

# <자료구조>

## 4. 스택

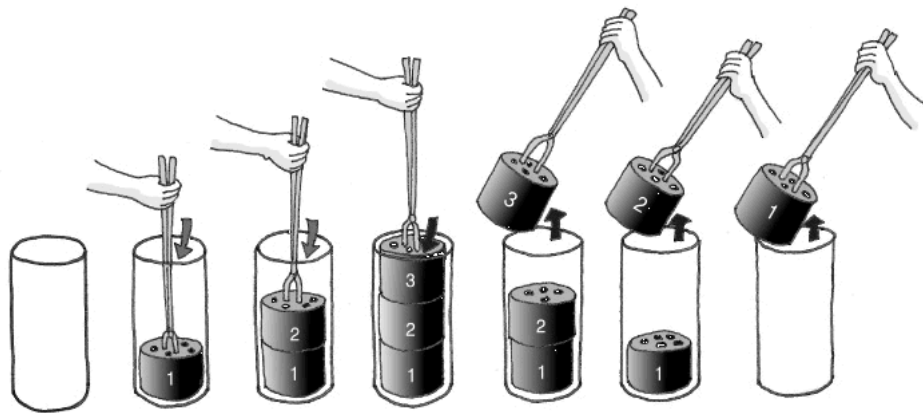
한국외국어대학교  
컴퓨터.전자시스템공학전공  
2016년 1학기  
고 석 훈

# 학습 목표

- 자료구조 스택에 대해서 이해한다.
- 스택의 특징과 연산 방법을 알아본다.
- 스택의 다양한 응용 분야를 알아본다.

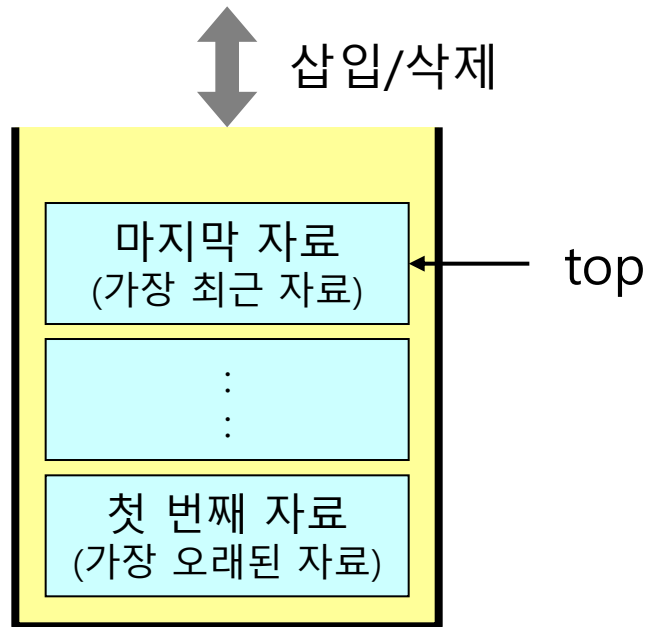
# 스택(Stack)

- 한쪽 끝에서만 자료를 삽입, 삭제하는 선형 자료구조
  - 데이터를 삽입(push)하면, 자료구조의 맨 위에 쌓인다.
  - 데이터를 삭제(pop)하면, 맨 위의 원소가 삭제된다.



# 스택의 구조

- 스택에서 삽입/삭제가 이루어지는 위치
- 마지막에 저장된 원소를 가리킨다.



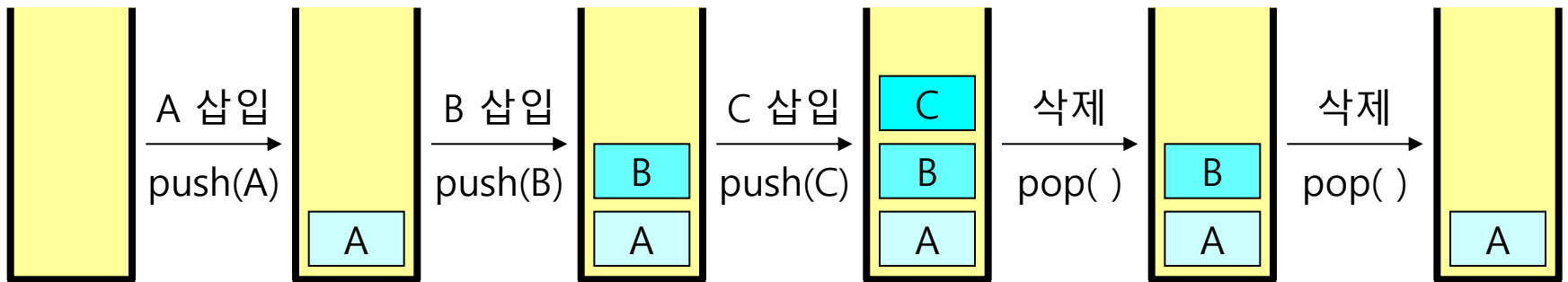
# 스택의 두 가지 연산

## ● push(value): 원소 삽입 연산

- 만일, 스택이 가득 찬 상태라면 오류 발생(stack overflow)
- 스택의 top에 새로운 원소를 추가한다.

