Basic Bulletin

20ste jaargang mei 2013 Nummer 1

HCC BASIC interessegroep

Inhoud

Contacten

Functie	Naam	Telefoonnr.	E-mail
Voorzitter	Jan van der Linden	071-3413679	voorz@basic-gg.hcc.nl
Secretaris	Gordon Rahman Tobias Asserstraat 6 2037 JA Haarlem	023-5334881	secr@basic-gg.hcc.nl
Penningmeester	Piet Boere	0348-473115	penm@basic-gg.hcc.nl
Bestuurslid	Titus Krijgsman	075-6145458	t.krijgsman8@upcmail.nl
Redacteur	M.A. Kurvers Schaapsveld 46 3773 ZJ Barneveld	0342-424452	m.a.kurvers@hccnet.nl
Ledenadministratie	Fred Luchsinger	0318-571187	f.luchsinger@kader.hcc.nl
Webmaster	Jan van der Linden	071-3413679	j.vd.linden@kader.hcc.nl

http://www.basic.hcc.nl

Redactioneel

We gaan afscheid nemen van de ‘Nieuwsbrief’. Dat betekent niet dat ik ze niet meer schrijf, maar dat ik in vervolg lesjes, truckjes en ideeën schrijf. Daarom zal de naam niet meer ‘Nieuwsbrief’ zijn, maar ‘Basic Bulletin’.

Ook zou ik het heel leuk vinden, mocht u een leuke tekst hebben of wat dan ook, dit naar mij te e-mailen. Op die manier kunnen we elkaars ideeën uitwisselen en elkaar helpen met vragen en antwoorden.

Marco Kurvers

BASIC leren – PowerBASIC hoofdstuk 8.

Modulair programmeren

Simpele programma's zijn degelijk kort en sequentieel. Met een uitvoer en een logische start vanaf de eerste regel naar de laatste regel, met misschien enkele lussen voor variatie. Als programma's meer complexiteit hebben zullen het aantal codesecties, dat opnieuw gebruikt wordt, meer worden. In plaats van deze secties telkens weer te schrijven, kunt u ze uitpakken en ze in functies of procedures maken. Hierdoor kan de complete structuur van het programma duidelijker worden.

In top-down design van het programma zal de eerste stap op het programma niet hetzelfde zijn als op het ontwerp van deze details. Categorie: ze zijn geschreven voor deze lager-niveau routines, en de ontwerper concentreert zich op hoe de verschillende functies in elkaar passen. Op de volgende stap zal de ontwerper modules creëren bij gebruik van soorten structuren of functionele blokken genoemd als subroutines, procedures, functies en units.

Een *subroutine* is een gelabelde set instructies die uitgevoerd worden wanneer een GOSUB bereikt is. Een *procedure* is zoals een mini programma (ook genoemd als een subprogramma) dat een deel van uw hoofdprogramma uitvoert. Een *functie* is zoals een procedure, maar resulteert een stringwaarde of een numerieke waarde, meestal gerelateerd aan de parameters doorgegeven aan de functie. Al deze structuren bestaan uit statements, lussen en condities. Door de verwerking in een van deze functionele blokken te plaatsen kunt u de complexiteit isoleren, waardoor de rest van het programma eenvoudiger te begrijpen is. Wanneer het tijd is wijzigingen aan te brengen, zult u dit maar één keer hoeven te doen in plaats van op elke plaats dezelfde berekening op te moeten zoeken en die te wijzigen.

Voor nog meer programma flexibiliteit en kracht kunt u units gebruiken. Een *unit* is een afzonderlijk bestand, een module met code die door meer verschillende complete programma's gebruikt kan worden. Units zijn geheel samengesteld uit procedures en functies.

De procedures en gebruiker-gedefinieerde functies van PowerBASIC gaan verder dan de eenvoudige structurering aangeboden door subroutines. Hoewel miljoenen BASIC programma's zijn geschreven met GOSUB/RETURN als hun primaire organisatie-apparaat, raden we u aan deze meer geavanceerde structuren te gebruiken. U kunt gemakkelijk al uw programma's schrijven zonder gebruikmaking van het GOSUB statement.

De procedures en functies in PowerBASIC bieden ware recursie, parameters doorgeven en toegang tot lokale, statische en globale variabelen. Met deze mogelijkheden kunt u "blokken" van het programma isoleren, die taken uitvoeren, en verhinderd wordt dat hun interne variabelen met de variabelen van het hoofdprogramma zich vermengen, zodat u het met elkaar kunt uitvoeren.

Procedures en functies zijn meer dan verschillend. Het meest belangrijke onderscheidt tussen hen is dat *functies resulteren een waarde*. Ze worden daarom impliciet aangeroepen in aanwezige expressies. *Procedures resulteren geen waarde*. Ze worden altijd expliciet aangeroepen met een CALL statement. Bijvoorbeeld:

```
a = b + CubeRoot(c)      ' een functie aanroep in een expressie
CALL OutChar(a)          ' een procedure aanroep
```

PowerBASIC 3.0 accepteert twee variaties van procedures en functies. Eerste, functies hoeven niet meer ingeroepen te worden binnen een expressie. U kunt nu CALL een functie en zijn resultaatwaarde zal genegeerd worden. Tweede, u kunt impliciet een procedure aanroepen dat gedeclareerd is met een DECLARE statement. Bijvoorbeeld:

```
CALL SoortFunctie(Parameter)  ' negeert de resultaatwaarde

DECLARE SUB SoortProcedure(Parameter AS INTEGER)
SoortProcedure 42
```

Subroutines (GOSUB)

Subroutines zijn de ouderwetse manieren van georganiseerd programmeren. Ze bestaan uit gelabelde groepen van statements die eindigen met een RETURN. Voor het uitvoeren van een subroutine gebruikt u een GOSUB instructie met een label die gekoppeld is aan het eerste statement van de subroutine. Zodra een RETURN statement gepasseerd is, wordt er direct teruggekeerd naar het statement na de aangeroepen GOSUB. Bijvoorbeeld:

```
DIM Afspraken%(1:20)
{statements}
GOSUB TelAfspraken
PRINT Totaal%
END
TelAfspraken:
     Totaal% = 0
     FOR i = 1 TO 20
          Totaal% = Totaal% + Afspraken%(i)
     NEXT i
RETURN
```

Zodra de GOSUB gepasseerd is, zal het programma naar de label *TelAfspraken* springen. Een totaal is berekend en de besturing zal na een RETURN terugkeren naar de aangeroepen regel. De volgende regel van het programma is dan uitgevoerd, een PRINT statement die weergeeft wat het totaal bedrag van de afspraken kost.

Voor het potentiële misbruik (bijvoorbeeld springen in het midden van GOSUB blokken of meerdere RETURN statements in een GOSUB blok, of botsingen tussen variabelen met dezelfde naam in subroutines en het hoofdprogramma), worden subroutines niet aanbevolen. Er is niets in een GOSUB blok wat een procedure of functie zo gemakkelijk kan handelen. Als u, na alle lezingen in gestructureerd programmeren en ontwerpen, besluit GOSUB statements te gebruiken, kunt u proberen het concept te gebruiken over een enkele ingangspunt en een enkele uitgangspunt van routines. Dit betekent dat de enige ingang in een GOSUB zal de eerste regel zijn van de GOSUB en de enige uitgang zal de RETURN statement zijn die aanwezig is in de laatste regel. Dus de volgende foutieve gestructureerde GOSUB:

```
FouteVoorbeeld:
     x = x + 1
Ingang2:
     x = x * 2
     y = y * 2
     IF y < 1 THEN RETURN
     .                          ' andere uitvoer
     .
     z = x / y
     .
     RETURN
```

kan herschreven worden als volgt:

```
GoedeVoorbeeld:
     IF conditie THEN x = x + 1
     x = x * 2
     y = y * 2
     IF y >= 1 THEN
          .
          .
          z = x / y
          .
     END IF
     RETURN
```

In dit simpele voorbeeld hebben we een conditionele vlag nodig die ons verteld of de eerste regel uitgevoerd moet worden of niet. Maar eenmaal binnenin is de GOSUB onafhankelijk en is er geen enkele invloed erbuiten. Dit voorbeeld laat een triviale vorm zien dat behulpzaam is voor grotere programma's. Als u eenmaal weet dat er maar één ingang en één uitgang is van een subroutine, is het gemakkelijker als een zwarte doos te behandelen voor foutopsporing.

Functies

PowerBASIC kent drie soorten functies: voor-gedefinieerde functies (zoals COS en LEFT$) die gedefinieerd zijn in de taal, en twee soorten *gebruiker-gedefinieerde functies*. De *handleiding* is speciaal alleen gemaakt voor de voor-gedefinieerde functiedefinities. De twee soorten gebruiker-gedefinieerde functies zijn DEF FN en FUNCTION ("waar" functies).

