1、int x[6][4],(*p)[4]: p=x;　则*(p+2)指向哪里?

- x[0][1]
- x[0][2]
- x[1][0]
- x[2][0]

解释：x 为二维数组，p 是一个数组指针，将 p 指向长度为 4 的 int 数组，那么 p 指向的元素是 x 的第一行元素的首个,p+2 指的就是第三行的首个元素，所以 p[2]所指即为 x[2][0]

2、下面有关 C++中为什么用模板类的原因，描述错误的是？

- 可用来创建动态增长和减小的数据结构
- 它是类型无关的，因此具有很高的可复用性
- 它运行时检查数据类型，保证了类型安全
- 它是平台无关的，可移植性

解释：(1)可用来创建动态增长和减小的数据结构 （2）它是类型无关的，因此具有很高的可复用性。 （3）它在编译时而不是运行时检查数据类型，保证了类型安全 （4）它是平台无关的，可移植性 （5）可用于基本数据类型

3、以下程序的输出是

```
1  class  Base {
2          public:
3          Base(int  j): i(j)   {}
4          virtual  ~Base() {}
5          void  func1() {
6                 i *= 10;
```

```
7                 func2();
8           }
9           int  getValue() {
10                func2();    i;
11          }
12          protected:
13          virtual  void  func2() {
14                i++;
15          }
16          protected:
17          inti;
18    };
19    class  Child:  public  Base {
20          public:
21          Child(intj): Base(j) {}
22          void  func1() {
23                i *= 100;
24                func2();
25          }
2           protected:
            void  func2() {
                  i += 2;
            }
      };
      int  main() {
            Base * pb =  new  Child(1);
            pb->func1();
            cout << pb->getValue() << endl;
          deletepb;
```

```
}
```

- 11
- 101
- 12
- 102

解释：

Base * pb = new Child(1)， 首先创建子类对象，初始化为 1；

func1()不是虚函数，所以 pb->func1()执行的是基类的 func1 函数，i= 10，然后调用 func2()函数；这里的 func2 是虚函数，要往下派生类寻找，找到后执行派生类中的 func2(),此时，i = 12；最后执行 pb->getValue(),结果为 12，故选 C

4、下列代码的输出为：

```
1   class  CParent
2   {
3           public: virtual  void  Intro()
4           {
5                   printf( "I'm a Parent, "); Hobby();
6           }
7           virtual  void  Hobby()
8           {
9                   printf( "I like football!");
10          }
11  };
12  class  CChild :  public  CParent {
13          public: virtual  void  Intro()
14          {
15                  printf( "I'm a Child, "); Hobby();
```

```
16          }
17          virtual  void  Hobby()
18          {
19                printf(  "I like basketball!\n");
20          }
2   };
    int  main(  void)
    {
          CChild *pChild =  new  CChild();
          CParent *pParent = (CParent *) pChild;
          pParent->Intro();
          return(0);
    }
```

- I'm a Parent， I like football!
- I'm a Parent， I like basketball!
- I'm a Child， I like basketball!
- I'm a Child， I like football!

解释：(CParent *)这个要不要都没关系的。。。输出：I'm a Child, I like basketball!

这题动态联编，Intro()和 Hobby()都是虚函数且通过指针调用，基类指针会向派生类中寻找，找到后执行派生类的函数，所以输出的结果是 I'm a Child, I like basketball!

5、在普通成员函数中调用虚函数

```
1. class BaseA
2. {
3. public:
4.     void disp()
5.     {
```

```
        check();
    };
    virtual void check()
    {
        cout << "A :: check()"<<endl;
    }
};

class BaseB: public BaseA
{
public:
    void check()
    {
        cout << "B :: check()"<<endl;
    }
};

int main(int argc, char *argv[])
{
    BaseA *A = new BaseA;
    A->disp();

    A = new BaseB;
    A->disp();

    BaseA first;
    first.disp();

    BaseB second;
    second.disp();

    system("pause");
}
```

输出结果：

```
A :: check()
B :: check()
A :: check()
B :: check()
```

6、下列一段 C++代码的输出是?

```
1  #include "stdio.h"
```

```
2   class  Base
3     {
4     public:
5           int  Bar(char  x)
6           {
7                 return(int)(x);
8           }
9           virtual  int  Bar(int  x)
10          {
11                return(2* x);
12          }
13    };
14    class  Derived :  public  Base
15    {
16    public:
17          int  Bar(char  x)
18          {
19                return(int)(-x);
20          }
21          int  Bar(int  x)
22          {
2                 return(x /  2);
            }
      };
      int  main(void)
      {
            Derived Obj;
            Base *pObj = &Obj;
            printf("%d,", pObj->Bar((char)(100)));
```

```
        printf("%d,", pObj->Bar(100));
    }
```

- 100，-100
- 100，50
- 200，-100
- 200，50

解释：

Derived Obj;
Base *pObj = &Obj;
printf("%d,", pObj->Bar((char)(100)))
printf("%d,", pObj->Bar(100));

第一个 Bar（char）是非虚函数，因此是静态绑定，静态绑定是指指针指向声明时的对象，pObj 声明时为 Base 类，因此调用的是 Base 类的 Bar（char）

第二个 Bar（char）是虚函数，因此是动态绑定，动态绑定是指指针指向引用的对象，pObj 引用 Derived 对象，因此调用的是 Derived 类的 Bar（int）

7、下面有关类的静态成员和非静态成员，说法错误的是？

- 静态成员存在于内存，非静态成员需要实例化才会分配内存
- 非静态成员可以直接访问类中静态的成员
- 静态成员能访问非静态的成员
- 非静态成员的生存期决定于该类的生存期，而静态成员则不存在生存期的概念

解释：因为静态成员存在于内存，非静态成员需要实例化才会分配内存，所以静态成员函数不能访问非静态的成员。因为静态成员存在于内存，所以非静态成员函数可以直接访问类中静态的成员

8、在 32 位小端的机器上，如下代码输出是什么：

```
1  char  array[12] =
   {0x01,  0x02,  0x03,  0x04,  0x05,  0x06,  0x07,  0x08};
2
3  short* pshort = (short*)array;
4  int* pint = (int*)array;
5  int64 *pint64 = (int64 *)array;
   printf("0x%x , 0x%x , 0x%x , 0x%x", *pshort , *(pshort+2) ,
   *pint64 , *(pint+2));
```

- 0x201 , 0x403 , 0x807060504030201 , 0x0
- 0x201 , 0x605 , 0x807060504030201 , 0x0
- 0x201 , 0x605 , 0x4030201 , 0x8070605
- 0x102 , 0x506 , 0x102030405060708 , 0x0

解释：选 B，

小端机器的数据高位字节放在高地址，低位字节放在低地址。x86 结构为小端模式。

pshort 占用 2 个字节，在内存中的 16 进制为 0x01 0x02，对应的 16 进制数为 0x0201。

pshort + 2 指向 array 数组的下标为 4 的元素，占用 2 个字节，在内存中的 16 进制为 0x05 0x06，对应的 16 进制数为 0x0605。

pint64 的 int64 类型不确定，但根据名字可以看出占用 8 个字节，对应的 16 进制形式为 0x807060504030201。

pint + 2 占用 4 个字节，指向的 array 数组的下标为 8 的元素，8-11 个元素没有指定数组的初始化值，默认为 0，因此*(pint + 2)对应的 16 进制为 0。

9、函数模板与类模板的区别

（1）函数模板允许隐式调用和显式调用，而类模板只能显示调用。

（2）函数模板的实例化是由编译器在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

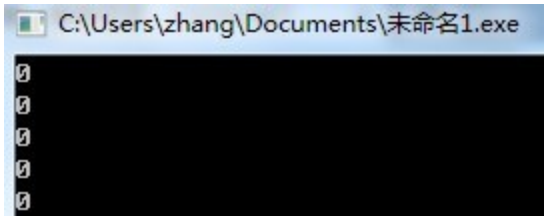
（3）注意：模板类的函数声明和实现必须都在头文件中完成，不能像普通类那样声明在.h 文件中实现在.cpp 文件中

10、数组的引用

在《C++ Primer 第四版》的第七章中，讲到了通过引用传递数组，和其他类型一样，数组形参可声明为数组的引用。如果形参是数组的引用，编译器不会将数组实参转化为指针，而是传递数组的引用本身。在这种情况下，数组大小成为形参与实参类型的一部分，编译器检查数组实参的大小与形参的大小是否匹配。


```cpp
#include <iostream>
using namespace std;
void output(int (&a)[5])
{
    for(int i = 0; i < 5; i++)
        cout<<a[i]<<endl;
}
int main()
{
    int a[5]={0};
    output(a);
    getchar();
```

```
    return 0;
}
```

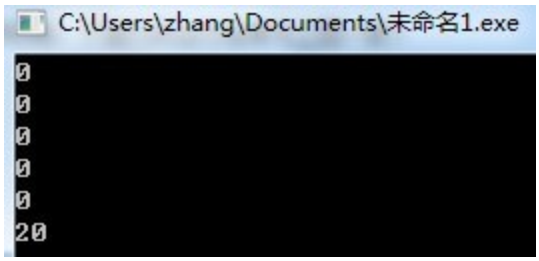
输出结果为： 

也可以在函数中直接声明一个对数组的引用：

[cpp] view plaincopyprint?

