比特就业课假期作业-数据结构&C++&Linux作业答案

作业说明：

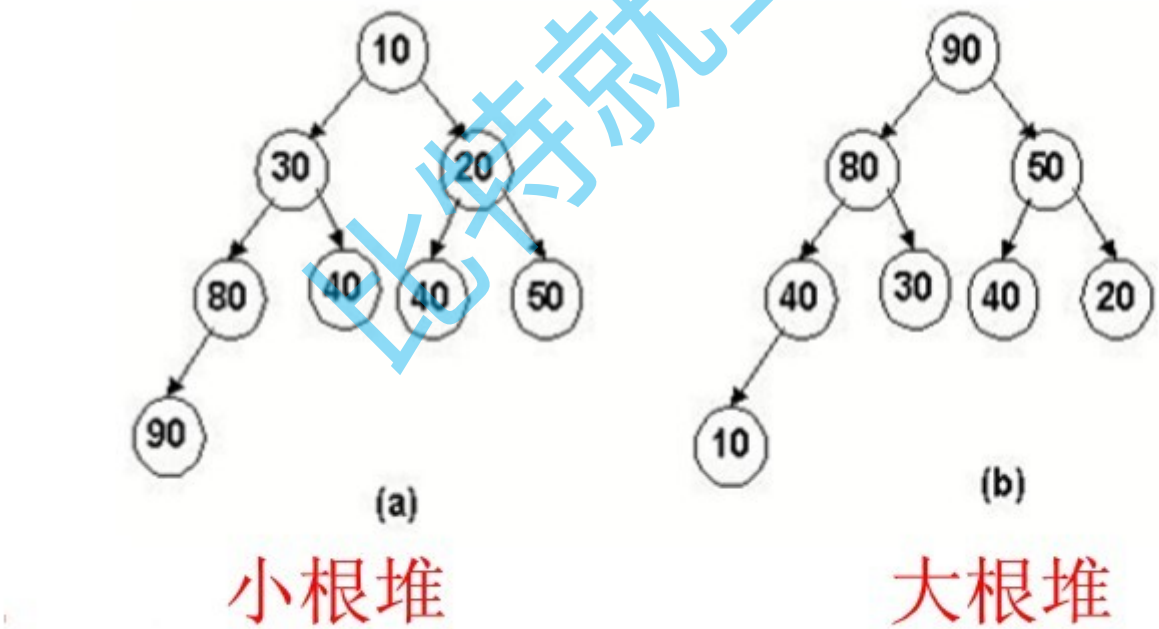
> 1、本次作业涵盖内容为C语言，C++，Linux，数据结构相关知识点
>
> 2、如果对试卷上的题目，或者答案有问题，可以联系自己老师哦~~

day01

一、选择题

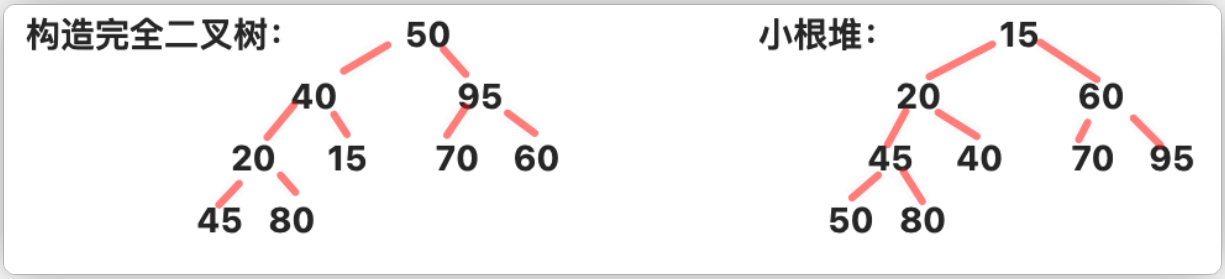
1、

答案解析：D
1.最大堆和最小堆是二叉堆的两种形式。
2.最大堆：根结点的键值是所有堆结点键值中最大者，且每个结点的值都比其孩子的值大。
3.最小堆：根结点的键值是所有堆结点键值中最小者，且每个结点的值都比其孩子的值小。
综上：选择D选项



2、

答案解析：A
小根堆：顾名思义，作为根的数值要最小，首先构造完全二叉树，然后让根与其左右子节点比，将他们三个最小的值选出来作为根，反复交换使所有结点满足小根即可。构造小根堆示意图如下：

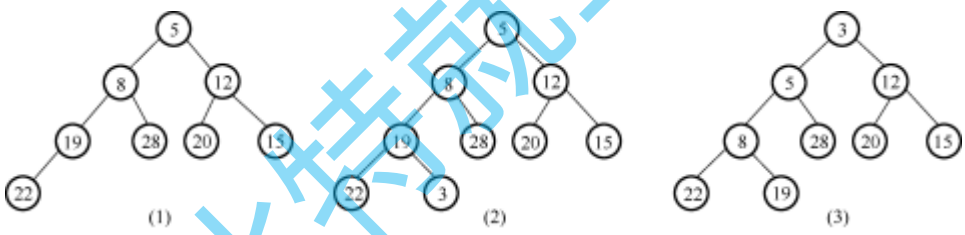


3、

答案选择：B
1.把大根堆画成完全二叉树后，堆插入元素相当于是从最后面添加叶子节点18，先是18和10比较，18>10，将18与10对调，继续判断18与25的大小。 所以总共比较2次
综上：选择B选项

4、

答案解析：A
根据关键字序列得到的小顶堆的二叉树形式如下图所示



插入关键字3时，先将其放在小顶堆的末端，如图(2)所示。再将该关键字向上进行调整，得到的结果如图(3)所示。所以，调整后的小顶堆序列为3，5，12，8，28，20，15，22，19
综上：选择A选项

5、

答案解析： A
1.建堆时间复杂度：O(n)
2.调整堆、插入、删除：O(log2n)
3.堆排序平均时间复杂度：O(nlog2n)
4.空间复杂度：O(1)
综上：选择A选项

二、编程题

1、**【解题思路】：**

此题主要考察字符串的统计操作，为了判断两个字符串是否互为字母异位词，实际上就是判断两个字符串中各自出现字符的次数是否相等，因此使用一个 hash表进行统计，s每出现一个字符，hash对应位置加1，t每出现一个字符，hash对应位置减1，最终判断hash表的值即可，如果全为0，说明次数相等，否则不等。

【代码实现】：

```
bool isAnagram(char *s, char *t)
{
    if(strlen(s) != strlen(t))
        return false;

    //26个字母哈希映射表
    int hash[26] = {0};

    //统计s字符的次数++， 统计t字符的次数--
    for(int i=0; i<strlen(s); ++i)
    {
        hash[s[i]-'a']++;
        hash[t[i]-'a']--;
    }

    //检查hash,如果出现不为0的数字，说明s和t中的某个字符出现的次数不对等
    for(int i=0; i<26; ++i)
    {
        if(hash[i] != 0)
            return false;
    }
    return true;
}
```

2、**【解题思路】：**

此题主要考察字符串的简单操作，判断两个字符串是否相似，只需针对字符串的左右两个部分进行元音字母的个数统计，最后判断是否相等即可。

【代码实现】：

```
//判断是否为元音字母
int is_a_vowel(char c)
{
    return (c=='a'||c=='A'||c=='e'||c=='E'||c=='i'||c=='I'||c=='o'||c=='O'||c=='u'||c=='U' ? 1 : 0);
}

bool halvesAreAlike(char *s)
{
    int left_cnt = 0, right_cnt = 0;
    //将字符串分为左右两个部分,i为左部分下标，j为右部分下标
    for(int i=0, j=strlen(s)-1; i<j; ++i,--j)
    {
        left_cnt += is_a_vowel(s[i]);
        right_cnt += is_a_vowel(s[j]);
```

```
    }
    return left_cnt == right_cnt;
}
```

day02

一、选择题

1、

答案解析：D
参考下图排序算法当中的时间复杂度， 空间复杂度， 以及稳定性。
综上：选择D选项

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况		
插入排序	插入排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	Shell 排序	O(N^1.3)	O(N)	O(N^2)	O(1)	不稳定
选择排序	选择排序	O(N^2)	O(N^2)	O(N^2)	O(1)	不稳定
	堆排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定
交换排序	冒泡排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	快速排序	O(N*lgN)	O(N*lgN)	O(N^2)	O(lgN)	不稳定
归并排序	归并排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(N)	稳定

2、

答案解析：C
A选项：选择排序是每次选择未排序子列中最大（最小）的放到最后，显然4不是最值，所以A不对；
B选项：冒泡排序是相邻两两比较，把最大的顶上去，显然边上两个元素不是最值，所以B也不对；
C选项：插入排序是从第二项开始与前面每一项比较，如果小于那一项，则插入那一项前面，C中第二项9比15小，所以放到15前面。
D选项：希尔排序是先分组，然后针对组内采取插入排序，如果是希尔排序，那么9和15颠倒，20和-1也应该颠倒，所以D也不对；
综上：选择C选项

3、

答案解析：C
直接插入排序(straight insertion sort)的做法是：
每次从无序表中取出第一个元素，把它插入到有序表的合适位置，使有序表仍然有序。

大概是:24直接放进去　　24
第一趟:15比24小放到24前面，比较1次　　15 24
第二趟:32比24大放24后面，比较1次　　15 24 32
第三趟:28比32小，比24大，比较2次　　15 24 28 32
第四趟:19比32小，比28小，比24小，比15大，比较4次　　15 19 24 28 32
综上：选择C选项

4、

答案解析：B
参考下图排序算法当中的时间复杂度， 空间复杂度， 以及稳定性。
综上：选择D选项

类别	排序方法	时间复杂度			空间复杂度	稳定性
		平均情况	最好情况	最坏情况		
插入排序	插入排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	Shell 排序	O(N^1.3)	O(N)	O(N^2)	O(1)	不稳定
选择排序	选择排序	O(N^2)	O(N^2)	O(N^2)	O(1)	不稳定
	堆排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定
交换排序	冒泡排序	O(N^2)	O(N)	O(N^2)	O(1)	稳定
	快速排序	O(N*lgN)	O(N*lgN)	O(N^2)	O(lgN)	不稳定
归并排序	归并排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(N)	稳定

5、

答案解析：A、B、C、D
堆排序的时间复杂度， 空间复杂度， 稳定性参考考如下图：

堆排序	O(N*lgN)	O(N*lgN)	O(N*lgN)	O(1)	不稳定

二、编程题

1、**【解题思路】：**

此题主要考察对字符串的计数，可以使用一个数组，保存`word1`的每一个字符出现的次数，每出现一个字符就把相应的字母次数加1，相反的，只要`word2`的每一个字符出现，就把相应字母次数减1，最后判断数组，只要出现次数超过3的，即不相等。

【代码实现】：

```
bool checkAlmostEquivalent(char *word1, char *word2)
{
    //用来记录字母出现次数
    int hash[26] = {0};
    for(int i = 0; i<strlen(word1); ++i)
    {
        //word1出现的字母就加1
        hash[word1[i] - 'a']++;

        //word2出现的字母就相应的减1
        hash[word2[i] - 'a']--;
    }

    for(int i = 0;i < 26;++i)
    {
        //判断绝对值数是否大于3
        if(abs(hash[i]) > 3)
            return false;
    }
    return true;
}
```

2、**【解题思路】：**

此题主要考察字符串的操作，可以借助栈结构。用数组模拟栈，将数字和其余字符分别入栈。以'['和']'为界限，当'['时，说明前面的是数字，此时将数字入数字栈。 当']'时，说明一个括号内已经走完，将此中字符出栈，其次数的数字也出栈。 注意此时为字符逆序，所以需要翻转再入栈。

