第 3 章

栈

学习目标

(1) 理解栈的逻辑特点。

(2) 掌握栈的两种存储结构及其特点。

(3) 学会利用栈及其操作解决实际问题。

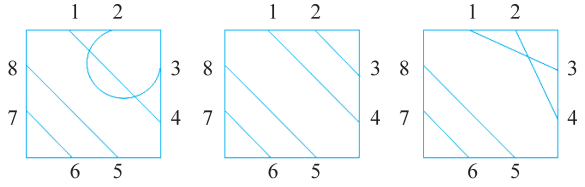
栈是一种常见的数据结构，具有后到先服务的特点。在实际情境中，可以通过一些问题的特点抽象出栈的数据模型。在生活中，常常有这些情况：一系列汽车进入死胡同之后，最先进来的车只能最后出去；在期末考试后，所有学生将试卷依次交在试卷堆的最上方，老师评分时最后才能判到第一个交卷的学生的卷子。这些情况中有一些相似点，一些先到的事情总是较后执行，后到的事件总是优先服务，这是因为只在胡同出口的一端、试卷堆的上方进行存取操作比较方便，而在另一端则不允许存取或者存取不便。所以，对于那些具有后进先出特点的数据，通常使用“栈”这种数据结构来构建模型。

引例 1：马踏棋盘。马踏棋盘也叫骑士周游问题，是指在一个 8×8 的国际象棋棋盘上，从某一位置开始，每次走一个“日”字，将棋盘中所有位置都走 1 遍，要求每个方格只能进入一次。这是一个精典的数学问题，在找路的过程中，需要不断地尝试，当发现到达某个棋盘格时马已经无路可走，即它可以到达的所有方向的棋牌格均已经走过，就需要退回上一步，直到发现有新的路线为止，如图 3.1 所示。那么在进行回溯操作时，怎样把最近尝试的路线删除呢？这种找路的方式可以使用栈来实现。

21	12	7	2	19
6	17	20	13	8
11	22	1	18	3
16	5	24	9	14
23	10	15	4	25

图 3.1　马踏棋盘示意图

引例 2：开关盒布线。给定一个矩形区域，其外围有若干管脚。需要将这些管脚通过电线两两相连，同时不允许电线之间交叉，因为两条电线交叉就会产生电流，导致短路。那么应该如何设置布线方案呢？如图 3.2 所示，3 种布线方案中哪个可行呢？同样可以利用栈结构来确保任意两个管脚所连接的电线都在其他电线的同侧。



(a) 案例1　(b) 案例2　(c) 案例3

图 3.2　开关盒布线示例

本章先讨论栈逻辑结构中后进先出的特性、实现顺序存储和链式存储两种存储结构的定义和基本操作，之后就函数调用栈对递归调用与栈的联系进行分析，最后通过两个需要使用栈的实际应用案例，在算法的编写中使用栈的思想完成相应操作。栈和线性表的知识框架相似，并且将要学习的树与图的章节中都离不开栈的身影。因此，本章既是对线性表的延续，也为后续章节打下基础。

3.1 栈的基本概念

3.1.1 逻辑结构及基本操作

栈(Stack)是一种插入和删除操作受限的线性表，可以进行插入和删除操作的一端称为栈顶(Top)，另一端称为栈底(Bottom)。栈的逻辑结构与线性表相同。栈顶元素是栈中可以被删除的元素，栈底的位置是固定的。在栈中插入元素的操作也称为入栈、进栈或压栈，删除元素的操作也称为出栈、退栈或弹栈。不含任何元素的栈称为空栈。

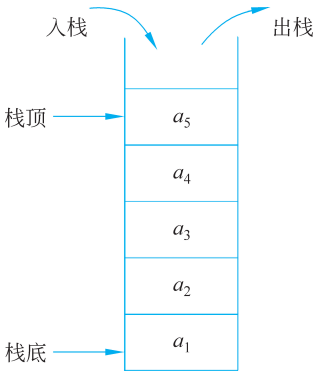


图 3.3 栈的示意图

假设将元素 a_1、a_2、a_3、a_4、a_5 依次压入空栈中，得到的栈 $S=(a_1,a_2,a_3,a_4,a_5)$，如图 3.3 所示，此时 a_1 为栈底元素，a_5 为栈顶元素。接着将 5 个元素依次出栈，得到的出栈次序为 a_5、a_4、a_3、a_2、a_1。我们发现，最后入栈的元素将会最先出栈。因此，栈的操作特性可以明显地概括为后进先出(Last In First Out，LIFO)。

下面就具体案例对栈的基本操作进行简要的分析。假设现在有一个玻璃瓶和若干小球，每个小球上有各自的编号和颜色，且瓶口只允许一个小球通过，那么这个玻璃瓶就可以看作是一个以小球为元素的栈，栈的若干基本操作可以用玻璃瓶与小球的操作来近似看待。现在，把在玻璃瓶中放置和取出小球的操作与栈的基本操作联系起来。

(1) 初始化：拿到一个空的玻璃瓶，并获得瓶子和小球的操作权。

(2) 销毁：不允许再对玻璃瓶进行操作。

(3) 清空：将玻璃瓶中的小球全部取出。

(4) 判空：查看玻璃瓶中是否还有小球。

(5) 求栈的长度：查看玻璃瓶中小球的个数。

(6) 查看栈顶元素：在玻璃瓶中有小球的情况下，查看最上方的小球。

(7) 入栈：将一个小球在玻璃瓶不满的情况下放入玻璃瓶。

(8) 出栈：在玻璃瓶中有小球的情况下，查看最上方的小球并取出。

在接下来的抽象数据类型定义中，会对栈的基本操作进行更为详细的说明。

3.1.2 抽象数据类型

虽然对插入和删除操作的位置限制减少了栈操作的灵活性，但同时也使得栈的操作更容易实现。下面给出栈的抽象数据类型定义。

栈的抽象数据类型如下。

ADT 栈(Stack)定义如下。

数据(Data)：$D=\{a_i \mid a_i \in \text{DataType}, i=1,2,\cdots,n, n\geqslant 0\}$，$D$ 具有先进后出特性。

