第3章 栈和队列

3.1 验证性实验

实验题 1：实现顺序栈的各种基本运算的算法

目的：领会顺序栈的存储结构和掌握顺序栈中各种基本运算算法的设计。

内容：编写一个程序 sqstack.cpp，实现顺序栈（假设栈中的元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-1.cpp 完成以下功能。

（1）初始化栈 s。

（2）判断栈 s 是否非空。

（3）依次进栈元素 a、b、c、d、e。

（4）判断栈 s 是否非空。

（5）输出出栈序列。

（6）判断栈 s 是否非空。

（7）释放栈。

根据《教程》中 3.1.2 节的算法得到 sqstack.cpp 程序，其中包含如下函数。

- InitStack(SqStack *&s)：初始化顺序栈 s。
- DestroyStack(SqStack *&s)：销毁顺序栈 s。
- StackEmpty(SqStack *s)：判断顺序栈 s 是否为空栈。
- Push(SqStack *&s,ElemType e)：元素 e 进顺序栈。
- Pop(SqStack *&s,ElemType &e)：元素 e 出顺序栈。
- GetTop(SqStack *s,ElemType &e)：取顺序栈的栈顶元素 e。

对应的程序代码如下（设计思路详见代码中的注释）：

```
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100
typedef char ElemType;
typedef struct
{   ElemType data[MaxSize];
    int top;                          //栈顶指针
} SqStack;                            //声明顺序栈类型
void InitStack(SqStack *&s)           //初始化顺序栈
{   s=(SqStack *)malloc(sizeof(SqStack));
    s->top=-1;
}
void DestroyStack(SqStack *&s)        //销毁顺序栈
{
    free(s);
}
bool StackEmpty(SqStack *s)           //判断栈是否为空栈
{
    return(s->top==-1);
```

```
}
bool Push(SqStack *&s,ElemType e)          //进栈
{   if (s->top==MaxSize-1)                 //栈满的情况,即栈上溢出
        return false;
    s->top++;
    s->data[s->top]=e;
    return true;
}
bool Pop(SqStack *&s,ElemType &e)          //出栈
{   if (s->top==-1)                        //栈为空的情况,即栈下溢出
        return false;
    e=s->data[s->top];
    s->top--;
    return true;
}
bool GetTop(SqStack *s,ElemType &e)        //取栈顶元素
{   if (s->top==-1)                        //栈为空的情况,即栈下溢出
        return false;
    e=s->data[s->top];
    return true;
}
```

实验程序 exp3-1.cpp 的结构如图 3.1 所示,图中方框表示函数,方框中指出函数名;箭头方向表示函数间的调用关系;虚线方框表示文件的组成,即指出该虚线方框中的函数存放在哪个文件中。

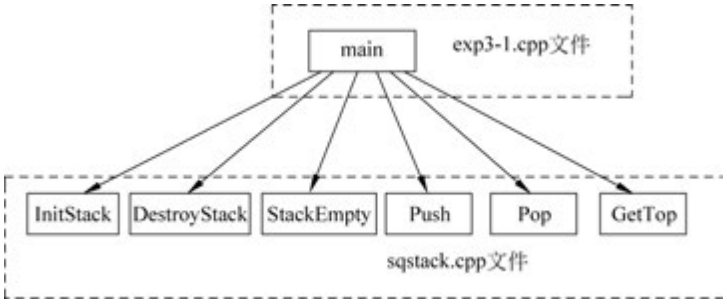


图 3.1 exp3-1.cpp 程序的结构

实验程序 exp3-1.cpp 的代码如下:

```
#include "sqstack.cpp"                     //包含顺序栈的基本运算算法
int main()
{   ElemType e;
    SqStack *s;
    printf("顺序栈 s 的基本运算如下:\n");
    printf("  (1)初始化栈 s\n");
    InitStack(s);
    printf("  (2)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (3)依次进栈元素 a,b,c,d,e\n");
```

```
    Push(s,'a');
    Push(s,'b');
    Push(s,'c');
    Push(s,'d');
    Push(s,'e');
    printf("  (4)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (5)出栈序列:");
    while (!StackEmpty(s))
    {   Pop(s,e);
        printf("%c ",e);
    }
    printf("\n");
    printf("  (6)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (7)释放栈\n");
    DestroyStack(s);
    return 0;
}
```

exp3-1.cpp 程序的执行结果如图 3.2 所示。



图 3.2　exp3-1.cpp 程序的执行结果

实验题 2：实现链栈的各种基本运算的算法

目的：领会链栈的存储结构和掌握链栈中各种基本运算算法的设计。

内容：编写一个程序 listack.cpp，实现链栈（假设栈中的元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-2.cpp 完成以下功能。

（1）初始化栈 s。

（2）判断栈 s 是否非空。

（3）依次进栈元素 a、b、c、d、e。

（4）判断栈 s 是否非空。

（5）输出出栈序列。

（6）判断栈 s 是否非空。

（7）释放栈。

根据《教程》中 3.1.3 节的算法得到 listack.cpp 程序，其中包含如下函数。

- InitStack(LinkStNode *&s)：初始化链栈 s。
- DestroyStack(LinkStNode *&s)：销毁链栈 s。
- StackEmpty(LinkStNode *s)：判断链栈 s 是否为空栈。
- Push(LinkStNode *&s,ElemType e)：元素 e 进链栈。
- Pop(LinkStNode *&s,ElemType &e)：元素 e 出链栈。
- GetTop(LinkStNode *s,ElemType &e)：取链栈的栈顶元素 e。

对应的程序代码如下(设计思路详见代码中的注释)：

```
#include <stdio.h>
#include <malloc.h>
typedef char ElemType;
typedef struct linknode
{   ElemType data;                          //数据域
    struct linknode *next;                  //指针域
} LinkStNode;                               //链栈类型定义
void InitStack(LinkStNode *&s)              //初始化链栈
{   s=(LinkStNode *)malloc(sizeof(LinkStNode));
    s->next=NULL;
}
void DestroyStack(LinkStNode *&s)           //销毁链栈
{   LinkStNode *p=s->next;
    while (p!=NULL)
    {   free(s);
        s=p;
        p=p->next;
    }
    free(s);                                //s 指向尾结点,释放其空间
}
bool StackEmpty(LinkStNode *s)              //判断栈是否为空栈
{
    return(s->next==NULL);
}
void Push(LinkStNode *&s,ElemType e)        //进栈
{   LinkStNode *p;
    p=(LinkStNode *)malloc(sizeof(LinkStNode));
    p->data=e;                              //新建存放元素 e 的结点 p
    p->next=s->next;                        //插入 p 结点作为开始结点
    s->next=p;
}
bool Pop(LinkStNode *&s,ElemType &e)        //出栈
{   LinkStNode *p;
    if (s->next==NULL)                      //栈空的情况
        return false;
    p=s->next;                              //p 指向开始结点
    e=p->data;
    s->next=p->next;                        //删除 p 结点
```

```
    free(p);                                    //释放 p 结点
    return true;
}
bool GetTop(LinkStNode * s,ElemType &e)         //取栈顶元素
{   if (s->next == NULL)                        //栈空的情况
        return false;
    e = s->next->data;
    return true;
}
```

实验程序 exp3-2.cpp 的结构如图 3.3 所示，图中方框表示函数，方框中指出函数名；箭头方向表示函数间的调用关系；虚线方框表示文件的组成，即指出该虚线方框中的函数存放在哪个文件中。

