**《空间数据结构基础》**

**课程设计报告（地信19级）**

**姓名 武成龙**

**班级 地信19-1班**

**学号 07192329**

**环境与测绘学院**

1. **C++面向对象程序设计基础**

**【实验简介】**学会用算法语言C++描述抽象数据类型，使用模板建立数据结构。理解数据结构的组成分为两部分，第一部分是数据集（数据元素），第二部分是在此数据集上的操作。从面向对象的观点看，这两部分代表了对象的属性和方法。掌握用C++描述数据结构的基本方法，即通过建立类来描述抽象数据类型。类的数据成员提供对象属性，成员函数提供操作方法，方法是公共接口，用户通过调用方法实现对属性的访问。

**【实验内容】**

1. 定义三维空间的坐标点TPoint
2. 描述三维空间的球TBall，实现其主要操作（如计算体积和表面积，输出空间坐标等）。

**【主要代码】**

头文件：

#pragma once

#ifndef point\_h

#define point\_h

#include <iostream>

#include <math.h>

using namespace std;

const double pi = 3.1415926;

class TPoint

{

protected:

//定义三维坐标

double x;

double y;

double z;

public:

TPoint() { x = 0; y = 0; z = 0; }//缺省参数的构造函数

TPoint(double px, double py, double pz) { x = px; y = py; z = pz; }//有默认参数的构造函数

void move(double px, double py, double pz) { x = px; y = py; z = pz; }//修改坐标

void show() { cout << "点的X坐标：" << x << "\t" << "Y坐标：" << y << "\t" << "Z坐标：" << z << "\n"; }//显示坐标

void get\_p(double &px, double &py, double &pz) { px = x; py = y; pz = z; }

};//获得坐标

class TBall:public TPoint

{

private:

double radius;//定义半径

public:

TBall() { radius = 0; x = 0; y = 0; }//缺省构造函数

TBall(double px, double py, double pz,double r) {x = px;y = py; z = pz;radius = r;}//带参数的构造函数

void show();//输出球心，半径,表面积，体积

void move(double px, double py, double pz) { x = px; y = py; z = pz; }//修改球心坐标

void set\_r(double nr) { radius = nr; }//修改半径

friend void p\_b(TPoint p, TBall b);//判断点与球的关系（在球内，在球面上，在球外）

};

void TBall::show()

{

cout << "球心的X坐标：" << x << "\t" << "Y坐标：" << y << "\t" << "Z坐标：" << z << "\t" << "半径：" << radius << "\t";

cout << "球表面积为：" << 4 \* pi \* pow(radius, 2) << "\t" << "球体积为：" << 4 \* pi \* pow(radius, 3) / 3 << "\n";

}

//友元函数

void p\_b(TPoint p, TBall b)

{

double px = 0, py = 0, pz = 0;

p.get\_p(px, py, pz);//获得点坐标

double flag = 0;

flag = sqrt(pow(b.x-px,2)+ pow(b.y - py, 2)+ pow(b.z - pz, 2)) - b.radius;

if (flag > 0)

{

cout << "点在球外";

}

else if(flag == 0)

{

cout << "点在球面上";

}

else if (flag < 0)

{

cout << "点在球内";

}

}

#endif point\_h

源文件：

#include <iostream>

#include "point.h"

int main()

{

TPoint A(1, 2, 3);

TBall B(2, 3, 4, 5);

A.show();

B.show();

p\_b(A, B);

cout << endl;

B.move(10, 11, 12);

B.show();

p\_b(A, B);

cout << endl;

A.move(5, 11, 12);

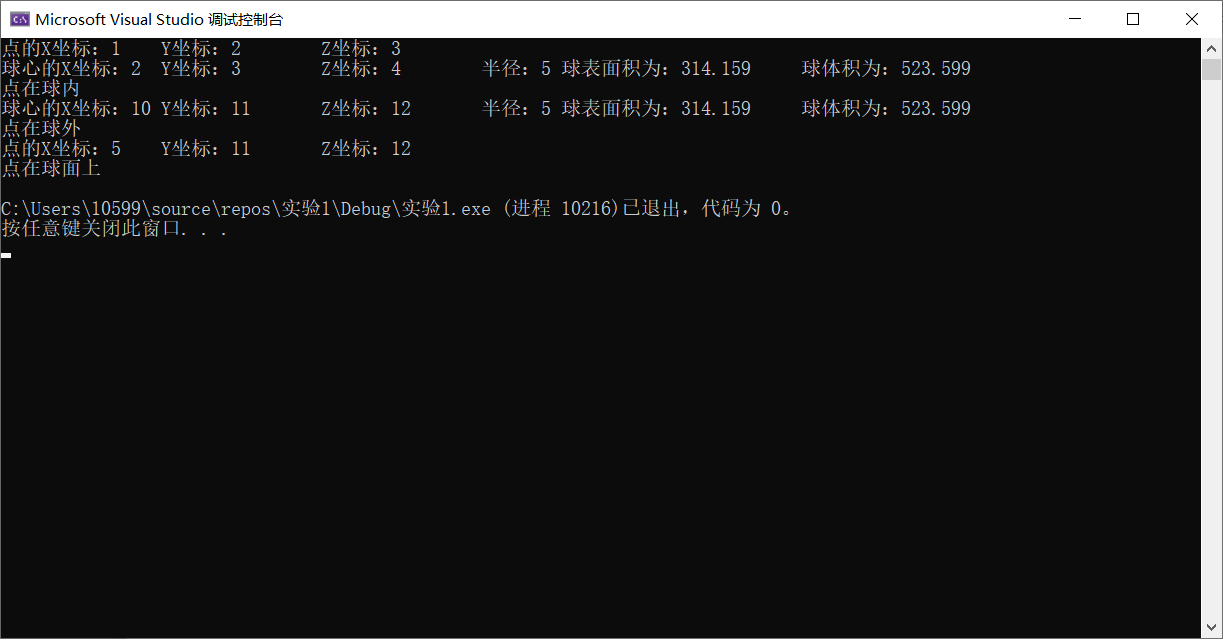
A.show();

p\_b(A, B);

cout << endl;

}

**【实验过程】**



**【实验体会】**

先创建一个三维点类TPoint，然后公有继承创建球类TBall。要注意的是。为了子类能访问父类的坐标，需要把TPoint的坐标设为protected。我还创建了一个借助数学函数（math.h）判断点与球的关系（在球内，在球面上，在球外）的友元函数p\_b。由于TPoint的对象不能访问坐标（保护成员），我增加了一个获取坐标的成员函数get\_p，使用引用传递的方式获取坐标。

