|  |
| --- |
|  |
| **课 程 实 验 报 告**  **课程名称：面向对象程序设计**  **院 系 ：计算机科学与技术**  **专业班级 ： IOT1601**  **学 号 ：**  **姓 名 ：**  **指导教师 ：**  **2015 年 12 月 20 日** |

**目 录**

[实验一.面向过程的整型栈编程 1](#_Toc438484867)

[1.需求分析 1](#_Toc438484868)

[1.1题目要求 1](#_Toc438484869)

[1.2需求分析 1](#_Toc438484870)

[2.系统设计 2](#_Toc438484871)

[2.1概要设计 2](#_Toc438484872)

[2.2详细设计 4](#_Toc438484873)

[3.软件开发 5](#_Toc438484874)

[4.软件测试 6](#_Toc438484875)

[5.特点与不足 11](#_Toc438484876)

[5.1技术特点 11](#_Toc438484877)

[5.2不足和改进的建议 12](#_Toc438484878)

[6.过程和体会 12](#_Toc438484879)

[6.1遇到的主要问题和解决方法 12](#_Toc438484880)

[6.2课程设计的体会 12](#_Toc438484881)

[7.源码和说明 12](#_Toc438484882)

[7.1文件清单及其功能说明 12](#_Toc438484883)

[7.2用户使用说明书 12](#_Toc438484884)

[7.3源代码 12](#_Toc438484885)

[实验二.面向对象的整型栈编程 22](#_Toc438484886)

[1.需求分析 22](#_Toc438484887)

[1.1题目要求 22](#_Toc438484888)

[1.2需求分析 22](#_Toc438484889)

[2.系统设计 23](#_Toc438484890)

[2.1概要设计 23](#_Toc438484891)

[2.2详细设计 25](#_Toc438484892)

[3.软件开发 27](#_Toc438484893)

[4.软件测试 27](#_Toc438484894)

[5.特点与不足 32](#_Toc438484895)

[5.1技术特点 32](#_Toc438484896)

[5.2不足和改进的建议 32](#_Toc438484897)

[6.过程和体会 32](#_Toc438484898)

[6.1遇到的主要问题和解决方法 32](#_Toc438484899)

[6.2课程设计的体会 32](#_Toc438484900)

[7.源码和说明 33](#_Toc438484901)

[7.1文件清单及其功能说明 33](#_Toc438484902)

[7.2用户使用说明书 33](#_Toc438484903)

[7.3源代码 33](#_Toc438484904)

[实验三.基于算符重载的整型栈编程 43](#_Toc438484905)

[1.需求分析 43](#_Toc438484906)

[1.1题目要求 43](#_Toc438484907)

[1.2需求分析 43](#_Toc438484908)

[2.系统设计 44](#_Toc438484909)

[2.1概要设计 44](#_Toc438484910)

[2.2详细设计 47](#_Toc438484911)

[3.软件开发 48](#_Toc438484912)

[4.软件测试 48](#_Toc438484913)

[5.特点与不足 53](#_Toc438484914)

[5.1技术特点 53](#_Toc438484915)

[5.2不足和改进的建议 53](#_Toc438484916)

[6.过程和体会 53](#_Toc438484917)

[6.1遇到的主要问题和解决方法 53](#_Toc438484918)

[6.2课程设计的体会 54](#_Toc438484919)

[7.源码和说明 54](#_Toc438484920)

[7.1文件清单及其功能说明 54](#_Toc438484921)

[7.2用户使用说明书 54](#_Toc438484922)

[7.3源代码 54](#_Toc438484923)

[实验四. 面向对象的整型队列编程 65](#_Toc438484924)

[1.需求分析 65](#_Toc438484925)

[1.1题目要求 65](#_Toc438484926)

[1.2需求分析 65](#_Toc438484927)

[2.系统设计 66](#_Toc438484928)

[2.1概要设计 66](#_Toc438484929)

[2.2详细设计 69](#_Toc438484930)

[3.软件开发 70](#_Toc438484931)

[4.软件测试 70](#_Toc438484932)

[5.特点与不足 74](#_Toc438484933)

[5.1技术特点 74](#_Toc438484934)

[5.2不足和改进的建议 75](#_Toc438484935)

[6.过程和体会 75](#_Toc438484936)

[6.1遇到的主要问题和解决方法 75](#_Toc438484937)

[6.2课程设计的体会 75](#_Toc438484938)

[7.源码和说明 76](#_Toc438484939)

[7.1文件清单及其功能说明 76](#_Toc438484940)

[7.2用户使用说明书 76](#_Toc438484941)

[7.3源代码 76](#_Toc438484942)

[实验五. 基于组合的整型队列编程 86](#_Toc438484943)

[1.需求分析 86](#_Toc438484944)

[1.1题目要求 86](#_Toc438484945)

[1.2需求分析 86](#_Toc438484946)

[2.系统设计 87](#_Toc438484947)

[2.1概要设计 87](#_Toc438484948)

[2.2详细设计 89](#_Toc438484949)

[3.软件开发 90](#_Toc438484950)

[4.软件测试 90](#_Toc438484951)

[5.特点与不足 94](#_Toc438484952)

[5.1技术特点 94](#_Toc438484953)

[5.2不足和改进的建议 95](#_Toc438484954)

[6.过程和体会 95](#_Toc438484955)

[6.1遇到的主要问题和解决方法 95](#_Toc438484956)

[6.2课程设计的体会 95](#_Toc438484957)

[7.源码和说明 95](#_Toc438484958)

[7.1文件清单及其功能说明 95](#_Toc438484959)

[7.2用户使用说明书 96](#_Toc438484960)

[7.3源代码 96](#_Toc438484961)

[实验六. 基于继承的整型队列编程 108](#_Toc438484962)

[1.需求分析 108](#_Toc438484963)

[1.1题目要求 108](#_Toc438484964)

[2.系统设计 109](#_Toc438484965)

[2.1概要设计 109](#_Toc438484966)

[2.2详细设计 109](#_Toc438484967)

[3.软件开发 110](#_Toc438484968)

[4.软件测试 111](#_Toc438484969)

[5.特点与不足 112](#_Toc438484970)

[5.1技术特点 112](#_Toc438484971)

[5.2不足和改进的建议 112](#_Toc438484972)

[6.过程和体会 113](#_Toc438484973)

[6.1遇到的主要问题和解决方法 113](#_Toc438484974)

[6.2课程设计的体会 113](#_Toc438484975)

[7.源码和说明 113](#_Toc438484976)

[7.1文件清单及其功能说明 113](#_Toc438484977)

[7.2用户使用说明书 113](#_Toc438484978)

[7.3源代码 114](#_Toc438484979)

实验一.面向过程的整型栈编程

**1.需求分析**

**1.1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、出栈等。整型栈类型及其操作函数采用非面向对象的纯C语言定义，请将完成上述操作的所有函数采用面向过程的方法编程， 然后写一个main函数对栈的所有操作函数进行测试。

struct STACK{

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 initSTACK(STACK \*const p, int m);//初始化p指空栈：可存m个元素

void initSTACK(STACK \*const p, const STACK&s); //用s初始化p指空栈

int size (const STACK \*const p); //返回p指的栈的最大元素个数max

int howMany (const STACK \*const p); //返回p指的栈的实际元素个数pos

int getelem (const STACK \*const p, int x); //取下标x处的栈元素

STACK \*const push(STACK \*const p, int e); //将e入栈，并返回p

STACK \*const pop(STACK \*const p, int &e); //出栈到e，并返回p

STACK \*const assign(STACK\*const p, const STACK&s);//赋给p指栈,返回p

void print(const STACK\*const p); //打印p指向的栈元素

void destroySTACK(STACK\*const p); //销毁p指向的栈，释放

**1.2需求分析**

编程要求实现面向过程的整型栈，故本项目即实现面向过程的整型栈的后台开发，并设计配套UI以方便使用者对已经实现的面向过程的整型栈进行使用及测试。

故本项目应包括如下部分：

1. 实现面向过程的整型栈的接口
2. 编写配套UI以对上述程序进行测试

**2.系统设计**

**2.1概要设计**

系统的总体结构图如图1.1所示。本项目共由初始化模块，功能选择模块，打印模块，入栈模块，出栈模块及栈指派模块组成。

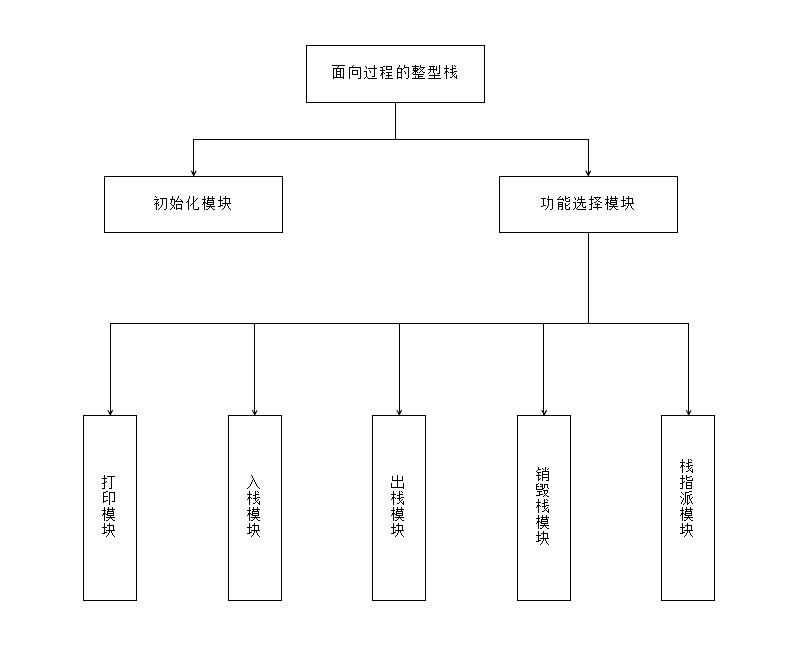


图 1.1 总体结构图

系统的总体流程图如图1.2所示。用户进入系统后应首先输入桟的初始化大小以决定以何种方式初始化栈，即以整数n还是以系统预设栈s初始化栈。

在初始化栈后用户可以通关功能选择模块选择将要执行的功能，特别地，用户在选择退出系统后将清空系统资源并退出本系统。

用户在选择打印栈后系统将展示栈已容纳元素个数，栈可容纳元素个数，栈内所有元素。

用户在选择入栈后系统将引导用户输入想要入栈的元素（整型数），再将用户输入的元素入栈。

用户在选择出栈后系统出栈，并打印出栈元素的值。

用户在选择销毁栈后系统将销毁栈。

用户在选择指派栈后系统会把系统预设栈指派给当前使用的栈。

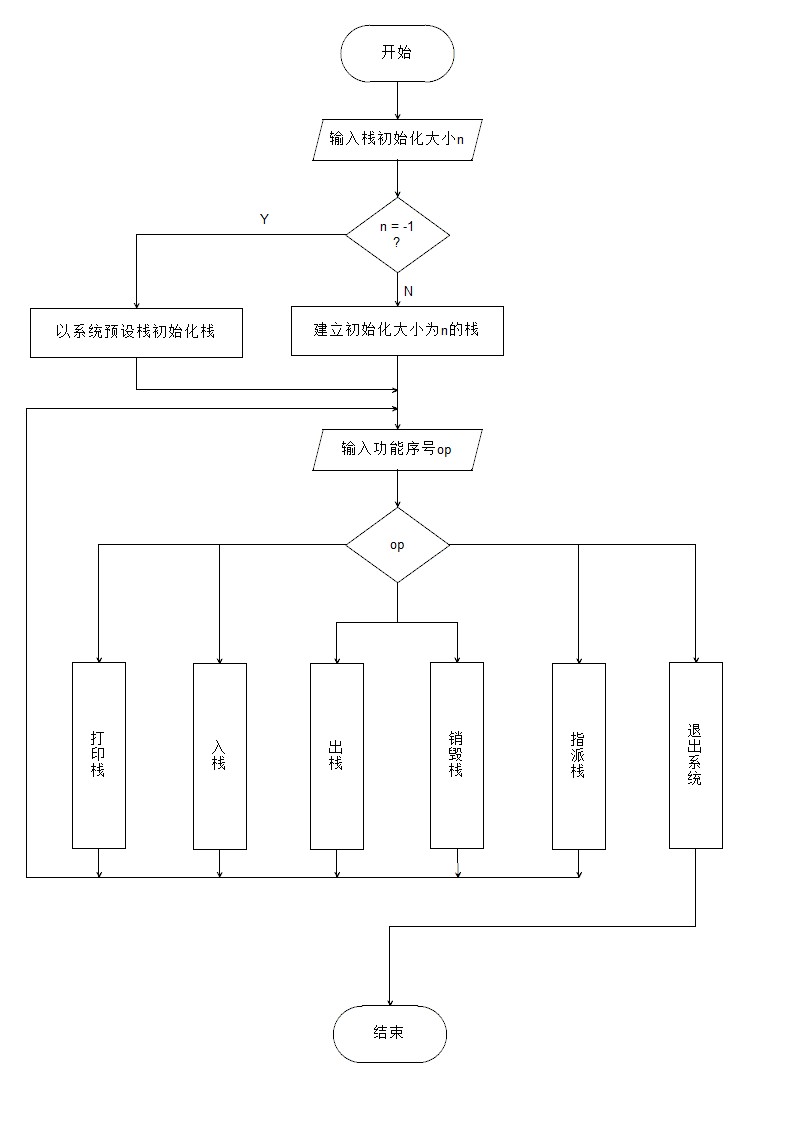


图 1.2 总体流程图

所有操作中由于入栈牵扯到是否需要对栈进行扩容的问题，因此需要设计详实的入栈的算法以应对栈空间不足的情况。入栈的流程如图1.3所示。

即在输入入栈元素e后首先判定栈是否满，若满即对栈基址进行扩容，然后拷贝原栈元素并入栈，否则直接将e入栈。

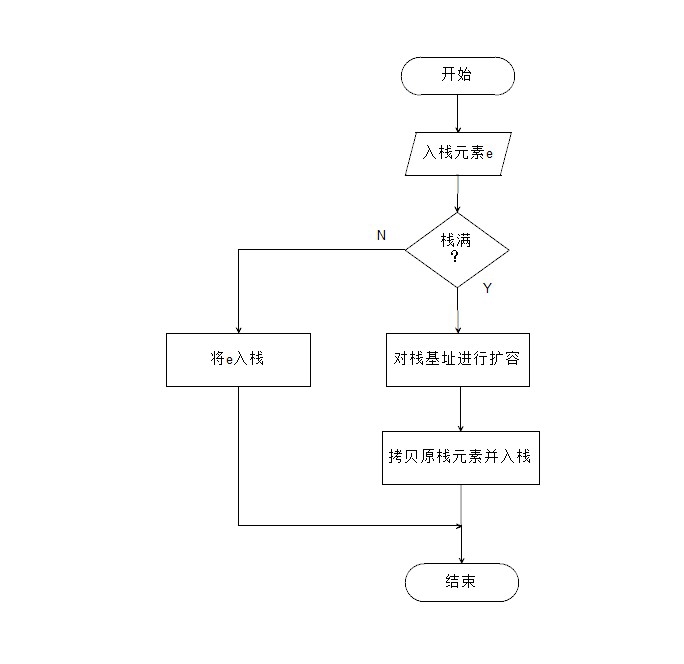


图 1.3 入栈流程图

**2.2详细设计**

1. 初始化模块：

模块功能：引导用户对所使用的栈进行初始化，包括使用整数m和栈s对栈进行初始化。

入口参数：STACK \* const StackUser 用户栈

STACK \* const StackSys 系统预设栈

出口参数：无

业务流程：参考图1.2

子程序：

* 1. void initSTACK(STACK \*const p, int m)。
  2. void initSTACK(STACK \*const p, const STACK &s)

1. 功能选择模块：int ChooseFunc()

模块功能：引导用户选择系统功能，包括打印栈、入栈、出栈、销毁栈、指派栈、退出系统。

入口参数：无

出口参数：用户选择的系统功能

业务流程：参考图1.2

1. 打印栈模块：void Func1(STACK \* const StackUser)

模块功能：打印显示栈的信息。包括栈的容量，已容纳元素个数，已容纳元素具体信息。

入口参数：STACK \* const StackUser

出口参数：无

子程序：

* 1. void print(const STACK\*const p)
  2. int size(const STACK \*const p)
  3. int howMany(const STACK \*const p)

1. 入栈模块：void Func2(STACK \* const StackUser)

模块功能：引导用户输入入栈元素，并将之入栈

入口参数：STACK \* const StackUser

出口参数：无

子程序：

* 1. STACK \*const push(STACK \*const p, int e)

1. 出栈模块：void Func3(STACK \* const StackUser)

模块功能：出栈并显示出栈元素。

入口参数：STACK \* const StackUser

出口参数：无

子程序：

* 1. STACK \*const pop(STACK \*const p, int &e)

1. 销毁栈模块：void Func4(STACK \* const StackUser)

模块功能：销毁栈。

入口参数：STACK \* const StackUser

出口参数：无

子程序：

* 1. void destroySTACK(STACK\*const p)

1. 栈指派模块：void Func5(STACK \* const StackUser)

模块功能：讲系统预设栈指派给用户当前所使用的栈。

入口参数：STACK \* const StackUser

出口参数：STACK \* const StackSys

子程序：

* 1. STACK \*const assign(STACK\*const p, const STACK&s)