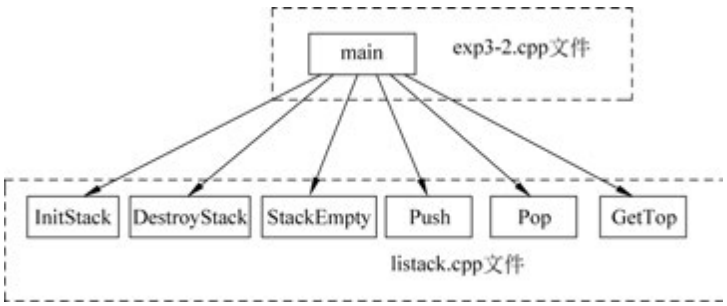


图 3.3　exp3-2.cpp 程序的结构

实验程序 exp3-2.cpp 的代码如下：

```
#include "listack.cpp"                          //包含链栈的基本运算算法
int main()
{   ElemType e;
    LinkStNode * s;
    printf("链栈 s 的基本运算如下:\n");
    printf("  (1)初始化栈 s\n");
    InitStack(s);
    printf("  (2)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (3)依次进栈元素 a,b,c,d,e\n");
    Push(s,'a');
    Push(s,'b');
    Push(s,'c');
    Push(s,'d');
    Push(s,'e');
    printf("  (4)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (5)出栈序列:");
    while (!StackEmpty(s))
    {   Pop(s,e);
        printf("%c ",e);
    }
    printf("\n");
```

```
    printf("  (6)栈为 %s\n",(StackEmpty(s)?"空":"非空"));
    printf("  (7)释放栈\n");
    DestroyStack(s);
    return 0;
}
```

exp3-2.cpp 程序的执行结果如图 3.4 所示。



图 3.4　exp3-2.cpp 程序的执行结果

实验题 3：实现环形队列的各种基本运算的算法

目的：领会环形队列的存储结构和掌握环形队列中各种基本运算算法的设计。

内容：编写一个程序 sqqueue.cpp，实现环形队列(假设栈中的元素类型 ElemType 为 char)的各种基本运算，并在此基础上设计一个程序 exp3-3.cpp 完成以下功能。

(1) 初始化队列 q。

(2) 判断队列 q 是否非空。

(3) 依次进队元素 a、b、c。

(4) 出队一个元素，输出该元素。

(5) 依次进队元素 d、e、f。

(6) 输出出队序列。

(7) 释放队列。

根据《教程》中 3.2.2 节的算法得到 sqqueue.cpp 程序，其中包含如下函数。

- InitQueue(SqQueue *&q)：初始化环形队列 q。
- DestroyQueue(SqQueue *&q)：销毁环形队列 q。
- QueueEmpty(SqQueue *q)：判断环形队列 q 是否为空。
- enQueue(SqQueue *&q,ElemType e)：环形队列进队一个元素 e。
- deQueue(SqQueue *&q,ElemType &e)：环形队列出队一个元素 e。

对应的程序代码如下(设计思路详见代码中的注释)：

```
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100
```

```
typedef char ElemType;
typedef struct
{    ElemType data[MaxSize];
     int front,rear;                          //队首和队尾指针
} SqQueue;                                    //声明环形队列类型
void InitQueue(SqQueue * &q)                  //初始化队列 q
{    q = (SqQueue * )malloc (sizeof(SqQueue));
     q -> front = q -> rear = 0;
}
void DestroyQueue(SqQueue * &q)               //销毁队列 q
{
     free(q);
}
bool QueueEmpty(SqQueue * q)                  //判断队列 q 是否为空
{
     return(q -> front == q -> rear);
}
bool enQueue(SqQueue * &q,ElemType e)         //进队
{    if ((q -> rear + 1) % MaxSize == q -> front)
          return false;                       //队满,上溢出,返回 false
     q -> rear = (q -> rear + 1) % MaxSize;
     q -> data[q -> rear] = e;
     return true;
}
bool deQueue(SqQueue * &q,ElemType &e)        //出队
{    if (q -> front == q -> rear)
          return false;                       //队空,下溢出,返回 false
     q -> front = (q -> front + 1) % MaxSize;
     e = q -> data[q -> front];
     return true;
}
```

实验程序 exp3-3.cpp 的结构如图 3.5 所示,图中方框表示函数,方框中指出函数名;箭头方向表示函数间的调用关系;虚线方框表示文件的组成,即指出该虚线方框中的函数存放在哪个文件中。

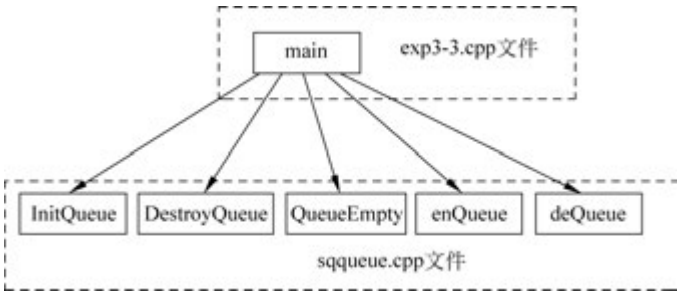


图 3.5　exp3-3.cpp 程序的结构

实验程序 exp3-3.cpp 的代码如下：

```
#include "sqqueue.cpp"                          //包含环形队列的基本运算算法
int main()
{   ElemType e;
    SqQueue *q;
    printf("环形队列基本运算如下:\n");
    printf("  (1)初始化队列 q\n");
    InitQueue(q);
    printf("  (2)依次进队列元素 a,b,c\n");
    if (!enQueue(q,'a')) printf("\t 提示:队满,不能进队\n");
    if (!enQueue(q,'b')) printf("\t 提示:队满,不能进队\n");
    if (!enQueue(q,'c')) printf("\t 提示:队满,不能进队\n");
    printf("  (3)队列为%s\n",(QueueEmpty(q)?"空":"非空"));
    if (deQueue(q,e)==0)
        printf("队空,不能出队\n");
    else
        printf("  (4)出队一个元素%c\n",e);
    printf("  (5)依次进队列元素 d,e,f\n");
    if (!enQueue(q,'d')) printf("\t 提示:队满,不能进队\n");
    if (!enQueue(q,'e')) printf("\t 提示:队满,不能进队\n");
    if (!enQueue(q,'f')) printf("\t 提示:队满,不能进队\n");
    printf("  (6)出队列序列:");
    while (!QueueEmpty(q))
    {   deQueue(q,e);
        printf("%c ",e);
    }
    printf("\n");
    printf("  (7)释放队列\n");
    DestroyQueue(q);
    return 0;
}
```

exp3-3.cpp 程序的执行结果如图 3.6 所示。执行结果表明环形队列 q 中最多只能存放 MaxSize－1(＝4)个元素。

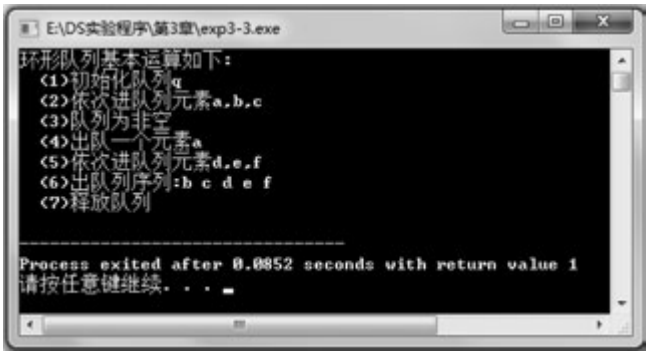


图 3.6　exp3-3.cpp 程序的执行结果

实验题 4：实现链队的各种基本运算的算法

目的：领会链队的存储结构和掌握链队中各种基本运算算法的设计。

内容：编写一个程序 liqueue.cpp，实现链队（假设栈中的元素类型 ElemType 为 char）的各种基本运算，并在此基础上设计一个程序 exp3-4.cpp 完成以下功能。