## ● pop: 원소 추출 연산

- 만일, 스택에 원소가 하나도 없으면 오류 발생(stack empty)
- 스택의 top으로부터 원소를 삭제하고, 그 값을 알려준다.



# 스택 ADT

이 름 : Stack

데이터 : 0개 이상의 원소를 가진 유한 순서 리스트

연 산 :

initStack(Stack) ::= initialize Stack;  
// 스택을 초기화하는 연산

isEmpty(Stack) ::= check whether Stack is empty or not;  
// 스택이 공백인지 확인하는 연산

isFull(Stack) ::= check whether Stack is full or not;  
// 스택이 가득 찼는지 확인하는 연산

push(Stack, value) ::= insert item onto the top of Stack;  
// 스택의 top에 원소(value)를 삽입하는 연산

pop(Stack) ::= delete and return the top item of Stack;  
// 스택의 top에 있는 원소를 삭제하고 반환하는 연산

# isEmpty 알고리즘

```
isEmpty(S)
    if (top = -1) return true;           // ①
    else return false;                  // ②
end isEmpty( )
```

- ① top의 위치가 -1이면 저장된 자료가 하나도 없으므로 true 리턴  
(스택의 저장공간 인덱스: 0 ~ STACK\_SIZE - 1)
- ② top의 위치가 -1이 아니면 저장된 자료가 있으므로 false 리턴

# isFull 알고리즘

```
isFull(S)
    if (top = STACK_SIZE - 1) return true;      // ①
    else return false;                          // ②
end isFull( )
```

- ① top이 스택의 마지막 공간( $STACK\_SIZE - 1$ )을 가리키고 있다면  
더 이상 저장할 공간이 없으므로 true 리턴  
(스택의 저장공간 인덱스:  $0 \sim STACK\_SIZE - 1$ )
- ② 아니면 저장할 공간이 있으므로 false 리턴



# Push 알고리즘

```
push(S, x)
  if ( isFull(S) ) then overflow error;      // ①
  else
    top ← top + 1;                          // ②
    S[top] ← x;                             // ③
  end if
end push( )
```

- ① 스택에 더 이상 저장할 공간이 없으면  
오버플로우(overflow) 오류를 발생하고 연산 종료
- ② top을 1 증가하여 새로운 저장 위치를 가리키도록 함
- ③ top이 가리키는 위치에 자료 x 저장

# Pop 알고리즘

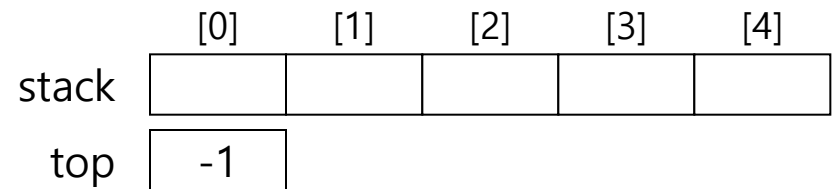
```
pop(S)
  if ( isEmpty(S) ) then empty error;      // ①
  else
    value ← S[top];                        // ②
    top ← top - 1;                         // ③
    return value;                          // ②
  end if
end pop( )
```

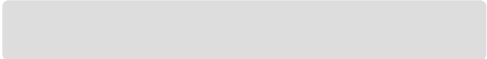

- ① 스택이 공백인 경우 반환할 자료가 없으므로 공백(empty) 오류를 발생하고 연산 종료
- ② 현재 top이 가리키는 위치의 자료를 value에 저장하여 반환하고,
- ③ top을 1 감소하여 마지막 저장 위치를 가리키도록 함

# 스택 구현 1: 순차 자료구조 [1/6]

- 1차원 배열을 이용하여 스택 구현

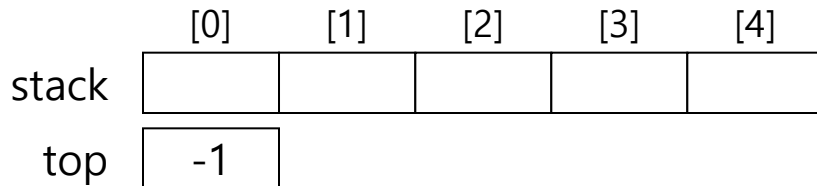
```
#define STACK_SIZE 5  
int stack[STACK_SIZE];  
int top;
```



