


제 17장 동적 메모리와 연 결 리스트

이번 장에서 학습할 내용

- 
- 동적 메모리 할당의 이해
 - 동적 메모리 할당 관련 함수
 - 연결 리스트

동적 메모리 할당에
대한 개념을
이해하고 응용으로
연결 리스트를
학습합니다.



동적 할당 메모리의 개념

- 프로그램이 메모리를 할당받는 방법
 - 정적(static) 할당
 - 동적(dynamic) 할당

메모리도 필요할 때마다
요청해서 사용하면
좋은데...



정적 메모리 할당

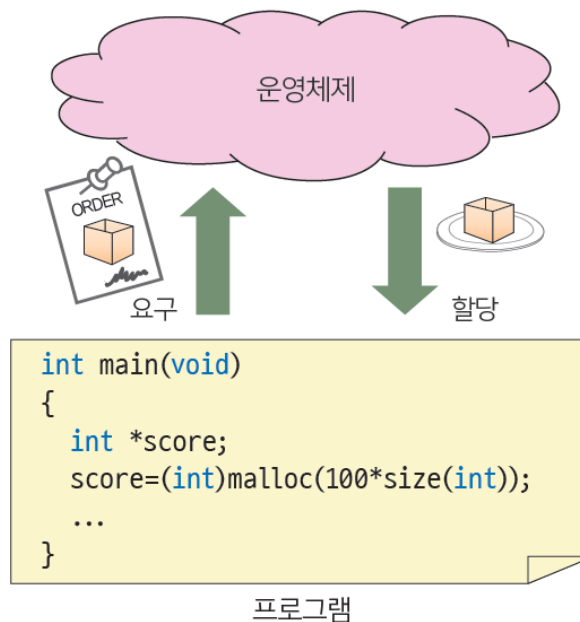
- 정적 메모리 할당

- 프로그램이 시작되기 전에 미리 정해진 크기의 메모리를 할당받는 것
- 메모리의 크기는 프로그램이 시작하기 전에 결정
- (예) `int score_s[100];`
- 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못함
- 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비

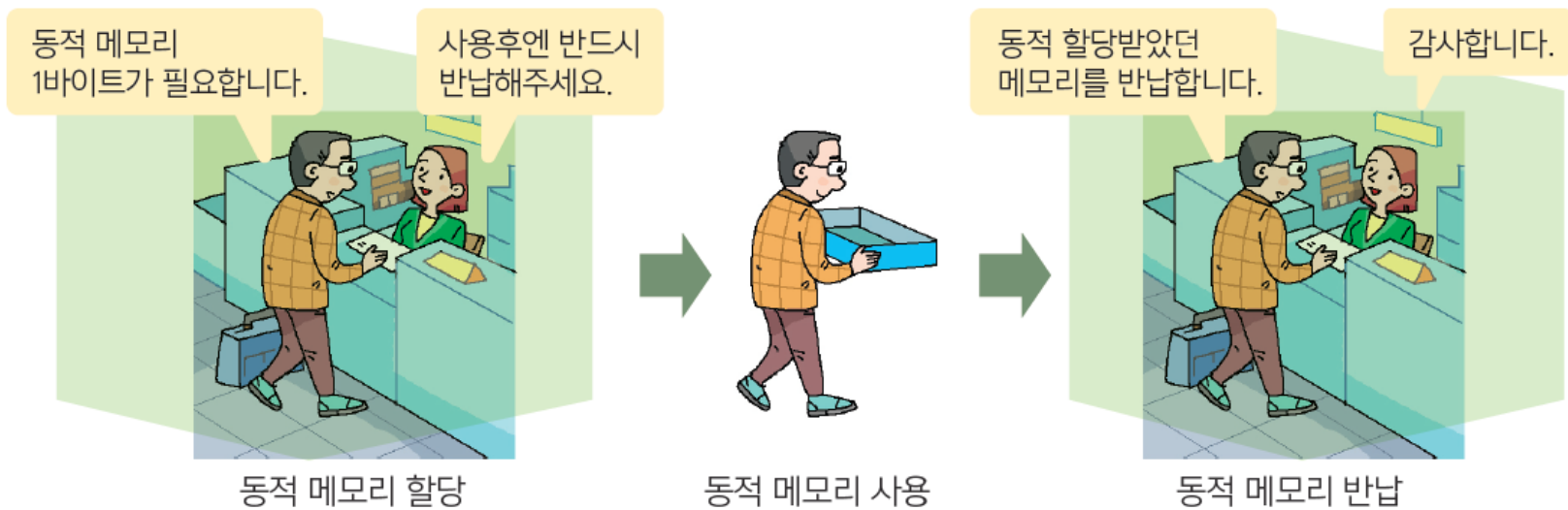
동적 메모리 할당

• 동적 메모리 할당

- 실행 도중에 동적으로 메모리를 할당받는 것
- 사용이 끝나면 시스템에 메모리를 반납
- `score = (int *) malloc(100*sizeof(int));`
- 필요한 만큼만 할당을 받고 메모리를 매우 효율적으로 사용



