# 第3章　栈和队列

栈、队列、优先级队列和双端队列是4种特殊的线性表，它们的逻辑结构与线性表相同，只是其运算规则较线性表有更多的限制，故又称它们为运算受限的线性表。栈和队列被广泛应用于各种系统的程序设计中。

## 3.1　栈

栈是一种最常用和最重要的数据结构，它的用途非常广泛。例如，汇编处理程序中的句法识别和表达式计算就是基于栈实现的。栈还经常使用在函数调用时的参数传递和函数值的返回。

### 3.1.1　栈的定义

通常，**栈**(stack)**可定义为只允许在表的末端进行插入和删除的线性表**。允许插入和删除的一端称为栈顶(top)，而不允许插入和删除的另一端称为栈底(bottom)。当栈中没有任何元素时则成为空栈。

设给定栈 $S=(a_1, a_2, \cdots, a_n)$，则称最后加入栈中的元素 $a_n$ 为栈顶。栈中按 $a_1, a_2, \cdots, a_n$ 的顺序进栈。而退栈的顺序反过来，$a_n$ 先退出，然后 $a_{n-1}$ 才能退出，最后退出 $a_1$。换句话说，后进者先出。因此，栈又称为**后进先出**(last in first out，LIFO)的线性表，如图3.1所示。

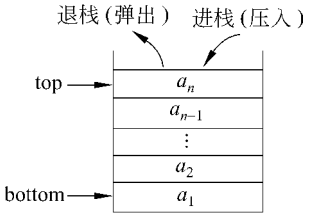


图3.1　栈

与线性表类似，可以借助C++类来定义栈的抽象数据类型，如程序3.1所示。

程序3.1　栈的类定义

```
const int maxSize = 50;
enum bool{false,true};
template <class T>
class Stack {                                    //栈的类定义
public:
    Stack(){};                                   //构造函数
    virtual void Push(T& x) = 0;                 //新元素 x 进栈
    virtual bool Pop(T& x) = 0;                  //栈顶元素出栈，由 x 返回
    virtual bool getTop(T& x)const = 0;          //读取栈顶元素，由 x 返回
    virtual bool IsEmpty() const = 0;            //判断栈空否
    virtual bool IsFull() const = 0;             //判断栈满否
    virtual int getSize() const = 0;             //计算栈中元素个数
};
```

### 3.1.2 顺序栈

栈的抽象数据类型有两种典型的存储表示：基于数组的存储表示和基于链表的存储表示。基于数组的存储表示实现的栈称为顺序栈，基于链表的存储表示实现的栈称为链式栈。

顺序栈可以采用顺序表作为其存储表示，为此，可以在顺序栈的声明中用顺序表定义它的存储空间。本书为简化问题起见，使用一维数组作为栈的存储空间。存放栈元素的数组的头指针为 * elements，该数组的最大允许存放元素个数为 maxSize，当前栈顶位置由数组下标指针 top 指示。如果栈不空时 elements[0]是栈中第一个元素。

程序 3.2　顺序栈的类定义

```
#include <assert.h>
#include <iostream.h>
const int maxStack = 20;
const int stackIncreament = 20;                    //栈溢出时扩展空间的增量
template <class T>
class SeqStack {                                   //顺序栈的类定义
public:
    SeqStack(int sz =50);                          //建立一个空栈
    ~SeqStack( ) {delete[]elements;}               //析构函数
    void Push(T& x);
    //如果 IsFull( ),则溢出处理;否则把 x 插入栈的栈顶
    bool Pop(T& x);
    //如果 IsEmpty( ),则不执行退栈,返回 false;否则退掉位于栈顶的元素,返回 true,
    //退出的元素值通过引用型参数 x 返回
    bool getTop(T& x);
    //如果 IsEmpty( ),则返回 false;否则返回 true,并通过引用型参数 x 得到栈顶元素的值
    bool IsEmpty( )const {return (top == -1) ? true: false;}
    //如果栈中元素个数等于 0,则返回 true,否则返回 false
    bool IsFull( )const {return (top == maxSize-1) ? true: false;}
    //如果栈中元素个数等于 maxSize,则返回 true,否则返回 false
    int getSize( ) (return top+1;)                 //函数返回栈中元素个数
    void MakeEmpty( ) {top = -1;}                  //清空栈的内容
    friend ostream& operator << (ostream& os, SeqStack<T>& s);
    //输出栈中元素的重载操作
private:
    T *elements;                                   //存放栈中元素的栈数组
    int top;                                       //栈顶指针
    int maxSize;                                   //栈最大可容纳元素个数
    void overflowProcess( );                       //栈的溢出处理
};
```

栈的构造函数用于在建立栈的对象时为栈的数据成员赋初值。函数中动态建立的栈数组的最大尺寸为 maxSize，由函数参数 sz 给出，并令 top = -1，置栈为空。在这个函数实现中，使用了一种断言(assert)机制，这是 C++ 提供的一种功能。若断言语句 assert 参数

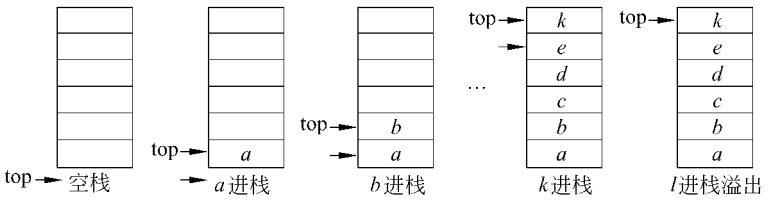
表中给定的条件满足，则继续执行后续的语句；否则出错处理，终止程序的执行。这种断言语句格式简洁，逻辑清晰，不但降低了程序的复杂性，而且提高了程序的可读性。

程序 3.3　顺序栈的构造函数

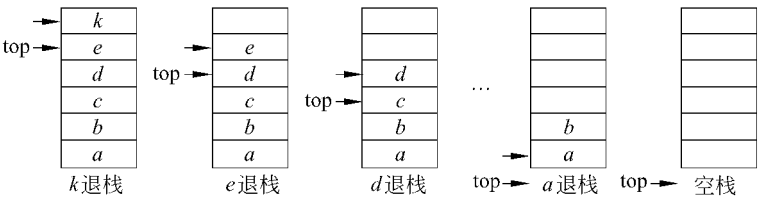
```
template <class T>
SeqStack<T>::SeqStack(int sz):top(-1), maxSize(sz) {
//建立一个最大尺寸为 sz 的空栈，若分配不成功则错误处理
    elements = new T[maxSize];              //创建栈的数组空间
    assert(elements != NULL);               //断言：动态存储分配成功与否
};
```

top 指示的是最后加入的元素的存储位置。在实现进栈操作时，应先判断栈是否已满。栈的最后允许存放位置为 maxSize-1，如果栈顶指针 top == maxSize-1，则说明栈中所有位置均已使用，栈已满。这时若再有新元素进栈，将发生栈溢出，程序转入溢出处理。如果 top < maxSize-1，则先让栈顶指针加 1，指到当前可加入新元素的位置，再按栈顶指针所指位置将新元素插入。这个新插入的元素将成为新的栈顶元素如图 3.2(a)所示。