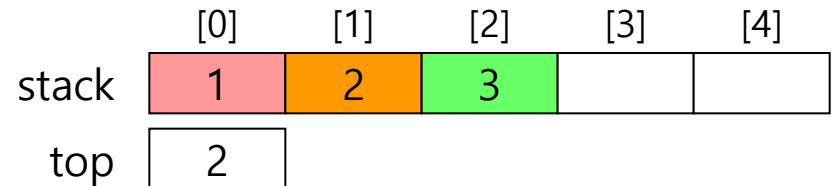
- 스택의 크기 : 배열의 크기
- 스택에 저장되는 원소의 순서 : 배열 원소의 인덱스 순서
  - ◆ 인덱스 0번 : 스택의 첫번째 원소
  - ◆ 인덱스 STACK\_SIZE-1번 : 스택의 STACK\_SIZE번째 원소
- 변수 top : 스택에 저장된 마지막 원소에 대한 인덱스 저장
  - ◆ 공백 상태 : 
  - ◆ 포화 상태 : 

# 스택 구현 1: 순차 자료구조 [2/6]

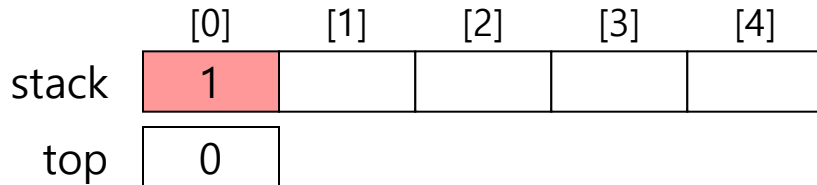
① create(stack, 5)



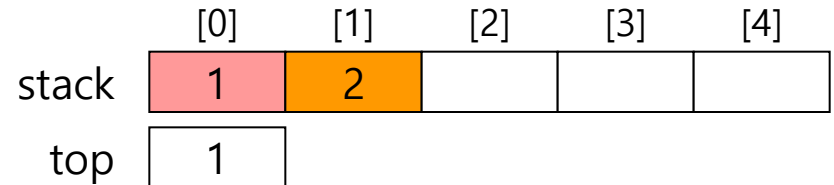
④ push(stack, 3);



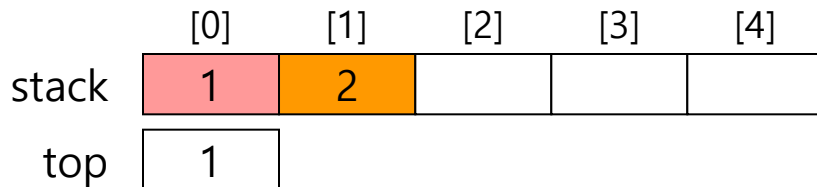
② push(stack, 1);



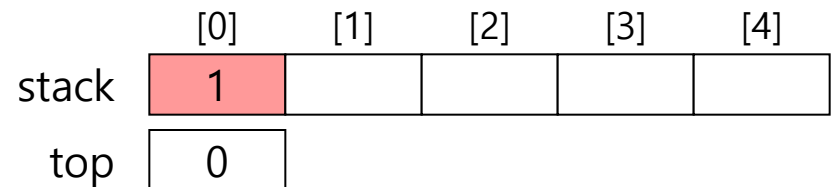
⑤ pop(stack);



③ push(stack, 2);

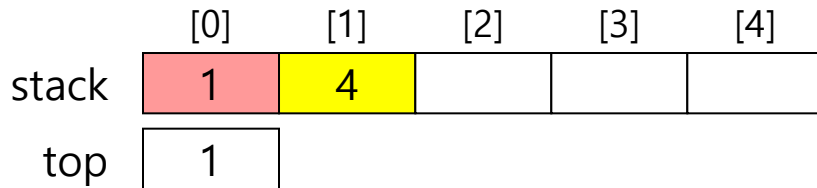


⑥ pop(stack);

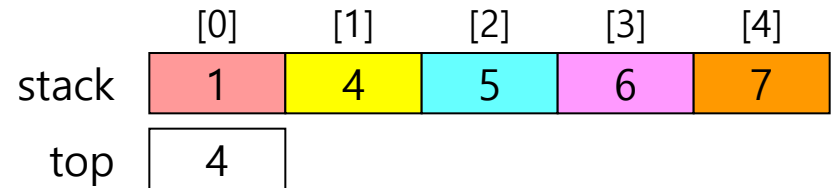


# 스택 구현 1: 순차 자료구조 [3/6]

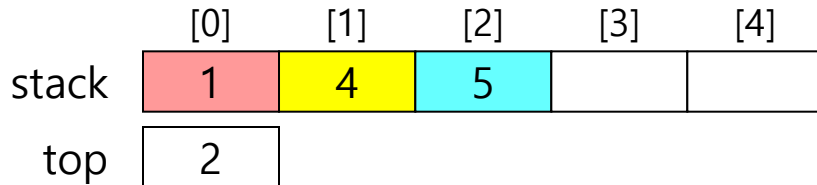
⑦ push(stack, 4);



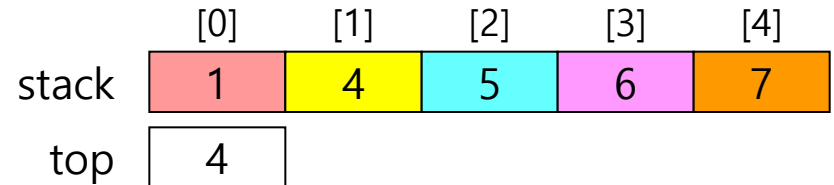
⑩ push(stack, 7);



