Programming Languages

Subprograms

CSCI-GA.2110-001
Fall 2013

Subprograms

- the basic abstraction mechanism
  - promotes code reuse
  - increases readability & maintainability
- two kinds: *functions* vs. *procedures*.
- functions correspond to the mathematical notion of computation:

$$\text{input} \longrightarrow \text{output}$$

- procedures affect the environment, and are called for their side-effects
- side-effects refer to a change in program state beyond the scope of the procedure.
- pure functional model possible but rare (Haskell, Clean)
- hybrid model most common: functions can have side effects

Environment of the computation

- declarations introduce names that denote entities
- at execution-time, entities are bound to values or to locations:

  name $\longrightarrow$ value *functional*

  name $\longrightarrow$ location $\longrightarrow$ value *imperative*
- exceptions exist:

  C++ e.g., `#define NINE 9`
- value binding takes place during function invocation
- names are bound to locations on scope entry
- locations are bound to values by assignment

Parameter passing

The rules that describe the binding of arguments to formal parameters, i.e., the meaning of a reference to a formal in the execution of the subprogram.

```
function f (a, b, c) ... // parameters: a, b, c

f(i, 2/i, g(i,j));       // arguments: i, 2/i, g(i,j)
```

- *by value*: formal is bound to value of actual
- *by reference*: formal is bound to location of actual
- *by copy-return*: formal is bound to value of actual; upon return from routine, actual gets copy of formal
- *by name*: formal is bound to expression for actual; expression evaluated whenever needed; writes to parameter are allowed (and can affect other parameters!)
- *by need*: formal is bound to expression for actual; expression evaluated the first time its value is needed; cannot write to parameters

Performance considerations

- **by value**: the value of the actual is copied to the stack frame.
  - Copying can be expensive for large objects.
  - Once copied, modification/access is same as a local variable.
- **by copy-return**: similar to "value" except parameter copy happens twice.
- **by reference**: the address of the actual is copied to the stack frame.
  - Copying is fast, since size of a memory address is small.
  - Modification/access requires 2 levels of indirection: all accesses must be preceded by a dereference.
- **by name**:
  - Evaluations performed every time a formal parameter is referenced.
  - Performance depends on the expression. (e.g., function expressions cause a function invocation every time.)
- **by need**:
  - Performance is similar to "value": evaluation is only performed once.

Parameter passing in Ada

- goal: separate semantic intent from implementation
- parameter modes:
  - `in` : read-only in subprogram (default)
  - `out` : write in subprogram
  - `in out` : read-write in subprogram
- independent of whether binding by value, by reference, or by copy-return
- functions can only have `in` parameters

Syntactic sugar

- Default values for in-parameters (Ada)

```
function Incr (Base: Integer;
               Inc: Integer := 1) return Integer;
```

- `Incr(A(J))` equivalent to `Incr(A(J), 1)`
- also available in C++

```
int f (int first,
       int second = 0,
       char *handle = 0);
```

- named associations (Ada):

```
Incr(Inc => 17, Base => A(I));
```

Parameter passing in C

- C: parameter passing by value, no semantic checks. Assignment to formal is assignment to local copy
- if argument is pointer, effect is similar to passing designated object by reference

```
void incr (int *x) {
  (*x)++;
}
incr(&counter); /* pointer to counter */
```

- no need to distinguish between functions and procedures: `void` return type indicates side-effects only

Parameter-passing in C++

- default is by-value (same semantics as C)
- explicit reference parameters:

```
void incr (int& y) {
  y++;
}
incr(counter);  // compiler knows profile of incr,
                //   builds reference
```

- semantic intent indicated by qualifier:

```
void f (const double& val);  // passed by reference,
                             // cannot be unbound
```

Parameter-passing in Java

- by value only
- semantics of assignment differs for primitive types and for classes:
  - primitive types have value semantics
  - objects have reference semantics
- consequence: methods can modify objects
- for formals of primitive types: assignment allowed, affects local copy
- for objects: `final` means that formal is read-only

Block structure

```
procedure Outer (X: Integer) is
  Y: Boolean;

   procedure Inner (Z: Integer) is
     X: Float := 3.0; -- hides outer x

     function Innermost (V: Integer) return Float is
     begin
       return X * Float(V * Outer.X);   -- use Inner.X
                                       -- and Outer.X
     end Innermost;

   begin
     X := Innermost(Z); -- assign to Inner.X
   end Inner;

begin
  Inner(X); -- Outer.X, the other one is out of scope
end;
```

Parameter passing anomalies

```
program example;
  var
    global: integer := 10;
    another: integer := 2;
  procedure confuse (var first, second: integer);
  begin
    first := first + global;
    second := first * global;
  end;
begin
  confuse(global, another);  /* first and global */
                             /*  are aliased     */
end
```

