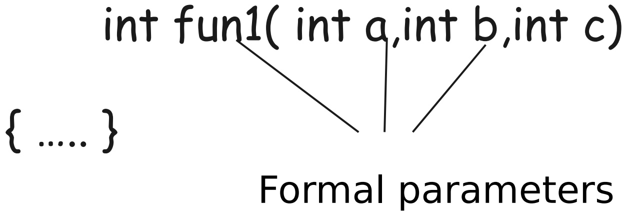
Recitation - 03

Jahnavi - jp5867@nyu.edu

Formal and Actual Parameters:

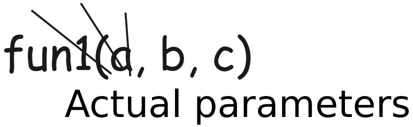
Formal Parameters: The identifier used in a method to stand for the variable that is passed into the method by a caller.



```
int fun1( int a,int b,int c)
{ ….. }
```

Formal parameters

Actual Parameters: The actual variable/value that is passed into the method by a caller.



```
fun1(a, b, c)
```

Actual parameters

Parameter passing methods:

Mechanism which determines what kind of association is in between the formal and actual parameters.

- Pass by value
- Pass by copy-return
- Pass by reference
- Pass by name
- Pass by return

Pass by value:

Formal parameters are bound to value of actual parameters - C, Java

-> Formal = Actual.copy()

Use pass by value when when you are only using the parameter for some computation, not changing it for the client program.

Java uses only pass by value. However, the context changes depending upon whether we're dealing with Primitives or Objects:

-> For Primitive types, parameters are pass-by-value

-> For Object types, the object reference is pass-by-value

Pass by reference:

-> Formal parameters are bound to location of actual parameters. - Pascal and C++

-> We pass the reference of an argument in the calling function to the corresponding formal parameter of the called function.

Advantages : Less overhead

```
#include <stdio.h>

void swapnum(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;

  swapnum(a, b);
  printf("A is %d and B is %d\n", a, b);
  return 0;
}
```

Pass by name:

-> Formal parameters are evaluated every time they are used. They won’t be evaluated at all if they are unused. This is similar to replacing the by-name parameters with the passed expressions - Algol 60

-> A language that implements call-by-name passes a pair of functions that provide read and write access to the variable.

-> In case of simple variables call by name is similar to call by reference.

```
int i;
int a[ ] = new int[2];
void p(x) {
         i = i + 1;
         x = x + 1;
}
void main() {
       i = 0;
       a[0] = 0;
       a[1] = 1;
       p( a[i] );
}
```

Pass by name will change the value of a[1]

Pass by value-return:

-> In this, the value of the actual parameters are copied into the called method Activation Record.After the called method ends, the final values of the parameters are copied back into the arguments - Ada, FORTRAN

-> Value-result is equivalent to call-by-reference except when there is aliasing.

```
int x = 0;
int main() {
  foo(x);
}

void foo(int a) {
  x = 3;
  a++;
}
```

Value of x : 1

Pass by result:

-> No value is transmitted to the caller

-> Formal parameter acts as a local variable, and is copied to the caller when the control is passed back to the caller.

This is used for OUT params

Example 1:

```
program example ;
var
    global : integer := 10;
    another : integer := 2 ;
    procedure confuse (var first , second : integer ) ;
    begin
         first := first + global ;
        second := first * global ;
    end ;
begin
confuse (global , another ) ;
```

Pass by value: global:=10; another:=2
Pass by value-result: global:=20 ;another:=200
Pass by reference: global:=20 ; another:=400
Pass by name: global:=20; another:=400

Example 2:

```
begin
    integer n;
    procedure p(k: integer);
    begin
        n := n+1;
        k := k+4;
        print(n);
    end;
    n := 0;
    p(n);
    print(n);
end;
```

Pass by value: 1 1
Pass by value-result: 1 4
Pass by reference: 5 5
Pass by name: 5 5

Example 3:

```
begin
    array a[1..10] of integer;
    integer n;
    procedure p(b: integer);
    begin
        print(b);
        n := n+1;
        print(b);
        b := b+5;
    end;
    a[1] := 10; a[2] := 20; a[3] := 30; a[4] := 40;
    n := 1; p(a[n+2]);
    print(a);
end;
```

```
Pass by value: 30 30 [10, 20, 30, 40]
Pass by value-result: 30 30 [10, 20, 35, 40]
Pass by reference: 30 30 [10, 20, 35, 40]
Pass by name: 30 40 [10, 20, 30, 45]
```

Example 4:

```
int i = 1;
void p(int f, int g) {
        g++;
        f = 5 * i;
}
 int main() {
        int a[] = {0, 1, 2};
        p(a[i], i);
        printf("%d %d %d %d\n", i, a[0], a[1], a[2]);
}
```

Pass by value: 1 0 1 2
Pass by value-result: 2 0 5 2
Pass by reference: 2 0 10 2
Pass by name: 2 0 1 10

Example 5:

```
program main;
        i: integer;
        a: array[1..2] of integer;
        procedure f(x:integer, y:integer)
        begin
                x := x + 1;
                y := y + 1;
        end
begin
        i := 1;
        a[1] := 1; a[2] := 2;
        f(i,a[i]);
        print(a[1],a[2]);
end;
```

