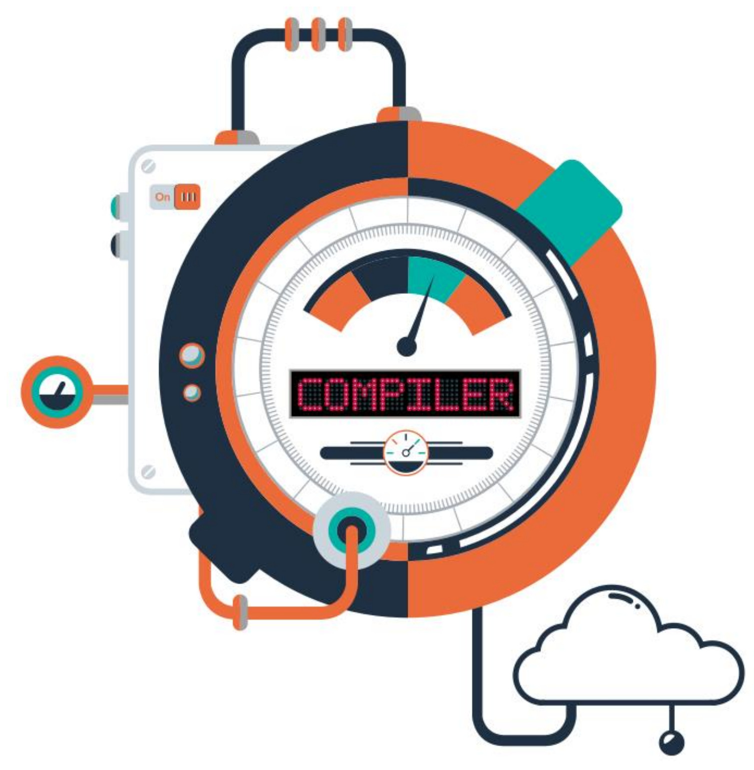


내공 있는 프로그래머로 길러주는

컴파일러의 이해

Chapter 09
구조적 자료형과 실행 시간 환경

목차

01 구조적 자료형

02 메모리 구성

03 메모리 할당 전략

04 매개변수 전달 방법

학습목표

- 레코드, 배열 등 구조적 자료형에 대해 이해할 수 있다.
- 메모리 구성에 대해 이해할 수 있다.
- 정적 메모리 할당, 스택 메모리 할당, 힙 메모리 할당 등 메모리 할당 전략에 대해 이해할 수 있다.
- 값 호출, 참조 호출, 이름 호출, 값-결과 호출 등 매개변수 전달 방법에 대해서 이해할 수 있다.

9.1 구조적 자료형

- 실행 시 필요한 모든 환경인 실행 시간 환경에 대해 설명한다. 실행 시간 환경은 매우 다양한데, 구조적 자료가 어떻게 구현되는지 구조적 자료형과 메모리 구성 및 메모리 할당 전략을 살펴보고, 변수 및 데이터가 어떻게 접근하는지 매개변수 전달 방법에 대해서도 다룬다
- **구조적 자료형(structured data type)**과 **기본 자료형(elementary data type)**에 대해 알아보자.
  - 기본 자료형은 하나의 이름이 하나의 자료 객체(data object)를 가진 것으로 정수, 실수, 문자형을 말한다.
  - 구조적 자료형은 하나의 이름에 여러 개의 자료 객체가 있는 것으로 레코드, 배열, 문자열, 집합, 트리 등을 말한다.
- **레코드**
  - 많은 언어에서는 이질(heterogeneous)의 자료 개체들을 하나로 묶고 여기에 그룹 이름을 부여하여 하나의 자료형을 선언할 수 있는데, 이러한 형태의 구조적 자료형을 레코드 또는 구조체라고 한다.
  - PL/I, 코볼, 파스칼, C, 알골 등과 같은 언어는 레코드를 사용할 수 있는데 레코드의 표현은 언어마다 조금씩 다르다.
  - 예를 들어 코볼 언어의 레코드는 [그림 9-1]과 같다.

9.1 구조적 자료형

```
01 ADDRESS.
 02 PERSON.
    03 GIVEN-NAME  PIC  X(15).
    03 FAMIL-NAME  PIC  X(20).
 02 STREET.
    03 NAME        PIC  X(20).
    03 HOUSE-NUM   PIC  X(3).
 02 CITY.
    03 ZIP-NUM      PIC  X(4).
    03 NAME         PIC  X(20).
```

그림 9-1 코볼 언어의 레코드

- [그림 9-1]에서 레코드 ADDRESS는 PERSON, STREET, CITY와 같은 그룹 항목으로 되어 있고, 이것들은 다시 하나 이상의 기본 자료형으로 구성되어 있다. X(n)은 자료의 속성을 나타내는 PIC 구로 n바이트로 구성된 자료의 길이이다.
- 코볼 언어에서 레코드에 대한 일반적인 구문은 계층 번호(level number)와 자료 이름(data name), 자료의 형과 길이에 따른 속성(attribute)에 의해 다음과 같이 표현된다.
- $level_num_1$ $data_name_1$ PIC $attribute_1$

9.1 구조적 자료형

- [그림 9-1]에서 레코드 ADDRESS는 PERSON, STREET, CITY와 같은 그룹 항목으로 되어 있고, 이것들은 다시 하나 이상의 기본 자료형으로 구성되어 있다. X(n)은 자료의 속성을 나타내는 PIC 구로 n바이트로 구성된 자료의 길이이다.

- 메모리에서 레코드가 어떻게 저장되는지 살펴보자.
  - 자료형의 속성으로 서술된 이름(기본 항목)은 자신의 기억 공간을 필요로 한다. 그러므로 레코드에서 속성을 지닌 모든 자료 항목은 메모리에서 장소를 연속으로 할당 받음.
  - [그림 9-1]의 레코드 ADDRESS에 대한 메모리 할당과 자료 이름의 관계
  -

그림 9-2 레코드 ADDRESS에 대한 메모리 할당

9.1 구조적 자료형

- [그림 9-2]의 레코드 ADDRESS에 대한 메모리 할당에서 만약 α가 레코드 ADDRESS의 시작 주소라고 할 때, 각 항목의 자료 길이를 바이트로 계산하면 다음과 같다.
-

표 9-1 [그림 9-2]에 대한 시작 주소

이름	이름에 대한 주소	길이(바이트)
ADDRESS	α	82
PERSON	α	35
GIVEN-NAME	α	15
FAMIL-NAME	α+15	20
STREET	α+35	23
NAME IN STREET	α+35	20
HOUSE-NUM	α+55	3
CITY	α+58	24
ZIP-NUM	α+58	4
NAME IN CITY	α+62	20

9.1 구조적 자료형

■ 배열

- 동질(homogeneous)의 자료 객체들을 묶고 여기에 그룹 이름을 부여하는 배열
- 프로그램을 실행하는 동안 특정한 원소를 상대 주소로 직접 접근할 수 있도록 허용
- C 언어에서 배열이 다음과 같이 선언될 때, 배열 score는 정수형 자료 100개를 메모리에 연속적으로 기억시킬 수 있는 기억 공간을 필요로 한다.
  - int score [100];
- 배열의 요소들은 연속적으로 저장되면 빠르게 접근할 수 있어 실행 시간이 단축된다. 다음과 같이 가정하고 위치를 계산해보자.
  - 각 배열 요소의 크기를 u라고 하면 배열 A의 i번째 요소는 다음 위치에서 시작한다.
  - base + (i - low + 1) × w ... (9.1)
  - 여기서 low는 배열 첨자의 하한 값이며 1이라고 하자(C 언어의 경우 하한 값이 0이므로 주소 계산이 달라진다). base는 배열에 할당된 메모리의 상대 주소이다. 즉 base는 A[low]의 상대 주소이다. 또한 w는 각 요소의 크기이다. 식 (9.1)을 식 (9.2)로 변환해보자.
  - (i + 1) × w + (base - low × w) ... (9.2)
  - 식 (9.2)는 컴파일 시간에 부분적으로 실행될 수 있다. 부분식 d = base - low × w는 배열의 선언이 나타나면 바로 계산될 수 있다. 즉 배열 A에 대한 기호표 항목 안에 d의 값이 계산 및 저장되면 A[i]의 상대 주소는 단순히 (i + 1) × w를 d에 더함으로써 계산된다.