另一个极端情况出现在栈底：如果在退栈时发现是空栈，即 top == -1，则退栈操作将执行栈空处理。栈空处理一般不是出错处理，而是使用这个栈的算法结束时需要执行的处理。若当前 top≥0，则可将栈顶指针减 1，等于栈顶退回到次栈顶位置如图 3.2(b)所示。




(a) 进栈示例




(b) 退栈示例

图 3.2　进栈和退栈的情况

程序 3.4　栈的其他成员函数的实现

```
template <class T>
void SeqStack<T>::overflowProcess( ) {
//私有函数：扩充栈的存储空间
    T *newArray = new T[maxSize + stackIncreament];
    if (newArray == NULL) {cerr << "存储分配失败!" << endl; exit(1);}
    for (int i = 0; i <= top; i++) newArray[i] = elements[i];
    maxSize = maxSize + stackIncreament;
    delete []elements;
```

```
    elements = newArray;
};

template <class T>
void SeqStack<T>::Push(T& x) {
//共有函数：若栈不满则将元素 x 插入该栈的栈顶，否则溢出处理
    if (IsFull()== true) overflowProcess(); //栈满则溢出处理
    elements[++top] = x;                    //栈顶指针先加 1，再进栈
};

template <class T>
bool SeqStack<T>::Pop(T& x) {
//共有函数：若栈不空则函数返回该栈栈顶元素的值，然后栈顶指针减 1
    if (IsEmpty() == true) return false;    //判断栈空否，若栈空则函数返回
    x = elements[top--];                    //栈顶指针减 1
    return true;                            //退栈成功
};

template <class T>
bool SeqStack<T>::getTop(T& x) {
//共有函数：若栈不空则函数返回该栈栈顶元素的地址
    if (IsEmpty() == true) return false;    //判断栈空否，若栈空则函数返回
    return elements[top];                   //返回栈顶元素的值
    return true;
};

template <class T>
ostream& operator << (ostream& os, SeqStack<T>& s) {
//输出栈中元素的重载操作
    os << "top =" << s.top << endl;         //输出栈顶位置
    for (int i = 0; i <= s.top; i ++)       //逐个输出栈中元素的值
        os << s.elements[i] << "  ";
    os << endl;
    return os;
};
```

读取栈顶元素值的函数 getTop(T&)与退栈函数 Pop(T&)的区别在于前者没有改变栈顶指针的值，后者改变了栈顶指针的值。

当栈满时要发生溢出，为了避免这种情况，需要为栈设立一个足够大的空间。但如果空间设置得过大，而栈中实际只有几个元素，也是一种空间浪费。此外，程序中往往同时存在几个栈，因为各个栈所需的空间在运行中是动态变化着的。如果给几个栈分配同样大小的空间，则在实际运行时，可能有的栈膨胀得快，很快就产生了溢出，而其他的栈可能此时还有许多空闲的空间。这时就必须调整栈的空间，防止栈的溢出。

例如，程序同时需要两个栈时，可以定义一个足够的栈空间。该空间的两端分别设为两

个栈的栈底，用 $b[0](=-1)$ 和 $b[1](=\text{maxSize})$ 指示。让两个栈的栈顶 $t[0]$ 和 $t[1]$ 都向中间伸展，直到两个栈的栈顶相遇，才认为发生了溢出，如图 3.3 所示。

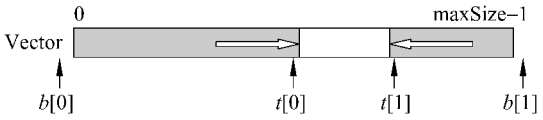



图 3.3　两个栈的情形

注意，每次进栈时 $t[0]$ 加 1，$t[1]$ 减 1；退栈时 $t[0]$ 减 1，$t[1]$ 加 1。

两栈的大小不是固定不变的。在实际运算过程中，一个栈有可能进栈元素多而体积大些，另一个栈则可能小些。两个栈共用一个栈空间，互相调剂，灵活性强。

在双栈的情形下，各栈的初始化语句为 t[0] = b[0] = −1，t[1] = b[1] = maxSize。栈满的条件为 t[0]+1 == t[1]，即当两个栈的栈顶指针相遇才算栈满；栈空的条件为 t[0] = b[0]或 t[1] = b[1]，此时栈顶指针退到栈底。

程序 3.5　双栈的插入和删除操作的实现

```
bool Push (DualStack& DS, T x, int d) {
//在双栈中插入元素 x。d=0,插入第 0 号栈;d≠0,插入第 1 号栈
    if (DS.t[0]+1 == DS.t[1]) return false; //栈满,函数返回
    if (d == 0) DS.t[0]++;                  //栈顶指针加 1
    else DS.t[1]--;
    DS.Vector[DS.t[d]] = x;                 //进栈
    return true;
};

bool Pop(DualStack& DS, T& x, int d) {
//从双栈中退出栈顶元素,通过 x 返回。d=0,从第 0 号栈退栈;d≠0,从第 1 号栈退栈
    if (DS.t[d] == DS.b[d]) return false;   //栈空,函数返回
    x = DS.Vector[DS.t[d]];                 //取出栈顶元素的值
    if (d == 0) DS.t[0]--;                  //栈顶指针减 1
    else DS.t[1]++;
    return true;
};
```

$n(n>2)$ 个栈的情形有所不同，采用多个栈共享栈空间的顺序存储表示方式，处理十分复杂，在插入时元素的移动量很大，因而时间代价较高。特别是当整个存储空间即将充满时，这个问题更加严重。解决的办法就是采用链接存储表方式作为栈的存储表示。

### 3.1.3　链式栈

链式栈是线性表的链接存储表示。采用链式栈来表示一个栈，便于结点的插入与删除。在程序中同时使用多个栈的情况下，用链接存储表示不仅能够提高效率，还可以达到共享存储空间的目的。

从图 3.4 可知，**链式栈的栈顶在链表的表头。因此，新结点的插入和栈顶结点的删除都在链表的表头，即栈顶进行**。下面给出链式栈的类声明。由于第 2 章给出的单链表结点是

