Pascal uitgediept
Recursie

Herman Post
MSX Computer & Club Magazine nummer 72 - december '94 / januari '95

Scanned, ocr'ed and converted to PDF by MSXHans, 2001

Recursie is deze keer het onderwerp, maar een aantal lezers verzocht om voorbeelden te geven van het gebruik van de compiler directives; het onderwerp van de vorige aflevering.

Fibonacci

Voor de niet wiskundige onder ons, Fibonacci was de bijnaam van Leo-nardo van Pisa, een wiskundige die leefde van ~1180 - ~1250. Hij was een van de voorstanders van het Arabische cijferen met decimaal stelsel, in plaats van het abacus rekenen. De wiskundige rij die naar hem genoemd is, bestaat uit een serie getallen, waarbij ieder getal de som is van de twee dat getal voorafgaande getallen. Beginnen we met l, l wordt de reeks: 1, 1, 2(1+1), 3 (1+2), 5(2+3), 8(3+5), 13(5+8), 21(8+13), 34, 55,...

In de vorige aflevering van 'Pascal uitgediept' heb ik besproken welke compiler directives er in turbo Pascal beschikbaar zijn. Zoals meestal ging ik hierbij erg diep op alles in en nam aan dat alle gebruikte termen direct duidelijk zouden zijn. Dat dit echter niet zo is, werd mij duidelijk door een aantal vragen die ik over deze materie mocht ontvangen. In deze aflevering zal ik dan ook wat voorbeelden geven met de meest gebruikte compiler directives. Daarnaast zal ik proberen duidelijk te maken wat recursie is en voor de gevorderden uitleggen welke onverwachte moeilijkheden er bij recursie kunnen optreden.

I/O error afhandeling

De meest gebruikte compiler directive in Pascal is ongetwijfeld de {$I+/ - } Met deze compiler directive kunt u voorkomen, dat uw programma terugkeert naar DOS op het moment dat u een niet bestaande file probeert te lezen. In het voorbeeldprogramma is te zien dat de controle plaatsvindt bij het reset-ten van de file en niet bij het assignen. Als de functie IORESULT de waarde 0 oplevert, dan is de file aanwezig en is deze te openen. Het statement CLOSE levert geen foutmelding op en hoeft niet binnen de controle te vallen. Ik heb in dit geval alleen op de plaats waar het nodig is, deze controle aangezet, zodat u kunt zien waar een foutmelding zou optreden. U kunt natuurlijk de controle boven in uw programma aanzetten en daarna nooit meer uitzetten.

```
PROGRAM IOCheck;
VAR fil   : TEXT;
    regel : STRING[255];
BEGIN
  ASSIGN(fil,'demo.txt');
  {$i-}
  RESET(fil);
  {$i+}
  IF IORESULT<>0 THEN
    WRITELN('File DEMO.TXT is niet aanwezig!')
  ELSE
    REPEAT
      READLN(fil,regel);
      WRITELN(regel)
    UNTIL EOF(fil);
  CLOSE(fil);
END.
```

Range check

De compiler directive {$R+/ -} is vooral handig tijdens het ontwikkelen van een programma om er voor te zorgen dat ingegeven waarden niet buiten een array vallen. Als dit namelijk wel gebeurt, wordt een aantal andere variabelen aangetast. In het ergste geval kan een programma hierop vastlopen. In het voorbeeldprogramma ziet u een demonstratie van dat overschrijven van niet bedoelde variabelen. De beide lijsten worden met FILLCHAR gevuld met een beginwaarde. Lij st_a wordt helemaal met de waarde 55 en lij st_b met de waarde 44 gevuld. Hierna wordt alleen een deel van lij st_b gevuld met de waarde 66. Omdat er echter buiten het bereik van de array wordt geschreven zal het programma een fout-melding op het scherm zetten. Verwijdert u echter de tussenstaande compiler directive, dan zult u zien dat ook in lij st_a de eerste 51 waarden zijn veranderd in 66 en er dus een onverwachte verandering in uw variabelen heeft plaatsgevonden. In dit geval is het als voorbeeld natuurlijk verwacht en is de overschrijving met wel 51 bytes wel erg fors, maar het gebeurt erg gemakkelijk in een eigen programma dat een teller juist één byte te ver doortelt en er daardoor onduidelijke fouten ontstaan. Plaatst u de range-check compiler directive dan voorkomt u dit soort foutjes, doordat ze erg vlug opvallen.

```
PROGRAM RangeCheck;
VAR lijst_a : ARRAY[0..99] OF BYTE;
    lijst_b : ARRAY[0..99] OF BYTE;
    tel     : INTEGER;
BEGIN
  FILLCHAR(lijst_a,100,55);
  FILLCHAR(lijst_b,100,44);
  {$R+}
  FOR tel:=50 TO 150 DO lijst_b[tel]:=66;
  FOR tel:=0 TO 99 DO
    WRITELN(lijst_a[tel],lijst_b[tel]:10);
END.
```

