第4章

栈与队列

[4-1] a） 试基于 3.2.2 节的列表模板类 List，实现栈结构；

【解答】

仿照教材中由Vector类派生Stack类的方法，也可如代码x4.1所示由List类派生Stack类。

```
1 #include "../List/List.h" //以列表为基类，派生出栈模板类
2 template <typename T> class Stack: public List<T> { //将列表的首/末端作为栈顶/底
3 public: //size()、empty()以及其它开放接口，均可直接沿用
4    void push ( T const& e ) { insertAsLast ( e ); } //入栈：等效于将新元素作为列表的首元素插入
5    T pop() { return remove ( last() ); } //出栈：等效于删除列表的首元素
6    T& top() { return last()->data; } //取顶：直接返回列表的首元素
7 };
```

代码x4.1 由List类派生Stack类

这里直接将列表结构视作栈结构，并借助List类已有的操作接口，实现Stack类所需的操作接口。具体地，列表的头部对应于栈底，尾部对应于栈顶。于是，栈顶元素总是与列表的末节点相对应；为将某个元素压入栈中，只需将其作为末节点加至列表尾部；反之，为弹出栈顶元素，只需删除末节点，并返回其中存放的元素。

请注意，这里还同时默认地继承了List类的其它开放接口，但它们的语义与Vector类所提供的同名操作接口可能不尽相同。比如查找操作，List::find()通过返回值（位置）为NULL来表示查找失败，而Vector()则是通过返回值（秩）小于零来表示查找失败。因此在同时还使用了这些接口的算法中，需要相应地就此调整代码实现。

比如，在基于栈结构实现的八皇后算法placeQueens()（教材101页代码4.9）中，（局部）解存放于栈solu中，一旦经过solu.find(q)操作确认没有冲突，即加入一个皇后。教材中所实现版本中的栈结构派生自Vector类，故可通过检查find()所返回的秩是否非负来判定查找是否成功。若改由List类派生出栈结构，则需相应地调整此句——我们将此留给读者独立完成。

另外，将列表节点与栈元素之间的对应次序颠倒过来，从原理上讲也是可行的。但是，同样出于以上考虑，我们还是更加倾向于采用以上的对应次序。如此，栈中元素的逻辑次序与它们在向量或列表中的逻辑次序一致，从而使得栈的标准接口之外的接口具有更好的兼容性。

以hanoi()算法（17页代码x1.13）为例，为了显示盘子移动的过程，必须反复地遍历栈中的元素（盘子）。若采用与代码x4.1相反的对应次序，则针对栈基于Vector和List的两种实现，需要分别更改显示部分的代码；反之，则可以共享同一份代码。

b） 按照你的实现方式，栈 ADT 各接口的效率如何？

【解答】

对于如代码x4.1所示实现的栈结构，各操作接口均转换为常数次列表的基本操作，故与基于向量派生的栈结构一样，所有接口各自仅需$O(1)$时间。

[4-2] a）试基于2.2.3节的向量模板类Vector，实现队列结构；

【解答】

仿照以上基于List实现栈结构的方法与技巧，请读者独立完成。

b）在实现过程中你遇到了哪些困难？你是如何解决的？

【解答】

请读者结合自己的实现过程，独立给出解答。

[4-3] 设B为A = { 1, 2, 3, ..., n }的任一排列。

a）试证明，B是A的一个栈混洗，当且仅当对于任意 $1 \le i < j < k \le n$，P中都不含如下模式：

{ ..., k, ..., i, ..., j, ...}

【解答】

先证明“仅当”。为此可以采用反证法。

首先请注意，对于输入序列中的任意三个元素，其在输出序列中是否存在一个可行的相对排列次序，与其它元素无关。因此不妨只关注这三个元素{ i, j, k }。

接下来可注意到，无论如何，元素i和j必然先于k（弹出栈A并随即）压入中转栈S。若输出序列{ k, i, j }存在，则意味着在此三个元素中，k必然首先从栈S中弹出。而在k即将弹出之前的瞬间，i和j必然已经转入栈S；而且根据“后进先出”的规律，三者在栈S中（自顶向下）的次序必然是{ k, j, i }。这就意味着，若要k率先从栈S中弹出，则三者压入输出栈B的次序必然是{ k, j, i }，而不可能是{ k, i, j }。

