

---

# 데이터 타입 2

---

Programming Languages

2018 1학기

한양대학교 공과대학 컴퓨터소프트웨어학부  
홍석준

# 목차

---

- 레코드
- 공용체
- 포인터

# 레코드 타입(Record type)

## □ 레코드(Record)

- 개개의 원소들이 이름으로 식별되고, 그 구조의 시작부분으로부터의 오프셋을 통하여 접근되는 데이터 원소들의 집합체
- 이질적인 데이터 원소들의 집합
- COBOL에서 도입, 이후 대부분의 프로그래밍 언어에 포함
- C, C++, C# 에서 struct 데이터 타입으로 지원
- Java와 C#에서 레코드는 데이터 클래스로서 정의될 수 있음

## □ 레코드 정의

- 레코드의 원소들, 즉 필드(fields)이 색인으로 참조되지 않음
- 필드들은 식별자로 명칭되며, 필드에 대한 참조는 식별자를 사용

## □ 배열과 레코드의 비교

- 레코드 타입의 원소 타입은 이질적 (배열은 타입들이 동질적)
- 원소 접근은 레코드가 배열보다 빠름 (배열은 주소 계산 필요)

# 레코드 타입(Record type) 선언의 예

## ❑ C

```
struct student {  
    char name[20];  
    int score;  
    char grade;  
}
```

```
struct student st;
```

# 레코드 타입(Record type) 선언의 예

## ❑ COBOL

```
01  EMPLOYEE-RECORD.  
    02  EMPLOYEE-NAME.  
        05  FIRST    PICTURE IS X(20).  
        05  MIDDLE   PICTURE IS X(10).  
        05  LAST     PICTURE IS X(20).  
    02  HOURLY-RATE PICTURE IS 99V99.
```

## ❑ Ada

```
type Employee_Name_Type is record  
    First : String (1..20);  
    Middle : String (1..10);  
    Last : String (1..20);  
end record;  
type Employee_Record_Type is record  
    Employee_Name: Employee_Name_Type;  
    Hourly_Rate: Float;  
end record;  
Employee_Record: Employee_Record_Type;
```

# 레코드 타입(Record type) 참조

## □ 레코드 필드의 참조

- **OF**(COBOL), **.** (대부분), **%** 등
- COBOL의 레코드 필드 참조 방식

`field_name OF record_name_1 OF . . . OF record_name_n`

- COBOL의 참조 예

`MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD`

- Ada의 예

`Employee_Record.Employee_Name.Middle`

# 공용체 타입(Union type)

## □ 공용체(Union)

- 그 변수가 프로그램 실행 중에 다른 시기에 다른 타입의 값을 저장할 수 있는 타입
- 예) C의 union

```
union number {  
    int value;  
    char data[4];  
} x;  
x.value = 321; ...  
strcpy(x.data, "123"); // 앞의 321을 덮어쓰  
... // 그 후 실수로 x.value++하면?
```

# 공용체 타입(Union type)

## □ 자유 공용체(Free Union)

- C/C++의 union
- 타입 검사를 위한 언어적 지원이 없음

## □ 판별 공용체(Discriminated union)

- ALGOL 68에서 첫 제공, 현재 Ada, ML, Haskell, F#에서 지원
- 타입 지시자(태그 혹은 판별자)를 통한 타입 검사
- Pascal의 판별 공동체의 예

```
type intreal =  
  record tagg : Boolean of  
    true : (blint : integer);  
    false : (blreal : real);  
  end;  
var blurb : intreal;  
blurb.tagg := false;    { real 로 여김 } blurb.blreal := 47.0; { OK }  
blurb.tagg := true;     { int 로 여김 }  blurb.blint  := 47;    { OK }
```



# 포인터(pointer) 타입

## □ 포인터 타입

- 그 변수가 메모리 주소와 특수값 nil로 구성되는 값들의 범위를 갖는 타입
- 다른 용도를 위해서 설계
  - 간접 주소 지정(주소 지정의 유연성)
  - 동적 기억 공간을 관리하는 한 가지 방식을 제공
    - 포인터는 동적으로 할당되는 힙 기억공간의 한 위치를 접근하는데 사용될 수 있음
    - 힙으로부터 동적으로 할당되는 힙-동적 변수들은 흔히 이들과 연관되는 식별자를 갖지 않고, 포인터나 참조 변수들에 의해서만 참조
    - 이름 없는 변수를 무명 변수(anonymous variable)라 부름
- 참조 타입(reference type) : 포인터는 다른 변수를 참조하기 위해서 사용
  - cf) 값 타입(value type) : 데이터 저장을 위해서 사용

# 포인터(pointer) 사용

## □ 포인터 변수는 값으로 메모리 주소를 가짐

- 메모리 주소를 값처럼 얻어 내는 방법 (C 혹은 C++)
  - 일반 변수 앞에 &를 붙이기
  - malloc/new를 통해서 힙-동적 변수의 주소를 가져옴

```
int *ptr;  
int count, init;  
...
```

```
ptr = &init;  
count = *ptr;
```



```
count = init;
```

