Liberty BASIC v4.5.1
Over de Drempel

Een Liberty BASIC E-Boek voor beginners en gevorderden
Met veel voorbeelden en afbeeldingen

Email: m.a.kurvers@tronicasoftware.nl

Forum: Home | Makurvers (freeforums.net)

Inhoud

1. Deel 1: Introductie en Liberty BASIC

Welkom Liberty BASIC gebruikers.

Dit boek zal een goede instap zijn om te leren programmeren. Je kunt zowel vanaf het begin van het boek beginnen als ergens in het boek beginnen. Wat je zelf wilt.
Het boek bestaat uit drie delen: de BASIC programmeertaal, de instap in Liberty BASIC en Liberty BASIC van een andere kant gezien. Dit is ook de reden waarom ik het boek 'Over de Drempel' noem, omdat we met Liberty BASIC op twee manieren kunnen programmeren.

Voor het eerste deel van het boek heb ik ondersteuning gehad met een boek genaamd 'MSX BASIC handboek voor iedereen' geschreven door A.C.J. Groeneveld. ISBN: 9063981007
Met dank aan: Uitgeverij Stark Texel

In dit boek zul je regelmatig opdrachten tegenkomen. Probeer ze te maken. In de appendix staan de opdrachten uitgewerkt.

1a. Wat is programmeren

Programmeren is **het schrijven van een computerprogramma**, een concrete reeks instructies (code) die een computer kan uitvoeren. Dit is de taak van een softwareontwikkelaar of programmeur. Programmeren wordt in het algemeen niet direct in machinetaal gedaan, maar in een programmeertaal. Zulke programmeertalen zijn hogere programmeertalen. Zij hebben een compiler of een interpreter die de code moeten vertalen naar machinetaal. Doordat er hogere programmeertalen zijn, kan men zijn eigen programma's schrijven. Een hogere programmeertaal is voor ons beter te begrijpen dan de machinetaal.

Er bestaan verschillende programmeertalen, o.a. Pascal, Python, C, Cobol en uiteraard BASIC. Er zullen nog wel meer programmeertalen bestaan, maar er is er één dat in dit boek aanwezig is: BASIC.

1b. De programmeertaal BASIC

De naam van de programmeertaal BASIC is een afkorting van Beginners All-purpose Symbolic Instruction Code. Zoals de naam doet vermoeden, werd de taal BASIC in eerste instantie ontworpen voor beginners op het gebied van de computerwereld. Sinds de introductie van BASIC is er door diverse software-ontwerpers een keur van zogenaamde dialecten ontworpen zodat het nu niet meer mogelijk is om van een standaard BASIC te spreken. De tegenwoordige BASIC-dialecten zijn vaak zeer uitgebreid en lijken niet meer op het eerste, eenvoudige, BASIC waar alle nu meer dan 300 verschillende dialecten van afstammen.

Naast de BASIC-dialecten zijn er ook BASIC-versies. Deze BASIC programmeertalen stammen niet meer af van het standaard BASIC, maar zijn helemaal vernieuwd. Om toch het als een BASIC programmeertaal te beschouwen, hebben ze zoveel mogelijk van het standaard BASIC overgenomen. Eén van de BASIC versies is **Liberty BASIC**. Een eenvoudige BASIC programmeertaal waarmee we Windows programma's kunnen schrijven, maar toch op de manier zoals BASIC is.

Om kennis te krijgen in programmeren in BASIC, zal Liberty BASIC een fijne leerzame BASIC programmeertaal zijn. Dit boek gaat voornamelijk over de programmeertaal Liberty BASIC, maar in de laatste hoofdstukken ook over wat Liberty BASIC voor nog meer mogelijkheden heeft om goede programma's te kunnen schrijven.

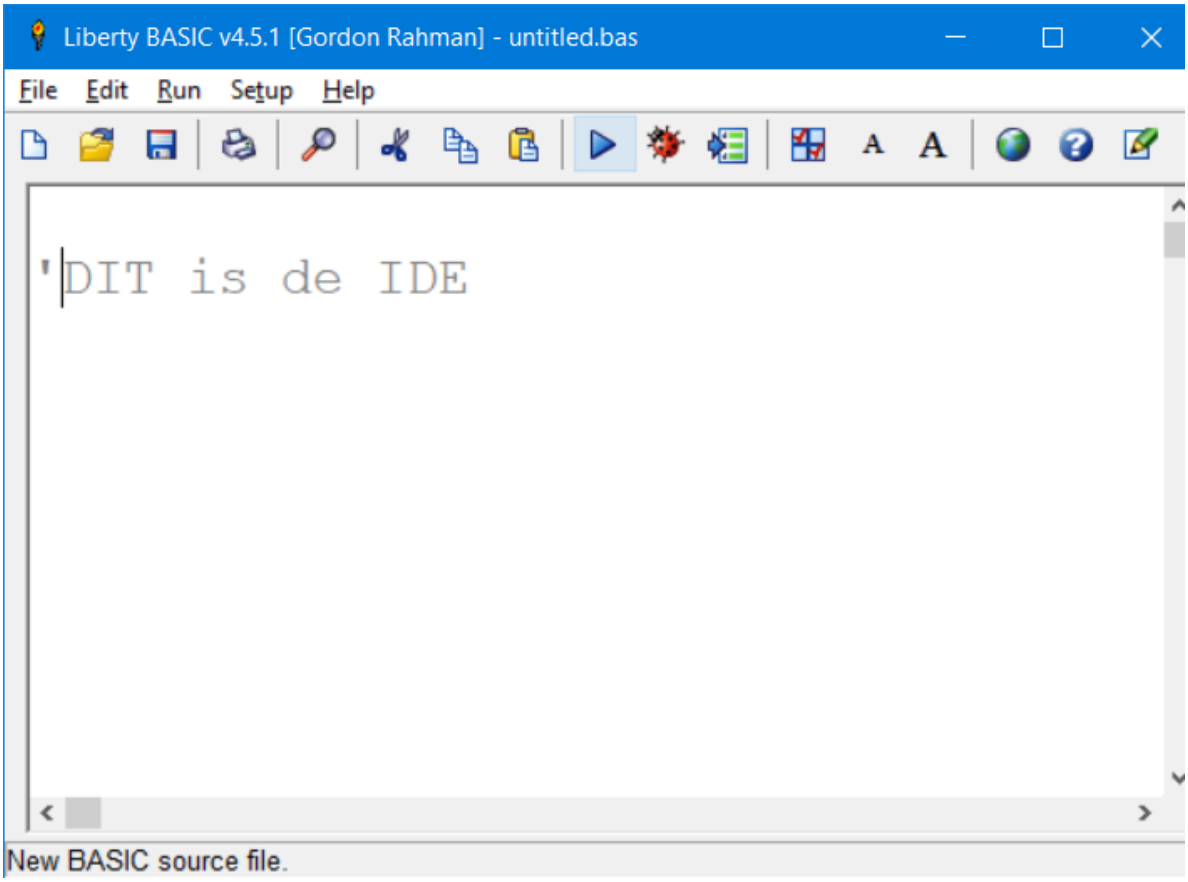
Als je Liberty BASIC nog niet hebt, ga dan naar de site www.libertybasic.com en download **Liberty BASIC.** Liberty BASIC is shareware. Het is gratis, maar het wordt aangeraden om het te kopen als het bevalt. Om de voorbeelden uit dit boek uit te proberen gebruik je Liberty BASIC.

Heel veel plezier met dit boek,

Marco Kurvers

1c. Liberty BASIC verkennen

Als we Liberty BASIC starten, zien we een venster met een knipperende cursor, een menu en een knoppenbalk. Dit is de IDE (Integrated Development Environment) van Liberty BASIC, de “programmeurs ontwikkel omgeving”. In het venster, waar de cursor in knippert, kunnen we BASIC of Liberty BASIC programma’s schrijven.



Programmacode wordt niet direct uitgevoerd. Wat we intypen moeten we starten door naar het menu Run te gaan of door op de blauwe pijl te klikken op de knoppenbalk.

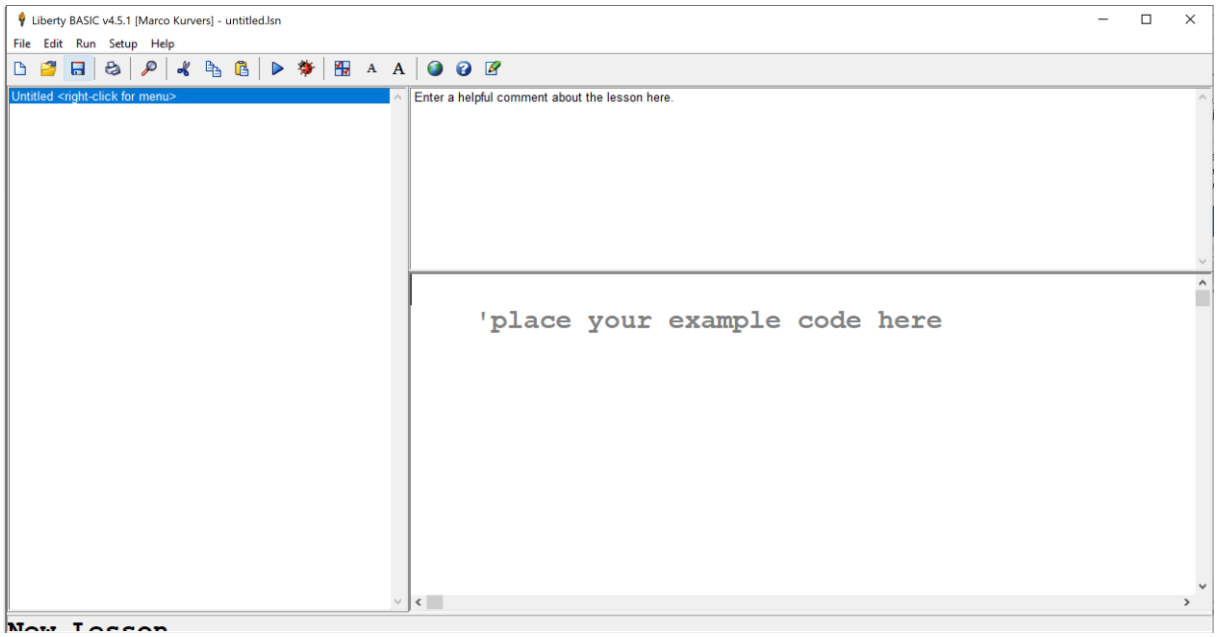
De bovenste blauwe balk van het venster heet de “titelbalk”. Daaronder volgt de Menubalk. En daaronder de knoppenbalk.

Naast de bekende knoppen die we in alle software applicaties wel tegenkomen, zien we ook nieuwe knoppen. Naast de start knop (de blauwe pijl) zien we een lieveheersbeestje. Deze knop zorgt ervoor dat we de code kunnen debuggen, anders gezegd: kunnen controleren wat de code doet tijdens de uitvoer. Hiermee is het mogelijk fouten te vinden als we ze niet in de code zelf kunnen vinden. Zie verder daarover hoofdstuk 12.

De knop rechts van het lieveheersbeestje is de Preferences knop, ook te bereiken via het menu Setup. In dit venster kunnen bepaalde instellingen verricht worden, zoals het aan- of uitzetten van de syntax-kleurcodering en het instellen van de bestandsextensie. Standaard worden de programma’s opgeslagen met de BAS extensie, maar dit kan gewijzigd worden.
De overige knoppen is het verkleinen en vergroten van de code die je typt, om naar Liberty BASIC website te kunnen, de HELP over Liberty BASIC te kunnen openen en de release notes te kunnen bekijken.

Met menu-item New in het menu File kunnen we twee mogelijkheden kiezen waar we mee willen beginnen: een BASIC Source File of een BASIC Lesson.

Met een BASIC Source File keuze, komen we in de IDE zoals te zien is in bovenstaande afbeelding.
Met de keuze BASIC Lesson, komen we in een wat uitgebreidere IDE, zoals hieronder te zien is.



Zoals je ziet kan de titelbalk van kleur anders zijn. Het ligt eraan met welke Windows besturingssysteem je werkt.

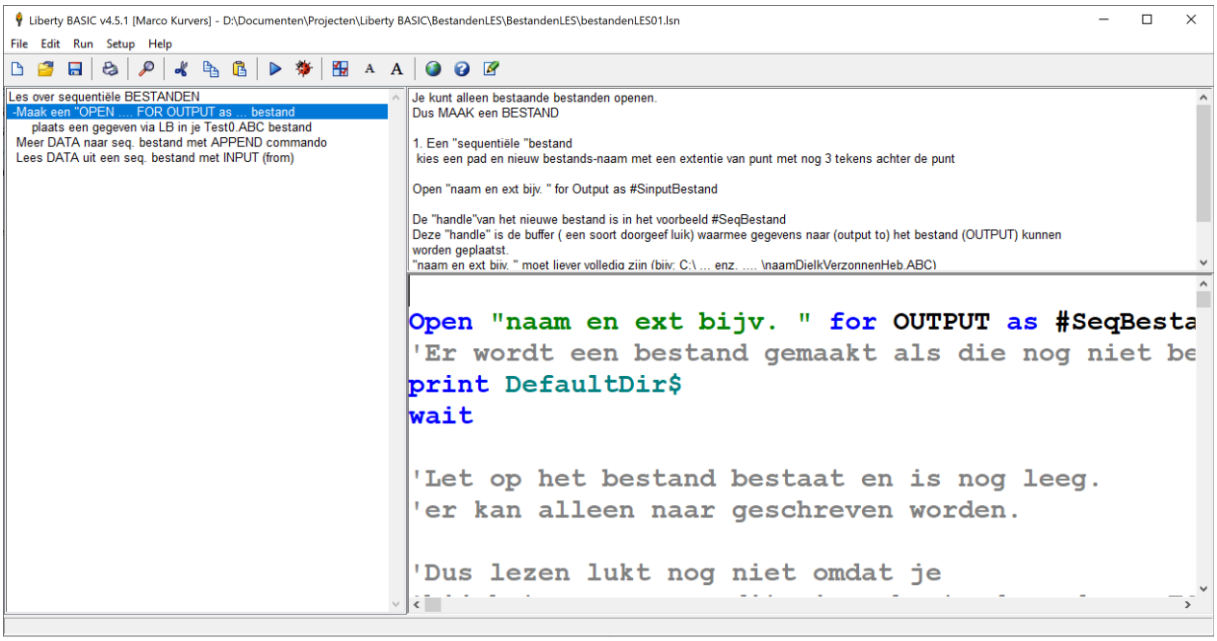
Links is een venster waar je een lessenlijst kunt aanmaken. Het is een boomvenster, omdat de lijst verdeeld kan worden als een mappenstructuur, zoals de Verkenner. Elk hoofditem kan uit meer andere items bestaan en het hoofditem kan ook inhoud hebben.

Het bovenste venster rechts is een uitlegvenster. Hier noteer je uitleg of informatie over de code die in het venster onderaan geschreven wordt.

Op die manier kunnen lessen gemaakt worden en elk item die je in het linker venster kiest kan een eigen les hebben. Wil je geen code laten zien of helemaal niets weergeven, dan mogen de rechter vensters ook leeg blijven.

Elk item die je kiest en code heeft, kun je uitvoeren. Het is niet zo dat alle code van alle items uitgevoerd worden.

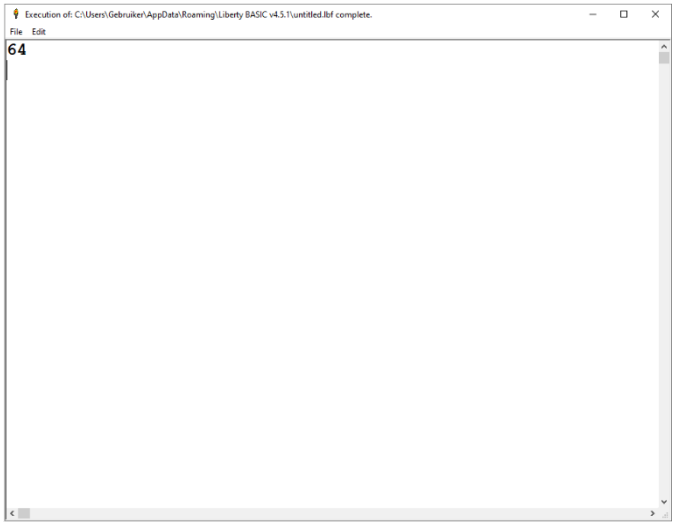
Hieronder zie je een afbeelding met volledige inhoud.



1d. Het mainwin venster

Liberty BASIC kent nog een ander venster. Dit venster wordt vooral door beginners als uitvoervenster gebruikt. Je kunt het zien als een soort console venster waar de uitvoer wordt getoond wanneer we een programma runnen. Dit venster wordt het **mainwin** venster genoemd.

We kunnen de gegevens op het mainwin venster weergeven op dezelfde manier als dat je in andere BASIC programmeertalen doet. Je zorgt voor goede uitvoer, zoals tekst. Je stelt vragen, vraagt om invoer en je voert berekeningen uit. In plaats van het console venster, gebruiken we nu het mainwin venster.



Het mainwin venster ziet er net zo uit als een normaal Windows venster, zie afbeelding.

Door in Liberty BASIC een opdracht te geven om het getal 64 af te drukken (te printen), zal het linksboven op het mainwin venster verschijnen.

Je ziet ook de cursor onder het getal staan. Maar het is niet mogelijk om in het venster te kunnen typen. Je kunt het alleen lezen, maar niet beschrijven.

Wel is het mogelijk om uitvoer in het mainwin venster te knippen, te kopiëren en te plakken.

Je kunt de uitvoer selecteren en met de Delete toets verwijderen. De Backspace toets werkt niet in de mainwin.

Ook is het mogelijk om de uitvoer als een tekstbestand op te slaan. Je kunt het zelfs uitprinten. Deze mogelijkheden zijn op de mainwin gewoon beschikbaar.

Met het welbekende PRINT commando kunnen we in Liberty BASIC nog steeds hetzelfde doen zoals we gewend zijn in andere BASIC programmeertalen.

Leren programmeren voor beginners kan in Liberty BASIC een goed begin zijn. Liberty BASIC kent veel mogelijkheden die we ook vinden in andere BASIC versies en dialecten. Deze beginselen kunnen we met z'n allen doornemen en kijken hoe we dit in Liberty BASIC kunnen gebruiken.

We beginnen met het PRINT commando. Een welbekend commando waar elk BASIC boek mee begint.

1e. Het PRINT commando

Start Liberty BASIC, zorg voor een nieuw leeg editorvenster en typ onderstaand statement in:

```
PRINT "Hallo Allemaal"
```

Druk op Enter.

Zoals we in eerdere BASIC programmeertalen gewend zijn, zou het PRINT commando direct uitgevoerd moeten worden. Liberty BASIC doet dat echter niet. Alles wat we intypen kunnen we pas uitvoeren als we op de RUN knop (blauwe pijl) hebben geklikt.

Klik op de blauwe pijl.

Als je niets ziet, schuif dan andere vensters even opzij. Het kan zijn dat het mainwin venster erachter ligt.

We droegen de computer op om een tekst af te drukken (PRINT betekent afdrukken); de computer gehoorzaamde ons feilloos.

Probeer ook de volgende regel eens:

```
PRINS "Hallo Allemaal"
```

Iedere BASIC-programmeur begrijpt wat er bedoeld wordt. Echter, de computer – tenslotte maar een dom instrument – begrijpt niet dat het sleutelwoord eigenlijk PRINT had moeten zijn en geeft een **Syntax error** (fout in schrijfwijze).

Hieruit zal één ding duidelijk zijn:

De computer is een dom instrument en als zodanig niet in staat om fouten te maken. De computer gehoorzaamt je feilloos wanneer je in staat bent om de opdrachten precies volgens de voorschriften in zijn taal te formeren.

Opdracht: Typ eens onderstaande PRINT regels in om het PRINT commando te leren kennen. Kijk op het venster wat er gebeurt en probeer de code met de uitvoer te begrijpen:

```
PRINT "Half om half"
PRINT "Wat een kanjer!"
PRINT 25
```

Vroeger in BASIC moesten de regels getypt worden beginnend met regelnummers om programma's te kunnen schrijven. Zonder een regelnummer werd de regel (zoals een PRINT commando) direct uitgevoerd. In Liberty BASIC is dit niet meer het geval, maar regelnummers mogen nog steeds worden gebruikt.

Probeer eens onderstaande regel:

```
10 PRINT "Hallo Allemaal"
```

Als je deze regel uitvoert dan gehoorzaamt Liberty BASIC nog steeds en print de tekst feilloos uit. Liberty BASIC ondersteunt nog steeds regelnummers, maar onderschat niet de regelnummers uit de oude BASIC talen. Liberty BASIC ziet een regelnummer anders. Daardoor is de volgorde van de regelnummers ook niet meer belangrijk. Later in het boek kom ik erop terug.

We kunnen het kleine programmaatje uitbreiden met nog een regel:

```
10 PRINT "Hallo allemaal"
20 PRINT "Hoe gaat het ermee?"
```

Als we dit gaan uitvoeren, zal in het mainwin venster het volgende verschijnen:

```
Hallo allemaal
Hoe gaat het ermee?
```

We zien dat, na het uitvoeren, de twee opgedragen teksten keurig onder elkaar worden afgedrukt. Blijkbaar werkt de computer de ingetoetste programmaregels op volgorde af, en vroeger was dat zelfs op volgorde van de regelnummers, totdat alle regels 'op' zijn. We kunnen het programma op deze wijze willekeurig groter maken en steeds weer laten uitvoeren.

Een leuk grapje kan men uithalen door het programma als volgt uit te breiden:

```
10 PRINT "Hallo allemaal"
20 PRINT "Hoe gaat het ermee?"
30 GOTO 10
```

Na op de uitvoerknop te hebben geklikt, blijft de tekst maar op het venster verschijnen. Het commando GOTO 10 zorgt ervoor, dat wanneer de computer bij regel 30 is aangekomen, hij weer 'teruggaat' naar regel 10 waardoor het programma geen einde kan vinden. Nogmaals blijkt dat de computer een dom instrument is dat de moeilijkste maar ook de domste en meest onzinnige opdrachten zonder protest uitvoert. De enige voorwaarde is, dat de juiste formulering is gebruikt.

Het steeds maar ronddraaiende programma kan door de Control toets samen met de Pause/Break toets stopgezet worden. Liberty BASIC zal wel een *keyboard interrupt* melding geven. Het is een foutmelding. Liberty BASIC vindt het kennelijk niet fijn dat een programma op die manier afgebroken wordt.

Vervang regel 10 nu eens door:

```
10        PRINT "Hallo allemaal"
```

Voer het programma nog eens uit. Liberty BASIC is niet gevoelig voor spaties, maar probeer eens alle spaties te verwijderen:

```
10PRINT "Hallo allemaal"
```

Nu zal Liberty BASIC een Syntax error melding geven. We moeten dus, als we een regelnummer willen gebruiken, deze met een commando scheiden met een spatie.
Overigens is de spatie tussen het PRINT commando en het aanhalingsteken niet verplicht.

Ook wanneer we het PRINT commando in kleine letters schrijven, zal het programma nog steeds uitstekend werken.
We kunnen concluderen dat het gebruik van kleine of grote letters bij sleutelwoorden niets uitmaakt.

Het zal duidelijk zijn dat het programma wel anders gaat werken wanneer we de tekst tussen de aanhalingstekens in kleine letters of allemaal in hoofdletters gaan zetten.

We kunnen vaststellen dat:

- Sleutelwoorden met kleine en grote letters mogen worden geschreven.
- Een sleutelwoord aan elkaar dient te worden geschreven: PRINT in plaats van PRI NT.
- Er tussen aanhalingstekens precies de tekst dient te worden opgenomen die ook moet worden gebruikt.
- Er verder naar believen spaties mogen worden gebruikt.

Alhoewel de principes van het programmeren vrij eenvoudig zijn, zal het door het grote aantal sleutelwoorden toch een behoorlijke oefening vergen voordat je Liberty BASIC volledig beheerst.

Later in het boek wordt ook het GOTO commando verder besproken.

1f. Verschillende waarden

Wanneer we het over waarden hebben, hebben we het niet alleen over numerieke waarden (waarden die een hoeveelheid aangeven), maar ook over alfanumerieke waarden (tekstwaarden, teksten of **strings**).
In het volgende voorbeeld zijn enkele waarden genoemd.

123	22.5	“Janssen”
0.124	-1000000	“Dit is een tekst”
“Boekje”	“ABCDEFGHIJ”	10203.4

We kunnen dus zeggen dat alles wat we direct opgeven *constant* is en daardoor niet meer veranderd tijdens het uitvoeren van het programma.
We kunnen de waarden wel wijzigen tijdens het programmeren van het programma, maar dat is dan ook de enige mogelijkheid.

Merk op dat we in de computerwereld in plaats van de decimale komma altijd een punt gebruiken.

In de vorige paragraaf zagen we reeds, dat we alfanumerieke waarden door de computer kunnen laten afdrukken.
In het reeds eerder genoemd programma:

```
10 PRINT "Hallo allemaal"
20 PRINT "Hoe gaat het ermee?"
30 GOTO 10
```

worden op regel 10 en 20 de alfanumerieke waarden **Hallo allemaal** en **Hoe gaat het ermee?** afgedrukt op het venster. Alfanumerieke waarden dienen altijd tussen aanhalingstekens “” te worden vermeld in een programma.
Numerieke waarden behoeven *niet* tussen aanhalingstekens te worden vermeld. Indien numerieke waarden toch tussen aanhalingstekens worden geplaatst, noemen we deze waarden niet meer numeriek maar *alfanumeriek*.

Wis de vorige regels zodat je weer een schone editor hebt en neem onderstaande twee regels over:

```
10 PRINT 125
20 PRINT "125"
```

Het programma zal, na het klikken op de blauwe pijl, twee keer het getal 125 op het venster afdrukken. Toch noemen we het in regel 10 een numerieke waarde en de waarde op regel 20 een alfanumerieke waarde, ook wel een *stringwaarde* genoemd.

1g. Rekenen met waarden

We kunnen de computer gebruiken als rekenmachine door hem bijvoorbeeld de waarden bij elkaar op te laten tellen. Tik bijvoorbeeld eens, in een nieuw leeg editorvenster, in:

```
PRINT 123 + 200
```

Na je op de blauwe pijl hebt geklikt, zal het antwoord 323 op het venster verschijnen.

Opdracht: Wijzig de getallen 123 en 200 in andere getallen en voer het uit. Kijk op het venster wat er gebeurt.

Ook moeilijkere sommen worden door Liberty BASIC zonder problemen uitgevoerd.
Zoals je bij bovenstaande PRINT regel ziet, mag je ook de regelnummers weglaten.

```
10 PRINT 12.445 + 123.543
20 PRINT 12 / 6.5
30 PRINT 123 * 456
40 PRINT 1000 – 1
```

Dit programma zal na het runnen in een oogwenk de uitkomsten van de opgegeven sommen op het venster laten zien.
Merk op dat het sterretje '*' als vermenigvuldigingsteken en de schuine streep '/' als deelteken worden gebruikt.

Het rekenen gaat natuurlijk alleen met **numerieke** waarden.
Wanneer we

```
PRINT "JAN" - "PIET"
```

intoetsen, dan zal Liberty BASIC antwoorden met een **Type mismatch** foutmelding; de berekening is niet uit te voeren.

Indien we echter de regel:

```
PRINT "Hallo allemaal" + "Hoe gaat het ermee?"
```

intoetsen, dan zal Liberty BASIC de uitkomst **Hallo allemaalHoe gaat het ermee?** afdrukken. Blijkbaar kan de computer twee alfanumerieke waarden wel *optellen*.

Conclusie:

De computer kan rekenen met numerieke waarden. Alfanumerieke waarden kunnen slechts worden opgeteld: het resultaat is dan de aaneenschakeling van deze waarden.

Aan de ingewikkeldheid van een opgedragen som zijn geen grenzen. De opdracht:

```
PRINT 12 * (2 / 3.5) – 11 * (12345 + 1.22) / 13
```

zal door de computer in een fractie van een seconde worden uitgevoerd. Ook de opdracht:

```
PRINT "ABCDEFGH" + "IJKLMNOPQ" + "RSTUVWXYZ"
```

wordt onmiddellijk uitgevoerd, waarbij de uitkomst wordt gevormd door het alfabet.

Je weet nu dat de waarden constant zijn. Liberty BASIC kent geen namen als constanten, maar later zul je zien dat er wel constanten zijn die nodig zijn in de GUI, zie daar in Deel 2.

Opdracht: **We zagen de regel 'PRINT "Hallo allemaal" + "Hoe gaat het ermee?"'**
Wijzig de PRINT regel zo, dat je als uitvoer krijgt: 'Hallo allemaal. Hoe gaat het ermee?'
Laat het '+' teken wel staan.

Opdracht: **Kijk hieronder naar de PRINT regel.**
Maak een PRINT regel en schrijf in de tekst wat je denkt wat er gebeurd. Maak er een tweede PRINT regel bij met de oplossing.

PRINT "Deze"+ "tekst." + " En de volgende" + tekst"

1h. Variabelen en het commando INPUT

Een variabele is een lade met een opgeborgen waarde erin. Er zijn er twee soorten van: een numerieke lade en een stringlade (alfanumerieke lade). Een string is een tekenreeks waar we van alles in kunnen stoppen, zoals symbolen, letters en ook getallen.

Als we intoetsen:

```
getal = 12.5
```

dan hebben we op dat moment aan de variabele getal de constante 12.5 toegekend. Wanneer we dan later intoetsen:

```
PRINT getal
```

dan zal de waarde, toegekend aan variabele getal, weer afgedrukt worden.

