TDDD55 Compilers and Interpreters

TDDB44 Compiler Construction

LINKÖPINGS UNIVERSITET

Memory Management and Run-Time Systems

Adrian Pop, Martin Sjölund, Peter Fritzson
PELAB/IDA, Linköpings universitet, 2022

Run-Time Systems Support Program Execution

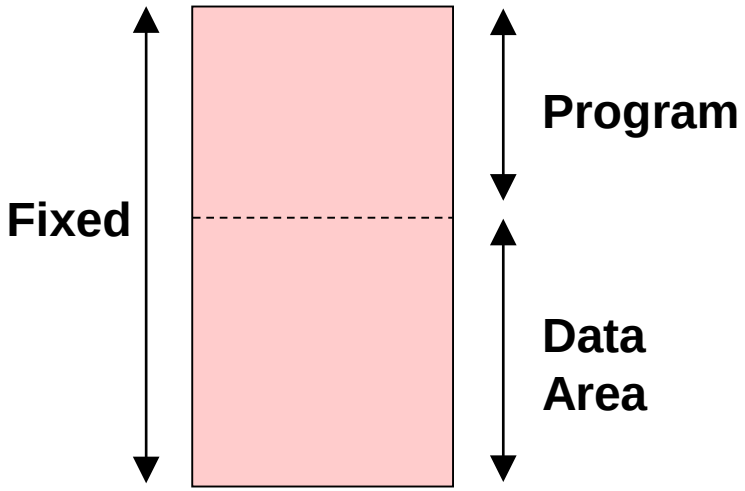
LINKÖPINGS UNIVERSITET

- Memory management of a program during execution.
  This includes *allocation* and *de-allocation* of memory cells.
- Address calculation for variable references.
- For references to non-local data, finding the right object taking scope into consideration.
- Recursion, which means that several instances of the same procedure are active (activations of a procedure) at the same time during execution.
- Dynamic language constructs, such as dynamic arrays, pointer structures, etc.
- Different sorts of parameter transfer

Two different memory management strategies: **static** and **dynamic** memory management, determined by the language to be executed.

Static Memory Management

- All data and its size must be known during compilation, i.e. the memory space needed during execution is known at compile-time.
- The underlying language has no recursion.
- Data is referenced to by absolute addresses.



- Static memory management needs no run-time support, because everything about memory management can be decided during compilation.
- An example of such a language is FORTRAN77, whereas FORTRAN90 has recursion.

Dynamic Memory Management (1)

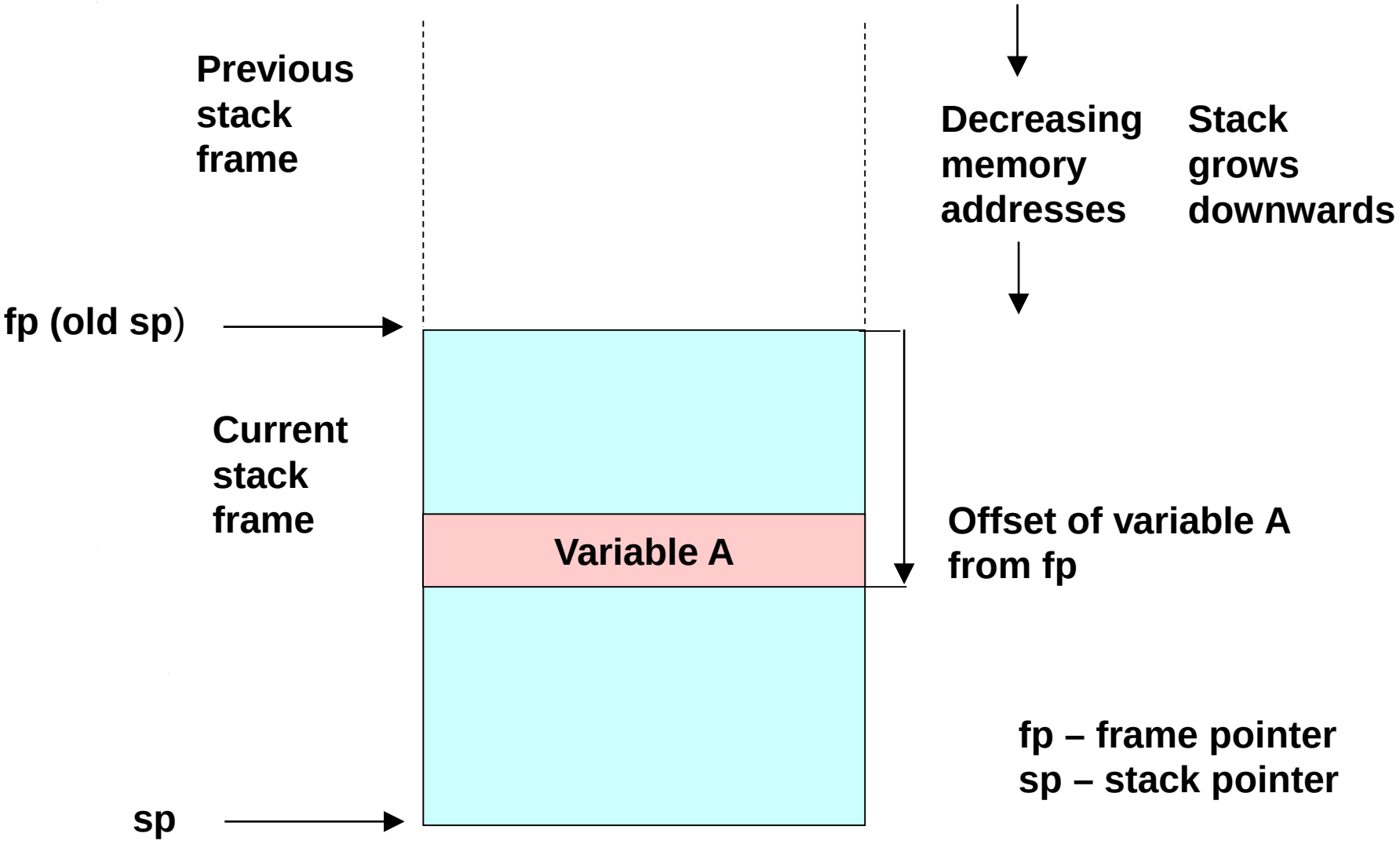
- Data size is not known at compiler time (e.g. dynamic arrays, pointer structures)
- There is recursion
- Examples of such languages are: Pascal, C, Algol, Java, … Basically most of the general-purpose programming languages

