

华中科技大学

课程实验报告

课程名称：C++程序设计

实验名称：面向过程的整型队列编程

院 系：计算机科学与技术学院

专业班级：计算机科学与技术 2001 班

学 号：U202011641

姓 名：刘景宇

指导教师：马光志

2021 年 12 月 12 日

一、需求分析

1. 题目要求

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：向队列尾部添加一个整型元素、从队列首部移除一个整型元素等。整型循环队列类型 **Queue** 及其操作函数采用非面向对象的 **C** 语言定义，请将完成上述操作的所有如下函数采用 **C** 语言编程，然后写一个 **main** 函数对队列的所有操作函数进行测试，请不要自己添加定义任何新的函数成员和数据成员。

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
struct Queue{
    int* const  elems;      //elems 申请内存用于存放队列的元素
    const  int  max;        //elems 申请的最大元素个数 max
    int  head, tail;        //队列头 head 和尾 tail，队空 head=tail;初始 head=tail=0
};
void initQueue(Queue *const p, int m);    //初始化 p 指队列：最多申请 m 个元素
void initQueue(Queue *const p, const Queue&s); //用 s 深拷贝初始化 p 指队列
void initQueue(Queue *const p, Queue&&s); //用 s 移动初始化 p 指队列
int  number (const Queue *const p); //返回 p 指队列的实际元素个数
int  size(const Queue *const p);      //返回 p 指队列申请的最大元素个数 max
Queue*const enter(Queue*const p, int e); //将 e 入队列尾部，并返回 p
Queue*const leave(Queue*const p, int &e); //从队首出元素到 e，并返回 p
Queue*const assign(Queue*const p, const Queue&q); //深拷贝赋 s 给队列并返回 p
Queue*const assign(Queue*const p, Queue&&q); //移动赋 s 给队列并返回 p
char*print(const Queue *const p, char*s); //打印 p 指队列至 s 并返回 s
void destroyQueue (Queue *const p); //销毁 p 指向的队列
```

编程时应采用 **VS2019** 开发，并将其编译模式设置为 **X86** 模式，其他需要注意的事项说明如下：

(1) 用 **initQueue(Queue *const p, int m)**对 **p** 指向的队列初始化时， 为其 **elems** 分配 **m** 个整型元素内存，并初始化 **max** 为 **m**，以及初始化 **head=tail=0**。

