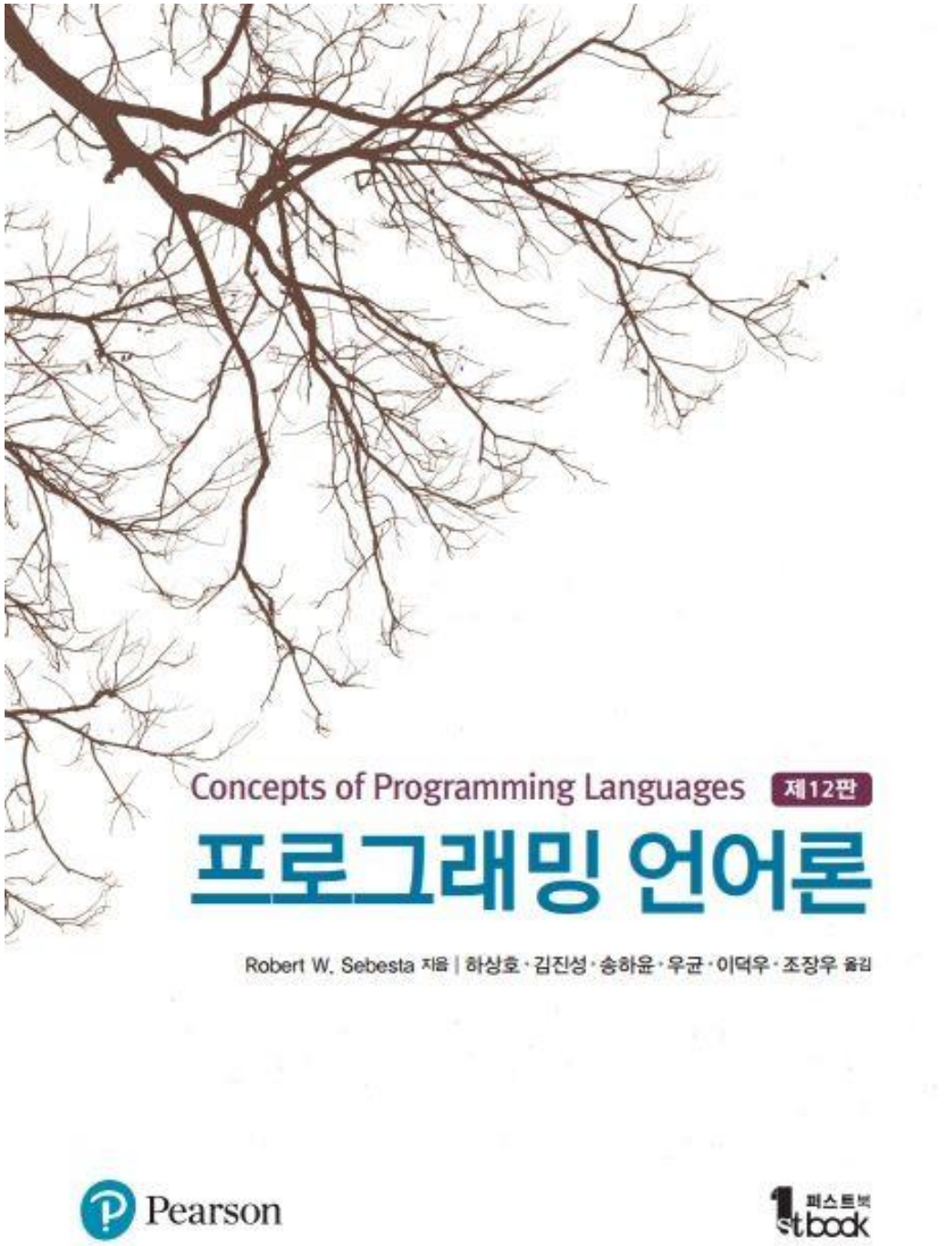
6장

데이터 타입


Concepts of Programming Languages
제12판
프로그래밍 언어론
Robert W. Sebesta 지음 | 하상호 · 김진성 · 송하윤 · 우균 · 이덕우 · 조장우 옮김
Pearson
1st book 퍼스트북