（1）初始化链队 q。

（2）判断链队 q 是否非空。

（3）依次进队元素 a、b、c。

（4）出队一个元素，输出该元素。

（5）依次进队元素 d、e、f。

（6）输出出队序列。

（7）释放链队。

根据《教程》中 3.2.3 节的算法得到 liqueue.cpp 程序，其中包含如下函数。

- InitQueue(LinkQuNode *&q)：初始化链队 q。
- DestroyQueue(LinkQuNode *&q)：销毁链队 q。
- QueueEmpty(LinkQuNode *q)：判断链队 q 是否为空。
- enQueue(LinkQuNode *&q,ElemType e)：链队进队一个元素 e。
- deQueue(LinkQuNode *&q,ElemType &e)：链队出队一个元素 e。

对应的程序代码如下（设计思路详见代码中的注释）：

```
#include <stdio.h>
#include <malloc.h>
typedef char ElemType;
typedef struct DataNode
{   ElemType data;
    struct DataNode *next;
} DataNode;                                  //声明链队数据结点类型
typedef struct
{   DataNode *front;
    DataNode *rear;
} LinkQuNode;                                //声明链队类型
void InitQueue(LinkQuNode *&q)               //初始化链队 q
{   q=(LinkQuNode *)malloc(sizeof(LinkQuNode));
    q->front=q->rear=NULL;
}
void DestroyQueue(LinkQuNode *&q)            //销毁链队 q
{   DataNode *p=q->front, *r;                //p 指向队头数据结点
    if (p!=NULL)                             //释放数据结点占用的空间
    {   r=p->next;
        while (r!=NULL)
        {   free(p);
            p=r;r=p->next;
        }
    }
    free(p);
```

```
    free(q);                                    //释放链队结点占用的空间
}
bool QueueEmpty(LinkQuNode * q)                 //判断链队 q 是否为空
{
    return(q->rear == NULL);
}
void enQueue(LinkQuNode * &q,ElemType e)        //进队
{   DataNode * p;
    p = (DataNode * )malloc(sizeof(DataNode));
    p->data = e;
    p->next = NULL;
    if (q->rear == NULL)                        //若链队为空,则新结点既是队首结点又是队尾结点
        q->front = q->rear = p;
    else
    {   q->rear->next = p;                      //将 p 结点链到队尾,并将 rear 指向它
        q->rear = p;
    }
}
bool deQueue(LinkQuNode * &q,ElemType &e)       //出队
{   DataNode * t;
    if (q->rear == NULL)                        //链队为空
        return false;
    t = q->front;                               //t 指向第一个数据结点
    if (q->front == q->rear)                    //链队中只有一个结点
        q->front = q->rear = NULL;
    else                                        //链队中有多个结点
        q->front = q->front->next;
    e = t->data;
    free(t);
    return true;
}
```

实验程序 exp3-4.cpp 的结构如图 3.7 所示,图中方框表示函数,方框中指出函数名;箭头方向表示函数间的调用关系;虚线方框表示文件的组成,即指出该虚线方框中的函数存放在哪个文件中。

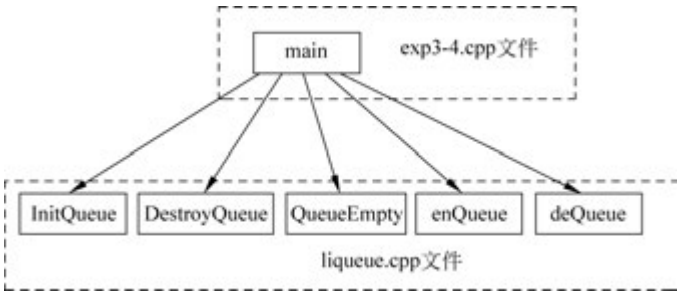


图 3.7　exp3-4.cpp 程序的结构

实验程序 exp3-4.cpp 的代码如下：

```
#include "liqueue.cpp"                          //包含链队的基本运算算法
int main()
{   ElemType e;
    LinkQuNode *q;
    printf("链队的基本运算如下:\n");
    printf("  (1)初始化链队 q\n");
    InitQueue(q);
    printf("  (2)依次进链队元素 a,b,c\n");
    enQueue(q,'a');
    enQueue(q,'b');
    enQueue(q,'c');
    printf("  (3)链队为 %s\n",(QueueEmpty(q)?"空":"非空"));
    if (deQueue(q,e) == 0)
        printf("\t 提示:队空,不能出队\n");
    else
        printf("  (4)出队一个元素 %c\n",e);
    printf("  (5)依次进链队元素 d,e,f\n");
    enQueue(q,'d');
    enQueue(q,'e');
    enQueue(q,'f');
    printf("  (6)出链队序列:");
    while (!QueueEmpty(q))
    {   deQueue(q,e);
        printf("%c ",e);
    }
    printf("\n");
    printf("  (7)释放链队\n");
    DestroyQueue(q);
    return 0;
}
```

exp3-4.cpp 程序的执行结果如图 3.8 所示。



图 3.8　exp3-4.cpp 程序的执行结果

3.2 设计性实验

实验题 5：用栈求解迷宫问题的所有路径及最短路径

目的：掌握栈在求解迷宫问题中的应用。

内容：编写一个程序 exp3-5.cpp，改进《教程》3.1.4 节中的求解迷宫问题程序，要求输出如图 3.9 所示的迷宫的所有路径，并求第一条最短路径及其长度。

在本实验中用 mg 作为迷宫数组，用 St 数组作为顺序栈，Path 数组保存一条迷宫路径，将它们都设置为全局变量。设计的功能算法如下。

- mgpath(int xi,int yi,int xe,int ye)：求解迷宫问题，即输出从入口(xi,yi)到出口(xe,ye)的全部路径和最短路径(包含最短路径长度)。与《教程》3.1.4 节中的求解迷宫问题程序相比，其改进的方法是当找到一条路径时不使用 return 语句退出，而是出栈一次，重新回溯走另一条路径并输出，并用 minlen 记录最短路径长度，用 Path 数组记录最短路径。
- dispapath()：输出一条路径并求最短路径。
- dispminpath()：输出第一条最短路径及其长度。

实验程序 exp3-5.cpp 的结构如图 3.10 所示，图中方框表示函数，方框中指出函数名；箭头方向表示函数间的调用关系；虚线方框表示文件的组成，即指出该虚线方框中的函数存放在哪个文件中。

