Programming Languages

Recitation

Summer 2014

Recitation Leader

- Joanna Gilberti
- Email: jlg204@cs.nyu.edu
- Office: WWH, Room 328

- Web site: http://cims.nyu.edu/~jlg204/courses/PL/index.html

Homework Submission Guidelines

- Submit all homework to me via email (at jlg204@cs.nyu.edu only) on the due date.
- Email subject: “PL– homework #” (EXAMPLE: “PL – Homework 1”)
- Use file naming convention on all attachments:

  “firstname_lastname_hw_#” (example: "joanna_gilberti_hw_1")
- In the case multiple files are being submitted, package all files into a ZIP file, and name the ZIP file using the naming convention above.

What's Covered in Recitation

- Homework Solutions
- Questions on the homeworks
  - For additional questions on assignments, it is best to contact me via email.
  - I will hold office hours (time will be posted on the recitation Web site)
- Run sample programs that demonstrate concepts covered in class

Iterative Languages
Scoping

- Sample Languages
  - C: static-scoping
  - Perl: static and dynamic-scoping (use to be only dynamic scoping)
- Both gcc (to run C programs), and perl (to run Perl programs) are installed on the cims servers
- Must compile the C program with gcc, and then run the generated executable
- Must include the path to the perl executable in the perl script

Basic Scope Concepts in C

- block scope (nested scope)
- → run scope_levels.c (sl)
- ** block scope: set of statements enclosed in braces {}, and variables declared within a block has block scope and is active and accessible from its declaration to the end of the block
- Analysis:
  - Variable, 'i', is visible within outer block (within the main function)
  - Variables 'i', and 'j' are visible only within the inner block (enclosed by {})

Analysis (continued):

- Variable ‘i’ is defined in two places, but since they are declared in two different scopes the two variables cannot be accessed at the same time
- Demonstrated in output: ‘i’ prints 23 before and after the loop gets executed (loop is in the inner scope)
- The for-loop loops through ten different values for ‘i’ while the inner-scoped portion of the code is being executed
- After the for-loop (inner scope block) completes executing, the variables within the inner-scoped block are no longer accessible
- The value for ‘i’ in the outer-scoped block is now accessible and gets printed

Basic Scope Concepts in C

- program scope

  → run program_block_scope.c (pbs)

  program scope: variables declared outside of a function have program scope

Analysis:

- Variables x and y are declared globally with program scope
- The declared function, function_1() prints x and y – the values of the global variables x and y are printed (no local declarations)
    - The function function_1() is called twice from two nested blocks: (1) from the outer block in the main() function, and (2) from the inner block in the main() function.
        - The first function_1() call (outer block) prints the global values for both x and y called from the outer block
        - The second function_1() call (inner, nested block) prints the global values for both x and y again called from the inner nested block
    - The print statement in the main() function in the outer block prints the value of x that was locally declared in the main() function, but the global value of y
    - Finally, the print statement in main(), but in the inner, nested block prints the values for the locally declared variables x and y

Basic Scope Concepts in C

- the static specifier

→ run static_specifier.c (ss)

- the ‘static’ specifier indicates that the memory location of a variable is temporary, and so a variable’s reserved space in memory can be erased or relocated when the variable is out of its scope.

Analysis:

- The function add_two() is declared, and is called several times from the main() function
- Two arguments are passed to the add_two() function, which are added when the function executes, and the sum is returned
- A counter is set to keep track of how many times the function is called
- A variable is declared locally with the 'static' specifier in the add_two() function that has function scope – this is the counter that keeps track on how many times the add_two() function is called
- The values stored by the counter are retained because the duration of the variable is permanent – although the scope of the counter is within the block of the add_two() function, the memory location of the counter and value saved in the location are not changed after the add_two() function is called and the execution control is returned back to the main() function.
- The counter variable is used as a counter to keep the number of calls received by the add_two() function. The
- printf() function in the add_two() function prints out the value saved by the counter variable each time the
- add_two() function is called. In addition, counter is incremented by one each time after the printf() function is executed
- The for-loop in the main() function calls the add_two() function five times
- The values for i and j are passed, and their sum is returned and printed
- Evident from the output, the value for the counter is saved and incremented each time the add_two() function is
- called, and is retained after the function exits because the counter variable is declared with 'static'

Basic Scope Concepts in C

- more on ‘static’

→ run more_static_specifier.c (mss)