**2 通讯簿管理（顺序表的应用）**

**【问题描述】**通讯簿是一个线性表，可以存储一定数量的联系人记录，提供查找、插入、删除和修改等操作。通讯簿的特点是以查找为主要操作，要求快速查找到指定对象的位置，故宜采用具有随机访问功能的顺序表。

**【数据结构】**使用顺序表SeqList建立通讯簿。作为表项的联系人记录，至少应包括以下属性：

{ 序号，姓名，与本人关系，电话号码}

其中序号具有唯一性。序号和姓名可作为查找的主要关键字。与本人关系可枚举为“亲人”、“朋友”和“同事”，主要作用是为联系人分组，并作为次要关键字。

将上述联系人记录定义为一个结构（struct），在主程序中建立模板类顺序表SeqList的对象时用该结构实例化表结点的类型。

**【主要功能】**程序应为用户提供操作选择界面，必要的操作包括：查找某人电话号码，添加新记录，修改记录，删除记录，打印亲人清单、朋友清单或同事清单以及退出等。另外，为初始化方便，原始数据可存储在磁盘文件中。

**【主要代码】**

头文件seq\_list.h是之前写过的顺序表头文件，不再列出代码。

头文件head.h：

#pragma once

#ifndef head\_H

#define head\_H

#include<string>

#include<fstream>

#include "seq\_list.h"

enum rela

{ 其他 = 0, 亲人, 朋友, 同事 };

string getz(rela r1) //将枚举类转化为文字输出

{

switch (r1)

{

case 1:return (string)"亲人";

case 2:return (string)"朋友";

case 3:return (string)"同事";

}

}

struct person

{

string number;

string name;

rela r;

string tel;

person() //默认构造函数

{

number = '\0'; name = "\0"; r = 其他; tel = "\0";

}

person(string number1, string name1, rela r1, string tel1)

{

number = number1; name = name1; r = r1; tel = tel1;

}

void printp()

{

cout << endl << "当前联系人为：" << endl;

cout << "编号： " << number << endl;

cout << "姓名： " << name << endl;

cout << "分组：" << getz(r) << endl;

cout << "电话号码： " << tel << endl;

}

};

//读取初始通讯录

bool read(seq\_list<person>& temp)

{

ifstream openp; //打开文件读取初始通讯录

openp.open("通讯录.txt");

if (!openp.is\_open()) {

cout << "未成功打开文件" << endl;

return false;

}

string n1; string name1; rela r1; string tel1; //用于存储读取的临时数据

for (int i = 0; i < 6; i++) //文件没结束

{

openp >> n1 >> name1 >> (int&)r1 >> tel1; //用于写入到线性表中

person tempper(n1, name1, r1, tel1); //用于写入到线性表中

bool flag = temp.insert(temp.length(), tempper);

if (flag == false) //插入不成功

cout << "插入错误" << endl;

}

openp.close();

return true;

}

//输出全部通讯录

void print(seq\_list<person>& temp)

{

person tempp;//临时存放待输出联系人

for (int i = 0; i < temp.length() ; i++)

{

temp.get\_data(i + 1, tempp);

tempp.printp();

}

}

//分组输出

void printz(seq\_list<person>& temp)

{

person tempp;//临时存放待输出联系人

rela r1;

do

{

cout << "请输入要查找的组别：" << endl;

cout << "1. 亲人； 2. 朋友； 3. 同事：" << endl;

cin >> (int&)r1;

if (r1 != 亲人 && r1 != 朋友 && r1 != 同事)

{

cout << "请输入正确的分组！" << endl << endl; continue;

}

for (int i = 0; i < temp.length() ; i++) //从头遍历，寻找分组满足条件的元素

{

temp.get\_data(i + 1, tempp);

if (tempp.r == r1)

tempp.printp();

}

break;//分组打印结束，退出

} while (true);

}

//添加联系人

void addper(seq\_list<person>& temp)

{

string n1; string name1; rela r1; string tel1; //用于写入新数据

cout << "请输入想要添加的联系人信息" << endl;

cout << "请输入新联系人编号：" << endl;

person tempp;//临时存放每次读取到的联系人

do

{

cin >> n1;

int flag = 0; //判定是否与已有编号重复

for (int i = 0; i < temp.length() ; i++)

{

temp.get\_data(i + 1, tempp);

if (tempp.number == n1)

{

cout << "编号已经存在，请输入一个新编号" << endl;

flag = 1;

break;

}

}

if (flag == 0) //flag没被置1表示没有发现重复项

break;

} while (true);

cout << "请输入新联系人的姓名：" << endl;

cin >> name1;

cout << "请输入新联系人的分组，1. 亲人； 2. 朋友； 3. 同事：" << endl;

cin >> (int&)r1;

cout << "请输入新联系人的电话号码：" << endl;

cin >> tel1;

person tempper1(n1, name1, r1, tel1); //要插入的联系人

bool insert1 = temp.insert(temp.length() , tempper1); //插入

if (insert1)

cout << "新建联系人成功！" << endl;

}

//通过编号获得联系人是第几个元素

int numberfind(seq\_list<person>& temp, int n) //n表示在不同场景中调用，控制提示文字的改变

{

person tempp;//临时存放待输出联系人

string fi;//想要查找的编号

cout << "请输入想要";

switch (n)

{

case 1:cout << "查找"; break;

case 2:cout << "删除"; break;

case 3:cout << "修改"; break;

}

cout << "的联系人编号：" << endl;

cin >> fi;

for (int i = 0; i < temp.length() ; i++) //从头遍历，寻找分组满足条件的元素

{

temp.get\_data(i + 1, tempp);

if (tempp.number == fi)

return i;

}

return -1;

}

//通过姓名获得联系人是第几个元素

int namefind(seq\_list<person>& temp, int n) //n表示在不同场景中调用，控制提示文字的改变

{

person tempp;//临时存放待输出联系人

string fstring;//想要查找的姓名

cout << "请输入想要";

switch (n)

{

case 1:cout << "查找"; break;

case 2:cout << "删除"; break;

case 3:cout << "修改"; break;

}

cout << "的联系人姓名：" << endl;

cin >> fstring;

for (int i = 0; i < temp.length() ; i++) //从头遍历，寻找分组满足条件的元素

{

temp.get\_data(i + 1 , tempp);

if (tempp.name == fstring)

return i;

