

华中科技大学

课程实验报告

课程名称：面向对象程序设计

实验名称：整型栈及队列综合实验

院 系：计算机科学与技术

专业班级：CS1409

学 号：U201414808

姓 名：王林

指导教师：辜希武

联系方式：13419606224

2016 年 11 月 22 日

目录

目录.....	- 1 -
C++面向对象查询设计实验	- 2 -
一、需求分析.....	- 2 -
1、题目要求	- 2 -
2、需求分析	- 4 -
二、系统设计.....	- 4 -
1、概要设计	- 4 -
2、详细设计	- 9 -
三、软件开发.....	- 0 -
1、软件测试	- 1 -
四、特点与不足	- 8 -
1.技术特点.....	- 8 -
2.不足和改进的建议	- 8 -
五、过程和体会	- 8 -
1.遇到的主要问题和解决方法.....	- 8 -
2.课程设计的体会	- 8 -
六、源码和说明	- 9 -
1.文件清单及其功能说明	- 9 -
2. 用户使用说明书	- 9 -
3.源代码	- 9 -

C++面向对象查询设计实验

一、需求分析

1、题目要求

整型栈是一种先进后出的存储结构，对其进行的操作通常包括判断栈是否为空、向栈顶添加一个整型元素、将栈顶元素出栈等操作。以下各栈均为整型栈类型，请先用面向过程的方法编程，然后再使用面向对象的方法编程，并采用运算符重载和虚函数实现整型栈。

整型队列是一种先进先出的存储结构，对其进行的操作通常包括判断队列是否为空、向队列顶添加一个整型元素、出队列等。请分别使用“对象聚合”和“对象继承”两种方法，编程实现两个整型队列类的所有函数成员。

```
//=====面向过程的 STACK=====//
struct POSTK{
    int  *elems;           //申请内存用于存放栈的元素
    int   max;             //栈能存放的最大元素个数
    int   pos;             //栈实际已有元素个数，栈空时 pos=0;
};
void initPOSTK(POSTK *const p, int m);           //初始化 p 指的栈：最多 m 个元素
void initPOSTK(POSTK *const p, const POSTK&s); //用栈 s 初始化 p 指的栈
int  size (const POSTK *const p);               //返回 p 指栈的最大元素个数 max
int  howMany (const POSTK *const p);            //返回 p 指栈的实际元素个数 pos
int  getelem (const POSTK *const p, int x); //取下标 x 处的栈元素
POSTK *const push(POSTK *const p, int e);      //将 e 入栈，并返回 p 值
POSTK *const pop(POSTK *const p, int &e);      //出栈到 e，并返回 p 值
POSTK *const assign(POSTK*const p, const POSTK&s); //赋 s 给 p 指的栈，返 p 值
void print(const POSTK*const p);                //打印 p 指向的栈
void destroyPOSTK(POSTK*const p);              //销毁 p 指向的栈
//=====面向对象的 STACK=====//
class OOSTK{
    int  *const  elems;           //申请内存用于存放栈的元素
    const int    max;             //栈能存放的最大元素个数
    int    pos;                   //栈实际已有元素个数，栈空时 pos=0;
public:
    OOSTK(int m);                 //初始化栈：最多 m 个元素
    OOSTK(const OOSTK&s); //用栈 s 拷贝初始化栈
    int  size ( ) const;          //返回栈的最大元素个数 max
    int  howMany ( ) const; //返回栈的实际元素个数 pos
    int  getelem (int x) const;   //取下标 x 处的栈元素
    OOSTK& push(int e);           //将 e 入栈，并返回当前栈
    OOSTK& pop(int &e); //出栈到 e，并返回当前栈
    OOSTK& assign(const OOSTK&s); //赋 s 给栈，并返回被赋值的当前栈
    void print( ) const;          //打印栈
    ~OOSTK( );                   //销毁栈
};
```

```
//=====运算符重载面向对象的 STACK=====//
class STACK{
    int *const elems;        //申请内存用于存放栈的元素
    const int max;           //栈能存放的最大元素个数
    int pos;                 //栈实际已有元素个数，栈空时 pos=0;
public:
    STACK(int m);            //初始化栈：最多 m 个元素
    STACK(const STACK&s);    //用栈 s 拷贝初始化栈
    virtual int size ( ) const; //返回栈的最大元素个数 max
    virtual operator int ( ) const; //返回栈的实际元素个数 pos
    virtual int operator[ ] (int x) const; //取下标 x 处的栈元素
    virtual STACK& operator<<(int e); //将 e 入栈，并返回当前栈
    virtual STACK& operator>>(int &e); //出栈到 e，并返回当前栈
    virtual STACK& operator=(const STACK&s); //赋 s 给当前栈并返回该栈
    virtual void print( ) const; //打印栈
    virtual ~STACK( ); //销毁栈
};

//=====由 2 个 STACK 组成的队列=====//
class QUE2S{
    STACK s1, s2;           //一个队列可由两个栈聚合而成
public:
    QUE2S(int m);           //初始化队列：每栈最多 m 个元素
    QUE2S(const QUE2S &q); //用队列 q 拷贝构造新队列
    operator int ( ) const; //返回队列的实际元素个数
    QUE2S& operator<<(int e); //将 e 入队列，并返回当前队列
    QUE2S& operator>>(int &e); //出队列到 e，并返回当前队列
    QUE2S& operator=(const QUE2S &q); //赋 q 给当前队列并返回该队列
    void print( ) const; //打印队列
    ~QUE2S( ); //销毁队列
};

//=====从 STACK 继承的队列=====//
class QUEIS: public STACK{ //STACK 作为构成队列的第 1 个栈
    STACK s;              //s 作为构成队列的第 2 个栈
public:
    QUEIS(int m);          //初始化队列：每栈最多 m 个元素
    QUEIS(const QUEIS &q); //用队列 q 拷贝初始化队列
    virtual operator int ( ) const; //返回队列的实际元素个数
    virtual QUEIS& operator<<(int e); //将 e 入队列，并返回当前队列
    virtual QUEIS& operator>>(int &e); //出队列到 e，并返回当前队列
    virtual QUEIS& operator=(const QUEIS &q); //赋 q 给队列并返回该队列
    virtual void print( ) const; //打印队列
    virtual ~QUEIS( ); //销毁队列
};

然后写一个 main 函数分别对各种栈和队列的所有操作函数进行测试。
```

2、需求分析

五次实验，1、实现面向过程实验设计，完成栈的相关操作；2、面向对象程序设计，完成栈的相关操作；3、运算符重载实验，面向对象，完成运算符重载的相关函数；4、类的聚合，用栈实现队列，面向对象程序设计；5、类的继承、用栈实现队列，面向对象程序设计。

二、系统设计

1、概要设计

整型栈是一种先进后出的存储结构，对其进行的操作通常包括判断栈是否为空、向栈顶添加一个整型元素、将栈顶元素出栈等操作。以下各栈均为整型栈类型，请先用面向过程的方法编程，然后再使用面向对象的方法编程，并采用运算符重载和虚函数实现整型栈。

系统分为5个模块，分别实现面向过程的STACK、面向对象的STACK、运算符重载面向对象的STACK、由两个STACK组成的队列、从STACK继承的队列这5个功能。各模块的功能设计如下所示：

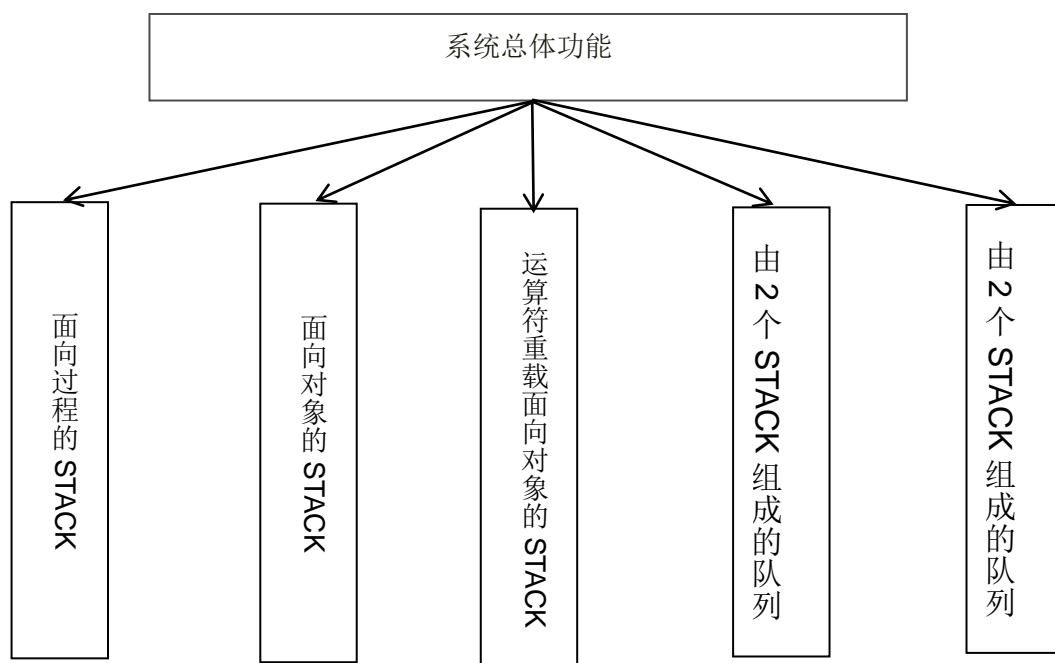


图 2.1 系统体设计图

1) 面向过程的 STACK

采用面向过程程序设计方法，用C语言实现整形栈对整形栈进行的操作通常包括初始化栈(两种方式)、计算当前栈的容量、计算当前栈的元素个数、查询某个位置的元素、向栈顶添加一个整差点型元素、将栈顶元素出栈、用其他栈给当前栈赋值等操作。模块设计调用图如下所示：

