Memory management and run-time systems

Run-time systems can support the execution of a program in the form of:

- 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

There are two different memory management strategies:
static and **dynamic** memory management. The underlying language determines the method.

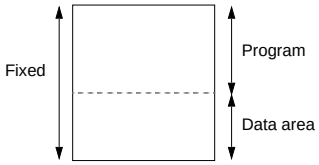
Static memory management

- 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 FORTRAN4, FORTRAN77.

FORTRAN90 has recursion.



Dynamic memory management

- Data size is not known at compile time (e.g. dynamic arrays, pointer structures)
- There is recursion

Examples of such languages are: PASCAL, C, ALGOL.

Run-time support is required for languages with dynamic memory management:

- The call chain must be stored somewhere and references to non-local variables must be dealt with.
- Variables can not 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.

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

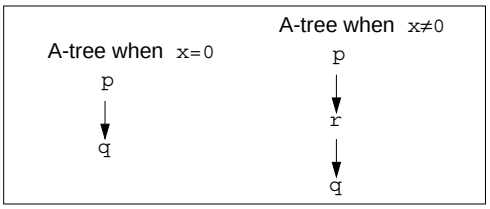
An activation tree shows how procedures are activated and terminated during an execution of a program.

Note that a program can have different activation trees in different executions.

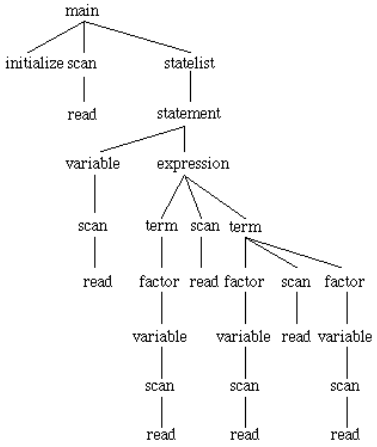
Example. Activation tree (Rep.)

```
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:



An activation tree for a recursive descent parser (e.g. in lab 2) can look like this: (Rep.)



Note that for certain input the tree shows the sequence of all procedure activations within the program. But at each point in time during execution it is just a straight path from the root to the actual procedure which shows all active procedures.

Formal and actual parameters (Rep.)

Arguments in the head of the 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

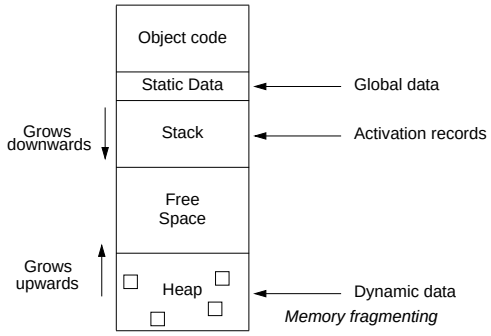
All information which is needed for an activation of a procedure is put in a record which is called an activation record.

The activation record remains on the stack during the life span of the procedure.

The activation record contains:

- Local data
- Temporary data
- Return address
- Parameters
- Pointers to previous activation records (*dynamic link*, *control link*)
- *Static link* (*access link*) or *display* for finding the correct references to non-local data
- Dynamically allocated data (*dope vectors*)
- Space for a return value (where needed)
- Space for saving the contents of a register

Typical memory organisation (Pascal-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?

1. Static link (*access link*)
2. Display

Example

```
program prog;                    { "Block0" }
var a, b, c : integer;           { Block1 }
  procedure p1;
  var b, c : real;               { Block 2 }
      procedure p2;
      var c : real;              { Block 3 }
      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

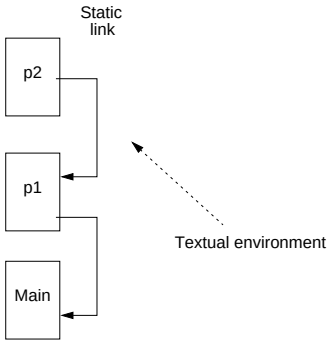
<blockno, offset>: `B3.c := B2.b + B1.a`

or by using relative blockno: `0.c := 1.b + 2.a`

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

The stack and references when `p2` executes if we use:

Static link



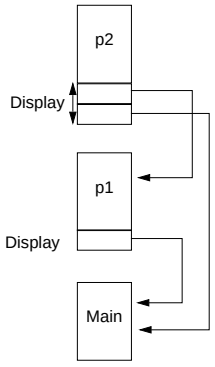
- The static link is a pointer to the most recent activation record for the surrounding bloc
- k (*textual environment*).

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

  For `a` follow the static link 2 steps.

- This method is practical and uses little space. With deeply nested procedures it will be slow.

Display

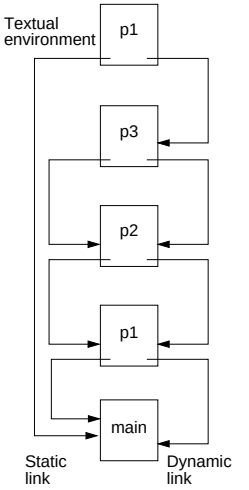


- Display is a table with pointers (addresses) to surrounding procedures' activation records.
- The display can be stored in the activation records.
- Display is faster than static link for deep nesting, but requires more space.

Dynamic link, *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}.
```



Call chain
(on return from `p1` we continue with `p3`)

The stack at 2nd call for p1

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).

De-allocation is **not** performed automatically as in stack allocation.

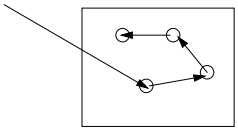
- Pascal

  `new(p)` allocation
  `dispose(p)` manual de-allocation:
  hard work, can lead to bugs

- LISP

  garbage collection: automatic, resource-consuming.

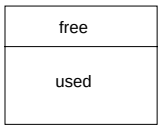
Free list:



○ released space

fragmenting

Optimally it would be:



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

A fixed number of elements, i.e. the size is known at compile time.

