高级程序设计

第0次上机

第一关：单调栈

题目描述

作为第0次上机测试，本次OJ主要是帮助大家熟悉上机系统，了解上机流程。 在本次OJ中，你需要先实现一个指定的数据结构，并解决一个问题。

Part-1

栈是一种限定仅在表尾进行插入和删除操作的线性表。表尾这一端被称为栈顶(top)，相对地，把另一端称为栈底。在表尾插入元素的操作被称为push，删除表尾元素的操作被称为pop。栈具有后进先出的特性。

单调栈是栈的一种变体，顾名思义，就是栈内元素按照单调递增或递减的顺序排列的栈。如果有新的元素入栈，栈调整过程中会将所有破坏单调性的栈顶元素出栈，并且出栈的元素不会再次入栈。你可以结合后文的调用示例进一步理解单调栈。

这一部分要求大家实现一个栈内元素类型为`int`的**单调递增栈**类，该类（MonotononicStack）定义于MonotonicStack.h，你需要实现该类的各个接口。

Part-2

这一部分要求大家需要实现`function`函数（它是类MonotonicStack的一个成员函数），它接收两个参数：`int`数组`arr`和它的大小`n`。 对数组中的任意元素`arr[i]`，我们定义它的价值为`v[i]=arr[i]-arr[j]`，其中，`j`是满足`j>i`且`arr[j]<arr[i]`的最小下标，如果不存在这样的`j`，那么规定`v[i]=arr[i]`。 `function`返回每个元素的价值之和。 **注：虽然本次OJ对时间复杂度不做要求，但本题期望的时间复杂度应该为O(n)。** 注：你可以在Part-2的实现中使用Part-1中的所有接口和`s`成员变量，这两部分的测试将独立进行。

调用示例

题目保证返回结果一定在`int`型范围内。 我们将按照任意合法的顺序调用Part-1中的接口，你需要保证接口严格按照要求实现。

示例1

```
MonotonicStack* ms = new MonotonicStack;
ms->push(8);
ms->push(4);
ms->push(6);
ms->push(2);
ms->push(3);
cout<< ms->top() <<endl;
ms->pop();
cout<< ms->top() <<endl;
ms->pop();
cout<< ms->size() <<endl;
delete ms;
```

输出：

```
3
2
0
```

说明： 元素8首先入栈，之后元素4入栈时，为满足单调性，弹出8，4入栈，下一个元素6大于栈顶元素4，直接入栈。2入栈时，弹出栈顶元素6，新的栈顶元素4仍然大于2，继续弹出直至栈空后，2入栈，之后3大于2，所以3入栈。 此时栈内从栈底到栈顶依次是2，3两个元素，第一个输出的top为3，之后弹出3，下一个top是2，接着弹出2，此时栈空，输出size为0。

示例2

```
MonotonicStack* ms=new MonotonicStack;
ms->push(10);
ms->push(1);
cout<<ms->top()<<endl;
ms->push(1);
cout<<ms->size()<<endl;
ms->push(6);
cout<<ms->top()<<endl;
ms->pop();
cout<<ms->top()<<endl;
cout<<ms->size()<<endl;
delete ms;
```

输出：

```
1
2
6
1
2
```

示例3

```
MonotonicStack* ms=new MonotonicStack;
int arr[]={8,4,6,2,3};
cout<<ms->function(arr,5)<<endl;
delete ms;
```

输出

```
15
```

比0号元素8下标大且值小于8的最小下标的元素为1号元素4，所以0号元素的价值为8-4=4。同理可得1号元素的价值为4-2=2，2号元素的价值为6-2=4。 而对于3号元素2来说，不存在比2小并且下标大于2的元素，所以它的价值就是本身2。同理可得4号元素的价值为本身3。 五个元素的价值依次是[4,2,4,2,3]，求和就是结果15。

MonotonicStack/MonotonicStack.h

```
#include <iostream>
\#include <stack>
using namespace std;
class MonotonicStack
{
private:
  stack<int> s;
public:
  int size();
  void pop();
  int top();
  int push(int element);
  int function(const int* arr,int n);
};
```

MonotonicStack/MonotonicStack.cpp

```
#include "MonotonicStack.h"

int MonotonicStack::size(){
    return s.size();
}

void MonotonicStack::pop(){
    if(s.size()>0){
        s.pop();
    }
}

int MonotonicStack::top(){
    if(s.size()>0){
        return s.top();
    }
}

int MonotonicStack::push(int element){
```

```
    while(s.size()!=0) {
    if (element >= s.top()) {
        s.push(element);
        return 0;
    }
    else {
        s.pop();
    }
}
    if(s.size()==0){
        s.push(element);
        return 0;
    }
}

int MonotonicStack::function(const int* arr,int n){
    int value=0;
    for(int i=0;i<n;i++){
        for(int j=i+1;j<n;j++){
            if(arr[j]<arr[i]){
                value+=arr[i]-arr[j];
                break;
            }
            if(j==n-1){
                value+=arr[i];
            }
        }
    }
    value+=arr[n-1];
return value;
}
```

第1次上机

第一关：插入的字符

任务描述

小蓝鲸有两个非空字符串，str1和str2。str2是由str1中的字符重新排列以后，在随机位置插入一个字符后形成的。请你帮小蓝鲸找出str2中被随机插入的字符。

输入

两个**非空的**，**只含小写字母的**字符串。

输出：

被随机插入的**小写字母**。

样例1：

输入： `abcd` `abcde` 输出： `e`

样例2：

输入： `a` `aa` 输出： `a`

样例3：

输入： `dfdsd` `ddddsf` 输出： `d`

p6.c

```
#include<cstring>
#include<iostream>
using namespace std;
// 补充solution函数，参数为题目输入的str1和str2，返回值为所求的小写字母
char solution( char* str1,  char* str2)
{
    char answer;

    int len1=strlen(str1);
    int len2=strlen(str2);

    for(int i=0;i<len1;i++){
        for(int j=0;j<len2;j++){
            if(str1[i]==str2[j]){
                str2[j]='0';
                break;
            }
        }
    }
    for(int j=0;j<len2;j++){
        if(str2[j]!='0'){
            answer=str2[j];
            break;
        }
    }

    return answer;
}
// 我们为你提供了main函数，处理了输入和输出，你只需要关注solution函数
int main()
{
    char s1[1000];
    char s2[1000];
    cin >> s1 >> s2;
    cout << solution(s1,s2);
    return 0;
}
```

第二关：差二消除

任务描述

小蓝鲸想要检测一串数字是否为约定的有效的数字串，请你写个程序帮帮小蓝鲸吧。 本题的数字串只包含4种数字：1，2，3，4，有效的数字串需满足以下条件：

- 串中两个相邻的数字，若**后面的数字减去前面的数字等于2**，则这两个数字可以消除。消除后继续依次法处理数字串中剩下的数字，如数字串`"1 2 4 3"`，2和4相邻且满足4-2=2，因此2和4可从串

中消除，数字串变为`"1 3"`，1和3同理可继续消除，最终数字串为空串。

- 若数字串中**所有的数字**均可消除，则称该数字串为有效数字串。

输入

第一行为数字串中数字的数量n（n > 0）。 第二行为只包含四种数字：1，2，3，4的数字串，串中每个数字间用空格分隔。

输出：

若输入数字串为有效数字串则输出true，否则输出false。

样例1：

输入：`4` `1 3 2 4` 输出：`true`

样例2：

输入：`4` `1 2 4 3` 输出：`true`

样例3：

输入：`2` `1 4` 输出：`false`

p7.c

```
#include<iostream>
using namespace std;
int main(){
    int num;
    int string[1000]={0};
    int time=100;
    scanf("%d",&num);
    for(int i=0;i<num;i++){
        scanf("%d",&string[i]);
    }
    while(time!=0){
    for(int i=0;i<num-1;i++){
        if(string[i+1]-string[i]==2){
            string[i]=0;
            string[i+1]=0;
            for(int j=i+2;j<num;j++){
                string[j-2]=string[j];
            }
            num-=2;
            break;
        }

    }
    time--;
    if(string[0]==0){
        break;
    }
    }
    if(string[0]==0){
        printf("true");
    }
    if(string[0]!=0){
```

```
        printf("false");
    }
    return 0;
}
```

第2次上机

第一关：小蓝鲸与哥德巴赫猜想

任务描述

小蓝鲸在数学课上学到了哥德巴赫猜想，即： 任意一个大于2的偶数，都等于某两个素数之和 现在小蓝鲸想设计程序验证这一数学猜想。

题目要求对于每一个输入的偶数x进行哥德巴赫猜想验证。 对于每一个偶数x，寻找两个质数a、b，使得a + b = x，为了方便程序正确性验证，我们要尽可能使得b - a的值最大，即a尽可能取较小的数

输入

第一行输入一个正整数n，代表需要验证的偶数个数 接下来的n行，每行输入一个偶数x

输出：

题目要求对于每一个输入的偶数x进行哥德巴赫猜想验证并输出其对应的分解之后的结果，输出共n行，每一行均满足如下形式： `x=a+b` 其中x代表验证的偶数，a、b分别为找到的两个质数，且 `a≤b`

数据范围

`1≤n≤100` `2≤x≤2147483647`

样例1：

输入： 1 4 输出： 4=2+2

样例2：

输入： 2 6 8 输出： 6=3+3 8=3+5

样例3：

输入： 4 10 12 20 22

输出： 10=3+7 12=5+7 20=3+17 22=3+19

p0.c

```
#include <iostream>
#include <cmath>
using namespace std;
bool isPrime(int num){
    if(num<=1){
        return false;
    }
    for(int i=2;i<=sqrt(num);i++){
        if(num%i==0){
            return false;
        }
    }
```

```
    return true;
}
int main(){
    int n;
    cin >> n;
    for(int i=0;i<n;i++){
        long int x;
        cin >> x;
        long int a,b;
        for(long int i=2;i<x/2+1;i++){
            if(isPrime(i)&&isPrime(x-i)){
                a=i;
                b=x-i;
                cout << x << "=" << a << "+" << b <<endl;
                break;
            }
        }

    }
    return 0;
}
```

第二关：小蓝鲸与黑洞数

任务描述

小蓝鲸学习到了一种黑洞数，其定义为： 任何各位数字不全相同的三位正整数经以下变换后均能变为495，于是495被称为三位数的黑洞数。

变换步骤：对于任意一个各位数字不全相同的三位正整数，将组成该正整数的三个数字重新组合，分别生成一个最大数和一个最小数；用最大数减去最小数，得到一个新的三位数；再对新的三位数重复上述操作，最多重复**七次**。 请设计一个C/C++程序，验证一个三位数（通过键盘输入）是否为黑洞数，是则输出以“-”隔开的每步变换后得到的三位数；否则输出"NOT"。

另外，我们人为规定，如果变换的过程中出现了用上述方法得到的数不为三位数（如122操作后得到99），此时也认为122不为黑洞数。

输入

待验证的三位数n。

输出

如果是黑洞数，依照变换规则输出其变换过程。 如果不是黑洞数，输出NOT。

样例1

输入： 379

输出： 379-594-495

样例2

输入： 666

输出： NOT

p1.c

```
#include<iostream>
 using namespace std;
 int main(){
     int n;
     cin >> n;
     int num[8]={0};
     int j=0;
     num[j]=n;
     j++;
     int times=0;
     while(times<7&&n!=495){
     int x,y,z;
     z=n%10;
     y=(n%100-z)/10;
     x=(n-y*10-z)/100;
     if(z>=x&&z>=y&&x>=y){
         n=z*99-y*99;
     }
     else if(z>=x&&z>=y&&y>=x){
         n=z*99-x*99;
     }
     else if(x>=y&&x>=z&&y>=z){
         n=x*99-z*99;
     }
     else if(x>=y&&x>=z&&y<=z){
         n=x*99-y*99;
     }
     else if(y>=x&&y>=z&&x>=z){
         n=y*99-z*99;
     }
     else if(y>=x&&y>=z&&x<=z){
         n=y*99-x*99;
     }
     if(n<100){
         times=7;
         break;
     }
     if(n==495){
         num[j]=495;
         break;
     }
     if(n!=495){
         num[j]=n;
         j++;
         times++;
     }

     }
     if(times==7&&num[j]!=495){
         cout<<"NOT";
     }
     else{
         cout<<num[0];
         if(num[0]!=495){
         for(int i=1;i<=j;i++){
```

```
            cout<<"-"<<num[i];
         }
        }
    }

    return 0;
}
```

第三关：小蓝鲸与排队

任务描述

在一次升旗仪式中，CS的大部分小蓝鲸已经按身高从低到高排好了队伍。假设人数 n 和各位同学的身高都从键盘输入，且队伍里没有人身高相同。这时候来了一位迟到的同学，其身高为 k（也是从键盘输入，且与已有身高都不同）。请设计一个C/C++程序，在队伍里寻找迟到同学应该排入的位置。 已知n<=1000

输入

输入共三行 第一行输入n，代表有n个同学。 第二行输入k，代表新来的小蓝鲸的身高。 第三行输入n个正整数，代表n个同学的身高，为了方便起见，我们不考虑身高的具体数值的合理性（方便助教出用例，你也许会看到身高五米的小蓝鲸）。

输出

输出一个正整数m，代表新来小蓝鲸应该排在第多少位，为了方便起见，我们用0~n来代替现实生活中的1~n+1。

样例1

输入： 1 2 3

输出： 0 （新来的小蓝鲸身高2米，原本小蓝鲸身高3米，因此小蓝鲸应该排在第0位）

样例2

输入： 3 4 1 2 3

输出： 3

p2.c

```
#include<iostream>
 using namespace std;
 int main(){
    int n;
    cin >> n;
    int k;
    cin >> k;
    int stu[1001]={0};
    for(int i=0;i<n;i++){
        cin >> stu[i];
    }
    if(k<stu[0]){
        cout<<0;
    }
    else if(k>stu[n-1]){
        cout<<n;
```

```
    }
    else{
    for(int i=0;i<n;i++){
        if(k>stu[i]&&k<stu[i+1]){
            i++;
            cout<<i;
            break;
        }
    }
    }
    return 0;
}
```