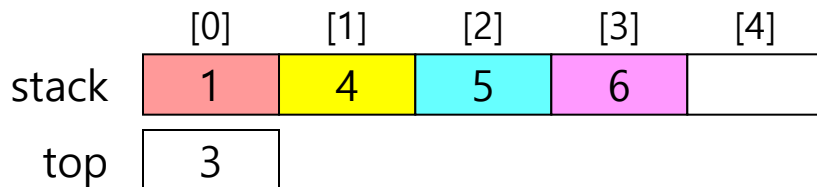
⑧ push(stack, 5);



⑪ push(stack, 8);



⑨ push(stack, 6);



**stack overflow!**

# 스택 구현 1: 순차 자료구조 [4/6]

```
#include <stdio.h>
#include <stdlib.h>

#define STACK_SIZE 5

int stack[STACK_SIZE];
int top = -1;

void printStack()
{
    int i;

    printf(" STACK [ ");
    for (i = 0; i <= top; i++)
        printf("%d ", stack[i]);
    printf("]\n");
}
```

```
void push(int value)
{
    if (top >= STACK_SIZE - 1) {
        printf("Error: Stack is Full!\n");
        return;
    }
    else stack[++top] = value;
}

int pop()
{
    if (top == -1) {
        printf("Error: Stack is Empty!\n");
        return 0;
    }
    else
        return stack[top--];
}
```

# 스택 구현 1: 순차 자료구조 [5/6]

```
void main(void)
{
    int v;

    printStack();

    printf("push(1)\n"); push(1); printStack();
    printf("push(2)\n"); push(2); printStack();
    printf("push(3)\n"); push(3); printStack();

    v = pop(); printf("pop %d\n", v); printStack();
    v = pop(); printf("pop %d\n", v); printStack();

    printf("push(4)\n"); push(4); printStack();
    printf("push(5)\n"); push(5); printStack();
    printf("push(6)\n"); push(6); printStack();
    printf("push(7)\n"); push(7); printStack();
    printf("push(8)\n"); push(8); printStack();
}
```

```
C:\WStackWDebugWStack.exe
STACK [ ]
push<1>
STACK [ 1 ]
push<2>
STACK [ 1 2 ]
push<3>
STACK [ 1 2 3 ]
pop 3
STACK [ 1 2 ]
pop 2
STACK [ 1 ]
push<4>
STACK [ 1 4 ]
push<5>
STACK [ 1 4 5 ]
push<6>
STACK [ 1 4 5 6 ]
push<7>
STACK [ 1 4 5 6 7 ]
push<8>
Error: Stack is Full!
STACK [ 1 4 5 6 7 ]
```

# 스택 구현 1: 순차 자료구조 [6/6]

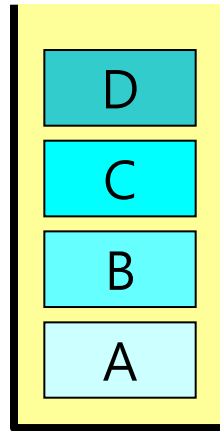
- 스택을 순차 자료구조로 구현하면,
- 장점
  - 1차원 배열을 사용하여 쉽고 단순하게 구현
- 단점
  - 크기가 고정된 배열을 사용하므로 스택의 크기 변경 어려움
  - 순차 자료구조의 단점을 그대로 가지고 있다.



# 스택 응용 1: 문자열 역순 만들기

`str1 = "ABCD";`

**for**  $i \leftarrow 0$  to length - 1  
    **push**(stack, str1[i]);



`str2 = "DCBA";`

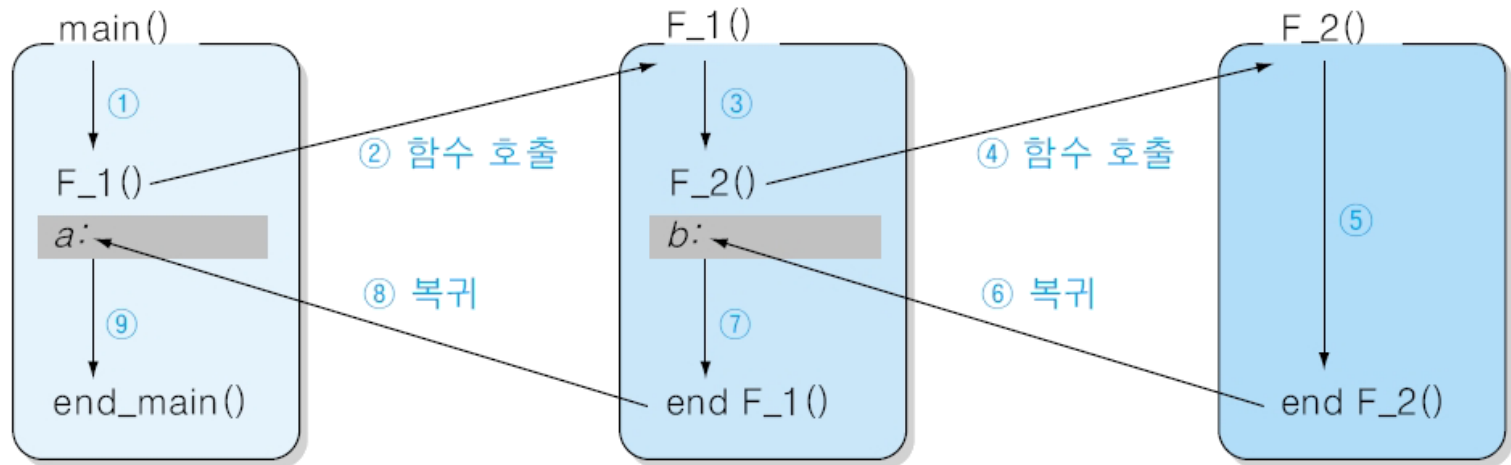
**for**  $i \leftarrow 0$  to length - 1  
    str2[i]  $\leftarrow$  **pop**(stack);

