CMPSC 461: Programming Language Concepts, Spring 2024
Assignment 3 Practice Notes Packet

Prof. Suman Saha

September 15, 2025

Problem 1: Scopes and Bindings: Knowledge Check

Answer each question below;

1. For a non-static local variable in a C function, what is its scope and what is its lifetime?
2. For a static local variable in a C function, what is its scope and what is its lifetime?
3. For a non-static global variable in a C function, what is its scope and what is its lifetime?
4. For a static global variable in a C function, what is its scope and what is its lifetime?

Solution

1. A non-static local variable has the scope of the function or block in which the variable is declared. The lifetime of a non-static local variable is the function or block since it is destroyed when the function terminates or the block finishes execution.
2. The lifetime of a local static variable begins at its declaration during the first call to the function or block where it is defined. The variable remains in existence until the program terminates.
3. The scope of a non-static global variable is the entire C program. By using the extern keyword, it is visible even outside of the file in which the variable is declared. If a local variable has the same name as the global variable, the global variable will be invisible in the scope of the local variable. The lifetime of a non-static global variable is the entire program execution.
4. The scope of a static global variable is the scope of the file in which the variable is declared. It is invisible outside the file it is declared. Similar to (3), if a local variable has the same name as the global variable, the global variable will be invisible in the scope of the local variable. The lifetime of a static global variable is the entire program execution

Problem 2: Static and Dynamic Scoping

1. What determines an object's lifetime?
2. How do we keep track of what's visible and where in a program? Name the type of structure used and how it stores the information.
3. What are the key differences between lexical and dynamic scoping?
4. What causes the dynamic scoping symbol table to change after compilation?

Solution

1. The lifetime of an object is how long it stays in memory and is usable, linked to how variables are scoped and managed.

   Static Scoping: An object's lifetime is determined by its declaration and lasts until the end of the block or function where it is declared.

   Dynamic Scoping: An object's lifetime is based on the execution flow and call stack, meaning it lasts as long as its scope is active during function calls and the current execution path.

2. The scope of variables (what's visible and where) is tracked using a symbol table. It stores details about variables, functions, and other identifiers, including their scope, data type, and memory location. As the code is compiled, the symbol table is built to help resolve names and their locations within different scopes.

3. In lexical scoping, binding is based on nesting of blocks while in dynamic, it depends on the flow of execution at runtime. Additionally, lexical scoping is determined at at compile time while dynamic is determined at runtime.

4. In dynamic scoping, the symbol table can be modified during program execution. As functions are invoked, new variables may be added to the symbol table, and once those functions complete, their entries might be removed or updated.

Problem 3: Nested Scopes and Links

Consider the following pseudo-code, assuming nested subroutines and static scoping.

```
main() {
    int a = 5;
    int b = 3;
    function A(int x) {
        int b = x + 1;
        B(b);
    }
    function B(int y) {
        int c = a + b + y;
        print c;
    }
    function C(int z) {
        int a = z + b - 1;
        B(b);
    }
A(1);
C(2);
}
```

1. What does the program print?
2. Draw a diagram of the runtime stack when function `B` is last called. For each frame, show the static and dynamic links.
3. Refer to the runtime stack, briefly explain how function `B` finds variable `a and b`.
4. What does the program print when dynamic scoping is used?

Solution

1. **In main():**

   `int a = 5;`
   - `int b = 3;`
   - Calls `A(1)`, so `x = 1` in `A`.

   In A(x):

   `int b = x + 1;` → `b = 1 + 1 = 2`. This is a new local `b`, which does not affect the global `b = 3`.
   - Calls `B(b)`, so `y = 2` in `B`.

   In B(y):

   `int c = a + b + y;`
     - `a = 5` (from `main()`)
     - `b = 3` (from `main()`)
     - `y = 2` (passed as argument from `A`)
   - `c = 5 + 3 + 2 = 10`

  - `print c;` → Outputs `10`.

Next, `main()` calls `C(2)`.

In C(z):

`a = z + b - 1;` → `a = 2 + 3 - 1 = 4` (this updates `a` inside `C`, but does not affect the global `a`).

  - Calls `B(b)`, so `y = 3` in `B`.

In B(y) (second call):

`int c = a + b + y;`

- `a = 5` (from `main()`)
- `b = 3` (from `main()`)
- `y = 3` (passed as argument from `C`)

  - `c = 5 + 3 + 3 = 11`
  - `print c;` → Outputs `11`.

