

# CONCEPTS OF PROGRAMMING LANGUAGES

## Chapter 5

이름, 바인딩,  
영역

ROBERT W. SEBESTA

12/E

ISBN 0-321-49362-1

# Chapter 5 학습목표

---

- 변수의 속성
- 바인딩의 개념
- 영역
- 영역과 존속기간
- 참조 환경
- 이름상수

# 서론

---

- 명령형 언어는 폰노이만 구조의 추상화
  - 메모리, 프로세서
- 변수의 속성
  - 이름, 주소, 값, 타입, 영역, 존속기간
  - 특히 데이터 타입의 설계는 영역, 존속기간, 타입 검사, 초기화, 타입 호환성 등을 고려해야 함

## ※ 언어의 계열에 관한 지칭

- 언어 버전+ : Fortran 95+는 Fortran 95이후의 모든 버전
- C-기반언어 : C, Objective-C, C++, Java, C#

# 이름

---

- 변수, 함수, 형식매개변수, 클래스의 이름
- 식별자(**identifier**)라고도 함
- 설계 고려사항:
  - 대소문자 구별?
  - 특수어는 예약어 혹은 키워드?

# 이름 (continued)

---

- 길이
  - 너무 짧으면 이름이 함축적일 수 없음
  - 언어의 예:
    - C99: 무제한이나 처음 63자만 의미; 외부 이름은 최대 31자
    - C#, Java: 무제한, 모든 문자 유의미
    - C++: 무제한이나, 구현자가 제한하기도 함

# 이름 (continued)

---

- 첫 문자는 알파벳

(예) 허용하는 문자={alphabet, digit, '-', '\_'}

`<ident> → <letter>{<letter>|<digit>|<특수기호>}*`

- 특수 문자의 사용

- PHP: 모든 변수는 \$로 시작
- Perl: 모든 변수는 특수문자(\$,@,% )로 시작;  
특수문자에 따라 변수의 타입 지정
- Ruby: @로 시작하는 변수는 인스턴스 변수,  
@@로 시작하는 변수는 클래스 변수

# 이름 (continued)

---

- 대소문자 구별(case sensitivity)
  - 단점 : 판독성 저하 (유사해 보이지만 다르다)  
( e.g. rose, ROSE, Rose ... )
  - C언어에서는 변수의 이름을 소문자로 구성
  - C++, java, C#에서 미리 정의된 이름은 대소문자 혼합  
( e.g. IndexOutOfBoundsException)  
→ 작성력 저하

# 이름 (continued)

---

- 특수어

- 판독성에 도움; 문장 절을 구분하거나 분리하기 위해 사용
- 대부분 언어는 특수어 == 예약어(특수어 재정의 X)
- 예약어 : 사용자정의이름으로 사용될 수 없는 특수어
- 예약어의 문제점 : 개수가 너무 많으면 이름 구성하기 어려움 (e.g., COBOL은 300개의 예약어!)



# 변수

---

- 변수는 메모리 셀의 추상화
- 다음 6가지 속성을 가짐 :
  - 이름(name)
  - 주소(address)
  - 값(value)
  - 타입(type)
  - 존속기간(lifetime)
  - 영역(scope)

# 변수의 속성

---

- 이름 : 모든 변수가 이름을 갖는 것은 아님
- 주소 : 변수와 연관된 메모리 주소
  - 변수는 실행 동안 다른 시간에 다른 주소를 갖을 수 있음  
(eg) 함수 호출 시마다 지역 변수의 주소는 다른 주소
  - 두 변수 이름이 동일한 메모리 위치를 참조할 때 이 변수들을 별칭(alias)이라 함
    - 포인터, 참조변수, union(C and C++)을 이용해 생성
    - 판독성 저하(별칭 모두를 기억해야 함)

# 변수의 속성 (continued)

---

- 타입

: 변수의 값 범위 +

타입의 값에 대한 가능한 연산들의 집합

- Java에서 `int` 타입은 값의 범위(?) 와 `+, -, *, /, %` 등의 산술연산을 명세함
- `floating point type`은 정밀도까지 명세함

