PROCEDURAL PARADIGM

ALGOL - 2^{ND} GENERATION LANGUAGE

History

- In the mid 1950s it became apparent to many scientists that a universal, machine indepedent programming language would be valuable.
- In 1960 the final report on ALGOL -> ALGOL-60
- The ALGOL-60 was a very simple and clear language

- The language specification was done using the BNF notation

Program Structure

- One of the major contributions of ALGOL-60 was that it used a hierarchical structure throughout its design.

- ***ALGOL-60 allows nested control structures*** (such as the FOR-loop) ***and nested environments***

Sample ALGOL-60 Program

```
begin
   integer N;
   Read Int(N);

   begin
      real array Data[1:N];
      real sum, avg;
      integer i;
      sum := 0;
      for i := 1 step 1 until N do
        begin real val;
           Read Real (val);
           Data[i] := if val < 0 then -val else val;
        end
      for i := 1 step 1 until N do
         sum := sum + Data[i];
      avg := sum/N;
      Print Real (avg);
   end
end
```

Back

<u>Program Structure</u>

- The statements are of two types:

declarative statements
imperative statements

- The declarative statements were used for variable declarations, procedure declarations and switch declarations

- The lack of ***<u>input-output statements was a mistake</u>***

- Major contributions of ALGOL:

1) the assignment operation

2) the block structure

- The fact that one statement can be replaced by a sequence of statements is a good example of regularity in the design of the languages:

a regular language is far easier to learn and there are fewer exceptions that have to be memorised

- Major contributions of ALGOL:

- Each block defines a nested scope

- The advantage of the nested scope is that the variables are "visible" throughout the scope of the block

- ALGOL through ***the implementation of the block structure avoids variable re-declaration***
- ***Allows for both dynamic and static scoping in the programs***
- ***Algol supports only static scoping.***
- ***However will compare the two scoping methods***

- <u>Dynamic scoping</u>:

The meanings of the statements and expressions are determined by the dynamic structure of the computations evolving in time

- **Static scoping**:

the meaning of the statements and expressions are determined by the static nature of the program

- The block allows efficient storage management on a stack

Sample Code

Syntax

- The syntax was greatly improved when compared with that of the first generation languages

- Introduced FOR-loop, a WHILE-loop and the SWITCH statement to handle multiple cases

Syntax

- ***The syntax was machine independent***

- ***The variable names did not have restrictions on the number of characters used***

- ***ALGOL-60 introduced the concept of keywords which made the syntax more readable and easier to understand while eliminating errors resulting from the use of keywords as variable names***

Data Types and Structures

- Three data types: integer, real and `string

- ***Regularity was a major goal of ALGOL-60***

- ***Regularity principle:***

regular rules, without exceptions, are easier to learn, use, describe and implement

Data Types and Structures

- A special application of the regularity principle is the ***zero-one-infinity principle:***

The only reasonable numbers in a programming languages design are zero, one and infinity

- ***Arrays in ALGOL are generalized and dynamic***

- ***ALGOL-60 had strong typing and keywords***

- ***ALGOL was designed to allow recursion***

- Communication is done by parameter passing

- ***pass by value has no side effects but it can be very expensive when dealing arrays***

- ***pass by name uses substitution to prevent naming conflicts between input-output parameters***

2nd Generation Languages

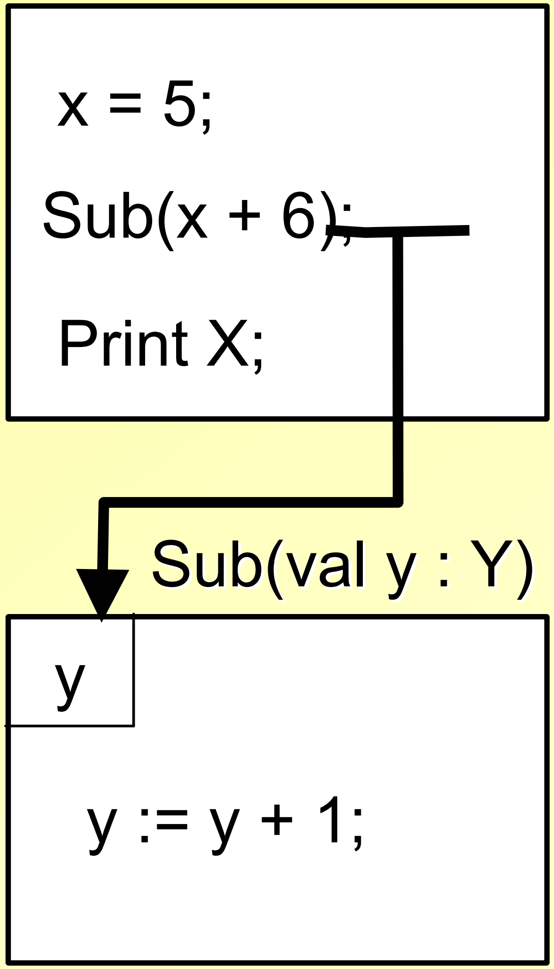
- Were characterized by:
- ***Use of the block structure***
- ***Structured control structures which eliminate much of the need for confusing networks of GOTO statements by hierarchically structuring the control flow***
- ***The syntax structures were free format with machine independent lexical conventions (reserved words)***

Parameter Passing Techniques

- The two best known parameter passing methods are ***call-by-value*** and ***call-by-reference***
- The ***call-by-value*** has the advantage that the actual parameter value is not modified
- The ***call-by-reference*** has the advantage that it is very efficient when dealing scalar data structures such as arrays

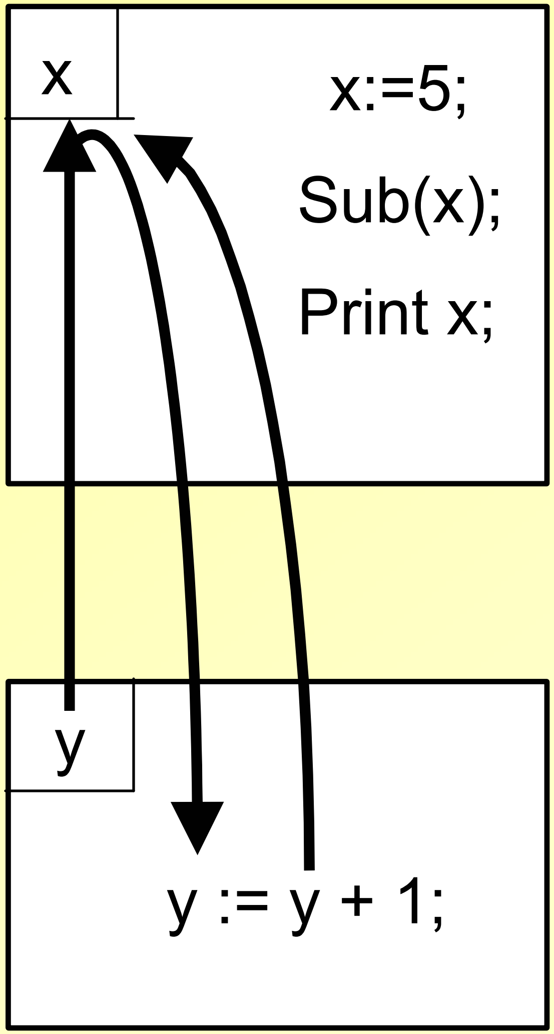
Call by Value

- The *value* of the actual parameter is copied into the formal parameter
- Y is assigned the value of X + 6 which is 11
- When the statement Print X is executed, the value of X is still 5



Call by Reference

- *Reference* to the actual parameter is copied into the formal parameter
- Before the subroutine Sub(x) is called, the value of x is 5
- When Sub(x) is called, the variable y gets the address of x
- Hence when y:=y+1 is computed, the resulting value (6) is put into x's memory location
- When Print x is executed the value of x is 6



Sub(ref y : Y)

CALL BY RESULT

- Used in ADA to implement out mode parameters
- *Formal paramater* acts as an *un-initialised local variable* which is given a value during the execution of the procedure
- On leaving the procedure, the value of the *formal parameter* is assigned to the *actual parameter* which must be a variable

CALL BY RESULT

```
Procedure Read_Negative_data(neg_number : out integer)
number : integer;
Begin
   get(number);
   while number >= 0 loop
       put_line("number not negative, try again )
       get(number);
   end loop;
   neg_number := number;
End Read_Negative;
Read_Negative_data(amount);
```

- the value of the integer variable amount is not updated to the value of neg_number until procedure Read_Negative is left

CALL BY VALUE-RESULT

- Amalgamation of call by value and call by result.
- The *formal parameter* acts as a *local variable* which is initialised to the value of the *actual parameter*.
- Within the procedure, changes to the *formal parameter* affect only the local copy.
- When the procedure completes its execution, the *actual parameter* is updated to the final value of the *formal parameter*.

CALL BY VALUE-RESULT

```
Procedure Update(balance: in out integer)
transaction : integer;
Begin
  for j in 1..10 loop
     get(transaction);
     balance := balance + transaction;
  end loop;
End Update;
Update(currentaccount);
```

- The actual parameter *currentaccount* is only updated when the procedure is Update completes its execution.

