 0;
}
```

Top

free131(白日?做梦!) ( ) 信誉: 100 2006-4-17 10:18:33 得分: 0

有一个数组 a[1000]存放 0--1000;要求每隔二个数删掉一个数, 到末尾时循环至开头继续进行, 求最后一个被删掉的数的原始下标位置。

以 7 个数为例：

{0,1,2,3,4,5,6,7} 0->1->2 (删除) -->3->4->5(删除)-->6->7->0 (删除) , 如此循环直到最后一个数被删除。

方法 1：数组

```
#include <iostream>
using namespace std;
#define null 1000

int main()
{
    int arr[1000];
    for (int i=0;i<1000;++i)
        arr[i]=i;
    int j=0;
    int count=0;
    while(count<999)
    {
        while(arr[j%1000]==null)
            j=(++j)%1000;
        j=(++j)%1000;
        while(arr[j%1000]==null)
            j=(++j)%1000;
        j=(++j)%1000;
        while(arr[j%1000]==null)
            j=(++j)%1000;
        arr[j]=null;
        ++count;
    }
    while(arr[j]==null)
        j=(++j)%1000;

    cout<<j<<endl;
    return 0;
}
```

方法 2：链表

```
#include<iostream>
using namespace std;
#define null 0
struct node
{
    int data;
    node* next;
};
int main()
{
    node* head=new node;
    head->data=0;
    head->next=null;
    node* p=head;
    for(int i=1;i<1000;i++)
    {
        node* tmp=new node;
        tmp->data=i;
        tmp->next=null;
        head->next=tmp;
        head=head->next;
    }
    head->next=p;
    while(p!=p->next)
    {
        p->next->next=p->next->next->next;
        p=p->next->next;
    }
    cout<<p->data;
    return 0;
}
```

方法 3：通用算法

```
#include <stdio.h>
#define MAXLINE 1000 //元素个数
```

```

/*
MAXLINE 元素个数
a[] 元素数组
R[] 指针场
suffix 下标
index 返回最后的下标序号
values 返回最后的下标对应的值
start 从第几个开始
K 间隔
*/
int find_n(int a[],int R[],int K,int& index,int& values,int s=0) {
    int suffix;
    int front_node,current_node;
    suffix=0;
    if(s==0) {
        current_node=0;
        front_node=MAXLINE-1;
    }
    else {
        current_node=s;
        front_node=s-1;
    }
    while(R[front_node]!=front_node) {
        printf("%d\n",a[current_node]);
        R[front_node]=R[current_node];
        if(K==1) {
            current_node=R[front_node];
            continue;
        }
        for(int i=0;i<K;i++){
            front_node=R[front_node];
        }
        current_node=R[front_node];
    }
}

```

```

index=front_node;
values=a[front_node];

return 0;
}

int main(void) {
int a[MAXLINE],R[MAXLINE],suffix,index,values,start,i,K;
suffix=index=values=start=0;
K=2;

for(i=0;i<MAXLINE;i++) {
a[i]=i;
R[i]=i+1;
}
R[i-1]=0;
find_n(a,R,K,index,values,2);
printf("the value is %d,%d\n",index,values);
return 0;
}

```

试题：

```

void test2()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<10; i++)
    {
        str1[i] = 'a';
    }
    strcpy( string, str1 );
}

```

解答：对试题 2，如果面试者指出字符数组 `str1` 不能在数组内结束可以给 3 分；如果面试者指出 `strcpy(string, str1)` 调用使得从 `str1` 内存起复制到 `string` 内存起所复制的字节数具有不确定性可以给 7 分，在此基础上指出库函数 `strcpy` 工作方式的给 10 分；

`str1` 不能在数组内结束:因为 `str1` 的存储为: `{a,a,a,a,a,a,a,a,a}`,没有'\0'(字符串结束符)  
所以不能结束

`strcpy( char *s1,char *s2)`他的工作原理是, 扫描 `s2` 指向的内存, 逐个字符付到 `s1` 所指向的内存, 直到碰到'\0',因为 `str1` 结尾没有'\0', 所以具有不确定性, 不知道他后面还会付什么东东。

正确应如下

```
void test2()
{
    char string[10], str1[10];
    int i;
    for(i=0; i<9; i++)
    {
        str1[i] = 'a'+i; //把 abcdefghi 赋值给字符数组
    }
    str1[i]='\0';//加上结束符
    strcpy( string, str1 );
}
```

第二个 code 题是实现 `strcmp`

```
int StrCmp(const char *str1, const char *str2)
```

做是做对了, 没有抄搞, 比较乱

```
int StrCmp(const char *str1, const char *str2)
```

```
{
```

```
    assert(str1 && str2);
```

```
    while (*str1 && *str2 && *str1 == *str2) {
```

```
        str1++, str2++;
    }
```

```
    if (*str1 && *str2)
```

```
        return (*str1-*str2);
```

```
    elseif (*str1 && *str2==0)
```

```
        return 1;
```

```
    elseif (*str1 == 0 && *str2)
```

```
        return -1;
```

```
    else
```

```

    return 0;
}

int StrCmp(const char *str1, const char *str2)
{
    //省略判断空指针(自己保证)
    while(*str1 && *str1++ == *str2++);
    return *str1-*str2;
}

```

第三个 code 题是实现子串定位

```
int FindSubStr(const char *MainStr, const char *SubStr)
```

做是做对了，没有抄搞，比较乱

```
int MyStrstr(const char* MainStr, const char* SubStr)
```

```
{

```

```
const char *p;
```

```
const char *q;
```

```
const char * u = MainStr;
```

```
//assert((MainStr!=NULL)&& ( SubStr!=NULL));//用断言对输入进行判断
```

```
while(*MainStr) //内部进行递增
```

```
{

```

```
    p = MainStr;
```

```
    q = SubStr;
```

```
    while(*q && *p && *p++ == *q++);
```

```
    if(!*q )
```

```
{

```

```
    return MainStr - u +1 ;//MainStr 指向当前起始位， u 指向
```

```
}
```

```
    MainStr ++;
```

```
}
```

```
    return -1;
}
```

分析：

```

int arr[] = {6,7,8,9,10};
int *ptr = arr;
*(ptr++)+=123;
printf(" %d %d ", *ptr, *(++ptr));
输出: 8 8

```

过程：对于`*(ptr++)+=123;`先做加法 6+123, 然后 ++, 指针指向 7; 对于`printf(" %d %d ", *ptr, *(++ptr));`从后往前执行, 指针先++，指向 8, 然后输出 8, 紧接着再输出 8

华为全套完整试题

高级题

6、已知一个单向链表的头, 请写出删除其某一个结点的算法, 要求, 先找到此结点, 然后删除。

```

slnodetype *Delete(slnodetype *Head,int key){}
{
    Head=Pointer->next;
    free(Pointer);
    break;
}
Back = Pointer;
    Pointer=Pointer->next;
if(Pointer->number==key)
{
    Back->next=Pointer->next;
    free(Pointer);
    break;
}
void delete(Node* p)
{
    if(Head = Node)
        while(p)
    }

```

有一个 16 位的整数, 每 4 位为一个数, 写函数求他们的和。

解释：

整数 1101010110110111

和 1101+0101+1011+0111

感觉应该不难，当时对题理解的不是很清楚，所以写了一个函数，也不知道对不对。

疑问：

既然是 16 位的整数，1101010110110111 是 2 进制的，那么函数参数怎么定义呢，请大虾指教。

答案：用十进制做参数，计算时按二进制考虑。

/\* n 就是 16 位的数，函数返回它的四个部分之和 \*/

```
char SumOfQuarters(unsigned short n)
```

```
{
```

```
    char c = 0;
```

```
    int i = 4;
```

```
    do
```

```
    {
```

```
        c += n & 15;
```

```
        n = n >> 4;
```

```
    } while (--i);
```

```
    return c;
```

```
}
```

有 1,2,...一直到 n 的无序数组,求排序算法,并且要求时间复杂度为  $O(n)$ ,空间复杂度  $O(1)$ ,使用交换,而且一次只能交换两个数. (华为)

```
#include<iostream.h>
```

```
int main()
```

```
{
```

```
    int a[] = {10,6,9,5,2,8,4,7,1,3};
```

```
    int len = sizeof(a) / sizeof(int);
```

```
    int temp;
```

```

    for(int i = 0; i < len; )
    {
        temp = a[a[i] - 1];
        a[a[i] - 1] = a[i];
        a[i] = temp;

        if ( a[i] == i + 1)
            i++;
    }

    for (int j = 0; j < len; j++)
        cout<<a[j]<<",";
}

return 0;
}

```

(慧通)

1 写出程序把一个链表中的接点顺序倒排

```

typedef struct linknode
{
    int data;
    struct linknode *next;
}node;
//将一个链表逆置
node *reverse(node *head)
{
    node *p,*q,*r;
    p=head;
    q=p->next;
    while(q!=NULL)
    {
        r=q->next;
        q->next=p;
        p=q;
        q=r;
    }
}

```

