# Java

## 1. 下列说法正确的是（）？

A.对于局部内部类，只有在方法的局部变量被标记为final或局部变量是effctively final的，内部类才能使用它们

B.成员内部类位于外部类内部，可以直接调用外部类的所有方法（静态方法和非静态方法）

C.由于匿名内部类只能用在方法内部，所以匿名内部类的用法与局部内部类是一致的

D.静态内部类可以访问外部类的成员变量



选AB

## 2. 如果一个接口Glass有个方法setColor()，有个类BlueGlass实现接口Glass，则在类BlueGlass中正确的是？  ( )

A.protected void setColor() { …}

B.void setColor() { …}

C.public void setColor() { …}

D.以上语句都可以用在类BlueGlass中

|  |
| --- |
| JAVA 子类重写继承的方法时,不可以降低方法的访问权限，子类继承父类的访问修饰符要比父类的更大，也就是更加开放，假如我父类是protected修饰的，其子类只能是protected或者public，绝对不能是friendly(默认的访问范围)或者private，当然使用private就不是继承了。还要注意的是，继承当中子类抛出的异常必须是父类抛出的异常的子异常，或者子类抛出的异常要比父类抛出的异常要少。  选C |

# C/C++

## 1. 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

class A{

    public:

    void test(){printf("test A");}

};

int main(){

    A\*pA=NULL;

    pA->test();

return 0;

}

A.编译出错

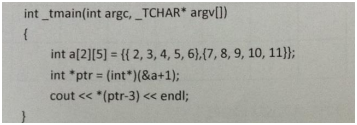
B.程序运行奔溃

C.输出"test A"

D.输出乱码

|  |
| --- |
| 其实这个是可以正常运行的，选C  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();  这语句的意图是：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。 |

## 2. 如下代码的输出结果是（）



A.3

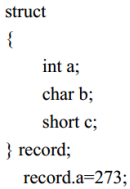
B.4

C.9

D.10

|  |
| --- |
| 选C  数组名是数组这种类型的变量名，所以对数组名取地址是取的整个数组的地址，所以&a+1自然要跨过整个数组的长度，本题即跨过2\*5 = 10个int的长度。  有了上述概念，再来分析这道题，int \*ptr = (int \*)(&a+1)，p此时指向的地址应该是a[1][4]后面的地址，由于ptr是int型指针，ptr-3应该是ptr向前移动3个元素，即ptr-3指向a[1][2]，所以\*(ptr -3) = 9 |
| a是数组首地址，也就是a[0]的地址，&a是对象（数组）首地址，a+1是数组下一元素的地址，即a[1],&a+1是下一个对象的地址。 |

## 3. 某计算机存储器按字节变址，采用小端方式存放数据。假定编译器规定int 型和 short 型长度分别为 32 位和 16 位，并且数据按边界对齐存储。某 C 语言程序段如下：



若record 变量的首地址为 0xC008，则地址 0xC008 中内容及 record.c 的地址是（ ）。

A.0x00、 0xC00D

B.0x00、 0xC00E

C.0x11、 0xC00D

D.0x11、 0xC00E

|  |
| --- |
| 大端模式：数据低位放在内存高位；  小端模式：数据低位放在内存低位；  将a从十进制化成16进制： a = 273 = 0x0000 0111(32位)  一个字节存储8位，因此0x0000 0111分为0x00，0x00，0x01，0x11四个字节来储存。其中0x11是数据的低位，根据小端模式应该存在地址的地位，也就是0xC008  数据存储时要内存对齐，内存对齐的三原则不记得的可以百度一下。  说的太麻烦，给张图。  ----改----  对不起各位牛友，评论区有人说这个图画的有点问题，看了一下确实有点问题，不过将就还能看。错误的地方就在存a的四个字节不是1，1，0，1，而是11,01,00,00。  https://uploadfiles.nowcoder.com/images/20170612/9602113_1497275928883_BCA2A9D993D9AB1072A35FE99645AB22 |

## 4. 假设在一个 32 位 little endian 的机器上运行下面的程序，结果是多少？

#include <stdio.h>

int main(){

  long long a = 1, b = 2, c = 3;

  printf("%d %d %d\n", a, b, c);

 return 0;

