**class Person{**

**private:**

**in age;**

**public:**

**Person()**

**{**

**cout<<"无参构造函数"<<endl;**

**}**

**Person(const Person& other)**

**{**

**cout<<"拷贝构造函数"<<endl;**

**}**

**void setAge(int age)**

**{**

**this->age = age;**

**}**

**void show()**

**{**

**cout<<age<<endl;**

**}**

**};**

**void fun(int a)**

**{**

**cout<<"a:"<<a<<endl;**

**a+2;**

**}**

**void fun(Person p)**

**{**

**p.setage(23);**

**}**

**void fun(Person& p) //////////两种示例 查看调用不同**

**{**

**p.setage(23);**

**}**

**int main()**

**{**

**int b=10;**

**fun(b);**

**cout<<"b:"<<b<<endl;////////////看两个数的变化 有无异议**

**Person a;**

**fun(a);**

**a.show();**

**return 0;**

**}**

**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**

**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{**

**t=a[i];**

**a[i]=a[j];**

**a[j]=t;**

**}**

**cout<<a;}**

**//第二种**

**string str;**

**cin>>str;**

**str.replace;**

**cout<<a;**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**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，但时易于管理。**

**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;**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**这道题目的结果是什么啊?**

**define 只是定义而已，在编择时只是简单代换XX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**

**1**

**2**

**3**

**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;**

**...**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**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"))**

**{**

**...**

**}**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**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);**

**}**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**==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{**

**SRM\_no %= MAX\_SRM;**

**if(MY\_SRM.state==IDLE)**

**{**

**break;**

**}**

**}**

**if(I>=MAX\_SRM)**

**return (NULL\_SRM);**

**else**

**return SRM\_no;**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**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;**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**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"<}**

**else**

**{**

**cout<<"16 bit"<}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**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； llinux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。**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**17.在不用第三方参数的情况下，交换两个参数的值**

**#include**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**方法二：**

**i^=j;**

**j^=i;**

**i^=j;**

**1**

**2**

**3**

**方法三：**

**// 用加减实现，而且不会溢出**

**a = a+b-(b=a)**

**1**

**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）**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**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**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**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;**

**1**

**2**

**3**

**4**

**5**

**6**

**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:**

**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。**

**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。**

**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。**

**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;**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**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贮空间。**

**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;**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**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;**

**};**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**答:**

**会引起无限递归**

**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;**

**};**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**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**

**解析C语言中的sizeof**

**一、sizeof的概念**

**sizeof是C语言的一种单目操作符，如C语言的其他操作符++、–等。它并不是函数。sizeof操作符以字节形式给出了其操作数的存储大小。操作数可以是一个表达式或括在括号内的类型名。操作数的存储大小由操作数的类型决定。**

**二、sizeof的使用方法**

**1、用于数据类型**

**sizeof使用形式：sizeof（type）**

**数据类型必须用括号括住。如sizeof（int）。**

**2、用于变量**

**sizeof使用形式：sizeof（var\_name）或sizeof　var\_name**

**变量名可以不用括号括住。如sizeof　(var\_name)，sizeof　var\_name等都是正确形式。带括号的用法更普遍，大多数程序员采用这种形式。**

**注意：sizeof操作符不能用于函数类型，不完全类型或位字段。不完全类型指具有未知存储大小的数据类型，如未知存储大小的数组类型、未知内容的结构或联合类型、void类型等。**

**如sizeof(max)若此时变量max定义为int　max(),sizeof(char\_v)　若此时char\_v定义为char　char\_v　[MAX]且MAX未知，sizeof(void)都不是正确形式。**

**三、sizeof的结果**

**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。**

**1、若操作数具有类型char、unsigned　char或signed　char，其结果等于1。**

**ANSI　C正式规定字符类型为1字节。**

**2、int、unsigned　int　、short　int、unsigned　short　、long　int　、unsigned　long　、 float、double、long　double类型的sizeof　在ANSI　C中没有具体规定，大小依赖于实现，一般可能分别为2、2、2、2、 4、4、4、8、10。**

