

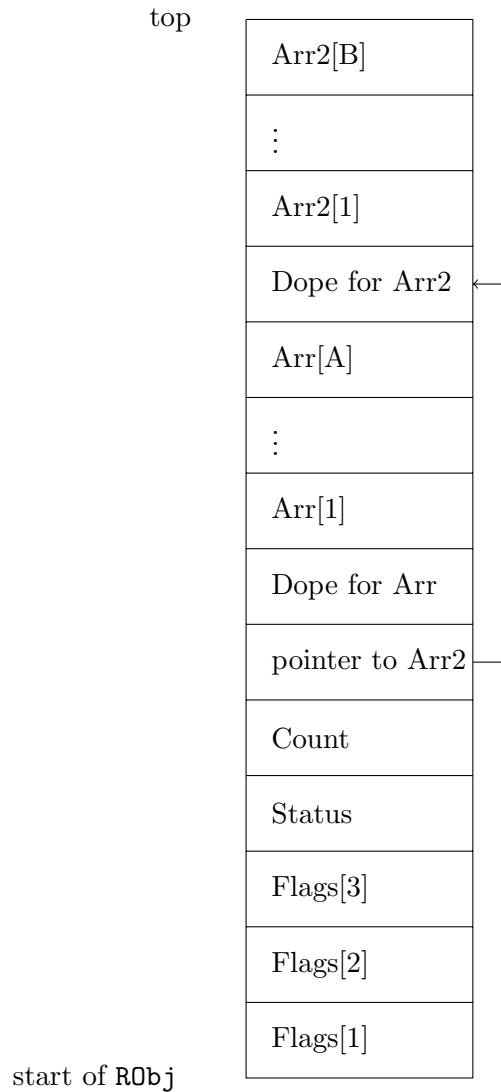
# Concepts of Programming Languages (INFOTECH)

Summer Semester 2006  
Steffen Keul, Stefan Staiger

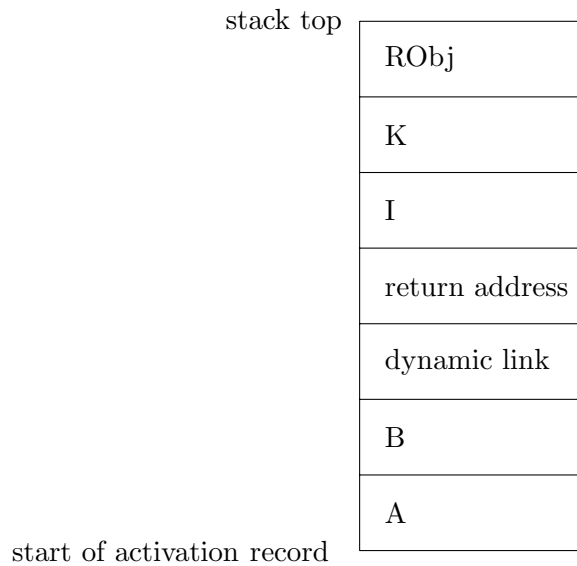
## Solutions to Assignment #1

### 1 Record and array layout

- At what time is the size of the variable `RObj` determined?
  - The size of `RObj` is determined when the procedure `Test` is called. It cannot be determined earlier since we need the values of `A` and `B`.
- Describe the layout of the record `RObj`. Choose one possible way to represent the arrays inside `RObj`. You are allowed to use different representations for each of the arrays.
  - Rule: First come the components of statically known size to allow efficient access. Other components are placed at the end of the record. **Note:** Some languages don't allow such an reordering.
  - We use padding to respect the machine's alignment restrictions.
  - Booleans can be packed together in a single word, but remember that this might cost some additional instructions to read or modify one of them.
  - Since the component size and number of elements for `Flags` is statically known, we can omit the dope for this array. On the other hand, for `Arr` and `Arr2`, we need a dope due to the unknown number of elements.
  - The exact contents of the dope depend on the language features which must be supported. For example, the dope in Ada could contain lower and upper bound of the possible array indices, and perhaps also the size each component has ("stride").
  - In our example below, the dope is stored directly in front of the array elements. We introduce an artificial pointer to the beginning of `Arr2`, although that address could also be found within the dope for `Arr`. This way, we save one add instruction for every access to a member of `Arr2`.
  - Here's a possible layout of the record `RObj`:

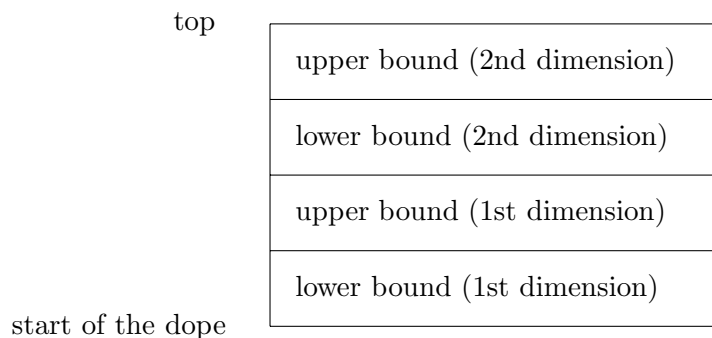


- Describe the layout of the activation record (on the stack) in detail.
  - The activation record typically contains the return address, the parameters and the local variables. Depending on the language, it sometimes also contains dynamic (and static) links to activation records of other procedures.
  - The exact order in which the parameters are passed normally depends on the underlying system. In our example, we pass parameters left-to-right.
  - Again, we arrange the components in such a way that all components of statically known size come first.
  - The layout of RObj is as described above, so we don't repeat the details here.