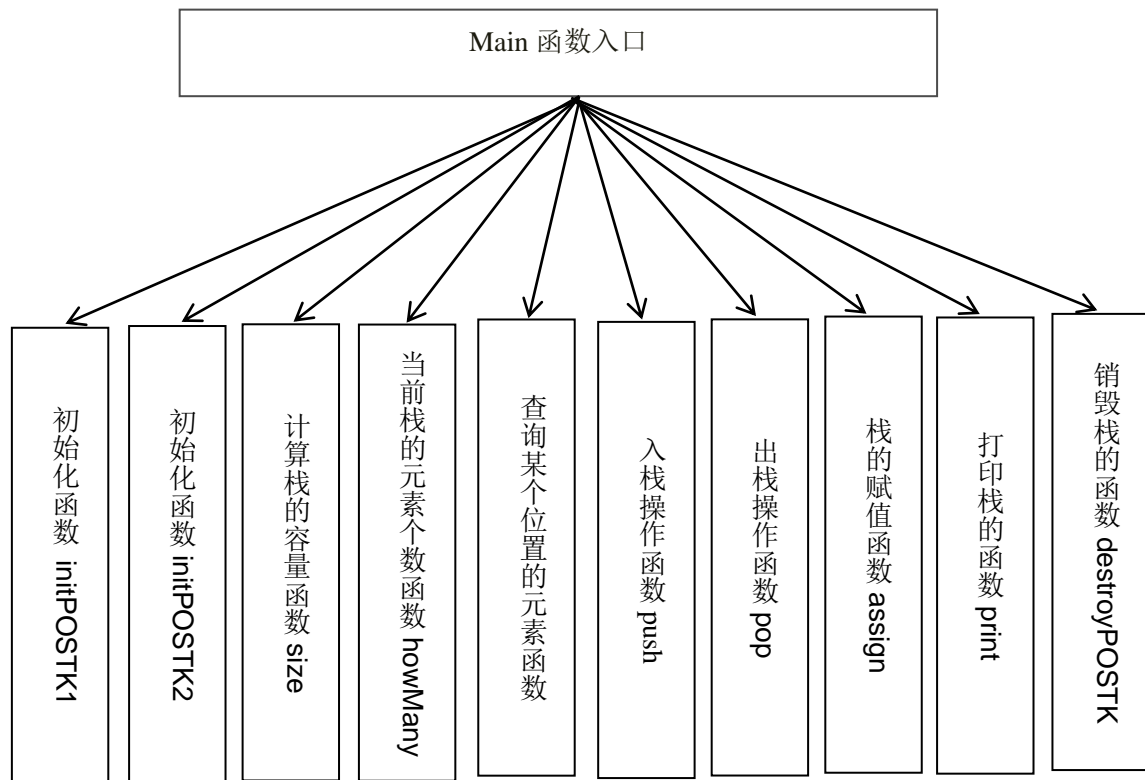


图 2.2 面向过程函数调用图

图中箭头表示调用关系

2) 面向对象的 STACK

采用面向对象程序设计方法，用 C++ 语言实现整型栈对整形栈进行的操作通常包括初始化栈（两种方式）、计算当前栈的容量、计算当前栈的元素个数、查询某个位置的元素、向栈顶添加一个整型元素、将栈顶元素出栈、用其他栈给当前栈赋值等操作。模块设计调用图如下所示：

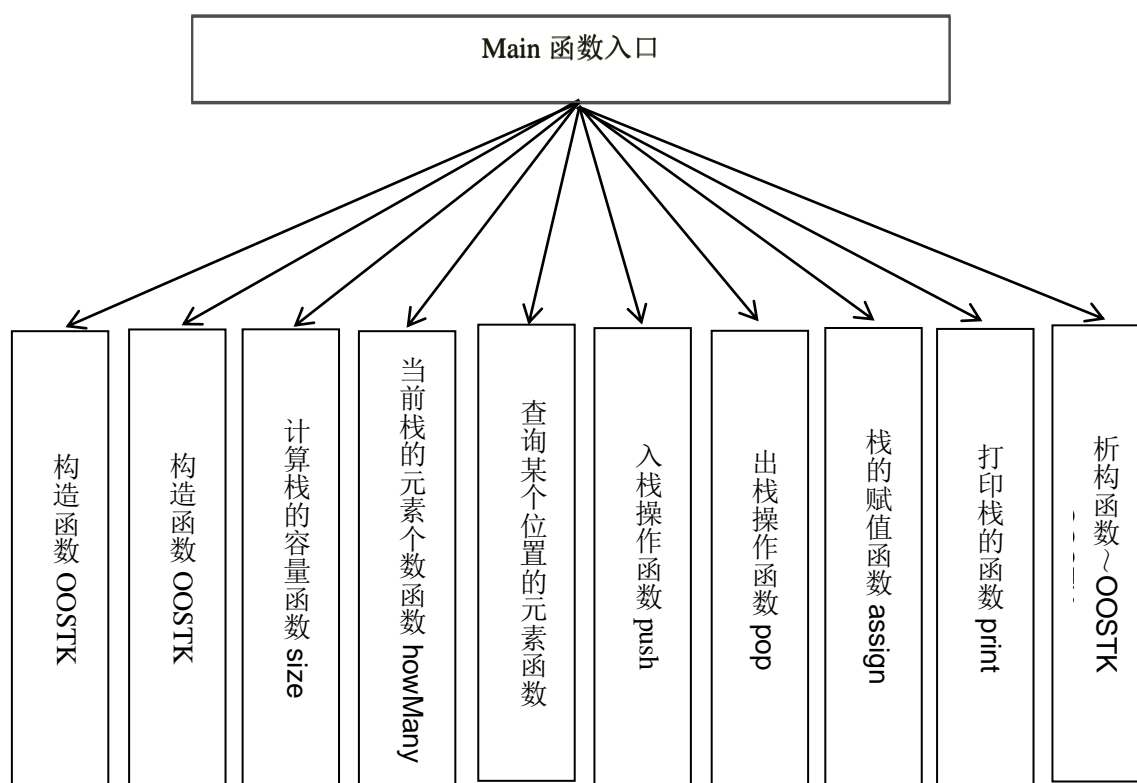


图 2.3 面向对象函数调用图

图中箭头表示调用关系

3) 运算符重载面向对象的 STACK

采用面向对象程序设计方法，用运算符重载和虚函数实现整型栈。对整形栈进行的操作通常包括初始化栈（两种方式）、计算当前栈的容量、计算当前栈的元素个数、查询某个位置的元素、向栈顶添加一个整型元素、将栈顶元素出栈、用其他栈给当前栈赋值等操作。模块设计调用图如下所示：

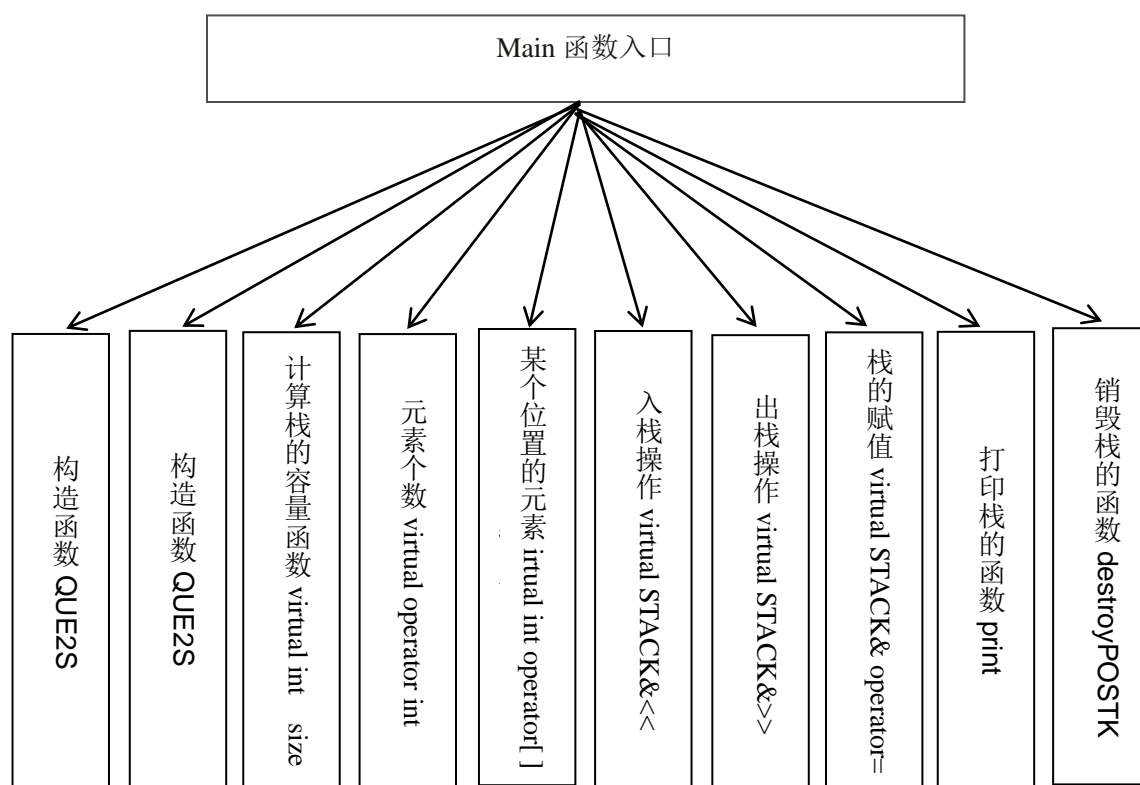


图 2.4 运算符重载函数调用图

图中箭头表示调用关系

4) 由 2 个 STACK 组成的队列

整型队列是一种先进先出的存储结构，用对象聚合的方法，通过两个栈来实现整型队列。对其进行的操作通常包括初始化队列（两种方式）、计算当前队列容量、计算当前队列元素个数、查询某个位置元素、向队列顶添加一个整型元素、出队列、用其他队列对当前队列赋值操作等。模块设计调用图如下所示：

