Basic Bulletin

20ste jaargang juli 2013 Nummer 2

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

In dit BASIC Bulletin is er een onderwerp dat voor u best belangrijk kan zijn: hoe communiceren de parameters en de argumenten met elkaar? Wat gebeurt er als er een waarde of variabele doorgegeven wordt? Wat er zich tussen de deuren van de procedures en functies afspeelt, zal ik aan u met voorbeelden uitleggen.

De komende bulletins zullen er vaak onderwerpen zijn over Liberty BASIC. Er zullen onderwerpen komen die er net zo uit gaan zien als bij PowerBASIC. Veel theorie en voorbeelden dus.

Marco Kurvers

BASIC leren – PowerBASIC hoofdstuk 8 (vervolg).

Modulair programmeren

Programmeerstroom, of in de technische informatica ook genoemd: Program flow

U hebt geen directe programmeerstroom door een procedure of functie met een geïnitialiseerde stap (zoals u wel moet doen met eenregelige functies in interpretive BASIC). De compiler ziet uw definities waar ze gepositioneerd zijn. Eigenlijk zijn de posities van de procedure en functie definities immaterieel binnen een programma. Een functie kan worden gedefinieerd in regel 1 of in regel 1000 van een programma, ongeacht waar het is gebruikt. Nochtans is het beter om niet de procedures en functies te verstrooien, zoals wilde bloemen in een weide. Volg in plaats daarvan deze richtlijnen om een consistent formulier voor uw programma's te ontwikkelen:

- Gebruik $INCLUDE om gemeenschappelijke routines onder te brengen.
- Gebruik functies om vaak gebruikte berekeningen uit te voeren.
- Behandel uw procedure- en functiedefinities als geïsoleerde eilanden van code. Spring er niet in of uit met GOTO, GOSUB of RETURN statements – u kunt onvoorspelbare of rampzalige resultaten krijgen, of beide.
- Groepeer de procedures door de functie- of tijdstip-uitvoering en isoleer hen vervolgens in een unit. Op deze manier, zodra de set routines is geschreven, getest en gecompileerd naar een unit, kunnen veel verschillende programma's gekoppeld worden in de unit zonder alle broncode die de unit maakt te hoeven hercompileren, telkens wanneer er een wijziging wordt aangebracht in het hoofdprogramma.

In tegenstelling tot subroutines kan ook de uitvoering van het programma niet per ongeluk "vallen" door een procedure of een functie. Wat betreft het uitvoeringspad van een programma is, zijn de procedure- en functiedefinities onzichtbaar. Bijvoorbeeld, wanneer dit vier-regel programma uitgevoerd is:

```
CALL PrintZomaarIets
SUB PrintZomaarIets
     PRINT "Geprint van binnen PrintZomaarIets"
END SUB
```

het bericht Geprint van binnen PrintZomaarIets verschijnt maar één keer per aanroep.

Onthoud dat procedure- noch functiedefinities genest mogen worden; dat wil zeggen, u kunt geen andere procedure of functie binnenin een procedure- of functiedefinitie definiëren. Maar u mag wel andere procedures en functies binnenin een procedure- of functiedefinitie aanroepen.

Argumentcontrole

Anders dan andere BASIC compilers, controleert PowerBASIC om zeker te weten dat het aantal en type argumenten die de procedure en –functie aanroepen overeenkomen met het aantal en type formele parameters in de corresponderende definities. Probeer bijvoorbeeld onderstaand programma te compileren:

```
FUNCTION Dummy(a,b)
      .
      .
      .
END FUNCTION
t = Dummy(3)
```

resulteert in een “Parameter mismatch” fout, omdat t = Dummy(3) alleen maar één argument geeft aan *Dummy*, terwijl deze gedefinieerd is om twee argumenten te vereisen.

Geavanceerde onderwerpen

Met wat u tot nu toe geleerd heeft, krijgt u een begin over het schrijven van goede, heldere, gestructureerde programma’s. Echter, om onvoorziene of hard-aan-debug problemen te voorkomen, zijn sommige onderwerpen ingewikkeldere lichtpuntjes waar u zich van bewust moet zijn. Met de parameter doorgeef methoden, het toepassingsgebied, het recursief en zeer belangrijk de PowerBASIC units, komen deze allen in de rest van dit hoofdstuk aan de orde.

Doorgegeven parameters

Er zijn subtielere maar belangrijke verschillen tussen functies en procedures. Om deze verschillen te begrijpen, is het nodig te weten hoe PowerBASIC de procedure- en functieaanroepen aanstuurt tijdens de uitvoer.

Tabel 8.1 Verschillen tussen procedures en functies

Actie	**DEF FN**	**Functies**	**Procedures**
Resulteer een waarde	Ja	Ja	Nee
Methode aanroepen	Aanroepen binnen expressies; gebruik CALL om het resultaat te vermijden	Aanroepen binnen expressies; gebruik CALL om het resultaat te negeren	Gebruik het CALL statement of laat de haakjes weg en roep alleen aan bij de naam
Parameter doorgeven	Geef door bij kopie	Geef door bij referentie, bij waarde, bij kopie	Geef door bij referentie, bij waarde, bij kopie
Standaard variabelen	SHARED	LOCAL	LOCAL
Array argumenten	Nee	Ja	Ja
Toegankelijk buiten een unit?	Nee	Ja	Ja

Wat betekent dit allemaal? Ten eerste, bekijk dit kleine programma dat de functie *CylVol* definieert en aanroept:

```
FUNCTION CylVol(radius, hoogte)
     CylVol = radius * radius * 3.14159 * hoogte
END FUNCTION

r = 4.7 : h = 12.1
vol = CylVol(r,h)
```

Bekijk nu de run-time verwerking die noodzakelijk is voor het uitvoeren van dit programma. Als eerste kent u waarden toe aan de variabelen *r* en *h*. U roept dan *CylVol* aan en geeft u de numerieke informatie door in *r* en *h*. Maar wacht—hoe verstrekt u deze informatie aan de functie?

Er zijn twee mogelijkheden: (1) gebruik 4.7 als de radius en 12.1 als de hoogte, of (2) gebruik variabele *r* als de radius en variabele *h* als de hoogte.

Constanten als 4.7 kunnen doorgegeven worden bij waarde of bij kopie; variabelen als *r* kunnen doorgegeven worden bij waarde, bij kopie of bij referentie. Laten we eens elk van deze doorgegeven techniek bekijken.

Geef-door-bij waarde betekent: de actuele waarden van de parameters zijn doorgegeven. In het voorbeeld zijn de actuele waarden 4.7 en 12.1 doorgegeven. De procedure of functie *kan niet de originele parameters wijzigen*.

Geef-door-bij kopie betekent: een kopie van elk argument in een aanroep is geplaatst in een tijdelijke opslag. Een pointer verwijst naar zo'n tijdelijke locatie en is dan doorgegeven naar de aangeroepen routine. Zodra de procedure of functie beëindigd is met de uitvoer, zal de tijdelijke opslag verdwijnen; dus *de originele waarden van de parameters blijven ongewijzigd*.

Geef-door-bij referentie betekent: een pointer voor elke parameter die bepaalt waar die doorgegeven parameter opgeslagen is. In het voorbeeld zijn de adressen van de variabelen *r* en *h* doorgegeven als parameters. De aangeroepen routine kan dan *de actuele waarden krijgen en wijzigen*.

PowerBASIC gebruikt al deze drie doorgegeven regelingen; de manieren ervan wordt in een moment duidelijk.

De geef-door-bij waarde en geef-door-bij kopie methoden staan elke constante of expressie toe voor gebruik als een argument. Tijdens de uitvoer wordt de expressie geëvalueerd en beperkt tot een simpele waarde die doorgegeven wordt naar de functie. Bijvoorbeeld:

```
v = CylVol(r, h*2 + 4.1)
```

De geef-door-bij waarde en geef-door-bij kopie hebben geen problemen met het communiceren van *h*2 + 4.1* naar *CylVol*. De expressie is geëvalueerd en het resultaat wordt verstuurd naar de functie.

De geef-door-bij referentie methode, die het manipuleren van de waarde toelaat, verwerkt geen constanten en expressies (een expressie als *h*2 + 4.1* heeft geen geheugenadres—alleen variabelen hebben dat). De geef-door-bij referentie methode werkt alleen wanneer een argument naar een procedure een enkele variabele argument of een hele array is.

Geef-door-bij waarde en geef-door-bij kopie laten variabelen en expressies toe. Geef-door-bij referentie gunt alleen variabelen en arrays.

Het voordeel van geef-door-bij referentie is dat de aangeroepen routine invloed kan hebben op de waarden van de doorgegeven variabelen en daarmee informatie kan teruggeven aan de aanroepfunctie. Aangezien een routine met doorgegeven waarde aan de hand van een adres is opgegeven, weet het waar die variabele wordt gevestigd en is daarom in staat om te lezen en te schrijven. Een routine die een doorgegeven variabele bij waarde heeft kan geen invloed hebben op de originele variabele, omdat het niet weet waar de originele variabele opgeslagen is.

Variabelen die doorgegeven worden bij referentie kunnen gewijzigd worden in een aangeroepen routine; variabelen doorgegeven bij waarde of bij kopie kunnen dat niet.

Bekijk dit programma:

```
a = 0 : b = 2 : c = 3
CALL Toevoegen(a,b,c,totaal)
PRINT a, totaal
END

SUB Toevoegen(i,j,k,som)
     som = i + j + k
END SUB
```

Variabele *totaal* bevat de som van *a*, *b* en *c* zodra *Toevoegen* de som resulteert. *Toevoegen* kon worden gedefinieerd om de waarde van elk argument te wijzigen; echter, omdat alleen de vierde parameter is toegewezen, zal alleen *totaal* worden beïnvloed door de aanroep *Toevoegen*.

DEF FN functie argumenten zijn alleen geef-door-bij kopie; functie en procedure argumenten zijn geef-door-bij referentie, bij kopie en bij waarde.

