# 第3章　栈与队列

栈和队列是两种存取受到限制的线性数据结构。本章介绍栈和队列的定义、存储结构和基本操作，并以算术表达式求值为例介绍栈和队列的应用。

## 3.1　栈的基本概念

### 3.1.1　栈的定义与特点

栈(stacks)是限定在一端插入和删除的线性表。允许插入和删除的一端称为栈顶(top)，另一端称为栈底(bottom)。栈的修改是按照后进先出(last in first out，LIFO)的原则进行的，因此栈又称为后进先出(或先进后出)线性表。栈的基本结构如图 3.1 所示。

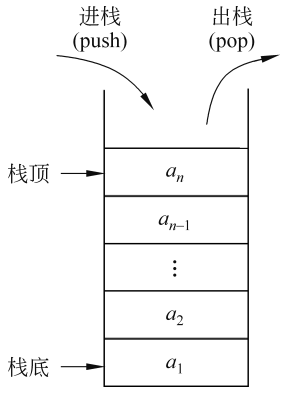


图 3.1　栈的基本结构

### 3.1.2　栈的两类存储结构

与线性表类似，栈也有顺序存储结构和链式存储结构。

**1. 顺序栈**

顺序栈，即栈的顺序存储结构。顺序栈利用一组地址连续的存储单元依次存储由栈底到栈顶的数据元素，同时附加 top 指针指示栈顶元素在顺序栈中的位置。顺序栈结构如图 3.2 所示。

利用栈底位置相对不变这个特点，两个顺序栈可以共享一个一维数据空间来互补余缺。其实现方法是将两个栈的栈底分设在一维数据空间的两端，并让它们各自的栈顶由两端向中间延伸，如图 3.3 所示。这样，两栈可以相互协调空间的使用，只有当整个数据空间被这两栈占满时才发生上溢。因此，上溢出现的频率比将这个一维数据空间一分为二给两个栈使用时要小。

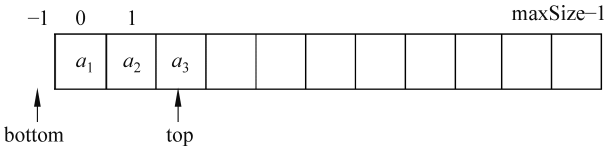


图 3.2　顺序栈示意图

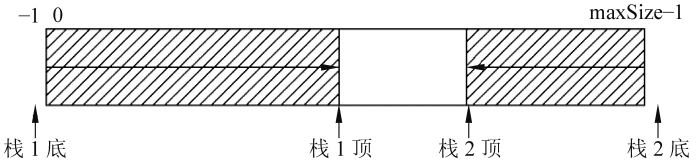


图 3.3　两个栈共享空间示意图

**2. 链栈**

链栈，即栈的链式存储结构。链栈动态分配元素存储空间，链栈无栈满问题，操作时无须考虑上溢问题，而且克服了顺序栈操作带来的数据元素移动问题。

由于栈的操作仅限制在栈顶进行，因此不必设置头结点，头指针即栈顶指针。链栈结构如图 3.4 所示。

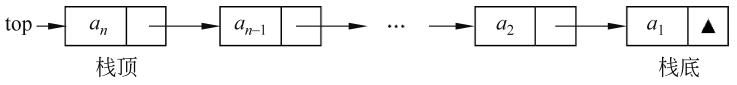



图 3.4　链栈示意图

## 3.2　顺序栈的算法实现

遵照和第 2 章类似的思路，在设计栈的结构及功能时，仍需要遵循面向对象的类的设计方法。首先，根据上面顺序栈的结构设计对应的类。

由于顺序栈本质上仍使用顺序表的存储结构，因此不妨直接继承第 2 章中已经定义的顺序表类，在此基础上添加或修改函数实现栈的不同功能。

同时，将顺序表的末端作为栈顶，如果需要栈顶元素，则直接将顺序表中的最后一个元素返回。在后续的功能补充当中，时刻注意顺序栈作为顺序表的派生类，拥有顺序表的所有属性和方法，并可调用顺序表的公有属性与方法。如果代码中出现读者未见过的函数，不妨回顾顺序表的定义进行检查。

顺序栈类定义如下。

```
1.  #include "../Vector/Vector.h"
2.
3.  //以顺序表为基类,派生出栈模板类
4.  template <typename T>
5.  class Stack : public Vector<T>
6.  { //将顺序表的末端作为栈顶
7.  public:
8.      //查看栈顶元素
9.      T top() { return this->get_elem(this->size()); }
10. };
11. //size()、empty()功能与原顺序表相同,直接继承
```

