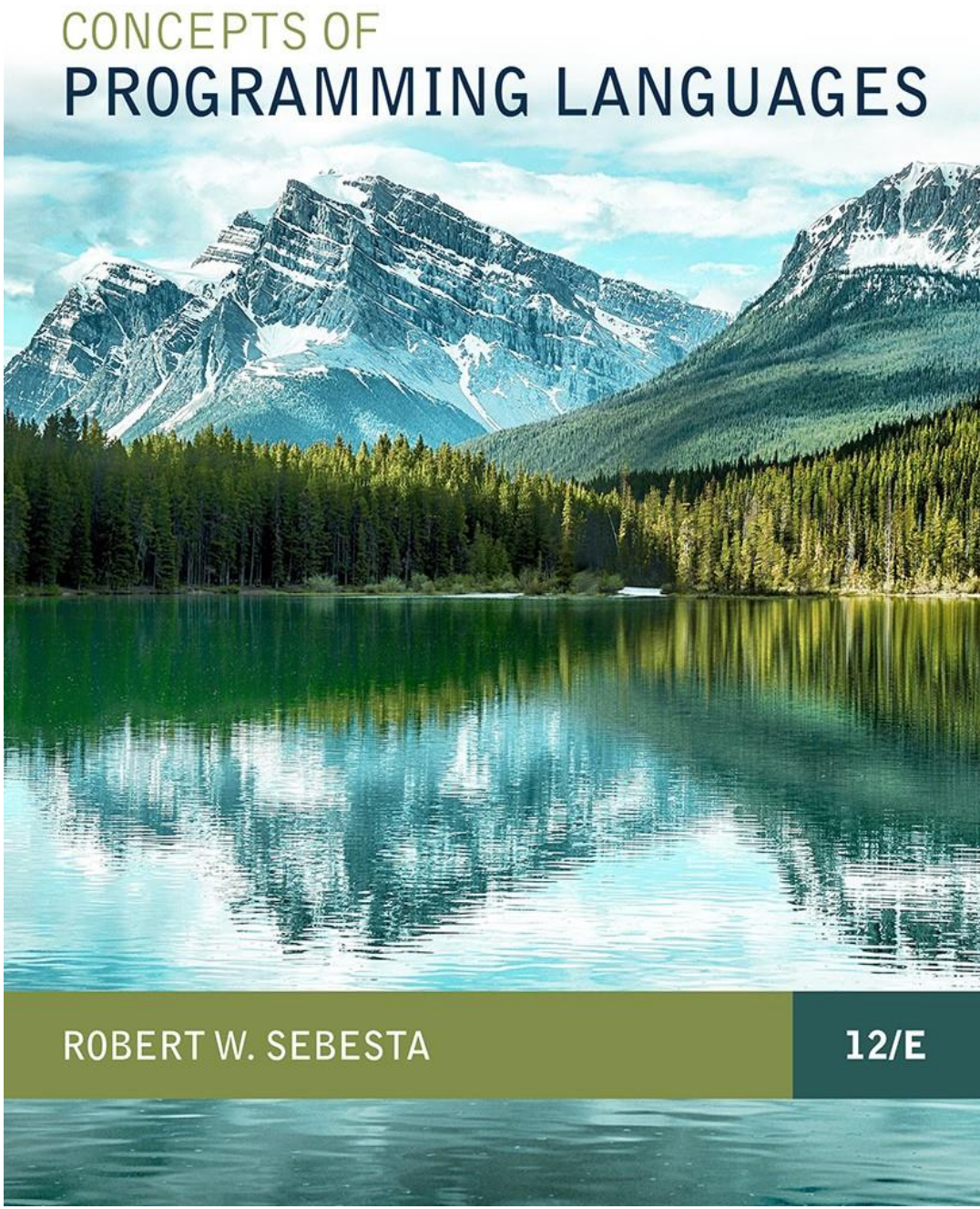
Chapter 6

Data Types



ISBN 0-321-49362-1

학습 목표

- 변수의 기본 데이터 타입
- 구조화 데이터 타입인 배열, 문자열, 레코드, 튜플, 리스트, 공용체 타입
- 포인터 타입과 참조 타입
- 타입 검사
- 강 타입
- 타입 동등

서론

- 데이터 타입 : 데이터 값들의 모임 +
  그 값에 대해 미리 정의된 연산들의 집합
- 서술자(descriptor) : 변수의 속성들을 저장하는 메모리 영역
- 모든 데이터 타입의 설계 시 고려사항:
  - 연산의 종류 정의
  - 연산의 명세 방법

Primitive Data Types(기본 데이터 타입)

- 거의 모든 프로그래밍언어에서 제공
- Primitive data types: Those not defined in terms of other data types

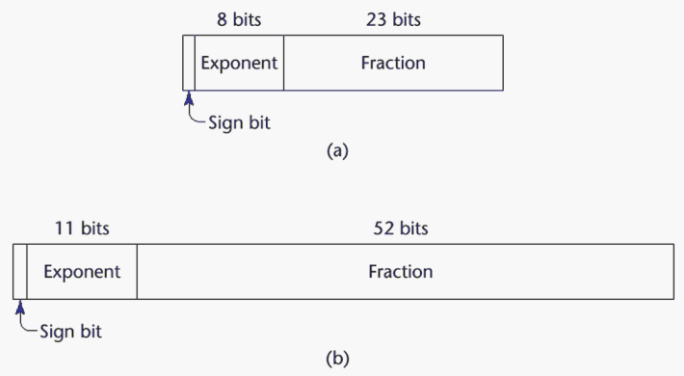
- Numeric type
  - Integer, Floating-Point, Complex, Decimal
- Boolean type
- Character type

Primitive Data Types: Integer

- Almost always an exact reflection of the hardware so the mapping is trivial
- There may be as many as eight different integer types in a language
- Java's signed integer sizes: **`byte`**, **`short`**, **`int`**, **`long`**

Primitive Data Types: Floating Point

- Model real numbers, but only as approximations
- Languages for scientific use support at least two floating-point types (e.g., **`float`** and **`double`**; sometimes more
- Usually exactly like the hardware, but not always
- IEEE Floating-Point Standard 754



Primitive Data Types: Complex

- Some languages support a complex type, e.g., C99, Fortran, and Python
- Each value consists of two floats, the real part and the imaginary part
- Literal form (in Python):

  `(7 + 3j)`, where `7` is the real part and `3` is the imaginary part

Primitive Data Types: Decimal

- For business applications (money)
  - Essential to COBOL
  - C# offers a decimal data type
- Store a fixed number of decimal digits, in coded form (BCD)
- *Advantage*: accuracy
- *Disadvantages*: limited range, wastes memory