Enkele variabelen namen bestaan als argumenten in een procedure- of functie aanroep, en deze variabelen kunnen alleen gewijzigd worden wanneer ze doorgegeven worden bij referentie. Er zijn twee manieren om een enkele variabele door te geven, zodat een functie of een procedure niet de waarde ervan kan wijzigen. Als eerste kunt u de variabele tussen haakjes plaatsen. Dit dwingt PowerBASIC het als een expressie te beschouwen en het door te geven bij kopie. Ten tweede, wanneer u de parameterlijst schrijft voor de functie of procedure, moet u het BYVAL sleutelwoord achter de parameternaam plaatsen. Dit verhindert PowerBASIC de parameter door te laten geven bij waarde. Bijvoorbeeld:

```
EenInt% = 7
AndereInt% = 8
PRINT "In het begin, EenInt% = ";EenInt%
PRINT "AndereInt% = ";AndereInt%
GeenVerandering EenInt%, AndereInt%
PRINT "Na de aanroep van GeenVerandering,    EenInt% = ";EenInt%
PRINT "                                   AndereInt% = ";AndereInt%

SUB GeenVerandering(BYVAL Num1 AS INTEGER, BYVAL Num2 AS INTEGER)
     Num1 = 5
     Num2 = 10
     PRINT "In SUB GeenVerandering Num1 = ";Num1
     PRINT "                    en Num2 = ";Num2
END SUB
```

Procedures en functies accepteren ook constanten en expressies als doorgegeven bij waarde. DEF FN functies accepteren constanten, variabelen en expressies, maar ze kunnen niet hun waarden wijzigen, omdat DEF FN functieparameters altijd doorgegeven worden bij kopie.

U bent vrij om toe te kennen aan een formele parameter binnen een DEF FN functiedefinitie. Wellicht is het handig om de DEF FN formele parameters als tijdelijke variabelen te gebruiken. Maar nogmaals, goede programmeer praktijken fronsen hierbij, omdat dan uw programmalogica moeilijker te volgen is. Hierdoor zal echter niet de waarde van de werkelijke parameter gewijzigd worden. Bijvoorbeeld:

```
DEF FNSom(a,b,c)
     a = a + b + c
```

```
     PRINT a
END DEF

x = 1 : y = 2 : z = 3
t = FNSom(x,y,z)
PRINT x
```

De *som* toekenning van de formele parameter *a* beïnvloedt niet de waarde van variabele *x*.

Wanneer een variabele doorgegeven wordt naar een DEF FN functie, veroorzaakt dat een gegevenskopie van die variabele in een lokale opslag. Er ontstaan problemen mocht de variabele erg groot zijn (bijvoorbeeld een array). Daarom staat PowerBASIC niet toe om array variabelen door te geven bij waarde, hoewel u afzonderlijke elementen van matrices kunt doorgeven. Dus hele arrays kunnen alleen doorgegeven worden aan procedures of functies, niet aan DEF FN functies.

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.

Scope

In interpretive BASIC zijn alle variabelen *globaal*. Hiermee wordt bedoeld dat het niet uitmaakt waar een variabele aanwezig is in een programma. De variabelen zijn beschikbaar voor het hele programma, of deze in het belangrijkste hoofdprogramma of in de meest triviale lus in een duistere subroutine staan maakt niet uit. Het begrip van hoe en waar een variabele voor de rest van het programma beschikbaar is heet *scope*. Als u zich niet bewust bent van de scope regels voor de taal, kunnen er bugs inkruipen, zoals in het volgende interpretive BASIC voorbeeld.

```
'   hoofdprogramma
WerknemerAantal = 10
n = 1
GOSUB RekenControle
n = n + 1
GOSUB PrintControle
RekenControle:
GOSUB RekenInhoudingen
RETURN

RekenInhoudingen:
     FOR n = 1 TO WerknemerAantal
          .
          .
          .
     NEXT n
RETURN
PrintControle:
     .
     .
     .
RETURN
```

Omdat alle variabelen in interpretive BASIC globaal zijn, zal variabele *n* in de duistere subroutine *RekenInhoudingen* en variabele *n* in het hoofdprogramma één en dezelfde zijn. Het gevolg is, wanneer na controle terug wordt gegaan naar het hoofdprogramma, zal *n* niet de waarde 1 terugkrijgen maar zal de waarde van *Werknemer* + 1 als gevolg van de FOR/NEXT lus in *RekenInhoudingen* krijgen.

Sommige programmeertalen vereisen dat u variabelen “declareert”. Dat is, u bedenkt welke variabele u wilt gebruiken en wat voor type het moet hebben. Als u dan een vergissing maakt, zal de compiler u dat melden. Interpretive BASIC vereist dit niet, dus een misstap van een vinger kan fouten vermeerderen.

PowerBASIC biedt de optie aan om het declareren van variabelen te vereisen. Zie $DIM in de Help Documentatie van PowerBASIC.

Lokale variabelen

Om dit probleem te voorkomen, ondersteunt PowerBASIC *lokale* variabelen binnen procedures en functies. In tegenstelling tot een globale variabele bestaat een lokale variabele alleen in de routine waar die gedeclareerd is. Zijn scope (effect) is daarom begrensd en heeft het gelukkige resultaat om dergelijke problemen, zoals in het vorige voorbeeld, te beperken. Lokale variabelen zijn een goede reden om sub routines voor eeuwig af te zweren. Bijvoorbeeld, bestudeer de functie *VoegToeRecepten*:

```
DEF FNVoegToeRecepten
     LOCAL x, y, totaal
     FOR x = 1 TO 12
          FOR y = 1 TO 30
               totaal = totaal + recepten(x,y)
          NEXT y
     NEXT x
     FNVoegToeRecepten = totaal
END DEF
```

Nu variabelen *x* en *y* lokaal gedeclareerd worden naar FN*VoegToeRecepten*, kunt u de namen van variabelen *x* en *y* elders in dit programma (programmeurs met een wiskundige achtergrond willen *x* en *y* voor elke variabele) gebruiken zonder dat de waarden van *x* en *y* in FN*VoegToeRecepten* gewijzigd worden (of vice versa).

De lokale variabelen in FN*VoegToeRecepten* bestaan alleen zolang die functie aan het uitvoeren is—voor en na de functie bestaan ze nergens. Om het te illustreren, bekijk dit codegedeelte dat FN*VoegToeRecepten* aanroept:

```
1 x = 35
2 ditJaar = FNVoegToeRecepten
3 PRINT x
```

Het feit dat de naam *x* gegevens is aan andere variabelen in het berekeningsproces met de retourwaarde van FN*VoegToeRecepten*, heeft geen invloed op *x* in regel 1 en 3. Door de magie van de stack allocatie is de *x* in FN*VoegToeRecepten* een afzonderlijke variabele, een die niet bestaat na het resultaat van de functie.

Lokale variabelen moeten voor elke uitvoerbare statements gedeclareerd worden in een procedure of functie. Normaal gesproken worden variabelen die in een procedure of functie aanwezig zijn beschouwd als lokaal, tenzij ze ergens anders gedeclareerd zijn. Zoals variabelen in DEF FN functies die beschouwd worden als gedeeld (shared), dus niet lokaal.

Lokale variabelen mogen ook arrays zijn. Declareer eerst het array lokaal voordat u deze dimensioneert:

De toewijzing van lokale arrays worden automatisch ongedaan gemaakt na het verlaten van een procedure of functie.

```
DEF FNDummy
     LOCAL a, lokArray()
     DIM lokArray(50)
     .
     .
     .
END DEF
```

Lokale variabelen zorgen ervoor dat u kleine geïsoleerde codeblokken kunt bouwen zodat er specifieke functies uitgevoerd kunnen worden. Programma's die deze functies aanroepen hoeven geen zorgen te maken over de functies die deze variabelen wijzigen, behalve de doorgegeven variabelen, maar zelfs dat kan beschermd worden door de parameters door te geven bij waarde of bij kopie in plaats van bij referentie.

Onthoud dat lokale variabelen hun inhoud verliezen zodra de functie of procedure klaar zijn met de uitvoering. Met elke inroep zijn de lokalen schoon gecreëerd en geïnitialiseerd met nul (of null voor strings).

Bijvoorbeeld:

```
SUB Dummy
     PRINT c
     c = c + 1
END SUB
```

U kunt al die tijd *Dummy* aanroepen, maar het zal telkens 0 printen.

Globale variabelen: De SHARED attribuut
In procedures en functies kunnen ook variabelen worden gedeclareerd met het SHARED attribuut. Een shared (of gedeelde) variabele is het tegenovergestelde van een lokale variabele: het is zichtbaar en bruikbaar in de rest van het programma. Sommige variabelen worden al vaak genoemd als globale variabelen.

```
FUNCTION Six
     SHARED a
     a = 6
     Six = a
END FUNCTION

PRINT Six, a
```

Omdat de SHARED declaratie is gebruikt, is de variabele *a* in *Six* en de variabele *a* in het PRINT statement hetzelfde.

Niet gedeclareerde variabelen binnen DEF FN functiedefinities zijn standaard als SHARED; als voorbeeld in de DEF FN functie FN*VoegToeRecepten* zoals eerder getoond, is het array *recepten* een shared variabele. Binnen een procedure- en functiedefinitie is het standaard LOCAL. Het is zeer aan te bevelen om elke variabele expliciet te declareren die in een procedure- of functiedefinitie aanwezig is.
Puristen zullen beweren dat er geen globale variabelen moeten worden gebruikt (dat is een belangrij-

ke reden voor puristen te spotten op BASIC). Dit kan echter leiden tot extreem lange parameter lijsten (misschien meer dan de limiet van 16 voor elke routine in PowerBASIC), maar in ieder geval waardoor het moeilijker wordt om het in een programma te lezen. Voorzichtig gedaan kunnen globalen nuttig zijn in het modulair maken van programma's. In uw hulpfuncties die gebruikt zullen worden in vele toepassingen, moet u echter proberen eventuele globale variabelen te voorkomen en in plaats daarvan toegang krijgen tot alle variabelen in de parameter lijsten.

