10. Dynamic Memory Allocation

Hyunchan, Park

http://oslab.jbnu.ac.kr

Division of Computer Science and Engineering

Jeonbuk National University

학습 내용

- Memory Allocation
- Dynamic Memory Allocation
- Linked List: Basic

Memory Allocation

Volatile and Non-volatile storage devices

- Primary storage: Main memory
  - 주기억장치로 사용하는 DRAM 등의 휘발성 저장 장치
  - 성능이 높지만, 적은 저장 공간 제공
    - 프로그램 내의 변수와 같이 용량은 적지만 자주 접근하는 자료를 저장
- Secondary storage: Storage devices
  - 보조기억장치로 사용하는 HDD, SSD 등은 비휘발성 저장 장치
  - 느리지만, 많은 저장 공간을 제공
    - 시스템 종료 시에도 보관하여야 할 데이터를 적재하고, 시스템 재기동 시 다시 로드
  - 일반적으로 파일(file)의 형태로 데이터를 저장함

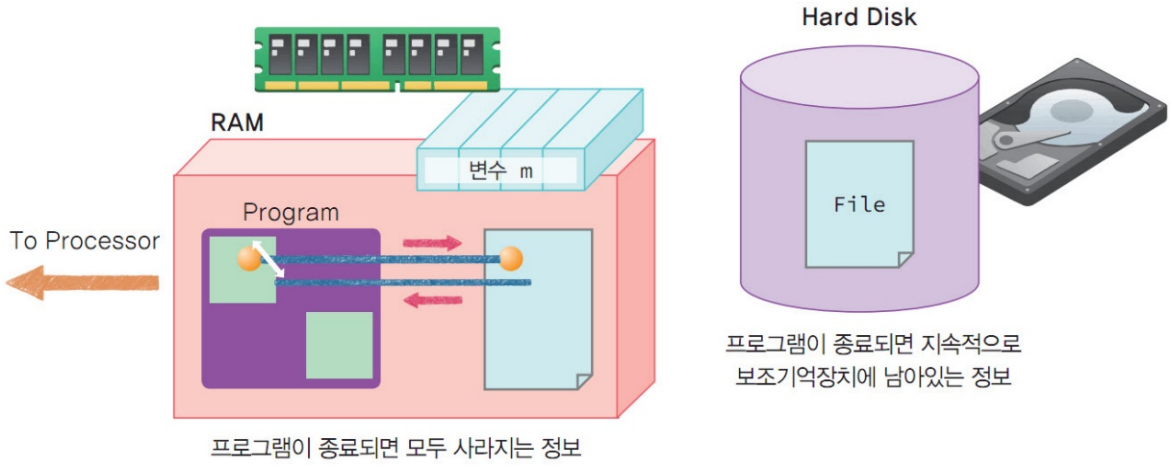


그림 15-2 주기억장치의 저장공간과 파일의 차이

Memory

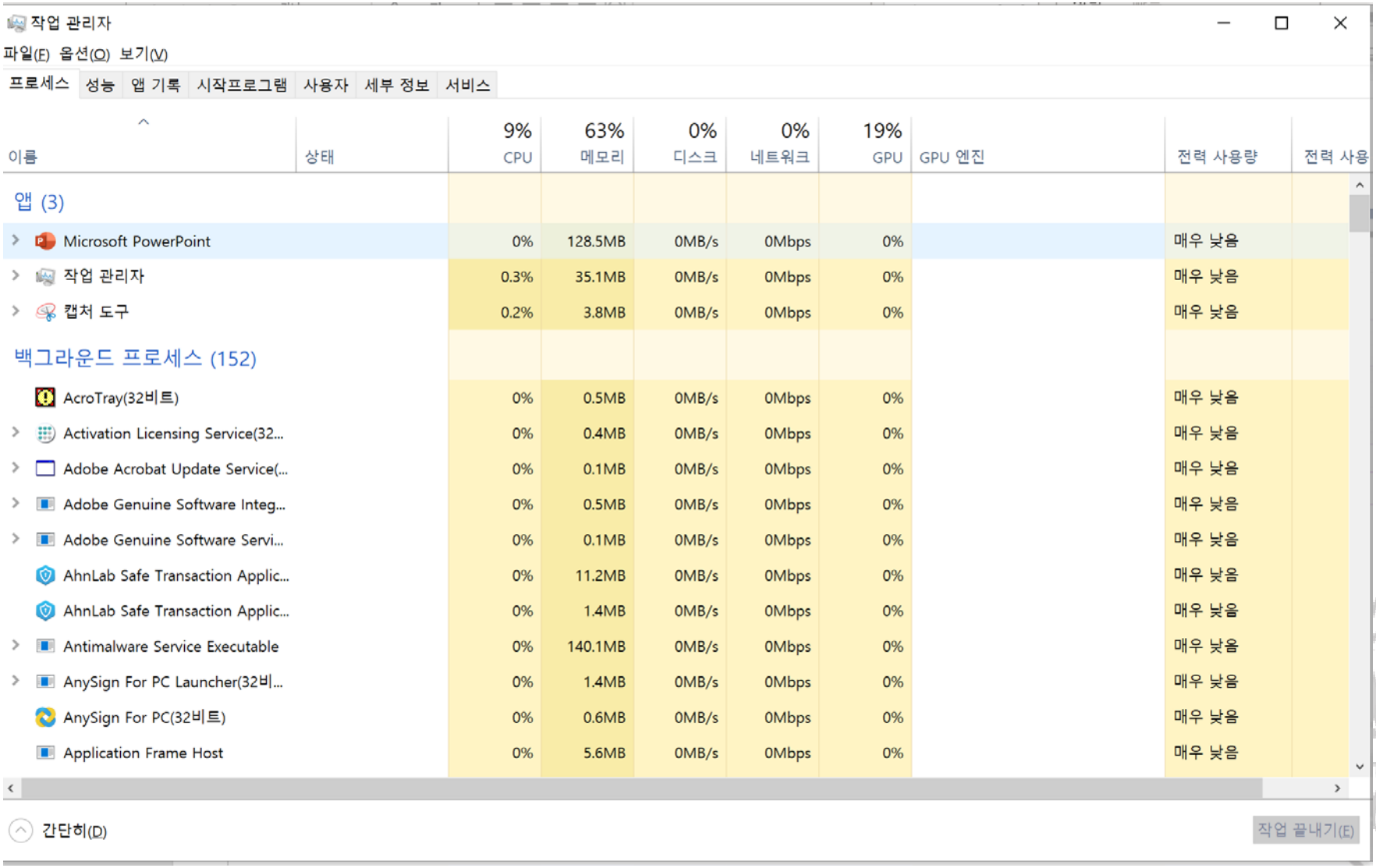
- 메모리는 한정된 자원 (예. 8GB, 16GB)
  - OS와 여러 프로세스가 동시에 물리 메모리 공간을 공유하며 실행됨
  - OS는 자기 자신과 여러 프로세스에 대해 최대한 효율적으로 메모리를 할당하고자 함
    - 이를 위해 OS는 가상 메모리 관리 기법을 사용
- 가상 메모리 공간 (Virtual Memory Space)
  - 각 프로세스는 자기 자신만의 독립되고, 고립된 (isolated) 메모리 공간을 가짐
    - 프로세스들은 서로 다른 사람의 space 를 건드릴 수 없음!
    - 공간의 크기는? 일반적으로 32 비트 주소 공간 (각 주소 마다 1B 저장: total 4GB = 2^32)
      - 64비트 프로세스의 경우, 48비트 혹은 56 비트만 사용 (256 TB = 2^48 or 64 PB = 2^56)

