STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

3.5 A Mazing Problem

3.6 Evaluation of Expressions

- Maze: Representation
  - Using a 2D array, maze[m][p]

entrance →

왼쪽 최상단

0: open path

1: blocked path

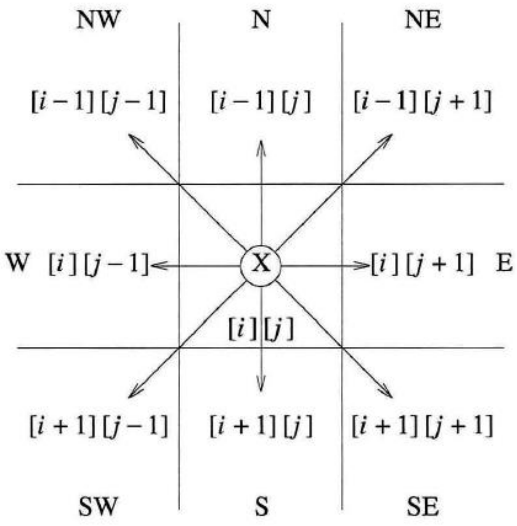
0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

→ exit

오른쪽 최하단

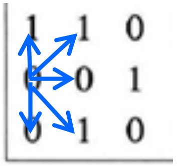
Figure 3.8: An example maze (can you find a path?)

- Current location X: **maze[i][j]**
  - Possible 8 moves

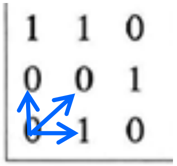


0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

  - But, not every position has eight neighbors



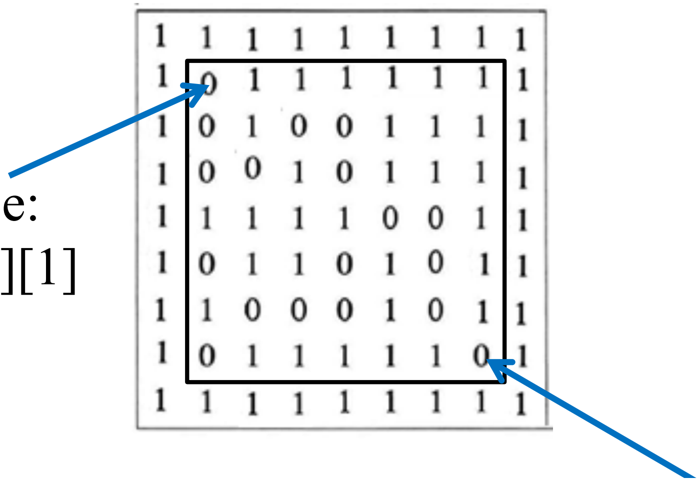
1	1	0
0	0	1
0	1	0



1	1	0
0	0	1
0	1	0

체크할 숫자가 다름

- m × p maze → (m+2)×(p+2) array
  - To avoid checking for the border conditions



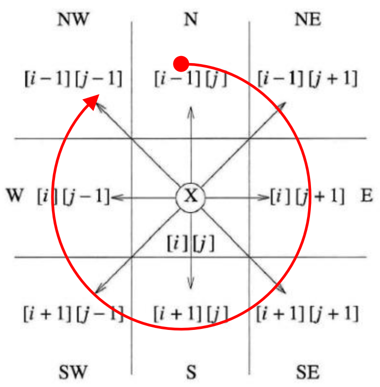
1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	1
1	0	0	1	0	1	1	1	1
1	1	1	1	1	0	0	1	1
1	0	1	1	0	1	0	1	1
1	1	0	0	0	1	0	1	1
1	0	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1

entrance:
maze[1][1]

exit:
maze[*m*][*p*]

- Possible directions: 1D array, ***move***

```
typedef struct  {
          short int vert;
          short int horiz;
} offsets;
offsets move[8];   /* array of moves for each direction  */
```



수직 수평

Name	Dir	*move[dir].vert*	*move[dir].horiz*
N	0	−1	0
NE	1	−1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	−1
W	6	0	−1
NW	7	−1	−1

- Current: **maze[row][col]** → Next: **maze[nextRow][nextCol]**
  - nextRow = row + move[dir].vert;
  - nextCol = col + move[dir].horiz;

- Records maze positions already checked: **2D array**, **mark**
  - ex) when visiting a position, maze[row][col]
    → mark[row][col] = 1

이미 방문했던 곳을 기록!
(= 길이 막혔을때 다시 돌아가기 위함)

maze

entrance [1,1]

