λ

Control Structures and Abstraction

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

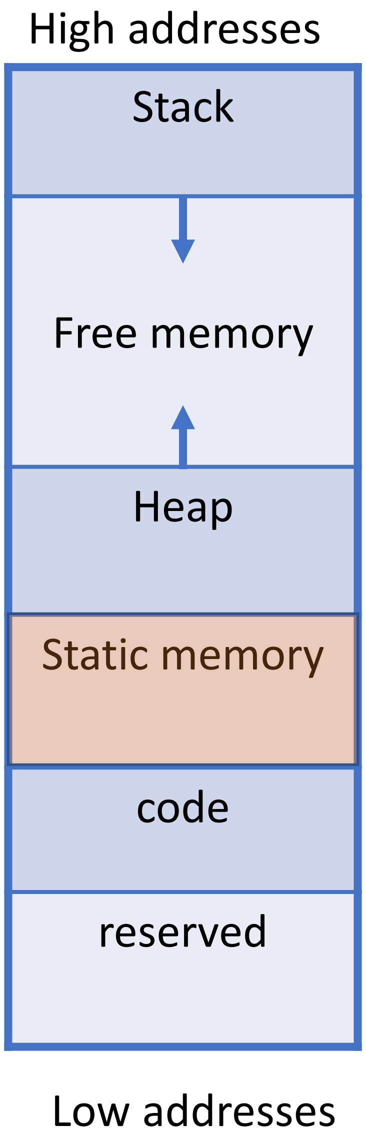
Today

- Recap
- Dynamic scoping
- Abstraction of control
- Methods of parameter passing

Recap

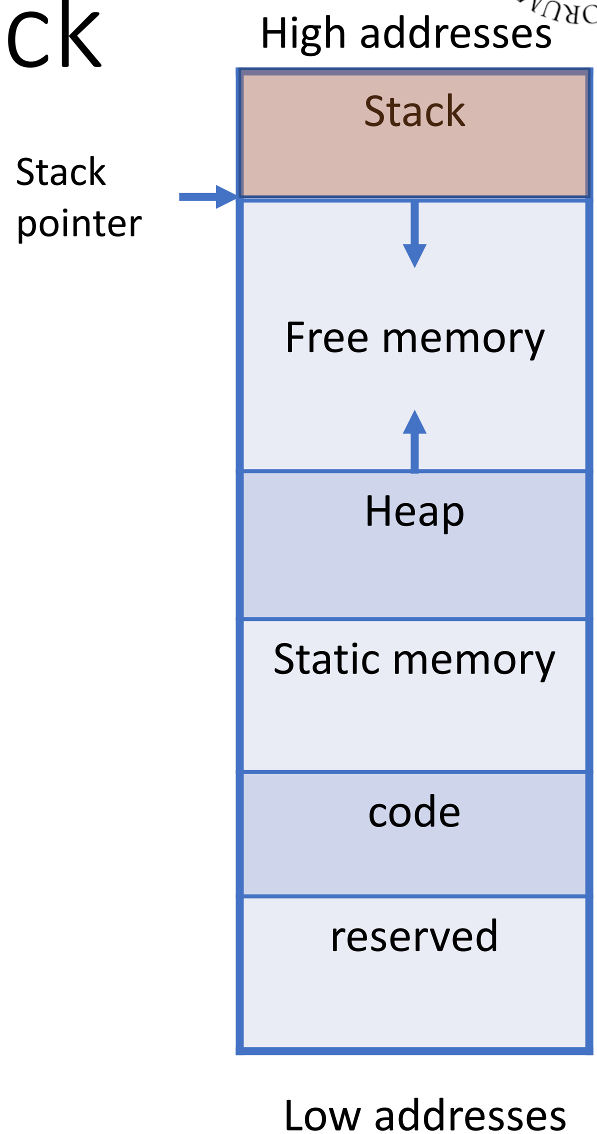
Static allocation

- 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



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



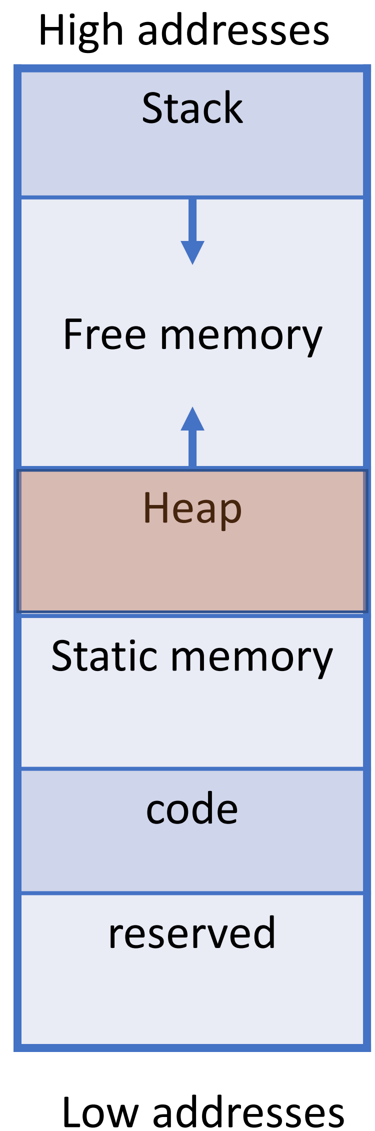
Example

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

Dynamic chain pointer
Static chain pointer
Return address
Address for result Address of the location where the final value of `fact(n)` should be placed
Parameters `n`
Local variables
Intermediate results `fact(n-1)`

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

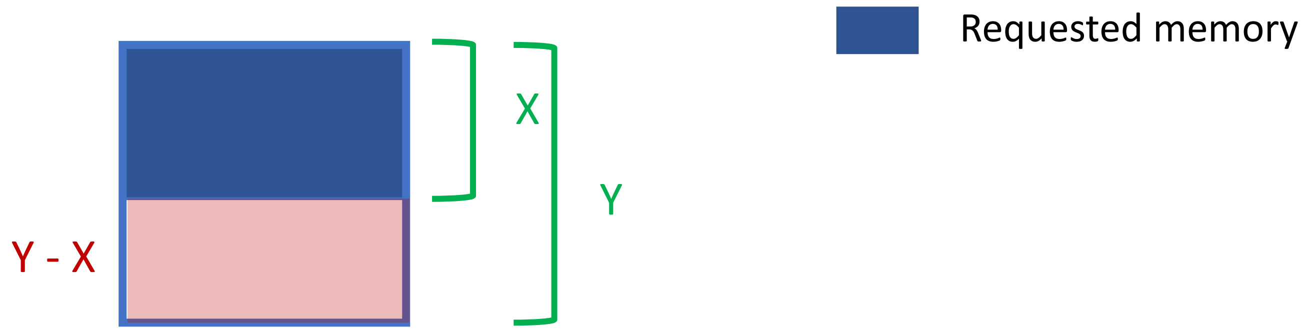


Two types of heap management methods

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

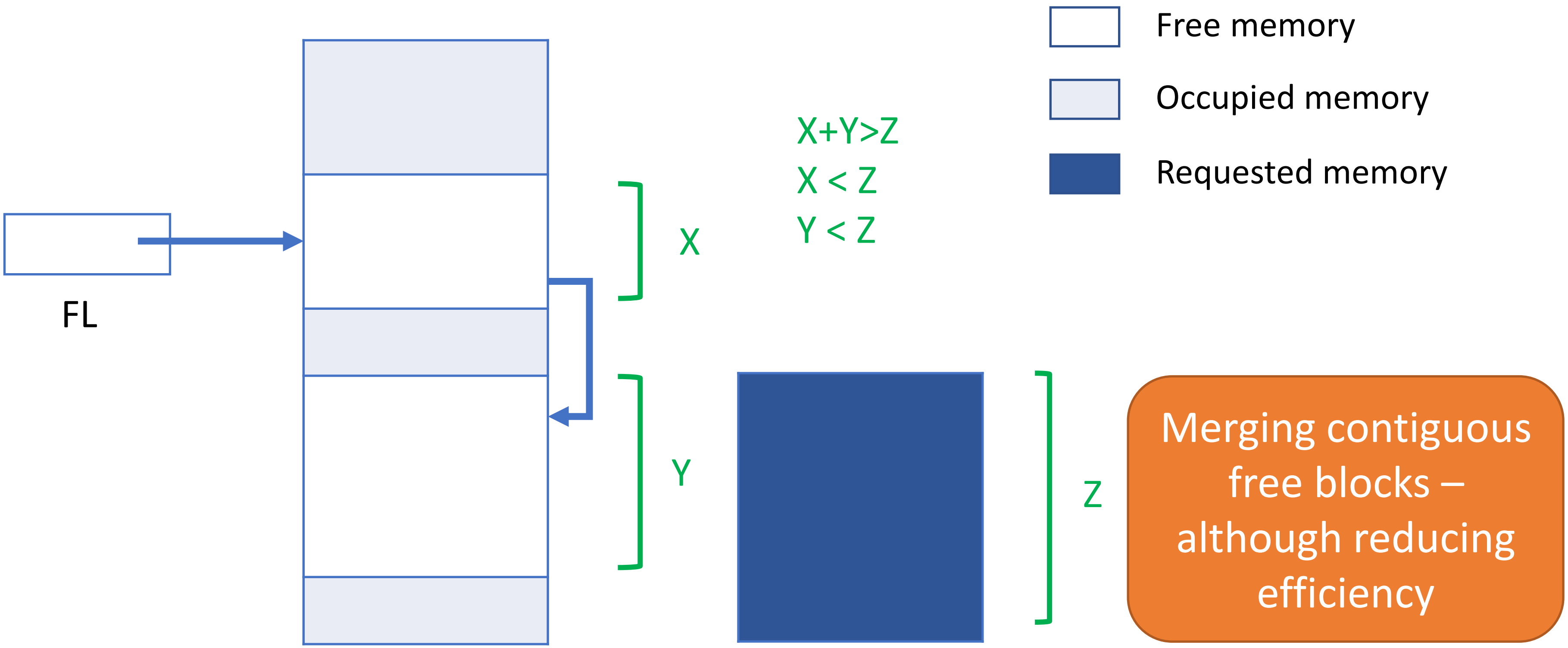
Internal Fragmentation

- 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



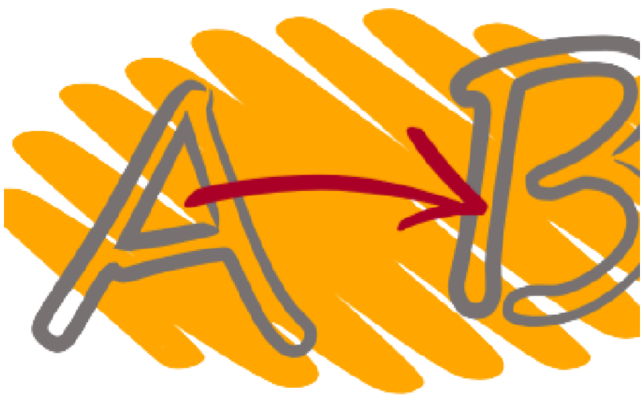
How to deal with fragmentation?

