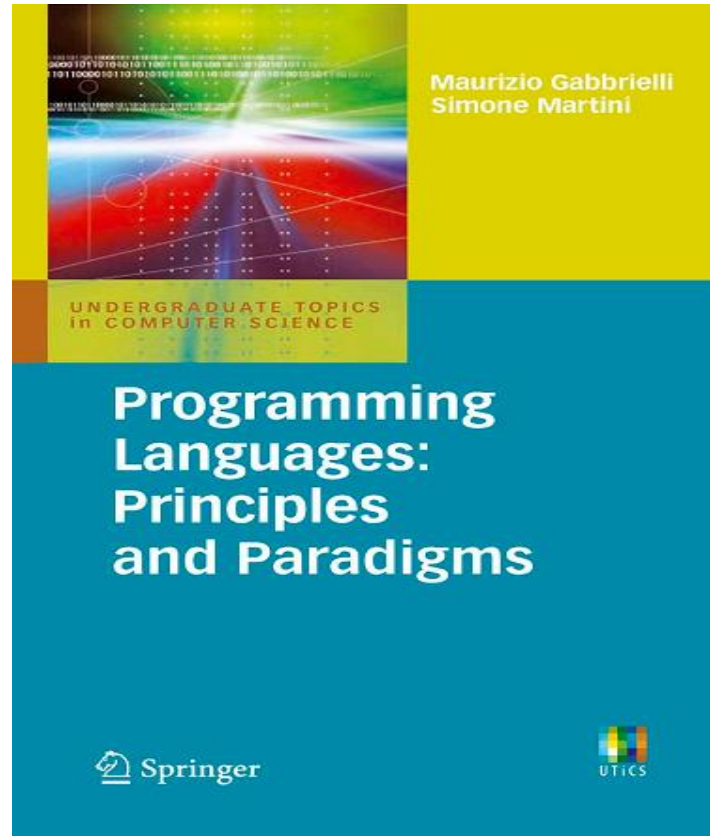


University of Information Technology (UIT)



Chapter-5 Memory Management

Techniques for Memory Management

- Memory management is one of the functions of the interpreter associated with an abstract machine.
- This functionality manages the allocation of memory for programs and for data
- In the case of a low-level abstract machine, the hardware, for example, memory management is very simple and can be entirely static.
- Before execution of the program begins, machine language program and its associated data is loaded into an appropriate area of memory, where it remains until its execution ends.

Techniques for Memory Management

- In the case of a high-level language are more complicated.
- First of all, if the language permits recursion, static allocation is insufficient.
- We can statically establish the maximum number of active procedures at any point during execution in the case of languages without recursion
- we have recursive procedures this is no longer true because the number of simultaneously active procedure calls can depend on the parameters of the procedures or, on information available only at runtime.

Techniques for Memory Management

Example 5.1 Consider the following fragment:

```
int fib (int n) {
    if (n == 0) return 1;
    else if (n == 1) return 1;
    else return fib(n-1) + fib(n-2);
}
```

- If called with an argument n , computes the value of the n th Fibonacci number.
- $\text{Fib}(0) = \text{Fib}(1) = 1$; $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$, for $n > 1$.
- It is clear that the number of active calls to `Fib` depends on the value of the argument, n .

Techniques for Memory Management

- Every procedure call requires its own memory space to store parameters, intermediate results, return addresses, and so on
- In recursive procedures, static allocation of memory is no longer sufficient and we have to allow dynamic memory allocation and deallocation operations, which are performed during the execution of the program.
- Such dynamic memory processing can be implemented in a natural fashion using a stack for procedure (or in-line block) activations since they follow a LIFO (Last In First Out) policy—the last procedure called (or the last block entered) will be the first to be exited.
- When it is not possible to use a stack to manage the memory, a particular memory structure called a heap is used.

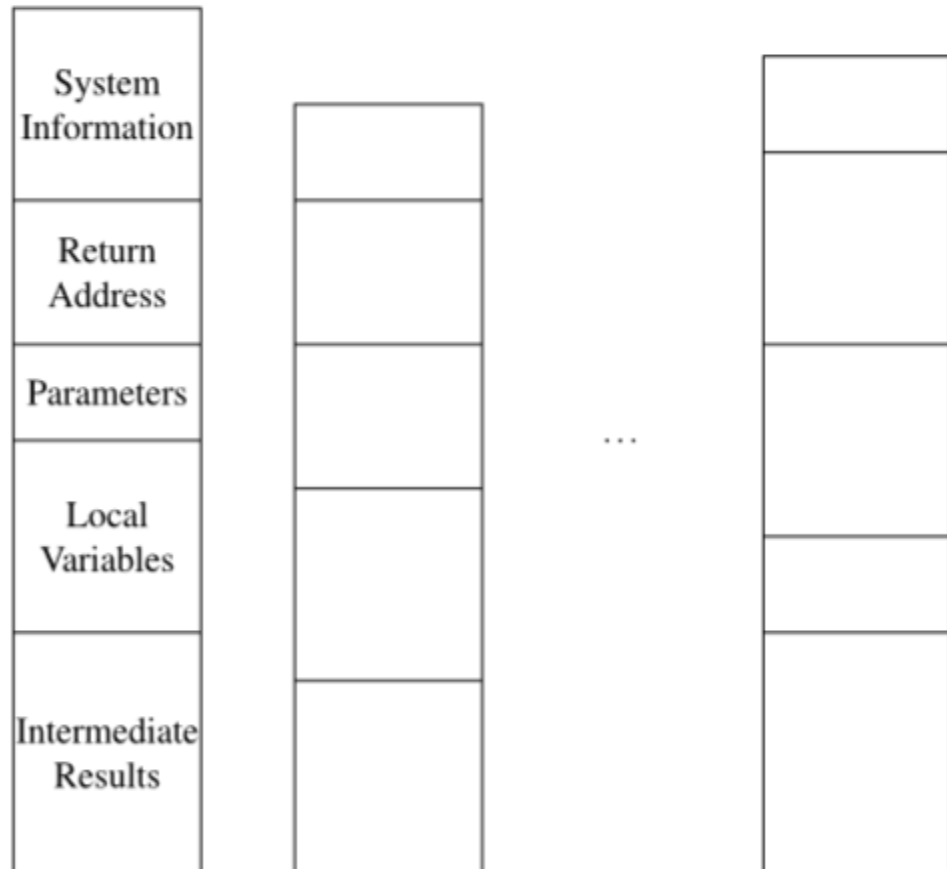
Static Memory Management

- Static memory management is performed by the compiler before execution starts.
- Statically allocated memory objects reside in a fixed zone of memory (which is determined by the compiler) and they remain there for the entire duration of the program's execution.
- Typical elements for which it is possible statically to allocate memory are global variables.
- The object code instructions produced by the compiler can be considered another kind of static object, given that normally they do not change during the execution of the program, so in this case also memory will be allocated by the compiler.
- Constants are other elements that can be handled statically
- Finally, various compiler-generated tables, necessary for the runtime support of the language are stored in reserved areas allocated by the compiler.

