Chapter 7 QUEUES & STACKS

QUEUES 队列

- 对队列的操作**只能在后端插入，在前端删除、检索和修改**（不适用于数组和向量，适用于链表和双端队列）
  - push(enqueue)：从后推入
  - pop(dequeue)：删除前端
  - FIFO(first-in, first-out)：**先进先出**（栈则相反-后进先出-LIFO-last-in,first-out）
- 数组和向量无法用于实现队列，因为实现插入和修改操作过于繁琐
- 链表和双端队列可以实现队列

补充：

在路径搜索方法中涉及队列

宽度优先使用队列，深度优先使用栈

能用递归的地方也可以用栈，递归实质上就是栈操作

- **队列也是模板类**

```
template <class T, class container = deque<T>>

explicit queue(const Container& - Container());
```

- **队列的元素与实现**
  - 特点：不需要新结构实现，只需借助已有容器结构实现

    第二个参数T的容器类型为实现该队列的**底层容器类型**（不指定则为**默认值 - 双端队列**）
  - 队列的元素实际存储在实现其的容器之中（同样都为T）
  - 示例
    - 声明一般为第一种 但需要知道底层容器是什么

- explicit - 显式；implicit - 隐式

```
1.  // Precondition: this queue object has been initialized
    //                with a copy of this Container object.
    explicit queue (const Container&  = Container( ));
```

Here are equivalent examples
for a queue of items of type plane;
the items are stored in a deque:

```
queue<Plane> runWay;
queue<Plane, deque<Plane> > runWay;
queue<Plane> runWay (deque<Plane>( ));
queue<Plane, <deque<Plane> > runWay (deque<Plane>( ));
```

```
2.  // Poscondition: true has been returned if this queue object
    //                is empty. Otherwise, false has been
    //                returned.
    bool empty( ) const;

3.  // Postcondition: the number of items in this queue object
    //                has been returned.
    unsigned size( ) const;

4.  // Postcondition: the item x has been inserted at the back of
    //                this queue.
    void push (const value_type& x);   // Container class has
                                       //  typedef T value_type
```

- front( )
  - 重载 - 自动匹配是否需要常量const

```
5.  // Precondition: this queue is not empty.
    // Postcondition: a reference to the item at the front of
    //                this queue has been returned.
    T& front( );
    const T& front();

6.  // Precondition: this queue is not empty.
    // Postcondition: the item that had been at the front of
    //                this queue before this call has been
    //                deleted from this queue.
    void pop( );
```

⭐队列没有迭代器Iterators，因为只有前端元素可访问和操作

```
// 用例
queue<int> my_queue; // 此处没有声明底层容器，即为默认的双端队列
for (int i=0; i<10; i++)
    my_queue.push(i*i); // 从队列后端插入新元素（顺序）
while (!my_queue.empty())
{
    cout << my_queue.front() << endl; // 输出前端元素 此处自动匹配执行非常量的构造函数
    my_queue.pop(); // 删除前端元素（然后可以输入下一元素）
}
```

Container Adapter 容器适配器

定义：提供了简化的接口，允许通过特定的数据结构来组织和管理数据。容器适配器不是标准的容器，而是建立在标准容器之上的一种包装器，提供了不同的接口和功能。

三种常见的容器适配器：栈（stack）、队列（queue）和优先队列（priority_queue）

队列

- 借助于已实现的底层容器的方法实现新队列的操作

- 为新容器定义新的方法
- **队列元素实际上是存入底层容器container中**
- 不能类比于派生继承，因为此处队列方法已变，导致程序行为不可替换。即若原本为双端队列方法push_back()，此时实现队列操作，但队列操作只有push()，没有push_back()。

▼ 可作为底层容器的类型

- 双端队列可作为容器（插入删除操作时间复杂度为常量）
  - 例外：push操作的最坏时间复杂度为O(n)
- 链表可作为容器（插入删除操作时间复杂度为常量）
- 向量不可以作为容器（前端操作的时间复杂度为O(n)，且没有pop_front）

⭐实现对已有容器改造的最简单方式：**封装一个底层容器的对象**（将底层容器作为数据成员）

- 队列类依赖于底层容器类方法
  - push( )
    - 相当于push( )到底层容器的后端 于是调用底层容器的pushback( )
    - c.pushback( )
  - pop( )
    - 调用底层容器的pop_front( )
    - c.pop_front( )
  - front( )
    - c.front()
  - back( )
    - c.back()
  - size( )
    - 队列有多少元素取决于底层容器对象有多少元素
    - return c.size( )
  - empty( )

- bool类型
- 队列是否为空取决于底层容器对象是否为空
- return c.empty( )

```
class queue{
    protected：
        Container c;

    public:
        bool empty(){
            return c.empty;}
        void push(const value_type& x){
            c.push_back(x);}
        void pop(){
            c.pop_front();}
        const T& front() const{
            return c.front();}
```

