

1 有一种语言称为 lua，里面的数字只有一种类型（number），实际上是双精度浮点数。没有各种位数的整数，如 32 位、64 位整数等。那么关于该语言的说法错误的是？（B）

A.该语言可以用 number 类型的变量作为数组下标

B.该语言可以表示任意 32 位数字整数的数字 ID

C.该语言无法实现 32 位数字整数的按位与、或、异或运算

D.该语言可以正常进行双精度浮点数运算

下面程序会输出什么：(1 2 2 2)

```
static int a=1;
void fun1(void){    a=2;  }
void fun2(void){    int a=3;  }
void fun3(void){    static int a=4;  }
int main(int argc,char** args){
    printf("%d",a);
    fun1( );
    printf("%d",a);
    fun2( );
    printf( "%d", a);
    fun3( );
    printf("%d",a);
}
```

解析：

```
printf( "%d" ,a); //输出全局静态变量，所以输出 1
fun1( );          //a=2 所以会修改全局静态变量，输出 2
printf("%d",a);
fun2( );          //int a=3 是在 func2 里的局部变量，所以调用结束就释放了，不影响全局
的 a 值，所以输出 2
printf( "%d", a);
fun3( );          //也是局部变量，虽然是静态的但是不会影响全局 a 的值，仍然输出 2
printf("%d",a);
```

以下关于头文件，说法正确的是（）

A.#include <filename.h>，编译器寻找头文件时，会从当前编译的源文件所在的目录去找

B.#include "filename.h"，编译器寻找头文件时，会从通过编译选项指定的目录去找

C.多个源文件同时用到的全局整数变量，它的声明和定义都放在头文件中，是好的编程习惯

D.在大型项目开发中，把所有自定义的数据类型、全局变量、函数声明都放在一个头文件中，各个源文件都只需要包含这个头文件即可，省去了要写很多#include 语句的麻烦，是好的编程习惯。

- A:错误: #include 分两种, #include<>和#include""
B:对的
C:全局变量不应当放到头文件中,
D:错误, 按类别放入头文件

// 请问下面的程序一共输出多少个“-”? 为什么?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int main(void) {
    int i;
    for (i=0; i<2; i++) {
        fork();
        printf("-\n");
    }
    return 0;
}
```

答案: (8 个)

- 1.printf("-\n");对于行输出/n 有清除缓存的作用;
 - 2.fork () 可以复制父进程的缓存, 变量值等信息;
 - 3.i=0 时, 父进程 A 产生一个子进程 A1, 此时输出两行“-”;
 - 4.i=1 时, fork 使父进程 A 产生子进程 A2, A1 产生子进程 A3, 此时 A-A3 共产生 4 行“-”
(因为现在 A, A1 的输出行缓冲均为空);
- 总数为 6: $2(A) + 2(A1) + 1(A2) + 1(A3) = 6$;
对比 <http://www.nowcoder.com/profile/594239/test/1146384/7254#summary>
由于 A2, A3 会继承 A 和 A1 的输出行缓冲区, 所以会分别输出两个“-”, 加起来为: $2(A) + 2(A1) + 2(A2) + 2(A3) = 8$;

关于 static 用途说法正确的是? (BCD)

- A.声明静态外部类
B.声明静态全局变量

- C.声明静态函数
- D.声明静态局部变量

类型声明符在 C 语言里面主要有三个用途:

- 1.声明静态局部变量
- 2.声明静态外部全局变量
- 3.声明静态外部函数

解答:

- 1: 设置静态局部变量, 变量只定义一次, 不能被别的函数使用
- 2: 设置静态全局变量, 变量不能被外部文件所使用
- 3: 在类中设置静态函数, 只能访问静态变量

下面代码会输出什么: (4)

```
int main(int argc,char**argv){
    int a[4]={1,2,3,4};
    int*ptr=(int*)&a+1;
    printf("%d",*(ptr-1));
}
```

解析:

a 的类型是 int *
&a 的类型是 int **
&a+1,移动四个位置, 指向了 4 的下一个位置,
ptr 也指向 4 的下一个位置,
ptr 是 int *, ptr-1 移动一个位置, 指向了 4

在 32 位机器上, 下列代码中, sizeof (a) 的值
是: 24

```
class A
{
    int i;
    union U
    {
        char buff[13];
        int i;
    }u;
    void foo() { }
```

```

        typedef char* (*f)(void*);
        enum{red, green, blue} color;
    }a;

```

解析:

实例化 class A

int i 占 4 个字节

union U 实例化为 u 占取 16 个字节 (char 数组占 13 个字节, 但因为最大类型为 int, 所以占取只能为 4 字节的整数倍即最小 16 字节)

空函数不占取字节

未实例化指针不占取字节【特别注意】

枚举类型占取 4 个字节

总共占取 4+16+4=24 个字节

 请将下列构造函数补充完整, 使得程序的运行结果是 5

```

#include<iostream>
using namespace std;
class Sample{
public:
    Sample(int x){
        _____
    }
    ~Sample(){
        if(p) delete p;
    }
    int show(){
        return *p;
    }
private:
    int*p;
};

int main(){
    Sample S(5);
    cout<<S.show()<<endl;
    return 0;
}

```

答案:

A: *p=x;

B: p=new int(x);

C: `*p=new int(x);`

D: `p=&x;`

解析:

A 选项 `*p=x;` 代表将 `p` 指向的变量赋值为 `x`。

但这题中, `p` 没有初始值, 是个空指针, 所以无法使用这种方式赋值, 应该先给 `p` 赋值, 才能给 `*p` 赋值。

`p = new int(x);`

C 也是同理

给定如下代码: `int x[4]={0}; int y[4]={1};` 数组 `x` 和 `y` 的值为 ()

A.{0, 0, 0, 0}, {1, 1, 1, 1}

B.{0, 0, 0, 0}, {1, 0, 0, 0}

C.{0, 不确定}, {1, 不确定}

D.与编译器相关

下面关于虚函数和函数重载的叙述不正确的是()

A.虚函数不是类的成员函数

B.虚函数实现了 C++ 的多态性

C.函数重载允许非成员函数, 而虚函数则不行

D.函数重载的调用根据参数的个数、序列来确定, 而虚函数依据对象确定

解答:

虚函数也是类的成员函数, A 说法是不正确的;

虚函数和函数重载都实现了 C++ 的多态性, 但表现形式不一样, 函数重载调用根据参数个数、参数类型等进行区分, 而虚函数则是根据动态联编来确定调用什么, 故 BD 说法正确

函数重载可以是类的成员函数也可以是非成员函数, 比如:

`int fun(int a);`

`int fun(int a, int b);`

这就是非成员重载, 虚函数必须是成员函数了, 否则就失效了, 所以 C 对
扩展一下, 提醒大家注意虚函数和纯虚函数的区别

虚函数可以在子类中进行重载, 也可以不重载而沿用父类中的方法。但纯虚函数必须重载, 因为在其声明类中没有函数实现。`virtual void func () =0;`

注意看: 纯虚函数没有函数体的哈~

包含纯虚函数的类为抽象类, 抽象类不能声明对象, 只能作为基类

【知识点】关于函数的多态性:

多态性分为编译时多态性和运行时多态性，
编译时多态性通过静态编联完成，例如函数重载，运算符重载；
运行时多态性则是动态编联完成，主要通过虚函数来实现；

函数重载不需要是成员函数，在类外声明或定义的函数同样可以对其进行重载

重载的调用主要根据参数个数，参数类型，参数顺序来确定， **函数重载是忽略返回值的**

在一台主流配置的 PC 机上，调用 $f(35)$ 所需的时间大概是__C__。

- A.几毫秒
- B.几秒
- C.几分钟
- D.几小时

解析：

计算执行次数得到递推公式

$$f(n)=f(n-1)+....+f(0)$$

$$f(n)=2(f(n-2)+...+f(0))$$

$$f(n)=2^2(f(n-3)+...+f(0))$$

...

$$f(n)=2^{(n-1)}f(0)=2^{(n-1)}$$

$$f(35)=2^{34} \quad \text{大改 100 多亿}$$

计算机应该是每秒几百万次

所以算下来需要几个小时

malloc 函数进行内存分配是在什么阶段？

答：执行阶段

下列情况中，不会调用拷贝构造函数的是（B）

- A.用一个对象去【初始化】同一个类的另一个新对象时
- B.将类的一个对象赋值给该类的另一个对象时
- C.函数的形参对象，调用函数进行形参和实参结合时
- D.函数的返回值是类的对象，函数执行返回调用时

解析：

- 1、用一个对象去【初始化】同一个类的另一个新对象时
- 2、函数的形参对象，调用函数进行形参和实参结合时
- 3、函数的返回值是类的对象，函数执行返回调用时 将一个对象赋值给另一个对象，两个对象都存在，调用的是赋值构造函数，不涉及内存的分配。当被赋值的对象不存在调用的是拷贝构造函数。

而且，拷贝构造函数没有处理静态数据成员。

关于拷贝构造函数的几个细节：

1. 拷贝构造函数里能调用 `private` 成员变量吗？

解答：这个问题是在网上见的，当时一下子有点晕。其时从名子我们就知道拷贝构造函数其时就是一个特殊的构造函数，操作的还是自己类的成员变量，所以不受 `private` 的限制。

2. 以下函数哪个是拷贝构造函数,为什么？

```
X::X(const X&);  
X::X(X);  
X::X(X&, int a=1);  
X::X(X&, int a=1, int b=2);
```

解答：对于一个类 `X`，构造函数的第一个参数是下列之一：

- a) `X&`
- b) `const X&`
- c) `volatile X&`
- d) `const volatile X&`

且没有其他参数或其他参数都有默认值,那么这个函数是拷贝构造函数.

1. `X::X(const X&);` //是拷贝构造函数
2. `X::X(X&, int =1);` //是拷贝构造函数

3. 一个类中可以存在多于一个的拷贝构造函数吗？

解答：类中【可以存在超过一个拷贝构造函数】。

```
class X {  
public:  
    X(const X&);    // const 的拷贝构造  
    X(X&);         // 非 const 的拷贝构造  
};
```

注意,如果一个类中只存在一个参数为 `X&` 的拷贝构造函数,那么就不能使用 `const X` 或 `volatile X` 的对象实行拷贝初始化.

```

class X {
public:
    X();
    X(X&);
};

const X cx;
X x = cx;    // error

```

如果一个类中没有定义拷贝构造函数,那么编译器会自动产生一个默认的拷贝构造函数。
这个默认的参数可能为 `X::X(const X&)`或 `X::X(X&)`,由编译器根据上下文决定选择哪一个。

`static` 属于类并不属于具体的对象,所以 `static` 成员是 **不允许在类内初始化的**,那么 `static const` 成员是不是在初始化列表中呢?
答案是 NO

一是 `static` 属于类,它在未实例化的时候就已经存在了,而构造函数的初始化列表,只有在实例化的时候才执行。
二是 `static` 成员不属于对象。我们在调用构造函数自然是创建对象,一个跟对象没直接关系的成员要它做什么呢

构造函数中,成员变量一定要通过初始化列表来初始化的有以下几种情况:

- 1、`const` 常量成员,因为常量只能在初始化,不能赋值,所以必须放在初始化列表中;
- 2、引用类型,引用必须在定义的时候初始化,并且不能重新赋值,所以也要写在初始化列表中;
- 3、没有默认构造函数的类类型,因为使用初始化列表可以不必写构造函数;

三元条件运算符(问号表达式) `--> ? :` 的结合性为从右像左相结合

如果类的定义如下,则以下代码正确并且是良好编程风格的是:(A)

```

class Object
{
public:
    virtual ~Object() {}
}

```



```
//...  
};
```

```
A.std::auto_ptr<Object> pObj(new Object);  
B.std::vector<std::auto_ptr<Object*>> object_vector;  
C.std::auto_ptr<Object*> pObj(new Object);  
D.std::vector<std::auto_ptr<Object>> object_vector;
```

解析：

智能指针实质是一个栈对象，而并非指针类型。C++的 `auto_ptr` 的作用是动态分配对象以及当对象不再需要时自动执行清理。

使用 `std::auto_ptr`，要 `#include <memory>`。

- (1) 尽量不要使用 “`operator=`”（如果使用了，请不要再使用先前对象）。
- (2) 记住 `release()` 函数不会释放对象，仅仅归还所有权。
- (3) `std::auto_ptr` 最好不要当成参数传递（读者可以自行写代码确定为什么不能）。
- (4) `auto_ptr` 存储的指针应该为 `NULL` 或者指向动态分配的内存块。
- (5) `auto_ptr` 存储的指针应该指向单一物件（是 `new` 出来的，而不是 `new[]` 出来的）。`auto_ptr` 不能指向数组，因为 `auto_ptr` 在析构的时候只是调用 `delete`，而数组应该要调用 `delete[]`。
- (6) 使用 `auto_ptr` 作为成员变量，以避免资源泄漏。
- (7) `auto_ptr` 不能共享所有权，即不要让两个 `auto_ptr` 指向同一个对象。
- (8) `auto_ptr` 不能作为容器对象，STL 容器中的元素经常要支持拷贝，赋值等操作，在这过程中 `auto_ptr` 会传递所有权，那么 `source` 与 `sink` 元素之间就不等价了。

C++将父类的析构函数定义为虚函数，下列正确的是哪个？（A）

释放父类指针时能正确释放子类对象
释放子类指针时能正确释放父类对象
这样做是错误的
以上全错

在 `printf` 中的 `%` 作为转义符，两个 `%` 才相当于 1 个 `%`

-----、
关于 `struct` 和 `class`，下列说法正确的是（）

- A. `struct` 的成员默认是 `public`，`class` 的成员默认是 `private`
- B. `struct` 不能继承，`class` 可以继承
- C. `struct` 可以有无参构造函数

D. struct 的成员变量只能是 public

解析：

在 C 语言中结构体是不能继承的，但是在 C++ 中也有结构体的概念，此时结构体是可以继承的，只不过结构体继承的默认访问权限为 public，

类继承的默认访问权限为 private。当然 C++ 中结构体的成员变量访问权限也有三种：public、protected、private，默认访问权限是 public；

而在 C++ 类中的成员变量访问权限也是三种，默认访问权限为 private。C++ 中结构体可以有构造函数，这点和类是一样的。

使用模板类的原因：

- (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

以下函数用法正确的个数是：

```
void test1()
{
    unsigned char array[MAX_CHAR+1],i;
    for(i=0;i<=MAX_CHAR;i++){
        array[i]=i;
    }
}
char*test2()
{
    char p[] = "hello world";
    return p;
}
char *p =test2();
void test3(){
    char str[10];
    str++;
    *str='0';
}
```

解析：

第一个问题：

重点不在于 CHAR_MAX 的取值是多少，而是在于 i 的取值范围是多少。

一般 char 的取值范围是 -128 到 127，而 u char 则是 0~255，所以 i 的取值范围是 0~255。

所以当 CHAR_MAX 常量大于 255 时，执行 i++ 后，i 不能表示 256 以上的数字，所以导致无限循环。

第二个问题：

重点在于函数中 p 的身份，他是一个指针，还是数组名；

如果是指针 p，则 p 指向存放字符串常量的地址，返回 p 则是返回字符串常量地址值，调用函数结束字符串常量不会消失（是常量）。所以返回常量的地址不会出错。

如果是数组 p，则函数会将字符串常量的字符逐个复制到 p 数组里面，返回 p 则是返回数组 p，但是调用函数结束后 p 被销毁，里面的元素不存在了。

例子中 p 是数组名，所以会出错，p 所指的地址是随机值。

若是把 char p[]="hello"; 改成 char *p="hello"; 就可以了。

第三个问题：

重点在于 str++; 这实际的语句就是 str=str+1; 而 str 是数组名，数组名是常量，所以不能给常量赋值。（可以执行 str+1，但是不能 str=。）

```
-----  
  
3*foo(14,6)  
  3*foo(8,3)  
    3*foo(2,1)  
      3*foo(-4,0)  
        1  
  
3 * 3 * 3 * 3 * 1 = 81
```

1、栈区（stack）— 由编译器自动分配释放，存放为运行函数而分配的局部变量、函数参数、返回数据、返回地址等。

2、堆区（heap）— 一般由程序员分配释放，new, malloc 之类的，若程序员不释放，程序结束时可能由 OS 回收。

3、全局区（静态区）（static）— 存放全局变量、静态数据、常量。程序结束后由系统释放。

4、文字常量区 — 常量字符串就是放在这里的。程序结束后由系统释放。

5、程序代码区 — 存放函数体（类成员函数和全局函数）的二进制代码。

`dynamic_cast` :

继承体系安全向下转型或跨系转型; 找出某对象占用内存的起始点

`static_cast`:

同旧式 C 转型, 如 `int` 到 `double`

`const_cast`:

常用于去除某个对象的常量性

`reinterpret_cast`

不具备移植性, 常见用途是转化函数指针类型

下面是折半查找的实现, `data` 是按升序排列的数据, `x` 是查找下标, `y` 是查找的上标, `v` 是查找的数值, 返回 `v` 在 `data` 的索引, 若没找到返回-1。代码不正确是 1,2,4

```
public int bsearch(int[] data, int x, int y, int v) {
    int m;
    while(x<y){ //1
        m = x + (y-x)/2; //2
        if(data[m] == v) return m; //3
        else if(data[m] > v) y = m; //4
        else x = m; //5
    }
    return -1; //6
}
```

解析:

上下标没有写清楚, 题目所指的应该是 `[x,y)`, 这样 5 应该是 `m-1`

而在下标为 `[x,y]` 的情况下, 1,4,5 都是有问题的。。。正确版本应该是这样吧

```
while(x<=y) {
    m = x + (y-x)/2; //2
    if(data[m] == v) return m; //3
    else if(data[m] > v) y = m-1; //4
    else x = m+1; //5
}
```

补充: 这里下标是个坑, 记住上限有没有包含就可以对付 1,4,5 处的问题 (熟记理解两个版本的代码区别), 然后是 2, 写成 `x+(y-x)/2` 是防止 `xy` 都很大的情况下 `x+y` 越界。这样的话应对二分查找应该够了

下列给定程序中, 函数 `fun` 的功能是: 把形参 `a` 所指数组中的最小值放在元素 `a[0]` 中, 接着把 `a` 所指数组中的最大值放在 `a[1]` 元素中; 再把 `a` 所指数组元素中的次小值放在 `a[2]` 中, 把 `a`

索取数组元素中的次大值放在 **a[3]**，以此类推。

例如:若 **a** 所指数组中的数据最初排列为: **9,1,4,2,3,6,5,8,7**;按规则移动后，数据排列为:**1,9,2,8,3,7,4,6,5**。形参 **n** 中存放 **a** 所指数组中数据的个数。

规定 **fun** 函数中的 **max** 存放的当前所找的最大值,**px** 存放当前所找最大值得下标。请在程序的下画线处填入正确的内容并将下画线删除，使程序得出正确的结果。

解答: (选择排序变种问题)

```
#include<stdio.h>
#define N 9
void fun(int a[ ], int n)
{
    int i, j, max, min, px, pn, t;
    for (i = 0; i < n - 1; i += 2)
    {
        max = min = a[i];
        px = pn = i;
        for (j = i + 1; j < n; j++)
        {
            if (max < a[j])
            {
                max = a[j];
                px = j;
            }

            if (min > a[j] )
            {
                min = a[j];
                pn = j;
            }
        }
        //下面两个顺序如果变换的话，结果会有不对
        if (pn != i)
        {
            t = a[i];
            a[i] = min;
            a[pn] = t;
            if (px == i)
                px = pn;
        }
        if (px != i + 1)
        {
```

```

        t = a[i + 1];
        a[i + 1] = max;
        a[px] = t;
    }
}
}
int main( )
{
    int b[N] = {9, 1, 4, 2, 3, 6, 5, 8, 7};
    printf("\nThe original data:\n");
    for (int i = 0; i < N; i++)
        printf("% 4d", b[i]);
    printf("\n");
    fun(b, N);
    printf("\nThe data after mocinng \n");
    for (int i = 0; i < N; i++)
        printf("% 4d", b[i]);
    printf("\n");
}

```

32 位机上根据下面的代码，问哪些说法是正确的？（C）

```

signed char a = 0xe0;
unsigned int b = a;
unsigned char c = a;

```

- A. `a > 0 && c > 0` 为真
- B. `a == c` 为真
- C. b 的十六进制表示是：0xffffffe0
- D. 上面都不对

解答：

同等位数的类型之间的赋值表达式不会改变其在内存之中的表现形式，因此通过 `unsigned char c = a;` 语句，c 的位存储形式还是 0xe0

对于 B 选项，编译器首先检查关系表达式 `"=="` 左右两边 a,c 的类型，如果某个类型是比 int 的位宽小的类型，就会先进行 Integer Promotion，将其提升为 int 类型，至于提升的方法，是先根据原始类型进行位扩展（如果原始类型为 unsigned ,进行零扩展，如果原始类型为 signed,进行符号位扩展）至 32 位，再根据需要进行 unsigned to int 形式的转换。

因此：

a 为 signed char 型，位宽比 int 小，执行符号位扩展，被提升为 0xffffffe0;
 c 为 unsigned char 型，位宽比 int 小，执行零扩展，被提升为 0x000000e0;

经过以上步骤，再对两边进行比较，显然发现左右是不同的，因此 `==` 表达式值为 false。

再举一个例子:

```
signed int a = 0xe0000000
unsigned int b = a;
cout<< (b == a) <<endl;
```

结果为 1, 因为 a、b 位宽已经达到了 int 的位宽, 均不需要 Integer Promotion, 只需要对 a 执行从 unsigned to signed 的转换, 然而这并不影响其在内存中的表现形式, 因此和 b 比较起来结果为真。

STL 中的 unordered_map 和 priority_queue 使用的底层数据结构分别是什么?(B)

- A.rbtree,queue
- B hashtable,heap
- C.rbtree,heap
- D hashtable,queue

解析:

unordered_map: 是所谓的哈希 map, 很容易就选了 hashtable

priority_queue: 是所谓的优先级队列, 说白了就是一个二叉堆, 所以底层应该用 heap 实现, 并非名字中的 queue

所以选 B

拓展:

map: map 内部实现了一个红黑树, 该结构具有自动排序的功能, 因此 map 内部的所有元素都是有序的, 红黑树的每一个节点都代表着 map 的一个元素, 因此, 对于 map 进行的查找, 删除, 添加等一系列的操作都相当于是对红黑树进行这样的操作, 故红黑树的效率决定了 map 的效率。

unordered_map: unordered_map 内部实现了一个哈希表, 因此其元素的排列顺序是杂乱的, 无序的

int a=5, 则 ++(a++) 的值是?

答案: 编译出错

解析:

++ 是一目运算符, 自增运算, 它只能用于一个变量, 即变量值自增 1, 不能用于表达式。

++(a++) 里, 小括号优先。

(a++) 是表达式, 按运算规则, 不能对表达式作自增运算。

两个线程并发执行以下代码, 假设 a 是全局变量, 初始值是 1, 那么以下输出中 (ABCD) 是

可能的。

A.3_2_

B.2_3_

C.3_3_

D.2_2_

解析：

感觉都有可能吧：

++不能认为是原子操作，a 是全局变量，在内存中，则++a 一般被分为从内存取 a 到寄存器、+、回写到内存三步，考虑到并发。

设两个线程，[1]和[2]：

A

[1]读 a

[1]+1

[1]写 a // a = 2

[1]再读 a

[2]读 a

[2]+1

[2]写 a // a = 3

[2]再读 a

[2]写到屏幕上 // 3_

[1]写到屏幕上 // 2_

B

先执行所有[1]再执行所有[2]就得到 B 了。

C

[1]读 a

[1]+1

[1]写 a // a = 2

[2]读 a

[2]+1

[2]写 a // a = 3

[1]再读 a

[2]再读 a

...

就得到 3_3_了，但如果没有再读 a 的步骤，就是_2_3 或者_3_2 了。

D

[1]读 a

[2]读 a


```
[1]+1
[2]+1
[1]写 a // a = 2
[2]写 a // a = 2
```

就得到 2_2_了

下面代码的输出是什么？

```
class A
{
public:
    A() { }
    ~A() {cout<<"~A"<<endl;}
};

class B:public A
{
public:
    B(A &a):_a(a)
    {

    }
    ~B()
    {
        cout<<"~B"<<endl;
    }
private:
    A _a;
};

int main(void)
{
    A a;          //很简单，定义 a 的时候调用了一次构造函数
    B b(a);
}
```

答案：