(2) 对于 **initQueue(Queue *const p, const Queue& q)**初始化，用已经存在的对象 **q** 深拷贝构造新对象***p**时，新对象***p**不能和对象 **q**的 **elems** 共用同一块内存，新对象***p**的 **elems** 需要分配和 **q** 为 **elems** 分配的同样大小的内存，并且将已经存在 **q** 的 **elems** 的内容深拷贝至新分配的内存；新对象***p**的 **max**、**head**、**tail** 应设置成和已经存在的对象 **s** 相同。

(3) 对于 **initQueue(Queue *const p, Queue&& q)**初始化，用已经存在的对象 **q** 移动构造新对象，新对象使用对象 **q** 为 **elems** 分配的内存块，并将其 **max**、**head**、**tail** 设置成和 **s** 的对应值相同，然后将 **s** 的 **elems** 设置为空表示内存已经移动给新对象，将 **s** 的 **max**、**head**、**tail** 设置为 **0**。

(4) 对于 `Queue*const assign(Queue*const p, const Queue&q)` 深拷贝赋值, 用等号右边的对象 `q` 深拷贝赋值给等号左边的对象, 等号左边的对象如果已经有内存则应先释放以避免内存泄漏, 然后分配和对象 `q` 为 `elems` 分配的同样大小的内存, 并且设置其 `max`、`head`、`tail` 和 `q` 的对应值相同。

(5) 对于 `Queue*const assign(Queue*const p, Queue&&q)` 移动赋值, 若等号左边的对象为 `elems` 分配了内存, 则先释放改内存一面内存泄漏, 然后使用等号右边对象 `q` 为 `elems` 分配的内存, 并将其 `max`、`head`、`tail` 设置成和对象 `q` 的对应值相同; 对象 `q` 的 `elems` 设置为空指针以表示内存被移走给等号左边的对象, 同时其 `max`、`head`、`tail` 均应设置为 0。

(6) 队列应实现为循环队列, 当队尾指针 `tail` 快要追上队首指针 `head` 时, 即如果满足 $(tail+1)\%max=head$, 则表示表示队列已满, 故队列最多存放 `max-1` 个元素; 而当 `head=tail` 时则表示队列为空。队列空取出元素或队列满放入元素均应抛出异常, 并且保持其内部状态不变。

(7) 打印队列时从队首打印至队尾, 打印的元素之间以逗号分隔。

2. 需求分析

队列在日常生活中十分常见, 例如: 银行排队办理业务、食堂排队打饭等等, 这些都是队列的应用。排队的原则就是先来后到, 排在前面的人就可以优先办理业务, 那么队列也一样, 队列遵循先进先出的原则。