【代码实现】：

```
//字符串栈的最大空间数
#define L_MAX_SIZE 10000

//数字栈的最大空间数
#define N_MAX_SIZE 100

//反转字符串
void reverse(char *str)
{
    int len = strlen(str);
    for (int i = 0, j = len - 1; i < j; i++, j--)
    {
        char tmp = str[i];
        str[i] = str[j];
        str[j] = tmp;
    }
}

char * decodeString(char *s)

{
```

```
int nTop = -1, lTop = -1;
char lStack[L_MAX_SIZE] = ""; //字符串栈
int nStack[N_MAX_SIZE] = {0}; //数字栈
int numTmp = 0; //存储左括号[的左边数字

for (int i = 0; i < strlen(s); i++)
{
    if (s[i] == '[')
    {
        nStack[++nTop] = numTmp;
        numTmp = 0;
        lStack[++lTop] = s[i];
        continue;
    }
    if (s[i] >= '0' && s[i] <= '9')
    {
        numTmp = numTmp * 10 + s[i] - '0';
        continue;
    }
    if (s[i] != ']')
    {
        lStack[++lTop] = s[i];
        continue;
    }

    //临时字符串
    char tmpAlph[1000] = "";
    int k = 0;
    while (lTop != -1 && lStack[lTop] != '[')
    {
        tmpAlph[k++] = lStack[lTop--];
    }

    //将'['出栈, 同时注意栈中数据为逆序，需要反转
    lTop--;
    reverse(tmpAlph);

    //重复次数
    int times = nStack[nTop--];
    for (int t = 0; t < times; t++)
    {
        for (int j = 0; j < k; j++)
        {
            //组合times次数的字符串
            lStack[++lTop] = tmpAlph[j];
        }
    }
}
char* ans = (char*)calloc(sizeof(char), L_MAX_SIZE);
// 注意不能直接strcpy，因为lStack可能不只是答案，因为lTop中间的值可能是++又--过了
// 因此，有效字符应为存储在lStack中，长度为lTop + 1的字符串
memcpy(ans, lStack, sizeof(char) * (lTop + 1));

return ans;
```

```
}
```

day03

一、选择题

1、

```
答案解析：A
函数重载的概念：在相同作用域中，多个函数的函数名相同，参数列表不同，即可构成重载。
        参数列表不同体现在：参数个数不同、参数类型不同、类型次序不同
        与返回值类型是否相同无关
```

2、

```
答案解析：E
A：引用是给已存在变量取别名，必须在定义时初始化，否则报错；指针没有要求，一般建议没有合适指向最好指向
  NULL，A选项正确
B: 引用一旦和某个实体结合后，就不能引用其他的实体，语法上没有办法实现，普通类型的指针可以指向任意相同
  类型的变量。
  int a = 10, b = 20;
  int* p1 = &a;  // p1是普通类型的指针，既可以指向a
  p1 = &b;      // 也可以指向B

  int* const p2 = &a; // const限制指针本身，p2的指向不能在修改了
  p2 = &b;     // 编译报错
  B选项正确
C：引用在定义时必须初始化，因此没有空引用一说，但是指针没有合法指向时最好指向NULL，故C选项正确。
D：从概念上来讲，引用就是别名，故D选项正确。
E：引用的底层是用指针实现的，如果函数的参数是引用类型的，代码表层看起来是以值的方式来传递的，其实背后
  也是按照指针的方式传递的。故E错误
F：在C++中，总共有三种方式传参：传地址、传值、传引用
```

3、

```
答案解析：D
首先：从语法角度来讲，没有任何语法问题，故代码可以通过编译，A选项是错误的。
其次：const修饰的内容在C++中已经是常量了，并且在代码编译时具有宏替换的属性
因此：上述代码编译完成后，已变成一下内容
int main(void)
{
   const int a = 10;
   int * p = (int *)(&a);    // p指向a
   *p = 20;              // a是常量无法直接替换，但是通过指针p是可以的，此时a被修改为20
   cout<<"a = "<<10<<", *p = "<<*p<<endl;  // 因此打印a=10, *p = 20
   return 0;
}
故选择D
```

4、

答案解析：B
类大小的计算方式：将类中非静态成员变量相加，注意内存对齐即可
空类中不包含任何成员变量，表面看空类大小应该为0，但实际在vusal studio、g++等主流编译器中空类大小都是1
如果空类的大小是0，会存在以下问题：

```
  A   a1, a2;
  |_______|
  |_______|
  |_______|a1  a2
  |_______|
```

假设空类大小为0，那a1和a2将来就在同一个位置存储，编译器就无法区分a1和a2对象了，因此主流编译器中将空类大小设置的是, 故选择B

5、

答案解析：C
为了在成员函数中区分：到底是那个对象在调用该成员函数，成员函数内部如何区分正在操作那个对象中的成员变量，编译器给非静态成员函数自动添加了一个隐藏的this指针，该指针无需用户自己传递，编译器会自定传递的。
因此只有成员函数才具有隐藏的this指针，故A和B错误
由于静态成员函数不需要通过对象调用，因此也没有隐藏的this指针，即：只有类的非静态成员函数才有隐藏的this指针，故D错误

二、编程题

1、**【解题思路】**

此题主要考察对字符串的操作，从原字符串中找到每个单词，然后使用reverse方法进行逆转，具体如下：

1. 通过字符串的find方法，从start位置开始往后找最近的一个空格：即pos = s.find(start, ' '); [start, pos)就表示一个单词。
2. 使用标准库提供的reverse(begin, end)方法，对[start, pos)区间中的单词进行翻转
3. 修改start到下一个单词的其实位置：即start = pos+1
4. 如果start没有走到字符串的末尾，循环1 2 3

【代码实现】

```
class Solution {
public:
    string reverseWords(string s)
    {
        // [start, pos)标记一个单词
        int start = 0;
        int pos = 0;
        while (start < s.size())
        {
            // 从start位置开始，从前往后找最近的一个空格
            pos = s.find(' ', start);

            // 如果pos是-1，说明[start, s.size())刚好是最后一个单词，调整下pos
            if (pos == -1)
                pos = s.size();

            // 逆置[start, pos)表示的单词
```

```
            std::reverse(s.begin()+start, s.begin() + pos);

            // 将start挪动到下一个单词的起始位置
            start = pos + 1;
        }

        return s;
    }
};
```

2、**【解题思路】**

统计相邻并相同字符出现的次数，然后保存最大值即可

【代码实现】

```
class Solution
{
public:
    int maxPower(string s)
    {
        int chCount = 1, maxCount = 1;
        for (int i = 1; i < s.size(); ++i)
        {
            // 如果两个字符相同，给该字符出现次数+1
            if (s[i] == s[i - 1])
            {
                ++chCount;
            }
            else
            {
                // 如果两个字符不相等，说明s[i]是第一次出现，将才会Count设置为1
                chCount = 1;
            }

            // 获取最大次数
            if(chCount > maxCount)
                maxCount = chCount;
        }
        return maxCount;
    }
};
```


day04

一、选择题

1、

答案解析：
A：错误，析构函数既可以在类内定义，也可以在类外定义，在类外定义时，函数名前需加 类名::
B：正确，因为析构函数没有参数，即无法构成重载，故一个类中最多只能定义一个析构函数
C：错误，析构函数名是在类名前添加~符号，表示析构函数功能与构造函数功能相反
D: 错误，析构函数不能显式定义任何参数；都已经要死了，还要啥自行车

2、

答案解析：B
A：错误，析构函数主要是释放对象中管理的资源的，并不是所有对象都管理资源，所以不存在内存泄漏一说
B: 正确，对象销毁时，编译器会自动调用析构函数，delete this时，也会调用析构函数，形成无限递归导致栈溢出
C：错误，代码没有语法问题，可以通过编译
D：错误，析构函数负责释放对象中管理的资源，对象本身的空间不是由析构函数负责释放的

3、

答案解析：B
对象的构造和析构一般遵循：早构造的晚析构
构造的次序为：c a b d
全局对象和static修饰的对象，生命周期与程序一直，程序在退出时才会销毁
局部对象是在函数退出的时候销毁的，new出来的对象在delete时销毁
因此：最早析构的是A 其次是B 在然后是 D 最后是C

4、

答案解析：A
先创建对象a，再创建对象b，再创建包含3个AB对象的数组c
p是个指针数组，里面放的是对象的地址，并没有创建新的对象
故创建对象的个数应该是5个

5、

答案解析：
A：正确，全局和static修饰的变量都在数据段
B：正确，静态成员可以通过类名或者对象访问，也可以在成员函数中直接访问
C：错误，非静态成员在成员函数中最终都是通过this访问的，而静态成员函数没有this指针，故无法访问费静态成员
D：正确，非静态成员变量最终保存在对象中，对象销毁了其也不存在了；静态成员变量在程序启动时就创建了，程序退出时候被销毁

二、编程题

1、**【解题思路】**

循环进行一下操作，对字符串进行压缩

1. 统计连在一起的相同字符出现的次数
2. 将字符以及该字符出现的次数追加到新字符串中

注意：压缩完成后，如果结果比原字符还要长，则不进行压缩

【代码实现】

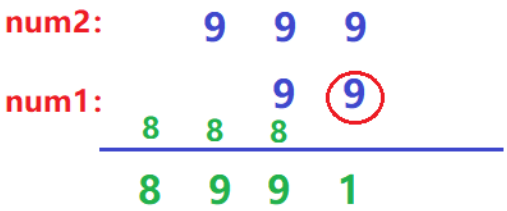
```
class Solution {
public:
    string compressString(string S) {
        string ret;
        int count = 1;
        for (int i = 0; i < S.size(); i ++) {
            // 统计相同字符出现的次数
            if (i+1 < S.size() && S[i] == S[i+1]) {
                count++;
                continue;
            }

            // 将结果保存起来
            ret.push_back(S[i]);
            ret.append(to_string(count));

            count = 1;
        }

        // 注意：压缩后结果比源字符串还长则不压缩
        return ret.size() >= S.size() ? S : ret;
    }
};
```

2、**【解题思路】**

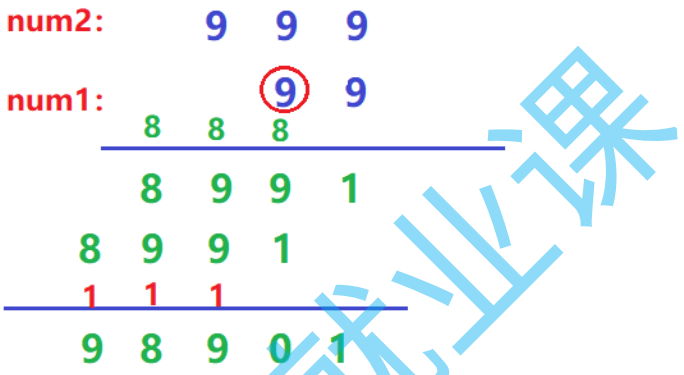
此题是用字符串来模拟实现乘法，常出现在大数运算中，即数据无法用内置类型表示时，采用字符串来存储数字，最终需要用字符串来模拟实现加减乘除，即大数的四则运算。在面试中也多次被考察到。大概思路如下:

1. 特殊情况考虑：如果两个数据中有一个为0，直接返回"0"
2. 如果都不为0，按照如下方式模拟乘法运算：

第一次循环：取num1中的最低位9，与num2中的每一位相乘，有进位时向前进位



第二次循环：取num1中次低位9，与num2中的每一位相乘，有进位时向前进位，注意：此时需要与前一次乘完的结果错位相加



【注意】

1. 最好使用位数较短的作为外层循环
2. 两个数相乘，结果最多有num1.size() + num2.size()位，比如：999*99=98901，结果有5位数；如果结果有4位置，最后需要将最高位前0删除掉，比如：100 * 99 = 9900
3. 本次乘完后与上一次结果相加时，需要有错位

