Clickseo.com

C Programming

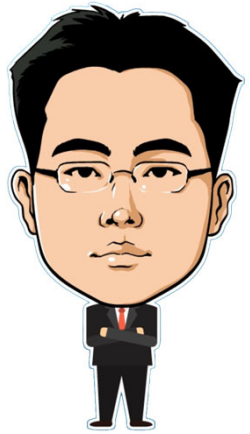
동적 메모리 할당

(Dynamic Memory Allocation)

Seo, Doo-Ok

Clickseo.com

clickseo@gmail.com

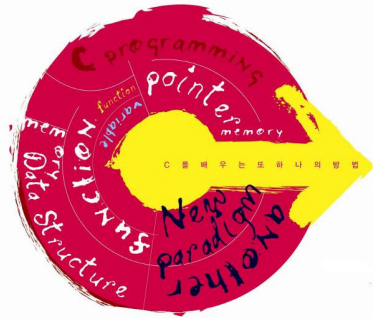


목 차

- 동적 메모리 할당
- 2차원 배열과 동적 메모리 할당
- 단순 연결 리스트

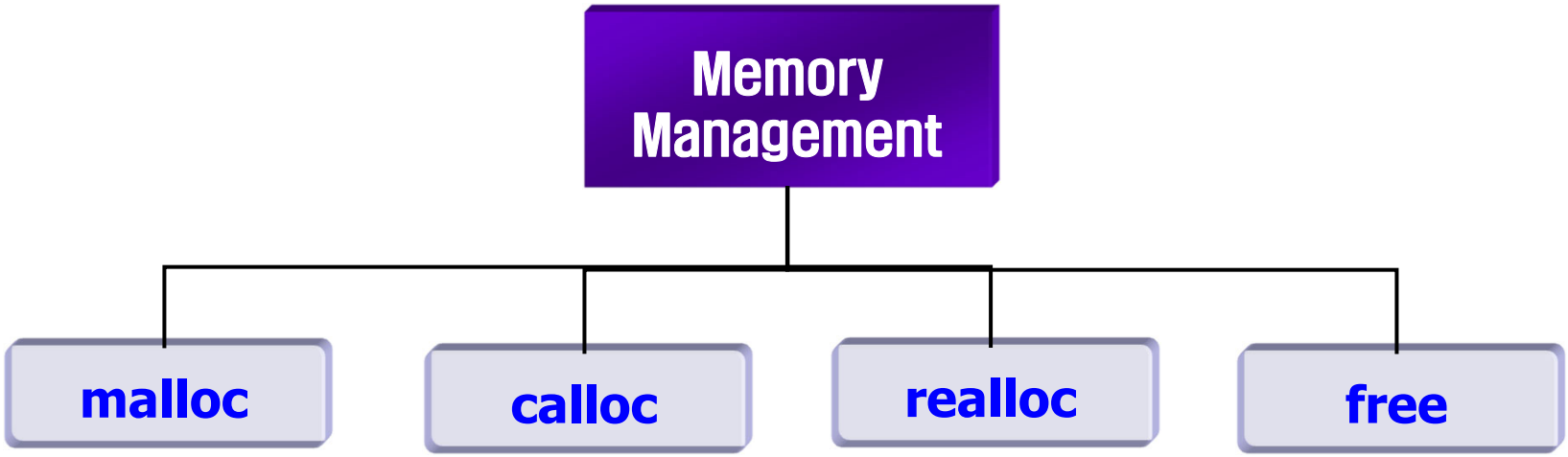
동적 메모리 할당

- **동적 메모리 할당**
  - 메모리 할당 함수 : malloc
  - 인접 메모리 할당 함수 : calloc
  - 메모리 재할당 함수 : realloc
  - 메모리 해제 함수 : free
- 2차원 배열과 동적 메모리 할당
- 단순 연결 리스트



동적 메모리 할당 (1/7)

- **동적 메모리 할당**
  - 동적 메모리관리에 쓰이는 4가지 함수
    - 표준 라이브러리 헤더 파일 **<stdlib.h>** 에서 찾을 수 있다.
    - 메모리 할당 : **malloc, calloc, realloc**
    - 메모리 해제 : **free**

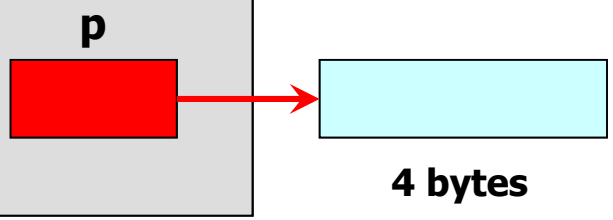


동적 메모리 할당 (2/7)

- **메모리 할당 : malloc**
  - malloc 함수는 매개변수로 필요한 메모리의 바이트의 수를 가지며, 그 바이트 수를 수용할 수 있는 크기의 메모리 블록을 할당한다.
    - 할당된 메모리의 첫 번째 바이트를 void 포인터로 되돌린다.
    - 할당된 메모리는 초기화되어 있지 않다(쓰레기 값).

```
void  * malloc(size_t  size);
```

```
int         *p = NULL;

p = (int *)malloc(sizeof (int) )
if (p == NULL)
{
            printf("메모리 할당 실패!!! ₩n");
            exit(100);
}
```

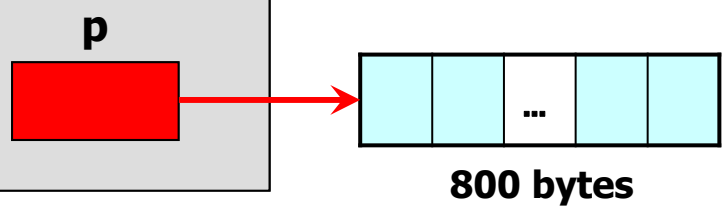


동적 메모리 할당 (3/7)

- **인접 메모리 할당 : calloc**
  - 일반적으로 배열을 위한 메모리 할당에 주로 쓰인다.
    - 특정 크기의 요소들을 배열로 담을 수 있을 만큼 연속적인 메모리 블록을 할당
    - 할당된 배열의 첫 번째 요소를 가리키는 포인터를 반환
    - 할당된 메모리를 초기화(즉, 할당된 메모리의를 0 으로 초기화)

```
void  * calloc(size_t  element-count,  size_t  element-size);
```

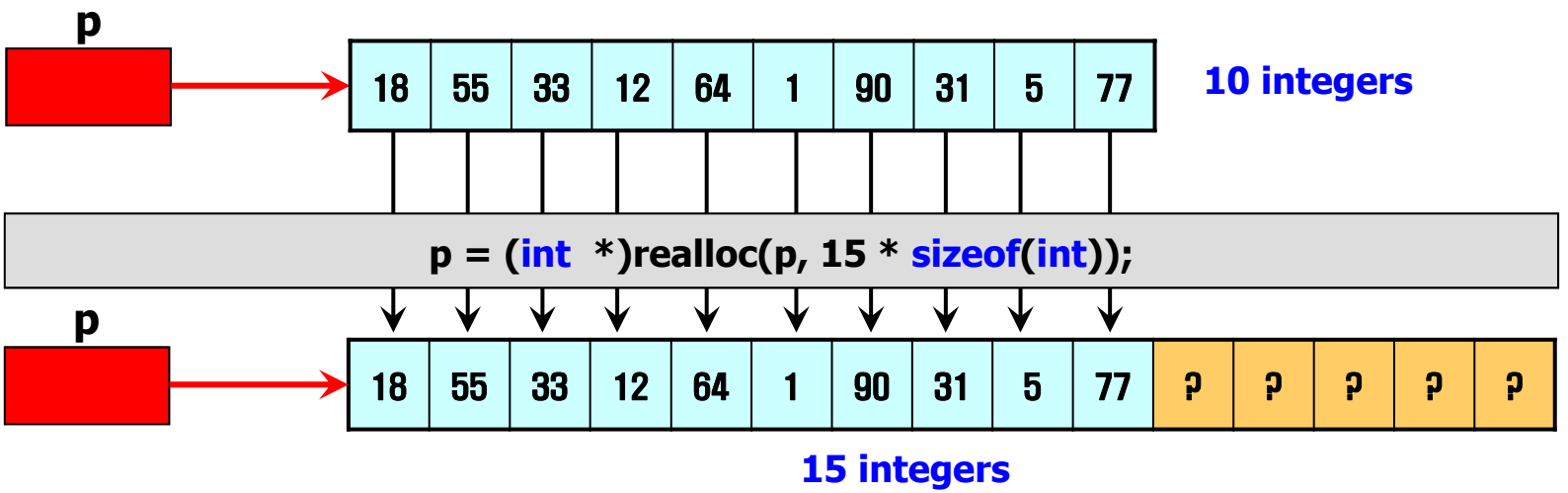
```
int         *p = NULL;

p = (int *)calloc(200, sizeof (int) )
if (p == NULL)
{
            printf("메모리 할당 실패!!! ₩n");
            exit(100);
}
```



동적 메모리 할당 (4/7)

- **메모리 재할당 : realloc**
  - 메모리 재할당에 쓰인다.

