제 5 장

이름, 바인딩, 영역

제 5 장 이름, 바인딩, 영역

❖ 목적 :

- 서론- 변수에 대한 기본적인 의미적 논점들을 소개
- 이름 - 식별자
- 변수 – 변수의 속성(타입, 주소, 값 등)
- 바인딩의 개념 – 바인딩과 바인딩 시간
- 영역 – 정적 영역 규칙과 동적 영역 규칙
- 영역과 존속기간
- 참조 환경
- 이름 상수

5.1 서론

- **변수에 관한 기본적인 의미론**
  - 변수 – 메모리 셀(memory cell)에 대한 언어의 추상화
  - 이름(name)과 변수
  - 변수의 속성: 타입(6장에서 설명), 주소, 값
  - 별칭
  - 바인딩과 바인딩 시간
  - 변수의 영역(scope)과 존속 기간(life time)
  - 타입 검사, 강 타입

5.1 서론

❖ **절차형 혹은 명령형 언어를 설명하는 방법** :

➢ 절차형 언어와 함수형 언어를 비교하는 것에 의해 특징 –

	명령형 언어	함수형 언어
변수	기억 장소를 표시하기 위한 변수 사용	수학의 변수와 연관 있음. 변수의 값은 변경되지 않음
배정문	값을 변경시키기 위한 배정문 사용	a=3은 수학에서 말하는 a는 3이다. 이 값은 그 변수의 유효 범위 안에서는 변경되지 않는다.
순서	순차적으로 실행	순서가 없다. 배정문의 의미가 아니므로 순서가 없다.
	폰노이만의 구조를 따르는 언어(반복문 추가)	

➢ 폰노이만 구조

- ✓ 데이터와 프로그램은 모두 동일한 기억 장소에 저장
- ✓ 명령어를 실행하는 cpu는 기억 장소와 떨어져 있어서 명령어와 데이터는 기억 장소로 부터 cpu에 전달되어야 하고 연산 결과는 다시 기억 장소에 전달.
- ✓ 반복문- 폰노이만 구조에서는 연산을 구현하는 가장 효율적인 방법, 빠르게 실행(재귀 함수 사용을 권장하지 않음)

5.2 이름

- 이름(name)은 식별자(identifier)와 혼용하여 사용
  - 식별자 – 어떤 개체를 식별하기 위해서 사용되는 문자열
  - 식별자와 변수의 공통점과 차이점
- **이름 설계 고려 사항**
  - 최대 길이
    - ✓ Fortran 90, 95+, C의 외부 이름은 31 문자로 제한
      - ♣ 외부 이름 – external name, 함수 외부에서 정의되고, 링커에 의해서 처리되어야 하는 이름
    - ✓ Java, C#, Ada, C의 내부 이름(internal name)은 제한 없음
    - ✓ C++는 제한 없음(언어 구현자들이 때때로 제한)
  - 연결 문자의 허용 여부
    - ✓ 밑줄 문자('_') : 1970년대와 1980년대는 널리 사용, 지금은 훨씬 덜 사용.
    - ✓ C 기반 언어에서 밑줄 문자는 낙타 표기법으로 대체
      - ♣ 변수나 함수명을 작성할 때 사용하는 표기법 – 낙타 표기법, 파스칼표기법
      - ♣ 낙타 표기법(camel notation) – 낙타 등 같이 생겼다고 함.
        - 이름에 포함된 모든 단어는 첫번째 단어를 제외하고 대문자로 시작.
        - 예 – myStack, typeName, iPhone, uHealthcare
      - ♣ 파스칼 표기법 - 첫 단어를 대문자로 시작하는 표기법
        - PowerPoint, MyStack

5.2 이름

- ✓ PHP – 모든 변수 이름은 달러 기호로 시작
- ✓ Perl – 변수 이름 앞의 특수 문자 $(스칼라 변수), @(배열), 또는 %(해시)는 그 변수의 타입을 명세

➢ 대, 소문자의 구별

- ✓ C 언어 기반 언어 – 이름에서 대문자와 소문자 구별
- ✓ C, C++, Java
- ✓ C++에서 다음 3가지 이름은 구별
  - ♣ Rose, ROSE, Rose – 가독성의 심각한 손상(매우 유사하게 보이는 이름들이 모두 다른 개체들을 가리킴)

❖ **특수어(special word)**

- ✓ 키워드나 예약어 등을 포함하는 특수어는 수행될 행동들을 명칭화하여 프로그램의 가독성을 높이게 하는 데 사용.
- ✓ 대부분의 언어에서는 특수어는 예약어로 분류.