Primitive Data Types: Boolean

- Simplest of all
- Range of values: two elements, one for "true" and one for "false"
- Could be implemented as bits, but often as bytes
  - Advantage: readability

Primitive Data Types: Character

- Stored as numeric codings
- Most commonly used coding: ASCII
- An alternative, 16-bit coding: Unicode (UCS-2)
  - Includes characters from most natural languages
  - Originally used in Java
  - Now supported by many languages
- 32-bit Unicode (UCS-4)
  - Supported by Fortran, starting with 2003

Character String Types

- Values are sequences of characters
- 설계 고려사항 :
  - 스트링이 문자 배열의 종류인가 또는 기본 타입인가?
  - 스트링의 길이는 정적 또는 동적인가?
- 대표적인 연산 :
  - Assignment and copying
  - Comparison (=, >, etc.)
  - Catenation
  - Substring reference
  - Pattern matching

String Type in Certain Languages

- C and C++
  - Not primitive
  - Use **`char`** arrays and a library of functions that provide operations
- SNOBOL4 (a string manipulation language)
  - Primitive
  - Many operations, including elaborate pattern matching
- Fortran and Python
  - Primitive type with assignment and several operations
- Java (and C#, Ruby, and Swift)
  - Primitive via the `String` class
- Perl, JavaScript, Ruby, and PHP
  - Provide built-in pattern matching, using regular expressions

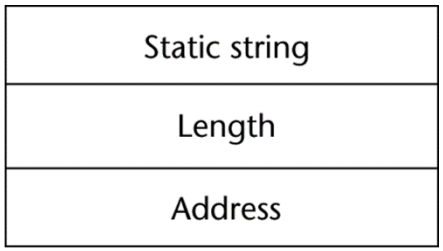
스트링 길이 선택사항

- 정적 길이 스트링: 스트링이 생성될 때 설정
  - Java의 `String` class, Python의 스트링, C++표준 클래스 라이브러리, C#과 F#의 .NET 클래스 라이브러리
- 제한된 동적 길이 스트링 : 길이를 유지하는 게 아니라 특수 문자가 스트링의 끝을 의미하도록 사용
  - C, C++의 C-style 스트링
- 동적 길이 스트링 : no maximum
  - JavaScript, Perl, SNOBOL4

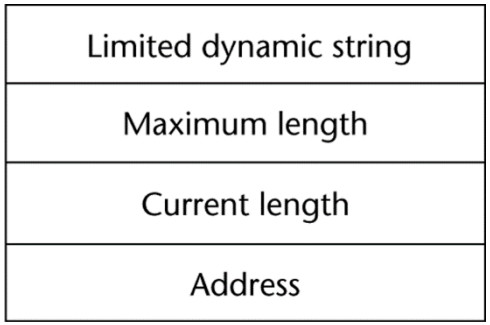
스트링 타입의 구현

- 정적 스트링 : compile-time descriptor
- 제한된 동적 스트링 : run-time descriptor(C와 C++의 경우는 스트링의 끝이 널문자이므로 서술자 불필요)
- 동적 스트링 : 현재 길이만 저장하는 run-time descriptor; 할당/회수가 가장 큰 문제

Compile- and Run-Time Descriptors



정적 스트링을 위한
컴파일-시간 서술자



제한된 동적 스트링을
위한 실행-시간 서술자

사용자-정의 순서 타입

- 순서 타입은 값의 가능한 범위가 양의 정수 집합과 연관되는 타입
- 기본 순서 타입의 예(Java)
  - **`integer`**
  - **`char`**
  - **`Boolean`**

열거 타입(Enumeration Types)

- 모든 가능한 이름 상수 값들을 정의하는 타입
- C# example

```
enum days {mon, tue, wed, thu, fri, sat,
  sun};
```

- 설계 고려사항
  - 열거 상수가 한 개 이상의 타입 정의에 나타나도 되는가? 그렇다면 이러한 상수 참조시 그 타입이 어떻게 검사되는가?
  - 열거 타입 값들이 정수로 강제변환 되는가?
  - 다른 타입들이 열거 타입으로 강제변환 되는가?

열거 타입의 평가

- 판독성에 도움 : **e.g., color**를 숫자로 코딩할 필요 없음
- 신뢰성에 도움 : **e.g.,** 컴파일러가 검사:
  - 산술연산 불가
  - 열거 변수는 정의된 범위 밖의 값을 배정 불가
- **C** : 열거변수를 정수변수처럼 취급, 위 장점 없음
- **C#, F#, Swift, Java 5.0** : 열거변수가 **int**로 강제 변환 안됨

배열 타입

- 배열(array) : 동질적인 데이터 원소들의 집단체이며, 개개의 원소는 집단체에서 첫 번째 원소와의 상대적인 위치에 의해서 식별됨

배열의 설계 고려 사항

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- What is the maximum number of subscripts?
- Can array objects be initialized?
- Are any kind of slices supported?

Array Indexing

- *Indexing* (혹은 subscripting) 은 인덱스와 원소간의 mapping

  array_name (index_value_list) → an element

- 인덱스 구문
  - Fortran과 Ada는 ( ) 사용
    - Ada는 명시적으로 ( )를 사용함(배열 참조와 함수호출은 모두 사상(mapping)이므로 이들간에 균일성(uniformity)을 보이기 위해서)
  - 대부분 다른 언어는 [ ]를 사용

Arrays Index(Subscript) Types

- FORTRAN, C: integer only
- Java: integer type only
- Index의 범위 검사(신뢰성에 중요 요소)
  - C, C++, Perl, Fortran 등 대부분은 범위검사 X
  - Java, ML, C#은 범위검사를 명세

첨자 바인딩과 배열 유형

- 배열 변수에 대한 첨자타입의 바인딩은 정적이나, 첨자값 범위는 정적/동적 바인딩

- 첨자값 범위, 기억공간 바인딩, 기억공간종류에 따른 배열의 종류
  - 정적 배열
  - 고정 스택-동적 배열
  - 고정 힙-동적 배열
  - 힙-동적 배열

첨자 바인딩과 배열 유형(continued)

- **정적 배열** : 첨자 범위가 정적으로 바인드되고 기억공간 할당도 정적이다 (즉, 실행시간 이전)
  - 장점: 효율성(동적 할당/회수가 없음)

    (ex) C,C++에서 static 배열

- **고정 스택-동적 배열** : 첨자 범위가 정적으로 바인드되나 기억공간 할당이 선언문 실행시 이루어짐
  - 장점: 공간 효율성 (동시에 활성화되지 않으면 공간 공유가능)

    (ex) C,C++에서 배열

첨자 바인딩과 배열 유형 (continued)

