**《空间数据结构基础》**

**上机实验报告（2017级）**

**姓名 张清昱**

**班级 地信17-1班**

**学号 07172336**

**环境与测绘学院**

**C++中的类与对象**

**【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. 调试相关程序
2. 理解教材P9-P24页的相关知识

**【程序代码】**

头文件Point.h：

#ifndef POINT\_H

#define POINT\_H

#include<iostream>

**using** **namespace** std;

**class** Point

{

**private**:

**int** x;

**int** y;

**public**:

    Point(**int** a,**int** b);

    ~Point();

**int** get\_x();

**int** get\_y();

**private**:

**friend** ostream& **operator**<<(ostream& **out**,Point& p);

**public**:

**void** set\_x(**int** a);

**void** set\_y(**int** b);

    Point **operator**+(Point p);

};

Point::Point(**int** a,**int** b)

{

    x=a;

    y=b;

}

Point::~Point(){}

**int** Point::get\_x()

{

**return** x;

}

**int** Point::get\_y()

{

**return** y;

}

ostream& **operator**<<(ostream& **out**,Point& p)

{

**return** **out**<<"private p.x="<<p.x<<endl<<"private p.y="<<p.y<<endl;

}

**void** Point::set\_x(**int** a)

{

    x=a;

}

**void** Point::set\_y(**int** b)

{

    y=b;

}

Point Point::**operator**+(Point p)

{

    Point ResultPoint(0,0);

    ResultPoint.set\_x(x+p.get\_x());

    ResultPoint.set\_y(y+p.get\_y());

**return** ResultPoint;

}

#endif

源文件Point.cpp：

#include "Point.h"

#include <iostream>

**int** main()

{

    Point P1(6,3);

    cout<<"P1.x="<<P1.get\_x()<<endl;

    cout<<"P1.y="<<P1.get\_y()<<endl;

**int** x,y;

    cout<<"请输入P2点的x坐标：";

    cin>>x;

    cout<<"请输入P2点的y坐标：";

    cin>>y;

    Point P2(x,y);

    cout<<"P2.x="<<P2.get\_x()<<endl;

    cout<<"P2.y="<<P2.get\_y()<<endl;

    Point P3(8,18);

    cout<<P3;

    Point \*p4=**new** Point(10,20);

    cout<<"p4.x="<<p4->get\_x()<<endl<<"p4.y="<<p4->get\_y()<<endl;

    Point P5(1,1);

    Point P6(2,2);

    Point P7=P5+P6;

    cout<<"p7.x="<<P7.get\_x()<<endl<<"p7.y="<<P7.get\_y()<<endl;

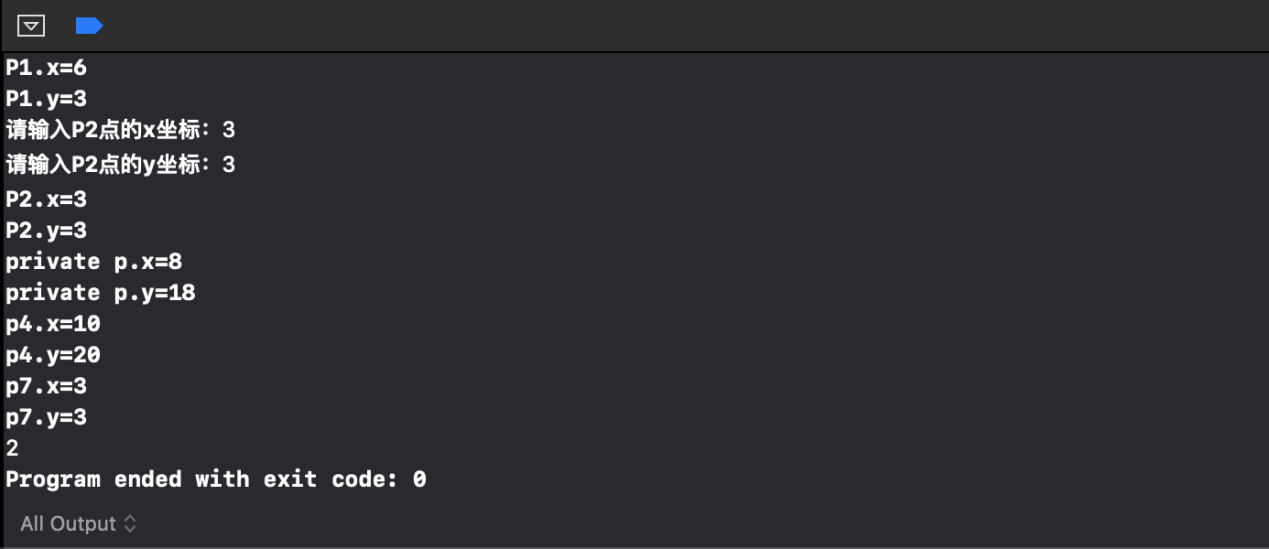
**int** pause;

    cin>>pause;

}

**【实验过程】**

1. 打开编译软件，新建一个工程
2. 将文件内的源代码进行复制粘贴
3. 将两个文件内所有的注释全部删除
4. 编译文件，输入必要数值，程序可直接运行



**【实验体会】**

本次实验相对来说难度较低，但还是很考验我们的基础观察能力和对C++语言的理解，开始我以为源文件内头部少了声明，想都没想就先加上了，后来发现只要声明了自定义的头文件，头文件内的声明也会被继承到源文件内，这是我遗漏的一个很重要的点，会在以后的学习中用到。而且开始注释的‘/’和‘/\* \*/’被混用了，导致了程序不能直接运行，部分有用的代码被注释掉了，这警示我们，哪怕是注释，也要注意它的的规则与用法。

**顺序表**

**【实验简介】**

学会用算法语言C++描述抽象数据类型，使用模板建立顺序表数据结构。深入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顺序表数据类型。类的数据成员提供如maxSize等对象属性，成员函数提供插入等操作方法，方法是公共接口，用户通过调用方法实现对属性或保护函数的访问。

**【实验内容】**

1.将所学的顺序表进行编写

2.将顺序表用于简单实现

3.使用顺序表各函数的功能，加深理解

4.调试教材中顺序表的模板类定义和操作

**【程序代码】**

头文件seqList.h：

#ifndef seqList\_h

#define seqList\_h

#include <iostream>

#include <stdlib.h>

**using** **namespace** std;

**const** **int** defaultSize = 100;

**template** <**class** T>

**class** SeqList{

**protected**:

    T \*data;

**int** maxSize;

**int** last;

**void** reSize(**int** newSize);

**public**:

    SeqList(**int** sz = defaultSize);

    SeqList(SeqList<T>& L);

    ~SeqList(){**delete**[] data;}

**int** Size()**const**{**return** maxSize;}

**int** Length()**const**{**return** last+1;}

**int** Search(T& x)**const**;

**int** Locate(**int** i)**const**;

**bool** getData(**int** i, T& x)**const**;

**void** setData(**int** i, T& x);  *//int& x == int &x*

**bool** Insert(**int** i, T& x);

**bool** Remove(**int** i, T& x);

**bool** IsEmpty(){**return** (last == -1)?**true**:**false**;}

**bool** IsFull(){**return** (last == maxSize)?**true**:**false**;}

**void** Expand();

**int** getBlank(){**return** maxSize-last;}

**void** input();

**void** output();

    SeqList<T> **operator**=(SeqList<T>&L);