Statische variabelen
Statische variabelen zijn een kruising tussen lokale en gedeelde variabelen. Zoals een lokale variabele zal een statische variabele zich niet mengen met andere variabelen die dezelfde naam hebben—het is alleen zichtbaar binnen de procedure of functie die het declareert. Zoals een gedeelde variabele neemt een statische variabele permanent geheugen in beslag en daarom verliest deze niet zijn waarde tussen de aanroep van de functie of procedure. Het is geïnitialiseerd met nul of null (in het geval van strings), maar alleen wanneer het programma begint.

```
SUB Dummy
     STATIC i
     INCR i
     PRINT i
END SUB

i = 16
CALL Dummy
CALL Dummy
PRINT i
```

De uitvoer is:

```
1
2
16
```

De statische variabele *i* in *Dummy* is verschillend van variabele *i* in het hoofdprogramma. Anders dan een lokale variabele behoudt het echter zijn waarde tussen de aanroepingen van de omsluitende procedure. Het begint met 0, zoals elke variabele, en is tweemaal opgeteld bij de twee *Dummy* aanroepen.

Met het passend gebruik van argumenten en lokale variabelen kunnen procedures en functies totaal onafhankelijk gemaakt worden in de programma's waar zij worden weergegeven. Met het gebruik van het PowerBASIC $INCLUDE metastatement en gescheiden compilatie, kunnen deze procedures en functies moeiteloos in nieuwe programma's opgenomen worden.

De volgende keer in het laatste deel over modulair programmeren, laat ik u meer zien wat we met procedures en functies kunnen doen. Een speciale BASIC techniek is *recursief*. Ook komt het gebruik van *units* aan de orde om te weten te komen hoe we procedures en functies onafhankelijk kunnen compileren. De sleutelwoorden PUBLIC en EXTERNAL vertellen over wat voor een karakter we de variabelen kunnen geven.

Lusstructuren – Nader bekeken.

De beslissingsstructuren, waar ik het de vorige keer over had, zijn niet de enige structuren die het programmaverloop veranderen. Een ander soort structuur die ook het verloop drastisch kan veranderen hoeft niet speciaal een beslissing te maken. Zonder dat er wat bepaald wordt of iets waar is, wordt de structuur al uitgevoerd. Deze structuren worden *lusstructuren* of ook wel *herhalingsstructuren,* genoemd.

Waarom gebruiken we een lus?

We gebruiken een lus als we een opdracht of een reeks opdrachten meerdere malen uit willen voeren, en dat hoeven niet speciaal steeds dezelfde te zijn. Daar bedoel ik mee, misschien moet er een lijst gevuld worden of een bestand ingelezen worden. De opdrachten zijn natuurlijk wel steeds dezelfde die in een lus uitgevoerd worden, maar ze zullen telkens in de herhaling een ander doel hebben.

Verschillende soorten lussen.

Er zijn verschillende soorten lussen. We kennen er één die alleen maar kan werken als een herhaling – tot het einde van de hoogste of de laagste teller waarde. Andere lussen kunnen een combinatie zijn tussen een herhaling en een beslissing. De beslissing wordt niet bepaald door een IF … THEN statement om er dan uit te gaan springen – nee, dit kan de lus zelf doen. We moeten niet gebruik maken van een FOR … NEXT lus. Deze lus kan niet bepalen of een bepaalde conditie waar is of niet. Laten we eens de verschillende lussen onder de loep nemen.

FOR … NEXT lus

Met deze lus kunnen we een blok code een aantal keren laten herhalen. Het herhalen wordt gedaan door een beginwaarde en een eindwaarde op te geven. De waarde moet echter wel met een variabele worden gedaan. Bijvoorbeeld:

```
FOR i = 1 TO 10
      .
      .
      .
NEXT i
```

De code tussen FOR en NEXT herhaalt het PRINT statement 10 keer, maar onderstaand voorbeeld doet dat ook:

```
FOR i = 1 TO 19 STEP 2
     PRINT i
NEXT i
PRINT i
```

De uitvoer zal verrassend zijn:

```
1
3
5
7
9
11
13
15
17
```

```
19
21
```

Zodra de lus uitgevoerd is, zal de lusteller na NEXT zijn laatste verhoging krijgen. Vandaar dat het PRINT statement na NEXT de waarde 21 weergeeft.

Het STEP statement is optioneel en voor stappen van 1 hoeven we STEP dus niet te gebruiken. We kunnen met de FOR … NEXT lus ook omlaag tellen. Het STEP statement moet dan wel altijd gebruikt worden, ongeacht we met stappen van 1 omlaag willen tellen.

```
FOR i = 10 TO 1 STEP -1
     .
     .
     .
NEXT i
```

Vergeet u het STEP statement, dan zal de lus niets doen.

WHILE … WEND of WHILE … END WHILE
Er zou in de programmeerwereld geen flexibele structuur kunnen bestaan als de lussen zelf niet zouden kunnen beslissen hoe lang er herhaald mag worden. Een ervan is een zeer algemene lus die wel in de meeste BASIC dialecten bestaat, maar toch in bepaalde Basic versies, zoals in de .NET versie, ook anders wordt genoemd.

Commodore was de eerste die deze WHILE … WEND lus uitbracht in BASIC 7.0 en de lus bestaat nog steeds (met een lange baard). Anders dan bij de FOR … NEXT lus, moeten we de lusteller zelf verhogen of verlagen, daarom is de WHILE … WEND lus geen herhalingslus maar een beslissingslus. Een voorbeeld:

```
i = 1
WHILE i <= 19
     PRINT i
     i = i + 2
WEND
PRINT i
```

De uitvoer zal hetzelfde zijn als bij de FOR … NEXT lus die STEP 2 gebruikt. Ook zal de waarde 21 weergegeven worden. Nadat de conditie gecontroleerd is door WHILE, zal de verhoging hoger zijn dan de gecontroleerde waarde. Alleen de waarde 21 zal voor een beëindiging van de lus zorgen. Om het probleem met de te hoge waarde 21 te voorkomen, zouden we de conditie onderaan moeten hebben. Maar met de WHILE … WEND lus kan dat niet.

Wees niet bang om de WHILE … END WHILE. Deze lus is de vernieuwde WHILE die alleen werkt in Visual Basic .NET. Alle gewone BASIC dialecten gebruiken de standaard WHILE … WEND structuur.

DO … LOOP en UNTIL
De meest krachtigste beslissingslus is de DO … LOOP lus. Commodore bracht ook deze uit in BASIC 7.0 en is ook een overgebleven BASICA lus, net als de WHILE … WEND lus.

We kunnen DO … LOOP in verschillende manieren gebruiken. Het UNTIL statement kunnen we bovenaan of onderaan gebruiken en bepaald wanneer de lus beëindigd moet worden. Het is aan u hoe de lus moet werken.

```
i = 1
DO UNTIL i > 10
     PRINT i
     i = i + 1
LOOP
```

Anders dan bij de WHILE lus, moet nu een conditie worden bepaald of variabele i > 10. Het UNTIL statement is het tegenovergestelde van WHILE. Zouden we dezelfde conditie gebruiken, dan zal de lus niets doen.

Wat we niet kunnen doen bij een WHILE lus, kan wel bij een DO … LOOP lus: onderaan controleren.

```
i = 1
DO
     PRINT i
     i = i + 1
LOOP UNTIL i > 10
```

We zien nu ook direct de oplossing met het probleem van de laatste ophoging wat de FOR … NEXT lus deed. Door UNTIL onderaan te gebruiken en de teller voor het PRINT statement te plaatsen, zal de waarde 21 niet meer weergegeven worden.

```
i = -1
DO
     i = i + 2
     PRINT i
LOOP UNTIL i >= 19
```

Na de lus uitvoer zal de waarde van variabele i nog steeds 19 zijn.

Maar er zullen vast wel meer oplossingen zijn om zulke schoonheidsfoutjes te vermijden!

Hoewel de DO … LOOP lus ook zonder een UNTIL statement kan werken, is het toch af te raden. Gebruik liever maar wel het UNTIL statement. De DO … LOOP lus kan zelf niet bepalen wanneer het beëindigd moet worden, want er is dan geen conditie aanwezig. Het gebruik van een IF … THEN statement is dan het enige redmiddel. Echt een redmiddel mocht het niet anders kunnen!

Het ligt helemaal aan het BASIC dialect hoe u uit de lus kunt stappen. Tegenwoordig kan dat makkelijk met een EXIT DO statement en hoeven we het niet meer te doen met een GOTO statement. Dit geldt voor alle lussen. Elke lus heeft een EXIT om eruit te kunnen.

Lussen nesten.

In elke programmeertaal is het toegestaan structuren te nesten, zoals bij de beslissingsstructuren. Het kan ook bij de lusstructuren. We moeten goed opletten welk NEXT statement, WEND of END WHILE statement of LOOP met een eventueel UNTIL statement bij de juiste lus hoort. Inspringen is dan ook het belangrijkste van het programmeren.

In interpretive BASIC was het nesten van structuren een lastige opgave, want het was onmogelijk om in te springen. Ten eerste gebruikten programmeurs een hulpmiddel, bijvoorbeeld:

```
10 FOR N = 1 TO 50
20 : FOR M = 40 TO 0 STEP -2
30 : : PRINT N * M
```

```
40 : NEXT M
50 : A$ = INKEY$ : IF A$ = "" THEN 50
60 NEXT N
```

Zonder in te springen hadden we gauw de neiging om bijvoorbeeld de twee NEXT statements om te keren. Lussen mogen elkaar niet kruisen.

Ten tweede gebruikten programmeurs altijd de variabele achter NEXT die optioneel is. Hoewel dat nu niet meer nodig is omdat BASIC veel gestructureerder is dan toen, is het toch aan te bevelen om de variabele erachter te plaatsen. Maak er een gewoonte van!