```
~B
~A
~A
```

~A

解析:

要想搞明白该问题,需要理解基类构造析构函数、子类构造析构函数和子类成员变量构造析构函数的调用顺序。

对于构造函数:基类构造函数 > 子类成员变量构造函数 > 子类构造函数

对于析构函数:子类析构函数 > 子类成员变量析构函数 > 基类析构函数

可以看出构造函数的调用过程和析构函数的调用过程正好相反。

main 函数中首先构造变量 a,然后是 b。在构造 b 时首先调用 b 的基类 A 的构造函数,然后调用 b 中成员变量_a 的构造函数,最后调用 b 的构造函数。

main 函数调用结束返回时,变量的释放顺序跟变量的构造顺序正好相反。首先释放变量 b,然后是变量 a。

在释放变量 b 时,首先调用 b 的析构函数,然后析构变量 b 的成员_a,析构_a 时调用_a 的析构函数。再调用 b 的基类的析构函数。

然后是释放变量 a,调用 a 的析构函数。

本例子中应该将 A 的析构函数更改为 virtual 的,防止使用多态机制时出现子类对象无法释放的情况,本例子中没有用到多态机制,不存在该问题。

当一个类的某个函数被说明为 virtual,则在该类的所有派生类中的同原型函数都是虚函数。

解析:

基类的成员函数 设为 virtual,其派生类 的相应的函数也会自动变为虚函数

拓展:

其实吧,这题可以联想到重载,覆盖,隐藏的区别。

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）
-

看以下代码：

```
A *pa = new A[10];
```

```
delete pa;
```

则类 A 的构造函数和析构函数分别执行了几次（D）

A.1 1

B.10 10

C.1 10

D.10 1

`new` 的话执行构造函数，执行十次。`delete` 只删除了 `pa[0]` 执行一次析构

首先这个写法就是错误的,应该改成 `delete[] pa;`才对. 这样会调用 10 次 A 的构造函数和 10 次 A 的析构函数. 即答案是 B

但是这还没有完. 如果 A 是简单类型,比如 `int ,char ,double` 或者结构体, 那么动态创建简单类型的数组, 是可以调用 `delete pa;`这样来析构的,效果是和 `delete[] pa` 的效果一样,不会报错.

但是如果,A 是自己定义的一个类,那么动态创建了对象数组,用 `delete pa;`就会使程序崩溃,因为动态创建一个对象的内存实际上比 A 要大,有一些附加的内存需要存放附加的信息

函数：

```
fun (char* p) {  
    return p;  
}
```

的返回值是：（B）

A.无确切值

B.形参 p 中存放的地址值//注意与 C 的区别

C.一个临时存储单元的地址

D.行参 p 自身的地址值

解析:

这段代码只能在 C 语言中运行, 在 C++中是不行的, 由于 C 中允许默认返回值类型为 int。
题目答案选 p, 返回值就是【形参 p】的地址

```
-----  
  
void recursive(int n, int m, int o)  
{  
    if (n <= 0)  
    {  
        printf("%d,%d\n", m, o);  
    }  
    else  
    {  
        recursive(n - 1, m + 1, o);  
        recursive(n - 1, m, o + 1);  
    }  
}
```

以上函数的时间复杂度 ($O(n*m*o)$)

解析:

这个函数, 时间复杂度只与 n 有关, m 和 o 只是打酱油的。

n 会递归两次 n-1

n-1 会递归两次 n-2

n-2 会递归两次 n-3

所以, 是一个完全二叉树, 总共是 2^{n-1} 次, 时间复杂度 $O(2^n)$

给出以下定义:

char acX[]="abcdefg";

char acY[]={ 'a', 'b', 'c', 'd', 'e', 'f', 'g'};

则正确的叙述为 (C)

- A.数组 acX 和数组 acY 等价
- B.数组 acX 和数组 acY 的长度相同
- C.数组 acX 的长度大于数组 acY 的长度
- D.数组 acX 的长度小于数组 acY 的长度

解析:

acX 是字符串 系统最后添加"\0"作为结束符
acY 是数组 acY 是数组 acY 是数组 没有结束符!!!!

有如下程序段：

```
class A
{
    int _a;
public:
    A(int a): _a(a)
    {
    }
    friend int f1(A &);
    friend int f2(const A &);
    friend int f3(A);
    friend int f4(const A);
};
```

以下调用哪个是错误的：

- A.f1(0)//error
- B.f2(0)
- C.f3(0)
- D.f4(0)

解析：

~~非常量引用的初始值必须为左值~~

要理解这个先得理解左值和右值的概念

一个区分左值与右值的便捷方法是：看能不能对表达式取地址，如果能，则为左值，否则为右值。

本题举例：

执行 f1(0),实参 0 要传成 A 对象，那么执行

A &a1 = 0; //这是不行的。

执行 f2(0),实参 0 要传成 A 对象，那么执行

const A &a2 = 0;//这是可行的。

由多个源文件组成的 C 程序，经过编辑、预处理、编译，链接等阶段会生成最终的可执行程序。下面哪个阶段可以发现被调用的函数未定义（C）

- A.预处理
- B.编译
- C.连接
- D.执行

解析:

A: 预处理是 C 语言程序从源代码变成可执行程序的第一步,主要是 C 语言编译器对各种预处理命令进行处理,包括头文件的包含、宏定义的扩展、条件编译的选择等。

B: 编译之前,C 语言编译器会进行词法分析、语法分析 (-fsyntax-only),接着会把源代码翻译成中间语言,即汇编语言。编译程序工作时,先分析,后综合,从而得到目标程序。所谓分析,是指词法分析和语法分析;所谓综合是指代码优化,存储分配和代码生成。值得一提的是,大多数的编译程序直接产生机器语言的目标代码,形成可执行的目标文件,但也有的编译程序则先产生汇编语言一级的符号代码文件,然后再调用汇编程序进行翻译加工处理,最后产生可执行的机器语言目标文件。

C: 链接是处理可重定位文件,把它们的各种符号引用和符号定义转换为可执行文件中的合适信息(一般是虚拟内存地址)的过程。

//预处理:处理#include、宏替换和条件编译

//编译:生成目标文件(.cpp->.o)

//链接:链接多个目标文件

//例子:

//main.cpp

```
int main(){
    fun();
}
```

//编译不可通过,提示函数 fun()未声明

```
void fun();
int main(){
    fun();
}
```

//编译可通过,链接不可通过,提示函数 fun()未定义

下述程序的输出是__A__。

#include<stdio.h>

```
int main()
{
    static char *s[] = {"black", "white", "pink", "violet"};
    char **ptr[] = {s+3, s+2, s+1, s}, ***p;
    p = ptr;
    ++p;
    printf("%s", **p+1);
}
```

```
        return 0;
    }
```

- A.ink
- B.pink
- C.white
- D.hite

解析:

`char **ptr[]` 是一个还是一个指针数组,不过是一个二级指针数组,存的是 `s` 这个数组中,每个元素的地址

`p` 是一个三级指针,`ptr` 这个二级指针数组的数组名也是一个三级指针,

`p++` 只是往后跳了一个元素的位置,`*p` 就是第二个元素的内容,也就是指向 `s` 数组中 `s+3` 这个元素的地址,

`**p` 就得到了 `s` 数组中 `s+3` 这个元素的内容, `s` 数组中保存的是字符串的首地址,那就得到了"pink"这个元素的首地址,

在`**p+1` 就是这个地址向后偏移一个字节,也就指到了'i'上,

所以 `%s, **p+1` 还是"ink"

位运算应用口诀

清零取反要用与, 某位置一可用或

若要取反和交换, 轻轻松松用异或

打印数字 1-9 的全排列组合:

```
#include "stdio.h"
#define N 9
int x[N];
int count = 0;

void dump() {
    int i = 0;
    for (i = 0; i < N; i++) {
        printf("%d", x[i]);
    }
    printf("\n");
}
```

```

void swap(int a, int b) {
    int t = x[a];
    x[a] = x[b];
    x[b] = t;
}

void run(int n) {
    int i;
    if (N - 1 == n) {
        dump();
        count ++;
        return;
    }
    for (i = n; i < N; i++) {
        swap(n, i);
        run(n + 1);
        swap(n, i);
    }
}

int main() {
    int i;
    for (i = 0; i < N; i++) {
        x[i] = i + 1;
    }
    run(0);
    printf("* Total: %d\n", count);
}

```

以下程序输出结果是__0 1 2__

```

class A
{
public:
    A():m_iVal(0){test();}
    virtual void func() { std::cout<<m_iVal<<' ';}
    void test(){func();}
public:
    int m_iVal;
};

```



```

class B : public A
{
    public:
        B(){test();}
        virtual void func()
        {
            ++m_iVal;
            std::cout<<m_iVal<<' ';
        }
};

int main(int argc ,char* argv[])
{
    A*p = new B;
    p->test();
    return 0;
}

```

解析：

B 继承自 A, 先 A 构造函数，输出 0 ，然后 B 构造函数，B 的 test 继承自 A ，然后找 func 函数找到 B 自己的，所以输出 1 ，然后 p->test() 再执行一次，输出 2

拓展：

本问题涉及到两个方面：

- 1.C++继承体系中构造函数的调用顺序。
- 2.构造函数中调用虚函数问题。

C++继承体系中，初始化时构造函数的调用顺序如下

- (1)任何虚拟基类的构造函数按照他们被继承的顺序构造
- (2)任何非虚拟基类的构造函数按照他们被继承的顺序构造
- (3)任何成员对象的函数按照他们声明的顺序构造
- (4)类自己的构造函数

据此可知 A*p = newB;先调用 A 类的构造函数再调用 B 类的构造函数。

构造函数中调用虚函数,虚函数表现为该类中虚函数的行为，即在父类构造函数中调用虚函数，虚函数的表现就是父类定义的函数的表现。**why?** 原因如下：

假设构造函数中调用虚函数，表现为普通的虚函数调用行为，即虚函数会表现为相应的子类函数行为，并且假设子类存在一个成员变量 int a; 子类定义的虚函数的新的行为会操作 a 变量，在子类初始化时根据构造函数调用顺序会首先调用父类构造函数，那么虚函数回去操作 a，而因为 a 是子类成员变量，这时 a 尚未初始化，这是一种危险的行为，作为一种明智的选择应该禁止这种行为。所以虚函数会被解释到基类而不是子类。

对以下数据结构中 **data** 的处理方式描述正确的是(C)

```
struct Node
{
    int size;
    char data[0];
};
```

- A.data 将会被编译成一个 `char *`类型指针
- B.全部描述都不正确
- C.编译器会认为这就是一个长度为 0 的数组,而且会支持对于数组 **data** 的越界访问
- D.编译器会默认将数组 **data** 的长度设置为 1

下面关于 `typedef char *String_t;` 和 `#define String_d char *` 这两句在使用上的区别描述错误的是?

- A.`typedef char *String_t` 定义了一个新的类型别名, 有类型检查
- B.`#define String_d char *` 只是做了个简单的替换, 无类型检查
- C.前者在预编译的时候处理, 后者在编译的时候处理
- D.同时定义多个变量的时候有区别, 主要区别在于这种使用方式 `String_t a,b; String_d c,d;` `a,b,c` 都是 `char*`类型, 而 `d` 为 `char` 类型

解析:

`typedef char *String_t` 定义了一个新的类型别名, 有类型检查。而 `#define String_d char *` 只是做了个简单的替换, 无类型检查, 前者在编译的时候处理, 后者在预编译的时候处理。同时定义多个变量的时候有区别, 主要区别在于这种使用方式 `String_t a,b; String_d c,d;` `a,b,c` 都是

`char*`类型, 而 `d` 为 `char` 类型

注:

```
String_d char* c,d;
```

所以这个地方 `c` 是 `char*`, 而 `d` 是 `char` 类型【特别注意哈】

由于 `typedef` 还要做类型检查。。`#define` 没有。。所以 `typedef` 比 `#define` 安全。。

拓展:

`#define` 是预处理指令, 在编译预处理时进行简单的替换, 不作正确性检查, 不关含义是否

正确照样带入，只有在编译已被展开的源程序时才会发现可能的错误并报错。例如：

```
#define PI 3.1415926
```

程序中的：area=PI*r*r 会替换为 3.1415926*r*r

如果你把#define 语句中的数字 9 写成字母 g 预处理也照样带入。

2) typedef 是在编译时处理的。它在自己的作用域内给一个已经存在的类型一个别名，但是 You cannot use the typedef specifier inside a function definition。

3) typedef int * int_ptr;

与#define int_ptr int *

作用都是用 int_ptr 代表 int * ,但是二者不同，正如前面所说，#define 在预处理 时进行简单的替换，而 typedef 不是简单替换，而是采用如同定义变量的方法那样来声明一种类型。也就是说;

```
#define int_ptr int *
```

```
int_ptr a, b; //相当于 int * a, b; 只是简单的宏替换
```

```
typedef int* int_ptr;
```

```
int_ptr a, b; //a, b 都为指向 int 的指针,typedef 为 int* 引入了一个新的助记符
```

这也说明了为什么下面观点成立

```
typedef int * pint ;
```

```
#define PINT int *
```

那么：

```
const pint p ;//p 不可更改，但 p 指向的内容可更改
```

```
const PINT p ;//p 可更改，但是 p 指向的内容不可更改。
```

pint 是一种指针类型 const pint p 就是把指针给锁住了 p 不可更改

而 const PINT p 是 const int * p 锁的是指针 p 所指的物体。

3) 也许您已经注意到#define 不是语句 不要在行末加分号，否则 会连分号一块置换。

如下一段神奇的代码实现的功能是什么？

```
int miracle(unsigned int n)
{
    int m= n==0 ? 0:1;
    while (n=(n&(n-1)))
    {
        m++;
    }
}
```

```

    }
    return m;
}

```

- A.n 的二进制表示中“0”的个数
- B.n 的二进制表示的倒序值
- C.n 的二进制表示中“1”的个数
- D.一个均匀的哈希函数

解析：

$n \& (n-1)$ 表示将 n 中最右的一个 1 变成 0.

考虑二进制减法的手算过程，从最右位开始，是 1 就变成 0；是 0 就需要向左借一，左边也是 0 的话仍需向更左边的位借，直到遇见 1。此时再与 n 按位与，就将 n 左右一个出现的 1 变为了 0。

每次将一个 1 变为 0， m 增加 1。直到 n 变为了 0， m 就是 n 中 1 的个数。

```

struct s
{
    int x: 3;
    int y: 4;
    int z: 5;
    double a;
}
求 sizeof(s)

```

答案：16

解析：

x, y, z 分别占用 3, 4, 5 位， int 是 4 个字节 32 位，相当于 xyz 占用 4 个字节， double 占 8 个字节，按照对齐原则，前面补 4 位， $4+4+8=16$ 。有一个条件是默认 8 字节对齐，如果前面加上一句 `#pragma pack(4)` // 设定为 4 字节对齐 结果就是 12.

有些信息在存储时，并不需要占用一个完整的字节，而只需占用几个或一个二进制位。例如在存放一个开关量时，只有 0 和 1 两种状态，用一位二进制即可。为了节省存储空间，并使处理简单，C 语言提供了一种数据结构，称为“位域”或“位段”。所谓“位域”是把一个字节中的二进制位划分为几个不同的区域，并说明每个区域的位数。每个域有一个域名，允许在程序中按域名进行操作。这样就可以把几个不同的对象用一个字节的二进制位域来表示。

题目中的 x, y, z 是一个 int 型的 3 个位域，因此以一个 int 来计算， a 是一个 double 型，字符对齐，因此共占 16 个字节。

关于更多位域对齐知识，请参考下面的博客：

<http://www.cnblogs.com/bigrabbit/archive/2012/09/20/2695543.html>

对静态成员的不正确描述（C）

A.静态成员不属于对象，是类的共享成员

B. 静态数据成员要在类外初始化

C.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

D.非静态成员函数也可以操作静态数据成员

解析：

静态成员函数属于类，类名调用，不属于某个对象，而 this 指的是当前的对象，静态成员函数不拥有 this 指针，C 说法不正确

两个等价线程并发的执行下列程序，a 为全局变量，初始为 0，假设 printf、++、--操作都是原子性的，则输出肯定不是哪个？【A】

```
void foo() {  
    if(a <= 0) {  
        a++;  
    }  
    else {  
        a--;  
    }  
    printf("%d", a);  
}
```

A.01

B.10

C.12

D.22

解析：

每个线程进 foo 函数不止一次，那么我们暂且假设两个线程分别进入 foo 函数 X 次，假设给线程编号，线程 1 有 m 次被堵在 a++，线程 2 有 n 次被堵在 a++处，那么线程 1 必然会执行 (X-m)次 a--，线程 2 必然会执行 (X-n)次 a--，那么最终 a 的值为 (m+n)-((X-m)+(X-n))=2(m+n)-2X，那么 a 必然是偶数，

因此只有答案 A 是奇数，选择他不用怀疑。

看懂黑体字就是王道

参考：<http://www.cnblogs.com/obama/archive/2013/05/06/3061529.html>

```

-----
#include <iostream>
#include <vector>
using namespace std;
int main(void)
{
    vector<int>array;
    array.push_back(100);
    array.push_back(300);
    array.push_back(300);
    array.push_back(300);
    array.push_back(300);
    array.push_back(500);
    vector<int>::iterator itor;
    for(itor=array.begin();itor!=array.end();itor++)
    {
        if(*itor==300)
        {
            itor=array.erase(itor);
        }
    }
    for(itor=array.begin();itor!=array.end();itor++)
    {
        cout<<*itor<<" ";
    }
    return 0;
}

```

下面这个代码输出的是()

- A.100 300 300 300 300 500
- B.100 300 300 300 500
- C.100 300 300 500
- D.100 300 500
- E.100 500
- F.程序错误

答案：

vector::erase()：从指定容器删除指定位置的元素或某段范围内的元素

vector::erase()方法有两种重载形式

如下：

iterator erase(iterator_Where);

iterator erase(iterator_First,iterator_Last);

如果是删除指定位置的元素时：

返回值是一个迭代器，指向删除元素下一个元素；

如果是删除某范围内的元素时：返回值也表示一个迭代器，指向最后一个删除元素的下一个元素；

在本题中，当 `*itor==300` 成立时，删除第一个值为 300 的元素，同时 `itor` 指向下一个元素（即是第二个值为 300 的元素），

在 `for(;;itor++)` 执行 `itor`，`itor` 指向第三个值为 300 的元素，进入下一个循环

进入循环满足 `*itor==300`，重复上面的过程，执行完循环，`itor` 执行值为 500 的元素。

所有整个过程中，只删除了 2 个值为 300 的元素。

下列对函数 `double add(int a, int b)` 进行重载，正确的是（ABC）

A. `int add(int a, int b, int c)`

B. `int add(double a, double b)`

C. `double add(double a, double b)`

D. `int add(int a, int b)`

看错题了，，呵呵哒

以下 STL 的容器存放的数据，哪个肯定是排好序的（D）

A. `vector`

B. `deque`

C. `list`

D. `map`

解析：

`Map` 中的元素是自动按 `key` 升序排序，所以是排好序的

以下叙述中错误的是？（C）

A. 数值型常量有正值和负值的区分

B. 常量可以用一个符号名来代表

C. 定义符号常量必须用类型名来设定常量的类型

D. 常量是在程序运行过程中值不能被改变的量

解析：

数值型常量有整型常量、实型常量，不论是整型常量还是实型常量都有正值和负值之分，所以 A 正确。

在 C 语言的预编译处理中，可以用符号名代表一个常量，定义时不必指定常量类型【宏定义】，

所以 C 错误,B 正确。

常量的定义就是常量是在程序运行过程中值不能被改变的量,所以 D 正确。

【离散数学】

如果 A,B,C 为布尔型变量,“^”和“v”分别代表布尔类型的“与”和“或”,下面那一项是正确的 (I II III)

I. $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$

II. $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$

III. $(B \wedge A) \vee C = C \vee (A \wedge B)$

union Test

```
{
    char a[4];
    short b;
};
Test test;
test.a[0]=256;
test.a[1]=255;
test.a[2]=254;
test.a[3]=253;
printf("%d\n",test.b);
```

问题: 在 80X86 架构下, 输出什么值?

A.-128

B.-256

C.128

D.256

解析:

答案解析: 首先要知道大小端模式, 80X86 下是小端模式: 当然可以编写下测试就可以了, short 占 2 个字节, 设左高地址, 右低地址:

a[1]	a[0]
1111 1111	0000 0000

short 占用的是这 a[1]、a[0]两个字节, 最高位是 1 是一个负数, 在计算机中采用补码表示, 那么二进制表示为: 1000 0001 0000 0000, 转化为十进制就是-256.

char 类型的取值范围是-128~127, unsigned char 的取值范围是 0~256

这里 a[0]=256,出现了正溢出, 将其转换到取值范围内就是 0, 即 a[0]=0;

同理, a[1]=-1, a[2]=-2, a[3]=-3,在 C 语言标准里面, 用补码表示有符号数, 故其在计算机中的表示形式如下:


```
a[0]=0,      0000 0000
a[1]=-1,     1111 1111
a[2]=-2,     1111 1110
a[3]=-3,     1111 1101
```

short 是 2 字节 (a[0]和 a[1]), 由于 80X86 是小端模式, 即数据的低位保存在内存的低地址中, 而数据的高位保存在内存的高地址中, 在本例中, a[0]中存放的是 b 的低位, a[1]中存放的是 b 的高位, 即 b 的二进制表示是: 1111 1111 0000 0000, 表示-256, 故选 B。

看如何用 const 很多同志都分享什么就近原则啊, 看这个看那个,, 我真想说好累的记法!
看我的方法:

如: int const * const p;

找到第一个 const, 后面除了有个 const 之外 就是 * 和 p, 那么见证奇迹的时刻来了,
const * p 这就是我们要的结果, p 指向的内容不可变

找到第二个 const, 后面只有个 p, 那么见证奇迹的时候到了
const p,, 这就是结果, p 不可变

++规则总结如下: 找到 const, 再看后面除了 const 的内容, 是什么东西, 那个东西不能变
即是

```
typedef struct
{
char flag[3];
short value;
} sampleStruct;
union
{
char flag[3];
short value;
} sampleUnion;
```

假设 sizeof(char)=1,sizeof(short)=2,那么 sizeof(sampleStruct) = 6 , sizeof(sampleUnion) = 4

解析:

输出结果是: 6, 4

字符类型占 1 字节, 可以从任何地址开始

short 类型占 2 字节, 必须从 2 字节倍数地址开始

int 类型占 4 字节, 必须从 4 字节倍数地址开始

0000 a[0]

0001 a[1]

0002 a[2]

0003 空一个字节

0004 b 这个是 2 倍数的地址

0005

结构体在内存组织上是顺序式的，联合体则是重叠式，各成员共享一段内存，所以整个联合体的 `sizeof` 也就是每个成员 `sizeof` 的最大值。

【注意】为什么不用四字节对齐？

答：需要先取成员变量中的最大值，这里是 `short`，为 2，然后与对齐字节 4 比较，取小值，所以是 2

1<<3+2 的值是（32）。

分析：

按照运算符优先级，应先计算 3+2 的值，等于 5，然后把 1 的二进制表示左移 5 位得到 32

关于抽象类和纯虚函数的描述中，错误的是（D）

- A.纯虚函数的声明以“=0;”结束
- B.有纯虚函数的类叫抽象类，它不能用来定义对象
- C.抽象类的派生类如果不实现纯虚函数，它也是抽象类
- D.纯虚函数不能有函数体

解析：

纯虚函数和抽象类：

含有纯虚函数的类是抽象类，不能生成对象，只能派生。他派生的类的纯虚函数没有被改写，那么，它的派生类还是个抽象类。

定义纯虚函数就是为了让基类不可实例化,因为实例化这样的抽象数据结构本身并没有意义.或者给出实现也没有意义

如果一个类中至少有一个纯虚函数，那么这个类被称为抽象类（`abstract class`）。

举例

```
class virtualClass
{
    virtual int func() = 0; // 纯虚函数的
    virtual int func2(); // 虚函数的
    int func3();
};
```

`virtualClass c;` 不能这样实例化。

纯虚函数能有函数体