}

return -1;

}

//查找联系人

void find(seq\_list<person>& temp)

{

char m; //用于判定查找方式

int result; //查找结果

person tempper;//若找到，用于存储找到的联系人

do

{

cout << "请输入想通过哪种方式查找" << endl;

cout << "1. 通过编号查找 2. 通过姓名查找" << endl;

cin >> m;

if (m == '1')

result = numberfind(temp, 1); //选择了编号查找

else if (m == '2')

result = namefind(temp, 1); //选择了姓名查找

else //选项错误，重新输入选项

{

cout << "请输入正确的方式！" << endl;

continue;

}

if (result == -1)

{

cout << "没有找到该联系人" << endl;

break; //没找到，结束查找

}

else

{

cout << "找到了该联系人" << endl;

temp.get\_data(result, tempper);

tempper.printp();

break; //找到了，结束查找

}

} while (true);

}

//删除联系人

void deleteper(seq\_list<person>& temp)

{

//先查找是否有要删除的联系人

int result;//用于表示查找到要删除的联系人与否

char m;//用于表示用哪种方式修改

do

{

cout << "请输入想通过哪种方式删除" << endl;

cout << "1. 通过编号删除 2. 通过姓名删除" << endl;

cin >> m;

if (m == '1')

result = numberfind(temp, 2); //选择了编号删除

else if (m == '2')

result = namefind(temp, 2); //选择了姓名删除

else //选项错误，重新输入选项

{

cout << "请输入正确的方式！" << endl;

continue;

}

if (result == -1)

{

cout << "无该联系人" << endl;

break; //想删除的联系人不存在

}

else

{

person tempper;

temp.remove(result, tempper);

cout << "删除成功！" << endl;

break; //找到了，删除

}

} while (true);

}

//修改联系人

void reset(seq\_list<person>& temp)

{

//先查找是否有要修改的联系人

int result;//用于表示查找到要修改的联系人与否

char m;//用于表示用哪种方式修改

do

{

cout << "请输入想通过哪种方式修改" << endl;

cout << "1. 通过编号修改 2. 通过姓名修改" << endl;

cin >> m;

if (m == '1')

result = numberfind(temp, 3); //选择了编号修改

else if (m == '2')

result = namefind(temp, 3); //选择了姓名修改

else //选项错误，重新输入选项

{

cout << "请输入正确的方式！" << endl;

continue;

}

if (result == -1)

{

cout << "无该联系人" << endl;

break; //想修改的联系人不存在

}

else

{

person tempper1; //存放老数据（编号与姓名）

temp.get\_data(result, tempper1);

person tempper2; //新联系人

tempper2.number = tempper1.number;//编号不变

tempper2.name = tempper1.name;//姓名不变

rela r1; //新分组

string tel1; //新号码

cout << "请输入新分组,，1. 亲人； 2. 朋友； 3. 同事：" << endl;

cin >> (int&)r1;

cout << "请输入新号码" << endl;

cin >> tel1;

tempper2.r = r1;

tempper2.tel = tel1;

temp.set\_data(result, tempper2);

cout << "修改成功！" << endl;

break; //找到了，修改

}

} while (true);

}

//开始运行

void start(seq\_list<person>& temp)

{

char m;//用于选择进行什么操作

char flag = '1';//用于是否退出

do

{

do //选择操作

{

cout << "请输入需要的操作： " << endl;

cout << "1： 查看全部联系人" << endl;

cout << "2： 分组查看联系人" << endl;

cout << "3： 插入新联系人" << endl;

cout << "4： 查找联系人" << endl;

cout << "5： 删除联系人" << endl;

cout << "6： 修改联系人信息" << endl;

cout << endl << " 退出请按 0" << endl;

cin >> m;

if (m >= '0' && m <= '6')

break;

else

cout << "请输入正确的操作！" << endl;

} while (true);

switch (m) //根据选择的操作执行动作

{

case '1': print(temp); break;

case '2': printz(temp); break;

case '3': addper(temp); break;

case '4': find(temp); break;

case '5': deleteper(temp); break;

case '6': reset(temp); break;

default: flag = '0';

}

if (flag == '0')//若上一级菜单已经选择退出，不必再次询问

break;

do

{ //是否继续执行下一操作

cout << endl << "请问是否继续？ " << endl << "0. 退出 1. 继续 " << endl;

cin >> flag;

if (flag == '1' || flag == '0')

break;

else

{

cout << "请输入正确的操作！" << endl;

}

} while (true);

if (flag == '0') //退出

break;

} while (true);

}

#endif head\_H

源文件：

#include <iostream>

#include "head.h"

int main()

{

seq\_list<person> address\_list;//创建顺序表

if (!read(address\_list))

{

cout << endl << "读取文件失败，退出程序！";

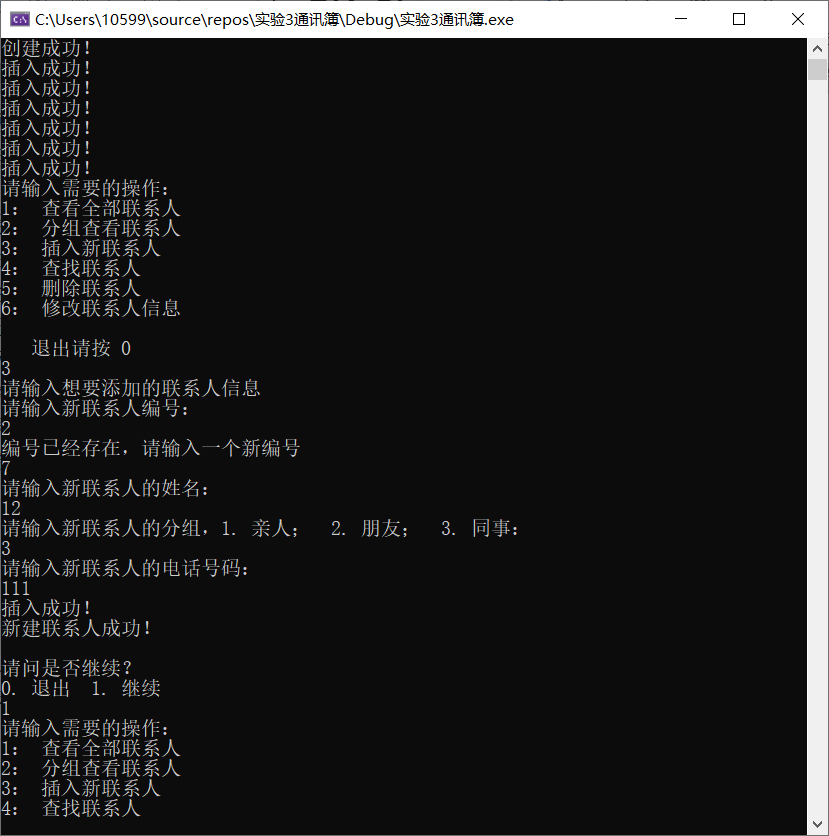
return 0;