# 스택 응용 2: 시스템 스택 [1/5]

- 시스템 스택
  - 프로그램에서의 호출과 복귀에 따른 수행 순서를 관리
  - 가장 마지막에 호출된 함수가 가장 먼저 실행을 완료하고 복귀하는 후입선출 구조이므로, 후입선출 구조의 스택을 이용하여 수행순서 관리
- 함수 호출이 발생하면
  - 호출한 함수 수행에 필요한 지역변수, 매개변수 및 수행 후 복귀할 주소 등의 정보를 스택 프레임(stack frame)에 저장하여 시스템 스택에 삽입
- 함수의 실행이 끝나면
  - 시스템 스택의 top 원소(스택 프레임)를 삭제(pop)하면서 프레임에 저장되어있던 복귀주소를 확인하고 복귀
- 함수 호출과 복귀에 따라 이 과정을 반복하여 전체 프로그램 수행이 종료되면 시스템 스택은 공백 스택이 된다.

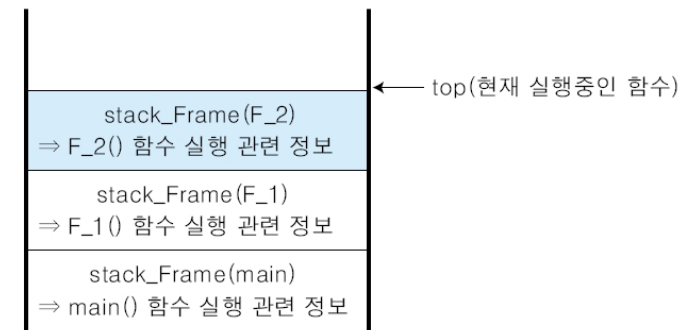
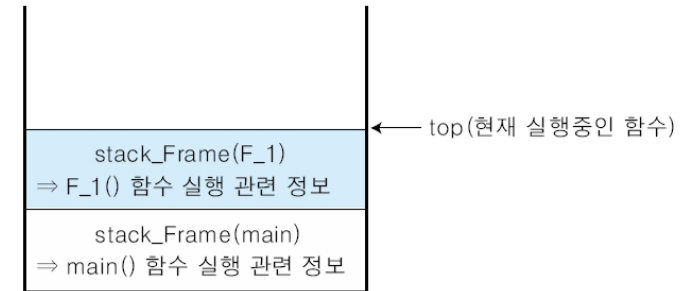
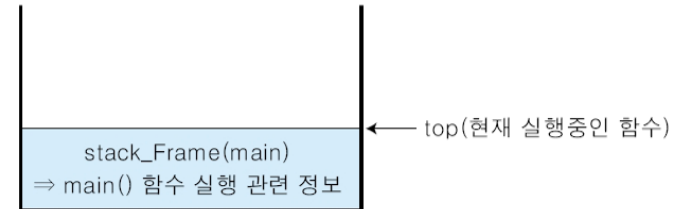
# 스택 응용 2: 시스템 스택 [2/5]