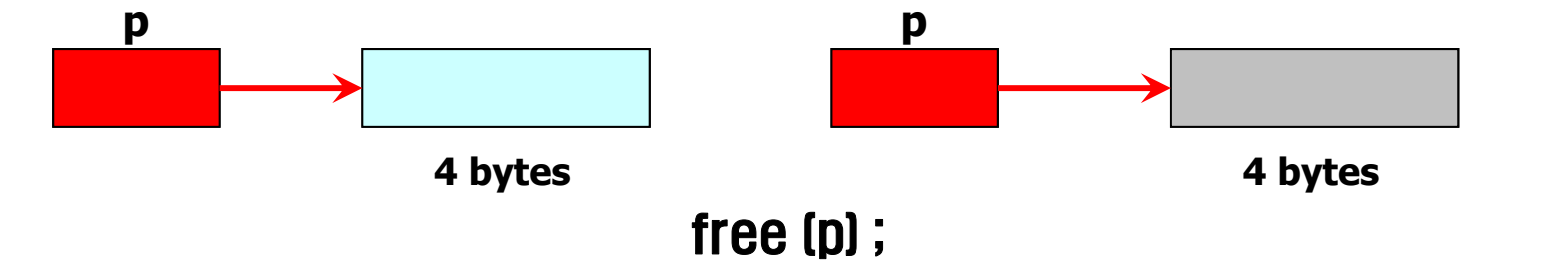
```
void  * realloc(void  *p, size_t  new_Size);
```

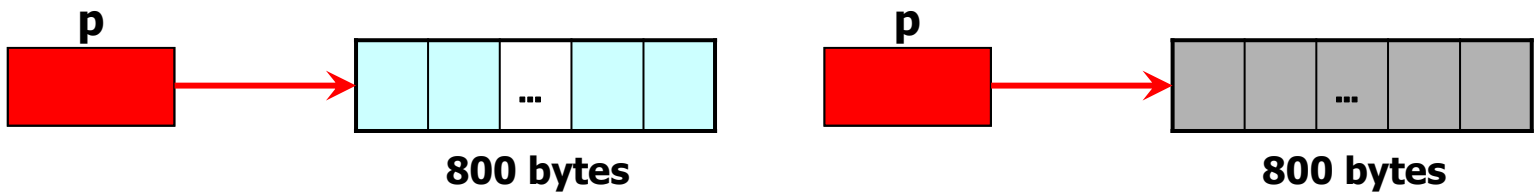


동적 메모리 할당 (5/7)

- **메모리 해제 : free**
  - malloc, calloc, realloc에 의해 할당된 메모리가 더 이상 필요 없을 때는 free 함수를 사용하여 해제한다.

```
void  free(void  *p);
```

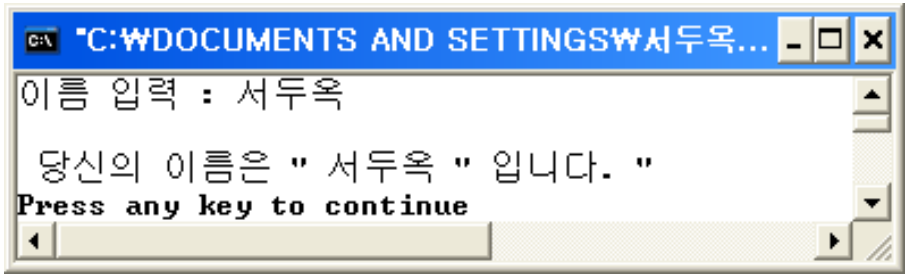


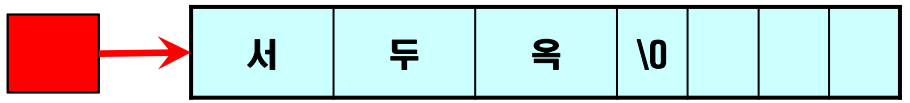


동적 메모리 할당 (6/7)

프로그램 예제 : 동적 메모리 할당 -- malloc와 free

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

int main(void)
{
        // char name[10];
        char    *name = NULL;

        name = (char *)malloc(10);
        if(name == NULL)
        {
                printf("메모리 할당 실패!!! \n");
                exit(100);
        }

        printf("이름 입력 : ");
        gets(name);

        printf("\n 당신의 이름은 \" %s \" 입니다. \" \n", name);

        free(name);

        return 0;
}
```





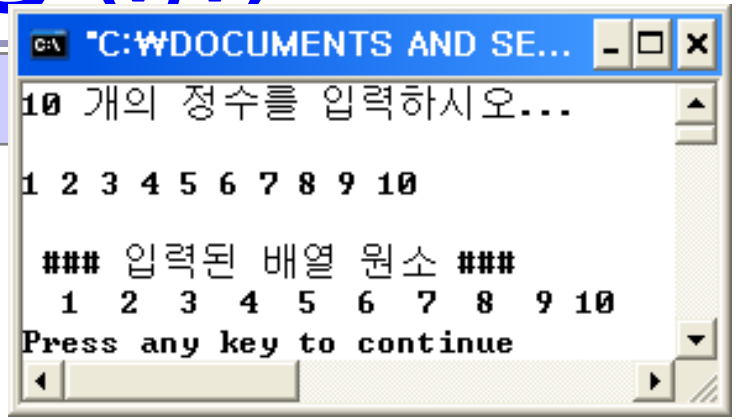
name

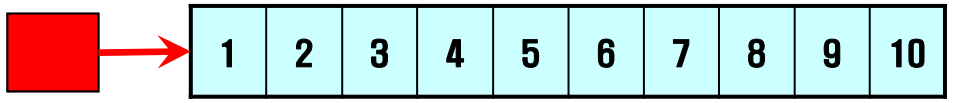
10 bytes

동적 메모리 할당 (7/7)

프로그램 예제 : 1차원 동적 배열 -- calloc

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

#define SIZE 10

int main(void)
{
        int     i, *arr = NULL;         // int   array[SIZE];

        // arr = (int *)malloc(SIZE * sizeof(int));
        arr = (int *)calloc(SIZE, sizeof(int));
        if(arr == NULL)
        {
                puts("메모리 할당에 실패!!! ");
                exit(100);
        }
        printf("%d 개의 정수를 입력하시오... \n\n", SIZE);
        for(i = 0; i < SIZE; i++)
                scanf("%d",  arr + i );

        printf("\n ### 입력된 배열 원소 ### \n");
        for(i = 0; i < SIZE; i++)
                printf("%3d",  *(arr + i) );
        printf("\n");

        free(arr);

        return 0;
```