Static Scoping

- ***The scope of program variable is the range of statements in which the variable is visible***
- The scope rules of a language determine how a particular occurrence of a name is associated with a variable
- ALGOL-60 introduced a method of binding names to the non-local variables called static scoping
- ***The visibility of the identifiers and the process of binding of names to declarations is determined at compile time - this is known as static scoping***

Static Scoping

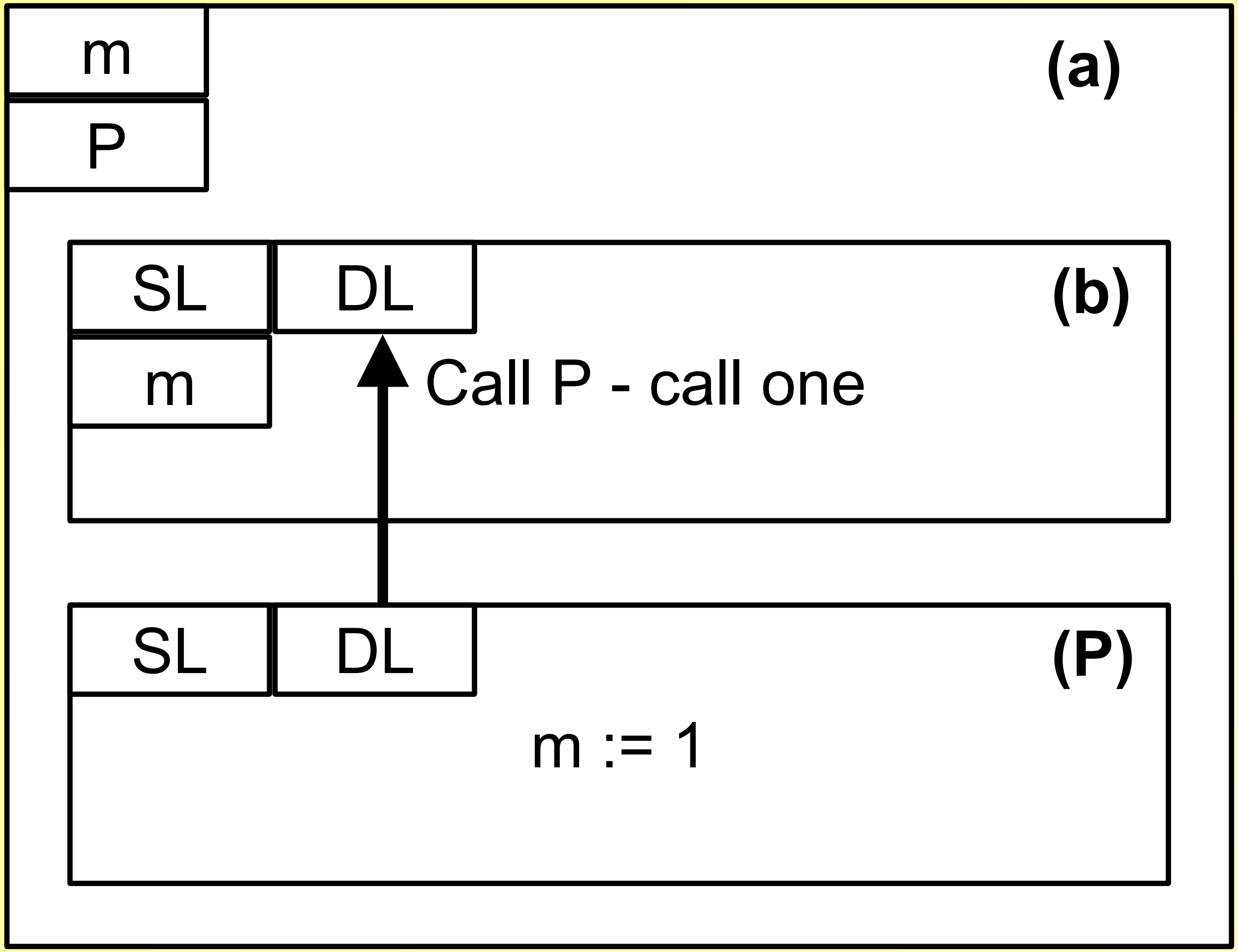
- ***The advantage of static scope is that it allows type checking to be performed by the compiler***

Static Scoping

Consider the example of the ALGOL code shown below:

```
a:begin integer m;
      procedure P;
          m := 1;
b:    begin integer m;
          P; -> call 1;
      end;
      P; -> call 2;
  end
```

Static Scoping Example



- m : = 1 refers to the variable declaration in the outer block **a** (for both calls)
- the contour of P is nested in inside block a even though it is called from block **b**

Dynamic Scoping

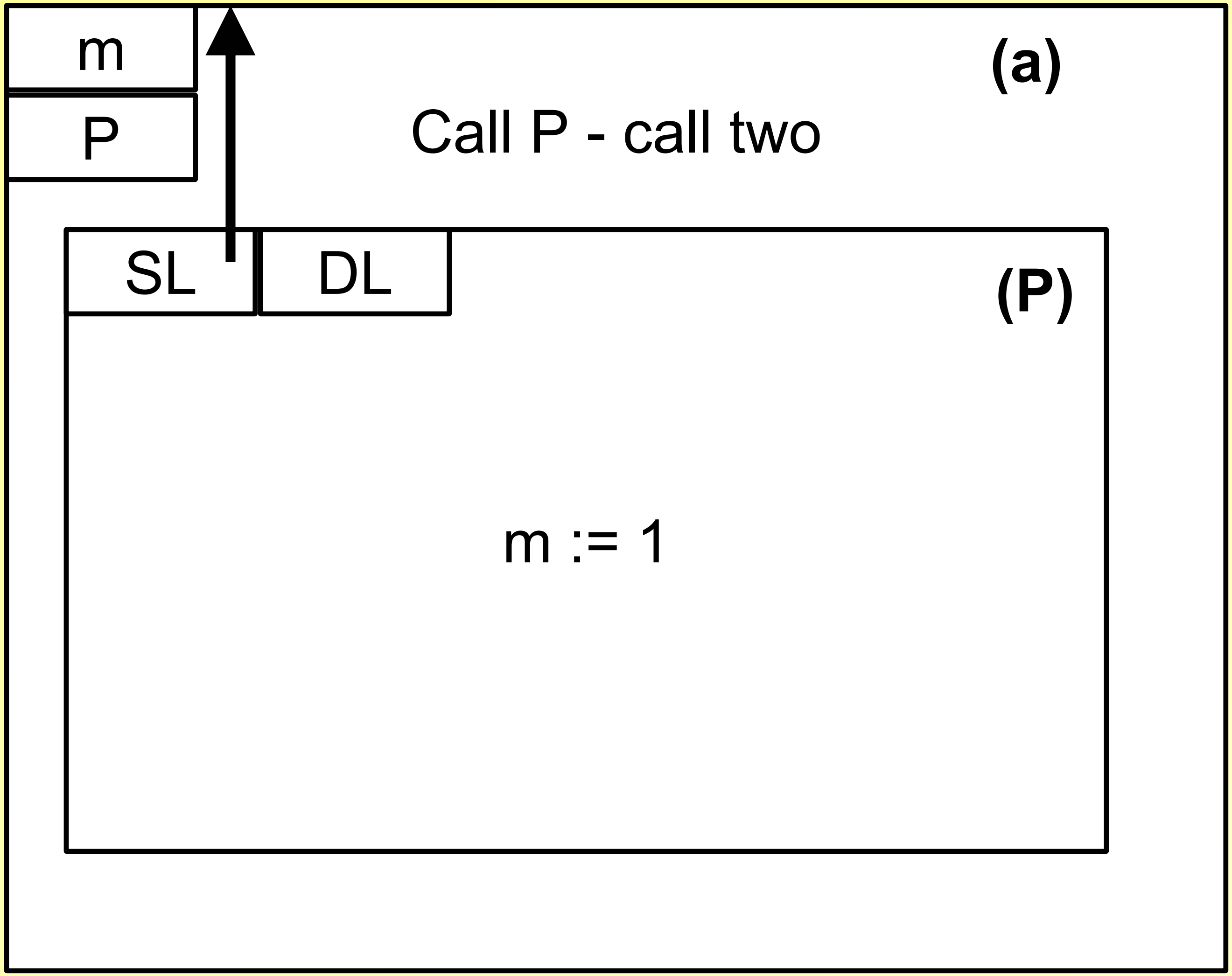
- ***In dynamic scoping, the binding between the use of an identifier and its declaration depends on the execution of the program and therefore it is delayed until run-time***

- There are two major problems with dynamic scoping:

1. ***There is no way to statically type-check references to non-locals***

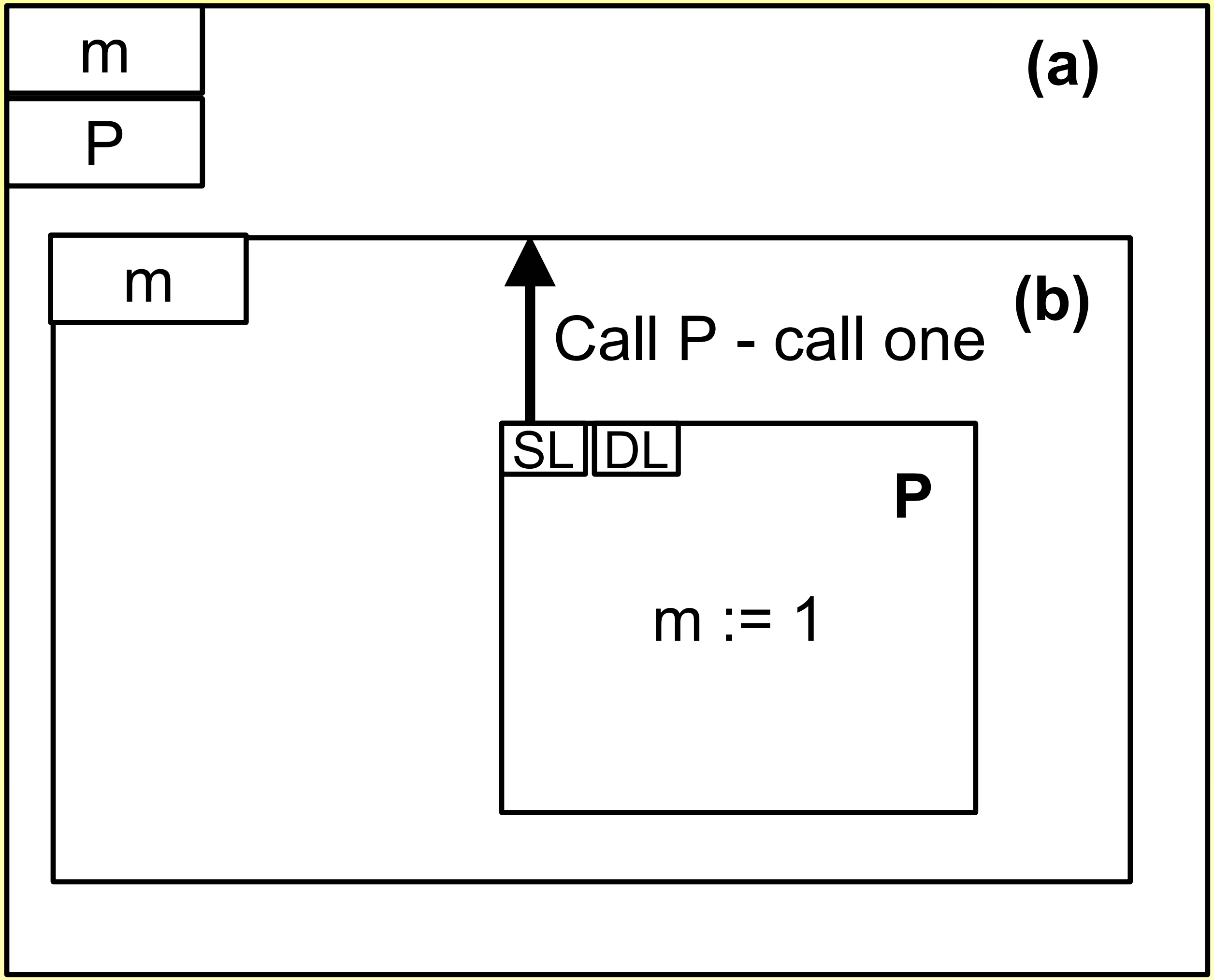
2. ***Dynamic scoping makes programs a lot more difficult to read***

