

ORDER OF PROCEDURE DECLARATIONS

- In **block structured languages** (where procedures and functions may be nested) the order in which procedures are declared also effects their visibility.
- Ada (and other block structured languages such as Pascal) require that any use of an identifier must be preceded by its declaration.
- The procedure currently being declared cannot see any following procedures.
- This can be overcome by using an **incomplete declaration** (called a **prototype** in C):

```
#include <stdio.h>

float meanValue(float, float);

void main(void) {
    float x, y;

    printf("Enter 2 real numbers ");
    scanf("%f%f",&x,&y);

    printf("\nThe mean is %f\n",meanValue(x,y));
}

float meanValue(float x1, float x2){
    return ((x1+x2)/2.0);
}
```

C
meanValue
function

```
#include <stdio.h>

void meanValue(float, float);

void main(void) {
    float x, y;

    printf("Enter 2 real numbers ");
    scanf("%f%f",&x,&y);

    meanValue(x,y);
}

void meanValue(float x1, float x2) {
    printf("\nThe mean is %f\n",(x1+x2)/2.0);
}
```

C
meanValue
procedure

COMP205 IMPERATIVE LANGUAGES

13. PROGRAM COMPOSITION II

- 1) Scope
 - a) Ada scope rules
 - b) C scope rules
- 2) Parameter passing
 - a) Ada parameter modes
 - b) Parameter passing mechanisms

SCOPE RULES

- **Scope rules** govern the visibility and life time of data items (i.e. the parts of a program where they can be used).
- They also **bind** names to types.
 - 1) Where this is determined at compile time this is called **static binding**. The opposite is **dynamic binding**.
 - 2) Dynamic scope is a feature of logic languages such as PROLOG and functional languages such as LISP.

The advantages of static scoping is that it allows type checking to be carried out at compile time.
- In most imperative languages the scope of a declaration starts at the end of the declaration and extends to the end of the block.

ADA SCOPE RULES

[A] A data item is visible from where it is declared to the end of the block in which the declaration is made.

```
procedure SUM_AND_PROD is
  X_ITEM : constant := 2;
  Y_ITEM : constant := X_ITEM*2;
begin
  PUT(X_ITEM+Y_ITEM);
  NEW_LINE;
  PUT(X_ITEM*Y_ITEM);
  NEW_LINE;
end SUM_AND_PROD;
```

[B] A data item declared within a block cannot be seen from outside that block, and is referred to as local data item, i.e. local to a block.

```
-----
procedure SUM(P_ITEM, Q_ITEM : INTEGER) is
  TOTAL : INTEGER ;
begin
  TOTAL := P_ITEM + Q_ITEM;
  PUT(TOTAL);
  NEW_LINE;
end SUM;
-----
procedure PRODUCT(S_ITEM, T_ITEM : INTEGER) is
  PROD : INTEGER ;
begin
  PROD := S_ITEM * T_ITEM;
  PUT(PROD);
  NEW_LINE;
end PRODUCT;
-----
```

[C] Anything declared in a block is also visible in all enclosed blocks within it (in which the data item is not redeclared).

```
procedure SUM_AND_PRODUCT(P_ITEM, Q_ITEM: INTEGER) is
  ANSWER : INTEGER;
-----
  procedure SUM(A_ITEM,B_ITEM: INTEGER) is
    begin
      ANSWER := A_ITEM + B_ITEM;
    end SUM;
-----
  procedure PRODUCT(C_ITEM,D_ITEM: INTEGER) is
    begin
      ANSWER := C_ITEM * D_ITEM;
    end PRODUCT;
-----
begin
  SUM(P_ITEM,Q_ITEM); PUT(ANSWER); NEW_LINE;
  PRODUCT(P_ITEM,Q_ITEM); PUT(ANSWER); NEW_LINE;
end SUM_AND_PRODUCT;
```

[D] Several data items with the same name cannot be used in the same block, however several items with the same name can be declared in "different" blocks.

