Report

An essay on programming

Author(s):
Wirth, Niklaus

Publication Date:
1999

Permanent Link:

https://doi.org/10.3929/ethz-a-006653169 →

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Computersysteme

Niklaus Wirth

An Essay on Programming

March 1999

An Essay on Programming

Niklaus Wirth, 31. 3. 1999

Many times in my career in programming, when confronted with a certain problem, I found a program supposed to accomplish the very task. Typically, however, it turned out to be roughly, but not exactly what I needed. So I was forced to study the program in order to incorporate the supposedly minor modifications or additions. This then became a rather depressing assignment, because programs are not as well designed, and certainly not meant to be scrutinized, as I used to teach and preach. The attempt characteristically failed as I became disgusted with the result, because the program had been ill suited to accommodate my modifications and became even more enigmatic through them.

As a result, I have learnt to abandon such attempts of adaptation fairly quickly, and to start the design of a new program according to my own ideas and standards. It is more fun that way! In the following, I report on my most recent experience following the pattern described here. The program is the *AsciiCoder*, well–known in Oberon circles. It serves to transform texts, that is, streams of 8–bit bytes, into streams of 6–bit bytes, which can be transmitted safely over any data link on the world, because the 6–bit bytes represent printable charcaters only, and not control characters which might be eliminated or interpreted somewhere on the way. We will here concentrate our attention to the core problem, that of converting a stream into another stream, and we ignore all the rest like setting up commands and interpreting parameters.

We will call the translation of an 8–bit stream into a 6–bit stream *encoding*, and the inverse *decoding*. Both procedures follow the pattern of repeating the reading of an input character (byte), and writing an output character (byte) composed of a part of the input byte and a remainder of a previously read byte. The following two procedures are taken from the mentioned module. We assume that the reader is somewhat familiar with programming in Oberon.

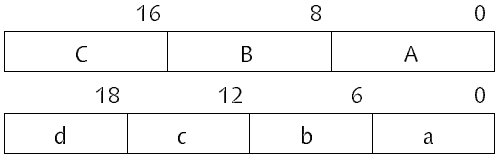
```
CONST Base = 48; StopBase = 35;

PROCEDURE encode(from, to: Texts.Text; pos: LONGINT);
   VAR byte, rest, div, factor: INTEGER;
      ch: CHAR;
      R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); Texts.Read(R, ch); byte := ORD(ch);
   rest := 0; div := 64; factor  := 1;
   WHILE ~R.eot DO
      Texts.Write(W, CHR(Base + rest + (byte MOD div) * factor)); rest := byte DIV div;
      IF div = 4 THEN
         Texts.Write(W, CHR(Base + rest)); rest := 0; div := 64; factor  := 1
      ELSE factor := factor * 4; div := div DIV 4
      END;
      Texts.Read(R, ch); byte := ORD(ch)
   END ;
   IF div = 64 THEN Texts.Write(W, CHR(StopBase))
   ELSIF div = 16 THEN Texts.Write(W, CHR(Base + rest)); Texts.Write(W, CHR(StopBase + 1))
   ELSIF div = 4 THEN Texts.Write(W, CHR(Base + rest)); Texts.Write(W, CHR(StopBase + 2))
   END ;
   Texts.WriteLn(W); Texts.Append(to, W.buf)
END encode;

PROCEDURE decode(from, to: Texts.Text; pos: LONGINT);
   VAR rest, div, factor, byte: INTEGER;
      ch: CHAR;
      R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); Texts.Read(R, ch); factor := 1; div := 256;
   WHILE ~R.eot & (ch >= CHR(Base)) & (ch < CHR(Base + 64)) DO
      byte := ORD(ch) – Base;
      IF factor # 1 THEN
         Texts.Write(W, CHR(rest + (byte MOD div) * factor));
```

```
        rest := byte DIV div; div := div * 4; factor := factor DIV 4
      ELSE rest := byte; div := 4; factor := 64
      END ;
      Texts.Read(R, ch)
    END ;
    byte := ORD(ch) - StopBase; Texts.Append(to, W.buf)
  END decode;
```

When encoding, incoming groups of three 8–bit bytes are transformed into groups of four 6–bit bytes. (Note that the output bytes also consist of 8 bits, representing the 6–bit value plus an offset of 32). When decoding, groups of four 6–bit bytes are transformed into groups of three 8–bit bytes. This is shown in the following figure, and the results are specified below:




```
a = A MOD 64                          A = b MOD 4 * 64 + a
b = B MOD 16 * 4 + A DIV 64           B = c MOD 16 * 16 + b DIV 4
c = C MOD 4 * 16 + B DIV 16           C = d * 4 + c DIV 16
d = C DIV 4
```

We know that multiplication by 2^n is equivalent to shifting left n positions, division by 2^n to shifting right n positions, and taking the modulus 2^n to masking off all but the last n bits. We leave it open whether the use of multiplication and division instead of explicit shifting and masking is promoting clarity of exposition. But surely it is detrimental to the efficiency of a program. Therefore, decent compilers recognize the cases where multiplication or division can be implemented by fast shifting. But no compiler can do this when the multiplier or divisor is a variable rather than a constant. And this is the case in the given program.

