Chapter 5
기본 의미론

속성, 바인딩, 그리고 의미함수들
선언, 블록, 영역, 그리고 심볼 테이블
이름 식별 (name resolution) 그리고 다중적재 (overloading)
할당, 수명, 그리고 환경
변수와 상수

소개

- 구문 상의 명세: BNF, 신택스 다이어그램 등등
- 의미 상의 명세
  - 언어 참조 매뉴얼
    - 자연어가 사용됨
    - 엄밀하게 기술하기 어려우나, 이해는 쉬움
  - 번역기를 직접 만듦 (정의적 번역기에 의한 방법)
    - 의미론 부분은 프로그램을 실행해보면 알 수 있음
    - 기계 종속적
  - 형식적 정의 (Formal definition)
    - 엄밀하지만 읽기 어려움 (13 장 참조)
- "언어 참조 매뉴얼" 접근 방법을 사용하자

속성

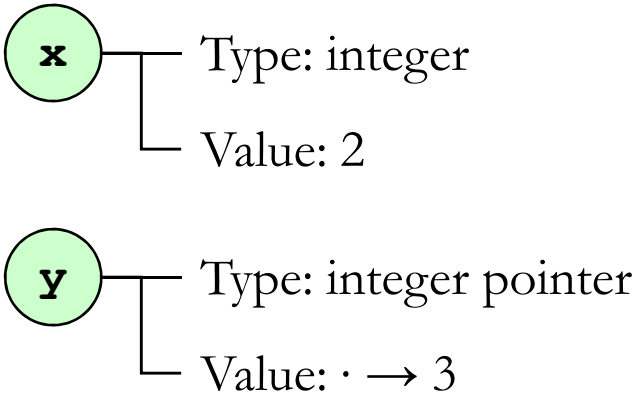
- 속성
  - 언어 개체의 특성인데, 주로 식별자의 특성
  - 다른 언어 개체들의 경우 (예: 연산자 심볼), 특성이 이미 지정되어 있음
- 속성의 예
  - 변수의 위치
  - 표현식의 값
  - 식별자의 타입: 가능한 연산이 뭔지를 알게 함
  - 프로시저(procedure) 코드: 프로시저의 연산
  - 특정 타입의 크기: 값의 범위를 알 수 있음

바인딩(Binding) ← 중요

- 바인딩
  - 이름에 속성을 연관시키는 과정
  - 선언이나 정의는 속성을 식별자에 연관시킴
- 바인딩의 예

```
// C++ example

int x, *y;
x = 2;
y = new int(3);
```



바인딩 타임

- 바인딩이 발생하는 시간
- 바인딩 시간을 크게 나누자면
  - 정적 바인딩 (Static binding): 프로그램 실행 전에 발생
  - 동적 바인딩 (Dynamic binding): 프로그램 실행 중에 발생
- 좀 더 세분화해보면
  - 언어 정의 시간 (Language definition time)
  - 언어 구현 시간 (Language implementation time)
  - 프로그램 번역 시간 (Program translation time)
  - 링크 시간 (Link time)
  - 메모리에 로드하는 시간 (Load time)
  - 실행 시간 (Execution time (run time))

바인딩 타임의 예

- 표현식의 값
  - 변수를 포함하고 있으면 실행 시간
  - 상수들만 가지고 있으면 번역 시간
- 식별자의 타입
  - 컴파일러 시스템의 컴파일 시간 (예, Java)
  - 인터프리터의 실행 시간 (예, LISP)
- 정수(integer)가 가지는 숫자의 최대 개수
  - 언어의 정의 시간 (예, Java)
  - 언어의 구현 시간 (예, C)
- 변수의 위치
  - 정적 변수는 프로그램이 로드되는 시간 (**`static`** in C)
  - 자동 변수는 실행 시간 (**`auto`** in C)

심볼 테이블과 환경

- 심볼 테이블 (← 중요)
  - 언어 번역기(컴파일러 또는 인터프리터)가 가지고 있는 자료구조로 바인딩 정보를 가지고 있음
  - 수학적으로 보자면 함수

    심볼테이블 : 이름 → 속성
- 컴파일러 시스템
  - 속성들은 여러 개의 함수로 처리됨

    심볼테이블 : 이름 → 정적인 속성

    환경 : 이름 → 위치

    메모리 : 위치 → 값
- 인터프리터 시스템
  - 심볼테이블과 환경이 조합되어 있음

    환경: 이름 → 속성 (위치와 값을 가지고 있음)

선언

- 바인딩을 구성하는 주요한 방법
  - 암시적 바인딩: 선언에서 암시적으로 가정됨
  - 명시적 바인딩: 선언에서 직접적으로 명시됨
  - 예) 다음의 C 선언에서,

    ```
    int x;
    ```

    - 타입은 명시적 바인딩
    - 변수의 메모리 위치는 암시적 바인딩
- 암시적 선언
  - 변수의 이름 짓는 방식이 바인딩을 결정함
  - 예)
    - FORTRAN: I, J, K, L, M, 또는 N 으로 시작하는 변수는 정수
    - BASIC: 변수 뒤에 %가 붙으면 정수, $는 스트링

선언을 가리키는 다른 용어들

- 어떤 언어들은 정의와 선언을 구분함
  - 정의: 가능한 모든 속성들을 바인드하는 선언
  - 선언: 일부 속성만 바인드하는 선언
- C 언어 예 ← 중요
  - 함수 선언(즉 원형)은 함수의 타입만 바인드함
  - 외부 변수 선언(extern)은 위치 속성을 바인드하지 않음
  - 서로 재귀적은 타입을 정의하기 위해 불완전한 타입 선언을 사용함

C 선언의 예 (중요!)

- **`int x = 0;`**
  - 데이터 타입과 초기값을 명확히 지정함
  - 암시적으로는 영역(후에 서술)와 메모리의 위치를 지정함
- **`int f(double);`**
  - 명시적으로 타입을 지정함 (**`double`** → **`int`**)
  - 그러나 그 이외의 것은 전혀 지정 안함: 즉 뒤에서 함수의 실제 코드를 지정해야 함
- 앞의 것은 C에서 **정의(definition)**이고, 뒤의 것은 선언이다.

블록과 지역성

- 블록 (Block)
  - 선언을 가지고 있는 표준적인 언어 구성물
  - 할당하는 최소 단위
- 선언이나 참조의 지역성 (Locality)
  - 지역적: 선언과 참조가 같은 블록에 있음
  - 비지역적(non-local): 선언이 참조가 있는 블록이 아닌 다른 블록에 있음
- 따라서, 비지역적인 참조에 상응하는 선언을 찾기 위해 나름의 규칙이 요구됨

블록 구조

- 블록 구조 프로그램
  - 프로그램은 블록으로 구성되어 있음, 이 블록들은 중첨될 수 있음
  - 대부분의 Algol의 영향을 받은 언어들은 이 구조를 이용함
- 블록의 종류
  - 프로시저 블록: Pascal
  - 프로시저가 아닌 블록: Ada, C

```
-- Ada example

declare x: integer;
begin
  …
end
```

```
(* Pascal example *)

program ex;
var x: integer
begin
  …
end
```

선언을 가지고 있는 다른 것들

- 구조적인 데이터 타입
  - 선언을 내재하고 있는 (즉 안에 가지고 있는) 선언들
  - 예: struct in C, class in C++, Java, Smalltalk
- 모듈 또는 패키지
  - 선언들의 집합
  - 그 자체는 선언이 아님
  - 예)
    - Ada: package and task
    - Java: package
    - ML and Haskell: module
    - C++: namespace

영역

- 바인딩의 영역
  - 프로그램에서 바인딩이 유지되는 영역
- 이름의 영역
  - 같은 이름이 여러 번 선언될 수 있음
  - 같은 이름을 서로 다르게 선언하는 것들이 제대로 구별되어야 함
- 영역과 블록
  - 사용하기 전에 선언하라는 "사용 전 선언" 규칙: 일반적으로 영역는 선언이 포함된 블록의 마지막까지이다.
  - 어떤 언어 구성물에서는, 영역이 블록의 시작 부분까지 역으로 확장되기도 함 (classes in Java and C++, Scheme의 최상위 레벨 선언)

영역 규칙

- 렉시컬 (정적 영역) 규칙
  - 정적 영역 규칙에서 바인딩의 영역는 상응하는 선언이 포함된 블록의 내부이다
  - 정적 영역 규칙은 대부분의 블록 구조 언어의 표준 영역 규칙이다
- 동적 영역 규칙
  - 바인딩의 영역이 실행 경로에 의해 결정됨
  - 심볼 테이블이나 환경이 동적으로 관리되야 함