**3、当操作数是指针时，sizeof依赖于编译器。例如Microsoft　C/C++7.0中，near类指针字节数为2，far、huge类指针字节数为4。一般Unix的指针字节数为4。**

**4、当操作数具有数组类型时，其结果是数组的总字节数。**

**5、联合类型操作数的sizeof是其最大字节成员的字节数。结构类型操作数的sizeof是这种类型对象的总字节数，包括任何垫补在内。**

**让我们看如下结构：**

**struct　{char　b;　double　x;}　a;**

**在某些机器上sizeof（a）=12，而一般sizeof（char）+　sizeof（double）=9。**

**这是因为编译器在考虑对齐问题时，在结构中插入空位以控制各成员对象的地址对齐。如double类型的结构成员x要放在被4整除的地址。**

**6、如果操作数是函数中的数组形参或函数类型的形参，sizeof给出其指针的大小。**

**四、sizeof与其他操作符的关系**

**sizeof的优先级为2级，比/、%等3级运算符优先级高。它可以与其他操作符一起组成表达式。如i\*sizeof（int）；其中i为int类型变量。**

**五、sizeof的主要用途**

**1、sizeof操作符的一个主要用途是与存储分配和I/O系统那样的例程进行通信。例如：**

**void　\*malloc（size\_t　size）,**

**size\_t　fread(void　\*　ptr,size\_t　size,size\_t　nmemb,FILE　\*　stream)。**

**2、sizeof的另一个的主要用途是计算数组中元素的个数。例如：**

**void　\*　memset（void　\*　s,int　c,sizeof(s)）。**

**六、建议**

**由于操作数的字节数在实现时可能出现变化，建议在涉及到操作数字节大小时用sizeof来代替常量计算。**

**本文主要包括二个部分，第一部分重点介绍在VC中，怎么样采用sizeof来求结构的大小，以及容易出现的问题，并给出解决问题的方法，第二部分总结出VC中sizeof的主要用法。**

**1、 sizeof应用在结构上的情况**

**请看下面的结构：**

**struct MyStruct**

**{**

**double dda1;**

**char dda;**

**int type**

**};**

**1**

**2**

**3**

**4**

**5**

**6**

**对结构MyStruct采用sizeof会出现什么结果呢？sizeof(MyStruct)为多少呢？也许你会这样求：**

**sizeof(MyStruct)=sizeof(double)+sizeof(char)+sizeof(int)=13**

**但是当在VC中测试上面结构的大小时，你会发现sizeof(MyStruct)为16。你知道为什么在VC中会得出这样一个结果吗？**

**其实，这是VC对变量存储的一个特殊处理。为了提高CPU的存储速度，VC对一些变量的起始地址做了"对齐"处理。在默认情况下，VC规定各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。下面列出常用类型的对齐方式(vc6.0,32位系统)。**

**类型**

**对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）**

**Char**

**偏移量必须为sizeof(char)即1的倍数**

**int**

**偏移量必须为sizeof(int)即4的倍数**

**float**

**偏移量必须为sizeof(float)即4的倍数**

**double**

**偏移量必须为sizeof(double)即8的倍数**

**Short**

**偏移量必须为sizeof(short)即2的倍数**

**各成员变量在存放的时候根据在结构中出现的顺序依次申请空间，同时按照上面的对齐方式调整位置，空缺的字节VC会自动填充。同时VC为了确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍数，所以在为最后一个成员变量申请空间后，还会根据需要自动填充空缺的字节。**

**下面用前面的例子来说明VC到底怎么样来存放结构的。**

**struct MyStruct**

**{**

**double dda1;**

**char dda;**

**int type**

**}；**

**1**

**2**

**3**

**4**

**5**

**6**

**为上面的结构分配空间的时候，VC根据成员变量出现的顺序和对齐方式，先为第一个成员dda1分配空间，其起始地址跟结构的起始地址相同（刚好偏移量0刚好为sizeof(double)的倍数），该成员变量占用sizeof(double)=8个字节；接下来为第二个成员dda分配空间，这时下一个可以分配的地址对于结构的起始地址的偏移量为8，是sizeof(char)的倍数，所以把dda存放在偏移量为8的地方满足对齐方式，该成员变量占用 sizeof(char)=1个字节；接下来为第三个成员type分配空间，这时下一个可以分配的地址对于结构的起始地址的偏移量为9，不是sizeof (int)=4的倍数，为了满足对齐方式对偏移量的约束问题，VC自动填充3个字节（这三个字节没有放什么东西），这时下一个可以分配的地址对于结构的起始地址的偏移量为12，刚好是sizeof(int)=4的倍数，所以把type存放在偏移量为12的地方，该成员变量占用sizeof(int)=4个字节；这时整个结构的成员变量已经都分配了空间，总的占用的空间大小为：8+1+3+4=16，刚好为结构的字节边界数（即结构中占用最大空间的类型所占用的字节数sizeof(double)=8）的倍数，所以没有空缺的字节需要填充。所以整个结构的大小为：sizeof(MyStruct)=8+1+ 3+4=16，其中有3个字节是VC自动填充的，没有放任何有意义的东西。**