Dynamic Memory Management (2)
Run-Time Support

Run-Time support is needed for languages with dynamic memory management:

- The call chain must be stored somewhere and references to non-local variables must be dealt with.
- Variables cannot be referenced by absolute addresses, but by *<blockno, offset>*.
- All data belonging to a block (procedure) is gathered together in an *activation record (stack frame*).
- At a procedure call memory is allocated on the stack and each call involves constructing an activation record.

A Stack Frame with Frame and Stack Pointers



Some Concepts (Rep.)

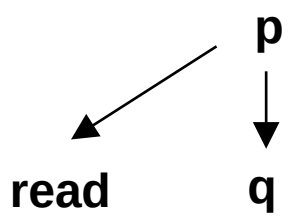
- **Activation**
  - Each call (execution) of a procedure is known as **activation** of the procedure.
- **Life span of an activation**
  - The life span of an activation of a procedure p lasts from the execution's first statement to the last statement in p's procedure body.
- **Recursive procedure**
  - A procedure is recursive if it can be activated again during the life span of the previous activation.
- **Activation tree**
  - An activation tree shows how procedures are activated and terminated during an execution of a program.
  - Note that a program can have different activation trees in different executions.
- **Call chain**
  - All current activations (ordered by activation time)
  - - a path in the activation tree
  - - a sequence of procedure frames on the run-time stack

Example of Activation Tree (Rep.)

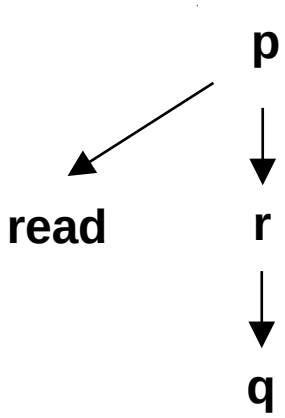
```
program p;
  procedure q;
    ...
    end (* q *);

    procedure r;
      ...
      q;
    end (* r *);

begin (* p *)
    read(x);
    if x = 0
      then q;
      else r;
end (* p *);
```

Two different activation trees for the program:

Activation tree when x=0



Activation tree when x≠0



Formal and Actual Parameters (Rep.)

- Arguments declared in the head of a procedure declaration are its **formal** parameters and arguments in the procedure call are its **actual** parameters.

- In the example below:
  `i` is a formal parameter
  `k` is an actual parameter

```
procedure A(i: integer);
begin (* A *)
   ...
   A(k);
   ...
end (* A *);
```

Activation Record

All information which is needed for an activation of a procedure is put in a record which is called an *activation record*. The activation record remains on the stack during the life span of the procedure.

An activation record contains:

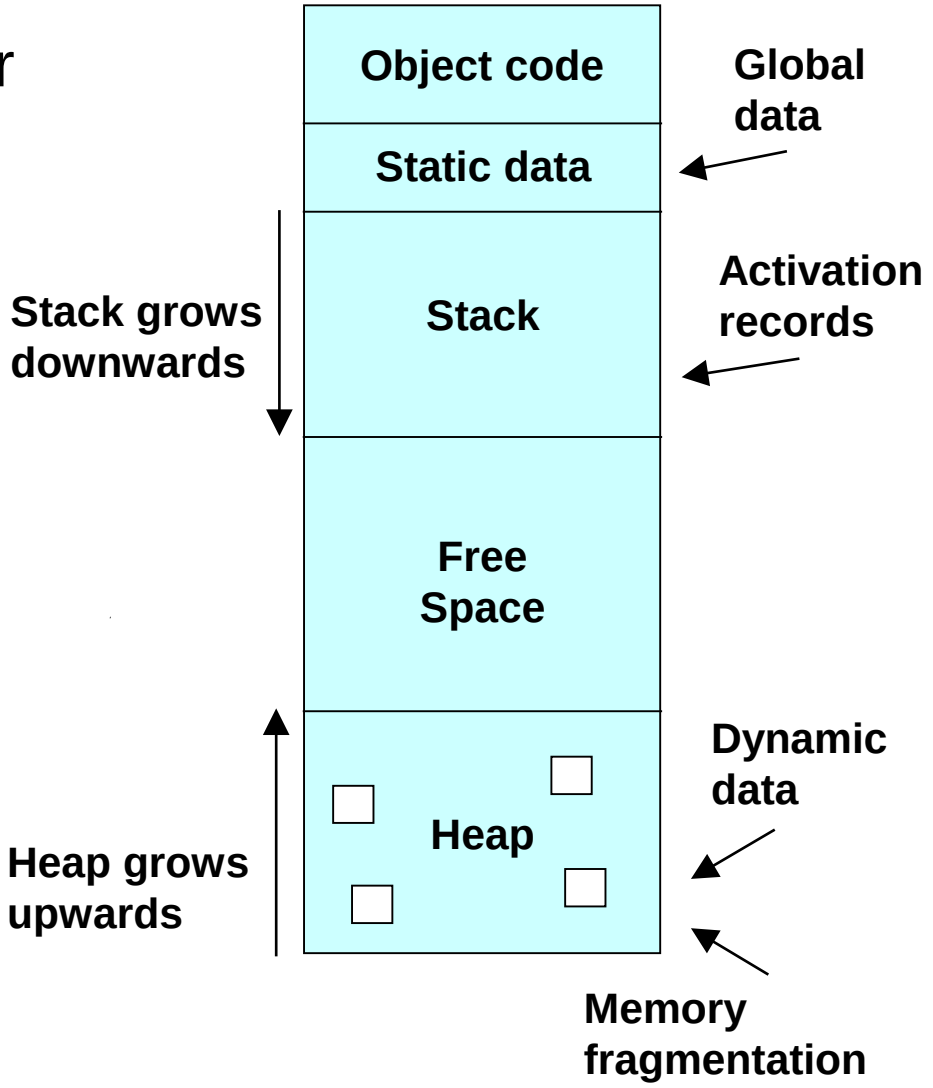
- Local and temporary data
- Return address
- Parameters

```
procedure p1
    var A: (* ... *)
    procedure p2
        (* reference to A *)
    end (* p2 *)
end (* p1 *)
```