```
class virtualClass
{
virtual int func() = 0
{
int a = 0;
}
virtual int func2();
int func3();
};
```

考虑函数原型 `void hello(int a,int b=7,char* pszC="")`, 下面的函数调用中, 属于不合法调用的是

- A.hello(5)
- B.hello(5,8)
- C.hello(6,"#")
- D.hello(0,0,"#")

解析:

A 正确, 参数是按照从左向右赋值, 而第二个和第三个参数都缺省, 如果系统不提供, 将会用缺省值替代。

B 和 A 一样, 只不过第二个参数没有用缺省值, 用了自己给的值。

C 错误, 编译器编译时不是根据类型对号入座, 而是依次从左向右赋值, 先给第一个参数赋值, 在给第二个参数赋值时发现类型不对, 就会发生编译错误。

D 正确, 用了自己的值, 没有用缺省值。

关于函数模板, 描述错误的是? (A)

A. 函数模板必须由程序员实例化为可执行的函数模板

B. 函数模板的实例化由编译器实现

C. 一个类定义中, 只要有一个函数模板, 则这个类是类模板

D. 类模板的成员函数都是函数模板, 类模板实例化后, 成员函数也随之实例化

解析:

函数模板必须由编译器根据程序员的调用类型实例化为可执行的函数

编译器完成模板实例化

模板的成员函数只有调用时才会被实例化, 而不是和模板实例化一起被实例化

关于 c++ 的 `inline` 关键字, 以下说法正确的是(D)

A. 使用 `inline` 关键字的函数会被编译器在调用处展开

- B.头文件中可以包含 inline 函数的声明//【注意】仅仅声明是不够的
- C.可以在同一个项目的不同源文件内定义函数名相同但实现不同的 inline 函数
- D.定义在 Class 声明内的成员函数默认是 inline 函数【正确答案】
- E.优先使用 Class 声明内定义的 inline 函数
- F.优先使用 Class 实现的 inline 函数的实现

解析:

内联函数:

Tip: 只有当函数只有 10 行甚至更少时才将其定义为内联函数.

定义: 当函数被声明为内联函数之后, 编译器会将其内联展开, 而不是按通常的函数调用机制进行调用.

优点: 当函数体比较小的时候, 内联该函数可以令目标代码更加高效. 对于存取函数以及其它函数体比较短, 性能关键的函数, 鼓励使用内联.

缺点: 滥用内联将导致程序变慢. 内联可能使目标代码量或增或减, 这取决于内联函数的大小. 内联非常短小的存取函数通常会减少代码大小,

但内联一个相当大的函数将戏剧性的增加代码大小. 现代处理器由于更好的利用了指令缓存, 小巧的代码往往执行更快.

结论: 一个较为合理的经验准则是, 不要内联超过 10 行的函数. 谨慎对待析构函数, 析构函数往往比其表面看起来要更长, 因为有隐含的成员和基类析构函数被调用!

另一个实用的经验准则: 内联那些包含循环或 switch 语句的函数常常是得不偿失 (除非在大多数情况下, 这些循环或 switch 语句从不被执行).

有些函数即使声明为内联的也不一定会被编译器内联, 这点很重要; 比如虚函数和递归函数就不会被正常内联. 通常, 递归函数不应该声明成内联函数.(递归调用堆栈

的展开并不像循环那么简单, 比如递归层数在编译时可能是未知的, 大多数编译器都不支持内联递归函数). 虚函数内联的主要原因则是想把它函数体放在类定义内,

为了图个方便, 抑或是当作文档描述其行为, 比如精短的存取函数.

-inl.h 文件:

Tip: 复杂的内联函数的定义, 应放在后缀名为 -inl.h 的头文件中.

内联函数的定义必须放在头文件中, 编译器才能在调用点内联展开定义. 然而, 实现代码理论上应该放在 .cc 文件中, 我们不希望 .h 文件中有太多实现代码, 除非在

可读性和性能上有明显优势. 如果内联函数的定义比较短小, 逻辑比较简单, 实现代码放在 .h 文件里没有任何问题. 比如, 存取函数的实现理所当然都应该放在类定义

内. 出于编写者和调用者的方便, 较复杂的内联函数也可以放到 .h 文件中, 如果你觉得这样会使头文件显得笨重, 也可以把它萃取到单独的 -inl.h 中. 这样把实现和类定义分离开来, 当需要时包含对应的 -inl.h 即可.

A 如果只声明含有 inline 关键字, 就没有内联的效果. 内联函数的定义必须放在头文件中, 编译器才能在调用点内联展开定义. 有些函数即使声明为内联的也不一定会被编译器内联, 这点很重要; 比如虚函数和递归函数就不会被正常内联. 通常, 递归函数不应该声明成内联函数.

B 内联函数应该在头文件中定义, 这一点不同于其他函数. 编译器在调用点内联展开函数的

代码时，必须能够找到 `inline` 函数的定义才能将调用函数替换为函数代码，而对于在头文件中仅有函数声明是不够的。

C 当然内联函数定义也可以放在源文件中，但此时只有定义的那个源文件可以用它，而且必须为每个源文件拷贝一份定义(即每个源文件里的定义必须是完全相同的)，当然即使是放在头文件中，也是对每个定义做一份拷贝，只不过是编译器替你完成这种拷贝罢了。但相比于放在源文件中，放在头文件中既能够确保调用函数是定义是相同的，又能够保证在调用点能够找到函数定义从而完成内联(替换)。对于由两个文件 `compute.C` 和 `draw.C` 构成的程序来说，程序员不能定义这样的 `min()` 函数，它在 `compute.C` 中指一件事情，而在 `draw.C` 中指另外一件事情。如果两个定义不相同，程序将会有未定义的行为：

为保证不会发生这样的事情，建议把 `inline` 函数的定义放到头文件中。在每个调用该 `inline` 函数的文件中包含该头文件。这种方法保证对每个 `inline` 函数只有一个定义，且程序员无需复制代码，并且不可能在程序的生命期中引起无意的不匹配的事情。

D 正确。 定义在类声明之中的成员函数将自动地成为内联函数，例如：

```
1
class A {    public:    void Foo(int x, int y) { ... }    // 自动地成为内联函数    }
```

EF 在每个调用该 `inline` 函数的文件中包含该头文件。这种方法保证对每个 `inline` 函数只有一个定义，且程序员无需复制代码，并且不可能在程序的生命期中引起无意的不匹配的事情。最好只有一个定义！

一个类声明了纯虚函数，其派生类中没有对该函数定义，那该函数在派生类中仍为纯虚函数，凡是包含纯虚函数的类都是抽象类。

通过重载的依据是函数名，参数类型，参数个数，返回类型。

类的静态成员是所有对象共有的

内联是在编译是将目标代码插入的

静态成员变量必须在类外初始化，静态成员常量在类中初始化

如果一个类中声明了纯虚函数，其派生类中没有对该函数定义，那该函数在派生类中仍为纯虚函数，凡是包含纯虚函数的类都是抽象类。

通常重载函数调用的依据是函数名、参数类型、参数个数。

类的静态成员是属于类的而不是属于哪个对象的，因此可以说是所有对象所共有的。内联函数是在编译时将目标代码插入的

以下程序打印的两个字符分别是（A）

```
1.typedef struct object object
2.struct object
3.{
4.    char data[3];
5.};
6.
7.int main(void)
8.{
9.    object obj_array[3]={{'a','b','c'},
10.                        {'d','e','f'},
11.                        {'g','h','i'}};
12.    object*cur=obj_array;
13.    printf("%c %c\n",*(char*)((char*)(cur)+2),*(char*)(cur+2));
14.
15.    return 0
16.}
```

答案:

- A.c g
- B.b d
- C.g g
- D.g c

解析:

cur 本为 object 类型指针 第一个占位符输出时，参数中将 cur 先强制转化为字符型指针然后加 2，即指向 obj_array 数组的第一个元素中的字符 c，而第二个占位符输出时，参数中先让 cur 指针加 2，即到达 obj_array 第 3 个元素的地址处，然后强制转化为字符型指针，再用*获取 1 个字节的内容即字符 g

此题主要是考察指针的步长，指针加 1 是和指针所指向的类型相关的，如果 cur 强制被转化成字符类型的指针，那么加 2，就是加 2 个字符的大小，即为 c，如果 cur 是对象指针，那么加 2 的话，就指向第 3 个 object 类型的对象。

有如下程序

```
#include <stdio.h>
#define SQR( X ) X * X
main( )
{
    int a=10, k=2, m=1;
```

```

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

```

程序运行后的输出结果是? (1)

解析:

$a/ = 2+1*2+1/2+1*2+1=7$ $a=10/7=1$

输出数组的全排列:

```

void perm(int list[], int k, int m)
{
    if (k==m)
    {
        copy(list,list+m,ostream_iterator<int>(cout," "));
        cout<<endl;
        return;
    }

    for (int i=k; i<m; i++)
    {
        swap(list[k],list[i]);
        perm(list,k+1,m);
        swap(list[k],list[i]);
    }
}

```

$k==m$ and $\text{perm}(\text{list},k+1,m)$;for 循环的作用是为先把 index 值为 k 的元素后面的元素一次与 index 为 k 的元素交换,相当于得到 index 为 k 的元素可能的取值情况, 然后使用递归得到 index 为 k+1 的元素位置可能的所有取值。然而对 index 为 k 的位置元素进行取值的时候, 操作过后需要还原避免取下一个值的时候错误,因此就有了第二个 swap 操作。由上分析可知,第一个空格应该为 $k==m$, 当 index 值到了 m 的时候输出即可, 因为 index 为 m 后面已经没有元素与其进行对调.

以下数字在表示为 double (8 字节的双精度浮点数) 时存在舍入误差的有 (ABC)

A.2 的平方根

B.10 的 30 次方

C.0.1

D.0.5

E.100

8字节的共64位，按照标准的浮点数表示方法，应该是1位符号位，11位指数位，52位尾数位

对于 B ($2^{90} < B < 2^{100}$)来说，指数位是够了，但是尾数位会不会够呢？ $B = 2^{30} * 5^{30}$ 也就是说将 B 表示成二进制后，其后30位全为0，从其第一个不为0到最后一个不为0的二进制

表示位中，至少需要 $90 - 30 = 60$ 位来存储，而其尾数位只有52位，必然会产生舍入误差，所

以 B 是的

对于 C 来说，将 C 表示成二进制便知 $10[0.1] = 2[0.00011001100110011.....]$ ，亦为无限循环小数，所以将 0.1 表示成二进制的 `double` 型必然也会产生舍入误差

下列关于数组与指针的区别描述正确的是？(B)

A.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。

B.用运算符 `sizeof` 可以计算出数组的容量（字节数）

C.指针可以随时指向任意类型的内存块。

D.用运算符 `sizeof` 可以计算出指针所指向内容的容量（字节数）

解析：

A.还可以在堆上创建动态数组

B.`sizeof(数组名)`就是数组的容量

C.`const` 指针不可以

D. `char* str = "hello"; sizeof(str)`不能计算出内容的容量，只是指针的容量。

下面程序段包含4个函数,其中具有隐含 `this` 指针的是(D)

```
int f1();
class T
{
    public:static int f2();
    private:friend int f3();
    protect:int f4();
};
```

A.f1

B.f2

C.f3

D.f4

静态成员函数属于整个类所拥有，没有 `this` 指针
友元函数不是这个类的成员，没有
类的非静态成员函数 有

只有非静态类成员才有 `this` 指针，友元函数不是这个类的成员，静态成员函数没有 `this` 指针

```
-----  
void func(char(& p)[10])  
{  
    printf("%d\n", sizeof(p));  
}
```

`p` 是装 10 个 `char` 类型数据的数组的引用，其结果类似于 `char p[10]; printf("%d\n", sizeof(p));`

所以结果为 10

C++11 STL 中的容器

一、顺序容器：

`vector`：可变大小数组；

`deque`：双端队列；

`list`：双向链表；

`forward_list`：单向链表；

`array`：固定大小数组；

`string`：与 `vector` 相似的容器，但专门用于保存字符。

二、关联容器：

按关键字有序保存元素：（底层实现为红黑树）

`map`：关联数组；保存关键字-值对；

`set`：关键字即值，即只保存关键字的容器；【关键字不重复】

`multimap`：关键字可重复的 `map`；

`multiset`：关键字可重复的 `set`；

无序集合：

`unordered_map`：用哈希函数组织的 `map`；

`unordered_set`：用哈希函数组织的 `set`；

`unordered_multimap`：哈希组织的 `map`；关键字可以重复出现；

`unordered_multiset`：哈希组织的 `set`；关键字可以重复出现。

=====

三、其他项：

stack、queue、valarray、bitset

关于代码输出正确的结果是()

```
int main(int argc, char *argv[])
{
    string a="hello world";
    string b=a;
    if (a.c_str()==b.c_str())
    {
        cout<<"true"<<endl;
    }
    else cout<<"false"<<endl;
    string c=b;
    c="";
    if (a.c_str()==b.c_str())
    {
        cout<<"true"<<endl;
    }
    else cout<<"false"<<endl;
    a="";
    if (a.c_str()==b.c_str())
    {
        cout<<"true"<<endl;
    }
    else cout<<"false"<<endl;
    return 0;
}
```

A.false false false

B.true false false

C.true true true

D.true true false

解析：

理由如下：

1.c_str()返回值是 const char*，返回一个指向正规 C 字符串的指针；

2.string b=a 是 C++中 string 类的赋值操作，b 会开辟一个与 a 同等长度的内存空间，把 a 的字符串拷贝到 b 的内存空间中；

所以 if(a.c_str()==b.c_str())比较的是 2 个 const char*，很显然是不等的。

意思是说比较的是两个指针变量是吧，只是指针指向的内容相等，但指针地址不同。

instance 是 java 的二元运算符，用来判断他左边的对象是否为右面类（接口，抽象类，父类）的实例

下列一段 C++代码的输出是? (B)

```
#include "stdio.h"
class Base
{
public:
    int Bar(char x)
    {
        return (int)(x);
    }
    virtual int Bar(int x)
    {
        return (2 * x);
    }
};
class Derived : public Base
{
public:
    int Bar(char x)
    {
        return (int)(-x);
    }
    int Bar(int x)
    {
        return (x / 2);
    }
};
int main(void)
{
    Derived Obj;
    Base *pObj = &Obj;
    printf("%d,", pObj->Bar((char)(100)));
    printf("%d,", pObj->Bar(100));
}
```

A.100, -100

B.100, 50

C.200, -100

D.200, 50

解析:

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)

对成员函数指针的调用

指向公有非静态的成员函数，调用时必须创建一个对象。

```
class Container{
```

```
    public:
```

```
    void print(){
```

```
        printf("printf()");
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    void (Container::*p)();
```

```
//指明是指向成员函数的指针
```

```
    p = &Container::print;
```

```
    Container c;
```

```
    (c.*p)();
```

```
    return 0;
```

```
}
```

指向静态函数

```
class Container{
```

```
    public:
```

```
    static void print(){
```

```
        printf("printf()");
```

```
    }
```

```
};
```

```
int main()
```

```

{
    void (*p)();
    p = &Container::print;
    p();
    return 0;
}

```

若有以下程序

```

#include<stdio.h>
main()
{
    int s=0,n;
    for(n=0;n<4;n++)
    {
        switch(n)
        {
            default: s+=4;
            case 1: s+=1;break;
            case 2: s+=2;break;
            case 3: s+=3;
        }
    }
    printf("%d\n",s);
}

```

则程序的输出结果是? (B)

- A.10
- B.11
- C.13
- D.15

解析:

n=0 s=4 s=5

n=1 s=6

n=2 s=8

n=3 s=11

break 语句的作用是终止正在执行的 **switch** 流程,跳出 **switch** 结构或者强制终止当前循环,从当前执行的循环中跳出。题干中第一次循环 **n** 值为 0,执行 **default** 语句后的 **s+=4**,**s** 的值变为 4,执行 **case1** 语句后的 **s+=1**,**s** 的值变为 5,遇到 **break** 语句跳出 **switch**

语句,进入第二次循环。第二次循环 n 的值为 1,执行 case1 后的 s+=1, s 的之变为 6,遇到 break 语句跳出 switch 语句,进入第三次循环。第三次循环时 n 的值为 2,执行 case2 后的 s+=2,s 的值变为 8,遇到 break 语句跳出 switch 语句,进入第四次循环。第四次循环时 n 的值为 3,执行 case3 后的 s+=3,s 的值变为 11。再判断循环条件为假,退出循环打印 s 的值 11。

如果有一个类是 myClass , 关于下面代码正确描述的是:(C)

```
myClass::~~myClass(){
delete this;
this = NULL;
}
```

- A.正确, 我们避免了内存泄漏
- B.它会导致栈溢出
- C.无法编译通过
- D.这是不正确的, 它没有释放任何成员变量。

解析:

delete 的过程实际是先调用析构函数, 然后释放内存
在析构函数内调用 delete 会造成【递归调用】

面有关 c++内存分配堆栈说法错误的是? (B)

- A.对于栈来讲, 是由编译器自动管理, 无需我们手工控制; 对于堆来说, 释放工作由程序员控制
- B.对于栈来讲, 生长方向是向上的, 也就是向着内存地址减小的方向; 对于堆来讲, 它的生长方向是向下的, 是向着内存地址增加的方向增长。
- C.对于堆来讲, 频繁的 new/delete 势必会造成内存空间的不连续, 从而造成大量的碎片, 使程序效率降低。对于栈来讲, 则不会存在这个问题
- D.一般来讲在 32 位系统下, 堆内存可以达到 4G 的空间, 但是对于栈来讲, 一般都是有一定的空间大小的。

解析:

对于堆来讲, 生长方向是向上的, 也就是向着内存地址增加的方向; 对于栈来讲, 它的生长方向是向下的, 是向着内存地址减小的方向增长。

```
void func()
```

```
{
    char b[2]={0};
    strcpy(b,"aaaa");
}
```

以下说法那个正确(Debug 版崩溃，Release 版正常)

解析：

因为在 Debug 中有【ASSERT 断言】保护，所以要崩溃，而在 Release 优化中就会【删掉 ASSERT】，所以会出现正常运行。但是不推荐如此做，因为这样会覆盖不属于自己的内存，这是搭上了程序崩溃的列车，即未定义行为，出现什么后果都有可能的，程序员的职责是保证不越界，而不是追问越界之后会发生什么。就像生活中违法不一定立刻被抓，但是迟早是要被抓的。

有如下类模板定义：（）

```
template<class T> class BigNumber{
    long n;
public:
    BigNumber(T i):n(i){}
    BigNumber operator+(BigNumber b)
    {
        return BigNumber(n+b.n);
    }
};
```

已知 b1,b2 是 BigNumber 的两个对象,则下列表达式中错误的是?(D)

- A.3+3
- B.b1+3
- C.b1+b2
- D.3+b1

解析：

选项中+ 前面的操作数必须是一个 BigNumber 的对象，3 不是此对象，所以错了！

【特别注意】虚函数是为了实现动态绑定，【不能声明为虚函数】的有：

1、静态成员函数； 2、类外的普通函数； 3、构造函数； 4、【友元函数】

此外，还有一些函数可以声明为虚函数，但是没有意义，但编译器不会报错，如：

1、赋值运算符的重载成员函数：因为复制操作符的重载函数往往要求形参与类本身的类型一致才能实现函数功能，故形参类型往往是基类的类型，因此即使声明为虚函数，也把虚函数当普通基类普通函数使用。

2、内联函数：内联函数目的是在代码中直接展开（**编译期**），而虚函数是为了继承后能动态绑定执行自己的动作（**动态绑定**），因此本质是矛盾的，因此即使内联函数声明为虚函数，编译器遇到这种情况是不会进行 `inline` 展开的，而是当作普通函数来处理。因此声明了虚函数不能实现内联的，即内联函数可以声明为虚函数，但是毫无了内联的意义

内联函数是编译器就展开的，把代码镶嵌进程序，虚函数是动态绑定，运行期决定的。所以内联函数不能是虚函数

关于 `static` 变量，请选择下面所有说法正确的内容。

正确答案: A B C

- A.若全局变量仅在单个 C 文件中访问，则可以将这个变量修改为静态全局变量，以降低模块间的耦合度
- B.若全局变量仅由单个函数访问，则可以将这个变量改为该函数的静态局部变量，以降低模块间的耦合度
- C.设计和使用访问动态全局变量、静态全局变量、静态局部变量的函数时，需要考虑重入问题
- D.静态全局变量过大，可那会导致堆栈溢出

解析：

对于 D: 静态变量放在程序的全局数据区，而不是在堆栈中分配，所以不可能导致堆栈溢出，D 是错误的。

答案是 A、B、C。

`static` 表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，也可以形成静态 `static` 代码块，但是 Java 语言中没有全局变量的概念。

被 `static` 修饰的成员变量和成员方法独立于该类的任何对象。也就是说，它不依赖类特定的实例，被类的所有实例共享。

只要这个类被加载，Java 虚拟机就能根据类名在运行时数据区的方法区内定找到他们。因此，

【`static` 对象可以在它的任何对象创建之前访问，无需引用任何对象】

用 `public` 修饰的 `static` 成员变量和成员方法本质是全局变量和全局方法，当声明它类的对象时，不生成 `static` 变量的副本，而是类的所有实例共享同一个 `static` 变量。

`static` 变量前可以有 `private` 修饰，表示这个变量可以在类的静态代码块中，或者类的其他静态成员方法中使用（当然也可以在非静态成员方法中使用 - 废话），但是

不能在其他类中通过类名来直接引用，这一点很重要。实际上你需要搞明白，`private` 是访问权限限定，`static` 表示不要实例化就可以使用，这样就容易理解多了。

static 前面加上其它访问权限关键字的效果也以此类推。

以下哪种语法在 C++中是错误的？其中 X 为一 C++类（D）

- A.const X * x
- B.X const * x
- C.const X const * x
- D.X * const x

解析：

D 的语法是正确的 是**没有初始化**。但是如果把 D 答案放在一个类中作为成员，不初始化它，只要不实例化这个类就可以编译通过。

所以还是题目本身的问题。

math.h 的 abs 返回值(C)

- A.不可能是负数
- B.不可能是正数
- C.都有可能
- D.不可能是 0

解析：

c 中的函数申明为 int abs(int num);

正常情况下,

num 为 0 或正数时，函数返回 num 值；

当 num 为负数且不是最小的负数时（不要问我最小的 int 类型负数是多少，上面那个图里面有真相），函数返回 num 的对应绝对值数，即将内存中该二进制位的符号位取反，并把后面数值位取反加一；

当 num 为最小的负数时（即 0x80000000），由于正数里 int 类型 32 位表示不了这个数的绝对值，所以依然返回该负数。

这就是设计这个库函数的时候为什么把返回值设置为 int 而不是 unsigned 的原因，当然如果把返回值设置为 unsigned 是不是更加合理呢，这个也许有更好的解释，期待...

C++中函数可以嵌套调用，但是不可以嵌套定义。

写出下列程序在 X86 上的运行结果（B）

```

struct mybitfields
{
    unsigned short a : 4;
    unsigned short b : 5;
    unsigned short c : 7;
} test

```

```

void main(void)
{
    int i;
    test.a = 2;
    test.b = 3;
    test.c = 0;

    i = *((short *)&test);
    printf("%d\n", i);
}

```

- A.30
- B.50
- C.60
- D.20

解析：

这个题的为难之处呢，就在于前面定义结构体里面用到的冒号，如果你能理解这个符号的含义，那么问题就很好解决了。这里的冒号相当于分配几位空间，也即在定义结构体的时候，分配的成员 a 4 位的空间， b 5 位， c 7 位，一共是 16 位，正好两个字节。下面画一个简单的示意：

变量名 位数

```

test      15 14 13 12 11 10 9 | 8 7 6 5 4 | 3 2 1 0
test.a           |               | 0 0 1 0
test.b           | 0 0 0 1 1 |
test.c  0  0  0  0  0  0 0 0 |       |

```

在执行 `i = *((short *)&test);` 时，取从地址 `&test` 开始两个字节（short 占两个字节）的内容转化为 short 型数据，即为 0x0032，再转为 int 型为 0x00000032，即 50

按位加权计算，得出 $32+16+2=50$

若有以下程序

```

#include <stdio.h>
char *a =" you";
char b[ ] =" Welcome you to China!";
main()
{
    int i,j=0; char * p;

```

