Chapter 3 - Names, Scopes, and Bindings

Early languages ... "high level" vs "low level" assembly

- abstraction: names instead of memory locations (although assembly had names)
- easier constructs for understanding
- other advantages came later ... e.g. portability

Names -- primary abstraction

- Assembly language primarily had "labels" -- names that represent machine locations
- High level languages added abstraction to names
  - Names are mostly Identifiers but '+' can be a name
- mostly names are alpha-numeric, mostly first character alpha
- binding -- when and to what a name refers
- referencing environment -- complete set of bindings

Binding time

- binding -- an association between the name and what it names
- binding time -- when this association is created
- Language design time
- Language implementation time
- Coding time
- Compile time
- Link time
- Load time
- Run time

Binding time (page 2)

- Earlier binding time -- more efficiency
- Later binding times -- more flexibility
- static vs dynamic -- before run time vs at run time
  - But "static" may not seem to be static, e.g auto variables
- Time difference between compiler and interpreter solutions
  - compile time memory layout
  - interpreter may evaluate "declaration" multiple times
    - late binding of types can provide polymorphism
  - not all storage is bound to a name, C(malloc), Pascal(new) ...

"Object Lifetime" / Storage Management

- names vs objects
  - creation and destruction of objects
  - creation and destruction of bindings (may not do objects)
  - deactivation and reactivation of bindings
  - references to variables, subroutines, types, ...
- Object lifetimes
  - Static: "absolute address", lifetime is program
  - Stack: subroutine call allocated, return deallocated, LIFO
  - Heap: arbitrary times allocated and deallocated

Storage Management (page 2)

- Static allocation / bindings
  - global variables
  - function code
  - "local" static/own variables
  - large constants (small ones in instruction stream)
  - "invisible" support structures
  - Fortran before Fortran 90 -- local variables (no recursion)
    - Assembler functions often used static variables
  - Constants and constant values
    - "constant" dependent on runtime value
    - constant means different things (bc)
- Stack allocation
  - required for recursion
    - each call to a subprogram uses different memory
    - allocated on entry, deallocated on exit
  - typical use of stack for subprograms -- "Call frame" or "activation record)
    - arguments to subprogram
    - return address
    - bookkeeping information
    - local variables
    - local temporaries
    - Most CPUs have a "Frame Pointer" register (fp)

Storage Management (page 3)

- Stack allocation (page 2)
  - subprogram
    - preamble -- sets up frame on entry (prologue)
    - body -- subprogram code
    - postamble -- cleans up the stack (epilogue)
  - location on the stack can not be determined at compile time
  - access is usually +/- from the fp
  - useful for languages without recursion

- Heap allocation
  - required for dynamically allocated data
  - many different methods to manage a heap
  - Issues: speed, fragmentation, re-use, re-use method
  - placement: architecture dependent
  - garbage collection -- no explicit free, find unreferenced memory and auto free
  - automatic vs manual allocation/free
    - speed
    - error prone
    - algorithm complexity

Scope Rules

Scope: textual region where a binding is active

- static vs dynamic (Most languages are static)
- What creates scope, closes scope
  - subprogram?
  - {...}
  - specific scope declarations: namespace X { .... }
- Possible, multiple scope levels -- referencing environment
- Binding rules -- when is scope enforced: deep or shallow

Static Scope (aka lexical scope)

- determined at compile time by syntax
- simplest: basic -- one scope, global, no declarations
- Fortran pre 90: all global, subprogram local scope, no declarations, i-n integers
  - named common across compile units,
- Fortran 90 changed rules

Scope Rules (page 2)

Static Scope (continued)

- Algol 60 allowed recursion
  - local scope, unique objects per call
  - "own" variable -- global but in subprogram scope (C static)
  - Added nested subprograms, with new scope
    - many languages now have this
    - name resolution rules
      - closest nested scope
      - inner declaration may hide outer
      - ways to select scope, scopename:name, ::X ...
- "built-in" / "predefined" scopes
- name visibility in scope
  - full scope
  - declaration to end of scope
  - mutually recursive functions, records with pointer to self
    - forward declarations
  - declare before use can have issues with full scope
    - fpc likes scope.p ... but most likely shouldn’t
    - many languages do declaration to end of scope
  - some (C#) silently uses "local declarations"
    - class a { const int N = 10; void foo() { const int M=N; const int N = 20; ... (M is 20)

Scope Rules (page 3)

- Python: no declarations, x = ... in T and in S inside T, 2 unique x objects
- declaration vs definition
  - C: struct x; ...... struct x {....};
- redeclarations -- may cause problems

Access to non-local objects (subprograms)

- Access to global is easy ... direct
- Non-local, non-global access is harder

  func a { int b; func c { var d; func f { var g; ... b = d + g; } } }

  - Stack based storage, activation record
  - what if f is recursive?
  - static chains
  - displays (not used that often, book doesn’t mention them here)

Modules -- changing scoping and access rules

- Early programs -- single file
- As programs grew, modularity helped control complexity, separate compilation
  - Advantage: speed, don’t have to recompile entire program for a small change
- Various versions of this appeared: C simple separate compilation
- Modules -- way to collect related functionality for separate compilation
  - Added "information hiding"
  - Added new scoping rules

Scope Rules (page 3)

Modules (continued)

- Contained a variety of "objects" -- subroutines, constants, variables ...
- Typically a way to control what was visible from "outside" (export)
- A way to get access to a Module (import)
- Appeared late 70s, early 80s
  - Clu, Modula, Modula-2, Modula-3, Turing, Ada 83 ...
- Term package replaces module in some current languages
- C++ has "namespace" ... multiple file can define it, using clause for access
- Separate compilation
  - libraries and parts of a program
  - can recompile one without recompiling the other
  - Modula-2 had definition files and implementation files
  - when does a change require recompilation?