Static Memory Management

- In the case in which the language does not support recursion, it is possible statically to handle the memory for other components of the language.

Fig. 5.1 Static memory management



Dynamic Memory Management Using Stack

- Most modern programming languages allow block structuring of programs.
- Blocks are entered and left using the LIFO scheme.
- When a block A is entered, and then a block B is entered, before leaving A, it is necessary to leave B.
- It is therefore natural to manage the memory space required to store the information local to each block using a stack.

Example 5.2 Let us consider the following program:

```
A: { int a = 1;  
    int b = 0;  
  
    B: { int c = 3;  
        int b = 3;  
    }  
    b=a+1;  
}
```


Dynamic Memory Management Using Stack

Fig. 5.2 Allocation of an activation record for block A in Example 5.2

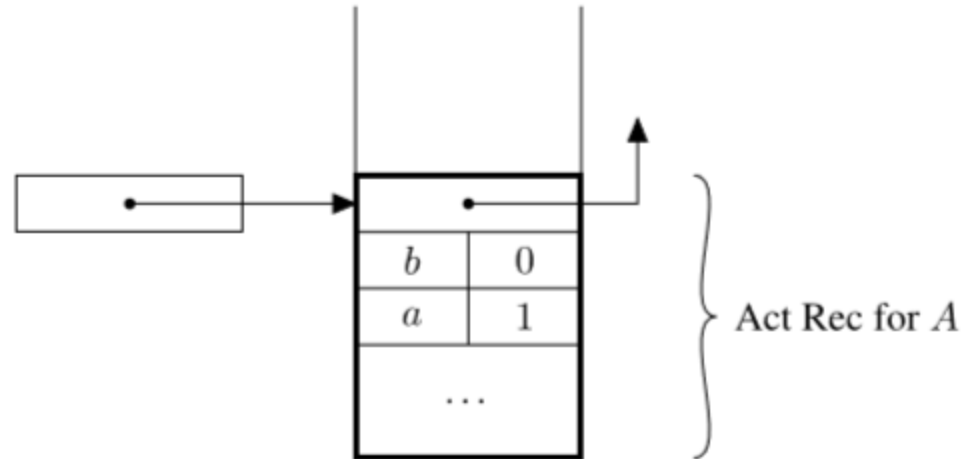
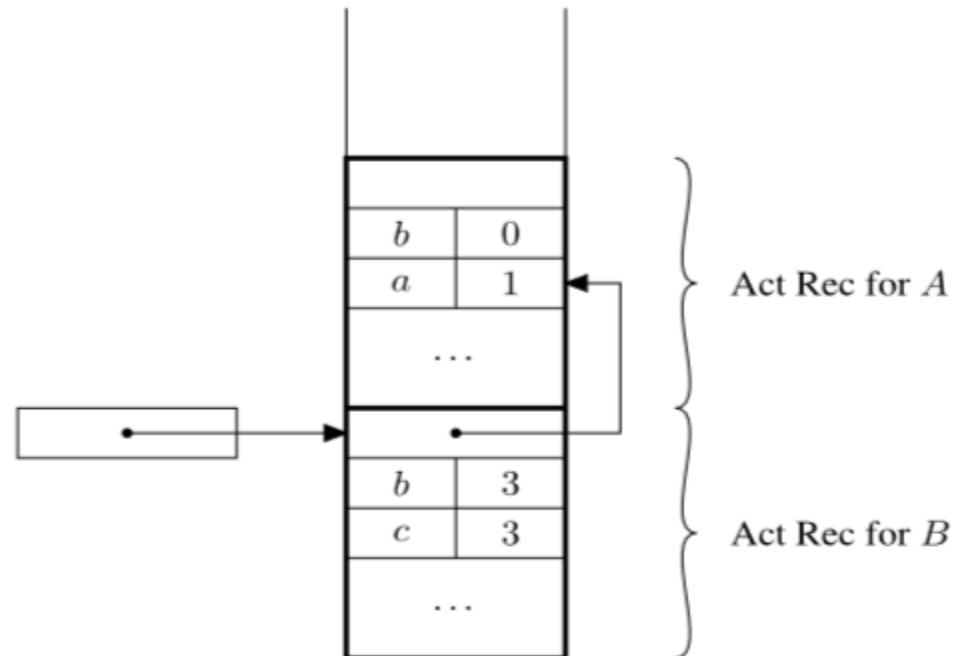
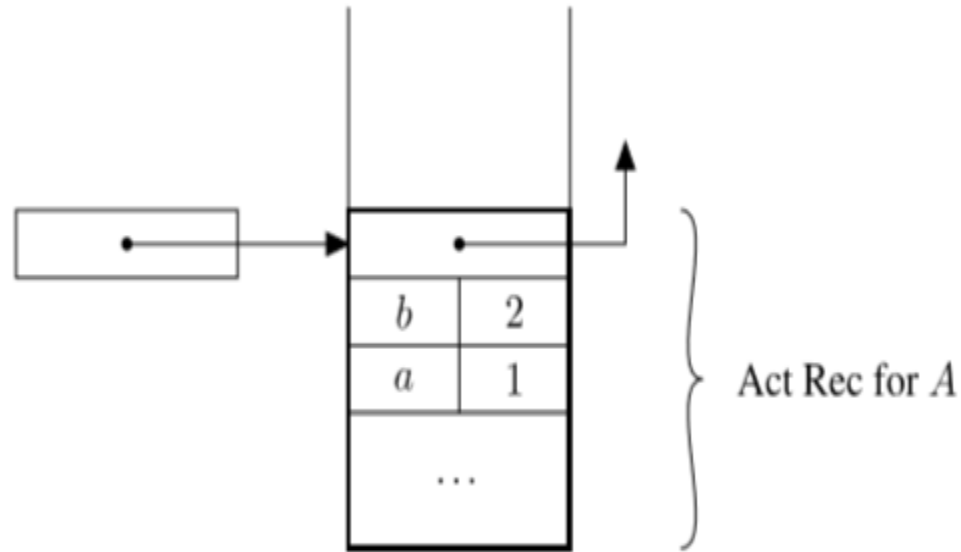