Wanneer we vervolgens intypen:

```
Hoeveelheid = 26
```

dan is op dat moment de waarde 26 aan variabele Hoeveelheid toegekend. Typen we nu in:

```
PRINT getal + Hoeveelheid
```

dan zal de computer netjes de twee waarden toegekend aan de variabelen getal en Hoeveelheid bij elkaar optellen en het resultaat op het venster laten zien.

We kunnen de waarde van een variabele ook wijzigen. Indien we intypen:

```
Hoeveelheid = 622.5
```

en daarna:

```
PRINT Hoeveelheid + getal
```

dan zien we dat de variabele Hoeveelheid een andere waarde heeft verkregen. Niet alleen de variabele Hoeveelheid, maar ook alle andere variabelen mogen steeds weer andere waarden verkrijgen.

Bovenstaande voorbeelden behandelen slechts één van de typen variabelen, namelijk de numerieke variabelen. We kennen echter ook een heel ander soort variabelen, namelijk de alfanumerieke variabelen.

Laten we bijvoorbeeld eens intypen:

```
LET Naam$ = "Geert-Jan"
```

Het woordje LET betekent: laat of laat zijn. LET Naam$ = "Geert-Jan" betekent dus zoveel als laat variabele Naam$ gelijk zijn aan Geert-Jan. Het sleutelwoord LET zouden we ook in onze eerdere voorbeelden kunnen hebben gebruikt, maar het mag in deze constructie zonder meer worden weggelaten. We hadden dus ook gewoon:

```
Naam$ = "Geert-Jan"
```

kunnen intypen.

De variabele Naam$ is een alfanumerieke variabele. Deze heeft dan ook verplicht een dollarteken ‘$’ als laatste teken. Alfanumerieke variabelen kunnen allerlei karakters bevatten. Net als bij alfanumerieke waarden kunnen we alfanumerieke variabelen alleen maar optellen. Probeer bijvoorbeeld eens in te tikken:

```
PRINT Naam$
```

en

```
PRINT Naam$ + " is mijn naam."
```

In het laatste voorbeeld werden een alfanumerieke variabele en een alfanumerieke waarde bij elkaar opgeteld. Het resultaat bestond uit een aaneenschakeling van deze variabele en waarde.

Om het begrip variabele nog wat verder uit te diepen, hebben we de medewerking van nog een Liberty BASIC sleutelwoord nodig, namelijk het sleutelwoord INPUT. Met dit sleutelwoord is het namelijk mogelijk om een variabele een waarde te geven door de waarde in te toetsen, terwijl het programma werkt.

De syntax van INPUT is:

INPUT [“stringwaarde”;] <numerieke variabele>|<stringvariabele>

De stringwaarde kan geen variabele zijn, maar moet als een stringconstante tussen aanhalingstekens gegeven worden. Wel is de stringwaarde optioneel.
Wanneer je de stringwaarde weglaat, moet je ook de puntkomma weglaten. In de uitvoer zal het commando een vraagteken ‘?’ geven om de gebruiker te vertellen dat er om een invoerwaarde wordt gevraagd.

Typ bijvoorbeeld het volgende programma eens in:

```
10 PRINT "Rekenprogramma"
20 INPUT "Hoe heet je "; Naam$
30 PRINT "Goedendag " + Naam$ + " hoe is het? "
40 INPUT "Geef een getal in "; Getal
50 PRINT Getal + Getal; " is het dubbele van "; Getal
60 GOTO 40
```

Zo hebben we met relatief weinig sleutelwoorden al een heel programma geschreven.

In regel 10 wordt op het venster vermeld dat het hier om een rekenprogramma gaat. Niets bijzonders, slechts een eenvoudige alfanumerieke constante wordt op het venster afgedrukt.

In regel 20 komt het nieuwe sleutelwoord **INPUT** aan de beurt. Wanneer we het programma gaan uitvoeren, zien we dat de tekst **Hoe heet je** netjes op het venster wordt afgedrukt en dat de computer daarna wacht. De computer verwacht nu namelijk van je, dat je een alfanumerieke waarde intoetst en dat je, zoals altijd, deze ingeving met een enter toets afsluit. Jouw ingeving wordt in variabele **Naam$** bewaard.

In regel 30 presenteert de computer de optelling van een alfanumerieke waarde, een alfanumerieke variabele en weer een alfanumerieke waarde. Hierdoor begroet de computer ons hoffelijk en noemt hij ons zelfs bij de naam; een bewijs dat in Naam$ inderdaad de eerder gepleegde ingeving is geplaatst.

In regel 40 vraagt de computer weer om een ingeving. Ditmaal dient de ingeving een numerieke waarde te zijn; dat is te zien aan de naam van de variabele in het INPUT-commando. Deze naam eindigt niet op een dollarteken en dit duidt op een numerieke variabele.
Indien we tijdens deze INPUT proberen een niet geldige numerieke waarde in te geven, dan geeft Liberty BASIC in plaats daarvan een nul als resultaat.

In regel 50 wordt de optelling van de variabele **Getal** bij zichzelf afgedrukt. Op het venster verschijnt het dubbele van de ingegeven waarde. Na deze waarde wordt de string **is het dubbele van** afgedrukt, gevolgd door de waarde van de inhoud van numerieke variabele **Getal**. Uit deze programmaregel leren we meteen dat we in een PRINT-commando verschillende zaken achter elkaar aan kunnen afdrukken door deze met een puntkomma van elkaar te scheiden.

Tenslotte gaven we de computer op regel 60 het commando om weer naar programmaregel 40 terug te gaan, de computer vraagt weer om een ingeving. Wanneer we het programma niet met de Control toets en de Pause/Break toets onderbreken, zal hij voortdurend met het programma bezig blijven.

We zien dat het gebruik van variabelen een nieuwe dimensie geeft aan de computer. Door het gebruik van variabelen kunnen we eenzelfde programma steeds van andere waarden voorzien waardoor steeds nieuwe situaties worden berekend.

1i. Soorten van variabelen

Variabelen kunnen we gebruiken om waarden in op te bergen. Maar laten we eens kijken wat voor variabelennamen we mogen gebruiken. Wat is toegestaan?

Liberty BASIC bepaalt het volgende:

- Variabelennamen mogen hoofdletters hebben, eventueel gemengd met kleine letters.
- Variabelennamen mogen gemengd zijn met cijfers, maar ze mogen niet met een cijfer beginnen.
- Variabelennamen mogen niet gemengd zijn met symbolen, uitgezonderd de punt. De punt heeft een speciale functie dat later in het boek besproken wordt.
- Variabelen zonder een dollarteken zijn integers, maar ook getallen met aantal decimalen achter de komma (hier de punt in plaats van de komma) en ook precisie waarden (waarden met de exponentiële letter **e**) zijn toegestaan.
- Variabelen zonder een dollarteken accepteren alle numerieke waarden. Liberty BASIC bepaalt zelf de maximale grens als een getal wordt toegekend. In andere BASIC dialecten gelden daar speciale tekens voor om datatypen aan te duiden.

1j. Arrayvariabelen

Het kan voorkomen dat we meerdere waarden onder één en dezelfde variabele naam willen bewaren. In dat geval zullen we, behalve de variabele naam, ook dienen aan te geven welke waarde van alle waarden we bedoelen, die we opgeslagen hebben. Een variabele waaronder we meer dan één waarde willen bewaren, kunnen we vergelijken met een tabel. We noemen zo'n variabele een array-variabele of kortweg een array (rij).

Zo'n array dienen we een bepaalde grootte (een bepaald aantal mogelijkheden tot opslag van waarden) toe te kennen. Dit doen we met behulp van het sleutelwoord DIM.

Syntax:

DIM <arrayvariabele>(<index1>[,<index2])

De arrayvariabele kan een numerieke variabele of een stringvariabele zijn. De indexen zijn het aantal elementen die je gebruiken wilt.
In Liberty BASIC kunnen we maximaal 2 indexen gebruiken. We kunnen de indexen ook dimensies noemen.

Voorbeeld: DIM arrayvariabele(20)
Dit betekent: Dimensioneer een numerieke arrayvariabele met een index (dimensie) van 21 elementen.

We gaan een programma schrijven:

```
10 DIM Tabel(3)
```

Wanneer we dit programma in werking stellen, dan wordt een variabele, genaamd **Tabel**, toegewezen. Deze variabele biedt mogelijkheid tot opslag van vier waarden, in dit geval numerieke waarden met eventueel floating-point waarden (waarden met decimalen achter de komma). Deze vier waarden worden opgeslagen onder:

Tabel(0) …eerste waarde
Tabel(1) …tweede waarde
Tabel(2) …derde waarde
Tabel(3) …vierde waarde

Verder in het programma kunnen we deze tabel gaan vullen:

```
20 LET Tabel(0) = 12
30 LET Tabel(1) = 96
40 LET Tabel(3) = -45.5
50 LET Tabel(2) = Tabel(3) * Tabel(1) / Tabel(0)
60 PRINT Tabel(0), Tabel(1), Tabel(2), Tabel(3)
```

Wanneer we dit programma uitvoeren, dan merken we na enige narekening dat de variabele Tabel inderdaad vier afzonderlijke waarden bevat die we kunnen veranderen en waarmee we kunnen rekenen.

In programmaregel 60 zien we overigens, dat bij een PRINT commando de afzonderlijke gegevens met een komma van elkaar mogen worden gescheiden. We zagen zoiets al eerder met ook een puntkomma. Het gebruik van een komma heeft tot gevolg dat de gegevens ietwat uit elkaar, zo mogelijk op dezelfde regel, worden afgedrukt.

We kunnen het programma uitbreiden met het volgende gedeelte:

```
60 INPUT "Welke tabelwaarde: "; Nummer
70 PRINT "Waarde: "; Tabel(Nummer)
80 GOTO 60
```

De oude regel 60 vervang je en maak je daarvoor een nieuwe regel 60.

Let op!
Indien je **niet** de oude regel 60 vervangt, maar gewoon een nieuwe regel 60 eronder zet, zal Liberty BASIC ze alle twee laten staan. De reden daarvan komt doordat Liberty BASIC het niet als echte regelnummers ziet en zal bij het uitvoeren een foutmelding geven. Later in het boek daarover meer.

Wanneer we dit programma in werking stellen, vraagt de computer ons op regel 60 welk tabelelement we willen zien. We moeten dan een waarde 0,1,2 of 3 ingeven om geen nul-waarde te krijgen. Wanneer we een waarde ingeven, wordt deze in **Nummer** opgeslagen.

Op programmaregel 70 wordt dan vervolgens het tabelelement met het zojuist ingegeven nummer op het venster getoond. We zien dat we het bedoelde tabelelement niet alleen met een getal maar ook met een andere variabele kunnen aanduiden.

Een arrayvariabele kan in meer dan één richting bepaald zijn. Het volgende programmavoorbeeld geeft een voorbeeld van het werken met een tweedimensionale array. Merk op dat er tevens meerdere commando's op één programmaregel zijn geplaatst. Dit is toegestaan indien deze commando's met een dubbele punt van elkaar zijn gescheiden.

De eerste programmaregel bevat een REM-regel. REM is een afkorting van REMARK (=opmerking). Met deze regel wordt slechts een stukje commentaar in het programma opgenomen dat door de computer wordt genegeerd.

Begin in Liberty BASIC met een nieuw bestand en typ onderstaand programma in:

```
10 REM Voorbeeld twee dimensionale array
20 DIM TB(2, 2)
30 TB(0, 0) = 12 : TB(0, 1) = 345 : TB(0, 2) = -22
40 TB(1, 0) = 99 : TB(1, 1) = 871 : TB(1, 2) = 5
50 TB(2, 0) = TB(0, 0) + TB(1, 0) : TB(2, 1) = TB(0, 1) + TB(1, 1)
60 TB(2, 2) = TB(0, 2) + TB(1, 2)
70 PRINT TB(0, 0), TB(0, 1), TB(0, 2)
80 PRINT TB(1, 0), TB(1, 1), TB(1, 2)
90 PRINT TB(2, 0), TB(2, 1), TB(2, 2)
100 PRINT "Einde"
```

We zien in dit programma dat het commando LET is weggelaten. Het kan in Liberty BASIC ook zonder.
Als we het programma uitvoeren, dan zien we hoe de tabel op het venster is opgebouwd.

```
 12         345        -22
 99         871        5
 111        1216       -17
 Einde
```

De volgorde van de arrayelementen, dat de opgegeven nummers zijn, die voor deze uitvoer zorgen, zie je op regel 70, 80 en 90. Het woord ‘Einde’ wordt onder de uitvoer van de array geprint.

In Liberty BASIC 4 is het helaas niet mogelijk om meer dimensionale arrays te gebruiken. In de toekomst zal Liberty BASIC 5 een meerdimensionale array tot vier dimensies ondersteunen. Nu moeten we het doen met ééndimensionale en tweedimensionale arrays.

Tot slot van deze paragraaf zul je zien dat arrays niet alleen met numerieke maar ook met alfanumerieke variabelen zijn toegestaan. Zo kan bijvoorbeeld in een tabel die met:

```
DIM Adres$(100, 3)
```

is toegewezen, de naam, straat, woonplaats en telefoonnummer van 101 kennissen worden opgeslagen.

2. Uitdrukkingen

2a. Wat is een uitdrukking

We kunnen in Liberty BASIC de computer de volgende som laten oplossen:

```
PRINT 2 * (Waarde + 5.5) / (2 + Aantal)
```

De variabelen Waarde en Aantal dienen dan wel te zijn gevuld met bepaalde waarden.

De opgave (de som) die na het PRINT-commando staat, noemt men in de computerwereld een **uitdrukking** of ook wel genoemd een **expressie**. Vormen van uitdrukkingen zijn bijvoorbeeld ook:

2 + 3 25 / (3 – 4 * Q)	A / B “Hallo” + Naam$	“Jan” + “Karel” 12.55 + 3

Deze uitdrukkingen, numeriek of alfanumeriek, geven aan:

- welke bewerkingen dienen te geschieden (optellen, aftrekken, vermenigvuldigen, etc.);
- op welke variabelen deze bewerkingen dienen te worden toegepast;
- op welke waarden deze bewerkingen dienen te worden toegepast;
- op welke volgorde de bewerkingen dienen te worden toegepast.

In een uitdrukking vinden we de volgende elementen:

- variabelen: Deze hebben een waarde die al eerder bepaald zijn.
- constanten: Deze hebben een vastgestelde waarde.
- bewerkingscodes: Deze geven aan welke bewerkingen dienen te geschieden.
- voorrangstekens: Deze worden altijd gevormd door haakjes. Met deze voorrangstekens kan een afwijkende voorrang in bewerking worden vastgelegd.

We kunnen een uitdrukking als volgt definiëren:

> **Een uitdrukking of expressie is een in principe uitvoerbare samenstelling van constanten, variabelen, bewerkingen en voorrangstekens.**

Op de eerste twee elementen, de constanten en variabelen, zijn we in de voorgaande hoofdstukken reeds diep ingegaan. In de volgende paragrafen gaan we op bewerkingen, voorrang en voorrangstekens in.

2b. Algebraïsche bewerkingen

Liberty BASIC kent de volgende algebraïsche bewerkingen:

+	optellen: Met het plusteken geven we aan dat de uitdrukkingen links en rechts van dit teken bij elkaar dienen te worden opgeteld. Dit is de enige bewerking die ook op alfanumerieke waarden mag worden toegepast.
-	aftrekken: Met het minteken geven we aan dat de uitdrukkingen links en rechts van dit teken van elkaar moeten worden afgetrokken. Er wordt dus een verschil van de uitdrukkingen bepaalt. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.

*	vermenigvuldigen: Met het sterretje, asterisk of vermenigvuldigingsteken, geven we aan dat de uitdrukkingen rechts en links van dit teken met elkaar dienen te worden vermenigvuldigd. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.
/	delen: Met de deelstreep geven we aan dat de uitdrukkingen links en rechts van dit teken door elkaar moeten worden gedeeld. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.
^	machtsverheffen: Met het 'dakje' geeft men aan dat de uitdrukking links van dit teken in een macht dient te worden verheven. De uitdrukking rechts van dit teken geeft aan om welke macht het gaat. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.
INT()	integer: In deze functie kan elke bewerking worden toegepast. Een deling kan in deze functie worden toegepast om er een integer-deling van te maken. Andere BASIC dialecten gebruiken de \ deelstreep of kennen het sleutelwoord DIV. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.
MOD	restbepaling: Met dit sleutelwoord geeft men aan dat de uitdrukking links en rechts van dit sleutelwoord eerst door elkaar worden gedeeld. De restwaarde die bij de deling overblijft is de uitkomst van deze bewerking. Deze bewerking mag alleen op numerieke uitdrukkingen worden toegepast.

Hieronder volgt een voorbeeldprogramma:

```
10 INPUT "Eerste waarde "; A
20 INPUT "Tweede waarde "; B
30 PRINT "Optelling "; A + B
40 PRINT "Aftrekking "; A - B
50 PRINT "Vermenigvuldiging "; A * B
60 PRINT "Deling "; A / B
70 PRINT "Integere deling "; INT(A / B)
80 PRINT "Rest bij deling "; A MOD B
90 PRINT "Macht "; A ^ B
100 GOTO 10
```

2c. Functionele bewerkingen

Liberty BASIC kent een groot aantal functionele bewerkingen. We hebben in de vorige paragraaf er één gehad: de functie INT().

Een functionele bewerking is een bewerking die kan worden toegepast op een uitdrukking of op een stelsel van uitdrukkingen. Een functionele bewerking heeft altijd de vorm van: **functionele bewerking (uitdrukking...)**
Om het één en ander wat nader toe te lichten, volgen hieronder enkele voorbeelden van functionele bewerkingen.

Typ bijvoorbeeld eens in:

```
PRINT INT(12.9)
```

De computer zal antwoorden met het getal 12. De functionele bewerking INT heeft tot gevolg dat het grootste gehele getal kleiner dan de tussen de haakjes staande uitdrukking wordt berekend.

```
PRINT LEFT$("ABCDEFGHIJKLMNOP", 5)
```

De computer antwoordt hier met de uitkomst ABCDE. De functie LEFT$ heeft tot gevolg dat het linker gedeelte van de opgegeven alfanumerieke uitdrukking wordt bepaald. Het aantal tekens is gegeven in de tweede tussen haakjes gegeven uitdrukking, die numeriek is.

```
PRINT RIGHT$("ABCDEFGHIJKLMNOP", 5)
```

De computer antwoordt hier met de uitkomst LMNOP. De functie RIGHT$ heeft tot gevolg dat het rechter gedeelte van de opgegeven alfanumerieke uitdrukking wordt bepaald.

```
PRINT MID$("ALLEMAAL", 3, 4)
```

De computer antwoordt hier met de uitkomst LEMA. Met de functie MID$ kan dus een middendeel van een string worden teruggegeven. De derde parameter is optioneel. Indien je die weglaat, zal de rest van de string vanaf plaats 3 worden teruggegeven.

```
PRINT ABS(-12.8)
```

De computer antwoordt hier met de uitkomst 12.8. De functie ABS, dat ABSolute betekent, zorgt ervoor dat het resultaat van een uitdrukking altijd positief blijft. Ofwel, de originele waarde zonder teken teruggeeft.

Een voorbeeldprogramma:

```
10 INPUT "Geef een waarde in: "; Waarde
20 LET A = INT(Waarde)
30 LET B = Waarde
40 LET C = ABS(Waarde)
50 PRINT "Integere waarde: "; A
60 PRINT "Absolute waarde: "; C
70 IF B < 0 THEN PRINT "Negatief"
80 PRINT LEFT$("ABCDEFGHIJKLMNOPQRSTUVWXYZ", Waarde)
90 GOTO 10
```

Dit voorbeeldprogramma vraagt eerst een numerieke ingeving in Waarde.
Vervolgens wordt in programmaregel 20 de functie INT toegepast en in regel 30 alleen de waarde toegepast; de resultaten worden onder de variabele namen A en B bewaard.
In regel 40 wordt de absolute waarde van Waarde bepaald.
In regel 50 wordt dan de integere waarde op het venster getoond.
In regel 60 wordt de absolute waarde op het venster getoond.
In regel 70 komen we een nieuw sleutelwoord tegen: IF. Deze regel kan gelezen worden als: als variabele B kleiner is dan nul (dus als een negatieve waarde werd ingegeven), druk dan de tekst ‘Negatief’ af. Met het IF-sleutelwoord kunnen we dus een voorwaarde stellen. Als aan deze voorwaarde wordt voldaan, wordt het daarna opgenomen commando uitgevoerd. Op dit sleutelwoord komen we verderop nog uitgebreid terug.
In regel 80 worden de eerste letters van het alfabet afgedrukt. Het aantal letters is bepaald door de waarde van variabele Waarde die wij ingaven.

Merk op dat Liberty BASIC op regel 80 geen fout geeft bij een negatieve waarde, zoals je eigenlijk zou verwachten. Oudere BASIC dialecten geven wel een foutmelding.

Meer over de functies van Liberty BASIC kun je vinden in hoofdstuk 3.

2d. Relationele bewerkingen

Liberty BASIC kent een aantal relationele bewerkingen. Een relationele bewerking is een bewerking die uitdrukking geeft aan een relatie.

Typ bijvoorbeeld eens in:

```
PRINT 2 = 2
```

De computer zal het antwoord 1 geven. Dit antwoord betekent dat de bewering 2 = 2 inderdaad waar is. De uitdrukking links van het gelijkteken en de uitdrukking rechts van het gelijkteken hebben de relatie, dat ze in waarde gelijk zijn aan elkaar. Typ bijvoorbeeld eens in:

```
PRINT (5 + 2) = (8 - 1)
```

Ook hier zal het oordeel van de computer **waar** (1) luiden; 5 + 2 is inderdaad gelijk aan 8 - 1. Typ nu eens in:

```
PRINT 5 + 3 = 7
```

Het oordeel van de computer is nu **niet waar** (0). De uitdrukkingen 5 + 3 en 7 hebben niet de relatie gelijkheid met elkaar.

Het gelijkteken in de hiervoor gegeven voorbeelden is een relationele bewerking. Een relationele bewerking dient altijd plaats te vinden tussen twee numerieke uitdrukkingen of tussen twee alfanumerieke uitdrukkingen. Het resultaat is altijd 1 (**waar)** of 0 (**niet waar)**. Nog een voorbeeld met alfanumerieke uitdrukkingen:

```
PRINT "Jan" = "J" + "an"
PRINT "Jan" = "Janneman"
```

De eerste opdracht zal resulteren in het antwoord 1 (**waar**), de tweede in het antwoord 0 (**niet waar**).
De bewerking “is gelijk aan” (=) is slechts één van de relationele bewerkingen die Liberty BASIC kent. Hieronder volgt een tabel met alle relationele bewerkingen:

=	is gelijk aan
<	is kleiner dan
>	is groter dan
<>	is kleiner of groter dan (ofwel: is ongelijk aan)
<=	is kleiner dan of gelijk aan
>=	is groter dan of gelijk aan

Een voorbeeldprogramma:

```
10 INPUT "Waarde: "; A
20 INPUT "Nog een waarde: "; B
30 IF A < B THEN PRINT "De eerste waarde is kleiner dan de tweede."
40 IF A = B THEN PRINT "De twee waarden zijn gelijk aan. "
50 IF A > B THEN PRINT "De eerste waarde is groter dan de tweede. "
60 IF A <= B THEN PRINT "De eerste waarde is kleiner dan of gelijk aan de tweede."
70 GOTO 10
```

Na enig proberen zal het volgende blijken:
Indien een uitdrukking de uitkomst 1 heeft, dus waar is, wordt het commando (of worden de commando's) achter THEN in een IF...THEN constructie uitgevoerd. Is de uitdrukking echter niet waar (de uitkomst is dan 0) dan gebeurt dit niet.
Probeer nu eens het volgende programma:

```
10 INPUT "Waarde: "; A
20 IF A THEN PRINT "Waar"
30 GOTO 10
```

en laat dit programma eens uitvoeren. Geef diverse waarden in, negatief, positief en ook eens een 0. De computer zal steeds antwoorden met de tekst ”Waar”, tenzij een 0 werd ingegeven.

Conclusie:

Een relationele bewerking geeft bij uitvoering de uitkomst waar of niet waar. Deze oordelen worden gesymboliseerd in de waarden 1 of 0. In de IF...THEN-constructie kan het al dan niet waar zijn van een uitdrukking worden getest. Indien een uitdrukking als waar wordt bevonden, wordt het programmaregelgedeelte achter het sleutelwoord THEN uitgevoerd; tussen IF en THEN mag elke uitdrukking worden geplaatst. Deze uitdrukking behoeft niet verplicht een relationele bewerking te bevatten. De IF...THEN-constructie beschouwt een uitdrukking als waar indien deze ongelijk is aan nul (zie het laatste voorbeeld, de uitkomst hoeft dus niet verplicht gelijk te zijn aan 1) en als niet waar indien deze wel gelijk is aan nul.

In Liberty BASIC is er een probleem als we getallen met decimalen willen controleren.

Wanneer we zeggen:

```
PRINT 0.1 > 0
```

zien we op het venster een 1 (waar) verschijnen.

Wanneer we zeggen:

```
IF 0.1 THEN PRINT "Waar"
```

dan wordt echter niet de tekst Waar geprint, ook al is toch 0.1 groter dan 0.

Het lijkt erop dat Liberty BASIC het naar beneden afrondt en het als een 0 ziet, maar dan had de relationele bewerking achter PRINT ook niet waar moeten zijn.

Op deze bijzonderheid moeten we niet teveel op kijken. Het is kennelijk altijd beter om getallen met decimalen achter de komma relationeel te controleren en niet booleaans (0 of 1). In het tweede deel van het boek kom ik terug op booleaanse waarden.

Merk op dat het gelijkteken in BASIC een dubbele functie uitoefent.

Eerste functie: LET A = 12.34	in een dergelijke constructie vormt het gelijkteken geen relationele bewerking maar geeft het slechts aan dat een variabele dient te worden gelijkgesteld aan een uitdrukking.
Tweede functie: PRINT A = B	in een dergelijke constructie vormt het gelijkteken een relationele bewerking.

2e. Logische bewerkingen

Liberty BASIC biedt verscheidene logische bewerkingen.
Probeer, zeker als beginner, te begrijpen wat logische bewerkingen zijn en hoe ze werken.
Een logische bewerking vergelijkt twee uitdrukkingen en doet vervolgens een uitspraak in de zin van WAAR (ongelijk aan 0) of NIET WAAR (gelijk aan 0). Aan de hand van de meest gemakkelijke logische bewerking licht ik dit wat nader toe. Typ bijvoorbeeld elke regel in en voer het één voor één uit:

```
IF 2 + 3 = 5 AND 4 + 6 = 10 THEN PRINT "Hoera"
IF 2 + 4 = 6 AND 6 + 6 = 11 THEN PRINT "Een vreemde zaak"
IF 2 + 4 = 6 OR 70 - 3 = 60 THEN PRINT "Dit is gek!"
IF "Jan" < "Karel" AND 4 * 4 = 16 THEN PRINT "Hoera"
```

De eerste regel resulteert evenals de vierde regel in het afdrukken van de tekst ‘Hoera’ op het venster. De tweede regel blijft zonder resultaat. De derde regel drukt wel de tekst ‘Dit is gek!’ af, ondanks dat één van de twee bewerkingen NIET WAAR is.

Verklaring: De logische bewerking AND onderzoekt de uitdrukking links en rechts van deze bewerking. Alleen indien beide bewerkingen WAAR zijn (1 als uitkomst hebben) dan krijgt de totale uitdrukking tussen IF en THEN de beoordeling WAAR (de waarde 1) en wordt het gedeelte achter THEN uitgevoerd.

De logische bewerking OR in de derde regel onderzoekt niet of beide bewerkingen WAAR zijn, maar of de een of het ander WAAR is, waardoor we toch de tekst te zien krijgen.

De vierde regel geeft een extra moeilijkheid; er wordt een bewering gedaan met twee teksten waarbij wordt beweerd dat de eerste tekst kleiner is dan de tweede tekst. Indien de uitdrukkingen, waarop een relationele bewerking wordt toegepast, alfanumeriek zijn, dan worden deze uitdrukkingen alfabetisch uit gewaardeerd. Een tekst die bij een alfabetische rangschikking voor een tweede tekst terecht zou komen, is ook kleiner dan deze tweede tekst.
Over het algemeen kan men stellen dat karakters volgens het karaktertabel van de computer worden vergeleken waarbij een karakter met de laagst gecodeerde waarde ook het kleinst is.

De eerste voorbeeldregel zou in normaal Nederlands als volgt kunnen worden vertaald:

Als 2 + 3 gelijk is aan 5 **en** 4 + 6 is gelijk aan 10, druk dan de tekst ‘Hoera’ af.
De logische bewerking AND kan men dus vertalen met het woordje EN.

Als 2 + 4 gelijk is aan 6 **of** 70 – 3 is gelijk aan 60, druk dan de tekst ‘Dit is gek!’ af.
De logische bewerking OR kan men dus vertalen met het woordje OF.

Hieronder is een tabel met alle logische bewerkingen die in Liberty BASIC bestaan:

Logische bewerking	Betekenis	Uitleg
AND	en	Deze logische bewerking wordt tussen twee uitdrukkingen geplaatst. Hij geeft de uitkomst WAAR (ongelijk aan nul) alleen indien de beide uitdrukkingen WAAR (ongelijk aan nul) zijn.
OR	of	Deze logische bewerking wordt tussen twee uitdrukkingen geplaatst. Hij geeft de uitkomst WAAR indien minstens één van de twee uitdrukkingen WAAR is.