操作(Operations)如下。

(1) 初始化 InitStack。

过程：栈进行初始化。

(2) 销毁 DestroyStack。

过程：销毁栈。

(3) 判栈空 StackEmpty。

过程：判断栈中是否有元素。

输出：若栈 S 为空栈，则返回 1，否则返回 0。

(4) 求栈长度 StackLength。

过程：获取栈中的元素个数。

输出：栈中元素个数的值。

(5) 求栈顶 GetTop。

过程：查看栈顶元素。

输出：若栈不为空，输出栈顶元素的值；否则抛出“栈空”提示。

(6) 入栈 Push。

输入：将要入栈的元素 e。

过程：将要插入的元素置于栈顶。

输出：若栈不满，元素 e 入栈；否则，输出“栈满”提示。

(7) 出栈 Pop。

过程：删除栈顶元素。

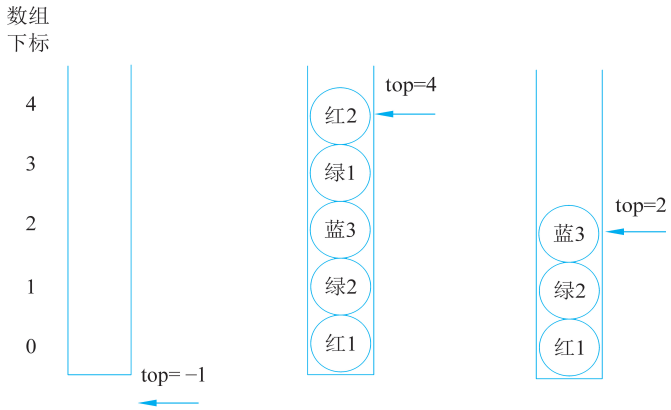
输出：若栈不空，输出栈顶元素；否则，抛出“栈空”提示。

3.2 栈的顺序存储实现

3.2.1 存储结构

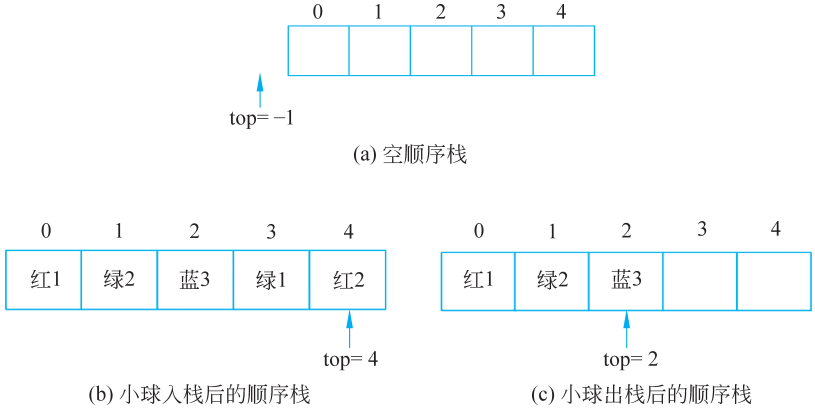
栈的顺序存储结构称为顺序栈(Sequential Stack)，顺序栈在结构上与顺序表极其相似，相邻元素之间不论是在逻辑上还是在结构上都是前后相继的，所以仍需使用数组实现，只需要在顺序表的基础上规定栈顶和栈底分别在数组的哪一端。一般将数组下标为 0 的一端设置为栈底，并且需要设定一个指针 top 指示栈顶元素在数组中的位置，当有元素入栈时，top 的值加 1，出栈时减 1。特别地，当栈为空时，top=−1。图 3.4 是顺序栈操作的示意图。

玻璃瓶和小球的案例可以很好地体现顺序栈的存储结构。初始化一个最大容量为 5 的空栈，top=−1，此时无法进行出栈操作，如图 3.5(a)所示；将 5 个小球依次入栈，每有一个小球入栈，top 加 1，并且在新的栈顶添加入栈元素；top 值为 4 时，栈满，无法再进行入栈的操作，如图 3.5(b)所示；接着出栈两次，等价于取出两次小球，并获得小球信息，每次将 top 减 1，top 指向 a3，栈的长度等于 3，如图 3.5(c)所示；销毁栈时，栈的空间被释放。



(a) 初始化空栈　(b) 5个小球依次入栈　(c) 出栈两次

图 3.4　栈操作示意图



(a) 空顺序栈

(b) 小球入栈后的顺序栈　(c) 小球出栈后的顺序栈

图 3.5　顺序栈操作示意图

下面用 C++ 的类描述顺序栈类及其成员函数。

```
const int MAXSIZE=100;
template <typename DataType>
class SeqStack{
    DataType data[MAXSIZE];                    //存放顺序栈元素
    int top;                                   //指示顺序栈的栈顶位置
public:
    void InitStack();
    int StackLength();
    int StackEmpty();
    DataType GetTop();
    void Push(DataType e);
    DataType Pop();
};
```

在上述定义后，可以通过变量定义栈

```
SeqStack S;
```

下面讲解顺序栈 S 基本操作的实现。

3.2.2 基本操作的实现

实现顺序栈的基本操作时，由于栈中元素的数据类型相同，所以先使用 DataType 来代替，遇到具体问题时，用基本类型或结构体类型对其进行替换。需要注意的是，由于顺序栈是通过数组开辟的内存，所以销毁栈时系统会自动释放，不需要手动进行任何操作。程序 3.1 是构造一个空栈的算法实现，程序 3.2 是求栈的长度的算法实现，程序 3.3 是判断栈是否为空的算法实现，程序 3.4 是查看栈顶元素的算法实现，程序 3.5 是入栈操作的算法实现，程序 3.6 是出栈操作的算法实现。

程序 3.1　栈的初始化

```
void SeqStack::InitStack() {  //构造一个空栈,将栈顶指向-1的位置,表示栈中没有元素
    top=-1;
}
```