- Single free list
  - Direct use of 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
- Multiple free lists
  - Buddy system: k lists, with the k^{th} list having blocks of size 2^k
  - Fibonacci heap: similar, but uses Fibonacci numbers as block size (that grows more slowly than 2^k)

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



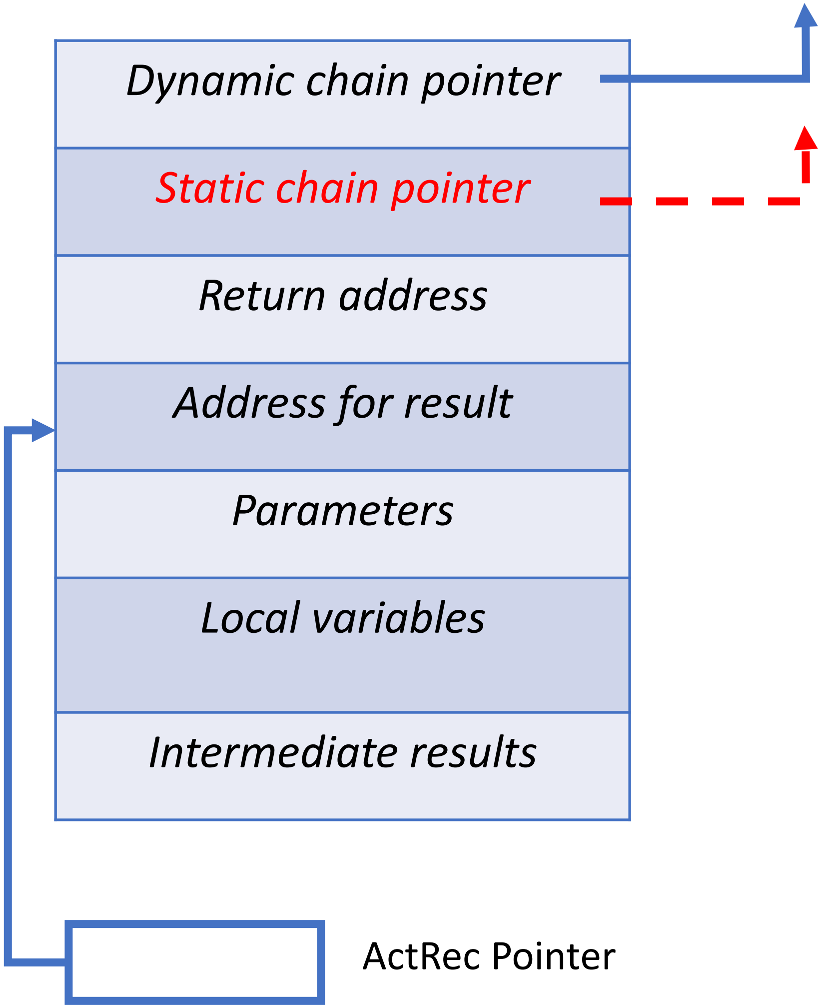
Static (lexical) scoping

How to determine the correct association for non local references?

- 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
- To this aim we can use the static chain

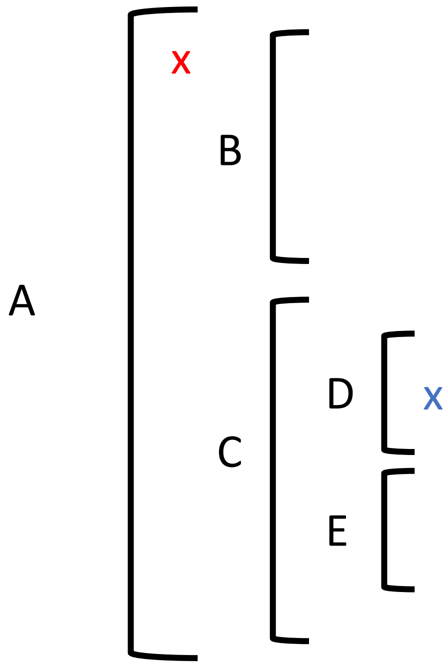
Activation record for static scoping

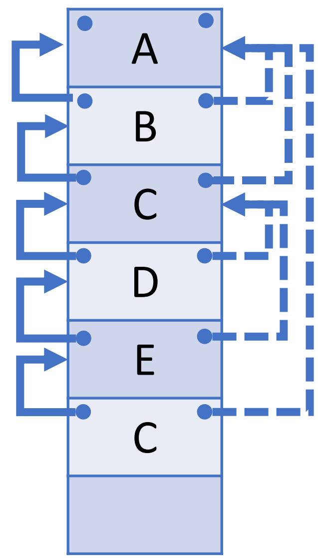
- Static chain pointer: Pointer to the activation record of the block that immediately contains the text of the current block
- A static link depends on the static nesting of the declarations of the procedure



Follow the static chain

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





In order to retrieve the correct reference to x (a non-local reference in D), we need to follow the static chain

Static chaining: two issues

- How many steps we need to carry out in the static chain at runtime?
  - We would like to be efficient at runtime without checking each activation record declaration along the static chain
- How do we know what is the static chain pointer of a new activation record?

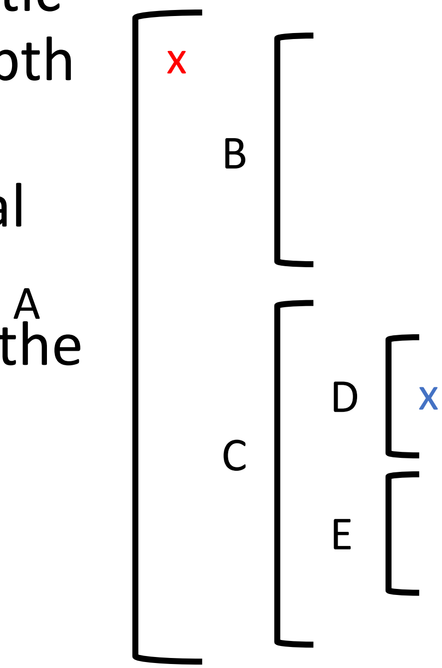
How many steps in the static chain?

- Given a variable used in a block, at compilation time we can compute
  - The chain offset: how many steps back in the static chain we have to carry out in order to reach the activation record of the block in which it is declared
  - The local offset: the offset in the activation record to reach the variable declaration
- At runtime, we can use the chain and local offset to reach the definition of the variable

At compilation time

- The static depth of a block is an integer associated with a static scope whose value is the depth of nesting of that scope
- 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
- We can determine (chain, offset, local offset)
- At runtime we know that x in D is (2, local offset)

Assuming that A is the main



SD (A) = 0
SD (B) = SD (C)= 1
SD (D) = SD (E) = 2

CO (x, D) = SD (D) –SD (A) = 2 - 0 = 2

How to determine the static link of the callee?

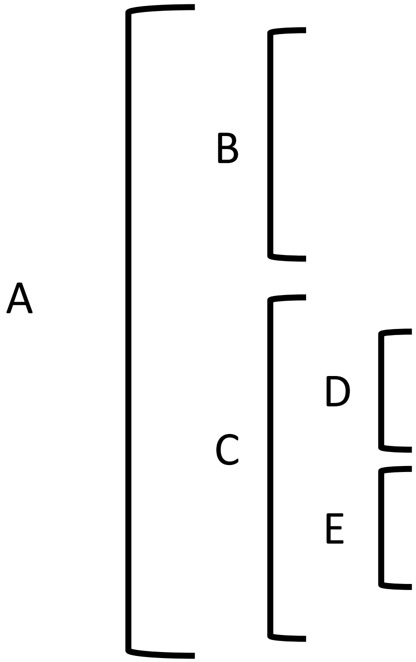
- Usually, when a new block is entered, the caller should determine the static chain pointer and pass it to the callee
- Infos can be determined at compilation time
- The caller has the following information:
  - Static nesting of blocks determined by the compiler: if the caller is at nesting level m and the callee n, the distance between them is k=m-n+1
  - Its own activation record

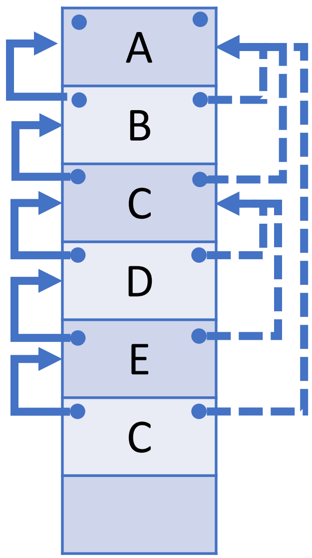
How to determine the static link of the callee?

- 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) - $k = 0$ (the distance between C and P is 0)
  b. P is in a block at k steps from C (called routine external to the caller) - $k > 0$ (the distance between C and P is k > 0)
     - For the visibility rules, P has to necessarily be located in an outer blocks which includes the caller's block, otherwise we could not call it
     - 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

How to determine the static pointer of the callee?

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





A → B (k = SD (A) – SD (B) + 1 = 0 – 1 + 1 = 0)
B → C (k = SD (B) – SD (C) + 1 = 1 -1 + 1 = 1)
C → D (k = SD (C) – SD (D) + 1 = 1 -2 +1 =0)
D → E (k = SD(D) – SD (E) = 2 – 2 + 1 = 1)
E → C (k = SD (E) – SD (C) = 2 – 1 + 1 = 2)

- From A to B we pass the static pointer of A to B
- From B to C we do a step of the chain and we pass the static pointer of A to C
- From C to D we pass the static pointer of C to D
- From D to C we do two steps of the chain and we pass the static pointer of A to C