- 같은 방법으로 다차원 배열을 다음과 같이 선언한다.
- int board[2][3];
- board는 각각 원소가 연속된 메모리 영역을 할당하지만 board[1][2]가 자료 영역에서 몇 번째 셀에 해당되는지 쉽게 알 수 없다. 그러나 논리적인 배열의 구조는 메모리에서 1차원 배열로 정돈되기 때문에 손쉽게 상대적인 위치를 계산해낼 수 있는 반면에 다차원 배열은 언어에 따라 하한 값과 상한 값이 달라질 수 있다. C 언어에서는 하한 값으로 0이 고정되어 있으나 파스칼에서는 하한 값이 음수 또는 임의의 정수로 서술될 수 있다. 또한 포트란 언어는 하한 값이 1로 고정되어 있다.
- 2차원 배열을 메모리에 저장하기 위해 1차원 배열로 변환하는 방법은 행 우선(row-major), 열 우선(column-major), 슬라이스(slice) 등 여러 가지 방법이 있다. [그림 9-3]은 행 우선 및 열 우선 방법으로 저장된 2×3 배열 A의 내용을 보여준다. 슬라이스는 배열에서 어떤 부분의 구조이다.

9.1 구조적 자료형

그림 9-3 2차원 배열의 메모리 저장 형태

- 다차원 배열을 1차원 배열로 저장하는 방법
  - 언어마다 사용하는 방법이 다르므로 이런 특성을 반영한다면 매우 효율적인 프로그래밍이 가능할 것이다. C, 파스칼, PL/I 등은 행 우선 방법을 사용하고, 포트란은 열 우선 방법을 사용한다.
  - 행 우선 방법으로 저장된 2차원 배열의 경우 A[i, j]의 상대 주소는 식 (9.3)과 같다.
  - base + (((i - low1) × n2) + (j - low2 + 1)) × w ... (9.3)

9.1 구조적 자료형

- 여기서 low1과 low2는 i와 j의 하한 값이고, n2는 열의 크기로 j가 취할 수 있는 값의 크기이다. 즉 high2를 j에 대한 상한 값이라고 하면 n = high2 - low2 + 1이다. A[i, j]의 상대 주 소를 계산하는데 컴파일 시간에 i와 j의 값만 모르고 다른 값들을 모두 안다면 식 (9.3)은 식 (9.4)와 같이 다시 쓸 수 있다.
- ((i × n) + (j + 1)) × w + (base - ((low1 × n) + low2) × w) … (9.4)
- 식 (9.4)의 뒷부분은 컴파일 시간에 계산할 수 있다. 같은 방법으로 열 우선에 대해서도 2차 원 배열의 경우 A[i, j]의 상대 주소는 식 (9.5)와 같다.
- base + (((j - low1) × n1) + (i - low2 + 1)) × w … (9.5)
- n1은 행 단위의 원소 개수로 i가 취할 수 있는 값의 크기이다.

9.1 구조적 자료형

■ [예제 9-1] 행우선과 열우선에 따른 주소 계산하기

- 다음과 같이 선언된 2차원 행렬에 대해 하한 값을 1로 하여 행 우선과 열 우선으로 메모리에 저장하고, 행 우선과 열 우선에 대해 B(3,2)의 주소를 계산해보자.
- int B(4,4);
- **[풀이]**
  - 행 우선 방법으로 메모리에 저장하면 다음과 같다.

  - 열 우선 방법으로 메모리에 저장하면 다음과 같다.

B(1,1)	B(2,1)	B(3,1)	B(4,1)	B(1,2)	B(2,2)	B(3,2)	B(4,2)	B(1,3)	B(2,3)	B(3,3)	B(4,3)	B(1,4)	B(2,4)	B(3,4)	B(4,4)

↑ (B(3,2))

9.1 구조적 자료형

- 원소 B(3,2)에 대한 상대 주소를 계산한다. 그림의 화살표를 보면 행 우선 방법은 열 번째이고 열 우선 방법은 일곱 번째임을 알 수 있다. 또한 각 원소가 정수형이므로 4바이트를 할당한다면 주소는 base로부터 행 우선의 경우 40, 열 우선의 경우 28이어야 한다.
- 식 (9.3)에 의하면 i는 3, j는 2, low1은 하한 값이므로 1, low2도 1, n2는 열의 크기이므로 4, w는 4이므로 다음과 같다.
- base + (((i - low1) × n2) + (j - low2 + 1)) × w
- = base + (((3 - 1) × 4) + (2 - 1 + 1)) × 4
- = base + (8 + 2) × 4
- = base + 40
- 같은 방법으로 열 우선으로 계산하기 위해 식 (9.5)를 사용한다. 여기서 i는 3, j는 2, low1은 하한 값이므로 1, low2도 1, n1은 행의 크기이므로 4, w는 4이므로 다음과 같다.
- base + (((j - low1) × n1) + (i - low2 + 1)) × w
- = base + (((2 - 1) × 4) + (3 - 1 + 1)) × 4
- = base + (4 + 3) × 4
- = base + 28

9.2 메모리 구성

- 프로시저가 실행되는 것을 프로시저의 활성(activation)
  - 만약 어떤 프로시저가 재귀적(recursive)이라면 몇 개의 활성이 동시에 존재할 수 있다.
- 프로시저 정의(procedure definition)는 가장 간단한 형태로 식별자와 문장을 연관시키는 선언이다.
  - 식별자는 프로시저 이름이고, 문장은 프로시저 몸체(body)이다.
  - 예를 들어 [그림 9-4]의 퀵 정렬(quick sorting) 프로그램을 생각해보자.

9.2 메모리 구성

```
① program sort(input, output);
②   var a : array[0..10] of integer;
③   procedure readarray;
④     var i : integer;
⑤     begin
⑥       for i := 1 to 9 do read(a[i])
⑦     end ;
⑧   function partition(y, z : integer) : integer;
⑨      var i, j, x, v : integer;
⑩      begin ...
⑪      end;
⑫   procedure quicksort(m, n : integer);
⑬      var i : integer;
⑭      begin
⑮        if (n > m) then
⑯                  begin
⑰                     i := partition(m, n);
⑱                     quicksort(m, i-1);
⑲                     quicksort(i+1, n)
⑳                  end
㉑      end;
㉒   begin
㉓      a[0] := -9999;
㉔      a[10] := 9999;
㉕     readarray;
㉖     quicksort(1,9)
㉗   end.
```

그림 9-4 퀵 정렬 프로그램

9.2 메모리 구성

IT COOKBOOK

- [그림 9-4]의 ③~⑦행은 readarray라는 프로시저 정의를 하고 있으며, 여기서 ⑤~⑦행은 프로시저 몸체이다. 프로시저가 호출되었다는 것은 프로시저 이름이 실행 가능한 문장에 나타난 경우이다. 22~27 행은 주프로그램이며, 25행에서 프로시저 readarray를 호출하고 26행에서 프로시저 quicksort를 호출한다.
- 프로시저 정의에서 사용되는 식별자에는 매개변수가 있다. 매개변수는 형식 매개변수와 실 매개변수가 있는데, [그림 9-4]에서는 ⑫행의 식별자 m과 n이 프로시저 quicksort의 형식 매개변수이고, ⑱행의 m과 i-1이 실 매개변수이다. 즉 프로시저를 호출하는 데 있는 매개변수가 실 매개변수이고, 프로시저 호출을 받는 데 있는 매개변수가 형식 매개변수이다.
- 프로그램이 실행되는 동안 프로시저 사이에는 제어의 흐름이 존재한다. 제어는 순차적으로 이동하는데, 프로시저는 몸체의 처음부터 실행되다가 모든 실행이 끝나면 프로시저가 호출된 지점 바로 다음 위치에 제어를 돌려준다.
- 프로시저 p에 대한 존속 시간(life time)은 프로시저 몸체가 실행되는 기간을 말한다. 존속 시간에는 p가 다시 프로시저를 호출하여 그 프로시저를 실행하는 시간도 포함된다.
- [그림 9-4]의 프로그램을 실행시킨 결과는 [그림 9-5]와 같다. 여기서 partition(1,9)가 반환되는 값은 4라고 가정하며, 활성 quicksort(1,9)의 존속 시간은 enter quicksort(1,9)와 leave quicksort(1,9) 사이이다.