程序 3.2　求栈的长度

```
int SeqStack::StackLength(){  //数组中下标为 top 的元素 data[top]为栈顶元素,
                              //所以 top+1 就是栈的长度
    return top+1;
}
```

程序 3.3　栈的判空

```
int SeqStack::StackEmpty(){   //判空操作需要检查 top 的值是否为-1,
                              //如果为-1,则为空,否则不为空
    return top==-1 ? 1:0;
}
```

程序 3.4　查看栈顶元素

```
DataType SeqStack::GetTop(){  //在栈不空的情况下查询 top 在数组中指向的元素。如果
                              //栈不空,输出栈顶元素;否则输出"该栈为空栈"
    if(top!=-1)
        return data[top];
    throw "该栈为空栈";
}
```

程序 3.5　入栈操作

```
void SeqStack::Push(DataType e){              //在栈不满的情况下令 top 加 1,在 top
                                              //指向的位置填入新的元素 e
    if(MAXSIZE>top+1)
        data[++top]=e;
    else
    {
        throw "该栈为满栈";
    }
}
```

程序 3.6　出栈操作

```
DataType SeqStack::Pop(){  //在栈不为空的条件下根据 top 的值获得栈顶元素,并令其出栈
    if(top!=-1)
        return data[top--];
    throw "该栈为空栈";
}
```

3.2.3　共享栈的介绍和基本操作

顺序栈还有一种特殊的表现形式,那就是共享栈(Shared Stack)。共享栈和顺序栈都使用数组存储栈中的元素,只是共享栈在一个数组的区域存储了两个顺序栈,在一定程度上避免了栈空间的浪费。假设数组大小为 MAXSIZE,那么共享栈会将数组下标为 0 和 MAXSIZE−1 作为两个栈的栈底,两个栈的栈顶随着元素的入栈向数组中间扩展,图 3.6 是共享栈空间的示意图。

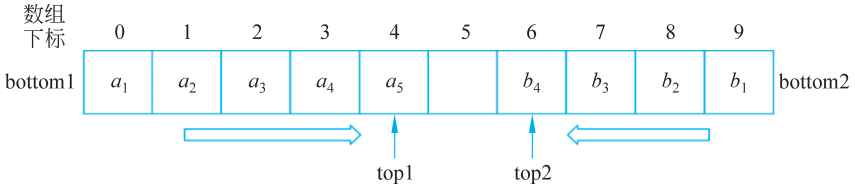


图 3.6　共享栈空间示意图

图中表示的是一个大小为 10 的数组形成的共享栈,两个栈的栈底固定在数组下标为 0 和 9 的位置,栈 1 有数据入栈时,栈顶向右移动,栈 2 则向左移动,出栈时方向相反。同样地,共享栈的销毁也不需要操作。两个栈共享空间的抽象数据类型定义如下:

```
#define MAXSIZE 10                          //定义顺序栈中元素的最大个数
template <typename DataType>
class SharedStack{
    DataType data[MAXSIZE];                 //存放栈中元素的数组
    int top1, top2;                         //指向栈 1 和栈 2 的栈顶元素
public:
    void InitStack();
    int StackLength(int i);
    int StackEmpty(int i);
    DataType GetTop(int i);
    void Push(DataType e, int i);
    DataType Pop(int i);
};
```

下面给出共享栈的部分基本操作。程序 3.7 是栈的初始化的算法实现,程序 3.8 是判断栈是否为空的算法实现,程序 3.9 是求栈的长度的算法实现,程序 3.10 是查看栈顶元素的算法实现,程序 3.11 是入栈操作的算法实现,程序 3.12 是出栈操作的算法实现。

程序 3.7　栈的初始化

```
void SharedStack::InitStack(){                          //初始化栈顶和栈底
```

```
    top1=-1;
    top2=MAXSIZE;
}
```

程序 3.8　栈的判空

```
int SharedStack::StackEmpty(int i){       //根据传入参数 i 的值确定判断哪个栈空
    assert(i==1||i==2);
    if(i==1){                  //栈 1 空的条件是 top1=-1,栈 2 空的条件 top2=MAXSIZE
        if(top1==-1)
            return 1;
        return 0;
    }
    if(top2==MAXSIZE)
        return 1;
    return 0;
}
```

程序 3.9　求栈的长度

```
int SharedStack::StackLength(int i){ //根据传入参数 i 的值确定查询栈 1 或者栈 2 的
                                     //长度
    assert(i==1||i==2);              //如果 i=1,返回栈 1 的长度为 top1+1,否则,返回
                                     //栈 2 的长度为 MAXSIZE-top2
    if(i==1)
        return top1+1;
    return MAXSIZE-top2;
}
```

程序 3.10　查看栈顶元素

```
DataType SharedStack::GetTop(int i){     //根据 i 的值确定查询哪个栈的栈顶元素
    assert(i==1||i==2);                  //如果 i=1,查询 top1 在数组中指向的元素,否
                                         //则返回 top2 在数组中指向的元素
    if(i==1){
        if(top1!=-1)
            return data[top1];
        throw "该栈为空栈";
    }
    if(top2!=MAXSIZE)
        return data[top2];
    throw "该栈为空栈";
}
```

程序 3.11　入栈操作

```
void SharedStack::Push(DataType e,int i){     //判断需要对哪一个栈进行入栈操作
    assert(i==1||i==2);       //在栈不满的情况下,对栈 1 进行入栈操作时,需要令 top1
                              //加 1,栈 2 入栈需要让 top2 减 1
    if(top1<top2-1)
    {
```

```
            if(i==1)
                data[++top1]=e;
            else
                data[--top2]=e;
        }
        else
        {
            throw "该栈为满栈";
        }
    }
```

程序 3.12　出栈操作

```
DataType SharedStack::Pop(int i)
{                                   //判断是哪一个栈进行出栈操作
    assert(i==1||i==2);             //在栈不为空的条件下,栈 1 获得栈顶元素,并令 top1 减 1
    if(i==1){
        if(top1!=-1)
            return data[top1--];
        throw "该栈为空栈";

    if(top2!=MAXSIZE){              //栈 2 获得栈顶元素,并令 top2 加 1。最后,返回栈 1 或栈 2
                                    //的栈顶元素
        return data[top2++];
    throw "该栈为空栈";
}
```