# 변수의 속성 (continued)

---

- 값

- : 변수와 연관된 위치의 메모리 셀 내용

- 변수의 l-value는 주소, 변수의 r-value는 값

- (예) `count = count + 1;`

- 추상적인 메모리 셀 : 변수와 연관된 셀들의 물리적 모임

- (예) 부동소수점 값이 4개의 물리적 바이트로 구성된다면 한 개의

- 부동소수점 값은 한 개의 추상적인 메모리 셀을 차지

# 바인딩의 개념

---

- 바인딩(binding) : 개체와 속성 간의 연관  
(ex) 변수와 타입 혹은 값 간의 연관  
연산과 기호 간의 연관
- 바인딩 시간(binding time) : 바인딩이 발생하는 시기

# 가능한 바인딩 시간

---

- 언어 설계 시간 - (ex) 연산자 기호와 연산
- 언어 구현 시간 - (ex) floating point 타입과 내부 구현
- 컴파일 시간 - (ex) 변수에 타입 속성 연관
- 적재 시간 - (ex) static 변수와 메모리셀 연관
- 실행 시간 - (ex) nonstatic 지역 변수와 메모리셀 연관

# 바인딩 시간의 예

---

(예) Java의 배정문 count = count + 5;에서

- **count**의 타입 - 컴파일시간
- **count**의 표현가능한 값들의 집합 -- 언어구현시간
- **+**기호의 피연산자에 따른 의미 -- 컴파일시간
- 상수 5의 내부표현 -- 언어구현시간
- **count**의 값 -- 이 배정문의 실행시간

# 정적 바인딩과 동적 바인딩

---

- 정적(**static**) 바인딩 : 프로그램 실행 전에 바인딩이 일어나고, 실행 동안 변경되지 않음
- 동적(**dynamic**) 바인딩 : 프로그램 실행 중 바인딩이 발생하거나, 실행 중 변경될 수 있음



# 타입 바인딩

---

- 타입은 어떻게 명세되는가?
- 타입 바인딩이 일어나는 시기는?
- 정적이면, 명시적 혹은 묵시적인 선언문에 의해 기술될 수 있음

# Explicit/Implicit Declaration

---

- *explicit declaration*(명시적 선언) : 변수의 타입 선언문을 사용
- *implicit declaration*(묵시적 선언) : 선언문 보다는 디폴트 매카니즘을 통해 변수의 타입을 명세
- Basic, Perl, Ruby, JavaScript, PHP는 묵시적 선언을 제공 (Fortran은 명시적/묵시적 선언)
  - 장점 : 작성력(a minor convenience)
  - 단점 : 신뢰성

# Explicit/Implicit Declaration (continued)

---

- Some languages use type inferencing to determine types of variables (context)
  - C# – a variable can be declared with `var` and an initial value. The initial value sets the type
  - Visual Basic 9.0+, ML, Haskell, and F# use type inferencing. The context of the appearance of a variable determines its type

# Dynamic Type Binding

---

- Dynamic Type Binding (JavaScript, Python, Ruby, PHP, and C# (limited))
- Specified through an assignment statement  
e.g., JavaScript

```
list = [2, 4.33, 6, 8];
```

```
list = 17.3;
```

- Advantage: flexibility (generic program units)
- Disadvantages:
  - High cost (dynamic type checking and interpretation)
  - Type error detection by the compiler is difficult

# 변수의 속성 : 존속기간(lifetime)

---

- 기억공간 바인딩
  - 할당(allocation) – getting a cell from some pool of available cells
  - 회수(deallocation) – putting a cell back into the pool
- 변수의 존속기간 : 변수가 특정 메모리 위치에 바인딩 되어있는 기간

# 존속기간에 따른 변수 유형

---

- (1) 정적 변수(static variable)
- (2) 스택-동적 변수(stack-dynamic variable)
- (3) 명시적 힙-동적 변수(explicit heap-dynamic variable)
- (4) 묵시적 힙-동적 변수(implicit heap-dynamic variable)

