华为面试题及答案

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理，假定你将那个变写错了，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

以下是引用片段：

```
#include
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

7、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 IT 人才网(it.ad0.cn) 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

8、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。

9、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

10、队列和栈有什么区别

队列先进先出，栈后进先出

11、写出下列代码的输出内容

以下是引用片段：

```
#include
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

12、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” 以下是引用片段：

```
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：

以下是引用片段：

```
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，以免造成内存汇泄露
return 0;
}
```

方法 2：

以下是引用片段：

```
#include
#include
main()
{
char str[]="hello,world";
int len=strlen(str);
char t;
for(int i=0; i
{
t=str;
str=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:

以下是引用片段：

```
#include
#include
int main(void)
{
long l;
char *str ="98765432";
l = atol(lstr);
```

printf("string = %s integer = %ld\n", str, l);
return(0);
}

13.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

14.直接链接两个信令点的一组链路称作什么

PPP 点到点连接

15.接入网用的是什么接口　V5 接口

16.voip 都用了那些协议

H.323 协议簇、SIP 协议、Skype 协议、H.248 和 MGCP 协议

17.软件测试都有那些种类

黑盒：针对系统功能的测试

白合：测试函数功能，各函数接口

18.确定模块的功能和模块的接口是在软件设计的那个队段完成的

概要设计阶段

19.

```
enum string
{x1,x2,x3=10,x4,x5,}x;
```

问 x= 0x801005，0x8010f4 ;

20.

```
unsigned char *p1;
unsigned long *p2;
p1=(unsigned char *)0x801000;
p2=(unsigned long *)0x810000;
```

请问 p1+5= ;

p2+5= ;

选择题:

21.Ethternet 链接到 Internet 用到以下那个协议

A.HDLC;B.ARP;C.UDP;D.TCP;E.ID

22.属于网络层协议的是:

A.TCP;B.IP;C.ICMP;D.X.25

23.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;

24.

```
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

找错题:

25.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[j][k]=0;
```

把循环语句内外换一下

26.

```
#define Max_CB 500
void LmiQueryCSmd(Struct MSgCB * pmsg)
{
unsigned char ucCmdNum;
......
for(ucCmdNum=0;ucCmdNum<Max_CB;ucCmdNum++)
{
......;
}
```

死循环,unsigned int 的取值范围是 0~255

27.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++);
```

答:结果与编译器相关,得到的可能不是平方值.

28.

```
typedef unsigned char BYTE
int examply_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;
}
.......
}
```

问答题:

29.IP Phone 的原理是什么?

IP 电话(又称 IP PHONE 或 VoIP)是建立在 IP 技术上的分组化、数字化传输技术,其基本原理是：通过语音压缩算法对语音数据进行压缩编码处理,然后把这些语音数据按 IP 等相关协议进行打包,经过 IP 网络把数据包传输到接收地,再把这些语音数据包串起来,经过解码解压处理后,恢复成原来的语音信号,从而达到由 IP 网络 传送语音的目的。

30.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

31.1 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?

1 号信令接续慢，但是稳定，可靠。

7 号信令的特点是：信令速度快，具有提供大量信令的潜力，具有改变和增加信令的灵活性，便于开放新业务，

在通话时可以随意处理信令，成本低。目前得到广泛应用。

32.列举 5 种以上的电话新业务

如“闹钟服务”、“免干扰服务”、“热线服务”、“转移呼叫”、“遇忙回叫”、“缺席用户服务”、“追查恶意呼叫”、“三方通话”、“会议电话”、“呼出限制”、“来电显示”、“虚拟网电话”等

```
int b;
b=SQUARE(a++);
```

答:结果与编译器相关,得到的可能不是平方值.

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 <filename.h> 和 #include “filename.h” 有什么区别? (5 分)

答：对于#include <filename.h> ，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。

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();
}
```

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。

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

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，但其原先的内容已经被清除，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 & operate =(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::operate =(const String &other)   // 13 分
  {
    // (1) 检查自赋值                 // 4 分
    if(this == &other)
                return *this;// 文章来源 草根 IT 网(www.caogenit.com)

// (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
    return *this;
```

一. 单选题(每题 4 分,15 题,共 60 分)

1.考虑函数原型 void hello(int a,int b=7,char* pszC=”*”),下面的函数调用钟,属于不合法调用的是:

A

hello(5) B.hello(5,8) C.hello(6,”#”) D.hello(0,0,”#”)

2.下面有关重载函数的说法中正确的是:

A.重载函数必须具有不同的返回值类型 B.重载函数形参个数必须不同

C.重载函数必须有不同的形参列表 D.重载函数名可以不同

3.分析一下程序的运行结果:

```
#include<iostream.h>
class CBase
{
public:
CBase(){cout<<”constructing CBase class”<<endl;}
~CBase(){cout<<”destructing CBase class”<<endl;}
};
class CSub : public CBase
{
public:
CSub(){cout<<”constructing CSub class”<<endl;}
~CSub(){cout<<”destructing CSub class”<<endl;}
};
void main()
{
CSub obj;
}
```

A. constructing CSub class
constructing CBase class
destructing CSub class
destructing CBase class
B. constructing CBase class
constructing CSub class
destructing CBase class
destructing CSub class
C. constructing CBase class
constructing CSub class
destructing CSub class
destructing CBase class
D. constructing CSub class
constructing CBase class
destructing CBase class
destructing CSub class

4.在一个 cpp 文件里面,定义了一个 static 类型的全局变量,下面一个正确的描述是:
A.只能在该 cpp 所在的编译模块中使用该变量
B.该变量的值是不可改变的
C.该变量不能在类的成员函数中引用
D.这种变量只能是基本类型(如 int,char)不能是 C++类型

5.观察下面一段代码:

```
class ClassA
{
public:
virtual ~ ClassA(){};
virtual void FunctionA(){};
};
class ClassB
{
public:
virtual void FunctionB(){};
};
class ClassC : public ClassA,public ClassB
{
public:
};
ClassC aObject;
ClassA* pA=&aObject;
ClassB* pB=&aObject;
ClassC* pC=&aObject;
```

关于 pA,pB,pC 的取值,下面的描述中正确的是:
A.pA,pB,pC 的取值相同.
B.pC=pA+pB
C.pA 和 pB 不相同
D.pC 不等于 pA 也不等于 pB

6.参照 1.5 的代码,假设定义了 ClassA* pA2,下面正确的代码是:
A.pA2=static_cast<ClassA*>(pB);
B.void* pVoid=static_cast<void*>(pB);
pA2=static_cast<ClassA*>(pVoid);
C.pA2=pB;
D.pA2=static_cast<ClassA*>(static_cast<ClassC*>(pB));

7.参照 1.5 的代码,下面那一个语句是不安全的:
A.delete pA B.delete pB C.delete pC

8.下列程序的运行结果为:

```
#include<iostream.h>
void main()
{
int a=2;
int b=++a;
cout<<a/6<<endl;
}
```

A.0.5 B.0 C0.7 D.0.6666666 –

9.有如下一段代码:

```
#define ADD(x,y) x+y
int m=3;
m+=m*ADD(m,m);
```

则 m 的值为:
A.15 B.12 C.18 D.58

10.如下是一个带权的图,图中结点 A 到结点 D 的关键路径的长度是:
A.13 B.15 C.28 D.58

11.下面的模板声明中,正确的是:
A.template<typename T1,T2>
B.template<class T1,T2>
C.template<class T1,class T2>
D.template<typename T1;typename T2>

12.在 Windows 编程中下面的说法正确的是:
A.两个窗口,他们的窗口句柄可以是相同的 B.两个窗口,他们的处理函数可以是相同的
C.两个窗口,他们的窗口句柄和窗口处理函数都不可以相同.

13.下面哪种情况下,B 不能隐式转换为 A?
A.class B:public A{}
B.class A:public B{}
C.class B{operator A();}
D.class A{A(const B&);}

14.某公司使用包过滤防火墙控制进出公司局域网的数据,在不考虑使用代理服务器的情况下,下面描述错误的是”该防火墙能够( )”.
A.使公司员工只能访问 Internet 上与其业务联系的公司的 IP 地址.
B.仅允许 HTTP 协议通过,不允许其他协议通过,例如 TCP/UDP.
C.使员工不能直接访问 FTP 服务器端口号为 21 的 FTP 地址.
D.仅允许公司中具有某些特定 IP 地址的计算机可以访问外部网络

15.数字字符 0 的 ASCII 值为 48,若有以下程序:

```
main()
```

```
{
char a=’1’,b=’2’;
printf(“%c,”,b++);
printf(“%d\n”,b-a);
}
```

程序运行之后的输出结果是:

A.3,2 B.50,2 C.2,2 D.2,50

二. 填空题(共 40 分)

本程序从正文文件 text.in 读入一篇英文短文,统计该短文中不同单词和它的出现次数,并按词典编辑顺序将单词及它的出现次数输出到正文文件 word.out 中.

程序用一棵有序二叉树存储这些单词及其出现的次数,一边读入一边建立.然后中序遍历该二叉树,将遍历经过的二叉树上的节点的内容输出.

程序中的外部函数

```
int getword(FILE* pFile,char* pszWordBuffer,int
nBufferLen);
```

从与 pFile 所对应的文件中读取单词置入 pszWordBuffer,并返回 1;若单词遇文件尾,已无单词可读时,则返回 0.