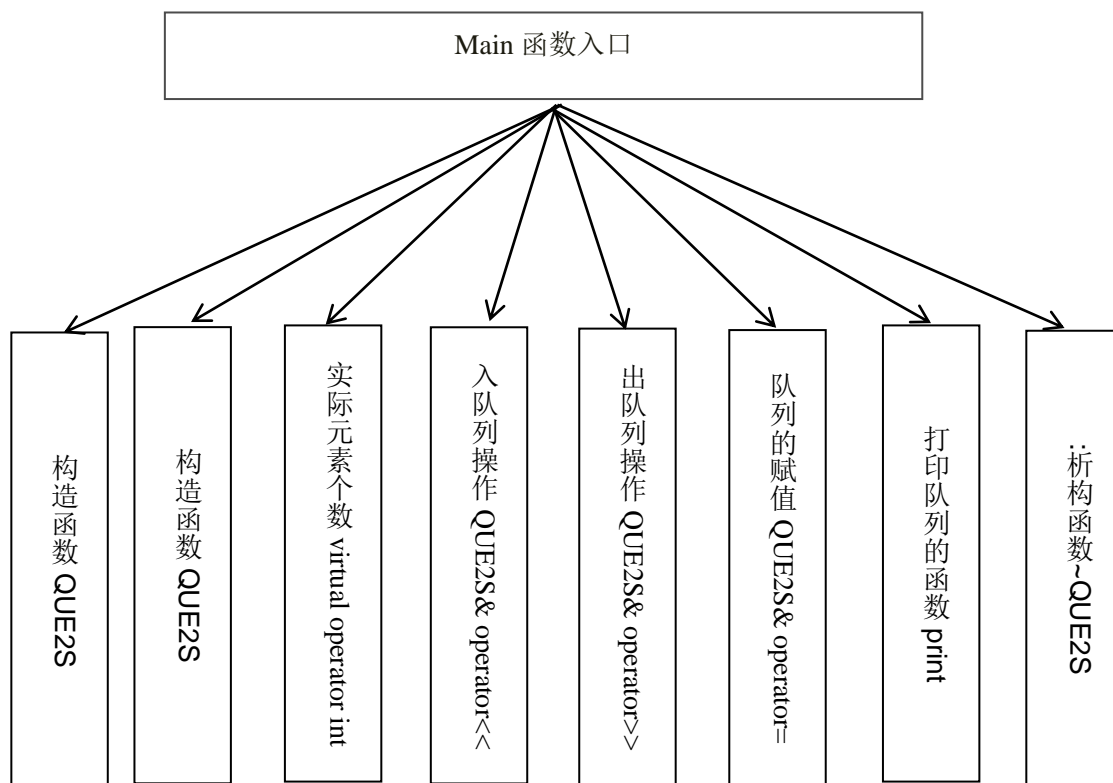


图 2.5 两个栈实现队列函数调用图

图中箭头表示调用关系

5) 从 STACK 继承的队列

采用面向对象程序设计方法，用 C++ 语言对整形栈进行的操作通常包括初始化栈（两种方式）、计算当前栈的容量、计算当前栈的元素个数、查询某个位置的元素、向栈顶添加一个整型元素、将栈顶元素出栈、用其他栈给当前栈赋值等操作。模块设计调用图如下所示：

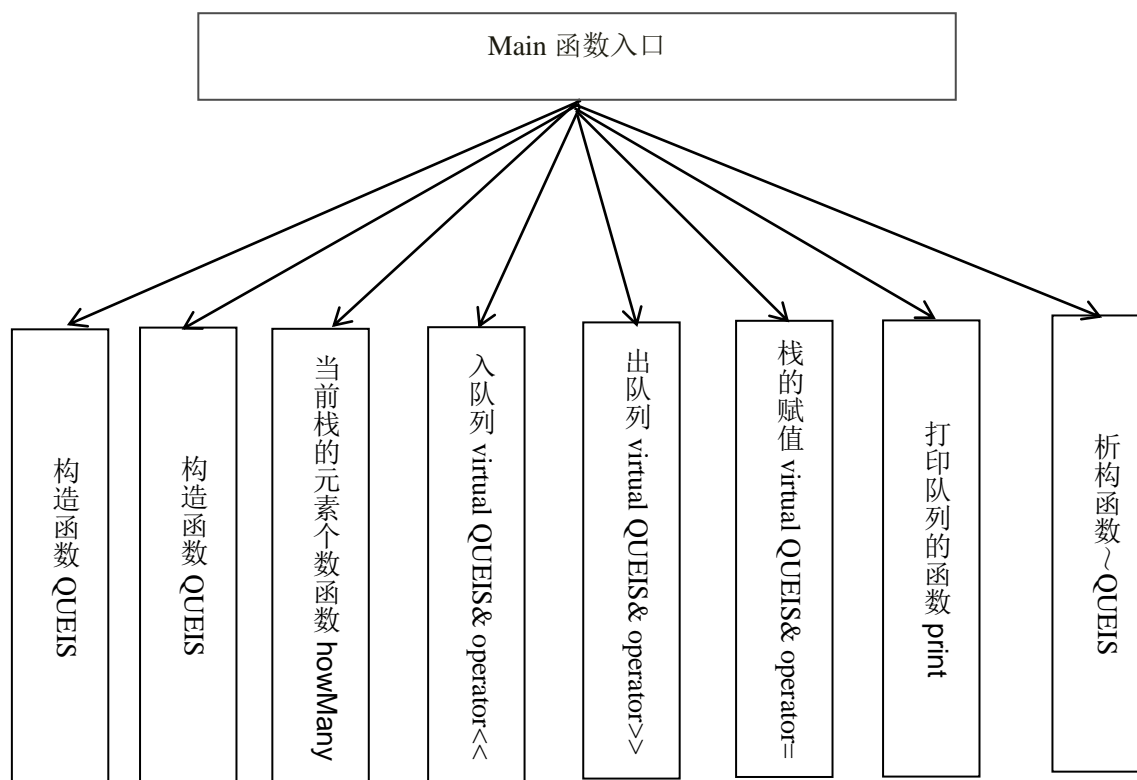


图 2.6 继承实现队列函数调用图

图中箭头表示调用关系

2、详细设计

设计每个模块的实现算法（处理流程）、所需的局部数据结构。具体介绍每个模块/子程序的功能、入口参数、出口参数、流程（图）等。

1) 面向过程的 STACK

栈的结构为：

```
struct POSTK{
    int *elems;    //store the elements
    int max;       //the max num
    int pos;       //the number of the actual elements
};
```

函数设计：

1.初始化函数：initPOSTK

返回值为空；

初始化函数参数为栈指针 p，和栈的容量 m，通过 initPOSTK(P,m)调用来初始化一个栈。

然后重载这个初始化函数，参数是栈指针 p，栈 s 的引用，通过 initPOSTK(P,&s)调用来初始化一个栈。

2.计算栈的容量函数：size

返回值为 int

函数参数是栈指针 p，通过 size（p）调用返回一个整形数据，即当前栈的容量。

3.计算当前栈的元素个数函数:howMany

返回值为 int

函数参数是栈指针 p, 通过 haoMany(p)调用返回一个整形数据, 即当前栈的元素个数。

4. 查询某个位子的元素: getelem,

返回值为 int

函数参数是栈指针 p, 位置用一个整形数据表示, 通过 getelem(p, x)调用返回栈中处于 x 位置的数据。

5. 入栈函数: push

返回值为栈;

函数参数为栈指针以及要入栈的元素 e, 通过 push(p,e)调用来将元素 e 入栈, 并返回当前栈。

6. 出栈函数: pop

返回值为栈

函数参数为栈指针以及一个变量引用&e, 通过 pop(p,&e)调用来进行出栈操作, 将出栈的元素的值赋值给 e, 并返回当前栈。

7. 给栈赋值函数: assign

返回值为栈

函数参数是栈指针以及一个栈的引用&s, 通过 assign(p,&s)调用来用栈 s 初始化 p 指向的栈, 并返回当前栈。

8. 打印当前栈函数: print

返回值为空

函数参数为栈指针, 通过 print(p),调用, 打印当前栈的所有元素

9. 销毁栈函数: destroyPOSTK

返回值为空

函数参数为(栈指针), 通过 destroyPOSTK(p)来销毁 p 指向的栈。

2) 面向对象的 STACK

定义一个类, 数据成员如下:

```
int *const elems;      //申请内存用于存放栈的元素
const int max;         //栈能存放的最大元素个数
int pos;
```

函数成员定义如下:

1. 构造函数: OOSTK

构造函数无返回值

初始化函数参数为栈的容量 m, 通过 OOSTK S (m) 调用来构造一个栈。

然后重载这个初始化函数, 参数是栈 s 的引用, 通过 OOSTK m (&S) 调用来构造一个栈。

2. 计算栈的容量函数: size

返回值为 int

函数没有参数, 栈 s 通过 s.size()调用 size 函数, 得到当前栈的容量

3. 计算栈的当前语速个数函数: howMany

返回值为 int

没有参数, 栈 s 通过 s.howMany()调用 howany 函数, 得到当前栈的元素个数

4. 查询某个位置的元素函数: getelem

返回值为 int

函数参数: 位置变量 x, 整型, 栈 s 通过 s.getelem(x)调用的到栈 s 中 x 处的元素

5. 入栈操作函数: push

返回值为栈的引用

参数为要入栈的元素 e, 整型, 栈 s 通过 s.push(e)调用来将元素 e 插入到栈 s 中

6. 出栈操作函数: pop

返回值为栈的引用

参数为变量 `e` 的引用，整型，栈 `s` 通过 `s.pop(e)` 调用来将元素栈 `s` 最顶部的元素出栈，并赋值给 `e`

7. 给栈赋值函数: `assign`

这里的赋值要通过深拷贝实现

返回值为栈的引用

参数为另一个栈的引用，栈 `s` 通过 `s.assign(t)` 来调用函数，通过栈 `m` 给栈 `s` 赋值，并返回当前栈

8. 打印栈的所有元素函数 `print`

返回值为空

无参数，栈 `s` 通过 `s.print()` 调用函数来打印当前栈的所有元素

9. 析构函数: `~OOSTK`

析构函数无返回值和参数

程序运行结束时自动调用析构函数，释放栈 `s` 的空间

3) 运算符重载面向对象的 `STACK`

定义一个类，数据成员如下：

```
int *const elems;      //申请内存用于存放栈的元素
const int max;         //栈能存放的最大元素个数
int pos;               //栈实际已有元素个数，栈空时 pos=0;
```