- Pointers to previous activation records (*dynamic link*, *control link*)
- *Static link* (*access link*) or *display* for finding the correct references to non-local data (e.g. in enclosing scopes)
- Dynamically allocated data (*dope vectors*)
- Space for a return value (where needed)
- Space for saving the contents of registers

Typical Memory Organization (Pascal/Java-like language)

- **Static data**
  - The memory requirement for data objects must be known at compile time and the address to these objects is not changed during execution, so the addresses can be hard-coded in the object code.
- **Stack**
  - Space for activation records is allocated for each new activation of procedures.
- **Heap**
  - Allocation when necessary.



How are non-local variables referenced?

- Static link (access link)
- Display

Example:

```
program prog;            (* Block B0,predefined vars)
var a, b, c: integer;    (* Block B1,  Globals *)
procedure p1;
  var b, c: real;        (* Block B2 *)
  procedure p2;
    var c: real;         (* Block B3 *)
    begin
      c := b + a;        (* B3.c := B2.b + B1.a *)
    end (* p2 *);
  begin
    p2;
  end (* p1);
begin
  p1;
end (* prog *).
```

In the procedures the variables are referenced using <blockno, offset>:

```
B3.c := B2.b + B1.a
```

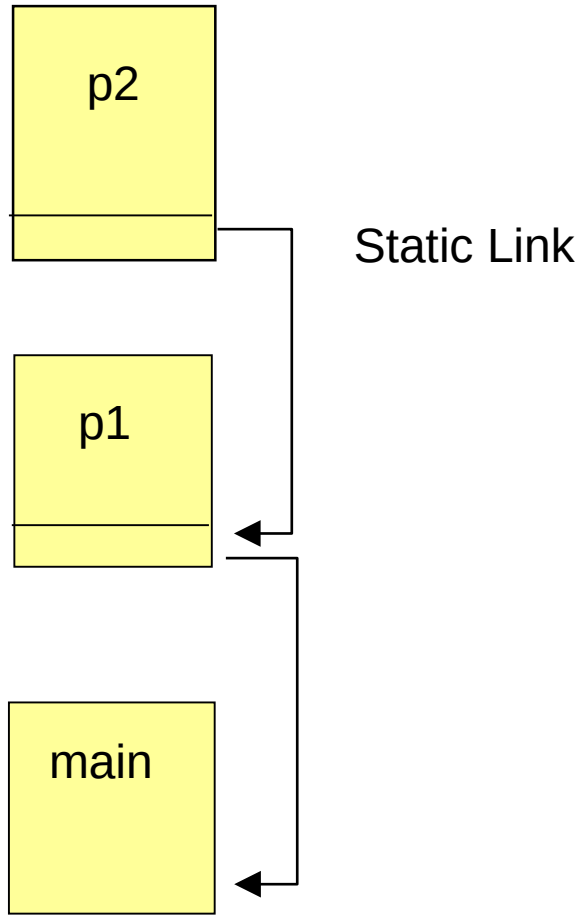
or by using relative blocknumber:

```
0.c := 1.b + 2.a
```

(0: current block, 1: nearest surrounding block, etc.)