0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

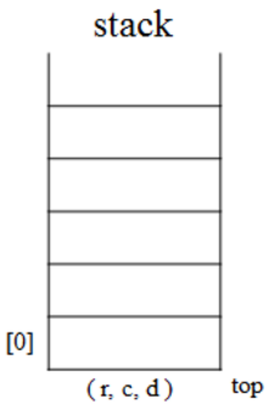
exit [5,4]

mark

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

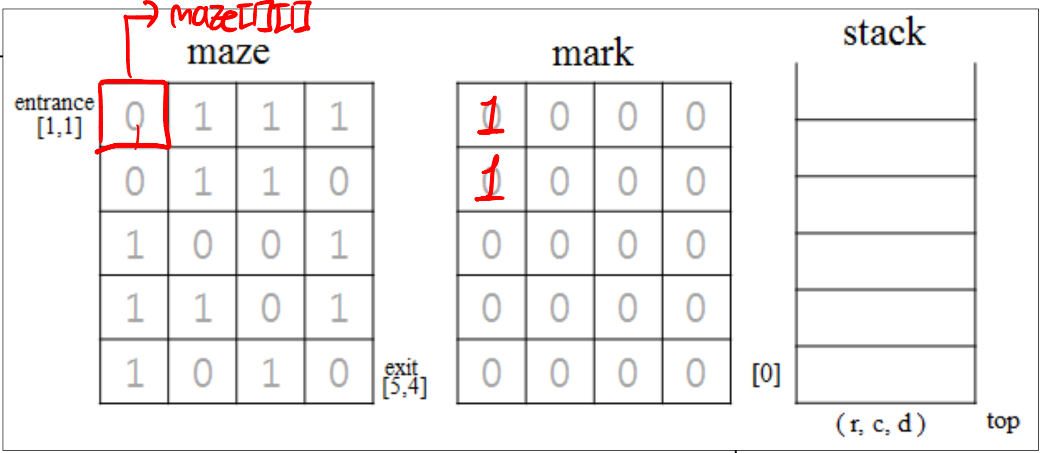
- Keeps pass history: **Stack**

```
#define MAX_STACK_SIZE 100
typedef struct {
            short int row;
            short int col;
            short int dir;
} element;
element  stack[MAX_STACK_SIZE];
```



```
initiallize a stack to the maze's entrance coordinates and direction to north;
while (stack is not empty) {
  /* move to position at top of stack */
  <rol, col, dir> = pop from top of stack;              처리해야할 방향이 더 있을때까지
   while (there are more moves from current position) {
      <nextRow, nextCol> = coordinate of next move;
      dir = direction of move;
      if ((nextRow == EXIT_ROW) && (nextCol == EXIT_COL))
        success;
      if (maze[nextRow][nextCol] == 0) && mark[nextRow][nextCol] == 0) {
      /* legal move and haven't been there */
        mark[nextRow][nextCol] = 1;
        /* save current position and direction */
        push<row, col, dir> to the top of the stack;
        row = nextRow;
        col = nextCol;
        dir = north;
      }
   }
}
printf("No path found\n");
```

Program 3.11: Initial maze algorithm



maze			
0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

mark			
1	0	0	0
1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

```
void path(void)   {
  int i, row, col, nextRow, nextCol, dir;
  int found=FALSE;
  element position;
  mark[1][1]=1;   top=0;
  stack[0].row=1;   stack[0].col=1;
  stack[0].dir=1;
  while (top>-1 && !found) {
    position = pop();
    row = position.row;      col = position.col;     dir = position.dir;
    while (dir<8 && !found) {
         /* move in direction dir */
         nextRow = row + move[dir].vert;   nextCol = col + move[dir].horiz;

         if (nextRow==EXIT_ROW && nextCol==EXIT_COL)
              found = TRUE;
         else if ( !maze[nextRow][nextCol] && !mark[nextRow][nextCol]) {
              mark[nextRow][nextCol]) = 1;
              position.row = row;   position.col = col;    position.dir = ++dir;
              push(position);
              row = nextRow;   col = nextCol;   dir = 0;
         }
         else ++dir;
    }
  }
```

initialize

initialize

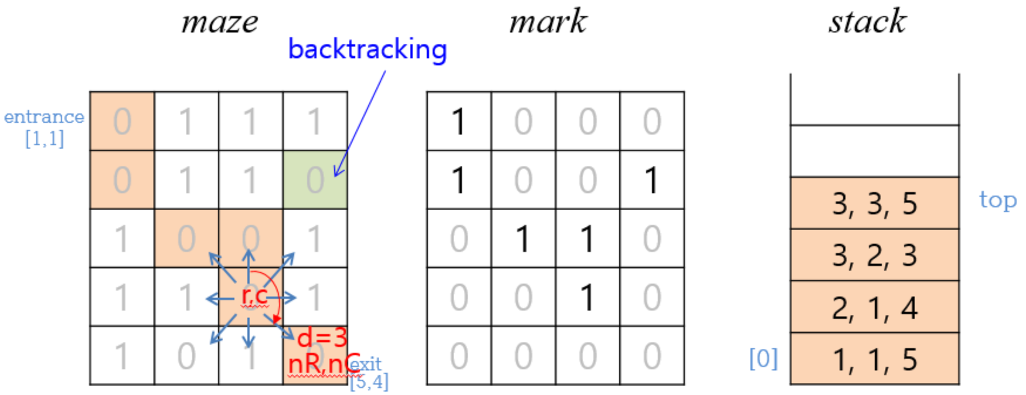
① 2.1.5가 아니고 1.1.5인 이유! 이전의 위치정보를 저장하고,

② row와 col을 next로 바꿔줌

Program 3.12: Maze search function

```
  if ( found ) {
      printf("The path is:\n");
      printf("row  col\n");
      for (i=0; i<=top; i++)
              printf("%2d%5d", stack[i].row, stack[i].col");
      printf("%2d%5d\n", row, col);
      printf("%2d%5d\n", EXIT_ROW, EXIT_COL);
   }
   else printf("The maze does not have a path\n");
}
```