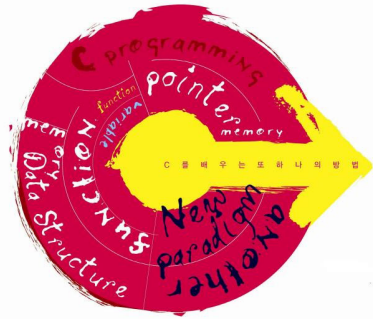


arr

40 bytes

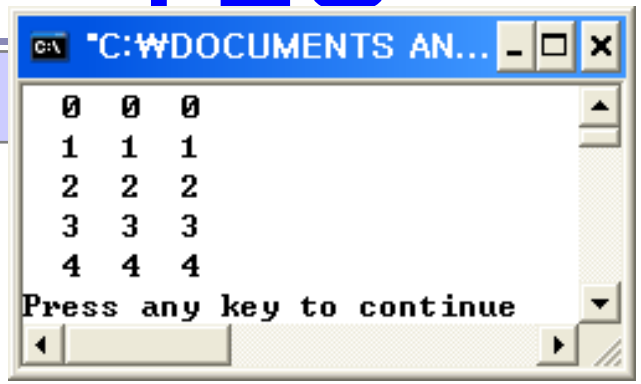
2차원 배열과 동적 메모리 할당

- 동적 메모리 할당 함수
- **2차원 배열과 동적 메모리 할당**
  - main 함수 - 명령행 인자
- 단순 연결 리스트



2차원 배열과 동적 메모리 할당

프로그램 예제 : 2차원 동적 배열 -- calloc



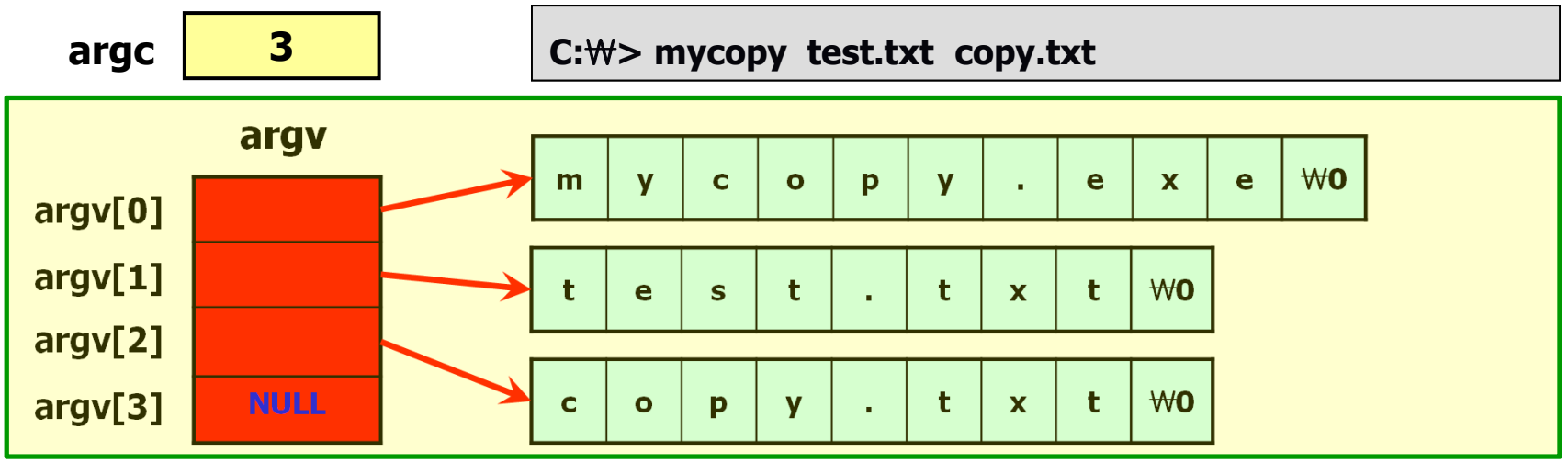

```
#include <stdio.h>
#include <stdlib.h>
#define ROW     5
#define COL     3
int main(void)
{
        int     **table = NULL;         // int   table[ROW][COL];
        int     i, j;

        table = (int **)calloc(ROW + 1, sizeof(int *) );
        for(i = 0; i < ROW; i++)
                table[i] = (int *)calloc(COL, sizeof(int) );
        for(i=0; i<ROW; i++)
        {
                for(j=0; j<COL; j++)
                        printf("%3d", table[i][j] + i );
                printf("\n");
        }

        for(i = 0; i < ROW; i++)
                free(table[i]);
        free(table);

        return 0;
}
```

main 함수 – 명령행 인자 (1/4)

- **명령행 인자**

```
int  main( int  argc, char  *argv[], char  **env )
int  main( int  argc, char  **argv, char  **env )
```

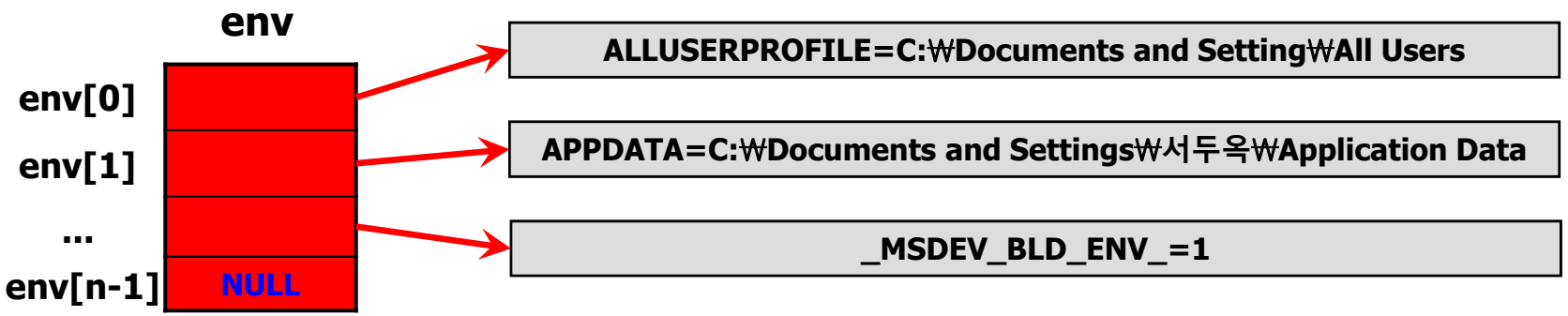
- **argc** : 명령행 인자의 개수
- ****argv** : 명령행 인자(문자열)가 있는 메모리의 시작 주소
- ****env** : 현재 시스템에 설정되어 있는 환경 변수



main 함수 – 명령행 인자 (2/4)

- **명령행 인자** : 환경 변수
  - ****env** : 현재 시스템에 설정되어 있는 환경 출력

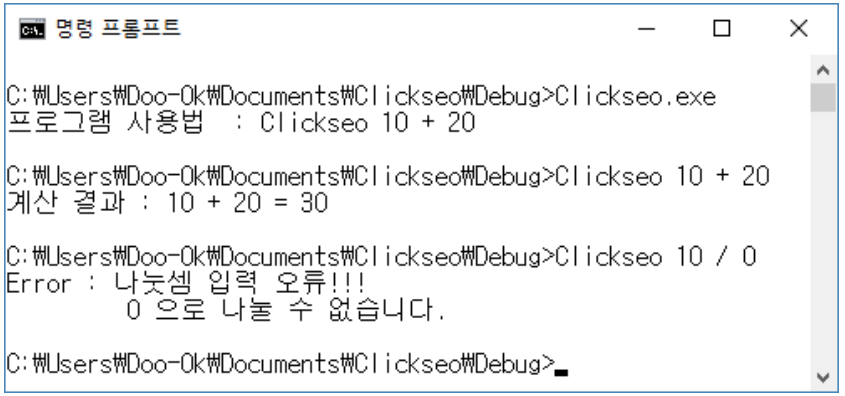
```
#include <stdio.h>

int main ( int argc, char **argv, char **env )
{
        while( *env )
                puts( *env++ );

        return 0;
}
```



main 함수 – 명령행 인자 (3/4)

