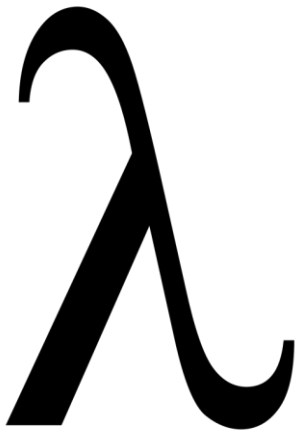


Memory Management

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Today

- Recap
- Memory allocation
  - Static allocation
  - Dynamic allocation with a stack
  - Dynamic allocation with a heap
- Scoping rules and memory
  - Static scoping
  - Dynamic scoping

Recap

Names and denotable objects

- A name and the object it denotes are not the same thing.
  - A name is just a character string
  - Denoted object can be a complex object (variable, function, type, …)
  - A single object can have more than one name ("aliasing")
  - A single name can denote different objects at different times
- The lifetime of an object does not necessarily coincide with the lifetime of an association for that object
  - Can be longer (e.g., a variable passed to a procedure by reference)
  - Can be shorter (e.g., a region of memory dynamically deallocated)
- Binding: association between name and object
- When are bindings made?
  - "static": everything that happens prior to execution
  - "dynamic": everything that happens during execution.

Environment

- Not all the associations between names and denotable objects are fixed once and for all.
- Environment: the collection of associations between names and denotable objects that exist at runtime at a specific point in a program and at a specific moment in the execution
- Declaration: a mechanism (implicit or explicit) that creates an association in an environment.

```
int x;
int f(){
    return 0;
}
type T = int;
```

Variable declaration

Function declaration

Type declaration

Blocks

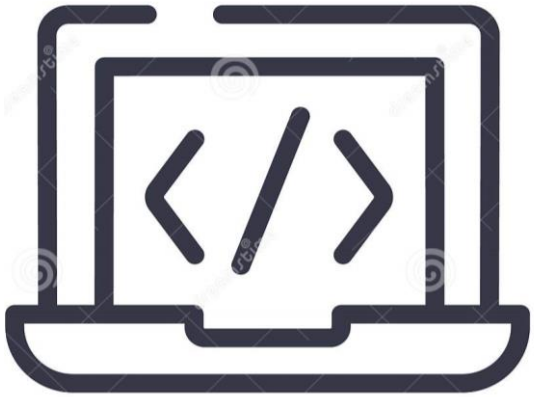
- Block: a section of the program identified by opening and closing signs that contains declarations local to that region
- We distinguish between
  - Block associated with a procedure: body of the procedure with local declarations
  - In-line (anonymous) block: it can appear in any position in which a command appears

Block environment and lifetime

- The environment in a block can be subdivided in:
  - Local environment: associated with the entry into a block (local variables, formal parameters)
  - Non-local environment: associations inherited from other blocks external to the current one
  - Global environment: part of the non-local environment that contains associations common to all blocks (global variables)

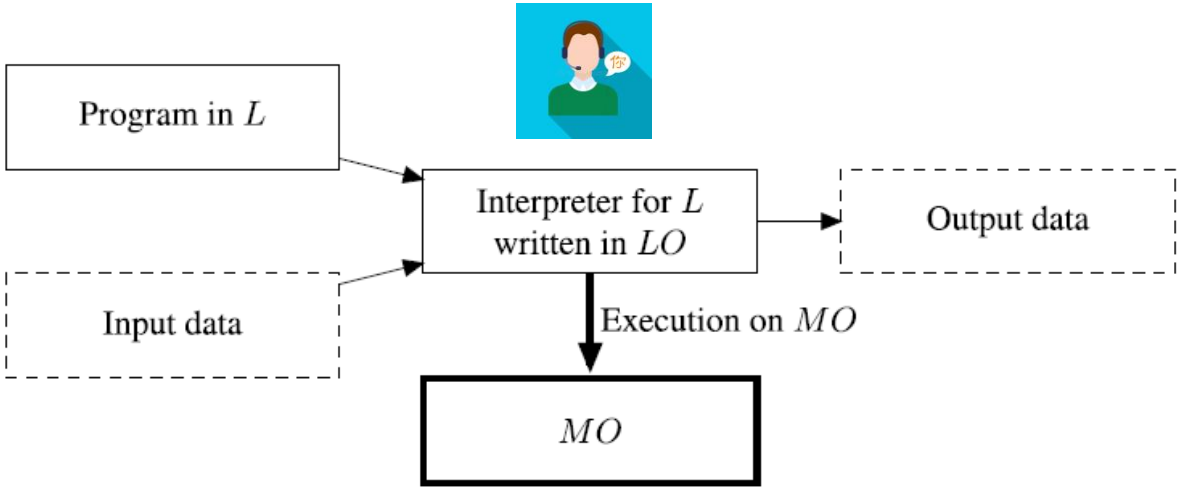
Static and dynamic scoping

- Rules of visibility. A local declaration in a block is visible in this block, and in all nested blocks, as long as these do not contain another declaration of the same name, that hides or masks the previous declaration
- Static scoping: a block A is nested in block B if block B textually includes block A
- Dynamic scoping: a block A is nested in block B if block B has been most recently activated and has not yet been deactivated



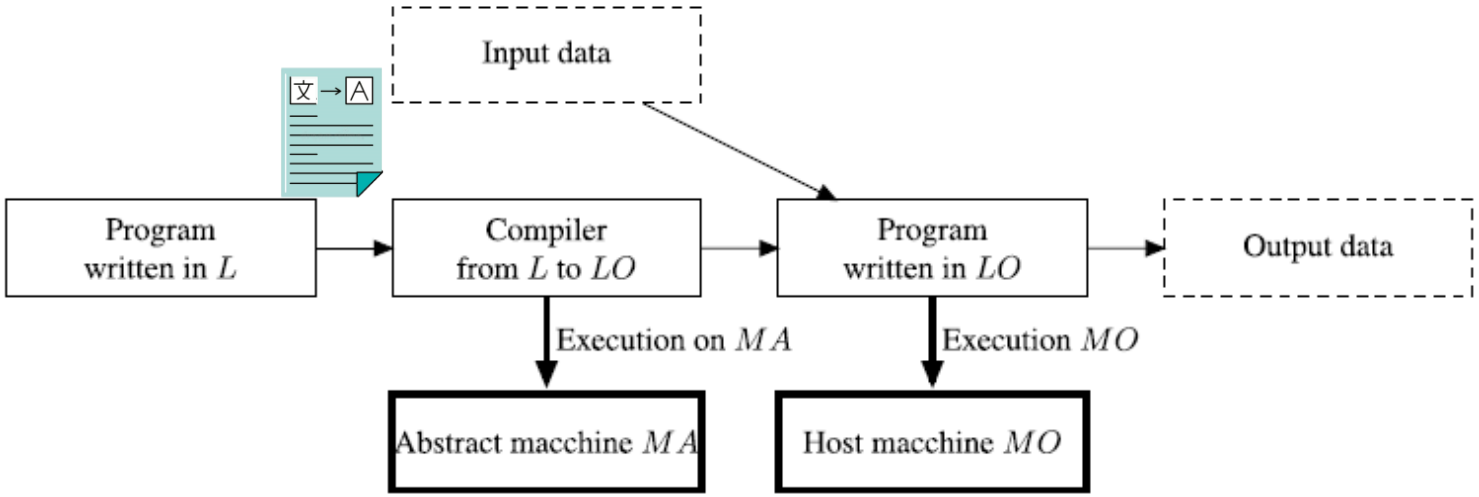
Interpreters and compilers

Purely interpreted implementation



- Interpreter: a program written in the language $\mathcal{LO}$ that is executed on $\mathcal{MO}_{\mathcal{LO}}$ and that understands and executes programs written in $\mathcal{L}$
- The interpreter translates from $\mathcal{L}$ to $\mathcal{LO}$, instruction by instruction
- Not very efficient, mainly because of the interpreter

Purely compiled implementation

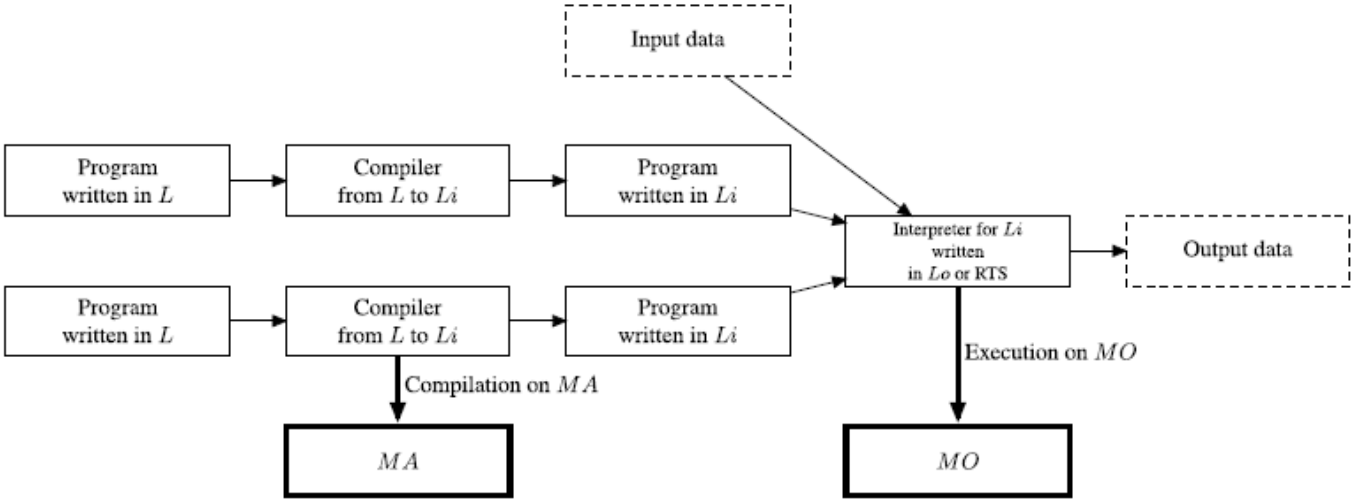


- Translation of the entire program from $\mathcal{L}$ to $\mathcal{LO}$
- The translation is done by a program called compiler
- The compiler does not have to be written in $\mathcal{LO}$
- The compiler can be executed on an abstract machine $\mathcal{MA}$ different from $\mathcal{MO}_{\mathcal{LO}}$
- More efficient but the produced code is larger

Compilation + Interpretation

- All implementations of programming languages use both. At least:
  - Compilation (= translation) from external to internal representation
  - Interpretation for I/O operations (runtime support)
- Can be modeled by identifying an Intermediate Abstract Machine $\mathcal{MI}$ with language $\mathcal{LI}$
  - A program in $\mathcal{L}$ is compiled to a program in $\mathcal{LI}$
  - The program in $\mathcal{LI}$ is executed by an interpreter for $\mathcal{MI}$

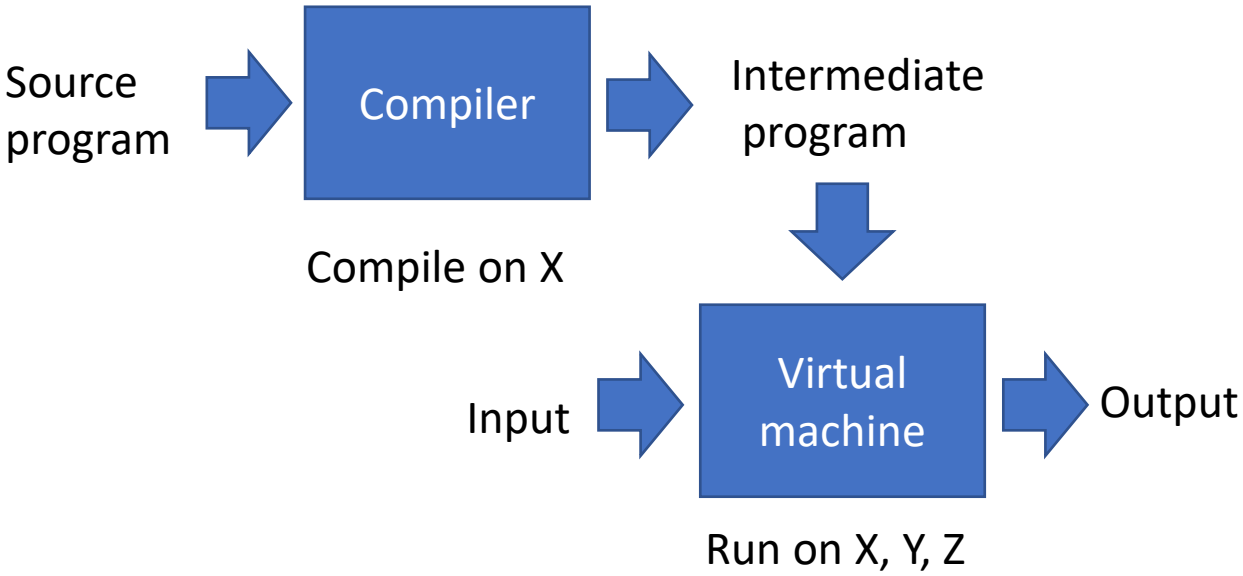
The intermediate machine



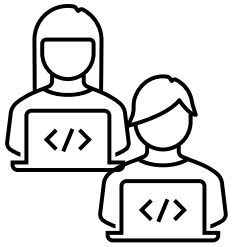
- Neither purely compiled nor purely interpreted
- The compiler translates the program into an intermediate language $\mathcal{LI}$
- This is then interpreted by a $\mathcal{MO}_{\mathcal{LO}}$-program written in $\mathcal{LO}$
- Different schemes
  - Java: compiler to bytecode, then JVM
  - C: the compiler usually creates code that must be linked and executed

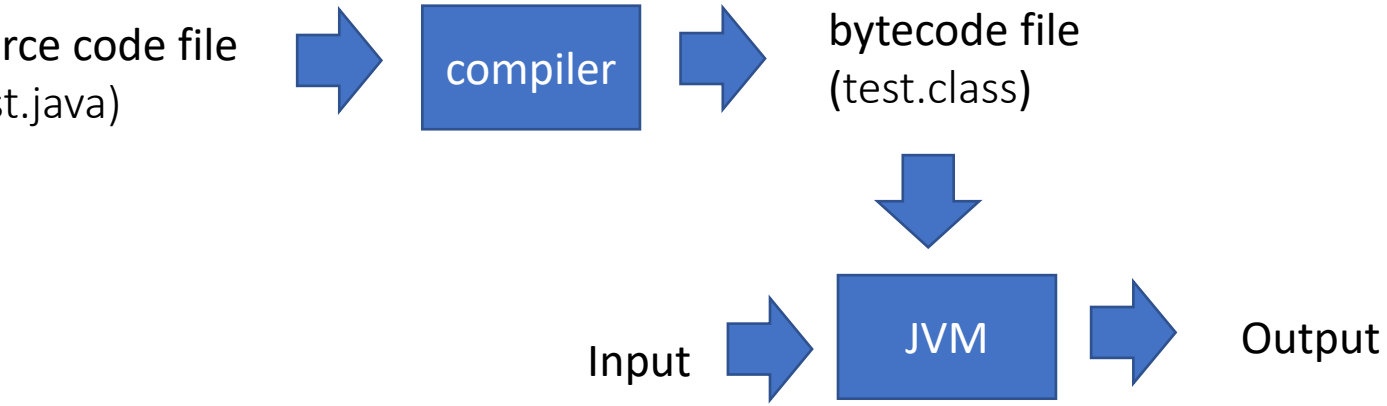
Compilation and execution on virtual machines

- Compiler generates intermediate programs
- Virtual machine interprets the intermediate program



Some examples: Java





In order to execute a program in Java ($\mathcal{L}_{\text{Java}}$), an abstract machine $\mathcal{M}_{\text{Java}}$ leverages the $\mathcal{M}_{\text{JVM}}$ that somebody has already implemented

Is it compilative or interpretative?

- Apart for the extreme cases
  - $\mathcal{L}$ = $\mathcal{LI}$ → purely interpreted
  - $\mathcal{LI}$ = $\mathcal{LO}$ → purely compiled
- Depending on the differences and similarities between $\mathcal{LI}$ and $\mathcal{LO}$, we talk about an implementation being mainly compilative or mainly interpreted
  - Java, with $\mathcal{LI}$ being bytecode, is very different from the machine language
    - So the implementation is mainly interpretative
    - but some JVM are based on compilation
  - C: $\mathcal{LI}$ is more or less the machine language
    - So the implementation is mainly compilative
    - But not identical: The language contains system calls, library functions etc.

Memory allocation

Memory allocation

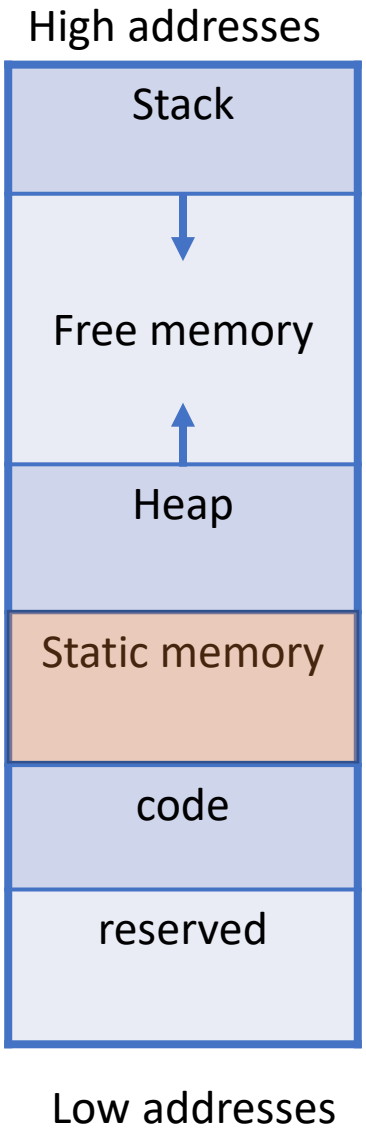
Three mechanisms for memory allocation

- Static. Memory allocated at compile time
- Dynamic. Memory allocated at execution time
  - Stack: objects allocated LIFO (Last In First Out)
    - We do not know how many recursive active procedures we are going to have
  - Heap: objects can be allocated and deallocated at any time
    - In some cases, the stack is not enough, e.g., when the language allows explicit memory allocation and deallocation operations.