Program 3.12: Maze search function

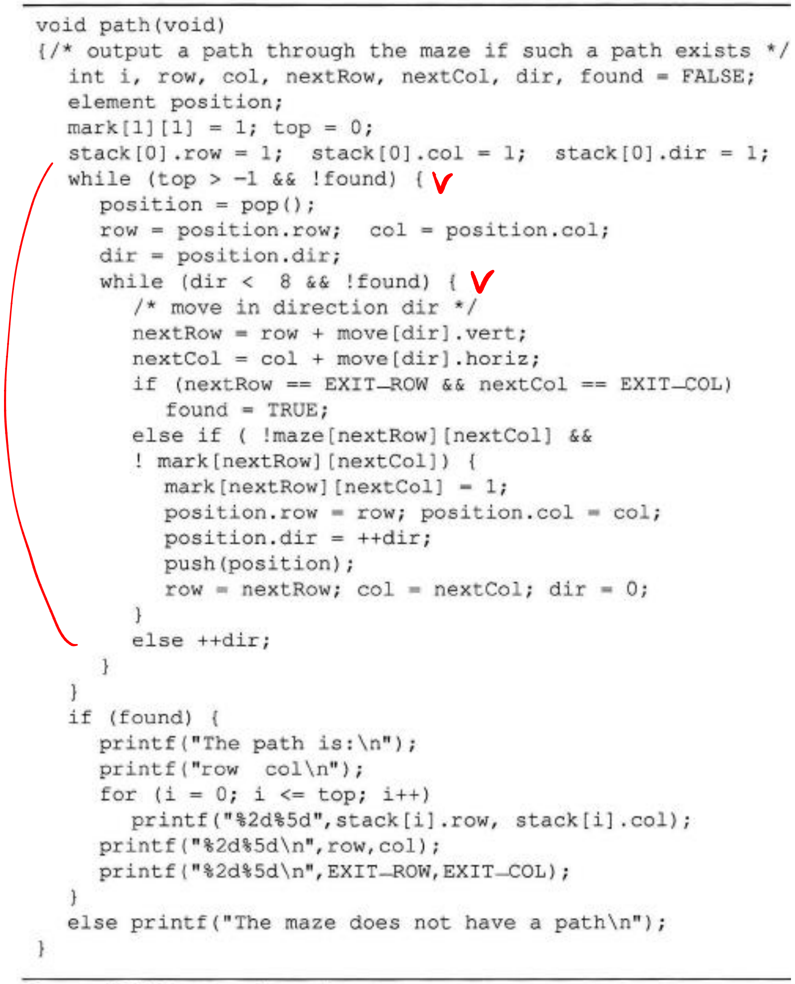


0	1	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	0	1	0

1	0	0	0
1	0	0	1
0	1	1	0
0	0	1	0
0	0	0	0

3, 3, 5
3, 2, 3
2, 1, 4
1, 1, 5

- Analysis of path
  - Computing Time: O(…) ⇒ O(mxp)
    - Each position within the maze is visited no more than once



```
void path(void)
{/* output a path through the maze if such a path exists */
   int i, row, col, nextRow, nextCol, dir, found = FALSE;
   element position;
   mark[1][1] = 1; top = 0;
   stack[0].row = 1;  stack[0].col = 1;  stack[0].dir = 1;
   while (top > -1 && !found) { ✓
      position = pop();
      row = position.row;  col = position.col;
      dir = position.dir;
      while (dir <  8 && !found) { ✓
         /* move in direction dir */
         nextRow = row + move[dir].vert;
         nextCol = col + move[dir].horiz;
         if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
            found = TRUE;
         else if ( !maze[nextRow][nextCol] &&
         ! mark[nextRow][nextCol]) {
            mark[nextRow][nextCol] = 1;
            position.row = row; position.col = col;
            position.dir = ++dir;
            push(position);
            row = nextRow; col = nextCol; dir = 0;
         }
         else ++dir;
      }
   }
   if (found) {
      printf("The path is:\n");
      printf("row  col\n");
      for (i = 0; i <= top; i++)
         printf("%2d%5d",stack[i].row, stack[i].col);
      printf("%2d%5d\n",row,col);
      printf("%2d%5d\n",EXIT_ROW,EXIT_COL);
   }
   else printf("The maze does not have a path\n");
}
```

