순천대학교
SUNCHON
NATIONAL UNIVERSITY

Chapter 04. 스택(Stack)

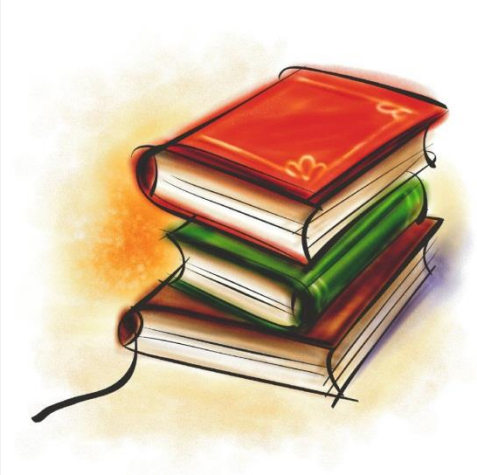
Chapter 04. 스택(Stack)

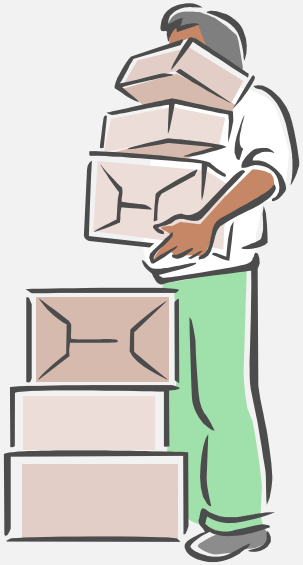
Chapter 04-1:

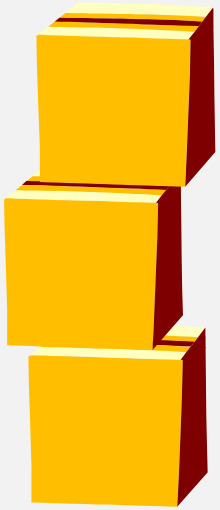
스택의 이해와 ADT 정의

스택(Stack)

- 스택(stack): 쌓아놓은 더미

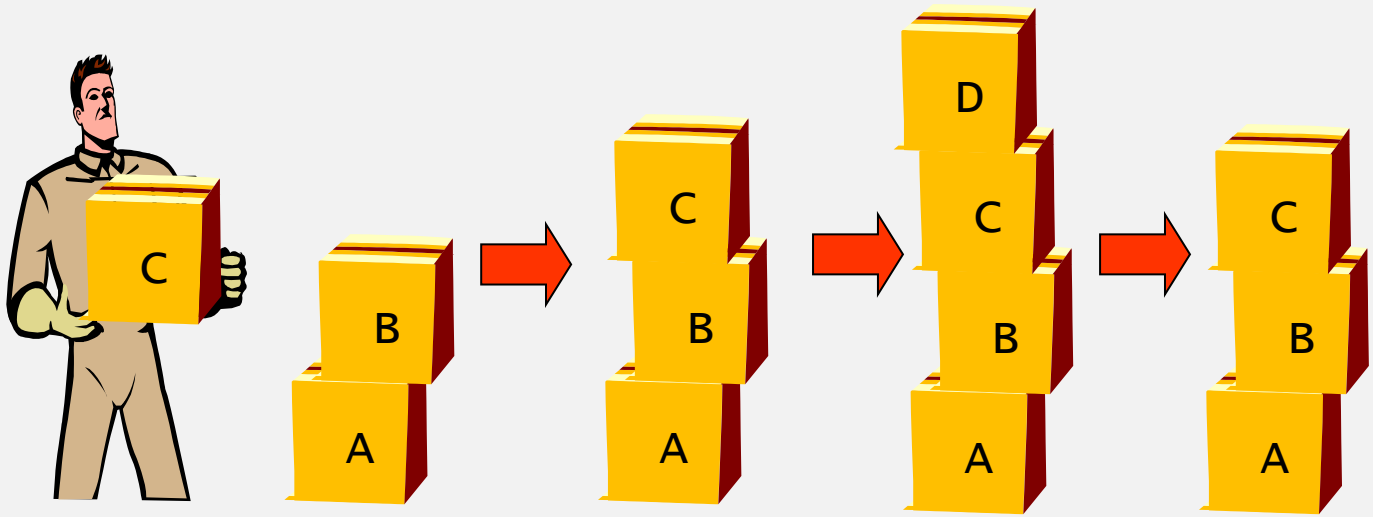


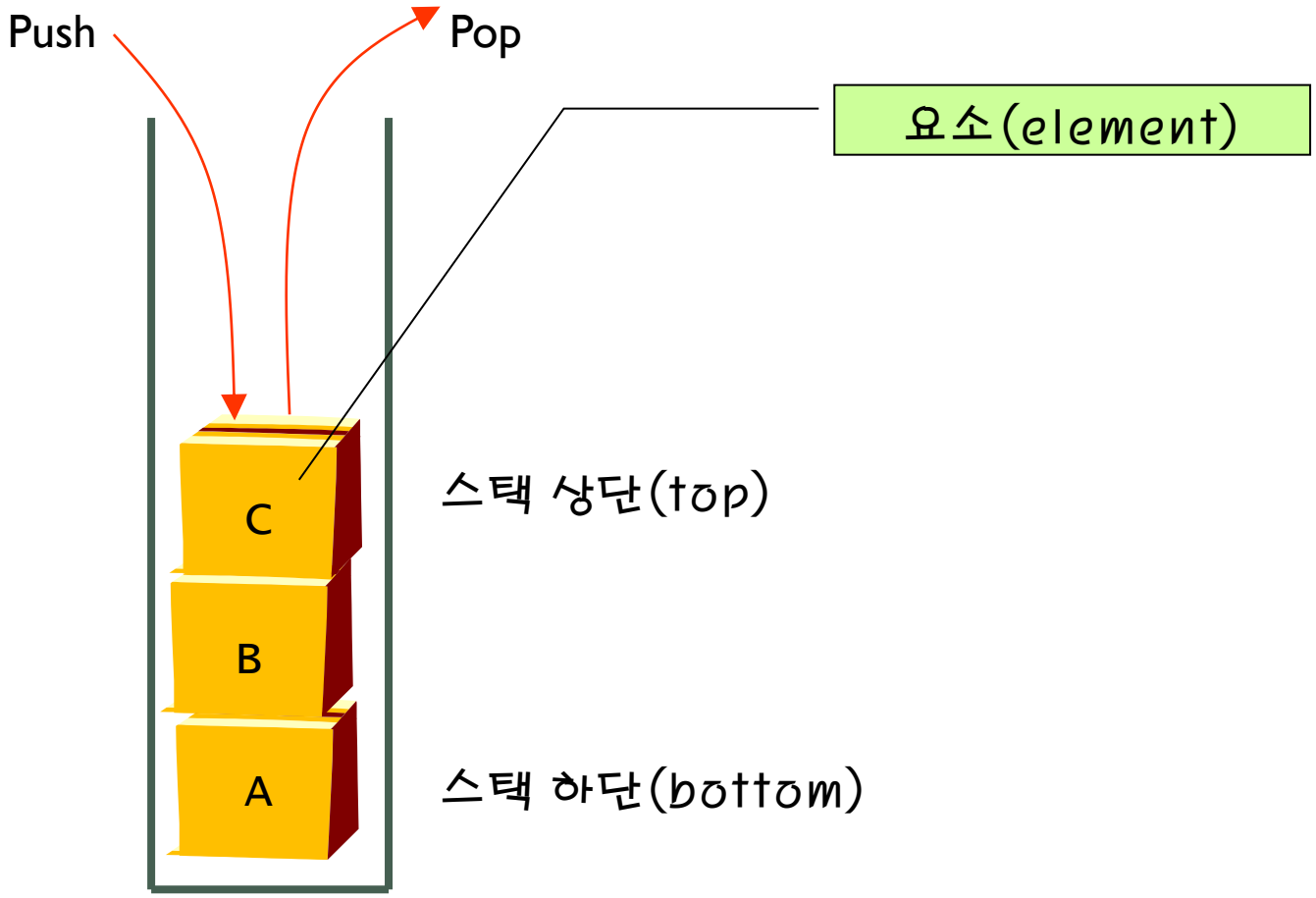




스택의 특징

- **후입선출(LIFO:Last-In First-Out)**: 가장 최근에 들어온 데이터가 가장 먼저 나감.




Push
Pop
요소(element)
C
B
A
스택 상단(top)
스택 하단(bottom)

스택(Stack)의 이해

- 스택은 '먼저 들어간 것이 나중에 나오는 자료구조'로써 초코볼이 담겨있는 통에 비유할 수 있다.
- 스택은 'LIFO(Last-in, First-out) 구조'의 자료구조이다.