```
}
```

```
head->next=NULL;
```

```
head=p;
```

```
return head;
```

```
}
```

2 写出程序删除链表中的所有接点

```
void del_all(node *head)
```

```
{
```

```
node *p;
```

```
while(head!=NULL)
```

```
{
```

```
p=head->next;
```

```
free(head);
```

```
head=p;
```

```
}
```

```
cout<<"释放空间成功!"<<endl;
```

```
}
```

3 两个字符串，**s,t**;把**t**字符串插入到**s**字符串中，**s**字符串有足够的空间存放**t**字符串

```
void insert(char *s, char *t, int i)
```

```
{
```

```
char *q = t;
```

```
char *p = s;
```

```
if(q == NULL) return;
```

```
while(*p != '\0')
```

```
{
```

```
    p++;
```

```
}
```

```
while(*q != 0)
```

```
{
```

```
    *p = *q;
```

```
    p++;
```

```
    q++;
```

```
}
```

```
*p = '\0';
}
```

分析下面的代码：

```
char *a = "hello";
char *b = "hello";
if(a==b)
printf("YES");
else
printf("NO");
```

这个简单的面试题目,我选输出 **no**(对比的应该是指针地址吧),可在 VC 是 YES 在 C 是 NO 了呢, 是一个常量字符串。位于静态存储区, 它在程序生命期内恒定不变。如果编译器优化的话, 会有可能 **a** 和 **b** 同时指向同一个 **hello** 的。则地址相同。如果编译器没有优化, 那么就是两个不同的地址, 则不同

写一个函数, 功能: 完成内存之间的拷贝

memcpy source code:

```
270 void* memcpy( void *dst, const void *src, unsigned int len )
271 {
272     register char *d;
273     register char *s;
274
275     if (len == 0)
276         return dst;
277
278     if (is_overlap(dst, src, len, len))
279         complain3("memcpy", dst, src, len);
280
281     if ( dst > src ) {
282         d = (char *)dst + len - 1;
283         s = (char *)src + len - 1;
284         while ( len >= 4 ) {
285             *d-- = *s--;
286         }
287         if ( len > 4 ) {
288             d = (char *)dst + len - 5;
289             s = (char *)src + len - 5;
290             while ( len > 4 ) {
291                 *d-- = *s--;
292                 len -= 4;
293             }
294             if ( len > 0 ) {
295                 *d = '\0';
296             }
297         }
298     }
299     else {
300         d = (char *)dst;
301         s = (char *)src;
302         while ( len > 0 ) {
303             *d++ = *s++;
304             len--;
305         }
306     }
307     return dst;
308 }
```

```

286     *d-- = *s--;
287     *d-- = *s--;
288     *d-- = *s--;
289     len -= 4;
290 }
291     while ( len-- ) {
292     *d-- = *s--;
293 }
294 } else if ( dst < src ) {
295     d = (char *)dst;
296     s = (char *)src;
297     while ( len >= 4 ) {
298         *d++ = *s++;
299         *d++ = *s++;
300         *d++ = *s++;
301         *d++ = *s++;
302     len -= 4;
303 }
304     while ( len-- ) {
305         *d++ = *s++;
306 }
307 }
308     return dst;
309 }

```

公司考试这种题目主要考你编写的代码是否考虑到各种情况，是否安全（不会溢出）  
各种情况包括：

- 1、参数是指针，检查指针是否有效
- 2、检查复制的源目标和目的地是否为同一个，若为同一个，则直接跳出
- 3、读写权限检查
- 4、安全检查，是否会溢出

**memcpy** 拷贝一块内存，内存的大小你告诉它

**strcpy** 是字符串拷贝，遇到'\0'结束

*/\* memcpy—— 拷贝不重叠的内存块 \*/*

```

void memcpy(void* pvTo, void* pvFrom, size_t size)
{
    void* pbTo = (byte*)pvTo;
    void* pbFrom = (byte*)pvFrom;
    ASSERT(pvTo != NULL && pvFrom != NULL); //检查输入指针的有效性
    ASSERT(pbTo >= pbFrom+size || pbFrom >= pbTo+size); //检查两个指针指向的内存是否重
    叠
    while(size-->0)
        *pbTo++ == *pbFrom++;
    return(pvTo);
}

```

华为面试题：怎么判断链表中是否有环？

```

bool CircleInList(Link* pHead)
{
    if(pHead == NULL || pHead->next == NULL)//无节点或只有一个节点并且无自环
        return (false);
    if(pHead->next == pHead)//自环
        return (true);
    Link *pTemp1 = pHead;//step 1
    Link *pTemp = pHead->next;//step 2
    while(pTemp != pTemp1 && pTemp != NULL && pTemp->next != NULL)
    {
        pTemp1 = pTemp1->next;
        pTemp = pTemp->next->next;
    }
    if(pTemp == pTemp1)
        return (true);
    return (false);
}

```

两个字符串，**s,t**;把**t**字符串插入到**s**字符串中，**s**字符串有足够的空间存放**t**字符串

```
void insert(char *s, char *t, int i)
```

```

{
    memcpy(&s[strlen(t)+i],&s[i],strlen(s)-i);
    memcpy(&s[i],t,strlen(t));
    s[strlen(s)+strlen(t)]='\0';
}

```

1. 编写一个 C 函数，该函数在一个字符串中找到可能的最长的子字符串，且该字符串是由同一字符组成的。

```

char * search(char *cpSource, char ch)
{
    char *cpTemp=NULL, *cpDest=NULL;
    int iTemp, iCount=0;
    while(*cpSource)
    {
        if(*cpSource == ch)
        {
            iTemp = 0;
            cpTemp = cpSource;
            while(*cpSource == ch)
                ++iTemp, ++cpSource;
            if(iTemp > iCount)
                iCount = iTemp, cpDest = cpTemp;
            if(!*cpSource)
                break;
        }
        ++cpSource;
    }
    return cpDest;
}

```

2. 请编写一个 C 函数，该函数在给定的内存区域搜索给定的字符，并返回该字符所在位置索引值。

```

int search(char *cpSource, int n, char ch)
{
    int i;

```

```

    for(i=0; i<n && *(cpSource+i) != ch; ++i);
    return i;
}

```

一个单向链表，不知道头节点，一个指针指向其中的一个节点，问如何删除这个指针指向的节点？

将这个指针指向的 `next` 节点值 `copy` 到本节点，将 `next` 指向 `next->next`，并随后删除原 `next` 指向的节点。

```

#include <stdio.h>
void foo(int m, int n)
{
    printf("m=%d, n=%d\n", m, n);
}

int main()
{
    int b = 3;
    foo(b+=3, ++b);
    printf("b=%d\n", b);
    return 0;
}

```

输出： m=7,n=4,b=7(VC6.0)

这种方式和编译器中得函数调用关系相关即先后入栈顺序。不过不同编译器得处理不同。也是因为 C 标准中对这种方式说明为未定义，所以各个编译器厂商都有自己得理解，所以最后产生得结果完全不同。

因为这样，所以遇见这种函数，我们首先要考虑我们得编译器会如何处理这样得函数，其次看函数得调用方式，不同得调用方式，可能产生不同得结果。最后是看编译器优化。

2.写一函数，实现删除字符串 `str1` 中含有的字符串 `str2`。

第二个就是利用一个 KMP 匹配算法找到 `str2` 然后删除（用链表实现的话，便捷于数组）

```

/*雅虎笔试题(字符串操作)
给定字符串 A 和 B,输出 A 和 B 中的最大公共子串。
比如 A="aocdfa" B="pmcdfa" 则输出"cdf"
*/
//Author: azhen
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

char *commonstring(char shortstring[], char longstring[])
{
    int i, j;

    char *substring=malloc(256);

    if strstr(longstring, shortstring)!=NULL)           //如果....., 那么返回 shortstring
        return shortstring;

    for(i=strlen(shortstring)-1; i>0; i--)           //否则, 开始循环计算
    {
        for(j=0; j<=strlen(shortstring)-i; j++)
        {
            memcpy(substring, &shortstring[j], i);
            substring[i]='\0';
            if strstr(longstring, substring)!=NULL)
                return substring;
        }
    }
    return NULL;
}

main()
{

```

```

char *str1=malloc(256);
char *str2=malloc(256);
char *comman=NULL;

gets(str1);
gets(str2);

if(strlen(str1)>strlen(str2))           //将短的字符串放前面
    comman=commanstr(str2, str1);
else
    comman=commanstr(str1, str2);

printf("the longest comman string is: %s\n", comman);
}

```

11.写一个函数比较两个字符串 str1 和 str2 的大小, 若相等返回 0, 若 str1 大于 str2 返回 1, 若 str1 小于 str2 返回-1