Dynamic Scoping - Example 1



m := 1 refers to the m declared in block **a**

Dynamic Scoping - Example 2



m := 1 refers to the variable declared in block **b**

Pointers

- A pointer type is one in which the variables consists of a memory address and a special value: NIL

- The pointer types have been designed for two uses:
  1. ***to provide some of the power of indirect addressing used in assembly language***

  2. ***to provide a method of dynamic storage management***

Pointers

- two types

  - ***normal pointer***

  - ***dereferencing pointer***

Pointers

- Pointers point to objects
- ***Pointers are variables***
- A pointer is usually given it's value by taking the *address of* a variable or subprogram

pointer-variable = Address-of(variable-name)

pointer-variable =Address-of(function-name)

Pointers

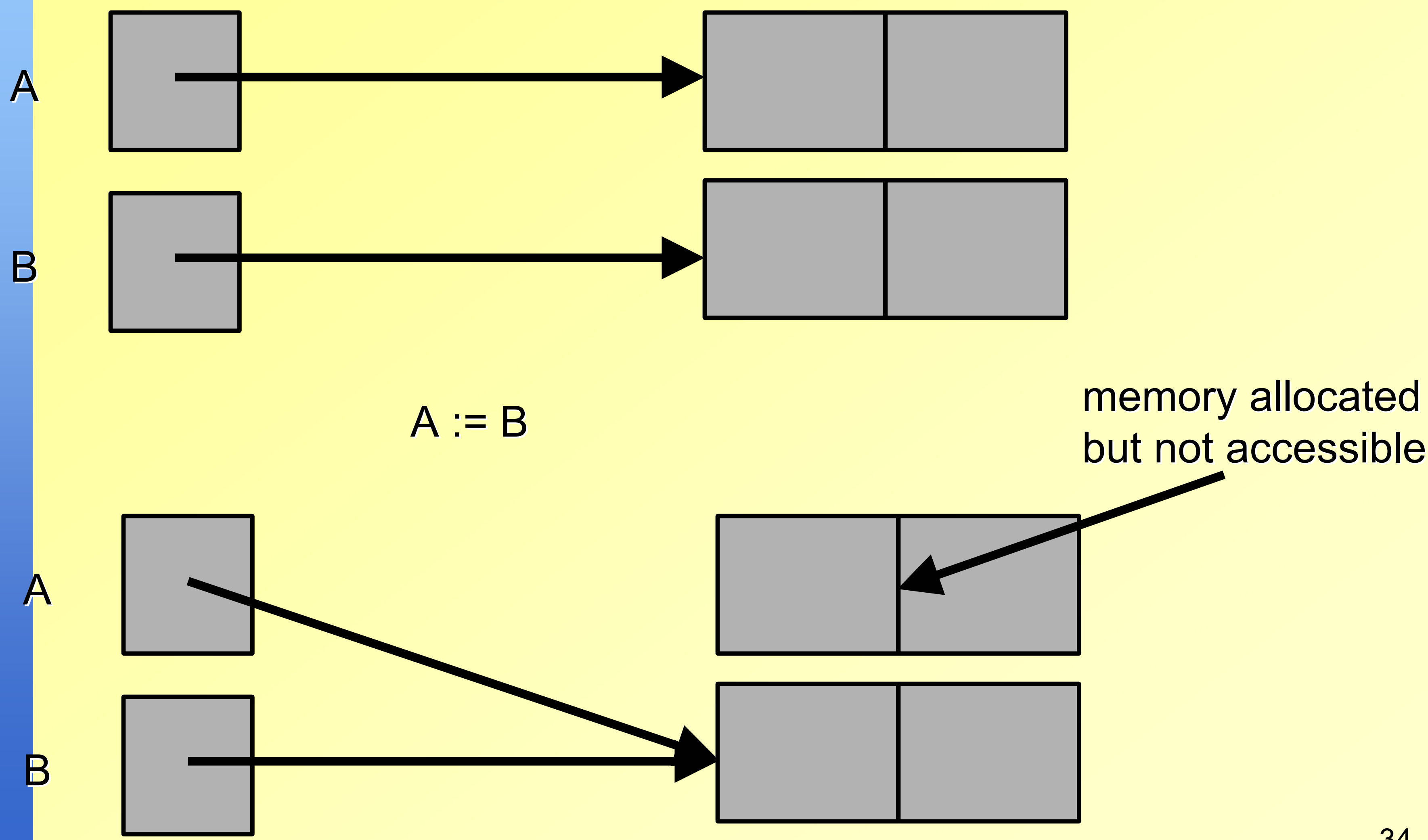
- *Dereferencing* a pointer is the process of following the pointer to the variable or subprogram whose address it holds

  <Dereference(pointer-variable)>

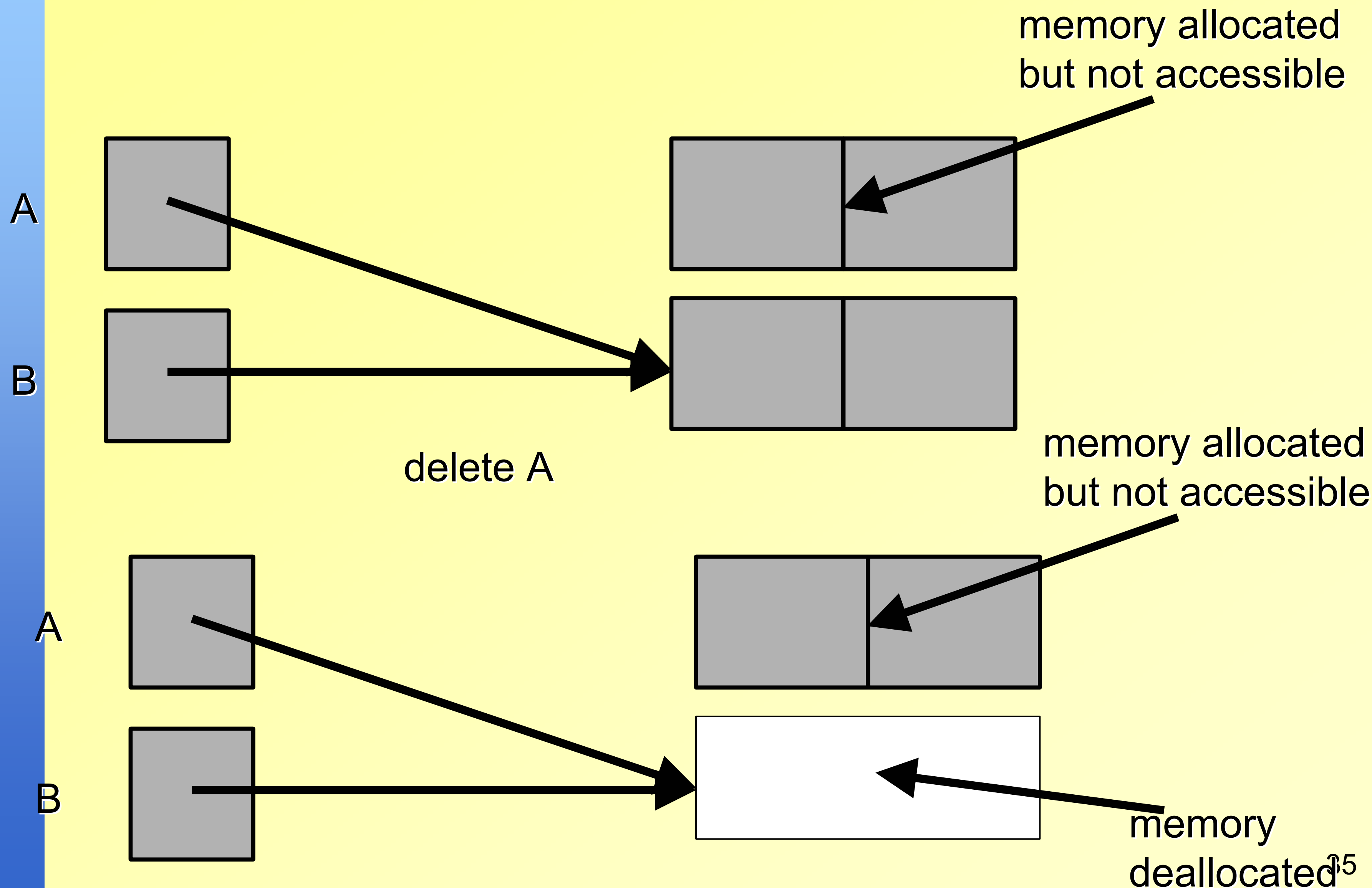
- Problem 1 - ***dangling pointers***

- Problem 2 – ***memory leaks***

Example of a Memory Leak



Example of a Dangling Pointer



Pointers vs References

- C++ includes a special kind of pointer type that is used primarily for the parameters in function definitions: References