Absolute code

Vorige keer vertelde ik bij de ({$A+ / - } directive dat hiermee recursieve routines te schrijven zijn. Ik ging er hierbij vanuit, dat iedereen deze term wel zou begrijpen en

ging er daarom niet verder op in. Ten onrechte. Een recursieve routine is een procedure of functie die zichzelf aanroept. Dit is een redelijk ingewikkelde manier van programmeren, maar er zijn vaak erg korte constructies mee te maken. Een beroepsprogrammeur die ik hierover sprak kwam met de mededeling dat recursie eigenlijk alleen geschikt is voor het berekenen van de 'rij van Fibonacci'. Zie kader. Recursie is echter voor veel meer te gebruiken. Het kan worden gebruikt bij het doorlopen van binaire bomen, richtingen zoeken bij een instructie paint, het weghalen van bommen in een spelletje minesweeper, het berekenen van bezier-krommen—zie grafische objecten MCCM 71—het berekenen van fractals en natuurlijk bij het berekenen van de termen in de rij van Fibonacci. In het voorbeeld gebruik ik recursie voor het namaken van de BASIC functie BIN$. Een functie die een decimaal of hexade-cimaal getal omzet naar een binaire notatie zonder voorloopnullen. Jacco merkt in zijn artikel over bezier-krommen terecht op dat recursieve routines vaak trager zijn dan nietrecursieve methoden, maar omdat het gaat om de uitleg wil ik het toch recursief verwerken. Voor mensen die niet zo dol zijn op de programmeersport, heb ik de nietrecursieve versie ook vermeld. De truc van recursie zit hem in de waarde die meegegeven wordt aan de functie. Doordat de functie hierop een bewerking uitvoert, en de veranderde waarde opnieuw naar de functie stuurt ontstaat er een zichzelf aanroepende routine. In de routine zit een eindvoorwaarde verwerkt en daardoor komt de machine niet in een eindeloze lus. De inputvaria-bele mag nooit variabel zijn, de reden vindt u onderin bij de uitleg voor gevorderden. De echt recursieve aanroep zit hier in het stukje B in: =Bin (input
DIV 2). Hier wordt opgevraagd en toegekend in één stap, waardoor het wat onoverzichtelijk wordt. Om het wat duidelijker te maken daarom hier een procedure die hetzelfde binaire getal op het scherm afdrukt. Hierin zijn de gemaakte stappen wat minder groot, en daarmee is de zaak overzichtelijker. De functie is echter veel bruikbaarder in eigen programma's.

```
{$A-}
PROCEDURE PrintBin(input: INTEGER);

BEGIN
  IF input>l THEN  PrintBin(input DIV 2);
  WRITE(input AND 1)
END;
{$A+}
```

Probeert u eens te volgen wat er gebeurt als u deze procedure aanroept met input 2. De eerste keer dat u de procedure binnen komt is input groter dan l, en wordt de procedure opnieuw aangeroepen met '2 DIV 2' (=1). Nu wordt weer gekeken of input groter is dan 1. Dit is niet zo want input=l. Nu wordt de wr ite uitgevoerd die in dit geval een l op het scherm zet. Het einde van de procedure is bereikt en er wordt teruggesprongen naar het punt waar de procedure is aangeroepen. Dit punt ligt in de procedure zelf en daar wordt dan ook weer verdergegaan. Toen we hier echter wegsprongen had input de waarde 2. De opdracht wr ite die er onder staat wordt weer uitgevoerd, en drukt een O af. (2 AND l =0) De procedure roept zichzelf in dit geval maar één keer aan. Als u het getal 256 meegeeft zal het hele proces zich 8 keer afspelen, (meegegeven 256, eerste aanroep 128, tweede aanroep 64, derde aanroep 32, vierde aanroep 16, vijfde aanroep 8, zesde aanroep 4, zevende aanroep 2, achtste aanroep 1) Let u er ook even op dat oneven getallen door de DIV instructie naar

beneden worden afgerond. Als u datgene wat hiervoor
\ staat nog niet goed begrijpt, bestudeer het geheel dan nog eens goed. Recursie is een lastige bezigheid die u niet in een paar minuten onder de knie hebt. Als u bijvoorbeeld dacht dat de functie en de procedure hetzelfde resultaat voor alle getallen oplevert moet u het getal O maar eens invoeren. U zult zien dat de procedure de nul wel weergeeft, maar de functie niet. Zoek uit waarom... Nee, het zit niet in de vergelijking met 0 of 1.

