Algol and Pascal

ALGOL

IT'S NOT FORMULA TRANSLATION, IT'S AN ALGORITHMIC LANGUAGE

Meanwhile, in the 1950s…

- In the mid 1950s, there was a small explosion of early languages similar to FORTRAN
- This was mostly because FORTRAN was proprietary and could only be used on IBM machines
- Computer scientists had realised that a universal machine independent compilable language would be extremely valuable
- Preliminary works in 1958 resulted in ALGOL-58, which was never implemented. Several dialects were though (JOVIAL, NELIAC)
- In 1960 however, ALGOL-60 appeared

ALGOL-60

- Very simple language written in BNF notation.
- Only consisted of 15 pages of rules, compared to it's competitors which had hundreds to thousands of pages of complex BNF rule statements.
- Major contribution: used a consistent hierarchical structure all the way through it's design
- Nested control structures (unlike FORTRAN which just allowed nested for loop) and nested environments!
- Very little need for Goto!

ALGOL-60

- Two types of statements
  - Declarative statements
  - Imperative statements
- Declarative statements declared variables, procedures, and switch statements
- Imperative statements were either computational or control flow statements.
- **Serious problem: No IO!!**
- The only computational statement:
  - Variable := expression

Major contributions of ALGOL

- The assignment operator was different from the Boolean operator
  - := vs =
- Block Structure, which allows for nested scopes
  - This allowed one to use a sequence of statements whenever one was permitted by simply enclosing them with BEGIN and END
  - This is excellent for regularity, as you can replace one statement at any time with a group of statements
  - **A regular language is easier to learn as there are less exceptions that need to be memorised**

ALGOL's Blocks

- Each block statement defines a new nested scope
- The advantage of this is that variables are visible throughout the scope of the block
- This regularity of scope was not shared by FORTRAN
  - FORTRAN's scopes were bound to subprograms, common blocks were variable but you couldn't see them unless you re-declared their existence. Violates the abstraction principle and just causes problems.
- ALGOL uses the block structure to avoid re-declaration by allowing the programmer to define any number of scopes to any depth. This makes programming so much faster!

ALGOL Scopes

- Algol has two types of scope:
  - Dynamic scope:
    - The meanings of statements and expressions are determined by the dynamic structure of computations at the time
    - That means functions, global variables, etc.
  - Static scope:
    - The meanings of statements and expressions are determined by the static structure of the program.
    - Nested scopes can see variables above them!
- We'll get back to these later.

ALGOL Syntax

- Syntax was far better than first generation languages
- ALGOL had IF-THEN and GOTO, but also FOR, WHILE, and SWITCH
- The syntax was machine independent!
- No restrictions on variable name length!
- ALGOL-60 even had reserved keywords, eliminating the problems FORTRAN had with accidentally using IF as a variable.

ALGOL Data Types and Structs

- ALGOL had three data types.
  - Integer
  - Real
  - Boolean
- No double (this was machine dependant at the time) and no complex (not primitive enough)
- ALGOL-60 wanted to be super regular and as such it followed the regularity principle closely.
- ALGOL-60 follows the zero-one-infinity principle too, as arrays are generalised, can have many dimensions, *and* can specify negative numbers as bounds! (eg. [-100:200])
- You can only index by integers though.

Other cool features of ALGOL

- ALGOL had strong typing (no exceptions) and keywords
- Also, unlike FORTRAN, ALGOL was designed to allow recursion (remember dynamic scoping?)
- ALGOL also had two forms of parameter passing:
  - Pass by Value
  - Pass by *Name*
- No pass by reference (hooray)

Wait, pass by *Name*?

- Pass by Value is something you all know and love
  - The value of the parameter is copied wholesale into a new variable. This means no side effects can occur but it also means that you use a lot of memory and in some cases time…. Especially when passing arrays.
- Pass by Name however is something else entirely
- Instead of copying data, you pass it using name substitution.
- Like reference passing, but unlike reference passing, variables are evaluated *dynamically* rather than statically.

Example Of Why This Is Cool

- If x is false here, y will never even be evaluated.
- If y is a function, that means you can use this to call a function only when x is false.
- Can shortcut logic statements *and* control structures when not important

```
boolean procedure and (x, y);
  boolean x, y;
begin
  if x then return y else return false
end;
```

Example of why this is not cool

- What happens?
  - swap(x,y)
  - temp := x
  - x := y
  - y := temp
- But what if we do something dumb?
  - swap(i, x[i])
  - temp := i
  - i := x[i]
  - x[i] := temp
- Now the number's been copied someplace else!

```
procedure swap (a, b);
integer a, b, temp;
begin
   temp := a;
   a := b;
   b:= temp
end;
```

2nd Gen Languages

- Used block structure
- Structured control structures that eliminated the need for confusing networks of GOTO statements by hierarchically organising control flow
- Syntax structures were free format with machine independent reserved words

Scoping, Pointers, and Activation Records

AN INTERLUDE

Static Scoping

- Static scoping means that the scope of the program variable is the range of the statements in which it is visible
- The scope rules of a language determine how a particular occurrence of a name is associated with a variable
- A variable is considered to be local to a scope if it was declared there
- Non-local variables are the variables that are visible, but not declared there.
- ALGOL introduced a method of binding names to non-local variables called static scoping.
- Nearly all of today's languages do this in some way
- The visibility of these variables is determined at **compile time**

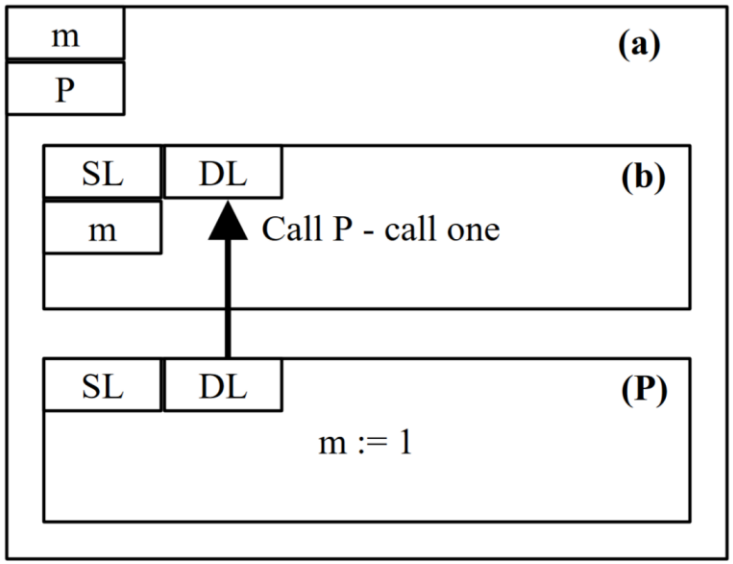
Why Static Scoping?

