An Experiment with the "record class" as suggested by C.A.R.Hoare.

The primary purpose of this experiment has been to try out the usefulness of the concept of the "record class" as suggested by C.A.R.Hoare. I have selected the algorithm to find the shortest route from one town to another via a map if only the direct town to town connections of this map are given.

Seven years ago I coded this in machine code and at that time it was one of the toughest programming jobs I had ever done. To say that, compared with this early effort, Hoare's record class allows a tremendous simplification in the formulation of the algorithm is of course a next to empty statement. But I invite anybody who is interested in the adequateness of the "record class" as a tool and has grasped the algorithm, to rewrite this program in ALGOL 60 as it stands. Let him embellish his program as much as he likes in the name of his standard of clarity and elegance (for so did I with mine!) and let him then compare them. I am so much convinced that the comparison will turn out in favour of the program making use of the record classes, that I am beginning to ask myself whether I have selected the one and only problem to the needs of which the record classes are really tailored.....

The secondary purpose of this experiment has been to show what beauty can be achieved (and at what price) if one refuses to make use of goto statements. In this example I have used the simple while clause without controlled variables (To make legal ALGOL 60 of it, one could insert "for dummy:= 0" in front of every occurrence of the character "while".) The preceding version of this program contains nine labels and nine goto statements and was accordingly messy.

Now some explication. The linking by the field "fol" serves a dual purpose. It connects towns in a single chain; from "start" to "last placed" it contains the towns for which the path of minimum length from "start" has been established and it does so in the order of increasing path length (which is also the order in which the towns are attached to this part of the chain in time), from "last placed" to "end" it contains, in irrelevant order, all towns with at least one direct connection to a town in the range from and including "start" up till and including "last placed". For all towns in this chain we keep track of the minimum distance from "start" found thus far and the field "previous" indicates the town, via which this minimal path finally leads to it. For each town, added to the first part of the chain, we make the second part of the chain up to date by inspecting all the roads leading out from it; after that we select from the second part of the chain the town with minimum distance found thus far and this is the next town to be transferred to the first part. As soon as "finish" has been transferred to the first part of the chain, the problem of the minimum path from "start" to "finish" has been solved.

The topology of the map is given as follows. The roads leading from a town are arbitrarily ordered, the source town contains a reference to the first road leading out of it, each road contains a reference to the next road, leading out of the same source town, if such a road is present, otherwise the field of such a road reference will contain "null". (Roads are on this map directed connections, towns have towns without roads leading out of it, the dead alley.)

The algorithm may end in two ways: either it finds the connection (last placed = finish) or it decides that the map does not provide for a connection from "start" to "finish". This is decided to be the case when the second part of the chain (from and excluding "last placed" up to and including "end") is empty at the moment, that a town from the second part has to be transferred to the first part; it cases this outcome by "last placed = null".

Now some comments on the structure of the program. The outermost while clause ("while last placed ≠ finish and last placed ≠ null") controls the repetition of the statement in which

1) the second part of the chain is made up to date as result of a newly placed town (i.e. just transferred to the first part of the chain);
2) from the updated second part a new town is selected and transferred to the first part (provided the second part is not empty, for then the algorithm ends with "last placed = null")

The while clause ("while trial ≠ null") controls the repetition of the statement in which the updating consequences of the road "trial", emanating from "last placed" are performed and the next road emanating is made the next trial.

The while clause "while scanned ≠ goal do" investigates whether goal = destination(trial) (i.e. the town, to which "trial" leads) occurs already in the chain; if not, it is quickly added, so that the next time it does belong to the chain and the while clause is terminated. This (and the preliminary distance "+ infinity") is one of the extra prices to be paid for the absence of goto statements. It first sight it is a trick, but after some consideration it becomes such a neat one, so perfectly sound, that it may develop into a method.

The conditional resetting of the distance is by necessity non-active if "goal" was already in the first part of the chain, on account of "+ infinity" active if "goal" has just been added to the chain.

When the second part of the chain has been updated, the new town must be selected. If the second part of the chain is non-empty, "premin" is made to point to the processor of the selected town. This is done in the while clause "while prescanned ≠ end do". Then the selected town is placed at the beginning of the second part (unless it is already there) and finally last placed is moved over one town.

Note. I am aware that the algorithm could be speeded up at the price of more fields in the town records.

```
record class town;
    begin reference first out(road), fol, previous(town); integer distance end;
record class road;
    begin reference destination(town), next out(road); integer length end;

reference start, finish, scanned, prescanned, last placed, goal, premin, end(town),
          trial(road);

"Now the data must be given. Of the records "town" the field "first out" must be
defined (may be = null), the fields of the records "road" must all be defined.
"start" and "finish" (both ≠ null) must be given."

end:= last placed:= start; fol(start):= previous(start):= null;
distance(start):= 0;
while last placed ≠ finish and last placed ≠ null do
begin trial:= first out(last placed);
      while trial ≠ null do
      begin goal:= destination(trial); scanned:= start;
            while scanned ≠ goal do
            begin if scanned = end then
                  begin fol(end):= goal; end:= goal; fol(goal):= null;
                        distance(goal):= + infinity
                  end;
                  scanned:= fol(scanned)
            end;
            if distance(last placed) + length(trial) < distance(goal) then
            begin distance(goal):= distance(last placed) + length(trial);
                  previous(goal):= last placed
            end;
            trial:= next out(trial)
      end;

      if last placed = end then
              last placed:= null
                         else
      begin premin:= last placed; prescanned:= fol(last placed);
            while prescanned ≠ end do
            begin if distance(fol(prescanned)) < distance(fol(premin)) then
                                                 premin:= prescanned;
                  prescanned:= fol(prescanned)
            end;
            if premin ≠ last placed then
            begin scanned:= fol(premin);
                  fol(premin):= fol(scanned);
                  fol(scanned):= fol(last placed);
                  fol(last placed):= scanned
            end;
            last placed:= fol(last placed)
      end
```