Names

1. Static Scope – Which Value Is Printed?

Consider the fragment:

```
int X = 0;
int Y;

void fie() {
  X++;
}

void foo() {
  X++;
  fie();
}

read(Y);
if (Y > 0) {
  int X = 5;
  foo();
} else {
  foo();
}

write(X);
```

Analysis: Under static (lexical) scope, the free occurrences of X in the definitions of `foo` and `fie` refer to the global X (declared outside all blocks), regardless of any local declarations. Thus, whether or not the `if` branch creates a local X (with value 5), the call to `foo()` will update the global X.

Initially, global $X = 0$. When `foo` is called, it performs:

- $X++$ (global X becomes 1),
- Then calls `fie()`, which does $X++$ (global X becomes 2).

Finally, `write(X)` prints the global X, which is 2.
Answer: The printed value is 2.

2. Dynamic Scope – Which Value(s) Are Printed?

Now consider:

```
int X;
X = 1;
int Y;

void fie() {
  foo();
  X = 0;
}

void foo() {
  int X;
  X = 5;
}

read(Y);
if (Y > 0) {
  int X;
  X = 4;
  fie();
} else {
  fie();
}

write(X);
```

Analysis (dynamic scope): Under dynamic scoping, the binding of a free variable is determined by the call chain at runtime.

Case A: $Y > 0$

- In the `if` branch, a local X is declared and set to 4.
- Then `fie()` is called. In `fie`, the call to `foo()` does not affect any caller's X because the local X is specific to `foo`.
- Returning to `fie`, the assignment $X = 0$ updates the X from the `if` branch (set to 4) to 0.
- After the `if` block, the only X visible is the global X, which remains unchanged (initially 1).
- Finally, `write(X)` prints the global X, which is 1.

Case B: $Y \leq 0$

- No new local X is declared. The call to `fie()` happens in an environment where the only binding is the global $X = 1$.
- Inside `fie`, after calling `foo()`, the assignment $X = 0$ updates the global X.
- Thus, `write(X)` prints 0.

Answer:

- If $Y > 0$, the program prints 1.
- If $Y \leq 0$, the program prints 0.

3. Code Insertion for Static vs. Dynamic Scope Differences

We wish to fill the gaps in the fragment below:

```
{
  int i;
  (*)        // [Gap 1]
  for (i = 0; i <= 1; i++) {
    int x;
    (**)     // [Gap 2]
    x = foo();
  }
}
```

Objective:

- (a) Under static scope: the two calls to `foo` assign the same value to x.
- (b) Under dynamic scope: the two calls assign different values to x.

Solution: Declare an outer variable and define `foo` in the outer scope so that its free reference to x is resolved lexically (to the outer variable) under static scope. Meanwhile, when an inner block declares its own x, under dynamic scope, that inner x will be the "most recent" binding.

An acceptable solution:

`Gap 1:` Before the loop, insert an outer declaration and definition of `foo`:

```
int x;
int foo() { x = 10; return x; }
```

`Gap 2:` No additional code is needed (or simply a comment); the inner declaration `int x;` remains.

Explanation:

- Under static scope, the body of `foo` (written in the outer block) refers to the outer x. Even though a new x is declared in the for-loop block, it does not affect the already resolved free variable in `foo`.
- Under dynamic scope, the call to `foo` will use the most recent binding for x in the dynamic chain (which is the x declared inside the loop).

Answer:

- Under static scope, both calls to `foo` will update the same outer x.
- Under dynamic scope, the two calls will update different x's.

4. Denotable Object Outlasting Its References

Example: A dynamically allocated object (e.g., an instance of a class allocated on the heap) whose pointer (or name) is stored in a local variable may persist even after that variable goes out of scope if the object is linked into a global data structure. For instance, a node allocated via `new` in C++ may remain alive (until explicitly deleted) even though all local pointers to it have been lost.

5. A Name Outlasting Its Denotable Object

Example: A pointer variable that remains in a data structure (say, in a global table) even after the object it pointed to has been deallocated (or has gone out of scope) is an example of a name (the pointer) whose lifetime exceeds that of the object (leading to a dangling pointer).

6. Static Scope, Call by Value

Consider:

```
{
  int x = 2;
  int fie(int y) {
    x = x + y;
  }
  {
    int x = 5;
    fie(x);
    write(x);
  }
  write(x);
}
```

Analysis: Under static scope, the free occurrence of x in `fie` is bound to the x in its defining environment (the outer x, initially 2). In the inner block, a local x is declared (value 5), but it is not used in `fie`.

When calling `fie(x)`, the value 5 (from the inner x) is passed by value to y. Then `fie` does:

- $x = 2 + 5 = 7$.

Inside the inner block, `write(x)` prints the local x (value 5). After the block, `write(x)` prints the outer x (now 7).

Answer: The output is 5 followed by 7.

7. Dynamic Scope, Call by Reference

Now consider dynamic scope with call by reference:

```
{
  int x = 2;
  int fie(int y) {
    x = x + y;
  }
  {
    int x = 5;
    fie(x);
    write(x);
  }
  write(x);
}
```

Analysis: Under dynamic scope, the free x in `fie` is resolved in the calling environment. The call to `fie(x)` with call by reference makes y an alias for the inner x (initially 5). Then in `fie`, $x = x + y$ refers to the inner x, and it becomes 10.

Inside the inner block, `write(x)` prints 10. The outer x remains unchanged, so after the block, `write(x)` prints 2.

Answer: The printed values are 10 and then 2.