### 3.2.1　顺序栈的建立和顺序栈入栈

**1. 算法功能**

构造一个空栈 $s$，插入元素作为新的栈顶元素，实现栈的建立。

**2. 算法思路**

顺序栈入栈时栈顶指针 top 先增 1，再将新元素按 top 指示位置插入。注意，若栈满时再进栈，顺序表将进行扩容，避免了溢出错误。

**3. 实例描述**

建立一个顺序栈，依次将 1、2、3、4 顺序进栈，建栈过程如图 3.5 所示。

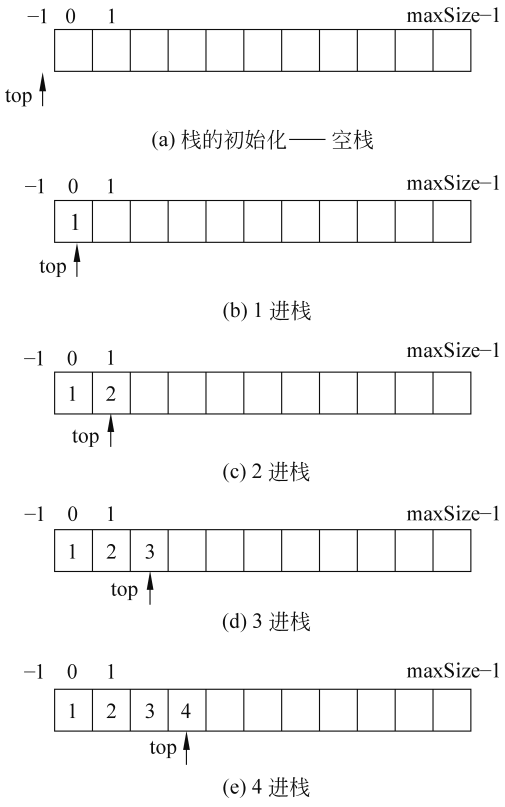



图 3.5　顺序栈建立和元素入栈过程

**4. 参考程序**

类中的相关函数如下。

```
1.  public:
2.      //入栈即将新元素插入顺序表末端
3.      void push(T const &e) { this->insert(e); }
```

定义文件中的主程序如下。

```
1.  #include <iostream>
2.  #include "Stack.h"
3.
4.  using namespace std;
5.
6.  void create_Stack(Stack<int>&s)
7.  {
8.      int e;
9.      cout <<"请输入数据(以 Ctrl+Z 结束输入):\n";
10.     while (cin >>e) s.push(e);
```

```
11.      cin.clear();                                    //更改 cin 的状态标识符
12.      rewind(stdin);                                  //清空输入缓存区
13.  }
14.
15.  int main()
16.  {
17.      Stack<int>stk;
18.      create_Stack(stk);
19.
20.      return 0;
21.  }
```

**5. 运行结果**

依次输入数据 1、2、3、4，按 Ctrl＋Z 组合键结束输入，建立顺序栈和元素入栈的过程如图 3.6 所示。

```
请输入数据（以Ctrl+Z结束输入）：
4 3 2 1 ^Z
```

图 3.6　建立顺序栈和元素入栈过程

### 3.2.2　顺序栈出栈

**1. 算法功能**

输出已知的顺序栈中的元素。

**2. 算法思路**

顺序栈出栈时，按栈顶指针 top 指示位置输出元素，并将栈顶指针 top 减 1。当栈为空时，则出栈结束。

**3. 实例描述**

将顺序栈 $s=\{1,2,3,4,5,6\}$ 中的元素逐个出栈，直到栈空，如图 3.7 所示。

**4. 参考程序**

类中的相关函数如下。

```
1.   public:
2.       //出栈即删除末端元素并返回其值
3.       T pop() { return this->remove(this->size()); }
```

定义文件中增添的相关函数和主函数修改如下。

```
1.   void print_Stack(Stack<int>&s) {
2.       cout <<"清空并输出,栈内共有" <<s.size() <<"个元素:\n";
3.       while (!s.empty()) cout <<s.pop() <<' ';
4.       cout <<endl;
5.   }
6.
7.   int main() {
8.       Stack<int>stk;
```

```
9.      create_Stack(stk);
10.     print_Stack(stk);
11.
12.     return 0;
13. }
```

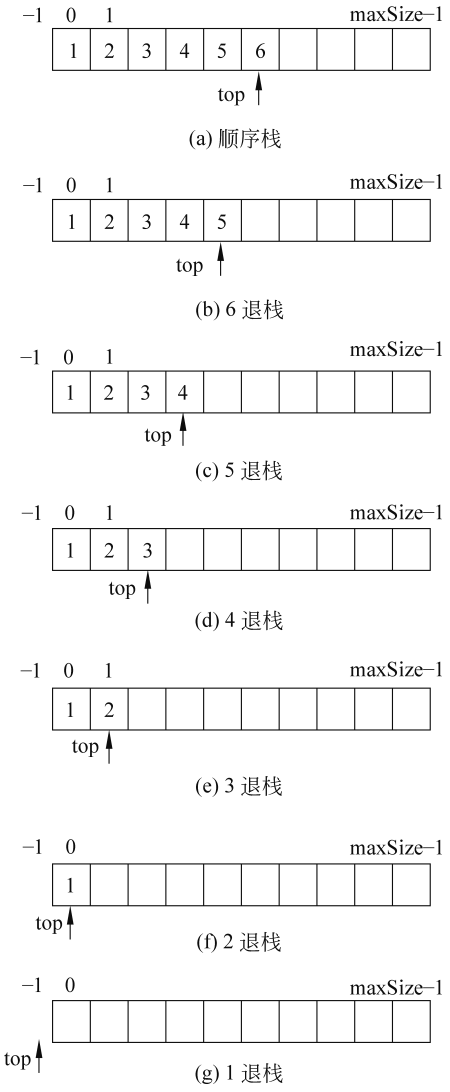



图 3.7　顺序栈出栈过程

### 5. 运行结果

顺序栈 $s=\{1,2,3,4,5,6\}$的元素出栈过程如图 3.8 所示。

```
请输入数据（以Ctrl+Z结束输入）：
1 2 3 4 5 6 ^Z
清空并输出，栈内共有6个元素：
6 5 4 3 2 1
```

