

第5章

递归



5.1

验证性实验

**实验题 1: 采用递归和非递归方法求解 Hanoi 问题**

目的: 领会基本递归算法设计和递归到非递归的转换方法。

内容: 编写程序 exp5-1.cpp, 采用递归和非递归方法求解 Hanoi 问题, 输出 3 个盘片的移动过程。

根据《教程》第 5 章的原理设计本实验的功能算法如下。

- Hanoi1(int n, char a, char b, char c): 求解 Hanoi 问题的递归算法。
- Hanoi2(int n, char x, char y, char z): 求解 Hanoi 问题的非递归算法。其原理参见《教程》第 5 章 5.2.3 小节。
- 栈的基本运算算法: 用于 Hanoi2 算法中。

实验程序 exp5-1.cpp 的结构如图 5.1 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。

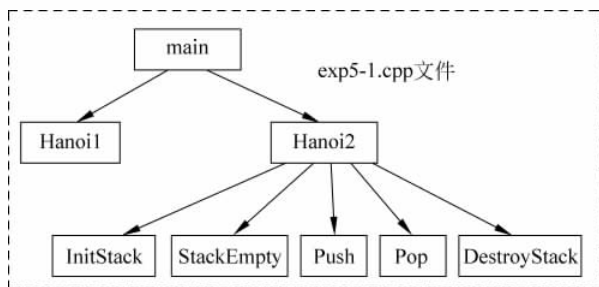


图 5.1 exp5-1.cpp 程序结构

实验程序 exp5-1.cpp 的程序代码如下:

```
#include <stdio.h>
#include <malloc.h>
#define MaxSize 100
// ---- 递归算法 ----
void Hanoi1(int n, char a, char b, char c)
{
    if (n == 1)
        printf("\t将第 %d 个盘片从 %c 移动到 %c\n", n, a, c);
    else
    {
        Hanoi1(n-1, a, c, b);
        printf("\t将第 %d 个盘片从 %c 移动到 %c\n", n, a, c);
        Hanoi1(n-1, b, a, c);
    }
}
// ---- 非递归算法 ----
```

```

typedef struct
{
    int n; // 盘片个数
    char x, y, z; // 3 个塔座
    bool flag; // 可直接移动盘片时为 true, 否则为 false
} ElemType; // 顺序栈中元素的类型

typedef struct
{
    ElemType data[MaxSize]; // 存放元素
    int top; // 栈顶指针
} StackType; // 声明顺序栈类型

// -- 求解 Hanoi 问题对应顺序栈的基本运算算法 -----
void InitStack(StackType * &s) // 初始化栈
{
    s = (StackType *) malloc(sizeof(StackType));
    s->top = -1;
}

void DestroyStack(StackType * &s) // 销毁栈
{
    free(s);
}

bool StackEmpty(StackType * s) // 判断栈是否为空
{
    return(s->top == -1);
}

bool Push(StackType * &s, ElemType e) // 进栈
{
    if (s->top == MaxSize - 1)
        return false;
    s->top++;
    s->data[s->top] = e;
    return true;
}

bool Pop(StackType * &s, ElemType &e) // 出栈
{
    if (s->top == -1)
        return false;
    e = s->data[s->top];
    s->top--;
    return true;
}

void Hanoi2(int n, char x, char y, char z)
{
    StackType * st; // 定义顺序栈指针
    ElemType e, e1, e2, e3;
    if (n <= 0) return; // 参数错误时直接返回
    InitStack(st); // 初始化栈
    e.n = n; e.x = x; e.y = y; e.z = z; e.flag = false;
    Push(st, e); // 元素 e 进栈
    while (!StackEmpty(st)) // 栈不空循环
    {
        Pop(st, e); // 出栈元素 e
        if (e.flag == false) // 当不能直接移动盘片时
        {
            e1.n = e.n - 1; e1.x = e.y; e1.y = e.x; e1.z = e.z;
            if (e1.n == 1) // 只有一个盘片时可直接移动
                e1.flag = true;
            else // 有一个以上盘片时不能直接移动

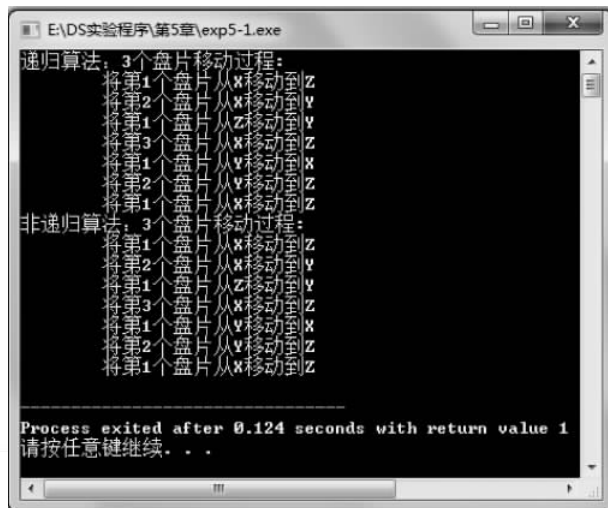
```

```

        e1.flag = false;
        Push(st, e1);                //处理 Hanoi(n-1, y, x, z) 步骤
        e2.n = e.n; e2.x = e.x; e2.y = e.y; e2.z = e.z; e2.flag = true;
        Push(st, e2);                //处理 move(n, x, z) 步骤
        e3.n = e.n-1; e3.x = e.x; e3.y = e.z; e3.z = e.y;
        if (e3.n == 1)                //只有一个盘片时可直接移动
            e3.flag = true;
        else
            e3.flag = false;          //有一个以上盘片时不能直接移动
        Push(st, e3);                //处理 Hanoi(n-1, x, z, y) 步骤
    }
    else                                //当可以直接移动时
        printf("\t将第 %d 个盘片从 %c 移动到 %c\n", e.n, e.x, e.z);
    }
    DestroyStack(st);                //销毁栈
}
// -----
int main()
{
    int n = 3;
    printf("递归算法: %d 个盘片移动过程:\n", n);
    Hanoi1(n, 'X', 'Y', 'Z');
    printf("非递归算法: %d 个盘片移动过程:\n", n);
    Hanoi2(n, 'X', 'Y', 'Z');
    return 1;
}

