Dynamic Memory and Linked List

동적 할당 메모리의 개념

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

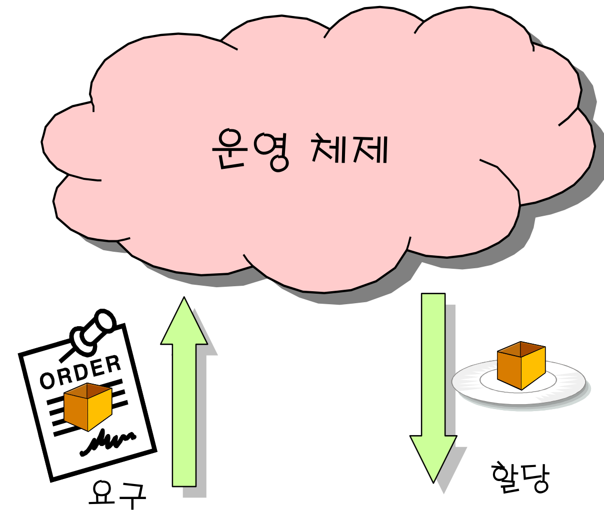
- 정적 메모리 할당
  - 프로그램이 시작되기 전에 미리 정해진 크기의 메모리를 할당받는 것
  - 메모리의 크기는 프로그램이 시작하기 전에 결정

```
int i, j;
int buffer[80];
char name[] = "data structure";
```

  - 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못함
  - 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비

동적 메모리

- 동적 메모리
  - 실행 도중에 동적으로 메모리를 할당받는 것
  - 사용이 끝나면 시스템에 메모리를 반납
  - 필요한 만큼만 할당을 받고 메모리를 매우 효율적으로 사용
  - malloc() 계열의 라이브러리 함수를 사용

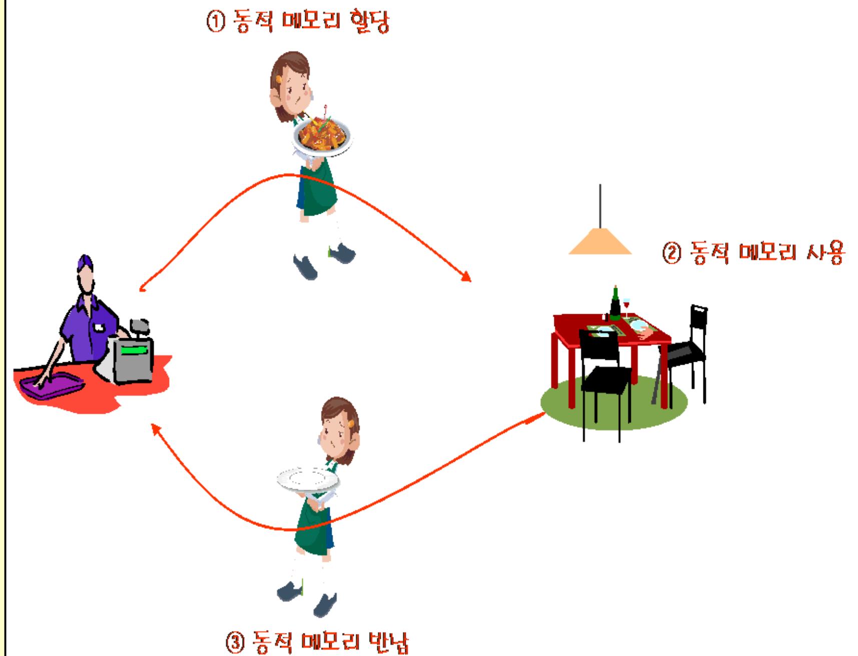



```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 int *p;
 p = (int *)malloc( sizeof(int) );
 ...
}
```

프로그램

동적 메모리 할당의 과정

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

int main(void)
{
  int *pi;       // 동적 메모리를 가리키는 포인터

  pi = (int *)malloc(sizeof(int)); // ① 동적 메모리 할당

  if( pi == NULL )            // 반환값이 NULL인지 검사
  {
    printf("동적 메모리 할당 오류\n");
    exit(1);
  }