图 3.8　顺序栈出栈算法演示

以上介绍的是顺序栈的功能及实现。顺序栈最终的 Stack.h 头文件及定义文件结构如图 3.9 和图 3.10 所示。

```
#ifndef _STACK_H_
#define _STACK_H_

#include "../Vector/Vector.h"
//以顺序表为基类，派生出栈模板类
template <typename T>
class Stack : public Vector<T>
{ //将顺序表的末端作为栈顶
public:
    //入栈即将新元素插入顺序表末端
    void push(T const &e) { this->insert(e); }
    //出栈即删除末端元素并返回其值
    T pop() { return this->remove(this->size() - 1); }
    //查看栈顶元素
    T top() { return this->get_elem(this->size()); }
};
//size()、empty()功能与原顺序表相同

#endif
```

图 3.9　顺序栈的头文件结构

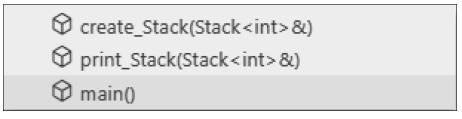



图 3.10　顺序栈的定义文件结构

另外，链栈的操作是单链表操作的特例，因此链栈的操作易于实现，在此不作赘述。

## 3.3　队列的基本概念

### 3.3.1　队列的定义与特点

队列是只允许在一端删除，在另一端插入的线性表。允许删除的一端称为队头（front），允许插入的一端称为队尾（rear），如图 3.11 所示。队列的修改是按照先进先出（first in first out，FIFO）的原则进行的。

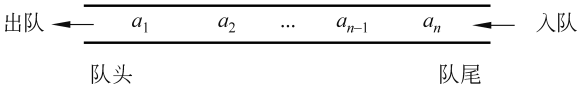



图 3.11　队列

队列数据结构在日常生活中随处可见。例如，在食堂排队买饭、在超市排队付钱等，遵守的规则都是先到先服务，这与队列的先进先出特点是一致的。

### 3.3.2 队列的存储结构

**1. 顺序队列**

顺序队列，即队列的顺序存储结构，也是利用一组地址连续的存储单元存放队列中的元素。由于队列中元素的插入和删除是在表的两端进行，因此必须设置队头指针 $q->front$ 和队尾指针 $q->rear$，分别指示当前的队头元素和队尾元素。

**2. 循环队列**

顺序队列会发生溢出现象。队满时再进行入队操作称为“上溢”，而队空时再进行出队操作称为“下溢”。上溢有两种情况。

一种是真正的溢出，如图 3.12 所示，即队尾指针 $q->rear$ 和队头指针 $q->front$ 存在如下关系：$q->rear-q->front=maxSize$，这时队列已满，不再有可供使用的数据空间。

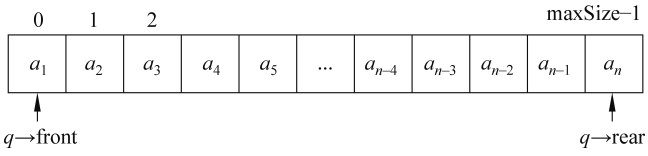



图 3.12 已满的顺序队列

另一种是假溢出，如图 3.13 所示，即 $q->rear=maxSize$，但是 $q->rear-q->front<maxSize$，这时仍有可用的数据空间，只不过队尾指针已经达界限的最大值而无法接纳入队的数据。假溢出是由于被删除的队列元素所占用的空间无法继续使用造成的。

解决假溢出的方法是将顺序队列设想成一个首尾相接的圆环，称为循环队列，如图 3.14 所示。

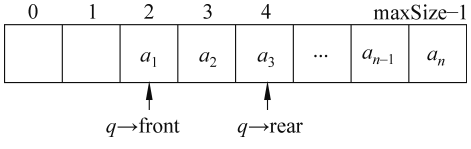



图 3.13 顺序队列假溢出

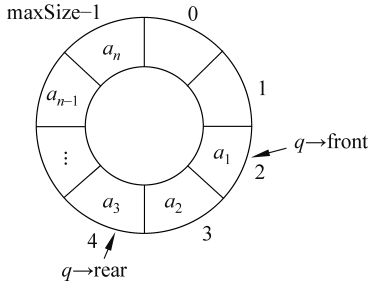



图 3.14 循环队列

循环队列中，队空和队满的条件都是 $q->rear=q->front$，因而无法区别队空和队满。解决这一问题的方法是浪费一个数据空间，将队满的条件改为 $(q->rear+1)\%maxSize=q->front$，而队空的条件不变。

**3. 链队列**

链队列，即队列的链式存储结构。链队列也需要标识队头和队尾的指针。与单链表相同，为了操作方便，一般使用带有头结点的链队列，并令头指针指向头结点。链队列在进队时无队满问题，但有队空问题。队空的条件是头指针和尾指针均指向头结点。链队列如