```

 exp5-1. cpp 程序的执行结果如图 5.2 所示。



```

E:\DS实验程序\第5章\exp5-1.exe
递归算法: 3个盘片移动过程:
将第1个盘片从X移动到Z
将第2个盘片从X移动到Y
将第1个盘片从Z移动到Y
将第3个盘片从X移动到Z
将第1个盘片从Y移动到X
将第2个盘片从Y移动到Z
将第1个盘片从X移动到Z
非递归算法: 3个盘片移动过程:
将第1个盘片从X移动到Z
将第2个盘片从X移动到Y
将第1个盘片从Z移动到Y
将第3个盘片从X移动到Z
将第1个盘片从Y移动到X
将第2个盘片从Y移动到Z
将第1个盘片从X移动到Z
Process exited after 0.124 seconds with return value 1
请按任意键继续...

```

图 5.2 exp5-1. cpp 程序执行结果

实验题 2: 求路径和路径条数问题

目的: 领会基本递归算法设计和递归执行过程。

内容：编写程序 exp5-2.cpp 求路径和路径条数问题。有一个 $m \times n$ 的网格，如图 5.3 所示是一个 2×5 的网格。现在一个机器人位于左上角，该机器人在任何位置上时，只能向下或者向右移动一步，问机器人达到网格的右下角(1,1)位置的所有可能的路径条数，并输出所有的路径。以 $m=2, n=2$ 为例说明输出所有路径的过程。

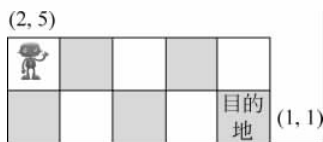


图 5.3 一个 2×5 的网格

 本实验中设计的功能算法如下。

- pathnum(int m , int n): 求解从 (m, n) 到目的地 $(1, 1)$ 的路径条数。
- disppath(int m , int n , PathType path[], int d): 输出从 (m, n) 到目的地 $(1, 1)$ 的所有路径。

pathnum(m, n) 算法思路是：设 $f(m, n)$ 为从 (m, n) 到 $(1, 1)$ 的路径条数，当 $m > 1$ 或者 $n > 1$ 时，可以从 (m, n) 向下移动一步，对应的路径条数为 $f(m-1, n)$ ，也可以向右移动一步，对应的路径条数为 $f(m, n-1)$ 。其递归模型如下：

$$f(m, n) = \begin{cases} 0 & \text{当 } m < 1 \text{ 或者 } n < 1 \\ 1 & \text{当 } m = 1 \text{ 并且 } n = 1 \\ f(m-1, n) + f(m, n-1) & \text{其他情况} \end{cases}$$