주제

- 서론
- 기본 데이터 타입
- 문자 스트링 타입
- 열거 타입
- 배열 타입
- 연관 배열
- 레코드 타입
- 튜플 타입
- 리스트 타입
- 공용체 타입
- 포인터 타입과 참조 타입
- 선택적 타입
- 타입 검사
- 강 타입
- 타입 동등
- 이론과 데이터 타입

배열 타입

- 배열(array)은 동일한 타입을 갖는 데이터 원소들의 집합체(aggregate)이고, 원소는 그 집합체의 첫번째 원소와의 상대적인 위치로 식별

- 설계시 고려사항
  - 어떤 타입이 첨자에 대해서 적법한가?
  - 원소 참조 시 첨자 식이 범위 검사되는가?
  - 첨자 범위는 언제 바인딩 되는가?
  - 배열 할당은 언제 일어나는가?
  - 톱니형 또는 직사각형 다차원 배열이 허용되는가? 또는 둘 다 허용되는가?
  - 배열이 할당될 때, 초기화 가능한가?
  - 슬라이스(slice)가 존재한다면, 어떤 종류의 슬라이스가 지원되는가?

배열과 인덱스

첨자(subscript) == 인덱스(index)

- 인덱싱은 인덱스로부터 원소로의 사상을 의미
  배열이름(인덱스 리스트) => 원소

- 인덱스 구문
  - 괄호 사용: ()
    - FORTRAN, PL/I, Ada
    - 배열 참조와 함수 호출 간의 일관성(uniformity)를 보여준다: 이 둘 모두 사상(mapping)을 의미
  - 대괄호 사용
    - 다른 대부분의 언어: []

배열 인덱스(첨자) 타입

- 인덱스 타입
  - FORTRAN, C 기반 언어: 정수만 가능

- 인덱스 범위 검사(인덱스 범위 오류는 빈번)
  - Java, C#: 범위 검사 명세
  - C, C++: 범위 검사 명세하지 않음

배열 유형

- 인덱스 범위 바인딩 시기, 기억공간 바인딩 시기, 할당 위치에 기준하여 배열을 4가지로 구분
  - 정적
  - 고정 스택-동적
  - 고정 힙-동적
  - 힙-동적

정적 배열, 고정 스택-동적 배열

- 정적 배열(static array)은 첨자 범위가 정적으로 바인딩, 기억공간 정적 할당
  - C/C++의 static 배열
  - 효율적(efficiency)이나 실행중 배열 공간 유지

- 고정 스택-동적 배열(fixed stack-dynamic array)은 첨자 범위는 정적 바인딩, 기억 공간은 선언문 세련화시간에 할당
  - C/C++의 함수 내부에 선언된 배열
  - 기억공간이 효율적이나 할당/반환 부담

고정 힙-동적 배열

- 고정 힙-동적 배열(fixed heap-dynamic array)은 첨자 범위와 기억 공간 바인딩이 실행중 사용자 프로그램이 요청할 때 이루어지고, 힙으로부터 할당되고, 이후에 그 바인딩은 고정
  - C, C++의 동적 배열, Java, C#의 배열
  - 유연성 vs. 힙 공간 할당 부담

```
int* arr = new int[10];  // 런타임에 크기 정함 → 힙 메모리
arr[5] = 123;
```

힙-동적 배열

- 힙-동적 배열(heap-dynamic array)은 첨자 범위, 기억 공간 바인딩이 동적, 이후에 바인딩은 변경 가능
  - 예: Java의 ArrayList, C#의 List, Python, Perl, JavaScript
  - 유연성 vs. 할당 및 회수 부담

new ArrayList<>( );

```
ArrayList <People> band = new ArrayList<>(10);

band.add("Paul");
band.add("Pete");
…
System.out.println("band.get(1));
```

배열 초기화

- 배열의 기억 공간 할당 시점에 배열 초기화
  - 예제

```
int list[] = {4, 5, 7, 83};
char name[] = “Gildong”;
char *names[] = {“Bob”, “Jake”, “Joe”};
String[] names = {“Bob”, “Jake”, “Joe”};
```