동적 메모리 할당 절차



동적 메모리 할당

Syntax

동적 메모리 할당

예

동적 메모리의 주소

`int *p;`

필요한 바이트 수

`p = (int *)malloc(100*sizeof(int));` // 100개의 정수 할당

p =



400 바이트 메모리

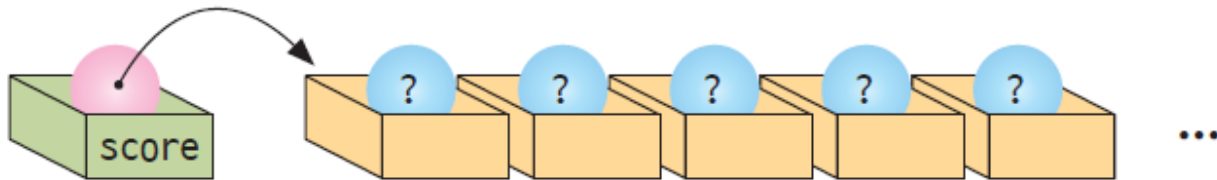
malloc() 반환값은 검사하여야 함

```
int *score;  
score = (int *)malloc(100*sizeof(int));
```

int 포인터로 변환

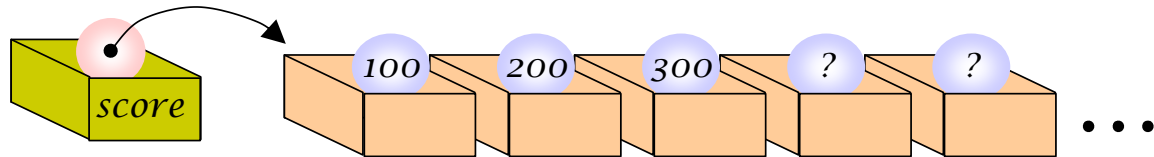
```
if( score == NULL ){  
    ... // 오류 처리  
}
```

메모리가 올바르게 할당되었는지를 체크



동적 메모리 사용

- 할당받은 공간은 어떻게 사용하면 좋을까?
- 첫 번째 방법: 포인터를 통하여 사용
 - `*score = 100;`
 - `*(score+1) = 200;`
 - `*(score+2) = 300;`
 - ...
- 두 번째 방법: 동적 메모리를 배열과 같이 취급
 - `score[0] = 100;`
 - `score[1] = 200;`
 - `score[2] = 300;`
 - ...



동적 메모리 반납

Syntax

동적 메모리 해제

예

```
score = (int *)malloc(100*sizeof(int));
```

```
...
```

```
free(score);
```

score가 가리키는 동적 메모리를 반납한다.

예제

- 정수 1개, 실수 1개, 문자 1개를 저장할 수 있는 공간을 할당받아서 사용한 후에 반납하는 코드를 작성해보자.

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

int main(void)
{
    int *pi;
    double *pf;
    char *pc;

    pi = (int *)malloc(sizeof(int));
    pf = (double *)malloc(sizeof(double));
    pc = (char *)malloc(sizeof(char));
    if (pi == NULL || pf == NULL || pc == NULL) { // 반환값이 NULL인지 검사
        printf("동적 메모리 할당 오류\n");
        exit(1);
    }
}
```

예제

```
*pi = 100;          // pi[0] = 100;  
*pf = 3.14;        // pf[0] = 3.14;  
*pc = 'a'; // pc[0] = 'a';  
  
free(pi);  
free(pf);  
free(pc);  
return 0;  
}
```

예제 #2

- 정수 10, 20, 30을 저장할 수 있는 동적 메모리를 생성해보자.