XOR	exclusief of	Deze logische bewerking wordt tussen twee uitdrukkingen geplaatst. Hij geeft slechts de uitkomst WAAR indien minstens en hoogstens één van de uitdrukkingen WAAR is.
NOT	niet	Deze logische bewerking wordt vóór een tussen haakjes gestelde uitdrukking geplaatst en draait de waarde van deze uitdrukking om. Indien deze uitdrukking WAAR is, veroorzaakt deze bewerking dus het antwoord NIET WAAR en omgedraaid. In Liberty BASIC zijn haakjes bij deze logische bewerking verplicht. In andere programmeertalen juist niet. Deze logische bewerking geeft altijd een -1 als de uitdrukking WAAR is, in tegenstelling wanneer we een vergelijking achter een PRINT commando afdrukken zal er juist een 1 als WAAR op het venster worden getoond.

Naast de logische bewerkingen bestaan er ook bitgewijze bewerkingen. Deze bewerkingen komen later in deel 2 van het boek terug.

2f. Voorrangsregels

Wanneer in een uitdrukking veel bewerkingen voorkomen, dan zal al snel de twijfel bestaan welke bewerking het eerste dient te geschieden. Bekijken we bijvoorbeeld de uitdrukking in de volgende programmaregel:

```
PRINT 2 * 3 + 4
```

De uitkomst zal luiden: 10. Dit wijst erop dat eerst de vermenigvuldiging en pas later de optelling is uitgevoerd. De vermenigvuldiging had blijkbaar voorrang op de optelling. We kunnen deze voorrang wijzigen door het een en ander tussen haakjes te plaatsen:

```
PRINT 2 * (3 + 4)
```

In dit geval wordt eerst de uitdrukking tussen haakjes uitgewerkt en pas daarna de vermenigvuldiging gedaan. De uitkomst luidt dan ook: 14.

Indien de moeilijkheid zich zou beperken tot vermenigvuldigen en optellen, dan zou er niet zoveel aan de hand zijn. Echter, in Liberty BASIC hebben we een keur van algebraïsche, relationele, functionele en logische bewerkingen. In dat woud van bewerkingen is het nuttig om te weten welke bewerkingen voorrang hebben op welke andere bewerkingen in Liberty BASIC. De voorrangsregels zijn niet zo erg moeilijk; ze komen behoorlijk overeen met de voorrangsregels die algemeen in rekenwerk worden toegepast.

Allereerst noemen we de haakjes. Een gouden regel is dat haakjes de allerhoogste voorrang hebben. De uitdrukking die het diepst tussen haakjes staat, wordt altijd het eerst door de computer uitgewerkt. Dit geldt ook voor de haakjes die verplicht zijn bij een functionele bewerking, zoals bij INT() en NOT().

Buiten deze haakjes liggen de voorrangsregels als volgt:

Voorrangs-volgorde	Bewerkingen	Opmerkingen
1	functionele bewerkingen	Door het verplichte gebruik van haakjes kunnen onderling geen voorrangsconflicten ontstaan.
2	algebraïsche bewerkingen	eerst ^ dan *, / en MOD dan + en -
3	relationele bewerkingen	=, <, >, <=, >=, <> Bij deze bewerkingen geldt geen volgorde.
4	logische bewerkingen	Eerst NOT() dan AND, OR, XOR waarbij deze bewerkingen geen volgorde geldt.

Indien er tussen gelijkwaardige bewerkingen conflictsituaties ontstaan, dan worden deze bewerkingen van links naar rechts in volgorde uitgevoerd.

Bij het uitwerken van uitdrukkingen dient de computer vaak vrij ingewikkelde berekeningen te maken. Alhoewel dit schijnbaar moeiteloos gebeurt, komt voor een berekening vaak veel kijken. Typ eens in:

```
PRINT 2 + 40 / INT(2 * 5 + 6.1)
```

Onmiddellijk produceert de computer het juiste antwoord. Laten we eens nagaan, welke bewerkingen de computer moest doen om tot dit antwoord te komen.
Uit de voorrangsregels volgt, dat de berekening als volgt dient te worden uitgevoerd:

Uitdrukking	`2 + 40 / INT(2 * 5 + 6.1)`
Stap 1	`2 + 40 / INT(10 + 6.1)`
Stap 2	`2 + 40 / INT(16.1)`
Stap 3	`2 + 40 / 16`
Stap 4	`2 + 2.5`
Stap 5	`4.5`

Tijdens deze uitvoerige berekeningen moeten nogal eens tussenuitkomsten door de computer worden bewaard. Deze tussenuitkomsten worden in zogenaamde systeemvariabelen bewaard en zijn in Liberty BASIC niet te benaderen.

2g. Tekst

Waar we het meest mee werken in Liberty BASIC is tekst. Tekst kan van alles zijn: stringconstanten, stringvariabelen. Maar dat niet alleen, ook de code zelf is tekst.

Een string is een tekenreeks, een tekst waar van alles in kan zitten. Alle letters, cijfers en symbolen kunnen erin zitten. Zelfs symbolen die je niet kunt zien wordt als tekst beschouwt.

Wanneer we een tekst intypen zoals “IK VIND JE AARDIG”, dan zijn we duidelijk niet met het intypen van getallen bezig, maar met het intypen van tekst. Zo’n stuk tekst wordt in BASIC begrippen een *string* genoemd (string betekent: een *lint* of *koord*). Vandaar de term tekenreeks.

Een string bestaat uit een rij tekens, die door de computer niet als getallen, maar op een andere wijze wordt behandeld. Deze tekens worden wel *alfanumerieke tekens* genoemd. Zij bevatten namelijk letters uit het alfabet (alfanumerieke informatie) en ook cijfers (numerieke informatie). Het is zelfs zo dat hierin ook tekens voorkomen die niet kunnen worden geprint, maar alleen op andere wijze worden verwerkt. Al deze tekens kunnen we via het toetsenbord invoeren, zie afbeelding.

afbeelding: een oud toetsenbord

In feite is tekst in BASIC natuurlijk niet iets nieuws. We hebben al eerder gezien hoe we het intypen en hoe het werkt. Er is al veel over uitgelegd hoe tekst in BASIC in elkaar zit, maar ik laat in deze paragrafen nog meer zien wat je ermee kunt doen en hoe het werkt.

In dit hoofdstuk ga ik dieper op in hoe uitdrukkingen als *expressies* werken.

Zo hebben we bijvoorbeeld een uitdrukking gezien zoals:

```
PRINT "Typ een waarde in"
```

Hierbij lieten we de tekst **Typ een waarde in** op het mainwin venster zetten, na de klik op de blauwe pijl.
Deze tekst is volgens de hierboven gegeven definitie een string. We hebben zelf al aan onze computer meegedeeld dat we met een *string* te maken hebben. Deze is namelijk door ons tussen aanhalingstekens " geplaatst. Op grond hiervan wist de computer dat hij met een string van gewone tekst te maken had.

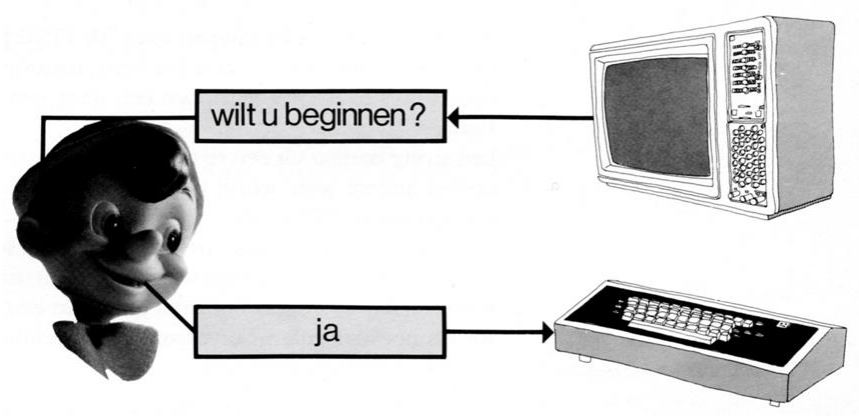
Een ander voorbeeld hadden we bij het INPUT statement:

```
INPUT "Hoe oud ben je? "; jr
```

Ook hier gebruikten we al een string, namelijk “Hoe oud ben je? “. In dit geval is die string eveneens tussen aanhalingstekens geplaatst (in het Engels wordt het aanhalingsteken *quote* genoemd). In de computertaal BASIC wordt niet of weinig over dubbele aanhalingstekens gesproken. Er wordt meer gezegd dat een string tussen aanhalingstekens staat dan steeds gezegd wordt dat het tussen dubbele aanhalingstekens staat. Daarom houd ik me in dit boek alleen op *aanhalingstekens*.

We zien uit deze voorbeelden dat de string wordt gebruikt in een soort gesprek tussen de computer en de bediener van de computer. In beide voorbeelden zet de computer een zin op het mainwin venster en de gebruiker moet als antwoord een getal intypen.

Het zou zeer wenselijk zijn wanneer de gebruiker ook gewoon een tekst zou mogen intypen, zoals bijvoorbeeld **ja** of **nee**. Dan kan een veel eenvoudiger conversatie ontstaan, zie afbeelding.



Wanneer de computer op deze tekst moet kunnen reageren, dan moet hij allereerst kunnen herkennen dat we met tekst werken. Ten tweede moet er dan iets met die tekst worden gedaan.

In hoofdstuk 1 hebben we gezien dat er variabelen zijn waaraan we een bepaalde getalwaarde toekennen. Zo zijn er ook variabelen waaraan we strings kunnen toekennen. We noemen deze variabelen *alfanumerieke variabelen*. De naam *stringvariabelen* wordt ook wel in plaats van alfanumerieke variabelen genoemd.

Om deze alfanumerieke variabelen van de numerieke variabelen te onderscheiden moet in BASIC zo’n variabele altijd met een *dollarteken* $ worden geëindigd.
Legale alfanumerieke variabele namen zijn bijvoorbeeld:

```
A$
N$
Naam$
```

Dit betekent dat bijvoorbeeld een variabele A getalwaarden kan bevatten, maar geen stringwaarden, terwijl de variabelen A$, N$ en Naam$ strings kunnen bevatten maar geen getalwaarden.

Voorbeeld:

```
A$ = "Ik vindt je zo aardig. "
PRINT A$
```

Als we het uitvoeren, verschijnt er op de mainwin:

```
Ik vindt je zo aardig.
```

We hebben nu gezien dat er alfanumerieke variabelen bestaan. Deze moeten met een $ worden afgesloten. De string moet tussen aanhalingstekens staan.

Overigens: je ziet in bovenstaand voorbeeld dat de taalfout (Ik vindt) keurig bewaard blijft. De computer corrigeert dus niet.

Opdracht: Kijk naar onderstaande lijst en probeer deze lijst als losse tekst op het mainwin venster te printen.
Gebruik bij de derde tekst twee aparte stringconstanten en zorg ervoor dat je een tab gebruikt in plaats van spaties. Gebruik het juiste teken voor de tab en het juiste teken om aaneen te schakelen.

Print de woorden ‘fijn’ en ‘weer’ in de vierde tekst als aparte woorden. Gebruik het juiste teken om aaneen te schakelen.

```
Hallo
Kijk eens aan!
We kunnen zeggen:     een eindje verder.
De woorden 'fijn' en 'weer' wordt als tekst: 'fijn weer'.
```

2h. Stringoperaties

We kunnen nu de meeste operaties, die in de vorige hoofdstukken ter sprake zijn gekomen, ook op een alfanumerieke variabele laten werken. Onderstaande afbeelding laat zien hoe dat in zijn werk ging in de oude BASIC dialecten. Naast de afbeelding staat dezelfde code in Liberty BASIC syntax.


```
A$ = "Mooi weer"
PRINT A$
INPUT "Of niet soms? "; B$
IF UPPER$(B$) <> "JA" THEN
      FOR I = 1 TO 4
             PRINT A$; " toch wel!"
      NEXT I
END IF
PRINT "Bedankt!"
```

Als we de uitvoer bekijken, zien we het volgende op het mainwin venster.

```
Mooi weer
Of niet soms? Nee
Mooi weer toch wel!
Mooi weer toch wel!
Mooi weer toch wel!
Mooi weer toch wel!
Bedankt!
```

Starten we het nog eens, dan

```
Mooi weer
Of niet soms? Ja
Bedankt!
```

We zien het verschil met de oude BASIC code en een keuzestructuur (zie hoofdstuk Structuren) in Liberty BASIC, dat veel beter gestructureerd lijkt. Ook al zitten we zo lekker te likkenbaden op het GOTO commando, die het juist zo gemakkelijk maakt om een gelijk aan voorwaarde te kunnen gebruiken. Als het niet gelijk is aan JA, wordt er altijd BEDANKT gemeld, maar we kunnen het niet hetzelfde doen met de Liberty BASIC code rechts van de afbeelding, want als we ‘gelijk aan’ zeggen, zal de FOR lus uitgevoerd worden met een JA en dat willen we niet.

Kijk ook goed dat de tekst ‘Bedankt!’ buiten de IF … END IF staat. Je kunt het ook niet oplossen door een ‘gelijk aan’ te gebruiken, het PRINT commando met de string “Bedankt!” onder THEN te verplaatsen en de FOR lus tussen de ELSE en een END IF te plaatsen. Je zou dan onderstaande keuzestructuur krijgen:

```
IF UPPER$(B$) = "JA" THEN
       PRINT "Bedankt! "
ELSE
       FOR I = 1 TO 4
              PRINT A$; " toch wel! "
       NEXT I
END IF
```

Nu zouden we alleen het woordje ‘Bedankt!’ te zien krijgen als we ‘Ja’ intypen en we willen ook juist het woordje ‘Bedankt!’ zien als we ‘Nee’ intypen of iets anders.

Zie meer over keuzes en lussen in hoofdstuk Structuren.

Merk ook op de nieuwe functie UPPER$(). Deze functie converteert alle tekstinvoer in hoofdletters. Echter niet wat de variabele zelf bevat. Voer je de tekst bijvoorbeeld in kleine letters in, dan zal UPPER$() het converteren en het in hoofdletters teruggeven, maar de stringinhoud in de variabele blijft dus ongewijzigd. Dit is een voorbeeld dat strings in BASIC niet te wijzigen zijn. Een andere functie die we al eerder zagen, de LEFT$() functie, doet het net zo. Die functie geeft het linkerdeel van een string of het linkerdeel van de inhoud van een stringvariabele terug, maar de inhoud van de stringvariabele zelf blijft ongewijzigd. We zullen dat later meer tegenkomen.

2i. Evaluaties van expressies

Alles wat we doen met tekst en getallen en deze willen berekenen of aaneen willen schakelen, betekent dat we uitdrukkingen of, zoals ik in het begin van dit hoofdstuk al genoemd heb, de *expressies* gaan evalueren.

Een expressie als 5 + 4 wordt geëvalueerd als 9 en 2 + 6 * 2 wordt geëvalueerd als eerst 6 * 2 dat 12 wordt en dan 2 + 12 dat 14 wordt.

Expressies kunnen we toekennen aan variabelen, maar er geldt wel een volgorde:

Niet de expressie wordt toegekend aan de variabele, maar de evaluatie, of te wel, het resultaat wordt toegekend aan de variabele.

Dit geldt ook wanneer we de expressie achter een commando gebruiken of in een functie meegeven.

```
PRINT 15 * SQR(9) + x
```

Hier is getal 9 een expressie in de functie SQR(). Het resultaat wordt met 15 vermenigvuldigd, dat geëvalueerd wordt tot 45. Pas daarna wordt de waarde van variabele x erbij opgeteld. Als de hele expressie geëvalueerd is, wordt het resultaat door het PRINT commando afgedrukt.

Onderstaande expressie is een aaneenschakeling van tekst. In de string staat de betekenis van wat aaneenschakeling betekent.

```
PRINT "Conca" + "tena" + "tie"
```

Maar in plaats van het ‘aaneenschakeling’ of ‘concatenatie’ te noemen, is er nog een derde betekenis: *samenvoeging*

Het maakt niet uit hoe je het zelf wilt noemen.

Wat ook expressies worden genoemd, zijn *condities*. Een conditie is een relationele expressie of in het Nederlands gezegd een *voorwaarde* en meer ervan bij elkaar in een logische bewerking noemen we ook een *voorwaarde*. Al deze uitdrukkingen worden vaak door elkaar genoemd. Het maakt daarom niet uit hoe je het noemen wilt. In dit boek houd ik het op relationele bewerkingen, logische bewerkingen en voorwaarden.

We kunnen in ieder geval één ding concluderen:

Een voorwaarde is een conditionele expressie, eventueel gemengd met relationele bewerkingen en met logische bewerkingen of niet gemengd met een enkele relationele bewerking.

Bijvoorbeeld:

`7 > 3`	Relationele (booleaanse) bewerking. Geeft een **waar** terug omdat 7 groter is dan 3.
`10 < 20 + 8`	Relationele bewerking met een rekenkundige bewerking. Rekenkundige bewerkingen worden eerst uitgevoerd, dan pas wordt het relationeel uitgevoerd. De relationele bewerking wordt dan 10 < 28 dat een **waar** teruggeeft.
`"Zij" < "hij"`	Tekst kan ook relationeel gecontroleerd worden. Het alfabet werkt in de computer op dezelfde volgorde, dus zou je denken dat deze voorwaarde **niet waar** is. Maar let op de kleine letter ‘h’. In de ASCII tabel hebben de kleine letters hogere waarden dan de

	hoofdletters. Dus hoofdletter kleiner dan kleine letter geeft dus een **waar**.
`a = 23 > 14 AND 11 > 8` `a = (23 > 14) AND (11 > 8)`	Beide relationele bewerkingen zijn **waar**, dus de logische bewerking AND kent een **waar** toe aan variabele a. Relationele bewerkingen hebben voorrang op de logische bewerkingen. De tweede regel met haakjes heeft dezelfde prioriteit. Haakjes zijn dus niet nodig, maar het kan de leesbaarheid verbeteren.
`b = 14 > 23 AND 11 > 8`	Omdat 14 niet groter is dan 23 geeft de logische bewerking AND toch een **niet waar**, ook al is 11 groter dan 8.
`c = 23 > 14 OR 8 > 11`	Hoewel 8 niet groter is dan 11, veroorzaakt deze voorwaarde toch een **waar**. De relationele bewerking 23 groter dan 14 zal een **waar** geven en omdat er een logische OR is, maakt het niet meer uit dat de tweede relationele bewerking **niet waar** is.
`d = 23 > 67 OR 8 > 11`	Nu beide relationele bewerkingen **niet waar** zijn, zal de logische OR een **niet waar** teruggeven.

`e = 23 > 67 XOR 11 > 8`
 	
`f = 23 > 14 XOR 11 > 8`
 	
`g = 14 > 23 XOR 8 > 11`
	Zie meer over XOR bovenaan in het tabel over de logische bewerkingen.
	Bij een Exclusief OR, dus XOR zal, wanneer minstens één van de relationele bewerkingen **waar** is, een **waar** worden teruggeven.
	Zijn beide relationele bewerkingen **niet waar** of **wel waar**, dan zal XOR een **niet waar** teruggeven.

Naast het aaneenschakelen van strings met de + operator, kunnen we het ook anders evalueren. Normaal zeggen we:

```
A$ = "A"
B$ = A$ + "2"
Snelweg$ = "De snelweg " + B$
PRINT Snelweg$
```

Maar het kan ook zo:

```
A$ = "A"
B$ = A$; 2
Snelweg$ = "De snelweg "; B$
```

Zou je zeggen:

```
B$ = A$ + 2
```

dan krijg je een foutmelding.

Maar zo kan het ook :

```
B$ = A$ + STR$(2)
```

In ieder geval zal op het mainwin venster ‘Snelweg A2’ worden geprint.

Opdracht: **Typ eens in: C = 8; 2**
Wat doet Liberty BASIC als je het uitvoert? Schrijf jouw antwoord op in commentaar.

Je mag de + operator en de puntkomma door elkaar gebruiken, maar denk eraan. Gebruik je de + operator en wil je een numerieke waarde aaneenschakelen, gebruik dan de STR$() functie. De functie is niet nodig als je de puntkomma gebruikt.

Een ander vreemd aspect is deze regel:

```
B = Tekst$ = "Woord"
```

Je zou denken dat dit een foutmelding veroorzaakt, maar dit werkt goed. Hier wordt niet Tekst$ toegekend aan B, maar juist gecontroleerd of Tekst$ gelijk is aan “Woord”. Als dit waar is, zal variabele B een 1 krijgen. Is het niet waar, dan zal B een 0 krijgen.
Variabele B zal dus werken als een booleaanse variabele.

Opdracht:
Schrijf een programma dat vraagt om twee getallen en ook zal vragen om een hoogste waarde (limiet).
Maak twee sommen met deze twee getallen: optellen en vermenigvuldigen.
Controleer of deze resultaten groter zijn dan de limietwaarde en laat de gebruiker dat weten met een PRINT commando.

Opdracht:
Schrijf een programma dat vraagt om een getal in te voeren. Gebruik twee variabelen om een minimum-waarde en een maximumwaarde eraan toe te kennen.
Bepaal met een logische AND of het getal binnen het bereik ligt.

Opdracht:
Doe hetzelfde als de vorige opdracht, maar doe het nu met een string.
Controleer of de ingevoerde tekst binnen het bereik ligt van het alfabet in hoofdletters.
Eventueel kun je een tweede controle uitvoeren of de tekst binnen het bereik ligt van het alfabet in kleine letters.

3. De functies van Liberty BASIC

In dit hoofdstuk zullen wat functies besproken worden van Liberty BASIC. Een aantal ervan zijn we al tegengekomen, maar ze zullen nogmaals worden opgenoemd met de juiste syntax.

3a. ABS(n)

Deze functie geeft de absolute waarde van n. Altijd zal het originele getal worden teruggegeven. Het minteken vervalt.

Syntax: <numerieke variabele> = ABS(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = ABS(-25)
Resultaat: Waarde van variabele n wordt 25.

3b. ACS(n)

Deze functie geeft de arc cosinus van waarde n terug.

Syntax: <numerieke variabele> = ACS(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = ACS(90 * radialen)
Resultaat: Geeft een waarde in radialen terug aan variabele n. De waarde 90 willen we in graden, maar door het te vermenigvuldigen met een radialen constante zal het in radialen berekent worden.

Zie meer hierover in Deel 2.

3c. AFTER$(s$, zoek$)

Deze functie zoekt in stringvariabele of stringwaarde s$ naar de waarde van zoek$. Indien het gevonden is, zal vanaf daar het deel in s$ worden teruggegeven. Wordt het niet gevonden, dan zal een lege string worden teruggegeven.

Syntax: <stringvariabele> = AFTER$(<stringwaarde>, <zoekstringwaarde>)
Voorbeeld: result$ = AFTER$(“hallo, ik ben thuis.”, “hallo, “)
Resultaat: Variabele result$ zal de string “ik ben thuis” terugkrijgen.

3d. AFTERLAST$(s$, zoek$)

Deze functie zoekt in stringvariabele of in stringwaarde s$ naar de waarde van zoek$. Indien het gevonden is zal, als er meer zijn, vanaf het laatst gevonden het deel van de string s$ worden teruggegeven.

Syntax: <stringvariabele> = AFTERLAST$(<stringwaarde>, <zoekstringwaarde>)
Voorbeeld: result$ = AFTERLAST$(“Dit is het laatste wat ik zoek”, “at”)
Resultaat: Variabele result$ zal de string “ ik zoek” terugkrijgen.

In de string zit het woord “at” er twee keer in. Het deel vanaf het laatst gevonden woord zal teruggegeven worden. Vandaar dat het resultaat met de spatie begint.

3e. ASC(c$)

Deze functie geeft de ascii code terug van het eerste teken van een string. Zie in de appendix het tabel met alle tekens die door Liberty BASIC worden gegeven.

Syntax: <numerieke variabele> = ASC(<stringwaarde>|<stringvariabele>)
Voorbeeld: result = ASC(“A”)
Resultaat: Geeft aan variabele result de waarde 65.

3f. ASN(n)

Deze functie geeft de arc sinus van waarde n terug.

Syntax: <numerieke variabele> = ASN(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = ASN(90 * radialen)
Resultaat: Geeft een waarde in radialen terug aan variabele n. De waarde 90 willen we in graden, maar door het te vermenigvuldigen met een radialen constante zal het in radialen berekent worden.

3g. ATN(n)

Deze functie geeft de arc tangens van waarde n terug.

Syntax: <numerieke variabele> = ATN(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = ATN(90 * radialen)
Resultaat: Geeft een waarde in radialen terug aan variabele n. De waarde 90 willen we in graden, maar door het te vermenigvuldigen met een radialen constante zal het in radialen berekent worden.

3h. CHR$(n)

Deze functie geeft het ascii teken van de gegeven n code, waarvan de code van 0 tot en met 255 gekozen kan worden.

Syntax: <stringvariabele> = CHR$(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: result$ = CHR$(65)
Resultaat: Geeft aan variabele result$ het teken A.

3i. COS(n)

Deze functie geeft de cosinus van waarde n terug.

Syntax: <numerieke variabele> = COS(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = COS(90 * radialen)
Resultaat: Geeft een waarde in radialen terug aan variabele n. De waarde 90 willen we in graden, maar door het te vermenigvuldigen met een radialen constante zal het in radialen berekent worden.

3j. DATE$(waarde) en TIME$(waarde)

In plaats van een DATE$ variabele is er nu een DATE$() functie. Deze functie kan de datum van vandaag teruggeven, de datum uit een gegeven aantal dagen berekenen vanaf 1 januari 1901 of uit een gegeven aantal dagen vanaf 1 januari 1901 de datum geven.

Syntax: <numerieke variabele>|<stringvariabele = DATE$([<numerieke waarde>|<stringwaarde>])
Voorbeelden:

```
PRINT DATE$() ' retourneert de datum van vandaag in een string
PRINT DATE$("mm/dd/yyyy") ' retourneert 11/30/1999 string
PRINT DATE$("mm/dd/yy") ' retourneert 11/30/99 string
PRINT DATE$("yyyy/mm/dd") ' retourneert 1999/11/30 string
PRINT DATE$("days") ' retourneert 36127 dagen vanaf Jan 1, 1901
PRINT DATE$("4/1/2002") ' retourneert 36980 dagen vanaf Jan 1, 1901
PRINT DATE$(36980)     ' retourneert 04/01/2002 string als een mm/dd/yyyy
                       ' met gegeven dagen vanaf Jan 1, 1901
```

In plaats van een TIME$ variabele, is er nu een TIME$() functie.

Voorbeelden:

```
PRINT TIME$()  'retourneert de tijd van nu in uren:minuten:seconden
PRINT TIME$("seconds") 'retourneert het aantal seconden vanaf middennacht
PRINT TIME$("milliseconds") 'retourneert het aantal milliseconden vanaf middennacht
```

3k. DECHEX$(n)

Deze functie geeft de hexadecimale waarde terug als een string uit de gegeven waarde n

Syntax: <string variabele> = DECHEX$(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: h$ = DECHEX$(255)
Resultaat: Geeft de hexadecimale waarde “FF” terug aan variabele h$.

3l. EOF(#handle)

Deze functie geeft een -1 terug als het einde van een bestand wordt bereikt. Zolang dat niet zo is, geeft de functie een 0.

Syntax: <EOF(#<filehandle>)
Resultaat: Wordt gebruikt in bijvoorbeeld een WHILE lus om gegevens uit een bestand te kunnen lezen zolang de functie nog geen -1 teruggeeft.

Zie meer over de functie in het hoofdstuk Bestandsbeheer.

3m. EXP(n)

Deze functie berekent e ^ n waarbij e = 2.7182818 . . .

Syntax: <numerieke variabele> = EXP(<numerieke waarde>|<numerieke variabele>)
Voorbeeld: n = EXP(5)
Resultaat Geeft de waarde 148.41315 terug aan variabele n.

3n. EVAL(expr$) en STR$(n) en VAL(s$)

Deze EVAL() functie evalueert de gegeven string expressie naar een numeriek resultaat. Het resultaat wordt berekent zoals een normale expressie berekent wordt. Arrays en functies zijn ook toegestaan.

Syntax: <numerieke variabele> = EVAL(<string expressie>|<string variabele>)
Voorbeelden:

```
n = EVAL("20 + 20")          'retourneert de waarde 40
n = EVAL("INT(17 / 4)")      'voert een integerdeling uit, retourneert de waarde 4
```

```
n = EVAL("Oppervlakte(8, 4)") 'berekent de oppervlakte in een gebruikersfunctie
                              'de EVAL functie retourneert het resultaat van de
                              'functie Oppervlakte
```

Een andere functie, EVAL$(expr$), geeft hetzelfde resultaat als de functie EVAL, maar retourneert het resultaat als een string.
In plaats van de EVAL$ functie, doet de code STR$(EVAL(n$)) hetzelfde.

De functie STR$ converteert een numerieke waarde naar een stringwaarde, zoals te zien is bij de functie EVAL. De tegenhanger is de functie VAL. Die functie converteert een stringwaarde naar een numerieke waarde, als dat mogelijk is.

Voorbeelden:

```
s$ = STR$(n)          'converteert waarde n en kent het toe aan s$
n = VAL(s$)           'converteert waarde van s$ en kent het toe aan n
```

Indien bij de VAL functie de waarde van s$ een geldige numerieke waarde is, wordt het geconverteerd naar een numerieke waarde. Is het geen geldige numerieke waarde in s$, dan wordt de waarde een 0.