- 함수 호출과 복귀에 따른 전체 프로그램의 수행 순서



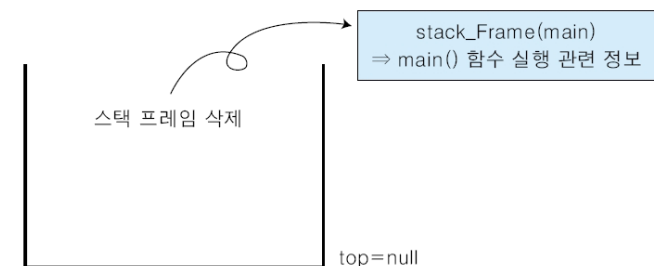
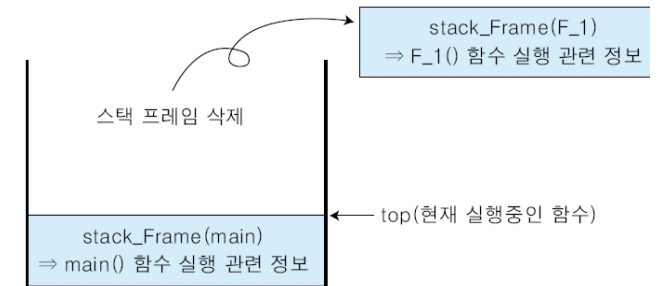
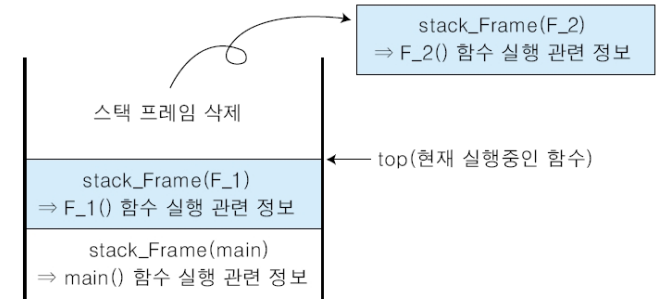
# 스택 응용 2: 시스템 스택 [3/5]

- 프로그램이 실행을 시작
  - `main()` 함수에 관련된 정보를 스택 프레임에 저장하여 시스템 스택에 삽입
- `main()` 함수 실행 중에 `F_1()` 함수 호출
  - 함수 호출과 복귀에 필요한 정보를 스택 프레임에 저장하여 시스템 스택에 삽입
  - 스택 프레임에는 함수의 수행이 끝나고 `main()` 함수로 복귀할 주소 `a`를 저장
  - 호출된 함수 `F_1()`을 실행
- `F_1()` 함수 실행 중에 `F_2()` 함수 호출
  - 함수 호출과 복귀에 필요한 정보를 스택 프레임에 저장하여 시스템 스택에 삽입
  - 스택 프레임에는 `F_1()` 함수로 복귀할 주소 `b`를 저장
  - 호출된 함수 `F_2()`를 실행



# 스택 응용 2: 시스템 스택 [4/5]

- 호출된 함수 F<sub>2</sub>( ) 실행 완료
  - 시스템 스택의 top에 있는 스택 프레임을 pop하고, F<sub>1</sub>( ) 함수의 b주소로 복귀
  - 복귀된 함수 F<sub>1</sub>( )의 b주소 이후 부분 실행
- 호출된 함수 F<sub>1</sub>( ) 실행 완료
  - 시스템 스택의 top에 있는 스택 프레임을 pop하고 main( ) 함수의 a주소로 복귀
  - 복귀된 main( ) 함수의 a주소 이후 부분 실행
- main( ) 함수 실행 완료
  - 시스템 스택의 top에 있는 스택 프레임을 pop하면 시스템 스택은 공백이 된다.



# 스택 응용 2: 시스템 스택 [5/5]

- 시스템 스택으로 설명이 가능한 원리
  - 지역변수의 영역(scope)과 존속기간(lifetime)
    - ◆ 지역변수는 스택 프레임에 위치
    - ◆ 변수 참조시 스택 top의 프레임부터 차례대로 검색
  - 자동변수(automatic variable)와 정적변수(static variable)의 차이
    - ◆ 지역변수는 해당 함수의 스택 프레임에 위치하고,  
정적변수는 main( )함수의 스택 프레임에 위치한다.
  - 재귀 호출(recursive-call)
    - ◆ 함수가 재귀 호출되면 새로운 스택 프레임이 생성된다.  
따라서, 호출 이전의 지역변수와 호출 이후의 지역 변수는  
서로 다른 스택 프레임에 위치한다.

# 스택 응용 3: 괄호 검사 [1/4]

## ● 괄호 검사

- 수식에 있는 괄호는 마지막에 열린 괄호를 먼저 닫아야 하는 후입선출 구조이므로, 스택을 이용하여 괄호를 검사할 수 있다.
- 수식을 왼쪽에서 오른쪽으로 하나씩 읽으면서 괄호 검사
  - ◆ 여는 괄호를 만나면 스택에 push
  - ◆ 닫는 괄호를 만나면 스택을 pop하여 구한 여는 괄호와 비교
    - ▶ 대응되는 괄호가 아니라면 여는 괄호의 대응되는 닫는 괄호가 빠진것임
    - ▶ 스택을 pop 하였는데 empty 상태였다면 잘못된 닫는 괄호가 심볼임
  - ◆ 수식이 끝나면 스택이 공백인지 확인
    - ▶ 공백이 아니라면 pop하여 구한 여는 괄호의 대응되는 닫는 괄호가 빠진것임