3.3　栈的链式存储实现

3.3.1　存储结构

在使用栈的数据结构时,存储栈的结构也未必是连续的,因此引入了栈的链式存储结构,栈的链式存储结构也称为链栈(Linked Stack),相邻元素之间只有逻辑上是前后相继的,在结构上并不一定前后相继。链栈一般用单链表表示,只不过在基本操作上是受限的。例如,以链式方式存储玻璃瓶中小球的信息,得到的结果如图 3.7 所示。

存储地址	数据域	指针域
1（头指针）	红色1号小球	99
15	紫色2号小球	NULL
64	蓝色3号小球	77
77	绿色2号小球	15
99	绿色1号小球	64

图 3.7　链栈的存储示例

由于单链表中头插和头删的操作比尾插和尾删更方便，所以一般以单链表的头节点作为栈顶，尾节点为栈底。在栈底并不能对栈进行操作，所以单链表中尾节点的位置也不需要存储。图 3.8 是链栈操作的示意图。

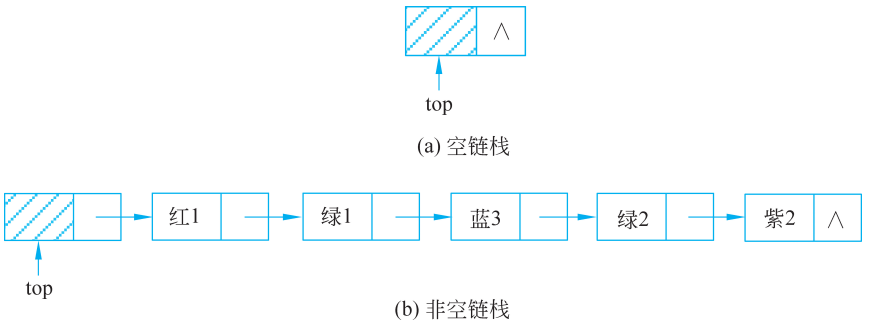


图 3.8　带头节点的链栈

对于小球放入玻璃瓶的操作，需要创建一个单链表的节点，将小球的颜色和编号赋值到该节点的数据域中，并将该节点插入带头节点的栈链中，即设置工作指针 p 指向待插入的节点，令 $p->next=top->next$ 且 $top->next=p$，修改原来栈链中头节点到栈顶元素的指针链，插入新的栈顶元素，如图 3.9 所示。对于小球取出玻璃瓶的操作，只需让头节点指针的 next 域指向当前栈链中栈顶元素的下一个元素即可，但为了能够释放原来栈顶元素的节点内存空间，首先仍需要先设置工作指针 p 指向原来栈链的栈顶元素，即 $p=top->next$，以便释放该节点所占的内存空间，如图 3.10 所示。

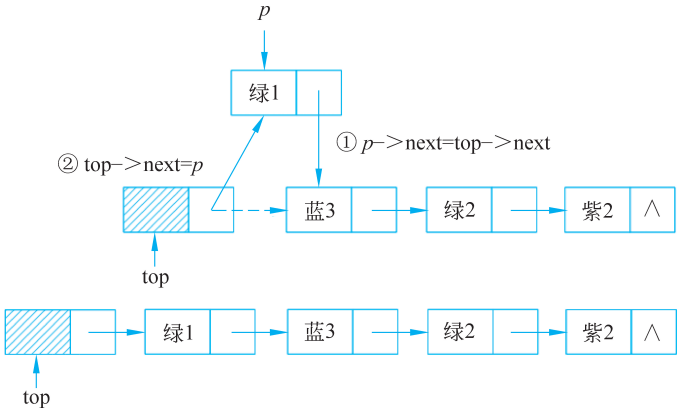


图 3.9　链栈的入栈操作

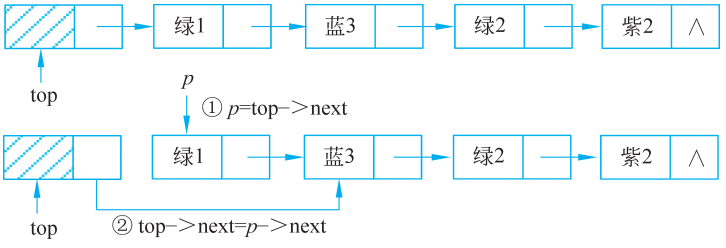


图 3.10　链栈的出栈操作

栈的链式存储结构定义如下：

```
struct Node                                    //链栈节点
{
    DataType data;
    Node * next;
};
```

下面用 C++ 的类描述顺序栈类及其成员函数。

```
emplate <typename DataType>
class LinkStack{
    Node * top;                                //头节点指针
public:
    void InitStack();
    void DestroyStack();
    int StackLength();
    int StackEmpty();
    DataType GetTop();
    void Push(DataType e);
    DataType Pop();
};
```

在上述定义后，可以通过变量定义语句定义栈 S：

```
LinkStack S;
```

下面给出链栈 S 基本操作的实现。

3.3.2 基本操作的实现

链栈与单链表的结构相似，所以基本操作的实现也与单链表类似，链栈中的元素同样是含有数据域和指针域的节点，指针域记录的是其后继节点的地址，如果节点没有后继，则将指针域置为空指针。程序 3.13 是链栈初始化的算法实现，程序 3.14 是销毁链栈的算法实现，程序 3.15 是判断链栈是否为空栈的算法实现，程序 3.16 是求链栈长度的算法实现，程序 3.17 是查看栈顶元素的算法实现，程序 3.18 是链栈入栈操作的算法实现，程序 3.19 是链栈出栈操作的算法实现。

程序 3.13　栈的初始化

```
void LinkStack::InitStack() {   //创建带有头节点的单链表时创建一个节点，并令 top 指向
                                //这个节点
    Node * p = new Node();
    p->next = NULL;
    top=p;
}
```