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

int main()
{
    int a[5]={0};
    int (&b)[5] = a;
    for(int i = 0; i < 5; i++)
        cout<<b[i]<<endl;
    cout<<sizeof(b);
    getchar();
    return 0;
}
```

输出结果为： 

从运行结果中可以看到 b 的大小和数组 a 的大小一样，也为 20。可见数组的引用同样可以看做是数组的一个别名。但如果将上面代码中的 int (&b)[5] = a; 改为 int &b[5] = a; 则编译无法通过，因为声明了一个引用数组。即便你将引用数组 b 中的每一个元素 b[i]看做对 a[i]的引用，然后对每一个元素进行初始化，也是错误的：int &b[5] = {a[0],a[1],a[3],a[3],a[4]};

一句话，你可以创建对数组的引用，但你不能创建一个元素都是引用的数组。因为引用创建的时候都必须初始化，引用的数组，无法给所有引用初始化。

数组的引用的应用

1、如上所说，可以当函数参数，并且可以传递维数过去。

2、维度的大小可以用一个模板的方式直接表示，这样更清楚明确

template<size_t SIZE> void fun(int (&arr)[SIZE]);

code：

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

void ArrRef(int arr[5])
{
    for (int i = 0; i < 5; ++i)
    {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

//使用模板通用化一些，对数组通用
template<size_t SIZE>
void ArrRefTemplete(int (&arr)[SIZE])
{
    memset(arr, 0, sizeof(int) * (SIZE - 3));
}

int main(int argc, char* argv[])
{
    const int MAX_INDEX(5);
    int aIndex[MAX_INDEX] = {1,3,4,5,6};

    //如果改了数组大小 MAX_INDEX 的值，两个大小对不上就会有问题
    ArrRef(aIndex);

    //如果改了数组大小 MAX_INDEX 的值，也无所谓
    ArrRefTemplete(aIndex);
```

```
30.
31.     for (int i = 0; i < MAX_INDEX; ++i)
32.     {
33.         cout << aIndex[i] << " ";
34.     }
35.     cout << "\n";
36.
37.     system("pause");
38.     return 0;
39. }
```

out：

```
1 3 4 5 6
0 0 4 5 6
请按任意键继续. . .
```

end

11、下面关于数组的描述错误的是（ ）

- 在 C++语言中数组的名字就是指向该数组第一个元素的指针
- 长度为 n 的数组，下标的范围是 0-n-1
- 数组的大小必须在编译时确定
- 数组只能通过值参数和引用参数两种方式传递给函数

解释：

将物理上前后相邻、类型相同的一组变量作为一个整体，称为数组类型的变量，简称(一维)数组。

数组名代表第一个数组元素的首地址，是一个指针常量，等价于一个指针字面值常量；

长度为 n 的数组，下标范围 0 — n-1；数组的大小必须在编译时确定。

数组可以通过值参数、指针参数和引用参数三种方式传递给函数

12、C++是不是类型安全的？

- 是
- 不是

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

13、有以下程序

```
#include <stdio. h>
main(){
        int  x =35,B;
        char  z =’B’;
        B =((x)&&(z <’b’));
        printf("%d\n",B);
}
```

程序运行后的输出结果是

- 1
- 0
- 35
- 66

解释：本题重点考查逻辑运算符和关系运算符的相关知识,已知变量 x 为整型变量,并赋值为 35,变量 z 为字符型变量,并赋值为'B'。语句 B =((x)&&(z <'b'));中,(x)的值为 1,'B'的 asscii 码小于'b'的 asscii 码,所以(z<'b')的值也为 1,1&&1 结果为 1。因此 A 选项正确。

14、

```
CONTAINER::iterator iter , tempIt;
for(iter = cont.begin() ; iter != cont.end() ; )
{
        tempIt = iter;
        ++iter;
        cont.erase(tempIt);

}
```

假设 cont 是一个 CONTAINER 的示例，里面包含数个元素，那么当 CONTAINER 为：

1、vector 2、list 3、map 4、deque 会导致上面的代码片段崩溃的 CONTAINER 类型是？

- 1，4
- 2，3
- 1，3
- 2，4

解释：第 1 个、第 4 个都是线性的类型存储，所以会存在崩溃

15、以下哪些做法是不正确或者应该极力避免的：【多选】（ ）

- **构造函数声明为虚函数**
- 派生关系中的基类析构函数声明为虚函数
- **构造函数中调用虚函数**

- **析构函数中调用虚函数**

答案：A C D

解释：所谓虚函数就是多态情况下只执行一个,而从继承的概念来讲,总是要先构造父类对象,然后才能是子类对象,如果构造函数设为虚函数,那么当你在构造父类的构造函数时就不得不显示的调用构造,还有一个原因就是为了防错,试想如果你在子类中一不小心重写了个跟父类构造函数一样的函数,那么你的父类的构造函数将被覆盖,也即不能完成父类的构造.就会出错.
在构造函数不要调用虚函数。**在基类构造的时候，虚函数是非虚，不会走到派生类中，既是采用的静态绑定。**显然的是：当我们构造一个子类的对象时，先调用基类的构造函数，构造子类中基类部分，子类还没有构造，还没有初始化，如果在基类的构造中调用虚函数，如果可以的话就是调用一个还没有被初始化的对象，那是很危险的，所以 C++中是不可以在构造父类对象部分的时候调用子类的虚函数实现。但是不是说你不可以那么写程序，你这么写，编译器也不会报错。**只是你如果这么写的话编译器不会给你调用子类的实现，而是还是调用基类的实现。**
在析构函数中也不要调用虚函数。在析构的时候会首先调用子类的析构函数，析构掉对象中的子类部分，然后在调用基类的析构函数析构基类部分，如果在基类的析构函数里面调用虚函数，会导致其调用已经析构了的子类对象里面的函数，这是非常危险的。

16、在 c++中，

```
1  constinti =  0;
2  int*j = (int*) &i;
3  *j =  1;
4  printf("%d,%d", i, *j)
```

输出是多少？

- 0，1
- 1，1
- 1，0
- 0，0

解释：选 A，const 修饰的常量值具有不可变性,c++编译器通常会对该变量做优化处理，在编译时变量 i 的值为已知的，编译器直接将 printf 输出的变量 i 替换为 0。尽管如此，编译器仍然会为变量 i 分配存储空间，通过修改内存的 hack 方式将变量 i 在内存中的值修改后并不影响 printf 的输出。如果将 i 更改为 volatile const int 类型的，编译器就不会对变量 i 做优化，printf 输出的结果就为 1。

const 在 C 语言中是表示**道义上保证变量的值不会被修改**，并不能实际阻止修改，通过指针可以修改常变量的值，但是会出现一些不可知的结果。几种情况不同，我们一个一个来看。

1、直接赋值

```
constinta=3;a=5;// const.c:6:2: error: assignment of read-only variable ‘a’
```

这种情况不用多说，编译错。

2、使用指针赋值，@孙健波提到的方法，在 gcc 中的 warning，g++中出现 error，是因为代码写得不对，由非 const 的变成 const 不用显式的转换，const 变为非 const 需要显式转换，这种情况应当使用显式的类型转换。

```
constinta=3;int*b=(int*)&a;printf("a = %d, *b
= %d\n",a,*b);*b=5;printf("a = %d, *b = %d\n",a,*b);
```

运行结果（注：使用 msvc 编译的结果一致）：

```
$ gcc const.c
$ ./a.out
a = 3, *b = 3
a = 5, *b = 5

$ g++ const.cpp
$ ./a.out
a = 3, *b = 3
a = 3, *b = 5
```