- The advantage is that it allows type checking to be performed by the compiler, cause it knows what that variable should be.
- Static scoping allows programmers to determine the connection binding between the use of the identifier and its declaration.
- It's obvious what things are!
- Consider the following:

The Following

```
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 is delayed till run time
- The correct attributes of a non-local variable cannot be determined statically, and certainly not by the compiler
- Anything refering to a non local variable could be referring to different things each time the statement is executed!

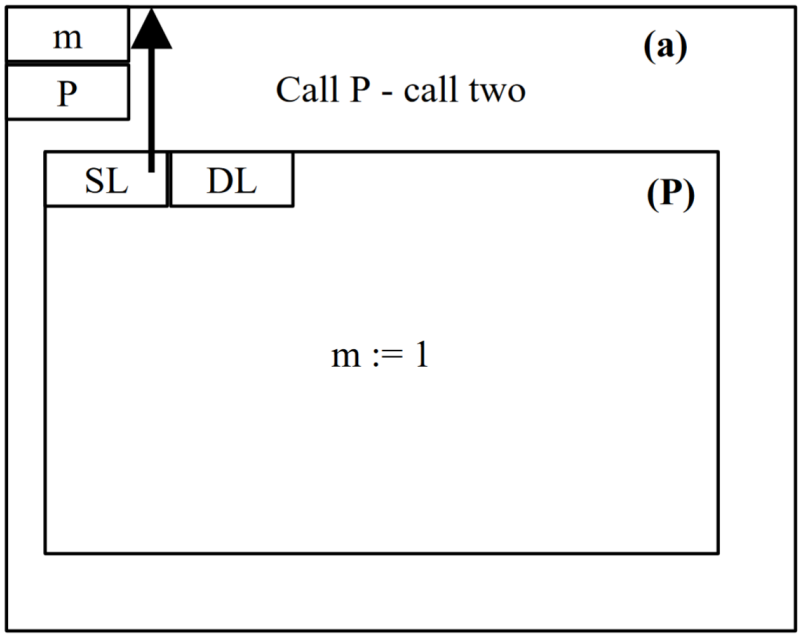
Problems with Dynamic Scope

- There are two major problems:
  - There is no way to type check references to non-locals at compile time
  - Dynamic scoping is confusing, and makes programs much harder to read
- Worst of all, during the time a subprogram is executed, the local variables are visible to any other subprogram. There is simply no way to protect your variables!

That same code again, now with dynamic scoping

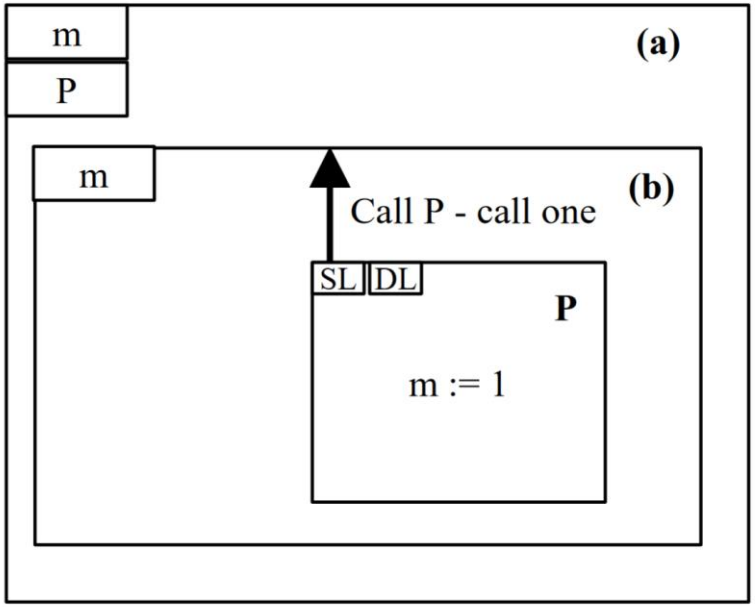
```
a:begin integer m;
procedure P;
m := 1;
b:begin integer m;
P; -> call 1;
end;
P; -> call 2;
end
```

Dynamic Scoping Example



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

Dynamic Scoping Example



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

Pointers

- A pointer type has a memory address in it as a value, and can have the value NIL or NULL
- The pointer has two uses
  - To provide some of the power of indirect addressing in assembly languages
  - To provide a better method of dynamic storage management
- An occurance of a pointer can be interpreted:
  - As a reference to the contents of a memory cell where a variable is bound.
  - As a reference to the value in the memory cell whose address is the one that the variable is bound to.

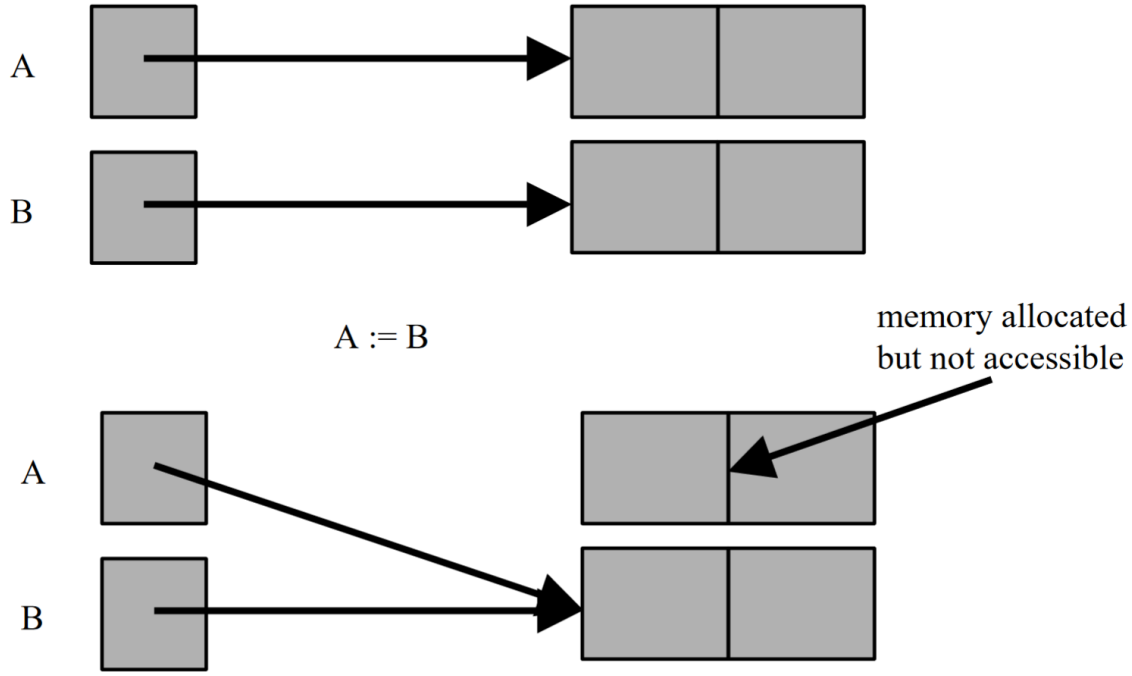
Less tricky pointers

- Pointers point to things, objects.
- They are most often used to implement dynamic structures (linked lists for example)
- Pointers are variables!
- Pointers hold:
  - Addresses
  - Addresses of variables
  - Addresses of functions
