
# 프로그래밍 기초

## 16장. 동적 메모리와 전처리 Part 1

# 학습목표

- **동적 메모리 할당 방식을 이해하고 설명할 수 있다.**
  - 정적 할당과 동적 할당의 필요성
  - 함수 malloc(), calloc(), realloc(), free()의 사용
  - 동적 메모리 할당으로 구현하는 자기참조 구조체

- **연결 리스트를 이해하고 설명할 수 있다.**
  - 연결 리스트의 이해와 구현
  - 연결 리스트의 노드, 헤드, 순회, 삽입, 삭제
  - 여러 파일로 구성하는 프로젝트와 사용자 정의 헤더파일
  - 연결 리스트의 구현

- **전처리 지시자를 이해하고 설명할 수 있다.**
  - 매크로와 인자를 활용한 매크로
  - #if와 #endif
  - #ifdef와 #endif, #ifndef #endif
  - 전처리 연산자 #, #@, ##, defined

# 정적, 동적 메모리 할당

- **정적 메모리 할당(static memory allocation)**
  - 프로그램이 실행되기 이전에 저장 공간의 수나 크기를 정하는 메모리 할당 방법
    - ➢ 메모리 크기와 개수는 컴파일 시점에 결정됨
    - ➢ 프로그램 또는 함수가 시작되면 메모리에 할당되어 사용
      - – 해당 함수나 프로그램이 종료되면 변수가 메모리에서 삭제되는 방식
    - ➢ 사용이 간편하고, 관리가 단순
    - ➢ 사용하지 않는 변수나 과도하게 큰 메모리 선언으로 메모리 낭비를 초래할 수 있음

- **동적 메모리 할당(dynamic memory allocation)**
  - **프로그램 실행 중에 필요한 메모리를 할당하는 방법**
    - ➢ 정적 할당 방식인 변수 선언에 비해 상대적으로 다소 어려움
  - 메모리 사용 예측이 정확하지 않고 실행 중에 메모리 할당이 필요한 경우에 적합한 방법

정적 메모리 할당 방식

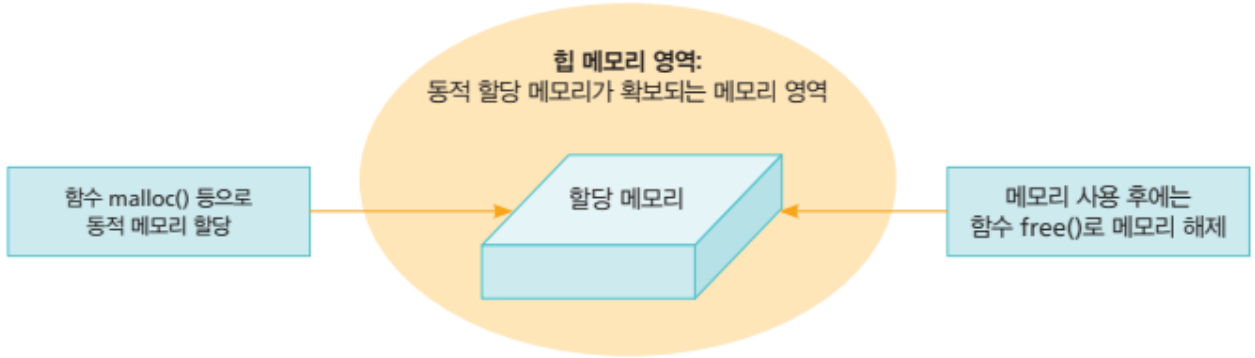
```
int i;
long prod = 1;
int facto[6];
char *lang[] = {"Algol", "Pascal",
                "C", "C++"};
```

동적 메모리 할당 방식

```
int *pi = NULL;

pi = (int *)malloc(sizeof(int));
*pi = 7;
```

# 동적 메모리 관련 함수

- **동적 메모리**
  - 함수 malloc()의 호출로 힙(heap) 영역에 확보
  - 메모리는 사용 후 함수 free()로 해제
    - ➢ 메모리 해제를 하지 않으면 메모리 부족과 같은 문제를 발생




- **동적 메모리 할당 함수**
  - malloc(), calloc(), realloc()
  - 반환 형: void 포인터(void *)
    - ➢ 메모리 할당을 요구한 자료형의 포인터로 변환이 필요
    - ➢ 헤더 파일 stdlib.h 필요
  - 동적으로 할당된 메모리를 해제 함수: free()

```
int *pi = NULL;        (int *)로 형 변환

pi = (int *)malloc(sizeof(int));
*pi = 7;
. . .

free(pi);
```