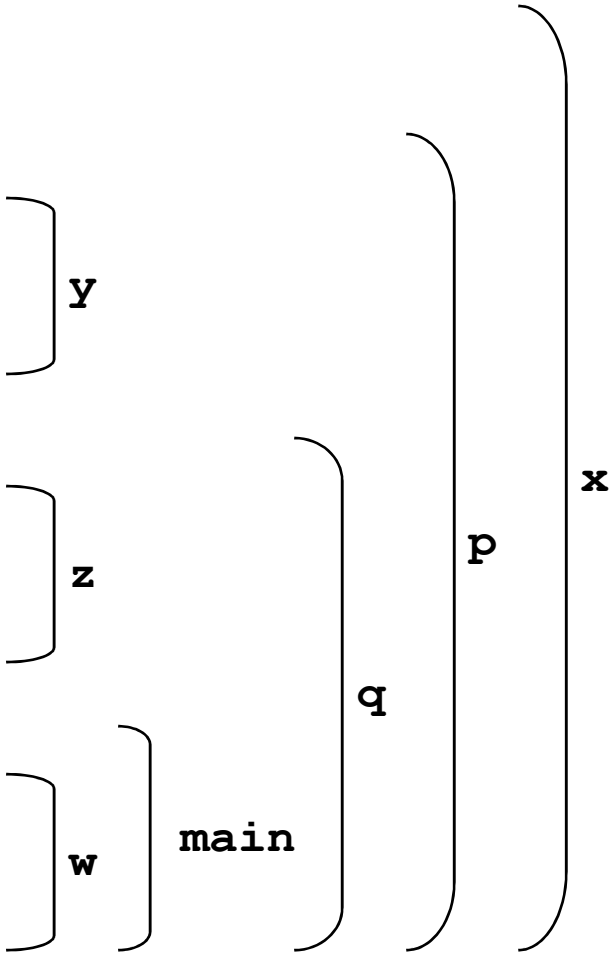
렉시컬 영역의 예 (C)

```
int x;

void p(void)
{ char y;
  ...
} /* p */

void q(void)
{ double z;
  ...
} /* q */

main()
{ int w[10];
  ...
}
```



C에선, 사용 전 선언 규칙이 적용됨

영역 구멍

- 영역 구멍이란?
  - 특정 블록의 지역적인 이름의 선언이 (외부 블록의) 과거의 동일한 이름 선언을 가려 버림
  - 이런 경우, 가려진 선언이 그 블록에 대해 영역 구멍이 있다고 함
- 가시성과 영역
  - 가시성: 선언된 이름이 보이는 지역 (영역 구멍이 제외됨)
  - 영역: 바인딩이 존재하는 지역 (영역 구멍이 포함됨)

영역 해결 연산자

```
// C++ example

int x;

void p(void)
{ char x;
  x = 'a';  // local x
  ::x = 42; // global x
  ...
} /* p */

main()
{ x = 2;    // global x
  ...
}
```

- 전역 정수 변수 **x** 는 함수 **p** 내부에서 영역 구멍을 가짐.
- C 에서, 전역 변수 **x** 는 **p** 안에서 참조 될 수 없음
- C++에서 , 전역 변수 **x** 는 영역 지정 연산자 **::** 에 의해 참조될 수 있음
- Ada 도 영역 지정 연산자 **.** 를 가지고 있음

C에서의 파일 영역 (중요!)

- 파일 영역
  - C에서, 전역 변수는 **static** 키워드를 앞에 붙이면, 파일 영역으로 바뀜
  - 파일 영역인 이름은 그 파일 내에서만 참조 가능
- 예

File 1

```
extern int x;
...
x
...
```

File 2

```
int x;
...
x
...
```

File 3

```
static int x;
...
x
...
```

순환적 선언 또는 자기 참조적 (self-referential) 선언

- 재귀적으로 함수를 선언하는 것은 잘 정의되어 있음

```
int factorial(int n) {
    ... factorial(n - 1) ...
}
```

- 변수는 어떨까?

```
int x = x + 1;
```

  - Ada 나 Java 에서는 허용되지 않음
  - C/C++에서는 지역 변수에 대해서는 허용되지만 의미 없음
- 사용하기 전에 선언하라는 규칙 안에서, 상호 재귀를 다루는 방식
  - 파스칼은 선행 참조 (forward declarations)
  - C/C++에서는 원형 선언

상호 재귀의 예 (C 언어)

```
#include <stdio.h>
void q(int);
void p(int x)
{
  if (x<0) return;
  printf("%d₩n", x);
  q(x);
}
void q(int x)
{
  if (x<0) return;
  printf("%d₩n", x);
  p(x-1);
}
void main()
{
  p(5);
}
```

자바 영역의 예

```
public class Scope
{ public static void f()
  { System.out.println(x); }
  public static void main(String[] args)
  { int x = 3;
    f();
  }
  public static int x = 2;
}
```

자바 클래스에서는 사용 전 선언 규칙이 적용되지 않음

- 클래스 정의에서, 선언의 영역은 전체 클래스임. 밑줄 그은 선언에 주의할 것
- 영역 규칙에 따라 결과가 다를 수 있음
  - 위의 코드는 2를 출력함 (자바는 정적 영역 규칙을 따름)
  - 동적 영역 규칙에 따르면 위의 코드는 3를 출력함

