



Informatik II

Languages, Compilers, and Theory

Lecture 6:

Scopes and Storage Management



Review

- Last time:

- Names

- A mnemonic character string which "names" or refers to something else
 - Example: variable foo may refer to the contents of memory location 10200

- Binding

- An association between two things
 - Example: A name and the thing it names

- Binding Time

- When a binding decision is made
 - Examples: Is the value of foo in a register or in memory? (*Decision made at compile time.*)



Today

- **Scopes**
 - Textual region of a program in which a binding is active
- **Object Lifetimes**
 - The period of time during which a name is bound to an object (variables, subroutines, etc.)
- **Storage Management**
 - How do we assign objects to storage (memory)?
 - How do we reclaim the storage when an object can no longer named?



Scopes

- Textual region of a program in which a binding is active
- Two varieties:
 - Static scope
 - Can determine at compile time exactly which names refer to which objects at which points in the program
 - Dynamic scope
 - Bindings from names to objects depend on the flow of execution of the program



Scopes

- Elaboration

- The process by which a set of bindings become active when control enters a scope
 - For example, allocation of storage to hold the objects

- Referencing Environment

- The set of active bindings at a given point in a program's execution
- Determined by **scope rules** for the programming language



Static Scope

- Bindings between names and objects can be determined at compile time
- Simple
 - Early versions of BASIC had a single, global scope
- Complex
 - Nested subroutines and modules allowed by modern programming languages require more complicated scope rules

Nested sub-routines

- Question:
 - In the body of function F1, which object does X refer to?
- Closest nested scope rule
 - Reference to variable refers to the object in the closest nested scope

```
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
```

A Problem Not Addressed by Nested Sub-routines

- Information hiding for complex abstract data types
- For simple ADTs, functions with static locals might work

```
/*  
  Place into *s a new name beginning with the letter l and continuing  
  with the ascii representation of an integer guaranteed to be distinct  
  in each separate call. s is assumed to point to space large enough  
  to hold any such name; for the short ints used here, seven characters  
  suffice. l is assumed to be an upper or lower-case letter.  
  sprintf 'prints' formatted output to a string.  
*/  
void gen_new_name (char *s, char l) {  
  static short int name_nums[52];  
  /* C guarantees that static local variables are initialized  
  to zeros */  
  int index = (l >= 'a' && l <= 'z') ? l-'a' : 26 + l-'A';  
  name_nums[index]++;  
  sprintf (s, "%c%d\0", l, name_nums[index]);  
}
```

- The variable `name_nums` retains its value across calls to `gen_new_name`
- Too simple for ADTs with multiple functions which need to share global state



Modules

- A module allows a collection of objects to be encapsulate so that
 - Objects inside the module are visible to one another
 - Objects on the inside are not visible on the outside unless explicitly exported
 - Objects outside are not visible on the inside unless explicitly imported

A module example

- A Stack abstraction in Modula
- We export push and pop functions
- Not visible outside of module
 - Top of stack pointer "top"
 - Stack array "s"

```
CONST stack_size = ...
TYPE element = ...
...
MODULE stack;
IMPORT element, stack_size;
EXPORT push, pop;
TYPE
    stack_index = [1..stack_size];
VAR
    s    : ARRAY stack_index OF element;
    top  : stack_index;      (* first unused slot *)


PROCEDURE error; ...

PROCEDURE push (elem : element);
BEGIN
    IF top = stack_size THEN
        error;
    ELSE
        s[top] := elem;
        top := top + 1;
    END;
END push;

PROCEDURE pop () : element; (* A Modula-2 function is just a *)
                          (* procedure with a return type. *)
BEGIN
    IF top = 1 THEN
        error;
    ELSE
        top := top - 1;
        RETURN s[top];
    END;
END pop;

BEGIN
    top := 1;
END stack;
```

```
VAR x, y : element;
...
push (x);
...
y := pop;
```



More module scope terminology

- Closed scopes
 - Modules into which names must be explicitly imported in order to be visible within the module
- Open scopes
 - Modules which do not require explicit imports in order for outside names to be visible