**3.软件开发**

本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 建立初始化大小为10的整型栈。

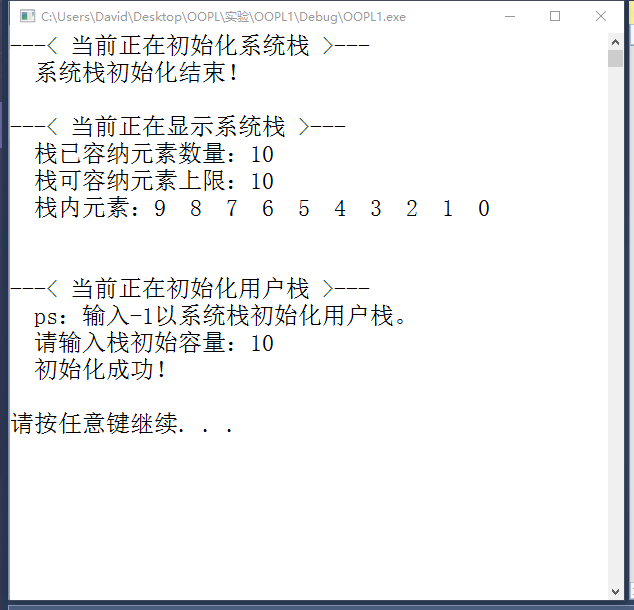


图1.4 使用整数m初始化栈

1. 打印栈的信息。

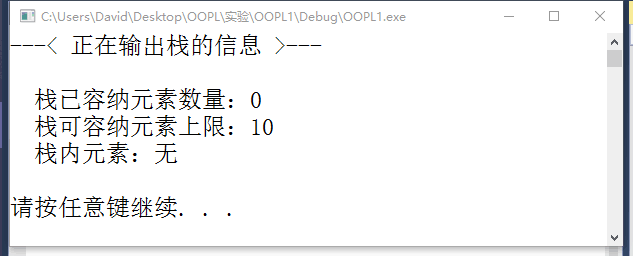


图1.5 打印栈中信息

1. 指派栈。

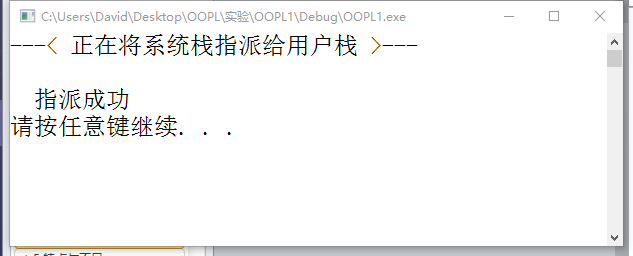


图1.6 指派栈

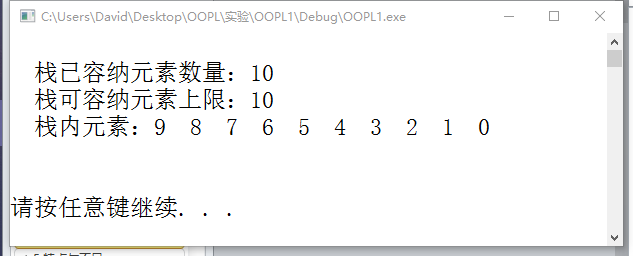


图1.7 打印栈信息

1. 出栈。

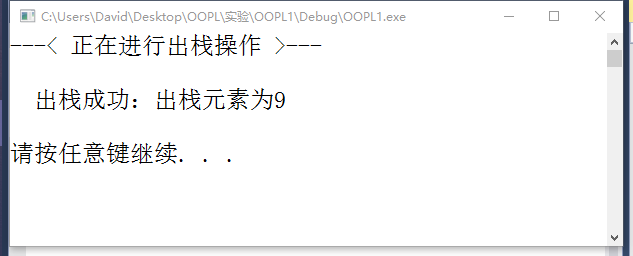


图1.8出栈

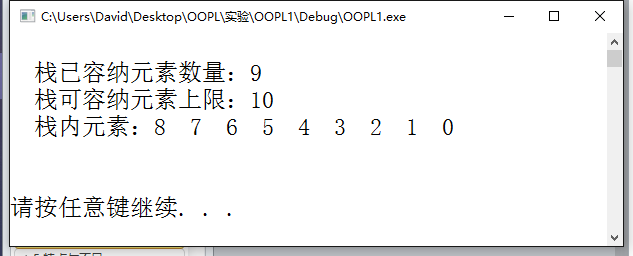


图1.9 打印栈信息

1. 入栈（未满）。

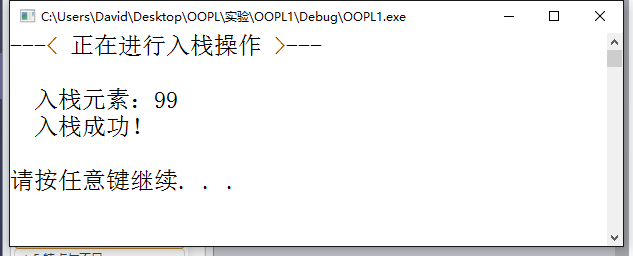


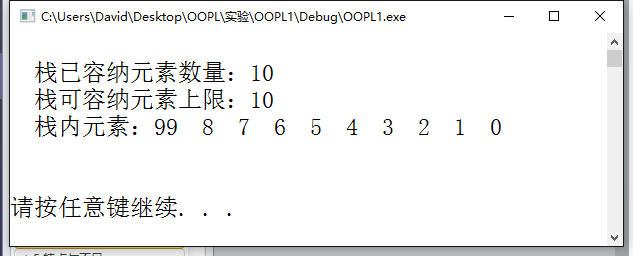
图1.10入栈

图1.11打印栈信息

1. 入栈（已满）。据图可知栈成功地完成了扩容。

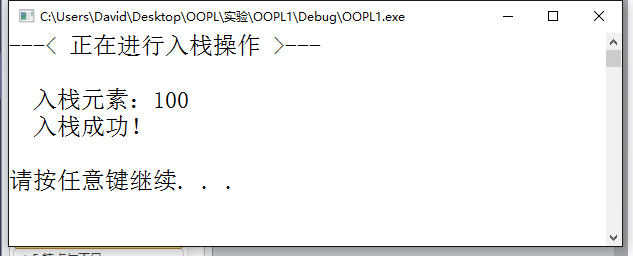


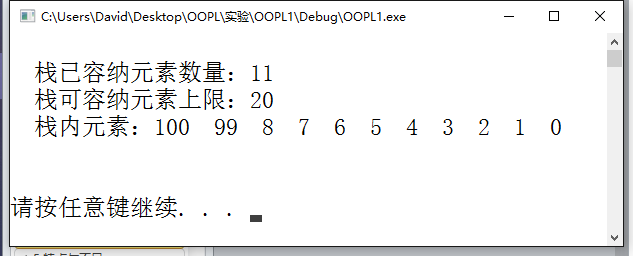
图1.12入栈

图1.13打印栈信息

1. 销毁栈。

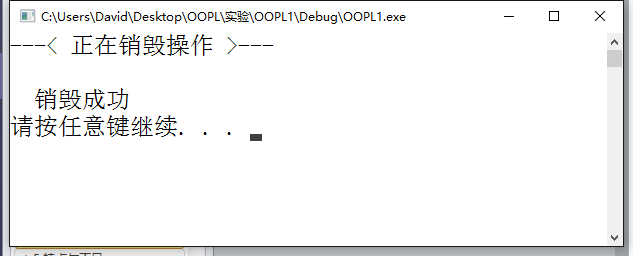


图1.14 销毁栈

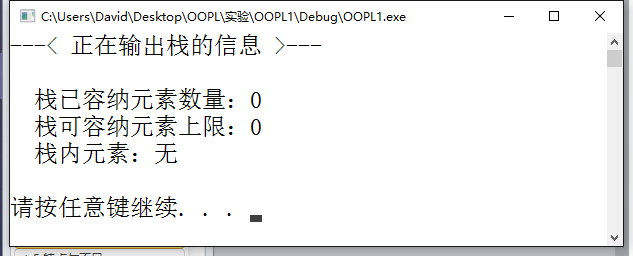


图1.15打印栈信息

1. 初始化桟（使用系统预设栈）

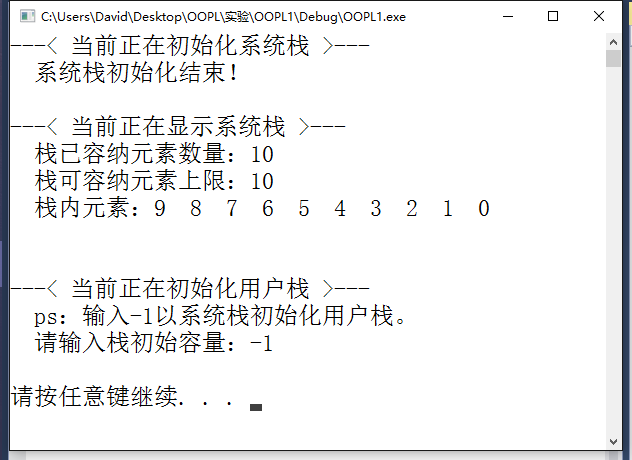


图1.16 初始化栈

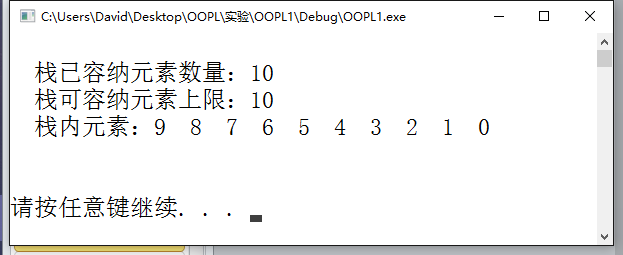


图1.17 打印栈

**5.特点与不足**

**5.1技术特点**

1. 能够做到自动为栈扩容
2. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
3. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. UI设计中有时为了节约篇幅，会采用一些自认为简洁明了的词汇，比如系统栈。实际上系统栈就是系统预设的一个栈，其内已经包含若干元素，方便用户使用它对栈进行初始化，或者assign给用户正在使用的栈。然而UI中并未给出对这个栈的解释，只能由用户体验后自行揣测其含义。这也是设计的一个不合理的地方。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

1. 输入字符流后scanf函数会将字符部分截取，而留下’\n’字符。解决方法：在每个输入函数后加上getchar()函数。

**6.2课程设计的体会**

很简单的实验，主要体现了C++兼容C的特性，及说明函数的重载。实验的主要难点倒在于UI的设计。不过目前设计的UI的复用性十分高，目测可以为以后的实验带来极大地便利。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL1\Debug下是vs生成的可执行调试文件。

OOPL1\OOPL1下是组成本项目的C++源代码文件。

**7.2用户使用说明书**

运行~\OPL1\Debug下的OOPL1.exe即可使用本程序。

**7.3源代码**

1. MyStack.h

#define STACK\_INCRE\_STEP 10

struct STACK {

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;

};

//默认输入的参数是正确的

void initSTACK(STACK \*const p, int m); //初始化p指向的栈：最多m个元素

void initSTACK(STACK \*const p, const STACK&s); //用栈s初始化p指向的栈

int size(const STACK \*const p); //返回p指向的栈的最大元素个数max

int howMany(const STACK \*const p); //返回p指向的栈的实际元素个数pos

int getelem(const STACK \*const p, int x); //取下标x处的栈元素

STACK \*const push(STACK \*const p, int e); //将e入栈，并返回p

STACK \*const pop(STACK \*const p, int &e); //出栈到e，并返回p

STACK \*const assign(STACK\*const p, const STACK&s); //赋s给p指的栈,并返回p

void print(const STACK\*const p); //打印p指向的栈

void destroySTACK(STACK\*const p); //销毁p指向的栈

void SysInitial(STACK \* const StackUser, STACK \* const StackSys); //初始化栈

void Func1(STACK \* const StackUser); //功能1

void Func2(STACK \* const StackUser); //功能2

void Func3(STACK \* const StackUser); //功能3

void Func4(STACK \* const StackUser); //功能4

void Func5(STACK \* const StackUser, STACK \* const StackSys); //功能5

int ChooseFunc();

void ScreenClr();

void ScreenPause();

1. main.cpp

#include "MyStack.h"

#include <stdlib.h>

#include <stdio.h>

int main()

{

// variables declaration

STACK \* StackUser = (STACK \*)malloc(sizeof(STACK)); //操作用栈

STACK \* StackSys = (STACK \*)malloc(sizeof(STACK)); //赋值用栈

int op;

//initial

if (StackUser == 0 || StackSys == 0)

exit(-1);

SysInitial(StackUser, StackSys);

//loop

while ((op = ChooseFunc()) != 0)

{

switch (op)

{

case 0:

op = -1;

break;

case 1:

Func1(StackUser);

break;

case 2:

Func2(StackUser);

break;

case 3:

Func3(StackUser);

break;

case 4:

Func4(StackUser);

break;

case 5:

Func5(StackUser, StackSys);

break;

}

}

//quit

ScreenClr();

printf("\n---< 正在退出系统 >---\n");

ScreenPause();

}

void SysInitial(STACK \* const StackUser, STACK \* const StackSys)

{

//初始化默认栈

int i, n, max, pos;

printf("---< 当前正在初始化系统栈 >---\n");

initSTACK(StackSys, 10);

for (i = 0; i < 10; i++)

{

push(StackSys, i);

}

printf(" 系统栈初始化结束！\n");

//显示默认栈

printf("\n---< 当前正在显示系统栈 >---\n");

max = size(StackSys);

pos = howMany(StackSys);

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

print(StackSys);

}

printf("\n");

//引导初始用户栈

printf("\n---< 当前正在初始化用户栈 >---\n");

printf(" ps：输入-1以系统栈初始化用户栈。\n");

do

{

printf(" 请输入栈初始容量：");

scanf\_s("%d", &n);

getchar();

if (n == -1)

{

initSTACK(StackUser, \*StackSys);

}

else if (n > -1)

{

initSTACK(StackUser, n);

printf(" 初始化成功！\n");

}

else

{

printf(" 请输入有效大小！\n");

}

} while (n < -1);

ScreenPause();

}

int ChooseFunc()

{

int n = -1; //用户选择的功能序号

/\* 输出功能选择提示语 \*/

ScreenClr();

printf("---< 选择程序功能 >---\n\n");

printf(" [1].查看栈信息\n");

printf(" [2].入栈操作\n");

printf(" [3].出栈操作\n");

printf(" [4].销毁栈\n");

printf(" [5].将系统栈赋给用户栈\n");

printf(" [0].退出系统\n");

printf("\n请输入功能代码：");

/\* 录入提示语 \*/

scanf\_s("%d", &n);

getchar();

/\* 输入验证 \*/

if (n < 0 || n > 5)

{

n = -1;

printf(" 错误的代码，请重新输入。\n");

ScreenPause();

}

return n;

}

void Func1(STACK \* const StackUser)

{

//initial

int max, pos, i;

//output

ScreenClr();

printf("---< 正在输出栈的信息 >---\n\n");

max = size(StackUser);

pos = howMany(StackUser);

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

print(StackUser);

}

printf("\n");

//wait

ScreenPause();

}

void Func2(STACK \* const StackUser)

{

//initial

int var;

//录入

ScreenClr();

printf("---< 正在进行入栈操作 >---\n\n");

printf(" 入栈元素：");

scanf\_s("%d", &var);

getchar();

push(StackUser, var);

printf(" 入栈成功！\n");

//wait

ScreenPause();

}

void Func3(STACK \* const StackUser)

{

//initial

int var;

//出栈

ScreenClr();

printf("---< 正在进行出栈操作 >---\n\n");

if (howMany(StackUser) < 1)

{

printf(" 出栈失败：栈为空\n");

}

else

{

pop(StackUser, var);

printf(" 出栈成功：出栈元素为%d\n", var);

}

//wait

ScreenPause();

}

void Func4(STACK \* const StackUser)

{

//initial

int var;

//销毁

ScreenClr();

printf("---< 正在销毁操作 >---\n\n");

destroySTACK(StackUser);

printf(" 销毁成功");

//wait

ScreenPause();

}

void Func5(STACK \* const StackUser, STACK \* const StackSys)

{

//initial

int var;

//指派

ScreenClr();

printf("---< 正在将系统栈指派给用户栈 >---\n\n");

assign(StackUser, \*StackSys);

printf(" 指派成功");

//wait

ScreenPause();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}

//基址分配失败，则elems=0,max=0,pos=0

void initSTACK(STACK \*const p, int m)

{

/\* initial struct elements \*/

p->elems = (int \*)malloc(m\*sizeof(int));

p->max = p->elems == NULL ? 0 : m;

p->pos = 0;

}

//基址分配失败，则elems=NULL,max=0,pos=0

void initSTACK(STACK \*const p, const STACK &s) //用栈s初始化p指向的栈

{

initSTACK(p, 0);

assign(p, s);

}

int size(const STACK \*const p)

{

return p->max;

}

int howMany(const STACK \*const p)

{

return p->pos;

}

int getelem(const STACK \*const p, int x)

{

return(p->elems)[x];

}

STACK \*const push(STACK \*const p, int e)

{

if (p->pos < p->max)

{

/\* 剩余空间充足：正常压栈 \*/

(p->elems)[(p->pos)++] = e;

}

else {

/\* 剩余空间不足：扩充栈，拷贝栈，再进行压栈 \*/

int \* EleBase = (int \*)malloc((p->max + STACK\_INCRE\_STEP)\*sizeof(int)); //扩充栈，基址

if (EleBase != NULL)

{

/\* 拷贝原栈元素并入栈 \*/

int i;

for (i = 0; i < p->pos; i++)

{

EleBase[i] = getelem(p, i);

}

EleBase[i] = e;

/\* 修改栈信息 \*/

p->max += STACK\_INCRE\_STEP;

p->pos++;

/\* 释放原栈 \*/

free(p->elems);

/\* 置入新栈 \*/

p->elems = EleBase;

}

}

return p;

}

STACK \*const pop(STACK \*const p, int &e)

{

e = (p->elems)[--(p->pos)];

return p;

}

STACK \*const assign(STACK\*const p, const STACK&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (s.max != p->max)

{

free(p->elems);

p->elems = (int \*)malloc(s.max\*sizeof(int));

p->max = s.pos;

}

/\* 拷贝 \*/

if (p->elems != NULL)

{

int i;

p->pos = s.pos;

for (i = 0; i < s.pos; i++)

{

(p->elems)[i] = (s.elems)[i];

}

}

else

{

p->max = 0;

p->pos = 0;

}

return p;

}

void print(const STACK\*const p)

{

int i;

for (i = p->pos; i > 0; i--)

{

printf("%d ", (p->elems)[i - 1]);

}

printf("\n");

}

void destroySTACK(STACK\*const p)

{

p->max = 0;

p->pos = 0;

free(p->elems);

p->elems = 0;

}

实验二.面向对象的整型栈编程

**1.需求分析**

**1.1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、出栈等。整型栈类型及其操作函数采用面向对象的C++语言定义，请将完成上述操作的所有函数采用C++编程， 然后写一个main函数对栈的所有操作函数进行测试。

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

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

STACK& push(int e); //将e入栈,并返回栈

STACK& pop(int &e); //出栈到e,并返回栈

STACK& assign(const STACK&s); //赋s给栈,并返回被赋值的栈

void print( ) const; //打印栈

~STACK( ); //销毁栈

};