第3次上机

第一关：Naive Pointer

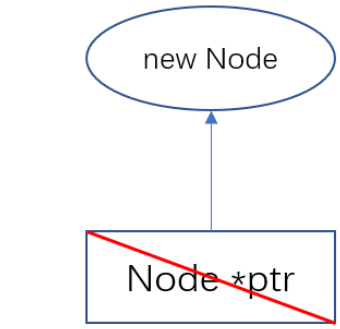
考察知识点

拷贝构造函数，析构函数

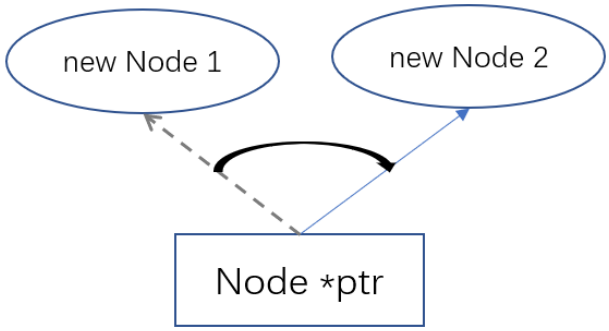
任务描述

智能指针是C++标准库提供的一种机制，用于防止内存泄漏，及时回收不再需要的内存。

当我们使用普通的指针变量时，一旦程序变得冗长复杂，难免会出现忘记释放内存的情况，导致可用的内存变少。

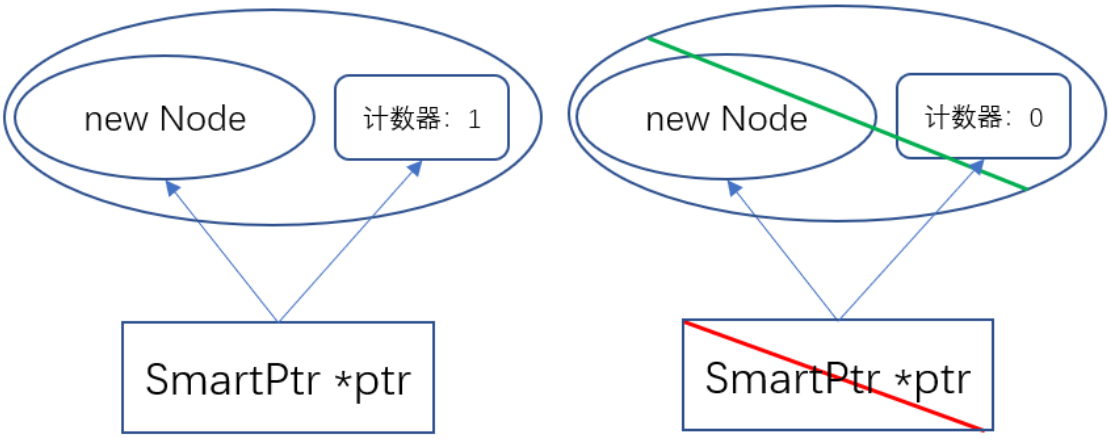


ptr变量消亡，但忘记释放Node空间

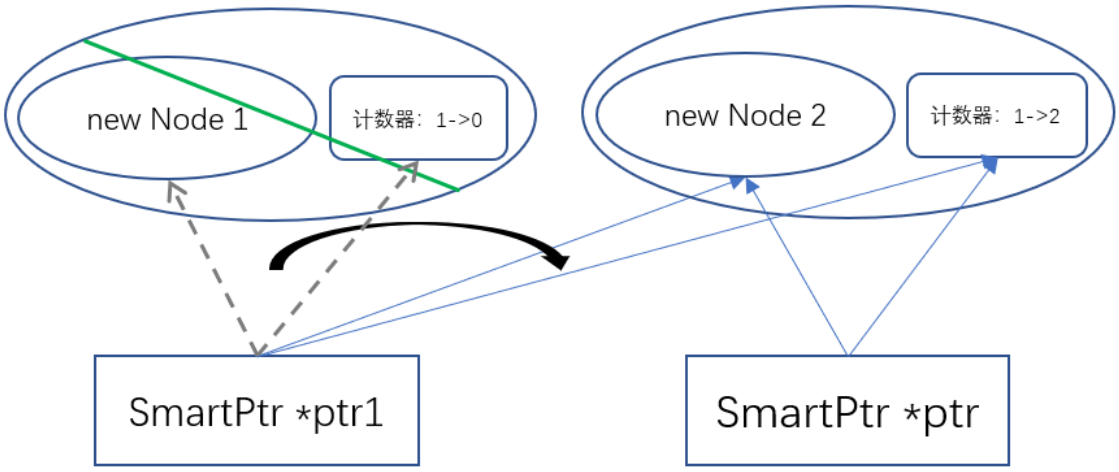


ptr变量改变指向，但没有释放Node2空间

如果使用智能指针，那么智能指针内部会维护一个指针指向目标地址，和一个相关联的计数器指针（他们在内存上的地址并无关联，只有逻辑上的联系）。计数器统计持有目标地址的指针数量，一旦计数器为0，那么就可以认为没有任何指针可以访问到该地址，该地址将不再被需要，此时，就可以释放内存。



ptr变量消亡，计数器为0，自动释放空间



ptr1变量改变指向，node1计数器减为0自动消亡，node2计数器加1

这样，我们就不需要关心内存的释放，而由智能指针选择在合适的时机释放内存。

本关要求你补全一个伪智能指针类`SmartPointer`，为了简单起见，**本关中该智能指针类只能指向动态分配内存的`Node`类型的对象，并且一个`Node`对象地址只会用于初始化一个智能指针**（`Node`类型是我们定义的占位类型，一个`Node`对象表示一块需要`SmartPointer`管理的空间）。

它在指针类内部维护两个关键成员变量：

```
Node* pointer; int* ref_cnt;
```

1. `Node`类型的指针变量`pointer`。
2. 一个指向整型变量的指针`ref_cnt`，它指向的值`*ref_cnt`表示当前拥有`pointer`所指向的`Node`对象地址的`SmartPointer`对象数。

该指针类有三个需要你完成的成员函数：

1. 拷贝构造函数，使用一个`SmartPointer`对象来初始化本对象。
2. `assign`函数，该函数接收一个`SmartPointer`对象`sptr`，并将本对象的`pointer`指针赋值为`sptr.pointer`。
3. 析构函数，销毁本`SmartPoiner`对象，也就是说，本对象不再持有`pointer`指针指向的`Node`地址。注意：为通过测试样例，需要保证析构函数可以被重复调用。

你需要在实现的过程中，维护两个成员变量的值，你可能需要思考在上述操作中，`ref_cnt`应该如何赋值和改变，使得对于任何`Node`对象，如果没有`SmartPointer`对象持有其地址，则其持有的空间被释放(即在`*ref_cnt`为0的时候释放`pointer`指向的`Node`对象)。

输出

测评文件将会根据`Node`类对象析构输出对应的id的顺序来判断程序是否正确，所以请**尽量不要修改头文件内容。**

样例1

```
SmartPointer sp1(new Node(1));//函数结束，此时node1应该被销毁
```

输出：`1`

样例2

```
SmartPointer sp1(new Node(1));SmartPointer* sp2=new SmartPointer(sp1);//函数结束，sp1被销毁，此时sp2仍持有node1的地址，node1不被销毁
```

输出：

样例3

```
SmartPointer sp1(new Node(123));sp1.~SmartPointer();sp1.assign(SmartPointer());//测试空指针赋值sp1.assign(*(new SmartPointer(new Node(456))));//Node 456仍然被堆空间中的某个指针持有，所以不会被释放
```

输出`123`

提示

1. 请注意代码的鲁棒性，如果存在能导致程序崩溃的代码，那么就可能不通过测试样例，所以请仔细思考特例条件，尤其是各种可能出现空指针的情况。
2. 请不要在代码文件中包含"shared_ptr"字段，我们将在测评时进行检测。

SmartPointer.h

```
#include <iostream>
class Node;
class SmartPointer
{
  Node *pointer;
  int *ref_cnt;

public:
  SmartPointer()
  {
    // 空指针
    pointer = nullptr;
    ref_cnt = nullptr;
  }
  SmartPointer(Node *p)
  {
    pointer = p;
```

```
    ref_cnt = new int(1);
  }

  // 需要完成的函数
  SmartPointer(const SmartPointer &sptr);
  void assign(const SmartPointer &sptr); // 指针赋值，将sptr赋值给本指针
  ~SmartPointer();                       // 析构函数，注意：为通过测试样例，需要保证析构函数可以被重复调用
};

class Node
{
  int id;

public:
  Node(int id)
  {
    this->id = id;
  }
  ~Node()
  {
    std::cout << id << ' ';
  }
};
```

SmartPointer.cpp

```
#include "SmartPointer.h"

SmartPointer::SmartPointer(const SmartPointer &sptr)
{
    pointer = sptr.pointer;
    if (pointer != nullptr) {
      ref_cnt = sptr.ref_cnt;
      ++(*ref_cnt);
    }
}

void SmartPointer::assign(const SmartPointer &sptr)
{
    if(pointer==sptr.pointer){
        return;
    }
    if(pointer&&--(*ref_cnt)==0){
        delete pointer;
        delete ref_cnt;
    }
    pointer=sptr.pointer;
    ref_cnt=sptr.ref_cnt;
    if(pointer!=nullptr){
         (*ref_cnt)++;
    }
}

SmartPointer::~SmartPointer()
{
    if(pointer&&--(*ref_cnt)==0){
```

```
            delete pointer;
            delete ref_cnt;
        }
    }
```

第二关：一元多项式

相关知识

友元函数是定义在类的外部，但是依然可以**访问类成员（包括private以及protected成员）**的函数。本题中，你需要使用友元函数来实现多项式的加法以及求导。

在本题中，我们使用了std::vector，你可以将其视作一个**动态数组**，如下是它的初始化、添加元素、遍历的方法：

```
    // 初始化的几种方式    std::vector<int> v = {1, 2, 3}; // 1. 直接使用一个初始化列表来初始化    std::vector<int> v2(10); // 2. 初始化一个长度为10的动态数组，但是其中元素没有被初始化    std::vector<int> v3(10, 0); // 3. 初始化一个长度为10的动态数组，初始化其中所有元素为0    // 向vector尾部添加元素的几种方式    v.push_back(4); // 现在v中包含1,2,3,4    v.push_back(5); // 现在v中包含1,2,3,4,5                    // 遍历vector的一种方式    size_t len = v.size(); // size()成员函数获取当前vector的大小，对于v而言，是5    for(size_t i = 0; i < len; ++i){        std::cout<< v[i] << " "; // 遍历v中的每一个元素并输出        // vector重载了下标运算符，所以你可以像数组一样使用它（当然，不能越界！）    }    int i = 2;    // 在vector中删除第i个元素的方式（从0开始计数）    // 请保证i < v.size()    // 在一个循环中调用如下函数可能有问题，请谨慎（本题中实际上也可以不用删除vector中元素）    v.erase(v.begin() + i); // v中现在的元素：1, 2,4,5
```

任务描述

本题中我们考虑一个形如`a0xn+a1xn−1+⋯an−1x+an`的多项式。

类`Polynomial`被用来表示上述的多项式，其构造函数如下：

- `Polynomial(const vector<int> &coefficients)`：`coefficients`表示系数`a0,a1,⋯,an`（请注意是从最高次项的系数到最低次项的系数）。注意到该构造函数前有关键字`explicit`，本题中你不需要关注该关键字，它是为了防止隐式的构造函数调用

你需要实现如下三个函数：

- `Polynomial add(const Polynomial &p1, const Polynomial &p2)`：该函数将`p1`以及`p2`按照多项式加法规则相加，并返回结果多项式，具体可见下方的样例
- `Polynomial derivate(const Polynomial &p)`：该函数对`p`进行求导，并返回结果多项式（数学小课堂：`xn（n为整数）`的一阶导数为`nxn−1`），具体可见下方的样例
- `void output() const`：该函数需要打印多项式，为了简化问题，你只需要按照从高次到低次打印**所有非0**系数以及它们对应项的次数，例如，多项式`6x3+3x+2`应该被输出为`6(3) 3(1) 2(0)`。特别地，如果一个多项式中所有的系数都为0，你不需要打印任何内容。无论是上述哪种情况，请以换行符结尾。

样例1：

输入：

```
Polynomial t({3, 0}); // 3x t.output();
```

输出：`3(1)`

说明： 我们构造了一个多项式`3x`（请注意因为有两项，所以必须使用{3,0}构造而非{3}构造，后者构造出的多项式应为`3`）并打印。由于常数项系数为0，所以只打印了一次项系数。

样例2：

输入：

```
Polynomial t({2, 1, 0}); // 2x^2 + xPolynomial t2({3}); // 3add(t, t2).output();
```

输出：`2(2) 1(1) 3(0)`

说明： 我们构造了多项式`2x2+x`以及`3`，并将两者相加得到多项式`2x2+x+3`，最后从最高次到最低次打印出所有非零系数。

样例3：

```
Polynomial t({1, 1, 1}); // x^2 + x + 1derivate(t).output(); // 2x + 1
```