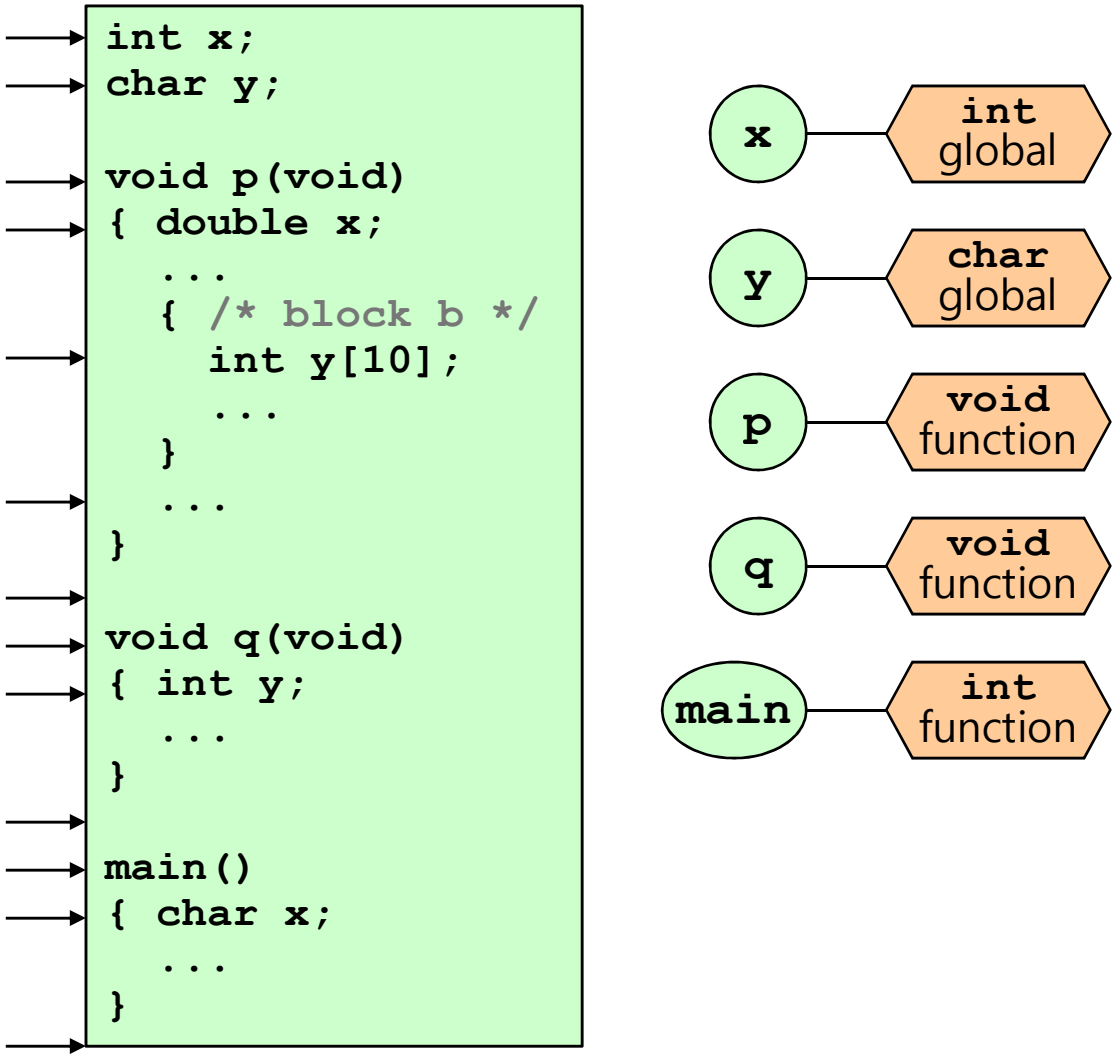
동적 영역 기법에 대한 평가

- 단점
  - 선언의 영역이 정적으로 결정되지 않음 (손으로 시뮬레이션해야 함)
  - 식별자의 타입이 정적으로 결정되지 않음 (정적인 형 체크가 불가능함)
- Historical Note
  - 최초에는 리스프(LISP)에서 사용됨. Scheme에서는 여전히 사용될 수 있지만, 사용되지 않고 있음. 어떤 언어들은 여전히 사용함 : VBScript, Javascript, Perl (옛날 버전들).
  - LISP 발명자인 맥카시는 이제 이것을 버그라고 함

심볼 테이블 유지/보수

- 렉시컬 영역 언어들 (정적 영역 언어들)
  - 심볼 테이블이 스택처럼 유지됨
  - 심볼 테이블이 실행 패스에 무관하게 정적으로 유지됨
- 동적 영역 언어들
  - 가장 밖에 있는 이름들에 대한 모든 바인딩들이 구성됨
  - 바인딩은 실행 패스에 따라서 유지됨

렉시컬 영역에서의 심볼 테이블

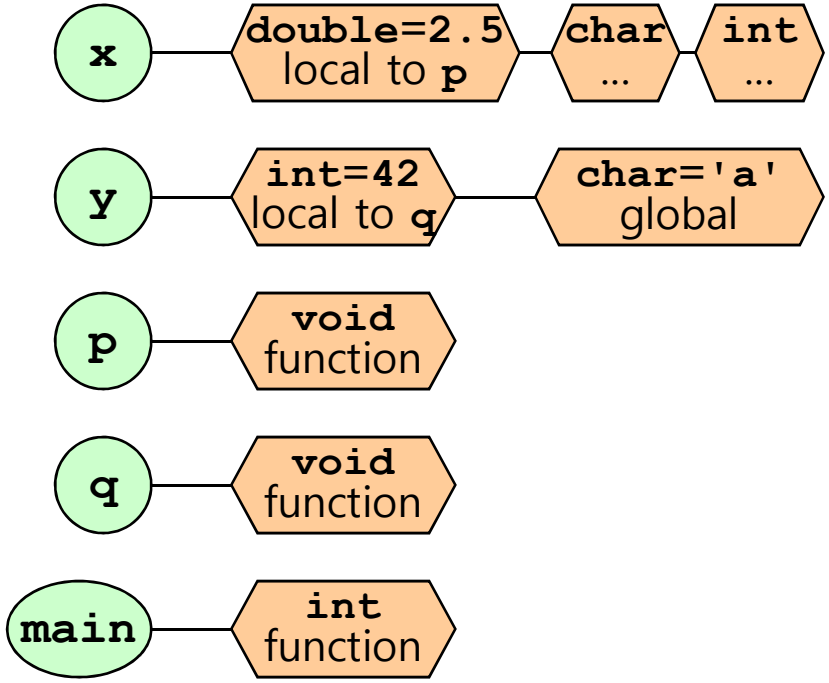


동적 영역에서의 심볼 테이블

```
#include <stdio.h>

int x = 1;
char y = 'a';

void p(void)
{ double x = 2.5;
  printf("%c\n",y)
  { /* block b */
    int y[10];
  }
}
void q(void)
{ int y = 42;
  printf("%d\n",x);
  p();
}
main()
{ char x = 'b';
  q();
  return 0;
}
```



Output

```
98
*
```

ASCII 42 = 'b'

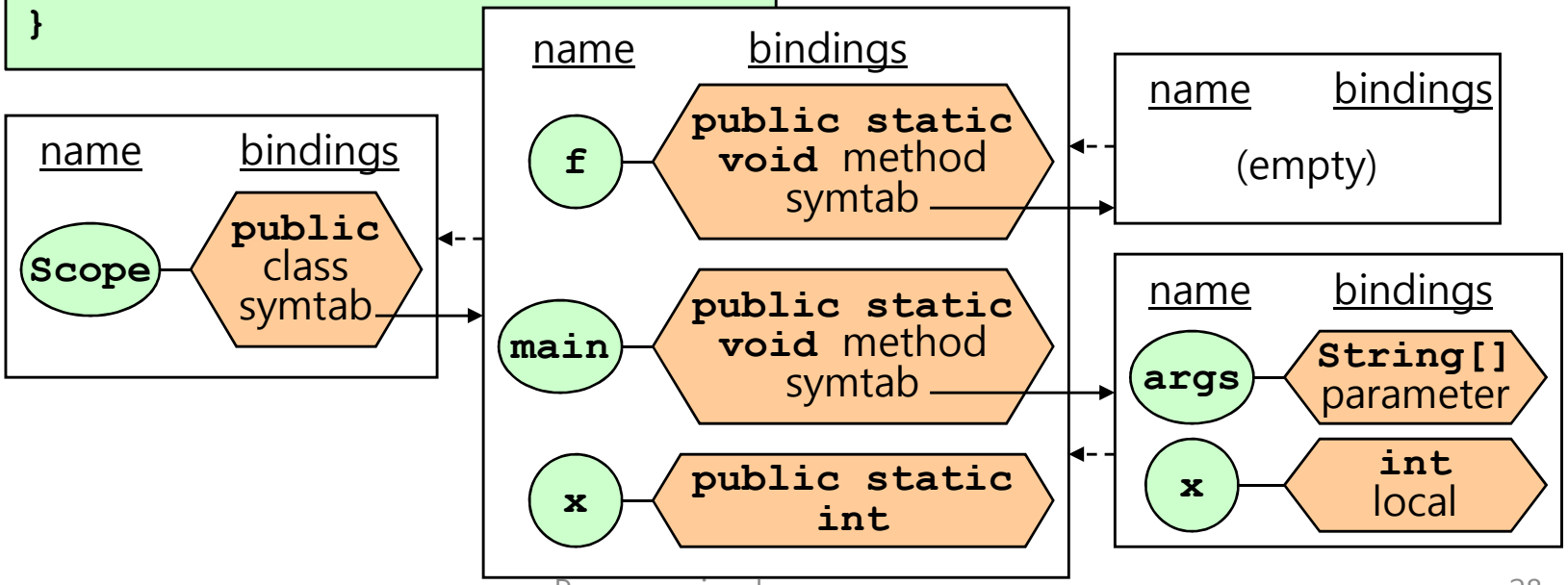
중첩된 심볼 테이블

- 영역 구조
  - 어떤 언어 구성물들은 자신들만의 이름 공간을 가지도록 허용됨
  - 예
    - Pascal: 함수, 프로시저
    - C++: 클래스, 함수, 구조체(struct), 이름공간
    - Java: 클래스, 메쏘드, 패키지
- 심볼 테이블의 스택
  - 언어가 영역 구조를 가진다면, 심볼 테이블의 스택을 구성하는 방법이 편리함

중첩된 심볼 테이블의 예

```
public class Scope
{ public static void f()
  { System.out.println(x); }
  public static void
   main(String[] args)
  { int x = 3;
    f();
  }
  public static int x = 2;
}
```

- 렉시컬 영역에서는, 왼쪽 코드의 심볼 테이블은 다음과 같음



다중적재

- 무엇이 다중적재인가?
  - 같은 영역 내에서 서로 다른 개체(entity)들에 대해 같은 이름(name)을 재사용함

    $entity_1/name_1$, $entity_2/name_2$

    $(entity_1, entity_2)/\underline{name}$
    - 위의 경우, name이 다중적재되어 있음
  - 연산자 다중적재, 함수 다중적재
- 다중적재 식별
  - 다중적재된 이름에 대해 주어진 사용방법을 통해 적절한 개체를 선택함
  - 일반적으로 호출 컨텍스트(호출에 내재된 정보들)를 사용하여 다중적재를 식별함