**1.2需求分析**

编程要求实现面向对象的整型栈，故本项目即实现面向对象的整型栈的后台开发，并设计配套UI以方便使用者对已经实现的面向对象的整型栈进行使用及测试。

故本项目应包括如下部分：

1. 实现面向对象的整型栈的接口
2. 编写配套UI以对上述程序进行测试

**2.系统设计**

**2.1概要设计**

系统的总体结构图如图2.1所示。本项目共由初始化模块，功能选择模块，打印模块，入栈模块，出栈模块及栈指派模块组成。

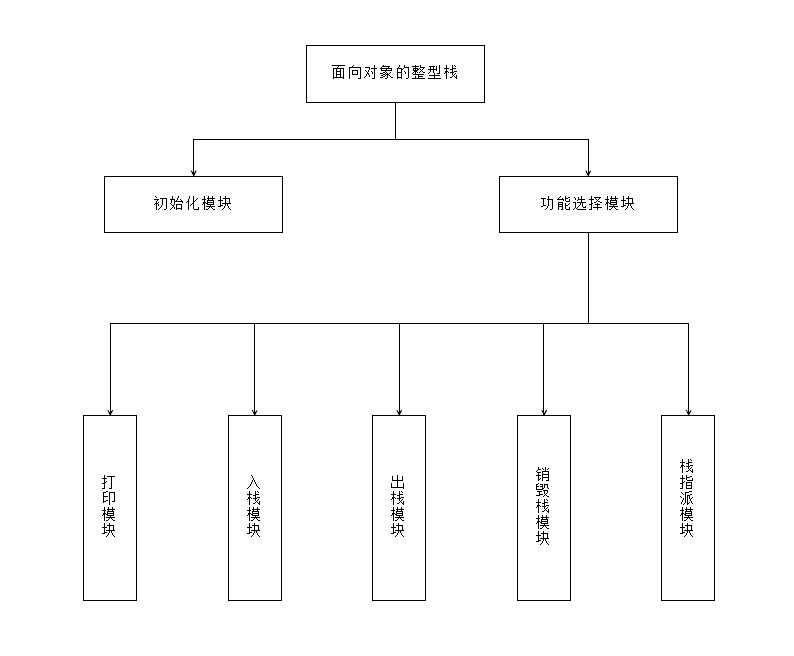


图 2.1 总体结构图

系统的总体流程图如图2.2所示。用户进入系统后应首先输入桟的初始化大小以决定以何种方式初始化栈，即以整数n还是以系统预设栈s初始化栈。

在初始化栈后用户可以通关功能选择模块选择将要执行的功能，特别地，用户在选择退出系统后将清空系统资源并退出本系统。

用户在选择打印栈后系统将展示栈已容纳元素个数，栈可容纳元素个数，栈内所有元素。

用户在选择入栈后系统将引导用户输入想要入栈的元素（整型数），再将用户输入的元素入栈。

用户在选择出栈后系统出栈，并打印出栈元素的值。

用户在选择销毁栈后系统将销毁栈。

用户在选择指派栈后系统会把系统预设栈指派给当前使用的栈。

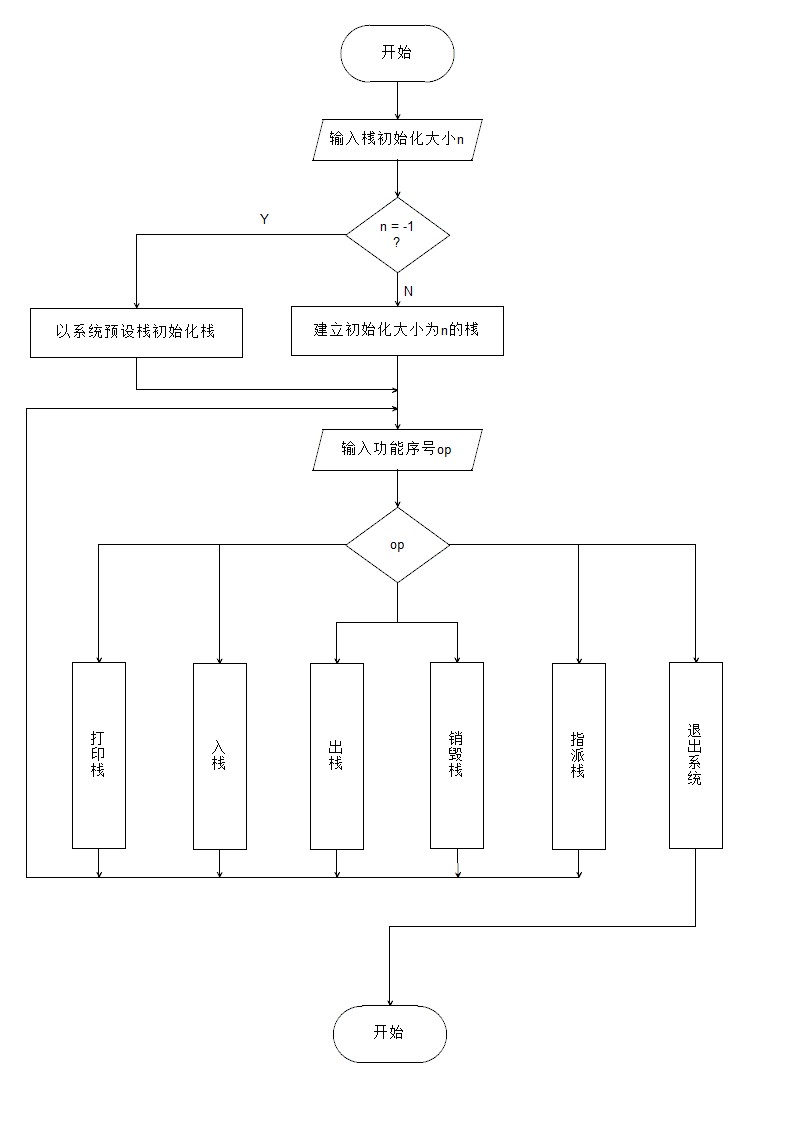


图 2.2 总体流程图

所有操作中由于入栈牵扯到是否需要对栈进行扩容的问题，因此需要设计详实的入栈的算法以应对栈空间不足的情况。入栈的流程如图2.3所示。

即在输入入栈元素e后首先判定栈是否满，若满即对栈基址进行扩容，然后拷贝原栈元素并入栈，否则直接将e入栈。

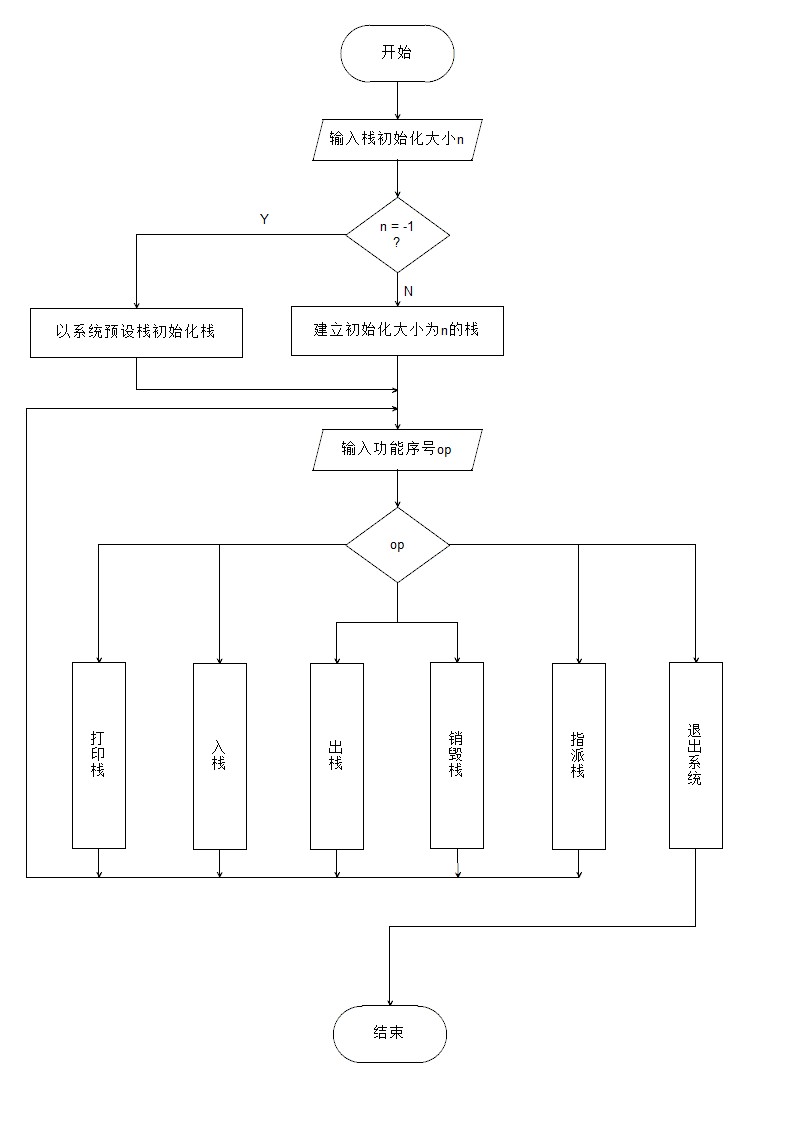


图 2.3 入栈流程图

**2.2详细设计**

1. 初始化模块：

模块功能：引导用户对所使用的栈进行初始化，包括使用整数m和栈s对栈进行初始化。

入口参数：STACK \* const StackUser 用户栈

STACK \* const StackSys 系统预设栈

出口参数：无

业务流程：参考图2.2

对应方法：

* 1. STACK(int m);
  2. STACK(const STACK&s))。

1. 功能选择模块：int ChooseFunc()

模块功能：引导用户选择系统功能，包括打印栈、入栈、出栈、销毁栈、指派栈、退出系统。

入口参数：无

出口参数：用户选择的系统功能

业务流程：参考图2.2

1. 打印栈模块：void Func1(STACK \* const StackUser)

模块功能：打印显示栈的信息。包括栈的容量，已容纳元素个数，已容纳元素具体信息。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. void print() const
  2. int size() const
  3. int howMany() const

1. 入栈模块：void Func2(STACK \* const StackUser)

模块功能：引导用户输入入栈元素，并将之入栈

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. STACK& push(int e);

1. 出栈模块：void Func3(STACK \* const StackUser)

模块功能：出栈并显示出栈元素。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. STACK& pop(int &e);

1. 销毁栈模块：void Func4(STACK \* const StackUser)

模块功能：销毁栈。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. ~STACK();

1. 栈指派模块：void Func5(STACK \* const StackUser)

模块功能：讲系统预设栈指派给用户当前所使用的栈。

入口参数：STACK \* const StackUser

出口参数：STACK \* const StackSys

对应方法：

* 1. STACK& assign(const STACK&s)

**3.软件开发**

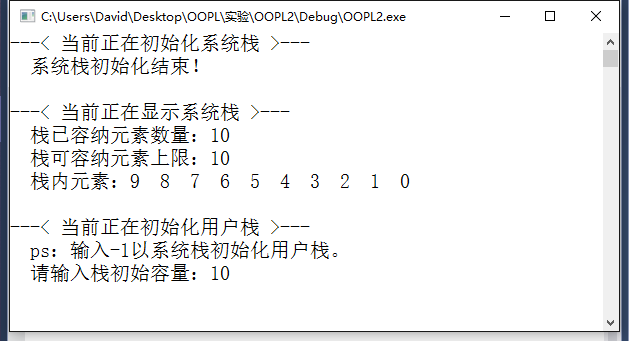
本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 建立初始化大小为10的整型栈。

图2.4 使用整数m初始化栈

1. 打印栈的信息。

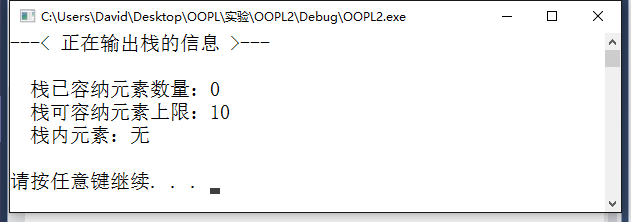


图2.5 打印栈中信息

1. 指派栈。

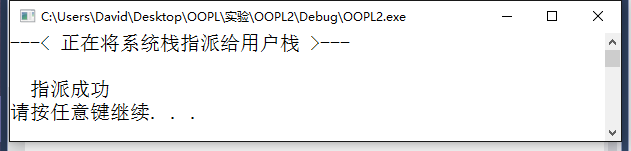


图2.6 指派栈

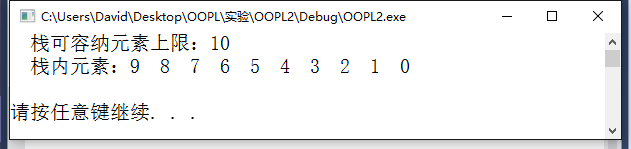


图2.7 打印栈信息

1. 出栈。

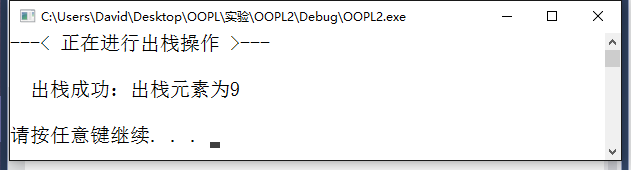


图2.8出栈

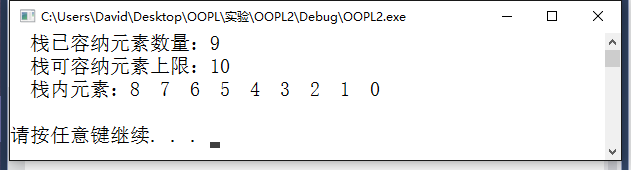


图2.9 打印栈信息

1. 入栈（未满）。

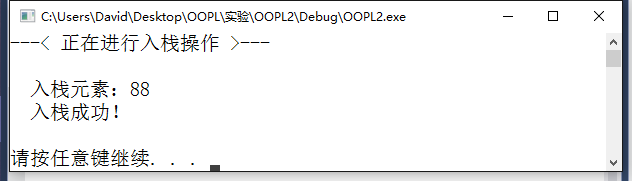


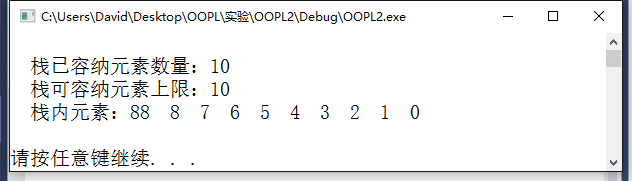
图2.10入栈

图2.11打印栈信息

1. 入栈（已满）。据图可知栈成功地完成了扩容。

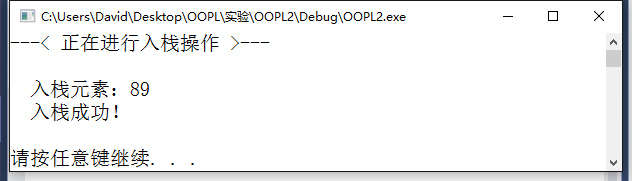


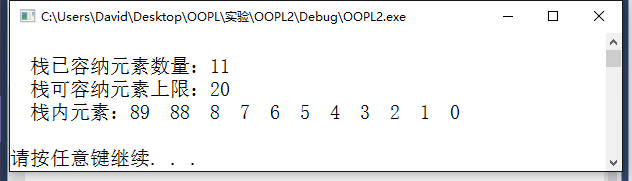
图2.12入栈

图2.13打印栈信息

1. 销毁栈。

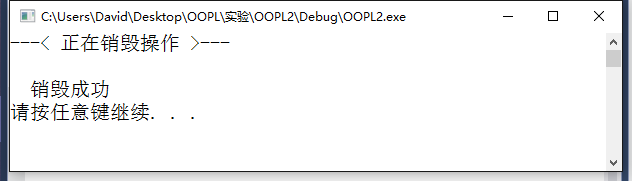


图2.14 销毁栈

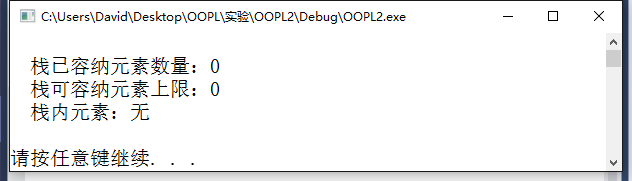


图2.15打印栈信息

1. 初始化桟（使用系统预设栈）

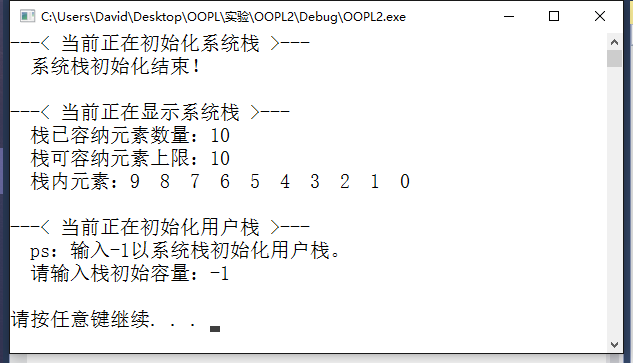


图2.16 初始化栈

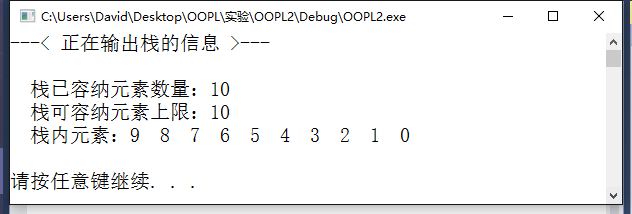


图2.17 打印栈

**5.特点与不足**

**5.1技术特点**

1. 能够做到自动为栈扩容
2. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
3. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. UI设计中有时为了节约篇幅，会采用一些自认为简洁明了的词汇，比如系统栈。实际上系统栈就是系统预设的一个栈，其内已经包含若干元素，方便用户使用它对栈进行初始化，或者assign给用户正在使用的栈。然而UI中并未给出对这个栈的解释，只能由用户体验后自行揣测其含义。这也是设计的一个不合理的地方。
3. 未使用C++特性的流式IO函数，而使用了兼容C的格式化IO函数。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

1. 输入字符流后scanf函数会将字符部分截取，而留下’\n’字符。解决方法：在每个输入函数后加上getchar()函数。
2. 无法在构造函数体内为常成员变量初始化。解决方法：在构造函数体之前使用成员变量(初始值)的方法为其初始化。
3. 无法在函数体中改变常成员变量的值。解决方法：采用变通的类似于\*(int \*)(&this->max)的方法。

**6.2课程设计的体会**

很简单的实验，无非是将对函数的调用切换为面向对象特性的对成员函数（即方法）的调用。这个过程体现了面向对象程序设计中的封装的特性，用户获得的数据都是经过封装的，类似于操作一个黑匣子，这样用户用着方便，程序设计者也可以节省很多对使用者恶意操作的考虑。而且这样做最直接的影响就是类的文件单独放置，而非与传统C语言程序一样各种函数杂糅在一起，使程序结构清晰，便于用户的把握。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL2\Debug下是vs生成的可执行调试文件。

OOPL2\OOPL2下是组成本项目的C++源代码文件。主要包括main.cpp（主文件）、STACK.cpp（类的成员函数的外部定义）、STACK.h（类的成员变量、成员函数的原型、主文件中的部分函数原型）。

**7.2用户使用说明书**

运行~\OPL2\Debug下的OOPL1.exe即可使用本程序。

**7.3源代码**

1. Stack.h

#pragma once

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

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

STACK& push(int e); //将e入栈,并返回栈

STACK& pop(int &e); //出栈到e,并返回栈

STACK& assign(const STACK&s); //赋s给栈,并返回被赋值的栈

void print() const; //打印栈

~STACK(); //销毁栈

};

void SysInitial(STACK\* & StackUser, STACK\* & StackSys);

int ChooseFunc();

void ScreenClr();

void ScreenPause();

void Func1(STACK \* const StackUser); //功能1

void Func2(STACK \* const StackUser); //功能2

void Func3(STACK \* const StackUser); //功能3

void Func4(STACK \* const StackUser); //功能4

void Func5(STACK \* const StackUser, STACK \* const StackSys); //功能5

1. STACK.cpp

#include "STACK.h"

#include <stdio.h>

#define STACK\_INCRE\_STEP 10

STACK::STACK(int m) :max(m), elems(new int[m]), pos(0)

{

if (!this->elems)

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

STACK::STACK(const STACK&s) : max(s.max), elems(new int[s.max]), pos(0)

{

if (this->elems)

{

// 分配成功，拷贝

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

{

push(s.getelem(i));

}

}

else

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

int STACK::size() const

{

return this->max;

}

int STACK::howMany() const

{

return this->pos;

}

int STACK::getelem(int x) const

{

return this->elems[x];

}

STACK& STACK::push(int e)

{

if (this->pos < this->max)

{

this->elems[this->pos++] = e;

}

else

{

int \* EleBase = new int[this->max + STACK\_INCRE\_STEP];

if (EleBase != 0)

{

/\* 拷贝原栈元素并入栈 \*/

int i;

for (i = 0; i < this->pos; i++)

{

EleBase[i] = this->getelem(i);

}

EleBase[i] = e;

/\* 修改栈信息 \*/

delete this->elems;

\*(int \*\*)(&this->elems) = EleBase;

\*(int \*)(&this->max) += STACK\_INCRE\_STEP;

this->pos++;

}

}

return \*(STACK\*)this;

}

STACK& STACK::pop(int &e)

{

e = this->elems[--this->pos];

return \*(STACK\*)this;

}

STACK& STACK::assign(const STACK&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (this->max != s.max)

{

delete this->elems;

\*(int \*\*)(&this->elems) = new int[s.max];

}

if (this->elems == 0)

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

}

else

{

\*(int \*)(&this->max) = s.max;

/\* 拷贝 \*/

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

{

this->elems[i] = s.elems[i];

}

this->pos = s.pos;

}

return \*(STACK\*)this;

}

void STACK::print() const

{

for (int i = this->pos; i > 0; i--)

{

printf("%d ", this->getelem(i - 1));

}

}

STACK::~STACK()

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

if (this->elems) //防止反复析构

{

delete this->elems;

\*(int \*\*)(&this->elems) = 0;

}

}

1. main.cpp

#include "STACK.h"

#include <stdio.h>

#include <stdlib.h>

int main()

{

// variables declaration

STACK \* StackUser; //操作用栈

STACK \* StackSys; //赋值用栈

int op;

//initial

SysInitial(StackUser, StackSys);

//loop

while ((op = ChooseFunc()) != 0)

{

switch (op)

{

case 0:

op = -1;

break;

case 1:

Func1(StackUser);

break;

case 2:

Func2(StackUser);

break;

case 3:

Func3(StackUser);

break;

case 4:

Func4(StackUser);

break;

case 5:

Func5(StackUser, StackSys);

break;

}

}

//quit

ScreenClr();

printf("\n---< 正在退出系统 >---\n");

delete StackUser;

delete StackSys;

ScreenPause();

}

void SysInitial(STACK\* & StackUser, STACK\* & StackSys)

{

//初始化默认栈

int i, n, max, pos;

printf("---< 当前正在初始化系统栈 >---\n");

StackSys = new STACK(10);

for (i = 0; i < 10; i++)

{

StackSys->push(i);

}

printf(" 系统栈初始化结束！\n");

//显示默认栈

printf("\n---< 当前正在显示系统栈 >---\n");

max = StackSys->size();

pos = StackSys->howMany();

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

StackSys->print();

}

printf("\n");

//引导初始用户栈

printf("\n---< 当前正在初始化用户栈 >---\n");

printf(" ps：输入-1以系统栈初始化用户栈。\n");

do

{

printf(" 请输入栈初始容量：");

scanf\_s("%d", &n);

getchar();

if (n == -1)

{

StackUser = new STACK(\*StackSys);

}

else if (n > -1)

{

StackUser = new STACK(n);

printf(" 初始化成功！\n");

}

else

{

printf(" 请输入有效大小！\n");

}

} while (n < -1);

ScreenPause();

}

int ChooseFunc()

{

int n = -1; //用户选择的功能序号

/\* 输出功能选择提示语 \*/

ScreenClr();

printf("---< 选择程序功能 >---\n\n");

printf(" [1].查看栈信息\n");

printf(" [2].入栈操作\n");

printf(" [3].出栈操作\n");

printf(" [4].销毁栈\n");

printf(" [5].将系统栈赋给用户栈\n");

printf(" [0].退出系统\n");

printf("\n请输入功能代码：");

/\* 录入提示语 \*/

scanf\_s("%d", &n);

getchar();

/\* 输入验证 \*/

if (n < 0 || n > 5)

{

n = -1;

printf(" 错误的代码，请重新输入。\n");

ScreenPause();

}

return n;

}

void Func1(STACK \* const StackUser)

{

//initial

int max, pos, i;

//output

ScreenClr();

printf("---< 正在输出栈的信息 >---\n\n");

max = StackUser->size();

pos = StackUser->howMany();

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

StackUser->print();

}

printf("\n");

//wait

ScreenPause();

}

void Func2(STACK \* const StackUser)

{

//initial

int var;

//录入

ScreenClr();

printf("---< 正在进行入栈操作 >---\n\n");

printf(" 入栈元素：");

scanf\_s("%d", &var);

getchar();

StackUser->push(var);

printf(" 入栈成功！\n");

//wait

ScreenPause();

}

void Func3(STACK \* const StackUser)

{

//initial

int var;

//出栈

ScreenClr();

printf("---< 正在进行出栈操作 >---\n\n");

if (StackUser->howMany() < 1)

{

printf(" 出栈失败：栈为空\n");

}

else

{

StackUser->pop(var);

printf(" 出栈成功：出栈元素为%d\n", var);

}

//wait

ScreenPause();

}

void Func4(STACK \* const StackUser)

{

//initial

int var;

//销毁

ScreenClr();

printf("---< 正在销毁操作 >---\n\n");

StackUser->~STACK();

printf(" 销毁成功");

//wait

ScreenPause();

}

void Func5(STACK \* const StackUser, STACK \* const StackSys)

{

//initial

int var;

//指派

ScreenClr();

printf("---< 正在将系统栈指派给用户栈 >---\n\n");

StackUser->assign(\*StackSys);

printf(" 指派成功");

//wait

ScreenPause();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}

实验三.基于算符重载的整型栈编程

**1.需求分析**

**1.1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、出栈等。整型栈类型及其操作函数采用面向对象的C++语言定义，请将完成上述操作的所有函数采用C++编程， 然后写一个main函数对栈的所有操作函数进行测试。

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,并返回栈

virtual STACK& operator>>(int &e); //出栈到e,并返回栈

virtual STACK& operator=(const STACK&s); //赋s给栈,并返回被赋值的栈

virtual void print( ) const; //打印栈

virtual ~STACK( ); //销毁栈

};