9.2 메모리 구성

-
```
enter main()
enter readarray
leave readarray
enter quicksort(1,9)
enter partition(1,9)
leave partition(1,9)
enter quicksort(1,3)
 ⋮
leave quicksort(1,3)
enter quicksort(5,9)
 ⋮
leave quicksort(5,9)
leave quicksort(1,9)
leave main().
```

그림 9-5 [그림 9-4] 프로시저의 출력 결과

9.2 메모리 구성

- 만약 어떤 프로시저의 새로운 활성이 그 프로시저의 이전 활성이 끝나기 전에 시작된다면 그 프로시저를 재귀적이라고 한다. quicksort 프로시저는 재귀적이다.
- 제어가 활성에 들어가고 다시 나오는 과정을 트리로 표현한 것을 활성 트리(activation tree)라고 한다. 활성 트리의 각 노드는 하나의 활성을 나타내고, 루트 노드는 전체 프로그램을 시작하는 주프로그램의 활성이다. 프로시저 p의 자식 노드는 프로시저 p가 호출하는 활성 노드이다. 활성은 왼쪽에서 오른쪽으로 호출되며 자식 노드는 자신의 오른쪽에 있는 활성이 시작되기 전에 끝내야 한다.

[예제 9-2] 프로시저 호출과 반환에 대한 활성 트리 그리기

- [그림 9-5]에 있는 프로시저 호출과 반환에 대한 활성 트리를 그려보자.
- [풀이] 각 프로시저는 프로시저의 첫 번째 글자로만 나타낸다. 활성 트리의 루트 노드는 주프로그램인 main을 나타낸다. main이 실행되는 동안 루트 노드의 첫 번째 자식 노드로 라벨이 r이라 붙은 readarray의 활성이 존재한다. 루트 노드의 두 번째 자식 노드는 quicksort(1,9)이다. 같은 방법으로 활성 트리를 만들어갈 수 있다. 활성 q(1,3)과 q(5,9)는 재귀적 프로시저이며, q(1,3)과 q(5,9)는 q(1,9)가 끝나기 전에 시작되고 q(1,9)보다 먼저 끝마쳐야 한다. 전체적인 활성 트리는 [그림 9-6] 과 같다. 여기서 m은 main, r은 readarray, q(1,9)는 quicksort(1,9), p(1,9)는 partition(1,9)를 나타낸다.

9.2 메모리 구성

m

r q(1,9)

p(1,9) q(1,3) q(5,9)

p(1,3) q(1,0) q(2,3) p(5,9) q(5,5) q(7,9)

p(2,3) q(2,1) q(3,3) p(7,9) q(7,7) q(9,9)

그림 9-6 [그림 9-5]의 프로시저에 대한 활성 트리

- 프로시저의 호출과 반환은 제어 스택control stack이라고 불리는 실행 시간 스택이 관리한다. 즉, 활성이 시작될 때 활성에 대한 노드를 스택에 삽입(push)하고 활성이 끝날 때 그 노드를 삭제(pop)한다. 그러므로 제어 스택의 내용은 활성 트리의 루트 노드에 대한 경로와 관계 있는 것이다.

[예제 9-3] 제어 스택 그리기

- [그림 9-6]의 q(2,3)에 대한 제어 스택을 그려보자.
- 루트로부터 q(2,3)으로 가는 경로를 나타낸 [그림 9-7]에서 이미 활성이 수행된 것은 점선으로 표시하고, 루트 m으로부터 q(2,3)으로 가는 경로는 실선으로 표시했다. 제어 스택은 실선으로 표시된 경로에 해당하는 노드를 저장하고 있다. 즉 제어 스택에는 다음과 같이 저장되어 있다.
- m, q(1,9), q(1,3), q(12,3)

9.2 메모리 구성

▪

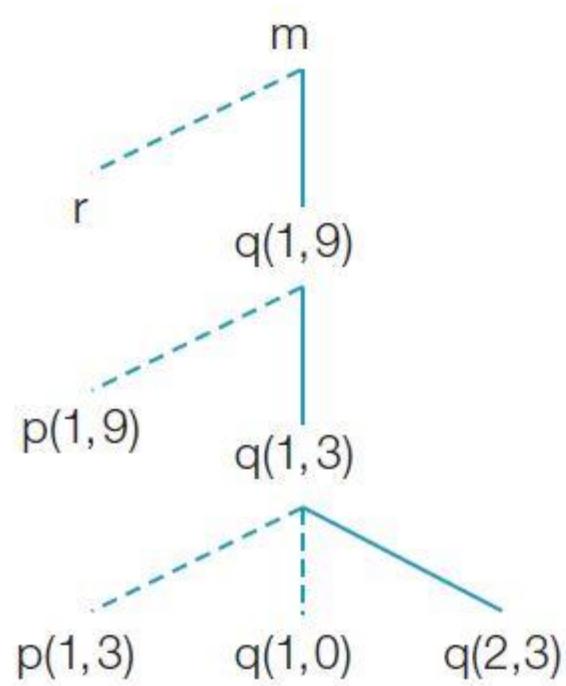


그림 9-7 루트로부터 q(2,3)으로 가는 경로

메모리 구성

- 전형적인 컴퓨터의 메모리 구성은 [그림 8-8]과 같이 메모리 코드 부분과 데이터 부분으로 더욱 세분할 수 있다.
  - 대부분의 컴파일 언어에서는 실행하는 동안에 코드 부분을 변경할 수 없으며, 코드와 자료 부분이 개념적으로 분리되어 있다. 더욱이 코드 부분은 실행 이전에 고정되기 때문에 모든 코드에 대한 주소는 컴파일 시간에 알 수 있다. 특히 각 프로시저와 함수의 시작점(entry point)은 컴파일 시간에 알 수 있다. 실행할 때의 코드 메모리 구성은 [그림 9-8]과 같다.

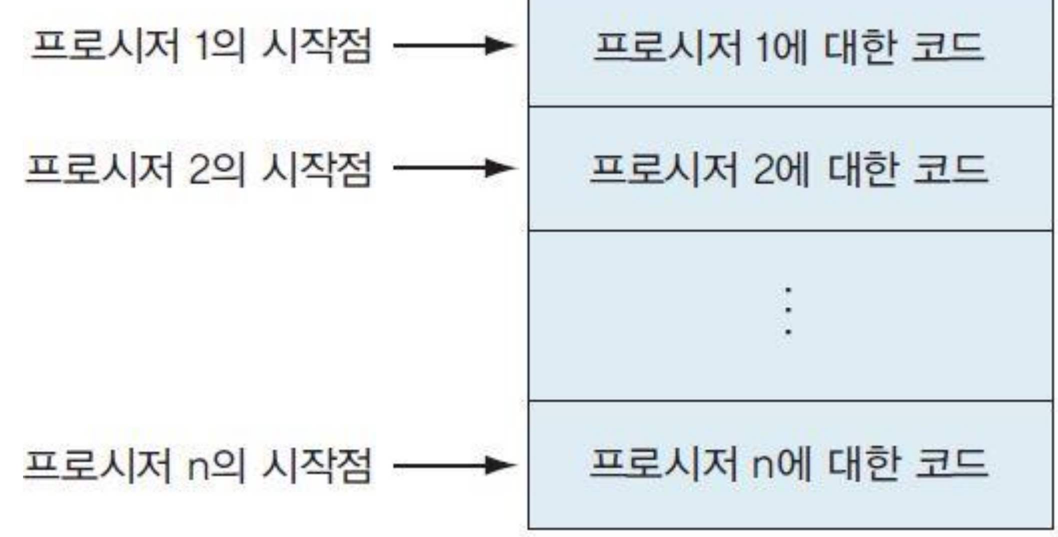


그림 9-8 실행 시의 코드 메모리 구성

9.2 메모리 구성

- 프로그램에서 전역 및 정적 자료는 일반적으로 코드와 유사한 형태로 고정 부분에 별개로 할당된다. C 언어의 외부 및 정적 변수와 파스칼 언어의 전역 변수가 이러한 유형에 속한다.
- 동적 자료 할당에 사용되는 메모리 부분은 여러 가지 다른 방법으로 구성될 수 있다. 전형적인 구조는 이 부분에 대한 메모리가 스택 부분과 힙 부분으로 나뉘는데, 스택 부분은 할당이 LIFO(last-in, first-out) 방식으로 발생하는 자료에 사용되고, 힙 부분은 LIFO 방식을 따르지 않는 동적 할당(예를 들어 C에서 포인터 할당)에 사용된다.
- 힙은 일반적으로 단순한 선형 메모리 부분이라는 것을 주목해야 한다. 포인터 할당 및 해제를 처리하는 데는 어떤 동적인 장치가 필요한데 그러한 할당을 처리하는 자료 구조가 바로 힙이다.
- 지금까지 설명한 실행 시간 메모리의 일반적인 구조는 [그림 8-8]과 같다. 좀 더 부연 설명을 하기 위해 [그림 8-8]을 간단하게 [그림 9-9]로 나타냈다.