```
#include <stdio.h>
#include <malloc.h>
#include <ctype.h>
#include <string.h>
#define SOURCE_FILE “text.in”
#define OUTPUT_FILE “word.out”
#define MAX_WORD_LEN 128
typedef struct treenode
{
char szWord[MAX_WORD_LEN];
int nCount;
struct treenode* pLeft;
struct treenode* pRight;
}BNODE;
int getword(FILE* pFile,char* pasWordBuffer,int
nBufferLen);
void binary_tree(BNODE** ppNode,char* pszWord)
{
if(ppNode != NULL && pszWord != NULL)
{
BNODE* pCurrentNode = NULL;
BNODE* pMemoNode = NULL;
int nStrCmpRes=0;
____(1)_____;pCurrentNode=*ppNode
while(pCurrentNode)
{
/*寻找插入位置*/
nStrCmpRes = strcmp(pszWord,
___(2)___ );pCurrentNode->nCount
if(!nStrCmpRes)
{
___(3)___; pCurrentNode->nCount++
return;
}
else
{
___(4)___; pMemoNode=pCurrentNode
pCurrentNode = nStrCmpRes>0? pCurrentNode->pRight :
pCurrentNode->pLeft;
}
}
}
pCurrent=new BNODE;
if(pCurrentNode != NULL)
{
memset(pCurrentNode,0,sizeof(BNODE));
strncpy(pCurrentNode->szWord,pszWord,MAX_WORD_LE
N-1);
pCurrentNode->nCount=1;
}
if(pMemoNode==NULL)
{
___(5)___; *ppNode= pCurrentNode
}
else if(nStrCmpRes>0)
{
pMemoNode->pRight=pCurrentNode;
}
else
{
pMemoNode->pLeft=pCurrentNode;
}
}
void midorder(FILE* pFile,BNODE* pNode)
{
if(___(6)___) return;!pNode||!pFile
midorder(pFile,pNode->pLeft);
fprintf(pFile,”%s
%d\n”,pNode->szWord,pNode->nCount);
midorder(pFile,pNode->pRight);
}
void main()
{
FILE* pFile=NULL;
BNODE* pRootNode=NULL;
char szWord[MAX_WORD_LEN]={0};
pFile=fopen(SOURCE_FILE,”r”);
if(pFile==NULL)
{
```

```
printf(”Can’t open file %s\n”,SOURCE_FILE);
return;
}
while(getword(pFile,szWord,MAX_WORD_LEN)==1)
{
binary_tree(___(7)___);// pRootNode,szWord
}
fclose(pFile);
pFile=fopen(OUTPUT_FILE,”w”);
midorder(pFile,pRootNode);
fclose(pFile);
}
```

三. 附加题(每题 30 分,2 题,共 60 分)

1. 从程序健壮性进行分析,下面的 FillUserInfo 函数和 Main 函数分别存在什么问题?

```
#include <iostream>
#include <string>
#define MAX_NAME_LEN 20
struct USERINFO
{
int nAge;
char szName[MAX_NAME_LEN];
};
void FillUserInfo(USERINFO* parUserInfo)
{
stu::cout<<”请输入用户的个数:”;
int nCount=0;
std::cin>>nCount;
for(int i=0;i<nCount;i++)
{
std::cout<<”请输入年龄:”;
std::cin>>parUserInfo[i]->nAge;
std::string strName;
std::cout<<”请输入姓名:”;
std::cin>>strName;
strcpy(parUserInfo[i].szName,strName.c_str());
}
}
int main(int argc,char* argv[])
{
USERINFO arUserInfos[100]={0};
FillUserInfo(arUserInfos);
printf(”The first name is:”);
printf(arUserInfos[0].szName);
printf(”\n”);
return 0;
}
```

2. 假设你在编写一个使用多线程技术的程序,当程序中止运行时,需要怎样一个机制来安全有效的中止所有的线程?请描述其具体流程.

```
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，可对于这种情况（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惟一选择。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=.这个操作符象流操作符一样，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。

引 用是C++引入的新语言特性，是C++常用的一个重要内容之一，正确、灵活地使用引用，可以使程序简洁、高效。我在工作中发现，许多人使用它仅仅是想当 然，在某些微妙的场合，很容易出错，究其原由，大多因为没有搞清本源。故在本篇中我将对引用进行详细讨论，希望对大家更好地理解和使用引用起到抛砖引玉的 作用。

一、引用简介

引用就是某一变量（目标）的一个别名，对引用的操作与对变量直接操作完全一样。

引用的声明方法：类型标识符 &引用名=目标变量名；

【例1】：int a; int &ra=a; //定义引用ra,它是变量a的引用，即别名

说明：

（1）&在此不是求地址运算，而是起标识作用。

（2）类型标识符是指目标变量的类型。

（3）声明引用时，必须同时对其进行初始化。

（4）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，且不能再把该引用名作为其他变量名的别名。

ra=1; 等价于 a=1;

（5）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。故：对引用求地址，就是对目标变量求地址。&ra与&a相等。

（6）不能建立数组的引用。因为数组是一个由若干个元素所组成的集合，所以无法建立一个数组的别名。

二、引用应用

1、引用作为参数

引用的一个重要作用就是作为函数的参数。以前的C语言中函数参数传递是值传递，如果有大块数据作为参数传递的时候，采用的方案往往是指针，因为 这样可以避免将整块数据全部压栈，可以提高程序的效率。但是现在（C++中）又增加了一种同样有效率的选择（在某些特殊情况下又是必须的选择），就是引 用。

【例2】：

```
void swap(int &p1, int &p2) //此处函数的形参p1, p2都是引用
{ int p; p=p1; p1=p2; p2=p; }
```

为在程序中调用该函数，则相应的主调函数的调用点处，直接以变量作为实参进行调用即可，而不需要实参变量有任何的特殊要求。如：对应上面定义的swap 函数，相应的主调函数可写为：

```
main( )
{
  int a,b;
  cin>>a>>b; //输入 a,b 两变量的值
  swap(a,b); //直接以变量 a 和 b 作为实参调用 swap 函数
  cout<<a<< ' ' <<b; //输出结果
}
```

上述程序运行时，如果输入数据10 20并回车后，则输出结果为20 10。

由【例2】可看出：

（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。

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。

2、常引用

常引用声明方式：const 类型标识符 &引用名=目标变量名；

用这种方式声明的引用，不能通过引用对目标变量的值进行修改,从而使引用的目标成为const，达到了引用的安全性。

【例3】：

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

这不光是让代码更健壮，也有些其它方面的需要。

【例4】：假设有如下函数声明：

```
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 。

3、引用作为返回值

要以引用返回函数值，则函数定义时要按以下格式：
类型标识符 &函数名（形参列表及类型说明）
{函数体}

说明：

（1）以引用返回函数值，定义函数时需要在函数名前加&

（2）用引用返回一个函数值的最大好处是，在内存中不产生被返回值的副本。

【例 5】以下程序中定义了一个普通的函数 fn1（它用返回值的方法返回函数值），另外一个函数 fn2，它以引用的方法返回函数值。

```
#include <iostream.h>
float temp; //定义全局变量 temp
float fn1(float r); //声明函数 fn1
float &fn2(float r); //声明函数 fn2
float fn1(float r) //定义函数 fn1，它以返回值的方法返回函数值
{
  temp=(float)(r*r*3.14);
  return temp;
}
float &fn2(float r) //定义函数 fn2，它以引用方式返回函数值
{
  temp=(float)(r*r*3.14);
  return temp;
}
void main() //主函数
{
  float a=fn1(10.0); //第 1 种情况，系统生成要返回值的副本（即临时变量）
  float &b=fn1(10.0); //第 2 种情况，可能会出错(不同 C++系统有不同规定)
  //不能从被调函数中返回一个临时变量或局部变量的引用
  float c=fn2(10.0); //第 3 种情况，系统不生成返回值的副本
  //可以从被调函数中返回一个全局变量的引用
  float &d=fn2(10.0); //第 4 种情况，系统不生成返回值的副本
  //可以从被调函数中返回一个全局变量的引用
  cout<<a<<c<<d;
}
```

引用作为返回值，必须遵守以下规则：

（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，可对于这种情况（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）引用与一些操作符的重载：

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惟一选择。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=。这个操作符象流操作符一样，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 6】 测试用返回引用的函数值作为赋值表达式的左值。

