1. Introduction. This program takes an integer n as input and prints "Hello world" n times. (There is no Pascal code in this section.)

2. We give part of the program here, and it will continue later.

```
⟨ Compiler directives 4 ⟩
program HELLO(input, output);
  var ⟨ global variables 3 ⟩
```

3. What global variables do we need? For one thing, we need the n.

⟨ global variables 3 ⟩ ≡

```
n: integer;
```

See also sections 5 and 9.

This code is used in section 2.

4. For *integer* variables to be treated as 32-bit by the Pascal compiler, on FPC we need a special compiler directive.

⟨ Compiler directives 4 ⟩ ≡

```
@{@&$mode iso@}
```

This code is used in section 2.

5. The string pool and file I/O. The WEB feature of string pools was designed at a time when Pascal compilers did not have good support for strings. Now it may be no longer necessary, but to illustrate the feature we will maintain a string pool.

More specifically, we will maintain a large array of characters, named *str*. All characters of all strings from the string pool go into this array: the nth string occupies the positions from *str_start*$[n]$ to *str_start*$[n+1]-1$ (inclusive) in this array, where *str_start* is an auxiliary array of integers. Also, the number of strings currently in the string pool is stored in an integer variable called *str_count*.

By convention, the first 256 strings are the one-character (one-byte) strings. For this program we don't need too many additional strings. In fact we need just a few strings, but we'll support 10 strings with a total of 1000 characters.

```
define max_strings = 256 + 10
define max_total_string_length = 1000
```

```
⟨ global variables 3 ⟩ +≡
str: array [0 .. max_total_string_length − 1] of char;
str_start: array [0 .. max_strings − 1] of integer;
str_count: integer;
```

6. To use this string pool, we have a procedure that reads out characters from it one-by-one. Specfically, *print*(k) prints the kth string, and *println* and *printnl* are convenience macros.

```
define println(#) ≡
        begin print(#); writeln;
        end
define printnl(#) ≡
        begin writeln; print(#);
        end
procedure print(n : integer);
  var i: integer;
  begin @{writeln('For␣', n, '␣will␣print␣characters␣from␣', str_start[n], '␣to␣', str_start[n + 1] − 1);
    @}
    for i ← str_start[n] to str_start[n + 1] − 1 do
      begin write(str[i]);
      end;
  end;
```

7. We'll have a procedure to populate this array by reading from the pool file, but unfortunately that means we need to figure out file input. How this is done depends on the Pascal compiler. In FPC, a file of characters can be declared as a variable of type *TextFile*, initialized with *Assign* and *Reset*, then read with *read*.

```
procedure initialize_str_array;
  var pool_file: TextFile; x, y: char;   { for the first two digits on each line }
    length: integer; i: integer;
  begin str_count ← 0; str_start[0] ← 0;
  for i ← 0 to 255 do
    begin str[i] ← chr(i); str_start[i + 1] ← str_start[i] + 1; str_count ← str_count + 1;
    end;
  Assign(pool_file, 'hello.pool'); Reset(pool_file);
  while ¬eof(pool_file) do
    begin read(pool_file, x, y);
    if x = '*' then ⟨ check pool checksum 8 ⟩
    else begin length ← 10 * (ord(x) − "0") + ord(y) − "0";
      str_start[str_count + 1] ← str_start[str_count] + length;
      for i ← str_start[str_count] to str_start[str_count + 1] − 1 do
        begin read(pool_file, str[i]);
        end;
      readln(pool_file); str_count ← str_count + 1;
      end
    end;
  end;
```

8. To ensure that the pool file hasn't been modified since tangle was run, we can use the @$ (= @$ = at-sign, dollar-sign) feature. We can reuse (abuse?) the y and *length* variables for reading characters and maintaining the checksum read from the file.

```
⟨ check pool checksum 8 ⟩ ≡
  begin length ← ord(y) − "0";
  while ¬eof(pool_file) do
    begin read(pool_file, y);
    if ("0" ≤ ord(y)) ∧ (ord(y) ≤ "9") then length ← length * 10 + (ord(y) − "0");
    end;
  if length ≠ @$ then
    begin writeln('Corrupted␣pool␣file:␣got␣length:␣', length : 1,
        ';␣rerun␣tangle␣and␣recompile.'); Halt(1);
    end
  end
```

This code is used in section 7.

9. Main program. Apart from n, we also need an i to loop over.

```
⟨ global variables 3 ⟩ +≡
i: integer;
```

10. Here finally is the "main" block of the program.

```
begin initialize_str_array; print("How␣many␣times␣should␣I␣say␣hello?␣"); read(n);
printnl("OK,␣here␣are␣your␣"); write(n : 1); println("␣hellos:␣");
for i ← 1 to n do
  begin println("Hello,␣world!");
  end;
print("There,␣said␣hello␣"); write(n : 1); println("␣times.");
end.
```

11. Index. If you're reading the woven output, you'll see the index here.

⟨ Compiler directives 4 ⟩ Used in section 2.
⟨ check pool checksum 8 ⟩ Used in section 7.
⟨ global variables 3, 5, 9 ⟩ Used in section 2.

Hello