배열 연산

- 배열 연산은 배열 단위의 연산
  - 배정, 접합, 동등/비동등 비교, 슬라이스

- 언어 예
  - APL: 가장 강력한 배열 처리 언어

ΦV: V의 원소를 역순으로	ΦM: M의 열을 역순으로
θM: M의 행을 역순으로	ØM: M의 전치행렬 계산
÷M: M의 역행렬 계산	

  - A+B // A, B 벡터이면 벡터의 합
  - A+.xB // x 를 먼저하고, +를 수행

배열 연산

- Python: 배정(참조 변경), 접합(+), 원소 멤버쉽(in), 비교(is, ==)

```
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)
print(a is b)
```

```
c=a
a[1]=0
print(c)
d = a+b
```

```
a[0] = -1
print(a)
print(d)
```

- C 기반 언어: 배열 연산 제공하지 않으나, Java, C++, C#에서는 메쏘드를 통해서 지원
  - C 언어로 위의 예를 구현하시오
  - Java 언어로 위의 예를 구현하시오.

직사각형 배열 vs. 톱니형 배열

- 직사각형 배열(rectangular arrays)은 모든 행이 동일한 개수의 원소를, 모든 열이 동일한 개수의 원소를 갖는 다차원 배열
  - 직사각형 테이블을 모델링
  - 언어 예: Fortran, C#

```
myArray[3, 7] // in C#
```

- 톱니형 배열(jagged arrays)은 열들의 크기가 동일할 필요가 없는 다차원 배열
  - 배열의 원소가 배열인 경우("an array of arrays")
  - 언어 예: C, C++, Java, C#

```
myArray[3][7] // in C#

int [][]ma = new int [MAX][]; // in Java
for (int i = 0; i <= MAX; i++)
        ma[i] = new int [i+1];
```

슬라이스

- 슬라이스(slice)는 배열의 부분적 구조
  - 배열 일부분에 대한 참조 메커니즘을 제공(예를 들면, 행렬에서 한 개의 행이나 열을 참조 가능)
  - 배열 연산을 제공하는 언어에서 지원

슬라이스 예제

- In Python

```
vector = [2,4,6,8,10,12,14,16]
mat = [[1,2,3], [4,5,6], [7,8,9]]
```

  - vector(3:6) => [8,10,12] // 첨자는 0부터 시작
    - 첫번째 숫자: 슬라이스의 첫번째 원소 첨자
    - 두번째 숫자: 슬라이스의 마지막 원소 첨자보다 1만큼 큰 정수
  - mat[1] //mat[1] = [4,5,6]
  - mat[0][0:2] // [1,2,3] => [1,2]
  - vector[0:7:2] //[2, 6, 10, 14]

배열 구현

- 배열 원소 접근 함수 구현
  - 배열 원소에 대한 첨자 식을 원소의 주소로 사상
  - 원소에 대한 접근 코드는 컴파일러가 생성

- 1차원 배열의 원소 접근 함수

주소(list[k]) = 주소(list[하한]) + (k-하한)*원소_크기

  - In C, 주소(list[k])

다차원 배열 구현

- 다차원 배열의 저장 순서 (1차원 배열로 사상)
  - 행-우선 순서(row major order): 대부분의 언어
  - 열-우선 순서(column major order): Fortran

  - Ex.

3	5	7
1	3	8
6	2	5

다차원 배열 구현 (계속)

- 다차원 배열에서 접근 함수 (행-우선 순서)
  - 배열의 각 차원에 대해서 한 개의 곱셈과 덧셈 필요

주소(a[i,j]) = 주소(a[0,0])
+ ((i번째 행보다 앞선 행의 개수)*(열의 크기)
+ (j번째 열의 왼쪽에 위치한 원소 개수)) *원소_크기