既然以上规律与其余元素无关，{ k, i, j }即可视作判定整体输出序列不可行的一个特征，我们不妨称之为“禁形”（forbidden pattern）。

再证明“当”。

实际上只要按照算法x4.1，则对于不含任何禁形的输出序列，都可给出其对应的混洗过程。

```
1  stackPermutation( B[1, n] ) { //B[]为待甄别的输出序列，其中不含任何禁形
2     Stack S; //辅助中转栈
3     int i = 1; //模拟输入栈A（的栈顶元素）
4     for k = 1 to n { //通过迭代，依次输出每一项B[k]
5        while ( S.empty() || B[k] != S.top() ) //只要B[k]仍未出现在S栈顶
6           S.push( i++ ); //就反复地从栈A中取出顶元素，并随即压入栈S
7        //assert: 只要B[]的确不含任何禁形，则以上迭代就不可能导致栈A的溢出
8        //assert: 以上迭代退出时，S栈必然非空，且S的栈顶元素就是B[k]
9        S.pop(); //因此，至此只需弹出S的栈顶元素，即为我们所希望输出的B[k]
10    }
11 }
```

算法x4.1 确认不含任何禁形的序列都是栈混洗

该算法尽管包含两重循环，但其中实质的push()和pop()操作均不超过$O(n)$次，故其总体时间复杂度应线性正比于输入序列的长度。

算法x4.1只需略作修改，即可实现对栈混洗的甄别：对于{ 1, 2, 3, ..., n }的任一排列，判定其是否为栈混洗。请读者参照以上分析以及注释，独立完成此项工作。当然，你所改进的算法，必须依然具有$O(n)$的时间复杂度。

b） 若对任意 1 ≤ i < j < n，B 中都不含模式：

{ ..., j + 1, ..., i, ..., j, ...}

则 B 是否必为 A 的一个栈混洗？若是，试给出证明；否则，试举一反例。

【解答】

可以证明此类序列B必为A的一个栈混洗，故亦可将：

{ j + 1, i, j }

视作新的一类禁形。为此，不妨将：

{ k, i, j }

{ j + 1, i, j }

分别称作“915”式禁形、“615”式禁形。

显然，此类禁形是a）中禁形的特例，故只需证明“当”：只要B中含有“915”式禁形，则必然也含有“615”式禁形——当然，两类禁形中的i和j未必一致。

以下做数学归纳。假定对于任何的k - i < d，以上命题均成立，考查k - i = d的情况。

不妨设i < j < k - 1，于是元素k - 1在B中相对于i的位置无非两种可能：

1）k - 1居于i的左侧（前方）

此时，{ k - 1, i, j }即为“915”式禁形，由归纳假设，必然亦含有“615”式禁形。

2）k - 1居于i的右侧（后方）

此时，{ k, i, k - 1 }即构成一个“615”式禁形。

c） 若对任意 1 < j < k ≤ n，B 中都不含模式

{ ..., k, ..., j - 1, ..., j, ...}

则 B 是否必为 A 的一个栈混洗？若是，试给出证明；否则，试举一反例。

【解答】

此类序列B未必是A的一个栈混洗，故不能将“945”式特征：

{ k, j - 1, j }

称作禁形。作为反例，不妨考查序列：

B[] = { 2, 4, 1, 3 }

不难验证，其中不含任何的“945”式模式（{ 3, 1, 2 }、{ 4, 1, 2 }、{ 4, 2, 3 }）。但反过来，若对序列B[]应用算法x4.1，却将导致错误（请读者独立验证这一点，并指出错误的位置及原因），这说明该序列并非A的栈混洗。

当然，作为对b）中结论的又一次验证，不难看出该序列的确包含“615”式禁形：

{ 4, 1, 3 }

[4-4] 设 S = { 1, 2, 3, ..., n }，试证明：

a） S 的每个栈混洗都分别对应于由 n 对括号组成的一个合法表达式，且反之亦然；

【解答】

采用数学归纳法。

假设以上命题对少于n对括号的表达式（以及长度短于n的序列）均成立，现考查n的情况。

我们令混洗操作序列中的push/pop操作，与表达式中的左/右括号彼此对应。于是，在任意合法的表达式中，必然存在一对紧邻的左、右括号；相应地，在栈混洗对应的栈操作序列中，也必然存在一对紧邻的push和pop操作。进一步地，将这对括号从表达式中删除后，依然得到一个表达式——只不过长度减二；将这对push和pop操作删除后，也依然得到一个栈混洗所对应的栈操作序列——其长度亦减二。由归纳假设，缩短后的表达式与栈混洗彼此对应。