【代码实现】

```
class Solution {
public:
    string multiply(string num1, string num2) {
    // 1. 特殊情况处理，如果有一个是0，结果肯定是0
    if(num1 == "0" || num2 == "0")
        return "0";

    // 2. 保证num1比num2短，将来使用num1作为外循环
    int LSize = num1.size();
    int RSize = num2.size();
    if(LSize > RSize)
    {
        num1.swap(num2);
        swap(LSize, RSize);
    }
```

```
    // 3. LSize位 * RSize位，结果最多有LSIze+RSize位，比如：99*99 结果最多为4位
    string strRet(LSize+RSize, '0');

    // 4. 用字符串模拟乘法运算：从num1中取一位，与num2中的每一位相乘，乘完之后结果与上一趟相加
    int offset = 0;  // 相加的时候注意偏移
    for(int i = LSize-1; i >= 0; --i)
    {
        // 取到num1的每一位
        int data = num1[i] - '0';
        char step = 0;

        int retIdx = strRet.size() - 1 - offset;

        for(int j = RSize-1; j >= 0; --j)
        {
            // 使用data与num2的每一位从右往左依次相乘，错位加上上一次结果
            int ret = (num2[j]-'0') * data;
            ret =ret +  strRet[retIdx] - '0';
            step = 0;
             if(ret >= 10)
            {
                step = ret/10;
                ret %= 10;
            }

            strRet[retIdx] = ret + '0';    // 结果保存在当前位
            strRet[retIdx-1] += step;     // 进位需与前一位相加
            retIdx--;
        }

        // 下一趟需要往前偏移一位相加
        offset++;
    }

    // 如果结果中多了一位，将前置的0删除掉
    if (strRet[0] == '0')
        strRet.erase(strRet.begin());

        return strRet;
    }
};
```


day05

一、选择题

1、

答案解析：D
A：正确，友元的作用
B：正确
C：正确，严格说此题中的成员函数只非静态成员函数，非静态成员函数必须要通过对象调用，调用时都需要传递this
D：错误，友元函数不是类的成员函数，不需要通过对象调用，即不用传递this

2、

答案解析：
在进入main函数之前，所有的全局变量以及全局对象已经被创建好了，然后才执行main函数中的代码
故在进入main函数前，需要先调用构造函数创建全局对象

3、

答案解析：A
对于自定义对象，无法直接使用已存在的操作符直接操作，比如：日期类型对象直接比较大小，
因为一般对象中包含有多个成员变量，编译器无法知道到底按照什么规则进行对象的比较，故需要进行响应运算符的重载，即告诉编译器，对象遇到对应操作符时，应该如何操作。如果将运算符重载为类的成员函数，第一个参数实际为隐藏的this指针。
由于+具有两个操作数，重载成成员函数后，可以少给一个形参，因为还有隐藏的this，故A正确

4、

答案解析：C
A：正确，malloc/free是C语言标准库提供的库函数，使用时需要包含对应头文件，而new/delete是运算符，或者关键字，可以直接使用，不需要包含任何头文件
B：正确，new分为两步-->1. 调用void* operator new(size_t size)申请空间 2. 调用构造函数初始化空间内容
C：错误，malloc和free在C++中也可以使用，只不过申请和释放对象的空间时最好使用new和delete
D：正确，new操作成功之后，对象就创建好了，直接用对应的指针接收，不需要强转，而malloc返回的是void*，接收返回值时必须要强转

5、

答案解析：
A：正确，编译器本来就支持int类型数据直接加减
B：正确，+必须有两个操作数，被重载成类的成员函数时，看起来只有一个参数，实际还有一个隐藏的this指针，并且是该函数的第一个参数，因此在调用时+的左侧必须是BigNumber类型的对象
C：正确，参考B解析
D：错误，参考B解析

二、编程题

1、**【思路解析】**

此题要充分利用数组有序的特性，给两个下标left和right，left在区间左侧，right在区间最右侧，当left<right时，循环进行以下操作：

1. 使用left和right位置的数据相加，保存到sum中
2. 如果sum == target，则找到了，保存left和right并返回

如果sum < target：则left++往后移动

如果sum > target：则right--往前移动

【代码实现】

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int left = 0;
        int right = numbers.size()-1;
        vector<int> ret;
        while(left < right)
        {
            int sum = numbers[left] + numbers[right];
            if(sum == target)
            {
                ret.push_back(left);
                ret.push_back(right);
                break;
            }
            else if(sum < target)
            {
                left++;
            }
            else
            {
                right--;
            }
        }

        return ret;
    }
};
```


2、**【解题思路】**

该题在往年笔试面试中非常常见，题目比较简单，但是要求时间复杂度的话稍微有点点难度。

- **方式一**

  如果存在出现超过一半的元素，则排序后该元素一定在数组的最中间

  1. 利用STL提供的sort方法直接排序
  2. 找到最中间的元素，即：numbers.size()/2
  3. 统计该元素出现的总次数，如果超过一半输出该元素，否则输出0

  时间复杂度：O(NlogN)

- **方式二**

  使用map统计每个数字出现的次数，如果哪个数字出现的次数超过了元素总个数的一半，则找到了该数据。

  时间复杂度：O(NlogN)

- **方式三**

设计一个计数count，记录x数据出现的次数。遍历数组，拿到numbers[i]后，进行如下操作：

1. 如果count等于0，表明该元素第一次出现，使用x记录该元素，并将count设置为1
2. 否则：如果numbers[i] == x，count++；否则count--

上述操作结束后，x中标记的元素可能是出现次数刚好超过一般的元素。最后在遍历一遍次数，确认x是否真的超过一半。

时间复杂度：O(N)

【注意】

在oj中，上述三种方式都可以，从时间复杂度上来看，方式三最佳。如果笔试时想不到方式三，可按照方式一或者方式二临时应付也是可以的。但是方式三必须要掌握，在面试中被考察时，面试官最想看到的是方式三，如果按照方式一或者方式二实现，面试官可能会让从时间复杂度上来优化算法。

【代码实现】

```
// 方式一：
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        // 因为用到了sort，时间复杂度O(NlogN)，并非最优
        if(numbers.empty())
            return 0;

        sort(numbers.begin(),numbers.end()); // 排序，取数组中间那个数
        int middle = numbers[numbers.size()/2];

        int count=0; // 出现次数
        for(int i=0;i<numbers.size();++i)
        {
            if(numbers[i]==middle) ++count;
        }

        return (count>numbers.size()/2) ? middle : 0;
    }
};

// 方式二：
// 因为map底层是红黑树，所以在统计每个元素出现次数时的时间复杂度为O(NlogN)
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        int n = numbers.size();
        //map 记录出现次数
        map<int, int> m;
        int count;
        for (int i = 0; i < n; i++) {
            count = ++m[numbers[i]];    // 统计numbers[i]数组出现的次数

            // 如果该数字出现次数超过了数组总大小的一半，则找到了
            if (count > n/2)
                return numbers[i];
        }
```

```
        return 0;
    }
};

// 方式三:
class Solution {
public:
    int MoreThanHalfNum_Solution(vector<int> numbers) {
        if(numbers.empty())
            return 0;

        int count = 0;
        int x = numbers[0];
        for(int i = 0; i < numbers.size(); ++i)
        {
            if(count == 0)
            {
                x = numbers[i];
                count = 1;
            }
            else
            {
                if(numbers[i] == x)
                    ++count;
                else
                    --count;
            }
        }

        count = 0;
        for(int i = 0; i < numbers.size(); ++i)
        {
            if(numbers[i] == x)
                ++count;
        }

        return (count * 2 > numbers.size() ? x : 0);
    }
};
```


day06

一、选择题

1、

答案解析：
A：错误，类模板的主要作用是让编译器按照用户所写的格式借助实例化来生成代码，比如有一个vector的模板类
vector<int> v1; // 编译器根据实例化结果会生成一份专门处理int类型的代码
vector<double> v2; // 编译器根据实例化结果会生成一份专门处理double类型的代码
B：正确，参考A选项
C：错误，数据成员的类型可以不容，这个要根据具体的应用场景来，比如：封装一个泛型的顺序表
D：错误，成员函数是否需要返回值，需要根据应用场景来

2、

答案解析：C
A：错误，类模板没有隐式实例化，只能显式实例化，比如：sum<int> s; sum<int>才是真正的类名，sum只是类的名字，故不能使用sum::直接调用静态成员函数，故编译失败
B：错误，去掉static之后，foo必须通过对象调用，不对了
C：正确，sum是类模板，并不是真正的类名，必须显式实例化，即C的写法
D：错误，<int>必须跟在sum之后，语法格式错误

3、

答案解析：C
A：正确，STL容器不是线程安全的，可以参考上课的模拟实现或者侯捷编写《STL源码剖析》
B：正确，vs中是按照2倍方式扩容的；linux下是按照1.5倍方式扩容的
C：错误：sort中使用了快排、堆排、插入排序算法复合起来的，因此不稳定
D：正确：stack的底层就是使用使用deque实现的

4、

答案解析：C
A：错误，stack 和 queue 底层默认都是使用 deque实现的
B：错误，stack 只能获取栈顶的元素，queue只能获取队头和队尾的元素，不能获取任意位置
C：正确，stack 和 queue 中模板参数列表的第二个参数是底层使用容器，默认使用deque
D：错误，stack 和 queue 插入和删除的时间复杂度为O(1)，注意它们没有遍历的操作

5、

答案解析：D
A：正确，vector底层是连续空间，类似于数组，支持随机访问
B：正确，list底层是带头结点双向循环链表，任意位置插入和删除只需要修改指针指向即可，时间复杂度O(1)
C：正确，vector底层是连续空间，其迭代器上实际为T*类型的指针，故上述操作均支持
D：错误，list底层是链表结构，链表不支持随机访问，因此list中没有提供对[]的重载

二、编程题

1、**【答案解析】**

该题思路有点类似于快排的分割，找到满足条件的数据，偶数位上找奇数，奇数位上找偶数，然后直接进行交换。

设置两个下标，i和j，如果i和j小于nums大小时,循环进行以下操作：

1. 让i从前往后在偶数位上找奇数
2. 让j从前往后在奇数位上找偶数

3. 如果找到了，则交换i和j位置上的数据

【代码实现】

```
class Solution {
public:
    vector<int> sortArrayByParityII(vector<int>& nums) {
        int i = 0,j = 1;
        int n = nums.size();
        while(i < n && j < n)
        {
            // 偶数位上找奇数
            while(i < n && 0 == (nums[i]&1))
                i += 2;

            // 奇数位上找偶数
            while(j < n && 0 != (nums[j]&1))
                j+=2;

            if(i >= n || j >= n)
                break;
            swap(nums[i], nums[j]);
        }

        return nums;
    }
};
```


2、**【解题思路】**

看到这道题，最直观的方式就是枚举出数组的所有子数组，并求出其和，然后找到最大的结果。一个长度为n的数组，其子数组总共有n*(n+1)/2个子数组，计算出所有子数组的和，时间复杂度为O(N^2), 通常不会考虑。

解法一：直接遍历

比如对于数组：{1，-2, 3, 10，-4, 7, 2，-5}，采用PreSum保存上一次子数组最大和，采用CurSum保存本次子数组最大和。

第一步：加上一个数字1，此时和为1

第二步：加上数字-2，和变成-1

第三步：加上数字3，和变成2

注意观察：由于此前累计的和是-1，如果用-1加上3，得到的和是2，比3本身还小，也就是说，从第一个数字开始的子数组的和会小于从第三个数字开始的子数组的和，因此不考虑从第一个数字开始的子数组，之前累计的和也被抛弃。

步骤	操作	累加的子数组和	最大的子数组和
1	加1	1	1
2	加-2	-1	1
3	抛弃前面的和-1，加3	3	3
4	加 10	13	13
5	加 -4	9	13
6	加 7	16	16
7	加 2	18	18
8	加 -5	13	18

从第三个数字重新开始累加，此时和为3，接下来：

第四步：加10，得到和为13

第五步：加-4，得到和为9，由于-4是一个负数，累加之后得到和比原来和还要小，因此需要将以前和保存下来，因为它可能为最大子数组的和。

第六步：加 7，得到和为16

第七步：加2，得到和为18，超过以前的最大和，此时更新以前的最大和

第八步：加-5， 得带和为13， 小于此前最大和，抛弃

结束

方法总结：

给两个变量CurSum和MaxSum，CurSum记录当前元素位置的最大和，MaxSum记录目前统计过的子数组最大和。

遍历数组，依次取数组中的元素，按照以下操作进行：

如果CurSum小于0，说明前面子数组累加完是负数，从当前元素位置开始，重新累加

如果CurSum大于等于0，直接在上一次结果上累加，因为加上本次结果后，可能会比上次更大

每次累加完成后，检测本次累加结果是否超过前面子数组的最大和MaxSum，如果超过则更新

【代码实现】

