

#6

20174627 김혜진

< 연습문제 >

#6. 다음 JavaScript 골격 프로그램을 생각해보자.

//main 프로그램

**var** x;

**function** sub1( ) {

**var** x;

**function** sub2( ) {

**. . .**

}

}

**function** sub3( ) {

**. . .**

}

위의 프로그램 실행이 다음의 순서로 진행된다고 가정하라.

**Main이 sub1을 호출한다.**

**Sub1이 sub2를 호출한다.**

**Sub2가 sub3을 호출한다.**

1. **정적 영역을 가정하고서, 다음에서, x에 대한 참조에 대해서 x의 어느 선언이 올바른가?**
2. sub1 = **sub1 -> sub1( )의 var x 참조**

정적 영역에서 x에 대한 참조를 변수와 연관하려면 그 선언문을 찾아야 하는데, 방법은 이러하다. 먼저, 지역적으로 선언문을 찾고, 현재 영역을 포함한 영역에서 점차 더 큰 영역 순으로 선언문을 찾아야 한다.

해당 문제의 경우 main( )에서 sub1( )을 호출하였기 때문에 main()에 있는 전역변수가 은폐(hidden)되는데, sub1( )에서 x에 대한 선언문이 찾아졌기 때문에 정적 영역에서 x에 대한 참조에 대해서 x는 sub1( )에서의 선언이 올바르다.

1. sub2 = **sub2 -> sub1( )의 var x 참조**

정적 영역에서 지역 변수의 영역은 선언문부터 그 선언문이 나타난 블록 끝 까지다. 이는 프로그램의 실행 순서와 상관없이 프로그램 블록 관계에 의해 실행된다는 뜻이다.

정적 영역에서 현재 영역과 가장 가까운 정적 선조를 정적 부모라고 칭하는데, sub2( )에는 var x의 선언문이 없기 때문에 바깥 블록 즉, 정적 부모 영역인 sub1( )에서 x에 대한 참조를 하게 된다.

1. sub3 **= sub3 -> 전역 변수 var x 참조**

위에서 정적 영역에서 변수의 영역은 프로그램의 실행 순서와 상관없이 프로그램 블록 관계에 의해 실행된다고 말한바가 있다.

sub3( )도 마찬가지로 var x에 대한 선언문이 없기 때문에 바깥 블록에 위치한 정적 부모 영역으로 가서 x에 대한 참조를 해주어야 한다. 하지만, sub3( )의 경우 바깥 블록이 없기 때문에 main( )에서 선언된 전역 변수 var x를 통해 sub3( )에서 x에 대한 참조를 해준다.

1. **동적 영역을 가정하고서, a의 문제를 다시 해결하라.**

동적 영역은 부 프로그램의 호출 시퀀스에 기반하여 실행 시간을 결정한다. 쉽게 말해 x에 대한 참조는 변수 영역이 프로그램 블록 관계에 얽매이지 않고, 호출 순서에 의해 실행된다는 말이다.

x에 대한 참조는 현 지점의 실행에 이르게 한 부 프로그램의 호출 체인의 역순으로 탐색을 해보면 알 수 있다.

먼저, main( )이 sub1( )을 호출하였고 sub1( )에서 x가 선언되었다. 다음으로 sub1( )이 sub2( )를 호출하였고, sub2( )가 sub3( )를 호출하였기 때문에 모두 sub1( )에서 선언 된 x에 대해 참조를 하면 된다.

1. sub1 -> sub1( )의 var x 참조 = sub1( ) 내부에 선언된 변수 참조
2. sub2 -> sub1( )의 var x 참조 = sub2( )를 호출한 sub1( )에 선언된 변수 참조
3. sub3 -> sub1( )의 var x 참조 = sub3( )를 호출한 sub2( )와 이를 호출한 sub1( )의 x 참조

#9. 다음 Python 프로그램을 생각해보자.

x = 1;

y = 3;

z = 5;

**def** sub1( ) :

a = 7;

y = 9;

z = 11;

…

**def** sub2( ) :

**global** x;

a = 13;

x = 15;

w = 17;

…

**def** sub3( ) :

**nonlocal** a;

a = 19;

b = 21;

z = 23;