DEF FN functies

DEF FN functies kunnen eenregelig of uit meerdere regels zijn. Een eenregelige functie is: een functie die alleen één regel definieert. Een meerdere regel functie kan meer regels dan een hebben.

Eenregelige DEF FN functies

De syntaxis voor het definiëren van een eenregelige functie is:

```
DEF FN identifier [ (parameterlijst) ] = expressie
```

Een *identifier* is een naam (zo noemt men het als een identifier) die u maakt voor de functie. De parameterlijst is een optioneel deel dat uit één of meer identifiers bestaat. Deze identifiers representeren gegevensobjecten die naar de functie gestuurd worden zodra de functie tijdens de uitvoering is aangeroepen. Er zijn twee formaten die u kunt gebruiken om de parameters in een functie te definiëren. Als eerste kunt u de parameternamen tonen in hetzelfde formaat zoals u elke toekenning aan een variabele geeft, zoals zo:

```
DEF FN identifier(var1%, var2&, var3$)
```

deze definieert een functie met drie parameters, een integer, een lange integer en een string. Het tweede formaat lijkt op dit:

```
DEF FN identifier(var1 AS INTEGER, var2 AS LONG, var3 AS STRING)
```

De parametertypes zijn hier hetzelfde. Het enige verschil is dat u niet langer meer een type identifier gebruikt. Voor elke functie kunt u niet meer dan 16 parameters gebruiken. De *expressie* definieert de uitvoering. De functie zal de waarde (of ook wel het resultaat) ervan teruggeven.

Stel bijvoorbeeld een meteorologisch programma voor dat een convertering nodig heeft tussen graden

Celsius (die inbegrepen is) en graden Fahrenheit (die weergegeven wordt op het scherm en geaccepteerd wordt van het toetsenbord).

Eenregelige functies zijn perfect voor dit programmatype:

```
DEF FNCtoF(gradenC) = (1.8 * gradenC) + 32
DEF FNFtoC(gradenF) = (gradenF - 32) * .555555
```

Om de temperatuur weer te geven, welke bij het converteren de Celsius waarden vasthoud, gebruik dan FNCtoF (lees als “functie C to F”) in elke statement die een numerieke expressie accepteert, bijvoorbeeld:

```
temp = 100
PRINT FNCtoF(temp)       ' zeg dit als "functie C naar F van temp"
```

Om de waarden te converteren van Fahrenheit naar Celsius, gebruik FNFtoC:

```
INPUT "Voer temperatuur hoogte van vandaag in: ", hoogte
temp = FNFtoC(hoogte)
```

Meerdere regels DEF FN functies
In de meerdere regels functies in PowerBASIC speelt een grotere rol dan dat bij de eenvoudige eenregelige functies in interpretatieve BASIC is. PowerBASIC DEF FN functies spreiden zich over meerdere programmaregels en het effect en gebruik ervan lijkt op een subrouitine die ook een waarde resulteert. De formele syntaxis voor het declareren van een meerdere regels functie is:

```
DEF FNidentifier [ (parameterlijst) ] [SHARED]
     [LOCAL variabele lijst]
     [STATIC variabele lijst]
     [SHARED variabele lijst]
     .
     . statements
     .
     [EXIT DEF]
     .
     .
     .
     [FNidentifier = expressie]
END DEF
```

waarvan *identifier* de functienaam declareert. De *parameterlijst* is een optionele door komma’s gescheiden lijst van formele parameters die variabelen vertegenwoordigen en toepasbaar zijn als de functie is aangeroepen.

Alle variabelen die aanwezig zijn in een DEF FN functie zijn shared (verdeeld) met de rest van het programma, tenzij ze expliciet zijn gedeclareerd als LOCAL of STATIC. Ook zijn alle DEF FN functies onzichtbaar buiten de unit of programma waar ze in aanwezig zijn.

Een faculteit voorbeeld
Om het te illustreren, kunt u een meerdere regels functie *Faculteit* stellen, de statisticus beste vriend. Hoewel faculteiten niet geïncludeerd zijn in PowerBASIC als ingebouwde wiskundige operatoren, kunt u eenvoudig uw eigen meerdere regels faculteit functie schrijven door het voorbeeld te volgen die hier

gegeven is. Hier zijn regelnummers gebruikt uitsluitend voor toelichting—zie de toelichting die volgt. Die zijn natuurlijk optioneel en hebben geen doel in de actuele functie.

U herinnert zich misschien dat de faculteit een positief geheel getal *n*, geschreven als *n!* en uitgesproken als "n faculteit", het product van de positieve gehele getallen kleiner dan of gelijk aan *n* is. Bijvoorbeeld, *6!* Is gelijk aan *6 * 5 * 4 * 3 * 2 * 1* dat uitkomt op *720*. De faculteit *0!* bestaat als *1*.

```
100 DEF FNFaculteit##(nummer%)
110   LOCAL tel%, totaal##
120   IF nummer% < 0 OR nummer% > 1754 THEN
130     FNFaculteit## = -1
140   ELSE
150     totaal## = 1
160     FOR tel% = nummer% TO 2 STEP -1 ' we kunnen ook omhoog werken
170       totaal## = totaal## * tel%
180     NEXT tel%
190     FNFaculteit## = totaal##
200   END IF
210 END DEF
```

Functiedefinities zijn tussen haakjes bij de DEF FN en END DEF statements. Het inspringen van de statements tussen DEF FN en END DEF, met een aantal spaties of een tab-toets, verduidelijkt de structuur. Dit voorbeeld gebruikt een standaard van twee spaties bij elk niveau; u kunt één, vijf, tien of niets gebruiken. Ruimte betekent niets voor de compiler, het inbrengen verbetert de leesbaarheid voor de mensen. Soms moeten we terug en uitzoeken wat een programma doet of *veronderstellen* wat het doet. Het inspringen geeft aanwijzingen wat de programma's voornemen; het verduidelijkt ook de algemene structuur van het programma.

Regel 100 geeft de functie zijn naam en consequent een type (## voor uitgebreide precisie). FN*Faculteit##* heeft één formele parameter, het integer *nummer%*.

Regel 110 declareert twee "lokale" variabelen, *tel%* en *totaal##*. *Lokale variabelen* zijn momenteel variabelen die toegankelijk en zichtbaar zijn alleen binnen de functie en procedure definities.

Regel 120 controleert op fouten van het argument die doorgegeven wordt aan FNFaculteit##. Geen mogelijkheid is er om te achterhalen wat de faculteit is van een negatief getal (geen dergelijk dier) of van een waarde zo groot dat het een resultaat produceert buiten het bereik van 10^{4932} uitgebreide precisie (faculteiten krijgen een snelle hoge waarde—1754! is maar liefst 1.98×10^{4930}). In dat geval laat u de functie de waarde -1 resulteren.

Programma's die gebruikmaken van FNFaculteit## moeten erkennen dat een waarde van -1 een foutconditie geeft en zich ook zo gedraagt.

Regel 150 tot 180 definieert een algoritme om de faculteiten te berekenen. Dit deel van een meerdere regels functie kan zo lang of zo kort als nodig. Regel 190 zet de resulterende waarde van FN*Faculteit##* bij het maken van een toekenning naar de functienaam. Als u niet zorgt voor een toekenning naar de functienaam is de resulterende waarde ongedefinieerd, u moet dus altijd zeker weten dat de functie toegekend is voor het einde.

De definitie van FN*Faculteit##* is beëindigd bij de END DEF statement in regel 210. END DEF keert

terug naar de uitvoer van de statement dat de functie aanriep in de eerste plaats (de END statement geeft een uitwerking in PowerBASIC syntaxis—het wordt gebruikt om een aantal structuurregels te beëindigen).

Is het elke keer verwonderlijk hoeveel permutaties mogelijk zijn met een stapel speelkaarten? FN*Faculteit##*(52) = 8.06581751709438786x10^{67}. Dus als u te wachten staat om te worden behandeld met een hand vol schoppen in bridge, heeft u beter wat vrije tijd.

FN*Faculteit##* is gedefinieerd met een integer formele parameter, waardoor drijvende komma argumenten afgerond worden tot integers voordat ze doorgegeven worden; bijvoorbeeld FN*Faculteit##*(2.7) is hetzelfde als FN*Faculteit##*(3). Als u FN*Faculteit##* aanroept met een getal dat groter is dan *PowerBASIC's* converteer-naar-integer routine kan hebben (groter dan +32767 of kleiner dan -32768), dan krijgt u een uitvoerfout 6, "Overflow".

