Names, Scopes, and Bindings I

Hwansoo Han
Name, Scope, and Binding

- **Name**
  - A mnemonic character string to represent something else
  - Names in most languages are identifiers
  - Symbols (like '+') can also be names
    - ‘+’ represents an add operation

- **Binding**
  - An association between a name and the thing it names

- **Scope of a binding**
  - The part of the program in which the binding is active
Named vs. Unnamed Data

- Programming languages has the ability to name data
  - Refer to data using symbolic identifiers rather than addresses

- Not all data is named!
  - Dynamic storage in C or Pascal is referenced by pointers, not by names
Times for Implementation Decisions (1)

- Language design time
  - Program structure,
  - Possible type

- Language implementation time
  - Number of bits for fundamental types
  - Coupling I/O to system files,
  - Max and min sizes of heap and stack,
  - Runtime exceptions (e.g. arithmetic overflow),
Times for Implementation Decisions (2)

- Program writing time
  - Algorithms,
  - Data structures,
  - Names

- Compile time
  - Machine code,
  - Data layout in memory

- Link time
  - Layout of all modules in memory
  - Binding a name to an object in another module

- Load time
  - Choice of physical addresses
Times for Implementation Decisions (3)

- **Run time**
  - Binding between a value and a variable
  - Size of string

- **Run time subsumes**
  - Start-up time
  - Module-entry time
  - Elaboration time (declaration is first "seen" with init-value)
  - Subroutine call time
  - Block entry time
  - Statement execution time
Binding Time

Binding time
- Time at which a binding between two things is made
- All implementation decisions in PL

Representative two bindings
- Static binding - binding is made before run time
- Dynamic binding - binding is made during run time
Binding – efficiency vs. flexibility

- Early binding times
  - Generally lead to greater efficiency

- Later binding times
  - Generally lead to greater flexibility

- Compiled languages
  - Tend to have early binding times

- Interpreted languages
  - Tend to have later binding times
Object Lifetime (1)

- **Key events for objects**
  - Creation of objects
  - Creation of bindings
  - References to variables (which use bindings)
  - (Temporary) deactivation of bindings
  - Reactivation of bindings
  - Destruction of bindings
  - Destruction of objects

- **Lifetime of an object**
  - Period between creation and destruction of the object
Object Lifetime (2)

- Lifetime of a binding
  - Period of time from creation to destruction of a binding
  - If an object outlives binding, it becomes a garbage
  - If binding outlives object, it becomes a dangling reference

- Scope of a binding
  - Textual region of a program in which the binding is active

- Object lifetime generally corresponds to storage allocation mechanisms
  - Static object, Stack object, Heap object
Storage Management: Static

- Static allocation
  - Code
  - Global variables
  - Static variables
  - Explicit constants (including strings, sets, etc.)
- Scalars
  - Small scalars may be stored in the immediate fields of instructions
  - E.g. ADD r1, r2, 4
Storage Management: Stack-Based

- Stack-Based Allocation
  - Local variables

- Central stack for
  - Parameters
  - Local variables
  - Temporaries

- Why a stack?
  - Allocate space for recursive routines
  - Reuse space
Stack Frame

- Contents of a stack frame (activation record)
  - Arguments and return values
  - Local variables
  - Temporaries
  - Bookkeeping (saved registers, static link, etc.)

- Reference mechanism
  - Fixed locations within a stack frame
  - Locations are assigned at compile time
  - Access with displacement addressing mode (base-offset)
    - fixed offsets from the stack pointer (sp), or frame pointer (fp)
Stack Frame (cont’d)

- Direction of stack growth (usually lower address)
  - fp
  - sp

- Arguments to callee
- Temporaries
- Local variables
- Bookkeeping
- Return address

- fp (when C() is running)

- A()
- B()
- C()
- D()
- E()
Stack Maintenance

- Maintenance of stack is responsible for
  - \textit{Calling sequence} at call site (caller)
  - Subroutine (callee) \textit{prolog} and \textit{epilog}

- Save \textit{space}
  - Putting as much stuff in prolog & epilog as possible

- Save \textit{time}
  - Putting stuff in the caller instead what's known at both caller and callee (e.g., put values in registers and execute func)
Storage Management: Heap-Based

- Heap is used for dynamic allocation
  - In-use blocks
  - Free blocks

- Fragmentation
  - Internal fragmentation (cross-hatched space)
  - External fragmentation
    - Due to discontinuous free blocks, a request block cannot be allocated even if the total free blocks are more than the size of requested block
Garbage Collection

- Objects for heap-based allocation
  - Dynamic allocation is explicitly specified
  - Explicit deallocation (freeing object) may be omitted

- Garbage collection
  - Implicit deallocation of objects
  - Identify garbage (unreachable objects) and reclaim space

- Garbage
  - Objects no longer used – hard to determine at run-time
  - Unreachable objects – easier to determine
    - Guaranteed no use later (there are no ways to reference them)
Summary

- Three concepts
  - Name
  - Binding
  - Scope

- Binding times
  - Several times of binding from design to run-time
  - Two bindings: static- and dynamic-binding

- Object lifetime and storage management
  - Static, stack-based, heap-based management