Bij een WHILE … WEND (END WHILE) lus en een DO … LOOP met UNTIL lus, wordt vaak commentaar geplaatst achter het statement waar geen conditie staat, bijvoorbeeld:

```
WHILE conditie
      .
      .
      .
END WHILE       ' conditie

DO    ' conditie
      .
      .
      .
LOOP UNTIL conditie

DO UNTIL conditie
      .
      .
      .
LOOP       ' conditie
```

Op die manier weet men altijd wat de lus aan het doen is.

We mogen alle soorten lussen met elkaar nesten, maar pas op dat u niet zelf uw programma in de nesten werkt. Moet er genest worden, zorg er dan voor dat u eerst de binnenste structuur veilig stelt door bijvoorbeeld de code in een procedure of functie te plaatsen. U hoeft deze dan alleen maar in de buitenste lus het aan te roepen.

Geavanceerde kenmerken.

Conditie gestuurde lussen

De lussen die ik steeds beslissingslussen heb genoemd, kunnen we ook *conditie gestuurde lussen* noemen. Ze hebben niet voor niets te maken met condities.

Zoals we eerder zagen, kunnen we de conditie bij een DO … LOOP UNTIL onderaan gebruiken. Willen we echter niet dat de lus eenmaal doorlopen wordt, dan moeten we de UNTIL achter DO gebruiken.

Er is echter nog een DO … LOOP lus, namelijk niet met een UNTIL maar met een WHILE. Ook die manier bestond al in BASIC 7.0. Net als met UNTIL kan een WHILE zowel naast DO als naast LOOP worden geplaatst. De WHILE blijft functioneren als een ZOLANG, dus staat het statement naast DO, dan zal de lus zich gedragen als een WHILE … WEND.

Ook deze kenmerken kunnen het programmeren bemoeilijken. Niet voor niets is programmeren lastig, alleen maar omdat er zoveel mogelijkheden zijn in de structuren. Dus ook de code in uw programma kan op meerdere wegen naar Rome leiden.

Verzameling gestuurde lussen
Deze lussen zijn bedoeld om het uitvoeringsblok voor ieder element (van een bepaald type) van een verzameling te doorlopen. Helaas ondersteunen niet alle BASIC dialecten deze lusstructuur. In Visual Basic 6.0, Visual Basic for Applications en Visual Basic .NET komt deze lusstructuur wel voor.

De structuur ziet er als volgt uit:

```
Dim K As Kat
For Each K In MijnKattenVerzameling
     .
     .
     .
Next K
```

Hier wordt geen To statement gebruikt. Elk object wordt uit de verzameling gehaald zolang het einde van de verzameling nog niet is bereikt. We kunnen dit vergelijken met een boek dat doorgebladerd wordt totdat de laatste bladzijde bekeken is.

Elk item van de verzameling wordt toegekend aan objectvariabele K. Zo kunnen we bepalen of we de juiste kat hebben. Is de kleur wit of zwart?

Recursie als alternatief voor een lus
Als een functie zichzelf herhaaldelijk aanroept, wordt dit *recursie* genoemd. Op deze manier zijn veel zaken die met een lus mogelijk zijn ook te verwerkelijken. Een voorbeeld is een som van alle getallen tot getal.

```
1 + 2 + .... + getal
```

Een voorbeeld van een recursieve functie en een lus die hetzelfde doet:

```
Function TelOpTot(ByVal getal As Integer) As Integer
     If getal > 0 Then
          TelOpTot = getal + TelOpTot(getal - 1)
     End If
     TelOpTot = 0
End Function
```

In Visual Basic .NET ziet de functie er zo uit:

```
Function TelOpTot(ByVal getal As Integer) As Integer
     If getal > 0 Then
          Return (getal + TelOpTot(getal - 1))
     End If
     Return 0
End Function
```

De loop ziet er zo uit:

```
Function TelOpTot(ByVal getal As Integer) As Integer
     Dim resultaat As Integer = 0
     ' Voor andere BASIC dialecten: ken de waarde 0 na de Dim regel toe
     While getal > 0
          resultaat += getal
          ' Voor andere BASIC dialecten: resultaat = resultaat + getal
          getal -= 1
          ' Voor andere BASIC dialecten: getal = getal - 1
     End While ' Andere BASIC dialecten: Wend
     Return 0
End Function
```

Oneindige lussen
Sommige lussen worden (onbedoeld) oneindig uitgevoerd. Dit soort lussen kan ontstaan door het ontbreken van een voorwaarde om de lus te beëindigen of een voorwaarde waar nooit aan voldaan wordt (of kan worden). Een recursie die niet termineert kan ook een reden zijn waarom een oneindige lus ontstaat.

Als u de beslissingsstructuren en de lusstructuren goed beheerst en als het nodig is goed combineert, dan zal uw programma veel beter werken.

Grafisch programmeren in Basic.

Er zijn nog meer mogelijkheden te ontdekken in de grafische wereld. Wat zagen we toch mooie grafische mogelijkheden in GW-BASIC. Net als de vorige keer ben ik verder gegaan in Visual Basic 2010. De code is ook te gebruiken in Visual Basic 2005 en 2008.

Lijnen en cirkels.
We kunnen met dezelfde wiskundige berekeningen met lijnen en cirkels verschillende weergaven maken. Onderstaande code heeft twee soorten grafische Draw regels. Voer ze niet alle twee tegelijk uit, maar zet er elk een in commentaar. Dit om het overlappen van het tekenen te voorkomen.

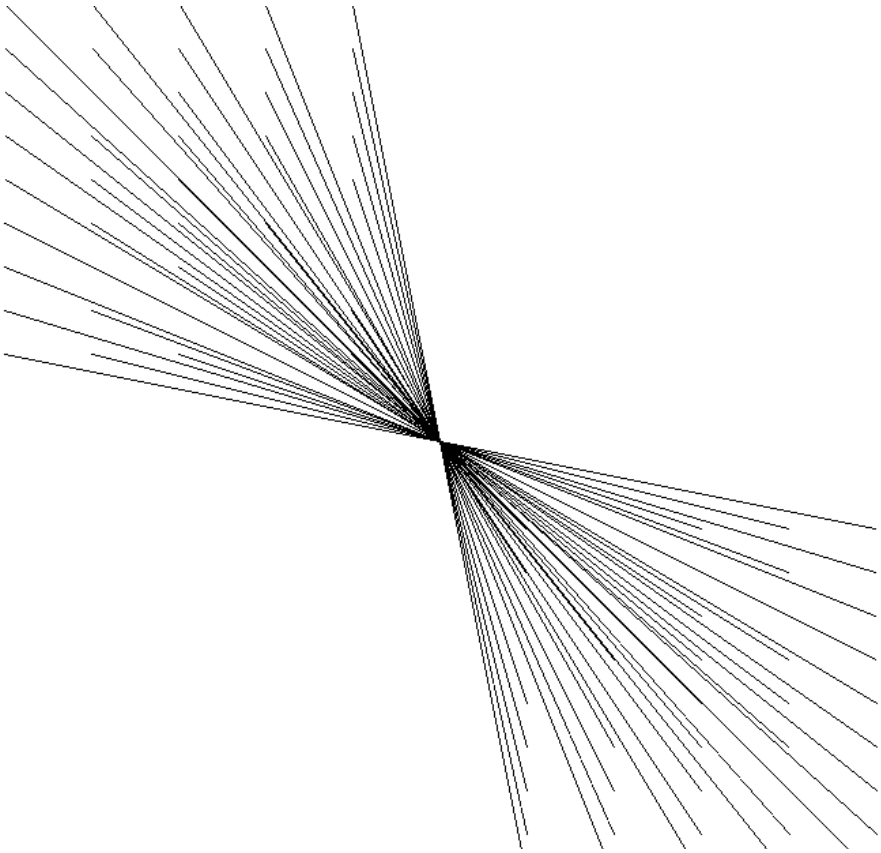
```
Private Sub Form1_Paint(…, ByVal e As ...PaintEventArgs) Handles …
     Dim U As Integer = 160, V As Integer = 160
     Dim H As Double = 0.5

     For N As Integer = 64 To 360 Step 64
          For M As Integer = 64 To 320 Step 32
               Dim W As Double = Math.PI / (N + M) / 2
               Dim X1 As Integer = Int(U - V * Math.Sin(W) - H)
               Dim Y1 As Integer = Int(U + V * Math.Cos(W) - H)
               e.Graphics.DrawLine(Pens.Black, _
                                   (320 + X1) + N, Y1 + M, _
                                   (320 + X1) - N, Y1 - M)
               'e.Graphics.DrawEllipse(Pens.Black, _
                                       X1 + N, (32 + Y1) - M, N, M)
          Next
     Next
End Sub
```

De eerste tekening is gedaan met de methode DrawLine. De tweede tekening is gedaan met de methode DrawEllipse. Merk op dat de variabelen niet alleen gegeven worden om van plaats te veranderen samen met X1 en Y1, maar ook om de breedte en hoogte van de ellips te gebruiken. Daarmee krijgen we het effect dat de ellips begint met een smalle en eindigt met een brede. U kunt ook de breedte en hoogte in nieuwe Sin en Cos functies laten rekenen voor meer leuke effecten.

Misschien vraagt u zich af waarom ik de X-as met DrawLine laat beginnen met 320 + X1. De reden is dat anders te dicht langs de kant van het formulier getekend zal worden. Het is leuker om de tekening een beetje er vanaf te laten beginnen, een soort marge.

