4. 스택 (Stack)

한국항공대학교 안준선

주요 내용

- 스택의 개념과 구현
- 추상 자료형의 장점과 구현 방법
- 문맥 자유 문법(Context Free Grammar)
- 수식 표기법과 관련 알고리즘
- 함수 호출 구현을 위한 스택의 사용

4. 스택

- abstract data type
  - 자료집합의 기능(명세, specification)과 구현의 분리
  - 자료 구조의 사용을 위하여 자료 구조 내부의 구현을 이해할 필요가 없음.(→ 이식성, 가독성 향상)
  - 자료 집합 접근을 위한 일정한 방법 제공 (함수, 프로시져, 메쏘드)
  - 접근 방법의 제공은 다양한 방법으로 구현 가능
  - 일반적으로 제공되는 연산은 상수 시간에 수행이 가능
  - 예 : 스택, 큐 등
- 스택(Stack)
  - 나중에 저장된 데이터가 먼저 나오도록 순서가 정해진 저장구조
    : LIFO (Last In First Out)
  - 제공 연산
    - init : ( ) ⇢ $Stack^T$
    - pop : $Stack^T$ ⇢ (T | Empty) x $Stack^T$)
    - push : $Stack^T$ x T ⇢ $Stack^T$ (| Overflow)

배열로 구현하는 스택

- 배열로 구현하는 스택
  - 특징
    - 미리 스택의 크기가 정해짐.
      → 미리 정해진 스택 크기를 초과해서 원소가 push 될 때에는 overflow가 발생하거나, 스택의 크기를 재 설정해야 함.
    - 스택의 제일 위를 가리키는 변수에 의해 push와 pop을 수행
  - 스택을 위한 기억공간의 할당

```
#define MAX 10
int stack[MAX];
int top;
```

    - 최대 MAX개의 원소를 저장할 수 있음.
    - top : 다음에 저장될 위치 또는 가장 최근에 저장된 위치를 가리킴.
      (후자의 경우 : top=-1 ⇢ empty, top= MAX-1 ⇢ stack_full)

배열로 구현하는 스택

- 배열의 초기화 (empty 스택으로 만듬)

```
void init_stack(void) {
    top = -1;
}
```

- 스택에 원소 집어넣기 : push

```
int push(int t) {
    if (top >= MAX - 1) {
        printf("\n    Stack overflow.");
        return -1;
    }
    stack[++top] = t;
    return t;
}
```

배열로 구현하는 스택

- 스택에 저장된 원소를 빼오기 : pop

```
int pop(void) {
    if (top < 0) {
        printf("\n    Stack underflow.");
        return -1;
    }
    return stack[top--];
}
```

- 기타 함수 : 스택의 모든 원소를 출력

```
void print_stack(void) {
    int i;
    printf("\n  Stack contents : Top --> Bottom\n");
    for (i = top; i >= 0; i--)
        printf("%-6d", stack[i]);
}
```

연결 리스트로 구현하는 스택

- 연결 리스트를 이용한 스택의 구현
  - 특징
    - 배열과 달리 동적인 할당을 사용하여 구현되므로 메모리를 원소의 개수에 비례해서 사용하게 된다.
    - push 동작 : 헤드 노드 다음에 새로운 원소를 삽입
    - pop 동작 : 헤드 노드 다음 노드를 삭제하면서 해당 노드의 값을 반환
  - 스택 노드의 선언 : 연결리스트와 동일

```
typedef struct _node {
    int key;
    struct _node *next;
} node;

node *head, *tail;
```

연결 리스트로 구현하는 스택

- 스택의 초기화

```
void init_stack(void) {
    head = (node*)malloc(sizeof(node));
    tail = (node*)malloc(sizeof(node));
    head->next = tail;
    tail->next = tail;
}
```

