연결구조 구현하기 (코딩)

■ 들어가기에 앞서

○ 본 자료는 포인터를 이용하여 연결 구조(연결 리스트, 트리 등)를 구현(코딩)하기 위해 필요한 내용을 담고 있다.

- 단, 포인터와 함수에 대한 최소한의 내용은 알고 있어야 한다.

○ 코드와 메모리 그림을 같이 보면서 학습하고, 특히 메모리 그림을 확실하게 이해하고 숙지해야 한다.

■ 메모리 그림에서 표현된 변수의 구성요소

○ 변수는 도형으로 표시

- 포인터 변수임을 확실하게 표현하기 위해 포인터 변수는 둥근 사각형으로 표시하고, 그 외의 변수는 각진 사각형으로 표시

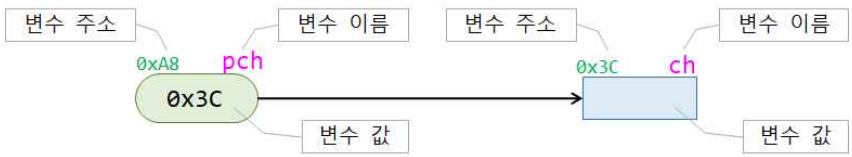
○ 변수 이름은 도형의 우상단에 표시하고, 변수의 주소는 도형의 좌상단에 표시

○ 변수에 저장되는 값은 도형 안에 표시

- 비어 있는 경우는 변수가 초기화가 안 되어 쓰레기 값을 저장하고 있음을 의미
- 포인터 변수의 경우, 저장된 값에 의해 가리키는 메모리를 화살표로 표시

(예제)

```
char ch;          // ch에는 쓰레기 값이 저장되어 있음
char *pch = &ch;  // ch의 주소를 pch에 대입
```

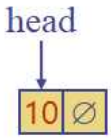


※ 포인터의 경우, 변수에 '저장된 주소 값'과 변수 '자체의 주소 값'를 혼동하지 않도록 주의
위 예제에서 변수 pch 에 저장된 값은 0x3C 이고, 변수 pch의 주소는 0xA8 이다.

■ 코딩할 기본 내용

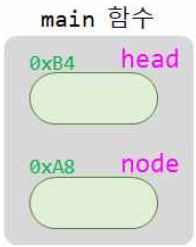
○ 하나의 노드를 가지는 단순 연결 리스트 만들기

- 노드의 데이터는 정수 10
- 노드를 가리키는 포인터(헤드 포인터) 유지
- 헤더 노드는 없음



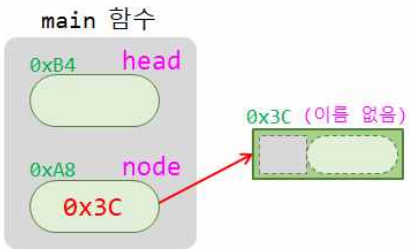
◘ 단계 1: 함수 사용 안 하는 버전으로 작성하기

```
 1: typedef struct Node{
 2:    struct Node *next;
 3:    int data;
 4: } Node;                  // 구조체 Node 정의
 5:
 6: int main() {
 7:    Node *head, *node;        // A 두 개의 구조체 포인터 변수 선언
 8:
 9:    node = (Node *) malloc(sizeof(Node));  // B 노드 할당 및 주소 대입
10:    node->data = 10;                  // 노드의 멤버 변수에 값 대입
11:    node->next = NULL;                // C 노드의 멤버 변수에 값 대입
12:
13:    head = node;              // D 변수 node에 저장된 값을 변수 head에 대입
14:
15:    printf("%d\n", head->data);    // 출력. node->data를 출력해도 동일함
16:
17:    free(head);      // 할당받은 메모리 해제. free(node)라고 해도 동일함
18:
19:    return 0;
20: }
```



