Siegfried Zuhr, Noud van Kruysbergen

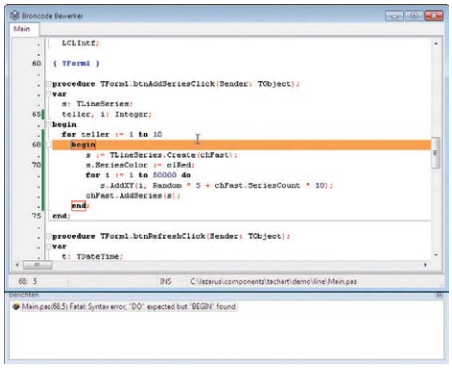
Zoek de vout

Programmeercursus Pascal, deel 2: het debuggen

In het eerste deel van deze cursus in de vorige c't heb je kennis gemaakt met de ontwikkelinterface Lazarus voor de programmeertaal Pascal. De eerste programma's die we daarbij gemaakt hebben waren redelijk overzichtelijk, zodat je daar weinig fouten in zult hebben gemaakt. Maar wat als de code een stuk uitgebreider en complexer wordt en er toch iets fout gaat? Hoe kom je er dan achter waar het precies spaak loopt? Met de ingebouwde debugger kun je snel en systematisch achterhalen waar de schoen wringt.

Debuggen is het opsporen van fouten in een applicatie. Je kunt gerust stellen dat er geen applicaties zijn die geen fouten bevatten. Hoe structureel je ook programmeert, een fout kan er altijd insluipen, variërend van een typfout tot een conceptuele ontwerpfout. Het is de kunst om ze op te sporen en te herstellen.

Alles wat je in de hogere programmeertaal Pascal programmeert moet met de compiler worden omgezet naar een lagere programmeertaal waar een computer wat mee kan. Meestal is dat assembler of machinecode. Daarmee wordt de code uitvoerbaar en kun je er ook een echt programma van maken. De compiler moet dan wel snappen wat je geprogrammeerd hebt. Hij zal in eerste instantie dan ook kijken of er geen taalfouten in de code staan. Tijdens het compileren controleert de compiler de gebruikte taalelementen en de structuur (ook wel syntaxis genoemd) van gereserveerde woorden als `if`, `then`, `record`, `true`, `false` en tekens als de `;` Het eerste wat de compiler dan ook doet is de syntaxis controleren van je broncode. Als daar fouten in zitten, wordt aangegeven waar die fout zit en wat er verkeerd is. Dat voorkomt 'slordig' programmeerwerk. Je wordt immers gedwongen je code netjes te houden om onverwachte effecten te voorkomen.



De compiler kijkt eerst of je de structuur van de programmeertaal goed hebt aangehouden. In dit geval zijn we bij de syntaxis van een for-loop vergeten daar een `do` achter te zetten.

Deze controle kan ook handig zijn als je het overzicht verliest op de `begin` en `end` paren in meerdere geneste `if..then..else` statements of deze bijvoorbeeld vergeet aan het einde van een `case` statement. Pascal gaat daar heel ver in. Je krijgt niet alleen foutmeldingen, maar ook waarschuwingen die je kunnen helpen je code zo zuiver mogelijk te houden. Als we bijvoorbeeld een variabele declareren die we nergens gebruiken, maakt de compiler daar melding van. Dat heeft verder geen gevolgen voor de uitvoer van het programma, maar is altijd zonde. De compiler is daar best slim in: hij ziet het zelfs als je wel een waarde aan de variabele toekent, maar daar verder niets mee doet.

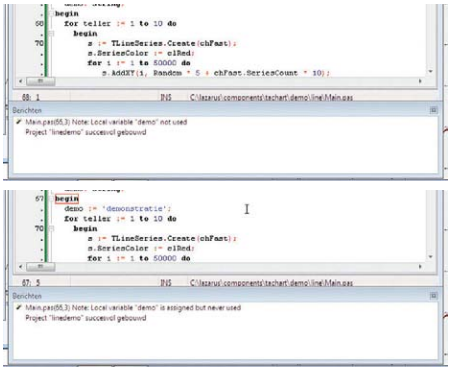
De genoemde gereserveerde woorden zijn een wezenlijk onderdeel van de taal Pascal en kunnen daarom niet voor andere zaken als namen van variabelen en procedures gebruikt worden. Bedenk daarbij dat een variabele eigenlijk niet meer is dan een naam die verwijst naar een bepaalde positie in het geheugen. De waarde die daar wordt opgeslagen is de inhoud van de variabele. Door bewerkingen uit te voeren op (de naam van) de variabele verander je in werkelijkheid dus de inhoud van het bijbehorende geheugenadres.

