Programmeren – Blok A

http://www.win.tue.nl/~wstomv/edu/2ip05/

College 4

Tom Verhoeff

Technische Universiteit Eindhoven
Faculteit Wiskunde en Informatica
Software Engineering & Technology

Opmerkingen aan T.Verhoeff@TUE.NL

Persistente opslag van waarden

Waarden van variabelen gaan verloren zodra programma stopt.

Waarden **persistent** (blijvend) opslaan gaat in **bestanden** (files).

Pascal programma's manipuleren bestanden via **file** variabelen.

Zo'n file variabele is een **communicatie-poort** naar een bestand.

We beperken ons hier tot **tekstbestanden**.

Structuur van tekstbestand volgens Pascal

In Pascal bestaat een tekstbestand uit

- *nul* of *meer* **regels**,
- *afgesloten* met een **end-of-file** markering.

Iedere regel bestaat uit

- *nul* of *meer* **tekens** van type *Char*,
- *afgesloten* met een **end-of-line** markering.

Tekstbestanden: voorbeelden

Lege tekst: ⟨*eof*⟩

Tekst bestaande uit één *lege regel*: ⟨*eoln*⟩ ⟨*eof*⟩

Tekst met vier regels:

```
a b c

1 2

```

a ␣ b ␣ c ⟨*eoln*⟩ ⟨*eoln*⟩ 1 ␣ 2 ␣ ⟨*eoln*⟩ ␣ ⟨*eoln*⟩ ⟨*eof*⟩

Tekstbestanden manipuleren

Tekstbestanden manipuleren gaat via variabelen van type *TextFile*.

Zo'n variabele registreert de **toestand** van de communicatie-poort:

- koppeling aan extern bestand: naam
- richting van communicatie: lezen of schrijven
- positie van lees/schrijfkop: tussen verwerkte en onverwerkte deel

Een tekstbestand kan alleen **sequentieel** worden doorlopen.

TextFile variabele koppelen aan bestand

var
  f: *TextFile*;
...
  {@ f is nog niet gekoppeld, dan wel niet (meer) geopend }
  AssignFile (f, ⟨*stringformule*⟩)
  {@ f is gekoppeld aan extern bestand bij ⟨*stringformule*⟩ }

Voorbeeld:

```
AssignFile ( f, naam + '.txt' )
```


Gekoppelde *TextFile* variabele openen en sluiten

...
  {@ f is gekoppeld aan een bestand dat reeds bestaat }
  Reset (f)
  {@ f is geopend voor lezen met leeskop aan begin }
...
  {@ f is gekoppeld (al dan niet aan een bestaand bestand) }
  Rewrite (f)
  {@ f is **leeg** geopend voor schrijven met schrijfkop aan begin }
  { N.B. Als het bestand al bestond, dan is het nu leeg! }
...
  {@ f is geopend (voor lezen of schrijven) }
  CloseFile (f)
  {@ externe bestand gekoppeld aan f is netjes afgesloten }

Er is ook *Append* (f) om een bestaande tekst uit te breiden

Standaardvariabelen *input* en *output*

input en *output* zijn voorgedeclareerde variabelen van type *TextFile*, voorgekoppeld aan de **console**:

input is voorgeopend voor **lezen** (van het toetsenbord)

output is voorgeopend voor **schrijven** (naar het beeldscherm)

```
var { N.B. Dit gebeurt automatisch; niet overnemen! }
  input, output: TextFile;
...
  AssignFile ( input, '' )
; AssignFile ( output, '' )
; Reset ( input )
; Rewrite ( output )
```

Waarden lezen en end-of-lines lezen

. . .
  {@ f is geopend voor lezen, leeskop staat op waarde van type T }
  read (f, ⟨*variabele van type* T⟩)
  {@ variabele = gelezen waarde, leeskop is waarde net gepasseerd }
  { N.B. Indien nodig wordt een **rijtje** tekens **geconverteerd** }
. . .
  {@ f is geopend voor lezen, leeskop staat niet op het einde }
  readln (f)
  {@ leeskop staat net voorbij 'eerstvolgende' end-of-line }
  { N.B. Indien nodig wordt een **rijtje** tekens overgeslagen! }

Zonder argument f werken ze op *input*.