```

for(i=0;b[i]!='\0';i++)
{
    if (*a == b[i])
    {
        p=a;
        for(j=i;*p!='\0';j++)
        {
            if(*p == b[j]) break;
            p++;
        }
        if(*p == '\0')
            break;
    }
    printf("%s",&b[i]);
}

```

则程序的输出结果是?(D)

- A.China!
- B.to China!
- C.me you to China!
- D.you to China!

解释:

该程序首先定义*a 和 b [] ,并进行初始化。主函数中通过外层 for 循环语句,遍历字符数组 b [] ,并且将符合 if 条件语句的字符数赋给数组 p;for 内层循环语句,遍历字符数组 a []。再将符合 if 条件语句的结果输出。因此 D 选项正确。

关于"深拷贝", 下列说法正确的是(A)

- A.会拷贝成员数据的值和会拷贝静态分配的成员对象
- B.只会拷贝成员数据的值
- C.只会拷贝静态分配的成员对象
- D.只会拷贝动态分配的成员对象

解析:

在对与对象不产生修改的操作时, 使用浅拷贝。即多个指针指向同一对象的内存
否则, 深拷贝 (或写实拷贝), 形如 开辟对象等大的内存, 并用指向。

以下程序输出结果是_____。

```
class A
{
public:
    virtual void func(int val = 1)

    { std::cout<<"A->"<<val <<std::endl;}
    virtual void test()

    { func();}
};
class B : public A
{
public:
    void func(int val=0)
{std::cout<<"B->"<<val <<std::endl;}
};
int main(int argc ,char* argv[])
{
    B*p = new B;
    p->test();
return 0;
}
```

解析：

- A.A->0
- B.B->1
- C.A->1
- D.B->0
- E.编译出错
- F.以上都不对

解析：

B

缺省参数是静态绑定的，对于这个特性，估计没有人会喜欢。所以，永远记住：

“绝不重新定义继承而来的缺省参数（Never redefine function’ s inherited default parameters

v

“绝不重新定义继承而来的缺省参数（Never redefine function's inherited default parameters value.）”

记住：virtual 函数是动态绑定，而缺省参数值却是静态绑定。意思是你可能会在“调用一个定义于派生类内的 virtual 函数”的同时，却使用基类为它所指定的缺省参数值。

结论：绝不重新定义继承而来的缺省参数值！（可参考《Effective C++》条款 37）

对于本例：

```
1
2
3
B*p = newB;
```

```
p->test();
```

p->test()执行过程理解：

(1) 由于 B 类中没有覆盖（重写）基类中的虚函数 test()，因此会调用基类 A 中的 test()；

(2) A 中 test()函数中继续调用虚函数 fun()，因为虚函数执行动态绑定，p 此时的动态类型（即目前所指对象的类型）为 B*，因此此时调用虚函数 fun()时，执行的是 B 类中的 fun()；所以先输出“B->”；

(3) 缺省参数值是静态绑定，即此时 val 的值使用的是基类 A 中的缺省参数值，其值在编译阶段已经绑定，值为 1，所以输出“1”；

最终输出“B->1”。所以大家还是记住上述结论：绝不重新定义继承而来的缺省参数值！

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

函数外部问 x 等于什么？（C）

- A.5
- B.12
- C.0
- D.随机值

解析：

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

函数外全局变量，系统初始化为 0，函数内局部变量，是随机值，局部变量需要初始化才能使用

已知表达式++a 中的"++"是作为成员函数重载的运算符,则与++a 等效的运算符函数调用形式为(A)

- A.a.operator++()
- B.a.operator++(0)
- C.a.operator++(int)
- D.operator++(a,0)

解析:

C++规定后缀形式有一个 int 类型参数，当函数被调用时，编译器传递一个 0 做为 int 参数的值给该函数，这里是前缀

++单目运算符，所有重载只有一个参数，而 A 中的参数就是 this 指针，bcd 都可排除

下面的说法那个正确(A)

```
#define NUMA 10000000
#define NUMB 1000
int a[NUMA], b[NUMB];

void pa()
{
    int i, j;
    for(i = 0; i < NUMB; ++i)
        for(j = 0; j < NUMA; ++j)
            ++a[j];
}

void pb()
{
    int i, j;
    for(i = 0; i < NUMA; ++i)
        for(j = 0; j < NUMB; ++j)
            ++b[j];
}
```

- A.pa 和 pb 运行的一样快
- B.pa 比 pb 快
- C.pb 比 pa 快

D.无法判断

解析:

测试时 pb 比 pa 快, 数组 a 比数组 b 大很多, 可能跨更多的页, 缺页率高或者缓存命中更低, 所以 pb 快

一般情况下, 我们认为把大的循环放在里面, 效率会比较高

但是当如此题一样, 涉及不同的内存存取时, 把大的循环放在外面可以增大缓存命中率, 大幅提高效率。

下列程序编译时会出现错误, 请根据行号选择错误位置(AD)

```
#include <iostream>
using namespace std;
class A{
    int a1;
protected:
    int a2;
public:
    int a3;
};
class B: public A{
    int b1;
protected:
    int b2;
public:
    int b3;
};
class C:private B{
    int c1;
protected:
    int c2;
public:
    int c3;
};
int main(){
    B obb;
    C obc;
    cout<<obb.a1;//1
    cout<<obb.a2;//2
    cout<<obb.a3;//3
```

```

    cout<<obc.b1;//4
    cout<<obc.b2;//5
    cout<<obc.b3;//6
    cout<<obc.c3;//7
    return 0;
}

```

- A.1,2
- B.2,5,7
- C.3,4,7
- D.4,5,6

解析:

类的继承后方法属性变化:

private 属性不能够被继承。

使用 **private** 继承，父类的 **protected** 和 **public** 属性在子类中变为 **private**;

使用 **protected** 继承，父类的 **protected** 和 **public** 属性在子类中变为 **protected**;

使用 **public** 继承，父类中的 **protected** 和 **public** 属性不发生改变;

private, public, protected 访问标号的访问范围:

private: 只能由 1.该类中的函数、2.其友元函数访问。

不能被任何其他访问，该类的对象也不能访问。

protected: 可以被 1.该类中的函数、2.子类的函数、以及 3.其友元函数访问。

但不能被该类的对象访问。

public: 可以被 1.该类中的函数、2.子类的函数、3.其友元函数访问，也可以由 4.该类的对象访问。

注：友元函数包括 3 种：设为友元的普通的非成员函数；设为友元的其他类的成员函数；设为友元类中的所有成员函数。

i 的初始值为 0，i++在两个线程里面分别执行 100 次，能得到最大值是 200，最小值是 2。

解析:

最小值是 2 情况分析

线程 1

(内存)

线程 2

i = 0

1. 读取 $i = 0$
2. 执行 $i++$ 99 次

3. 线程 2 从内存总读入 i

$i = 0$

4. $i = 99 \rightarrow$ 将 $i = 99$ 写回内存 $i = 99$

5. 执行 $i++$ 一次, $i = 1$

6. 将 $i = 1$ 写入到内存中

$i = 1$

7. 读入 $i = 1$, 执行一次 $i++$
得到 $i = 2$
共计 100 次累加操作结束

8. 读入 $i = 1$,

执行 $i++$ 99 次直至结束得到

$i = 100$

9. 将 $i = 100$ 写回内存中

$i = 100$

10 将 $i = 2$ 写入到内存中

$i = 2$

如下函数, 在 32bits 系统 $\text{foo}(2^{31}-3)$ 的值是__1__(这里的^是指数的意思)

1
2

```
int foo (int x)
    return x & -x
```

解析:

此题应该考虑符号位

$2^{31}=10000000\ 00000000\ 00000000\ 00000000$ (这里的第一位是符号位, 以补码形式存储, 此时输出应该为-INT_MAX)

$2^{31}-3=01111111\ 11111111\ 11111111\ 11111101$ (源码=补码, 这里有两种思路理解: 思路一: $2^{31}-3$ 不溢出, 得到正数, 思路二: 两个补码相加)

$-(2^{31}-3)=11111111\ 11111111\ 11111111\ 11111101$ (源码), $10000000\ 00000000\ 00000000\ 00000011$ (补码)

两个补码做按位与后得到结果为 $00000000\ 00000000\ 00000000\ 00000001$ 结果为 1

或者:

$2^{31} = 0,10000000,00000000,00000000,00000000$ (原码)

$2^{31}-3 = 01111111,11111111,11111111,11111101$ (原码和补码相同)

$-(2^{31}-3)$ 原码 = $11111111,11111111,11111111,11111101$ (原码)

$-(2^{31}-3)$ 补码 = $10000000, 00000000, 00000000, 00000011$ (补码)

机器运算时是以补码形式计算的所以 $x \& -x = 01111111,11111111,11111111,11111101 \& 10000000, 00000000, 00000000, 00000011 = 1$

假设某 C 工程包含 a.c 和 b.c 两个文件,在 a.c 中定义了一个全局变量 foo, 在 b.c 中想访问这一变量时该怎么做?

- A.在 b.c 中同样定义同名的 foo
- B.a.c 中声明时 extern int foo
- C.b.c 中声明时 extern int foo
- D.在一个工程中就可以访问到, 不用做任何操作

解析:

extern 声明多文件共享变量的方法总结一下:

1.在一个源文件中定义,在其他需要使用的源文件中用 extern 声明。(仅一处定义,多处 extern)

2.在一个源文件中定义,在其对应的头文件中 extern 声明,在其他需要使用该共享变量的源文件中包含该头文件即可。

(更加标准的做法) 本题考察的是第一种做法。

总之，谁引用谁声明

直接调用的函数（不是库函数）：

read: unix 系统函数，POSIX 标准，无缓冲区
fread, 标准 C 库函数，有缓冲区

write, fwrite 同 read

以下代码中，A 的构造函数和析构函数分别执行了几次：

```
A*pa=new A[10];  
delete []pa;
```

答案：10 10

解析：

A*pa = new A[10]; 调用 new 分配原始未类型化的内存，在分配的内存上运行构造函数，即运行 10 次构造函数，对新建的对象进行初始化构造，返回指向分配并构造好的对象的数组指针

delete []pa; 对数组中的 10 个对象分别运行析构函数，然后释放申请的空间

x、y、t 均为 int 型变量，则执行语句：t=3; x=y=2; t=x++||++y; 后，变量 t 和 y 的值分别为 ____。

答案：

t=1 y=2

特别是 int 相关的类型在不同位数机器的平台下长度不同。C99 标准并不规定具体数据类型的长度大小，只规定级别。作下比较：

（1）16 位平台

char	1 个字节 8 位
short	2 个字节 16 位
int	2 个字节 16 位
long	4 个字节 32 位
指针	2 个字节 16 位

(2) 32 位平台

char	1 个字节 8 位
short	2 个字节 16 位
int	4 个字节 32 位
long	4 个字节 32 位
long long	8 个字节 64 位
指针	4 个字节 32 位

(3) 64 位平台

char	1 个字节
short	2 个字节
int	4 个字节
long	8 个字节 (区别)
long long	8 个字节
指针	8 个字节 (区别)

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。

编译器总是根据类型来调用类成员函数。但

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：一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。

```
#include <iostream.h>
class animal
{
public:
    animal(int height, int weight)
    {
        cout<<"animal construct"<<endl;
    }
};
class fish:public animal
{
public:
    int a;
    fish():animal(400,300), a(1)
    {
        cout<<"fish construct"<<endl;
    }
};
void main()
{
    fish fh;
}
```

=====

\0 空字符(NULL) 000
\ddd 任意字符 三位八进制
\xhh 任意字符 二位十六进制

=====

以下代码运行结果为（ ）

```
#include<stdio.h>
int main()
{
    uint32_t a = 100;
    while (a > 0)
    {
        --a;
    }
    printf("%d", a);
    return 0;
}
```

- A.-1
- B.100
- C.0
- D.死循环

解析：

无符号数可以取到 0 取不到负数
如果条件是 $a \geq 0$ 则死循环
另外 `typedef unsigned long uint32_t;` inttypes.h

=====

以下代码中的两个 `sizeof` 用法有问题吗？

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

答案：内外都有问题

解析：

```
char str[] = "aBcDe";
cout << "str 字符长度为: " << sizeof(str) / sizeof(str[0]) << endl;
// 输出为 6，sizeof (str) 包括 '\0'
```

=====

`clock()`就是该程序从启动到函数调用占用 CPU 的时间

`time(&t);`为获取系统时间

`localtime(&t);` 将一个 UTC 时间转为本地时间

=====

有以下程序

```
1  #include <stdio. h>
2  main()
3  {
4      int a[ 3 ] = {0}, i, j, k = 2;
5      for( i = 0; i < k; i ++ )
6          for( j = 0; j < k; j ++ )
7              a[ j ] = a[ i ] + 1;
8      printf("%d\n", a[ 1 ]);
9  }
```

程序运行后的输出结果是（3）

一共四步：

`i = 0`

```
j = 0 a[0] = a[0] + 1 = 1
j = 1 a[1] = a[0] + 1 = 2
i = 1
j = 0 a[0] = a[1] + 1 = 3
j = 1 a[1] = a[1] + 1 = 3
```

以下代码编译有错误，哪个选项能解决编译错误？（D）

```
1  class A {
2      public:
3          int GetValue() const {
4              vv = 1;
5              return vv;
6          }
7      private:
8          int vv;
9  };
```

- A. 改变成员变量"vv"为"mutable int vv"
- B. 改变成员函数"GetValue"的声明，以使其不是 const 的
- C. 都不能修复编译错误
- D. 都可以修复编译错误

解析：

mutalbe 的中文意思是“可变的，易变的”，跟 constant（既 C++中的 const）是反义词。

在 C++中，mutable 也是为了突破 const 的限制而设置的。被 mutable 修饰的变量，将永远处于可变的狀態，即使在一个 const 函数中。

我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成 const 的。但是，有些时候，我们需要在 const 的函数里面修改一些跟类状态无关的数据成员，那么这个数据成员就应该被 mutalbe 来修饰。

下面描述正确的是（C）

```
1  int *p1 = new int[10];
2  int *p2 = new int[10]();
```

- A. p1 和 p2 申请的空间里面的值都是随机值
- B. p1 和 p2 申请的空间里的值都已经初始化
- C. p1 申请的空间里的值是随机值，p2 申请的空间里的值已经初始化
- D. p1 申请的空间里的值已经初始化，p2 申请的空间里的值是随机值

解析：

在 C++primer(第 5 版)中关于 new 的讨论有：

1、new 单个对象

new 在自由空间分配内存，但其无法为其分配的对象命名，因次是无名的，分配之后返回一个指向该对象的指针。

1 int *pi = new int; // pi 指向一个动态分配的，未初始化的无名对象
此 new 表达式在自由空间构造一个 int 类型对象，并返回指向该对象的指针。

默认情况下，动态分配的对象是默认初始化的，这意味着内置类型或组合类型的对象的价值是无定义的，而类类型对象将用默认构造函数进行初始化。

2、new(多个对象)数组

new 分配的对象，不管单个对象还是多个对象的分配，都是默认初始化。但可以对数组进行值初始化，方法就是：在大小之后添加一对空括号。

```
1 int *pia = new int[10]; // 10 个未初始化 int
2 int *pia2 = new int[10](); // 10 个值初始化为 0 的 int
```

int 是内置类型，故而如果 new 以后，没有在后面的大括号里添加括号，就不会调用构造函数，仅仅是分配了内存而已。

对于内置类型而言，new 仅仅是分配内存，除非后面显示加(),相当于调用它的构造函数，对于自定义类型而言，只要一调用 new，那么编译器不仅仅给它分配内存，还调用它的默认构造函数初始化，即使后面没有加()

下列代码可以通过编译吗？如何修改使其通过编译？

```
1 template <class T>
2 struct sum {
3     static void foo(T op1 , T op2) {
4         cout << op1 << op2;
5     }
6 }
7 sum::foo(1, 3);
```

A. 编译通过

B. 应该去掉 static 关键字

C. 调用应该如下： sum<int>:: foo(1, 3)

D. 调用应该如下： sum:: <int>foo(1, 3)

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

```
{  
函数体  
}
```

其中 **template** 和 **class** 是关键字, **class** 可以用 **typename** 关键字代替, 在这里 **typename** 和 **class** 没区别, <>括号中的参数叫 **模板形参**, 模板形参和函数形参很相像, **模板形参不能为空**。一旦声明了模板函数就可以用模板函数的形参名声明类中的成员变量和成员函数, 即可以在该函数中使用**内置类型**的地方都可以使用模板形参名。模板形参需要调用该模板函数时提供的模板实参来初始化模板形参, 一旦编译器确定了实际的模板实参类型就称他实例化了函数模板的一个实例。比如 **swap** 的模板函数形式为

template <class T> void swap(T& a, T& b){},

当调用这样的模板函数时类型 **T** 就会被被调用时的类型所代替, 比如 **swap(a,b)**其中 **a** 和 **b** 是 **int** 型, 这时模板函数 **swap** 中的形参 **T** 就会被 **int** 所代替, 模板函数就变为 **swap(int &a, int &b)**。而当 **swap(c,d)**其中 **c** 和 **d** 是 **double** 类型时, 模板函数会被替换为 **swap(double &a, double &b)**, 这样就实现了函数的实现与类型无关的代码。

所以应该改成这样了。有点 **define** 的味道哦。

```
template<class T> struct sum{  
    static void foo(T op1 , T op2){  
        cout << op1 <<op2;  
    }  
};  
sum<int>::foo(1,3);
```

```
template<class T> struct sum{  
    static void foo(T op1 , T op2){  
        cout << op1 <<op2;  
    }  
};  
sum<int>::foo(1,3);
```

下面代码的输出结果是()

```
1  int main() {  
2      int pid;  
3      int num=1;  
4      pid=fork();  
5      if(pid>0) {  
6          num++;  
7          printf("in parent:num:%d addr:%x\n", num, &num);  
8      }  
9      else if(pid==0) {  
10         printf("in child:num:%d addr:%x\n", num, &num);  
11     }  
12 }
```

- A. 父子进程中输出的 num 相同, num 地址不相同
- B. 父子进程中输出的 num 不同, num 地址相同
- C. 父子进程中输出的 num 相同, num 地址也相同
- D. 父子进程中输出的 num 不同, num 地址不相同

解析:

fork () 之后, 操作系统会复制一个与父进程完全相同的子进程, 虽说是父子关系, 但是在操作系统看来, 他们更像兄弟关系, 这 2 个进程共享代码空间, 但是数据空间是互相独立的, 子进程数据空间中的内容是父进程的完整拷贝, 指令指针也完全相同, 但只有一点不同, 如果 fork 成功, 子进程中 fork 的返回值是 0, 父进程中 fork 的返回值是子进程的进程号, 如果 fork 不成功, 父进程会返回错误。可以这样想象, 2 个进程一直同时运行, 而且步调一致, 在 fork 之后, 他们分别作不同的工作, 也就是分岔了。这也是 fork 为什么叫 fork 的原因。

勘误: 子进程的 pid 是 0, 子进程的 getpid() 是它自己的进程号; 父进程中的 pid 值为子进程进程号, 只有父进程执行的 getpid() 才是他自己的进程号。

以下程序统计给定输入中每个大写字母的出现次数(不需要检查输入合法性)

```
1 void AlphabetCounting(char a[], int n) {
2     int count[26]={}, i, kind=0;
3     for(i=0; i<n; ++i) (1);
4     for(i=0; i<26; ++i) {
5         if(++kind>1) putchar(';');
6         printf("%c=%d", (2));
7     }
8 }
```

以下能补全程序, 正确功能的选项是(D)

- A. ++count[a[i]-'Z']; 'Z'-i, count['Z'-i]
- B. ++count['A'-a[i]]; 'A'+i, count[i]
- C. ++count[i]; i, count[i]
- D. ++count['Z'-a[i]]; 'Z'-i, count[i]
- E. ++count[a[i]]; 'A'+i, count[a[i]]

解析:

题意为输入设定全部是大写 (ASCII 码 A-Z 为 65-90, 递增), 所以有两种情况:

一、count[0;25] 存储 A-Z 的个数, 即 count[0] 存储 A 的个数, 于是 (1) ++count[a[i]-'A']; (2) 'A'+i, count[i];

二、count[0;25] 存储 Z-A 的个数, 即 count[0] 存储 Z 的个数, 于是 (1) ++count['Z'-a[i]]; (2) 'Z'-i, count[i]。

所以答案为 D。

在 32 位小端的机器上, 如下代码输出是什么: (B)

```
1 char array[12] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};
2 short *pshort = (short *)array;
```

```

3    int *pint = (int *)array;
4    int64 *pint64 = (int64 *)array;
5    printf("0x%x , 0x%x , 0x%llx , 0x%llx", *pshort , *(pshort+2) , *pint64 , *(pint+2));

```

A.0x201 , 0x403 , 0x807060504030201 , 0x0
 B.0x201 , 0x605 , 0x807060504030201 , 0x0
 C.0x201 , 0x605 , 0x4030201 , 0x8070605
 D.0x102 , 0x506 , 0x102030405060708 , 0x0

解析:

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。

char array[12] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08};

1, char 一字节，直观呈现的结果为：0x08-07-06-05-04-03-02-01（从后到前下标为 **0-7**）；

2, short 两字节，直观呈现的结果为：0x0807-0605-0403-0201（从后到前下标为 **0-3**）；

*pshort 从 0 开始，所以 0x201；

*(pshort+2)从 2 开始，所以 0x605；

3, int64 八字节，直观呈现的结果为 0x0807060504030201（从后到前下标为 **0**）；

*pint64 从 0 开始，所以 0x807060504030201；

4, int 四字节，直观呈现为 0x08070605-04030201（从后到前下标为 **0-1**）；

*(pint+2)从 2 开始，但是最多只到下标 1，后面位置默认为 0。

Fill the blanks inside class definition

```

1    class Test
2    {
3        public:
4            ____ int a;
5            ____ int b;
6        public:
7            Test::Test(int _a , int _b) : a( _a )
8            {
9                b = _b;
10           }
11    };
12    int Test::b;
13    int main(void)
14    {
15        Test t1(0 , 0) , t2(1 , 1);
16        t1.b = 10;
17        t2.b = 20;
18        printf("%u %u %u %u", t1.a , t1.b , t2.a , t2.b);
19        return 0;

```

```
20 }
```

Running result : 0 20 1 20

正确答案: B C

- A.static/const
- B.const/static
- C.--/static
- D. conststatic/static
- E.None of above

解析:

对于成员变量 a, 若它为 const 类型, 那么必须要使用 `Test::Test(int _a , int _b) : a(_a)` 这种初始化形式, 若它为普通成员变量, 也可以采取 `Test::Test(int _a , int _b) : a(_a)` 这种形式, 所以 a 可以为 const 或者普通类型, 由于 b 没有采取 `Test::Test(int _a , int _b) : b(_b)` 这种形式, 所以 b 一定不是 const 类型, 有 `main()` 中的 `t1.b` 和 `t2.b` 的输出都是 20 可以知道, b 是静态变量。

下列关于虚函数的说法正确的是 ()

正确答案: B C D

- A. 在构造函数中调用类自己的虚函数, 虚函数的动态绑定机制还会生效
- B. ~~在析构函数中调用类自己的虚函数, 虚函数的动态绑定机制还会生效~~
- C. 静态函数不可以是虚函数
- D. 虚函数可以声明为 inline

解析:

答案为 bcd

C、静态函数不可以是虚函数

因为静态成员函数没有 `this`, 也就没有存放 `vptr` 的地方, 同时其函数的指针存放也不同于一般的成员函数, 其无法成为一个对象的虚函数的指针以实现由此带来的动态机制。静态是编译时期就必须确定的, 虚函数是运行时期确定的。

D、虚函数可以声明为 inline

inline 函数和 virtual 函数有着本质的区别, inline 函数是在程序被编译时就展开, 在函数调用处用整个函数体去替换, 而 virtual 函数是在运行期才能够确定如何去调用的, 因而 inline 函数体现的是一种编译期机制, virtual 函数体现的是一种运行期机制。

因此, 内联函数是个静态行为, 而虚函数是个动态行为, 他们之间是有矛盾的。

函数的 inline 属性是在编译时确定的, 然而, virtual 的性质则是在运行时确定的, 这两个不能同时存在, 只能有一个选择, 文件中声明 inline 关键字只是对编译器的建议, 编译器是否采纳是编译器的事情。

我并不否认虚函数也同样可以用 inline 来修饰, 但你必须使用对象来调用, 因为对象是没有

所谓多态的，多态只面向行为或者方法，但是 C++ 编译器，无法保证一个内联的虚函数只会被对象调用，所以一般来说，编译器将会忽略掉所有的虚函数的内联属性。

相关知识点：什么函数不能声明为虚函数？

一个类中将所有的成员函数都尽可能地设置为虚函数总是有益的。

设置虚函数须注意：

- 1: 只有类的成员函数才能说明为虚函数；
- 2: 静态成员函数不能是虚函数；
- 3: 内联函数不能为虚函数；
- 4: 构造函数不能是虚函数；
- 5: 析构函数可以是虚函数，而且通常声明为虚函数。

请问下列代码的输出结果有可能是哪些（）？