函数成员定义如下

1. 构造函数: `STACK`

构造函数无返回值

初始化函数参数为栈的容量 `m`，通过 `STACK S(m)` 调用来构造一个栈。因为 `max` 和 `elems` 均为制度变量，则需要通过参数表来初始化变量。

然后重载这个初始化函数，参数是栈 `s` 的引用，通过 `STACK s(&S)` 调用来构造一个栈。同样通过参数列表来初始化变量。

2. 计算栈的容量函数: `size`

返回值为 `int`

函数没有参数，栈 `s` 通过 `s.size()` 调用 `size` 函数，得到当前栈的容量

3. 计算栈的当前元素个数函数: `operator int ()`

返回值为 `int`

通过类型强制转换，运算符重载，来实现将栈类型变量转化整型，函数得到栈的大小，作为整型数据传出

没有参数，栈 `s` 通过 `int (STACK)` 得到栈的大小，得到当前栈的元素个数

4. 查询某个位置的元素函数: `operator []`

返回值为 `int`

重载运算符 `[]` 来得到栈 `s` 位于 `x` 处的元素

函数参数: 位置变量 `x`，整型，栈 `s` 通过 `s[x]` 调用的到栈 `s` 中 `x` 处的元素

5. 入栈操作函数: `operator <<`

返回值为栈的引用

重载运算符 `<<`，实现入栈操作

参数为要入栈的元素 `e`，整型，栈 `s` 通过 `s<<e` 调用来将元素 `e` 插入到栈 `s` 中

6. 出栈操作函数: `operator >>`

返回值为栈的引用

重载运算符 `>>` 实现入栈操作

参数为变量 `e` 的引用，整型，栈 `s` 通过 `s>>e` 调用来将元素栈 `s` 最顶部的元素出栈，并赋值给 `e`

7. 给栈赋值函数: `operator =`

返回值为栈的引用

这里的赋值要通过深拷贝实现，通过重载=运算符，方法为借尸还魂法：

```
this->~STACK();  
    new (this)STACK(s);  
    cout<<"Assign the stack from s successfully\n";  
    return *this;
```

参数为另一个栈的引用，栈 s 通过 s=m 来调用函数，通过栈 m 给栈 s 赋值，并返回当前栈

8.打印栈的所有元素函数 print

返回值为空

无参数，栈 s 通过 s.print(),调用函数来打印当前栈的所有元素

9.析构函数:~OOSTK

析构函数无返回值和参数

程序运行结束时自动调用析构函数，释放栈 s 的空间

4) 由 2 个 STACK 组成的队列

总体设计：队列是满足先进先出的特点，而栈是先进后出的特点，通过两个栈，元素先进栈 1,然后再从栈 1 出栈导栈 2,则在栈 1 中正好满足先进先出的顺序。构造两个栈的对象，通过栈的聚合实现。

1.构造函数：QUE2S

构造函数无返回值

初始化函数参数为栈的容量 m，通过 QUE2S q (m) 调用来构造一个队列。因为 max 和 elems 均为制度变量，则需要通过参数表来初始化变量。

然后重载这个初始化函数，参数是队列 Q 的引用，通过 QUE2S q (&Q) 调用来构造一个队列。同样通过参数列表来初始化变量。

2.计算队列的当前元素个数函数:operator int ()

返回值为 int

通过类型强制转换，运算符重载，来实现将队列类型变量转化整型，函数得到队列的大小，作为整型数据传出

没有参数，队列 q 通过通过 int (QUE2S) 得到队列的大小，得到当前队列的元素个数

3..入队列操作函数: operator <<

返回值为队列的引用

重载运算符<<,实现入队列操作

参数为要入队列的元素 e，整型，栈 s 通过 q<<e 调用来将元素 e 插入到队列 q 中

4.出队列操作函数:operator >>

返回值为队列的引用

重载运算符>>实现出队列操作

参数为变量 e 的引用，整型，队列 q 通过 q>>e 调用来将队列最顶部的元素出队列，并赋值给 e

5.给队列赋值函数: operator =

返回值为队列的引用

这里的赋值要通过深拷贝实现，通过重载=运算符，方法为借尸还魂法：

```
this->~QUE2S();  
    new (this)QUE2S(q);  
    cout<<"Assign the queue by Q succedddfully\n";  
    return *this;
```

参数为另一个队列的引用，队列 q 通过 q=m 来调用函数，通过队列 m 给队列 s 赋值，并返回当前队列

6.打印队列的所有元素函数 print

返回值为空

无参数，队列 q 通过 q.print(),调用函数来打印当前队列的所有元素

7.析构函数:~QUE2S

析构函数无返回值和参数

程序运行结束时自动调用析构函数，释放队列 q 的空间

5) 从 STACK 继承的队列

总体设计：队列是满足先进先出的特点，而栈是先进后出的特点，通过两个栈，元素先进栈 1,然后再从栈 1 出栈导栈 2,则在栈 1 中正好满足先进先出的顺序。一个栈来自于继承，一个栈直接构造对象，通过栈的继承实现。

1.构造函数：QUE2S

构造函数无返回值

初始化函数参数为栈的容量 m，通过 QUE2S q (m) 调用来构造一个队列。因为 max 和 elems 均为制度变量，则需要通过参数表来初始化变量。

然后重载这个初始化函数，参数是队列 Q 的引用，通过 QUE2S q (&Q) 调用来构造一个队列。同样通过参数列表来初始化变量。

2.计算队列的当前元素个数函数:operator int ()

返回值为 int

通过类型强制转换，运算符重载，来实现将队列类型变量转化整型，函数得到队列的大小，作为整型数据传出

没有参数，队列 q 通过通过 int (QUE2S) 得到队列的大小，得到当前队列的元素个数

3.入队列操作函数: operator <<

返回值为队列的引用

重载运算符<<,实现入队列操作

参数为要入队列的元素 e，整型，栈 s 通过 q<<e 调用来将元素 e 插入到队列 q 中

4.出队列操作函数:operator >>

返回值为队列的引用

重载运算符>>实现出队列操作

参数为变量 e 的引用，整型，队列 q 通过 q>>e 调用来将队列最顶部的元素出队列，并赋值给 e

5.给队列赋值函数: operator =

返回值为队列的引用

这里的赋值要通过深拷贝实现，通过重载=运算符，方法为借尸还魂法：

```
this->~QUEIS();  
new (this)QUEIS(q);  
cout<<"Assign the queue by q sucessfully\n";  
return *this;
```

参数为另一个队列的引用，队列 q 通过 q=m 来调用函数，通过队列 m 给队列 s 赋值，并返回当前队列

6.打印队列的所有元素函数 print

返回值为空

无参数，队列 q 通过 q.print(),调用函数来打印当前队列的所有元素

7.析构函数:~QUE2S

析构函数无返回值和参数

程序运行结束时自动调用析构函数，释放队列 q 的空间

三、软件开发

开发环境：Ubuntu 16.04 操作系统 g++ 编译器 vim 编辑器 gdb 调试器 Intel(R) Core(TM) i5-4200H
CPU @ 2.80GHz 64bits 硬件信息

五次实验，每次实验都有各自的头文件 data.h，其中包含了变量定义，函数定义，宏定义等相关信息。

每次实验都有自己的源文件，包含了函数实现，测试函数部分。

编译格式: `g++ .cpp .cpp -o test`, 直接在终端下进入特定文件目录, 然后输入编译命令进行编译。

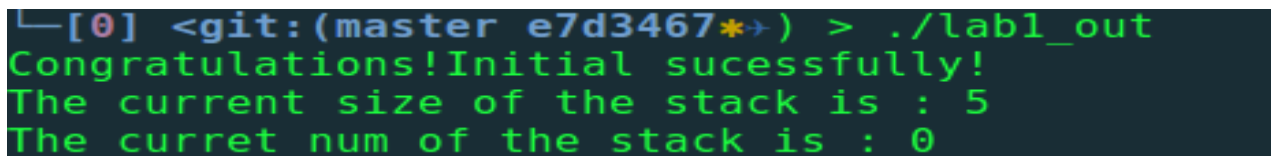
1、软件测试

1) 面向过程的 STACK

直接在终端下运行可执行文件 `./lab1_out`, 得到终端输出:

1. 初始化函数: `initPOSTK`

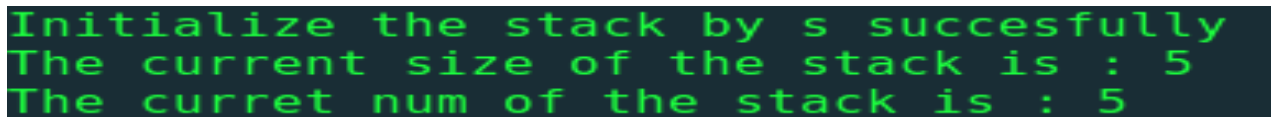
测试例: `initPOSTK(P,5);`



```
[0] <git:(master e7d3467*) > ./lab1_out
Congratulations!Initial sucessfully!
The current size of the stack is : 5
The curret num of the stack is : 0
```

图 3.1 构造函数测试

`initPOSTK(P,S);`




```
Initialize the stack by s sucesfully
The current size of the stack is : 5
The curret num of the stack is : 5
```

图 3.2 构造函数测试

如图, 两种构造函数的实现方式均可以正确构造栈 P, 测试通过。

2. 计算栈的容量函数: `size`

测试例: `size(P)`



```
The current size of the stack is : 5
```

图 3.3 size 函数测试

如图, 调用 `size` 函数, 正确输出当前栈的容量。

3. 计算当前栈的元素个数函数: `howMany`

测试例: `howMany(P)`



```
The curret num of the stack is : 5
```

图 3.4 howMany 函数测试

如图, 调用 `howMany` 函数, 正确输出当前栈的元素个数。

4. 查询某个位子的元素: `getelem`,

测试例: `getelem(P,x)`



```
The 0th num of the stack is : 1
```

图 3.5 gtelem 函数测试

如图, 调用 `getelem` 函数, 得到栈中某个位置的元素值。

5. 入栈函数: `push`

测试例: `push(P,2);`


```
The 0th num of the stack is : 1
Intert Sucessfully!
The current num of the stack is : 2
Print the stack is :
| 2 |
| 1 |
| -top- |
-----
```