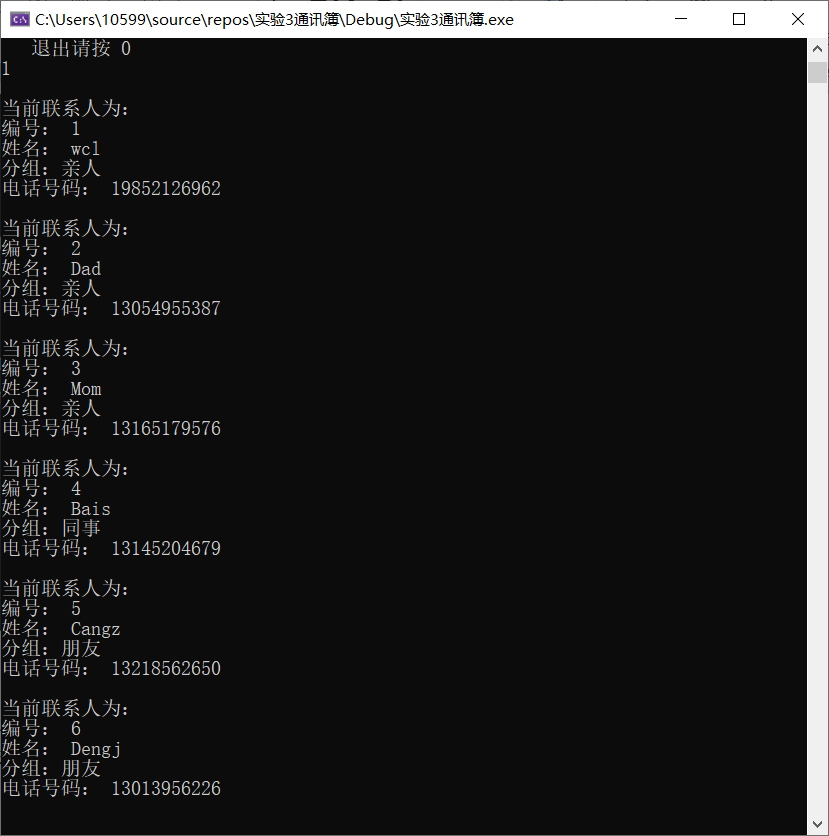
}

start(address\_list);

}

**【实验过程】**





**【实验体会】**

由于顺序表可以进行随机访问，查找某人电话号码，添加新记录，修改记录，删除记录，打印亲人清单、朋友清单或同事清单以及退出十分便捷。需要注意的是last和循环的配合使用。

**4 将中缀表达式转换为后缀表达式**

【问题描述】表达式转换。输入的中缀表达式为字符串，转换得到的后缀表达式存入字符数组中并输出。

例如： a\*(x+y)/(b-x) 转换后得： a x y + \* b x - /

【数据结构】

* 定义一个暂时存放运算符的转换工作栈opst。
* 中缀表达式字符串char \*infix;
* 后缀表达式字符串char \*postfix;

【主要代码】

头文件：

#pragma once

#ifndef head\_H

#define head\_H

#include <iostream>

using namespace std;

template <class T>

struct linknode //节点类

{

T data; //数据

linknode\* link; //链表指针

//data缺省的构造函数

linknode(linknode<T>\* p = NULL)

{

link = p;

}

//初始化data的构造函数

linknode(const T& temp, linknode<T>\* p = NULL)

{

data = temp; link = p;

}

};

template <class T>

class list

{

protected:

linknode<T>\* first; //头指针

public:

list(){first = new linknode<T>;}//默认构造函数

list(const T& x){first = new linknode<T>(x);}//构造函数

list(list<T>& x);//复制构造函数

~list() { makempty(); }//析构函数

void makempty();//把表变为空表

int getlength();//返回表长度

linknode<T>\* gethead(){return first;}//返回附加头节点地址

linknode<T>\* search(T x);//返回含x的节点的地址

linknode<T>\* locate(int i);//返回第i个元素的地址

bool getdata(int i, T& x)const;//把第i个元素的值给x

bool setdata(int i, T x);//把第i个元素的值改为x

bool insert(int i, T x);//把x插入到第i个元素后面

bool remove(int i, T& x);//删除第i个元素的值

bool isempty()const //判断是否空

{

first->link == NULL ? true : false;

}

bool isfull()const //是否满

{

return false;

}

void inputfr(T end);//前插法

void inputbc(T end);//后插法

void output();//输出

list<T>& operator=(list<T>& l);//重载等号

};

//把表变为空表函数

template <class T>

void list<T>::makempty()

{

linknode<T>\* temp; //当前结点

//当链表不空的时候，删除除了表头结点的所有节点

while (first->link != NULL)

{

temp = first->link;

first->link = temp->link;

delete temp;

}

}

//前插法函数

template <class T>

void list<T>::inputfr(T end)

{

makempty();

T temp;

linknode<T>\* newnode;

do {

cout << "请输入数据(输入-1结束)" << endl;

cin >> temp;

if (temp == end)

break;

newnode = new linknode<T>(temp);

if (newnode == NULL)

{

cerr << "内存分配错误" << endl; exit(1);

}

newnode->link = first->link;

first->link = newnode;

} while (true);

}

//后插法函数

template <class T>

void list<T>::inputbc(T end)

{

makempty();

linknode<T>\* newnode; //待插入节点

linknode<T>\* last = first; //尾节点

T temp;

do

{

cout << "请输入数据(输入-1结束)" << endl;

cin >> temp;

if (temp == end)

break;

newnode = new linknode<T>(temp);

if (newnode == NULL)

{

cerr << "内存分配错误" << endl; exit(1);

}

last->link = newnode;

last = newnode;

} while (true);

last->link = NULL;

}

//输出函数

template<class T>

void list<T>::output()

{

linknode<T>\* temp = first->link;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->link;