Dynamic Scope

- Bindings between names and object depend on the flow of control at run-time
 - Order in which sub-routines are called is important
- Dynamic scope rules are usually simple
 - The current binding between name and object is the one most recently encountered during execution



Static versus Dynamic Scope

- Static scoping
 - Program prints 1
- Dynamic scoping
 - Program prints 1 or 2 depending on value read at line 8
- Why?
 - Is the assignment to "a" at line 3 an assignment to the global "a" from line 1 or the local "a" from line 5

```
1: a : integer    -- global declaration
2: procedure first
3:   a := 1
4: procedure second
5:   a : integer  -- local declaration
6:   first ()
7:   a := 2
8:   if read_integer () > 0
9:     second ()
10:  else
11:    first ()
12: write_integer (a)
```



Dynamic Scoping

- What are the advantages?
 - Makes it easy to customize sub-routines
 - Example:

```
begin --nested block
  print_base: integer := 16
  print_integer(n)
```

 - The variable `print_base` controls the base which `print_integer` uses to print numbers
 - `print_integer` can be set early in a global scope to a default value, and temporarily overridden in a global scope to some other base



Dynamic Scoping

- Problem: unpredictable referencing environments

```
max_score : integer    -- maximum possible score

function scaled_score (raw_score : integer) : real
  return raw_score / max_score * 100
...
procedure foo
  max_score : real := 0    -- highest percentage seen so far
  ...
  foreach student in class
    student.percent := scaled_score (student.points)
    if student.percent > max_score
      max_score := student.percent
```

- Global variable `max_score` is used by `scaled_score()`
- `max_score` is redefined in `foo()`
- `scaled_score()` is called by `foo()`
- Whoops!



Mixed Scoping

- Perl supports both static and dynamic scoping

Dynamic Scoping

```
$i = 1;
sub f {
    local($i) = 2;
    return g();
}
sub g { return $i; }
print g(), f();
```

Output:

1 2

Static Scoping

```
$i = 1;
sub f {
    my($i) = 2;
    return g();
}
sub g { return $i; }
print g(), f();
```

Output:

1 1



Scoping Summary

- Static scoping
 - Used by mostly in compiled, high-level languages
 - C, C++, Java, Pascal, ...
 - Name to variable bindings can be determined at compile time
 - Efficient
- Dynamic scoping
 - Used by many interpreted languages
 - Original LISP, APL, Snobol, and Perl
 - Name to variable bindings may need to be determined at run-time
 - Flexible