Fig. 5.3 Allocation of activation records for blocks A and B in Example 5.2



Dynamic Memory Management Using Stack

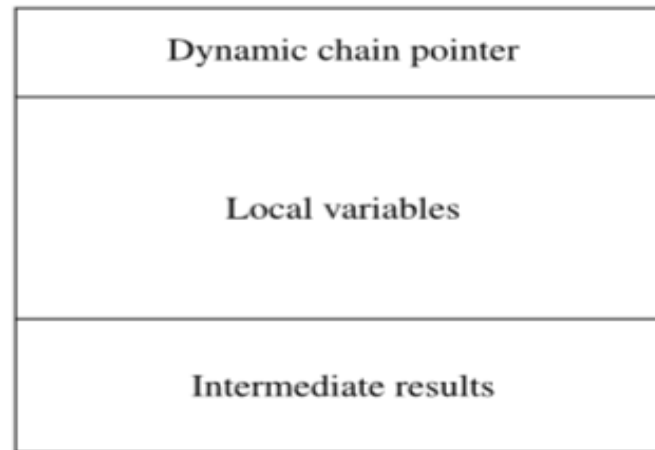
Fig. 5.4 Organisation after the execution of the assignment in Example 5.2



- The memory space, allocated on the stack, dedicated to an in-line block or to an activation of a procedure is called the activation record, or frame.
- The stack on which activation records are stored is called the runtime (or system) stack.

Activation Records for In-line Blocks

Fig. 5.5 Organisation of an activation record for an in-line block



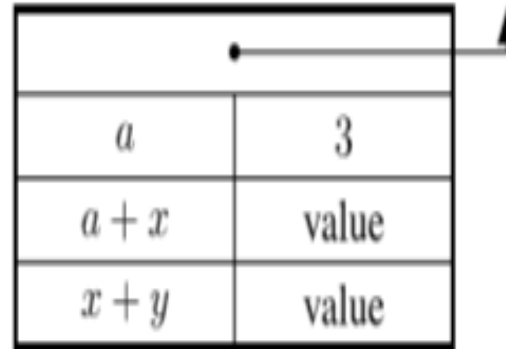
- The various sectors of the activation record contain the following information:
- **Intermediate results** When calculations must be performed, it can be necessary to store intermediate results, for example, the activation record for the block:

```
{int a =3;
  b= (a+x)/ (x+y); }
```

could take the form shown in Fig. 5.6

Activation Records for In-line Blocks

Fig. 5.6 An activation record with space for intermediate results



- **Local variables** Local variables which are declared inside blocks, must be stored in a memory space whose size will depend on the number and type of the variables.
- **Dynamic chain pointer** This field stores a pointer to the previous activation record on the stack. This information is necessary because activation records have different sizes. Some authors call this pointer the dynamic link or control link. The set of links implemented by these pointers is called the dynamic chain.

Activation Records for Procedures

- The case of procedures and functions is analogous to that of in-line blocks but with some additional complications due to the fact that, when a procedure is activated, it is necessary to store a greater amount of information to manage correctly the control flow.

Fig. 5.7 Structure of the activation record for a procedure

Dynamic Chain Pointer
Static Chain Pointer
Return Address
Address for Result
Parameters
Local Variables
Intermediate Results

Activation Records for Procedures

- **Intermediate results, local variables, dynamic chain pointer** The same as for in-line blocks.
- **Static chain pointer** This stores the information needed to implement the static scope rules.
- **Return address** Contains the address of the first instruction to execute after the call to the current procedure/function has terminated execution.
- **Returned result** Present only in functions. Contains the address of the memory location where the subprogram stores the value to be returned by the function when it terminates. This memory location is inside the caller's activation record.
- **Parameters** The values of actual parameters used to call the procedure or function are stored here.