Static allocation

Static allocation

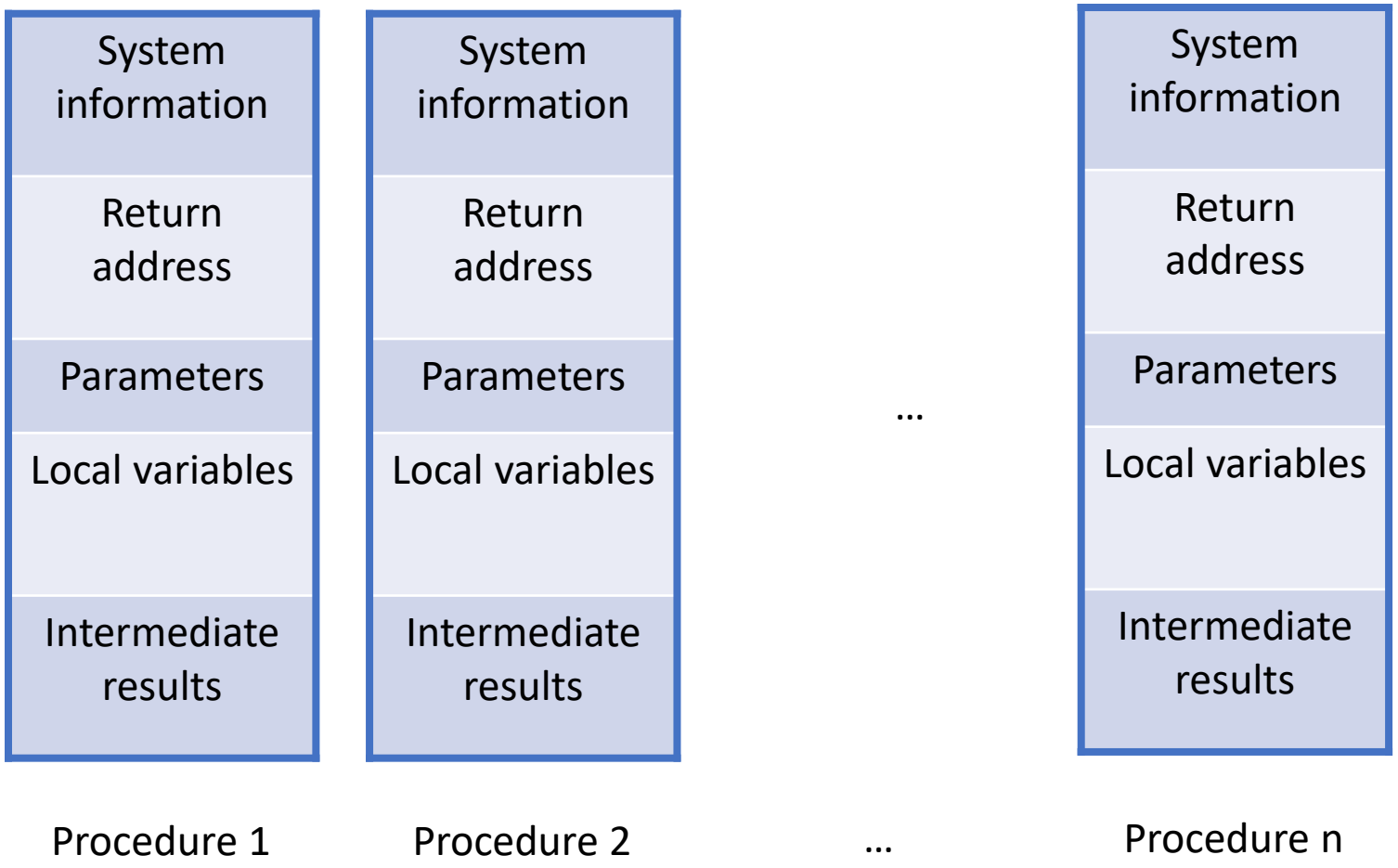
- Performed by the compiler before the execution
- An object has an absolute address (fixed zone of the memory) that is maintained throughout the execution of a program



Static allocation

- The following are usually statically allocated
  - Global variables
  - Constants (not depending on runtime execution)
  - Tables used at runtime (for type checking, garbage collection etc.)
  - For languages not supporting recursion: procedure local information (variables, parameters, return values, …)

Static allocation for subprograms



Static allocation does not allow for recursion

- Let us imagine to have static allocation and the following code

```
{
 int fact (int n) {
    if (n<=1) return 1;
    else return n*fact(n-1);
 }
fact(3)}
```

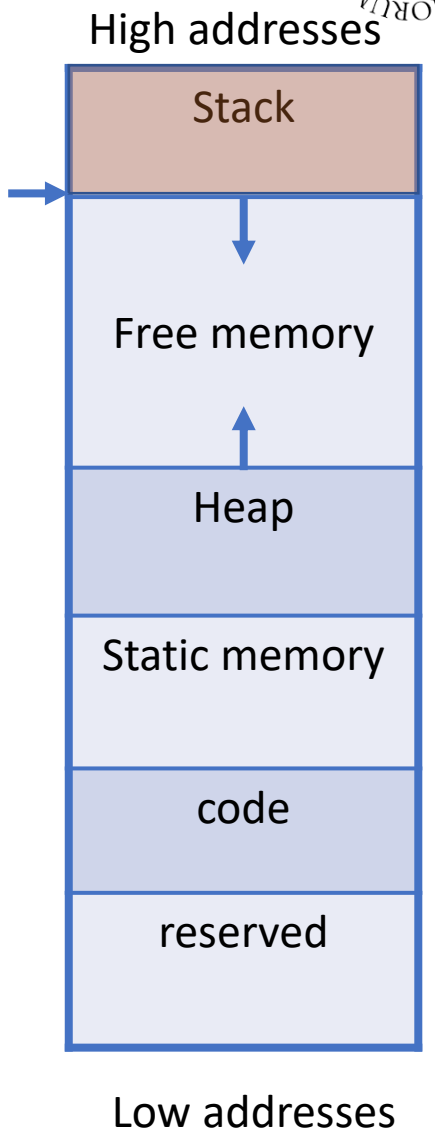
- There is then a unique static area of memory for storing `n`
  - First iteration: Store 3
  - Second iteration: Store 2
  - The original value (3) is now lost …
- In case of recursive calls, we would lose the original value

How to deal with recursive procedures?

Dynamic allocation with stack

Dynamic allocation: stack

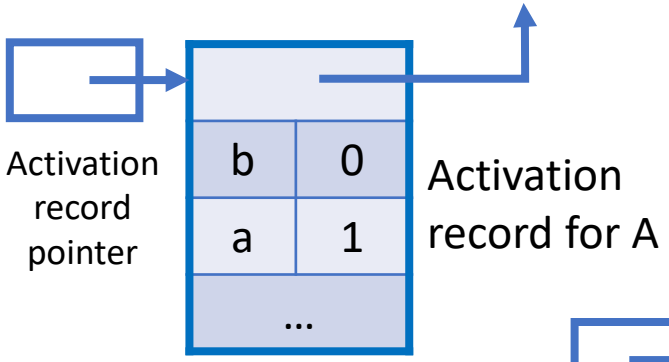
- For every runtime instance of a subprogram, we have an activation record (or frame), that contains the information about this instance
- In a similar (but simpler) way, each block also has an activation record
- Since we have block-structuredness (nested blocks), the stack (LIFO) is the natural data structure for this
- The stack on which activation records are stored is called the runtime (or system) stack.
- While not necessary, a stack can also be used in languages without recursion, to reduce memory usage



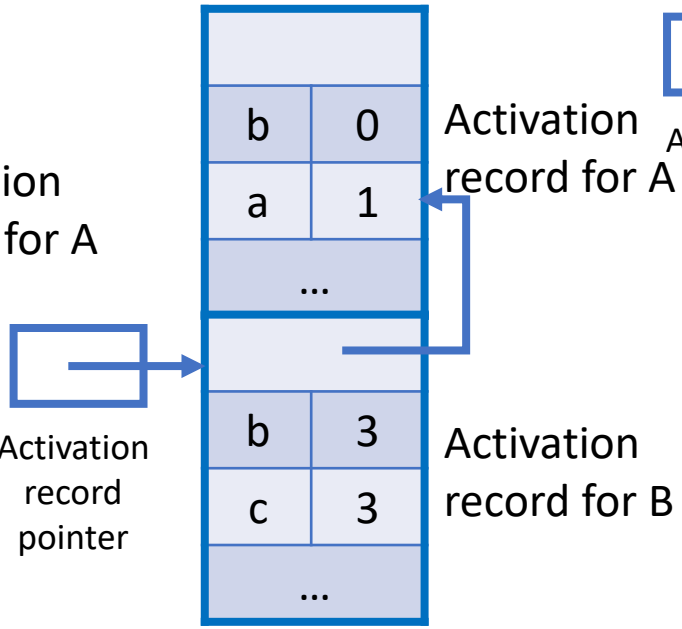
The overall idea

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

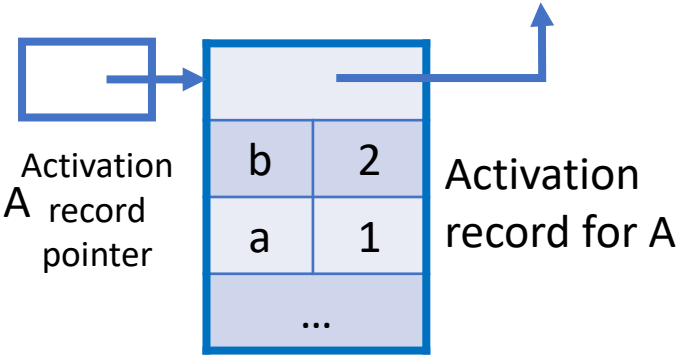
After entering block A (and assignments)



After entering block B (and assignments)



After exiting block B (and assignment)

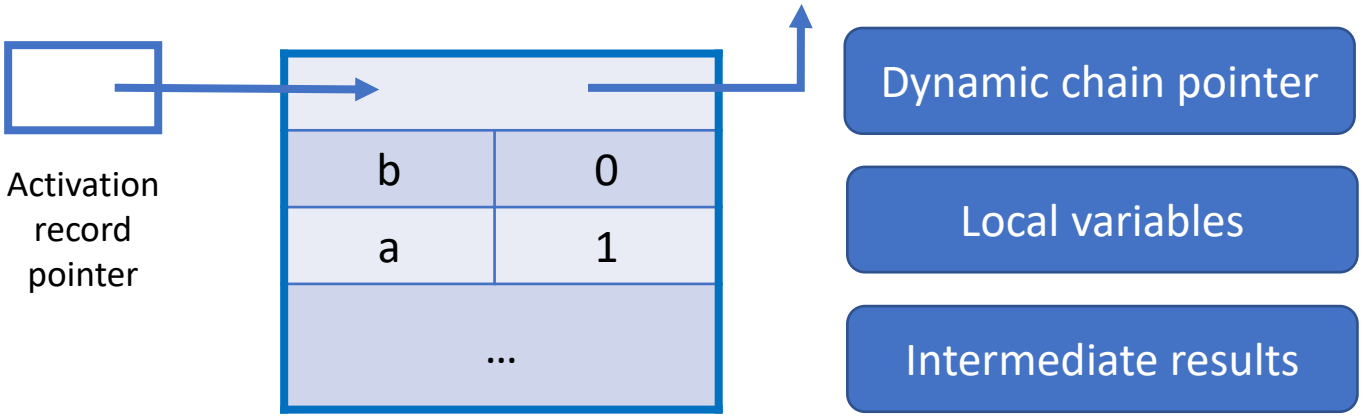


How does an activation record look like …

- … for an in-line block?
- … for a procedure?

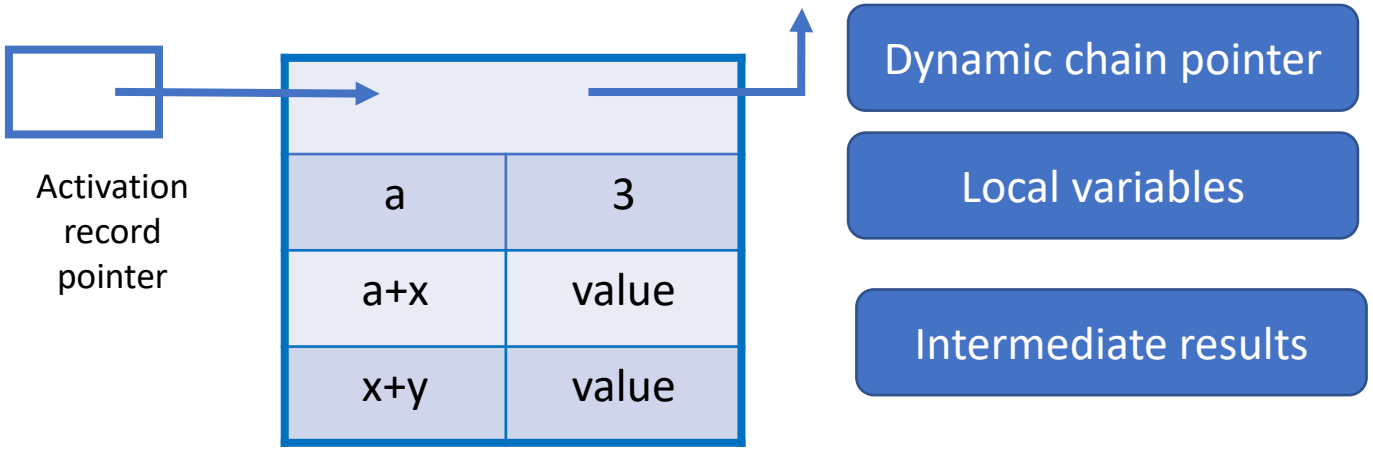
Activation record for in-line blocks

- This record consists of three parts:
  - Intermediate results
  - Local variables
  - Pointer for the dynamic chain



An example

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



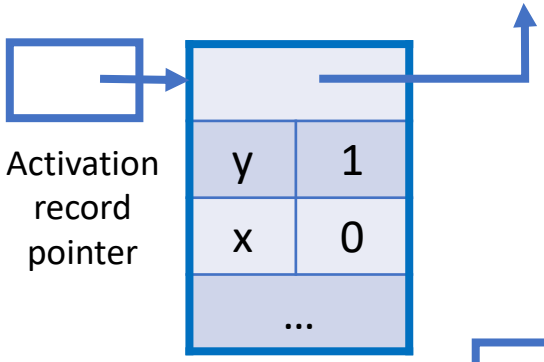
How does the activation record for in-line blocks work?

- Dynamic link (or control link): Pointer to previous record on the stack
- On entry to block: Push of the activation record
  - Dynamic link of the new record set
  - Activation record pointer set to the new record
- On exit from block: Pop of the activation record
  - Set the activation record pointer to the activation record on top of stack
- Dynamic chain: set of links implemented by these pointers

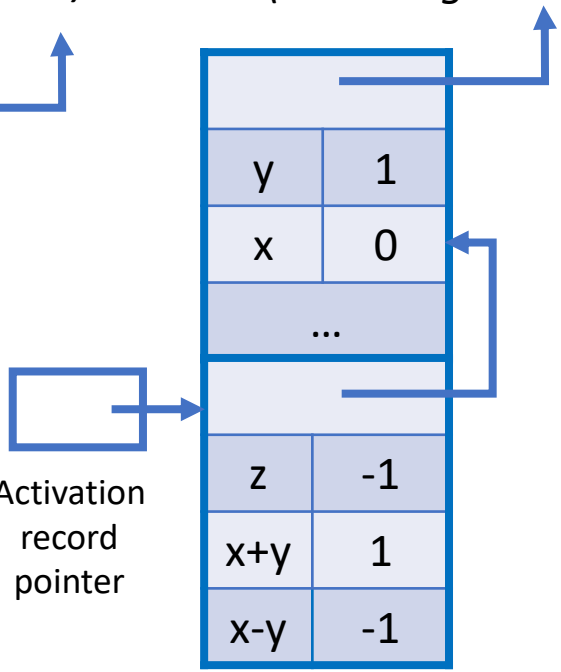
Example

```
{int x=0;
 int y=x+1;
  {
    int z=(x+y)*(x-y);
  };
};
```

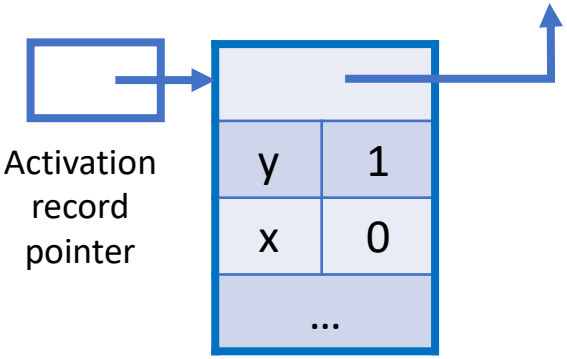
After entering external block (and assignments)



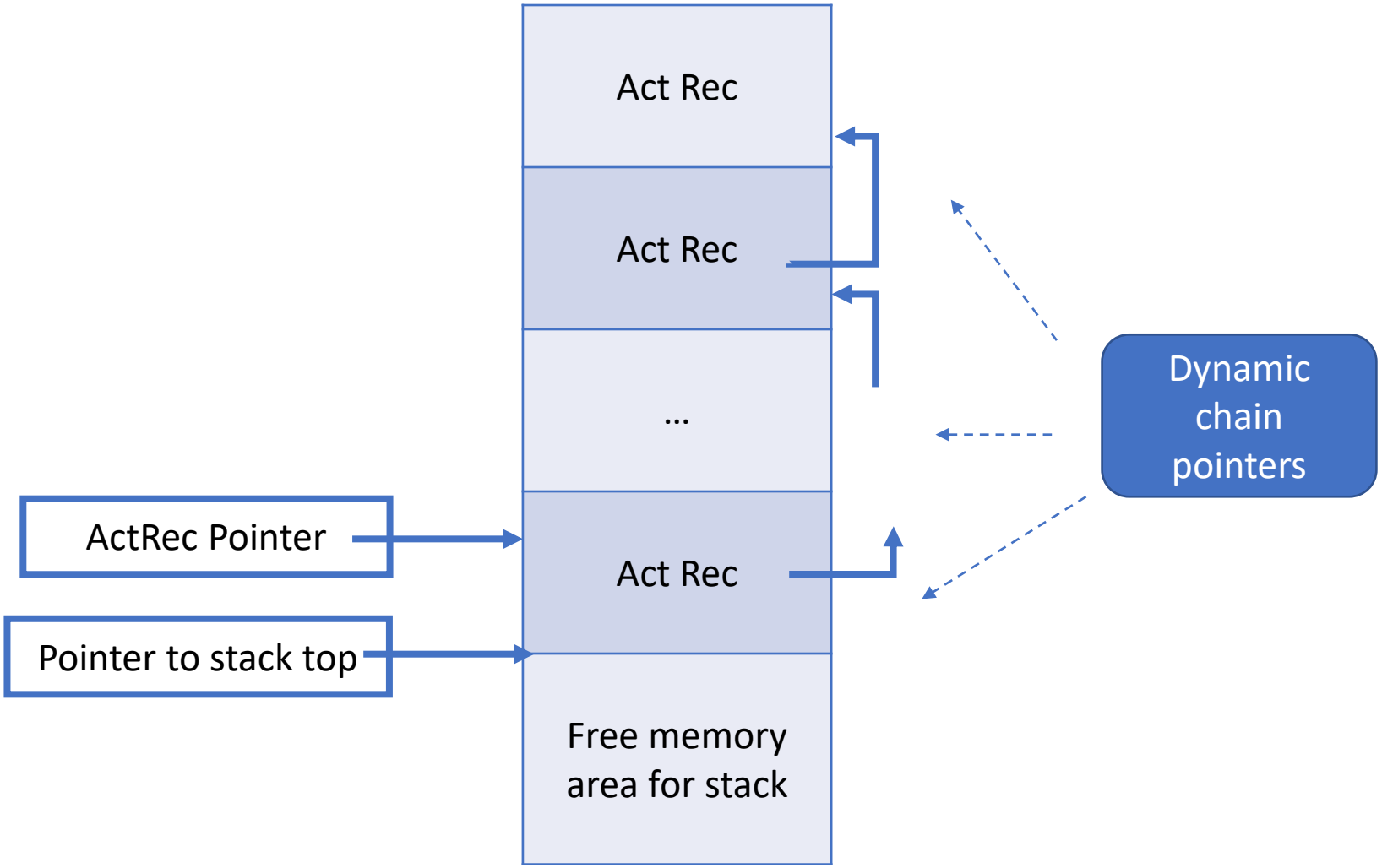
After entering inner block (and assignments)