```
1  #include<stdint.h>
2  #include<stdio.h>
3  union X
4  {
5      int32_t a;
6      struct
7      {
8          int16_t b;
9          int16_t c;
10     };
11 };
12 int main() {
13     X x;
14     x.a=0x20150810;
15     printf("%x,%x\n", x.b, x.c);
16     return 0;
17 }
```

正确答案: A C

- A. 2015, 810
- B. 50810, 201
- C. 810, 2015
- D. 20150, 810

解析：

32bit 宽的数 0x12345678

在 Little-endian 模式 CPU 内存中的存放方式（假设从地址 0x4000 开始存放）为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

而在 Big- endian 模式 CPU 内存中的存放方式则为:

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

以基本类型划分来看,从四个选项选出不同的一个。

正确答案: A 你的答案: A (正确)

enum
char
float
int

解析:

enum 不是基本类型。

下列代码编译时会产生错误的是 (D)

```
1  #include <iostream>
2  using namespace std;
3  struct Foo {
4      Foo() {}
5      Foo(int) {}
6      void fun() {}
7  };
8  int main(void) {
9      Foo a(10); //语句 1
10     a.fun(); //语句 2
11     Foo b(); //语句 3
12     b.fun(); //语句 4
13     return 0;
14     16.
15 }
```

正确答案: D

语句 1
语句 2
语句 3
语句 4

解析:

```
1  Foo b(); //语句 3      这是是函数定义语句。 所以语句 4 会报错。
2
3  Foo b;   这才是调用构造函数。
```

有如下程序:

```
1  #include <math.h>
2  #include <iostream>
3  using namespace std;
4
5  class Point
6  {
7      friend double Distance( const Point & p1, const Point & p2 )      /* ① */
8      {
9          double dx  = p1.x_ - p2.x_;
10         double dy  = p1.y_ - p2.y_;
11         return(sqrt( dx * dx + dy * dy ) );
12     }
13
14
15 public:
16     Point( int x, int y ) : x_( x ), y_( y )
17     {
18     }
19
20
21 private:
22     int      x_;
23     int      y_;
24 };
25
26
27 int main( void )
28 {
29     Point  p1( 3, 4 );
30     Point  p2( 6, 9 );
31
32     cout << Distance( p1, p2 ) << endl;      /* ③ */
33     return(0);
34 }
```


下列叙述中正确的是

正确答案: A

- A. 程序编译正确
- B. 程序编译时语句①出错
- C. 程序编译时语句②出错
- D. 程序编译时语句③出错

解析:

因为友元函数没有 **this** 指针, 则参数要有三种情况:

- 1 要访问非 **static** 成员时, 需要对象做参数;
 - 2 要访问 **static** 成员或全局变量时, 则不需要对象做参数;
 - 3 如果做参数的对象是全局对象, 则不需要对象做参数;
- 这个友元函数可以不需要对象做参数, 都在 **main** 函数作用范围内。

友元函数专题:

1、为什么要引入友元函数: 在实现类之间数据共享时, 减少系统开销, 提高效率

具体来说: 为了使其他类的成员函数直接访问该类的私有变量

即: 允许外面的类或函数去访问类的私有变量和保护变量, 从而使两个类共享同一函数

优点: 能够提高效率, 表达简单、清晰

缺点: 友元函数破坏了封装机制, 尽量不使用成员函数, 除非不得已的情况下才使用友元函数。

2、什么时候使用友元函数:

- 1) 运算符重载的某些场合需要使用友元。
- 2) 两个类要共享数据的时候

3、怎么使用友元函数:

友元函数的参数:

因为友元函数没有 **this** 指针, 则参数要有三种情况:

- 1、 要访问非 **static** 成员时, 需要对象做参数; --常用(友元函数常含有参数)

2、 要访问 **static** 成员或全局变量时，则不需要对象做参数

3、 如果做参数的对象是全局对象，则不需要对象做参数

友元函数的位置：

因为友元函数是类外的函数，所以它的声明可以放在类的私有段或公有段且没有区别。

友元函数的调用：

可以直接调用友元函数，不需要通过对象或指针

友元函数的分类：

根据这个函数的来源不同，可以分为三种方法：

1、普通函数友元函数：

a) 目的：使普通函数能够访问类的友元

b) 语法：声明位置：公有私有均可，常写为公有

声明： **friend + 普通函数声明**

实现位置：可以在类外或类中

实现代码：与普通函数相同（不加不用 **friend** 和类::）

调用：类似普通函数，直接调用

c) 代码：

[cpp] view plain copy

```
1. class INTEGER
2. {
3. private:
4.     int num;
5. public:
6.     friend void Print(const INTEGER& obj); //声明友元函数
7. };
8. void Print(const INTEGER& obj) //不使用 friend 和类::
9. {
10.     //函数体
```

注意这里是声明，不是定义

```

11. }
12. void main()
13. {
14.     INTEGER obj;
15.     Print(obj); //直接调用
16. }

```

2、类 Y 的所有成员函数都为类 X 友元函数—友元类

a) 目的：使用单个声明使 Y 类的所有函数成为类 X 的友元

它提供一种类之间合作的一种方式，使类 Y 的对象可以具有类 X 和类 Y 的功能

具体来说：

前提：A 是 B 的友元（=》A 中成员函数可以访问 B 中有所有成员，包括私有成员和公有成员--老忘）

则：在 A 中，借助类 B，可以直接使用 ~B。私有变量 ~ 的形式访问私有变量

b) 语法：声明位置：公有私有均可，常写为私有(把类看成一个变量)

声明：friend + 类名---不是对象啊

调用：

c) 代码：

[cpp] view plain copy

```

1. class girl;
2.
3. class boy
4. {
5. private:
6.     char *name;
7.     int age;
8. public:
9.     boy();
10.    void disp(girl &);
11. };
12.
13. void boy::disp(girl &x) //函数 disp()为类 boy 的成员函数，也是类 girl 的友元函数

```

```

14. {
15.     cout<<"boy's name is:"<<name<<",age:"<<age<<endl; //正常情况, boy 的成员函数
        disp 中直接访问 boy 的私有变量
16.     cout<<"girl's name is:"<<x.name<<",age:"<<x.age<<endl;
17.     //借助友元, 在 boy 的成员函数 disp 中, 借助 girl 的对象, 直接访问 girl 的私有变
        量
18.     //正常情况下, 只允许在 girl 的成员函数中访问 girl 的私有变量
19. }
20.
21. class girl
22. {
23. private:
24.     char *name;
25.     int age;
26.     friend boy;    //声明类 boy 是类 girl 的友元
27. public:
28.     girl();
29. };
30. void main()
31. {
32.     boy b;
33.     girl g;
34.     b.disp(g); //b 调用自己的成员函数, 但是以 g 为参数, 友元机制体现在函数 disp
        中
35. }

```

3、类 Y 的一个成员函数为类 X 的友元函数

a) 目的: 使类 Y 的一个成员函数成为类 X 的友元

具体而言: 而在类 Y 的这个成员函数中, 借助参数 X, 可以直接以 X.私有变量的形式访问私有变量

b) 语法: 声明位置: 声明在公有中 (本身为函数)

声明: **friend + 成员函数的声明**

调用: 先定义 Y 的对象 y---使用 y 调用自己的成员函数---自己的成员函数中使用了友元机制

c) 代码:

[cpp] view plain copy

```

1. class girl;

```

```

2. class boy
3. {
4. private:
5.     char *name;
6.     int age;
7. public:
8.     boy();
9.     void disp(girl &);
10. };
11.
12. class girl
13. {
14. private:
15.     char *name;
16.     int age;
17. public:
18.     girl(char *N,int A);
19.     friend void boy::disp(girl &); //声明类 boy 的成员函数 disp()为类 girl 的友元函数
20. };
21.
22. void boy::disp(girl &x)
23. {
24.     cout<<"boy's name is:"<<name<<",age:"<<age<<endl; //访问自己(boy)的对象成员,直接访问自己的私有变量
25.     cout<<"girl's name is:"<<x.name<<",age:"<<x.age<<endl;
26.     //借助友元,在 boy 的成员函数 disp 中,借助 girl 的对象,直接访问 girl 的私有变量
27.     //正常情况下,只允许在 girl 的成员函数中访问 girl 的私有变量
28. }
29. void main()
30. {
31.     boy b();
32.     girl g();
33.     b.disp(g);
34. }

```

4、在模板类中使用友元 `operator<<`(对<<运算符的重载)

a)使用方法:

在模板类中声明:

[cpp] view plain copy

```
1. friend ostream& operator<< <>(ostream& cout, const MGraph<VexType, ArcType>& G);
```

在模板类中定义：

[cpp] view plain copy

```
1. template<class VexType, class ArcType>
2. ostream& operator<<(ostream& cout, const MGraph<VexType, ArcType>& G)
3. {
4.     //函数定义
5. }
```

b)注意：

把函数声明非模板函数：

[cpp] view plain copy

```
1. friend ostream& operator<< (ostream& cout, const MGraph& G);
```

把函数声明为模板函数：

[cpp] view plain copy

```
1. friend ostream& operator<< <>(ostream& cout, const MGraph<VexType, ArcType>& G);
```

或：

[cpp] view plain copy

```
1. friend ostream& operator<< <VexType, ArcType>(ostream& cout, const MGraph<VexType, ArcType>& G);
```

说明：

在函数声明中加入 `operator<< <>`：是将 `operator<<` 函数定义为函数模板，将函数模板申明为类模板的友元时，是一对一绑定的

实际的声明函数：这里模板参数可以省略，但是尖括号不可以省略

[cpp] view plain copy

```
1. friend ostream& operator<< <VexType, ArcType>(ostream& cout, const MGraph<VexType, ArcType>& G);
```

5、友元函数和类的成员函数的区别：成员函数有 `this` 指针，而友元函数没有 `this` 指针。

6、记忆：A 是 B 的友元 《=》 A 是 B 的朋友 《=》 借助 B 的对象，在 A 中可以直接 通过 B。成员变量（可以是公有，也可以为私有变量） 的方式访问 B

关于 do 循环体 while(条件表达式)，以下叙述中正确的是？ D

- A. 条件表达式的执行次数总是比循环体的执行次数多一次
- B. 循环体的执行次数总是比条件表达式执行次数多一次
- C. 条件表达式的执行次数与循环体的执行次数一样
- D. 条件表达式的执行次数与循环体的执行次数无关

看以下代码：

```
1   class A
2   {
3       public:
4       ~A();
5   };
6   A::~A()
7   {
8       printf("delete A ");
9   }
10  class B : public A
11  {
12      public:
13      ~B();
14  };
15  B::~B()
16  {
17      printf("delete B ");
18  }
```

请问执行以下代码

```
1   A *pa = new B();
2   delete pa;
```

输出的串是（）

正确答案: A

delete A

delete B

delete B delete A

delete A delete B

解析:

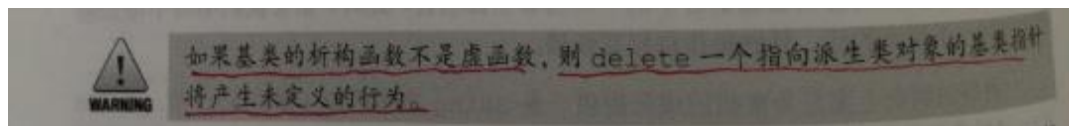
A 基类析构函数未加 `virtual`，因此不会调用子类析构

析构函数做最后的“清场工作”，

一般当派生类对象从内存中撤销时，先调用派生类的析构函数，再调用基类析构函数；但，若用 `new` 建立了临时对象，在用 `delete` 撤销对象是，系统会只执行基类的析构函数，而不执行派生类的析构函数。

如果将基类的析构函数声明为虚函数，由基类所有派生类的析构函数自动成为虚函数（即使析构函数名不同）。

将析构函数加上 `virtual`，实现具体对象的 **动态关联**，在运行阶段，先调用派生类析构，在调用基类析构



关于函数的描述正确的是___。

正确答案: D 你的答案: D (正确)

虚函数是一个 `static` 型的函数

派生类的虚函数与基类的虚函数具有不同的参数个数和类型

虚函数是一个非成员函数

基类中说明了虚函数后，派生类中起对应的函数可以不必说明为虚函数

解析:

- 1 在派生类中重新定义该虚函数时，关键字 `virtual` 可以写也可以不写。
- 2 一个虚函数无论被公有继承多少次，它仍然保持其虚函数的特性。
- 3 虚函数**必须是其所在类的成员函数，而不能是友元函数，也不能是静态成员函数（static）**

以关键字 `virtual` 的成员函数称为虚函数，主要是用于运行时多态，也就是动态绑定。

虚函数必须是类的成员函数，**不能使友元函数，也不能是构造函数**【原因：因为建立一个派生类对象时，必须从类层次的根开始，沿着继承路径逐个调用基类的构造函数，直到自己的构造函数，不能选择性的调用构造函数】

不能将虚函数说明为全局函数，也不能说明为 `static` 静态成员函数。因为虚函数的动态绑定必须在类的层次依靠 `this` 指针实现。

再添加一点:

虚函数的重载特性：一个派生类中定义基类的虚函数是函数重载的一种特殊形式。

重载一般的函数：函数的返回类型和参数的个数、类型可以不同，仅要求函数名相同；

而**重载虚函数：**要求函数名、返回类型、参数个数、参数类型和顺序都完全相同。

纯虚函数：是在基类中说明的虚函数，它在基类中没有是在定义，要求所有派生类都必须定义自己的版本。

纯虚函数的定义形式：`virtual 类型 函数名(参数表)=0`，该函数赋值为 0，表示没有实现定义。在基类中定义为 0，在派生类中实现各自的版本。

纯虚函数与抽象类的关系：

抽象类中至少有一个纯虚函数。

如果抽象类中的派生类没有为基类的纯虚函数定义实现版本，那么它仍然是抽象类，相反，定义了纯虚函数的实现版本的派生类称为具体类。

抽象类在 C++ 中有以下特点：

1. 抽象类只能作为其他类的基类；
2. 抽象类不能建立对象；
3. 抽象类不能用作参数类型、参数返回类型或显示类型转换。
- 4.

构造函数调用的时机：

- 1、当用类的一个对象初始化该类的另一个对象时
- 2、如果函数的形参是类的对象,调用函数时,进行形参和实参结合时.
- 3、如果函数的返回值是类的对象,函数执行完成返回调用者时.
- 4、需要产生一个临时类对象时

```
1  int main() {
2      int a;float b,c;
3      scanf("%2d%3f%4f",&a,&b,&c);
4      printf("\na=%d,b=%d,c=%f\n",a,b,c);
5  }
```

若运行时从键盘上输入 9876543210I,则上面程序的输出结果是

正确答案: B

a=98,b=765,c=4321.000000

a=98,b=0,c=0.000000

a=98,b=765.000000,c=4321.000000

a=98,b=765.0,c=4321.0

解析：

`printf` 函数执行的时候，会先把这三个数字压入栈里，然后再执行打印。压入栈的时候按照数据本身的长度来，首先把 `c` 和 `b` 压入，并且每一个都是 8 个字节（`printf` 自动转化为 `double`）。然后再压入 `a` 是 4 个字节。然后再执行打印。打印的时候按照用户指定的格式来出栈。首先打印 `a`，`a` 打印正常。然后又打印 4 个字节长度的 `b`，在栈里面由于 `b` 长度是八个字节，并且 `b` 目前是 64 位的表示方式，数据的后面全是 0。（`float` 变 `double`），电脑是小端存储方式，0 存储在距离 `a` 近的地方。打印 `b` 的时候，打印的 4 个字节都是 0。然后再打印 `c`，`c` 用正常的方式打印，会一下子读取 8 个字节，正好，读出来的八个字节前面四个字节全是 0，自己可以算一下，实在太小了，因此为 0。

栈底

栈顶

高字节。。。。。。。。。。低字节

4321	0000	765	0000	98
4 字节	4 字节	4 字节	4 字节	4 字节
打印 c		打印 b		打印 a

附：浮点数（单精度的 **float** 和双精度的 **double**）在内存中以二进制的科学计数法表示，表达式为 $N = 2^E * F$ ；其中 **E** 为阶码（采用移位存储），**F** 为尾数。

float 和 **double** 都由符号位、阶码、尾数三部分组成，**float** 存储时使用 4 个字节，**double** 存储时使用 8 个字节。各部分占用位宽如下所示：

	符号位	阶码	尾数	长度
float	1	8	23	32
double	1	11	52	64

编译和执行如下 c 语言代码,系统将会输出什么？

```
1  #include<stdio.h>
2  int main()
3  {
4      char c='0';
5      printf("%d %d", sizeof(c), sizeof('0'));
6      return 0;
7  }
```

正确答案: A 你的答案: C (错误)

1 4
2 2
1 1
2 1

解析：

C 语言: char a = 'a'; sizeof(char) = 1 sizeof(a) = 1 sizeof('a') = 4

C++语言: char a = 'a'; sizeof(char) = 1 sizeof(a) = 1 sizeof('a') = 1

字符型变量是 1 字节这个没错，奇怪就奇怪在 C 语言认为'a'是 4 字节，而 C++语言认为'a'是 1 字节。

原因如下：

C99 标准的规定，'a'叫做整型字符常量(integer character constant)，被看成是 int 型，所以在 32 位机器上占 4 字节。

ISO C++标准规定，'a'叫做字符字面量(character literal)，被看成是 char 型，所以占 1 字节

在 64 位系统下，分别定义如下两个变量：char *p[10]; char(*p1)[10];请问，sizeof(p) 和 sizeof (p1)分别值为_____。

正确答案: B 你的答案: D (错误)

- 4, 40
- 80, 8
- 10, 10
- 8, 80
- 40, 4
- 4, 4

解析:

重点理解 p 跟谁结合了，跟[]结合，则 p 就是一个数组;跟*结合，p 就是一个指针；
首先[]()的优先级一样，均大于*

char *p[10]，p 与[]结合，所以 p 就是一个数组，数组的元素比较特殊，是指针，指针大小为 8，所以是 10*8=80；

char(*p1)[10]，与*结合，所以是一个指针，大小为 8

下面关于数组的描述错误的是：

正确答案: C D 你的答案: D (错误)

在 C++语言中一维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是确定

数组只能通过值参数和引用参数两种方式传递给函数

解析:

选 CD

A.数组名字退化为指针，指向数组第一个元素，所以 A 正确

B.长度为 n 的数组，下标从 0 开始计算一直到 n-1，所以 B 正确

C.数组大小是静态的，所以在运行时必须能确定，编译时不一定需要确定，所以 C 错误

D.数组还可以通过数组指针传递给函数，所以 D 错误

下列程序的输出结果:

```
1  #include <iostream>
2  using namespace std;
3  class A
4  {
5  public:
6      void print ()
7      {
8          cout << "A:print()";
9      }
10 };
11 class B: private A
12 {
13 public:
14     void print ()
15     {
16         cout << "B:print()";
17     }
18 };
19 class C: public B
20 {
21 public:
22     void print ()
23     {
24         A:: print ();
25     }
26 };
27 int main()
28 {
29     C b;
30     b.print();
31 }
```

正确答案: C 你的答案: B (错误)

A:print()

B:print()

编译出错

解析:

B 的继承为私有继承,对于 C 已经不能再调用 A 的所有方法了

下列代码编译时会产生错误的是 ()

正确答案: D 你的答案: C (错误)

```
[cpp]
01. #include <iostream>
02. using namespace std;
03. struct Foo
04. {
05.     Foo() { }
06.     Foo(int) { }
07.     void fun() { }
08. };
09. int main(void)
10. {
11.     Foo a(10);    //语句1
12.     a.fun();      //语句2
13.     Foo b();      //语句3
14.     b.fun();      //语句4
15.     return 0;
16. }
```

语句 1

语句 2

语句 3

语句 4

解析:

语句 3 应该为 `Foo b`, 但语句 3 没出错语句 4 出错了, 出错提示 `request for member 'fun' in 'b', which is of non-class type 'Foo()'`。感觉似乎语句 3 被解释成函数声明了, `b` 成了函数指针, 没有成员 `fun()`。不知道这样理解对不对。。。 。。。

`Foo b();` 声明了返回类型为 `Foo`, 参数为空的函数。因此导致 `b.fun()` 编译错误。

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 ()

正确答案: B 你的答案: D (错误)

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

解析：

答案：B

可以看出 fun 函数的形参是一个指针数组，也就是指针指向一个地址，地址中存放的内容也是指针。

A，二维数组，不符合

B，二级指针，也就是指针指向的内容也还是存放指针的，符合

C，二级指针数组，数组的内容是二级指针，不符合

D，数组指针，不符合

BSS（Block Started by Symbol）通常是指用来存放程序中未初始化的全局变量和静态变量的一块内存区域。特点是：可读写的，在程序执行之前 BSS 段会自动清 0。所以，未初始化的全局变量在程序执行之前已经成 0 了。

数据段：数据段（data segment）通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配。

代码段：代码段（code segment/text segment）通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读，某些架构也允许代码段为可写，即允许修改程序。在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆（heap）：堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减。当进程调用 malloc 等函数分配内存时，新分配的内存就被动态添加到堆上（堆被扩张）；当利用 free 等函数释放内存时，被释放的内存从堆中被剔除（堆被缩减）

栈(stack)：栈又称堆栈，是用户存放程序临时创建的局部变量，也就是说我们函数括弧“{}”中定义的变量（但不包括 static 声明的变量，static 意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场。从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区

头文件已经正常包含，以下代码在 VS IDE 上编译和运行结果是

```
1  class A{
2      public:
3          void test() {printf("test A");}
4  };
5  int main() {
6      A*pA=NULL;
7      pA->test();
8  }
```

正确答案: C 你的答案: A (错误)

编译出错

程序运行崩溃

输出"test A"

输出乱码

解析:

根据<<深度探索 C++对象模型>>里面说到的, 普通成员函数在编译器实现时, 大致经历一下转化过程:

1. 改写函数的签名以安插一个额外的参数 - **this** 指针
2. 将每一个对非静态成员数据成员的存取操作改为经由 **this** 指针来存取。
3. 将成员函数重写成外部函数, 函数名称经过“mangling”处理, 使其在程序中有唯一的名字。

(重写成外部函数, 是希望在实现成员函数的时候, 使其效率接近普通非成员函数, 尽可能避免带来额外开销)

A*pA=null;

pA->test(); //当调用成员函数时, 只是将实参 null 传给 **this** 指针 (对应 1.)

test 成员函数中并无任何需要通过 **this** 指针访问的数据成员 (对应 2.), 因此没有带来任何影响

第一, 这道题目应该是写法有点问题, 应该是这样的 (注意类后面加了分号, **test** 后面加了括号)

```
1  #include<iostream>
2  using namespace std;
3  class A{
4  public:
5      void test()
6      { printf("test A"); }
7  };
8  int main() {
9      A* pA = NULL;
10     pA->test();
11     return 0;
12 }
```

其实这个是可以正常运行的

原因如下:

因为对于非虚成员函数，C++这门语言是静态绑定的。这也是C++语言和其它语言 Java, Python 的一个显著区别。以此下面的语句为例：

```
1 pA->test();
```

这句话的意图是：调用对象 pA 的 test 成员函数。如果这句话在 Java 或 Python 等动态绑定的语言之中，编译器生成的代码大概是：

找到 pA 的 test 成员函数，调用它。（注意，这里的找到是程序运行的时候才找的，这也是所谓动态绑定的含义：运行时才绑定这个函数名与其对应的实际代码。有些地方也称这种机制为迟绑定，晚绑定。）

但是对于 C++。为了保证程序的运行时效率，C++的设计者认为凡是编译时能确定的事情，就不要拖到运行时再查找了。所以 C++的编译器看到这句话会这么干：

1：查找 pA 的类型，发现它有一个非虚的成员函数叫 test。（编译器干的）

2：找到了，在这里生成一个函数调用，直接调 A:: test (pA)。

所以到了运行时，由于 test ()函数里面并没有任何需要解引用 pA 指针的代码，所以真实情况下也不会引发 segment fault。这里对成员函数的解析，和查找其对应的代码的工作都是在编译阶段完成而非运行时完成的，这就是所谓的静态绑定，也叫早绑定。

正确理解 C++的静态绑定可以理解一些特殊情况下 C++的行为。

有如下一段程序：