};

#endif */\* seqList\_h \*/*

源文件main.cpp：

#include <iostream>

#include <stdlib.h>

#include "seqList.h"

**using** **namespace** std;

**template** <**class** T>

SeqList<T>::SeqList(**int** sz){     *//构造函数，通过sz定义数组的长度*

**if**(sz > 0)

    {

        maxSize = sz;last = -1;

        data = **new** T[maxSize];

        cout << "初始化完成，该顺序表空间大小为：" << sz << endl;

**if** (data == **NULL**)

        {

            cerr << "存储分配错误！" << endl;exit(1);

        }

    }

}

**template** <**class** T>

SeqList<T>::SeqList(SeqList<T>&L){      *//复制构造函数，用参数表中给出的已有顺序表初始化新建的顺序表*

    maxSize = L.Size();

    last = L.Length() - 1;

    T value;

    data = **new** T[maxSize];

**if**(data == **NULL**)

    {

        cerr << "存储分配错误！" << endl;exit(1);

    }

**for**(**int** i = 1;i<=last+1;i++)

    {

        L.getData(i, value);data[i-1]=value;

    }

}

**template** <**class** T>

**void** SeqList<T>::reSize(**int** newSize){       *//私有函数：扩充顺序表的存储数组空间为newSize个*

**if**(newSize<=0)

    {

        cerr << "无效的数组大小！" << endl;**return**;

    }

**if**(newSize != maxSize){

        T \* newarray = **new** T[newSize];

**if**(newarray == **NULL**)

        {

            cerr << "存储分配错误！" << endl;exit(1);

        }

**int** n = last+1;

        T \* srcptr = data;

        T \* desptr = newarray;

**while**(n--) \*desptr++=\*srcptr++;

**delete** []data;

        data = newarray;maxSize = newSize;

    }

}

**template** <**class** T>

**int** SeqList<T>::Search(T& x)**const**{

**for** (**int** i = 0; i<=last; i++)

**if**(data[i] == x)

**return** i+1;

**return** 0;

}

**template** <**class** T>

**int** SeqList<T>::Locate(**int** i)**const**{

**if**(i >= 1 && i <= last+1)

**return** data[i-1];

**else**

**return** 0;

}

**template** <**class** T>

**bool** SeqList<T>::getData(**int** i, T& x)**const**{

**if**(i > 0 && i <= last+1)

    {

        x = data[i-1];

**return** **true**;

    }

**else**

**return** **false**;

}

**template** <**class** T>

**void** SeqList<T>::setData(**int** i, T& x){

**if**(i > 0 && i <= last+1)

        data[i-1] = x;

}

**template** <**class** T>

**bool** SeqList<T>::Insert(**int** i, T& x){

**if**(last == maxSize)

    {

        cout << "插入失败！顺序表已满！" << endl;

**return** **false**;

    }

**if**(i < 0||i > last+1)

    {

        cout << "插入失败！指定位置不存在！" << endl;

**return** **false**;

    }

**for**(**int** j = last;j >= i-1;j--)

        data[j+1] = data[j];

    data[i-1] = x;

    last++;

**return** **true**;

}

**template** <**class** T>

**bool** SeqList<T>::Remove(**int** i, T& x){

**if**(last == -1)

    {

        cout << "删除失败！顺序表为空！" << endl;

**return** **false**;

    }

**if**(i < 1||i > last+1)

    {

        cout << "删除失败！指定位置不存在！" << endl;

**return** **false**;

    }

    x = data[i-1];

**for**(**int** j = i;j <= last;j++)

        data[j-1] = data[j];

    last--;

**return** **true**;

}

**template** <**class** T>

**void** SeqList<T>::Expand(){

**if** ((**double**)maxSize/(**double**)last >= 0.5)

    {

        reSize(2 \* maxSize);

        cout << "顺序表内空间占用过半，已自动扩展顺序表空间至原来的两倍" << endl;

    }

}

**template** <**class** T>

**void** SeqList<T>::input(){

    cout << "开始对顺序表进行插入操作，请输入表中想要插入的元素个数：";

**while** (1) {

        cin >> last;

**if**(last <= maxSize) **break**;

        cout << "表元素个数输入有误，范围不能超过" << maxSize << "，请重新输入：";

    }

**for** (**int** i = 0; i < last; i++)

    {cin >> data[i];cout << "//已输入顺序为" << i+1 << "的元素" << endl;}

}

**template** <**class** T>

**void** SeqList<T>::output(){

    cout << "顺序表当前元素最后位置为：" << last << endl;

**for**(**int** i = 0; i < last; i++)

        cout << "#" << i+1 << " " << data[i] << endl;

}

**template** <**class** T>

SeqList<T> SeqList<T>::**operator**=(SeqList<T> &L){

    maxSize = L.Size();

    last = L.Length() - 1;

    T value;

    data = **new** T[maxSize];

**if**(data == **NULL**)

    {

        cerr << "存储分配错误！" << endl;

        exit(1);

    }

**for**(**int** i = 1;i<=last+1;i++)

    {

        L.getData(i, value);

        data[i-1]=value;

    }

**return** \***this**;

}

**int** main()

{

**int** sz;

    cout << "准备建立顺序表，请输入顺序表初始最大容量：";

    cin >> sz;

    SeqList<**int**> L1(sz);

**if** (L1.IsEmpty())

        cout << "该顺序表暂空" << endl;

    L1.input();

    L1.output();

**if** (L1.IsFull())

        cout << "该顺序表已满" << endl;

**else**

        cout << "该顺序表未满，剩余" << L1.getBlank() << "个空位" << endl;

**int** b,a,c,d;  *//b此处存放顺序表的某个位置，a用来存放想要插入/删除顺序表的数值，c用来存放Remove函数弹出的删除的数据*

    cout << "请输入插入数字的位置：";

    cin >> b;

    cout << "请输入插入数字的数值：";

    cin >> a;

**if**(L1.Insert(b, a))

        cout << "插入成功" << endl;

    L1.output();

    cout << "请输入想要查询位置的数值：";

    cin >> d;

    cout << d << "所在的位置为：" << L1.Search(d) << endl;

    cout << "请输入想要确认数值的位置：";

    cin >> d;

    cout << d << "位置处所存储的数字为：" << L1.Locate(d) << endl;

    cout << "请输入想要删除的数字的位置：";

    cin >> b;

**if**(L1.Remove(b, c))

        cout << "删除成功，删除的数字为："<< c << endl;

    L1.output();

    L1.Expand();

**if** (L1.IsFull())

        cout << "该顺序表已满" << endl;

**else**

        cout << "该顺序表未满，剩余" << L1.getBlank() << "个空位" << endl;

    SeqList<**int**> L2;

    L2 = L1;

    L2.output();