예제 10-4 : 명령행 인자 – 사칙 연산 (1/2)

```
#include <stdio.h>
#include <stdlib.h>   // exit, atoi

int main( int argc, char *argv[] )
{
        if ( argc < 4 )
        {
                printf("프로그램 사용법 : Clickseo 10 + 20 \n");
                exit(1);
        }

        int     a, b, res;
        char    op;
```



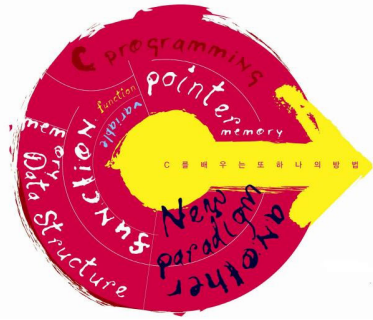
main 함수 – 명령행 인자 (4/4)

예제 10-4 : 명령행 인자 – 사칙 연산 (2/2)

```
        a = atoi( argv[1] );
        b = atoi( argv[3] );
        op = argv[2][0];

        if ( b == 0 )
        {
                printf("Error : 나눗셈 입력 오류!!! \n");
                printf("\t 0 으로 나눌 수 없습니다. \n");
                exit(2);
        }

        switch ( op )
        {
                case '+': res = a + b; break;
                case '-': res = a - b; break;
                case '*': res = a * b; break;
                case '/': res = a / b; break;
                default: printf("지원하지 않는 연산자 입니다!!! \n");
                         exit(3);
        }

        printf("계산 결과 : %d %c %d = %d \n", a, op, b, res );

        return 0;
}
```

단순 연결 리스트

- 동적 메모리 할당 함수
- 2차원 배열과 동적 메모리 할당
- **단순 연결 리스트**
  - 선형 리스트
  - 연결 자료구조
  - 단순 연결 리스트



선형 리스트 (1/6)

- **리스트 (List)**
  - 목록, 대부분의 목록은 도표(Table) 형태로 표시
  - 추상 자료형 리스트는 이러한 목록 또는 도표를 추상화한 것

이름 리스트	좋아하는 음식 리스트	오늘의 할일 리스트
서두옥	김치찌개	자료구조 수업
홍길동	크림스파게티	보고서 작성
서하은	불고기 피자	드라마 시청
서영은	잡채	청소 하기
…	…	…

선형 리스트 (2/6)

● 선형 리스트 (Linear List)

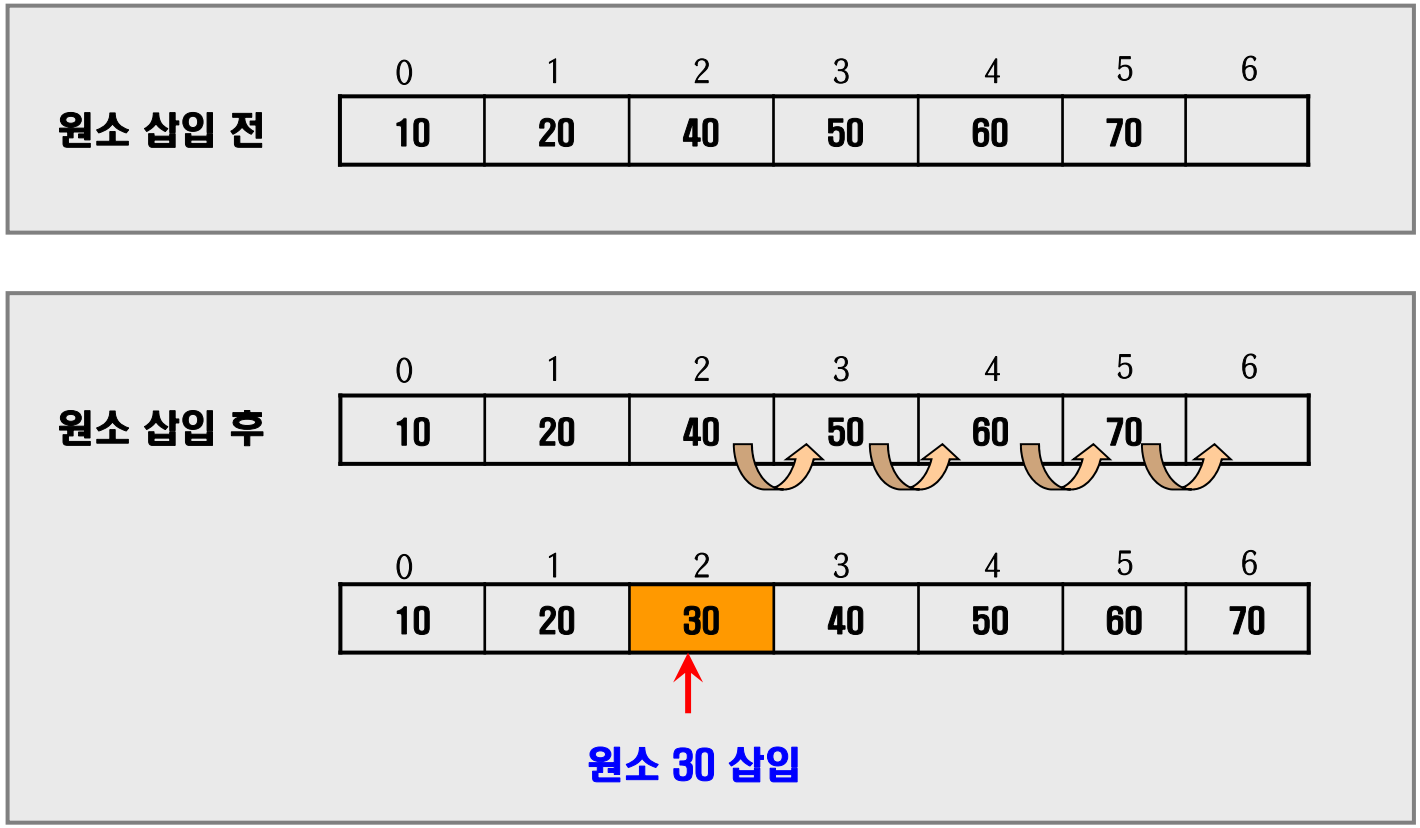
❍ 순서 리스트(Ordered List)

- 리스트에서 나열한 원소들간에 순서를 가지고 있는 리스트
- "원소들간의 논리적인 순서와 물리적인 순서가 같은 구조 (순차 자료구조)"

이름 리스트		좋아하는 음식 리스트		오늘의 할일 리스트	
1	서두옥	1	김치찌개	1	자료구조 수업
2	홍길동	2	크림스파게티	2	보고서 작성
3	서하은	3	불고기 피자	3	드라마 시청
4	서영은	4	잡채	4	청소 하기