Program 3.12: Maze search function

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

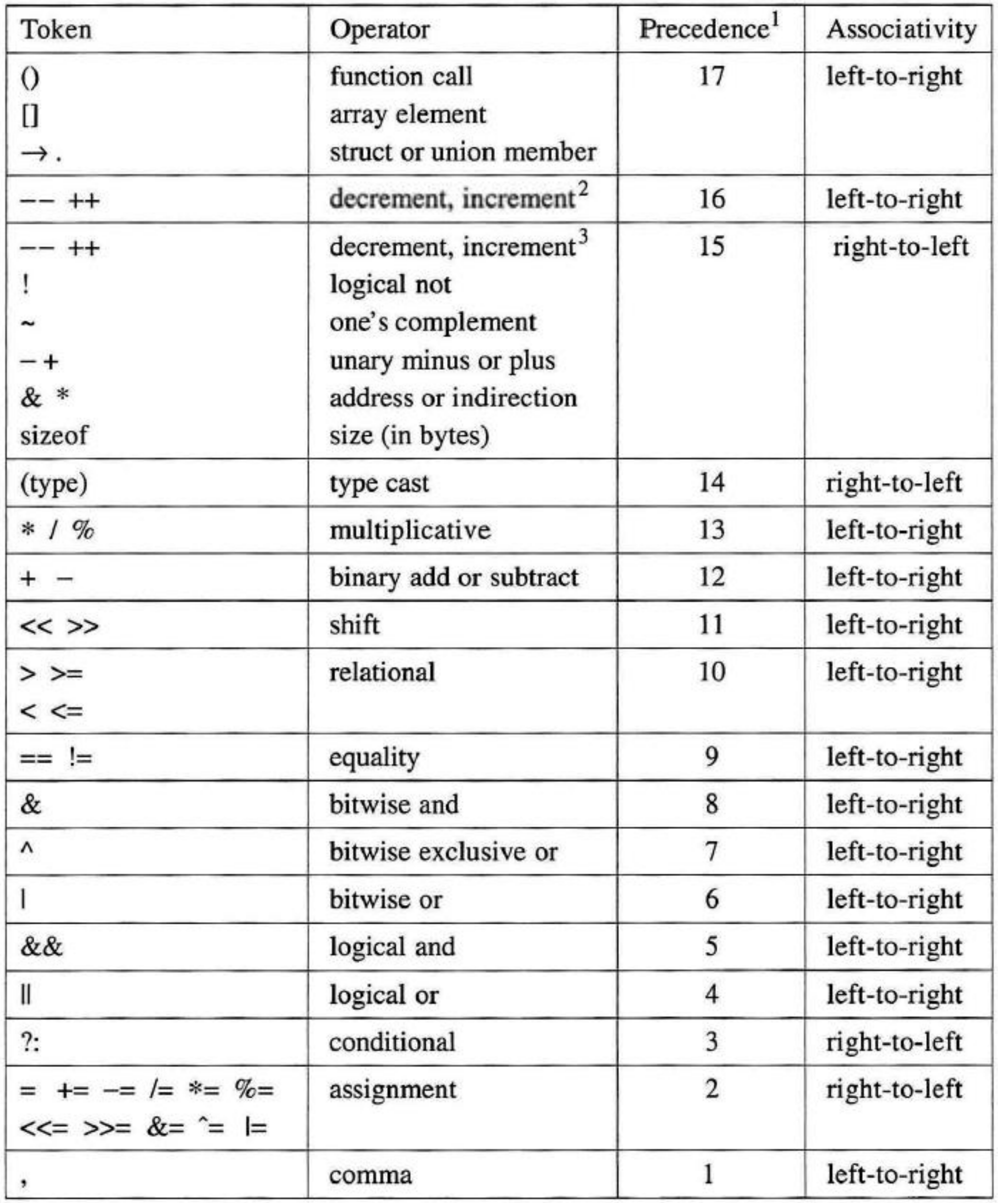
3.5 A Mazing Problem

3.6 Evaluation of Expressions

3.6.1 Expressions

- Expression statements
  - e.g)
    - ((rear+1==front)||((rear==MAX_QUEUE_SIZE-1) &&!front))
    - x = a / b – c + d * e – a * c
  - Contain operators, operands, and parentheses

- Understanding the meaning
  - Figure out the *order* in which the operations are performed
  - e.g. If a = 4, b = c = 2, d = e = 3
    - x = ((a/b)-c)+(d*e)-(a*c) = ((4/2)-2)+(3*3)-(4*2) = 1 (O)
    - x = (a/(b-c+d))*(e-a)*c = (4/(2-2+3))*(3-4)*2 = -2.66666… (X)



Token	Operator	Precedence[1]	Associativity
() [] → .	function call array element struct or union member	17	left-to-right
-- ++	decrement, increment[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
|	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
||	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ˆ= |=	assignment	2	right-to-left
,	comma	1	left-to-right