Typfout en typefout

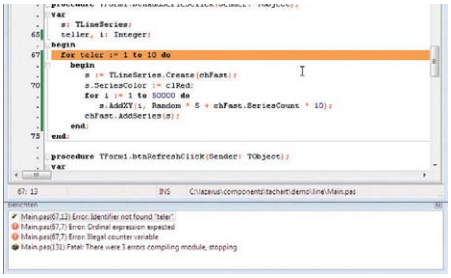
Met dat in het achterhoofd wordt het inzichtelijker wat een typfout voor gevolgen kan hebben. Als je bijvoorbeeld de inhoud van de variabele `teller` wilt hebben om te kijken of een programmaloop al een aantal malen doorlopen is, krijg je met de namen `tellen` en `teler` tijdens het compileren al een foutmelding omdat die namen niet bestaan. Deze andere variabelennamen zijn niet bekend, en verwijzen dus niet naar een specifiek geheugenadres. Als de compiler dat zou toestaan, zouden dat willekeurige adressen worden en loop je het risico dat je hele systeem plat gaat.

Typfouten worden daarom door de compiler al herkend voordat het programma uitvoerbaar wordt. Een andere categorie fouten zijn typefouten. Als de variabele `teller` een getal en dus van het type `Integer` is, is het niet mogelijk daar een tekst aan toe te kennen. Andersom is het niet mogelijk om een getal in de variabele `sText` van het type `string` te stoppen. Maar let op: de tekst '10' is wel toe te kennen aan deze variabele, maar dat heeft een hele andere betekenis dan het getal 10. Ook hier helpt het weer in het achterhoofd te houden dat deze variabelen alleen verwijzers zijn naar bepaalde geheugenadressen. Een integer wordt op een heel andere manier in het geheugen opgeslagen dan variabelen van het type `string` of andere. Het uitlezen van een adres waar een getal in staat als string, of andersom, leidt dan tot rare en onvoorspelbare effecten.

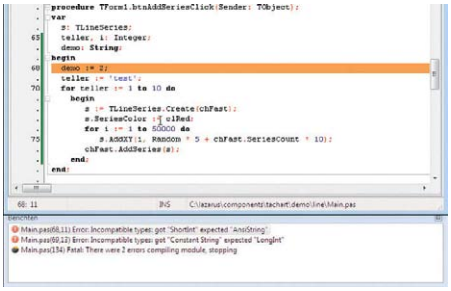
De compiler slaat dan ook alarm als je probeert een getal in een variabele van het type `string` te proppen of het resultaat van een deling probeert op te vangen in een `integer`-variabele.



Als je variabelen wel declareert, maar niet gebruikt, krijg je daar een waarschuwing van de compiler voor. De compiler ziet het zelfs als je wel een waarde aan die variabele toekent, maar daar verder niets mee doet. Dan is het zonde om die regels in je code te hebben staan en daarom kun je ze voor de overzichtelijkheid beter verwijderen.



Bij het maken van een typfout bij de namen van variabelen geeft de compiler een foutmelding. Hier is bij de variabele teller een l weggevallen.



In deze code wordt geprobeerd een string aan een integervariabele toe te kennen en een integer in een stringvariabele te stoppen. De compiler haalt beide fouten er feilloos uit.

Maar hoe goed de compiler van Lazarus ook 'meekijkt' met wat je aan het maken bent, hij kan niet beoordelen of datgene wat je ingetypt hebt ook is wat je eigenlijk bedoelt. Als je bijvoorbeeld een stukje tekst uit een string wilt halen, dan weet de compiler niet of het inderdaad over het stukje gaat dat je wilt hebben. Daarbij beginnen sommige functies en procedures te tellen bij 1, terwijl andere zogenaamd zero-based zijn en bij 0 beginnen. Dan kom je net niet op het gewenste resultaat uit.