# 존속기간에 따른 변수 유형(continued)

---

(1) 정적 변수—실행 전에 메모리 셀에 바인딩되고, 실행 기간동안 동일한 메모리셀을 유지

(ex) 함수에서 **static** 변수 (C and C++)

클래스 정의에 포함된 **static** 변수

→ 이 변수는 인스턴스변수가 아닌 클래스변수라 하며  
클래스가 사레화되기 이전에 정적으로 생성됨

(C++, Java, C#)

- **Advantages:** efficiency (direct addressing),  
history-sensitive subprogram support
- **Disadvantage:** lack of flexibility (no recursion)

# 존속기간에 따른 변수 유형(continued)

---

(2) 스택-동적 변수 - 함수 호출되어 함수내 선언문 실행시, 스택에 변수를 할당하면서 기억장소 바인딩됨

- If scalar, all attributes except address are statically bound
  - local variables in C subprograms (not declared `static`) and Java methods
- Advantage: allows recursion; conserves storage
- Disadvantages:
  - Overhead of allocation and deallocation
  - Subprograms cannot be history sensitive
  - Inefficient references (indirect addressing)



# 존속기간에 따른 변수 유형(continued)

---

**(3) 명시적 힙-동적 변수** - 프로그래머가 명시하는  
명시적 실행시간 명령어에 의해 할당/회수되는 이름이  
없는 메모리 셀

- 힙에 할당/회수
- 포인터나 참조변수를 통해 참조 가능

(ex) dynamic objects in C++ (via `new` and `delete`),  
all objects in Java

- Advantage: provides for dynamic storage management
- Disadvantage: inefficient and unreliable

# 존속기간에 따른 변수 유형(continued)

---

## (4) 묵시적 힙-동적 변수 — 배정문에 의해 힙에

할당과 반환이 됨

- all variables in APL; all strings and arrays in Perl, JavaScript, and PHP
- Advantage: flexibility (generic code)
- Disadvantages:
  - Inefficient, because all attributes are dynamic
  - Loss of error detection
  - Perl, JavaScript, PHP

# 변수의 속성 : 영역

---

- **영역(scope)** : 변수의 가시적인(visible) 문장들의 범위  
(※ 변수가 어떤 문장에서 참조가능하면 변수는 그 문장에서 “가시적”임)
  - 지역변수 : 프로그램 단위/블록 내부에서 선언
  - 비지역 변수 : 프로그램 단위에서 선언되지 않으나 가시적인 변수
  - 전역변수 : 비지역 변수의 특수한 부류
- **영역 규칙(scope rule)** :
  - 이름의 참조가 어떤 변수들과 연관되는지 결정하는 것
  - 비지역변수에 대한 참조와 선언과의 연관을 규정
  - 정적영역규칙과 동적영역규칙

# 정적 영역규칙(static scoping): 비지역 변수에 이름을 바인딩하는 방법(ALGOL60)

---

- 프로그램 텍스트에 기반
  - 변수 영역을 실행 전에 결정
  - 이름 참조를 변수와 연관시키기 위해서, 해당 선언문을 찾는다.
- 탐색과정:
  1. x를 지역 선언문에서 찾는다.
  2. x가 발견되지 않으면 현재 영역을 포함한 더 큰(가장 가까운) 영역의 선언문에서 x를 찾는다.
  3. 2의 과정을 x가 발견될 때까지 반복
  4. 가장 큰 영역에서도 x가 발견되지 않으면 오류
- 정적조상(static ancestors): 현재 영역을 포함한 더 큰 영역
- 정적부모(static parent) : 현재 영역에서 가장 가까운 정적조상