Hetzelfde proces ontstaat met argumenten van de ingebouwde PowerBASIC functies die integer argumenten vereisen, maar drijvende komma waarden krijgen; bijvoorbeeld, de code LOCATE 2.7,1 plaatst de cursor op regel 3.

Formele en actuele parameters
De variabelen die in een functie definitie parameterlijst verschijnen, worden genoemd als formele parameters. Zij dienen alleen om de functie te definiëren en staan volledig los van andere variabelen in het programma met dezelfde naam. Om dit te laten zien, overweeg dan dit korte programma:

```
100 x = 56
110 PRINT x
120 DEF FNGebied(x,y)
130   x = x * y
140   FNGebied = x
150 END DEF
160 PRINT FNGebied(2,3), x
```

Variabele *x* in de regels 100 en 160 van dit programma is ongerelateerd met de formele parameter *x* die gedefinieerd is in de functie *Gebied* in regel 130 (en eigenlijk al in regel 120). Als dit programma wordt gestart zal *x* zijn waarde aan weerszijden van de aanroep van FN*Gebied* behouden: het afdrukken zal dan aan beide kanten 56 zijn.

Tijdens de uitvoer worden de waarden aan de functie geboden, soms zelfs als daadwerkelijke parameters. In het laatste voorbeeld zijn de numerieke constanten 2 en 3 de actuele parameters toegepast aan FN*Gebied*. De daadwerkelijke parameters kunnen ook variabelen zijn:

```
a = 2 : b = 3
PRINT FNGebied(a, b)
```

of expressies:

```
PRINT FNGebied(a + 2, b * 3)
```

Functie gegevenstypes
Functies kunnen elk van de elf soorten numerieke types resulteren (integer, long of quad integer, byte, word of dubbele word, single-, double- of uitgebreide-precisie drijvende komma, BCD vaste komma of BCD drijvende komma), evenals regelmatige tekenreeksen. Een functietype, zoals een variabele, wordt bepaald door zijn naam; gebruik elke declaratietype met teksttekens of gebruik de instructietype DEF. Bijvoorbeeld,

```
DEFLNG F       ' Alle variabelen die met een "F" beginnen, inclusief
               ' DEF FN functies, zijn standaard long integers
DEF FNIntSquareRoot%(x) = INT(SQR(x))   ' Korte integer als gevolg van
                                        ' type indicator
```

en

```
DEF FNHerhaalEerste$(a$) = LEFT$(a$, 1) + a$     ' string functie
PRINT FNHerhaalEerste$("Hallo")
```

Als u probeert een string functie toe te wijzen aan een numerieke variabele of een numerieke functie aan een string variabele, dan ontstaat er uitvoeringsfout 13, “Type mismatch.”

“Waar” functies in PowerBASIC
In verschil tot een- en meerdere-regel DEF FN functies ondersteund PowerBASIC “waar” functies, zoals de FUNCTION/END FUNCTION statements. FUNCTION heeft meer mogelijkheden dan DEF FN:

- De parameters kunnen worden gegeven als: by value, by copy of by reference, waardoor in de functie de gewijzigde gegevens doorgegeven kunnen worden
- Een hele array kan als een enkele parameter gegeven worden
- U kunt de standaard variabele type kiezen binnen de functie als een SHARED, STATIC of LOCAL
- Variabelen worden automatisch lokaal (uit de code buiten de functie verborgen) tenzij u ze SHARED of STATIC maakt als het standaard variabele type, of tenzij u een bepaalde variabele kenmerk opgeeft met de instructie SHARED of STATIC
- U kunt opgeven of de code privé (onzichtbaar) of openbaar (zichtbaar) moet zijn, dat buiten de module of programma staat (standaard is het openbaar), gebruik daarbij het sleutelwoord PRIVATE of PUBLIC
- Het heeft niet de naam gevende beperkingen dat DEF FN heeft
- In sommige gevallen bent u niet geïnteresseerd in de resultaatwaarde van de functie, u kunt dan gebruik maken van het CALL statement (net zoals u dat doet met procedures.)

De syntaxis van FUNCTION is

```
FUNCTION id [(plijst)] [STATIC|SHARED|LOCAL] [PUBLIC|PRIVATE]
     [LOCAL variabelenlijst]
     [STATIC variabelenlijst]
     [SHARED variabelenlijst]
     .
     .    statements
     .
     [EXIT FUNCTION]
     .
     .
END FUNCTION
```

Functies mogen geen arrays of flex strings resulteren. Ook mag een functie niet meer dan 16 parameters in een keer hebben; deze beperking geldt ook voor DEF FN functies en procedures.

Procedures (SUB)

Procedures zijn codeblokken omgeven met SUB en END SUB statements. De formele syntaxis voor het declareren van een procedure is

```
SUB id [(plijst)] [STATIC|SHARED|LOCAL] [PUBLIC|PRIVATE]
     [LOCAL variabelenlijst]
     [STATIC variabelenlijst]
     [SHARED variabelenlijst]
     .
     . statements
     .
     [EXIT SUB]
     .
     .
     .
END SUB
```

of

```
SUB id INLINE
     .
     . $INLINE metastatements
     .
END SUB
```

De naam *id* declareert de procedurenaam. De *plijst* is een optionele lijst van formele parameters dat variabelen vertegenwoordigt die doorgeven als de procedure aangeroepen is. U kunt niet meer dan 16 parameters voor elke procedure gebruiken, net als met waar functies en DEF FN functies.

Het standaard variabele type binnenin het procedure lichaam is lokaal, tenzij deze met SHARED of STATIC in het hoofd van de procedure gedeclareerd is. Inbegrepen met het sleutelwoord PUBLIC in het hoofd van de procedure maakt het de procedure zichtbaar buiten de unit of programma die daarin wordt weergegeven.

Dat wil dus zeggen: dankzij het sleutelwoord PUBLIC worden de weergegeven procedures buiten het declaratiegebied, zelfs in een andere unit, probleemloos aangeroepen.

Het sleutelwoord PRIVATE maakt het de procedure onzichtbaar. Bij het niet opgeven van het sleutelwoord PUBLIC of PRIVATE zijn de procedures standaard PRIVATE.

Met behulp van de INLINE procedurevorm wordt aangegeven dat de procedure een variabel aantal parameters heeft zonder type en alleen inline assemblercode bevat (zie later hierover in hoofdstuk 15).

Heeft u bestaande programma's die INLINE procedurevormen gebruiken, dan moet u deze procedures converteren naar gelinkte assembleertaal procedures (gebruik het $LINK metastatement) of inregel assembleercode (gebruik ASM) sinds dit door de laatste *PowerBASIC* versie de INLINE procedures ondersteund. Als u INLINE.COM bestanden eerder gebruikt, kunt u ze converteren naar ASM statements met behulp van INLN2ASM.

Het meest belangrijke verschil tussen functies en procedures is dat procedures geen waarde resulteren; zij worden ook niet binnen expressies aangeroepen, hebben geen type en maak geen toekenning aan de procedurenaam. Procedures worden ingeroepen met het CALL statement, zoals het op ingeroepen GOSUB subroutines lijkt.

Overweeg dit programma die de procedure *PrintTotaal* definieert en aanroept:

```
w = 1 : x = 2 : y = 0 : z = 3
```

```
SUB PrintTotaal(a,b,c,d)
     totaal = a + b + c + d
     PRINT totaal
END SUB
CALL PrintTotaal(w,x,y,z)
```

Onthoud, zoals bij functies, dat procedures overal in uw code geplaatst kunnen worden. Na het instellen van de variabelen in de eerste regel, springt het programma naar het CALL statement na de SUB definitie. U kunt ook een procedure aanroepen zonder gebruik van het CALL statement als de procedure geen parameters heeft of als u de haakjes weglaat, bijvoorbeeld:

```
HuidigePagina = 1

TitelScherm
DrukOpToets 10, 15
PRINT "Tot ziens!"
END

SUB TitelScherm
     CLS
     PRINT "Welkom bij Mijn Programma!"
     PRINT
END SUB

SUB DrukOpToets(Rij AS INTEGER, Kolom AS INTEGER)
     LOCATE Rij, Kolom
     PRINT "Druk op <ELKE> toets om verder te gaan."
     DO
     LOOP UNTIL INKEY$ <> ""
END SUB
```