这里使用 g++编译时，a 的值之所以没有改变，是因为编译时 a 是常量，然后被编译器编译为立即数了。因此对使用 b 指针修改不会更改 a 的值。

值得注意的是，如果 a 被定义为全局常变量，使用指针修改会引发 segment fault。

在函数的原型中，我们也常用 const 修饰指针，表示函数的实现者在道义上不会去修改这个指针所指向的空间。例如我们熟知的 strcpy 函数，原型如下：

```
char*strcpy(char*dst,constchar*src);
```

传入的参数 src 类型是 const char*，表示函数内部实现不会修改 src 所指向的空间。之所以说是道义上，是因为在内部通过上述指针强制类型转换的方式可以修改该空间的值。另外，如果我们声明了我们不会修改传入指针的所指向的空间，那么我们也不应当去修改这块空间，因为这个传入的指针可能会是一个不可写的内存，然后出现段错误。

const 只是对变量名进行修饰，也就是说你不可以通过这个变量名修改它所在的内存

而并不是说这块内存不可修改，如果你可以通过其他形式访问到这块内存，同样可以进行修改

```
const int a = 1;
int* p = (int*)&a; // 通过强制类型转换得到 a 所在的内存地址
*p = 2; // 通过指针指针直接写入新数据
printf("a = %d, *p = %d",a,*p);
```

17、以下 prim 函数的功能是分解质因数。括号内的内容应该为？

```
1  voidprim(intm,  intn)
2    {
3            if(m > n)
```

```
4           {
5                   while(              ) n++;
6                   (       );
7                   prim(m, n);
8                   cout << n << endl;
9           }
10      }
```

- m/n m/=n
- m/n m%=n
- m%n m%=n
- **m%n m/=n**

解释：n 从 2 开始第一处为 m%n，代表取余。当余数是 0 的时候表示除尽，跳出 while 循环，即找出一个质因数。此时一个质因数即为 n

然后 m/=n 即让 m 除去这个质因数，然后再进入求新 m 质因数的递归。

举例：m=6，n=2

m>n;

m%n=0,跳出 while，n 没有加 1。此时 m=6，n=2

m/=n，此时 m=3，n=2 （2 为一个质因数）

递归 prim(m, n)，即 prim(3, 2);

m>n;

m%n=1,n++,此时 m=3，n=3，继续 while 循环

m%n=0,跳出 while 循环，此时 m=3，n=3 （3 为另一个质因数）

m/=n,此时 m=1，n=3

递归 prim(m, n)，即 prim(1, 3);

不满足条件（m > n），返回上层

输出质因数 n=3

输出质因数 n = 2

18、使用 printf 函数打印一个 double 类型的数据，要求：输出为 10 进制，输出左对齐 30 个字符，4 位精度。以下哪个选项是正确的？

- %-30.4e
- %4.30e
- %-30.4f
- %-4.30f

解释：

-： 左对齐

30： 最小字段宽度

.4： 精确度保留小数 4 位

f： double 精度浮点数

e： 科学计数法

19、选择填空：

```
1  #include
2  voidtest(void*data) {
3          unsigned  int  value =  （此处应填入）
4          printf("%u", value);
5  }
6  usingnamespacestd;
7  intmain() {
8          unsigned  intvalue = 10;
9          test(&value);
```

```
10          return0;
11  }
```

- *data
- (unsigned int)(*data)
- (unsigned*)data
- *((unsigned int *)data)

注意，参数类型是 void，所以先要进行指针转换：（unsigned int *)然后再取值。

以下代码的输出结果是？

```
char *p="abc";
char *q="abc123";
while(*p=*q)
print("%c %c",*p,*q);
```

- aabbcc
- aabbcc123
- abcabc123
- 代码段错误

解释：答案选 D，因为两个指针都指向的字符串常量，不能被重新赋值，*p=*q 是错误的。

20、下面有关 malloc 和 new，说法错误的是？

- new 建立的是一个对象， malloc 分配的是一块内存.
- new 初始化对象，调用对象的构造函数，对应的 delete 调用相应的析构函数，malloc 仅仅分配内存，free 仅仅回收内存
- new 和 malloc 都是保留字，不需要头文件支持
- new 和 malloc 都可用于申请动态内存，new 是一个操作符，malloc 是是一个函数

解释：new/delete 都是要分两步操作的：new 分配内存，并且调用对象的构造函数初始化一个对象；delete 调用相应的析构函数，然后释放内存 malloc/free 只是分配内存/回收内存， 所以 A、B 对；malloc 需要头文件"stdlib.h"或者"malloc.h" C 错；new/delete 都是内建的操作符，而 malloc 是一个函数，其函数原型是：

```
1  void*malloc(unsigned  intnum_bytes);
```

21、请说出 const 与#define 相比，有何优点 ？

- 宏常量有数据类型，而 const 常量没有数据类型
- 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
- 编译器可以对 const 进行类型安全检查。而对#define 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。

解释：#define 和 const 相比有如下劣势：

1.const 定义常量是有数据类型的，而#define 宏定义常量却没有

2.const 常量有数据类型，而宏常量没有数据类型。编译器可以对 const 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，并且在字符替换中可能会产生意料不到的错误

3.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。因此 A 是错误的，BC 是正确的。

22、extern "C"{}的作用以及能解决什么问题？

- 在 C++源文件中的语句前面加上 extern "C"，表明它按照类 C 的编译和连接规约来编译和连接，而不是 C++的编译的连接规约
- 主要是解决在 C++代码中调用 C 代码
- 主要是解决在 C 代码中调用 C++代码
- 上述描述都不正确

解释：AC，extern "C", 按照类 C 的编译和连接规约来编译和连接，而不是 C++的编译的连接规约,既然按照 C 的规则来，那么源代码肯定是 C，即在 C 代码中调用 C++。

23、正数原码反码补码都一样

1、bit 位：位是计算机中存储数据的最小单位，指二进制数中的一个位数，其值为“0”或“1”。
2、byte 字节：字节是计算机存储容量的基本单位，一个字节由 8 位二进制数组成。在计算机内部，一个字节可以表示一个数据，也可以表示一个英文字母，两个字节可以表示一个汉字。1B＝8bit

24、C++类体系中，不能被派生类继承的有?

- 构造函数
- 静态成员函数
- 非静态成员函数
- 赋值操作函数

解释：答案：A

思路：构造函数是不能被继承的,但是可以被调用,如果父类重新定义了构造函数,也就是没有了默认的构造函数,子类创建自己的构造函数的时候必须显式的调用父类的构造函数。而其余的三个在我们平常的使用中都可以被继承使用。

答案选 A。编译器总是根据类型来调用类成员函数。但是一个派生类的指针可以安全地转化为一个基类的指针。这样删除一个基类的指针的时候，C++不管这个指针指向一个基类对象还是一个派生类的对象，调用的都是基类的析构函数而不是派生类的。如果你依赖于派生类的析构函数的代码来释放资源，而没有重载析构函数，那么会有资源泄漏。所以建议的方式是将析构函数声明为虚函数。

也就是 delete a 的时候，也会执行派生类的析构函数。一个函数一旦声明为虚函数，那么不管你是否加上 virtual 修饰符，它在所有派生类中都成为虚函数。但是由于理解明确起见，建议的方式还是加上 virtual 修饰符。

构造方法用来初始化类的对象，与父类的其它成员不同，它不能被子类继承（子类可以继承父类所有的成员变量和成员方法，但不继承父类的构造方法）。因此，在创建子类对象时，为了初始化从父类继承来的数据成员，系统需要调用其父类的构造方法。

如果没有显式的构造函数，编译器会给一个默认的构造函数，并且该默认的构造函数仅仅在没有显式地声明构造函数情况下创建。

构造原则如下：

1. 如果子类没有定义构造方法，则调用父类的无参数的构造方法。

2. 如果子类定义了构造方法，不论是无参数还是带参数，在创建子类的对象的时候,首先执行父类无参数的构造方法，然后执行自己的构造方法。

3. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数，则会调用父类的默认无参构造函数。

4. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数且父类自己提供了无参构造函数，则会调用父类自己的无参构造函数。

5. 在创建子类对象时候，如果子类的构造函数没有显示调用父类的构造函数且父类只定义了自己的有参构造函数，则会出错（如果父类只有有参数的构造方法，则子类必须显示调用此带参构造方法）。