- pointers:

  the value of the pointer variable (ie where it points to) can be changed

  the value of the variable the pointer points to can also be changed

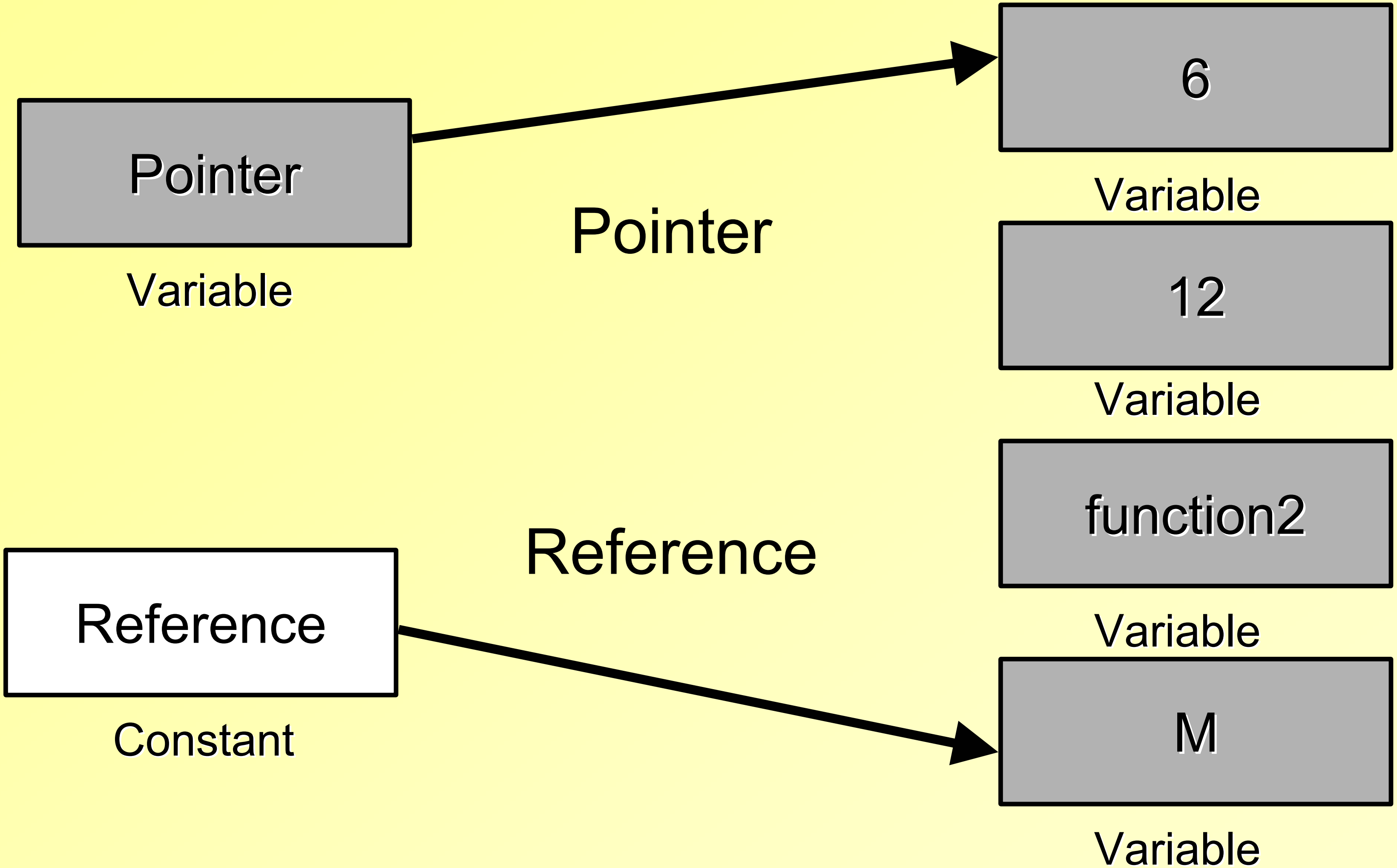
Pointers vs References

- References are constant pointers:

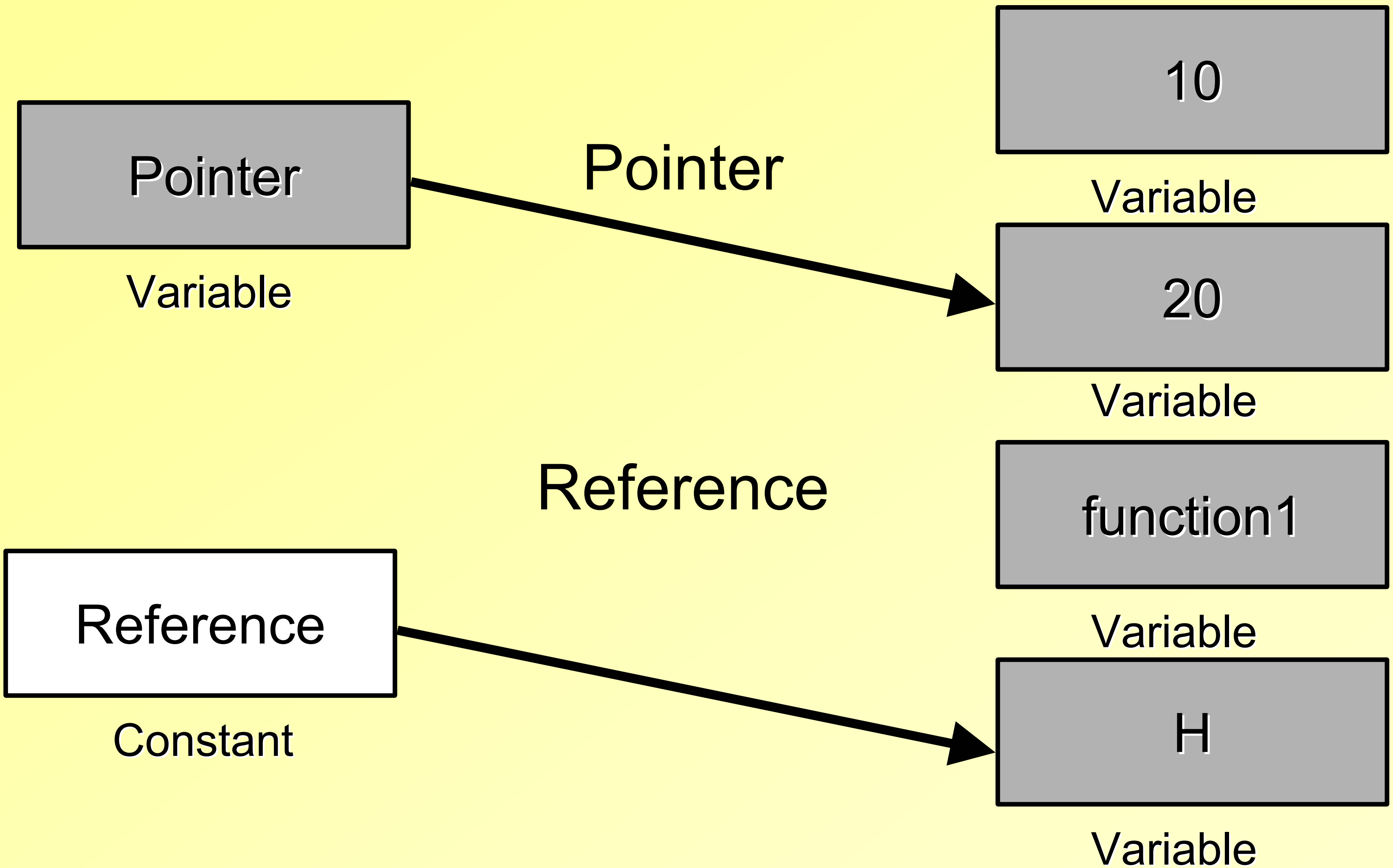
    reference pointed at something and cannot be moved

    the value of the variable the reference points to can be changed

Pointers vs References



Pointers vs References



Strong and Weak Typing

- ***Strong typing is very important when considering language design as it helps eliminate errors before run-time***

- The advantage of strong typing is that the type of variables known at compile time and **the compiler can catch errors**

- Weak typing means **the type of variable may not be known until run-time and as a result errors may go undetected**

Activation Record - Concepts

- ALGOL allows procedures to be recursive and it is possible that there may be several instances of the same procedure active at one time

- To know the state of the procedure activation it is necessary to know:

1. ***the code that makes up the body of the procedure***
2. ***the place in the code where the activation of the procedure is now executing***
3. ***the value of the variables visible to this activation***

Activation Record - Concepts

- The activation record contains the variable parts which define a particular execution of a subprogram

Instruction Part (IP)

Environment Part (EP)

Activation Record - Concepts

- The ***instruction part*** (or instruction pointer) designates the current instruction being (or to be) executed in this activation of the subprogram
- The ***environment part*** defines both the local and non-local *context* to be used for this activation of the subprogram - it determines how the instructions are interpreted

Activation Records - Concepts

- The ***local context*** contains: ***local variables, actual parameters and register***
- When a subprogram is called, it must provide some reference to the calling program so that it will know which caller to resume when it completed its execution

Activation Records - Concepts

- How? provide a pointer to the calling program activation record - ***the pointer is stored in the activation record of the subprogram invoked and it is known as dynamic link***

- ***A sequence of dynamic links that reach from the callee to the caller is known as a dynamic chain***

Activation Records - Concepts

- ***A static link (static scope pointer), points to the bottom of the activation record instance of an activation of the static parent***
- The ***non-local context*** contains: *the dynamic call sequence* (or dynamic chain) and *the static scope*
- There are two ways to access non-local variables in statically scoped languages: using ***static chains*** or using ***displays***

Activation Records - Concepts

- **The static chains technique**

  Step 1 - find the instance of the activation record in which the variable was allocated

  Step 2 - use the local offset of the variable within the activation record to access it

Activation Records - Concepts

- Finding the correct activation record instance of a non-local variable is simple ***because the nesting level is known at compile time, the compiler can determine if a variable is non-local but also the length of the chain needed to reach the activation record instance containing the non-local variable***

- Let ***static-depth*** be an integer associated with a static scope that indicates how deeply it is nested in the outermost scope

Activation Records - Concepts

- The ***length of the static chain*** needed to reach the correct activation record instance for a ***non-local variable X*** is the *difference between* the *static depth of the procedure containing the reference* to the variable X and the *static depth of the procedure containing the declaration of X*
- The ***difference*** is called the ***nesting depth***, or ***chain-offset***

Activation Records - Concepts

- The static chain technique has the disadvantage that the ***access to a non-local variable in scopes beyond the static parent is costly***

- **The display technique**
- The static links are collected in a single array called a display rather than being stored in the activation records

Activation Records - Concepts

- **The display technique**

- The contents of the display at any specific time is a list of addresses of the accessible activation record instances in the stack, one for each scope, in the order in which they are nested
- Access to non-local variables using a display requires only two steps regardless of the number of scope levels between the reference and the declaration of the variable

Activation Records - Concepts

- Step 1: the link to the correct activation record which resides in the display is found using a statically computed value called the ***display-offset*** which is closely related to the ***chain-offset***

- Step2: the ***local-offset*** within the activation record instance is computed and used in the same way as in the static chain implementation

- A non-local reference is represented by an ordered pair of integers - [display-offset|local-offset]