- A pointer is usually given the reference of something

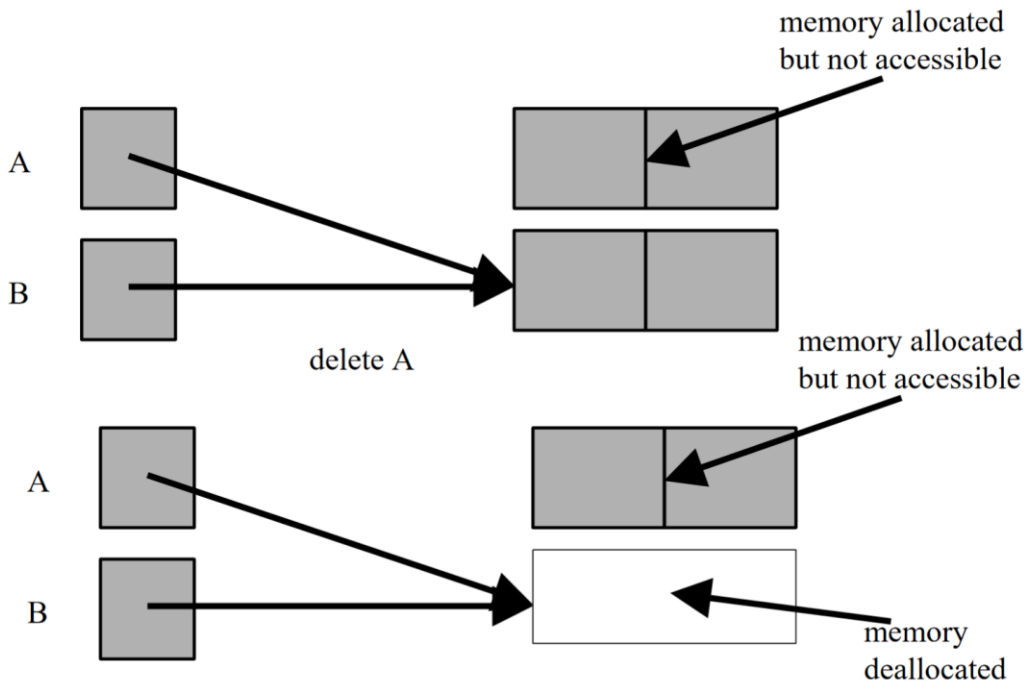
Dereferencing a pointer

- Dereferencing a pointer is the process of following the pointer to the variable or function that it points to.
- A pointer may be NULL. This is called a Dangling Pointer.
- If you dereference it, your program will break.
- Incorrect use of pointers can lead to memory leaks
- To deal with dangling pointers, one can use tombstones
- To deal with memory leaks, one can use reference counters or garbage collection

Memory Leaks



Dangling Pointers too!



Pointers v References

- References are pointers for Function calls
- Pointers:
  - You can change the value
  - You can change where it points
- References
  - You can change the value
  - You can't change where it points

Strong v Weak Typing

- Strong typing is very important for reliable languages as it allows elimination of type mismatch errors before run time
- There is no real definition on what Strong or Weak typing *is* per se, but basically a strongly typed language is one in which each name has exactly one type associated with it, and you have to declare it before run time
- Weak typing means the type of variable may not be known until run time and errors may go undetected
- But it's also very writeable, and extremely powerful.

Activation Records

- ALGOL allows procedures to be recursive, and there may be very many instances of the same procedure executing at the same time.
- What is needed is a method for the dynamic creation of activation records!
- To know the state of the procedure, it is neccisary to know:
  - The code that makes up the procedure
  - The place in the code where we are executing currently
  - The value of the variables visible to this instance

Activation Records

- The code doesn't vary between instances of procedures
- So you only need the bits that define the state
- Namely, where you are (the Instruction part)
- And what the variables are (the Environment part)

IP vs EP

- The Instruction part designates what the current instruction is
- The Environment part defines both the local *and* the non local conext for this instance of the function or subprogram.
- Context is simple, what names are bound in surrounding procedures?