Memory Allocation

- OS의 메모리 관리
  - OS는 해당 "가상" 메모리 공간에 대해
    필요할 때만, 필요한 만큼만 실제 물리 메모리를 할당해 줌 (짠돌이)
  - 예) 호텔을 예약하는데, 일단 100개 객실이 있는 호텔을 통째로 다 빌려준다고 말함
    - 실제 객실은 100개일수도 있고, 200개 일수도 있고, 10개일수도 있음!
  - Private hotel~! You are our only guest!
    - 거짓말 이지만 진짜! 실제로 다른 손님과 절대 만나는 일이 없도록 관리해 줌
  - 딴 사람한테도 그렇게 예약해줌 (over-booking)
  - 실제로 손님이 왔을 때에, 실제 객실을 나눠 줌
  - 만약 실제 객실 수보다 많이 오면?
    - 객실 손님이 자고 있을 때, 슬쩍 방 전체를 아주 아주 넓은 창고로 옮김 (Secondary storage)
    - 해당 객실에 새 손님을 받음
  - 창고 용량도 넘어가면??? OOM!! (Out-Of-Memory)
    - 호텔 문 닫고 다 내쫓음 (프로세스 강제 종료)
- 따라서 실제로 메모리 공간을 사용할 때는, OS에게 메모리 할당을 요청해야 함

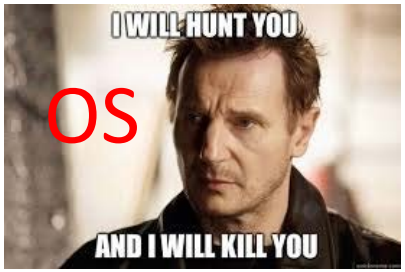
Memory Allocation

Memory Allocation



Memory Allocation

- OS는 두 가지 방식으로 메모리 할당을 수행함: Static and Dynamic
- Static (정적 할당)
  - 프로그램이 수행되어 새로운 프로세스를 생성하는 과정에서 메모리를 할당하고, 해당 프로세스가 종료되기 이전까지는 할당이 해제되거나 내용이 변경되지 않음
  - 사용할 메모리 공간의 크기를 정하는 주체: 컴파일러
    - 프로그램이 실행되기 이전에 컴파일러에 의해 변수의 저장 공간 크기가 정해짐
  - 대상 자료 형태: 변수, 배열, 구조체로 선언된 자료들
    - 예) int i; 로 선언하고 나면, 실제 수행 중에 그 크기와 저장 위치를 변경하는 것은 불가능
  - 단점: 실행 이전에 사용할 메모리의 공간 크기가 정해진다는 건?
    - 예) struct friend list[10]; 카톡에서 친구 10명에 관한 구조체 데이터를 저장하기 위한 배열
    - 만약 10명 이상이면? → 소스 코드를 고치고, 새로 컴파일하고, 새로 수행해야 함
    - 넉넉하게 한 100만명 잡으면? → 실제 사용량에 비해 너무 많은 메모리를 할당해 비효율적
    - 메모리 사용량 예측이 부정확한 경우, 정보 저장에 실패하거나 메모리를 낭비하게 됨



Memory Allocation

- 필요할 때, 필요한 만큼 메모리 공간을 할당하고, 필요없을 때는 해제하고 싶다!
  - OS and neighbors: “Good!!”
- Dynamic (동적 할당)
  - 프로세스의 실행 중에 필요한 메모리를 할당하는 방법
    - CS에서 Dynamic 이란 용어는 “프로세스의 실행 중” 으로 해석하면 됨
  - 메모리 사용 예측이 정확하지 않고 실행 중에 메모리 할당이 필요할 때 사용
  - 예) 카톡 친구가 한 명 추가될 때 마다,
    - 해당 친구의 정보를 저장하기 위해,
    - 메모리 공간을 필요한 만큼만 추가로 할당받아 저장한다.
    - 그리고 친구 삭제하면 해당 공간을 할당 해제 (deallocation or free) 하여 OS에게 되돌려준다.
  - 단점: 사용이 (아주 약간) 불편함
    - 메모리를 매번 명시적으로 할당/해제 해야 함
      - 필요한 메모리 양을 계산하고, 시스템콜을 사용하여 OS에 요청
    - 포인터의 사용이 필요함
      - 새로 할당받은 메모리 공간을 지칭하기 위함
      - 포인터가 없는 언어에서도 동적 할당은 필수적이며, 다양한 형태로 지원 (예. Java의 ArrayList)

Memory Allocation

정적 메모리 할당 방식

```
int i;
long prod = 1;
int facto[6];
char *[] = {"algol", "pascal" "C",
      "C++", "Java","C#"};
```

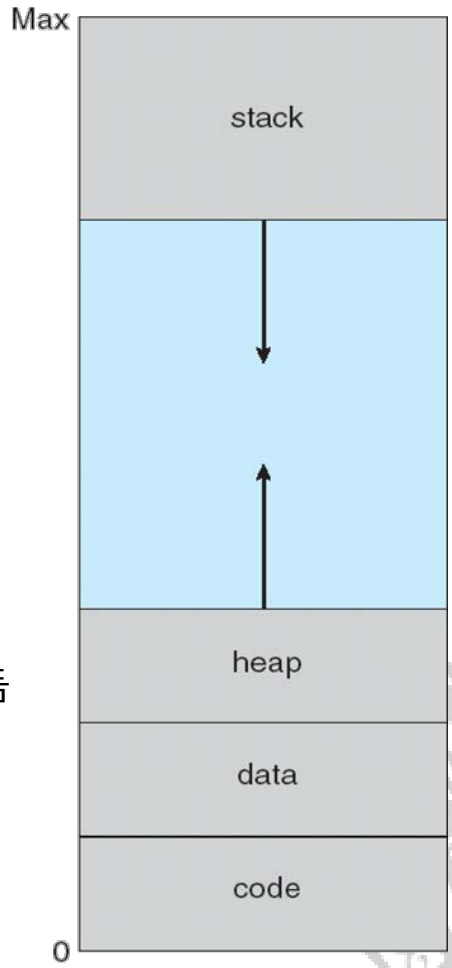
동적 메모리 할당 방식

```
int *pi = NULL;
//메모리 할당 함수 malloc()으로 동적메모리 할당
pi = (int *)malloc(sizeof(int));
//동적메모리 할당 성공 검사
if (pi == NULL) {
   printf("메모리 할당에 문제가 있습니다.");
   exit(1);
};
//내용 값 저장
*pi = 3;
```