Program Output: `10 11`

2. At the time when `B` is called from `C(2)`, the following stack would exist:

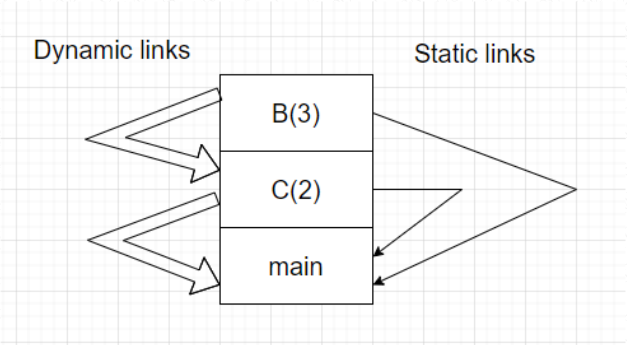
   main(): Variables: `a = 5`, `b = 3` Dynamic link: None (it's the first function call)

   C(2):

   - Variables: `z = 2`, `a = 4` (local to `C`)
   - Static link: `main` (because `C` is declared in `main`)
   - Dynamic link: `main` (because `main` calls `C`)

   B(3) (last call):

   - Variables: `y = 3`, `c = 11`
   - Static link: `main` (because `B` is declared in `main`)
   - Dynamic link: `C` (because `C` calls `B`)



3. For `a`: `B` looks to the enclosing function `main()` (via the static link) and finds `a = 5` in `main`.
   For `b`: Similarly, `B` follows the static link to `main()` and finds `b = 3`.
   In **B(y)** (first call from `A`):

   - `a = 5` (from `main`).
   - `b` would now be the value of `b` from `A`, which is `2`.

- `c = a + b + y = 5 + 2 + 2 = 9`.
- `print c;` → Outputs `9`.

In **B(y)** (second call from `C`):

- `a = 4` (local to `C`, because dynamic scope follows the most recent call).
- `b = 3` (from `main`, because `C` doesn't modify `b`).
- `c = a + b + y = 4 + 3 + 3 = 10`.
- `print c;` → Outputs `10`.

Program Output with dynamic scoping: `9 10`

Problem 4: Nested Scopes and Links

Consider the following pseudo-code, assuming nested subroutines and static scoping:

```
main() {
    int g = 46;
    int x = 61;
    function Z(int a) {
        int x = a * 3;
        S(x);
    }
    function M(int n) {
        int g = n
        if(n % 2 == 0){
            C(n / 2) ;
        }
    }
    function P(int r) {
        print x;
        M(r);
    }
    function S(int k){
        int q = k - 8;
        M(q);
        print k;
    }
    function C(int l){
        int x = l;
        print g;
        P(l);
    }
    // body of main
    Z(10);
    print g;
}
```

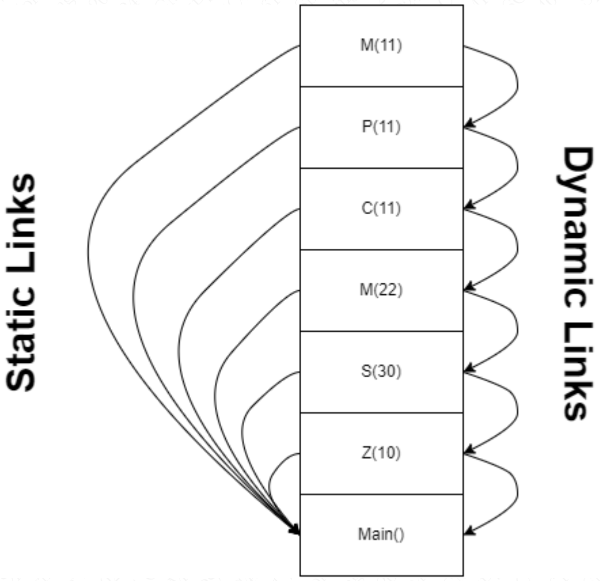
1. What does the program print?
2. Draw a diagram of the runtime stack when function `M` is last called. For each frame, show the static and dynamic links.
3. Refer to the runtime stack, briefly explain how function `P` finds variable `x`.

Solution

1. "46 61 30 46" or "46613046"
   Sine we are using static scoping, what variable refer to in each function is determined before runtime. Let's take a look at the code path.
   - $Z(10)$ calls $S(30)$
   - $S(30)$ calls $M(22)$

- $M(22)$ calls $C(11)$
- $C(11)$ prints g defined at line 2 and then calls $P(11)$
- $P(11)$ prints x defined at line 3 and then calls $M(11)$
- Returning to line 21, it prints parameter k
- Finally at line 30 it prints g defined at line 2

2. See the figure below.



3. Function `P` uses static links to locate the frame of `main`. Within that frame, it finds `x` which is at a statically known offset.