Arrays toepassen

In tegenstelling tot DEF FN functies geven waar functies en procedures u de mogelijkheid hele arrays als argumenten door te geven. De functie of procedure definitie moet verklaren dat zij een array argument verwacht door een passende vermelding in de lijst met formele parameters. Array argumenten worden aangeduid met een set van toegevoegde haakjes aan een formele parameter identifier, optioneel omsloten met een numerieke constante. Deze waarde bepaalt de aantal dimensies in de array, niet de grootte van de array. Om dit te illustreren,

```
SUB TellingNullen(a(1), grootte, telling)
     ' telling resulteert de aantal nullen van de elementen in
     ' eendimensionaal, singel-precisie array a, die grootte + 1
     ' elementen heeft
     telling = 0
     FOR i = 0 TO grootte
          IF a(i) = 0 THEN INCR telling
     NEXT i
END SUB
```

De aanwezigheid van *a*(1) in de lijst met parameters definieert *TellingNullen*'s eerste argument als een eendimensionale array. Het zegt niet hoe groot de array zal zijn. Die taak ligt bij het tweede argument,

grootte. De telling wordt gebruikt om de aantal nullen, gevonden in de array *a*, te resulteren. Het aanroepen van *TellingNullen* gaat bijvoorbeeld zo:

```
grootte = 100
DIM Priemen(grootte)
CALL GeenPriemen    ' alle geen priemen krijgen een geen nul waarde
CALL TellingNullen (Priemen(), grootte, priemTelling)
PRINT "Er zijn"; priemTelling; " priemgetallen <="; grootte
END
```

De volgende keer ga ik hier mee verder en ga ik dieper op in over het gebruik van modulair programmeren, zoals hoe we de programmastroom gaan gebruiken en over geavanceerde onderwerpen met gebruik van procedures en functies.

Variabelen en de gegevens.

Een programmastroom beheren kan het belangrijkste en ook het moeilijkste zijn tijdens het programmeren van uw project of programma. We weten allemaal als BASIC programmeurs en belangstellenden dat dit echter niet op de eerste plaats staat. Wat het meest voorkomt, waardoor schoonheidsfoutjes uit den boze is, zijn de variabelen. We moeten onderstaande punten goed begrijpen:

- Welke variabelen hebben we nodig?
- Als we weten wat we nodig hebben, hoe moeten we ze dan noemen?
- Wat zijn de capaciteiten van de variabelen? Bijvoorbeeld, de limietwaarden en de toegang.
- Wanneer mogen we ze dan gebruiken? Het antwoord hiervan heeft ook weer met de capaciteiten van de variabelen te maken.

Variabelen gebruiken is het beheren van hun waarden. Meestal wordt dit verkeerd onderschat. We moeten goed beseffen dat we de gegevens willen gebruiken; we hebben enkel de variabelen nodig om de gegevens in te kunnen bewaren.

Variabelennamen
Voordat we gegevens kunnen uitwisselen moeten we ze ergens kwijt kunnen. Het geheugen wordt daarvoor gebruikt die uit honderden vakjes bestaat. Deze vakjes zijn de adressen waar de gegevens in opgeslagen worden. Dankzij de variabelen hoeven wij die vakjes niet te gebruiken. De variabelen verwijzen naar de vakjes zodat wij de gegevens erin kunnen schrijven en eruit kunnen lezen.

Om variabelen te kunnen gebruiken moeten we ze namen geven, net zoals wij namen hebben. Zonder een naam kunnen we er niets mee doen.

Een geheugenvakje als: **#FFA221 = 10** is minder duidelijk dan: **A = 10**

Maar dan nog weten we eigenlijk niet… ‘Wat is de bedoeling van **A**?’
Het is daarom heel lastig duidelijke namen te bedenken voor variabelen. Stel dat we de aantal appels willen tellen die geplukt zijn. Een variabele als **A** kunnen we wel gebruiken en ook al begint het met de eerste letter van *Appels*, het is toch beter een andere naam te gebruiken. Een andere programmeur zou niet weten dat variabele *A* met Appels bedoeld wordt.

‘Ja, maar ik wel!’

Dat weten we, maar stel dat u vervangen moet worden. Als een programmeur met het project verder moet, heeft die persoon geen idee wat het allemaal betekent als u geen duidelijke variabelen hebt ge-

bruikt. Het is tijdverspilling als de programmeur dus eerst uit moet zoeken waar de declaraties zijn en wat de doelen van de variabelen zijn, en het kan nog erger: niet gedeclareerde variabelen!

In de meeste BASIC dialecten hoeven we de variabelen niet te declareren. Dat is jammer, want dat zou een betere programmastroom geven voor het programma.

Laten we bovenstaande punten eens goed behandelen.

Welke variabelen hebben we nodig?
Uit het voorbeeld van *A = 10* weten we dat deze variabele numerieke waarden accepteert. Niet gedeclareerde variabelen zonder een type-symbool zijn standaard numeriek. Dit betekent dat we een foutmelding krijgen met toekenningen als:

```
A$ = 10    of    A = "tekst"
```

Oplossing: Draai beide om. Het dollarteken is altijd voor tekstgegevens, nooit voor numerieke gegevens.

Sommige BASIC dialecten, vooral PowerBASIC en Liberty Basic, kennen zeer veel variabelensoorten, zowel met type-symbolen als met type-namen. Ik kan u niet vertellen welke variabelen u nodig heeft, maar bekijk eens onderstaande tabel. Hier staan de meest gebruikte variabelen die altijd hetzelfde doel hebben. Ze kunnen ook voor een ander doel worden gebruikt, maar dit is de standaard.

N.B. *Sommige variabelen worden in hoofdletters gegeven, maar de omschrijving geldt ook voor kleine letters*

Variabele	**Omschrijving**
X	Variabele voor de horizontale coördinaatpunt.
Y	Variabele voor de verticale coördinaatpunt.
Z	Variabele voor het tekenen van afbeeldingen. Wordt gebruikt voor de Z-as in 3D tekeningen, maar kan ook dienen voor de diepte (lagen).
U	Variabele die een bepaalde evenwicht of schaal van de X-as beheert, bijvoorbeeld het middelpunt van het coördinatensysteem.
V	Variabele die een bepaalde evenwicht of schaal van de Y-as beheert, bijvoorbeeld het middelpunt van het coördinatensysteem.
W	Variabele voor trigonometrische hoekfuncties.
RD / Radialen	Deze twee variabelen worden gebruikt om met π / 180 de graden om te rekenen in radialen. Normaal werd altijd de variabele RD gebruikt, omdat toen die tijd variabelen niet langer dan 2 tekens lang konden zijn (BASIC interpreters). Het mag nog steeds, maar een duidelijke naam als Radialen is geen slecht idee.
I	Al vanaf de eerste BASIC dialecten is dit de standaard lus variabele. Deze variabele is een FOR lus variabele, want in een WHILE 'past' deze gewoon niet, alsof de verkeerde maat voor de kleding wordt gebruikt. Hoewel we hem overal mogen gebruiken, weet en kent iedereen deze variabele. Het is daarom een goed idee deze variabele nog steeds op die manier te hanteren.
N	Ook deze variabele kunnen we als variabele I beschouwen. Het verschil is wel dat deze variabele een doel heeft. Het kan uit de lus stappen voor een bepaalde voorwaarde en daar wordt variabele I niet voor gebruikt. De variabele N wordt dikwijls ook genomen om een aantal vormen te bepalen, zoals elementen van een array of het aantal keren tekenen van vertexpunten.
M	Voor meer functionele tekenmethoden moet de lus wel eens genest worden. Om duidelijk aan te geven welke lus binnen een andere ligt, is variabele M de geschikte variabele binnen de N lus. Deze techniek hebt u ook kunnen zien met GW-BASIC. In plaats van tekentechnieken kunnen het ook andere doeleinden zijn. Onthoud dat ook variabele Y wel eens binnen variabele X genest wordt. Als dat

	met tekenen te maken heeft, kunnen er hele andere effecten ontstaan.
T	Deze lus variabele is een oude lusteller. We hebben deze altijd als teller gebruikt en we kunnen niet meer zonder deze T. Wat ook de eindwaarde mag zijn, we tellen altijd tot een maximum, bijvoorbeeld het tellen van een aantal elementen, of tot een gegeven constante. Nooit zullen we aan variabele T vragen: ben je groter of kleiner dan …. ? Het is daarom ook wel het 'broertje' van de variabelen I en N.
NR	Afkorting voor NUMMER. Wordt ook wel samengevoegd met namen die anders met NUMMER te lang worden.

Het is en blijft uw verantwoording welke variabelen nodig zijn. Het tweede punt heeft daarom de vraag hoe u ze wilt noemen als u weet welke u wilt gebruiken.