Stack Management

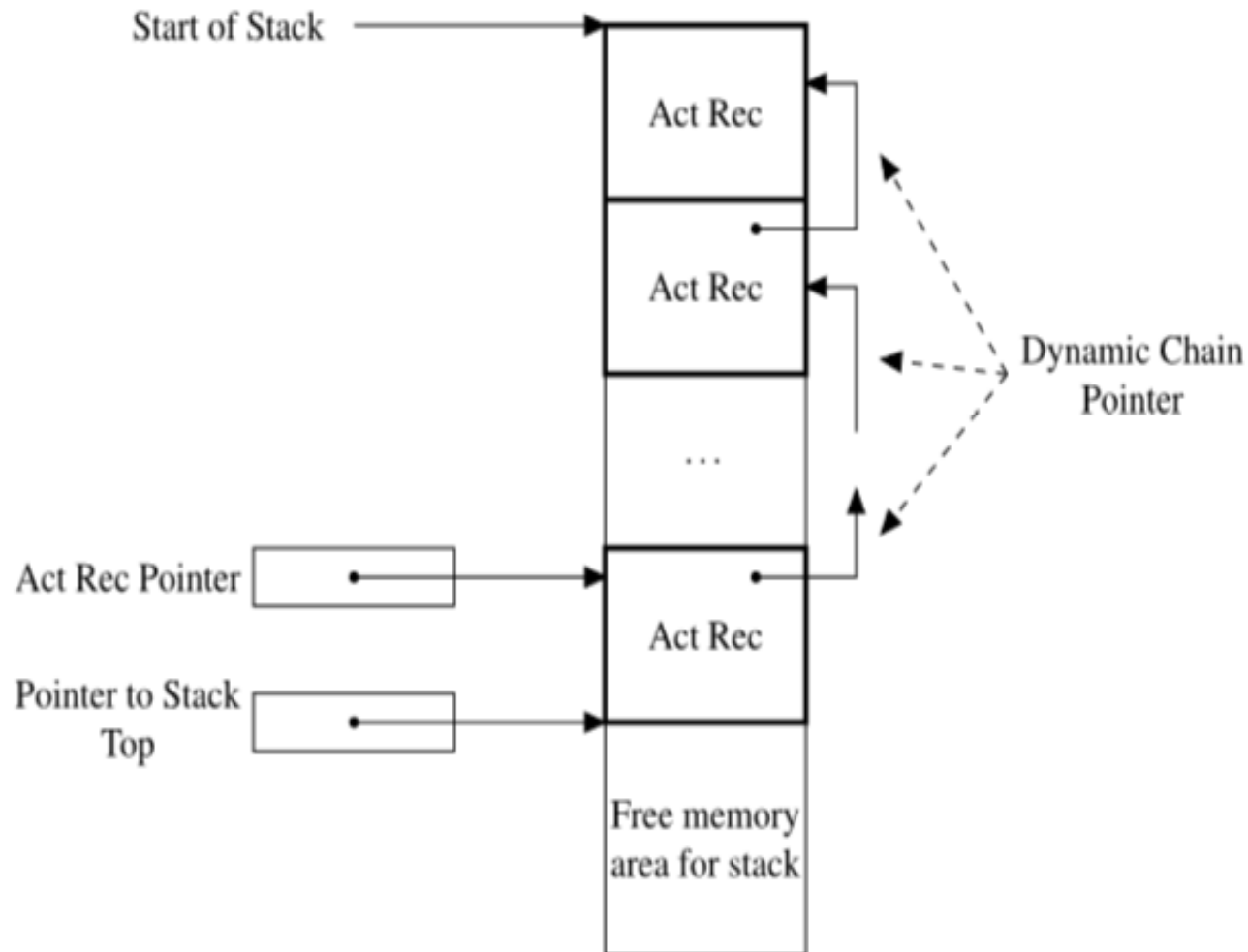


Fig. 5.8 The stack of activation records

Stack Management

- As shown in the figure, an external pointer to the stack points to the last activation record on the stack
- This pointer, which we call the activation record pointer, is also called the frame or current environment pointer.
- In the figure, we have also indicated where the first free location is. This second pointer can also be omitted if the activation-record pointer always points to a position
- Activation records are stored on and removed from the stack at runtime. When a block is entered or a procedure is called, the associated activation record is pushed onto the stack; it is later removed from the stack when the block is exited or when the procedure terminates.
- The runtime management of the system stack is implemented by code fragments which the compiler (or interpreter) inserts immediately before and after the call to a procedure or before the start and after the end of a block.

Stack Management

- We use “caller” and “callee” to indicate, respectively, the program or procedure that performs a call (of a procedure) and the procedure that has been called.
- Stack management is performed both by the caller and by the callee. To do this, as well as handling other control information, a piece of code called the calling sequence is inserted into the caller to be executed, in part, immediately before the procedure call. The remainder of this code is executed immediately after the call. In addition, in the callee two pieces of code are added: a prologue, to be executed immediately after the call, and an epilogue, which is executed when the procedure ends execution. These three code fragments manage the different operations needed to handle activation records and correctly implement a procedure call.

Stack Management

- **Modification of program counter** This is clearly necessary to pass control to the called procedure. The old value(incremented) must be saved to maintain the return address.
- **Allocation of stack space** The space for the new activation record must be pre-allocated and therefore the pointer to the first free location on the stack must be updated as a consequence.
- **Modification of activation record pointer** The pointer must point to the new activation record for the called procedure; the activation record will have been pushed onto the stack.
- **Parameter passing** This activity is usually performed by the caller, given that different calls of the same procedure can have different parameters.
- **Register save** Values for control processing, typically stored in registers, must be saved. This is the case, for example, with the old activation record pointer which is saved as a pointer in the dynamic chain.
- **Execution of initialisation code** Some languages require explicit constructs to initialise some items stored in the new activation record.

Stack Management

- When control returns to the calling program, i.e. when the called procedure terminates, the epilogue (in the called routine) and the calling sequence (in the caller) must perform the following operations:
- Update of program counter This is necessary to return control to the caller.
- Value return The values of parameters which pass information from the caller to the called procedure, or the value calculated by the function, must be stored in appropriate locations usually present in the caller's activation record and accessible to the activation record of the called procedure.
- Return of registers The value of previously saved registers must be restored. In particular, the old value of the activation record pointer must be restored.
- Execution of finalisation code Some languages require the execution of appropriate finalisation code before any local objects can be destroyed.
- Deallocation of stack space The activation record of the procedure which has terminated must be removed from the stack. The pointer to (the first free position on) the stack must be modified as a result.



Stack Management

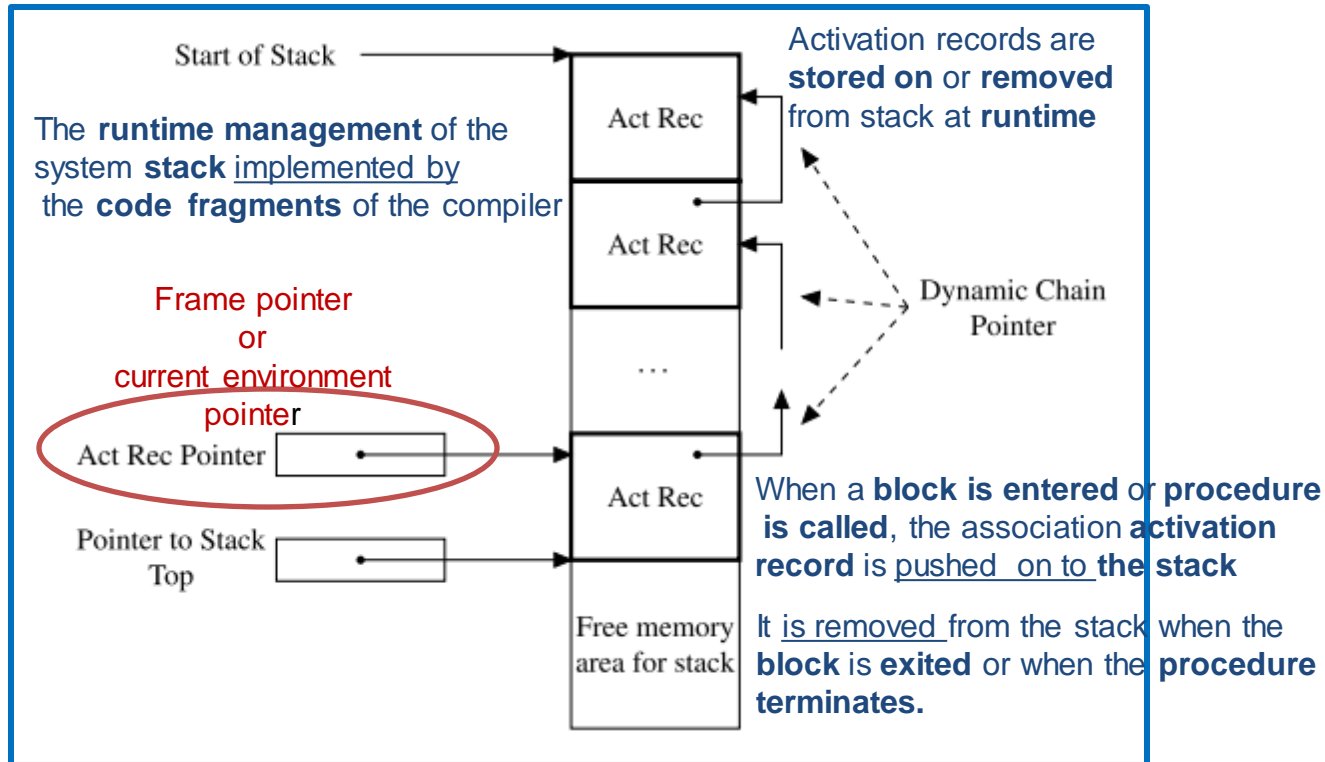


Figure: Structure of activation record for a procedure



Stack Management



Stack management is performed by the **caller** and **callee**.