- push

```
int push(int k) {
    node *t;
    if ((t = (node*)malloc(sizeof(node))) == NULL) {
        printf("\n    Out of memory...");
        return -1;
    }
    t->key = k; t->next = head->next; head->next = t;
    return k;
}
```

연결 리스트로 구현하는 스택

- pop

```
int pop(void) {
    node *t;
    int i;
    if (head->next == tail)  /* if empty */ {
        printf("\n    Stack underflow.");
        return -1;
    }
    t = head->next; i = t->key;
    head->next = t->next; free(t);
    return i;
}
```

연결 리스트로 구현하는 스택

- 스택 비우기

```
void clear_stack(void) {
    node *t, *s;
    t = head->next;
    while (t != tail) {
        s = t; t = t->next;
        free(s);
    }
    head->next = tail;
}
```

스택

- 추상 자료형의 구현

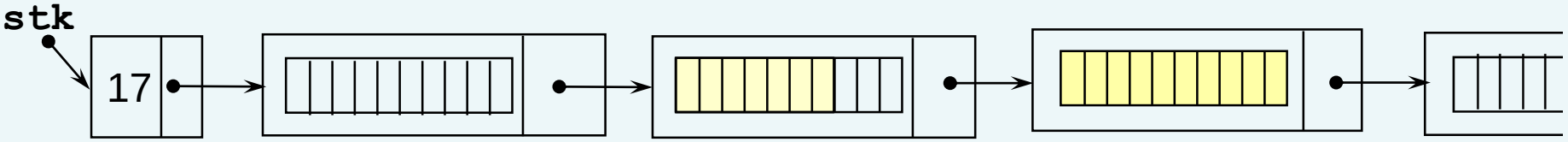
```
typedef struct _stack {
    int top;
    int data[MAX];
} stack;

stack *init_stack();
int pop(stack *);
int push(stack *, int v);
void clear_stack(stack *);
....

stack *stk1 = init_stack();
push(stk1, 1);
push(stk1, 2);
v = pop(stk1);
```

```
typedef struct _stack {
    node *head;
} stack;
```

스택

- 배열과 리스트의 비교
  - 배열의 단점
    - 처음에 충분히 많은 메모리를 할당해야 한다.
  - 리스트의 단점
    - 포인터 값 저장 공간이 필요
    - 원소의 push와 pop 연산 시에 메모리의 할당과 반환이 이루어져야 한다. (C의 경우 많은 계산 시간이 소요된다.)
  - 대안들
    - 하나의 노드에 여러 개의 원소를 저장
      - » 각 노드 내에서는 배열을 사용한 구현법을 이용하고, 노드의 용량이 부족하면 새로운 노드를 할당한다.
      - » 포인터로 인한 기억 장소의 낭비와 메모리 연산의 횟수를 줄임
    - pop 연산 시에 삭제되는 메모리를 재사용.
    - 프로그래밍은 더 복잡해짐

하나의 노드에 여러 개의 원소 저장




```
typedef struct {
    int num;
    node *head;
} stack;

typedef struct {
    int value[10];
    node *next;
} node;
```

```
void push (stack *stk, int v) {
    if (stk->num%10 == 0) {
        node *n = (node *)
            malloc(sizeof(node));
        n->value[0] = v;
        stk->num++;
        n->next = stk->head->next;
        stk->head->next = n;
    }
    else {
        int i = stk->num%10;
        stk->head->next->value[i] = v;
        stk->num++;
    }
}
```

스택의 응용 : 수식 계산

- 문맥 자유 문법(CFG, Context Free Grammar)
  - 문법: 특정 형태의 문자열의 집합을 표현하는 방법
  - 생성규칙들의 집합으로 표현
  - 생성규칙의 형태

<논터미널> → <형태정의> | <형태정의> | .... | <형태정의>

- <논터미널> : 특정 문자열들의 집합의 원소를 나타내는 기호
- <형태정의> : 터미널(최종적인 문자) 또는 논터미널을 0개 이상 연결 (0개일때는 ε로 표현)