Problem 5: Nested Scopes and Links

Consider the following pseudo-code, assuming nested subroutines and dynamic scoping:

```
main() {
    int a = 1738;
    int b = 135;
    function good(int c) {
        function luck(int d) {
            int h = d + 4;
            print h;
        }
        function have(int e) {
            function fun(int f){
                int h = f / 2;
            }
            a = e * h;
            print a;
            fun(e);
        }
        int h = c + 3;
        print a;  //// Should give 0
        luck(c - 1);
        print h;
        have(h);
        print h;
    }
    function fifty_fifty(int g){
        int a = 0;
        good(g / 5);
        print a;
    }
    // body of main
    fifty_fifty(b);
    print a;
}
```

1. What does the program print?
2. Draw a diagram of the runtime stack when function fun is last called. For each frame, show the static and dynamic links.
3. Refer to the runtime stack, briefly explain how function fun finds variable a.

Solution
1.

Function `fifty_fifty(b)` is called:

$$a = 0$$

`good(135 / 5)` is called, which translates to:

`good(27)`

Inside `good(27)`:

$$h = 27 + 3 = 30$$

Prints:

$$a = 0$$

`luck(27 - 1)` is called, i.e., `luck(26)`:

$$h = 26 + 4 = 30$$

Prints:

$$h = 30$$

Back in `good(27)`, `luck` has finished, so it prints:

$$h = 30$$

print h, Prints:

$$h = 30;$$

`have(30)` is called:

Inside `have(30)`,

$$a = 30 \times 30 = 900$$

Prints:

$$a = 900$$

it calls `fun(30)`:

$$h = \frac{30}{2} = 15$$

Back in `good(27)`, `have` has finished, so it prints:

$$h = 30$$

Back in `fifty_fifty(b)`, it prints:

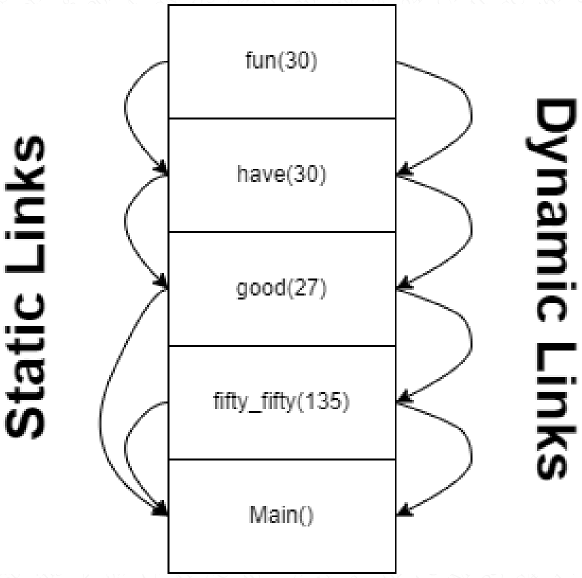
$$a = 900$$

Finally, the main program prints:

$$a = 1738$$

0 30 30 900 30 900 1738

2. At the point where `fun(30)` is called, the runtime stack looks like this:



Static links:

- `fun` → `have`
- `have` → `good`
- `good` → `fifty_fifty`

Dynamic links:

- `fun` → `have`
- `have` → `good`
- `good` → `fifty_fifty`

3. Since dynamic scoping is assumed, `fun` looks for the variable `a` by traversing up the dynamic call stack. It does not find `a` in its own scope, so it looks at the calling function (`have`), where `a` is defined as 900. Therefore, `fun` uses the value of `a` from `have`.

Problem 6: Binding rules

Consider following pseudo-code, assuming dynamic scoping rules:

```
procedure main()
  x:int := 4
  y:int := 6

  procedure four()
    x := x * y
    print(x)

  procedure six(f: procedure)
    x:int := 5
    four()

  procedure one()
    y:int := 1
    six(four)

  // main body
  one()
  print(x)
```

1. If the language uses deep binding, what will the output be? Explain your answer.

2. If the language uses shallow binding, what will the output be? Explain your answer.

Solution

1. With deep binding, the output is "4 4".
   Deep binding creates the binding when the procedure is referenced. In this case, `four` is referenced in `one`, so `x` and `y` in `four` are bound to `x` of `main` and `y` of `one`, respectively. Both of the `print(x)` statements print `x` in `main`, which equals to 4 when printed.