```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int *list;
    list = (int *)malloc(3 * sizeof(int));
    if (list == NULL) { // 반환값이 NULL인지 검사
        printf("동적 메모리 할당 오류\n");
        exit(1);
    }
    list[0] = 10;
    list[1] = 20;
    list[2] = 30;
    free(list);
    return 0;
}
```

동적 메모리 할당

동적 메모리 해제

Lab: 동적 배열을 이용한 성적 처리

- 성적 처리 프로그램을 작성한다고 하자. 사용자한테 학생이 몇 명인지 물어보고 적절한 동적 메모리를 할당한다. 사용자로부터 성적을 받아서 저장하였다가 다시 출력한다.

A computer monitor with a black frame and a black stand. The screen is blue and displays white text.

```
학생의 수: 3  
학생 #1 성적: 10  
학생 #2 성적: 20  
학생 #3 성적: 30  
성적 평균=20.00
```

Solution

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

int main(void)
{
    int *list;
    int i, students, sum=0;

    printf("학생의 수: ");
    scanf("%d", &students);

    list = (int *)malloc(students * sizeof(int));
    if (list == NULL) { // 반환값이 NULL인지 검사
        printf("동적 메모리 할당 오류\n");
        exit(1);
    }
}
```

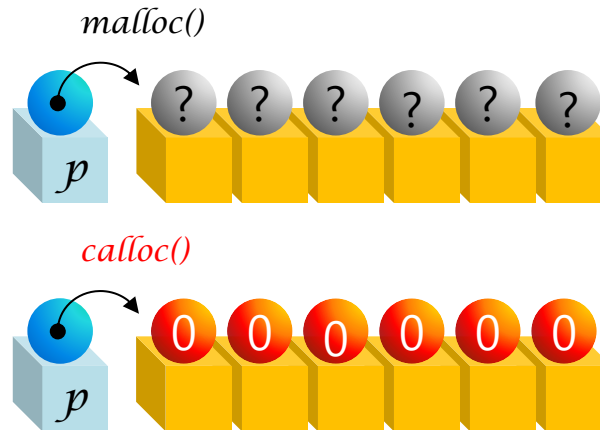
Solution

```
for (i = 0; i<students; i++) {  
  
    printf("학생 #%d 성적: ", i+1);  
    scanf("%d", &list[i]);  
}  
for (i = 0; i<students; i++)  
    sum += list[i];  
printf("성적 평균=%.2f \n", (double)sum/students);  
free(list);  
return 0;  
}
```


calloc()

- calloc()은 0으로 초기화된 메모리 할당
- calloc()은 항목 단위로 메모리를 할당
- (예)

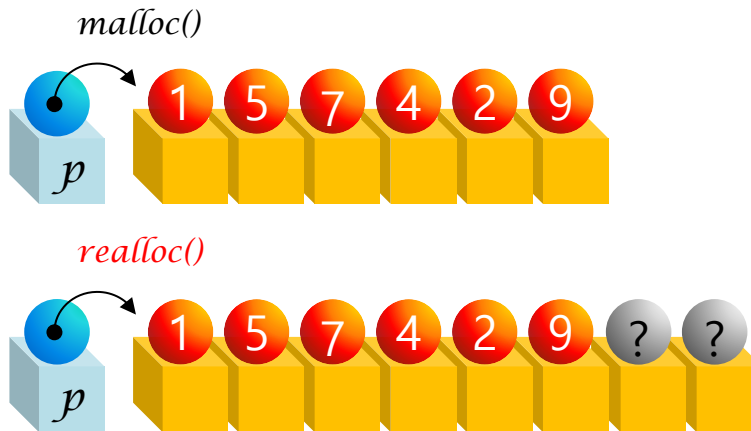
```
int *p;  
p = (int *)calloc(5, sizeof(int));
```



realloc()

- realloc() 함수는 할당하였던 메모리 블록의 크기를 변경
- (예)

```
int *p;  
p = (int *)malloc(5 * sizeof(int));  
p = realloc(p, 7 * sizeof(int));
```



Tip



동적 메모리를 반납하지 않으면 어떤 일이 발생할까?

동적 메모리는 프로그래머가 명시적으로 반납하지 않으면 해제되지 않는다. 운영 체제는 메모리의 일정한 부분을 힙(heap)으로 잡아서 여기에서 동적 메모리를 할당한다. 따라서 동적 메모리는 크기가 정해져 있어서 어떤 프로그램에서 많이 사용해버리면 다른 프로그램은 제한을 받게 된다. 실제로 동적 메모리가 반납이 안 되면 전체 프로그램들이 서서히 느려지게 된다.