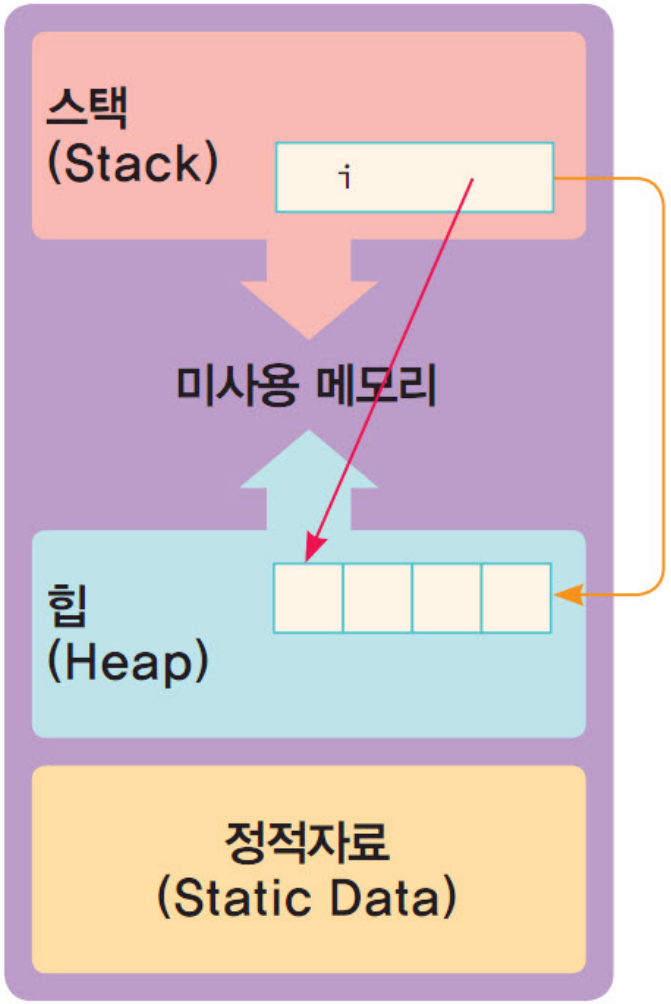
그림 16-1 정적 메모리 할당 방식과 동적 메모리 할당 방식

(참고) 메모리 할당 영역

- Code (text)
  - 프로그램 코드가 복제되어 실행에 사용
- Data
  - Global and static local variables
- Heap
  - 동적으로 할당받은 메모리가 위치함
  - 동적 메모리 할당의 요청/해제에 따라, 늘어나거나 줄어듦
- Stack
  - 함수 호출에 따라 동적으로 변경되는 부분
  - Function call 에 따라 스택이 쌓이면서 늘어나고, return 에 의해 다시 줄어듦
  - 지역 변수 (local variable), 함수 호출에 따른 인자 등이 저장됨
- Code and data: 프로세스가 실행될 때, 크기가 정해지고 변하지 않음
- Heap and stack: 프로세스의 수행에 따라 계속 크기가 변경됨



(참고) 메모리 할당 영역



포인터 변수 i가 지역 변수라면 i는 스택에 할당되고 maclloc()에 의한 메모리 공간은 힙 영역에 할당된 후, malloc()의 반환값인 주소값이 i에 저장

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

그림 16-3 메모리 영역

Dynamic Memory Allocation

동적 메모리 관련 함수

- 동적 메모리
  - 함수 malloc()의 호출로 힙(heap) 영역에 확보
  - 메모리는 사용 후 함수 free()를 사용해 해제

  - 만일 메모리 해제를 하지 않으면,
    메모리 부족과 같은 문제를 일으킬 수 있으니,
    꼭 해제하는 습관을 가질 것
  - (...나한텐 피해가는 거 없는데?)
    - ??: 이거 누가 짰어? 나가!

동적 메모리 관련 함수

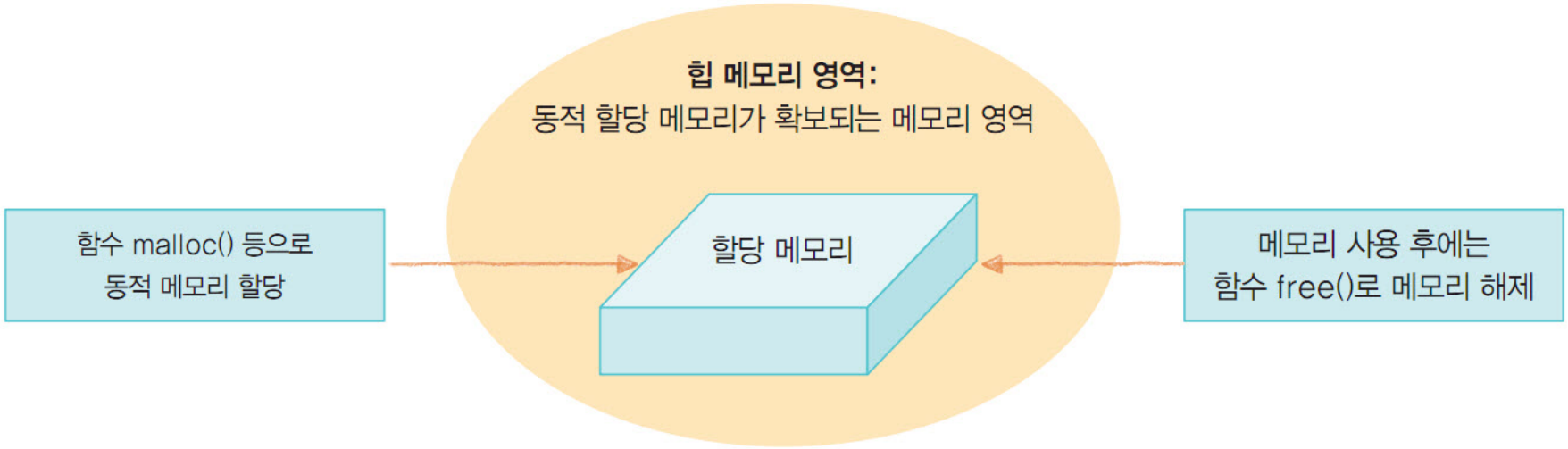


그림 16-2 동적 메모리 할당과 해제

동적 메모리 관련 함수

- 동적 메모리 할당 함수: malloc(), calloc(), realloc() 3가지
  - Return type: void *
    - Void 형: 특정한 형태가 없음을 뜻함
    - 메모리에 적재할 자료의 포인터 형으로 변환(casting)해서 사용
      - 예) int *data = (int *) malloc(sizeof(int));
  - 헤더파일 stdlib.h 필요
- 동적으로 할당된 메모리를 해제하여 반환
  - 함수 free()

메모리	함수 원형	기능
메모리 할당 (기본값 없이)	`void * malloc(size_t)`	인자만큼의 메모리 할당 후 기본 주소 반환
메모리 할당 (기본값 0으로)	`void * calloc(size_t , size_t)`	뒤 인자 만큼의 메모리 크기로 앞 인자 수 만큼 할당 후 기본 주소 반환
기존 메모리 변경 (이전값 그대로)	`void * realloc(void *, size_t)`	앞 인자의 메모리를 뒤 인자 크기로 변경 후, 기본 주소 반환
메모리 해제	`void free(void *)`	인자를 기본 주소로 갖는 메모리 해제

메모리 할당: malloc()

자료형 size_t는 자료형의 크기를 의미하며, unsigned int 형이다.

함수 malloc() 함수원형

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

함수 malloc()은 인자인 자료형 크기 size만큼의 메모리를 할당하여 성공하면 할당된 공간의 void 포인터를 반환하며, 실패하면 NULL을 반환

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

반환값은 이 값을 받는 자료유형의 포인터로 변환하여 포인터 변수에 저장된다.

함수 malloc()의 인자는 할당할 변수의 크기를 sizeof 연산자를 이용하여 지정한다.