输出：`2(1) 1(0)`

说明： 我们构造了多项式`x2+x+x`，对其求导得到`2x+1`，最后从最高次到最低次打印出所有非零系数。

提示

前三个样例只会测试`Polynomial`的`output`函数； 第四、五样例会测试`add`函数； 第六、七、八样例会测试`derivate`函数。

Polynomial.h

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

// ATTENTION: you should not modify any of the following interfaces
class Polynomial {
  friend Polynomial add(const Polynomial &p1, const Polynomial &p2);
  friend Polynomial derivate(const Polynomial &p);

public:
  Polynomial() {}

  explicit Polynomial(const vector<int> &coefficients);
  Polynomial(const Polynomial &other);

public:
  void output() const;

private:
  vector<int> coefficients_; // bonus: maybe more efficient data structures?
  // TODO: add any members if you want
};
```

Polynomial.cpp

```
#include "Polynomial.h"
#include <assert.h>

Polynomial::Polynomial(const vector<int> &coefficients) {
  coefficients_ = coefficients;
}

Polynomial::Polynomial(const Polynomial &other) {
  this->coefficients_ = other.coefficients_;
}

Polynomial add(const Polynomial &p1, const Polynomial &p2) {
   size_t len1 = p1.coefficients_.size();
   size_t len2 = p2.coefficients_.size();
   size_t max_len = std::max(len1,len2);
   std::vector<int> p3(max_len, 0);
   if(len1>=len2){
     for(size_t i=0;i<len1;i++){
       if(i<len1-len2){
         p3[i]=p1.coefficients_[i];
       }
       else{
         p3[i]=p1.coefficients_[i]+p2.coefficients_[i-len1+len2];
       }
     }
   }
else{
  for(size_t i=0;i<len2;i++){
       if(i<len2-len1){
         p3[i]=p2.coefficients_[i];
       }
       else{
         p3[i]=p2.coefficients_[i]+p1.coefficients_[i-len2+len1];
       }
     }
}
return Polynomial(p3);
}

Polynomial derivate(const Polynomial &p) {
   std::vector<int> p4;
    for (size_t i = 0; i < p.coefficients_.size()-1; ++i) {
        int derivative_coeff = p.coefficients_[i] * (p.coefficients_.size()-i-
1);
        if (derivative_coeff != 0) {
            p4.push_back(derivative_coeff);
        }
    }
    return Polynomial(p4);
}


void Polynomial::output() const {
  size_t len = coefficients_.size();
  for(size_t i = 0; i < len; ++i){
```

```
        if(coefficients_[i]!=0){
        cout<< coefficients_[i] << "("<<len-i-1<<")"<<" ";
        }
    }
    cout<<endl;
}
```

第4次上机

第一关：链表删除节点

任务描述

在一个由单向链表实现的队列中，给定一个char *c* , 请删除链表中所有满足content == *c* 的节点, 并输出移除的节点数。你需要保证 *head* 仍为新链表的头部，且未删除节点与先前的顺序一致。

代码文件说明

MyList.h给出了单链表MyList的类定义，你需要在MyList.cpp中完成函数 *removeNode* 的实现。其他函数的具体描述请参考代码中的注释。

示例

示例 1：

```
int main() {
    MyList mylist;
    mylist.add('N');
    mylist.add('N');
    mylist.add('J');
    mylist.add('J');
    mylist.add('U');
    mylist.add('U');
    mylist.removeNode('J'); // 移除2个含'J'的节点    mylist.printList(); // 从头部遍历打印mylist    return 0;
}
```

这段代码执行完后的输出应为：

```
2N N U U
```

示例 2：

```
int main() {
    MyList mylist;
    mylist.add('N');
    mylist.add('J');
    mylist.add('U');
    mylist.removeNode('O'); // 不存在含'O'的节点，输出0    mylist.printList(); // 从头部遍历打印mylist
    return 0;
}
```

这段代码执行完后的输出应为：

```
ON J U
```

示例 3：

```
int main() {
    MyList mylist;
    mylist.add('N');
    mylist.add('N');
    mylist.add('N');
    mylist.removeNode('N'); // 移除所有的节点
    mylist.printList(); // 从头部遍历打印mylist
    return 0;
}
```

这段代码执行完后的输出应为：

```
3
```

MyList.h

```
// MyList.h
#include<iostream>

// 单向链表实现的队列
class MyList {
private:
    struct Node
    {
        char content;
        Node *next;
    } *head; // 链表的头节点

public:
    MyList();  // 构造函数
    ~MyList(); // 析构函数
    void add(char c);  // 添加新的节点
    void removeNode(char c); // TODO: 移除所有content为c的节点
    void printList(); // 从表头开始顺序打印元素
};
```

MyList.cpp

```
// MyList.cpp
#include "MyList.h"

using namespace std;

// 构造函数
MyList::MyList()
{
    head = NULL;
}

// 析构函数
```

```
MyList::~MyList()
{
    while (head != NULL)
    {
        Node *n = head;
        head = head->next;
        delete n;
    }
}

// 添加新的节点
void MyList::add(char c)
{
    Node *n = new Node;
    if (n == NULL)
    {
        cout << "Overflow\n";
        exit(-1);
    }
    else {
        n->content = c;
        n->next = NULL;
        if (head == NULL) {
            head = n;
        }
        else {
            Node *tail = head;
            while (tail->next != NULL) {
                tail = tail->next;
            }
            tail->next = n;
        }
    }
}

/* TODO:
 * 移除所有content等于c的节点，没有则不移除
 * 确保 head 仍是新链表的头部，且未删除节点的顺序保持一致
 * 输出移除的节点数量(number)
 */
void MyList::removeNode(char c)
{
    int number = 0;
    Node *p=new Node;
    p=head;
    while(p!=NULL){
        if(p->content==c&&p==head){
            head=head->next;
            delete p;
            Node *p=new Node;
            p=head;
            number++;
        }
        else if(p->content==c){
            Node *q=head;
            Node *t=head;
            while(q!=p){
                q=q->next;
```

```
            }
            while(t->next!=p){
                t=t->next;
            }
            p=p->next;
            t->next=p;
            delete q;
            number++;
        }
        else{
            p=p->next;
        }
    }
    cout << number << endl;
}

// 从表头开始按加入顺序打印元素
void MyList::printList()
{
    Node* n = head;
    while (n != NULL)
    {
        cout << n->content << " ";
        n = n->next;
    }
    cout << endl;
}
```

第二关：时间解析器

任务描述

你需要自行设计实现一个时间字符串的解析器类 *TimeParser*

用户可以通过键盘输入格式为“**HH:MM:SS**”代表24小时制的 **时：分：秒** 的字符串。你需要设计一个类解析该字符串所代表的时间。

任务要求

1. 解析器需要提供一个有参构造函数，接受字符串参数；
2. 解析器需要提供三个函数: **int getHour(), int getMin(), int getSec()** 以分别返回解析得到的时/分/秒整数值；
3. 当用户输入不满足以下条件时，以上三个函数都应输出 -1：
    1. 不满足时间有效值域（00：00：00 ~ 23：59：59），如“24：60：60”;
    2. 小于10的值未补零，如“5：6：7”; 若为“05：06：07”则能正常解析；
    3. 使用了非“：”的分隔符，如“23-59-59”;
4. 除上述的三种情况，你可以假设用户输入都合法且能被正常解析。

代码文件说明

TimeParser.h给出了部分成员函数定义，你可以补全所需的成员变量并在TimeParser.cpp中完成相关函数的实现。

提示

- string类中提供了substr()成员函数以提取部分字符串 https://cplusplus.com/reference/string/string/substr/
- 标准库中的std::stoi()函数可以将字符串转换为整数 https://cplusplus.com/reference/string/stoi/

示例

这里给出部分测试用例，可以参考该用例设计实现解析器类：

```
int main() {
    std::string s1 = “10:14:01”;
    TimeParser tp(s1);
    std::cout << tp.getHour() << ' ';    // 返回 10
    std::cout << tp.getMin() << ' ';    // 返回 14
    std::cout << tp.getSec() << endl;    // 返回 1
    std::string s2 = “24:60:60”; // 违反了上述要求，输出-1
    TimeParser tp2(s2);    std::cout << tp2.getHour() << ' '; // -1
    std::cout << tp2.getMin() << ' '; // -1
    std::cout << tp2.getSec() << endl; // -1
    return 0;
}
```

这段代码执行完后的输出应为：

```
10 14 1-1 -1 -1
```

你也可以使用以下的主函数在你自己的IDE中进行调试，但请注意你在系统上的提交请不要包含main函数。(代码中 &gt 与 &lt 请分别替换成 > 和 <)

```
int main() {
    std::string s;
    std::cin >> s;
    TimeParser tp(s);
    std::cout << tp.getHour() << ' ';
    std::cout << tp.getMin() << ' ';
    std::cout << tp.getSec() << endl;
    return 0;
}
```

TimeParser.h

```
#include <iostream>
#include <string>

using namespace std;

// TODO: 补全TimeParser类并实现以下函数
class TimeParser {
private:
        string str;
        int hour;
        int min;
        int sec;
        int len;
```

```
public:
        TimeParser(string s);  // 有参构造函数
        int getHour(); // 返回时
        int getMin(); // 返回分
        int getSec(); // 返回秒
};
```

TimeParser.cpp

```
#include <iostream>
#include "TimeParser.h"

using namespace std;

TimeParser::TimeParser(string s)
{
    str=s;
    len=str.size();
}

int TimeParser::getHour()
{
    if(len!=8){
        return -1;
    }
    int count=0;
    for(int i=0;i<8;i++){
        if(str[i]==':'){
            count++;
        }
    }
    if(count!=2){
        return -1;
    }

    std::string str1= str.substr(0,2);
    hour=std::stoi (str1);
    std::string str2= str.substr(3,2);
    min=std::stoi (str2);
    std::string str3= str.substr(6,2);
    sec=std::stoi (str3);
    if(hour>=0&&hour<=23&&min>=0&&min<=59&&sec>=0&&sec<=59){
        return hour;
    }
    else{
        hour=-1;
        return hour;
    }
}

int TimeParser::getMin()
{
   if(len!=8){
       return -1;
   }
    int count=0;
    for(int i=0;i<8;i++){
```

```
        if(str[i]==':'){
            count++;
        }
    }
    if(count!=2){
        return -1;
    }

    if(hour>=0&&hour<=23&&min>=0&&min<=59&&sec>=0&&sec<=59){
        return min;
    }
    else{
        min=-1;
        return min;
    }
}

int TimeParser::getSec()
{
    if(len!=8){
        return -1;
    }
    int count=0;
    for(int i=0;i<8;i++){
        if(str[i]==':'){
            count++;
        }
    }
    if(count!=2){
        return -1;
    }

    if(hour>=0&&hour<=23&&min>=0&&min<=59&&sec>=0&&sec<=59){
        return sec;
    }
    else{
        sec=-1;
        return sec;
    }
}
```

第5次上机

第一关：多继承α

任务描述：

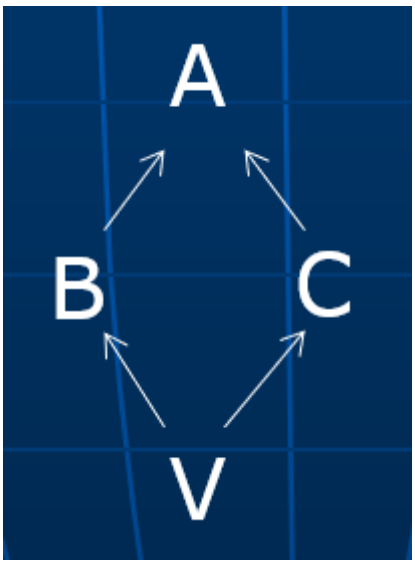
模拟实现以下类并完成相应功能： 交通工具类（Vehicle）描述了不同交通工具的共有特点。汽车（Car）、船（Boat）、水陆两用汽车（AmphibianCar）分别描述了三种不同的交通工具。具体说明如下：

1. Vehicle类为Car和Boat的抽象基类，有一个访问权限为protected、string类型的成员 name。
2. Vehicle类有一个drive() 纯虚函数，不同子类的重写（overwrite）方式不同。
3. AmphibianCar类继承Car和Boat两个基类，且需要实现drive函数的**重写**。
4. drive()方法的具体实现可参考测试用例输出。

"Vehicle.h" 头文件代码：

```
#include <iostream>
#include <string>using namespace std;
class Vehicle {
    protected:
    string name;
    public:
    Vehicle(string name): name(name) {};
    virtual void drive() = 0;
};
```

考察要点：菱形继承



如图所示，B和C从V中继承，A继承于B和C，这样的继承方式就会导致A中有V的2个拷贝，那么在访问V的成员时就会出现访问不明确 *(Member is ambiguous)*的二义性问题。解决这种问题的一种方式是使用**虚继承**，详细内容可参考课程网站PPT。

样例1：

输入：

```
  Vehicle* C = new Car("My car");
  C->drive(); // 输出Vehicle name + " drive on road"
  Vehicle* B = new Boat("My boat");
  B->drive(); // 输出Vehicle name + " drive on river"
```

输出：

```
My car drive on road
My boat drive on river
```

样例2：

输入：

```
  AmphibianCar* A = new AmphibianCar("My amphibian car");
  A->drive(); // 输出Vehicle name + " drive on road or river"
  A->Car::drive(); // 调用父类同名函数  A->Boat::drive(); // 同上
```

输出：

```
My amphibian car drive on road or river
My amphibian car drive on road
My amphibian car drive on river
```

Vehicle.cpp