Als we weten wat we nodig hebben, hoe moeten we ze dan noemen?
De vraag kunnen we alleen beantwoorden als we de theorie hebben gehad. Willen we een adressenprogramma schrijven met Naam, Straat en Huisnummer, Woonplaats, dan weten we voor de praktijk al meteen wat voor variabelen we met de invoer nodig hebben.

Hoe we ze willen noemen? Simpel, zoals we het al genoemd hebben. Een Straat noemen we geen Appel en het Huisnummer geen Teller.

Wat zijn de capaciteiten van de variabelen? Bijvoorbeeld, de limietwaarden en de toegang.
Bekijk eens onderstaande invoercode:

```
INPUT "Naam: "; Naam$
INPUT "Straat: "; Straat$
INPUT "Huisnummer: "; Huisnr
INPUT "Woonplaats: "; Plaats$      ' kan ook WPlaats$ worden genoemd
```

Waar we vooral op moeten letten is de invoerregel met het huisnummer. Hoewel dit gewoon werkt, is deze variabele *Huisnr* geen goed idee. De naam is wel goed, maar zijn capaciteit is beperkt. Als we geen getal invoeren, maar tekst of een andere niet afgerond getal, dan zal (in de meeste BASIC dialecten) de melding 'Redo from start' verschijnen. Gelukkig stopt dat niet de uitvoer van het programma, maar het irriteert natuurlijk wel om zo'n melding te krijgen.

De oplossing is om voor de invoer *tijdelijke* variabelen te gebruiken. Deze variabelen gelden alleen voor de invoer en mogen niet in de rest van het programma gebruikt worden. De variabelen moeten na de invoer toegekend worden aan de variabelen die wel in het programma nodig zijn. Deze techniek noemen we ook wel *gegevens doorvoeren*. In moderne IDE Basic versies (VB 6.0 en VB.NET, en ook in niet-IDE Liberty Basic) is het nu gewoon om het op die manier te doen. Onderstaande codefragment laat de oplossing zien om een foutmelding te voorkomen:

```
INPUT "Naam: "; Naam$
INPUT "Straat: "; Straat$
DO
     INPUT "Huisnummer: "; Huisnr$
LOOP UNTIL VAL(Huisnr$) > 0
INPUT "Woonplaats: "; Plaats$
Adres(n).Naam$ = Naam$
Adres(n).Straat$ = Straat$
Adres(n).Huisnr = VAL(Huisnr$)
Adres(n).Woonplaats$ = Plaats$
```

Hoe die manier precies werkt kunt u zien in het laatste onderwerp van deze BASIC Bulletin.

Het laatste punt: waar mogen we de variabelen gebruiken?
Het laatste punt omschrijf ik nu iets anders, want het heeft te maken met de voorgaande punt. Bovenstaande codefragment heeft er vooral mee te maken. De variabelen die we voor de invoer gebruiken, moeten een beperkte toegang krijgen. Onderstaande lijst is belangrijk om met zulke variabelen om te gaan en een nette, goede, stroom in uw programma te krijgen.

- Declareer variabelen voor een bepaalde invoer *lokaal*. Werkt u met een BASIC dialect die geen lokale variabelen kent, zorg er dan voor dat u zelf de variabelen *nergens* anders gebruikt. Dit geldt niet voor variabelen die genoemd zijn in bovenstaande tabel, omdat zij altijd voor lussen, enzovoort, gebruikt worden.
- Kent uw BASIC dialect modules of units, declareer daar dan alleen de variabelen die in het hele programma nodig zijn, zoals een record type (in bovenstaand codefragment is dat *Adres*). De variabele voor dat type kan dan onder de type-declaratie worden geplaatst.
- De meeste variabelen zijn alleen nodig in procedures en functies. Door zoveel mogelijk variabelen in de juiste codeblokken bij elkaar te houden en zo weinig mogelijk in een globale module, hebt u ook daardoor een betere gegevensdoorstroming.
- Merkt u dat er teveel variabelen zijn, probeer dan uit te zoeken welke doelen zij hebben. De variabelen die hetzelfde doel kunnen hebben, kunt u 'inpakken'; het is niet verkeerd er een array van te maken of als uw BASIC dialect het kent, maak dan records of class-types. Meer daarover in het laatste onderwerp van deze BASIC Bulletin.

Beslissingen nemen.

Een programmastroom, ook wel de *program flow* of de *control flow* genoemd, werkt niet sequentieel. In een programma moeten beslissingen genomen worden welke codefragmenten uitgevoerd mogen worden. Deze codefragmenten zijn de vertakkingen in het programma. Dit wordt de boomstructuur genoemd.

Condities
Beslissingen nemen we door condities op te geven. Deze condities zijn de voorwaarden die we door middel van operatoren kunnen samenvoegen tot één voorwaarde. We kunnen beslissen dat we dit én dat of dat we dit óf dat kunnen nemen, of dat we dit én dat óf zus én zo kunnen nemen.

Een voorwaarde resulteert altijd op een 'waar' of 'onwaar'. In programmeertalen is dat een 'true' of een 'false'. Bijvoorbeeld:

```
A < 10 AND B >= 100
```

Dit is een conditie dat een voorwaarde bepaald of het 'waar' of 'onwaar' is, namelijk A kleiner dan 10 én B groter of gelijk aan 100. Hoe u het wilt noemen maakt niet uit, voorwaarde of conditie. Het principe is hetzelfde:

```
IF <voorwaarde> THEN     of
IF <conditie> THEN
```

Bovenstaande conditie gebruiken we bijvoorbeeld zo:

```
IF A < 10 AND B >= 100 THEN
```

THEN of ELSE?
Is de voorwaarde 'waar', dan wordt de code achter THEN uitgevoerd. Is het echter 'onwaar', dan wordt de code achter ELSE uitgevoerd. Andere programmeertalen kennen de ELSE-vertakking, maar voor BASIC is dat niet het geval. In de oude BASICA tijd bestond er nog geen ELSE. Langzaam maar

zeker kwam het statement opdagen, maar in elk BASICA dialect was de IF … THEN … ELSE structuur weer anders:

```
IF … THEN      ' zonder ELSE (VIC 20 en C64 BASIC 2.0)
IF … THEN … :ELSE: …     ' met ELSE tussen twee dubbelepunten
                         ' C64 – Simon's BASIC
IF … THEN … :ELSE …      ' met ELSE voorafgaand door een dubbele-
                         ' punt (C128 BASIC 7.0)
```

GW-BASIC was de eerste BASICA versie met een ELSE zonder dubbelepunt, maar helaas verdween de handige BEGIN … BEND codeblok-structuur die in BASIC 7.0 toegepast werd.

Voor de gewone BASIC dialecten kunnen we het volgende schrijven:

```
IF A = 10 AND B > 0 THEN
     B = B – 1
ELSE
     B = 100
END IF
```

Deze beslissing kan verwarrend zijn. Wanneer wordt B gelijk gemaakt aan 100? Aan de operator kunnen we zien dat dit niet alleen geldt als B niet groter is dan 0, want ook al zou B groter zijn dan 0; de ELSE wordt alsnog uitgevoerd als A niet gelijk is aan 10. Misschien is dat niet de bedoeling. We willen dat beide ('waar' en 'onwaar') alleen werkt als A gelijk is aan 10, anders moet gewoon de hele beslissing overgeslagen worden.

De oplossing is: beslissingen nesten!

Werken met geneste beslissingen
Voordat ik daarmee verder ga, eerst hier eens een oplossing bekijken nog zonder te nesten:

```
IF A = 10 AND B > 0 THEN
     B = B – 1
ELSE
     IF A = 10 THEN
          B = 100
     END IF
END IF
```

Meestal wordt zo'n probleem op die manier opgelost, maar u zult wel zien dat dit dubbelop werkt. Twee keer A = 10 controleren, alleen maar om ervoor te zorgen dat B op de juiste wijze de waarde 100 krijgt, geeft geen goede programmastroom aan een beslissingsstructuur. We moeten echt de nesttechniek gebruiken om een betere programmastroom te krijgen.

```
IF A = 10 THEN
     IF B > 0 THEN
          B = B – 1
     ELSE
          B = 100
     END IF
END IF
```

De functie IIF

Sommige BASIC dialecten kennen de functie IIF. Hier kan ook een beslissing worden genomen en kan een ‘true’ of een ‘false’ waarde door de functie teruggegeven worden. De syntaxis is:

```
<var> = IIF(<conditie>,<true-waarde>,<false-waarde>)
```

Een voorbeeld is:

```
N = IIF(I > Teller, 1, 0)
```

Dit voorbeeld is typisch het gebruik van een *vlag*. Variabele I kan een lusteller zijn. Is deze een bepaalde waarde gepasseerd, dan zal variabele N de waarde 1 krijgen, anders de waarde 0.