- 할당 이후, 간접연산자 *pi를 이용하여 원하는 값을 수정 가능
  - 이때 malloc()으로 할당받은 메모리 공간에 적재된 값이 변경되는 것
- pi를 다른 메모리 공간의 주소로 수정 가능
  - 기존 메모리 공간은?
  - 해당 주소를 알아야 free() 를 할 수 있으므로, 이렇게 유실되는 경우가 없어야 함

할당받은 메모리 공간과 포인터의 이해

* 이 그림을 잘 이해해야 함

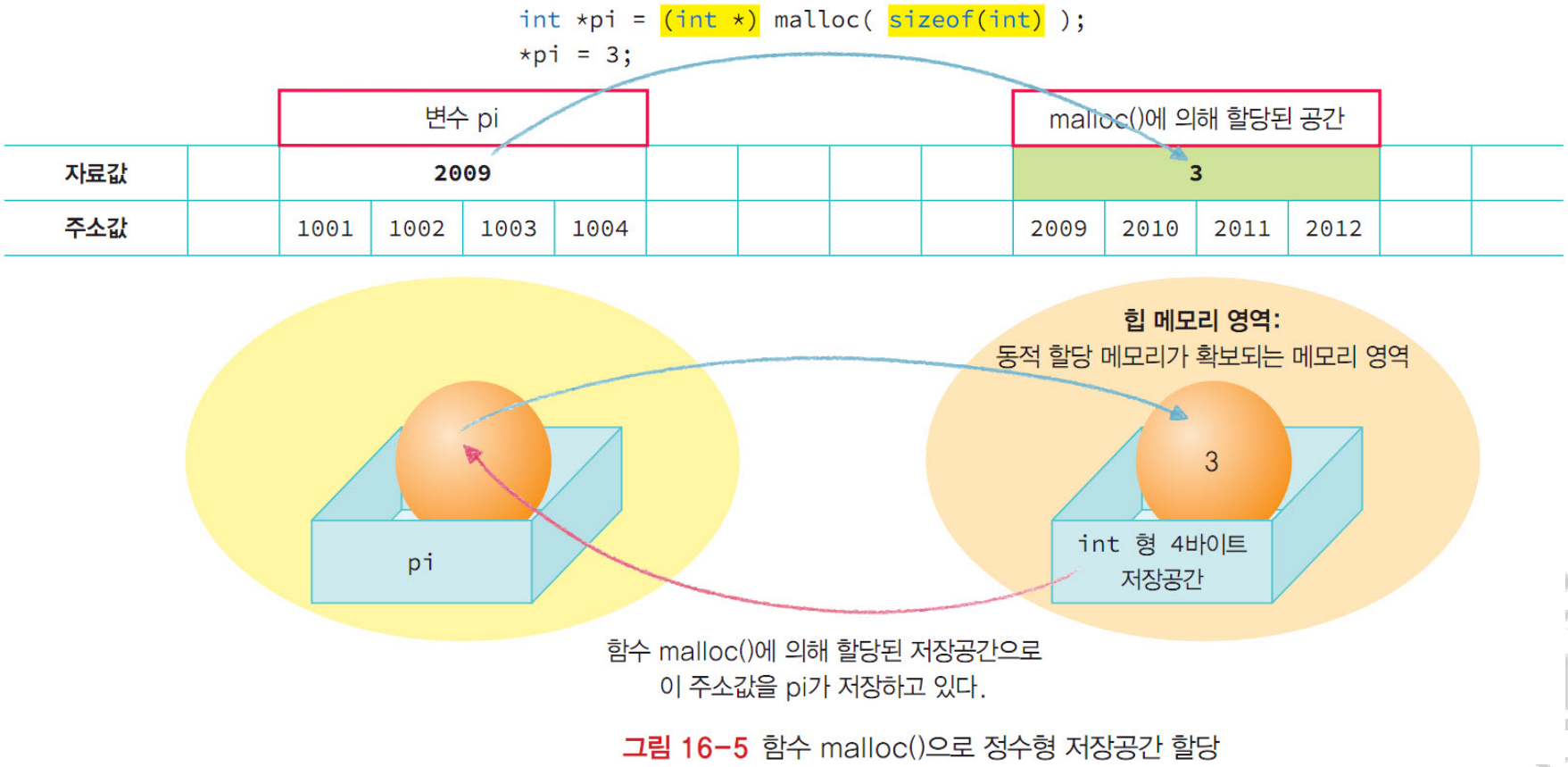


그림 16-5 함수 malloc()으로 정수형 저장공간 할당

메모리 해제: free()

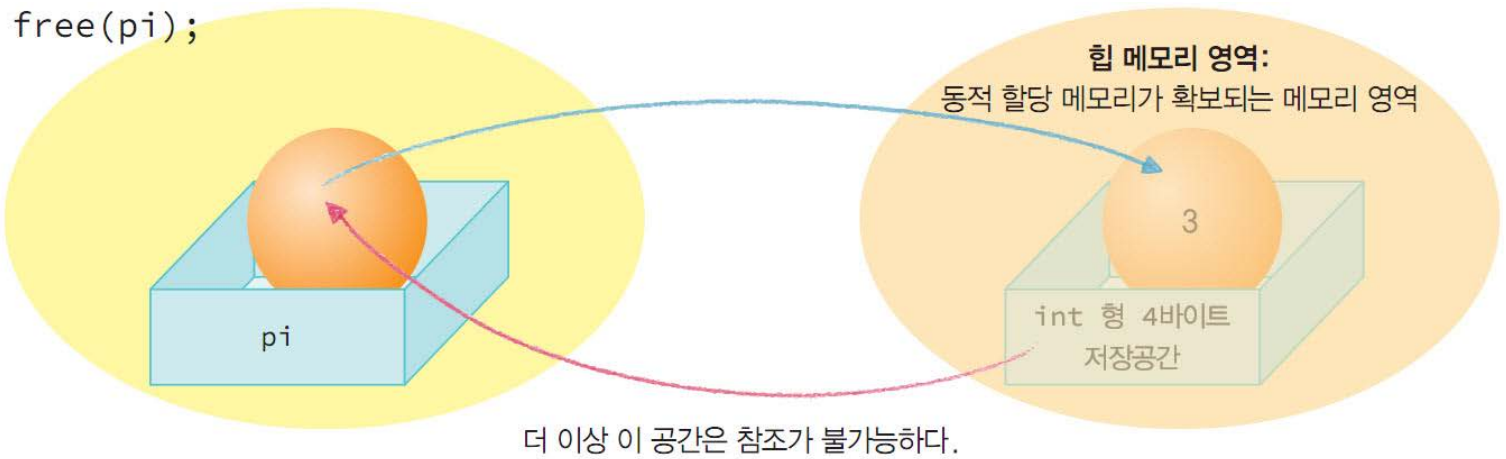
함수free() 함수원형

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

동적으로 할당된 메모리를 제거한다.

```
free(pi);
```

- free(pi)
  - 함수 malloc()의 반환 주소를 저장한 변수 pi를 해제
    - 인자로 해제할 메모리 공간의 주소값을 갖는 포인터를 이용하여 호출
  - 변수 pi가 가리키는 4바이트의 자료값이 해제되어 더 이상 사용할 수 없음



[예제 1] malloc() and free()