# 동적 메모리 관련 함수

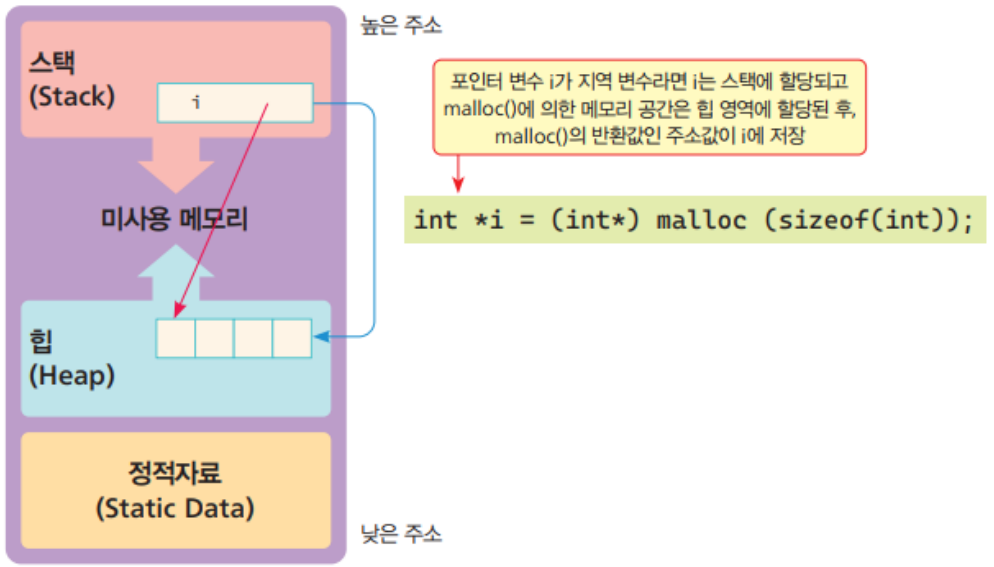
## 동적 메모리 관련 함수

- #include <stdlib.h> 필요

| 메모리 연산 | 초기값 | 함수 원형 | 기능 |
|---|---|---|---|
| 메모리 할당 | 없음 | `void *malloc(size_t size)` | 지정된 크기(byte) 만큼 메모리 할당<br>반환값: 생성된 메모리 주소<br>- size: 할당할 메모리 크기(byte) |
| | 0 | `void *calloc(size_t num, size_t size)` | 할당된 메모리 공간을 0으로 초기화<br>- num: 생성할 요소의 개수<br>- size: 요소의 크기 |
| 기존 메모리 크기 변경 | 이전 값 | `void *realloc(void *ptr, size_t new_size)` | malloc이나 calloc으로 할당된 메모리 블록의 크기를 변경<br>- ptr: 크기를 변경할 메모리 포인터<br>- new_size: 변경할 새 메모리 크기 |
| 메모리 해제 | 해당 없음 | `void free(void *ptr)` | 동적 할당된 메모리 공간(ptr)을 해제<br>- ptr: 해제할 메모리 포인터 |

# 메모리 할당 영역

- **메모리 영역**
  - 힙(Heap) 영역
    - 동적 메모리가 할당되는 부분
  - 데이터(Data) 영역
    - 전역 변수와 정적 변수가 할당되는 저장공간
  - 스택(Stack) 영역
    - 함수 호출에 의한 형식 매개변수
    - 함수 내부의 지역변수가 할당되는 저장공간



# 함수 malloc()

- **동적 메모리 할당 기본 함수 malloc()**
  - void *malloc(size_t size);
    - 인자: 할당할 메모리 크기를 지정
    - 형 변환(type casting)이 필요

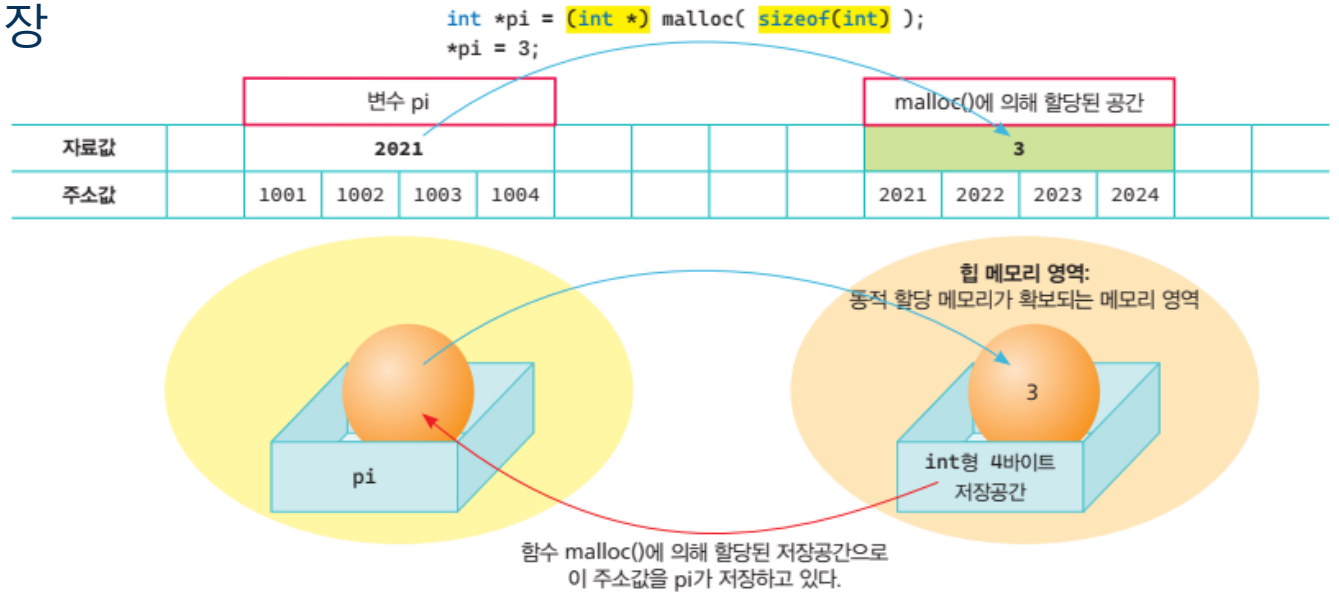
```
int *pi = (int *)malloc(sizeof(int));
*pi = 3;
```