- **고정 힙-동적 배열** : 고정 스택-동적 배열과 유사 ; 첨자범위와 기억공간 바인딩은 동적이나 힙 할당 이후 고정됨 장점 : 유연성 (배열의 크기는 맞춤화)

  (ex) Java의 모든 배열, C#, C, C++

- **힙-동적 배열** : 첨자범위의 바인딩과 기억공간 할당은 동적이며 배열의 존속기간중 여러 번 변경가능한 배열
  - 장점 : 유연성 (배열은 실행 중 증감 가능)
  - Perl, JavaScript, Python, Ruby

배열 초기화

- 기억공간 할당 시 초기화
  - C, C++, Java, C#, Swift
  - C# example :

```
int list [] = {4, 5, 7, 83};
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"};
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

이질 배열(Heterogeneous Arrays)

- 원소들이 동일한 타입이 아닌 배열
- Perl, Python, JavaScript, Ruby

배열 초기화

- C-기반 언어
  - `int list[] = {1, 3, 5, 7}`
  - `char *names[] = {"Mike", "Fred", "Mary Lou"};`
- Python
  - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
   puts [0, 9, 36, 81] in list
```

배열 연산

- 배정, 접합, 비교(동등과 비동등), 슬라이스
- APL은 단항 연산자는 물론 벡터와 행렬을 위한 가장 강력한 배열처리연산을 제공 ((ex) reverse column elements)
- Python에서 배열 원소는 객체에 대한 참조이고 객체들은 임의의 타입일 수 있기 때문에 이질적임; 배열 배정 가능하나 참조 변경임; 또한 배열 접합(+)과 원소 멤버십(in) 연산, 비교연산(is, ==) 제공
- C-기반 언어들은 배열연산 제공하지 않음(Java, C++, C#은 메소드를 통해 가능)

직사각형 배열과 톱니형 배열

- **직사각형 배열(rectangular array)** : 모든 행들이 동일 개수의 원소를 갖고, 모든 열들이 동일 개수의 원소를 갖고 있는 다차원 배열
- **톱니형 배열(jagged matrix)** : 행들의 크기가 동일할 필요가 없는 다차원 배열
  - 다차원 배열이 실제로는 배열들의 배열일 때 가능
- C, C++, Java는 톱니형 배열 지원
  (ex) myArray[3][7]
- F#과 C# 은 직사각형 배열과 톱니형 배열 지원

슬라이스(slice)

- 배열의 슬라이스는 배열의 어떤 부분 구조;
  배열의 부분을 한 단위로 참조하기 위한 메커니즘
- 슬라이스는 배열 연산을 가진 언어에서만 유용함

Slice Examples

- Python

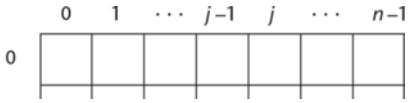
```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

  - `vector[3:6]` is a three-element array
  - `mat[0][0:2]` is the first and second element of the first row of `mat`
  - `vector[0:7:2]`

배열의 구현

- Access function maps subscript expressions to an address in the array
- 1차원 배열의 접근함수 :

address(list[k]) = address (list[lower_bound])
+ ((k – lower_bound) * element_size)

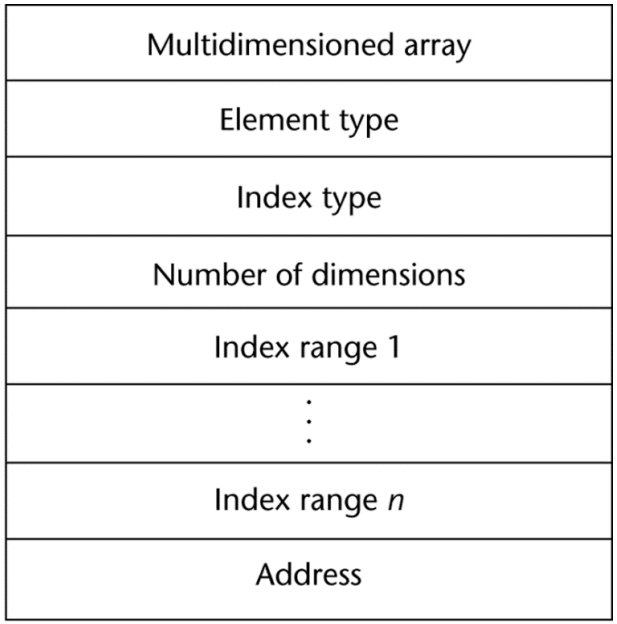


Array
Element type
Index type
Index lower bound
Index upper bound
Address

1차원 배열에 대한 컴파일-시간 서술자

다차원 배열 접근

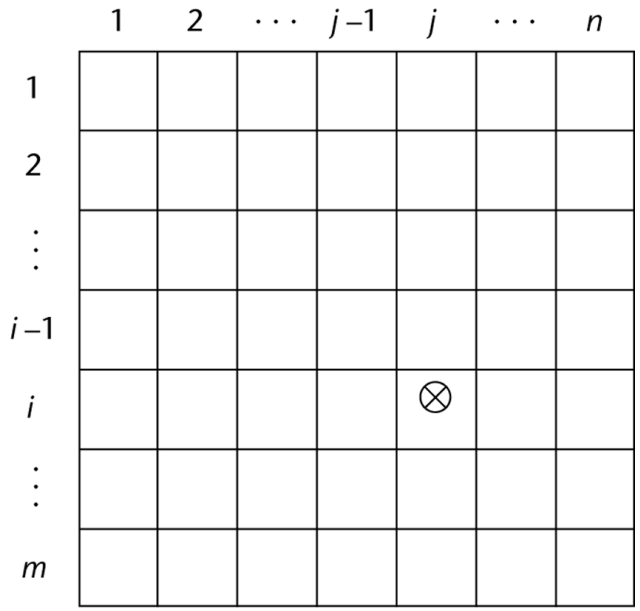
- 1차원으로 사상되는 두 가지 방법 :
  - 행-우선 순서(row major order) - 대부분의 언어
  - 열-우선 순서(column major order) - in Fortran
  - 다차원 배열에 대한 컴파일-타임 서술자



다차원 배열에서 한 원소의 위치

- 일반 형식

Location (a[I,j]) = address of a [row_lb,col_lb] +
(((i - row_lb) * n) + (j - col_lb)) * element_size



연상 배열(Associative Arrays)

- 원소 수와 동일한 개수의 키값(key)으로 인덱싱되는 무순서 데이터 원소들의 모임
  - 사용자-정의 키가 저장되어야 함: (키, 값)
- 설계 고려사항:
  - 원소들에 대한 참조의 형식은?
  - 배열의 크기가 정적 or 동적?
- Built-in type in Perl, Python, Ruby, Swift

연상 배열(in Perl)

- 원소들이 해시함수로 저장/인출되므로 해시(hash)라 함
- 이름은 **%로 시작; 리터럴들은 ( )로 구분**

```
%hi_temps = ("Mon"=>77, "Tue"=>79, "Wed"=>65, …);
```

- 첨자는 {키}로 표시

```
$hi_temps{“Wed”} = 83;    //  변경/추가
```

- 원소의 삭제는 **delete** 사용

```
delete $hi_temps{"Tue"};
```

레코드 타입

- 레코드(*record*)는 이질적인 데이터 원소들의 집단체; 개개의 원소들이 이름으로 식별되고 그 구조의 시작 부분으로부터의 오프셋을 통하여 접근
- 설계 고려사항 :
  - 필드에 대한 참조의 구문 형식은 ?
  - 생략 참조가 허용되는가?

레코드 정의(COBOL)

- COBOL은 수준 번호를 사용하여 중첩 레코드를 표시한다.