9.2 메모리 구성

IT COOKBOOK

그림 9-9 간단한 메모리 구조

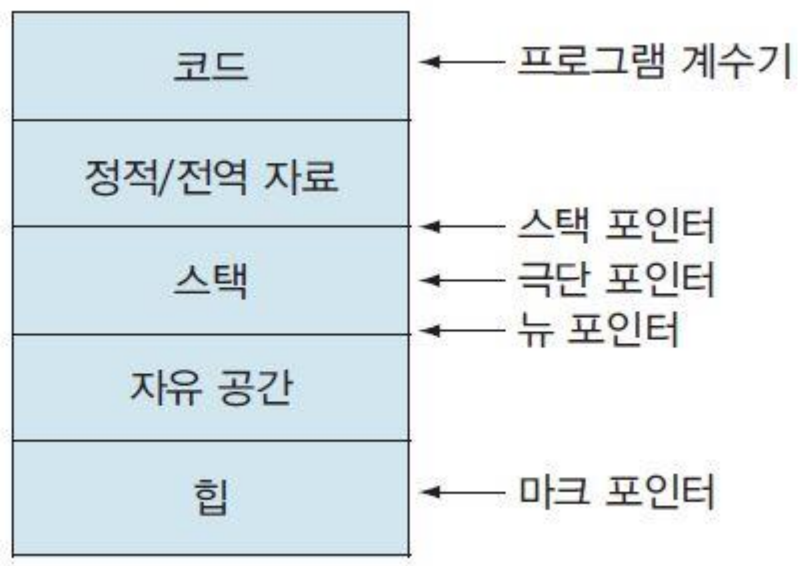


그림 8-8 p-기계의 메모리 구조

- [그림 9-9]에서 화살표는 스택과 힙의 확대 방향을 가리킨다. 전통적으로 스택은 메모리에서 아래 방향으로 확대되는 형태로 그리며, 그래서 톱은 실제적으로 그려진 구역의 바닥에 그려진다. 힙은 스택과 유사하게 그려지지만 LIFO 구조가 아니고, 그 확대 및 축소는 화살표가 가리키는 것보다 더욱 복잡하다.

9.2 메모리 구성

- 파스칼과 C 언어에서는 프로시저 활성을 관리하기 위해 제어 스택을 확장해서 사용한다. 호출이 일어날 때 활성은 실행을 잠시 멈추고, 프로그램 계수기와 레지스터 값 같은 기계의 상태 정보를 스택에 저장한다. 호출이 끝나고 제어가 반환될 때 호출 바로 뒤의 위치로 프로그램 계수기가 설정되고 관련된 레지스터 값이 다시 저장된 다음 그 활성은 실행을 계속한다.
- 프로시저가 한 번 실행되는 데 필요한 정보는 메모리의 연속 블록을 사용하여 관리되는데, 이러한 연속 블록을 활성 레코드(activation record) 혹은 활성 프레임(activation frame)이라 한다. 활성 레코드는 실 매개변수, 반환 값, 지역 자료, 임시 값을 저장하는 공간을 포함한다. 일반적인 활성 레코드는 [그림 9-10]과 같다.

반환 값에 대한 공간
실매개변수에 대한 공간
제어 링크(사용하지 않을 때도 있다)
접근 링크(사용하지 않을 때도 있다)
기계 상태 저장 공간
지역 자료에 대한 공간
임시 변수에 대한 공간

그림 9-10 **일반적인 활성 레코드**

9.2 메모리 구성

- 모든 언어와 컴파일러가 [그림 9-10]의 공간을 다 사용하는 것은 아니다. 또한 공간에 대한 자료의 순서를 포함하는 특수 상세 사항은 목표 기계의 구조, 컴파일되는 언어의 특성, 심지어 컴파일러 작성자의 취향에 따라 다를 수 있다. 파스칼과 C 언어는 프로시저가 호출될 때 실행 스택에 프로시저의 활성 레코드를 삽입하고, 제어가 호출한 쪽으로 되돌아갈 때 스택에서 활성 레코드를 삭제한다.
- [그림 9-10]의 각 공간에 대해 살펴보면 다음과 같다.
  - 반환 값에 대한 공간 : 호출한 프로시저로 결과 값이 전달될 때 사용한다. 반환 값은 효율을 위해 종종 레지스터에 담아 되돌려주기도 한다.
  - 실 매개변수에 대한 공간 : 호출 시 매개변수가 전달되면 사용한다. 활성 레코드에 매개변수를 저장한다고 했지만 실제로는 효율을 위해 기계 레지스터를 통해 매개변수를 전달한다.
  - 제어 링크 : 생략 가능한 공간으로 자신을 호출한 프로시저의 활성 레코드를 가리킨다.
  - 접근 링크 : 생략 가능한 공간으로 다른 활성 레코드에 있는 비지역 자료nonlocal data를 참조하는 데 사용된다. 파스칼 언어에서는 접근 링크가 필요하다.
  - 기계 상태 저장 공간 : 프로시저가 호출되기 바로 전의 기계 상태에 대한 정보를 저장한다.
  - 지역 자료에 대한 공간 : 프로시저가 실행될 때 지역적으로 사용되는 자료를 저장한다.
  - 임시 변수에 대한 공간 : 식을 계산하는 도중 발생하는 임시 값을 저장한다

9.2 메모리 구성

IT COOKBOOK

- 언어에 따라서는 활성 레코드가 정적 부분(포트란 77), 스택 부분(C, 파스칼), 혹은 힙 부분(리스프)에 할당된다. 활성 레코드가 스택에서 유지될 때 그것을 스택 프레임이라고도 부른다.
- 프로세서 레지스터도 실행 시간 환경의 일부분이다. 레지스터는 임시 변수, 지역 변수, 전역 변수를 저장하는 데 사용될 수 있다.
- 실행 시간 환경의 설계에서 특별히 중요한 부분은 프로시저나 함수가 호출될 때 발생하는 연산의 순서 결정이다. 이러한 연산은 활성 레코드에 대한 메모리 할당, 매개변수의 계산과 저장, 호출이 이뤄지는 데 필요한 레지스터를 저장하고 설정하는 것을 포함하며, 일반적으로 호출 순서(calling sequence)로 언급된다. 프로시저나 함수가 반환될 때 호출자가 접근할 수 있는 장소에 반환 값을 두는 것, 레지스터의 재조정, 활성 레코드에 대한 메모리 재조정 등이 호출 순서에 해당된다.

9.3 메모리 할당 전략

IT COOKBOOK

- 실행 시간 환경은 실행 과정을 관리하는 데 필요한 모든 정보를 유지해야 하고 메모리를 관리하는 역할
  - 특히 실행 과정을 관리하는 데 필요한 정보를 유지하고 메모리를 관리하는 역할을 하는 목적 컴퓨터의 레지스터와 메모리 구조가 실행 환경에 포함된다.
  - [그림 9-9]의 메모리 구조에서 자료 영역에 대해 메모리 할당 전략이 필요하다.
- 정적 및 동적이라는 용어는 각각 컴파일 시간과 실행 시간을 구별하는 데 사용된다.
  - 메모리 할당이 정적이라면 프로그램이 실행될 때 무슨 일을 하는지 보지 않고 프로그램 텍스트만을 보고 메모리 할당을 결정한다.
  - 메모리 할당이 동적이라면 프로그램이 실행되는 중에 메모리 할당을 결정한다.
    - 동적 할당을 위해 많은 컴파일러에는 스택 메모리 할당과 힙 메모리 할당이 있다.
- **세 가지 메모리 할당 전략**
  - **정적 메모리 할당**(static storage allocation) : 프로그램이 실행될 때 이미 해당 메모리의 크기가 결정되는 방법으로 포트란 77 언어에서 사용된다.
  - **스택 메모리 할당(**stack storage allocation) : 메모리를 스택으로 관리하는 방법으로 C, C++, 파스칼, 에이다와 같은 언어에서 사용된다.
  - **힙 메모리 할당**((heap storage allocation) : 필요할 때마다 동적으로 메모리를 할당하고 해체하는 방법으로 함수 언어(functional language)에서 사용된다

9.3 메모리 할당 전략

- **정적 메모리 할당**(static storage allocation) :
  - 정적 메모리 할당에서는 전역 변수뿐만 아니라 모든 변수가 정적으로 할당된다.
  - 각 프로시저는 실행 전에 정적으로 할당되는 하나의 활성 레코드만 가지고 있다.
  - 모든 변수는 지역적이든 전역적이든 고정 주소(fixed address)에 의해 직접적으로 접근할 수 있다.