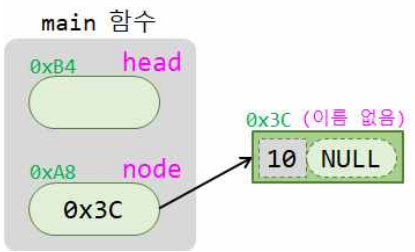
A 라인 7 실행 후

- 두 개의 포인터 변수 정적 할당



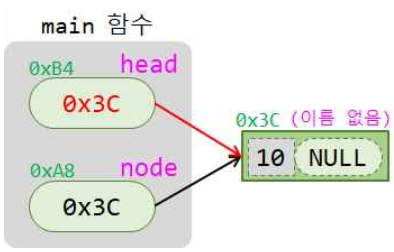
B 라인 9 실행 후

- 구조체 동적 할당 및 연결
- 주의!! node는 할당된 구조체 변수의 이름이 아님



C 라인 11 실행 후

- node를 이용하여 구조체에 접근



D 라인 13 실행 후

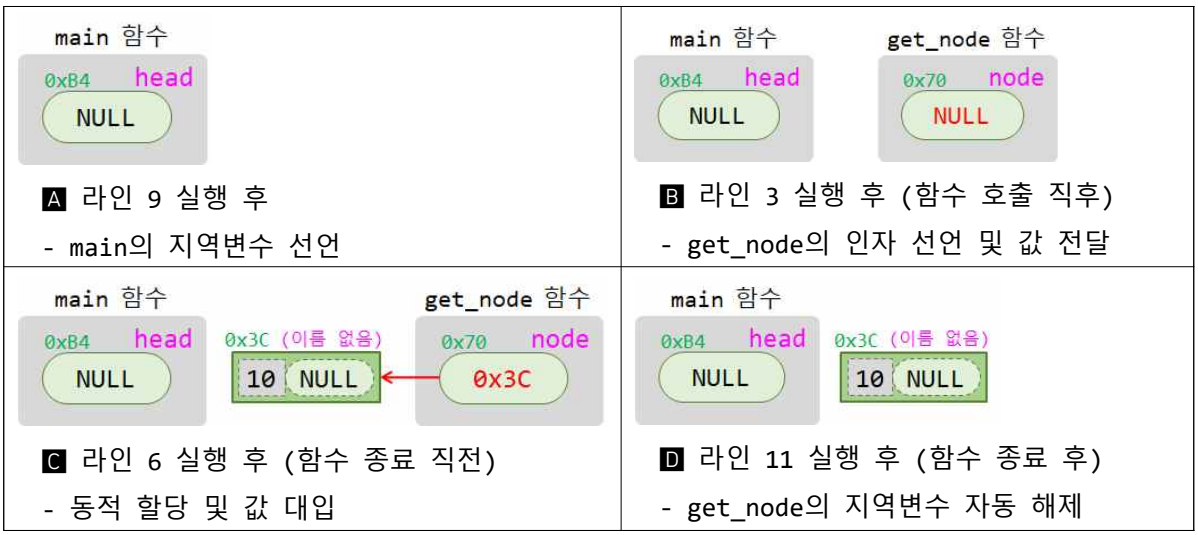
- node의 값을 head에 대입

※ 참고 : 변수 node를 사용하지 않고 할당된 노드를 바로 head에 연결해도 되지만, 다음에 보게 될 코드와 비교하기 위해 변수 node를 사용하였다.

◘ 단계 2: 노드 할당하고 초기화하는 부분을 함수로 작성하기

○ 다음과 같이 코드를 작성하면? **정상 작동하지 않는다.** (이유는 아래 메모리 그림 참조)

```
1: typedef struct Node{ ... } Node; // 구조체 Node 정의
2:
3: void get_node(Node *node) {      // B 함수 시작 (인자 선언 및 인자 값 대입)
4:    node = (Node *) malloc( sizeof(Node) );   // 노드 할당 및 주소 대입
5:    node->data = 10;                  // 멤버 변수에 값 대입
6:    node->next = NULL;                // C 멤버 변수에 값 대입
7: }
8: int main() {
9:    Node *head = NULL;             // A 구조체 포인터 변수 선언
10:
11:   get_node(head);                // D head에 노드 연결 (?)
12:
13:   printf("%d\n", head->data);    // 여기서 에러 발생
14:
15:   free(head);      // 할당받은 메모리 해제
16:
17:   return 0;
18: }
```



○ get_node의 인자 node는 get_node의 지역 변수이므로,
이 값이 바뀌는 것은 main의 변수 head에 아무런 영향을 주지 않는다.
따라서 **main의 변수 head의 값은 함수 호출 후에도 여전히 NULL**이고,
라인 13에서 head가 가리키는 구조체를 참조하려할 때, 실행 오류가 발생한다.
(get_node의 인자 이름을 node 대신 head로 해도 마찬가지)

○ get_node의 인자 node는 아무런 역할을 하지 않아, 인자가 없어도 동작과정은 동일하다.