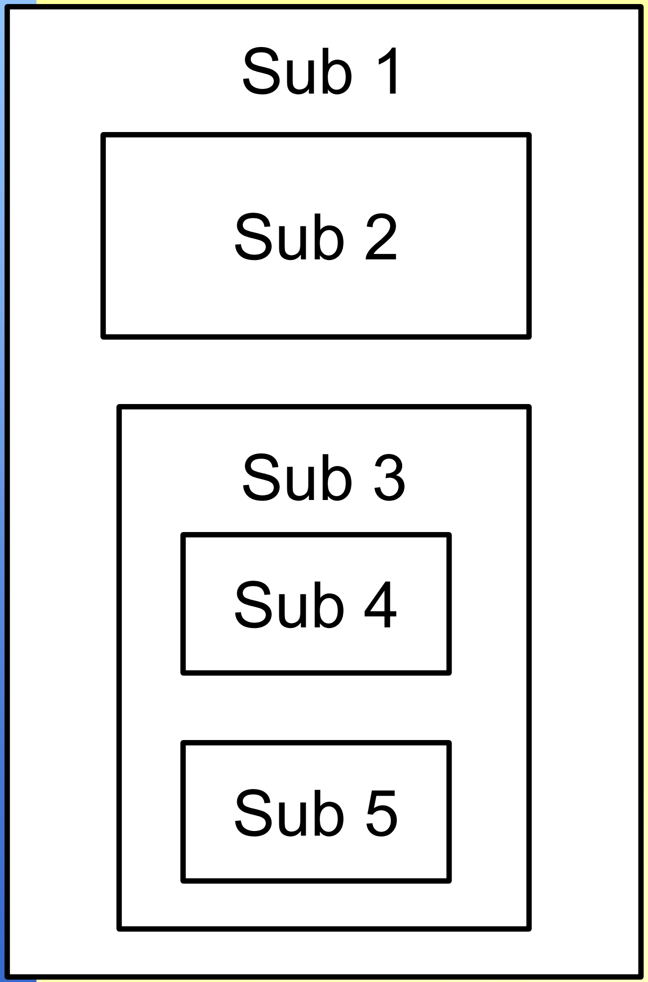
Activation Records - Concepts

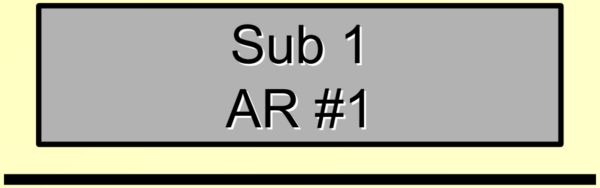
- A pointer at position N in the display points to an activation record instance for a procedure with a static depth of N
- ***The disadvantage of the display method is that it requires extra memory for the array implementing the display***

Activation Record

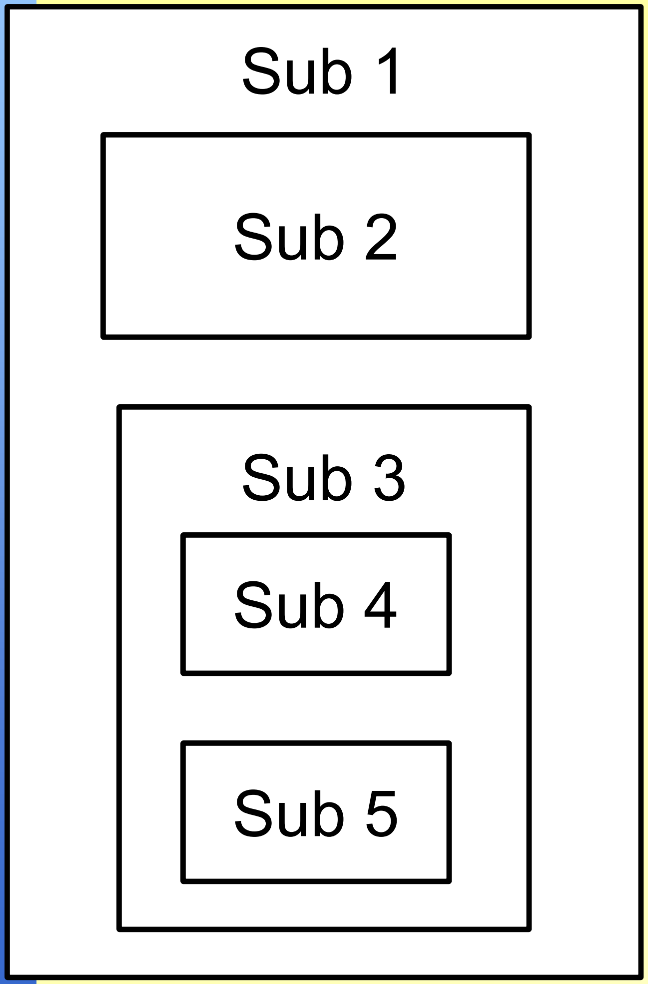
Address	Instruction Pointer
x | y | z[0] z[1] ...	Local Variables
m | n | p | functptr	Formal Parameters
R1 | R2 | R3 | X | Y | Z	Register Store
Pointer	Dynamic Link
Pointer	Static Link