- If there are multiple values for one name… which one is closest?
- If you want to check the value of a name, one looks it up in the Environment part.

Dynamic Links

- The local context contains:
  - Local variables
  - Parameters
  - Register
- This doesn't tell you who called you though
- To do that, a pointer is made to the calling activation record, and stored in the activation record alongside other contexts
- This is a Dynamic Link.
- Usually used to figure out where you need to go once you're done executing.

Static Links

- Static links point to the *bottom* of the activation record of the *next scope out* from the function or subprocess
- This is how global variables, class variables, and other scope visibilities work.
- There are two ways of accessing non-local variables in statically scoped languages:
  - Static Chains
  - Displays

Static Chains

- The actual address of a variable may change each excecution
- However, all variables are somewhere in the stack
- Step 1: Find the instance of the activation record in which the variable was allocated.
  - You do this by checking each enclosing scope sequentially until you find the one you need
- Step 2: Use the local offset of that variable within that record to access it

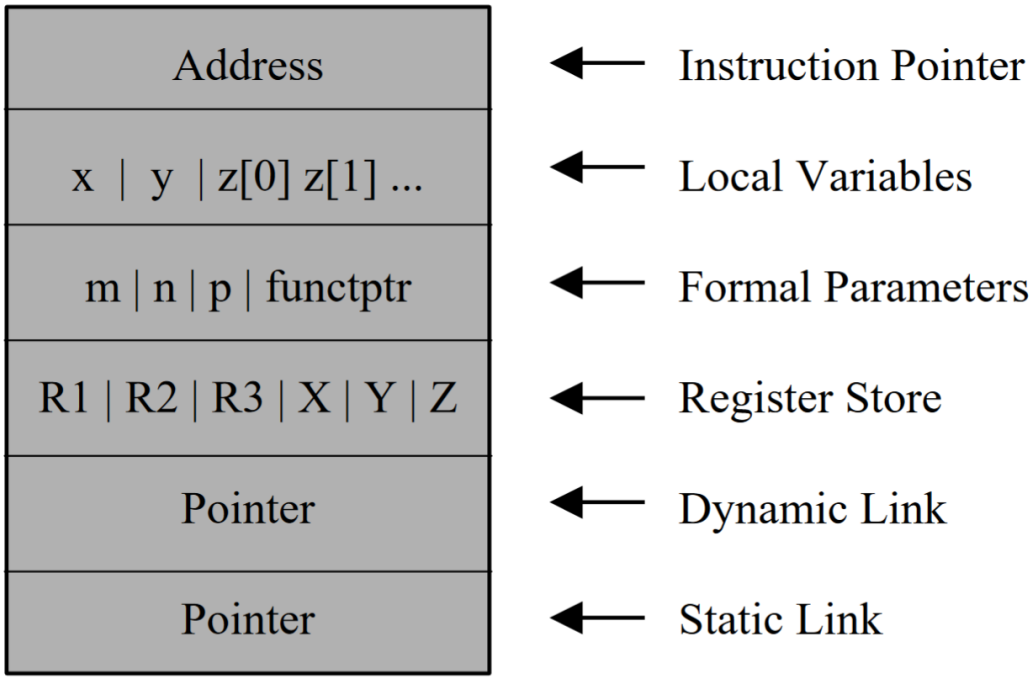
Display Technique

- Static links are collected into a single array called a display rather than being stored in the activation records
- This lets you walk through the display to find the variables you have access to rather than following an increasingly complex chain of pointers.
- Step 1: Find the activation that you belong to in the display
- Step 2: Using the display, find the offset to the scope you need and use that to calculate where the variable you want is.

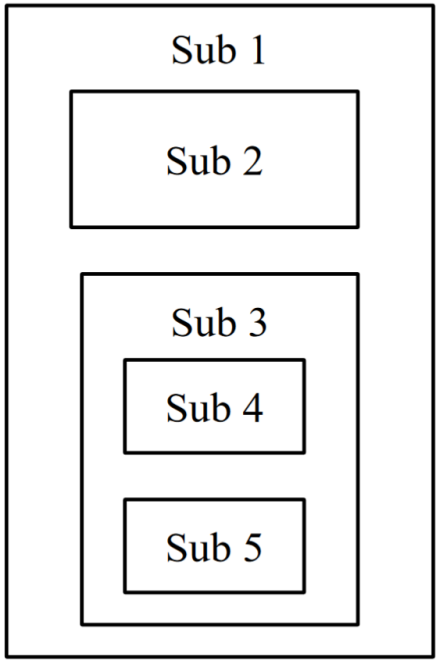
Static Chains v Display

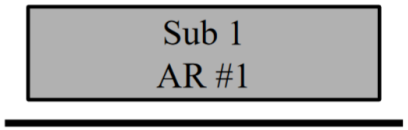
- Static chains can be slow, and require lots of time to walk through all the links
- Displays require more memory to actually build the display.
- Both have problems, one speed, one size.

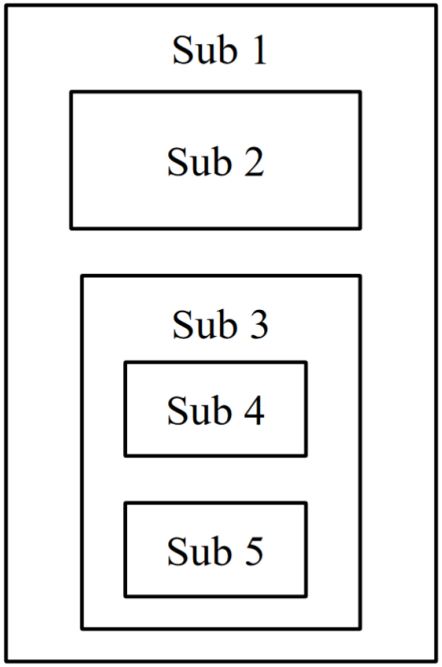
Activation records example

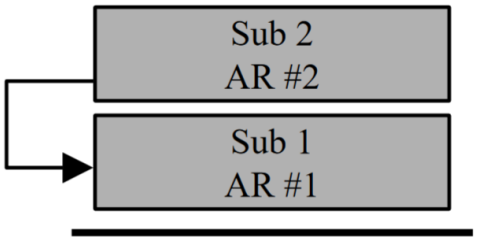


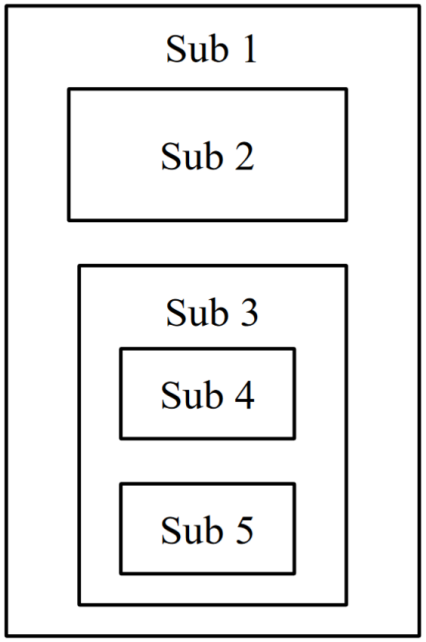
Dynamic Chain, Example