Waarden schrijven en end-of-lines schrijven

. . .
  {@ f is geopend voor schrijven }
  write (f, ⟨*formule*⟩)
  {@ waarde van formule is geschreven, schrijfkop staat erachter }
  { N.B. Indien nodig wordt waarde **geconverteerd** naar **rijtje** tekens }
. . .
  {@ f is geopend voor schrijven }
  writeln (f)
  {@ één end-of-line is geschreven, schrijfkop staat erachter }

Zonder argument f werken ze op *output*.

Afkortingen voor opeenvolgende leesopdrachten

- *read* (f, v_0, v_1, v_2, . . .)

  is kort voor

  read (f, v_0) ; *read* (f, v_1) ; *read* (f, v_2) ; . . .

- *readln* (f, v_0, v_1, v_2, . . .)

  is kort voor

  read (f, v_0, v_1, v_2, . . .) ; *readln* (f)

Afkortingen voor opeenvolgende schrijfopdrachten

- *write* (f, e_0, e_1, e_2, . . .)

  is kort voor

  write (f, e_0) ; *write* (f, e_1) ; *write* (f, e_2) ; . . .

- *writeln* (f, e_0, e_1, e_2, . . .)

  is kort voor

  write (f, e_0, e_1, e_2, . . .) ; *writeln* (f)

Eerste standaardpatroon om tekstbestand door te lezen

```
const
  Afsluiter = '.'; { om laatste regel te herkennen }

var
  regel: String; { om regels te doorlopen }
...
  {@ f eindigt op Afsluiter }
  readln ( f, regel )

  while regel ≠ Afsluiter do begin
    ... verwerk regel...
  ; readln ( f, regel )
  end { while }
  {@ regel = Afsluiter, leeskop staat op teksteinde }
```


Vereisten voor eerste leespatroon

- Het tekstbestand is afgesloten met een speciale voorafbekende regel.
- De regellengte laat het toe elke regel geheel in te lezen vóór verwerking.

Einde van bestand en van regel detecteren

- *Boolean* formule voor detectie van End-of-file: *Eof* (*f*)

  Toepasbaar als *f* is geopend (voor lezen of schrijven)

  Resultaat: of lees/schrijfkop op einde van tekstbestand staat

- *Boolean* formule voor detectie van End-of-line: *Eoln* (*f*)

  Toepasbaar als *f* is geopend om te lezen

  Resultaat: of leeskop op einde van regel staat

Zonder argument *f* werken ze op *input*.

Tweede standaardpatroon om tekstbestand door te lezen

```
var
  regel: String; { om regels te doorlopen }
...

  while not Eof ( f ) do begin
    {@ leeskop staat op regelbegin }
    readln ( f, regel )
    ... verwerk regel...
  end { while }
  {@ Eof ( f ), leeskop staat op teksteinde }
```

Programma dat regels telt in input

```
1  program LineCounter;
2    { Tel aantal regels in input }
3
4  var
5    aantalRegels: Integer; { aantal gelezen regels }
6
7  begin
8    aantalRegels := 0
9
10 ; while not Eof do begin { verwerk eerstvolgende regel }
11     readln  (* wat er op die regel staat is niet van belang *)
12   ; aantalRegels := aantalRegels + 1
13   end { while }
14
15 ; writeln ( 'Aantal regels = ', aantalRegels )
16 end.
```

Tik Ctrl-Z (^Z) gevolgd door een return voor end-of-file.

Derde (algemene) standaardpatroon om tekstbestand te lezen

while not *Eof* (*f*) **do begin** { er zit nog wat in *f* }
{ verwerk eerstvolgende regel, leeskop staat op regelbegin }

while not *Eoln* (*f*) **do begin** { er staat nog iets op deze regel }
{ verwerk eerstvolgende teken }
read (*f*, *c*)
; ... reageer op teken *c* ...
end { while }
{@ *Eoln* (*f*), leeskop staat op regeleinde }

; *readln* (*f*)
; ... reageer op regeleinde ...
end { while }
{@ *Eof* (*f*), leeskop staat op teksteinde }

Programma dat regels en tekens telt in input

```
1  program TextCounter;
2    { Tel aantal regels en tekens in input }
3
4  var
5    aantalRegels: Integer; { aantal gelezen regels }
6    aantalTekens: Integer; { aantal gelezen tekens }
7    teken: Char;           { om teken in te lezen }
8
9  begin
10
11   ... (* zie volgende slide *)
12
13 ; writeln ( 'Aantal regels = ', aantalRegels )
14 ; writeln ( 'Aantal tekens = ', aantalTekens )
15 end.
```