할당할 메모리 크기

해당 자료형의 포인터로 형 변환 필요

  - 반환
    - 성공하면, 할당된 메모리의 시작 주소(void 포인터로 반환)
    - 실패하면 NULL을 반환

  - int *의 변수에 할당된 메모리의 주소 저장
    - 간접 연산자 *pi를 이용
    - 원하는 값을 수정 가능



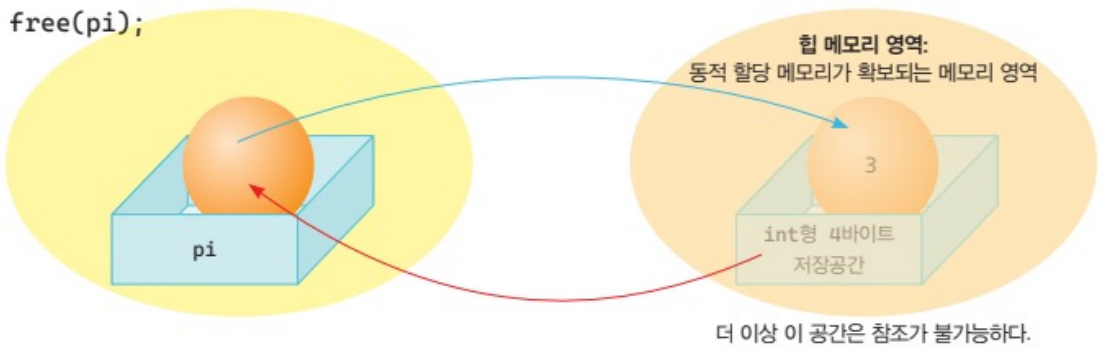
# 함수 free()

- **메모리 해제에 이용되는 함수: free(void *)**

```
int *pi = (int *)malloc(sizeof(int));

*pi = 3;

free(pi);
```

할당한 메모리 해제

- **free(pi)**
  - 함수 malloc()의 반환 주소를 저장한 변수 pi를 해제
    - ➢ 인자로 해제할 메모리의 주소를 갖는 포인터를 이용
  - 변수 pi가 가리키는 4바이트의 자료값이 해제되어 더 이상 사용할 수 없음



# 함수 malloc()을 이용하여 int형 저장공간을 확보하여 처리

- **함수 malloc()을 이용하여 int형 메모리 공간을 하나 할당**
  - 만일 메모리 공간이 부족하여 요청된 공간을 할당하지 못한다면, 함수 malloc()은 NULL을 반환
  - 값 7을 저장한 후 출력

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

int main(void)
{
    int* pi = NULL;
    pi = (int*) malloc(sizeof(int)); //동적메모리 할당

    if (pi == NULL) //동적메모리 할당 검사
    {
        printf("메모리 할당에 문제가 있습니다.");
        exit(1);
    }
    *pi = 7; //동적 메모리에 7 저장
    printf("주소: pi = %p, 저장 값: *p = %d\n", pi, *pi);

    free(pi); //동적 메모리 해제
    return 0;
}
```

메모리 공간이 부족하여 요청된 공간을 할당하지 못하면 NULL을 반환

```
주소: pi = 0x155004100, 저장 값: *pi = 7
```

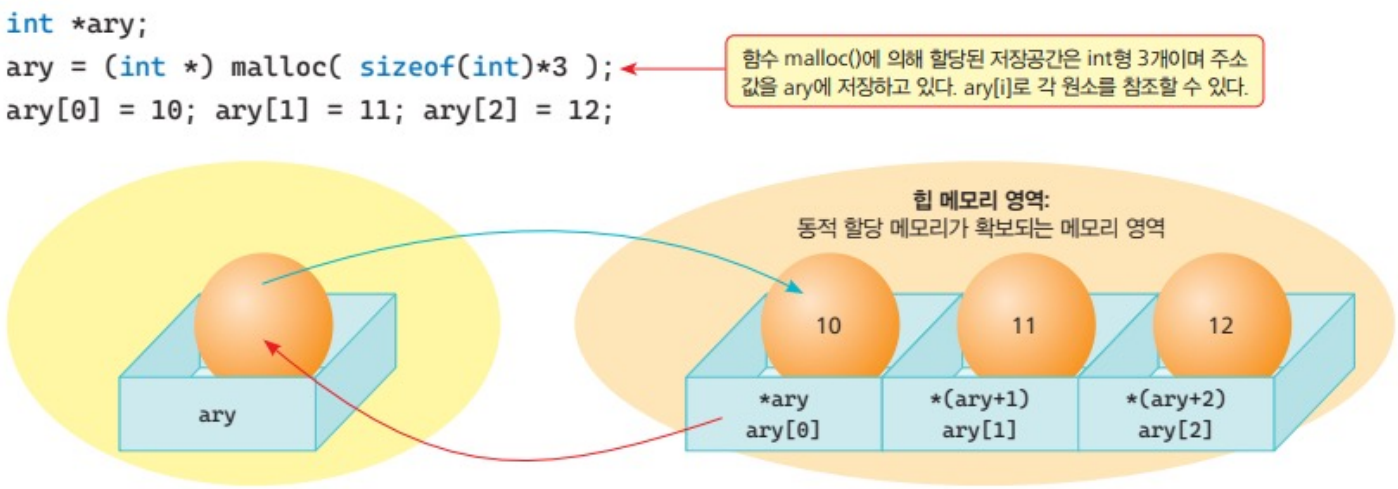
# 함수 malloc()에 의한 배열 공간 할당

- **배열과 같이 여러 메모리 공간을 동적으로 할당하는 방법**
  - int형 배열을 위한 메모리 공간 할당 방법
    - malloc(sizeof(int) * 개수)

