Midterm grades have been posted (Average 245, Median 260). Sample solutions are available on Canvas; **exams will be handed out in recitations today**; look at sample solution before asking questions. Grading was done as follows:

- Problem 1: Xinyu
- Problem 2: Zining
- Problem 3: Chun
- Problem 4: Uli

Fourth homework: Will cover pointers and scoping
Names, Bindings, and Memory

Scott: Chap. 3.1 - 3.4; ALSU Chap. 7.1 - 7.3

What’s in a name? — each name “means” something!

- has a meaning, i.e., represents a semantic object (e.g.: a type description, an integer value, a function value, a memory address, etc.)
- has associated “attributes” (e.g.: type, memory location, read/write permission, storage class, access restrictions, etc.)
Names, Bindings, and Memory

**Binding** – association of a name with an attribute (e.g., a name and a memory location, a function name and its “meaning”, a name and a value)

- **Compile time** – during compilation process – **static** (e.g.: macro expansion, type definitions)
- **Link time** – separately compiled modules/files are joined together by the **linker** (e.g., adding the standard library routines for I/O (stdio.h), external variables)
- **Run time** – when program executes – **dynamic**

Compiler needs bindings to know meaning of names during translation and execution.
Binding Times - Choices

- **Early binding** times – more efficient (faster) than at runtime
- **Late binding** times – more flexible (postpone binding decision until more “information” is available)

- Examples of static binding (early):
  - functions in C
  - types in C

- Examples of dynamic binding (late):
  - method calls in Java or virtual function calls in C++
  - actual types of objects pointed to by a Java reference variable (class hierarchy)
  - dynamic typing in Scheme

Note: **dynamic linking** is somewhat inbetween static and dynamic binding; the function signature has to be known (static), but the implementation is linked and loaded at run time (dynamic).
How to Maintain Bindings

- **symbol table**: maintained by compiler during compilation
  names $\Rightarrow$ attributes

- **environment**: maintained by compiler generated code during program execution
  names $\Rightarrow$ memory locations

- **memory**: maps memory locations to values memory
  locations $\Rightarrow$ values

Questions

- How long do bindings for a name hold in a program?
- What initiates a binding?
- What ends a binding?
Scope Example

Block Structures Programming Languages

program L;
    var n: char; {n declared in L}

procedure W;
    begin
        write(n); {n referenced in W}
    end;

procedure D;
    var n: char; {n declared in D}
    begin
        n:= 'D'; {n referenced in D}
        W
    end;

begin
    n:= 'L'; {n referenced in L}
    W;
    D
end.
Lexical Scope

- Non-local variables are associated with declarations at *compile* time
- Find the smallest block syntactically enclosing the reference and containing a declaration of the variable

**Example:**
- The reference to \( n \) in \( W \) is associated with the declaration of \( n \) in \( L \)
- The output is?
Dynamic Scope

- Non-local variables are associated with declarations at run time
- Find the most recent, currently active run-time stack frame containing a declaration of the variable

Example:

- The reference to n in W is associated with two different declarations at two different times
- The output is?
Run-time stack contains frames for main program and each active procedure.

Each stack frame includes:

1. Pointer to stack frame of caller (control link for stack maintenance and dynamic scoping)
2. Return address (within calling procedure)
3. Mechanism to find non-local variables (access link for lexical scoping)
4. Storage for parameters, local variables, and final values
Context of Procedures

Two contexts:

- *static* placement in source code (same for each invocation)
- *dynamic* run-time stack context (different for each invocation)

Scope Rules

Each variable reference must be associated with a single declaration (ie, an offset within a stack frame).

Two choices:

1. Use static and dynamic context: *lexical scope*
2. Use dynamic context: *dynamic scope*

- Easy for variables declared locally, and same for *lexical* and *dynamic* scoping
- Harder for variables not declared locally, and not same for *lexical* and *dynamic* scoping
Next Lecture

More on runtime environments

Please keep working on the project