3o. HBMP(s$)

Deze functie retourneert de vensterhandle voor de ingeladen bitmap. De gegeven string s$ is de naam die gegeven moet worden bij het LOADBMP commando.

Syntax: <numerieke variabele> = HBMP(<stringwaarde>|<stringvariabele>)
Voorbeeld:

```
LOADBMP "title", "bmp\title.bmp"
handle = HBMP("title")
print handle
```

Resultaat: Geeft een numerieke adreswaarde terug aan variabele handle.

Deze functie is nuttig te gebruiken bij Windows API functies die te maken hebben met de GDI32. Adreswaarden van bitmaps of pixels moeten dan meegegeven kunnen worden bij een CALLDLL aanroep. De bitmapstrings kunnen niet direct meegegeven worden en er moet dus een adres door de HBMP functie worden teruggegeven die in een API functie nodig is.

Dieper uitleg over Windows API vergt een moeilijke techniek en valt daarom buiten dit boek.

3p. HEXDEC(h$)

Deze functie geeft de decimale waarde terug van een gegeven hexadecimale waarde. De hexadecimale waarde moet als een string worden gegeven.

Syntax: <numerieke variabele> = HEXDEC(<hexadecimale stringwaarde>|<hexadecimale stringvariabele>)
Voorbeeld: n = HEXDEC(“FF”)
Resultaat: Geeft de decimale waarde 255 terug aan variabele n. Het getal 255 is gelijk aan de hexadecimale waarde FF.

3q. HTTPGET$(url$)

Deze functie doet een HTTP-verzoek naar de adres-URL die is opgegeven in de url$ variabele en retourneert de inhoud van de webpagina voor dat adres als deze kan worden opgehaald. Dit is meestal een tekstreeks in HTML-formaat.

Syntax: <html variabele$> = HTTPGET$(<url stringwaarde>|<url stringvariabele>)
Voorbeeld:

```
htmltext$ = httpget$("http://www.libertybasic.com")
print htmltext$
```

Resultaat: Retourneert de inhoud van het gegeven adres in HTML formaat.

Opmerking: Als er geen pagina op dat adres wordt gevonden, kan de internetprovider soms een geldig HTML-document met foutdetails retourneren. Uw programma moet dit detecteren en adequaat reageren.

3r. HWND(#handle)

Deze functie retourneert het adres van een gegeven handle. De handle kan een venster of een control zijn. Het teruggegeven adres is nuttig bij het aanroepen van Windows API functies, omdat de functies de adressen van vensters en/of controls gebruiken en niet de handle zelf.

Syntax: <adresvariabele> = HWND(#<vensterhandle>[.<controlnaam>])
Voorbeelden: handle1 = HWND(#w)
handle2 = HWND(#w.btnKnop)
Resultaat: Variabele handle1 krijgt het adres van het venster #w.
Variabele handle2 krijgt het adres van de knop btnKnop die in venster #w staat.

3s. IDECODE$() en IDEFILENAME$()

De functie IDECODE$() retourneert de code inhoud van de Liberty BASIC editor. De functie IDEFILENAME$() retourneert de volle pad met bestandsnaam waar de code in bewaard is.

Meer over deze functie, het gebruik ervan, enzovoort, zie de Help.

Syntax en voorbeeld: code$ = IDECODE$()
pad$ = IDEFILENAME$()

Als de uitvoer van de code buiten de editor gebeurt, retourneren beide functies een lege string.

3t. INSTR(s1$, s2$, n)

Deze functie retourneert de plaats in string s1$ vanaf waar s2$ gevonden wordt. Indien er een start n wordt meegegeven, zal er vanaf plaats n worden gezocht. Zonder gegeven start n zal er vanaf de eerste plaats worden gezocht.
Indien de string van s2$ niet gevonden wordt, resulteert de functie een 0.

Syntax: <numerieke variabele> = INSTR(<stringwaarde>|<stringvariabele>,<stringwaarde>|<stringvariabele>[, <numerieke waarde>|<numerieke variabele>])
Voorbeelden:

```
n = INSTR("Dit is een test", "is")                  'retourneert de waarde 5
n = INSTR("Dit is een zoekstring", "zoek", 13)      'retourneert een 0, geen "zoek"
                                                    'vanaf plaats 13
```

3u. LEFT$(s$, n) en MID$(s$, m, n) en RIGHT$(s$, n)

Deze functies retourneren delen uit de s$ stringwaarden of variabelen.
De functie LEFT$() retourneert het linkerdeel van de string s$ met lengte n.
De functie MID$() retourneert het middendeel van de string s$ vanaf plaats m met lengte n.
De functie RIGHT$() retourneert het rechterdeel van de string s$ met lengte n vanaf het laatste teken.

Syntax: <stringvariabele> = LEFT$(<stringwaarde>|<stringvariabele>, <numieke waarde>|<numerieke variabele>)
<stringvariabele> = MID$(<stringwaarde>|<stringvariabele>, <numerieke waarde>|<numerieke variabele>[,<numerieke waarde>|<numerieke variabele>])
<stringvariabele> = RIGHT$(<stringwaarde>|<stringvariabele>, <numerieke waarde>|<numerieke variabele>)
Voorbeelden:

```
d$ = LEFT$("Dit is een test", 6)     'retourneert "Dit is"
d$ = MID$("Dit is een test", 5, 6)   'retourneert "is een"
d$ = MID$("Dit is een test", 5)      'retourneert "is een test"
d$ = RIGHT$("Dit is een test", 8)    'retourneert "een test"
```

Bij ongeldige waarden van de parameters m en n geeft Liberty BASIC geen foutmelding. Er zal niets geresulteerd worden.

3v. LEN(s$) en LEN(struct)

Deze functie geeft de lengte van een string terug of de lengte van een structure.

Syntax: <numerieke variabele> = LEN(<stringwaarde>|<stringvariabele>)|LEN(<structnaam>)
Voorbeelden:

```
n = LEN("De lengte van deze string")
'retourneert de lengte van de gegeven string

STRUCT Fruitmand, Appels AS ULONG, Peren AS ULONG, Bananen AS ULONG
n = LEN(Fruitmand.struct)
'retourneert de lengte van de Fruitmand structure.
```

Meer over de functie LEN en de structures, zie hoofdstuk Datastructuren en datatypes in Deel 3.

3w. LOWER$(s$) en UPPER$(s$)

Deze functies geven de gegeven string s$ terug in kleine letters (LOWER) of in hoofdletters (UPPER) terug. De gegeven string s$ blijft ongewijzigd.

Syntax: <stringvariabele> = LOWER$(<stringwaarde>|<stringvariabele>)
<stringvariabele> = UPPER$(<stringwaarde>|<stringvariabele>)
Voorbeeld: sl$ = LOWER(“Dit is”)
su$ = UPPER$(“Dit is”)
Resultaat: Variabele sl$ krijgt de string “dit is” terug.
Variabele su$ krijgt de string “DIT IS” terug.

3x. MIN(m, n) en MAX(m, n)

De functie MIN geeft de laagste waarde van de gegeven waarden m en n.
De functie MAX geeft de hoogste waarde van de gegeven waarden m en n.

Syntax: <numerieke variabele> = MIN(<numerieke waarde>|<numerieke variabele>, <numerieke waarde>|<numerieke variabele>)
<numerieke variabele> = MAX(<numerieke waarde>|<numerieke variabele>, <numerieke waarde>|<numerieke variabele>
Voorbeeld: n1 = MIN(3, 2)
n2 = MAX(3, 2)
Resultaat: Variabele n1 heeft de laagste waarde gekregen: 2
Variabele n2 heeft de hoogste waarde gekregen: 3

De volgorde van de parameters maakt niets uit. Gegeven (2, 3) geeft hetzelfde resultaat.

3y. REMCHAR$(s$, cs$) en REPLSTR$(s$, z$, v$)

De functie REMCHAR$ is een filterfunctie. Het haalt de gegeven tekens die in cs$ staan uit de string s$ en retourneert de nieuwe string.

Syntax: <stringvariabele> = REMCHAR$(<stringwaarde>|<stringvariabele>, <filtertekenstring>|<filtertekenstringvariabele>)
Voorbeeld:

```
PRINT REMCHAR$("Dizt ixs emen trest", "zxmr")
```

Resultaat: Print de string “Dit is een test” op de mainwin.

Bij het gebruik van deze functie moeten de juiste filtertekens gekozen worden. Indien een filterteken wordt gekozen die in de string hoort, wordt deze pardoes uitgefilterd. Plaats in de filterstring maar eens de letter *e* erbij en zie wat er gebeurd.

De functie REPLSTR$ is een zoek- en vervangfunctie. Andere BASIC dialecten kennen wel het MID$ commando dat ook zoiets doet. Het verschil is dat we in deze functie kunnen zoeken naar een string en die kunnen vervangen.

Syntax: <stringvariabele> = REPLSTR$(<stringwaarde>|<stringvariabele>, <stringwaarde>|<stringvariabele>, <stringwaarde>|<stringvariabele>)
Voorbeeld: s$ = REPLSTR$("Een soek en vervang functie", "soek", "zoek")
Resultaat: Variabele s$ krijgt de nieuwe string "Een zoek en vervang functie"

3z. WORD$(s$, n, sc$)

Deze functie is een krachtige en handige functie. Alleen Liberty BASIC kent deze functie.

Met deze functie kan gezocht worden naar woorden en uit de string worden gehaald. De eerste parameter s$ is de hele string waar de woorden inzitten die uitgefilterd moeten worden.
In de string moeten één dezelfde tekens in voorkomen dat WORD$ nodig heeft om elk woord te kunnen filteren. Dit wordt per woordnummer n gedaan per scheidingsteken sc$.

De functie kan niet zelf waarde n verhogen. Daarom moet een lus worden gebruikt om door de string te itereren, zodat de functie elk woord terug kan geven zodra het gegeven teken sc$ gevonden wordt.

Syntax: <stringvariabele> = WORD$(<stringwaarde>|<stringvariabele>, <numerieke waarde>|<numerieke variabele>[, <delimiterstringwaarde>|<delimiterstringvariabele>])
Voorbeeld:

```
a$ = "Deze string wordt gefilterd door elk woord terug te krijgen"
i = 1
n$ = " "
DO WHILE n$ <> ""
    n$ = WORD$(a$, i, " ")
    PRINT n$
    i = i + 1
LOOP
```

```
Deze
string
wordt
gefilterd
door
elk
woord
terug
te
krijgen
```

In dit voorbeeld kan ook het derde argument " " weggelaten worden. Zonder gegeven scheidingsteken zoekt de functie automatisch naar een spatie als scheidingsteken.

De functie is zeer geschikt om CSV bestanden, die de komma als scheidingsteken hebben, in één string in te laten lezen. Er kan dan net zo'n lus als deze uitgevoerd worden om alles bijvoorbeeld in een array op te slaan.

Er zijn uiteraard nog meer functies, maar dit waren ze even van paragraaf a tot en met paragraaf z.

4. Programma's schrijven

Eerder werd veel uitgelegd over programmeren, wat BASIC is, en hoe we omgaan met constanten, variabelen en uitdrukkingen in BASIC.
Maar dat is niet alles om te leren programmeren in BASIC. Na de tijd van de oude BASIC dialecten is er veel bijgekomen. De programmeertaal BASIC is uitgebreider geworden, maar zeker niet veranderd. Hoewel dat door velen wel wordt beschouwd; BASIC zou niet geschikt zijn om programmeren te leren en zou wegsterven. Er wordt dan gedacht dat andere programmeertalen, zoals Python of Pascal, beter zou zijn om te leren. BASIC is dan een vieze taal geworden.

Maar niets is minder waar. BASIC staat nog steeds overeind en we kunnen met dezelfde BASIC syntax nu ook programmeren in Windows. Zelf Windows programma's schrijven, wie zou dat niet willen leren programmeren? Maar voordat we gaan leren Windows programma's te schrijven, is het beter kennis te krijgen in de syntax van Liberty BASIC. We gaan het mainwin venster gebruiken om waarden af te drukken.

4a. Hoe schrijven we een programma

Liberty BASIC heeft geen interpreter, maar een compiler. Als we een regel code typen, zal het niet direct uitgevoerd worden. We moeten eerst op de blauwe pijl klikken of **Run** kiezen in het menu **Run** of de sneltoets **Shift-F5** indrukken.
Typ onderstaande regel in:

```
PRINT "De som 20 + 40 = "; 20 + 40
```

Dit statement wordt niet direct uitgevoerd. Door zelf op de blauwe pijl te klikken, wordt het mainwin venster geopend en zien we onderstaande tekst verschijnen:

```
De som 20 + 40 = 60
```

We hadden ook de regel zo kunnen typen:

```
10 PRINT "De som 20 + 40 = "; 20 + 40
```

Als we het uitvoeren, zal dezelfde tekst verschijnen.
Typ eens onderstaand programma in:

```
10 INPUT "Hoe heet je: "; Naam$
20 INPUT "Wat is jouw leeftijd: "; Leeftijd
30 PRINT "Jouw naam is: "; Naam$
40 PRINT "Je bent "; Leeftijd; " jaar oud."
50 LET Geboortejaar = 2023 - Leeftijd
60 PRINT "Jouw geboortejaar is: "; Geboortejaar
70 PRINT
80 GOTO 10
```

Als voorbeeld zien we het volgende op het venster:

```
Hoe heet je: Klaas
Wat is jouw leeftijd: 34
Jouw naam is: Klaas
Je bent 34 jaar oud.
Jouw geboortejaar is: 1989

Hoe heet je: Morgan
Wat is jouw leeftijd: 56
Jouw naam is: Morgan
Je bent 56 jaar oud.
Jouw geboortejaar is: 1967

Hoe heet je:
```

Zoals je ziet blijft de uitvoer maar doorgaan. Dat komt door regel 80. Regel 80 vertelt Liberty BASIC om weer terug te springen naar regel 10.

Druk op Control (Ctrl-toets) en Break (Pause/Break-toets) zodat de uitvoer gestopt wordt. Je krijgt wel even een **keyboard interrupt** melding, maar daar hoef je niets van aan te trekken. Na het sluiten van deze melding kun je klikken op het kruisje om het mainwin venster te sluiten.

Liberty BASIC komt weer met een venster. Deze keer wordt er gevraagd of je de uitvoer van het programma echt wilt sluiten. Klik op **Ja**.

In Liberty BASIC hoeven we de commando's niet in hoofdletters te schrijven. We mogen het in kleine letters schrijven of het mixen. Dat mag ook in variabele namen, maar we moeten wel uitkijken dat we niet een variabele naam **Naam** en een variabele naam **naam** in één programma gaan gebruiken. Liberty BASIC geeft dan een foutmelding. We mogen de letters van de variabele namen wel mixen met hoofdletters en kleine letters, maar de namen niet dubbel gebruiken.

4b. (Branch) labels

Het programma uit paragraaf 4a bestond uit regelnummers.
Bekijk het programma nog eens. Haal de regelnummers weg behalve regelnummer 10, zoals je hieronder ziet staan:

```
10 INPUT "Hoe heet je: "; Naam$
INPUT "Wat is jouw leeftijd: "; Leeftijd
PRINT "Jouw naam is: "; Naam$
PRINT "Je bent "; Leeftijd; " jaar oud."
```

```
LET Geboortejaar = 2023 - Leeftijd
PRINT "Jouw geboortejaar is: "; Geboortejaar
PRINT
GOTO 10
```

Run het programma nog eens. Zoals je ziet zal er niets anders zijn. Het enige wat er wel moet blijven staan is regelnummer **10** of anders gezegd: **label 10**.
Het GOTO commando moet weten naar waar hij naartoe moet springen. Daar kunnen we dus een getal als **10** voor gebruiken, maar het kan in Liberty BASIC ook anders. In plaats van getallen, zijn naam-labels ook toegestaan.

Een label in een regel wordt ook wel een **branch label** genoemd. Het woord branch betekent **tak** of **richting**.

Omdat we echter vertakken, of te wel springen naar, met een GOTO of een GOSUB, is een label op zich geen branch. Het is alleen maar een naam die we gebruiken om daarheen te branchen. Als er zomaar een label staat, zoals een regelnummer, en er wordt daar niet heen gesprongen, dan doet de label zelf niets.

In de Help van Liberty BASIC wordt een label wel een branch label genoemd, maar je kunt het beter een label of labelnaam noemen.

In plaats van regelnummer 10 als label, zouden we het ook **start** kunnen noemen.
Typ onderstaand programma in:

```
[start]
input "Geef een getal: "; getal
a = a + 1
print "De vermenigvuldiging "; a; " * "; getal; " = "; a * getal
goto [start]
```

De naam van een label moet altijd tussen rechte haken staan. De labelnamen mogen net als variabele namen worden gemixt met hoofdletters, kleine letters en cijfers. Maar uitgezonderd mogen symbolen ook en je mag zelfs beginnen met een cijfer.

Zoals je ziet mag de code in kleine letters geschreven worden, maar ik zal in dit boek in hoofdletters blijven om verwarring met variabele namen te voorkomen.

4c. Regels inspringen

Bovenstaand programma kan goed gelezen worden, maar naarmate een programma groter wordt, gaat het onleesbaar worden of is het slecht te volgen. In Liberty BASIC kunnen we regels inspringen om een programma meer leesbaar te maken.
Het programma, zoals je in de vorige paragraaf ziet, kunnen we ook zo schrijven:

```
[Start]
        INPUT "Geef een getal: "; getal
        a = a + 1
        PRINT "De vermenigvuldiging "; a; " * "; getal; " = "; a * getal
        GOTO [Start]
```

We kunnen het programma ook uitbreiden, zodat het niet meer oneindig wordt uitgevoerd. Je zult zien dat het inspringen best wel belangrijk is.

```
[Start]
        INPUT "Geef een getal: "; getal
        IF getal = 0 THEN END
        a = a + 1
        PRINT "De vermenigvuldiging "; a; " * "; getal; " = "; a * getal
        GOTO [Start]
```

Of we schrijven het zo:

```
[Start]
        INPUT "Geef een getal: "; getal
        IF getal <> 0 THEN
                a = a + 1
```

```
            PRINT "De vermenigvuldiging "; a; " * "; getal; " = "; a * getal
            GOTO [Start]
        END IF
END
```

Als de waarde van variabele getal ongelijk is aan nul zullen de drie regels, die ingesprongen tussen IF en END IF staan, worden uitgevoerd. Omdat daar GOTO [Start] staat, zal er gesprongen worden naar de label [Start] en wordt nogmaals om een getal gevraagd.

Is de relationele bewerking getal <> 0 echter NIET WAAR, dan wordt het hele codeblok niet uitgevoerd en gaat de uitvoer verder bij het commando END. Hoewel er geen regels meer uit te voeren zijn, is het END commando niet speciaal nodig. Er zijn situaties dat we END toch nodig hebben, zoals we in het vorige programma zagen.

4d. Commentaar

Vaak is een regel code niet gemakkelijk te begrijpen. Het zou dan handig zijn om in het Nederlands wat tekst met uitleg erbij te kunnen schrijven.

In sommige BASIC dialecten wordt het REM commando gebruikt om commentaar in het programma te geven. Liberty BASIC kent ook het REM commando. Onderstaande code kunnen we uitvoeren:

```
REM Dit is een voorbeeldprogramma
PRINT "Dit is een test."
```

Alleen de tekst achter het PRINT commando wordt afgedrukt.
We willen niet altijd dat een regel met commentaar begint:

```
PRINT "Een tekstuitvoer": REM Commentaar achter een statement
```

Het commando REM kan niet achter een deel van een statement komen te staan, zoals dit:

```
PRINT A, _      : REM Dit is een PRINT met eerste meegegeven variabele A
     B          : REM Dit is dezelfde PRINT met tweede meegegeven variabele B
```

Liberty BASIC kent een apostrof-teken ‘ ’ ’. Het teken doet hetzelfde als het REM commando doet, maar is veel flexibeler. Het commentaarteken mag overal staan, behalve tussen de coderegels in.

5. Structuren

Programma’s worden uitgevoerd in een bepaalde volgorde. In de computertaal wordt een volgorde een structuur genoemd. In oude BASIC dialecten waren de structuren beperkt en door slechte code werd het snel spaghetti-code.
Liberty BASIC kent, samen met andere BASIC programmeertalen, zoals Q(uick)BASIC, goede structuren om programma’s goed en leesbaar te kunnen schrijven.

5a. Wat is een structuur

Een structuur is dus een volgorde, maar een volgorde hoeft niet sequentieel (opéénvolgend) te zijn. Sommige sleutelwoorden bepalen zelf de uitvoervolgorde, zoals we in de vorige hoofdstukken zagen. Een GOTO commando bepaalt de uitvoervolgorde om ergens naartoe te springen, terwijl een IF statement bepaalt om iets uit te voeren als een relationele bewerking (ook wel een voorwaarde of conditie genoemd) WAAR is. Als het NIET WAAR is, zal de volgorde anders worden uitgepakt en veranderd de uitvoer.
De uitvoervolgorde hoeft niet hetzelfde te zijn als de structuur in het programma. Onderstaande twee programma’s hebben verschillende structuren, maar de uitvoer is hetzelfde:

```
PRINT "Getal 1"
PRINT "Getal 2"
PRINT "Getal 3"
PRINT "Getal 4"
PRINT "Getal 5"
END
```

```
FOR getal = 1 TO 5
        PRINT "Getal "; getal
NEXT getal
```

5b. GOTO commando, valstrik of hulpmiddel

Het spaghetti BASIC commando, dat samen met het GOSUB commando, nog steeds bestaat, is het GOTO commando. De structuren zijn tegenwoordig zo uitgebreid, dat je niet meer in BASIC deze oude commando's hoeft te gebruiken. Het GOTO commando kan zelfs in de weg liggen en kan voor de gekste bugs zorgen.

Toch blijkt dat de meeste BASIC gebruikers het GOTO commando nog steeds fijn vinden en het zelfs samen met de modernere structuren gebruiken. In plaats van dat het fouten kan veroorzaken, kan een GOTO of een GOSUB ook wel eens helpen om een moeilijke structuur, dat voor een ander statement niet of nauwelijks op te lossen is, te ondersteunen.

Het gebruik van GOTO en GOSUB zal, nu we modernere statements en structuren hebben, voor minder spaghetticode zorgen. Dit betekent dat we het kunnen inpassen wanneer we denken het nodig te hebben en kunnen we de belangrijkste zaken overlaten aan de modernere code.

Maar teveel gebruik van het GOTO commando kan ook averechts werken. Bedenk dus goed wanneer het commando nodig is en zorg ervoor dat je programma leesbaar en begrijpelijk blijft.
Het mixen van deze twee commando's en het gebruik van de tegenwoordige programmeertechniek zal in hoofdstuk 9 verder besproken worden.

5c. Keuzestructuren

In de vorige hoofdstukken zagen we relationele bewerkingen. Deze bewerkingen geven altijd een WAAR of een NIET WAAR terug. Deze resultaten zijn booleaanse waarden.
Liberty BASIC kent geen TRUE of FALSE, maar kan wel onderscheid maken of iets waar of onwaar is. Als het waar is dan is het resultaat een 1. Is het niet waar dan is het een 0.
Het hoeft niet altijd een 1 te zijn om waar te zijn. Alle getallen die niet nul zijn, zijn waar.

Syntax:
IF <voorwaarde> THEN waar-blok ELSE onwaar-blok
IF <voorwaarde> THEN
        waar-blok
ELSE
        onwaar-blok
END IF

Typ eens onderstaand programma in:

```
INPUT "Geef een waarde: "; waarde
IF waarde THEN
        PRINT "Is waar"
ELSE
        PRINT "Is niet waar"
END IF
```

Je ziet dat elke waarde ongelijk aan nul altijd waar is.
Als je het niet duidelijk vindt, kun je het ook zo schrijven:

```
INPUT "Geef een waarde: "; waarde
IF waarde <> 0 THEN
        PRINT "Is waar"
ELSE
        PRINT "Is niet waar"
END IF
```

De functie NOT() geeft het tegenovergestelde. Als een bewerking waar is, zal het een niet waar worden.
Omdat een bewerking dat waar is een 1 teruggeeft, zal de NOT() functie het als een -1 teruggeven. Als we dus typen:

```
PRINT NOT(1) = 1
```
dan zal op het venster een 0 verschijnen.

En bij een:

```
PRINT NOT(0) = -1
```

dan zal op het venster een 1 verschijnen.
Liberty BASIC is de enige BASIC programmeertaal dat een NOT() functie heeft. Andere BASIC programmeertalen, en zelfs andere programmeertalen dan BASIC, kennen een NOT operator. Het is dan ook een sleutelwoord, maar in Liberty BASIC dus niet.

We kunnen ook logische bewerkingen uitvoeren.
Probeer onderstaande code te begrijpen. Het heeft geen doel, maar het laat zien wat er gebeurt met de voorwaarden. De tekst achter de PRINT commando's vertellen wanneer iets waar is en wanneer niet. Je kunt de stappen onderaan ook doornemen. Die leggen ook uit wat de IF statements doen.

```
INPUT "Waarde 1: "; Waarde1
INPUT "Waarde 2: "; Waarde2
IF Waarde1 > 0 AND Waarde2 > 0 THEN
       PRINT "Waarde 1 en Waarde 2 zijn beide groter dan nul."
ELSE
       IF Waarde1 = 0 THEN PRINT "Waarde 1 is gelijk aan nul."
       IF Waarde2 = 0 THEN PRINT "Waarde 2 is gelijk aan nul."
       IF Waarde1 < 0 AND Waarde2 < 0 THEN _
              PRINT "Waarde 1 en Waarde 2 zijn beide kleiner dan nul."
END IF
IF Waarde1 = Waarde2 OR Waarde2 > 50 THEN
       PRINT "Waarde1 kan gelijk zijn aan Waarde2 of Waarde2 kan groter zijn dan 50."
ELSE
       PRINT "Waarde1 is waarschijnlijk niet gelijk aan Waarde2 of ";
       PRINT "Waarde2 is waarschijnlijk niet groter dan 50."
END IF
```

Het programma doet het volgende:

Stap 1	Er wordt om een getal gevraagd voor Waarde1.
Stap 2	Er wordt om een getal gevraagd voor Waarde2.
Stap 3	Er wordt met een logische bewerking AND twee relationele bewerkingen uitgevoerd. Indien Waarde1 groter is dan 0 EN Waarde2 groter is dan 0, dan zullen beide bewerkingen waar zijn en wordt de volgende PRINT regel uitgevoerd.
Stap 4	Als Stap 3 niet waar is, wordt er geen PRINT regel van Stap 3 uitgevoerd en gaat de uitvoer verder na het sleutelwoord ELSE. Er wordt een relationele bewerking uitgevoerd die test of Waarde1 gelijk is aan 0. Zo ja, dan zal het PRINT commando achter THEN worden uitgevoerd.
Stap 5	Zelfde relationele bewerking als bij Stap 4, maar nu bij variabele Waarde2.
Stap 6	Er wordt met een logische bewerking AND twee relationele bewerkingen uitgevoerd. Indien Waarde1 kleiner is dan 0 EN Waarde2 kleiner is dan 0, dan zullen beide bewerkingen waar zijn.
Stap 7	Als Stap 6 waar is, dan zal op de volgende regel een PRINT commando worden uitgevoerd. Let op de underscore '_' achter THEN. Liberty BASIC ziet dan de volgende regel alsof deze achter THEN staat. Het is belangrijk om dan ook de PRINT regel in te laten springen, ook al is er nu geen codeblok met een afsluitende END IF.
Stap 8	Na het commando END IF de hele IF structuur heeft afgesloten, gaat de uitvoer verder met een volgende logische bewerking.

Stap 9	Indien Waarde1 gelijk is aan Waarde2 OF Waarde2 groter is dan 50, dan zal het deze keer niet gaan of beide bewerkingen waar zijn, maar nu of de ene of de andere waar is.
Stap 10	Als dus de ene of de andere waar is, zal het PRINT commando de tekst afdrukken. In de tekst wordt er gespeculeerd welke waar kan zijn, want met een OR kunnen we daar zelf niet achter komen.
Stap 11	Eén ding weten we wel. Als beide bewerkingen niet waar zijn, zal het PRINT commando niet worden uitgevoerd en gaat de uitvoer verder na het ELSE sleutelwoord.
Stap 12	Nu worden de laatste twee PRINT statements uitgevoerd. Merk op dat bij Stap 11 een **kan** wordt gezegd, terwijl in deze twee PRINT statements een **waarschijnlijk niet** wordt gezegd. Bij een waar kan dus de ene of de andere waar zijn, maar bij een niet waar is dat dus waarschijnlijk niet. We kunnen niet zeggen dat het **niet kan**, want als programmeur weten we niet welke.

Merk op dat we IF statements mogen gebruiken in een codeblok van een ander IF statement. Dit noemen we het nesten van structuren. Eenvoudige commando's, zoals het PRINT commando, valt niet onder structuren. Deze commando's worden altijd op volgorde uitgevoerd en kunnen niet voor een ander structuur zorgen. We kunnen die dan ook niet nesten.