这里要求对整型队列进行 C 语言实现, 实现队列的先进先出, 并且将其功能进行模块化, 编写成函数, 为了避免用户异常调用函数, 函数参数多采用常引用。初始化队列时可以是生成空队列, 也可以是深拷贝一份, 或者浅拷贝一份, 其中对于浅拷贝得到的队列, 要在释放队列前判断空间是否已经释放。能够实现队列的复制, 并且可以完成从尾部入队、从首部出队等基本操作, 同时能够对于空队列时出队、满队列时入队等不合法操作抛出异常, 作为提示, 能够随时查看队列能存放的最大整数个数以及当前存放整数个数。

二、系统设计

1. 概要设计

对于面向过程的整型队列的实现, 为了能够实现模块化以及代码的复用性, 对于相应的数据结构进行设计以及功能实现进行模块化编写。

设计相应的数据结构, 包括存放队列元素的部分、最大元素, 以及队列的头和尾的序号。其中 `elems` 为指针, 指向 `Queue` 队列对应的首地址, `max` 为整型常量, 存储 `Queue` 中的最大存储元素的个数, 整型变量 `head` 和 `tail` 分别存放队列中首元素和尾元素在队列中的序号位置。(如图 1 所示)

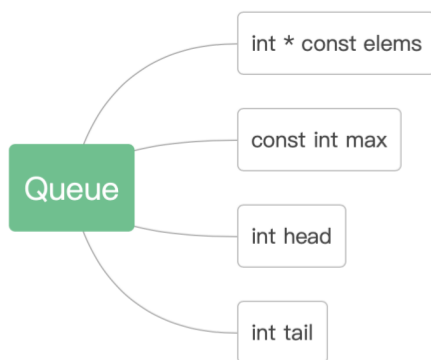


图 1 Queue 数据结构

将队列功能的实现进行函数化，包括初始化队列，其中、返回实际元素个数、最大元素、进队、出队、队列的复制、队列元素的输出显示等。（如图 2 所示）

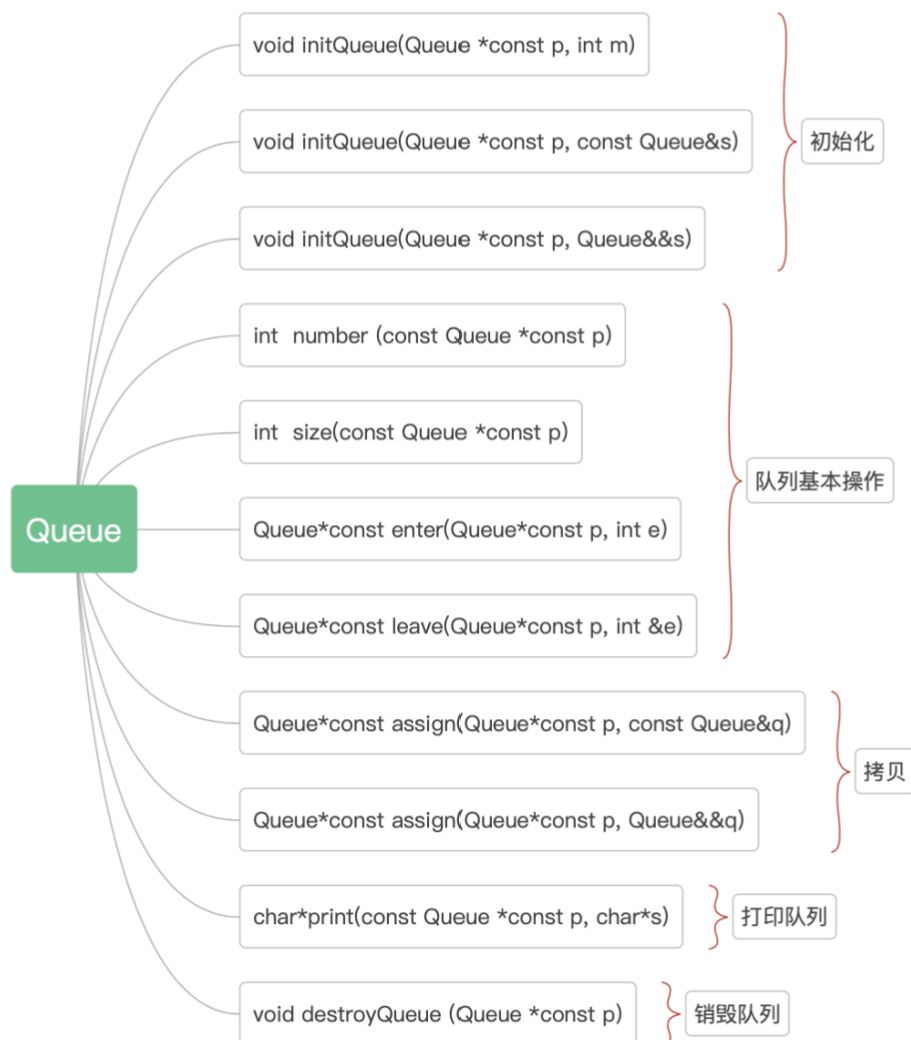


图 2Queue 基本功能

2. 详细设计

2.1 Queue 初始化

2.1.1 void initQueue(Queue *const p, int m)

功能：初始化 p 指向的队列，并且最多申请 m 个元素。

入口参数：队列指针 p，整数 m

出口参数：无

流程：对于传入参数 m 进行判断，如果 m 小于等于 0，不合法则抛出异常。否则分配 m 个整数大小的空间，并将首地址赋值给 p，max 设置为 m，队列头 head 和队列尾都初始化为 0。

2.1.2 void initQueue(Queue *const p, const Queue&s)

功能：用 s 深拷贝初始化 p 指向的队列

入口参数：队列指针 p，队列常引用 s

出口参数：无

流程：给 p 指向的队列的分配同样大小的空间，首地址赋值给 elems，将队列 s 中 elems 中的每个元素复制一份到 p 指向队列的 elems 中，并将队列 s 的其他参数对应赋值给 p 指向的队列。

2.1.3 void initQueue(Queue *const p, Queue&&s)

功能：用 s 移动初始化 p 指队列

入口参数：队列指针 p，队列常引用 s

出口参数：无

流程：将队列 s 中的所有元素赋值给 p 指向的队列中的对应参数，并且将 s 队列中的每个元素进行初始化，将 elems 赋值为空，将 max、head、tail 都赋值为 0。

2.2 Queue 基本操作

2.2.1 int number (const Queue *const p)

功能：返回 p 指队列的实际元素个数

入口参数：队列指针 p

出口参数：队列中实际元素个数

流程：利用队列的队首和队尾的编号，以及队列能存放的最大元素个数。利用以下计算元素个数的公式 $(\text{tail} - \text{head} + \text{max}) \% \text{max}$ 可以得到队列存放元素的个数。