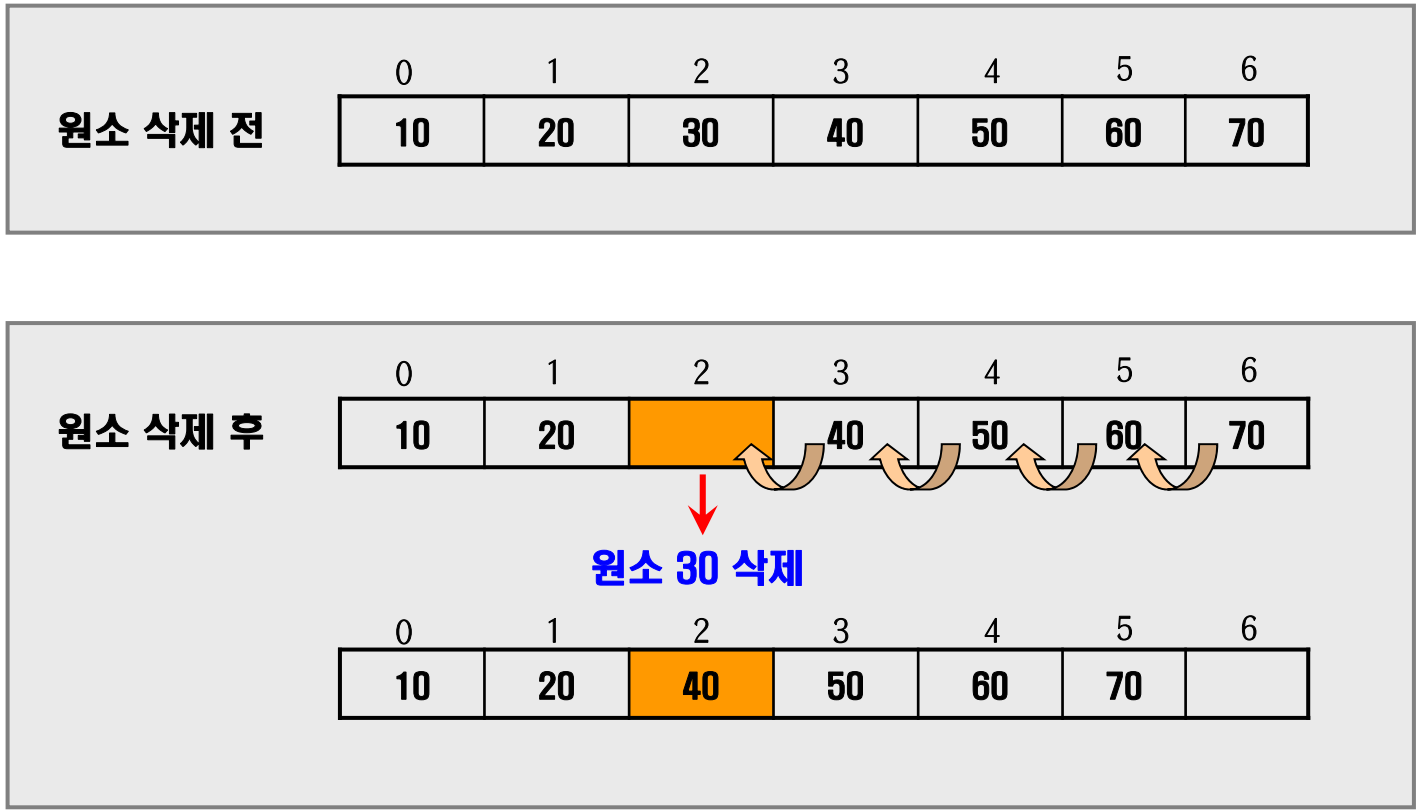
선형 리스트 (3/6)

- **선형 리스트 (cont'd)**
  - 선형 리스트에서의 원소 삽입



선형 리스트 (4/6)

- 선형 리스트 (cont'd)
  - 선형 리스트에서의 원소 삭제

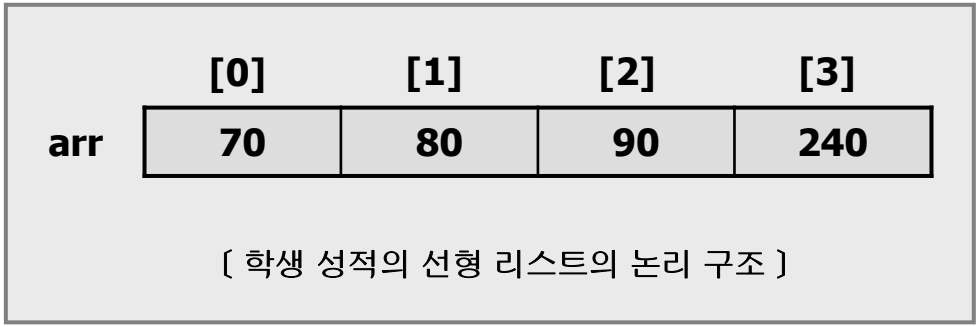


선형 리스트 (5/6)

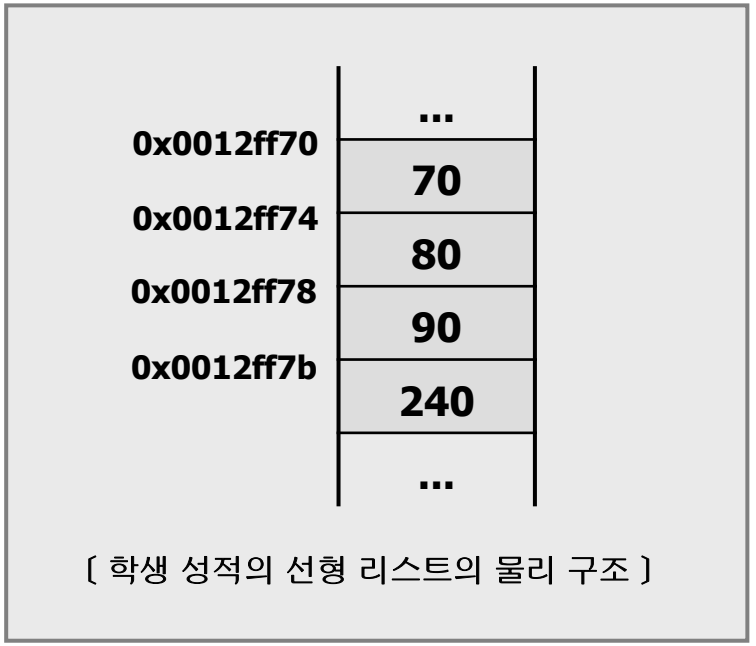
- **1차원 배열의 순차 표현**
  - 1차원 배열은 인덱스를 하나만 사용하는 배열

과 목	국어	영어	수학	총점
점 수	70	80	90	240

```
int arr[4] = {70, 80, 90, 240};
```



〔 학생 성적의 선형 리스트의 논리 구조 〕



〔 학생 성적의 선형 리스트의 물리 구조 〕

선형 리스트 (6/6)

- **2차원 배열의 순차 표현**
  - 행과 열의 구조로 나타내는 배열
    - 메모리에 저장될 때에는 1차원의 순서로 저장

과목 / 학생	국어	영어	수학	총점
1	70	80	90	240
2	50	60	70	180
3	60	70	80	210

```
int score[3][4] = {
                        {70, 80, 90},
                        {50, 60, 70},
                        {60, 70, 80}
                  };
```

연결 자료구조 (1/4)

● 순차 선형 리스트의 문제점

❍ "리스트가 순서를 유지해야 하므로 원소들의 삽입과 삭제가 어렵다."

- 삽입 연산이나 삭제 연산 후에 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가적인 작업과 시간이 소요된다.
- 원소들의 이동 작업으로 인한 오버헤드가 발생
  - 원소의 개수가 많고 삽입과 삭제 연산이 많이 발생하는 경우 더 많이 발생한다.

❍ "메모리 사용의 비효율성"

- 최대한의 크기를 가진 배열을 처음부터 준비해 두어야 하기 때문에 기억 장소의 낭비를 초래할 수 있다.

연결 자료구조 (2/4)

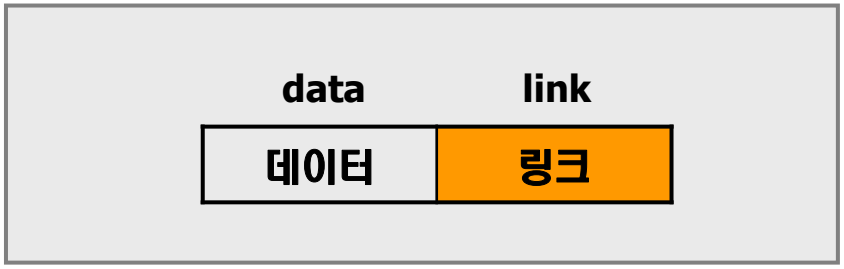
● 연결 리스트 (Linked List)

❍ 순차 자료구조에서의 연산 시간에 대한 문제와 저장 공간에 대한 문제를 개선한 자료 표현 방법

- 연결 자료구조 (Linked Data Structure)
- 비순차 자료구조 (Nonsequential Data Structure)
- 데이터 아이템을 줄줄이 엮은(Link, Chain) 것

❍ 노드 (Node) : <원소, 주소> 단위로 저장

- 데이터 필드(data field) : 원소의 값을 저장
- 링크 필드(link field) : 노드의 주소를 저장



연결 자료구조 (3/4)

- **자기 참조 구조체**
  - 자신의 구조체 자료형을 가리키는 포인터 멤버를 가질 수 있다.

