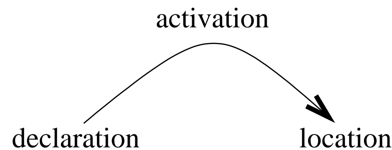
COMP 356
Programming Language Structures
Notes for Chapter 10 of *Concepts of Programming Languages*
Implementing Subprograms

1 Activation Records



Recall that an activation is an execution of a subprogram. In most languages, local variables are allocated when this execution begins (**activation time**).

The storage (for formals, local variables, function results etc.) needed for an activation is organized as an **activation record** (or **frame**). In a language with recursion, each simultaneous activation of a recursive subprogram can have different parameters, different values for local variables, return a different result ...

- each activation needs its own activation record
- the number of activation records needed isn't known at compile time
- allocation of activation records is dynamic, and local variables are stack dynamic (unless declared static)

In older versions of FORTRAN (no recursion), allocation of activation records was static, so local variables were static.

Typical activation record layout:

dynamic link
static link
saved state
parameters
function result
local variables
temp storage

where:

- the **dynamic link** (also called the **control link**) points to the activation record of the caller, and is used in subprogram return
- the **static link** (also called the **access link**) points to the activation record associated with the nearest enclosing scope of the subprogram definition (used for implementing static scope in languages where subprogram definitions can be nested)
- the saved state typically includes the program counter and register contents at the time the subprogram was called, and is used for restoring the context of the caller on subprogram return
- the temporary storage is used for evaluating expressions, storing addresses for pass-by-value-result, ...

Allocation of activation records can be:

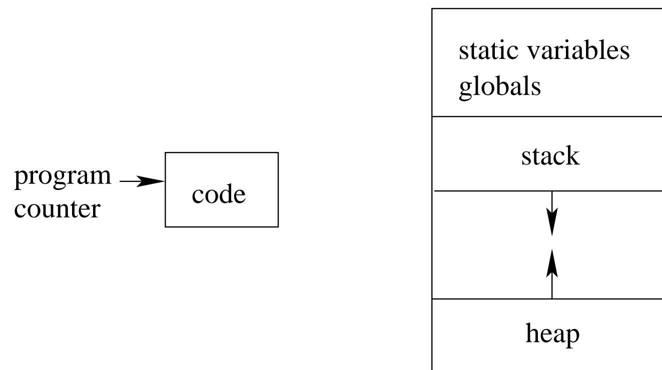
- on the heap
  - more flexible
  - used in Modula-3, LISP, Scheme, ...
- on the stack
  - more efficient
  - used in C, C++, Java, C#, Pascal, Ada, ...

2 Functions in C

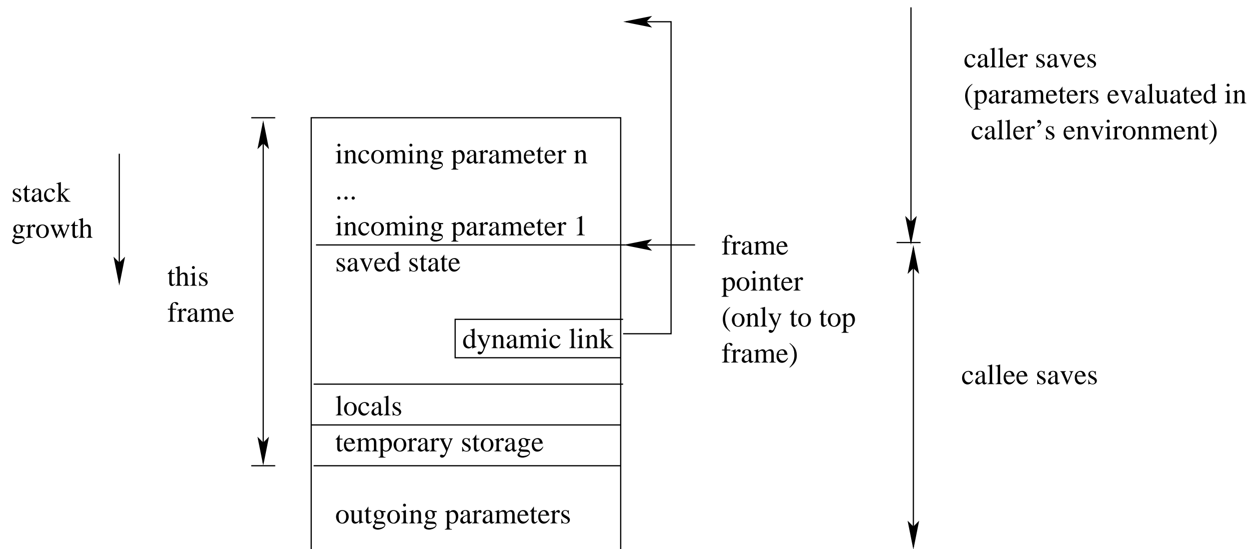
In C:

- the lifetime of local variables is contained within one activation (except for static variables)
- locals can be allocated at activation time and deallocated when the activation ends
- activation records can be allocated on a stack

A programming language obeys the **stack discipline** if the lifetime of locals is within one activation.
Memory layout for C programs:



Activation records in C:



Notes:

- locals and formals are accessed relative to the frame pointer because frames (activation records) vary in size
- the size of the incoming parameters varies for unchecked functions
- no static links are required (the only scope to nest function definitions in is the global scope)

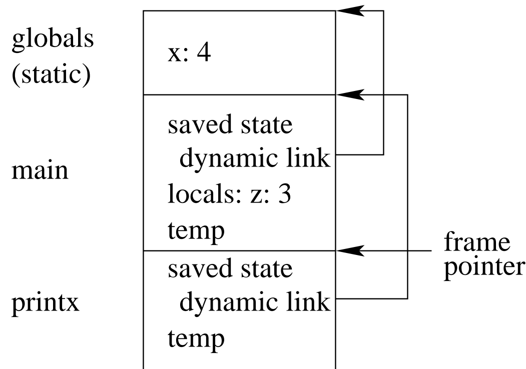
Function call sequence:

1. the caller evaluates the actual parameters and places them in the incoming parameters for the callee
2. the callee saves state information for the caller (registers, program counter, ...)
3. the callee allocates local variables and temporary storage
4. the body of the callee is executed, possibly leading to more activations
5. control returns to the caller
    - the dynamic link is used to restore the frame pointer
    - the caller's state is restored using the saved state
    - the stack is popped

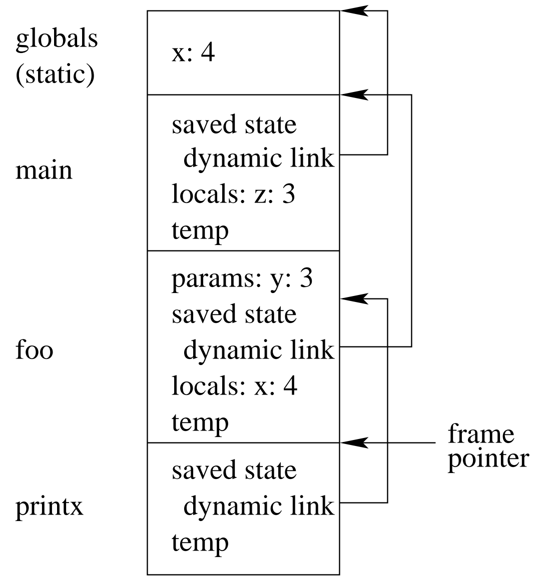
Consider the following program:

```
#include <stdio.h>

int x = 4;

void printx(void) {printf("%d\n", x);}

void foo(int y) {
  int x = 4;

  x = x + x * y;
  printx();
}

void main() {
  int z = 3;

  printx();
  foo(z);
}
```

The following diagram shows the runtime stack during the first and second activations of `printx()`:



runtime stack during the
first activation of printx()



runtime stack during the
second activation of printx()

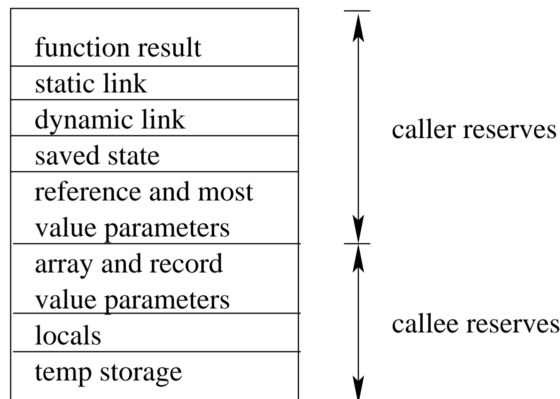
Blocks could be implemented by using an activation record for each block – but static links would then be required. Most C compilers do not use this approach. Instead, for each function they:

- calculate the maximum amount of storage required for variables local to nested blocks. This can be done because blocks are entered and exited in textual order.
- this storage is laid out after the local variables in the activation record
- offsets (from the frame pointer) for all variables local to nested blocks are computed statically. This allows such variables to be addressed in exactly the same way as other local variables, and automatically allows sharing of storage for blocks that don't overlap.

3 Subprograms in Pascal

In some languages (Pascal, Ada, JavaScript, Python, ...), subprogram definitions can be nested. Hence, Pascal activation records require static links to implement static scope (to find the declarations of nonlocal variables).

Activation records in Pascal:

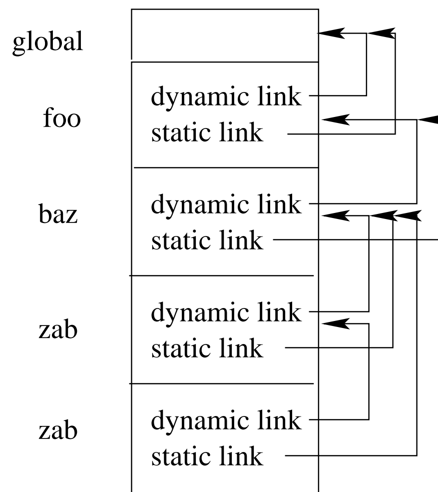


Consider the following program:

```
var b: integer;

procedure foo(var x: integer);
  var x1: integer;
  procedure baz(y: integer);
    procedure zab(z: integer);
      begin
        if z = 1 then zab(0)
        else x := y + z;
      end; (* zab *)
    begin (* baz *)
      zab(1);
    end; (* baz *)
  begin (* foo *)
    x1 := 3;
    baz(x1);
  end; (* foo *)

begin (* main *)
  foo(b);
end.
```

When procedure `zab()` is called, it must be able to find the storage for variable `x`, which is in the activation record for procedure `foo()`. `zab()` will do so by following static links to find the nearest enclosing scope with a declaration of `x` (i.e. `foo()`'s activation record). The sequence of static links connecting two activation records is called a **static chain**.

The following diagram shows the runtime stack during the second activation of `zab()`:



Notes:

- static links are required because the distance to the activation record representing the nearest enclosing scope can not be determined at compile time
- an activation record representing the nearest enclosing scope is guaranteed to be on the stack because of the visibility rules of Pascal (i.e. the only procedures that can call `zab()` are `zab()` and `baz()`, so an activation record for `baz()` must be on the stack whenever `zab()` is called
- the number of static links to follow to find the declaration of a nonlocal can be computed at compile time from the **nesting depth** of the procedure

For example, `zab()` is at nesting depth 2 with respect to `foo()` (where variable `x` is declared), so to find the declaration of `x`, `zab()` will need to follow 2 static links.

The address of a nonlocal variable can always be computed at compile time as:

- a number of static links to follow
- an offset in the activation record thus reached

When a subprogram calls a **local subprogram** (defined in its own body), the static link of the callee should point to the caller. For example, this is the situation when `baz()` calls `zab()`. Note that this is the only case where a subprogram can call a more deeply nested subprogram.

When a subprogram calls a nonlocal subprogram, the static link of the callee is set as follows:

1. find the absolute value of the difference in nesting depth between the two subprograms w.r.t. the program (call it d)
2. follow d + 1 static links from the caller
3. set the static link of the callee to point to the activation record found in the previous step

For example, suppose that `zab()` calls `baz()`. The difference in their nesting depths is 1, and following 2 static links from `zab()` leads to `foo()`'s activation record.

As another example, suppose that a subprogram at nesting depth 1 calls another subprogram at nesting depth 1 (the only case in C). The difference in nesting depths is 0, and following 1 static link from the caller leads to the global scope.

