이름, 바인딩, 영역 1

Concepts of Programming Languages

2024년 1학기

한양대학교 인공지능대학원
임덕선

목차

- 서론
- 이름
- 변수
- 바인딩(binding) 개념
- 영역(scope)
- 영역과 존속기간(life time)
- 참조환경

서론

- **Imperative languages are abstractions of von Neumann architecture**
  - Memory
  - Processor
- **Variables are characterized by attributes**
  - To design a type, must consider scope, lifetime, type checking, initialization, and type compatibility

이름 (변수명, 함수명, 클래스 명 등을 만드는 방법)

❑ 설계 고려사항

- 첫번째 문자는 반드시 알파벳
- 허용하는 문자들은? {alpha, digit, '-', '_'}
  - <id> –> <letter>{<letter>|<digit>|<특수기호>}*

- 길이의 제한

  예)
  - FORTRAN 95+ : 최대 31
  - C99 : 최대 31
  - C#, Ada, Java : no limit
  - C++ : no limit, 흔히 언어 구현자가 제한

- Implicit typing
  - 예) FORTRAN

이름 (변수명, 함수명, 클래스 명 등을 만드는 방법)

❑ 대소문자 구분(Case sensitivity)

- 단점 -> 가독성(판독성) 저하 : 유사하게 보이지만 이름이 다름
예) 클래스 이름은 대문자로 시작, 객체 이름은 소문자로 시작

- C++, Java, C#에서 미리 정의된 이름은 대소문자 혼합
  - e.g. IndexOutOfBoundsException

- C, C++, Java : 대소문자 구별
- 다른 언어들 : 대소문자 구별 않음

❑ 특수어(Special words)

- 가독성을 높이는 데 도움이 됨
- 문장과 프로그램의 구문 부분을 구별하는데 사용됨
  - Fortran과 같은 특정 상황

    ➔예) Real VarName (`Real = 3.4` (*`Real` is a variable*))
- 예약어는 사용자가 정의한 이름으로 사용할 수 없는 특수 단어

이름 (변수명, 함수명, 클래스 명 등을 만드는 방법)

❑ 예약어(reserved words)

- 사용자-정의 이름으로 사용될 수 없는 이름
  - 예) C에서int, float, if, while등
- 너무 많은 예약어를 포함하면, 사용자가 이름들을 구성하는데 어려움을 겪을 수 있음
  - 예) COBOL : 300개의 예약어
- 판독성에 도움
  - 문장 절을 구분하는데 사용

변수

- 프로그램 변수는 메모리 셀이나 셀들의 모임에 대한 추상화
- 다음 6개의 속성들이 특징 지워짐
  - 이름(name)
  - 주소(address)
  - 타입(type)
  - 값(value)
  - 존속기간(lifetime)
  - 영역(scope)

변수 속성

❑ 이름

- 대부분의 프로그래밍 언어에서 변수 이름의 일반적 형태
  - ✓ 첫 문자는 영문자
  - ✓ 둘째 문자부터는 영문자나 숫자

❑ 주소

- 변수와 연관된 메모리 주소
- 동일한 변수가 프로그램의 다른 시기에 다른 주소와 연관되는 것이 가능함
- 두 개의 변수 이름이 동일한 메모리 위치를 접근할 수 있다면, 이러한 변수들을 별칭(aliases)라 함
  - Ex) C프로그램에서 pa = &a(a와 pa는 같은 기억장소를 사용)
- l-value로 불림 (변수가 배정문의 좌측에 나타날 때 주소가 요구됨)
- 별칭은 판독성, 신뢰성을 저하

변수 속성

❑ 타입

- 변수의 값 범위를 결정
- 타입의 값들에 대해서 정의된 연산들의 집합 결정
  - 예) 정수형 (16bit의 경우)
    - ➢ 값의 범위 : -32768 ~ 32767
    - ➢ 연산의 종류 : +, -, * , /, mod

❑ 값

- 변수와 연관된 위치에 저장된 내용
- r-value 이라 불림 (변수가 우측에 나타날때 요구된 것은 값)
- 예) x = x + 1

바인딩 개념

❑ 바인딩(binding)

- 속성과 하나의 개체 간의 연관(association)
- 예) 변수와 그 타입 또는 값 사이, 또는 연산과 기호 사이의 연관

❑ 바인딩 시간(binding time)

- 바인딩이 일어나는 시기
  - 언어 설계 시간(design time) – 언어를 처음 설계할 때(언어 설계자)
  - 언어 구현 시간(implementation time) – 컴파일러 만드는 시간
  - 컴파일 시간(complie time)
  - 링크 시간(linking time)
  - 적재 시간(loading time)
  - 실행 시간(run time)