实验程序 exp5-2.cpp 的结构如图 5.4 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。

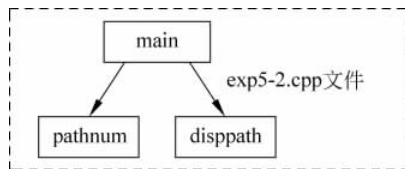


图 5.4 exp5-2.cpp 程序结构

 实验程序 exp5-2.cpp 的程序代码如下：

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

int pathnum(int m, int n) //求解从(m,n)到(1,1)的路径条数
{
    if (m < 1 || n < 1) return 0;
    if (m == 1 && n == 1) return 1;
    return pathnum(m-1, n) + pathnum(m, n-1);
}

typedef struct
{
    int i, j;
} PathType; //路径元素类型

int count = 0; //路径编号

void disppath(int m, int n, PathType path[], int d) //输出从(m,n)到(1,1)的所有路径
{
    if (m < 1 || n < 1) return;
    if (m == 1 && n == 1) //找到目的地,输出一条路径
    {
        d++; //将当前位置放入 path 中
        path[d].i = m; path[d].j = n;
        printf("路径 %d: ", ++count);
        for (int k = 0; k <= d; k++)
```

```

        printf("( %d, %d) ", path[k].i, path[k].j);
        printf("\n");
    }
    else
    {
        d++; //将当前位置放入 path 中
        path[d].i = m; path[d].j = n;
        disppath(m-1, n, path, d); //向下走一步
        disppath(m, n-1, path, d); //退回来, 向右走一步
    }
}

int main()
{
    int m = 2, n = 5;
    printf("m = %d, n = %d 的路径条数: %d\n", m, n, pathnum(m, n));
    PathType path[MaxSize];
    int d = -1;
    disppath(m, n, path, d);
    return 1;
}

```

 exp5-2. cpp 程序的执行结果如图 5.5 所示。

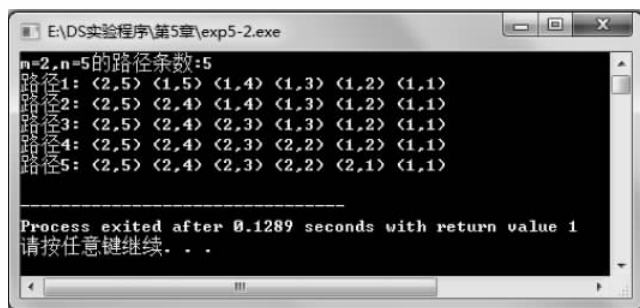


图 5.5 exp5-2. cpp 程序执行结果

当 $m=2, n=2$ 时, disppath 输出所有路径的过程如图 5.6 所示。首先 $path=[]$, 从 $(2,2)$ 开始, 即 $disppath(2,2,path,-1)$, 对应图中结点①。将 $(2,2)$ 加入 path, 调用 $disppath(1,$

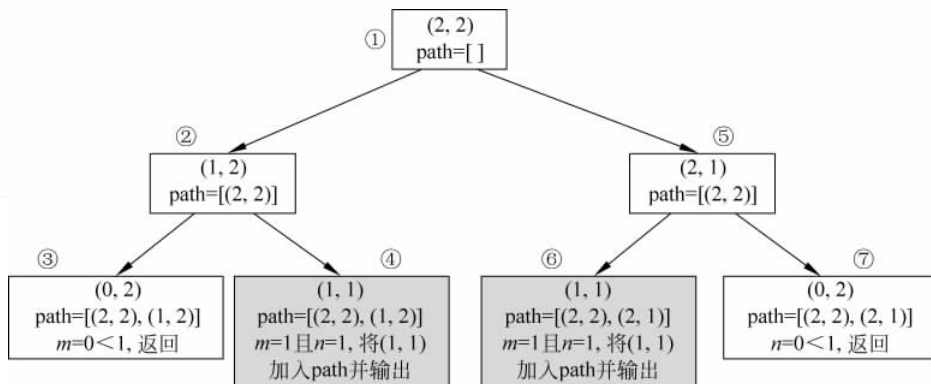


图 5.6 $m=2, n=2$ 时, disppath 的执行过程

2,path,0),对应图中结点②。执行 `disppath(1,2,path,0)`,将(1,2)加入 path,调用 `disppath(0,2,path,1)`,对应图中结点③,此时 $m < 1$,直接返回到结点②。再调用 `disppath(1,1,path,1)`,对应图中结点④,此时满足递归出口条件 $m=1, n=1$,将(1,1)加入 path,并输出 path 构成一条路径,然后返回到结点②,继续返回到结点①。接着调用 `disppath(2,1,path,0)`,对应图中结点⑤,执行过程同上。

因此,递归函数中的形参(非引用型)表示递归状态,在执行时由系统保存,所以可以方便地回退,如从结点④回退到结点①。

5.2

设计性实验



实验题 3: 恢复 IP 地址

目的: 掌握基本递归算法设计。

内容: 编写程序 `exp5-3.cpp` 恢复 IP 地址。给定一个仅包含数字的字符串,恢复它的所有可能的有效 IP 地址。例如,给定字符串为“25525511135”,返回“255.255.11.135”和“255.255.111.35”(顺序可以任意)。

本实验中,用字符数组 `s` 存放仅包含数字的字符串(共 n 个字符),并设计如下类型用于存放恢复的 IP 地址串:

```
typedef struct
{
    char data[MaxSize];           //ip 串
    int length;                   //串长度
} IP;
```

设计的功能算法如下。

- `addch(IP &ip,char ch)`: 在 ip 串的末尾添加一个字符 `ch`。
- `adddot(IP ip)`: 在 ip 串的末尾添加一个“.”,并返回结果。
- `solveip(char s[],int n,int start,int step,IP ip)`: 用于恢复 IP 地址串。一个合法的 IP 地址由 4 个子串构成,以“.”分隔,每个子串为 1~3 位,其数值大于 0 且小于或等于 255。算法中,`start` 用于扫描串 `s`; `step` 表示提取第几个子串。当扫描完 `s` 中所有字符,`step=4`,并且每个子串都合法时,才会产生一个合法的 IP 地址串 `ip`。

实验程序 `exp5-3.cpp` 的结构如图 5.7 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。

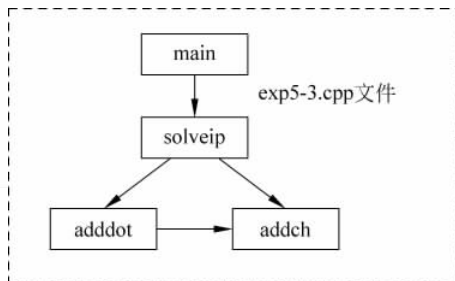


图 5.7 exp5-3.cpp 程序结构

 实验程序 exp5-3.cpp 的程序代码如下:

```
#include <stdio.h>
#define MaxSize 100
typedef struct
{   char data[MaxSize];
    int length;
} IP;
void addch(IP &ip, char ch)           //ip 的末尾添加一个字符 ch
{   ip.data[ip.length] = ch;
    ip.length++;
}
IP adddot(IP ip)                     //ip 的末尾添加一个".", 并返回结果
{   addch(ip, '.');
    return ip;
}
void solveip(char s[], int n, int start, int step, IP ip)    //恢复 IP 地址串
{   if (start <= n)
    {   if (start == n && step == 4)           //找到一个合法解
        {   for (int i = 0; i < ip.length - 1; i++) //输出其结果, 不含最后的一个"."
            printf(" %c", ip.data[i]);
            printf("\n");
        }
    }
    int num = 0;
    for (int i = start; i < n && i < start + 3; i++) //每个子串为 1~3 位
    {   num = 10 * num + (s[i] - '0');           //将 start 开始的 i 个数字转换为数值
        if (num <= 255)                         //为合法点, 继续递归
        {   addch(ip, s[i]);
            solveip(s, n, i + 1, step + 1, adddot(ip));
        }
        if (num == 0) break;                   //不允许前缀 0, 只允许单个 0
    }
}
int main()
{   char s[MaxSize] = "25525511135";
    int n = 11;
    IP ip;
    ip.length = 0;
    solveip(s, n, 0, 0, ip);
    return 1;
}
```

 exp5-3.cpp 程序的执行结果如图 5.8 所示。

实验题 4: 高效求解 x^n

目的: 掌握基本递归算法设计。

内容: 编写程序 exp5-4.cpp 高效求解 x^n 。要求最多使用 $O(\log_2 n)$ 次递归调用。

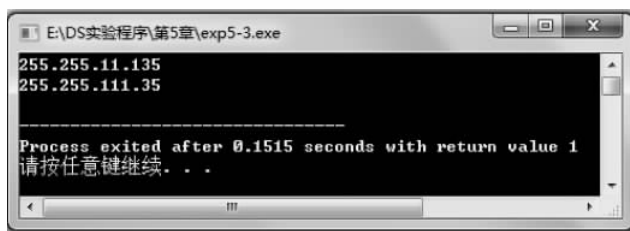


图 5.8 exp5-3. cpp 程序执行结果

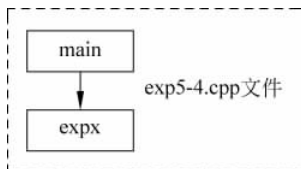
本实验设计的功能算法如下。

- `expx(double x, int n)`: 高效求解 x^n 。

`expx(x, n)` 算法的思路是: 设 $f(x, n) = x^n$, $f(x, n/2) = x^{n/2}$, 当 n 为偶数时, $f(x, n) = x^{n/2} \times x^{n/2} = f(x, n/2) \times f(x, n/2)$; 当 n 为奇数时, $f(x, n) = x \times x^{(n-1)/2} \times x^{(n-1)/2} = x \times f(x, (n-1)/2) \times f(x, (n-1)/2)$ 。对应的递归模型如下:

$$f(x, n) = \begin{cases} x & \text{当 } n = 1 \text{ 时} \\ f(x, n/2) * f(x, n/2) & \text{当 } n \text{ 为大于 1 的偶数时} \\ x * f(x, (n-1)/2) * f(x, (n-1)/2) & \text{当 } n \text{ 为大于 1 的奇数时} \end{cases}$$

实验程序 `exp5-4. cpp` 的结构如图 5.9 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 `exp5-4. cpp` 的程序代码如下:

图 5.9 exp5-4. cpp 程序结构