```

int strcmp ( const char * src,const char * dst)
{
    int ret = 0 ;
    while( ! (ret = *(unsigned char *)src - *(unsigned char *)dst) && *dst)
    {
        ++src;
        ++dst;
    }
    if ( ret < 0 )
        ret = -1 ;
    else if ( ret > 0 )
        ret = 1 ;
    return( ret );
}

```

3,求 1000! 的末尾有几个 0 (用素数相乘的方法来做, 如  $72=2*2*2*3*3$ ) ;

求出 1->1000 里,能被 5 整除的数的个数 n1,能被 25 整除的数的个数 n2,能被 125 整除的数

的个数 n3,

能被 625 整除的数的个数 n4.

1000!末尾的零的个数=n1+n2+n3+n4;

```
#include<stdio.h>
```

```
#define NUM 1000
```

```
int find5(int num){  
    int ret=0;  
    while(num%5==0){  
        num/=5;  
        ret++;  
    }  
    return ret;  
}  
int main(){  
    int result=0;  
    int i;  
    for(i=5;i<=NUM;i+=5)  
    {  
        result+=find5(i);  
    }  
    printf(" the total zero number is %d\n",result);  
    return 0;  
}
```

1. 有双向循环链表结点定义为:

```
struct node  
{ int data;  
    struct node *front,*next;  
};
```

有两个双向循环链表 A, B, 知道其头指针为 :pHeadA,pHeadB, 请写一函数将两链表中 data 值相同的结点删除

```
BOOL DeleteNode(Node *pHeader, DataType Value)
```

```
{  
    if (pHeader == NULL) return;
```

```

BOOL bRet = FALSE;
Node *pNode = pHead;
while (pNode != NULL)
{
    if (pNode->data == Value)
    {
        if (pNode->front == NULL)
        {
            pHeader = pNode->next;
            pHeader->front = NULL;
        }
        else
        {
            if (pNode->next != NULL)
            {
                pNode->next->front = pNode->front;
            }
            pNode->front->next = pNode->next;
        }
    }

    Node *pNextNode = pNode->next;
    delete pNode;
    pNode = pNextNode;

    bRet = TRUE;
    //不要 break 或 return, 删除所有
}
else
{
    pNode = pNode->next;
}
}

```

```

return bRet;
}

void DE(Node *pHeadA, Node *pHeadB)
{
if (pHeadA == NULL || pHeadB == NULL)
{
return;
}

Node *pNode = pHeadA;
while (pNode != NULL)
{
if (DeleteNode(pHeadB, pNode->data))
{
if (pNode->front == NULL)
{
pHeadA = pNode->next;
pHeadA->front = NULL;
}
else
{
pNode->front->next = pNode->next;
if (pNode->next != NULL)
{
pNode->next->front = pNode->front;
}
}
}

Node *pNextNode = pNode->next;
delete pNode;
pNode = pNextNode;
}
else
{

```

```
pNode = pNode->next;
}
}
}
```

2. 编程实现：找出两个字符串中最大公共子字符串,如"abccade","dgcadde"的最大子串为"cad"

```
int GetCommon(char *s1, char *s2, char **r1, char **r2)
{
int len1 = strlen(s1);
int len2 = strlen(s2);
int maxlen = 0;

for(int i = 0; i < len1; i++)
{
    for(int j = 0; j < len2; j++)
    {
        if(s1[i] == s2[j])
        {
            int as = i, bs = j, count = 1;
            while(as + 1 < len1 && bs + 1 < len2 && s1[++as] == s2[++bs])
                count++;

            if(count > maxlen)
            {
                maxlen = count;
                *r1 = s1 + i;
                *r2 = s2 + j;
            }
        }
    }
}
```

3. 编程实现：把十进制数(**long** 型)分别以二进制和十六进制形式输出，不能使用 **printf** 系列库函数

```
char* test3(long num) {
```

```

char* buffer = (char*)malloc(11);
buffer[0] = '0';
buffer[1] = 'x';
buffer[10] = '\0';

char* temp = buffer + 2;
for (int i=0; i < 8; i++) {
temp[i] = (char)(num<<4*i>>28);
temp[i] = temp[i] >= 0 ? temp[i] : temp[i] + 16;
temp[i] = temp[i] < 10 ? temp[i] + 48 : temp[i] + 55;
}
return buffer;
}

```

输入 N, 打印 N\*N 矩阵

比如 N = 3, 打印:

```

1 2 3
8 9 4
7 6 5

```

N = 4, 打印:

```

1 2 3 4
12 13 14 5
11 16 15 6
10 9 8 7

```

解答:

```

1 #define N 15
int s[N][N];
void main()
{
int k = 0, i = 0, j = 0;
int a = 1;
for( ; k < (N+1)/2; k++ )

```

```

{
while( j < N-k ) s[i][j++] = a++; i++; j--;
while( i < N-k ) s[i++][j] = a++; i--; j--;
while( j > k-1 ) s[i][j--] = a++; i--; j++;
while( i > k )   s[i--][j] = a++; i++; j++;
}
for( i = 0; i < N; i++ )
{
for( j = 0; j < N; j++ )
cout << s[i][j] << '\t';
cout << endl;
}
}
2 define MAX_N 100
int matrix[MAX_N][MAX_N];

/*
* (x,y) : 第一个元素的坐标
* start: 第一个元素的值
* n: 矩阵的大小
*/
void SetMatrix(int x, int y, int start, int n) {
    int i, j;

    if (n <= 0) //递归结束条件
        return;
    if (n == 1) { //矩阵大小为 1 时
        matrix[x][y] = start;
        return;
    }
    for (i = x; i < x + n-1; i++) //矩阵上部
        matrix[y][i] = start++;

    for (j = y; j < y + n-1; j++) //右部
        matrix[j][x] = start++;
}

```

```

matrix[j][x+n-1] = start++;

for (i = x+n-1; i > x; i--)    //底部
    matrix[y+n-1][i] = start++;

for (j = y+n-1; j > y; j--)    //左部
    matrix[j][x] = start++;

SetMatrix(x+1, y+1, start, n-2); //递归
}

void main() {
    int i, j;
    int n;

    scanf("%d", &n);
    SetMatrix(0, 0, 1, n);

    //打印螺旋矩阵
    for(i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            printf("%4d", matrix[i][j]);
        printf("\n");
    }
}

```

斐波拉契数列递归实现的方法如下：

```

int Funct( int n )
{
    if(n==0) return 1;
    if(n==1) return 1;
    return Funct(n-1) + Funct(n-2);
}

```

请问，如何不使用递归，来实现上述函数？

请教各位高手！

解答：int Funct( int n ) // n 为非负整数

```
{  
    int a=0;  
    int b=1;  
    int c;  
    if(n==0) c=1;  
    else if(n==1) c=1;  
    else for(int i=2;i<=n;i++) //应该 n 从 2 开始算起  
    {  
        c=a+b;  
        a=b;  
        b=c;  
    }  
    return c;  
}
```

解答：

现在大多数系统都是将低字节放在前面，而结构体中位域的申明一般是先声明高位。

100 的二进制是 001 100 100

低位在前 高位在后

001---s3

100---s2

100---s1

所以结果应该是 1

如果先申明的在低位则：

001---s1

100---s2

100---s3

结果是 4

1、原题跟 little-endian, big-endian 没有关系

2、原题跟位域的存储空间分配有关，到底是从低字节分配还是从高字节分配，从 Dev C++ 和 VC7.1 上看，都是从低字节开始分配，并且连续分配，中间不空，不像谭的书那样会留空位

3、原题跟编译器有关，编译器在未用堆栈空间的默认值分配上有所不同，Dev C++未用空间分配为

01110111b，VC7.1 下为 11001100b，所以在 Dev C++ 下的结果为 5，在 VC7.1 下为 1。

注：PC 一般采用 **little-endian**，即高高低低，但在网络传输上，一般采用 **big-endian**，即高低低高，华为是做网络的，所以可能考虑 **big-endian** 模式，这样输出结果可能为 4

判断一个字符串是不是回文

```
int IsReverseStr(char *aStr)
{
    int i,j;
    int found=1;
    if(aStr==NULL)
        return -1;
    j=strlen(aStr);
    for(i=0;i<j/2;i++)
        if(*(aStr+i)!=(aStr+j-i-1))
    {
        found=0;
        break;
    }
    return found;
}
```

**Josephu** 问题为：设编号为 1, 2, ... n 的 n 个人围坐一圈，约定编号为 k ( $1 \leq k \leq n$ ) 的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

数组实现：