Naast de welbekende IF … THEN … ELSE … END IF structuur, kent Liberty BASIC nog een andere keuzestructuur.
Stel dat je een variabele of uitdrukking drie keer wilt laten vergelijken op een andere variabele of uitdrukking. We zouden onderstaand programma kunnen schrijven:

```
INPUT "Welk element? "; element
IF element = 0 THEN
        PRINT "Je wilt liever thuis blijven."
ELSE
        IF element = 1 THEN
                PRINT "Ga je boodschappen doen?"
        ELSE
                IF element = 2 THEN
                        PRINT "lekker uit eten, mmmmm!"
                ELSE
                        PRINT "Onbekend element!"
                END IF
        END IF
END IF
```

In andere BASIC versies en dialecten is het mogelijk om het zo te schrijven:

```
INPUT "Welk element? "; element
IF element = 0 THEN
        PRINT "Je wilt liever thuis blijven."
ELSEIF element = 1 THEN
        PRINT "Ga je boodschappen doen?"
ELSEIF element = 2 THEN
        PRINT "Lekker uit eten, mmmmm!"
ELSE
        PRINT "Onbekend element!"
END IF
```

Helaas ondersteunt Liberty BASIC geen ELSE met erachter een nieuwe IF. Elke IF structuur moet in een aparte codeblok geschreven worden. Zoals je het verschil ziet, kost het in Liberty BASIC meer inspringen en END IF regels en misschien meer tijd om de code te begrijpen.

Zoals ik van vele hobbyisten begrepen heb, is die manier toch handiger dan een ELSEIF. Voor mij was het even wennen, maar ik zie nu in dat het vaak beter is om een volgende IF na een ELSE in te springen.
Een ander voordeel is dat elke geneste IF … END IF blok een eigen ELSE kan hebben, in tegenstelling tot blokken ELSEIF die dat niet zelf hebben.

Liberty BASIC kent nog een kortere schrijfwijze om op hetzelfde neer te komen. Q(uick)BASIC en Visual Basic kennen die korte schrijfwijze ook.

Neem onderstaand programma over en probeer de nieuwe structuur te begrijpen:

```
INPUT "Welk element? "; element
SELECT CASE element
        CASE 0
                PRINT "Je wilt liever thuis blijven."
        CASE 1
                PRINT "Ga je boodschappen doen?"
        CASE 2
                PRINT "Lekker uit eten, mmmmm!"
        CASE ELSE
                PRINT "Onbekend element!"
END SELECT
```

Het voordeel is nu dat elke waarde of uitdrukking vergeleken kan worden met de gegeven variabele element achter SELECT CASE.

Liberty BASIC mist echter wel een belangrijke mogelijkheid: het controleren of een relationele bewerking waar of niet waar is. In Visual Basic kunnen we het sleutelwoord **is** toepassen als we een relationele bewerking willen controleren.

Als we relationele bewerkingen achter de CASE willen controleren, dan kan dat op een manier dat alleen in Liberty BASIC is toegestaan. Andere BASIC programmeertalen kennen alleen de bovenstaande SELECT CASE structuur.

Wijzig bovenstaand programma zo, dat je onderstaande code krijgt:

```
INPUT "Welk element? "; element
SELECT CASE
        CASE element < 0
                PRINT "Je bent negatief!"
        CASE element = 0
                PRINT "Je wilt liever thuis blijven."
        CASE element = 1
                PRINT "Ga je boodschappen doen?"
        CASE element = 2
                PRINT "Lekker uit eten, mmmmm!"
        CASE ELSE
                PRINT "Onbekend element!"
END SELECT
```

Omdat het mogelijk is zelf relationele bewerkingen met een vooraf gegeven variabele of uitdrukking achter een CASE te schrijven, werkt een SELECT CASE structuur in Liberty BASIC veel functioneler. Je bent niet meer beperkt tot aan één variabele of uitdrukking die je met een SELECT CASE kunt controleren. Je bent vrij om achter elke CASE commando een andere relationele bewerking te nemen.

De CASE ELSE werkt altijd op dezelfde manier, hoe je ook de SELECT … END SELECT structuur toepast.

Achter elke CASE kan elke relationele bewerking staan, maar ook logische bewerkingen zijn toegestaan.

Breid het programma uit door onderstaande CASE boven de CASE ELSE te plakken:

```
    CASE element >= 0 AND element <= 10
       PRINT "Dit element ligt in het bereik van 0 tot en met 10."
```

Zou je zeggen:
Element > 0 AND element < 10
dan betekent dit als: element ligt in het bereik van 1 tot en met 9 of wel 1 tot 10.
De getallen 0 en 10 vallen er dan buiten.
Net als bij een IF … THEN … END IF, mag je ook een SELECT CASE … END SELECT nesten. Dat wil zeggen: achter een CASE mag je een nieuwe SELECT CASE … END SELECT schrijven. Zowel bij een IF structuur als bij een SELECT CASE structuur is het nesten onbeperkt mogelijk.

Er is een techniek om, alles wat te ver ingesprongen wordt, eruit te knippen en apart in codeblokken te schrijven. Deze codeblokken zijn dan *aan te roepen* door ze namen te geven.
Hoewel zulke codeblokken ook met structuren te maken hebben, zal ik dit in een apart hoofdstuk, hoofdstuk 9, behandelen.

5d. Lusstructuren

BASIC kent één lusstructuur zonder relationele bewerking. Het is de FOR … NEXT lus.
In voorgaande hoofdstukken werd deze lus al genoemd. De codeblok in deze lus wordt een aantal keren uitgevoerd, aan de hand van een opgegeven start en eind met eventueel in aantal stappen.

Syntax:
FOR <variabele> = <uitdrukking> TO <variabele> = <uitdrukking> [STEP <n>] … NEXT [<variabele>]

Onderstaand voorbeeld laat zien hoe een FOR … NEXT lus werkt.

```
FOR i = 1 TO 10
        PRINT i; " ";
NEXT i  'hier is variabele i optioneel
```

Dit drukt af:

```
1 2 3 4 5 6 7 8 9 10
```

De tekst achter de apostrof is een commentaarregel, zie in paragraaf 4b meer over commentaar.
Alleen bij gebruik van het REM commando moet er vooraf een dubbelepunt ‘:’ staan. Dat mag hier ook, maar een apostrof kost minder schrijfwerk en houdt bovendien code en uitlegtekst uit elkaar. Een dubbelepunt hoeft hier dus niet en een spatie is niet nodig, zoals je ziet.

We mogen een NEXT commando gebruiken zonder de variabele op te geven, die als teller achter FOR werkt.

Onderstaande code laat het zien met een STEP commando.

```
FOR even = 0 TO 20 STEP 2
        PRINT even; " ";
NEXT
```

We kunnen ook omlaag tellen in een FOR lus.

```
FOR getal = 20 TO 1 STEP -1
        PRINT getal
NEXT getal
```

Omlaag tellen met stappen van 1 kunnen we niet zonder het STEP commando doen. Alleen met omhoog tellen met stappen van 1 kan zonder het STEP commando.

Net als in een keuzestructuur kunnen we ook lussen nesten, zoals hieronder:

```
FOR j = 1 TO 10
        FOR i = 1 TO 10
                Tabel(i, j) = i * j
                'eventueel nog meer code hier
        NEXT i
NEXT j
```

Maar in de oude BASIC dialecten mogen alle variabelen van de FOR statements achter het NEXT commando opgegeven worden, gescheiden met komma’s. Tegenwoordig mag dat niet meer.
Als je probeert te schrijven:

```
NEXT i, j
```

Dan zal de compiler van Liberty BASIC dit niet toestaan.

Opdracht:

Schrijf een programma om een tafeltje te maken. Laat de gebruiker een getal invoeren en maak een FOR … NEXT lus waarmee de tafel van het getal wordt afgedrukt.

Opdracht:
Schrijf een programma dat het alfabet letter voor letter laat aangroeien totdat het hele alfabet compleet is. Maak gebruik van de LEFT$() functie en een FOR … NEXT lus.

Probeer het programma uit te breiden om het alfabet vanaf de laatste letter aan te laten groeien.
Tip! Gebruik de RIGHT$() functie.

Opdracht *moeilijk*:
Schrijf het programma nu zo, dat je met het complete alfabet begint en letter voor letter eraf laat gaan tot-dat de laatste letter is overgebleven. Probeer dit te doen zowel aan de linkerkant van het alfabet als aan de rechterkant van het alfabet, maar doe dit apart. Maak weer gebruik van een FOR … NEXT lus. Denk er-aan hoe je hier het STEP commando gebruiken moet.

5e. Herhalingsstructuren

Sommige BASIC dialecten en -versies kennen naast de FOR lusstructuur nog andere soort lusstructuren. Het worden **Herhalingsstructuren** genoemd. Een herhaling of lus dat iets doet zolang een voorwaarde **waar** is.

Al in BASIC 7.0 van de Commodore 128 en ook in Q(uick)BASIC bestonden de herhalingsstructuren. Deze lus-sen werken als volgt:

Herhalingsstructuur	**Omschrijving**
WHILE <voorwaarde> … WEND	Herhaal zolang <voorwaarde> **waar** is.
DO WHILE <voorwaarde> … LOOP	Doe een herhaling zolang <voorwaarde> **waar** is. Lijkt op de WHILE … WEND structuur.
DO … LOOP WHILE <voorwaarde>	Doe minimaal één keer iets en herhaal het nog eens zolang <voorwaarde> **waar** is.
DO UNTIL <voorwaarde> … LOOP	Doe een herhaling totdat <voorwaarde> **waar** is.
DO … LOOP UNTIL <voorwaarde>	Doe minimaal één keer iets en herhaal het nog eens totdat <voorwaarde> **waar** is.

Gebruik een WHILE … WEND structuur wanneer je een reeks instructies een onbepaald aantal keren wilt herha-len, zolang een voorwaarde **waar** is. Als je meer flexibiliteit wilt met waar je de conditie test of op welk resultaat je het test, geeft het je misschien de voorkeur aan de DO ... LOOP structuur. Als je de uitspraken een bepaald aan-tal keren wilt herhalen, is de FOR … NEXT statement meestal een betere keuze.

```
i = 0
WHILE i < 10
        PRINT i,
        i = i + 1
WEND
```

Op het mainwin venster zal verschijnen:

```
0       1       2       3       4       5       6       7       8       9
```

Onderstaande FOR lus doet hetzelfde:

```
FOR i = 0 TO 9 'Let op deze 9. Merk op dat er geen 10 staat!
        PRINT i,
NEXT i
```

Bedenk goed waarom de voorwaarde in de WHILE lus **i < 10** is en niet 9.

Omdat elke voorwaarde een relationele bewerking met eventueel een logische bewerking heeft, is de voor-waarde-syntax bij een keuzestructuur hetzelfde als bij een herhalingsstructuur.

Een FOR lus is de enige lusstructuur die geen voorwaarde kan hebben, maar BASIC geeft niet een fout-melding als je het toch doet.

```
for n = -10 to a <> 10
    print n
next n
```

```
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
0
1
```

Deze lus zorgt ervoor dat er van -10 tot en met 1 geteld wordt. Hiermee kan er worden bepaald of de laatste stap wel wordt uitgevoerd of niet. Als de relationele bewerking **a <> 10** waar is, wordt er een **1** afgedrukt. Is de relationele voorwaarde echter niet waar, dan zou je denken dat er twee nullen zullen worden geprint. Dat is niet het geval. Als de voorwaarde niet waar is, zal er alleen maar van -10 tot en met 0 worden afgedrukt.

Tijdens het wijzigen van de waarde van variabele a, kan de FOR lus één keer teveel tellen zoals in deze afbeelding te zien is.
Het is ook mogelijk om te bepalen of er met een 0 gestart moet worden of met een 1, door de voorwaarde achter het FOR commando te schrijven. Dit geeft een volgende conclusie:

Een FOR lus bepaalt tijdens de loop geen voorwaarde. De WHILE … WEND en de DO … LOOP statements doen dat wel.

Enkele voorbeelden volgen hieronder en probeer te begrijpen hoe de verschillende herhalingsstructuren werken.

```
getal = 0
do
    oudGetal = getal
    input "Voer een getal in: "; invoerGetal
    getal = invoerGetal + oudGetal
    helft = getal / 2
    print "Oud: "; oudGetal, "Nieuw: "; getal, "Helft: "; helft
loop until invoerGetal = 0
```

```
Voer een getal in: 6
Oud: 0         Nieuw: 6       Helft: 3
Voer een getal in: 4
Oud: 6         Nieuw: 10      Helft: 5
Voer een getal in: 0
Oud: 10        Nieuw: 10      Helft: 5
```

In de afbeelding zie je een herhaling die pas de voorwaarde als laatste controleert. Er wordt een waarde van de vorige waarde telkens erbij opgeteld. Zoals je ziet moet de invoer apart worden gebruikt, want je kunt het getal variabele niet controleren of deze nul is. Het getal wordt namelijk gewijzigd en daarom kun je niet dezelfde variabele als invoer gebruiken.

Merk ook op dat, wanneer je een nul invoert, alles toch weer berekent en afgedrukt wordt.

We hadden ook een DO WHILE … LOOP kunnen gebruiken, zoals je hieronder ziet.

```
invoerGetal = 1
DO WHILE invoerGetal <> 0
        oudGetal = invoerGetal
        INPUT "Voer een getal in: "; invoerGetal
        getal = invoerGetal + oudGetal
        helft = getal / 2
        PRINT "Oud: "; oudGetal, "Nieuw: "; getal, "Helft: "; helft
LOOP
```

Maar als je de code uitvoert, zul je merken dat er geen verschil is. Ook zorgt het wijzigen van de voorwaarde voor een ander herhalingsstructuur, waardoor de code binnen de lus aangepast moet worden.

Dat niet alleen. Ook moeten we nu beginnen met de toekenning **invoerGetal = 1** in plaats van **getal = 0**. Zouden we **getal = 0** laten staan in plaats van **invoerGetal = 1**, dan zal de lus niet uitgevoerd worden. Probeer maar eens.

Toch is er een oplossing. Door vóór de DO … LOOP eerst om een getal te vragen kunnen we de invoer van een getal en de berekeningen samen het afdrukken op het mainwin venster omdraaien, zodat met het invoeren van een 0 waarde de lus niet nogmaals herhaald wordt.

```
getal = 0
INPUT "Voer een getal in: "; invoerGetal
DO
    oudGetal = getal
    getal = invoerGetal + oudGetal
    helft = getal / 2
    PRINT "Oud: "; oudGetal, "Nieuw: "; getal, "Helft: "; helft
    INPUT "Voer een getal in: "; invoerGetal
```

```
LOOP UNTIL invoerGetal = 0
```

Je kunt ook de UNTIL <voorwaarde> achter DO schrijven. Dat verbetert het probleem nog wat meer.
Verplaats de UNTIL code achter de LOOP naar de DO, zodat er dit staat:

```
DO UNTIL invoerGetal = 0
       . . .
LOOP
```

Nu is bij de eerste invoer geen mogelijkheid meer om met een 0 in de lus te komen.
Omdat er voor de LOOP weer om invoer wordt gevraagd en er een 0 ingetoetst kan worden, zal UNTIL een WAAR resulteren en de lus wordt beëindigd.

Sommige BASIC dialecten kennen een onbepaalde DO … LOOP lus, dus zonder voorwaarde. In Liberty BASIC is dat niet toegestaan en ben je verplicht een voorwaarde op te geven.

5f. Een lus of een herhaling eerder verlaten

In de paragraaf over het GOTO commando weten we dat het niet altijd een slecht idee is om het commando te gebruiken, maar in een lus- of herhalingsstructuur werkt het commando niet naar behoren. Het kan zelfs zijn dat Liberty BASIC het programma niet uitvoert op de manier hoe een structuur, waar het GOTO commando in zit, geschreven is.

Problemen die door GOTO kunnen ontstaan of zelfs niet door Liberty BASIC worden goedgekeurd, zijn:

- In een lus naar een label springen dat er buiten staat.
  Dit veroorzaakt geen fout, maar kan leiden tot een ongestructureerde code.
- Springen naar een label die verkeerd is gespeld of niet meer bestaat.
  Liberty BASIC geeft 'helaas' het GOTO commando de schuld, maar het kan net zo goed de labelnaam zijn die niet juist is.
- Springen naar een label die buiten de scope staat.
  Dit heeft te maken met wanneer er geprobeerd wordt te springen in een lokale omgeving, zoals binnen een subroutine of functie. Eruit springen is niet toegestaan en Liberty BASIC zal dan een foutmelding geven dat de labelnaam niet gevonden kan worden.
  Meer daarover in hoofdstuk 9.

Overigens, wat we niet met het GOTO commando mogen doen, mag ook niet met een GOSUB commando. Ook daarmee gelden dezelfde regels.

Als we hiermee niet uit een lus mogen springen, hoe kunnen we dan eerder de lus verlaten?

Liberty BASIC kent een aantal EXIT commando's om verschillende soorten lussen te kunnen verlaten. Het EXIT commando kan dat niet alleen. We moeten erachter schrijven welke lus we willen verlaten.

Onderstaand voorbeeld laat zien hoe we eerder een WHILE … WEND lus verlaten met een EXIT.

```
i = 1
WHILE i < 100
    IF i = 50 THEN EXIT WHILE
    i = i + 1
    PRINT i
WEND
PRINT "Uit de lus: "; i
```

Als je echter een DO WHILE … LOOP lus met dezelfde code gaat schrijven, dan is een EXIT WHILE niet toegestaan, ook al controleren we met een WHILE in plaats van een UNTIL. Maar die twee hebben in een DO … LOOP structuur niets met het EXIT commando te maken. Je zou dan de WHILE achter EXIT moeten wijzigen in DO, zodat het IF statement wordt:

```
IF i = 50 THEN EXIT DO
```

Elke manier is toegestaan. Zeker als er meerdere wegen naar Rome zijn, zoals onderstaand voorbeeld:

```
FOR i = 1 TO 100
    IF i = 50 THEN EXIT FOR
    PRINT i
NEXT
PRINT "Uit de lus: "; i
```

Dit kan wel eens lastig zijn. Welke lus heb ik nou nodig, een gewone lus-teller (FOR … NEXT) of een lus die herhaald totdat iets waar is ([DO … LOOP UNTIL] [DO UNTIL … LOOP]) of een lus die herhaald zolang iets waar is ([WHILE … WEND] [DO … LOOP WHILE] [DO WHILE … LOOP].

Omdat de laatste (DO WHILE … LOOP) op dezelfde manier werkt als de oude BASIC herhalingsstructuur WHILE … WEND, maakt het voor deze twee niet uit welke je kiest.

Ik kan een conclusie geven:

Gebruik alleen een FOR … NEXT lus als je zeker weet dat je er niet eerder uit hoeft en geen voorwaarde hoeft te controleren in de lus. Wil je wel een voorwaarde controleren, gebruik dan een herhalingslus.

Indien je in de lus een voorwaarde wilt controleren, zoals verkeerde invoer, dan kan dat altijd. Zorg er altijd voor dat de extra keuzestructuren, die je in de lussen gebruikt, niet per ongeluk de voorwaarde van de lus zelf beïnvloedt. Er is een oplossing om deze structuren apart te houden van de lussen, maar dat je het toch in de lussen uit kunt voeren. Zie meer daarover in hoofdstuk 9.

Vroegtijdig eruit gaan met een EXIT betekent niet gelijk dat je ongestructureerd programmeert. Het is niet erg om het te doen, maar het is beter om er niet teveel gebruik van te maken.

6. Arrays en de READ en DATA gegevenscommando's

In hoofdstuk 4 'Variabelen' heb ik een kleine paragraaf met een toelichting over arrayvariabelen besproken. Arrays kunnen dienen als lijsten en zelfs als tabellen. We dimensioneren arrays en we vullen de elementen met waarden.

Het gebruik ervan kan op verschillende manieren:

- De elementen vullen met constanten.
- De elementen vullen met gegevens die we kunnen lezen uit een datareeks.
- De elementen vullen via een gegevensbestand.

Het eerste punt is inderdaad het geval dat je constanten, en natuurlijk constanten die je toekend aan variabelen, kunt toekennen aan de elementen van de arrays.

6a. Arrays gebruiken met gegevensinvoer

Arrays hebben de volgende karakteristieken:

- Je kunt ze overal dimensioneren. Het hoeft niet per sé bovenaan het programma te zijn. Handig wanneer je een array nog niet nodig hebt.
- Ze zijn *globaal*. Ze hebben geen scope (zie volgend hoofdstuk over subroutines en functies).
- Ze zijn *statisch*, dat wil zeggen: ze hebben een vast aantal elementen. Na het dimensioneren is het niet meer mogelijk meer elementen eraan toe te voegen.
- Arrays zijn herdimensioneerbaar. Als dat wordt toegepast, gaat de inhoud verloren.
- Een arrayvariabele is **niet** dezelfde als een gewone variabele. Dit betekent dat een arrayvariabele naam en dezelfde gewone variabele naam door Liberty BASIC niet als dezelfde variabelen wordt gezien. Het programma hieronder laat dat zien.

Typ eens onderstaand programma in:

```
DIM Leeftijd(10)
DIM Naam$(10)

I = 1
DO
        INPUT "Typ je naam in: "; Naam$             'Variabele Naam$ is niet de array Naam$()
        INPUT "Typ je leeftijd in: "; Leeftijd      'Dit is ook het geval bij Leeftijd
        Naam$(I) = Naam$
        Leeftijd(I) = Leeftijd
        I = I + 1
LOOP UNTIL I > 10 OR Naam$ = ""
```

Nu kun je door de array itereren *(=bladeren)*. Omdat je weet hoeveel elementen je hebt, kun je een FOR … NEXT lus gebruiken.

```
FOR element = 1 TO 10
        PRINT Naam$(element); " is "; Leeftijd(element); " jaar oud."
NEXT element
```

Stel dat je tijdens het invoeren eerder een lege string hebt ingevoerd dan het aantal elementen dat de array heeft. Voorbeeld: Je voert vier namen en leeftijden in en de vijfde naam laat je leeg. De DO … LOOP lus stopt ermee en later wordt de FOR … NEXT lus uitgevoerd.
Als je de uitvoer ziet, dan zul je gewoon vier namen en vier leeftijden zien, maar print maar eens wat onder het NEXT commando. Het resultaat zullen veel lege regels zijn, namelijk zes lege regels.

De oplossing is om een WHILE lus te gebruiken die controleert zolang het element van de Naam$() array niet leeg is, zie voorbeeld.

```
element = 1
WHILE element <= 10 AND Naam$(element) <> ""
        PRINT Naam$(element); " is "; Leeftijd(element); " jaar oud."
        element = element + 1
WEND
```

6b. Arrays gebruiken met READ en DATA

In plaats van het invoeren van gegevens en toekennen aan een array, kunnen we ook gegevens inlezen en toekennen aan een array.
Het inlezen zou je denken aan een gegevensbestand, maar dat bespreek ik in hoofdstuk 10.

BASIC kent twee oude commando's die al in de microcomputertijd bestonden. Met de oude BASIC dialecten kon al met gegevens gewerkt worden, met READ en DATA.

De syntax van het READ commando is:

READ <var1>[, <var2>, var[3], … , <varn>]

Het commando heeft één of meer variabelen nodig, zodat de gegevens uit het DATA commando gelezen kunnen worden.

De syntax van het DATA commando is:

DATA <n1>[, <n2>, … <nn>][, <s1>[, <s2>, … , <sn>]][, "<s1>"[, "<s2>", … , "<sn">]]

Misschien geeft deze syntax nog geen goede duidelijkheid, maar de voorbeelden zullen verklaren hoe het werkt.

Het READ commando heeft in Liberty BASIC één beperking. In andere BASIC dialecten mogen we een array element als variabele gebruiken om in te lezen. In Liberty BASIC kan dat niet.

De gegeven variabelen achter READ moeten gescheiden worden met komma's. Maar hoe werkt het precies? Onderstaand voorbeeld laat zien hoe het werkt. Voer het uit en zie het resultaat op het mainwin venster.

```
READ a, b
PRINT a, b

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Zoals we het resultaat zien, wordt er van links naar rechts gelezen. Je zou denken dat we dan tien variabelen achter het READ commando nodig hebben om alles te kunnen lezen.

Dankzij het gebruik van een array, kunnen we alle gegevens inlezen en toekennen aan de array elementen. Het gebruik van een lusstructuur is het beste om dat te doen.

```
DIM num(9)

FOR i = 0 TO 9
        READ n
        num(i) = n
NEXT i
FOR i = 0 TO 9
        PRINT num(i)
NEXT i
DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

We kunnen ook tekst inlezen. In de DATA regel kan dat op twee manieren.

```
DIM naam$(4)
```

```
FOR i = 0 TO 4
        READ s$
        naam$(i) = s$
NEXT i
FOR i = 0 TO 4
        PRINT naam$(i)
NEXT i
DATA Marie, Jan, "Peter", "Huis aan huis", "Brievenbus"
```

Voer het programma uit en zie het resultaat.

We hoeven in een DATA regel niet per sé aanhalingstekens te gebruiken met tekst, maar met uitzondering als er spaties in de tekst voorkomen. We hadden dus de aanhalingstekens bij de tekst “Peter” en bij de tekst “Brieven-bus” weg kunnen laten, maar bij de tekst “Huis aan huis” niet.

Zoals we eerder zagen kun je meer variabelen achter READ gebruiken. Houd er rekening mee dat je de DATA regels zo structureert, dat de juiste gegevens bij de juiste variabelen ingelezen worden.

Het eerste voorbeeld gebruikt twee variabelen, a en b.
Bijvoorbeeld hieronder met het eerste voorbeeld in een lus.

```
FOR i = 1 TO 5
        READ a, b
        PRINT a, b
NEXT i
DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Het resultaat zal zijn:

1	2
3	4
5	6
7	8
9	10

De oneven waarden zullen alleen in variabele a gelezen worden.
De even waarden zullen alleen in variabele b gelezen worden.

Merk op dat je de FOR lus op de juiste manier schrijft. Hoe meer variabelen je gebruikt om in te lezen, hoe kleiner het maximum moet zijn die je in de FOR lus opgeeft. Je ziet dus voor 10 getallen dat je een lus nodig hebt die maar tot 5 mag tellen.

Numerieke waarden en strings mogen ook gemengd in een DATA regel staan.

```
FOR i = 1 TO 5
        READ n$, j
        PRINT n$; " is "; j; " jaar oud."
NEXT i
DATA Marie, 34, Jan, 40, Piet, 45, Klaas, 50, Cynthia, 28
```

Denk eraan dat alleen aanhalingstekens nodig zijn wanneer spaties in de strings staan. Je mag natuurlijk altijd aanhalingstekens gebruiken, dus ook bij deze DATA regel. Altijd aanhalingstekens bij strings geeft zeker de voorkeur omdat daarmee direct het verschil is te zien waar de strings staan en waar de getallen staan. Maar wat je zelf het beste vindt.

Om bovenstaande gegevens in een array te vullen, moeten we uit de DATA regel weer bepalen hoeveel elementen we nodig hebben. Soms is dat niet fijn en zeker niet als je wel 20 DATA regels hebt.

Een oplossing is om eerst de DATA regels in te lezen en tegelijkertijd een teller te gebruiken, die daarna bepaald hoeveel elementen we nodig hebben. We vullen dus niet direct de array, maar doen dat als we voor de tweede keer dezelfde gegevens inlezen.

Het voordeel is dat we de array kunnen dimensioneren met de teller, zodat we altijd de array groter of kleiner kunnen maken wanneer we de DATA regels aanpassen.

Een probleem is het opnieuw lezen van de DATA regels. Deze zouden we dan moeten kopiëren en eronder moeten zetten.
Gelukkig heeft Liberty BASIC daar een oplossing voor.

6c. Gegevens opnieuw lezen met gebruik van het commando RESTORE

De vorige code met een FOR lus die van 1 tot 5 loopt is niet altijd handig. Vooral niet als we niet zeker zijn of we later het aantal gegevens in de DATA willen wijzigen. Bovendien is het ook niet fijn als we moeten rekenen tot hoever de lusteller moet lopen als we meer variabelen achter het READ commando gebruiken.

Ook voor één variabele achter het READ commando, moet er een oplossing zijn om tijdens het lezen van de gegevens te laten bepalen tot wanneer het lezen moet stoppen. Een extra teller variabele kan dan worden gebruikt om de array te dimensioneren.

De code ziet er dan zo uit:

```
i = 0
naam$ = ""
WHILE naam$ <> "end"
        READ naam$
        i = i + 1
WEND
DIM Gegevens$(i - 1, 1)
RESTORE
i = 0
naam$ = ""
WHILE naam$ <> "end"
        READ naam$, jaar
        IF naam$ <> "end" THEN
                Gegevens$(i, 0) = naam$
                Gegevens$(i, 1) = STR$(jaar)
                i = i + 1
        END IF
WEND
aantal = i
FOR i = 0 TO aantal
        PRINT Gegevens$(i, 0), Gegevens$(i, 1)
NEXT i
DATA Marie, 34, Jan, 40, Piet, 45, Klaas, 50, Cynthia, 28
DATA Karel, 22, Mies, 38, Zoë, 18, Marco, 50, "end", 0
```

Het is natuurlijk de vraag waarom ook een 0 als laatste waarde in de tweede DATA regel staat. Ook al wordt daar niet op voorwaarde getest, maar op het woord “end”. De reden is dat er twee gegevens tegelijk gelezen worden. Als we de 0 weglaten, dan zou READ een waarde missen en geeft Liberty BASIC een foutmelding.

Als we speciaal willen dat alleen het woord “end” het lezen moet afsluiten, dan moet bovenstaande code anders aangepakt worden. Het lezen doen we dan met één variabele. Onderstaande code laat de wijziging zien.