**1.2需求分析**

编程要求实现面向对象的使用了运算符重载技术的整型栈，故本项目即实现面向对象的运用了运算符重载技术的整型栈的后台开发，并设计配套UI以方便使用者对已经实现的面向对象的运用了运算符重载技术的整型栈进行使用及测试。

故本项目应包括如下部分：

1. 实现面向对象的运用了运算符重载技术的整型栈的接口
2. 编写配套UI以对上述程序进行测试

**2.系统设计**

**2.1概要设计**

系统的总体结构图如图3.1所示。本项目共由初始化模块，功能选择模块，打印模块，入栈模块，出栈模块及栈指派模块组成。

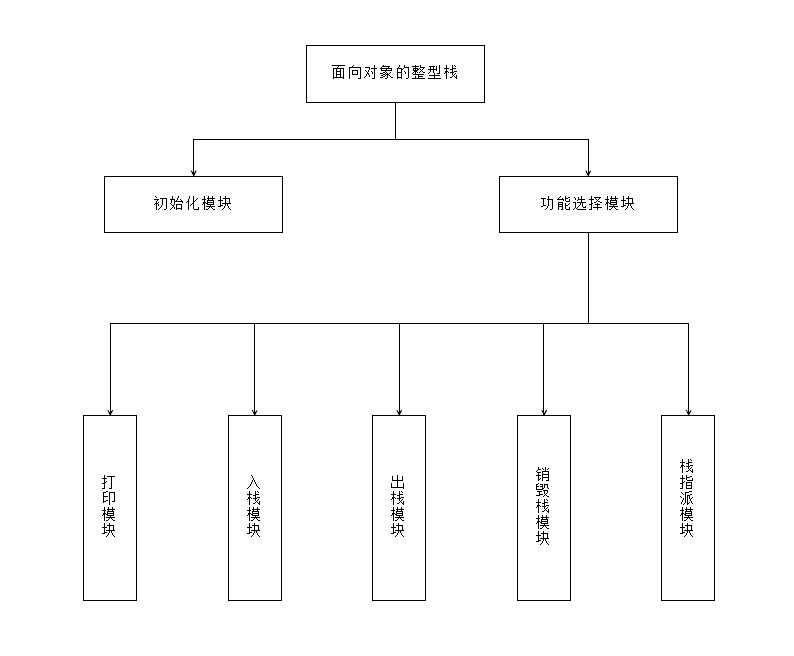


图 3.1 总体结构图

系统的总体流程图如图3.2所示。用户进入系统后应首先输入桟的初始化大小以决定以何种方式初始化栈，即以整数n还是以系统预设栈s初始化栈。

在初始化栈后用户可以通关功能选择模块选择将要执行的功能，特别地，用户在选择退出系统后将清空系统资源并退出本系统。

用户在选择打印栈后系统将展示栈已容纳元素个数，栈可容纳元素个数，栈内所有元素。

用户在选择入栈后系统将引导用户输入想要入栈的元素（整型数），再将用户输入的元素入栈。

用户在选择出栈后系统出栈，并打印出栈元素的值。

用户在选择销毁栈后系统将销毁栈。

用户在选择指派栈后系统会把系统预设栈指派给当前使用的栈。

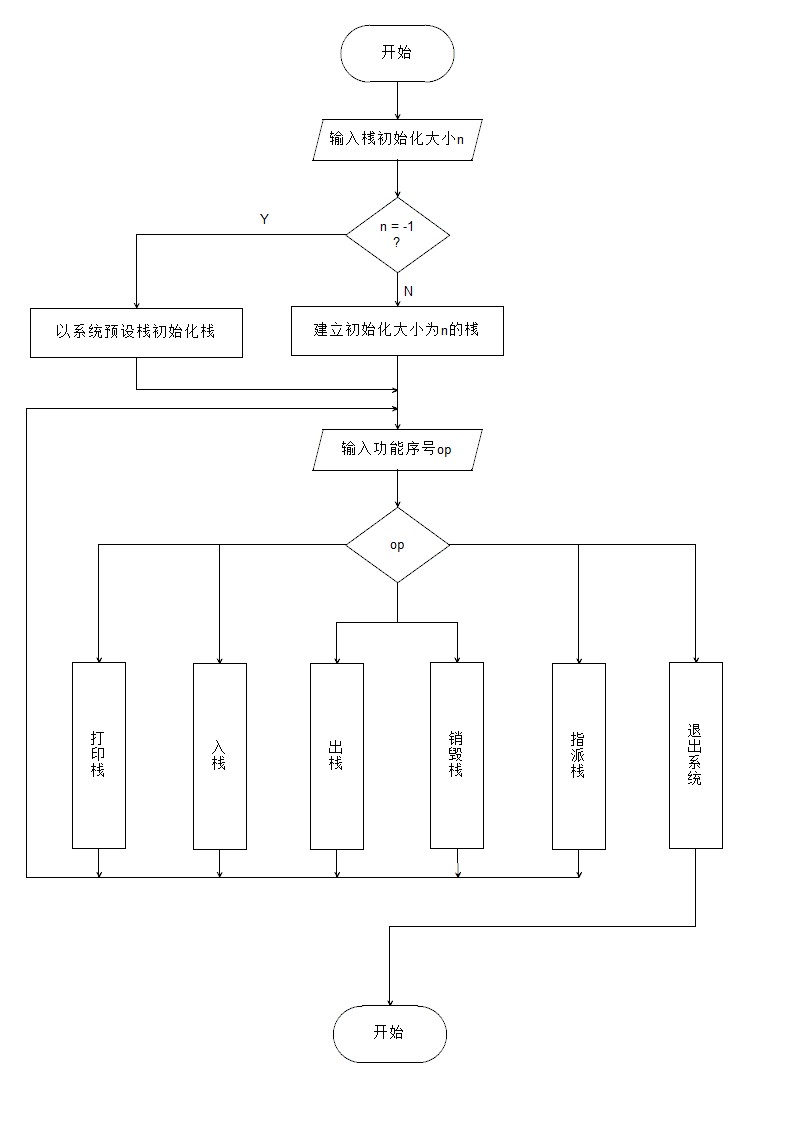


图 3.2 总体流程图

所有操作中由于入栈牵扯到是否需要对栈进行扩容的问题，因此需要设计详实的入栈的算法以应对栈空间不足的情况。入栈的流程如图3.3所示。

即在输入入栈元素e后首先判定栈是否满，若满即对栈基址进行扩容，然后拷贝原栈元素并入栈，否则直接将e入栈。

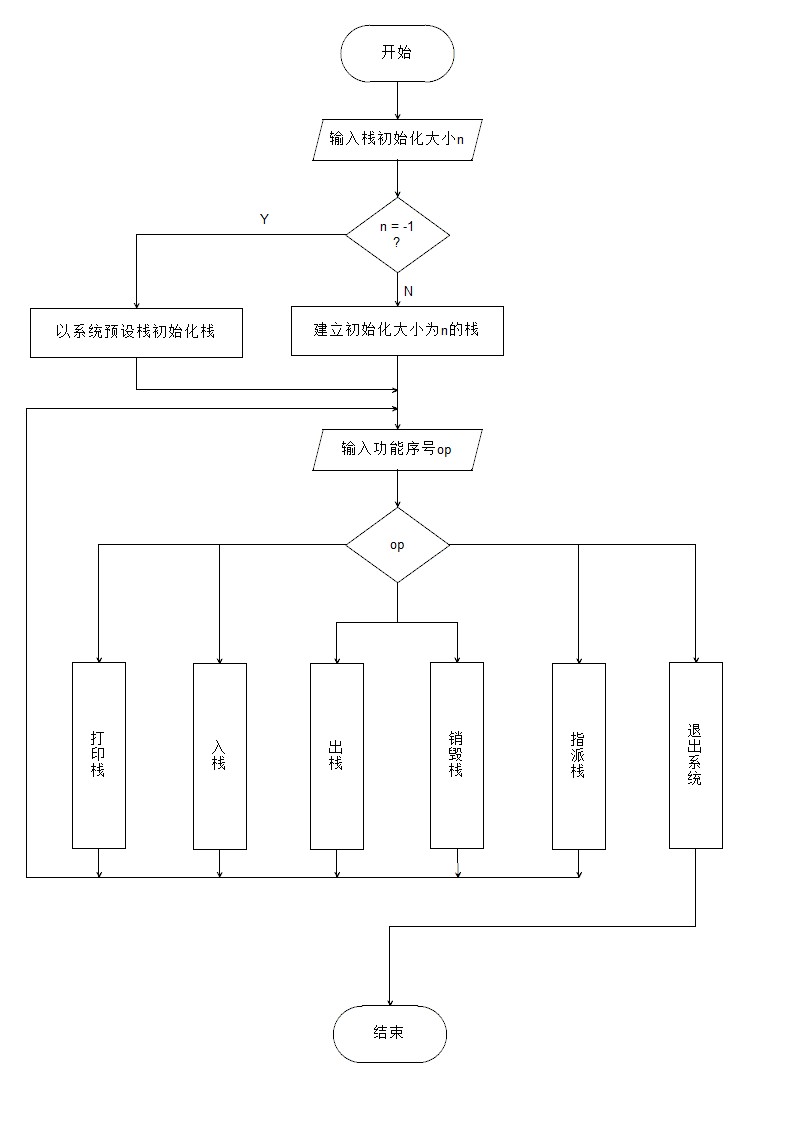


图 3.3 入栈流程图

**2.2详细设计**

1. 初始化模块：

模块功能：引导用户对所使用的栈进行初始化，包括使用整数m和栈s对栈进行初始化。

入口参数：STACK \* const StackUser 用户栈

STACK \* const StackSys 系统预设栈

出口参数：无

业务流程：参考图3.2

对应方法：

* 1. STACK(int m);
  2. STACK(const STACK&s))。

1. 功能选择模块：int ChooseFunc()

模块功能：引导用户选择系统功能，包括打印栈、入栈、出栈、销毁栈、指派栈、退出系统。

入口参数：无

出口参数：用户选择的系统功能

业务流程：参考图3.2

1. 打印栈模块：void Func1(STACK \* const StackUser)

模块功能：打印显示栈的信息。包括栈的容量，已容纳元素个数，已容纳元素具体信息。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. virtual void print( ) const
  2. virtual int size ( ) const
  3. virtual operator int ( ) const

1. 入栈模块：void Func2(STACK \* const StackUser)

模块功能：引导用户输入入栈元素，并将之入栈

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. virtual STACK& operator<<(int e);

1. 出栈模块：void Func3(STACK \* const StackUser)

模块功能：出栈并显示出栈元素。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. virtual STACK& operator>>(int &e);

1. 销毁栈模块：void Func4(STACK \* const StackUser)

模块功能：销毁栈。

入口参数：STACK \* const StackUser

出口参数：无

对应方法：

* 1. virtual ~STACK( );

1. 栈指派模块：void Func5(STACK \* const StackUser)

模块功能：讲系统预设栈指派给用户当前所使用的栈。

入口参数：STACK \* const StackUser

出口参数：STACK \* const StackSys

对应方法：

* 1. virtual STACK& operator=(const STACK&s)

**3.软件开发**

本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 建立初始化大小为10的整型栈。

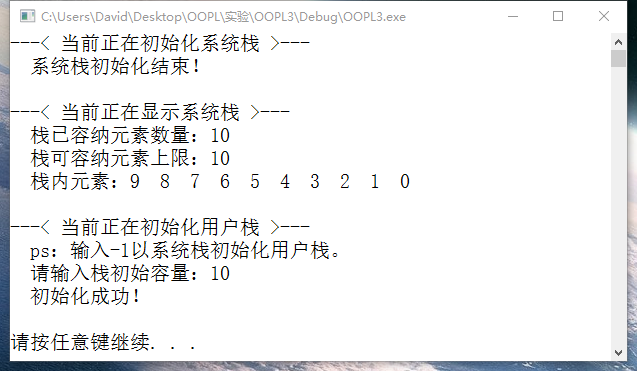


图3.4 使用整数m初始化栈

1. 打印栈的信息。

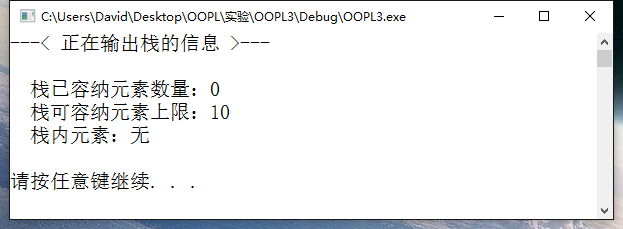


图3.5 打印栈中信息

1. 指派栈。

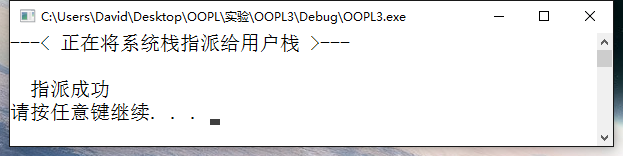


图3.6 指派栈

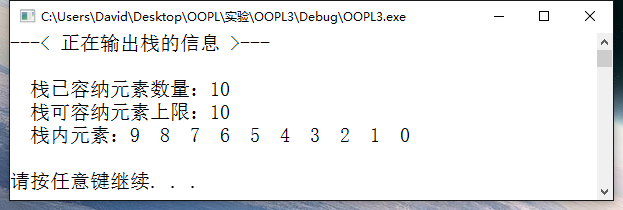


图3.7 打印栈信息

1. 出栈。

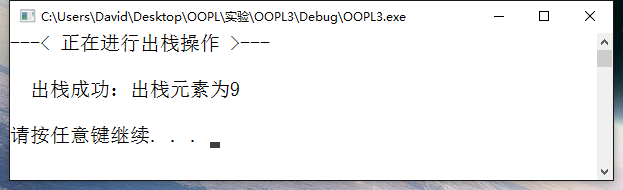


图3.8出栈

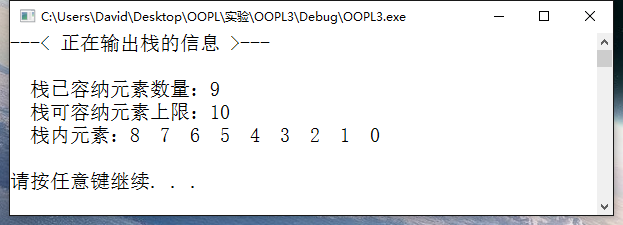


图3.9 打印栈信息

1. 入栈（未满）。

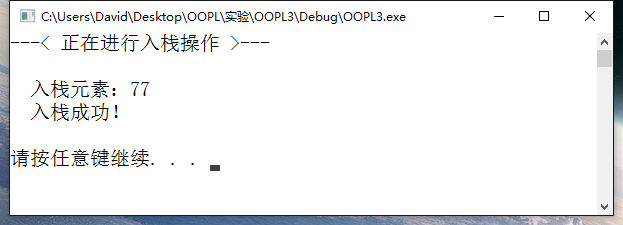


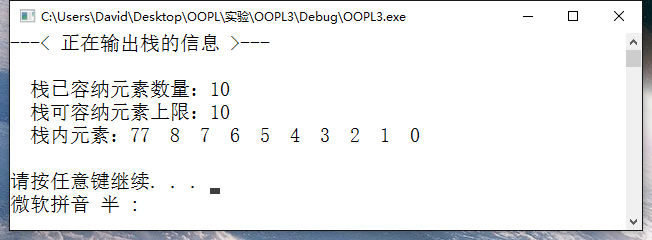
图3.10入栈

图3.11打印栈信息

1. 入栈（已满）。据图可知栈成功地完成了扩容。

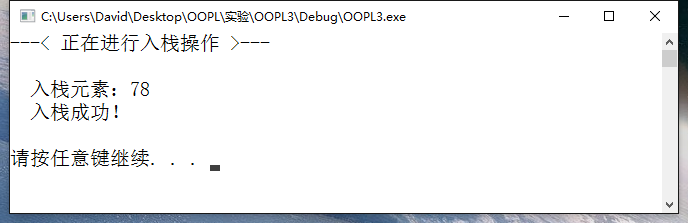


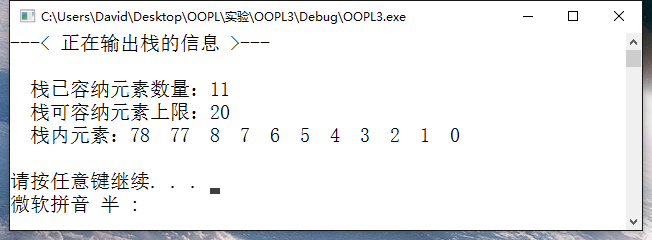
图3.12入栈

图3.13打印栈信息

1. 销毁栈。

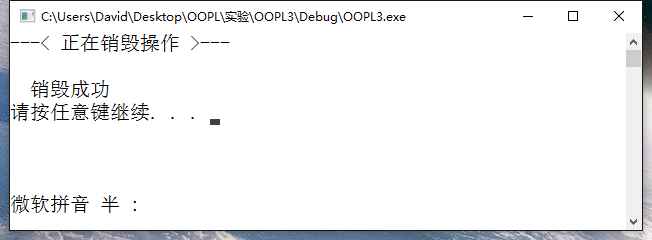


图3.14 销毁栈

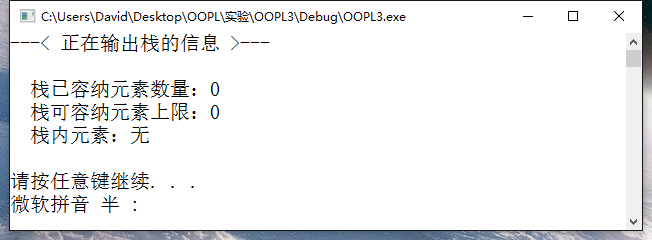


图3.15打印栈信息

1. 初始化桟（使用系统预设栈）

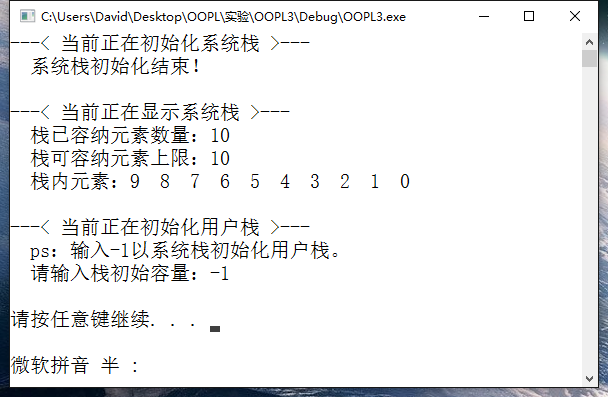


图3.16 初始化栈

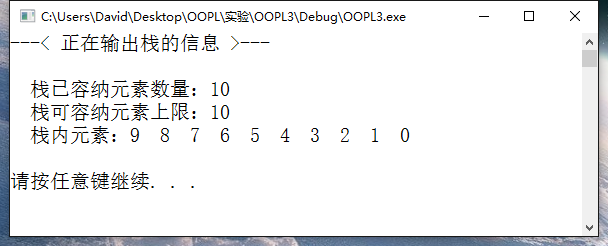


图3.17 打印栈

**5.特点与不足**

**5.1技术特点**

1. 能够做到自动为栈扩容
2. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
3. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. UI设计中有时为了节约篇幅，会采用一些自认为简洁明了的词汇，比如系统栈。实际上系统栈就是系统预设的一个栈，其内已经包含若干元素，方便用户使用它对栈进行初始化，或者assign给用户正在使用的栈。然而UI中并未给出对这个栈的解释，只能由用户体验后自行揣测其含义。这也是设计的一个不合理的地方。
3. 未使用C++特性的流式IO函数，而使用了兼容C的格式化IO函数。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

1. 输入字符流后scanf函数会将字符部分截取，而留下’\n’字符。解决方法：在每个输入函数后加上getchar()函数。
2. 无法在构造函数体内为常成员变量初始化。解决方法：在构造函数体之前使用成员变量(初始值)的方法为其初始化。
3. 无法在函数体中改变常成员变量的值。解决方法：采用变通的类似于\*(int \*)(&this->max)的方法。

**6.2课程设计的体会**

本实验的主要特色在于使用了C++中的运算符重载特性，这是我之前在面向对象程序设计中所从未使用过的。所谓运算符重载无非是换了一种调用成员函数的方法而已，但是这样的设计会使程序更加贴合人类的思维，比如使用一个简单的”=”符号就可以隐式调用成员函数之中的深拷贝，而非之前的调用assign函数，如此便大大提高程序的简洁性也让人更容易理解。

不过运算符重载也会带来一些问题，那就滥用运算符重载带来的问题。首先运算符只是一个符号，其含义可能是不确定的，因此对于用户而言初次使用可能会很强大的运算符重载时会出现很多因为理解不当而出现的问题。依然举例而言，=可以代表浅拷贝，也可以代表深拷贝，具体代表什么含义还需要了解程序的代码，或是由程序的作者给出文档解释。但是如果使用了形如DeepCopy()的方法名便可以很好地规避这个问题。

总而言之，运算符重载是一个很强大的工具，如果用得好的话。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL3\Debug下是vs生成的可执行调试文件。