```
1 int f1(float);
2 int f2(char);
3 int f3(float);
4 int f4(float);
5 int (*pf)(float);
```

则以下不合法的是()

正确答案: C 你的答案: D (错误)

```
int (*p)(float)=&f1;
pf=&f4;
pf=&f2;
pf=f3;
```

解析：

知识点：

- 函数指针变量：

函数指针变量的声明方法为：

返回值类型 (* [指针变量名](#)) ([形参列表](#));

根据定义，

```
int(*pf)(float);
```

```
int (*p)(float)=&f1;
```

pf,p 都是函数指针变量。

- 函数地址

C 在编译时，每一个函数都有一个入口地址，该入口地址就是函数指针所指向的地址。

函数地址的获取，可以是函数名，也可以在函数名前加取地址符&

C 错误是因为函数行参类型不匹配。

函数名加不加取地址符&都是正确的。BD 都对

下面有关函数模板和类模板的说法正确的有？

正确答案: A B C D 你的答案: B (错误)

函数模板的实例化是由编译程序在处理函数调用时自动完成的

类模板的实例化必须由程序员在程序中显式地指定

函数模板针对仅参数类型不同的函数

类模板针对仅数据成员和成员函数类型不同的类

解析：

让我们从故事中找到答案：

首先揭开谜底；满堂彩 A B C D

1、概述：

模板（Template）是一种强大的 C++ 软件复用特性，通常有两种形式：函数模板和类模板。函数模板针对仅参数类型不同的函数（答案 C ok）；类模板针对仅数据成员和成员函数类型不同的类（答案 D ok）。函数模板和类模板可以是程序员只需制定一个单独的代码段，就可表示一整套称为函数模板特化的相关（重载）函数或是表示一整套称为类模板特化的相关的类。这种技术称为泛型程序设计（generic programming）。使用模板的好处在于，可以使程序员编写与类型无关的代码。

函数模板与类模板的区别

模板是一个类家族的抽象，它只是对类的描述，编译程序不为类模板(包括成员函数定义)创建程序代码，但是通过对类模板的实例化可以生成一个具体的类以及该具体类的对象。

与函数模板不同的是：函数模板的实例化是由编译程序在处理函数调用时自动完成的（答案 A 正确），而类模板的实例化必须由程序员在程序中显式地指定（答案 B 正确）

std::vector::iterator 重载了下面哪些运算符？

正确答案: A C D 你的答案: C (错误)

++

>>

*（前置）

==

解析：

选择 ACD

++，--用于双向迭代，迭代器最基本的功能；

*用于复引用迭代器用于引用迭代器对应的元素，也是基本操作；

==用于判断两个迭代器是否相等，迭代的时候需要判断迭代器是否到了某个位置；

有以下程序

```
1  #include <stdio.h>
2  void fun ( int n ,int *s ) {
3      int f;
4      if(n==1)
5          *s = n+1
6      else {
7          fun( n-1, &f) ;
8          *s = f ;
9      }
10 }
11 main() {
12     int x =0;
13     fun( 4,&x );
14     printf("%d\n",x);
15 }
```

程序的输出结果是？

正确答案：C

3

1

2

4

共 4 次调用 fun

fun (4, &x) -> fun(3, &f) ->fun(2, &f) ->fun(1, &f)

fun(1, &f)返回后，使得 fun (2, &f) 中的 f 变为 2

依次向上递推，

fun (4, &x) 中的 f 也为 2

然后, *s = f = 2

x 的值为 2

发表于 2016-08-26 15:24:49

下面有关 c++ 静态数据成员, 说法正确的是?

正确答案: D 你的答案: A (错误)

- 不能在类内初始化
- 不能被类的对象调用
- 不能受 private 修饰符的作用
- 可以直接用类名调用

通常静态数据成员在类声明中声明, 在包含类方法的文件中初始化.
初始化时使用作用域操作符来指出静态成员所属的类.

但如果静态成员是整型或是枚举型 const, 则可以在类声明中初始化!!

如果改成有的静态数据成员是可以直接在类中初始化就对了

程序的完整编译过程分为是: 预处理, 编译, 汇编等, 如下关于编译阶段的编译优化的说法中不正确的是 () ?

正确答案: A 你的答案: A (正确)

- 死代码删除指的是编译过程直接抛弃掉被注释的代码
- 函数内联可以避免函数调用中压栈和退栈的开销
- For 循环的循环控制变量通常很适合调度到寄存器访问
- 强度削弱是指执行时间较短的指令等价的替代执行时间较长的指令

解析:

答案: 选 A

死代码的含义是指永远不会被执行到的代码段, 而不是直接抛弃被注释的代码

比如 while(false){}

继承中的父类的私有变量是在子类中存在的, 不能访问是编译器的行为, 是可以通过指针操作内存来访问的。具体例子如下, 例子中为了便于打印变量, 将变量更改为了 int 类型:

```
1    #include <iostream>
2
3    class A
```

```

4      {
5      public:
6          A() : p1(1), p2(2), p3(3) {}
7      public:
8          int p1;
9      private:
10         int p2;
11     protected:
12         int p3;
13     };
14
15     class B: public A {};
16
17     int main()
18     {
19         A a;
20         std::cout << *(int *)&a << std::endl;
21         std::cout << *((int *)&a + 1) << std::endl;
22         std::cout << *((int *)&a + 2) << std::endl;
23         B b;
24         std::cout << *(int *)&b << std::endl;
25         std::cout << *((int *)&b + 1) << std::endl;
26         std::cout << *((int *)&b + 2) << std::endl;
27     }

```

编译运行如下程序会出现什么结果

```

1  #include <iostream>
2  using namespace std;
3
4  class A
5  {
6      A()
7      {
8          printf("A()");
9      }
10 public:
11     static A &get()
12     {
13         static A a;
14         return a;
15     }
16 };
17 int main()
18 {

```

```
19         A::get();
20         return 0;
21     }
```

正确答案: A 你的答案: A (正确)

输出 A()

编译错误

链接错误

以上都不对

解析:

答案 A

解释: 调用静态函数本身不会执行构造函数, 但 `get ()`实例化了一个对象, 所以在 `get ()` 里面调用了构造

如下程序

```
1         #include "stdio.h"
2
3     class A
4     {
5     public:
6         virtual void Test()
7         {
8             printf("A test\n");
9         }
10    };
11    class B: public A
12    {
13    public:
14        void func()
15        {
16            Test();
17        }
18        virtual void Test()
19        {
20            printf("B test\n");
21        }
22    };
23    class C: public B
24    {
25    public:
26        virtual void Test()
27        {
28            printf("C test\n");
29        }
30    };
```

```

31 void main()
32 {
33     C c;
34     ((B *)(&c))->func();
35     ((B)c).func();
36 }

```

该程序的执行结果

正确答案: A 你的答案: A (正确)

C test

B test

B test

B test

B test

C test

A test

C test

解析:

答案 A

解释:

`((B *)(&c))->func();` 是多态行为

`((B)c).func();` 不是多态行为。

这道题目可以这么去拆分就很好理解了:

`((B *)(&c))->func()` ==》 `B *temp; temp = &c; temp->func();` ==》 这不就是一个典型的多态问题么, 用基类指针指向派生类对象, 所以肯定调用的是 C 对象的 func 函数, 输出 C test

`((B)c).func();` ==》 不涉及指针的操作, 自然就没有多态行为的发生。

解析 2:

答案选 A, 推荐的解析我觉得不对, 应该是这么分析: `((B *)(&c))->func();` 此句调用 func, 先分析 func 函数, 该函数不是虚函数, 自然不会是多态问题, 所以不要往多态方面分析。指针是指向类 C 的对象 c, 此时尽管加了 (B *), 可是依然指向的是 c 的首地址, 因此调用的是 c 中的 func 函数, 而 func 中 Test() 之前省略了 this 指针, 此时 this 当然是对象 c, 那也只能调用 c 中的成员函数 Test(), 所以输出 C test。

而 `((B)c).func();` 此代码是先建立类 B 的对象 c, 然后调用 c 的成员函数 func(), 当然 func 中的 Test() 是属于类 B 的对象 c 的成员函数, 自然会输出 B test。

代码生成阶段的主要任务是：

正确答案: C 你的答案: C (正确)

把高级语言翻译成机器语言

把高级语言翻译成汇编语言

把中间代码变换成依赖具体机器的目标代码

把汇编语言翻译成机器语言

解析：

源码 -> (扫描) -> 标记 -> (语法分析) -> 语法树 -> (语义分析) -> 标识语义后的语法树 -> (源码优化) -> 中间代码 -> (代码生成) -> 目标机器代码 -> (目标代码优化) -> 最终目标代码

不考虑任何编译器优化(如:NRVO),下述代码的第 10 行会发生

```
1 #include <stdio.h>//1
2 class B//2
3 {//3
4 };//4
5 B func(const B& rhs) {//5
6     return rhs;//6
7 }//7
8 int main(int argc, char **argv) {//8
9     B b1, b2;//9
10    b2=func(b1); //10
11} //11
```

正确答案: D 你的答案: B (错误)

一次默认构造函数, 一次拷贝构造函数, 一次析构函数, 一次 (拷贝赋值运算符) operator=

二次拷贝构造函数, 一次析构函数

一次 (拷贝赋值运算符) operator=, 一次析构函数

一次拷贝构造函数, 一次析构函数, 一次 (拷贝赋值运算符) operator=

解析：

b2=func(b1); //10

一次拷贝构造函数发生在 func 函数调用完成, 返回 B 类型的对象时, 因为返回的不是引用类型, 所以会生成一个对象,

不妨称为 TEMP, 将返回的对象通过拷贝构造函数复制给 TEMP, 然后, 返回值所对应的对象会被析构。如果返回值是引用类型,

则不会调用拷贝构造函数。

赋值运算符在 func 函数执行完成后，将上面提到的 TEMP，通过赋值运算符赋值给 b2,值得注意的是赋值运算符重载函数如果不自己定义，程序会认为是调用缺省的赋值运算符重载函数。

在 linux 编程中，以下哪个 TCP 的套接字选项与 nagle 算法的开启和关闭有关？

正确答案: B 你的答案: B (正确)

TCP_MAXSEG
TCP_NODELAY
TCP_SYNCNT
TCP_KEEPAALIVE

解析：

当有一个 TCP 数据段不足 MSS，比如要发送 700Byte 数据，MSS 为 1460Byte 的情况。nagle 算法会延迟这个数据段的发送，等待，直到有足够的填充成一个完整数据段。也许有人会问，这有什么影响呢？没有太大的影响，总体上来说，这种措施能节省不必要的资源消耗。但是要发送的总体数据很小时，这种措施就是拖后腿了。比如，用户请求一个网页，大约十几 KB 的数据，TCP 先发送了八九个数据包，剩下几百字节一直不发送，要等到另一个 RTT 才发送，这时候前面发送数据的 ACK 已经返回了。这样的用户体验是很不好的。所以，现在很多服务器都选择主动关闭 nagle 算法，因为带宽够大，资源消耗不是问题，速度反而是个大问题。

从上述描述中，禁用 nagle，实质就是不在延迟 TCP_NODELAY

执行下面语句：

```
1  int countx=0,x=8421;
2  while(x) {
3      countx++;
4      x=x&(x-1);
5  }
```

当程序跳出循环后，countx 的值为 6

解析：

这题很热呀，

$x \& (x-1)$ 代表什么意思呢？ 每执行一次 $x = x \& (x-1)$ ，会将 x 用二进制表示时最右边的一个 1 变为 0，因为 $x-1$ 将会将该位(x 用二进制表示时最右边的一个 1)变为 0。实际就是看把十进制转换成二进制，里面有几个 1

函数的作用是统计 x 二进制数中 1 的个数。

这个作用是对整型中 1 的个数进行统计， $x = x \& (x-1)$ 的作用是每次循环把 x 的二进制中从右往左数的最后一位 1 变成 0，直到变成全 0 为止，循环结束。

8421 的二进制是 0010 0000 1110 0101，所以结果是 6

还有类似的题目，迭代式为 $x = x | (x+1)$ ，作用是统计 x 二进制中 0 的个数

拷贝构造函数 MyClass(const MyClass & x);

赋值运算符 MyClass operator= (const MyClass & x);
析构函数 ~MyClass();

编译运行如下程序会出现什么结果

```
1  #include <stdio.h>
2  class A
3  {
4      A()
5      {
6          printf("A()");
7      }
8  };
9  void main()
10 {
11     A a;
12 }
```

则输出

正确答案: B 你的答案: A (错误)

A()

编译错误

链接错误

以上都不对

解析:

答案: B

编译出错, 因为构造方法是私有的, 不能被调用。这种情况下不能创建对象。

单例模式会把构造方法声明为私有的, 但是会提供一个 **public** 的静态方法, 用来获取对象实例。

关于"while(条件表达式)循环体", 以下叙述正确的是?(假设循环体里面没有 break)

正确答案: B 你的答案: D (错误)

循环体的执行次数总是比条件表达式的执行次数多一次

条件表达式的执行次数总是比循环体的执行次数多一次

条件表达式的执行次数与循环体的执行次数一样

条件表达式的执行次数与循环体的执行次数无关

解析:

题目定位: C 语言基础

注意 2 种形式的 while 循环:

(1) **while(条件表达式) {循环体}**: 先判断条件,满足条件后执行循环体,执行完后接着判断条件,执行,直到最后一次判断条件后不成立,跳出循环。条件表达式至少会执行一次,这种情况下,条件表达式的执行次数总是会比循环体的执行次数多一次。

(2) **do {循环体} while(条件表达式)**: 先执行循环体,再判断条件,若条件满足则进入循环体进行执行,否则跳出循环。所以,循环体至少会执行一次。这种情况下,循环体的执行次数总是会比条件表达式的执行次数多一次。

请大家注意区分。

有以下程序段:

```
1 char *p, *q;
2 p = (char *)malloc(sizeof(char) * 20);
3 q = p;
4 scanf( "%s %s", p, q);
5 printf( "%s %s\n", p, q);
```

若从键盘输入: **abc def**✓, 则输出结果是 ()

正确答案: A 你的答案: A (正确)

```
def def
abc def
abc d
d d
```

解析:

解释: p q 指向同样的地址,

scanf("%s %s", p, q);"abc"赋值给 p,然后"def"赋值给 q (即 p),把之前的赋值覆盖了。

将逻辑代码:

```
1 if (x % 2) {
2     return x - 1;
3 } else {
4     return x;
5 }
```

用表达式: **return x & -2;** 替代, 以下说法中不正确的是 ()

正确答案: C 你的答案: A (错误)

计算机的补码表示使得两段代码等价

用第二段代码执行起来会更快一些

这段代码只适用于 x 为正数的情况

第一段代码更适合阅读

解析:

正数的补码就是其本身

负数的补码是在其原码的基础上, 符号位不变, 其余各位取反, 最后+1,

以 8 位字长为例 (64 位也一样)

-2 = (1000 0010) 原 = (1111 1101) 反 = (1111 1110) 补

源代码表示末位不为 1 时将会减 1, 否则返回源码, 这与 -2 的补码相与的结果是相同的。

设变量已正确定义, 以下不能统计出一行中输入字符个数 (不包含回车符) 的程序段是
正确答案: D

```
n=0;while(ch=getchar()!='\n')n++;
n=0;while(getchar()!='\n')n++;
for(n=0;getchar()!='\n';n++);
n=0;for(ch=getchar();ch!='\n';n++);
```

解析:

D 只获取了一次字符。

【特别注意】指针常量和常量指针区别

下面说法哪些正确:

正确答案: A B C 你的答案: B C D (错误)

```
const int a; // a 是常数
int const a; // a 是常数
int const *a; // a 是常量指针
const int *a; // a 是指针常量
int const *a; // a 是指针常量
```

解析:

答案: A B C

A, B, const int a; int const a; 这两个写法是等同的, 表示 a 是一个 int 常量。

C, D, E, const int *a; 表示 a 是一个指针, 可以任意指向 int 常量或者 int 变量, 它总是把它所指向的目标当作一个 int 常量。也可以写成 int const* a; 含义相同。

正确答案 AB

- const int a; // a 是常数
- int const a; // a 是常数

题目中 C 其实写错了，应该是 `int * const a;` // a 是常量指针。判断时不要记什么“const 左效，最左右效”之类的规则，要理解。从右往左读，离 a 最近的是 const，意味着 a 本身是一个常量对象，对象的类型由声明符的其余部分确定。声明符下一个符号是 *，是 int 指针，所以 a 是 const pointer.

这个规律只在指针的情况下使用

另外，要注意 `const int *a;` 和 `int const *a;`，用同样的方法可以判定是 pointer to const（指向常量的指针）。

在下列排序算法中，哪几个算法的时间复杂度与初始排序无关()

正确答案: B D E 你的答案: D E (错误)

插入排序
堆排序
冒泡排序
归并排序
选择排序

解析：

自己总结的口诀：选快希堆不稳（是不稳定的排序），

堆归选基均不变（运行时间不发生变化，与初始状态无关）

9

运算 $(93 \& -8)$ 的结果为 1

你的答案 (错误)

1 8

参考答案 88

解析：

93 的二进制 为 01011101

-8 的原码为 10001000

-8 的补码为 1111000

取与为 01011000 为 88

有如下语句序列：

`char str[10]; cin>>str;`

当从键盘输入“I love this game”时，str 中的字符串是

正确答案: D

"I love this game"

"I love thi"

"I love"

"I"

解析：

cin>>

该操作符是根据后面变量的类型读取数据。

输入结束条件：遇到 Enter、Space、Tab 键。

对结束符的处理：丢弃缓冲区中使得输入结束的结束符(Enter、Space、Tab)

写出下列代码的输出内容（）

```
1  #include<stdio.h>
2  int inc(int a)
3  {
4      return (++a);
5  }
6  int multi(int *a, int *b, int *c)
7  {
8      return (*c=*a* *b);
9  }
10 typedef int (FUNC1)(int in);
11 typedef int (FUNC2)(int*, int*, int*);
12 void show(FUNC2 fun, int arg1, int *arg2)
13 {
14     FUNC1 p=&inc;
15     int temp=p(arg1);
16     fun(&temp, &arg1, arg2);
17     printf("%d\n", *arg2);
18 }
19 int main()
20 {
21     int a;
22     show(multi, 10, &a);
23     return 0;
24 }
```

正确答案: B

100

110

120

0

解析：

typedef int (FUNC1)(int); 是函数指针定义
show(multi,10,&a); FUNC2 类型函数指针 fun 指向函数 multi 的首地址
FUNC1 p=&inc; FUNC1 类型 函数指针 p 指向函数 inc 的首地址
int temp=p(arg1); 此时调用函数 inc, 参数为 10, 返回值为 11
fun(&temp,&arg1,arg2); 调用函数 multi, 参数为 (11,10, arg2) arg2 为指针变量负责
带回返回值
printf("%d\n",*arg2); 输出 110

派生类强制转化成基类 (dynamic_cast) 会做相应成员变量和函数的裁剪, 仅能访问从基类继承来的部分成员

在 C++ 中,

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

输出是多少?

正确答案: A 你的答案: B (错误)

0, 1

1, 1

1, 0

0, 0

解析:

这个题一定要注意是在 C++ 中的运行结果。

在 C 语言中

```
1  void main() {  
2      const int i = 0;  
3      int *j = (int *)&i;  
4      *j = 1;  
5      printf("%d,%d", i, *j);  
6      system("pause");  
7  }
```

结果输出为 1,1

在 C++ 中

```
1  #include<iostream>  
2  using namespace std;  
3  int main(void) {  
4      const int i=0;  
5      int *j = (int *)&i;  
6      *j = 1;  
7      printf("%d,%d", i, *j);  
8      system("pause");
```

```

9         return 0;
10    }

```

结果输出为 0,1

分析：C 语言中的 `const` 是运行时 `const`，编译时只是定义，在运行才会初始化。C 语言中 `const` 变量不能用于成为数组长度等作为编译时常量的情况，原因就在此。C 语言 `const` 变量在运行时改变了是可以再次读出改变后的值的。

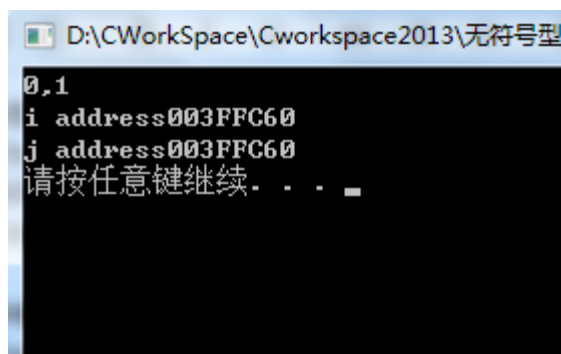
C++中，`const` 变量是编译时的常量，可以向 `#define` 定义的常量一样使用。故 C++中 `const` 变量的值在编译时就已经确定了，直接对 `const` 变量进行了值的替换，因此当 `const` 变量的值改变时，`const` 的变量值是得不到更新的。

这几行代码在 C 和 C++中都会改变 `const` 变量的值，只不过 C++中 `const` 变量在编译的时候已经确定了，而 C 中的 `const` 变量可以在运行时改变后再次读取。以下代码核实了 C++中的 `const` 变量确实被改变了。

```

1  #include<stdio.h>
2  #include<iostream>
3  using namespace std;
4  int main(void) {
5      const int i=0;
6      int *j = (int *)&i;
7      *j = 1;
8
9      printf("%d,%d\n", i, *j);
10     cout << "i address"<<&i << endl;
11     cout << "j address"<<j << endl;
12
13     return 0;
14 }

```



```

D:\CWorkspace\Cworkspace2013\无符号型
0,1
i address003FFC60
j address003FFC60
请按任意键继续. . .

```

同一个地址，即同一个变量。C++中 `const` 变量确实被改变了。i 的值没有更新而已。

下面说法正确的是()

- A. 一个空类默认一定生成构造函数, 拷贝构造函数, 赋值操作符, 引用操作符, 析构函数
- B. 可以有多个析构函数
- C. 析构函数可以为 `virtual`, 可以被重载
- D. 类的构造函数如果都不是 `public` 访问属性, 则类的实例无法创建

D 选项，单例模式

```
1. class CSingleton
2. {
3. private :
4.     CSingleton() //构造函数是私有的
5.     {
6.     }
7.     static CSingleton *m_pInstance;
8. public :
9.     static CSingleton * GetInstance()
10.    {
11.        if (m_pInstance == NULL) //判断是否第一次调用
12.            m_pInstance = new CSingleton();
13.        return m_pInstance;
14.    }
15.};
```

16. 用户访问唯一实例的方法只有 GetInstance()成员函数。如果不通过这个函数，任何创建实例的尝试都将失败，因为类的构造函数是私有的。GetInstance()使用 懒惰初始化，也就是说它的返回值是当这个函数首次被访问时被创建的。这是一种防弹设计——所有 GetInstance()之后的调用都返回相同实例的指针：

17. <http://blog.csdn.net/hackbuteer1/article/details/7460019>

What is the result of the following program?

```
1  char* f(char *str, char ch) {
2      char *it1 = str;
3      char *it2 = str;
4      while (*it2 != '\0') {
5          while (*it2 == ch) { it2++; }
6          *it1++ = *it2++;
7      }
8      return str;
9  }
10 void main(int argc, char *argv[]) {
11     char *a = new char[10];
12     strcpy(a, "abcdcccd");
13     cout << f(a, 'c');
14 }
```

正确答案: D 你的答案: B (错误)

abdcccd

abdd


```
abcc
abddcccd
Access Violation
```

解析:

最开始这道题我也选的 B 但是仔细分析后发现答案是 D 的。看下面这段代码:

```
while(*it2 != '\0')
{
    while(*it2 == ch) { it2++; }
    *it1++ = *it2++;
}
```

it1 的前两个字符为 ab 没有异议, 当 it2 的指针指向 c 时执行 it2++, 运行后 it2 指向 d, 然后下一个字母不为 c, 所以 it1 的指针内复制为 d, 即此时 it1 为 abd, 之后遇到 3 个 c, 执行 it2++, 直到 it2 指向 d 时才将 d 赋值给 it1, 也就是此时 it1=abdd, 但是接下来 it2 已经为空了, 也就是“\0”, 所以就不执行循环了, 但是 it1 内本身后面还有 cccd, 前面的四个字母只是 it2 覆盖成 abdd, 所以最终的答案是 abddcccd 也就是 D

以下有关 C 语言的说法中, 错误的是_____。

正确答案: D 你的答案: D (正确)

内存泄露一般是指程序申请了一块内存, 使用完后, 没有及时将这块内存释放, 从而导致程序占用大量内存。

无法通过 malloc(size_t) 函数调用申请超过该机器物理内存大小的内存块。

无法通过内存释放函数 free(void*) 直接将某块已经使用完的物理内存直接还给操作。

可以通过内存分配函数 malloc(size_t) 直接申请物理内存。

解析:

D

内存泄漏也称作“存储渗漏”, 用动态存储分配函数动态开辟的空间, 在使用完毕后未释放, 结果导致一直占据该内存单元。直到程序结束。(其实说白了就是该内存空间使用完毕之后未回收) 即所谓内存泄漏。

free 释放的内存不一定直接还给操作系统, 可能要到进程结束才释放。

可以直到 malloc 不能直接申请物理内存, 它申请的是虚拟内存

下面这段代码运行时会出现什么问题?