Het gebruik van deze functie kan best nuttig zijn, vooral met het bepalen van de vlaggen. Bekijk eens onderstaand voorbeeld:

```
B = IIF(N = 1, TRUE, FALSE)
```

Hoewel deze werkt, kan de regel nog korter worden geschreven:

```
B = N = 1
```

Heeft uw BASIC dialect niet de IIF functie, dan kunt u de verkorte vlagtoekenning gebruiken, alle BASIC talen kennen deze techniek. Voor de andere IIF voorbeelden kunt u ook de normale IF … THEN … ELSE gebruiken.

Waar men vooral snel de mist in gaat is onderstaand ‘schoonheidsfoutje’:

```
N = IIF(M = 10, M, N)
```

Tijdens het programmeren heb je niet snel in de gaten wat hier aan de hand is, maar als u een slimmerik bent dan kunt u wel zien wat er gebeurd als M niet gelijk is aan 10. Dus mocht dat inderdaad het geval zijn, dan gebeurd er dit:

```
N = N
```

Ziet u waarom? Laten we eens de functie ‘uitpakken’ en in een gewone IF … THEN … ELSE structuur zetten:

```
IF M = 10 THEN
     N = M
ELSE
     N = N     ' dit is het schoonheidsfoutje
END IF
```

Het is dus niet altijd handig de IIF functie te gebruiken, vooral niet als u de ELSE vertakking niet nodig hebt.

De prioriteiten in de condities

In de condities gelden bepaalde regels, net als bij het rekenen. Alles werkt van links naar rechts en de condities hebben een hogere prioriteit dan de beslissingsoperatoren. Dat betekent dat al het rekenwerk en controleerwerk eerder wordt gedaan en dan pas alles tot één conditie samengevoegd wordt. Onderstaande lijst geeft nog eens een verduidelijking.

1. Alle rekenoperatoren (+ -, * / enzovoort)
2. Alle controleoperatoren (<, >, <=, >=, <> en NOT)
3. Alle beslissingsoperatoren (AND, OR, ANDALSO en ORELSE)

Haakjes gebruiken is in condities ook toegestaan. Deze hebben dezelfde prioriteiten als in expressies. In BASIC werkt onderstaande conditie:

```
A > B OR C < D AND K <> 100
```

op dezelfde manier als:

```
(A > B) OR (C < D) AND (K <> 100)
```

Echter, in andere talen (C++ bijvoorbeeld) is de prioriteitsvolgorde anders, omdat in die programmeertalen eerst de beslissingsoperatoren uitgevoerd worden. Haakjes gebruiken is daardoor verplicht.

De NOT operator is een operator apart. In BASIC is de prioriteit lager waardoor haakjes niet nodig zijn, maar in andere talen is dat ook weer andersom. In C++ werkt het symbool **!** als een NOT operator.

```
IF NOT conditie THEN     ' de volgende regel werkt hetzelfde
IF NOT (conditie) THEN

if (!conditie)      // in C++ staat het uitroepteken binnen de haakjes
if (!(conditie))    // of zo (vergeet niet: then bestaat niet in C++)
```

In Pascal is de prioriteit ook anders zodat haakjes nodig zijn. Ze zijn echter niet verplicht wanneer geen beslissingsoperatoren worden gebruikt.

Een schoonheidsfoutje kan ook gebeuren als een bepaalde volgorde ingewikkeld of onjuist is:

```
IF NOT A AND B THEN
```

Dit moeten we niet lezen als: niet A en B dan … zoals wij wel eens zeggen. De NOT operator staat op de tweede plaats en de AND operator op de derde plaats, zodat eerst NOT A uitgevoerd wordt en dan pas, zonder NOT, conditie B. Een oplossing zou kunnen zijn:

```
IF NOT A AND NOT B THEN
```

Dit werkt normaal, maar stel dat we van A tot en met E willen? We moeten dan wel heel erg veel NOT operatoren gebruiken. Niet handig dus. Een betere oplossing is, haakjes gebruiken:

```
IF NOT (A AND B) THEN
```

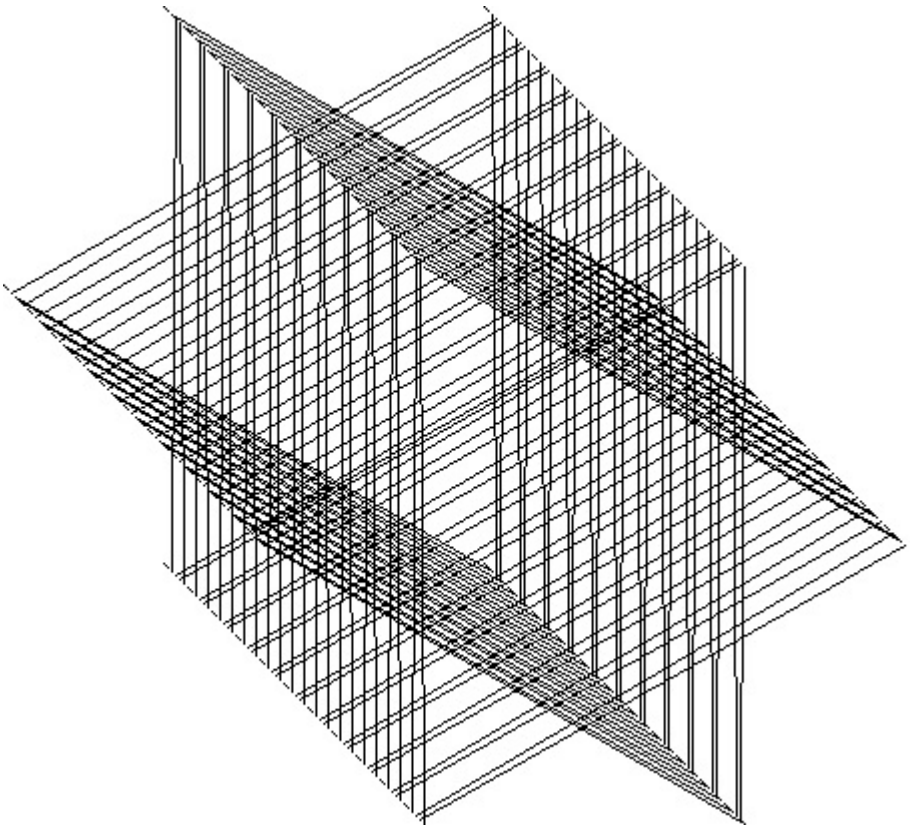
Hoe dan ook, wilt u een beslissing nemen? Neem dan altijd de juiste!

Grafisch programmeren in BASIC.

Na het mooie boek ‘Grafisch programmeren in GW-BASIC’ ben ik van plan verder te gaan met het tekenen in BASIC. Andere BASIC dialecten komen ook aan bod, zoals Dark Basic, maar vooral Visual Basic 2010 zal de hoofdzaak zijn. Er is niet veel verschil vanaf VB 2005 tot en met VB 2010 wat de grafische methoden betreft.

Programma 1 – Gezichtsbedrog

Het eerste programma en voorbeeld dat ik laat zien is iets waar je ogen een verkeerd beeld geven. Dit programma tekent driehoeken in verschillende hoeken in graden, maar omdat er zoveel driehoeken zijn lijkt het net alsof er diagonale rechthoeken zijn getekend.