**下面再举个例子，交换一下上面的MyStruct的成员变量的位置，使它变成下面的情况：**

**struct MyStruct**

**{**

**char dda;**

**double dda1;**

**int type**

**}；**

**1**

**2**

**3**

**4**

**5**

**6**

**这个结构占用的空间为多大呢？在VC6.0环境下，可以得到sizeof(MyStruc)为24。结合上面提到的分配空间的一些原则，分析下VC怎么样为上面的结构分配空间的。（简单说明）**

**struct MyStruct**

**{**

**char dda;//偏移量为0，满足对齐方式，dda占用1个字节；**

**double dda1;//下一个可用的地址的偏移量为1，不是sizeof(double)=8**

**//的倍数，需要补足7个字节才能使偏移量变为8（满足对齐**

**//方式），因此VC自动填充7个字节，dda1存放在偏移量为8**

**//的地址上，它占用8个字节。**

**int type；//下一个可用的地址的偏移量为16，是sizeof(int)=4的倍**

**//数，满足int的对齐方式，所以不需要VC自动填充，type存**

**//放在偏移量为16的地址上，它占用4个字节。**

**}；//所有成员变量都分配了空间，空间总的大小为1+7+8+4=20，不是结构**

**//的节边界数（即结构中占用最大空间的类型所占用的字节数sizeof**

**//(double)=8）的倍数，所以需要填充4个字节，以满足结构的大小为**

**//sizeof(double)=8的倍数。**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**所以该结构总的大小为：sizeof(MyStruc)为1+7+8+4+4=24。其中总的有7+4=11个字节是VC自动填充的，没有放任何有意义的东西。**

**VC对结构的存储的特殊处理确实提高CPU存储变量的速度，但是有时候也带来了一些麻烦，我们也屏蔽掉变量默认的对齐方式，自己可以设定变量的对齐方式。**

**VC 中提供了#pragma pack(n)来设定变量以n字节对齐方式。n字节对齐就是说变量存放的起始地址的偏移量有两种情况：第一、如果n大于等于该变量所占用的字节数，那么偏移量必须满足默认的对齐方式，第二、如果n小于该变量的类型所占用的字节数，那么偏移量为n的倍数，不用满足默认的对齐方式。结构的总大小也有个约束条件，分下面两种情况：如果n大于所有成员变量类型所占用的字节数，那么结构的总大小必须为占用空间最大的变量占用的空间数的倍数；**

**否则必须为n的倍数。下面举例说明其用法。**

**#pragma pack(push) //保存对齐状态**

**#pragma pack(4)//设定为4字节对齐**

**struct test**

**{**

**char m1;**

**double m4;**

**int m3;**

**};**

**#pragma pack(pop)//恢复对齐状态**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**以上结构的大小为16，下面分析其存储情况，首先为m1分配空间，其偏移量为0，满足我们自己设定的对齐方式（4字节对齐），m1占用1个字节。接着开始为 m4分配空间，这时其偏移量为1，需要补足3个字节，这样使偏移量满足为n=4的倍数（因为sizeof(double)大于n）,m4占用8个字节。接着为m3分配空间，这时其偏移量为12，满足为4的倍数，m3占用4个字节。这时已经为所有成员变量分配了空间，共分配了16个字节，满足为n的倍数。如果把上面的#pragma pack(4)改为#pragma pack(16)，那么我们可以得到结构的大小为24。（请读者自己分析）**