```
1  class A
2  {
3  public:
4      void f()
5      {
```

```

6             printf("A\n");
7         }
8     };
9
10    class B: public A
11    {
12    public:
13        virtual void f()
14        {
15            printf("B\n");
16        }
17    };
18
19    int main()
20    {
21        A *a = new B;
22        a->f();
23        delete a;
24        return 0;
25    }
26
27

```

正确答案: B 你的答案: C (错误)

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

解析:

基类 A 的 f() 没有声明为虚函数，所以没有 B 覆盖

下面有关类的静态成员和非静态成员，说法错误的是？

正确答案: C 你的答案: B (错误)

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

解析:

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

#include 命令的功能是 ()。

正确答案: C 你的答案: A (错误)

在命令处插入一个头文件

在文件首部插入一个头文件

在命令处插入一个文本文件

在文件首部插入一个程序文件

解析:

预处理器发现**#include** 后, 就会寻找指令后面<>中的文件名, 并把这个文件的内容包含到当前的文件中, 被包含的文件中的文本将替换源代码文件中的**#include** 指令

下面程序的输出结果是

```
1 char *p1= "123", *p2 = "ABC", str [50] = "xyz";
2 strcpy (str+2, strcat (p1,p2));
3 cout << str;
```

正确答案: D 你的答案: D (正确)

xyz123ABC

z123ABC

xy123ABC

出错

解析:

分析: p1 和 p2 指向的是常量存储区的字符串常量, 没法连接, 会有问题

下面代码输出什么

```
1 #include<stdio.h>
2 int main( )
3 {
4     unsigned int a = 6;
5     int b = -20;
6     (a + b > 6) ? printf(">6") : printf("<=6");
7     return 0;
8 }
```

输出结果是 1

你的答案 (错误)

1 <=6

参考答案 >6

解析:

答案 >6

解释: $a+b$, b 需要转换成 unsigned

C++primer 第四版说, C++ 允许将负数赋给 unsigned 类型。所以把负值赋给 unsigned 对象的时候, 其结果是该负数对该类型的取值个数求模后的值。所以本题 $a+b$, 将 b 转换为 unsigned, $= -20/2$ 的 32 次方 =

下列关于内存分配和释放的函数及其区别描述正确的有?

正确答案: C D 你的答案: C (错误)

C++ 语言的标准内存分配函数: malloc, calloc, realloc, free 等。

C 中为 new/delete 函数。

malloc 和 calloc 的区别是 1 块与 n 块的区别和初始化

realloc 调用形式为 (类型*) realloc(*ptr, size): 将 ptr 内存大小增大到 size。

解析:

1. malloc, calloc, realloc, free 属于 C 函数库, 而 new/delete 则是 C++ 函数库;

2. 多个 -alloc 的比较:

 alloc: 唯一在栈上申请内存的, 无需释放;

 malloc: 在堆上申请内存, 最常用;

 calloc: malloc+初始化为 0;

 realloc: 将原本申请的内存区域扩容, 参数 size 大小即为扩容后大小, 因此此函数要求 size 大小必须大于 ptr 内存大小。

用 C++ 语法来看, 下列的哪个赋值语句是正确的?

正确答案: A B C D 你的答案: A D (错误)

```
char a=12
int a=12.0
int a=12.0f
int a=(int) 12.0
```

解析:

从 C/C++ 的角度去分析此题, A、B、C、D 四项在编译的过程中: A 项在赋值为 0-127 时无警告, 但 B/C/D 均有损失精度的警告, 编译还是能通过的。B、C、D 项的区别是: B 项的默认为 double 型转向 int 型, C 项因等号后“12.0f”的 ‘f’ 使其转换为 float 型, D 项直接通过在等号的右边加“(int)”强制类型转换, 与 B 项相同的是: 是从默认的双精度型转换为 int 型。A 项会对应的 ASCII 字符, 但在大于 127 时会发生常量截断会有警告。所以综上, 此题 A、B、C、D 是都正确的

关于中间件特点的描述,不正确的是 ()

正确答案: A 你的答案: A (正确)

中间件运行于客户机/服务器的操作系统内核中, 提高内核运行效率

中间件应支持标准的协议和接口

中间件可运行于多种硬件和操作系统平台上

跨越网络, 硬件, 操作系统平台的应用或服务可通过中间件透明交互

解析:

中间件位于操作系统之上, 应用软件之下, 而不是操作系统内核中

声明纯虚函数的类是抽象类, 不能实例化

基类被虚继承才是虚基类

下面哪种 C/C++ 分配内存的方法会将分配的空间初始化为 0

正确答案: B 你的答案: D (错误)

malloc()
calloc()
realloc()
new[]

解析:

1) malloc 函数: void *malloc(unsigned int size)

在内存的动态分配区域中分配一个长度为 size 的连续空间, 如果分配成功, 则返回所分配内存空间的首地址, 否则返回 NULL, 申请的内存不会进行初始化。

2) calloc 函数: void *calloc(unsigned int num, unsigned int size)

按照所给的数据个数和数据类型所占字节数, 分配一个 num * size 连续的空间。
calloc 申请内存空间后, 会自动初始化内存空间为 0, 但是 malloc 不会进行初始化, 其内存空间存储的是一些随机数据。

3) realloc 函数: void *realloc(void *ptr, unsigned int size)

动态分配一个长度为 size 的内存空间, 并把内存空间的首地址赋值给 ptr, 把 ptr 内存空间调整为 size。

申请的内存空间不会进行初始化。

4) new 是动态分配内存的运算符, 自动计算需要分配的空间, 在分配类类型的内存空间时, 同时调用类的构造函数, 对内存空间进行初始化, 即完成类的初始化工作。动态分配

内置类型是否自动初始化取决于变量定义的位置，在函数体外定义的变量都初始化为 0，在函数体内定义的内置类型变量都不进行初始化。

以下代码的输出结果是？

```
1  #define a 10
2
3  void foo();
4  main() {
5
6      printf("%d..", a);
7      foo();
8      printf("%d", a);
9  }
10 void foo() {
11     #undef a
12     #define a 50
13 }
```

正确答案: A 你的答案: B (错误)

10..10
10..50
Error
0

解析：

```
1  #define a 10
2  void foo();
3  void prin();
4
5  int main()
6  {
7      prin();
8      printf("%d ", a);
9      foo();
10     printf("%d ", a);
11
12 }
13 void foo()
14 {
15     #undef a
16     #define a 50
```

```

17 }
18 void prin()
19 {
20     printf("%d ", a);
21 }

```

上面代码输出 50 10 10，可以看出 `define` 只是在预处理阶段将 `a` 替换为相应数值，具体替换的值只与 `define` 在文件中的位置有关，与是否在函数内无关。

下面关于一个类的静态成员描述中,不正确的是()

正确答案: C 你的答案: D (错误)

静态成员变量可被该类的所有方法访问
 该类的静态方法只能访问该类的静态成员函数
 该类的静态数据成员变量的值不可修改
 子类可以访问父类的静态成员
 静态成员无多态特性

解析：

类的静态成员属于整个类 而不是某个对象，可以被类的所有方法访问，子类当然可以父类静态成员；

静态方法属于整个类，在对象创建之前就已经分配空间，类的非静态成员要在对象创建后才有内存，所有静态方法只能访问静态成员，不能访问非静态成员；
 静态成员可以被任一对象修改，修改后的值可以被所有对象共享。

Which of following C++ code is correct?

正确答案: C 你的答案: C (正确)

```

int f() { int *a = new int(3); return *a; }
int *f() { int a[3] = {1, 2, 3}; return a; }
vector<int> f() {vector<int> v(3); return v; }
void f(int *ret) { int a[3] = {1, 2, 3}; ret = a; return; }

```

解析：

A: 和大家说的一样，`new` 申请的内存存在离开作用域时由于没有 `delete`，导致内存泄漏。

（这块内存对于系统来说是被占用的，但是实际上已经没有人使用了，所以这块内存永远无法释放！有兴趣的可以看看 `malloc` 的源码分析）

B: 数组不能复制和拷贝

C: `vector` 容器可以复制或拷贝

D: 同 B

当函数参数实例化形参或者函数返回值时，一般调用对象的拷贝构造函数！

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  {
27      A a;
28      B b;
29      A *p = &a;
30      p->foo();
31      p->fun();
32      p = &b;
33      p->foo();
34      p->fun();
35      A *ptr = (A *)&b;
36      ptr->foo();
37          ptr->fun();
38      return 0;
39  }
40
41
```


正确答案: B 你的答案: B (正确)

121434
121414
121232
123434

解析:

当定义基类指针指向子类对象时, 通过这个指针调用子类和基类的同名函数时, 基类定义为虚函数的话, 就调用子类的这个函数, 否则调用基类的这个函数

以下代码打印的结果是 (假设运行在 i386 系列计算机上):

```
1 struct st_t
2 {
3     int status;
4     short *pdata;
5     char errstr[32];
6 };
7
8 st_t st[16];
9 char *p = (char *) ( st[2].errstr + 32 );
10 printf("%d", ( p - (char *) (st) ) );
```

正确答案: C 你的答案: C (正确)

32
114
120
1112

解析: s

sizeof(st_t)是 40

st [2]. errstr + 32 相当于 st[3]的地址

(p - (char *) (st)) 是 st[3]的地址减去 st 的首地址。

```
1 char *p1;int64 *p2;
2 p1=(char *)0x800000;
3 p2=(int64 *)0x800000;
4 char *a=p1+2
5 int64_t *b=p2+2
```

那么 a= 1 ,b= 2

你的答案 (错误)

1 参考答案 0x800002

2 参考答案 0x800010

解析:

答: 0x800002, 0x800010

我们定义指针的时候给指针一个类型就是为了方便指针的加减操作。p1 是 char 类型指针, 每个 char 占一个字节, 所以 p1+2 就是在 p1 的基础上加 2 个 char 的长度, 就是两个字节。p2 是指向 64 位 int 型的指针, 所以 p2+2 就是 p2 加上两个 64 位 int 的长度, 也就是加上 128 位, 即 16 个字节。用 16 进制表示是 0x10

所以 a=0x800000+0x2=0x800002

a=0x800000+0x10=0x800010

在 C++STL 中常用的容器和类型, 下面哪些支持下标"[]"运算?

正确答案: A C D F I

vector

list

deque

map

set

unordered_map

unordered_set

stack

string

解析:

支持随机访问就支持[]

vector: 随机访问迭代器, 复杂度 $O(1)$

deque: 同上, $O(1)$

map: 双向迭代器, 不过由于是关联容器, 需要通过 key 访问 value 的方法, $O(h)$, h 为树的高度

unordered_map: 前向迭代器, 同上, 平摊复杂度 $O(1)$, 最差 $O(n)$, 也与散列函数的好坏有关。

string: 同 vector

标准 STL 序列容器：vector 、 string 、 deque 和 list 。

标准 STL 关联容器：set 、 multiset 、 map 和 multimap 。

非标准序列容器 slist 和 rope 。 slist 是一个单向链表， rope 本质上是一“ 重型 ”string 。

非标准的关联容器 hash_set 、 hase_multiset 、 hash_map 和 hash_multimap 。

几种标准的非 STL 容器，包括数

组、bitset 、 valarray 、 stack 、 queue 和 priority_queue 。

顺序容器类型	
顺序容器	
Vector	支持快速随机访问
List	支持快速插入 / 删除
Deque	双端队列
顺序容器适配器	
Stack	后进先出（ LIFO ）栈
Queue	先进后出（ FIFO ）队列
Priority_queue	有优先级管理的队列

关联容器类型	
Map	关联数组：元素通过键来存储和读取
Set	大小可变的集合，支持通过键实现的快速读取
Multimap	支持同一个键多次出现的 map 类型
Multiset	支持同一个键多次出现的 set 类型

开发 C 代码时,经常见到如下类型的结构体定义:

```
1  typedef struct list_t{
2      struct list_t *next;
3      struct list_t *prev;
4      char data[0];
5  }list_t;
```

请问在 32 位系统中,sizeof(list_t)的值为?

正确答案: B 你的答案: D (错误)

4byte

8byte

5byte

9byte

解析:

在用作定义时 char[0]是空数组,是不占空间的。如果定义 char[1],那么就是长度为 1 的数组,使用 char[0]则表示这唯一的数据,注意差别啦

```
1  Class A *pclassa=new ClassA[5];
2  delete pclassa;
```

则类 ClassA 的构造函数和析构函数的执行次数分别为()

正确答案: A 你的答案: C (错误)

5,1

1,1

5,5

1,5

解析:

Class A *pclassa=new ClassA[5]; new 了五个对象,所以构造 5 次,然后 Pclass 指向这五个对象

delete pclassa; 析构一次,delete[]pclassa 这样就析构 5 次

问题描述:

```
1  #pragma pack(2)
2  class BU
3  {
4      int number;
5      union UBffer
6      {
7          char buffer[13];
```

```

8             int number;
9         }ubuf;
10        void foo() {}
11        typedef char*(*f) (void*);
12        enum{hdd,ssd,blueray}disk;
13    }bu;

```

sizeof(bu)的值是()

正确答案: C 你的答案: E (错误)

- 20
- 21
- 22
- 23
- 24
- 非以上选项

解析:

```
#pragma pack(2)
```

```
class BU
```

```

{
    int number; // 4
    union UBffer
    {
        char buffer[13]; // 13
        int number; // 4
    }ubuf; // union 的大小取决于它所有的成员中，占用空间最大的一个成员的大小，并且
    // 需要内存对齐，这里因为#pragma pack(2)，所以 union 的大小为 14，如果不写#pragma
    // pack(2)，那么 union 大小为 16 【因为与 sizeof (int) =4 对齐】
    void foo(){} //0
    typedef char*(*f)(void*); //0
    enum{hdd,ssd,blueray}disk; // 4
}bu;

```

因此 sizeof (union) = 4+14 +0 +0 +4 = 22

“引用”与多态的关系？

正确答案: B 你的答案: A (错误)

两者没有关系

引用可以作为产生多态效果的手段

一个基类的引用不可以指向它的派生类实例

以上都不正确

解析:

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。

```
Class A;
```

```
Class B : Class A{...};
```

```
B b;
```

```
A& ref = b。
```

Which of the following calling convention(s) support(s) support variable-length parameter(e.g. printf)?

正确答案: A 你的答案: A (正确)

cdecl

stdcall

pascal

fastcall

解析:

函数调用约定（calling convention）函数调用约定不仅决定了发生函数调用时函数参数的入栈顺序，还决定了是由调用者函数还是被调用函数负责清除栈中的参数，还原堆栈。函数调用约定有很多方式，除了常见的__cdecl，__fastcall 和 __stdcall 之外，C++的编译器还支持 thiscall 方式，不少 C/C++编译器还支持 naked call 方式。

1.cdecl

编译器的命令行参数是/Gd。__cdecl 方式是 C/C++编译器默认的函数调用约定，所有非 C++成员函数和那些没有用 __stdcall 或 __fastcall 声明的函数都默认是 __cdecl 方式，它使用 C 函数调用方式，函数参数按照从右向左的顺序入栈，函数调用者负责清除栈中的参数，由于每次函数调用都要由编译器产生清除（还原）堆栈的代码，所以使用 __cdecl 方式编译的程序比使用 __stdcall 方式编译的程序要大很多，但是 __cdecl 调用方式是由函数调用者负责清除栈中的函数参数，所以这种方式支持可变参数，比如 printf 和 windows 的 API wsprintf 就是 __cdecl 调用方式。

32 位机器上定义如下结构体:

```
struct xx
```

```
{
```

```
    long long _x1;
```

```
    char _x2;
```

```
    int _x3;
```

```
    char _x4[2];
```

```
    static int _x5;
```

```
};
```

```
int xx::_x5;
```

1 请问 sizeof(xx)的大小是()

- A. 19
- B. 20
- C. 15
- D. 24

解析：

首先_x5 是静态变量可以不用管它，其次是要考虑字节对齐的问题。对于结构体中没有含有结构体变量的情况，有两条原则：

- 1) 结构体变量中成员的偏移量必须是成员大小的整数倍；
- 2) 结构体的最终大小必须是结构体最大简单类型的整数倍。

x1 的偏移量是 0，长度是 8，符合；

x2 的偏移量是 8，长度是 1，符合；

x3 的偏移量是 9，长度是 4，不符合，需要在 x2 之后填充 3 字节使得 x3 的偏移量达到 12；

x4 的偏移量是 16，长度是 2，符合；

此时总长度为 $(8) + (1+3) + (4) + (2) = 18$ ，而最大简单类型为 long long 长度为 8，因此需要在 x4 之后再填充 6 字节，使得总长度达到 24 可被 8 整除。因此 sizeof(xx)的结果为 24。

C++中关于堆和栈的说法，哪个是错误的：

正确答案: C 你的答案: A (错误)

堆的大小仅受操作系统的限制，栈的大小一般一般较小

在堆上频繁的调用 new/delete 容易产生内存碎片，栈没有这个问题

堆和栈都可以静态分配

堆和栈都可以动态分配

解析：

选 C，静态分配是指在编译阶段就能确定大小，由编译器进行分配，堆不可以进行静态分配，堆的申请都是在执行过程中进行的。

A，堆和栈的大小都可以设置，栈一般只有几 KB。

B，堆在动态分配时，要申请连续的内存空间，释放后会产生碎片。

D，堆是使用 malloc()、calloc()、realloc()等函数动态分配的，而使用 alloca()函数可以动态分配栈的内存空间，释放的时候由编译器自己释放。

有如下程序段：

```
1  #include <iostream>
2  using namespace std;
3
4  class A {
5      public:
6      ~A() {
```

```

7             cout << "~A()";
8         }
9     };
10    class B{
11    public:
12        virtual ~B() {
13            cout << "~B()";
14        }
15    };
16    class C: public A, public B {
17    public:
18        ~C() {
19            cout << "~C()";
20        }
21    };
22    int main() {
23        C * c = new C;
24        B * b1 = dynamic_cast<B *>(c);
25        A * a2 = dynamic_cast<A *>(b1);
26        delete a2;
27    }

```

则程序输出:

正确答案: D 你的答案: B (错误)

~C() ~B() ~A()

~C() ~A() ~B()

A) B) 都有可能

以上都不对

解析:

答案解析: 创建一个类对象 c, 然后动态类型转换, 让一个 B *b1 指针指向 c, 再一次动态类型转换, 让一个基类 A*a2 指针指向 b1, 当 delete a2 时, 调用析构函数, 但是基类 A 的析构函数不是虚函数, 所以只调用 A 的析构函数, 结果应该是: ~A()

动态的多态通过虚函数实现, 基类指针指向派生类的对象, 若指针调用的函数派生类中存在, 且在基类中声明为虚函数, 则调用的函数是派生类中的函数。

析构函数总是要声明为虚函数, 这样析构时, 先调用派生类的析构函数, 再调用基类的析构函数, 防止内存造成泄露

A 类的析构函数未声明为虚函数, 所以 A 类的指针, 只可以调用 A 类的析构函数

以下代码是否完全正确，执行可能得到的结果是_____。

```
1  class A{
2      int i;
3  };
4  class B{
5      A *p;
6  public:
7      B() {p=new A;}
8      ~B() {delete p;}
9  };
10 void sayHello(B b) {
11 }
12 int main() {
13     B b;
14     sayHello(b);
15 }
```

正确答案: C 你的答案: A (错误)

程序正常运行

程序编译错误

程序崩溃

程序死循环

解析:

为了清晰可见，我们从新把题目代码码一遍：

```
1  class A{
2      int i;
3  };
4  class B{
5      A *p;
6  public:
7      B() {p=new A;}
8      ~B() {delete p;}
9      /*
10     B(const B& ths) {
11         p = ths.p;
12     }*/
13 };
14 void sayHello(B x) {
15 }
16 int main() {
17     B b;
```

```

18         sayHello(b);
19     }

```

这里的错误原因是编译器在生成 default copy construction 的时候使用的 bitwise copy 语义，也就是只是简单的浅拷贝。上面被注释掉的程序就是编译器自动添加的部分。从而导致在 sayHello 中向参数 x 传递值时，调用了 bitwise copy 的拷贝构造函数，使得 x 对象和 b 对象中的值完全一致，包括 p 指针的值，在 x 离开作用域（也就是 sayHello 函数结束），x 发生析构，调用 delete 销毁了指针 p，同时在 main 函数结束的时候，析构 b 时又会调用一次 delete 删除指针 p。

也就是本程序会 delete 一直已经被 delete 的指针。可以做如下改进，来修复程序：

```

1   class A{
2       int i;
3   };
4   class B{
5       A *p;
6   public:
7       B() {p=new A;}
8       ~B() {delete p;}
9       B(const B& other) {
10           p = new A;           //构建新的指针
11           *p = *(other.p);    //将指向的内容复制，依然指向不同的位置
12       }
13 };
14 void sayHello(B b) {
15 }
16 int main() {
17     B b;
18     sayHello(b);
19 }

```

如上，在 B 中添加 copy 构造函数

程序出错在什么阶段__?

```

1   int main(void) {
2       http://www.taobao.com
3       cout << "welcome to taobao" << endl;
4   }

```

正确答案: F 你的答案: B (错误)

预处理阶段出错

编译阶段出错

汇编阶段出错

链接阶段出错

运行阶段出错

程序运行正常

解析:

<http://blog.csdn.net/szchtx/article/details/21647159>

双斜杠之后的 `www.csdn.net` 被当做注释了, 那么前面的 `http:` 是否合法? 这就是 C++ 中一个几乎不会被用到的语法, 标签。

带标签的语句是一种特殊的语句, 在语句前面有一个标识符 (即标签, 上段代码中的 `http`) 和一个冒号。使用 `goto label` 就可以跳到标签处执行, 比如可以在代码中写 `goto http`, 这样就会执行 `cout` 语句了。

`case` 就是一种标签, `case` 关键字和它对应的值一起, 称为 `case` 标签。类中的 `public`、`private`、`protect` 也是标签, 称为成员访问标签。

扯得这么高大上, 其实就是一个 `goto` 语句

`case` 标签必须是**整型常量**表达式

请记住**整型常量**这四个字, 不满足这个特性的不能作为 `case` 值, 编译会报错。这也决定了 `switch` 的参数必须是**整型**的。

整型, 意味着浮点数是不合法的, 如 `case 3.14:` 不可以; 常量, 意味着变量是不合法的, 如 `case ival: ival` 不能是变量。

(1) C++ 中的 `const int`, 注意仅限于 C++ 中的 `const`, C 中的 `const` 是只读变量, 不是常量;

(2) 单个字符, 如 `case 'a':` 是合法的, 因为文字字符是常量, 被转成 ASCII 码, 为整型;

(3) 使用 `#define` 定义的整型, `#define` 定义的一般为常量, 比如 `#define pi 3.14`, 但是也必须是整型才可以;

(4) 使用 `enum` 定义的枚举成员。因为枚举成员是 `const` 的, 且为整型。如果不手动指定枚举值, 则默认枚举值为从 0 开始, 依次加 1。如下这段代码正常运行:

有如下模板定义:

```
1  template <class T>
2  T fun(T x, T y) {
3      return x*x+y*y;
4  }
```

在下列对 fun 的调用中，错误的是（）

正确答案: B 你的答案: D (错误)

```
fun(1, 2)
fun(1.0, 2)
fun(2.0, 1.0)
fun<float>(1, 2.0)
```

解析:

答案: B

首先这个题目有误，模板函数应该定义为（加返回值类似声明）：

```
template <class T>
```

```
T fun(T x,T y){
return x*x+y*y;}

```

然后这里 T 要求类型一致

A，类型一致，为 int 型

B，类型不一致，错我

C，类型一致，为 folat 型

D，用<float>进行声明，后面的实参会强制类型转换为 float，所以也是类型一致的。

如下程序

```
1      void main()
2  {
3      float a = 1;
4      cout << boolalpha << ((int)a == (int &a));
5      float b = 0;
6      cout << boolalpha << ((int)b == (int &b));
7  }
```

该程序输出结果为

正确答案: D 你的答案: A (错误)

```
true true
false false
true false
false true
```

解析:

```
1 float a=1;
2 int b = (int) a;
```

```
3 int c = (int &)a;
```

浮点 a 的二进制表示方法为：0 01111111 00000000...(23 位 0)

(int)a 表示以浮点数 a 为参数构造整数（即 float 转换为 int）

(int &)a 则是告诉编译器将 a 看成 int 对待（不做任何转换），所以(int &)a 值为 1065353216.