程序 3.14　栈的销毁

```
void LinkStack::DestroyStack(){   //当栈顶指针不为空时,自栈顶向后遍历所有节点
    while(top!=NULL){             //令工作指针 p 指向栈顶,栈顶指针移动到它的 next
                                  //域,依次释放 p 所指向的内存空间
        Node * p=top;
        top=top->next;
        delete p;
    }
}
```

程序 3.15　栈的判空

```
int LinkStack::StackEmpty(){   //根据 top->next 指向的是否为空指针判断链栈是否为空
    return top->next==NULL? 1:0;
}
```

程序 3.16　求栈的长度

```
int LinkStack::StackLength(){   //当工作指针 p 不空时,自栈顶向后遍历所有节点,并计数
    int len=-1;
    Node * p=top;
    while(p!=NULL){
        p=p->next;
        len++;
    }
    return len;
}
```

程序 3.17　查看栈顶元素

```
DataType LinkStack::GetTop(){   //当栈顶指针不空时,返回其所指向的数据域
    if(top->next!=NULL)
        return top->next->data;
    throw "该栈为空栈";
}
```

程序 3.18　入栈操作

```
void LinkStack::Push(DataType e){   //在栈链中入栈元素
    Node * p=new Node();
    p->data=e;
    p->next=top->next;    //通过改变 p->next 和 top->next 的值就可以插入新的栈顶元素
    top->next=p;
}
```

程序 3.19　出栈操作

```
DataType LinkStack::Pop(){          //在栈不为空的情况下返回 top 指向的元素
    if(top->next==NULL)
    {
        throw "该栈为空栈";
```

```
    }
    DataType e=top->next->data;
    Node *p=top->next;
    top->next=p->next;                    //修改 top 指向原栈顶元素的下一元素
    delete p;                             //释放出原栈元素节点的内存空间
    return e;
}
```

3.4 栈和递归

计算机在执行程序时，会使用栈来记录每层函数调用与执行的过程，多层嵌套函数调用时与栈密切相关。学习程序设计时，我们就学习过递归函数的使用，它是一种自我调用的特殊多层嵌套函数，是算法设计中常用的手段，可以使一个大型复杂问题的描述和求解看起来简洁而清晰。因此，递归算法往往比非递归算法更容易设计，尤其是当问题本身或所涉及的数据结构是递归定义的时候，使用递归方法更加合适。接下来将研究递归函数调用时是如何使用栈这种数据结构的。

3.4.1 函数调用栈

在大部分操作系统中，每个运行中的可执行程序都配有一个调用栈(Callstack)或执行栈(Execution Stack)。借助调用栈可以跟踪一个程序中的所有函数，记录它们之间相互的调用关系，并保证在每一次调用完毕后可以准确返回。

调用栈的基本单位是帧(Frame)，每次函数调用时，都会相应地创建一帧，记录该函数实例在可执行程序中的返回地址，以及局部变量、传入参数等，并将该帧压入调用栈中，成为新的栈顶。函数一旦运行完毕，对应的帧随即弹出，运行控制权将交还给该函数的上层调用函数，并按照该帧中记录的返回地址确定在可执行程序中继续执行的位置。在一些程序的报错中，也可以看出是在函数调用栈中的哪一帧出现了问题，从而快速找到程序中哪个函数出现了问题。

通常，当在一个函数的运行期间调用另一个函数时，在运行被调用函数之前，系统需先完成以下 3 件工作：

(1) 将所有的实参、返回地址等信息传递给被调用函数保存；

(2) 为被调用函数的局部变量分配存储区；

(3) 将控制转移到被调函数的入口。

而从被调用函数返回调用函数之前，系统也应完成以下 3 件工作：

(1) 保存被调函数的计算结果；

(2) 释放被调函数的数据区；

(3) 依照被调函数保存的返回地址，将控制转移到调用函数。

当有多个函数构成嵌套调用时，按照“先调用后返回，后调用先返回”的原则，系统需要将整个程序运行时所需的数据空间安排在一个栈中。每调用一个函数时，就为它在栈顶分配一个存储区，每当从一个函数退出时，就释放它的存储区，则当前正运行的函数的数据区

必在栈顶。

3.4.2 递归调用过程

在函数调用栈中，各帧中记录着前一帧的起始地址，以使其出栈之后能够找到前一帧的函数调用，保证递归函数在函数名相同的情况下也可以正常调用和返回。下面以一个非常经典的递归算法，即计算阶乘的递归函数来观察递归函数调用下栈帧的变化，如图 3.11 所示。

```
long Fac(long n){
    assert(n>=0);
    if(n==0||n==1) return 1;
        return n * Fac(n-1);
}
int main(){
    int i=3;
    cout<<Fac(i)<<endl;
    return 0;
}
```