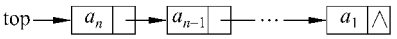



图 3.4　链式栈

用 struct 定义的，在链式栈的情形中可以直接使用，所以在程序 3.6 中没有定义链式栈的结点。

程序 3.6　链式栈的类定义

```
#include <iostream.h>
#include "List.h"                                          //使用了单链表结点
template <class T>
class LinkedStack {                                        //链式栈类定义
public:
    LinkedStack(): top(NULL) {}                            //构造函数，置空栈
    ~LinkedStack() {makeEmpty();};                         //析构函数
    void Push(T x);                                        //进栈
    bool Pop(T& x);                                        //退栈
    bool getTop(T& x)const;                                //读取栈顶元素
    bool IsEmpty()const {return (top == NULL) ? true: false;}
    int getSize()const;                                    //求栈的元素个数
    void makeEmpty();                                      //清空栈的内容
    friend ostream& operator << (ostream& os, LinkedStack<T>& s);
    //输出栈中元素的重载操作
private:
    LinkNode<T> *top;                                      //栈顶指针，即链头指针
};

template <class T>
void LinkedStack<T>::makeEmpty() {
//逐次删去链式栈中的元素直至栈顶指针为空
    LinkNode<T> *p;
    while (top != NULL)                                    //逐个结点释放
        {p = top; top = top->link; delete p;}
};

template <class T>
void LinkedStack<T>::Push(T x) {
//将元素值 x 插入链式栈的栈顶，即链头
    LinkNode<T> *s = new LinkNode<T> (x);                  //创建新的含 x 结点
    if (s == NULL) {cerr << "存储分配失败!" << endl; exit(1);}
    s->link = top; toop =s;
};

template <class T>
bool LinkedStack<T>::Pop(T& x) {
//删除栈顶结点，返回被删栈顶元素的值
```

```
    if (IsEmpty( ) == true) return false;                    //若栈空则不退栈,返回
    LinkNode<T> *p = top;                                    //否则暂存栈顶元素
    top = top->link;                                         //栈顶指针退到新的栈顶位置
    x = p->data; delete p;                                   //释放结点,返回退出元素的值
    return true;
};

template <class T>
bool LinkedStack<T>::getTop(T& x) const {
//返回栈顶元素的值
    if (IsEmpty( ) == true) return false;                    //若栈空则返回 false
    x = top->data;                                           //栈不空则返回栈顶元素的值
    return true;
};

template <class T>
int LinkedStack<T>::getSize( ) {
    LinkNode<T> *p = top; int k = 0;
    while (top != NULL) {top = top->link; k++;}
    return k;
};

template <class T>
ostream& operator << (ostream& os, LinkedStack<T>& s) {
//输出栈中元素的重载操作
    os << "栈中元素个数 =" << s.getSize( ) << endl;          //输出栈中元素个数
    LinkNode<T> *p = S.top; int i = 0;                       //逐个输出栈中元素的值
    while (p != NULL)
        {os << p->data << "  "; p=p->link;}
    os << endl;
    return os;
};
```

如果同时使用 $n$ 个链式栈,其头指针数组可以用以下方式定义:

```
LinkNode<T> *s = new LinkNode<T>[n];
```

在多个链式栈的情形中,link 域需要一些附加的空间,但其代价并非很大。

### **3.1.4 栈的应用之一——括号匹配

举例说明,在一个字符串"($a$ *($b$+$c$)−$d$)"中位置 1 和位置 4 有左括号"(",位置 8 和位置 11 有右括号")"。位置 1 的左括号匹配位置 11 的右括号,位置 4 的左括号匹配位置 8 的右括号。而对于字符串"($a$ +$b$))(",位置 6 的右括号没有可匹配的左括号,位置 7 的左括号没有可匹配的右括号。

我们的目的是建立一个算法,输入一个字符串,输出匹配的括号和没有匹配的括号。

可以观察到，如果从左向右扫描一个字符串，那么每个右括号将与最近遇到的那个未匹配的左括号相匹配。这个观察的结果使我们联想到可以在从左向右的扫描过程中把所遇到的左括号存放到栈中。每当在后续的扫描过程中遇到一个右括号时，就将它与栈顶的左括号(如果存在)相匹配，同时在栈顶删除该左括号。程序 3.7 给出相应的算法，其时间复杂度为 $O(n)$或 $\Theta(n)$，其中 $n$ 是输入串的长度。

程序 3.7　判断括号匹配的算法

```
#include <iostream.h>
#include <string.h>
#include <stdio.h>
#include "SeqStack.cpp"
const int maxLength = 100;                          //最大字符串长度
void PrintMatchedPairs(char *expression) {
    SeqStack<int> s(maxLength);                     //栈 s 存储
    int j, length = strlen(expression);
    for (int i = 1; i <= length; i++) {             //在表达式中搜索"("和")"
        if (expression[i-1] == '(') s.Push(i);      //左括号，位置进栈
        else if (expression[i-1] == ')') {          //右括号
            if (!s.IsEmpty( ) && s.Pop(j))          //栈不空，退栈成功
                cout << j << "与" << i << "匹配" << endl;
            else cout << "没有与第" << i << "个右括号匹配的左括号!" << endl;
        }
    }
    while (!s.IsEmpty( )) {                         //栈中还有左括号
        s.Pop(j);
        cout << "没有与第" << j << "个左括号相匹配的右括号!" << endl;
    }
}
```

同时使用 3 个栈，稍微修改一下程序，就可以同时解决在 C 和 C ++ 程序中的"{"与"}"、"["与"]"、"("与")"的匹配问题。

### **3.1.5　栈的应用之二——表达式的计算

在计算机中执行算术表达式的计算是通过栈来实现的。

**1. 表达式**

如何将表达式翻译成能够正确求值的指令序列，是语言处理程序要解决的基本问题。作为栈的应用实例，下面讨论表达式的求值过程。

任何一个表达式都是由操作数(亦称运算对象)、操作符(亦称运算符)和分界符组成。通常，算术表达式有 3 种表示。

(1) 中缀(infix)表示 ：<操作数> <操作符> <操作数>。例如，$A+B$。

(2) 前缀(prefix)表示 ：<操作符> <操作数> <操作数>。例如，$+AB$。

(3) 后缀(postfix)表示 ：<操作数> <操作数> <操作符>。例如，$AB+$。

我们平时所使用的表达式都是中缀表示。下面就是表达式的中缀表示：

$$A+B*(C-D)-E/F$$

为了正确执行这种中缀表达式的计算，必须明确各个操作符的执行顺序。为此，为每个操作符都规定了一个优先级，如表 3.1 所示。一般表达式的操作符有 4 种类型：①算术操作符，如双目操作符(+、-、*、/ 和%)以及单目操作符(-)。这些操作符主要用于算术操作数。②关系操作符，包括<、<=、==、!=、>=、>。这些操作符主要用于比较，不但适用于算术操作数，而且适用于字符型操作数。③逻辑操作符，如与(&&)、或(||)、非(!)。④括号“(”和“)”。它们的作用是改变运算顺序。操作数可以是任何合法的变量名和常数。

**表 3.1　C++ 中操作符的运算优先级**

| 优先级 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 操作符 | -(单目)，! | *，/，% | +，- | <，<=，>，>= | ==，!= | && | \|\| |

C++ 规定一个表达式中相邻的两个操作符的计算次序：优先级高的先计算；如果优先级相同，则自左向右计算；当使用括号时，从最内层的括号开始计算。

由于中缀表示中有操作符的优先级问题，还有可加括号改变运算顺序的问题，所以对于编译程序来说，一般不使用中缀表示处理表达式。解决办法是用后缀表示(较常用)和前缀表示。因为用后缀表示计算表达式的值只用一个栈，而前缀表示和中缀表示同时需要两个栈，所以**编译程序一般使用后缀表示求解表达式的值**。