```
A : array[1..100] of integer;
```

Stored directly in the activation record.

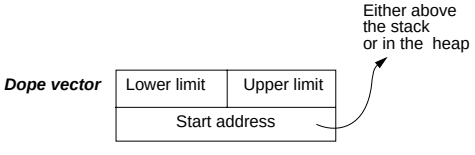
Dynamic arrays

The size is unknown at compile time:

```
B: array[1..max] of integer;
```

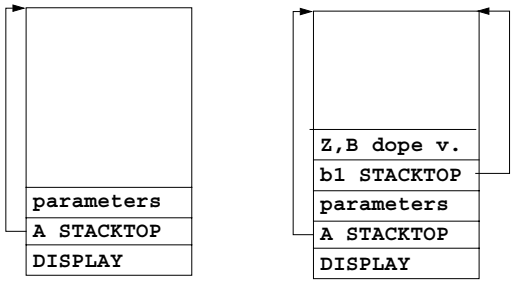
`max` not known at compile time.

Dope vector (*data descriptor*) is used for dynamic arrays. Dope vectors are stored in the activation record.

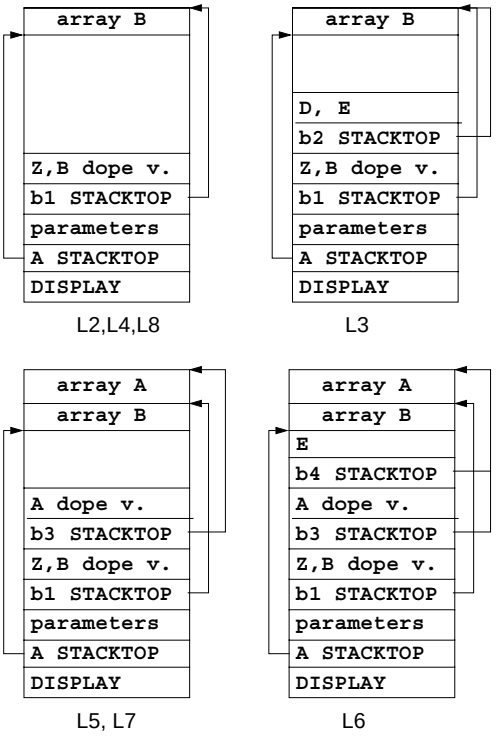


Dynamic arrays and block structures in ALGOL

```
PROCEDURE A(X,Y); INTEGER X, Y;
L1: BEGIN  REAL Z;                    (block B1:) ❶
           ARRAY B[X:Y];
        L2: BEGIN REAL D,E;                    ❷
        L3:    •••
            END;

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



before L1 before L2



L2,L4,L8 L3

L5, L7 L6

Parameter passing (Rep.)

There are different ways of passing parameters in different programming languages. Here are some 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.

2. *Call by value*

   - The value of the actual parameter is passed
   - The actual parameter can not change value

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

3. *Call by value-result* (hybrid) (Rep.)

   - 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.

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.
   - Unpleasant effects!

   Found in Algol, Mathematica, Lazy functional lang.

Example: (Rep.)

```
procedure swap(x, y : integer);
var temp : integer;
begin
 temp := x;
 x := y;
 y := temp;
end;
...
i := 1;
a[i]:=10;  {a: array[1..5] of integer}
print(i,a[i]);
swap(i,a[i]);
print(i,a[1]);
```

Printout from both the **`print`** statements:

Call by reference	Call by value	Call by value-result	Call by name
1 10	1 10	1 10	1 10
10 1	1 10	10 1	error†

† The following happens:

† 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 *)
```


Static memory management in FORTRAN77

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

All memory is allocated **statically**
⇒ variables are referenced by absolute address.

- The data area (which corresponds to the activation record) is often put 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.

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

Return address
J
I
Temp
... Code for SUB ...

At procedure call in FORTRAN77

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.

On return:

1. Re-set the registers.
2. Jump back.

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

- Nested procedures/blocks (PASCAL, ALGOL)
- Dynamic arrays (ALGOL)
- Recursion
- Heap allocation (PASCAL, C, C++, Java)

Problems:

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

Events when 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.

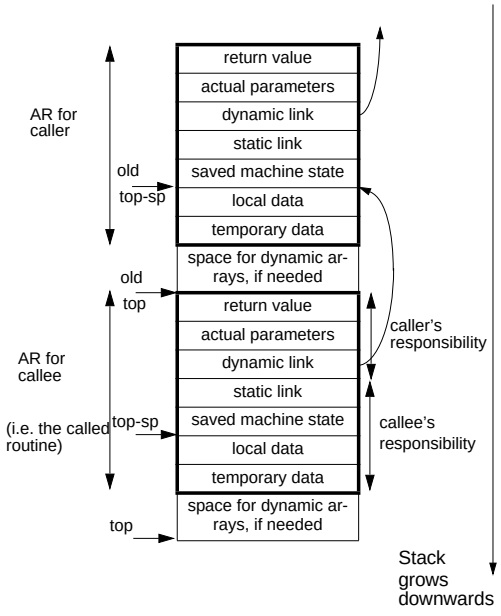
At return:

Q's responsibility

- Save return value in own AR (NB! P can access the return value after the jump).
- Re-set 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 procedure call in ALGOL, Pascal:

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. Re-set dynamic link.
2. Re-set the registers
3. Re-set display / static link
4. Jump back.