```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```

레코드 필드의 참조

- 레코드 필드 참조
  1. COBOL
  field_name OF record_name_1 OF ... OF record_name_n
  2. 그 외 언어들 (dot notation)
  record_name_1.record_name_2. ... record_name_n.field_name

- 완전 자격 참조(fully qualified reference)는 모든 레코드 이름을 포함한다.

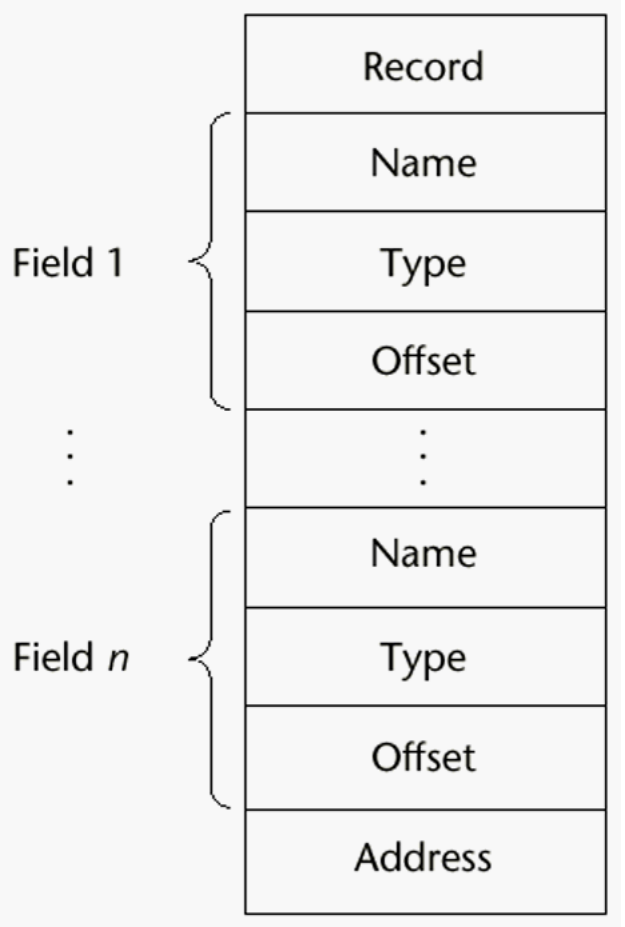
- 생략 참조(elliptical reference)는 참조가 모호하지 않다면, 레코드 이름을 생략한다.
  (ex) COBOL에서,
  FIRST, FIRST OF EMP-NAME, FIRST of EMP-REC

배열과 비교 평가

- 레코드는 데이터 값들이 이질적인 모임일 때 사용
- 배열 원소의 접근은 레코드 필드의 접근보다 더 느림
  (∵ 필드 이름은 정적이나, 배열 첨자는 동적임)
- 동적 첨자도 레코드 필드 접근에 사용될 수 있지만, 이는 타입 검사를 허용하지 않을 것이고 처리속도도 더 느려질 것이다.

레코드 타입의 구현

- 레코드의 시작주소에 상대적인 오프셋 주소가 각 필드에 연관된다.
- 실행-시간 서술자는 불필요



레코드에 대한 컴파일-시간 서술자

튜플 타입

- 튜플은 원소들이 명명되지 않는다는 것을 제외하면 레코드와 유사
- Python, ML, F#에서는 함수가 여러 개의 값을 반환할 때, 배열이 함수에 매개변수로 전달되고 쓰기보호되어야 할 때 튜플을 사용
- Python
  - 튜플은 변경불가(immutable)
  - 튜플은 튜플 리터럴을 할당함으로써 생성

    ```
    myTuple = (3, 5.8, 'apple')
    ```
  - 첨자를 [ ]와 결합시켜 참조(첨자는 1로 시작)

    ```
    myTuple[1]
    ```
  - `+ : catenation,  del : delete`

튜플 타입 (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

  - 튜플 참조:

    `#1(myTuple)` : 첫번째 원소

  - 새로운 튜플 타입 정의:

```
type intReal = int * real;
```

    (The asterisk is just a separator)

- F#

```
let tup = (3, 5, 7)
let a, b, c = tup
```

  This assigns a tuple to a tuple pattern `(a, b, c)`

리스트 타입

- 함수형 언어에서 처음 도입되었으나, 최근 일부 명령형 언어에 도입됨
- LISP와 Scheme의 리스트는 괄호로 구분되고 콤마를 사용 않음

  (A B C D) 와 (A (B C) D)

- 데이터와 코드는 동일한 구문 형식을 가짐
  - 데이터로서 (A B C)
  - 코드로서 (A B C)는 매개변수 B와 C를 갖는 함수 A에 대한 호출
- 인터프리터가 구별하기 위해 데이터이면 리스트 앞에 쿼트(')를 표시

  '(A B C)는 데이터로 처리

리스트 타입(continued)

- 리스트 연산(Scheme)
  - CAR : 리스트 파라미터의 첫 원소를 리턴

    (CAR '(A B C)) → A
  - CDR : 첫 원소를 제거한 후의 나머지 리스트를 리턴

    (CDR '(A B C)) → (B C)
  - CONS : 첫 파라미터를 두번째 파라미터 리스트의 첫 원소로 넣는다

    (CONS 'A (B C)) → (A B C)
  - LIST : 새로운 리스트를 리턴

    (LIST 'A 'B '(C D)) → (A B (C D))

리스트 타입(continued)

- 리스트 연산(ML)
  - 리스트는 [ ]로 표현, 원소는 콤마로 구분
  - 리스트 원소는 동일한 타입이어야 함
  - Scheme의 `CONS` 함수는 binary operator in ML, `::`

    `3 :: [5, 7, 9]` → `[3, 5, 7, 9]`
  - Scheme의 `CAR` 와 `CDR` 함수는 각각 `hd` 와 `tl`

리스트 타입 (continued)