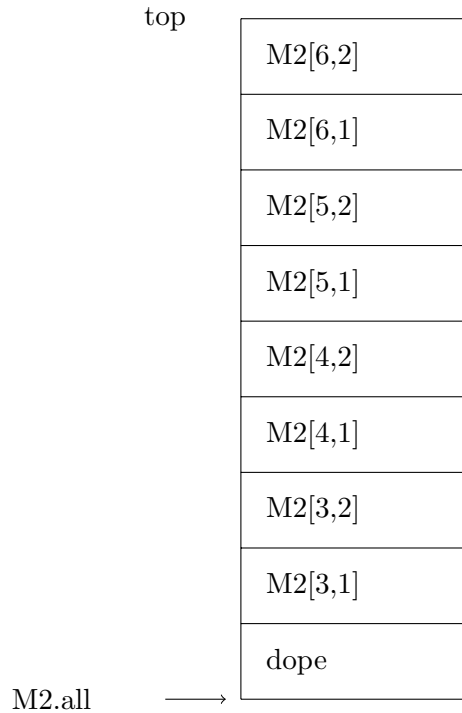


## 2 Passing arrays to procedures

- Where are the contents of M1 stored in procedure `Main`? On the stack or on the heap? Where are the contents of M2 stored?
  - The contents of M1 are stored on the stack, but the components of M2 are placed on the heap because we used the allocator `new`. Note that the pointer to these heap locations will be stored on the stack. The dope for M2 can be stored on the heap along with the elements (used here) or together with the pointer.
- For the access to M in the procedure `Do_Something` the array dope is needed. Why?
  - We need the dope to calculate the address at which we find the correct element. Since the type `Matrix` allows arguments to have a statically unknown size in the first dimension, we cannot calculate that address without the information passed in the dope.
- Show the contents of the dope. Show how the elements of `M2.all` are arranged in memory, assuming row-major array layout.
  - The dope must tell us lower and upper bounds for both dimensions. The component size and the number of dimensions are statically known and thus we don't need these values in the dope. So the dope could look like this:



- row major: arrange the elements row by row, that is, first all elements of the first row, then all elements of the second row and so on.
- Therefore, the elements of `M2.all` are arranged like this:



- How is the `dope` passed to the procedure `Do_Something`?
  - When the procedure `Do_Something` is called, an activation record is allocated on the stack (cf. exercise 1). That activation record contains a pointer to the heap location where we find the `dope` and the elements (as seen in the picture above).
  - If we instead use the alternative way and store the `dope` together with the pointer, the `dope` is contained in the activation record on the stack.

### 3 Recursive procedure

- How many times is the procedure `Recursive` entered for one execution of the main program? In what order are the assignment statements executed during one execution of the whole program?
  - The program executes in the following order:
    1. main program starts with the call to `Recursive (False)`
    2. assignment 3 is executed
    3. control is transferred back to the main program
    4. call to `Recursive (True)`
    5. assignment 1 is executed
    6. call to `Recursive (False)`
    7. assignment 3 is executed
    8. control is transferred back to `Recursive (True)` (from step 4)
    9. assignment 2 is executed
    10. control is transferred back to the main program

- Thus `Recursive` is entered three times, and the assignments are executed in the order 3 – 1 – 3 – 2
- What value of `X` is used when assignment 2 is executed?
  - The value 3 is used, since `X` is a local variable and thus cannot be changed by other instances of `Recursive` (or other procedures).
- List all the changes of the stack while the program executes. Include changes to variables and changes of the stack pointer.
  - We describe the relative changes to the previous state and omit dynamic links:

1. *main program starts with the call to `Recursive (False)`;*  
The activation record for `Recursive` is added to the stack:

stack top

X
return address
Condition (False)

2. *assignment 3 is executed;*  
The value 7 is stored in the location for `X`
3. *control is transferred back to the main program;*  
The activation record created in step 1 is removed
4. *call to `Recursive (True)`;*  
A new activation record is created and added to the stack:

stack top

X
return address
Condition (True)

5. *assignment 1 is executed;*  
The value 3 is stored in the location for `X`
6. *call to `Recursive (False)`;*  
A second activation record is added to the stack; thus the stack now looks like this:

stack top

X
return address
Condition (False)
X (3)
return address
Condition (True)

7. *assignment 3 is executed;*

The value 7 is stored in the top-most X which belongs to the current instance of **Recursive**

8. *control is transfered back to Recursive (True);*

The second activation record from step 6 is removed. The stack now looks like this:

stack top

X (3)
return address
Condition (True)

9. *assignment 2 is executed;*

has no impact on the stack (as far as we can tell here with the dots)

10. *control is transfered back to the main program;*

The activation record from step 4 is removed; thus, the stack now looks the same he did at the beginning.

- Show the call stack when assignment 2 is executed for the first time.

– We can easily derive this from the previous observations:

stack top

X (3)
return address
Condition (True)