```
#include <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。

4、引用和多态

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 7】：

```
class    A;
class    B：public A{……};
B    b;
A    &Ref = b; // 用派生类对象初始化基类对象的引用
```

Ref 只能用来访问派生类对象中从基类继承下来的成员，是基类引用指向派生类。如果 A 类中定义有虚函数，并且在 B 类中重写了这个虚函数，就可以通过 Ref 产生多态效果。

三、引用总结

（1）在引用的使用中，单纯给某个变量取个别名是毫无意义的，引用的目的主要用于在函数参数传递中，解决大块数据或对象的传递效率和空间不如意的问题。

（2）用引用传递函数的参数，能保证参数传递中不产生副本，提高传递的效率，且通过 const 的使用，保证了引用传递的安全性。

（3）引用与指针的区别是，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。

（4）使用引用的时机。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。

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)

```
# i nclude <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 Ox010A）

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,
mumber.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；Ox42 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 & operater =(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。文章来源 草根 IT 网 (www.caogenit.com)

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,other.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. #i nclude<file.h> 与 #i nclude "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
 # i nclude "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"
{
 # i nclude "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 */
 # i nclude "cExample.h"
int add( int x, int y )
{
return x + y;
}
// c++实现文件，调用 add：cppFile.cpp
extern "C"
{
 # i nclude "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 文章来源 草根 IT 网(www.caogenit.com)
 # i nclude "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： # i nclude "cExample.h" */
extern int add( int x, int y );
int main( int argc, char* argv[] )
{
add( 2, 3 );
```

```
return 0;
}
```

15.extern “C”含义

extern "C"包含双重含义，其一：被它修饰的目标是“extern”的；其二：被它修饰的目标是“C”的。

1）被 extern “C”限定的函数或变量是 extern 类型的；

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他模块中使用。

注意：extern int a；

仅仅是在声明一个变量，并不是定义变量 a，并未为 a 分配内存空间。变量 a 在所有模块中作为一种全局变量只能被定义一次，否则会出现连接错误。

通常，在模块的头文件中对模块提供给其他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。

与 extern 对应的关键字是 static，被它修饰的全局变量和函数只能在本模块中使用。因此，一个函数或变量只可能被本模块使用时，其不可能被 extern "C"修饰。

2）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

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（不同的编译器可能产生的名字不同，但是都采用了相同的机制）。_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)编译产生的符号是不相同的，后者为_foo_int_float。

extern "C"作用：实现 C++与 C 及其它语言的混合编程。

3）extern "C"的惯用法

A）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。

例如：

```
/*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;
}
```

B)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。

例如：

```
//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;
}
```

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; ...}

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：是指子类重新定义复类虚函数的方法。

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 接口也就完全不同于 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。

一、选择题 （每小题 2 分，共 12 分）

1 C++源文件的缺省扩展名为（ ）A

A.cpp

B.exe

C.obj

D.lik

2 程序运行中需要从键盘上输入多于一个数据时，各数据之间应该使用（ ）符号作为分隔符 A

A.空格或逗号

B.逗号或回车

C.逗号或分号

D.空格或回车

3 在每个 C++程序中都必须包含有这样一个函数，该函数的名称为（ ）A

A.main

B.MAIN

C.name

D.function

4 假定 a 为一个短整型（short int）数组名，则元素 a[8]的地址为()B

A.a+4

B.a+8

C.a+16

D.a+32

5 假定 AB 为一个类，则执行“ABa(4),b[4],*p[5]”语句时，自动调用该类够造函数的次数为（ ）B

A.4

B.5

C.8

D.13

6 当需要使用 istrstream 流类定义一个流类对象并联系一个字符串时，应在文件开始使用#include 命令，使之包含（ ）文件 A

A.iostream.h

B.iomanip.h

C.fstream.h
D.ststrea.h

二 填空题（每小题 3 分，共 30 分）

1. 在 C++中，函数的参数有两种传递方式，它们是值传递和（址传递）。
2. 当一个成员函数被调用时，该成员函数的（this 指针）指向调用它的对象。
3. 在公有继承的情况下，基类数据成员在派生类中的访问权限 （私有不可以访问，其他同基类一致） 。
4. 用 new 申请某一个类的动态对象数组时，在该类中必须能够匹配到 （无参） 构造函数，否则应用程序会产生一个编译错误。
5. 静态数据成员在类外进行初始化，且静态数据成员的一个拷贝被类的所有对象 （共享）。
6. 面向对象的程序设计有四大特征，它们是（封装，多态，继承，抽象） 。
7. 在 C++类中，有一种不能定义对象的类，这样的类只能被继承，称之为 （虚基类） ，定义该类至少具有一个 （虚函数） 。
8. 在 C++类中，const 关键字可以修饰对象和成员函数，const 对象不能 （作为左值）。

9.假定一个枚举类型的定义为“enum RB{ab,ac,ad,ae}x=ad;”则 x 的值为（2）

10.若需要把一个类 AB 定义一个类 CD 的友元素，则应在类 CD 的定义中加入一条语句为（friend class AB）。

三 给出下列程序运行后的输出结果（每小题 5 分，共 20 分）

1. 下面程序的运行结果是（3） 。

```
#include <stdio.h>
void main()
{
   char a=’a',b=’j';
   float x;
   x=(b-a)/(’F'-’A');//文章来源 草根 IT 网 (www.caogenit.com)
   printf(”%d\n”,(int)(3.14*x));
}
```

2. 下面程序的运行结果是（ ）。

```
#include “iostream.h”
void main( )
{
   int i=1;
while (i<=15){
      i++;
      if (i%3!=2) continue;
      else cout <<” i=” <<i<<endl;
   }
}
```

```
i=2
i=5
i=8
i=11
i=14
```

3. 下面程序的运行结果是（ ）。

```
#include “iostream.h”
class test
{
private:
   int num;
   float fl;
public:
   test( );
   int getint( ){return num;}
   float getfloat( ){return fl;}
   ~test( );
};
test::test( )
{
   cout << “Initalizing default” << endl;
   num=0;fl=0.0;
}
test::~test( )
{
   cout << “Desdtructor is active” << endl;
}
void main( )
{
   test array[2];
   cout << array[1].getint( )<< ” ” << array[1].getfloat( ) <<endl;
}
```

```
Initalizing defaul
Initalizing defaul
0 0.0
Desdtructor is active
Desdtructor is active
```

4. 下面程序的运行结果是（ ）。

```
#include <iostream.h>
class A
{
public:
   A(){cout<<” A::A() called.\n” ;}
   virtual ~A(){cout<<” A::~A() called.\n” ;}
};
class B:public A
{
public:
   B(int i){
```

```
    cout<<”B::B() called.\n”;
    buf=new char[i];}
    virtual ~B()
    {
      delete []buf;
      cout<<”B::~B() called.\n”;
    }
  private:
    char *buf;
  };
  void fun(A *a)
  {
    delete a;
  }
void main()
  {
    A *a=new B(15);
    fun(a);
  }
```

A::A() called.
B::B() called.
B::~B() called.
A::~A() called.

5. 下面程序的运行结果是（ ）。

```
  #include <stdio.h>
  int a[ ]={1,3,5,7,9};
  int *p[ ]={a,a+1,a+2,a+3,a+4};
  void main( )
  {
    printf(”%d\t%d\t%d\n”,a[4],*(a+2),*p[1]);

printf(”%d\t%d\t%d\n”,**(p+1)+a[2],*(p+4)-*(p+0),*(a+3)
%a[4]);
  }
```

9 5 3
8 4 7

四、 问答题（每小题 5 分，共 20 分）

1. 若程序员没有定义拷贝构造函数，则编译器自动生成一个缺省的拷贝构造函数，它可能会产生什么问题?