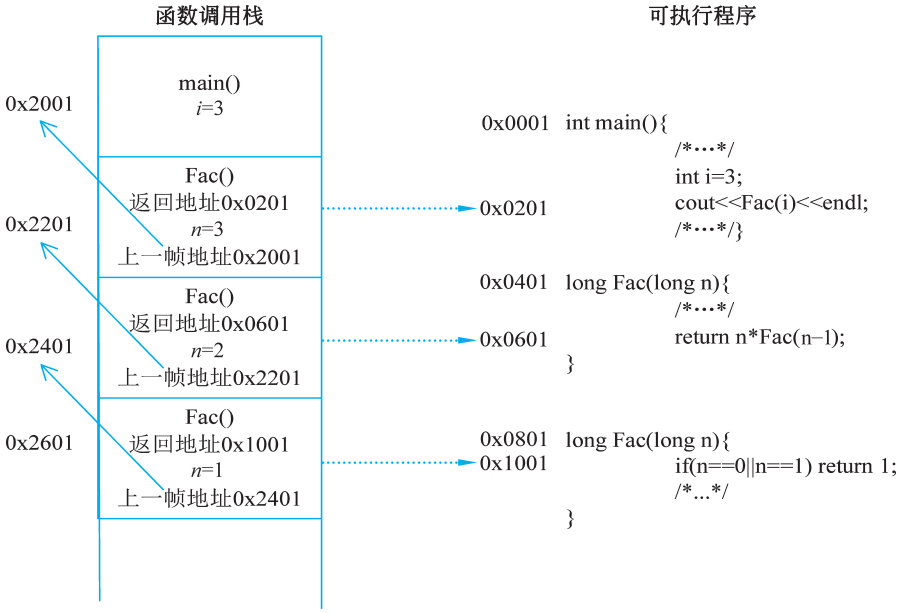


图 3.11 阶乘函数调用栈帧图

从图中可以看出，同一函数在调用栈中各帧的结构完全相同。随着递归函数的逐次调用，函数调用栈中依次压入若干帧函数调用的数据，$n=3$、2、1，需要调用 Fac()函数 3 次，每调用一次函数，调用栈就会进栈一次；当 $n=1$ 时，函数执行完毕后，递归函数调用到达了终点，开始逐帧返回，在一帧调用结束后，找到上一帧地址的位置，就是一帧出栈的过程。随着 Fac(3)计算完成，最终返回，递归函数的调用结束了，在最后 main 函数正常返回时，标志着整个程序的正常结束。

在程序设计中设计递归算法时，有时会遇到栈溢出(Stack Overflow)的报错，这是因为在递归算法中没有正确地设计好调用栈的返回。计算机的内存是有限的，一直有栈帧入栈，而没有栈帧出栈，最终会导致内存栈溢出。所以在递归算法设计的时候，能够合理设计递归返回的出口是非常重要的。

操作系统管理调用栈时，栈帧中难以区分哪些参数和变量对计算过程有实质作用，因此无法找到共性方法进行优化，再加上还要存储大量的地址，指明上一帧的位置与程序继续执行的位置，这往往导致程序的空间复杂度很高。同时，大量的入栈和出栈操作也会令实际的运行时间增加不少。因此在追求更高效率的算法中，应尽可能地避免递归，尤其是过度递归。

3.5 应用

3.5.1 马踏棋盘

1. 问题描述

在棋盘中，将马这颗棋子随机放在棋盘的某个方格中，马按走棋规则(走“日”字)进行移动，并且马在移动时没有蹩马腿的限制。要求每个方格只进入一次，走遍棋盘上全部方格，并由数字依次填入一个棋盘大小的矩阵。例如，在5×5的棋盘中，若下标从1开始，那么以点(3,3)为起始点的棋子马可以如图3.1所示依次遍历整个棋盘。

2. 问题分析

从用户给出的初始位置开始判断，按照顺时针顺序，每次以栈顶路点为起点准备产生一个新的路点，并验证此路点的可用性，需要考虑的是当前路点是否超出棋盘范围和此路点是否走过。如果新路点可用，则将新路点的信息入栈，在棋盘矩阵中更改新路点位置的数值，并执行下一次循环，寻找下一个新的路点；如果一个路点的可扩展路点数为0，则进行回溯，不光没有新路点进栈，还要将栈顶路点信息出栈，在棋盘矩阵中重置该位置的数值为0，一直重复进行以上步骤，直到栈满或者栈空。下面给出该问题的数据模型定义：

```
int board [6][6]={0};                          //记录棋盘内路径顺序的矩阵
int dir_x[8]={-2,-1,1,2,2,1,-1,-2};            //横坐标移动方向
int dir_y[8]={1,2,2,1,-1,-2,-2,-1};            //纵坐标移动方向
template <typename DataType>
struct DataType{                               //定义栈中需要存储的路点信息节点
    int row;                                   //横坐标
    int col;                                   //纵坐标
    int dir_num;                               //值为0~7,指示接下来访问哪个方向
};
```

路点的信息是将要存放在栈中的数据类型，包括这个路点的横纵坐标和路点扩展的方向，路点扩展的方向存储了从这个路点开始找下一个路点将从哪个路径方向开始。从dir_x和dir_y来看，路径方向0到7分别为(−2,+1)、(−1,+2)、(+1,+2)、(+2,+1)、(+2,−1)、(+1,−2)、(−1,−2)、(−2,−1)。假设路点1(3,3)通过方向0(−2,+1)找到了可用的路点2(1,4)，栈中就要把路点1的dir_num改为1，之后再以它为起点时就不会再从方

向 0 找起，而是要从方向 1 开始；若路点 2 没有可用的路点，需要将路点 2 的信息出栈，并令 board[1][4]=0；之后以路点 1(3,3)继续从方向 1(−1,2)开始找，如果最后栈的容量等于棋盘长度的平方，说明马从起始点符合要求地跳到了最后，如果最后栈的容量等于零，说明马每一步都遍历了所有的可能后把所有情况都排除了，马就不能在这一棋盘的这一起点一次性遍历所有位置。

3. 算法实现

经过以上分析，现在要对马踏棋盘问题的算法进行总结。

(1) 初始化一个存放路点信息的栈，将输入的第一个路点入栈。

(2) 顺时针扫描栈顶路点走“日”字时可走的新路点，如果找到，就将这个路点入栈，没找到说明这条线路的这一步是死路，栈顶元素出栈。

(3) 在栈中元素小于 n^2 的条件下，通过 while 循环执行(2)语句，栈中元素等于 n^2，说明找到了马踏棋盘的路径，如果栈为空，说明无解。

下面给出实现上述求解策略的 C++ 程序，算法中使用了 3.2 节中顺序栈的初始化、入栈、出栈和获得栈的长度等操作，还添加了一个返回栈顶元素引用的方法，方便对栈顶元素的值进行修改。程序 3.20 是求解马踏棋盘问题的算法实现。