**return** 0;

}

**【实验过程】**

1.打开编译软件后新建一个空白工程，命名为seqList

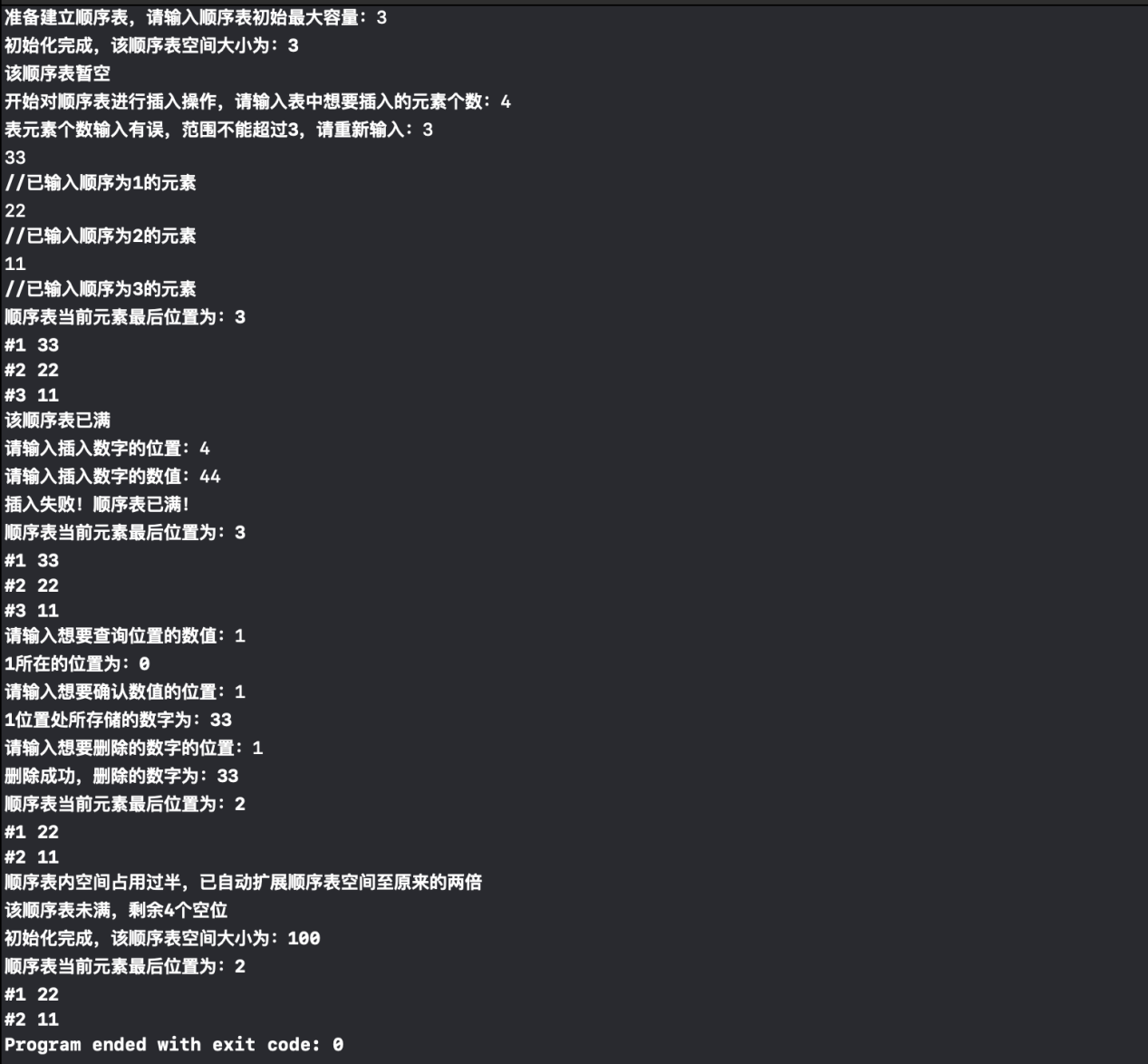
2.将课本上的顺序表代码输入到编程窗口中

3.对输入的代码进行调试直到其能编译通过

4.编写主函数对顺序表的使用进行程序实现

5.利用顺序表各函数的功能对顺序表进行操作

6.根据实际情况自己改编函数，改写头文件与主文件



**【实验体会】**

在进行实验前一定要多读几遍课本，不仅要了解各个顺序表的各函数的功能，更要深入理解其内涵与算法理由，便于实验的顺利进行。在输入代码时，一定要事无巨细，输入不正确也会影响运行。一定要将编写的各个函数都在主函数里至少使用一次，这样可以使代码可信度更高。

**用模板和非模板定义的顺序表排序类**

**【实验简介】**

学会用算法语言C++描述抽象数据类型，理解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. 调试PPT中的相关程序
2. 理解两种方式的不同与相同之处并进行比较

**【程序代码】**

1.用模板定义用于排序的数据表类

头文件datalist.h：

#ifndef DATALIST\_H

#define DATALIST\_H

#include <iostream>

#include <stdlib.h>

**using** **namespace** std;

**template** <**class** Type>

**class** dataList{

**private**:

    Type \*Element;

**int** ArraySize;

**void** Swap (**int** m1, **int** m2);

**int** MaxKey (**int** low, **int** high);

**public**:

    dataList(**int** size=10):ArraySize(size),

    Element(**new** Type[size]){}

    ~dataList () {**delete** []Element;}

**void** Sort ();

**friend** ostream& **operator** << (ostream&outStream, dataList<Type>& outList)

    {

        outStream << "数组内容: \n";

**for** (**int** i = 0; i < outList.ArraySize; i++)

            outStream << outList.Element[i] << " ";

        outStream << endl;

        outStream << "数组当前大小: " << outList.ArraySize << endl;

**return** outStream;

    }

**friend** istream& **operator** >> (istream&inStream, dataList<Type>& inList){

        cout << "录入数组当前大小: ";

        inStream >> inList.ArraySize;

        cout << "录入数组元素值: \n";

**for** (**int** i = 0; i < inList.ArraySize; i++)

        {

            cout << "元素" << i << ":";

            inStream >> inList.Element[i];

        }

**return** inStream;

    }

};

#endif

头文件selecttm.h：

#ifndef SELECTTM\_H

#define SELECTTM\_H

#include "datalist.h"

**using** **namespace** std;

**template**<**class** Type>

**void** dataList

<Type> :: Swap (**int** m1,  **int** m2)

{

    Type temp = Element [m1];

    Element [m1] = Element [m2];

    Element [m2] = temp;

}

**template** <**class** Type>

**int** dataList<Type>::MaxKey (**int** low, **int** high)

{

**int** max = low;

**for** (**int** k = low+1;k <= high;k++)

**if** ( Element[max] < Element[k] )

            max = k;

**return** max;

}

**template** <**class** Type>

**void** dataList<Type>::Sort ( )

{

**for** ( **int** i = ArraySize -1; i > 0; i-- )

    {

**int** j = MaxKey (0, i);

**if** ( j != i ) swap (j, i);

    }

}

#endif

源文件main.cpp：

#include <iostream>

#include "datalist.h"

#include "selecttm.h"

**using** **namespace** std;

**const** **int** SIZE=10;

**int** main()

{

    dataList<**int**>TestList(SIZE);

    cin>>TestList;

    cout<<TestList<<endl;