图 3.6 push 函数测试

如图所示，开始栈中只有一个元素，元素 2 入栈后，提示入栈成功，并且打印栈显示当前栈的状态正确。

6.出栈函数: pop

测试例: pop(P,x)

```
Pop sucessfully!It's : 2
```

图 3.7 pop 函数测试

如图，从栈中出一个元素，进行退栈操作，提示退栈成功，并且显示退栈的元素，如图，运行正确。

7.给栈赋值函数:assign

测试例: P=assign(P,S)

```
Assign to stack from s sucessfully!
```

图 3.8 assign 函数测试

如图，进行 assign 操作，屏幕提示操作成功，并且栈已经发生改变。

8.打印当前栈函数:print

测试例: print(P);

```
Print the stack is :
| 0 |
| 4 |
| 3 |
| 2 |
| 1 |
| -top- |
-----
```

图 3.9 print 函数测试

如图，以图形化的方式打印出栈的状态，包括元素的个数、值、以及顺序。

9.销毁栈函数:destroyPOSTK

测试例: destroyPOSTK(P);

程序推退出时，手动调用 destroy 函数，释放空间，同时屏幕提示销毁栈策成功。

2) 面向对象的 STAC

1.构造函数: OOSTK

测试例: OOSTK s1(S);

```
Initialize the stack by s
```

图 3.10 构造函数测试

如图，通过栈 s 初始化栈 s1,屏幕提示成功。

2.计算栈的容量函数:size

测试例： `s1.size();`



```
The curent size of the stack is : 10
```

图 3.11 size 函数测试

如图，正确打印出当前栈的容量

3.计算栈的当前语速个数函数:howMany

测试例： `s1.howMany();`



```
The current number of the stack is :5
```

图 3.12 howMany 函数测试

如图，正确打印当前栈的元素个数。

4.查询某个位置的元素函数: getelem

测试例： `s1.getelem(0)`



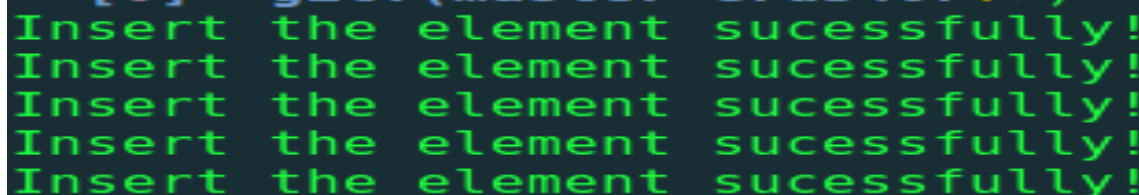
```
The element is : 1 at the 0
```

图 3.13 getelem 函数测试

如图，正确打印出位于栈第 0 位处的元素 0

5.入栈操作函数:push

测试例： `S.push(1).push(2).push(3).push(4).push(5)`



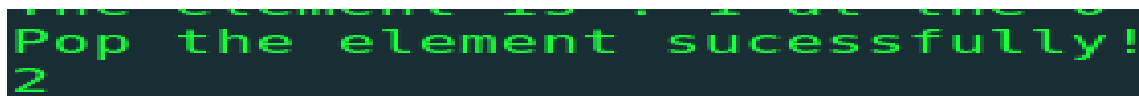
```
Insert the element sucessfully!  
Insert the element sucessfully!  
Insert the element sucessfully!  
Insert the element sucessfully!  
Insert the element sucessfully!
```

图 3.14 push 函数测试

如图，连续入栈操作，提示操作正确，没入栈一次打印一条信息。

6.出栈操作函数:pop

测试例： `s1.pop(x)`



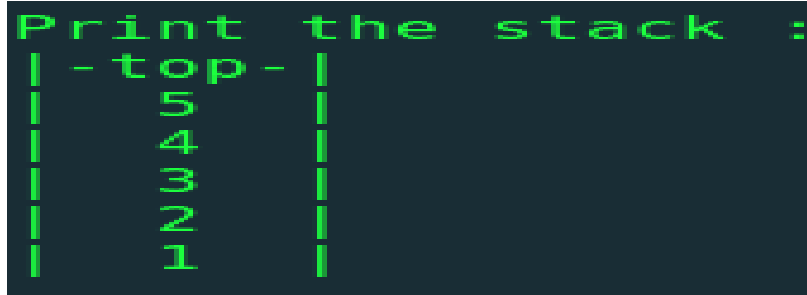
```
Pop the element sucessfully!  
2
```

图 3.15 pop 函数测试

如图，屏幕提示操作成功，并且正确打印出栈的元素。

8.打印栈的所有元素函数 print

测试例： `s1.print();`



```
Print the stack :  
| -top- |  
| 5 |  
| 4 |  
| 3 |  
| 2 |  
| 1 |  
|-----|
```

图 3.16 print 函数测试

如图，正确打印出当前栈的状态，用图形化显示出来，显示元素个数、元素值、元素顺序。

3) 运算符重载面向对象的 STACK

1.构造函数: STACK

测试例: STACK s1(S)

```
Initialize the stack by s
```

图 3.17 构造函数测试

如图，用栈 S 初始化栈 s1，屏幕提示操作成功。

2.计算栈的容量函数:size

测试例: s1.size()

```
The content of the stack is : 10
```

图 3.17 size 函数测试

如图，正确打印出当前栈的容量。

3.计算栈的当前元素个数函数:operator int ()

测试例: int many=s1;

```
The current number of the stack is : 5
```

图 3.18 int 重载函数测试

4.查询某个位置的元素函数: operator []

测试例: s1[5]

```
The inquired element is : 6
```

图 3.19 []重载函数测试

如图，正确打印出栈 s1[5] 处的元素。

5.入栈操作函数: operator <<

测试例: S<<(1)<<(2)<<(3)<<(4)<<(5)

```
Insert the element successfully!  
Insert the element successfully!  
Insert the element successfully!  
Insert the element successfully!  
Insert the element successfully!
```

图 3.20 <<重载函数测试

如图，进行连续入栈操作，每次打印入栈信息，屏幕提示操作成功。

6.出栈操作函数:operator >>

测试例: s1>>(x)

```
Pop the element successfully  
the element is : 6
```

图 3.21 >>重载函数测试

如图，进行出栈操作，屏幕提示操作成功，并且打印出出栈的元素。

7.给栈赋值函数:operator =

测试例: s1 = S

```
Initialize the stack by s  
Assign the stack from s successfully
```

图 3.22 =重载函数测试

如图，屏幕提示赋值操成功。

8.打印栈的所有元素函数 print

测试例：s1.print()

```
print the stack :
| -top- |
|   6   |
|   5   |
|   4   |
|   3   |
|   2   |
|   1   |
|-----|
```

图 3.23 print 函数测试

以图形化的方式打印出栈的当前状态，显示了栈中的元素个数，元素值，元素位置。

9.析构函数:~OOSTK

程序运行结束后，系统自动调用析构函数，屏幕打印析构信息，

4) 由 2 个 STACK 组成的队列

1.构造函数：QUE2S

测试例：QUE2S queue(10)

```
Construct Queue
```

图 3.24 构造函数测试

如图，构造成功，屏幕打印信息。

2.计算队列的当前元素个数函数:operator int （）

测试例：int many = queue

```
The num of the queue is : 5
```

图 3.25 int 重载函数测试

调用强制类型 int，得到 当前队列的元素个数。

3..入队列操作函数: operator <<

测试例：queue<<1<<2<<3<<4<<5

```
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
```

图 3.26 <<重载函数测试

连续入队列操作，每次入队打印一则信息，如图，入队成功。

4.出队列操作函数:operator >>

测试例：queue >> e;

```
Quit the queue successfully
Quit the queue successfully
Quit the queue successfully
```

图 3.27 >>重载函数测试

每次出队列时打印信息，如图，连续三次出队列成功。

5.给队列赋值函数: operator =

测试例: queue=(Q)

```
Initialize the queue from Q
Assign the queue by Q succeddfully
```

图 3.28 =重载函数测试

如图，通过重载运算符函数=进行队列的赋值操作，屏幕打印信息操作成功。

6.打印队列的所有元素函数 print

测试例: queue.print()

```
Print the queue :
| -top- |
|   1   |
|   9   |
|   8   |
|   6   |
|   5   |
| -----|
```

图 3.29 print 函数测试

如图，打印整个队列，包含了元素的个数，位置，元素值。

7.析构函数:~QUE2S

程序运行结束退出前，调用析构函数，析构队列对象。

5) 从 STACK 继承的队列

1.构造函数: QUE2S

测试例: QUEIS queue2(10)

```
Initialize the queue
CONSTRUCT~
CONSTRUCT~
```

图 3.30 构造函数测试

调用构造函数，构造一个队列，队列中两个栈一个来自于继承一个来自于构造函数。

2.计算队列的当前元素个数函数:operator int ()

测试例: int many = queue2

```
Te num of the queue is : 3
```

图 3.31 int 重载函数测试

调用类型强制转换函数，得到队列的元素个数。

3.入队列操作函数: operator <<

测试例: Q<<1<<2<<3<<4<<5

```
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
Insert into the queue successfully
```

图 3.32 <<重载函数测试

如图，连续入队列操作，每次如队列都会打印信息，提示入队列成功。

4.出队列操作函数:operator >>

测试例：queue2>>e

```
Quit the queue successfully
Quit the queue successfully
```

图 3.33 >>重载函数测试

连续出队列函数，每次出队列都打印信息，提示出队列成功。

5.给队列赋值函数: operator =

测试例：queue2=(Q)

```
Initialize the queue form s
Assign the queue by q sucessfully
```