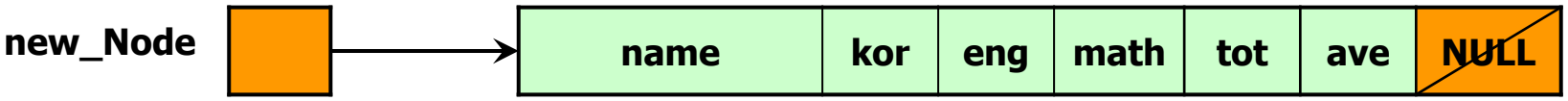
```
struct  _score
{
    char            name[12];
    int             kor, eng, math, tot;
    float           ave;
    struct _score   *link;
};

typedef struct _score SCORE;
```

  - link 멤버는 자신과 같은 구조의 구조체 주소를 저장하고 있다가 필요 시 저장된 주소의 구조체에 접근하는 것을 목표로 한다.

연결 자료구조 (4/4)

- **자기 참조 구조체 (cont'd)**
  - 구조체 노드의 생성

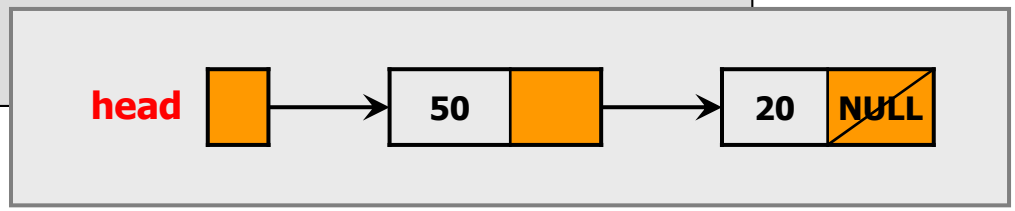
```
SCORE    *head, *new_Node; // struct score *head, *new_Node;

head = NULL;

// SCORE 크기의 메모리 할당
new_Node = (SCORE *)malloc(sizeof(SCORE));
if(new_Node == NULL)
{
    printf("메모리 할당 실패!!! \n");
    exit(100);
}
```



단순 연결 리스트 (1/11)

- **단순 연결 리스트 (Singly linked List)**
  - 연결 리스트
  - 선형 연결 리스트(linear linked list)
  - 단순 연결 선형 리스트(singly linked linear list)

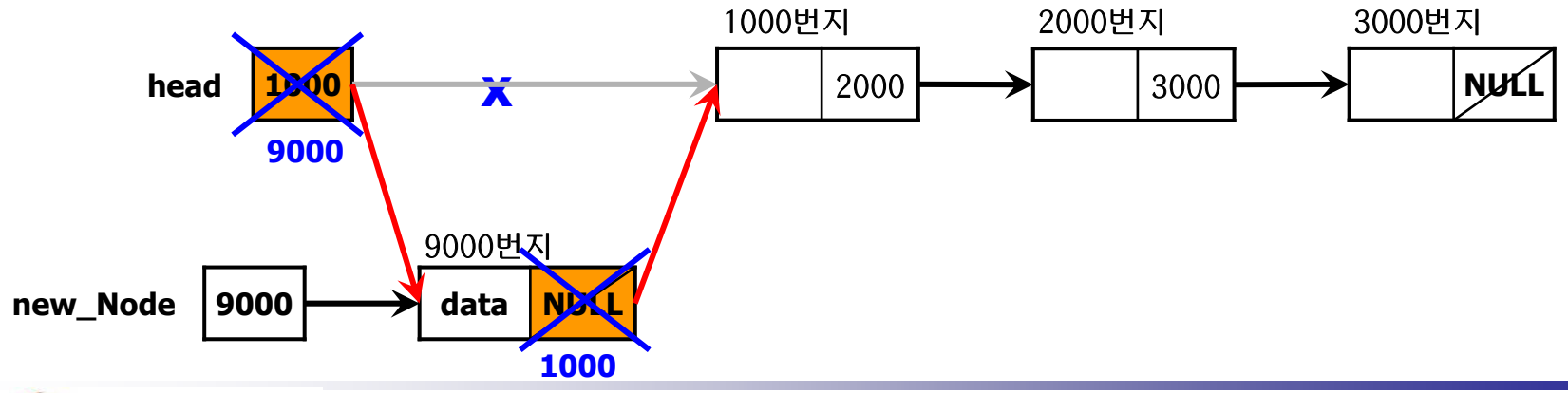
```
typedef struct _node
{
    int             data;
    struct _node    *link;
}NODE;

NODE    *head;
```



단순 연결 리스트 (2/11)

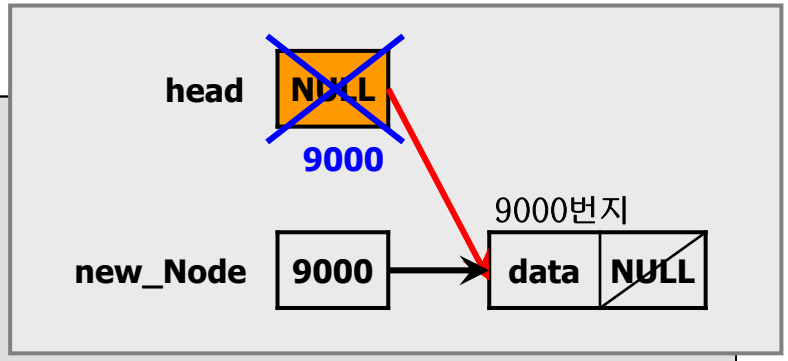
- **단순 연결 리스트 삽입 알고리즘**
  - 리스트의 첫 번째 노드 삽입 알고리즘

```
insertFirstNode(head, data)
    new_Node ← makeNode(data);          // 삽입할 노드의 메모리 할당 및 초기화
    new_Node.link = head;
    head ← new_Node;
end insertFistNode()
```



단순 연결 리스트 (3/11)

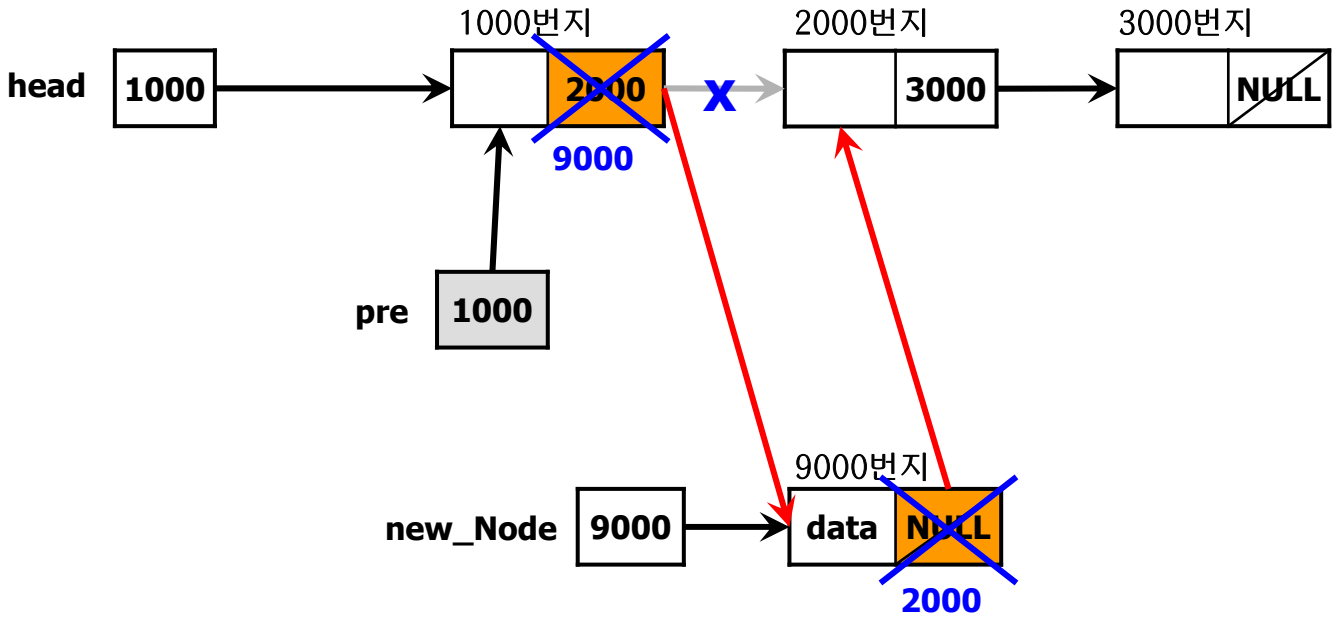
- 단순 연결 리스트 삽입 알고리즘 (cont'd)
  - 리스트의 중간 노드 삽입 알고리즘




```
insertMiddleNode(head, pre, data)
    new_Node ← makeNode(data);
    if (head = NULL) then
          head ← new;
    else
    {
          new_Node.link ← pre.link;
          pre.link ← new_Node;
    }
end insertMiddleNode()
```