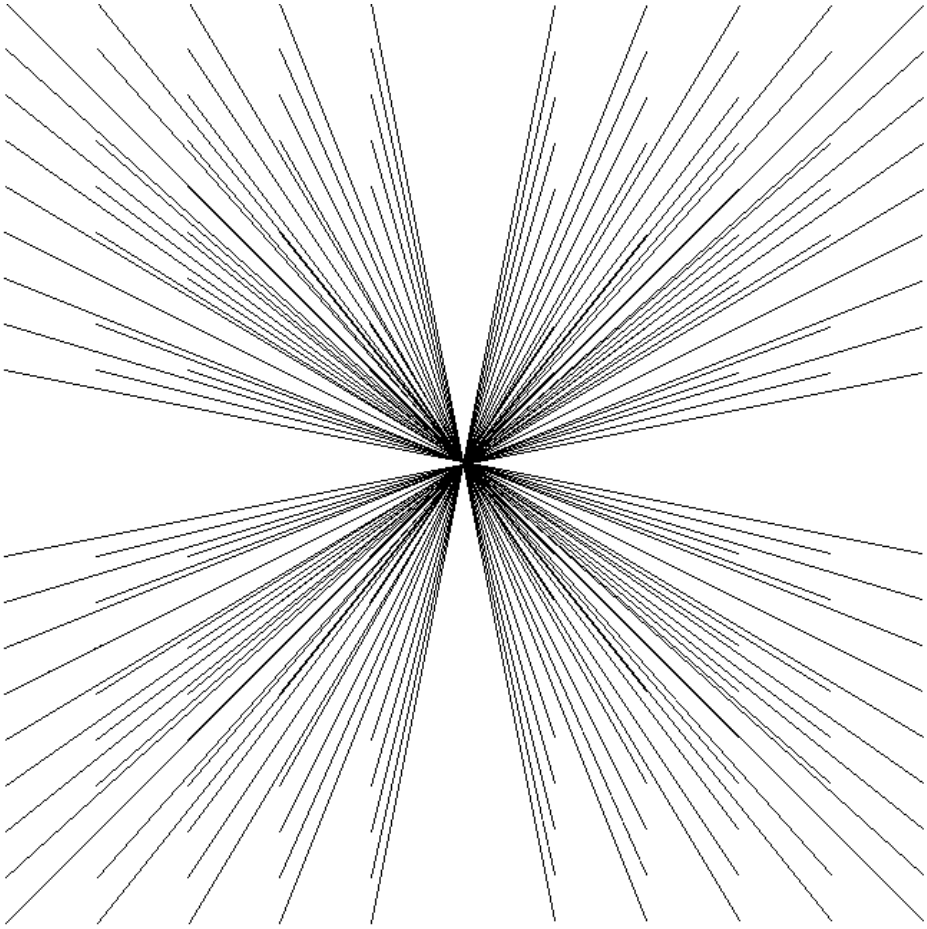
Nogmaals, vergeet niet voor de tweede tekening het commentaar symbool voor de methode DrawLine te plaatsen en die voor de methode DrawEllipse te verwijderen.Ook deze lijnen geven een spiegelbeeld. Ze worden gedraaid en vanuit een bepaalde hoek getekend.



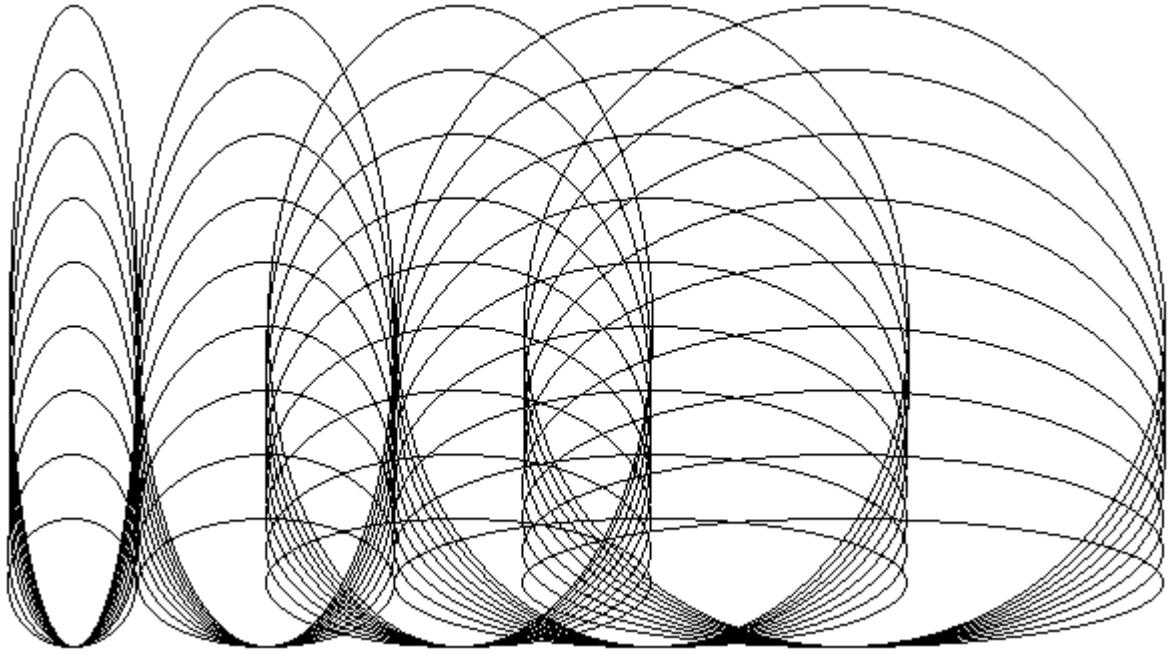
We kunnen nog zo'n spiegelbeeld maken, maar dan linksonder laten beginnen naar rechtsboven. Om dat te kunnen doen, hebben we een tweede DrawLine methode nodig.

We hoeven ook nu geen nieuwe berekeningen te maken. Voeg maar eens onderstaande DrawLine methode in de code toe, onder de andere DrawLine methode en de uitvoer is verbluffend!

```
e.Graphics.DrawLine(Pens.Black, (320 + X1) + N, 640 - (Y1 + M), (320 + X1) - N,
640 - (Y1 - M))
```



Als we de DrawEllipse methode laten tekenen, krijgen we onderstaande tekening:



Het is alsof meerdere methoden elkaar ‘belagen’ terwijl we weten dat maar één methode de tekening maakt. Het sluiten van de ellipsranden zorgt voor een leuke onderkant. Doordat we als breedte variabele N gebruiken, wordt de ellips steeds breder.

De vragen zijn nu: waarom tekenen de ellipsen over elkaar terwijl we dezelfde N waarde gebruiken, en hoe komt het dat toch de ellipsen op de juiste manier op hun plaats komen? Hoewel dit geen voorbeeld is dat we dagelijks zullen gebruiken, geeft dit wel de indruk hoe de grafische methoden in de wiskundige functies en lusstructuren zich gedragen.

De eerste vraag die ik beantwoord is als volgt: u ziet dat het startpunt van de X-as steeds X1 + N is. Dat heeft natuurlijk een hogere waarde dan N zelf heeft. Uit de breedte wordt het middelpunt berekend, zodat de juiste ellips getekend wordt.

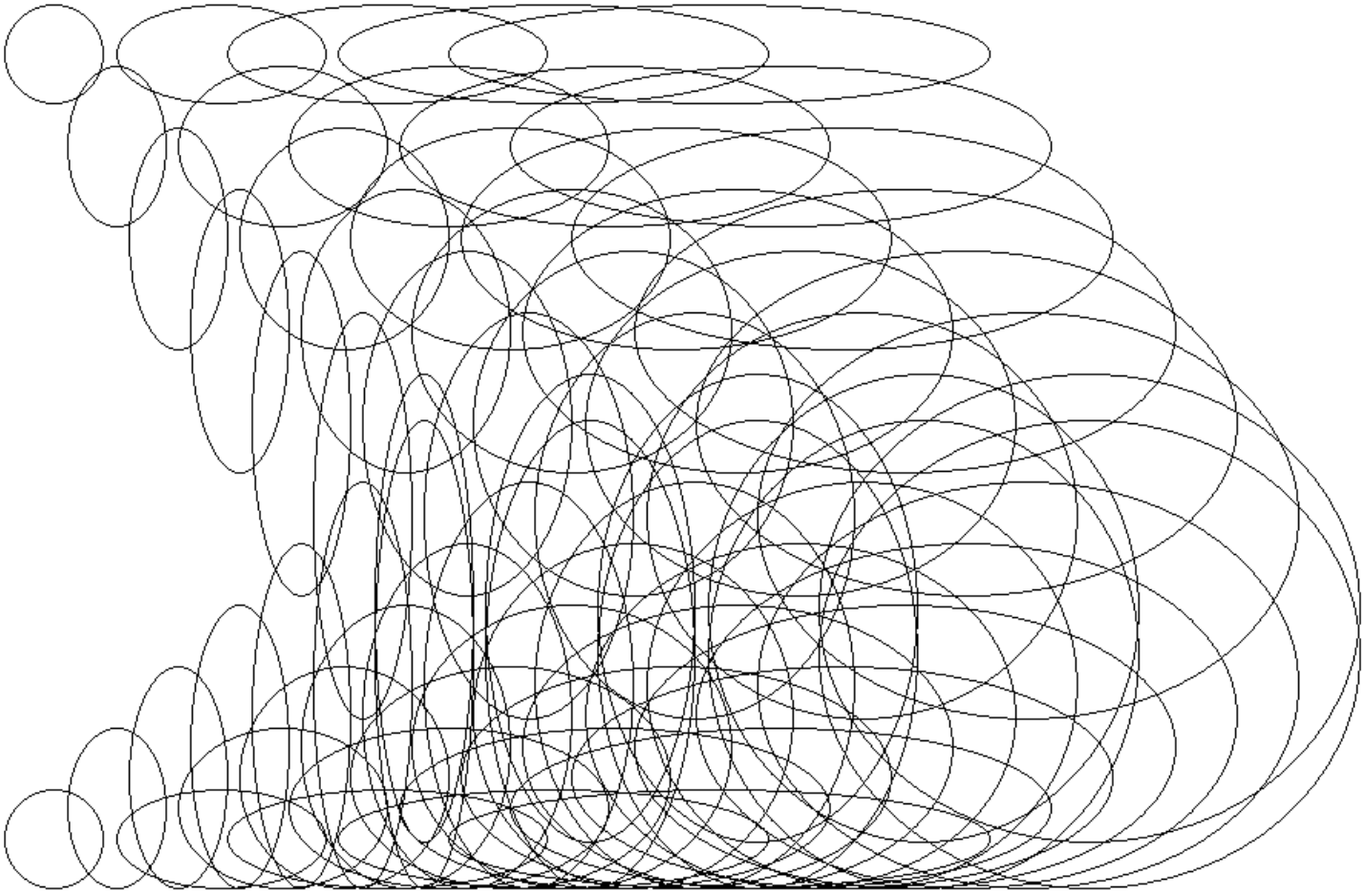
De tweede vraag die ik beantwoord is: relatieve berekeningen. Dit houdt in dat telkens het middelpunt genomen wordt om bij de volgende herhaling van daaruit verder te tekenen. Alleen: niet helemaal het middelpunt, want er wordt absoluut X1 bij opgeteld als coördinaat. Maar om bij N te blijven, zal er altijd voor gezorgd worden dat de volgende kolom met ellipsen binnen of tegen de vorige kolom blijft.