As conscientious programmers we attempt to remove this deficiency. We recognize that only 3 (or 4) cases occur, with values *factor* = 64, 16, 4, 1 and *div* = 4, 16, 64, 256 (for procedure *decode*). As old hands in the profession we quickly insert a case statement, or better, since there are so few cases, a cascaded if statement. The result is shown below, where the two variables *div* and *factor* are replaced by a single $k = 0, 1, 2, 3$.

```
PROCEDURE encode(from, to: Texts.Text; pos: LONGINT);
  VAR byte, rest, k: INTEGER;
    ch: CHAR;
    R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); k := 0; rest := 0;
  Texts.Read(R, ch); byte := ORD(ch);
  WHILE ~R.eot DO
    IF k = 0 THEN
      Texts.Write(W, CHR(Base + rest + byte MOD 64)); rest := byte DIV 64; k := 1
    ELSIF k = 1 THEN
      Texts.Write(W, CHR(Base + rest + (byte MOD 16) * 4)); rest := byte DIV 16; k := 2
    ELSE Texts.Write(W, CHR(Base + rest + (byte MOD 4) * 16)); rest := byte DIV 4; k := 0;
      Texts.Write(W, CHR(Base + rest)); rest := 0
    END;
    Texts.Read(R, ch); byte := ORD(ch)
  END ;
  IF k = 0 THEN Texts.Write(W, CHR(StopBase))
  ELSIF k = 1 THEN Texts.Write(W, CHR(Base + rest)); Texts.Write(W, CHR(StopBase + 1))
  ELSIF k = 2 THEN Texts.Write(W, CHR(Base + rest)); Texts.Write(W, CHR(StopBase + 2))
  END ;
  Texts.WriteLn(W); Texts.Append(to, W.buf)
END encode;
```

```
PROCEDURE decode(from, to: Texts.Text; pos: LONGINT);
   VAR rest, k, byte: INTEGER;
      ch: CHAR;
      R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); Texts.Read(R, ch); k := 0;
   WHILE ~R.eot & (ch >= CHR(Base)) & (ch < CHR(Base + 64)) DO
      byte := ORD(ch) - Base;
      IF k = 0 THEN rest := byte; k := 1
      ELSIF k = 1 THEN Texts.Write(W, CHR(byte MOD 4 * 64 + rest)); rest := byte DIV 4; k := 2
      ELSIF k = 2 THEN Texts.Write(W, CHR(byte MOD 16 * 16 + rest)); rest := byte DIV 16; k := 3
      ELSE Texts.Write(W, CHR(byte MOD 64 * 4 + rest)); rest := byte DIV 64; k := 0
      END ;
      Texts.Read(R, ch)
   END ;
   byte := ORD(ch) - StopBase; Texts.Append(to, W.buf)
END decode;
```

There is little argument about the improved clarity of the new version. The cases are clearcut and easily verifyable. Nevertheless, the fact that the program has become longer – case separation always lengthens programs – is unpleasant and bothersome. We sense that there must be a better solution.

The idea to replace variable factors and divisors by constants through introducing separate specification of cases has mislead us. The better idea is found in collecting groups of three (four) bytes in a 24–bit buffer, 24 being the least common multiple of 6 and 8. Case distinction has vanished. Only a single test is required to determine when the buffer is full and must be flushed. An additional benefit is the complete symmetry between encoding and decoding. This leads to the following version, where *ASH(x, n)* denotes shifting x by n positions:

```
PROCEDURE encode(from, to: Texts.Text; pos: LONGINT);
   VAR buf: LONGINT; n: INTEGER; ch: CHAR;
      R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); Texts.Read(R, ch); n := 0; buf := 0;
   WHILE ~R.eot DO
      buf := buf DIV 100H + ORD(ch) * 10000H; INC(n);
      IF n = 3 THEN  (*buffer full*) n := 4;
         REPEAT Texts.Write(W, CHR(buf MOD 40H + 20H)); buf := buf DIV 40H; DEC(n) UNTIL n = 0
      END ;
      Texts.Read(R, ch)
   END ;
   IF n > 0 THEN
      buf := ASH(buf, (n-3)*8);
      WHILE n >= 0 DO Texts.Write(W, CHR(buf MOD 40H + 20H)); buf := buf DIV 40H; DEC(n) END
   END ;
   Texts.Append(to, W.buf)
END encode;

PROCEDURE decode(from, to: Texts.Text; pos: LONGINT);
   VAR buf: LONGINT; n: INTEGER; ch: CHAR;
      R: Texts.Reader;
BEGIN Texts.OpenReader(R, from, pos); Texts.Read(R, ch); n := 0; buf := 0;
   WHILE ~R.eot DO
      buf := buf DIV 40H + (ORD(ch) - 20H) * 40000H; INC(n);
      IF n = 4 THEN (*buffer full*) n := 3;
         REPEAT Texts.Write(W, CHR(buf MOD 100H)); buf := buf DIV 100H; DEC(n) UNTIL n = 0
      END ;
      Texts.Read(R, ch)
   END ;
   IF n > 0 THEN
      buf := ASH(buf, (n-4)*6);
      WHILE n >= 0 DO Texts.Write(W, CHR(buf MOD 100H)); buf := buf DIV 100H; DEC(n) END
   END ;
   Texts.Append(to, W.buf)
END decode;
```

What is the point of this essay? Is it an orgy in nitpicking? A display of pedantry or some other sin? No; nitpicking and pedantry both have a negative connotation. A positive description could be, for example, care for details, or, urge for perfection. In professional programming, both are not a despicable luxury, but a simple necessity.

Surely, mentioning efficiency as the goal would only earn a sympathetic smile in these times of superfast and superscalar processors. After all, who would care whether a translation takes 100 ns or 100 ms! (The case of 100 s vs. 100 hours would be perceived differently, though). As teachers, however, we recognize another value in programming: it is in essence the construction of abstractions, the engineering of (abstract) machines. In addition to being efficient and correct, a good design is based on good style carried through to perfection. And if the economic pressures of industrial realities do no longer permit the "luxury" of perfect design, we as teachers should allow (force?) the students the luxury of carrying a design through to perfection, neither for economic award nor for a good grade, but for the simple experience of it. Good style is learnt on hand of simple, unsophisticated examples, like the one presented here.