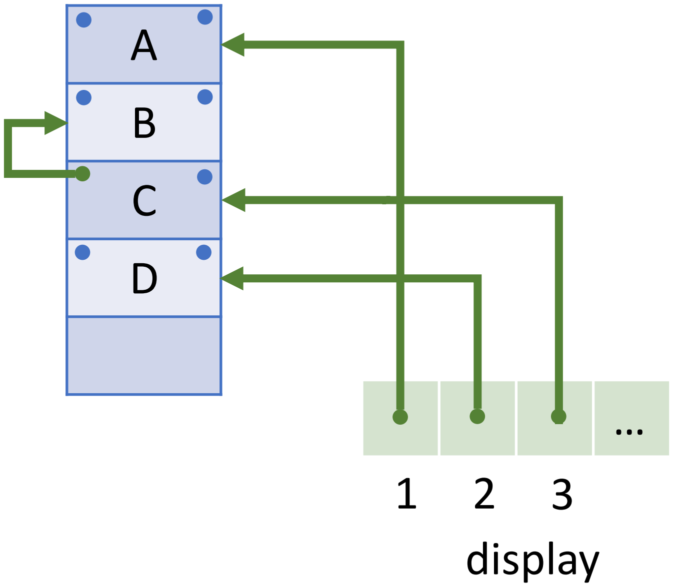
Display

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

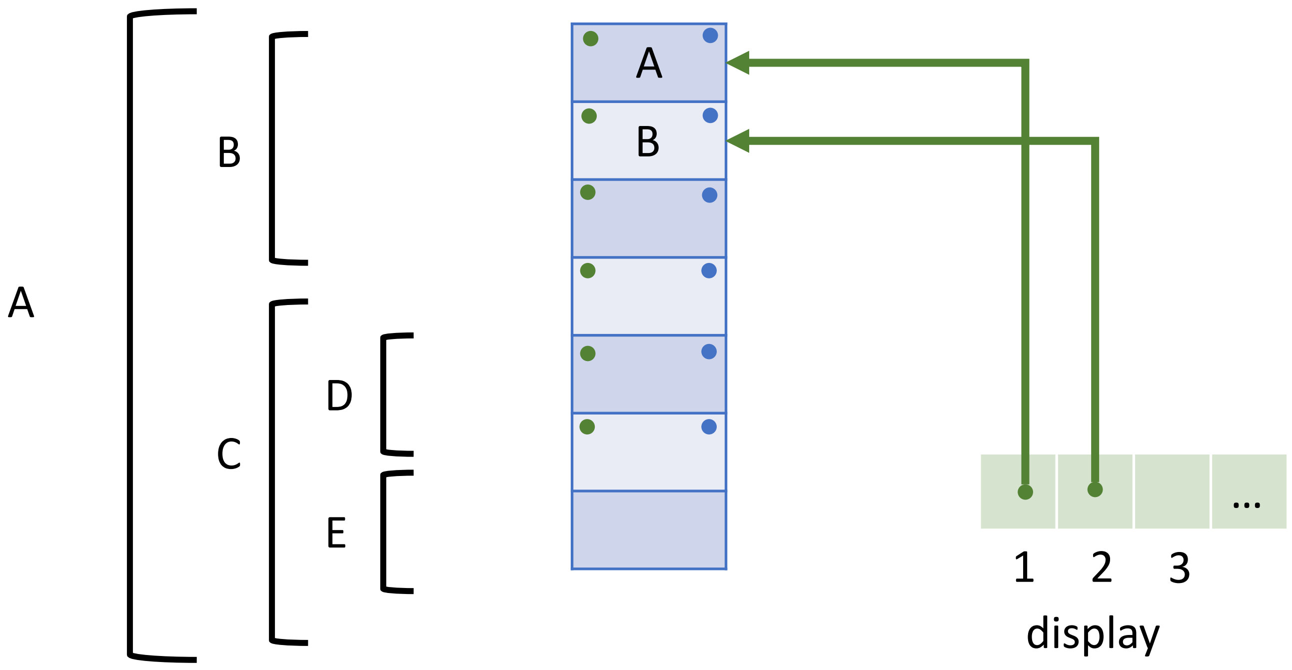


Maintaining the display

- 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

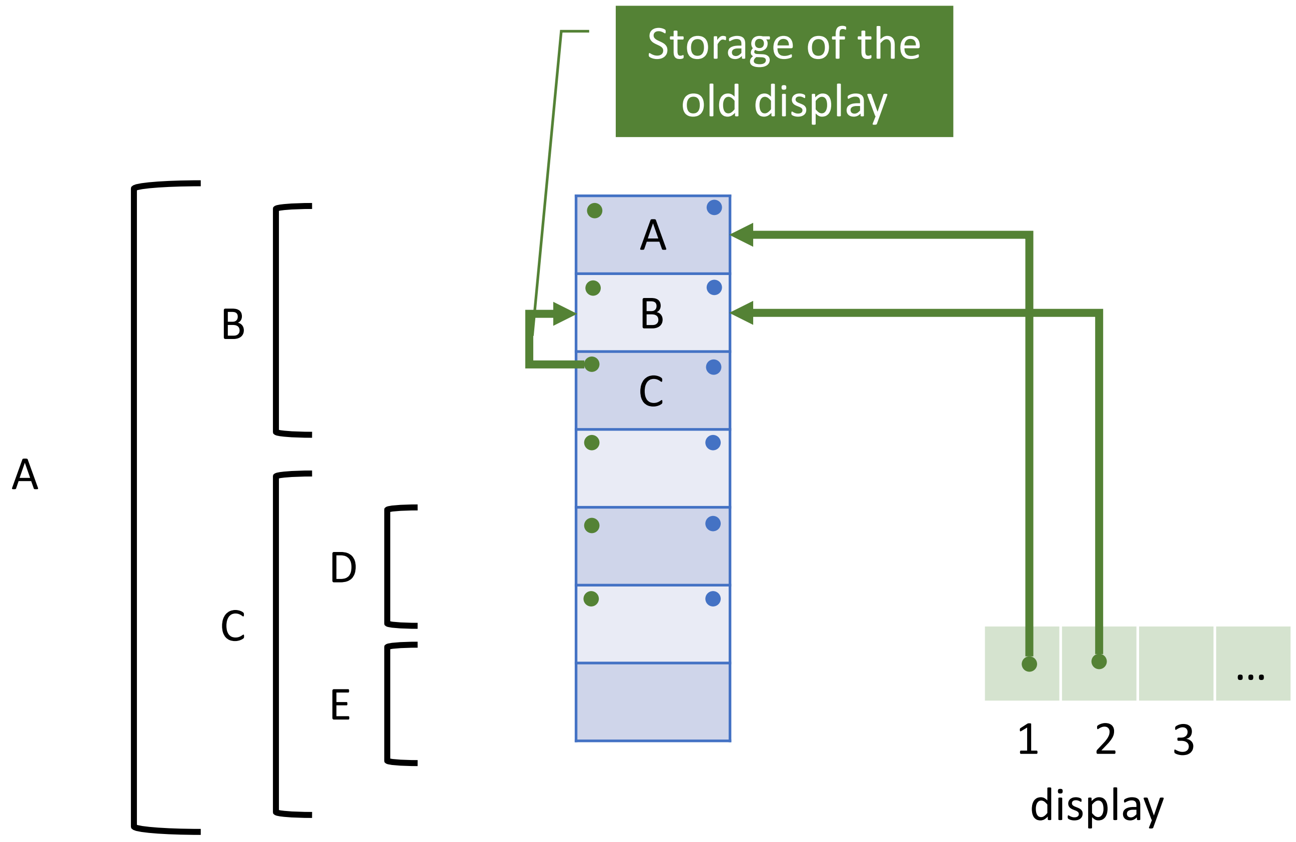
Example

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



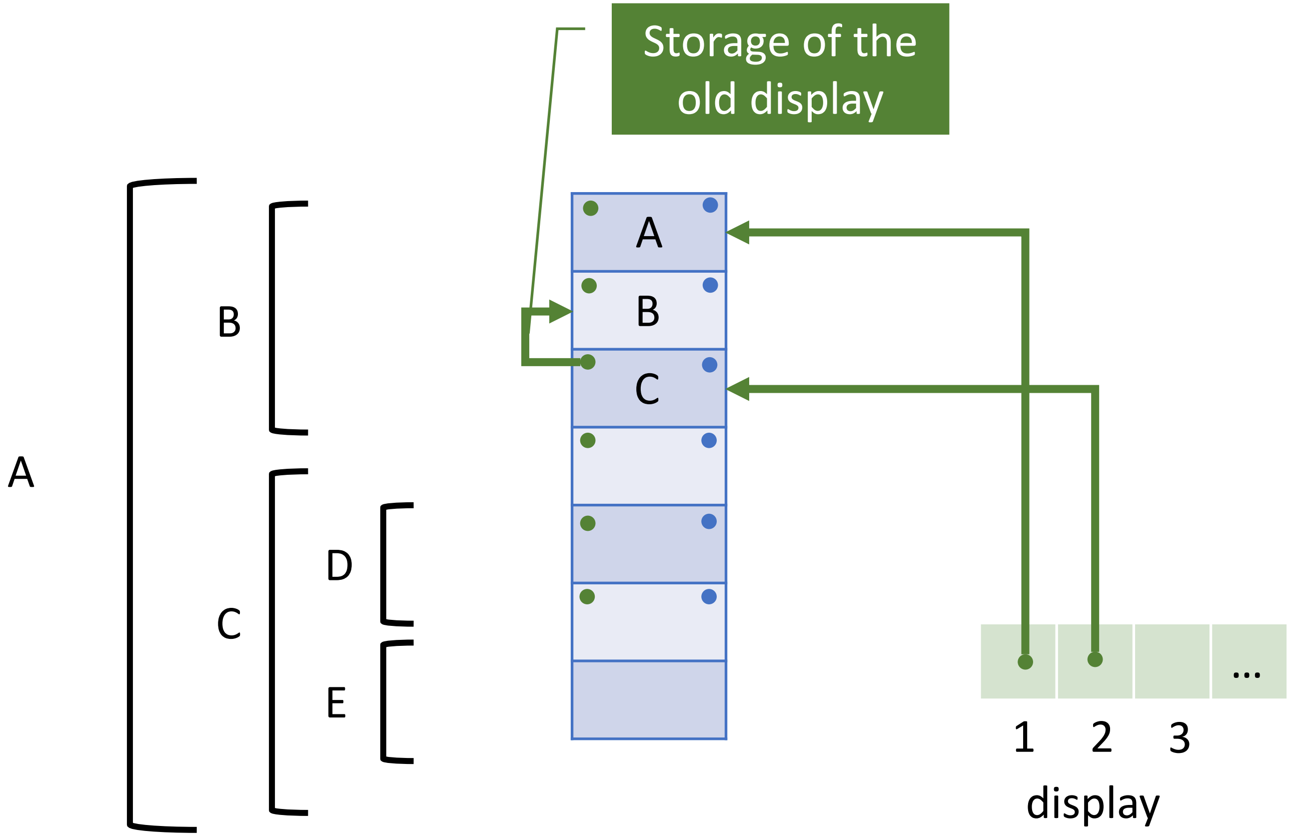
Example

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



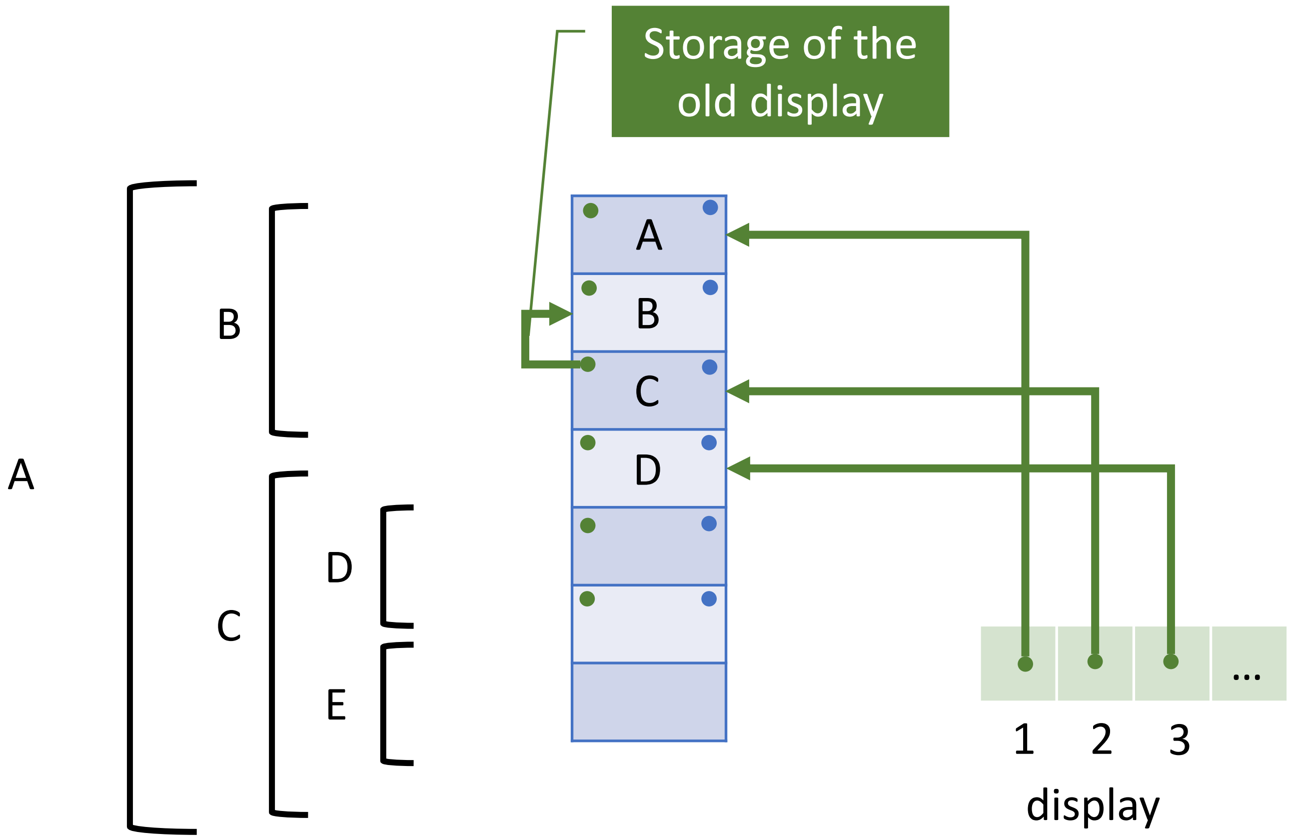
Example

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