  *pi = 100;   // ② 동적 메모리 사용
  printf("%d\n", *pi);

  free(pi);    // ③ 동적 메모리 반납

  return 0;
}
```



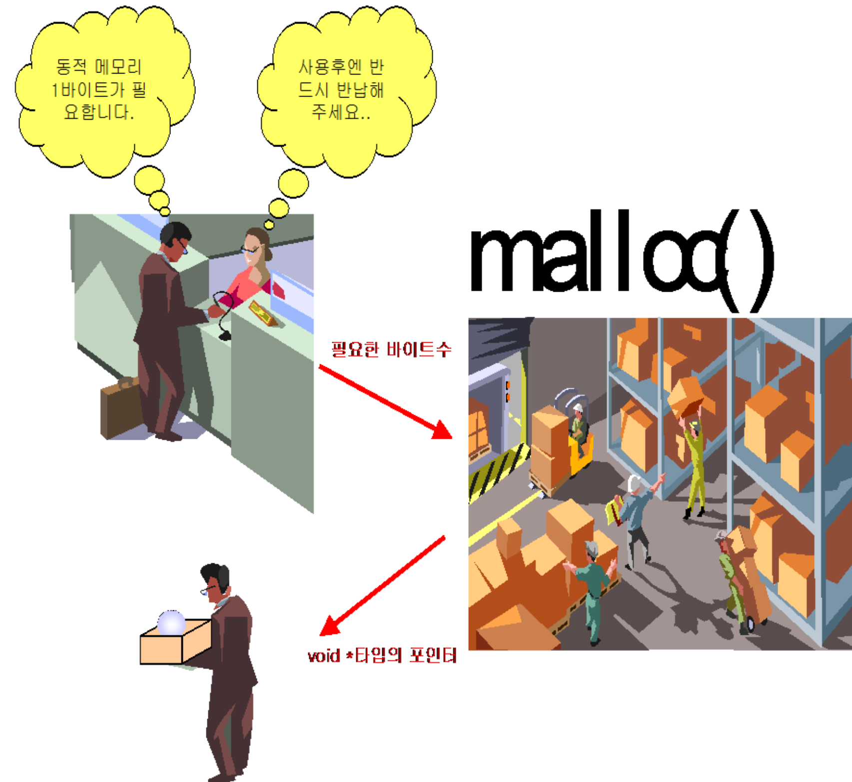
malloc()과 free()

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

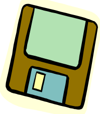
- malloc()은 바이트 단위로 메모리를 할당
- size는 바이트의 수
- malloc()함수는 메모리 블럭의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환

```
void free(void *ptr);
```

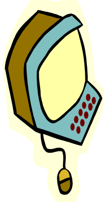
- free()는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- ptr은 malloc()을 이용하여 동적 할당된 메모리를 가리키는 포인터



malloc1.c

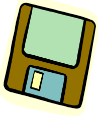


```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  int main( void )
5.  {
6.          char *pc = NULL;
7.
8.          pc = (char *)malloc( sizeof(char) );
9.          if( pc == NULL )
10.         {
11.                 printf( "메모리 할당 오류\n" );
12.                 exit(1);
13.         }
14.         *pc = 'm';
15.         printf( "*pc = %c\n", *pc );
16.         free( pc );
17.
18.         return 0;
19. }
```

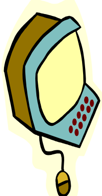


1000 바이트가 할당되었습니다.
메모리를 반납하였습니다.

malloc2.c

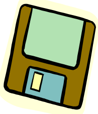


```
1.   // 메모리 동적 할당
2.   #include <stdio.h>
3.   #include <stdlib.h>

4.   int main(void)
5.   {
6.        char *pc = NULL;
7.        int i = 0;

8.        pc = (char *)malloc(100*sizeof(char));
9.        if( pc == NULL )
10.       {
11.            printf("메모리 할당 오류\n");
12.            exit(1);
13.       }
14.       for(i=0;i<26;i++)
15.       {
16.            *(pc+i) = 'a'+i;          // 알파벳 소문자를 순서대로 대입
17.       }
18.       *(pc+i) = 0;   // NULL 문자 추가

19.       printf("%s\n", pc);
20.       free(pc);