```
// 方法一:
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array)
    {
        int MaxSum = 0xffffffff;  // 保存子数组最大和

        int CurSum = 0;          // 保存当前统计子数组和
```

```
        for(size_t i = 0; i < array.size(); ++i)
        {
            // 前一个元素累加完之后，CurSum和小于0，抛弃前面结果，从当前位置开始累加
            if(CurSum < 0)
            {
                CurSum = array[i];
            }
            else
            {
                // 如果前一次累加完成之后，结果不小于0，一直往上累加
                CurSum += array[i];
            }

            // 每次累加完成之后，更新MaxSum
            if(CurSum > MaxSum)
                MaxSum = CurSum;
        }

        return MaxSum;
    }
};

// 方法二：动态规划
// 假设使用F[i]表示i结尾的子数组的最大和，a为数组的名字，根据方式一的理解有以下表达式：
//     F(i) = max(F(i-1)+a[i], a[i])

class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        // F用来保存以i结果的子数组的最大和
        vector<int> F(array.size());
        F[0] = array[0];
        for(size_t i = 0; i < array.size(); ++i)
        {
            if(F[i-1] + array[i] > array[i])
                F[i] = F[i-1] + array[i];
            else
                F[i] = array[i];
        }

        // 找出统计结果中的最大和
        int MaxSum = F[0];
        for(size_t i = 1; i < F.size(); ++i)
        {
            if(F[i] > MaxSum)
                MaxSum = F[i];
        }

        return MaxSum;
    }
};
```

```
// 方法三：
// 方式二缺陷：其实没有必要给一个数组保存每次加上之后的子数组状态，给一个变量来进行标记即可。
class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        int MaxSum = array[0];
        int CurSum = array[0];

        for(size_t i = 1; i < array.size(); ++i)
        {
            CurSum = CurSum + array[i];
            if(array[i] > CurSum)
                CurSum = array[i];

            if(CurSum > MaxSum)
                MaxSum = CurSum;
        }

        return MaxSum;
    }
};
```

day07

一、选择题

1、

答案解析：A
A：错误，vector的插入可能会导致迭代器失效，因为插入时可能会导致扩容，一旦扩容之前的迭代器就会失效
B：正确：map底层的红黑树采用的是链式结构存储的，插入元素不会对其他节点产生任何影响，故不会失效
C：正确：以为vector删除后，后序元素需要向前搬移，在vs下删除位置及以后的迭代器都失效了
D：正确：map删除时，不会对其他节点产生任何影响，因此其他位置迭代器不会失效

2、

答案解析：D
要在类外访问，基类中的成员必须要是公有的。故排除A和C
私有继承方式，基类中公有的成员在子类中的权限也降级为私有的，因此也不能通过子类对象直接在类外访问，C错误
故只能选择D

3、

答案解析：C
A：正确，public继承方式，派生类和基类是is-a关系，即子类对象可以看成是一个基类对象
B：正确，基类引用可以引用子类对象，因为子类对象模型上部分和基类的对象模型是完全相同的，或者参考A解析
C：错误，如果基类成员是private的，在子类中不可见，即不可以访问
D：正确，参考B

4、

答案解析：B
Eye是眼睛，Head是头，头上长有眼睛，即Head中有一对眼睛，has-a使用组合，is-a使用继承，故选择B

5、

答案解析：
A：不可以，假设构造函数可以作为虚函数，那必须通过虚表找到构造函数的地址，而虚表地址是在构造函数中才放入对象前4个字节的，构造函数没有执行，对象都不完整，找不到虚表就无法调用构造函数，前后矛盾了，故构造函数不能作为虚函数
B：可以，如果子类中涉及到资源管理时，最好将基类中析构函数设置为虚函数，否则可能会存在资源泄漏
C：语法上可以，但是没有意义，因为内联函数在编译时会展开，展开了之后代码就确定了，就无法实现动态调用了
D：不可以，因为静态成员函数没有this指针，可以不通过对象调用，如果直接使用类名::方式调用，那就无法拿到虚表，因为虚表在对象的前4个字节

二、编程题

1、**【解题思路】**

此题比较简单，主要是考察同学们二维vector的使用。

现假设以为数组为：1 2 3 4 5 6 7 8

每个元素对应下标：0 1 2 3 4 5 6 7

假设转换为两行撕裂的位数数组，结果为：

第0行：1 2 3 4

第1行：5 6 7 8

假设以为数组中元素的下标为i，仔细观察，会发现其在二维数组中的行下标为：i/n，列下表为：i%n

大概思路如下：

1. 检测一维数组original是否可以转化为m*n的二维数组，不行则返回一个空的vector<vector>()
2. 循环获取数组中的每个元素，利用该元素在一维数组中的下标计算其在二维数组中的行和列下标，然后将元素放到二维数组对应的位置

【代码实现】

```
class Solution {
public:
    vector<vector<int>> construct2DArray(vector<int>& original, int m, int n) {
        if(m*n != original.size())
            return vector<vector<int>>();

        vector<vector<int>> ret(m, vector<int>(n, 0));
        for(int i = 0; i < original.size(); ++i)
        {
            ret[i/n][i%n] = original[i];
        }

        return ret;

    }
```

```
};
```

2、**【解题思路】**

通过观察所给用例，发现：

依次将[1, target[target.size()]]范围中的每个数字Push，每次Push完之后，检测该元素与target[index]是否相等，相等则保留，取target中的下一个数字；否则说明本次Push的元素不在target数组中，将该元素Pop掉即可。

【代码实现】

```
class Solution {
public:
    vector<string> buildArray(vector<int>& target, int n) {
        vector<string> vret;

        int index = 0;
        int data = 1;
        while(index < target.size())
        {
            vret.push_back("Push");
            if(data != target[index])
                vret.push_back("Pop");
            else
                index++;

            data++;
        }
        return vret;
    }
};
```


day08

一、选择题

1、

```
答案解析：D
子类要重写基类的虚函数必须：
1. 基类的成员函数必须是虚函数，即前面必须加virtual关键字
2. 函数的原型必须一致：返回值类型  函数名(参数列表)必须要一直，有两个例外
    a. 析构函数----子类与基类析构函数函数名不同
    b. 协变---基类虚函数返回基类的指针或引用，子类虚函数返回子类的指针或引用----返回值类型不一样
3. 访问权限可以不同
4. 子类重写基类虚函数时，前面可以不加virtual，一般推荐写上
故选择D
```

2、

答案解析：C
A：错误，static成员变量是程序启动时就创建好了
B：错误，静态成员函数可以在普通成员函数中使用的，只要是类的成员函数，都可以调用静态成员函数
C：正确，静态成员函数没有this指针，所以不能作为虚函数
D：静态成员变量可以在任意成员函数中访问

3、

答案解析：A
如果子类中涉及到资源管理时，最好将基类的析构函数设置为虚函数，否则当基类指针指向new出来的子类对象时，子类对象中的资源将无法完整释放

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

    virtual ~A()
    {
        cout<<"~A()"<<endl;
    }
};

class B : public A
{
public:
    B()
    {
        cout<<"B()"<<endl;
        p = new int[10];
    }

    ~B()
    {
        cout<<"~B()"<<endl;
        delete[] p;
    }
    int* p;
};

int main()
{
    A* pa = new B();
    delete pa;
    return 0;
}
```

如果将基类析构函数前virtual去掉之后，会发现当delete pa时，子类的析构函数没有调用，即pa指向的子类对象中的资源

并没有被正确释放。

4、

答案解析：D
抽象类特性：不能实例化对象，但是可以定义抽象类的指针或者引用
A：错误，函数参数时需要拷贝抽象类的对象
B：错误，抽象类不能实例化对象
C：错误，函数返回值需要拷贝抽象类的对象
D：正确，可以定义抽象类的指针

5、

答案解析：B
A类中foo是普通函数，fun是虚函数
B类中fun重写了A类中的虚函数
故：p->foo(); 由于p是A*，基类的指针，所有p->foo()始终调用的都是基类的foo，故foo调用都输出1
p->fun(); 基类指针调用fun虚函数，多态条件满足，运行时调用那个类的fun，取决于p指向那个类的对象

```
A a;
B b;
A *p = &a;   // p指向基类的对象
p->foo();    // 调用基类的普通foo函数，输出1
p->fun();    // 调用基类的fun虚函数，输出2
p = &b;      // p指向子类对象
p->foo();    // 调用基类的普通foo函数，输出1
p->fun();       // 调用子类的fun虚函数，输出4
A *ptr = (A *)&b; // ptr实际指向的仍旧是子类对象，此处的强转具有迷惑性，并不会产生新的对象
ptr->foo();      // 调用基类的普通foo函数，输出1
ptr->fun();      // 调用子类的fun虚函数，输出4
```

最后最终输出：121414

二、编程题

1、**【解题思路】**

1. 空树直接返回
2. 非空
   a. 在二叉搜索树中找到链表的首节点
   b. 要转化为有序的链表，二叉搜索树中序遍历刚好可以得到一个有序序列，因此采用中序遍历的规则进行转化，在转化过程中，只需要改变每个节点左右指针域的指向即可。

【代码实现】

```
class Solution {
public:
  // pPrev标记刚刚转化的节点，pRoot表示现在要转化的二叉树，pRoot前一个处理的节点是pPrev
  void _Convert(TreeNode* pRoot, TreeNode*& pPrev)
  {
    // 空树：不用转化，直接返回
    if(nullptr == pRoot)
      return;
```

```
        // 将pRoot的左子树转化为双向链表
        _Convert(pRoot->left, pPrev);

        // pRoot的left指向其前一个处理的节点，即pPrev
        // pRoot的right域没有办法在本次递归中处理，因为下一个节点不知道
        // 在本次中只能处理当前节点的left
        pRoot->left = pPrev;

        // 前一个节点的right指针域没有处理，right指针域指向后一个节点，即pRoot
        if(pPrev)
            pPrev->right = pRoot;

        pPrev = pRoot;

        // 将pRoot的右子树转化为双向链表
        _Convert(pRoot->right, pPrev);
    }

    TreeNode* Convert(TreeNode* pRootOfTree)
    {
        if(nullptr == pRootOfTree)
            return nullptr;

        // 找双向链表的第一个节点，即树中最小的节点
        TreeNode* pHead = pRootOfTree;
        while(pHead->left)
            pHead = pHead->left;

        // 使用prev标记刚刚转化过的节点
        TreeNode* prev = nullptr;
        _Convert(pRootOfTree, prev);
        return pHead;
    }
};
```


2、**【解题思路】**

如果在面试中被问到，可以对二叉树分情况和面试官讨论：

1. 如果二叉树采用孩子双亲表示法链接，求两个节点的最近公共祖先，实际就转化为两个链表相交求交点
2. 如果二叉树是二叉搜索树： a. 如果树是空，直接返回nullptr，没有最近公共祖先节点 b. 如果两个节点中有一个是根节点，最近公共祖先一定是根节点 c. 如果两个节点一个比根节点大，一个比根节点小，最近公共祖先一定是根节点 d. 如果两个节点都比根节点小，递归到根的左子树中查找 e. 如果两个节点都比根节点大，递归到根的右子树中查找
3. 如果是普通的二叉树 方法一：写一个判断节点是否在二叉树中的方法，可以参考类似二叉搜索树的方法解决 a. 如果树是空，直接返回nullptr，没有最近公共祖先节点 b. 如果两个节点中有一个是根节点，最近公共祖先一定是根节点 c. 如果两个节点一个在根节点的左子树，一个在根节点的右子树，最近公共祖先一定是根节点 d. 如果两个节点都在左子树中，递归到左子树中找 e. 如果两个节点都在右子树中，递归到右子树中找