例如，日常使用中缀表达式 $A+B*(C-D)-E/F$，计算的执行顺序如图 3.5 所示，$R_1$，$R_2$，$R_3$，$R_4$，$R_5$为中间计算结果。

**2. 应用后缀表示计算表达式的值**

中缀表示是最普通的一种书写表达式的形式，而且在各种程序设计语言和计算器中都使用它。用中缀表示计算表达式的值需要利用两个栈来实现：一个暂存操作数；另一个暂存操作符。利用 "stack.h" 中定义的模板 Stack 类，建立两个不同数据类型的 Stack 对象。

下面讨论的是利用后缀表示计算表达式的值。后缀表示也称 RPN 或逆波兰记号，它是中缀表示的替代形式，参加运算的操作数总在操作符前面。例如，中缀表示 $A+B*(C-D)-E/F$ 所对应的后缀表示为 $ABCD-*+EF/-$。

利用后缀表示计算表达式的值时，从左向右顺序地扫描表达式，并使用一个栈暂存扫描到的操作数或计算结果。例如，与图 3.5 所示的中缀表达式计算等价的后缀表达式计算顺序如图 3.6 所示。在后缀表达式的计算顺序中已经隐含了加括号的优先次序，因而括号在后缀表达式中不出现。

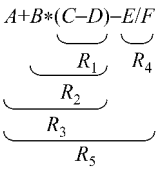



图 3.5　中缀表达式的计算顺序

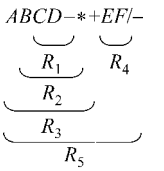



图 3.6　后缀表达式的计算顺序

本节的讨论只涉及双目操作符，不考虑单目操作符。

通过后缀表示计算表达式值的过程(见图 3.7)：顺序扫描表达式的每项，然后根据它的

类型做如下相应操作：如果该项是操作数，则将其压入栈中；如果该项是操作符＜op＞，则连续从栈中退出两个操作数 $Y$ 和 $X$，形成运算指令 $X$＜op＞$Y$，并将计算结果重新压入栈中。当表达式的所有项都扫描并处理完后，栈顶存放的就是最后的计算结果。

| 步 | 扫描项 | 项类型 | 动　　作 | 栈中内容 |
|---|---|---|---|---|
| 1 | | | 置空栈 | 空 |
| 2 | $A$ | 操作数 | 进栈 | $A$ |
| 3 | $B$ | 操作数 | 进栈 | $A\ B$ |
| 4 | $C$ | 操作数 | 进栈 | $A\ B\ C$ |
| 5 | $D$ | 操作数 | 进栈 | $A\ B\ C\ D$ |
| 6 | － | 操作符 | $D$、$C$ 退栈，计算 $C-D$，结果 $R_1$ 进栈 | $A\ B\ R_1$ |
| 7 | * | 操作符 | $R_1$、$B$ 退栈，计算 $B*R_1$，结果 $R_2$ 进栈 | $A\ R_2$ |
| 8 | ＋ | 操作符 | $R_2$、$A$ 退栈，计算 $A+R_2$，结果 $R_3$ 进栈 | $R_3$ |
| 9 | $E$ | 操作数 | 进栈 | $R_3\ E$ |
| 10 | $F$ | 操作数 | 进栈 | $R_3\ E\ F$ |
| 11 | / | 操作符 | $F$、$E$ 退栈，计算 $E/F$，结果 $R_4$ 进栈 | $R_3\ R_4$ |
| 12 | － | 操作符 | $R_4$、$R_3$ 退栈，计算 $R_3-R_4$，结果 $R_5$ 进栈 | $R_5$ |

图 3.7　通过后缀表示计算表达式值的过程

下面通过模拟一个简单的计算器的＋，－，＊，/等运算，进一步说明后缀表达式的求值问题。计算器接收浮点数，计算表达式的值。计算数据和操作都包含在类 Calculator 中，通过一个简单的主程序来调用类的成员函数进行计算。

程序 3.8　Calculator 类的定义

```
#include <math.h>
#include <iostream.h>
#include "SeqStack.cpp"
class Calculator {
//模拟一个简单的计算器。此计算器对从键盘读入的后缀表达式求值
public:
    Calculator(int sz):s(sz) {}                          //构造函数
    double Run(char e[]);                                //执行表达式计算
    void Clear();
private:
    SeqStack<double> s;                                  //栈对象定义
    void AddOperand(double value);                       //进操作数栈
    bool Get2Operands(double& left, double& right);      //从栈中退出两个操作数
    void DoOperator(char op);                            //形成运算指令，进行计算
};
```

因为计算器只要开机就一直运行着，所以需要在开始计算表达式之前先调用成员函数 Clear()将栈清空。然后执行成员函数 Run()输入一个后缀表达式，输入流以＃结束。程序 3.9 给出 Calculator 类各私有成员函数的实现。

程序 3.9　Calculator 类私有成员函数的实现

```
void Calculator::DoOperator(char op) {
//私有成员函数：取两个操作数，根据操作符 op 形成运算指令并计算
    double left,right,vlaue; bool result;
    result = Get2Operands(left, right);                          //取两个操作数
    if (result == true)                                          //如果操作数取成功，计算并进栈
        switch (op) {
        case '+': value = left+right; s.Push(value); break;      //加
        case '-': value = left-right; s.Push(value); break;      //减
        case '*': value = left*right; s.Push(value); break;      //乘
        case '/': if (right == 0.0) {                            //除
                      cerr << "Divide by 0!" << endl;
                      Clear();                                   //若除 0，则报错，清栈
                  }
                  else {value = left/right; s.Push(value);}
                  break;
        }                                                        //没有除 0，做除法
    else Clear();                                                //取操作数出错，清栈
};

bool Calculator::Get2Operands(double& left, double& right) {
//私有成员函数：从操作数栈中取出两个操作数
    if (s.IsEmpty() == true)                                     //检查栈空否
        {cerr << "缺少右操作数!" << endl; return false;}        //栈空，报错
    s.Pop(right);                                                //取右操作数
    if (s.IsEmpty() == true)                                     //检查栈空否
        {cerr << "缺少左操作数!" << endl; return false;}        //栈空，报错
    s.Pop(left);                                                 //取左操作数
    return true;
};

void Calculator::AddOperand(double value) {
//私有成员函数：将操作数的值 value 进操作数栈
    s.Push(value);
};
```

所有的内部运算都在 DoOperator( )的控制下，以调用 Get2Operands( )开始。如果 Get2Operands( )返回 false，则表示操作失败，没有取到两个操作数，执行清栈处理；否则 DoOperator( )执行字符变量 op(+，-，*，/)所指定的操作，并将结果进栈。

