Programmeren Bulletin

24ste jaargang april 2017 Nummer 1

hcc! programmeren

Interessegroep

Inhoud

Redactioneel

Het klokproject is een goed voorbeeld hoe we met een script eenvoudig de wijzers kunnen laten draaien. Unity heeft zelf objecten die de posities van de wijzers automatisch berekent. Wiskundige functies als SIN en COS zijn niet meer nodig.

Python heeft tegenover andere programmeertalen een aparte structuur. Vooral de samengestelde statements zitten heel anders in elkaar. Vreemd genoeg kan het juist een goede structuur geven in plaats van dat de manier van Python nadelig wordt gezien.

Van Game Maker Studio liet ik de IDE zien en zag u de werking van de objecten en hoe de scripts uitgevoerd worden. Zelf heb ik besloten niet langer meer aandacht te besteden aan Game Maker, omdat Unity veel meer mooie techniek geeft wat bouwen en programmeren in 2D en 3D betreft. Wilt u meer weten van Game Maker Studio? Kijk in de documentatie die u kunt vinden in Google of ga naar de website YoYoGames voor meer informatie.

De naam PowerBASIC zegt het al. Het geeft extra kracht in de BASIC taal. PowerBASIC ondersteunt pointers om met lijsten en recordcollecties te werken zonder een opgegeven limiet. Bovendien werkt het ook veel sneller dan met gewone arrayvariabelen en recordvariabelen.

Marco Kurvers

Unity 3D – Het klokproject.

Unity 3D is een handig programma om 3D objecten op de scene te plaatsen. Een scene is de camera die gericht is op het gedeelte van de wereld die je ziet. Later meer over de scene.
Het verschil met XNA Game Studio, waarmee je hetzelfde kunt doen, is echter dat XNA geen IDE venster heeft. U kunt dan niet direct zien wat er weergegeven wordt als er een object getekend wordt.

Het klokproject

Een goed voorbeeld om te laten zien hoe een project met 3D objecten gemaakt wordt, is het klokproject. Start Unity en kies voor een nieuw project. Kies voor de 3D modus. U hoeft geen extra assets te kiezen (mocht de keuze erbij staan).

U zult hetzelfde Unity venster zien zoals u die in de vorige Bulletin zag, maar nu met een lege scene en met alleen de Main Camera, die ervoor zorgt dat we de scene in de game kunnen zien.

We gaan beginnen met het plaatsen van de wijzers. Daarna maken we het script, dat voor het bewegen van de wijzers moet zorgen. De code zal ook uitgebreid uitgelegd worden.

Volg onderstaande stappen voor het plaatsen van de objecten.

Kies in het menu **GameObject** het menu-item **Light** en klik in het submenu op het menu-item **Directional Light**.

Klik op het object Directional Light.

Ga naar de **Inspector** en wijzig de positie als volgt:	X: 0	Y: 3	Z: 0
Wijzig de rotatie als volgt:	X: 50	Y: -30	Z: 0
Wijzig de schaal als volgt:	X: 1	Y: 1	Z: 1

Kies in het menu **GameObject** het menu-item **3D Object** en klik in het submenu op het menu-item **Cube**.

Klik op het object Cube en ga naar de Inspector.

Wijzig de positie als volgt:	X: 0	Y: 1	Z: 0
Wijzig de rotatie als volgt:	X: 0	Y: 0	Z: 0
Wijzig de schaal als volgt:	X: 0.5	Y: 2	Z: 0.5

De Cube moet het uur zijn. We kunnen de Cube ook zo noemen. In dit voorbeeld, dat ik straks laat zien, heb ik het echter anders gedaan. Elke Cube plaats ik in een leeg hoofdobject met wel de naam. Alleen de hoofdobjecten zullen in het script aanwezig zijn. Ook zal er een hoofdobject zijn voor de hele klok. Dat is handig, omdat dan alle objecten geselecteerd zijn. De hele klok kan dan bijvoorbeeld gekopieerd en geplakt worden op een andere plaats, zonder nieuwe cubes te hoeven maken. Houd er wel rekening mee de namen van de gekopieerde objecten te wijzigen om fouten te voorkomen.

Ga naar het menu GameObject en klik op **Create Empty**. Er zal een leeg object worden gemaakt. Klik met de rechter muisknop op het object en klik op het menu-item **Rename**. Wijzig de naam van het object in **Uren** of zoals in dit voorbeeld in **Hours**.

Klik op het Cube object. Houd de linker muisknop ingedrukt en sleep deze op het Uren of Hours object. De Cube zal als sub-object in het Hours object staan.

Maak weer een Cube object aan via het menu. Klik op het object en ga naar de Inspector.

Wijzig de positie als volgt:	X: 0	Y: 1.5	Z: 0
Wijzig de rotatie als volgt:	X: 0	Y: 0	Z: 0
Wijzig de schaal als volgt:	X: 0.25	Y: 3	Z: 0.25

Maak een leeg object aan met Create Empty, zie hierboven. Noem deze **Minuten** of **Minutes**. Sleep het Cube object (niet degene die in het Hours object staat) op het Minutes object.

Maak een Cube object aan via het menu. Klik op het object en ga naar de Inspector.

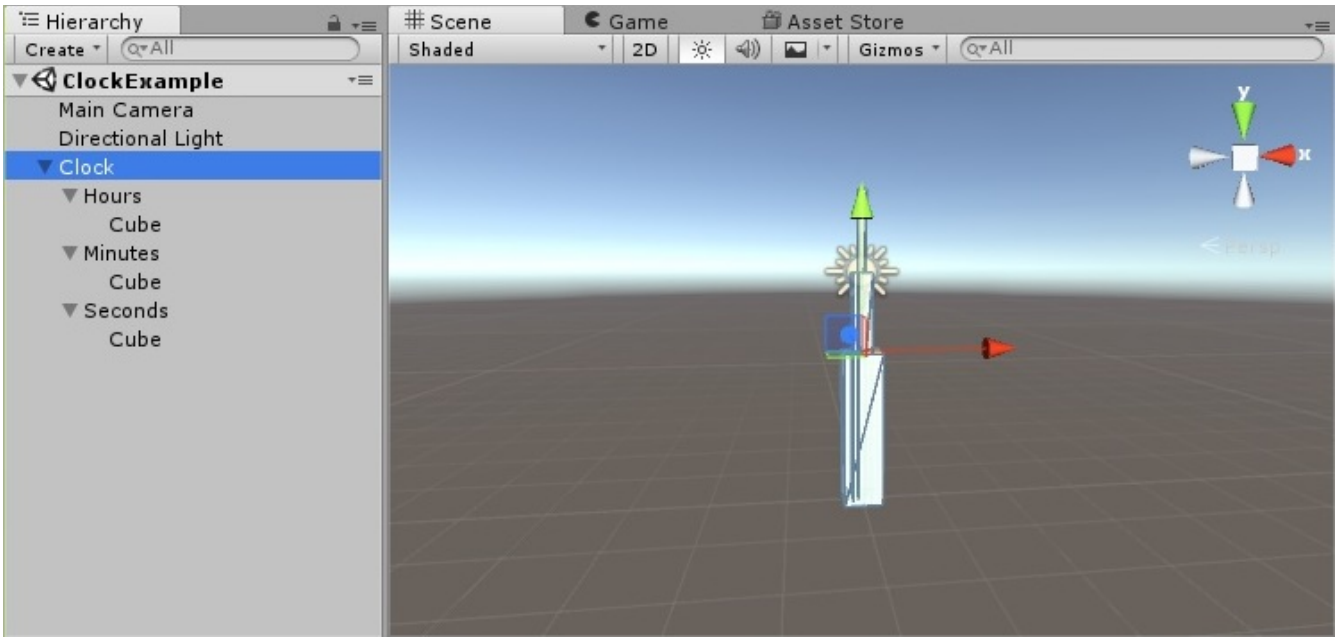
Wijzig de positie als volgt:	X: 0	Y: 2	Z: 0
Wijzig de rotatie als volgt:	X: 0	Y: 0	Z: 0
Wijzig de schaal als volgt:	X: 0.1	Y: 4	Z: 0.1