}

cout << endl;

}

//复制构造函数

template<class T>

list<T>::list(list<T>& x)

{

first = new linknode<T>;

if (first == NULL)

{

cerr << "内存分配错误" << endl; exit(1);

}

linknode<T>\* src = NULL;

linknode<T>\* des = NULL;

T temp;

src = x.gethead()->link;

des = first;

while (src != NULL)

{

temp = src->data;

des->link = new linknode<T>(temp);

if (des->link == NULL)

{

cerr << "内存分配错误" << endl; exit(1);

}

des = des->link;

src = src->link;

}

}

// 返回表长度函数

template<class T>

int list<T>::getlength()

{

linknode<T>\* temp = first->link;

int count = 0;

while (temp != NULL)

{

count++;

temp = temp->link;

}

return count;

}

//返回含x的节点的地址函数

template<class T>

linknode<T>\* list<T>::search(T x)

{

linknode<T>\* temp = first->link;

while (temp != NULL)

{

if (temp->data == x)

break;

temp = temp->link;

}

return temp;

}

//返回第i个元素

template<class T>

linknode<T>\* list<T>::locate(int i)

{

if (i <= 0)

{

cout << "错误" << endl;

return NULL;

}

linknode<T>\* temp = first->link;

for (int a = 1; a < i; a++)

{

if (temp == NULL)

{

cout << "表长度不够" << endl;

return NULL;

}

temp = temp->link;

}

return temp;

}

//把第i个元素的值给x函数

template<class T>

bool list<T>::getdata(int i, T& x) const

{

if (first->link == NULL)

{

cout << "空表" << endl; return false;

}

linknode<T>\* temp = first->link;

for (int a = 1; a < i; a++)

{

if (temp == NULL)

{

cout << "表长不够" << endl;

return false;

}

temp = temp->link;

}

x = temp->data;

return true;

}

//把第i个元素的值改为x的函数

template <class T>

bool list<T>::setdata(int i, T x)

{

if (first->link == NULL)

{

cout << "空表" << endl; return false;

}

linknode<T>\* temp = first->link;

for (int a = 1; a < i; a++)

{

if (temp == NULL)

{

cout << "表长不够" << endl;

return false;

}

temp = temp->link;

}

temp->data = x;

return true;

}

//把x插入到第i个元素之后

template<class T>

bool list<T>::insert(int i, T x)

{

if (i < 0)

return false;

linknode<T>\* newnode = new linknode<T>(x);

linknode<T>\* temp = first;

for (int a = 0; a < i; a++)

{

temp = temp->link;

if (temp == NULL)

{

cout << "表长度不够,修改失败" << endl;

return false;

}

}

newnode->link = temp->link;

temp->link = newnode;

return true;

}

//把第i个元素删除函数

template<class T>

bool list<T>::remove(int i, T& x)

{

if (i <= 0)

return false;

linknode<T>\* del = NULL;

linknode<T>\* temp = first;

for (int a = 1; a < i; a++)

{

temp = temp->link;

if (temp == NULL)

{

cout << "表长度不够,修改失败" << endl;

return false;

}

}

del = temp->link;

temp->link = del->link;

x = del->data;

delete del;

return true;

}

template<class T>

list<T>& list<T>::operator=(list<T>& l)

{

makempty();

linknode<T>\* src = l.gethead()->link;

linknode<T>\* des = first;

T temp;

while (src != NULL)

{

temp = src->data;

des->link = new linknode<T>(temp);

if (des->link == NULL)

{

cerr << "内存分配错误" << endl; exit(1);

}

des = des->link;

src = src->link;

}

return \*this;

}

template <class T>

class linkedstack: public list<T>

{

public:

linkedstack() { top = NULL; } //构造函数

~linkedstack() { makempty(); } //析构函数

void push(const T& x); //进栈

bool pop(T& x); //出栈

bool gettop(T& x) const; //读取栈顶元素

bool isempty() const { return top == NULL ? true : false; } //是否空

bool isfull() const { return false; }//是否满

int getsize() const; //元素个数

void makempty(); //置空

friend ostream& operator<<(ostream& os, linkedstack<T>& s)

{

os << "栈中元素个数为：" << s.getsize() << endl;

linknode<T>\* temp = s.top;

for (int i = 0; i < s.getsize(); i++)

{

cout << "第" << i + 1 << "个元素为：" << temp->data << endl;

temp = temp->link;

}

return os;

}

private:

linknode<T>\* top;

};

//进栈函数

template<class T>

void linkedstack<T>::push(const T& x)

{

linknode<T>\* temp = new linknode<T>(x);

temp->link = top;

top = temp;

}

//出栈函数

template<class T>

bool linkedstack<T>::pop(T& x)

{

if (isempty())

return false;

linknode<T>\* temp = top;

top = top->link;

x = temp->data;

delete temp;

return true;

}

//读取栈顶元素

template<class T>

bool linkedstack<T>::gettop(T& x) const

{

if (isempty())

return false;

x = top->data;

return true;

}

//置空函数

template<class T>

void linkedstack<T>::makempty()

{

linknode<T>\* temp = NULL;

while (top != NULL)

{

temp = top;

top = temp->link;

delete temp;

}

}

//返回元素个数函数

template<class T>

int linkedstack<T>::getsize() const

{

int size = 0;

linknode<T>\* temp = top;

while (temp != NULL)

{

temp = temp->link;

size++;

}

return size;

}

bool check(char h) //检查输入的表达式是否合法

{

if ((h >= 'a' && h <= 'z') || (h >= 'A' && h <= 'Z') || (h >= '0' && h <= '9') || h == '.')//是操作数

return true;

else if (h == '#' || h == '(' || h == '\*' || h == '/' || h == '%' || h == '+' || h == '-' || h == ')') //是操作符

return true;

else //非法字符

{

cout << "表达式中有不合法的字符！请重新输入" << endl;

return false;

}

}

void input(char\* temp)

{

while (true) //输入原始表达式

{

cout << "请输入要转换的表达式" << endl;

cin.getline(temp, 100); //整行输入

int checkflag = 0; //判定是否有非法字符

int length = strlen(temp); //输入的表达式的长度

for (int i = 0; i < length; i++) //逐个检查是否有非法字符

{

if (!check(temp[i])) //出现非法字符

{

checkflag = 1; //判定变为真

break; //只要发现一个非法字符，整个表达式及非法，无继续判断

}

}

if (checkflag == 0) //若无非法字符

{

temp[length] = '#'; //最后添加表达式结束标志#

temp[length + 1] = '\0'; //字符串结尾

cout << endl << "原表达式为（#表示结尾）：" << temp << endl << endl;

break;