```
#include <stdio.h>
#include <malloc.h>
int Josephu(int n, int m)
{
```

```

int flag, i, j = 0;
int *arr = (int *)malloc(n * sizeof(int));
for (i = 0; i < n; ++i)
    arr[i] = 1;
for (i = 1; i < n; ++i)
{
    flag = 0;
    while (flag < m)
    {
        if (j == n)
            j = 0;
        if (arr[j])
            ++flag;
        ++j;
    }
    arr[j - 1] = 0;
    printf("第%d个出局的人是: %d号\n", i, j);
}
free(arr);
return j;
}

int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    printf("最后胜利的是%d号!\n", Josephu(n, m));
    system("pause");
    return 0;
}

```

链表实现:

```

#include <stdio.h>
#include <malloc.h>
typedef struct Node
{

```

```

int index;
struct Node *next;
}JosephuNode;
int Josephu(int n, int m)
{
    int i, j;
    JosephuNode *head, *tail;
    head = tail = (JosephuNode *)malloc(sizeof(JosephuNode));
    for (i = 1; i < n; ++i)
    {
        tail->index = i;
        tail->next = (JosephuNode *)malloc(sizeof(JosephuNode));
        tail = tail->next;
    }
    tail->index = i;
    tail->next = head;

    for (i = 1; tail != head; ++i)
    {
        for (j = 1; j < m; ++j)
        {
            tail = head;
            head = head->next;
        }
        tail->next = head->next;
        printf("第%d 个出局的人是: %d 号\n", i, head->index);
        free(head);
        head = tail->next;
    }
    i = head->index;
    free(head);
    return i;
}
int main()

```

```

{
    int n, m;
    scanf("%d%d", &n, &m);
    printf("最后胜利的是%d 号! \n", Josephu(n, m));
    system("pause");
    return 0;
}

```

已知 `strcpy` 函数的原型是：

```
char * strcpy(char * strDest,const char * strSrc);
```

1. 不调用库函数，实现 `strcpy` 函数。

2. 解释为什么要返回 `char *`。

解说：

1. `strcpy` 的实现代码

```
char * strcpy(char * strDest,const char * strSrc)
{
    if ((strDest==NULL)|| (strSrc==NULL)) file://[1]
        throw "Invalid argument(s)"; // [2]
    char * strDestCopy=strDest; file:// [3]
    while ((*strDest++=*strSrc++)!='\0'); file:// [4]
    return strDestCopy;
}
```

错误的做法：

[1]

(A) 不检查指针的有效性，说明答题者不注重代码的健壮性。

(B) 检查指针的有效性时使用 `((!strDest)||(!strSrc))` 或 `((!strDest&&!strSrc))`，说明答题者对

C 语言中类型的隐式转换没有深刻认识。在本例中 `char *` 转换为 `bool` 即是类型隐式转换，这种功能虽然灵活，但更多的是导致出错概率增大和维护成本升高。所以 C++ 专门增加了 `bool`、`true`、`false` 三个关键字以提供更安全的条件表达式。

(C) 检查指针的有效性时使用 `((strDest==0)|| (strSrc==0))`，说明答题者不知道使用常量的好处。直接使用字面常量（如本例中的 0）会减少程序的可维护性。0 虽然简单，但程序中可能出现很多处对指针的检查，万一出现笔误，编译器不能发现，生成的程序内含逻辑错误，很难排除。而使用 `NULL` 代替 0，如果出现拼写错误，编译器就会检查出来。

[2]

(A)`return new string("Invalid argument(s)");`，说明答题者根本不知道返回值的用途，并且他对内存泄漏也没有警惕心。从函数中返回函数体内分配的内存是十分危险的做法，他把释放内存的义务抛给不知情的调用者，绝大多数情况下，调用者不会释放内存，这导致内存泄漏。

(B)`return 0;`，说明答题者没有掌握异常机制。调用者有可能忘记检查返回值，调用者还可能无法检查返回值（见后面的链式表达式）。妄想让返回值肩负返回正确值和异常值的双重功能，其结果往往是两种功能都失效。应该以抛出异常来代替返回值，这样可以减轻调用者的负担、使错误不会被忽略、增强程序的可维护性。

[3]

(A)忘记保存原始的 `strDest` 值，说明答题者逻辑思维不严密。

[4]

(A)循环写成 `while (*strDest++=*strSrc++);`，同[1](B)。

(B)循环写成 `while (*strSrc]!='0') *strDest++=*strSrc++;`，说明答题者对边界条件的检查不力。循环体结束后，`strDest` 字符串的末尾没有正确地加上'0'。

一、请填写 `BOOL` , `float`, 指针变量 与“零值”比较的 `if` 语句。 (10 分)

请写出 `BOOL flag` 与“零值”比较的 `if` 语句。 (3 分)

标准答案：

`if ( flag )`

`if ( !flag )` 如下写法均属不良风格，不得分。

`if (flag == TRUE)`

`if (flag == 1 )`

`if (flag == FALSE)`

`if (flag == 0)`

请写出 `float x` 与“零值”比较的 `if` 语句。 (4 分)

标准答案示例：

```
const float EPSINON = 0.00001;  
  
if ((x >= - EPSINON) && (x <= EPSINON)
```

不可将浮点变量用“==”或“!=”与数字比较，应该设法转化成“>=”或“<=”此类形式。

如下是错误的写法，不得分。

```
if (x == 0.0)  
  
if (x != 0.0)
```

请写出 `char *p` 与“零值”比较的 `if` 语句。 (3 分)

标准答案：

```
if (p == NULL)  
  
if (p != NULL) 如下写法均属不良风格，不得分。  
  
if (p == 0)  
  
if (p != 0)  
  
if (p)  
  
if (!)
```

二、以下为 Windows NT 下的 32 位 C++ 程序，请计算 `sizeof` 的值 (10 分)

```
char str[] = "Hello" ;  
  
char *p = str ;  
  
int n = 10;
```

请计算

sizeof ( str ) = 6 (2 分)

sizeof ( p ) = 4 (2 分)

sizeof ( n ) = 4 (2 分) void Func ( char str[100] )

{

请计算

sizeof( str ) = 4 (2 分)

}

void \*p = malloc( 100 );

请计算

sizeof ( p ) = 4 (2 分)

三、简答题(25 分)

1、头文件中的 `ifndef/define/endif` 干什么用? (5 分)

答：防止该头文件被重复引用。

2、`#include?` 和 `#include "filename.h"` 有什么区别? (5 分)

答：对于`#include?`，编译器从标准库路径开始搜索 `filename.h`

对于`#include "filename.h"`，编译器从用户的工作路径开始搜索 `filename.h`

3、`const` 有什么用途? (请至少说明两种)(5 分)

答：(1)可以定义 `const` 常量

(2)`const` 可以修饰函数的参数、返回值，甚至函数的定义体。被 `const` 修饰的东西都

受到

强制保护，可以预防意外的变动，能提高程序的健壮性。

4、在 C++ 程序中调用被 C 编译器编译后的函数，为什么要加 `extern "C"`？ (5 分)

)

答：**C++**语言支持函数重载，**C**语言不支持函数重载。函数被**C++**编译后在库中的名字与

**C**语言

的不同。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。

名字。

**C++**提供了**C**连接交换指定符号 `extern"C"`来解决名字匹配问题。

5、请简述以下两个 `for` 循环的优缺点(5 分)

```
for (i=0; i<N; i++)
```

```
{
```

```
if (condition)
```

```
DoSomething();
```

```
else
```

```
DoOtherthing();
```

```
}
```

```
if (condition)
```

```
{
```

```
for (i=0; i<N; i++)
```

```
DoSomething();
```

```
}
```

```
else
```

```
{
```

```
for (i=0; i<N; i++)
```

```
DoOtherthing();
```

```
}
```

优点：程序简洁

缺点：多执行了  $N-1$  次逻辑判断，并且打断了循环“流水线”作业，使得编译器不能

对循环进行优化处理，降低了效率。 优点：循环的效率高

缺点：程序不简洁

四、有关内存的思考题(每小题 5 分，共 20 分)

```
void GetMemory(char *p)
```

```
{
```

```
p = (char *)malloc(100);
```

```
}
```

```
void Test(void)
```

```
{
```

```
char *str = NULL;
```

```
GetMemory(str);

strcpy(str, "hello world");

printf(str);

}
```

请问运行 **Test** 函数会有什么样的结果？

答：程序崩溃。

因为 **GetMemory** 并不能传递动态内存，

**Test** 函数中的 **str** 一直都是 **NULL**。

**strcpy(str, "hello world");** 将使程序崩溃。