Objects

- Next idea for modularity and re-usability
- new features: inheritance and dynamic method invocation
- Someplace in here we got operator overloading as well as method overloading
- some had new visibility rules
- introduced ideas of setters and getters

Scope Rules (page 4)

Dynamic Scope

- binding depends on run time
- binding is most recent binding that is active
- Languages: bc, APL, Snobol, Tcl, TeX, early lisp, perl
- Type checking may need to be done at run time
- Other things may be dynamic too, var2.bc

Implementing Scope

- Static scope: symbol table
  - enter ID, new_scope, exit_scope
  - key/value DB
  - can be saved in executable for runtime lookups in debugger
  - Compiler Construction class spends lots of time on this
- Dynamic scope: runtime DB for lookups, can be expensive
  - Can key/value DB
  - may be list of objects (linear may not be bad here)
  - depends on the semantics of the language
- aliases:
  - two or more names that refer to a single "object"
  - can cause issues with optimization
  - C99 added a "restrict" qualifier ... no aliases
  - parameters can cause this also

Scope Rules (page 5)

- overloading:
  - same name/feature for two or more objects, often subprograms
  - + works on multiple types
  - need some mechanism of selection, e.g. parameter types
  - Ada: allows same name in different enumeration types
    - type Month is (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sept, Oct, Nov, Dec)
    - type Base is (Bin, Oct, Dec, Hex);
    - can determine correct on via type of expressions
    - can specify. e.g Month’(Dec)
    - some languages require type prefix (Modula-3, C#)
  - Ada, C++ lots of overloading, subprograms and operators
  - Haskel allows "creating" infix operators that call functions
    - let a @@ b = a * 2 + b
  - Haskel also allows overloading with types
  - overloading vs coercion, polymorphism (covered later)

Binding of referencing environments with subprogram parameters/pointers

Reference to a subprogram

- Some languages allow "procedure/function parameters"
- Some languages allow pointers to subprograms
- deep vs shallow binding of the reference environment (deep-shallow.txt)
- static scoping needs deep binding here
- subroutine closure: [reference to subprogram, reference to reference environment]
- dynamic bindings: Need the bindings at the time of call
  - e.g. IF bc had function pointers, need current set of names bound
- static bindings: depends on language
- C: has function pointers
  - functions are never nested
  - environment: local and global
  - call creates local, global is global
  - no closure needed, just pointer to function
- Languages with nested functions/procedures have issues
  - need the referencing environment at time when procedure is passed

Binding of referencing environments (page 2)

- Example 1, Python: (parameters)

```
def A(I,P):
    def B():
        print(I)
    if I > 1:
        P()
    else:
        A(2, B)
def C():
    pass  # do nothing
A(1,C)
```

- What is printed? 1 or 2?
  - shallow: 2
  - deep: 1
    - Why?

Binding of referencing environments (page 3)

- Example 2, Pascal-like (local variables)

```
var  i : integer;  (* global *)
procedure Z ( procedure X ) { .... X(1) ... }
procedure P() {
    var j : integer;
    procedure R(value m:integer) {
        if m >= 3 then { writeln(j * m); X(R); }
        writeln(m);
        R(m+i);
    }
j := i * 2;
    R(1);
}
... i := 10 ... P();
```

- Access to i?
- Access to i and j in P()
- Access to i and j in R()
- Static link -- pointer to next out enclosing scope
- Display -- array of pointers to currently active scope
- Referencing environment of R when called in Z as X?
  - closure: pointer to procedure & static link or copy of display
- Not a problem in C or Modula-2 (only level 1 procedures as parameters)
- Not a problem in languages which don’t pass subprograms

Kinds of values

First-Class Value

- passed as a parameter
- returned from a subroutine (e.g. function)
- assigned into a variable

Second-Class Value

- passed as a parameter
- not returnable or assignable

Third-Class Value

- can’t be passed as a parameter, returned or assigned (e.g. Label)

Subprograms show the most variance

- 1st Class - C#, Fortran, Modula-2, Modula-3, Pascal Ada 95, C, C++
- 2nd class in other imperative languages
- 3rd class in Ada 83.
- Some dynamic languages may have a dangling subprobram closure. (references to procedures returned)
- Read examples 3.32 - 3.41 and section 3.10

Macro Expansion

- Macros started in assemblers
  - Textual replacement for repetetive instruction sequences
- Macros moved to high level languages
  - Textual replacement
  - Can Cause issues
- Example: C
  - #define NAME value // avoids "named constants"
  - #define SWAP(a,b) {int t = (a); (a) = (b); (b) = t; }
  - Call? SWAP(m++,n++)?
- Most modern languages do not have macros.