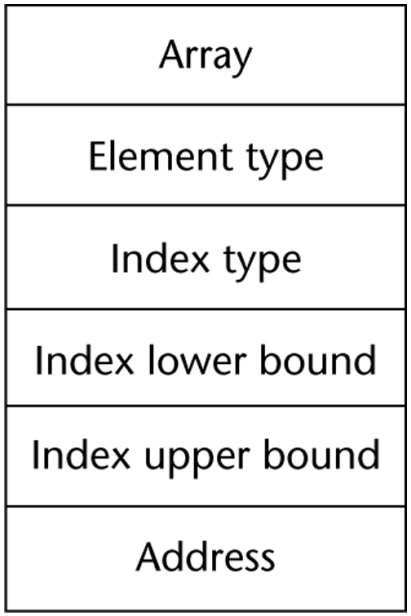
  - Ex. C의 int a[m][n] 배열에서,

주소(a[i][j]) =

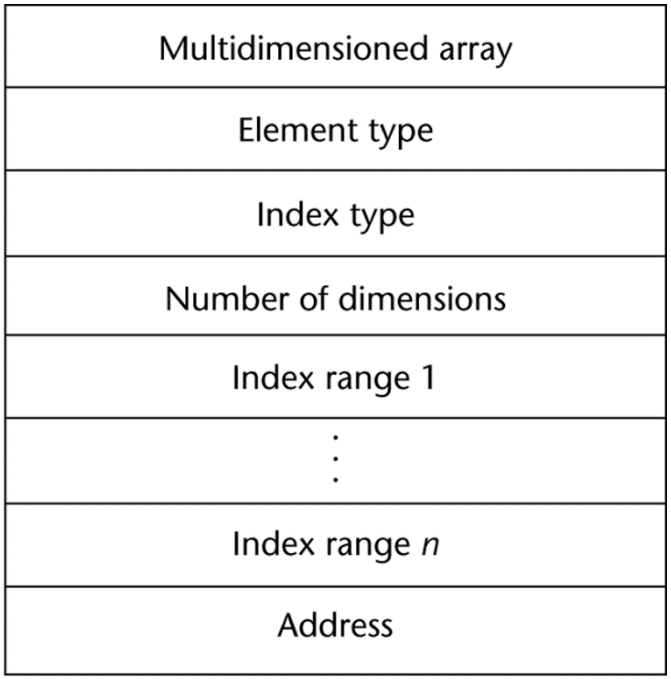
배열 서술자

- 배열 원소 접근 함수에 필요한 정보 포함
- 인덱스 범위에 대한 실행시간 검사
- 첨자 범위의 정적/동적에 따른 컴파일-시간, 실행-시간 서술자

일차원 배열

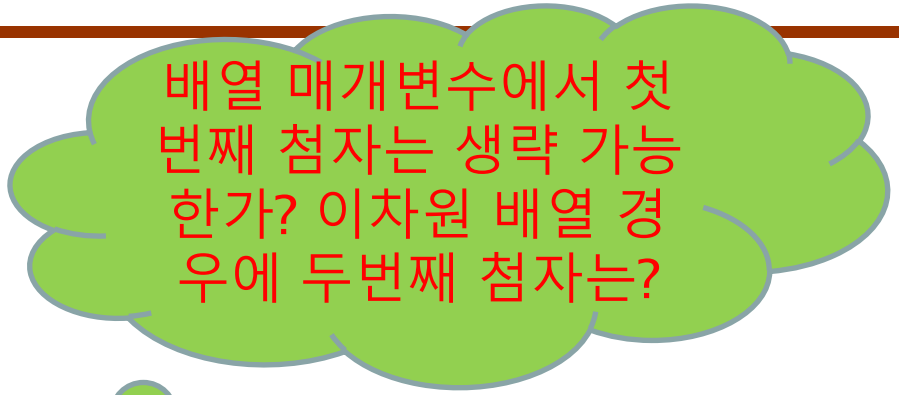


다차원 배열



예제

- 언어 예: C




```
double sum(double a[], int n)
{…}

int sum (int a[][10], int n, int m)
{…}
```

연상 배열

- 연상 배열(associative array)은 원소들 간에 순서가 없으며, 키 값을 통해서 원소에 접근하는 배열
  - 사용자-정의 키가 원소와 함께 저장

- 언어 예
  - Python, Perl에서 지원, Java, C++, C#에서는 클래스 라이브러리로 지원

In Python // dictionary

```
d = {1:'apple', 2:'banana'} // (key:value)
d[1]
```

In Perl // hash

```
%hi_temp = {"Mon"=>77, "Tue"=>79, "Wed"=> 65, ...};
$hi_temp {"Wed"} = 83;
```

레코드 타입

- 레코드(record) 타입은 이질적일 수 있는 데이터 원소들의 집단체(aggregate)
  - 각 원소들은 인덱스가 아닌 이름으로 식별
  - COBOL에서 도입, 대부분 언어에서 제공
  - 언어 예: C, C++, C#에서 struct 타입으로 지원, Python에서는 hash로 지원

- 설계 고려사항
  - 필드 참조의 구문 형식은?
  - 생략 참조가 허용되는가?