方法二：受孩子双亲表示法启发，如果能够知道节点到根的路径，问题就解决了 获取节点pNode的路径，因为公共祖先从下往上找，因此将路径中的节点保存在栈中

a. 如果是空树，直接返回

b. 将根节点入栈，如果根节点和pNode是同一个节点，该节点到根的路径找到，否则：

c. 递归在根节点的左子树中查找，如果找到，返回

d. 如果在根节点的左子树中未找到，递归到根节点的右子树中找

e. 如果右子树中没有找到，说明root一定不再路径中

获取最近公共祖先

【方法一】

1. 实现一个检测节点是否在二叉树中的方法
2. 开始求最近公共祖先
   - 检测p和q是否有一个在root的位置，如果在最近公共祖先就是root
   - 检测p在root左右子树中的情况 再检测q在root左右子树中的情况
   - 如果p和q都在root的左子树中，说明最近公共祖先一定在root的左子树中，递归到root的左子树中求解
   - 如果p和q都在root的右子树中，说明最近公共祖先一定在root的右子树中，递归到root的右子树中求解
   - 否则：p和q分别再root的左右子树中，最近公共祖先一定就是root

【方法二】

1. 在二叉树中获取两个节点的路径 2. 如果两个路径中节点个数不一样，节点多的出栈，直到两个栈中元素相等相等时：再比较两个栈顶是不是同一个节点，如果是，最近公共祖先找到， 如果不是，两个栈同时进行出栈，继续比较，直到找到为止

【代码实现】

```
// 方式一:
class Solution {
public:
    bool isNodeInTree(TreeNode* pRoot, TreeNode* pNode)
    {
        // 空树：直接返回
        if(nullptr == pRoot || nullptr == pNode)
            return false;

        // 检测是否为根节点--如果是直接返回true
        if(pRoot == pNode)
            return true;

        // 现在根的左子树中检测，如果不在再到根的右子树中检测
        return isNodeInTree(pRoot->left, pNode) || isNodeInTree(pRoot->right, pNode);
    }

    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // 如果是空树，不存在最近公共祖先节点
        if(nullptr == root)
            return nullptr;

        // 只要p或者q有一个在根节点的位置，则最近公共祖先一定是根节点
```

```
        if(p == root || q == root)
            return root;

        bool pInLeft = false, pInRight = false;
        bool qInLeft = false, qInRight = false;
        // 检测p是否子啊root的左子树中
        if(isNodeInTree(root->left, p))
        {
            pInLeft = true;
            pInRight = false;
        }
        else
        {
            pInRight = false;
            pInRight = true;
        }

        // 检测q是否在root的左子树中
        if(isNodeInTree(root->left, q))
        {
            qInLeft = true;
            qInRight = false;
        }
        else
        {
            qInRight = false;
            qInRight = true;
        }

        // 根据检测p和q在root子树中的情况获取最近公共祖先
        if(pInLeft && qInLeft)   // p和q都在root的左子树中
        {
            return lowestCommonAncestor(root->left, p, q);
        }
        else if(pInRight && qInRight) // p和q都在root的右子树中
        {
            return lowestCommonAncestor(root->right, p, q);
        }
        else
        {
            // p和q分别在root的左右子树中，最近公共祖先一定是根节点
            return root;
        }
    }
};

// 方式二:
class Solution {
public:
    bool GetNodePath(TreeNode* pRoot, TreeNode* pNode, stack<TreeNode*>& path)
    {
        // 空树：直接返回

         if(nullptr == pRoot)
```

```
        return false;

    // 先将根节点放在路径中
     path.push(pRoot);

    // 如果根节点和pNode相等，路径找到
     if(pNode == pRoot)
        return true;

    // 如果根节点和pNode不相等，先递归在左子树中找，找到则退出
     bool isPath = false;
     if(isPath = GetNodePath(pRoot->left, pNode, path))
        return true;

    // 未找到时，再到根节点的右子树中找
     if(isPath = GetNodePath(pRoot->right, pNode, path))
        return true;

    // 如果左右子树中都没有找到，说明根节点不在路径中
     path.pop();
     return false;

}

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    // 如果是空树，不存在最近公共祖先节点
    if(nullptr == root)
        return nullptr;

    // 获取两个节点在二叉树中的路径，并保存在栈中
    stack<TreeNode*> pPath;
    stack<TreeNode*> qPath;
    GetNodePath(root, p, pPath);
    GetNodePath(root, q, qPath);

    // 找最近公共祖先
    size_t pSize = pPath.size();
    size_t qSize = qPath.size();
    TreeNode* pCommonAncestor = nullptr;
    while(pSize && qSize)
    {
        // 如果两个栈中元素不相等，长的先出栈，直到两个栈中元素相等
        if(pSize > qSize)
        {
            pPath.pop();
            pSize--;
        }
        else if(pSize < qSize)
        {
            qPath.pop();
            qSize--;
        }

        else
```

```
        {
            // 如果栈顶元素相等，即为最近公共祖先
            // 否则：两个栈同时出栈
            if(pPath.top() == qPath.top())
            {
                pCommonAncestor = pPath.top();
                break;
            }
            else
            {
                pPath.pop();
                qPath.pop();
                pSize--;
                qSize--;
            }
        }
    }

    return pCommonAncestor;
  }
};
```

day09

一、选择题

1、

答案解析：D
实现多态的条件：
1. 基类中必须有虚函数，并且子类必须重写基类的虚函数
2. 必须通过基类的指针或者引用调用虚函数
在运行时，根据基类指针或引用指向不用类的对象，调用对应类的虚函数

CChild类中已经将CParent类中的Intro和Hobby两个虚函数重写了，并且在main函数中Intro也是通过指针调用的，即多态的实现条件已经完满满足了，调用那个类的虚函数，只需看基类指针指向那个类的对象即可。
```
 CChild *pChild = new CChild();
 CParent *pParent = (CParent *) pChild; // 基类的指针本来就可以直接指向子类对象，此处强转具有迷惑性，并不会生成基类的对象，pParent指向的实际也是子类对象，
```
故pParent->Intro(); 最终调用的是子类的虚函数，故选择D

2、

```
答案解析：C
类大小的计算方式：将类中非静态成员变量相加，注意内存对齐
如果一个类中包含有虚函数，编译时会在对象前增加4个字节保存虚表的地址
如果按照4字节对齐：            如果按照1字节对齐
对象模型为：                  对象模型为：
-----------                 ------------
| 虚表指针 | 4              | 虚表指针 |
|  char  | 1                |  char  |
| 补3个字节 | 3              |   int   |
|  int   | 4                -------------
------------
总共12个字节                  总共9个字节
故选择C
```

3、

```
答案解析：A
A：错误，虚函数必须要是类的成员函数
B：正确，虚函数作用就是为了实现动态多态的
C：正确，函数重载只要在同一个作用域，函数名相同，参数列表不同即可，而虚函数必须是成员函数
D：正确，函数重载在编译时需要根据传递实参类型来确定调用那个函数；虚函数需要通过对象拿到对应的虚表，调用
  对应的虚函数
```

4、

```
答案解析：B
频繁调用并且短小的函数，适合编译时展开，少了函数调用的开销，可以提高程序的运行效率，即适合作为内联函数
```

5、

```
答案解析：C
A：正确，子类指针指向的肯定是子类对象，从对象前4个字节中拿到的肯定是子类的虚表地址，访问的是子类虚函数
B：正确，普通函数的调用，主要看调用该函数的类型，那个类的对象，访问那个类的非虚函数
C：错误，要看父类指针指向的是父类还是子类的对象，指向那个类的对象，就访问那个类的虚函数
D：正确，参考B
```

二、编程题

1、**【解题思路】**

此题主要是对unordered_set的特性以及使用考察。

通过观察题目所给用例，发现单词中字母大小写均存在，所以构造每行字母集合时必须包含大小写，具体如下：

1. 将每行字符的大小写序列提前构造好，然后依次放到3个unordered_set中
2. 逐个获取每个单词，对每个单词分别进行以下操作：
    - 用3个bool类型的变量来标记，单词中的字母都在那些行出现，初始值全设置为false
    - 逐个获取单词中的每个字母，然后检测该单词是在哪行出现了，将对应的标记修改为true
    - 单词检测完后，检测下三个bool标记，如果有超过1个比较被设置为true，说明该单词需要借助两行字母才可以打印出来；否则保存到结果中

【代码实现】

```
class Solution {
public:
    vector<string> findWords(vector<string>& words) {
        vector<string> vret;
        string lineString[3] = {"qwertyuiopQWERTYUIOP", "asdfghjklASDFGHJKL", "zxcvbnmZXCVBNM"};
        // 将三行内容放到unordered_set中
        unordered_set<char> sets[3];
        for(int i = 0; i < 3; ++i)
        {
            for(auto ch : lineString[i])
                sets[i].insert(ch);
        }

        // 逐个取到每个单词
        for(auto word : words)
        {
            bool flags[3] = {false, false, false};

            // 检测该单词中的每个字符是否在同一行出现了
            for(auto ch : word)
            {
                for(int i = 0; i < 3; ++i)
                {
                    if(sets[i].find(ch) != sets[i].end())
                    {
                        flags[i] = true;
                    }
                }
            }

            int count = 0;
            for(bool flag : flags)
            {
                if(flag)
                    count++;
            }

            if(count == 1)
            {
                vret.push_back(word);
            }
        }

        return vret;
    }
};
```


2、**【解题思路】**

此题是对unordered_map的特性进行考察，题目比较简单，思路如下：

1. 使用unordered_map统计不同种类石头的块数
2. 获取一块块宝石，在石头统计的结果中统计宝石的次数

【代码实现】

```
class Solution {
public:
    int numJewelsInStones(string jewels, string stones) {
        // 使用unordered_map统计不同种类的石头总共有多少个
        unordered_map<char, int> m;
        for(auto ch : stones)
            m[ch]++;

        // 统计宝石的个数
        int count = 0;
        for(auto ch : jewels)
            count += m[ch];

        return count;
    }
};
```

day10

一、选择题

1、

```
答案解析：B
A选项：del命令是DOS下的命令， 并不是linux下的命令
B选项：正确。-r命令行参数为递归删除，-f命令行参数为强制删除
C选项：rm命令没有-a命令行参数。
D选项：/tmp/*是指的删除/tmp/文件夹下的所有， 但是并不会删除/tmp这个文件夹
所以， 选择B
```

2、

```
答案解析：C
A选项：find命令是linux下的文本查找工具。
B选项：gzip命令是linux下的压缩工具。
C选项：grep命令是一种强大的文本搜索工具。
D选项：sort命令用于将文本文件内容加以排序。
综上：选择C选项
```

3、

答案解析：A
A选项：useradd是增加用户的命令。
B选项：usermod是修改用户账号信息的命令。
C选项：groupadd是添加组账号的命令。
D选项：userdel是删除用户的命令。

4、

答案解析：A
tar压缩分为两种压缩， 一种是gzip压缩， 一种是bz2压缩。
压缩的命令范式为：tar -zcvf 目标文件 源文件
z表示使用gzip压缩
j表示使用bzip2压缩
c表示压缩
x表示解压缩
v:表示展示压缩和解压缩的过程
f后紧跟压缩产生的文件
综上：只有A选项符合命令范式且符合gzip压缩

5、

答案解析：C
1.-rw-r--r-x: 第一个“-”代表的是文件类型， 其后“rw-”代表针对文件所有者的权限， 其后“r--”代表针对文件所数组用户的权限，“r-x”代表针对其他用户的权限
2.三位的权限可以是使用8进制的数字来表示， 可以认为是三个比特位， 当拥有了某一个权限， 则该位位1。例如：rwx : 则二进制表示位111， 转化为8进制则为7。
3.本道题， 针对文件所有者的权限“rw-“， 则8进制数字为6；针对文件所数组用户的权限“r--”， 则8进制数字为4；针对其他用户的权限“r-x”， 则8进制数字为5;
4.组合起来则为645， 选择C选项

二、编程题

1、**【解题思路】：**

本题主要考察的是位运算， 利用二进制，记录一下每一位上1出现的次数。然后模k，如果余1的话就说明只出现了一次的数这一位是就是1，(求解过程解释：这是分别求解每一个数的对应位的二进制是否为1，因为某个数出现k次，如果这个位是1 ，至少就是k的整数倍，如果这个位1的个数不是k的整数倍，说明就不是出现k次的数，就把ans的这个位置1，直到把32个位检查完毕)，当然，也可以使用暴力法进行统计求解，建议可以尝试实现然后对比代码的时间复杂度，一般暴力求解很容易想到，但往往不是面试官想看到的答案。