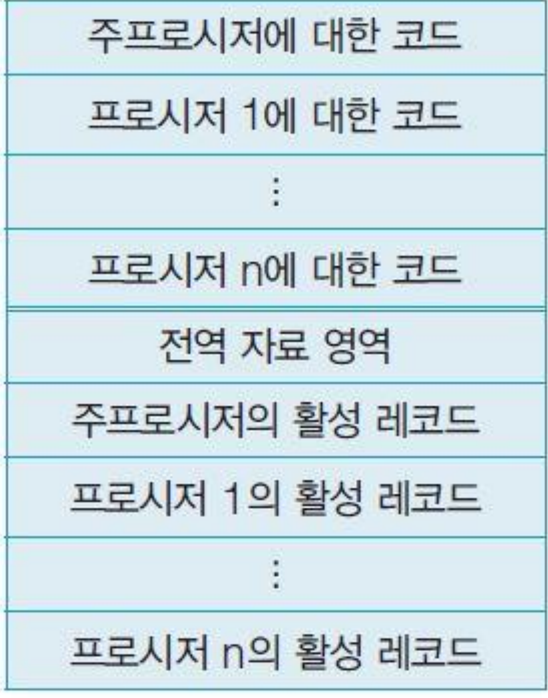


그림 9-11 정적 메모리 할당에 대한 전체 프로그램 메모리

9.3 메모리 할당 전략

- 정적 메모리 할당에서는 각 활성 레코드에 복귀 주소(return address) 이외의 다른 정보를 유지할 필요가 없으므로 정보 측면에서 상대적으로 오버헤드가 작다.
- 또한 정적 메모리 할당의 호출 순서가 매우 단순하다. 프로시저가 호출될 때 각각의 매개변수는 호출되는 프로시저의 활성에서 적절한 매개변수의 위치에 저장된다.
- 그런 다음 호출 프로그램 코드에 있는 복귀 주소가 저장되며, 호출된 프로시저 코드의 시작부로 분기가 실행된다. 복귀할 때는 복귀 주소로 분기가 실행된다.

▪ **[예제 9-4]** 정적 메모리 할당에 대한 메모리 구조

▪ [그림 9-12]의 포트란 77 프로그램을 보고 정적 메모리 할당에 대한 메모리 구조에 대해 설 명해보자.

9.3 메모리 할당 전략

–

```
PROGRAM TEST1
     COMMON MAXSIZE
     INTEGER MAXSIZE
     REAL TABLE(20), TEMP
     MAXSIZE = 20
     READ * , TABLE(1), TABLE(2), TABLE(3)
     CALL QUMEAN(TABLE, 3, TEMP)
     PRINT *, TEMP
     END

     SUBROUTINE QUMEAN(A, SIZE, QMEAN)
     COMMON MAXSIZE
     INTEGER MAXSIZE, SIZE
     REAL A(SIZE), QMEAN, TEMP
     INTEGER K
     TEMP = 0.0
     IF ((SIZE .GT. MAXSIZE). or. (SIZE. LT. 1)) GOTO 99
     DO 100 K = 1, SIZE
        TEMP = TEMP + A(K)*A(K)
100  CONTINUE
 99  QMEAN = SQRT(TEMP/SIZE)
     RETURN
     END
```

그림 9-12 포트란 77 프로그램

9.3 메모리 할당 전략

IT COOKBOOK

- [그림 9-12]의 프로그램은 주프로시저 TEST1과 부프로시저 QUMEAN으로 구성되어 있다. 이 프로그램에는 주프로시저와 QUMEAN 프로시저에서 COMMON MAXSIZE 선언에 의해 주어지는 1개의 전역 변수가 있다. COMMON 문장은 전역 변수로 서로 다른 프로시저에서 메모리를 공유하는 것을 허용하는 문장이다.
- [그림 9-13]은 [그림 9-12]에 대해 메모리에서 정수와 실수 값 사이의 크기를 고려하지 않은 실행 시간 환경이다.

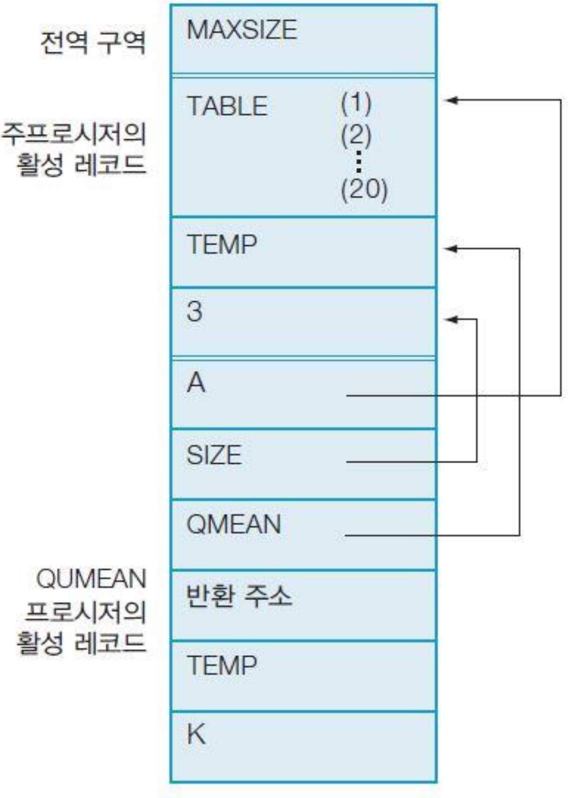


그림 9-13 [그림 9-12]에 대한 실행 시간 환경

9.3 메모리 할당 전략

- [그림 9-13]에서 화살표는 매개변수 A, SIZE, QMEAN이 주프로시저로 부터의 호출 동안 참조되는 값을 가리킨다.
- 포트란 77에서는 매개변수 값이 묵시적으로 참조 호출 값이며, 따라서 호출자의 실 매개변수인 TABLE, 3, TEMP의 위치(location) 값은 QUMEAN의 매개변수 위치 값으로 복사된다.

- **스택 메모리 할당** :
  - 재귀적 호출이 허용되고 지역 변수가 각각 호출할 때마다 새롭게 할당되는 언어에서는 활성 레코드가 정적으로 할당될 수 없다.
  - 대신에 활성 레코드가 스택 기반 형태로 할당되어야 하며, 이 경우에 새로운 활성 레코드는 새로운 프로시저가 호출되면 스택의 톱에 할당되고 호출이 종료되면 활성 레코드가 삭제된다.
  - 스택 메모리 할당 방법은 실행 시간 스택(run-time stack) 또는 호출 스택(call stack)이라고도 한다.
- **[예제 9-5] 실행 시간 스택에서 이뤄지는 삽입과 삭제에 대한 활성 레코드**
  - [그림 9-6]의 활성 트리를 통해 제어가 이동함에 따라 실행 시간 스택에서 이뤄지는 삽입과 삭제에 대한 활성 레코드를 나타내보자.
  - [풀이] [그림 9-14]는 실행 시간 스택에서 이뤄지는 활성 레코드를 보여준다.

9.3 메모리 할당 전략

활성 트리에서의 위치	스택상의 활성 레코드	설명
m	m a array	m의 프레임
m r	m a array r i integer	r의 활성
m r q(1,9)	m a array q(1,9) i integer	r의 프레임이 제거되고 q(1, 9)가 삽입된다.
m r q(1,9) p(1,9) q(1,3) p(1,3) q(1,0)	m a array q(1,9) i integer q(1,3) i integer	제어가 q(1, 3)으로 막 되돌아왔다.