浅拷贝问题，主要原因为类中如果有指针成员变量时，当调用拷贝构造函数时只拷贝地址从而使两个对象的指针变量指向了一个地址空间。

2. 简述成员函数、全局函数和友元函数的差别。

成员函数只能由该类所实例化的对象来进行调用。[静态成员除外]
全局函数可以在任意位置进行调用。
友元函数可以让本类和友元类对象调用。

3. 简述结构化的程序设计、面向对象的程序设计的基本思想。

结构化程序设计为从程序代码的开始处按照顺序方式执行至代码的结束位置。是一种顺序的方式，函数与变量没有明显的联系
面向对象主要把处理事情的事物和方法结合为一体成为一个类，一个类具备处理一件事情的数据变量和处理方法，把数据和方法有机的结合为了一体，使每一件事情都具备一定的独立性，形成一个模块。增加了内聚性，降低了耦合性。同时也增加了代码的可读性以及代码的重用性。

4. 结构 struct 和类 class 有什么异同?

在 c 语言中 struct 只能对数据进行聚合，而 c++的 class 把数据以及对数据的处理方法也同时聚合为一体，增加了内聚性。
此外 class 拥有可再生性和可抽象性,实现的代码的复用。集中体现在派生的功能和多态的功能。
同时 class 也比 struct 具备更好的封装性，体现在三种访问权限上。
在 C++ 中的 struct 和 class 的结构基本一致，只是 struct 的默认权限为 Public 而 class 为 private。

五 下列 shape 类是一个表示形状的抽象类，area( )为求图形面积的函数，total( )则是一个通用的用以求不同形状的图形面积总和的函数。请从 shape 类派生三角形类(triangle)、矩形类（rectangle），并给出具体的求面积函数（18 分）

```
Class Shape
{
Public:
 Shape(){}
 ~shape(){}
 Virtual int area();
}
/* 求所有面积    */
Int total( Vector< Shape*> vecshape);
Vector< Shape*> vecShape;
Int total(Vector< Shape*> vecshape)
{
 Vector< Shape*>::iterator VecIt;
 Int ntotal = 0;
 For( VecIt=vecShape.begin(); VecIt!=vecShape.end();
VecIt++)
{
 ntotal += (*VecIt)->area();
}
Return ntotal;
}
Class trangle :public shape
{
Public:
```

```
 Trangle(){}
 ~trangle(){}
 Int area();
Private:
 Int m_nheigh;
 Int m_nwidth;
}
/* 求三角形面积     */
Int Trangle::area()
{
 Return m_nheigh*m_nwidth/2;
}
Class rectangle :public shape
{
Public:
 Rectangle(){}
 ~rectangle(){}
 Int area();
Private:
 Int m_nheigh;
 Int m_nwidth;
}
/* 求矩形面积     */
Int Rectangle::area()
{
 Return m_nheigh*m_nwidth;
}
```

威盛公司软件 C++英文笔试题面试题

1.How good do you see yours programming skills?Please circle your answe
rs

C: Fair/Good/Excellent

C++: Fair/Good/Excellent

2.Please estimate your programming experince:

How many lines of code are your biggest C Program ever written exclud
ing the

standard linked library?

How many lines of code are your biggest C++ program ever written excl
uding the

standard linked library?

3.Please explain the following terms

Data Encapsulation

Inheritance

Polymorphism

4.What is a virtual base class?How do you declare such a class?How woul
d you use

it in a design?

5.What is a template or container class?How do you declare such a class
?

6.Which are the access control levels for C++ language?

7.What is RTTI?How do you achieve RTTI in your design?

8.What are the major differences between static and non-static member f
unctions?

9.How do you call a regular member function from a static member functi
on?

Please use psendo-code to provide your answer

10.How do you declare/define a type of pointer to a class member functi
on?

Please use psendo-code to provide your answer

11.Please explain the following tyoes:

Here is a short lise of combinations and their meanings:

1.Reference-Can change the referenced object

2.Const-Reference

3.Const-Pointer-

4.Pointer-Const-

5.Const-Pointer-Const-

```
CFoo Instance
CFoo &ReferenceToInstance=Instance;
//1
const CFoo &ConstReferenceToInstance=Instance;    //2
const CFoo *pConstPointer=&Instance;                //3
CFoo *const pPointerConst=&Instance;                //4
const CF00 *const pPointerConst=&Instance;          //5
```

12.What are top-down and bottom-up approach?How do you usually use them
?

13.Please use pseudo-code to design a set of stack operations with temp
late

14.Please use pseudo-code to design a set of double linked list operati
ons with

template

Optional Questions:(for extra credits)

15.Please write a unix makefile for Question13

16.Please explain these common sections:text,data,bss

微软 C,C++面试题大全版(二)

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 intialization 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?

1、static_cast Operator

MSDN:

The expression static_cast < type-id > (expression) converts expression to the type of type-id based solely on the types present in the expression. No run-time type check is made to ensure the safety of the conversion.

Syntax

static_cast < type-id > (expression)

The static_cast operator can be used for operations such as converting a pointer to a base class to a pointer to a derived class. Such conversions are not always safe. For example:

```
class B { ... };
class D : public B { ... };
void f(B* pb, D* pd)
{
   D* pd2 = static_cast<D*>(pb);        // not safe, pb may
                                        // point to just B
   B* pb2 = static_cast<B*>(pd);        // safe conversion
   ...
}
```

In contrast to dynamic_cast, no run-time check is made on the static_cast conversion of pb. The object pointed to by pb may not be an object of type D, in which case the use of *pd2 could be disastrous. For instance, calling a function that is a member of the D class, but not the B class, could result in an access violation.

The dynamic_cast and static_cast operators move a pointer throughout a class hierarchy. However, static_cast relies exclusively on the information provided in the cast statement and can therefore be unsafe. For example:

```
class B { ... };
class D : public B { ... };
void f(B* pb)
{
  D* pd1 = dynamic_cast<D*>(pb);
  D* pd2 = static_cast<D*>(pb);
}
```

If pb really points to an object of type D, then pd1 and pd2 will get the same value. They will also get the same value if pb == 0.

If pb points to an object of type B and not to the complete D class, then dynamic_cast will know enough to return zero. However, static_cast relies on the programmer’s assertion that pb points to an object of type D and simply returns a pointer to that supposed D object.

Consequently, static_cast can do the inverse of implicit conversions, in which case the results are undefined. It is left to the programmer to ensure that the results of a static_cast conversion are safe.

This behavior also applies to types other than class types. For instance, static_cast can be used to convert from an int to a

char. However, the resulting char may not have enough bits to hold the entire int value. Again, it is left to the programmer to ensure that the results of a static_cast conversion are safe.
The static_cast operator can also be used to perform any implicit conversion, including standard conversions and user-defined conversions. For example:

```
typedef unsigned char BYTE
void f()
{
  char ch;
  int i = 65;
  float f = 2.5;
  double dbl;
  ch = static_cast<char>(i);          // int to char
  dbl = static_cast<double>(f);       // float to double
  ...
  i = static_cast<BYTE>(ch);
  ...
}
```

The static_cast operator can explicitly convert an integral value to an enumeration type. If the value of the integral type does not fall within the range of enumeration values, the resulting enumeration value is undefined.
The static_cast operator converts a null pointer value to the null pointer value of the destination type.
Any expression can be explicitly converted to type void by the static_cast operator. The destination void type can optionally include the const, volatile, or __unaligned attribute.
The static_cast operator cannot cast away the const, volatile, or __unaligned attributes.
static_cast 在功能上基本上与 C 风格的类型转换一样强大，含义也一样。它也有功能上限制。例如，你不能用 static_cast 象用 C 风 格的类型转换一样把 struct 转换成 int 类型或者把 double 类型转换成指针类型，另外，static_cast 不能从表达式中去除 const 属 性，因为另一个新的类型转换操作符 const_cast 有这样的功能。
2、const_cast Operator
MSDN:
The const_cast operator can be used to remove the const, volatile, and __unaligned attribute(s) from a class.
Syntax
const_cast < type-id > (expression)
A pointer to any object type or a pointer to a data member can be explicitly converted to a type that is identical except for the const, volatile, and __unaligned qualifiers. For pointers and references, the result will refer to the original object. For pointers to data members, the result will refer to the same member as the original (uncast) pointer to data member. Depending on the type of the referenced object, a write operation through the resulting pointer, reference, or pointer to data member might produce undefined behavior.
The const_cast operator converts a null pointer value to the null pointer value of the destination type.
const_cast 用于类型转换掉表达式的 const 或 volatileness 属性。通过使用 const_cast，你向人们和编译器强调你通过类型转换想做的只是改变一些东西的 constness 或者 volatileness 属性。这个含义被编译器所约束。如果你试图使用 const_cast 来完成修改 constness 或者 volatileness 属性之外的事情，你的类型转换将被拒绝。
3、dynamic_cast Operator
MSDN:
The expression dynamic_cast<type-id>(expression) converts the operand expression to an object of type type-id. The type-id must be a pointer or a reference to a previously defined class type or a “pointer to void”. The type of expression must be a pointer if type-id is a pointer, or an l-value if type-id is a reference.
Syntax
dynamic_cast < type-id > (expression)
If type-id is a pointer to an unambiguous accessible direct or indirect base class of expression, a pointer to the unique subobject of type type-id is the result. For example:

```
class B { ... };
class C : public B { ... };
class D : public C { ... };
void f(D* pd)
{
  C* pc = dynamic_cast<C*>(pd);    // ok: C is a direct base class
                                   // pc points to C subobject of pd
  B* pb = dynamic_cast<B*>(pd);    // ok: B is an indirect base class
                                   // pb points to B subobject of pd
  ...
}
```