➢ 키워드(keyword) : 주어진 문맥에서만 단지 특별한 의미를 갖는 프로그래밍 언어의 단어, 이는 변수 이름으로 사용 가능.

➢ 예약어(reserved word) : 변수 이름으로 사용할 수 없는 프로그래밍 언어의 특수어

5.2 이름

- ✓ In Fortran, - 특수어가 키워드
- ✓ Integer Apple - Integer가 선언문이라는 것을 표시
- ✓ Integer = 4 - Integer가 변수라는 것을 표시
- ✓ Integer Real - Real을 Integer 타입으로 선언
- ✓ Real Integer - Integer를 Real 타입으로 선언

- ✓ Fortran 컴파일러와 Fortran 프로그램을 읽는 사람들은 예약어와 식별자를 문맥으로 구별해야 함.

- ✓ 예약어를 많이 사용하면 일반 식별자를 구성하기가 어려움.
  - ♣ Cobol 언어는 예약어를 300개 가지고 있음 – 많이 사용하기 어려운 예약어 (LENGTH BOTTOM, DESTINATION, COUNT)

5.3 변수

- 변수는 기억 장소의 셀이나 셀들의 모임에 대한 추상화

- **변수의 6가지 속성**
  - 이름(name), 주소(address), 값(value), 타입(type), 존속 시간(lifetime), 영역(scope) => 존속 시간(5.4.3절)과 영역(5.5절)은 뒤에 서술
- **주소**
  - 변수와 연관된 기억 장소의 주소
  - 많은 언어에서는 동일한 변수가 프로그램의 다른 시기에 다른 주소와 연관되는 것이 가능
    - ✓ sub1, sub2가 동일한 지역변수 sum을 가질 때
    - ✓ 재귀적으로 호출되는 부프로그램이 지역변수를 가질 때
  - 변수의 주소는 **l-value(location 혹은 left)**로 불린다.

5.3 변수

- **별칭(alias)**
  - ✓ 두 개 이상의 변수가 동일한 기억 장소를 가질 때
  - ✓ 가독성 저하, 프로그램 검증을 매우 어렵게 하는 요소임.
  - ✓ 예를 들어
    - ♣ 변수 total과 sum이 별칭이면 total의 값 변경은 sum 의 값을 변경시킨다.
    - ♣ C에서 공용체, Fortran의 equivalence
      - 공용체 – 메모리의 같은 영역에 상이한 여러 자료 형을 사용하도록 하는 것

```
union flextype {
                    int intE1;
                    float floatE1;
                  } ;
union flextype el1;
Float x;
 . . .
El1.intE1 = 27;
X=el1.floatE1;
```

- **타입(type)**
  - 타입은 변수가 가질 수 있는 값의 범위와 그 타입의 값에 대해서 정의되는 연산들의 집합을 결정
  - 예: C나 java의 int
- **값(value)**
  - 변수에 연관된 기억 장소 셀이나 셀들의 내용
  - 변수의 값을 **r-value**라고 부름.

5.4 바인딩 개념

- **바인딩(binding) 이란** -
  - 프로그램의 기본 단위에 이 기본단위가 택할 수 있는 여러 가지 속성(attribute)중에서 일부를 선정하여 결정해주는 행위(변수의 타입, 변수의 기억장소, 변수의 값 등)
  - **바인딩 시간(binding time)** - 바인딩이 일어나는 시간
- **가능한 바인딩 시간**
  - **언어 설계 시간** - 프로그래밍 언어에서 허용되는 자료구조, 프로그램 구조, 택일문에 관한 것.(혼합형 연산시 어떤 연산, 상수가 10진법 수라는 것, +가 덧셈 연산자라는 의미)
  - **언어 구현 시간**(컴파일러 설계 시간) - 한 언어에서 허용되는 정수의 자릿수, 실수의 유효숫자 개수, 수의 기계 내에서의 표현법
  - **컴파일 시간** – 정적(static) 바인딩으로 컴파일 시간
    - ✓ 프로그래머에 의해서 선택된 바인딩 – 명시적 선언(explicit declaration)시 변수 이름, 변수에 대한 type, 프로그램 문장 구조
    - ✓ translator에 의해서 선택된 바인딩 - 묵시적 선언(implicit declaration)-coercion 설명, 어떻게 array들이 저장되고 어떻게 array에 대한 descriptor가 만들어지는가?
  - **적재(loading) 시간** - actual parameter의 실제 address할당(ALGOL, PASCAL의 global 변수의 기억 장소 할당)