- 초코볼 통에 초코볼을 넣는다. — Push
- 초코볼 통에서 초코볼을 꺼낸다. (삭제한다) — pop
- 이번에 꺼낼 초코볼의 색이 무엇인지 통 안을 들여다 본다. — peek

} 스택의 기본 연산

일반적인 자료구조의 학습에서 스택의 이해와 구현은 어렵지 않다. 오히려 활용의 측면에서 생각할 것들이 많다!, 쟁반구조…

스택의 ADT 정의

- `void StackInit(Stack * pstack);`
  - 스택의 초기화를 진행한다.
  - 스택 생성 후 제일 먼저 호출되어야 하는 함수이다.

ADT를 대상으로 배열 기반의 스택
또는 연결 리스트 기반의 스택을 구현할 수 있다.

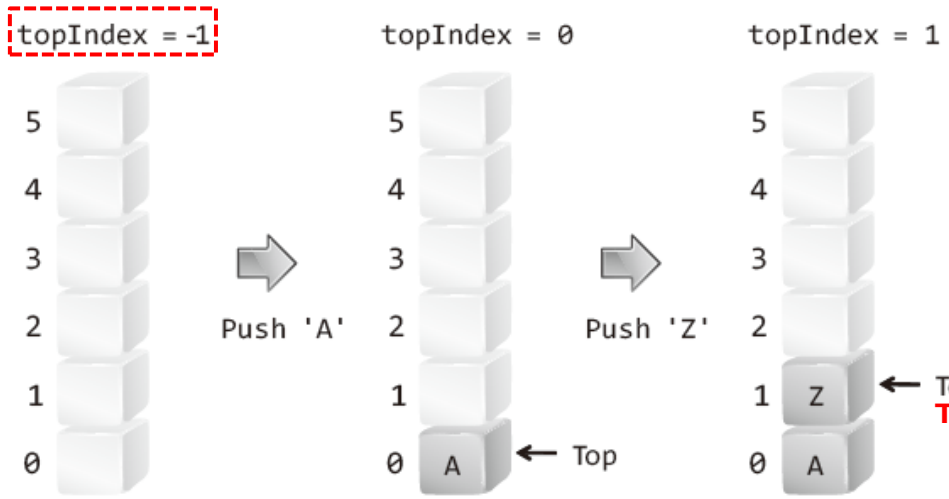
- `int SIsEmpty(Stack * pstack);`
  - 스택이 빈 경우 TRUE(1)을, 그렇지 않은 경우 FALSE(0)을 반환한다.
    - 현재 POP했을 대 얻을 데이터가 있는가 없는가?
- `void SPush(Stack * pstack, Data data);` PUSH 연산
  - 스택에 데이터를 저장한다. 매개변수 data로 전달된 값을 저장한다.
- `Data SPop(Stack * pstack);` POP 연산
  - 마지막에 저장된 요소를 삭제한다.
  - 삭제된 데이터는 반환이 된다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.
    -"Data의 존재 여부를 판단하는 구문은 포함하지 않는다."
- `Data SPeek(Stack * pstack);` PEEK 연산
  - 마지막에 저장된 요소를 반환하되 삭제하지 않는다.
  - 본 함수의 호출을 위해서는 데이터가 하나 이상 존재함이 보장되어야 한다.

Chapter 04-2:

스택의 배열 기반 구현

구현의 논리

두 번의 PUSH 연산



▶ [그림 06-1: 배열 기반 스택의 push 연산]

인덱스가 0인 위치를 스택의 바닥으로 정의해야 배열 길이에 상관없이 바닥의 인덱스 값이 동일해진다.

Top이 유지 되어야 한다.

· 인덱스 0의 배열 요소가 '스택의 바닥으로 정의되었다.

· 마지막에 저장된 데이터의 위치를 기억해야 한다. - Top

· push	Top을 위로 한 칸 올리고, Top이 가리키는 위치에 데이터 저장
· pop	Top이 가리키는 데이터를 반환(빼고)하고, Top을 아래로 한 칸 내림

스택의 헤더파일

```
#define TRUE        1
#define FALSE       0
#define STACK_LEN   100

typedef int Data;

typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;

typedef ArrayStack Stack;

void StackInit(Stack * pstack);                 // 스택의 초기화
int SIsEmpty(Stack * pstack);                   // 스택이 비었는지 확인
                                                  // 비어 있으면 (topIndex=-1)이면 True...
void SPush(Stack * pstack, Data data);          // 스택의 push 연산
Data SPop(Stack * pstack);                      // 스택의 pop 연산
Data SPeek(Stack * pstack);                     // 스택의 peek 연산
```

배열 기반을 고려하여 정의된 스택의 구조체!

배열 기반 스택의 구현: 초기화 및 기타 함수

```
typedef struct _arrayStack
{
    Data stackArr[STACK_LEN];
    int topIndex;
} ArrayStack;
```

```
void StackInit(Stack * pstack)
{
    pstack->topIndex = -1;
}
```

-1은 스택이 비었음을 의미

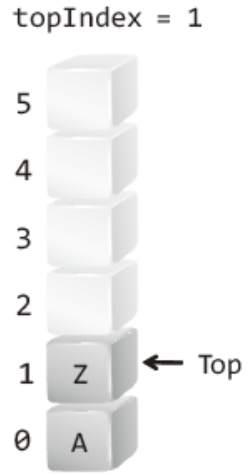
```
int SIsEmpty(Stack * pstack)
{
    if(pstack->topIndex == -1)
        return TRUE;
    else
        return FALSE;
}
```

빈 경우 TRUE를 반환

배열 기반 스택의 구현: PUSH, POP, PEEK

```
void SPush(Stack * pstack, Data data)
{
    pstack->topIndex += 1;
    pstack->stackArr[pstack->topIndex] = data;
}
```

// topIndex 증가후 topIndex를 근거로 데이터를 저장




```
Data SPop(Stack * pstack)
{
    int rIdx;

    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }

    rIdx = pstack->topIndex;
    pstack->topIndex -= 1;

    return pstack->stackArr[rIdx];
}
```

```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack))
    {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->stackArr[pstack->topIndex];
}
```

//topIndex가 가리키는 데이터를 반환!!!

// topIndex 를 임시 인덱스(rIndx)에 넣고, topIndex를 감소, rIndx를 근거로 데이터를 반환
// ?? 현재 위치의 배열값은 어떻게 삭제하는가?

배열 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

ayBaseStack

ArrayBaseStack.h
ArrayBaseStack.c
ArrayBaseStackMain.c

```
5 4 3 2 1
```

실행결과

Chapter 04-3:

스택의 연결 리스트 기반 구현

연결 리스트 기반 스택의 논리와 헤더파일의 정의

이렇듯 메모리 구조만 보아서는 스택임이 구분되지 않는다! – 메모리 구조만으로 자료 구조가 결정되는 것이 아님.

ADT(기능)을 어떻게 결정하느냐에 따라 자료구조가 달라 진다.

//head에 데이터를 추가하는 단순 연결리스트..

▶ [그림 06-2: 스택의 구현에 활용할 리스트 모델]

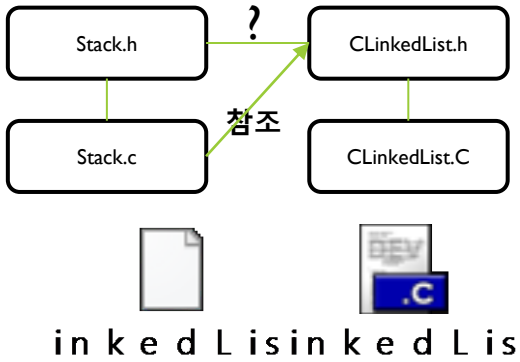
저장된 순서의 역순으로 데이터(노드)를 참조(삭제)

하는 연결 리스트가 바로 연결 기반의 스택이다!

```
typedef int Data;

typedef struct _node
{
    Data data;
    struct _node * next;
} Node;

typedef struct _listStack
{
    Node * head;
} ListStack;
```

//추가/삭제 모두 head에서 일어나므로....

```
typedef ListStack Stack;

void StackInit(Stack * pstack);
int SIsEmpty(Stack * pstack);

void SPush(Stack * pstack, Data data);
Data SPop(Stack * pstack);
Data SPeek(Stack * pstack);
```



RPT 문제 : CLinkedList.h와 CLinkedList.c를 단순히 활용하여 스택을 구현하라!?

연결 리스트 기반 스택의 구현 1

```
void StackInit(Stack * pstack)
{
     pstack->head = NULL;
}
```

```
int SIsEmpty(Stack * pstack)
{
     if(pstack->head == NULL)
         return TRUE;
     else
         return FALSE;
}
```

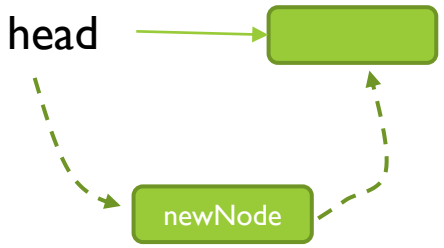
```
Data SPop(Stack * pstack)
{
     Data rdata;
     Node * rnode;

     if(SIsEmpty(pstack)) {
         printf("Stack Memory Error!");
         exit(-1);
     }

     rdata = pstack->head->data;
     rnode = pstack->head;

     pstack->head = pstack->head->next;
     free(rnode);
     return rdata;
}
```

새 노드를 머리에 추가하고, 삭제 시 머리부터 삭제하는 단순 연결 리스트의 코드에 지나지 않는다.