图 3.15 所示。

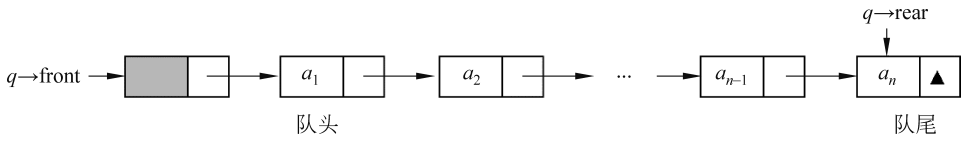



图 3.15　链队列

## 3.4　顺序队列的算法实现

首先，进行顺序队列的类设计。

根据上述顺序队列的结构，顺序队列应该和顺序表类似，应该包括最大容量、存储顺序表元素数组的数组指针等属性，具有类似的构造函数、析构函数以及返回长度、是否为空、是否为满等函数；同时，也应该包括队头指针和队尾指针这两个顺序表中没有的属性。故此处的顺序队列没有选择继承顺序表，而是选择了重新独立声明。

顺序队列类定义如下。

```
1.  #define DEFAULT_CAPACITY 10
2.
3.  //以顺序表的方式实现队列
4.  //因为设计有容量限制,所以未继承可扩展的 Vector
5.  template<typename T>
6.  class Queue {
7.  protected:
8.      int head, tail;
9.      int capacity;
10.     T * elem;
11.
12. public:
13.     int size() const { return tail -head; }            //返回规模大小
14.     bool empty() const { return !size(); }             //判断是否为空
15.     bool full() const { return tail >=capacity -1; }  //是否队满
16.
17.     //返回队首元素
18.     T front() { return elem[head +1]; }
19.
20.     //返回队尾元素
21.     T rear() { return elem[tail]; }
22. };
```

需要注意的是，由于队头指针和队尾指针的存在，顺序队列类和顺序表类的定义存在不少差异，最主要的区别是顺序队列类中没有 length 属性，因为 head 和 tail 指针就可以很好地完成相应的任务。因此，这里没有采用之前顺序栈类以顺序表类为基类派生定义的方法。在这里也提醒读者，如果希望继承某个类的特性，应该仔细观察派生类和基类的差别，以防出错。

### 3.4.1 顺序队列的建立和顺序队列入队

**1. 算法功能**

用尾插法动态建立顺序队列。

**2. 算法思路**

顺序队列入队时队尾指针先增1,再将新元素按队尾指针指示的位置插入。注意,若队满时再进队将产生溢出错误。

**3. 实例描述**

初始化顺序队列,并使关键字1、2、3、4、7依次入队,如图3.16所示。

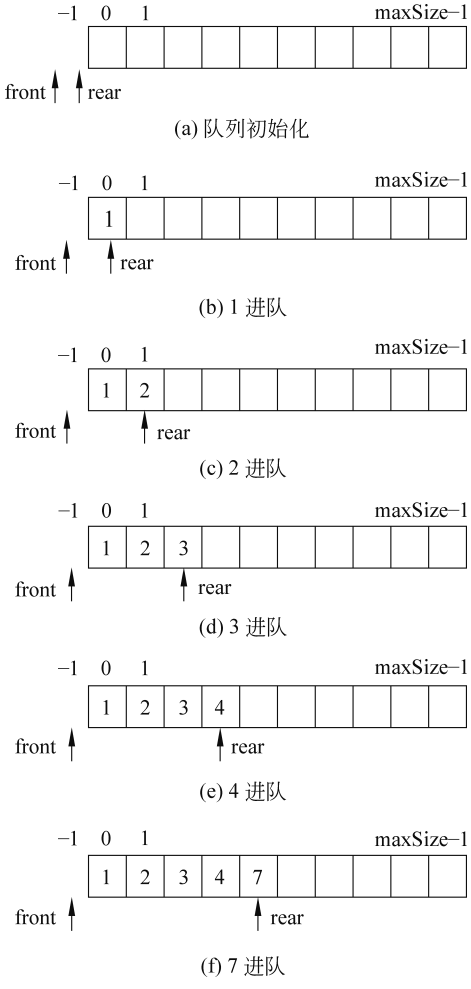



图3.16 顺序队列入队过程图示

**4. 参考程序**

Queue@Vector.h文件类中的相关函数如下。

```
1.  public:
2.      //构造函数初始化
```

```
3.      Queue(int c =DEFAULT_CAPACITY) : capacity(c)
4.      {
5.          elem =new T[capacity];
6.          head =tail =-1;
7.      }
8.      ~Queue() { delete[] elem; }                    //析构函数释放空间
9.
10.     //入队,即未满时在尾部插入
11.     void enqueue(T const &e) { elem[++tail] =e; }
```

定义文件中的主程序如下。

```
1.  #include "Queue@Vector.h"
2.  #include <iostream>
3.  using namespace std;
4.
5.  void create_Queue(Queue<int>&q) {
6.      int e;
7.      cout <<"请输入数据(以 Ctrl+Z 结束输入):\n";
8.      while (cin >>e)
9.          if (q.full()) cout <<"队列已满!\n";
10.         else q.enqueue(e);
11.     cin.clear();                                   //更改 cin 的状态标识符
12.     rewind(stdin);                                 //清空输入缓存区
13. }
14.
15. int main() {
16.     Queue<int>que;
17.     create_Queue(que);
18.     return 0;
19. }
```