程序 3.20　马踏棋盘问题求解

```
#include<iostream>
#include<iomanip>
#define MAXSIZE 25
using namespace std;
int board [6][6]={0};
int dir_x[8]={-2,-1,1,2,2,1,-1,-2};
int dir_y[8]={1,2,2,1,-1,-2,-2,-1};
struct DataType{
    int row;
    int col;
    int dir_num;
};
class LabStack{
    DataType data[MAXSIZE];
    int top;
public:
    DataType& GetTop_ref();          //获取栈顶元素引用的方法,方便修改栈顶元素内容
    bool IsOutOrVisited(DataType e);
    void FindStep();
    void Push(DataType e);
    void InitStack();
    int StackLength();
    int StackEmpty();
    DataType Pop();
};
DataType& LabStack::GetTop_ref(){
    if(top!=-1)
        return data[top];
    cout << "该栈为空栈"<< endl;
```

```
    exit(-1);
}
bool LabStack::IsOutOrVisited(DataType e) //判断该节点是否不可用
{
    if(e.row<6 && e.row>0 && e.col<6 && e.col>0)
    {
        if(board[e.row][e.col]==0)
            return false;
        return true;
    }
    return true;
}
void LabStack::FindStep()
{
    DataType& e=GetTop_ref();          //获取栈顶元素引用,方便修改 e.dir_num 的值
    DataType e1;                       //创建新节点
    e1.row=e.row;                      //新节点要以原节点为起点
    e1.col=e.col;
    int i;
    for(i=e.dir_num;i<8;i++)
    //将新节点可能的 8 个方向依次尝试,从 e.dir_num 开始防止访问过的方向再次访问
    {
        e1.row+=dir_x[i];
        e1.col+=dir_y[i];
        if(IsOutOrVisited(e1))         //如果这个方向不可用就要变回之前的位置
        {
            e1.row-=dir_x[i];
            e1.col-=dir_y[i];
        }
        else           //如果新节点可用说明 e 通过方向 i 到了 e1,更新 e.dir_num 为 i+1
        {
            e.dir_num=i+1;
            e1.dir_num=0;
            Push(e1);                  //将新的可用节点 e1 入栈
            board[e1.row][e1.col]=StackLength();  //更新棋盘编号
            break;
        }
      }
      if(i==8)                         //所有的方向都找不到可用的节点
      {
        Pop();                         //弹出该节点
        board[e.row][e.col]=0;         //并且该节点位置重置为 0
    }
}
int main()
{
    int i, j;
    cout<<"请输入棋子起始的横纵坐标:"<<endl;
    cin>>i>>j;
    LabStack S;
```

```
    S.InitStack();
    board[i][j]=1;                          //起始位置在棋盘上编号为 1
    DataType et={i,j,0};
    S.Push(et);                             //第一个可用节点入栈
    while(S.StackLength()<MAXSIZE)
    {
        S.FindStep();                       //栈不满也不空时调用函数找可用节点
        if(S.StackLength()==0)              //栈空说明无解,程序结束
        {
            cout<<"no answer"<<endl;
            return 0;
        }
    }
    for(int row=1;row<6;row++)              //栈满说明有解,打印结果
    {
        for(int col=1;col<6;col++)
        {
            cout<<setw(2)<<board[row][col]<<"  ";
        }
        cout<<endl;
    }
    return 0;
}
```

在时间复杂度方面，假设棋盘的大小为 n，就需要连续地走对满足条件的 n^2 个位置，而每一步棋至多有 8 种走法。所以在最坏的情况下，时间复杂度为 $O(8^{n^2})$，这是一个时间复杂度非常高的算法，所以在 n 比较大时，很难在短时间内找到解或确定无解。在空间复杂度方面，在存储棋盘内容和栈中的路点信息时，都最多需要用到 n^2 的空间，所以空间复杂度为 $O(n^2)$。

3.5.2 开关盒布线

1. 问题描述

在开关盒布线问题中，给定一个矩形的布线区域，区域外有若干管脚，每两个管脚之间通过一条电线连接，电线的连接要在矩形内部布线。两条线路不允许交叉。每对连接的管脚称为一个网组。假设一开关盒中有八个管脚和四个网组，如图 3.2 所示，那么图 3.2(a)和图 3.2(b)表示的网组是(1,4)、(2,3)、(5,8)、(6,7)，但是图 3.2(a)有交叉，图 3.2(b)没有交叉，我们规定，只要一系列网组的连接布线可以做到不交叉，这个开关盒就是可布线的。而对于图 3.2(c)来说，不管线怎么接，用直线还是曲线，都不可能做到不交叉，因此这个开关盒就是不可布线的。我们的问题是，输入一个开关盒布线的所有管脚之间的一系列网组，确定它是否可以布线。

2. 问题分析

为了解决开关盒布线问题，可以从任意一个管脚开始，按顺时针或逆时针方向沿着开关盒的边界进行遍历。如果从管脚 1 开始沿顺时针方向遍历图 3.2(a)中的管脚，那么遍历的管脚顺序是 1,2,…,8。管脚 1 和 4 是一个网组，于是管脚 1 至 4 之间出现的所有管脚构成

第一个分区，管脚 4 至 1 之间出现的所有管脚构成另一个分区。把管脚 1 插入栈，然后继续处理，直到管脚 4。这个过程使我们仅在处理完一个分区之后才能进入下一个分区。下一个是管脚 2，它与管脚 3 是一个网组，它们把当前分区分成两个分区。与前面的做法一样，把管脚 2 插入栈，然后继续处理，直到管脚 3。由于管脚 3 和管脚 2 是一个网组，而管脚 2 正处在栈顶，因此这表明已经处理完一个分区，可从栈顶删除管脚 2。接下来将遇到管脚 4，而与它同是一个网组的管脚 1 正处在栈顶。现在，对一个分区的处理已经完毕，可从栈顶删除管脚 1。按照这种方法继续下去，可以完成对所有分区的处理，而且当 8 个管脚都检查之后，栈为空。

处理不可布线的开关盒时，将会出现什么样的情况呢？图 3.2(c)的网组是(1,3)、(2,4)、(5,8)和(6,7)。开始时，管脚 1 和 2 入栈。当检查到管脚 3 时，因为它与栈顶的管脚不能构成一个网组，所以它入栈。尽管已经扫描到管脚 1 和管脚 3，但还不能结束由这两个管脚所定义的第一个分区的处理过程，因为管脚 2 的网组布线将不得不跨越这个分区的边界。结果是，当检查了所有的管脚时，栈不是空的。