- F# 리스트
  - 원소들이 세미콜론으로 구분되고,
    `hd` 와 `tl` 이 `List` class의 메소드임
  - 그 외는 ML과 동일
- Python 리스트
  - 리스트 데이타타입은 Python의 배열
  - Scheme, Common LISP, ML, F#과 달리,
    Python의 리스트는 mutable
  - 원소들은 어떤 타입도 가능
  - 배정문으로 리스트 생성

```
myList = [3, 5.8, "grape"]
```

리스트 타입 (continued)

- Python 리스트 (continued)
  - 리스트 원소는 첨자로 참조가능(인덱스는 0으로 시작)

```
x = myList[1]   # x = 5.8
```

  - 리스트 원소 삭제시 del

```
del myList[1]
```

  - 리스트 함축 – set notation으로 유도

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

    `range(12)` 는 `[0,1,2,3,4,5,6,7,8,9,10,11]` 생성함

    따라서, 구성된 리스트 : `[0, 9, 36, 81]`

Union(공용체) types

- 공용체는 그 변수가 실행 중에 다른 시기에 다른 타입의 값을 저장할 수 있는 타입
- 설계 고려사항
  - 타입 검사가 필요한가?

판별 공용체와 자유 공용체

- C, C++는 타입검사가 없는 공용체 구조를 제공 → 자유공용체(*free union*)라 함

```
union flexType {
  int intEl;
  float floatEl;
};
union flexType el1;
float x;
…
el1.intEl=27;
x = el1.floatEl;      // 타입 검사되지 않음
```

- 공용체에 대한 타입 검사를 위해 각 공용체 구조는 타입 지시자를 포함함; 이 지시자는 판별자(*discriminant*)라 부름;
  - ML, Haskell, F#

공용체의 평가

- 자유 공용체는 안전하지 않음
  - 공용체 참조에 대한 타입 검사 않음
- Java와 C#은 공용체를 지원하지 않음
  - 프로그래밍 언어에서 안전성에 대한 관심 증가함을 반영

포인터 타입과 참조 타입

- 포인터 타입 변수는 메모리 주소 혹은 ***nil*** 의 값을 갖는 변수
- 간접주소지정 제공
- 동적 메모리를 관리하는 방식 제공
  - 포인터는 동적으로 할당되는 힙이라 불리는 기억공간 영역의 한 위치를 접근하는데 사용
  - *힙-동적 변수*는 포인터나 참조타입 변수에 의해서만 접근

설계 고려사항

- 포인터 변수의 영역과 존속기간은?
- 힙-동적 변수(포인터가 참조하는 값)의 존속기간은?
- 포인터는 가리키는 값의 타입에 대해서 제약받는가?
- 포인터가 동적 메모리 관리, 간접주소지정, 또는 두 가지 모두를 위해서 사용되는가?
- 포인터 타입, 참조 타입, 또는 두 가지 모두를 지원하는가?

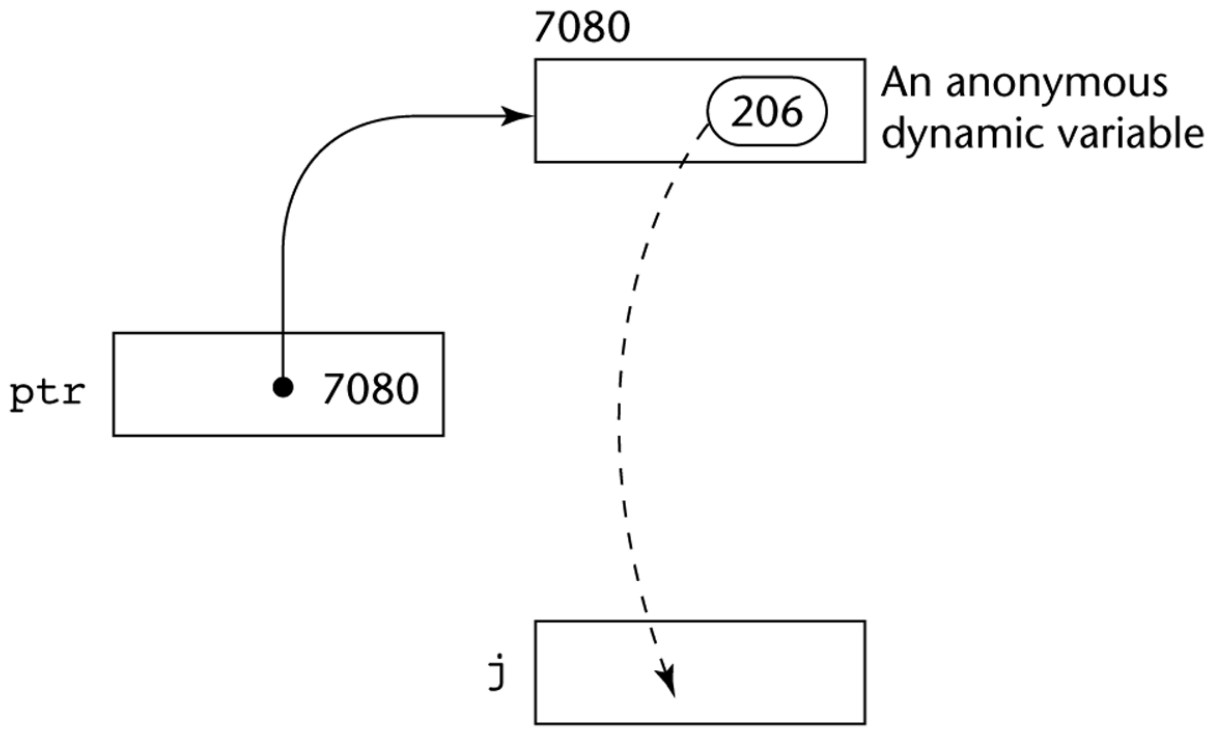
포인터 연산

- 기본적인 포인터 연산 : **배정**과 **역참조**
- **배정**은 포인터 변수의 값을 유용한 주소로 설정
- **역참조**는 포인터의 값에 의해 표현되는 위치에 저장된 값을 가져온다.
  - 역참조는 명시적/묵시적 가능
  - C++는 역참조 연산자(*)를 통해 명시적 연산

    (ex) `j = *ptr`

    : `ptr`에 위치한 값을 `j`에 배정

포인터 연산 예시



배정 연산 j = *ptr;

포인터의 문제

- 허상 포인터(Dangling pointer) : 허상 참조
  - 이미 회수된 힙-동적 변수의 주소를 포함하는 포인터(arrayPtr1)