* 논터미널을 재귀적으로 사용 가능

문맥자유문법의 예

- 예

  <biDigit> → 1 | 0
  <twoDigit> → <biDigit><biDigit>
  <biNum> → <biNum><biDigit> | <biDigit>
  <evenBiDigit> → 0 | <biNum>0

  <char> → a | b | c | d
  <myS> → ε | <char> | a<myS>a | b<myS>b
  | c<myS>c | d<myS>d

스택의 응용 : 수식 계산

- 수식의 표기법
  - 중위법(infix notation) : 오퍼랜드 사이에 연산자가 위치

```
<op> → + | - | * | / | %
<exp> →  <id> | <num>
        |  <exp> <op> <exp> | (exp)
```

ex) A * ((B+C) / (D-E))

  - 후위법(postfix notation) : 오퍼랜드 뒤에 연산자가 옴

```
<op> → + | - | * | / | %
<exp> → <id> | <num>
        |  <exp> <exp> <op>
```

ex) A B C + D E - / *

스택의 응용

- 전위법 (prefix notation) : 오퍼랜드 전에 연산자가 옴

```
<op> → + | - | * | / | %
<exp> → <id> | <num>
        |  <op> <exp> <exp>
```

ex) * A-+ B C - D E

- 후위법과 전위법은 연산자의 우선순위나 괄호가 필요 없음
  - 후위법의 경우 오퍼레이터는 순서대로 계산하면 됨.
    - 오퍼레이터가 나오면 그 전에 계산된 두 개의 오퍼랜드를 사용
  - 전위법의 경우 연산자 다음에 두 개의 숫자가 위치하면 계산을 수행함

스택의 응용

- 후위표기법 수식의 계산
  - 방법 : 스택을 사용
    - » 숫자는 스택에 push 한다.
    - » 연산자를 만나면 스택에서 두 개의 숫자를 pop하여 연산을 수행하고 결과를 스택에 push한다.
  - 프로그램

```
int calc(char *p) {
    int i;
    init_stack();
    while (*p) {
        if (*p >= '0' && *p <= '9') {
            i = 0;
            do {
                i = i * 10 + *p - '0';
                p++;
            } while (*p >= '0' && *p <= '9');
            push(i);
        }
```

스택의 응용

- 프로그램(계속)

```
        else if (*p == '+') {
            push(pop() + pop());
            p++;
        }
        else if (*p == '*') {
            push(pop() * pop());
            p++;
        }
        else if (*p == '-') {
            i = pop();
            push(pop() - i);
            p++;
        }
        else if (*p == '/') {
            i = pop();
            push(pop() / i);
            p++;
        }
        else p++;
    }
    return pop();
}
```

스택의 응용

- 중위표기법을 후위 표기법으로 변환하는 방법 : 모든 연산에 대하여 괄호가 표기되었을 경우
  - ')' 괄호와 하나의 연산자가 대응하므로, ')'가 나오면 최근의 연산을 수행하면 됨.
  - 변환 방법
    1. '(' 문자는 무시하고 넘어감
    2. 피연산자(수, 문자)는 그대로 출력
    3. 연산자는 스택에 push 함
    4. ')'를 만나면 연산자를 스택에서 pop 해서 출력함.
  - 실행 예 : (A+(B*C))

출력	스택 (상단⇢하단)
A	
A	+
A B	+
A B	*+
A B C	*+
A B C *	+
A B C * +	

스택의 응용

- 프로그램 : postfix1
  - » 가정
    - 양의 정수에 대한 수식이고 연산자는 +, -, *, / 만 사용
    - 모든 연산은 괄호로 묶여 있음

```
void postfix1(char *dst, char *src) {
    char c;
    init_stack();
    while (*src)
        if (*src == ')') {
            *dst++ = pop(); *dst++ = ' '; src++;
        }
        else if (*src == '+' || *src == '-' ||
                 *src == '*' || *src == '/')
            push(*src); src++;
        else if (*src >= '0' && *src <= '9') {
            do {
                *dst++ = *src++;
            } while (*src >= '0' && *src <= '9');
            *dst++ = ' ';
        }
        else  // *src is a blank
            src++;
    *dst = 0;
}
```