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

typedef struct {
    char id[10];
    int grade1, grade2, grade3;
    float avg;
} student;

int main(int argc, char* argv[]) {
    FILE *fp;
    student *data;

    if(argc != 2) {
            printf("< Usage: ./%s filename >\n", __FILE__);
            return 1;
    }

    if ((fp = fopen(argv[1], "r")) == NULL) {
        perror("Open");
        exit(1);
    }

    data = (student*) malloc(sizeof(student));

    while(fread(data, sizeof(student), 1, fp) == 1) {
                fprintf(stdout, "%s %d %d %d %.2f\n", data->id, data->grade1, data->grade2, data->grade3, data->avg);
    }

    free(data);

    fclose(fp);

    return 0;
}
```

```
ubuntu@41983:~/hw9$ ./m1 unix.bin
123 70 80 90 80.00
456 80 80 80 80.00
789 90 90 90 90.00
ubuntu@41983:~/hw9$ ./m1
< Usage: ./m1.c filename >
```

- 소스 파일 이름을 출력.
  계속 바꾸기가 신경쓰여서..
- 사실 뒤에 ".c"는 빼주는 처리를 해야하지만...

(참고) Useful C macros for debug messages

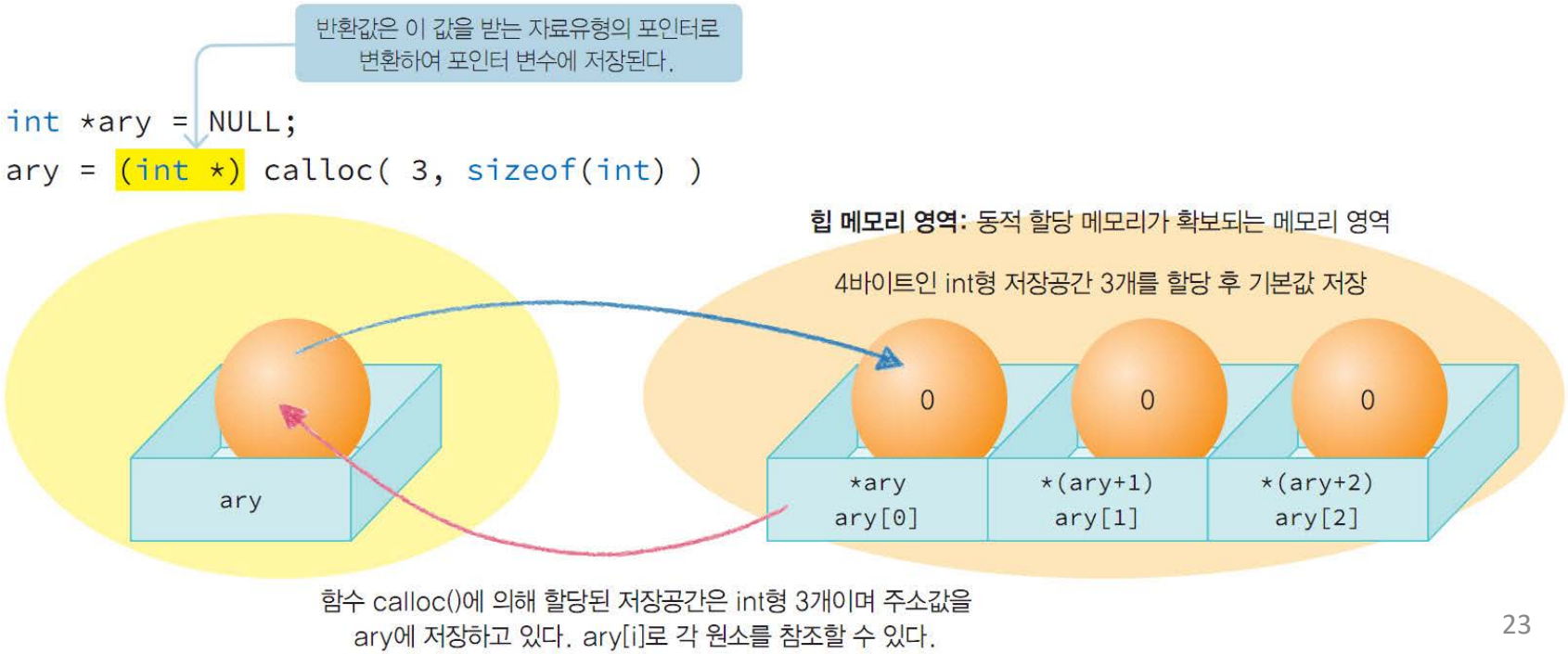
- __FILE__
  - 소스 파일 명을 출력
  - 컴파일러에 전달된 파일 이름에 따라, 절대 경로가 출력될 수 있음
- __LINE__
  - 현재 라인 번호를 출력
- __func__
  - 함수 이름을 출력
  - __FUNCTION__ 이라는 동일한 기능의 매크로도 있으나, C 표준이 아니고, 몇몇 컴파일러에서 지원 (__func__는 C99)

```c
#include <stdio.h>

int test(void) {
        printf("%s %3d %s\n", __FILE__, __LINE__, __func__);
        return 1;
}

int main(void) {
        return test();
}
```

```
ubuntu@41983:~/hw9$ vi hw9.c
ubuntu@41983:~/hw9$ gcc -o hw9 hw9.c
ubuntu@41983:~/hw9$ ./hw9
hw9.c   4 test
```

메모리 할당: calloc()

- 할당받은 공간을 0으로 초기화해줌
  - 기존 공간에 저장된 쓰레기 값으로 인한 예측하지 못한 문제를 예방
- 인터페이스의 변경
  - 마치 고수준 I/O 의 fread()/fwrite() 처럼,
  - (자료의 개수, 자료 크기) 로 구성되어, 조금 더 편리한 인터페이스 제공
  - 예) malloc() 에서는 그냥 3 * sizeof(int) 로 전달



메모리 할당: realloc()

- 이미 확보한 저장공간을 새로운 크기로 변경
  - 함수 realloc()에 의하여 다시 확보하는 영역
    - 기존의 영역을 이용하여 그 저장 공간을 변경하는 것이 원칙
      - 새로운 영역을 다시 할당하여 이전의 값을 복사할 수도 있음
    - 성공적으로 메모리를 할당하면 변경된 저장공간의 시작 주소를 반환
      - 실패하면 NULL을 반환
    - 인자
      - 첫 인자: 변경할 저장공간의 주소
        - NULL 을 주면, 그냥 malloc()과 동일하게 동작
      - 두 번째 인자: 변경하고 싶은 저장공간의 총 크기

함수 realloc() 함수원형

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

할당되는 총 메모리 크기이다.

- 함수 realloc()은 이미 확보한 메모리 p를 다시 지정한 크기 size로 변경하는 함수이며, 이미 확보한 p가 NULL이면 malloc()과 같은 기능을 수행

메모리 할당: realloc()