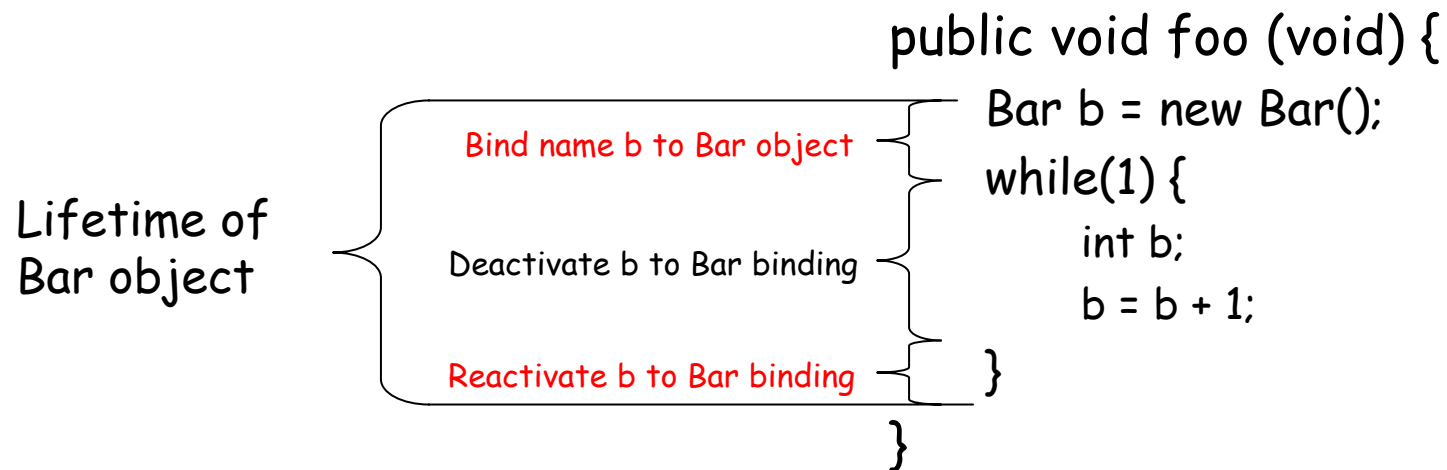


Object Lifetimes

- Key events in object lifetime
 - Object is created
 - Bindings to object are created
 - References to variables, subroutines, types made using bindings
 - Deactivation and reactivation of temporarily unusable bindings
 - Destruction of bindings
 - Destruction of object
- Binding lifetime: time between creation and destruction of a binding
 - Example: Time during which a Java reference refers to an object in memory
- Object lifetime: time between creation and destruction of an object
 - Example: Time during which an object "lives" in memory

Object Lifetimes

■ Example:



Note: The Bar object created by foo() isn't necessarily destroyed upon sub-routine return. It can no longer be referenced after foo() returns, but it will likely be destroyed during garbage collection at some later point in the program's execution.



Object Lifetimes

- Object lifetimes correspond to one of three principal storage allocation mechanisms
 - Static allocation
 - Objects are allocated at compile- or run-time to fixed storage locations
 - Example: global variables
 - Stack allocation
 - Objects are allocated at run-time as needed in a last-in, first-out order
 - Example: local variables
 - Heap allocation
 - Objects are allocated and de-allocated at run-time in an arbitrary order
 - Example: dynamically allocated objects