5.4 바인딩 개념

- **실행시간** - 동적 바인딩
  - ✓ 모듈(subprogram or block)의 시작 시간 - C와 PASCAL에서 실매개변수를 형식 매개변수로 바인딩
  - ✓ - C와 PASCAL에서 형식 매개변수 혹은 지역 변수(local variable)에 대한 기억 장소 할당(storage allocation)
  - ✓ 실행 시간의 사용 시점 - 배정문에서 값을 변수에 바인딩(APL, LISP, ML과 같은 인터프리터 언어들은 변수의 기억 장소 할당)

❖ **예제** : int count; /* C 나 Jave 문장 */

....

count = count + 5;

- ✓ count의 자료 형 - 컴파일 시간
- ✓ count에 대한 가능한 자료 형의 집합 - 언어정의시간:
- ✓ count의 가능한 값들의 집합 – 언어구현시간(컴파일러 설계 시간)
- ✓ count의 값 – 배정문의 실행시간
- ✓ '+'가 더하기 연산자라는 의미 – 언어 정의 시간
- ✓ 연산자 '+'에 대한 가능한 의미의 집합(정수의 연산자인지? 실수의 연산자인지? 문자열의 연산자인지?) – 컴파일 시간
- ✓ 리터럴 5의 내부 표현 - 언어구현시간 (컴파일러 설계 시간)
- ✓ 5가 십진법의 수라는 것 – 언어 정의시간

5.4 바인딩 개념

- **바인딩의 종류**
  - **정적 바인딩** - 바인딩이 실행시간 전에 일어나고, 실행 중에 변하지 않은 상태로 유지된다.
  - **동적 바인딩** - 바인딩이 실행시간 중에 일어나거나, 실행 중에 변경될 수 있다.
- **자료 형 바인딩**
  - 고려 사항
    - ✓ 자료 형이 어떻게 명세되는가?, 자료 형 바인딩이 언제 일어나는가?
  - 변수의 정적 자료 형 바인딩
    - ✓ 자료 형이 명시적(explicit) 혹은 묵시적(implicit)으로 명세
    - ✓ 명시적 선언은 변수의 자료 형을 선언하기 위해서 사용되는 문장.
      - ♣ 1960년대 중반 이후 설계된 언어들은 대부분 명시적 선언문 요구
    - ✓ 묵시적 선언은 변수의 자료 형을 명세하는 디폴트 규칙
      - ♣ 컴파일러나 인터프리터에 의해 바인딩이 이루어짐
        - Fortran에서, 식별자가 *i, j, k, l, m, n*으로 시작되면 integer 타입이고, 그렇지 않으면 real 타입으로 묵시적으로 선언
      - ♣ 약간의 편리성을 제공하지만, 철자 오류나 프로그래머 오류 탐지가 어렵고, 우연히 선언하지 않은 변수에 예기치 않은 속성 부여 가능 – 신뢰성 떨어짐
      - ♣ 묵시적 문제점을 특정 타입의 이름을 특정 특수 문자로 시작하게 해서 해결.
        - Perl에서, 스칼라 변수는 $로 시작되고, 배열은 @로 시작되고, 해시 구조체는 %로 시작된다.

5.4 바인딩 개념

- 동적 자료 형 바인딩
  - ✓ 변수에 대한 자료 형 바인딩은 선언문이 아닌 배정문에서 값이 할당되는 것에 의해 바인딩.
    - ♣ 배정문이 실행될 때, 좌측 변수는 우측의 값, 변수, 또는 식의 자료 형에 바인딩 된다.
  - ✓ 유연성 제공(generic 프로그램 작성 가능)
    - ♣ 부프로그램의 매개변수가 어떠한 타입의 값도 받아들일 수 있다는 것을 의미
    - ♣ 어떠한 데이터가 들어와도 그 데이터가 저장될 변수는 그 데이터가 입력 후 그 변수에 할당될 때 올바른 타입으로 할당.
  - ✓ 오류 탐지 능력 저하 / 실행시간 증가
    - ♣ JavaScript 프로그램에서,
    - ♣ list = [10.2, 3.5] ; - list가 길이가 2인 1차원 배열에서
    - ♣ ⋮
    - ♣ list = 47 - 스칼라 변수로 바뀜
  - ✓ Phthon, Ruby, Javascript, PHP