○ 또한, 함수가 종료된 후에는 동적 할당된 구조체에 접근할 수 있는 수단이 없다.

※ 해결해야 할 문제

: 함수 호출 시, main의 변수 head의 값을 동적 할당된 구조체의 주소로 변경해야 한다.

(위 그림에서 main의 head의 값을 0x3C로 변경해야 함)

⛒ 방법 A : 함수의 리턴 값을 활용하는 방법

- 함수에서 구조체의 주소 값을 리턴하고, main에서는 리턴된 값을 head에 대입
  - malloc() 함수 사용법과 유사

```
 1: typedef struct Node{ ... } Node; // 구조체 Node 정의
 2:
 3: Node *get_node() {              // 함수 시작
 4:    Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
 5:    node->data = 10;                  // 멤버 변수에 값 대입
 6:    node->next = NULL;                // A 멤버 변수에 값 대입
 7:
 8:    return node;                      // node에 저장된 값 리턴
 9: }
10: int main() {
11:    Node *head = NULL;             // 구조체 포인터 변수 선언
12:
13:    head = get_node();          // B 함수 호출 (head에 노드 추가)
14:
15:    printf("%d\n", head->data);    // 정상적으로 출력
16:
17:    free(head);      // 할당받은 메모리 해제
18:
19:    return 0;
20: }
```

A 라인 6 실행 후 (함수 종료 직전)
- 동적 할당 및 값 대입

B 라인 13 실행 후 (함수 종료 후)
- **함수의 리턴 값이 head에 저장**

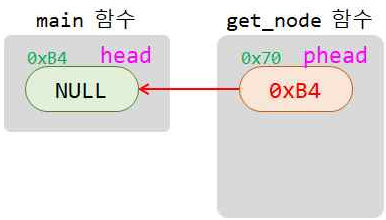
- 함수 종료 직전까지의 동작과정은 이전과 동일하고, 함수 종료 시 **get_node의 node에 저장된 값(구조체의 주소)이 리턴되어** 라인 13에서 **main의 변수 head에 저장**
- main에서 get_node로 전달해야 할 정보가 없기 때문에, 인자는 필요 없음

※ 이 방법은 코드 형태도 간단하고 이해하기 쉽지만,
main에 리턴해야 할 값이 두 개 이상인 경우에는 사용할 수 없다.

⛒ 방법 B : 이중 포인터를 활용하는 방법

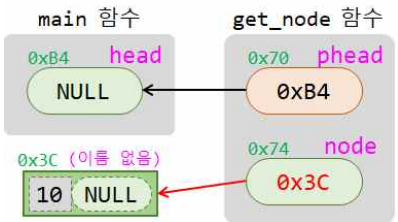
○ '변수 head의 주소를 인자로 전달'하여, get_node 함수에서 직접 head의 값 변경

```
 1: typedef struct Node{ ... } Node; // 구조체 Node 정의
 2:
 3: void get_node(Node **phead) {     // A 함수 시작 (head의 주소 대입됨)
 4:    Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
 5:    node->data = 10;                  // 멤버 변수에 값 대입
 6:    node->next = NULL;                // B 멤버 변수에 값 대입
 7:
 8:    *phead = node;                    // C phead가 가리키는 변수에 node의 값 대입
 9: }
10: int main() {
11:    Node *head = NULL;                // 구조체 포인터 변수 선언
12:
13:    get_node( &head );                // D 함수 호출 (head의 주소 전달)
14:
15:    printf("%d\n", head->data);       // 정상적으로 출력
16:
17:    free(head);       // 할당받은 메모리 해제
18:
19:    return 0;
20: }
```



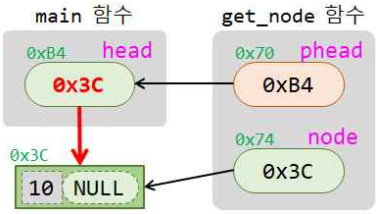
A 라인 3 실행 후 (함수 호출 직후)

- get_node의 인자 phead에 main에서 전달된 head의 주소 대입



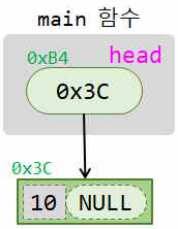
B 라인 6 실행 후

- node에 동적할당된 구조체 주소 값 대입 및 구조체 멤버에 값 대입



C 라인 8 실행 후 (함수 종료 직전)