- In this case the declared quantities have nothing to do with one another (other than sharing the same name).

```

-----
procedure SUM(P_ITEM, Q_ITEM : INTEGER) is
  TOTAL : INTEGER;
begin
  TOTAL := P_ITEM + Q_ITEM;
  PUT(TOTAL);
  NEW_LINE;
end SUM;
-----
procedure PRODUCT(P_ITEM, Q_ITEM : INTEGER) is
  TOTAL : INTEGER;
begin
  TOTAL := P_ITEM * Q_ITEM;
  PUT(TOTAL);
  NEW_LINE;
end PRODUCT;
-----

```

[E] Where a name is used in a block and reused in a sub-block nested within it, the sub-block declaration will override the super-block declaration during the life time of the sub-block.

- This is referred to as *occlusion*. The identifier which is hidden because of the inner declaration is said to be occluded.

```

procedure SUM_AND_PROD is
  X_ITEM : constant := 2;
  Y_ITEM : constant := X_ITEM*2;
-----
  procedure SUM(P_ITEM, Q_ITEM : INTEGER) is
    X_ITEM : INTEGER;
  begin
    X_ITEM := P_ITEM + Q_ITEM;
    PUT(X_ITEM);
    NEW_LINE;
  end SUM;
-----
begin
  SUM(X_ITEM, Y_ITEM);
end SUM_AND_PROD;

```

ADA SCOPE RULES SUMMARY

- 1) Scope of declaration extends from where it is made to the end of the block in which it is made.
- 2) Anything declared in a block is not visible outside of that block.
- 3) Anything declared in a block is visible to all enclosed blocks within it.
- 4) Several items with the same name cannot be used in the same block, however several items with the same name can be declared in different blocks.
- 5) Where a name is used in a block and reused in a sub-block nested within it, the sub-block declaration will override (occlude) the super-block declaration during the life time of the sub-block.

C SCOPE RULES

C SCOPE RULES

- The scope of a data item in C is governed by its *storage class*.
- Generally data items belong to one of two storage classes:
 - 1) `extern` (external) - used to define global variables visible throughout a program.
 - 2) `auto` (automatic) - used to define local variables (including formal parameters to functions) visible only inside a block. They exist only while the block of code in which they are declared is executing.
- Note that a local variable overrides a global variable of the same name.
- C also supports two other storage classes `static` and `register`.

PARAMETER PASSING

- a) Ada parameter modes
- b) Parameter passing mechanisms

PARAMETER PASSING

- Parameters can be classified into three groups or modes. They can be used to:
 - 1) Pass information to a routine.
 - 2) Receive information from a routine.
 - 3) Pass information to a routine where it is updated before being returned.
- Ada identifies these as the `in`, `out` and `in out` parameter modes.

```

procedure EXAMPLE is
  X: integer:= 1;
  Y: integer:= 2;
  Z: integer:= 3;

  procedure DO_IT (A: in integer;
                  B: in out integer;
                  C: out integer) is
  begin
    C:= B; B:= B+A;
  end DO_IT;

begin
  put(X);put(Y); put(Z); new_line;
  DO_IT(X,Y,Z);
  put(X);put(Y);put(Z); new_line;
end EXAMPLE;

```

ADA PARAMETER MODES

Note that by default a parameter is an `in` parameter.

1	2	3
1	3	2

PARAMETER PASSING MECHANISMS

- We can identify a number of types of parameter passing mechanism:
 - 1) Call by value
 - 2) Call by constant value.
 - 3) Call by reference.
 - 4) Call by reference-value.
 - 5) Call by result.
 - 6) Call by copy-restore or value-result.

CALL BY VALUE

- **Call by value** is the most common and is used by languages such as C, Pascal, Modula-2, ALGOL 60.
- The formal parameter acts as a local variable which is initialised with the value of the actual parameter and may then be changed.
- However any changes made to the formal parameter will no effect the value of the actual parameter.

```

void main(void) {
  int a = 1, b = 2, c = 3;

  printf("Before doit:  a=%d, b=%d, c=%d\n",a,b,c);
  doit(a,b,c);
  printf("After doit:   a=%d, b=%d, c=%d\n",a,b,c);
}

• • •

void doit(int a, int b, int c) {
  printf("Start of doit: a=%d, b=%d, c=%d\n",a,b,c);
  a = b*100; b = b+c;
  printf("End of doit:  a=%d, b=%d, c=%d\n",a,b,c);
}