//헤드가 가리키는 노드의 **데이터**를 rdata에 백업
//헤드가 가리키는 노드의 **주소값**을 rnode에 백업
//헤드를 옮김
//메모리 free
//rdata 반환

연결 리스트 기반 스택의 구현 2

```
void SPush(Stack * pstack, Data data)
{
    Node * newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;
    newNode->next = pstack->head;  //head의 주소값을 새노드의 next에

    pstack->head = newNode;        //새 노드의 주소값을 head에
}
```




```
Data SPeek(Stack * pstack)
{
    if(SIsEmpty(pstack)) {
        printf("Stack Memory Error!");
        exit(-1);
    }

    return pstack->head->data;
}
```

연결 기반 스택의 활용: main 함수

```
int main(void)
{
    // Stack의 생성 및 초기화 ///////
    Stack stack;
    StackInit(&stack);

    // 데이터 넣기 ///////
    SPush(&stack, 1); SPush(&stack, 2);
    SPush(&stack, 3); SPush(&stack, 4);
    SPush(&stack, 5);

    // 데이터 꺼내기 ///////
    while(!SIsEmpty(&stack))
        printf("%d ", SPop(&stack));

    return 0;
}
```

배열 기반 리스트 관련 main 함수와 완전히 동일하게 정의된 main 함수!

ListBaseStack.h
ListBaseStack.c
ListBaseStackMain.c

:BaseStack.

```
5 4 3 2 1
```

실행결과

Chapter 04-4:

계산기 프로그램 구현

구현할 계산기 프로그램의 성격

다음과 같은 문장의 수식을 계산할 수 있어야 한다.

(3 + 4) * (5 / 2) + (7 + (9 – 5))

다음과 같은 문장의 수식계산을 위해서는 다음 두 가지를 고려해야 한다.

- 소괄호를 파악하여 그 부분을 먼저 연산한다.
- 연산자의 우선순위를 근거로 연산의 순위를 결정한다.

계산기 구현에 필요한 알고리즘은 스택과는 별개의 것이다.

다만 그 알고리즘의 구현에 있어서 스택이 매우 요긴하게 활용된다.

세 가지 수식의 표기법: 전위, 중위, 후위

- 중위 표기법(infix notation) 예) 5 + 8 / 2 → 9

  수식 내에 연산의 순서에 대한 정보가 담겨 있지 않다. 그래서 소괄호와 연산자의 우선순위라는 것을 정의하여 이를 기반으로 연산의 순서를 명시한다.

- 전위 표기법(prefix notation) 예) + 5 / 8 2 → 9

  수식 내에 연산의 순서에 대한 정보가 담겨 있다. 그래서 소괄호가 필요 없고 연산의 우선순위를 결정할 필요도 없다.

- 후위 표기법(postfix notation) 예) 5 8 2 / + → 9

  전위 표기법과 마찬가지로 수식 내에 연산의 순서에 대한 정보가 담겨 있다. 그래서 소괄호가 필요 없고 연산의 우선순위를 결정할 필요도 없다.

소괄호와 연산자의 우선순위를 인식하게 하여
중위 표기법의 수식을 직접 계산하게 프로그래밍 하는 것보다
후위 표기법의 수식을 계산하도록 프로그래밍 하는 것이 더 쉽다!

- 표기법 예제

중위 표기법	전위 표기법	후위 표기법
2+3*4	+2*34	234*+
a*b+5	+5*ab	ab*5+
(1+2)+7	+7+12	12+7+

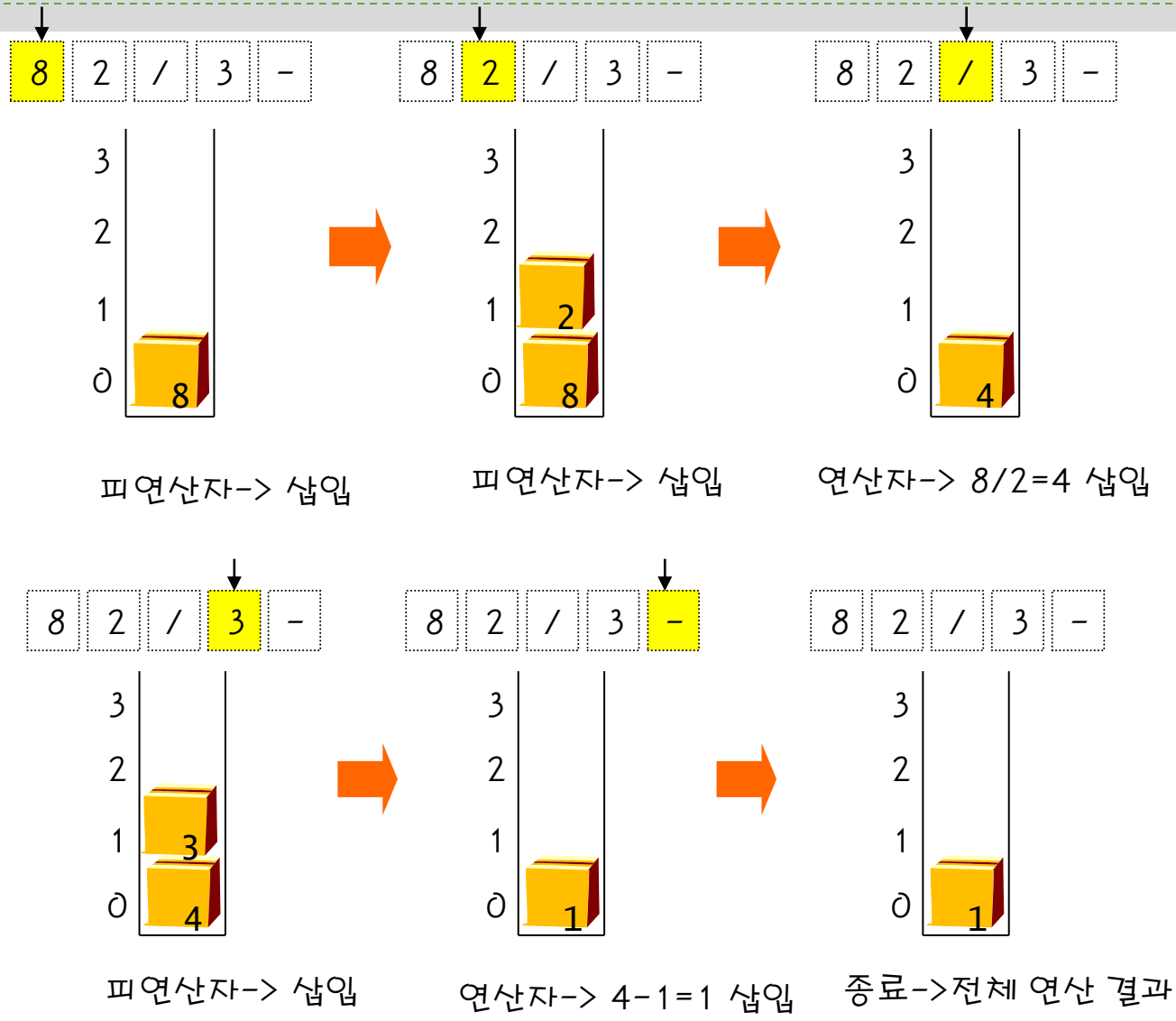
- 컴퓨터에서의 수식 계산순서
  - 중위표기식-> 후위표기식->계산
  - 2+3*4 -> 234*+ -> 14
  - 모두 스택을 사용
  - 먼저 후위표기식의 계산법을 알아보자

후위 표기식 계산

- 수식을 왼쪽에서 오른쪽으로 스캔하여 피연산자이면 스택에 저장하고 연산자이면 필요한 수만큼의 피연산자를 스택에서 꺼내 연산을 실행하고 연산의 결과를 다시 스택에 저장
- (예) 82/3-32*+

토큰	스택						
	[0]	[1]	[2]	[3]	[4]	[5]	[6]
8	8						
2	8	2					
/	4						
3	4	3					
-	1						
3	1	3					
2	1	3	2				
*	1	6					
+	7						

후위 표기식 계산


8 2 / 3 -
3
2
1
0
8
피연산자-> 삽입
8 2 / 3 -
2
8
피연산자-> 삽입
8 2 / 3 -
4
연산자-> 8/2=4 삽입
8 2 / 3 -
3
4
피연산자-> 삽입
8 2 / 3 -
1
연산자-> 4-1=1 삽입
8 2 / 3 -
1
종료->전체 연산 결과=1

중위 → 후위 : 소괄호 고려하지 않고 1

▶ [그림 06-3: 수식 변환의 과정 1/7]

수식을 이루는 왼쪽 문자부터 시작해서 하나씩 처리해 나간다.