Input doorlezen en tellen

```
1    aantalRegels := 0
2  ; aantalTekens := 0
3
4  ; while not Eof do begin { verwerk eerstvolgende regel }
5
6      while not Eoln do begin { verwerk eerstvolgende teken }
7        read ( teken )  (* het teken is verder niet van belang *)
8      ; aantalTekens := aantalTekens + 1
9      end { while }
10
11   ; readln
12   ; aantalRegels := aantalRegels + 1
13   end { while }
```

Te tellen bestandsnaam inlezen

```
1  var
2    naam: String; { naam van te tellen tekstbestand }
3    inFile: TextFile; { file om te tellen }
4    nRegels, nTekens: Integer; { aantal regels cq. tekens in inFile }
5    teken: Char; { om teken in te lezen }
6
7  begin
8    write ( 'Welk tekstbestand tellen? ' )
9  ; readln ( naam )
10 ; AssignFile ( inFile, naam )
11 ; Reset ( inFile )
12
13 ; ... (* zie volgende slide *)
14
15 ; write ( naam, ' bevat ', aantalRegels, ' regels en ' )
16 ; writeln ( aantalTekens, ' tekens' )
17 ; CloseFile ( inFile )
18 end.
```


inFile doorlezen en tellen

```
1    nRegels := 0
2  ; nTekens := 0
3
4  ; while not Eof ( inFile ) do begin
5      { verwerk eerstvolgende regel }
6
7      while not Eoln ( inFile ) do begin
8        { verwerk eerstvolgende teken }
9        read ( inFile, teken )  (* teken verder niet van belang *)
10     ; nTekens := nTekens + 1
11     end { while }
12
13   ; readln ( inFile )
14   ; nRegels := nRegels + 1
15   end { while }
```


Bestandsnamen op command-line gebruiken

```
1  var
2    paramNr: Integer; { om command-line argumenten te doorlopen }
3    inFile: TextFile; { te tellen tekstbestand }
4    nRegels, nTekens: Integer; { aantal gelezen regels cq. tekens }
5    teken: Char; { om teken in te lezen }
6
7  begin
8    for paramNr := 1 to ParamCount do begin
9      AssignFile ( inFile, ParamStr ( paramNr ) )
10   ; Reset ( inFile )
11
12   ; ... (* zie vorige slide *)
13
14   ; writeln ( nRegels : 12, nTekens : 12, '  ', ParamStr ( paramNr ) )
15   ; CloseFile ( inFile )
16   end { for paramNr }
17 end.
```


Software-architectuur

Architectuur beschrijft (o.a.):

- Hoe software is opgedeeld in stukken
- Hoe deze stukken onderling gekoppeld zijn

Enkele eenvoudige architectuur-patronen

1. Lezen ; Rekenen ; Schrijven
2. Lezen & Rekenen ; Schrijven
3. Lezen ; Rekenen & Schrijven
4. Lezen & Rekenen & Schrijven

Minder delen die elk ingewikkelder zijn is vaak lastiger te begrijpen, omdat dan 'alles doorelkaar loopt'.

Architectuur bij verwerken van bestanden

Het programma *TextCounter* volgt patroon 2.

Programma's die een invoerbestand omzetten in een uitvoerbestand leiden veelal tot patroon 4.

Zulke programma's zijn daardoor minder goed te doorzien.

Programma om *input* naar *output* te kopiëren

```
1  program Copy;
2    { Kopieer input naar output }
3
4  var
5    teken: Char; { om teken in te lezen en te schrijven }
6
7  begin
8    while not Eof do begin { verwerk eerstvolgende regel }
9
10     while not Eoln do begin { verwerk eerstvolgende teken }
11       read ( teken )
12     ; write ( teken )
13     end { while }
14
15   ; readln
16   ; writeln
17   end { while }
18 end.
```


Een voorbeeld voor modificeren van tekstbestanden

Een tekstbestand **squash**-en houdt in:

1. niet-spaties kopiëren
2. spaties aan regelbegin kopiëren
3. opeenvolgende spaties tussen woorden vervangen door één spatie
4. spaties aan regeleinde verwijderen
5. lege regels verwijderen

Squash-eisen zijn inconsistent

Wat te doen met een regel die uit **alleen spaties** bestaat?