    TestList.Sort();

    cout<<TestList<<endl;

**return** 0;

}

1. 用于排序的数据表类（非模板类）

源文件main.cpp：

#include<iostream>

**using** **namespace** std;

**class** dataList

{

**public**:

**int** \*Element;

**int** ArraySize;

**void** Swap(**const** **int** mark1,**const** **int** mark2 );

**int** MaxKey(**const** **int** low,**const** **int** high);

**public**:

    dataList(**int** size=10)

    {

        ArraySize=size;

        Element=**new** **int**[size];

    }

    ~dataList()

    {

**delete**[] Element;

    }

**void** Sort();

**friend** ostream &**operator**<<(ostream &OutStream,**const** dataList &OutList);

**friend** istream &**operator**>>(istream &InStream,**const** dataList &InList);

};

istream& **operator**>>(istream&InStream,dataList&InList)

{

    cout<<"Enter array Current Size:";

    InStream>>InList.ArraySize;

    cout<<"Enter array elements:\n";

**for** (**int** i=0;i<InList.ArraySize;i++)

    {

        cout<<"Element"<<i<<":";

        InStream>>InList.Element[i];

    }

**return** InStream;

}

**void** dataList::Sort()

{

**for**(**int** i=ArraySize-1;i>0;i--)

    {

**int** j=MaxKey(0,i);

**if**(j!=i) Swap(j,i);

    }

}

ostream& **operator**<<(ostream&OutStream,**const** dataList&OutList)

{

    OutStream<<"Array Contents:\n";

**for**(**int** i=0;i<OutList.ArraySize;i++)

        OutStream<<OutList.Element[i]<<" ";

    OutStream<<endl;

    OutStream<<"Array Current Size:"<<OutList.ArraySize<<endl;

**return** OutStream;

}

**void** dataList::Swap(**const** **int** mark1,**const** **int** mark2)

{

**int** temp = Element[mark1];

    Element[mark1] = Element[mark2];

    Element[mark2] = temp;

}

**int** dataList::MaxKey(**const** **int** low,**const** **int** high)

{

**int** max = low;

**for**(**int** k=low+1;k<=high;k++)

**if**(Element[max]<Element[k]) max=k;

**return** max;

}

**int** main()

{

**const** **int** SIZE = 50;

    dataList TestList(SIZE);

    cin >> TestList;

    cout << "Testing SelectionSort:\n" << TestList << endl;

    TestList.Sort();

    cout << "After sorting:\n" << TestList << endl;

**return** 0;

}

**【实验过程】**

1.打开编译器，建立新的工程

2.初步将PPT内的程序进行拷贝

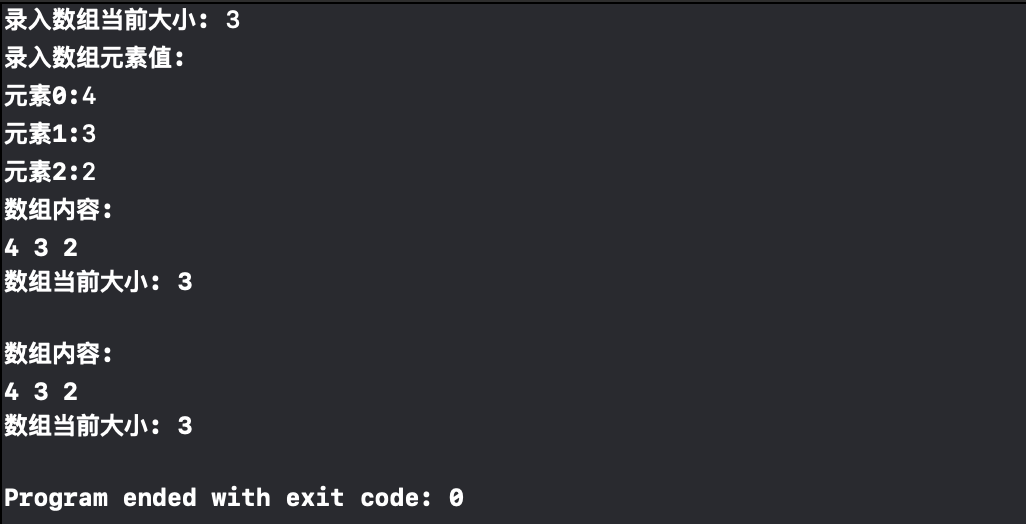
3.编译后，系统提示大约有100个错误

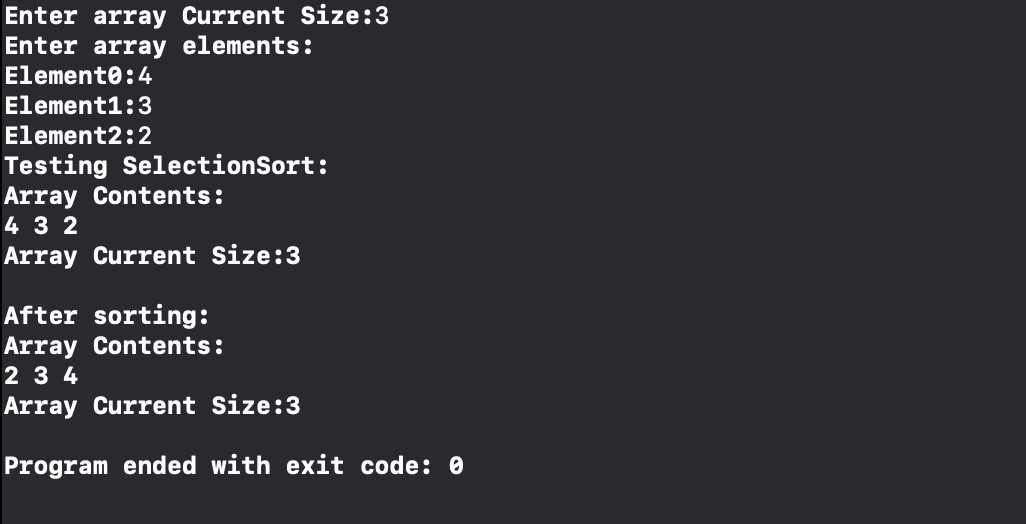
4.对程序设置断点，进行单步调试，根据提示输入数据并观察各个变量的实时数据信息

5.发现问题在于O与o的大小写上，并且没有对命名空间std进行声明

6.修改代码

7.多次调试后直到可以运行为止





**【实验体会】**

本次实验中大小写的真的尤其需要注意，在我的编译器中，ostream、Ostream、OutStream、outStream之间不能相互识别出来，必须严格注意大小写，以及命名空间std的声明必须注意，以保证使用输入输出流的重载正确。两种方法各有各的优势与劣势，不使用模板的情况下一个文件就可以完成操作，但是模板类更能体现出C++程序的特色。

**带附加表头结点的单链表的类**

**【实验简介】**

学会用算法语言C++描述抽象数据类型，使用模板建立单链表数据结构。掌握用C++描述带附加表头的单链表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】**