6. 如果子类调用父类带参数的构造方法，需要用初始化父类成员对象的方式，比如：

```
1   #include <iostream.h>
2   class  animal
3   {
4   public:
5           animal(int  height,  int  weight)
6           {
7                   cout<<"animal construct"<<endl;
8           }
9   };
10  class  fish:public  animal
11  {
12  public:
13          inta;
14          fish():animal(400,  300), a(1)
15          {
16                  cout<<"fish construct"<<endl;
17          }
18  };
19  void  main()
20  {
21          fish fh;
22  }
```

25、下面这段代码运行时会出现什么问题？

```
class  A
{
public:
        void  f()
        {
              printf("A\n");
        }
};

class  B:  public  A
{
public:
        virtual  void  f()
        {
              printf("B\n");
        }
};

int  main()
{
        A *a =  new  B;
        a->f();
        delete a;
        return0;
}
```

- 没有问题，输出 B
- **不符合预期的输出 A**
- 程序不正确
- 以上答案都不正确

```
class A
{
public:
	//void f()
	//{
	//	printf("A\n");
	//}
};

class B: public A
{
public:
	void f()
	{
		printf("B\n");
	}
};

int main()
{
	A *a = new B;
	a->f();
	delete a;
```

```
        return 0;
}
```

这样会有编译错误，f 不是类 A 的成员

26、如果友元函数重载一个运算符时，其参数表中没有任何参数则说明该运算符是：

- 一元运算符
- 二元运算符
- 选项 A) 和选项 B) 都可能
- 重载错误

解释：C++中用友元函数重载运算符至少有一个参数，重载一目运算符要有一个参数，重载二目运算符要有两个参数。如果重载运算符没有参数这个是正确的，例如前++，不需要参数，关键是 friend 表示友元函数不属于本类的函数，所以一定要有参数

operator++(int) 匹配 i++

operator++() 匹配++i

27、重复多次 fclose 一个打开过一次的 FILE *fp 指针会有什么结果？

- **导致文件描述符结构中指针指向的内存被重复释放，进而导致一些不可预期的异常**
- 不会出现异常，释放一个已经释放的指针，系统会自动忽略
- 运行异常

- 以上答案都不正确

解释：重复多次 fclose 一个打开过一次的 FILE *fp 指针会有什么结果，并请解释。(等价于 free(p),同一个指针不能释放内存两次)

【参考答案】考察点：导致文件描述符结构中指针指向的内存被重复释放，进而导致一些不可预期的异常

28、

```
1  time_t t;
```

哪个选项可以将 t 初始化为当前程序的运行时间？

- **t = clock();**
- time(&t);
- time(&t);
- t = localtime();
- None of the above

29、下列程序的输出结果为：

```
1  #include<iostream.h>
2  void  main()
3  {
4              char*  a[  ]  =  {  "hello",  "the",  "world"};
5              char**  pa  =  a;
6              pa++;
7              cout<<*pa<<endl;
8  }
```

- theworld

- **the**
- hello
- hellotheword

解释：分析：a 是指针的数组

char** p = a; //char** p = &a[0]

p++;//p 是指针自增+4，而 a 中元素是指针，每个正好四个字节，因此 p++后恰好 p= &a[1]

*p=a[1];输出"the"，输出结果为 B

30、下面模板声明中，哪些是非法的（ ）

- template<class Type>class C1;
- **template<class T, U, class V>class C2;**
- template<class C1, ypename C2>class C3{};
- **template<typename myT, class myT>class C4{};**

解释：答案 B D

解释：B 选项的 U 参数没有指定类型， D 选项的 2 个形参名同名。

函数模板的格式：

template <class 形参名，class 形参名，.....> 返回类型 函数名(参数列表)

{

函数体

}

类模板的格式为：

template<class 形参名 ，class 形参名，...> class 类名

{ ... };

31、unsigned int a= 0x1234; unsigned char b=*(unsigned char *)&a; 在 32 位大端模式处理器上变量 b 等于()?

- **0x00**
- 0x12
- 0x34
- 0x1234

解释：/ 答案：0

// unsigned int a= 0x1234; 其中 int 是 4 字节, 大端存储 ，补齐 16 进制表示为: 0x00 00 12 34

// unsigned char b=*(unsigned char *)&a; 由于大端存储, 所以上述 int a 变量的最低地址存储的是

// 十六进制表示中最左边的 1 字节, 为 0x00.

32、若有以下程序

```
1  struct st{ int n; struct st * nest; };
2  struct st a[3]={ 5,&a[1],7,&[2],9, '\0' },*p;
3  p=&a[0];
```

则以下选项中值为 6 的表达式是

- p->n
- (*p).n
- p->n++
- **++(p->n)**

33、以下代码的输出是（ ）

```
1  inta[5]={1,2,3,4,5};
```

```
2  int*ptr=(int*)(&a+1);
3  printf("%d,%d",*(a+1),*(ptr-1));
```

- 1，2
- 2，5
- 2，1
- 1，5

解释：数组名的值是一个指针常量，也就是数组第一个元素的地址。*(a+1)等同于 a[1]，*(a+1)=2。&a+1 指向数组最后一个元素的下一个位置，故*(ptr-1)指向数组的最后一个元素。选 B。a 代表的是 int * 每次步长为一个 int &a 代表的是 int[]* 每次步长为所指向的数组的大小，故 ptr 指向的是数组 a 最后一个元素的下一个元素，所以 ptr-1 指向的是数组 a 的最后一个元素 a+1 指向的是数组 a 的第二个元素，所以正确答案是 B

34、指出下面程序哪里可能有问题？

```
1  class  CBuffer
2  {
3          char* m_pBuffer;
4          int  m_size;
5          public:
6                  CBuffer()
7                  {
8                          m_pBuffer=NULL;
9                  }
10                 ~CBuffer()
11                 {
```

```
12                          Free();
13                  }
14                  void  Allocte(intsize)  (1)  {
15                          m_size=size;
16                          m_pBuffer=  new  char[size];
17          }
18          private:
19                  void  Free()
20                  {
21                          if(m_pBuffer!=NULL) (2)
22                          {
23                                  delete  m_pBuffer;
24                                  m_pBuffer=NULL;
25                          }
26                  }
27          public:
28                  void  SaveString(const  char*
29  pText)  const(3)
                    {
                            strcpy(m_pBuffer, pText); (4)
                    }
                    char* GetBuffer()  const
                    {
                            return  m_pBuffer;
                }
}
void  main (int  argc,  char* argv[])
{
        CBuffer buffer1;
        buffer1.SaveString("Microsoft");
```

```
        printf(buffer1.GetBuffer());
}
```

- 1
- 2
- 3
- 4

解释：改正后主要改正 SaveString 函数

将

```
1
2  void  SaveString(constchar* pText)  const
3  {
4          strcpy(m_pBuffer,  pText);
5  }
6
```

改为

```
1
2  void  SaveString(constchar* pText) (1)
3  {
4          Allocte(strlen(pText)+1); (2)
5          strcpy(m_pBuffer,pText);
6  }
7
```

原因：

（1）const 成员函数表示不会修改数据成员，而 SaveString 做不到，去掉 const 声明

（2）m_pBuffer 指向 NULL，必须用 Allocte 分配空间才能赋值。

（3）另外需要将 Allocte 成员函数声明为私有成员函数更符合实际

delete m_pBuffer; 最好改成 delete[] m_pBuffer，不会造成内存泄露，但是不建议使用; if the type is int,char,float, both delete[]p and delete p are ok,but if the type is class object, the answer will be A

35、turbo c 环境下，下面程序运行的结果是()

```
1   int  main()
2   {
3           printf("\n");
4           int  a[5] = {1,  2,  3,  4,  5};
5           int*p, **k;
6           p = a;
7           k = &p;
8           printf("%d", *(p++));
9           printf("%d", **k);
10          return0;
11  }
```

- 11
- 21
- 22
- 12

解释：D，主要是*(p++)的运算顺序。先*p，再自增。

36、若以下选项中的变量 a,b,y 均以正确定义并赋值,则语法正确的 switch 语句是?