# 포인터(pointer) 사용

## □ 역참조(Dereferencing)

- 포인터 변수가 내용으로 가지고 있는 주소를 취하여 그 주소에 저장된 값을 가져오기

- \*을 사용

- 예)

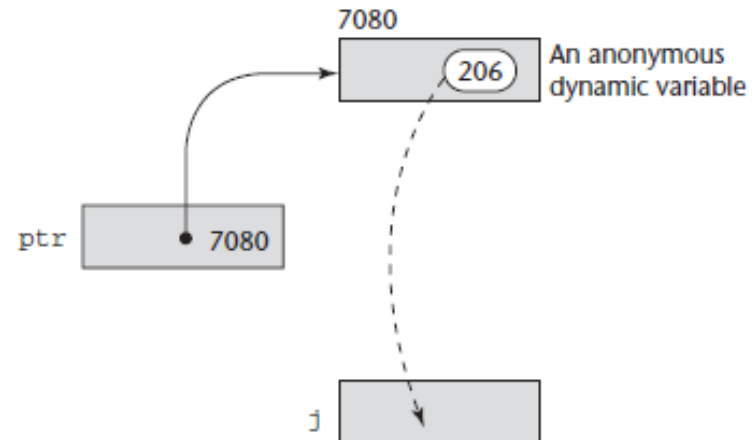
```
int j;
```

```
int * ptr;
```

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

```
*ptr = 206;
```

```
j = *ptr;
```



# 포인터(pointer) 사용

## □ 포인터가 구조체를 가리키는 경우

- 레코드에 대한 포인터가 그 레코드에 속한 필드를 참조하는데 사용할 수 있는 두 가지 방식이 있음.

```
struct student { char * name;  
    int score;  
    char grade;  
};
```

...

```
struct student *s;
```

...

- ① `(*s).score = 90;`
- ② `s-> grade = 'A' ;// 역참조와 필드 참조를 통합`

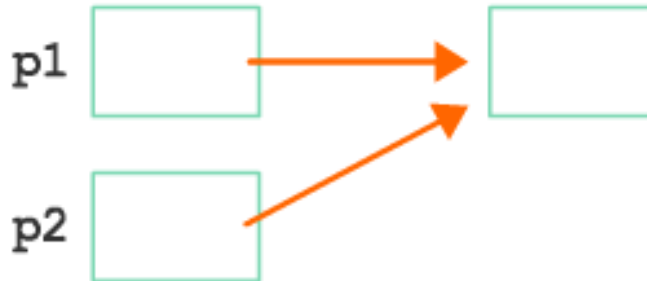
## □ 힙 메모리 관리를 위해서 포인터를 사용

- 명시적인 할당 연산을 제공
  - ✓ 내장 부프로그램 : C의 malloc
  - ✓ 할당 연산자 제공 : C++ , Java의 new

# 포인터의 문제

## □ 허상 포인터(dangling pointer) 혹은 허상 참조(dangling reference)

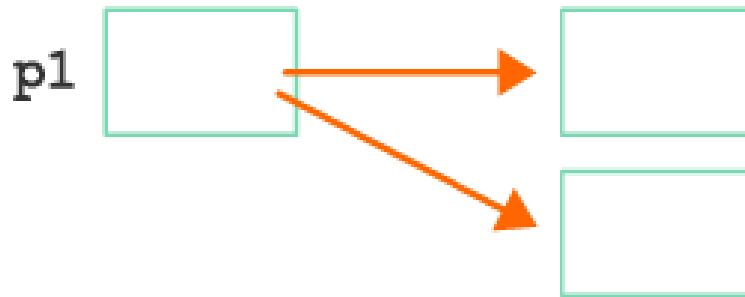
- 이미 회수된 힙-동적 변수의 주소를 포함하고 있는 포인터
- 이 포인터가 가리키고 있는 위치에 새로운 힙-동적 변수가 할당된다면? -> 더 큰 문제
- 허상 포인터 생성
  1. 새로운 힙-동적 변수가 생성되고, 포인터 p1이 그 변수를 가리키도록 설정
  2. 포인터 p2에 p1의 값을 할당
  3. p1이 가리키는 힙-동적 변수가 명시적으로 회수