가장 나쁜 경우는 계속하여서 할당만하고 전혀 반납하지 않는 경우이다. 이 경우에는 운영 체제가 멈출 수도 있다. 아래의 코드는 아주 잘못되어 있다.

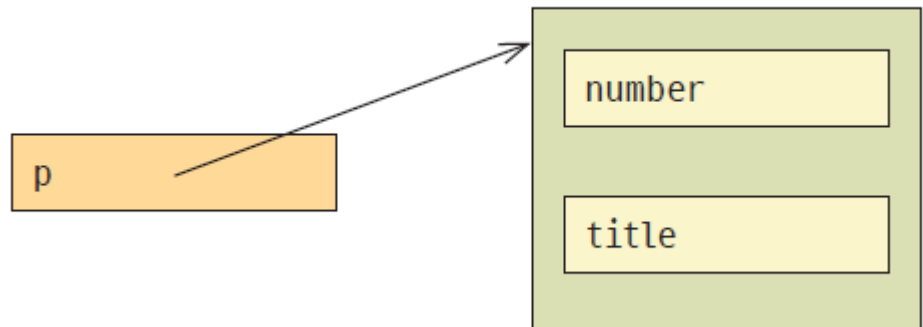
```
void sub()
{
    int *p;
    p = malloc( 100 * sizeof(int) );
    p = malloc( 100 * sizeof(int) );
    ...
    return;
}
```

이전의 메모리 블록에 대한 주소가 사라진다.

구조체를 동적 생성해보자.

- 구조체를 저장할 수 있는 공간을 할당받아서 사용해본다.

```
struct Book {  
    int number;  
    char title[50];  
};  
  
struct Book *p;  
p = (struct Book *)malloc(2 * sizeof(struct Book));
```



예제

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Book {
    int number;
    char title[50];
};
```