레코드 필드 참조

COBOL의 레코드 선언

```
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.
```

C의 구조체 선언

```
struct  EMPLOYEE-RECORD {
    struct EMPLOYEE-NAME {
        char  FIRST [20];
        char  Middle [20];
        char  LAST[20]              }
    float HOURLY-RATE PICTURE
}
```

- 필드 참조 구문

```
// in COBOL
  field_name OF record_name_1 OF ... OF record_name_n

// 다른 언어: 도트 표기법 사용
 record_name_n.record_name_n-1. ... record_name_1.field_name
```

튜플 타입

- 튜플(tuple)은 레코드와 유사하나 원소들이 명명되지 않는 데이터 타입

- In Python
  - 변경 불가 튜플 타입 지원: 주로 이질적 데이터 저장
  - 값-전달 매개변수 전달 시, 함수가 여러 개의 값 반환 시 유용

```
myTuple = (3, 5.8, 'apple') # 요소들은 동일 타입일 필요가 없다
myTuple[0]          # 인덱싱을 통한 참조: 튜플의 첫번째 원소 참조
myTuple[1] = 5              # Error: 튜플은 변경 불가(immutable)
myTuple += (2, 3)           # '+'는 접합 연산자로 새로운 튜플 생성
del                         # 삭제
```

리스트 타입

- LISP에서 처음으로 도입, 최근에 일부 명령형 언어에 도입
- In LISP, Scheme
  - 리스트는 괄호로 구분되고, 원소들은 콤마로 구분되지 않는다:
    (A B C D)
    (A (B C) D)
  - 데이터와 코드는 동일한 형식을 갖는다
    - (A B C)가 코드일 때, 함수 A가 매개변수 B C에 적용
  - 리스트가 데이터일 경우에 어포스트로피로 표시한다.
    - '(A B C)는 데이터이다.

리스트 기본 연산

- 리스트 분리 함수

  (CAR '(A B C)) // 리스트의 첫번째 원소 반환

  (CDR '(A B C)) // 리스트의 첫번째 원소를 제외하고 반환

- 리스트 생성 함수

  (cons 'A '(B C)) // 첫번째 매개변수를 두번째 매개변수 리스트에
  // 포함시켜서 새로운 리스트 생성

  (list 'A 'B '(C D)) // 매개변수들을 원소로 포함하는 리스트 생성/반환

리스트 예

- Python의 리스트 타입
  - 주로 동질적 데이터를 저장하나 이질적 데이터도 저장 가능
  - 리스트는 변경 가능(mutable)
  - 튜플처럼 사용 가능: 인덱싱, 접합, 슬라이싱(부분 참조), in 등

```
myList = [3, 5.8, “grape”]
x = myList[1]    //x에 5.8을 할당: 리스트 원소들은 0부터 인덱싱
del myList[1]    // 두번째 원소 삭제
```

  - 리스트 함축(list comprehension) 지원: 집합 표기법 기반

```
[x*x for x in range (1, 12) if x % 3 == 0]
```

- C#, Java에서 List, ArrayList 클래스로 지원

공용체 타입

- 공용체(union)는 그 변수가 프로그램 실행 중에 다른 시기에 다른 타입의 값이 저장될 수 있는 타입

- 자유 공용체(free union)
  - 타입 검사를 지원하지 않음
  - 언어 예: C, C++

```
union flexType {
   int intEl;
   float floatEl;
};

union flexType e;
float x;
…
e.intEl = 27;
x = e.floatEl; // 타입 검사?
```

- 판별 공용체(discriminated union)
  - 타입 검사 지원
  - 판별자(discriminant) 또는 태그(tag)라는 타입 지시자 포함
  - 언어 예: ML, Haskell, F#

공용체 평가

- 공용체가 안전한가?
  - 자유 공용체 vs. 판별 공용체

- 언어 예: C,C++
  - 공용체 참조에 대한 타입 검사 지원하지 않음
  - 타입 안전하지 않은 이유 중 하나
  - Java, C#은 공용체를 포함하지 않음