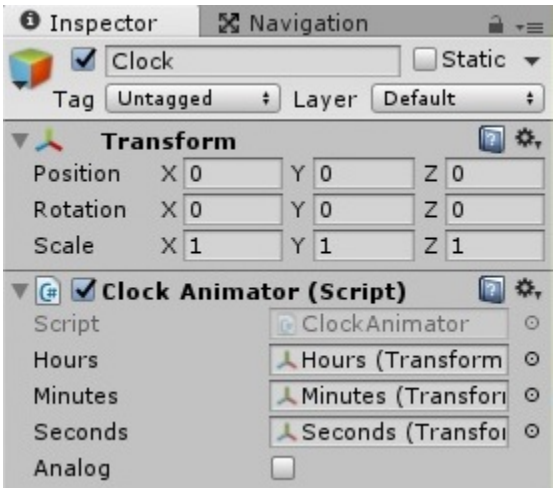
Maak een leeg object aan met Create Empty. Noem deze **Seconden** of **Seconds**. Sleep het Cube object op het Seconds object.

Voordat we het script kunnen maken, moet u eerst het volgende weten: scripts kunnen niet zomaar worden gestart. Elk script moet gekoppeld worden aan een object om te kunnen werken. Het script van het klokproject gebruikt de drie wijzers om daaraan de posities te wijzigen met het DateTime object. We hebben drie objecten gemaakt, maar aan welk object moeten we het script koppelen? Het antwoord is: aan geen één. Wat we nodig hebben is een hoofdobject dat de hele klok bezit, net zoals ik eerder heb verteld dat we dan handig van meerdere objecten een kopie kunnen maken. Doordat we het script dan gaan koppelen aan het hoofdobject, zullen de wijzers als instanties in het script aanwezig zijn in de Inspector van het hoofdobject. Voor elke wijzer een script maken zou in principe ook kunnen, maar alle wijzers in één script gebruiken is overzichtelijker.

Maak een leeg object aan met Create Empty. Noem deze **Klok** of **Clock**. Sleep de Hours, Minutes en Seconds hoofdobjecten op het Clock object.

Als alles goed is gegaan met bovenstaande stappen, ziet u de hiërarchie en de scene zoals de afbeelding.



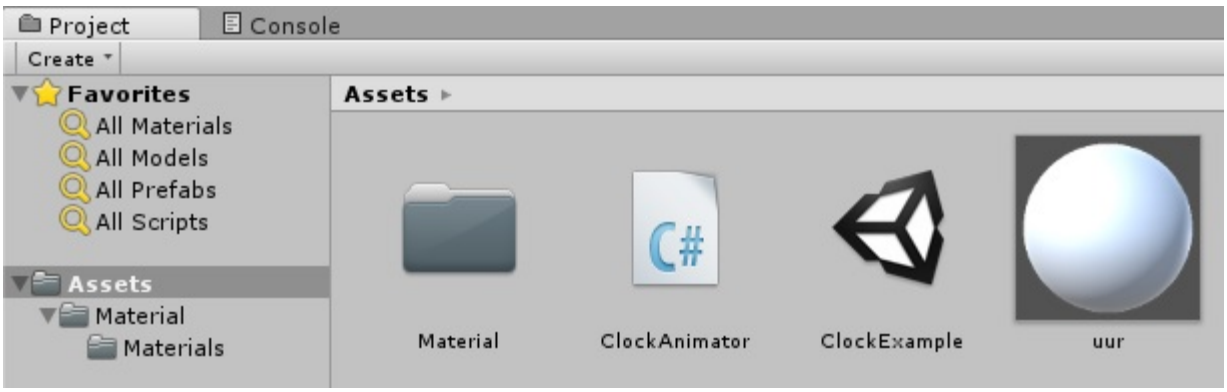


Naast de scene is de Inspector. Het Clock object, dat geselecteerd is, bevat ook een Transform. Omdat het Clock object het hoofdobject is van alle wijzers, hoeft er geen positie ingesteld te worden.

Het tweede gedeelte is het script dat we later maken. We moeten er voor zorgen dat het script in de Inspector komt, want dat gebeurt niet zomaar.

Ook de Hours, Minutes en Seconds objecten staan erbij met de juiste Transform object instanties. Dit is allemaal gedeclareerd in het script om ervoor te zorgen dat de juiste posities van de objectinstanties doorgegeven worden.
De Analog optie is geen object maar een eigenschap. Ook deze is gedeclareerd als variabele in het script en dus geen onderdeel van de Inspector zelf. Over de Analog variabele wordt later meer uitgelegd.

Onderaan ziet u de Assets voor het project. Assets zijn onderdelen die voor de kleuren, geluiden, materialen, prefabs en voor de besturingen moeten zorgen. Alle onderdelen moeten naar de juiste objecten worden gesleept in de hiërarchie. Dat moet ook worden gedaan met het script dat u op de afbeelding ziet staan, de ClockAnimator.



De ClockExample is de scene naam. Elke scene moet ook opgeslagen worden, net zoals een Visual Studio formulier of klasse opgeslagen wordt. Het Unity project is dus niet het enige bestand. De rest van wat u ziet staan is op dit moment niet van belang en valt buiten dit onderwerp.

Het script van het klokproject.

Om het script te kunnen maken, moet u de Visual Studio Community Unity Scripts hebben. U hoeft Visual Studio niet te starten. Unity zorgt ervoor dat de scripteditor geopend wordt.

Klik met de rechter muisknop op de Assets map, die u op de afbeelding geselecteerd ziet staan.

Ga naar het menu-item **Create** en klik op **C# Script**. Er wordt een script gemaakt met de naam **NewBehaviourScript**. Gelijk ziet u de Inspector veranderen in een alleen-lezen codevenster. Dit is het geraamte van het script.

Wijzig de naam van het script in de naam **ClockAnimator**.

Open het script door er op te dubbelklikken. Unity opent het scriptvenster.

Indien u ziet dat de naam van de klasse niet ClockAnimator is, want dat kan gebeuren als de scriptnaam in Unity gewijzigd is, dan kunt u deze zelf wijzigen.

Het script heeft twee functies: de Update() functie en de Start() functie. De Update() functie is degene die we nodig hebben. Tijdens de uitvoer zal steeds de Update() functie uitgevoerd worden. In de functie zullen de posities van de drie wijzers met de tijduren, de tijdminuten en de tijdseconden berekend worden, alsof de wijzers echt bewegen.
De Start() functie hebben we niet nodig. Die functie kunt u verwijderen.

Het geraamte van het script, zonder de Start() functie, zal er als volgt uitzien:

```
using UnityEngine;
using System;

public class ClockAnimator : MonoBehaviour {

     // Update is called once per frame
     private void Update () {

     }
}
```

Eerst declareren we drie variabelen die de uren, minuten en seconden om moeten zetten naar het aantal graden. Dit is nodig om de wijzers in de juiste hoekrichting te laten wijzen.

Typ onderstaande code onder de *public class* regel:

```
private const float
     hoursToDegrees = 360f / 12f;
     minutesToDegrees = 360f / 60f;
     secondsToDegrees = 360f / 60f;
```

De variabelen zijn constanten die alleen in de expressies worden berekend samen met het TimeSpan object of het DateTime object. Deze twee objectkeuzes worden bepaald aan de hand van of wel of niet de klok op analoog staat.