}

A.1,2,3

B.1,0,2

C.1,3,2

D.3,2,1

|  |
| --- |
| printf函数的原型是printf（const char\*,...）;  第二个参数是任意个数目的参数，所以printf函数不是分个把参数入栈，而是一股脑全部压入堆栈，因此对于8字节的数据来说，堆栈中的数据是1000 0000 2000 0000 3000 0000，%d按四字节输出，因此，第一个%d输出1，第二个%d输出0，第三个%d输出2。  如果改成printf("%d ", a);printf("%d ", b);printf("%d\n", c);那结果就是1，2，3.  选B |

## 5. 以下代码是否完全正确，执行可能得到的结果是\_\_\_\_。

class A{

   int i;

};

class B{

   A \*p;

public:

   B(){p=new A;}

   ~B(){delete p;}

};

void sayHello(B b){

}

int main(){

   B b;

   sayHello(b);

}

A.程序正常运行

B.程序编译错误

C.程序崩溃

D.程序死循环

|  |
| --- |
| 选C  为了清晰可见，我们从新把题目代码码一遍：  class A{  int i;  };  class B{  A \*p;  public:  B(){p=new A;}  ~B(){delete p;}  /\*  B(const B& ths){  p = ths.p;  }\*/  };  void sayHello(B x){  }  int main(){  B b;  sayHello(b);  }  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：  class A{  int i;  };  class B{  A \*p;  public:  B(){p=new A;}  ~B(){delete p;}  B(const B& other){  p = new A; //构建新的指针  \*p = \*(other.p); //将指向的内容复制，依然指向不同的位置  }  };  void sayHello(B b){  }  int main(){  B b;  sayHello(b);  }  如上，在B中添加copy 构造函数 |

6. 以下代码的执行结果是().

int main(){

   int i=-2147483648;

   return printf("%d,%d,%d,%d",~i,-i,1-i,-1-i);

}

A.0,2147483648,2147483649,2147483647

B.0,-2147483648,-2147483647,2147483647

C.2147483647,2147483648,2147483649,2147483647

D.2147483647,-2147483648,-2147483647,2147483647

|  |
| --- |
| 选D  总结以上内容：正数的原码、反码、补码形式一致，负数的反码为原码的数值位取反，补码为反码+1也即是原码的数值位取反再+1，计算机中以补码表示数据和运算，而32位最小负整数的补码为 1000 0000  0000  0000  0000  0000  0000  0000。 |

## 6. 若有定义int（\*pt）[3]；则下列说法正确的是：

A.定义了基类型为int的三个指针变量

B.定义了基类型为int的具有三个元素的指针数组pt

C.定义了一个名为\*pt、具有三个元素的整型数组

D.定义了一个名为pt的指针变量，它可以指向每行有三个整数元素的二维数组

|  |
| --- |
| 选D。  int (\*pt)[3]，首先看括号内，\*pt说明pt是一个指针，其指向的内容是int[3]，具有3个int元素的数组。  D选项说，可以指向每行有三个整数元素的二维数组，即int[][3]。  int(\*pt)[3] = NULL;  int arr[2][3] = {0};  pt = arr; |

## 7. 在32位机器上，下列代码中

class A

{

int i;

union U

{

char buff[13];

int i;

}u;

void foo() { }

typedef char\* (\*f)(void\*);

enum{red, green, blue} color;

}a;

sizeof(a)的值是（）

A.20

B.21

C.22

D.24

E.非以上选项

|  |
| --- |
| 答案：D  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  枚举类型占取4个字节  总共占取4+16+4=24个字节 |

# 三、J2EE

1. J2EE中，当把来自客户机的HTTP请求委托给servlet时，会调用HttpServlet的（    ）方法.

A.service

B.doget

C.dopost

D.init

|  |
| --- |
| A  HttpServlet容器响应Web客户请求流程如下：  1）Web客户向Servlet容器发出Http请求；  2）Servlet容器解析Web客户的Http请求；  3）Servlet容器创建一个HttpRequest对象，在这个对象中封装Http请求信息；  4）Servlet容器创建一个HttpResponse对象；  5）Servlet容器调用HttpServlet的service方法，这个方法中会根据request的Method来判断具体是执行doGet还是doPost，把HttpRequest和HttpResponse对象作为service方法的参数传给HttpServlet对象；  6）HttpServlet调用HttpRequest的有关方法，获取HTTP请求信息；  7）HttpServlet调用HttpResponse的有关方法，生成响应数据；  8）Servlet容器把HttpServlet的响应结果传给Web客户。  doGet() 或 doPost() 是创建HttpServlet时需要覆盖的方法. |