```
#include "Vehicle.h"
class Car;
class Boat;
class AmphibianCar;

class Car : public Vehicle{
public:
     Car(string name) : Vehicle(name){};
     virtual void driveAsCar(){
       cout << name << " drive on road" << endl;
     }
     void drive(){
        driveAsCar();
     }
};

class Boat : public Vehicle{
public: Boat(string name) : Vehicle(name){};
     virtual void driveAsBoat(){
        cout << name << " drive on river" << endl;
     }
   void drive(){
      driveAsBoat();
     }
};

class AmphibianCar : public Car, public Boat{
public:
    AmphibianCar(string name) : Car(name), Boat(name){};
    void driveAsCar(){
      cout << Car::name << " drive on road as car" << endl;
    }
    void driveAsBoat(){
        cout << Boat::name << " drive on river as boat" << endl;
    }
};
```

第二关：多继承β

任务描述：

基于上题的情景，**重新设计**交通工具的子类，具体说明如下：

1. 题目提供的Vehicle类与之前一致。
2. Car与Boat对象调用drive()方法时**输出**保持不变。（参考样例1）
3. 设计实现AmphibianCar类，使得其满足以下要求：

- AmphibianCar类仍继承Car和Boat两个基类, 并额外实现2个方法 **driveAsCar**()与**driveAsBoat**()（参考样例1）

- 当通过不同的父类指针进行动态绑定时，drive()函数将根据父类的指针类型输出不同内容，以体现**多态性**。具体来说，当父类指针为Car时，drive()的输出与**driveAsCar**()一致；为Boat时，与**driveAsBoat**()一致 （参考样例2）

  你应该明白，虽然水陆两用车可以在水上航行也可以在路上驾驶，但它不可能同时出现在水里和路上。

提示：

1. 原则上可把此题与上题看做无关的两题，需要的话可以重新设计基类的继承方式与结构等。
2. 测试用例中，AmphibianCar类的指针或对象不会直接调用drive()方法。
3. 虚继承与普通多继承有什么区别？使用虚继承会解决/带来什么问题？多继承带来的问题还有哪些解决方式？

样例1：

输入：

```
  Vehicle* C = new Car("My car");
  C->drive(); // 输出Vehicle name + " drive on road"
  Vehicle* B = new Boat("My boat");  B->drive(); // 输出Vehicle name + " drive on river"
  AmphibianCar* A = new AmphibianCar("My amphibian car");
  A->driveAsCar(); // 输出Vehicle name + " drive on road as car"
  A->driveAsBoat(); // 输出Vehicle name + " drive on river as boat"
```

输出：

```
My car drive on road
My boat drive on river
My amphibian car drive on road as car
My amphibian car drive on river as boat
```

样例2：

输入：

```
  AmphibianCar* A = new AmphibianCar("My amphibian car");
  Car* CarMode = A;  Boat* BoatMode = A;
  CarMode->drive(); // 将Car中的drive()方法重写为driveAsCar()
  BoatMode->drive(); // 将Boat中的drive()方法重写为driveAsBoat()
```

输出：

```
My amphibian car drive on road as car
My amphibian car drive on river as boat
```

Vehicle.h

```
#include "Vehicle.h"
class Car;
class Boat;
class AmphibianCar;

class Car : public Vehicle{
```

```
public:
     Car(string name) : Vehicle(name){};
     virtual void driveAsCar(){
       cout << name << " drive on road" << endl;
     }
     void drive(){
        driveAsCar();
     }
};

class Boat : public Vehicle{
public: Boat(string name) : Vehicle(name){};
     virtual void driveAsBoat(){
        cout << name << " drive on river" << endl;
     }
   void drive(){
      driveAsBoat();
     }
};

class AmphibianCar : public Car, public Boat{
public:
    AmphibianCar(string name) : Car(name), Boat(name){};
    void driveAsCar(){
      cout << Car::name << " drive on road as car" << endl;
    }
    void driveAsBoat(){
        cout << Boat::name << " drive on river as boat" << endl;
    }
};
```

第6次上机

第一关：计算器

任务描述

`AddCalculator` 和 `SubCalculator` 继承自 `BaseCalculator`，分别实现加法运算和减法运算。

请完善代码，使 `AddCalculator::calculate` 计算加法，`SubCalculator::calculate` 计算减法，它们重载的操作符 `<<` 能够按照顺序输出计算过的所有结果，每行一个，保留两位小数。

相关知识

`ostringstream`

定义在头文件 `sstream` 中。`ostringstream` 类继承自 `ostream` 类，它提供了一种方便的方法来将数据写入到一个字符串中。你可以使用 `<<` 运算符向 `ostringstream` 对象中写入数据，然后使用 `str()` 成员函数来获取包含写入数据的字符串。

`setprecision`

定义在头文件 `iomanip` 中。如果 `setprecision` 与 `fixed` 合用，可以控制小数点右边的数字个数。例如：

```
cout << fixed << setprecision(2) << 2.71828 << endl;
```

会输出：`2.72`

样例：

输入：

```
BaseCalculator* cal1 = new AddCalculator();
BaseCalculator* cal2 = new SubCalculator();
cal2->calculate(11.4, 51.4);
cout << cal2;cal1->calculate(23.33, 15.51);
cal1->calculate(3.14159, 2.71828);
cout << cal1;cal2->calculate(834047409, 510756141);
cout << cal2;
```

输出：

```
-40.00
38.84
5.86
-40.00
323291268.00
```

测试说明

保证过程中所有浮点数在`double`表示范围内

Calculator.h

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

class BaseCalculator {
protected:
public:
    ostringstream oss;
    virtual void calculate(double, double){}
    friend ostream &operator<<(ostream &, const BaseCalculator*);
};

// 加法计算器类
class AddCalculator : public BaseCalculator {
    void calculate(double, double);
};

// 减法计算器类
class SubCalculator : public BaseCalculator {
    void calculate(double, double);
};
```

Calculator.cpp

```
#include "Calculator.h"
#include <iomanip>
ostream & operator<<(ostream & out, const BaseCalculator* cal) {
    out << cal->oss.str();
    return out;
}

void AddCalculator::calculate(double a, double b) {
    double result = a + b;
    oss << fixed << setprecision(2) << result << endl;
}

void SubCalculator::calculate(double a, double b) {
    double result = a - b;
    oss << fixed << setprecision(2) << result << endl;
}
```

第二关：vector上的链式调用

考察知识点

基本C++知识；函数指针

任务描述

假设我们有一个元素类型为`int`的数组，如果我们希望筛选出其中所有的偶数组成一个新的数组，一个可能的写法如下：

```
for(int i = 0; i < len; ++i){
    if (arr[i]%2 == 0){
        // do your stuff ...
    }
}
```

如果需要筛选的条件变得更多（同时是正数、最后一位为4……），你可能就需要再使用一个循环，或者在上面的循环体内增加新的筛选条件。现实中，想必你常常因为编写这样的循环而感到懊恼（当然现在有了chatGPT，一天写一百个也不是问题！），有时还会遇到数组访问越界的问题……

让我们实现是一个支持链式调用的`Vector`类，你可以通过如下的方式从原数组中获取所有的偶数：

```
bool is_even(int x){
    return x % 2 == 0;
}
Vector better_vec({1, 2, 3, 4});
better_vec.filter(is_even).output(); // 2 4
```

其中，`Vector`的成员函数`filter`函数接受一个传入的函数指针（相信经过上一次OJ你已经知道它是什么了），它用来过滤数组中的所有元素，并返回由所有满足`is_even`的元素组成的新`Vector`；`output`也是`Vector`的成员函数，用于打印`Vector`中的所有整数。

聪明的你可能已经猜到了，`filter`的返回值是一个`Vector`，正因如此我们才能像上面一样调用`output`，我们把这种调用风格叫做**链式调用**。我们甚至可以加上多个`filter`，像这样：`v.filter(large_than_zero).filter(is_even).output()`。

你的任务是实现下面四个函数（别担心！它们每个实现起来都不会超过10行代码）：

- `Vector filter(filter_func f)`：返回一个新的`Vector`对象。你需要遍历原`Vector`中的所有整数，并将满足`f(element) == true`的整数组成一个新的`Vector`（原`Vector`内容不应发生变化）。其中，`filter_func`的定义已经在`Vector.h`中给出，它接受一个`int`变量，返回一个`bool`值，`filter_func`判断变量是否符合条件，例如`bool le_zero(int x){return x > 0;}`
- `Vector map(map_func f)`：返回一个新的`Vector`对象，你需要遍历原`Vector`中所有的元素，对其调用`f`并将其结果（也就是`f(element)`）组成一个新的`Vector`（原`Vector`内容不应发生变化）。其中，`map_func`的定义已经在`Vector.h`中给出，它接受一个`int`变量，返回一个`int`值，`map_func`描述了怎么将一个`int`变量变换为另一个`int`变量，例如`int mul_ten(x) {return x*10;}`
- `Vector &for_each(map_func f)`：返回当前`Vector`对象的引用，与`map`类似，不过你需要直接在当前`Vector`上应用`f`。其中，`map_func`的定义已经在`Vector.h`中给出，它接受一个`int`变量，返回一个`int`值
- `void output()`:打印当前`Vector`中的所有元素，按顺序打印，每两个元素之间用一个空格间隔，最后需要打印换行符

`Vector`的构造函数已经在`Vector.h`中给出。你可以将其看做是一个`vector<int>`的简单封装，不过`Vector`一旦被构造，其中的元素数量就固定了，因为我们没有提供任何增加或删除元素的接口。

你可以在你的测试中加入如下的测试函数（真实的测试中可能不会使用这两个函数）：

```
// map funcint neg(int x) { return -x; }
// filter funcbool is_neg(int x) { return x < 0; }
```

以下的样例假设你已经在测试代码中加入了如上的两个函数。

样例1

输入：

```
Vector foo({-1, 0, 1});
foo.map(neg).output();
foo.output();
```

输出：

```
1 0 -1
-1 0 1
```

说明： 我们构造了一个`Vector`，其中有-1，0，1三个整数，并调用`map`将其中所有元素取负，最后调用`output`输出所有整数。

由于`map`返回的是一个新的`Vector`，经过以上的操作之后原`Vector`中的内容不应该有变化，因此我们对原`Vector`调用`output`后仍打印`-1 0 1`

样例2

输入：

```
Vector foo({-1, -2, 0, 1, 3});
foo.map(neg).filter(is_neg).output();
foo.filter(is_neg).output();
```

输出：

```
-1 -3
-1 -2
```

说明： 我们构造了一个`Vector`，其中有-1，-2, 0，1,3五个整数，并调用`map`将其中所有整数取负，接着使用`filter`筛选出其中为负数的整数，最后调用`output`输出所有整数。

由于`map`返回的是一个新`Vector`，我们对原`Vector`调用`filter`筛选出其中为负数的整数，最后打印出应为`-1 -2`

样例3

输入：

```
Vector foo({-1, -2, 0, 1, 3});
foo.for_each(neg).filter(is_neg).output();
foo.output();
```

输出：

```
-1 -3
1 2 0 -1 -3
```

说明： 与样例二类似，但是由于我们对原`Vector`调用了`for_each`，因此第二次调用output时打印出的`Vector`内容应该已经发生了变化，为逐个元素取负之后的`1 2 0 -1 -3`

提示

每个样例都会测试`output`。 第一个样例只会额外测试`map`； 第二个样例只会额外测试`filter`； 第三、四个样例会测试`map`与`filter`链式调用； 第五、六个样例会测试`for_each`、`map`与`filter`链式调用。

Vector.h

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

typedef int (*map_func)(int);
typedef bool (*filter_func)(int);

class Vector {
public:
  Vector(const vector<int> &vec) : vec_(vec) {}

  Vector filter(filter_func f) const;
  Vector map(map_func f) const;
  Vector &for_each(map_func f);

  void output() const;

private:
  vector<int> vec_;
};
```

Vector.cpp

```
#include "Vector.h"
#include <iostream>

// map func
int neg(int x) { return -x; }
// filter func
bool is_neg(int x) { return x < 0; }

Vector Vector::map(map_func f) const {
    vector<int> n;
    int j=0;
    for (size_t i = 0; i < vec_.size(); ++i) {
            n.push_back(f(vec_[i]));
            j++;
    }
    return Vector(n);
}

Vector Vector::filter(filter_func f) const {
    vector<int> n;
    int j=0;
    for (size_t i = 0; i < vec_.size(); ++i) {
        if(f(vec_[i])==1){
            n.push_back(vec_[i]);
            j++;
        }
    }
    return Vector(n);
}

Vector &Vector::for_each(map_func f) {
    for (size_t i = 0; i < vec_.size(); ++i) {
        vec_[i]=f(vec_[i]);
    }
    return *this;
}

void Vector::output() const {
    for (size_t i = 0; i < vec_.size(); ++i) {
        if(vec_[i]==90){
            break;
        }
        cout<<vec_[i]<<" ";
    }
    cout<<endl;
}
```

第三关：迭代器

任务描述

迭代器是容器类（如数组，链表，队列，栈等）提供的、用于遍历和访问容器内对象的辅助类。在本题中，我们考虑一种简化的迭代器，它为我们自定义的整型数组类提供元素的遍历和访问。 自定义的整型数组类如下：

```
class MyArray{
    int * arr;
    int size;public:
    MyArray(int size);
    ~MyArray();
};
```

它根据参数创建和维护特定长度的动态整型数组，并且提供迭代器`Iterator`来访问数组`arr`中的元素。由于迭代器常与特定类关联，所以我们使用`MyArray`的内部类来实现`Iterator`，定义如下：

```
class MyArray {
    int * arr;
    int size;
     public:
       class Iterator{
         ...
       public:
           ...
           bool get(int &value) const;
           bool put(int value);
           ...
       };
    public:
    MyArray(int size);
    ~MyArray();
    Iterator begin();
    Iterator end();
}
```

可以看到，`MyArray`类新增了两个接口`Iterator begin()`和`Iterator end()`来提供两个特殊的迭代器对象：

- 每个迭代器对象都提供两个接口`bool get(int &value)`和`bool put(int value)`
    - ```
      bool get(int &value)
      ```

      ：获取当前迭代器所指向的元素
        - 如果迭代器位于一个合法的位置：`arr[0]`-`arr[size-1]`，则通过`value`返回对应元素，返回值为`true`
        - 否则不改变`value`的值，返回`false`
    - ```
      bool put(int value)
      ```

      ：向迭代器指向的元素存放值

      ```
      value
      ```

        - 如果迭代器位于一个合法的位置：`arr[0]`-`arr[size-1]`，则将对应数组元素的值更新为`value`，返回`true`
        - 否则不改变任何值，返回`false`
- `Iterator begin()`：返回的迭代器对象，指向数组的第一个元素`arr[0]`
    - 具体而言，下列代码中`iter.get(v)`获取的应该是`ma.arr[0]`的值