스택의 응용

- 중위표기법을 후위 표기법으로 변환하는 방법 : 괄호가 완전하지 않은 경우
  - 연산에 우선 순위를 부여하여 우선순위가 높은 연산이 먼저 수행되도록(먼저 출력 되도록) 한다.
  - 괄호는 우선순위가 가장 높으므로 ‘)’ 를 만나면 최근 ‘(’ 다음 연산을 모두 수행하도록 한다.
  - 변환 방법
    1. ‘(’ 문자는 스택에 push
    2. ‘)’를 만나면 ‘(’를 만날 때까지 연산자를 스택에서 pop 해서 출력함.
    3. 연산자는 스택에서 우선순위가 낮은 연산자를 만날 때가지 모두 pop 하여 출력하고 자신을 push 한다.
    4. 피연산자는 그대로 출력

스택의 응용

- 실행 예 : (2*(3+6/2)+2)/4 + 3

	출력	스택 (상단⇢하단)
(		(
2	2	(
*	2	* (
(	2	(* (
3	2 3	(* (
+	2 3	+ (* (
6	2 3 6	+ (* (
/	2 3 6	/ + (* (
2	2 3 6 2	/ + (* (
)	2 3 6 2 / +	* (
+	2 3 6 2 / + *	+ (
2	2 3 6 2 / + * 2	+ (
)	2 3 6 2 / + * 2 +	
/	2 3 6 2 / + * 2 +	/
4	2 3 6 2 / + * 2 + 4	/
+	2 3 6 2 / + * 2 + 4 /	+
3	2 3 6 2 / + * 2 + 4 / 3	+
끝	2 3 6 2 / + * 2 + 4 / 3 +	

스택의 응용

- 프로그램

```
int get_stack_top(void) {
    return (top < 0) ? -1 : stack[top];
}

int is_stack_empty(void){
    return (top < 0);
}

int is_operator(int k) {
    return (k=='+' || k=='-' || k=='*' || k=='/');
}

int precedence(int op) {
    if (op == '(') return 0;
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    else return 3;
}
```

스택의 응용

- 프로그램 (계속)

```
void postfix(char *dst, char *src){
    char c;
    init_stack();
    while (*src) {
       if (*src == '(') {
           push(*src); src++;
       }
       else if (*src == ')') {
           while (get_stack_top() != '(') {
              *dst++ = pop(); *dst++ = ' ';
           }
           pop(); src++;
       }
       else if (is_operator(*src)) {
           while (!is_stack_empty() &&
                  precedence(get_stack_top())>=
                   precedence(*src)) {
              *dst++ = pop(); *dst++ = ' ';
           }
           push(*src); src++;
       }
```

스택의 응용

- 프로그램 (계속)

```
        else if (*src >= '0' && *src <= '9') {
            do {
               *dst++ = *src++;
            } while (*src >= '0' && *src <= '9');
            *dst++ = ' ';
        }
        else src++;
    }
    while (!is_stack_empty()) {
        *dst++ = pop(); *dst++ = ' ';
    }
    dst--; *dst = 0;
}
```