다중적재의 예

- 대부분의 언어에서, +연산자는 다중적재됨
  - 정수 덧셈 (say, **`ADDI`**)
  - 부동 소수점 덧셈(say, **`ADDF`**)
- 모호함을 해결하는 실마리: 피연산자 (operand)의 타입
- 다음과 같이 타입이 섞인 경우는?

```
2 + 3.2
```

  - C/C++ 에서는 프로모션됨 (암시적 형 변환)
  - Ada에서는 에러로 간주

함수 다중적재

```
int max(int x, int y)              // max #1
{ return x > y ? x : y; }

double max(double x, double y) // max #2
{ return x > y ? x : y; }

int max(int x, int y, int z)    // max #3
{ return x > y ? (x > z ? x : z) : (y > z ? y : z); }
```

- 이름 식별

`max(2,3)`	calls max #1
`max(2.1,3.2)`	calls max #2
`max(1,3,2)`	calls max #3

다중적재 해결 문제

- 암시적 변환은 모호한 호출을 초래함

```
max(2.1, 3)
```

  - C++: 호출할 후보가 너무 많음 (max #1 or max #2)
  - Ada: 호출한 후보가 없음
  - Java: 암시적 변환은 정보 손실이 없는 경우에만 사용됨
- 호출 컨텍스트에 반환 형도 포함되는가?
  - C++, Java: 포함되지 않음
  - Ada: 포함됨

Ada에서의 함수 다중적재

```
procedure overload is

function max(x: integer; y: integer) -- max #1
   return integer is
begin
   ...
end max;

function max(x: integer; y: integer) -- max #2
   return float is
begin
   ...
end max;

a: integer;
b: float;
begin -- max_test
   a := max(2,3); -- call to max # 1
   b := max(2,3); -- call to max # 2
end overload;
```

Ada에서는, 반환 형도 호출 컨텍스트에 포함됨

C++에서의 함수 다중적재

```
#include <iostream>
using namespace std;
typedef struct { int i; double d; } IntDouble;

bool operator < (IntDouble x, IntDouble y)
{ return x.i < y.i && x.d < y.d; }

IntDouble operator + (IntDouble x, IntDouble y)
{ IntDouble z;
  z.i = x.i + y.i;
  z.d = x.d + y.d;
  return z;
}

int main()
{ IntDouble x = {1,2.1}, y = {5,3.4};
  if (x < y) x = x + y;
  else y = x + y;
  cout << x.i << " " << x.d << endl;
  return 0;
}
```

Ada 에서의 함수 다중적재

```
procedure opover is

type IntDouble is
record
  i: Integer;
  d: Float;
end record;

function "<" (x,y: IntDouble) return Boolean is ...

function "+" (x,y: IntDouble) return IntDouble is ...

x, y: IntDouble;
begin
  x := (1,2.1);
  y := (5,3.4);
  if (x < y) then x := x + y;
  else y := x + y;
  end if;
  put(x.i); put(" "); put(x.d); new_line;
end opover;
```

연산자 다중적재의 알아둘 점

- 연산자 오버로딩으로도 결합성이나 우선순위같은 문법적인 특성은 변치 않음
- 연산자의 prefix 형태를 위해 특별한 표현이 사용됨

  `x + y`

  – Ada의 prefix 형태 : **`"+"(x, y)`**

  – C++의 prefix 형태: **`operator + (x, y)`**

다른 형태의 이름 재사용

- 서로 다른 개체에 대해 이름을 재사용
  - 각각의 개체 종류에 따라 서로 분리된 이름 공간 필요
  - 이런 재사용은 다중적재는 아님

C example

```
typedef struct A A;
struct A
{   int data;
    A * next;
};
```

structure tag name

type name

Java example

```
class A
{ A A(A A)
  { A:
    for(;;)
    { if (A.A(A) == A)
        break A;
    }
    return A;
  }
}
```

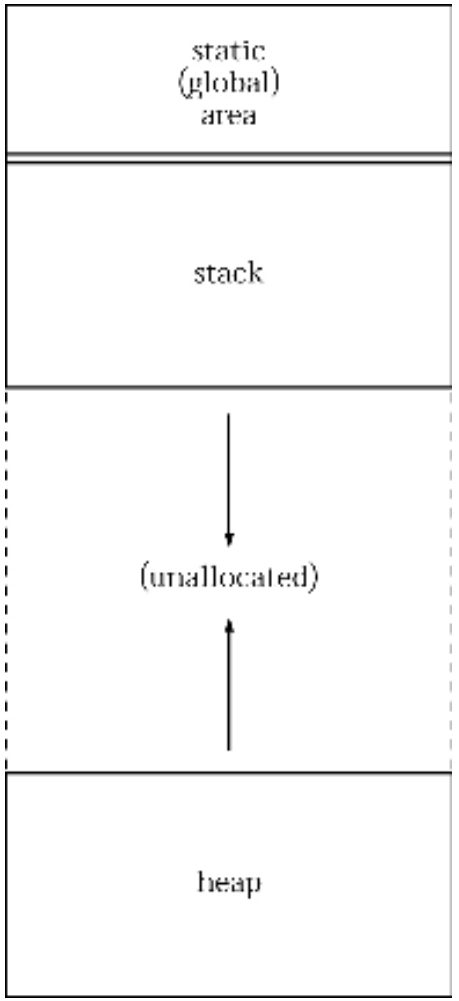
Which is which?
- class name
- method name
- parameter name
- label name

환경

- 환경 구성 시간
  - 정적인 환경: FORTRAN
  - 동적인 환경: LISP
  - 혼합: 대부분의 알골 스타일(Algol-style) 언어들
- Algol-style 언어들은 다양한 할당 시간을 가짐
  - 전역 변수들
    - 정적인 할당
    - 메모리에 로드될 때 할당됨
  - 지역 변수들
    - 대부분 동적 할당
    - 선언이 실행될 때, 할당됨 (즉, 프로그램의 제어가 그 선언을 지나칠 때 할당됨)

전형적인 환경

- 전형적인 알골 스타일 언어의 구성요소들
  - 정적 할당을 위한 스택
  - LIFO 스타일 동적 할당을 위한 스택
  - 온 디맨드 동적 할당을 위한 힙



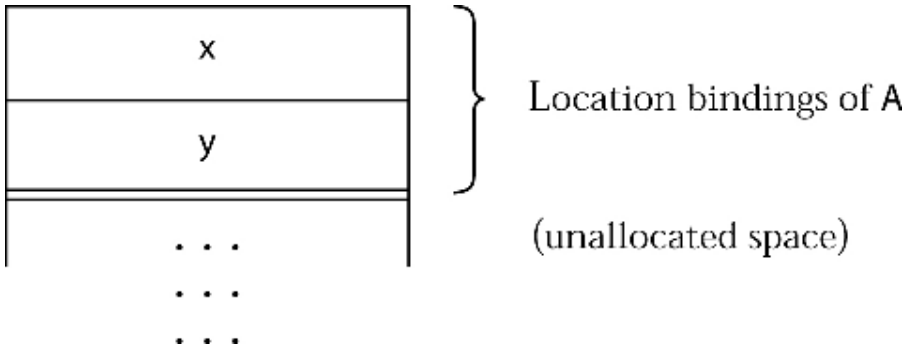
활성 레코드(Activation Record)

- 활성(Activation)
  - 서브프로그램의 호출
  - 각 활성마다 서브프로그램의 환경이 구성됨
- 활성 레코드(Activation Record)
  - 활성을 위해 할당되는 메모리 영역
  - 서브프로그램 환경 + 회계 정보
- 실행 시간 스택(Run-Time Stack)
  - 블록에 들어가고 나오는 방식은 LIFO-style
  - 프로시저 호출 및 반환은 LIFO-style
  - 활성 레코드가 저장되는 곳은 실행 시간 스택

실행 시간 스택 조작

```
A: {   int x;
       char y;
   B: {   double x;
          int a;
       } /* end B */
   C: {   char y;
          int b;
          D: {   int x;
                 double y;
             } /* end D */
          } /* end C */
      } /* end A */
```