This type of conversion is called an “upcast” because it moves a pointer up a class hierarchy, from a derived class to a class it is derived from. An upcast is an implicit conversion.
If type-id is void*, a run-time check is made to determine the actual type of expression. The result is a pointer to the complete object pointed to by expression. For example:

```
class A { ... };
class B { ... };
void f()
{
  A* pa = new A;
  B* pb = new B;
  void* pv = dynamic_cast<void*>(pa);
```

```
  // pv now points to an object of type A
  ...
  pv = dynamic_cast<void*>(pb);
  // pv now points to an object of type B
}
```

If type-id is not void*, a run-time check is made to see if the object pointed to by expression can be converted to the type pointed to by type-id.

If the type of expression is a base class of the type of type-id, a run-time check is made to see if expression actually points to a complete object of the type of type-id. If this is true, the result is a pointer to a complete object of the type of type-id.

For example:

```
class B { ... };
class D : public B { ... };
void f()
{
  B* pb = new D;              // unclear but ok
  B* pb2 = new B;
  D* pd = dynamic_cast<D*>(pb);      // ok: pb actually points to a D
  ...
  D* pd2 = dynamic_cast<D*>(pb2);    //error: pb2 points to a B, not a D
                        // pd2 == NULL
  ...
}
```

This type of conversion is called a “downcast” because it moves a pointer down a class hierarchy, from a given class to a class derived from it.

dynamic_cast，它被用于安全地沿着类的继承关系向下进行类型转换。这就是说，你能用 dynamic_cast 把指向基类的指针或引用转 换成指向其派生类或其兄弟类的指针或引用，而且你能知道转换是否成功。失败的转换将返回空指针（当对指针进行类型转换时）或者抛出异常（当对引用进行类型 转换时）。

dynamic_casts 在帮助你浏览继承层次上是有限制的。它不能被用于缺乏虚函数的类型上，也不能用它来转换掉 constness。

4、reinterpret_cast Operator

MSDN:

The reinterpret_cast operator allows any pointer to be converted into any other pointer type. It also allows any integral type to be converted into any pointer type and vice versa. Misuse of the reinterpret_cast operator can easily be unsafe. Unless the desired conversion is inherently low-level, you should use one of the other cast operators.

Syntax

reinterpret_cast < type-id > (expression)

The reinterpret_cast operator can be used for conversions such as char* to int*, or One_class* to Unrelated_class*, which are inherently unsafe.

The result of a reinterpret_cast cannot safely be used for anything other than being cast back to its original type. Other uses are, at best, nonportable.

The reinterpret_cast operator cannot cast away the const, volatile, or __unaligned attributes.

使用这个操作符的类型转换，其的转换结果几乎都是执行期定义（implementation-defined）。因此，使用 reinterpret_casts 的代码很难移植。 reinterpret_casts 的最普通的用途就是在函数指针类型之间进行转换。

比如转换函数指针的代码是不可移植的（C++不保证所有的函数指针都被用一样的方法表示），在一些情况下这样的转换会产生不正确的结果（参见条款 M31），所以你应该避免转换函数指针类型。

5、如果你使用的编译器缺乏对新的类型转换方式的支持，你可以用传统的类型转换方法代替 static_cast, const_cast, 以及 reinterpret_cast。也可以用下面的宏替换来模拟新的类型转换语法：

```
#define static_cast(TYPE,EXPR)          ((TYPE)(EXPR))
#define const_cast(TYPE,EXPR)           ((TYPE)(EXPR))
#define reinterpret_cast(TYPE,EXPR)     ((TYPE)(EXPR))
```

这些模拟不会象真实的操作符一样安全，但是当你的编译器可以支持新的的类型转换时，它们可以简化你把代码升级的过程。

没有一个容易的方法来模拟 dynamic_cast 的操作,但是很多函数库提供了函数，安全地在派生类与基类之间进行类型转换。如果你没有这些函数 而你有必须进行这样的类型转换，你也可以回到 C 风格的类型转换方法上，但是这样的话你将不能获知类型转换是否失败。当然，你也可以定义一个宏来模拟 dynamic_cast 的功能，就象模拟其它的类型转换一样：

```
#define dynamic_cast(TYPE,EXPR)      (TYPE)(EXPR)
```

请记住，这个模拟并不能完全实现 dynamic_cast 的功能，它没有办法知道转换是否失败。

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料不到的错误。

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 ） + 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:

```
# i nclude<iostream>
# i nclude<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<<" ";
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 ;//文章来源 草根 IT 网(www.caogenit.com)
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 ;
}
```

41. 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 > maxnumber )
{
sec_max = maxnumber ;
maxnumber = data ;
}
else
{
if ( data > sec_max )
sec_max = data ;//文章来源  草根 IT 网
(www.caogenit.com)
}
}
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;
}
```

46.

输入一个 n ，然后在屏幕上打印出 NxN 的矩阵!

例如，

输入一个 3 ，则

1 2 3

8 9 4

7 6 5

输入一个 4 ，则

1　　2　3　4

12 13 14 5

11 16 15 6

10　9　8　7

参考答案：

```
#include<stdio.h>
#include<conio.h>
#define N 10
void printCube(int a[][N],int n);
void main()
{
   int a[N][N],n;
   printf("input n:\n");
   scanf("%d",&n);
   printCube(&a[0],n);
   getch();
}
void printCube(int a[][N],int n)
```