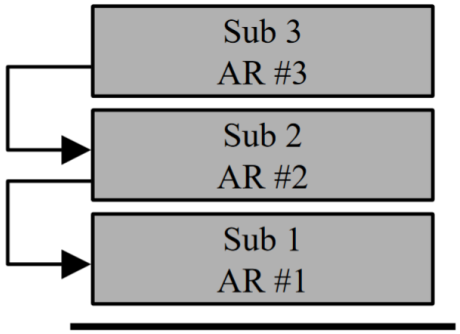


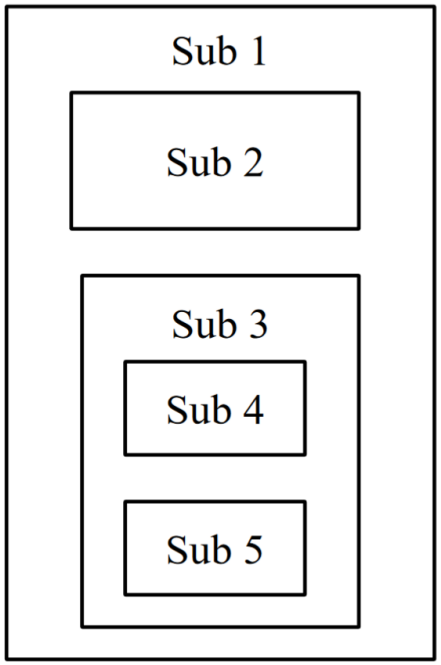


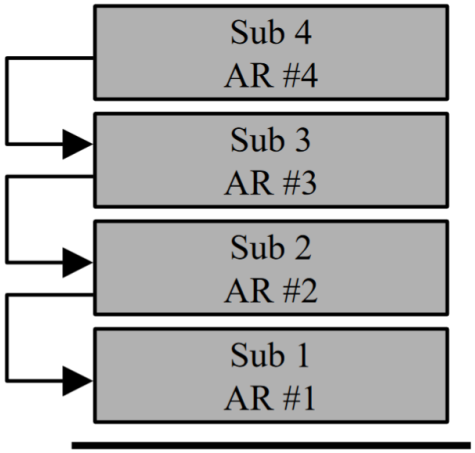

Sub 1
Sub 2
Sub 3
Sub 4
Sub 5

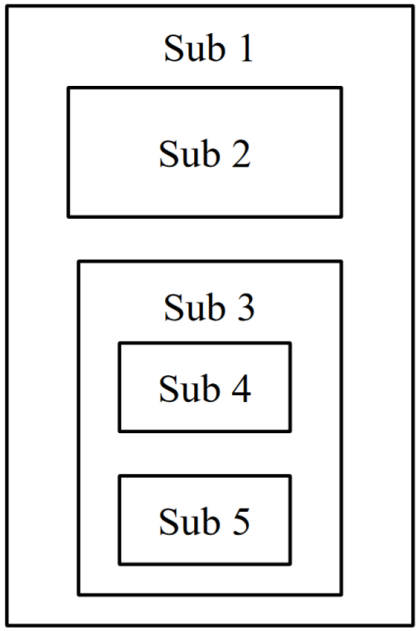

Sub 2
AR #2
Sub 1
AR #1

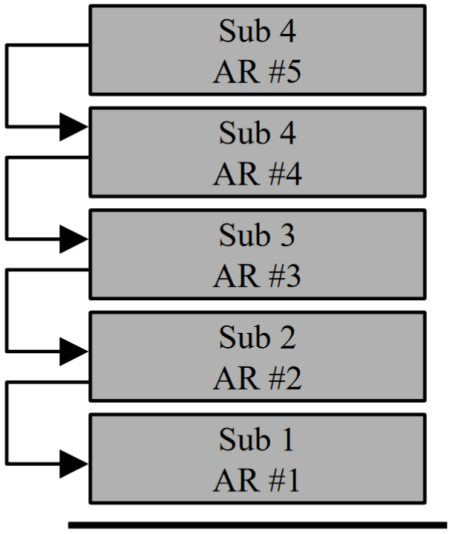

Sub 1
Sub 2
Sub 3
Sub 4
Sub 5

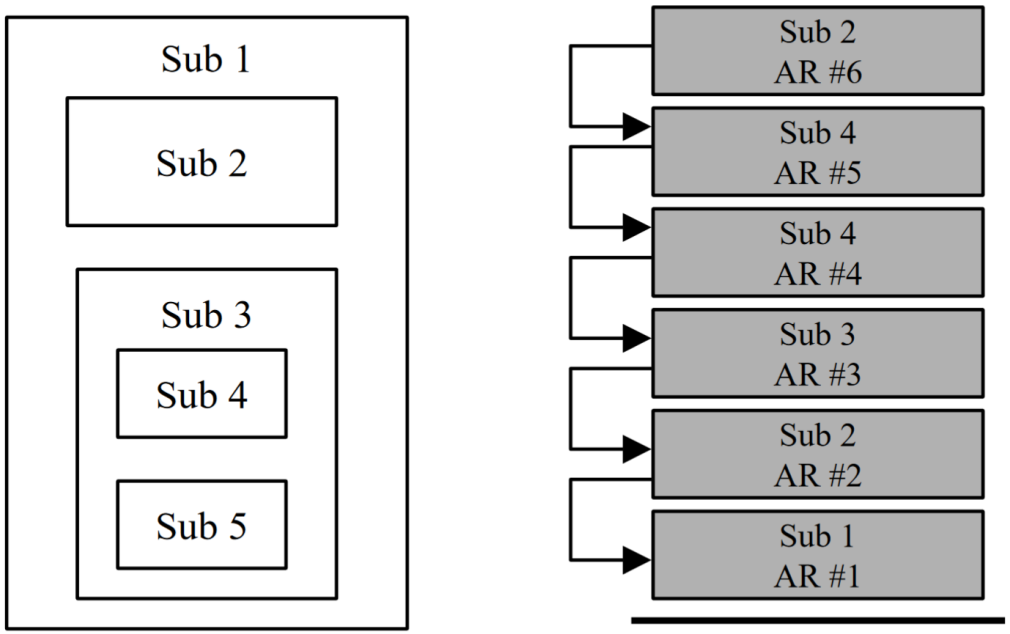

Sub 3
AR #3
Sub 2
AR #2
Sub 1
AR #1


Sub 1
Sub 2
Sub 3
Sub 4
Sub 5

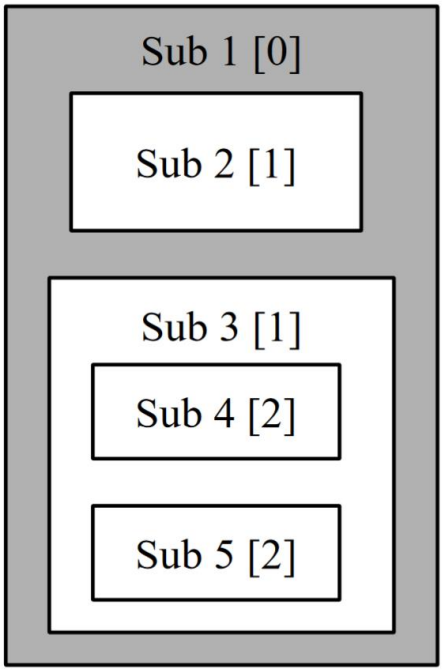

Sub 4
AR #4
Sub 3
AR #3
Sub 2
AR #2
Sub 1
AR #1

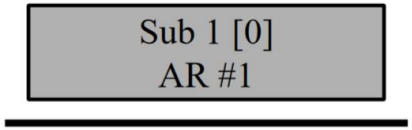

Sub 1
Sub 2
Sub 3
Sub 4
Sub 5

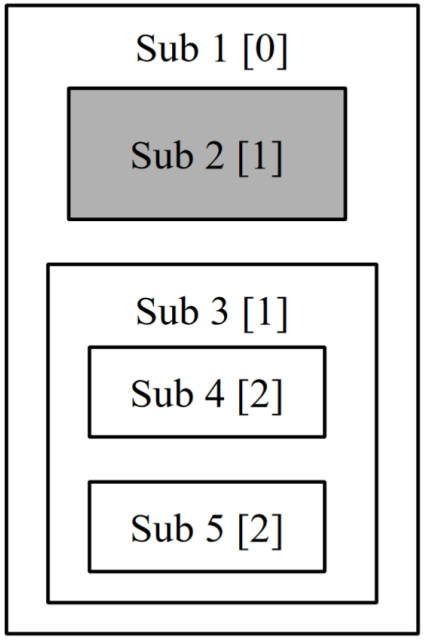

Sub 4
AR #5
Sub 4
AR #4
Sub 3
AR #3
Sub 2
AR #2
Sub 1
AR #1


Sub 1
Sub 2
Sub 3
Sub 4
Sub 5
Sub 2
AR #6
Sub 4
AR #5
Sub 4
AR #4
Sub 3
AR #3
Sub 2
AR #2
Sub 1
AR #1

Static Chain, Example



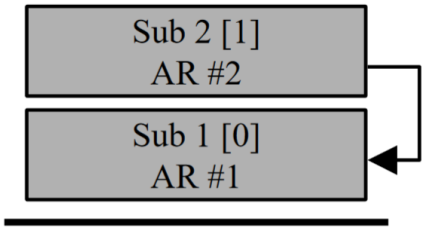


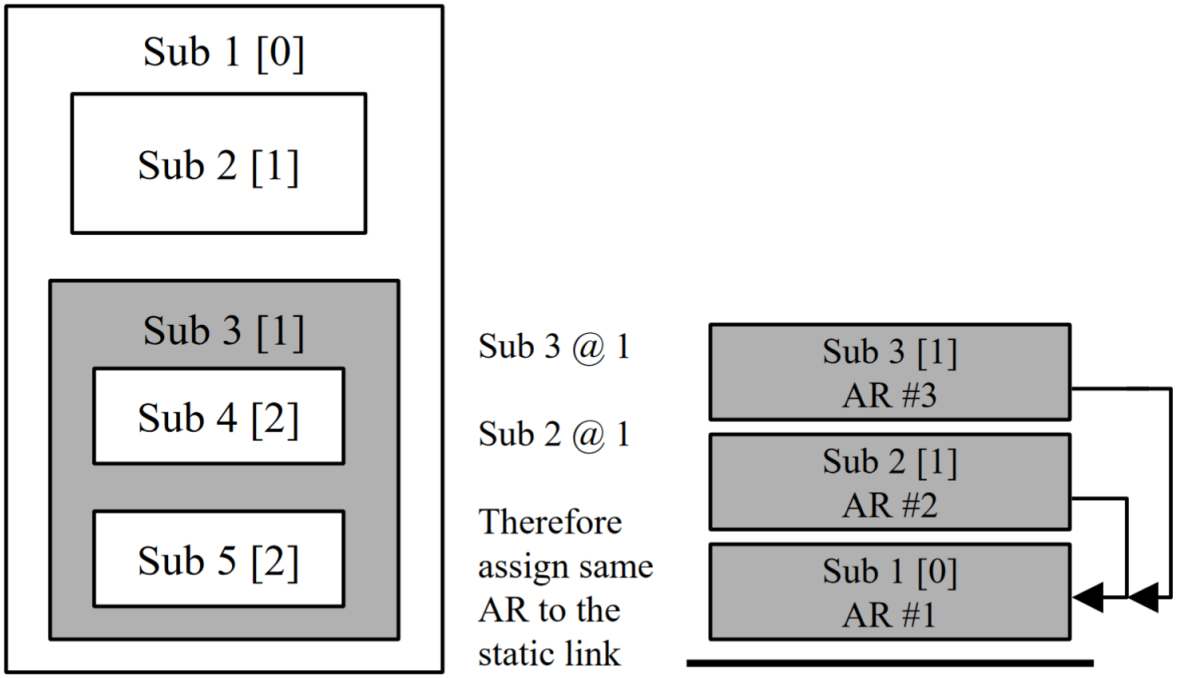


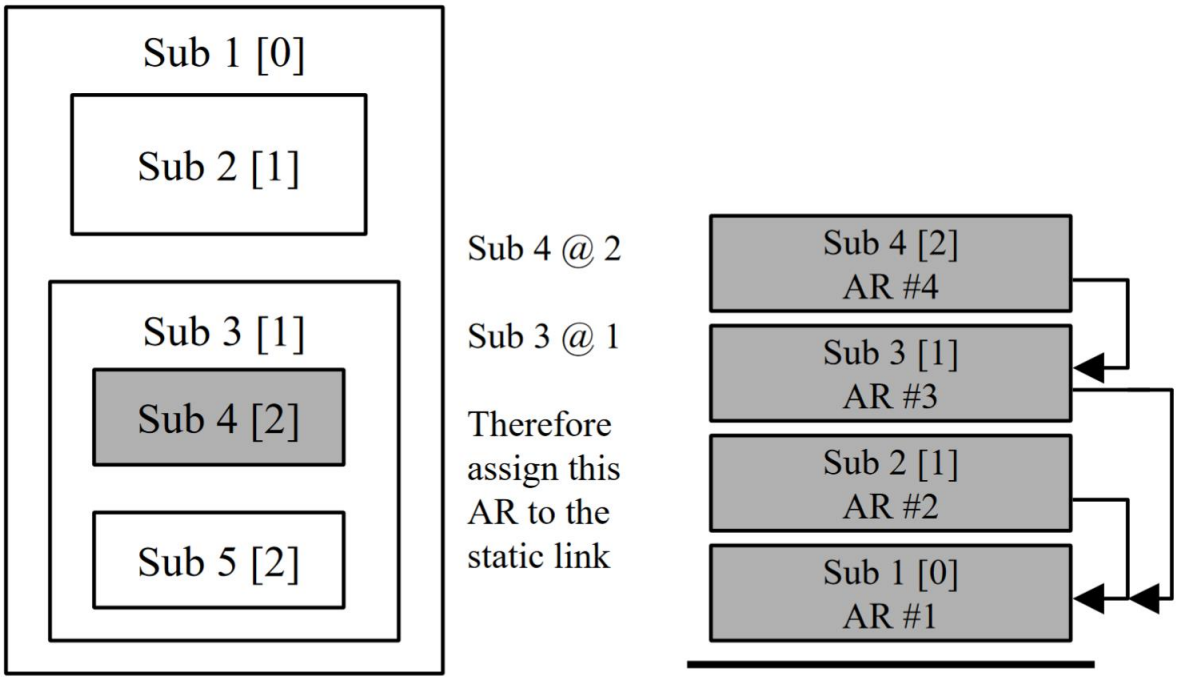
Sub 2 @ 1

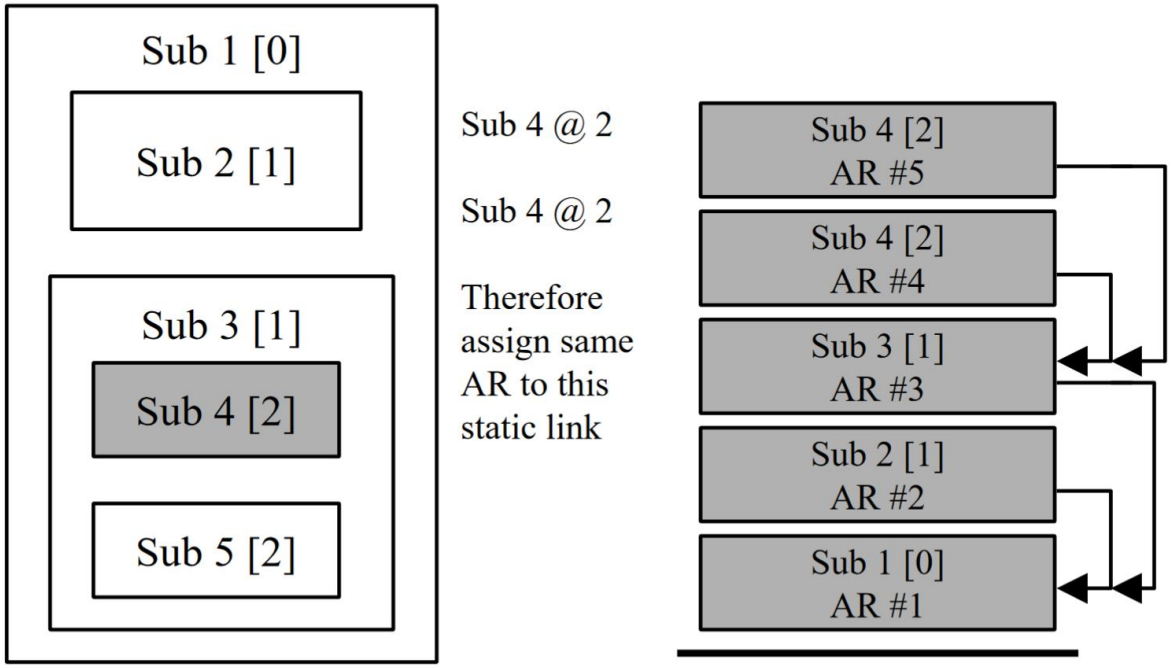
Sub 1 @ 0

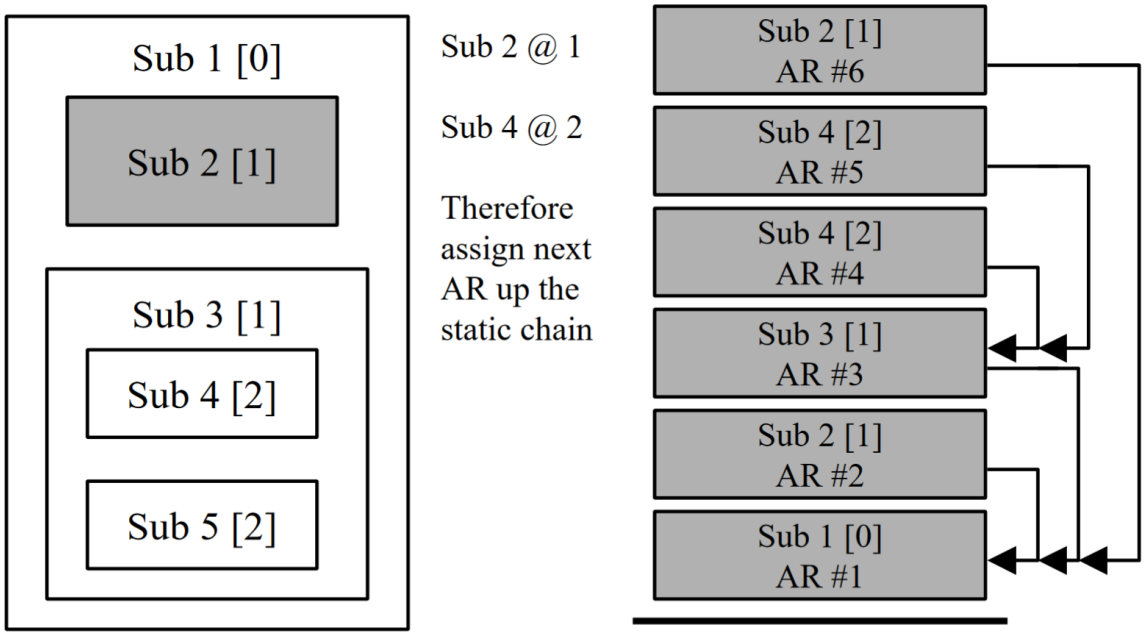
Therefore assign this AR to the static link




Sub 1 [0]
Sub 2 [1]
Sub 3 [1]
Sub 4 [2]
Sub 5 [2]
Sub 3 @ 1
Sub 2 @ 1
Therefore assign same AR to the static link
Sub 3 [1] AR #3
Sub 2 [1] AR #2
Sub 1 [0] AR #1


Sub 1 [0]
Sub 2 [1]
Sub 3 [1]
Sub 4 [2]
Sub 5 [2]
Sub 4 @ 2
Sub 3 @ 1
Therefore assign this AR to the static link
Sub 4 [2] AR #4
Sub 3 [1] AR #3
Sub 2 [1] AR #2
Sub 1 [0] AR #1


Sub 1 [0]
Sub 2 [1]
Sub 3 [1]
Sub 4 [2]
Sub 5 [2]
Sub 4 @ 2
Sub 4 @ 2
Therefore assign same AR to this static link
Sub 4 [2] AR #5
Sub 4 [2] AR #4
Sub 3 [1] AR #3
Sub 2 [1] AR #2
Sub 1 [0] AR #1


Sub 1 [0]
Sub 2 [1]
Sub 3 [1]
Sub 4 [2]
Sub 5 [2]
Sub 2 @ 1
Sub 4 @ 2
Therefore assign next AR up the static chain
Sub 2 [1] AR #6
Sub 4 [2] AR #5
Sub 4 [2] AR #4
Sub 3 [1] AR #3
Sub 2 [1] AR #2
Sub 1 [0] AR #1

Calling and Exiting Procedures

- Calling:
  - Save caller state
  - Transmit parameters to callee
  - Establish callee dynamic link
  - Establish callee static link
  - Enter the Callee
- Exiting:
  - Delete callee state
  - Restore caller state
  - Continue execution using the pointer address in the Instruction Part

Pascal

THE FIRST LEARNING LANGUAGE

The horror of PL/1