그림 9-14 **실행 시간 스택에서 이뤄지는 활성 레코드**

9.3 메모리 할당 전략

IT COOKBOOK

- [그림 9-14]에서 점선은 이미 실행을 끝낸 활성 레코드이다.
- 프로그램은 프로시저 m의 활성에서 시작된다. 제어가 m(main)의 몸체 안에서 첫 번째로 r을 호출하면 프로시저 r이 그 활성을 시작하고 r의 활성 레코드가 스택에 저장된다.
- 제어가 r의 실행을 끝내면 r에 대한 활성 레코드가 스택에서 삭제된다. 이때 스택에는 m에 대한 활성만 남는다. m의 활성에서 제어가 실 매개변수 1과 9를 가지고 q를 호출하면 스택의 톱에는 q에 대한 활성이 할당된다.
- [그림 9-14]의 마지막 그림 전에는 몇 개의 활성이 발생했다가 사라진다.
- 마지막 그림에서 p(1,3)과 q(1,0)은 q(1,3)의 존속 시간 동안에 활성이 시작되고 끝난다.
- 그러므로 이것들의 활성 레코드는 스택의 톱에 q(1,3)만을 남겨놓는다. ■

- 호출 순서와 활성 레코드는 서로 다르다. 호출 순서 안에 있는 코드는 보통 두 가지로 나뉘는데 호출 프로시저(calling procedure, caller)와 피호출 프로시저(called procedure, callee)가 그것이다. 실행 시 호출자와 피호출자 사이의 작업을 명확하게 구분할 수는 없다.
- 소스 언어, 목적 기계, 운영체제에 따라 작업의 분담이 달라지기 때문이다.
- 호출자와 피호출자가 어떻게 협력하여 스택을 관리하는지에 대한 예를 [그림 9-15]에 나타냈다.

9.3 메모리 할당 전략

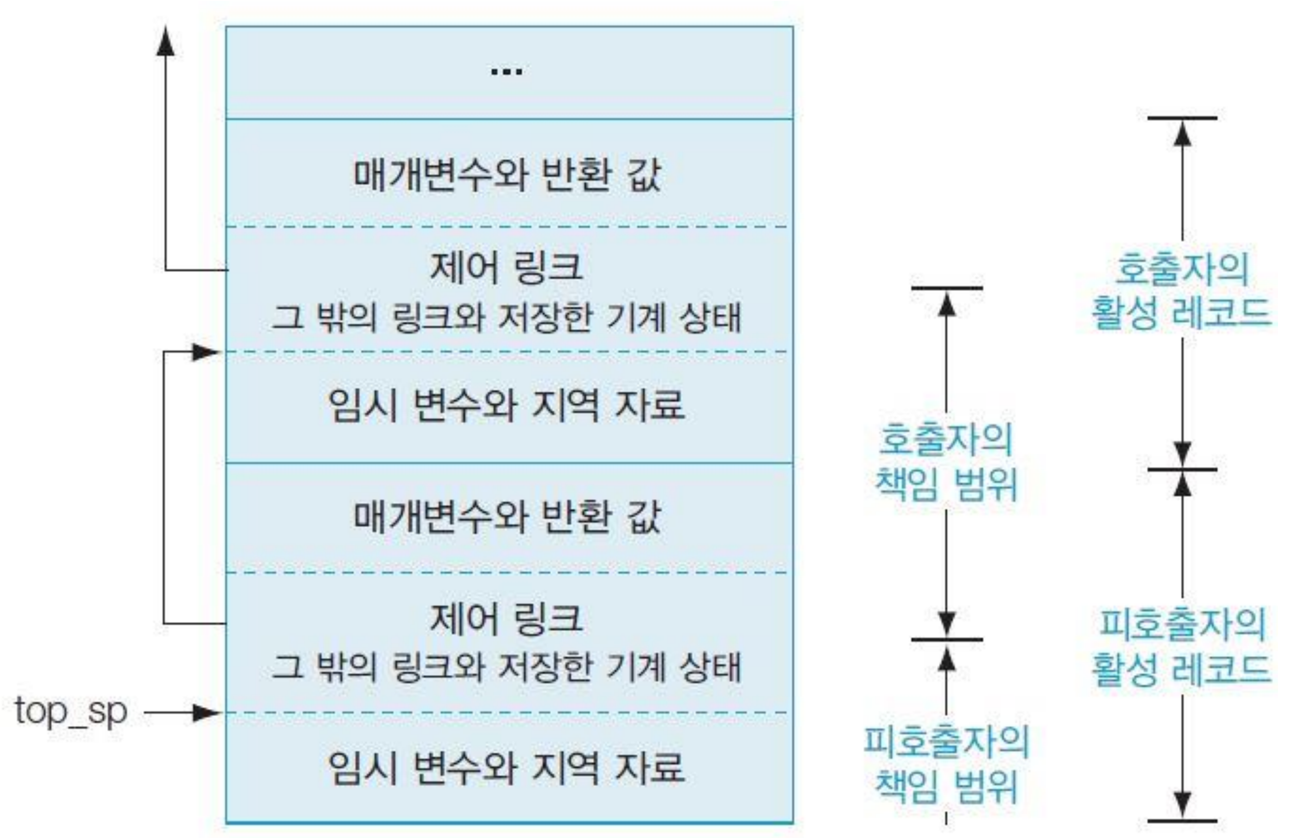


그림 9-15 호출자와 피호출자 사이의 스택 관리

- [그림 9-15]에서처럼 레지스터 top_sp는 활성 레코드의 기계 상태 공간 끝을 가리킨다. 피호출자의 활성 레코드 내에 있는 이 지점은 호출자에게 알려지고, 호출자는 제어가 피호출자에게 전달되기 전에 top_sp를 설정해야 한다.

9.3 메모리 할당 전략

- 호출 순서 및 호출자와 피호출자 사이의 스택 관리는 다음과 같다.
  - ① 호출자는 실 매개변수를 평가한다.
  - ② 호출자는 복귀 주소와 top_sp의 이전 값을 피호출자의 활성 레코드에 저장한다. 그 다음 호출자는 [그림 9-15]에 나타낸 위치로 top_sp를 증가시킨다. 즉 top_sp는 호출자의 지역 자료와 임시 기억 변수, 그리고 피호출자의 매개변수와 상태 공간을 지나서 이동한다.
  - ③ 피호출자는 레지스터 값과 다른 상태 정보를 저장한다.
  - ④ 피호출자는 자신의 지역 자료를 초기화하고 실행을 시작한다.

- 이에 대응하는 적절한 복귀 순서는 다음과 같다.
  - ① [그림 9-15]와 같이 피호출자는 반환 값을 매개변수 다음에 저장한다.
  - ② 피호출자는 기계 상태 필드status field의 정보를 이용하여 top_sp와 다른 레지스터를 복원하고, 호출자가 상태 필드에 저장했던 복귀 주소로 분기한다.
  - ③ top_sp가 감소되었지만 호출자는 반환 값이 있는 위치를 top_sp의 현재 값으로부터 상대적인 위치로 알 수 있다. 따라서 호출자는 이 값을 사용할 수 있다.