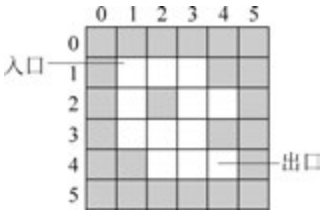


图 3.9 迷宫示意图

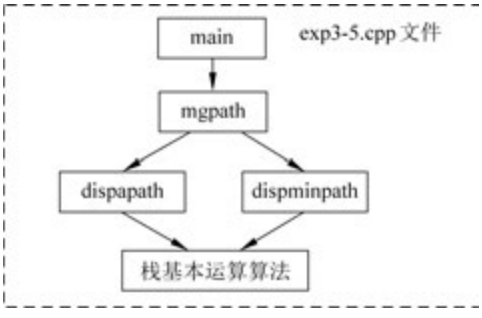


图 3.10 exp3-5.cpp 程序结构

实验程序 exp3-5.cpp 的代码如下(设计思路详见代码中的注释)：

```
#include<stdio.h>
#include<malloc.h>
#define M 4                                  //行数(不计外围的两行)
#define N 4                                  //列数(不计外围的两列)
#define MaxSize 100                          //栈最多元素个数
int mg[M+2][N+2]={                           //一个迷宫,其四周要加上均为1的外框
    {1,1,1,1,1,1},{1,0,0,0,1,1},{1,0,1,0,0,1},
    {1,0,0,0,1,1},{1,1,0,0,0,1},{1,1,1,1,1,1}
```

```
};
/********** 求解迷宫问题中的栈 - 开始 **********/
typedef struct
{    int i,j;
     int di;
} ElemType;                                    //声明栈元素类型
ElemType shortestpath[MaxSize];                //定义存放最短路径的数组
int minlen = MaxSize;                          //最短路径长度
int count = 1;                                 //路径数计数
typedef struct
{    ElemType data[MaxSize];
     int top;                                  //栈顶指针
} SqStack;                                     //声明顺序栈类型
void InitStack(SqStack * &s)                   //初始化顺序栈
{    s = (SqStack * )malloc(sizeof(SqStack));
     s -> top = -1;
}
void DestroyStack(SqStack * &s)                //销毁顺序栈
{
     free(s);
}
bool StackEmpty(SqStack * s)                   //判断栈空否
{
     return(s -> top == -1);
}
bool Push(SqStack * &s,ElemType e)             //进栈
{    if (s -> top == MaxSize - 1)              //栈满的情况,即栈上溢出
          return false;
     s -> top++;
     s -> data[s -> top] = e;
     return true;
}
bool Pop(SqStack * &s,ElemType &e)             //出栈
{    if (s -> top == -1)                       //栈为空的情况,即栈下溢出
          return false;
     e = s -> data[s -> top];
     s -> top -- ;
     return true;
}
bool GetTop(SqStack * s,ElemType &e)           //取栈顶元素
{    if (s -> top == -1)                       //栈为空的情况,即栈下溢出
          return false;
     e = s -> data[s -> top];
     return true;
}
/********** 求解迷宫问题中的栈 - 结束 **********/
void dispapath(SqStack * st)                   //输出一条路径并求最短路径
{    int k;
     printf("%5d: ",count++);                  //输出第 count 条路径
     for (k = 0;k <= st -> top;k++)
```

```
            printf("(%d,%d) ",st->data[k].i,st->data[k].j);
        printf("\n");
        if (st->top+1<minlen)                          //比较找最短路径
        {   for (k=0;k<=st->top;k++)                   //将最短路径存放在 path 中
                shortestpath[k]=st->data[k];
            minlen=st->top+1;                          //将最短路径长度存放在 minlen 中
        }
    }
    void dispminpath()                                 //输出第一条最短路径
    {   printf("最短路径如下:\n");
        printf("长度: %d\n",minlen);
        printf("路径: ");
        for (int k=0;k<minlen;k++)
            printf("(%d,%d) ",shortestpath[k].i,shortestpath[k].j);
        printf("\n");
    }
    void mgpath(int xi,int yi,int xe,int ye)           //求迷宫路径
    {   int i,j,i1,j1,di;
        bool find;
        ElemType e,e1;
        SqStack *st;                                   //定义一个栈 st
        InitStack(st);                                 //初始化 st
        e.i=xi;e.j=yi;e.di=-1;
        Push(st,e);                                    //入口进栈
        mg[xi][yi]=-1;                                 //修改入口的 mg 值
        while (!StackEmpty(st))                        //栈不空时循环
        {   GetTop(st,e);                              //取栈顶方块 e
            i=e.i;j=e.j;di=e.di;
            if (i==xe && j==ye)                        //找到了出口
            {   dispapath(st);                         //输出一条路径
                mg[i][j]=0;                            //让出口变为其他路径可走方块
                Pop(st,e);                             //出口退栈,即回退一个方块
            }
            else
            {   find=false;                            //找下一个可走方块(i,1j1)
                while (di<3 && !find)
                {   di++;                              //转向下一个方位
                    switch(di)
                    {   case 0:i1=i-1; j1=j; break;
                        case 1:i1=i; j1=j+1;break;
                        case 2:i1=i+1; j1=j;break;
                        case 3:i1=i, j1=j-1; break;
                    }
                    if (mg[i1][j1]==0) find=true;
                }
                if (find)                              //找到了下一个可走方块(i1,j1)
                {   st->data[st->top].di=di;           //修改栈顶方块的 di 值
                    e1.i=i1;e1.j=j1;e1.di=-1;
                    Push(st,e1);                       //下一个可走方块(i1,j1)进栈
                    mg[i1][j1]=-1;                     //避免重复走到该方块
```

```
            }
            else                                    //没有路径可走,则退栈(i,j)方块
            {   mg[i][j] = 0;                       //让该位置变为其他路径可走方块
                Pop(st,e);                          //将无路可走的方块出栈
            }
        }
    }
    DestroyStack(st);
    dispminpath();                                  //输出最短路径
}
int main()
{   int xi = 1,yi = 1,xe = M,ye = N;
    printf("迷宫中从(%d,%d)到(%d,%d)的全部路径如下:\n",xi,yi,xe,ye);
    mgpath(xi,yi,xe,ye);
    return 0;
}
```

exp3-5.cpp 程序的执行结果如图 3.11 所示,该迷宫从入口(1,1)到出口(4,4)共有 4 条路径,第一条最短路径长度为 7。

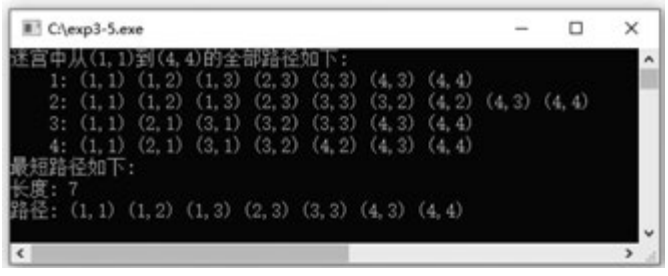


图 3.11 exp3-5.cpp 程序的执行结果

实验题 6:编写病人看病模拟程序

目的:掌握队列应用的算法设计。

内容:编写一个程序 exp3-6.cpp,反映病人到医院排队看病的情况。在病人排队过程中主要重复下面两件事。

(1) 病人到达诊室,将病历本交给护士,排到等待队列中候诊。

(2) 护士从等待队列中取出下一位病人的病历,该病人进入诊室就诊。

要求模拟病人等待就诊这一过程。程序采用菜单方式,其选项及功能说明如下。

1:排队——输入排队病人的病历号,加入病人排队队列中。

2:就诊——病人排队队列中最前面的病人就诊,并将其从队列中删除。

3:查看排队——从队首到队尾列出所有排队病人的病历号。

4:不再排队,余下依次就诊——从队首到队尾列出所有排队病人的病历号,并退出运行。

5:下班——退出运行。

本实验采用一个链队 qu 存放病人排队情况,链队的结构和实现参见《教程》中

3.2.3 节。除链队外，设计的功能算法如下。

- dispQueue(qu)：从队头开始输出所有病人编号。
- SeeDoctor()：模拟病人看病的过程。

实验程序 exp3-6.cpp 的结构如图 3.12 所示，图中方框表示函数，方框中指出函数名，箭头方向表示函数间的调用关系，虚线方框表示文件的组成，即指出该虚线方框中的函数存放在哪个文件中。