- switch(a*a+b*b) { default：break; case3 ： y=a+b; break; case2 ： y=a-b; break; }
- switch(a+b) { case1 ： case3 ： y=a+b;break; case0 ： case4 ： y=a-b;}
- switch(a+9) { case a ： y=a-b; case b ： y=a+b; }
- switch a*b { case 10 ： y=a+b; default ： y=a-b;}

解释：B 选项中,case1 和 case2 有错误,;C 选项中 case a 和 case b 不正确,case 后面应该跟常量表达式;D 选项中,switch a*b,有误。因此 A 选项正确。

37、若 fp 已定义为指向某文件的指针,且没有读到该文件的末尾,C 语言函数 feof(fp)的函数返回值是？

- EOF
- 非 0
- -1
- 0

解释：本题考查文件的定位,feof 函数的用法是从输入流读取数据,如果到达稳健末尾(遇文件结束符),eof 函数值为非零值,否则为 0,所以选项 D 正确。

38、**当类中含有 const、reference 成员变量以及基类的构造函数都需要初始化列表。**

无论是在构造函数初始化列表中初始化成员，还是在构造函数体中对它们赋值，最终结果都是相同的。不同之处在于，使用构造函数初始化列表初始化数据成员，没有定义初始化列表的构造函数在构造函数体中对数据成员赋值。

对于 const 和 reference 类型成员变量，它们只能够被初始化而不能做赋值操作，因此只能用初始化列表。

还有一种情况就是， **类的构造函数需要调用其基类的构造函数的时候。**请看下面的代

码：

```
1   #include <iostream>
2   using namespace std;
3
4   class A                 //A 是父类
5   {
6   private:
7       int a;              //private 成员
8   public:
9       A() {}
10      A(int x):a(x) {}        //带参数的构造函数对 a 初始化
11      void printA()           //打印 a 的值
12      {
13          cout << "a = " << a << endl;
14      }
15  };
16
17  class B : public A //B 是子类
18  {
19  private:
20      int b;
21  public:
22      B(int x, int y) : A(x)      //需要初始化 b 以及父类的 a
23      {
24          //a = x;          //a 为 private，无法在子类被访问，编译错误
25          //A(x);           //调用方式错误，编译错误
26          b = y;
27      }
28      void printB() //打印 b 的值
29      {
30          cout << "b = " << b << endl;
31      }
32  };
33
```

```
34    int main()
35    {
36          B b(2,3);
37
38          b.printA();          //调用子类的 printA()
39          b.printB();          //调用自己的 printB()
40
41          return 0;
42    }
```

从上面的程序可以看到，如果在子类的构造函数中需要初始化父类的 private 成员，直接对其赋值是不行的（代码 24 行），只有调用父类的构造函数才能完成对它的初始化。**但在函数体内调用父类的构造函数也是不合法的**（代码 25 行），只有采取 22 行中的初始化列表调用子类构造函数的方式。程序的执行结果如下：

```
1     a = 2
2     b = 3
```

39、下面程序的输出是（ ）

```
1   class  A
2   {
3   public:
4           void  foo()
5           {
6                 printf("1");
7           }
8           virtual  void  fun()
9           {
10                printf("2");
11          }
12  };
```

```
13  class  B:  public  A
14  {
15  public:
16          void  foo()
17          {
18                printf("3");
19          }
20          void  fun()
21          {
22                printf("4");
23          }
24  };
25  int  main(void)
26  {
2           A a;
            B b;
            A *P = &a;
            p->foo();
            p->fun();
            p = &b;
            p->foo();
            p->fun();
            A *ptr = (A *)&b;
            ptr->foo():
            ptr->fun();
            return0;
    }
```

- 121434
- 121414
- 121232
- 123434

解释：B

1，首先声明为 A 类型的指针指向实际类型为 A 的对象，调用的肯定是 A 的方法，输出 1 2,

2，然后声明为 A 类型的指针指向实际类型为 B 的对象，则非虚函数调用 A 的方法，输出 1，虚函数调用实际类型 B 的方法，输出 4

3，声明类型为 A 的指针指向实际类型为 B 的对象，进行一个强制类型转换，其实这种父类指针指向子类会自动进行类型转换，所以是否强制类型转换都不影响结构，原理同上一步，结果输出 1 4，所以最终输出为 121414

40、若有以下定义和语句：

```
1  intu=010, v= 0x10, w=10;
2  printf(“%d, %d,%d/n”,u,v,w);
```

则输出结果是：

- 8,16,10
- 10,10,10
- 8,8,10
- 8,10,10

解释：各种进制之间的转换，简单题，0x 表示十六进制，0 表示八进制。

41、

```
enum  string{
        x1,
```

```
        x2,
        x3=10,
        x4,
        x5,
} x;
```

问 x 等于什么？

- 5
- 12
- 0
- 随机值

解释：如果是函数外定义那么是 0，如果是函数内定义，那么是随机值，因为没有初始化

42、下面程序的执行结果：

```
1   class A{
2           public:
3                   long a;
4   };
5   class B : public A {
6           public:
7                   long b;
8   };
9   void seta(A* data, int idx) {
10          data[idx].a = 2;
11  }
12  int main(int argc, char *argv[]) {
13          B data[4];
14          for(int i=0; i<4; ++i){
```

```
15                  data[i].a = 1;
16                  data[i].b = 1;
17                  seta(data, i);
18          }
            for(int i=0; i<4; ++i){
                      std::cout << data[i].a << data[i].b;
            }
            return0;
}
```

- 11111111
- 12121212
- 11112222
- 21212121
- 22221111

解释：这道题应该注意 指针类型加减 时步长的问题。

A 大小为 4

B 大小为 8

那么：

```
void seta(A* data, int idx) {
   data[idx].a = 2;
}
```

由于传入的实参为 B 类型，大小为 8，而形参为 A 类型，大小为 4，data[idx] 取 data + idx 处的元素，这时指针 data 加 1 的长度不是一个 B 长度，而是一个 A 长度，或者说是 1/2 个 B 长度。这时该函数中 data[0~3] 指向的是原

data[0].a,data[0].b,data[1].a,data[1].b, 由于隐式类型转换的缘

故， data[0].a, data[0].b,data[1].a,data[1].b 处的值全部由于 data[idx].a = 2; 操作变为

2。这道题如果改为 void seta(B* data, int idx)，那么形参中 data 指针加 1 步长为 8，结果就是 21212121。但是由于步长为 4，所以结果就是 22221111。

43、设已经有 A,B,C,D4 个类的定义，程序中 A,B,C,D 析构函数调用顺序为？

```
1  C c;
2  void  main()
3  {
4          A*pa=new  A();
5          B b;
6          static  D d;
7          delete pa;
8  }
```

- A B C D
- A B D C
- A C D B
- A C B D

解释：选 B，因为类 A、B 都是局部变量，类 D 是静态局部变量，所以先释放类 A、B，当主函数执行往再释放 D，最后释放全局变量类 C，所以顺序为 A B D C。这道题主要考察的知识点是 ：全局变量，静态局部变量，局部变量空间的堆分配和栈分配，其中全局变量和静态局部变量时从 静态存储区中划分的空间，二者的区别在于作用域的不同，全局变量作用域大于静态局部变量（只用于声明它的函数中），而之所以是先释放 D 在释放 C 的原因是， 程序中首先调用的是 C 的构造函数，然后调用的是 D 的构造函数，析构函数的调用与构造函数的调用顺序刚好相反。局部变量 A 是通过 new 从系统的堆空间中分配的，程序运行结束之后，系统是不会自动回收分配给它的空间的，需要程序员手动调用 delete 来释放。局部变量 B 对象的空间来自于系统的栈空间，在该方法执行结束就会由

系统自动通过调用析构方法将其空间释放。之所以是 先 A 后 B 是因为，B 是在函数执行到 结尾 "}" 的时候才调用析构函数， 而语句 delete a；位于函数结尾 "}" 之前。

44、以下程序的输出结果为

```
#include "stdio.h"
int func(int x, int y)
{
        return(x + y);
}
int main()
{
        int a = 1, b = 2, c = 3, d = 4, e = 5;
        printf(" % d\n", func((a + b, b + c, c + a), (d, e)));

        return0;
}
```

- 15
- 5
- 9
- 出错

解释：注意括号中逗号运算符返回值是最后的表达式的值，题中第一个参数为 c+a=4，后一个参数为 e=5，所以结果为 9

45、对以下程序，正确的输出结果是()

```
#include <stdio.h>
#define SUB(x,y) x-y
```

```
3  #define ACCESS_BEFORE(element,offset,value) *SUB(&element,
4  offset) = value
5  intmain() {
6          intarray[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
7          inti;
8          ACCESS_BEFORE(array[5], 4, 6);
9          printf("array: ");
10         for(i = 0; i < 10; ++i) {
11                 printf("%d", array[i]);
12         }
13         printf("\n");
14         return(0);
   }
```