计算器的主要工作是通过共有函数 Run( )完成计算后缀表达式的值。在 Run( )中，有一个主循环，从输入流中读取字符，直到读入字符'#'时结束。如果读入的字符是操作符('+','-','*','/')，则调用函数 DoOperator( )完成相关的计算。如果读入的字符不是操作符，则 Run( )把它看作一个操作数。

程序 3.10　Calculator 的实现

```
double Calculator::Run(char e[]) {
//读字符串并求一个后缀表达式的值。以字符'#'结束
    char ch; double newOperand, result; int i = 0;
    ch = e[i++];
    while (ch != '#') {
        switch(ch) {
            case '+': case '-': case '*': case '/':        //是操作符,执行计算
                DoOperator(ch); break;
            default:
                newOperand = (double) ch-'0';                    //转换为操作数
                AddOperand(newOperand);                          //将操作数放入栈中
        }
        ch = e[i++];
    }
    return result;
};

void Calculator::Clear() {                                  //清栈
    s.MakeEmpty();
};
```

在主程序中,可以先建立计算器对象 Calculator CALC,再执行计算程序 CALC.Run(),输入表达式字符流之后,在栈顶就能得到预期的结果。

**3. 利用栈将中缀表示转换为后缀表示**

使用栈可将表达式的中缀表示转换成它的前缀表示和后缀表示。由于篇幅所限,本节仅讨论比较实用的将中缀表示转换为后缀表示的方法。

在中缀表达式中操作符的优先级和括号使得求值过程复杂化,把它转换成后缀表达式,可简化求值过程。为了实现这种转换,需要考虑各个算术操作符的优先级,如表 3.2 所示。

**表 3.2　各个算术操作符的优先级**

| 操作符 ch | # | ( | *, /, % | +, - | ) |
|---|---|---|---|---|---|
| isp | 0 | 1 | 5 | 3 | 6 |
| icp | 0 | 6 | 4 | 2 | 1 |

isp 为栈内(in stack priority)优先数,icp 为栈外(in coming priority)优先数。从表 3.2 中可以看到,左括号的栈外优先数最高,它一来到立即进栈,但当它进入栈中后,其栈内优先数变得极低,以便括号内的其他操作符进栈。其他操作符进入栈中后优先数都加 1,这样可体现在中缀表达式中相同优先级的操作符自左向右计算的要求,让位于栈顶的操作符先退栈并输出。操作符优先数相等的情况只出现在括号配对或栈底的"#"与输入流最后的"#"配对时。前者将连续退出位于栈顶的操作符,直到遇到"("为止,然后将"("退栈以对消括号;后者将结束算法。

扫描中缀表达式将它转换为后缀表达式的算法描述如下。

(1) 操作符栈初始化,将结束符#进栈。然后读入中缀表达式字符流的首字符 ch。

(2) 重复执行以下步骤,直到 ch ='#',同时栈顶的操作符也是'#',停止循环。

① 若 ch 是操作数直接输出,读入下一个字符 ch。

② 若 ch 是操作符,判断 ch 的优先级 icp 和当前位于栈顶的操作符 op 的优先级 isp:

- 若 icp(ch) > isp(op),令 ch 进栈,读入下一个字符 ch。
- 若 icp(ch) < isp(op),退栈并输出。
- 若 icp(ch) == isp(op),退栈但不输出,若退出的是"(",读入下一字符 ch。

(3) 算法结束,输出序列即为所需的后缀表达式。

例如,给定中缀表达式为 $A+B*(C-D)-E/F$,应当转换成 $ABCD-*+EF/-$,按上述算法应执行的转换过程(包括栈的变化和输出)如图 3.8 所示。

| 步 | 扫描项 | 项类型 | 动　作 | 栈的变化 | 输　出 |
|---|---|---|---|---|---|
| 0 | | | '#'进栈 | # | |
| 1 | *A* | 操作数 | | # | *A* |
| 2 | + | 操作符 | isp('#') < icp('+'), 进栈 | #+ | *A* |
| 3 | *B* | 操作数 | | #+ | *AB* |
| 4 | * | 操作符 | isp('+') < icp('*'), 进栈 | #+* | *AB* |
| 5 | ( | 操作符 | isp('*') < icp('('), 进栈 | #+*( | *AB* |
| 6 | *C* | 操作数 | | #+*( | *ABC* |
| 7 | − | 操作符 | isp('(') < icp('−'), 进栈 | #+*(− | *ABC* |
| 8 | *D* | 操作数 | | #+*(− | *ABCD* |
| 9 | ) | 操作符 | isp('−') > icp(')'), 退栈 | #+*( | *ABCD*− |
| | | | '('==')', 退栈 | #+* | *ABCD*− |
| 10 | − | 操作符 | isp('*') > icp('−'), 退栈 | #+ | *ABCD*−* |
| | | | isp('+') > icp('−'), 退栈 | # | *ABCD*−*+ |
| | | | isp('#') < icp('−'), 进栈 | #− | *ABCD*−*+ |
| 11 | *E* | 操作数 | | #− | *ABCD*−*+*E* |
| 12 | / | 操作符 | isp('−') < icp('/'), 进栈 | #−/ | *ABCD*−*+*E* |
| 13 | *F* | 操作数 | | #− / | *ABCD*−*+*EF* |
| 14 | # | 操作符 | isp('/') > icp('#'), 退栈 | #− | *ABCD*−*+*EF*/ |
| | | | isp('−') > icp('#'), 退栈 | # | *ABCD*−*+*EF*/− |
| | | | 结束 | | |

图 3.8　利用栈的转换过程

## 3.2　栈与递归

### 3.2.1　递归的概念

递归(recurve)在计算机科学和数学中是一个很重要的工具,它在程序设计语言中用来定义句法,在数据结构中用来解决表或树形结构的搜索和排序等问题。数学家在研究组合问题时用到递归。递归是一个重要的课题,在计算方法、运筹学模型、行为策略和图论的研

究中，从理论到实践，都得到了广泛的应用。本节将对递归做简要的说明，并举例说明它的各种应用。在以后各章中将利用它来研究诸如树、搜索和排序等问题。

例如，在计算浮点数 $x$ 的 $n$（自然数）次幂时，一般可以把它看作 $n$ 个 $x$ 连乘：

$$x^n = \underbrace{x \times x \times x \times \cdots \times x \times x}_{n\text{个}}$$

而在求前 $n$ 个自然数的和时，则可以把它看作是 $n$ 个自然数的连加：

$$S(n) = \sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + (n-1) + n$$

如果已经求得 $x^n$ 或 $S(n)$，那么在计算 $x^{n+1}$ 或 $S(n+1)$ 时，可以直接利用前面计算过的结果以求得答案：$x^{n+1} = x^n \times x$ 或 $S(n+1) = S(n) + (n+1)$。这样做既简洁又有效。像这种利用前面运算来求得答案的过程称为递归过程。

在数学及程序设计方法学中为递归下的定义：若一个对象部分地包含它自己，或用它自己给自己定义，则称这个对象是递归的；而且若一个过程直接地或间接地调用自己，则称这个过程是递归的过程。