```

```

Before doit:  a = 1, b = 2, c = 3
Start of doit: a = 1, b = 2, c = 3
End of doit:  a = 200, b = 5, c = 3
After doit:   a = 1, b = 2, c = 3

```

CALL BY CONSTANT VALUE

- In languages such as Ada (also ALGOL 68) the formal parameter is a local constant rather than a local variable and thus can not be changed.
- The latter mechanism is thus referred to as **call by constant value**.

```

procedure EXAMPLE is
  N1: float:= 4.2;
  N2: float:= 9.6;

  procedure MEAN_VALUE
    (X1, X2: in float) is
  begin
    put((X1+X2)/2.0);
    new_line;
  end MEAN_VALUE;

begin
  MEAN_VALUE(N1, N2);
end EXAMPLE;

```

CALL BY REFERENCE

- The disadvantage of call by value (and call by constant value) is that a copy is always made of the actual parameter to obtain the formal parameter.
- The need to make a copy can be avoided using the **call by reference** mechanism.
- Found in languages such as ALGOL 68, MODULA II and Pascal using VAR parameter mode.
- In call by reference the formal parameters become "aliases" for the actual parameters. Consequently everything that happens to the formal parameters also happens to the actual parameters.
- Approach has all the dangers of aliasing.

```

program PASSING (input, output);
var VALUE_1, VALUE_2: real;

-----
procedure MEAN_VALUE (NUM_1: real; var NUM_2: real);
begin
  NUM_2 := (NUM_1+NUM_2)/2;
end;
-----

begin
  readln(VALUE_1, VALUE_2);
  write('VALUE_1 = '); write(VALUE_1);
  write(', VALUE_2 = '); write(VALUE_2); writeln;
  MEAN_VALUE(VALUE_1, VALUE_2);
  write('MEAN VALUE_2 = '); write(VALUE_2); writeln;
end.

```

CALL BY REFERENCE VALUE

- Supported by C
- In call by reference value the address of the actual parameter is passed.
- This can then be used by the called routine to access the actual parameters.
- Consequently, in C, we can simulate the effect of call by reference using the call by reference value mechanism.

```

void main(void) {
float n1 = 4.2, n2 = 9.6;
meanValue(&n1, n2);
printf("Mean = %f\n",n1);
}
n1 = 4.200000, n2 = 9.600000
Mean = 6.900000

void meanValue(float *n1, float n2) {
printf("n1 = %f, n2 = %f\n",*n1,n2);
*n1 = (*n1+n2)/2.0;
}

```

Remember that:

- The * operator when used in a declaration defines a "pointer to a data type".

- When used elsewhere it is interpreted as "the value contained at the address pointed at" (i.e. the pointer is dereferenced).
- The operator & is interpreted as "the address of ...".

CALL BY RESULT

- Call by result is used in Ada to implement out mode parameters.
- The formal parameter acts as an uninitialised local variable which is given a value during execution of the procedure.
- The value of the formal parameter is then assigned to the actual parameter on returning from the routine.

```

procedure EXAMPLE is
  N1: float:= 4.2;
  N2: float:= 9.6;
  MEAN: float;

  procedure MEAN_VALUE
    (X1, X2: in float; X3: out float) is
  begin
    X3:= (X1+X2)/2.0;
  end MEAN_VALUE;

begin
  MEAN_VALUE(N1, N2, MEAN);
  put(MEAN);
  new_line;
end EXAMPLE;

```

CALL BY COPY RESTORE

- **Call by copy restore** is an 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 routine, changes to the formal parameter only affect the local copy.
- On returning from the routine the final value of the formal parameter is assigned to the actual parameter.
- Call by copy restore is supported by Ada to achieve "in-out" parameter operation.
- Has the same disadvantages as those associated with call by value.

```

procedure EXAMPLE is
  N1: float:= 4.2;
  N2: float:= 9.6;

  procedure MEAN_VALUE (X1: in out float;
    X2: in float) is
  begin
    X1:= (X1+X2)/2.0;
  end MEAN_VALUE;

begin
  MEAN_VALUE(N1, N2);
  put(N1);
  new_line;
end EXAMPLE;

```

SUMMARY

- 1) Scope
 - a) Ada scope rules
 - b) C scope rules
- 2) Parameter passing
 - a) Ada parameter modes
 - b) Parameter passing mechanisms