【代码实现】：

```
class Solution {
public:
    int foundOnceNumber(vector<int> &arr, int k)
    {
        int cnt[32] = {0};
        for(int i=0; i<arr.size(); ++i)
        {

            //统计每一个数的二进制1
```

```
        for(int j=0; j<32; ++j)
        {
            int mask = 1 << j;
            if(arr[i] & mask)
                cnt[j]++;
        }
    }

    int ans = 0;
    for(int j=0; j<32; ++j)
    {
        if(cnt[j] % k)
            ans |= (1<<j);
    }
    return ans;
  }
};
```

2、**【解题思路】：**

本题主要考察的是位运算,此题需要交换32个比特位，且不考虑有符号问题，解题思路为每次取出n的最低位，放到变量的对应高位，在将n右移一位，直到循环32次分别取出32个比特位为止。

【代码实现】：

```
class Solution {
public:
  uint32_t reverseBits(uint32_t n)
  {
    uint32_t ans = 0;
    for(int i=0; i<32; ++i)
    {
      //每次取n的最低位，在左移相应的位数进行逆置存储
      ans |= ((n&1) << (32-i-1));
      n >>= 1;
    }
    return ans;
  }
};
```


day11

一、选择题

1、

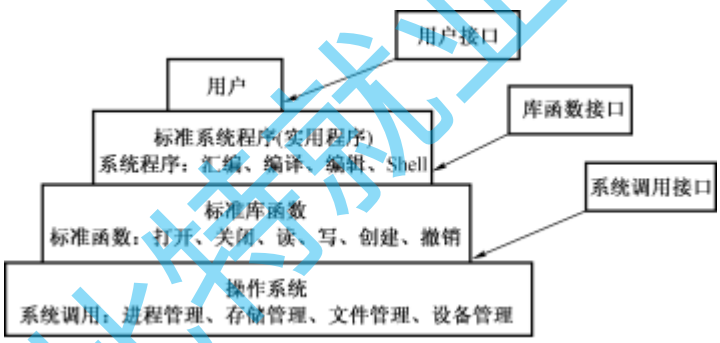
答案解析：B
1.vim进入底行模式之后， 替换字符串的命令范式为 %s/[待替换的字符串]/[替换成为的字符串]/g
　　1.1 其中%表示全文替换
　　1.2 s是替换字符串命令的起手势， 必须携带
　　1.3 g表示替换光标所在行的所有待替换的字符串
2.所以在上述的4个选项当中， 符合全文替换的选项为B

2、

答案解析：A
A选项： 为info break的缩写， 为查看断点信息的命令
B选项： 为next的缩写， 逐过程调试的命令， 对标win的F10
C选项： 为查看代码的命令
D选项： 为查看调用堆栈的命令
综上：选择A选项

3、

答案解析：
正确答案：A
参考下面的图片：



4、

答案解析：D
A选项：父进程先于子进程退出， 则其子进程为孤儿进程
B选项：子进程先于父进程退出， 则子进程就会成为僵尸进程。 父进程可以使用wait或者waitpid来回收子进程的状态信息。
C选项：孤儿进程会被1号进程所领养， 由1号进程回收孤儿进程（原某个进程的子进程）的状态信息
D选项：僵尸进程因为资源不会完全释放，因此有可能会造成资源泄漏，但是孤儿进程不会。
综上：选择D选项， 孤儿进程本质上不会造成危害

5、

答案解析：D
1.pwd的路径在linux操作系统当中的路径是正确的 "/usr/bin/pwd"
2.程序替换成功后，当前进程运行的程序将成为pwd程序，因此当前程序的最后一句printf将不会被执行，因为当前进程在运行完pwd程序后将退出
综上：上述代码应该execl替换函数之后，就会替换成为pwd程序， 执行完pwd之后程序就结束了。选择D选项

二、编程题

1、**【解题思路】：**

此题表面一看，是对字符串进行求解，第一想到的方法即为暴力比较，在t中每次取一个字符在s中进行判断，此法时间复杂度为O(n^2), 一般第一眼能够想到的方法往往都不是面试官最想看到的方法，所以另辟蹊径，发现其实这个题是一个位运算，思路为两个相同的字符异或操作结果为0，所以只需直接将两个字符串的所有字符异或一遍，最终剩下的字符即为答案。

【代码实现】：

```
class Solution {
public:
    char findTheDifference(string s, string t)
    {
        char ans = 0;
        int i = 0, j = 0;
        while(i<s.size() || j<t.size())
        {
            ans ^= (s[i++] ^ t[j++]);
        }
        return ans;
    }
};
```

2、**【解题思路】：**

此题主要考察对字符串的操作，其中需要注意的一点就是好字符串是连续的字符串。 遍历子字符串的起始下标 i，并检验 i开头的长度为3的子串是否为好字符串，即是否不含有重复字符。与此同时，我们维护长度为3好子串的个数即可。

【代码实现】：

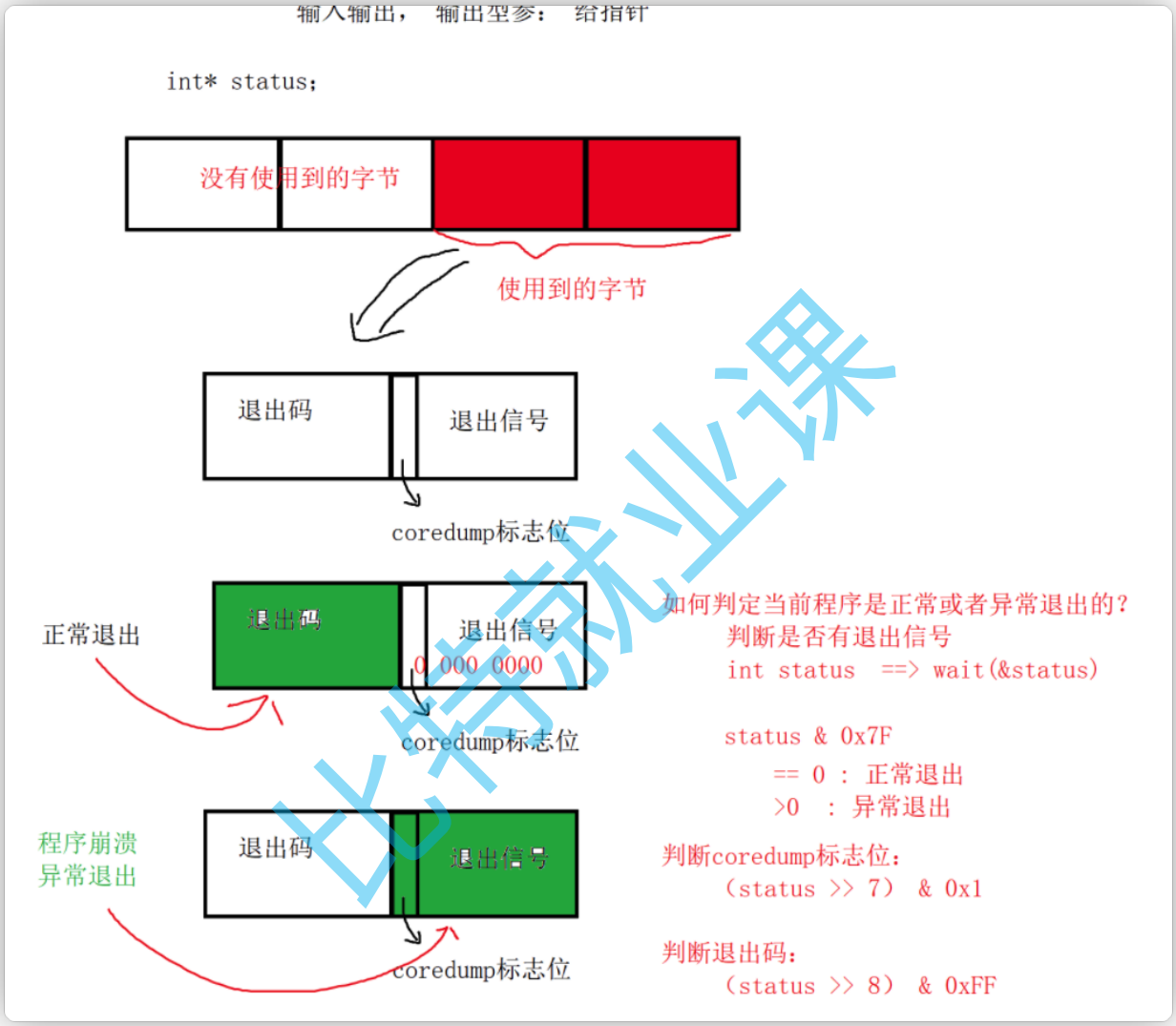
```
class Solution {
public:
    int countGoodSubstrings(string s)
    {
        int cnt = 0;
        int len = s.size();
        for(int i=0; i<len-2; ++i)
        {
            //判断三个字母是否重复
            if((s[i]!=s[i+1]) && (s[i]!=s[i+2]) && (s[i+1]!=s[i+2]))
                cnt++;
        }
        return cnt;
    }
};
```

day12

一、选择题

1、

答案解析：D
D选项：当options被设置为WNOHANG则函数非阻塞，且当没有子进程退出时，waitpid返回0
A,B,C选项都是正确的。
综上：选择D选项
扩展：如何通过status获取退出码， coredump标志位， 退出信号， 参考下图：



2、

答案解析：B
Linux世界里的两句流行语：
1.一切皆文件：Linux中所有内容都是以文件的形式保存和管理；
2.有问题找男人：可以使用man来查看命令帮助手册，如man pwd
3.软链接文件相当于是文件的快捷方式
综上：选择B选项

3、

答案解析：A
1.在Linux系统中,一切设备都看作文件。而每打开一个文件,就有一个代表该打开文件的文件描述符。程序启动时默认打开三个I/O设备文件:标准输入文件stdin,标准输出文件stdout,标准错误输出文件stderr,分别得到文件描述符0,1,2。
文件描述符0：标准输入文件stdin
文件描述符1：标准输出文件stdout
文件描述符2：标准错误输出文件stderr
2.管道文件描述符需要程序员额外创建， 不属于程序创建之初创建的文件描述符
综上：选择A选项

4、

答案解析：B
1.软链接可以跨文件系统，硬链接不可以
2.硬链接不管有多少个，都指向的是同一个I节点，会把结点连接数增加，只要结点的连接数不是0，文件就一直存在不管你删除的是源文件还是链接的文件。只要有一个存在文件就存在。 当你修改源文件或者连接文件任何一个的时候，其他的文件都会做同步的修改。软链接不直接使用i节点号作为文件指针, 而是使用文件路径名作为指针。所以删除链接文件对源文件无影响，但是删除源文件，链接文件就会找不到要指向的文件。软链接有自己的inode, 并在磁盘上有一小片空间存放路径名。
3.软连接可以对一个不存在的文件名进行连接 。
4.软连接可以对目录进行连接。
综上：选择B选项

5、

答案解析：A，B，C
1.管道(pipe)：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
2.有名管道 (named pipe)： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
3.信号量(semophore)： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
4.消息队列(message queue)： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
5.信号 (signal)： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
6.共享内存(shared memory)： 共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
7.套接字(socket)： 套接字也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。
综上：选择A, B,C

二、编程题

1、**【解题思路】：**

此题主要考察对字符串的操作,要查找不常用单词，即找出两个字符串中只出现一次的单词即可，方法是将两个字符串的单词全部提取、排序，然后删掉重复单词，最后剩下的单词就是两个字符串中只出现一次的单词，即不常见单词。