Die spaties staan zowel aan **begin** als aan **einde**.

De specificatie eist dan zowel **kopiëren** (2) als **verwijderen** (4)!

Toegevoegde eis 6: Verwijder ze, samen met de regel.

Programma voor squashen van tekstbestand (aanhef)

```
1  program Squash;
2    { output is gesquashte kopie van input:
3      1. niet-spaties kopieren
4      2. spaties aan regelbegin kopieren,
5         tenzij regel alleen spaties bevat
6      3. opeenvolgende spaties tussen woorden vervangen
7         door een enkele spatie
8      4. spaties aan regeleinde verwijderen
9      5. lege regels verwijderen
10     6. regels met alleen spaties verwijderen
11   }
```


Programma voor squashen van tekstbestand (ontwerp)

- Kan regel voor regel, want regels zijn onafhankelijk van elkaar
- Iedere niet-spatie moet gekopieerd worden (1)
- Wat met spaties moet gebeuren hangt af van hun plaats (2, 3, 4), ‘spaar’ ze daarom op tot volgende niet-spatie of regeleinde
- Het volstaat om de ‘gespaarde’ spaties te tellen
- Houd bij of ‘gespaarde’ spaties aan regelbegin staan of niet

Programma voor squashen van tekstbestand (implementatie)

```
1  var
2    teken: Char;  { om teken in te lezen }
3    nSpaties: Integer;  { # opeenvolgende gelezen onverwerkte spaties }
4    aanBegin: Boolean;  { of de onverwerkte spaties aan regelbegin staan }
5
6  begin
7
8    while not Eof do begin
9      { verwerk de eerstvolgende regel }
10     ... (* zie volgende slide *)
11   end { while }
12
13 end. { Squash }
```

Een regel squashen (implementatie)

```
1     nSpaties := 0  { nog geen spaties 'gespaard' }
2   ; aanBegin := True { gespaarde spatie aan regelbegin }
3
4   ; while not Eoln do begin
5       read ( teken )
6       { verwerk teken }
7     ; ... (* zie volgende slide *)
8     end { while }
9
10  ; readln
11  ; if aanBegin then { lege regel (5) of alleen spaties (6) }
12      { verwijder regel, dus geen writeln }
13    else { eventuele spaties aan regeleinde verwijderen (4) }
14      writeln
```


Een teken squashen (implementatie)

```
1     ; if teken = ' ' then { spatie: 'sparen' }
2         nSpaties := nSpaties + 1
3       else begin { niet-spatie: kopieren (1) }
4         if nSpaties > 0 then begin { er zijn gespaarde spaties }
5           if aanBegin then { aan regelbegin kopieren (2) }
6             write ( ' ' : nSpaties )
7           else { tussen woorden reduceren tot 1 spatie (3) }
8             write ( ' ' )
9         end { if }
10      ; write ( teken )
11      ; nSpaties := 0
12      ; aanBegin := False
13      end { else }
```


Programma *Squash* samengeperst (voorbeeld van slechte stijl)

```
1  program Squash;
2  var c: char; s: integer; b: boolean;
3  begin
4    while not Eof do begin s:=0; b:=true;
5      while not Eoln do begin read(c);
6        if c=' ' then s:=s+1
7        else begin
8          if s>0 then if b then write(' ':s)
9                      else write(' ');
10         write(c); s:=0; b:=false
11       end
12     end;
13     readln; if not b then writeln
14   end
15 end.
```


ISBN-10 code: case-opdracht, *ord* functie op type *Char*

Een ISBN-10 code $a_1 a_2 \ldots a_{10}$ is valide precies wanneer

$$\left(\sum k : 1 \le k \le 10 : k * a_k\right) \bmod 11 \;=\; 0 \qquad \text{(met 'X' = 10)}$$

```
1     som := 0
2
3   ; for k := 1 to Length ( code ) do begin
4       case code [ k ] of
5         '0' .. '9': waarde := ord ( code [ k ] ) - ord ( '0' )
6       ; 'X', 'x': waarde := 10
7       end { case }
8     ; som := som + k * waarde
9     end { for k }
10
11  ; codeValide := ( som mod 11 = 0 )
```