5.4 바인딩 개념

- ✓ 단점 –
  - ♣ 1)프로그램이 덜 신뢰적이다. – 컴파일러의 오류 탐지 능력이 떨어짐
    - 동적 바인딩은 임의의 변수에 임의의 자료 형의 값이 할당되는 것을 허용한다. 그래서 우측 상의 잘못된 자료 형이 오류로서 탐지되지 않고, 오히려 좌측 상의 자료 형이 잘못된 자료 형으로 단순히 변경된다.
  - ♣ 2) 구현하는 비용이 많이 들어간다. – 또한 실행 시간에 실행 시간 서술자도 필요하다.
  - ♣ 3)실행 시간 이 적어도 10배 이상의 시간이 소용된다.
    - 인터프리터로 구현되면 컴파일 시간에 a+b 에 대해서 a와 b에 대한 자료형이 컴파일 시간에 알려지지 않으므로 a+b에 대한 기계 명령어를 생성할 수 없다.

5.4 바인딩 개념

- **타입 검사**(type checking)는 연산자의 피연산자들이 호환 가능한 타입인지를 확인하는 행위
  - 배정문의 경우, 좌변과 우변이 피연산자
  - **호환 가능 타입**(compatible type)이란 연산자에 대해서 적법하거나 컴파일러에 의해서 적법한 타입으로 묵시적으로 변환 가능한 타입
    - ✓ 두 변수가 호환 가능 타입이면, 한 변수의 값을 다른 변수에 저장할 수 있다는 의미
  - **타입 강제 변환**(coercion)이란 호환 가능 타입을 적법한 타입으로의 자동 변환
  - **타입 오류**는 부적절한 타입의 피연산자에 연산자를 적용했을 때 발생한다.
    - ✓ 정적 타입 검사
      - ♣ 모든 타입 바인딩이 정적이면, 거의 모든 타입 검사가 정적 가능
      - ♣ 컴파일 시간 오류 탐지 가능
    - ✓ 동적 타입 검사
      - ♣ 타입 바인딩이 동적이면, 타입 검사는 동적이여야 한다.
      - ♣ 프로그램 유연성 증가
      - ♣ JavaScript 등
  - 프로그램 실행시간 중에 한 개의 메모리 셀에 다른 시간에 다른 타입의 값이 저장되는 것이 허용될 때는 동적 타입 검사가 필요
    - ✓ C의 공용체, Fortran의 Equivalence

5.4 바인딩 개념

- 컴파일러 시큐리티 –
  - 소스 코드의 정적 분석을 통해 보안 취약점 등을 검출하는 것
  - 주로 버퍼 보안 검사 – 함수의 반환 주소, 예외 처리기의 주소, 형식 매개 변수를 덮어쓰는 일부 버퍼 오버런

- **기억 공간 바인딩과 존속 기간**
  - 기억 장소 할당(allocation)
    - ✓ 가용 기억 공간으로 부터 셀을 가져온다.
  - 기억 장소 회수(deallocation)
    - ✓ 셀을 가용 기억 공간에 다시 돌려준다.
  - 변수의 존속기간(lifetime)
    - ✓ 변수가 특정 메모리 셀에 바인딩되어 있는 시간
    - ✓ 존속기간에 따른 변수의 분류
      - ♣ 정적(static)
      - ♣ 스택-동적(stack-dynamic)
      - ♣ 명시적 힢-동적(explicit heap-dynamic)
      - ♣ 묵시적 힢-동적(implicit heap-dynamic)

5.4 바인딩 개념

- 정적 변수(static variable)
  - ✓ 변수가 실행 전에 메모리 셀에 바인딩되고, 그 바인딩이 실행 종료시까지 유지.
    - ♣ 장점:
      - 매우 쉽게 구현
      - 효율성(efficiency)
      - 할당/회수에 따른 실행 시간 부담이 없다.
    - ♣ 단점:
      - 유연성 감소
      - recursion이 허용되지 않음
    - ♣ 예제 :
      - Fortran IV의 모든 변수, C, C++, Java의 static 변수
- 스택 동적 변수(stack-dynamic variable)
  - ✓ 기억 장소는 실행시간 스택으로 부터 할당되고, 제어를 호출자에 반환할 때 회수된다.
    - ♣ 장점:
      - 재귀 함수 허용
      - 변수들간에 기억 장소 공유

5.4 바인딩 개념

- ♣ 단점:
  - 할당과 회수에 따른 실행 시간 부담
  - 간접 주소 지정이 요구되기 때문에 가능한 더 느린 접근
    - 간접주소지정 방식은 주소가 operand의 값이 아닌 다른 주소를 가지고 있는 것
- ♣ 예제 :
  - C, C++, java의 지역변수