```
#include <stdio.h>

double expx(double x, int n)
{
    if (n == 1)
        return x;
    else if (n % 2 == 0) //当 n 为大于 1 的偶数时
        return expx(x, n/2) * expx(x, n/2);
    else //当 n 为大于 1 的奇数时
        return x * expx(x, (n-1)/2) * expx(x, (n-1)/2);
}

int main()
{
    double x;
    int n;
    printf("x:"); scanf("%lf", &x);
    printf("n:"); scanf("%d", &n);
    printf("%g 的 %d 次方: %g\n", x, n, expx(x, n));
    return 1;
}
```

 exp5-4. cpp 程序的一次执行结果如图 5.10 所示。

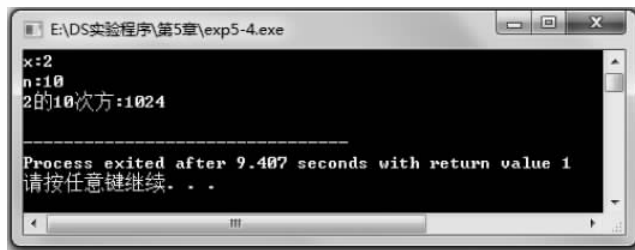


图 5.10 exp5-4. cpp 程序执行结果

实验题 5: 用递归方法逆置带头结点的单链表

目的: 掌握单链表递归算法设计方法。

内容: 编写一个程序 exp5-5. cpp, 用递归方法逆置一个带头结点的单链表。

 本实验设计的功能算法如下。

- Reverse(LinkNode * p, LinkNode * &L): 逆置带头结点的单链表 L。

Reverse(p, L) 算法的思路是: 逆置以 p 为首结点指针的单链表(不带头结点), 逆置后 p 指向尾结点, 它是“大问题”; Reverse($p \rightarrow next, L$) 是“小问题”, 用于逆置以 $p \rightarrow next$ 为首结点指针的单链表, 逆置后 $p \rightarrow next$ 指向尾结点。递归模型如下:

Reverse(p, L) $\equiv L \rightarrow next = p$ 以 p 为首结点指针的单链表只有一个结点时

Reverse(p, L) $\equiv$ Reverse($p \rightarrow next, L$); 其他情况

$p \rightarrow next \rightarrow next = p$;

$p \rightarrow next = \text{NULL}$;

实验程序 exp5-5. cpp 的结构如图 5.11 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。

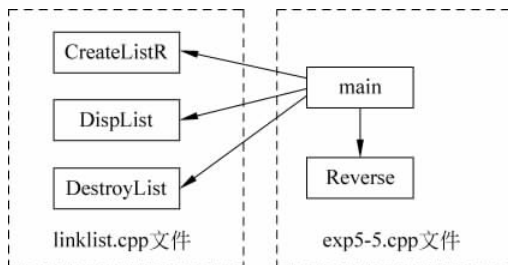


图 5.11 exp5-5. cpp 程序结构

 实验程序 exp5-5. cpp 的程序代码如下:

```
#include "linklist.cpp" //包含单链表的基本运算算法
void Reverse(LinkNode * p, LinkNode * &L)
{   if(p->next == NULL) //以 p 为首结点指针的单链表只有一个结点时
```

```

    {   L->next = p;           //p 结点变为尾结点
        return;
    }
    Reverse(p->next,L);        //逆置后的尾结点是 p->next
    p->next->next = p;          //将结点链接在尾结点之后
    p->next = NULL;            //尾结点 next 域置为 NULL
}