```
MyArray ma(10);
MyArray::Iterator iter = ma.begin();
iter.put(20); // true, set ma.arr[0]=20
int v = -1;
iter.get(v); // true, v=20
```

    - 当然，你需要自己思考，如果`MyArray`类中的`arr`数组为空（长度为0），`Iterator begin()`返回的迭代器应该指向哪儿?
- `Iterator end()`：返回的迭代器对象，指向数组`arr`的末尾，但不是任何数组中的元素
    - 末尾实际上仅仅起到一个标记遍历结束的作用，它并不对应任何数组元素。当然你可以把它设置成指向`arr[size]`，但注意用该迭代器访问数组元素(`get`和`put`)是无法成功且没有意义的。
    - 具体而言，我们通常会看到迭代器的如下用法，实现遍历容器中的每个元素：

```
for(MyArray::Iterator iter = ma.begin(); iter != ma.end(); iter++){
int v;    iter.get(v);    ...}
```

      通过判断迭代器`iter`是不是迭代到了容器的末尾，让整个遍历可以结束。

或许你已经意识到，为了使迭代器更加方便使用，我们需要为其重载一些操作符，它们的含义如下：

- `operator++`
    - 将迭代器指向数组中下一个相邻元素
    - 考虑前置和后置
- `operator--`
    - 将迭代器指向数组中上一个相邻元素
    - 考虑前置和后置
- `operator==`
    - 判断两个迭代器是否指向同一个位置（元素）
- `operator!=`
    - 与上一个操作符相反的语义
- `operator+(int len)`
    - 将迭代器移动到当前元素向后的第`len`个元素
- `operator-(int len)`
    - 将迭代器移动到当前元素向前的第`len`个元素

注意

- 数组的第一个元素 `arr[0]` 和最后一个元素 `arr[size-1]` 并不相邻（数组不是环状的），所以你需要考虑上述操作符在跨过数组前后边界时应该如何处理。要求只有一个：跨过数组边界的迭代器指向非法位置，这些迭代器不能用于访问数组元素——调用 `get` 和 `put` 时返回值为 `false`。
- 元素 `arr[size-1]` 的下一个"相邻元素"是末尾（也就是 `end()` 迭代器指向的元素），末尾的上一个相邻元素是 `arr[size-1]`。这样的设计会使通过迭代器从头到尾遍历元素更自然。

你的任务

- 完善 `MyArray` 类中的4个成员函数。**注意：`MyArray` 类的对象析构时，其所有的迭代器也应随之失效——调用 `get` 和 `put` 时返回值为 `false`。**
- 自定义 `Iterator` 类的数据成员，并添加相应的构造和析构函数。实现 `get` 和 `put` 两个成员函数，并添加前述所列的操作符重载，使得上述关于 `Iterator` 的功能可以满足。

（仅作）思考

你或许已经发现，通过迭代器访问容器中的对象和通过指针访问非常的相似，能否通过操作符重载进一步简化 `get` 和 `put`？通过迭代器访问相比于通过指针访问，有哪些优劣？

iterator.h

```
class MyArray {
    int* arr;
    int size;
    bool tf;
public:
    class Iterator {
    private:
        int* ptr;
        int* begin_ptr;
        int* end_ptr;
        bool* tf_ptr;
    public:
        Iterator(int* p, int* begin, int* end,bool *Tf)
                : ptr(p), begin_ptr(begin), end_ptr(end), tf_ptr(Tf){}

        bool get(int& value) const {
          if(ptr==nullptr){
            return false;
          }
            if (ptr >= begin_ptr && ptr < end_ptr ) {
                value = *ptr;
                return true;
            } else {
                return false;
            }
        }

        bool put(int value) {
           if(ptr==nullptr){
            return false;
          }
            if (ptr >= begin_ptr && ptr < end_ptr ) {
                *ptr = value;
                return true;
```

```
            } else {
                return false;
            }
        }

        Iterator& operator++() {
            if (ptr < end_ptr) {
                ++ptr;
            }
            return *this;
        }

        Iterator operator++(int) {
            Iterator temp = *this;
            if (ptr < end_ptr) {
                ++ptr;
            }
            return temp;
        }

        Iterator& operator--() {
            if (ptr > begin_ptr) {
                --ptr;
            }
            return *this;
        }

        Iterator operator--(int) {
            Iterator temp = *this;
            if (ptr > begin_ptr) {
                --ptr;
            }
            return temp;
        }

        bool operator==(const Iterator& other) const {
            return ptr == other.ptr;
        }

        bool operator!=(const Iterator& other) const {
            return ptr != other.ptr;
        }

        Iterator operator+(int len) {
            int* new_ptr = ptr + len;
            if (new_ptr >= end_ptr) {
                new_ptr = nullptr;
            }
            return Iterator(new_ptr, begin_ptr, end_ptr,tf_ptr);
        }

        Iterator operator-(int len) {
            int* new_ptr = ptr - len;
            if (new_ptr < begin_ptr) {
                new_ptr = nullptr;
            }
            return Iterator(new_ptr, begin_ptr, end_ptr,tf_ptr);
        }
```

```
    };

public:
    MyArray(int sz) {
        arr = new int[sz];
        size = sz;
        tf=true;
    }

    ~MyArray() {
        delete[] arr;
        size = 0;
        tf=false;
    }

    Iterator begin() {
        if(arr==nullptr){
          return Iterator(nullptr,nullptr,nullptr,&tf);
        }
        return Iterator(arr, arr, arr + size,&tf);
    }

    Iterator end(){
         if(arr==nullptr){
          return Iterator(nullptr,nullptr,nullptr,&tf);
        }
        return Iterator(arr + size, arr, arr + size,&tf);
    }
};
```

作业二编程题

第一关：MyStack

任务描述

本题需要你完成MyStack类：栈的一种实现（链表表示）。

代码文件说明

本题为你提供了两份代码文件：MyStack.h和MyStack.cpp。 MyStack.h给出了MyStack类定义，而你需要在MyStack.cpp中补全MyStack类的成员函数的函数体。函数的具体描述已经在代码中用注释给出。

类似第0次上机测试（单调栈），需要你实现的部分已经在MyStack.cpp中用TODO注释标出，你只需要关注这些函数的实现，而无需关注测试的输入和输出。

提示：你可以根据自己的需要，在MyStack类的定义中添加其他合适的数据成员或者成员函数。

示例

为了便于理解，我们给出一个简单的MyStack的使用实例。

请注意：由于系统问题，代码块中的小于号'<'会被显示成`<`，如果你需要使用到以下代码，请自行修改对应位置的符号！

```
int main() {
    MyStack s;
    s.push('N');
    s.push('J');
    s.push('U');
    cout << s.size() << endl; // 3
    cout << s.pop() << endl; // U
    cout << s.pop() << endl; // J
    cout << s.pop() << endl; // N
    cout << s.size() << endl; // 0
    return 0;
}
```

这段代码执行完后的输出应为：

```
3
U
J
N
0
```

你可以使用类似的方式在你自己的IDE中进行调试或测试，但请注意你在系统上的提交请不要包含main函数。

MyStack.h

```
// MyStack.h
#include<iostream>

class MyStack {
    // 用链表实现“栈”
private:
    struct Node
    {
        char content;
        Node *next;
    } *top;

public:
    MyStack();  // 构造函数
    ~MyStack(); // 析构函数，你需要在这里归还额外申请的资源
    void push(char c);  // 字符c入栈
    char pop();         // 栈顶元素出栈，返回出栈元素（我们没有定义空栈pop操作，测试用例中不会涉及）
    int size();         // 返回栈的大小（栈内元素数量）

    // 请注意，以上函数的函数体请你在MyStack.cpp中实现！
};
```

MyStack.cpp

```
// MyStack.cpp
#include "MyStack.h"

using namespace std;
```

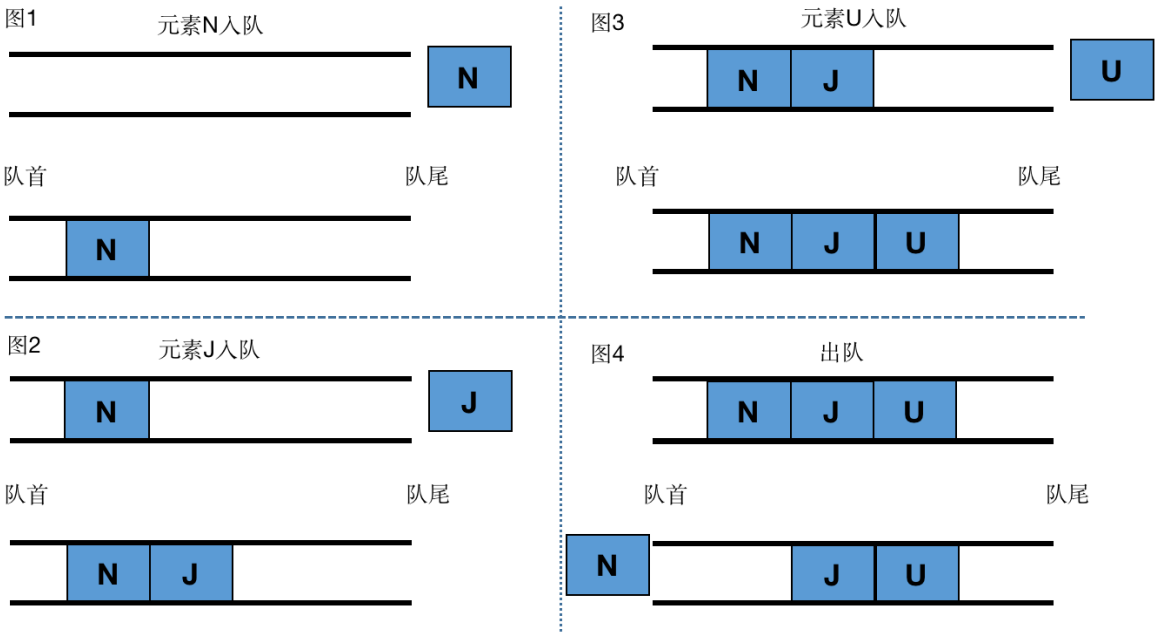
```
// 构造函数
MyStack::MyStack()
{
    top=NULL;
}

// 析构函数，你需要在这里归还额外申请的资源
MyStack::~MyStack()
{
    while(top!=NULL){
        Node *p=top;
        top=top->next;
        delete p;
    }
}

// 字符c入栈
void MyStack::push(char c)
{
    Node *p=new Node;
    p->content=c;
    p->next=top;
    top=p;
}

// 栈顶元素出栈，返回出栈元素（我们没有定义空栈pop操作，测试用例中不会涉及）
char MyStack::pop()
{
    char result = 0;
    result=top->content;
    Node *p=top;
    top=top->next;
    delete p;
    return result;
}

// 返回栈的大小（栈内元素数量）
int MyStack::size()
{
    int result = 0;
    Node *p=top;
    while(p!=NULL){
        p=p->next;
        result++;
    }
    return result;
}
```

第二关：MyQueue

任务描述

这道题需要你完成MyQueue类：队列的一种实现。

栈是一种后进先出（LIFO）的数据结构，而队列则是一种先进先出（FIFO）的数据结构：元素总是从队列的队尾入队，从队列的队首出队。下图给出了队列的出入队逻辑：



代码文件说明

本题为你提供了两份代码文件：MyQueue.h和MyQueue.cpp。 MyQueue.h仅给出了MyQueue类的部分定义（包含部分成员函数的声明），你需要在MyQueue.h中补全MyQueue类的定义（如，补充必要的数据成员），并且需要在MyQueue.cpp中补全相关函数的实现。函数的具体描述已经在代码中用注释给出。

同样地，需要你实现的部分已经在MyQueue.h和MyQueue.cpp中用TODO注释标出，你无需关注测试的输入和输出。

提示：你可以根据自己的需要，在MyQueue类的定义中添加其他合适的数据成员或者成员函数。

示例

为了便于理解，我们给出一个简单的MyQueue的使用实例。

请注意：由于系统问题，代码块中的小于号'<'会被显示成`<`，如果你需要使用到以下代码，请自行修改对应位置的符号！

```
int main() {
    MyQueue q;
    q.push('N');
    q.push('J');
    q.push('U');
    cout << q.size() << endl; // 3
    cout << q.pop() << endl; // N
    cout << q.pop() << endl; // J
    cout << q.pop() << endl; // U
    cout << q.size() << endl; // 0
    return 0;
}
```

这段代码执行完后的输出应为：

```
3
N
J
U
0
```

你可以使用类似的方式在你自己的IDE中进行调试或测试，但请注意你在系统上的提交请不要包含main函数。

MyQueue.h

```
// MyQueue.h
#include<iostream>
#include "MyStack100.h"

class MyQueue {
private:
   MyStack stack1;
   MyStack stack2;
    // 你可以用任何数据结构实现“队列”
    // 我们鼓励你使用MyStack类：请思考，如何维护两个栈结构，以满足队列结构的要求？
    // 我们帮你引用了头文件“MyStack100.h”，这份头文件中含有MyStack类的满分实现，你可以放心使用
    // 你也可以用其他方式实现队列，比如数组或链表。
public:
    MyQueue();  // 构造函数
               // 我们对MyQueue的析构函数没有做具体要求，你可以按需自行定义实现。
    void push(char c); // 元素从队尾入队
    char pop();        // 队首元素出队，返回出队的元素（我们没有定义空队列pop操作，测试用例中不会涉及）
    int size();        // 返回队列的大小