}

}

}

bool isnum(char h) //判断当前字符是操作数还是操作符

{

if ((h >= 'a' && h <= 'z') || (h >= 'A' && h <= 'Z') || (h >= '0' && h <= '9') || h == '.')

return true;

else

return false;

}

int isp(char h) //返回当前操作符栈内优先级

{

if (h == '#')

return 0;

else if (h == '(')

return 1;

else if (h == '\*' || h == '/' || h == '%')

return 5;

else if (h == '+' || h == '-')

return 3;

else

return 6;

}

int icp(char h) //返回当前操作符栈外优先级

{

if (h == '#')

return 0;

else if (h == '(')

return 6;

else if (h == '\*' || h == '/' || h == '%')

return 4;

else if (h == '+' || h == '-')

return 2;

else

return 1;

}

void transfer(char\* temp, char\* out)

{

linkedstack<char> opst;

opst.push('#'); //将#先压入栈

char topch;//用于取栈顶元素

char outch;//用于取退栈元素

int i1 = 0, i2 = 0;

char ch = temp[i1++]; //利用ch从原中缀表达式中读取字符

while (opst.isempty() == false) //逐个处理

{

if (isnum(ch)) //操作数

{

out[i2++] = ch;//输出到后缀表达式

ch = temp[i1++];

}

else //操作符

{

opst.gettop(topch); //取栈顶操作符给topch

if (isp(topch) < icp(ch)) //新操作符优先级高

{

opst.push(ch);//新操作符入栈

ch = temp[i1++]; //取下一操作符

}

else if (isp(topch) > icp(ch)) //新操作符优先级低

{

opst.pop(outch);//退栈

out[i2++] = outch;//输出退栈元素

}

else //栈顶元素和当前操作符优先级相等，即括号或井号

{

opst.pop(outch);

if (outch == '(')

ch = temp[i1++];

}

}

if (ch == '#') //收尾工作，把最后一个操作符输出,并且跳出循环

{

opst.pop(outch);//退栈

out[i2++] = outch;//输出退栈元素

break;

}

}

out[i2] = '\0'; //输出字符串添加结尾

}

#endif head\_H

源文件：

#include <iostream>

#include "head.h"

int main()

{

char\* infix = new char; //中缀表达式字符串

char\* postfix = new char[100];//后缀表达式字符串

input(infix);

transfer(infix, postfix);

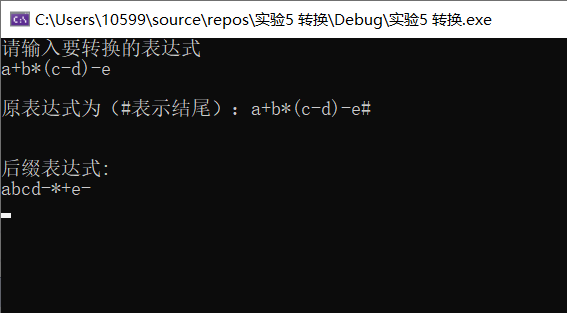
cout << endl << "后缀表达式:" << endl;

cout << postfix << endl;

return 0;

}

【实验过程】



【实验体会】

除了括号和结尾标志以外，其他运算符号的栈外优先级和栈内优先级之间要相差1。另外不能完全按照书上的程序编写。

**8 二叉树的操作**

**【实验简介】**二叉树是树形结构的一种重要类型。通过本次实验，熟悉二叉树结点的结构，掌握二叉树的基本操作以及具体实现，学会利用递归方法编写对二叉树这种递归数据结构进行处理的算法。

**【实验内容】**

编写程序，实现对二叉树的以下操作：

1. 建立二叉树。
2. 按任一种遍历次序输出二叉树中的所有结点。
3. 求二叉树的深度。
4. 求二叉树中的所有节点数。
5. 求二叉树中的所有叶子节点数。
6. 清除二叉树，使之编程一只空树。

**【主要代码】**

头文件：

#pragma once

#ifndef head\_h

#define head\_h

#include<iostream>

#include <fstream>

using namespace std;

template <class t>

struct bintree\_node

{

t data;//数值域

bintree\_node<t>\* left\_child, \* right\_child;//链域

bintree\_node() //不带默认参数的构造函数

{

left\_child = NULL;

right\_child = NULL;

}

bintree\_node(t x, bintree\_node<t>\* l = NULL, bintree\_node<t>\* r = NULL)

{//带默认参数的构造函数

data = x;

left\_child = l;

right\_child = r;

}

};

template<class t>

class bintree

{

public:

bintree() { root = NULL; }//不带默认参数的构造函数

bintree(t value) { ref\_value = value;root = NULL; }//带默认参数的构造函数

~bintree() { destroy(root); };//析构函数

bintree\_node<t>\* get\_root(){ return root; }//取根

int Hight() { return height(root); }//返回高度

void bintree\_count(bintree\_node<t>\* subtree,int& m1,int& m2);//m1是总结点数,m2是叶子结点数

void print\_bintree( bintree\_node<t>\* tree);//前序遍历输出二叉树

void set\_ref(t ref) { ref\_value = ref; }//结束符设置

private:

t ref\_value;//数据停止标志

bintree\_node<t>\* root;//根节点

static void create(istream& in, bintree\_node<t>\*& subtree, t ref); //静态成员函数，从文件读入建树，递归方式

friend istream& operator>>(istream& in, bintree<t>& tree) {//>>重载

create(in, tree.root,tree.ref\_value);

return in;

}

void destroy(bintree\_node<t>\*& subtree);//删除

int height(bintree\_node<t>\* subtree); //返回树高度

};

template<class t>

void bintree<t>::destroy(bintree\_node<t>\*& subtree)

{//后序遍历，递归删除

if (subtree != NULL)

{

destroy(subtree->left\_child);

destroy(subtree->right\_child);

delete subtree;

}

}

template<class t>

int bintree<t>::height(bintree\_node<t>\* subtree)

{//后序遍历，递归访问

if (subtree == NULL)

return 0;

else

{

int i = height(subtree->left\_child);

int j = height(subtree->right\_child);

return (i > j) ? i + 1 : j + 1;

}

}

template<class t>

void bintree<t>::create(istream& in, bintree\_node<t>\*& subtree,t ref)