```
int *reary, *cary;
cary = (int *) calloc( 3, sizeof(int) );
reary = (int *) realloc( cary, 4*sizeof(int) );
```

이전에 calloc(), malloc(), realloc()에 의하여 이미 할당된 저장공간의 기본 주소이다.

새로이 할당된 저장공간의 기본 주소가 저장된다.

이미 있는 공간과 새로이 확보될 저장공간의 합인 전체 크기이다.

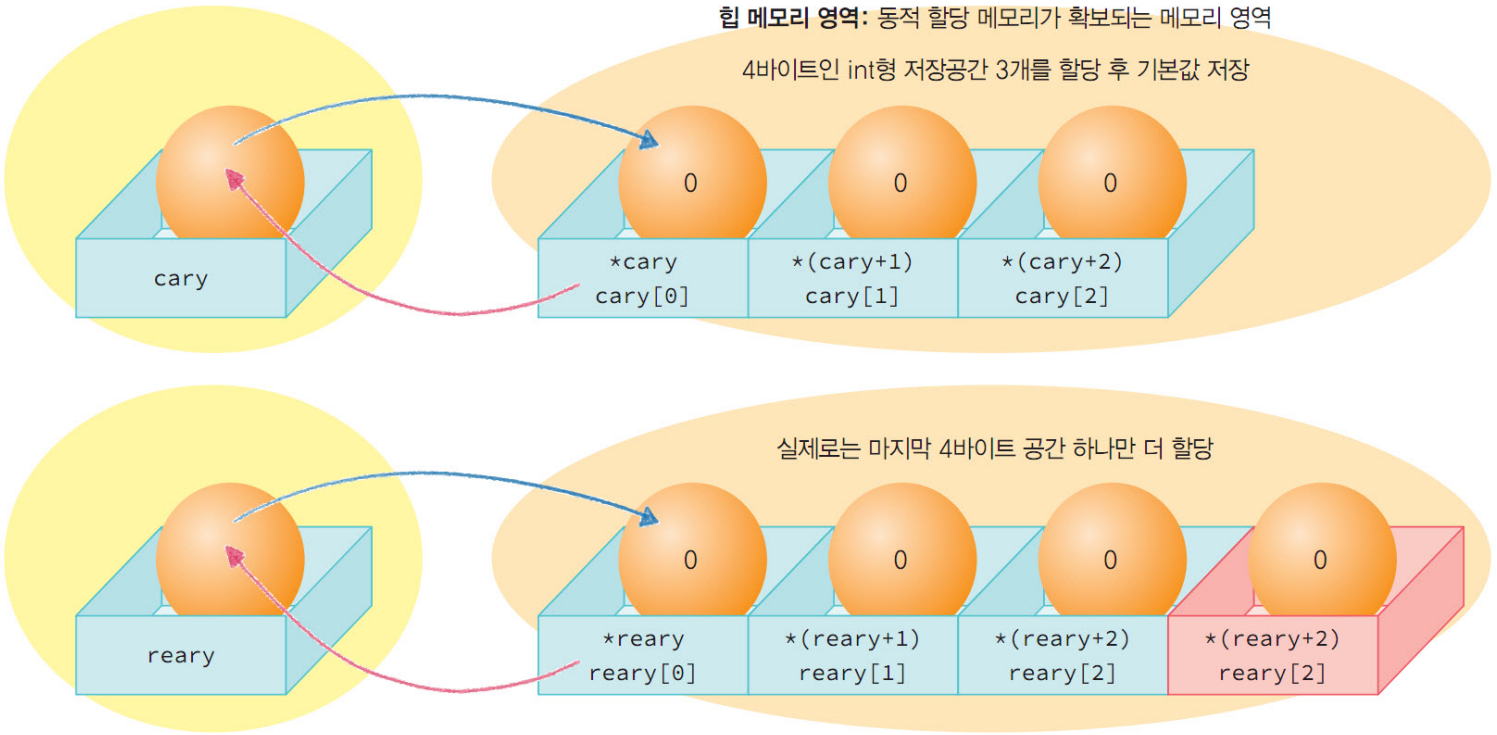


그림 16-12 함수 realloc()에 의한 메모리 공간의 재할당

Linked List: Basic

연결 리스트

- 연결 리스트
  - 순차적 자료 표현에 적합한 구조
  - 동적으로 항목이 추가되고, 항목 간의 순서가 변경되는 데이터의 관리에 적합
- 배열과의 비교
  - 컴파일 시 배열의 크기가 이미 결정되어, 실행 중간에 배열 크기 수정이 불가능
  - 순서 변경의 어려움
    - 맨 앞이나 중간에 새로운 항목이 삽입되면?
      - 삽입되는 항목 이후의 이미 저장된 항목들을 모두 뒤로 이동?
      - 많은 양의 데이터 복사로 수행 속도 저하
    - 중간에 하나 삭제하는 경우도 마찬가지
- (왜 갑자기 유닉스 수업에서?)
  - 동적 메모리를 활용한 과제를 수행하려면 필수..
  - 자세한 내용은 자료 구조 혹은 알고리즘 수업에서 좀 더 공부하자!

배열의 단점 예제

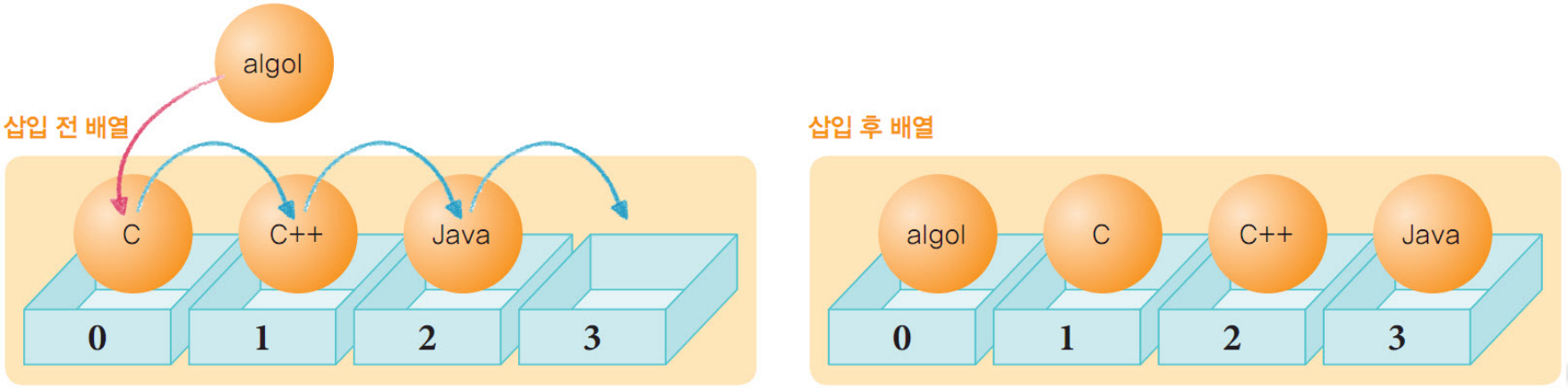


그림 16-17 배열 처음 위치에 새로운 원소 삽입

연결 리스트 구조

- 연결 리스트 기본 구조
  - 헤드에서 시작하여 가리키는 곳을 계속 따라가면 순차적 자료를 표현
- 연결 리스트 예
  - 헤드(head)는 "미수"를 가리키고
    - "미수"는 다시 "현순"을 가리키고
    - 계속해서 "윤원", "현화", "수성", "나혜"
    - 그리고 다시 나혜는 마지막이라 가리키는 사람이 없는 것(NULL)과 같은 구조

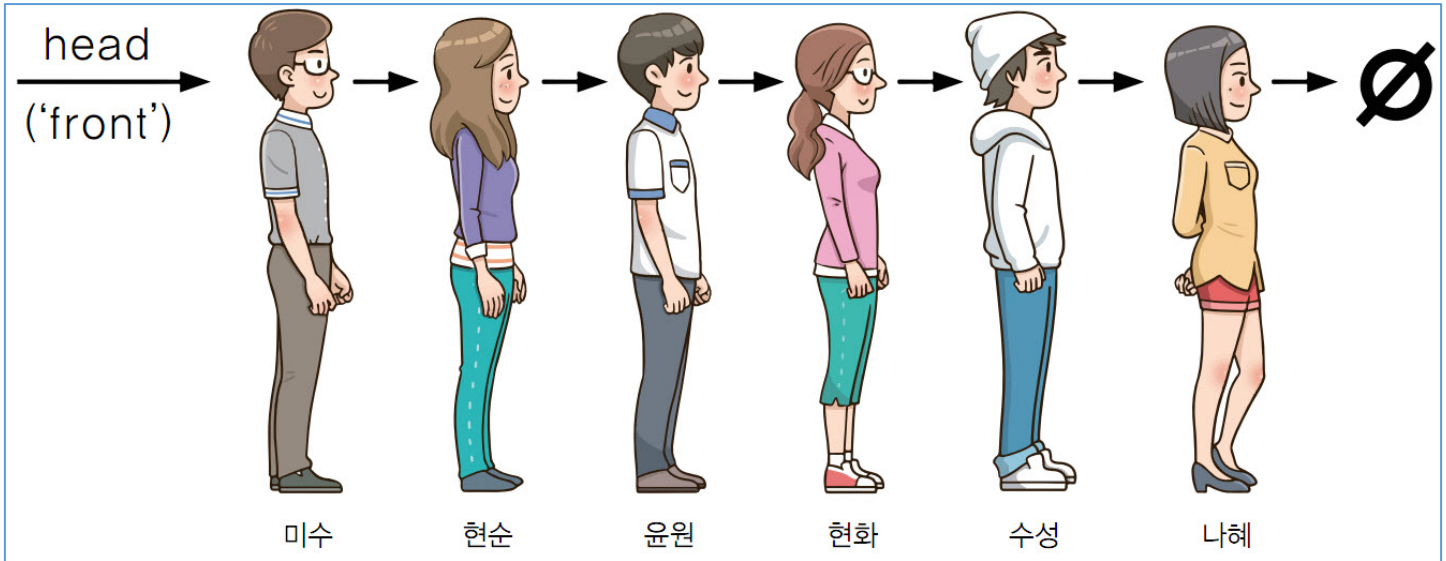


그림 16-18 연결 리스트의 이해

연결 리스트 구조: 노드

- 연결 리스트 내의 각 항목은 “Node” 라는 형태로 구성
- 노드의 자료: 필요한 여러 변수의 조합으로 구성
  - 노드 간의 링크: 자기 참조 구조체의 포인터로 구현
- Head : 항상 첫 번째 노드를 가리키는 포인터
- Tail : 마지막 노드를 가리키는 포인터

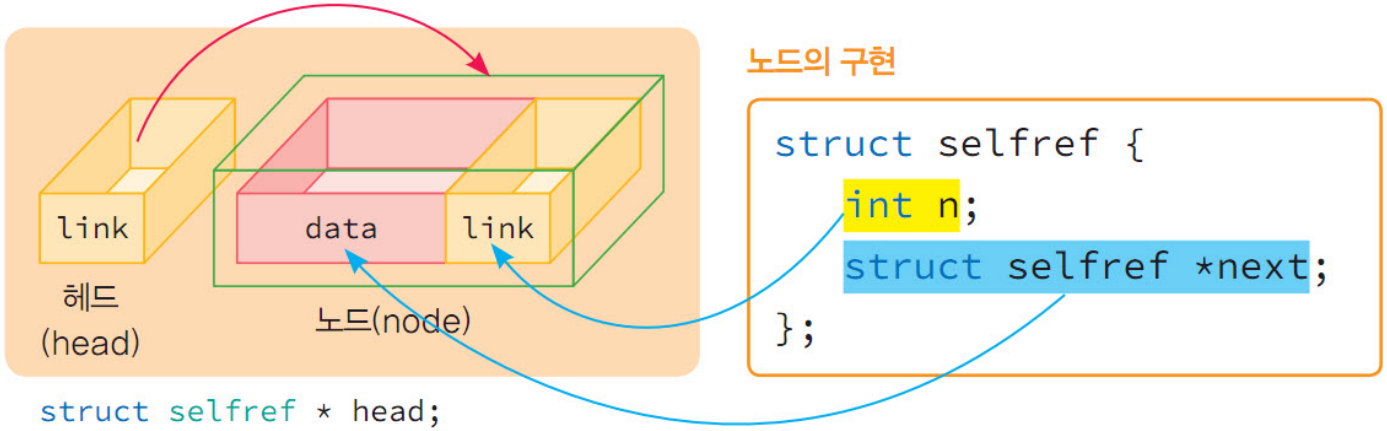


그림 16-19 연결 리스트의 헤드와 노드

연결 리스트 구조: 자기 참조 구조체

- 자기참조 구조체(self reference struct)
  - 구조체의 멤버 중의 하나가 자기 자신의 구조체 포인터 변수를 갖는 구조체