```
char *GetMemory(void)
```

```
{
```

```
char p[] = "hello world";
```

```
return p;
```

```
}
```

```
void Test(void)
```

```
{
```

```
char *str = NULL;
```

```
str = GetMemory();
```

```
printf(str);
```

```
}
```

请问运行 **Test** 函数会有什么样的结果？

答：可能是乱码。

因为 **GetMemory** 返回的是指向“栈内存”的指针，该指针的地址不是 **NULL**，但其原现的

内容已经被清除，新内容不可知。

```
void GetMemory2(char **p, int num)

{
    *p = (char *)malloc(num);

}

void Test(void)

{
    char *str = NULL;

    GetMemory(&str, 100);

    strcpy(str, "hello");

    printf(str);

}
```

请问运行 **Test** 函数会有什么样的结果？

答：

(1)能够输出 **hello**

(2)内存泄漏

```
void Test(void)

{
    char *str = (char *) malloc(100);

    strcpy(str, "hello");

    free(str);

    if(str != NULL)

    {
        strcpy(str, "world");

        printf(str);

    }

}
```

请问运行 **Test** 函数会有什么样的结果？

答：篡改动态内存区的内容，后果难以预料，非常危险。

因为 `free(str);` 之后，`str` 成为野指针，

`if(str != NULL)` 语句不起作用。

## 五、编写 `strcpy` 函数(10 分)

已知 `strcpy` 函数的原型是

```
char *strcpy(char *strDest, const char *strSrc);
```

其中 `strDest` 是目的字符串，`strSrc` 是源字符串。

(1) 不调用 C++/C 的字符串库函数，请编写函数 `strcpy`

```
char *strcpy(char *strDest, const char *strSrc);

{
    assert((strDest!=NULL) && (strSrc !=NULL)); // 2 分

    char *address = strDest; // 2 分

    while( (*strDest++ = * strSrc++) != '\0' ) // 2 分

    NULL ;

    return address ; // 2 分

}
```

(2)strcpy 能把 strSrc 的内容复制到 strDest, 为什么还要 char \* 类型的返回值?

答: 为了实现链式表达式。 // 2 分

例如 int length = strlen( strcpy( strDest, "hello world" ) );

六、编写类 String 的构造函数、析构函数和赋值函数(25 分)

已知类 String 的原型为:

```
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  
  
};
```

请编写 **String** 的上述 4 个函数。

标准答案：

```
// String 的析构函数  
  
String::~String(void) // 3 分  
  
{  
  
    delete [] m_data;  
  
    // 由于 m_data 是内部数据类型，也可以写成 delete m_data;  
  
}  
  
// String 的普通构造函数  
  
String::String(const char *str) // 6 分  
  
{  
  
    if(str==NULL)  
  
    {  
  
        m_data = new char[1]; // 若能加 NULL 判断则更好  
  
        *m_data = '\0';  
  
    }  
  
    else
```

```

{
    int length = strlen(str);

    m_data = new char[length+1]; // 若能加 NULL 判断则更好

    strcpy(m_data, str);

}

}

// 拷贝构造函数

String::String(const String &other) // 3 分

{
    int length = strlen(other.m_data);

    m_data = new char[length+1]; // 若能加 NULL 判断则更好

    strcpy(m_data, other.m_data);

}

// 赋值函数

String & String::operator =(const String &other) // 13 分

{
    // (1) 检查自赋值 // 4 分

    if(this == &other)

        return *this;

    // (2) 释放原有的内存资源 // 3 分
}

```

```
delete [] m_data;  
  
// (3)分配新的内存资源，并复制内容 // 3 分  
  
int length = strlen(other.m_data);  
  
m_data = new char[length+1]; // 若能加 NULL 判断则更好  
  
strcpy(m_data, other.m_data);  
  
// (4)返回本对象的引用 // 3 分
```

## 1、内存分配方式

内存分配方式有三种：

- (1) 从静态存储区域分配。内存程序编译的时候就已经分配好，这块内存程序的整个运行期间都存在。例如全局变量，`static` 变量。
- (2) 在栈上创建。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。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- (3) 从堆上分配，亦称动态内存分配。程序在运行的时候用 `malloc` 或 `new` 申请任意多少的内存，程序员自己负责在何时用 `free` 或 `delete` 释放内存。动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

## 2、常见的内存错误及其对策

发生内存错误是件非常麻烦的事情。编译器不能自动发现这些错误，通常是在程序运行时才能捕捉到。而这些错误大多没有明显的症状，时隐时现，增加了改错的难度。有时用户怒气冲冲地把你找来，程序却没有发生任何问题，你一走，错误又发作了。常见的内存错误及其对策如下：

- \* 内存分配未成功，却使用了它。

编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。常用解决办法是，在使用内存之前检查指针是否为 **NULL**。如果指针 **p** 是函数的参数，那么在函数的入口处用 **assert(p!=NULL)** 进行

检查。如果是用 **malloc** 或 **new** 来申请内存，应该用 **if(p==NULL)** 或 **if(p!=NULL)** 进行防错处理。

\* 内存分配虽然成功，但是尚未初始化就引用它。

犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

\* 内存分配成功并且已经初始化，但操作越过了内存的边界。

例如在使用数组时经常发生下标“多 1”或者“少 1”的操作。特别是在 **for** 循环语句中，循环次数很容易搞错，导致数组操作越界。

\* 忘记了释放内存，造成内存泄露。

含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。

动态内存的申请与释放必须配对，程序中 **malloc** 与 **free** 的使用次数一定要相同，否则肯定有错误（**new/delete** 同理）。

\* 释放了内存却继续使用它。

有三种情况：

(1) 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。

(2) 函数的 `return` 语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存存在函数体结束时被自动销毁。

(3) 使用 `free` 或 `delete` 释放了内存后，没有将指针设置为 `NULL`。导致产生“野指针”。

**【规则 1】**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。

**【规则 2】**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。

**【规则 3】**避免数组或指针的下标越界，特别要当心发生“多 1”或者“少 1”操作。

**【规则 4】**动态内存的申请与释放必须配对，防止内存泄漏。

**【规则 5】**用 `free` 或 `delete` 释放了内存之后，立即将指针设置为 `NULL`，防止产生“野指针”。

### 3、指针与数组的对比

C++/C 程序中，指针和数组在不少地方可以相互替换着用，让人产生一种错觉，以为两者是等价的。

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。数组名对应着（而不是指向）一块内存，其地址与容量在生命期内保持不变，只有数组的内容可以改变。

指针可以随时指向任意类型的内存块，它的特征是“可变”，所以我们常用指针来操作动态内存。指针远比数组灵活，但也更危险。

下面以字符串为例比较指针与数组的特性。

#### 3.1 修改内容

示例 3-1 中，字符数组 `a` 的容量是 6 个字符，其内容为 `hello`。`a` 的内容可以改变，如 `a[0] = 'X'`。指针 `p` 指向常量字符串“`world`”（位于静态存储区，内容为 `world`），常量字符串的

内容是不可以被修改的。从语法上看，编译器并不觉得语句 `p[0] = 'X'` 有什么不妥，但是该语句企图修改常量字符串的内容而导致运行错误。

```
char a[] = "hello";
a[0] = 'X';
cout << a << endl;
char *p = "world"; // 注意 p 指向常量字符串
p[0] = 'X'; // 编译器不能发现该错误
cout << p << endl;
```

示例 3.1 修改数组和指针的内容

### 3.2 内容复制与比较

不能对数组名进行直接复制与比较。示例 7-3-2 中，若想把数组 `a` 的内容复制给数组 `b`，不能用语句 `b = a`，否则将产生编译错误。应该用标准库函数 `strcpy` 进行复制。同理，比较 `b` 和 `a` 的内容是否相同，不能用 `if(b==a)` 来判断，应该用标准库函数 `strcmp` 进行比较。

语句 `p = a` 并不能把 `a` 的内容复制指针 `p`，而是把 `a` 的地址赋给了 `p`。要想复制 `a` 的内容，可以先用库函数 `malloc` 为 `p` 申请一块容量为 `strlen(a)+1` 个字符的内存，再用 `strcpy` 进行字符串复制。同理，语句 `if(p==a)` 比较的不是内容而是地址，应该用库函数 `strcmp` 来比较。

```
// 数组...
char a[] = "hello";
char b[10];
strcpy(b, a); // 不能用 b = a;
if(strcmp(b, a) == 0) // 不能用 if (b == a)
...
// 指针...
int len = strlen(a);
char *p = (char *)malloc(sizeof(char)*(len+1));
strcpy(p,a); // 不要用 p = a;
if(strcmp(p, a) == 0) // 不要用 if (p == a)
...
```

## 示例 3.2 数组和指针的内容复制与比较

### 3.3 计算内存容量

用运算符 `sizeof` 可以计算出数组的容量（字节数）。示例 7-3-3（a）中，`sizeof(a)` 的值是 12（注意别忘了”）。指针 `p` 指向 `a`，但是 `sizeof(p)` 的值却是 4。这是因为 `sizeof(p)` 得到的是一个指针变量的字节数，相当于 `sizeof(char*)`，而不是 `p` 所指的内存容量。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。

注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。示例 7-3-3（b）中，不论数组 `a` 的容量是多少，`sizeof(a)` 始终等于 `sizeof(char*)`。