调试教材中带附加表头节点的单链表的模板类定义和操作

**【程序代码】**

头文件List.h：

#include<iostream>

**using** **namespace** std;

**template** <**class** T>

**struct** LinkNode

{

    T data;

    LinkNode<T> \*link;

    LinkNode(LinkNode<T> \*ptr = **NULL**)

    {

        link = ptr;

    }

    LinkNode(**const** T& item, LinkNode<T> \*ptr = **NULL**)

    {

        data = item;

        link = ptr;

    }

};

**template** <**class** T>

**class** List

{

**protected**:

    LinkNode<T> \*first;

**public**:

    List() {first = **new** LinkNode<T>;}

    List(**const** T& x) {first = **new** LinkNode<T>(x);}

    List(List<T>& L);

    ~List(){ makeEmpty(); }

**void** makeEmpty();

**int** Length()**const**;

    LinkNode<T> \*getHead() **const** {**return** first;}

    LinkNode<T> \*Search(T x);

    LinkNode<T> \*Locate(**int** i);

**bool** getData(**int** i, T&x);

**void** setData(**int** i, T&x);

**bool** Insert(**int** i, T&x);

**bool** Remove(**int** i, T&x);

**bool** IsEmpty() **const**{**return**  first->link == **NULL**?**true**:**false**;}

**bool** IsFull()**const**{**return** **false**;}

**void** Sort();

**void** inputFront(T endTag);

**void** inputRear(T endTag);

**void** output();

    List<T>& **operator**=(List<T>& L);

};

源文件main.cpp：

#include<stdlib.h>

#include"List.h"

**using** **namespace** std;

**template**<**class** T>

List<T>::List(List<T>& L)

{

    T value;

    LinkNode<T> \*srcptr = L.getHead();

    LinkNode<T> \*destptr = first = **new** LinkNode<T>;

**while** (srcptr->link != **NULL**)

    {

        value = srcptr->link->data;

        destptr->link = **new** LinkNode<T>(value);

        destptr = destptr->link;

        srcptr = srcptr->link;

    }

    destptr->link = **NULL**;

}

**template** <**class** T>

**void** List<T>::makeEmpty()

{

    LinkNode<T> \*q;

**while** (first->link != **NULL**) {

        q = first->link;

        first->link = q->link;

**delete** q;

    }

}

**template**<**class** T>

**int** List<T>::Length()**const**

{

    LinkNode<T> \*p=first->link;

**int** count=0;

**while**(p!=**NULL**)

    {

        p = p->link;

        count++;

    }

**return** count;

}

**template** <**class** T>

LinkNode<T> \*List<T>::Search(T x)

{

    LinkNode<T> \*current = first->link;

**while** (current != **NULL** && current->data != x)

        current = current->link;

**return** current;

}

**template** <**class** T>

LinkNode<T> \*List<T>::Locate(**int** i)

{

**if** (i < 0) **return** **NULL**;

    LinkNode<T> \*current = first;

**int** k = 0;

**while** (current != **NULL** && k < i)

    {

        current = current->link;

        k++;

    }

**return** current;

}

**template** <**class** T>

**bool** List<T>::getData(**int** i, T& x)

{

**if** (i <= 0)

**return** **NULL**;

    LinkNode<T>  \*current = Locate(i);

**if** (current == **NULL**)

**return**  **false**;

**else**

    {

        x = current->data;

**return** **true**;

    }

}

**template** <**class** T>

**void** List <T>::setData(**int** i, T& x)

{

**if** (i <= 0)

**return**;

    LinkNode<T>  \*current = Locate(i);

**if** (current == **NULL**)

**return**;

**else**  current->data = x;

}

**template** <**class** T>

**bool** List<T>::Insert(**int** i, T& x)

{

    LinkNode<T> \*current = Locate(i-1);

**if** (current == **NULL**) **return** **false**;

    LinkNode<T> \*newNode = **new** LinkNode<T>(x);

**if** (newNode == **NULL**)

    {

        cerr << "内存分配错误！" << endl;

        exit(1);

    }

    newNode->link = current->link;

    current->link = newNode;

**return** **true**;

}

**template** <**class** T>

**bool** List<T>::Remove(**int** i, T& x)

{

    LinkNode<T> \*current = Locate(i-1);

**if** (current == **NULL** || current->link == **NULL**)

**return** **false**;

    LinkNode<T> \*del = current->link;

    current->link = del->link;

    x = del->data;

**delete** del;

**return** **true**;

}

**template** <**class** T>

**void** List<T>::output()

{

    LinkNode<T>  \*current = first->link;

**while** (current != **NULL**)

    {

        cout << current->data <<endl;

        current = current->link;

    }

}

**template** <**class** T>

**void** List<T>::inputFront(T endTag)

{

    LinkNode<T> \*newNode;

    T val;

    makeEmpty();

    cin>>val;

**while** (val != endTag)

    {   newNode = **new** LinkNode<T>(val);

**if** ( newNode == **NULL** )

        {

            cerr << "内存分配错误！" << endl;

            exit(1);

        }

        newNode->link = first->link;      *//插在表前端*

        first->link = newNode;

        cin >> val;

    }

}

**template** <**class** T>

**void** List<T>::inputRear(T endTag)

{

    LinkNode<T> \*newNode, \*last;

    T val;

    makeEmpty();

    cin>>val;

    last = first;

**while** (val != endTag)

    {

        newNode = **new** LinkNode<T>(val);

**if** ( newNode == **NULL** )

        {

            cerr<<"内存分配错误！"<<endl;

            exit(1);

        }

        last->link = newNode;      *//插在表尾*

        last = newNode;

        cin >> val;

    }

}

**template**<**class** T>

**void** List<T>::Sort()

{

    LinkNode<T> \*head, \*pre, \*cur, \*next, \*end, \*temp;

    head = first;

    end = **NULL**;

**while**(head -> link != end)

    {

**for**(pre = head, cur = pre -> link, next = cur -> link;

            next != end;

            pre = pre -> link, cur = cur -> link, next = next -> link)

        {

**if**(cur -> data > next -> data)

            {

                cur -> link = next -> link;

                pre ->link = next;

                next -> link = cur;

                temp = next;

                next = cur;

                cur = temp;

            }

        }

        end = cur;

    }

    cout << "排序成功！" << endl;

}

**template**<**class** T>

List<T>&List<T>::**operator**=(List<T>& L)

{

    T value;

    LinkNode<T> \*srcptr = L.getHead();

    LinkNode<T> \*destptr = first = **new** LinkNode<T>;

**while** (srcptr->link != **NULL**)

    {

        value = srcptr->link->data;

        destptr->link = **new** LinkNode<T>(value);

        destptr = destptr->link;

        srcptr = srcptr->link;

    }

    destptr->link = **NULL**;

**return** \***this**;

}

**int** main()

{

**int** d,e,f;

List<**int**> L1;

cout << "初始化链表开始（使用后插法），输入4时结束插入" << endl;

    L1.inputRear(4);

    L1.output();

    L1.Remove(2, d);

    cout << "被删除的数字为：" << d << endl;

    L1.output();

    cout << "请输入想要插入的数字：";

    cin >> e;

    cout << "请输入想要插入的数字的位置(不能大于" << L1.Length()+1 << "）：";

    cin >> f;