To handle the **control information**, a piece of **code**, **calling sequence** is inserted into the **caller** to be executed.



In the **callee**, **two pieces of codes**, ***prologue*** and ***epilogue*** are added.



Prologue is executed immediately after a **call** and **epilogue** is executed when the **procedure ends execution**.



These three code fragments manage the **different operations** needed to handle **activation records** and correctly implement a **procedure call**.

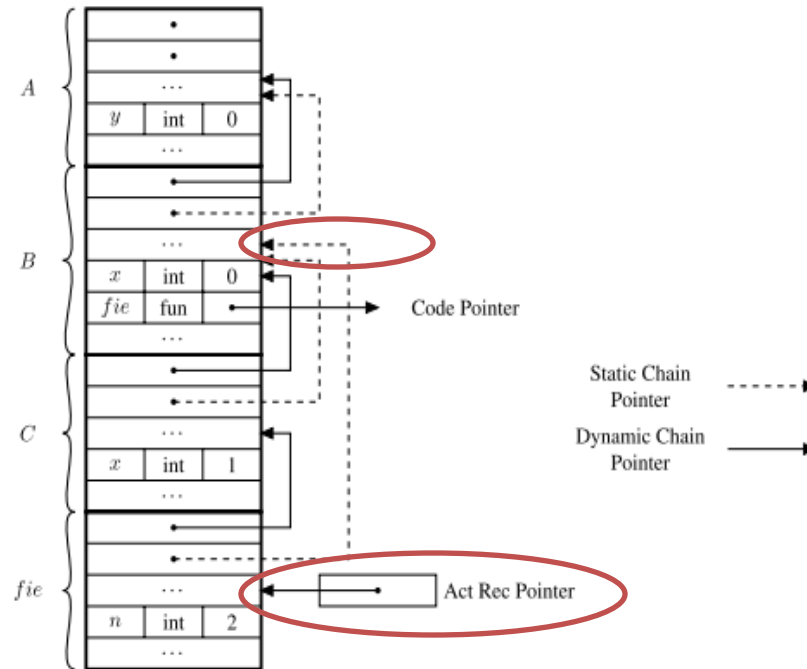


Static Scope: Static Chain



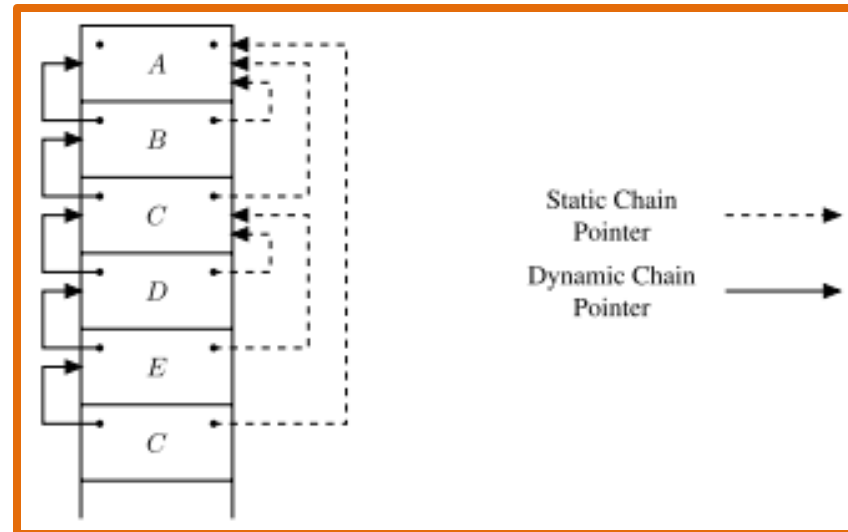
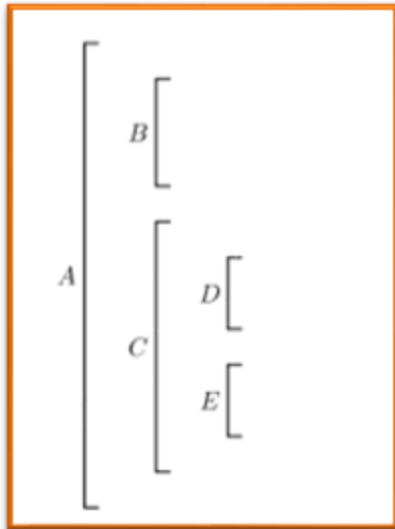
Static scope is defined by the **textual structure** of the program.

```
A: { int y=0;  
  B: { int x=0;  
    void fie (int n) {  
      x = n+1;  
      y = n+2;  
    } // end of fie fun  
  C: { int x=1;  
      fie (2);  
      write (x);  
    } // end of C  
  } // end of B  
  write (y);  
} //end of A
```





Static Chain for Block Structure





Static Chain Pointer

To be able to locate the information correctly at runtime,



The **activation record** for the procedure call is linked with a static chain pointer



These pointers are used to represent the **static nesting structure of the block** within the program.



The **runtime management** of the **static chain** is one of the functions performed by the **calling sequence, prologue** and **epilogue code**



Such a management of the **static chain** can be performed by the **caller and the callee**.



Static Chain Pointer



When a **new block** is entered, the **caller** calculates the **static chain pointer** and then *passes* it to the **called routine**.



This **computation** is **simple** and separated into two cases

The **called routine**
is external to the
caller

Called *inside*
calling routine

Dynamic Management Using a Heap

- In the case in which the language includes explicit commands for memory allocation, as for example do C and Pascal, management using just the stack is insufficient. Consider for example the following C fragment:

```
int *p, *q; /* p,q NULL pointers to integers */
p = malloc (sizeof (int));
    /* allocates the memory pointed to by p */
q = malloc (sizeof (int));
    /* allocates the memory pointed to by q */
*p = 0;    /* dereferences and assigns */
*q = 1;    /* dereferences and assigns */
free(p);   /* deallocates the memory pointed to by p */
free(q);   /* deallocates the memory pointed to by q */
```

Dynamic Management Using a Heap

- Given that the memory deallocation operations are performed in the same order as allocations (first p, then q), the memory cannot be allocated in LIFO order.
- To manage explicit memory allocations, which can happen at any time, a particular area of memory, called a heap, is used.
- This term is used in computing also to mean a particular type of data structure which is representable using a binary tree or a vector, used to implement efficiently priorities
- In the programming language, a heap is simply an area of memory in which blocks of memory can be allocated and deallocated relatively freely.
- Heap management methods fall into two main categories according to whether the memory blocks are considered to be of fixed or variable length.