3. 算法实现

经过以上分析，现在要对开关盒布线问题算法进行总结。

(1) 初始化一个存储待匹配管脚的栈。

(2) 遍历管脚数组中各个网组的编号，如果栈为空，将网组编号入栈，否则观察网组编号和栈顶编号是否匹配，若匹配成功，则栈顶编号出栈，匹配失败时，则将不匹配的编号入栈，直到遍历完这个数组为止。

(3) 查看最终栈中的信息，若栈为空，说明整个开关盒是可布线的，否则便是开关盒不可布线。

下面给出实现上述求解策略的 C++ 程序，算法中涉及的数据结构与 3.2 节中顺序栈的实现相同，使用了栈的初始化、入栈操作、出栈操作和栈的判空操作。程序 3.21 是求解开关盒布线问题的算法实现。

程序 3.21　开关盒布线问题求解

```
#include<iostream>
#include<SqStack.h>
#include <assert.h>
using namespace std;
typedef int DataType;
void CheckBox(int net[],int n){
    SqStack S;
    S.InitStack();                    //初始化栈
    for(int i=0;i<n;i++)              //遍历 net 数组
    {
        if(!S.StackEmpty())
            if(net[i]==net[S.GetTop()])
                S.Pop();              //若元素与栈顶元素相等，则网组匹配，弹出栈顶元素
            else
                S.Push(i);            //若不等，说明和栈顶元素不匹配，先将元素下标入栈
        else
            S.Push(i);                //栈为空时，也要将元素下标入栈
```

```
    }
    if(S.StackEmpty())                    //最终栈为空,说明可布线,否则不可布线
        cout<<"此开关盒可布线"<<endl;
    else
        cout<<"此开关盒不可布线"<<endl;
}
int main(){
    int n;
    cout<<"请输入管脚的个数:"<<endl;
    cin>>n;
    cout<<"请输入管脚数组:"<<endl;
    int * net=new int[n];
    for(int i=0;i<n;i++)
        cin>>net[i];
    CheckBox(net,n);
    return 0;
}
```

在时间复杂度方面，假设开关盒中有 n 个管脚，遍历一次管脚数组需要 n 次操作，遍历时进行的入栈或者出栈操作的时间复杂度均为 $O(1)$，所以总的时间复杂度为 $O(n)$。在空间复杂度方面，使用的栈只用了 n 个大小的空间，所以空间复杂度为 $O(n)$。

习题

1. 单选题

（1）栈操作数据的原则是（　　）。

A. 先进先出　　B. 后进先出

（2）（　　）不是栈的基本操作。

A. 删除栈顶元素　　B. 删除栈底元素

C. 判断栈是否为空　　D. 将栈置为空栈

（3）和顺序栈相比，链栈有一个比较明显的优势，即（　　）。

A. 通常不会出现栈满的情况　　B. 通常不会出现栈空的情况

C. 插入操作更容易实现　　D. 删除操作更容易实现

（4）设有一个空栈，栈顶指针为 1000H，每个元素需要一个存储单元，执行 Push、Push、Push、Pop、Push、Pop、Push、Pop、Push 操作后，栈顶指针的值为（　　）。

A. 1002H　　B. 1003H　　C. 1004H　　D. 1005H

（5）设栈的初始状态为空，当字符序列“n1_”作为栈的输入时，输出长度为 3，且可用作 C 语言标识符的出栈序列有（　　）个。

A. 4　　B. 5　　C. 3　　D. 6

（6）设计一个判别表达式中左、右括号是否配对出现的算法，采用（　　）数据结构最佳。

A. 线性表的顺序存储结构　　B. 队列

C. 线性表的链式存储结构　　D. 栈

(7) 设有一个顺序共享栈 Share[0:$n-1$],其中第一个栈顶指针 topl 的初值为-1,第二个栈顶指针 top2 的初值为 n,则判断共享栈满的条件是(　　)。

A. top2$-$top1$==$1　　B. topl$-$top2$==$1

C. topl$==$top2　　D. 以上都不对

(8) 某栈的输入序列是 a、b、c、d,则下列序列中不可能是它的输出序列的是(　　)。

A. a,c,b,d　　B. b,c,d,a　　C. c,d,b,a　　D. d,c,a,b

(9) 一个栈的输入序列是从 1 到 n 的整数,若输出序列的第一个元素是 n,则输出的第 i 个元素是(　　)。

A. 不确定　　B. $n-i+1$　　C. i　　D. $n-i$

(10) 输入序列为 ABC,要变成 ACB,经过的栈操作为(　　)。

A. Push,Pop,Push,Push,Pop,Pop

B. Push,Push,Push,Pop,Pop,Pop

C. Push,Push,Pop,Pop,Push,Pop

D. Push,Pop,Push,Push,Pop,Pop

2. 简答题

(1) 简述栈和线性表的区别。

(2) 设有一个栈,请写出栈常用的两种存储结构和判定栈空和栈满的条件。

(3) 设有一个栈顶指针为 top 的链栈,写出下列操作语句。

① 插入一个 x 节点;

② 进行出栈操作,并将出栈元素存入 x。

(4) 简述如何实现两栈共享空间。

(5) 写出中缀表达式$(a+b)*(c-d*e)\hat{}f$ 的后缀表达式,并写出转换过程中栈的每一步变化的结果。

3. 算法设计题

(1) 试编写一个算法,检查一个程序中的花括号、方括号和圆括号是否配对。

(2) 假设以 I 和 O 分别表示入栈和出栈操作。栈的初态和终态均为空,入栈和出栈的操作序列可表示为仅由 I 和 O 组成的序列,可以操作的序列称为合法序列,否则称为非法序列。写出一个算法,判定所给的操作序列是否合法。若合法,返回 true;否则,返回 false(假定被判定的操作序列已存入一维数组中)。

(3) 利用栈结构,把十进制转换为二至九进制之间的任一进制输出。