2.2.2 int size(const Queue *const p)

功能：返回 p 指队列申请的最大元素个数 max

入口参数：队列指针 p

出口参数：队列中能存放的最大元素个数

流程：返回 p 指向的队列中的 max。

2.2.3 Queue*const enter(Queue*const p, int e)

功能：将 e 入队列尾部，并返回 p

入口参数：队列指针 p，入队整数 e

出口参数：入队后队列的常引用

流程：首先判断队列是否满，如果满，则抛出异常，否则将 e 存放到队尾在队列中的下一个元素，并更新队列队尾的编号，返回更新后的队列的引用。

2.2.4 Queue*const leave(Queue*const p, int &e)

功能：从队首出元素到 e，并返回 p

入口参数：队列指针 p，用来接收队首元素的整数变量 e

出口参数：入队后队列的常引用

流程：首先判断队列是否空，如果空，则抛出异常，否则取出队列中的第一个元素赋值给 e，并更新队首在队列中的编号，返回更新后的队列的引用。

2.3 拷贝 Queue

2.3.1 Queue*const assign(Queue*const p, const Queue&q)

功能：深拷贝赋 s 给队列并返回 p

入口参数：队列指针 p，队列常引用 q

出口参数：深拷贝后队列的引用

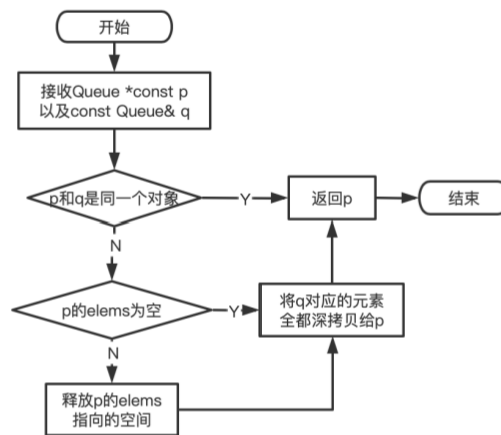


图 3 深拷贝赋值流程图

2.3.2 Queue*const assign(Queue*const p, Queue&&q)

功能：移动赋 s 给队列并返回 p

入口参数：队列指针 p，队列常引用 s

出口参数：无

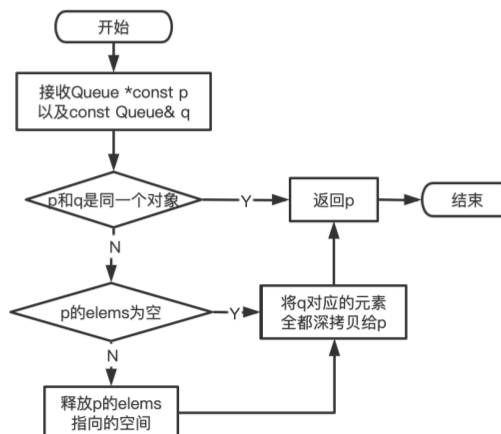


图 4 移动赋值流程图

2.4 打印 Queue

`char*print(const Queue *const p, char*s)`

功能：打印 p 指队列至 s 并返回 s

入口参数：队列指针 p，存放队列元素字符的字符指针 s

出口参数：队列元素字符串对应的字符指针

流程：从队首到队尾遍历一遍队列中的元素，每读到一个元素就将其与一个逗号分隔符拼接成字符串 s 中，返回队列元素字符串的字符指针。

2.5 销毁 Queue

`void destroyQueue (Queue *const p)`

功能：销毁 p 指向的队列

入口参数：队列指针 p

出口参数：无

流程：判断 p 指向队列中的 elems 是否为空，如果不为空，就将 elems 指向的空间释放，并赋值为 nullptr，同时将 max 赋值为 0。

三、软件开发

硬件环境：PC 机，Intel Core i5 四核 CPU 1.4GHz，8G 内存

使用操作系统：MacOS Big Sur11.6

程序运行平台：VS2019

编译模式：x86

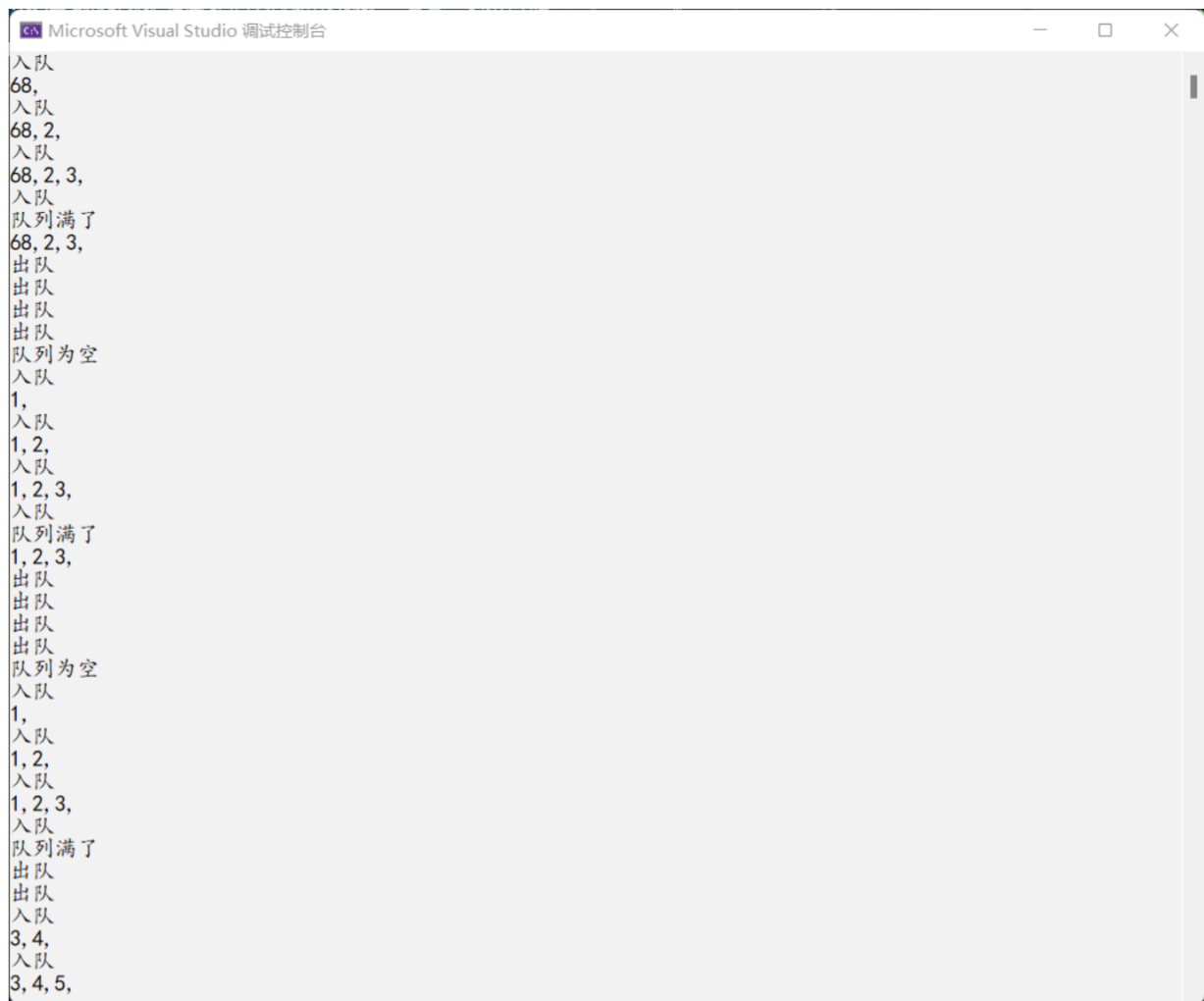
四、软件测试

测试结果以及得分如下图所示。

可以看到所有功能基本实现。

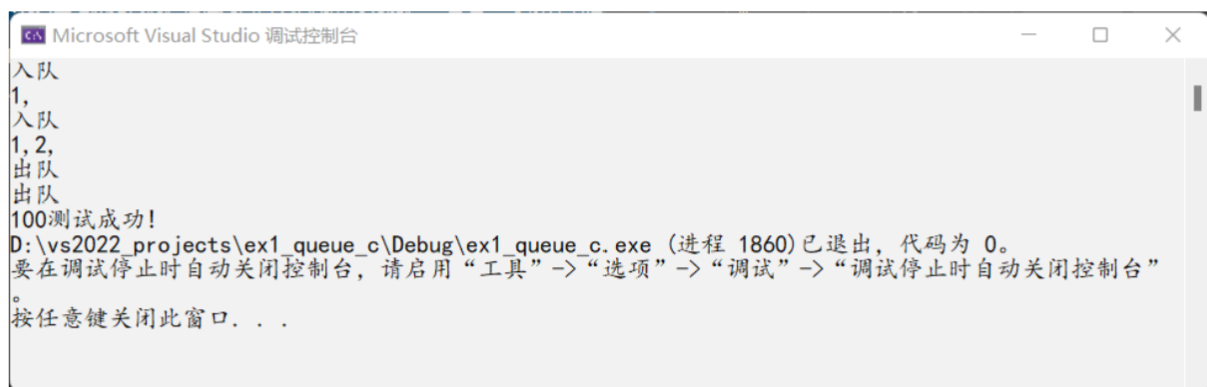


图 5 测试得分