    L1.Insert(f, e);

    L1.output();

    cout << "请输入想要查询的数值：";

    cin >> d;

    cout << d << "所在的位置为：" << L1.Search(d) << endl;

    cout << "请输入想要确认数值的位置：";

    cin >> d;

    cout << d << "处所存储的数字为：" << L1.Locate(d)->data << endl;

    L1.getData(d, e);

    cout << d << "处所存储的数字为（使用getData方法）：" << e << endl;

    cout << "请输入想要替换的数字：";

    cin >> e;

    cout << "请输入想要替换的数字的位置(不能大于" << L1.Length() << "）：";

    cin >> f;

    L1.setData(f, e);

    L1.output();

    List<**int**> L2;

    L2 = L1;

    L2.Sort();

    L2.output();

**return** 0;

}

**【实验过程】**

1. 打开编译器后新建一个空白文件

2.将课本上所学的带附加表头的单链表代码输入到编程窗口中

3.对输入的代码进行调试至其能编译通过

4.编写主函数对单链表的使用进行程序实现

5.补充未完成的Sort函数以及实现前插与后插算法

6.利用单链表各函数的功能对单链表进行操作



**【实验体会】**

在进行实验前一定要多读几遍课本，不仅要了解各个顺序表的各函数的功能，更要深入理解其内涵与算法理由，便于实验的顺利进行。在输入代码时，一定要事无巨细，输入不正确也会影响运行。一定要将编写的各个函数都在主函数里至少使用一次，这样可以使代码可信度更高。在编写排序函数时，更要考虑到算法的时间复杂度和空间复杂度。

**字符串类**

**【实验简介】**

学会用算法语言C++描述抽象数据类型，使用模板建立字符串类数据结构。掌握用C++描述字符串数据结构的基本方法，即通过建立类来描述抽象数据类型。理解字符串运算的原理，掌握主要算法的实现。本实验主要内容是要求编写并调试朴素的模式匹配。

**【实验内容】**

实现字符串模式匹配：给定任意一个长度为a的字符串A和一个长度为b的字符串B（a>b），串A为目标（Target），串B为模式（pattern），要求在字符串A中查找（匹配）是否有与字符串B相等的子串。如果匹配成功，返回B在A中的起始位置，否则匹配不成功，返回-1。

**【程序代码】**

头文件str.h：

#ifndef str\_h

#define str\_h

#include <iostream>

#include <string.h>

#include <stdlib.h>

**using** **namespace** std;

**const** **int** maxLen = 128;        *//字符串存储空间的最大长度*

**class** String {

*//对象：长度不小于0的字符有限序列*

**public**:

    String ( **const** String & sb );  *//构造函数，复制已有的串对象sb来创建一个新串*

    String ( **const** **char** \*ch\_arr );

*//构造函数，从已有的字符数组\*ch\_arr复制字符来创建一个新串*

    String ();                    *//构造函数，创建一个空串*

    ~String () { **delete** []ch; }  *//析构函数，释放动态分配的串空间*

**int** Length () **const** { **return** slen; } *//返回字符串的长度*

    String &**operator** () (**int** pos, **int** len);  *//重载()运算符，从串中pos位置起连续提取 len 个字符形成子串返回*

**int** **operator** == ( **const** String &sb )

**const** { **return** strcmp (ch, sb.ch) == 0; } *//重载==运算符，判断串是否相等*

**int** **operator** != ( **const** String &sb )

**const** { **return** strcmp (ch, sb.ch) != 0; } *//重载!=运算符，判断串是否不相等*

**int** **operator** ! ( ) **const** { **return** slen == 0; }  *//重载!( )运算符，判断是否为空串*

    String &**operator** = ( **const** String &sb );    *//串赋值，从另一串sb复制而得*

    String &**operator** += ( **const** String &sb );  *//串连接，将串sb接到\*this串后面*

**char** &**operator** [ ] ( **int** i );               *//取串中第i个字符*

**friend** ostream &**operator** << (ostream &**out**,String &P); *//输出字符串*

**int** Search(String Target, String pattern);

**private**:

**char** \*ch;           *//串的存储数组*

**int** slen;           *//串的字符个数(串长度)*

**int** \*f;             *//串的失效函数值*

};

#endif */\* str\_h \*/*

源文件main.cpp：

#include "str.h"

String::String (**const** String &sb)    *//复制构造函数：从已有串sb复制*

{

    ch = **new** **char**[maxLen+1];  *//字符串存储空间*

    f = **new** **int**[maxLen+1];    *//串的失效函数值存储空间*

**if** ( !ch ) {

        cerr << "存储分配错 \n";

        exit(1);

    }

    slen = sb.slen;

    strcpy ( ch, sb.ch );

}

String::String (**const** **char** \*ch\_arr)

{

*//复制构造函数: 从已有字符数组\*ch\_arr复制*

    ch = **new** **char**[maxLen+1];

    f = **new** **int**[maxLen+1];

**if** ( !ch ){

        cerr << "存储分配错 \n";

        exit(1);

    }

    slen = (**int**)strlen (ch\_arr);

    strcpy ( ch, ch\_arr );

}

String::String()

{

*//构造函数：创建一个空串*

    ch = **new** **char**[maxLen+1];

    f = **new** **int**[maxLen+1];

**if** ( !ch ) {

        cerr << "存储分配错\n";

        exit(1);

    }

    slen = 0;

    ch[0] = '\0';

}

String &String:: **operator** () (**int** pos, **int** len)

{

*//从串中第 pos 个位置起连续提取 len 个字符作为子串返回*

    String \*temp = **new** String;   *//动态分配*

**if** ( pos < 0 || pos+len-1 >= maxLen || len < 0 )

    {

        temp->slen = 0;           *//子串长度为0，返回空串*

        temp->ch[0] = '\0';        *//添加串结束符*

    }

**else**

    {                    *//提取子串*

**if** (pos+len -1 >= slen)

            len = slen - pos;

        temp->slen = len;*//子串长度*

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

            temp->ch[i] = ch[j];      *//传送串数组*

        temp->ch[len] = '\0';*//子串结束，添加串结束符*

    }

**return** \*temp;

}

String &String::**operator** = (**const** String &sb)

{

*//串赋值：从已有串sb复制*

**if** ( &sb != **this** ) {

**delete** [ ] ch;

        ch = **new** **char** [maxLen+1];  *//重新分配*

**if** ( ! ch ) {

            cerr << "内存不足!\n ";  exit (1);

        }

        slen = sb.slen;              *//串复制*

        strcpy ( ch, sb.ch );

    }

**else**  cout << "字符串自身赋值出错! \n";

**return** \***this**;

}

String &String::**operator** += (**const** String &sb)

{

*//串连接*

**char** \* temp =ch;               *//暂存原串数组*

    slen += sb.slen;       *//串长度累加*

    ch = **new** **char** [maxLen+1];

**if** ( ! ch ) {

        cerr << "字符串下标超界!\n";  exit (1) ;