{//前序遍历，读入文件建树

t item;

if (!in)

{

cout << "文件不能打开！" << endl;

}

else

{

in >> item;

if (item != ref)

{

subtree = new bintree\_node<t>(item);

if (subtree == NULL)

{

cout << "内存分配错误！" << endl; exit(1);

}

create(in,subtree->left\_child,ref);

create(in,subtree->right\_child,ref);

}

else

subtree = NULL;

}

}

template<class t>

void bintree<t>::print\_bintree(bintree\_node<t>\* tree)

{//前序遍历，输出

if (tree != NULL)

{

cout << tree->data;

if (tree->left\_child != NULL || tree->right\_child != NULL)

{

cout << "(";

print\_bintree(tree->left\_child);

cout << ",";

print\_bintree(tree->right\_child);

cout << ")";

}

}

}

template<class t>

void bintree<t>::bintree\_count(bintree\_node<t>\* subtree,int& m1, int& m2)

{//前序遍历，统计所有结点和叶子结点

if (subtree != NULL)

{

m1++;//统计所有结点

if (subtree->left\_child == NULL && subtree->right\_child == NULL)

m2++;//统计叶子结点

bintree\_count(subtree->left\_child, m1, m2);

bintree\_count(subtree->right\_child, m1, m2);

}

}

#endif head\_h

源文件：

#include <iostream>

#include <fstream>

#include"head.h"

using namespace std;

int main()

{

int a = 0, b = 0;

ifstream open\_num;

open\_num.open("num.txt", ios::in);

bintree<char> num;

num.set\_ref('0');//把结束符设置为‘0’

open\_num >> num;//读入文件建树

open\_num.close();

num.print\_bintree(num.get\_root());//前序遍历输出

cout << endl;

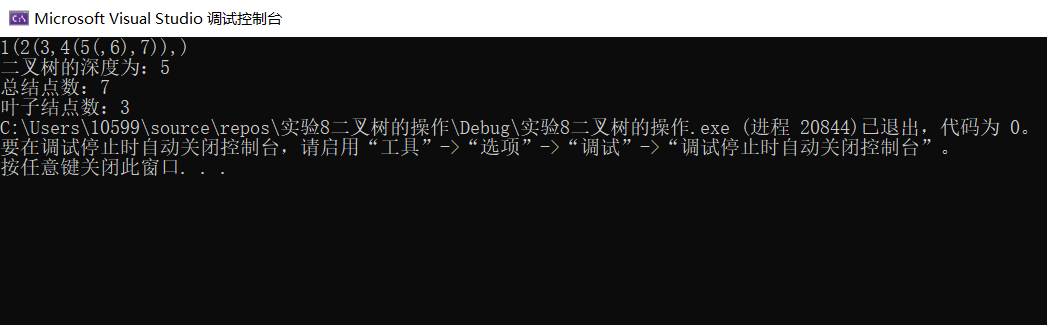
cout << "二叉树的深度为：" << num.Hight() << endl;//后序遍历求深度

num.bintree\_count(num.get\_root(), a, b);

cout << "总结点数：" << a << "\n" << "叶子结点数：" << b;

}

**【实验过程】**



**【实验体会】**

二叉树的许多成员函数都是用递归方式编写的，其中我认为难度最大的是把文件输入流和>>运算符重载结合，通过友元函数重载的同时，还需要把调用的create函数声明为静态，多个对象就可以共用create函数。另外，根据二叉树存储的数据类型不同，需要使用set\_ref函数设置数据停止标志。统计结点使用引用传递方式，把结点数和叶子结点数传给了两个变量，一个函数就可以得到两个数。

**5 字符串**

【实验简介】字符串是由零个或多个字符的顺序排列所组成的数据结构，字符串在计算机处理中使用非常广泛。通过本次实验理解字符串运算的原理，掌握主要算法的实现。

【实验内容】

建立字符串类，并实现求子串、字符串赋值、字符串连接等运算符重载函数，实现字符串的模式匹配功能。编写一个能够统计字符串中各个字符出现频度的函数。

【主要代码】

头文件：

#pragma once

#ifndef head\_H

#define head\_H

#define default\_size 128

#include <iostream>

#include <string>

using namespace std;

class Astring

{//字符类定义

private:

char\* ch;//存放字符串的数组

int length;//实际长度

int max\_size;//最大长度

public:

Astring(int sz = default\_size);//使用默认参数的构造函数

Astring(const char\* init);//构造函数

~Astring() { delete[]ch; }//析构函数

int get\_length()const { return length; };//返回实际长度

void operator()(Astring& temp,int pos, int len);//求子串

bool operator==(Astring& ob)const { return strcmp(ch, ob.ch) == 0; }//判断字符串是否相等，相等返回1，不等返回0

friend bool operator!=(Astring& ob1, Astring& ob2) { return strcmp(ob1.ch, ob2.ch) != 0; }//判断字符串是否不等，相等返回0，不等返回1

bool operator!()const { return length == 0; }//判断是否为空，空为1，非空为0

Astring& operator=(Astring& ob);//字符串赋值重载，必须有&，否则和this指针无法比较

Astring& operator+=(const Astring& ob);//字符串连接重载

int find(Astring& pat)const;//模式匹配,返回第一个匹配到的下标

friend ostream& operator <<(ostream& os, Astring& temp);//<<重载，使之可以显示Astring对象的属性

int count(char a);//统计出现次数

void num();//统计小写字母，大写字母，空格出现次数

};

Astring::Astring(int sz) {

max\_size = sz;

ch = new char[max\_size + 1];

if (ch == NULL) {

cout << "存储分配错误！";

}

length = 0;

ch[0] = '\0';//初始化

cout << "创建成功！" << endl;

}

Astring::Astring(const char\* init) {

int len = strlen(init);

max\_size = (len > default\_size) ? len : default\_size;

ch = new char[max\_size + 1];

if (ch == NULL) {

cout << "存储分配错误！";

}

length = len;

strcpy\_s(ch, len + 1, init);//拷贝

cout << "创建成功！" << endl;

}

void Astring::operator()(Astring& temp, int pos, int len)

{//求子串，pos是开始取值的下标，len是子串长度

if (len < 0 || pos < 0 || len + pos - 1 > max\_size)

{

temp.length = 0;

temp.ch[0] = '\0';

}

else

{

if (len + pos - 1 >= length)

{

len = length - pos;

}

temp.length = len;

for (int i = 0, j = pos; i < len; i++, j++)