```
int main(void)
{
```

```
    struct Book *p;
```

```
    p = (struct Book *)malloc(2 * sizeof(struct Book));
```

```
    if (p == NULL) {
```

```
        printf("메모리 할당 오류\n");
        exit(1);
```

```
    }
```

```
    p[0].number = 1;
```

```
    strcpy(p[0].title, "C Programming");
```

```
    p[1].number = 2;
```

```
    strcpy(p[1].title, "Data Structure");
```

```
    free(p);
```

```
    return 0;
```

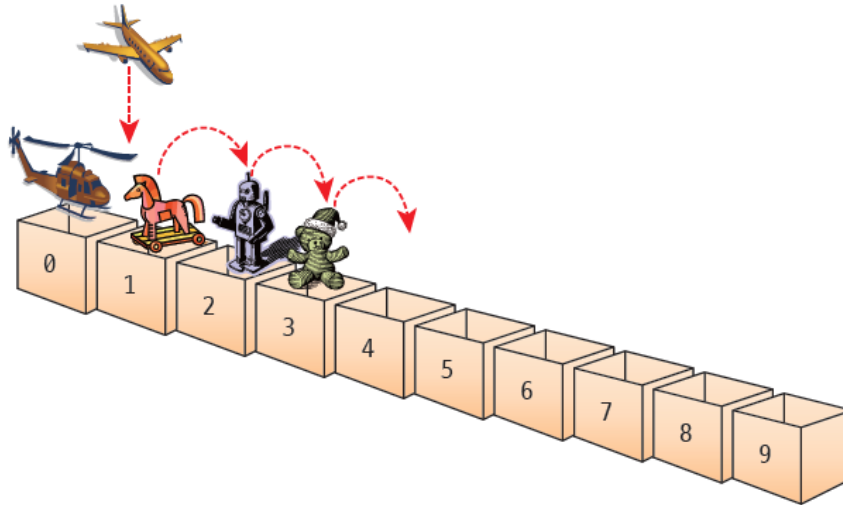
```
}
```

구조체 배열 할당



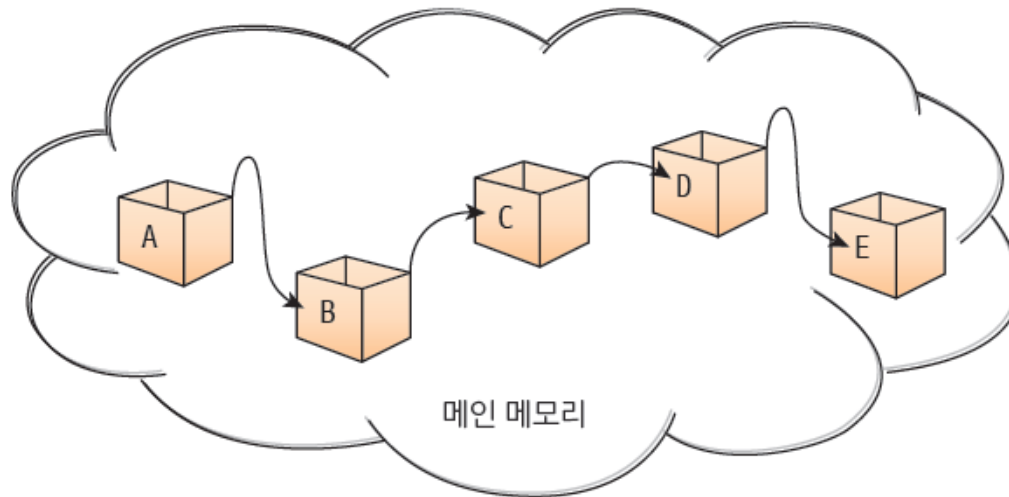
배열 vs 연결 리스트

- 배열(array)
 - 장점: 구현이 간단하고 빠르다