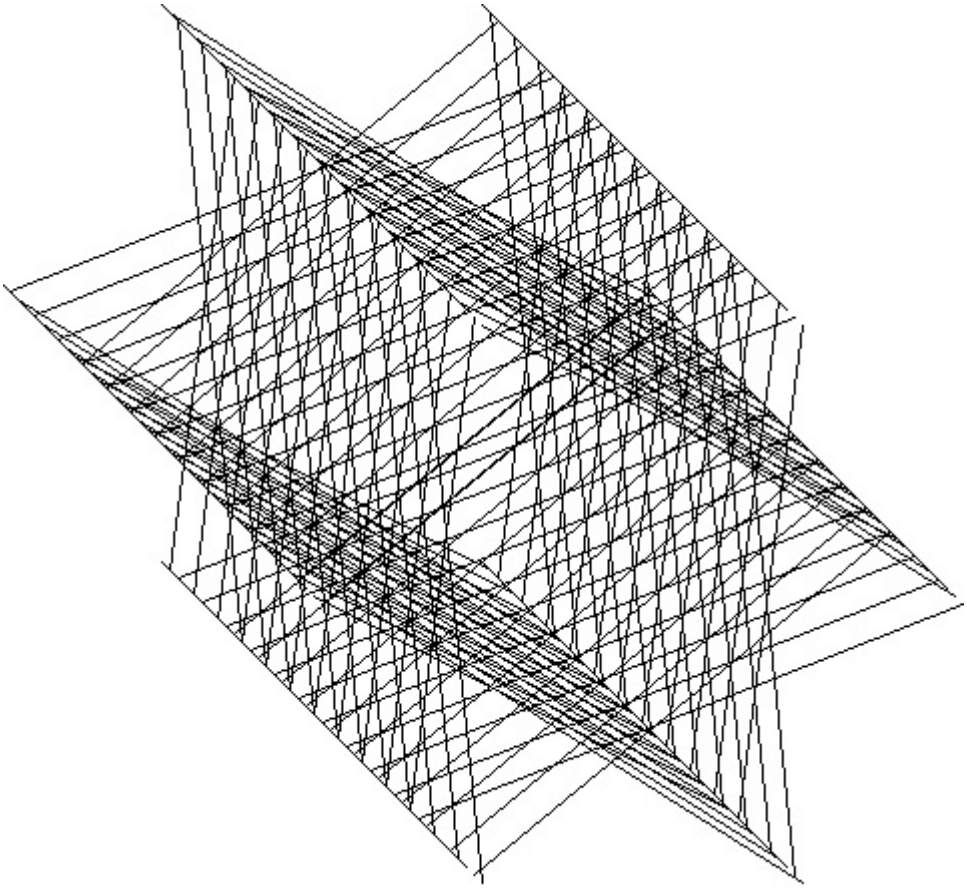


```
Public Class frmGraph1

    Private Sub frmGraph1_Paint(…, ByVal e As …PaintEventArgs) …
        Dim W As Double = 60 * Math.PI / 180
        Dim H As Double = 0.5
        Dim V As Integer = 160, U As Integer = 160
        For J As Integer = 0 To 64
            For K = 0 To 128 Step 4
                Dim X1 As Integer = Int(U + V * Math.Cos((J + K) * W) + H)
                Dim Y1 As Integer = Int(U - V * Math.Sin((J + K) * W) + H)
                Dim X2 As Integer = Int(U + V * Math.Cos((J - K) * W) + H)
                Dim Y2 As Integer = Int(U - V * Math.Sin((J - K) * W) + H)
                e.Graphics.DrawLine(Pens.Black, _
                                    X1 + K, Y1 + K, X2 + K + 2, Y2 + K + 2)
            Next
        Next
    End Sub
End Class
```

U kunt zien dat het net lijkt alsof er schuine rechthoeken zijn getekend terwijl het alleen maar driehoeken zijn. Deze driehoeken snijden elkaar. Bovendien heeft elke driehoek een dubbele lijn. Dat komt doordat in *DrawLine* telkens + 2 opgeteld wordt. Hoe hoger de waarde, hoe meer afstand de dubbele lijnen hebben.

In de tweede tekening heb ik dat ook gedaan door een andere waarde op te geven voor de dubbele lijnen. Wat echter ook ontstaat is een snijpunt effect.



Wijzig de DrawLine regel zoals hieronder:

```
e.Graphics.DrawLine(Pens.Black, X1 + K, Y1 + K, X2 + K + 32, Y2 + K + 32)
```

Dit effect zorgt ervoor dat het snijpunt voor een spiegelende lijn zorgt precies wanneer het punt de tegenovergestelde lijn nadert. Of dat ook werkelijk op die manier gebeurt is moeilijk te zeggen. Door steeds een andere waarde op te geven, in plaats van 32, zult u telkens andere effecten zien. Maar de hoofdzaak dat er driehoeken getekend worden blijft bestaan.

Voor de volgende keer zal ik eens kijken wat we nog meer kunnen doen met het tekenen in de wiskunde. Ook DarkBASIC en Liberty BASIC komen erbij. Liberty BASIC is een zeer geschikte BASIC taal om te tekenen.

Printers aansturen zonder gebruik van LPRINT.

In vele BASIC variaties wordt nog vaak gebruik gemaakt van DOS uitvoer (GW-BASIC, QuickBASIC, PowerBASIC enzovoort) betreffende over het correct beheren van Windows afdrukken:

1. Sluit de printerpoort aan het einde van de afdruktaak.
2. Afdrukken met printers die niet DOS compatible zijn, zoals USB GDI printers of virtuele printers (fax printer drivers, PDF schrijvers, enzovoort).

Afdrukken in BASIC

DOS BASIC programma's gebruiken de LPRINT statements om af te drukken, en direct de gespecificeerde karakters naar de LPT1 te jzenden: alleen de parallelle poort.

Gelukkig heeft de BASIC taal al een oplossing voor dit probleem. Bekijk eens deze code:

```
10 LPRINT "Hallo wereld"
```

U kunt hetzelfde gedrag verkrijgen door het te veranderen als:

```
10 OPEN "LPT1:" FOR OUTPUT AS #1
20 PRINT #1,"Hallo wereld"
30 CLOSE #1
```

met twee hoofdvoordelen:

1. U kunt verschillende poorten specificeren in regel 10, zoals LPT2:, LPT3:, of zelfs een FILE van disk.
2. U kunt de printer poort sluiten aan het eind van het afdrukken, zodat Windows keurig uw werk uit de wachtrij kan halen.

Praktisch gezien hebt u alleen het OPEN commando nodig aan het begin van elke afdruk en het CLOSE commando aan het eind, dan elke LPRINT statement moet worden veranderd als PRINT #1, dat gemakkelijk gedaan kan worden met de Search & Replace dialoogscherm.

BASIC kan niet direct USB, DOT4 en andere Windows poorten adresseren, want die waren niet in de tijdperk van DOS beschikbaar, dus u kunt ze niet invoegen in regel 10. Bovendien zijn meer en meer printers tegenwoordig GDI (ook bekend als alleen-Windows of host-gebaseerde printers), die niet kan worden aangedreven door een DOS programma, noch als ze zijn aangesloten op de LPT1 of als u de output van LPT1 naar een van de poorten boven met een eenvoudige omleiding nut, zoals de opdracht NET USE Windows doorstuurt.

U heeft dus een Windows programma nodig, zoals *Printfil*, die de BASIC afdruktaak converteert in een GDI Windows taak.

Printfil kan worden geconfigureerd om de LPT1 output vast te leggen, maar als programmeur mag u de voorbeeldcode zoals hierboven als volgt veranderen:

```
10 D$="C:\BASIC\PRINTFIL.TXT"
20 REM D$="LPT1:"
30 OPEN D$ FOR OUTPUT AS #1
40 PRINT #1,"Hallo wereld"
50 CLOSE #1
```

Afhankelijk van regel 20 in commentaar staat of niet, kunt u uw BASIC programma direct naar de LPT1 poort (net als LPRINT) of naar een bestand op schijf maken. Nu kunt u “C:\BASIC\PRINTFIL.TXT” invoegen in het “bestand om te controleren” veld van de Printfil configuratie dialoogvenster dat Printfil automatisch vastlegt van uw elementaire afdruktaken en ze naar elke printer, geïnstalleerd in uw Windows Control Panel (inclusief USB, GDI, DOT4 en virtuele printers), door te sturen zonder te hoeven configureren voor het vastleggen van de LPT1: uitvoer.

Wilt u meer weten over Printfil of wilt u de Engelstalige pagina van dit onderwerp bekijken? Kijk dan op:

www.printfil.com/article/basic-print-windows-printer.htm

en direct de pagina over Printfil:

Enumeraties en records.

BASIC IDE programmeertalen kennen de mogelijkheid van enumeraties en records. Ook PowerBASIC kent die mogelijkheid, hoewel er helemaal geen sprake is van records.

Waarom noem ik het dan records en wanneer kunnen we het records noemen?

Een ‘echt’ record bestaat niet in BASIC, dus het sleutelwoord daarvan valt al af. Het gebruik van het sleutelwoord TYPE is voor BASIC voldoende om het een record te noemen. Dan zouden we het onderwerp ook ‘Enumeraties en types’ kunnen noemen. Inderdaad, maar in principe maakt dat niet uit.

Het maakt pas uit wanneer we gebruik zouden maken van VB.NET 2003 en hoger, want die Basic versies zijn zo gestructureerd opgebouwd dat ze eruit zien als C++. Het sleutelwoord TYPE bestaat niet meer en is vervangen door ‘STRUCTURE’. Het voordeel is nu dat we in een Structure type ook methoden en eigenschappen mogen gebruiken. Ze lijken dan ook wel op klassen. Het verschil is wel dat een Structure type geen constructor en destructor heeft.