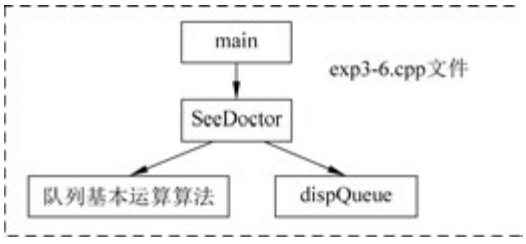


图 3.12　exp3-6.cpp 程序的结构

实验程序 exp3-6.cpp 的代码如下(设计思路详见代码中的注释)：

```
#include<stdio.h>
#include<malloc.h>
/********** 求解病人看病队列 - 开始 **********/
typedef int ElemType;                          //病人编号为整数
typedef struct DataNode
{   ElemType data;
    struct DataNode *next;
} DataNode;                                    //链队数据结点类型
typedef struct
{   DataNode *front;
    DataNode *rear;
} LinkQuNode;                                  //链队类型
void InitQueue(LinkQuNode *&q)                 //初始化
{   q=(LinkQuNode *)malloc(sizeof(LinkQuNode));
    q->front=q->rear=NULL;
}
void DestroyQueue(LinkQuNode *&q)              //销毁队列
{   DataNode *p=q->front,*r;                   //p指向队头数据结点
    if (p!=NULL)                               //释放数据结点占用空间
    {   r=p->next;
        while (r!=NULL)
        {   free(p);
            p=r;r=p->next;
        }
    }
    free(p);
    free(q);                                   //释放链队结点占用空间
}
bool QueueEmpty(LinkQuNode *q)                 //判定队列是否为空
{
```

```
    return q->rear == NULL;
}
void enQueue(LinkQuNode *&q,ElemType e)                    //进队 e
{   DataNode *p;
    p=(DataNode *)malloc(sizeof(DataNode));
    p->data=e;
    p->next=NULL;
    if (q->rear==NULL)                                     //若链队为空,则新结点作为首、尾结点
        q->front=q->rear=p;
    else
    {   q->rear->next=p;                                   //将 p 结点链到队尾,并将 rear 指向它
        q->rear=p;
    }
}
bool deQueue(LinkQuNode *&q,ElemType &e)                   //出队 e
{   DataNode *t;
    if (q->rear==NULL)                                     //队列为空
        return false;
    t=q->front;                                            //t 指向首结点
    if (q->front==q->rear)                                 //队列中只有一个结点时
        q->front=q->rear=NULL;
    else                                                   //队列中有多个结点时
        q->front=q->front->next;
    e=t->data;
    free(t);
    return true;
}
/********** 求解病人看病队列 - 结束 ************/
void dispQueue(LinkQuNode *q)                              //遍历全部队列元素
{   DataNode *t=q->front;                                  //t 指向队头结点
    while(t!=NULL)                                         //遍历所有数据结点
    {   printf("%d ",t->data);
        t=t->next;
    }
    printf("\n");
}
void SeeDoctor()                                           //模拟病人看病的过程
{   int sel,no;
    ElemType e;
    bool flag=true;
    LinkQuNode *qu;                                        //定义病人队列 qu
    InitQueue(qu);                                         //初始化 qu
    while(flag)                                            //循环执行
    {   printf("> 1:排队 2:就诊 3:查看排队 4.不再排队,余下依次就诊 5:下班 请选择:");
        scanf("%d",&sel);
        switch(sel)
        {   case 1:printf("输入病历号:");
                   scanf("%d",&no);
                   enQueue(qu,no);
                   break;
```

```
            case 2:if(QueueEmpty(qu))                    //队空
                       printf(" 没有排队的病人!\n");
                   else                                  //队不空
                   {   deQueue(qu,e);
                       printf(" >>病人 %d 就诊\n",e);
                   }
                   break;
            case 3:if(QueueEmpty(qu))                    //队空
                       printf(" 没有排列的病人!\n");
                   else                                  //队不空
                   {   printf(" >>排队病人:");
                       dispQueue(qu);
                   }
                   break;
            case 4:if(QueueEmpty(qu))                    //队空
                       printf(" >>没有排列的病人!\n");
                   else                                  //队不空
                   {   printf(" >>病人按以下顺序就诊:");
                       while(!QueueEmpty(qu))
                       {   deQueue(qu,e);
                           printf("%d ",e);
                       }
                       printf("\n");
                   }
                   DestroyQueue(qu);                     //释放链队
                   flag=false;                           //退出
                   break;
            case 5:if(!QueueEmpty(qu))                   //队不空
                       printf(" 请排队的病人明天就医!\n");
                   flag=false;                           //退出
                   DestroyQueue(qu);                     //释放链队
                   break;
        }
    }
}
int main()
{   SeeDoctor();
    return 0;
}
```

exp3-6.cpp 程序的一次执行过程如图 3.13 所示。首先病人 1、2 排队，病人 1 就诊，接着病人 3、4 参加排队，通过查看队列发现排队病人是 2、3、4，然后病人 2 就诊，最后病人 3、4 依次就诊。

实验题 7：求解栈元素排序问题

目的：掌握栈应用的算法设计。

内容：编写一个程序 exp3-7.cpp，按升序对一个字符栈进行排序，即最小元素位于栈顶，注意最多只能使用一个额外的栈存放临时数据，并输出栈排序的过程。

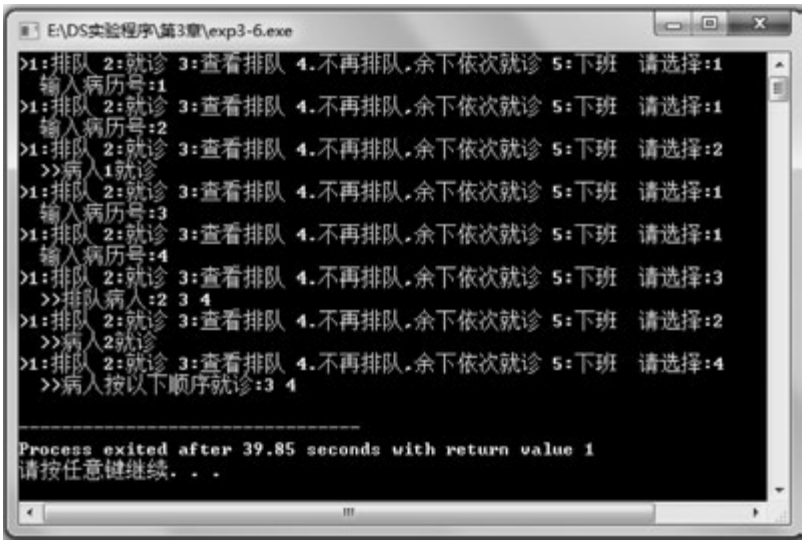


图 3.13 exp3-6.cpp 程序的一次执行过程

本实验中采用一个额外的栈 tmpst 存放临时数据。设计的功能算法为 StackSort(SqStack *&st),即对栈 st 中的元素排序。其思路是处理栈 st 的某个栈顶元素 e,出栈元素 e,将其存放在 tmpst 中。若临时栈 tmpst 为空,直接将 e 进入栈 tmpst;若栈 tmpst 不为空,将它的元素退栈(放入栈 st 中)直到 tmpst 栈顶元素小于 e,再将 tmp 进入栈 tmpst 中,如图 3.14 所示为处理栈 st 的栈顶元素 3 的过程。进行这样的过程直到 st 为空,而 tmpst 中的元素从栈底到栈顶是递增的。再将 tmpst 中的所有元素退栈并进到栈 st 中。

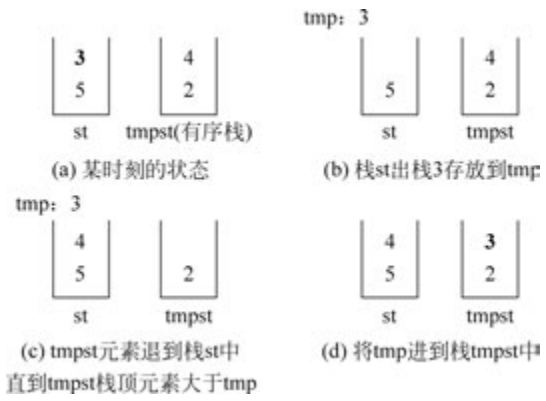


图 3.14 处理 st 的栈顶元素 3 的过程

实验程序 exp3-7.cpp 的结构如图 3.15 所示,图中方框表示函数,方框中指出函数名;箭头方向表示函数间的调用关系;虚线方框表示文件的组成,即指出该虚线方框中的函数存放在哪个文件中。