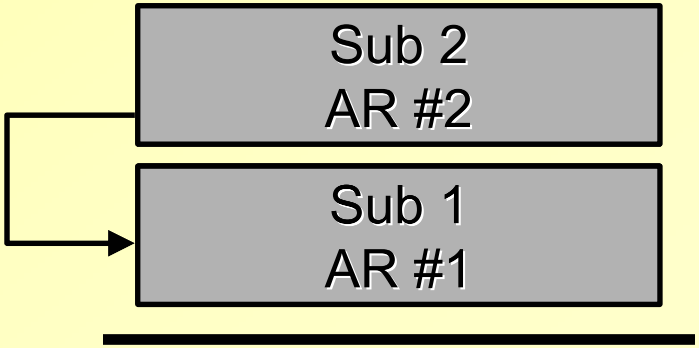
Dynamic Chain - Example



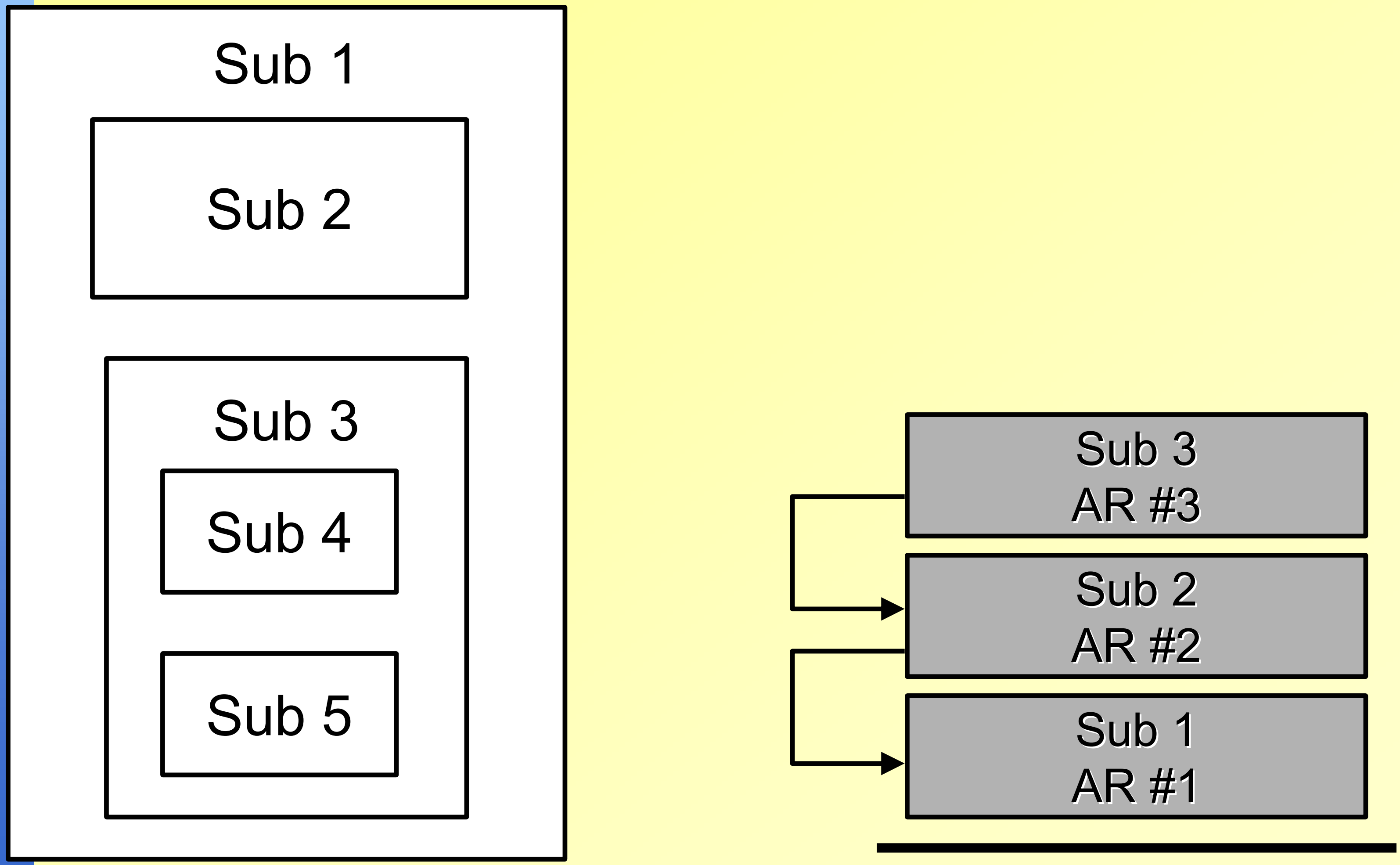


Dynamic Chain - Example

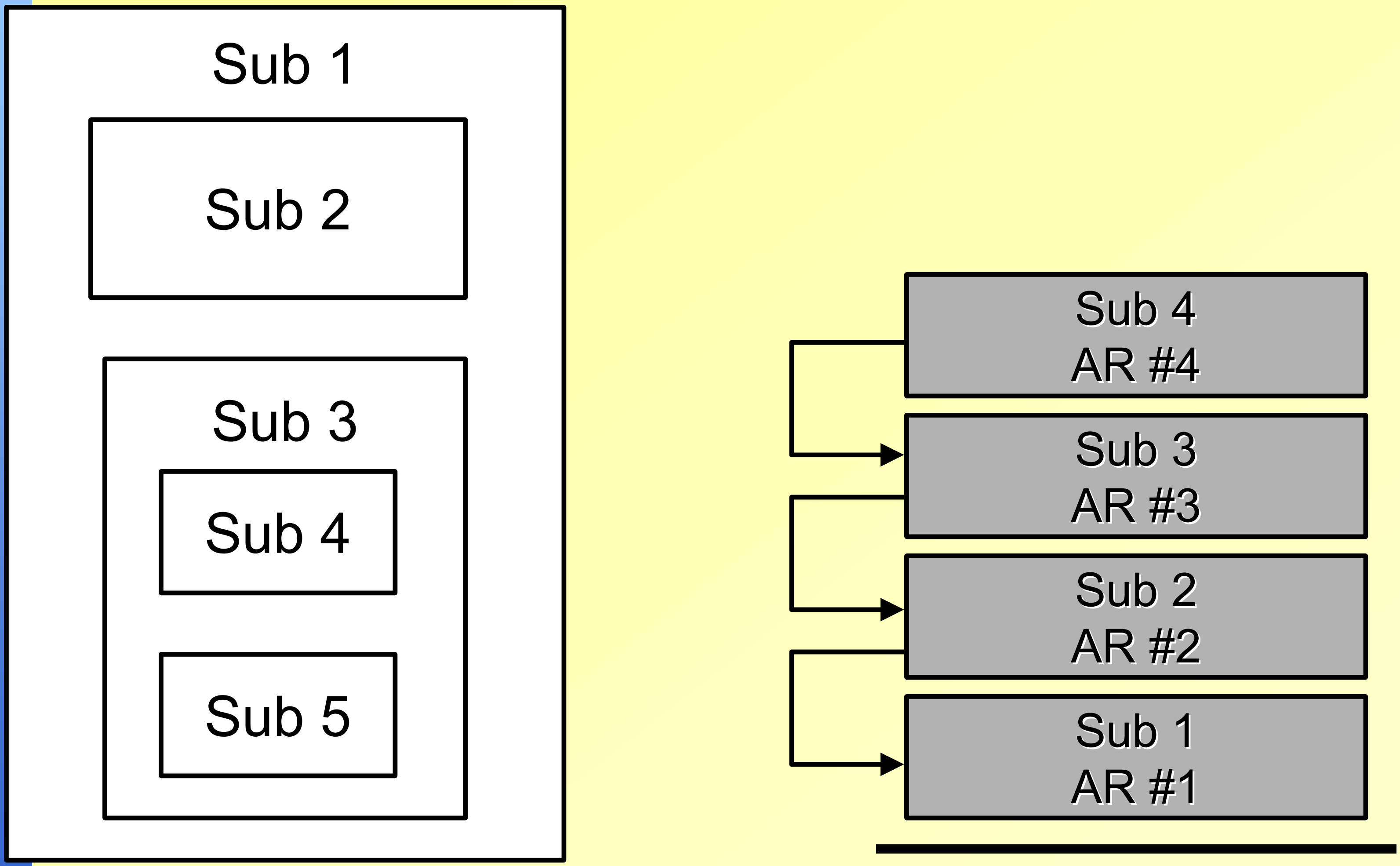




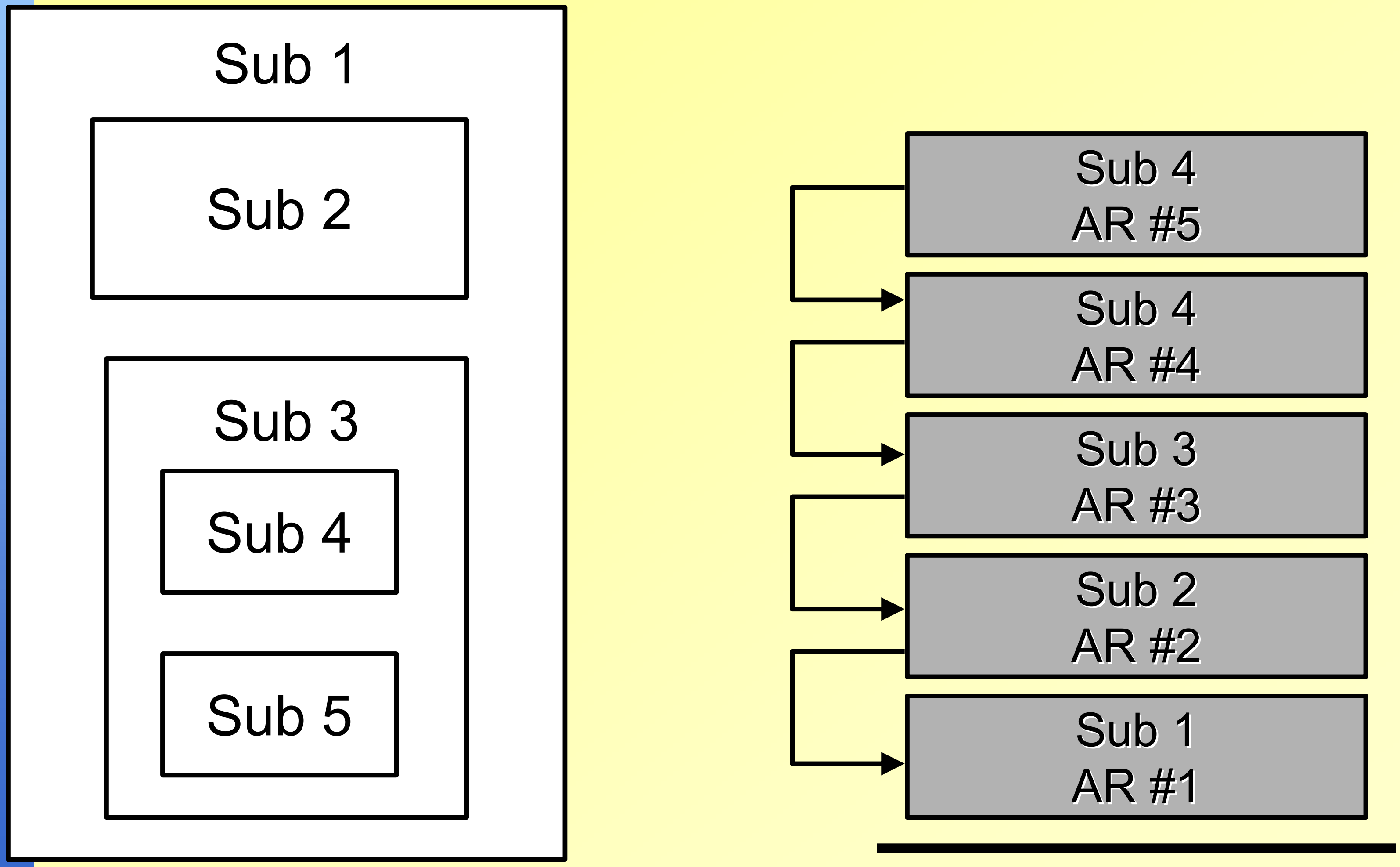
Dynamic Chain - Example



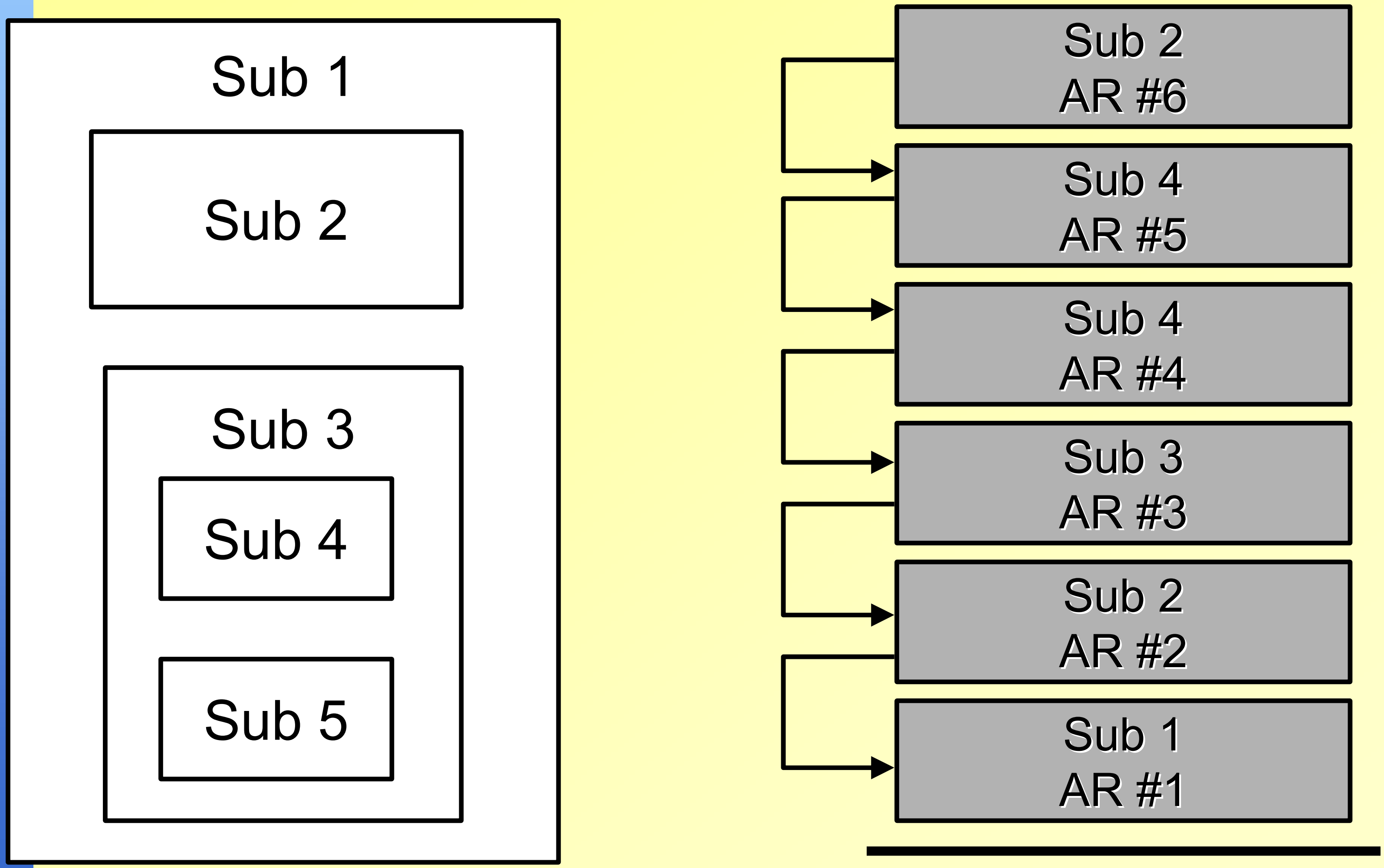
Dynamic Chain - Example



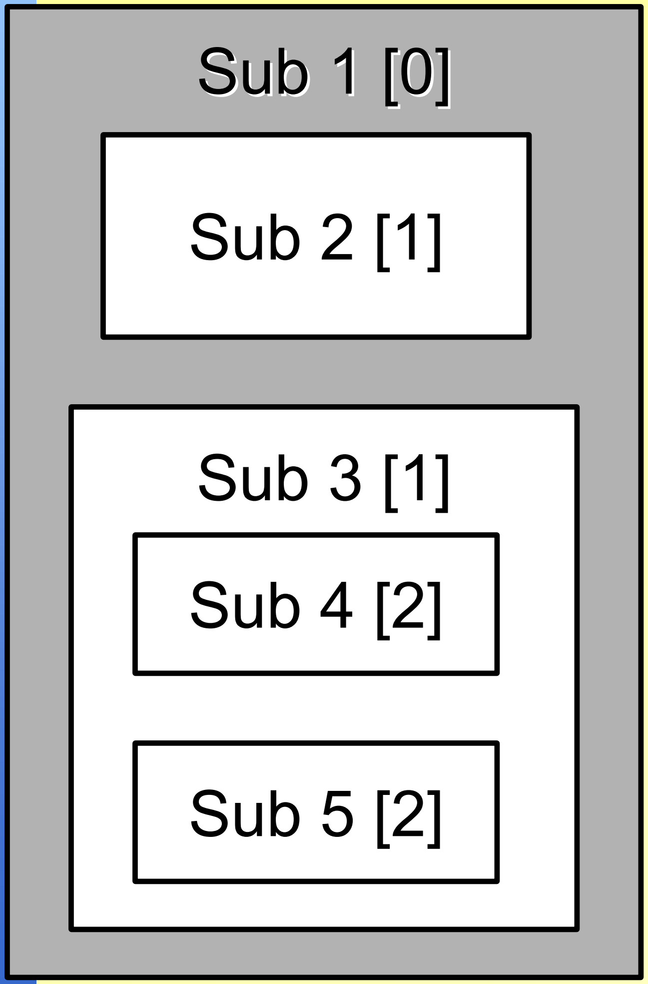
Dynamic Chain - Example

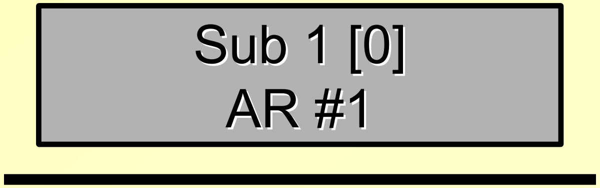


Dynamic Chain - Example

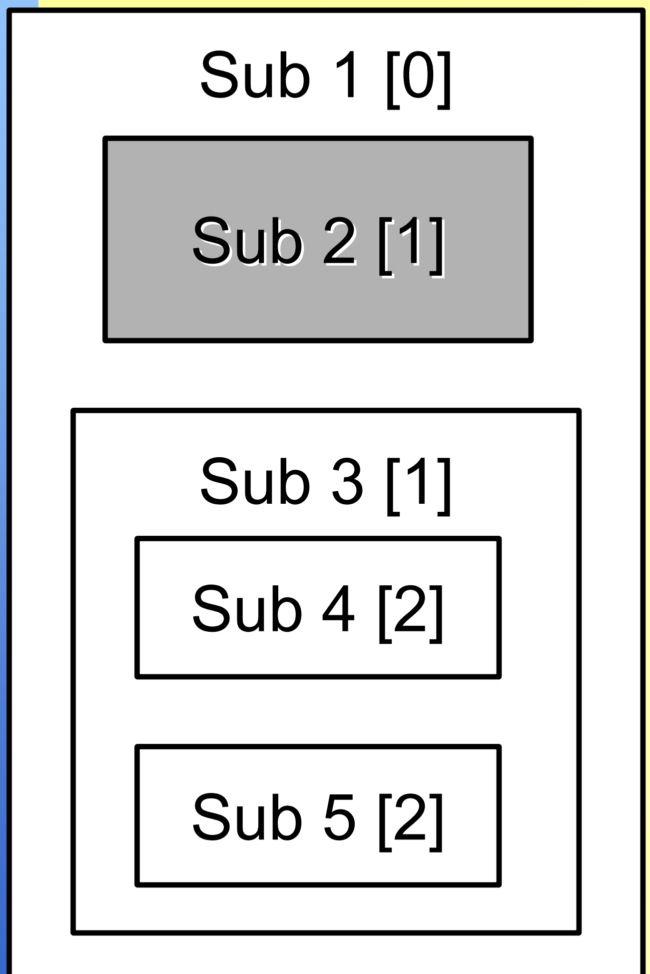


Static Chain - Example





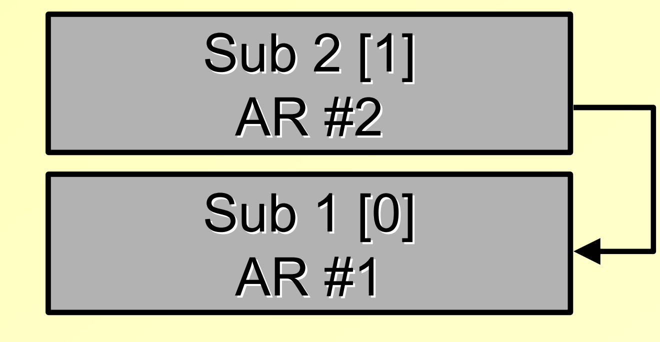