OOPL3\OOPL3下是组成本项目的C++源代码文件。主要包括main.cpp（主文件）、STACK.cpp（类的成员函数的外部定义）、STACK.h（类的成员变量、成员函数的原型）、MyHeader.h（主文件中的部分函数原型）。

**7.2用户使用说明书**

运行~\OPL3\Debug下的OOPL3.exe即可使用本程序。

**7.3源代码**

1. Stack.h

#pragma once

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,并返回栈

virtual STACK& operator>>(int &e); //出栈到e,并返回栈

virtual STACK& operator=(const STACK&s); //赋s给栈,并返回被赋值的栈

virtual void print( ) const; //打印栈

virtual ~STACK( ); //销毁栈

};

1. STACK.cpp

#include "STACK.h"

#include <stdio.h>

#define STACK\_INCRE\_STEP 10

STACK::STACK(int m):max(m), elems(new int[m]), pos(0)

{

if (!this->elems)

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

STACK::STACK(const STACK&s) : max(s.max), elems(new int[s.max]), pos(0)

{

if (this->elems)

{

// 分配成功，拷贝

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

{

(\*this)<<s[i];

}

}

else

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

int STACK::size() const

{

return this->max;

}

STACK::operator int ( ) const

{

return this->pos;

}

int STACK::operator[ ] (int x) const

{

return this->elems[x];

}

STACK& STACK::operator<<(int e)

{

if (this->pos < this->max)

{

this->elems[this->pos++] = e;

}

else

{

int \* EleBase = new int[this->max + STACK\_INCRE\_STEP];

if (EleBase != 0)

{

/\* 拷贝原栈元素并入栈 \*/

int i;

for (i = 0; i < this->pos; i++)

{

EleBase[i] = (\*this)[i];

}

EleBase[i] = e;

/\* 修改栈信息 \*/

delete this->elems;

\*(int \*\*)(&this->elems) = EleBase;

\*(int \*)(&this->max) += STACK\_INCRE\_STEP;

this->pos++;

}

}

return \*(STACK\*)this;

}

STACK& STACK::operator>>(int &e)

{

e = (\*this)[--this->pos];

return \*(STACK\*)this;

}

STACK& STACK::operator=(const STACK&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (this->max != s.max)

{

delete this->elems;

\*(int \*\*)(&this->elems) = new int[s.max];

}

if (this->elems == 0)

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

}

else

{

\*(int \*)(&this->max) = s.max;

/\* 拷贝 \*/

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

{

this->elems[i] = s[i];

}

this->pos = s.pos;

}

return \*(STACK\*)this;

}

void STACK::print() const

{

for (int i = this->pos; i > 0; i--)

{

printf("%d ", (\*this)[i - 1]);

}

}

STACK::~STACK(void)

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

if (this->elems) //防止反复析构

{

delete this->elems;

\*(int \*\*)(&this->elems) = 0;

}

}

1. MyHeader.h

#pragma once

#include "STACK.h"

void SysInitial(STACK\* & StackUser, STACK\* & StackSys);

int ChooseFunc();

void ScreenClr();

void ScreenPause();

void Func1(STACK \* const StackUser); //功能1

void Func2(STACK \* const StackUser); //功能2

void Func3(STACK \* const StackUser); //功能3

void Func4(STACK \* const StackUser); //功能4

void Func5(STACK \* const StackUser, STACK \* const StackSys); //功能5

1. Main.cpp

#include "STACK.h"

#include <stdio.h>

#include <stdlib.h>

#include "MyHeader.h"

int main()

{

// variables declaration

STACK \* StackUser; //操作用栈

STACK \* StackSys; //赋值用栈

int op;

//initial

SysInitial(StackUser, StackSys);

//loop

while ((op = ChooseFunc()) != 0)

{

switch (op)

{

case 0:

op = -1;

break;

case 1:

Func1(StackUser);

break;

case 2:

Func2(StackUser);

break;

case 3:

Func3(StackUser);

break;

case 4:

Func4(StackUser);

break;

case 5:

Func5(StackUser, StackSys);

break;

}

}

//quit

ScreenClr();

printf("\n---< 正在退出系统 >---\n");

delete StackUser;

delete StackSys;

ScreenPause();

}

void SysInitial(STACK\* & StackUser, STACK\* & StackSys)

{

//初始化默认栈

int i, n, max, pos;

printf("---< 当前正在初始化系统栈 >---\n");

StackSys = new STACK(10);

for (i = 0; i < 10; i++)

{

(\*StackSys)<<i;

}

printf(" 系统栈初始化结束！\n");

//显示默认栈

printf("\n---< 当前正在显示系统栈 >---\n");

max = StackSys->size();

pos = \*StackSys;

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

StackSys->print();

}

printf("\n");

//引导初始用户栈

printf("\n---< 当前正在初始化用户栈 >---\n");

printf(" ps：输入-1以系统栈初始化用户栈。\n");

do

{

printf(" 请输入栈初始容量：");

scanf\_s("%d", &n);

getchar();

if (n == -1)

{

StackUser = new STACK(\*StackSys);

}

else if (n > -1)

{

StackUser = new STACK(n);

printf(" 初始化成功！\n");

}

else

{

printf(" 请输入有效大小！\n");

}

} while (n < -1);

ScreenPause();

}

int ChooseFunc()

{

int n = -1; //用户选择的功能序号

/\* 输出功能选择提示语 \*/

ScreenClr();

printf("---< 选择程序功能 >---\n\n");

printf(" [1].查看栈信息\n");

printf(" [2].入栈操作\n");

printf(" [3].出栈操作\n");

printf(" [4].销毁栈\n");

printf(" [5].将系统栈赋给用户栈\n");

printf(" [0].退出系统\n");

printf("\n请输入功能代码：");

/\* 录入提示语 \*/

scanf\_s("%d", &n);

getchar();

/\* 输入验证 \*/

if (n < 0 || n > 5)

{

n = -1;

printf(" 错误的代码，请重新输入。\n");

ScreenPause();

}

return n;

}

void Func1(STACK \* const StackUser)

{

//initial

int max, pos, i;

//output

ScreenClr();

printf("---< 正在输出栈的信息 >---\n\n");

max = StackUser->size();

pos = \*StackUser;

printf(" 栈已容纳元素数量：%d\n", pos);

printf(" 栈可容纳元素上限：%d\n", max);

printf(" 栈内元素：");

if (pos == 0)

{

printf("无");

}

else

{

StackUser->print();

}

printf("\n");

//wait

ScreenPause();

}

void Func2(STACK \* const StackUser)

{

//initial

int var;

//录入

ScreenClr();

printf("---< 正在进行入栈操作 >---\n\n");

printf(" 入栈元素：");

scanf\_s("%d", &var);

getchar();

(\*StackUser)<<var;

printf(" 入栈成功！\n");

//wait

ScreenPause();

}

void Func3(STACK \* const StackUser)

{

//initial

int var;

//出栈

ScreenClr();

printf("---< 正在进行出栈操作 >---\n\n");

if (\*StackUser < 1)

{

printf(" 出栈失败：栈为空\n");

}

else

{

\*StackUser>>var;

printf(" 出栈成功：出栈元素为%d\n", var);

}

//wait

ScreenPause();

}

void Func4(STACK \* const StackUser)

{

//initial

int var;

//销毁

ScreenClr();

printf("---< 正在销毁操作 >---\n\n");

StackUser->~STACK();

printf(" 销毁成功");

//wait

ScreenPause();

}

void Func5(STACK \* const StackUser, STACK \* const StackSys)

{

//initial

int var;

//指派

ScreenClr();

printf("---< 正在将系统栈指派给用户栈 >---\n\n");

\*StackUser=\*StackSys;

printf(" 指派成功");

//wait

ScreenPause();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}

实验四.面向对象的整型队列编程

**1.需求分析**

**1.1题目要求**

整型队列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队列是否为空、向队列顶添加一个整型元素、出队列等。整型队列类型及其操作函数采用面向对象的C++语言定义，请将完成上述操作的所有函数采用C++编程， 然后写一个main函数对队列的所有操作函数进行测试。

class QUEUE{

int \*const elems; //申请内存用于存放队列的元素

const int max; //队列能存放的最大元素个数

int head, tail; //队列头和尾，队列空时head=tail;初始时head=tail=0

public:

QUEUE(int m); //初始化队列：最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

virtual operator int ( ) const; //返回队列的实际元素个数

virtual QUEUE& operator<<(int e); //将e入队列,并返回队列

virtual QUEUE& operator>>(int &e); //出队列到e,并返回队列

virtual QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

virtual void print( ) const; //打印队列

virtual ~QUEUE( ); //销毁队列

};

**1.2需求分析**

编程要求实现面向对象的使用了运算符重载技术的整型队列，故本项目即实现面向对象的运用了运算符重载技术的整型队列的后台开发，并设计配套UI以方便使用者对已经实现的面向对象的运用了运算符重载技术的整型队列进行使用及测试。

故本项目应包括如下部分：

1. 实现面向对象的运用了运算符重载技术的整型队列的接口
2. 编写配套UI以对上述程序进行测试

**2.系统设计**

**2.1概要设计**

系统的总体结构图如图4.1所示。本项目共由初始化模块，功能选择模块，打印模块，入队模块，出队模块及队指派模块组成。

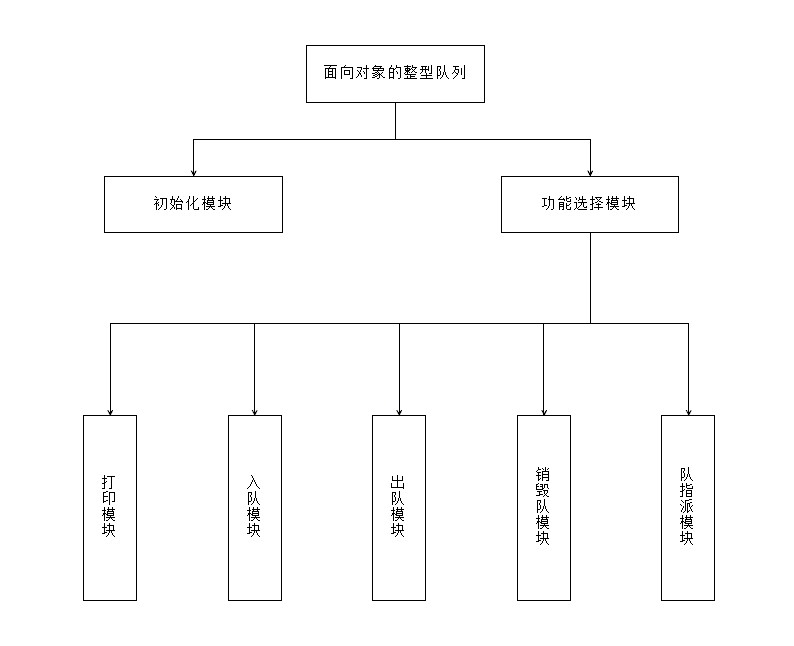


图 4.1 总体结构图

系统的总体流程图如图4.2所示。用户进入系统后应首先输入桟的初始化大小以决定以何种方式初始化队列，即以整数n还是以系统预设队列q初始化队。

在初始化队后用户可以通关功能选择模块选择将要执行的功能，特别地，用户在选择退出系统后将清空系统资源并退出本系统。

用户在选择打印队列后系统将展示队列已容纳元素个数，队列可容纳元素个数，队列内所有元素。

用户在选择入队后系统将引导用户输入想要入队的元素（整型数），再将用户输入的元素入队。

用户在选择出队后系统出队，并打印出队元素的值。

用户在选择销毁队列后系统将销毁队。

用户在选择指派队列后系统会把系统预设队指派给当前使用的队。

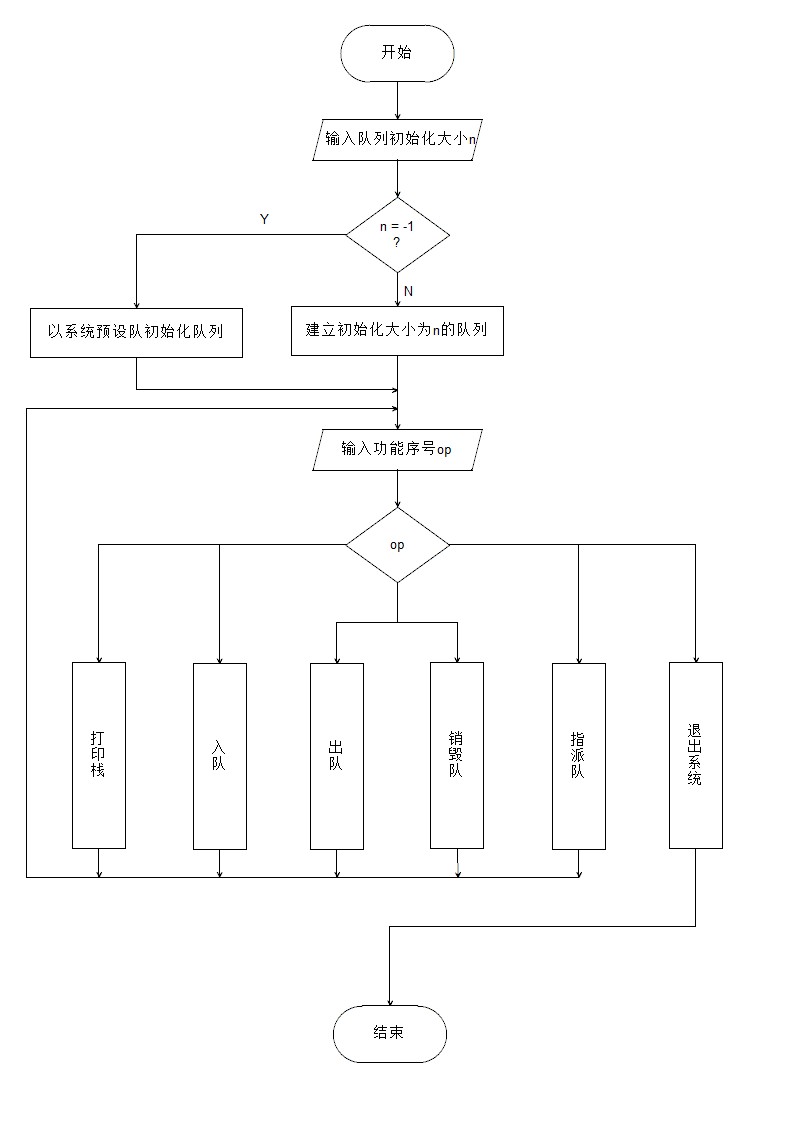


图 4.2 总体流程图

系统中入队操作由于每个人对队列的结构的理解不同，因此实现起来可能有所不同，因此本处着重介绍入队的算法。

首先介绍本实验中采用队列的物理结构。

本队列的物理结构是线性表，当表的容量为n+1可容纳n(max)个元素，队首指针(head)永远指向队首元素，队尾指针(tail)永远指向队尾后的空位。显然当队首指针等于队尾指针时队内元素的个数为0。

特别地，本队是一个循环队列。因此队内元素的个数可表示为：

有了如上的预备，则下面图4.3所示的算法流程就很容易实现了。

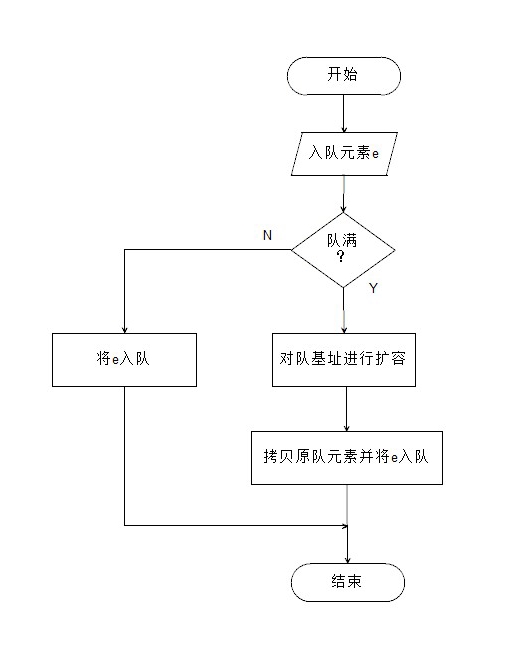


图 4.3 入队流程图

**2.2详细设计**

1. 初始化模块：void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys);

模块功能：引导用户对所使用的队列进行初始化，包括使用整数m和系统预设队q对用户所使用的队列进行初始化。

入口参数：QUEUE\* & QUEUEUser 用户使用队

QUEUE\* & QUEUESys 系统预设队

出口参数：无

业务流程：参考图4.2

对应方法：

* 1. QUEUE(int m)
  2. QUEUE(const QUEUE&s);

1. 功能选择模块：int ChooseFunc()

模块功能：引导用户选择系统功能，包括打印队、入队、出队、销毁队、指派队、退出系统。

入口参数：无

出口参数：用户选择的系统功能

业务流程：参考图4.2

1. 打印队模块：void Func1(QUEUE \* const QUEUEUser)

模块功能：打印显示队的信息。包括队的容量，已容纳元素个数，已容纳元素具体信息。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual void print( ) const
  2. virtual int size ( ) const
  3. virtual operator int ( ) const

1. 入队模块：void Func2(QUEUE \* const QUEUEUser)

模块功能：引导用户输入入队元素，并将之入队

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual QUEUE & operator<<(int e);

1. 出队模块：void Func3(QUEUE \* const QUEUEUser)

模块功能：出队并显示出队元素。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual QUEUE & operator>>(int &e);

1. 销毁队模块：void Func4(QUEUE \* const QUEUEUser)

模块功能：销毁队。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual ~ QUEUE ( );

1. 队指派模块：void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys)

模块功能：讲系统预设队指派给用户当前所使用的队。

入口参数：QUEUE \* const QUEUEUser

出口参数：QUEUE \* const QUEUESys

对应方法：

* 1. virtual QUEUE & operator=(const QUEUE &s)