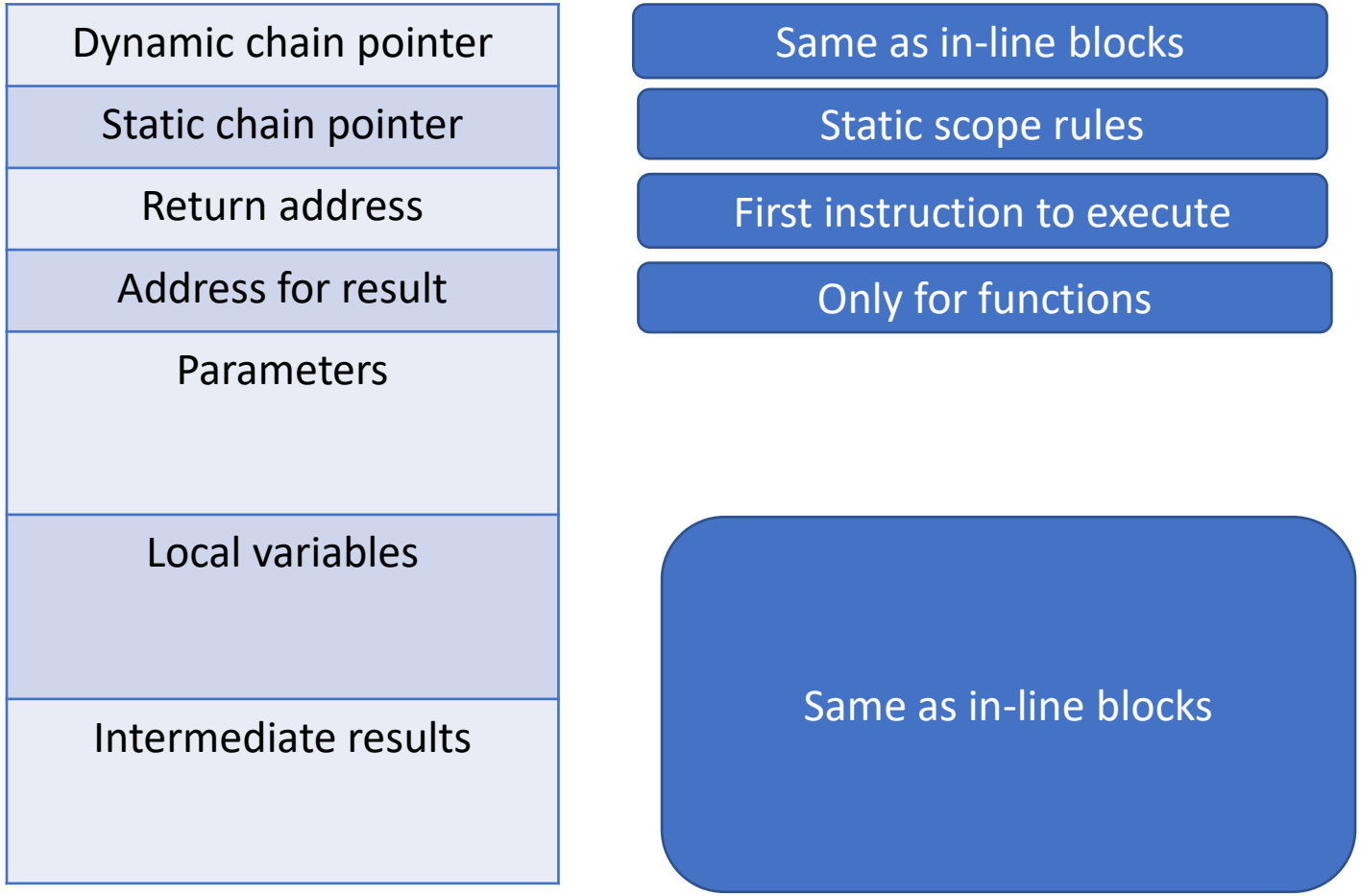
After exiting inner block (and assignment)



Management of the stack

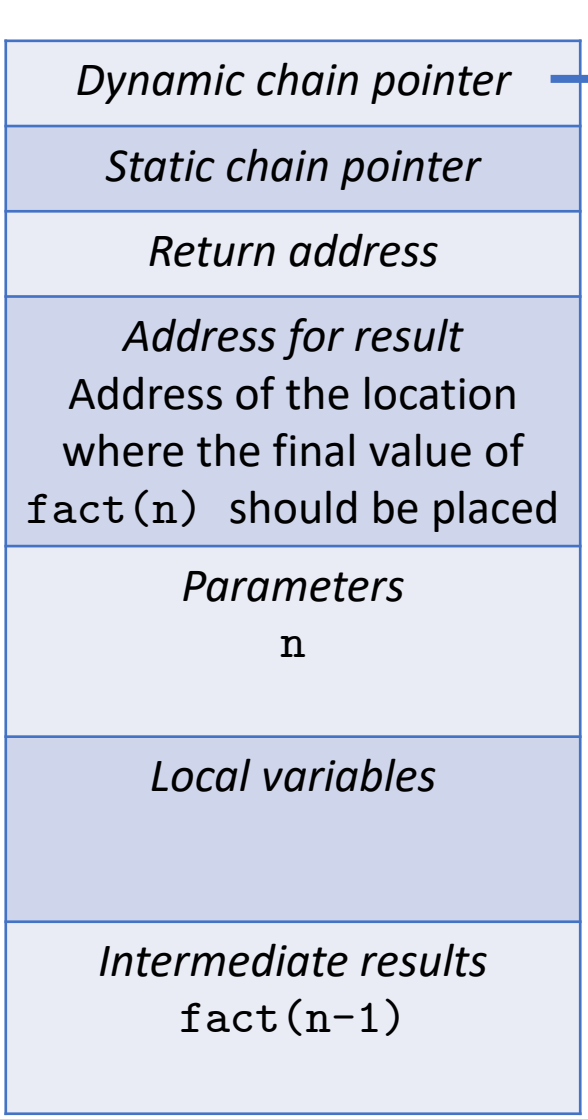


Activation records for procedures



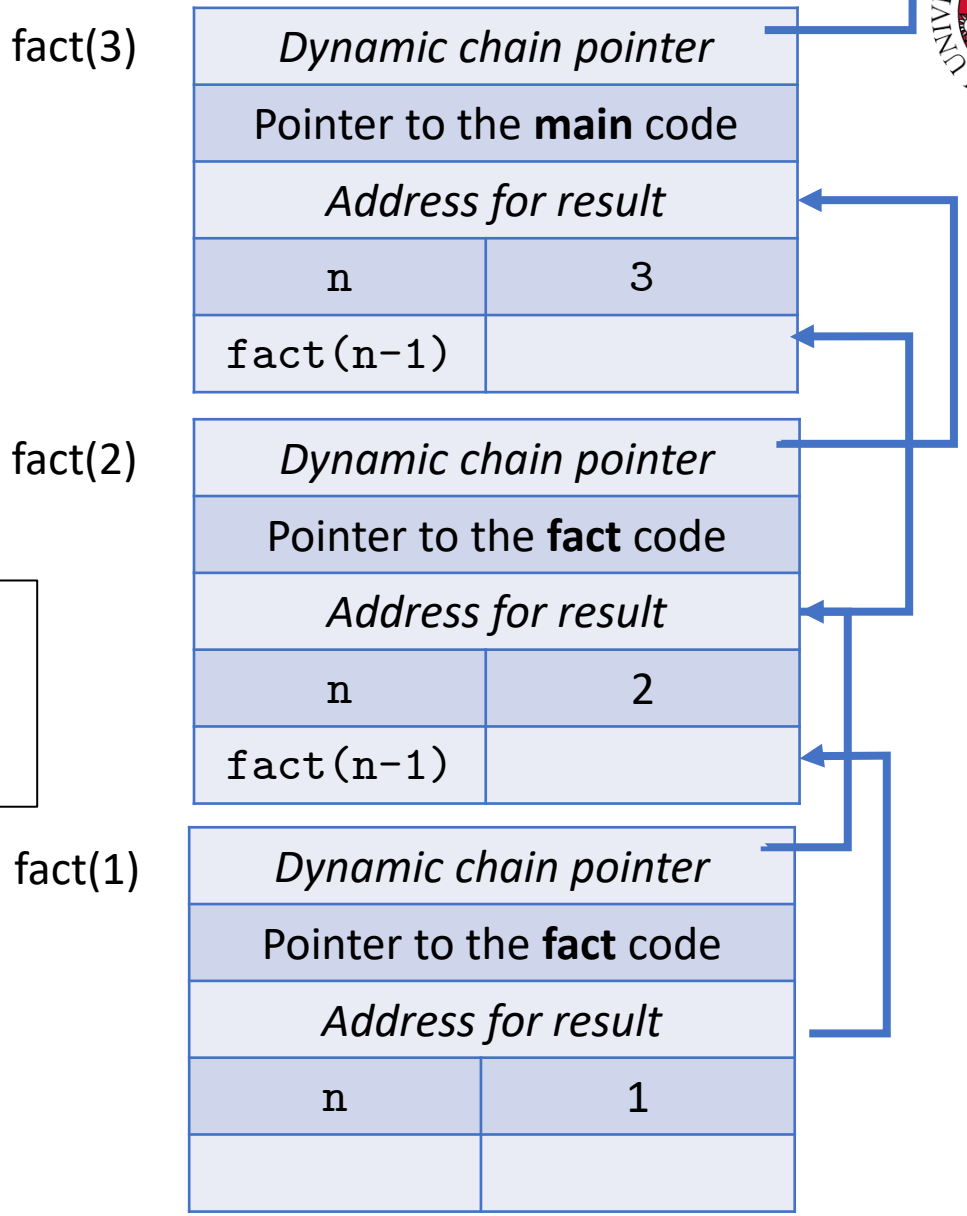
Example

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

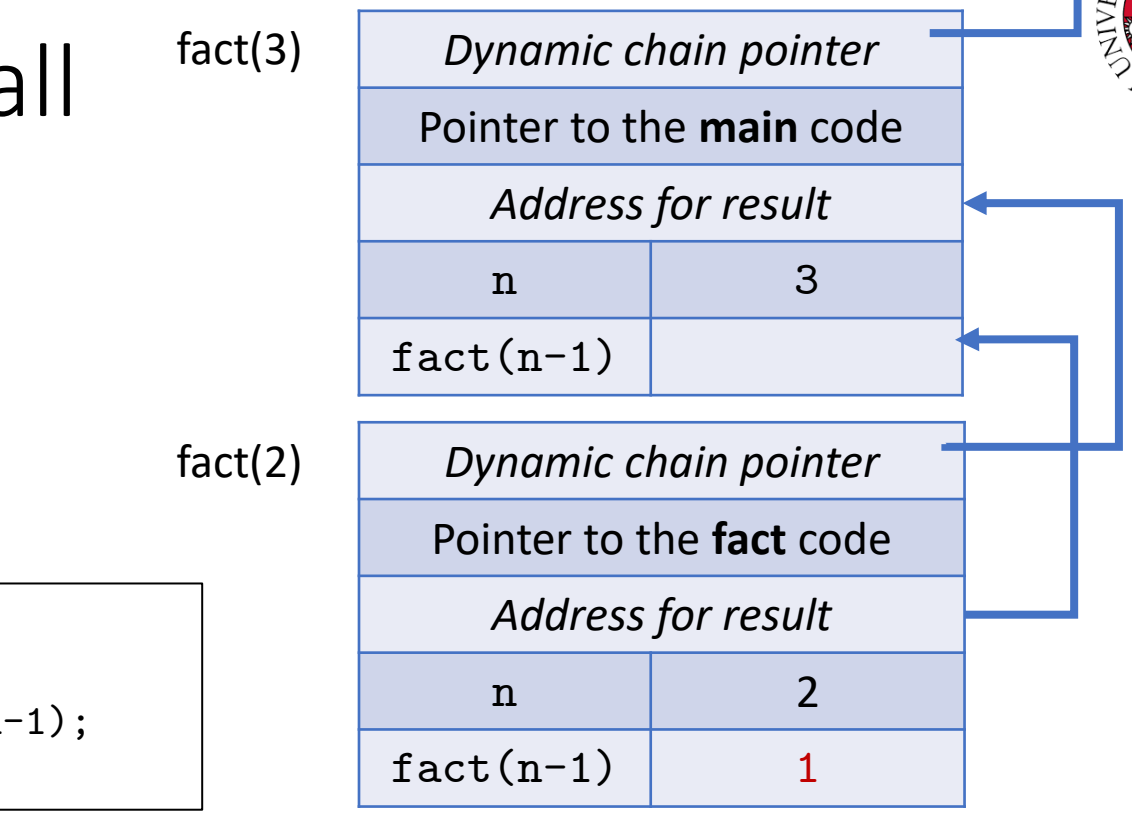


Example: recursive call fact(3)

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



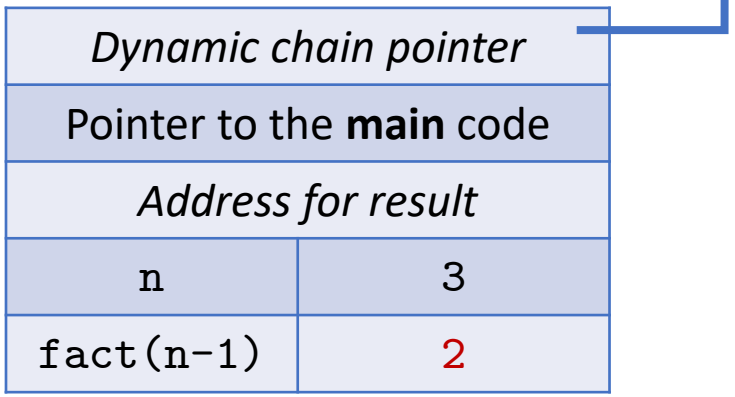
Example: recursive call fact(3)




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

Example: recursive call fact(3)

fact(3)




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

Stack management

- Stack management is performed by the caller and the callee
- Implementation with a stack uses:
  - Sequence of call (caller)
  - Prologue (callee)
  - Epilogue (callee)
  - Sequence of return (caller)
- When the block is called, they have to:
  - modify the program counter, allocate the new activation record on the stack, modify the activation record pointer, pass parameters
- When the block has been executed they have to:
  - update the program counter, return values in case of functions, deallocate the stack space, update the pointers in the stack

Dynamic allocation with a stack

- The pointer of the activation record points to the activation record of the active block
- The address of the activation record is not known at compile time
- The information in the activation record is accessible via its offset
  - The address is the activation record address plus the offset
  - The offset is determined statically

In practice

- In many languages, anonymous blocks can be implemented without a stack
- The compiler can analyze all the nested blocks
  - Space can be allocated for all variables
  - This may waste some memory
  - But no loss of efficiency due to stack management