b） S 共有 Catalan(n) = (2n)!/(n + 1)!/n!个栈混洗。

【解答】

根据以上结论，只需统计n对括号所能组成的合法表达式数目T(n)。

由n对括号组成的任一合法表达式S_n，都可唯一地分解和表示为如下形式：

$$S_n = (S_k)S_{n-k-1}$$

其中，S_k和S_{n-k-1}均为合法表达式，且分别由k和n - k - 1对括号组成。

鉴于k的取值范围为[0, n)，故有如下边界条件和递推式：

$$T(0) = T(1) = 1$$

$$T(n) = \sum_{k=0}^{n-1} T(k)\cdot T(n-k-1)$$

这是典型的Catalan数式递推关系，解之即得题中结论。

[4-5] Internet 超文本 HTML 文档，由成对出现的标志（tag）划分为不同的部分与层次。
类似于括号，与起始标志<myTag>相对应地，结束标志为</myTag>。
常用的 HTML 标志有：文档体（<body>和</body>）、节的头部（<h1>和</h1>）、左对齐（<left>和</left>）、段落（ 和 ）、字体（和）等。

a） 试拓展 paren()算法（教材 93 页代码 4.5），以支持对以上 HTML 标志的嵌套匹配检查；

【解答】

增加switch结构的分支，对每一种HTML标志，都相应地增加一条case语句，且处理方式与paren()算法已给出的分支完全一致。具体的实现请读者独立完成。

b） 继续扩展，以支持对任意“<myTag>...</myTag>”形式标志的嵌套匹配检查。

【解答】

此时，实际上需要处理的匹配括号有无数种，故显然不能简单地逐个增加一条语句。

一种可行的方法需要借助栈结构。具体地，每当遇到一个<myTag>标记，即令其入栈（相当于左括号）。每当遇到一个</myTag>标记，即与当前栈顶处的标记比对。倘若二者匹配，则弹出栈顶标记，然后继续读入并处理下一标记；否则，即可断定该文本（至少）在此处出现失配。

当然，待整个HTML文本扫描完毕，还需再次检查辅助栈。此时唯有栈为空，方可判定整个文本中的标记完全匹配。

[4-6] 教材 95 页代码 4.7 中的 evaluate()算法，需借助 readNumber()函数，根据当前字符及其后续的若干字符，解析出当前的操作数。试实现该函数。

【解答】

一种可行的实现方式，如代码x4.2所示。

```
1 void readNumber ( char*& p, Stack<float>& stk ) { //将起始于p的子串解析为数值，并存入操作数栈
2    stk.push ( ( float ) ( *p - '0' ) ); //当前数位对应的数值进栈
3    while ( isdigit ( * ( ++p ) ) ) //只要后续还有紧邻的数字（即多位整数的情况），则
4       stk.push ( stk.pop() * 10 + ( *p - '0' ) ); //弹出原操作数并追加新数位后，新数值重新入栈
5    if ( '.' != *p ) return; //此后非小数点，则意味着当前操作数解析完成
6    float fraction = 1; //否则，意味着还有小数部分
7    while ( isdigit ( * ( ++p ) ) ) //逐位加入
8       stk.push ( stk.pop() + ( *p - '0' ) * ( fraction /= 10 ) ); //小数部分
9 }
```

代码x4.2 操作数的解析

[4-7] 教材 95 页代码 4.7 中的 evaluate()算法，需借助 orderBetween(op1, op2)函数，判定操作符 op1 和 op2 之间的优先级关系。试利用如代码 4.6（教材 94 页）所示的优先级表，实现该函数。

【解答】

一种可行的实现方式，如代码x4.3所示。

```
1 Operator optr2rank ( char op ) { //由运算符转译出编号
2    switch ( op ) {
3       case '+' : return ADD; //加
4       case '-' : return SUB; //减
5       case '*' : return MUL; //乘
6       case '/' : return DIV; //除
7       case '^' : return POW; //乘方
8       case '!' : return FAC; //阶乘
9       case '(' : return L_P; //左括号
10      case ')' : return R_P; //右括号
11      case '\0': return EOE; //起始符与终止符
12      default  : exit ( -1 ); //未知运算符
13   }
14 }
```

```
15
16 char orderBetween ( char op1, char op2 ) //比较两个运算符之间的优先级
17 { return pri[optr2rank ( op1 ) ][optr2rank ( op2 ) ]; }
```