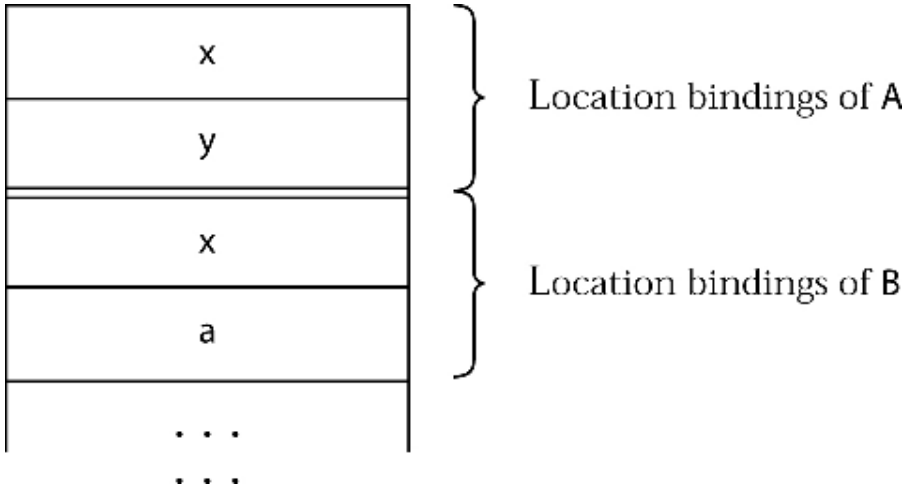
point 1



실행 시간 스택 조작

```
A: {   int x;
       char y;
   B: {   double x;
          int a;
       } /* end B */
   C: {   char y;
          int b;
          D: {   int x;
                 double y;
             } /* end D */
          } /* end C */
      } /* end A */
```

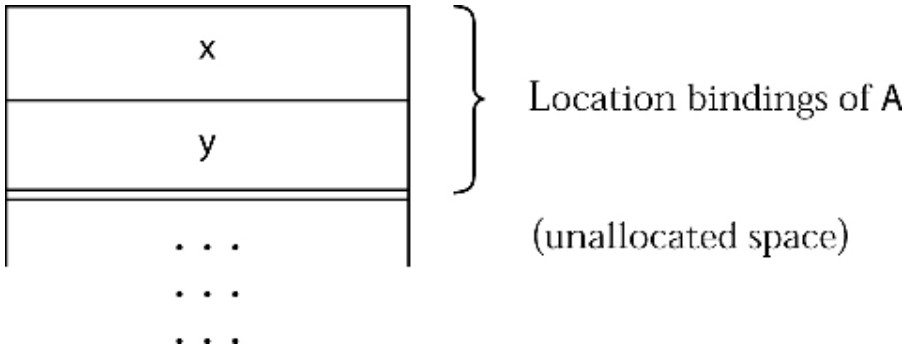
point 2



실행 시간 스택 조작

```
A: {   int x;
       char y;
   B: {   double x;
          int a;
       } /* end B */
   C: {   char y;
          int b;
          D: {   int x;
                 double y;
             } /* end D */
          } /* end C */
      } /* end A */
```

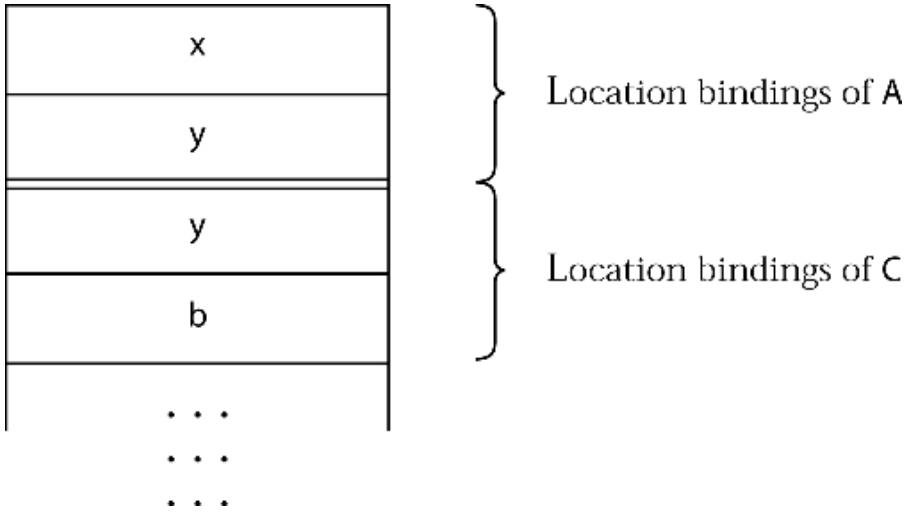
point 3



실행 시간 스택 조작

```
A: {   int x;
       char y;
   B: {   double x;
          int a;
       } /* end B */
   C: {   char y;
          int b;
          D: {   int x;
                 double y;
             } /* end D */
          } /* end C */
       } /* end A */
```

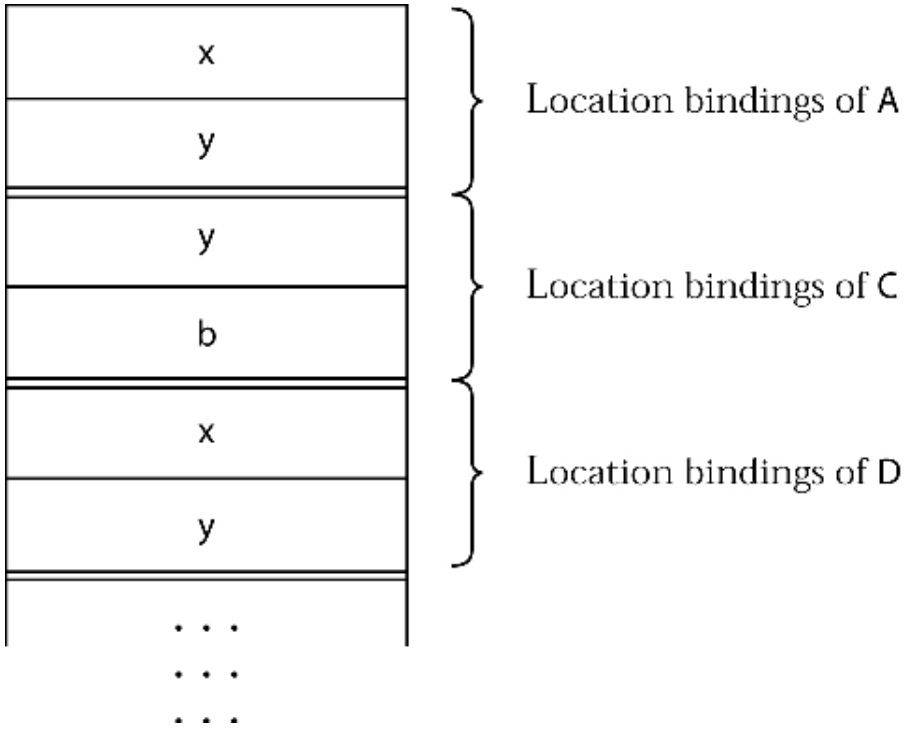
point 4



실행 시간 스택 조작

```
A: {   int x;
       char y;
   B: {   double x;
          int a;
       } /* end B */
   C: {   char y;
          int b;
          D: {   int x;
                 double y;
             } /* end D */
          } /* end C */
      } /* end A */
```

point 5



힙 조작

- 힙 (자유 공간)
  - 수동적으로 할당되는 객체들을 위한 메모리 풀
- 힙 해제
  - 수동 해제: 이를 위해서는 특정 함수나 연산자가 사용됨 (**`free`** in C, **`delete`** in C++)
  - 자동 해제: 쓰레기 수집기가 사용됨 (더 안전하지만 느림, Java)
- Ada의 접근 방법
  - delete 연산을 제공하지는 않으나, 사용자가 해제 연산을 정의할 수 있도록 해줌 (**`Unchecked_Deallocation`**)

포인터와 참조인출(Dereferencing)

- 포인터
  - 다른 객체의 참조를 값으로 가지는 객체
- 참조인출
  - 포인터의 값을 통해 객체를 참조함
- 힙 객체를 다루기 위해서는 포인터는 반드시 필요함 (암시적이건, 명시적이건)

```
/* C example */

int *x;
  // pointer declaration

x = (int*)malloc(sizeof(int));
  // memory allocation

*x = 5;
  // dereferencing

free(x);
  // deallocation
```

수명 (Lifetime)