{

temp.ch[i] = ch[j];

}

temp.ch[len] = '\0';

}

}

Astring& Astring::operator=(/\*const \*/Astring& ob) {

if (&ob != this) {

delete[]ch;

ch = new char[ob.max\_size + 1];

if (ch == NULL)

cout << "内存分配错误！" << endl;

length = ob.length;

strcpy\_s(ch, length + 1, ob.ch);//拷贝

}

else

cout << "字符串自身赋值出错！" << endl;

return \*this;

}

Astring& Astring::operator+=(const Astring& ob)

{

char\* temp = ch;

int n = length + ob.length;

int m = (max\_size > n) ? max\_size : n;

ch = new char[m + 1];

if (ch == NULL)

cout << "存储分配错!";

max\_size = m; length = n;

strcpy\_s(ch, max\_size + 1, temp);//拷贝

strcat\_s(ch, max\_size + 1, ob.ch);//连接

delete[]temp;

return \*this;

}

int Astring::find(Astring& pat)const {//普通的模式匹配

int i, j;

if (length < pat.length) {

cout << "\n长度超限，匹配失败！";

return -1;

}

for (i = 0; i < length - pat.length; i++) {

for (j = 0; j < pat.length; j++) {

if (ch[i + j] != pat.ch[j])//+j

break;

}

if (j == pat.length)

return i;

}

return -1;

}

ostream& operator <<(ostream& os, Astring& temp)

{//重载<<，使之可以显示Astring对象的属性

os << temp.ch << endl;

os << "当前字符串最大容量为：" << temp.max\_size << endl;

os << "已有字符 " << temp.length << " 个" << endl;

return os;

}

int Astring::count(char a) {

int num=0;

for (int i = 0; i < length; i++)

{

if (ch[i] == a)

num++;

}

return num;

}

void Astring::num()

{

int c = 0;//每个字符出现次数

int num = 0;

cout << "共有" << length << "个字符" << endl;

for (char a = 'a'; a <= 'z'; a++)

{//统计小写字母

c = count(a);

num += c;

if (c != 0)

{

cout << "出现字母 " << a << "共 " << c << " 次" << endl;

}

}

for (char a = 'A'; a <= 'Z'; a++)

{//统计大写字母

c = count(a);

num += c;

if (c != 0)

{

cout << "出现字母 " << a << "共 " << c << " 次" << endl;

}

}

//统计空格

c = count(' ');

num += c;

cout << "出现空格共 " << c << "次" << endl;

//统计其他字符

cout << "出现其他字符共 " << length - num << "次" << endl;

}

#endif head\_H

源文件：

#include"head.h"

#include <iostream>

using namespace std;

int main()

{

Astring A;

Astring B("Consistent");

Astring C(" studying!");

cout << endl << "字符串B： " << B << endl;

B(A, 2, 3);//产生子串

cout << endl << "B的一个子串是： " << A;

cout << "B与A模式匹配结果： " << B.find(A) << endl;

A = C;

cout << endl << "赋值后字符串A： " << A;

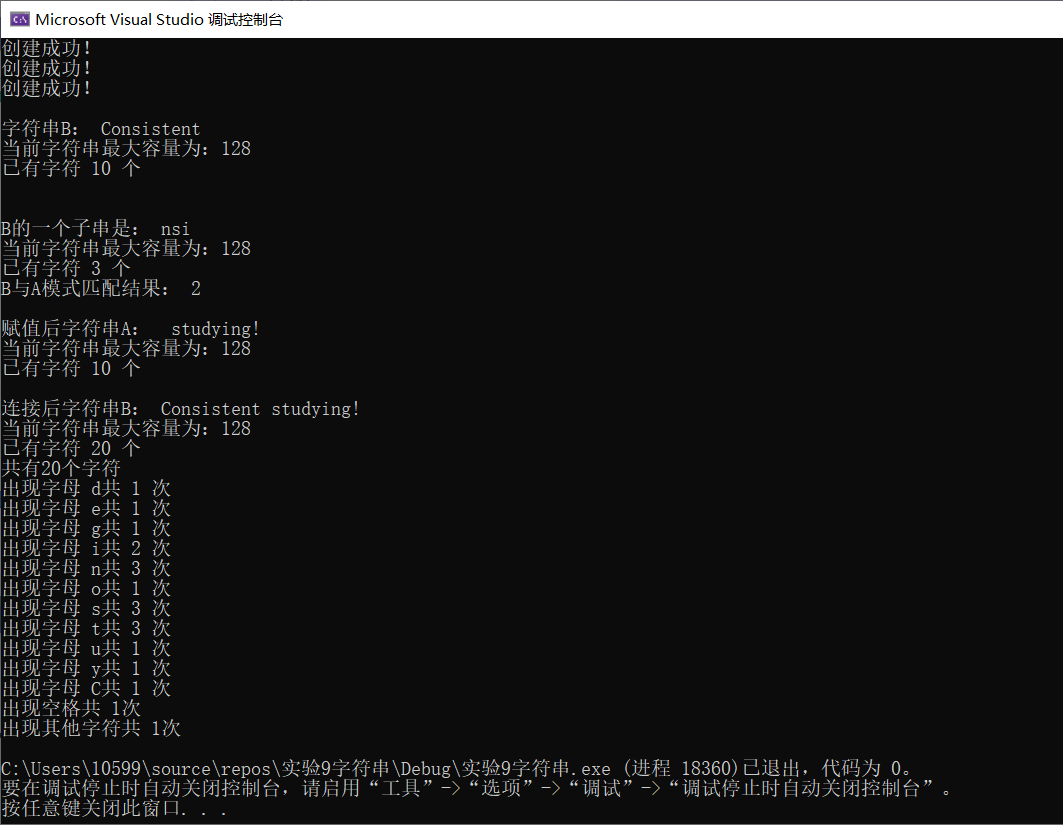
B += C;

cout << endl << "连接后字符串B： " << B;

B.num();//统计小写字母，大写字母，空格出现次数

}

【实验过程】



【实验体会】

课本上求子串的代码会触发一个断点，我把对（）的重载做了修改，加上了一个队已有Astring对象的引用，直接修改已有对象。要注意遍历过程中考虑字符串实际长度，最后有个’\0’。模式匹配我使用朴素匹配方式。统计字符串中各个字符出现频度的函数只能做到统计小写字母、大写字母和空格，或许可以使用ASCII码遍历，但是太过浪费时间。