- **phead를 이용하여 head의 값 변경**



D 라인 13 실행 후 (함수 종료 후)

- get_node 함수의 지역변수 자동 해제

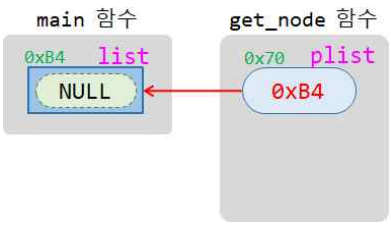
○ 이 방법의 핵심은 **head의 주소를 get_node 함수에 인자로 전달**하여, get_node 함수에서 **간접 참조를 통해 변수 head에 저장된 값을 변경**한다는 점이다.

※ 이 방법은 함수에 의해 변경되어야 할 값이 두 개 이상인 경우에도 사용할 수 있지만, 이중 포인터를 이해하고 사용할 줄 알아야 한다.

⛒ 방법 C : head를 구조체로 감싸는 방법

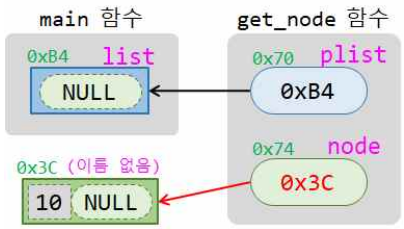
○ head의 자체의 주소가 아니라 'head를 감싼 구조체의 주소를 전달'하여, head의 값 변경

```
 1: typedef struct Node{ ... } Node; // 구조체 Node 정의
 2:
 3: typedef struct List{
 4:    struct Node *head;
 5: } List;                  // 구조체 List 정의
 6:
 7: void get_node(List *plist) {      // A 함수 시작 (list의 주소 대입됨)
 8:    Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
 9:    node->data = 10;               // 멤버 변수에 값 대입
10:    node->next = NULL;             // B 멤버 변수에 값 대입
11:
12:    plist->head = node;    // C plist가 가리키는 구조체의 head에 node의 값 대입
13: }
14: int main() {
15:    List list = {NULL};            // A list 구조체 선언 및 초기화 (정적할당)
16:
17:    get_node( &list );             // D 함수 호출 (list의 주소 전달)
18:
19:    printf("%d\n", list.head->data);    // 정상적으로 출력
20:
21:    free(list.head);    // 할당받은 메모리 해제
22:
23:    return 0;
24: }
```



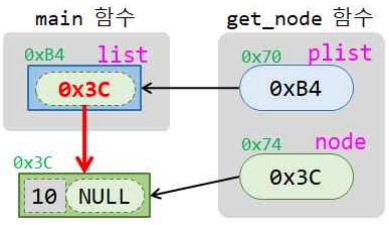
A 라인 7 실행 후 (함수 호출 직후)

- get_node의 인자 phead에 main에서 전달된 head의 주소 대입



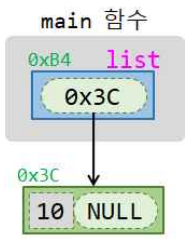
B 라인 10 실행 후

- node에 동적할당된 구조체 주소 값 대입 및 구조체 멤버에 값 대입



C 라인 12 실행 후 (함수 종료 직전)

- **phead를 이용하여 head의 변수 값 변경**



D 라인 17 실행 후 (함수 종료 후)

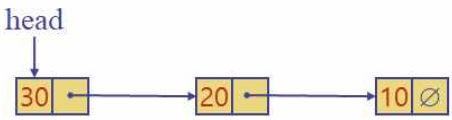
- get_node 함수의 지역변수 자동 해제

○ 이 방법은 방법 B와 본질적으로 동일하고, 문법적 측면에서 이중 포인터를 피할 수 있다는 점과 List 라는 자료형을 따로 정의함으로써 의미적으로 명확해 진다는 장점이 있다. 하지만, 구조체로 감싸서 오히려 코드가 복잡해지는 단점도 있다.

■ 응용 문제

○ n개의 정수를 입력받아, 리스트의 맨 앞에 차례로 추가하는 프로그램을 작성하라.