- different results if by reference or by copy-return
- semantics should not depend on implementation of parameter passing
- passing by value with copy-return is less error-prone

Storage outside of the block

- with block structure, the lifetime of an entity usually coincides with the invocation of the enclosing subprogram
- if the same entity is to be used for several invocations, it must be global to the construct
  - in C,C++, can be declared `static` instead
- simplest: declare in the outermost context
- three storage classes:
  - static
  - stack-based (automatic)
  - heap-allocated

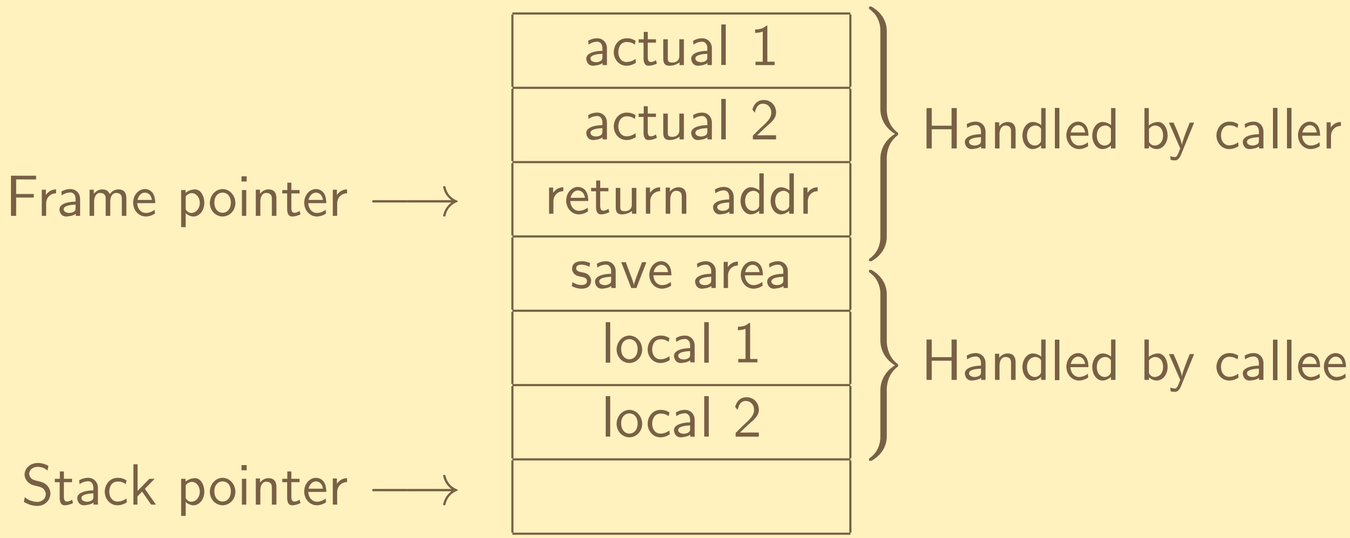
Bounded Nesting

- C, C++, Java:
  - no nested functions
  - blocks are merged with activation record of enclosing function
  - static storage available
- Pascal, Ada:
  - arbitrary nesting of packages and subprograms
  - packages provide static storage

Run-time organization

- each subprogram invocation creates an activation record
- recursion imposes stack allocation
- activation record hold actuals, linkage information, saved registers, local entities
- caller: place actuals on stack, return address, linkage information, then transfer control to callee
- prologue: save registers, allocate space for locals
- epilogue: place return value in register or stack position, update actuals, restore registers, then transfer control to caller
- binding of locations: actuals and locals are at fixed offsets from frame pointers
- complications: variable # of actuals, dynamic objects

Activation record layout



Variable number of parameters

```
printf("this␣is␣%d␣a␣format␣%d␣string", x, y);
```

- within body of `printf`, need to locate as many actuals as placeholders in the format string
- solution: place parameters on stack in *reverse* order
  (actuals at positive offset from FP, locals at negative offset from FP)

actual n
actual n-1
...
actual 1 (format string)
return address

Calling conventions

In practice, activation records may differ from the traditional layout:

- Order of parameters
- Bookkeeping information
- Passing parameters and other information via hardware registers
- For space reasons, by-value parameters may not be copied
- Inlining: no activation record at all

The responsibility for performing certain actions may differ.

- Should the caller or callee reclaim the stack space?
- Who should save the register values?

Subprogram callers and callees must *completely agree* on who does what, how, and when. These details are encompassed in a protocol known as the *calling convention*.

Calling conventions

Why do we need to care about calling conventions?