Stack could be not enough

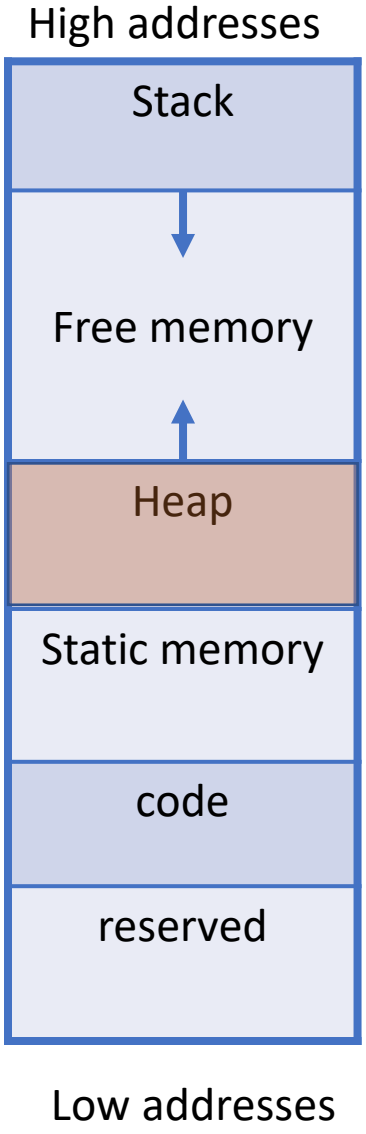
- In case of languages with commands for memory allocation
- Given that the memory deallocation operations are performed in the same order as allocations (first p, then q), the memory cannot be allocated in LIFO order.

```
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 allocation using a heap

Heap

- Heap: Region of memory in which blocks (and sub-blocks) can be allocated and deallocated at arbitrary moments
- This is necessary when the language allows:
  - Explicit allocation of memory at runtime
  - Objects of varying size (e.g., arrays of variable size)
  - Objects whose lifetime is not LIFO
- Heap management is nontrivial
  - Efficient allocation of space: Avoiding fragmentation
  - Speed of access

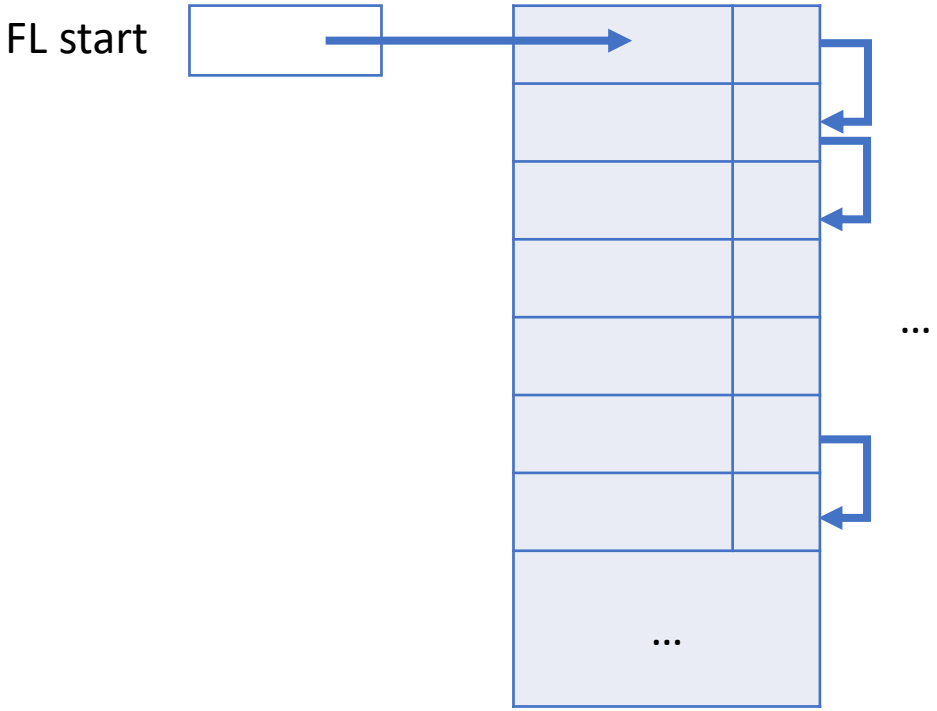


Two types of heap management methods

- We can distinguish two heap management methods:
  - Blocks of fixed size
  - Blocks of variable size

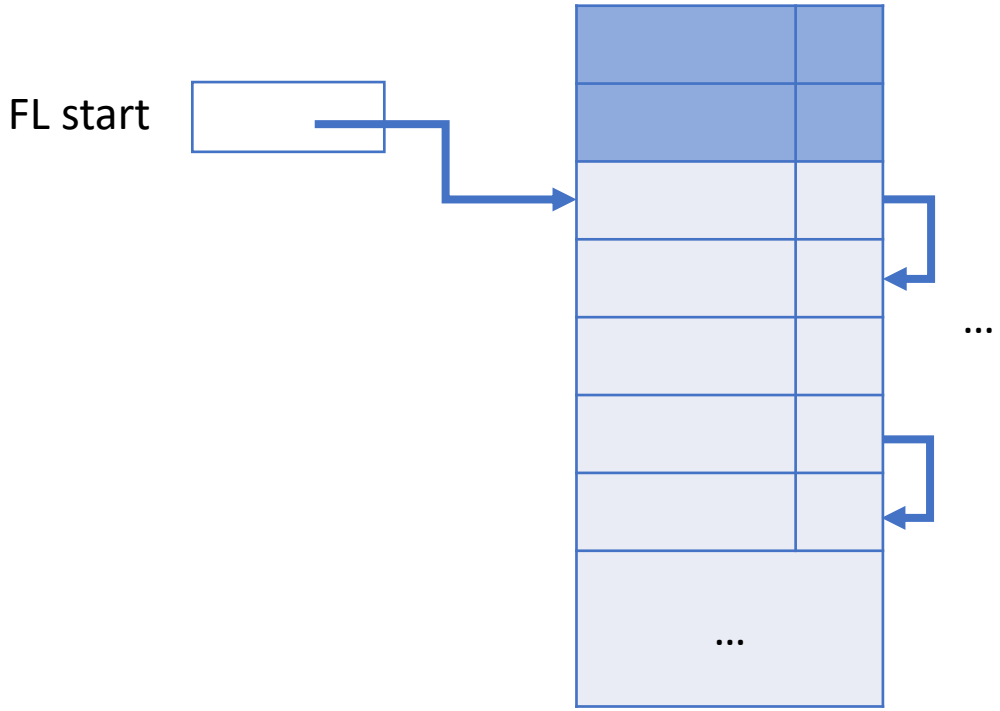
Blocks of fixed size

- The heap is divided into blocks of (quite small) fixed size
- At the start: all of these blocks are linked in a free list



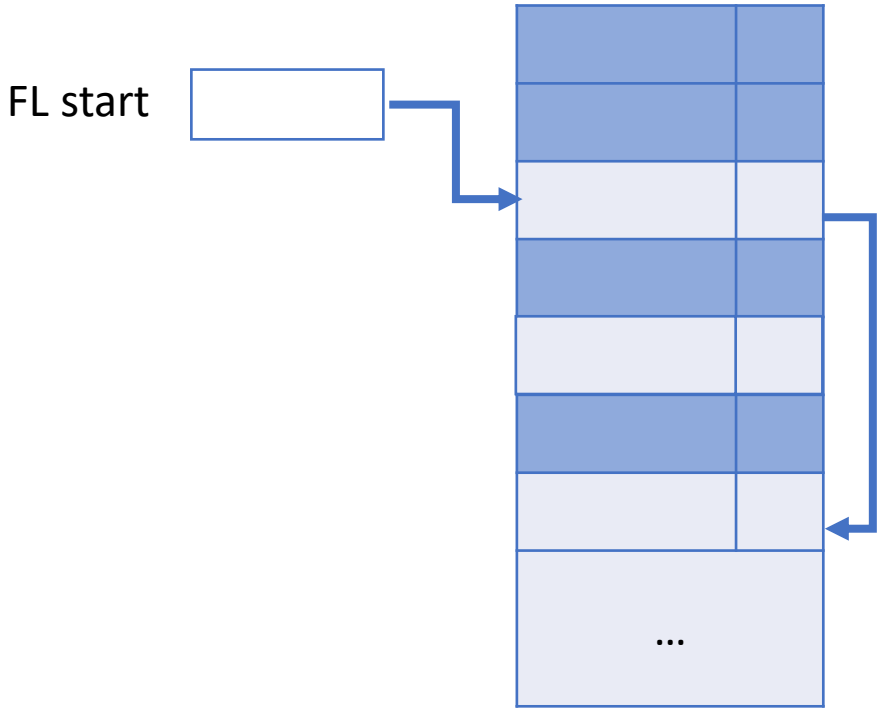
Blocks of fixed size

- At runtime, when an operation requires the allocation of a memory block (e.g., `malloc`) the first element is removed from the list
- Allocation of one or more contiguous blocks



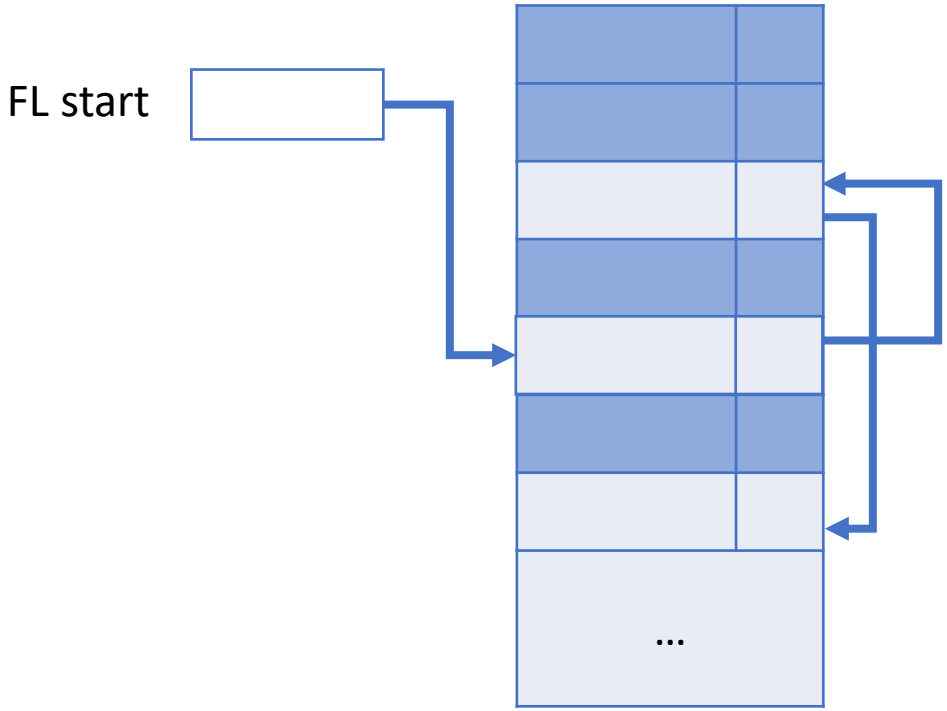
Allocation and deallocation

- Deallocation (`free`): Restore the freed blocks to the free list (connected to the head)



Allocation and deallocation

- Deallocation (`free`): Restore the freed blocks to the free list (connected to the head)



Blocks of variable size

- What if we need to allocate a block of variable dimension (e.g., an array of variable dimension)?
- Heap-based management system using variable-length blocks

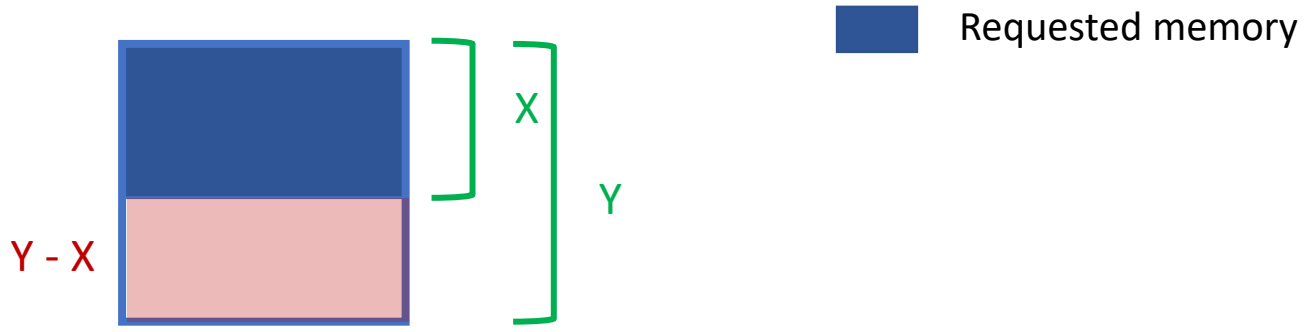
- Heap initially contains a single block
- Allocation: Find a free block of the appropriate size
- Deallocation: Restore the block to the free list

Problems with the heap

- We need to take into account efficiency and memory occupation
- The operations must be efficient
- Avoid wasting memory
  - Internal fragmentation
  - External fragmentation

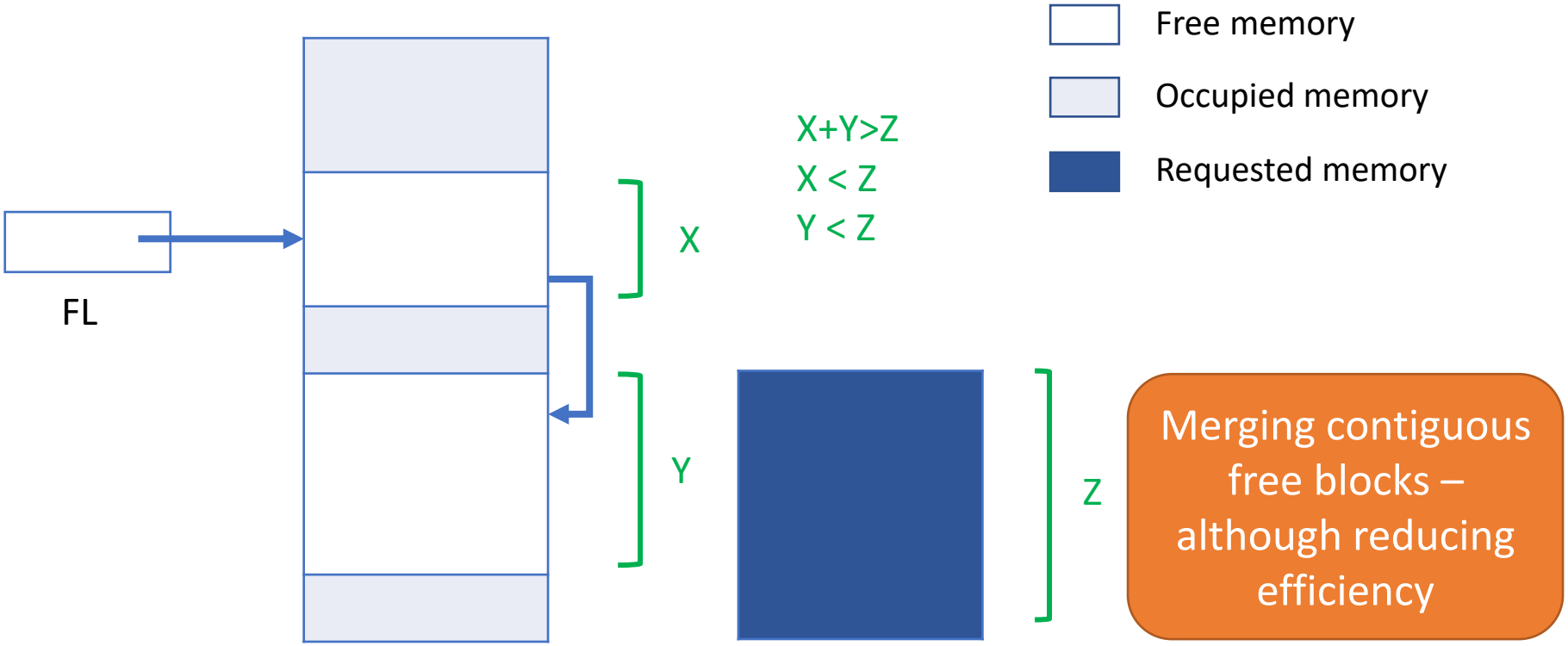
Internal Fragmentation

- The needed space is of size X
  - A block of size Y > X is allocated
  - Space of size Y - X is wasted



External Fragmentation

- The needed space is available but not usable, as it is broken up into pieces that are too small



How to deal with fragmentation?

- Single free list
  - Direct use of the free list
  - Free memory compaction
- Multiple free lists
  - Buddy system
  - Fibonacci heap

Single free list

Single free list

- At the start: a single block, whose size is the size of the heap
- At each allocation request of size n, the first n words of memory are allocated and the pointer to the free memory is incremented by n; deallocated block collected on a free list.
- When the heap is over, we can use two strategies:
  - Direct use of free list: when a block is deallocated adjacent blocks are checked and if free, they are compacted
  - Free memory compaction: when the end of the space is reached, all blocks active moved to the end and free memory is contiguous

Direct use of the free list

- At each allocation request: find a block of the appropriate size
  - First fit: First block that is large enough
    - Faster, worse use of memory
  - Best fit: Find the smallest block that is large enough
    - Slower, better use of memory
- If the chosen block is much larger than needed, it is divided into two pieces, and the unused one is added to the free list
- When a block is deallocated adjacent blocks are checked and if free, they are compacted

Free memory compaction

- When the end of the space is reached, all blocks active moved to the end, while free memory is all contiguous and the heap pointer points to the start of such a single block – only if blocks of allocated memory can be moved.

Multiple free list

Multiple Free lists

- With a single list: allocation is linear in the number of free blocks
- Multiple lists: different free lists for blocks of different sizes, managed for instance through
  - Buddy system
  - Fibonacci heap

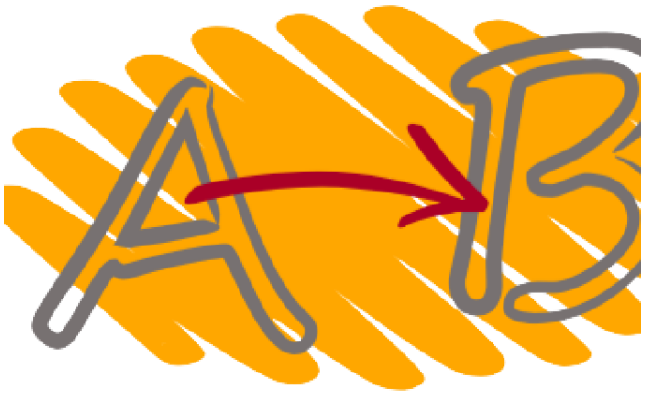
Dynamic size of multiple lists

- Buddy system: k lists, with the k^{th} list having blocks of size 2^k
  - If size 2^k is requested, but not available in the k^{th} list, allocate a block of size 2^{k+1} divided into 2: half is used, the other half is appended to the free list of the list 2^k
  - If a block of size 2^k is deallocated, unite it with its other half, if this is available
- Fibonacci heap: similar, but uses Fibonacci numbers as block size (that grows more slowly than 2^k)
  - Computation is a bit less efficient
  - Experiments show better use of memory (lower internal fragmentation)

Scoping rules

Implementation of scoping rules

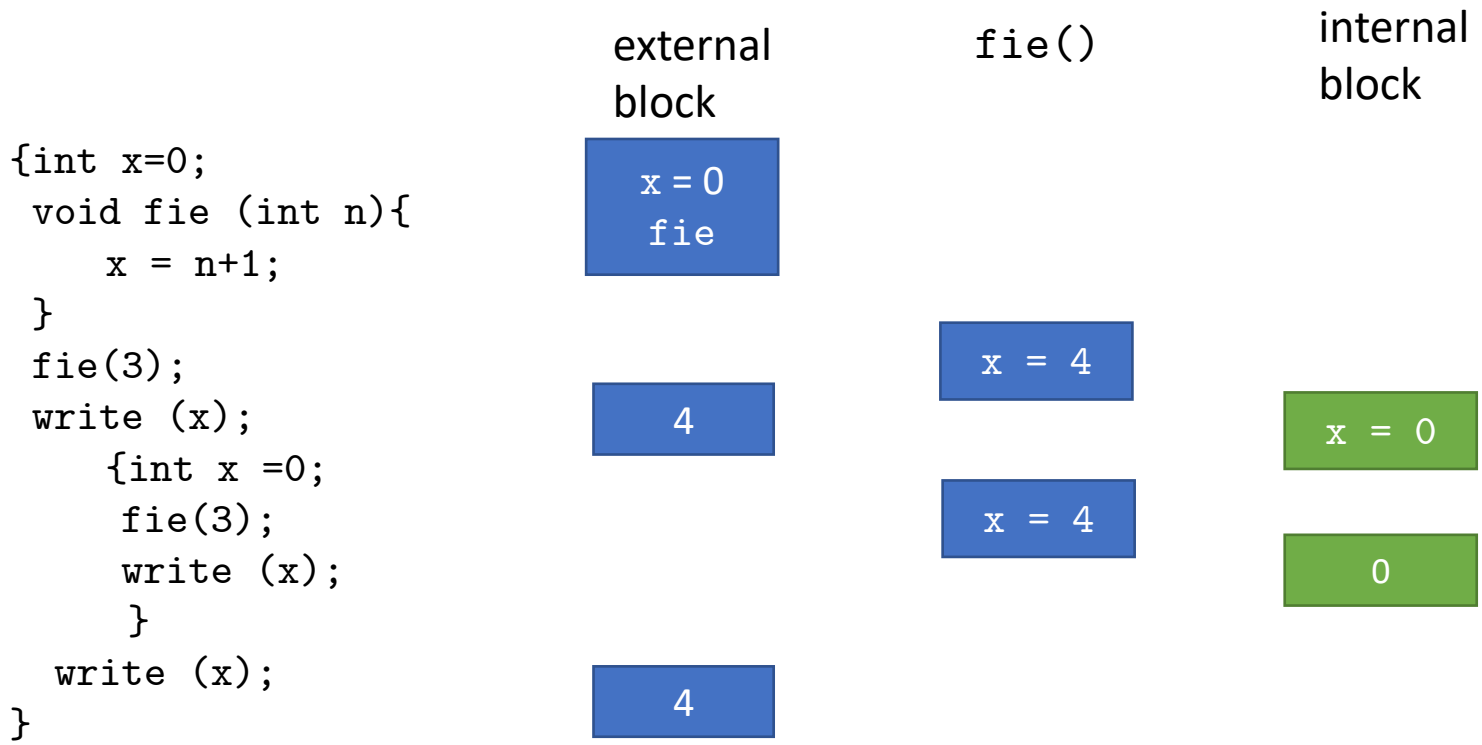
- For resolving non-local references we need to find the activation record that corresponds to the right block in which the name has been declared
- The order in which to examine the activation records depends on the type of scoping considered
- Static scoping
  - Static chain
  - Display
- Dynamic scoping
  - Association-list
  - Central table of the environment



Static (lexical) scoping

Static scoping

- In static scoping a non-local name is resolved in the block that textually includes it



How to determine the correct association?

- In static scoping the order in which the activation records have to be consulted for resolving non-local references is not the one of the stack
- The first activation record within which to look is defined by the textual structure of the program

An example

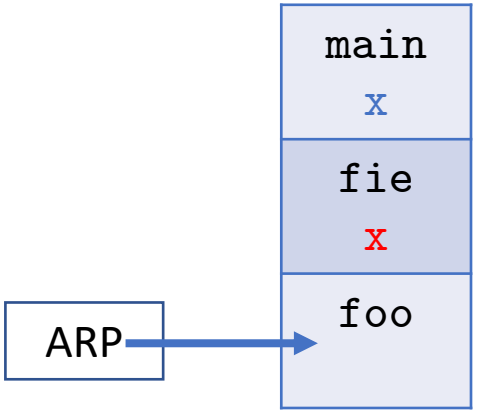
```
{ int x=10;

  void foo(){
    x++;
  }

  void fie() {
    int x=0;
    foo();
  }

  fie();
}
```

- `foo` is called inside `fie`
- `foo` always accesses the same variable `x`
- `x` is stored in a certain activation record (here `main`)
- the access record for `foo` is at the top of the stack



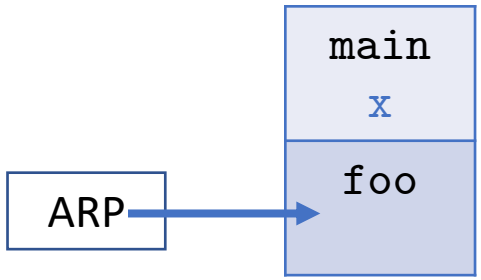
Another example