(int&)a:将 a 的引用强制转换为整型,意思是 a 所在的内存，本来定义的时候为 float 类型并初始为 1.0f，但现在我要按 int 类型解释这段内存（也就是说 a 所在的内存地址中的数据本来是按 float 型存储表示的，你非要按 int 型来解释不可）。

1.0f 在内存中的存储为

0 011 1111 1 000 0000 0000 0000 0000 0000.

把他按整型数解释为 $2^{29}+2^{28}+2^{27}+2^{26}+2^{25}+2^{24}+2^{23}=1065353216$

(int&)a 相当于*(int*)&a，*(int*)(&a)，*((int*)&a)

(int)a a 在内存中的值转换成 int 类型

【浮点数存储】

通用多态是指

正确答案: D 你的答案: C (错误)

强制多态和包含多态

重载多态和强制多态

参数多态和重载多态

包含多态和参数多态

解析:

重载多态和强制多态是 指特定多态。
参数多态和包含多态是 指通用多态。

在 c++语言中，这种多态性可以通过强制多态、重载多态、类型参数化多态、包含多态 4 种形式来实现。

类型参数化多态和包含多态统称为一般多态性，用来系统地刻画语义上相关的一组类型。

重载多态和强制多态统称为特殊多态性，用来刻画语义上无关联的类型间的关系

```
//1. 参数多态
```

```
//包括函数模板和类模板
```

```
//2. 包含多态 virtual
```

```

class A{
|
|       virtual void foo()
|       {
|           printf("A virtual void foo()");
|       }
|
| };
|
class B : public A {
|
|       void foo() {
|           printf("B void foo()");
|       }
| };
|
void test() {
|
|       A *a = new B();
|
|       a->foo(); // B void foo()
|
| }

```

//3. 重载多态

//重载多态是指函数名相同，但函数的参数个数或者类型不同的函数构成多态

```

void foo(int);
|
void foo(int, int);
|

```

//4. 强制多态

//强制类型转换

下列关于异常处理的描述中，理解不正确的是：

正确答案: D 你的答案: D (正确)

C++语言的异常处理机制通过 3 个保留字 throw、try 和 catch 实现。
任何需要检测的语句必须在 try 语句块中执行，并由 throw 语句抛出异常。
throw 语句抛出异常后，catch 利用数据类型匹配进行异常捕获。

一旦 catch 捕获异常，不能将异常用 throw 语句再次抛出。

解析：

catch 异常后，同样可以用 throw 语句在此抛出，如果没有解决，会直接向上层继续抛，直至 main 函数，然后异常停止！

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：

正确答案: D 你的答案: A (错误)

一元运算符

二元运算符

选项 A) 和选项 B) 都可能

重载错误

解析：

友元函数重载时,参数列表为 1,说明是 1 元,为 2 说明是 2 元 成员函数重载时,参数列表为空,是一元,参数列表是 1,为 2 元

以下选项中不属于 C 语言标识符的是？

正确答案: A 你的答案: C (错误)

常量

用户标识符

关键字

预定义标识符

解析：

C 语言中的标识符有：关键字、预定义标识符、用户标识符

若为成员函数重载，参数一般为类对象的引用，另一个参数由 this 指针所指向，故不显示。

若为友元函数重载，则有两个参数！

代码执行后，a 和 b 的值分别为？

```
1  class Test{
2      public:
3          int a;
4          int b;
5          virtual void fun() {}
6          Test(int temp1 = 0, int temp2 = 0)
7          {
8              a=temp1 ;
```

```

9             b=temp2 ;
10        }
11        int getA()
12        {
13            return a;
14        }
15        int getB()
16        {
17            return b;
18        }
19    };
20
21    int main()
22    {
23        Test obj(5, 10);
24        // Changing a and b
25        int* pInt = (int*)&obj;
26        *(pInt+0) = 100;
27        *(pInt+1) = 200;
28        cout << "a = " << obj.getA() << endl;
29        cout << "b = " << obj.getB() << endl;
30        return 0;
31    }

```

正确答案: A 你的答案: C (错误)

```

200 10
5 10
100 200
100 10

```

解析:

这么需要考虑虚函数表，指向虚函数表的指针在 32 位系统下占用 4 个字节，其地址分布在整个类成员变量的地址的首部，接下来就是变量 a 的地址、b 的地址。当将 test 对象 obj 赋给指向整型的 pInt 后，指针 pInt 指向了地址的首部也就是虚函数表指针，所以*(pInt+0)=100 改变的是虚函数表的值，接下来*(pInt+1)=200 改变的是变量 a 的值，变量 b 没有变换。

定义网络传输数据包为

```

1    class packet{
2        int size;
3        void data[0];
4    }

```

其中 data 的作用是？

正确答案: C 你的答案: A (错误)

维护数据包空间的连续性
数据分割位
指向独立的数据空间
无任何作用

解析:

选 C 是没错的。

1.这个叫柔性数组，它的作用跟指针差不多，但是指针占空间，而它不占空间，这就意味着可以节省空间。

2.该数组的内存地址就和它后面的元素地址相同，意味着无需初始化，数组名就是后面元素的地址，直接就能当指针使用。例如，制作动态 buffer，可以这样分配空间 `malloc(sizeof(structXXX) + buff_len)`；直接就把 buffer 的结构体和缓冲区一块分配了。这样使用释放一次即可，如果使用指针，则需要释放两次。

3.也可以写成 `data[1]`或 `data[]`，是考虑到可移植性的原因，因为有些编译器不支持 0 数组。

如何捕获异常可以使得代码通过编译？

```
1 class A {  
2     public:  
3         A() {}  
4 };  
5 void foo() {  
6     throw new A;  
7 }
```

正确答案: B 你的答案: C (错误)

```
catch (A && x)  
catch (A * x)  
catch (A & x)  
以上都是
```

解析:

`throw new A;` `throw` 出的是 `A *`类型的指针所以 `catch (A * x)`;

若要重载+、=、<<、=和[]运算符，则必须作为类成员重载的运算符是

正确答案: D

+和=
=和<<

==和<<
=和[]

解析：

答案 D

解释：

- (1)只能使用**成员函数重载**的运算符有：=、()、[]、->、new、delete。
- (2)单目运算符最好重载为成员函数。
- (3)对于复合的赋值运算符如+=、-=、*=、/=、&=、!=、~=、%=、>>=、<<=建议重载为成员函数。
- (4)对于**其它运算符**，建议重载为友元函数。

运算符重载的方法是定义一个重载运算符的函数，在需要执行被重载的运算符时，系统就自动调用该函数，以实现相应的运算。也就是说，运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。重载运算符的函数一般格式如下：

```
函数类型 operator 运算符名称 (形参表列)
{
    对运算符的重载处理
}
```

重载为类成员函数时参数个数=原操作数个数-1（后置++、--除外）

重载为友元函数时 参数个数=原操作数个数，且至少应该有一个自定义类型的形参

参考 C++编程思想 12.4.1

必须是成员的有：=、()、[]、->、->*

```
1  int* pint = 0;
2  pint += 6;
3  cout << pint << endl;
```

以上程序的运行结果是：

正确答案: C 你的答案: F (错误)

12

72

24

0

6

任意数

解析:

第一句的意思是将 pint 指针指向 0 地址处，由于指针类型是 int，每次加 1 相当于移动四个字节，（在 int 为四个字节的机器上）；加上 6，地址为 0x18

What is the result of the following program?

```
1  char *f(char *str , char ch)
2  {
3      char *it1 = str;
4      char *it2 = str;
5      while(*it2 != '\0')
6      {
7          while(*it2 == ch)
8          {
9              it2++;
10         }
11         *it1++ = *it2++;
12     }
13     return str;
14 }
15
16 int main(void)
17 {
18     char *a = new char[10];
19     strcpy(a , "abddcccd");
20     cout<<f(a, 'c');
21     return 0;
22 }
```

正确答案: D 你的答案: E (错误)

abddcccd

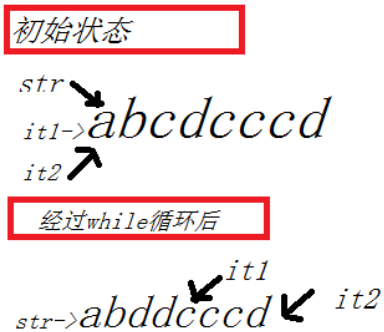
abdd

abcc

abddcccd

Access violation

解析:



在 `return str;`前面加一句 `*it1 = '\0'`; 这样输出就是 `abdd`, 没有这句就是 `abddcccd`

友元能通过类的对象访问类中的所有成员吗?

正确答案: A 你的答案: A (正确)

是
错

解析:

友元函数不能直接访问类的成员, 只能访问对象成员

下列运算符中, 在 C++ 语言中不能重载的是:

正确答案: C 你的答案: C (正确)

*
>=
::
delete

解析:

在 C++ 中, `sizeof` 运算符, `.` 成员运算符, `*` 成员指针运算符, `::` 作用域解析运算符以及 `?:` 条件运算符不能被重载, 因此答案选 C

全局对象在 `main` 退出后, 程序结束前析构吗?

正确答案: A 你的答案: B (错误)

是
错

解析:

全局对象的生命周期跨越整个程序运行时间，优先于 `main` 函数进行初始化，在 `main` 函数返回时撤销.即析构！

下列代码的输出为：

```
1  #include<iostream>
2  #include<vector>
3  using namespace std;
4
5  int main(void)
6  {
7      vector<int>array;
8      array.push_back(100);
9      array.push_back(300);
10     array.push_back(300);
11     array.push_back(500);
12     vector<int>::iterator itor;
13     for (itor = array.begin(); itor != array.end(); itor++)
14     {
15         if (*itor == 300)
16         {
17             itor = array.erase(itor);
18         }
19     }
20     for (itor = array.begin(); itor != array.end(); itor++)
21     {
22         cout << *itor << " ";
23     }
24     return 0;
25 }
```

正确答案: B 你的答案: B (正确)

100 300 300 500

100 300 500

100 500

程序错误

解析：

`vector` 中使用一次 `erase()` 方法后，原来的容器会被新的容器所取代，现在的容器指针指向的是一个野指针，所以需要重新对容器指针进行赋值操作：`itor=array.begin()`；添加这条语句后就可以完全去掉两个 `300` 元素

C++中构造函数和析构函数可以抛出异常吗?

正确答案: C 你的答案: A (错误)

- 都不行
- 都可以
- 只有构造函数可以
- 只有析构函数可以

解析:

答案是 C

- 1.不建议在构造函数中抛出异常;
 - 2.构造函数抛出异常时,析构函数将不会被执行,需要手动去释放内存
-
- 1.析构函数不应该抛出异常;
 - 2.当析构函数中会有一些可能发生异常时,那么就必须要要把这种可能发生的异常完全封装在析构函数内部,决不能让它抛出函数之外;
 - 3.析构函数异常相对要复杂一些,存在一种冲突状态,程序将直接崩溃:异常被称为“栈展开(stack unwinding)”【备注】的过程中时,从析构函数抛出异常,C++运行时系统会处于无法决断的境遇,因此C++语言担保,当处于这一点时,会调用 `terminate()` 来杀死进程。因此,当处理另一个异常的过程中时,不要从析构函数抛出异常,抛出异常时,其子对象将被逆序析构

关于函数输入参数的正确描述都有哪些?()

正确答案: A C D 你的答案: B (错误)

始终用 `const` 限制所有指向只读输入参数的指针和引用
值传递只用于原始类型(`int, float...`)的输入参数
优先按 `const` 的引用取得其他用户定义类型的输入
如果函数需要修改其参数副本,则可以考虑通过值传递代替通过引用传递

解析:

- a) 始终用 `const` 限制所有指向只输入参数的指针和引用。
- b) 优先通过值来取得原始类型和复制开销比较低的值的对象。
- c) 优先按 `const` 的引用取得其他用户定义类型的输入。
- d) 如果函数需要其参数的副本,则可以考虑通过值传递代替通过引用传递。这在概念上等同于通过 `const` 引用传递加上一次复制,能够帮助编译器更好的优化掉临时变量。

有哪几种情况只能用 `intialization list` 而不能用 `assignment`?

正确答案: A B C 你的答案: A C (错误)

当类中含有 `const` 成员变量
基类无默认构造函数时,有参的构造函数都需要初始化表。
当类中含有 `reference` 成员变量
当类中含有 `static` 成员变量

解析:

因为 **const 对象以及引用只能初始化而不能赋值**，所以只能使用**成员初始化列表**。

对于非内置类型，在进入函数体之前，如果没有提供显式初始化，会调用默认构造函数进行初始化。若没有默认构造函数，则编译器尝试调用默认构造函数将会失败，所以**如果没有默认构造函数，则必须在初始化列表中显示的调用构造函数**。

static 成员在**执行构造函数前就已经构造好了**，即使不存在类对象，也可以被使用，不需要初始化列表

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?

正确答案: D 你的答案: B (错误)

EOF

非 0

-1

0

解析:

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。

以下叙述中不正确的是

正确答案: D 你的答案: A (错误)

在不同的函数中可以使用相同名字的变量

函数中的形式参数是局部变量

在一个函数内定义的变量只在本函数范围内有效

在一个函数内的复合语句中定义的变量在本函数范围内有效

解析:

自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量，只在该函数内有效。在复合语句中定义的自动变量只在该复合语句中有效。**自动变量属于动态存储方式**，只有在使用它，即定义该变量的函数被调用时才给它分配存储单元，开始它的生存期。函数调用结束，释放存储单元，结束生存期。因此函数调用结束之后，自动变量的值不能保留。在复合语句中定义的自动变量，在退出复合语句后也不能再使用，否则将引起错误。

有如下程序段，请问 **k** 的值是

```
1  enum {  
2      a, b=5, c, d=4, e  
3  } k;  
4  k =c;
```

正确答案: D 你的答案: B (错误)

3
4
5
6

解析:

`enum{a, b=5, c, d=4, e} k;`

默认 $a = 0$ ，既然 $b = 5$ ，根据那么枚举类型的后一个元素在不赋值的情况下，比前一个元素大 1，那么 c 默认为 6， $d = 4$ ， e 默认为 5

只能使用成员函数重载的运算符有：`=`、`()`、`[]`、`->`、`new`、`delete`。

下面有关 `volatile` 说法正确的有？

正确答案: A B C 你的答案: A C D (错误)

当读取一个变量时，为提高存取速度，编译器优化时有时会先把变量读取到一个寄存器中；以后再取变量值时，就直接从寄存器中取值

优化器在用到 `volatile` 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

`volatile` 适用于多线程应用中被几个任务共享的变量

一个参数不可以既是 `const` 又是 `volatile`

解析:

D 为什么是错的:

`volatile` 作用是避免编译器优化，说明它是随时会变的，它不是 `non-const`，和 `const` 不矛盾。被 `const` 修饰的变量只是在当前作用范围无法修改，但是可能被其它程序修改。所以 `const volatile int i = 0;` 表示:

任何对 i 的直接修改都是错误的，但是 i 可能被意外情况修改掉，不要做无意义的优化。

关于浅复制和深复制的说法，下列说法正确的是

正确答案: A B C D 你的答案: B C D (错误)

浅层复制: 只复制指向对象的指针，而不复制引用对象本身。

深层复制: 复制引用对象本身。

如果是浅复制，若类中存在成员变量指针，修改一个对象一定会影响另外一个对象

如果是深拷贝，修改一个对象不会影响到另外一个对象

解析:

深层拷贝 也叫做深拷贝，个人的理解应该是 拷贝了引用对象的内存，这样就会产出两块不同的内存区，而浅拷贝则是两条或多条指针指向了相同的同一块内存区域。

下列说法错误的有 ()

正确答案: A C D 你的答案: C D (错误)

在类方法中可用 `this` 来调用本类的类方法

在类方法中调用本类的类方法时可直接调用

在类方法中只能调用本类中的类方法

在类方法中绝对不能调用实例方法

解析:

A: 类方法是指类中被 **static** 修饰的方法, 无 **this** 指针。

C: 类方法是可以调用其他类的 **static 方法的**。

D: **可以在类方法中生成实例对象再调用实例方法**。(这个我也打错了, 想想应该是这个意思)

拷贝构造函数的特点是 ()

正确答案: B D 你的答案: A D (错误)

该函数名同类名, 也是一种构造函数, ~~该函数返回自身引用~~

该函数~~只有一个参数~~, 是对某个对象的引用

每个类都必须有一个拷贝初始化构造函数, 如果类中没有说明拷贝构造函数, 则编译器系统会自动生成一个缺省拷贝构造函数, 作为该类的~~保护成员~~

拷贝初始化构造函数的作用是将一个已知对象的数据成员值拷贝给正在创建的另一个同类的对象

解析:

拷贝函数和构造函数没有返回值, A 错;

拷贝构造函数的参数可以使一个或多个, 但左起第一个必须是类的引用对象, B 错;

若类定义中没有声明拷贝构造函数, 则编译器会自动生成一个缺省的拷贝构造函数, 但是不会是该类的保护成员, C 错;

通过拷贝函数可以将另一个对象作为对象的初值, D 对

某 32 位系统下, C++ 程序如下所示, sizeof 的值应为?

1 char str[] = "http://www.renren.com" (长度为 21)

2 char *p = str ;

请计算

1 sizeof (str) = ? (1)

2 sizeof (p) = ? (2)

3 void Foo (char str[100]) {

4 sizeof(str) = ? (3)

5 }

6 void *p = malloc(100);

7 sizeof (p) = ? (4)

正确答案: C 你的答案: B (错误)

22, 22, 100, 100

4, 4, 4, 4

22, 4, 4, 4

22, 4, 100, 4

解析：

C

函数传递时数组退化为指针，

数组长度就是你定义数组时方括号中数字的大小。

字符串长度就是字符的个数。

字符串所占空间大小就是字符串长度+1（1 个结束符的长度）；

作为函数的形参传递时数组退化为指针

`char c[10]={'x','y','z'},d[]={'x','y','z'};` 不以 `'\0'` 作为结束标志的，它就没有串结束标志， 字符数组的长度，就是数组的长度

`sizeof(c)=10, sizeof(d)=3`

分析一下这段程序的输出

```
1  #include<iostream>
2      using namespace std;
3      class B
4      {
5      public:
6          B()
7          {
8              cout <<"default constructor"<<" ";
9          }
10         ~B()
11         {
12             cout <<"destructed"<<" ";
13         }
14         B(int i): data(i)
15         {
16             cout <<"constructed by parameter"<< data <<" ";
17         }
18         private: int data;
19     };
20     B Play( B b)
21     {
22         return b;
23     }
24     int main(int argc, char *argv[])
25     {
26         B temp = Play(5);
27         return 0;
```

正确答案: A 你的答案: B (错误)

```
constructed by parameter5  destructed  destructed
constructed by parameter5  destructed
default constructor"  constructed by parameter5 destructed
default constructor"  constructed by parameter5 destructed  destruc
ted
```

解析:

1. 调用 Play 函数需要将 5 隐式类型转换为 Play 函数中的形参 b，会调用 B 的 B(int i): data(i)，打印“constructed by parameter5”。
 2. Play 函数返回时需要调用 B 的复制构造函数给对象 temp 初始化。
 3. Play 函数返回后需要调用 b 的析构函数，将 Play 函数的形参释放，打印“destructed”。
 4. main 函数返回后需要释放 temp，打印“destructed”。
-
5. 代码可以通过编译吗？如果不能应该如何修改？

```
1  template<class T> class Foo{
2      T tVar;
3      public:
4          Foo(T t) : tVar(t) { }
5  };
6
7  template<class T> class FooDerived:public Foo<T>
8  {
9  };
10
11 int main()
12 {
13     FooDerived<int> d(5);
14     return 0;
15 }
```

正确答案: D 你的答案: A (错误)

6. 代码可以正确通过编译。
 7. 编译错误，FooDerived 是一个继承模板类的非模板类，它的类型不能改变。
 8. 编译错误，tVal 变量是一个不确定的类型。
 9. 编译错误，可以在 FooDerived 类中添加一个构造函数解决问题。
-

请选择下列程序的运行结果

```
1  #include<iostream>
2  using namespace std;
3  class B0//基类 B0 声明
4  {
5  public://外部接口
6  virtual void display()//虚成员函数
7  {
8      cout<<"B0::display0"<<endl;}
9  };
10 class B1:public B0//公有派生
11 {
12 public:
13     void display() { cout<<"B1::display0"<<endl; }
14 };
15 class D1: public B1//公有派生
16 {
17 public:
18     void display() { cout<<"D1::display0"<<endl; }
19 };
20 void fun(B0 ptr)//普通函数
21 {
22     ptr.display();
23 }
24 int main()//主函数
25 {
26     B0 b0;//声明基类对象和指针
27     B1 b1;//声明派生类对象
28     D1 d1;//声明派生类对象
29     fun(b0);//调用基类 B0 函数成员
30     fun(b1);//调用派生类 B1 函数成员
31     fun(d1);//调用派生类 D1 函数成员
32 }
```

正确答案: A 你的答案: C (错误)

B0::display() B0::display() B0::display()
B0::display() B0::display() D1::display()
B0::display() B1::display() D1::display()
B0::display() B1::display() B1::display()

解析:

```
void fun (B0 ptr)//普通函数
{
    ptr.display();
}
```

这里使用的不是按地址传递，这样会转化为基类对象，直接调用基类的成员函数，如果是指针传递，改为 B0 *ptr, ptr->display(), 可以实现多态

c++不能重载的运算符有.（点号），::（域解析符），?:（条件语句运算符），sizeof（求字节运算符），typeid，static_cast，dynamic_cast，interpret_cast（三类类型转换符）。---出自 C++ primer plus

以下哪个函数可以在源地址和目的地址的位置任意的情况下，在源地址和目的地址的空间大小任意的情况下实现二进制代码块的复制？

正确答案: B 你的答案: A (错误)

```
memcpy()
memmove()
memset()
strcpy()
```

解析：

memcpy 与 memmove 的目的都是将 N 个字节的源内存地址的内容拷贝到目标内存地址中。但当源内存和目标内存存在重叠时，memcpy 会出现错误，而 memmove 能正确地实施拷贝，但这也增加了一点点开销。

所以选 B

有如下程序段：

```
1      #include "stdio.h"
2
3      class A
4      {
5          public:
6              A()
7              {
8                  printf("1");
9              }
10             A(A &a)
11             {
12                 printf("2");
13             }
14             A &operator=(const A &a)
15             {
```

```

16         printf("3");
17         return *this;
18     }
19 };
20 int main()
21 {
22     A a;
23     A b = a;
24 }

```

则程序输出为:

正确答案: A 你的答案: D (错误)

12

13

无法确定

编译出错

解析:

A a,定义一个对象,毫无疑问调用构造函数

A b=a, 这是定义了对象 b, 且以 a 对 b 进行初始化, 这个时候需要调用拷贝构造函数。

如果写成 A a; A b; b=a; 则是调用后面重载的赋值函数, 这种情况应该输出 113。

这个题主要考察赋值和初始化的区别。

用户双击鼠标时产生的消息序列, 下面正确的是 ()

正确答案: D 你的答案: A (错误)

WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN, WM_LBUTTONUP

WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONUP, WM_LBUTTONDBLCLK

WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDOWN, WM_LBUTTONDBLCLK

WM_LBUTTONDOWN, WM_LBUTTONUP, WM_LBUTTONDBLCLK, WM_LBUTTONUP

解析:

双击即点击左键两下, 第一次触发 LBUTTONDOWN 和 LBUTTONUP, 第二次点击时触发双击事件 LBUTTONDBLCLK (doubleclick), 放掉再触发 LBUTTONUP