# 스택 응용 3: 괄호 검사 [2/4]

```
testPair( Expression exp )
    stack ← null;
    while (true) do {
        symbol ← getSymbol(exp);
        switch {
            case symbol = "(" or "{" or "[" :
                push(stack, symbol);
            case symbol = ")" :
                left_sym ← pop(stack);
                if (left_sym ≠ "(") then return false; // missing 'pair of left_sym' before ')'
            case symbol = "}" :
                left_sym ← pop(stack);
                if (left_sym ≠ "{") then return false; // missing 'pair of left_sym' before '}'
            case symbol = "]" :
                left_sym ← pop(stack);
                if (left_sym ≠ "[") then return false; // missing 'pair of left_sym' before ']'
            case symbol = null :
                if (isEmpty(stack)) then return true; // correct!
                else return false; // missing 'pair of pop(Stack)'
        }
    }
end testPair( )
```



# 스택 응용 3: 괄호 검사 [3/4]

## ● 괄호 검사 예 1

{ ( A + B ) - 3 } \* 5 + [ { cos ( x + y ) + 7 } - 1 ] \* 4



# 스택 응용 3: 괄호 검사 [4/4]

## ● 괄호 검사 예 2

{ ( A + B ) - 3 } \* 5 + [ { cos ( x + y ) + 7 ) - 1 ] \* 4



# 수식 표기법 [1/4]

## ● 수식 표기법의 종류

### ■ 중위표기법(infix notation)

- ◆ 연산자를 피연산자의 가운데 표기하는 방법
- ◆ 예)  $A + B$ ,  $A + B * C$

### ■ 전위표기법(prefix notation)

- ◆ 연산자를 앞에 표기하고 그 뒤에 피연산자를 표기하는 방법
- ◆ 예)  $+ A B$ ,  $+ A * B C$

### ■ 후위표기법(postfix notation)

- ◆ 연산자를 피연산자 뒤에 표기하는 방법
- ◆ 예)  $A B +$ ,  $A B C * +$

# 수식 표기법 [2/4]

- 중위 표기식을 전위 표기식으로 바꾸는 방법
  1. 수식의 각 연산자에 대해서 우선순위에 따라 괄호를 사용하여 다시 표현한다.
  2. 각 연산자를 그에 대응하는 왼쪽 괄호의 앞으로 이동시킨다.
  3. 괄호를 제거한다

**중위 표기식:  $A * B - C / D$**

**1단계:**

**2단계:**

**3단계:**

# 수식 표기법 [3/4]

- 중위 표기식을 후위 표기식으로 바꾸는 방법
  1. 수식의 각 연산자에 대해서 우선순위에 따라 괄호를 사용하여 다시 표현한다.
  2. 각 연산자를 그에 대응하는 오른쪽 괄호의 뒤로 이동시킨다.
  3. 괄호를 제거한다

**중위 표기식:  $A * B - C / D$**

**1단계:**

**2단계:**

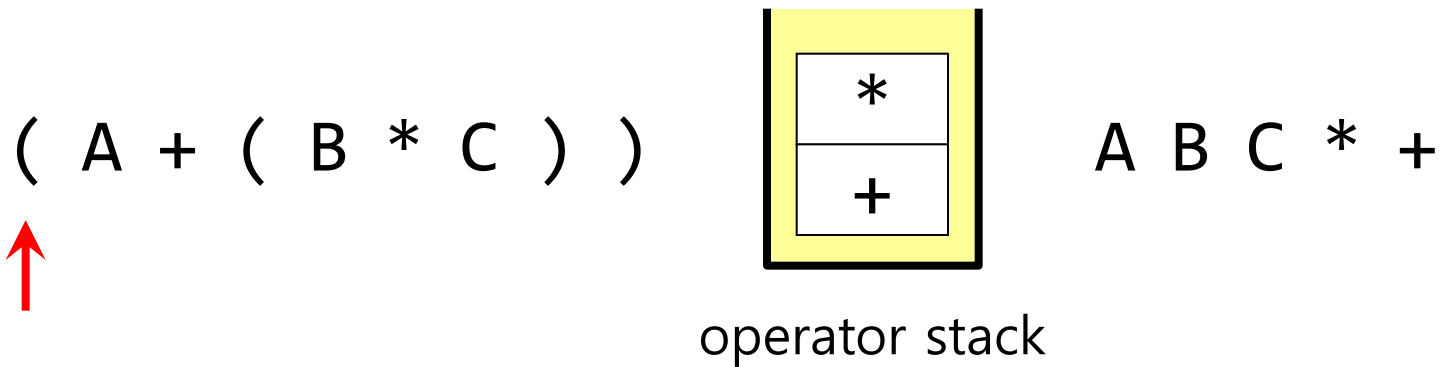
**3단계:**

# 수식 표기법 [4/4]

중위표기법(infix notation)	전위표기법(prefix notation)	후위표기법(postfix notation)
$A + B$	$+ A B$	$A B +$
$A + B * C$	$+ A * B C$	$A B C * +$
$A + B - C + D$		
$( A + B ) * C$		
$A + B * C - D$		
$( A + B ) * C - D$		

# 스택 응용 4: 후위 표기식 변환 [1/2]

- 스택을 이용한 중위 표기식의 후위 표기식 변환 방법
  1. 왼쪽 괄호를 만나면 무시하고 다음 문자를 읽는다.
  2. 피연산자를 만나면 출력한다.
  3. 연산자를 만나면 스택에 push한다.
  4. 오른쪽괄호를 만나면 스택을 pop하여 출력한다.
  5. 수식이 끝나면, 스택이 공백이 될 때까지 pop하여 출력한다.