9.3 메모리 할당 전략

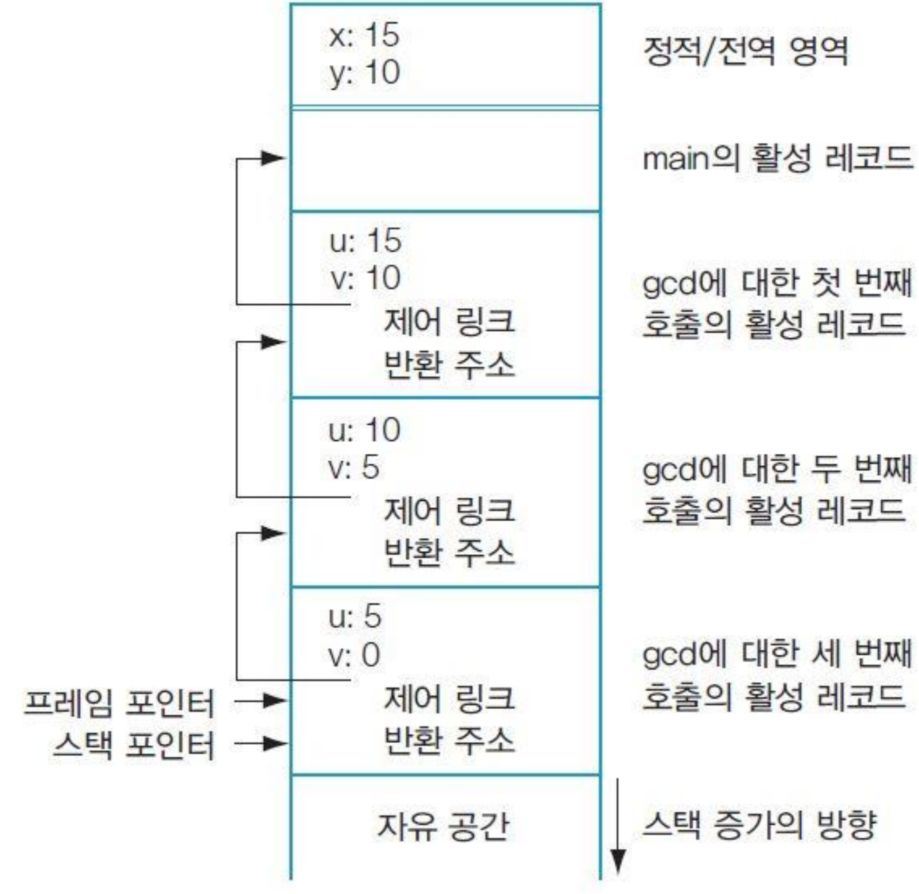
IT COOKBOOK

- **[예제 9-6] 실행 시간 환경 중 스택 메모리 할당**
- 음수가 아닌 2개의 정수에 대해 최대 공약수를 구하는 유클리드(Euclid) 알고리즘을 생각해보자. 이를 C 언어로 간단하게 구현한 재귀적 프로그램은 다음과 같다. 이 프로그램을 보고 실행 시간 환경 중에 스택 메모리 할당을 설명해보자.

```
#include <stdio.h>
int x, y;
int gcd(int u, int v)
{ if (v == 0) return u;
  else return gcd(v, u % v);
}
main()
{ scanf("%d%d", &x, &y);
  printf("%d₩n", gcd(x, y));
  return 0;
}
```

9.3 메모리 할당 전략

- [풀이] 사용자가 값 15와 10을 입력하면 x는 15, y는 10이 되고, main은 처음에 gcd(15,10)을 호출한다. 이 호출은 u가 15, v가 10이므로 15 % 10 = 5가 되어 재귀적으로 gcd(10,5)를 호출한다. 이것은 10 % 5 = 0이므로 세 번째 재귀적으로 gcd(5,0)을 호출한 다음 값 5를 반환한다. 이 과정에 대한 실행 시간 환경은 다음 그림과 같다.



9.3 메모리 할당 전략

- **힙 메모리 할당** :
  - 스택 메모리 할당 전략은 C, 파스칼, 에이다와 같은 표준적인 명령형(imperative) 언어 사이에서 가장 공통적인 방법이지만, 활성이 끝난 뒤에도 지역 변수의 값을 유지해야 하는 경우나 호출된 활성이 호출자가 사라진 뒤에도 살아 있어야 하는 경우에는 사용하지 못한다.
  - 이런 경우에는 무기한 살아 있거나 프로그램이 명시적으로 제거할 때까지 살아 있는 자료를 저장하는 힙을 사용하여 해결할 수 있다.
  - 지역 변수는 보통 해당 프로시저가 끝나면 접근할 수 없게 되지만, 많은 언어에서는 해당 프로시저의 활성화에 종속되지 않는 존속 시간을 가진 객체나 다른 자료를 사용할 수 있다.
    - 예를 들어 C++와 자바는 new로 생성한 객체를 프로시저 간에 전달할 수 있다. 따라서 이런 객체는 자신을 생성한 프로시저가 종료된 후에도 계속 존재할 수 있다.
  - 힙 메모리 할당은 연속적인 기억 공간을 꾸러미로 만들어 활성 레코드나 다른 객체에 할당하는 전략이다. 이 꾸러미는 다른 순서로 해체될 수 있으므로 힙은 사용 중인 영역과 사용되지 않고 있는 영역(free area)으로 섞어서 구성할 수 있다.
  - 활성 레코드의 힙 메모리 할당은 [그림 9-16]을 통해 알 수 있다.

9.3 메모리 할당 전략

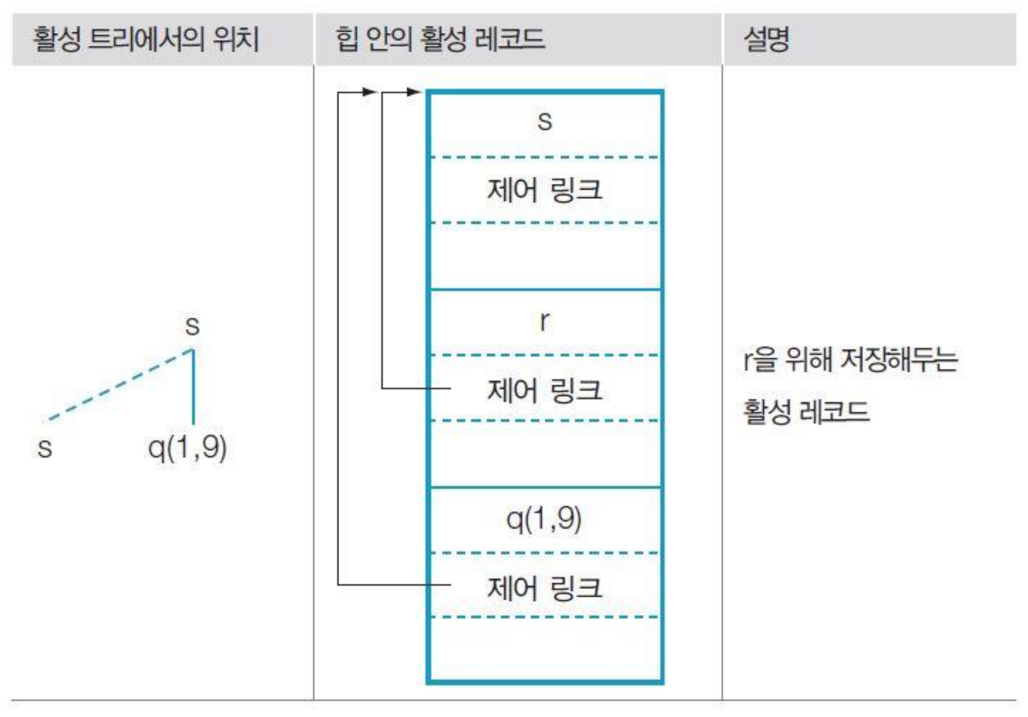


그림 9-16 힙에 대한 활성 레코드

- [그림 9-14]의 세 번째 그림을 보면 프로시저 r에 대한 활성이 끝나고 나서 r에 대한 활성 레코드가 삭제되지만, [그림 9-16]에서는 r에 대한 활성이 끝나도 r에 대한 활성 레코드가 계속 유지된다.
- 힙 메모리 할당은 모든 참조가 없어질 때만 활성 레코드를 해제할 수 있으며, 활성 레코드를 동적으로 실행하는 동안 임의의 시점에 해제되어야 하기 때문에 동적 메모리 할당이라고도 부른다.

9.4 매개변수 전달 방법

- 포트란 77은 참조 호출 방법을 채택하고, C는 값 호출 방법을 채택했다.
- 하나의 프로시저가 또 다른 프로시저를 호출할 때 매개변수를 통해 서로 값을 주고받는다. 이러한 매개변수에는 **실 매개변수와 형식 매개변수**가 있다. 즉 실 매개변수와 형식 매개변수는 서로 값을 주고받는다. 이처럼 실 매개변수와 형식 매개변수 사이에 값을 주고받는 것을 **매개변수 전달(parameter passing)**이라고 하는데, 값을 주고받는 전달 방법에 따라 프로그램의 결과가 달라질 수 있다. 그래서 여러 가지 매개변수 전달 방법이 존재한다.
- 매개변수 전달 방법을 설명하기 전에 필요한 개념을 하나 이해하기 위해 다음과 같은 치환문을 생각해보자.
  - a[i] = a[j]
  - 산술식 a[j]는 값을 나타내는 r-값(right value)이고, a[i]는 a[j]의 값이 있는 메모리의 위치인 l-값(left value, location value)을 나타낸다.
  - l-값은 산술식이 나타내는 기억 위치를 나타내고 r-값은 메모리에 저장될 값을 나타내는 용어이다.
  - 접두사 l과 r은 각각 치환문의 왼쪽과 오른쪽을 의미한다.