需要注意的是,在本章队列的程序中,不同队列的定义文件是通用的,后续的程序中读者可以通过简单修改头文件引用语句直接使用上述的定义文件程序。

**5. 运行结果**

依次输入元素 1、2、3、4、7,按 Ctrl+Z 组合键结束输入,建立顺序队列,程序执行过程和结果如图 3.17 所示。

```
请输入数据（以Ctrl+Z结束输入）：
1 2 3 4 7 ^Z
```

图 3.17　顺序队列建立和元素入队算法演示

### 3.4.2　顺序队列出队

**1. 算法功能**

输出已知队列队头的元素。

**2. 算法思路**

出队时队头指针先增 1,再将队头指针所指示位置的元素输出。注意,若队空时再出队将做溢出处理。

**3. 实例描述**

将顺序队列 $q=\{1,3,2,7\}$中的元素逐个出队,直到队空为止,如图 3.18 所示。

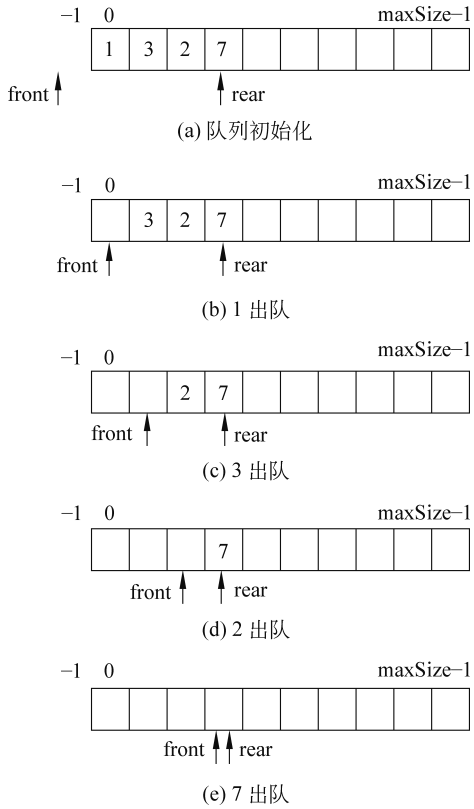



图 3.18 顺序队列出队过程

**4. 参考程序**

Queue@Vector.h 文件类中的相关函数如下。

```
1.   public:
2.       //出队,弹出首部并返回值
3.       T dequeue() { return elem[++head]; }
```

定义文件中增添的相关函数和主函数修改如下。

```
1.   void print_Queue(Queue<int>&q) {
2.       cout <<"清空并输出,队列内共有" <<q.size() <<"个元素:\n";
3.       while (!q.empty()) cout <<q.dequeue() <<' ';
4.       cout <<endl;
5.   }
6.
7.   int main() {
8.       Queue<int>que;
```

```
9.      create_Queue(que);
10.     print_Queue(que);
11.
12.     return 0;
13. }
```

**5. 运行结果**

顺序输入元素 1、3、2、7，按 Ctrl＋Z 组合键结束输入，建立顺序队列，并逐个输出队头元素，直到队空，如图 3.19 所示。

```
请输入数据（以Ctrl+Z结束输入）：
1 3 2 7 ^Z
清空并输出，队列内共有4个元素：
1 3 2 7
Press any key to continue . . .
```

图 3.19　顺序队列出队算法演示

以上介绍的是为顺序队列的功能及实现。顺序队列最终的头文件结构和定义文件结构分别如图 3.20 和图 3.21 所示。

```
#ifndef _QUEUE
#define _QUEUE

#define DEFAULT_CAPACITY 10

//以顺序表的方式实现队列
//因为设计有容量限制，所以未继承可扩展的Vector
template <typename T>
class Queue
{
    protected:
        int head , tail ;
        int capacity ;
        T *elem ;

    public:
        Queue(  int c = DEFAULT_CAPACITY ): capacity(c)//构造函数初始化
        {
            elem = new T[capacity] ;
            head = tail = -1 ;
        }

        ~Queue() { delete []elem ; }//析构函数释放空间

        int size() const { return tail-head ; }//返回规模大小
        bool empty() const { return !size() ; }//判断是否为空
        bool full() const { return tail ≥ capacity-1 ; }//判断是否为满

        //入队，即未满时在尾部插入
        void enqueue( T const& e ) { elem[++tail] = e ; }
        //出队，弹出首部并返回值
        T dequeue() { return elem[++head] ; }
        //返回队首元素
        T front() { return elem[head+1] ; }
        //返回队尾元素
        T rear() { return elem[tail] ; }

};

#endif
```

图 3.20　顺序队列最终的头文件结构

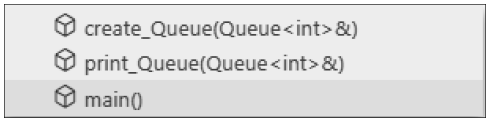



图 3.21 顺序队列最终的定义文件结构

## 3.5 循环队列的算法实现

首先,进行循环队列类的设计。

在之前循环队列存储结构的介绍中,如果从属性上看,它与顺序队列的区别并不大,也包括队首指针、队尾指针、最大容量和数组指针。

循环队列类定义如下。

```
#define DEFAULT_CAPACITY 10

template <typename T>
class Queue
{
protected:
    int head, tail;
    int capacity;
    T *elem;

public:
    int size() const                                    //返回规模大小
    {
        return (tail -head +capacity) %capacity;
    }

    //返回队首元素
    T front() { return elem[(head +1) %capacity]; }

    //返回队尾元素
    T rear() { return elem[tail]; }
};
```