```
i = 0
naam$ = ""
WHILE naam$ <> "end"
    READ naam$
    i = i + 1
WEND
DIM Gegevens$(i - 1, 1)
RESTORE
i = 0
j = 0
naam$ = ""
WHILE naam$ <> "end"
    READ naam$
    IF naam$ <> "end" THEN
        IF j > 1 THEN j = 0
        Gegevens$(i, j) = naam$
        j = j + 1
        i = i + 1
    END IF
WEND
aantal = i
FOR i = 0 TO aantal
    PRINT Gegevens$(i, 0), Gegevens$(i, 1)
NEXT i
```

```
DATA Marie, 34, Jan, 40, Piet, 45, Klaas, 50, Cynthia, 28
DATA Karel, 22, Mies, 38, Zoë, 18, Marco, 50, "end"
```

Voer het programma uit en kijk goed naar het mainwin venster.
Het PRINT commando drukt elke twee gegevens gescheiden met een komma af. Je zou denken dat het in twee kolommen zou verschijnen, maar dat gebeurd echter niet.

Het probleem is een addertje onder het gras. Het is even kijken naar het programma waar het aan kan liggen en als je het tellen van je array elementen tijdens het lezen van de DATA begrijpt, dan zie je misschien wat er fout gaat.

Elke naam en jaar moeten in de tweede index komen van de array. De eerste index bepaalt alleen maar per paar hoeveel er zijn. Dat verklaart waarom de eerste WHILE … WEND teveel elementen telt, omdat er niet per paar gelezen wordt.

Ook is het goed kijken in de tweede WHILE … WEND wat daar fout gaat. Het heeft te maken met het IF statement. Hoewel het IF statement juist is en niet het probleem is, want uiteraard moet na elk paar de variabele j weer op nul worden gezet, moet toch het IF statement aangepast worden. Het probleem is namelijk weer de teller variabele i die teveel telt en eigenlijk per paar moet tellen.
De oplossing is door de teller op te nemen in het IF statement. Telkens wanneer variabele j weer nul is, zal de teller met één worden verhoogd, zodat het per paar nu wel goed zal gaan.

Onderstaande code geeft de oplossing. Probeer het verschil met het vorige voorbeeld te begrijpen.

```
i = 0
naam$ = ""
WHILE naam$ <> "end"
    READ naam$
    IF naam$ <> "end" THEN READ jaar
    i = i + 1
WEND
DIM Gegevens$(i - 1, 1)
RESTORE
i = 0
j = 0
naam$ = ""
WHILE naam$ <> "end"
    READ naam$
    IF naam$ <> "end" THEN
        IF j > 1 THEN
            j = 0
            i = i + 1
        END IF
        Gegevens$(i, j) = naam$
        j = j + 1
    END IF
WEND
aantal = i
FOR i = 0 TO aantal
    PRINT Gegevens$(i, 0), Gegevens$(i, 1)
NEXT i
DATA Marie, 34, Jan, 40, Piet, 45, Klaas, 50, Cynthia, 28
DATA Karel, 22, Mies, 38, Zoë, 18, Marco, 50, "end"
```

Opdracht:
Breid bovenstaand programma uit door ook achternamen te laten lezen en te printen. Denk aan het IF statement in de eerste WHILE … WEND code en denk eraan dat je de array aanpast en de voorwaarde 'j > 1' in het IF statement aanpast.

Opdracht:
Heb je de vorige opdracht kunnen doen? Probeer dan eens meer elementen te gebruiken, zoals het adres en de woonplaats. Pas de variabele j goed aan, zodat de voorwaarde overeenkomt met het aantal elementen van de tweede array index.

Met RESTORE kunnen we dus de DATA regels opnieuw lezen. Het is ook mogelijk om een deel van alle gegevens opnieuw te lezen. Het RESTORE commando kan een labelnaam hebben dat ook bij de juiste DATA regels

moet komen te staan. Het werkt op dezelfde manier als bij een GOSUB en een GOTO. Alleen het RESTORE commando is een pointer voor het READ commando en kan de structuur voor het lezen van de DATA regels bepalen.

In de Help van Liberty BASIC kun je een voorbeeld vinden over RESTORE met een labelnaam.

7. Gestructureerd programmeren

Eerder in het boek kwamen we verschillende vertakkingen tegen. Deze vertakkingen, structuren genoemd, werden vroeger geprogrammeerd met regelnummers. De commando's GOTO en GOSUB met een RETURN waren de hoofdzakelijke vertakkingen om een structuur in een programma aan te kunnen brengen.

Het gevaar was dat, met een slechte oplettendheid, al snel een programma ongestructureerd geprogrammeerd werd. Zoals al eerder uitgelegd, moet met een GOTO en een GOSUB zorgvuldig mee worden omgegaan. Wie liever de nieuwe structuren met subroutines en functies gebruiken wil, kan dat best en zal een GOTO en een GOSUB niet nodig zijn.

In Deel 3 van het boek zul je echter zien dat het niet altijd kan om zonder een GOTO of GOSUB te programmeren. Er zijn situaties dat ze nodig zijn.

7a. Globaal programmeren met labels

Wie de oude BASIC kent, weet dat het een programmeertaal was dat globaal werkte. Die oude manier om te programmeren kan nog steeds. Liberty BASIC ondersteunt ook die oude manier.

Sinds de tijd van QuickBASIC is de programmeertaal structureel veranderd. Al heel wat jaren kunnen we lokaal programmeren, door elke blokcode lokaal uit te voeren. Er ontstaan verplichtingen, waar programmeurs, die de oude BASIC dialecten kennen, er flink aan moeten wennen.

Hoe komt het dat men er veel aan moet wennen? Wat is het verschil en wat wil lokaal programmeren eigenlijk zeggen?

Voordat ik het daarmee verder over ga hebben, zullen we eens zien hoe we globaal kunnen programmeren in Liberty BASIC.

Als we globaal programmeren dan programmeren we met *labels*, zoals ik het eerder in het boek al over gehad heb. Nogmaals, verwar het niet met een branch, zoals je in de Help of in voorbeelden op Internet tegenkomt. Al eerder heb ik verteld dat een label niet voor de branch (vertakking) zorgt, maar alleen voor zorgt dat waar een label staat daarheen gesprongen kan worden.

Om te weten te komen hoe een label werkt, is er een onderstaand tabel dat aangeeft waar ze voor nodig zijn, hoe ze werken en wat voor voordelen en nadelen ze hebben.

Labels zijn regelnummers of bestaan uit namen.	Regelnummer-labels kunnen zich als echte regelnummers gedragen. Liberty BASIC onderhoudt geen volgorde en per ongeluk dezelfde nummers zijn niet toegestaan. Deze worden ook niet overschreven. Naam-labels kun je net zo noemen zoals je ook de variabelennamen noemt. Een naam-label mag beginnen met één of meer cijfers. Het enige verschil met regelnummer-labels is dat de naam-labels tussen rechte haken moeten staan.[1]
Labels hebben geen scope.	Een scope is een deel van de code dat alleen herkenbaar is als dat deel *aangeroepen* wordt. Dit noemen we een *lokale* aanroep. De labels worden niet lokaal aangeroepen. Er wordt naar een naam-label *gesprongen*. Dit kan op twee manieren: met een sprong als één richting of als een functiesprong, waarvan de plaats waar gesprongen werd onthouden wordt en

	daar weer terug kan gaan zonder zelf dat te hoeven doen. Doordat er gesprongen wordt, functioneel of niet, blijft alles bestaan. Variabelen zijn dan ook overal herkenbaar en hebben geen scope. Dit noemen we globaal programmeren. Dit is één van de voordelen met het gebruik van labels. Omdat die manier ook gemakkelijker is, wordt het nog steeds veel gebruikt.
Oppassen bij veel gebruik van labels.	Omdat globaal programmeren het gemakkelijk maakt – want je hoeft nergens op te letten of een variabele nog zijn waarde heeft – kan toch bij teveel gebruik van labels problemen ontstaan. Labels kunnen namelijk in het hele programma gebruikt worden. Het programmastroomschema kan hierdoor in de war raken en alles kan door elkaar heen uitgevoerd worden.
De labels ‘zien’ elkaar niet!	Wanneer er naar een label gesprongen wordt, dan is er geen einde in zicht. Elke volgende label kan uitgevoerd worden, ook al zou je dat niet willen. Het gebruik van vlaggen, dat speciale variabelen zijn die alleen een 0 of een 1 krijgen, zijn noodzakelijk om te voorkomen dat de volgende label met de codeblok die erachter komt, uitgevoerd wordt. De volgende label wordt niet gezien. Het lijkt dan alsof deze er niet staat.

Eerder in het boek zagen we dat we met een GOTO commando ergens naartoe kunnen springen en een codeblok nogmaals uit kunnen voeren. Vaak is het beter om van te voren in te plannen welke code nogmaals uitgevoerd moet worden.

Terug springen
Als je terug wil springen, zorg er dan voor dat je in de code aan het volgende denkt:

- Stel het blok met de code veilig. Dat wil zeggen, laat het blok alleen uitvoeren wanneer het mogelijk is. Zorg ervoor dat het blok een *einde* heeft om te voorkomen dat het volgende blok ook uitgevoerd wordt, bijvoorbeeld waar zo’n GOTO commando in staat. Anders is het gevolg dat je een oneindige uitvoering in je programma krijgt.
- Het veilig stellen kun je doen met een vlag variabele. Eerst geef je de vlag een nul. Je bepaalt met de vlag of de volgende code uitgevoerd mag worden. Als de vlag nul is, dan heeft het nog niet het GOTO commando bereikt. Voordat je met een GOTO terug springt, zet je de vlag op 1 en zorg je ervoor dat het blok, waar het GOTO commando in staat, overgeslagen wordt.

Vooruit springen
Als je vooruit wil springen, dan heb je redenen nodig waarom. De volgende punten geven een voorbeeld wat je kunt doen met vooruit springen.

- Een voorwaarde in een blok is niet juist. Je wilt een volgende deel overslaan en terug gaan om een stuk code nog eens uit te voeren. Meestal wordt zoiets gedaan in een lus. Er wordt dan gesprongen naar een NEXT commando, als de rest van de code in een lus niet uitgevoerd mag worden.
  Natuurlijk kan het ook zonder een GOTO commando, maar dan moet het resultaat van de voorwaarde andersom zijn, namelijk wanneer het wel juist is. Het blok heb je dan tussen THEN en END IF. Is de voorwaarde niet juist, dan wordt gewoon de code na de END IF uitgevoerd en dat kan bijvoorbeeld een NEXT commando zijn.
- Wil je vooruit springen, maar daarna weer terug gaan, dan is het beste om een GOSUB en een RETURN te gebruiken. Een GOSUB voert altijd een globale subroutine uit binnen het programma en keert weer terug met een RETURN.
- Stel het codeblok ook weer veilig. Zorg ervoor dat je niet per ongeluk ergens met een GOTO naartoe springt, waar toevallig de code bedoeld is als een subroutine. Gevolg: Liberty BASIC komt een RETURN tegen zonder een GOSUB en een foutmelding verschijnt.

Wanneer je met labels werkt, dan is een naam-label niet altijd voldoende om te kunnen vertellen wat het doel is van het codeblok. Eerst commentaar geven, over wat de bedoeling is van de naam-label, geeft meer duidelijkheid en de kans op per ongeluk verkeerd uitvoeren wordt dan kleiner.

Hieronder het eerste voorbeeld.

```
' twee keer uitvoeren met een GOTO
```

```
vlag = 0
a$ = "Eerste keer"
[tweede]
    PRINT a$
    IF vlag = 1 THEN GOTO [einde]
a$ = "Tweede keer"
vlag = 1
GOTO [tweede]
[einde]
```

Als je het uitvoert, dan krijg je de tekst *Eerste keer* en de tekst *Tweede keer* te zien.
Dankzij de controle op de vlag, kunnen we dezelfde PRINT regel twee keer uitvoeren.

Het tweede voorbeeld laat het zien met een GOSUB en RETURN subroutine.

```
' twee keer uitvoeren met een GOSUB ... RETURN
vlag = 0
a$ = "Eerste keer"
[lus]
    PRINT a$
    IF vlag = 1 THEN GOTO [einde]
    GOSUB [tweede]
    GOTO [lus]
[einde]
    END
[tweede]
    a$ = "Tweede keer"
    vlag = 1
    RETURN
```

Voer je die code uit, dan krijg je dezelfde uitvoer, ook al is het met een andere structuur.
De code ziet er wat rommeliger uit dan het eerste voorbeeld, maar dat komt omdat we na de GOSUB terug willen naar de label [lus].

De naam zegt het al: lus.
Kunnen we dan niet in plaats daarvan een echte lus gebruiken?

Voer eens onderstaande code uit.

```
vlag = 0
a$ = "Eerste keer"
WHILE vlag = 0
    PRINT a$
    GOSUB [tweede]
WEND
END ' of een GOTO naar een ander codeblok in het programma
[tweede]
    a$ = "Tweede keer"
    vlag = 1
    RETURN
```

Je zal zien dat de WHILE lus niet voor een goede oplossing zorgt. Dit komt doordat de vlag op 1 wordt gezet in de subroutine. Dat wordt ook in het tweede voorbeeld gedaan, maar het probleem is nu dat de voorwaarde in de WHILE niet meer juist is waardoor de tweede tekst nooit geprint wordt.

Een ander belangrijk commando is het END commando. Zouden we in de tweede en derde voorbeelden het END commando weglaten, dan zal pardoes voor de tweede keer de code na [tweede] uitgevoerd worden, waardoor er weer een RETURN wordt tegengekomen die niet zonder een GOSUB uitgevoerd kan worden. Gevolg: een foutmelding verschijnt.

Labels hebben een voordeel. Ze kunnen lokaal worden in aanroepende codeblokken, zoals verder in dit hoofdstuk uitgelegd wordt.

Opdracht:
Er is een oplossing om toch de WHILE lus te kunnen gebruiken. Probeer dezelfde code te handhaven, maar dan zo dat je toch beide teksten kunt printen. Let ook op de vlag variabele. De kans op een oneindige uitvoer is groot als je niet de vlag variabele aanpast.

Opdracht:
Schrijf een soortgelijk programma als het laatste voorbeeld, maar dan nu met meer teksten, zoals onderstaande uitvoer:

1 keer
2-de keer
3-de keer
…

Gebruik een INPUT commando om het aantal keren dat er geprint moet worden te bepalen.

Opdracht extraatje:
Bepaal met IF statements wanneer je 'ste' moet printen en wanneer 'de'. Hint: gebruik de MOD functie met een deling van 10 als er meer dan 10 getallen geprint worden. Het getal 20 is bijvoorbeeld met 'ste'. Conroleer de 1 en de 8 apart die ook 'ste' hebben.

Uit de laatste twee opdrachten zul je merken dat we de vlag variabele nu als teller beschouwen. Het is daarom verstandig om teller variabelen en vlag variabelen uit elkaar te houden. Vlaggen worden alleen gecontroleerd wanneer deze uit (0) staan of aan (1) staan.

7b. Subroutines en functies

De labels en de branches daarvan (GOTO of GOSUB) hebben een voordeel. De codeblokken zijn overal toegankelijk en alle gegevens blijven aanwezig.
Totdat er een probleem ontstaat:

- Er kunnen per ongeluk dezelfde variabelen in meer codeblokken worden gebruikt. Liberty BASIC waarschuwt ons niet en de oude gegevens worden dan overschreven dat tot verkeerde uitvoer kan leiden. Labels zijn overal te gebruiken. Liberty BASIC waarschuwt ons niet wanneer een label blok al in gebruik is en het missen van een GOSUB wanneer Liberty BASIC een RETURN tegenkomt kan daardoor regelmatig gebeuren.
- Er is geen bescherming in een codeblok. Je bent vrij om ook andere blokken uit te voeren zonder eerst te controleren of dat wel de bedoeling is, zoals in de vorige paragraaf uitgelegd is. Het gebruik van vlaggen is een optie, maar niet noodzakelijk. Jouw eigen oplossing is het belangrijkste.

Er zijn mogelijkheden in Liberty BASIC om deze problemen te vermijden. Codeblokken kunnen beschermt worden door deze lokaal aan te roepen. Al in eerdere BASIC programmeertalen is deze structuur toegepast.

Door gebruik te maken van zelf geschreven subroutines en functies, kunnen we elk codeblok een eigen karakter geven, zonder dat andere code er wat van afweet.

De subroutines en functies hebben de volgende syntax:

Sub Subroutinenaam param1, param2, …, paramn … End Sub
Function Functienaam(param1, param2, …, paramn) … Functienaam = resultaat End Function

Het aanroepen gaat op deze manier:

CALL Subroutinenaam arg1, arg2, …, argn
|variabele =| |commando| Functienaam(arg1, arg2, …, argn)

Subroutines moeten altijd met een CALL commando aangeroepen worden. Functies moeten zonder CALL aangeroepen worden.
Functies worden gebruikt in een toekenning of achter een commando, zoals het PRINT commando. Functies kunnen ook andere functies aanroepen, mits zij als argumenten meegegeven worden. Zij kunnen ook als argument worden aangeroepen in een aanroepende subroutine. Daarentegen is het niet toegestaan om een subroutine als argument aan te roepen.

Een voorbeeld is een Test subroutine.

```
SUB Test
        PRINT "Dit is een test."
END SUB
```

Als je dit uitvoert, dan gebeurt er niets.
De subroutine moet eerst aangeroepen worden om de code in de subroutine uit te laten voeren.
Voor labels geldt er geen uitvoervolgorde. Dat komt omdat er geen grens bestaat voor en na een label definitie en er zonder er naartoe te springen het uitgevoerd kan worden.

Bij subroutines en functies geldt er wel een volgorde en wel als volgt:

Subroutines en functies kunnen alleen voor de definitie aangeroepen worden. Niet erna.

Deze volgorde geldt alleen tussen het hoofdprogramma en de sub- en functiedefinities. Er is geen volgorde bij het aanroepen vanuit de sub- en functiedefinities zelf. Ze kunnen overal aangeroepen worden, omdat de definities dan al bekend zijn.

Bovenstaande subroutine Test kan aangeroepen worden en dat wordt gedaan met het CALL commando.

```
CALL Test
SUB Test
        PRINT "Dit is een test."
END SUB
```

Zoals eerder gezegd kan de aanroep en de subroutine definitie niet andersom staan. Probeer maar eens de aanroep na de subroutine te plaatsen en voer het eens uit. Je zult zien dat er niets uitgevoerd wordt.

Een moeilijkheid om subroutines en functies te begrijpen is het aanroepen. Snel de neiging hebben om met een GOTO of een GOSUB ergens heen te springen is er niet bij. In deze programmeerstructuur geldt er een wet:
Alles wat er in een sub- of functiedefinitie staat is, ten opzichte van de rest van het programma, lokaal.

Probeer eens onderstaand programma.

```
CALL Test1
SUB Test1
        i = 50
        PRINT i
        CALL Test2
END SUB
SUB Test2
        i = i + 50
        PRINT i
END SUB
```

Als je het uitvoert, zul je zien dat de waarde van variabele i niet bij de tweede PRINT het getal 100, maar het getal 50 geprint wordt. De variabele i in de sub Test2 is een hele andere variabele dan die in de sub Test1.

Bij gebruik van functies geldt hetzelfde.

```
CALL Test
SUB Test
        i = 50
        i = FnTest()
        PRINT i
END SUB
FUNCTION FnTest()
        i = i + 50
        FnTest = i
END FUNCTION
```

Als je dit uitvoert, zal het getal 50 geprint worden, omdat variabele i in de functie niet dezelfde variabele i is als in de sub Test.

Wie niet gewend is om met subroutines en functies te programmeren, zal dit verwarrend en lastig vinden. De variabelen zijn in de subroutines en functies *lokaal*, waardoor ze niets van elkaar afweten.

Er is een mogelijkheid om in de functie toch het getal 50 te laten onthouden en een ander getal 50 erbij op te tellen, om dan vervolgens het resultaat terug te kunnen geven aan de variabele in de subroutine.

7c. Argumenten en parameters

Wat we met labels niet kunnen doen is waarden meegeven zodat we alleen de meegegeven waarden gebruiken, zonder ons te moeten storen aan andere waarden. We willen bijvoorbeeld niet weten hoeveel spijkers er in een fruitmand liggen. Omdat we met labels globaal programmeren, komen we dat wel te weten, want de waarden bestaan nog steeds.

Alles in de subroutines en functies zijn lokaal en komen we niets te weten over de waarden uit andere subroutines en functies. Zelfs niet als dezelfde variabelennamen gebruikt worden, zoals we in de vorige paragraaf zagen met de twee i variabelen.

De enige manier om bijvoorbeeld een appel door te kunnen geven aan de fruitmand, is door deze mee te geven in de aanroep van een subroutine.

Onderstaand programma laat zien hoe we een mand kunnen vullen met producten. In de FOR lus zien we dat we twee variabelen in de CALL nodig hebben. De ingevoerde naam en de lus-teller.

```
DIM mand$(5)

FOR i = 1 TO 5
        INPUT naam$
        INPUT i
        CALL Mandje naam$, i
NEXT i

SUB Mandje naam$, i
        mand$(i) = naam$
END SUB
```

De variabelen naam$ en i worden in de aanroep *argumenten* genoemd. In de subroutine zijn het *parameters*. Hoewel we denken dat de variabelen worden doorgestuurd naar de subroutine, is dat echter niet het geval. Alleen de waarden van de variabelen worden doorgestuurd. Dit betekent dat de parametervariabelen niets van de argumentvariabelen afweten. De parameters zijn lokaal, net zoals we een variabele in de subroutine zouden initialiseren en gebruiken. Na de aanroep bestaan de parameters niet meer, maar de argumenten kun je na de aanroep gewoon blijven gebruiken.

Omdat de variabelen zelf niet doorgestuurd worden, maar de waarden, hoef je ook niet speciaal als parameters dezelfde variabelennamen te nemen. Dit had ook gekund:

```
SUB Mandje product$, element
```

De waarde van argument naam$ wordt door parameter product$ genomen en kan in de subroutine worden gebruikt. Dit geldt net zo voor argument i waarvan de waarde door parameter element genomen wordt.

Na de uitvoer van de subroutine, dus na de aanroep, gaan de waarden van de parameters weer verloren. Dit noemen we *by value* genoemd als bij waarde. Een andere mogelijkheid om de waarden die in de parameters veranderen te behouden, is om een referentie te maken.

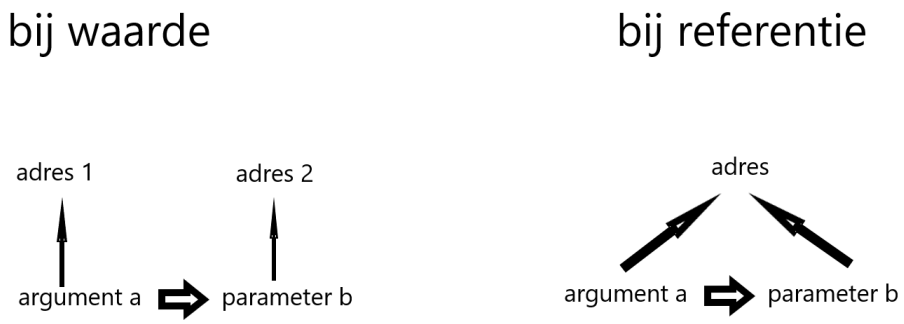
7d. Bij waarde en bij referentie (byval en byref)

Onderstaand voorbeeld laat zien hoe de variabele a zijn waarde niet meer behoud, omdat de wijziging bij referentie wordt teruggegeven.

```
a = 20
CALL Test a
PRINT a

SUB Test BYREF b
    b = 10
END SUB
```

Als je het programma start, zie je het getal 10 verschijnen en niet het getal 20. Door de parameter te beginnen met BYREF, zal parameter b het adres van argument a krijgen, zodat bij wijziging van de waarde ook het gewijzigd zal worden van argument a.

Een voorbeeld:



Of een parameter bij waarde is of bij referentie, lokaal blijft lokaal. De waarde gaat nu niet verloren, maar de parametervariabele zal niet blijven bestaan.

In andere BASIC dialecten zoals Visual Basic en Q(uick)BASIC, worden beide sleutelwoorden BYVAL en BYREF gebruikt. Het sleutelwoord BYVAL is optioneel. Standaard zijn de parameters altijd bij waarde. Liberty BASIC kent niet het sleutelwoord BYVAL en zijn de parameters altijd standaard bij waarde.

Wat gebeurt er als we zeggen: CALL Test 2. Als de parameter BYREF is, dan zal het argument de eventuele wijziging terugkrijgen, maar het argument is een getal (een constante) en geen variabele. We weten dat we kunnen zeggen ‘a = b’, maar ‘2 = b’?

Andere programmeertalen geven een foutmelding als een argumentconstante wordt gegeven terwijl de parameter bij referentie is. Er zijn ook BASIC dialecten die een fout geven.
Liberty BASIC vermijdt echter de fout en doet net alsof er geen argument gegeven werd. Probeer maar eens argument a te wijzigen in een constante en start het programma. Je zult zien dat variabele a, na de aanroep, dezelfde waarde behoudt en er verder niets gebeurd.

Dat Liberty BASIC niet reageert op een per ongeluk gegeven constante, kan vervelend zijn. Liever zouden we willen dat er verwezen wordt naar het argument of de parameter die de fout veroorzaakt, zodat we zelf het kunnen oplossen, door of de parameter te wijzigen in een bij waarde (dus BYREF weghalen) of door het argument te wijzigen in een variabele.

7e. Subroutines en functies aanroepen in de definities

We zagen al in een voorbeeld een functie die in een SUB definitie aangeroepen wordt. We mogen meerdere keren aanroepen in een definitie. Ook een subroutine aanroepen in een FUNCTION definitie is mogelijk.

Ook de argumenten die je in een subroutine of functie in een definitie meegeeft worden lokaal als parameters als ze bij waarde zijn (zie bovenstaande afbeelding). Alles geldt hetzelfde als wanneer je buiten een subroutine- of functiedefinitie aanroept.

Het lastige is om te weten wanneer je iets in een definitie wilt aanroepen en wanneer erbuiten. Eén ding is belangrijk om te weten.

Een subroutine of functie buiten de definitie aanroepen betekent dat je ze aanroept in het hoofdprogramma.

Zoals eerder uitgelegd, roep je een subroutine of functie altijd boven de definities aan. Niet eronder en ook niet ertussen, want die worden nooit door Liberty BASIC gezien.