▶ [그림 06-4: 수식 변환의 과정 2/7]

피 연산자를 만나면 무조건 변환된 수식이 위치할 자리로 이동시킨다.

중위 → 후위 : 소괄호 고려하지 않고 2

▶ [그림 06-5: 수식 변환의 과정 3/7]

연산자를 만나면 무조건 쟁반으로 이동한다.

▶ [그림 06-6: 수식 변환의 과정 4/7]

숫자를 만났으니 변환된 수식이 위치할 자리로 이동!

중위 → 후위 : 소괄호 고려하지 않고 3

/ 연산자의 우선순위가 높으므로 + 연산자 위에 올린다.

▶ [그림 06-7: 수식 변환의 과정 5/7]

쟁반에 기존 연산자가 있는 상황에서의 행동 방식!

☞ 쟁반에 위치한 연산자의 우선순위가 높다면

- 쟁반에 위치한 연산자를 꺼내서 변환된 수식이 위치할 자리로 옮긴다.
- 그리고 새 연산자는 쟁반으로 옮긴다.

☞ 쟁반에 위치한 연산자의 우선순위가 낮다면

- 쟁반에 위치한 연산자의 위에 새 연산자를 쌓는다.

우선순위가 높은 연산자는 우선순위가 낮은 연산자 위에 올라서서, 우선순위가 낮은 연산자가 먼저 자리를 잡지 못하게 하려는 목적!

중위 → 후위 : 소괄호 고려하지 않고 4

▶ [그림 06-8: 수식 변환의 과정 6/7]

피 연산자는 무조건 변환된 수식의 자리로 이동!

▶ [그림 06-9: 수식 변환의 과정 7/7]

나머지 연산자들은 쟁반에서 차례로 옮긴다!

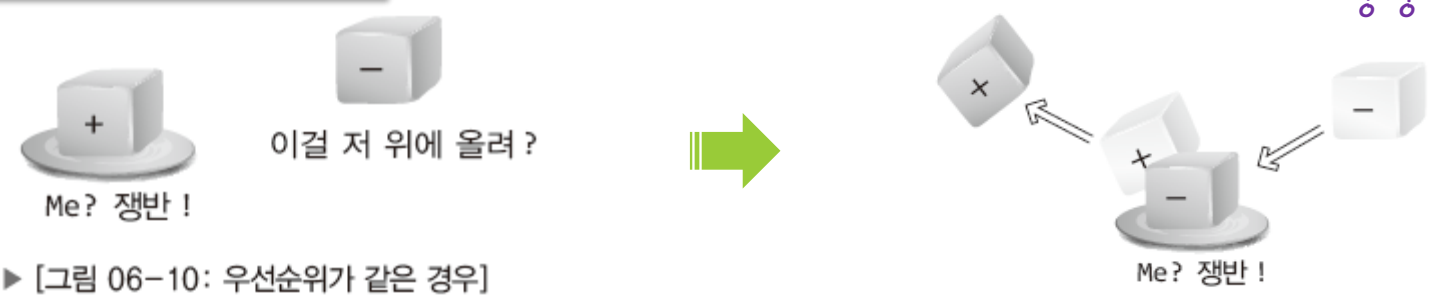
중위 → 후위 : 정리하면

변환 규칙의 정리 내용

- 피 연산자는 그냥 옮긴다.
- 연산자는 쟁반으로 옮긴다.
- 연산자가 쟁반에 있다면 우선순위를 비교하여 처리방법을 결정한다.
- 마지막까지 쟁반에 남아있는 연산자들은 하나씩 꺼내서 옮긴다.

중위 → 후위 : 고민 될 수 있는 상황

우선 순위가 같으면??



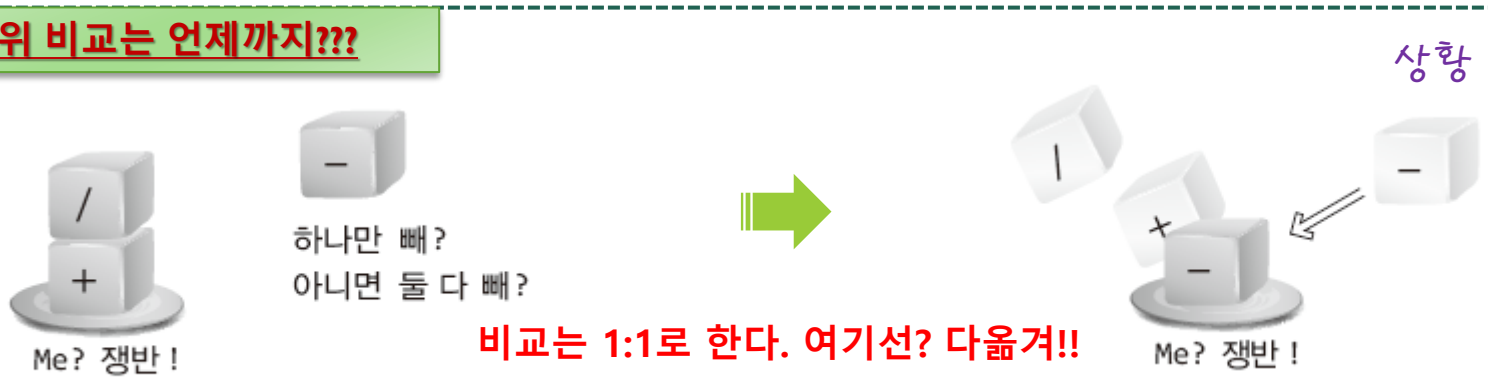
▶ [그림 06-10: 우선순위가 같은 경우]

먼저 도착한 연산자가 우선순위가 높다고 가정하자!!

+ 연산자가 우선순위가 높다고 가정하고(+ 연산자가 먼저 등장했으므로) 일을 진행한다.

즉 + 연산자를 옮기고 그 자리에 – 연산자를 가져다 놔야 한다."

우선 순위 비교는 언제까지???



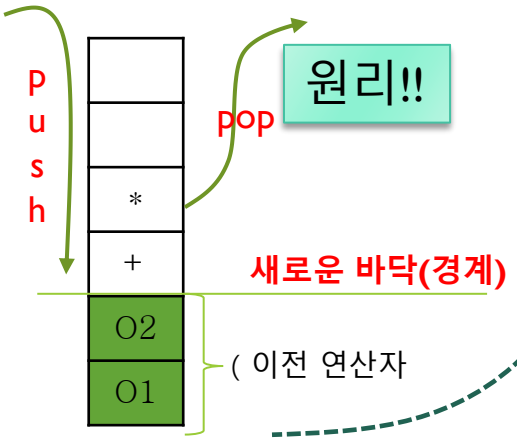
▶ [그림 06-12: 둘 이상의 연산자가 쌓여 있는 경우]

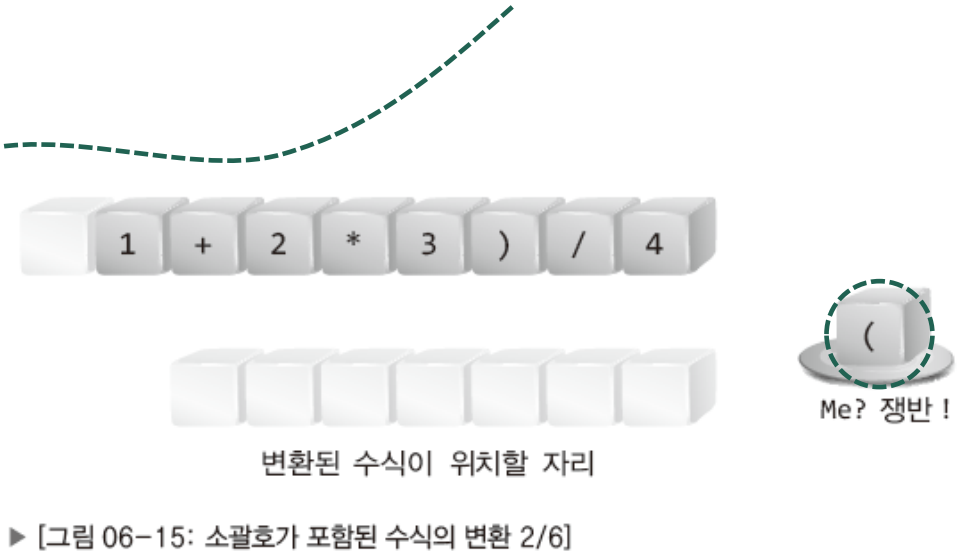
중위 → 후위 : 소괄호 고려 1

후위 표기법의 수식에서는 먼저 연산이 이뤄져야 하는 연산자가 뒤에 연산이 이뤄지는 연산자보다 앞에 위치해야 한다. 따라서 소괄호 안에 있는 연산자들이 후위 표기법의 수식에서 앞부분에 위치해야 한다.