Code doorlopen

Als je programmeerwerk geen typ- en typefouten meer bevat, is de volgende stap om te controleren of het programma ook echt doet wat je wilt dat het doet. De IDE (Integrated Development Environment) van Lazarus komt je daarbij te hulp. Daar zijn allerlei hulpmiddelen ingebouwd om het ontwikkelen van applicaties te ondersteunen.

Een heel eenvoudige – maar uiterst doeltreffende – manier om mee te beginnen is de mogelijkheid om de geprogrammeerde code stap voor stap uit te laten voeren. Dat doe je met de opties 'Starten / Step Into' en 'Starten / Step Over' of met de respectievelijke functietoetsen F7 en F8. Als je op een van die twee drukt, begint de compiler met het compileren van je code en stopt dan bij het uitvoeren van de eerste regel. Dan wordt weer overgeschakeld naar de IDE en staat de cursor op de regel die als eerstvolgende uitgevoerd zal gaan worden. Als je dan weer op F7 of F8 drukt, wordt deze regel uitgevoerd. Dit is het moment waarop het verschil tussen F7 ('Step Into') en F8 ('Step Over') duidelijk wordt. Met F8 wordt de regel uitgevoerd waarop de cursor op dat moment staat en gaat de IDE verder met de volgende regel. Het verschil tussen F7 en F8 wordt pas echt duidelijk als je het zelf in de praktijk uitprobeert.

Met F7 wordt gekeken of de regel verwijst naar andere stukken code en bijvoorbeeld een functie of procedure is, en zo ja, dan zal hij daar naar toe springen en daar doorgaan met het uitvoeren van de eerste regel in die functie. Je kunt met F7 dan alle stappen van de hele code doorlopen. Als je op een gegeven moment zeker weet dat een bepaalde functie in ieder geval goed wordt uitgevoerd, hoef je die functie niet iedere keer stapsgewijs te doorlopen, maar kun je hem gewoon laten uitvoeren. Door dan op F8 te drukken wordt de functie uitgevoerd zonder er naartoe te springen en ga je door met de volgende regel in hetzelfde codedeel. Als je stapsgewijs door de code heen wandelt, komt er voor de regel die op dat moment uitgevoerd gaat worden een groen pijltje te staan, terwijl de regel zelf in het grijs gehighlight wordt.

Een aanvulling op de functietoetsen F7 en F8 is toets F4 of het menu-item 'Starten / Start tot cursor'. Je hoeft namelijk niet telkens aan het begin van je programma te beginnen met het doorlopen van de code. Met F4 kun je de code laten uitvoeren tot aan de regel waar je op dat moment op staat. Het gebruik van F4 is bijvoorbeeld handig als je een procedure wilt testen die eenmalig wordt uitgevoerd, bijvoorbeeld na het aanklikken van een knop. Je zet de cursor dan op de beginregel van die procedure en na het drukken op F4 zal de applicatie pas stoppen als je op de bedoelde knop hebt geklikt. Maar ook bij het onderzoeken van een loop kun je F4 gebruiken: door het telkens kiezen van F4 wordt de loop doorlopen zolang de voorwaarde voor if, repeat of while geldig is.

Als je altijd op een bepaalde plek wilt pauzeren met het uitvoeren van de code, kun je een zogeheten 'breakpoint' instellen. Met de knop F5 markeer je de regel waar de cursor op dat moment op staat. Je kunt ook klikken

Bij het per regel debuggen van de code laat een groen pijltje zien welke regel op dat moment uitgevoerd wordt. Je kunt dan kiezen of je de betreffende regel wilt uitvoeren ('Step Over', F8) of precies wilt weten wat er in die regel gebeurt als er een functie of procedure wordt aangeroepen ('Step Into', F7).

Door het instellen van een 'breakpoint' (rode cirkel in de kantlijn) wordt de code uitgevoerd tot dat punt bereikt is en stopt dan (pijltje in rode cirkel).