    // 请注意，以上函数的函数体请你在MyQueue.cpp中实现！
};
```

MyQueue.cpp

```
// MyQueue.cpp
#include "MyQueue.h"

MyQueue::MyQueue()
{
}
// 元素从队尾入队
void MyQueue::push(char c)
{
        stack1.push(c);
}
// 队首元素出队，返回出队的元素（我们没有定义空队列pop操作，测试用例中不会涉及）
char MyQueue::pop()
{
   char result;
   if(stack2.size()==0){
     while(stack1.size()!=0){

     stack2.push(stack1.pop());
```

```
     }
     result=stack2.pop();
     }
     else{
         result=stack2.pop();
     }
    /*else if(stack1.size()==0){
     while(stack2.size()!=0){

     stack1.push(stack2.pop());
     }
     result=stack1.pop();
     }*/

    return result;
}
// 返回队列的大小（队列内元素的个数）
int MyQueue::size()
{
    int result;
    result=stack1.size()+stack2.size();
    return result;
}
```

作业三编程题

第一关：Matrix

任务描述

补充下面的代码，使其得到对应输出。只需要给出补充代码，测试框架仅测试能使下列代码成功运行的必要功能，不会测试额外功能

```
#include <iostream>
using namespace std;
// 补充代码
int main(){
    Matrix m1(3,4);
    int i, j;
    for(i = 0; i < 3; ++i){
        for(j = 0; j < 4; j++){
            m1[i][j] = i * 4 + j;
        }
    }
    for(i = 0; i < 3; ++i){
        for(j = 0; j < 4; j++){
            cout << m1(i,j) << ", ";
        }
        cout << endl;
    }
    cout << "next" << endl;
    Matrix m2;
    m2 = m1;
    for(i = 0; i < 3; ++i){
        for(j = 0; j < 4; j++){
            cout << m2[i][j] << ",";
```

```
            m2[i][j] += 1;
        }
        cout << endl;
    }
    cout << "next" << endl;
    for(i = 0; i < 3; ++i){
        for(j = 0; j < 4; j++){
            cout << m1(i,j) << ", ";
        }
        cout << endl;
    }
    cout << "next" << endl;
    for(i = 0; i < 3; ++i){
        for(j = 0; j < 4; j++){
            cout << m2(i,j) << ", ";
        }
        cout << endl;
    }
    return 0;
}
```

输入：无

输出：

```
0,1,2,3,
4,5,6,7,
8,9,10,11,
next
0,1,2,3,
4,5,6,7,
8,9,10,11,
next
0,1,2,3,
4,5,6,7,
8,9,10,11,
next
1,2,3,4,
5,6,7,8,
9,10,11,12,
```

Matrix.h

```
class Matrix {
private:
    int row, col;
    int **matrix;
public:
    Matrix(){
        matrix=nullptr;
        row=0;
        col=0;
    }
    Matrix(int r,int c){
        row=r;
        col=c;
```

```
        matrix=new int *[row];
        for(int i=0;i<row;i++){
            matrix[i]=new int[col];
        }
    }
    ~Matrix(){
        for(int i=0;i<row;i++){
            delete[] matrix[i];
        }
        delete[] matrix;
    }
    int &operator()(int i,int j){
        return matrix[i][j];
    }
    int *operator[](int i){
        return matrix[i];
    }
    Matrix &operator=(const Matrix& other){
        row=other.row;
        col=other.col;
        matrix=new int *[row];
        for(int i=0;i<row;i++){
            matrix[i]=new int[col];
        }
        for(int i=0;i<row;i++){
            for(int j=0;j<col;j++){
                matrix[i][j]=other.matrix[i][j];
            }
        }
        return *this;
    }
};
```

第二关：Plant

任务描述

我们有两个植物类`Plant1`和`Plant2`，它们都有一种属性叫价格，但`Plant1`类的植物拥有生命值的属性，而`Plant2`类的植物没有。在某些情况下，我们需要对这两类的植物一起进行操作。

需要实现以下几个函数：

```
//植物被x点伤害攻击，生命值应下降x，如果生命值<=0，那么植物死亡（Plant2不会因为攻击而死亡）
void Plant1::beAttacked(int x);
//构造函数，price为植物价格，hp为初始生命值
Plant1::Plant1(int price,int hp);
//析构函数，会直接调用析构函数消除植物（可以看作死亡）
Plant1::~Plant1();
Plant2::Plant2(int price);
Plant2::~Plant2();
//两个 不同类 的植物交换价格
void changePrice(Plant1& plant1, Plant2& plant2);
//获得所有植物的总价格，包括所有Plant1存活的实体和所有Plant2存活的实体
int getTotalPrice();
//获得Plant1的两个实体的总价格
int getTotalPrice(const Plant1& plant1, const Plant1& plant2);
//获得Plant1的所有实体的总价格
```

```
int getTotalPlant1Price();
```

注意：你可以为这两个类添加新的成员变量和成员函数等等，但**不可以修改已有的接口和类型。**另外测试用例1-8只会涉及以上提到的属性和需要实现的接口，其余测试用例会测试下面所讲到的内容。

`Plant2`类的植物虽然不会因为攻击而死亡，但它也有自己独特的功能。它有两个时间节点，`time1`和`time2`，该类植物从种植开始，生长`time1`天**之后**，每天便可以生产一株新的需要立马种植的**完全相同**的植物（可以看作每个新植物都会立马被种植），生长`time2`天**之后**，该植物就会死亡。例如某植物在第一天种植，time1=1，time2=2，那么会在第二天开始生产新植物，在第三天死亡（第三天不能生产新植物）。 对于`Plant2`，还需要实现下面几个函数：

```
Plant2::Plant2(int price,int time1,int time2); //需要完成的构造函数
//该植物类的功能，功能过程如上面所述。n表示该植物从种植开始经过的天数，n可能大于time2，具体功能可看样例4
//测试中该函数不会对同一个对象多次调用，可以理解为创建一个Plant2对象后，最多对该对象调用一次
//可以理解为植物为关卡道具，该函数调用即植物被种植，n为该植物被种植开始到关卡结束剩余的时间
Plant2::void func(int n);
//获得Plant2类的存活的实体的总数量，会通过此接口来验证func函数
int getTotalPlant2Num();
```

样例1：

输入：

```
Plant1 t(20, 3);
Plant1 t2(30, 4);
cout << getTotalPrice(t, t2);
```

输出：`50`

样例2：

输入：

```
Plant1 t(20, 3);
Plant2 t3(40);
cout << getTotalPrice();
```

输出：`60`

样例3：

```
Plant1 t(20, 3);
Plant2 t3(40);
changePrice(t, t3);
cout << getTotalPlant1Price();
```

输出：`40`

样例4：

```
Plant2 t(4, 1, 3); //目前已有存活的植物
tt.func(4); //植物t被种植，即开始发挥功能，距离关卡结束还有4天
cout << getTotalPlant2Num(); //关卡结束，输出目前存活的植物数量
```

输出：`6`

解释：

func函数调用的时候即为将植物t种植下去经过4天，该种植物的time1=1，time2=3.第一天：植物t生长，但未生出新植物第二天：植物t生长超过1天，生出新植物t2第三天：植物t生出植物t3，植物t2生长超过1天，生出新植物t4第四天：植物t生长超过3天，死亡。植物t2生出t5,植物t3生出t6，植物t4生出t7目前有6株植物存活。

样例5：

```
Plant2 t(4, 1, 3);
Plant2 t1(4, 1, 3);
t.func(4);t1.func(4);
cout << getTotalPlant2Num();
```

输出：`12` 解释：

代码表示我们本来有t和t1两株植物，并一起将它们种植经过4天的结果。植物的生长过程和样例4一样。测试用例中一起种植的植物需要求的生长天数也相同，也就是func的参数相同（如本题都是4）。

提示

合理地利用友元函数和类的静态成员将对此题有很大帮助。

Plant.h

```
#include<iostream>
using namespace std;
class Plant2;
class Plant1
{
private:
    int price; //该植物的价格，每个实体价格可能不一样
    int HP; //该植物的生命值，每个实体生命值可能不一样
    static int Price1;

public:

    void beAttacked(int x); //植物被x点伤害攻击，生命值应下降x

    Plant1(int price,int hp); //要完成的构造函数，另外也可以自行添加其他构造函数
    ~Plant1(); //析构函数
   friend void changePrice(Plant1& plant1, Plant2& plant2);
   friend int getTotalPrice(const Plant1& plant1, const Plant1& plant2);
   friend int getTotalPlant1Price();
    friend int getTotalPrice();
};
```

```
class Plant2
{
private:
    int price; //该植物的价格，每个实体价格可能不一样
    static int num;
    static int Price2;
    //=========

    int time1=0;
    int time2=0;

public:

    Plant2(int price); //要完成的构造函数，另外也可以自行添加其他构造函数
    ~Plant2(); //析构函数

    //=========
    friend void changePrice(Plant1& plant1, Plant2& plant2);
    friend int getTotalPrice();
    friend int getTotalPlant2Num();
    Plant2(int price,int time1,int time2); //需要完成的构造函数

    //该植物类的功能，n表示该植物从种植开始经过的天数，n可能大于time2，具体可看示例
    void func(int n);
};

//两个 不同类 的植物交换价格
void changePrice(Plant1& plant1, Plant2& plant2);

//获得所有植物的总价格，包括所有Plant1实体和所有Plant2实体
int getTotalPrice();

//获得Plant1的两个实体的总价格
int getTotalPrice(const Plant1& plant1, const Plant1& plant2);

//获得Plant1的所有实体的总价格
int getTotalPlant1Price();

//==========

//获得Plant2的总数量
int getTotalPlant2Num();
```

Plant.cpp

```
#include"Plant.h"
int Plant2::num=0;
int Plant1::Price1=0;
int Plant2::Price2=0;
void Plant1::beAttacked(int x)
{
    HP=HP-x;
    if(HP<=0){
        Plant1::Price1-=price;
    }
}
```

```
Plant1::Plant1(int price,int hp)
{
    Plant1::price=price;
    HP=hp;
    Plant1::Price1+=price;
}
Plant1::~Plant1()
{
    Plant1::Price1-=price;
}

Plant2::Plant2(int price)
{
    Plant2::price=price;
    Plant2::num++;
    Plant2::Price2+=price;
}
Plant2::~Plant2()
{
    Plant2::num--;
    Plant2::Price2-=price;
}

void changePrice(Plant1& plant1, Plant2& plant2)
{
    int temp=plant1.price;
    plant1.price=plant2.price;
    plant2.price=temp;
    Plant1::Price1=Plant1::Price1+plant1.price-plant2.price;
    Plant2::Price2=Plant2::Price2+plant2.price-plant1.price;
}

int getTotalPrice()
{
    return Plant2::Price2+Plant1::Price1;
}

int getTotalPrice(const Plant1& plant1, const Plant1& plant2)
{
    return plant1.price+plant2.price;
}

int getTotalPlant1Price()
{
    return Plant1::Price1;
}

//========
Plant2::Plant2(int price,int time1,int time2)
{
    Plant2::price=price;
    Plant2::time1=time1;
    Plant2::time2=time2;
    Plant2::num++;
    Plant2::Price2+=price;
}

void Plant2::func(int n)
```

```
{
    for(int i=1;i<=n;i++){
        if(i==1+Plant2::time2){
            Plant2::num--;
            return;
        }
        else if(i>=1+Plant2::time1){
            Plant2::num++;
            func(n-i+1);
        }

    }
}

int getTotalPlant2Num()
{
    return Plant2::num;
}
```

作业四编程题

第一关：String

任务描述

实现一个String类，需要实现以下函数以及重载以下操作符。 注：不可以使用STL库，尤其禁止直接使用string

```
class String
{
private:
    int len;    // 字符串长度
    char *str_p;    // 字符串数据
public:
    const char* getStr_p() { return str_p; }
    String();
    String(const char *s);
    String(const String &s);

    ~String();     //注意内存泄漏和野指针问题
    //（提示：释放空间后指针置为__？这是一个良好的编程习惯）
    void print();  //输出字符串，最后输出换行符\n

    //重载操作符=，[]
    String& operator=(const String &s);
    String& operator=(const char* s);
    char & operator[](int index);
    //TODO:+  字符串拼接 a+b=ab
    //TODO:==,!=,<  返回值为bool类型，按ASCII码值比较，约定 a<aa
};
```

请仔细查看样例以及注释中的要求. 除了写出来的接口，其余都只需要实现与String类型的重载，不需要实现与其他类型的（如char*）

样例：

```
String s0; //构造函数
s0.print();
//输出：
s0="abc";
s0.print();
//输出：abc
String s1(s0);  //构造函数
s1.print();
//输出：abc
s1.~String(); //析构函数，s1的析构不会影响s0
s0.print();
//输出：abc
String s2("d");  //构造函数
s2.print();
//输出：d
String s3 = s2;  //重载 =
s2.~String();  //析构函数，s2的析构不会影响s3
s3.print();
//输出：d
String s4 = s3 + s3 + s3;  //重载 +，加法会考察连续+，且加法不会影响加数
s4.print();
//输出：ddd
cout << s4[2] << endl;  //重载 []
//输出：d
cout << (s4 == s3) << endl;  //重载 ==
//输出0
cout << (s4 != s3) << endl;  //重载 !=
//输出1
cout << (s4 < s3) << endl;  //重载 <
//输出0
```

String.h

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

class String {
private:
    int len;    // 字符串长度
    char *str_p;    // 字符串数据

public:
    const char *getStr_p() { return str_p; }

    String();

    String(const char *s);

    String(const String &s);

    ~String();     //注意内存泄漏和野指针问题

    void print();  //输出字符串，最后输出换行符\n
```

```
    //重载操作符=，[]
    String &operator=(const String &s);

    String &operator=(const char *s);

    char &operator[](int index);

    String &operator+=(const String &s);

    friend String operator+(const String &s1,const String &s2);
    friend bool operator==(const String &s1,const String &s2);
    friend bool operator!=(const String &s1,const String &s2);
    friend bool operator<(const String &s1,const String &s2);


//TODO:+  字符串拼接 a+b=ab
//TODO:==,!=,<  返回值为bool类型，按ASCII码值比较，约定 a<aa
};

String operator+(const String &s1,const String &s2);
bool operator==(const String &s1,const String &s2);
bool operator!=(const String &s1,const String &s2);
bool operator<(const String &s1,const String &s2);
```