Zouden we elke kolom een eigen kleur geven, dan is het effect helemaal duidelijk.

Zet de tekenregels weer in commentaar en voeg eens onderstaande twee regels toe:

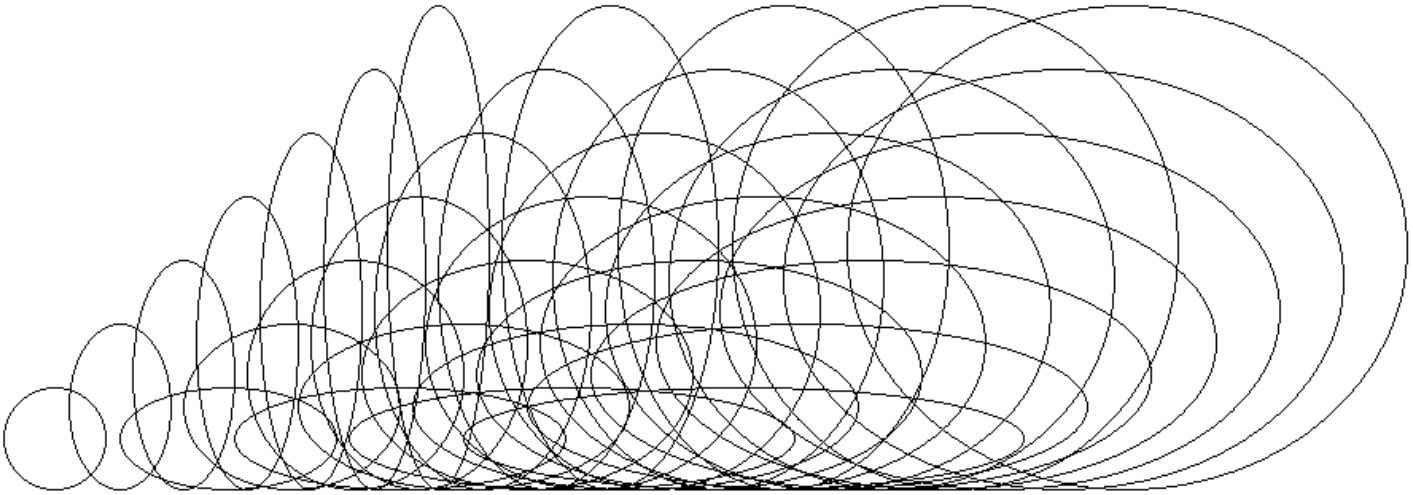
```
e.Graphics.DrawEllipse(Pens.Black, X1 + (N + M), 320 + (32 + Y1) - M, N, M)
e.Graphics.DrawEllipse(Pens.Black, X1 + (N + M), 320 + (32 - Y1) + M, N, M)
```

Nu krijgen we een heel ander soort effect. Nee, het is geen rommeltje, want kijk eens goed. De ellipsen die hier getekend zijn, zijn zowel staand als liggend en gaan trapsgewijs omhoog naar rechts en tegenovergesteld ook omlaag naar rechts.



Hoe hoger we de stappen nemen, achter STEP, hoe meer te zien is wat er gebeurd. We zullen dan wel minder ellipsen zien. Zet eens een van de regels in commentaar en kijk dan eens hoe het getekend wordt.

Onderstaande tekening is bijvoorbeeld alleen met de eerste regel getekend:



Nu kunt u zien dat de ellipsen alleen in het midden van het formulier elkaar overlappen. Ze tekenen niet door naar boven. Ook de tweede regel tekent niet door naar beneden, maar stopt ook in het midden.

Kijk nog eens naar de volledige tekening. De grootste ellips aan de rechterkant zou haast bovenaan horen, ook aan de rechterkant. Maar dat is dus niet zo.

Parameters en argumenten.

Bij het aanroepen van procedures en functies geven we vaak waarden mee. Dat kunnen variabelen, constanten en expressies zijn. Elke parameter heeft een bepaald karakter dat met het argument overeen moet komen.

Parameter -> Argument:	Het argument krijgt de parameter
Parameter <- -> Argument:	Het argument krijgt de parameter, maar de parameter krijgt het ook weer terug.

Voorbeelden zijn:

`N = A`	Het argument krijgt N bij referentie.
`N = ByVal A`	Het argument krijgt N bij kopie.
`2 = ByVal A`	Het argument krijgt 2 bij waarde.

Ook in het onderwerp van PowerBASIC staat er dat BASIC drie soorten doorgeef methoden kent. Zelf vind ik dat niet. De laatste twee methoden, bij kopie en bij waarde, werken op dezelfde manier. Zouden we echter bij de laatste doorgeef methode het sleutelwoord ByVal vergeten, dan wordt er een bij referentie geprobeerd en dat werkt niet.

Waarom noemt men het bij kopie als er een variabele als parameter wordt doorgegeven? Het antwoord heeft te maken met wat er precies gebeurt zodra er doorgegeven wordt, want de argumenten krijgen niet zo maar de inhoud van de parameters, zeker niet wanneer het een bij referentie methode is.

BASIC verbergt iets, wat een C programmeur dagelijks tegenkomt: pointers. Een pointer is de sleutel van de deur om het adres van de parameter door te geven aan het argument. Dit is alleen nodig als we een bij referentie doorgeven. Die andere twee hebben die sleutel niet nodig, omdat er een kopie

van de parameterwaarde wordt gemaakt. Het argument heeft zijn eigen adres met de waarde van de parameter. Vandaar dat het argument de waarde verliest zodra de procedure of functie beëindigd wordt.

Als we de waarde van het argument wijzigen en het argument is gedefinieerd als een bij referentie, zal de parameter ook de gewijzigde waarde krijgen, omdat ze beide hetzelfde adres hebben. Dat is het werk van pointers.

Gelukkig hoeven wij in BASIC niet het pointer '*' symbool en het adres '&' symbool te gebruiken als we parameters en argumenten nodig hebben. Bij objecten, zoals klasobjecten, moeten in C++ ook pointers worden gebruikt. Die manier van instanties creëren wordt *allocatie* genoemd.

In BASIC maken we ook wel eens instanties aan, zoals onderstaande declaratie:

```
Dim objKat As New Kat
```

of

```
Dim objKas As Kat
objKat = New Kat
```

Of op de oude manier:

```
Dim objKat As Kat
Set objKat = New Kat
```

Het Kat type moet echter wel een *objectklasse* zijn, niet een gewoon type.

Nog even terug: pointers
De vraag is: wie is de pointer en wie heeft dan het adres?

Het argument zal de pointer zijn. De parameter is het adres. Logisch, want de waarde zit opgeborgen in het adres en het argument moet het ontvangen. We kunnen ook zeggen: het argument moet als adres zijnde *verwijzen* naar de parameter. Dit komt alleen voor als we een bij referentie willen doorgeven. In andere situaties moeten we in BASIC het sleutelwoord ByVal opgeven. In C++ is dat sleutelwoord er niet; geen pointer en adres symbolen? Dan is er ook geen referentie, maar automatisch een bij kopie of een bij waarde methode.

Onderstaande functie is gemaakt:

```
Function TestPar(ByVal A As Integer, _
                    B As Integer, ByRef C As Byte) As Integer
     A = 50
     B = B - 10
     If B > 0 Then C = 1 Else C = 0
     TestPar = 0
End Function
```

Bekijk onderstaand tabel die aangeeft wat er gebeurd is na de aanroep van de functie.

`TestPar(10,10,5)`	Werkt niet! Argument B is niet ByVal. Standaard is een argument altijd ByRef als u geen doorgeef sleutelwoord opgeeft. U kunt geen waarde toekennen aan een waarde. Dit geldt ook voor de derde parameter.

`TestPar(N,I,V)`	Werkt! Argument A krijgt de waarde van N, maar verliest deze na de aanroep. Ook de waarde 50 gaat verloren. Als de waarde van argument B groter is dan 0, krijgt parameter V de waarde 1 van argument C, anders de waarde 0 van argument C. Parameter I krijgt ook de gewijzigde waarde van argument B.
`TestPar(X+1,Y,Z-10)`	Werkt niet! Ook al gebruikt u variabele Z als parameter, zodra er een berekening in de parameter plaatsvindt, zal het ByRef argument C niet werken. Expressies zijn gewoon niet toegestaan.

Een ander BASIC dialect – BBC BASIC.

Wie BBC BASIC niet kent kan beginnen met een leuk dialect. BBC BASIC is een programmeertaal voor Windows om technische applicaties te schrijven, maar ook om games te maken op een makkelijke manier.

In de oude tijd

De vraag is hoe ik aan BBC BASIC kom. Nou, dat kan ik u wel vertellen. In begin jaren ’90 werkte ik in een groot lokaal, genaamd ‘lokaal 9’. Het was een robotica lokaal waar verschillende CMC machines en robotarmen geprogrammeerd werden. Naast de 5de generatietalen Scara en Asea, werd ook BBC BASIC gebruikt om robotarmen te programmeren.