바인딩 시간

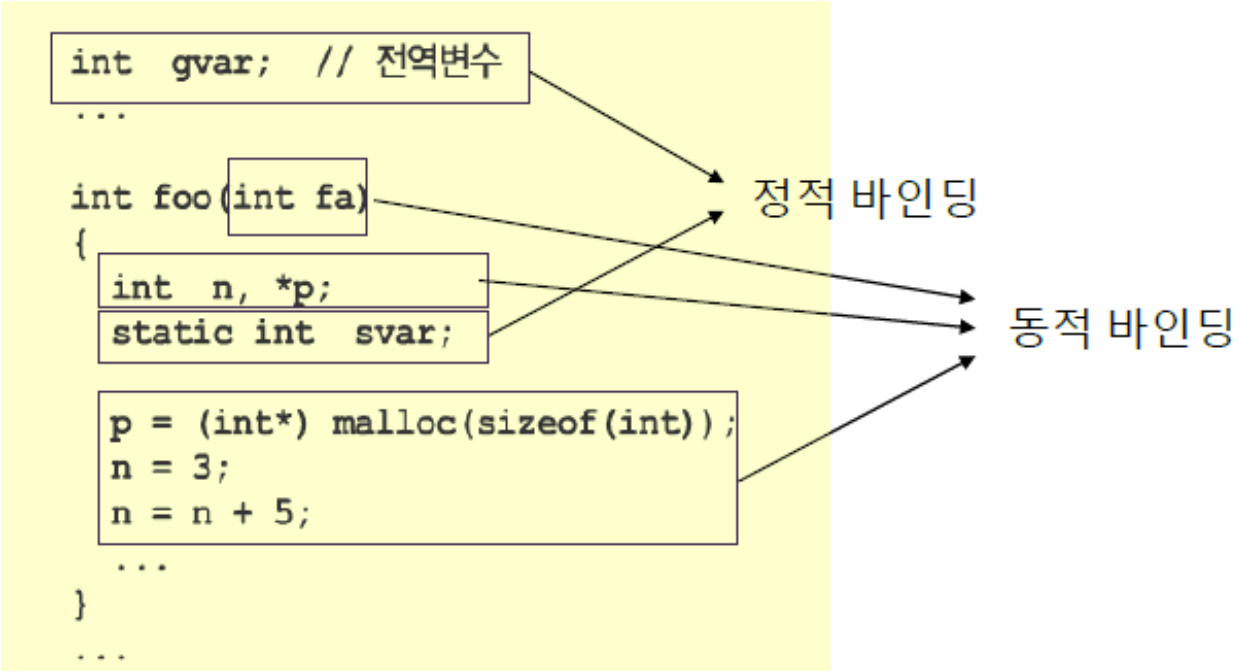
❑ 정적 바인딩(Static binding)

- 프로그래밍 언어를 설계할 때
  - 연산자 기호 : 연산 (‘+’ : 더하기)
- 프로그래밍 언어를 구현할 때
  - 정수형 값의 범위 : 16-bit/32-bit
  - 실수형의 표현
- 프로그램을 컴파일 할때
  - 변수의 타입 (C의 경우)
- 프로그램을 메모리에 Load할 때
  - 전역 변수의 주소(C의 경우)
  - Static 변수의 주소(C의 경우)

바인딩 시간

❑ 동적 바인딩(Dynamic binding)

- 프로그램을 실행할 때
  - Static이 아닌 지역변수의 주소(C의 경우)
  - 부프로그램의 인자 (parameter)의 주소
  - 동적으로 할당된 기억장소의 주소



정적과 동적 바인딩

- 바인딩이 실행 전에 일어나서 변경되지 않으면, 그 바인딩은 정적**(static)**

- 바인딩이 실행 중**(run time** 때**)**에 일어나서 변경될 수 있으면, 그 바인딩은 동적**(dynamic)**

변수의 타입 바인딩

❑ 정적 타입 바인딩(static type binding)

- 대부분의 컴파일 언어에서 프로그램을 컴파일할 때 타입 바인딩이 이루어 짐
- 대부분 명시적 선언(explicit declaration) 또는 묵시적 선언(implicit declaration)에 의해 생성함

❑ 동적 타입 바인딩(dynamic type binding)