De volgende regels code zijn de declaraties voor de objectinstanties en de analog variabele die later te zien zullen zijn in de Inspector:

```
public Transform hours, minutes, seconds;
public bool analog;
```

In de Update() functie gebeurt de rest. Typ onderstaande code in de functie:

```
if (analog)
{
      TimeSpan timespan = DateTime.Now.TimeOfDay;
      hours.localRotation = Quaternion.Euler(0f, 0f,
            (float)timespan.TotalHours * -hoursToDegrees);
      minutes.localRotation = Quaternion.Euler(0f, 0f,
            (float)timespan.TotalMinutes * minutesToDegrees);
      seconds.localRotation = Quaternion.Euler(0f, 0f,
            (float)timespan.TotalSeconds * -secondsToDegrees);
}
else
{
      DateTime time = DateTime.Now;
      hours.localRotation = Quaternion.Euler(0f, 0f, time.Hour * -hoursToDegrees);
      minutes.localRotation = Quaternion.Euler(0f, 0f,
            time.Minute * -minutesToDegrees);
      seconds.localRotation = Quaternion.Euler(0f, 0f,
            time.Second * -secondsToDegrees);
}
```

De analog variabele bepaald of de klok analoog moet lopen of niet. Analoog wil zeggen dat de wijzers niet per seconde draaien, maar op elk moment wanneer de Update() functie wordt uitgevoerd, dus per frame. Anders zal de klok wel per seconde lopen en zult u zien dat de secondewijzer in stappen per seconde zal draaien. Als er geen analog keuze is, zal ook daarom in de Euler() functie niet de time waarden omgezet worden in float.

Het TimeSpan object bepaalt precies de tijd van de dag. Om de wijzers te kunnen zien draaien, worden niet de posities gebruikt maar de rotaties. Er worden lokale rotaties gebruikt om de wijzers na elk frame vast te zetten, zodat de Euler() functie in het Quaternion object een nieuwe richting terug kan geven.

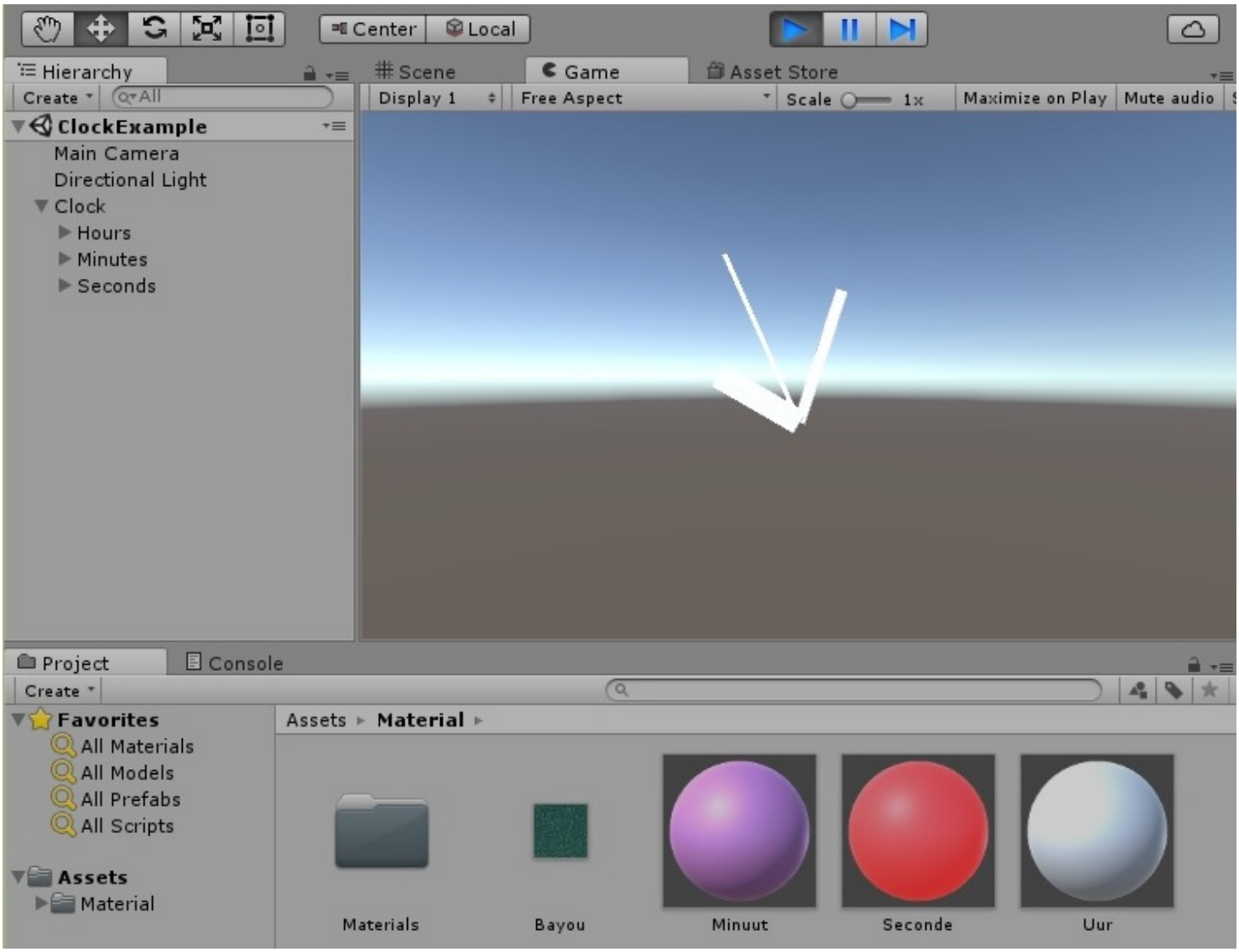
Sla de code op en ga terug naar Unity. Als u nu op het Clock object klikt, ziet u in de Inspector nog niet het script object met de inhoud.

Elke asset moet worden gekoppeld aan het object waar het toebehoort, ook zo het script. Het script moet dus gekoppeld worden aan het Clock object.

Klik op het script in de Assets map en sleep deze naar het Clock object. In de Inspector ziet u de inhoud van het script verschijnen, de scriptnaam, de drie objectinstanties en de analog variabele, die in de Inspector als een eigenschap werkt – een keuzevak. U ziet echter naast de objectinstanties niet de objectnamen staan die nodig zijn om de wijzers in de scene te kunnen koppelen. Zonder dat deze erin staan, werkt de klok niet. De waarden van de objectinstanties in de code, moeten getransformeerd worden met de objecten die in de Hiërarchie staan. Vandaar dat de objectinstanties van het type **Transform** zijn.

Sleep elk object Hours, Minutes en Seconds vanuit de Hiërarchie naar het juiste object instantie in de Inspector.

Start nu het project door op het driehoekje bovenaan te klikken. Als alles goed is gegaan, kunt u de klok zien lopen op de juiste tijd.
Zet de klok stil en vink nu in de Inspector het analog vakje aan. Start de klok nog eens en zie nu het verschil; De wijzers lijken nu gladder te lopen en dat komt doordat er niet per seconde wordt gedraaid, maar nu per frame.



In de Assets ziet u in de Material map ook een textuur en drie kleuren aangemaakt. Het is mogelijk zelf sprites en kleuren als materialen te maken. Zo kunt u elk 3D object aankleden met texturen en kleuren. U hoeft hetmateriaal dat u wilt alleen maar naar het juiste object te slepen.

Liberty BASIC API Reference

Het clipboard, of ook wel klembord genoemd, gebruiken we eigenlijk wel vaak. Zonder dat we het in de gaten hebben, gebruiken we het gemiddeld 15 keer op een dag. Dat lijkt weinig, maar als we het gemiddelde gaan verdelen in het aantal uren, dat we op een dag gebruiken voor tekstverwerking, is het plotseling niet meer weinig.