代码x4.3 运算符优先级关系的判定

[4-8] 教材 95 页代码 4.7 中的 evaluate()算法，为将常规表达式转换为 RPN 表达式，需借助 append()函数将操作数或运算符追加至字符串 rpn 的末尾。
试实现该函数。（提示：需针对浮点数和字符，分别重载一个接口）

【解答】

一种可行的实现方式，如代码x4.4所示。

```
1 void append ( char*& rpn, float opnd ) { //将操作数接至RPN末尾
2    int n = strlen ( rpn ); //RPN当前长度（以'\0'结尾，长度n + 1）
3    char buf[64];
4    if ( opnd != ( float ) ( int ) opnd ) sprintf ( buf, "%.2f \0", opnd ); //浮点格式，或
5    else                                  sprintf ( buf, "%d \0", ( int ) opnd ); //整数格式
6    rpn = ( char* ) realloc ( rpn, sizeof ( char ) * ( n + strlen ( buf ) + 1 ) ); //扩展空间
7    strcat ( rpn, buf ); //RPN加长
8 }
9
10 void append ( char*& rpn, char optr ) { //将运算符接至RPN末尾
11    int n = strlen ( rpn ); //RPN当前长度（以'\0'结尾，长度n + 1）
12    rpn = ( char* ) realloc ( rpn, sizeof ( char ) * ( n + 3 ) ); //扩展空间
13    sprintf ( rpn + n, "%c ", optr ); rpn[n + 2] = '\0'; //接入指定的运算符
14 }
```

代码x4.4 将操作数或操作符统一接至RPN表达式末尾

这里，在接入每一个新的操作数或操作符之前，都要调用realloc()函数以动态地扩充RPN表达式的容量，因此会在一定程度上影响时间效率。

在十分注重这方面性能的场合，读者可以做适当的改进——比如，仿照教材2.4.2节中可扩充向量的策略，凡有必要扩容时即令容量加倍。

[4-9] 试以表达式"(0!+1)*2^(3!+4)-(5!-67-(8+9))"为例，给出 evaluate()算法的完整执行过程。

【解答】

该表达式的求值过程，如表x4.1所示。其中每一步所对应的当前字符，均以方框注明；表达式的结束标识'\0'，则统一用$示意；各行左侧为栈底，右侧为栈顶。

请参考对应的注解，体会运算符栈和操作数栈随算法执行的演变过程及规律。

表x4.1 表达式求值算法实例

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$		表达式起始标识入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (		左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	0	操作数0入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (!	0	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	1	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (+	1	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (+	1 1	操作数1入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	2	运算符'+'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$	2	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2	运算符'*'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2 2	操作数2入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^	2 2	运算符'^'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 3	操作数3入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (!	2 2 3	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 6	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (+	2 2 6	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (+	2 2 6 4	操作数4入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 10	运算符'+'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^	2 2 10	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2 1024	运算符'^'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$	2048	运算符'*'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ -	2048	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 5	操作数5入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (!	2048 5	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 120	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 120	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 120 67	操作数67入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 53	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 53	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53 8	操作数8入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (+	2048 53 8	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (+	2048 53 8 9	操作数9入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53 17	运算符'+'出栈执行

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9)])$	$ - (-	2048 53 17	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9)])$	$ - (	2048 36	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9)])$	$ -	2048 36	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$]	$	2012	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$]		2012	表达式起始标识出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$			返回唯一的元素2012

[4-10] 教材 95 页代码 4.7 中的 evaluate()算法，对乘方运算符"^"的求值采用了向左优先结合律，比如表达式“2^3^5”将被理解为“(2^3)^5”。

试按照通常习惯，将该运算符调整为满足向右优先结合律，比如上例应被理解为“2^(3^5)”。要求对该算法的修改尽可能小。

【解答】

就此类问题而言，与其去修改代码4.7中的evaluate()算法，不如直接调整代码4.6中的优先级表。实际上，只需将其中的pri['^']['^']由'>'改作'<'。

经过如此调整之后，当表达式当前扫描至操作符'^'，且此时的操作符栈顶元素亦为'^'时，后者不会随即执行计算，而是令前者入栈。从优先级的角度来看，如此可保证靠后（而非靠前）的'^'运算符优先执行计算。