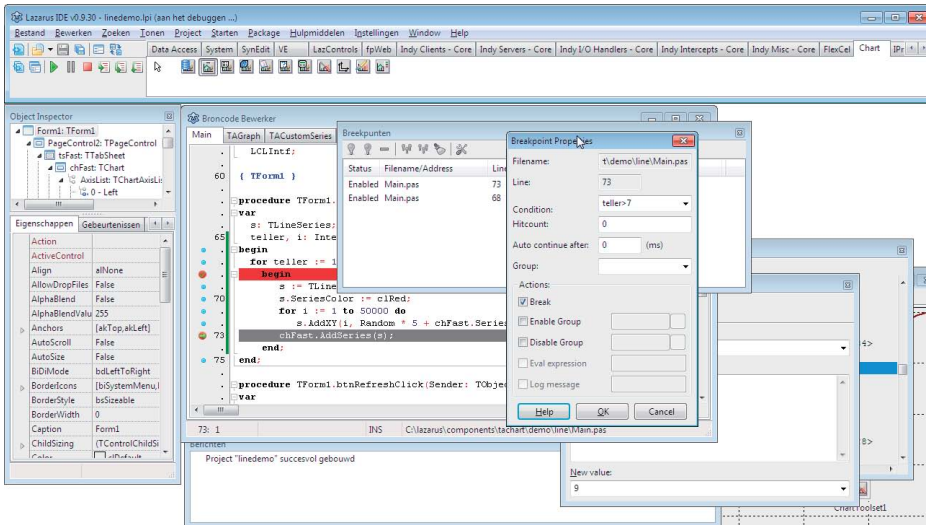
op de linker kantlijn van die regel. Daar verschijnt vervolgens een rode stip die aangeeft dat hier een breakpoint is ingesteld. Als je de applicatie dan start, zal de code doorlopen worden tot aan dit punt en dan overschakelen naar de IDE, waarna je met F7, F8 of F4 verder kunt gaan. Door op de rode punt te klikken, verwijder je het breakpoint weer en zal de uitvoer van het programma daar niet meer stoppen.

Wat is de waarde?

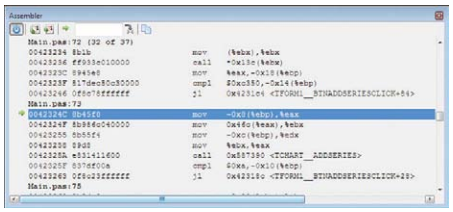
Het is natuurlijk heel leuk dat je kunt volgen hoe de code doorlopen wordt, maar dat brengt je bij het opsporen van fouten nog niet veel verder. Bij het doorlopen van de code opent automatisch een venster met daarin de assemblercode. In dat venster staat in machinetaal wat de volgende acties zijn die door de processor van de computer uitgevoerd gaan worden. Dat is voor gewone stervelingen niet te volgen en daar heb je dus ook niet veel aan. Als je met F7, F8 of F4 op de plek gekomen bent waar je meer over wilt weten, dan kun je de cursor op een variabele zetten waarna de IDE de huidige inhoud van de betreffende variabele in een pop-up laat zien. Dat levert al heel wat meer informatie op, al is dit wel een 'vluchtige' manier om even snel te kijken wat een waarde is. De pop-up verdwijnt weer als je verder gaat.

Je kunt ook 'watches' aanmaken. De IDE heeft de mogelijkheid om een lijst te maken met variabelen waarvan je de inhoud wilt volgen. Dit is de 'watch list'. Die kun je openen op je desktop ('Tonen / Debug schermen / Watches' of met Ctrl+Alt+W). Door op de +-knop te klikken verschijnt een venster met 'Watch Properties', waarbij je bij 'Expression' de variabele 'teller' invoert. Bij het uitvoeren van de code wordt in dit venster dan in realtime de waarde van de variabele teller getoond. Je kunt hier meerdere variabelen toevoegen om het verloop van het programma te kunnen volgen.

Een mogelijkheid om de waarde en het type van een variabele in de gaten te houden krijg je door rechts te klikken op die variabele en in het snelmenu dan 'Debug / Inspect' te selecteren. Als je hier 'Debug / Evaluate/Modify' aanklikt, kun je de waarde van een variabele ook aanpassen. Typ de

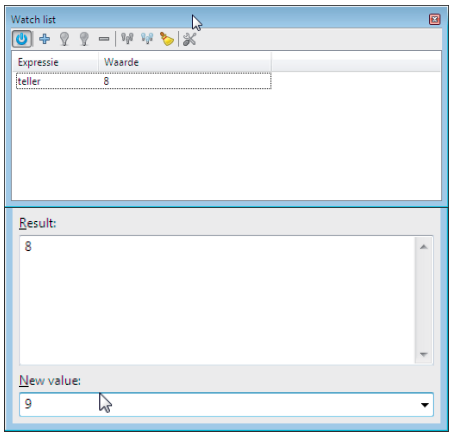


Bij de eigenschappen van de breakpoints kun je aangeven dat er pas gestopt hoeft te worden als aan een bepaalde voorwaarde is voldaan. Hier wordt de for-loop eerst 7 keer doorlopen zonder te pauzeren, pas bij de achtste keer stopt de uitvoer op dit breakpoint.