Clipboard API Functies

Tekst kan gekopieerd worden naar het clipboard en geplakt worden vanuit het clipboard met een tekstverwerker of tekstvenster. Als een programma meer flexibiliteit vereist, dan is er de mogelijkheid toegang te krijgen tot het clipboard met API functies. Tekst, afbeeldingen en wav geluiden kunnen gekopieerd worden naar het clipboard en vanuit het clipboard teruggezet worden voor gebruik in Liberty BASIC programma's.

OpenClipboard

Voor toegang naar het Windows Clipboard via de API, moet het programma eerst een aanroep maken met OpenClipboard. Het argument is del venster handle en de teruggegeven waarde is ongelijk aan nul als de functie met succes is uitgevoerd. Als de handle parameter NULL (0) is, zal de geopende clipboard verbonden zijn met de huidige taak.

```
calldll #user32, "OpenClipboard", _
     hWnd as ulong, _
     r as boolean
```

Het schoonmaken van het clipboard

Voordat u van alles op het clipboard plaatst, moet u eerst zeker weten het schoon gemaakt te hebben met EmptyClipboard. Maakt u het niet schoon voor het updaten, dan zullen alle formatteringen, die u niet expliciet overschrijft, overblijven.

```
calldll #user32, "EmptyClipboard", _
     r as boolean
```

CloseClipboard

Roep CloseClipboard aan wanneer de clipboard routines compleet zijn. Het zal een ongelijk aan nul teruggeven als het succesvol is.

```
calldll #user32, "CloseClipboard", _
     r as boolean
```

SetClipboardData

Om wat op het clipboard te plaatsen als deze open is, gebruik SetClipboardData. Het programma moet de opmaak van de gegevens kunnen aangeven, zodat het clipboard weet of het tekst of een afbeelding is. Sommige mogelijke formaten zijn:

_CF_TEXT
_CF_BITMAP
_CF_WAVE

Het programma moet de geheugen handle naar de gegevens versturen. Als de tekst is verzonden naar het clipboard, worden de geheugenallocatiefuncties gebruikt om een handle te maken naar de tekst in het geheugen. Zie later een demo als een stap-voor-stap voorbeeld. Het is nodig de functies GlobalAlloc, GlobalLock, GlobalUnlock en GlobalFree te gebruiken. Als het programma een afbeelding stuurt naar het clipboard, moet het een handle leveren naar de bitmap, geresulteerd door de Liberty

BASIC hbmp() functie. Als de functie succesvol is, zal de teruggegeven waarde de handle van de gegevens zijn.

```
calldll #user32, "SetClipboardData", _
     format as ulong, _       'het gegevenstype
     handle as ulong, _       'geheugenhandle naar de gegevens
     rethandle as ulong       'handle van de gegevens op het clipboard
```

GetClipboardData
Gebruik GetClipboardData om gegevens vanuit het clipboard te resulteren. Het eerste argument is het type van het gezochte formaat. Het programma moet opgeven of het naar tekst, een afbeelding of een wav zoekt. Als de gegevens in het opgegeven formaat op het clipboard is, zal de functie een handle teruggeven naar de gegevens. Is de teruggegeven waarde null, dan zal het niet aanwezig zijn op het clipboard.

```
calldll #user32, "GetClipboardData", _
     format as ulong, _  'gezochte formaat
     rethandle as ulong  'gegevenshandle op clipboard, 0=geen gegevens
                         'in dit formaat

calldll #user32, "GetClipboardData", _
     _CF_TEXT as ulong, txtHandle as ulong

calldll #user32, "GetClipboardData", _
     _CF_BITMAP as ulong, hBmp as ulong

calldll #user32, "GetClipboardData", _
     _CF_WAVE as ulong, hWav as ulong
```

De gegevens moeten worden beheerd door het programma afhankelijk van de indeling. Als de tekst ontvangen is vanuit het clipboard, is de teruggave de geheugenpointer naar de tekst. Gebruik de winstring() functie om de actuele tekst op te halen. Als een afbeelding ontvangen is vanuit het clipboard, gebruik dan de LOADBMP functie om het te laden vanuit de handle, teruggegeven door de functie. Als eenmaal het programma een Liberty BASIC naam gegeven heeft met LOADBMP, is het mogelijk om DRAWBMP of BMPSAVE te gebruiken, net als een afbeelding geladen van de harde schijf of diskette. Als de handle van een wav ontvangen is vanuit het clipboard, kan het niet afgespeeld worden met PLAYWAVE. Het programma moet het afspelen met de API functie sndPlaySound, omdat het een wav is in het geheugen. Zie het derde programma onderaan voor de voorbeelden van het ontvangen en gebruiken van tekstgegevens, afbeelding en wav formaat.

Plaats tekst op het clipboard

```
'Zend tekst naar clipboard met API aanroepen
Text$ = "Liberty BASIC!" : Size = len(Text$)

'Wijs een geheugenblok toe voor de grootte van de tekst
calldll #kernel32, "GlobalAlloc", _
     _GMEM_MOVEABLE as ulong, _    'type
     Size as ulong, _              'grootte van geheugenblok om toe te wijzen
     hMemory as ulong              'handle naar geheugen

'vergrendel het geheugenblok
calldll #kernel32, "GlobalLock", _
     hMemory as ulong, _           'geheugenhandle
     lpMemory as long              'pointer naar vergrendeld geheugen

'kopieer de tekst in het vergrendeld geheugen
calldll #kernel32, "RtlMoveMemory", _
     lpMemory as ulong, _          'pointer naar vergrendeld geheugen
     Text$ as ptr, _               'de te kopiëren tekst naar geheugen
     Size as long, _               'lengte van de te kopiëren tekst
     ret as void

'ontgrendel het geheugen
calldll #kernel32, "GlobalUnlock", _
     hMemory as ulong, _           'handle naar geheugen
     ret as boolean                'ongelijk aan nul = succes

'open het clipboard
calldll #user32, "OpenClipboard", _
     hwndOwner as ulong, _         'vensterhandle of 0
     r as boolean

'maak clipboard schoon
calldll #user32, "EmptyClipboard", ret as boolean

'kopieer de teksstring naar het clipboard
calldll #user32, "SetClipboardData", _
     _CF_TEXT as ulong, _          'gegevenstype
     hMemory as ulong, _           'handle naar geheugenblok
     setReturn as ulong            'ongelijk aan nul = succes

'sluit het clipboard
calldll #user32, "CloseClipboard", r as boolean

'als de verbindingspoging is mislukt, maak dan het geheugen vrij
if setReturn = 0 then
     calldll #kernel32, "GlobalFree", _
          hMemory as ulong, _      'handle naar geheugen
          ret as ulong   '0=succes, ongelijk aan nul=geheugenhandle
end if
if setReturn then
     print "Tekst gekopieerd naar clipboard."
else
     print "Niet mogelijk om tekst te kopiëren naar clipboard."
```

```
end if
end
```

Plaats een afbeelding op clipboard en haal een afbeelding van clipboard

```
'Deze kleine demo laat zien hoe een afbeelding naar het clipboard wordt
'verzonden.
'Het laat ook zien hoe een afbeelding van het clipboard wordt gehaald en
'weergegeven wordt in een Liberty BASIC programma.

nomainwin
WindowWidth=340:WindowHeight=480
graphicbox #1.g, 10,10,300,300
statictext #1.s, "",10,320,600,50
open "Clipboard Demo" for window as #1
print #1, "trapclose [quit]"
h=hwnd(#1)

filedialog "Open","*.bmp",bmpfile$
if bmpfile$="" then [quit]
loadbmp "forclip",bmpfile$
hBitmap=hbmp("forclip")  'haal bmp handle

'open clipboard
calldll #user32, "OpenClipboard", h as ulong, result as boolean

'maak clipboard schoon
calldll #user32, "EmptyClipboard", ret as boolean

'plaats bmp gegevens op clipboard
calldll #user32, "SetClipboardData", _CF_BITMAP as ulong, _
     hBitmap as ulong, rethandle as ulong

'kijk of bmp gegevens op clipboard is
calldll #user32, "GetClipboardData", _CF_BITMAP as ulong, _
     hBmp as ulong

'als de bmp gegevens op het clipboard is, laad dan de gegevens en
'teken het
if hBmp<>0 then
     loadbmp "demo", hBmp
     print #1.g, "down;fill lightgray;drawbmp demo 0 0;flush"
end if

calldll #user32, "CloseClipboard", result as boolean
print #1.s, "Return for CF_BITMAP ";hBmp
wait

[quit]
if hBmp<>0 then unloadbmp "demo"
close #1:end
```