Fixed-Length Blocks

- In this case, the heap is divided into a certain number of elements, or blocks, of fairly small fixed length, linked into a list structure called the free list, as shown in Fig.5.9.
- At runtime, when an operation requires the allocation of a memory block from the heap (for example using the malloc command), the first element of the free list is removed from the list, the pointer to this element is returned to the operation that requested the memory and the pointer to the free list is updated so that it points to the next element.
- When memory is, on the other hand, freed or deallocated (for example using free), the freed block is linked again to the head of the free list. The situation after some memory allocations is shown in Fig. 5.10.

Fixed-Length Blocks

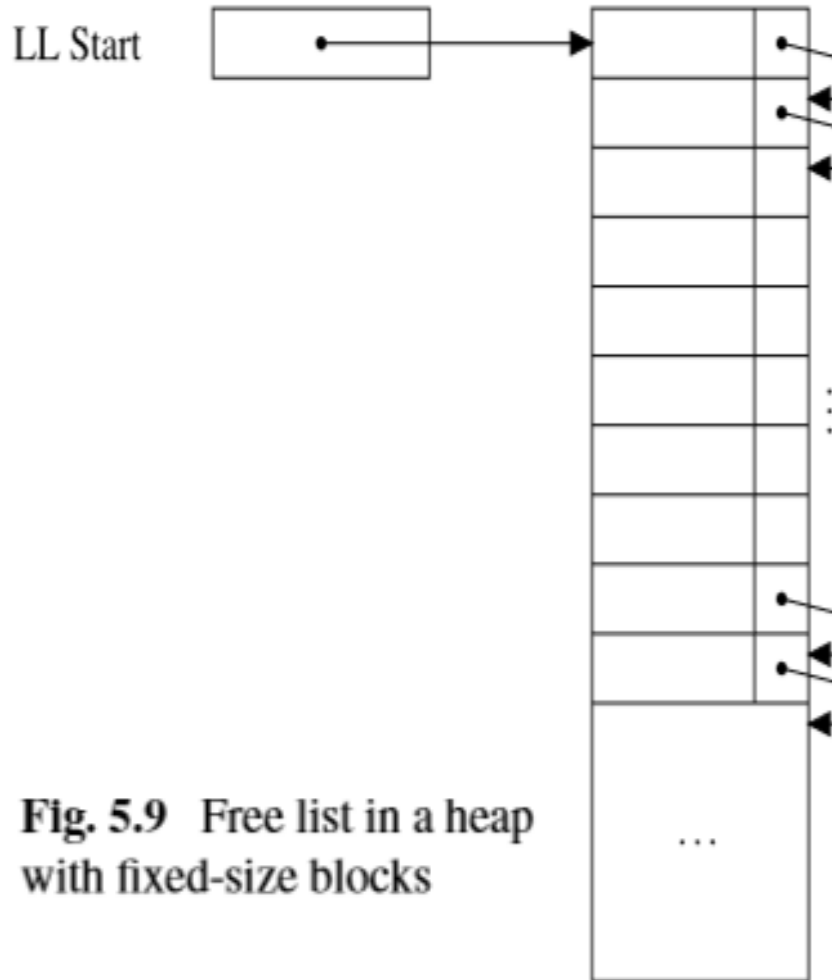


Fig. 5.9 Free list in a heap with fixed-size blocks

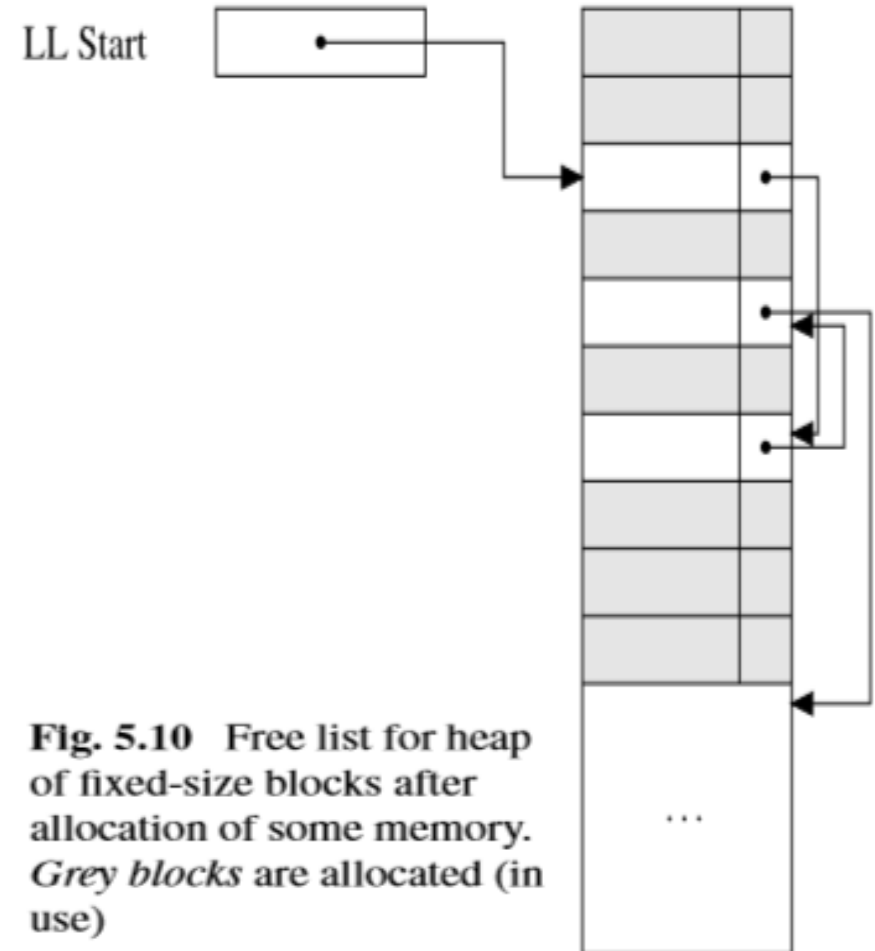


Fig. 5.10 Free list for heap of fixed-size blocks after allocation of some memory. *Grey blocks* are allocated (in use)



Why Dynamic Scope is used?