Bij het doorlopen van de code kun je in het Assembler-venster realtime zien welke machinecode er uitgevoerd gaat worden. Echt informatief is dat echter niet.

Als het programma stopt omdat het bijvoorbeeld een breakpoint tegenkomt, kun je met de muis over de variabelen bewegen om te zien wat hun waarde op dat moment is. Hier blijkt de teller waarde 7 te hebben, en moet de loop dus nog drie keer doorlopen worden.



In de lijst met 'watches' kun je de inhoud van variabelen bijhouden, en in het venster 'Evalueer/bewerk' kun je van een bepaalde variabele niet alleen zijn inhoud bekijken, maar deze ook ter plekke veranderen.

nieuwe waarde in bij 'New value' en klik op 'Modify'. Zo kun je een for-loop bijvoorbeeld veel sneller later doorlopen door de teller de eindwaarde mee te geven. In dit venster heb je ook de mogelijkheid om weer naar het Watch- en Inspect-venster te gaan.

Naast de lijst met watches is er ook een lijst met breakpoints. Via 'Tonen / Debug schermen / breekpunten' of met Ctrl+Alt+B krijg je te zien welke breakpoints er in de gehele applicatie zijn gezet. Dat lijkt op zich weinig spectaculair, maar je hebt hier de mogelijkheid om er bepaalde condities aan te hangen. Als je een breakpoint selecteert en rechtsboven op de tools klikt, kom je bij de 'Breakpoint Properties'. Hier kun je overigens ook komen door in de code zelf op de rode stip voor het breakpoint met de rechtermuisknop te klikken en dan 'View Breakpoint Properties' te selecteren. Je kunt er bijvoorbeeld voor kiezen om het programma pas te onderbreken als je dit punt al een aantal keren gepasseerd bent, bijvoorbeeld als de fout pas optreedt bij de achtste keer dat een loop wordt doorlopen. Dan kun je het breakpoint activeren na zeven keer en dan kijken wat er vervolgens gebeurt met F7 enF8.

Je kunt bij de eigenschappen ook een getal invoeren bij 'Hitcount'. Telkens als het programma langs dit breakpoint komt, hoogt het een interne teller op en als die bij het hier opgegeven aantal komt, wordt het programma hier onderbroken. Dat is handig als bijvoorbeeld een routine een vast aantal malen doorlopen moet worden en het niet van een variabele afhankelijk is of er gestopt moet worden of niet. Als je bij 'Auto continue after' een tijd in milliseconden neerzet zal de uitvoer van het programma na die tijd automatisch verder lopen, waarbij je bijvoorbeeld even de tijd hebt om de waarden van een paar variabelen te bekijken om te kijken waar het ongeveer misloopt.

Met dit alles heb je al een hoop mogelijkheden om te zien wat er gebeurt en waarom. Toch kunnen er nog allerlei situaties zijn waarin je vastloopt. Zo kunnen bepaalde resultaten geen foutmelding opleveren, maar wel een verkeerde waarde hebben. Of het resultaat is niet opvraagbaar, bijvoorbeeld in een situatie waarin het niet getoond kan worden door de IDE. Dat is bijvoorbeeld het geval wanneer een With do constructie gebruikt wordt. Maar ook bij

```
While Length(naam) < 7 do
    Insert(' ',naam,Pos('-',naam)+1)
```

kun je niet zien welke positie de Pos()-functie teruggeeft. In dat geval moet je een tussenstap maken met een extra variabele totdat je weet dat het allemaal goed werkt:

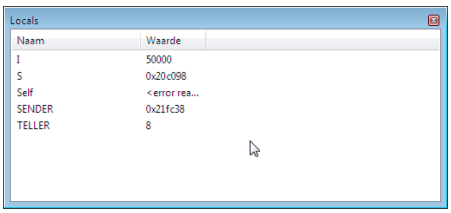
```
x := Pos ('-',naam);
While Length(naam) < 7 do
    Insert(' ',naam,x+1)
```

De waarde van de variabele x is dan gewoon te volgen in de IDE en het watch-venster.