 - 단점: 크기가 고정된다. 중간에서 삽입, 삭제가 어렵다.



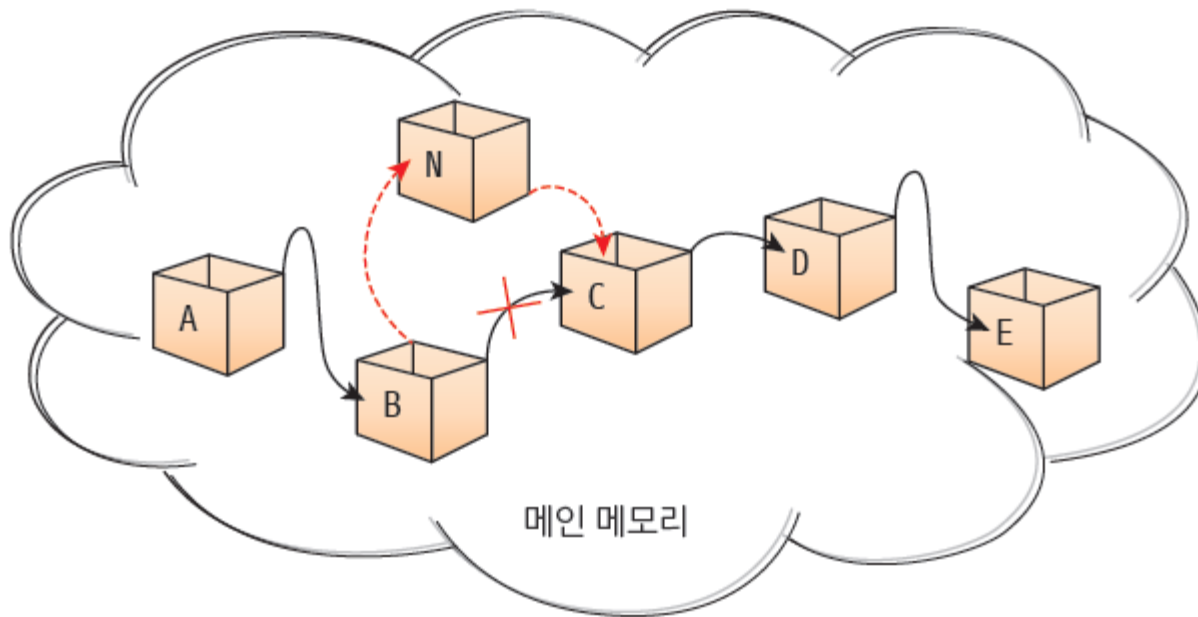
배열 vs 연결 리스트

- 연결 리스트(linked list)
 - 각 항목이 포인터를 사용하여 다음 항목을 가리킨다.
 - 랜덤 접근이 어렵다.
 - 스택, 큐, 트리, 그래프 등을 구현하는데 널리 사용된다



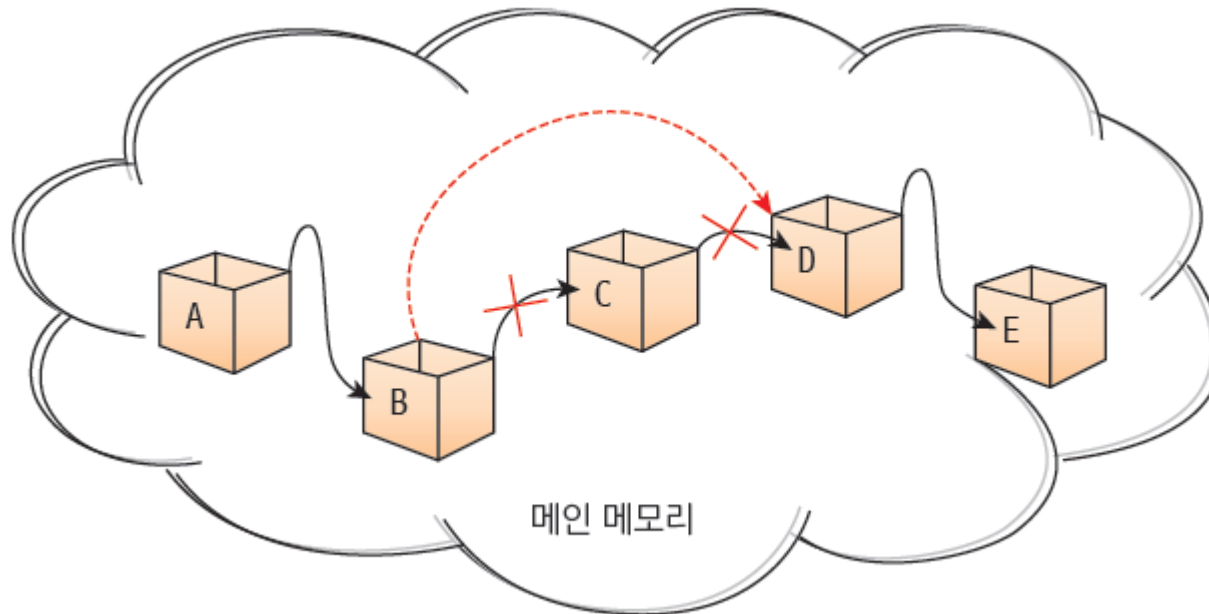
연결 리스트에서의 삽입 연산

- 중간에 데이터를 삽입, 삭제하기 쉽다.



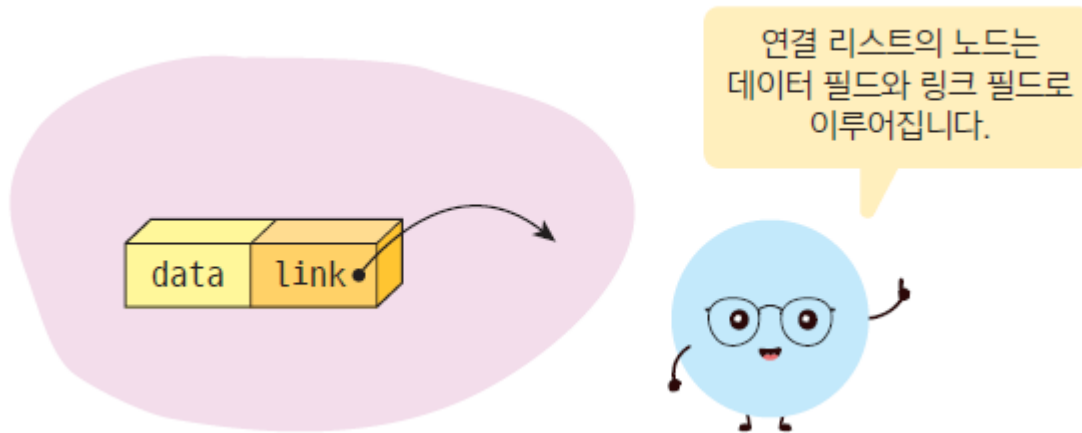
연결 리스트에서의 삭제 연산

- 중간에 데이터를 삽입, 삭제하기 쉽다.



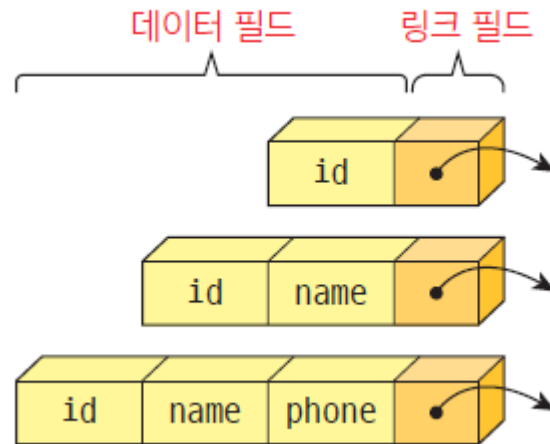
연결 리스트의 구조

- 노드(node) = 데이터 필드(data field)+ 링크 필드(link field)



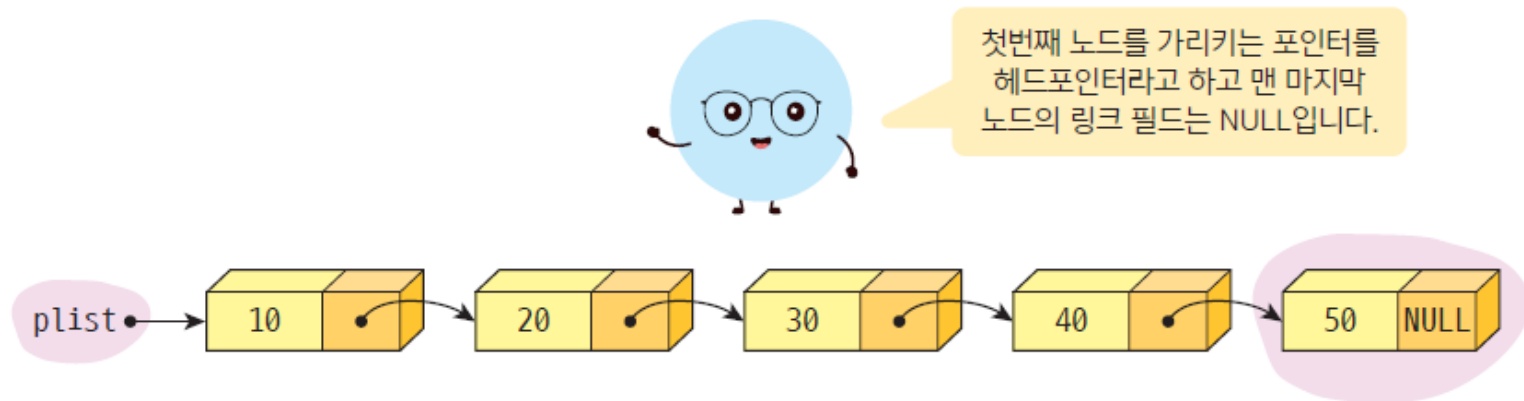
데이터 필드