…

…

**정적 영역이 사용된다고 가정하고, sub1, sub2, sub3의 각 몸체에서 가시적인 모든 변수들을 그 변수가 선언된 프로그램 단위와 함께 나열하라.**

|  |  |
| --- | --- |
| **범위** | **참조** |
| **sub1( )** | 전역 변수는 **x = 1**  sub1의 지역 변수는 **a = 7, y = 9, z = 11** |
| **sub2( )** | sub2의 지역 변수는 **a = 13**  전역 변수(global 키워드를 사용하여 값 변경)는 **x = 15**  sub2의 지역 변수 참조는 **w = 17**  전역 변수는 **y = 3, z = 5** |
| **sub3( )** | 전역 변수(sub2에서 참조하여 변경 된 x)는 **x = 15**  전역 변수는 **y = 3**  sub2의 지역변수 참조는 **a = 19**  sub3의 지역 변수는 **b = 21**  지역 변수는 **z = 23** |

#10. 다음 C 프로그램을 생각해보자.

**void** fun(**void**) {

**int** a, b, c; /\* 정의 1 \*/

…

**while**(…) {

**int** b, c, d; /\* 정의 2 \*/

… <-------------------------------- 1

**while**(…) {

**int** c, d, e; /\* 정의 3 \*/

… <---------------------- 2

}

… <-------------------------------- 3

}

… <------------------------------------------- 4

}

**위 함수에서 화살표로 지시된 4개 지점 각각에 대해, 각 가시적인 변수와 이 변수를 정의하는 정의문의 번호를 함께 나열하라.**

|  |  |
| --- | --- |
| **지점** | **참조환경** |
| **1** | a는 정의1  b, c, d는 정의2  :: 정의 1의 b, c는 정의 2에 의해 은폐(hidden) |
| **2** | a는 정의1  b는 정의2  c, d, e는 정의3 (정의2의 c 은폐)  :: 정의 2의 c, d는 정의 3에 의해 은폐(hidden) |
| **3** | a는 정의1  b, c, d는 정의2(정의1의 b 은폐)  :: 정의 3의 c, d, e는 참조 영역을 벗어났기 때문에 정의 2에서 은폐(hidden)된 변수들이 참조 됨 |
| **4** | a, b, c는 정의1  :: 정의 2의 b, c, d는 참조 영역을 벗어났기 때문에 정의 1에 은폐(hidden)된 변수들이 참조 가능 |

#11. 다음 C 골격 프로그램을 생각해보자.

**void** fun1(**void**); /\* 원형 \*/

**void** fun2(**void**); /\* 원형 \*/

**void** fun3(**void**); /\* 원형 \*/

**void** main(**void**) {

**int** a, b, c;

…

}

**void** fun1 (**void**) {

int b, c, d;

…

}

**void** fun2 (**void**) {

int c, d, e;

…

}

**void** fun3 (**void**) {

**int** d, e, f;

…

}

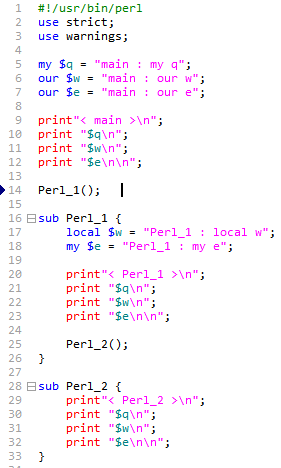
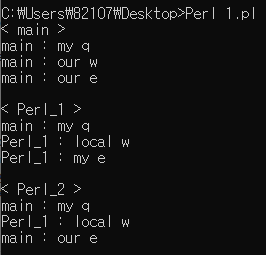
호출 순서가 다음과 같이 주어져 있고 동적-영역 규칙이 사용된다고 가정하면, 마지막으로 호출된 함수가 실행되는 동안에 어떤 변수가 가시적인가? 각 가시적인 변수에 대해 그 변수가 정의된 함수 이름을 포함시켜라.