- 저장할 수 있는 객체
  - 메모리 셀들의 모임
  - 환경에 의해 할당된 저장 공간의 영역
- 객체의 수명
  - 환경에서 그 객체가 할당되어 있는 기간
- 수명과 영역
  - 변수의 수명과 영역은 밀접하게 관련되어 있으나 같은 건 아님 (cf. C/C++의 지역 정적 변수들)
  - 영역에 따르면: 지역(local), 전역(global)
  - 수명에 따르면: 정적(static), 동적(dynamic)

C 에서의 지역 정적 변수

```
int p(void)
{ static int p_count = 0;
   /* initialized only once - not each call! */
  p_count += 1;
  return p_count;
}

main()
{ int i;
  for (i = 0; i < 10; i++)
  { if (p() % 3) printf("%d\n",p());
  }
  return 0;
}
```

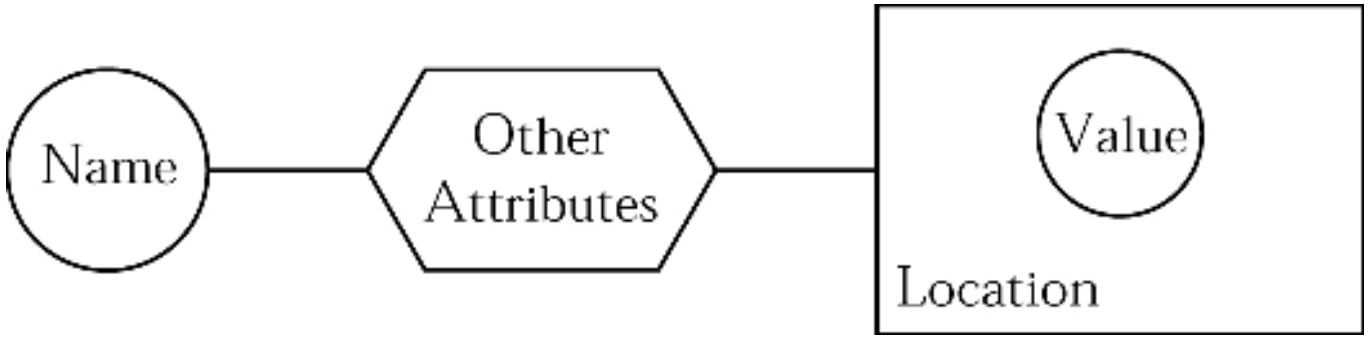
변수 p_count 는 함수 p를 호출한 개수를 가짐
따라서, p는 역사에 민감한다 (history sensitive).
출력이 무엇일지 알아맞혀볼 것!

변수와 상수

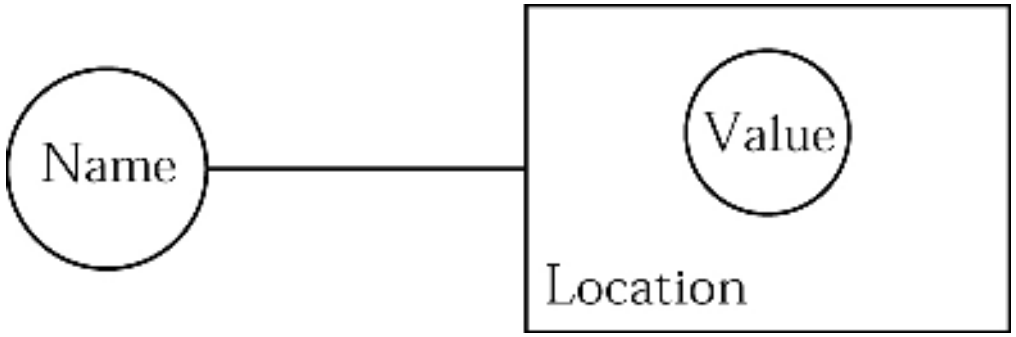
- 변수
  - 실행 시간 동안 값이 바뀔 수 있는 객체
- 상수
  - 수명 동안 값이 바뀌지 않는 객체
- 리터럴
  - 값인데, 그 이름으로부터 명시적인 언어 개체
    - (a language entity whose value is explicit from its name)
  - 상수의 한 종류이나 절대 메모리가 할당되지 않음
    - (a kind of constants but may never be allocated)

변수들의 다이어그램

- 도식적인 표현(Schematic Representation)



- 상자-원 다이어그램(Box-Circle Diagram)



L-value 와 R-value

- 변수의 L-value 와 R-value
  - l-value (LHS value): 위치
  - r-value (RHS value): 저장되는 값
- 언어의 예
  - ML은 참조값만을 가지며, r-value는 명시적임

    `x := !x + 1`

    `!x` 는 `x`의 r-value 를 의미
  - C는 주소 연산자 (`&`) 를 가지고, 참조인출 연산자 (`*`) 를 가지지만, l-values 와 r-values 간의 구별은 보통 암시적임

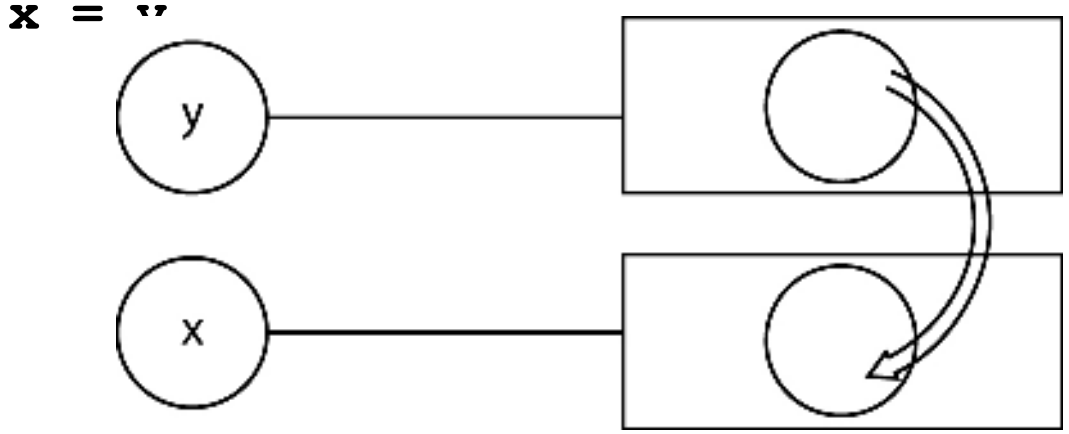
배정

- 일반 문법
  - infix 표현

    변수 배정_연산자 표현식
- 의미
  - 기억장소 의미론
    - 값 복사에 의해 할당함
  - 포인터 의미론
    - 공유 (얕은 복사 shallow copying)로 할당함
    - 복제 (깊은 복사 deep copying)로 할당함