- array: 1 6 3 4 5 6 7 8 9 10
- array: 6 2 3 4 5 6 7 8 9 10
- 程序可以正确编译连接，但是运行时会崩溃
- 程序语法错误，编译不成功

解释：这道题大家走出考场后争议非常大。咱啥也不说，直接进 mingw 跑一下 gcc：

gcc 提示的错误是“赋值号的左边操作数需要一个左值”。其原因是调用宏的那句被预处理器

替换成了：*&array[5]-4 =6;

由于减号比赋值优先级高，因此先处理减号；由于减号返回一个数而不是合法的左值，所以编译报错。

46、关于内联函数正确的是（ ）

- 类的私有成员函数不能作为内联函数
- 在所有类说明中内部定义的成员函数都是内联函数
- 类的保护成员函数不能作为内联函数
- 使用内联函数的地方会在运行阶段用内联函数体替换掉

解释：感觉只有 B 对。A 是可以的，私有成员函数可以内联，C 也可以，D 应该是在编译阶段替换，B 中当在类声明时就定义的成员函数，若已声明为虚函数，此函数仍是非内联函数，而是以多态性出现。B 应该也是错的啊

47、有如下程序段：

```
1   #include  "stdio.h"
2
3   class  A
4   {
5   public:
6           int  _a;
7           A()
8           {
9                   _a =  1;
10          }
11          voidprint()
12          {
```

```
13                printf(“d”, _a);
14          }
15  };
16  classB:  publicA
17  {
18  public:
19          int_a
20          B()
21          {
2                   _a =  2;
            }
    };
    intmain()
    {
            B b;
            b.print();
            printf(“d”, b._a);
    }
```

请问程序输出：

- 22
- 11
- 21
- 12

解释：选 D， 因为在继承的时候，允许子类存在与父类同名的成员变量，但是并不覆盖父类的成员变量，他们同时存在。 因为给孩子类中没有定义 print 函数，所以会按照就近原则去寻找父类中是否有 print 函数。恰好父类中有这个函数，于是调用父类的 print 函数 b.print（ ），而这个函数会调用父类的 a 变量。

48、下面程序段的输出结果是

```
1  char*p1 = ”123”, *p2 = ”ABC”, str[50] = “xyz”;
2  strcpy(str + 2, strcat(p1, p2));
3  printr(“s\n”, str);
```

- xyz123ABC
- z123ABC
- xy123ABC
- **出错**

解释：D

```
1  strcat(p1, p2)
```

p1 没有足够空间保存连接后的字符串，出错。

49、以下选项中非法的 C 语言字符常量是？

- ’\007\’
- ’\b\’
- ’aa’
- **’\09’**

50、已知一运行正常的程序中有这样两个语句：int p1,p2=&a; p1=b; 由此可知，变量 a 和 b 的类型分别是（）。

- **int 和 int**
- int*和 int
- int 和 int*
- int*和 int*

51、有如下程序段：

```
1  int i, n = 0;
2  float x = 1, y1 = 2.1 / 1.9, y2 = 1.9 / 2.1;
3  for( i = 1; i < 22; i++ )
```

```
4    x = x * y1;
5  while( x != 1.0 )
6  {
7    x = x * y2; n++;
8  }
9  printf( “ %d / n ”, n );
```

请问执行结果是：

- 21
- 22
- 无限循环
- 程序崩溃

解释：不可将浮点变量用“==”或“！=”与任何数字比较。

千万要留意，无论是 float 还是 double 类型的变量，都有精度限制。所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。

假设浮点变量的名字为 x，应当将

```
if (x == 0.0)          // 隐含错误的比较
```

转化为

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

其中 EPSINON 是允许的误差（即精度）。

.....

好奇怪的循环问题

```
1
#include <stdio.h>
2
Void main ()
3
{
4
      Float percent;
5
      for(percent = 0.0; percent <= 1.0; percent += 0.1)
6
          printf("%3.1f/n", percent);
7
}
8
```

上面这个循环运行 10 次

```
1  for    (percent = 0; percent < =10; percent += 1)
2  printf("%3.1f/n", percent);
```

改成整数后，就运行 11 次了

.....

这是数据精度的问题，浮点数的比较一般不用==判断，而是比较差值是否大于 1E-6 甚至更高精度。

```
1  for    (percent = 0.0; percent <= 1.0; percent += 0.1)
2          printf("%3.1f/n", percent);
```

将%3.1 改为%3.7f，或%3.8f 甚至更多位，可以明显发现数据的值不是完全的 0.0,0.1,0.2,0.3 这样加 0.1。我们认为的 1.0 在内存里是 1.000000119(我是%3.9f 时得出的)，它大于 1.0，不执行了，所以只执行 10 次了。

```
1  for    (percent = 0.0; percent <= 2.0; percent += 0.1)
2          printf("%3.9f/n", percent);    //这时可以看到 0 到 1.0 的真实数据
```

原文:http://blog.csdn.net/toonny1985/article/details/4794866

52、下面有关 c++线程安全，说法错误的是？

- 线程安全问题都是由全局变量及静态变量引起的
- 若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全
- c++标准库里面的 string 保证是线程安全的
- POSIX 线程标准要求 C 标准库中的大多数函数具备线程安全性

解释：两个难点：

线程安全问题都是由全局变量及静态变量引起的 ？

答案：参考：http://blog.csdn.net/ghevinn/article/details/37764791

这里不要去扣字眼了，我当初看了半天 全局变量及静态变量。。 我想是或者。。。囧

c++标准库里面的 string 保证是线程安全的 ??

参考：http://ar.newsmth.net/thread-b3208a82b888e6.html

53、Which of the following statements are true?

- We can create a binary tree from given inorder and preorder traversal sequences.
- We can create a binary tree from given preorder and postorder traversal sequences.
- For an almost sorted array, insertion sort can be more effective than Quicksort.
- Suppose T(n) is the runtime of resolving a problem with n elements, T(n) = Θ(1) if n = 1; T(n) = 2T(n/2) + Θ(n) if > 1; so T(n) is Θ(n log n).
- None of the above.

D 和 Merge-sort 类似

D 令 O(n) --> f(n) 设 n = 2^k

T(n) = T(2^k) = 2T(2^k-1) + f(2^k)

= 2(2(T(2^k-2) + f(2^k-1)) + f(2^k)

= 2*2T(2^k-2) + 2f(2^k-1) + f(2^k)

= ....

= 2^kT(1) + 2^(k-1)f(2) + 2^(k-2)f(2*2) + ... 2^0f(2^k)

设 g(1) = T(1) = O(1) n = 1 时

设 g(1) = a f(n) = bn a、b 为常数

T(n) = T(2^k)

= 2^k*a + 2^(k-1)*2b + 2^(k-2)*2^2b + ...2^0*2^k*b

= 2^k*a + kb*2^k

= an + kbn

= an + bnlog2n

= O(nlog2n)

53、下面有关 java hashmap 的说法错误的是？

- HashMap 的实例有两个参数影响其性能：“初始容量” 和 “加载因子”。
- HashMap 的实现不是同步的，意味着它不是线程安全的
- HashMap 通过开放地址法解决哈希冲突
- HashMap 中的 key-value 都是存储在 Entry 数组中的

54、malloc 函数进行内存分配是在什么阶段?