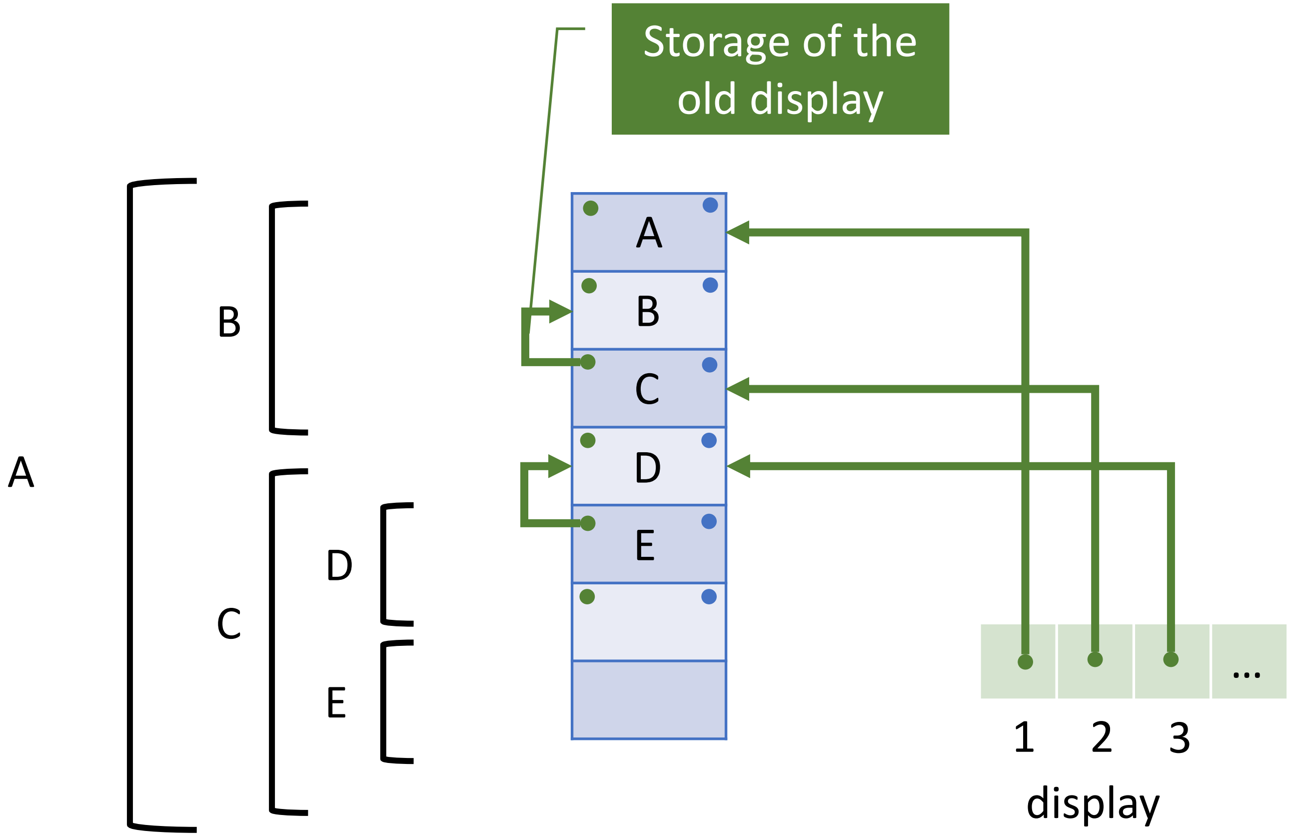
Example

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



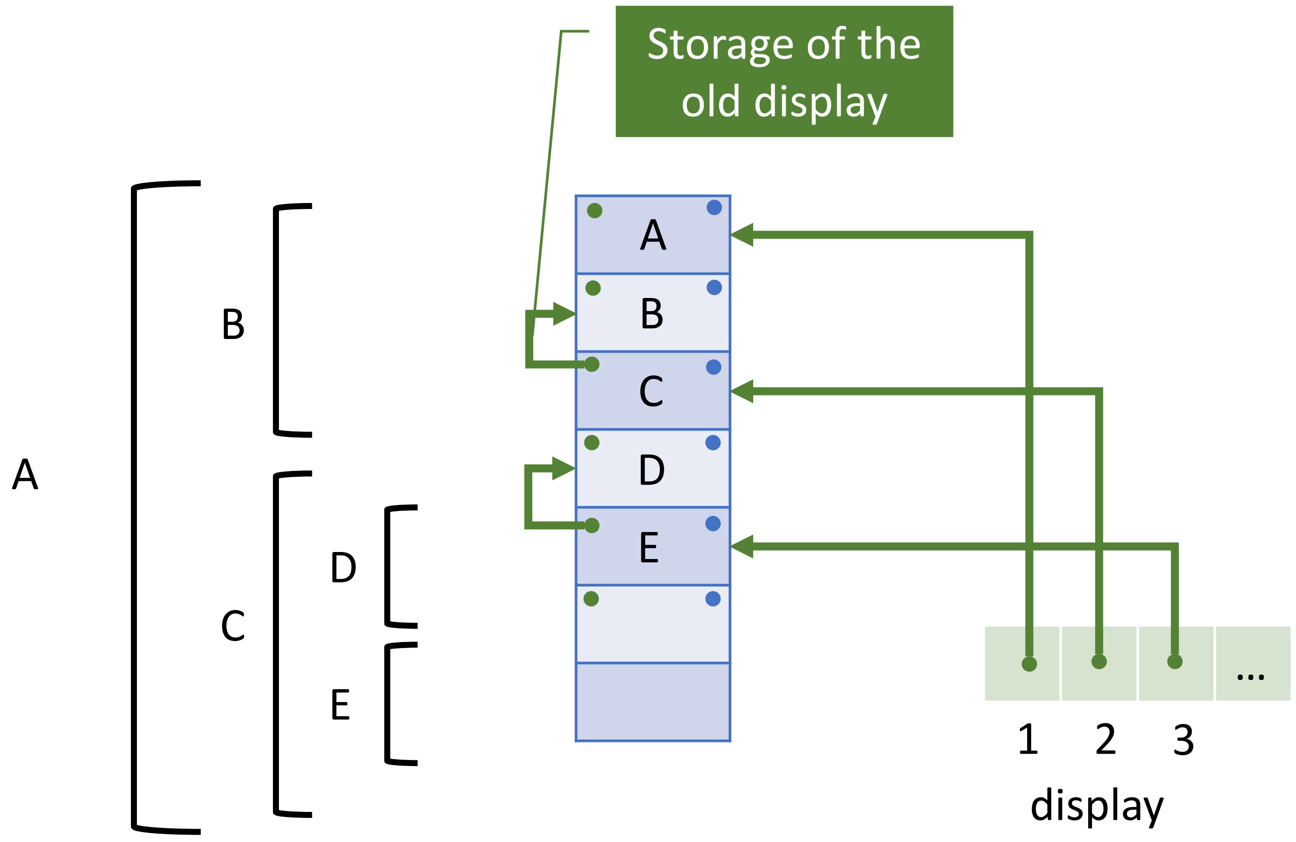
Example

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



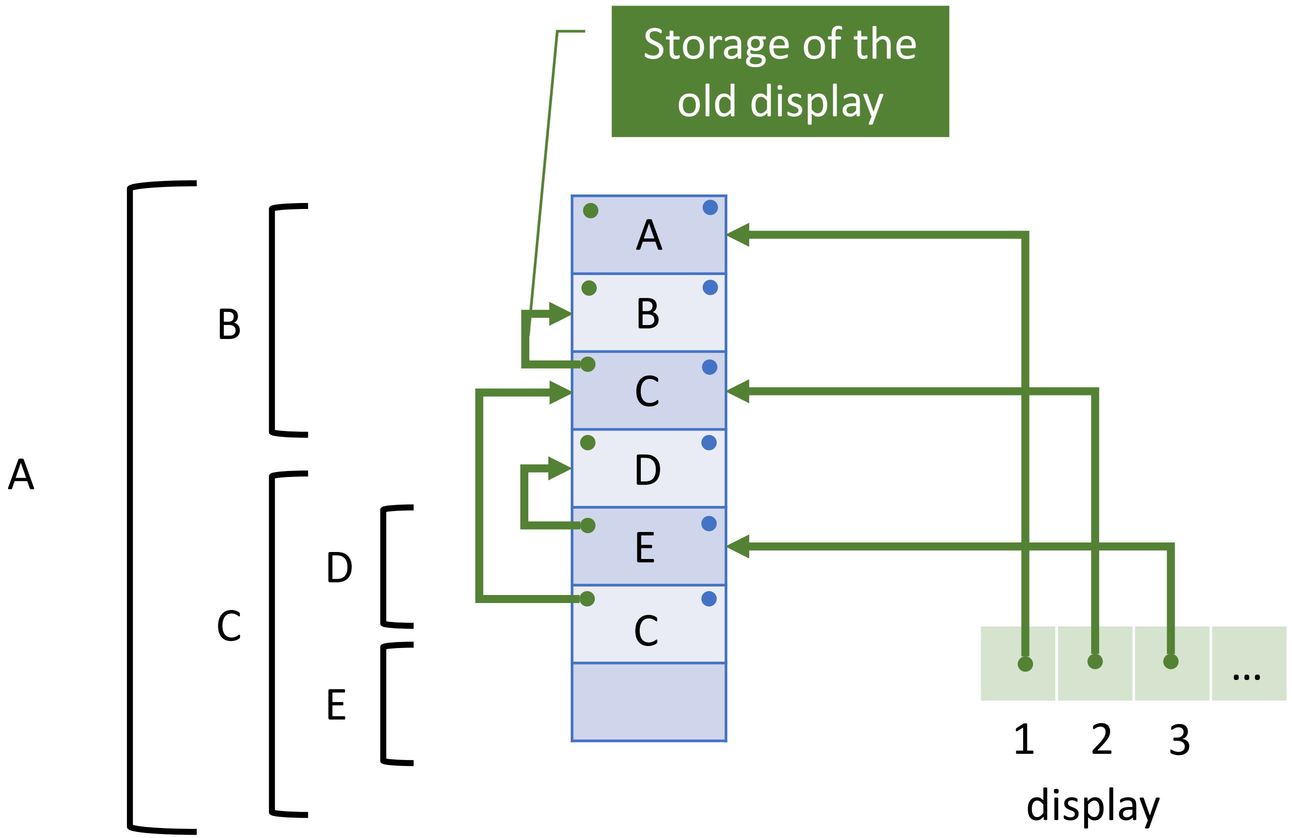
Example

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



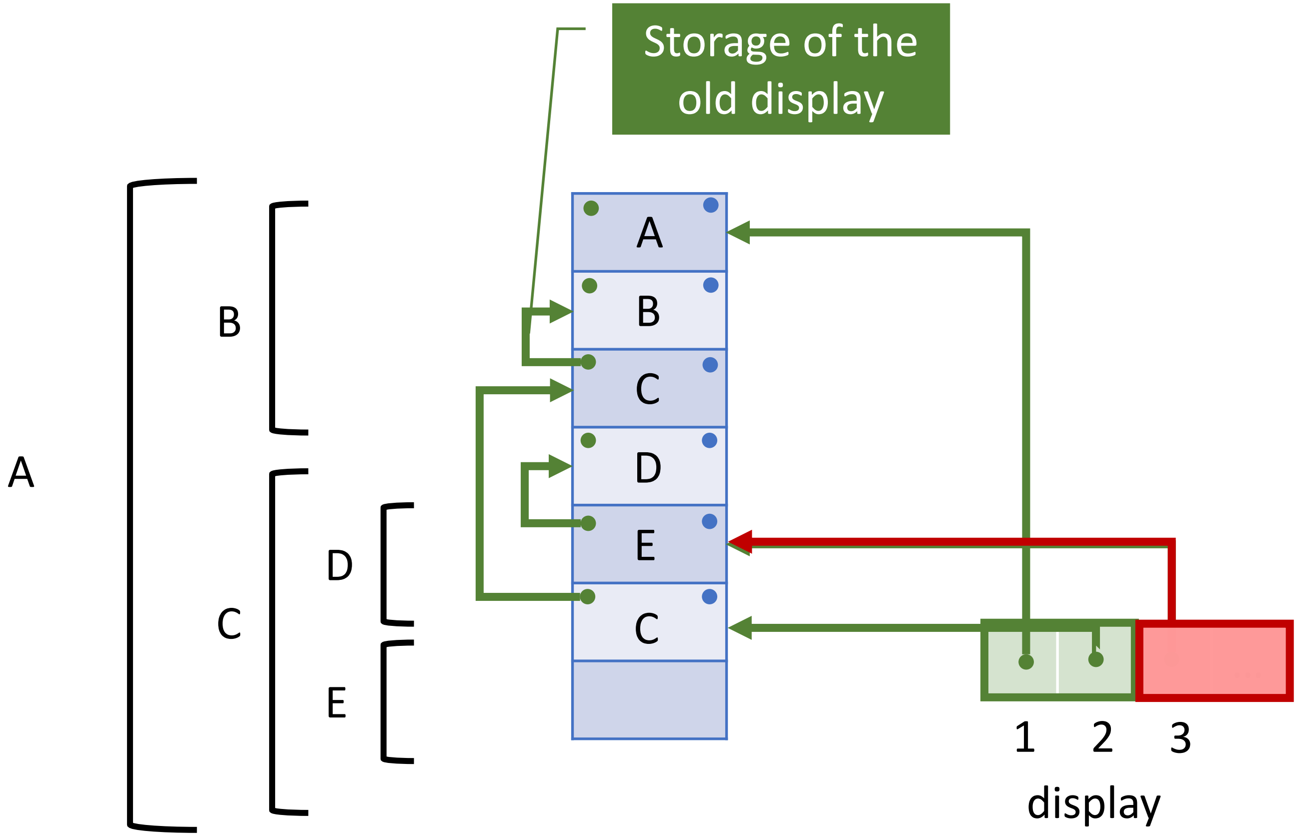
Example

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



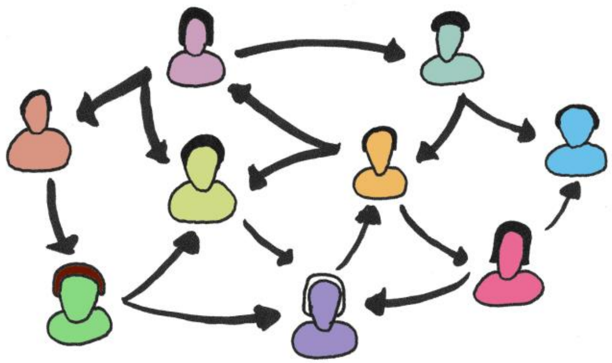
Example

- 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