Static Chain - Example



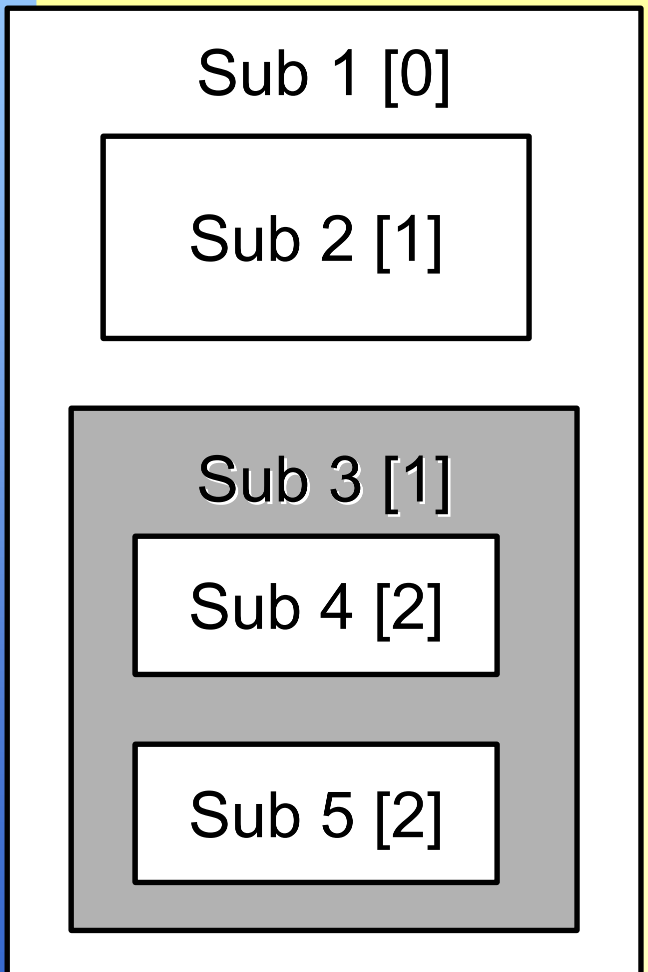
Sub 2 @ 1

Sub 1 @ 0

Therefore assign this AR to the static link



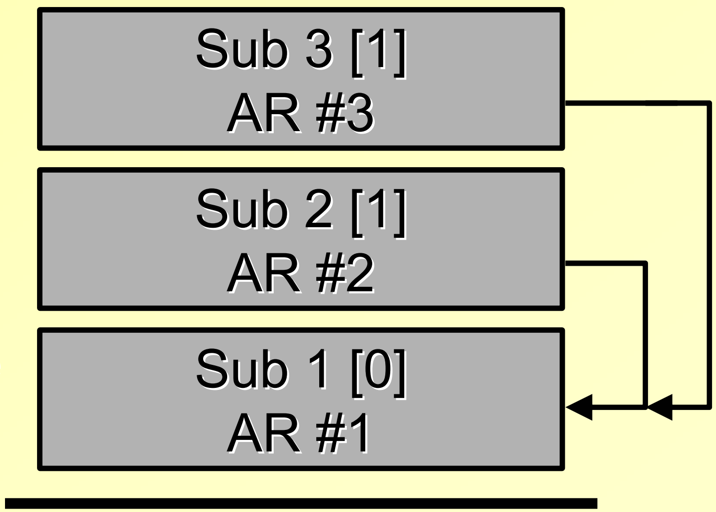
Static Chain - Example



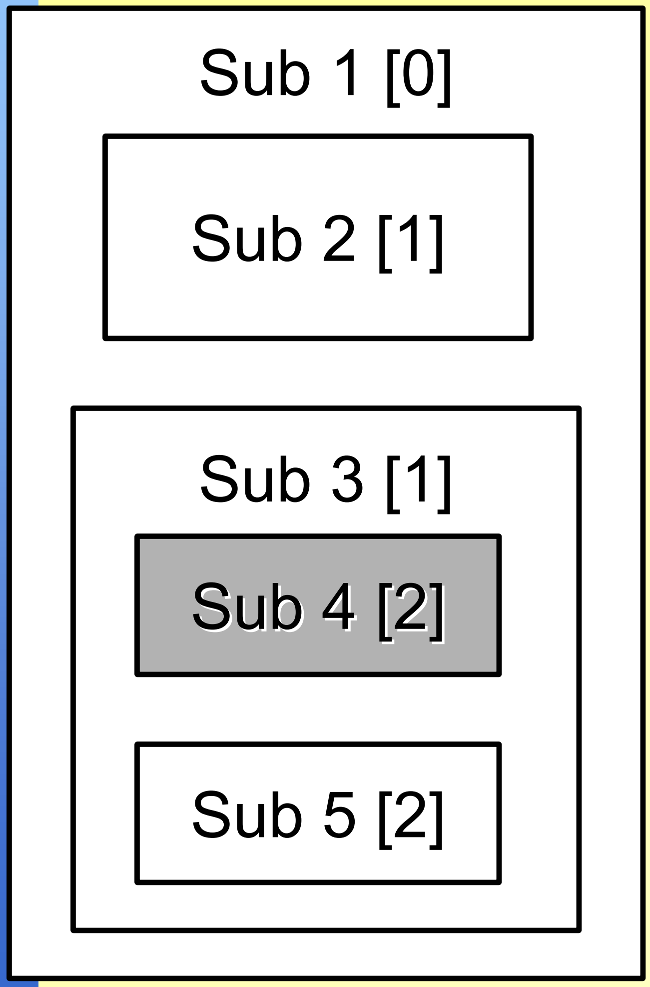
Sub 3 @ 1

Sub 2 @ 1

Therefore assign same AR to the static link



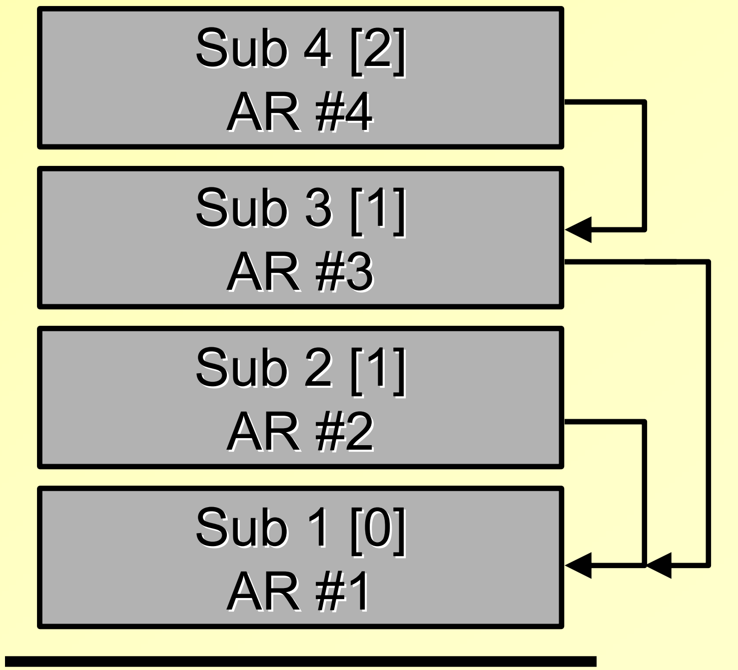
Static Chain - Example



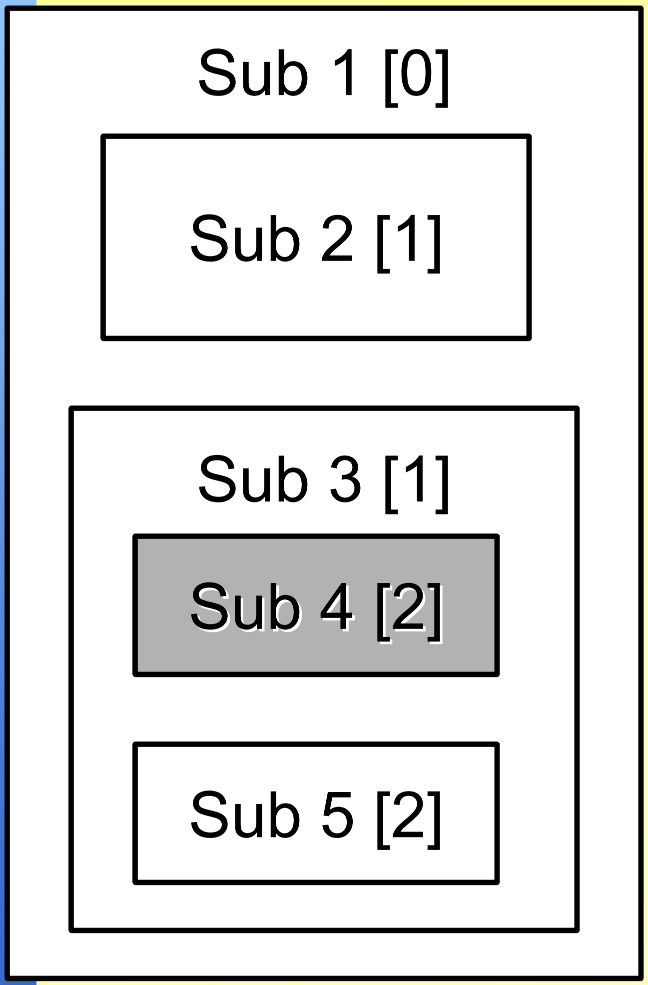
Sub 4 @ 2

Sub 3 @ 1

Therefore assign this AR to the static link



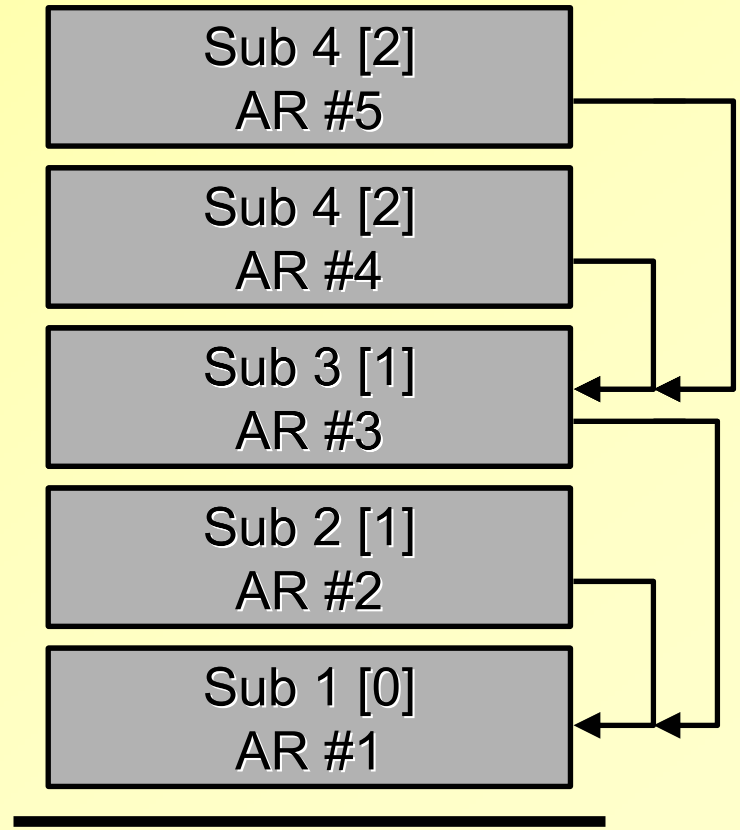
Static Chain - Example