```
int *ary = (int *)malloc(sizeof(int) * 3);
```

  - 변수 ary를 이용하여 다음과 같이 배열 형태나 주소 형태로 메모리 접근 가능

```
ary[0] = 10; ary[1] = 11; ary[2] = 12;
```



# 실습 예제 2: 배열을 위한 메모리 할당 (실습)

<02arymalloc.c>

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

int main(void)
{
    int n = 0;
    printf("입력할 점수의 개수를 입력 >> ");
    scanf("%d", &n);

    int* ary = NULL;

    // 동적 메모리 할당
    ary = (int *)malloc(sizeof(int) * n);
    if (ary == NULL)
    {
        printf("메모리 할당에 문제가 있습니다.");
        exit(1);
    }
    printf("%d개의 점수 입력 >> ", n); //점수 입력
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        scanf("%d", (ary + i));
        sum += *(ary + i); //sum += ary[i];
    }

    printf("입력 점수: ");
    for (int i = 0; i < n; i++)
        printf("%d ", *(ary + i)); // 표준입력 자료 출력
    printf("\n");

    // 결과 출력
    printf("합: %d 평균: %.1f\n", sum, (double)sum / n);
    free(ary); // 동적 메모리 해제

    return 0;
}
```

```
입력할 점수의 개수를 입력 >> 5
5개의 점수 입력 >> 89 70 67 92 99
입력 점수: 89 70 67 92 99
합: 417  평균: 83.4
```

# 구조체 배열 메모리 동적 할당 (실습)

<02malloc_memset.c>

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

typedef struct
{
    int number;
    char title[50];
} Book;

int main()
{
    Book *p = NULL;
    p = (Book *)malloc(sizeof(Book) * 2);
    if (p == NULL)
    {
        printf("메모리 할당 오류\n");
        exit(1);
    }

    // 메모리 공간 초기화
    memset(p, 0, sizeof(Book) * 2);

    p[0].number = 1;
    strcpy(p[0].title, "C Programming");

    p[1].number = 2;
    strcpy(p[1].title, "Data Structure");

    for (int i = 0; i < 2; i++)
    {
        printf("[%d] title: %s\n",
                (p + i)->number, (p + i)->title);
    }

    free(p);
    return 0;
}
```

구조체 배열 Book[2] 와 동일한 크기

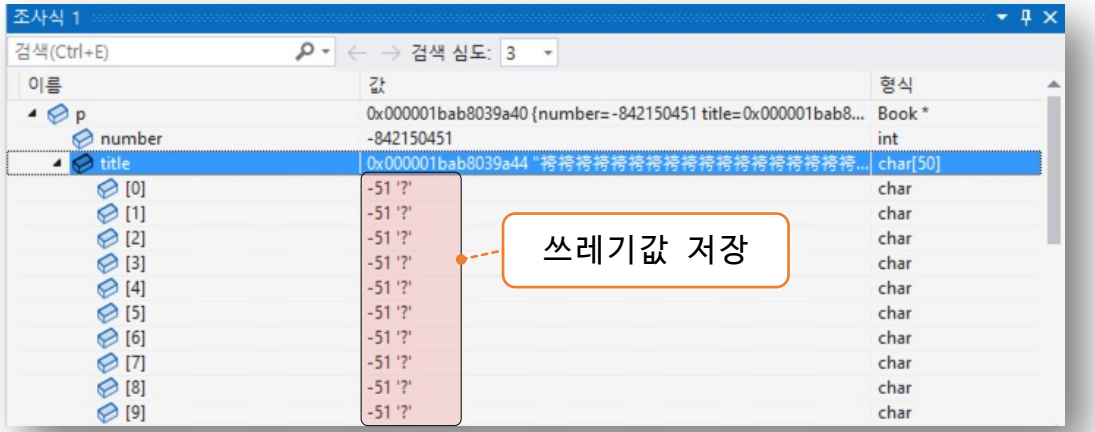
**실행 결과**

```
[1] title: C Programming
[2] title: Data Structure
```

# malloc 이후 메모리 초기화 (Visual Studio)

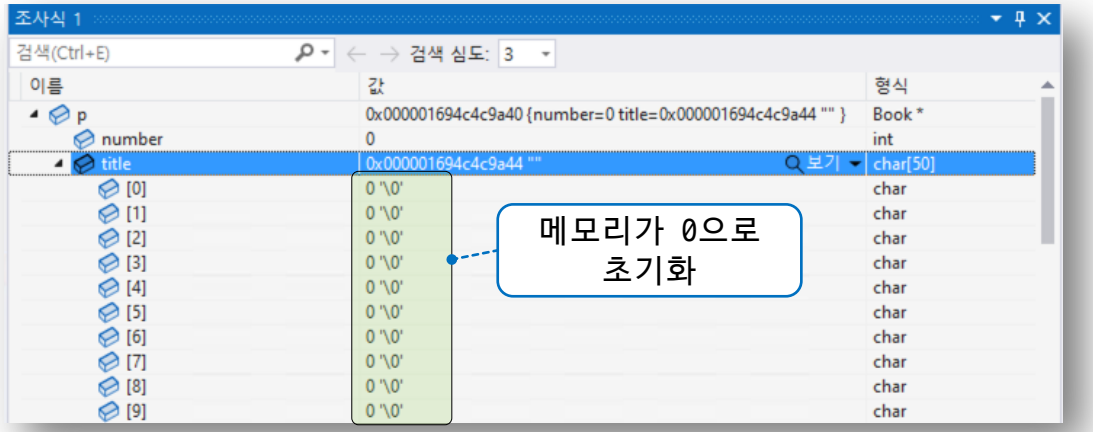
- **메모리 초기화 코드가 없는 경우**
  - 쓰레기 값이 저장될 수 있음

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




- **메모리 초기화 코드를 추가한 경우**
  - `memset(p, 0, sizeof(Book) * 2);`

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

memset(p, 0, sizeof(Book) * 2);
```