在以下 3 种情况下，常常要用到递归的方法。

**1. 定义是递归的**

数学上常用的阶乘函数、幂函数、斐波那契数列等，它们的定义和计算都是递归的。例如阶乘函数，它的定义为

$$n! = \begin{cases} 1, & n = 0 \\ n(n-1)!, & n > 0 \end{cases}$$

对应这个递归的函数，可以使用递归过程来求解。

程序 3.11　计算阶乘的递归函数

```
long Fact(long n) {
    if (n == 0) return 1;                    //终止递归的条件
    else return n * Fact(n-1);               //递归步骤
}
```

在程序 3.11 中利用了 if … else …语句把递归结束条件与其他表示继续递归的情况区别开来。if 语句块判断递归结束的条件，而 else 语句块处理递归的情况。在计算 $n!$ 时，if 语句块判断唯一的递归结束条件 $n = 0$，并返回值 1；else 语句块通过计算表达式 $n(n-1)!$，并返回计算结果以完成递归。

图 3.9 描述了执行 Fact(4)的函数调用顺序。假设最初是主程序 main()调用了函数。在函数体中，else 语句以参数 3，2，1，0 执行递归调用。最后一次递归调用的函数因参数 $n = 0$ 执行 if 语句。一旦到达递归结束条件，调用函数的递归链中断，同时在返回的途中计算 1 * 1，2 * 1，3 * 2，4 * 6，最后将计算结果 24 返回给主程序。

还可以举出一些函数递归定义的例子。但仅从以上两个例子中已经可以得到以下三点认识。

（1）**对于一个较为复杂的问题，当能够分解成一个或几个相对简单的且解法相同或类似的子问题时，只要解决了这些子问题，原问题就迎刃而解了**。这就是分而治之的递归求解方法，也称**减治法或分治法**。参看图 3.9，计算 4!时先计算 3!。只要求出 3!，就可以求

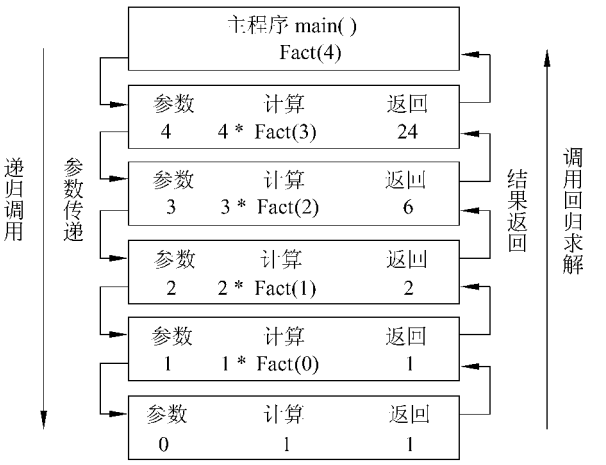



图 3.9　求解阶乘 $n!$ 的过程

出 4!。

(2) **当分解后的子问题可以直接解决时,就停止分解**。这些可以直接求解的问题称为**递归结束条件**。如图 3.9 中递归结束条件是 0! = 1。

(3) **递归定义的函数可以用递归过程来编程求解。递归过程直接反映了定义的结构**。

**2. 数据结构是递归的**

某些数据结构就是递归的。例如,链表就是一种递归的数据结构。链表结点 LinkNode 的定义由数据域 data 和指针域 link 组成;而 link 则由 LinkNode 定义。

从概念上讲,可将一个头指针为 first 的单链表定义为一个递归结构:

(1) first 为 NULL,是一个单链表(空表);

(2) first ≠ NULL,其指针域指向一个单链表,仍是一个单链表。

**对于递归的数据结构,采用递归的方法来编写算法特别方便**。例如,搜索非空单链表最后一个结点并返回其地址,就可以使用递归形式的过程。

程序 3.12　搜索单链表最后一个结点的算法

```
template <class T>
LinkNode <T> * FindRear(LinkNode <T> * f) {
    if (f == NULL) return NULL;
    else if (f->link == NULL) return f;
        else return FindRear(f->link);
};
```

如果 f->link == NULL,表明 $f$ 已到达最后一个结点,此时可返回该结点的地址,否则以 f->link 为头指针继续递归执行该过程。

又例如,在一个非空单链表中搜索其数据域的值等于给定值 $x$ 的结点,并在首次找到时返回其结点地址。在此算法中,递归结束条件有两个:①链表已经全部扫描完但没有找到满足要求的结点,此时 f == NULL;②在 f != NULL 同时 f->data == x 时找到要求的结点。

程序 3.13　在以 f 为头指针的单链表中搜索其值等于给定值 x 的结点

```
template <class T>
LinkNode <T> * Search(LinkNode<T> * f , T& x) {
    if (f == NULL) return;                    //搜索失败
    else if (f->data == x) return f;          //搜索成功
        else return Search(f->link, x);       //从下一个结点开始继续搜索
};
```

不仅是单链表，第 5 章介绍的树形结构，是以多重链表作为其存储表示的，它也是递归的结构。所以关于树的一些算法，也可以用递归过程来实现。

**3. 问题的解法是递归的**

有些问题只能用递归方法来解决。一个典型的例子就是汉诺塔(Tower of Hanoi)问题。问题的提法："传说婆罗门庙里有一个塔台，台上有 3 根标号为 $A$，$B$，$C$ 的用钻石做成的柱子，在 $A$ 柱上放着 64 个金盘，每个都比下面的略小一点。把 $A$ 柱上的金盘全部移到 $C$ 柱上的那一天就是世界末日。移动的条件是一次只能移动一个金盘，移动过程中大金盘不能放在小金盘上面。庙里的僧人一直在移个不停。因为全部的移动是 $2^{63}-1$ 次，如果每秒移动一次，需要 500 亿年。"

一位计算机科学家提出了一种快速求解汉诺塔问题的递归解法。用图解来示意 4 个盘子的情形。设 $A$ 柱上最初的盘子总数为 $n$，问题的解法如下。

如果 $n = 1$，则将这一个盘子直接从 $A$ 柱移到 $C$ 柱上。否则，执行以下 3 步：

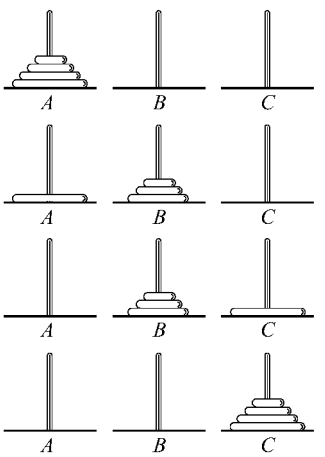


图 3.10　汉诺塔问题的解答

(1) 用 $C$ 柱做过渡，将 $A$ 柱上的 $n-1$ 个盘子移到 $B$ 柱上；

(2) 将 $A$ 柱上最后一个盘子直接移到 $C$ 柱上；

(3) 用 $A$ 柱做过渡，将 $B$ 柱上的 $n-1$ 个盘子移到 $C$ 柱上。

移动过程如图 3.10 所示，图中 $n = 4$。利用这个解法，将移动 $n$ 个盘子的汉诺塔问题归结为移动 $n-1$ 个盘子的汉诺塔问题。与此类似，移动 $n-1$ 个盘子的汉诺塔问题又可归结为移动 $n-2$ 个盘子的汉诺塔问题……最后总可以归结到只移动一个盘子的汉诺塔问题，这样问题就解决了。