```
{ int x=10;

  void foo(){
    x++;
  }

 foo();
}
```

- `foo` is called inside `main`
- `foo` always accesses the same variable `x`
- `x` is stored in a certain activation record (here `main`)
- the access record for `foo` is at the top of the stack



In both cases

- Determine first the activation record where x is located
- Access x via the offset from this activation record

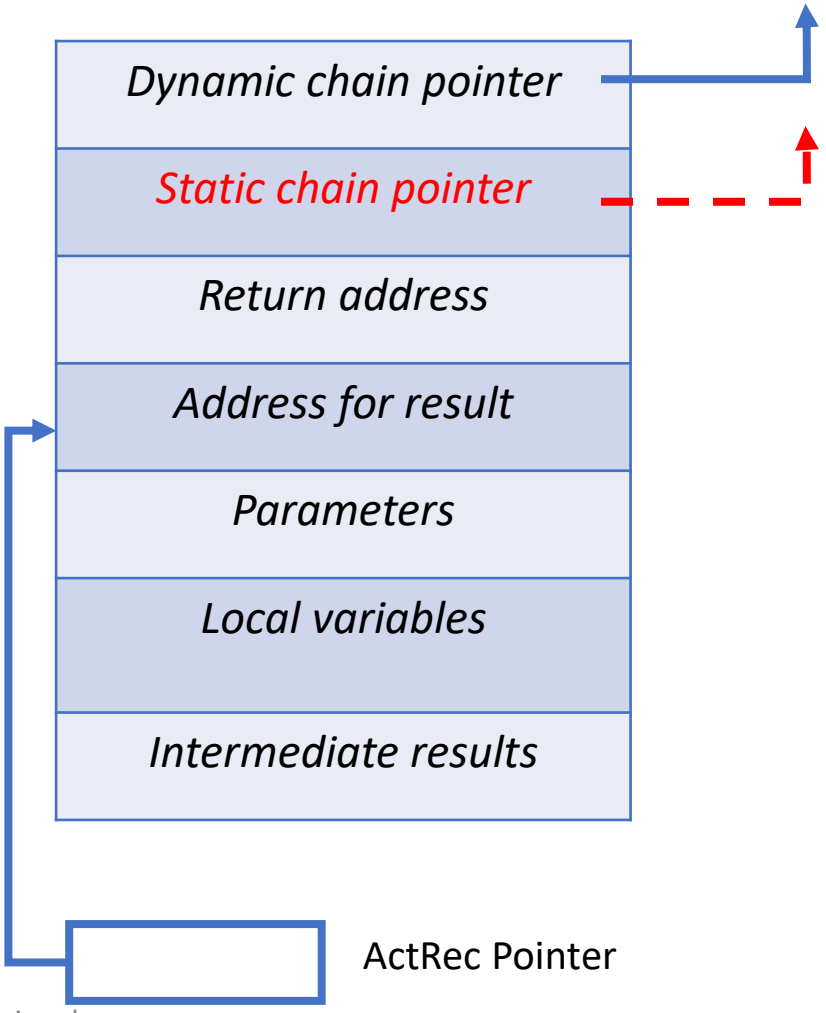
How to deal with static scoping?

- Two techniques:
  - Static chaining
  - Display

Static chaining

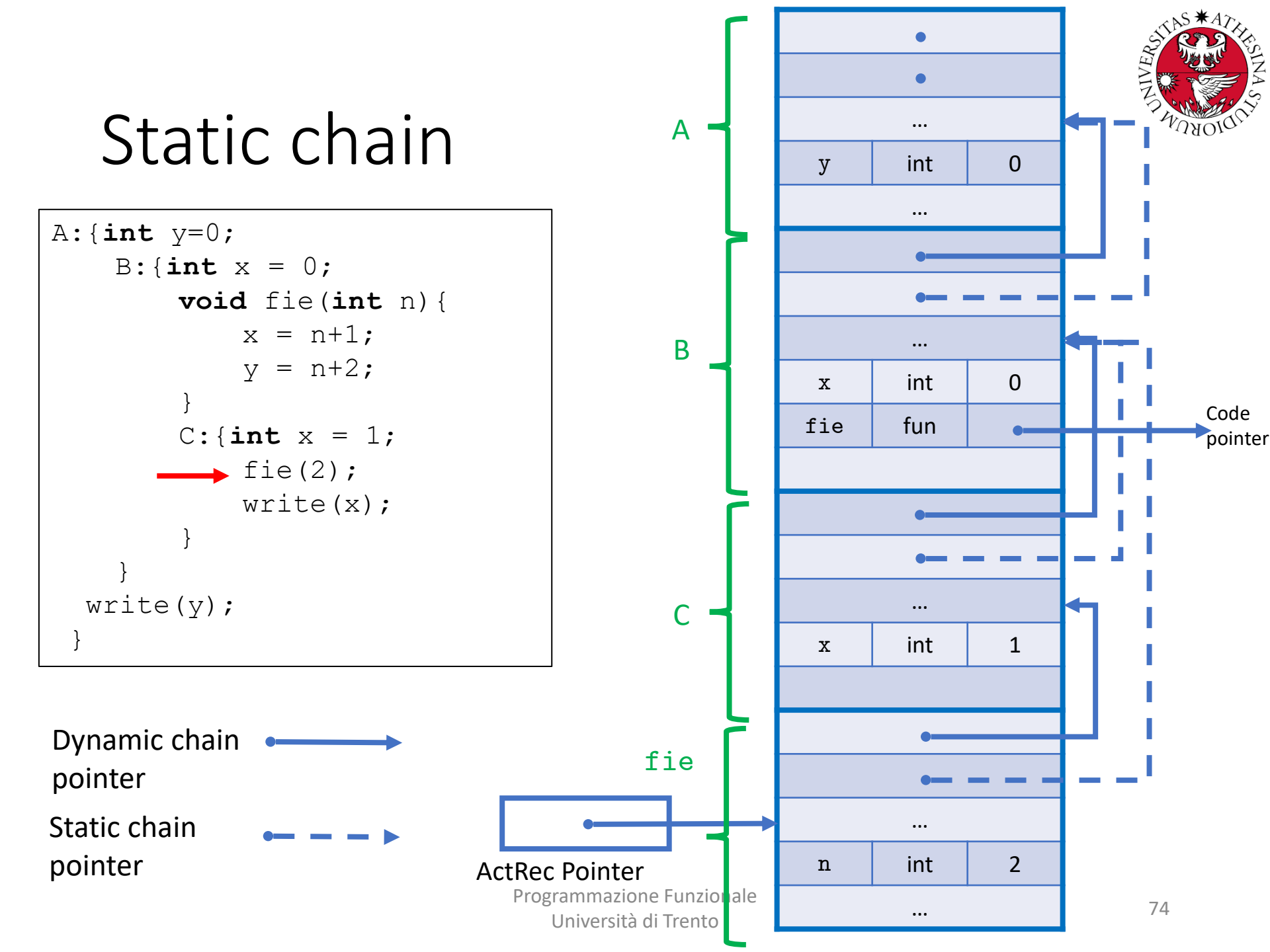
Activation record for static scoping

- Dynamic chain pointer (already seen)
- Static chain pointer: Pointer to the activation record of the block that immediately contains the text of the current block
- A dynamic link depends on the execution sequence of the program
- A static link depends on the static nesting of the declarations of the procedure



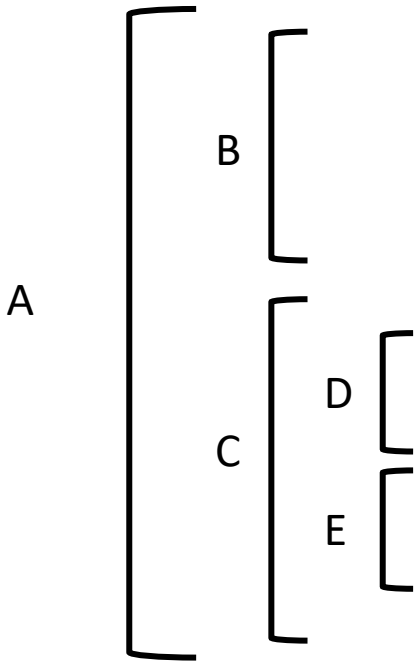
Static chain

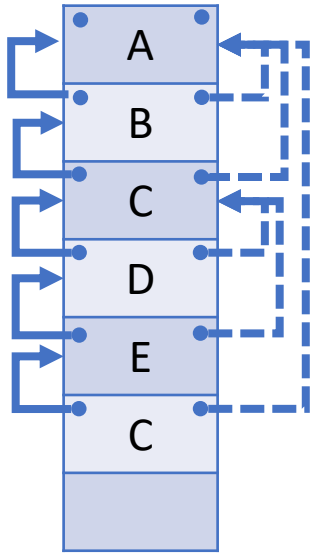
```
A:{int y=0;
    B:{int x = 0;
        void fie(int n){
            x = n+1;
            y = n+2;
        }
        C:{int x = 1;
            fie(2);
            write(x);
        }
    }
  write(y);
 }
```



Example: structure of blocks

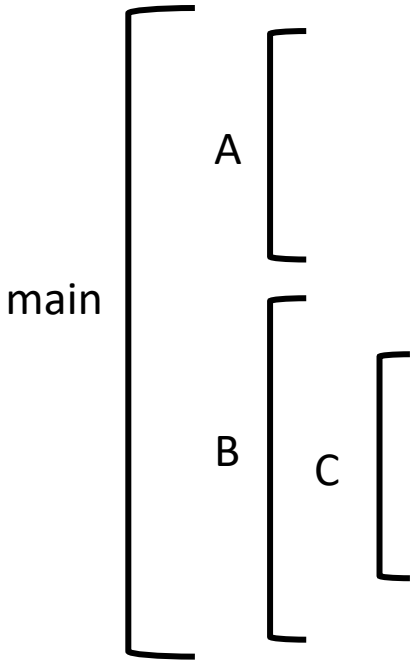
- Sequence of calls: A, B, C, D, E, C

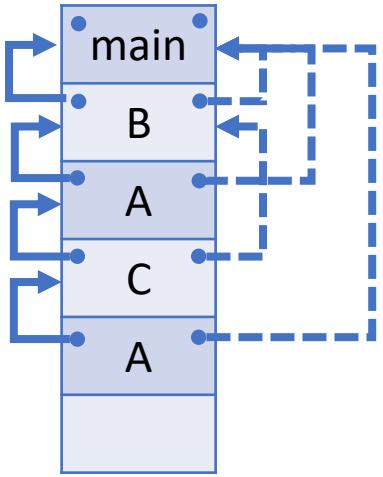




Another example