- When code calls out to a library, it must follow the calling conventions of the library.
- A mixture of calling conventions may exist.
- The compiler usually worries about the calling conventions. But...
- Programmers can specify the calling convention to gain time/space advantages.

The most popular conventions:

- C (`cdecl`): Parameters placed on the stack in right-to-left order. Caller required to clear the stack parameters.
- Microsoft Standard (`stdcall`): called function required to clear the stack parameters.
- Fast Call (`fastcall`): up to two parameters placed in hardware registers. Rest on the stack.
- Microsoft C++ (`thiscall`): the “this” pointer passed through the CX register (x86)

Objects of dynamic size

```
declare
  X: String(1..N); -- N global, non-constant
  Y: String(1..N);
begin ...
```

Where is the start of `Y` in the activation record?

- **Solution 1**: use indirection: activation record holds pointers
  simpler implementation, costly dynamic allocation/deallocation
- **Solution 2**: local indirection: activation record holds offset into stack
  faster allocation/deallocation, complex implementation

Run-time access to globals

```
procedure Outer is       -- recursive
  Gbl: Integer;
  procedure Inner is     -- recursive
    Loc: Integer;
  begin
    ...
    if Gbl = Loc then    -- how do we locate Gbl?
    ...
  end;
begin
  ...
end;
```

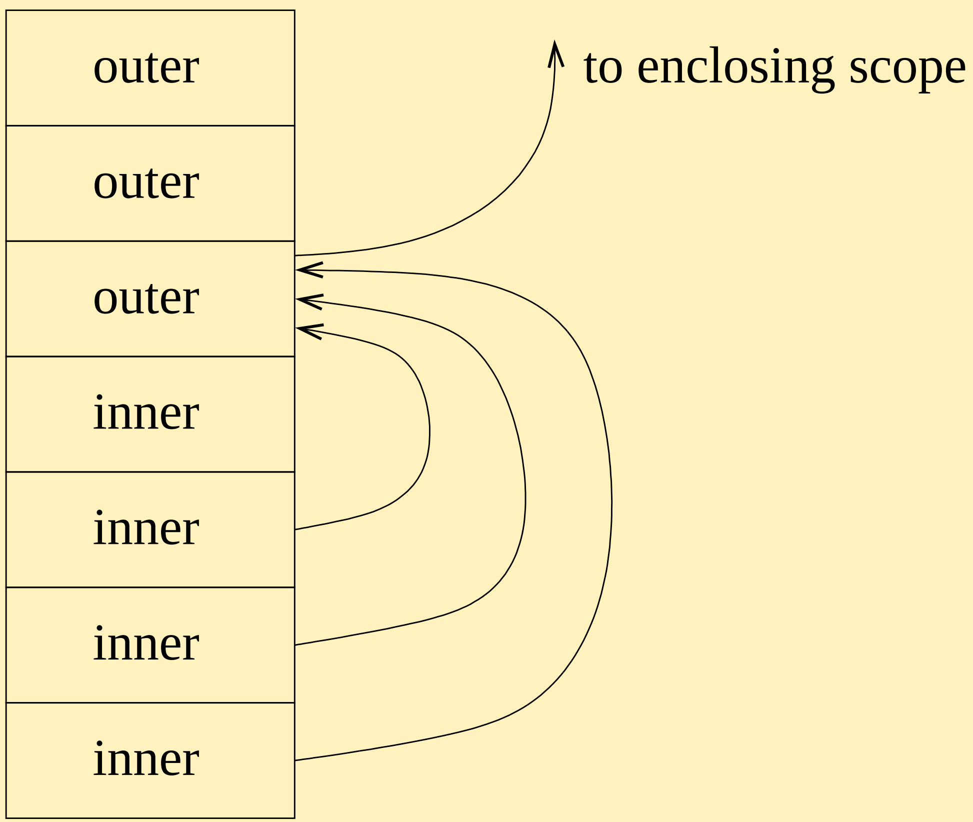
- Need run-time structure to locate activation record of statically enclosing scopes.
- Environment includes current activation record and activation records of parent scopes.