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete []= arrayPtr2;
```

- 분실된 힙-동적 변수
  - 더 이상 접근할 수 없는 할당된 힙-동적 변수(쓰레기라 부름)
    - 포인터 p1이 새롭게 생성된 힙-동적 변수를 가리키도록 설정
    - p1이 나중에 또 다른 새롭게 생성된 힙-동적 변수를 가리키도록 설정
    - 첫 번째 할당된 힙-동적 변수는 이제 접근불가

      → 메모리 누수(*memory leakage*)

C와 C++의 포인터

- 직접 주소를 다루므로 유연하지만 조심해서 사용
- 언제, 어디에 할당된 변수든 포인터는 가리킬 수 있음
- 용도 : 동적 메모리 관리와 addressing
- 포인터 산술연산 가능
- 명시적인 역참조 연산자(*)와 주소 연산자(&)
- 도메인 타입은 고정될 필요 없음 (**void** * 사용)

  void * 은 어떤 타입도 가리킬 수 있으며 타입검사 가능 (역참조 불가능)

C and C++ 포인터 산술연산

```
float stuff[100];
float *p;
p = stuff;
```

```
*(p+5) ==  stuff[5] ==  p[5] == *(stuff+5)
*(p+i) ==  stuff[i] ==  p[i] == *(stuff+i)
```

참조 타입

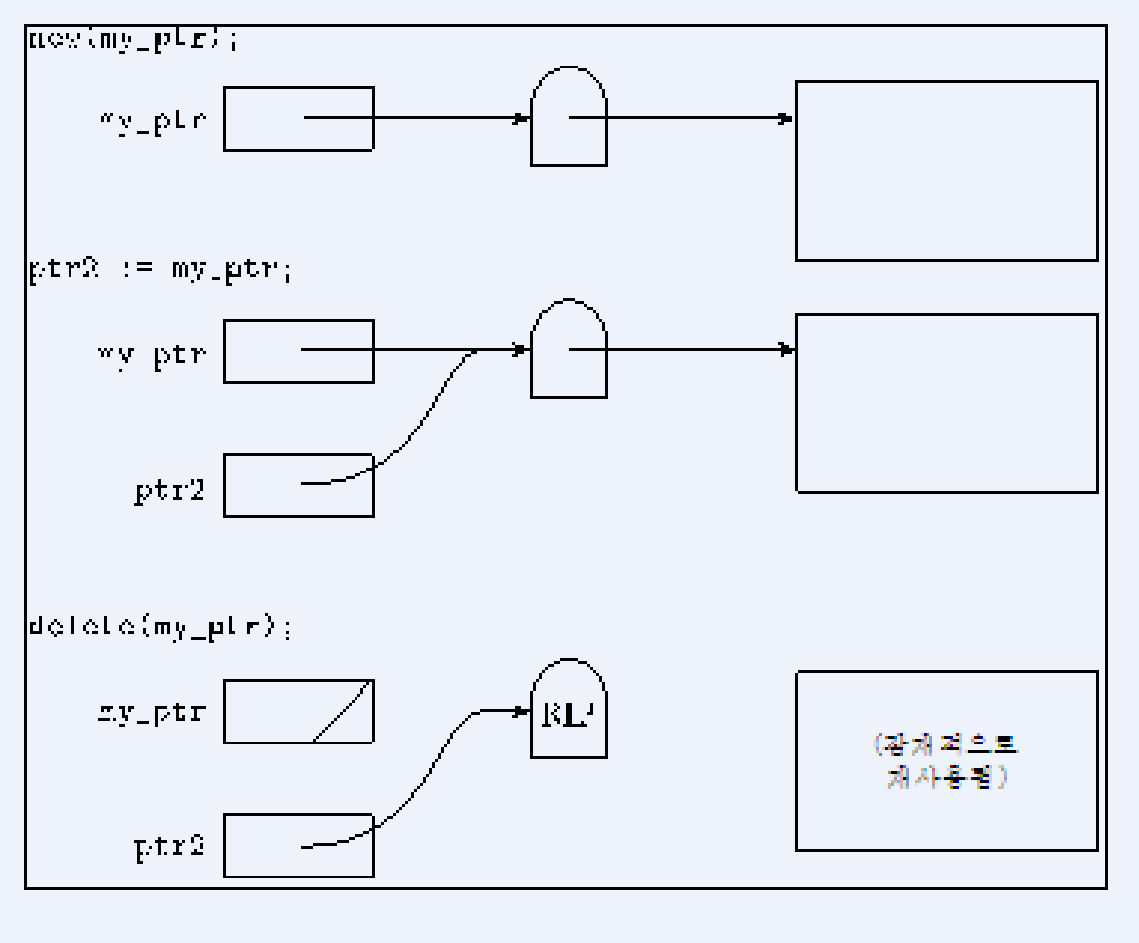
- C++ 는 형식매개변수에 사용되는 참조 타입이라는 특별한 포인터 타입을 포함
  - pass-by-reference와 pass-by-value의 장점
- Java는 C++의 참조 변수를 확장하여 전체적으로 포인터를 대체하였음
  - 참조는 주소가 있다기 보다는 객체에 대한 참조임
- C#은 Java의 참조와 C++의 포인터를 가짐

Pointer의 평가

- 허상 포인터와 쓰레기에 관한 문제는 힙 관리에 관한 것임
- 포인터는 goto와 유사—변수에 의해 접근될 수 있는 메모리셀의 범위를 확대시킴
- 포인터나 참조변수는 동적 자료구조를 위해 필수적이다 – 따라서 이들 없이 언어를 설계할 수 없음

허상 포인터 문제

- **비석접근방법*(Tombstone)*** : 모든 힙-동적 변수는 비석이라 불리는 특정 셀을 포함
  - 이 셀은 힙-동적 변수에 대한 포인터
  - 실제의 포인터 변수는 단지 비석만을 가리키고 힙-동적 변수를 가리키지 않음
  - 힙-동적 변수가 회수될 때 비석은 **nil**로 설정된 채 남아있음
  - Costly in time and space



비석접근방법 *(Tombstone)*

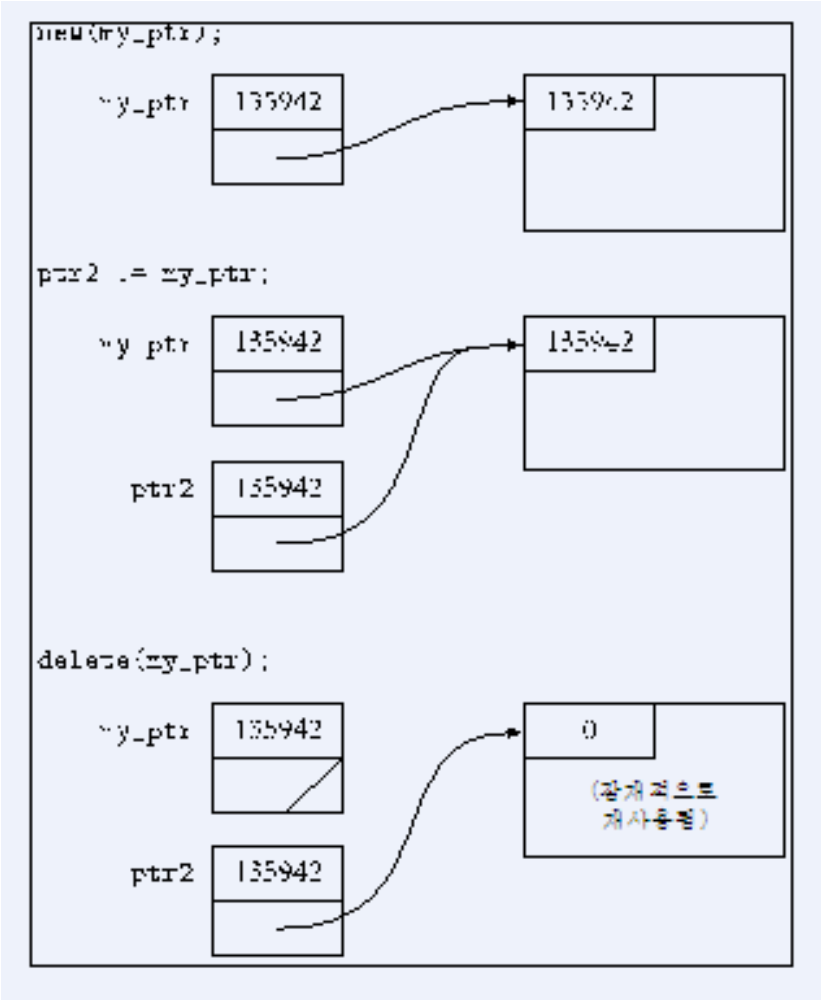
허상 포인터 문제