Haal tekst, afbeelding en wav gegevens van het clipboard

```
'Om deze demo te gebruiken, doe het volgende:
'Open notepad, typ wat tekst en kies Kopiëren OF open Paint, open een
'afbeelding, selecteer alles of doe alleen het deel ervan en kies Kopiëren
'OF open een wav in een geluidsrecorder en kies Kopiëren.
'Hebt u tekst gekopieerd naar het clipboard, dan zal het verschijnen in de
'teksteditor.
'Hebt u een afbeelding gekopieerd naar het clipboard, dan zal het getekend
'worden in de graphicbox.
'Hebt u een wav gekopieerd, dan zal het worden afgespeeld.

nomainwin
WindowWidth=640:WindowHeight=480
texteditor #1.t, 10,10,300,300
graphicbox #1.g, 320,10,300,300
statictext #1.s, "",10,320,600,50
button #1.b, "Do another!",[again],UL,10,380
open "Clipboard Demo" for window as #1
print #1,"trapclose [quit]"
h=hwnd(#1)
[again]
calldll #user32, "OpenClipboard", h as long, result as long
calldll #user32, "GetClipboardData", _CF_TEXT as ulong, txt as ulong
if txt<>0 then
     print #1.t, "!cls"
     print #1.t, "Text content of clipboard:"
     print #1.t, ""
     print #1.t, winstring(txt)
     print #1.t, "!origin 1 1"
end if

calldll #user32, "GetClipboardData", _CF_BITMAP as ulong, hBmp as ulong
if hBmp<>0 then
     loadbmp "demo", hBmp
     print #1.g, "cls;down;drawbmp demo 0 0;flush"
end if

calldll #user32, "GetClipboardData", _CF_WAVE as ulong, hWav as ulong
if hWav<>0 then
     SND.SYNC = 0 : SND.ASYNC = 1
     SND.NODEFAULT = 2 : SND.MEMORY = 4
     mode=SND.MEMORY or SND.ASYNC or SND.NODEFAULT
     calldll #winm, "sndPlaySoundA", _
          hWav as ulong, mode as long, result as boolean
end if

calldll #user32, "CloseClipboard", result as boolean
msg$="Return for CF_TEXT ";txt
msg$=msg$+chr$(13)+"Return for CF_BITMAP ";hBmp
msg$=msg$+chr$(13)+"Return for CF_WAVE ";hWav
print #1.s, msg$
wait

[quit]
```

```
if hBmp<>0 then unloadbmp "demo"
close #1
end
```

De drie demo’s laten zien hoe het clipboard in Liberty BASIC opgenomen kan worden. Op die manier kunt u functies in uw programma’s maken die zelf de API calls aanroepen. Telkens weer dezelfde API aanroepen vergt veel programmeerwerk en tijd.

In de volgende Bulletin komen de environments, enum functies en meervoudige timers aan bod.

Programmacode – Tips, trucs, foutjes en vergissingen.

Programma's kunnen we niet honderd procent correct schrijven. Fouten maken we altijd wel, van kleine foutjes die niet door de compiler worden gezien, en zelfs ook niet altijd door de programmeur zelf, tot de grote fouten die zelfs zoveel verwarring kunnen schoppen dat het debuggen van de programma's veel tijd kost.

Het lijkt u misschien onwerkelijk, maar fouten kunnen ook onzinnig zijn. Het zijn fouten die niet als fouten herkend worden. Elke fout heeft zijn eigen karakteristiek. Denk maar aan het opzoeken naar een fout die niet gegeven werd door de compiler, maar wel verkeerde waarden weergeeft. Waar zit de fout dan? Om toch in het programma de code te vinden die de verkeerde waarden veroorzaakt, zouden we tientallen breakpoints kunnen plaatsen of het programma stap voor stap uit kunnen voeren. Maar onzinnigheid overwint het 'lui doen' met programmeren. Vooral beginners, die het zoeken naar fouten lastig vinden, plaatsen extra controle in hun programma. Op die regels worden dan breakpoints gezet, of daarna. Een onzinnige regel is bijvoorbeeld: test = test. De variabele 'test' wordt globaal gedeclareerd, of in de hele klasse. Lokale variabelen in procedures en functies verliezen hun waarden na de aanroep. Door in de codeblok 'test = lokale variabele' te gebruiken, kan daarop gecontroleerd worden of de doorgegeven waarde goed is of niet. De andere, 'test = test' wordt meestal een aantal regels verder gebruikt om nog eens een breakpoint erop te zetten. Niks mis mee, maar als de waarden weer goed zijn, wordt meestal vergeten deze zogenaamde controle regels te verwijderen, wat een rommelig programma maakt.

Ander soort foutjes kunnen juist geen invloed hebben op de waarden. Deze foutjes zijn vergissingen. Bekijk eens onderstaande code:

```
if a = 0 then
     print "Is gelijk aan nul."
end if
if a > 0 then
     print "Is groter dan nul."
else
     print "Is kleiner dan of gelijk aan nul."
end if
```

In de code zitten geen fouten, maar er is wel een vergissing gemaakt. Deze vergissing zorgt voor slechte code dat eigenlijk niet hoeft. Als variabele 'a' gelijk zal zijn aan nul, zullen er twee regels geprint worden, en wel:

```
Is gelijk aan nul.
Is kleiner dan of gelijk aan nul.
```

Deze vergissing kan op twee gestructureerde manieren worden opgelost. Let op! Niet alle programmeertalen ondersteunen mogelijkheid twee. Er is ook een derde mogelijkheid die op de regel lijkt die in de 'else' staat.

Mogelijkheid 1:

```
if a = 0 then
     print "Is gelijk aan nul."
else
     if a > 0 then
          print "Is groter dan nul."
     else
          print "Is kleiner dan nul."
     end if
```

```
end if
```

Mogelijkheid 2:
```
if a = 0 then
     print "Is gelijk aan nul."
else if a > 0 then
     print "Is groter dan nul."
else
     print "Is kleiner dan nul."
end if
```

Mogelijkheid 3:
```
if a <= 0 then
     print "Is kleiner dan of gelijk aan nul."
else
     print "Is groter dan nul."
end if
```

Het is aan de programmeur zelf hoe de code het best er uit ziet. Geen van de mogelijkheden veroorzaken fouten. Het is de leesbaarheid en de structuur. Wel is mogelijkheid drie niet dezelfde als de andere twee, maar als de programmeur alleen wil weten of een waarde kleiner of gelijk is of groter is, dan is de derde mogelijkheid voldoende. Men kan zelfs die mogelijkheid omkeren. Eerst vergelijken of de waarde groter is dan nul, zodat de andere print regel in de else kan staan. We kunnen de tekst ook aanpassen:

```
if a > 0 then
     print "Is positief."
else
     print "Is niet positief of nul."
end if
```

Hoe dan ook, nooit zouden we met één if code met een else de ‘gelijk aan nul’ kunnen vermijden, tenzij we die negeren, zoals onderstaande code:

```
if a > 0 then
     print "Is groter dan nul."
end if
if a < 0 then
     print "Is kleiner dan nul."
end if
```

Fouten vermijden waar de compiler niets aan kan doen
Er bestaan nog andere soorten fouten die we niet in de code kunnen vinden, behalve als u een slimmerik bent! Deze fouten worden runtime fouten, of ook wel uitvoeringsfouten genoemd.