단순 연결 리스트 (4/11)

- **단순 연결 리스트 삽입 알고리즘 (cont'd)**
  - 중간 노드로 삽입하는 과정 : **리스트가 빈 리스트가 아닐 때...**
    1) 리스트에서 새로운 노드(new_Node)가 삽입될 이전 노드(pre)의 위치를 탐색
    2) 각 노드의 링크 필드 수정



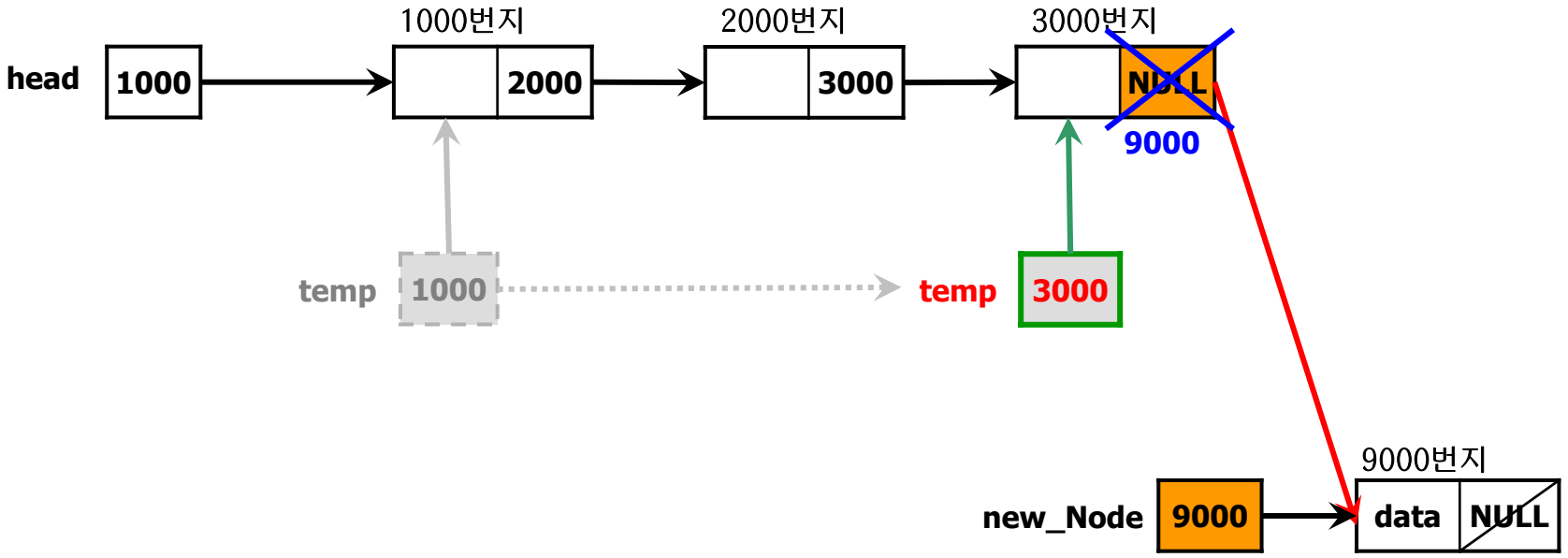
단순 연결 리스트 (5/11)

- **단순 연결 리스트 삽입 알고리즘 (cont'd)**
  - 리스트의 마지막 노드 삽입 알고리즘

```
insertLastNode(head, data)
    new_Node ← makeNode(data);
    if (head = NULL) then
          head ← new_Node;
    else
    {
          temp ← head;
          while (temp.link != NULL) do
                temp ← temp.link;

          temp.link ← new_Node;
    }
end insertLastNode()
```

단순 연결 리스트 (6/11)

- **단순 연결 리스트 삽입 알고리즘 (cont'd)**
  - 마지막 노드로 삽입하는 과정 : **리스트가 빈 리스트가 아닐 때...**
    1) 리스트의 마지막 노드를 탐색
    2) 리스트의 마지막 노드(temp)가 새로운 노드(new_Node)를 가리키게 한다.



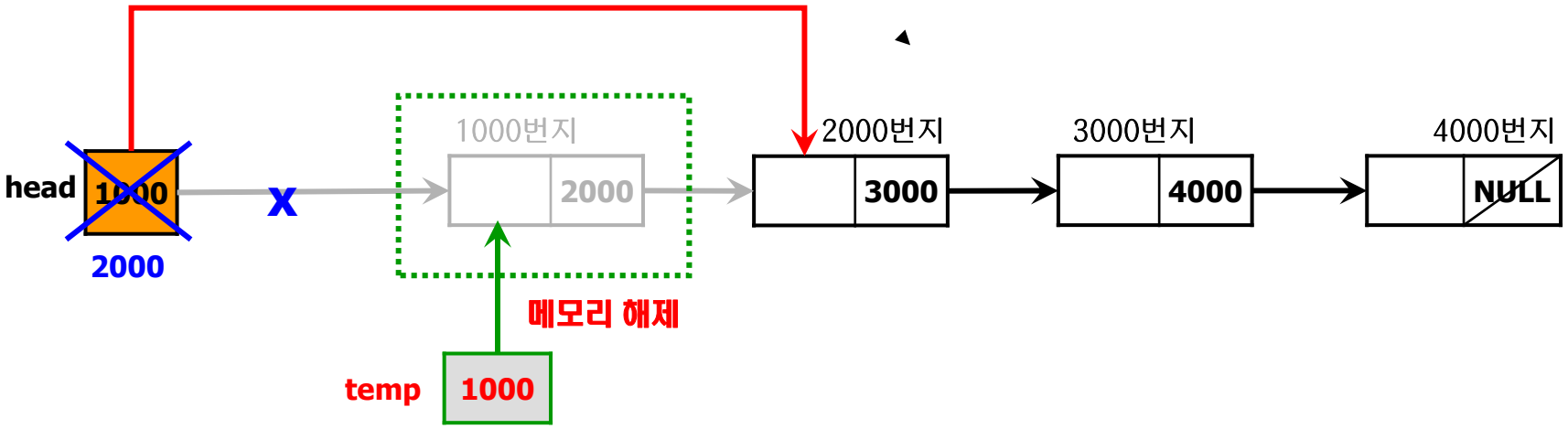
단순 연결 리스트 (7/11)

- **단순 연결 리스트 삭제 알고리즘**
  - 리스트에서 조건을 만족하는 노드 삭제 알고리즘

```
deleteNode(head, data)
    if (head = NULL) then error;
    else {
          temp ← head;
          while (temp != NULL)
          {
                    if (temp.data = data) then
                    {
                         if (temp = head) then deleteFirstNode();
                         else if (temp = NULL) then deleteLastNode();
                         else deleteMiddleNode();
                    }
                    pre ← temp;
                    temp ← temp.link;
          }
    }
end deleteNode()
```