实验程序 exp3-7.cpp 的代码如下(设计思路详见代码中的注释):

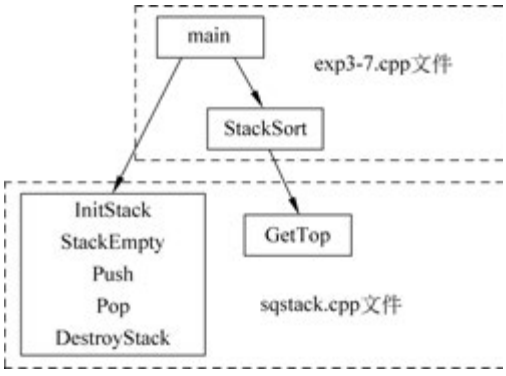


图 3.15　exp3-7.cpp 程序的结构

```
#include "sqstack.cpp"                          //包含顺序栈的基本运算算法
void StackSort(SqStack *&st)                    //对栈 st 中的元素排序
{   SqStack *tmpst;
    InitStack(tmpst);
    ElemType e,e1;
    while (!StackEmpty(st))                     //栈 st 不为空时循环
    {   Pop(st,e);                              //出栈元素 e
        printf("    st:出栈 %c => ",e);
        while(!StackEmpty(tmpst))
        {   GetTop(tmpst,e1);
            printf("tmpst:取栈顶元素 %c ",e1);
            if (e1 > e)
            {   printf("因 %c > %c ",e1,e);
                printf("tmpst:退栈 %c ",e1);
                Pop(tmpst,e1);
                printf("s:进栈 %c ",e1);
                Push(st,e1);
            }
            else
            {   printf("因 %c < %c,退出循环 ",e1,e);
                break;
            }
        }
        Push(tmpst,e);
        printf("tmpst:进栈 %c\n",e);
    }
    while (!StackEmpty(tmpst))
    {   Pop(tmpst,e);
        Push(st,e);
    }
    DestroyStack(tmpst);
}
int main()
{   ElemType e;
    SqStack *s;
```

```
    InitStack(s);
    printf("(1)依次进栈元素 1,3,4,2\n");
    Push(s,'1');
    Push(s,'3');
    Push(s,'4');
    Push(s,'2');
    printf("(2)栈 s 排序过程:\n");
    StackSort(s);
    printf("(3)栈 s 排序完毕\n");
    printf("(4)s 的出栈序列:");
    while (!StackEmpty(s))
    {   Pop(s,e);
        printf("%c ",e);
    }
    printf("\n");
    DestroyStack(s);
    return 0;
}
```

exp3-7.cpp 程序的执行过程如图 3.16 所示。

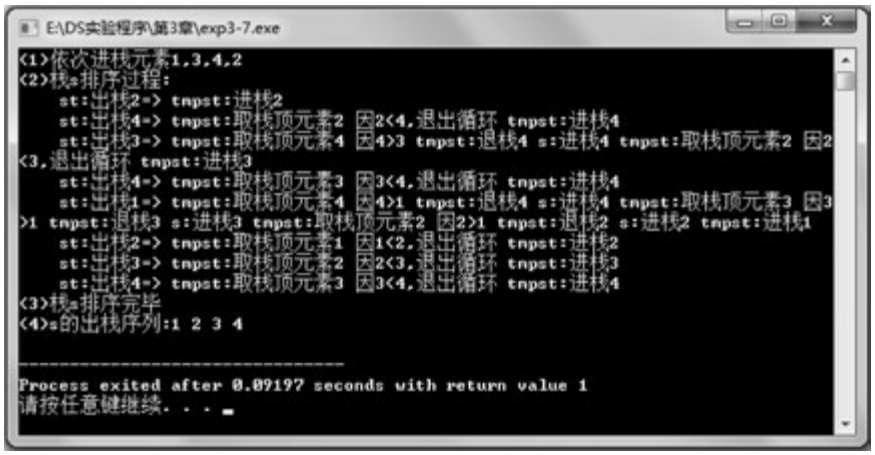


图 3.16　exp3-7.cpp 程序的执行过程

3.3 综合性实验

实验题 8：用栈求解 n 皇后问题

目的：深入掌握栈应用的算法设计。

内容：编写一个程序 exp3-8.cpp 求解 n 皇后问题，即在 $n \times n$ 的方格棋盘上放置 n 个皇后，要求每个皇后不同行、不同列、不同左右对角线。如图 3.17 所示为八皇后问题的一个解。在本实验中，皇后个数 n 由用户输入，其值不能超过 20，输出所有的解；采用类似于用

栈求解迷宫问题的方法。

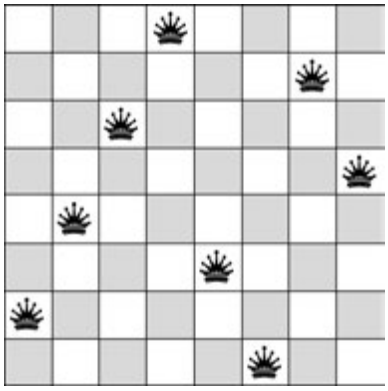


图 3.17　八皇后问题

用 col[1..n]数组存放 n 个皇后的位置，其中 col[i]表示第 i($1 \leqslant i \leqslant n$)个皇后的列号，即第 i 个皇后的位置是(i,col[i])。例如图 3.17 所示的八皇后问题的解表示为 col[1..8]={4,7,3,8,2,5,1,6}。本实验中使用一个顺序栈 St(为了简便，栈运算直接通过操作其成员实现)，其类型如下：

```
typedef struct
{    int col[MaxSize];                    //col[i]存放第 i 个皇后的列号
     int top;                             //栈顶指针
} StackType;                              //声明顺序栈类型
```

如同用栈求解迷宫问题，找到出口后栈中所有方块构成一条迷宫路径一样，这里栈 St 中保存当前已经放好的皇后的位置，若其中的皇后个数为 n，表示得到一个解。设计的功能算法如下。

- dispasolution(StackType St)：输出一个皇后问题解，即 n 个皇后的位置是(1,St.col[1]),(2,St.col[2]),…,(n,St.col[n])。
- place(StackType St,int k,int j)：对于第 k 个皇后，测试(k,j)(j 从第 1 列开始)是否与栈中已放置好的第 1～k－1 个皇后有冲突，St.col[k]用于存放第 k 个皇后的列号，即该皇后的位置为(k,St.col[k])。对于位置(i,j)($1 \leqslant i \leqslant k-1$)上的皇后，是否与位置($k$,St.col[$k$])有冲突呢？不同行是显然的，若同列则有 St.col[k]==j。再考虑两条对角线冲突，如图 3.18 所示，若它们在任意一条对角线上，则构成一个等边直角三角形，即|j－St.col[k]|==|k－i|。所以，只要满足以下条件就表示存在冲突，否则表示不存在冲突：

  (St.col[k]==j) ‖ (abs(j－St.col[k])==abs(k－i))

- queen(int n)：用于求解 n 皇后问题，并输出所有解。

实验程序 exp3-8.cpp 的结构如图 3.19 所示，图中方框表示函数，方框中指出函数名；箭头方向表示函数间的调用关系；虚线方框表示文件的组成，即指出该虚线方框中的函数存放在哪个文件中。