```
{
   int i,j,round=1;
   int m=1;
   for(i=0;i<n;i++)
a[0]=m++; 文章来源 草根 IT 网(www.caogenit.com)
   for(i=n-1;i>=n/2;i--)
   {
for(j=round;j<=i;j++)
   a[j]=m++;
for(j=i;j>=round;j--)
   a[j-1]=m++;
for(j=i;j>round;j--)
   a[j-1][round-1]=m++;
for(j=round;j<i;j++)
   a[round][j]=m++;
round++;
   }
   for(i=0;i<n;i++){
for(j=0;j<n;j++)
  printf("%3d",a[j]);
printf("\n");
   }
}
```

选择排序、快速排序、希尔排序、堆排序不是稳定的排序算法，
冒泡排序、插入排序、归并排序和基数排序是稳定的排序算法。

冒泡法：
这是最原始，也是众所周知的最慢的算法了。他的名字的由来因为它的工作看来象是冒泡： 复杂度为 O(n*n)。当数据为正序，将不会有交换。复杂度为 O(0)。
直接插入排序：O(n*n)
选择排序：O(n*n)
快速排序：平均时间复杂度 log2(n)*n，所有内部排序方法中最高好的，大多数情况下总是最好的。
归并排序：log2(n)*n
堆排序：log2(n)*n
希尔排序：算法的复杂度为 n 的 1.2 次幂

这里我没有给出行为的分析，因为这个很简单，我们直接来分析算法：
首先我们考虑最理想的情况
1.数组的大小是 2 的幂，这样分下去始终可以被 2 整除。假设为 2 的 k 次方，即 k=log2(n)。
2.每次我们选择的值刚好是中间值，这样，数组才可以被等分。
第一层递归，循环 n 次，第二层循环 2*(n/2).....
所以共有 n+2(n/2)+4(n/4)+...+n*(n/n) =
n+n+n+...+n=k*n=log2(n)*n
所以算法复杂度为 O(log2(n)*n)
其他的情况只会比这种情况差，最差的情况是每次选择到的 middle 都是最小值或最大值，那么他将变成交换法（由于使用了递归，情况更糟）。但是你认为这种情况发生的几率有多大？？呵呵，你完全不必担心这个问题。实践证明，大多数的情况，快速排序总是最好的。
如果你担心这个问题，你可以使用堆排序，这是一种稳定的 O(log2(n)*n)算法，但是通常情况下速度要慢 于快速排序（因为要重组堆）。

这几天笔试了好几次了，连续碰到一个关于常见排序算法稳定性判别的问题，往往还是多选，对于我以及和我一样拿不准的同学可不是一个能轻易下结论的题目，当然如果你笔试之前已经记住了数据结构书上哪些是稳定的，哪些不是稳定的，做起来应该可以轻松搞定。
本文是针对老是记不住这个或者想真正明白到底为什么是稳定或者不稳定的人准备的。

首先，排序算法的稳定性大家应该都知道，通俗地讲就是能保证排序前 2 个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。在简单形式化一下，如果 Ai = Aj, Ai 原来在位置前，排序后 Ai 还是要在 Aj 位置前。

其次，说一下稳定性的好处。排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位相同的元素其顺序再高位也相同时是不会改变的。另外，如果排序算法稳定，对基于比较的排序算法而言，元素交换的次数可能会少一些(个人感觉，没有证实)。

回到主题，现在分析一下常见的排序算法的稳定性，每个都给出简单的理由。

(1)冒泡排序

冒泡排序就是把小的元素往前调或者把大的元素往后调。比较是相邻的两个元素比较，交换也发生在这两个元素之间。所以，如果两个元素相等，我想你是不会再无聊地把他们俩交换一下的；如果两个相等的元素没有相邻，那么即使通过前面的两两交换把两个相邻起来，这时候也不会交换，所以相同元素的前后顺序并没有改变，所以冒泡排序是一种稳定排序算法。

(2)选择排序

选择排序是给每个位置选择当前元素最小的，比如给第一个位置选择最小的，在剩余元素里面给第二个元素选择第二小的，依次类推，直到第 n-1 个元素，第 n 个元素不用选择了，因为只剩下它一个最大的元素了。那么，在一趟选择，如果当前元素比一个元素小，而该小的元素又出现在一个和当前元素相等的元素后面，那

么交换后稳定性就被破坏了。比较拗口，举个例子，序列 5 8 5 2 9， 我们知道第一遍选择第 1 个元素 5 会和 2 交换，那么原序列中 2 个 5 的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

(3)插入排序

插入排序是在一个已经有序的小序列的基础上，一次插入一个元素。当然，刚开始这个有序的小序列只有 1 个元素，就是第一个元素。比较是从有序序列的末尾开始，也就是想要插入的元素和已经有序的最大者开始比起，如果比它大则直接插入在其后面，否则一直往前找直到找到它该插入的位置。如果碰见一个和插入元素相等的，那么插入元素把想插入的元素放在相等元素的后面。所以，相等元素的前后顺序没有改变，从原无序序列出去的顺序就是排好序后的顺序，所以插入排序是稳定的。

(4)快速排序

快速排序有两个方向，左边的 i 下标一直往右走，当 a[i] <= a[center_index]，其中 center_index 是中枢元素的数组下标，一般取为数组第 0 个元素。而右边的 j 下标一直往左走，当 a[j] > a[center_index]。如果 i 和 j 都走不动了，i <= j, 交换 a[i]和 a[j],重复上面的过程，直到 i>j。 交换 a[j]和 a[center_index]，完成一趟快速排序。在中枢元素和 a[j]交换的时候，很有可能把前面的元素的稳定性打乱，比如序列为 5 3 3 4 3 8 9 10 11， 现在中枢元素 5 和 3(第 5 个元素，下标从 1 开始计)交换就会把元素 3 的稳定性打乱，所以快速排序是一个不稳定的排序算法，不稳定发生在中枢元素和 a[j]交换的时刻。

(5)归并排序

归并排序是把序列递归地分成短序列，递归出口是短序列只有 1 个元素(认为直接有序)或者 2 个序列(1 次比较和交换),然后把各个有序的段序列合并成一个有序的长序列，不断合并直到原序列全部排好序。可以发现，在 1 个或 2 个元素时，1 个元素不会交换，2 个元素如果大小相等也没有人故意交换，这不会破坏稳定性。那么，在短的有序序列合并的过程中，稳定是是否受到破坏?没有，合并过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序也是稳定的排序算法。

(6)基数排序

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序，最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以其是稳定的排序算法。

(7)希尔排序(shell)

希尔排序是按照不同步长对元素进行插入排序，当刚开始元素很无序的时候，步长最大，所以插入排序的元素个数很少，速度很快；当元素基本有序了，步长很小，插入排序对于有序的序列效率很高。所以，希尔排序的时间复杂度会比 o(n^2)好一些。由于多次插入排序，我们知道一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱，所以 shell 排序是不稳定的。

(8)堆排序

我们知道堆的结构是节点 i 的孩子为 2*i 和 2*i+1 节点，大顶堆要求父节点大于等于其 2 个子节点，小顶堆要求父节点小于等于其 2 个子节点。在一个长为 n 的序列，堆排序的过程是从第 n/2 开始和其子节点共 3 个值选择最大(大顶堆)或者最小(小顶堆),这 3 个元素之间的选择当然不会破坏稳定性。但当为 n /2-1, n/2-2, ...1 这些个父节点选择元素时，就会破坏稳定性。有可能第 n/2 个父节点交换把后面一个元素交换过去了，而第 n/2-1 个父节点把后面一个相同的元素没有交换，那么这 2 个相同的元素之间的稳定性就被破坏了。所以，堆排序不是稳定的排序算法

1 快速排序（QuickSort）

快速排序是一个就地排序，分而治之，大规模递归的算法。从本质上来说，它是归并排序的就地版本。快速排序可以由下面四步组成。

（1） 如果不多于 1 个数据，直接返回。

（2） 一般选择序列最左边的值作为支点数据。

（3） 将序列分成 2 部分，一部分都大于支点数据，另外一部分都小于支点数据。

（4） 对两边利用递归排序数列。

快速排序比大部分排序算法都要快。尽管我们可以在某些特殊的情况下写出比快速排序快的算法，但是就通常情况而言，没有比它更快的了。快速排序是递归的，对于内存非常有限的机器来说，它不是一个好的选择。

2 归并排序（MergeSort）

归并排序先分解要排序的序列，从 1 分成 2，2 分成 4，依次分解，当分解到只有 1 个一组的时候，就可以排序这些分组，然后依次合并回原来的序列中，这样就可以排序所有数据。合并排序比堆排序稍微快一点，但是需要比堆排序多一倍的内存空间，因为它需要一个额外的数组。

3 堆排序（HeapSort）

堆排序适合于数据量非常大的场合（百万数据）。

堆排序不需要大量的递归或者多维的暂存数组。这对于数据量非常巨大的序列是合适的。比如超过数百万条记录，因为快速排序，归并排序都使用递归来设计算法，在数据量非常大的时候，可能会发生堆栈溢出错误。

堆排序会将所有的数据建成一个堆，最大的数据在堆顶，然后将堆顶数据和序列的最后一个数据交换。接下来再

次重建堆，交换数据，依次下去，就可以排序所有的数据。

4 Shell 排序（ShellSort）

Shell 排序通过将数据分成不同的组，先对每一组进行排序，然后再对所有的元素进行一次插入排序，以减少数据交换和移动的次数。平均效率是 O(nlogn)。其中分组的合理性会对算法产生重要的影响。现在多用 D.E.Knuth 的分组方法。

Shell 排序比冒泡排序快 5 倍，比插入排序大致快 2 倍。Shell 排序比起 QuickSort，MergeSort，HeapSort 慢很多。但是它相对比较简单，它适合于数据量在 5000 以下并且速度并不是特别重要的场合。它对于数据量较小的数列重复排序是非常好的。

5 插入排序（InsertSort）