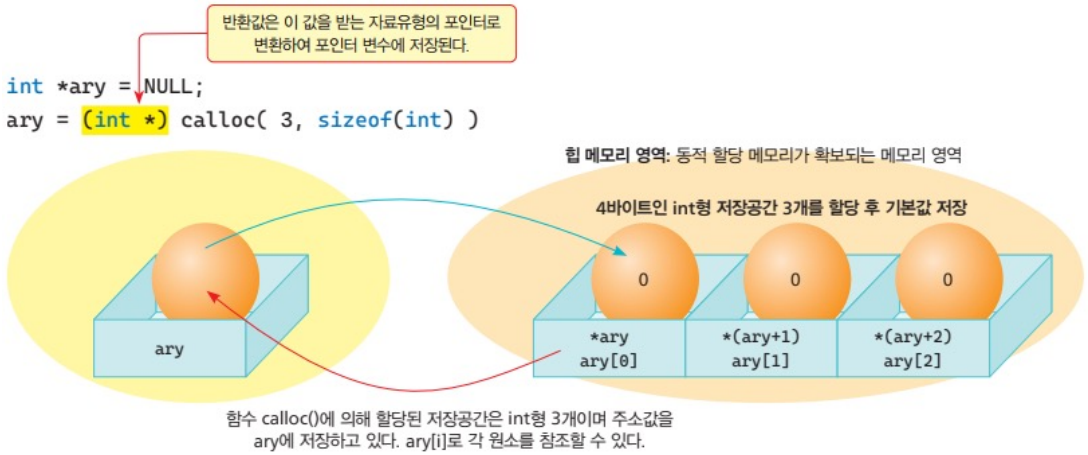
# 함수 calloc()

- **함수 calloc()**
  - 초기값을 자료형에 맞게 0으로 저장
    - 함수 malloc(): 메모리 공간을 확보하고 초기값을 저장하지 않으면 쓰레기 값이 저장
  - calloc() 함수 원형

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

    - num: 할당되는 원소의 개수
    - size: 한 원소의 크기
    - (num * size) 크기의 메모리 공간 할당 후 0으로 초기화

- **함수 호출 calloc(3, sizeof(int))로 할당**
  - int형 원소 3개
  - 저장공간에 기본값 0



# 실습 예제 16-3: 함수 calloc() 예제

<03calloc.c>

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

void myprintf(int* ary, int n);
int main(void)
{
    int* ary = NULL;
    ary = (int *)calloc(3, sizeof(int));
    if (ary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.\n");
        exit(EXIT_FAILURE);
    }
    myprintf(ary, 3); //모두 기본 값인 0 출력

    free(ary); //동적메모리 해제
    myprintf(ary, 3); //모두 쓰레기 값 출력

    return 0;
}

void myprintf(int* ary, int n)
{
    for (int i = 0; i < n; i++)
        printf("ary[%d] = %d\n", i, *(ary + i));
}
```

int형 크기 3개 할당

ary 메모리는 해제된 상황
- 쓰레기 값 출력

**실행 결과**

```
ary[0] = 0
ary[1] = 0
ary[2] = 0
ary[0] = 1883947056
ary[1] = 19324
ary[2] = 2043
```

free(ary)이후 ary 메모리에 접근

# 함수 realloc()

- **realloc(): 이미 확보한 저장공간을 새로운 크기로 변경**
  - 기존의 영역을 이용하여 그 저장 공간을 변경하는 것이 원칙
    - 새로운 메모리 영역을 다시 할당하여 이전의 값을 복사할 수도 있음
  - 성공적으로 메모리를 할당하면 변경된 저장공간의 시작 주소를 반환
    - 실패하면 NULL을 반환

  - 함수 원형

