2. All the declarations except for the identifiers appearing in the **export** list, are only in the scope within the monitor.
3. Only the names of procedures in the **export** list can be called by a process statement of the form:

   ```
   monitor_identifier.export_procedure_identifier[actual parameters]
   ```

4. The monitor body (**begin** statement-sequence), which is optional (i.e. not all monitor need a body), is executed immediately when the program is initiated, to give initial values to the monitor variables. It is just executed once during the program execution.
5. The compiler guarantees that access to the code within a monitor is done under mutual exclusion. A process that tries to execute a monitor procedure when there is already a process executing one of the procedures in the *same* monitor becomes blocked on a what is called a monitor **boundary queue**. In general, several processes may be blocked on this queue by the time an occupying process completes its monitor procedure call. Mutual exclusion is then passed to *one* of the blocked processes. In Pascal-FC, monitor boundary queues are defined to be FIFO.

5.4.3 Mutual exclusion with monitors

As an example of application of monitors to mutual exclusion, we consider the Ornamental Gardens problem with monitor. A general ornamental gardens problem is the case that we have a number of turnstiles (rather than only two) concurrent updating the number *count* of people in the garden. The solution of the problem is now very easy: we define a monitor to control the updating of *count* under mutual exclusion.

```
PROGRAM
GARDMON; (*Ornament gardens - monitor version*) CONST
  max=10;  (*number of turnstiles*)
MONITOR Tally;
  EXPORT
    inc, print;   (*export list*)
 VAR
    count: integer; (*global variable*)
 PROCEDURE inc;
  BEGIN
    count:=count+1
  END; (*inc*)
  PROCEDURE print;
  BEGIN
    writeln(count)
  END;  (*print*)
  BEGIN (*body of Tally*)
    count:=0
  END;   (*Tally*)
```

```
PROCESS TYPE  turnstiletype;
VAR
  loop:integer;
BEGIN
  for loop:=1 TO 20 DO
    Tally.inc     (*call monitor procedure inc*)
END;    (*turnstiletype*)
VAR
   turnstile: ARRAY[1..max] OF turnstiletype;
   procloop:integer;
BEGIN  (*main*)
  COBEGIN
     FOR procloop:=1 TO max DO
         turnstile[procloop]
  COEND;
  Tally.print
END.
```

The key feature of the program:

1. The shared data structure, such as *count*, is declared in the monitor; the code in the monitor body is executed to give initial values of the monitor variables before processes begin to call the monitor.
2. The monitor procedures *inc* and *print* sit passively until called from a process.
3. If a process wishes to increment *count*, all it needs to do is to call the monitor procedure *inc*.
4. The monitor data structure is not directly visible from outside the monitor; it can only be accessed by executing one of the 'official' operations implemented by the exported procedures. For example, it is necessary to call the monitor procedure *print* from the main program in order to view the final result, even though the monitor *Tally*'s mutual exclusion is not required any more after the completion of concurrent phase of execution.
5. The mutual exclusive access to the shared variable *count* is enforced automatically by the complier when it generates code for a call to an exported monitor procedure: it is not possible to access the data except under mutual exclusion.
6. In comparison with semaphores, monitors are more structured:
   - easier to understand: all the code that manipulates a given data structure must be located in one place;
   - easier to modify: a change in a monitor does not lead to change in other parts;
   - safer to use: a monitor can be written by a senior person with unimpeachable competence and trustworthiness; then the users of the monitor need only call a procedure.