```
char a[] = "hello world";
char *p = a;
cout << sizeof(a) << endl; // 12 字节
cout << sizeof(p) << endl; // 4 字节
```

#### 示例 3.3（a） 计算数组和指针的内存容量

```
void Func(char a[100])
{
    cout << sizeof(a) << endl; // 4 字节而不是 100 字节
}
```

#### 示例 3.3（b） 数组退化为指针

### 4. 指针参数是如何传递内存的？

如果函数的参数是一个指针，不要指望用该指针去申请动态内存。示例 7-4-1 中，`Test` 函数的语句 `GetMemory(str, 200)` 并没有使 `str` 获得期望的内存，`str` 依旧是 `NULL`，为什么？

```
void GetMemory(char *p, int num)
{
    p = (char *)malloc(sizeof(char) * num);
}
```

```
void Test(void)
{
    char *str = NULL;
    GetMemory(str, 100); // str 仍然为 NULL
    strcpy(str, "hello"); // 运行错误
}
```

示例 4.1 试图用指针参数申请动态内存

毛病出在函数 `GetMemory` 中。编译器总是要为函数的每个参数制作临时副本，指针参数 `p` 的副本是 `_p`，编译器使 `_p = p`。如果函数体内的程序修改了 `_p` 的内容，就导致参数 `p` 的内容作相应的修改。这就是指针可以用作输出参数的原因。在本例中，`_p` 申请了新的内存，只是把 `_p` 所指的内存地址改变了，但是 `p` 丝毫未变。所以函数 `GetMemory` 并不能输出任何东西。事实上，每执行一次 `GetMemory` 就会泄露一块内存，因为没有用 `free` 释放内存。

如果非得要用指针参数去申请内存，那么应该改用“指向指针的指针”，见示例 4.2。

```
void GetMemory2(char **p, int num)
{
    *p = (char *)malloc(sizeof(char) * num);
}

void Test2(void)
{
    char *str = NULL;
    GetMemory2(&str, 100); // 注意参数是 &str，而不是 str
    strcpy(str, "hello");
    cout << str << endl;
    free(str);
}
```

示例 4.2 用指向指针的指针申请动态内存

由于“指向指针的指针”这个概念不容易理解，我们可以用函数返回值来传递动态内存。这种方法更加简单，见示例 4.3。

```
char *GetMemory3(int num)
{
    char *p = (char *)malloc(sizeof(char) * num);
    return p;
}

void Test3(void)
{
    char *str = NULL;
    str = GetMemory3(100);
    strcpy(str, "hello");
    cout<< str << endl;
    free(str);
}
```

#### 示例 4.3 用函数返回值来传递动态内存

用函数返回值来传递动态内存这种方法虽然好用，但是常常有人把 `return` 语句用错了。这里强调不要用 `return` 语句返回指向“栈内存”的指针，因为该内存函数结束时自动消亡，见示例 4.4。

```
char *GetString(void)
{
    char p[] = "hello world";
    return p; // 编译器将提出警告
}

void Test4(void)
{
    char *str = NULL;
    str = GetString(); // str 的内容是垃圾
    cout<< str << endl;
}
```

#### 示例 4.4 `return` 语句返回指向“栈内存”的指针

用调试器逐步跟踪 `Test4`，发现执行 `str = GetString` 语句后 `str` 不再是 `NULL` 指针，但

是 `str` 的内容不是“hello world”而是垃圾。

如果把示例 4.4 改写成示例 4.5，会怎么样？

```
char *GetString2(void)
{
    char *p = "hello world";
    return p;
}

void Test5(void)
{
    char *str = NULL;
    str = GetString2();
    cout << str << endl;
}
```

示例 4.5 `return` 语句返回常量字符串

函数 `Test5` 运行虽然不会出错，但是函数 `GetString2` 的设计概念却是错误的。因为 `GetString2` 内的“hello world”是常量字符串，位于静态存储区，它在程序生命期内恒定不变。无论什么时候调用 `GetString2`，它返回的始终是同一个“只读”的内存块。

## 5、杜绝“野指针”

“野指针”不是 `NULL` 指针，是指向“垃圾”内存的指针。人们一般不会错用 `NULL` 指针，因为用 `if` 语句很容易判断。但是“野指针”是很危险的，`if` 语句对它不起作用。“野指针”的成因主要有两种：

(1) 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为 `NULL` 指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为 `NULL`，要么让它指向合法的内存。例如

```
char *p = NULL;
char *str = (char *) malloc(100);
```

(2) 指针 **p** 被 **free** 或者 **delete** 之后, 没有置为 **NULL**, 让人误以为 **p** 是个合法的指针。

(3) 指针操作超越了变量的作用范围。这种情况让人防不胜防, 示例程序如下:

```
class A
{
public:
    void Func(void){ cout << "Func of class A" << endl; }
};

void Test(void)
{
    A *p;
    {
        A a;
        p = &a; // 注意 a 的生命期
    }
    p->Func(); // p 是“野指针”
}
```

函数 **Test** 在执行语句 **p->Func()** 时, 对象 **a** 已经消失, 而 **p** 是指向 **a** 的, 所以 **p** 就成了“野指针”。但奇怪的是我运行这个程序时居然没有出错, 这可能与编译器有关。

## 6、有了 **malloc/free** 为什么还要 **new/delete**?

**malloc** 与 **free** 是 C++/C 语言的标准库函数, **new/delete** 是 C++ 的运算符。它们都可用于申请动态内存和释放内存。

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

因此 C++ 语言需要一个能完成动态内存分配和初始化工作的运算符 **new**, 以及一个能

完成清理与释放内存工作的运算符 `delete`。注意 `new/delete` 不是库函数。我们先看一看 `malloc/free` 和 `new/delete` 如何实现对象的动态内存管理，见示例 6。

```
class Obj
{
public :
    Obj(void){ cout << "Initialization" << endl; }
    ~Obj(void){ cout << "Destroy" << endl; }
    void Initialize(void){ cout << "Initialization" << endl; }
    void Destroy(void){ cout << "Destroy" << endl; }
};

void UseMallocFree(void)
{
    Obj *a = (obj *)malloc(sizeof(obj)); // 申请动态内存
    a->Initialize(); // 初始化
    //...
    a->Destroy(); // 清除工作
    free(a); // 释放内存
}

void UseNewDelete(void)
{
    Obj *a = new Obj; // 申请动态内存并且初始化
    //...
    delete a; // 清除并且释放内存
}
```

示例 6 用 `malloc/free` 和 `new/delete` 如何实现对象的动态内存管理

类 `Obj` 的函数 `Initialize` 模拟了构造函数的功能，函数 `Destroy` 模拟了析构函数的功能。函数 `UseMallocFree` 中，由于 `malloc/free` 不能执行构造函数与析构函数，必须调用成员函数 `Initialize` 和 `Destroy` 来完成初始化与清除工作。函数 `UseNewDelete` 则简单得多。

所以我们不要企图用 `malloc/free` 来完成动态对象的内存管理，应该用 `new/delete`。由于内部数据类型的“对象”没有构造与析构的过程，对它们而言 `malloc/free` 和 `new/delete` 是等价的。

既然 `new/delete` 的功能完全覆盖了 `malloc/free`, 为什么 C++ 不把 `malloc/free` 淘汰出局呢? 这是因为 C++ 程序经常要调用 C 函数, 而 C 程序只能用 `malloc/free` 管理动态内存。

如果用 `free` 释放“`new` 创建的动态对象”, 那么该对象因无法执行析构函数而可能导致程序出错。如果用 `delete` 释放“`malloc` 申请的动态内存”, 理论上讲程序不会出错, 但是该程序的可读性很差。所以 `new/delete` 必须配对使用, `malloc/free` 也一样。

## 7. 内存耗尽怎么办?

如果在申请动态内存时找不到足够大的内存块, `malloc` 和 `new` 将返回 `NULL` 指针, 宣告内存申请失败。通常有三种方式处理“内存耗尽”问题。

(1) 判断指针是否为 `NULL`, 如果是则马上用 `return` 语句终止本函数。例如:

```
void Func(void)
{
    A *a = new A;
    if(a == NULL)
    {
        return;
    }
    ...
}
```

(2) 判断指针是否为 `NULL`, 如果是则马上用 `exit(1)` 终止整个程序的运行。例如:

```
void Func(void)
{
    A *a = new A;
    if(a == NULL)
    {
        cout << "Memory Exhausted" << endl;
        exit(1);
    }
}
```

```
 }  
 ...  
 }
```

(3) 为 `new` 和 `malloc` 设置异常处理函数。例如 Visual C++ 可以用 `_set_new_hander` 函数为 `new` 设置用户自己定义的异常处理函数，也可以让 `malloc` 享用与 `new` 相同的异常处理函数。详细内容请参考 C++ 使用手册。

上述 (1) (2) 方式使用最普遍。如果一个函数内有多处需要申请动态内存，那么方式 (1) 就显得力不从心（释放内存很麻烦），应该用方式 (2) 来处理。

很多人不忍心用 `exit(1)`，问：“不编写出错处理程序，让操作系统自己解决行不行？”

不行。如果发生“内存耗尽”这样的事情，一般说来应用程序已经无药可救。如果不 `exit(1)` 把坏程序杀死，它可能会害死操作系统。道理如同：如果不把歹徒击毙，歹徒在老死之前会犯下更多的罪。

有一个很重要的现象要告诉大家。对于 32 位以上的应用程序而言，无论怎样使用 `malloc` 与 `new`，几乎不可能导致“内存耗尽”。我在 Windows 98 下用 Visual C++ 编写了测试程序，见示例 7。这个程序会无休止地运行下去，根本不会终止。因为 32 位操作系统支持“虚存”，内存用完了，自动用硬盘空间顶替。我只听到硬盘嘎吱嘎吱地响，Window 98 已经累得对键盘、鼠标毫无反应。

我可以得出这么一个结论：对于 32 位以上的应用程序，“内存耗尽”错误处理程序毫无用处。这下可把 Unix 和 Windows 程序员们乐坏了：反正错误处理程序不起作用，我就不写了，省了很多麻烦。

我不想误导读者，必须强调：不加错误处理将导致程序的质量很差，千万不可因小失大。

```
void main(void)  
{  
    float *p = NULL;  
    while(TRUE)
```

```
{  
    p = new float[1000000];  
    cout << "eat memory" << endl;  
    if(p==NULL)  
        exit(1);  
}
```

示例 7 试图耗尽操作系统的内存

## 8、**malloc/free** 的使用要点

函数 **malloc** 的原型如下：

```
void * malloc(size_t size);
```

用 **malloc** 申请一块长度为 **length** 的整数类型的内存，程序如下：

```
int *p = (int *) malloc(sizeof(int) * length);
```

我们应当把注意力集中在两个要素上：“类型转换”和“**sizeof**”。

\* **malloc** 返回值的类型是 **void \***，所以在调用 **malloc** 时要显式地进行类型转换，将 **void \*** 转换成所需要的指针类型。

\* **malloc** 函数本身并不识别要申请的内存是什么类型，它只关心内存的总字节数。我们通常记不住 **int**, **float** 等数据类型的变量的确切字节数。例如 **int** 变量在 16 位系统下是 2 个字节，在 32 位下是 4 个字节；而 **float** 变量在 16 位系统下是 4 个字节，在 32 位下也是 4 个字节。最好用以下程序作一次测试：

```
cout << sizeof(char) << endl;  
cout << sizeof(int) << endl;
```

```
cout << sizeof(unsigned int) << endl;
cout << sizeof(long) << endl;
cout << sizeof(unsigned long) << endl;
cout << sizeof(float) << endl;
cout << sizeof(double) << endl;
cout << sizeof(void *) << endl;
```

在 `malloc` 的“`()`”中使用 `sizeof` 运算符是良好的风格，但要当心有时我们会昏了头，写出 `p = malloc(sizeof(p))` 这样的程序来。

\* 函数 `free` 的原型如下：

```
void free( void * memblock );
```

为什么 `free` 函数不象 `malloc` 函数那样复杂呢？这是因为指针 `p` 的类型以及它所指的内存的容量事先都是知道的，语句 `free(p)` 能正确地释放内存。如果 `p` 是 `NULL` 指针，那么 `free` 对 `p` 无论操作多少次都不会出问题。如果 `p` 不是 `NULL` 指针，那么 `free` 对 `p` 连续操作两次就会导致程序运行错误。

## 9、`new/delete` 的使用要点

运算符 `new` 使用起来要比函数 `malloc` 简单得多，例如：

```
int *p1 = (int *)malloc(sizeof(int) * length);
int *p2 = new int[length];
```

这是因为 `new` 内置了 `sizeof`、类型转换和类型安全检查功能。对于非内部数据类型的对象而言，`new` 在创建动态对象的同时完成了初始化工作。如果对象有多个构造函数，那么 `new` 的语句也可以有多种形式。例如

```
class Obj
{
```

```
public :  
    Obj(void); // 无参数的构造函数  
    Obj(int x); // 带一个参数的构造函数  
    ...  
}  
void Test(void)  
{  
    Obj *a = new Obj;  
    Obj *b = new Obj(1); // 初值为 1  
    ...  
    delete a;  
    delete b;  
}
```

如果用 **new** 创建对象数组，那么只能使用对象的无参数构造函数。例如

```
Obj *objects = new Obj[100]; // 创建 100 个动态对象
```

不能写成

```
Obj *objects = new Obj[100](1); // 创建 100 个动态对象的同时赋初值 1
```

在用 **delete** 释放对象数组时，留意不要丢了符号'[]'。例如

```
delete []objects; // 正确的用法  
delete objects; // 错误的用法
```

后者相当于 **delete objects[0]**，漏掉了另外 99 个对象。

## 10、一些心得体会

我认识不少技术不错的 C++/C 程序员，很少有人能拍拍胸脯说通晓指针与内存管理（包括我自己）。我最初学习 C 语言时特别怕指针，导致我开发第一个应用软件（约 1 万行 C 代码）时没有使用一个指针，全用数组来顶替指针，实在蠢笨得过分。躲避指针不是办法，后来我改写了这个软件，代码量缩小到原先的一半。

我的经验教训是：

- (1) 越是怕指针，就越要使用指针。不会正确使用指针，肯定算不上是合格的程序员。
- (2) 必须养成“使用调试器逐步跟踪程序”的习惯，只有这样才能发现问题的本质。

**strlen 与 sizeof 的区别**

---

1. **sizeof** 操作符的结果类型是 **size\_t**，它在头文件中 **typedef** 为 **unsigned int** 类型。  
该类型保证能容纳实现所建立的最大对象的字节大小。

2. **sizeof** 是算符，**strlen** 是函数。

3. **sizeof** 可以用类型做参数，**strlen** 只能用 **char\*** 做参数，且必须是以"\0"结尾的。  
**sizeof** 还可以用函数做参数，比如：

```
short f();  
printf("%d\n", sizeof(f()));
```

输出的结果是 **sizeof(short)**，即 2。

4. 数组做 **sizeof** 的参数不退化，传递给 **strlen** 就退化为指针了。

5. 大部分编译程序 在编译的时候就把 **sizeof** 计算过了 是类型或是变量的长度这就是  
**sizeof(x)** 可以用来定义数组维数的原因

**char str[20] = "0123456789";**

```
int a=strlen(str); //a=10;  
int b=sizeof(str); //而 b=20;
```

6. `strlen` 的结果要在运行的时候才能计算出来，时用来计算字符串的长度，不是类型占内存的大小。

7. `sizeof` 后如果是类型必须加括弧，如果是变量名可以不加括弧。这是因为 `sizeof` 是个操作符不是个函数。

8. 当适用于一个结构类型时或变量， `sizeof` 返回实际的大小，  
当适用一静态地空间数组， `sizeof` 归还全部数组的尺寸。  
`sizeof` 操作符不能返回动态地被分派了的数组或外部的数组的尺寸

9. 数组作为参数传给函数时传的是指针而不是数组，传递的是数组的首地址，  
如：

```
fun(char [8])  
fun(char [])  
都等价于 fun(char *)
```

在 C++ 里参数传递数组永远都是传递指向数组首元素的指针，编译器不知道数组的大小  
如果想在函数内知道数组的大小， 需要这样做：

进入函数后用 `memcpy` 拷贝出来，长度由另一个形参传进去

```
fun(unsigned char *p1, int len)  
{  
    unsigned char* buf = new unsigned char[len+1]  
    memcpy(buf, p1, len);  
}
```

10. 我们能常在用到 `sizeof` 和 `strlen` 的时候，通常是计算字符串数组的长度

看了上面的详细解释，发现两者的使用还是有区别的，从这个例子可以看得很清楚：