```
struct selfref {
    int n;
    struct selfref *next;
    //struct selfref one;    //컴파일 오류 발생
}
```

error C2079: 'one'은(는) 정의되지 않은 struct 'selfref'을(를) 사용합니다.

- 구조체 selfref
  - 멤버로 int 형 n과 struct selfref * 형 next로 구성
    - 즉, 멤버 next의 자료형은 지금 정의하고 있는 구조체의 포인터 형
  - 구조체 selfref는 자기 참조 구조체
    - 구조체의 멤버 중의 하나가 자기 자신의 구조체 포인터 변수
  - 구조체는 자기 자신 포인터를 멤버로 사용할 수 있으나
    - 자기 자신은 멤버로 사용 불가능: 재귀적 참조로 인해 크기를 결정할 수 없음

연결 리스트 구조

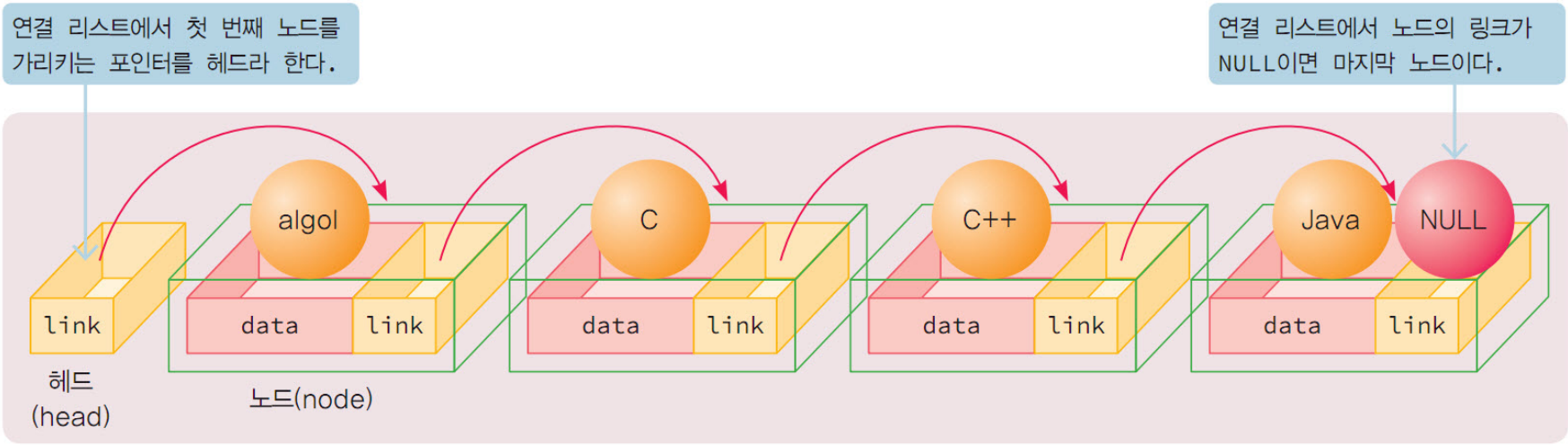


그림 16-20 연결 리스트의 헤드와 노드

연결 리스트의 장단점

- 연결 리스트 장점
  - 항목 수를 프로그램 내부에서 메모리가 허용하는 한 늘릴 수 있다는 것
    - 배열과는 달리 프로그램 실행 전에 미리 기억장소를 확보해 둘 필요가 없음
  - 프로그램 실행 중이라도 필요할 때 노드를 동적으로 생성
    - 기존의 연결 리스트에 삽입 또는 추가 가능
  - 항목 들이 메모리 공간에 연속적으로 저장될 필요가 없음
    - 중간에 노드를 삽입 또는 삭제하더라도 배열에 비하여 다른 노드에 영향을 적게 미침
  - 결론적으로 연결 리스트는 동적으로 노드를 생성하고 관리함으로써,
    - 리스트 크기의 증가 감소에 따라 효율적으로 대처할 수 있으며
    - 노드의 삽입과 삭제와 같은 자료의 재배치를 빠르게 처리
- 단점: random access
  - 배열에 비하여 임의 접근(random access)에 많은 시간이 소용
  - 노드 검색은 헤드에서부터 링크를 따라가는 순차적 검색만이 가능

[예제 2] 연결 리스트 사용 (v1. 오류가 있음)

hw9 > C m2.c

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

typedef struct student_t {
    char id[10];
    int grade1, grade2, grade3;
    float avg;
    struct student_t *next;
} student;

int main(int argc, char* argv[]) {
    FILE *fp;
    student *head, *cur, *data;

    if(argc != 2) {
        printf("< Usage: ./%s filename >\n", __FILE__);
        return 1;
    }

    if ((fp = fopen(argv[1], "r")) == NULL) {
        perror("Open");
        exit(1);
    }

    data = (student*) malloc(sizeof(student));
    head = data;    //헤드를 처음 생성한 student object 로 지정
    data->next = NULL;  //다음 항목은 아직 없으므로, NULL 로 초기화

    printf("sizeof student= %ld %ld\n", sizeof(student), sizeof(student*));

    while(fread(data, 28, 1, fp) == 1) {    //sizeof(student) 가 변경됨에 따라, 기존 파일 저장 형태와 달라짐.
        fprintf(stdout, "%s %d %d %d %.2f\n", data->id, data->grade1, data->grade2, data->grade3, data->avg);

        data->next = (student*) malloc(sizeof(student)); //다음 정보를 저장할 노드 생성하고, 이전 노드가 새로운 노드를 가리키게 함
        data = data->next; //data 를 이용해 새로운 데이터를 읽어야 하므로, data 포인터를 새 노드를 가리키도록 업데이트
        data->next = NULL; //다음 항목은 아직 없으므로, NULL 로 초기화
    }

    free(data); //잘못된 코드: 최종 생성된 노드 하나만 할당 해제함. 연결 리스트에 등록된 모든 노드를 해제해야 함

    fclose(fp);

    return 0;
}
```

```
ubuntu@41983:~/hw9$ ./m2 unix.bin
sizeof student= 40 8
123 70 80 90 80.00
456 80 80 80 80.00
789 90 90 90 90.00
```