(연산자의 우선순위는 그 어떤 사칙 연산자들보다 낮다고 간주! 그래서) 연산자가 등장할 때까지 쟁반에 남아 소괄호의 경계 역할을 해야 함!

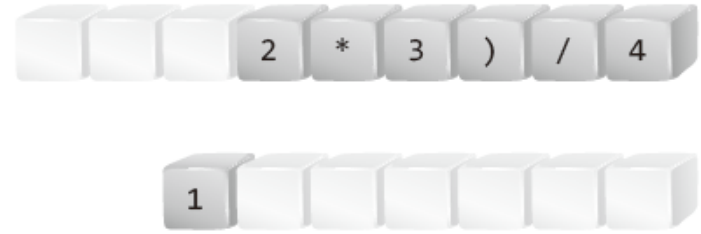
▶ [그림 06-14: 소괄호가 포함된 수식의 변환 1/6]





▶ [그림 06-15: 소괄호가 포함된 수식의 변환 2/6]

중위 → 후위 : 소괄호 고려 2



▶ [그림 06-16: 소괄호가 포함된 수식의 변환 3/6]

▶ [그림 06-17: 소괄호가 포함된 수식의 변환 4/6]

▶ [그림 06-18: 소괄호가 포함된 수식의 변환 5/6]

) 연산자를 만나면 쟁반에서 (연산자 만날 때까지 연산자를 변환된 수식의 자리로 옮긴다!

중위 → 후위 : 소괄호 고려 3

▶ [그림 06-19: 소괄호가 포함된 수식의 변환 6/6]

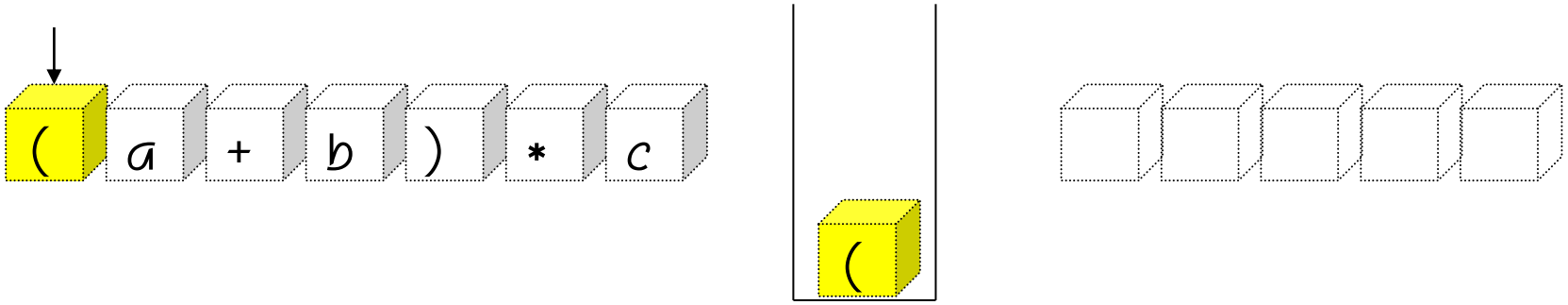
조금 달리 설명하면 (연산자는 쟁반의 또 다른 바닥이다! 그리고) 연산자는 변환이 되어야 하는 작은 수식의 끝을 의미한다. 그래서) 연산자를 만나면 (연산자를 만날 때까지 연산자를 이동 시켜야 한다.

지금까지 설명한 내용에 해당하는 코드의 구현에 앞서 중위 표기법의 수식을 후위 표기법의 수식으로 바꾸는 연습을 할 필요가 있다.

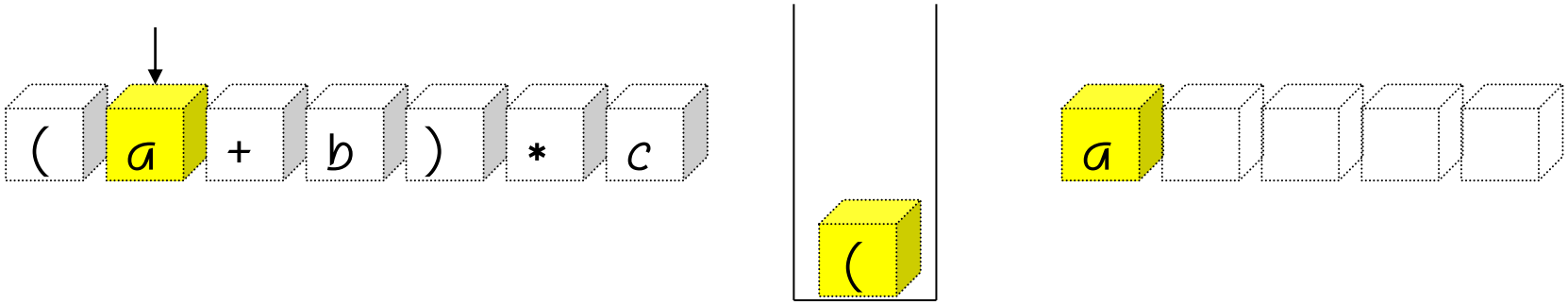
중위표기법->후위표기법 (정리)

- 중위표기와 후위표기
  - 중위 표기법과 후위 표기법의 공통점은 피연산자의 순서는 동일
  - 연산자들의 순서만 다름(우선순위순서)
    ->연산자만 스택에 저장했다가 출력하면 된다.
  - 2+3*4 -> 234*+
- 알고리즘
  - 피연산자를 만나면 그대로 출력
  - 연산자를 만나면
    - 스택에 저장했다가
    - 스택보다 우선 순위가 낮은 연산자가 나오면 그때 출력
  - 왼쪽 괄호는 우선순위가 가장 낮은 연산자로 취급
  - 오른쪽 괄호가 나오면 스택에서 왼쪽 괄호 위에 쌓여있는 모든 연산자를 출력