■ 값 호출

- 값 호출(call by value)은 매개변수를 전달하는 가장 일반적인 방법으로 C, 파스칼, 에이다에서는 기본적이다.
- 실 매개변수와는 별도로 형식 매개변수에 대한 메모리를 할당하는 방법이다.
- 형식 매개변수는 지역 변수처럼 다뤄지고, 호출자는 형식 매개변수에 대한 메모리에 실 매개변수의 r-값을 복사하는 방법으로 구현한다.
- 값 호출의 특징은 형식 매개변수에 대한 연산이 호출자 활성 레코드 안의 값에는 아무런 영향을 미치지 않는다는 것이다.
- [그림 9-17]은 두 수를 서로 교환하는 파스칼 프로그램이다

9.4 매개변수 전달 방법

■

```
① program reference(input, output);
② var a, b : integer;
③ procedure swap(var x,y : integer);
④     var temp:integer;
⑤     begin
⑥        temp:= x;
⑦        x:= y;
⑧        y:= temp;
⑨     end;
⑩  begin
⑪    a:=1; b:=2;
⑫    swap(a,b);
⑬    writeln('a=',a);
⑭    writeln('b=',b)
⑮  end.
```

그림 9-17 두 수를 교환하는 파스칼 프로그램

9.4 매개변수 전달 방법

- [그림 9-17]에서 ③행에 있는 var를 없애면 이 프로그램은 값 호출 방법이다.
- ⑫행에 있는 swap(a, b)를 호출하면 a와 b 값을 변화시키지 않는다.
- 이 과정을 살펴보면, ⑫행에서 swap(a, b)를 호출하면 실 매개변수 a와 b는 a:=1, b:=2이므로 swap(1, 2)가 되고 ③행에 있는 swap 프로시저를 실행한다.
- 이때 형식 매개변수는 x와 y이다. 따라서 형식 매개변수에 대한 메모리를 별도로 할당하고 x에는 1을, y에는 2를 할당한다.
- 그런 다음 지역 변수 temp에 대한 메모리를 할당하고 ⑥행을 실행한다. temp에는 1, x에는 2가 할당되고, y에 는 다시 temp의 값 1이 할당된다. 여기까지는 x = 2, y = 1이 할당되어 x와 y의 값이 서로 교환되었음을 알 수 있다.
- 하지만 제어가 호출자로 반환되고 swap에 대한 활성 레코드가 해체될 때 지역 변수인 x, y, temp에 대한 메모리를 삭제하기 때문에 호출자의 활성 레코드에 는 아무 영향을 주지 못한다.
- 값 호출 프로시저는 비지역 변수나 포인터로 전달되는 값을 통해 호출자에 영향을 줄 수 있다. 그렇지 않으면 호출된 프로그램에서 값을 출력하는 방법을 선택해야 한다.

9.4 매개변수 전달 방법

■ 참조 호출

- 참조 호출(call by reference, call by address) 방법은 매개변수가 참조에 의해 전달될 때 호출자가 피호출자에게 실 매개변수의 메모리에 대한 주소를 가리키는 포인터를 전달한다.
- 포트란 77에서 매개변수 전달 방법은 참조 호출밖에 없다. 또한 파스칼에서 참조 호출은 var 키워드의 사용으로 이뤄지며, C++에서는 매개변수 선언 시 특수 기호 &의 사용으로 이뤄진다.
- 참조 호출은 만약 실 매개변수가 l-값을 가지고 있는 산술식이거나 변수라면 그 l-값 자신을 전달한다. 그러나 실 매개변수가 b + c나 5와 같이 l-값을 가지지 않은 산술식이라면 그 산술식은 호출자 활성 레코드 내의 새로운 위치에 주소를 전달한다.
- 피호출자가 프로시저 안에 있는 형식 매개변수를 목적 코드에서 참조하려면 피호출 프로시저로 전달된 포인터를 통한 간접 참조(indirect reference) 방식을 이용해야 한다.

[예제 9-7] swap(a, b)에 대해 구현되는 과정

- [그림 9-17]의 프로시저에서 swap(a, b)에 대해 구현되는 과정을 설명해보자.
- [풀이]
  - 실 매개변수 a와 b에 대한 주소를 피호출자 활성 레코드의 x와 y에 해당하는 위치 arg1과 arg2에 각각 복사한다. ⑥행에서 temp에 arg1을 가리키는 위치의 내용을 복사한다. arg1이 가리키는 위치의 내용을 arg2가 가리키는 위치의 값으로 설정한다. 마지막으로 arg2가 가리키는 위치의 내용을 temp의 값에 복사한다.

■ 이름 호출

- 이름 호출(call by name)은 가장 복잡한 매개변수 전달 방법이다. 이 방법은 형식 매개변수의 이름이 사용될 때마다 그에 대응하는 실 매개변수 자체가 사용된 것처럼 매번 다시 계산 및 시행된다.
- 이름 호출은 값 호출과 같이 알골 60에서 매개변수 전달 방법으로 제공되었으나 부작용이 존재하는 경우에 기대하지 않았던 결과가 나타나기도 하고, 매개변수가 평가될 때마다 호출되어야만 하는 성크(thunk)라는 프로시저를 구현해야 하므로 구현하기가 어려워 잘 사용되지 않았다.
- **[예제 9-8] swap(a, b)에 대해 이름 호출로 구현되는 과정**
  - [그림 9-17]의 프로시저에서 swap(a, b)에 대해 이름 호출로 구현되는 과정을 설명해보자.
  - [풀이] 다음과 같이 x가 나타날 때마다 a를, y가 나타날 때마다 b를 대체하는 방법이다.
  - temp := a;
  - a := b;
  - b := temp;

9.4 매개변수 전달 방법

IT COOKBOOK

값-결과 호출

- 값-결과 호출(call by value-result)은 값 호출과 참조 호출을 합성한 방법으로 입력 복사 출력 복사(copy-in copy-out), 복사 –회복(copy-restore)이라고도 한다. 형식 매개변수에 대한 메모리에 실 매개변수 의 값을 복사하고 호출 전에 실 매개변수로부터 계산된 l-값에 형식 인자의 마지막 값을 복사 한다. 즉 매개변수의 값이 복사되고 프로시저에서 사용되며, 그런 다음 매개변수의 최종 값은 프로시저가 종료될 때 매개변수의 위치에 다시 복사된다.
- **[예제 9-9] 값-결과 호출과 참조 호출 비교하기**
  - 다음과 같은 C 코드에서 값-결과 호출과 참조 호출을 구분해보자.

```
void p(int x, int y)
{ ++x;
  ++y;
}
main( )
{ int a = 1;
  p(a, a);
  return 0;
}
```

  - [풀이] 참조 전달이 사용된다면 a는 p가 호출된 후에 값 3을 가지며, 값-결과 호출이 사용된다면 a는 값 2 를 가진다.

9.4 매개변수 전달 방법

- **[예제 9-10] 참조 호출, 값 호출, 이름 호출의 결과 값 구하기**
  - 다음과 같은 PL/I 형태의 프로그램에서 참조 호출, 값 호출, 이름 호출을 하는 경우의 결과 값을 구해보자.
  - P : PROCEDURE;
  - DECLARE A(3), I;
  - I=1; A(1)=2; A(2)=4;
  - CALL Q(A(I));
  - Q : PROCEDURE(B)
  - A(1)=3; I=2; PUT LIST(B);
  - END Q;
  - END P;
  - [풀이]
  - 1. 참조 호출
  - I=1; A(1)=2; A(2)=4;
  - CALL Q(A(I));로부터 addr(A(1))=addr(B)이므로 A(1)=2이다.
  - A(1)=3; I=2; PUT LIST(B);
  - A(1)=3이고 PUT LIST(B);로부터 A(1)을 출력해야 하므로 3이 출력된다.

9.4 매개변수 전달 방법

- 2. 값 호출
  - I=1; A(1)=2; A(2)=4;
  - CALL Q(A(I));에서 A(1)=2 값을 B에 넘겨주는데 B는 메모리를 따로 잡아서 2를 저장한다.
  - A(1)=3; I=2; PUT LIST(B);
  - PUT LIST(B);로부터 B를 출력하면 2가 저장되어 있으므로 2를 출력한다.
- 3. 이름 호출
  - I=1; A(1)=2; A(2)=4;
  - CALL Q(A(I));에서 값을 B에 넘겨주는 것이 아니라 B가 나타날 때마다 A(I)를 대체한다.
  - A(1)=3; I=2; PUT LIST(B);
  - PUT LIST(B);는 PUT LIST(A(I));이므로 A(I)를 출력하면 된다. 현재 I는 2이므로 A(2)를 출력하여 4가 출력된다.