```
            GOTO ENDSORT;
         END;
      END;
   ENDSORT: END SORT;
```

There are too many other errors to discuss in detail. An isolated border cell with value `'1'B` will cause grief, as will a non-square maze. Even for a maze as trivial as

```
000
111
000
```

the correct path is never found; when we ran it, after an indeterminate amount of poking around outside the maze the program reported that the middle cell on the left border forms a "path." When subscript range checking is turned on, the program aborts.

It is an enlightening exercise to patch the maze program, providing just enough corrections to permit it to handle reasonable inputs. But patching only serves to emphasize the shortcomings of this organization. After a brief attempt, most readers will agree that the best cure is not revision but a total rewrite.

Don't patch bad code — rewrite it.

The maze program is big enough that it is well worth while planning its structure carefully. One of the better ways of doing this is what is often called "top-down design." In a top-down design, we start with a very general pseudo-code statement of the program like

```
solve mazes
```

and then elaborate this statement in stages, filling in details until we ultimately reach executable code. Not only does this help to keep the structure fairly well organized, and avoid getting bogged down in coding too early, but it also means that we can back up and alter bad decisions without losing too much investment.

How do we "solve mazes"? The loop

```
WHILE (there's a maze)
   solve it
```

breaks the job into two clean pieces — checking whether a maze exists, and processing it. Usually the easiest way to find out whether there is a maze is to try to read it in, using a separate input procedure. This is a convenient organization, as we shall see here and again in Chapter 5.

Refining further,

```
WHILE (READMAZE() = YES)
   IF (there's a path)
      print it
   ELSE
      print 'no path'
```