Deze fouten kunnen optreden door verkeerde invoer, bestanden inlezen die niet bestaan, berekeningen uitvoeren die niet uitgevoerd kunnen worden, en zo kunnen we er meer opnoemen.

Deze fouten kunnen optreden als u het volgende vergeet:

- Controleren of er in een expressie gedeeld wordt door nul;
- Controleren of een opgegeven bestand bestaat;
- Controleren of een bepaalde waarde voldoet aan de variabele van het juiste type. Soms kan zo’n fout wel worden gezien door de compiler, maar het hoeft niet. Tegenwoordig mogen we gewoon een numerieke waarde samenvoegen aan een stringwaarde.

Er zijn twee manieren om fouten te vermijden en zelf meldingen weer te geven:

- Met een IF … THEN … ELSE controle, maar die afhandeling werkt niet altijd naar behoren;
- Met foutafhandeling statements. Sommige programmeertalen hebben die mogelijkheid. Treedt er een fout op, dan zal direct het juiste codeblok uitgevoerd worden. Het wordt ook wel 'exception' genoemd. BASIC kende vroeger de TRAP … RESUME statements om fouten af te handelen. Met 'Exception' wordt er eerst voorafgegaan met 'Try' om de uitvoering te testen. Visual Basic 6 en ouder kende alleen een ON ERROR GOTO <label> en een ON ERROR GOTO 0 om de afhandeling te sluiten. Andere programmeertalen hebben de mogelijkheid een bericht meteen weer te geven als er wat mis gaat. Programmeertaal C++ heeft die mogelijkheid, al is de structuur tegenwoordig wel veranderd en verbeterd. Onderstaand voorbeeld laat een 'thread' afhandeling zien in een oude C++ versie.

Thread afhandeling in C++ zonder een except blok

```
#include <string>
#include <iostream>
#include <thread>

using namespace std;

// De functie die we willen om de nieuwe thread uit te voeren.
void task1(string msg)
{
    cout << "task1 zegt: " << msg << endl;
}

int main()
{
     // Ontdekt een nieuwe thread en start het. Blokkeert de uitvoer niet.
    thread t1(task1, "Hallo");

     // Wacht op de nieuwe handeling tot het verricht is.
     // Is er geen handeling meer, stop dan de thread uitvoering.
    t1.join();
}
```

Tegenwoordig kan het thread statement direct worden gebruikt zonder een functie nodig is. Een bericht achter de thread is dan voldoende.

Regels splitsen

In de meeste programmeertalen kunnen we lange regels splitsen. Programmeertaal C is daar een voorbeeld van. Voor de compiler maakt het niet uit of de regel compleet of gesplitst is. Het controleert alleen of de regel beëindigd wordt met een puntkomma.

```
if (a == 0)
     printf("Is gelijk aan nul.");
```

Oudere BASIC programmeertalen hebben niet de mogelijkheid om regels te kunnen splitsen. Het probleem is de interpreter, die het nadeel hiervan is.

Toen QuickBASIC kwam, bracht Microsoft er verandering in. Er kwam een methode zoals de bovenstaande drie mogelijkheden, maar wel zonder een symbool als een puntkomma. Daarna is BASIC op die manier verder gegaan. Maar Microsoft kwam niet als enige met die verandering. Commodore was de eerste waarmee codeblokken in een IF … THEN … ELSE structuur kon worden geprogrammeerd. Niet met een END IF, maar met een BEGIN … BEND blok; een structuur die wat op Pascal lijkt.

Maar Microsoft ging verder. BASIC moest achter de feiten aan en er kwam een onderstreep symbool om regels te kunnen splitsen. Ook een idee om zoveel mogelijke END IF statements te vermijden en parameters van procedures en functies konden onder elkaar worden gezet. Een goed voorbeeld is Liberty BASIC, waarmee gestructureerd programmeren een feit is.

Labels
Regelnummers waren een doorn in het oog als het om programmeren ging. Na elke regel moest er een grote stap worden gemaakt voor een regelnummer. Meestal was dat in stappen van tien. Toch was dat niet altijd voldoende. Het probleem werd alleen maar groter als er naar regelnummers gesprongen moest worden. GOTO statements waren een ramp en de enige redding om gestructureerd te programmeren waren de subroutines.

In Liberty BASIC hoeven we niet naar een andere regel te gaan met een GOTO of een GOSUB. Met de labels springen we naar bepaalde codeblokken, waarmee met een naam tussen rechte haken begint. Meestal zijn dat de events van de controls. Deze events worden met een optionele ‘print’ statement aangeroepen. Zie meer daarover in de API onderwerpen van Liberty BASIC.

Er waren ook oudere BASIC programmeertalen en dialecten die al labels kenden. Helaas moesten die wel aangeroepen worden met een GOTO of een GOSUB. Het voordeel was dat we ons niet hoefden te bekommeren om de regelnummers.

Python – Samengestelde statements, regels en inspringen.

Samengestelde statements bevatten (groepen van) andere statements; ze beïnvloeden de controle op de uitvoering van die andere statements op een bepaalde manier. Samengestelde statements omspannen meerdere regels, hoewel in eenvoudige incarnaties een hele samengestelde instructie in één regel kan worden opgenomen.

De if, while en for statements implementeren traditionele control flow constructies. Het try statement geeft exception handlers aan en/of verschoont de code voor een statementgroep, terwijl het with statement de initialisatie uitvoer en de laatste code rondom een codeblok ondersteunt. Functie en class definities zijn ook syntaxis gezien samengestelde statements.

Samengestelde statements bestaan uit één of meer 'clausules.' Een clause bevat een kop en een 'suite'. De clause van samengestelde statements zijn allemaal op hetzelfde inspringend niveau. Elke clause kop begint met een uniek geïdentificeerd sleutelwoord en eindigt met een dubbelepunt. Een suite is een groep statements gecontroleerd door een clause. Een suite kan één of meer eenvoudige statements, eindigend met een puntkomma, op dezelfde regel hebben, gevolgd door de dubbelepunt bij de kop, of het kan op meerdere regels met één of meer statements worden ingedeeld. Alleen de laatste suitevorm kan samengestelde statements bevatten; het volgende is illegaal, vooral omdat het zo niet duidelijk is aan welke, als clause, een volgende else-clause zou toebehoren:

```
if test1: if test2: print(x)
```

Merk ook op dat de puntkomma strakker dan de dubbele punt in dit verband bindt, zodat in het volgende voorbeeld geen of alle print() aanroepen worden uitgevoerd:

```
if x < y < z: print(x); print(y); print(z)
```

Het if statement wordt gebruikt voor conditionele uitvoer:

```
if <conditie[s]>: <suite>
     [elif <conditie[s]>: <suite>]
     [else: <suite>]
```

De condities zullen één voor één worden geëvalueerd totdat er een gevonden is die waar is; die suite wordt dan uitgevoerd en geen ander deel van het if statement wordt uitgevoerd of geëvalueerd. Zijn alle condities onwaar, dan zal de suite van de else clause, indien aanwezig, worden uitgevoerd.

Het while statement wordt gebruikt voor herhaalde uitvoer zolang een conditie waar is:

```
while <conditie[s]>: <suite>
     [else: <suite>]
```

De conditie wordt herhaaldelijk getest en als het waar is wordt de eerste suite uitgevoerd. Is de conditie onwaar (waarbij mogelijk de eerste keer is getest), zal de suite van de else clause, indien aanwezig, worden uitgevoerd en de lus uitvoer wordt beëindigd.

Een break statement die uitgevoerd wordt in de eerste suite, beëindigd de uitvoer van de lus zonder de suite van de else clause uit te voeren. Een continue statement die uitgevoerd wordt in de eerste suite, slaat de rest van de suite over en gaat terug om weer de conditie te testen.