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

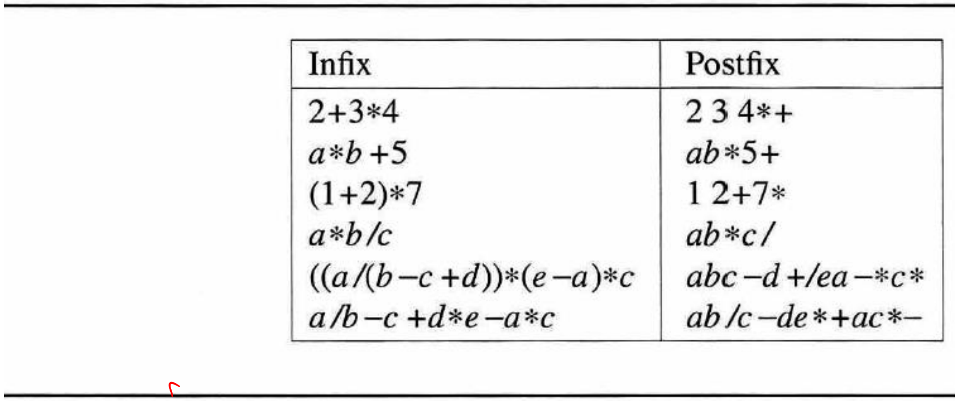
Figure 3.12: Precedence hierarchy for C

3.6.2 Evaluating Postfix Expressions

- Infix notation 중위표기법
  - The standard way of writing expressions
  - Placed a binary operator in-between its two operands
  - Not used by compilers to evaluate expressions

후위 표기법

- Postfix notaton → 컴퓨터는 이렇게 인식함
  - Parenthesis-free notation
  - Each operator appears after its operands



Infix	Postfix
2+3*4	2 3 4*+
*a***b* +5	*ab**5+
(1+2)*7	1 2+7*
*a***b*/*c*	*ab***c*/
((*a*/(*b*−*c* +*d*))*(*e*−*a*)**c*	*abc*−*d* +/*ea*−**c**
a/*b*−*c* +*d***e*−*a***c*	*ab*/*c*−*de**+*ac**−

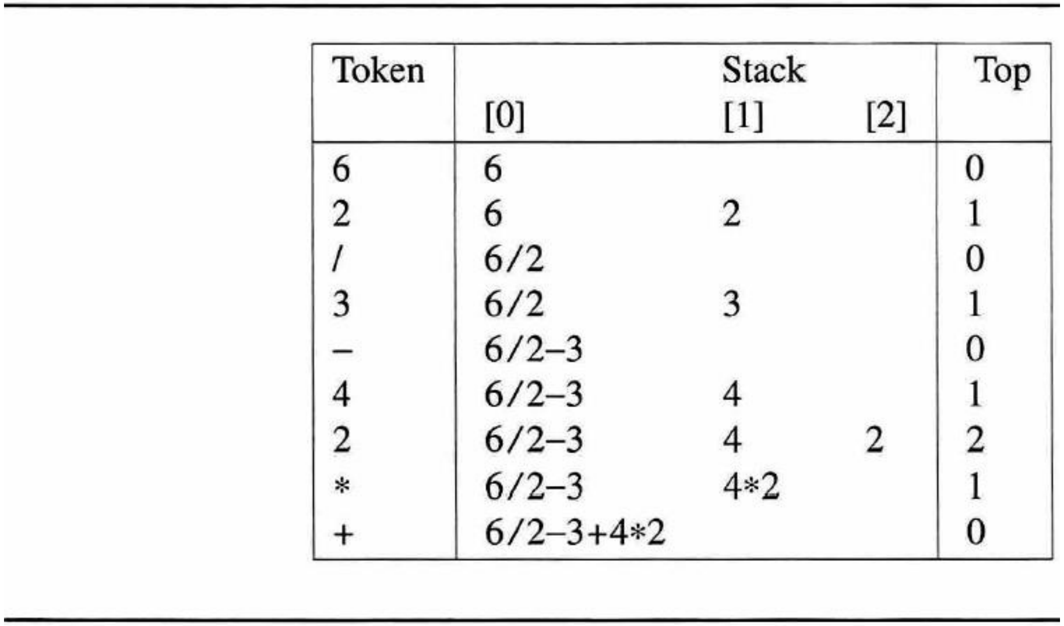
Figure 3.13: Infix and postfix notation

- Evaluating postfix expressions: p.129
  - Input string: 6 2/3-4 2*+

Postfix → infix

To evaluate an expression :

1) make a single left-to-right scan of it
2) place the operands on a stack until we find an operator
3) remove, from the stack, the correct number of operands for the operator
4) perform the operation, and place the result back on the stack
5) continue in this fashion until we reach the end of the expression.
6) We then remove the answer from the top of the stack



Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
−	6/2−3			0
4	6/2−3	4		1
2	6/2−3	4	2	2
*	6/2−3	4*2		1
+	6/2−3+4*2			0