1. main은 fun1을 호출; fun1은 fun2를 호출; fun2는 fun3를 호출.
2. main은 fun1을 호출; fun1은 fun3를 호출.
3. main은 fun2를 호출; fun2는 fun3를 호출; fun3는 fun1을 호출.
4. main은 fun3를 호출; fun3는 fun1을 호출.
5. main은 fun1을 호출; fun1은 fun3를 호출; fun3는 fun2를 호출.
6. main은 fun3를 호출; fun3는 fun2를 호출; fun2는 fun1을 호출.

|  |  |  |
| --- | --- | --- |
| **문제** | **마지막 호출함수** | **가시적인 변수** |
| **a** | **fun3** | a는 main의 a  b는 fun1의 b  c는 fun2의 c  d는 fun3의 d  e는 fun3의 e  f는 fun3의 f |
| **b** | **fun3** | a는 main의 a  b는 fun1의 b  c는fun1의 c  d는 fun3의 d  e는 fun3의 e  f는 fun3의 f |
| **c** | **fun1** | a는 main의 a  b는 fun1의 b  c는 fun1의 c  d는 fun1의 d  e는 fun3의 e  f는 fun3의 f |
| **d** | **fun1** | a는 main의 a  b는 fun1의 b  c는 fun1의 c  d는 fun1의 d  e는 fun3의 e  f는 fun3의 f |
| **e** | **fun2** | a는 main의 a  b는 fun1의 b  c는 fun2의 c  d는 fun2의 d  e는 fun2의 e  f는 fun3의 f |
| **f** | **fun1** | a는 main의 a  b는 fun1의 b  c는 fun1의 c  d는 fun1의 d  e는 fun2의 e  f는 fun3의 f |

< 프로그래밍 연습문제 >

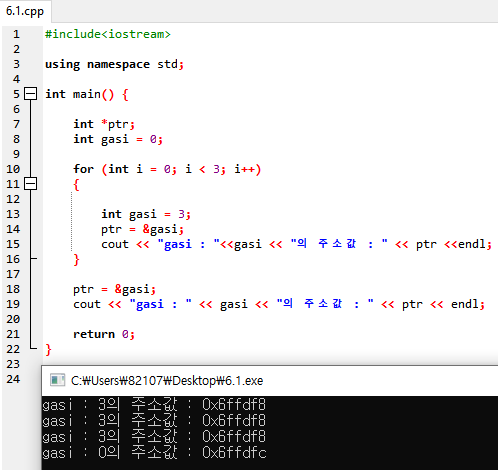
#1. Perl은 정적 영역과 일종의 동적 영역을 모두 지원한다. 이 두 가지의 영역을 모두 사용하고, 이 두 가지 영역의 효과에서의 차이를 분명히 보여주는 Perl 프로그램을 작성하라. 이 장에 기술된 동적 영역과 Perl에서 구현된 동적 영역 간의 차이를 분명하게 설명하라.

|  |
| --- |
| 해당 프로그램에서 my는 정적 영역이며, local은 동적 영역이다. Main( )에서 q, w, e를 출력한 후, Perl\_1의 결과를 보면 main( )의 w, e 값이 Per\_1의 local w 및 my e 값으로 바뀐 것을 확인할 수 있다. w는 동적 영역 참조를 이용한 것이고, e는 정적 영역 참조를 이용한 것 이다. Perl\_2의 결과를 보면 Perl\_2에서 w는 동적 영역 참조로 호출 시퀀스에 의해 Perl\_1에 있는 선언문 w를 참조한다. 하지만, Perl\_1에서 my로 정적 영역 참조를 통해 할당 받은 e는 해당 지역 안에서 선언문을 찾지 못해 main( )에서 선언문을 찾아 그 값을 참조하여 결과를 출력하게 된다. |

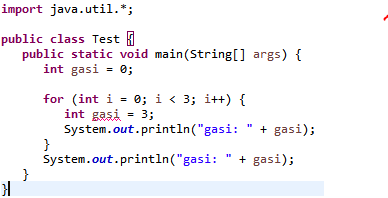
#6. for문에 선언된 변수의 영역을 결정하는 테스트 프로그램을 C++, Java, 그리고 C#으로 작성하라. 특히, 그 코드는 변수가 for문의 몸체 이후에서 가시적인지를 판단해야 한다.