➢ 명시적 힢-동적 변수(explicit heap-dynamic variable)

✓ 프로그래머가 명세하는 명시적 실행시간 명령어에 의해서 할당되고 회수되는 이름이 없는 추상 메모리 셀

- ♣ 이러한 변수는 힙으로 부터 할당되고 회수되며, 단지 포인터나 참조 변수를 통하여 참조될 수 있음.
- ♣ 예제: 다음 C++ 코드에서,
  - int * intnode; // 포인터 생성
  - intnode = new int; // 힢-동적 변수 생성, 포인터 변수가 참조
  - …
  - delete intnode; // 힢-동적 변수 반환
- ♣ C 언어의 malloc, free
- ♣ Java에서,
  - 객체는 명시적 힢-동적 변수이고, 참조 변수를 통해서 접근
  - 힢-동적 변수 반환은 – garbage collection
- ♣ 동적 구조체(연결리스트나 트리)에 유용하게 사용되나, 포인터나 참조 변수의 올바른 사용 어려움과 실행시간 비용 초래

5.4 바인딩 개념

- 묵시적 힢-동적 변수
  - ✓ 할당과 회수가 배정문에 의해서 일어난다.
    - ♣ 값을 배정받을 경우에만 힢 기억 장소에 바인딩
    - ♣ 변수의 모든 속성은 이때 바인딩된다.
    - ♣ 장점 :
      - 고도의 유연성
    - ♣ 단점 :
      - 비효율적, 컴파일러의 오류 탐지 능력 손실
    - ♣ 예 :
      - Perl, JavaScript의 문자열과 배열

5.5 영역

- 변수의 **영역**(scope)은 변수가 가시적인 문장의 범위 즉, 그 이름의 사용이 허락되고 있는 프로그램의 범위
  - 가시적(visible) – 변수가 어떤 문장에서 참조될 수 있으면
  - 지역적(local) – 변수가 프로그램 단위나 블록 내부에 정의 된 경우
  - 비지역(non-local) – 프로그램 단위나 블록 내부에서 가시적이나 그곳에 선언되지않은 변수
  - 전역(global) -
- **정적 영역**
  - 변수의 영역이 실행 전에 결정
  - 프로그램 단위 정의에 기반(부프로그램)
    - ✓ 부프로그램은 자신의 영역 생성
    - ✓ 부프로그램 내부에 다른 부프로그램 중첩 가능(JavaScript, Pascal, Ada 등)
  - 변수 참조에 대해서 그 변수와 연관된 선언문을 찾는다:
    - ✓ 먼저 지역적인 선언문을 탐색한다.
    - ✓ 발견되지 않으면 해당 선언문이 찾아질 때까지 점차적으로 지역 영역을 포함한 더 큰 영역에서 찾아본다.
    - ✓ 영역을 포함한 더 큰 영역을 정적 조상(static ancestor)라 하며, 가장 가까운 정적 조상을 정적 부모(static parent)라 한다.

5.5 영역

❖ 정적 영역 예 : JavaScript 함수인 big

```
Function big(){
  function sub1(){
      var x = 7;
       sub2();
  }
  function sub2(){
        var y=x;  //x는 big 에 선언된 x의 값을 참조
  }
  var x=3;
  sub1();
}
```

❖ 정적 영역과 x의 정적 조상(정적으로 유효한 것)

- Sub1이 sub2의 정적 조상이 아니다
- X는 sub1과 big에서 선언
- Sub1에서 선언된 x는 지역 변수

5.5 영역

- 정적 영역의 예 : ALGOL

```
1. A : begin integer i, j; real x, y;
2. B : procedure test(integer a, b)
3.         begin boolean i;
4.             ..............
5.              x: = i * j + y ;
6.             ..............
7.           end B;
8.         ..............
9.  C : begin integer x, y; real i, j;
10.             ........
11. D : begin boolean j;
12.             .......
13.              call test(x, y);
14.             .......
15.               end D;
16.            .........
17.  end C;
18.   ....
19. end A;
```

5.5 영역

- 정적 영역의 예

변수 / 선언된 자료 형	i	j
Integer	1,2,8,18~19	1~8, 18~19
Boolean	3~7	11~15
real	9~17	9~10, 16~17