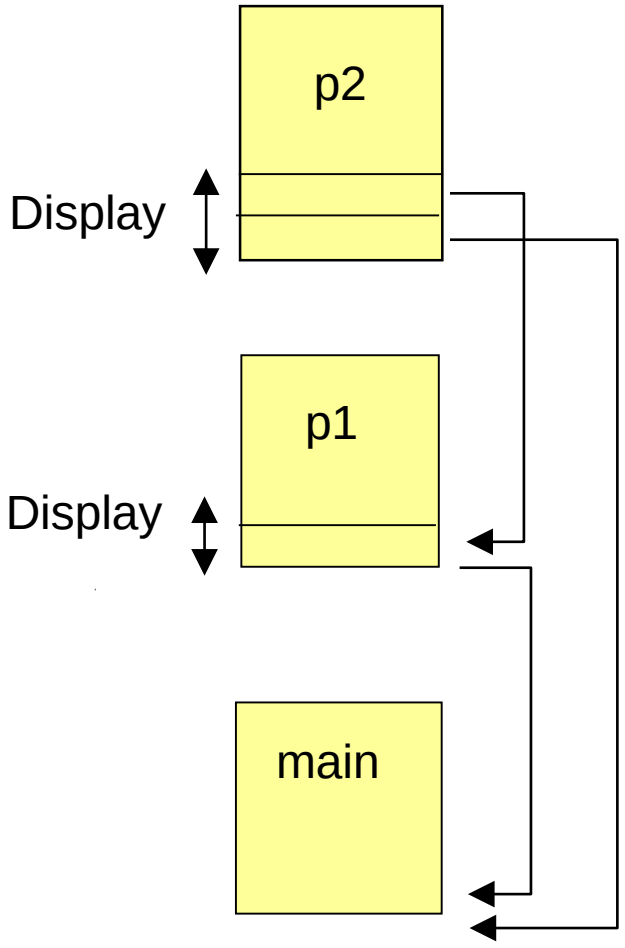
Non-local references through Static Link

- The *static link* is a pointer to the most recent activation record for the **textually surrounding** block
- Example. Use relative block number for statement inside procedure p2:
  `0.c := 1.b + 2.a`
  For variable a follow the static link 2 steps.
- This method is practical and uses little space. With deeply nested procedures it will be slow.



Non-local references through Display

- Display is a table with pointers (addresses) to surrounding procedures' activation records.
- The display can be stored in the activation records.
- Display is faster than static link for deep nesting, but requires more space.
- Display can be slightly slower than static link for very shallow nesting.



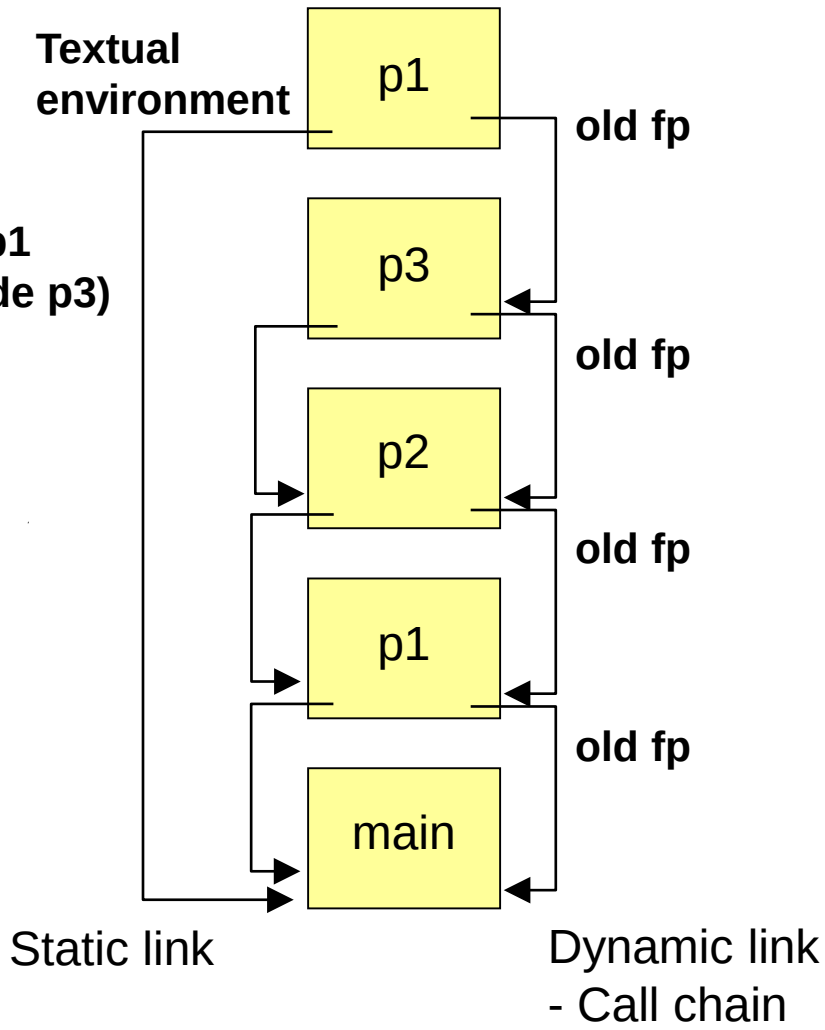
Dynamic Link, i.e., Control Link

- Dynamic link specifies the call chain
- Not the same as static link if there is a recursive call chain, e.g.

```
program foo;
  procedure p1;
    procedure p2
      procedure p3;
      begin (* p3*)
          p1;
          ...
      end (* p3 *);
    begin (* p2 *)
        p3;
        ...
    end (* p2 *)
  begin (* p1 *)
    p2;
    ...;
  end (* p1)
begin (* main *)
  p1;
end (* main *)
```