[4-11] 教材 95 页代码 4.7 中 evaluate()算法执行过程中的某一时刻，设操作符栈共存有 502 个括号。

a） 此时栈的规模（含栈底的'\0'）至多可能多大？为什么？

b） 请示意性地画出当时栈中的内容。

【解答】

由该算法的原理不难看出，在其执行过程中的任何时刻，操作符栈中所存每一操作符相对于其直接后继（若存在）的优先级都要（严格地）更高。

当然，这一性质只对相邻操作符成立，故并不意味着其中所有的操作符都按优先级构成一个单调序列。在该算法中，（左）括号扮演了重要的角色——无论它是栈顶操作符，或者是表达式中的当前操作符，都会（因对应的pri[][]表项为'<'而）执行压栈操作。就效果而言，如此等价于将递增的优先级复位，从而可以开始新的一轮递增。对照如代码4.6所示的优先级表即不难验证，其它操作符均无这一特性。

因此，在（左）括号数固定的条件下，为使操作符栈中容纳更多的操作符，必须使每个（左）括号的上述特性得以充分发挥。具体地，在每个（左）括号入栈之前，应使每个优先级别的操作符都出现一次（当然，也至多各出现一次）。这里，'+'和'-'同处一级，'*'和'/'同处一级，'^'自成一级，'!'也自成一级。

需特别注意的是，根据代码4.6中的优先级表，任何时刻操作符'!'在操作符栈中只可能存有一个，而且必定是栈顶。对于合法的表达式，此后出现的下一操作符不可能是'('。而无论接下来出现的是何种操作符（即便是'!'本身），该操作符都会随即出栈并执行对应的计算。

综合以上分析，为使操作符栈的规模最大，其中所存的操作符应大致排列表x4.2所示。

表x4.2 （左）括号数固定时，运算符栈的最大规模

0	1	2	3	4	5	6	7	8	9	10	11	12	...	2008	2009	2010	2011	2012
\0	+ -	* /	^	(	+ -	* /	^	(	+ -	* /	^	(	...	(	+ -	* /	^	!
0				1				2				3		502				503

不难看出，此时操作符栈的规模为：

$$(502 + 1) \times 4 + 1 = 2013$$

[4-12] 对异常输入的处置能力是衡量算法性能的重要方面，即教材 1.1.4 节所谓的鲁棒性。为考查教材 95 页代码 4.7 中 evaluate()算法的这一性能，现以非正常的表达式"(12)3+!4*+5"作为其输入。

a） 试给出在算法退出之前，操作数栈和操作符栈的演化过程；

【解答】

evaluate()算法对该“表达式”的求值过程，如表x4.3所示。其中，运算符栈和操作数栈的栈底/栈顶都在左侧/右侧。

表x4.3 非法表达式"(12)3+!4*+5"的“求值”过程

表达式	运算符栈	操作数栈	注解
(1 2) 3 + ! 4 * + 5 $	$		表达式起始标识入栈
[(] 1 2) 3 + ! 4 * + 5 $	$ (		左括号入栈
([1 2]) 3 + ! 4 * + 5 $	$ (	12	操作数12入栈
(1 2 [)] 3 + ! 4 * + 5 $	$	12	左括号出栈
(1 2) [3] + ! 4 * + 5 $	$	12 3	操作数3入栈
(1 2) 3 [+] ! 4 * + 5 $	$ +	12 3	运算符'+'入栈
(1 2) 3 + [!] 4 * + 5 $	$ + !	12 3	运算符'!'入栈
(1 2) 3 + ! [4] * + 5 $	$ + !	12 3 4	操作数4入栈
(1 2) 3 + ! 4 [*] + 5 $	$ +	12 3 24	运算符'!'出栈执行
(1 2) 3 + ! 4 [*] + 5 $	$ + *	12 3 24	运算符'*'入栈
(1 2) 3 + ! 4 * [+] 5 $	$ +	12 72	运算符'*'出栈执行
(1 2) 3 + ! 4 * [+] 5 $	$	84	运算符'+'出栈执行
(1 2) 3 + ! 4 * [+] 5 $	$ +	84	运算符'+'入栈
(1 2) 3 + ! 4 * + [5] $	$	84 5	操作数5入栈
(1 2) 3 + ! 4 * + 5 [$]	$	89	运算符'+'出栈执行
(1 2) 3 + ! 4 * + 5 [$]		89	表达式起始标识出栈
(1 2) 3 + ! 4 * + 5 $			返回唯一的元素89

b） 该算法是否能够正常终止？若异常退出，试解释原因；否则，试给出算法的输出；

【解答】

由表x4.3可见，尽管上述表达式明显不合语法，但evaluate()算法却依然能够顺利求值，并正常退出。实际上此类实例纯属巧合，更多时候该算法在处理非法表达式时都会异常退出。