- 대부분의 인터프리터(interperter) 언어에서 프로그램을 실행할때 타입 바인딩이 이루어짐
- 대부분의 값의 배정 방법에 의해서 타입이 지정됨
  - 예) JavaScript
    - list = [2, 4.33, 6, 8]
    - list = 17.3
- 단점
  - 프로그램 실행 비용 증가
    - > 타입검사(type checking)가 프로그램 실행시 이루어져야 함
  - 타입 오류를 발견하기가 어려움

```
n = 3;  d = 5;  s = 7.4;  // n,d:정수형, s:실수형
n = s;                    // 입력 실수(원래 의도는 n = d;). 오류는 아님.
                          // 변수 n의 타입이 실수형으로 바뀜.
```

변수의 타입 바인딩

❑ 명시적 타입 바인딩(explicit type binding)

- 변수의 타입을 지정하는 선언문에 의해서 타입이 지정
- 예) C에서
  - int age, score; or float average;
- 장점 : 신뢰성 향상

❑ 묵시적 선언(implicit declaration)

- 언어 설계 시 정의된 규칙에 의해서 변수의 타입이 지정
- 프로그램에서 변수가 처음 나타나는 지점에서 타입이 지정됨
- 예) Fortran에서
  - 정수형 : 변수 이름이 i~n으로 시작
  - 실수형 : 변수 이름이 기타 문자로 시작
- 타입 추론(type inference)에 의한 타입 바인딩
  - 예) ML 언어 : fun circumf(r) = 3.14 * r * r //r: 실수형
    - C#

```
var sum = 0;   // int type
var total = 0.0; // float type
var name = "Fred"; //string type
```

- 장점 : 작성력 향상
- 단점 : 신뢰성 하락

기억 공간의 바인딩(Storage Binding)

❑ 기억 장소 할당(allocation)

- 변수 타입의 크기 만큼의 공간을 변수에 할당(변수와 기억 장소 간의 바인딩 설정)

❑ 기억 장소 회수(deallocation)

- 변수에 할당된 기억 장소를 사용되지 않는 공간으로 반환(변수와 기억장소 간의 바인딩해제)

❑ 존속기간(lifetime)

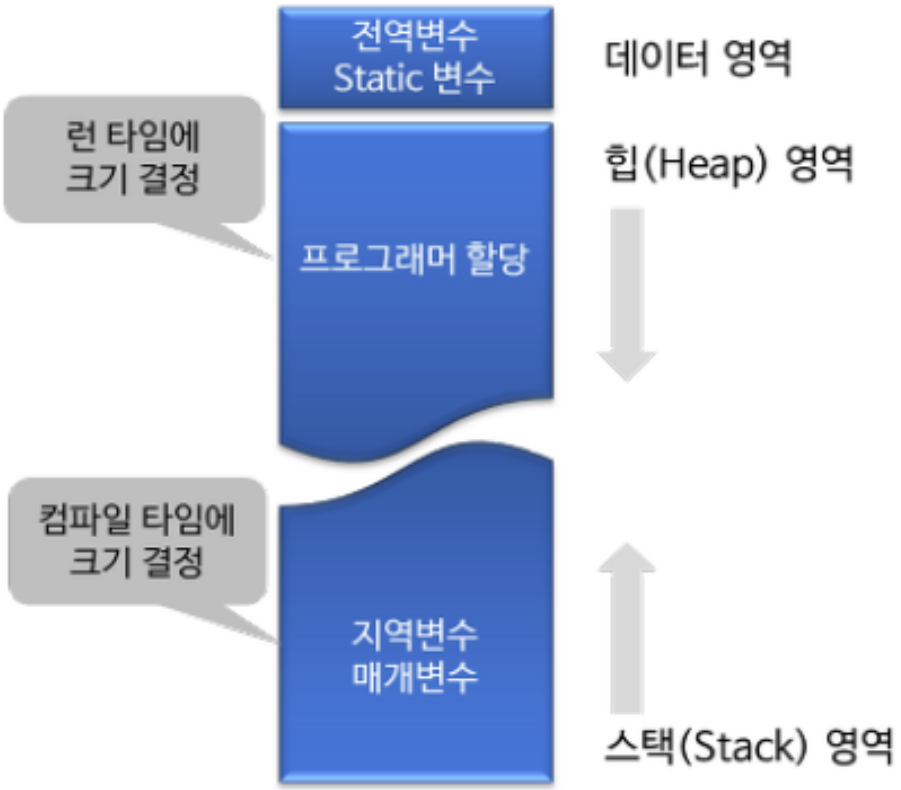
- 변수에 기억장소가 할당되는 시점부터 반환되는 시점까지의 기간(변수가 기억장소에 바인딩되어 있는시간)

❑ 변수에 기억장소가 할당되는 시점부터 반환되는 시점까지의 기간(변수가 기억장소에 바인딩되어있는 시간)