Global linkage

- *static chain*: pointer to activation record of statically enclosing scope
- *display*: array of pointers to activation records
- does not work for function values
  - functional languages allocate activation records on heap
- may not work for pointers to functions
  - simpler if there is no nesting (C, C++, Java)
  - can check static legality in many cases (Ada)

Static Links

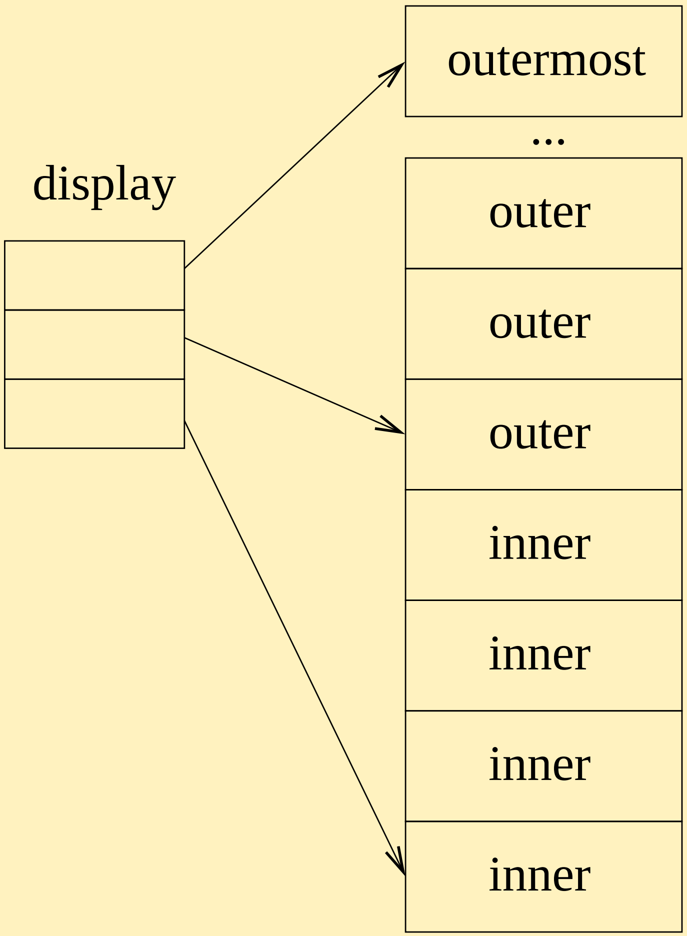
Activation record holds pointer to activation record of enclosing scope.
Set up as part of call prologue.



To retrieve entity n scopes out, need n dereference operations.

Display

Global array of pointers to current activation records



$O(1)$ display lookup: one entry per scoping level (known at compile time), plus dereference.

Returning composite values

- intermediate problem: functions that return values of non-static sizes:

```
function Conc3 (X, Y, Z: String) return String is
begin
  return X & ":" & Y & ":" & Z;
end;

Str := Conc3(This, That, The_Other);
```

- best not to use heap, but still need indirection
- simple solutions: forbid it (Pascal, C) or use heap automatically (Java)

Subprogram parameters in Ada

```
procedure Outer (...) is
  type Proc is access procedure (X: Integer);
  procedure Perform (Helper: Proc) is begin
    Helper(42);
  end;
  procedure Action (X: Integer) is ...
  procedure Proxy is begin
    Perform(Action'access);
  end;
begin
  ...
end;
```

`Action'access` creates pair: (ptr to `Action`, env of `Action`). Known as a *closure*.

How does `Proxy` know what `Action`'s environment is?

Simplest implementation of environment is a pointer (static link); can be display instead.

The limits of stack allocation

```
type Ptr is access function (X: Integer) return Integer;

function Make_Incr (X: Integer) return Ptr is
  function Incr (Base: Integer) return Integer is
  begin
    return Base + X;  -- reference to formal of Make_Incr
  end;
begin
  return Incr'access;  -- will it work?
end;

Add_Five: Ptr := Make_Incr(5);

Total: Integer := Add_Five(10);  -- where does Add_Five
                                 --  find X ?
```

First-class functions

Allowing functions as first-class values forces heap allocation of activation records.

- environment of function definition must be preserved until the point of call: activation record cannot be reclaimed if it creates functions
- functional languages require more complex run-time management
- higher-order functions: functions that take (other) functions as arguments and/or return functions
  - powerful
  - complex to implement efficiently
  - imperative languages restrict their use
  - (a function that takes/returns pointers to functions can be considered a higher-order function)

Higher-order functions

Both arguments and result can be (pointers to) subprograms:

```
type Func is access function (X: Integer) return Integer;
function Compose (First, Second: Func) return Func is
declare
  function Result (X: Integer) return Integer is
  begin
    return Second(First(X));   -- implicit dereference
                              --   on call
  end;
begin
  return Result'Access;
end;
```

This is illegal in Ada, because `First` and `Second` won't exist at point of call.

Restricting higher-order functions

- C: no nested definitions, so environment is always global
- C++: ditto, except for nested classes
- Ada: static checks to reject possible dangling references
- Modula: pointer to function illegal if function not declared at top-level
- ML, Haskell: no restrictions – `compose` is easily definable:

```
fun compose f g x = f (g x)
```