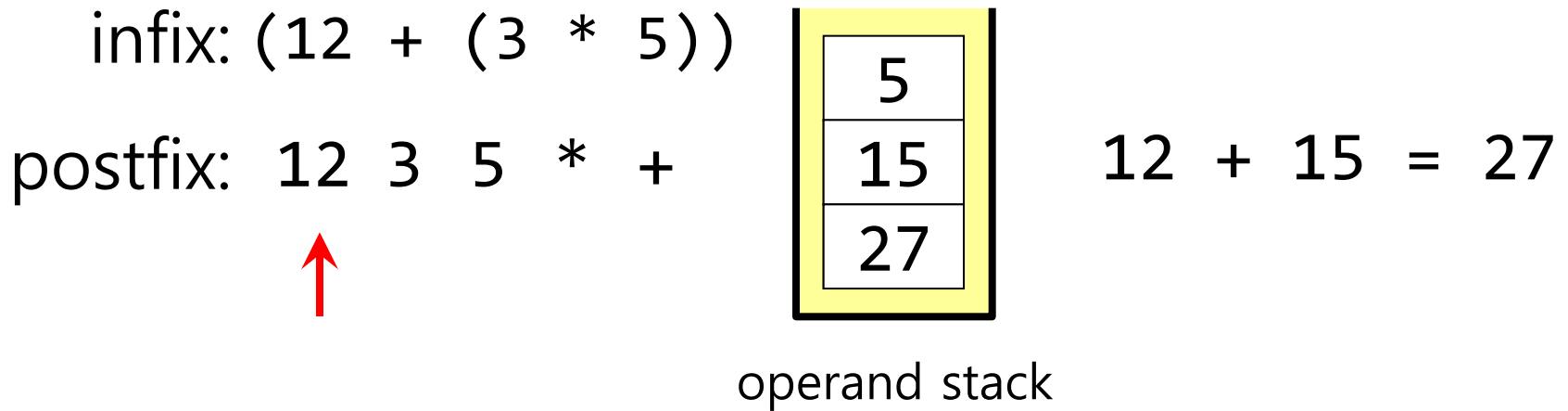
# 스택 응용 4: 후위 표기식 변환 [2/2]

```
infix_to_postfix(Expression exp)
  while (true) do {
    symbol ← getSymbol(exp);
    case {
      symbol = operand :           // 피연산자 처리
        print(symbol);
      symbol = operator :         // 연산자 처리
        push(stack, symbol);
      symbol = ")" :              // 오른쪽괄호 처리
        print(pop(Stack));
      symbol = null :             // 수식의 끝 처리
        while (not isEmpty(stack)) do
          print(pop(Stack));
    }
  }
end infix_to_postfix( )
```



# 스택 응용 5: 후위 표기식의 계산 [1/2]

- 스택을 이용한 후위 표기식의 계산 방법
  1. 피연산자를 만나면 스택에 push 한다.
  2. 연산자를 만나면 필요한 만큼의 피연산자를 스택에서 pop하여 연산하고, 연산결과를 다시 스택에 push 한다.
  3. 수식이 끝나면, 마지막으로 스택을 pop하여 출력한다.



# 스택 응용 5: 후위 표기식의 계산 [2/2]

```
evalPostfix(Expression exp)
  while (true) do {
    symbol ← getSymbol(exp);
    case {
      symbol = operand :           // 피연산자 처리
        push(stack, symbol);
      symbol = operator :         // 연산자 처리
        opr2 ← pop(stack);
        opr1 ← pop(stack);
        result ← opr1 op(symbol) opr2;
        push(stack, result);
      symbol = null :             // 후위수식의 끝
        print(pop(stack));
    }
  }
end evalPostfix( )
```

# 요약

- 스택: 한쪽 끝에서만 자료를 삽입, 삭제하는 선형 자료구조
  - 후입선출(LIFO, Last-In-First-Out)
  - 스택 top : 삽입/삭제가 이루어지는 위치
  - push(value) : 원소의 삽입 연산, pop : 원소의 삭제 연산
- 스택의 구현
  - 순차 자료구조로 구현한 스택은 1차원 배열을 사용하여 손쉽게 구현
  - 배열을 이용하여 구현한 스택은 크기 변경이 어려움
- 스택의 응용
  - 수식 표기법의 종류: 중위표기법(infix notation), 전위표기법(prefix notation), 후위표기법(postfix notation) 이 있다.
  - 스택을 이용하여 중위표기식을 후위표기식으로 바꿀 수 있으며, 후위표기식은 스택을 이용하여 쉽게 계산할 수 있다.