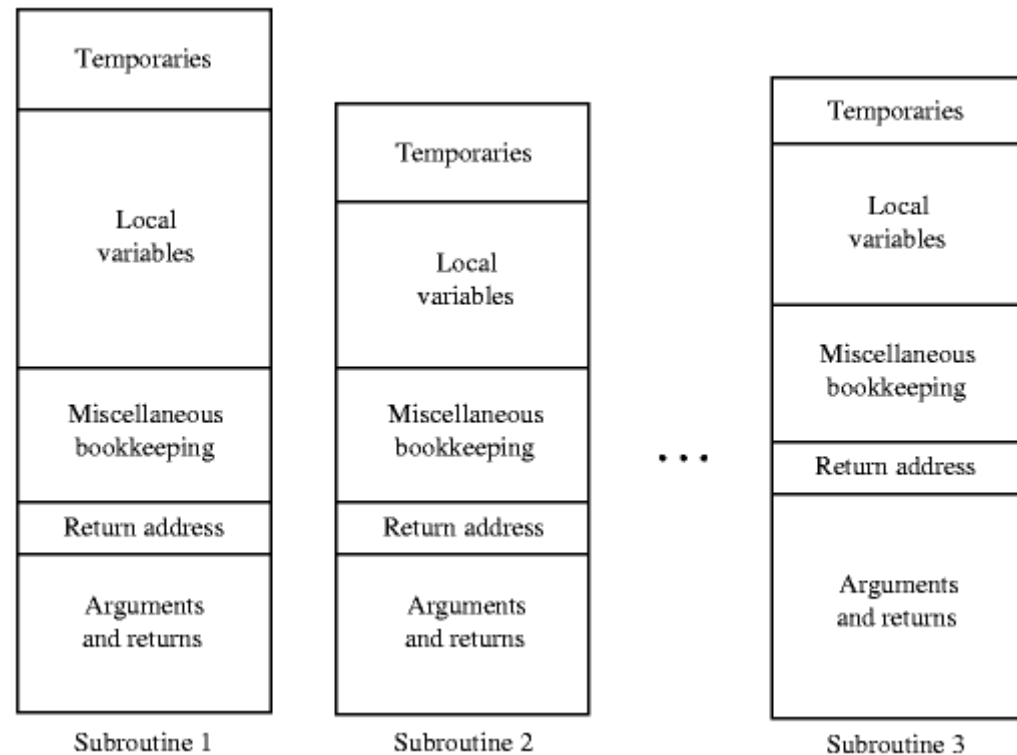


Static Allocation

- What gets allocated statically?
 - Global variables
 - The program's code (subroutines, etc.)
 - Exception: dynamically linked subroutines
 - Debugging information produced by the compiler
 - Constant values which should not change during program execution
 - Example: "i=%d\n" is constant in `printf("i=%d\n",i);`
 - When language doesn't support recursive subroutine calls then local variables, subroutine arguments, return values, compiler generated temporaries, etc.

Static allocation of locals

- Example:
 - No recursion means that there can only be one instance (**activation**) of any given subroutine active at a time
 - Therefore we can statically reserve storage for the single possible activation of each subroutine in the program





Stack Allocation

- When languages allow recursion, each subroutine can have many simultaneous activations

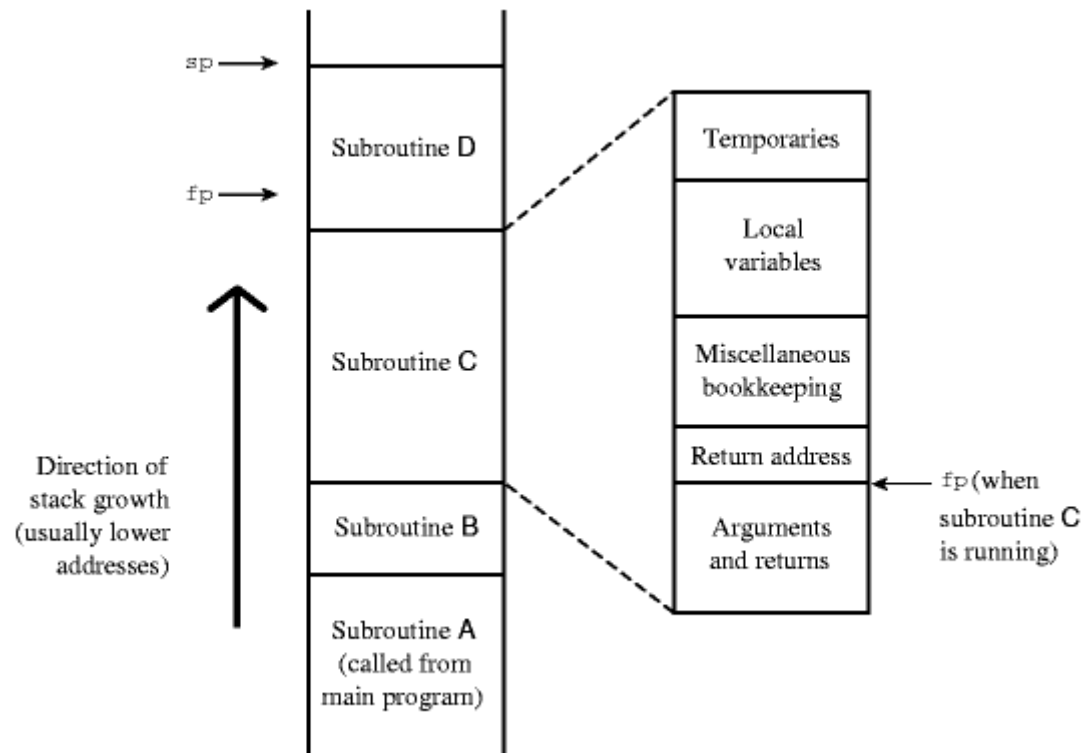
- Example:

- ```
public int foo (int n) {
 int a, b, c;
 a = random(); b = random();
 if (n > 0) { c = foo(n-1); }
 c = c * (a + b);
}
```

