CSE 3302 Notes 4: Names & Scope

(Last updated 10/1/15 1:18 PM)

References:

Gabbrielli-Martini: 4

Aside: Ever heard of Scopeware?

4.1. BINDING TIME - “early” or “late”

Binding = Commitment: Existence, Type(s), Value, Representation, Location, Mutability

Design:

Language
Libraries

Program Writing

Build:

Compilation
Linkage

Runtime:

Loading
Execution

4.2. OBJECT LIFETIME AND STORAGE MANAGEMENT

Issues

Recursion
Threads/Processes/Reentrant Code
Separate Code and Data Address Spaces (such as `code` and `s` in Pascal-S and PL/0)
Virtual Memory, Caches, and Mappings

Static Allocation

Characteristics - single instances, fixed size, global or side-effect (non-pure-functional)

Stack(-Dynamic) Allocation

Useful for support of recursion and functions in general

Size of stack frame (activation record) and offsets for a function are usually known at compile-time

C - historically no function nesting, so just local variables, globals, or statics with additional scope levels ("block structure") allowed within functions (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/block.c`)

Allocate maximum possible space immediately upon entering function, or
Allocate depending on control flow (`alloca()` to extend stack frame)

PL/0 (similar for Pascal-S and Pascal) - scopes nest only for procedures/functions

Heap(-Dynamic) Allocation

Most flexible "temporally" - for pointer-based data structures

```
#include <stdio.h>
#include <stdlib.h>

char bigStatic[2000000000];

main()
{
char bigStack[10000000];

char *bigHeap;

printf("Ready to malloc\n");
bigHeap=(char*) malloc(10000000);
printf("malloc successful\n");
}
```

4.3. Scope Rules

When is a particular binding of name to . . . relevant?

Referencing environment: (Gabbrielli, p. 70)

Associations (bindings) between names and (denotable) objects at

1. Position in program.
2. Time during execution.

but could be complicated by nesting and polymorphism/overloading.
Includes *global*, *non-local*, and *local* components (Gabbrielli, p. 73)

Lexical/Static Scoping

(Gabbrielli, p. 78) - A use of a name:

is mapped *uniquely* to a declaration (run-time ordering does not matter)
has instances respecting lexical nesting (e.g. for recursion)

Globals

Variables within functions

Nested blocks

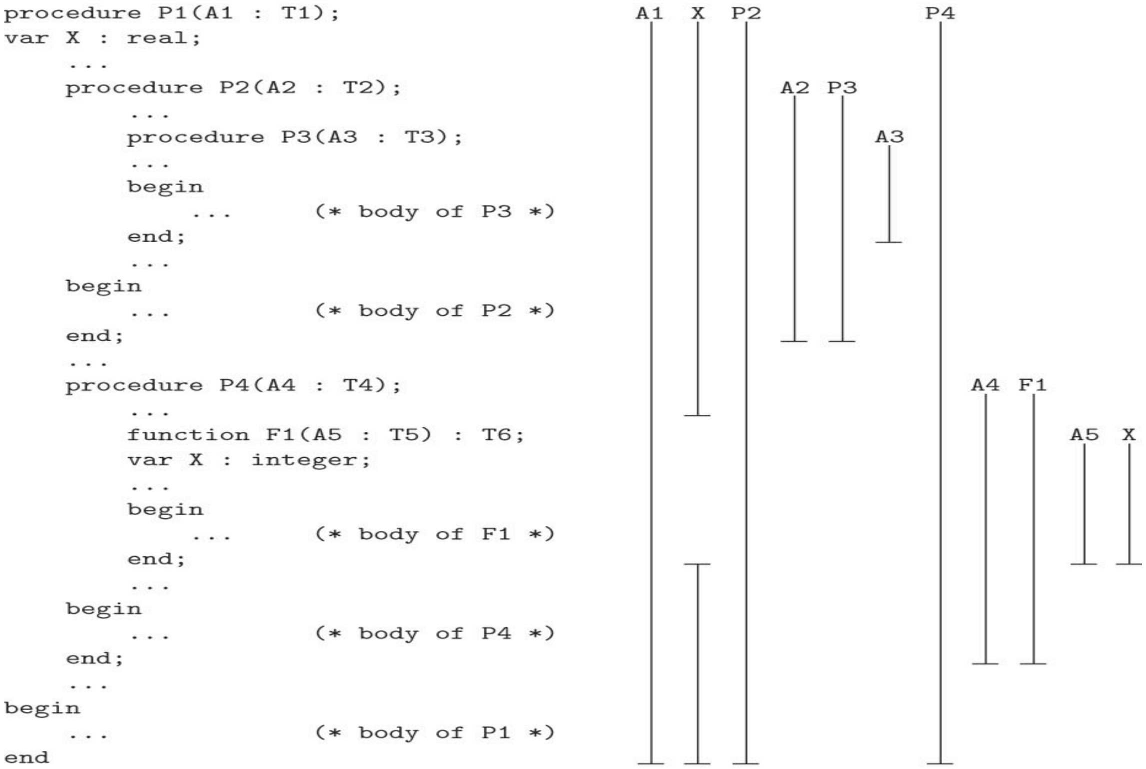
Existence independent of execution:

C `static` refers to allocation
Class variables

Nested Subroutines

Non-nesting: A significant connection between C to COBOL, FORTRAN, and assembler.
Pascal: `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/notes04.pas`
JavaScript: `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/notes04.html` demonstrates how global names are properties of "window" object

(Gabbrielli, p. 75-76 is very detailed regarding names and associations to objects, along with problems regarding dangling references to inappropriate use of an expired binding to storage.)



```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...     (* body of P3 *)
        end;
        ...
    begin
        ...         (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...     (* body of F1 *)
        end;
        ...
    begin
        ...         (* body of P4 *)
    end;
    ...
begin
    ...             (* body of P1 *)
end
```

Gabbrielli, p. 82-85 is useful for these issues.

Pascal also has `forward` declarations to allow mutual recursion without nesting or to deal with complicated situations like the Pascal-S interpreter:

1. `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/pascal-s.structure.txt` provides nesting structure (see `expression`).
2. `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/pascals.pdf` provides the call graph. (Contrast with Appendix B of the Pascal-S report)
3. `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES02/pascals.pas` gives complete code.
4. `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/pascals.dot` is the call graph as input to Graphviz.

Important detail - for such code (potentially with a variety of call paths and recursion), how are necessary bindings referenced at run-time? (Gabbrielli, chapter 5)

Declaration Order

Pascal - Scope of declaration is entire surrounding block. Can't use until declared.

C - Scope of declaration begins with the declaration, but definition may appear later.

JavaScript

Declarations are "hoisted" to beginning of a function or global scope
(see Crockford, p. 102)

Block scoping may be kludged using an *immediately invoked function expression* (IIFE, `http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/iife.html`)

Scheme

`define` ordering does not matter - names available throughout block
`let` has its own nested scope, but comes in other variations

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/notes04.rkt`

Dynamic Scoping (aside - Perl craziness)

Gabbrielli, p. 80, def 4.5

A use of a name:

is mapped to a declaration based on run-time ordering
has instances operating in a stack-like fashion (according to run-time ordering)

Each name operates LIFO as contexts are entered and exited.

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/array.txt`

4.4. Meaning of Names Within a Scope

Aliases

```
x^=y;
y^=x;
x^=y;
```

(tagged) unions

Overloading

Arithmetic operations applying to multiple types
C++ - use `[]` to treat binary search tree as array

Polymorphism

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/poly.cpp` - *type signature* to determine which of identically named functions gets called

4.5. Opening Scopes

Pascal `with` (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/with.pas`)

Opens one or more instances of record structures to simplify referencing
Ambiguity is resolved by nested with/LIFO assumption

JavaScript `with` (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/with.html`)

Property must already exist . . . otherwise a global variable results

`site.ebrary.com.ezproxy.uta.edu/lib/utarlington/reader.action?ppg=126&docID=10763621&tm=1435436076995`

C++ (aside)

Namespaces:

Allows grouping of classes, functions, data, and types under a name to avoid name conflicts.
There may be several declarations for a particular namespace.
Qualified names outside `namespace` declarations may only be uses (not definitions)
Each class is a namespace. `::x` refers to a name `x` in the global namespace

`using` Declarations - simply short-cut a path of qualifications (`::`)

`using` Directives

Open entire namespace
May easily introduce name conflicts, so bad practice to put in header files

C++ Argument-Dependent Look-Up - If use of a function name is not resolved within its containing scopes, then try the namespaces of its arguments. (Consider operator overloading.)

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/slams.notes04.cpp`

`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/rational.notes04.cpp`

4.6. HEAP ALLOCATION AND SUBROUTINE CLOSURES

Subroutine Closures

Problem with lexical/static binding

Static chain pointer (notes 5) created *at same time as* reference to function (*closure*)

Difficulty when reference lasts longer than stack frame

Solution - anything needed for closure gets heap allocation

JavaScript (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/closure.html`)

`site.ebrary.com.ezproxy.uta.edu/lib/utarlington/reader.action?ppg=198&docID=10763621&tm=1435436200192`

Scheme (`http://ranger.uta.edu/~weems/NOTES3302/NEWNOTES/NOTES04/closure.rkt`)