【代码实现】：

```
class Solution {
public:

    //分割字符串，将单词保存在vs
```

```
void split_string(vector<string> &vs, string &s)
{
    string::iterator it = s.begin();
    size_t start = 0;
    size_t pos = s.find(' ');
    while(pos != string::npos)
    {
        string tmp(it+start, it+pos);
        vs.push_back(tmp);
        start = pos + 1;
        pos = s.find(' ', pos+1);
    }
    string tmp(it+start, it+s.size());
    vs.push_back(tmp);
}
//删除重复单词
void erase_repeat_word(vector<string> &vs)
{
    vector<string>::iterator slow = vs.begin();
    vector<string>::iterator fast = slow + 1;
    while(slow != vs.end())
    {
        if (*fast == *slow)
        {
            while (fast!=vs.end() && *fast == *slow)
                fast++;
            slow = vs.erase(slow, fast);
            if(slow != vs.end())
                fast = slow + 1;
        }
        else
        {
            slow = fast;
            fast = slow + 1;
        }
    }
}
vector<string> uncommonFromSentences(string s1, string s2)
{
    vector<string> vs;

    //提取s1单词
    split_string(vs, s1);

    //提取s2单词
    split_string(vs, s2);

    //对单词进行排序
    sort(vs.begin(), vs.end());

    //删除重复单词
    erase_repeat_word(vs);
```

```
        return vs;
    }
};
```

2、**【解题思路】：**

此题主要考察字符串的操作，根据题意，需要将连续相同字符压缩为字符+出现次数的格式, 考虑借助双指针 i, j 解题，算法流程为：

1、令i指向字符串的首个字符, j向前遍历，直到访问到不同字符时停止，此时j指向下一个不同字符的起始位置，那么j-i便是首个字符的连续出现次数，一旦知道了字符出现的次数，就可以完成字符的压缩操作（先将字符放入ans中，然后在将该字符出现的次数链接到字符后面，即完成一个字符的压缩）。2、接下来，从下个字符开始，重复以上操作，直到遍历完成即可。3、根据题目要求，最终返回原字符串和压缩字符串中长度较短的字符串即可。

【代码实现】：

```
class Solution {
public:
    string compressString(string S)
    {
        string ans;
        int i=0, j=0;
        while(i < S.size())
        {
            while(j<S.size() && S[j]==S[i])
                j++;

            ans += S[i];
            ans += to_string(j-i);
            i = j;
        }

        return ans.size() < S.size() ? ans : S;
    }
};
```


day13

一、选择题

1、

```
答案解析：A, D
A选项：描述正确
B选项：ctrl+c产生SIGINT信号
C选项：int raise(int sig); 所以raise接口可以向当前调用进程发送任意信号；
D选项：函数原型为unsigned int alarm(unsigned int seconds); 可以通过alarm接口可以在n秒钟之后产生一个SIGALRM信号
综上：选择A, D
```

2、

答案解析：A， C
A:SIGKILL, 9号信号为强杀信号， 操作系统定义该信号不可以自定义处理方式， 不能被阻塞。
B:SIGINT, 2号信号为终止进程的信号。 该信号可以自定义处理方式和阻塞
C:SIGSTOP, 19号信号默认执行的动作为停止一个进程， 操作系统定义该信号不可以自定义处理方式， 不能被阻塞。
D:SIGQUIT, 3号信号为终止进程并且产生coredump文件的信号。 该信号可以自定义处理方式和阻塞
综上： 选择A, C

3、

答案解析：D
A选项：sigaction接口也可以自定义信号处理方式
B选项：信号会打断进程当前的阻塞状态去处理信号
C选项：SIG_DFL为信号默认处理方式, 并非是忽略处理的方式
D选项：SIG_IGN为信号的忽略处理方式
综上：选择D

4、

答案解析：A, B, D
A选项： 进程的独立行导致了多进程程序需要借助进程间通信
B选项：线程的创建， 切换和销毁速度比进程快， 因为共用进程地址空间
C选项：大量计算的程序可以优先使用多线程
D选项：线程的崩溃会导致进程的崩溃， 整个程序会退出

5、

答案解析：D
A选项：用于销毁互斥锁
B选项：用于加锁保护临界区
C选项：用户非阻塞加锁
D选项：没有这个函数
综上： 选择D选项

二、编程题

1、**【解题思路】：**

此题主要考试字符串的统计，使用一个值对结构pair，统计字符串和出现的次数，然后遍历整个字符串，找出第K个只出现一次的字符串即可，如果不存在，就返回一个空串。

【代码实现】：

```
class Solution {
public:
    string kthDistinct(vector<string> &arr, int k)
    {
        vector<pair<string, int>> map;
```

```
        int j;
        for (int i = 0; i < arr.size(); ++i)
        {
            //在map中查找字符串，如果出现则次数++
            for (j = 0; j < map.size(); ++j)
            {
                if (arr[i] == map[j].first)
                {
                    map[j].second++;
                    break;
                }
            }
            //字符串没有出现过，放入map,且次数为1
            if (map.empty() || j >= map.size())
            {
                map.push_back(pair<string, int>(arr[i], 1));
            }
        }

        //查找第k个只出现一次的字符串
        for (int i = 0; i < map.size(); ++i)
        {
            if (map[i].second == 1)
                k--;
            if (k == 0)
                return map[i].first;
        }
        return "";
    }
};
```


2、**【代码实现】：**

```
class Solution {
public:
    vector<string> commonChars(vector<string> &words)
    {
        vector<string> result;
        if (words.size() == 0)
            return result;

        int hash[26] = {0}; // 用来统计所有字符串里字符出现的最小频率
        for (int i = 0; i < words[0].size(); i++)
        {
            // 用第一个字符串给hash初始化
            hash[words[0][i] - 'a']++;
        }

        int hashOtherStr[26] = {0}; // 统计除第一个字符串外字符的出现频率
        for (int i = 1; i < words.size(); i++)
```

```
        {
            memset(hashOtherStr, 0, 26 * sizeof(int));
            for (int j = 0; j < words[i].size(); j++)
            {
                hashOtherStr[words[i][j] - 'a']++;
            }
            // 更新hash，保证hash里统计26个字符在所有字符串里出现的最小次数
            for (int k = 0; k < 26; k++)
            {
                hash[k] = min(hash[k], hashOtherStr[k]);
            }
        }

        // 将hash统计的字符次数，转成输出形式
        for (int i = 0; i < 26; i++)
        {
            while (hash[i] != 0)
            {
                string s(1, i + 'a'); // char -> string
                result.push_back(s);
                hash[i]--;
            }
        }

        return result;
    }
};
```

day14

一、选择题

1、

答案解析：B,C
A选项：条件变量的等待接口：
int pthread_cond_wait(pthread_cond_t *restrict cond,pthread_mutex_t *restrict mutex);
条件变量进行同步的条件判断需要外部的全局条件判断实现，因此需要搭配互斥锁使用。
B选项：同A选项， 在等待接口当中需要传入互斥锁变量
C.条件变量在等待被唤醒时需要重新对条件进行判断， 防止被唤醒后，待要处理的资源被其他线程消耗了。

```
void* MakeStart(void* arg)
{
    (void)arg;
    while(1)
    {
        pthread_mutex_lock(&g_lock);
        while(g_bowl > 0) //该行代码就是对于资源的数量进行重复判断
        {
            pthread_cond_wait(&g_make_cond, &g_lock);
        }
        g_bowl++;

        printf("i am %p, i make %d\n", pthread_self(), g_bowl);
```

```
        pthread_mutex_unlock(&g_lock);

        //通知吃面的人来进行吃面
        pthread_cond_signal(&g_cond);
    }
    return NULL;
}
```

D 条件变量的使用中不同的角色需要等待在不同的条件变量等待对垒中，防止角色误唤醒

综上： 选择B, C

2、

答案解析：A,B,C,D

死锁的必要条件为：互斥条件， 请求和保持， 不可剥夺， 循环等待

综上： 选择A,B,C,D

3、

答案解析：

正确答案：B

按照覆盖地理范围和规模不同，可以将计算机网络分为局域网、城域网和广域网。局域网是一种在有限区域内使用的网络，它所覆盖的地区范围较小，一般在几千米之内，适用于办公室网络、企业与学校的主干局网络。故正确答案为B选项。

4、

答案解析：A

IP地址的分类

（1）网络号:用于识别主机所在的网络;

（2）主机号:用于识别该网络中的主机;

所以IP地址中的主机号，用于识别该子网网络中的主机

5、

答案解析：C

IP地址是IP协议提供的一种统一的地址格式，它为互联网上的每一个网络和每一台主机分配一个逻辑地址，以此来屏蔽物理地址的差异。Internet上的每台主机都有一个唯一的IP地址。IP协议就是使用这个地址在主机之间传递信息。

二、编程题

1、**【解题思路】：**

此题主要考察数组vector的操作，初步一想，统计每一行1的个数，然后在进行排序，就可以得到最弱的前K行。只是题目的特殊性，“军人总是排在一行中的靠前位置，1在0的前面”， 因此，我们可以通过二分查找的方法，找出一行中最后的那个1的位置。如果其位置为`pos`，那么这一行1的个数就为`pos+1`。特别地，注意特殊情况，如果这一行没有1，那么令`pos=-1`。当我们得到每一行的战斗力后，我们可以将它们全部放入一个小根堆中，并不断地取出堆顶的元素k次，这样我们就得到了最弱的k行的索引。

【代码实现】：

```
class Solution {
```

```
public:
    vector<int> kWeakestRows(vector<vector<int>> &mat, int k)
    {
        int m = mat.size(), n = mat[0].size();

        //pair: first为战斗力， second为行索引
        vector<pair<int, int>> power;

        for (int i = 0; i < m; ++i)
        {
            int left = 0, right = n - 1, pos = -1;
            while (left <= right)
            {
                int mid = (left + right) / 2;
                if (mat[i][mid] == 0)
                {
                    right = mid - 1;
                }
                else
                {
                    pos = mid;
                    left = mid + 1;
                }
            }
            power.push_back(pair<int,int>(pos + 1, i));
        }

        //创建小堆
        priority_queue q(greater<pair<int, int>>(), move(power));
        vector<int> ans;
        for (int i = 0; i < k; ++i)
        {
            ans.push_back(q.top().second);
            q.pop();
        }
        return ans;
    }
};
```


2、**【解题思路】：**

此题主要考察字符串的贪心求解，根据题意，对于一个平衡字符串s，若s能从中间某处分割成左右两个子串，若其中一个是平衡字符串，则另一个的L和R字符的数量必然是相同的，所以也一定是平衡字符串。为了最大化分割数量，我们可以不断循环，每次从s中分割出一个最短的平衡前缀，由于剩余部分也是平衡字符串，我们可以将其当作s继续分割，直至s为空时，结束循环。代码实现中，可以在遍历s时用一个变量d维护L和R字符的数量之差，当d=0 时就说明找到了一个平衡字符串，将答案加一

【代码实现】：

```
class Solution {
public:
```

```
    int balancedStringSplit(string s)
    {
        int ans = 0, d = 0;
        for (const auto &ch : s)
        {
            ch == 'L' ? ++d : --d;
            if (d == 0)
            {
                ++ans;
            }
        }
        return ans;
    }
};
```

day15

一、选择题

1、

```
答案解析：D
只列出典型的：
应用层：HTTP、FTP、DNS、SNMP
传输层：TCP、UDP
网络层：IP、ICMP
数据链路层：以太网协议
```

2、

```
答案解析：B, D
函数名称解析：n对应是network；h对应是host; s对应是short；l对应是long
A选项：htons为short类型主机字节序转换成为网络字节序
B选项：ntohs为short类型网络字节序转换成为主机字节序
C选项：htonl为long类型主机字节序转换成为网络字节序
D选项：ntohl为long类型网络字节序转换成为主机字节序
综上：选择B， D
```

3、

```
答案解析：A, D
A选项：TCP协议，采用流方式，SOCK_STREAM, 可靠
B选项为网络层协议
C选项为应用层协议， HTTP底层基于TCP， 使用流套接字
D选项：UDP协议，采用数据报方式，SOCK_DGRAM, 不可靠
综上：选择A, D
```