String.cpp

```
#include "String.h"
#include <cstring>

String::String(){
    str_p=new char[1];
    str_p[0]='\0';
    len=0;
}

String::String(const char *s) {
    str_p=new char[strlen(s)+1];
    len=strlen(s);
    strcpy(str_p,s);
}

String::String(const String &s) {
    str_p = new char[strlen(s.str_p) + 1];
    len = strlen(s.str_p);
    strcpy(str_p,s.str_p);
}

String::~String(){
    delete []str_p;
    str_p=NULL;
}

void String::print() {
    if(str_p==NULL){
        cout<<endl;
    }
    else{
```

```
        cout<<str_p<<endl;
    }
}

char &String::operator[](int index) {
    return str_p[index];
}

String &String::operator=(const char *s) {
    delete []str_p;
    str_p=new char[strlen(s)+1];
    strcpy(str_p,s);
    return *this;
}

String &String::operator=(const String &s) {
    strcpy(str_p,s.str_p);
    return *this;
}

String &String::operator+=(const String &s){
    strcat(str_p,s.str_p);
    return *this;
}

String operator+(const String &s1,const String &s2){
    String temp(s1);
    temp+=s2;
    return temp;
}

bool operator==(const String &s1,const String &s2){
    return strcmp(s1.str_p,s2.str_p)==0;

}

bool operator!=(const String &s1,const String &s2){
    return !strcmp(s1.str_p,s2.str_p)==0;
}

bool operator<(const String &s1,const String &s2){
    if(s1!=s2){
    return strcmp(s1.str_p,s2.str_p)<0;
    }
    else{
        return 0;
    }
}
```

第二关：Vector

考察知识点

操作符重载；动态数组

任务描述

Vector是一个动态数组，它有以下三个字段：

1. int *arr_: 存储元素的指针，初始值为nullptr
2. size_t sz_: 数组当前大小，初始应为0
3. size_t cap_: 数组当前容量，初始应为0

其中，sz表示*Vector*当前存储的元素个数，*cap*表示当前Vector的容量大小。容量大小是指Vector可以最多存储的元素数量，而不是当前存储的元素个数。当Vector存储的元素个数等于容量大小时，就需要扩容。

Vector中的push_back方法在插入新元素时，首先检查Vector的容量大小是否足够，如果容量不足，需要将容量扩大为原来的两倍。具体扩容的实现方式如下：

1. 如果arr_为nullptr，则分配初始容量1
2. 否则，分配当前容量的两倍大小
3. 将原数组中的所有元素拷贝到新数组中

请注意，测试代码中会检查你是否按照上述规则进行扩容，你需要通过capacity()返回正确的容量。
####样例

示例一

向空的Vector中添加一个元素

```
Vector v; // sz_ = 0, cap_ = 0, arr_ = nullptr
v.push_back(1); // sz_ = 1, cap_ = 1, arr_ = [1]
```

示例二

向已经满了的Vector中添加一个元素

```
Vector v(2, 0); // sz_ = 2, cap_ = 2, arr_ = [0, 0]
v.push_back(1); // sz_ = 3, cap_ = 4, arr_ = [0, 0, 1]
```

示例三

复制一个Vector

```
Vector v1(3, 2); // sz_ = 3, cap_ = 3, arr_ = [2, 2, 2]
Vector v2 = v1; // sz_ = 3, cap_ = 3, arr_ = [2, 2, 2]
v1[0] = 1; // v1: [1, 2, 2], v2: [2, 2, 2]
```

Vector.h

```
// Vector.h
// Note: The current file is Vector.h. Please implement each function in the
Vector.cpp!
// Note: The current file is Vector.h. Please implement each function in the
Vector.cpp!
// Note: The current file is Vector.h. Please implement each function in the
Vector.cpp!
```

```
#include <iostream>
/*
In this assignment, you will need to implement a simplified
version of std::vector, which supports some modifiers
*/
class Vector {
public:
  Vector();
  Vector(size_t count, int value = 0);
  Vector(const Vector &other);
  ~Vector();

  Vector &operator=(const Vector &other);

  int &operator[](size_t n);

  size_t size() const;
  size_t capacity() const;

  void push_back(const int &num);
  void pop_back();

private:
  int *arr_;
  size_t sz_; // the size of vector
  size_t cap_; // the capacity of vector
};
```

Vector.cpp

```
// Vector.cpp
#include "Vector.h"

Vector::Vector() {
    arr_=nullptr;
    sz_=0;
    cap_=0;
}

Vector::Vector(size_t count, int value) {
    sz_=count;
    cap_=count;
    arr_= new int[count];
    for(int i=0;i<count;i++){
        arr_[i]=value;
    }
}



    Vector::Vector(const Vector &other) {
    sz_=other.sz_;
    cap_=other.cap_;
    arr_=new int[cap_];
    for(int i=0;i<sz_;i++){
        arr_[i]=other.arr_[i];
    }
}
```

```
Vector::~Vector() {
    delete []arr_;
    arr_=nullptr;
}

Vector &Vector::operator=(const Vector &other) {
    if(&other==this){
        return *this;
    }
    sz_=other.sz_;
    cap_=other.cap_;
    delete []arr_;
    arr_=new int[other.cap_];
    for(int i=0;i<other.sz_;i++){
        arr_[i]=other.arr_[i];
    }
    return *this;
}


int &Vector::operator[](size_t n) {
    return arr_[n];
}

size_t Vector::size() const {
    return sz_;
}

size_t Vector::capacity() const {
    return cap_;
}

void Vector::push_back(const int &num) {
    if(sz_==cap_&&cap_==0){
        sz_=1;
        cap_=1;
        arr_=new int[1];
        arr_[0]=num;
    }
    else if(sz_==cap_){
        cap_*=2;
        int *arr=new int[cap_];
        for(int i=0;i<sz_;i++){
            arr[i]=arr_[i];
        }
        delete []arr_;
        arr_=new int[cap_];
        for(int i=0;i<sz_;i++){
            arr_[i]=arr[i];
        }
        delete []arr;
        arr_[sz_]=num;
        sz_++;
    }
    else{
        arr_[sz_]=num;
        sz_++;
```

```
    }
}

void Vector::pop_back() {
    sz_--;
}
```

第三关：List

考察知识点

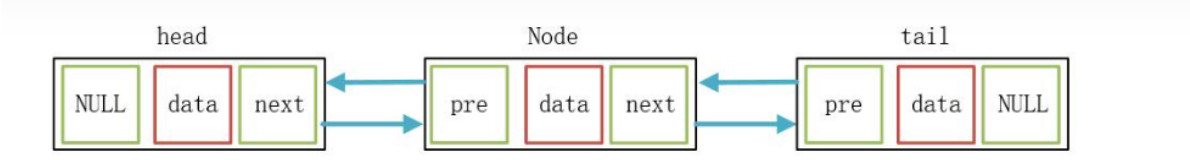
操作符重载；双向链表

任务描述

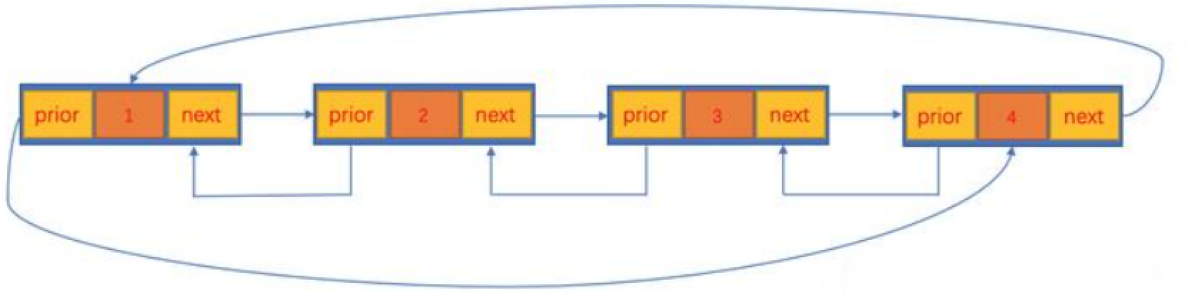
双向链表是一种链表结构，和普通的单向链表不同，它的每个链表节点不仅包含`next`指针，还包含一个`back`指针指向前一个节点，如下所示：

```
struct Element
{
    int num;
    // 可以添加任何你需要的数据类型,但与本关测评无关
    bool operator==(const Element e) const;
};
class ListNode
{
    Element e;
    ListNode *back = nullptr;
    ListNode *next = nullptr;
    ListNode();
    ListNode(const Element &e_);
    friend class List; // 为便于测评，这里设置为友元
};
```

普通的双向链表如下图所示



双向循环链表则是首尾相接的双向链表。



List是一个双向循环链表，它的定义如下：

```
 class List
{
    ListNode head; // 加入默认头节点,使得代码逻辑更统一
```

```
                    // 在进行链表操作时,你需要对head取地址
    size_t size_;  // 可以自行使用,不做要求
public:
    // 构造和析构
    List();                             // 默认构造函数,head的next和back应该指向自己
    List(const List &l_);               // 拷贝构造函数,内部深拷贝
    ~List();                            // 析构函数,要求释放空间,析构后的head和size变量不做
要求
    void operator=(const List &l_); // 赋值操作符,内部深拷贝
    size_t size() { return size_; };
    // 访问和更改,返回访问或更改是否成功
    bool push_front(const Element &e); // 在头节点的next处插入e,分配空间失败时返回
false
    bool push_back(const Element &e);
    bool pop_front(); // 弹出头节点的next节点并释放,不可弹出时返回false
    bool pop_back();
    bool remove(const Element &e);                // 删除链表中所有值为e的节点并释放,成
功删除返回true
    bool insert(const Element &e, ListNode *ln); // 在节点ln后插入e,分配空间失败时返回
false,测试用例中给定的ln均合法
    void erase(ListNode *ln);                     // 从链表中删除节点ln,释放ln的空间,
测试用例中给定的ln均合法
    ListNode *operator[](size_t i); // 获取正向第i个节点(从0开始,不包括head)的指针,不存
在则返回空
    // 遍历输出
    void print(); // 正向(next)输出链表中的元素num,不包括head,空格分隔,换行符结尾;空则输
出换行符
};
```

你需要仔细阅读上述定义代码，按照注释要求在`List.cpp`文件中实现对应的接口函数。

值得注意的是，stl库中使用迭代器来遍历list容器，本题目中为了简单起见，使用下标取地址的方式来获取每个`ListNode`节点。

此外，本题目中List的成员变量head节点是一个对象，所以当你需要比较head节点和指针时，要对head节点进行取地址。

编程注意事项和提示

1. new 和 delete

为了检验你是否正确分配和释放空间，`ListNode`类重载了new和delete这两个函数来记录内存信息。所以请你使用这两个函数来进行`ListNode`节点的创建和释放，**否则将不保证测评是否正确**，例如:

```
 ListNode *ln = new ListNode(e);
delete ln;
```

请不要使用带方括号的new和delete。

1. 编程技巧

你可能注意到了，题目中的接口有些十分相似，有些接口则可以用来实现另一些接口。所以，你可能需要灵活决定接口实现顺序，或者使用接口来实现接口，从而减轻编程压力。

样例

鉴于链表相关的题目debug难度，本关公开所有测试样例，希望同学们认真完成。

```
/**
 * 测试push_front push_back
 * 注意:同样需要保证构造函数、析构函数和print正确实现
 */
void test1()
{
    List l;
    l.push_front({2});
    l.push_back({3});
    l.push_front({1});
    l.push_back({4});
    l.print();
}
/** 输出
 * 1 2 3 4
 */
/**
 * 测试pop_front pop_back
 */
void test2()
{
    List l;
    l.push_back({1});
    l.push_back({2});
    l.pop_front();
    l.print();
    l.push_back({4});
    l.push_front({3});
    l.pop_back();
    l.print();
}
/** 输出
 * 2
 * 3 2
 */
/**
 * 测试pop_front pop_back
 */
void test3()
{
    List l;
    l.push_back({7890});
    l.pop_back();
    l.pop_front();
    l.push_front({2});
    l.push_back({3});
    l.push_front({1});
    l.push_back({4});
    l.print();
    l.pop_front();
    l.pop_back();
    l.print();
    l.pop_back();
```

```
    l.pop_front();
    l.print();
}
/** 输出
 * 1 2 3 4
 * 2 3
 *
 */
/**
 * 测试remove
 */
void test4()
{
    List l;
    l.push_front({1});
    l.push_front({2});
    l.push_front({3});
    l.push_front({1});
    l.push_front({3});
    l.push_front({1});
    l.print();
    l.remove({1});
    l.print();
    l.remove({2});
    l.print();
}
/** 输出
 * 1 3 1 3 2 1
 * 3 3 2
 * 3 3
 */
/**
 * 测试insert和earse
 */
void test5()
{
    List l;
    l.push_back({1});
    l.push_back({2});
    l.push_back({3});
    l.push_back({4});
    l.push_back({5});
    l.push_back({6});
    l.print();
    l.insert({23},l[1]);
    l.insert({56},l[5]);
    l.print();
    l.erase(l[5]);
    l.erase(l[1]);
    l.print();
}
/** 输出
 * 1 2 3 4 5 6
 * 1 2 23 3 4 5 56 6
 * 1 23 3 4 56 6
 */
/**
 * 测试拷贝构造函数和赋值操作符
```

```
 */
void test6()
{
    List l;
    l.push_back({1});
    l.push_back({2});
    l.push_back({3});
    l.push_back({4});
    l.push_back({5});
    l.push_back({6});
    List l_cp=l;
    l.remove({4});
    l.remove({2});
    List l_assign;
    l_assign=l;
    l_cp.print();
    l_assign.print();
}
/** 输出
 * 1 2 3 4 5 6
 * 1 3 5 6
 */
```