现在给出根据上述解法而得到的求解 $n$ 阶汉诺塔问题的算法。

程序 3.14　求解 n 阶汉诺塔问题的算法

```
#include <iostream.h>                         //输入输出流文件
void Hanoi(int n, char A, char B, char C) {
    if (n == 1)
        cout << "Move top disk from peg" << A << "to peg"
            << C << endl;                     //只有一个盘子，直接移动
    else {
        Hanoi(n-1, A, C, B);                  //将上面 n-1 个盘子移到 B 柱
        cout << "Move top disk from peg" << A << "to peg"
```

```
            << C << endl;                    //最后一个移到 C 柱
        Hanoi(n-1, B, A, C);                 //将 B 柱 n-1 个盘子移到 C 柱
    }
};
```

上述递归过程执行的顺序可用如图 3.11 所示 $n = 3$ 的图解描述。

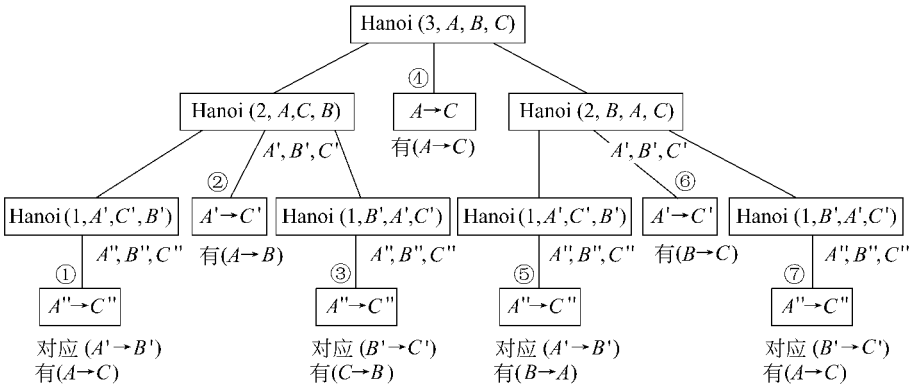



图 3.11　汉诺塔问题的递归调用树

上、下层表示程序调用关系，同一模块的各子模块从左向右顺序执行。各个处于递归结束位置的子模块执行盘片移动功能，最右端子模块执行结束后就实现了上一层模块的功能。编号①②…给出执行次序。

若设盘子总数为 $n$，在算法中盘子的移动次数 moves($n$)为

$$\text{moves}(n)=\begin{cases}0, & n=0\\ 2\text{moves}(n-1)+1, & n>0\end{cases}$$

注意，**递归与递推是两个不同的概念**。递推是利用问题本身所具有的递推关系对问题求解的一种方法。采用递推法建立起来的算法一般具有重要的递推性质，即当求得问题规模为 $i-1$ 的解后，由问题的递推性质，能从已求得的规模为 1，2，…，$i-1$ 的一系列的解，构造出问题规模为 $i$ 的解。若设这种问题的规模为 $n$，当 $n = 0$ 或 $n = 1$ 时，解或为已知，或能很容易地求得。例如，求 $n!$，求等比级数的第 $n$ 项等。

**递推问题可以用递归方法求解，也可以用迭代(重复)的方法求解**。

### 3.2.2　递归过程与递归工作栈

在图 3.9 所示的例子中，主程序调用 Fact(4)属于外部调用，其他调用都属于内部调用，即递归过程在其过程内部又调用了自己。调用方式不同，调用结束时返回的方式也不同。**外部调用结束后，将返回调用递归过程的主程序。内部调用结束后，将返回到递归过程内部本次调用语句的后继语句处**。此外，函数每递归调用一层，必须重新分配一批工作单元，包括本层使用的局部变量、形式参数(实际是上一层传来的实际参数的副本)等，这样可以防止使用数据的冲突，还可以在退出本层，返回到上一层后恢复上一层的数据。

**1. 递归工作栈**

为了保证递归过程每次调用和返回的正确执行，必须解决调用时的参数传递和返回地

址问题。因此,在每次递归过程调用时,必须做参数保存、参数传递等工作。在高级语言的处理程序中,是利用一个递归工作栈来处理的,如图 3.12 所示。

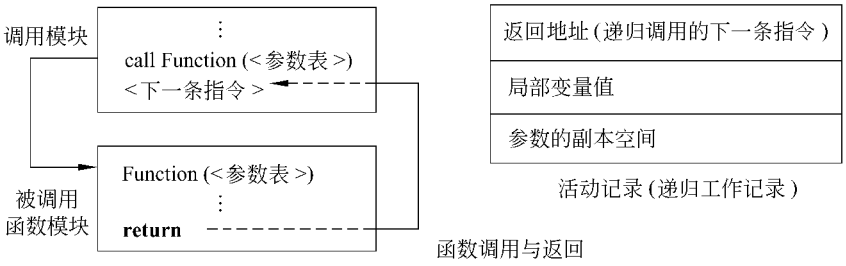



图 3.12　函数递归调用时的活动记录

每层递归调用所需保存的信息构成一个工作记录。通常它包括如下内容。

(1) 返回地址:即上一层中本次调用自己的语句的后继语句处。

(2) 在本次过程调用时,为与形式参数结合的实际参数创建副本。包括传值参数和传值返回值的副本空间,引用型参数和引用型返回值的地址空间。

(3) 本层的局部变量值。

**在每进入一层递归时,系统就要建立一个新的工作记录,把上述项目录入,加到递归工作栈的栈顶**。它构成函数可用的活动框架。**每退出一层递归,就从递归工作栈退出一个工作记录**。因此,栈顶的工作记录必定是当前正在执行的这一层的工作记录。所以又称为活动记录。

以图 3.9 所示的计算 Fact(4)为例,介绍递归过程中递归工作栈和活动记录的使用。

参见程序 3.15。最初对 Fact(4)的调用由主程序执行。当函数运行结束后控制返回到 RetLoc1 处,在此处将函数的返回值 24(即 4!)赋予整型变量 $n$,RetLoc1 在赋值运算符“=”处。函数 Fact(4)递归调用 Fact(3)时,调用返回处在 RetLoc2。在此处计算 $n*(n-1)!$,RetLoc2 在乘法运算符“*”处。