*We may have  $n$  activations of `foo()`, and each needs a place to store its own copy of `a`, `b`, `c`, input argument `n`, and any temporary values the compiler might have generated.*

# Stack Allocation

- Solution to the recursion problem: **stack allocation**
- "Push" stack to reserve storage for locals on subroutine calls
- "Pop" stack to de-allocate storage for locals when subroutine returns





# Heap Allocation

---

- We know how to allocate code, globals, constants, locals, temporaries, ...
- What's left:
  - Dynamically allocated objects
    - Example: objects created by Java new() operator
  - Why can't they be statically allocated?
    - Created dynamically...
  - Why can't they be allocated on the stack?
    - An object created dynamically by a subroutine may outlive the subroutine's activation
      - Example: object is assigned to a global or returned from the sub-routine

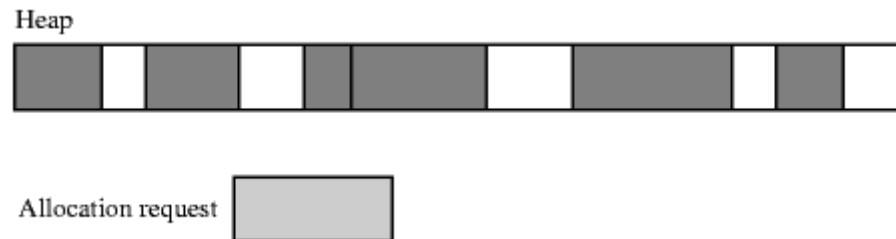


# Heap Allocation

---

- Heaps solve this problem
  - A heap is a region of storage in which blocks of memory can be allocated and de-allocated at arbitrary times
- How are heaps implemented?
  - How do we handle allocation and de-allocation requests?

# Heap Management



- We get an allocation request for  $n$  bytes of memory
- Heap has used (dark grey) and free (white) regions
- How do we pick a free region to satisfy the allocation request?
- Simple Answer:
  - Find the first free region which is big enough to meet the allocation request (called **first fit**)
  - Problem: internal fragmentation
    - If we requested  $n$  bytes, and the first available free region has  $n+k$  bytes then we waste  $k$  bytes by allocating that first region

# Heap Management



- A better answer (perhaps):
  - Find the first free region which is closest in size to the allocation request (called **best fit**)
  - Problem: more time consuming
    - Must search all free blocks to find the best fit
    - There may still be internal fragmentation, but hopefully less than first fit

# Heap Management



- Another problem: external fragmentation
  - We get an allocation request for  $n$  bytes and the heap has more than  $n$  bytes free, but
    - No single free region is  $n$  bytes or larger
  - We have enough free memory, but we can't satisfy the allocation request!
- Possible solution:
  - Region coalescing: when two adjacent regions are free with  $j$  and  $k$  free bytes respectively, merge them into a single  $j+k$  byte free region
  - An improvement, but can't eliminate all external fragmentation

# Heap Management



- How do we handle de-allocation?
  - Explicit:
    - Programmer must tell the heap manager that a region is no longer used by the program
      - Example: in C, use `free(p)` to free the region pointed to by p
  - Automatic:
    - Run-time system determines which object on the heap are live (are still bound to names) or dead (no longer bound to any name) and automatically de-allocates the dead objects
    - Called **garbage collection**



# Garbage Collection

---

- Why use garbage collection?
  - Prevents:
    - **Memory leaks**: programmers may forget to de-allocate dynamically allocated memory
    - **Dangling pointers**: programmers may de-allocate an object before destroying references to it
  - Reduces programming effort and results in more reliable programs



# Garbage Collection

---

- Why not to use garbage collection?
  - **Expensive**: determining which objects are live and which are dead takes time
  - **Bad for real-time systems**: usually can't guarantee when garbage collector will run and how long it will take
  - **Difficult to implement**: writing a garbage collector is hard work and makes the compiler and the language run-time more complicated
  - **Language design**: some languages weren't designed with garbage collection in mind thus making it difficult to add a garbage collector

# Advanced Language Features and Binding



---

- How do we implement static scopes in languages that allow nested sub-routines?
- How do we implement references to subroutines?