但是,循环队列的结构中着重强调循环队列的环状结构的初始化以及它的队为空、为满的判定条件,读者需要注意循环队列与顺序队列的不同。

### 3.5.1 循环队列的建立和循环队列入队

**1. 算法功能**

循环队列初始化,并在队尾插入新元素。

**2. 算法思路**

存储队列的数组被当成首尾相接的表处理。

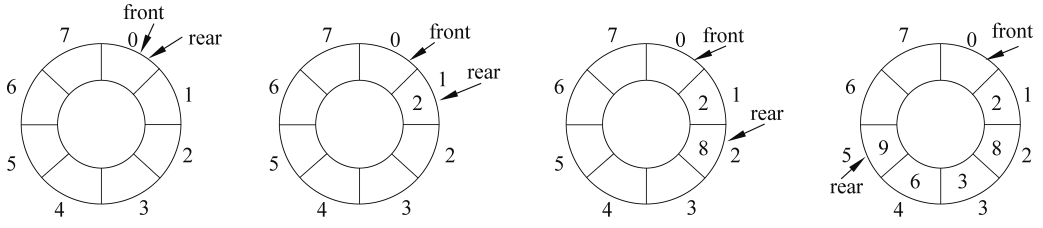
队列初始化：front＝rear＝0。

队尾指针进 1：rear＝(rear＋1)％ maxSize。

队满条件：(rear＋1)％ maxSize＝front。

**3. 实例描述**

建立循环队列{2,8,3,6,9}，循环队列入队过程如图 3.22 所示。其中，图 3.22(a)为循环对列初始化，图 3.22(b)是 2 入队，图 3.22(c)为 8 入队，图 3.22(d)是 3、6、9 依次入队。




(a) 循环队列初始化　(b) 2 入队　(c) 8 入队　(d) 3、6、9 依次入队

图 3.22　循环队列入队过程

**4. 参考程序**

CQueue.h 文件类中相关的函数如下。

```
1.   public:
2.       Queue(int c =DEFAULT_CAPACITY) :capacity(c)    //构造函数初始化
3.       {
4.           elem =new T[capacity];
5.           head =tail =0;
6.       }
7.
8.       ~Queue() { delete[] elem; }                    //析构函数释放空间
9.       //判断是否为空
10.      bool empty() const { return head ==tail; }
11.      //判断是否为满
12.      bool full() const { return head ==(tail +1) %capacity; }
13.
14.      //入队，即未满时在尾部插入
15.      void enqueue(T const &e)
16.      {
17.          if (full())
18.              return;
19.          tail =(tail +1) %capacity;
20.          elem[tail] =e;
21.      }
```

对于定义文件，只需要将 3.4.1 节顺序队列建立和入队功能中定义文件中引入头文件的 include 语句修改为如下语句即可。

```
1.   #include "CQueue.h"                              //引入循环队列
```

**5. 运行结果**

依次输入数据元素 2、8、3、6、9，以 Ctrl＋Z 结束输入，建立一个循环队列，如图 3.23 所示。

```
请输入数据（以Ctrl+Z结束输入）：
2 8 3 6 9 ^Z
```

图 3.23　循环队列入队算法演示

### 3.5.2　循环队列出队

**1. 算法功能**

循环队列队头元素出队。

**2. 算法思路**

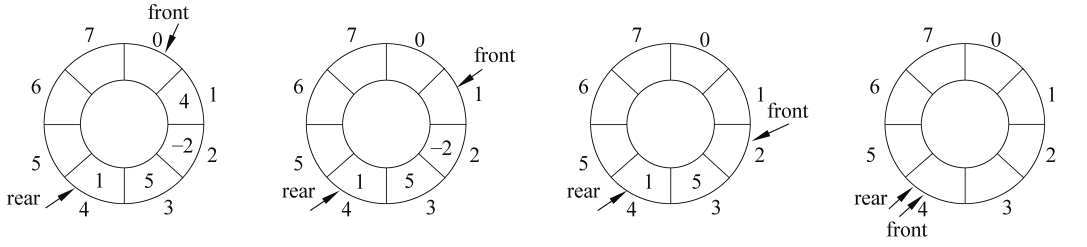
存储队列的数组被当作首尾相接的表处理。

队头指针进 1：front＝(front＋1)％ maxSize。

队空条件：front＝rear。

**3. 实例描述**

将循环队列 q＝{4，－2，5，1}中的数据元素依次出队，直到该循环队列为空，如图 3.24 所示。其中，图 3.24(a)是原始循环队列，图 3.24(b)是队头元素 4 出队，图 3.24(c)是队头元素－2 出队，图 3.24(d)是队头元素 5、1 依次出队。




(a) 循环队列　(b) 队头元素 4 出队　(c) 队头元素 -2 出队　(d) 队头元素 5、1 依次出队

图 3.24　循环队列出队过程

**4. 参考程序**

CQueue.h 文件类中相关的函数如下。

```
1.   public:
2.       //出队,非空时弹出首部并返回值
3.       T dequeue()
4.       {
5.           if (empty()) return {};
6.           head = (head +1) %capacity;
7.           return elem[head];
8.       }
```