图 3.34 =重载函数测试

如图，通过队列 s 来给队列 q 赋值，屏幕打印信息提示操作成功。

6.打印队列的所有元素函数 print

测试例：queue2.print()

```
Print the queue :
| -top- |
| 1     |
| 2     |
| 3     |
| 4     |
| 5     |
| -----|
```

图 3.35 print 构造函数测试

如图，打印了整个队列，包含了队列元素个数，元素位置，元素值。

7.析构函数:~QUE2S

程序运行结束退出时，自动调用析构函数，屏幕打印信息。

```
Desconstruct the queue
Desconstruct
Desconstruct
Desconstruct the queue
Desconstruct
Desconstruct
```

图 3.36 析构函数测试

四、特点与不足

1.技术特点

本次试验中，编程环境搭建在 Ubuntu 下，编译器直接采用 g++,编辑器采用 vim,调试器采用 gd 调试，对于程序的编译、链接、调试以及运行有了更深的理解。

本次实验中，对于实现了运算符的重载，其中有对于虚函数的应用。在用栈实现队列时，采用先进入栈 1，再进入站的方法实现，充分利用栈先进后出的特点和队列先进先出的特点，而且用了两种方式实现，分别是类的聚合和类的继承实现，体现了 C++面向对象的特点。

2.不足和改进的建议

不足之处在于：1、实验所花时间比较长，可能对 C++的特征理解还不够深刻；2、没有设计程序操作界面，只是让程序逐条运行；3、在有些地方没有加入检查，比如内存分配函数，数组越界等细节地方应当加入检查，维护程序的健壮性。

五、过程和体会

1.遇到的主要问题和解决方法

1、构造函数实现时，类体中定义了只读变量，开始不知道怎么初始化。解决方法是通过参数列表来初始化。

2、类的聚合实现试验中，聚合的类不知道怎么去修改和给成员赋值，因为类的成员作为私有成员不可见。方法是通过定义的函数去操作私有成员。

3、类的继承实现试验中，开始不知道怎么表示继承二构造的对象。解决方法是查阅资料，就用 STACK 类表示。

4、在类的赋值函数实现试验中，开始不知道怎么操作。通过问老师和查阅资料，解决方法是借尸还魂方法，先析构元对象，让后通过*this 指针来重新构造对象，比如：

```
QUEIS & QUEIS::operator = (const QUEIS &q){  
    this->~QUEIS();  
    new (this)QUEIS(q);  
    cout<<"Assign the queue by q sucessfully\n";  
    return *this;
```

2.课程设计的体会

本次实验，可以大致分为 5 个阶段，循序渐进，由浅入深，由面向过程过渡到面向对象，由简单的类的函数实现到运算符的函数重载，从类的操作到类的聚合、类的继承，从实践的角度了解了 C++面向对象程序设计的特征和方法，从而加深了对课本所学只是的理解。

本次实验不是很复杂，与课堂上学到的内容结合比较深，难点在于从面向过程转换到面向对象程序的设计好实现，思维的转变比较困难。

通过本次实验，加深了自己对于面向对象程序设计的思想和理解，提高了自己的变成能力，培养了自己的实践能力。

六、源码和说明

1.文件清单及其功能说明

```
//面向过程程序设计
头文件      data1.h
源程序      stack1.cpp
可执行程序  lab1_out

//面向对象程序设计
头文件      data2.h
源程序      stack2.cpp
可执行程序  lab2_out

//运算符重载程序设计
头文件      data3.h
源程序      stack3.cpp
可执行程序  lab3_out

//类的聚合程序设计
头文件      data4.h
源程序      que2s.cpp  stack.cpp
可执行程序  lab4_out   stack.cpp

//类的继承程序设计
头文件      data5.h
源程序      stack5.cpp
可执行程序  lab5_out
```

2.用户使用说明书

下载文件包，解压，有 5 个文件夹，代表 5 次实验，每个文件夹下存放这实验的源码以及可执行程序，在 Ubuntu 环境下，通过 chmod 命令添加权限，然后通过 ./lab1_out 直接运行程序即可。

3.源代码

/面向过程程序设计

【头文件 data1.h】

```
/******
```

```
> File Name: stack1.h
```

```
> Author:
```

```
> Mail:
```

```
> Created Time: 2016 年 10 月 14 日 星期五 18 时 50 分 20 秒
```

```
*****/
```

```
#ifndef _STACK1_H
```

```
#define _STACK1_H
```

```
#include<stdio.h>
```

```
#include<iostream>
```

```
#include<malloc.h>
```



```
#define TRUE 1
#define FALSE 0
#define OVERFLOW -1

struct POSTK{
    int *elems;    //store the elements
    int max;       //the max num
    int pos;       //the number of the actual elements
};

#define MAX_NUM 10

void initPOSTK(POSTK *const p,int m);
void initPOSTK(POSTK *const p,const POSTK &s);
int size(const POSTK *const p);
int howMany(const POSTK *const p);
int getelem(const POSTK *const p,int x);
POSTK *const push(POSTK *const p,int e);
POSTK *const pop(POSTK *const p,int &e);
POSTK *const assign(POSTK *const p,const POSTK &s);
void print(const POSTK *const p);
void destroyPOSTK(POSTK *const p);

#endif
```

【源程序 stack1.cpp】

```
/******
> File Name: stack1.cpp
> Author:
> Mail:
> Created Time: 2016 年 10 月 14 日 星期五 19 时 27 分 42 秒
*****/

#include "data1.h"
using namespace std;
int main(void){
    struct POSTK *P;
    struct POSTK S;
    int x;
    P=(struct POSTK *)malloc(sizeof(struct POSTK));
    S.elems=(int*)malloc(sizeof(int)*5);
    S.elems[0]=1;
    S.elems[1]=2;
    S.elems[2]=3;
    S.elems[3]=4;
    S.max=5;
```

```

S.pos=5;
initPOSTK(P,5);
cout<<"The current size of the stack is : "<<size(P)<<"\n";
cout<<"The curret num of the stack is : "<<howMany(P)<<"\n";
if(getelem(P,x)>=0){
    cout<<"The 0th num of the stack is : "<<getelem(P,x)<<"\n";
}
else
    cout<<"print the 0th num error!\n";
push(P,1);
if(getelem(P,x)>=0){
    cout<<"The 0th num of the stack is : "<<getelem(P,x)<<"\n";
}
else
    cout<<"print the 0th num error!\n";
push(P,2);
print(P);
pop(P,x);
P=assign(P,S);
print(P);
cout<<"The current size of the stack is : "<<size(P)<<"\n";
cout<<"The curret num of the stack is : "<<howMany(P)<<"\n";
destroyPOSTK(P);
}

/*****
 * initPOSTK
 *
 * p,m
 *****/
void initPOSTK(POSTK *const p,int m){
    if(p->elems=(int *)malloc(sizeof(int) * m)){
        p->max=m;
        p->pos=0;
        cout<<"Congratulations!Initial sucessfully!\n";
    }
}

void initPOSTK(POSTK * const p,const POSTK &s){
    if(p->elems=(int *)malloc(sizeof(int)*s.max)){
        p->max=s.max;
        p->pos=s.pos;
        for(int i=0;i<s.max;i++){
            *((p->elems)+i)=*(s.elems+i);
        }
        cout<<"Initialize the stack by s succesfully"<<"\n";
    }
}

```

```

int size(const POSTK *const p){
    return p->max;
}
int howMany(const POSTK * const p){
    return p->pos;
}
int getelem(const POSTK *const p,int x){
    if(x<p->pos){
        return *((p->elems)+x);
    }
    else return OVERFLOW;
}

POSTK *const push(POSTK *const p,int e){
    if(p->pos<p->max){
        p->elems[p->pos] =e;
        (p->pos)++;
        cout<<"Intert Sucessfully!\n";
        cout<<"The current num of the stack is : "<<howMany(p)<<"\n";
        return p;
    }
}

POSTK *const pop(POSTK *const p,int &e){
    if(p->pos>0){
        e=(p->elems)[--(p->pos)];
        cout<<"Pop sucessfully!It's : "<<e<<"\n";
        return p;
    }
}

POSTK *const assign(POSTK *const p,const POSTK &s){
    if(p->max<s.max){
        p->elems=(int *)realloc(p->elems,sizeof(int)*s.max);
    }
    for(int i=0;i<(s.max);i++){
        p->elems[i]=*(s.elems+i);
    }
    p->pos=s.pos;
    cout<<"Assign to stack from s sucessfully!"<<"\n";
    return p;
}

void print(const POSTK * const p){
    cout<<"Print the stack is : \n";
    for(int i=p->pos-1;i>=0;i--){
        cout<<"|  ";
        cout<<(p->elems)[i];
    }
}

```

```

        cout<<"  \n";
    }
    cout<<"|-top-\n";
    cout<<"-----";
    cout<<"\n";
}
void destroyPOSTK(POSTK * const p){
    free(p->elems);
    p->max=0;
    p->pos=0;
    cout<<"Destroy sucessfully!\n";
}

```

//面向对象程序设计

【头文件 data2.h】

```

/*****
> File Name: stack1.h
> Author:
> Mail:
> Created Time: 2016 年 10 月 14 日 星期五 18 时 50 分 20 秒
*****/

```

```

#ifndef _STACK1_H
#define _STACK1_H
#include<stdio.h>
#include<iostream>
#include<malloc.h>

#define TRUE 1
#define FALSE 0
#define OVERFLOW -1

#define MAX_NUM 10

```

```

class OOSTK{
    int *const elems;      //申请内存用于存放栈的元素
    const int max;         //栈能存放的最大元素个数
    int pos;               //栈实际已有元素个数，栈空时 pos=0;
public:
    OOSTK(int m);          //初始化栈：最多 m 个元素
    OOSTK(const OOSTK&s);  //用栈 s 拷贝初始化栈
    int size ( ) const;    //返回栈的最大元素个数 max
    int howMany ( ) const; //返回栈的实际元素个数 pos
    int getelem(int x) const; //取下标 x 处的栈元素
    OOSTK& push(int e);    //将 e 入栈，并返回当前栈
    OOSTK& pop(int &e);    //出栈到 e，并返回当前栈