插入排序通过把序列中的值插入一个已经排序好的序列中，直到该序列的结束。插入排序是对冒泡排序的改进。它比冒泡排序快 2 倍。一般不用在数据大于 1000 的场合下使用插入排序，或者重复排序超过 200 数据项的序列。

6 冒泡排序（BubbleSort）

冒泡排序是最慢的排序算法。在实际运用中它是效率最低的算法。它通过一趟又一趟地比较数组中的每一个元素，使较大的数据下沉，较小的数据上升。它是 O(n^2) 的算法。

7 交换排序（ExchangeSort）和选择排序（SelectSort）

这两种排序方法都是交换方法的排序算法，效率都是 O(n2)。在实际应用中处于和冒泡排序基本相同的地位。它们只是排序算法发展的初级阶段，在实际中使用较少。

8 基数排序（RadixSort）

基数排序和通常的排序算法并不走同样的路线。它是一种比较新颖的算法，但是它只能用于整数的排序，如果我们要把同样的办法运用到浮点数上，我们必须了解浮点数的存储格式，并通过特殊的方式将浮点数映射到整数上，然后再映射回去，这是非常麻烦的事情，因此，它的使用同样也不多。而且，最重要的是，这样算法也需要较多的存储空间。

9 总结

下面是一个总的表格，大致总结了我们常见的所有的排序算法的特点。

排序法	平均时间	最差情形	稳定度	额外空间	备注
冒泡	$O(n^2)$	$O(n^2)$	稳定	O(1)	n小时较好
交换	$O(n^2)$	$O(n^2)$	不稳定	O(1)	n小时较好
选择	$O(n^2)$	$O(n^2)$	不稳定	O(1)	n小时较好
插入	$O(n^2)$	$O(n^2)$	稳定	O(1)	大部分已排序时较好
基数	$O(\log_R B)$	$O(\log_R B)$	稳定	O(n)	B是真数(0-9), R是基数(个十百)
Shell	O(nlogn)	$O(n^s)$ $1<s<2$	不稳定	O(1)	s是所选分组
快速	O(nlogn)	$O(n^2)$	不稳定	O(nlogn)	n大时较好
归并	O(nlogn)	O(nlogn)	稳定	O(1)	n大时较好
堆	O(nlogn)	O(nlogn)	不稳定	O(1)	n大时较好

浙大网新 C/C++面试,笔试题

1. 写一个“标准”宏 MIN，这个宏输入两个参数并返回较小的一个。

#define MIN(A,B) ((A) <= (B) (A) : (B))

这个测试是为下面的目的而设的：

1). 标识#define 在宏中应用的基本知识。这是很重要的，因为直到嵌入(inline)操作符变为标准 C 的一部分，宏是方便产生嵌入代码的唯一方法，对于嵌入式系统来说，为了能达到要求的性能，嵌入代码经常是必须的方法。

2). 三重条件操作符的知识。这个操作符存在 C 语言中的原因是它使得编译器能产生比 if-then-else 更优化的代码，了解这个用法是很重要的。

3). 懂得在宏中小心地把参数用括号括起来

4). 我也用这个问题开始讨论宏的副作用，例如：当你写下面的代码时会发生什么事？

least = MIN(*p++, b);

3. 预处理器标识#error 的目的是什么？

如果你不知道答案，请看参考文献 1。这问题对区分一个正常的伙计和一个书呆子是很有用的。只有书呆子才会读 C 语言课本的附录去找出象这种

问题的答案。当然如果你不是在找一个书呆子，那么应试者最好希望自己不要知道答案。

死循环（Infinite loops）

4. 嵌入式系统中经常要用到无限循环，你怎么样用 C 编写死循环呢？//文章来源 草根 IT 网 (www.caogenit.com)

这个问题用几个解决方案。我首选的方案是：

while(1) { }

一些程序员更喜欢如下方案：

for(;;) { }

这个实现方式让我为难，因为这个语法没有确切表达到底怎么回事。如果一个应试者给出这个作为方案，我将用这个作为一个机会去探究他们这样做的

基本原理。如果他们的基本答案是：“我被教着这样做，但从没有想到过为什么。”这会给我留下一个坏印象。

第三个方案是用 goto

Loop:

…
goto Loop;
应试者如给出上面的方案，这说明或者他是一个汇编语言程序员（这也许是好事）或者他是一个想进入新领域的 BASIC/FORTRAN 程序员。
数据声明（Data declarations）
5. 用变量 a 给出下面的定义
a) 一个整型数（An integer）
b) 一个指向整型数的指针（A pointer to an integer）
c) 一个指向指针的的指针，它指向的指针是指向一个整型数（A pointer to a pointer to an integer）
d) 一个有 10 个整型数的数组（An array of 10 integers）
e) 一个有 10 个指针的数组，该指针是指向一个整型数的（An array of 10 pointers to integers）
f) 一个指向有 10 个整型数数组的指针（A pointer to an array of 10 integers）
g) 一个指向函数的指针，该函数有一个整型参数并返回一个整型数（A pointer to a function that takes an integer as an argument and returns an integer）
h) 一个有 10 个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数(An array of ten pointers to functions that take an integer argument and return an integer)
答案是：
a) int a; // An integer
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer
人们经常声称这里有几个问题是那种要翻一下书才能回答的问题，我同意这种说法。当我写这篇文章时，为了确定语法的正确性，我的确查了一下书。
但是当我被面试的时候，我期望被问到这个问题（或者相近的问题）。因为在被面试的这段时间里，我确定我知道这个问题的答案。应试者如果不知道
所有的答案（或至少大部分答案），那么也就没有为这次面试做准备，如果该面试者没有为这次面试做准备，那么他又能为什么出准备呢？
Static
6. 关键字 static 的作用是什么？
这个简单的问题很少有人能回答完全。在 C 语言中，关键字 static 有三个明显的作用：
1). 在函数体，一个被声明为静态的变量在这一函数被调用过程中维持其值不变。
2). 在模块内（但在函数体外），一个被声明为静态的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。它是一个本地的全局变量。
3). 在模块内，一个被声明为静态的函数只可被这一模块内的其它函数调用。那就是，这个函数被限制在声明它的模块的本地范围内使用。
大多数应试者能正确回答第一部分，一部分能正确回答第二部分，同是很少的人能懂得第三部分。这是一个应试者的严重的缺点，因为他显然不懂得本地化数据和代码范围的好处和重要性。
Const
7. 关键字 const 是什么含意？
我只要一听到被面试者说：“const 意味着常数”，我就知道我正在和一个业余者打交道。去年 Dan Saks 已经在他的文章里完全概括了 const 的所有用法，因此 ESP(译者：Embedded Systems Programming)的每一位读者应该非常熟悉 const 能做什么和不能做什么.如果你从没有读到那篇文章，只要能说出 const 意味着“只读”就可 以了。尽管这个答案不是完全的答案，但我接受它作为一个正确的答案。（如果你想知道更详细的答案，仔细读一下 Saks 的文章吧。）如果应试者能正确回答这 个问题，我将问他一个附加的问题：下面的声明都是什么意思？
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
前两个的作用是一样，a 是一个常整型数。第三个意味着 a 是一个指向常整型数的指针（也就是，整型数是不可修改的，但指针可以）。第四个意思 a 是一个 指向整型数的常指针（也就是说，指针指向的整型数是可以修改的，但指针是不可修改的）。最后一个意味着 a 是一个指向常整型数的常指针（也就是说，指针指向 的整型数是不可修改的，同时指针也是不可修改的）。如果应试者能正确回答这些问题，那么他就给我留下了一个好印象。顺带提一句，也许你可能会问，即使不用 关键字 const，也还是能很容易写出功能正确的程序，那么我为什么还要如此看重关键字 const 呢？我也如下的几下理由：
1). 关键字 const 的作用是为给读你代码的人传达非常有用的信息，实际上，声明一个参数为常量是为了告诉了用户这个参数的应用目的。如果你曾花很多时间清理其它人留下的垃圾，你就会很快学会感谢这点多余的信息。（当然，懂得用 const 的程序员很少会留下的垃圾让别人来清理的。）
2). 通过给优化器一些附加的信息，使用关键字 const 也许能产生更紧凑的代码。

3). 合理地使用关键字 const 可以使编译器很自然地保护那些不希望被改变的参数，防止其被无意的代码修改。简而言之，这样可以减少 bug 的出现。

Volatile

8. 关键字 volatile 有什么含意 并给出三个不同的例子。

一个定义为 volatile 的变量是说这变量可能会被意想不到地改变，这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。下面是 volatile 变量的几个例子：

1). 并行设备的硬件寄存器（如：状态寄存器）

2). 一个中断服务子程序中会访问到的非自动变量 (Non-automatic variables)

3). 多线程应用中被几个任务共享的变量

回答不出这个问题的人是不会被雇佣的。我认为这是区分 C 程序员和嵌入式系统程序员的最基本的问题。嵌入式系统程序员经常同硬件、中断、RTOS 等等打交道，所用这些都要求 volatile 变量。不懂得 volatile 内容将会带来灾难。

假设被面试者正确地回答了这是问题（嗯，怀疑这否会是这样），我将稍微深究一下，看一下这家伙是不是直正懂得 volatile 完全的重要性。

1). 一个参数既可以是 const 还可以是 volatile 吗？解释为什么。

2). 一个指针可以是 volatile 吗？解释为什么。

3). 下面的函数有什么错误：