    }

    strcpy (ch, temp);          *//拷贝原串数组*

    strcat (ch, sb.ch);          *//连接sb串数组*

**delete** []temp;

**return** \***this**;

}

**int** String::Search(String Target, String pattern)

{

**for**(**int** pos = 0; pos <= Target.Length(); pos++)

**if**(pattern == Target(pos,pattern.Length()))

**return** pos;

**return** -1;

}

ostream &**operator** << (ostream &out,String &P){ *//打印字符串*

**for**(**int** i=0;i<P.slen;i++)

        cout<<P.ch[i];

**return** out;

}

**int** main()

{

**char** a[30] = "aaaaabcLetabcaabcdde"; *//数组存储的字符串*

**char** b[6] = "aabcd";

**char** c[10] = "abcLetabc";

    String A(a);              *//定义为类对象的字符串*

    String B(b);

    String C(c);

    cout << "A串：" << A << endl;

    cout << "B串：" << B << endl;

    A += B;

    cout << A << endl;

    C = B;

    cout << C << "的长度是" << C.Length() << endl;

    cout << C.Search(A, B) << endl;

**return** 0;

}

**【实验过程】**

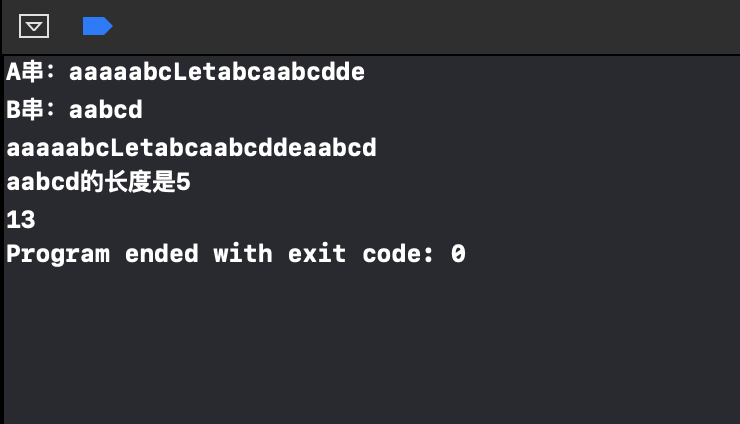
1.新建工程文件

2.将实验指导书内的代码复制粘贴

3.调试至可成功运行

4.以A为目标，B为对象，在主函数中实现字符串模式匹配，并返回结果

5.测试其他函数的可用性



**【实验体会】**

本次试验要求实现模式匹配，在主函数内完成的时候有两种方法，一是自己定义该种类型的函数在ADT中，方便以后的再次使用。二是写出一次性代码，只为实现A为目标，B为对象情况下返回的值。按顺序表数据结构定义字符串类，其数据成员中的一维数组ch为字符串存储空间。需要提供的主要操作有求字符串的长度，提取子串，判断两个串是否相等、是否不相等，字符串是否为空串，字符串复制，字符串连接，求串的失效函数值，模式匹配等。对我们的要求很高，但不管黑猫白猫能抓住耗子的就是好猫。

**二叉树的操作**

**【实验简介】**

建立链表结构的二叉树，由类BinaryTree定义。二叉树结点由类BinTreeNode定义。为了降低模板类的使用难度和缩小程序规模，算法没有使用栈和队列的模板类，只是在二叉树类成员中定义了两个数组作为栈和队列来使用（不保证栈和队列功能上的完整性）。

**【实验内容】**

理解二叉树的结构和性质，掌握建立二叉树和遍历二叉树的算法实现，能够用C++程序实现操作。对于输入的n个元素数据，建立一棵二叉树，并提供遍历和输出等操作。

**【程序代码】**

头文件BinaryTree.h：

#ifndef BinaryTree\_h

#define BinaryTree\_h

#include <iostream>

**using** **namespace** std;

**template** <**class** Type>

**class** BinaryTree;    *//二叉树类前视声明*

**template** <**class** Type>   *//二叉树结点类*

**class** BinTreeNode

{

**friend** **class** BinaryTree<Type>;

**private**:

    BinTreeNode<Type> \*leftChild, \*rightChild;

    Type data;

**public**:

    BinTreeNode () : leftChild (**NULL**), rightChild (**NULL**){}

    BinTreeNode (Type item, BinTreeNode<Type> \*left = **NULL**,BinTreeNode<Type> \*right = **NULL**)

        : data (item), leftChild (left), rightChild (right){}

    Type GetData () **const** { **return** data; }

};

**template** <**class** Type>

**class** BinaryTree

{

**public**:

    BinaryTree () : root(**NULL**){}

*//BinaryTree (Type value) : RefValue (value), root(NULL){}*

**virtual** ~BinaryTree ( ) { destroy ( root ); }*//删除二叉树*

**void** create(Type \*array);   *//建立二叉树*

**void** InordPrint();          *//中序遍历输出二叉树*

**void** pttree(){**if**(root!=**NULL**) printree(root); cout<<endl;}*//输出括号对形式的二叉树*

**void** PordPrint(){**if**(root!=**NULL**) Postorder(root);cout<<endl;}*//后序遍历输出二叉树*

**friend** ostream& **operator** << (ostream& **out**, BinaryTree<Type>& Tree)*//先序遍历输出二叉树*

    {

*// "<<"运算符重载函数*

**out** << "二叉树的前序遍历序列:\n";

        Tree.Traverse(Tree.root, **out**); *//调用私有成员函数*

**out** << endl;

**return** **out**;

    }

**private**:

    BinTreeNode <Type> \*root;   *//二叉树根结点*

    BinTreeNode <Type> \* st[30];  *//数组表示的栈，用于非递归中序遍历*

**int** top;                     *//栈顶指针*

    BinTreeNode <Type> \*qu[20];  *//数组表示的队列，用于建立二叉树*

**int** qi;                      *//队列的队首指针*

**void** Traverse (BinTreeNode<Type> \*node, ostream& **out**);              *//递归执行先序遍历*

**void** destroy(BinTreeNode<Type> \*node);   *//递归执行删除二叉树*

**void** printree(BinTreeNode<Type> \*t)**const**;   *//递归执行输出二叉树*

**void** Postorder(BinTreeNode<Type> \*t)**const**;  *//递归执行后序遍历*

};

#endif */\* BinaryTree\_h \*/*

源文件main.cpp：

#include "BinaryTree.h"

**template**<**class** Type>

**void** BinaryTree<Type>::destroy (BinTreeNode<Type> \*node)

{*//递归删除二叉树*

**if** ( node != **NULL** ) {

        destroy ( node->leftChild );

        destroy ( node->rightChild );

**delete** node;

    }

}

**template** <**class** Type>

**void** BinaryTree<Type>::Traverse (BinTreeNode<Type> \*node, ostream &out)

{

*//先序遍历并输出以node为根的二叉树*

**if** ( node != **NULL** ) {

        out << node->data <<' ';

        Traverse ( node->leftChild, out );

        Traverse ( node->rightChild, out );