```
char str[20] = "0123456789";  
int a = strlen(str); //a=10; >>> strlen 计算字符串的长度，以结束符 0x00 为字符串结束。  
int b = sizeof(str); //而 b=20; >>> sizeof 计算的则是分配的数组 str[20] 所占的内存空间的  
大小，不受里面存储的内容改变。
```

上面是对静态数组处理的结果，如果是对指针，结果就不一样了

```
char* ss = "0123456789";
```

`sizeof(ss)` 结果 4 ==> `ss` 是指向字符串常量的字符指针, `sizeof` 获得的是一个指针的之所占的空间,应该是

长整型的, 所以是 4

`sizeof(*ss)` 结果 1 ==> `*ss` 是第一个字符 其实就是获得了字符串的第一位'0' 所占的内存空间, 是 `char` 类

型的, 占了 1 位

`strlen(ss)= 10 >>>` 如果要获得这个字符串的长度, 则一定要使用 `strlen`

```
int x=35;
```

```
char str[10];
```

//问: `strlen(str)`和 `sizeof(str)`的值分别是多少?

// `strlen(str)` 值不确定, `strlen` 根据'\0'确定字符串是否结束。

// `sizeof(str)=10` `sizeof` 一个数组为数组长度

```
strcpy(str,"www.it315.org"/*共 13 个字母*/);
```

//问:此时 `x` 和 `strlen(str)`的值分别是多少?

// `x` 为 35

// `strcpy(char* dest, const char* src)`

// 根据 `src` 来复制 `dest`, 依照 `src` 的'\0'决定复制的长度, 而 `dest` 必须要提供足够的长度, 这里会引起溢出, `strlen` 返回 13, 但是数组外部的数据已经被破坏

//(作者注:我下面给出了更确切的答案 )

```
str="it315.org";//编译能通过吗?
```

// 数组不能赋值, 只能初始化。`char str[10] = "it315.org";`

// 而且初始化时编译器会检查数组的长度与初始化串的长度是否匹配

```
char *pstr;
```

```
strcpy(pstr,"http://www.it315.org");
```

```
//上句编译能通过吗？运行时有问题吗?  
// 可以通过编译，但是 pstr 指向了常量区，运行时最好只做读操作，写操作不保险  
//(作者注:我下面给出了更确切的答案 )
```

```
13. const char *p1;  
char * const p2;  
//上面两句有什么区别吗?  
// const char* 和 char const* 一样，都是表示指向常量的字符指针。  
// char * const 表示指向字符的常量指针  
  
p1=(const char *)str;  
//如果是 p1=str; 编译能够通过吗？明白为什么要类型转换？类型转换的本质是什么？  
// 可以通过编译。关于常量与非常量指针的关系是这样的：  
// const 指针可以指向 const 或者非 const 区域，不会造成什么问题。  
// 非 const 指针不能指向 const 区域，会引起错误。
```

```
strcpy(p1,"abc");//编译能够通过吗?  
// 不能通过， strcpy( char*, const char*); char* 不能指向 const char*  
  
printf("%d",str);//有问题吗?  
// 没有问题，输出的是 str 的地址信息。  
  
pstr=3000;//编译能过吗？如果不可以，该如何修改以保证编译通过呢?  
// 不能通过， char* pstr 表示 pstr 是个字符指针，不能指向 3000 的整形变量。  
// 修改的话，可以这样： pstr = (char*)3000，把 pstr 指向 3000 这个地址;
```

```
long y=(long)pstr;//可以这样做吗?  
// 可以， y 的值为 pstr 所指的地址。不过如果是纯粹要地址的话，最好是用 unsigned long.  
  
int *p=str;  
*p=0x00313200;  
printf("%s",str);//会是什么效果？提示 0x31 对应字符'1',0x32 对应字符'2'。  
// 首先编译未必会过关，有些编译器可能不允许 int * 直接指向 char*。最好是改为 int *p =  
(int*)str;
```

```
// 过关了效果就是什么东西都没有。int *p=str; p 为 str 所指的地址, *p 表示修改了 str 所指向的内存。  
// 由于 sizeof(int) 在 32 位机上, int 有 4 个字节 (其实具体要看编译器的配置文件, 好像是 limit.h, 一般是 4 个字节) 所以修改了 str[0]-str[3]  
// 由于 0x00313200 头尾都是 0, 所以字符串为'\0'开头, 什么都打印不出来。这里有个 Big-endian 和 little-endian 的问题。以 0x31323334 为例  
// little-endian 的机器上面, 0x31323334 在内存中排列顺序为 34 33 32 31, 输出为 4321, 如 INTEL 芯片的 pc  
// big-endian 机器上面为 31 32 33 34, 输出为 1234, 如 IBM POWERPC
```

p=3000;//p+1 的结果会是多少?

```
// 3000+sizeof(int); 指针+1 均为原来地址加上 sizeof(指针所指的数据类型)
```

```
char *pc=new char[100];//上述语句在内存中占据几个内存块, 怎样的布局情况?  
// 本身 pc 会占用函数栈一个 4 字节的指针长度 (具体是否为 4 个字节要看机器和编译器),  
// new 会在堆上申请 100 个字节 sizeof(char) 的连续空间。
```

```
void test(char **p)  
{  
    *p=new char[100];  
}  
//这个编译函数有问题吗? 外面要调用这个函数, 该怎样传递参数?  
// 该程序没有问题。需要在函数中对指针所指的地址进行变化是必须传入指针的地址。  
// 原因是这样的: 如果传入的为指针本身, 在函数调用的时候, 实参会被复制一个实例, 这样就不是原来的指针了, 对该指针本身进行的任何改变都不能传递回去了。  
// 可以这样理解, 如果传入的参数为 int, 那么对 int 本身的值的改变就传不回去啦, 加个* 也是一样的。
```

```
//能明白 typedef int (*PFUN)(int x,int y)及其作用吗?  
// 定义了一个函数指针类型的宏, 这样 PFUN 就表示指向返回值为 int, 且同时带 2 个 int 参数的函数指针类型了。  
// 可以用来定义这样的变量:  
// 比如有个函数为 int fun( int x, int y );  
// PFUN p = fun;
```

//(作者注:我下面给出了更确切的答案)

---

下面是我的一点补充:

第二题:

```
int x=35;  
char str[10];  
strcpy(str,"www.it315.org"/*共 13 个字母*/);  
//问:此时 x 和 strlen(str)的值分别是多少?
```

答:strlen 的值为 13,在 VC++ 环境下,x 的值是要改变的(其他编译器下没试).虽然表面上看来,在程序中并没有修改 x 的值,但是实际运行的结果是上面的 x 的值发生了修改,这是因为 strcpy 以后,把多余的数据拷贝进了 str 的邻居(int 类型的 x)中,所以 x 的数据也就变了.这是一个曾让我刻骨铭心的问题,在我刚出道时遇到这个问题,虽然在朋友的帮助下解决了这个问题,但一直不明白 x 的值为何变了,只有最后走上培训教师的岗位,才开始梳理自己曾经的困惑,才开始总结以前的经验供学员们借鉴.我觉得这个题目的价值非常之大,它能引起学员对字符串拷贝越界问题的足够重视,并且通过这个问题更能明白字符串的处理是怎么回事,更能明白字符串与字符数组的关系:字符串就是一个字符数组,只是把这个字符数组用在处理串的函数中时,这些函数不考虑数组的长度,只是记住数组的首地址,从首地址开始处理,并在遇到 0 时结束处理,

第四题:

```
char *pstr;  
strcpy(pstr,"http://www.it315.org");  
//上句编译能通过吗? 运行时有问题吗?
```

答: 编译可以通过,但是 pstr 没有进行有效的初始化,它指向了一个不确定的内存区,运行时会出现内存不可写错误!

最后一题:

//能明白 `typedef int (*PFUN)(int x,int y)` 及其作用吗?

答: 函数指针最大的用处在于它可以被一个模板方法调用,这是我在学 java 的设计模式时领悟到的.例如,有两个函数的流程结构完全一致,只是内部调用的具体函数不同,如下所示:

```
void func1()  
{  
    //一段流程代码和面向方面的代理,如安全检查,日志记录等  
    int sum = add( x , y );  
    //一段流程代码和面向方面的代理,如安全检查,日志记录等  
}
```

```
void func2()
{
    //与 func1 完全相同的一段流程代码和面向方面的代理,如安全检查,日志记录等
    int difference = sub( x , y );
    //与 func1 完全相同的一段流程代码和面向方面的代理,如安全检查,日志记录等
}
```

那么,可以只定义一个函数,如下所示

```
void func(PFUNC p)
{
    //与 func1 完全相同的一段流程代码和面向方面的代理,如安全检查,日志记录等
    int difference = p( x , y );
    //与 func1 完全相同的一段流程代码和面向方面的代理,如安全检查,日志记录等
}
```

调用程序在调用时,让参数 p 分别指向 add 和 sub 函数就可以了.