- 데이터 필드에는 우리가 저장하고 싶은 데이터가 들어간다. 데이터는 정수가 될 수도 있고 학번, 이름, 전화번호가 들어있는 구조체와 같은 복잡한 데이터가 될 수도 있다.



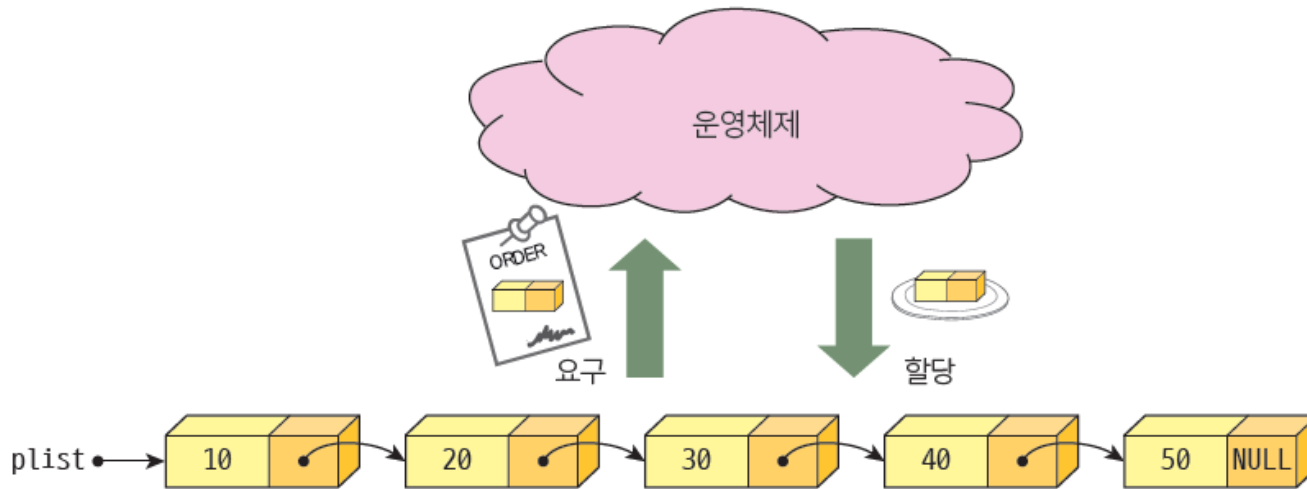
연결 리스트의 구조

- 헤드 포인터(head pointer): 첫번째 노드를 가리키는 포인터



노드 생성

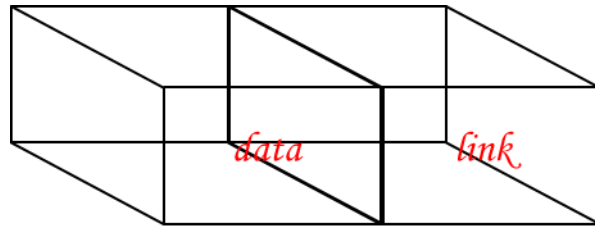
- 노드들은 동적으로 생성된다.



자기 참조 구조체

- **자기 참조 구조체(self-referential structure)**는 특별한 구조체로서 구성 멤버 중에 같은 타입의 구조체를 가리키는 포인터가 존재하는 구조체

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



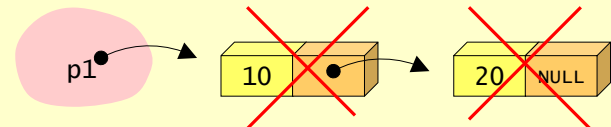
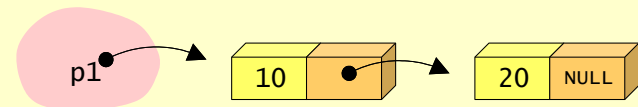
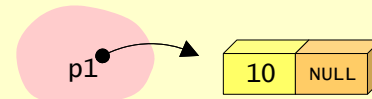
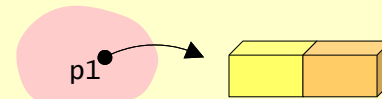
간단한 연결 리스트 생성