```
int square(volatile int *ptr)
{
return *ptr * *ptr;
}
```

下面是答案：

1). 是的。一个例子是只读的状态寄存器。它是 volatile 因为它可能被意想不到地改变。它是 const 因为程序不应该试图去修改它。

2). 是的。尽管这并不很常见。一个例子是当一个中服务子程序修该一个指向一个 buffer 的指针时。

3). 这段代码的有个恶作剧。这段代码的目的是用来返指针*ptr 指向值的平方，但是，由于*ptr 指向一个 volatile 型参数，编译器将产生类似下面的代码：

```
int square(volatile int *ptr)
{
int a,b;
a = *ptr;
b = *ptr;
return a * b;
}
```

由于*ptr 的值可能被意想不到地该变，因此 a 和 b 可能是不同的。结果，这段代码可能返不是你所期望的平方值！正确的代码如下：

```
long square(volatile int *ptr)
{
int a;
a = *ptr;
return a * a;
}
```

位操作（Bit manipulation）

9. 嵌入式系统总是要用户对变量或寄存器进行位操作。给定一个整型变量 a，写两段代码，第一个设置 a 的 bit 3，第二个清除 a 的 bit 3。在以上两个操作中，要保持其它位不变。

对这个问题有三种基本的反应

1). 不知道如何下手。该被面者从没做过任何嵌入式系统的工作。

2). 用 bit fields。Bit fields 是被扔到 C 语言死角的东西，它保证你的代码在不同编译器之间是不可移植的，同时也保证了的你的代码是不可重用的。我最近不幸看到 Infineon 为其较复杂的通信芯片写的驱动程序，它用到了 bit fields 因此完全对我无用，因为我的编译器用其它的方式来实现 bit fields 的。从道德讲：永远不要让一个非嵌入式的家伙粘实际硬件的边。

3). 用 #defines 和 bit masks 操作。这是一个有极高可移植性的方法，是应该被用到的方法。最佳的解决方案如下：

```
#define BIT3 (0 × 1<<3)
static int a;
void set_bit3(void)
{
a |= BIT3;
}
void clear_bit3(void)
{
a &= ~BIT3;
}
```

一些人喜欢为设置和清除值而定义一个掩码同时定义一些说明常数，这也是可以接受的。我希望看到几个要点：说明常数、|=和&=~操作。

访问固定的内存位置（Accessing fixed memory locations）

10. 嵌入式系统经常具有要求程序员去访问某特定的内存位置的特点。在某工程中，要求设置一绝对地址为 0 × 67a9 的整型变量的值为 0xaa66。编译器是一个纯粹的 ANSI 编译器。写代码去完成这一任务。

这一问题测试你是否知道为了访问一绝对地址把一个整型数强制转换（typecast）为一指针是合法的。这一问题的实现方式随着个人风格不同而不同。典型的类似代码如下：

```
int *ptr;
ptr = (int *)0 × 67a9;
*ptr = 0xaa55;
```

一个较晦涩的方法是：

*(int * const)(0 × 67a9) = 0xaa55;

即使你的品味更接近第二种方案，但我建议你在面试时使用第一种方案。

中断（Interrupts）

11. 中断是嵌入式系统中重要的组成部分，这导致了很多编译开发商提供一种扩展—让标准 C 支持中断。具代表事实是，产生了一个新的关键字 __interrupt。下面的代码就使用了__interrupt 关键字去定义了一个中断服务子程序(ISR)，请评论一下这段代码的。//文章来源 草根 IT 网(www.caogenit.com)

```
__interrupt double compute_area (double radius)
{
double area = PI * radius * radius;
printf(" Area = %f", area);
return area;
}
```

这个函数有太多的错误了，以至让人不知从何说起了：

1). ISR 不能返回一个值。如果你不懂这个，那么你不会被雇用的。

2). ISR 不能传递参数。如果你没有看到这一点，你被雇用的机会等同第一项。

3). 在许多的处理器/编译器中，浮点一般都是不可重入的。有些处理器/编译器需要让额处的寄存器入栈，有些处理器/编译器就是不允许在 ISR 中做浮点运算。此外，ISR 应该是短而有效率的，在 ISR 中做浮点运算是不明智的。

4). 与第三点一脉相承，printf()经常有重入和性能上的问题。如果你丢掉了第三和第四点，我不会太为难你的。不用说，如果你能得到后两点，那么你的被雇用前景越来越光明了。

代码例子（Code examples）

12．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");
}
```

这个问题测试你是否懂得 C 语言中的整数自动转换原则，我发现有些开发者懂得极少这些东西。不管如何，这无符号整型问题的答案是输出是 “>6”。原因是当表达式中存在有符号类型和无符号类型时所有的操作数都自动转换为无符号类型。因此-20 变成了一个非常大的正整数，所以该表达式计算出的结果大于 6。这一点对于应当频繁用到无符号数据类型的嵌入式系统来说是丰常重要的。如果你答错 了这个问题，你也就到了得不到这份工作的边缘。

13. 评价下面的代码片断：

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF;
/*1's complement of zero */
```

对于一个 int 型不是 16 位的处理器为说，上面的代码是不正确的。应编写如下：

```
unsigned int compzero = ~0;
```

这一问题真正能揭露出应试者是否懂得处理器字长的重要性。在我的经验里，好的嵌入式程序员非常准确地明白硬件的细节和它的局限，然而 PC 机程序往往把硬件作为一个无法避免的烦恼。

到了这个阶段，应试者或者完全垂头丧气了或者信心满满志在必得。如果显然应试者不是很好，那么这个测试就在这里结束了。但如果显然应试者做得不错， 那么我就扔出下面的追加问题，这些问题是比较难的，我想仅仅非常优秀的应试者能做得不错。提出这些问题，我希望更多看到应试者应付问题的方法，而不是答 案。不管如何，你就当是这个娱乐吧…

动态内存分配（Dynamic memory allocation）

14. 尽管不像非嵌入式计算机那么常见，嵌入式系统还是有从堆（heap）中动态分配内存的过程的。那么嵌入式系统中，动态分配内存可能发生的问题是什么？

这里，我期望应试者能提到内存碎片，碎片收集的问题，变量的持行时间等等。这个主题已经在 ESP 杂志中被广泛地讨论过了（主要是 P.J. Plauger, 他的解释远远超过我这里能提到的任何解释），所有回过头看一下这些杂志吧！让应试者进入一种虚假的安全感觉后，我拿出这么一个小节目：下面的代码片段的输 出是什么，为什么？

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL)
puts("Got a null pointer");
else
puts("Got a valid pointer");
```

这是一个有趣的问题。最近在我的一个同事不经意把 0 值传给了函数 malloc，得到了一个合法的指针之后，我才想到这个问题。这就是上面的代码，该 代码的输出是“Got a valid pointer”。我用这个来开始讨论这样的一问题，看看被面试者是否想到库例程这样做是正确。得到正确的答案固然重要，但解决问题的方法和你做决定的基 本原理更重要些。

Typedef

15. Typedef 在 C 语言中频繁用以声明一个已经存在的数据类型的同义字。也可以用预处理器做类似的事。例如，思考一下下面的例子：

```
#define dPS struct s *
typedef struct s * tPS;
```

以上两种情况的意图都是要定义 dPS 和 tPS 作为一个指向结构 s 指针。哪种方法更好呢？（如果有的话）为什么？

这是一个非常微妙的问题，任何人答对这个问题（正当的原因）是应当被恭喜的。答案是：typedef 更好。思考下面的例子：

dPS p1,p2;

tPS p3,p4;

第一个扩展为

struct s * p1, p2;

上面的代码定义 p1 为一个指向结构的指，p2 为一个实际的结构，这也许不是你想要的。第二个例子正确地定义了 p3 和 p4 两个指针。

晦涩的语法

16. C 语言同意一些令人震惊的结构,下面的结构是合法的吗，如果是它做些什么？

int a = 5, b = 7, c;

c = a+++b;

这个问题将做为这个测验的一个愉快的结尾。不管你相不相信，上面的例子是完全合乎语法的。问题是编译器如何处理它？水平不高的编译作者实际上会争论这个问题，根据最处理原则，编译器应当能处理尽可能所有合法的用法。因此，上面的代码被处理成：

c = a++ + b;

因此, 这段代码持行后 a = 6, b = 7, c = 12。

如果你知道答案，或猜出正确答案，做得好。如果你不知道答案，我也不把这个当作问题。我发现这个问题的最大好处是:这是一个关于代码编写风格，代码的可读性，代码的可修改性的好的话题