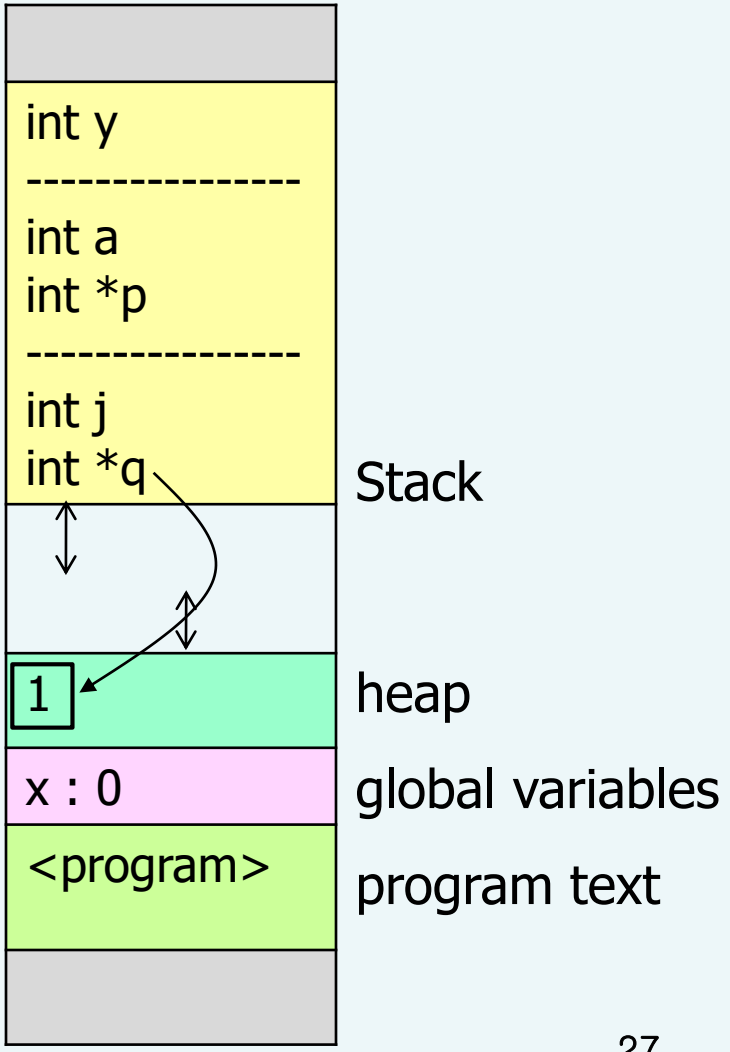
스택의 응용

- 스택의 응용 2 : System stack
  - system stack
    - stack used by a program at run-time to process function calls
  - activation record (stack frame)
    - contains ...
      - » parameters of the invoking function
      - » a return address
      - » a pointer to the previous stack frame
      - » local variables
    - if a function invokes another function, new stack frame is pushed.
    - if this function returns, current stack frame is poped resuming program counters, stack pointers, etc.