2. With shallow binding, the output is "5 4".
   Shallow binding creates the binding when the procedure is actually called. In this case, `four` is called in `six`, so `x` and `y` in `four` are bound to `x` of `six` and `y` of `one`, respectively. The `print(x)` statement in `four` prints `x` in `six`, which equals to 5 ($5 * 1 = 5$). The `print(x)` statement in `main` prints `x` in `main`, which is never modified and equals to 4.

Problem 7: Binding rules

Consider following pseudo-code, assuming dynamic scoping rules:

```
1  int x = 2;
2
3  function Fire(f) {
4      int x = 40;
5      f();
6  }
7
8  function Earth() {
9      print x;
10 }
11
12 function Water() {
13     int x = 20;
14     Fire(Earth);
15 }
16
17 Water();
```

1. If the language uses **shallow binding**, what will the output be? Justify your answer by showing the hierarchy of symbol tables at the print statement (assume each symbol table contains two columns: name and kind).
2. If the language uses **deep binding**, what will the output be? Justify your answer by explaining at **line 14**, which symbol table will be passed to function **Fire**.

Solution

1. The symbol tables are as follows :

global

name	**kind**
x	id
Fire	func
Earth	func
Water	func

Water

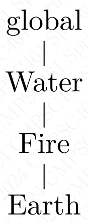
name	**kind**
x	id

Fire

name	**kind**
f	para (func)
x	id

Earth

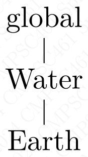
name	**kind**



global
|
Water
|
Fire
|
Earth

Since we are using shallow binding, *Earth* gets the environment at its call site. So output would be 40, as *Earth* is actually called at line 5.

2.



global
|
Water
|
Earth

In this case, *Earth* gets the environment at the time it's passed. Therefore, the output would be 20, as *Earth* is passed at line 14.

Problem 8: Scoping and Binding

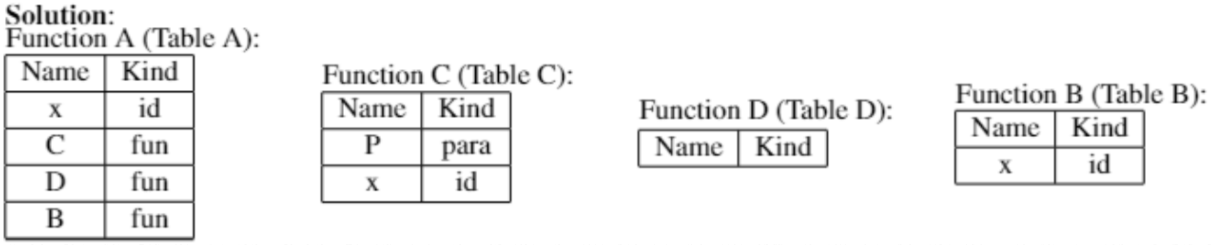
Consider the following pseudo-code with higher-order function support. Assume that the language has one scope for each function, and it allows nested functions (hence, nested scopes) :

```
function A () {
    int x = 5;
    function C (P) {
        int x = 3;
        P();
    }
    function D () {
        print x;
    }
    function B () {
        int x = 4;
        C (D);
    }
    B ();
}
```

1)What would the program print if this language uses dynamic scoping and shallow binding? Justify your answer by showing the tree of symbol tables when execution reaches the display expression.

2)What would the program print if this language uses dynamic scoping and deep binding? Justify your answer using the tree of symbol tables when execution reaches the display expression.

Solution The symbol tables are as follows :



Solution:

Function A (Table A):

Name	Kind
x	id
C	fun
D	fun
B	fun

Function C (Table C):

Name	Kind
P	para
x	id

Function D (Table D):

Name	Kind

Function B (Table B):

Name	Kind
x	id

1) The symbol table tree will be A → B → C → D. Therefore, the ”x” in function C will be printed. This x has a value of 3.

2) The symbol table tree will be A → B → D. Therefore, the ”x” in function B will be printed. This x has a value of 4.

Problem 9: Object Lifetime Tracing

Consider the following C++ code :

```
1  static myClass A;
2
3  int main() {
4      myClass B;
5      myClass* C = new myClass();
6      foo();
7      delete C;
8      return 0;
9  }
10
11 void foo() {
12     myClass* D = new myClass();
13     myClass E;
14 }
```

Consider one execution of the above program. The execution trace (a sequence of program statements executed at run time) of the program is

3 4 5 6 12 13 7 8

For each object associated with A, B, C, D and E, write down its lifetime using a subset of the above execution trace (e.g., "4 5 6 12 13"). Note, the answer subset might be non-strict, i.e. the whole trace.