反观上例也可看出，巧合的原因在于，在该“表达式”的求值过程中，每当需要执行某一运算时，在操作数栈中至少存有足够多操作数可供弹出并参与运算。

c） 试改进该 evaluate()算法，使之能够判别表达式的语法是否正确。

【解答】

就最低的标准而言，改进后的算法应该能够判定表达式是否合法。为此，除了需要检查括号的匹配，以及在每次试图执行运算时核对操作数栈的规模足够大，还需要确认每个操作符与其所对应操作数之间的相对位置关系符合中缀表达式（infix）的语法。最后一项检查的准则并不复杂：在每个操作符即将入栈时，操作数栈的规模应比操作符栈的规模恰好大一。

请读者根据以上提示，独立完成对原算法的改进工作。当然，就此问题的进一步要求是，在判定表达式非法后，还应能够及时报告问题的类型及其所在的位置，甚至给出修正的建议。

[4-13] RPN 表达式无需括号即可确定运算优先级，这是否意味着其所占空间必少于常规表达式？为什么？

【解答】

未必。实际上，尽管RPN表达式可以省去括号，但必须在相邻的操作数、操作符之间插入特定的分隔符（通常为空格）。这种分隔符必须事先约定，且不能用以表示操作数或操作符，故亦称做元字符（meta-character）。

不难看出，RPN表达式所引入元字符的数量，与操作数和操作符的总数相当，故其所占空间总量未必少于原表达式。

[4-14] PostScript 是一种典型的栈式语言，请学习该语言的基本语法，并编写简单的绘图程序。

【解答】

如代码x4.5所示，即为PostScript语言绘图程序的一个实例。

```
%!PS-Adobe-2.0
%
% Smiling faces drawing
%
% Written by: Junhui DENG
% Last update: Mar. 2009
%
/Times-Roman findfont
24 scalefont
setfont
%
```

```
/red        {1 0 0 setrgbcolor} def
/green      {0 1 0 setrgbcolor} def
/blue       {0 0 1 setrgbcolor} def
/yellow     {1 1 0 setrgbcolor} def
/black      {0 0 0 setrgbcolor} def
/white      {1 1 1 setrgbcolor} def
%
/dottedline {0 setlinewidth}  def
/fatline    {16 setlinewidth} def
/thinline   {4 setlinewidth}  def
%
/smile {
   newpath
   gsave
   rotate
   0 translate
   180 div dup scale
   yellow   0 0 180 0 360 arc fill
   red      -55 45 27 0 360 arc fill
   blue     55 45 27 0 360 arc fill
   fatline  white 0 -18 90 210 330 arc stroke
   thinline black 0 0 180 0 360 arc stroke
   grestore
} def
%
gsave
300 400 translate
180 0 0 smile
360 -15 0 {
   dup 6 div
   180
   2 index
   smile
   pop
} for
%
-65 -30 moveto
black (Hello, world!) show
grestore
%
```