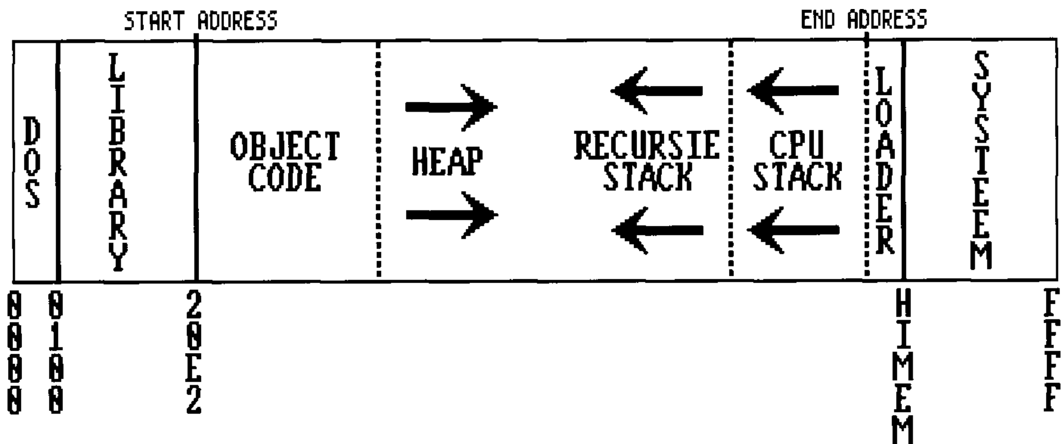
Diepgang

Om de naam van deze rubriek eer aan te doen ga ik nu even echt diep. Hebt u het bovenstaande niet begrepen, dan kunt u hier beter ophouden, dit is voor de echte fanatici. Tijdens het werken met recursieve routines maakt Pascal gebruik van een tweede stack. Dit is nodig omdat de lokale variabelen in de routine niet mogen worden overschreven. Dit is dan ook precies wat u met de compiler directive instelt. Deze tweede stack (recursie-stack) bevind zich 1kB ($400) onder de normale CPU-stack. Als uw recursieve aanroepen een diepte bereiken van ongeveer 300 aanroepen, dan zal de CPU-stack zich over de recursie-stack gaan schrijven en
is vastlopen bijna onvermijdelijk. Er zijn in Pascal, buiten de pointers die in # 66 stonden vermeld, nog drie pointers beschikbaar. Dit zijn HeapPtr, RecurPtr en StackPtr. Door de RecurPtr aan te passen kunt u een grotere stack voor de CPU creëren. De nieuwe plaats wordt berekend met: RecurPtr : =StackPtr -2*Maxdiepte-512, met Maxdiepte dat natuurlijk staat voor de diepte die uw recursieve routine kan bereiken. De extra 512 bytes zijn nodig als marge voor tussenresultaten bij de evaluatie van expressies. De HeapPtr en de Re -curPtr mag u hetzelfde gebruiken als alle andere integers. De StackPtr mag u alleen gebruiken in toekenningen en expressies. In de tekening hieronder kunt u zien hoe het geheugen is ingedeeld tijdens het uitvoeren van een programma. De ruimte die aangegeven staat als LOADER wordt alleen gebruikt als u vanuit het Pascal hoofdmenu met X (execute) een programma opstart. Hier staat dan de code die nodig is om terug te keren naar de Pascal omgeving. Als MemMan is geïnstalleerd, dan staat deze tussen de loader en het systeemgebied in. Let er op dat de Pascal heap een andere is als de MemMan heap. In het spraakgebruik kan hierin vlug verwarring ontstaan. Als Pascal gebruik maakt van zijn RecurPtr of zijn HeapPtr, dan wordt gecontroleerd of deze elkaar overschrijven. Als dit voorkomt, dan treedt er een runtime —255 Heap/stack collision—error op. Dit wordt gecontroleerd door te kijken of HeapPtr kleiner is dan RecurPtr. Als u de RecurPtr zelf aanpast, let er dan goed op dat deze groter is dan de HeapPtr.

De volgorde is wel altijd:

HeapPtr < Recur Ptr < StackPtr.

Het is natuurlijk niet nodig om te vertellen dat u deze pointers niet moet aanpassen als ze in gebruik zijn.




```
PROGRAM Recursie;
TYPE str16=STRING[16];
VAR tel : INTEGER;

      {$A-} {recursieve versie}
FUNCTION Bin(waarde : INTEGER) : STR16;
BEGIN
  BIN:='';
  IF waarde>0 THEN
    Bin:=Bin(waarde DIV 2)+CHR((waarde AND 1)+48)
END;
      {$A+}

      {niet recursieve versie}
FUNCTION BIN2(waarde : INTEGER) : str16;
VAR rij : str16;
BEGIN
  rij:='';
  WHILE waarde>0 DO
  BEGIN
    rij:=CHR((waarde and 1)+48)+rij;
    waarde:=waarde div 2
  END;
  bin2:=rij
END;

BEGIN
  FOR tel:=0 TO 20 DO
    WRITELN(Bin(tel):16,bin2(tel):20)
END.
```