# 포인터의 문제

## □ 분실된 힙-동적 변수(**lost heap-dynamic variable**)

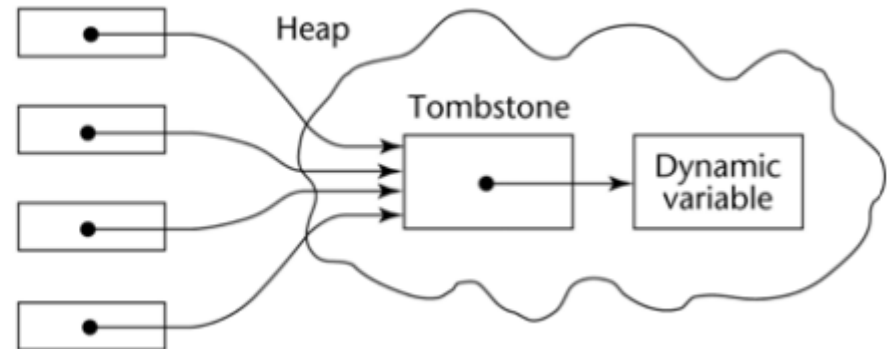
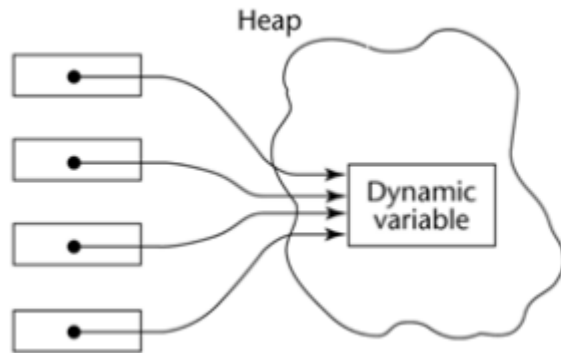
- 쓰레기(garbage) : 사용자 프로그램에서 더 이상 접근될 수 없는 할당된 힙-동적 변수 → 힙-동적 변수는 분실됨 : 메모리 누수 (memory leakage)
- 동적 변수의 명시적 회수가 요구될 경우에 발생
- 분실된 힙-동적 변수의 생성
  1. 포인터 p1이 새롭게 생성된 힙-동적 변수를 가리키도록 설정
  2. p1이 나중에 또 다른 새롭게 생성된 힙-동적 변수를 가리키도록 설정



# 포인터의 문제의 해결책

## □ 허상포인터 문제 해결책 - 비석 접근 방법(tombstone)

- 모든 힙-동적 변수는 힙-동적 변수를 가리키는 비석이라 불리는 특정 셀을 포함
- 실제 포인터 변수는 단지 비석을 가리키고 직접 힙-동적 변수를 가리키지는 않음
- 힙-동적 변수가 반환되면 비석값은 nil로 바뀜(없어지지 않는)
- nil로 설정된 비석을 가리키는 포인터에 대한 참조는 오류로서 탐지될 수 있음

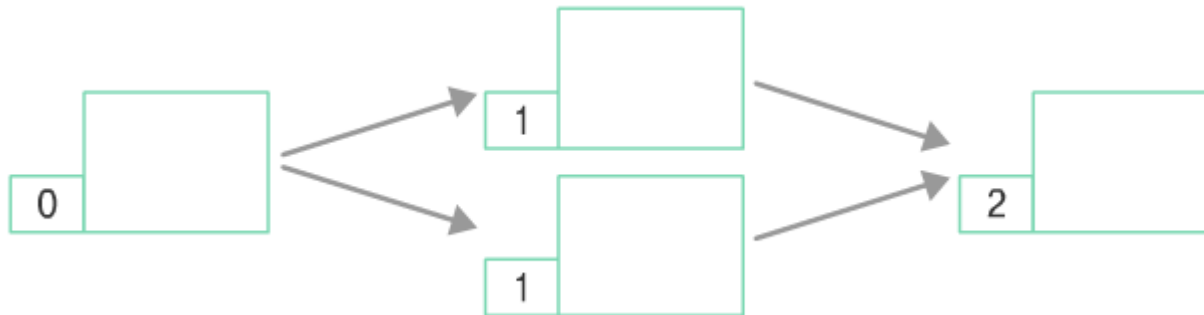


# 포인터의 문제의 해결책

## □ 분실된 힙-동적 변수 문제 해결책 - 쓰레기 회수

### – 참조 계수(reference count)

- 각 셀이 자신을 참조하는 포인터 개수를 기록
- 참조 계수값이 0이면 이 셀에 대한 포인터가 없다는 것을 의미-> 이 셀은 쓰레기가 되었고, 가용 공간 리스트에 반환
- 조기 접근 (eager approach)라고도 불리움





# 포인터의 문제의 해결책

## □ 분실된 힙-동적 변수 문제 해결책 - 쓰레기 회수

### – 표시 수집(mark-sweep)

- 지연 접근(lazy approach) : 메모리가 모자라면 시작
- 각 셀마다 1bit씩 더 필요(쓰레기 or 비쓰레기 표현)
  1. 처음에는 모두 쓰레기로 초기화
  2. 힙을 순회하며 도달 가능한 셀을 비쓰레기 표시로 바꿈
  3. 남은 셀은 모두 쓰레기이므로 회수

---

*Thank you for your attention !!*

---

Q and A