- 정적(static) 변수, 스택-동적(stack-dynamic)변수, 명시적-힙동적(explicit heap-dynamic) 변수, 묵시적-힙동적(implicit heap-dynamic) 변수

기억 공간의 바인딩(Storage Binding) – 메모리 영역과 변수들

- 데이터(Data) 영역
  - ✓ 전역변수와 static변수가 할당
  - ✓ 프로그램 시작 동시 할당, 프로그램이 종료되어야 메모리에서 소멸됨
- 스택(Stack) 영역
  - ✓ 함수 호출시 생성되는 지역변수와 매개변수가 저장되는 영역
  - ✓ 함수 호출이 끝나면 사라짐
- 힙(Heap) 영역
  - ✓ 필요에 의해 동적으로 메모리를 할당할때 사용



< 메모리의 영역 >

기억 공간의 바인딩– 정적 변수

❑ 정적 변수

- 프로그램 실행 전에 정적 영역의 기억장소가 바인딩
- 바인딩 된 기억 장소는 프로그램이 종료될 때까지 유지됨
  - C에서 static으로 선언된 모든 변수, Fortran 77에서 모든 변수

- 장점
  - 프로그램 실행의 효율성
    - 변수에 직접 접근할 수 있음(direct addressing)
    - 변수의 할당과 회수를 위한 부담이 초래되지 않음
  - 부프로그램(subprogram)에서 과거-민감(history-sensitive)변수 지원
    - 부프로그램을 호출해서 실행을 시작할 때, 이전 호출에서 마지막으로 배정된 값을 갖는다
    - 난수(random number)를 생성할 수 있는 부프로그램 작성에 사용됨
      - int random() { static int seed; ... }
- 단점
  - 정적변수만 갖는 언어 : 재귀적(recursive) 부프로그램 작성이 불가능함
  - 기억 공간이 변수들 간에 공유될 수 없음

기억 공간의 바인딩– 스택 동적(stack-dynamic) 변수

- **스택-동적 (stack-dynamic)변수**
  - 프로그램 실행되는 과정에서, 변수 선언문이 세련화(elaboration)될 때 stack 영역의 기억장소가 바인딩
    - 선언문과 연관된 실행코드가 실행될 때를 그 선언문은 세련화되다고 함
    - 예) int foo() { int n; ... }
  - 스칼라 변수의 경우, 다른 속성(이름, 타입, 영역)은 정적으로 바인딩
    - C에서 함수의 지역변수
  - 장점
    - 재귀적 부프로그램 작성이 가능함
    - 예) int roo() { int rv; ... roo() ...}
    - 지역 변수들 사이에 기억 장소를 공용할 수 있음
  - 단점
    - 프로그램 실행 시간의 증가
      - 기억 장소의 할당 및 반환 시간이 추가적으로 필요함
      - 변수에 간접적으로 접근해야함(in-direct addressing)
    - 부프로그램에서 과거 민감 변수(history sensitive)를 이용할 수 없음

기억 공간의 바인딩 – 명시적 힙-동적(Explicit heap-dynamic)

❑ 명시적 힙-동적(Explicit Heap-dynamic) 변수

- 기억 장소를 할당하는 문장을 실행할 때 heap 영역의 기억 장소가 바인딩 되고, 반환하는 문장을 실행할 때 반환
- 포인터나 참조 변수(reference variable)를 통해서 접근할 수 있음
- 연결리스트(Linked-list)나 tree와 같이 프로그램 실행 중에 크기가 커지거나 줄어드는 경우에 사용
- 장점
  - 필요에 따라 기억 공간을 효율적으로 관리하며 사용할 수 있음
- 단점
  - 포인터나 참조 변수를 올바르게 사용하기 어려움 -> 신뢰성 저하
  - 기억장소의 동적 할당 및 반환에 필요한 실행 시간 증가

C

```
int  *p;
...
p = (int*) malloc(sizeof(int));
... *p ...
free(p);
```

C++

```
int  *p;
...
p = new int;
... *p ...
delete p;
```

기억 공간의 바인딩 – 묵시적 힙-동적(implicit heap-dynamic)

❑ 묵시적 힙-동적(implicit heap-dynamic) 변수

– 배정문(assignment statement)을 실행할 때 기억장소가 바인딩

- JavaScript의 문자열(string)

```
var   st = "Hong";    // 길이: 4
st += " Gil-Dong";   // 길이: 13으로 확장됨
```

- JavaScript의 배열(array)

```
var   na = new Array(34, 52, 19); // 크기: 3
na[8] = 68;                       // 크기: 9로 확장됨
```