int main()
{   LinkNode * L;
    char a[] = "12345678";
    int n = 8;
    CreateListR(L,a,n);
    printf("L:"); DispList(L);
    printf("逆置 L\n");
    Reverse(L->next,L);
    printf("L:"); DispList(L);
    DestroyList(L);
    return 1;
}

```

 exp5-5. cpp 程序的一次执行结果如图 5.12 所示。

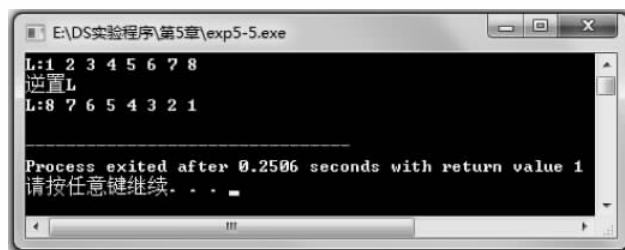


图 5.12 exp5-5. cpp 程序执行结果

实验题 6: 用递归方法求单链表中倒数第 k 个结点

目的: 掌握单链表递归算法设计方法。

内容: 编写一个程序 exp5-6. cpp, 用递归方法求单链表中倒数第 k 个结点。

 本实验设计的功能算法如下。

- $kthNode(LinkNode * L, int k, int \&i)$: 求倒数第 k 个结点。

$kthNode(L, k, i)$ 算法的思路是: 返回不带头结点单链表 L 中的倒数第 k 个结点 (i 用于全局计数倒数第几个结点, 从 0 开始), 它是“大问题”; $kthNode(L->next, k, j)$ 是“小问题”, 显然有 $i=j+1$ 。递归模型如下:

$$kthNode(L, k, i) = \begin{cases} NULL & \text{当 } L=NULL \text{ 时} \\ L & \text{若 } p=kthNode(L->next, k, i), \text{ 有 } i+1=k \text{ 成立} \\ p & \text{其他情况} \end{cases}$$

实验程序 exp5-6. cpp 的结构如图 5.13 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。

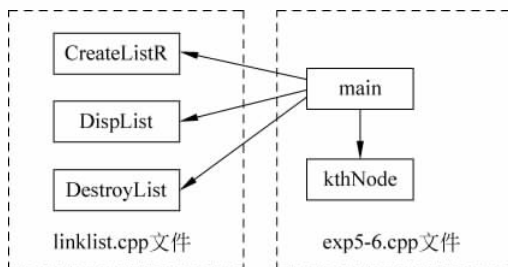


图 5.13 exp5-6.cpp 程序结构

实验程序 exp5-6.cpp 的程序代码如下：

```
#include "linklist.cpp" //包含单链表的基本运算算法
LinkNode * kthNode(LinkNode * L, int k, int &i) //求倒数第 k 个结点
{
    LinkNode * p;
    if (L == NULL) return NULL; //空表返回 NULL
    p = kthNode(L->next, k, i);
    i++;
    if (i == k) return L;
    return p;
}

int main()
{
    LinkNode * L, * p;
    char a[] = "12345678";
    int n = 8, k = 2, i = 0;
    CreateListR(L, a, n);
    printf("L:"); DispList(L);
    p = kthNode(L->next, k, i);
    if (p != NULL)
        printf("倒数第 %d 个结点: %c\n", k, p->data);
    else
        printf("没有找到\n");
    DestroyList(L);
    return 1;
}
```

exp5-6.cpp 程序的一次执行结果如图 5.14 所示。

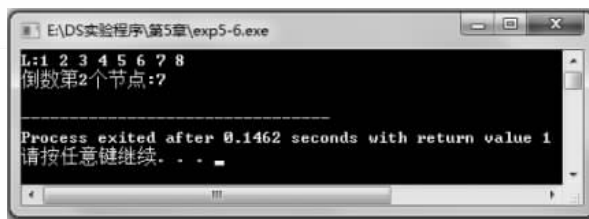


图 5.14 exp5-6.cpp 程序执行结果

5.3

综合性实验

实验题 7: 用递归方法求解 n 皇后问题

目的: 深入掌握递归算法设计方法。

内容: 编写一个程序 exp5-7.cpp, 用递归方法求解 n 皇后问题。 n 皇后问题的描述参见第 3 章实验题 8。

本实验设计的功能算法如下。

- print(int n): 输出一个解。
- place(int k , int j): 测试(k, j)位置能否摆放皇后, 其原理参见第 3 章实验题 7。
- queen(int k , int n): 用于在 $1 \sim k$ 行放置皇后。

queen 算法的思路是: 采用整数数组 $q[N]$ 求解结果, 因为每行只能放一个皇后, $q[i]$ ($1 \leq i \leq n$) 的值表示第 i 个皇后所在的列号, 即该皇后放在($i, q[i]$)的位置上。

设 queen(k, n) 是在 $1 \sim k-1$ 列上已经放好了 $k-1$ 个皇后, 用于在 $k \sim n$ 行放置 $n-k+1$ 个皇后, 是“大问题”; queen($k+1, n$) 表示在 $1 \sim k$ 行上已经放好了 k 个皇后, 用于在 $k+1 \sim n$ 行放置 $n-k$ 个皇后, 显然 queen($k+1, n$) 比 queen(k, n) 少放置一个皇后, 是“小问题”。

求解 n 皇后问题的递归模型如下:

$$\text{queen}(k, n) \equiv \begin{cases} n \text{ 个皇后放置完毕, 输出解} & \text{若 } k > n \\ \text{对于第 } k \text{ 行的每个合适的列位置 } j, \text{ 在其上放置一个皇后;} & \text{其他情况} \\ \text{queen}(k+1, n); \end{cases}$$

得到递归过程如下:

```
queen(int k, int n)
{
    if (k > n)
        输出一个解;
    else
        for (j = 1; j <= n; j++)                //在第 k 行中找所有的列位置
            if (第 k 行的第 j 列合适)
            {
                在(k, j)位置处放一个皇后, 即 q[k] = j;
                queen(k + 1, n);
            }
}
```

实验程序 exp5-7.cpp 的结构如图 5.15 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。

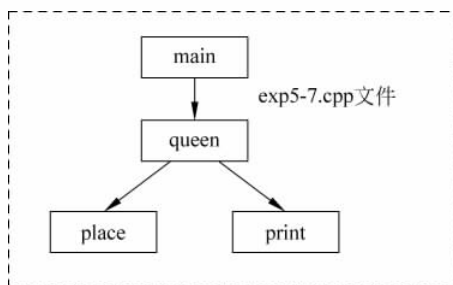


图 5.15 exp5-7. cpp 程序结构

 实验程序 exp5-7. cpp 的程序代码如下：

```

#include <stdio.h>
#include <stdlib.h>
const int N = 20;           //最多皇后个数
int q[N];                   //存放各皇后所在的列号
int count = 0;              //存放解个数
void print(int n)           //输出一个解
{
    count++;
    int i;
    printf(" 第 %d 个解: ", count);
    for (i = 1; i <= n; i++)
        printf("( %d, %d) ", i, q[i]);
    printf("\n");
}
bool place(int k, int j)    //测试(k, j)位置能否摆放皇后
{
    int i = 1;
    while (i < k)            //i = 1~k-1 是已放置了皇后的行
    {
        if ((q[i] == j) || (abs(q[i] - j) == abs(i - k)))
            return false;    //有冲突时返回假
        i++;
    }
    return true;            //没有冲突时返回真
}
void queen(int k, int n)    //放置 1~k 的皇后
{
    int j;
    if (k > n)
        print(n);           //所有皇后放置结束
    else
    {
        for (j = 1; j <= n; j++) //在第 k 行上穷举每一个位置
            if (place(k, j))    //在第 k 行上找到一个合适位置(k, j)
            {
                q[k] = j;
                queen(k + 1, n);
            }
    }
}
int main()
{
    int n;                   //n 存放实际皇后个数
    printf(" 皇后问题(n < 20) n:");
}
    