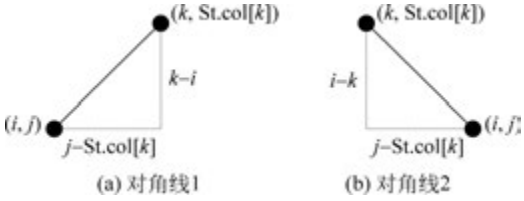


图 3.18 两个皇后构成对角线的情况

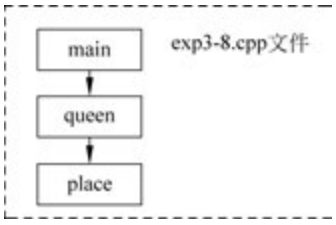


图 3.19 exp3-8.cpp 程序的结构

实验程序 exp3-8.cpp 的代码如下(设计思路详见代码中的注释):

```
#include <stdio.h>
#include <stdlib.h>
#define MaxSize 100
typedef struct
{   int col[MaxSize];                       //col[i]存放第 i 个皇后的列号
    int top;                                //栈顶指针
} StackType;                                //声明顺序栈类型
void dispasolution(StackType St)            //输出一个解
{   static int count = 0;                   //静态变量用于统计解的个数
    printf("  第 %d 个解:",++count);
    for (int i = 1;i <= St.top;i++)
        printf("(%d,%d) ",i,St.col[i]);
    printf("\n");
}
bool place(StackType St,int k,int j)        //测试(k,j)是否与第 1～k-1 个皇后有冲突
{   int i = 1;
    if (k == 1) return true;                //放第一个皇后时没有冲突
    while (i <= k-1)                        //测试与前面已放置的皇后是否有冲突
    {   if ((St.col[i] == j) || (abs(j-St.col[i]) == abs(i-k)))
            return false;                   //有冲突时返回 false
        i++;
    }
    return true;                            //没有冲突时返回 true
}
void queen(int n)                           //求解 n 皇后问题
{   int k;
    bool find;
    StackType St;                           //定义栈 st
    St.top = 0;                             //初始化栈顶指针
    St.top++; St.col[St.top] = 0;           //col[1] = 0,表示从第 1 个皇后开始,初始列号为 0
    while (St.top!= 0)                      //栈不空时循环
    {   k = St.top;                         //试探栈顶的第 k 个皇后
        find = false;                       //尚未找到第 k 个皇后的位置,find 设置为 false
        for (int j = St.col[k] + 1;j <= n;j++)    //为第 k 个皇后找一个合适的列号
            if (place(St,k,j))              //在第 k 行找到一个放皇后的位置(k,j)
            {   St.col[St.top] = j;         //修改第 k 个皇后的位置(新列号)
                find = true;                //找到第 k 个皇后的位置,find 设置为 true
                break;                      //找到后退出 for 循环
```

```
            }
        if (find)                          //在第 k 行找到一个放皇后的位置(k,j)
        {   if (k == n)                    //若所有皇后均放好,输出一个解
                dispasolution(St);
            else                           //还有皇后未放时将第 k+1 个皇后进栈
            {   St.top++;
                St.col[St.top] = 0;        //新进栈的皇后从第 0 列开始试探
            }
        }
        else                               //若第 k 个皇后没有合适位置,回溯
            St.top--;                      //即将第 k 个皇后退栈
    }
}
int main()
{   int n;                                 //n 存放实际的皇后个数
    printf("皇后问题(n<20) n=");
    scanf("%d",&n);
    if (n>20)
        printf("n 值太大\n");
    else
    {   printf(" %d 皇后问题求解如下:\n",n);
        queen(n);
    }
    return 0;
}
```

exp3-8.cpp 程序的一次执行结果如图 3.20 所示,表示六皇后问题有 4 种放置方式,如图 3.21 所示。

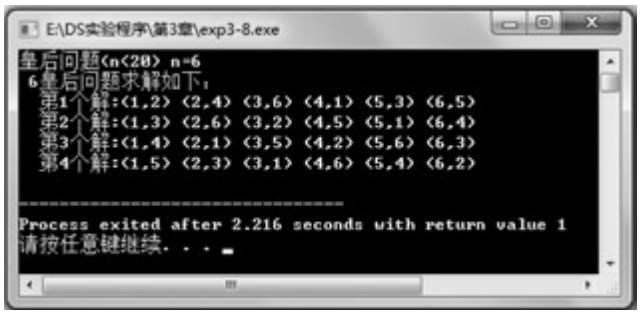


图 3.20　exp3-8.cpp 程序的一次执行结果

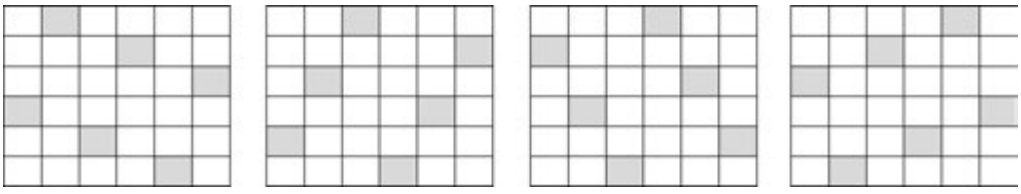


图 3.21　六皇后问题的 4 种放置方式

实验题 9：编写停车场管理程序

目的：深入掌握栈和队列应用的算法设计。

内容：编写满足以下要求的停车场管理程序 exp3-9.cpp。设停车场内只有一个可停放 n 辆汽车的狭长通道，且只有一个大门可供汽车进出。

汽车在停车场内按车辆到达时间的先后顺序依次由南向北排列(大门在最北端，最先到达的第一辆车停放在停车场的最南端)，若停车场内已停满 n 辆车，则后来的汽车只能在门外的便道(即候车场)上等候，一旦有车开走，则排在便道上的第一辆车即可开入；当停车场内的某辆车要离开时，在它之后进入的车辆必须先退出停车场为它让路，待该辆车开出大门外，其他车辆再按原次序进入停车场，每辆停放在停车场的车在它离开停车场时必须按停留的时间长短交纳费用。整个停车场的示意图如图 3.22 所示。

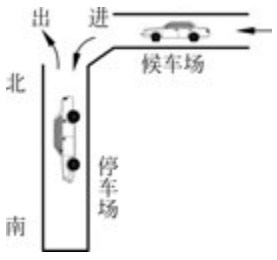


图 3.22 停车场示意图

用栈模拟停车场，用队列模拟停车场外的便道，按照从键盘获取的数据序列进行模拟管理。每一组输入数据包括 3 个数据项：汽车到达或者离开、汽车牌照号码以及到达或离开的时刻。对每一组输入数据进行操作后的输出信息为：若是汽车到达，则输出汽车在停车场内或便道上的停车位置；若是汽车离开，则输出汽车在停车场内停留的时间和应交纳的费用(在便道上停留的时间不收费)。这里栈采用顺序存储结构，队列采用环形队列。

另外还需设一个临时栈，用于临时停放为要给离开的汽车让路而从停车场退出来的汽车，也用顺序结构实现。

用户输入的命令有以下 5 种。

(1) 汽车到达。

(2) 汽车离开。

(3) 输出停车场中的所有汽车牌号。

(4) 输出候车场中的所有汽车牌号。

(5) 退出系统运行。

其中，栈 s 和队列 q 分别采用顺序栈和环形队列。设计的功能算法如下。

- InitStack(SqStack *&s)：初始化栈 s。
- StackEmpty(SqStack *s)：判断栈 s 是否为空栈。
- StackFull(SqStack *s)：判断栈 s 是否为满栈。
- Push(SqStack *&s,ElemType e)：元素 e 进栈。
- Pop(SqStack *&s,ElemType &e)：元素 e 出栈。
- DispStack(SqStack *s)：从栈顶到栈底输出元素。
- InitQueue(SqQueue *&q)：初始化队列 q。
- QueueEmpty(SqQueue *q)：判断队列 q 是否为空。
- QueueFull(SqQueue *q)：判断队列 q 是否为满。
- enQueue(SqQueue *&q,ElemType e)：元素 e 进队。
- deQueue(SqQueue *&q,ElemType &e)：元素 e 出队。