– 쓸모가 없어진 기억장소는 보통 시스템에 의한 쓰레기 수집(garbage collection)을 통해 반환됨

기억 공간의 바인딩 – 묵시적 힙-동적(implicit heap-dynamic)

❑ 묵시적 힙-동적(implicit heap-dynamic) 변수

- 장점
  - 유연성 향상
    - ✓ 기억장소를 필요할 때 할당하고, 필요가 없어졌을 때 반환
    - ✓ 모든 타입에 적용될 수 있는 포괄형 코드(generic code) 작성이 가능
- 단점
  - 기억장소의 동적 할당 및 반환에 필요한 실행 시간 증가
  - 컴파일할때 오류 감지 능력 상실

변수의 영역(Scope)

❑ 변수의 영역(Scope)

- 프로그램에서 변수가 가시적(visible)인 영역
- 프로그램의 어떤 지점에서 변수를 사용할 수 있으면, 변수는 그 지점에서 가시적임

❑ 지역변수(Local variable)

- 부프로그램이나 블록 내부에 선언된 변수
- 변수가 선언된 부프로그램이나 블록 내부에서는 가시적이나, 외부에서는 가시적이지 않음

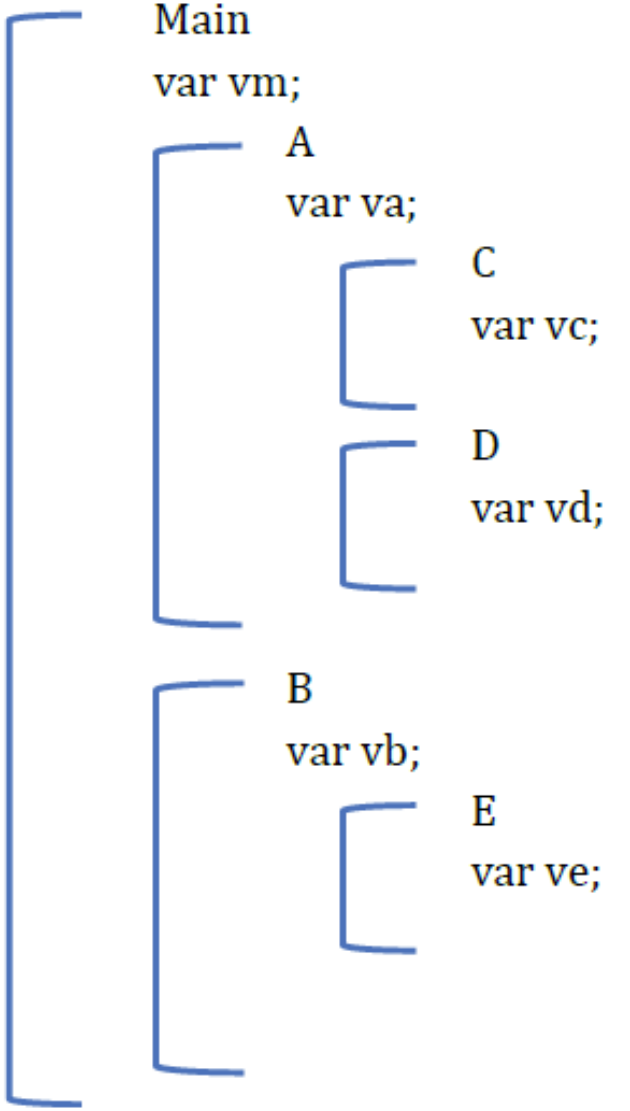
❑ 비지역변수(Nonlocal variable)

- 부프로그램이나 블록 내부에서는 가시적이나, 내부에 선언되어 있지 않음

❑ 전역 변수(Global variable)

- 프로그램 전체 영역에서 가시적인 변수

변수의 영역(Scope) 예제



예)
vm은 Main 블록의 지역변수
Va는 A블록 의 지역변수 … (각 블록이름으로 선언)

vc는 C블록에서는 가시적이나
A 블록에서나 Main, D블록에서는 가시적X

C 블록에서 볼때 가시적인 변수는 va와
vm이 될 수 있음 -> 비지역변수

Main에서 선언된 vm은 모든 블록에서
가시적인 변수 -> 전역변수

영역 규칙 (Scope rule)

❑ 영역 규칙

- 프로그램의 어떤 지점에서 사용된 변수 이름을 어느 선언문에서 선언된 변수와 대응시켜줄 것인지를 결정하는 규칙

❑ 두 가지 규칙

- 정적 영역 규칙(Static scope rule)
- 동적 영역 규칙(Dynamic scope rule)