补充：回看递归章节最后的例子（封装函数）

- fields - 数据成员

Computer simulation 计算机模拟

- 模型是系统的简化形式
- 物理模型
- 数学模型

队列用例：洗车

- 时间相同时先处理DEPARTURE再处理ARRIVAL

4，8，12，16，23，999

time	event	waiting time

4	A	
8	A	
12	A	
14	D	0
16	A	
23	A	
24	D	6
34	D	12
44	D	18
54	D	21

(6+12+18+21)/5 = 11.4

```
class CarWash{
    CarWash();
    void runSimulation();
    void printResult();
}

queue<Car> carQueue;


CarWash::CarWash()
{
        currentTime = 0;
        numberOfCars = 0;
        sumOfWaitingTimes = 0;
        nextDepartureTime = INFINITY; // = 10000
} // default constructor

void CarWash::runSimulation()
{
    const string PROMPT =
        "\nPlease enter the next arrival time. The sentinel
is  ";
```

```
    const int SENTINEL = 999; //定义最大时间值

        queue<Car> carQueue; // 洗车队列

        int nextArrivalTime;

    cout << PROMPT << SENTINEL << endl;
    cin >> nextArrivalTime;

        while (nextArrivalTime != SENTINEL) // 判断下一辆车到
达时间
    {
        if (nextArrivalTime < nextDepartureTime) // 下一辆车
到达时间先于当前车辆离开时间
            {
             processArrival (nextArrivalTime, carQueue);
           cout << PROMPT << SENTINEL << endl;
           cin >> nextArrivalTime;
        } // if
        else
           processDeparture (carQueue);
      } // while SENTINEL not reached

    // Wash any cars remaining on the carQueue.
        while (nextDepartureTime < INFINITY)
             processDeparture (carQueue);
} // runSimulation

void processArrival (int nextArrivalTime, queue<Car>& carQu
eue)
{
        const string OVERFLOW = "Overflow";

        currentTime = nextArrivalTime;
        if (carQueue.size( ) == MAX_SIZE)
                cout << OVERFLOW << endl;
        else
        {
```

```
                numberOfCars++;
                if (nextDepartureTime == INFINITY)
                    nextDepartureTime = currentTime + WASH_
TIME;
                else
                    carQueue.push (Car (nextArrivalTime));
        } // not an overflow
} // method processArrival


void CarWash::processDeparture (queue<Car>& carQueue)
{
    int waitingTime;

        cout << "departure time = " << nextDepartureTime <<
endl;
        currentTime = nextDepartureTime;
        if (!carQueue.empty())
        {
                Car car = carQueue.front();
                carQueue.pop();
                waitingTime = currentTime - car.getArrivalT
ime();
                sumOfWaitingTimes += waitingTime;
                nextDepartureTime = currentTime + WASH_TIM
E;
        } // carQueue was not empty
    else
                nextDepartureTime = INFINITY;
} // method processDeparture

void CarWash::printResult( )
{
        const string NO_CARS_MESSAGE =
        "There were no cars in the car wash.";

    const string AVERAGE_WAITING_TIME_MESSAGE =
            "\nThe average waiting time, in minutes, was ";
```

```
        if (numberOfCars == 0)
            cout << NO_CARS_MESSAGE << endl;
        else
            cout << AVERAGE_WAITING_TIME_MESSAGE
                 << ((double)sumOfWaitingTimes / numberOfC
ars)
                 << endl;
} // method printResult
```

表示车洗完了：将nextDeparture设置为INFINITY

STACKS 栈

- 栈的特点：后进先出（LIFO）；队列特点：先进先出（FIFO）
- top - 栈顶
- push - 插入元素到栈顶
- pop - 栈顶输出
- 栈不指定底层容器**默认仍为双端队列**
- 集中在“后端”（栈顶）操作（双端队列、链表、**向量**都可作为其底层容器）
- **栈没有iterator！！（只有栈顶元素可被操作**

```
template <class T, Container = deque<T>>
// stack<int, vector<int>> myStack;

class Stack{
    protected:
        Container c; // 默认为deque
    public：
        bool empty(){
            return c.empty();
        }
        long size(){
            return c.size();
```

```
        }
        void push(const value_type& x){ // 为什么是常量？？
            return c.push_back(x);
        }
        T& top();
        ...
```

栈的应用

- **递归可以通过栈操作（迭代）替代**

Recall from chapter 4: decimal to binary:

```
// Precondition: n >= 0.
// Postcondition: The binary equivalent of n has been
//                printed. The worstTime(n) is O(log n).
void writeBinary (int n)
{
    if (n == 0 || n == 1)
        cout << n;
    else
    {
        writeBinary (n / 2);
        cout << n % 2;
    } // else
} // writeBinary
```

Here is a stack-based version:

```
void writeBinary (int n)
{       stack<int> myStack;
        myStack.push (n);
        while (n > 1)
        {   n = n / 2;
            myStack.push (n);
        } // pushing
        while (!myStack.empty())
        {   n = myStack.top();
            myStack.pop();
            cout << (n % 2);
        } // popping
        cout << endl << endl;
} // method writeBinary
```

中缀INFIX

- 操作符在操作数中间

后缀POSTFIX表达式

※中缀优先级高才推入 低或相同则推出

- 过程类似于栈运算
- 操作符紧接在其操作数之后
- 特征后缀表达式次序和中缀表达式（普通）次序相同
- 中缀表达式转换成后缀表达式
    - 如果stack为空或中缀运算符的优先级比stack顶部运算符高，则运算符进入stack；否则弹出stack顶部对象加入到后缀字符串

- 遇到左括号可直接入栈 但入栈后优先级变为最低 遇到右括号则栈内运算符直接出栈 直到丢弃左括号

前缀PREFIX表达式

- 操作符在操作数前
- 利用两个stack 分别放置操作数和操作符