实验程序 exp3-9.cpp 的结构如图 3.23 所示，图中方框表示函数，方框中指出函数名；箭头方向表示函数间的调用关系；虚线方框表示文件的组成，即指出该虚线方框中的函数存放在哪个文件中。

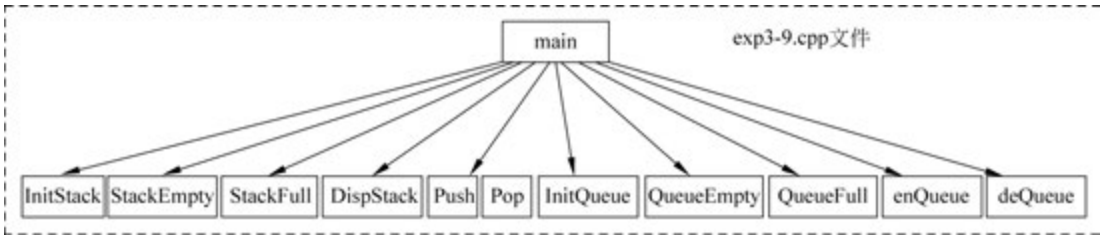


图 3.23　exp3-9.cpp 程序的结构

实验程序 exp3-9.cpp 的代码如下(设计思路详见代码中的注释)：

```
#include <stdio.h>
#include <malloc.h>
#define N 3                                    //停车场内最多的停车数
#define M 4                                    //候车场内最多的候车数
#define Price 2                                //每单位停车费用
typedef struct
{   int CarNo[N];                              //车牌号
    int CarTime[N];                            //进场时间
    int top;                                   //栈指针
} SqStack;                                     //声明顺序栈类型
typedef struct
{   int CarNo[M];                              //车牌号
    int front,rear;                            //队首和队尾指针
} SqQueue;                                     //声明环形队列类型
//以下为栈的运算算法
void InitStack(SqStack *&s)                    //初始化栈
{   s=(SqStack *)malloc(sizeof(SqStack));
    s->top=-1;
}
bool StackEmpty(SqStack *s)                    //判断栈是否为空
{
    return(s->top==-1);
}
bool StackFull(SqStack *s)                     //判断栈是否为满
{
    return(s->top==N-1);
}
bool Push(SqStack *&s,int e1,int e2)           //进栈
{   if (s->top==N-1)
        return false;
    s->top++;
    s->CarNo[s->top]=e1;
    s->CarTime[s->top]=e2;
    return true;
}
bool Pop(SqStack *&s,int &e1,int &e2)          //出栈
```

```
{   if (s->top == -1)
        return false;
    e1 = s->CarNo[s->top];
    e2 = s->CarTime[s->top];
    s->top--;
    return true;
}
void DispStack(SqStack *s)                    //显示栈中元素
{   for (int i = s->top;i >= 0;i--)
        printf("%d ",s->CarNo[i]);
    printf("\n");
}
//以下为队列的运算算法
void InitQueue(SqQueue *&q)                   //初始化队
{   q = (SqQueue *)malloc (sizeof(SqQueue));
    q->front = q->rear = 0;
}
bool QueueEmpty(SqQueue *q)                   //判断队是否为空
{
    return(q->front == q->rear);
}
bool QueueFull(SqQueue *q)                    //判断队是否为满
{
    return ((q->rear + 1) % M == q->front);
}
bool enQueue(SqQueue *&q,int e)               //进队
{   if ((q->rear + 1) % M == q->front)        //队为满的情况
        return false;
    q->rear = (q->rear + 1) % M;
    q->CarNo[q->rear] = e;
    return true;
}
bool deQueue(SqQueue *&q,int &e)              //出队
{   if (q->front == q->rear)                  //队为空的情况
        return false;
    q->front = (q->front + 1) % M;
    e = q->CarNo[q->front];
    return true;
}
void DispQueue(SqQueue *q)                    //显示队中元素
{   int i = (q->front + 1) % M;               //i初始化为队头元素下标
    printf("%d ",q->CarNo[i]);                //扫描队中全部元素
    while ((q->rear - i + M) % M > 0)         //i循环增1
    {   i = (i + 1) % M;
        printf("%d ",q->CarNo[i]);
    }
    printf("\n");
}
int main()
{   int comm,i,j;
    int no,e1,time,e2;
    SqStack *St,*St1;
    SqQueue *Qu;
```

```
    InitStack(St);
    InitStack(St1);
    InitQueue(Qu);
    do
    {   printf(">输入指令(1:到达 2:离开 3:停车场 4:候车场 0:退出):");
        scanf("%d",&comm);
        switch(comm)
        {
        case 1:                                     //汽车到达
            printf("  车号 到达时间:");
            scanf("%d%d",&no,&time);
            if (!StackFull(St))                     //停车场不满
            {   Push(St,no,time);
                printf("  停车场位置:%d\n",St->top+1);
            }
            else                                    //停车场已满
            {   if (!QueueFull(Qu))                 //候车场不满
                {   enQueue(Qu,no);
                    printf("  候车场位置:%d\n",Qu->rear);
                }
                else printf("  候车场已满,不能停车\n");
            }
            break;
        case 2:                                     //汽车离开
            printf("  车号 离开时间:");
            scanf("%d%d",&no,&time);
            for (i=0;i<=St->top && St->CarNo[i]!=no;i++);
            if (i>St->top)
                printf("  未找到该编号的汽车\n");
            else
            {   for (j=i;j<=St->top;j++)
                {   Pop(St,e1,e2);
                    Push(St1,e1,e2);                //倒车到临时栈 St1 中
                }
                Pop(St,e1,e2);                      //该汽车离开
                printf("  %d 汽车停车费用:%d\n",no,(time-e2)*Price);
                while (!StackEmpty(St1))            //将临时栈 St1 中的汽车重新回到 St 中
                {   Pop(St1,e1,e2);
                    Push(St,e1,e2);
                }
                if (!QueueEmpty(Qu))                //队不为空时将队头进栈 St
                {   deQueue(Qu,e1);
                    Push(St,e1,time);               //以当前时间开始计费
                }
            }
            break;
        case 3:                                     //显示停车场情况
            if (!StackEmpty(St))
            {   printf("  停车场中的车辆:");       //输出停车场中的车辆
                DispStack(St);
            }
            else printf("  停车场中无车辆\n");
            break;
```

```
        case 4:                                          //显示候车场情况
            if (!QueueEmpty(Qu))
            {   printf("  候车场中的车辆:");             //输出候车场中的车辆
                DispQueue(Qu);
            }
            else printf("  候车场中无车辆\n");
            break;
        case 0:                                          //结束
            if (!StackEmpty(St))
            {   printf("  停车场中的车辆:");             //输出停车场中的车辆
                DispStack(St);
            }
            if (!QueueEmpty(Qu))
            {   printf("  候车场中的车辆:");             //输出候车场中的车辆
                DispQueue(Qu);
            }
            break;
        default:                                         //其他情况
            printf("  输入的命令错误\n");
            break;
        }
    } while(comm!= 0);
    return 0;
}
```

exp3-9.cpp 程序的一次执行结果如图 3.24 所示。首先有 4 辆车依次进入，由于停车场内最多只能停放 3 辆车，所以有一辆车停放在候车场中。然后 101 车离开，共停车 5 小时，收费 10 元，候车场中的车辆进入停车场，此时看到停车场中有 3 辆车。



图 3.24　exp3-9.cpp 程序的一次执行结果