Figure 3.14: Postfix evaluation

- Representation : stack, expression
  - Assumptions
    - Operators: +, -, *, /, %
    - Operands: single digit integer

```
#define MAX_STACK_SIZE 100
#define MAX_EXPR_SIZE 100

int stack[MAX_STACK_SIZE];    /* global stack */
char expr[MAX_EXPR_SIZE];     /* global input string
                                 (a postfix expression) */
```

expr: 6 2/3-4 2*+

↑ 숫자 6이 아니고 정수 6

```
token = getToken( &symbol, &n );
```

expr: 6 2/3-4 2*+

```
precedence getToken(char *symbol, int *n)
{
      *symbol = expr[(*n)++];
                          *n = *n+1

      switch ( *symbol ) {
         case '(' :   return lparen;
         case ')' :   return rparen;
         case '+' :  return plus;
         case '-' :   return minus;
         case '/' :   return divide;
         case '*' :  return times;
         case '%' : return mod;
         case '\0' : return eos;
         default  : return operand;   /* no error checking,
                                            default is operand */
      }
}
```

```
typedef enum {lparan, rparen, plus,
               minus, times, divide,
               mod, eos, operand
} precedence;
```

Program 3.14: Function to get a token from the input string

```
int eval(void)  {
        precedence token;
        char symbol;
        int op1,op2;
        int n = 0;    /* counter for the expression string */
        int top = -1;
        token = getToken(&symbol, &n);
        while ( token != eos ) {
          if (token == operand)
               push( symbol – '0' );  /* convert: char → integer */
          else {
               op2 = pop();      op1 = pop();
               switch(token) {
                     case plus:     push(op1+op2); break;
                     case minus: push(op1-op2); break;
                     case times:  push(op1*op2); break;
                     case divide: push(op1/op2); break;
                     case mod:    push(op1%op2);
               }
          }
          token = getToken(&symbol, &n);
        }
        return pop();   /* return result */
}
```

```
typedef enum {lparan, rparen, plus,
              minus, times, divide,
              mod, eos, operand
} precedence;
```

expr: 6 2/3-4 2*+

→ return이 조건

token이 피연산자이면.. (정수들)

↳ 아스키코드!

↳ 스택에서 먼저꺼낸 피연산자가 2번째 피연산자가 됨 ☆

Input이 Postfix (후위표기법)

Program 3.13: Function to evaluate a postfix expression

3.6.3 Infix to Postfix

- Algorithm

  (1) Fully parenthesize the expression

  (2) Move all binary operators so that they replace their corresponding right parentheses

  (3) Delete all parentheses 괄호를 모두 없앨 수 있음

e.g) a/b-c+d*e-a*c

(1) ((((a/b)-c) + (d*e)) - (a*c))

(2) ((((a b / c-   (de* +   (ac*-

(3)   a b / c-   de* +   ac*-

a*(b +c)*d

→ abc+*d*

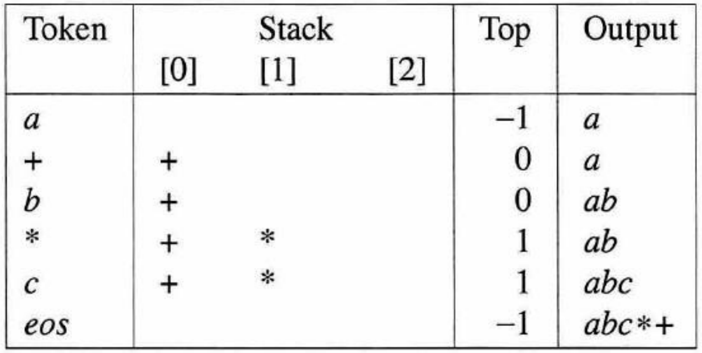
- Note
  - The order of operands is the same in infix and postfix;

    a/b-c+d*e-a*c
→ ab/c-de*+ac*-

  - We can form the postfix equivalent the infix expression by scanning left-to-right

"우선순위가 높은 연산자는 우선순위가 낮은 연산자 위에 올라서서 먼저 자리를 잡지 못하게 한다."

- Ex 3.3 [Simple expression]: a +b*c (=우선순위가 높으면 stock!)
  - Operands: Passed to the output immediately
  - Operators: Stacked if ICP > ISP,
    Unstacked if not
    (ICP: incoming precedence, ISP: in-stack precedence)
  - Unstacking occurs only when we reach *eos*



Token	Stack [0]	[1]	[2]	Top	Output
a				−1	*a*
+	+			0	*a*
b	+			0	*ab*
*	+	*		1	*ab*
c	+	*		1	*abc*
eos				−1	*abc**+