```

```
OOSTK& assign(const OOSTK&s); //赋 s 给栈，并返回被赋值的当前栈
void print( ) const;          //打印栈
~OOSTK();                     //销毁栈
};
#endif
【源程序 stack2.cpp】
/*****
> File Name: stack2.cpp
> Author:
> Mail:
> Created Time: 2016 年 10 月 21 日 星期五 18 时 51 分 59 秒
*****/

#include"data2.h"
#define STACK_DEBUG
// #undef STACK_DEBUG

using namespace std;
OOSTK::OOSTK(int m):max(MAX_NUM),elems(new int[sizeof(int)*m]){
    pos=0;
}
OOSTK::OOSTK(const OOSTK &s):max(s.max),elems(new int[sizeof(int)*s.max]){
    pos=s.pos;
    for(int i=0;i<pos;i++){
        elems[i]=s.elems[i];
    }
#ifdef STACK_DEBUG
    cout<<"Initialize the stack by s \n";
#endif
}
OOSTK::~~OOSTK(){
    if(elems==0)return;
#ifdef STACK_DEBUG
    cout<<"DESTRUCT~"<<"\n";
#endif
    delete elems;
    pos=0;
}

int OOSTK::size()const{
#ifdef STACK_DEBUG
    cout<<"The curent size of the stack is : "<<max<<"\n";
#endif
    return max;
}
```

```

int OOSTK::howMany()const{
#ifdef STACK_DEBUG
    cout<<"The current number of the stack is : "<<pos<<"\n";
#endif
    return pos;
}
int OOSTK::getelem(int x)const{
    if(x<pos){
#ifdef STACK_DEBUG
        cout<<"The element is : "<<elems[x]<<" at the "<<x<<"\n";
#endif
        return elems[x];
    }else
#ifdef STACK_DEBUG
        cout<<"ERROR!There is something wrong when inquire the elems."<<"\n";
#endif
        return 0;
}
void OOSTK::print()const{
#ifdef STACK_DEBUG
    cout<<"Print the stack : \n";
    cout<<"|-top-\n";
#endif
    for(int i=pos-1;i>=0;i--){
#ifdef STACK_DEBUG
        cout<<"|  ";
        cout<<elems[i];
        cout<<"  |\n";
#endif
    }
#ifdef STACK_DEBUG
    cout<<"-----\n";
#endif
}
OOSTK & OOSTK::push(int e){
    if(pos<max){
        elems[pos++]=e;
#ifdef STACK_DEBUG
        cout<<"Insert the element sucessfully!\n";
#endif
        return *this;
    }
    else
#ifdef STACK_DEBUG
        cout<<"Insert Error!\n";

```

```

#endif
}
OOSTK & OOSTK::pop(int &e){
    if(pos>0){
        e=elems[--pos];
#ifdef STACK_DEBUG
        cout<<"Pop the element sucessfully!\n";
#endif
        return *this;
    }
}
OOSTK & OOSTK::assign(const OOSTK &s){
    this->~OOSTK();
    new (this) OOSTK(s);
#ifdef STACK_DEBUG
    cout<<"assign the stack by s \n";
#endif
    return *this;
}

```

```

int main(){
    int x;
    OOSTK S(10);
    S.push(1).push(2).push(3).push(4).push(5);
    //OOSTK s1(10);
    OOSTK s1(S);
    s1.size();
    s1.howMany();
    s1.push(2);
    s1.print();
    s1.getelem(0);
    s1.pop(x);
    cout<<x<<"\n";
    s1.print();
}

```

//运算符重载程序设计

【头文件 data3.h】

```

/*****
> File Name: data.h
> Author: strawberrylin
> Mail: hust.wangli@gmail.com
> Created Time: 2016 年 10 月 28 日 星期五 18 时 41 分 06 秒
*****/

```

```

#ifndef _DATA_H
#define _DATA_H
#include<stdio.h>
#include<iostream>
#include<malloc.h>

#define TRUE 1
#define FALSE 0
#define OVERFLOW -1

#define MAX_NUM 10
//=====运算符重载面向对象的 STACK=====//
class STACK{
    int *const elems;      //申请内存用于存放栈的元素
    const int max;         //栈能存放的最大元素个数
    int pos;               //栈实际已有元素个数，栈空时 pos=0;
public:
    STACK(int m);          //初始化栈：最多 m 个元素
    STACK(const STACK&s);  //用栈 s 拷贝初始化栈
    virtual int size () const;      //返回栈的最大元素个数 max
    virtual operator int () const;  //返回栈的实际元素个数 pos
    virtual int operator[ ] (int x) const; //取下标 x 处的栈元素
    virtual STACK& operator<<(int e);   //将 e 入栈，并返回当前栈
    virtual STACK& operator>>(int &e);   //出栈到 e，并返回当前栈
    virtual STACK& operator=(const STACK&s); //赋 s 给当前栈并返回该栈
    virtual void print( ) const;        //打印栈
    virtual ~STACK( );                 //销毁栈
};
#endif

```

【源程序 stack3.cpp】

```

/*****
> File Name: stack3.cpp
> Author: strawberrylin
> Mail: hust.wanglin@gmail.com
> Created Time: 2016 年 10 月 28 日 星期五 19 时 28 分 58 秒
*****/

```

```

#include"data3.h"
using namespace std;
int main(void){
    int x=1;
    STACK S(10);
    S<<(1)<<(2)<<(3)<<(4)<<(5);
    STACK s1(S);
    int size=s1.size();

```



```

        cout<<"The content of the stack is :  "<<size<<"\n";
        int many=s1;
        cout<<"The current number of the stack is : "<<many<<"\n";
        s1<<(6);
        s1.print();
        cout<<"The inquired element is : "<<s1[5]<<"\n";
        s1>>(x);
        cout<<"the element is : "<<x<<"\n";
        s1.print();
        S<<6;
        s1 = S;
        s1.print();
        return 0;
    }
    STACK::STACK(int m):max(m),elems(new int[sizeof(int)*m]){
        pos=0;
    }
    STACK::STACK(const STACK &s):max(s.max),elems(new int[sizeof(int)*s.max]){
        pos=s.pos;
        for(int i=0;i<s.max;i++){
            elems[i]=s.elems[i];
        }
        cout<<"Initialize the stack by s\n";
    }
    STACK::~~STACK(){
        if(elems){
            delete elems;
            pos=0;
            cout<<"DESTRUCT~\n";
        }
    }
    int STACK::size()const{
        return max;
    }
    STACK::operator int()const {
        return pos;
    }
    int STACK::operator [](int x)const{
        if(x<pos)
            return elems[x];
        else
            cout<<"sorry!There is something wrong when inquore the element\n";
    }
    STACK& STACK::operator << (int e){
        if(pos<max) {

```

```

        elems[pos++]=e;
        cout<<"Insert the element successfully!\n";
        return *this;
    }
    cout<<"Sorry!Insert failed";
}
STACK& STACK::operator >>(int &e){
    if(pos){
        e=elems[--pos];
        cout<<"Pop the element successfully\n";
        return *this;
    }
    else
        cout<<"Sorry!Pop failed\n";
}
STACK& STACK::operator =(const STACK&s){
    this->~STACK();
    new (this)STACK(s);
    cout<<"Assign the stack from s successfully\n";
    return *this;
}
void STACK::print()const{
    cout<<"print the stack :  \n";
    cout<<"|-top-|\n";
    for(int i=pos-1;i>=0;i--){
        cout<<"|  ";
        cout<<elems[i];
        cout<<"  |\n";
    }
    cout<<"-----\n";
}

```

//类的聚合程序设计

【头文件 data4.h】

/*

 */

> File Name: sdata.h

> Author: strawberrylin

> Mail: hust.wangli@gmail.com

> Created Time: 2016 年 10 月 28 日 星期五 18 时 41 分 06 秒

 */

#ifndef _DATA_H

#define _DATA_H

#include<stdio.h>

#include<iostream>

```
#include<malloc.h>

#define TRUE 1
#define FALSE 0
#define OVERFLOW -1

#define MAX_NUM 10
//=====运算符重载面向对象的 STACK=====//
class STACK{
    int *const elems;      //申请内存用于存放栈的元素
    const int max;         //栈能存放的最大元素个数
    int pos;               //栈实际已有元素个数，栈空时 pos=0;
public:
    STACK(int m);          //初始化栈：最多 m 个元素
    STACK(const STACK&s);  //用栈 s 拷贝初始化栈
    virtual int size ( ) const;      //返回栈的最大元素个数 max
    virtual operator int ( ) const;  //返回栈的实际元素个数 pos
    virtual int operator[ ](int x) const; //取下标 x 处的栈元素
    virtual STACK& operator<<(int e);  //将 e 入栈，并返回当前栈
    virtual STACK& operator>>(int &e); //出栈到 e，并返回当前栈
    virtual STACK& operator=(const STACK&s); //赋 s 给当前栈并返回该栈
    virtual void print( ) const;        //打印栈
    virtual ~STACK( );                  //销毁栈
};
//=====由 2 个 STACK 组成的队列=====//
class QUE2S{
    STACK s1, s2;                //一个队列可由两个栈聚合而成
public:
    QUE2S(int m);                //初始化队列：每栈最多 m 个元素
    QUE2S(const QUE2S &q);        //用队列 q 拷贝构造新队列
    virtual operator int ( ) const; //返回队列的实际元素个数
    virtual QUE2S& operator<<(int e); //将 e 入队列，并返回当前队列
    virtual QUE2S& operator>>(int &e); //出队列到 e，并返回当前队列
    virtual QUE2S& operator=(const QUE2S &q); //赋 q 给当前队列并返回该队列
    void print( ) const;          //打印队列
    ~QUE2S( );                    //销毁队列
};
#endif
```

【源程序 que2s.cpp stack.cpp】

```

/*****
> File Name: que2s.cpp
> Author: strawberrylin
> Github: https://github.com/strawberrylin
> Created Time: 2016 年 11 月 05 日 星期六 14 时 18 分 59 秒
*****/