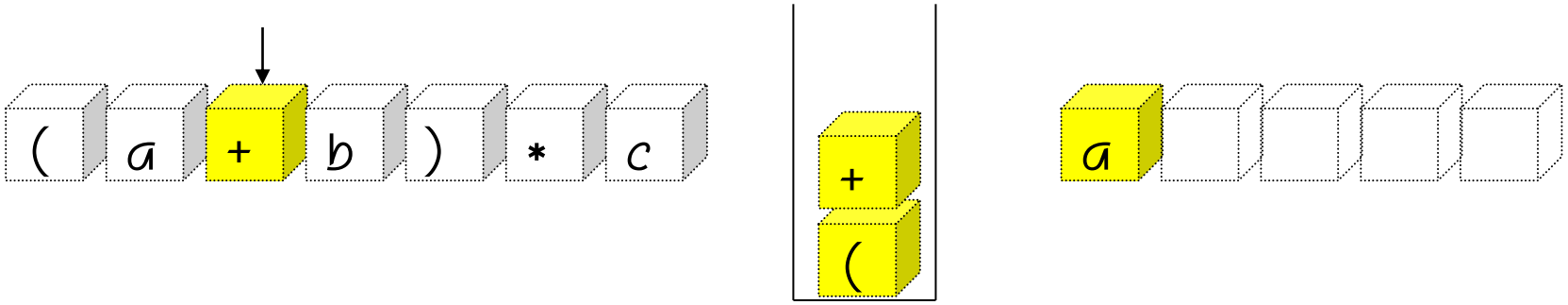
중위표기법->후위표기법



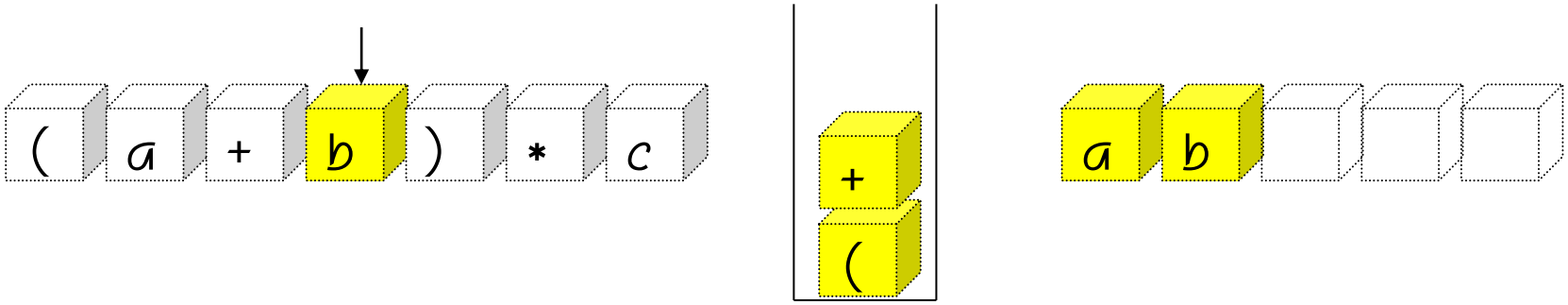
중위표기법->후위표기법



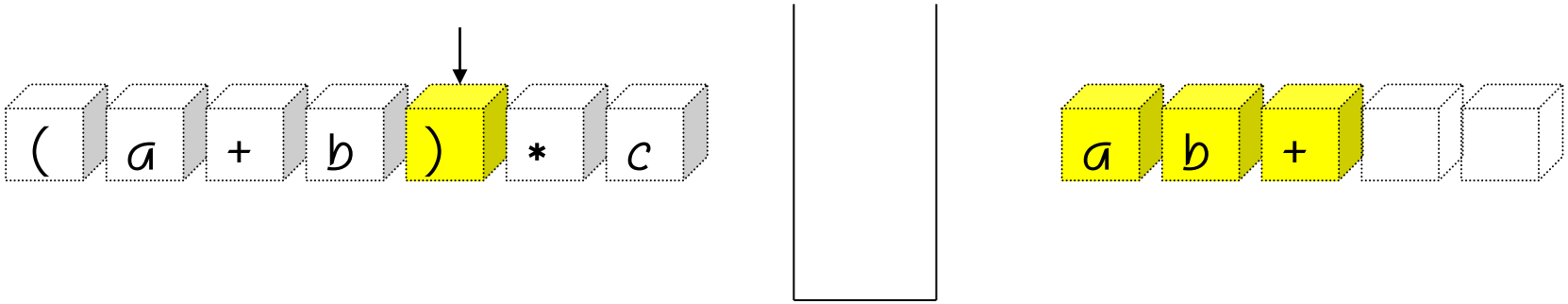
중위표기법->후위표기법



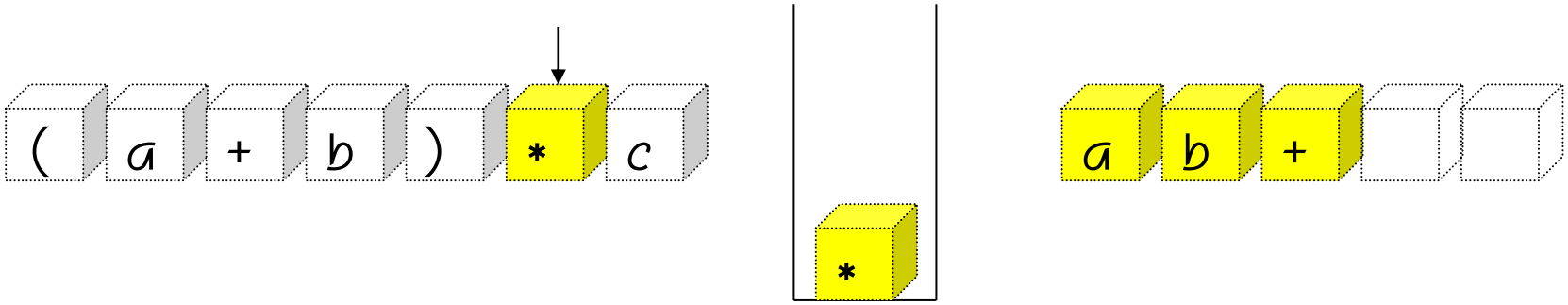
중위표기법->후위표기법



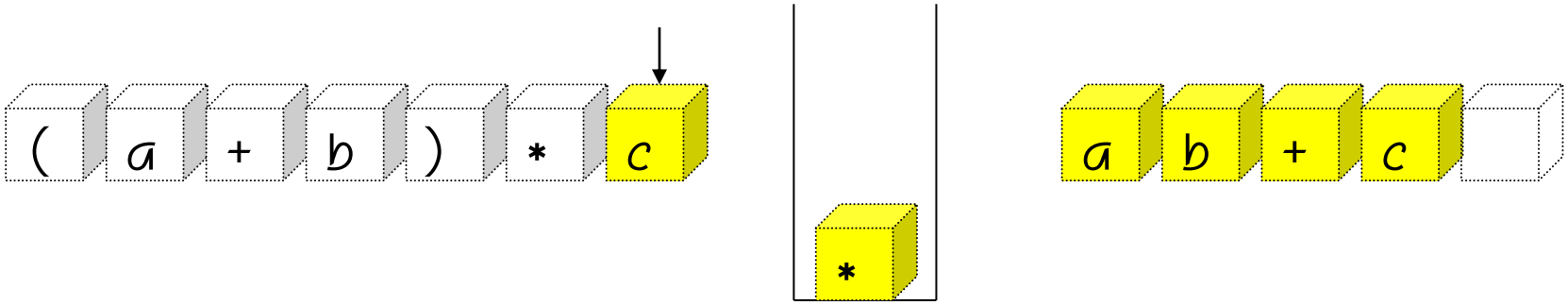
중위표기법->후위표기법



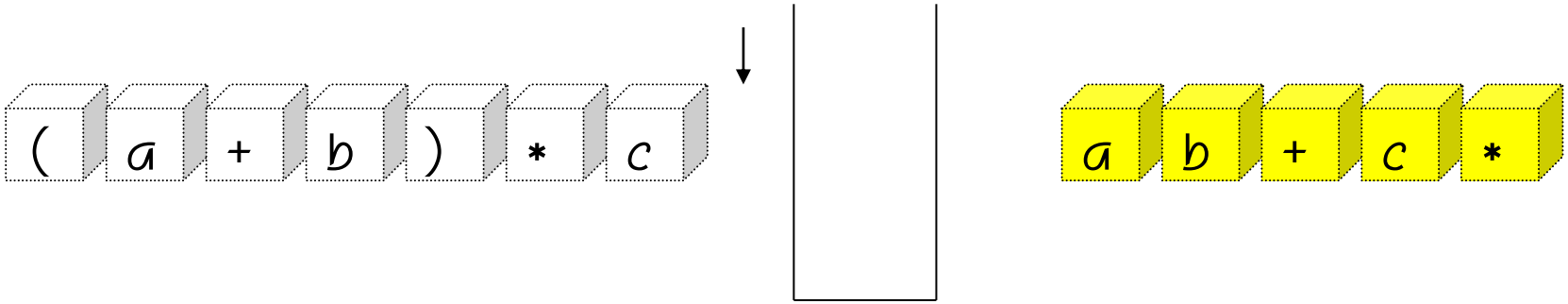
중위표기법->후위표기법



중위표기법->후위표기법



중위표기법->후위표기법



중위 → 후위 : 프로그램 구현 1

```
void ConvToRPNExp(char exp[])
{
    . . . .
}
```

변환 함수의 타입

함수 이름의 일부인 RPN은 후위 표기법의 또 다른 이름인 Reverse Polish Notation의 약자이다.

```
int main(void)
{
    char exp[] = "3-2+4";
    ConvToRPNExp(exp);
    . . . .
}
```

중위 표기법 수식을 배열에 담아 함수의 인자로 전달한다.

호출 완료 후 exp에는 변환된 수식이 담긴다.

중위 → 후위 : 프로그램 구현 2

Helper function??

함수 ConvToRPNExp의 첫 번째 helper function!

```
int GetOpPrec(char op)      // 연산자의 연산 우선순위 정보를 반환한다.
{
    switch(op)
    {
    case '*':
    case '/':
        return 5;           // 가장 높은 연산의 우선순위
    case '+':
    case '-':
        return 3;
    case '(':
        return 1;
    }

    return -1;              // 등록되지 않은 연산자임을 알림!
}
```

연산자의 우선순위에 대응하는 값을 반환한다. 값이 클수록 우선순위가 높은 것으로 정의되어 있다.

'('연산자는 ')' 연산자가 등장할 때까지 쟁반에 남아 있어야 하기 때문에 가장 낮은 우선순위를 부여! – 스택에 항상 남아 있어야 한다.

')'는 '('를 만날때까지 연산자를 빼내라는 Signal로서 쓰이므로 스택에 넣을 필요가 없다.
때문에) 연산자에 대한 반환 값은 정의되어 있을 필요가 없다!

중위 → 후위 : 프로그램 구현 3

이전 함수에서 연산자를 각각 전달하여 반환값을 전달 받은 후 대소비교를 통해 연산자의 우선순위를 판단하기 위한 함수

함수 ConvToRPNExp의 두 번째 helper function!

```
int WhoPrecOp(char op1, char op2)      두 연산자의 우선순위 비교 결과를 반환한다.
{
    int op1Prec = GetOpPrec(op1);
    int op2Prec = GetOpPrec(op2);

    if(op1Prec > op2Prec)          // op1의 우선순위가 더 높다면
        return 1;
    else if(op1Prec < op2Prec)     // op2의 우선순위가 더 높다면
        return -1;
    else
        return 0;                  // op1과 op2의 우선순위가 같다면
}
```

GetOpPrec() 호출하여 연산자 우선순위값 초기화

ConvToRPNExp 함수의 실질적인 Helper Function은 위의 함수 하나이다!