代码x4.5 PostScript语言的绘图程序

请读者阅读和运行该段代码，并通过添加注释，完成对其功能的分析。

[4-15] 为判断包含多种括号的表达式是否匹配，可否采用如下策略：

分别检查各种括号是否匹配；若它们分别匹配，则整体匹配

试证明你的结论，或者给出一个反例。

【解答】

在仅有一种括号时，括号匹配的判断准则，可以概括并精简为两条：

a）在表达式任意前缀中，左括号的数量都不少于右括号；

b）整个表达式中，左括号与右括号数量相等。

然而在有多种括号并存时，还需追加一条，以检查不同括号之间的相对位置：

x）在相互匹配的任何一对括号之间，则各种括号都是匹配的。

否则，即便每种括号均各自匹配，但不同括号之间仍可能存在相互“交错”的现象。比如，以下即是一个反例：

([{)] }

[4-16] 在 N 皇后搜索算法（教材 101 页代码 4.9）中，“忒修斯的线绳”与“粉笔”各是通过什么机制实现的？

【解答】

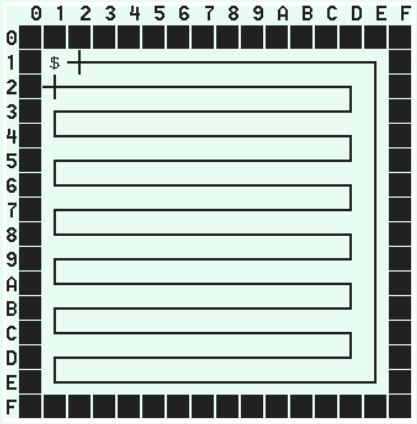
该算法所使用的栈solu就相当于“忒修斯的线绳”：压栈操作等效于前进一步并延长线绳；出栈操作等效于顺着线绳后退一步，同时收缩线绳。

这里通过循环按单调次序逐一检查每个格点，故不致于重复访问——这一机制，即等效于在迷宫中藉以作标记的“粉笔”。

[4-17] 考查如教材 103 页代码 4.13 所示的迷宫寻径算法。

a） 试举例说明，即便 n×n 迷宫内部没有任何障碍格点，且起始与目标格点紧邻，也可能须在搜索过所有共$(n-2)^2$个可用格点之后，才能找出一条长度为$(n-2)^2$的通路；

【解答】



图x4.1 迷宫算法低效的实例

符合上述条件的一个具体实例，如图x4.1所示。

根据该算法的控制流程，在每个格点都是固定地按照东、南、西、北的次序，逐个地试探相邻的格点。因此，尽管在此例中目标终点就紧邻于起始点的西侧，也只有在按照如图所示的路线，遍历过所有的$(n-2)^2$个格点之后，才能抵达终点。

b） 尝试改进该算法，使之访问的格点尽可能少，找出的路径尽可能短。

【解答】

一种简便而行之有效的策略是，每次都是按随机次序试探相邻格点。

为此，需要改写nextESWN()函数（教材102页代码4.10）以及相关的数据结构。

请读者照此提示，独立完成设计、编码和调试任务。

[4-18] Fermat-Lagrange 定理指出，任何一个自然数都可以表示为 4 个整数的平方和，这种表示形式称作费马-拉格朗日分解，比如：30 = $1^2 + 2^2 + 3^2 + 4^2$。

试采用试探回溯策略，实现以下算法：

a） 对任一自然数 n，找出一个费马-拉格朗日分解；

b） 对任一自然数 n，找出所有费马-拉格朗日分解（同一组数的不同排列视作等同）；

c） 对于不超过 n 的每一自然数，给出其费马-拉格朗日分解的总数。

【解答】

在枚举并检查各候选解的过程中，需充分利用费马-拉格朗日分解的性质进行剪枝。

请读者按照这一总体思路，独立完成具体的算法设计，以及编码和调试任务。

[4-19] 暂且约定按照自然优先级，并且不使用括号，考查在数字'0'~'9'间加入加号'+'、乘号'*'后所构成的合法算术表达式。

a） 试编写一个程序，对于任一给定的整数 S，给出所有值为 S 的表达式。比如：

```
100    =   0 + 12 + 34 + 5 * 6 + 7 + 8 + 9
           0 + 12 + 3 * 4 + 5 + 6 + 7 * 8 + 9
           0 + 1 * 2 * 3 + 4 + 5 + 6 + 7 + 8 * 9
           0 * 1 + 2 * 3 + 4 + 5 + 6 + 7 + 8 * 9
           0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 * 9
           0 + 1 * 2 * 3 * 4 + 5 + 6 + 7 * 8 + 9
           0 * 1 + 2 * 3 * 4 + 5 + 6 + 7 * 8 + 9
                  ...
2012   =   0 + 12 + 34 * 56 + 7 + 89
                  ...
```

b） 拓展你的程序，使之同时还支持阶乘'!'。比如

```
2012   =   0 ! + 12 + 3 ! ! + 4 * 5 ! + 6 ! + 7 + 8 * 9
                  ...
```