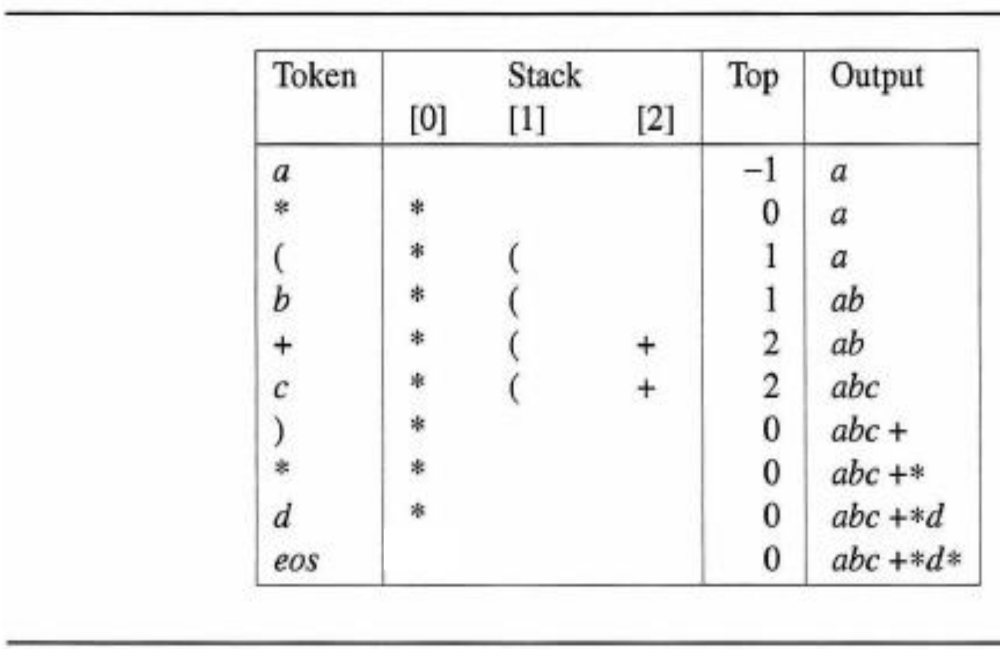
Figure 3.15: Translation of *a* +*b***c* to postfix

- a/b-c+d*e-a*c → …

ab/c-de*+ac*-

괄호가 있는경우

- Ex 3.4 [Parenthesized expression]:
  a * (b + c) * d



Token	Stack			Top	Output
	[0]	[1]	[2]		
a				−1	*a*
*	*			0	*a*
(	*	(		1	*a*
b	*	(		1	*ab*
+	*	(	+	2	*ab*
c	*	(	+	2	*abc*
)	*			0	*abc* +
*	*			0	*abc* +*
d	*			0	*abc* +**d*
eos				0	*abc* +**d**

Figure 3.16: Translation of *a**(*b* +*c*)**d* to postfix

left parenthesis:
It behaves like a **low-precedence** operator when it is on the stack;
We stack operators until we reach the right parenthesis;

e/(f+a*d)+c

→ …

/

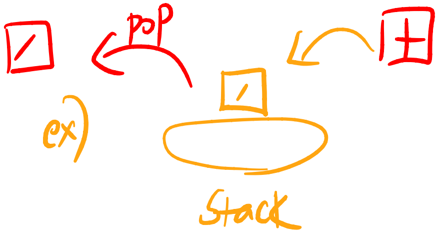
stack efad*+/c+

- Implementation
  - Uses two types of precedence
    - in-stack precedence (isp), incoming precedence (icp)

```
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos * /
int isp[] = {  0,19,12,12,13,13,13,0};
int icp[] = {20,19,12,12,13,13,13,0};
```

  - '(' has low isp, and high icp → 들어올때는 다 눌러버림
  - A operator is **removed** from the stack
    only if ICP <= **ISP**

stack에 있는 operator의 우선순위가 높을때, 튀어나옴



expr: e/(f+a*d)+c

```
void postfix(void) {
      char symbol;  precedence token;
      int n = 0;
      int top = 0;  /* place eos on stack */
      stack[0] = eos;
      for ( token=getToken(&symbol,&n); token!=eos; token=getToken(&symbol,&n) ) {
            if ( token == operand )
                  printf("%c", symbol);
            else if ( token == rparen ) {
                  /* unstack tokens until left parenthesis */
                  while ( stack[top] != lparen )
                     printToken( pop() );
                  pop();   /* discard the left parenthesis */
            }
            else  {
                /* remove and print symbols whose isp is greater
                    than or equal to the current token's icp */
                while ( isp[ stack[top] ] >= icp[ token ] )
                   printToken( pop() );
                push( token );
            }
      }
      while ( (token=pop()) != eos )  printToken( token );
      printf("\n");
}
```

→ 피연산자이면 stack에 넣지않음

→) 이면 (////) 이 사이 연산자들을 pop()해줌

→ 일반적인경우, stack에 있는 연산자가 우선순위가 높으면 pop()해버림.