Soms bestaat een variabele alleen maar heel even binnen een bepaalde procedure of functie of wordt hij door de compiler 'weggeoptimaliseerd'. Dan kun je een variabele aanmaken op een hoger niveau dan de functie of procedure zelf, bijvoorbeeld in de header van de unit. Deze is dan binnen de scope van de unit aanwezig.

Wanneer je een overzicht wilt hebben van de geldende lokale variabelen op het punt waar je jezelf in de code bevindt, dan kun je daar een overzicht van opvragen 'Tonen / Debug schermen / Lokale Variabelen' of met Ctrl+Alt+L. Bij het menu-item 'Tonen / Debug schermen' staan nog meer opties die informatie geven over de processen, maar deze vergen wat meer ervaring en inzicht.

Tijdens het compileren wordt in het Berichten-venster getoond of er eventueel problemen of waarschuwingen zijn voor de betreffende code. Het kan gebeuren dat variabelen niet gedeclareerd zijn of geen beginwaarde hebben, of, zoals we al eerder zagen, wel aangemaakt waren maar niet



In het venster van de 'Locals' kun je zien welke lokale variabelen er zijn op de plek waar je je op dat moment in de code bevindt.

meer gebruikt worden. Het devies is dan ook om deze berichten tot het minimum te beperken omdat ze altijd nog een mogelijke bron van onverwachte 'effecten' kunnen zijn.

Proberen maar

Naast deze debuggingtools van de IDE zijn er ook taalelementen toegevoegd om in dubieuze situaties uitkomst te bieden. Wanneer er taken uitgevoerd moeten worden die een foutsituatie kunnen opleveren, bijvoorbeeld bij het openen van een bestand, dan kan dat binnen een try .. except routine:

```
try
    IBDatabase.DatabaseName := LocalPath;
    IBDatabase.Connected := True;
    IBDatabase.Active := True;
except
    MessageDlg(' Lokale database niet gevonden. ',
    mtInformation,[mbOK],0);
    Application.Terminate;
end; // try
```

Als er bij het uitvoeren van de code iets niet goed gaat, wordt de code na except uitgevoerd. Daarmee kun je de juiste acties inbouwen wanneer er iets gebeurt wat niet de bedoeling is.

Er bestaat ook een try .. finally routine:

```
try
    Application.CreateForm(TPE_select, PE_select);
    PE_select.ShowModal;
finally
    PE_select.Release;
end; // try
```

Deze constructie is bedoeld voor dingen die altijd uitgevoerd moeten worden. Denk bijvoorbeeld aan Screen.Cursor := crHourGlass; wat altijd weer teruggezet moet worden naar Screen.Cursor := crDefault;, ook als er iets niet goed is gegaan. Of zoals in dit geval bij de relatie tussen create en free of release. Wat er ook gebeurt, de code na finally wordt altijd uitgevoerd en dat geeft specifieke mogelijkheden om problemen te beperken.

Converteren

We hebben eerder al gezien dat je van de compiler een foutmelding krijgt als je een string in een integer-variabele wilt stoppen of andersom. Er zijn wel routines om deze verschillende variabelentypen naar elkaar te converteren. Maar ook daar kan van alles in misgaan. Stel dat je een getal in een stringformaat hebt (bijvoorbeeld '10') en dat je daar een getal van moet maken. Daar is de functie Val voor bedoeld:

```
Val(tekst, getal, code);
```

In tekst staat de naam van de string waar het getal in staat ('10'), het resultaat komt in getal (10) en code krijgt de waarde 0 als alles goed is gegaan, zoals in dit geval. Maar als de string bijvoorbeeld ('10a12') is, kan de Val-functie daar geen kaas maken maken en krijgt code de index mee op welke positie in de string het misliep. Met de volgende regels kun je kijken of alles goed gegaan is, en een melding geven als dat niet zo is en waar het verkeerd liep. Let er daarbij op dat de teruggegeven waarde bij 1 begint te tellen.

```
var
    i, code : Integer;
    tekst: String;
begin
    Val(tekst,getal,code);
    if code<>0 then
        writeln('fout op positie ',code,' : ',tekst[code])
    else
        writeln('waarde : ',getal);
end.
```