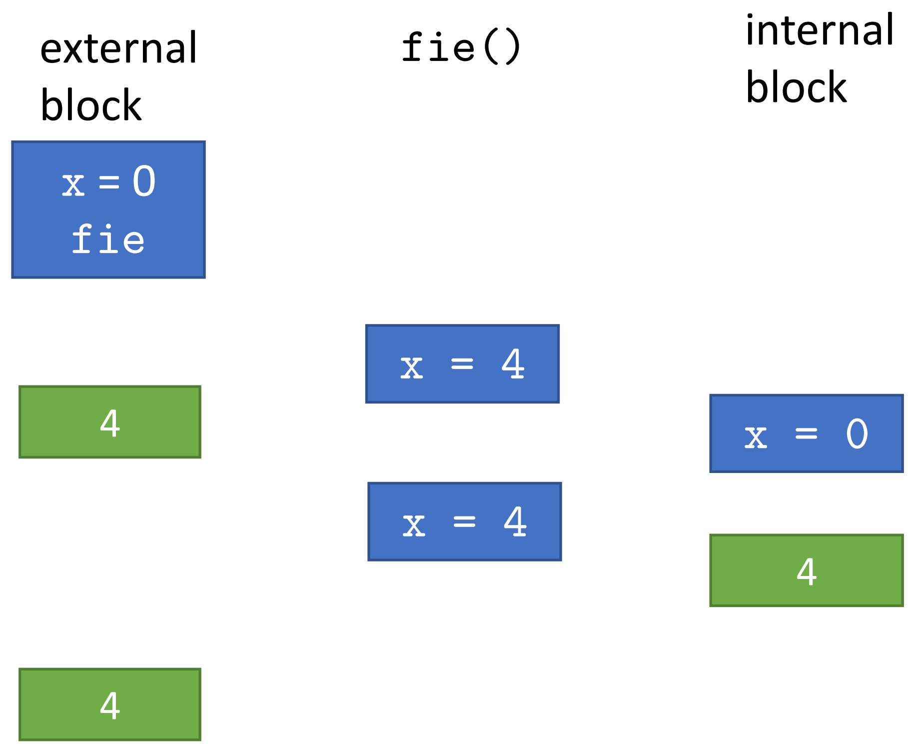


Dynamic scoping

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

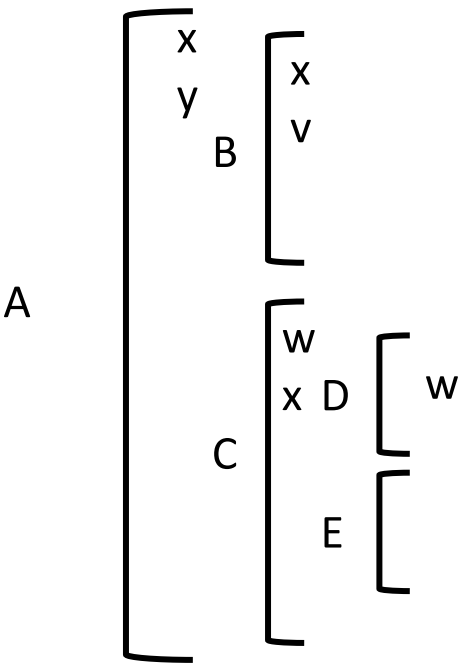
- Under dynamic scoping, the association between names and denotable objects depends on
  - The flow of control at runtime
  - The order in which subprograms are called
- The basic rule is simple:
  - The current association for a name is the one determined by the last association that has been called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