**3.软件开发**

本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 建立初始化大小为10的整型队列

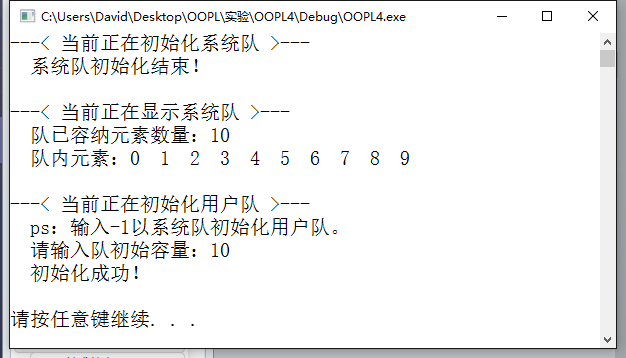


图4.4 初始化

1. 打印队列

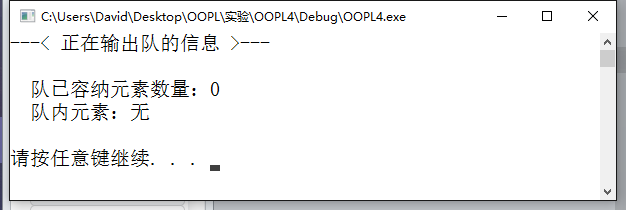


图4.5 打印队列

1. 将系统队赋值给当前使用的队列。

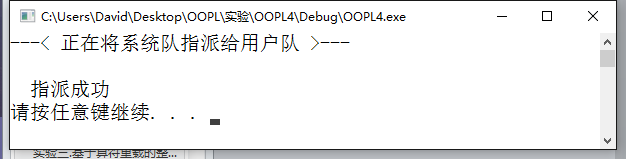


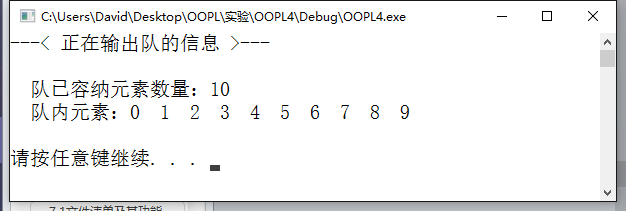
图4.6 指派队列

图4.7 打印队列

1. 出队

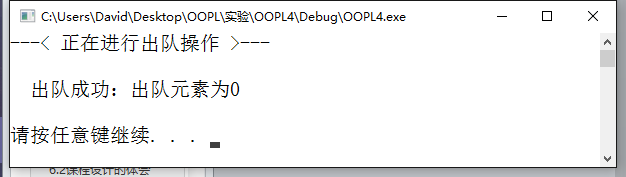


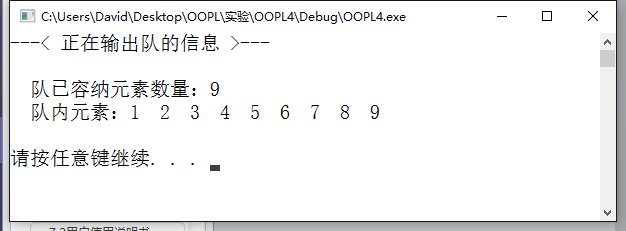
图4.8 出队

图4.9 打印队列

1. 入队（未满）

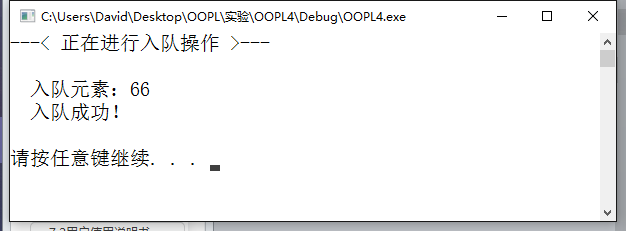


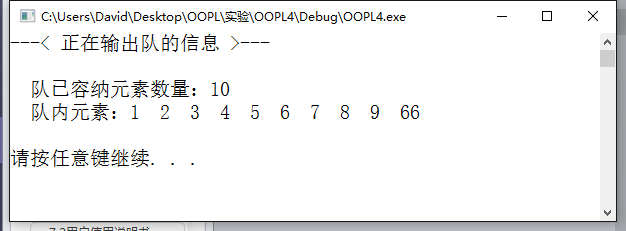
图4.10 入队

图4.11 打印队列

1. 入队（满）

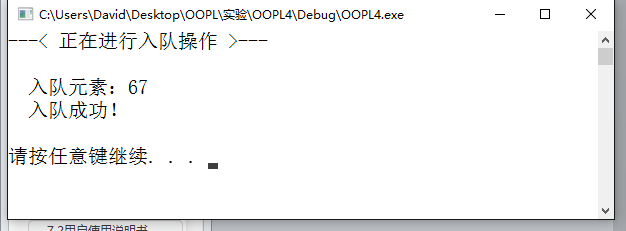


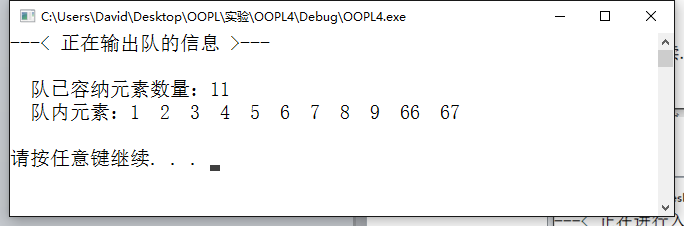
图4.12入队

图4.13打印队列

1. 销毁队

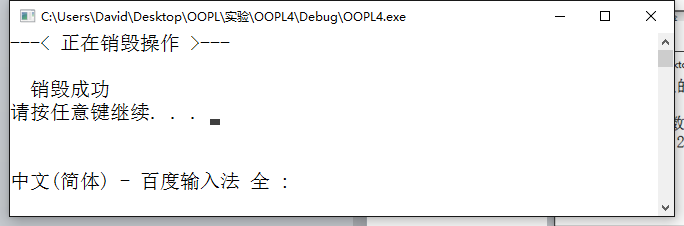


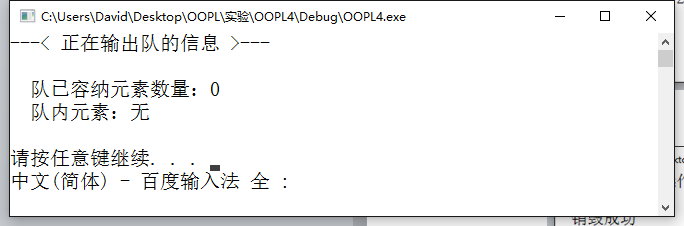
图4.14 销毁队

图4.15 打印队列

1. 使用系统预设队初始化队列

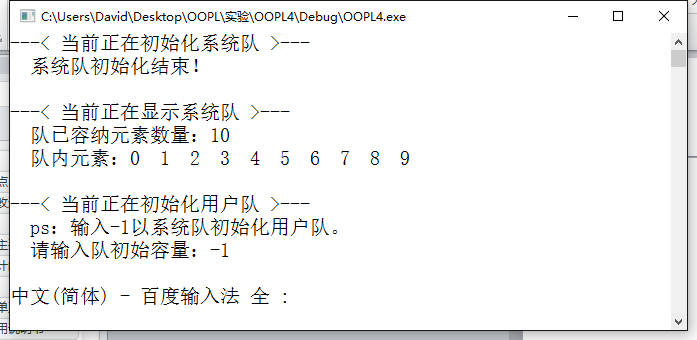


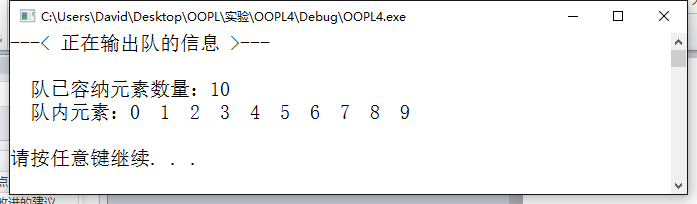
图4.14 销毁队

图4.15 打印队列

**5.特点与不足**

**5.1技术特点**

1. 能够做到自动为队列扩容，每次扩容的梯度是10
2. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
3. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. UI设计中有时为了节约篇幅，会采用一些自认为简洁明了的词汇，比如系统栈。实际上系统栈就是系统预设的一个栈，其内已经包含若干元素，方便用户使用它对栈进行初始化，或者assign给用户正在使用的栈。然而UI中并未给出对这个栈的解释，只能由用户体验后自行揣测其含义。这也是设计的一个不合理的地方。
3. 未使用C++特性的流式IO函数，而使用了兼容C的格式化IO函数。
4. 由于未提供显示队列容量的接口，因此无法显示队列的容量，使得测试结果缺乏说服力。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

1. 输入字符流后scanf函数会将字符部分截取，而留下’\n’字符。解决方法：在每个输入函数后加上getchar()函数。
2. 无法在构造函数体内为常成员变量初始化。解决方法：在构造函数体之前使用成员变量(初始值)的方法为其初始化。
3. 无法在函数体中改变常成员变量的值。解决方法：采用变通的类似于\*(int \*)(&this->max)的方法。
4. 在使用delete运算符后发生内存泄漏。解决方法：检查运行逻辑，确定在使用分配的内存的时候没有发生指针越界。在本程序中将elems(new int[s.max])修改为elems(new int[s.max+1])便解决了为队列扩容时delete旧地址空间而发生了内存泄漏的问题。

**6.2课程设计的体会**

本实验旨于实现面向对象的整型队列，主要难点在于循环队列的理解与边界的控制。在知识点的预备上，个人觉得与实验三没有什么不同。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL4\Debug下是vs生成的可执行调试文件。

OOPL4\OOPL4下是组成本项目的C++源代码文件。主要包括main.cpp（主文件）、QUEUE.cpp（类的成员函数的外部定义）、QUEUE.h（类的成员变量、成员函数的原型）、MyHeader.h（主文件中的部分函数原型）。

**7.2用户使用说明书**

运行~\OOPL4\Debug下的OOPL4.exe即可使用本程序。

**7.3源代码**

1. QUEUE.h

#pragma once

//最多元素m

//数组长度m+1

//tail处无元素

class QUEUE{

int \*const elems; //申请内存用于存放队列的元素

const int max; //队列能存放的最大元素个数

int head, tail; //队列头和尾，队列空时head=tail;初始时head=tail=0

public:

QUEUE(int m); //初始化队列：最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

virtual operator int ( ) const; //返回队列的实际元素个数

virtual QUEUE& operator<<(int e); //将e入队列,并返回队列

virtual QUEUE& operator>>(int &e); //出队列到e,并返回队列

virtual QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

virtual void print( ) const; //打印队列

virtual ~QUEUE( ); //销毁队列

};

1. MyHeader.h

#pragma once

#include "QUEUE.h"

void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys);

int ChooseFunc();

void ScreenClr();

void ScreenPause();

void Func1(QUEUE \* const QUEUEUser); //功能1

void Func2(QUEUE \* const QUEUEUser); //功能2

void Func3(QUEUE \* const QUEUEUser); //功能3

void Func4(QUEUE \* const QUEUEUser); //功能4

void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys); //功能5

1. QUEUE.cpp

#include "QUEUE.h"

#include <math.h>

#include <stdio.h>

#define QUEUE\_INCRE\_STEP 10

//最多元素m

//数组长度m+1

//tail处无元素

QUEUE::QUEUE(int m) :max(m), elems(new int[m + 1]), head(0), tail(0)

{

}

QUEUE::QUEUE(const QUEUE&s) : max(s.max), elems(new int[s.max+1]), head(0), tail(0)

{

if (this->elems)

{

// 分配成功，拷贝

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

{

(\*this) << s.elems[i];

}

head = s.head;

tail = s.tail;

}

else

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

QUEUE::operator int() const

{

return (tail - head + this->max + 1) % (this->max + 1);

}

QUEUE& QUEUE::operator<<(int e)

{

if (\*this < this->max)

{

this->elems[tail] = e;

this->tail = (this->tail + 1) % (this->max + 1);

}

else

{

int \* EleBase = new int[this->max + QUEUE\_INCRE\_STEP + 1];

if (EleBase != 0)

{

/\* 拷贝原栈元素并入队 \*/

int i = 0;

for (i = 0; i < \*this; i++)

{

EleBase[i] = this->elems[(head + i) % (this->max + 1)];

}

EleBase[i++] = e;

/\* 修改栈信息 \*/

this->head = 0;

this->tail = i;

delete this->elems;

\*(int \*\*)(&this->elems) = EleBase;

\*(int \*)(&this->max) += QUEUE\_INCRE\_STEP;

}

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator>>(int &e)

{

if (head != tail)

{

e = this->elems[head];

this->head = (this->head + 1) % (this->max + 1);

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator=(const QUEUE&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (this->max != s.max)

{

delete this->elems;

\*(int \*\*)(&this->elems) = new int[s.max + 1];

}

if (this->elems == 0)

{

\*(int \*)(&this->max) = 0;

this->head = 0;

this->tail = 0;

}

else

{

/\* 拷贝 \*/

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

{

this->elems[(s.head + i) % (s.max + 1)] = s.elems[(s.head + i) % (s.max + 1)];

}

//修改信息

this->head = s.head;

this->tail = s.tail;

\*(int \*)(&this->max) = s.max;

}

return \*(QUEUE\*)this;

}

void QUEUE::print() const

{

for (int i = 0; i < \*this; i++)

{

printf("%d ", this->elems[(head + i) % (max + 1)]);

}

}

QUEUE::~QUEUE(void)

{

\*(int \*)(&this->max) = 0;

this->head = 0;

this->tail = 0;

if (this->elems) //防止反复析构

{

delete this->elems;

\*(int \*\*)(&this->elems) = 0;

}

}

1. main.cpp

#include "QUEUE.h"

#include <stdio.h>

#include <stdlib.h>

#include "MyHeader.h"

int main()

{

// variables declaration

QUEUE \* QUEUEUser; //操作用队

QUEUE \* QUEUESys; //赋值用队

int op;

//initial

SysInitial(QUEUEUser, QUEUESys);

//loop

while ((op = ChooseFunc()) != 0)

{

switch (op)

{

case 0:

op = -1;

break;

case 1:

Func1(QUEUEUser);

break;

case 2:

Func2(QUEUEUser);

break;

case 3:

Func3(QUEUEUser);

break;

case 4:

Func4(QUEUEUser);

break;

case 5:

Func5(QUEUEUser, QUEUESys);

break;

}

}

//quit

ScreenClr();

printf("\n---< 正在退出系统 >---\n");

delete QUEUEUser;

delete QUEUESys;

ScreenPause();

}

void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys)

{

//初始化默认队

int i, n, num;

printf("---< 当前正在初始化系统队 >---\n");

QUEUESys = new QUEUE(10);

for (i = 0; i < 10; i++)

{

(\*QUEUESys)<<i;

}

printf(" 系统队初始化结束！\n");

//显示默认队

printf("\n---< 当前正在显示系统队 >---\n");

num = \*QUEUESys;

printf(" 队已容纳元素数量：%d\n", num);

printf(" 队内元素：");

if (num == 0)

{

printf("无");

}

else

{

QUEUESys->print();

}

printf("\n");

//引导初始用户队

printf("\n---< 当前正在初始化用户队 >---\n");

printf(" ps：输入-1以系统队初始化用户队。\n");

do

{

printf(" 请输入队初始容量：");

scanf\_s("%d", &n);

getchar();

if (n == -1)

{

QUEUEUser = new QUEUE(\*QUEUESys);

}

else if (n > -1)

{

QUEUEUser = new QUEUE(n);

printf(" 初始化成功！\n");

}

else

{

printf(" 请输入有效大小！\n");

}

} while (n < -1);

ScreenPause();

}

int ChooseFunc()

{

int n = -1; //用户选择的功能序号

/\* 输出功能选择提示语 \*/

ScreenClr();

printf("---< 选择程序功能 >---\n\n");

printf(" [1].查看队信息\n");

printf(" [2].入队操作\n");

printf(" [3].出队操作\n");

printf(" [4].销毁队\n");

printf(" [5].将系统队赋给用户队\n");

printf(" [0].退出系统\n");

printf("\n请输入功能代码：");

/\* 录入提示语 \*/

scanf\_s("%d", &n);

getchar();

/\* 输入验证 \*/

if (n < 0 || n > 5)

{

n = -1;

printf(" 错误的代码，请重新输入。\n");

ScreenPause();

}

return n;

}

void Func1(QUEUE \* const QUEUEUser)

{

//initial

int max, pos, i;

//output

ScreenClr();

printf("---< 正在输出队的信息 >---\n\n");

pos = \*QUEUEUser;

printf(" 队已容纳元素数量：%d\n", pos);

printf(" 队内元素：");

if (pos == 0)

{

printf("无");

}

else

{

QUEUEUser->print();

}

printf("\n");

//wait

ScreenPause();

}

void Func2(QUEUE \* const QUEUEUser)

{

//initial

int var;

//录入

ScreenClr();

printf("---< 正在进行入队操作 >---\n\n");

printf(" 入队元素：");

scanf\_s("%d", &var);

getchar();

(\*QUEUEUser)<<var;

printf(" 入队成功！\n");

//wait

ScreenPause();

}

void Func3(QUEUE \* const QUEUEUser)

{

//initial

int var;

//出队

ScreenClr();

printf("---< 正在进行出队操作 >---\n\n");

if (\*QUEUEUser < 1)

{

printf(" 出队失败：队为空\n");

}

else

{

\*QUEUEUser>>var;

printf(" 出队成功：出队元素为%d\n", var);

}

//wait

ScreenPause();

}

void Func4(QUEUE \* const QUEUEUser)

{

//initial

int var;

//销毁

ScreenClr();

printf("---< 正在销毁操作 >---\n\n");

QUEUEUser->~QUEUE();

printf(" 销毁成功");

//wait

ScreenPause();

}

void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys)

{

//initial

int var;

//指派

ScreenClr();

printf("---< 正在将系统队指派给用户队 >---\n\n");

\*QUEUEUser=\*QUEUESys;

printf(" 指派成功");

//wait

ScreenPause();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}

实验五.基于组合的整型队列编程

**1.需求分析**

**1.1题目要求**

整型队列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队列是否为空、向队列顶添加一个整型元素、出队列等。整型队列类型及其操作函数采用面向对象的C++语言定义，请将完成上述操作的所有函数采用C++编程， 然后写一个main函数对队列的所有操作函数进行测试。注意，请用实验三的SATCK组合形成新的类QUEQUE。请说明如果删掉virtual有什么不同。

class QUEUE{

STACK s1, s2;

public:

QUEUE(int m); //初始化队列：每个栈最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

virtual operator int ( ) const; //返回队列的实际元素个数

virtual QUEUE& operator<<(int e); //将e入队列,并返回队列

virtual QUEUE& operator>>(int &e); //出队列到e,并返回队列

virtual QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

virtual void print( ) const; //打印队列

virtual ~QUEUE( ); //销毁队列

};

**1.2需求分析**

编程要求实现面向对象的使用了运算符重载技术的整型队列，故本项目即实现面向对象的运用了运算符重载技术的整型队列的后台开发，并设计配套UI以方便使用者对已经实现的面向对象的运用了运算符重载技术的整型队列进行使用及测试。

故本项目应包括如下部分：

1. 实现面向对象的运用了运算符重载技术的整型队列的接口
2. 编写配套UI以对上述程序进行测试

**2.系统设计**

**2.1概要设计**

系统的总体结构图如图4.1所示。本项目共由初始化模块，功能选择模块，打印模块，入队模块，出队模块及队指派模块组成。

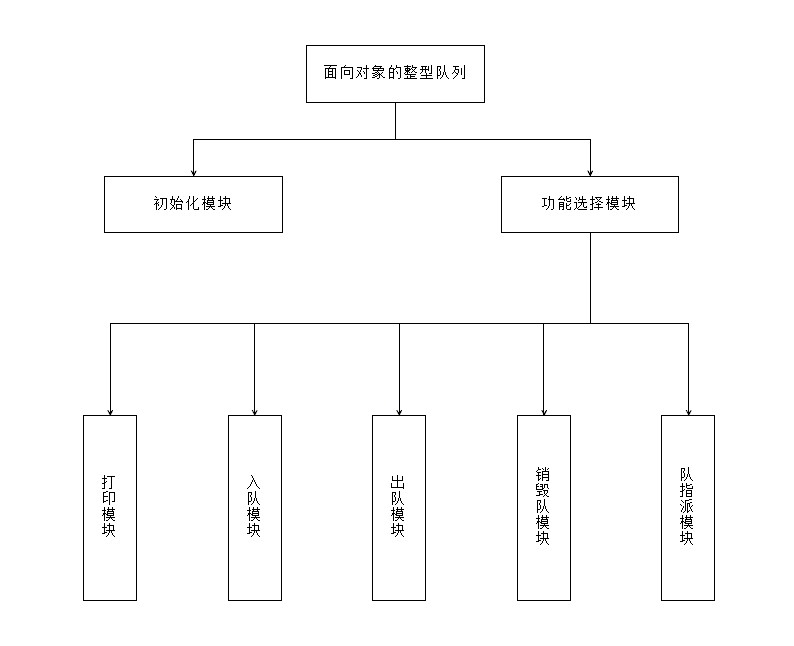


图 5.1 总体结构图

系统的总体流程图如图5.2所示。用户进入系统后应首先输入桟的初始化大小以决定以何种方式初始化队列，即以整数n还是以系统预设队列q初始化队。

在初始化队后用户可以通关功能选择模块选择将要执行的功能，特别地，用户在选择退出系统后将清空系统资源并退出本系统。

用户在选择打印队列后系统将展示队列已容纳元素个数，队列可容纳元素个数，队列内所有元素。

用户在选择入队后系统将引导用户输入想要入队的元素（整型数），再将用户输入的元素入队。

用户在选择出队后系统出队，并打印出队元素的值。

用户在选择销毁队列后系统将销毁队。

用户在选择指派队列后系统会把系统预设队指派给当前使用的队。

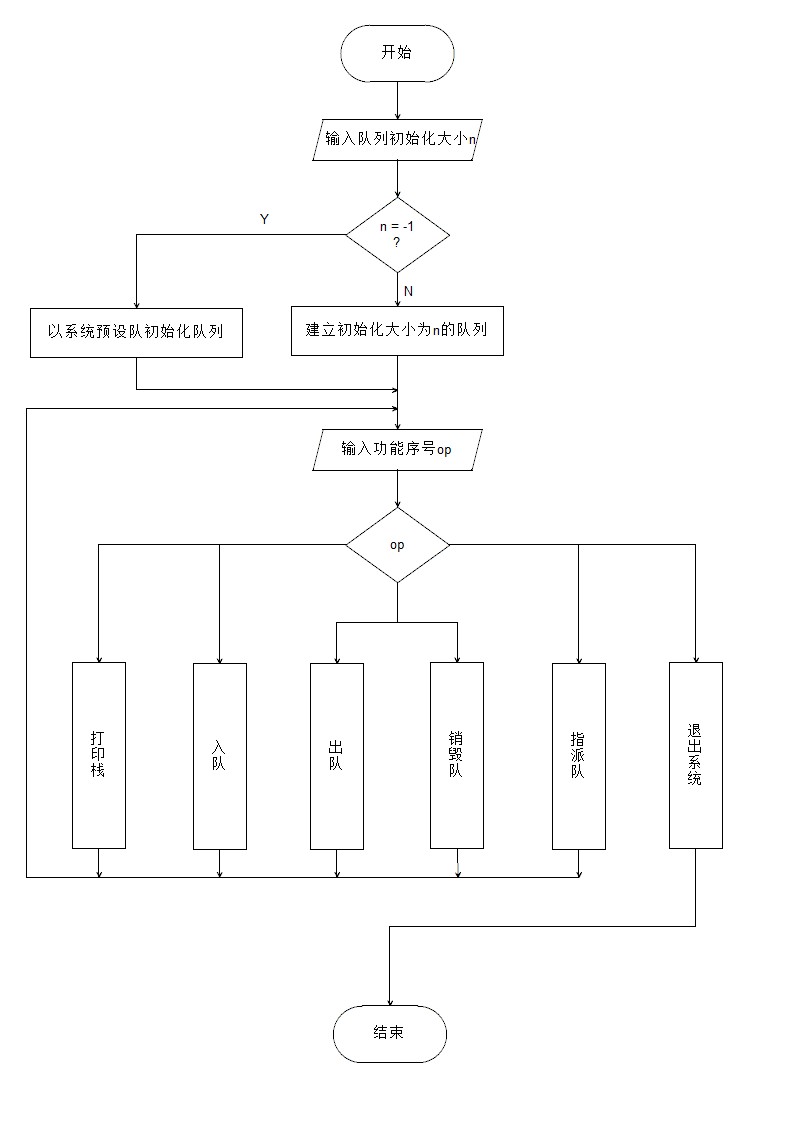


图 5.2 总体流程图

本次实验要求整合之前实现的STACK实现QUEUE。分析接口可知本次实验意图期望我们使用两个整型栈s1和s2实现队列。因此使用s1作为存放队列的容器，s2作为临时容器。