Dynamic scope is **much simpler than** static scope.



To resolve the **non-local reference** to **name x**,

- ✓ **start** from the **current activation record**
- ✓ **running backwards** from the stack
- ✓ *until* the **activation records** which include **name x** is found



For variable **association** between **names** and their denoted **object**,

- ✓ various **local environment** can be stored directly in the **activation record**

Direct Storage in Activation Record

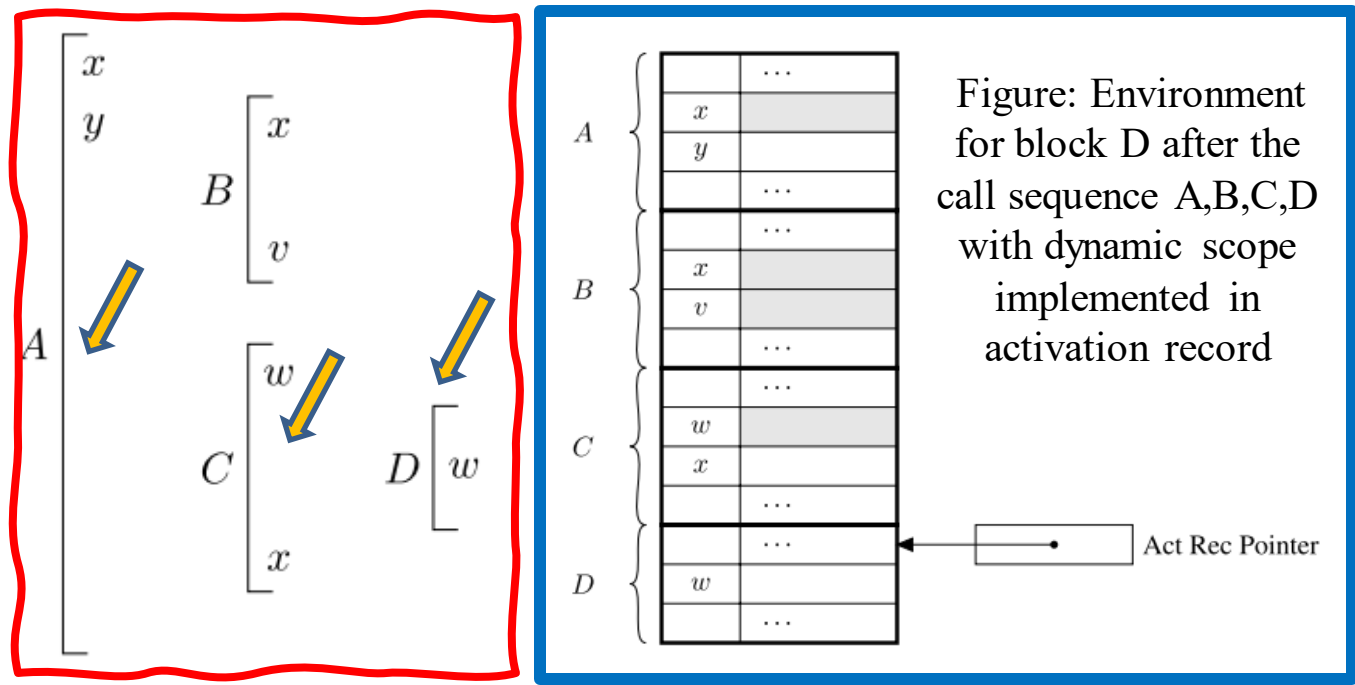


Figure: Environment for block D after the call sequence A,B,C,D with dynamic scope implemented in activation record

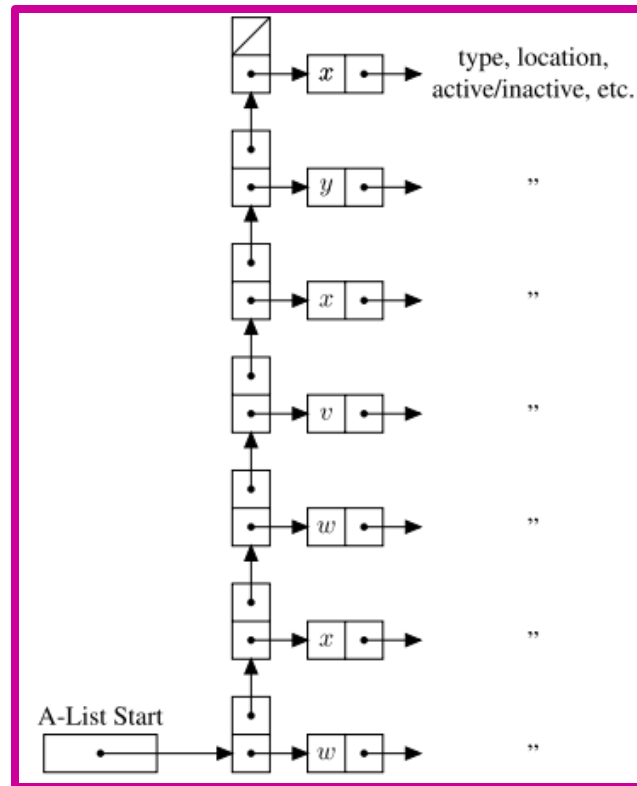
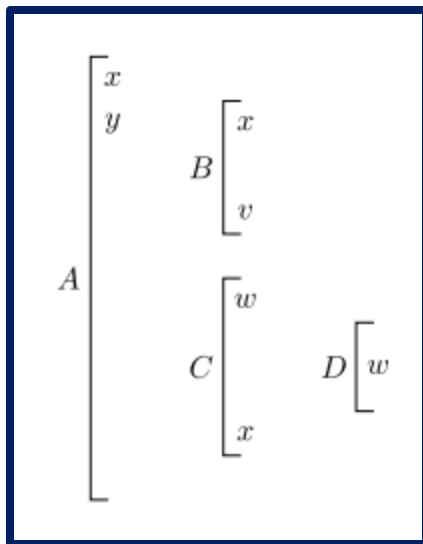


Association List (A-list)

- ☀ **Name-object association** can be separately stored in association list , **A-list**
- ☀ When the **execution of a program** enters a **new environment**,
 - ✓ the **new local associations** are inserted into the **A-list**.
- ☀ When **an environment** is left,
 - ✓ the **local associations** are removed *from* the **A-list**.
- ☀ The information about the denoted object contains,
 - ✓ the **location** objects in memory
 - ✓ **its types**
 - ✓ **flag** which indicates **association** for this **object** is **active or not**




Association List: Example





Disadvantages of Using A-List and Direct Storage



2.
The **inefficiency**
of this
runtime search

1.
Names must be
stored in
structures
present
at runtime



Central Referencing environment Table (CRT)

To restrict the **two disadvantages**, the cost of greater **inefficiency** in **block entry and exit**,

- ✓ Central Referencing environment Table (CRT) is used.