# 정적 영역규칙

- JavaScript 함수 `big()`을 고려해보자
  - 1) `sub2`의 변수 `x`에 대한 참조는?
    - 정적 부모(`sub2`를 선언한 함수)는 `big`이므로 `big`의 `x`
    - `sub1`은 `sub2`의 정적 조상 아님
  - 2) `sub1`외부에 선언된 `x`는 `sub1`으로부터 은폐

```
function big() {  
  function sub1() {  
    var x=7;  
    sub2();  
  }  
  function sub2() {  
    var y=x;  
  }  
  var x=3;  
  sub1();  
}
```

# Blocks

---

- 프로그램단위 내부에 정적 영역을 생성하는 방법--from ALGOL 60
- Example in C:

```
void sub() {  
    int count;    // while 루프의 코드로부터 은폐됨  
    while (...) {  
        int count;  
        count++;  
        ...  
    }  
    ...  
}
```

- Note: legal in C and C++,  
but not in Java and C# - too error-prone

# The LET Construct

---

- 대부분의 함수형 언어들은 **let** 구조 형식을 포함하고 있음
- A let construct has two parts
  - The first part binds names to values
  - The second part uses the names defined in the first part
- In Scheme:

```
(LET (  
  (name1 expression1)  
  ...  
  (namen expressionn)  
  expression  
)
```

(ex)

```
(LET (  
  (top (+ a b))  
  (bottom (- c d))  
  (/ top bottom)  
)
```

# The `LET` Construct (continued)

---

- In ML:

```
let
  val name1 = expression1
  ...
  val namen = expressionn
in
  expression
end;
```

(ex)

```
let
  val top = a + b
  val bottom = c - d
in
  top/bottom
end;
```



# 선언 순서

---

- C99, C++, Java, C#에서는 문장의 어느 곳에서든 변수 선언문을 허용
  - C99, C++, and Java : 모든 지역 변수의 영역은 선언문부터 블록 끝까지 임
  - C# : 블록에 선언된 모든 변수의 영역은 선언문의 위치에 상관없이 블록 전체임
    - 그러나 변수는 참조되기 이전에 반드시 선언이 되어 있어야 함
- C++, Java, C#에서 변수들은 `for` 문장에서 선언 가능
  - 그 변수들의 영역은 `for` 구조 내로 제한됨

# 전역 영역

---

- (C, C++, PHP, Python) 파일 내에 일련의 함수 정의들로 구성되는 프로그램 구조 지원
  - 함수정의 외부에서 변수 선언이 가능, 이들 변수들은 함수들에게 가시적임
- C, C++는 전역 변수의 선언과 정의를 갖음; 선언은 속성을 명세하고, 정의는 속성 명세 및 기억공간 할당을 야기함
  - 함수 정의 외부에 위치한 변수 선언은 그 변수가 다른 파일에 정의되어 있음을 명세  
(ex) `extern int sum;`

# 전역 영역 (continued)

---

- PHP

- 프로그램은 HTML markup documents에 함수 정의들과 섞여 내장되어 있음
- 함수 내에서 묵시적으로 선언된 변수의 영역은 함수 내에 지역적이다.
- 함수 외부에서 묵시적으로 선언된 변수의 영역은 그 선언문으로부터 프로그램 끝까지이나 그 이후에 오는 함수 정의들에 대해서는 건너뛴다.
- 전역 변수를 함수에서 참조하려면(예제 p243)

# 전역 영역 (continued)

---

- 전역 변수를 함수에서 참조하려면(예제 p243)

1) 동일한 이름의 지역변수가 있다면, `$GLOBALS` 배열에 그 전역변수 이름을 첨자로 이용함

2) 동일한 이름의 지역변수가 없다면, `global` 선언문에 변수를 포함시킴

```
(ex) $day = "Monday";  
$month = "January";  
function calendar() {  
    $day = "Tuesday";  
    global $month;  
    print "local day is $day <br />";  
    $gday = $GLOBALS['day'];  
    print "global day is $gday <br />";  
    print "global month is $month <br />";  
}  
calendar();
```