对于定义文件，将 3.4.2 节顺序队列出队功能中定义文件中引入头文件的 include 语句修改为以下语句即可。

```
1.   #include "CQueue.h"                                    //引入循环队列
```

**5. 运行结果**

依次输入元素 4、－2、5、1，以 Ctrl＋Z 结束输入，建立一个循环队列，然后将队列中的元

素出队，直到循环队列为空，如图 3.25 所示。

```
请输入数据（以Ctrl+Z结束输入）：
4 -2 5 1 ^Z
清空并输出，队列内共有4个元素：
4 -2 5 1
```

图 3.25　循环队列出队算法演示

以上介绍的是循环队列的功能及实现。循环队列最终的头文件结构如图 3.26 所示。

```
#ifndef _QUEUE
#define _QUEUE

#define DEFAULT_CAPACITY 10

template <typename T>
class Queue
{
protected:
    int head, tail;
    int capacity;
    T *elem;

public:
    Queue(int c = DEFAULT_CAPACITY) : capacity(c) //构造函数初始化
    {
        elem = new T[capacity];
        head = tail = 0;
    }

    ~Queue() { delete[] elem; } //析构函数释放空间

    int size() const { return (tail - head + capacity) % capacity; } //返回规模大小
    bool empty() const { return head == tail; }                       //判断是否为空
    bool full() const { return head == (tail + 1) % capacity; }       //判断是否为满

    //入队，即未满时在尾部插入
    void enqueue(T const &e)
    {
        if (full())
            return;
        tail = (tail + 1) % capacity;
        elem[tail] = e;
    }
    //出队，非空时弹出首部并返回值
    T dequeue()
    {
        head = (head + 1) % capacity;
        return elem[head];
    }
    //返回队首元素
    T front() { return elem[(head + 1) % capacity]; }
    //返回队尾元素
    T rear() { return elem[tail]; }
};

#endif
```

图 3.26　循环队列最终的头文件结构

## 3.6 链队列的算法实现

观察链队列的结构，不难发现它的结构和之前学习过的双向链表的结构具有很高的相似性。具体来说，如果将双向链表的头结点和尾结点视为链队列的队头指针和队尾指针，那么双向链表就能够很好地支持链队列的所有功能。在后续的使用中，双向链表派生出的链队列也将是本书中默认的队列结构。

因此，链队列类以 2.4 节中的双向链表为基类进行派生，定义如下。

```
#include "../List/DuList.h"

//以双向链表为基类,派生出队列模板类
template<typename T>
class Queue : public DuList<T>{
public:
    //链式队列不会存在填满的情况
    bool full() const { return false; }

    //返回队首元素
    T front() { return this->first()->data; }

    //返回队尾元素
    T rear() { return this->last()->data; }
};
```

### 3.6.1 链队列的建立和链队列入队

**1. 算法功能**

用双向链表存储队列，进行队列初始化并在队尾插入元素。

**2. 算法思路**

利用双向链表类中已经定义好的插入函数 insert_last()，使用尾插法创建、插入队列元素。

**3. 实例描述**

依次输入数据元素 7、－8、3、6、2，建立链队列的过程和用尾插法创建一个包含数据元素 7、－8、3、6、2 的双向链表的过程完全一致，在此不再赘述。

**4. 参考程序**

Queue.h 文件类中相关的函数如下。

```
public:
    //入队,即在尾部插入
    void enqueue(T const &e) { this->insert_last(e); }
```

对于定义文件，将 3.4.1 节顺序队列建立和入队功能中定义文件中引入头文件的

include 语句修改为以下语句即可。

```
1.  #include "Queue.h"                                    //引入链队列
```

**5. 运行结果**

依次在队尾插入数据 7、－8、3、6、2，按 Ctrl＋Z 组合键结束输入，建立链队列和元素入队的过程如图 3.27 所示。

```
请输入数据（以Ctrl+Z结束输入）：
7 -8 3 6 2 ^Z
```

图 3.27　建立链队列和元素入队的过程

### 3.6.2　链队列出队

**1. 算法功能**

链队列中的元素出队。

**2. 算法思路**

利用双向链表类中已经定义好的返回链表首个元素的函数 first()以及删除函数 remove()，返回并删除链表中的首个元素。

**3. 实例描述**

将链队列{－2，3，5，1，－3}中的数据元素依次出队，直到队空为止，过程类似从头至尾打印并删除双向链表中的所有元素。

**4. 参考程序**

Queue.h 文件类中相关的函数如下。

```
1.  public:
2.      //出队，弹出队头并返回值
3.      T dequeue() { return this->remove(this->first()); }
```

对于定义文件，将 3.4.2 节顺序队列出队功能中定义文件中引入头文件的 include 语句修改为以下语句即可。

```
1.  #include "Queue.h"                                    //引入链队列
```

**5. 运行结果**

依次输入元素－2、3、5、1、－3，按 Ctrl＋Z 组合键结束输入，建立链队列，然后将队列中的元素出队，直到队空，如图 3.28 所示。

```
请输入数据（以Ctrl+Z结束输入）：
-2 3 5 1 -3 ^Z
清空并输出，队列内共有5个元素：
-2 3 5 1 -3
```