❑ 중첩된 구조(Nested structure)

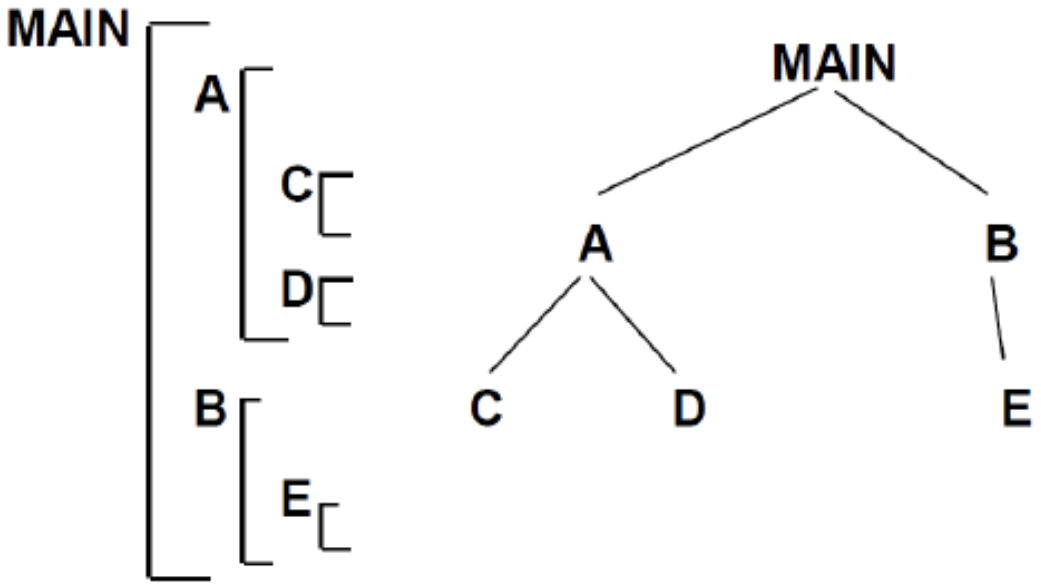
- 부프로그램이 내부에 중첩된(nested) 부프로그램을 정의할 수 있는 언어
  - 예) Ada, JavaScript, Common, LISP, python 등
- 중첩된 부프로그램을 정의할 수 없는 언어
  - 예) C 기반 언어
- 블록으로 중첩될 수 있음

```
int foo()
{ ...
  while (…) { int k; ... }
  ... }
```

블록 (Blocks)

- 프로그램 단위 내부에 정적 영역을 생성하는 방법을 제공
  - 블록을 제공하는 언어를 블록 구조 언어(block-structured language)
  - 블록 안에 선언된 변수들은 스택-동적(stack-dynamic)
  - 예) 알골-유사 언어(=Algol-like language)

- **블록(Blocks)과 정적 영역(Static scope)**
  - 블록과 정적 영역을 표현



정적 영역 규칙 (Scope rule)

- 부프로그램들의 공간적 배치 구조(프로그램 텍스트)에 근거함 즉 공간적 (spatial)
  - 변수 영역이 실행 전에 결정
  - 이름 참조를 변수와 연관시키기 위해서 , 해당 선언문을 찾음
  - 정적 부모(static parent) : 자신을 직접 둘러싸고 있는 부프로그램
    - Main : A와 B의 정적부모, A: C와 D의 정적 부모
  - 정적 조상(static ancestors) : 자신을 둘러싸고 있는 모든 부프로그램들
    - Main, A : C와 D의 정적 조상, A: C와 D의 정적부모