```
INPUT "Waarde 1: "; waarde1$
INPUT "Waarde 2: "; waarde2$

CALL DeWaarden waarde1$, waarde2$
END     'commando END is niet nodig. De onderstaande definities is geen uitvoercode dus het
        'programma wordt beëindigd.
SUB DeWaarden w1$, w2$
```

```
        PRINT w1$, w2$
        CALL Waarde1 w1$
        PRINT w1$
        w2$ = GeefWaarde(w1$)
        PRINT w2$
        PRINT GeefWaarde(w1$)
        PRINT w1$
        CALL Waarde2 w2$
        PRINT w2$
        PRINT GeefWaarde("Hallo")
        PRINT GeefWaarde(w1$ + w2$)
END SUB
'Volgende twee regels worden niet uitgevoerd.
A = 10
PRINT A
SUB Waarde1 s$
        s$ = "Waarde veranderd."
        PRINT s$
        PRINT "Waarde van w1$ is bij waarde en wordt niet gewijzigd."
END SUB
SUB Waarde2 BYREF s$
        s$ = "Waarde veranderd."
        PRINT s$
        PRINT "Waarde van w2$ is bij referentie en wordt wel gewijzigd."
END SUB
FUNCTION GeefWaarde(s$)
        GeefWaarde = UPPER$(s$)
END FUNCTION
```

Dat de twee regels A = 10 en PRINT A niet uitgevoerd worden, komt niet door het commando END. Ook zonder commando END zullen de twee regels niet worden gezien. De vraag of het wel gecompileerd wordt, kun je zelf beantwoorden door een ongeldige expressie te printen, als voorbeeld A / 0.

Je kunt nooit je programma starten als er een ongeldig statement staat, ook al zal het tussen de subroutine- of functiedefinities staan. Zulke tekst dat door Liberty BASIC niet ondersteund wordt, wordt dus direct opgemerkt.

7f. Doelen

Subroutines en functies geven de mogelijkheid je programma te verdelen in blokken. In principe kan dat ook met labels, maar je programma wordt daar niet vriendelijker op.

Subroutines maak je om verschillende redenen:

- Het hoofdprogramma klein te houden.
- Elk doel apart te schrijven, zoals onderdelen van een auto. Een auto maak je niet in één keer.
- Een groot doel ook weer te verdelen in kleinere doelen, zoals bovenstaand voorbeeld om aan te roepen in een definitie. Dit kan voordelen hebben wanneer je een kleinere doel toch ook in een ander doel moet gebruiken. Zo kun je de functie GeefWaarde() ook ergens anders in het programma aanroepen en hoeft het dus niet alleen maar in één subroutine. In het hoofdprogramma mag de functie ook aangeroepen worden.

Functies maak je om verschillende redenen:

- Als je weet of in de definitie een waarde teruggegeven moet worden. Het gebruik van een BYREF in een subroutine is niet altijd een goed idee.
- Als een eenregelige expressie niet voldoende is. Je wilt bijvoorbeeld eerst andere expressies berekenen en al die berekeningen als één resultaat teruggeven.
- Gebruik geen functie wanneer je alleen maar de code wilt verdelen wanneer je denkt dat het te groot wordt. Onthoud wel: *blokcode kan als subroutines, maar ook als functies, in een functie aangeroepen worden*.

Uit deze punten blijkt, dat je een programma kunt vergelijken met een boomstructuur. Probeer te begrijpen dat elke aanroep een tak van de boom is en na de aanroep de uitvoer verder gaat met de oudertak waar de aanroep was. Als er geen takken meer zijn, zal de uitvoer verder gaan in het hoofdprogramma (de stam).

Opdracht:

Schrijf een programma met subroutines en/of functies dat onderstaande doelen uitvoert. Bepaal zelf wanneer je subroutines en/of functies nodig denkt te hebben.

1. **Hoofdprogramma: doe zolang de invoerwaarde niet nul is.**
2. **Roep onderstaande doelen aan in de lus.**
3. **Maak een tafel van de ingevoerde waarde.**
4. **Maak een macht van twee van de ingevoerde waarde.**
5. **Deel de ingevoerde waarde door 10 willekeurige getallen. Zorg ervoor dat je niet door nul deelt. Gebruik een lus.**

Opdracht:
Schrijf een programma om twee getallen in te voeren. Gebruik dezelfde 'doe zolang' lus als van de vorige opdracht. Gebruik het eerste getal om de Kelvin te berekenen en het tweede getal om de Celsius te berekenen. Denk aan het getal 273 die je nodig hebt. Zorg er eventueel voor dat de temperatuur niet onder de -273 graden Celsius is, want dat is nul Kelvin. Een negatieve Kelvin bestaat niet!

Opdracht:
Haal uit de eerste opdracht punt 4 eruit en schrijf er een programma van. Gebruik een andere variabele als een macht in plaats van macht 2. Schrijf weer een aparte functie.

Opdracht:
Schrijf een programma om de Fahrenheit te berekenen. Als je de formule niet kent, kun je die opzoeken op Internet. Gebruik een eigen functie als doel en gebruik eventueel een getal die ingevoerd wordt.

7g. Voorwaardelijk aanroepen

In het hoofdstuk Structuren weten we dat een blokcode uitgevoerd kan worden als aan een voorwaarde wordt voldaan. In zo'n blokcode kan nog eens een voorwaarde worden gemaakt. Dat kan weer voor een andere blokcode zorgen.

Al deze blokken kunnen we nesten, zoals we zagen bij IF … THEN structuren en SELECT … END SELECT structuren. Veel blokken nesten is toegestaan, maar het programmeren maakt het er niet makkelijker op. Het is een goed idee om een blokcode apart te programmeren en in de voorwaarde aan te roepen. Dit noemen we ook wel *voorwaardelijk aanroepen*.

Oude BASIC dialecten kennen het ON commando om sequentieel te bepalen waar naartoe gesprongen moet worden. Deze oude branch- of vertakmethode wordt niet door Liberty BASIC ondersteund. Het is een oude voorwaardelijke structuur dat lijkt op de tegenwoordige SELECT CASE structuur.

We kunnen bijvoorbeeld schrijven:

```
SELECT CASE expressie1
        CASE 0
                CALL BlokA
        CASE 1
                CALL BlokB
        CASE 2
                CALL BlokC expressie2
        CASE 3
                CALL BlokD
        CASE ELSE
                CALL BlokError expressie1
END SELECT

SUB BlokA
        PRINT "BlokA wordt uitgevoerd."
END SUB

SUB BlokB
        PRINT "BlokB wordt uitgevoerd."
        PRINT "Ik zag twee beren, broodjes smeren."
END SUB

SUB BlokC expressie2
        PRINT "BlokC wordt uitgevoerd."
```

```
        SELECT CASE expressie2
                CASE 10
                        CALL BlokD
                CASE n          'kies voor n een getal of een expressie
                                '(zie hoofdstuk Structuren)
                '...            Maak hier meer CASEs met eventueel een CASE ELSE
        END SELECT
END SUB

SUB BlokD
        PRINT "BlokD wordt uitgevoerd.""
        PRINT "Kan ook uitgevoerd worden als een voorwaarde in BlokC waar is."
END SUB

SUB BlokError getal
        PRINT "BlokError wordt uitgevoerd."
        PRINT " Het resultaat "; getal; " van expressie1 is niet geldig!"
END SUB
```

Het zou niet leuk zijn wanneer je in plaats van alle aanroepingen in de cases alle blokcode daar in gaat zetten. Je zou dan ook de tweede SELECT CASE structuur in BlokC in de eerste SELECT CASE structuur moeten nesten. Zoals ik al zei is dat niet erg, maar het is niet verstandig om dat te doen. Blokken code buiten een hoofdprogramma of buiten een andere grote blok programmeren geeft rust in je programma. Fouten zoeken gaat dan gemakkelijker omdat je dan niet in alle geneste code hoeft te zoeken.
Stel dat er wat fout gaat in een blok die je aanroept vanuit BlokC, bijvoorbeeld een extra CASE dat veel code bevat. Je kunt niet de fout vinden door in de lijst van alle subroutines en functies te kijken. Dat lukt wel als je die extra code in een subroutine zet. Als in het blok een fout zit, dan weet je dat BlokC in orde is.

Bovenstaande uitleg is met een SELECT CASE structuur gedaan, maar hetzelfde geldt voor IF … THEN structuren met eventueel een ELSE blok. Dat kun je ook nesten.

Conclusie:
Gebruik je eigen programmeerstructuur. Nest blokken code wanneer je denkt dat het kan. Maak anders gebruik van aparte subroutineblokken en roep ze aan. Het voordeel is dat je deze blokken overal aan kunt roepen wanneer je denkt ze nodig te hebben. Je hoeft zo'n blok dan niet nog eens te schrijven.

7h. Subroutines en functies functioneel gebruiken

Uit de vorige paragraaf weten we dat we blokken code apart als subroutines kunnen schrijven en aan kunnen roepen wanneer we willen: normaal, voorwaardelijk of genest. Maar soms kunnen subroutines en functies ook anders werken, namelijk *functioneel*.

Soms geldt een blokcode niet altijd om apart te gebruiken omdat het hoofdprogramma of een andere subroutine of functie te groot wordt. Het verdelen van code kan ook voor functionaliteit zorgen.

1. Je maakt een plaat genaamd Plaat.
2. De plaat moet een grootte hebben: lengte * breedte. Test of de lengte en breedte geldige waarden zijn.
3. Eventueel ook een dikte (met een voorwaarde testen of het niet te dik wordt en de dikte niet negatief of nul is.
4. Je geeft hem een kleur.
5. Eventueel een boolean teruggeven of het gelukt is of niet.

Uit deze stappen is een functionele subroutine of functie ontstaan. Alle punten hebben waarden nodig die je als parameters in de definitie aanmaakt.

Je kunt nu meer platen in verschillende groottes en kleuren programmeren met één functioneel blok, die je alleen maar hoeft aan te roepen.
Wat er in de subroutine of functie allemaal gebeurt, hoeft niet meer naar worden gekeken. Deze techniek wordt in vele programmeertalen gebruikt. Bijvoorbeeld arrays die je in subroutines zo kunt maken dat alleen de argumenten die je in de aanroepingen meegeeft de arrays beïnvloedt, zonder het zelf hoeft te wijzigen. Dit wordt *encapsulatie* genoemd; als een doos die je alleen gebruikt zonder hem open hoeft te maken.

Encapsulatie (= Engels: encapsulation) komt vooral voor in de OOP techniek. Hoewel Liberty BASIC geen OOP techniek kent, kunnen we wel daar zeer dichtbij komen, zoals we nu zien met de subroutines en functies.

Meer hierover in Deel 3 van het boek.

Opdracht:
Probeer een programma te schrijven met een subroutine of functie dat bovenstaande punten gebruikt. Maak je eigen blok dat iets moet doen. Het hoeft dus geen plaat te zijn.
Zorg ervoor dat je het meerdere keren met andere argumentwaarden kunt aanroepen, zonder dat je de definitie van het blok hoeft te wijzigen.

Opdracht:
Schrijf een programma dat 10 soorten tafels weergeeft. Alle 10 tafels mogen willekeurige getallen zijn. Voer elke tafel één voor één uit en laat de gebruiker een toets indrukken om elke tafel te kunnen zien. Maak één tafelblok in een subroutine en zorg ervoor dat je de subroutine telkens aan kunt roepen voor alle tafels.

8. Bestandsbeheer

Gegevens kunnen we lezen met een READ commando en met wat DATA regels. Het probleem is wel dat DATA regels vast staan. We kunnen geen gegevens aanvullen, wijzigen en verwijderen, tenzij het ingelezen wordt in een array. Maar dan staan de DATA regels nog steeds vast, want je gaat dan verder werken met de array.
In hoofdstuk 8 hebben we dit gezien en weten we dat we de data zelf niet tijdens de uitvoering kunnen wijzigen.

Willen we juist de gegevens kunnen wijzigen, dan is er een oplossing: werken met bestanden. We hebben geen READ en DATA meer nodig en we kunnen altijd de bestandsgegevens bewerken en weer opslaan.

Werken met bestanden bestaat in BASIC al heel lang. Met de oude PRINT# en INPUT# commando's konden we gegevens wegschrijven en inlezen. Deze oude manier werkt ook in Liberty BASIC.

Liberty BASIC kent nog meer manieren om met bestanden te werken. In plaats van bestandsgegevens inzien, kunnen we ook naar bestanden zoeken en bijvoorbeeld controleren of een bestand bestaat.

8a. De tweedimensionale bestandsarray en het FILES commando

Liberty BASIC onderhoudt een array speciaal voor bestandsbeheer. Zo'n array is nodig, want Liberty BASIC heeft zelf geen functies voor bestandsbeheer.
Meestal wordt de array Info$(10, 10) gebruikt. Om een vergissing te voorkomen; het woord Info is geen sleutelwoord van Liberty BASIC. Je kunt zelf een andere naam voor de array bedenken.
Je hoeft ook niet met 10, 10 te beginnen. Een tweedimensionale array aanmaken met 0, 0 is voldoende. Wanneer de array doorgeven wordt als argument aan het FILES commando, wordt de lengte van beide indexen bepaalt.

De eerste index kan variërend zijn. Elk element boven 0 is een bestand waarmee met de tweede index de notities over het bestand wordt opgeslagen, zoals de lengte van het bestand. De tweede index is niet variërend. Dit bestaat altijd uit 4 elementen.

```
DIM Bestandsbeheer$(0, 0)
FILES dirNaam$, Bestandsbeheer$()
```

We kunnen controleren of er bestanden in de directory aanwezig zijn:

```
IF Bestandsbeheer$(0, 0) > 0 THEN _
        PRINT "De directory heeft bestanden."
```

Stel dat de directory niet zou bestaan. Bovenstaande code zal daar niet op attenderen. De lengte van het aantal bestanden zal gewoon nul zijn. Of nu een directory niet bestaat of leeg is, beide geven een nul terug.

Hoe kunnen we weten of een directory bestaat? Dat controleren we met de tweede index als element 3.

```
IF Bestandsbeheer$(0, 3) = "" THEN _
        PRINT "Directory niet gevonden."
```

Om te weten hoe de array in elkaar zit, kom ik hieronder met een tabel.

Element van eerste index	Element van tweede index	Functie
0	0	Aantal bestanden
0	1	Aantal subdirectory's
0	2	Welk station (drive) de bestanden in staan
0	3	Welke directory de bestanden in staan
<bestandsnummer>	0	Bestandsnaam
<bestandsnummer>	1	De grootte van het bestand
<bestandsnummer>	2	De datum en tijd van het bestand
<bestandsnummer>	3	Het attribuut van het bestand
<aantal bestanden + 1>	0	Geeft het volledige pad met een gevonden subdirectory
<aantal bestanden + 1>	<subdirectorynummer>	Geeft de subdirectory zonder het volledige pad

Door de elementen samen te gebruiken, kun je de bestanden en subdirectory's goed beheren. Door eerst te controleren hoeveel bestanden er zijn voordat je een bestandsnummer opgeeft, voorkom je fouten.

In mijn Proboard account kun je, als ze er zijn, de modules en sjablonen vinden die je kunt overnemen. Er zal ook een module File aanwezig zijn.
Meer over mijn links en mijn website, zie appendix Informatie en links.

In deel 3, hoofdstuk Datastructuren, wordt er uitgelegd om met namen te kunnen werken zonder dat je de elementnummers hoeft te onthouden.

Om alle bestanden te printen met de naam en grootte, zou je onderstaande code kunnen gebruiken.

```
aantal = val(Bestandsbeheer$(0, 0))
IF aantal > 0 THEN
        FOR bestand = 1 TO aantal
                PRINT Bestandsbeheer$(bestand, 0), Bestandsbeheer$(bestand, 1)
        NEXT bestand
ELSE
        PRINT "Er zijn geen bestanden."
END IF
```

Opdracht:
Schrijf een functie om te bepalen hoeveel bestanden er zijn.

Opdracht:
Schrijf een functie om te bepalen of een directory wel bestaat.

Opdracht:
Schrijf een programma dat alle bestanden en subdirectory's op het mainwin venster afdrukt.

Opdracht:
Schrijf een programma om te bepalen of een gegeven directory leeg is. Druk op het mainwin venster af of het geen bestanden heeft en of geen subdirectory's heeft. Laat ook een lijst zien van bestanden en/of subdirectory's mochten ze aanwezig zijn. Kijk in het bovenstaand tabel wat je nodig hebt.

8b. De gegevens van de bestanden; de commando's OPEN en CLOSE

De gegevens van de bestanden kunnen we inlezen en ook weer wegschrijven. Wie de oude BASIC programmeertalen kent, kan zich misschien nog herinneren hoe we dat programmeerden. De oude manier bestaat in Liberty BASIC ook. Het is vrij eenvoudig om met bestandsgegevens te werken.

Met het OPEN commando openen we een bestand om toegang te krijgen met de inhoud. Als we klaar zijn, sluiten we de toegang met het CLOSE commando.

Het OPEN commando is in Liberty BASIC heel uitgebreid. We kunnen ook DLL bestanden openen en kunnen we toegang krijgen tot de GUI van Liberty BASIC. Zie meer daarover in Deel 3.

Syntax:

OPEN <bestandsnaam> FOR INPUT | OUTPUT | APPEND | RANDOM | BINARY AS #handle [LEN = <n>]
CLOSE #handle

Waarbij de pipe symbool '|' betekent dat je één van de sleutelwoorden kunt gebruiken.

De LEN instelling werkt alleen met RANDOM bestanden.

Er bestaan verschillende soorten bestanden:

- Sequentiële bestanden – deze bestanden worden op volgorde van begin tot eind gelezen en geschreven. Er is geen mogelijkheid om een deel van het bestand te lezen of te schrijven.
- Binaire bestanden – deze bestanden worden gelezen of geschreven vanaf een opgegeven startplaats.
- Random Access bestanden – deze bestanden worden gelezen of geschreven per record. De lengte van de records wordt bepaald door het commando LEN.

8c. Sequentiële bestanden

Bestanden met tekstgegevens worden sequentieel gelezen en weggeschreven. Dit doen we met de bekende INPUT# en PRINT# commando's.

Bestandsgegevens lezen we met het woord INPUT. De gegevens wegschrijven doen we met OUTPUT. Wanneer een opgegeven bestand niet bestaat, geeft Liberty BASIC een foutmelding. Vaak gebruiken programmeurs een ON ERROR GOTO (zie hoofdstuk 11 Foutafhandelingen) om te voorkomen dat de fout het programma laat stoppen. We kunnen echter gebruik maken van de bestandsarray, zie vorige paragraaf, om te controleren of het bestand bestaat.

Onderstaande voorbeelden komen uit de Help van Liberty BASIC.

```
'maak een bestand aan
OPEN "test.txt" FOR OUTPUT AS #1
PRINT #1, "123 Sesame Street, New York, NY"
CLOSE #1
```

Indien het bestand "test.txt" niet bestaat, wordt deze automatisch aangemaakt. Controleer eerst of het bestand bestaat als je niet wilt dat deze automatisch aangemaakt wordt.

```
'INPUT
OPEN "test.txt" FOR INPUT AS #1
INPUT #1, txt$
PRINT "INPUT item is: ";txt$
CLOSE #1
```

Het resultaat is: INPUT item is: 123 Sesame Street

```
'LINE INPUT
OPEN "test.txt" FOR INPUT AS #1
LINE INPUT #1, txt$
PRINT "LINE INPUT item is: ";txt$
CLOSE #1
```

Het resultaat is: LINE INPUT item is: 123 Sesame Street, New York, NY

```
'INPUT$
OPEN "test.txt" FOR INPUT AS #1
txt$ = INPUT$(#1, 10) 'lees 10 tekens
PRINT "INPUT$ item is: ";txt$
CLOSE #1
```

Het resultaat is: INPUT$ item is: 123 Sesame

```
'INPUTTO$
OPEN "test.txt" FOR INPUT AS #1
txt$ = INPUTTO$(#1, " ") 'gebruik een spatie als scheidingsteken
PRINT "INPUTTO$ item is: ";txt$
CLOSE #1
```

Het resultaat is: INPUTTO$ item is: 123

Zoals je de programma's ziet mag je achter INPUT en PRINT een spatie voor het #-teken gebruiken.

Het verschil tussen een INPUT# en een LINE INPUT# is dat LINE INPUT# een hele regel gegevens inleest totdat er een newline code wordt gevonden (crln = carriage return & linefeed). Het commando INPUT# leest de gegevens tot een komma gevonden wordt. Je kunt INPUT# gebruiken om komma-gescheiden gegevens te lezen, maar de functie INPUTTO$ is de beste manier om dat te doen. De functie is beter geschikt als je een bepaald teken als scheidingsteken wilt gebruiken en je niet per sé wilt dat de komma overgeslagen wordt terwijl je het graag in de tekst wilt hebben.

Tot nu toe is steeds één gegeven gelezen, maar een bestand kan uit meer gegevens bestaan. Je hebt een herhalingslus nodig om meer gegevens in te kunnen lezen. Een gewone lus kan hiervoor niet gebruikt worden, omdat we niet weten hoeveel gegevens er zijn.

Met de EOF() functie wordt gecontroleerd of de pointer aan het eind van het bestand is of niet. De functienaam is een afkorting van End-Of-File.

```
OPEN bestandsnaam$ FOR INPUT AS #file
WHILE EOF(#file) = 0
       LINE INPUT #file, tekst$
       PRINT tekst$
WEND
```

De functie EOF() geeft als resultaat een boolean waarde, 0 is niet het einde en niet 0 is wel het einde. We kunnen in plaats van controleren op 0 ook de functie NOT() gebruiken.

```
WHILE NOT(EOF(#file)) ... WEND
```

Om de gegevens weer weg te schrijven, gebruik je OUTPUT.

```
  'maak een bestand (als deze niet bestaat)
  OPEN "test.txt" FOR OUTPUT AS #1
  'schrijf wat gegevens met regel scheidingstekens
  PRINT #1, "regel één "
  PRINT #1, "regel twee "
  'schrijf wat gegevens zonder regel scheidingstekens
  PRINT #1, "item drie ";
  PRINT #1, "item vier ";
  'meer gegevens met regel scheidingstekens toegevoegd
  PRINT #1, "item vijf"
  PRINT #1, "klaar"
  CLOSE #1

  'INPUT om te zien wat we weg hebben geschreven
  OPEN "test.txt" FOR INPUT AS #1
  txt$ = INPUT$(#1, LOF(#1))
  PRINT "Inhoud van het bestand: "
  PRINT
  PRINT txt$
  CLOSE #1
  END
```

Als resultaat krijg je:

```
Inhoud van het bestand:

regel één
regel twee
item drie item vier item vijf
klaar
```

Hoewel item vijf apart is weggeschreven, wordt deze in de andere items regel ingelezen. Dat komt door de puntkomma die achter item vier staat.

In plaats van met de functie EOF() elke regel te lezen in een WHILE lus, kun je ook de inhoud in één keer lezen.

Met het INPUT$ commando kun je opgeven hoeveel tekens je per keer wilt lezen. Met de LOF() functie, de afkorting van Length-Of-File, kun je alles in één keer inlezen. Soms kan dat handig zijn maar als de tekst geen scheidingstekens heeft, maar bestaat uit regels met carriage return & linefeed tekens dan werkt een LINE INPUT beter.

De functie INPUT$() is geschikt voor CSV bestanden. Gegevens die scheidingstekens bevatten. Zoals je gezien hebt kun je de functie INPUTTO$() gebruiken om zulke gegevens te lezen, maar met de functie INPUT$ kun je ervoor zorgen om bijvoorbeeld een array te vullen, door in één keer te lezen en de tekst uit de string te halen. Met INPUTTO$() heb je een lus nodig en kun je niet meteen de array vullen. Je moet dan daarna het nog een keer inlezen om dan de array te vullen, omdat met de eerste keer lezen bepaald kon worden hoeveel gegevens erin zitten.

Onderstaande code geeft een voorbeeld hoe we een array vullen met INPUTTO$(). Het volgende voorbeeld laat zien dat het beter kan met INPUT$ om de array te vullen door alles uit de stringvariabele te halen.

Het bestand bestaat uit de volgende gegevens: Dit ,is ,een ,test, met, aparte, woorden.

```
OPEN "Testbestand.txt" FOR INPUT AS #t
i = 0
WHILE NOT(EOF(#t))
        w$ = INPUTTO$(#t, ",")
        i = i + 1
WEND
aantal = i
CLOSE #t
DIM woorden$(aantal - 1)
OPEN "Testbestand.txt" FOR INPUT AS #t
i = 0
WHILE NOT(EOF(#t))
        w$ = INPUTTO$(#t, ",")
        woorden$(i) = w$
        i = i + 1
WEND
CLOSE #t
FOR i = 0 TO aantal - 1
        PRINT woorden$(i); " ";
NEXT i
```

Deze code wordt vaak gebruikt om een array te kunnen vullen. Omdat we geen array dynamisch kunnen vullen tijdens het lezen van de gegevens, moeten we de gegevens voor de tweede keer inlezen, want dan kun je weten hoeveel gegevens er zijn.

Onderstaande code laat een andere mogelijkheid zien om een array te vullen door maar één keer de gegevens te lezen.

```
OPEN "Testbestand.txt" FOR INPUT AS #t
woorden$ = INPUT$(#t, LOF(#t))
CLOSE #t
i = 0
w$ = ""
DO
    w$ = WORD$(woorden$, i + 1, ",")
    i = i + 1
LOOP UNTIL w$ = ""
aantal = i
DIM woorden$(aantal - 1)
FOR i = 0 TO aantal - 1
    woorden$(i) = WORD$(woorden$, i + 1, ",")
NEXT i
'verderop in het programma
FOR i = 0 TO aantal - 1
    PRINT woorden$(i);
NEXT i
```

Beide voorbeelden geven als resultaat:

Dit is een test met aparte woorden.

Je ziet dat de gegevens in één keer inlezen met INPUT$ een hoop werk minder is. Houd er wel rekening mee dat deze mogelijkheid alleen werkt als je gegevens gescheiden zijn door een teken, zoals een komma.
Wil je geen komma als scheidingsteken, dan kan een punt ook handig zijn om hele zinnen met de WORD$ functie aan de array elementen toe te kennen. Een komma wordt dan gewoon in een zin meegenomen.

Wil je liever wel in een lus woord voor woord of regel voor regel inlezen, dan is dat alsnog prima om te doen, maar denk er wel aan dat je dan eerst bepaalt hoeveel elementen je nodig hebt voor je array.

Als je een foutmelding krijgt omdat het bestand niet gevonden kan worden, dan kan dat één van de twee redenen hebben:

1. Het Testbestand.txt staat niet in dezelfde map als waar je programma in staat.
2. Je hebt waarschijnlijk nog niet je programma opgeslagen. Sla eerst je programma op in dezelfde map als waar het Testbestand.txt in staat.

Je kunt meer gegevens aan hetzelfde bestand toevoegen. Dit kan door APPEND in de OPEN statement te gebruiken in plaats van OUTPUT.

```
OPEN "Testbestand.txt" FOR APPEND AS #t
PRINT #t, "Nog ,meer ,nieuwe ,woorden ,toegevoegd."
CLOSE #t
```

Voer het één en laatste voorbeeld nog eens uit. Je zal zien dat de nieuwe woorden in de array zijn toegevoegd. Het voorbeeld daarvoor om twee keer te moeten lezen, werkt ook nog steeds. Hoe je het zelf wilt doen maakt niet uit.

Opdracht:
Schrijf een programma dat zelf een bestand aanmaakt en gegevens wegschrijft. Vul een array met ingevoerde gegevens om die weg te schrijven, of schrijf het meteen weg tijdens het invoeren.
Lees daarna alles weer in. Leeg het mainwin venster en druk de ingelezen gegevens op het scherm af.

8d. Binaire bestanden

In plaats van gegevens sequentieel als tekst weg te schrijven, kunnen we ook gegevens binair wegschrijven. We hoeven niet te denken aan nullen en enen. Dit werkt per teken.

We hoeven Liberty BASIC niet te vertellen of we in het bestand willen lezen of willen schrijven. Met binaire bestanden kunnen we het beide tegelijk doen.
Omdat een binair bestand niet sequentieel is, kunnen we elke plaats in het bestand bepalen en kiezen wat we willen doen: lezen of wegschrijven.

Onderstaand voorbeeld komt uit de Help van Liberty BASIC, maar ik zal proberen meer uitleg te geven over wat het doet.

```
'binair bestand voorbeeld
OPEN "myfile.bin" FOR BINARY AS #myfile
txt$ = "I like programming with Liberty BASIC."
PRINT "Original data in file is: ";txt$

'schrijf wat gegevens naar het bestand
PRINT #myfile, txt$

'geef de positie van de bestandspointer
nowPos = LOC(#myfile)

'verplaats de bestandspointer
nowPos = nowPos - 14
SEEK #myfile, nowPos

'lees de gegevens vanaf de huidige locatie
INPUT #myfile, txt$

'print txt$ in mainwin
```

```
PRINT "Data at ";nowPos;" is: ";txt$

'verplaats de bestandspointer
SEEK #myfile, 2

'schrijf wat gegevens ergens in het bestand
PRINT #myfile, "love"
PRINT STR$(LOC(#myfile) - 2); " bytes written"

'verplaats de bestandspointer naar het begin
SEEK #myfile, 0

'lees de gegevens
INPUT #myfile, txt$

'print gegevens in mainwin
PRINT "New Data is: ";txt$

'Sluit het bestand
CLOSE #myfile
END
```

De uitvoer zal zijn:

Original data in file is: I like programming with Liberty BASIC.
Data at 24 is: Liberty BASIC.
4 bytes written
New Data is: I love programming with Liberty BASIC.

Zoals je ziet heb je aan een binair bestand meer vrijheid omdat je kunt lezen en schrijven tegelijk. Je kunt van plaats veranderen en de gegevens wijzigen in het bestand.

De functie LOC() is de afkorting van LOCation. Het geeft de plaats terug waar de pointer in staat. De pointer verplaatst zich per byte van hoeveel tekens er weggeschreven worden of wanneer je het commando SEEK gebruikt. Het tweede argument dat je bij SEEK opgeeft is de plaats waar je de pointer plaatst. Houd er rekening mee dat SEEK absoluut werkt. Je kunt niet relatief de pointer verplaatsen.
Zie ook in het voorbeeld dat de eerste byte plaats nul is. De laatste byte is dus de lengte minus 1 van de inhoud.

Omdat de pointer absoluut werkt, is het lastig te vertellen hoeveel tekens je bijvoorbeeld gewijzigd hebt. Je kunt natuurlijk de LEN() functie gebruiken voor de lengte van de tekens die je gewijzigd hebt, maar in het voorbeeldprogramma laat Carl Gundel expres zien dat 2 bytes afgetrokken moeten worden om op het mainwin venster te kunnen printen dat 4 bytes geschreven zijn en niet 6 bytes.

Maar misschien is het nog niet helemaal duidelijk wat er gebeurt.

Het gaat om de 4 bytes vanaf het tweede teken.
Er staat eerst: I like. Met SEEK wordt de plaats op 2 gezet. Dat is de derde plaats in het bestand, omdat de eerste plaats voor SEEK plaats 0 is. Omdat zowel SEEK als LOC met een absolute plaats werken en de letters like worden overschreven met love, zal de pointer op plaats 6 komen te staan. Door die plaats met minus 2 te berekenen, zal het juiste aantal bytes geprint worden.

We gaan nieuwe gegevens in het bestand schrijven. Voeg onderstaande code tussen CLOSE en END toe.
Deze code komt niet uit de Help.

```
PRINT
PRINT "Write next line"
OPEN "myfile.bin" FOR BINARY AS #myfile
PRINT #myfile, " The result with the new text"

'lees alle gegevens
SEEK #myfile, 0
INPUT #myfile, txt$

'print het weer in de mainwin
PRINT "De data with added data is:"
PRINT txt$
CLOSE #myfile
```

Als je het programma weer uitvoert, dan zie je wat je niet zou verwachten. Hoewel ik in het PRINT commando zeg: 'De data with *added* data is:', zijn toch de nieuwe gegevens niet toegevoegd, maar hebben de bestaande gegevens overschreven.

Is er een mogelijkheid om gegevens toe te voegen?

Het is gek, maar wat ik nu laat zien is niet in de Help van Liberty BASIC te vinden. We kunnen ook in een binair bestand gegevens toevoegen door BINARY samen met APPEND te gebruiken. Wijzig bovenstaand stukje code met onderstaande code.