```

```

#include "data4.h"
using namespace std;
int main(){
    int e;
    QUE2S Q(10);
    Q<<1<<9<<8<<6<<5;
    QUE2S queue(10);
    queue<<1<<2<<3<<4<<5;
    cout<<"The num of the queue is : ";
    int many = queue;
    cout<<many<<"\n";
    queue.print();
    queue >> e;
    queue >> e;
    queue >> e;
    queue.print();
    queue=(Q);
    many=queue;
    cout<<"Tht num of the queue is : "<<many<<"\n";
    queue.print();
    return 0;
}
QUE2S::QUE2S(int m):s1(m),s2(m){
    cout<<"Construct Queue\n";
}
QUE2S::QUE2S(const QUE2S &q):s1(q.s1),s2(q.s2){
    s1=q.s1;
    s2=q.s2;
    cout<<"Initialize the queue from Q\n";
}
QUE2S::~~QUE2S(){
    cout<<"Destruct Queue\n";
}
QUE2S::operator int() const{
    return (int)s1;
}
QUE2S& QUE2S::operator << (int e){
    int num1 = s1;
    int num2;
    int temp;
    if(num1 < s1.size()){
        for(int i=0;i<num1;i++){
            s1 >> temp;
            s2 << temp;
        }
    }
}

```

```

    }
    s2 << e;
    num2=s2;
    for(int j=0;j<num2;j++){
        s2>>temp;
        s1<<temp;
    }
    cout<<"Insert into the queue successfully\n";
    return *this;
}
else
    cout<<"Insert into the queue failed\n";
}
QUE2S& QUE2S::operator >> (int &e){
    int num = s1;
    if(num){
        s1>>e;
        cout<<"Quit the queue successfully\n";
        return *this;
    }
    else
        cout<<"Quit tht queue failed\n";
}
QUE2S& QUE2S::operator = (const QUE2S &q){
    this->~QUE2S();
    new (this)QUE2S(q);
    cout<<"Assign the queue by Q succedddfully\n";
    return *this;
}
void QUE2S:: print()const{
    cout<<"Print the queue : \n";
    s1.print();
}

```

/*

 */

```

> File Name: stack.cpp
> Author: strawberrylin
> Mail: hust.wanglin@gmail.com
> Created Time: 2016 年 10 月 28 日 星期五 19 时 28 分 58 秒

```

```

#include "sdata.h"
using namespace std;
STACK::STACK(int m):max(m),elems(new int[sizeof(int)*m]){
    pos=0;
}

```

```
STACK::STACK(const STACK &s):max(s.max),elems(new int[sizeof(int)*s.max]){
    pos=s.pos;
    for(int i=0;i<s.max;i++){
        elems[i]=s.elems[i];
    }
}
STACK::~STACK(){
    if(elems){
        delete elems;
        pos=0;
    }
}
int STACK::size()const{
    return max;
}
STACK::operator int()const {
    return pos;
}
int STACK::operator [](int x)const{
    if(x<pos)
        return elems[x];
    else
        cout<<"sorry!There is something wrong when inquire the element\n";
}
STACK& STACK::operator << (int e){
    if(pos<max) {
        elems[pos++]=e;
        return *this;
    }
    cout<<"Sorry!Insert failed";
}
STACK& STACK::operator >>(int &e){
    if(pos){
        e=elems[--pos];
        return *this;
    }
    else
        cout<< "Sorry!Pop failed\n";
}
STACK& STACK::operator =(const STACK&s){
    this->~STACK();
    new (this)STACK(s);
    return *this;
}
void STACK::print()const{
```

```

        cout<<"|-top-|\n";
        for(int i=pos-1;i>=0;i--){
            cout<<"|  ";
            cout<<elems[i];
            cout<<"  |\n";
        }
        cout<<"-----\n";
    }
}

```

//类的继承程序设计

【头文件 data5.h】

/******

```

> File Name: sdata.h
> Author: strawberrylin
> Mail: hust.wangli@gmail.com
> Created Time: 2016 年 10 月 28 日 星期五 18 时 41 分 06 秒

```

*****/

```

#ifndef _DATA_H

```

```

#define _DATA_H

```

```

#include<stdio.h>

```

```

#include<iostream>

```

```

#include<malloc.h>

```

```

#define TRUE 1

```

```

#define FALSE 0

```

```

#define OVERFLOW -1

```

```

#define MAX_NUM 10

```

```

//=====运算符重载面向对象的 STACK=====//

```

```

class STACK{

```

```

    int *const elems;      //申请内存用于存放栈的元素
    const int max;         //栈能存放的最大元素个数
    int pos;               //栈实际已有元素个数，栈空时 pos=0;

```

```

public:

```

```

    STACK(int m);          //初始化栈：最多 m 个元素
    STACK(const STACK&s);  //用栈 s 拷贝初始化栈
    virtual int size() const;      //返回栈的最大元素个数 max
    virtual operator int() const;  //返回栈的实际元素个数 pos
    virtual int operator[] (int x) const;  //取下标 x 处的栈元素
    virtual STACK& operator<<(int e);    //将 e 入栈，并返回当前栈
    virtual STACK& operator>>(int &e);    //出栈到 e，并返回当前栈
    virtual STACK& operator=(const STACK&s); //赋 s 给当前栈并返回该栈
    virtual void print() const;          //打印栈
    virtual ~STACK();                    //销毁栈

```

```
};
//=====从 STACK 继承的队列=====//
class QUEIS: public STACK{    //STACK 作为构成队列的第 1 个栈
    STACK s;                //s 作为构成队列的第 2 个栈
public:
    QUEIS(int m);            //初始化队列：每栈最多 m 个元素
    QUEIS(const QUEIS &q);    //用队列 q 拷贝初始化队列
    virtual operator int () const;    //返回队列的实际元素个数
    virtual QUEIS& operator<<(int e);    //将 e 入队列，并返回当前队列
    virtual QUEIS& operator>>(int &e);    //出队列到 e，并返回当前队列
    virtual QUEIS& operator=(const QUEIS &q);    //赋 q 给队列并返回该队列
    virtual void print() const;    //打印队列
    virtual ~QUEIS();            //销毁队列
};
#endif
```

【源程序 queis.cpp stack.cpp】

```
/******
> File Name: queis.cpp
> Author: strawberrylin
> Github: https://github.com/strawberrylin
> Created Time: 2016 年 11 月 05 日 星期六 23 时 52 分 19 秒
*****/
```

```
#include "data5.h"
using namespace std;
int main(){
    int e;
    QUEIS Q(10);
    QUEIS queue2(10);
    Q<<1<<2<<3<<4<<5;
    queue2<<0<<9<<8;
    int many = queue2;
    cout<<"The num of the queue is : "<<many<<"\n";
    queue2.print();
    queue2>>e;
    queue2>>e;
    queue2.print();
    queue2=(Q);
    cout<<"The num of the queue is : "<<many<<"\n";
    queue2.print();
    return 0;
}
QUEIS::QUEIS(int m):s(m),STACK(m){
    cout<<"Initialize the queue\n";
}
```



```

QUEIS::QUEIS(const QUEIS &q):s(q.s),STACK(q){
    cout<<"Initialize the queue form s\n";
}
QUEIS::~~QUEIS(){
    cout<<"Desconstruct the queue\n";
}
QUEIS::operator int()const{
    int num = STACK::operator int();//fatherClass::operator name
    return num;
}
QUEIS& QUEIS::operator << (int e){
    int num1 = STACK::operator int();
    int num2;
    int temp;
    if(num1 < STACK::size()){
        for(int i=0;i<num1;i++) {
            STACK::operator >> (temp);
            s << temp;
        }
        s << e;
        num2 = s;
        for(int j=0;j<num2;j++){
            s >> temp;
            STACK::operator << (temp);
        }
        cout<<"Insert into the queue successfully\n";
        return *this;
    }
    else
        cout<<"Quitthe queue failed\n";
}
QUEIS& QUEIS::operator >> (int &e){
    int num = STACK::operator int();
    if(num){
        STACK::operator >> (e);
        cout<<"Quit the queue successfully\n";
        return *this;
    }
    else
        cout<<"Quit the queue failed\n";
}
QUEIS & QUEIS::operator = (const QUEIS &q){
    this->~QUEIS();
    new (this)QUEIS(q);
    cout<<"Assign the queue by q sucessfully\n";
}

```

```
        return *this;
    }
    void QUEIS:: print()const{
        cout<<"Print the queue : \n";
        STACK::print();
    }
```

```
/******
```

```
> File Name: stack.cpp
> Author: strawberrylin
> Mail: hust.wanglin@gmail.com
> Created Time: 2016 年 10 月 28 日 星期五 19 时 28 分 58 秒
```

```
*****/
```

```
#include "data5.h"
using namespace std;
STACK::STACK(int m):max(m),elems(new int[sizeof(int)*m]){
    cout<<"CONSTRUCT~\n";
    pos=0;
}
STACK::STACK(const STACK &s):max(s.max),elems(new int[sizeof(int)*s.max]){
    pos=s.pos;
    for(int i=0;i<s.max;i++){
        elems[i]=s.elems[i];
    }
}
STACK::~~STACK(){
    if(elems){
        delete elems;
        pos=0;
        cout<<"Desconstruct\n";
    }
}
int STACK::size()const{
    return max;
}
STACK::operator int()const {
    return pos;
}
int STACK::operator [](int x)const{
    if(x<pos)
        return elems[x];
    else
        cout<<"sorry!There is something wrong when inquare the element\n";
}
STACK& STACK::operator << (int e){
```

```
        if(pos<max) {
            elems[pos++]=e;
            return *this;
        }
        cout<<"Sorry!Insert failed";
    }
STACK& STACK::operator >>(int &e){
    if(pos){
        e=elems[--pos];
        return *this;
    }
    else
        cout<< "Sorry!Pop failed\n";
}
STACK& STACK::operator =(const STACK&s){
    this->~STACK();
    new (this)STACK(s);
    return *this;
}
void STACK::print()const{
    cout<<"|-top-|\n";
    for(int i=pos-1;i>=0;i--){
        cout<<"|  ";
        cout<<elems[i];
        cout<<"  |\n";
    }
    cout<<"-----\n";
}
```