[예제 2] 연결 리스트 사용 (v2. 오류 해결)

```
25      //Load data from file stream to the linked list
26      cur = (student*) malloc(sizeof(student));
27      head = cur;     //헤드를 처음 생성한 student object 로 지정
28      prev = cur;     //맨 마지막에 불필요하게 할당된 공간을 해제하고, 이전 노드의 link를 Null 로 업데이트 하기 위함
29      cur->next = NULL;  //다음 항목은 아직 없으므로, NULL 로 초기화
30
31      printf("sizeof student= %ld %ld\n", sizeof(student), sizeof(student*));
32
33      while(fread(cur, 28, 1, fp) == 1) {    //sizeof(student) 가 변경됨에 따라, 기존 파일 저장 형태와 달라짐.
34          //fprintf(stdout, "%s %d %d %d %.2f\n", cur->id, cur->grade1, cur->grade2, cur->grade3, cur->avg);
35
36          cur->next = (student*) malloc(sizeof(student)); //다음 정보를 저장할 노드 생성하고, 이전 노드가 새로운 노드를 가리키게 함
37          prev = cur;     //맨 마지막에 불필요하게 할당된 공간을 해제하고, 이전 노드의 link를 Null 로 업데이트 하기 위함
38          cur = cur->next; //cur 를 이용해 새로운 데이터를 읽어야 하므로, cur 포인터를 새 노드를 가리키도록 업데이트
39          cur->next = NULL; //다음 항목은 아직 없으므로, NULL 로 초기화
40      }
41      free(cur);      //마지막에 할당된 공간은 불필요하므로 해제
42      prev->next = NULL;  //마지막에 할당된 공간은 해제하였으므로 이전 노드의 next 필드를 NULL로 업데이트
43
44      printf("%p %p\n", cur, head);
45      if(head == cur) head = NULL;  //만약 cur 와 head 가 같다면 어떤 데이터도 로드하지 못한 상태. head 도 NULL 로 지정
```

노드 순회(node traversal)

- 노드 순회(node traversal)
  - 연결 리스트에서 모든 노드를 순서대로 참조하는 방법
  - 헤드부터 계속 노드 링크의 포인터로 이동하면 가능
    - 링크가 NULL이면 마지막 노드
    - 노드 순회 방법을 이용하여 각 노드의 자료를 참조할 수 있음

[예제 3] 연결 리스트 노드 순회

- 예제 2 뒤쪽에 순회하며 내용을 출력하는 코드 추가

```
13      student *head, *cur, *data, *prev;
```

```
46      // Node traversal
47      cur = head;
48
49      while(cur != NULL) {
50          fprintf(stdout, "%s %d %d %d %.2f\n", cur->id, cur->grade1, cur->grade2, cur->grade3, cur->avg);
51          prev = cur; //이전 노드의 주소를 복제
52          cur = cur->next;    //cur 를 다음 노드를 가리키도록 업데이트
53          free(prev); //이전 노드 할당 해제
54      }
55
56      fclose(fp);
57
58      return 0;
59  }
```

개인 과제 9: 파일+동적메모리할당

- 내용: 프로그램 2개 작성
  - 1. generator.c
    - 오른쪽 자료를 참고하여,
    - 임의의 개수만큼, (N ≤ 1000)
    - 임의의 학생 데이터 생성하여,
    - 이진 파일로 저장
    - 난수 생성을 위해 rand() 사용하고, random seed 를 적절하게 적용해 사용할 것

```
char* names[10] = { "Alice", "Bob", "Chris", "Dod", "Evan",
                    "Fint", "Gregg", "Brendan", "Roy", "Susan" };

struct pscore
{
    int num;        //학번
    char nme[10];   //이름 (위 이름 리스트에서 임의로 copy)
    float sc1;      //점수1
    float sc2;      //점수2
    float sum;      //점수 합계
};
```

  - 2. loader.c
    - Generator 가 생성한 파일을 읽어들여, 저장된 데이터를 로드함
    - 이때, 동적으로 메모리를 할당하여, 연결 리스트에 필요한 만큼 저장
    - 모든 데이터를 읽은 다음, 사용자로부터 1000 이하의 양의 정수 하나를 입력받아,
    - 해당 순서에 위치한 데이터를 출력함
      - 가장 앞에 있는 노드가 1번

- 제출 기한
  - 11/23 (월) 23:59 (지각 감점: 5%p / 12H, 1주 이후 제출 불가)
  - 두 소스 파일과 생성된 이진 파일 하나 (unix.bin) 를 압축하여 LMS “과제 9” 제출