# Implementing Static Scopes

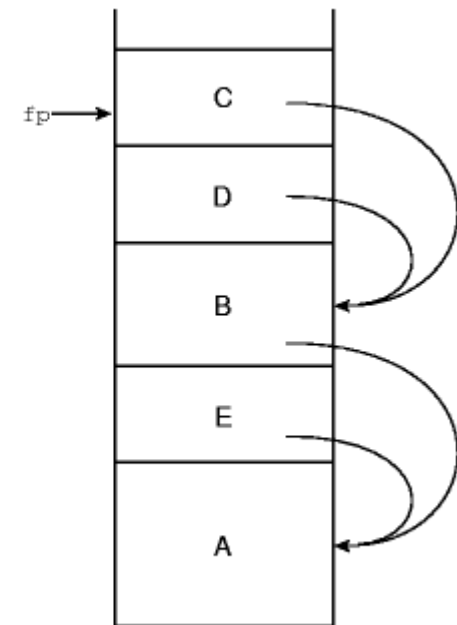
---

- Problem:
  - From c(), how do we reference non-local variables in a() and b()?

```
void a (void) {
 int foo1;
 void b (void) {
 int foo2;
 void c (void) {
 int foo3 = foo1 + foo2;
 }
 void d (void) {
 }
 }
 void e (void) {
 }
}
```

# Implementing Static Scopes

- Solution: static links
  - Each activation of a subroutine stores a pointer in its stack frame to the next enclosing scope
  - `c()` gets its locals from its own stack frame
  - `c()` gets non-locals in `b()` by dereferencing its static link once
  - `c()` gets non-locals in `a()` by dereferencing its static link to `b()` and then `b()`'s static link to `a()`





# Implementing subroutine references

---

- Big question:
  - How do we apply scope rules in languages where subroutines can be passed as values?
- Two answers:
  - Shallow binding:
    - Referencing environment is determined immediately before the referenced subroutine is called
    - Usually default in languages with dynamic scoping
  - Deep binding:
    - Referencing environment is determined when the reference to the subroutine is created

# Deep versus shallow binding

- Shallow binding
  - Needed in order for **line\_length** assignment in **print\_selected\_records** to reach the **print\_person** subroutine
- Deep binding
  - Needed in order for **threshold** assignment in the main program to reach **older\_than** subroutine

```
type person = record
 ...
 age : integer
 ...
threshold : integer
people : database

function older_than (p : person) : boolean
 return p.age ≥ threshold

procedure print_person (p : person)
 -- Call appropriate I/O routines to print record on standard
 -- output. Make use of non-local variable line_length to format
 -- data in columns.
 ...

procedure print_selected_records (
 db : database
 predicate, print_routine : procedure)
 line_length : integer

 if device_type (stdout) = terminal
 line_length := 80
 else
 -- Standard output is a file or printer.
 line_length := 132
 foreach record r in db
 -- Iterating over these may actually be
 -- a lot more complicated than a 'for' loop.
 if predicate (r)
 print_routine (r)

-- main program
...
threshold := 35
print_selected_records (people, older_than, print_person)
```



# Implementing sub-routine references

---

- Closures
  - A pointer to subroutine code and a pointer to the subroutine's referencing environment
- Why do we need a pointer to the referencing environment?
  - Must have some way for the sub-routine to access it's non-local, non-global variables
    - For dynamically scoped languages these include variables in the enclosing dynamically nested subroutines (routines that call other routines)
    - For statically scoped languages these include variables in enclosing statically nested routines



# Summary

---

- Scopes
  - Static versus Dynamic
- Object lifetimes
- Storage management
  - Static allocations, stacks, and heaps
- Advanced language features and bindings
  - Implementing static scopes for nested subroutines
  - Implementing subroutine references