```
{ int x;
  void A() { x=x+1;}
  void B() { int x;
    void C() (int y){
        int x;
        x=y+2;
        A();
    }
    x=0; A(); C(3);
  }
  x=10;
  B();
}
```





Static chain

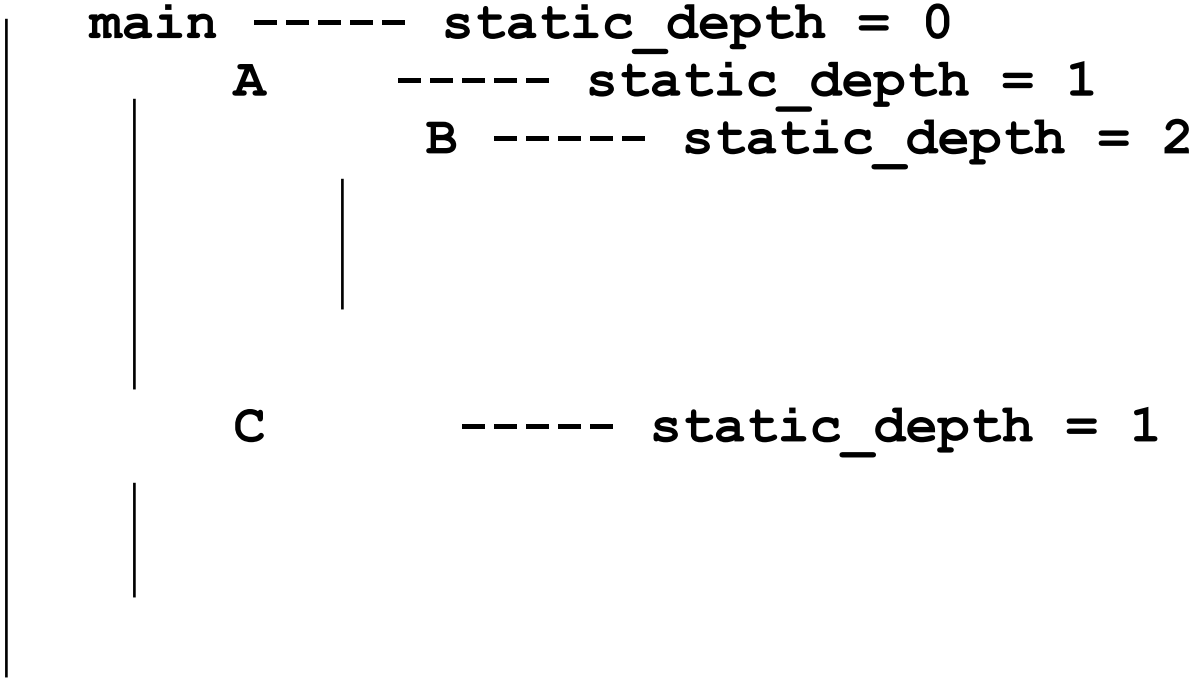
- A static chain is a chain of static links that connects certain activation record instances
- The static link in an activation record instance for subprogram A points to one of the activation record instances of A's static parent
- The static chain from an activation record instance connects it to all of its static ancestors
- To find the declaration for a reference to a nonlocal variable:
  - You could chase the static chain until the activation record instance that has the variable is found

How many steps in the static chain?

- Given a variable, at compilation time we can compute
  - The chain offset
  - The local offset
- At runtime, we can use the chain and local offset to reach the definition of the variable

Static depth

- An integer associated with a static scope whose value is the depth of nesting of that scope



Static chain

- The chain offset of a nonlocal variable is the difference between the static depth of the usage and that of the scope where it is declared
- A reference can be represented by the pair:

  (chain_offset, local_offset)

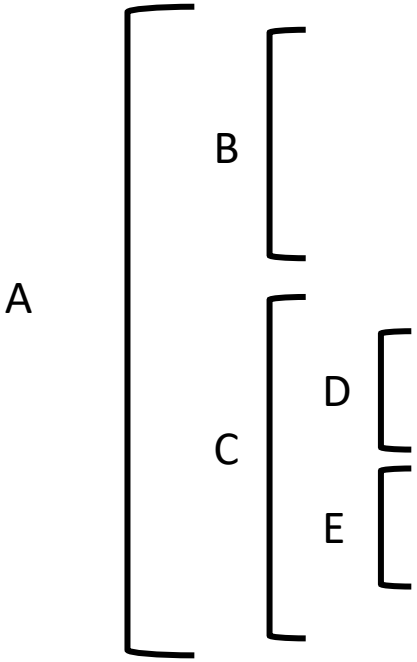
  where local_offset is the offset in the activation record of the variable being referenced

At compilation time

- For each non-local variable `x` in a procedure `P`, it can be determined
  - the `P`'s ancestor `Q` that contains the declaration of `x`
  - the number of static links to chase to find the activation record of `Q` that contains `x` (chain offset)
  - the local offset of `x` in the `Q`'s activation record

At runtime

- In order to access the non-local reference `x` from `P`
  - access the last active activation record of `Q` (using the static chain pointer and the chain offset)
  - Use the local offset starting from the address of the activation record of `Q`



- For instance, in this case, the procedure D can see the declarations in the main program, in A, C and D, each in k steps:
  (PP, 3); (A,2); (C,1); (D,0)
- k determines how many steps to perform along the static chain (chain offset)

How is the static link of the call determined?

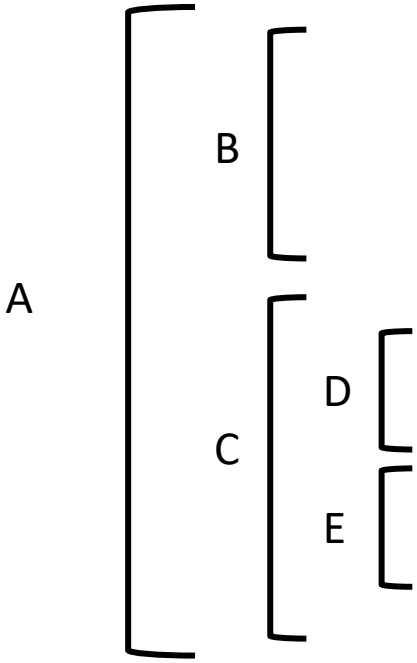
- In charge to the sequence call, prologue and epilogue code
- Usually, when a new block is entered, the caller should determine the static chain pointer and pass it to the callee
- The caller has the following information:
  - Static nesting of blocks determined by the compiler: if the caller is at nesting level m and the callee n, the distance between them is k=m-n-1
  - Its own activation record

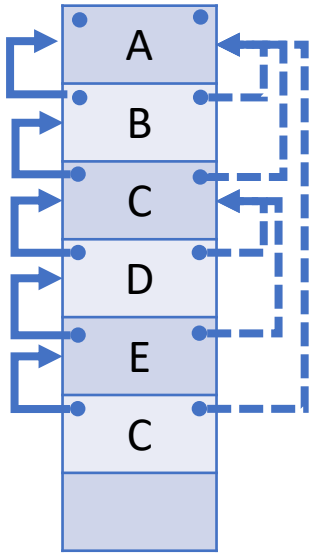
How can the caller determine the static pointer?

- When the caller C calls the callee P, we can have only two cases (according to the visibility rules):
  a. P is immediately included in C (callee declared inside the caller) - $k = 0$
  b. P is in a block at k steps from C (called routine external to the caller) - $k > 0$