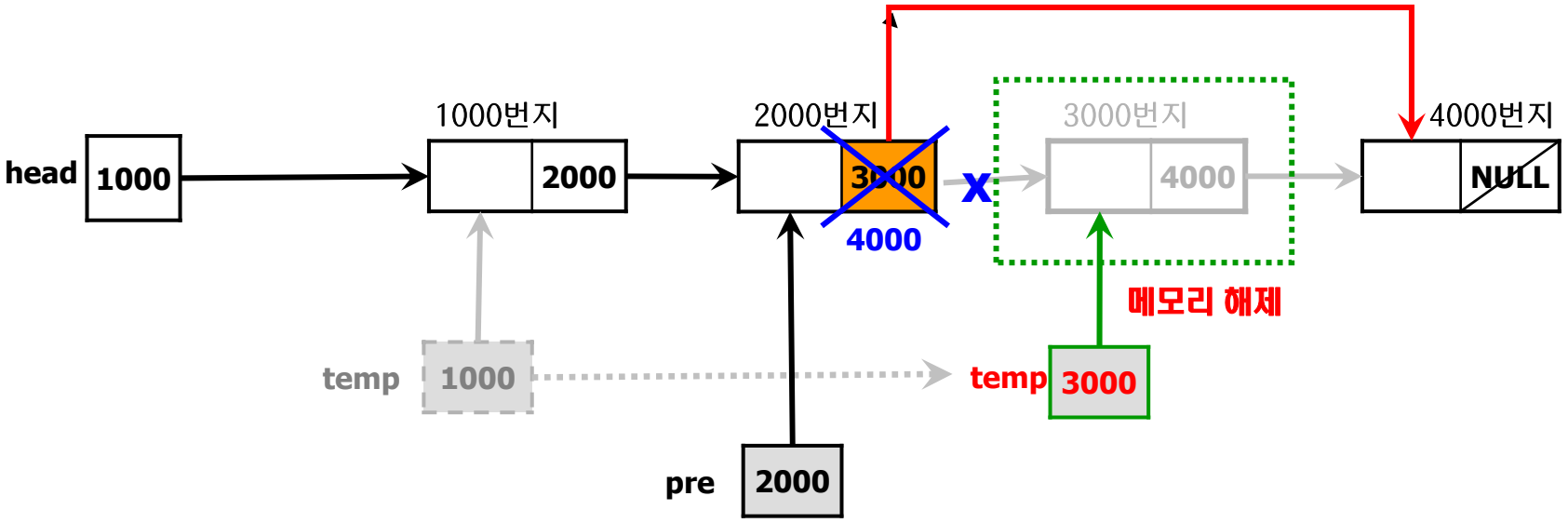
단순 연결 리스트 (8/11)

- **단순 연결 리스트 삭제 알고리즘 (cont'd)**
  - 리스트의 첫 번째 노드를 삭제
    - 삭제할 노드(old)의 다음 노드(old.link)를 head로 연결한다.



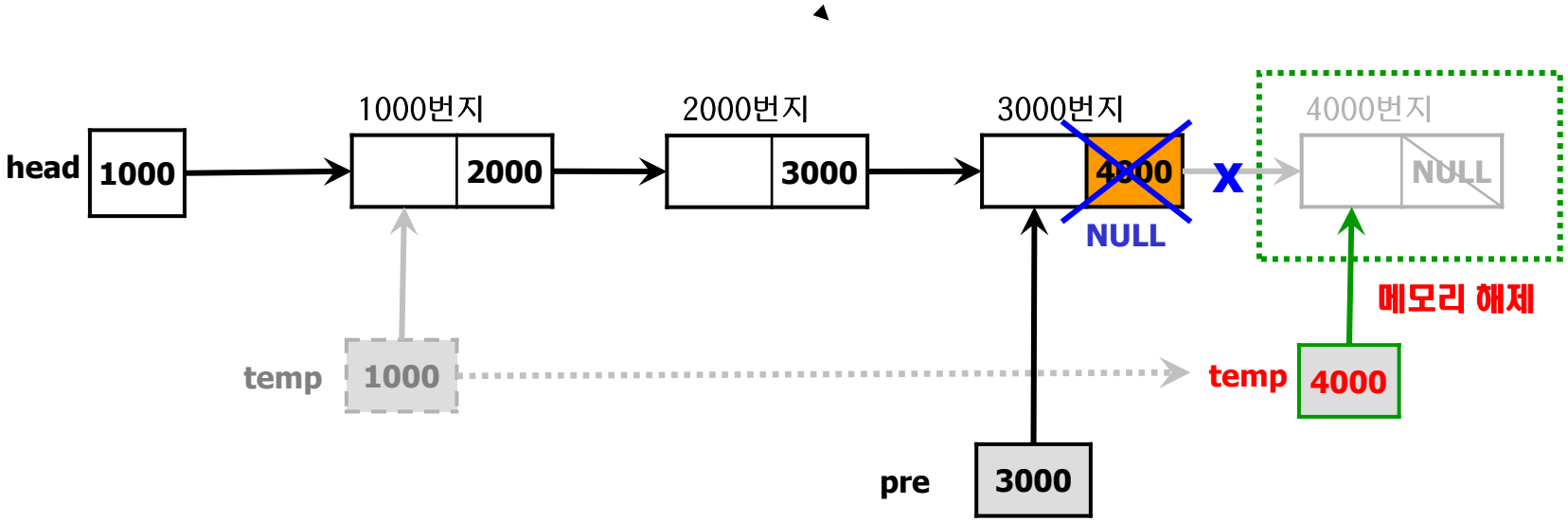
단순 연결 리스트 (9/11)

- **단순 연결 리스트 삭제 알고리즘 (cont'd)**
  - 리스트의 중간 노드를 삭제
    - 삭제할 노드(old) 탐색 후 다음 노드(old.link)를 이전 노드(pre)의 다음 노드(pre.link)로 연결한다.



단순 연결 리스트 (10/11)

- **단순 연결 리스트 삭제 알고리즘 (cont'd)**
  - 리스트의 마지막 노드를 삭제
    - 삭제할 노드(old) 탐색 후 이전 노드(pre)의 링크 필드(pre.link)를 NULL로 만든다.



단순 연결 리스트 (11/11)

- **단순 연결 리스트 탐색 알고리즘**
  - 리스트에서 조건을 만족하는 데이터를 가진 노드 탐색 알고리즘

```
searchNode(head, data)
    temp ← head;
    while (temp != NULL) do
    {
          if (temp.data = data) then
                return temp;
          temp ← temp.link;
    }
    if (temp = NULL)  then
          return NULL;
end searchNode()
```

참고문헌

[1] 서두옥, 이동호(감수), (열혈강의)"또 하나의 C : 프로그래밍은 셀프입니다", 프리렉, 2012.

[2] Paul Deitel, Harvey Deitel, "C How to Program", Global Edition, 8/E, Pearson, 2016.

[3] SAMUEL P. HARBISON Ⅲ, GUY L. STEELE, "C 프로그래밍 언어, C : A Reference Manual", 5/E, Pearson Education Korea, 2005.

[4] 문병로, "쉽게 배우는 알고리즘 - 관계 중심의 사고법", 개정판, 한빛아카데미, 2018.

[5] 주우석, "C・C++ 로 배우는 자료구조론", 한빛아카데미, 2015.

[6] Behrouz A. Forouzan, Richard F. Gilberg, 김진 외 7인 공역, "구조적 프로그래밍 기법을 위한 C", 도서출판 인터비젼, 2004.

[7] Brian W. Kernighan, Dennis M. Ritchie, 김석환 외 2인 공역, "The C Programming Language", 2/E, 대영사, 2004.

[8] 김일광, "C 프로그래밍 입문 : 프로그래밍을 모국어처럼 유창하게", 한빛미디어, 2004.

[9] 정재은, "다시 체계적으로 배우는 C 언어 포인터", 정보문화사, 2003.