변수 / 선언된 자료 형	x	y
integer	9~17	9~17
real	1~8,18~19	1~8,18~19

- 영역 구멍(hole-in-scope) : 블록 c에서 i, j, x, y가 재선언 되었을 경우에 블록 c에서 벗어날 때까지 이 변수들이 이전에 선언된 속성은 무효가 됨. 이런 현상을 영역 구멍

5.5 영역

- 블록
  - 실행 코드 중간에 정적 영역을 생성하는 수단
    - ✓ 자신의 지역 변수를 갖는 코드 부분
    - ✓ 이러한 변수는 스택-동적
    - ✓ 기억 공간은 실행이 코드 부분에 진입될 때 할당되고 코드 부분을 빠져나올때 회수된다. – 이러한 코드의 부분을 블록(block)
    - ✓ 블록 구조 언어(block-structured language)
    - ✓ C-기반 언어는 복합문(중괄호에 둘러싸인 일련의 문장들)이 선언문을 갖는 것을, 그래서 새로운 영역을 정의하는 것을 허용
- 동적 영역
  - 프로그램 단위의 호출 순서에 기반
  - APL, SNOBOL4, LISP의 초기 버전은 변수의 영역은 동적이다.
  - 변수에 대한 참조는 부프로그램 호출들의 체인을 역순으로 탐색하면서 선언문을 연결시킨다.
    1. 먼저 지역 선언을 탐색
    2. 다음에 동적 부모의 선언문을 탐색
    3. 선언문을 찾을 때까지 2의 과정을 반복

5.5 영역

- 동적 영역 예 : JavaScript 함수인 big

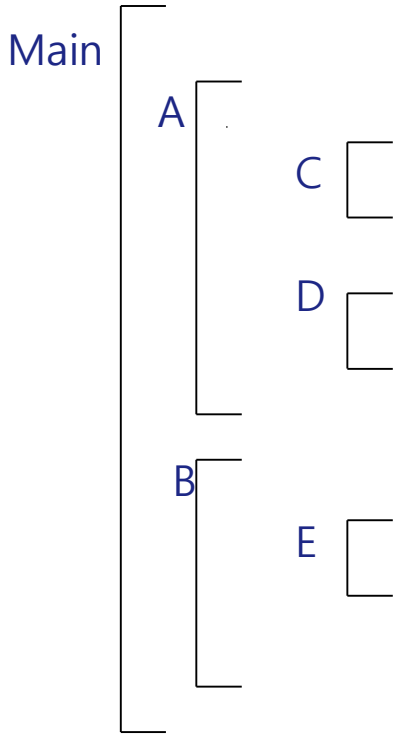
```
Function big(){
  function sub1(){
      var x = 7;
  }
  function sub2(){
        var y=x;  //x는 big 에 선언된 x의 값을 참조
  }
  var x=3;
  sub1();
}
```

- Sub2에서 참조되는 식별자 x는 동적이다.
  - 컴파일 시간에 결정될 수 없다. 그 식별자는 호출 시퀀스에 따라 x의 어느 한 선언에 속한 변수를 참조
  - 1)big이 sub1을 호출하고 sub1은 sub2를 호출한다.
    - ✓ 이 경우에 sub2에서 x에 대한 참조는 sub1에서 선언된 x
  - 2)sub2가 big으로 부터 직접 호출된다
    - ✓ 이 경우에 sub2의 동적 부모는 big이고 그 참조는 big에서 선언된 x이다.

5.5 영역

❖ 영역 평가

❖ 프로그램 구조(정적 영역)



- E가 D 영역에 속한 변수에 접근해야 한다면?
- E가 D를 호출하는 것이 필요해질 경우

5.5 영역

- 영역 평가
  - 정적 영역
    - ✓ 장점
      - ♣ 높은 가독성과 높은 신뢰성
      - ♣ 효율성
    - ✓ 단점
      - ♣ 프로그램 수정 과정에서, 전역 변수 사용이 조장 되고, 낮은 중첩 수준의 프로시저 조장 경향
      - ♣ 최종적 설계가 어색, 자연스럽지 못할 가능성
  - 동적 영역
    - ✓ 단점:
      - ♣ 신뢰성 저하 : 부프로그램의 지역변수는 실행중인 다른 모든 부프로그램에 가시적이다.
      - ♣ 비지역변수에 대한 타입 검사를 정적으로 수행할 수 없다.
      - ♣ 가독성 저하 : 비지역 변수 참조는 호출 순서에 기반
      - ♣ 비지역변수 접근에 더 많은 시간 소요
    - ✓ 장점:
      - ♣ 편리성: 매개변수를 명시적으로 전달할 필요 없다.