Enumeraties
Enumeraties vergemakkelijken de leesbaarheid van een programma. We kunnen waarden vervangen door namen. Deze namen zijn de identifier constanten die vaste waarden vasthouden. Onderstaand voorbeeld laat een enumeratietype zien met identifiervelden:

```
Public Enum EFruit
     Appel     ' of Appel = 0
     Peer
     Banaan
End Enum
```

U mag een variabele declareren van het type `EFruit`. Dit betekent echter niet dat de velden lokaal in het type staan. In een normaal type is dat wel het geval. Een enumeratie kan daarom met of zonder type aanduiding worden gebruikt, zoals onderstaande code laat zien:

```
Dim P As Integer

P = EFruit.Appel + 10
Debug.Print P
Debug.Print Appel + 20
Print Banaan
```

De identifier Appel wordt zonder het enumeratietype op te geven gewoon herkent en dat geldt ook voor Banaan. Zouden we dan op moeten passen dat we niet in meerdere enumeratietypes dezelfde identifiervelden gebruiken? Laten we eens twee enumeratietypes maken die elk een fruitmand type zijn:

```
Public Enum EFruitmand1
     Peer
     Banaan
     Perzik
     Meloen
End Enum
```

```
Public Enum EFruitmand2
     Kokosnoot
     Druiven
     Peer
     Appel
     Meloen
     Komkommer
End Enum

Dim Fruitmand1 As EFruitmand1, Fruitmand2 As EFruitmand2
```

Laten we nu eens onderstaande regel uitvoeren:

```
Print Peer
```

De compiler komt meteen met een foutmelding:

Compile error: Ambiguous name detected: Peer

We kunnen best dezelfde veldnamen gebruiken, maar we moeten dan wel verplicht het enumeratietype gebruiken. Vervangen we de regel in:

```
Print EFruitmand1.Peer; "    "; EFruitmand2.Peer
```

dan is er niets aan de hand en kan de compiler bepalen welke Peer gebruikt moet worden via het opgegeven enumeratietype. Denk er dus aan dat het globaal gebruiken van de velden ongeschikt is en tot foutmeldingen kan leiden.

Records
Zoals ik al eerder vertelde kent BASIC niet de echte recordtypes met een sleutelwoord ‘Record’, zoals in Delphi. Een normaal type in BASIC kan echter wel als een soort record dienen.

```
Public Type TFruitmand
     Appel As Integer
     Peer As Integer
     Banaan As Integer
End Type

Dim Fruitmand As TFruitmand
```

Nu mogen we waarden toekennen aan de recordvelden en ook lezen, anders dan bij de enumeratievelden die alleen-lezen zijn.
We zien dat we aan een fruitmand record niet veel aan hebben. Wat moeten we toekennen aan Appel en aan Peer? Daarom zijn zulke velden het meest geschikt voor enumeratievelden. Als we een record willen maken met een fruitmand, maak dan eens een recordtype zoals hieronder:

```
Public Type TFruitwinkel
     Fruitmand1 As EFruitmand1
     Fruitmand2 As EFruitmand2
End Enum

Public Fruitwinkel As TFruitwinkel
```

Onderstaande code laat zien hoe we de recordvelden kunnen gebruiken:

```
Dim F1 As EFruitmand1
Dim F2 As EFruitmand2

F1 = Fruitwinkel.Fruitmand1
F2 = Fruitwinkel.Fruitmand2
Print F1.Peer; "    "; F2.Komkommer
```

De code ziet er vreemd uit. Laten we het eens onder de loep nemen.

De variabele *Fruitwinkel* declareren we met het sleutelwoord *Public*, niet met het sleutelwoord *Dim*. Zouden we het toch met Dim declareren, dan zal Basic het record *Fruitwinkel* in een andere module niet kennen. Dit is ook de reden dat Dim declaraties standaard lokaal zijn en niet openbaar.

In de andere module zouden we graag onderstaande twee regels willen gebruiken:

```
Dim P As Integer

P = Fruitwinkel.Fruitmand1.Peer
```

Helaas staat de compiler dit niet toe. Een enumeratieveld, waarvan een veld van een enumeratietype in een record is gedeclareerd, kunnen we niet direct benaderen. We moeten een omweg gebruiken om de waarde van Peer te kunnen lezen. Vandaar dat we twee variabelen van de enumeratietypes eerst moeten declareren, dan het enumeratieveld vanuit het record toekennen aan de variabele en dan pas de enumeratievelden Peer en Komkommer kunnen benaderen.

Tja, het is een omweg maar het is niet anders. Dat geldt ook voor klassen en objecten en collecties. Daar ga ik het in de volgende Bulletin over hebben.

Veel plezier met het gebruik van enumeraties en records.

Cursussen

Liberty Basic:
Cursus en naslagwerk, beide met voorbeelden op CD-ROM, € 6,00 voor leden. Niet leden € 10,00.

Qbasic:
Cursus, lesmateriaal en voorbeelden op CD-ROM, € 6,00 voor leden. Niet leden € 10,00.

QuickBasic:
Cursusboek en het lesvoorbeeld op diskette, € 11,00 voor leden. Niet leden € 13,50.

Visual Basic 6.0:
Cursus, lesmateriaal en voorbeelden op CD-ROM, € 6,00 voor leden. Niet leden € 10,00.

Basiscursus voor senioren, Windows 95/98,
Word 97 en internet voor senioren, (geen diskette). € 11,00 voor leden. Niet leden € 13,50.

Computercursus voor iedereen: tekstverwerking met Office en eventueel met VBA, Internet en programmeertalen, waaronder ook Basic, die u zou willen leren.
Elke dinsdag, woendag en vrijdag in buurthuis Bronveld in Barneveld van 19:00 uur tot 21:00 uur op de dinsdag en van 9:00 uur tot 11:00 uur op de woensdag en vrijdag. Kosten € 5,00 per week.
Meer informatie? Kijk op 'http://www.i-t-s.nl/rdkcomputerservice/index.php' of neem contact op met mij.

Computerworkshop voor iedereen; heeft u vragen over tekstverwerking of BASIC, dan kunt u elke 2de en 4de week per maand terecht in hetzelfde buurthuis Bronveld in Barneveld van 19:15 uur tot 21:15 uur. Kosten € 2,00.
Meer informatie? Kijk op ‘http://www.buurthuisbronveld.nl’ of neem contact op met mij.

Software

Catalogusdiskette,	€ 1,40 voor leden. Niet leden € 2,50.
Overige diskettes,	€ 3,40 voor leden. Niet leden € 4,50.
CD-ROM’s,	€ 9,50 voor leden. Niet leden € 12,50.

Hoe te bestellen

De cursussen, diskettes of CD-ROM kunnen worden besteld door het sturen van een e-mail naar penm@basic-gg.hcc.nl en storting van het verschuldigde bedrag op:

ABN-AMRO nummer 49.57.40.314
HCC BASIC ig
Haarlem

Onder vermelding van het gewenste artikel. Vermeld in elk geval in uw e-mail ook uw adres aangezien dit bij elektronisch bankieren niet wordt meegezonden. Houd rekening met een leveringstijd van ca. 2 weken.
Teksten en broncodes van de nieuwsbrieven zijn te downloaden vanaf onze website (http://www.basic.hccnet.nl). De diskettes worden bij tijd en wijlen aangevuld met bruikbare hulp- en voorbeeldprogramma's.
Op de catalogusdiskette staat een korte maar duidelijke beschrijving van elk programma.

Alle prijzen zijn inclusief verzendkosten voor Nederland en België.

Vraagbaken

De volgende personen zijn op de aangegeven tijden beschikbaar voor vragen over programmeerproblemen. Respecteer hun privé-leven en bel alstublieft alleen op de aangegeven tijden.

Waarover	Wie	Wanneer	Tijd	Telefoon	Email
Liberty Basic	Gordon Rahman	ma. t/m zo.	19-23	(023) 5334881	grahman@planet.nl
MSX-Basic	Erwin Nicolai	vr. t/m zo.	18-22	(0516) 541680	basic@lordthanatos.com
PowerBasic CC	Fred Luchsinger	ma. t/m vr.	19-21		f.luchsinger@kader.hcc.nl
QBasic, QuickBasic	Jan v.d. Linden				j.vd.linden@kader.hcc.nl
Visual Basic voor Windows	Jeroen v. Hezik	ma. t/m zo.	19-21	(0346) 214131	j.a.van.hezik@kader.hcc.nl
Visual Basic .NET	Marco Kurvers	do. t/m zo.	19-22	(0342) 424452	m.a.kurvers@hccnet.nl
Basic algemeen, zoals VBA Office	Marco Kurvers	do. t/m zo.	19-22	(0342) 424452	m.a.kurvers@hccnet.nl
Web Design, met XHTML en CSS