List.h

```
#include <iostream>
#include "ListNode.h"

class List
{
    ListNode head; // 加入默认头节点,使得代码逻辑更统一
                   // 在进行链表操作时,你需要对head取地址
    size_t size_;  // 可以自行使用,不做要求
public:
    // 构造和析构
    List();                             // 默认构造函数,head的next和back应该指向自己
    List(const List &l_);               // 拷贝构造函数,内部深拷贝
    ~List();                            // 析构函数,要求释放空间,析构后的head和size变量不做要求
    void operator=(const List &l_); // 赋值操作符,内部深拷贝

    size_t size() { return size_; };

    // 访问和更改,返回访问或更改是否成功
    bool push_front(const Element &e); // 在头节点的next处插入e,分配空间失败时返回false
    bool push_back(const Element &e);
    bool pop_front(); // 弹出头节点的next节点并释放,不可弹出时返回false
    bool pop_back();
    bool remove(const Element &e);                // 删除链表中所有值为e的节点并释放,成功删除返回true
    bool insert(const Element &e, ListNode *ln); // 在节点ln后插入e,分配空间失败时返回false,测试用例中给定的ln均合法
    void erase(ListNode *ln);                     // 从链表中删除节点ln,释放ln的空间,测试用例中给定的ln均合法
```

```
    ListNode *operator[](size_t i); // 获取正向第i个节点(从0开始,不包括head)的指针,不存在则返回空

    // 遍历输出
    void print(); // 正向(next)输出链表中的元素num,不包括head,空格分隔,换行符结尾;空则输出换行符
};
```

List.cpp

```
#include <iostream>
#include "List.h"

List::List()
{
    head.next = &head;
    head.back = &head;
    size_=0;
}

List::List(const List &l_)
{
     head.next = &head;
    head.back = &head;
    size_=0;
   ListNode *p=l_.head.next;
    while(p!=&l_.head){
        push_back(p->e);
        p=p->next;
    }
}

List::~List()
{
    while(size_>0){
        pop_back();
    }
}

void List::operator=(const List &l_)
{
   if(this!=&l_){
        while(size_>0){
            pop_front();
        }
        ListNode *p=l_.head.next;
    while(p!=&l_.head){
        push_back(p->e);
        p=p->next;
    }
        size_=l_.size_;

    }

}

bool List::push_front(const Element &e)
```

```
{
    ListNode *ln = new ListNode(e);
    if(!ln){
        return false;
    }
    ln->next=head.next;
    head.next->back=ln;
    ln->back=&head;
    head.next=ln;
    size_++;
    return true;
}

bool List::push_back(const Element &e)
{
    ListNode *ln = new ListNode(e);
    if(!ln){
        return false;
    }
    ln->back=head.back;
    head.back->next=ln;
    ln->next=&head;
    head.back=ln;
    size_++;
    return true;
}

bool List::pop_front()
{
    if(size_==0){
        return false;
    }
    ListNode *p=head.next;
    p->next->back=&head;
    head.next=p->next;
    size_--;
    delete p;
    return true;
}

bool List::pop_back()
{
    if(size_==0){
        return false;
    }
    ListNode *p=head.back;
    p->back->next=&head;
    head.back=p->back;
    size_--;
    delete p;
    return true;
}

bool List::remove(const Element &e)
{
    ListNode *rem=head.next;
    while(rem!=&head){
        if(rem->e==e){
```

```
            ListNode *temp=rem;
            rem->back->next=rem->next;
            rem->next->back=rem->back;
            rem=rem->next;
            delete temp;
            size_--;
        }
        else {
            rem = rem->next;
        }
    }

    return true;
}

bool List::insert(const Element &e, ListNode *ln)
{
    ListNode *p = new ListNode(e);
    if(!p){
        return false;
    }
    p->next=ln->next;
    p->back=ln->back;
    ln->next->back=p;
    ln->next=p;
    size_++;
    return true;
}

void List::erase(ListNode *ln)
{
    ln->next->back=ln->back;
    ln->back->next=ln->next;
    delete ln;
    size_--;
}

ListNode *List::operator[](size_t i)
{
    if(i>=size_) return nullptr;
    ListNode *p=head.next;
    for(int j=0;j<i;j++){
        p=p->next;
    }
    return p;
}

void List::print()
{
    ListNode *p=head.next;
    while(p!=&head){
        std::cout<<p->e.num<<' ';
        p=p->next;
    }
    std::cout<<std::endl;
}
```

第四关：Map

任务描述

本关任务：实现一个指定键（Key）和值（Value）类型的Map。

相关知识

为了完成本关任务，你需要掌握：1.什么是Map，2.如何实现Map。

什么是Map

Map是一种关联式容器，一般称为“映射”、“映射表”。Map存储的元素是一个个形如`<Key, Value>`的键值对。**Map为使用者提供了根据Key访问Value的基本功能。**需要注意的是，**Map中存储的Key不允许重复**，一个Key至多对应一个Value。

举一个形象的例子：假设有一份成绩单，表上有两列数据，第一列数据是学号，第二列数据是成绩。你可以用Map记录这份成绩单：

```
学号（Key）     成绩（Value）
202200001      99
202200002      98
202200003      100
202200004      96
...
202200013      100
...
MG2022005      59
MF2022006      60
```

根据常识，学生的学号唯一，不重复，但学生的成绩可以相同，因此，以学号为Key，以成绩为值，使用Map记录这份名单是合理的。 有了这份Map，我们可以根据学号，查询学生的成绩：例如，`map[202200004]`的值为`96`，`map[MF2022006]`的值为`60`，而`map[202200003]`和`map[202200013]`的值都为`100`。

C++ STL中，map键值对的键和值可以是任意数据类型，包括C++基本数据类型（int、double 等）、使用结构体或类自定义的类型。**但这一关只要求你实现固定键值类型的Map：键的类型为string，值的类型为int。**

下面将给出详细的Map接口描述：

```
class Map {
private:
    /*
     * 数据成员
     * 不做规定，由你自己实现
     */
public:
    Map(); // 构造函数，构造一个空映射表，初始化数据成员
   ~Map(); // 析构函数，释放申请的内存空间
   bool insert(const string& key, int value); // 插入键值到映射表中
   // 成功插入返回true；若对应键已存在旧值，插入失败，返回false

   bool erase(const string &key); // 删除该键的键值对
   // 成功删除返回true；若不存在对应键的键值对，删除失败，返回false
```

```
    bool find(const string &key); // 返回映射表中是否含有对应键的键值对


     int & operator[](const string &key); // 数组下标访问操作符重载
    // 可以通过下标访问的方式直接用键得到值的引用
    // 需要注意的是，如果用下标访问的方式访问了不存在的key，你需要为该key创建一个键值对
    // 此法创建的值初始为0，你需要返回该值的引用\

    int size() const; // 返回映射表中当前记录的键值对总数
};
```

Map的使用（测试）示例和输出如下：

```
Map id2grade;
// 插入两个键值对
id2grade.insert("202200001", 99);
id2grade.insert("202200002", 98);
std::cout << id2grade.size() << std::endl; // 2
// 删除一个已经存在的键："202200002"后查找不到改键；删除一个不存在的键；
std::cout << id2grade.erase("202200002") << std::endl; // 1(true)
std::cout << id2grade.find("202200002") << std::endl; // 0(false)
std::cout << id2grade.erase("202200000") << std::endl; // 0(false)
std::cout << id2grade.size() << std::endl; // 1
// 用下标访问方式查询键对应的值，包含已存在的键值对和不存在的键值对
// 注意，用下标访问的方式访问了不存在的key，你需要为该key创建一个键值对，值的初始值为0
std::cout << id2grade["202200001"] << std::endl; // 99
std::cout << id2grade["202200002"] << std::endl; // 0
std::cout << id2grade.find("202200002") << std::endl; // 1(true)
std::cout << id2grade.size() << std::endl; // 2
// 用insert插入键值对，但该键已存在，插入失败，原有的键值对不变
std::cout << id2grade.insert("202200002", 100) << std::endl; // 0(false)
std::cout << id2grade["202200002"] << std::endl; // 0
std::cout << id2grade.size() << std::endl; // 2
// 用下标访问方式修改键值对
id2grade["202200002"] = 95;
id2grade["DZ00001"] = 100;
std::cout << id2grade["202200002"] << std::endl; // 95
std::cout << id2grade["DZ00001"] << std::endl; // 100
std::cout << id2grade.size() << std::endl; // 3
```

注意，Map是关联式容器而不是序列式容器，因此我们不会考察Map内元素遍历的输出顺序。

如何实现Map

Map的实现可以很简单，也可以很复杂。 你可以用简单的实现轻松得到本关全部的OJ分数，但我们也鼓励你尝试复杂的实现锻炼自己的编程能力。

简单的实现

定义一个结构体/类，表示键值对的类型：

```
struct MapNode {
    string k; // key
    int v; // value
};
```

然后，用你擅长的数组/链表：在Map中维护一个MapNode类型的动态数组或链表。在增删查改Map时遍历该数组/链表，依次判断每个数组/链表元素的键是否符合对应要求。 提示：你应该知道，数组对删除元素的操作并不友好 :(。

MyMap.h

```
// MyMap.h
// 注意：当前文件为MyMap.h，请在MyMap.cpp中实现各成员函数！
// 注意：当前文件为MyMap.h，请在MyMap.cpp中实现各成员函数！
// 注意：当前文件为MyMap.h，请在MyMap.cpp中实现各成员函数！
#include<string>
// #include<vector>
// #include<list>
using namespace std;
/*
如果你要实现开散列哈希映射，你可以使用这个哈希函数，也可以自行设计哈希函数
inline unsigned hashFunc(const string & s)
{
    unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
    unsigned int hash = 0;

    for (int i = 0; i < s.size(); ++i)
    {
        hash = hash * seed + (s[i]);
    }

    return (hash & 0x7FFFFFFF);
}
*/

class Map {
private:
     int size_;
    struct Node
    {
        string k; // key
        int v; // value
        Node *next;
    } *head; // 链表的头节点
public:
    Map(); // 构造函数，构造一个空映射表，初始化数据成员
   ~Map(); // 析构函数，释放申请的内存空间

   bool insert(const string& key, int value); // 插入键值到映射表中
   // 成功插入返回true；若对应键已存在旧值，插入失败，返回false

   bool erase(const string &key); // 删除该键的键值对
   // 成功删除返回true；若不存在对应键的键值对，删除失败，返回false

   bool find(const string &key); // 返回映射表中是否含有对应键的键值对

   int & operator[](const string &key); // 数组下标访问操作符重载
   // 可以通过下标访问的方式直接用键得到值的引用
   // 需要注意的是，如果用下标访问的方式访问了不存在的key，你需要为该key创建一个键值对
   // 此法创建的值初始为0，你需要返回该值的引用

   int size() const; // 返回映射表中当前记录的键值对总数
```

```
};
```

MyMap.cpp

```
#include"MyMap.h"

Map::Map() {
    size_=0;
    head=nullptr;
}

Map::~Map() {
    Node *p=head;
    while(p->next!=nullptr){
        Node*q=p;
        p=p->next;
        delete q;
    }
    head=nullptr;
    size_=0;
}

bool Map::insert(const string& key, int value) {
    if(head==nullptr){
        Node*p=new Node;
        head=p;
        head->k=key;
        head->v=value;
        head->next=nullptr;
        size_++;
        return true;
    }
    Node *p=head;
    while(p->next!=nullptr){
        if(key==p->k){
            return false;
        }
        else{
            p=p->next;
        }
    }
    if(key==p->k){
        return false;
    }
    Node *q=new Node;
    q->k=key;
    q->v=value;
    p->next=q;
    q->next=nullptr;
    size_++;
    return true;
}

bool Map::find(const string& key) {
    Node *p=head;
    while(p->next!=nullptr){
        if(p->k==key){
```

```
            return true;
        }
        else{
            p=p->next;
        }
    }
    if(p->k==key){
        return true;
    }
    else{
    return false;
    }
}

bool Map::erase(const string& key) {
    if(head==nullptr){
        return false;
    }
    else if(head->next==nullptr){
        if(head->k==key){
            head=nullptr;
            size_--;
            return true;
        }
        else{
            return false;
        }
    }
    else if(head->k==key){
        Node*p=head;
        head=head->next;
        delete p;
        size_--;
        return true;
    }
    Node *p=head->next;
    Node *q=head;
    q->next=p;
    while(p->next!=nullptr){
        if(p->k==key){
            q->next=p->next;
            p=nullptr;
            size_--;
            return true;
        }
        else{
            p=p->next;
            q->next=p;
        }
    }
    if(p->k==key){
        q->next=nullptr;
        delete p;
        size_--;
        return true;
    }
    return false;
}
```

```
int & Map::operator[](const string &key) {
    if(head==nullptr){
        Node *q=new Node;
        head=q;
        head->k=key;
        head->v=0;
        head->next=nullptr;
        size_++;
        return head->v;
    }
    Node *p=head;
    while(p->next!=nullptr){
        if(p->k==key){
            return p->v;
        }
        p=p->next;
    }
    if(p->k==key){
        return p->v;
    }
    Node *q=new Node;
    p->next=q;
    q->next=nullptr;
    q->k=key;
    q->v=0;
    size_++;
    return q->v;
}

int Map::size() const {
    return size_;
}
```