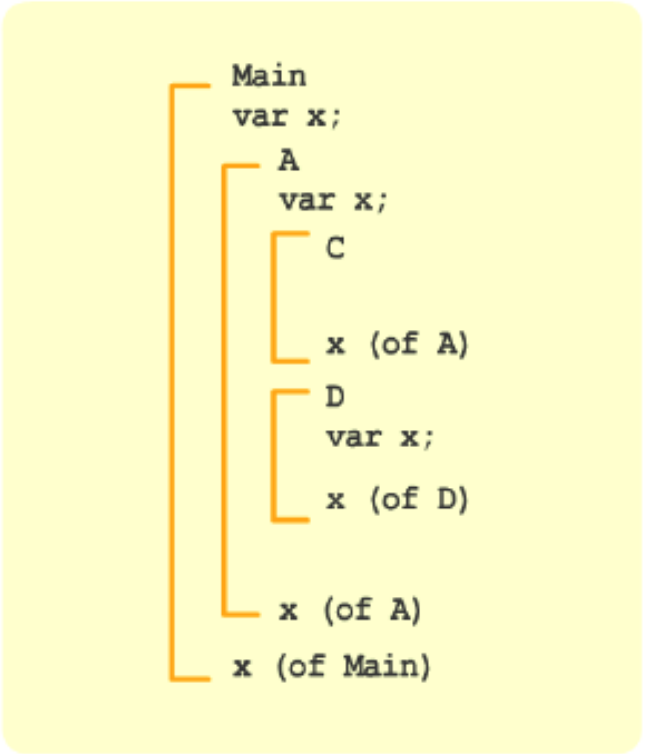
- 사용된 변수 이름에 대한 해당 변수의 선언문을 아래와 같은 순서로 찾음
  1. 변수 이름이 사용된 부프로그램내에서 찾음 (찾으면 지역변수)
  2. 변수 이름이 사용된 부프로그램의 정적 부모 내에서 찾음 (찾으면 비지역 변수)
  3. 선언문을 찾을 때까지 2번 과정을 반복, 모든 정적 조상 내에서 순서대로 찾음 (마지막에 찾으면 전역변수)
  4. 위 과정에서 찾지 못하면 오류

정적 영역 규칙 (Scope rule)

❑ 은폐되는 변수(Hidden variable)

- 변수 이름이 사용된 지점에서 가까운 곳에 선언문이 있으면, 그 곳의 정적 조상에 선언된 동일한 변수는 변수 사용지점에서는 보이지 않음

  - C 내부에서 사용된 변수 x는 A에서 선언된 변수, Main에서 선언된 x는 은폐
  - D 내부에서 사용된 변수 x는 D에서 선언된 변수, A와 Main에서 선언된 x는 은폐
  - A 내부에서 사용된 변수 x는 A에서 선언된 변수, Main에서 선언된 x는 은폐



- Ada에서 D 내부에서 은폐되어 있는 A에서 선언된 변수 x사용하기
  - A.X
- C++에서 , foo에서 은폐되어 있는 전역 변수로 선언된 변수 x사용하기
  - ::X

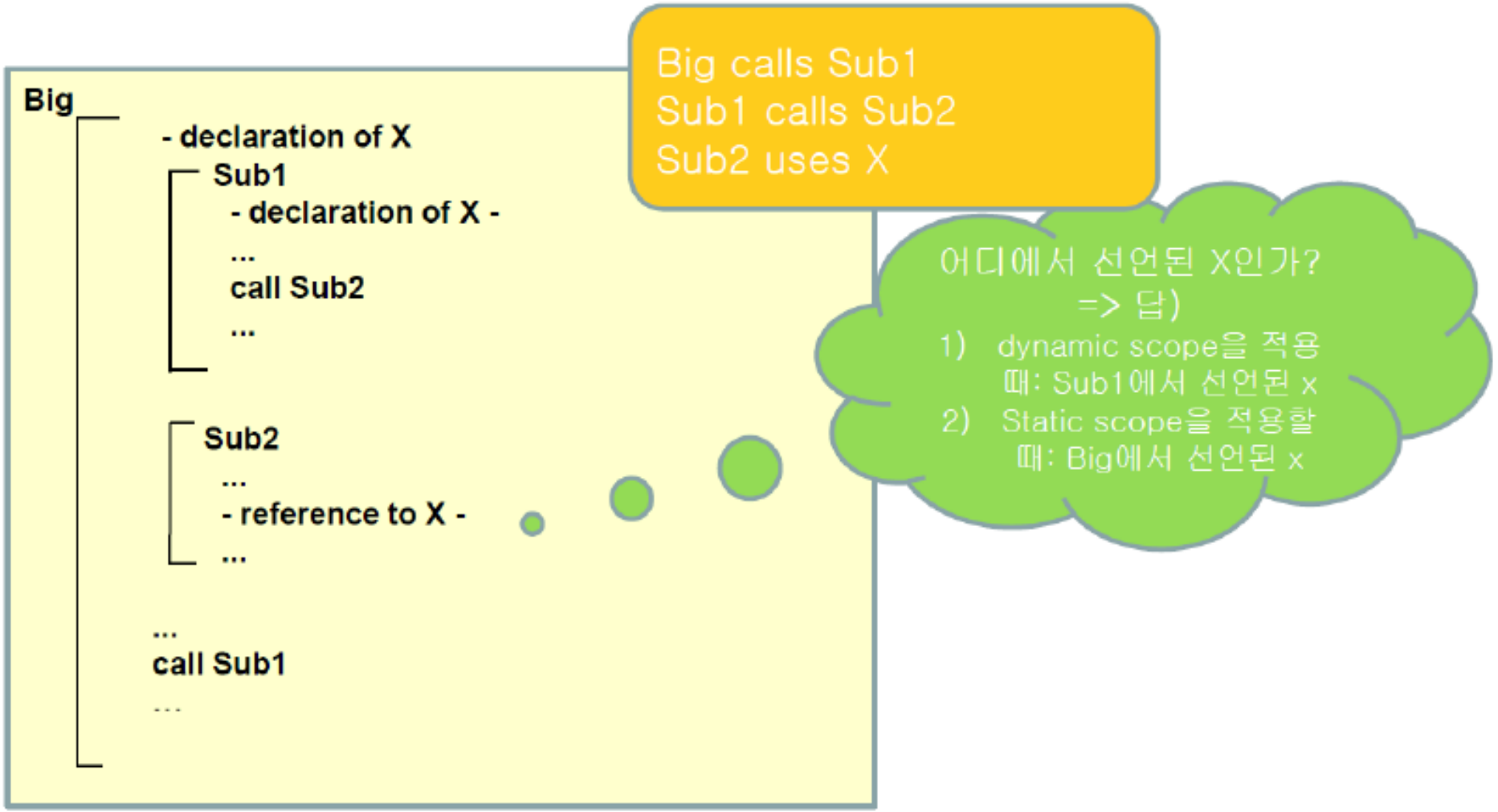
```
int  x;
...
int foo()
{ int  x;
  ...
  x (of foo)
  ::x
  ...
}
```