- In the early 1960s, IBM was a little annoyed that other people had developed languages
- This made it hard to sell their language/computer package as “the best”
- So they made a new language, PL/1
- PL/1 was made of every language to date, so that any code would work. Now everyone can use it!
- This was, as one could expect, horrible. Now totally valid code in PL/1 could not be necessarily understood by two different programmers.
- Computer scientists hated it so much they called it a cancer of programming (quite an insult!)

The aftermath of PL/1

- Everyone wanted to fix the problems leading up to PL/1, namely in how hard it was to learn new languages, but nobody had any good ideas.
- Extensible languages based on microkernel like ideas were tried, but for the same reason microkernels failed, they failed too.
- Algol was revised by Wirth – Algol-W – To be an instructional language
- But this sacrificed so much to do so it was basically unusable.
- In 1968, work to develop a new teaching language that wasn't a huge compromise began, called Pascal

Pascal: Goals

- Pascal wanted to achieve two main things:
  - Simplicity – It wanted to be a simple teaching language
  - Efficiency – It wanted to do this without sacrificing efficiency at compile and run times.
- ALGOL-60 was still the best language anyone had, so it played a major role in the design of Pascal
- Pascal borrowed the general syntactic format and block structured layout of ALGOL, but tried to make it simpler
- Pascal was completed by 1970, an extraordinary achievement, and indicative of how much people disliked PL/1!
- Pascal even had a concurrent version, available by 1975!

Pascal: Structure

- Pascal is formed from a single main program block that contains all variables, types, and subprograms that are used.
- This means it can only ever be exactly one huge file.
- Pascal can, like ALGOL, be nested to as deep as a programmer likes
- Each block in Pascal has:
  - A header, this specifies the parameters and results
  - Then, definitions of constants, types, and local variables
  - Then local subprogram definitions and finally,
  - The body of the program.

Declarations

- Variable declarations
  - Name : type
  - So something like “X : Integer”
- Type declarations
  - Name = type