**2、 sizeof用法总结**

**在VC中，sizeof有着许多的用法，而且很容易引起一些错误。下面根据sizeof后面的参数对sizeof的用法做个总结。**

**A． 参数为数据类型或者为一般变量。例如sizeof(int),sizeof(long)等等。这种情况要注意的是不同系统系统或者不同编译器得到的结果可能是不同的。例如int类型在16位系统中占2个字节，在32位系统中占4个字节。**

**B． 参数为数组或指针。下面举例说明.**

**int a[50]; //sizeof(a)=4\*50=200; 求数组所占的空间大小**

**int \*a=new int[50];// sizeof(a)=4; a为一个指针，sizeof(a)是求指针**

**//的大小,在32位系统中，当然是占4个字节。**

**C． 参数为结构或类。Sizeof应用在类和结构的处理情况是相同的。但有两点需要注意，第一、结构或者类中的静态成员不对结构或者类的大小产生影响，因为静态变量的存储位置与结构或者类的实例地址无关。**

**第二、没有成员变量的结构或类的大小为1，因为必须保证结构或类的每一**

**个实例在内存中都有唯一的地址。**

**下面举例说明，**

**Class Test{int a;static double c};//sizeof(Test)=4.**

**Test \*s;//sizeof(s)=4,s为一个指针。**

**Class test1{ };//sizeof(test1)=1;**

**D． 参数为其他。下面举例说明。**

**int func(char s[5]);**

**{**

**cout<//数的参数在传递的时候系统处理为一个指针，所**

**//以sizeof(s)实际上为求指针的大小。**

**return 1;**

**}**

**sizeof(func(“1234”))=4//因为func的返回类型为int，所以相当于**

**//求sizeof(int).**

**以上为sizeof的基本用法，在实际的使用中要注意分析VC的分配变量的分配策略，这样的话可以避免一些错误。**

**25.i最后等于多少?**

**int i = 1;**

**int j = i++;**

**if((i>j++) && (i++ == j)) i+=j;**

**1**

**2**

**3**

**答:**

**i = 5**

**26.**

**unsigned short array[]={1,2,3,4,5,6,7};**

**int i = 3;**

**\*(array + i) = ?**

**1**

**2**

**3**

**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;}**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**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**

**1**

**2**

**3**

**4**

**5**

**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**

**1**

**2**

**3**

**输出下面程序结果。**

**#include**

**class A**

**{**

**public:**

**virtual void print(void)**

**{**

**cout<<"A::print()"<}**

**};**

**class B:public A**

**{**

**public:**

**virtual void print(void)**

**{**

**cout<<"B::print()"<};**

**};**

**class C:public B**

**{**

**public:**

**virtual void print(void)**

**{**

**cout<<"C::print()"<}**

**};**

**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()**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**32**

**33**

**34**

**35**

**36**

**37**

**38**

**39**

**40**

**41**

**42**

**43**

**44**

**45**

**46**

**47**

**48**

**49**

**50**

**51**

**52**

**53**

**54**

**55**

**56**

**57**

**58**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**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( isnull([Order].Revenue,0) )**

**from customer full join [order]**

**on( [order].customerid=customer.id )**

**group by customer.id**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**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);**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**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);**

**}**

**1**

**2**

**3**

**4**

**35.程序改错**

**class mml**

**{**

**private:**

**static unsigned int x;**

**public:**

**mml(){ x++; }**

**mml(static unsigned int &) {x++;}**

**~mml{x--;}**

**pulic:**

**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;**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**8**

**9**

**10**

**11**

**12**

**13**

**14**

**15**

**16**

**17**

**18**

**19**

**20**

**21**

**22**

**23**

**24**

**25**

**26**

**27**

**28**

**29**

**30**

**31**

**基类的析构函数应该为虚函数**

**virtual ~mml{x–;}**

**36.101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 );**

**}**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

**39.写出程序结果:**

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

**{**

**printf("%d\n", sizeof(str));**

**}**

**1**

**2**

**3**

**4**

**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。**

**本文主要包括二个部分，第一部分重点介绍在VC中，怎么样采用sizeof来求结构的大小，以及容易出现的问题，并给出解决问题的方法，第二部分总结出VC中sizeof的主要用法。**

**1、 sizeof应用在结构上的情况**

**请看下面的结构：**

**struct MyStruct**

**{**

**double dda1;**

**char dda;**

**int type**

**};**

**对结构MyStruct采用sizeof会出现什么结果呢？sizeof(MyStruct)为多少呢？也许你会这样求：**

**sizeof(MyStruct)=sizeof(double)+sizeof(char)+sizeof(int)=13**

**但是当在VC中测试上面结构的大小时，你会发现sizeof(MyStruct)为16。你知道为什么在VC中会得出这样一个结果吗？**

**其实，这是VC对变量存储的一个特殊处理。为了提高CPU的存储速度，VC对一些变量的起始地址做了"对齐"处理。在默认情况下，VC规定各成员变量存放的起始地址相对于结构的起始地址的偏移量必须为该变量的类型所占用的字节数的倍数。下面列出常用类型的对齐方式(vc6.0,32位系统)。**

**类型**

**对齐方式（变量存放的起始地址相对于结构的起始地址的偏移量）**

**Char**

**偏移量必须为sizeof(char)即1的倍数**

**int**

**偏移量必须为sizeof(int)即4的倍数**

**float**

**偏移量必须为sizeof(float)即4的倍数**

**double**

**偏移量必须为sizeof(double)即8的倍数**

**Short**

**偏移量必须为sizeof(short)即2的倍数**

**各成员变量在存放的时候根据在结构中出现的顺序依次申请空间，同时按照上面的对齐方式调整位置，空缺的字节VC会自动填充。同时VC为了确保结构的大小为结构的字节边界数（即该结构中占用最大空间的类型所占用的字节数）的倍?**