```
void *realloc(void *ptr, size_t new_size)
```

    - ptr: 변경할 저장공간의 주소
    - new_size: 변경하고 싶은 저장 공간의 총 크기

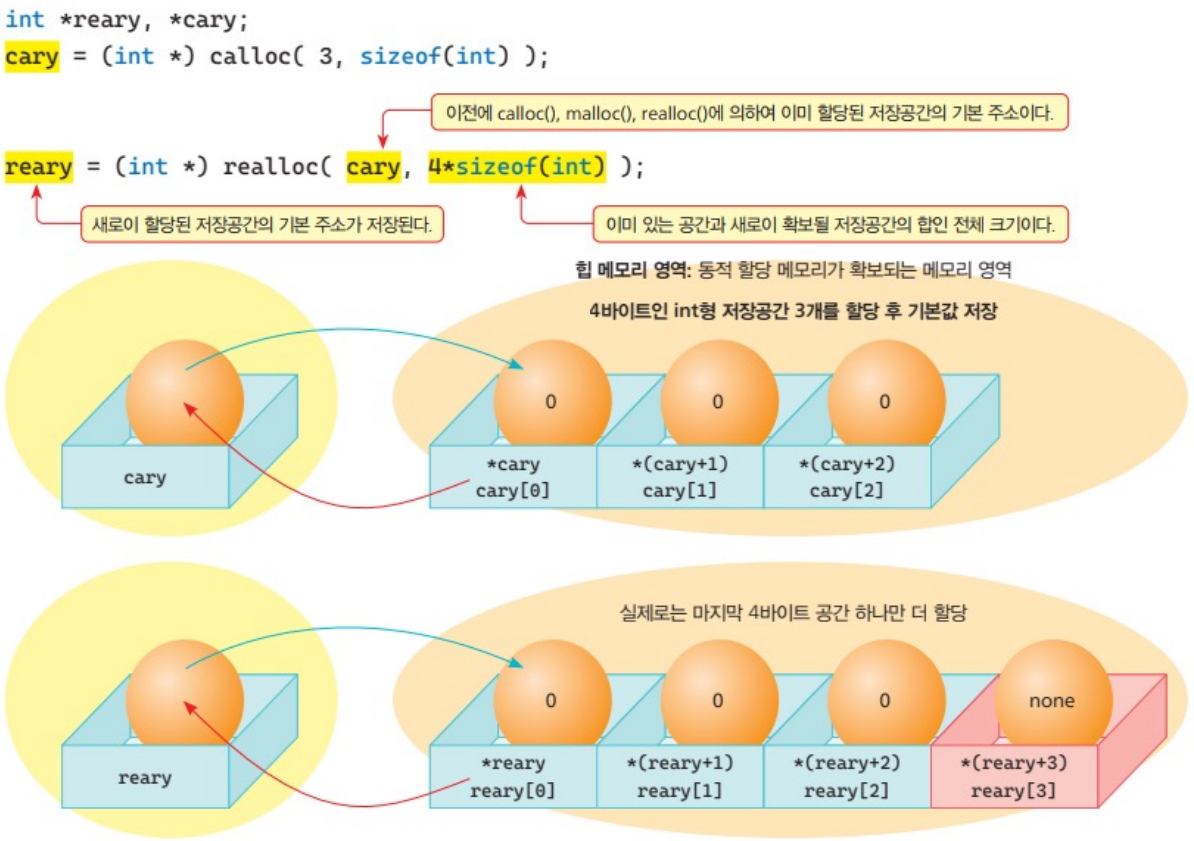
- **realloc(NULL, size)인 경우**
  - 지정된 크기만큼의 새로운 공간을 할당: malloc()과 같은 기능을 수행

# 함수 realloc() 사용

- cary
  - int형 3개의 저장공간의 시작 주소를 저장
  - 모든 값은 기본값인 0으로 저장

- reary
  - realloc(cary, 4*sizeof(int)) 호출
  - 변수 cary가 가리키는 메모리 크기 수정
  - 대부분 reary와 cary 값이 동일
    - ➢ 기존 메모리 주소에서 확장한 경우

- **함수 realloc()에 의해 확장되는 공간**
  - calloc()과 같이 0으로 저장되지 않음
  - reary[3]의 내용을 none으로 표기
    - ➢ 쓰레기 값 저장을 의미



# 실습 예제 16-4: 함수 realloc()을 이용하여 이미 할당된 메모리를 변경

<04realloc.c>

```c
#include <stdio.h>
#include <stdlib.h>
void myprintf(int *ary, int n);

int main(void)
{
    int *reary, *cary;

    cary = (int *)calloc(3, sizeof(int));
    if (cary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.\n");
        exit(EXIT_FAILURE);
    }
    printf("cary: %p\n", cary);

    cary[0] = 10;

    reary = (int *)realloc(cary, 4 * sizeof(int));
    if (reary == NULL)
    {
        printf("메모리 할당이 문제가 있습니다.\n");
        exit(EXIT_FAILURE);
    }
    printf("reary: %p\n", reary);

    // 첫 원소는 10, 이후 2개는 기본 값인 0,
    // 마지막 하나는 쓰레기 값
    myprintf(reary, 4);
    free(reary);

    return 0;
}

void myprintf(int *ary, int n)
{
    for (int i = 0; i < n; i++)
        printf("ary[%d] = %d ", i, *(ary + i));
    printf("\n");
}
```

```
cary: 0x11d6060e0
reary: 0x11d6060e0
ary[0] = 10 ary[1] = 0 ary[2] = 0 ary[3] = -842150451
```

두 메모리의 시작 주소가 같음

# 예제: 10개의 문자열을 저장하는 동적 메모리

<05malloc_str.c>

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

#define MAX_LEN 20
int main()
{
    char *list[10] ={NULL};
    char msg[MAX_LEN] = {0};
    for (int i = 0; i < 10; i++)
    {
        list[i] = (char *)malloc(sizeof(char) * MAX_LEN);
        if (list[i] == NULL)
        {
            printf("malloc() fail\n");
            exit(1);
        }
        sprintf(msg, "Test string: %02d", i + 1);
        strcpy(list[i], msg);
    }

    for (int i = 0; i < 10; i++)
        printf("%s\n", list[i]);

    return 0;
}
```

**실행 결과**

```
Test string: 01
Test string: 02
Test string: 03
Test string: 04
Test string: 05
Test string: 06
Test string: 07
Test string: 08
Test string: 09
Test string: 10
```

# 자기 참조 구조체 정의

- **자기 참조 구조체(Self reference structure)**
  - 구조체 멤버로 자신과 동일한 타입의 구조체 포인터를 포함하는 구조체