  - So something like “Newtype = Array”
- Pascal introduced the Char! Now you can have actual strings!

The many types of Pascal

- Pascal had many types. These are:
  - Enumerated type
  - Subrange type
  - Set type
  - Array type (you can't index them from anywhere like Algol, but you *can* index them with other types)
  - Record type (Groups data of different types, like an array with different data types in it. Almost like a C Struct)
  - Pointer type

Further Pascal Weirdness

- There are two types of subprograms in Pascal, functions and procedures.
- Functions return values. Procedures do not.
- There are six ways of binding names to their meanings:
  - Constant bindings
  - Type bindings
  - Variable bindings
  - Procedure/function bindings
  - Implicit enumeration bindings
  - Label bindings
- There are also two name constructors, Records (simple and variant) and Procedures

Control Structures in Pascal

- Pascal has many more control structures than in previous languages, but does so in order to try and teach people not to use the Goto.
- Goto still exists though.
- Control structures enforced single entry and single exit points, which encouraged structured (that is to say, Goto Free) programming.
- The for loop is *very* simple in Pascal. You can only increment or decrement by one each loop.
- Repeat-until and while-do allow for infinite loops
- Pascal introduced the first true Case statement, complete with self documentation and efficient construction.
- The Case statement allows options to be out of order. This complies with the **Labelling principle**

Parameter Passing

- Pascal uses the same static scope rules that ALGOL-60 used for variable scoping.
- When passing is required, Pascal uses pass-by-value and pass-by-reference
- Thankfully, Pascal did not copy pass-by-name.
- Pass-by-constant was originally part of Pascal, but due to confusion that could be brought on with scope based aliasing… this was quickly removed.

The Record Type

- The record type is more general than the array type.
- Kind of like a C Struct, the components can be anything, even records, and are bound to names that can be accessed with the "." (dot) operator.
  - Eg: "record.interiorRecord.somethingElse"