스택의 응용

- System stack

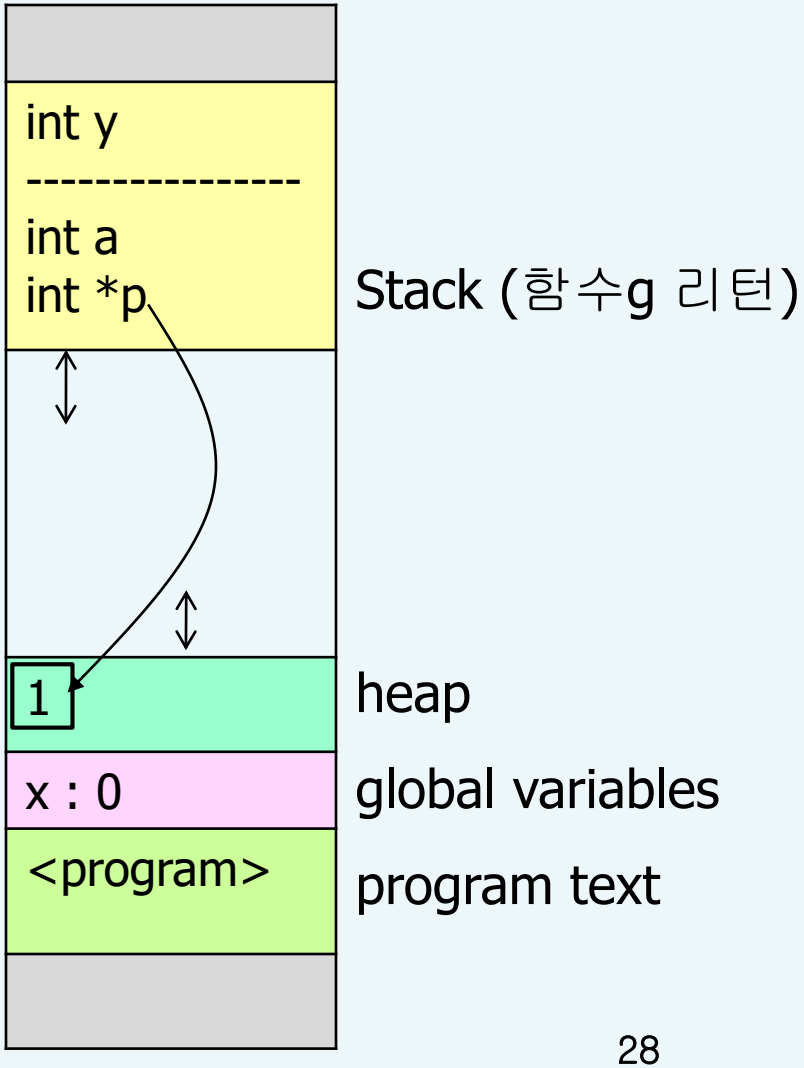
```
int x;

int f(int);
int *g(int);
main() {
        int y = x+1;
        f(y);
        g(y+1);
}
f(int a) {
        int *p;
        p = g(a);
}

int *g(int j) {
        int *q = malloc(4);
        *q = j;
        return q;
}
```



스택의 응용

- System stack

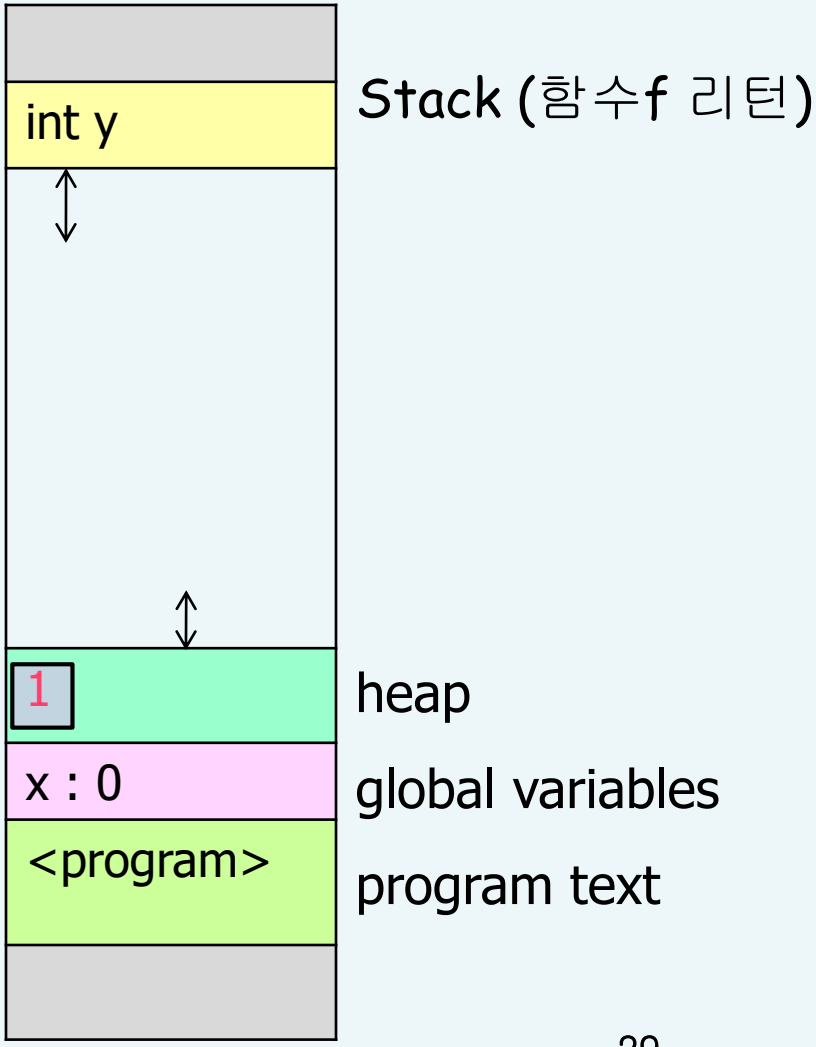
```
int x;

int f(int);
int *g(int);
main() {
        int y = x+1;
        f(y);
        g(y+1);
}
f(int a) {
        int *p;
        p = g(a);
}

int *g(int j) {
        int *q = malloc(4);
        *q = j;
        return q;
}
```