21.       return 0;
22.  }
```



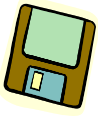
```
abcdefghijklmnopqrstuvwxyz
```

malloc3.c



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

3.    int main(void)
4.    {
5.            int *pi;

6.            pi = (int *)malloc(5 * sizeof(int));

7.            if(pi == NULL){
8.              printf("메모리 할당 오류\n") ;
9.              exit(1);
10.           }

11.           pi[0] = 100;            // *(pi+0) = 100;와 같다.
12.           pi[1] = 200;            // *(pi+1) = 200;와 같다.
13.           pi[2] = 300;            // *(pi+2) = 300;와 같다.
14.           pi[3] = 400;            // *(pi+3) = 400;와 같다.
15.           pi[4] = 500;            // *(pi+4) = 500;와 같다.

16.           free(pi);
17.           return 0;
18.   }
```

malloc4.c

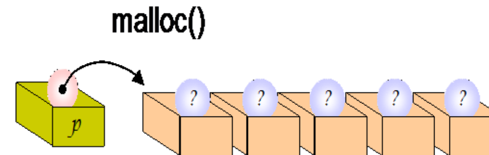


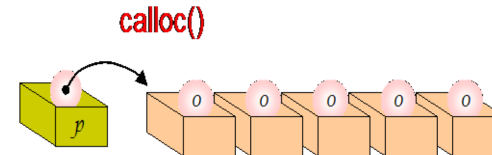
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>

4.  struct Book {
5.      int number;
6.      char title[10];
7.  };

8.  int main(void)
9.  {
10.     struct Book *p;

11.     p = (struct Book *)malloc(2 * sizeof(struct Book));

12.     if(p == NULL){
13.       printf("메모리 할당 오류\n") ;
14.       exit(1);
15.     }

16.     p->number = 1;
17.     strcpy(p->title,"C Programming");

18.     (p+1)->number = 2;
19.     strcpy((p+1)->title,"Data Structure");

20.     free(p);
21.     return 0;
22. }
```

calloc()과 realloc()

```
void *calloc(size_t n, size_t size);
```

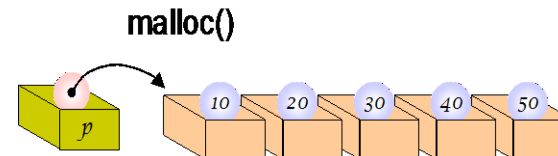
- calloc()은 malloc()과는 다르게 0으로 초기화된 메모리 할당
- 항목 단위로 메모리를 할당
- (예)
- int *p;
- p = (int *)calloc(5, sizeof(int));

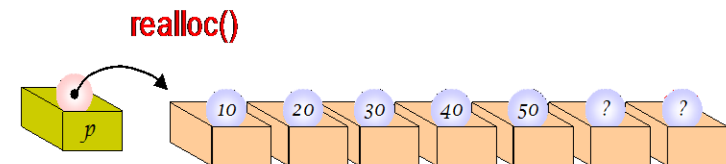