We kunnen BBC BASIC geen BASICA dialect noemen, omdat er in de oude tijd al gestructureerde statements waren. Een voorbeeld is de handige PROC … ENDPROC die heel erg op SUB … END SUB lijkt. Dat zelfs in die tijd op die manier geprogrammeerd kon worden zonder last te hebben met GOTO statements was toen voor mij een wonder. Ik leerde toen ook al op de MTS de programmeertaal Pascal en ik dacht: BASIC is niet zoals Pascal. Maar ik heb dus ondervonden dat BBC BASIC al heel lang bestaat als een goed gestructureerde programmeertaal.

Mogelijkheden in Visual Basic .NET

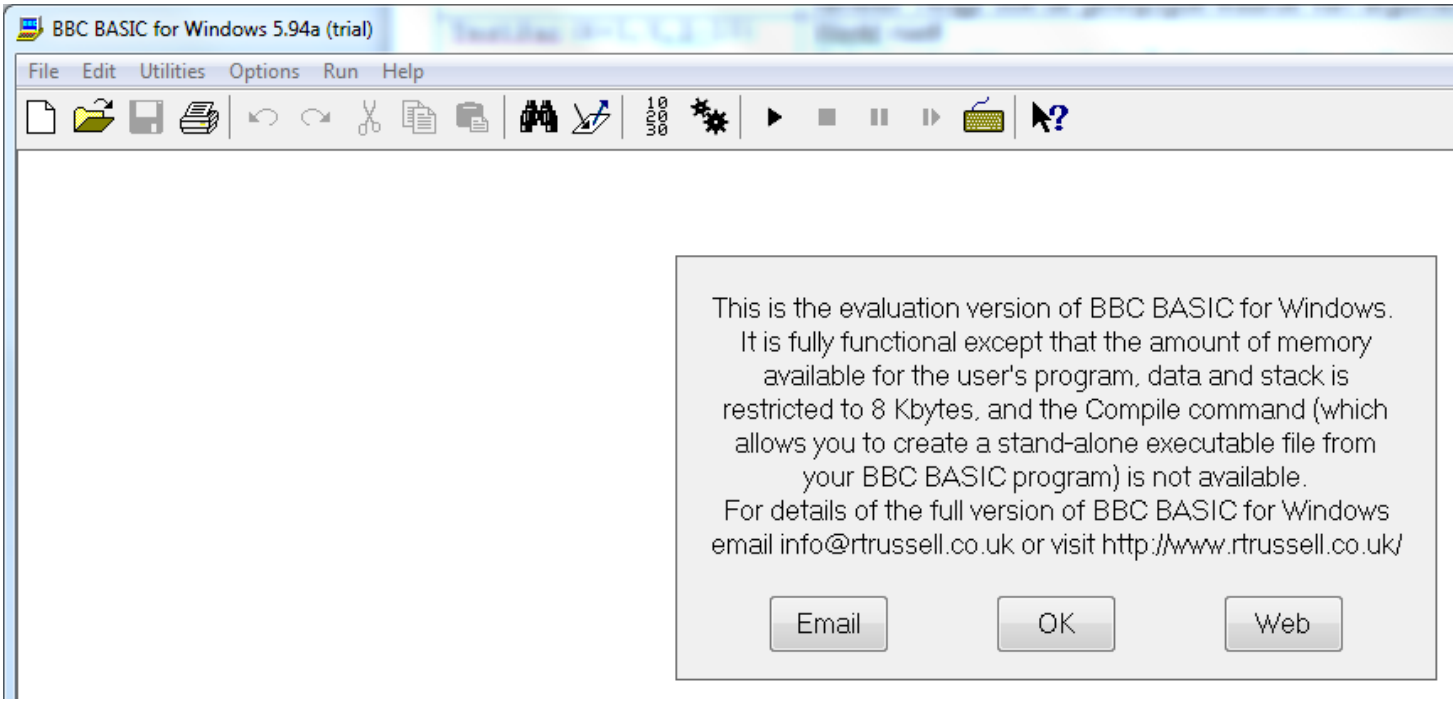
Wat we nooit in BASIC konden gebruiken en nu pas wel vanaf 2003 in de .NET versies, de C++ operatoren +=, -=, bestonden zelfs al in BBC BASIC. De manier hoe de code eruit ziet lijkt ook helemaal niet op BASIC. Het is dus even wennen als u de code voor het eerst ziet. Toch zal ik het de komende tijd uitleggen hoe de versie werkt, want het is gewoon een BASIC taal.

Beginnen met BBC BASIC

BBC BASIC kan gewoon van Internet worden gedownload. U kunt, om er eerst mee te oefen en de taal te leren, eerst de trial nemen. Die is helemaal gratis, maar u hebt dan wel een ruimte van maar 8KB en de optie om uw programma te compileren als een uitvoerbaar programma is dan uitgeschakeld.

De evaluatieversie, de trial, begint dan ook met een mededeling zoals hieronder.

U ziet ook gelijk het venster van BBC BASIC. Het witte gedeelte is de editor. BBC BASIC heeft geen IDE en om echte Windows onderdelen te kunnen gebruiken, moeten we de library’s toevoegen in de code. Dat is nog de oude gewoonte van de versie die nog overgebleven is. De manier zoals we vroeger ook BBC BASIC moesten gebruiken is dus niet veranderd.



Klik op OK en laten we beginnen.

BBC BASIC kent net als de andere BASIC dialecten het PRINT statement.

```
PRINT 52
```

Dit drukt de waarde 52 af als we dit starten. Klik op het driehoekje op de knoppenbalk bovenaan.

Niks anders dus en dat geldt ook voor tekst en expressies.

We zouden meerdere lege regels kunnen weergeven. Gewoonlijk zouden we dit doen:

```
PRINT : PRINT : PRINT
```

Dat mag ook in BBC BASIC, maar er is een kortere manier.

```
PRINT '''
```

Door gebruik te maken van enkele aanhalingstekens (quotes), kunt u lege regels weergeven. U kunt het ook met tekst samen gebruiken.

```
PRINT "Hallo" '' "Wereld"
```

We kunnen ook een bepaalde afstand nemen om vanaf de kant tekst weer te geven.

```
PRINT "Waarde = ", 20
```

Of met de TAB die we wel kennen:

```
PRINT TAB(10);"Hallo"
```

Maar de TAB in BBC BASIC heeft nog een parameter voor de y-coördinaat.

```
PRINT TAB(10,20);"Hallo"
```

De eerste parameter is de kolom (x-as) en de tweede parameter is de rij (y-as). Omdat de grid met (0,0) begint, zal de tekst op (11,21) worden weergegeven. Elk vakje is een karakter. Zou de tekst te

lang zijn, dan zal de rest op de volgende regel worden weergeven. Wat eronder zou staan wordt naar onder gescrold.

In BBC BASIC kunnen we lange regels splitsen, net als in andere BASIC dialecten. We kunnen niet gebruik maken van de onderlijning '_' symbool. We moeten schuine lijnen gebruiken.

```
REM Splitting long lines
PRINT "This is a very, very, very, very,"; \
\   " very, very, very, very, very, very,"; \
\   " very long line."
END
```

Variabelen
Kijk uit als u BBC BASIC wilt gebruiken, bijvoorbeeld het gebruik van variabelen. Tot nu toe zien we nog geen groot verschil tussen deze versie en de andere BASIC dialecten. Echter is BBC BASIC veel strenger wanneer we gebruik willen maken van variabelen.

Misschien is het u al opgevallen dat **PRINT** en **print** totaal verschillend zijn. Als bij het Options menu de keuze **Lowercase keywords** uitzet, dan kunnen we alleen maar gebruik maken van hoofdletter sleutelwoorden. BBC BASIC is zeer gevoelig voor namen; wat kan wel en wat kan niet?

Dit mag	Dit niet	Waarom niet?

`_my_float`
`AnInteger%`
`FirstName$`
`NUM1`
	`1_man_went_to_mow`
	`Went to mow a meadow`
	`PRINT_TOTAL`
		Start met een nummer.
		Heeft spaties.
		Zelfs gebruik van het onderlijn-teken wordt PRINT nog steeds gezien als een sleutelwoord.

Andere BASIC dialecten zijn wat gemakkelijker en vinden het wel prima als u onderlijning gebruikt, maar hier mag het dus niet altijd, omdat het niet als een hele variabelennaam wordt gezien, tenzij de letters ertussen voor BBC onbekend is.

We mogen variabelen niet eerder gebruiken dan wanneer ze geïnitialiseerd zijn, ook niet wanneer ze gebruikt zijn en we plotseling het statement CLEAR laten uitvoeren. Alles is gewist en we moeten de variabelen opnieuw initialiseren.

Variabelen worden in BBC BASIC niet gedeclareerd, wel arrays en objecten. Een toekenning is als initialisatie voldoende. Bent u PowerBASIC gewend? Mooi, dan kan ik u vertellen dat BBC BASIC ook de symbolen kent die we achter de variabelen gebruiken.

We kunnen bijvoorbeeld op de oude manier onderstaande regels gebruiken:

```
I%=0
I%=I%+20
```

Elke wiskundige zou dit vreemd vinden. Dit maakt een verwarrend gevoel. Hoe kan *I%* gelijk worden gemaakt aan *I%* plus 20? In een computertaal is dit toegestaan. Als BASIC dit calculeert, gebruikt het een kladblok. Wat het doet is: haal de huidige waarde uit *I%*, plaats het in het werkgebied en tel er 20 bij op. Neem dan het resultaat en zet het terug in de variabele genoemd *I%*.

Er is een kortere manier om direct BASIC te kunnen vertellen: bewaar geen variabele, maar tel direct de waarde op met de oude waarde en ken het resultaat toe. Hieronder ziet u het, een beetje C-stijlachtig, niet?

```
I%+=20
```

Het doet hetzelfde als de vorige regel deed, maar het bespaart extra typewerk en bytes voor de gebruikers die de trial versie gebruiken. U kunt dit ook zien als: *I%* is verhoogd met 20.