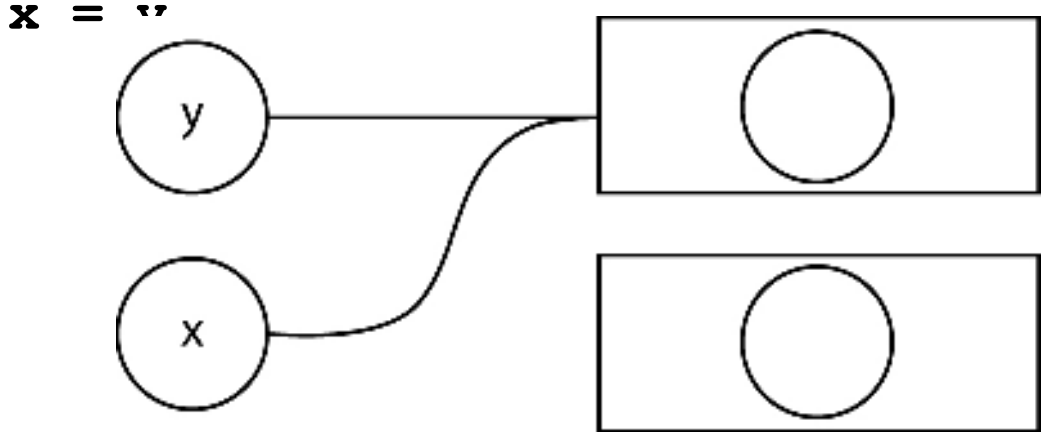
값 복사를 통한 배정

- 변수의 값이 복사됨



공유를 통한 배정

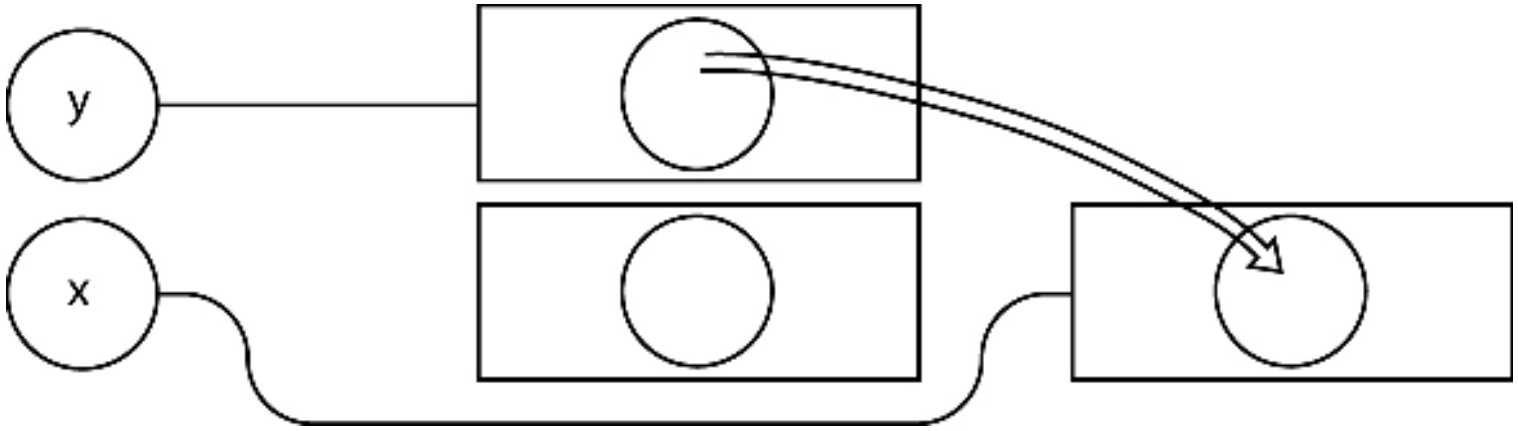
- 변수의 위치가 복사됨



복제(cloning)를 통한 배정

- 위치와 변수의 값이 복제됨

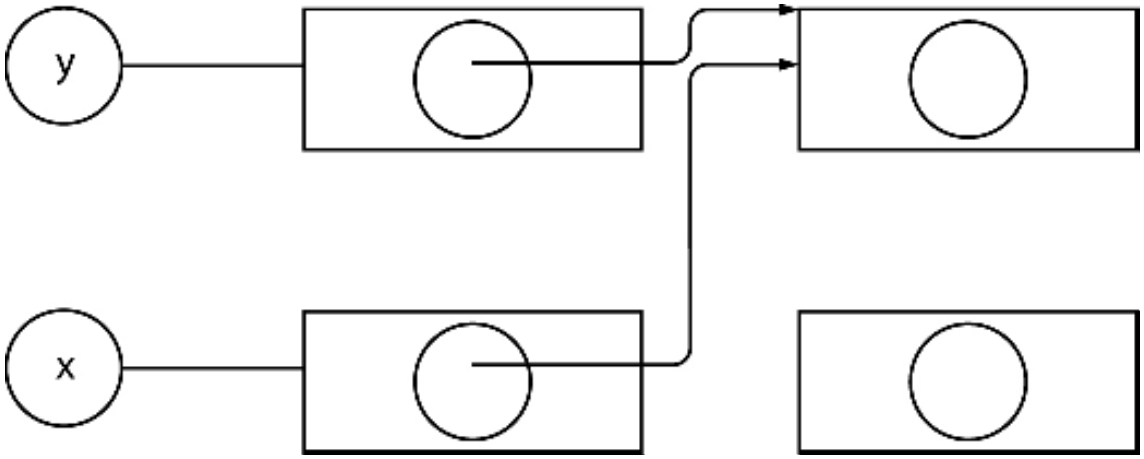
```
x = y
```



자바(Java)의 예

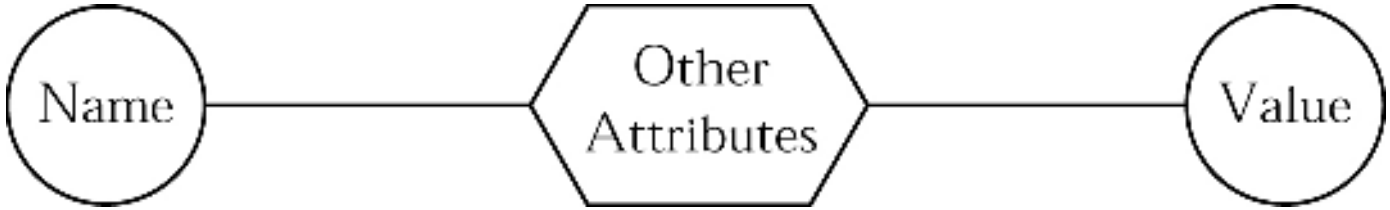
- 자바는 배정에서 앞의 언급한 모든 의미론을 지원함
  - 객체 변수의 배정: 공유 할당
  - 간단한 데이터의 배정: 값 복사 할당
  - 객체 복제는 **`clone`** 메쏘드로 지원됨
- 자바에서의 객체 배정을 자세히 살펴보면 다음과 같음

```
x = y
```



상수의 의미론

- 상수의 도식적 표현



- 상수는 값 의미론을 가짐
  - 한 번 값이 구성되면 값은 변하지 않음
  - 상수의 메모리 위치는 지정될 수 없음

상수들의 분류

- 리터럴과 이름이 있는 상수 (Named Constants)
  - 리터럴: 이름이 정확한 값을 나타냄
  - 이름이 있는 상수: 값의 의미를 나타내기 위한 이름
- Classification of Named Constants
  - 정적 상수 (현시 상수; manifest constants)
    - 컴파일 시간 정적 상수 (메모리 할당 안됨)
    - 메모리 적재 시간 정적 상수
  - 동적 상수

상수의 예 (Java)

- 자바에서 컴파일 시간의 상수
  `static final int zero = 0;`
- 자바에서 (메모리) 적재 시간의 상수
  `static final Date now = new Date();`
- 자바에서 동적인 상수:
  생성자 내에서 할당된 비정적인 최종 값 (any non-static final assigned in a constructor.)
- Java vs. C
  - 자바는 상수에 대해 매우 너그러운 관점을 가짐; 왜냐하면 컴파일 시간에 상수를 계산해서 제거하려고 고심하지 않기 때문
  - C는 상수에 대해 아주 엄격한 관점을 가지는 데, 컴파일 과정에서 상수들을 제거할 수 있도록 하기 위함임

상수 초기화

- C vs. C++
  - C에서 정적 상수 (또는 정적 변수)의 초기 값은 오직 리터럴로부터 계산되어야 함
  - C++에서는 이런 제한이 사라졌음

```
#include <stdio.h>
#include <time.h>

const int a = 2;
const int b = 27+2*2;          /* legal in C */

const int c = (int) time(0); /* illegal C code! */

int b = 27+a*a;                /* also illegal in C */
```

함수는 어떤가?

- 함수 상수
  - 대부분의 언어에서 함수의 이름이 상수임
- 함수 변수
  - 포인터를 통해 구현됨
- 함수 리터럴 (Function Literals (익명 함수; anonymous functions))
  - 대부분의 함수 언어는 익명 함수를 지원함

```
/* function pointer in C */
int gcd( int u, int v)
{ if (v == 0) return u;
  else return gcd(v, u % v);
}

int (*fv)(int,int) = gcd;

main()
{ printf("%d\n", fv(15,10));
  return 0;
}
```

Anonymous function in ML

```
(fn(x:int) => x * x) 2;
```

evaluates

```
val it = 4 : int
```

별명(Aliases)

- 별명
  - 같은 시간에 같은 객체에 대한 두 개 이상의 서로 다른 이름
  - 가독성에 좋지 않음 (잠재적으로 유해한 부작용을 초래함; 다음 슬라이드 참조)
- 부작용
  - 명령문의 실행 이후 변수 값이 바뀜
  - 일반적 정의: 비지역 변수나 함수/프로시저에 의한 입출력의 변경 (216쪽 문제 5.23)
- 잠재적으로 유해한 부작용
  - 쓰여진 명령문에서 부작용은 결정될 수 없는 경우
  - 코드의 앞 부분을 봐야 알 수 있음