     - For the visibility rules, P has to necessarily be located in an outer blocks which includes the caller's block.
     - The activation record of such an outer block should already be on the stack
- We hence have two cases:
  - If $k = 0$, C passes its own activation record pointer to P
  - If $k > 0$, C finds the pointer after k steps along the static chain

Example: how to determine the static pointer?

- Sequence of calls: A, B, C, D, E, C





$A \rightarrow B$ (k=0)
$B \rightarrow C$ (k=1)
$C \rightarrow D$ (k=0)
$D \rightarrow E$ (k=1)
$E \rightarrow C$ (k=2)

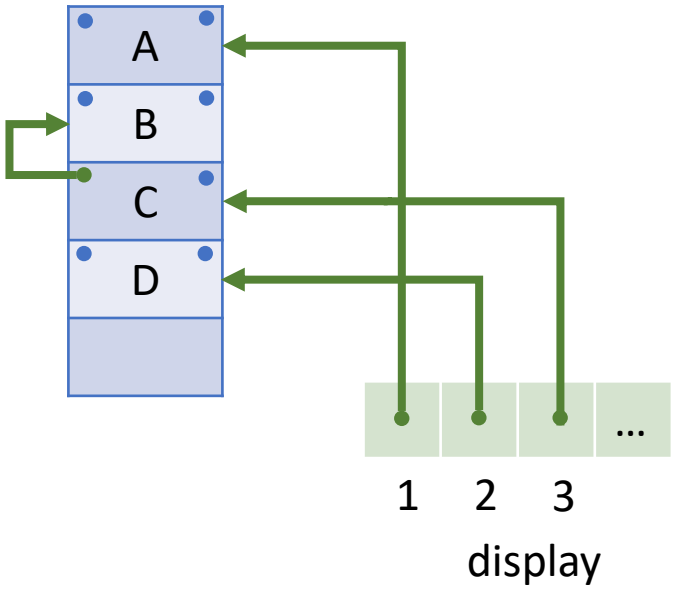
Display

The display

- Static scoping with static chain can be costly (to reach level k –> k accesses to memory)
- We can reduce the costs from scanning the chain to a constant using the display
- The display is a vector containing as many elements as the levels of block nesting in the program
- The k-th element of the vector contains the pointer to the activation record at nesting level k currently active.

The display

- The static chain is represented by an array, called the display
  - The k-th element of the display is a pointer to the activation record of the subprogram at nesting level k currently active
- If the display is in memory, a constant number of accesses (2) to memory is enough

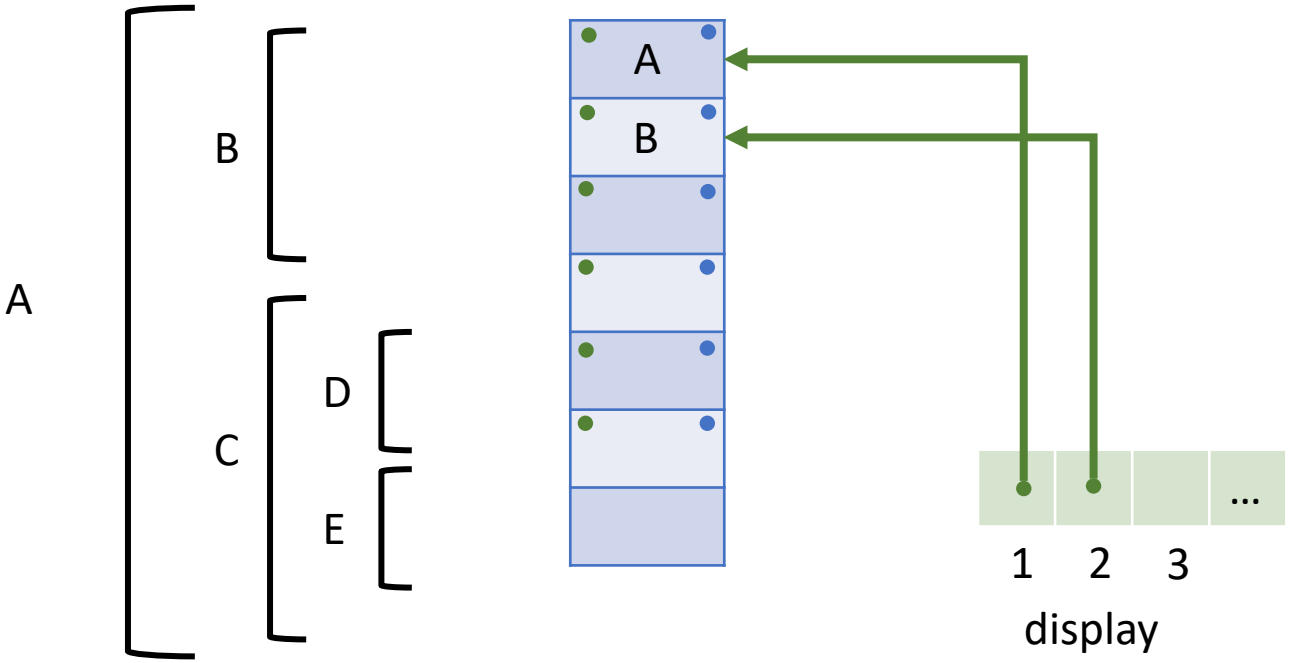


Maintaining the display

- When an environment is entered (at level k)
  - Save the old value at position k (if it exists)
  - Update the pointer stored in the vector
  - If the callee is at level n and the caller at level $m < n$, the active display is the one formed by the first m elements and the rest of the display is re-activated when the called routine ends.
  - If the caller is at level n and the callee at level n+1, we could still need the old values at level n+1
- At the exit of the environment
  - move the saved display pointer from the activation record back into the display at position k

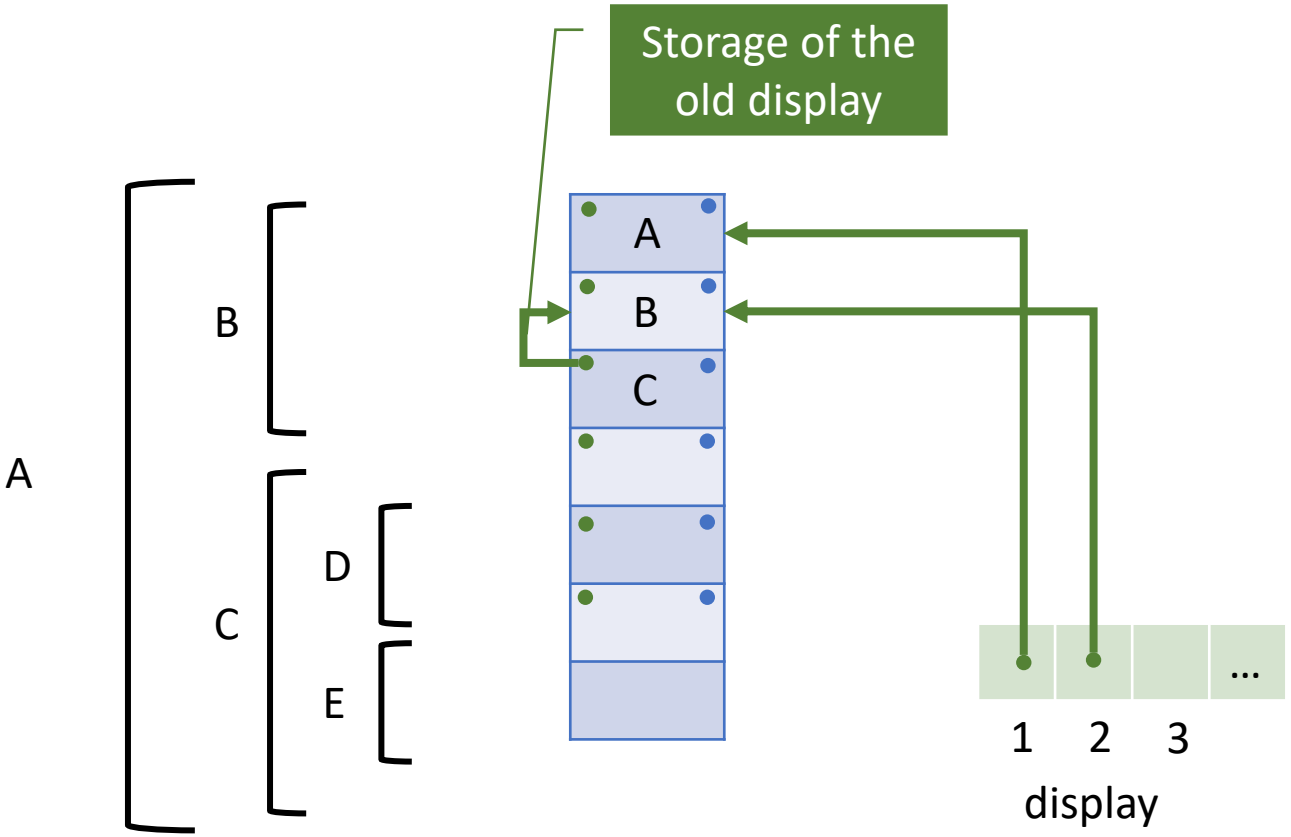
Example

- Sequence of calls: A, B, C, D, E, C



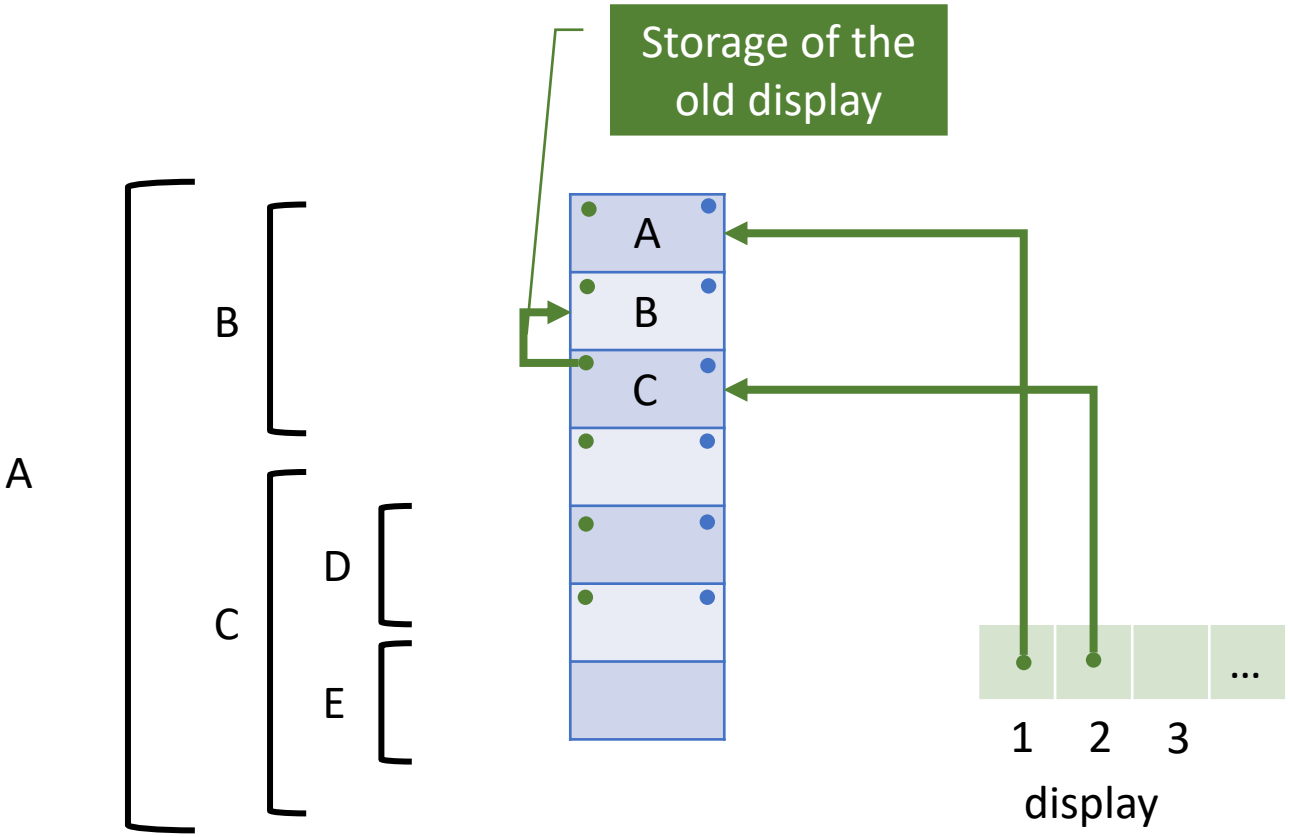
Example

- Sequence of calls: A, B, C, D, E, C



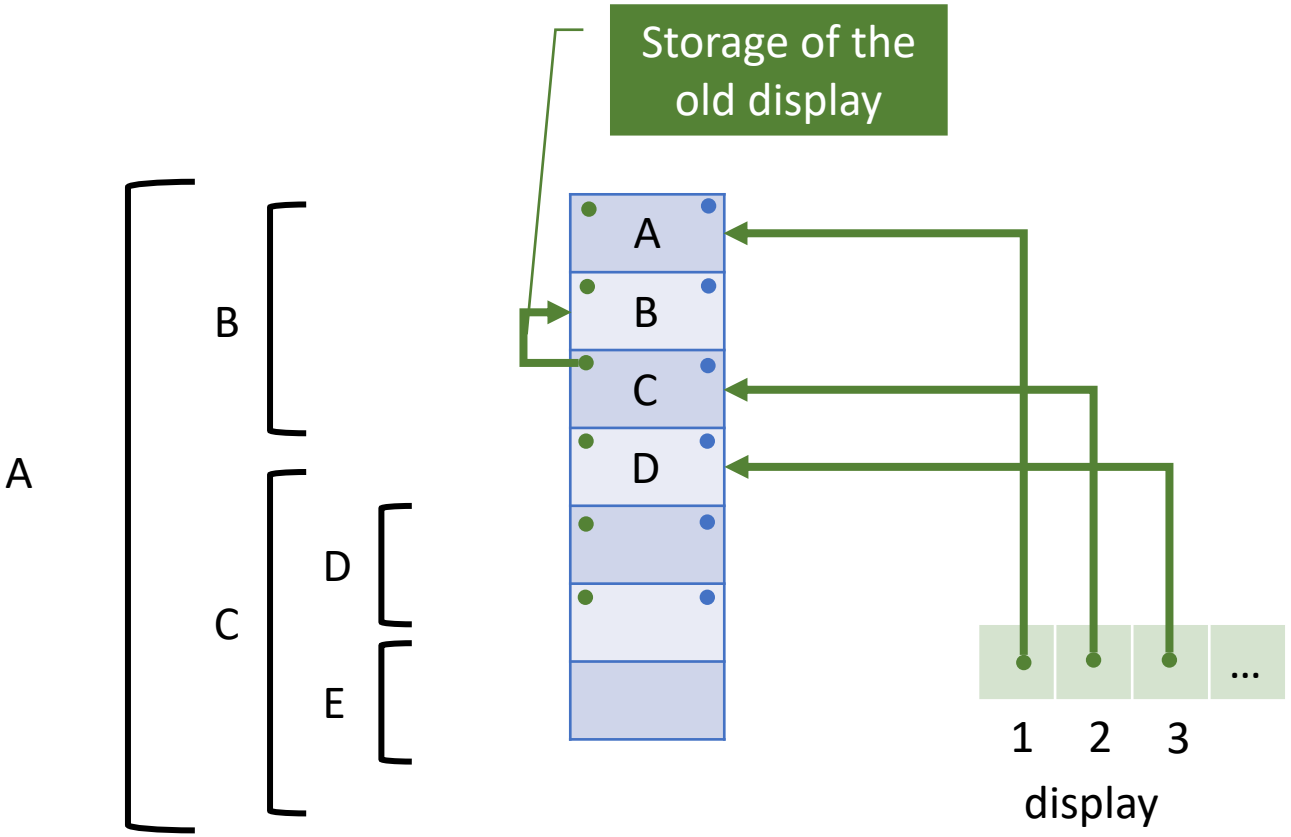
Example

- Sequence of calls: A, B, C, D, E, C



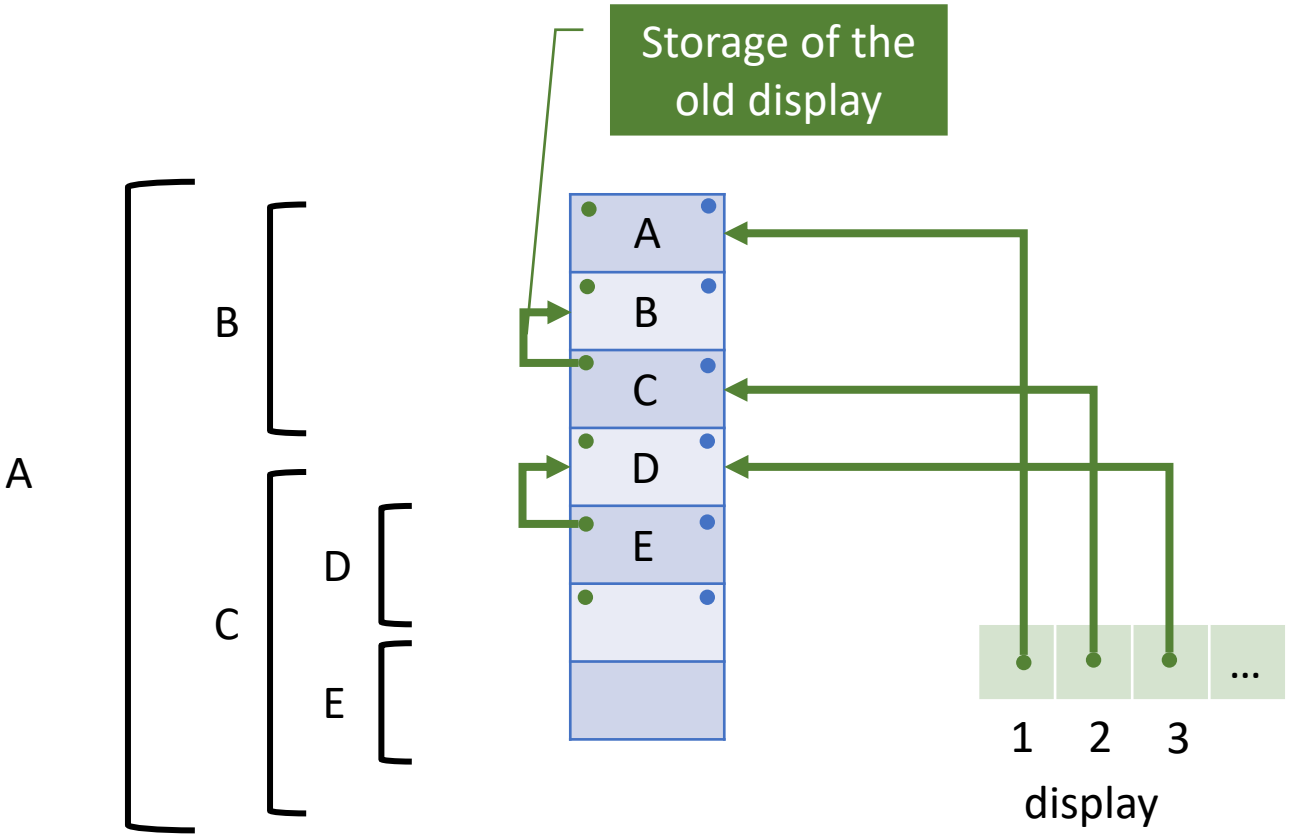
Example

- Sequence of calls: A, B, C, D, E, C



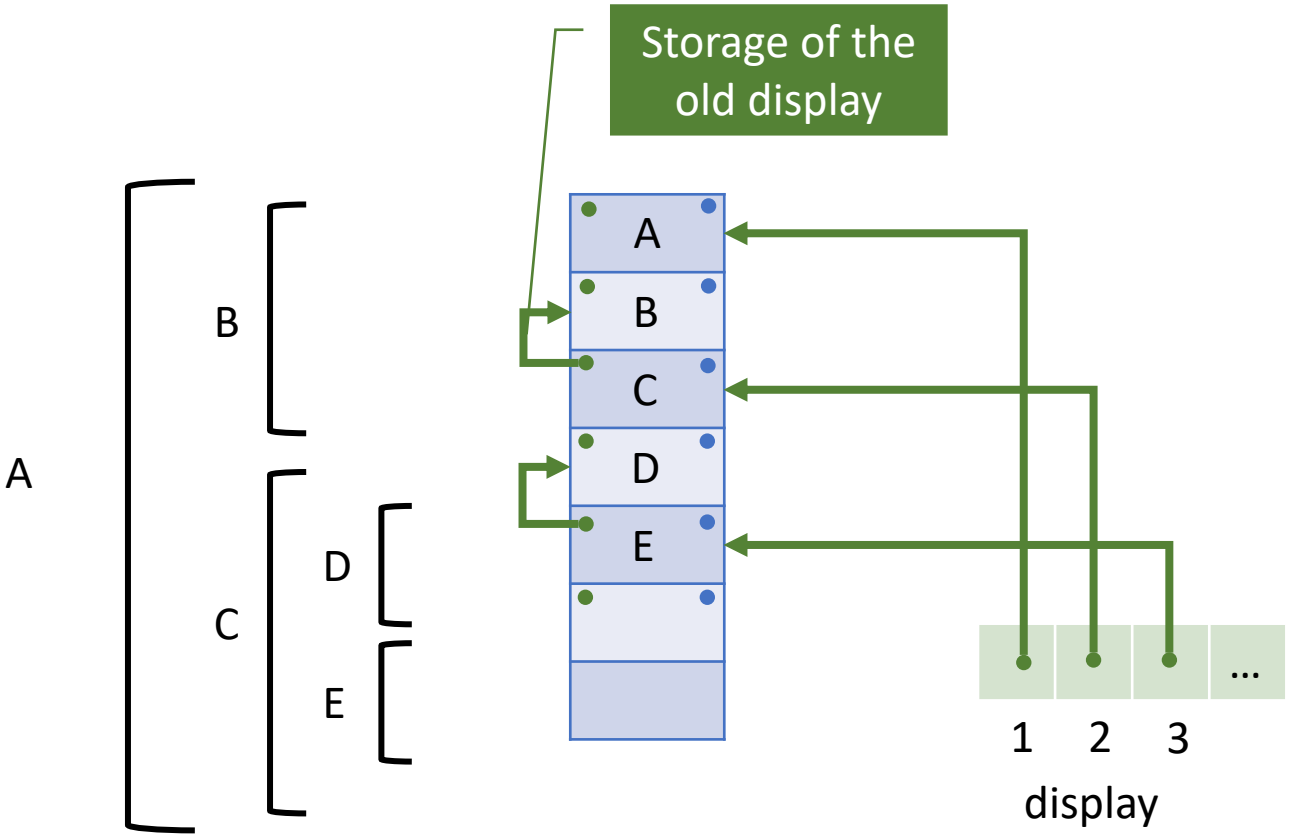
Example

- Sequence of calls: A, B, C, D, E, C



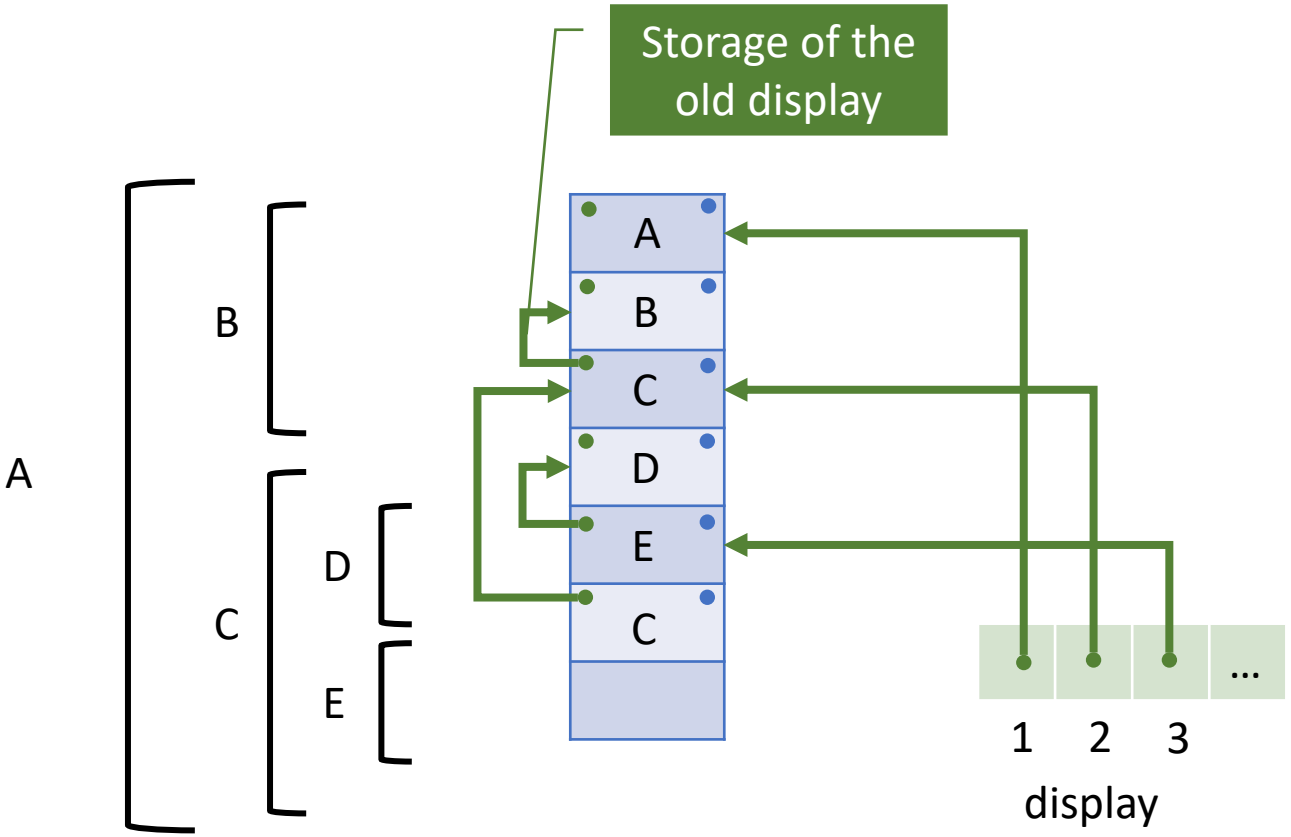
Example

- Sequence of calls: A, B, C, D, E, C



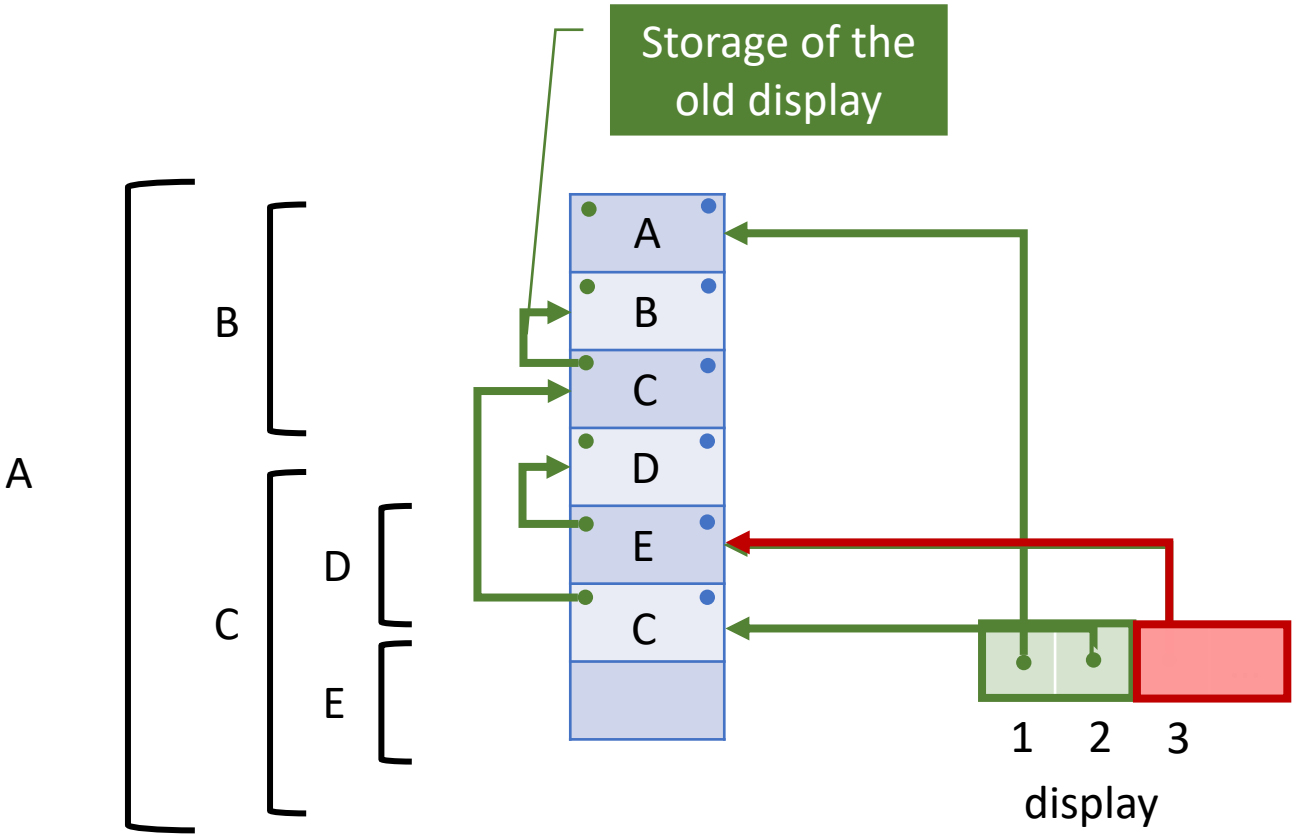
Example

- Sequence of calls: A, B, C, D, E, C



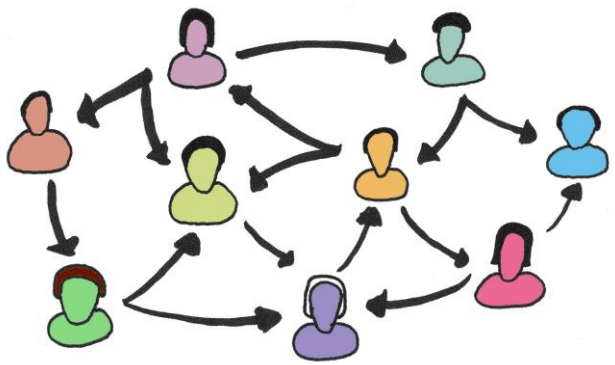
Example

- Sequence of calls: A, B, C, D, E, C



Static chain versus display

- The display can be kept in registers, if there are enough - it speeds up access and maintenance
- Overall: Static chain is better, unless the display can be kept in registers
- Modern implementations rarely use this technique, as static chains longer than 3 are rare

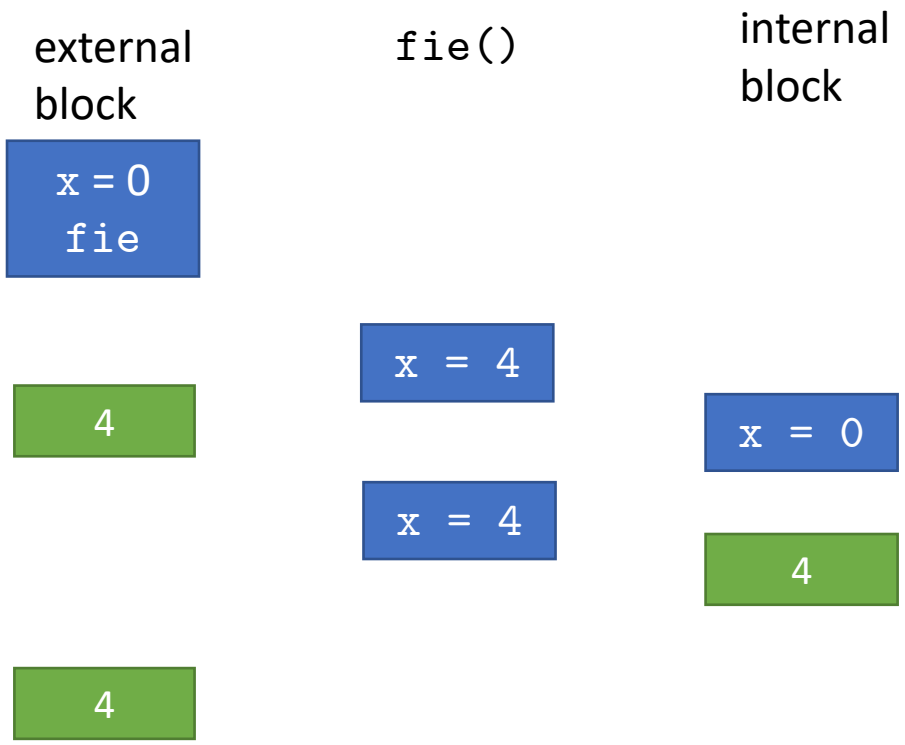


Dynamic scoping

Dynamic scoping

- In dynamic scoping a non-local name is resolved in the block that has been most recently activated and has not yet been deactivated