```
NODE* p1;  
p1 = (NODE*)malloc(sizeof(NODE));
```

```
p1->data = 10;  
p1->link = NULL;
```

```
NODE* p2;  
p2 = (NODE*)malloc(sizeof(NODE));  
p2->data = 20;  
p2->link = NULL;  
p1->link = p2;
```

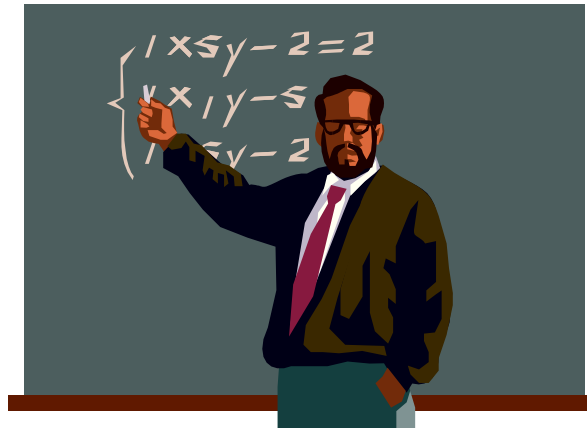
```
free(p1);  
free(p2);
```



참고: 수동 메모리 관리 vs 자동 메모리 관리

- C언어에서의 동적 메모리에서는 몇 가지의 문제가 발생했다.
 - 메모리를 해제하는 것을 잊을 수 있다. 사용을 마친 후 메모리를 해제하지 않으면 메모리 누수가 발생할 수 있다.
 - 너무 빨리 메모리를 해제할 수 있다. 누군가 사용 중인 메모리를 해제하는 것이다. 이로 인해 존재하지 않는 메모리 값에 액세스하려고 하면 프로그램이 종료될 수 있다.
- 이러한 문제는 바람직하지 않았기 때문에 최신 언어에는 자동 메모리 관리가 추가되었다. 가비지 수집기(garbage collector)이 이를 처리했다.
- 프로그래머에게 자동 메모리 관리는 많은 이점을 가져다 준다. 그러나 자동 메모리 관리에는 비용이 든다. 자동 메모리 관리 기능이 있는 많은 프로그래밍 언어는 가비지 수집기가 수집할 개체를 찾고 삭제하는 동안 모든 실행이 중지된다.
- 그러나 성능이 중요한 장기 실행 애플리케이션의 경우, 여전히 수동 메모리 관리가 사용된다. 가장 대표적인 언어가 우리가 배우는 **C언어**와 **C++ 언어**이다.

Q & A





THANK
YOU