```
Microsoft Visual Studio 调试控制台
入队
68,
入队
68, 2,
入队
68, 2, 3,
入队
队列满了
68, 2, 3,
出队
出队
出队
出队
队列为空
入队
1,
入队
1, 2,
入队
1, 2, 3,
入队
队列满了
1, 2, 3,
出队
出队
出队
出队
队列为空
入队
1,
入队
1, 2,
入队
1, 2, 3,
入队
队列满了
出队
出队
入队
3, 4,
入队
3, 4, 5,
```

图 6 测试结果 1



```
Microsoft Visual Studio 调试控制台
入队
1,
入队
1, 2,
出队
出队
100测试成功!
D:\vs2022_projects\ex1_queue_c\Debug\ex1_queue_c.exe (进程 1860) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口...
```

图 7 测试结果 2

五、特点与不足

1. 技术特点

通过数据结构的设计以及函数封装实现了模块化, 增加了复用性, 函数化、模块化的程序使得调试过

程更容易找到 bug 所在。对重要数据类型声明为常量，函数接口也设计为常量，增加了程序运行过程中的安全性。

2. 不足和改进的建议

队列的最大存放数据元素个数一旦经过初始化，就无法修改，对于开始对于数据量不定的问题，不能够保证取得一个合适的队列最大存放元素个数。

可以增加函数来实现队列的增加容量的功能，使得队列能够根据数据量持续增加自身所能存放的数据元素的个数，从而增加该程序的适应性。

六、过程和体会

1. 遇到的主要问题和解决方法

无法直接修改常量内容，通过强制类型转换，来改变常量中的值。

对于浅拷贝的考虑不周到，使得销毁队列的时候可能会出现再次释放已经被释放掉的内容，从而出现错误，在销毁队列函数中对 elems 是否为空进行单独的判断，避免重复释放。

2. 课程设计的体会

通过课程设计，对于 C++课程中学到的原理进行了实践，实现了队列的初始化、入队、出队等基本功能，对于队列的实现原理有了更深一步的理解。

对于浅拷贝、深拷贝过程中出现错误的情况进行 debug，从而提高了逻辑的严谨性。通过函数封装等，进一步理解了模块化编程。

通过 const 关键字的使用，更加理解安全性的用意。通过对整个程序的设计、编写、调试等强化了对程序的整体性的把握。

七、源码和说明

1. 文件清单及其功能说明

ex1_queue_c.h、ex1_queue_c.cpp、ex1_test.cpp、ex1_queue_c++.vcxproj、ex1_queue_c++.vcxproj.filters、ex1_queue_c++.vcxproj.user、queue.exe。

其中 queue.exe 为可执行程序，ex1_queue_c.h、ex1_queue_c.cpp、ex1_test.cpp 为源代码，ex1_queue_c++.vcxproj、ex1_queue_c++.vcxproj.filters、ex1_queue_c++.vcxproj.user 为 VS 工程文件。

2. 用户使用说明书

下载 queue.exe，执行 queue.exe

3. 源代码

ex1_queue_c.h

1. #define _CRT_SECURE_NO_WARNINGS
2. #include <stdio.h>
3. #include <string.h>