```
{int x=0;
 void fie (int n){
    x = n+1;
 }
 fie(3);
 write (x);
    {int x=0;
     fie(3);
     write (x);
     }
  write (x);
}
```



Dynamic scoping

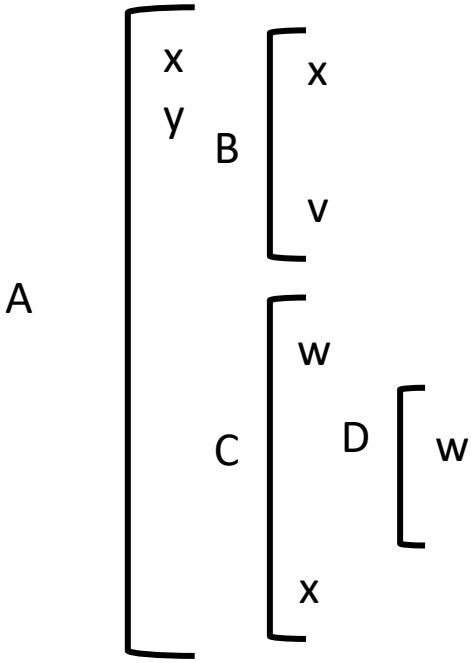
- Under dynamic scoping, the association between names and denotable objects depends on
  - The flow of control at runtime
  - The order in which subprograms are called
- The basic rule is simple:
  - The current association for a name is the one determined by the last association that has been called and not yet destroyed

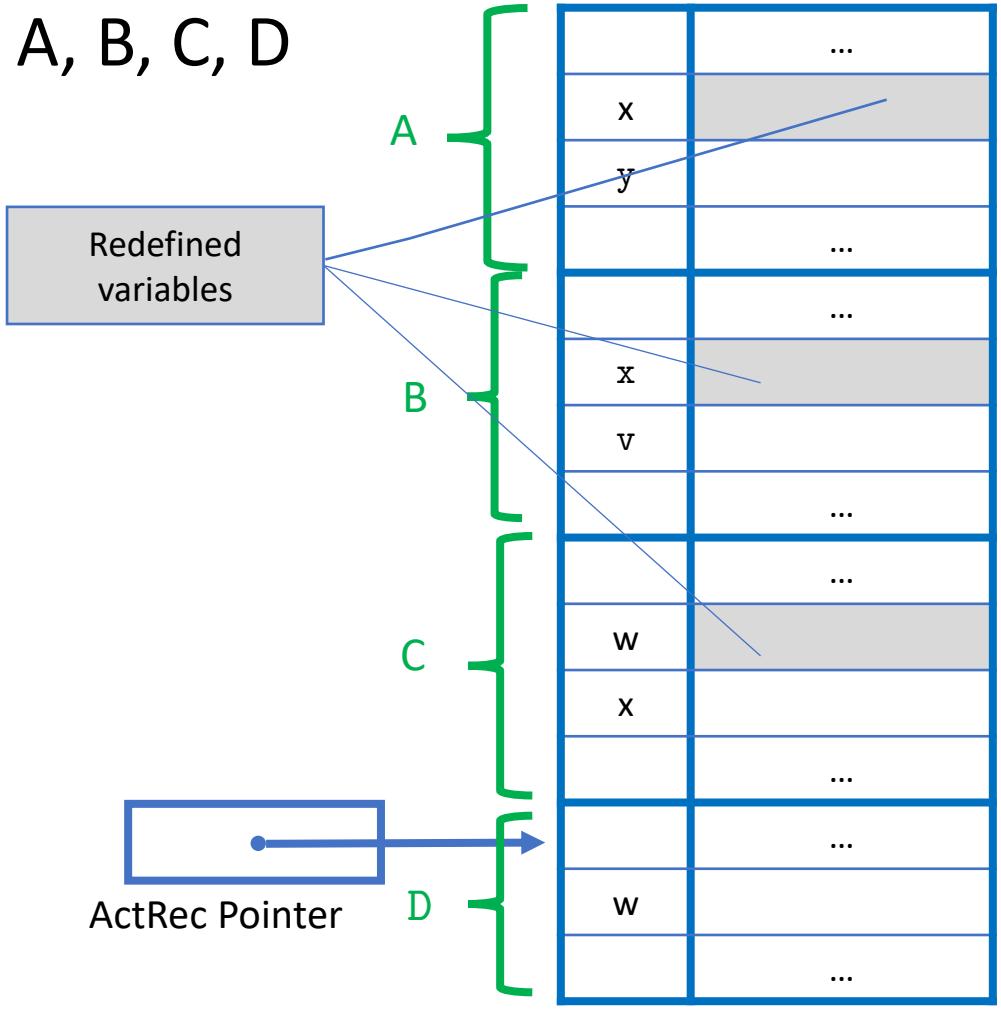
Implementation is simple

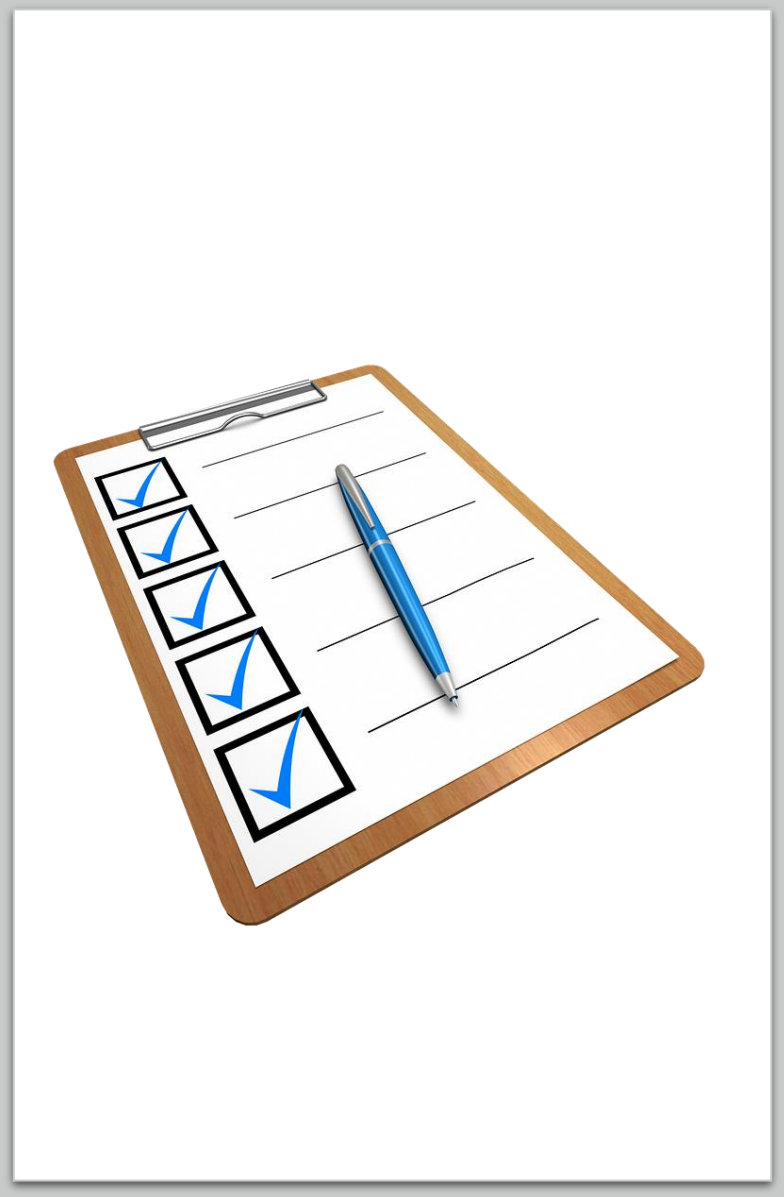
- Since the non-local environments are in the order in which they are activated at runtime, it is enough to look in the stack
- The names can be stored directly in the activation record
- Search for names via the stack until we do not find the activation record in which `x` is declared

Example

- Sequence of calls: A, B, C, D







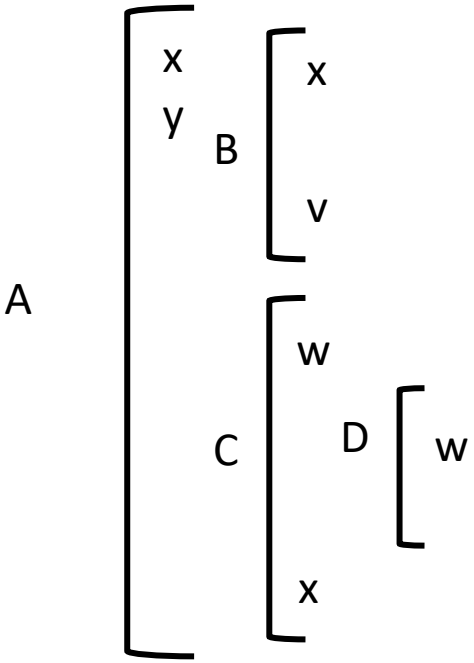
Association-List

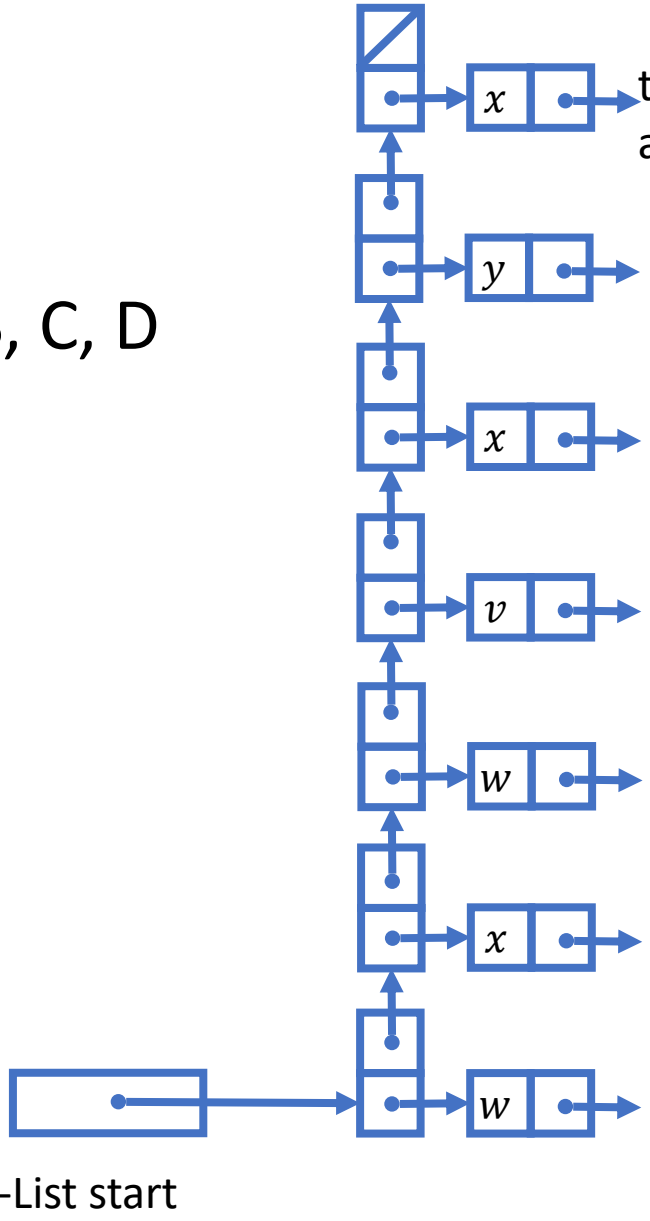
A-List

- The name-object associations are stored in an appropriate structure, managed as a stack
- 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 objects will contain the location in memory where the object is actually stored, its type, a flag which indicates whether the association for this object is active.

Example

- Sequence of calls: A, B, C, D



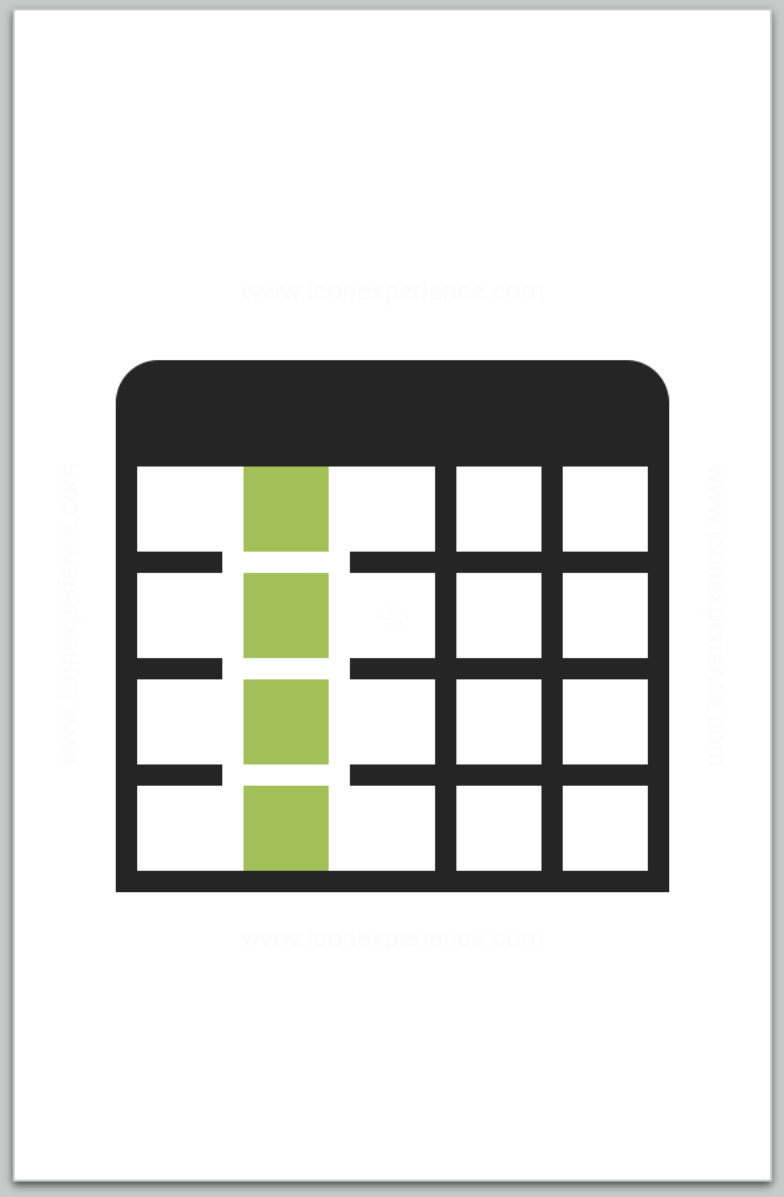


Activation record and A-list storing drawbacks

1. Names must be stored in structures present at runtime (differently from the case of static scoping)
2. The runtime search introduces inefficiency

A-List

- Simple to implement
- Memory use: Names are listed explicitly (as for the activation records)
- Management costs:
  - Entrance/exit from a block: Insertion/removal from a stack
- Access cost: Linear in the depth of the A-list (as for the activation records)
- The average access cost can be reduced, but at the cost of increasing the work on entrance/exit from a block



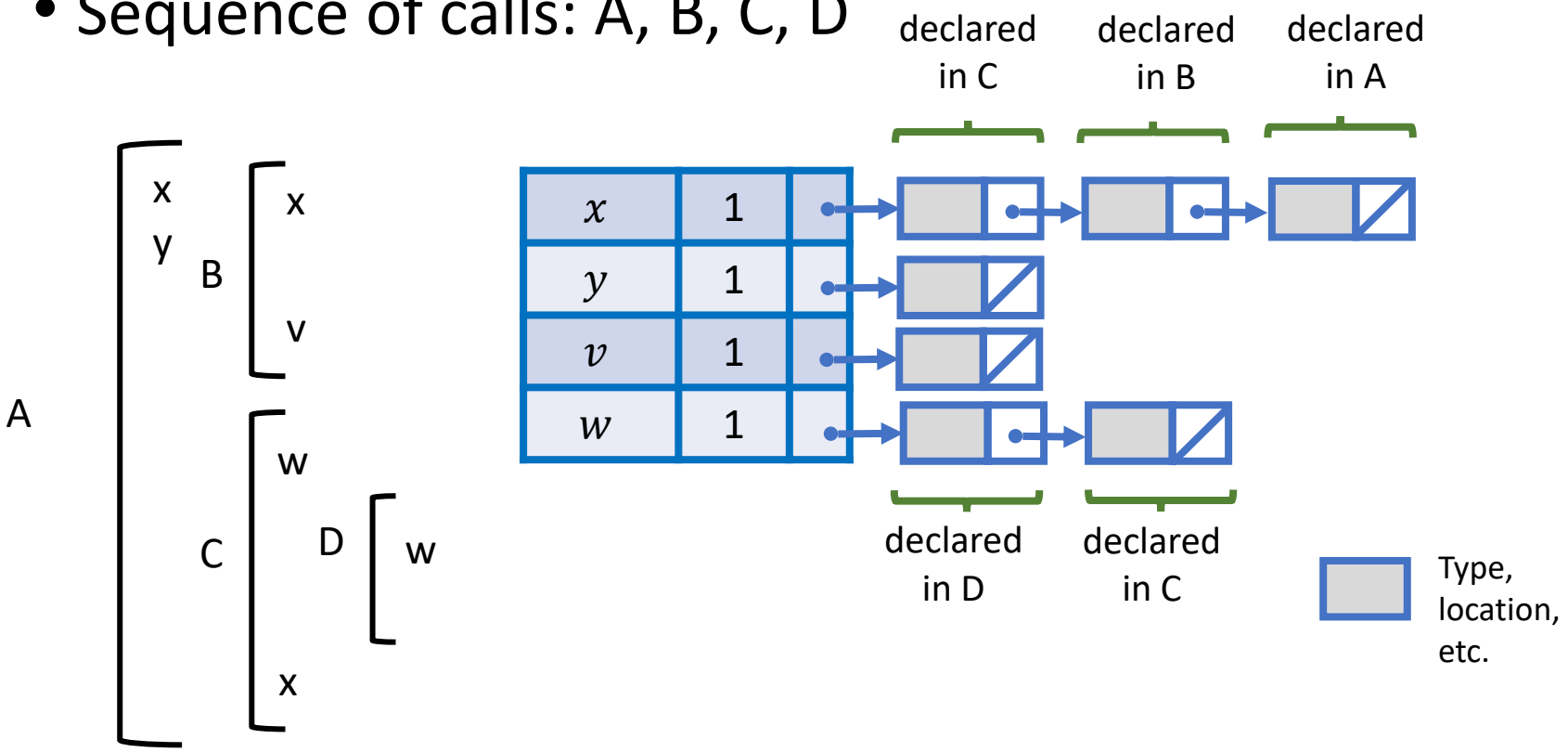
Central Referencing Environment Table

Central Referencing environment Table (CRT)

- All the blocks in the program refer to a central table (the CRT)
- The table stores all the distinct names of the program
  - If they are all known at compile time: access in constant time
  - Otherwise, use hash functions
- Each name has:
  - a flag indicating whether it is active
  - an association list (info about the object associated with the name)
    - most recent first
    - followed by the inactive ones
- Constant access time - Avoids the costly scanning of A-lists

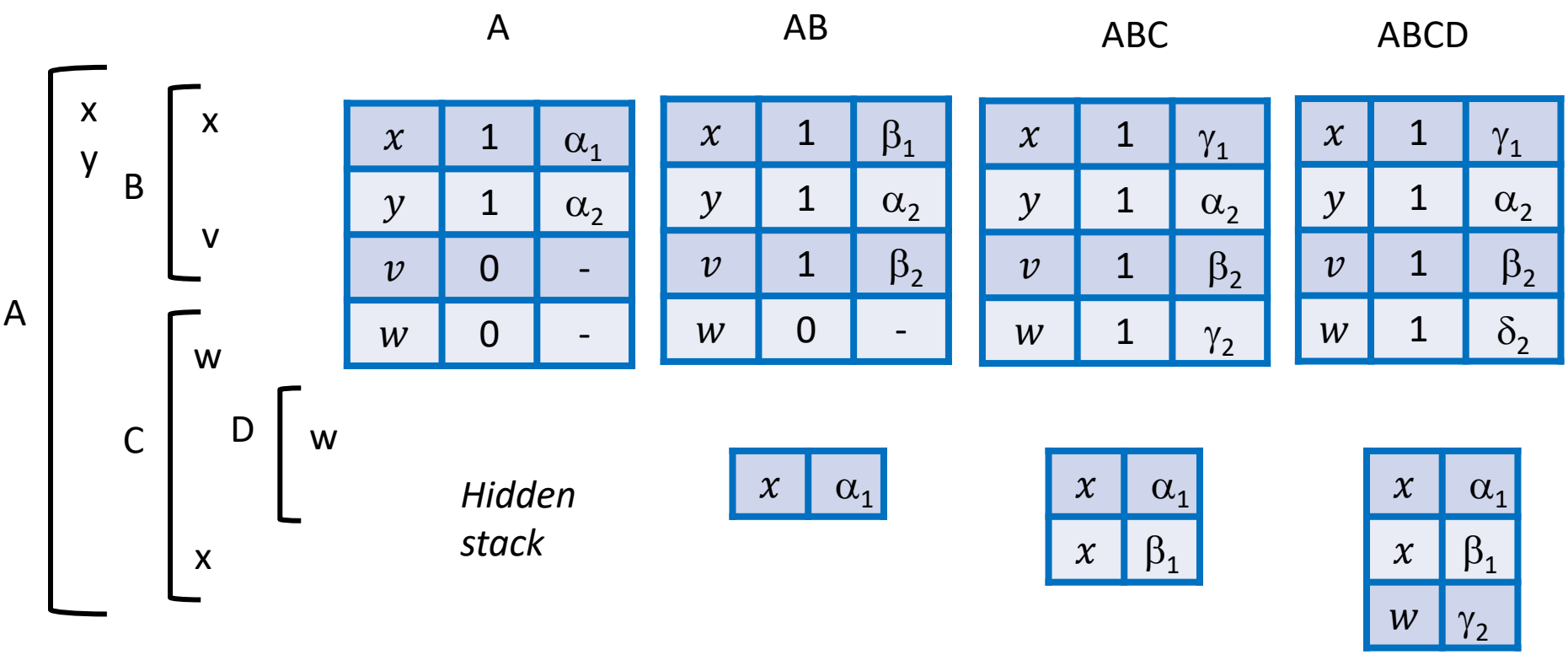
An example

- Sequence of calls: A, B, C, D



CRT and hidden stack

- Sequence of calls: A, B, C, D



CRT

- More complex than A-List
- Lower memory usage
  - If names are used statically, the names themselves are not needed
  - In any case, each name is stored only once
- Costs of management
  - Entry/exits from a block: management of all the lists of all the names present in the block
- Access time: constant (2 indirect accesses)

Exercise 3.1

- Consider the following program fragment written in a pseudo-language using static scoping.

```
void P1 {
     void P2 {body-of-P2}
     void P3 {
          void P4 { body-of-P4 }
          body-of-P3
     }
     body-of-P1
}
```

- Draw the activation record stack region that occurs between the static and dynamic chain pointers when the following sequence of calls, `P1, P2, P3, P4, P2` has been made (is it understood that at this time they are all active: none has returned).

Exercise 3.2

- Given the following code fragment in a pseudo-language with static scope and labelled nested blocks (indicated by A: { ... })

```
A: { int x = 5; goto C;
     B: {int x = 4; goto E;
     }
     C: {int x = 3;
          D: {int x = 2;}
          goto B;
          E: {int x = 1; // (**)
          }
     }
}
```

- The static chain is handled using a display. Draw a diagram showing the display and the stack when execution reaches the point indicated with the comment (**). As far as the activation record is concerned, indicate what the only piece of information required for display handling is.

Summary

- Memory allocation
  - Static allocation
  - Dynamic allocation with a stack
  - Dynamic allocation with a heap
- Scoping rules and memory
  - Static scoping: Static chain and display
  - Dynamic scoping: A-List and CRT

Next time

- Control structure and abstraction