- When a variable is declared with ‘static’ – when the variable goes out of scope (the block it was local to is finished) the variable stays in memory, retaining its value. The variable stays in memory until the program ends.
- The behavior resembles global variables, but static variables still obey scope rules and therefore cannot be accessed outside of their scope

Analysis:

- The j variable is accessible by both the up() and down() functions and retains its value – it is declared with ‘static’ but globally
- The k variable also retain their values but are two different variables in each of their own scopes
- In the up() function, both k and j are incremented, and in the down() function both k and j are decremented. The difference is:
  - k has separate scopes in each function, and even though their values retain for each function, they start from a different initial value because they have separate scopes.
  - j has one scope (program scope) – even the value of j is retained, and across both functions

Static vs. Dynamic Scoping

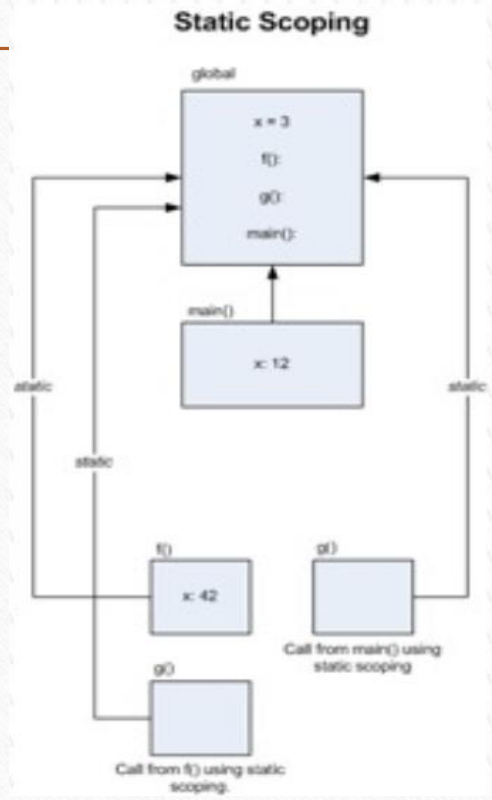
- two programs that do exactly the same thing:
  - one written in C (static scope)
  - one written in Perl (dynamic scope)

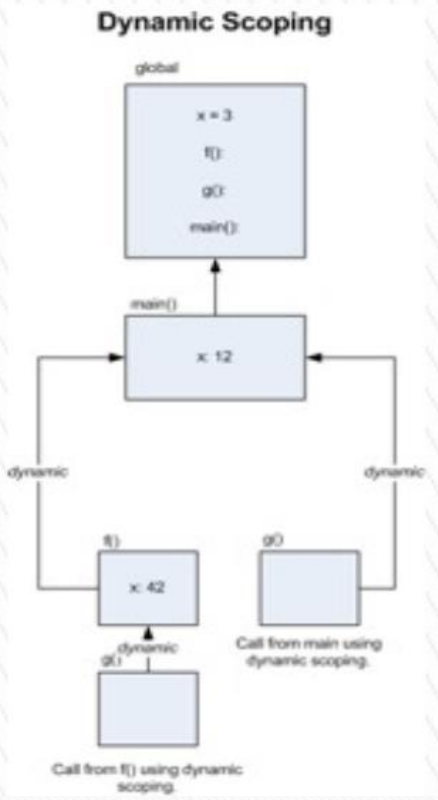
→run staticscope_cversion.c (sscv)

→dynamicscope_pversion.pl

Analysis: (see diagrams)

Static vs. Dynamic Scoping





Perl With Static and Dynamic Scope

- Two perl programs that do exactly the same thing:
  - program that demonstrates statically scoped

    → samplestaticscope.pl
  - program that demonstrates dynamically scoped

    → sampledynamicscope.pl

Analysis:

- Static scoping: active bindings are determined using the text of the program at compile time - Current binding - the matching declaration whose block most closely surrounds the point in the program where the name is mentioned.
- Dynamic scoping: active bindings are determined by the flow of execution at run time – Current binding - the one encountered the most recently during the execution and that has not yet been destroyed.
- 'my' makes a variable statically scoped
  - Only available to the subroutine (scope) in which it is declared
  - Not available to subroutines you call
  - Not available to subroutines that called you
  - Destroyed when the execution exits the block it's in
- 'local' makes a variable dynamically scoped
  - A "temporary global"
  - Available to subroutines you call
  - Not available to a subroutine that called you
  - Can shadow and protect an existing global
  - Destroyed when execution exits the block it's in