# 전역 영역 (continued)

## • Python

- 전역변수는 함수에서 참조가능하나, 함수에서 `global`로 선언해야 대입 가능함
- (예제)

```
day = "Monday"
def tester():
    print "The global day is: ", day
tester()
```

```
day = "Monday"
def tester():
    print "The global day is: ", day
    day = "Tuesday"
    print "The new value of day is: ", day
tester()
```

```
day = "Monday"
def tester():
    global day
    print "The global day is: ", day
    day = "Tuesday"
    print "The new value of day is: ", day
tester()
```

# 정적 영역의 평가

---

- Works well in many situations
- Problems:
  - In most cases, too much access is possible
  - As a program evolves, the initial structure is destroyed and local variables often become global; subprograms also gravitate toward become global, rather than nested

# 동적 영역규칙(dynamic scoping)

---

- 부프로그램들의 상호간의 공간적 배치관계가 아닌 부프로그램들의 호출 시퀀스에 기반
- 그 지역 내 선언문 탐색을 하여 실패이면 동적 부모, 즉 호출 함수의 선언문이 탐색대상이 되며 선언문이 발견될 때까지 그 동적 부모에 대해 탐색은 계속됨

# Scope Example

---

```
function big() {  
  function sub1() {  
    var x = 7;  
  }  
  function sub2() {  
    var y = x;  
  }  
  var x = 3;  
}
```

big **calls** sub1  
sub1 **calls** sub2  
sub2 **uses** x

- 정적 영역
  - Sub2의 x 참조는 big의 x
- 동적 영역
  - Sub2의 x 참조는 sub1의 x

(ex) big calls sub2 일때 sub2의 x는?



# 동적 영역의 평가

---

- Evaluation of Dynamic Scoping:
  - Advantage: convenience
  - *Disadvantages:*
    1. While a subprogram is executing, its variables are visible to all subprograms it calls
    2. Impossible to statically type check
    3. Poor readability– it is not possible to statically determine the type of a variable

# 영역과 존속기간

---

- 변수의 영역과 존속기간은 밀접한 관련이 있는 듯하지만 다른 개념임
- 정적 영역은 공간적 개념, 존속기간은 시간적 개념  
(ex) a **static** variable in a C or C++ function

# 영역과 존속기간

---

- 영역과 존속기간은 부프로그램 호출시에는 무관  
(ex)

```
void printhead() {  
    ...  
}  
void compute() {  
    int sum;  
    ...  
    printhead();  
}  
compute();
```

# 참조 환경(referencing environment)

- 어떤 문장의 **참조환경**은 그 문장에서 가시적인 모든 이름들의 집합
- 정적-영역 언어에서 참조환경은 그 지역 변수와 그 조상영역에 속한 가시적인 변수들로 구성
- Python의 예

```
g = 3;  # 전역변수
def sub1():
    a = 5;
    b = 7;
    ...           ← 1
def sub2():
    global g;  # 전역변수 g는 이제 여기서 배정가능함
    c = 9;
    ...           ← 2
def sub3():
    nonlocal c;  #비지역변수 c를 여기서 가시적이게 한다
    g = 11;      # 새로운 지역변수 생성
    ...           ← 3
```

# 참조 환경(referencing environment)

---

- 부프로그램은 실행이 시작되었으나 종료되지 않았다면 활성화되어 있음
- 동적-영역 언어에서 참조환경은 지역변수와 모든 활성화된 부프로그램의 모든 가시적인 변수들로 구성

```
void sub1() {  
    int a, b;  
    ...           ← 1  
}  
void sub2() {  
    int b, c;  
    ...           ← 2  
    sub1();  
}  
void main() {  
    int c, d;  
    ...           ← 3  
    sub2();  
}
```

# 이름 상수(Named Constants)

---

- A *named constant* is a variable that is bound to a value only when it is bound to storage
- Advantages: readability and modifiability
- Used to parameterize programs
- The binding of values to named constants can be either static (called *manifest constants*) or dynamic
- Languages:
  - C++ and Java: expressions of any kind, dynamically bound
  - C# has two kinds, `readonly` and `const`
    - the values of `const` named constants are bound at compile time
    - The values of `readonly` named constants are dynamically bound