    }

}

**template** <**class** Type>

**void** BinaryTree<Type>::create(Type \*nodevalue)

{

*//参数为存储了结点数据的数组，使用队列按层建立二叉树*

    BinTreeNode <Type> \*t;     *//用于建立二叉树结点*

**int** ai = 0;                  *//取数组的下标*

    t=**new** BinTreeNode<Type>( ); *//生成根结点*

    t->data = nodevalue[0];       *//为根结点赋值*

    t->leftChild = **NULL**;         *//新结点左孩子初值为空*

    t->rightChild = **NULL**;        *//新结点右孩子初值为空*

    root = t;   *//建立根结点*

    qi=0;     *//队列的队首指针*

    qu[qi] = t; *//根结点入队列*

    ai=1;     *//数组下标进1*

**while** (nodevalue[ai]!='#') { *//输入数据未结束*

**if**(nodevalue[ai]==' ')   *//下标为奇数且数据为空格*

            qu[ai]=**NULL**;         *//入队一个空结点（该结点未建立）*

**else**{

            t=**new** BinTreeNode<Type>( ); *//生成新结点*

            t->data=nodevalue[ai];       *//新结点赋值*

            t->leftChild=**NULL**;

            t->rightChild=**NULL**;

            qu[qi]->leftChild=t;    *//新结点成为双亲的左孩子*

            qu[2\*qi+1]=t;        *//新结点入队，2\*pi+1为队尾*

        }

        ai++;                    *//下标为偶数，取下一个数据*

**if**(nodevalue[ai]=='#')**break**;  *//输入数据结束*

**if**(nodevalue[ai]==' ')qu[ai]=**NULL**; *//遇空数据入队一个空结点*

**else**{

            t=**new** BinTreeNode<Type>( ); *//生成新结点*

            t->data=nodevalue[ai];       *//新结点赋值*

            t->leftChild=**NULL**;

            t->rightChild=**NULL**;

            qu[qi]->rightChild=t;  *//新结点成为双亲的右孩子*

            qu[2\*qi+2]=t;        *//新结点入队，2\*pi+1为队尾*

        }

        ai++;                   *//数组下标增1*

        qi++;                   *//队列下标增1*

    }*//end of while*

    cout << "建立二叉树已经完成" << endl;

}

**template** <**class** Type>

**void** BinaryTree<Type>::InordPrint()

{           *//中序遍历并输出以root为根的二叉树*

    top=-1;             *//栈顶指针*

    BinTreeNode <Type> \*p;

**if**(root==**NULL**) **return** ;

    p=root;           *//从根结点开始遍历*

**while**(p||top>=0){ *//若当前结点或栈不为空则遍历未完成*

**if**(p){

            st[++ top] = p;  *//进栈*

            p=p->leftChild;  *//向左走，遍历左子树*

        }

**else**{

            p = st[top--];       *//退至上一层*

            cout<<p->data<<' ';  *//输出出栈结点的数据*

            p=p->rightChild;    *//向右走*

        }

    }*//while*

    cout<<endl;

}

**template** <**class** Type>

**void** BinaryTree<Type>::printree(BinTreeNode<Type> \*t)**const**

{*//私有成员函数，输出括号对形式的二叉树*

**if**(t!=**NULL**)

    {  cout<<t->data;

**if**(t->leftChild!=**NULL** || t->rightChild!=**NULL**)

        {  cout<<'(';                    *//非叶结点时准备输出子树*

            printree(t->leftChild);             *//左孩子为参数递归执行*

**if**(t->rightChild!=**NULL**) cout<<',';  *//若当前结点有右孩子则输出逗号*

            printree(t->rightChild);           *//右孩子为参数递归执行*

            cout<<')';                      *//子树输出完毕*

        }

    }

}

**template** <**class** Type>

**void** BinaryTree<Type>::Postorder(BinTreeNode <Type> \*t)**const**

{*//私有成员函数，后序遍历并输出以t为根的二叉树*

**if** ( t != **NULL** ) {

        Postorder( t->leftChild);

        Postorder( t->rightChild);

        cout<<t->data<<' ';

    }

}

**int** main()

{

    BinaryTree <**char**> mybt; *//定义一个二叉树对象*

**char** data[20]={'a', 'b', 'c', ' ', 'e', 'f', 'g', ' ', ' ', 'h', 'i', 'j', '#'};

    mybt.create(data);  *//二叉树结点数据由数组data提供，建立链表存储的二叉树*

    mybt.pttree();     *//输出括号对形式的二叉树*

    cout<<endl;

    cout<<"非递归中序遍历并输出二叉树:"<<endl;

    mybt.InordPrint();  *//公有成员函数非递归输出二叉树的中序遍历序列*

    cout<<endl;

    cout<<"先序遍历并输出二叉树:"<<endl;

    cout<<mybt;        *//输出运算符重载方式递归输出二叉树的先序遍历序列*

    cout<<endl;

    cout<<"后序遍历并输出二叉树:"<<endl;

    mybt.PordPrint();    *//两层函数调用方式递归输出二叉树的后序遍历序列*

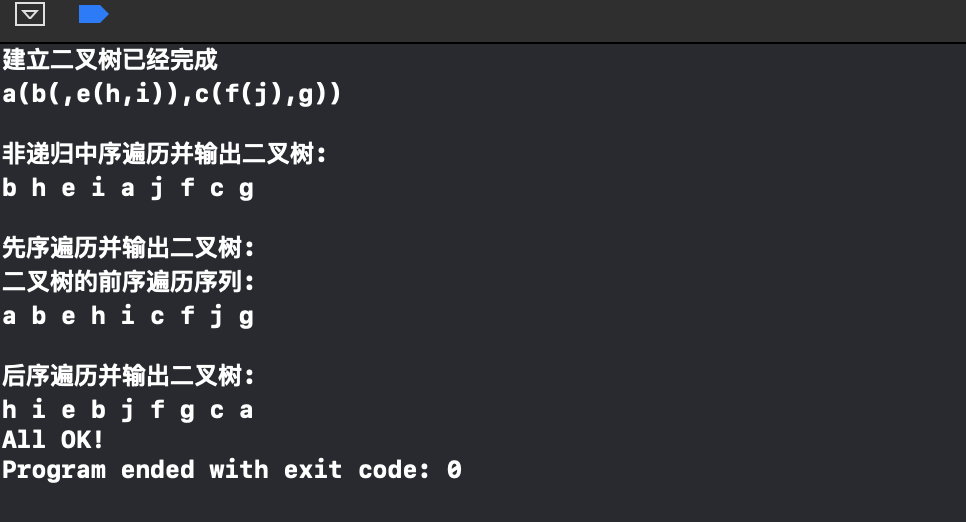
    cout<<"All OK!"<<endl;

**return** 0;

}

**【实验过程】**

1. 建立新的工程文件
2. 对实验指导书中的代码进行调试
3. 注释掉了二叉树声明里的第二个复制构造函数，由于其中的RefValue不知所以
4. 发现 << 无法重载，将源文件中的函数实现代码移动至头文件相应位置
5. 运行成功，成功输出二叉树三种遍历形式



**【实验体会】**

这次实验的过程中我不仅能完成实验内容，还能复习考试内容，对于二叉树内的各种实现又有了新的认识。在调试的过程中，不仅最后能使程序成功运行，而且能对ADT内的每个函数进行深入了解。实验宝贵又深刻，最重要的就是自己动手完成，这份时间和经历弥足珍贵。