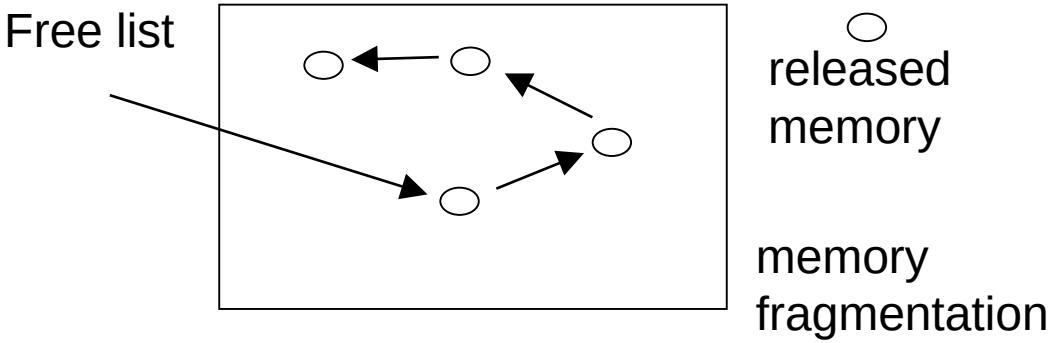
The stack at 2nd call for p1:

(On return from p1 we continue inside p3)



Heap Allocation (Rep.)

- In some languages data can dynamically be built during execution and its size is not known (e.g. strings of variable length, lists, tree structures, etc).
- Manual memory management
  - De-allocation is **not** performed automatically as in stack allocation. Hard work, can lead to bugs.
  - Pascal: **new(p)** (*allocation*) **dispose(p)** (* deallocation*)
    C: **p=malloc()** (*allocation*) **free(p)** (* deallocation*)
- Automatic memory management, with garbage collection (e.g. Lisp, Java)
  - De-allocation is automatic. Resource-consuming, but avoids bugs.



After memory compaction:

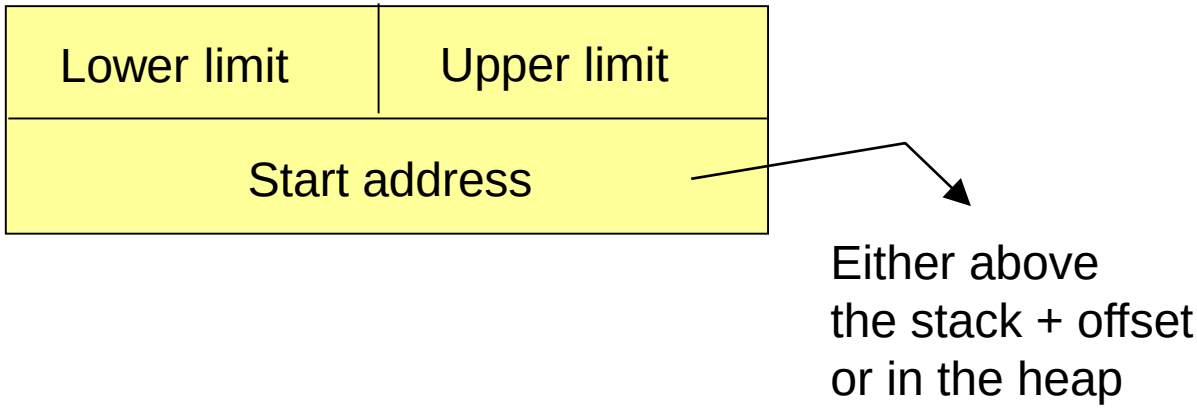
Data Storage and Referencing

- **Where is data stored and how is it referenced?**
  - (Semi-static) Static data can be allocated directly (consecutive in the activation record, data area).
  - Data is referenced by <*blockno, offset*>.
    blockno is specified as **nesting depth**.
- **Simple variables (boolean, integer, real ...)**
  - These have a fixed size and are put directly into the activation record, or in registers.
- **Static arrays**
  - Fixed number of elements, i.e. size is known at compile time.
    Example: `A: array[1..100] of integer;`
  - Stored directly in the activation record.

Dynamically Allocated Arrays

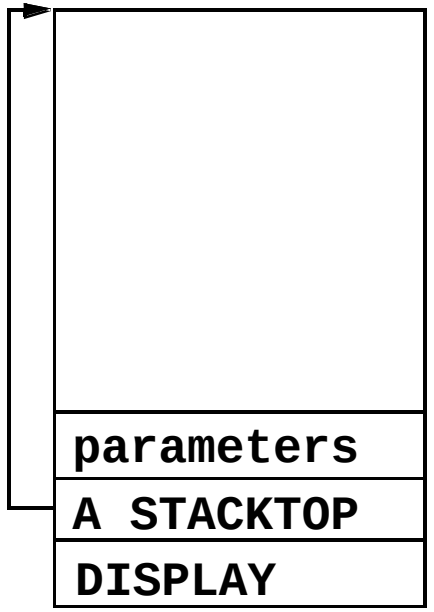
- The size is unknown at compile time:
  - Example: **`B: array[1..max] of integer;`**
  - **`max`** not known at compile time.
- *Dope vector* (*data descriptor*) is used for dynamically allocated arrays. Dope vectors are stored in the activation record.