C 언어의 시퀀스 포인트

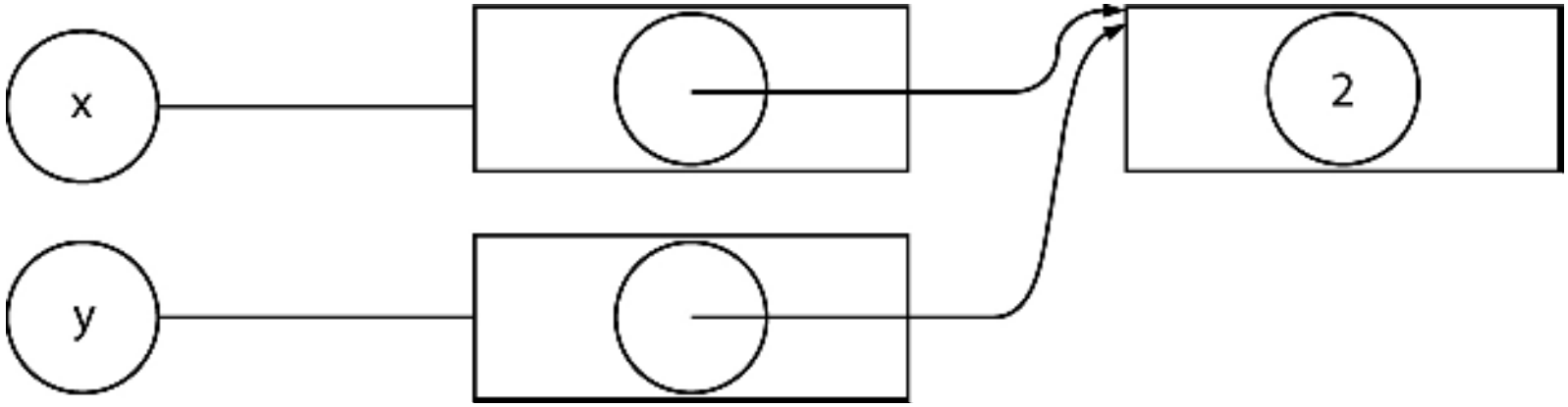
- 일반 : 프로그램의 실행이 다음 단계로 넘어가기 전에 모든 부수 효과들이 확실하게 평가되는 포인트
- C: ‘sequence point’라는 것은 어떤 시간대(전체 수식의 평가가 끝난 시점, 또는 ||. &&, ?:, 또는 콤마(comma) 연산자, 또는 함수 호출 바로 이전)의 위치를 의미하는 것으로, 모든 부작용이 일어나지 않는다고 보장하는 시점

C 언어의 시퀀스 포인트의 예

- C/C++에서 애매해서 정의되지 않은 경우
  - i = ++i ;
  - y = (4 + x++) + (6 + x++);
  - i++ * i++
  - a ^= b ^= a ^= b
  - a[i] = i++;
  - 등등
- 많은 C 책에서 잘못 설명하고 있으며, 절대 피해야 할 프로그래밍 형태임

유해한 별명의 예

```
main()
{ int *x, *y;
  x = (int *) malloc(sizeof(int));
  *x = 1;
  y = x;     /* *x and *y now aliases */
  *y = 2;
  printf("%d\n",*x);
  return 0;
}
```



별명은 어떻게 생기는가?

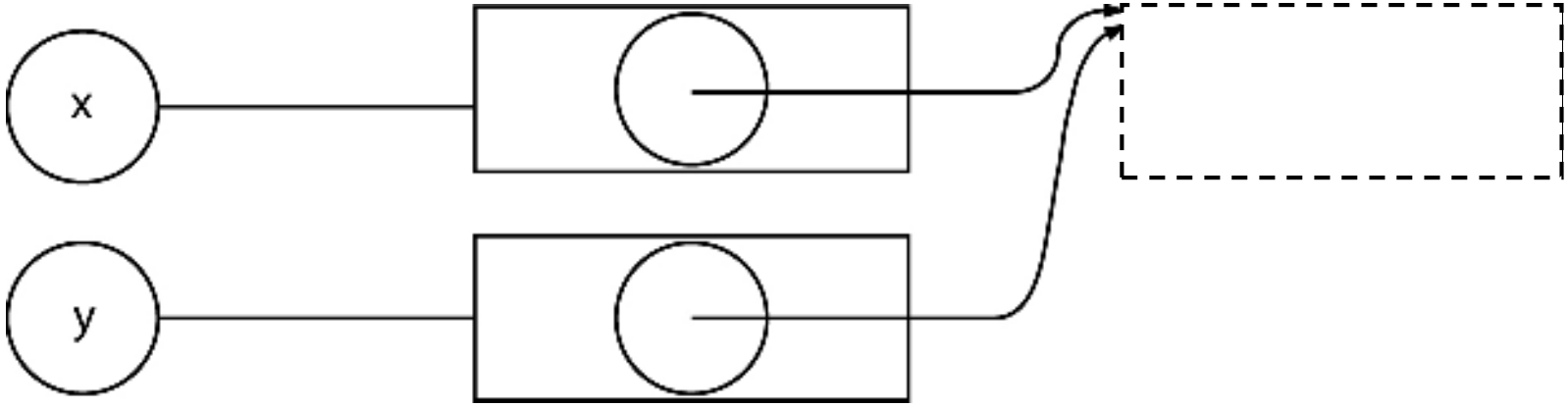
- 별명은 어떻게 생기는가?
  - 포인터 배정
  - 참조 전달(call-by-reference) 매개변수
  - 공유하는 방식으로 배정 (assignment by sharing)
  - 별명을 만들기 위한 명시적인 메커니즘: **EQUIVALENCE** and **COMMON** in FORTRAN
  - 변경가능한 레코드
- 포트란은 왜 별명에 대한 명시적인 메커니즘이 있는가?
  - 메모리를 절약하기 위해
  - 당시 메모리는 중요한 자원이었음

허상 참조(Dangling References)

- 허상 참조
  - 위치가 접근 가능하지만 해제되어 있음
  - 위치가 너무 일찍 해제되었음
  - 위험함!
- 어떻게 허상 참조가 만들어지나?
  - 포인터 배정 후 명시적인 해제
    - (pointer assignment and explicit deallocation)
  - 포인터 배정 후 암시적인 해제
    - (pointer assignment and implicit deallocation)
    - 블록을 빠져나감으로써 발생함
    - 함수를 빠져나감으로써 발생함

허상 참조의 예

```
main()
{ int *x, *y;
  x = (int *) malloc(sizeof(int));
  *x = 1;
  y = x;     /* *x and *y now aliases */
  free(x);   /* *y now a dangling reference */
  printf("%d\n",*y);  /* illegal reference */
  return 0;
}
```

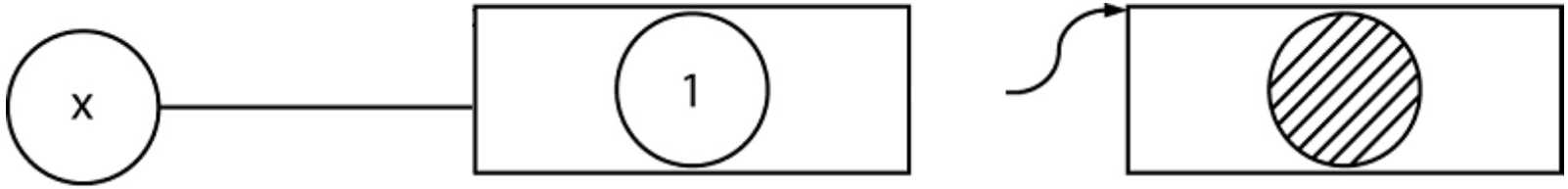


쓰레기

- 쓰레기 (허상 객체)
  - 잡혀 있지만, 접근할 수 없는 메모리
  - 해제를 미쳐 못한 경우 생김
  - 메모리를 낭비하는 것, 위험하진 않음
- 무엇이 쓰레기를 만드는가?
  - 명시적 할당과 접근 포인트는 다음 문제로 분실될 수 있음
    - 배정 (assignment)
    - 접근 포인트를 해제 (deallocation of the access point)

쓰레기의 예

```
main()
{ int *x;
  x = (int *) malloc(sizeof(int));
  x = 1;     /* OOPS! */
  ...
  return 0;
}
```



The above location turns into a garbage.

쓰레기 수집

- 자동으로 쓰레기를 회수하는 언어의 서브 시스템
- 대부분의 함수 언어 구현과 일부 객체지향 언어 구현은 쓰레기 수집기를 가지고 있음 (Section 8.5 참조)