在入队时直接将元素e入栈至栈s1中。

在出队时将s1中所有元素出栈再入栈进s2中，之后s2出栈即可实现队列的出队，此时再将s2中所有元素依次入栈进s1中即可。

特别地，在s1入栈导致s1自动扩容时，需要为s2扩容。

**2.2详细设计**

1. 初始化模块：void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys);

模块功能：引导用户对所使用的队列进行初始化，包括使用整数m和系统预设队q对用户所使用的队列进行初始化。

入口参数：QUEUE\* & QUEUEUser 用户使用队

QUEUE\* & QUEUESys 系统预设队

出口参数：无

业务流程：参考图5.2

对应方法：

* 1. QUEUE(int m)
  2. QUEUE(const QUEUE&s);

1. 功能选择模块：int ChooseFunc()

模块功能：引导用户选择系统功能，包括打印队、入队、出队、销毁队、指派队、退出系统。

入口参数：无

出口参数：用户选择的系统功能

业务流程：参考图5.2

1. 打印队模块：void Func1(QUEUE \* const QUEUEUser)

模块功能：打印显示队的信息。包括队的容量，已容纳元素个数，已容纳元素具体信息。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual void print( ) const
  2. virtual int size ( ) const
  3. virtual operator int ( ) const

1. 入队模块：void Func2(QUEUE \* const QUEUEUser)

模块功能：引导用户输入入队元素，并将之入队

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual QUEUE & operator<<(int e);

1. 出队模块：void Func3(QUEUE \* const QUEUEUser)

模块功能：出队并显示出队元素。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual QUEUE & operator>>(int &e);

1. 销毁队模块：void Func4(QUEUE \* const QUEUEUser)

模块功能：销毁队。

入口参数：QUEUE \* const QUEUEUser

出口参数：无

对应方法：

* 1. virtual ~ QUEUE ( );

1. 队指派模块：void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys)

模块功能：讲系统预设队指派给用户当前所使用的队。

入口参数：QUEUE \* const QUEUEUser

出口参数：QUEUE \* const QUEUESys

对应方法：

* 1. virtual QUEUE & operator=(const QUEUE &s)

**3.软件开发**

本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 建立初始化大小为10的整型队列

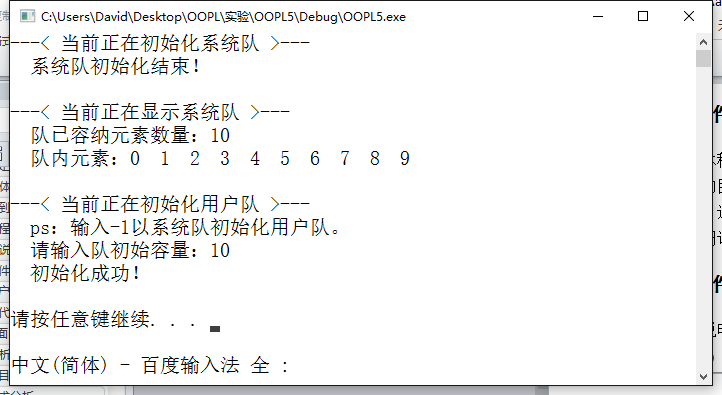


图5.3 初始化

1. 打印队列

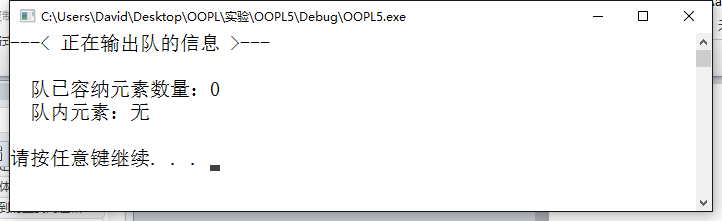


图5.4 打印队列

1. 将系统队赋值给当前使用的队列。

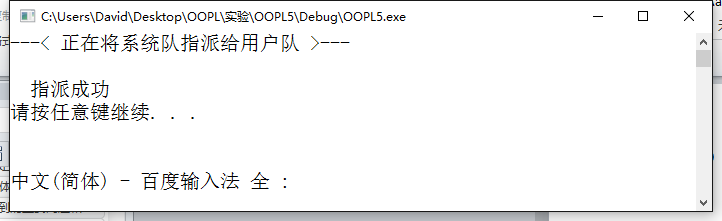


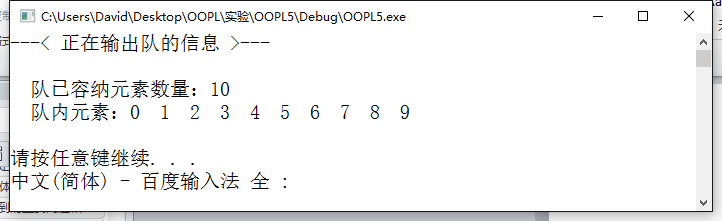
图5.5 指派队列

图5.6 打印队列

1. 出队

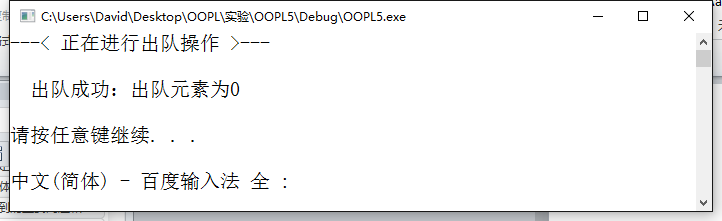


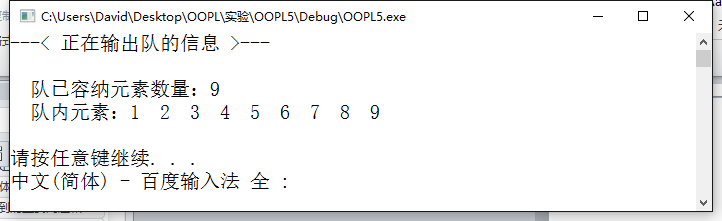
图5.7 出队

图5.8 打印队列

1. 入队（未满）

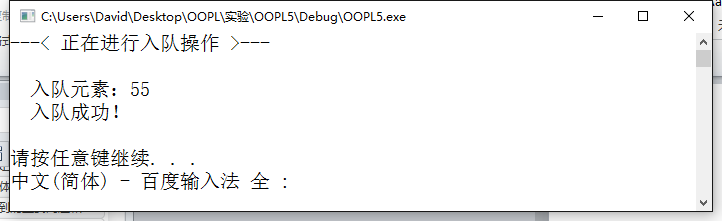


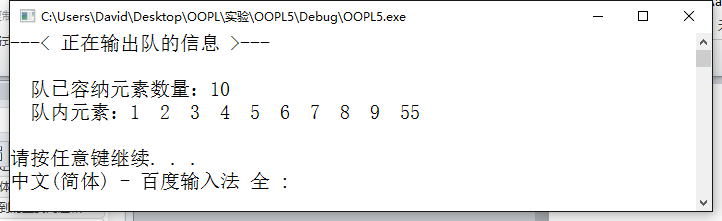
图5.9 入队

图5.10 打印队列

1. 入队（满）

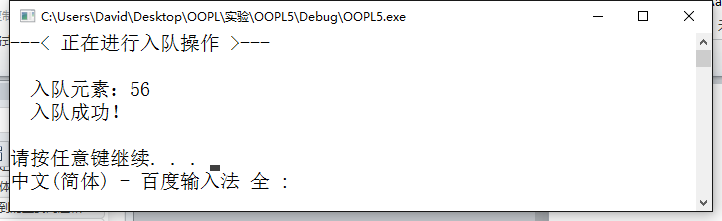


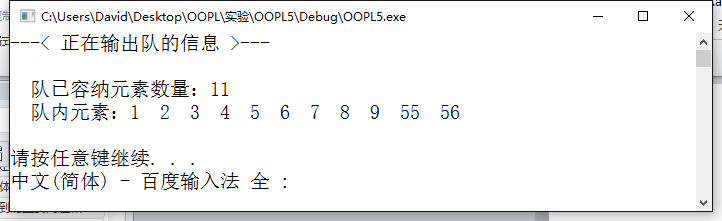
图5.11入队

图5.12打印队列

1. 销毁队

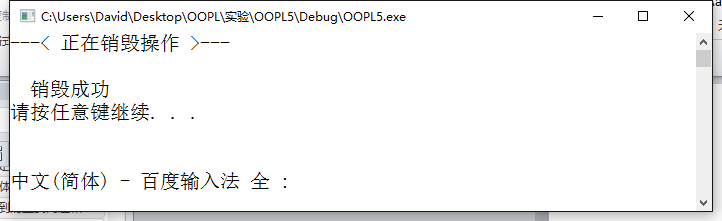


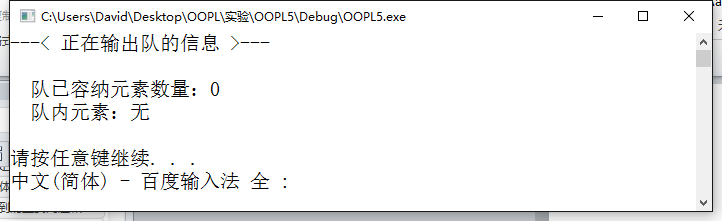
图5.13 销毁队

图5.14 打印队列

1. 使用系统预设队初始化队列

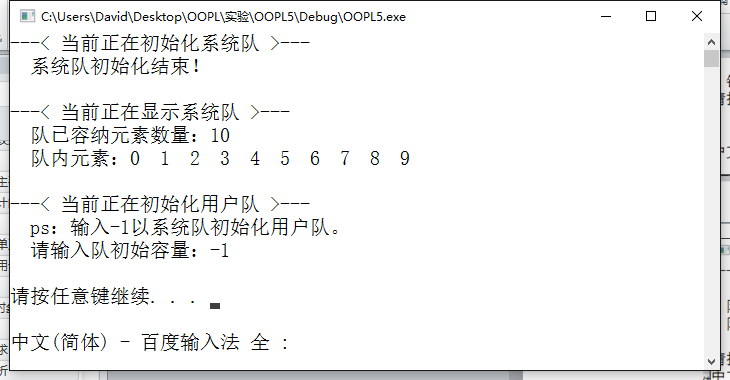


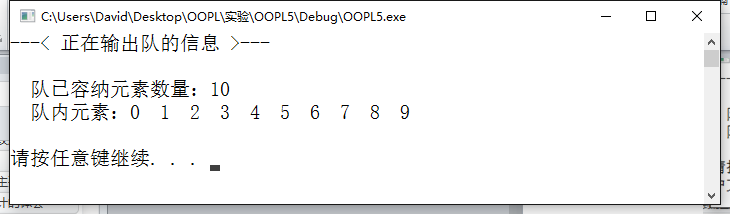
图5.15 销毁队

图5.16 打印队列

**5.特点与不足**

**5.1技术特点**

1. 能够做到自动为队列扩容，每次扩容的梯度是10
2. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
3. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. UI设计中有时为了节约篇幅，会采用一些自认为简洁明了的词汇，比如系统栈。实际上系统栈就是系统预设的一个栈，其内已经包含若干元素，方便用户使用它对栈进行初始化，或者assign给用户正在使用的栈。然而UI中并未给出对这个栈的解释，只能由用户体验后自行揣测其含义。这也是设计的一个不合理的地方。
3. 未使用C++特性的流式IO函数，而使用了兼容C的格式化IO函数。
4. 由于未提供显示队列容量的接口，因此无法显示队列的容量，使得测试结果缺乏说服力。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

无

**6.2课程设计的体会**

本实验旨于整合前几次实验的成果，以面向对象的整型栈的对象作为类的成员变量，通过以临时栈为倒腾缓冲区，实现面向对象的整型队列。在了解这点之后整个程序的实现就十分简单了，只用了不到一百行的代码（包括注释）就实现了指定的全部接口。

对于实验要求中所提出的问题，我的解答是：

本程序中并无父子继承关系，所以本程序中加上virtue关键字的类成员函数完全可以视作普通的类成员函数，因此类的成员函数的virtue关键字添加与否对程序结果并无影响。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL5\debug下是vs生成的可执行调试文件。

OOPL5\OOPL5下是组成本项目的C++源代码文件。主要包括main.cpp（主文件）、QUEUE.cpp（队列的成员函数的外部定义）、QUEUE.h（队列的成员变量、成员函数的原型）、STACK.cpp（栈的成员函数的外部定义）、STACK.h（栈的成员变量、成员函数的原型）、MyHeader.h（主文件中的部分函数原型）。

**7.2用户使用说明书**

运行~\OOPL5\Debug下的OOPL5.exe即可使用本程序。

**7.3源代码**

1. Stack.h

#pragma once

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

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

STACK& push(int e); //将e入栈,并返回栈

STACK& pop(int &e); //出栈到e,并返回栈

STACK& assign(const STACK&s); //赋s给栈,并返回被赋值的栈

void print() const; //打印栈

~STACK(); //销毁栈

};

1. STACK.cpp

#include "STACK.h"

#include <stdio.h>

#define STACK\_INCRE\_STEP 10

STACK::STACK(int m) :max(m), elems(new int[m]), pos(0)

{

if (!this->elems)

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

STACK::STACK(const STACK&s) : max(s.max), elems(new int[s.max]), pos(0)

{

if (this->elems)

{

// 分配成功，拷贝

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

{

push(s.getelem(i));

}

}

else

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

int STACK::size() const

{

return this->max;

}

int STACK::howMany() const

{

return this->pos;

}

int STACK::getelem(int x) const

{

return this->elems[x];

}

STACK& STACK::push(int e)

{

if (this->pos < this->max)

{

this->elems[this->pos++] = e;

}

else

{

int \* EleBase = new int[this->max + STACK\_INCRE\_STEP];

if (EleBase != 0)

{

/\* 拷贝原栈元素并入栈 \*/

int i;

for (i = 0; i < this->pos; i++)

{

EleBase[i] = this->getelem(i);

}

EleBase[i] = e;

/\* 修改栈信息 \*/

delete this->elems;

\*(int \*\*)(&this->elems) = EleBase;

\*(int \*)(&this->max) += STACK\_INCRE\_STEP;

this->pos++;

}

}

return \*(STACK\*)this;

}

STACK& STACK::pop(int &e)

{

e = this->elems[--this->pos];

return \*(STACK\*)this;

}

STACK& STACK::assign(const STACK&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (this->max != s.max)

{

delete this->elems;

\*(int \*\*)(&this->elems) = new int[s.max];

}

if (this->elems == 0)

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

}

else

{

\*(int \*)(&this->max) = s.max;

/\* 拷贝 \*/

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

{

this->elems[i] = s.elems[i];

}

this->pos = s.pos;

}

return \*(STACK\*)this;

}

void STACK::print() const

{

for (int i = this->pos; i > 0; i--)

{

printf("%d ", this->getelem(i - 1));

}

}

STACK::~STACK()

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

if (this->elems) //防止反复析构

{

delete this->elems;

\*(int \*\*)(&this->elems) = 0;

}

}

1. QUEUE.h

#include "STACK.h"

#pragma once

#include "STACK.h"

//s1作存储之用

//s2作出队倒腾之用

class QUEUE {

STACK s1, s2;

public:

QUEUE(int m); //初始化队列：每个栈最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

virtual operator int() const; //返回队列的实际元素个数

virtual QUEUE& operator<<(int e); //将e入队列,并返回队列

virtual QUEUE& operator>>(int &e); //出队列到e,并返回队列

virtual QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

virtual void print() const; //打印队列

virtual ~QUEUE(); //销毁队列

};

1. QUEUE.cpp

#include "QUEUE.h"

#include <math.h>

#include <stdio.h>

#define QUEUE\_INCRE\_STEP 10

//最多元素m

//数组长度m+1

//tail处无元素

QUEUE::QUEUE(int m) :s1(m), s2(m)

{

}

QUEUE::QUEUE(const QUEUE&s) : s1(s.s1), s2(s.s2)

{

}

QUEUE::operator int() const

{

return (int)s1;

}

QUEUE& QUEUE::operator<<(int e)

{

s1 << e;

//确保s1,s2大小一致

if (s1.size() != s2.size())

{

s2.~STACK();

s2 = STACK(s1.size());

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator>>(int &e)

{

int currentNum;

int temp;

currentNum = (int)s1;

if (currentNum)

{

//将s1中元素全部倒入s2中

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

{

s1 >> temp;

s2 << temp;

}

//将s2中栈顶元素出栈

s2 >> e;

//将s2中元素全部倒回s1中

currentNum = (int)s2;

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

{

s2 >> temp;

s1 << temp;

}

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator=(const QUEUE&s)

{

s1 = s.s1;

s2 = s.s2;

return \*(QUEUE\*)this;

}

void QUEUE::print() const

{

//打印排在队前的

for (int i = 0; i < \*this; i++)

{

printf("%d ", s1[i]);

}

}

QUEUE::~QUEUE(void)

{

s1.~STACK();

s2.~STACK();

}

1. MyHeader.h

#pragma once

#include "QUEUE.h"

void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys);

int ChooseFunc();

void ScreenClr();

void ScreenPause();

void Func1(QUEUE \* const QUEUEUser); //功能1

void Func2(QUEUE \* const QUEUEUser); //功能2

void Func3(QUEUE \* const QUEUEUser); //功能3

void Func4(QUEUE \* const QUEUEUser); //功能4

void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys); //功能5

1. main.cpp

#include "QUEUE.h"

#include <stdio.h>

#include <stdlib.h>

#include "MyHeader.h"

int main()

{

// variables declaration

QUEUE \* QUEUEUser; //操作用队

QUEUE \* QUEUESys; //赋值用队

int op;

//initial

SysInitial(QUEUEUser, QUEUESys);

//loop

while ((op = ChooseFunc()) != 0)

{

switch (op)

{

case 0:

op = -1;

break;

case 1:

Func1(QUEUEUser);

break;

case 2:

Func2(QUEUEUser);

break;

case 3:

Func3(QUEUEUser);

break;

case 4:

Func4(QUEUEUser);

break;

case 5:

Func5(QUEUEUser, QUEUESys);

break;

}

}

//quit

ScreenClr();

printf("\n---< 正在退出系统 >---\n");

delete QUEUEUser;

delete QUEUESys;

ScreenPause();

}

void SysInitial(QUEUE\* & QUEUEUser, QUEUE\* & QUEUESys)

{

//初始化默认队

int i, n, num;

printf("---< 当前正在初始化系统队 >---\n");

QUEUESys = new QUEUE(10);

for (i = 0; i < 10; i++)

{

(\*QUEUESys) << i;

}

printf(" 系统队初始化结束！\n");

//显示默认队

printf("\n---< 当前正在显示系统队 >---\n");

num = \*QUEUESys;

printf(" 队已容纳元素数量：%d\n", num);

printf(" 队内元素：");

if (num == 0)

{

printf("无");

}

else

{

QUEUESys->print();

}

printf("\n");

//引导初始用户队

printf("\n---< 当前正在初始化用户队 >---\n");

printf(" ps：输入-1以系统队初始化用户队。\n");

do

{

printf(" 请输入队初始容量：");

scanf\_s("%d", &n);

getchar();

if (n == -1)

{

QUEUEUser = new QUEUE(\*QUEUESys);

}

else if (n > -1)

{

QUEUEUser = new QUEUE(n);

printf(" 初始化成功！\n");

}

else

{

printf(" 请输入有效大小！\n");

}

} while (n < -1);

ScreenPause();

}

int ChooseFunc()

{

int n = -1; //用户选择的功能序号

/\* 输出功能选择提示语 \*/

ScreenClr();

printf("---< 选择程序功能 >---\n\n");

printf(" [1].查看队信息\n");

printf(" [2].入队操作\n");

printf(" [3].出队操作\n");

printf(" [4].销毁队\n");

printf(" [5].将系统队赋给用户队\n");

printf(" [0].退出系统\n");

printf("\n请输入功能代码：");

/\* 录入提示语 \*/

scanf\_s("%d", &n);

getchar();

/\* 输入验证 \*/

if (n < 0 || n > 5)

{

n = -1;

printf(" 错误的代码，请重新输入。\n");

ScreenPause();

}

return n;

}

void Func1(QUEUE \* const QUEUEUser)

{

//initial

int max, pos, i;

//output

ScreenClr();

printf("---< 正在输出队的信息 >---\n\n");

pos = \*QUEUEUser;

printf(" 队已容纳元素数量：%d\n", pos);

printf(" 队内元素：");

if (pos == 0)

{

printf("无");

}

else

{

QUEUEUser->print();

}

printf("\n");

//wait

ScreenPause();

}

void Func2(QUEUE \* const QUEUEUser)

{

//initial

int var;