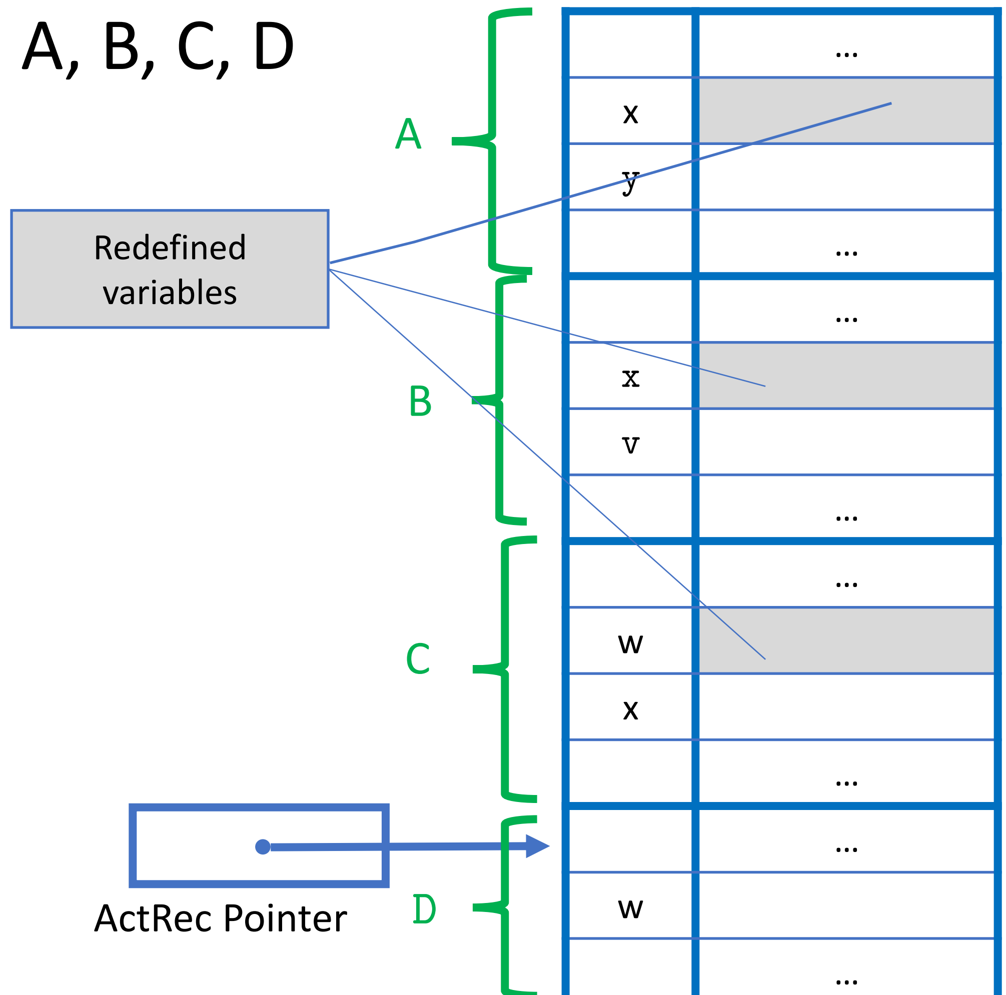
Implementation is simple

- 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





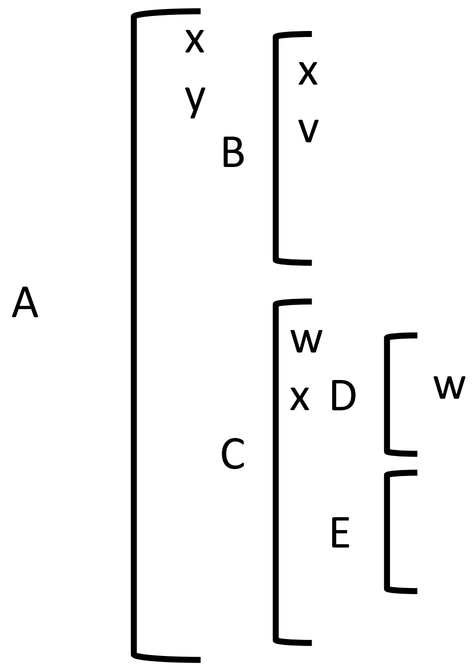
Association-List

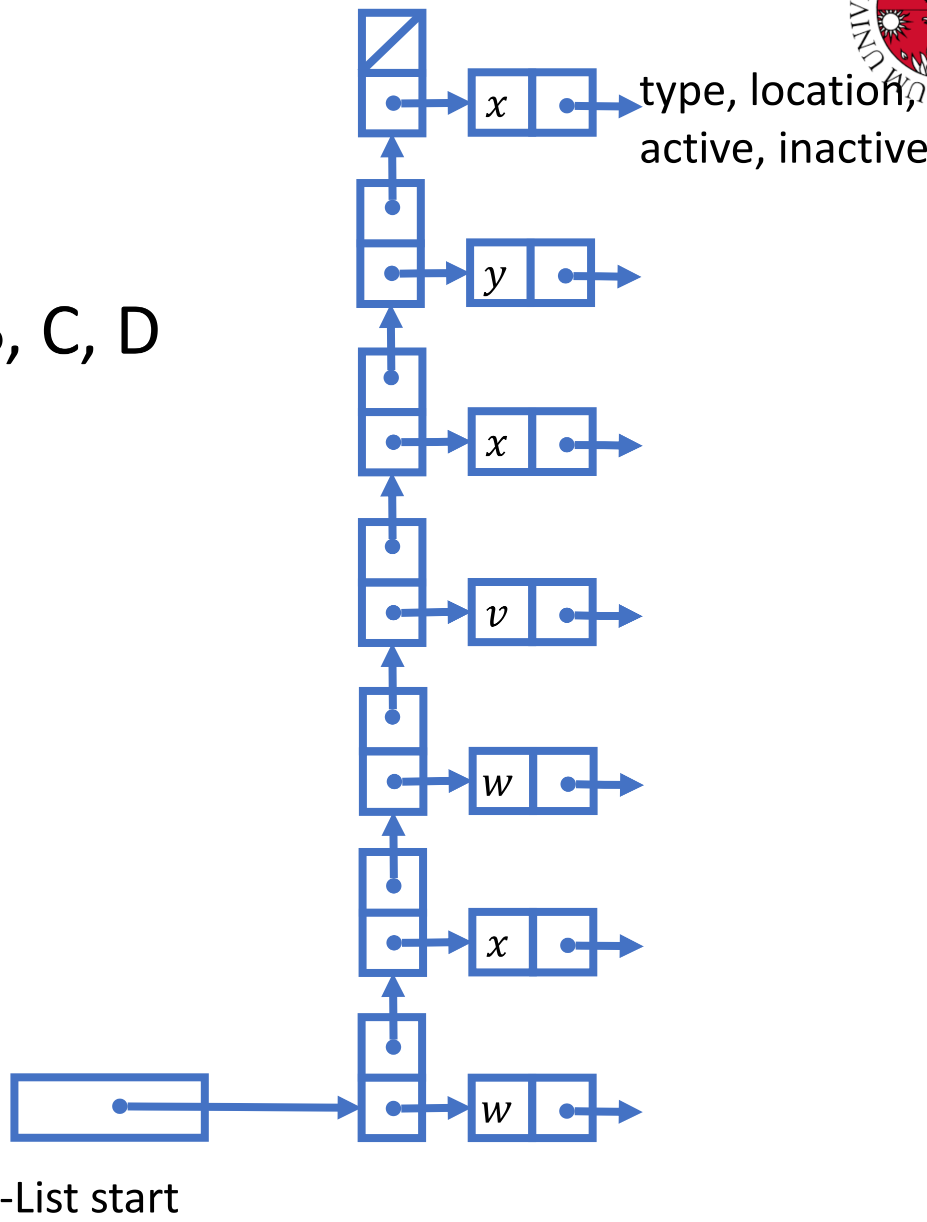
A-List

- 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



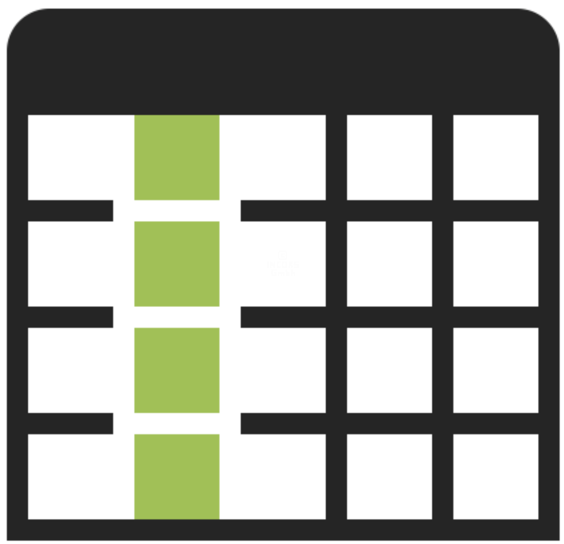


Activation record and A-list storing drawbacks

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



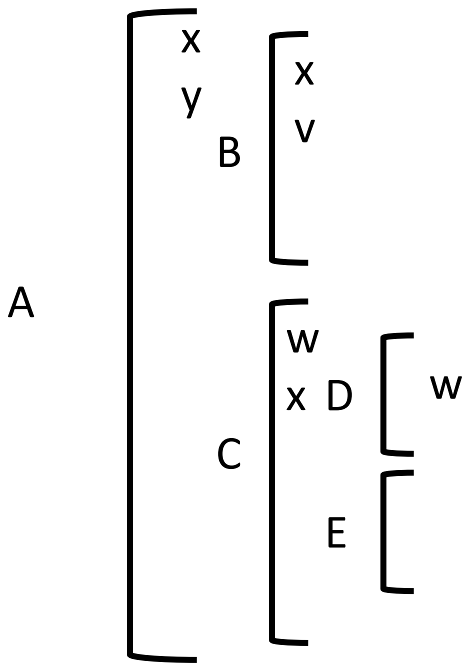
Central Referencing Environment Table

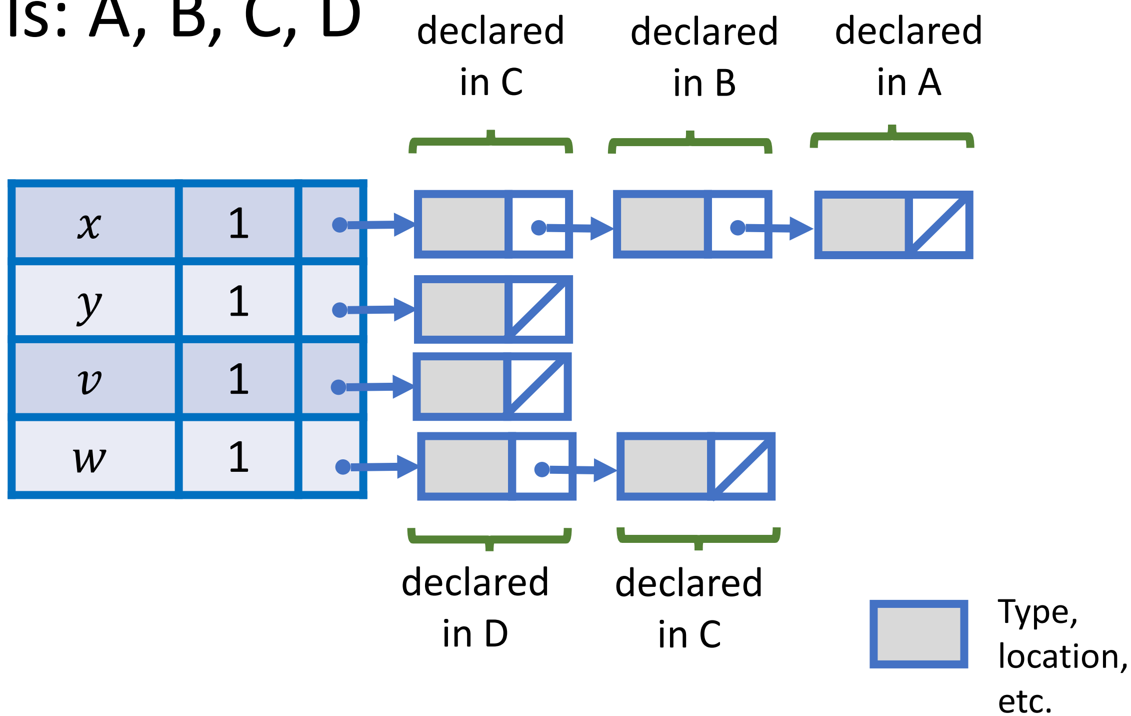
Central Referencing environment Table (CRT)

- 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

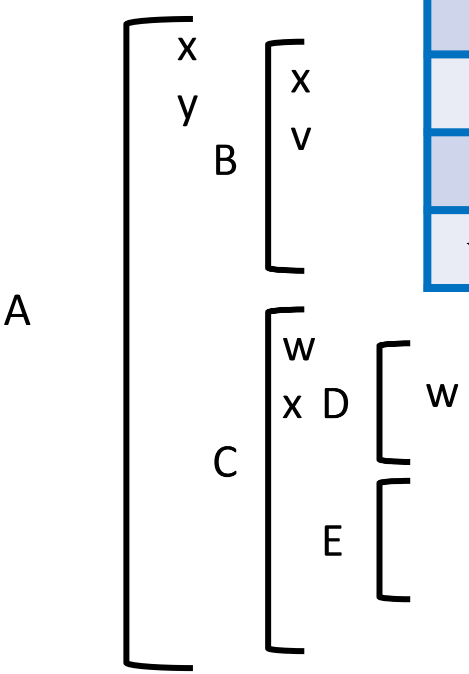
- Sequence of calls: A, B, C, D





CRT and hidden stack

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



A

x	1	α_1
y	1	α_2
v	0	-
w	0	-

AB

x	1	β_1
y	1	α_2
v	1	β_2
w	0	-

ABC

x	1	γ_1
y	1	α_2
v	1	β_2
w	1	γ_2

ABCD

x	1	γ_1
y	1	α_2
v	1	β_2
w	1	δ_2

Hidden stack

AB:

x	α_1

ABC:

x	α_1
x	β_1

ABCD:

x	α_1
x	β_1
w	γ_2

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

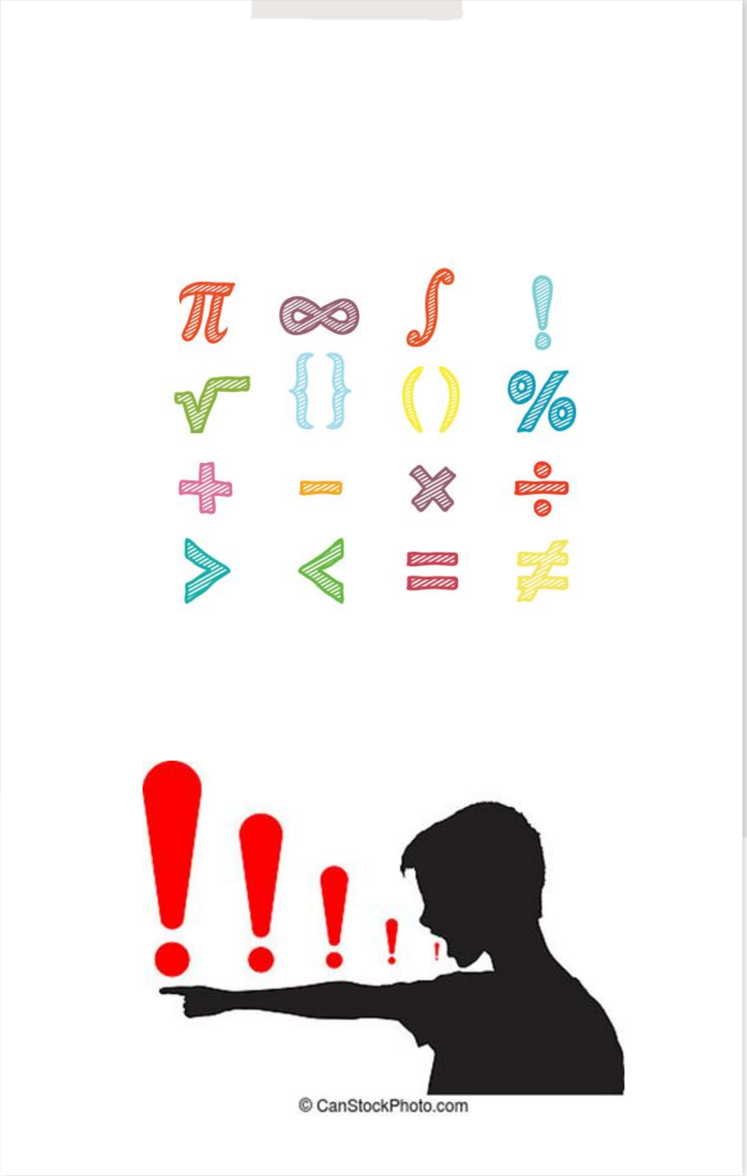
- Given the following code fragment in a pseudo-language with static scope and labelled nested blocks (indicated by A: { ... })

```
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.

Exercise 3.3

- Using some pseudo-language, write a fragment of code such that the maximum number of activation records present on the stack at runtime is not statically determinable



Expressions and commands

Expressions

- Syntactic entities whose evaluation either produces a value or does not terminate (undefined expression)
- Usually composed of:
  - A single entity (constant, variable)
  - An operator applied to a number of arguments (which are also expressions)
- Three notational systems:
  - Infix: a + b
  - Prefix (Polish): + a b
  - Suffix (reverse Polish): a b +

brackets and precedence rules to avoid ambiguity

If arity of operators is known –no need of parenthesis and precedence rules

Side effects

- Side effect: action that changes the state
- Hidden side effect: action that influences the result (partial or final) of a computation outside the context in which it is found
- In imperative languages expression evaluation can modify the value of variables

`(a+f(b))*(c+f(b))`	if `f` modifies the value of its operand as side effect, the result of the first call to `f` may differ from the second

- Languages follow various approaches:
  - Pure declarative: do not allow side effects
  - Others: forbid the use in expressions of functions that cause side effects
  - Others (e.g., Java): specifies the order of evaluation, from left to right

Undefined Operators

- Two evaluation strategies:
  - eager evaluation: first evaluating all the operands and then applying the operator to the values
  - lazy evaluation: operands are evaluated only when needed
- Some expressions in programming languages can be evaluated even when some operands are missing

`a == 0 ? b : b/a`	This expression in C demands for lazy evaluation because b/a would be evaluated even when a is equal to 0

- Lazy evaluation is more expensive to implement

Short-circuit evaluation

- Lazy evaluation of boolean expressions is often called short-circuit evaluation
- We arrive at the final value before knowing the value of all of the operands.

`a == 0 || b/a >2`	This expression in C with lazy/ short-circuit evaluation has value true With eager evaluation, we may get an error

- The order evaluation of subexpressions can influence the efficiency of the evaluation
- In ML → eager evaluation except for `andalso` and `orelse` that use the lazy evaluation

Commands

- Command: a syntactic entity whose evaluation does not necessarily return a value but can have a side effect
- Commands
  - Typical of the imperative paradigm
  - Not present in the functional and logical paradigms
  - In some cases yield a result (e.g., an assignment in C)
- The purpose of a command is the modification of the state
- The assignment command is the elementary construct in the computational mechanism for languages with commands.

Variables

- Variable: in mathematics, an unknown that can take values from a predefined domain (that cannot be modified anymore)
- In computer science it depends on the paradigm
- In classical imperative languages (Pascal, C, Ada, etc.,):
  - modifiable variable: the value can be modified
  - A container of values that have a name

`x` $\boxed{3}$

The variable with name `x`
The variable `x`

Different models

- In some imperative languages (object-oriented ones)
  - a variable is a reference to a value, which has a name and is stored in the heap
  - similar to pointers but without the possibility to directly accessing the location
- In logical languages
  - a variable can be modified only under certain conditions (instantiation)
- In pure functional languages (Lisp, ML, Hashkell, SmallTalk)
  - a variable is an identifier that stands for a value (not modifiable), as in mathematics

Assignments

- Assignment: basic command that modifies the value of a variable
- We distinguish between (l-value opAss r-value)
  - l-values: locations
  - r-values: values stored in locations
- In general a binary operator in infix form exp1 OpAss exp2
  - compute the l-value of exp1, determining the container loc
  - compute the r-value of exp2
  - modify the content of loc with the computed value

> - Some languages (e.g., Java) allow the left-hand side to be evaluated before the right-hand side
> - Others (e.g., C) leave the decision to the implementer.

Assignments

- In some languages (e.g., C) The assignment also produces a value

`x = 2;`	Besides assigning the value 2 to `x`, it also returns the value 2
`(y = (x = 2));`	It assigns the value 2 to `x` and `y`

- There exist also other assignment operators that can be used
  - For optimization reasons
  - For increasing code readability
- For example the **+=** operator `x+=1;`
  - It adds to the r-value of the expression on the left the quantity on the right
  - It assigns the result to the location obtained as the l-value of the expression on the left

```
b = 0;
a[index(3)]+=1;
```

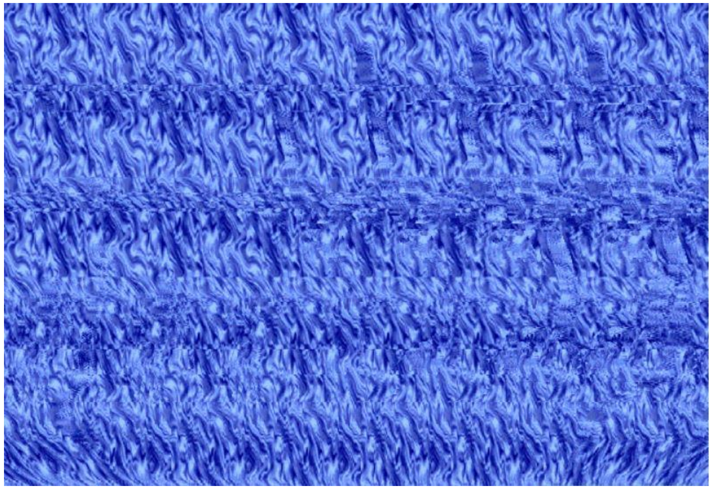
The result is that `a[1]+1` is assigned to `a[1]`

Other commands

- Sequential commands: `(C1;C2)`
- Composite commando or blocks: `{}`, `begin ... end`
- Conditional commands: `if` and `case`
- Iterative commands:
  - Unbounded (logical condition): `while, repeat, do`
  - Bounded: `do, for`

Abstraction

- Identify important properties of the thing that we want to describe
- Concentrate on relevant questions and ignore the others
- What is relevant depends on our aim
- In a programming language
  - Control abstraction: hide procedural data
  - Data abstraction: definition and usage of sophisticated data types



Abstraction of control

Abstraction of control

- Main mechanism: subprogram/procedure/function
- Subprogram: piece of code identified by its name, with a local environment and exchanging information with the rest of the code using parameters
- Two main constructs

definition

```
int foo (int n, int a) {
     int tmp=a;
     if (tmp==0) return n;
     else return n+1;
}
...
```

use

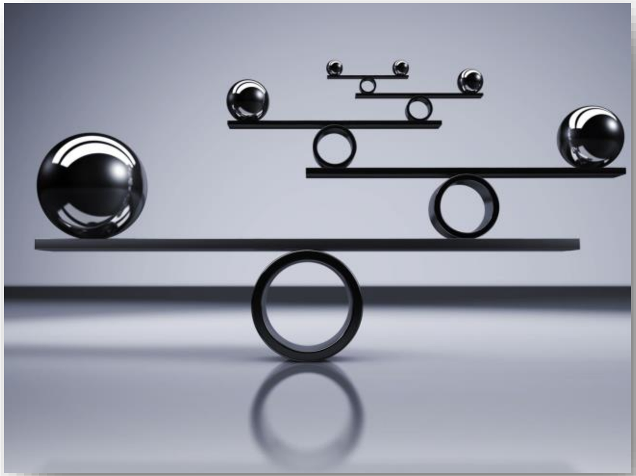
```
int x;
x = foo(3,0);
x = foo(x+1,1);
```

Mechanisms for exchanging information with external code

- Parameters
- Return value
- Nonlocal environment

Methods for passing parameters

- Three classes of parameters:
  - Input parameters
  - Output parameters
  - Input/output parameters
- Two principal methods
  - By value
  - By reference (or variable)



Call by value

Call by value

- The value is the actual one (r-value) assigned to the formal parameter, that is treated like a local variable
- Transmission from `main` to `proc` ⇨
- Modifications to the formal parameter do not affect the actual one
- On procedure termination, the formal parameter is destroyed (together with the local environment)
- No way to be used to transfer information from the callee to the caller!

An example

```
int y = 1;
void foo (int x) {
    x = x+1;
}
...
y = 1;
foo(y+1);
```

x assumes the initial value 2

x is incremented to 3

x is destroyed

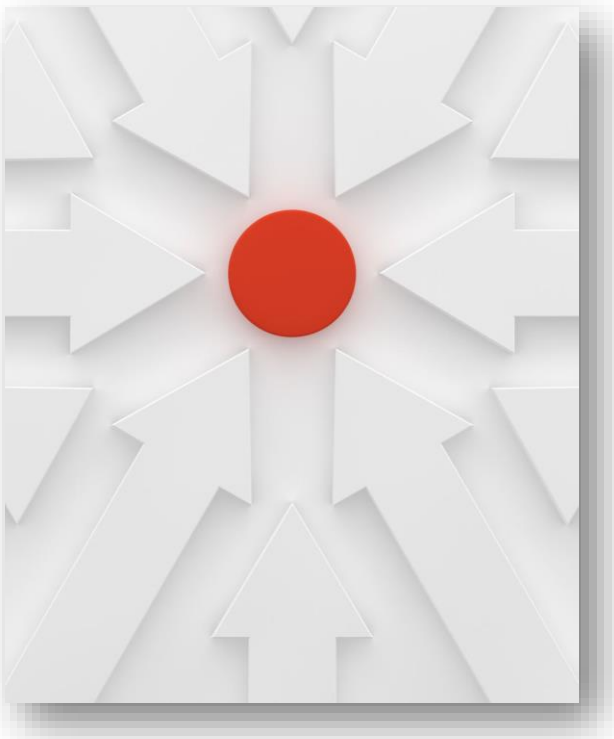
y+1 is evaluated, and its value assigned to x

y is still 1

- The formal parameter `x` is a local variable
- There is no link between `x` in the body of `foo` and `y` (`y` never changes its value)
- On exit from `foo`, `x` is destroyed
- It is not possible to transmit data from `foo` via the parameter

Some considerations

- Expensive for large amounts of data, as they must be copied
- Cheap in terms of cost of accessing the formal parameter (same as accessing local variable)
- Used in Java, Pascal, ML and C

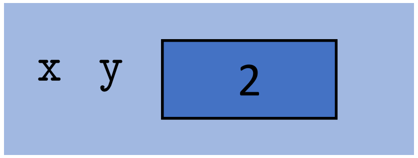


Call by reference

Call by reference (or variable)

- A reference (address) to the actual parameter (an expression with l-value) is passed to the function
- **The actual parameter must be an expression with l-value**
- References to the formal parameter are references to the actual one (aliasing)
- Transmission from and to `main` and `proc` ⇔
- Modifications to the formal parameter are transferred to the actual one
- On procedure termination the link between formal and actual is destroyed

An example