중위 → 후위 : 프로그램 구현 4

```
void ConvToRPNExp(char exp[])
{
    Stack stack;
    int expLen = strlen(exp);
    char * convExp = (char*)malloc(expLen+1);   변환된 수식을 담을 공간 마련

    int i, idx=0;
    char tok, popOp;

    memset(convExp, 0, sizeof(char)*expLen+1);   마련한 공간 0으로 초기화
    StackInit(&stack);

    for(i=0; i<expLen; i++)  {          일련의 변환 과정을 이 반복문 안에서 수행
        . . . .                         → exp 의 각 문자단위의 처리를 위한 루프
    }

    while(!SIsEmpty(&stack))            참이 반환될때(빌때) 까지 스택에 담긴 연산자를 순서대로 이동
        convExp[idx++] = SPop(&stack);  즉, 남아 있는 모든 연산자를 이동시키는 반복문

    strcpy(exp, convExp);  변환된 수식을 반환!
    free(convExp);                      (변환된 수식을 인자로 전달된 주소값의 메모리에 복사하기위해)
}
```

중위 → 후위 : 프로그램 구현 5

Exp[]의 내용이 숫자면 자리잡고, 연산자면 스택으로~

```
void ConvToRPNExp(char exp[])
{
    . . . .

    for(i=0; i<expLen; i++)
    {
        tok = exp[i];

        if(isdigit(tok))    tok에 저장된 문자가 피연산자라면
        {
            convExp[idx++] = tok;
        }
        else                tok에 저장된 문자가 연산자라면
        {
            switch(tok)
            {
                . . . .     연산자일 때의 처리 루틴을 switch문에 담는다!
            }
        }
    }
}
```

중위 → 후위 : 프로그램 구현 6

함수 ConvToRPNExp의 일부인 switch문

```
switch(tok)
{
case '(' :                  // 여는 소괄호라면,
    SPush(&stack, tok);     // 스택에 쌓는다.
    break;
case ')' :                  // 닫는 소괄호라면,
    while(1)                // 반복해서,
    {
        popOp = SPop(&stack);    // 스택에서 연산자를 꺼내어,
        if(popOp == '(')         // 연산자 ( 을 만날 때까지,
            break;
        convExp[idx++] = popOp;  // 배열 convExp에 저장한다.
    }
    break;
case '+': case '-':
case '*': case '/':
    while(!SIsEmpty(&stack) && WhoPrecOp(SPeek(&stack), tok) >= 0)
        convExp[idx++] = SPop(&stack);
    SPush(&stack, tok);
    break;
}
```

tok에 저장된 연산자를 스택에 저장하기 위한 과정

새로운 연산자와 스택에 있는 연산자와 우선순위 비교해서 pop 수행
→ 기존 스택 내 연산자가 새로운 연산자보다 우선순위가 낮아질 때 까지..

비교는 peek() 호출..

중위 → 후위 : 프로그램의 실행

xToPostfix.

```
int main(void)
{
    char exp1[] = "1+2*3";
    char exp2[] = "(1+2)*3";
    char exp3[] = "((1-2)+3)*(5-2)";

    ConvToRPNExp(exp1);
    ConvToRPNExp(exp2);
    ConvToRPNExp(exp3);

    printf("%s \n", exp1);
    printf("%s \n", exp2);
    printf("%s \n", exp3);
    return 0;
}
```

InfixToPostfix.h
InfixToPostfix.c
: ConvToRPNExp 함수의 선언과 정의

ListBaseStack.h
ListBaseStack.c
: 스택관련 함수의 선언과 정의

InfixToPostfixMain.c : main 함수의 정의

```
123*+
12+3*
12-3+52-*
```

실행결과

후위 표기법 수식의 계산

피연산자 두 개가 연산자 앞에 항상 위치하는 구조

1) 왼쪽부터 해석
2) 연산자를 만나면 앞 2개의 피연산자를 거꾸로 꺼내어 연산
3) 연산 결과 반환
4) 다시 해석~

3 + 2 * 4

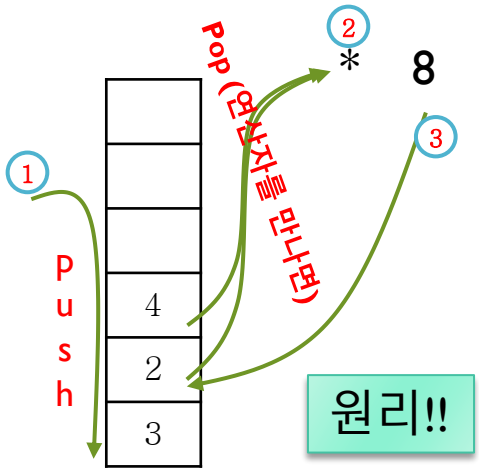
후위 표기법 수식으로

3 2 4 * +

3 2 4 * +

2와 4의 곱 진행

3 8 +



(1 * 2 + 3) / 4

후위 표기법 수식으로

1 2 * 3 + 4 /

1과 2의 곱 진행

2 3 + 4 /

2와 3의 합 진행

5 4 /

후위 표기법 수식 계산 프로그램의 구현

계산의 규칙

- 피연산자는 무조건 스택으로 옮긴다.
- 연산자를 만나면 스택에서 두 개의 피연산자를 꺼내서 계산을 한다.
- 계산결과는 다시 스택에 넣는다.

▶ [그림 06-20: 후위 표기법의 수식 계산 1]

▶ [그림 06-21 : 후위 표기법의 수식 계산 2]

후위 표기법 수식 계산 프로그램의 구현

```
int EvalRPNExp(char exp[])  //후위 표기법 수식이 담긴 메모리의 주소값을 인자로 전달
{
    Stack stack; //스택 초기화
    int expLen = strlen(exp); //후위 표기법 수식의 문자 개수 산출
    int i;
    char tok, op1, op2;

    StackInit(&stack); //스택 초기화

    for(i=0; i<expLen; i++) //문자 개수만큼..이 어서
    {
        tok = exp[i];   //문자를 하나씩 취득
        if(isdigit(tok) //숫자인 경우
        {
            SPush(&stack, tok - '0'); //'3'을 3으로
        }
        else
        {
            op2 = SPop(&stack);  //연산자면 두개의 피연산자를 꺼냄
            op1 = SPop(&stack);
                                                    }
            switch(tok)
            {
            case '+':
                SPush(&stack, op1+op2);
                break;
            case '-':
                SPush(&stack, op1-op2);
                break;
            case '*':
                SPush(&stack, op1*op2);
                break;
            case '/':
                SPush(&stack, op1/op2);
                break;
            }
        }
    }
    return SPop(&stack); //최종 연산결과를 반환~ 스택을 비움
```

후위 표기법 수식 계산 프로그램의 실행

PostCalculator.h
PostCalculator.c
: EvalRPNExp 함수의 선언과 정의

ListBaseStack.h
ListBaseStack.c
: 스택관련 함수의 선언과 정의

PostCalculatorMain.c : main 함수의 정의

```
int main(void)
{
    char postExp1[] = "42*8+";
    char postExp2[] = "123+*4/";

    printf("%s = %d \n", postExp1, EvalRPNExp(postExp1));
    printf("%s = %d \n", postExp2, EvalRPNExp(postExp2));

    return 0;
}
```

```
42*8+ = 16
123+*4/ = 1
```

실행결과

계산기 프로그램의 완성1

계산의 과정

중위 표기법 수식 → ConvToRPNExp → EvalRPNExp → 연산결과

후위 표기법 수식

계산기 프로그램의 파일 구성

• 스택의 활용	ListBaseStack.h, ListBaseStack.c
• 후위 표기법의 수식으로 변환	InfixToPostfix.h, InfixToPostfix.c
• 후위 표기법의 수식을 계산	PostCalculator.h, PostCalculator.c
• 중위 표기법의 수식을 계산	InfixCalculator.h, InfixCalculator.c
• main 함수	InfixCalculatorMain.c

InfixCalculator.h　　InfixCalculator.c

계산기 프로그램의 완성2

xCalculator.