- **struct selfref: 자기 참조 구조체**
  - 멤버로 int n과 struct selfref *next로 구성
    - 멤버 next의 자료형은 지금 정의하고 있는 구조체의 포인터 형
    - 구조체의 멤버 중의 하나가 자신의 구조체 포인터 변수
  - 구조체는 **자기 자신 포인터를 멤버로 사용**할 수 있으나
    - 자기 자신을 멤버로 사용 불가능

```
struct selfref {
    int n;
    struct selfref *next;

    //struct selfref one; // 컴파일 오류 발생
};
```

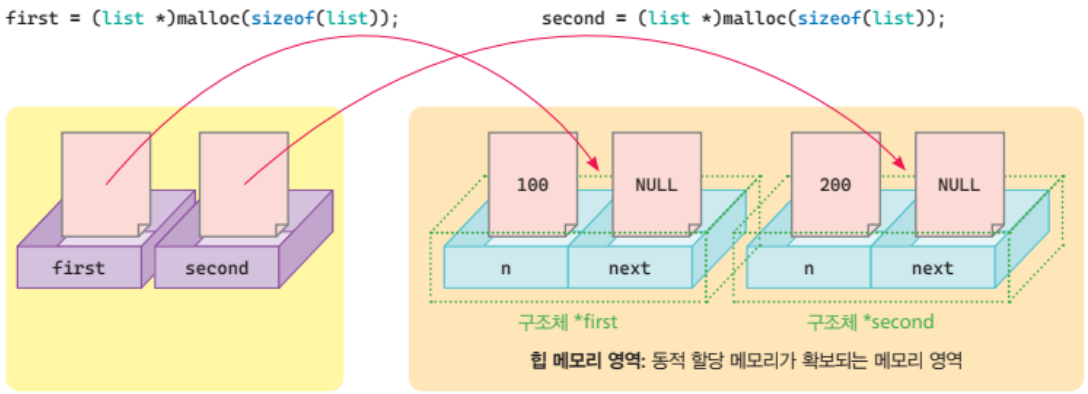
자기 참조 구조체

# 간단한 연결 리스트

- **연결 리스트(linked list)**
  - 자기 참조 구조체: 동일 구조체의 표현을 여러 개 만들어 연결할 수 있는 기능

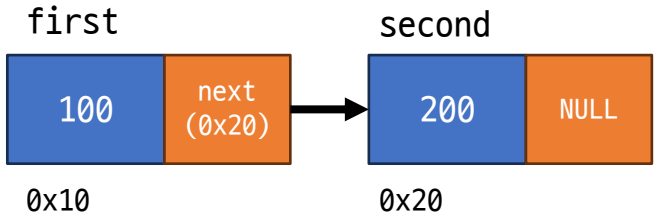
```
struct selfref { //자기참조 구조체 정의
    int n;
    struct selfref* next;
};
```

```
// 구조체 struct selfref를 list 형으로 정의
typedef struct selfref list; // 별칭 정의

// 저장 공간 할당
list *first = (list *)malloc(sizeof(list));
list *second = (list *)malloc(sizeof(list));

first->n = 100;
second->n = 200;

first->next = NULL;
second->next = NULL;
```

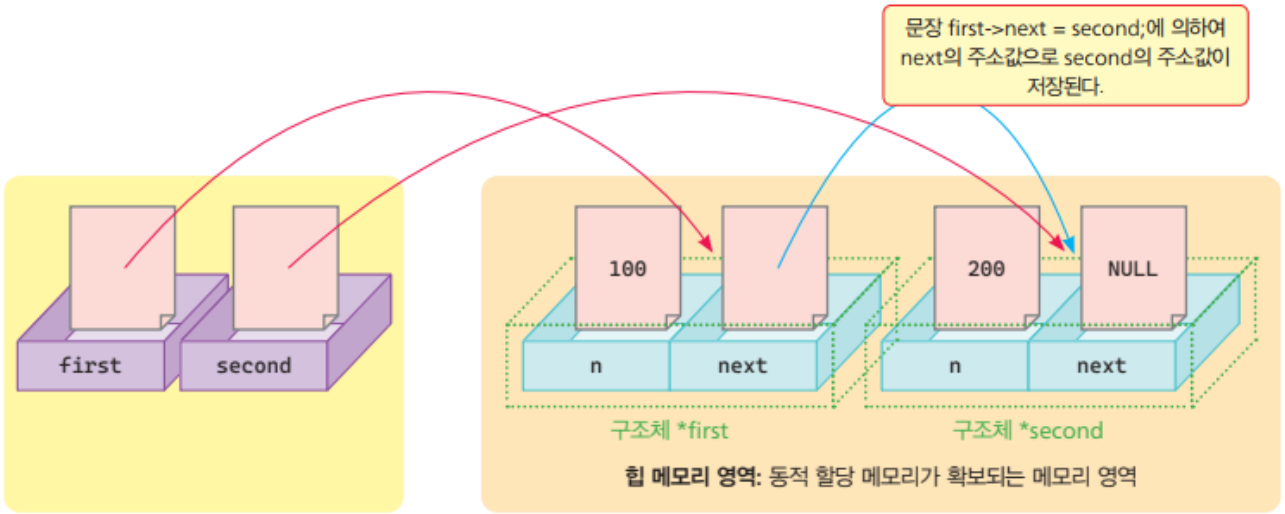



# 구조체 포인터의 활용

- **만일 구조체 *first가 다음 *second 구조체를 가리키도록 하려면**
  - 문장 `first->next = second;` 가 필요
    - ➢ 구조체 first의 멤버 next에 구조체 포인터 second의 주소를 저장
  - first가 가리키는 구조체 멤버 next를 사용하여 다음 구조체를 연결




```
//❹ first 다음에 second를 연결
first->next = second;  // 구조체 *first가 다음 *second 구조체를 가리키도록 하는 문장
```