```

4. #include <stdlib.h>
5. struct Queue{
6.     int* const elems; //elems 申请内存用于存放队列的元素
7.     const int max; //elems 申请的最大元素个数 max
8.     int head, tail; //队列头 head 和尾 tail, 队空 head=tail;初始 head=tail=0
9. };
10. void initQueue(Queue *const p, int m); //初始化 p 指队列: 最多申请 m 个元素
11. void initQueue(Queue *const p, const Queue&s); //用 s 深拷贝初始化 p 指队列
12. void initQueue(Queue *const p, Queue&&s); //用 s 移动初始化 p 指队列
13. int number (const Queue *const p); //返回 p 指队列的实际元素个数
14. int size(const Queue *const p); //返回 p 指队列申请的最大元素个数 max
15. Queue*const enter(Queue*const p, int e); //将 e 入队列尾部, 并返回 p
16. Queue*const leave(Queue*const p, int &e); //从队首出元素到 e, 并返回 p
17. Queue*const assign(Queue*const p, const Queue&q); //深拷贝赋 s 给队列并返回 p
18. Queue*const assign(Queue*const p, Queue&&q); //移动赋 s 给队列并返回 p
19. char*print(const Queue *const p, char*s); //打印 p 指队列至 s 并返回 s
20. void destroyQueue (Queue *const p); //销毁 p 指向的队列

```

ex1_queue_c.cpp