- 编译阶段
- 链接阶段
- 装载阶段
- 执行阶段

55、设有如下定义：

```
unsigned  longpulArray[]={6,7,8,9,10};
unsigned  long*pulPtr;
```

则下列程序段的输出结果为多少？

```
pulPtr=pulArray;
*(pulPtr+2)+=2;
printf("%d,%d\n",*pulPtr,*(pulPtr+2));
```

- 0 2
- 6 8
- 6 10
- 8 8

*(pulptr+2)+=2; 相当于 *(pulptr+2)=*(pulptr+2)+2;此时*(pulptr+2)=8,加上 2 结果是 10; 简单点来说就是 a+=2;等于 a=a+2 的变形，因此最后的输出结果是*(pulptr+2)=10

56、对静态成员的不正确描述（）

正确答案：C　　你的答案：C （正确）

静态成员不属于对象，是类的共享成员
静态数据成员要在类外定义和初始化
调用静态成员函数时要通过类或对象激活，所以静态成员函数拥有 this 指针
非静态成员函数也可以操作静态数据成员

57、运行下面这段 C 语言程序之后，输出在屏幕上的结果是：

```
Void foobar(int a,  int *b,  int **c)
{
        Int *p = &a;
        *p = 101;
        *c = b;
        b = p;
}

int main()
```

```
{
        Int a = 1;
        Int b = 2;
        Int c = 3;
        Int *p = &c;
        foobar(a, &b, &p);
        printf("a=%d, b=%d, c=%d, *p=%d\n", a, b, c, *p);
        return(0);
}
```

正确答案：A　　你的答案：A （正确）

a=1, b=2, c=3, *p=2
a=101, b=2, c=3, *p=2
a=101, b=101, c=2, *p=3
a=1, b=101, c=2, *p=3

58、下面函数的执行结果是输出

```
func(charpara[100])
{
        void*p = malloc(100);
        printf("%d, %d\n", sizeof(para), sizeof(p));
}
```

- 4，4
- 100，4
- 4，100
- 100，100

对于形参 char para[100]，其实参传递进来的是数组的地址，在 32 位系统中 sizeof（para）=4
对于指针 p，是连续 100 字节空间的首地址，sizeof 并不知道这个连续空间有多大，sizeof 得到的是这个指针变量的大小，是 4 字节

59、下面关于数组的描述错误的是（）

- 在 C++语言中数组的名字就是指向该数组第一个元素的指针
- 长度为 n 的数组，下标的范围是 0-n-1
- 数组的大小必须在编译时确定
- 数组只能通过值参数和引用参数两种方式传递给函数

解释：传参有三种方式：值参数、指针参数和引用参数

60、下面程序的功能是输出数组的全排列。请填空。

```
1   Void perm(intlist[],  intk,  intm)
2   {
3           if(      )
4           {
5                   copy(list,list+m,ostream_iterator<int>(cout," "));
6                   cout<<endl;
7                   return;
8           }
9           for(inti=k; i<=m; i++)
1           {
0
1                   swap(&list[k],&list[i]);
1                   (      );
1
1                   swap(&list[k],&list[i]);
2           }
    }
```

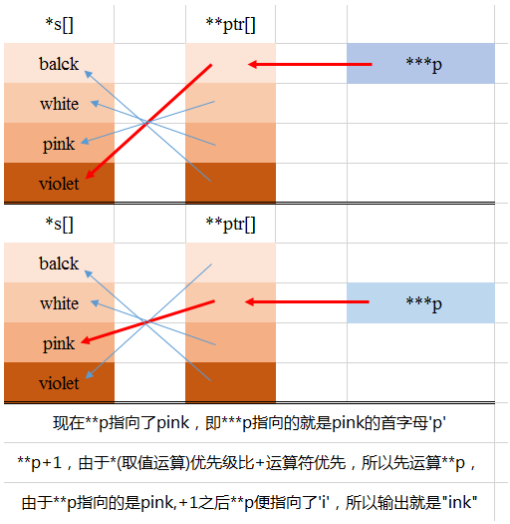
- k!=m 和 perm（list，k+1，m）
- k==m 和 perm（list，k+1，m）
- k!=m 和 perm（list，k，m）
- k==m 和 perm（list，k，m）

61、下述程序的输出是_____。

```
1   #include<stdio.h>
2
3   intmain()
```

```
4   {
5           staticchar*s[] = {"black",  "white",  "pink",  "violet"};
6           char**ptr[] = {s+3, s+2, s+1, s}, ***p;
7           p = ptr;
8           ++p;
9           printf("%s", **p+1);
1           return0;
0   }
```

- ink
- pink
- white
- hite



62、下面有关 java 和 c++的描述，错误的是？

- java 是一次编写多处运行，c++是一次编写多处编译
- c++和 java 支持多重继承
- Java 不支持操作符重载，操作符重载被认为是 c++的突出特征
- java 没有函数指针机制，c++支持函数指针

JAVA 没有指针的概念，被封装起来了，而 C++有;JAVA 不支持类的多继承，但支持接口多继承，C++支持类的多继承;C++支持操作符重载，JAVA 不支持;JAVA 的内存管理比 C++方便，而且错误处理也比较好;C++的速度比 JAVA 快。
C++更适用于有运行效率要求的情况，JAVA 适用于效率要求不高，但维护性要好的情况。

63、已有变量定义和函数调用语句，

```
1  inta=25;
2  print_value(&a);
```

则下面函数的正确输出结果是_____。

```
1  voidprint_value(int* x)
2  {
3          printf(“%x\n”,++*x);
4  }
```

- 25
- 26
- 19
- 1a

printf(“%x\n”,++*x); 先把 x 的值加上 1 再 16 进制输出。

64、栈通常采用的两种存储结构是什么？

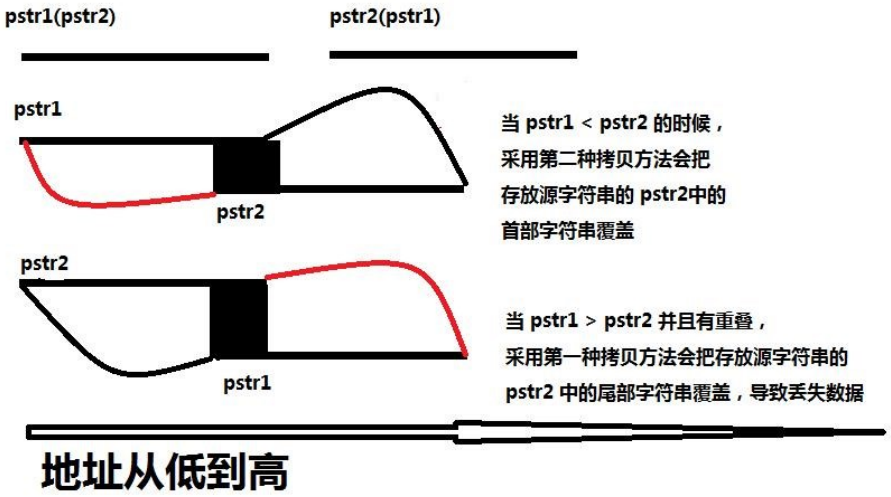
- **线性存储结构和链表存储结构**
- 散列方式和索引方式
- 链表存储结构和数组
- 线性存储结构和非线性存储结构

65、如果两段内存重叠，用 memcpy 函数可能会导致行为未定义。 而 memmove 函数能够避免这种问题，下面是一种实现方式，请补充代码。

```
1  #include <iostream>
2  using namespace std;
3  void* memmove(void* str1,const void* str2,size_t n)
4  {
5          char* pStr1= (char*) str1;
6          const char* pStr2=(const char*)str2;
7          if      ( ) {
8                  for(size_t i=0;i!=n;++i){
```

```
9                       *(pStr1++)=*(pStr2++);
1               }
0
        }
1
1       else{
1               pStr1+=n-1;
2               pStr2+=n-1;
1               for(size_t i=0;i!=n;++i){
3                       *(pStr1--)=*(pStr2--);
1               }
4
        }
1
5       return( );
}
```

- pStr1< pStr2 str1
- pStr1+n < pStr2 str2
- pStr1+n < pStr2 || pStr2+n<pStr1 str2
- pStr2+n<pStr1 str1



66、下面有关回调函数的说法，错误的是？

- 回调函数就是一个通过函数指针调用的函数
- 回调函数可能被系统 API 调用一次，也可能被循环调用多次
- 回调函数本身可以是全局函数 ，静态函数和某个特定的类的成员函数

- 回调函数可用于通知机制

  基础概念:回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就说这是回调函数。回调函数不是由该函数的实现方法直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。所谓的回调函数，就是预先在系统的对函数进行注册，让系统知道这个函数的存在，以后，当某个事件发生时，再调用这个函数对事件进行响应。 定义一个类的成员函数时在该函数前加 CALLBACK 即将其定义为回调函数，函数的实现和普通成员函数没有区别

67、使用 char* p = new char[100]申请一段内存，然后使用 delete p 释放，有什么问题？

- 会有内存泄露
- **不会有内存泄露，但不建议用**
- 编译就会报错，必须使用 delete []p;
- 编译没问题，运行会直接崩溃

the answer is B ， if the type is int,char,float, both delete[]p and delete p are ok,but if the type is class object, the answer will be A