< C++ >



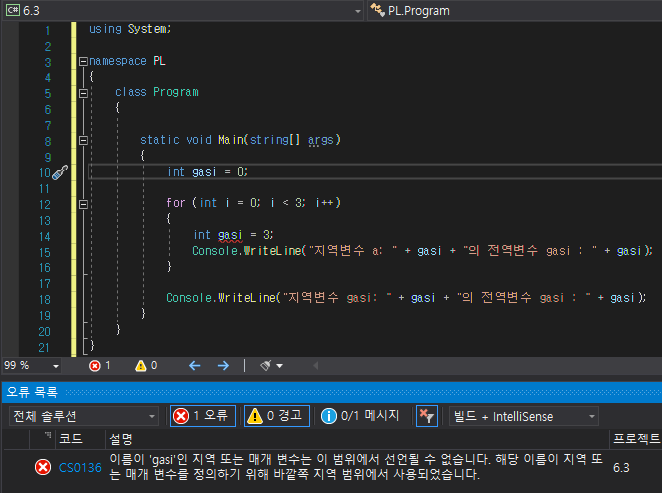
|  |
| --- |
| for문에 지역 변수로 gasi를 선언해준다. For문 블록을 나가게 되면 해당 변수는 사용이 불가능 하다. 또한, 포인터를 통해 gasi의 주소 값을 확인할 수 있다. 결과적으로 for문 안과 밖에 선언된 gasi의 주소 값이 서로 다른 것으로 보아 for문에 선언된 변수의 영역은 for문 안에서만 가시적이라는 것을 확인할 수 있다. |

< Java >



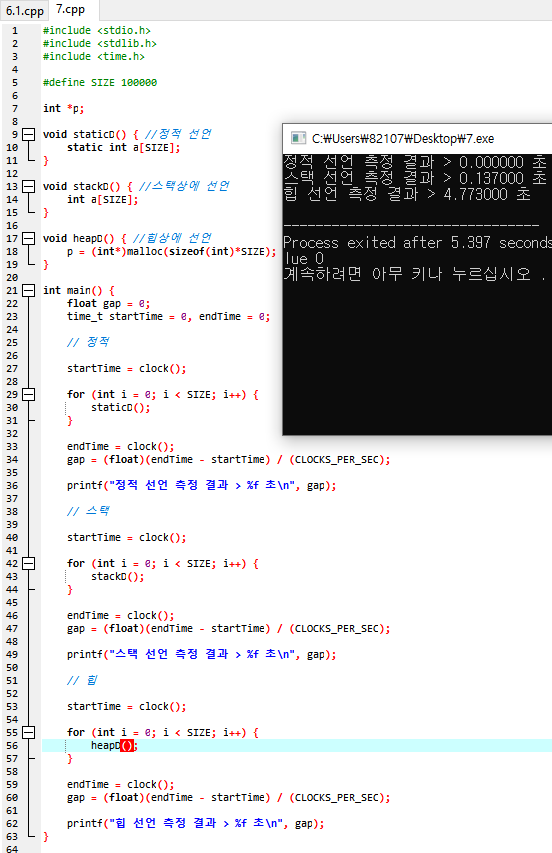
|  |
| --- |
| 이미지를 보면 int gasi = 3; 부분에 빨간 밑줄이 그어져 있는 걸 확인 할 수 있다. Main( ) 함수에 선언된 변수 gasi와 for( )문 안에 선언된 변수 gasi가 중복으로 선언 되었기 때문이다. 따라서 해당 프로그램은 실행되지 않고 오류가 뜬다. |

< C# >



|  |
| --- |
| 마찬가지로 for( )문에 들어가기 전에 main( )에서 먼저 선언 된 gasi 변수가 있는데 for( )문 에서 다시 선언 하였기 때문에 오류가 발생했다. |

# 7. C나 C++로 다음 세 개의 함수를 작성하라 : 규모가 큰 배열을 정적으로 선언하는 함수, 동일한 크기의 배열을 스택상에 선언하는 함수, 그리고 동일한 크기의 배열을 힙상에 생성하는 함수. 각 함수를 상당히 많은 횟수(적어도 100,000번)로 호출하고, 각 함수에서 요구되는 시간을 출력하라. 그 결과를 설명하라.



|  |
| --- |
| 정적 선언 > 스택 선언 > 힙 선언 순서로 빠르 다는 것을 알 수 있다. 정적으로 선언 된 배열은 프로그램이 실행되기 전에 바인딩이 일어나 함수 호출 시 메모리를 재할당 하는 일이 없어 프로그램 실행이 종료될 때 까지 동일한 셸에 유지되기 때문에 제일 빠르다. 스택 상에 선언 된 배열은 이미 할당 된 스택 상에 다시 할당 되어진다. 따라서 호출될 때 마다 할당이 반복되는 과정에서 오버헤드가 발생하기 때문에 두 번째로 빠르다. 힙 상에 선언 된 배열은 프로그램 실행 중에 힙으로부터 할당 되고, 할당 작업을 처리하기 위해서는 많은 시간이 소요된다. 따라서 세 번째로 빠르다. |