c） 继续拓展，引入乘方'^'、减法'-'、除法'/'等更多运算；

d） 再做拓展，引入括号。

【解答】

在枚举并检查各候选解的过程中，需充分利用各种运算符（以及括号）的性质进行剪枝。

请读者按照这一总体思路，独立完成算法设计，以及编码和调试任务。

[4-20] 试从以下方面改进 4.6.2 节的银行服务模拟程序，使之更加符合真实情况：

a） 顾客按更复杂的随机规律抵达；

b） 新顾客选择队列时不以人数，而以其服务时长总和为依据。

【解答】

请读者独立完成编码和调试任务。

[4-21] 自学 C++ STL 中 stack 容器和 queue 容器的使用方法，阅读对应的源代码。

【解答】

请读者独立完成阅读任务。

[4-22] 双端队列（deque[①]）是常规队列的扩展。顾名思义，该结构允许在其逻辑上的两端实施数据操作。具体地，与队头（front）端和队尾（rear）端相对应地，插入和删除操作各设有两个接口：

```
template <typename T> class Deque {
public:
   T& front(); //读取首元素
   T& rear(); //读取末元素
   void insertFront(T const& e); //将元素e插至队列前端
   void insertRear(T const& e); //将元素e插至队列末端
   T removeFront(); //删除队列的首元素
   T removeRear(); //删除队列的末元素
};
```

a） 实现如上定义的双向队列结构；

【解答】

比如，可以仿照代码x4.1中基于List结构派生出Stack结构，以及代码4.14（教材106页）中基于List结构派生出Queue结构的技巧。

具体地，将List结构的两端与Deque结构的两端相对应，相应的操作即可自然地转化为List结构在头、尾的插入和删除操作。

b） 你所实现的这些接口，时间复杂度各为多少？

【解答】

采用以上策略实现的Deque结构，各操作接口分别对应于List结构在首、末两端的插入或删除操作接口。根据3.3节对List结构性能的分析结论，得益于其内部的头、尾哨兵节点，这些操作均可在$O(1)$时间内完成。

[①] deque原本的读音与队列ADT的dequeue接口雷同，都是[di:kju]，为示区别，通常将deque改读作[dek]

[4-23] 从队列的角度回顾二路归并算法的两个版本，不难发现，无论 Vector::merge()（教材 63 页代码 2.29）还是 List::merge()（教材 82 页代码 3.22），所用到的操作无非两类：

从两个输入序列的前端删除元素
将元素插入至输出序列的后端

因此，若使用队列 ADT 接口来描述和实现该算法的过程，必将既简洁且深刻。
试按照这一理解，编写二路归并算法的另一版本，实现任意一对有序队列的归并。

【解答】

可以将Vector或List结构的首、末两端，与Queue结构的首、末两端相对应，并约定队列中的各元素从队首至队末，按单调非降次序排列。于是，对待归并序列（队列）的操作，仅限于调用front()接口（取各序列的首元素并比较大小）和dequeue()接口（摘出首元素中的小者）；而对合成序列（队列）的操作，仅限于调用enqueue()接口（将摘出的元素归入序列）。

请读者按照以上介绍和提示，独立完成编码和调试任务。

实际上，使用栈的ADT接口，也可简洁地描述和实现归并排序算法。当然，每次归并之后，还需随即对合成的序列（栈）做一次倒置操作reverse()（参见习题[4-25]）。

[4-24] 基于向量模板类 Vector 实现栈结构时，为了进一步提高空间的利用率，可以考虑在一个向量内同时维护两个栈。它们分别以向量的首、末元素为栈底，并相向生长。

为此，对外的入栈和出栈操作接口都需要增加一个标志位，用一个比特来区分实施操作的栈。
具体地，入栈接口形式为 push(0, e)和 push(1, e)，出栈接口形式为 pop(0)和 pop(1)。

a）试按照上述思路，实现这种孪生栈结构；

【解答】

请读者参照以上提示，独立完成。

b）当孪生栈的规模之和已达到向量当前的容量时，为保证后续的入栈操作可行，应采取什么措施？

【解答】

扩容。仿照2.4.3节的策略和方法，同样可以实现分摊意义上的高效。

[4-25] 试设计并实现 Stack::reverse()接口，将栈中元素的次序前后倒置。

【解答】

一种可行的方法是：先将栈中的元素逐一取出并依次插入某一辅助队列，然后再逐一取出队列中的元素并依次插回原栈。请读者根据以上介绍和提示，独立完成编码和调试任务。

[4-26] 试设计并实现 Queue::reverse()接口，将队列中元素的次序前后倒置。

【解答】

仿照上题的方法：先将队列中的元素逐一取出并依次插入某一辅助栈，然后再逐一取出栈中的元素并依次插回原队列。请读者根据以上介绍和提示，独立完成编码和调试任务。