# 실습 예제 16-5: 자기참조 구조체를 이용한 연결리스트의 기본 학습

<05selfrefstruct.c>

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

// 자기참조 구조체 정의
struct selfref
{
    int n;
    struct selfref *next;
};

int main(void)
{
    // ① 구조체 struct selfref를 자료형인 list 로 정의
    typedef struct selfref list;

    // ② 두 구조체 포인터 변수 first와 second를 선언
    // 함수 malloc()을 이용
    // 구조체의 멤버를 저장할 수 있는 저장공간을 할당
    list *first = NULL, *second = NULL;

    first = (list *)malloc(sizeof(list));
    second = (list *)malloc(sizeof(list));

    // ③ 구조체 포인터 first와 second의 멤버 n에
    // 각각 정수 100, 200을 저장하고, 멤버 next에는 NULL을 저장
    first->n = 100;
    second->n = 200;
    first->next = second->next = NULL;

    // ④ first 다음에 second를 연결
    // 구조체 *first가 다음 *second 구조체를 가리키도록 하는 문장
    first->next = second;

    printf("first 주소: %p\n", first);
    printf("first n: %d, next: %p\n", first->n, first->next);
    printf("first->next->n: %d\n\n", first->next->n);

    printf("second 주소: %p\n", second);
    printf("second n: %d, next: %p\n", second->n, second->next);

    free(first); // 동적메모리 할당 해제
    free(second);

    return 0;
}
```

second->n을 가리킴

```
first 주소: 0x154605c60
first n: 100, next: 0x154605e20
first->next->n: 200

second 주소: 0x154605e20
second n: 200, next: 0x0
```

second의 주소와 first->next주소가 동일

# 자기 참조 구조체 선언 방법

사용 가능

```
struct employee
{
    char *name;
    int salary;
    struct employee *next;
};
```

```
typedef struct Node
{
    int data;
    struct Node *next;
} Node;
```

반드시 struct Node 사용

```
typedef struct employee EMPLOYEE;

struct employee
{
    char *name;
    int salary;
    EMPLOYEE *next;
};

int main(void)
{
    EMPLOYEE *you = (EMPLOYEE *)malloc(sizeof(EMPLOYEE));
    EMPLOYEE *one = (EMPLOYEE *)malloc(sizeof(EMPLOYEE));
    .  .  .
}
```

선행 선언: 미리 EMPLOYEE 선언
- 구조체 내용은 아직 모름

포인터 변수는 사용 가능
- 포인터 크기만 알면 주소 연산이 가능하기 때문
- 항상 포인터 크기는 8바이트(64비트 컴퓨터)

컴파일 에러 발생

```
struct employee
{
    char *name;
    int salary;
    struct employee next;
} employee;
```

자기 참조용 변수 사용 안됨

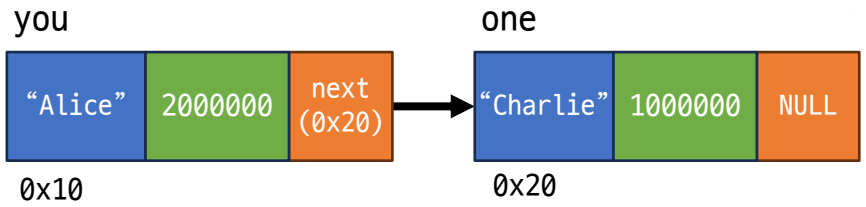
- 구조체 크기가 컴파일 시점에 확정됨
- 자기 자신을 포함하면 크기가 무한대가 됨

```
typedef struct employee
{
    char *name;
    int salary;
    employee *next;
} employee;
```

- employee가 정의되기 전에 사용
- typedef가 완전히 정의되기 전이라서 에러

# LAB 직원을 위한 자기참조 구조체 구현

- **직원의 정보를 표현하는 struct employee를 정의**
  - 2개의 구조체를 동적 메모리로 할당하여 적절한 정보를 입력
  - 구조체 employee는 자기 자신을 가리키는 자기 참조 구조체
    - ➢ 하나의 구조체 필드 next가 다른 구조체를 가리키도록 구현




<lab1semployee.c>

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

// 자기참조 구조체 정의
typedef struct employee {
    char* name;
    int salary;
    struct employee *next;
} employee;

int main(void)
{
    employee* you = (employee*)malloc(sizeof(employee));
    employee* one = (employee*)malloc(sizeof(employee));

    you->next = one->next = NULL;
    you->name = (char*)malloc(strlen("Alice") + 1);
    strcpy(you->name, "Alice");
    you->salary = 2000000;
    one->name = (char*)malloc(strlen("Charlie") + 1);
    strcpy(one->name, "Charlie");
    one->salary = 1000000;

    you->next = one;
    printf("%s %d\n", you->name, you->salary);
    printf("%s %d\n", one->name, one->salary);
    printf("%s %d\n", you->next->name, you->next->salary);

    //동적메모리 할당 해제
    free(one);
    free(you);

    return 0;
}
```

you의 next에 one의 주소 저장

```
Alice 2000000
Charlie 1000000
Charlie 1000000
```

# Questions?