Displays:

- are an optimization technique that replaces static links
- are implemented as an array of pointers to activation records
- the size of the array is the maximum nesting depth in the program
- use: if `disp` is the display, then `disp[i]` points to the activation record associated with the current scope at nesting depth `i`

To access a nonlocal declared at nesting depth `i`:

1. go to the activation record pointed to by `disp[i]`
2. find the nonlocal via an offset in this activation record

The advantage of displays is that they provide constant time access to nonlocals - exactly 2 steps are required regardless of the difference in nesting depths between the active subprogram and the subprogram where the variable is declared. The disadvantage is that they are expensive to maintain.

One simple implementation of display maintenance:

- use a global display for the current activation
- each caller saves the display in its activation record, and restores the display when the callee returns
- when a call occurs, the new display is constructed using techniques similar to those used for static links

When a subprogram (call it `foo()`) is passed as a parameter to another subprogram (call it `bar()`), the static link or display for `foo()` is constructed and passed along with it. The static link or display is constructed exactly as if `foo()` were being called (rather than passed as a parameter). When `bar()` calls the subprogram that it was passed:

- if the runtime system uses static links, the static link that was passed is used to initialize the static link of the new activation record
- if the runtime system uses displays, the display that was passed is used as the global display

This ensures that the subprogram parameter is always executed in the environment it was defined it, i.e. using static scope.

4 Other Aspects of Implementing Subprograms

4.1 Implementing Dynamic Scope

Under dynamic scope, an occurrence of a variable name is bound to the most recently seen declaration of that variable at runtime. The simplest implementation is to search the runtime stack for declarations of each nonlocal, using dynamic links in much the same way that static links are used to implement static scope. The main difference is that the number of dynamic links to follow can not be computed at compile time. This technique is called **deep access** because searches deep into the stack may be required. Various optimization techniques (collectively called **shallow access** techniques) can be used to provide constant time access to nonlocals.

No special provisions are needed to pass a subprogram as a parameter in dynamically scoped languages. No static links or displays are used, so a call to a subprogram parameter is the same as a call to any other subprogram.

4.2 Tail Recursion

Recursion is often an elegant problem solving technique, but subprogram call and return is expensive, and the number of activation records needed for recursive subprograms can exhaust the runtime stack.

One approach to avoiding recursive subprogram calls:

- transform recursive subprograms to make them tail recursive
- transform tail recursive subprograms to iterative subprograms

Definitions:

- a recursive subprogram call is **tail recursive** if it is the last expression evaluated in the body of the subprogram
- a subprogram is **tail recursive** if all recursive calls it contains are tail recursive

Example:

```
int fact(int n) {
  if (n < 2) return 1;
  else return (n * fact(n - 1));
}
```

Function `fact()` is not tail recursive, because the return value from the recursive call must be multiplied by `n` before it can be returned. The partial expression `n *` is called the **context** of the recursive call.

A subprogram can often be made tail recursive by:

- adding an extra parameter to "accumulate" the return value
- applying the context of the original recursive call to the accumulator
- returning the value of the accumulator as the base case of the recursion
- adding an additional "interface" function that provides the initial value for the accumulator

For example:

```
int fact_tr(int n, int accum) {
  if (n < 2) return accum;
  else return fact_tr(n - 1, n * accum);
}

int fact(int n) { return fact_tr(n, 1); }
```

Note that `fact_tr` is tail recursive. In particular, each activation record for `fact_tr` just sits on the stack, waiting to return the value returned by the next activation of `fact_tr`.

To make `fact_tr` iterative, we replace each tail recursive call by:

- reassigning the parameters appropriately
- adding a goto to the first statement of the subprogram

```
int fact_iter(int n) {
  int accum = 1;
start: if (n < 2) return accum;
       else {
         // order matters for the next 2 statements
         accum = n * accum;
         n = n - 1;
         goto start;
       }
}
```

Notes:

- the translation from a tail recursive subprogram to an iterative one can be automated (hence the `goto` above)
- not all recursive subprograms can be made tail recursive
- all implementations of Scheme are required by the language standard to be **properly tail recursive**, which means that a tail recursive call causes the activation record of the callee to replace the activation record of the caller on the stack (or heap)