. 잠금-키 접근 방법(*Locks-and-keys)* : 포인터 값은 순서쌍(키, 주소)으로 표현

- 힙-동적 변수는 그 변수와 정수 잠금 값을 저장하는 헤더 셀을 위한 기억공간으로 표현
- 힙-동적 변수가 할당될 때 잠금 값이 생성되어 힙-동적 변수의 잠금셀과 new의 호출에 명세된 포인터의 키 셀에 모두 저장
- 역참조된 포인터를 접근할 때 이 포인터의 키 값을 힙-동적 변수의 잠금값과 비교하여 일치하면 접근 가능, 불일치이면 실행시간 오류로 처리
- 포인터값의 다른 포인터로의 복사 → 키값도 함께 복사

  → ptr1이 delete 시 잠금셀 값이 0으로 설정

  → ptr2로 역참조시 잠금값과 키값 불일치하므로 접근불가



잠금-키 접근 방법(*Locks-and-keys*)

힙 관리

- 매우 복잡한 실행시간 프로세스
- 단일 크기 셀 vs. 가변 크기 셀
- 쓰레기 수집을 위한 두 가지 방법
  - 참조 계수기(reference counters) : 조기접근방법 (*eager approach*) – 점차적 회수; 접근불가 셀들이 생성될 때 수행
  - 표시-수집(mark-sweep) : 지연접근방법(*lazy approach*) -가용기억공간 리스트가 없을 때 회수

참조 계수기

- 참조 계수기 : 모든 메모리셀에 대해 셀을 현재 가리키고 있는 포인터의 개수를 저장
- 참조계수기의 값이 0이면 셀을 가리키는 포인터가 없음을 뜻하므로 그 셀은 쓰레기
  - *단점* : 공간 소요, 실행시간 소요, 원형으로 연결된 셀의 복잡성
  - *장점* : 본질적으로 점차적임; 응용 프로그램의 동작과 번갈아 수행됨으로 응용프로그램 실행에 지연을 초래하지 않음

표시-수집(Mark-Sweep)

- 런타임 시스템은 요구시 기억공간 셀을 할당하고, 포인터들을 셀들로부터 분리시킨다. 이 과정은 가용기억공간이 모두 할당될 때까지 이루어지다가 표시-수집 프로세스는 힙 공간의 쓰레기를 수집하게 됨
  - 모든 힙 셀은 수집 알고리즘이 사용하는 별도의 비트를 가짐

  (설정단계) 모든 셀은 초기에 쓰레기로 설정

  (표시단계) 모든 포인터들로 힙 공간을 추적하여, 도달 가능한 모든 셀들은 쓰레기가 아닌 것으로 표시

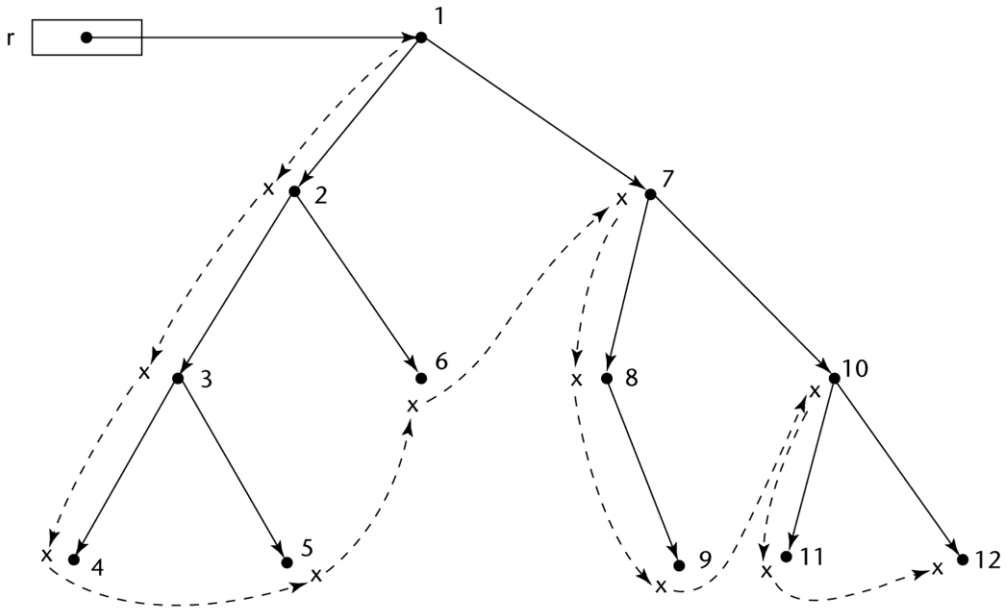
  (수집단계) 모든 쓰레기 셀들은 가용 공간 리스트에 반환

  - 단점 : 이 방법의 처음 버전에서는 너무 빈번한 수행이 문제; 대부분의 셀들을 추적하여 기록해야 하므로 응용 프로그램 실행에 상당한 지연을 초래; 표시-수집 알고리즘을 좀더 자주 수행하게 함으로써 이 문제점 해결 → 점차적 표시-수집 (incremental mark-sweep)이라 부름

Marking Algorithm