Bekijk eens onderstaande demonstratie welke operatoren er zijn:

```
I%+=1    : REM I% increases by 1
I%-=1    : REM I% decreased by 1
I%*=10   : REM I% multiplied by 10
I%/=2    : REM I% divided by 2
I%MOD=2  : REM I% MODed by 2
I%DIV=2  : REM I% DIVed (!) by 2
END
```

Maar BBC BASIC is niet zo moeilijk met de code. Onderstaande regel is ook toegestaan:

```
I% + = 20
```

Het is natuurlijk beter BASIC te kennen zoals we de andere BASIC dialecten ook kennen. Houd dus de operatoren bij elkaar om fouten te voorkomen.

Net als in de andere BASIC dialecten kunnen we gebruik maken van rekenfuncties, zoals onderstaand voorbeeld:

```
REM LN and EXP
A=3.4
B=LN(A)
C=EXP(B)
PRINT A;",";B;",";C
END
```

Voor gebruik van cirkels en hoeken kent BBC BASIC een constante, een variabele waarvan we de waarde niet mogen wijzigen. Deze heet: **PI.**

Alle trigonometrische functies werken met radialen. Er is een PI radiaal in 180 graden (halve cirkel) dus 1 graad = PI/180 radialen. Voor meer leven in de brouwerij is er een functie die een waarde converteert van radialen naar graden: DEG(X) en een functie RAD(X) die een waarde converteert van graden naar radialen.

Zo ook met sinus, cosinus en tangens van een gegeven hoek X. Zoals hierboven gezien werken deze ook allemaal in radialen, maar dankzij DEG en RAD kunnen we sneller werken als we willen tekenen.

```
PRINT SIN(RAD(60))
PRINT COS(PI/2)
```

De stringfuncties LEFT$, MID$ en RIGHT$ zijn ook bekend. Sommige BASIC dialecten ondersteunen de MID$ functie ook als een statement om bepaalde plaatsen in een tekst te vervangen door andere tekst. In BBC BASIC kan het ook met LEFT$, zoals onderstaand voorbeeld:

```
REM LEFT$ as an assignment
MyStr$="Hello, world"
LEFT$(MyStr$,6)="Byebye"
PRINT MyStr$
```

```
END
```

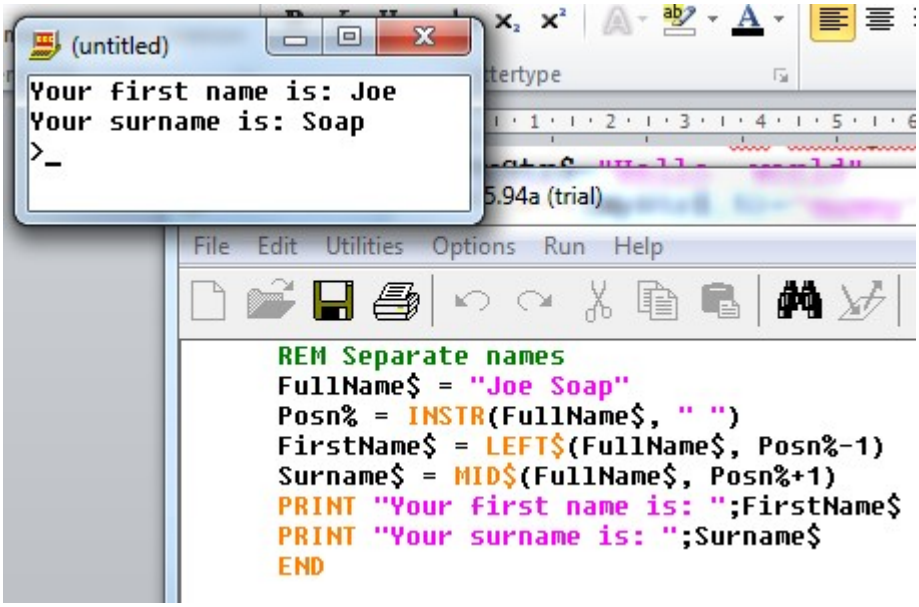
Zo dus ook met RIGHT$:

```
REM RIGHT$ as an assignment
MyStr$="Hello, world"
RIGHT$(MyStr$,5)="mummy"
PRINT MyStr$
END
```

Gelukkig is het bij MID$ nog steeds hetzelfde gebleven, dus die werkt zoals het altijd werkt:

```
REM MID$ demo
A$ = "Give me patience!!"
MID$(A$,9,8) = "strength"
PRINT A$
END
```

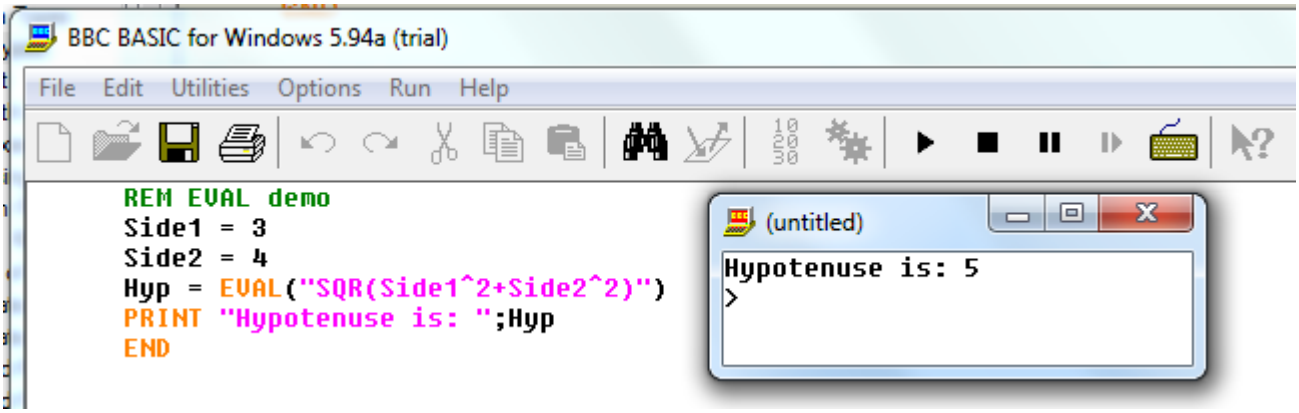
Hieronder een compleet voorbeeld met de uitvoer.



We weten wat we kunnen doen met de functie VAL. Sommige BASIC dialecten kennen ook de functie EVAL. Weet u of EVAL in BBC BASIC op dezelfde manier werkt als EVAL in het dialect dat u gebruikt?

```
PRINT VAL("20")
PRINT EVAL("144/12")
```

De tweede PRINT regel zal de waarde 12 weergeven. Nog niet overtuigd? Bekijk eens onderstaand voorbeeld:



U kunt in de string alles opgeven wat normaal als expressie geldig is, dus ook variabelen en functienamen. EVAL evalueert alles, zolang het resultaat een geldige waarde is.

IF…THEN…ELSE
We weten dat ELSE optioneel is. In sommige BASIC dialecten is THEN ook optioneel, alleen als er een GOTO of een GOSUB wordt gebruikt.

Tip! Werkt in een BASIC dialect die u gebruikt een GOSUB niet zonder THEN? Helaas, dan kent het BASIC dialect het niet en moet u THEN GOSUB gebruiken.

In BBC BASIC kan ook een IF regel zonder THEN worden gebruikt, maar ook weer in C-stijl. We kunnen de volgende regel gebruiken:

```
IF Score%>=40 Pass=1 ELSE Pass=0
```

Willen we meerdere regels gebruiken in een IF, dan is THEN wel verplicht.

```
IF Salary>1000000 THEN
  BuyYacht=1
  BuyVilla=1
  BuyHelicopter=1
ELSE
  PRINT "Work, work and more work"
ENDIF
```

Denk erom dat een blok met END én het juiste sleutelwoord aan elkaar afgesloten moet worden. Anders dan in andere BASIC dialecten, waar we END IF gewend zijn met een spatie ertussen, kent BBC BASIC het alleen als een compleet één woord statement.

CASE…OF…WHEN…OTHERWISE…ENDCASE
Hee, zou u denken, waar is de bekende SELECT CASE?
BBC BASIC kent geen SELECT CASE. Deze werkt op een hele andere manier. Even wennen, we zitten nog steeds in BASIC!
Willen we meerdere keuzes maken, dan gebruiken we CASE als volgt:

```
REM Simple menu system
CLS
PRINT "Press 1 for option 1"
PRINT "Press 2 for option 2"
PRINT "Press 3 for option 3"
INPUT "Enter choice: " Choice%
CASE Choice% OF
  WHEN 1: PRINT "You chose option 1"
```

```
  WHEN 2: PRINT "You chose option 2"
  WHEN 3: PRINT "You chose option 3"
ENDCASE
END
```

Kan er geen keuze worden gemaakt, dan kunnen we dat ook afvangen:

```
REM Geography quiz
PRINT "What is the capital of France:"
PRINT "a) Paris"
PRINT "b) London"
PRINT "c) Madrid"
INPUT "Enter a,b or c: " Reply$
CASE Reply$ OF
  WHEN "A", "a": PRINT "Correct"
  WHEN "B", "b": PRINT "Sorry, that's England"
  WHEN "C", "c": PRINT "Sorry, that's Spain"
  OTHERWISE: PRINT "Sorry, invalid response"
ENDCASE
END
```

In plaats van CASE ELSE in andere dialecten, kent BBC BASIC het als OTHERWISE.

Wat vindt u van BBC BASIC, lijkt het u wat om er eens mee te experimenteren of zou u BBC BASIC willen leren? Download de gratis versie en probeer de code eens. Zelf vind ik het een mooie BASIC taal. De volgende keer laat ik u nog mooiere voorbeelden zien, zoals met arrays en objecten. Hoe gaat BBC BASIC om met DIM en hoe werken de procedures (PROC…ENDPROC) en de functies (FN…ENDFN)? Dit komt de volgende keer allemaal aan bod.

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	Marco Kurvers	do. t/m zo.	19-22	(0342) 424452	m.a.kurvers@hccnet.nl
Office					
Web Design, met XHTML en CSS