Het for statement wordt gebruikt om een reeks elementen te herhalen (zoals een string, tupel of een lijst) of een ander object:

```
for <doellijst> in <expressielijst>: suite
     [else: <suite>]
```

De expressielijst wordt éénmaal geëvalueerd; het moet een herhaald object opleveren. Een resultaat van de expressielijst bepaalt de herhaling. De suite wordt per herhaling één keer uitgevoerd, in de volgorde van de oplopende indexen. Elk item wordt toegekend aan de doellijst bij gebruik van de standaardregels voor toekenningen en dan wordt de suite uitgevoerd. Als er geen items zijn of een herhaling stuit op een StopIteration exceptie, zal de else clause, indien aanwezig, uitgevoerd worden en de lus uitvoer zal worden beëindigd.

Een break statement die in de eerste suite uitgevoerd wordt, beëindigt de lus uitvoer zonder het uitvoeren van de suite van de else clause. Een continue statement uitgevoerd in de eerste suite slaat de rest van de suite over en gaat verder met het volgende item, of met de else clause als er geen volgende item was.

De suite mag toekenningen uitvoeren naar de variabele(n) in de doellijst; dit heeft geen invloed op het volgende item dat toegekend is.

De namen in de doellijst zijn niet verwijderd als de lus met de uitvoer klaar is, maar als de herhaling leeg is, dan zal het niet helemaal door de lus zijn toegewezen. De ingebouwde functie range() geeft een iteratie van integers terug die net zo geschikt is als de soortgelijke Pascal vorm:

```
     for i := a to b do
```

als Python die een lijst

```
     list(range(3))
```

de lijst `[0, 1, 2]` teruggeeft.

Python maakt geen gebruik van codeblokken zoals we in andere programmeertalen gewend zijn. Eerder gaf ik aan dat een clause eindigt met een dubbelepunt en dan pas de suite begint. Dat zijn meerdere statements in één blok, waarvan elk statement eindigt met een puntkomma.

Het kan in Python snel onleesbaar worden als we op bovenstaande dingen niet goed letten. Het is toegestaan de suite naast een clause op één regel te zetten, maar worden de programma's groter dan is het niet meer goed te overzien.

De oorzaak is dus het geen gebruik maken van de accolades om codeblokken te vormen. Codeblokken worden ingedeeld bij het inspringen van de regels.

Het aantal ruimtes in ingesprongen regels zijn variabel, maar alle statements in een blok moeten dezelfde ingesprongen afstand hebben, bijvoorbeeld:

```
if True:
     print("Waar")
else:
     print("Onwaar")
```

Het volgende blok veroorzaakt echter een fout:

```
if True:
          print("Antwoord")
          print("Waar")
```

```
else:
          print("Antwoord")
     print("Onwaar")
```

Dus, in Python zijn alle volgende regels, ingesprongen met hetzelfde aantal ruimtes, een blok. Het volgende voorbeeld heeft verschillende statementblokken:

U hoeft onderstaande logica niet te begrijpen. Kijk goed of u de variatie van de blokken kunt begrijpen zonder de welbekende accolades.

```
#!usr/bin/python

import sys

try:
     # open file stream
     file = open(file_name, "w")
except IOError:
     print("There was an error writing to", file_name)
     sys.exit()
file_finish = "|"
print("Enter '", file_finish,
     "' When finished")
while file_text != file_finish:
     file_text = raw_input("Enter text: ")
     if file_text == file_finish:
          continue
     file.write(file_text)
     file.write("\n")
# close the file
file.close()
file_name = raw_input("Enter filename: ")
if len(file_name) == 0:
     print("Next time please enter something")
     sys.exit()
try:
     file = open(file_name, "r")
except IOError:
     print("There was an error reading file")
     sys.exit()
file_text = file.read()
file.close()
print(file_text)
```

De statements eindigen in Python met een newline. Python kent het gebruik van voortzetting van de lijn met een (\) teken. Daardoor weet Python dat een regel verder moet gaan, bijvoorbeeld:

```
totaal = eerste_item + \
         tweede_item + \
         derde_item
```

De statements die binnen de rechte haken, accolades of haakjes zijn opgenomen hoeven de voortzetting met een (\) teken niet te hebben, bijvoorbeeld:

```
dagen = ['maandag', 'dinsdag', 'woensdag', 'donderdag', 'vrijdag']
```

PowerBASIC – Pointers (@).

Een pointer is een variabele die een 32-bits (4 byte) adrescode of gegevenslocatie ergens in het geheugen vasthoudt. Het wordt een pointer genoemd, omdat het *verwijst* naar die locatie. De locatie waarnaar verwezen wordt is het *doel* van de pointer. Pointers vertegenwoordigen een sterke toevoeging aan de BASIC programmering. Het adres wordt gedefinieerd tijdens de uitvoer. Uw programma kan naar elke geheugenlocatie verwijzen, alsof het een standaard variabele is. Zodra een pointer gebruikt wordt om toegang te krijgen tot een geheugenlocatie, wordt het 'Indirect adresseren' genoemd.

Pointers worden gedeclareerd met gebruik van het DIM statement en een doeltype moet opgegeven worden. De sleutelwoorden PTR en POINTER doen hetzelfde.

```
DIM i AS INTEGER PTR     'declareert i als een pointer naar een Integer
DIM i AS INTEGER POINTER 'declareert hetzelfde als PTR
```

De bovenste twee regels declareren i als een Integer pointer. Voordat het gebruikt kan worden, moet het geïnitialiseerd worden met een actueel adres van een variabele (met de VARPTR functie of STRPTR voor strings). Als u een variabele toekent aan een pointer variabele, geeft u het een adres voor later gebruik wanneer u wenst een actueel doel te verwijzen. Alleen de pointernaam verwijst de pointervariabele. Een naam van de pointer met een @ symbool prefix verwijst naar het doel van de pointer:

```
DIM Ptr1 AS BYTE PTR     'declareert Ptr1 als een byte pointer
DIM Ptr2 AS BYTE PTR     'declareert Ptr2 als een byte pointer
DIM Byte1 AS BYTE        'declareert Byte1 als een byte variabele
DIM Byte2 AS BYTE        'declareert Byte2 als een byte variabele
Ptr1 = VARPTR(Byte1)     'Ptr1 verwijst naar Byte1
@Ptr1 = 36               'Kent de waarde 36 toe aan Byte1
Ptr2 = VARPTR(Byte2)     'Ptr2 verwijst naar Byte2
@Ptr2 = @Ptr1 + 4        'Kent de waarde 40 (36 + 4) toe aan Byte2
```

Verwijst u een pointer *zonder* een apenstaartje @, dan verwijst u naar het 32-bit adres met de inhoud erbij. Als u de naam *met* een apenstaartje @ doet, dan verwijst u naar de doellocatie van het adres 'verwezen' door de pointer.

Bij het toekennen van het adres van een andere pointer naar een pointer, kunnen we op een ander niveau instellen. Pointers naar pointers zijn nuttig bij het opzetten van gekoppelde lijsten in het geheugen. U kunt dan toegang krijgen tot het doel door een tweede apenstaartje voor de naam van de pointer toe te voegen:

```
DIM y AS STRING POINTER
DIM z AS STRING POINTER
DIM TmpStr AS STRING
y = VARPTR(TmpStr)  'y verwijst naar TmpStr
z = VARPTR(y)       'z verwijst naar y
@y = "A"            'plaatst een "A" in TmpStr
@@z = "B"           'overschrijf het met een "B"
Display @y          'geef de doelwaarde van y weer
```

PowerBASIC ondersteunt tot 200 niveaus. Voor elk niveau voegt u een nieuw apenstaartje @ toe aan de pointernaam. U kunt alleen het @ prefix gebruiken met pointer variabelen.