```
void *realloc(void *memblock, size_t size);
```

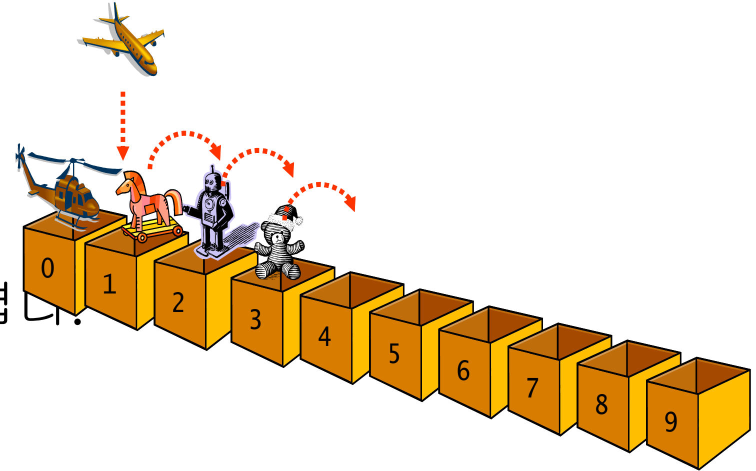
- realloc() 함수는 할당하였던 메모리 블록의 크기를 변경
- (예)
- int *p;
- p = (int *)malloc(5 * sizeof(int)));
- p = realloc(p, 7 * sizeof(int)));



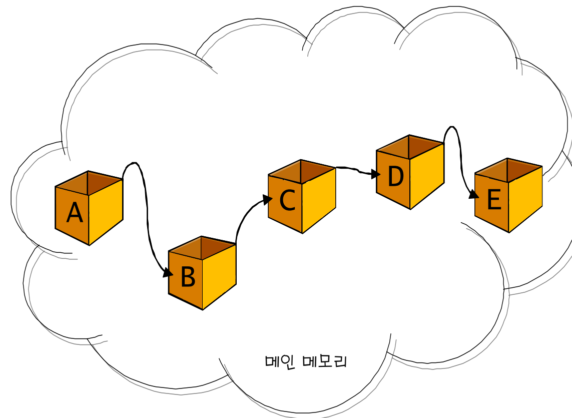


연결 리스트

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

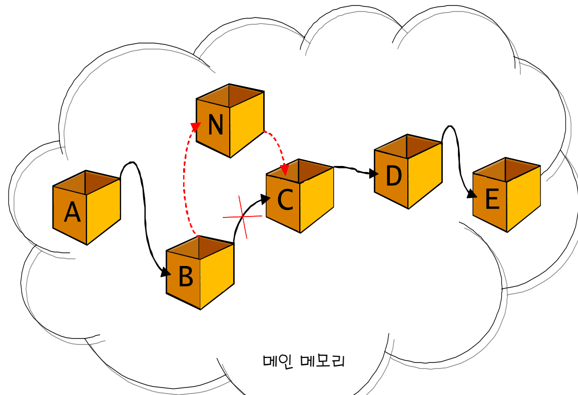


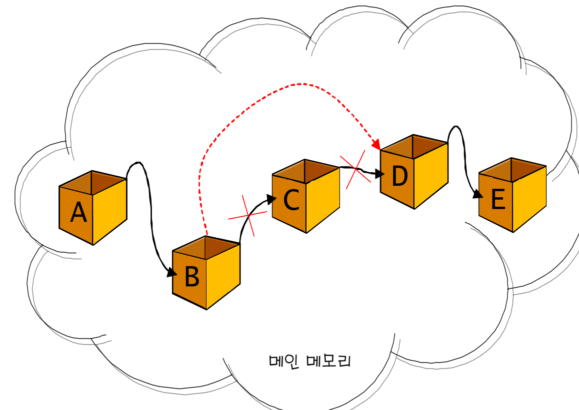
- 연결 리스트(linked list)
  - 각각의 원소가 포인터를 사용하여 다음 원소의 위치를 가리킨다.



연결 리스트의 장단점

- 중간에 데이터를 삽입, 삭제하는 경우

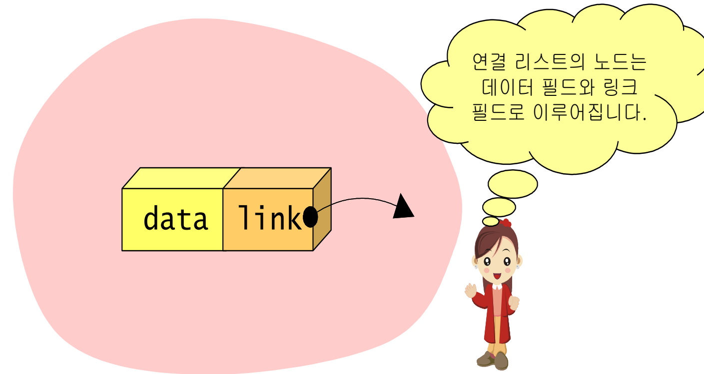




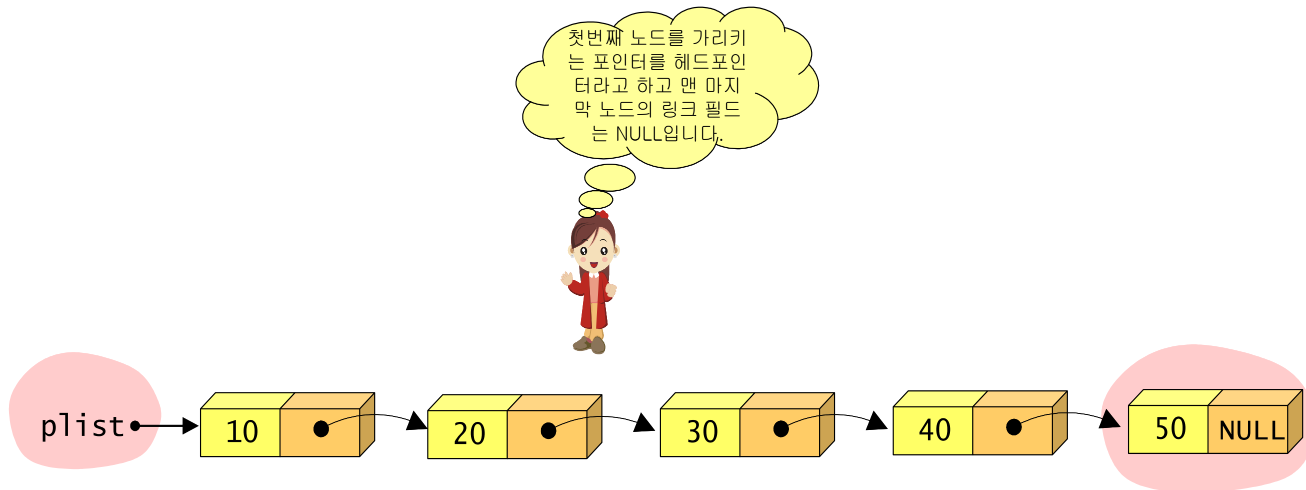
- 데이터를 저장할 공간이 필요할 때마다 동적으로 공간을 만들어서 쉽게 추가
- 구현이 어렵고 오류가 나기 쉽다.

연결 리스트의 구조

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



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



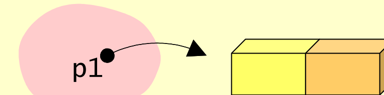
자기 참조 구조체

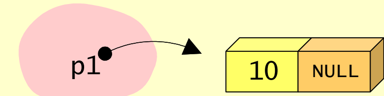
- **자기 참조 구조체(self-referential structure)**는 특별한 구조체로서 구성 멤버 중에 같은 타입의 구조체를 가리키는 포인터가 존재하는 구조체

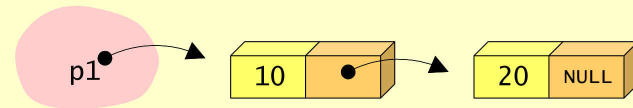
```
// 데이터의 정의
typedef struct data {
            int id;
            char name[20];
            char phone[12];
} DATA;