Dope vector:

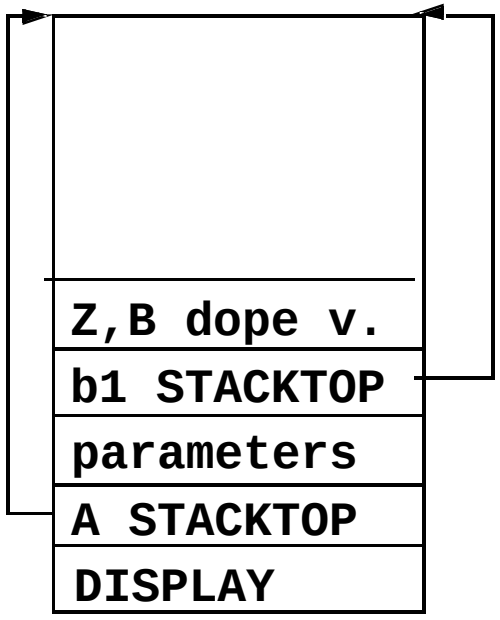


Dynamic Arrays and Block Structures in ALGOL (1)

```
PROCEDURE A(X,Y); INTEGER X, Y;
 L1: BEGIN  REAL Z;                    (block b1)
            ARRAY B[X:Y];
        L2: BEGIN REAL D,E;
        L3:    •••
               END;

        L4: BEGIN ARRAY A[1:X];
                L5: BEGIN REAL E;
                L6:    •••
                       END;
        L7: END;
L8: END;
```

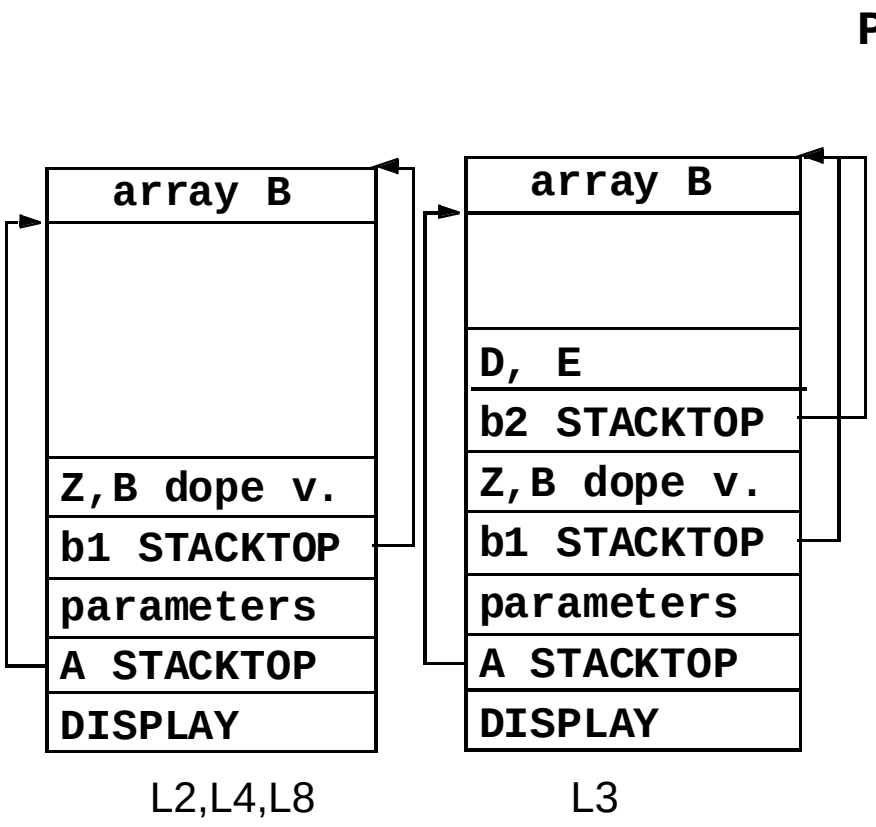


before L1



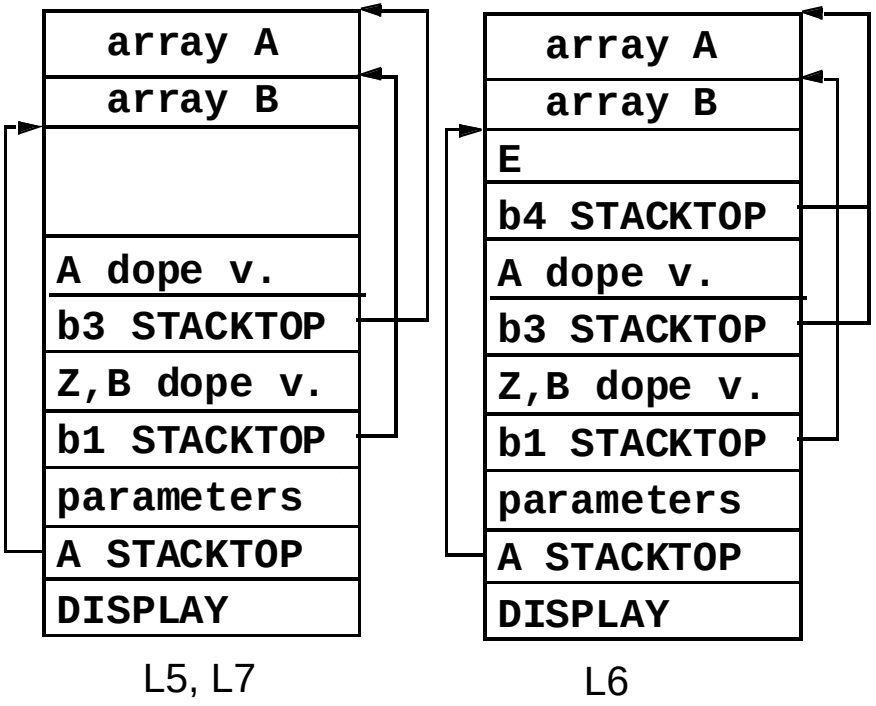
before L2

Dynamic Arrays and Block Structures in ALGOL (2)




```
PROCEDURE A(X,Y); INTEGER X, Y;
L1: BEGIN REAL Z;                          (block b1)
      ARRAY B[X:Y];
    L2:BEGIN REAL D,E;
    L3:    •••
       END;

    L4:BEGIN ARRAY A[1:X];
          L5:BEGIN REAL E;
          L6:    •••
             END;
    L7: END;
L8: END;
```

