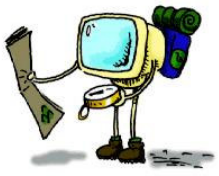
Programming Languages

Tevfik Koşar

Lecture - VIII
February 9th, 2006

Roadmap

- Allocation techniques
  - Static Allocation
  - Stack-based Allocation
  - Heap-based Allocation
- Scope Rules
  - Static Scopes
  - Dynamic Scopes



Storage Management

- Object lifetimes correspond to one of three storage allocation mechanisms:
  - Static:
    - absolute address retained throughout program's execution
  - Stack:
    - Allocated & deallocated in last-in, first-out order, with subroutine calls and returns)
  - Heap:
    - Allocated and deallocated at arbitrary times.
- Stack and heap is used for dynamic allocation

3

Static Allocation

- Static allocation for
  - Machine language translation of the source code
  - Global variables
  - Local but static variables
  - explicit constants (including strings, sets, etc)
  - scalars may be stored in the instructions

4

Static Allocation

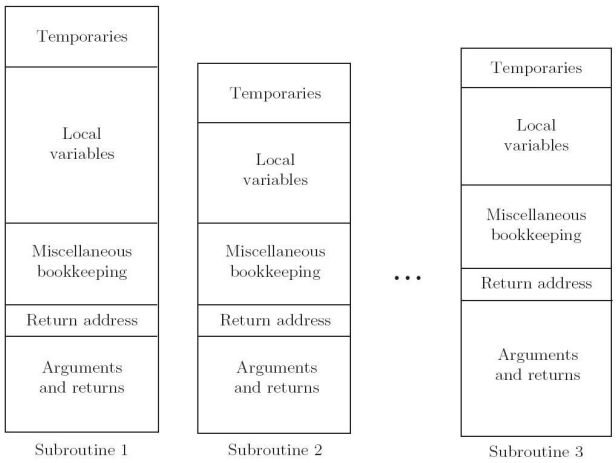


Figure 3.1: **Static allocation of space for subroutines in a language or program without recursion.**

5

Stack-based Allocation

- Why a stack?
  - If there is recursion in the language, static allocation is no more possible
  - allocate space for recursive routines (not necessary in FORTRAN – no recursion)
  - reuse space (in all programming languages)
- Central stack for
  - parameters
  - local variables
  - temporaries

6

Stack-based Allocation

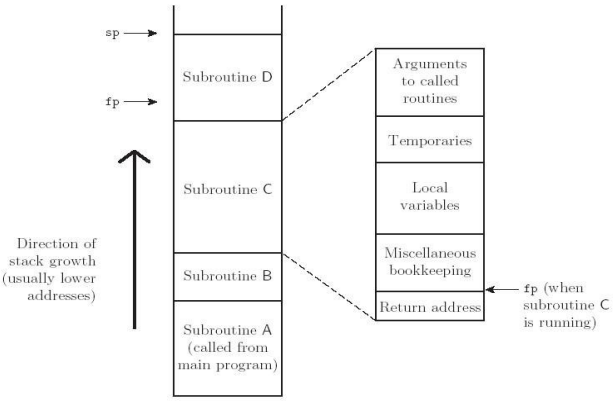


Figure 3.2: **Stack-based allocation of space for subroutines.** We assume here that subroutine A has been called by the main program, and that it then calls subroutine B. Subroutine B subsequently calls C, which in turn calls D. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame (activation record) of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

7

Stack-based Allocation

- Contents of a stack frame
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

8

Heap-based Allocation

- Allocation & deallocation can be done in random order.
- Space management issues:
  - Internal fragmentation
    - A block that is larger than required is allocated
    - Extra space is wasted
  - External fragmentation
    - Although there is sufficient space, the unused space is scattered and no one piece is large enough to satisfy a request

9

External Fragmentation

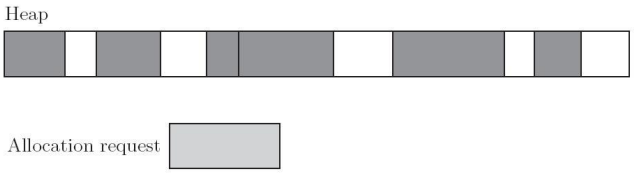


Figure 3.3: **External fragmentation.** The shaded blocks are in use; the clear blocks are free. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

10

Storage Management Techniques

- First fit: select the first block on the list that is large enough to satisfy the request
- Best fit: search the entire list to find the smallest block large enough to satisfy the request
  - Does a better job in preventing external fragmentation
  - Higher allocation cost → needs to search the entire list

- Divide large free blocks into smaller pieces
- Merge (coalesce) adjacent free blocks

- Use n fixed sized lists (eg. for k=1…n, use 2^k size blocks)
  - Decrease the search time

11

Scope Rules

- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted (see below)
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (theses variable are said to have a "hole" in their scope)
  - make references to variables

12

Scope Rules

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated

13

Scope Example

```
#include <stdio.h>

int A = 5;

void subfunc(){
    int A = 4;
    printf("%d", A);
}

void main(){
    int A = 3;
    subfunc();
    printf("%d", A);
}
```

Output: 43

14

Scope Example - II

```
Procedure A(..);

    Procedure B(..);

        Procedure C(..);
        begin … end;

        Procedure D(..);
        begin … end;

    begin … end;

    Procedure E(..);
    begin … end;

begin … end;
```

➡ Static Scope Rules

Static Chains

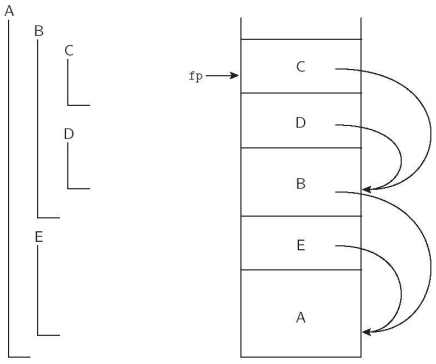


Figure 3.5: **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

Scope Rules

- The key idea in **static scope rules** is that bindings are defined by the physical (lexical) structure of the program.
- With **dynamic scope rules**, bindings depend on the current state of program execution
  - They cannot always be resolved by examining the program because they are dependent on calling sequences
  - To resolve a reference, we use the most recent, active binding made at run time

17

Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
  - early LISP dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

18

Scope Rules
Example: Static vs. Dynamic

```
program scopes (input, output );
 var a : integer;

 procedure first;
   begin
        a := 1;
   end;

 procedure second;
   var a : integer;
   begin
        first;
   end;

begin
   a := 2; second; write(a);
end.
```

19

Scope Rules
Example: Static vs. Dynamic

- If static scope rules are in effect (as would be the case in Pascal), the program prints a `1`
- If dynamic scope rules are in effect, the program prints a `2`
- Why the difference? The issue is whether the assignment to the variable `a` in procedure *first* changes the variable `a` declared in the main program or the variable `a` declared in procedure *second*.

20

Scope Rules
Example: Static vs. Dynamic

- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time
  - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - This is generally considered bad programming practice nowadays
    - Alternative mechanisms exist
      - static variables that can be modified by auxiliary routines
      - default and optional parameters

21

Scope Rules
Example: Static vs Dynamic

Dynamic (cont.):

- At run time we create a binding for `a` when we enter the main program.
- Then we create another binding for `a` when we enter procedure *second*
  - This is the most recent, active binding when procedure *first* is executed
  - Thus, we modify the variable local to procedure *second*, not the global variable
  - However, we write the global variable because the variable `a` local to procedure second is no longer active

22