// 노드의 정의
typedef struct NODE {
            DATA 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);
```





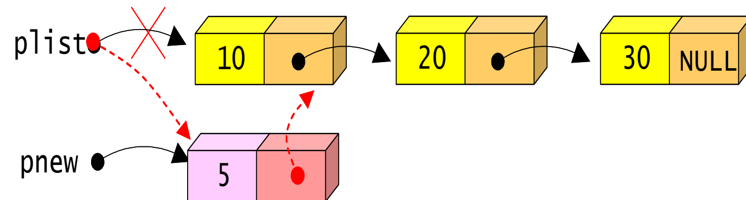


연결 리스트의 삽입 연산

```
NODE *insert_NODE(NODE *plist, NODE *pprev, DATA item);
```

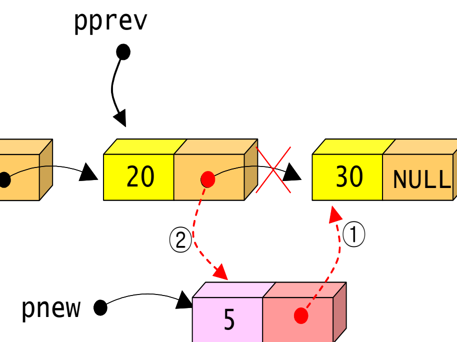
1. 리스트의 처음에 삽입하는 경우