→ 남아있는 연산자들을 모두 pop()하고 끝냄

Program 3.15: Function to convert from infix to postfix

```
typedef enum{1paran, rparen, plus, minus, times, divide, mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE];   /* global stack */
char expr[MAX_EXPR_SIZE];    /* global input string  */
```

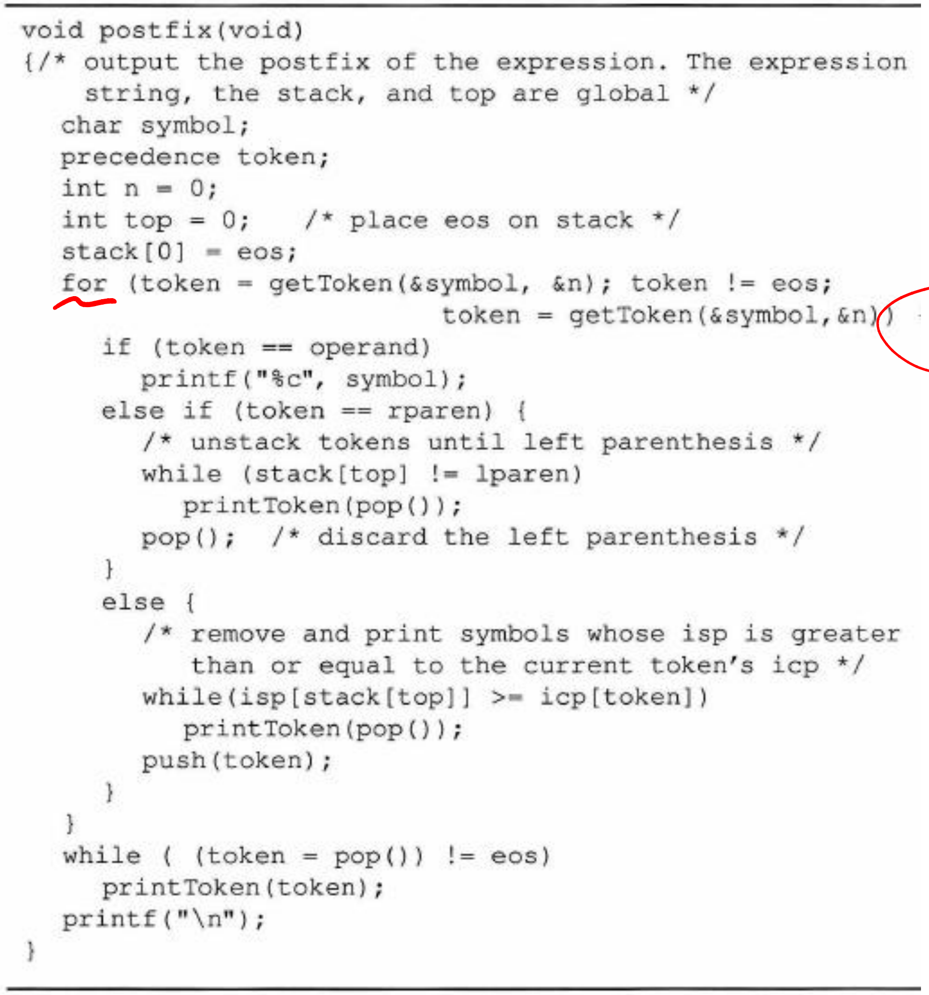
```
precedence getToken(char *symbol, int *n)
{
     *symbol = expr[(*n)++];

    switch (*symbol) {
      case '(' :   return lparen;
      case ')' :   return rparen;
      case '+' :  return plus;
      case '-' :   return minus;
      case '/' :   return divide;
      case '*' :  return times;
      case '%' : return mod;
      case '\0' : return eos;
       default  : return operand;
    }
}
```

Program 3.14: Function to get a token from the input string

- Analysis of postfix
  - *n:* the number of tokens in the expression
  - Time complexity: …

expr: e/(f+a*d)+c



```
void postfix(void)
{/* output the postfix of the expression. The expression
    string, the stack, and top are global */
   char symbol;
   precedence token;
   int n = 0;
   int top = 0;    /* place eos on stack */
   stack[0] = eos;
   for (token = getToken(&symbol, &n); token != eos;
                              token = getToken(&symbol,&n))
      if (token == operand)
         printf("%c", symbol);
      else if (token == rparen) {
         /* unstack tokens until left parenthesis */
         while (stack[top] != lparen)
            printToken(pop());
         pop();  /* discard the left parenthesis */
      }
      else {
         /* remove and print symbols whose isp is greater
            than or equal to the current token's icp */
         while(isp[stack[top]] >= icp[token])
            printToken(pop());
         push(token);
      }
   }
   while ( (token = pop()) != eos)
      printToken(token);
   printf("\n");
}
```

세타 n

$\Theta(n)$

$\Uparrow$

n번 돌기때문

$\exists$ lower bound and $\exists$ upper bound

STACKS AND QUEUES

3.1 Stacks

3.2 Stacks Using Dynamic Arrays

3.3 Queues

3.4 Circular Queues Using Dynamic Arrays

3.5 A Mazing Problem

3.6 Evaluation of Expressions