스택의 응용

- System stack

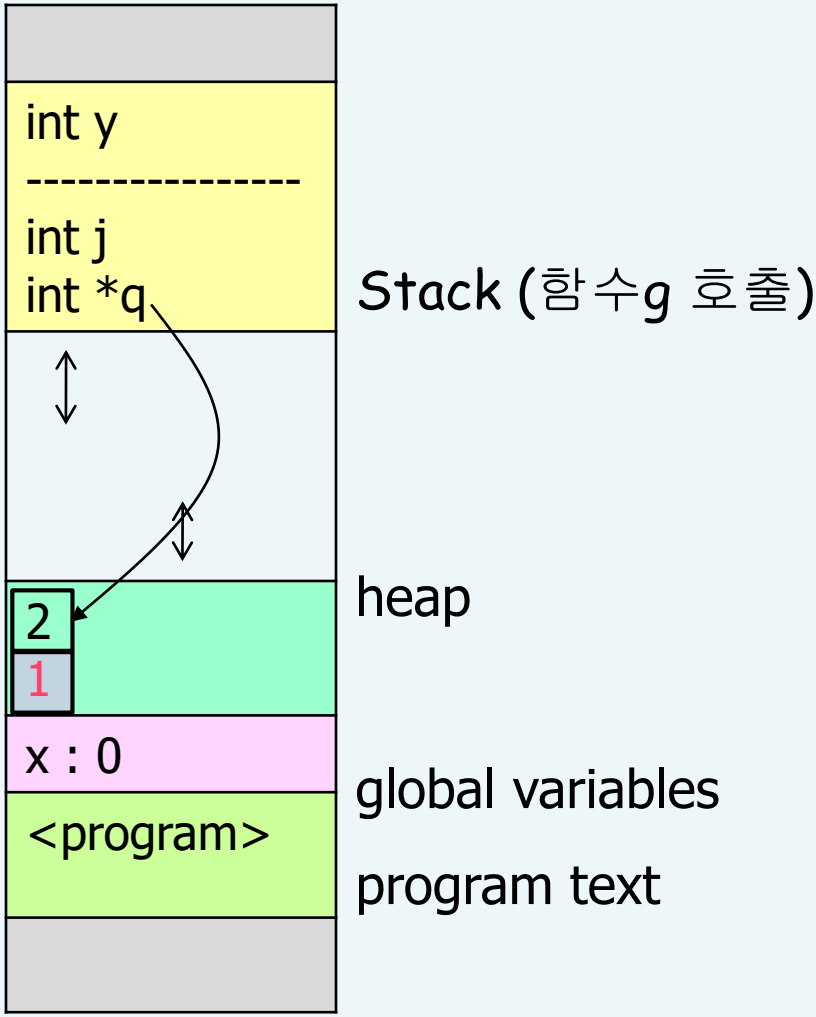
```
int x;

int f(int);
int *g(int);
main() {
        int y = x+1;
        f(y);
        g(y+1);
}
f(int a) {
        int *p;
        p = g(a);
}

int *g(int j) {
        int *q = malloc(4);
        *q = j;
        return q;
}
```