동적 영역 규칙 (Scope rule)

- **부프로그램 호출 시퀀스(calling sequence)에 기반, 즉 시간적(temporal)**
  - 프로그램 텍스트에 기반하지 않음
  - 동적 영역은 실행 시간에 결정

- **변수 탐색 과정**
  1. 변수 이름이 사용된 부프로그램 내에서 찾음 (찾으면 지역변수)
  2. 변수 이름이 사용된 부프로그램을 호출한 부프로그램 내에서 찾음
  3. 선언문을 찾을 때까지 2번 과정을 반복하여, 호출된 부프로그램을 역순으로 찾음.
  4. 위 과정에서 선언문을 찾지 못하면, 오류 처리함

동적 영역 규칙 (Scope rule) -예제



예제 – 정적과 동적 영역 규칙 (Scope rule)

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

<정적 영역 규칙>
함수 호출 순서와 상관없이 항상
sub2의 x는 big에서 선언된 x

<동적 영역 규칙>
Case 1 : big()->sub1()->sub2()

이때 sub2의 x는 sub1에서 선언된 x
(x=7)

Case 2 : big()->sub2()

이때 sub2의 x는 big에서 선언된 x
(x=3)

영역 (Scope) 과 존속기간(Lifetime)

- 영역과 존속기간은 연관되어 보임
  - 프로그램 텍스트에 기반하지 않음
    - 영역 : 변수 선언문부터 부프로그램의 끝부분까지
    - 존속기간 : 부프로그램이 호출되어 실행을 시작한 시점부터 부프로그램의 실행이 종료되어 return할 때까지

- 영역과 존속 기간은 완전히 별개의 개념
  - 영역은 공간적(spatial)개념, 존속기간은 시간적(temporal) 개념
  - 부프로그램에서 static으로 선언된 지역변수(정적변수)
    - 영역 변수 : 변수 선언문부터 부프로그램의 끝부분까지
    - 존속시간 : 프로그램의 시작 시점부터 프로그램의 종료 시점까지

```
void goo()
{ int  r;
  ...     }
void foo(...)
{ int          p;
  static int   q;
  ... goo(); ... }
```

	영역	존속기간
p	foo 내부	foo가 호출됨 ~ goo 실행 ~ foo에서 return
q	foo 내부	프로그램 시작 ~ 프로그램 종료
r	goo 내부	goo가 호출됨 ~ goo에서 return

참조 환경

- **프로그램의 어느 한 문장(statement) 참조 환경**
  - 문장에 가시적인 모든 이름(name)들의 집합
  - 정적 영역 규칙 적용되는 경우
    - 지역변수
    - 정적 조상(부모 포함)의 지역변수
    - 은폐되는 변수(hidden variable)는 제외
  - 동적 영역 규칙 적용되는 경우
    - 지역변수
    - 문장이 실행될 때까지의 활성 프로그램들의 지역변수
    - 은폐되는 변수는 제외
  - 활성 부프로그램이란 부프로그램이 호출되어 실행이 시작되었으나, 아직 종료되지 않은 상태에 있는 부프로그램

Q and A