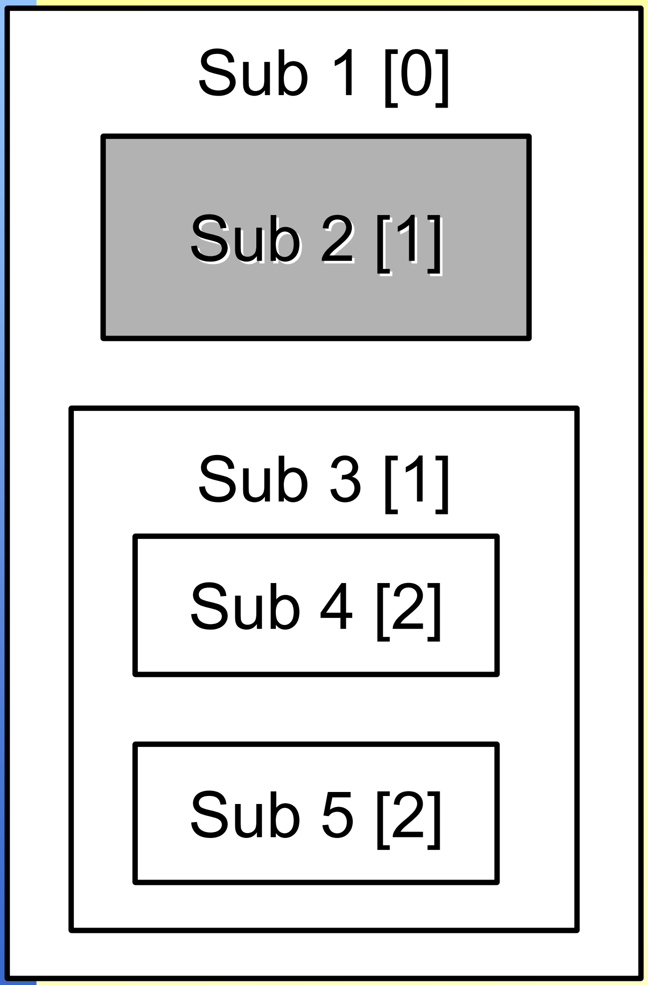
Sub 4 @ 2

Sub 4 @ 2

Therefore assign same AR to this static link



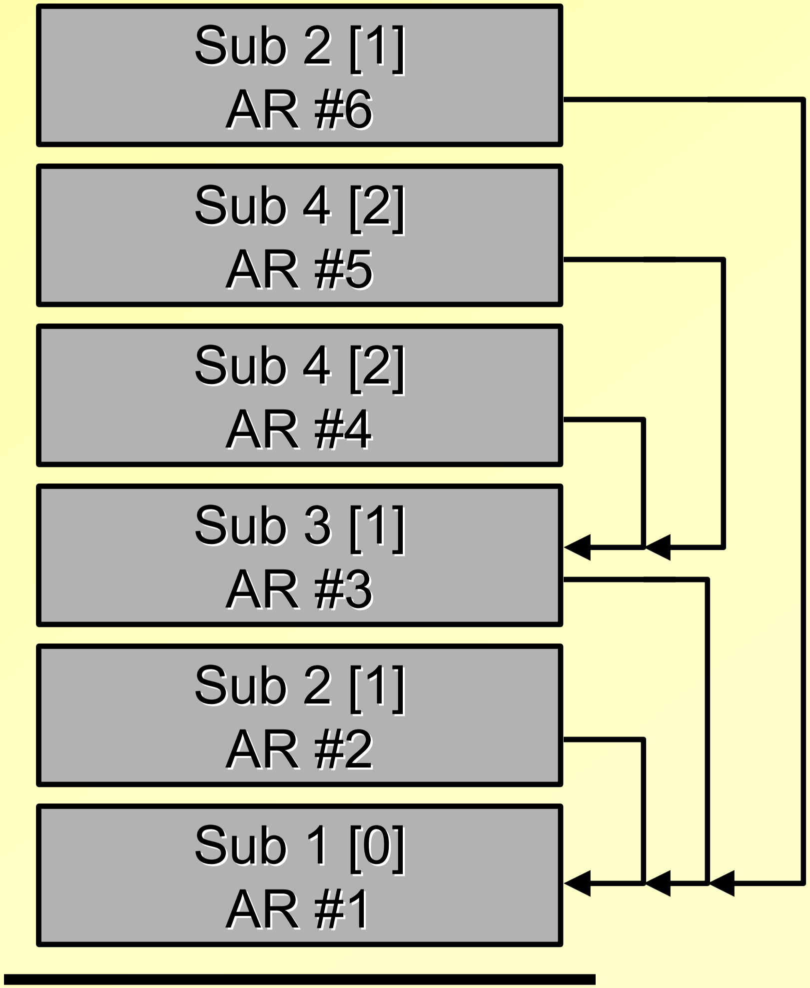
Static Chain - Example



Sub 2 @ 1

Sub 4 @ 2

Therefore assign next AR up the static chain



Steps to Calling a Procedure

Save caller’s state

Transmit parameters to callee

Establish callee’s dynamic link

Establish callee’s static link

Enter the callee

Steps to Exiting a Procedure

Delete callee's state

Restore the state of caller

Continue execution of the caller at the saved Instruction Pointer address