68、下列 C 代码中，不属于未定义行为的有___

- Int i=0;i=(i++);
- char *p=”hello”;p[1]=’E’;
- char *p=”hello”;char ch=*p++;
- int i=0;printf(“%d%d\n”,i++,i--);
- 都是未定义行为
- 都不是未定义行为

应该选 C 吧。其他都是未定义行为。

A，D：表达式的先后顺序，是由编译器决定的，有可能不同

B：常量字符串不能修改，指针改为数组可以。

C 语言 undefined behaviour 未定义行为

C 语言中的未定义行为（Undefined Behavior）是指 C 语言标准未做规定的行为。同时，标准也从没要求编译器判断未定义行为，所以这些行为有编译器自行处理，在不同的编译器可能会产生不同的结果，又或者如果程序调用未定义的行为，可能会成功编译，甚至一开始运行时没有错误，只会在另一个系统上，甚至是在另一个日期运行失败。当一个未定义行为的实例发生时，正如语言标准所说，“什么事情都可能发生”，也许什么都没有发生。

所以，避免未定义行为，是个明智的决定。本文将介绍几种未定义行为，同时欢迎读者纠错和补充。

1.同一运算符中多个操作数的计算顺序（&&、||、?和,运算符除外）

例如：x = f()+g(); //错误

f()和 g()谁先计算由编译器决定，如果函数 f 或 g 改变了另一个函数所使用变量的值，那么 x 的结果可能依赖于这两个函数的计算顺序。

参考： 《C 程序设计语言（第 2 版）》 P43

2.函数各参数的求值顺序

例如： printf("%d,%d\n",++n,power(2,n)); //错误

在不同的编译器可能产生不同的结果，这取决于 n 的自增运算和 power 调用谁在前谁在后。

需要**注意**的是，不要和逗号表达式弄混，都好表达式可以参考这篇文章：[c 语言中逗号运算符和逗号表达式]

参考： 《C 程序设计语言（第 2 版）》 P43

3.通过指针直接修改 const 常量的值

直接通过赋值修改 const 变量的值，编译器会报错，但通过指针修改则不会，例如:

```
1  intmain()
2  {
3          const  inta=1;
4          int*b=(int*)&a;
5          *b=21;
```

```
6           printf("%d, %d",a,*b);
7           return0;
    }
```

a 输出值也由编译器决定。

69、Initialize integer i as 0, what's the value of i after the following operation?i += i > 0 ? i++ : i--;

- -2
- -1
- 0
- 1
- 2

>的优先级高于 +=
因此表达式等价于 i = i + (i > 0) ? i++ : i--;
首先执行 i>0 表达式，位 false，返回 0，所以又等价于 i = i + 0 ? i++ : i--;
这是一个赋值表达式，赋值表达式的返回值是表达式的值，i+0 还等于 0，
因此 ? 前面整体返回 0，即 false，因此执行 i--
所以最后 i 的值是 -1

70、以下程序用来统计文件中字符的个数(函数 feof 用以检查文件是否结束,结束是返回非零)

```
1   #include<stdio.h>
2   main()
3   {
4           FILE  *fp;
5           long    num=0;
6           fp=fopen("fname.dat",  "r"    );
7           while(_______)
8           {
9                   fgetc(  fp  );
```

```
                num++  ;
        }
        printf(  "  num=  %  d\n",num);
        fclose(  fp  );
}
```

下面选项中,填入横线处不能得到正确结果的是?

feof(fp)= =NULL

! feof(fp)

feof(fp)

feof(fp) == 0

71、有以下程序

```
#include  <stdio.h>
void    fun  (  char    *p,int    n  )
{
        char    b  [6]  =  "abcde";  int    i;
        for    (  i  =  0,p  =  b;i  <  n;i  +  +  )
                p  [i]  =  b  [i];
}
main(  )
{
        char    a  [6]  =  "ABCDE";
        fun  (a,5);
        printf  ("%s\n",A  )  ;
}
```

程序运行后的输出结果是?

- abcde
- ABCDE
- edcba
- EDCBA

72、函数 fun 的声明为 int fun(int *p[4]),以下哪个变量可以作为 fun 的合法参数（）

- int a[4][4];
- **int **a;**
- int **a[4]
- int (*a)[4];

73、int a[3][4]，下面哪个不能表示 a[1][1]?

- *(&a[0][0]+5)
- *(*(a+1)+1)
- *(&a[1]+1)
- ***(a+5)**

74、**set, map, vector**
map<int, int, greater<int> >//map 降序
set<int, greater<int> >//set 降序

75、串′ abababaababaa′ 的 next 数组为()

- 012345678999
- 012121111212
- 011234223456
- 0123012322345

next 数组的求解方法是：第一位的 next 值为 0，第二位的 next 值为 1，后面求解每一位的 next 值时，根据前一位进行比较。首先将前一位与其 next 值对应的内容进行比较，如果相等，则该位的 next 值就是前一位的 next 值加上 1；如果不等，向前继续寻找 next 值对应的内容来与前一位进行比较，直到找到某个位上内容的 next 值对应的内容与前一位相等为止，则这个位对应的值加上 1 即为需求的 next 值；如果找到第一位都没有找到与前一位相等的内容，那么需求的位上的 next 值即为 1。

76、字符串′ ababaabab′ 的 nextval 为()

- (0, 1, 0, 1, 0, 4, 1, 0, 1)
- (0, 1, 0, 1, 0, 2, 1, 0, 1)
- (0, 1, 0, 1, 0, 0, 0, 1, 1)
- (0, 1, 0, 1, 0, 1, 0, 1, 1)

i	0	1	2	3	4	5	6	7	8
s	a	b	a	b	a	a	b	a	b
next	-1	0	0	1	2	3	1	2	3

nextval[0] = -1;

s[1]=b != s[next[1]]=s[0]=a; nextval[1]=next[1]=0;

s[2]=a = s[next[2]]=s[0]=a, nextval[2]=nextval[0]=-1;

s[3]=b = s[next[3]]=s[1]=b, nextval[3]=nextval[1]0;

s[4]=a = s[next[4]]=s[2]=a, nextval[4]=nextval[2]=-1;

s[5]=a != s[next[5]]=s[3]=b, nextval[5]=next[5]=3;

s[6]=b = s[next[6]]=s[1]=b, nextval[6]=nextval[1]=0;

s[7]=a = s[next[7]]=s[2]=a, nextval[7]=nextval[2]=-1;

s[8]=b = s[next[8]]=s[3]=b, nextval[8]=nextval[3]=0;

nextval -1 0 -1 0 -1 3 0 -1 0

有的时候下标从 1 开始即 0 1 0 1 0 4 1 0 1

77、C++内存分配中说法错误的是：_____。

- 对于栈来讲，生长方向是向上的，也就是向着内存地址增加的方向
- 对于堆，大量的 new/delete 操作会造成内存空间的不连续
- 堆容易产生 memory leak D，堆的效率比栈要低的多
- 堆的效率比栈要低得多
- 栈变量引用容易逃逸
- 以上都对

解释：堆的生长方向向上,栈向下，A 堆的生长方向向上,栈向下 ；EBP 栈基址大，生长向着内存地址减少的方向。

78、若有以下程序

```
1
2
3
4
5
6
7
8
9
10
#include <stdio.h>
main( )
{
        FILE * fp;
        int    i,a[ 6]={1,2,3,4,5,6},k;
        fp  =  fopen("data.dat", "w+");
        for    (i=0;i<6;i+ +)
        {
                fseek( fp,0L,0);
                fprintf(fp,"%d\n",a [ i]); }
```

```
                rewind  (fp);
                fscanf(fp, "%d" ,&k);
                fclose(fp);
                printf("%d\n",k);
        }
}
```

则程序的输出结果是?

- 123456
- 1
- 6
- 21

本题考查文件操作函数,fseek 用于二进制方式打开的文件,移动文件读写指针位置。将文件内部的位置指针重新指向一个流(数据流/文件)的开头。所以最后只保留的 6,答案为 C。

79、下列代码可以通过编译吗？如何修改使其通过编译？

```
template struct sum {
    staticvoidfoo(T op1 , T op2){
      cout << op1 <<op2;
    }
}
sum::foo(1,3);
```

- 编译通过
- 应该去掉 static 关键字
- 调用应该如下：
sum<int>:: foo(1,3)
- 调用应该如下：
 sum:: <int>foo(1,3)