程序 3.15　计算阶乘的递归算法 Fact(4)

```
    void main() {
        long n;                          //调用 Fact(4)时记录进栈
        n = Fact(4);
RetLoc1↑                                 //返回地址 RetLoc1 在赋值语句
        cout << n << endl;
    };

    long Fact(long n) {
        int temp;
        if (n == 0) return 1;            //活动记录退栈
        else temp = n * Fact(n-1);       //调用 Fact(n-1)时活动记录进栈
RetLoc2↑                                 //返回地址 RetLoc2 在计算语句
        return temp;                     //活动记录退栈
    }
```

就 Fact 函数而言，每层调用所创建的活动记录由 3 个域组成：传递过来的实际参数值 $n$ 的副本、返回上一层调用语句的下一条语句的位置和局部变量 temp 如图 3.13(a)所示。

Fact(4)的执行启动了一连串 5 个函数调用。图 3.13(b)描述了每次函数调用时的活动记录。主程序外部调用的活动记录在栈的底部，随内部调用一层层地进栈。递归结束条件出现于函数 Fact(0)的内部，从此开始一连串的返回语句。退出栈顶的活动记录，控制按返回地址转移到上一层调用递归过程处。

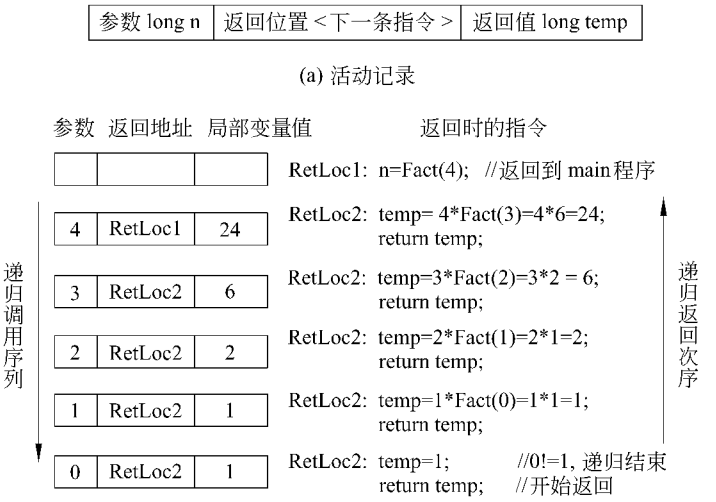



图 3.13　计算 Fact 时活动记录的内容

**2. 用栈实现递归过程的非递归算法**

**对于递归过程，可以利用栈将它改为非递归过程**。此时，可以先通过一个实例了解过程执行时的情况，直接考虑非递归算法。例如，求斐波那契数列的第 $n$ 项 Fib($n$)的公式为

$$\mathrm{Fib}(n)=\begin{cases} n, & n=0 \text{ 或 } 1 \\ \mathrm{Fib}(n-1)+\mathrm{Fib}(n-2), & n\geqslant 2 \end{cases}$$

它对应的递归过程如下。

程序 3.16　斐波那契数列的计算

```
long Fib(long n) {
    if (n <= 1) return n;                    //终止递归的条件
    else return Fib(n-1) + Fib(n-2);
    //递归步骤
};
```

其递归计算的次序可用如图 3.14 所示的递归调用树来描述。为了计算 Fib(4)，必须先计算 Fib(3)；为了计算 Fib(3)，必须先计算 Fib(2)，再计算 Fib(1) 和 Fib(0)，Fib(1)和 Fib(0)可以直接求值。求出 Fib(1)与 Fib(0)后，可以得到 Fib(2)的解，求出 Fib(2)与 Fib(1)后，可以

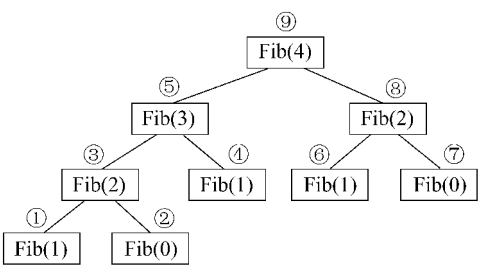



图 3.14　计算斐波那契数列的递归调用树

得到 Fib(3)的解……求解的顺序在图 3.14 上用数字①②③…表示。

为此,可以先计算 Fib(1),从 Fib(4)一直向左下走下去。为了回退,需要用栈记忆回退的路径,以便退回计算。另外为了区分是从左侧退回还是从右侧退回,需要在栈结点中增加一个标志信息 tag。向左递归,tag = 1;向右递归,tag = 2。

程序 3.17　用栈帮助求解斐波那契数列的非递归算法

```
#include "LinkedStack.cpp"
struct Node{                                    //栈结点的类定义
    long n;                                     //记忆走过的 n
    int tag;                                    //区分左、右递归的标志
};

long Fibnacci(long n) {                         //用栈求解 Fib(n)的值
    LinkedStack<Node> S; Node w; long sum = 0;
    do {
        while (n > 1) {w.n = n; w.tag = 1; S.Push(w); n--;}
        sum = sum + n;
        while (S.IsEmpty( ) == false) {
            S.Pop(w);
            if (w.tag == 1)
                {w.tag = 2; S.Push(w); n = w.n-2; break;}
        }
    } while (S.IsEmpty( ) == false);
    return sum;
};
```

该算法执行时栈的变化如图 3.15 所示。每次大循环中包含两个小循环,图中显示了各小循环执行后栈中的内容以及 $n$ 的值和 sum 的计算。最后在 sum 中得到 Fib($n$)的值。

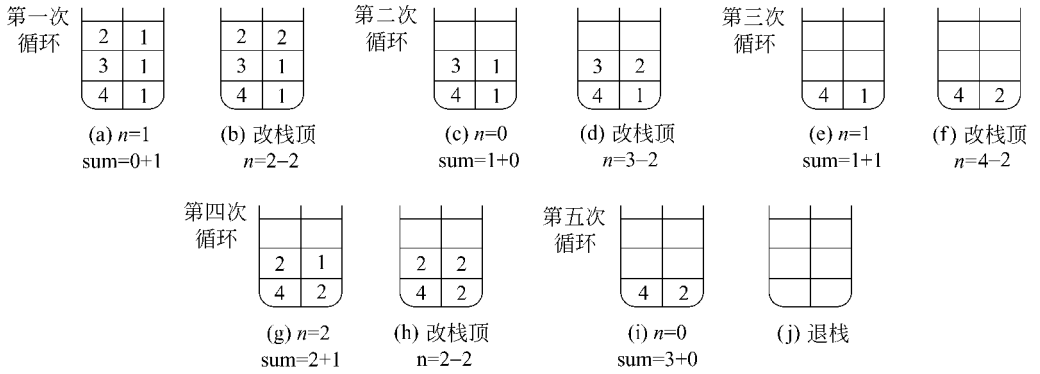



图 3.15　用栈求解斐波那契数列的第 $n$ 项 Fib($n$)时栈的变化

### 3. 用迭代法实现递归过程

在图 3.14 所示的计算斐波那契数列的递归调用树中,计算 Fib(4)时,需要先计算 Fib(3),再计算 Fib(2);计算 Fib(3)时,需要先计算 Fib(2),再计算 Fib(1)……因此,需重复