- 예를 들어, 10, 20, 30이 입력되면, 최종 리스트 모양은 30 -> 20 -> 10
- 앞서 설명한 방법으로 함수 작성



⛒ 방법 A

```c
typedef struct Node{
   struct Node *next;
   int data;
} Node;                                 // 구조체 Node 정의

Node *insert_first(Node *head, int data) {     // 함수 시작
   Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
   node->data = data;                   // 멤버 변수에 값 대입
   node->next = head;                   // 멤버 변수에 값 대입

   return node;                         // node에 저장된 값 리턴
}
void print_list(Node *node) {           // 리스트의 값을 차례로 출력하는 함수
   while( node ){
      printf("%d\n", node->data);       // 값 출력
      node = node->next;                // 다음 노드
   }
}
void free_list(Node *node) {        // 리스트에 할당된 노드들을 해제하는 함수
   if( node ){
      free_list(node->next);        // 현재 노드의 뒷부분 해제
      free(node);                   // 현재 노드 해제
   }
}
int main() {
   Node *head = NULL;               // 구조체 포인터 변수 선언
   int n, data, i;

   scanf("%d", &n);                 // 입력받을 정수의 개수

   for( i=0; i < n ; ++i ){
      scanf("%d", &data);
      head = insert_first(head, data);    // 리스트의 맨 앞에 노드 추가
   }

   print_list(head);           // 리스트 값 출력
   free_list(head);            // 할당받은 메모리 해제

   return 0;
}
```

⛒ 방법 B

```
 1: typedef struct Node{
 2:    struct Node *next;
 3:    int data;
 4: } Node;                              // 구조체 Node 정의
 5:
 6: void insert_first(Node **phead, int data) {    // 함수 시작
 7:    Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
 8:    node->data = data;                // 멤버 변수에 값 대입
 9:    node->next = *phead;              // 멤버 변수에 값 대입
10:
11:    *phead = node;                   // 새로 삽입한 노드를 head 로 지정
12: }
13: void print_list(Node *node) {
14:    while( node ){
15:       printf("%d\n", node->data);    // 값 출력
16:       node = node->next;             // 다음 노드
17:    }
18: }
19: void free_list(Node *node) {
20:    if( node ){
21:       free_list(node->next);      // 현재 노드의 뒷 부분 해제
22:       free(node);                 // 현재 노드 해제
23:    }
24: }
25: int main() {
26:    Node *head = NULL;             // 구조체 포인터 변수 선언
27:    int n, data, i;
28:
29:    scanf("%d", &n);               // 입력받을 정수의 개수
30:
31:    for( i=0; i < n ; ++i ){
32:       scanf("%d", &data);
33:       insert_first(&head, data);          // 리스트의 맨 앞에 노드 추가
34:    }
35:
36:    print_list(head);        // 리스트 값 출력
37:    free_list(head);         // 할당받은 메모리 해제
38:
39:    return 0;
40: }
```

⛒ 방법 C

```c
typedef struct Node{
   struct Node *next;
   int data;
} Node;                              // 구조체 Node 정의

typedef struct List{
   struct Node *head;
} List;                 // 구조체 List 정의

void insert_first(List *list, int data) {    // 함수 시작
   Node *node = (Node *) malloc( sizeof(Node) );  // 노드 할당 및 주소 대입
   node->data = data;                // 멤버 변수에 값 대입
   node->next = list->head;          // 멤버 변수에 값 대입

   list->head = node;                // 새로 삽입한 노드를 head 로
}
void print_list(Node *node) {
   while( node ){
      printf("%d\n", node->data);    // 값 출력
      node = node->next;             // 다음 노드
   }
}
void free_list(Node *node) {
   if( node ){
      free_list(node->next);      // 현재 노드의 뒷 부분 해제
      free(node);                 // 현재 노드 해제
   }
}
int main() {
   List list = {NULL};            // list 구조체 선언 및 초기화 (정적할당)
   int n, data, i;

   scanf("%d", &n);

   for( i=0; i < n ; ++i ){
      scanf("%d", &data);
      insert_first(&list, data);          // head에 노드 추가
   }

   print_list(list.head);         // 리스트 값 출력
   free_list(list.head);          // 할당받은 메모리 해제

   return 0;
}
```

※ 참고

객체 지향적인 관점과 코드의 일관성을 위해서는
print_list와 free_list 함수의 인자로 리스트 구조체의 정보를 사용하는 것이
더 바람직하지만, 여기서는 코드의 단순화를 위해서 노드의 정보를 인자로 사용