Pascal heeft ook een StrToInt-functie die hetzelfde doet:

```
getal := StrToInt(tekst);
```

Deze heeft als nadeel dat hij geen specifieke foutmelding teruggeeft, maar een EConvertError genereert. De functie TryStrToInt is dan veel handiger:

```
TryStrToInt(tekst, getal);
```

Deze functie levert een TRUE of FALSE op als de conversie lukt dan wel mislukt. Daar is in de praktijk makkelijker mee te werken dan met Val.

Naast een TryStrToInt zijn er ook de vergelijkbare functies TryStrToFloat, TryStrToTime en Try-StrToDate, maar ook TryEncodeDate en TryStrToBool. Met deze functies krijg je meer controle bij het verwerken van informatie die je van buitenaf krijgt via files of invoervelden.

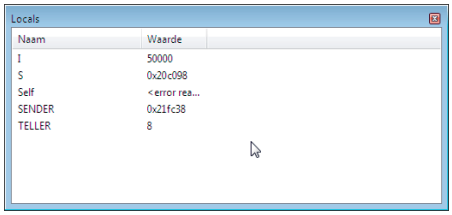
Het nadeel van deze functies is dat je er zelf nog code omheen moet schrijven om eventuele fouten af te vangen. Gelukkig is er ook een StrToIntDef-functie. Daar geef je een waarde aan mee die als resultaat teruggeven moet worden als de conversie niet gelukt is:

```
getal := StrToIntDef(tekst, default);
```

Als de conversie van tekst naar getal niet lukt, krijgt getal de waarde default. Het programma loopt dan gewoon door. Dat scheelt een heleboel foutopvangconstructies met if … then … else.

Al met al heeft Pascal een rijk scala aan mogelijkheden om om te gaan met allerlei situaties en fouten die kunnen voorkomen tijdens het schrijven, testen en gebruiken van een applicatie. Je kunt daarbij kiezen uit oplossingen binnen een programma zelf om fouten tijdens het uitvoeren te ondervangen, of je kunt fouten tijdens het programmeren opsporen met de debugmogelijkheden van de ontwikkelomgeving Lazarus. Bij de Stichting Programmeertaal Pascal kun je een gratis nummer van het blad Blaise aanvragen, waar een zeer uitgebreid artikel instaat over debuggen dat is toegespitst op Delphi. Alle lezers van c't kunnen een gratis downloadversie (PDF) van dat nummer aanvragen op www.delphigg.nl.

In de vorige c't zijn we begonnen met een inleiding in het programmeren in Pascal met de ontwikkelomgeving Lazarus [1]. Via de softlink bij dat artikel kun je een versie van die ontwikkelomgeving downloaden en gebruiken. In het volgende deel gaan we kijken naar classes. We leggen eerst uit wat dit relatief abstracte begrip inhoudt en gaan vervolgens met classes aan de slag om te kijken wat je er mee kunt en waarom ze überhaupt handig zijn. (nkr)



De programmeertaal Pascal heeft zelf ook een aantal mogelijkheden om fouten af te vangen. Bij het omzetten van een string naar een getal heb je bijvoorbeeld meerdere functies tot je beschikking, met elk hun voor en nadelen.

Cursus Pascal

Het doel van deze minicursus is dat je de basisbeginselen leert van het programmeren in Pascal met de ontwikkelomgeving Lazarus en dat je een aantal eenvoudige programma's hebt kunnen maken met behulp van componenten.

Deze cursus is geschreven in samenwerking met de Stichting Ondersteuning Programmeertaal Pascal.

Deel 1. De basis
Deel 2. Debuggen
– Wat is debuggen, wat zijn de mogelijkheden binnen Lazarus en wat kan binnen Pascal zelf
Deel 3. Classes
Deel 4. Multimedia
Deel 5. Netwerken, internet
Deel 6. Databases

Literatuur

[1] Detlef Overbeek, Noud van Kruysbergen, "Hallo Wereld!", Programmeercursus Pascal, deel 1: de basis, c't 4/2012, p.102 c't