```
pnew->link = plist;
plist = pnew;
```



2. 리스트의 중간에 삽입하는 경우

```
pnew->link = pprev->link;// ①
pprev->link = pnew;// ②
```



연결 리스트의 삽입 연산

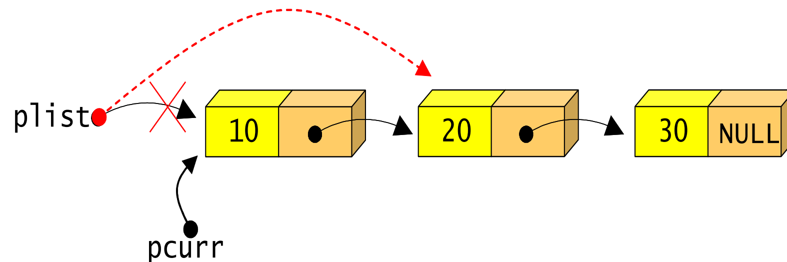
```
NODE *insert_node(NODE *plist, NODE *pprev, DATA item)
{
        NODE *pnew = NULL;

        if( !(pnew = (NODE *)malloc(sizeof(NODE))) )
        {
                printf("메모리 동적 할당 오류\n");
                exit(1);
        }

        pnew->data = data;
        if( pprev == NULL )    // 연결 리스트의 처음에 삽입
        {
                pnew->link = plist;
                plist = pnew;
        }
        else                   // 연결 리스트의 중간에 삽입
        {
                pnew->link = pprev->link;
                pprev->link = pnew;
        }
        return plist;
}
```

연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr);
```

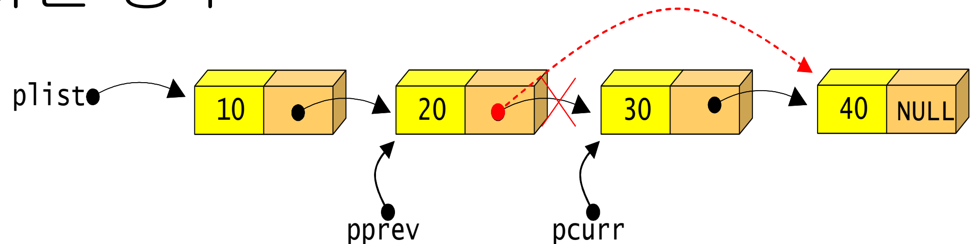
1. 리스트의 처음을 삭제하는 경우

```
plist = pcurr->link;
free(pcurr);
```



2. 리스트의 중간에 삽입하는 경우

```
pprev->link = pcurr->link;
free(pcurr);
```

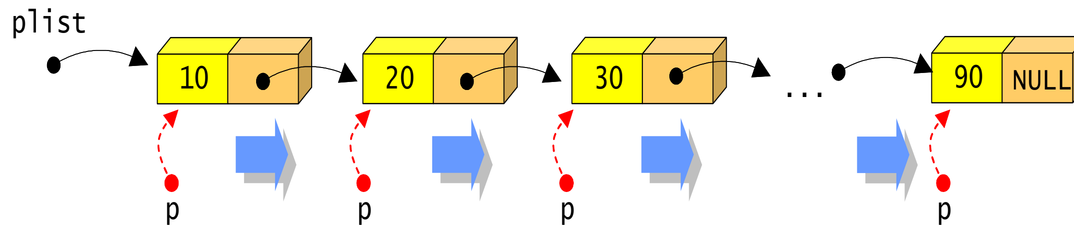


연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr)
{
		if( pprev == NULL )
				plist = pcurr->link;
		else
				pprev->link = pcurr->link;

		free(pcurr);
		return plist;
}
```

연결 리스트의 순회 연산

```
void print_list(NODE *plist)
{
        NODE *p;

        p = plist;
        printf("( ");

        while( p )
        {
                printf("%d ", p->data);
                p = p->link;
        }
        printf(")\n");
}
```



노드의 개수 세기

```
1.  int get_length(NODE *plist)
2.  {
3.          NODE *p;
4.          int length = 0;

5.          p = plist;

6.          while( p )
7.          {
8.                  length++;
9.                  p = p->link;
10.         }
11.         printf("리스트의 길이는 %d\n", length);
12.         return length;
13. }
```

합계 구하기

```
1.  int get_sum(NODE *plist)
2.  {
3.          NODE *p;
4.          int sum = 0;

5.          p = plist;

6.          while( p )
7.          {
8.                  sum += p->data;
9.                  p = p->link;
10.         }
11.         printf("리스트의 합계는 %d\n", sum);
12.         return sum;
13. }
```

Q & A