InfixCalculator.h

```
#ifndef __INFIX_CALCULATOR__
#define __INFIX_CALCULATOR__

int EvalInfixExp(char exp[]);

#endif
```

InfixCalculatorMain.c

```
int main(void)
{
    char exp1[] = "1+2*3";
    char exp2[] = "(1+2)*3";
    char exp3[] = "((1-2)+3)*(5-2)";

    printf("%s = %d \n", exp1, EvalInfixExp(exp1));
    printf("%s = %d \n", exp2, EvalInfixExp(exp2));
    printf("%s = %d \n", exp3, EvalInfixExp(exp3));
    return 0;
}
```

InfixCalculator.c

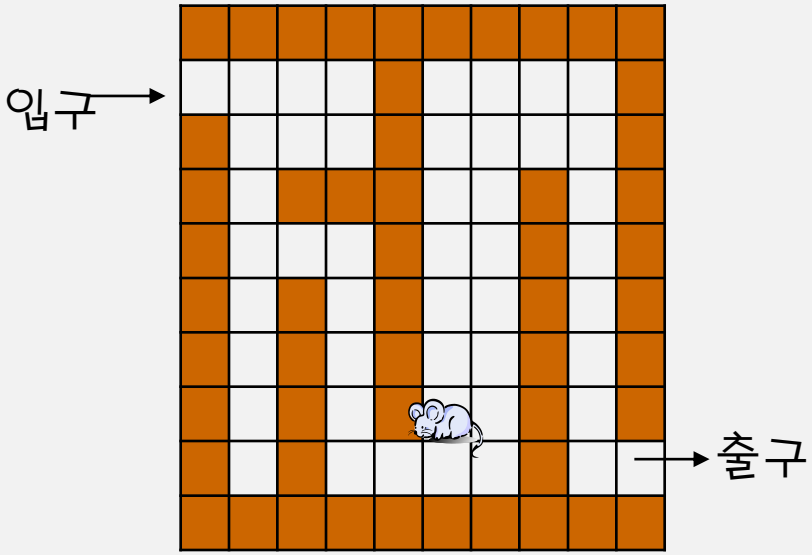
```
int EvalInfixExp(char exp[])
{
    int len = strlen(exp);
    int ret;
    char * expcpy = (char*)malloc(len+1);  // 문자열 저장공간 마련
    strcpy(expcpy, exp);                   // exp를 expcpy에 복사 - 원본을 유지하기 위해

    ConvToRPNExp(expcpy);      // 후위 표기법의 수식으로 변환
    ret = EvalRPNExp(expcpy);  // 변환된 수식의 계산

    free(expcpy);    // 문자열 저장공간 해제
    return ret;      // 계산결과 반환
}
```

실행결과

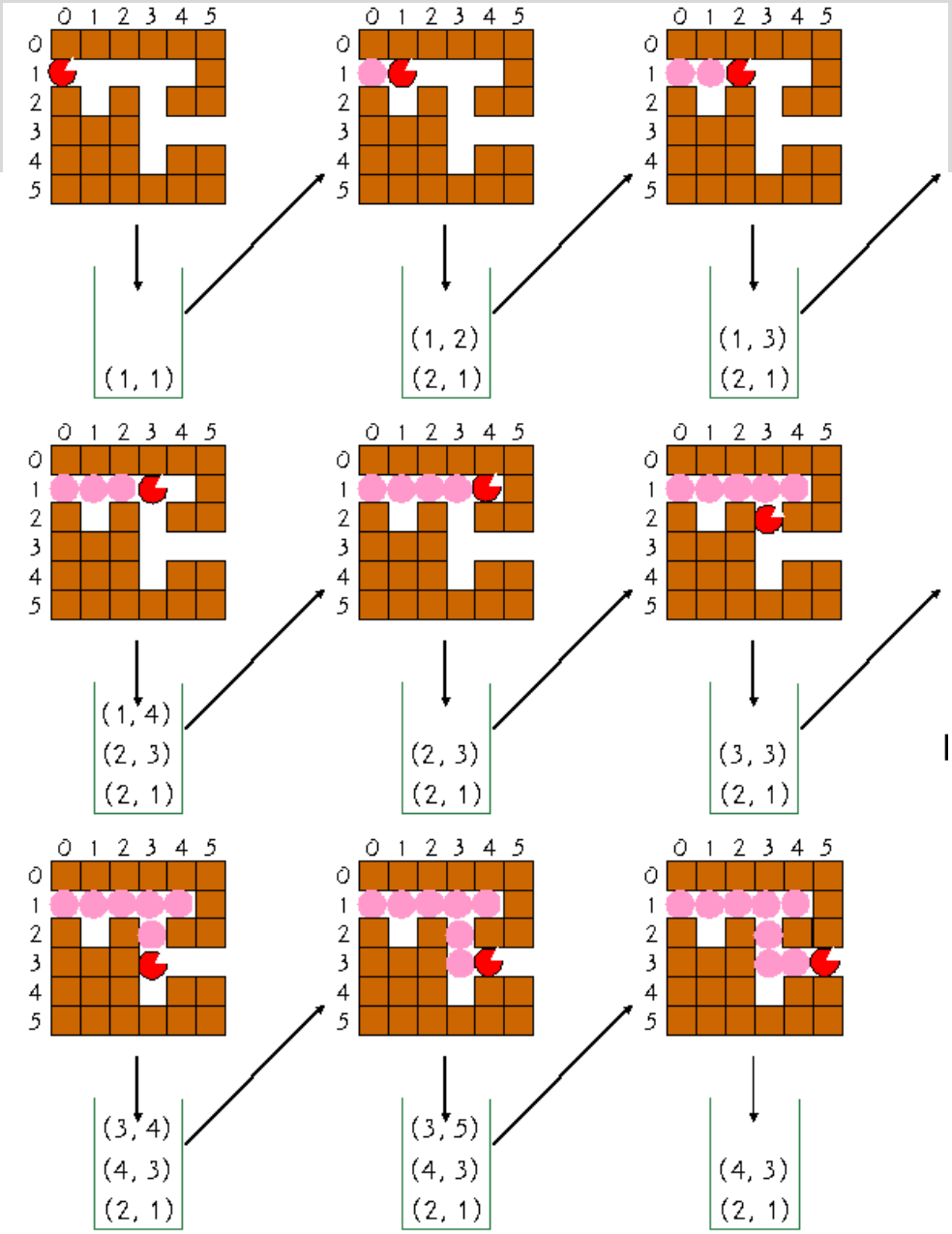
```
1+2*3 = 7
(1+2)*3 = 9
((1-2)+3)*(5-2) = 6
```

미로탐색문제

- 체계적인 방법 필요
- 현재의 위치에서 가능한 방향을 스택에 저장해놓았다가 막다른 길을 만나면 스택에서 다음 탐색 위치를 꺼낸다.




```
1 1 1 1 1 1 1 1 1 1
0 0 0 0 1 0 0 0 0 1
1 0 0 0 1 0 0 0 0 1
1 0 1 1 1 0 0 1 0 1
1 0 0 0 1 0 0 1 0 1
1 0 1 0 1 0 0 1 0 1
1 0 1 0 1 0 0 1 0 1
1 0 1 0 1 m 0 1 0 1
1 0 1 0 0 0 0 1 0 x
1 1 1 1 1 1 1 1 1 1
```



미로탐색 알고리즘

```
스택 s과 출구의 위치 x, 현재 생쥐의 위치를 초기화
while( 현재의 위치가 출구가 아니면 )
  do  현재위치를 방문한 것으로 표기
      if( 현재위치의 위, 아래, 왼쪽, 오른쪽 위치가 아직 방문되지 않았고 갈수 있으면 )
        then 그 위치들을 스택에 push
      if( is_empty(s) )
         then 실패
         else 스택에서 하나의 위치를 꺼내어 현재 위치로 만든다;
성공;
```

미로 프로그램

```
#define MAX_STACK_SIZE 100
#define MAZE_SIZE 6

typedef struct  StackObjectRec {
        short r;
        short c;
} StackObject;

StackObject  stack[MAX_STACK_SIZE];
int  top = -1;
StackObject here={1,0}, entry={1,0};

char maze[MAZE_SIZE][MAZE_SIZE] = {
        {'1', '1', '1', '1', '1', '1'},
        {'e', '0', '1', '0', '0', '1'},
        {'1', '0', '0', '0', '1', '1'},
        {'1', '0', '1', '0', '1', '1'},
        {'1', '0', '1', '0', '0', 'x'},
        {'1', '1', '1', '1', '1', '1'},
};
```

미로 프로그램

```
void pushLoc(int r, int c)
{
        if( r < 0 || c < 0 ) return;
        if( maze[r][c] != '1' && maze[r][c] != '.' ){
                StackObject tmp;
                tmp.r = r;
                tmp.c = c;
                push(tmp);
        }
}

void printMaze(char m[MAZE_SIZE][MAZE_SIZE])
{
        …
}
```

미로 프로그램

```
void printStack()
{
        int i;
        for(i=5;i>top;i--)
                printf("|     |\n");
        for(i=top;i>=0;i--)
                printf("|(%01d,%01d)|\n", stack[i].r, stack[i].c);
        printf("-------\n");
}
```

미로 프로그램

```
void main()
{
        int r,c;
        here = entry;
        printMaze(maze);
        printStack();
        while ( maze[here.r][here.c]!='x' ){
                printMaze(maze);
                r = here.r;
                c = here.c;
                maze[r][c] = '.';
                pushLoc(r-1,c);
                pushLoc(r+1,c);
                pushLoc(r,c-1);
                pushLoc(r,c+1);
```

미로 프로그램

```
                printStack();
                if( isEmpty() ){
                        printf("실패\n");
                        return;
                }
                else
                        here = pop();
                printMaze(maze);
                printStack();
                getch();
        }
        printf("성공\n");
}
```