```
for every pointer r do
      mark(r)

void mark(void *ptr) {
  if (ptr !=0)
    if (*ptr.marker is not marked){
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
    }
}
```



Dashed lines show the order of node_marking

가변-크기 셀

- 단일-크기 셀의 어려움 이상
- 대부분의 프로그래밍 언어에서 요구됨
- 만약 표시-수집 방법이면 다음 문제들이 추가로 발생
  - 힙 공간에 있는 모든 셀의 지시자에 초기설정의 어려움
  - 표시 과정도 간단치 않음
  - 가용 공간 리스트를 유지하는 오버헤드

타입 검사

- 피연산자와 연산자의 개념을 일반화하여 부프로그램과 배정문을 포함한다.
- *타입검사(Type checking)* 는 피연산자들이 호환가능 타입인지를 확인하는 행위
- 호환가능 타입(*compatible type)*은 연산자에 대하여 적법하거나, 컴파일러 생성 코드에서 적법하게 묵시적으로 변환되는 언어규칙에서 허용되는 타입
  - 이러한 자동 변환을 타입 강제변환(*coercion)*이라 부름.
- 타입 오류(*type error)* 는 부적절한 타입의 피연산자로 연산자 적용

타입 검사 (continued)

- 모든 타입 바인딩이 정적이면 타입검사는 정적으로 수행
- 모든 타입 바인딩이 동적이면 타입검사는 동적이어야 함
- 타입 오류가 항상 탐지된다면 프로그래밍 언어는 강 타입(*strongly typed*)
- 강 타입의 장점 : 타입 오류가 될 변수의 오용을 미리 탐지하도록 해줌

강 타입(Strong Typing)

Language examples:

- C 와 C++ 은 강타입이 아님 : 매개변수 타입 검사 안됨; 공용체도 타입 검사되지 않음
- Java와 C# 은 강타입임 (명시적인 type casting)
- ML 과 F#

강 타입 (continued)

- 타입 강제 변환 법칙(coercion rules)은 타입 검사 기능의 가치에 중요한 영향을 끼침

  → 오류탐지라는 장점을 상실 (C++ versus ML and F#)

- Java는 C++의 배정문 타입 강제변환의 절반 정도를 허용한다. 따라서 타입의 오류탐지는 C++보다 좋으나 ML, F#만큼은 효과적이지 않다.

타입 동등(Type Equivalence)

- 타입 호환성 규칙: 각 연산자의 허용 가능한 피연산자의 타입을 통해서 가능한 타입오류들을 명세
- 식에서 한 타입의 피연산자가 타입 강제변환 없이 다른 타입의 피연산자로 대체될 수 있으면 이 두 개의 타입은 동등(*equivalence*)하다.
- 타입 동등은 타입 호환성의 엄격한 형식(타입 강제변환 없는 호환성)
- 두 변수가 동등 타입이라면 한 변수의 값을 다른 변수에 할당 가능

타입 동등(Type Equivalence)

타입 동등을 정의하는 접근방법

① 이름 동등(name equivalence)
- 함께 선언되거나 또는 동일 식별자 이름으로 선언되면 동일형

② 구조 동등(structural equivalence)
- 구성요소가 모든 측면에서 같으면 동일형

* 선언 동등(declaration equivalence)
- 이름 동등과 구조 동등의 중간을 택함
- 자료형 이름을 재 선언하여 사용 할 경우 동일형 간주

타입 동등(Type Equivalence)

- 예)

```
type T = array [1 .. 100] of integer;
var x, y : array [1 .. 100] of integer;
    z : array [1 .. 100] of integer;
    w : T;
```


규칙 ① : x, y 동등.
w는 x나 z와 다른 형
규칙 ② : x, y, z, w 모두 동등

- 예)

```
type T2 = ...;
type T1 = T2
```


규칙 ① : T1, T2는 다른 형
규칙 ② : T1, T2는 동등
선언동등도 **T1, T2** 동등

- 예)

```
type T1 = array[1 . . 100] of real ;
     T2 = array[1 . . 10, 1 . . 10] of real ;
```


규칙 ② : T1과 T2는 다른 형
(서로 다른 구조임)

이름 타입 동등(Name Type Equivalence)

- **이름 타입 동등**은 두 변수가 동일한 선언문에 속하거나 동일한 타입이름을 갖는 선언문에서 정의되면 두 변수는 동등 타입을 가짐
- 구현하기는 쉬우나 더 제약적임 :
  - 정수형 타입의 부분 범위는 정수형 타입과 동등하지 않음

    (예) **type** Indextype **is** 1..100;

    count : Integer;

    index : Indextype;
  - 형식 매개변수는 그에 대응하는 실 매개변수와 동일한 타입이어야 함; 구조화된 또는 사용자 정의 타입이 매개변수 타입인 경우 단지 한번만 전역적으로 정의되어야 함

구조 타입 동등(Structure Type Equivalence)

- **구조 타입 동등**은 두 개의 변수가 동일한 구조를 갖는다면 두 변수는 동일한 타입을 갖는다는 것을 의미(두 개 타입의 전체구조 비교)
- 좀 더 유연하지만 구현이 더 어려움
  - 두 개의 레코드 타입이 구조적으로는 동일하나 다른 필드 이름을 사용한다면 두 레코드는 동등한가?
  - 두 개의 배열 타입이 첨자만 다를 뿐 동일하다면 두 배열은 동등한가?

    (e.g. `[1..10]` and `[0..9]`)
  - 두 열거형 타입이 그 리터럴의 철자가 다르다면 두 열거형은 동등한가?
  - 구조 타입 동등을 가지고 동일한 구조의 타입들을 구별할 수 없음

타입 동등 (continued)

- C는 이름 타입 동등과 구조 타입 동등 모두 사용
  - 이름 타입 동등 : struct, enum, union 선언문에서
  - 구조 타입 동등 : 다른 비스칼라 타입, 배열
    ※ typedef는 새로운 타입을 생성하는 것이 아니라 기존 타입에 새로운 이름을 정의하는 것임

Summary

- The data types of a language are a large part of what determines that language's style and usefulness
- The primitive data types of most imperative languages include numeric, character, and Boolean types
- The user-defined enumeration and subrange types are convenient and add to the readability and reliability of programs
- Arrays and records are included in most languages
- Pointers are used for addressing flexibility and to control dynamic storage management