//录入

ScreenClr();

printf("---< 正在进行入队操作 >---\n\n");

printf(" 入队元素：");

scanf\_s("%d", &var);

getchar();

(\*QUEUEUser) << var;

printf(" 入队成功！\n");

//wait

ScreenPause();

}

void Func3(QUEUE \* const QUEUEUser)

{

//initial

int var;

//出队

ScreenClr();

printf("---< 正在进行出队操作 >---\n\n");

if (\*QUEUEUser < 1)

{

printf(" 出队失败：队为空\n");

}

else

{

\*QUEUEUser >> var;

printf(" 出队成功：出队元素为%d\n", var);

}

//wait

ScreenPause();

}

void Func4(QUEUE \* const QUEUEUser)

{

//initial

int var;

//销毁

ScreenClr();

printf("---< 正在销毁操作 >---\n\n");

QUEUEUser->~QUEUE();

printf(" 销毁成功");

//wait

ScreenPause();

}

void Func5(QUEUE \* const QUEUEUser, QUEUE \* const QUEUESys)

{

//initial

int var;

//指派

ScreenClr();

printf("---< 正在将系统队指派给用户队 >---\n\n");

\*QUEUEUser = \*QUEUESys;

printf(" 指派成功");

//wait

ScreenPause();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}

实验六.基于继承的整型队列编程

**1.需求分析**

**1.1题目要求**

整型队列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队列是否为空、向队列顶添加一个整型元素、出队列等。整型队列类型及其操作函数采用面向对象的C++语言定义，请将完成上述操作的所有函数采用C++编程。注意，请用实验三的SATCK继承形成新的类QUEQUE。分析说明除构造函数以外的函数，加virtual说明与不加virtual说明有无区别。并说明为什么不将s2也作为基类。

class QUEUE：public STACK{

STACK s2;

public:

QUEUE(int m); //初始化队列：每个栈最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

operator int ( ) const; //返回队列的实际元素个数

QUEUE& operator<<(int e); //将e入队列,并返回队列

QUEUE& operator>>(int &e); //出队列到e,并返回队列

QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

void print( ) const; //打印队列

~QUEUE( ); //销毁队列

};

在完成上述程序及测试无误后，请使用队列解决如下舞伴问题。假定在一次舞会上，男士排成一队，女士排成另一队。每次舞曲响起时，从男队和女队的队头各出一人，配成舞伴跳完此曲，跳完后各自进入自己队列的尾部。若男女两队的初始人数分别为M和F（M和F均为素数，且M!=F），男队中排在位置m（m<=M）的男士，非常想和女队位置f（f<=F）的女士跳舞，问他在第几支曲舞曲才能和该女士跳舞?请编程解决上述问题。

**2.系统设计**

**2.1概要设计**

本系统共由排队模块，主角选择模块，跳舞模块这三个模块组成。三个模块在时间轴上的执行是连续的· 。

**2.2详细设计**

1. 排队模块：void SysInitial(QUEUE\* & QueueMale, QUEUE\* & QueueFemale);

模块功能：引导用户输入男士队伍和女士队伍的人数，并完成对男士和女士队伍中所有人员的编号。编号的结果会在用户输入男士/女士队伍的人数的时候输出。

入口参数：QUEUE\* & QueueMale 男士队列

QUEUE\* & QueueFemale 女士队列

出口参数：无

处理流程：输入男士数量→为男士编号并输出编号结果→输入女士数量→为女士编号并输出编号结果

1. 主角选择模块：void ChooseTarget(int &no\_male, int &no\_female)

模块功能：选择期望相遇的男士和女士的编号。

入口参数：no\_male 男士编号

no\_female 女士编号

出口参数：无

处理流程：输入男士编号→输入女士编号

1. 跳舞模块：void PrintEachTurn(int no\_male, int no\_female, QUEUE\* & QueueMale, QUEUE\* & QueueFemale)

模块功能：男士女士逐一出列跳舞，跳完后再次入列。如此循环直至指定男士和女士共跳一支舞或总舞蹈轮数大于99999。

入口参数：no\_male 男士编号

no\_female 女士编号

QUEUE\* & QueueMale 男士队列

QUEUE\* & QueueFemale 女士队列

出口参数：无

处理流程：如图6.1所示

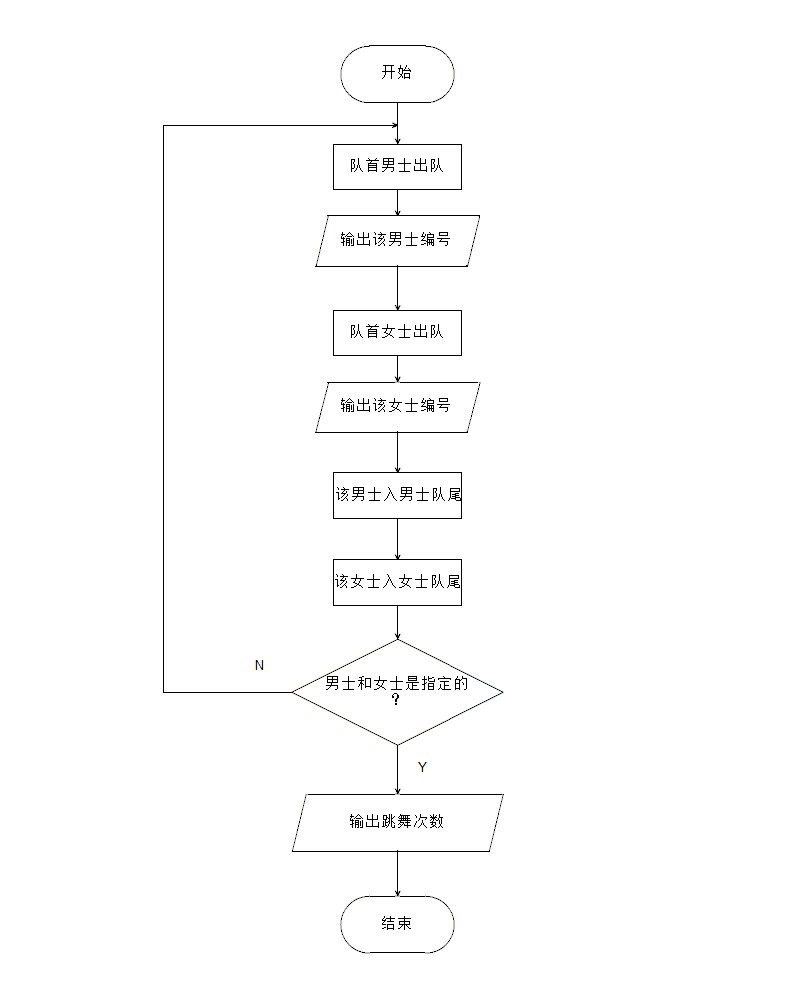


图6.1 跳舞模块流程图

**3.软件开发**

本程序采用visual studio 2010开发。在开发过程中只需建立项目，然后在项目的目录中逐一添加程序所用文件，再点击界面中的运行按钮即可自动完成编译、连接的流程。

调试采用vs自带的断点调试。

**4.软件测试**

说明：以下测试在时间轴上是连续的。

1. 排队模块测试：输入男士队列人数13，女士队列人数17。

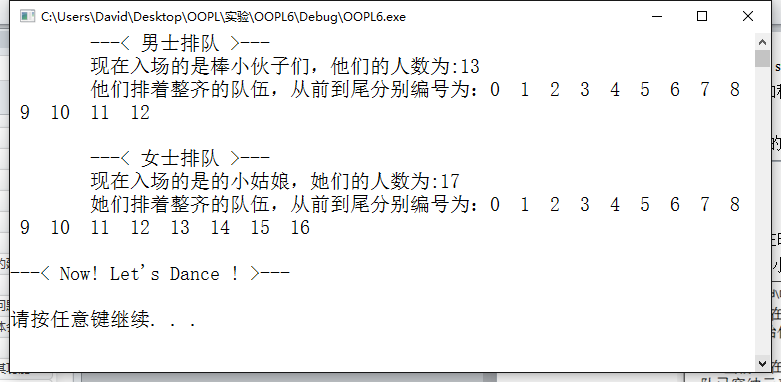


图6.2 排队模块测试

1. 主角选择模块：指定男士队列中编号为7的和女士队列中编号为11的为主角。

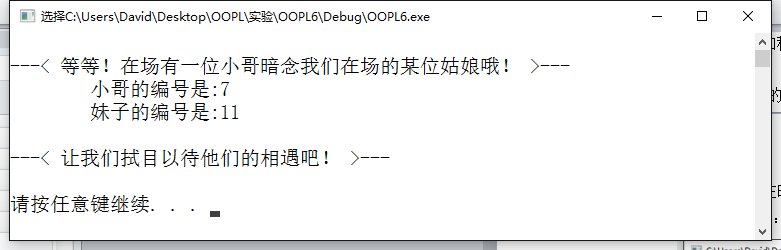


图6.3主角选择模块测试

1. 跳舞模块：

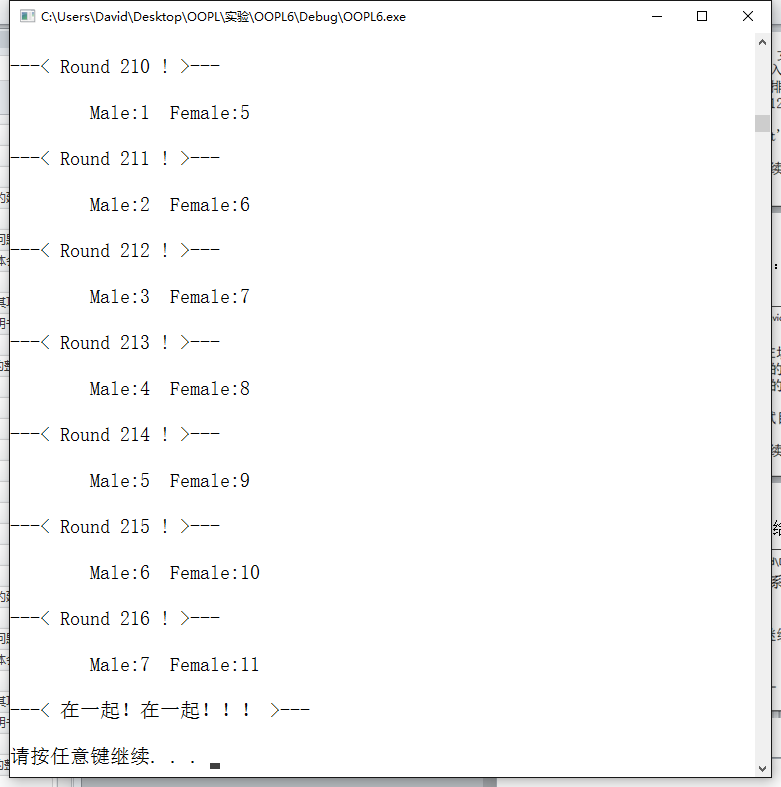


图6.4 跳舞模块测试

根据实验结果可知，该男士和该女士在第216轮舞蹈开始后同跳一只舞，与预期一致。

**5.特点与不足**

**5.1技术特点**

1. 极大地将数据处理部分与数据呈现部分隔离，使代码容易复用
2. 代码模块化程度高，使代码容易阅读

**5.2不足和改进的建议**

1. 部分情况下默认用户的输入是正确地（如需要输入数字的地方未输入除数字外的字符），并未完全地考虑部分极端用户可能的行为，因此在这些情况下，系统可能无法很好地响应用户。
2. 未使用C++特性的流式IO函数，而使用了兼容C的格式化IO函数。

**6.过程和体会**

**6.1遇到的主要问题和解决方法**

1. 输入字符流后scanf函数会将字符部分截取，而留下’\n’字符。解决方法：在每个输入函数后加上getchar()函数。
2. 如何在子类中调用父类中的经运算符重载的方法。解决方案：以作用域加完整成员函数名称加括号调用，即作为普通成员函数调用，形如 : this->STACK::operator>>(temp);

**6.2课程设计的体会**

本实验的数据结构部分可以看作因继承栈而使自身成为s1的实验五。在本点的基础上出发，基于继承的整数队列就很容易实现了。

对于实验要求中所提出的问题，我的解答是：

由于本程序中不存在父类指针指向子类对象，因此虚函数的多态特性并不能得以体现，因此此处virtue添加与否对程序功能并没有影响。

**7.源码和说明**

**7.1文件清单及其功能说明**

OOPL6\Debug下是vs生成的可执行调试文件。

OOPL6\OOPL6下是组成本项目的C++源代码文件。主要包括main.cpp（主文件）、QUEUE.cpp（队列的成员函数的外部定义）、QUEUE.h（队列的成员变量、成员函数的原型）、STACK.cpp（栈的成员函数的外部定义）、STACK.h（栈的成员变量、成员函数的原型）、MyHeader.h（主文件中的部分函数原型）。

**7.2用户使用说明书**

运行~\OOPL6\Debug下的OOPL6.exe即可使用本程序。

**7.3源代码**

1. Stack.h

#pragma once

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

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

STACK& push(int e); //将e入栈,并返回栈

STACK& pop(int &e); //出栈到e,并返回栈

STACK& assign(const STACK&s); //赋s给栈,并返回被赋值的栈

void print() const; //打印栈

~STACK(); //销毁栈

};

1. STACK.cpp

#include "STACK.h"

#include <stdio.h>

#define STACK\_INCRE\_STEP 10

STACK::STACK(int m) :max(m), elems(new int[m]), pos(0)

{

if (!this->elems)

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

STACK::STACK(const STACK&s) : max(s.max), elems(new int[s.max]), pos(0)

{

if (this->elems)

{

// 分配成功，拷贝

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

{

push(s.getelem(i));

}

}

else

{

//分配失败，置零

\*(int \*)(&this->max) = 0;

}

}

int STACK::size() const

{

return this->max;

}

int STACK::howMany() const

{

return this->pos;

}

int STACK::getelem(int x) const

{

return this->elems[x];

}

STACK& STACK::push(int e)

{

if (this->pos < this->max)

{

this->elems[this->pos++] = e;

}

else

{

int \* EleBase = new int[this->max + STACK\_INCRE\_STEP];

if (EleBase != 0)

{

/\* 拷贝原栈元素并入栈 \*/

int i;

for (i = 0; i < this->pos; i++)

{

EleBase[i] = this->getelem(i);

}

EleBase[i] = e;

/\* 修改栈信息 \*/

delete this->elems;

\*(int \*\*)(&this->elems) = EleBase;

\*(int \*)(&this->max) += STACK\_INCRE\_STEP;

this->pos++;

}

}

return \*(STACK\*)this;

}

STACK& STACK::pop(int &e)

{

e = this->elems[--this->pos];

return \*(STACK\*)this;

}

STACK& STACK::assign(const STACK&s)

{

/\* 当Smax不等于Pmax，则修改Pelem \*/

if (this->max != s.max)

{

delete this->elems;

\*(int \*\*)(&this->elems) = new int[s.max];

}

if (this->elems == 0)

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

}

else

{

\*(int \*)(&this->max) = s.max;

/\* 拷贝 \*/

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

{

this->elems[i] = s.elems[i];

}

this->pos = s.pos;

}

return \*(STACK\*)this;

}

void STACK::print() const

{

for (int i = this->pos; i > 0; i--)

{

printf("%d ", this->getelem(i - 1));

}

}

STACK::~STACK()

{

\*(int \*)(&this->max) = 0;

this->pos = 0;

if (this->elems) //防止反复析构

{

delete this->elems;

\*(int \*\*)(&this->elems) = 0;

}

}

1. QUEUE.h

#pragma once

#include "STACK.h"

class QUEUE : public STACK {

STACK s2;

public:

QUEUE(int m); //初始化队列：每个栈最多m个元素

QUEUE(const QUEUE&s); //用队列s拷贝初始化队列

operator int() const; //返回队列的实际元素个数

QUEUE& operator<<(int e); //将e入队列,并返回队列

QUEUE& operator>>(int &e); //出队列到e,并返回队列

QUEUE& operator=(const QUEUE&s); //赋s给队列,并返回被赋值的队列

void print() const; //打印队列

~QUEUE(); //销毁队列

};

1. QUEUE.cpp

#include "QUEUE.h"

#include <math.h>

#include <stdio.h>

#define QUEUE\_INCRE\_STEP 10

//s2作为临时容器

//继承自STACK的内容为主容器

QUEUE::QUEUE(int m) :STACK(m), s2(m)

{

}

QUEUE::QUEUE(const QUEUE&s) : STACK(s), s2(s.s2)

{

}

QUEUE::operator int() const

{

return this->STACK::operator int();

}

QUEUE& QUEUE::operator<<(int e)

{

this->STACK::operator<<(e);

//确保s,s2大小一致

if (this->size() != s2.size())

{

s2.~STACK();

s2 = STACK(this->size());

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator>>(int &e)

{

int currentNum;

int temp;

currentNum = \*this;

if (currentNum)

{

//将this中元素全部倒入s2中

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

{

this->STACK::operator>>(temp);

s2 << temp;

}

//将s2中栈顶元素出栈

s2 >> e;

//将s2中元素全部倒回this中

currentNum = (int)s2;

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

{

s2 >> temp;

this->STACK::operator<<(temp);

}

}

return \*(QUEUE\*)this;

}

QUEUE& QUEUE::operator=(const QUEUE&s)

{

this->STACK::operator=(s);

s2 = s.s2;

return \*(QUEUE\*)this;

}

void QUEUE::print() const

{

//打印排在队前的

for (int i = 0; i < \*this; i++)

{

printf("%d ", (\*this)[i]);

}

}

QUEUE::~QUEUE(void)

{

this->STACK::~STACK();

s2.~STACK();

}

1. MyHeader.h

#pragma once

#include "QUEUE.h"

void SysInitial(QUEUE\* & QueueMale, QUEUE\* & QueueFemale);

void ChooseTarget(int &no\_male, int &no\_female);

void ScreenClr();

void ScreenPause();

void PrintEachTurn(int no\_male, int no\_female, QUEUE\* & QueueMale, QUEUE\* & QueueFemale);

1. main.cpp

#include <stdio.h>

#include <stdlib.h>

#include "MyHeader.h"

int main()

{

// variables declaration

QUEUE \* QueueMale; //男士舞队

QUEUE \* QueueFemale; //女士舞队

int no\_male, no\_female;

//initial

SysInitial(QueueMale, QueueFemale);

ChooseTarget(no\_male, no\_female);

//loop

PrintEachTurn(no\_male, no\_female, QueueMale, QueueFemale);

//quit

ScreenClr();

printf("\n---< 男士女士们退场了 >---\n");

delete QueueMale;

delete QueueFemale;

ScreenPause();

ScreenClr();

}

void SysInitial(QUEUE\* & QueueMale, QUEUE\* & QueueFemale)

{

//初始化

int i, num\_male, num\_female;

printf("---< 男士女士们正在排队 >---\n");

/\* 男士排队 \*/

printf("\n\t---< 男士排队 >---\n");

printf("\t现在入场的是棒小伙子们，他们的人数为:");

scanf\_s("%d", &num\_male);

getchar();

QueueMale = new QUEUE(num\_male);

for (i = 0; i < num\_male; i++)

{

\*QueueMale << i;

}

printf("\t他们排着整齐的队伍，从前到尾分别编号为：");

QueueMale->print();

printf("\n");

/\* 女士排队 \*/

printf("\n\t---< 女士排队 >---\n");

printf("\t现在入场的是的小姑娘，她们的人数为:");

scanf\_s("%d", &num\_female);

getchar();

QueueFemale = new QUEUE(num\_female);

for (i = 0; i < num\_female; i++)

{

\*QueueFemale << i;

}

printf("\t她们排着整齐的队伍，从前到尾分别编号为：");

QueueFemale->print();

printf("\n");

printf("\n---< Now! Let's Dance ! >---\n");

ScreenPause();

ScreenClr();

}

void ChooseTarget(int &no\_male, int &no\_female)

{

int i;

printf("\n---< 等等！在场有一位小哥暗念我们在场的某位姑娘哦！ >---\n");

printf("\t小哥的编号是:");

scanf\_s("%d", &no\_male);

printf("\t妹子的编号是:");

scanf\_s("%d", &no\_female);

printf("\n---< 让我们拭目以待他们的相遇吧！ >---\n");

ScreenPause();

ScreenClr();

}

void PrintEachTurn(int no\_male, int no\_female, QUEUE\* & QueueMale, QUEUE\* & QueueFemale)

{

int male\_now, female\_now, count = 0;

printf("\n---< Music On ! >---\n");

do

{

printf("\n---< Round %d ! >---\n\n", ++count);

\*QueueMale >> male\_now;

printf("\tMale:%d", male\_now);

\*QueueFemale >> female\_now;

printf("\tFemale:%d", female\_now);

\*QueueMale << male\_now;

\*QueueFemale << female\_now;

printf("\n");

} while ((male\_now != no\_male || female\_now != no\_female) && count <= 99999);

if (count >= 99999)

{

printf("\n---< 你们无缘呐！ >---\n");

}

else

{

printf("\n---< 在一起！在一起！！！ >---\n");

}

ScreenPause();

ScreenClr();

}

void ScreenClr()

{

system("Cls");

}

void ScreenPause()

{

printf("\n");

system("pause");

}