4、

答案解析：C
backlog这个参数指示的是存放已经建立连接（established）并等待被accept的sockets的队列的长度。
综上：选择

5、

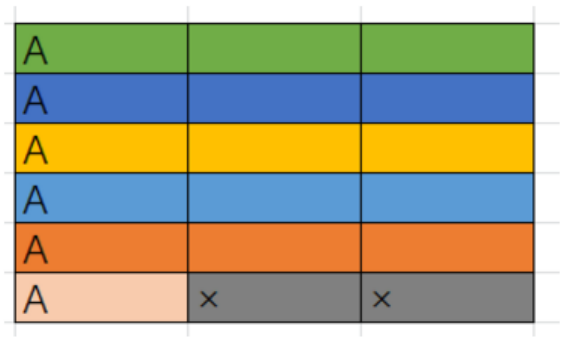
答案解析：C
A选项：ping命令用来检测两部主机之间的传输信道是否畅通，或远程主机是否正常。
B选项：route命令用来显示目前本机路由表的内容，并且还可以针对路由表中的记录来进行相应的添加、删除或修改等操作。
C选项：traceroute命令侦测主机到目的主机之前所经过的路由。
D选项：ifconfig命令用来来检测和设置本机的网络接口。
综上： 选择C选项

二、编程题

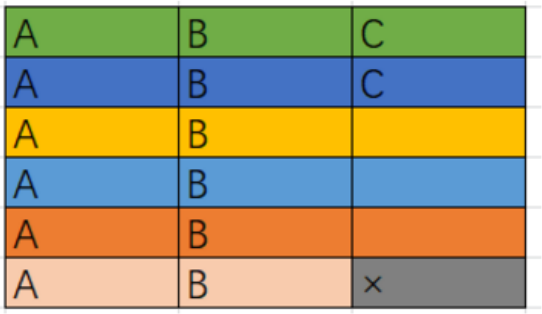
1、**【解题思路】：**

参考「桶思想」，详细说明各种情况
建立大小为 `n+1` 的桶子，个数为任务数量最多的那个任务，比如下图，等待时间 `n=2`，A 任务个数 6 个，我们建立 6 桶子，每个容量为 3：
我们可以把一个桶子看作一轮任务

1. 先从最简单的情况看起，现在就算没有其他任务，我们完成任务 A 所需的时间应该是 `(6-1)*3+1=16`，因为最后一个桶子，不存在等待时间。



2. 接下来我们添加些其他任务



可以看到 C 其实并没有对总体时间产生影响，因为它被安排在了其他任务的冷却期间；而 B 和 A 数量相同，这会导致最后一个桶子中，我们需要多执行一次 B 任务，现在我们需要的时间是 `(6-1)*3+2=17`

前面两种情况，总结起来：总排队时间 = (桶个数 - 1) * (n + 1) + 最后一桶的任务数

3. 当冷却时间短，任务种类很多时

比如上图，我们刚好排满了任务，此时所需时间还是 17，如果现在我还要执行两次任务 F，该怎么安排呢?

A	B	C	F
A	B	C	F
A	B	D	
A	B	D	
A	B	D	
A	B	×	

此时我们可以临时扩充某些桶子的大小，插进任务 F，对比一下插入前后的任务执行情况：

插入前：ABC | ABC | ABD | ABD | ABD | AB

插入后：ABCF | ABCF | ABD | ABD | ABD | AB

我们在第一个、第二个桶子里插入了任务F，不难发现无论再继续插入多少任务，我们都可以类似处理，而且新插入元素肯定满足冷却要求

继续思考一下，这种情况下其实每个任务之间都不存在空余时间，冷却时间已经被完全填满了。

也就是说，我们执行任务所需的时间，就是任务的数量

这样剩下就很好处理了，我们只需要算两个数：

1. 记录最大任务数量 N，看一下任务数量并列最多的任务有多少个，即最后一个桶子的任务数 X，计算 `NUM1=(N-1)*(n+1)+x`
2. `NUM2=tasks.size()`
   输出其中较大值即可
   因为存在空闲时间时肯定是 `NUM1` 大，不存在空闲时间时肯定是 `NUM2>=NUM1`

【代码实现】：

```
class Solution {
public:
    int leastInterval(vector<char> &tasks, int n)
    {
        int len = tasks.size();

        //任务用字母表示，所以最多26个
        vector<int> vec(26);

        //统计每一个任务出现的次数
        for(char &c : tasks)
            ++vec[c-'A'];

        //要查找出现次数最多的任务，因此从大到小排序
        sort(vec.begin(),vec.end(),greater<char>());

        int cnt=1;
        while(cnt<vec.size() && vec[cnt]==vec[0])
            cnt++;

        return max(len, cnt+(n+1)*(vec[0]-1) );
    }
};
```

2、**【解题思路】：**

根据题目，可以得出以下结论

- 当a能向b发消息时要满足条件：0.5 * a + 7 < b && b <= a
- 对两个条件进行合并，可知0.5 * a + 7 < a，即当 a * 0.5 > 7时，a才有对应的b可以发消息，在枚举a时先特判该条件
- 因此，要得出a能发多少消息，只需统计有多少满足条件的b即可
- 对数组进行排序，从后面往前枚举a，可保证a前面的元素b全都满足a >= b,因此只需判断前面的元素，b是否符合0.5 * a + 7 < b
- 根据有序性，可知第一个大于0.5 * a + 7的元素到i全都为符合条件的b

【代码实现】：

```
class Solution {
public:
    int numFriendRequests(vector<int> &ages)
    {
        int ans = 0, n = ages.size();
        sort(ages.begin(), ages.end());

        //从后面往前枚举a，保证a >= b
        for(int i=n-1; i>=0; --i)
        {
            int a = ages[i];

            //a能发消息的前提是要满足a * 0.5 > 7
            if(a * 0.5 <= 7)
                continue;

            //在区间[l, r]找第一个大于target的下标
            int target = a/2+7, left=0, right=i-1;
            while(right>=0 && ages[i]==ages[right])
            {
                //排除掉与age[i]相等的元素，一定可以互相发消息，因此请求数加2
                right--;
                ans += 2;
            }

            //记录最开始的末尾
            int end = right;
            while(left < right)
            {
                int mid = (left + right) / 2;
                if(ages[mid] <= target)
                    left = mid + 1;
                else
                    right = mid;
            }


            //i一定可以向区间[第一个大于target的下标, end]发消息
```

```
            if(end>=0 && ages[left]>target)
                ans = ans + end - left + 1;
        }
        return ans;
    }
};
```

day16

一、选择题

1、

```
答案解析：C
选项A: HTTP状态码500代表服务器内部发生错误
选项B: tcp协议在建立连接的时候，需要经历三次握手。 在断开连接的时候， 需要经历四次挥手。
选项C: DNS在传输层借助的是UDP协议， 所以该选项错误
选项D: 根据C选项， 该选项正确
综上: 由于题目是选择错误的， 因此选择C选项
```

2、

```
答案解析: B
A选项:UDP数据包的大小最大为65535
B/C选项：UDP是无连接的面向数据报的协议，不保证安全可靠，所以不会进行错误重传
D选项：UDP是面向数据报的协议， 不会有流控制
```

3、

```
答案解析：D
服务端的accept调用，是TCP的三次握手之后，已经建立了tcp链接，accept从队列里面获取新的连接
综上：选择D
```

4、

```
答案解析:C
由连接发起方 发送SYN段， 接收方介绍到发送发发送的SYN数据段之后， 回复SYN-ACK。所以，SYN-ACK是由被动连接方发送给连接方的。
综上：选择C
```

5、

答案解析：C
滑动窗口协议（Sliding Window Protocol），属于TCP协议的一种机制，用于网络数据传输时的流量控制，以避免拥塞的发生。该协议允许发送方在停止并等待确认前发送多个数据分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输，提高网络吞吐量。
A选项：差错控制是保证双发发送到的数据是在网络传输当中没有被篡改
C选项：流量控制--滑动窗口
D选项：拥塞控制--拥塞窗口
综上：选择C选项

二、编程题

1、**【解题思路】：**

此题主要考察类的设计：

1、记录每次设置值的时间与取值，

2、获取快照的时候，只需要二分查找离快照时间最近的取值即可，可以采用lower_bound函数

【代码实现】：

```
class SnapshotArray {
public:
    //构造数据类型
    struct Num
    {
        int val;
        int time;
        Num(int t, int v) :val(v), time(t)
        {}
        bool operator < (const Num& other)
        {
            return time < other.time;
        }
    };

    SnapshotArray(int length)
    {
        time = 0;
        size = length;
        snap_size = 0;
        nums.resize(size);
        for (int i = 0; i < size; ++i)
        {
            nums[i].push_back(Num(time, 0));
        }
    }

    void set(int index, int val)
    {
        nums[index].push_back(Num(++time, val));
    }
```

```
    int snap()
    {
        snaps.push_back(time);
        return snap_size++;
    }

    int get(int index, int snap_id)
    {
        Num n = Num(snaps[snap_id] + 1, 0);
        //查找第一个不小于n的值
        return (--lower_bound(nums[index].begin(), nums[index].end(), n))->val;
    }

    //添加成员
    private:
        int time;
        int size;
        int snap_size;
        vector<vector<Num> > nums;  //存放每一个时间下的某个index的取值
        vector<int> snaps;          // 存放快照时间
};

/**
 * Your SnapshotArray object will be instantiated and called as such:
 * SnapshotArray* obj = new SnapshotArray(length);
 * obj->set(index,val);
 * int param_2 = obj->snap();
 * int param_3 = obj->get(index,snap_id);
 */
```


2、**【解题思路】：**

要设计缓存机制，首先要对缓存有一个理解，即缓存就是拥有一段固定的空间，用于数据的存储，如果数据量超出了空间的大小，此时需要丢弃部分数据，其原则为**最近最少使用**，比如，空间有3个，将1和2按顺序存入，此时要将3存入空间，则空间不够，那么1是最近最少使用的数据，则将1丢弃之后，在将3存入。当简单理解缓存机制后，一起看看设计思路：

1、首先设计一个简易双链表，实现其对应的删除和插入方法，双链表存储一个key-value节点，且按最近使用从左往右排序，最先被使用的节点为链表的第一个节点，因此链表最后一个节点就是最久未使用的节点。

2、借助unordered_map充当哈希表hash，主要存储key所对应的链表节点地址，用于key-value的get和put。

3、初始化缓存结构：（即`LRUCache(n)`方法），初始化L(链表的头结点)和R(链表的尾结点)，并链接形成双链表

4、根据关键值获取数据：（即`get(key)`方法）,首先用hash判断key是否存在，若不存在返回-1，若存在，则返回对应的value,并将key对应的节点删除后插入到链表的头部,表示最近使用。

5、将数据存入缓存：（即put(key, value)方法），首先用hash判断key是否存在，若存在，则修改对应的value,同时将key对应的节点删除后插入到链表的头部，表示最近使用。若不存在，如果缓存已满，则删除链表最右边节点，因为最久未使用并更新hash表，然后将新节点插入链表头部，如果缓存有空间，则直接在链表头部插入即可。

【代码实现】：

```
class LRUCache {
public:
    LRUCache(int capacity)
    {
        n = capacity;
        L = new Node(-1,-1),R = new Node(-1,-1);
        L->next = R;
        R->prev = L;
    }

    int get(int key)
    {
        if(hash.count(key) == 0)
            return -1; //不存在关键字 key
        Node *p = hash[key];
        remove(p);
        insert(p);//将当前节点放在双链表的第一位
        return p->value;
    }

    void put(int key, int value)
    {
        if(hash.count(key)) //如果key存在，则修改对应的value
        {
            Node *p = hash[key];
            p->value = value;
            remove(p);
            insert(p);
        }
        else
        {
            if(hash.size() == n) //如果缓存已满，则删除双链表最右侧的节点
            {
                Node  *p = R->prev;
                remove(p);
                hash.erase(p->key); //更新哈希表
                delete p; //释放内存
            }
            //否则，插入(key, value)
            Node *p = new Node(key,value);
            hash[key] = p;
            insert(p);
        }
    }
    private:
        //定义双链表
        struct Node
        {
            int key,value;
            Node *prev ,*next;
            Node(int _key,int _value): key(_key),value(_value),prev(NULL),next(NULL){}
        }*L,*R;//双链表的最左和最右节点，不存储值。
```

```
    void remove(Node* p)
    {
        p->next->prev = p->prev;
        p->prev->next = p->next;
    }
    void insert(Node *p)
    {
        p->next = L->next;
        p->prev = L;
        L->next->prev = p;
        L->next = p;
    }

    int n;  //缓存大小
    unordered_map<int,Node*>hash;
};
/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```