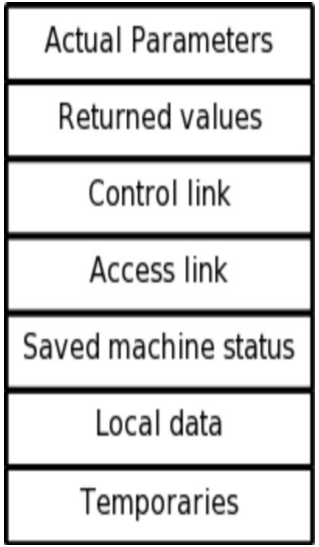
Pass by value: 1 2
Pass by value-result: 2 2
Pass by reference : 2 2
Pass by name : 1 3

Activation Record:

The information needed for each invocation of a procedure is kept in a runtime data structure called an **activation record**

Contents of AR:

1. Temporaries. - When the temporaries generated during expression evaluation.cannot be held in the registers, then the temporary area is used.
2. Data local to the procedure being activated.
3. Saved status from the caller, which typically includes the return address and the machine registers. The register values are restored when control returns to the caller.
4. The access link
5. The control link connects the ARs by pointing to the AR of the caller.
6. The returned value.
7. Actual parameters



Actual Parameters

Returned values

Control link

Access link

Saved machine status

Local data

Temporaries

Access non-local data:

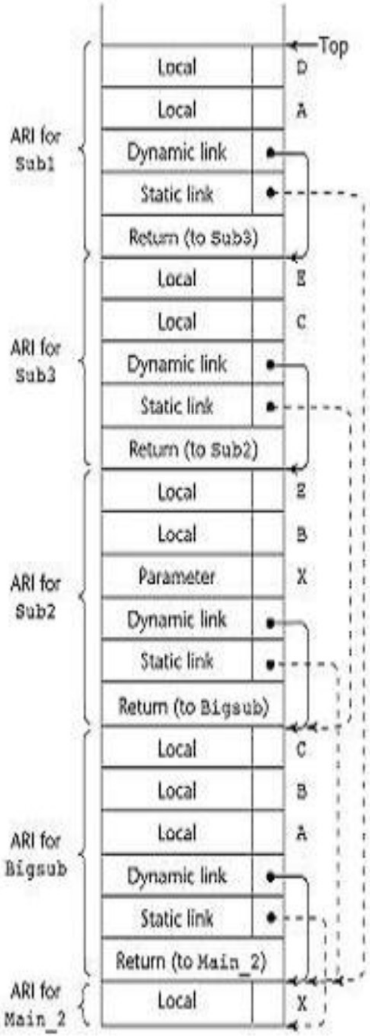
While looking for the non-local variables, we first must find the correct activation record instance and then determine the correct offset within that activation record instance.

1. **Static Links:**

   The static link in an activation record instance for subprogram *A* points to one of the activation record instances of *A*'s static parent.

   Here, if you want to look for a variable, you have to go down along the static links. Hence, it takes the time of nesting depth.

```
program MAIN_2;
|   var X : integer;
|
|   procedure BIGSUB;
|   |  var A, B, C : integer;
|   |
|   |  procedure SUB1;
|   |  |  var A, D : integer;
|   |  |  begin
|   |  |      A := B + C;  #1 references A in SUB1 (0,2)
|   |  |_ end;  { SUB1 }
|   |
|   |  procedure SUB2(X : integer);
|   |  |  var B, E : integer;
|   |  |
|   |  |  procedure SUB3;
|   |  |  |  var C, E : integer;
|   |  |  |  begin { SUB3 }
|   |  |  |    SUB1;
|   |  |  |    E := B + A:    #2 references A in BIGSUB (2,3)
|   |  |  |_ end; { SUB3 }
|   |  |
|   |  |  begin { SUB2 }
|   |  |    SUB3;
|   |  |    A := D + E;       #3 references A in BIGSUB (1,3)
|   |  |_ end; { SUB2 }
|   |
|   |  begin { BIGSUB }
|   |    SUB2(7);
|   |_ end; { BIGSUB }
|
|   begin
|     BIGSUB;
|_   end; { MAIN_2 }
```



Access non-local data:

2. Displays:

-> This is the optimised approach for non-local data look up.

-> A more efficient implementation uses an auxiliary array d, called the display, which consists of one pointer for each nesting depth. We arrange that, at all times, d[i] is a pointer to the highest activation record on the stack for any procedure at nesting depth i. In this implementation static links are NOT stored in the activation record but in a single array.

-> Accesses to non local variables require exactly two steps for every access: display_offset + local_offset takes you directly to the address.

-> Displays not widely used in modern languages.

Higher order functions:

-> Functions as first class values forces heap allocation of activation record rather than stack allocation.

-> The environment of definition of function must be preserved until the point of call. Hence, functional languages require more complex run-time management rather than stack based allocation.

Higher order functions are the ones that returns functions. But, the imperative languages like C, C++, Java restrict their use.

Higher order functions and scoping:

```
type Func is access function (x: integer) return integer;
function compose (first, second : Func) return Func is
begin
        function result( x : integer) return integer is
        begin
        return  ( second ( first ( x ) ) )
        end
return result'access;
End
```

**This is illegal in Ada, because First and Second won't exist at point of call.

Thank you