```

```

scanf("%d",&n);
if (n>20)
    printf("n 值太大,不能求解\n");
else
{
    printf(" %d 皇后问题求解如下: \n",n);
    queen(1,n);
    printf("\n");
}
return 1;
}

```

 exp5-7. cpp 程序的一次执行结果如图 5.16 所示。

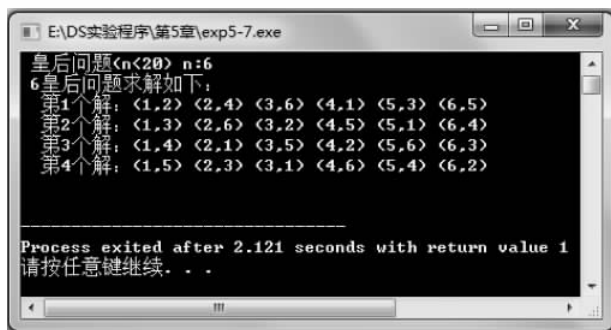


图 5.16 exp5-7. cpp 程序执行结果

实验题 8: 用递归方法求解 0/1 背包问题

目的: 深入掌握递归算法设计方法。

内容: 编写一个程序 exp5-8. cpp, 用递归方法求解 0/1 背包问题。0/1 背包问题是: 设有不同价值、不同重量的物品 n 件, 求从这 n 件物品中选取一部分物品的方案, 使选中物品的总重量不超过指定的限制重量 W , 但选中物品的价值之和为最大。每种物品要么被选中, 要么不被选中。

 本实验程序中, n 表示物品种数, $w[0..n-1]$ 数组存放物品重量, $v[0..n-1]$ 数组存放物品价值, W 表示限制的总重量。产生的解描述为: $\max v$ 存放最优解的总价值; $\max w$ 存放最优解的总重量; 用 x 数组存放最优解, 其中每个元素取 1 或 0, $x[i]=1$ 表示第 i 个物品放入背包中, $x[i]=0$ 表示第 i 个物品不放入背包中。

设计的功能算法如下。

- $\text{dispasolution}(\text{int } x[], \text{int } n)$: 输出 x 中保存的一个解。
- $\text{knap}(\text{int } i, \text{int } tw, \text{int } tv, \text{int } op[])$: 求解 0/1 背包问题。

$\text{knap}(i, tw, tv, op)$ 算法是已考虑了前 $i-1$ 件物品, 现在要考虑第 i 件物品。参数 i 表示考虑第 i 个物品; op 数组的含义与 x 数组一样, 它保存一种临时选择方案; tw 表示 op 方案对应的总重量; tv 表示 op 方案对应的总价值。若 $i \geq n$, 表示考虑了所有物品, 若 $tw \leq W$ 并且 $tv > \max v$, 表示找到一个满足条件的更优解, 将这个解保存; 否则, 考虑第 i 个物品, 有两种方案, 即选中它和不选中它。

显然, $\text{knap}(i, \text{tw}, \text{tv}, \text{op})$ 是“大问题”, 而 $\text{knap}(i+1, *, *, \text{op})$ 是“小问题”(需要考虑的物品个数比大问题少一个)。其递归模型如下:

$$\text{knap}(i, \text{tw}, \text{tv}, \text{op}) \equiv \begin{cases} \text{将找到一个满足条件的更优解保存到 } x \text{ 中} & \text{当 } i \geq n \text{ 时} \\ \text{选中第 } i \text{ 件物品: } \text{op}[i] = 1; \text{knap}(i+1, \text{tw} + w[i], \text{tv} + v[i], \text{op}); \\ \text{不选中第 } i \text{ 件物品: } \text{op}[i] = 0; \text{knap}(i+1, \text{tw}, \text{tv}, \text{op}); & \text{其他情况} \end{cases}$$

实验程序 *exp5-8.cpp* 的结构如图 5.17 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。

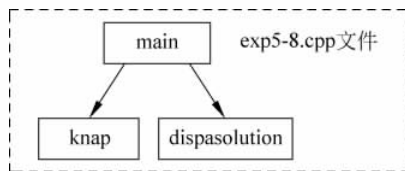


图 5.17 exp5-8.cpp 程序结构

 实验程序 *exp5-8.cpp* 的程序代码如下:

```

#include <stdio.h>
#define MAXN 20 //最多物品数
int maxv; //存放最优解的总价值
int maxw; //存放最优解的总重量
int x[MAXN]; //存放最终解
int W = 7; //限制的总重量
int n = 4; //物品种数
int w[] = {5, 3, 2, 1}; //物品重量
int v[] = {4, 4, 3, 1}; //物品价值
void knap(int i, int tw, int tv, int op[]) //考虑第 i 个物品
{
    int j;
    if (i >= n) //递归出口: 所有物品都考虑过
    {
        if (tw <= W && tv > maxv) //找到一个满足条件的更优解, 保存它
        {
            maxv = tv;
            maxw = tw;
            for (j = 1; j <= n; j++)
                x[j] = op[j];
        }
    }
    else //尚未找完所有物品
    {
        op[i] = 1; //选取第 i 个物品
        knap(i + 1, tw + w[i], tv + v[i], op);
        op[i] = 0; //不选取第 i 个物品, 回溯
        knap(i + 1, tw, tv, op);
    }
}

void dispasolution(int x[], int n) //输出一个解
{
    int i;
    printf("最佳装填方案是:\n");
    for (i = 1; i <= n; i++)
        if (x[i] == 1)
            printf("  选取第 %d 个物品\n", i);
    printf("总重量 = %d, 总价值 = %d\n", maxw, maxv);
}

```

```
}  
int main()  
{   int op[MAXN];           //存放临时解  
    knap(0,0,0,op);  
    dispasolution(x,n);  
    return 1;  
}
```

 exp5-8. cpp 程序的执行结果如图 5.18 所示。这里的 0/1 背包问题是：物品种数为 4，它们的重量分别是 5、3、2、1，价值分别是 4、4、3、1，限制的总重量为 7。最佳方案是选择后 3 个物品，总重量为 6，总价值为 8。

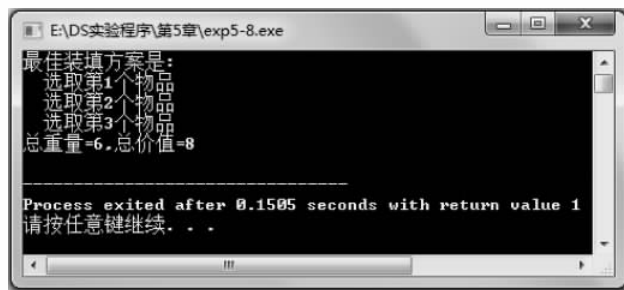


图 5.18 exp5-8. cpp 程序执行结果