Solution

A: Static Allocation
Lifetime: The entire duration of the program.
Execution Trace: The lifetime of A is the whole trace

3 4 5 12 13 6 7 8

B: Stack Allocation
Lifetime: From when it is declared in the main() function until main() terminates.

4 5 12 13 6 7 8

C: Heap Allocation for the object, Stack Allocation for the pointer
Lifetime of Pointer C: The pointer C itself is a local variable in main(), so its lifetime spans from its declaration at Line 5 until the end of main() (Line 8).
Execution Trace (Pointer):

5 12 13 6 7 8

Lifetime of the Object Pointed to by C: From when it is returned by foo() until it is deleted.
Execution Trace (Object):

13 6 7

D: Heap Allocation
Lifetime: From when it is allocated using new until the program terminates.

6 7 8

E: Stack Allocation in foo()
Lifetime: From when it is declared in foo() until foo() returns.

12 13

Problem 10: Static and Dynamic Scoping

Consider the following pseudo code. Assume that the language has one global scope, one scope per function, and one scope for each braced code block.

```
 1  int x = 10;
 2  int tom(int x) {
 3      {
 4          int x=50;
 5          jerry();
 6      }
 7  }
 8  int jerry() {
 9      print x+8;
10  }
11  tom(6);
```

1. If the language uses static scoping rules, what's the expected output from the print statement? Justify your answer.
2. If the language uses dynamic scoping rules, what's the expected output from the print statement? Justify your answer.

Solution

1. The output will be $10 + 8 = \underline{18}$, because x at line 9 refer to x at line 1.
2. The output will be $50 + 8 = \underline{58}$, because x at line 9 refer to x at line 4.

Problem 11: Static and Dynamic Scoping

Consider the following pseudo code:

```
int a;
int funcA(){
    int b, c;
    ...
    {
        int d, e;
        ...
    }
    int f;
    ...
    {
        ...
        int g, h;
        ...
    }
}
int funcB(){
    int i, j;
    ...
    {
        int k, l;
        ...
    }
    int m;
}
```

1. Draw a symbol table for each scope with entry consisting of name, type and data type. Include the global scope.
2. Order the tables in an tree based structure showcasing the hierarchy of scoping.

Solution 1. Symbol Tables

Global Scope:

Name	Type	Data Type
a	variable	int
$funcA$	function	int ()
$funcB$	function	int ()

funcA Scope:

Name	Type	Data Type
b	variable	int
c	variable	int
f	variable	int

Nested Block in funcA (Lines 5–8):

Name	Type	Data Type
d	variable	int
e	variable	int

Nested Block in funcA (Lines 12–15):

Name	Type	Data Type
g	variable	int
h	variable	int

funcB Scope:

Name	Type	Data Type
i	variable	int
j	variable	int
m	variable	int

Nested Block in funcB (Lines 20–23):

Name	Type	Data Type
k	variable	int
l	variable	int

2. Scope Hierarchy (Tree Structure)

Global Scope

↓

funcA Scope	funcB Scope
Nested Block (d, e) ↓ Nested Block (g, h)	Nested Block (k, l)

Problem 12: Static and Dynamic Scoping

Consider the following pseudo code:

```
1      int a = 50;
2      int b = 60;
3      int swap () {
4          int temp = a;
5          a = b;
6          b = temp;
7          print(a, b);
8      }
9      int main () {
10         int a = 10;
11         int b = 20;
12         swap();
13         print(a,b);
14     }
```

1. After executing main, what is the output of this code with static scoping?
2. After executing main, what is the output of this code with dynamic scoping?
3. What function scoping will *swap* use for variables a and b after *main* is called with dynamic scoping?

Solution

1. Line 7 will print 60, 50, as a and b in line 7 refer to a and b in global scope.
   Line 13 will print 10, 20, as a and b in line 13 refer to a and b in scope of *main*.

2. Line 7 will print 20, 10, as a and b in line 7 refer to a and b in scope at the call site of *swap* which is the scope of *main*
   Line 13 will print 20, 10, as a and b in line 13 refer to a and b in scope of *main*.

3. Since variables a and b are not present locally in *swap*, dynamic scoping will use the scoping of the previous caller in call frame which is the scoping of *main*.