```
int y = 1;
void foo (reference int x) {
    x = x+1;
}
...
y = 1;
foo(y);
```

x is another name for y

x is incremented to 2

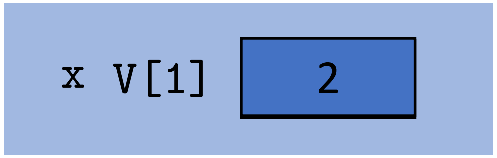
x and its link with y are destroyed

a reference is passed

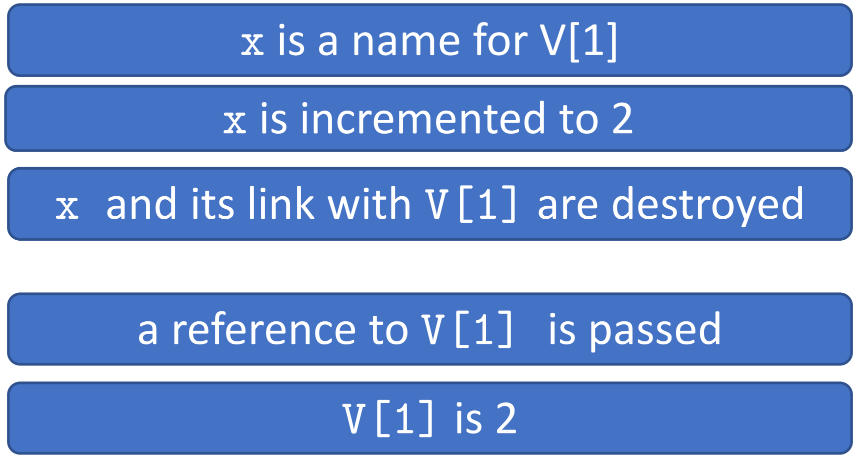
y is 2

- A reference (address, pointer) is passed
- `x` is an alias of `y`
- The actual value is an l-value
- On exit from `foo`, the link between `x` and the address of `y` is destroyed
- Transmission: Two-way between `foo` and the caller

Another example




```
int[] V = new V[10];
int i=0;
void foo (reference int x) {
    x = x+1;
}
...
V[1] = 1;
foo(V[i+1]);
```



- The actual parameter does not necessarily need to be a variable but can also be an expression
- In case of an expression the l-value is evaluated at call time
- During the calling sequence, the l-value of the actual parameter is stored in the activation record of the function

Some considerations

- Cheap in terms of storing (only an address need to be stored)
- Indirect access which can be implemented at a low cost

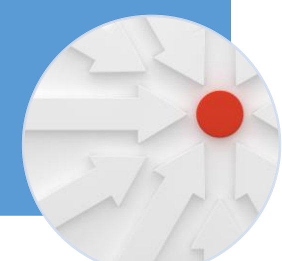
Call by value vs call by reference

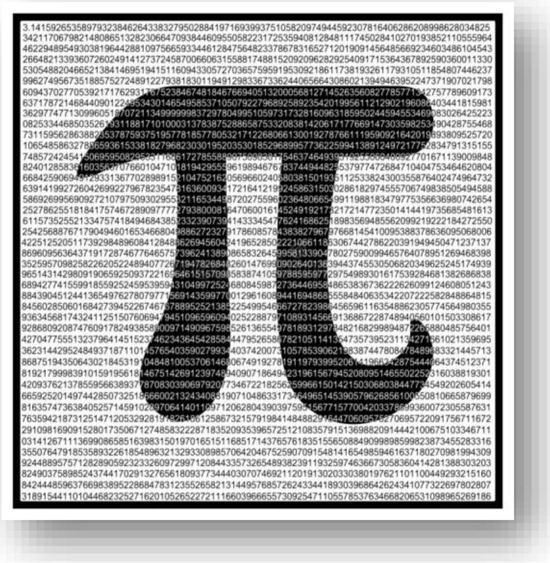
- Simple semantics. The body of the procedure does not need to know how the procedure was called (referential transparency)
- Implementation fairly simple
- Call could be expensive due to copy operations
- Need for other mechanisms to communicate with the called procedure

Call by value

- Complicated semantics: aliasing
- Simple implementation
- Call is efficient
- Reference to formal parameter slightly more expensive

Call by reference





Call by constant

Call by constant/read -only

- Call-by-value could be expensive
  - Large data items are copied even if they are not modified
  - If they are not modified, we can keep the semantics of passing by value, implementing it via call by reference
- Read-only parameter method: ⇨
  - Procedures are not allowed to change the value of the formal parameter (could be statically controlled by the compiler)
  - Implementation could be at the discretion of the compiler (“large” parameters passed by reference, “small" by value)
  - It can be thought as a sort of annotation
  - In Java: `final`

```
void foo (final int x){ //x cannot be modified
```

  - In C/C++: `const`

Call by result

Call by result

- The actual parameter is an expression that evaluates to an l-value
- No link between the formal and the actual parameter in the body
- The local environment is extended between an association between the formal parameter and a new variable
- When the procedure terminates, the value of the formal parameter is assigned to the location corresponding to l-value of the actual parameter
- Output-only communication: no way to communicate from `main` to `proc` ⇦

An example

```
void foo (result int x) {
     x = 8;
}
...
y = 1;
foo(y);
```


- Dual of the call by value
- No link between `x` and `y` in the body of `foo`
- When `foo` ends, the value of `x` is assigned to the location obtained with the l-value of `y`
- It is important when the l-value of `y` is determined (when the function is called or when it terminates)

Some considerations

- Same as the call by value: Large copying cost for large data
- Good for functions that must return more than one value

Call by value result

Call by value-result

- Bidirectional communication using the formal parameter as a local variable ⇔
- The actual parameter must be an expression that can yield an l-value
- At the call, the actual parameter is evaluated and the r-value assigned to the formal parameter.
- At the end of the procedure, the value of the formal parameter is assigned to the location corresponding to the actual parameter

An example

```
void foo (value-result int x) {
    x = x+1;
}
...
y = 8;
foo(y);
```

x is a local variable

the value of y assigned to x

x = 9

the value of x assigned to y

x is destroyed

y is 9

- No link between `x` and `y` in the body of `foo`
- When `foo` ends, the value of `x` is assigned to the location obtained with the l-value of `y`

Some considerations

- Large copying cost for large data

Call by name

Call by name

- Aim: give a precise semantics to parameter passing
- Copy-rule mechanism of the actual parameter to the formal on
- A call to P is the same as executing the body of P after substituting the actual parameters for the formal one
- “Macro expansion", implemented in a semantically correct way: every time the formal parameter appears we re-evaluate the actual one
- Input and output parameters ⇔
- Appears to be simple but … it is not that simple: it has to deal with variables with the same name
- No longer used by any imperative language

An example

```
int x=0;
int foo (name int y){
     int x = 2;
     return x + y;
}
...
int a = foo(x+1);
```

- Blindly applying the copy rule would lead us to a result of x+x+1=5
- Incorrect result as it would depend on the name of the local variable
- With a body `{int z = 2; return z + y;}` the result would have been z+x+1=3

- When the body contains the same name of the actual parameter, we say that it is captured by the local declaration
- In order to avoid substitutions in which the actual parameter is captured by the local declaration, we impose that the formal parameter – even after the substitution – is evaluated in the environment of the caller and not of the callee
- Substitute the actual parameter together with its evaluation environment – fixed at the time of the call

Actual parameter evaluation

```
int y;
void fie (int x){
     int y;
     x = x + 1; y = 0;
}
...
y = 1;
fie(y);
```

x is y (external)

x is 2

y (local) is 0

y is 2

- A pair <exp,env> is passed, where
  - exp is the actual parameter, not evaluated
  - env is the evaluation environment
- Every time the formula is used, exp is evaluated in env

Actual parameter evaluation

```
int i = 2;
int fie (name int y) {
    return y+y;
}
...
int a = fie(i++);
```

The first time i is 2

The second time i is 3

- `i++` means returning the current value of `i` and then incrementing the value of the variable by 1
- When `fie` is called, `i` has to be evaluated twice

after `fie`, `i` is 4 and `a` is 5

- The actual parameter must be evaluated every time the formal parameter is encountered

Call by name vs call by value-result

```
void fiefoo (valueresult int x,
valueresult int y) {
     x = x+1;
     y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);
```

call- by value-result

x is 1, y is A[1]

x is 2

y is 1

i is 2, A[1] is 1

```
void fiefoo (name int x,
name int y) {
     x = x+1;
     y = 1;
}
...
int i = 1;
int[] A = new int[5];
A[1]=4;
fiefoo(i,A[i]);
```

call- by name

x is i, y is A[i]

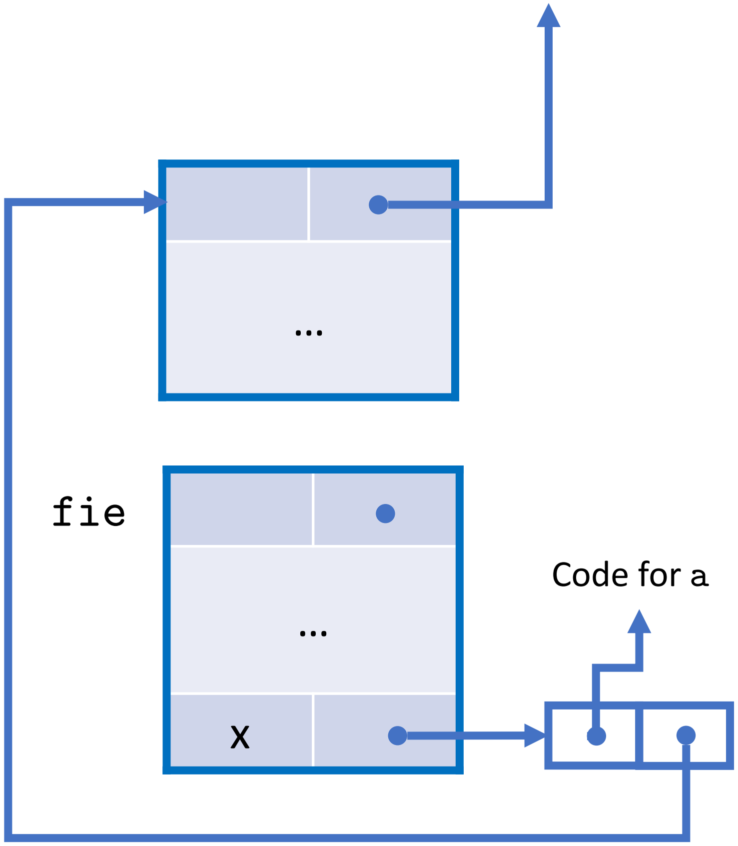
x is 2

y is 1

i is 2, A[1] is 4, A[2]=1

Some considerations

- More expensive, as the whole environment must be passed
- How do we pass the pair `<exp,env>` (closure)?
  - A pointer to the text of `exp`
  - A pointer to the activation record of the calling block
- This lets us pass functions as arguments to other procedures



To sum up

- It supports input and output parameters
- The actual parameter can be an arbitrary expression but has to evaluate to an l-value if the formal parameter is used at the left of an assignment
- It can happen that actual and formal are aliased
- The actual parameter is evaluated every time the formal parameter occurs
- The environment is extended with an association between the formal and a closure – composed of the actual parameter and the environment in which the call occurs

Summary

- Abstraction of control
- Methods of parameter passing
- Higher-order functions
  - Functions as parameters
  - Functions as results

Readings

- Chapter 6 and 7 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Exception handling
- Data structures