스택의 응용

- System stack

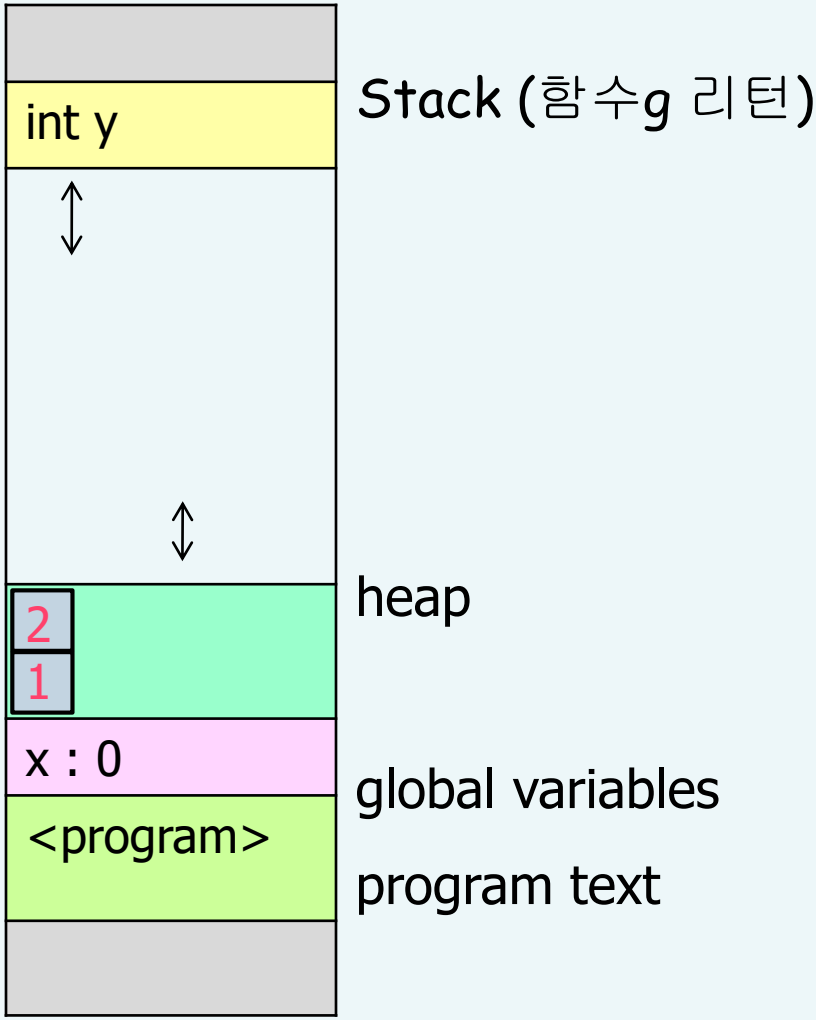
```
int x;

int f(int);
int *g(int);
main() {
        int y = x+1;
        f(y);
        g(y+1);
}
f(int a) {
        int *p;
        p = g(a);
}

int *g(int j) {
        int *q = malloc(4);
        *q = j;
        return q;
}
```



스택의 응용

- System stack

```
int x;

int f(int);
int *g(int);
main() {
        int y = x+1;
        f(y);
        g(y+1);
}
f(int a) {
        int *p;
        p = g(a);
}

int *g(int j) {
        int *q = malloc(4);
        *q = j;
        return q;
}
```



정리

- 스택의 개념과 구현
- 추상 자료형의 장점과 구현 방법
- 문맥 자유 문법(Context Free Grammar)
- 수식 표기법과 관련 알고리즘
- 함수 호출 구현을 위한 스택의 사용