```

1. #define _CRT_SECURE_NO_WARNINGS
2. #include <stdio.h>
3. #include <malloc.h>
4. #include <stdlib.h>
5. #include <cstring>
6. #include <iostream>
7. #include "ex1_queue_c.h"
8. void initQueue(Queue* const p, int m)
9. {
10.     if (m <= 0) {
11.         throw "Wrong argument";
12.     }
13.     if (p->elems) {
14.         free(*(int**)&p->elems);
15.     }
16.     *(int**)&p->elems = (int*)malloc(m * sizeof(int));
17.     *(int*)&p->max = m;
18.     p->head = p->tail = 0;
19. }
20. void initQueue(Queue* const p, const Queue& s)

```

```

21. {
22.     *(int**)&p->elems = (int*)malloc(s.max * sizeof(int));
23.     for (int i = 0; i < s.max; i++) {
24.         p->elems[i] = s.elems[i];
25.     }
26.     *(int*)&p->max = s.max;
27.     p->head = s.head;
28.     p->tail = s.tail;
29. }
30. void initQueue(Queue* const p, Queue&& s)
31. {
32.     *(int**)&p->elems = s.elems;
33.     *(int*)&p->max = s.max;
34.     p->head = s.head;
35.     p->tail = s.tail;
36.     if (s.elems)
37.         *(int**)&s.elems = nullptr;
38.     *(int*)&s.max = 0;
39.     s.head = s.tail = 0;
40. }
41. int number(const Queue* const p)
42. {
43.     if (!p->max)
44.         return 0;
45.     return (p->tail - p->head + p->max) % p->max;
46. }
47. int size(const Queue* const p)
48. {
49.     return p->max;
50. }
51. Queue* const enter(Queue* const p, int e)
52. {
53.     if ((p->tail + 1) % p->max == p->head)
54.         throw "Queue is full!";
55.     p->elems[p->tail] = e;
56.     p->tail = (p->tail + 1) % p->max;
57.     return p;
58. }
59. Queue* const leave(Queue* const p, int& e)

```

```

60. {
61.     if (p->head == p->tail) {
62.         throw "Queue is empty!";
63.     }
64.     e = p->elems[p->head];
65.     p->head = (p->head + 1) % p->max;
66.     return p;
67. }
68. Queue* const assign(Queue* const p, const Queue& q)
69. {
70.     if (p == &q) {
71.         return p;
72.     }
73.     if (*(int**)&p->elems) {
74.         free(*(int**)&p->elems);
75.     }
76.     *(int**)&p->elems = (int*)malloc(q.max * sizeof(int));
77.     for (int i = 0; i < q.max; i++) {
78.         p->elems[i] = q.elems[i];
79.     }
80.     *(int*)&p->max = q.max;
81.     p->head = q.head;
82.     p->tail = q.tail;
83.     return p;
84. }
85. Queue* const assign(Queue* const p, Queue&& q) {
86.     // 鍥旀攢鍥旀攢 s 鍥旀攢鍥旀攢鍥旀攢鍥旀攢鍥旀攢
87.     if (p == &q) {
88.         return p;
89.     }
90.     if (p->elems) { // 鍥旀攢 p 鍥�攢鍥�攢鍥�攢
91.         delete p->elems;
92.     } // 鍥�攢鍥�攢鍥�攢
93.     *(int**)&p->elems = q.elems;
94.     *(int*)&(p->max) = q.max;
95.     p->head = q.head;
96.     p->tail = q.tail;
97.     if (q.elems) { // 鍥�攢鍥�攢鍥�攢 free 鍥�攢鍥�攢 free 鍥�攢鍥�攢
98.         *(int**)&(q.elems) = nullptr;

```

```

99.     }
100.    *(int*)&(q.max) = 0;
101.    q.head = q.tail = 0;
102.    return p;
103. }
104. char* print(const struct Queue* const p, char* s)
105. {
106.     for (int i = p->head; i != p->tail; i = (i + 1) % p->max) {
107.         char* location = s + strlen(s);
108.         sprintf(location, "%d,", p->elems[i]);
109.     }
110.     return s;
111. }

112. void destroyQueue(Queue* const p)
113. {
114.     if (p->elems) {
115.         free(p->elems);
116.         *(int**)&(p->elems) = nullptr;
117.     }
118.     *(int*)&(p->max) = 0;
119. }

```

ex1_queue_test.cpp

```

1.  #include <iostream>
2.  using namespace std;
3.  extern const char * TestQueue(int &s);
4.  int main()
5.  {
6.      int s;
7.      string str=TestQueue(s);
8.      cout<<str<<endl<<s<<endl;
9.      return 0;
10. }

```