This construct provides a secure way of programming mutual exclusion, because:

- a shared variable V is declared such that it should be accessed in a CR tagged with the name V - the compiler can flag as an error any attempt to access it outside the CRs;
- all CRs tagged with the same variable name V are executed under mutual exclusion, but statements in CRs tagged with distinct variables can be executed concurrently,
- in effect, the **wait** and **signal** operations which would be required to protect a CS when using semaphores are automatically generated by the compiler, so that they cannot be overlooked.

An Example

For the ornamental gardens problem, we can easily have the following solution:

```
PROGRAM GARDENS; VAR
   count: SHARED integer;
PROCESS Turnstile1;
VAR loop:integer;
BEGIN
   FOR loop:=1 To 20 DO
      REGION count DO
         count:=count+1
END;
PROCESS Turnstile2;
VAR loop:integer;
BEGIN
   FOR loop:=1 To 20 DO
      REGION count DO
         count:=count+1
END;
BEGIN (* main program*)
  REGION count DO
       count:=0;
  COBEGIN Turnstile1; Turnstile2 COEND
END.
```

5.3 Conditional critical regions

CRs provide a more structured and securer way of implementing mutual exclusion than semaphores. However, they are not expressive enough to be as widely applicable as semaphores: CRs are not capable of simulating semaphores. They cannot solve the condition synchronization problem. Therefore, *conditional Critical regions* (CCRs) are introduced to meet such requirements.

Notation and semantics for CRs