Een pointer met een nul waarde is in PowerBASIC een null-pointer. Windows genereert een General Protection Fault (GPF) als u probeert toegang te krijgen tot een ongeldig pointer adres.

De kracht van pointers bevindt zich in de snelheid en flexibiliteit. Normaal was het zo, voor geheugentoegang, dat een BASIC programmeur combinaties moest gebruiken met PEEK en POKE. De programmeur kon hierdoor naar een geheugenadres gegeven in bytes verwijzen. Nam het doel met de gegevens een andere vorm aan, dan moest er geconverteerd worden. Pointers accepteren u elke doelgroep op een bepaalde wijze te adresseren, zelfs met een gebruiker gedefinieerd structuur. Handiger nog, omdat hiermee het aanroepen van PEEK en POKE niet langer nodig is; de toegang is veel sneller.

Laten we zeggen dat we alle karakters in een buffer willen en alle hoofdletters in kleine letters willen hebben. De code kan dan er als volgt uit zien:

```
SUB Lower(zStr AS STRING)
     DIM s AS BYTE PTR, ix AS INTEGER
     s = STRPTR(zStr)    'Geeft direct toegang tot de dynamische string
     FOR ix = 1 TO LEN(zStr)
          IF @s = 65 THEN @s = 97  ' A -> a
          INCR s
     NEXT
END SUB
```

Als een pointer wordt gebruikt naar een structuur, staat de prefix voor de structuurnaam wanneer u wilt om toegang tot een element van de structuur te krijgen. De structuurnaam zelf refereert tot het adres. Dit onderscheid is belangrijk bij de behandeling van structuren als een geheel. Het volgende voorbeeld laat twee manieren zien hoe een simpele dubbele sortering wordt gemaakt met een array van gebruiker gedefinieerde types. De eerste gebruikt conventionele BASIC methoden. De tweede gebruikt pointers om de snelheid en efficiëntie te illustreren.

```
'-- Voorbeeld 1 --
#COMPILE EXE
#DIM ALL
TYPE NameRec
      Last AS STRING * 20     'Achternaam
      First AS STRING * 20    'Voornaam
END TYPE
FUNCTION PBMAIN() AS LONG
      DIM Rec(1 TO 10) AS NameRec
      DIM ix AS LONG, ij AS LONG
      '-- Plaats wat gegevens in de records --
      FOR ix = 1 TO 10
            Rec(ix).First = CHOOSE$(ix,"Jacob","Michael","Joshua","Matthew", _
                                    "Ethan","Emily","Emma","Madison","Abigail","Olivia")
            Rec(ix).Last = CHOOSE$(ix,"SMITH","JOHNSON","WILLIAMS","JONES", _
                                    "BROWN","DAVIS","MILLER","WILSON","MOORE","TAYLOR")
      NEXT ix
      '-- Sorteer UDT array in oplopende volgorde bij gebruik van dubbel sorteren
      '-- ARRAY SORT Rec(),FROM 1 TO 10,ASCEND doet dit normaal
      FOR ix = 9 TO 1 STEP -1
            FOR ij = 1 TO ix
                  IF Rec(ij-1).Last > Rec(ij).Last THEN
                        SWAP Rec(ij-1), Rec(ij)
                  END IF
            NEXT ij
      NEXT ix
      #IF %DEF(%PB_CC32)
            FOR ix = 1 TO 10
                  PRINT TRIM$(Rec(ix).Last) + ", " + TRIM$(Rec(ix).First)
            NEXT ix
            PRINT
            PRINT "Press any key to quit ... "
            WAITKEY$
```

```
    #ELSE
          DIM msg AS STRING
          FOR ix = 1 TO 10
                msg = msg + TRIM$(Rec(ix).Last) +", " + TRIM$(Rec(ix).First) + $CRLF
                MSGBOX msg
          NEXT ix
   #ENDIF
END FUNCTION

'-- Voorbeeld 2 --------------
' Het verschil met voorbeeld 1 en dit voorbeeld is dat pointers (4 bytes)
' worden toegepast op hele records (40 bytes).
#COMPILE EXE
#DIM ALL
TYPE NameRec
  Last   AS STRING * 20   ' Achternaam
  First  AS STRING * 20   ' Voornaam
END TYPE
FUNCTION PBMAIN () AS LONG
    DIM Rec(1 TO 10) AS NameRec
    DIM RP AS NameRec POINTER
    DIM ix AS LONG, ij AS LONG
    DIM hFile AS DWORD

    '-- Plaats wat gegevens in de records --
    FOR ix = 1 TO 10
        Rec(ix).First = CHOOSE$(ix,"Jacob","Michael","Joshua","Matthew","Ethan", _
                                "Emily","Emma","Madison","Abigail","Olivia")
        Rec(ix).Last  = CHOOSE$(ix,"SMITH","JOHNSON","WILLIAMS","JONES", _
                                "BROWN","DAVIS","MILLER","WILSON","MOORE","TAYLOR")
    NEXT ix
    '-- Sorteer UDT array in oplopende volgorde met gebruik van dubbel sorteren
    '-- met pointers.
    '-- Onthoud dat dubbel sorteren niet aanbevolen is voor grote collecties
    '-- en begrijp dat ARRAY SORT Rec(),FROM 1 TO 20,ASCEND dit normaal doet.
    '-- Dit is alleen om pointers naar UDT arrays in actie te laten zien!
    RP = VARPTR(Rec(1))
    FOR ix = 9 TO 1 STEP -1
        FOR ij = 1 TO ix
            'Merk op dat pointers naar array elementen vanaf nul beginnen
            'en tussen rechte haken moeten staan!
            IF @RP[ij-1].Last > @RP[ij].Last THEN
                SWAP @RP[ij-1], @RP[ij]
            END IF
        NEXT ij
    NEXT ix
   #IF %DEF(%PB_CC32)
        FOR ix = 1 TO 10
            PRINT TRIM$(Rec(ix).Last)+ ", " +TRIM$(Rec(ix).First)
        NEXT ix
        PRINT
        PRINT "Press any key to quit ... "
        WAITKEY$
   #ELSE
       DIM msg AS STRING
       FOR ix = 1 TO 10
           msg = msg + TRIM$(Rec(ix).Last)+ ", " +TRIM$(Rec(ix).First) + $CRLF
           MSGBOX msg
       NEXT ix
   #ENDIF
END FUNCTION
```

Als u een member van een structuurtype declareert als een pointer, wordt het @ prefix gebruikt met de membernaam; niet de naam van het structuurtype. Het vorige voorbeeld kan worden verbeterd door het toevoegen van een paar pointers naar de structuur te maken om die naar het vorige en volgende record te laten verwijzen. Hiermee kunt u geheugen toewijzen voor een record alleen wanneer nodig is, in plaats van vooraf toekenning van een vaste grootte matrix van records. De gewijzigde structuur zou er ongeveer als volgt uitzien:

```
TYPE NameRec
   Last AS STRING * 20      ' Achternaam
   First AS STRING * 20     ' Voornaam
   Nxt AS NameRec PTR       ' Pointer naar volgend record
   Prv AS NameRec PTR       ' Pointer naar vorig record
END TYPE
DIM Rec AS NameRec
```

De pointer members zijn dan toegankelijk als deze:

```
Rec.@Nxt     ' volgend record
Rec.@Prv     ' vorig record
```

Het plaatsen van het @ prefix aan de voorkant van de structuurnaam (d.w.z. @*Rec*) zal een compile fout veroorzaken. *Rec* is zelf geen pointer.

Bij het berekenen van de lengte van het type zijn alle pointers intern opgeslagen als dubbelwoord (DWORD) variabelen, dus *NameRec* is 48 bytes lang (20 + 20 + 4 + 4). Wilt u de lengte weten van een type, dan is het gemakkelijker om PowerBASIC het voor je te laten berekenen met gebruik van de LEN functie dan het zelf te doen:

```
length = LEN(structure)
```