图 3.28　链队列出队算法演示

以上介绍的是链队列的功能及实现。链队列最终的头文件结构如图 3.29 所示。

```
#ifndef _QUEUE_H_
#define _QUEUE_H_

#include "../List/DuList.h"
//以双向链表为基类，派生出队列模板类
template <typename T>
class Queue : public DuList<T>
{
public:
    //链式队列不会存在填满的情况
    bool full() const { return false; }
    //入队，即在尾部插入
    void enqueue(T const &e) { this->insert_last(e); }
    //出队，弹出队头并返回值
    T dequeue() { return this->remove(this->first()); }
    //返回队首元素
    T front() { return this->first()->data; }
    //返回队尾元素
    T rear() { return this->last()->data; }
};

#endif
```

图 3.29　链队列最终的头文件结构

## 3.7　栈和队列的应用——算术表达式的转化和求值

**1. 算法功能**

算术表达式的表示方法有中缀表达式(infix)、前缀表达式(prefix)和后缀表达式(postfix)3 种。这里的前、中、后是针对运算符和操作数放置的位置而言的。

中缀表达式是将运算符(operator)写在两个操作数(operand)之间。例如，$A+B$，$A+B/(C-D)$。

前缀表达式是把运算符写在两个操作数之前。例如，$+AB$，$+A/B-CD$。

后缀表达式是把运算符写在两个操作数之后。例如，$AB+$，$ABCD-/+$。

表达式的转化可以分为中缀转前缀和中缀转后缀，其转换的方法非常类似，只需要在满足运算符的优先级(priority)的前提下，注意把运算符放在操作数之前或者之后。中缀表达式 $A+(B-C\times D)/E$ 的转化过程见表 3.1。

**表 3.1　中缀表达式 $A+(B-C\times D)/E$ 的转化过程**

| 步骤 | 说　明 | 前缀表达式 | 后缀表达式 |
|---|---|---|---|
| 1 | 乘法转换 | $A+(B-\times CD)/E$ | $A+(B-CD\times)/E$ |
| 2 | 减法转换 | $A+(-B\times CD)/E$ | $A+(BCD\times-)/E$ |
| 3 | 除法转换 | $A+/(-B\times CD)\ E$ | $A+(BCD\times-)\ E/$ |
| 4 | 加法转换 | $+A/(-B\times CD)\ E$ | $A\ (BCD\times-)\ E/+$ |
| 5 | 去括号 | $+A/-B\times CDE$ | $ABCD\times-E/+$ |

前缀和后缀表达式本质上是将运算顺序用表达式的排列顺序呈现出来，后缀表达式中越靠前的运算符越优先运算，而前缀表达式中恰好相反。这样一来就省去了括号，也不需要预先定义运算符的优先顺序，同时对于计算机来说这是一个更加容易处理的方式。由于后缀表达式比前缀表达式用得普遍，本算法着重介绍算术表达式的中缀表达式如何转换成后缀表达式，并利用后缀表达式求值。

**2. 算法思路**

1）中缀表达式转后缀表达式

如果将中缀表达式分为操作数和运算符两个部分，转换前后操作数的顺序显然是不变的，因此不妨将所有操作数视为一个队列，先进队的先输出，这样就能保证顺序不改变；而运算符的顺序则取决于它的优先级，可以将所有运算符用栈来存放，保证栈顶运算符优先级最高，如果新运算符优先级小于栈顶元素时，就输出栈顶运算符，直到新元素可以入栈为止，这样就能使运算符按照优先级输出。

为输入输出方便，这里用井号“#”记作表达式的结束符；同时为了算法简洁，在表达式的最左边也虚设一个#构成整个表达式的一对括号。

此时算法的具体规则如下。

（1）若导入的是操作数，则直接输出到队列。

（2）若当前运算符的优先级高于栈顶运算符的优先级，则入栈。

（3）若当前运算符的优先级低于栈顶运算符的优先级，则栈顶运算符出栈，并输出到队列，当前运算符再与新的栈顶运算符比较。

（4）若当前运算符的优先级等于栈顶运算符，且栈顶运算符为左圆括号，当前运算符为右圆括号，则栈顶运算符出栈，继续读下一符号。

（5）若当前运算符的优先级等于栈顶运算符，且栈顶运算符为#，当前运算符也为#，则栈顶运算符出栈，输出队列中的数值即可。

运算符的优先级如表 3.2 所示，其中 $\theta_1$ 为栈顶运算符，$\theta_2$ 为当前运算符。

**表 3.2　运算符的优先级**

| $\theta_1$ \ $\theta_2$ | + | − | × | / | ( | ) | # |
|---|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| − | > | > | < | < | < | > | > |
| × | > | > | > | > | < | > | > |
| / | > | > | > | > | < | > | > |
| ( | < | < | < | < | < | = | |
| ) | > | > | > | > | | > | > |
| # | < | < | < | < | < | | = |

2）后缀表达式的运算

后缀表达式中所有的运算符的运算都是以它之前紧邻的两个操作数进行的，因此，如果顺序读取表达式中的元素，将表达式中的操作数依次顺序存放在栈里，那么若遇到运算符，