Dynamic Arrays and Block Structures in ALGOL (3)




```
PROCEDURE A(X,Y); INTEGER X,Y;
L1: BEGIN REAL Z;          (block B1:)
      ARRAY B[X:Y];
    L2:BEGIN REAL D,E;
    L3:   •••
       END;
    L4:BEGIN ARRAY A[1:X];
          L5:BEGIN REAL E;
          L6:   •••
             END;
    L7: END;
L8: END;
```

Parameter Passing (1) (Rep.)
Call by Reference

- There are different ways of passing parameters in different programming languages. Here are four of the most common methods:

- *1. Call by reference (Call by location)*
  - The address to the actual parameter, *l-value*, is passed to the called routine's AR
  - The actual parameter's value can be changed.
  - Causes aliasing.
  - The actual parameter must have an l-value.
- Example: Pascal's **VAR** parameters, reference parameters in C++. In Fortran, this is the only kind of parameter.

Parameter Passing (2) (Rep.)
Call by Value

- *2. Call by value*
  - The value of the actual parameter is passed
  - The actual parameter cannot change value

- Example: Pascal's non-**VAR** parameters, found in most languages (e.g. C, C++, Java)

Parameter Passing (3) (Rep.)
Call by value-result (hybrid)

- *3. Call by value-result* (hybrid)
  - The value of the actual parameter is calculated by the calling procedure and is copied to AR for the called procedure.
  - The actual parameter's value is not affected during execution of the called procedure.
  - At return the value of the formal parameter is copied to the actual parameter, if the actual parameter has an l-value (e.g. is a variable).

- Found in Ada.

Parameter Passing (4) (Rep.)
Call by Name

- *4. Call by name*
  - Similar to macro definitions
  - No values calculated or passed
  - The whole expression of the parameter is passed as a procedure without parameters, a *thunk*.
  - Calculating the expression is performed by evaluating the thunk each time there is a reference to the parameter.
  - Some unpleasant effects, but also general/powerful.

- Found in Algol, Mathematica, Lazy functional languages

Example of Using the Four Parameter Passing Methods: (Rep.)

```
procedure swap(x, y : integer);
var temp : integer;
begin
  temp := x;
  x := y;
  y := temp;
end (* swap *);
```

```
...
i := 1;
a[i] := 10; (* a: array[1..5]
                  of integer *)
print(i, a[i]);
swap(i, a[i]);
print(i, a[1]);
```

Results from the 4 parameter passing methods
Printouts from the print statements in the above example

	Call by reference	Call by value	Call by value-result	Call by name
print	1 10	1 10	1 10	1 10
swap	10 1	1 10	10 1	Error!

Reason for the Error in the Call-by-name Example

The following happens:

```
x = text('i');
y = text('a[i]');

temp := i;        (* => temp:=1 *)
i := a[i];        (* => i:=10 since a[i]=10 *)
a[i] := temp;     (* => a[10]:=1 =>   index out of bounds *)
```

Note: This error does not occur in lazy functional languages using call-by-name since side-effects are not allowed.

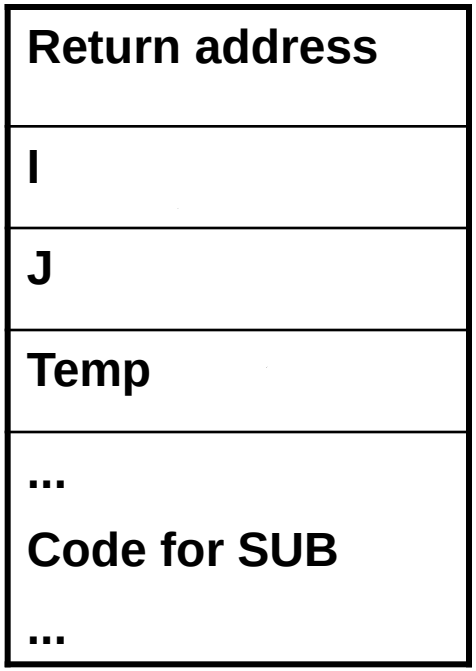
Static Memory Management
E.g. Fortran77 and (partly) CUDA/C on NVIDIA

- No procedure nesting, i.e., no block structure.
  - $\Rightarrow$ References to variables locally or globally.
  - $\Rightarrow$ No displays or static links needed.
- No recursion ($\Rightarrow$ stack not needed).
- All data are static ($\Rightarrow$ heap not needed).

- All memory is allocated **statically**
  - $\Rightarrow$ variables are referenced by absolute address.
  - The data area (i.e. the activation record) is often placed with the code
  - Inefficient for allocating space for objects which are perhaps used only a short time during execution.
  - But execution is efficient in that all addresses are placed and ready in the object code
  - Problematic for parallel code

Static Memory Allocation and Procedure Call/Return for Fortran77

```
SUBROUTINE SUB(J)
I = 1
J = I+3*J
END
```



- **At procedure call**

1. Put the addresses (or values) of the actual parameters in the data area.
2. Save register contents.
3. Put return address in the data area.
4. Execute the routine.
5. References to variables locally or globally.
6. No displays or static links needed.

- **On return:**

1. Reset the registers.
2. Jump back.

Memory management in Algol, Pascal, C, C++, Java

- **Language Properties**:
  - Nested procedures/blocks (PASCAL, ALGOL)
  - Dynamically allocated arrays (ALGOL, C99, C++, ...)
  - Recursion
  - Heap allocation (PASCAL, C, C++, Java*, ...)