- The record defines its own scope, just like the C Struct. You can't just state "somethingElse", you need to use the "."s to get to it.
- Variant records are a special record that may have the same name… but may or may not have the same components.
- At the time, this was horrible. Later languages better structured this with inheritance.

Language Design Issues

- Pascal has the forward declaration problem, just like C. This is to allow one-pass compilers.
- Pascal was, and still is, a very successful language.
- Still had flaws:
  - Forward declaration: Now you have to think of how the program will appear in memory as well as what you want it to do. Bad.
  - The array was a separate type, it should have been a collection of objects of the same type.

Further issues: Local Variables

- Pascal structures a program like this:
  - Header
  - Local variables
  - Local procedures
  - Begin <body> end;
- Note that every procedure must come after the local variables and before they are used.
- If you need to change or add a variable, there's a lot of scrolling to be done.
- This makes programs hard to understand or document if they're long.

Even more issues

- Parameter passing is a problem
  - In order to tell Pascal that you're passing by reference, you need to put the keyword "var" before the parameter. Forgetting this silently configures for pass by value instead. This *works* still, so you don't know it's happened.
- The language does not support the process of compiling modules independently
  - You can't have more than one file. Ever. All the code you will need must be in one file. No extensible libraries, no shared objects, nothing.
  - This is what modern versions of Pascal (Turbo Pascal, for example) seek to fix.

3rd Generation Languages

- Characterised by:
  - Emphasis on simplicity and efficiency
  - Data structures emphasise application
    - Programmers were given tools with which to define data types for their applications
  - Nested data structures to any depth
  - Simpler name structures
  - Simpler control structures
  - Introduction of control structures that were application oriented while still maintaining efficiency (ie, case statement)
- C was one of these, so we've now reached languages you know!