Using CRT-based technique,

- ✓ **environments** are defined by **arranging for all the blocks**
 - to refer to an **single central table (the CRT)**
- ✓ **all the names** are stored in this table.
- ✓ **a flag** which **indicate association** for each name is **active or not** is also included.
- ✓ **pointer value** point to the **information** about the object associated with the name (**memory location, type, etc.**)



CRT without Hidden Stack

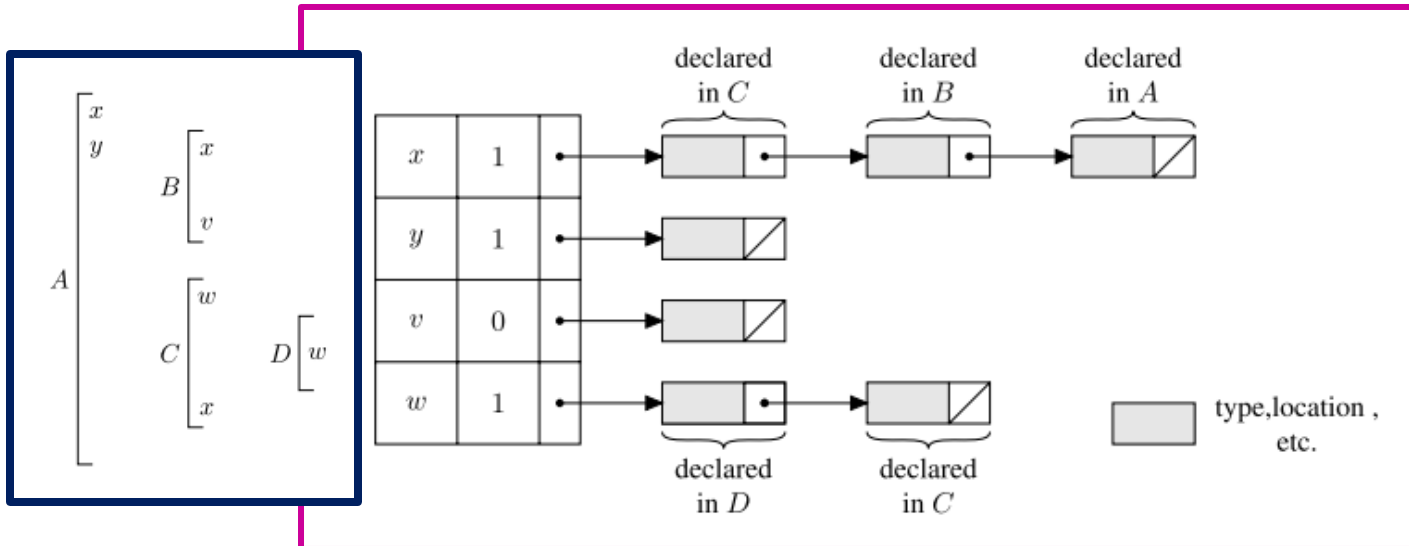
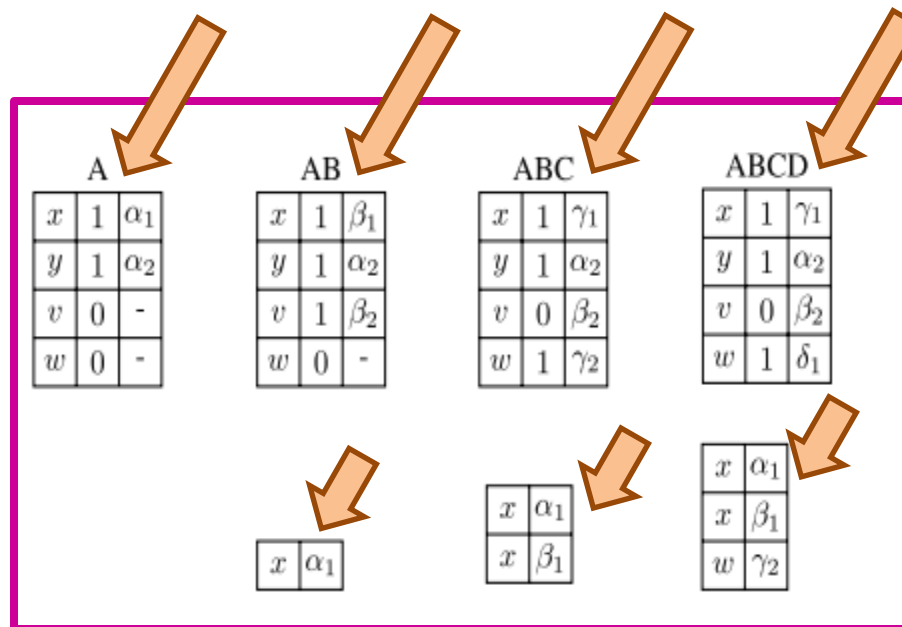
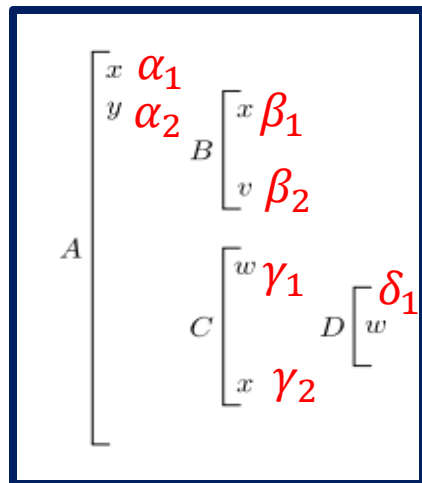


Figure: **Environment for block D** after the **call sequence A, B, C, D**, with dynamic scope implemented using a **CRT**



CRT with Hidden Stack





CRT with or without Hidden Stack

1

Using CRT with or without hidden stack, **access** to **association** in the environment requires,

- ✓ one access to the **table**
- ✓ one access to another **memory area** by a **pointer**

2

Therefore, **no runtime search** is required.



Summary

Static Scope :
Display

Static Scope :
Static Chain

Implementation of
Scope Rules

Dynamic
Scope: CRT

Dynamic Scope:
Association List

Dynamic Scope