- **Problems:**
  - References to non-local variables (solved by display or static link)
  - Call-by-name (ALGOL, Lazy Functional Languages)
  - Dynamic arrays (*dope vector*)
  - Procedures as parameters

Events when Procedure P Calls Q

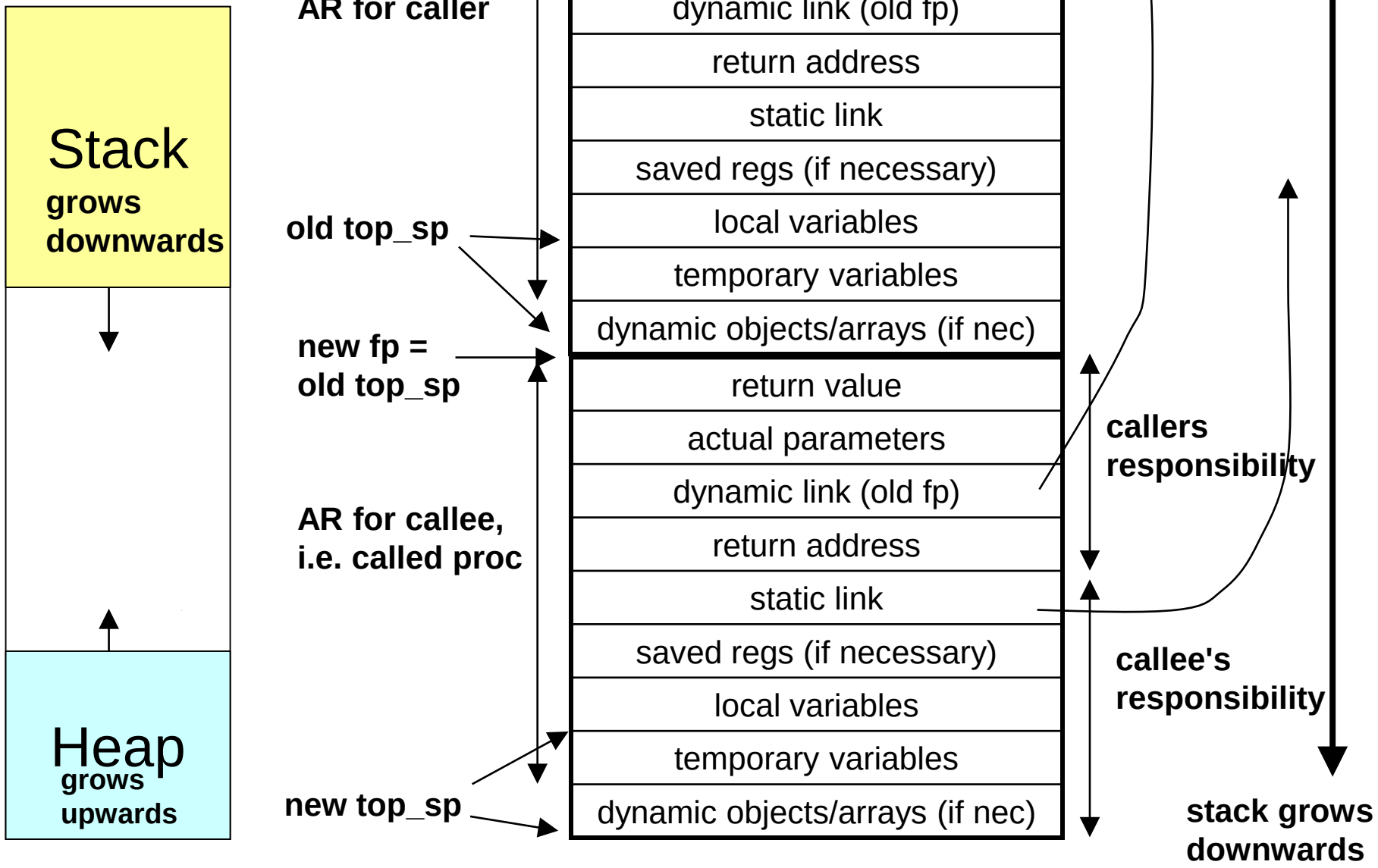
At call:

- P already has an AR (activation record) on the stack
- P's responsibility:
  - Allocate space for Q's AR.
  - Evaluate actual parameters and put them in Q's AR.
  - Save return address and dynamic links (i.e. top_sp) in new (Q's) AR.
  - Update (increment) top_sp.
- Q's responsibility:
  - Save register contents and other status info.
  - Initialise own local data and start to execute.

At return:

- Q's responsibility
  - Save return value in own AR (NB! P can access the return value after the jump).
  - Reset the dynamic link and register contents, ...
  - Q finishes with return to P's code.
- P's Responsibility
  - P collects the return value from Q, despite update of top_sp.

At Calls
Stack and Heap



Procedure Call/Return in Algol, Pascal, C, ...

At call:

1. Space for activation record is allocated on the stack.
2. Display / static link is set.
3. Move the actual parameters.
4. Save implicit parameters (e.g. registers).
5. Save return address.
6. Set dynamic link.
7. Execute the routine.

At return:

1. Reset dynamic link.
2. Reset the registers
3. Reset display / static link
4. Jump back.

Thank you!

- any questions?
- TDDB44 lab on Tue might be ran by somebody else
- next lecture L10 – will be a recording with questions in L11.