```
PRINT
PRINT "Write next data"
OPEN "myfile.bin" FOR APPEND BINARY AS #myfile

'lees de gegevens in
SEEK #myfile, 0
INPUT #myfile, txt$

'verplaats de bestandspointer naar het eind
newPos = LEN(txt$)
SEEK #myfile, newPos

'schrijf de nieuwe gegevens weg
PRINT #myfile, " The result with the new text"

'print het weer in de mainwin
SEEK #myfile, 0
INPUT #myfile, txt$
PRINT "De data with added data is:"
PRINT txt$
CLOSE #myfile
```

Maar ook al gebruiken we nu APPEND, dan alsnog worden de nieuwe gegevens niet toegevoegd als we de pointer niet aan het eind zetten. Daarom is dat nu wel gedaan.

In het OPEN statement staat er APPEND BINARY. Deze volgorde moet je ook blijven gebruiken. Je kunt niet BINARY APPEND zeggen.

In een binair bestand kunnen we geen regeleinde tekens schrijven. Een carriage return met een linefeed (CHR$(13) en CHR$(10) worden niet meegeschreven.

Wanneer we bij SEEK een plaats meegeven dat hoger is dan de lengte van de gegevens, zal er geen foutmelding verschijnen. Maar er zal ook niets gebeuren, want het aantal gegeven plaatsen dat teveel is zal genegeerd worden.

8e. Random Access bestanden

Random Access betekent Willekeurige Toegang.
Anders dan de vorige bestandstypes is dat deze een gegevensstructuur verwerkt. Dit type maakt gebruik van records.
Bij de OPEN syntax zag je een optionele optie LEN. Verwar het niet met de LEN() functie. De LEN optie heeft wel met een lengte te maken. Met de LEN optie geeft je niet de gegevenslengte op, maar de lengte van het record die je nodig hebt.
Met het commando FIELD declareer je de velden van het record. De syntax is:

FIELD #handle, <lengte1> AS <variabele1>[, <lengte2> AS <variabele2>, [… <lengte*n*> AS <variabele*n*]]

Hoe je ermee werkt zie je hieronder.

```
OPEN "Personen.dat" FOR RANDOM AS #dat LEN = 96
FIELD #dat,_
        30 AS Naam$,_
        30 AS Adres$,_
        30 AS Woonplaats$,_
```

```
        6 AS ID

'velden vullen en record wegschrijven
Naam$ = "John van Dijk"
Adres$ = "Kyrastraat 4"
Woonplaats$ = "Rondjestad"
ID = 1
PUT #dat, 1

'record lezen en de velden gebruiken om te printen
GET #dat, 1
PRINT "Naam: ", Naam$
PRINT "Adres: ", Adres$
PRINT "Woonplaats: ", Woonplaats$
PRINT "ID: ", ID
```

In principe kun je de velden niet als gewone variabelen beschouwen. De velden van een record zijn altijd globaal. Zelfs wanneer je het wegschrijven en inlezen in subroutines zou doen, zoals onderstaand programma laat zien.

```
OPEN "Personen.dat" FOR RANDOM AS #dat LEN = 96
FIELD #dat,_
    30 as Naam$,_
    30 as Adres$,_
    30 as Woonplaats$,_
    6 as ID
CALL LokaalPutTest
CALL LokaalGetTest
CLOSE #dat

SUB LokaalPutTest
    Naam$ = "John van Dijk"
    Adres$ = "Kyrastraat 4"
    Woonplaats$ = "Rondjestad"
    ID = 1
    PUT #dat, 1
END SUB

SUB LokaalGetTest
    GET #dat, 1
    PRINT "Naam: ", Naam$
    PRINT "Adres: ", Adres$
    PRINT "Woonplaats: ", Woonplaats$
    PRINT "ID: ", ID
END SUB
```

Een voordeel is met records dat je niet steeds de veldnamen als argumenten en parameters hoeft te definiëren in de subroutines. Het commando GET# weet in welke veldnamen de gegevens geplaatst moeten worden. Bovendien zou het anders heel lastig zijn wanneer je met de GUI programmeert, omdat niet altijd parameters gebruikt kunnen worden. Zie meer daarover in Deel 3 Hoofdstuk Datastructuren.

Een nadeel is dat je geschikte veldnamen moet kiezen om verwarring met gewone variabelen te voorkomen. Beschouw bijvoorbeeld het ID veld niet als gewone variabele. Stel dat je deze plotseling als lusteller zou gebruiken of je kent per ongeluk een string toe aan Naam$. Het commando PUT# zal zonder aarzelen de velden weer gebruiken om weg te schrijven.

Je kunt ook alles inlezen en in een array bewaren, maar je kunt niet de array gebruiken om daarmee weg te schrijven of om in te lezen. Je hebt toch weer de veldnamen nodig. Voor het navigeren van de records is een array handig, want je hoeft dan niet steeds elke record in te lezen om te bladeren. Maar computers zijn zo snel dat je dat eigenlijk niet in de gaten hebt.

Het advies is dus om alleen de velden te gebruiken, maar geen array. Het mag wel, maar iedereen kan zelf de manier kiezen.

Wil je een record verwijderen, dan heb je een oplossing nodig. Er is namelijk geen commando om een record uit het bestand te verwijderen.

De oplossing is: schrijf alles opnieuw weg door elke record in te lezen en direct weer weg te schrijven. Komt de teller bij het recordnummer dat verwijderd moet worden, sla dan dat record over en schrijf de rest verder weg. Laat wel de nieuwe recordteller met 1 hoger gaan.

Opdracht:
Schrijf een testprogramma en probeer eens uit wat er gebeurd als je een record op die manier gaat verwijderen, door simpelweg dat recordnummer over te slaan.
Controleer daarna het bestand of het aantal records één minder is of nog steeds hetzelfde aantal heeft. Je kunt dan zelf verklaren of het bestand altijd overschreven wordt of niet.

In het hoofdstuk Datastructuren kom ik terug over arrays om te kunnen werken met tekstbestanden en binaire bestanden.

Omdat de veldnamen globale aparte namen zijn, kunnen ze overal in het programma rondzwerven. Dat is niet fijn. Vooral niet als het programma groot is. Maar helaas is er geen andere manier om de veldnamen apart te houden. In het hoofdstuk Datastructuren leg ik een techniek uit dat we wel records kunnen maken maar niet geschikt zijn bij het commando FIELD#, hoewel het geen fout veroorzaakt. Je kunt dan een verklaring zien dat de veldnamen 'niet helemaal' globaal zijn.

Opdracht:
Breid bovenstaand programma zo uit dat je meer records kunt wegschrijven en inlezen. Verhoog je ID en ook het recordnummer bij PUT#. Lees alles uit met GET# en maak onderstaande uitvoer:

Naam: <naam gegevens>
Adres: <adres gegevens>
Woonplaats: <woonplaats gegevens>
ID: <ID nummer>

Je mag, als je dat liever wilt, ook labels gebruiken in plaats van subroutines.

In het hoofdstuk Datastructuren kom ik hierop terug en gaan we kijken hoe we kunnen weergeven en navigeren doormiddel van de GUI controls.

8f. De FILEDIALOG van Liberty BASIC

Met de FILEDIALOG kunnen we bestanden kiezen. Het gekozen bestand kan dan worden gebruikt om gegevens in te lezen of weg te schrijven.
De FILEDIALOG noem ik 'van Liberty BASIC', omdat het niet het Windows bestandsdialoogvenster object is. Het roept wel die van Windows aan, maar er wordt in de titelstring gecontroleerd welk venster geopend moet worden.

Syntax:

FILEDIALOG <titel$>, <sjabloon$>, <variabele$>

Aan de hand van de titel, bepaald Liberty BASIC of het OpenDialog venster of het SaveDialog venster geopend moet worden.

Omdat Carl Gundel deze commando's Engelstalig laat werken, moeten we de woorden 'Open' of 'Save' in de titel opnemen om het juiste venster geopend te krijgen. Een titel als 'Bestand opslaan' zal niet werken, want als één van de twee Engelstalige woorden niet in de titel staat dan zal standaard het OpenDialog venster van Windows worden geopend.

Een voorbeeld is hieronder:

```
FILEDIALOG "Bestand opslaan", "*.txt", filename$
IF filename$ <> "" THEN
    PRINT filename$
ELSE
    PRINT "Bestand is niet gekozen of bestaat niet."
END IF
```

Als je dit uitvoert, dan zal de knop 'Openen' er zijn, maar niet 'Opslaan'.

Wat echter niet in de Help staat is een manier om toch een andere titel te gebruiken en de juiste knop te krijgen. Door een ‘Open’ of ‘Save’ te scheiden met de titel door een code nul, weet Liberty BASIC welk venster geopend moet worden zonder last te hebben van ‘Open’ of ‘Save’ in de titel zelf. Wat achter code nul staat, wordt niet in de titel opgenomen.

```
FILEDIALOG "Bestand opslaan"; CHR$(0); "save", "*.txt", filename$
```

In het voorbeeld is in het sjabloon een ‘*.txt’ genomen. Een sjabloon kan meer extensies hebben gescheiden door een puntkomma.

```
FILEDIALOG "Bestand openen"; CHR$(0); "open", "*.txt;*.dat", filename$
```

Hoewel nu het woord ‘open’ niet hoeft, omdat standaard altijd het OpenDialog wordt gebruikt, is het toch een goed idee om ‘open’ of ‘save’ apart op te geven.

We mogen ook in het sjabloon het station en de map opgeven.

```
FILEDIALOG "Bestand openen"; CHR$(0); "save", "C:\Eigen tekst\*.txt;D:\Databestanden\*.dat",
filename$
```

Onthoud dat FILEDIALOG niets opent of opslaat. Het zorgt er alleen voor dat we het OpenDialog venster of het SaveDialog venster kunnen openen en een bestandsnaam kunnen kiezen, verder niet.

Nadat je een bestandsnaam hebt gekozen, kun je bepalen wat je ermee wilt: gegevens inlezen of wegschrijven. Omdat het FILEDIALOG niet weet wat je van plan bent met de bestandsnaam te gaan doen, zou je de gegevens kunnen wegschrijven, terwijl je voor openen hebt gekozen of andersom. (Zie het vreemde voorbeeld hierboven. Het zal je waarschijnlijk al opgevallen zijn dat ik een “save” meegeef terwijl ik als titel de tekst “Bestand openen” heb staan).

Laat FILEDIALOG het SaveDialog venster verschijnen als je van plan bent de gegevens op te slaan met een andere bestandsnaam dan die je eerder geopend had. Wil je gewoon hetzelfde bestand wijzigen, dan heb je FILEDIALOG niet nodig.

Opdracht:
Schrijf twee functies die je OpenDialog en SaveDialog noemt. Gebruik de parameters titel$, sjabloon$ en eventueel ook de parameter bestandsnaam$. Laat de functies de nieuw gekozen bestandsnaam teruggeven. Gebruik in beide functies de ‘open’ of ‘save’ achter CHR$(0), zodat die niet in de titel$ argument meegegeven hoeft te worden.

9. Foutafhandelingen

Wanneer er ergens in het programma een fout zit, stopt Liberty BASIC met het uitvoeren van het programma en geeft een foutmelding wat er mis is.

Liberty BASIC geeft echter niet aan waar de fout zit. Dat kan lastig zijn als je een groot programma hebt. In hoofdstuk 12 kun je zien hoe je een programma kunt debuggen en hoe een breakpoint werkt.

Fouten kunnen tijdens de uitvoer afgehandeld worden, zonder dat Liberty BASIC met de uitvoer stopt.

9a. Fouten afhandelen met ON ERROR GOTO

Liberty BASIC ondersteunt de oude QBasic ON ERROR GOTO. Ook Visual Basic 6 en VBA ondersteunen dat. Oude BASIC versies zoals BASIC 7.0 gebruiken een trapping om de foutmelding en het regelnummer te kunnen onderscheppen. De variabelen ERR$, ERL en ERN worden daarvoor gebruikt. De variabele ERL geeft het regelnummer aan en de variabele ERN geeft het foutnummer aan.

Liberty BASIC kan helaas niet aantonen waar de fout precies zit. Er is alleen een ERR variabele en een ERR$ variabele om het foutnummer en de foutmelding te laten zien.

Een fout zorgt ervoor dat Liberty BASIC met de uitvoer stopt en de foutmelding geeft. Met ON ERROR GOTO kunnen we fouten onderscheppen of ook wel genoemd *afhandelen*.

Typ eens onderstaande twee regels in:

```
INPUT "Geef een getal: "; getal
PRINT 100 / getal
```

Als je dit uitvoert dan werkt het goed zolang je getallen invoert die niet nul zijn. Typ je een nul in dan zal Liberty BASIC een foutmelding geven: Division by zero.
Deze foutmeldingen zijn geen compiler foutmeldingen, want het programma wordt goed gecompileerd. Dit zijn uitvoeringsfouten en zijn vaak moeilijk te vinden. Deze is een makkelijke, want met maar twee regels code is het gelijk te ontdekken hoe het komt dat de uitvoeringsfout verscheen.

We weten dat een computer een dom instrument is. Liberty BASIC ook. Vaak worden foutmeldingen gegeven over code waar eigenlijk niet de oorzaak is. Er wordt een foutmelding gegeven dat er geprobeerd wordt te delen door nul. Echter is de PRINT regel niet de oorzaak, maar wat er ingevoerd is.

Bij grote ingewikkelde programma's is de oorzaak moeilijk te vinden. Ergens in regel duizend wordt een foutmelding gegeven terwijl regel 500 de oorzaak kan zijn. Omdat Liberty BASIC ons niet kan vertellen waar de oorzaak vandaan komt, is het daarom het beste de fout te onderscheppen en zelf een melding te geven. De gebruiker kan dan geattendeerd worden dat deze een fout heeft veroorzaakt. We kunnen echter niet vertellen welke code het veroorzaakt heeft.

Met een ON ERROR GOTO kunnen we een foutafhandeling aanroepen. Dit wordt gedaan met labels. We kunnen geen subroutine aanroepen om de afhandeling uit te voeren.

Als je in de Help leest over ON ERROR GOTO, dan staat in het Engels, maar hieronder in het Nederlands, het volgende:

Als er een fout optreedt in een gebruikersfunctie of subroutine, zal Liberty BASIC de huidige functie of subroutine verlaten en doorgaan met het afsluiten van functies en subroutines totdat het een ON ERROR-handler vindt.

Dit lijkt erop dat als er een fout is, Liberty BASIC automatisch doorgaat met afsluiten van elke subroutine of functie, totdat het een ON ERROR vindt. Maar dat gebeurt eigenlijk niet.

Wat er in het vet wordt verteld, lijkt op een aanwijzing zoals onderstaande afbeelding.

```
call Deling 0
sub Deling n
    print 5 / n
    on error goto [fout]
    exit sub
[fout]
    print "Delen door nul kan niet."
    on error
end sub
```

Als je deze code uitvoert, (zet even de regel ON ERROR in commentaar) wordt de fout echter niet onderschept. De verklaring in de Help, zoals in het Nederlands vertaald in het vet bovenaan staat, zal niet juist zijn. Tenzij Carl Gundel wat anders bedoeld.

Een tweede instelling die in de verklaring staat, namelijk over een ON ERROR handler, die moet zorgen dat Liberty BASIC stopt met het afsluiten van subroutines of functies, is ook niet juist. Als je de ON ERROR weer uit de commentaar haalt en de code nogmaals uitvoert, zal de compiler aangeven dat Liberty BASIC geen ON ERROR kent zonder een GOTO. Ook een ON ERROR 0 of een ON ERROR GOTO 0, zoals andere BASIC programmeertalen kennen om een afhandeling te stoppen, wordt niet ondersteund door Liberty BASIC.

Typ onderstaande code in. Je zult zien dat deze correct werkt.

```
INPUT "Geef een getal: "; getal
INPUT "Geef een deling: "; deelGetal
CALL Deling getal, deelGetal

SUB Deling n, d
        ON ERROR GOTO [fout]
        PRINT n / d
        EXIT SUB
        [fout]
                PRINT "Delen door nul kan niet."
END SUB
```

De schrijfwijze is nu iets anders. Zo zie je dat hier de label ingesprongen is. Toen ik de code, in de afbeelding, schreef, was ik het inspringen vergeten. Maar dat maakt voor de compiler niks uit.

Het gebruik van een ON ERROR GOTO is handig om fouten af te handelen. Maar het hoeft niet altijd op die manier. We hadden in plaats van een fouthandler met de label [fout] ook gewoon een IF … THEN … ELSE kunnen gebruiken. Gewoon controleren of het deelGetal nul is of niet.
Bovendien kun je zelf de vraag stellen: Was het wel een delingfout? Een label voor foutafhandeling kan ons niet vertellen wat voor een fout is ontstaan.

Start het programma nog eens en typ in plaats van een getal eens wat tekst. Liberty BASIC geeft geen invoer foutmelding. Het invoeren van tekst zorgt ervoor dat de numerieke variabele een nul krijgt. Dit zorgt ervoor dat weer de tekst ‘Delen door nul kan niet.’ verschijnt.

Je ziet dat Liberty BASIC niet streng is. Het doet voor ons dingen dat fijn kan zijn, maar soms ook niet. Bij het invoeren van tekst, zou er eigenlijk een Type Mismatch melding moeten verschijnen, maar Liberty BASIC negeert gewoon de tekst en geeft een nul terug aan de variabele.

De fouthandler ON ERROR GOTO kan alleen uitvoeringsfouten onderscheppen. Geen compilerfouten. Daarom zijn uitvoerfouten moeilijker te vinden dan compilerfouten. Waar de compiler de fout geeft is dan ook meteen de regel die ook echt fout is, maar tijdens de uitvoer kan de regel, waar eigenlijk echt de fout is, heel ergens anders staan.

Zoals eerder verteld over labels, kan een label de veroorzaker zijn. Deze kan bijvoorbeeld verkeerd genoemd zijn. Maar degene die naar de label moet springen, GOTO of GOSUB, krijgt altijd de schuld.

Hetzelfde geldt voor ON ERROR GOTO. Het is geen gewone branch. Het springt alleen als er ergens een fout ontstaat. Meteen komt de branch in actie en wordt de code onder de label uitgevoerd. Net als bij gewone labels moeten we ervoor zorgen dat de uitvoer niet per ongeluk in de labelblok terecht komt. In bovenstaande voorbeeld heb ik een EXIT SUB staan voor de label, zodat als er geen fout is de subroutine netjes kan worden verlaten.

Gebruik je alleen labels in de code maar geen subroutines en functies, dan maakt het niet uit waar je de label, om de fout af te handelen, plaatst. Labels werken globaal, zie Hoofdstuk 9, maar lokaal wanneer ze voorkomen in subroutines en functies.
Dat geldt ook voor labels waarmee je fouten afhandelt. Staat er een ON ERROR GOTO in een subroutine of functie, zorg er dan voor dat je de label ook in dezelfde subroutine of functie plaatst. Het is zelfs het beste om de labels met de afhandelingscode als laatste code in een subroutine of functie te plaatsen.
Soms staat toch zo’n codeblok in het midden, maar dit heeft bepaalde redenen. Je kunt meer dan één fout afhandelen en soms moet na het afhandelen weer verder worden gegaan met de uitvoer. Dat kun je doen met het RESUME commando. Je kunt dan daarna een nieuwe fout afhandelen met een nieuwe ON ERROR GOTO.

Je kunt maar één fout per ON ERROR GOTO afhandelen. Spring ook nooit met een gewone GOTO commando naar een label die bedoeld is om een code uit te voeren met een foutafhandeling.

9b. Code verder uitvoeren

Soms willen we dat de code verder uitgevoerd wordt na de regel code die de fout veroorzaakt heeft.

Er is bijvoorbeeld wat we willen doen met de functie EVAL(), waarmee we expressies in een string uit kunnen voeren. Maar het moeten dan wel correcte expressies zijn, anders kan EVAL() het niet uitvoeren en wordt het programma gestopt met een foutmelding. Een ON ERROR GOTO past precies in zulke situaties om fouten af te handelen.

```
ba = 4
S$ = "4a + 8"
ON ERROR GOTO [fout]
PRINT EVAL(S$)
END
[fout]
    PRINT "Fout in Eval, expressie onjuist!"
```

Als je de code uitvoert, zie je in het mainwin venster de fouttekst verschijnen. De expressie ‘4a + 8’ is onjuist en de code na de [fout] label wordt uitgevoerd.

Maar stel dat het programma verder moet na de foutmelding? Je zal misschien denken dat na de melding het programma verder gaat met het commando END. Voeg voor het commando END nog een PRINT commando tussen met bijvoorbeeld een tekst dat zegt: “Klaar”

De tekst “Klaar” wordt echter niet uitgevoerd dus het commando END ook niet.

Als je toch wilt dat het programma verder moet dan kun je het commando RESUME gebruiken. Het lijkt op een RETURN en dat is het ook, maar op één verschil na. Het RESUME commando voert niet de code uit na de regel PRINT EVAL(S$) zoals het RETURN commando wel zal doen. Het vervelende is dat het programma in een oneindige lus terecht komt, omdat telkens getracht wordt de EVAL() functie uit te voeren.

```
ba = 4
S$ = "4a + 8"
ON ERROR GOTO [fout]
PRINT EVAL(S$)
PRINT "Klaar"
END
[fout]
    PRINT "Fout in Eval, expressie onjuist!"
    'S$ = "ba + 8"
    RESUME
```

Als je het programma nog eens uitvoert, dan zul je zien dat oneindig de foutmelding verschijnt.

Breek het programma af en haal het commentaarteken (apostrof ‘) weg. Voer het programma nog eens uit. Nu wordt het programma wel verder uitgevoerd, maar misschien niet op de manier zoals je zou willen. Om uit de foutafhandeling te komen, moet je zelf de fout corrigeren door een geldige expressie aan de variabele toe te kennen dat de EVAL() functie als argument heeft.
Het programma wordt dus verder uitgevoerd, maar ook met een goed antwoord erbij van de functie EVAL().

Er zijn BASIC versies die een handige RESUME commando hebben. Die hebben het RESUME commando met een argument als labelnaam om een bepaald gedeelte van het programma verder uit te voeren als een fout is afgehandeld. Jammer genoeg ontbreekt de parameter bij het RESUME commando in Liberty BASIC en zit er niets anders op dan een correctie te geven voordat je RESUME aanroept.

Een oplossing is, wanneer je meerdere foutafhandelingen achter elkaar wilt doen, door ze in aparte codeblokken te programmeren, zoals in het vorige voorbeeld staat, met subroutines. Je hebt dan RESUME niet nodig. Het programma stopt niet dankzij END SUB en je zou na de CALL dat kunnen controleren door er een PRINT regel in te plaatsen met een tekst.

```
CALL FoutTest
PRINT "Klaar"

SUB FoutTest
    ba = 4
    S$ = "4a + 8"
    ON ERROR GOTO [fout]
    PRINT EVAL(S$)
    EXIT SUB
    [fout]
        PRINT "Fout in Eval, expressie onjuist!"
END SUB
```

Als je deze code uitvoert, zul je zien dat het programma zonder RESUME probleemloos verder uitgevoerd wordt.

Commodore BASIC 7.0 was de eerste BASIC versie met een foutafhandeling mechanisme. Met de commando’s TRAP en RESUME konden we fouten afhandelen en had RESUME een argument om ergens heen te springen als de fout was afgehandeld.

Waarom nu het RESUME commando in Liberty BASIC geen parameter heeft is eigenlijk wel te begrijpen. Bovenstaand programma vertelt het al waarom. Het is de END SUB die ervoor zorgt dat er een RETURN ontstaat om na de aanroep van FoutTest verder te gaan. We hebben als conclusie in principe het RESUME commando niet meer nodig. Tenzij je liever met labels programmeert, want dan heb je RESUME wel nodig. Na de label uitvoer zal er anders geen weg meer terug zijn en wordt het programma beëindigd.

Zo zie je maar dat het maar goed is dat we in Liberty BASIC met subroutines en functies kunnen programmeren.

9c. Array elementen valideren

Met een IF statement kunnen we controleren of er gedeeld wordt door nul. Een ON ERROR GOTO is dan niet nodig. Maar soms kunnen we niet anders. Een voorbeeld is het gebruiken van array elementen. Soms weten we niet hoe groot een array is, bijvoorbeeld bij het inlezen van bestandsgegevens en toekennen aan een array.

Willen we de waarden uit een array halen, dan hebben we de elementen nodig. Bestaat een element niet, dan krijgen we een foutmelding. Dit kan onderschept worden met een ON ERROR GOTO. We kunnen dus array elementen valideren.

```
INPUT "Hoeveel elementen: "; aantal
DIM a$(aantal - 1)
INPUT "Kies element: "; element
INPUT "Geef waarde: "; waarde$
```

Nu weet je hoeveel elementen je hebt en zoals je ziet is het altijd het aantal minus 1 omdat de nul ook bestaat, tenzij je een array wilt gebruiken met elementen van 1 tot en met het aantal.

Je hoeft met deze code niet speciaal af te handelen of het gekozen element buiten het bereik ligt. Dat kan wel met een IF statement. Het kan ook gebeuren dat je niet het aantal weet.

Verder met de code, wordt het bijvoorbeeld:

```
ON ERROR GOTO [OutOfRange]
a$(element) = waarde$
END
[OutOfRange]
        PRINT "Element valt buiten het bereik."
```

Eerder zei ik dat we zonder RESUME ook verder kunnen met de uitvoer dankzij de END SUB. Dit is echter niet altijd handig. We willen ook wel eens weten tot hoever een element bestaat. Om ervoor te zorgen dat het programma niet in een oneindige lus komt, trekken we telkens één element af als deze niet bestaat.

```
DIM a$(20)
n = 25
ON ERROR GOTO [stop]
validate$ = a$(n)
PRINT n
END
[stop]
    n = n - 1
    RESUME
```

Als antwoord verschijnt het getal 20. Het hoogste element van de array.

We kunnen geen RESUME gebruiken wanneer we van laag naar hoog gaan tellen. We kunnen dan alleen een WHILE lus gebruiken en uit de lus gaan als het element buiten het bereik valt.

```
aantal = 25
DIM a$(aantal)
n = 0
eruit = 0
WHILE NOT(eruit)
    ON ERROR GOTO [stop]
    validate$ = a$(n)
    n = n + 1
    IF eruit = -1 THEN
        [stop]
            eruit = 1
            n = n - 1
    END IF
WEND
PRINT "Aantal elementen: "; n
```

Ook al heeft het IF statement geen nut – de waarde van variabele eruit zal nooit -1 zijn – toch is deze regel belangrijk. Het IF statement zorgt ervoor dat niet per ongeluk de [stop] label uitgevoerd wordt. De label moet alleen worden uitgevoerd als het valideren fout gaat. Liberty BASIC vindt het niet erg om midden in een THEN blok te springen, maar normaal doen we dat ook niet. Bij deze situaties is het belangrijk het wel te doen.

Maar het kan nog beter. Zonder een IF statement en de [stop] label buiten de lus plaatsen.

```
WHILE NOT(eruit)
    ON ERROR GOTO [stop]
    validate$ = a$(n)
    n = n + 1
WEND
PRINT "Aantal elementen: "; n - 1
END
[stop]
    eruit = 1
    n = n - 1
    RESUME
```

Ik heb nu in de PRINT regel de expressie ‘n - 1’. Waarom?
Nu een RESUME wel nodig is, zal het programma verder gaan na het valideren. Daar staat een ‘n = n + 1’, ondanks dat er met 1 afgetrokken wordt in de [stop] label. Dat is maar goed ook, anders was variabele n twee teveel geweest! Sterker nog, je kunt de regel ‘n = n – 1’ niet weghalen, want anders zal de lus weer oneindig blijven.

Experimenteer er maar eens mee. PRINT de waarde van variabele n maar eens in de [stop] label om het duidelijker te kunnen zien wat er gebeurd. Doe ook maar eens een PRINT in de WHILE lus voor nog een beter overzicht.

Opdracht:
Schrijf een UBound() functie om het aantal elementen van een array terug te krijgen. Andere BASIC versies, zoals QBasic, hebben al deze functie.
Bepaal zelf wat je wilt doen. Of het eerste voorbeeld met een IF statement om te zorgen dat de [stop] label niet uitgevoerd wordt of gebruik maken van buitenstaande [stop] label en een RESUME.

Tot slot nog een volledig testprogramma met twee Prog subroutines. De eerste staat op nul, want dat is het voorgaande voorbeeld. De tweede staat op 1. Deze geeft hetzelfde resultaat met een ander aantal elementen, maar heeft meer functionaliteit. Je kunt de functie Valideer() meerdere keren handig gebruiken mochten het aantal elementen veranderen. Bovendien is een RESUME niet nodig.
Merk ook op dat in de functie Valideer niet meer in de [stop] label de variabele n met 1 verlaagt hoeft te worden. Dat komt doordat variabele n alleen in de WHILE lus veranderd en omdat we nu niet meer de ON ERROR GOTO in de lus uitvoeren.

```
IF 0 THEN CALL Prog1
IF 1 THEN CALL Prog2
SUB Prog1
    aantal = 25
    DIM a$(aantal)
    CALL Test1
END SUB
SUB Prog2
    aantal = 50
    DIM a$(aantal)
    n = 0
    WHILE Valideer(n)
        n = n + 1
    WEND
    PRINT "Aantal elementen: "; n - 1
END SUB
FUNCTION Valideer(n)
    ON ERROR GOTO [stop]
    validate$ = a$(n)
    Valideer = 1
    EXIT FUNCTION
    [stop]
        Valideer = 0
END FUNCTION
SUB Test1
```

```
    n = 0
    eruit = 0
    WHILE NOT(eruit)
        ON ERROR GOTO [stop]
        validate$ = a$(n)
        n = n + 1
    WEND
    PRINT "Aantal elementen: "; n - 1
    EXIT SUB
    [stop]
        eruit = 1
        n = n - 1
        RESUME
END SUB
```

Conclusie:
Fouten afhandelen met een ON ERROR GOTO kan handig zijn, maar het is niet altijd nodig. We kunnen vaak fouten afhandelen door bijvoorbeeld te controleren of een bestand bestaat, zie hoofdstuk Bestandsbeheer, of dat er per ongeluk gedeeld wordt door nul.

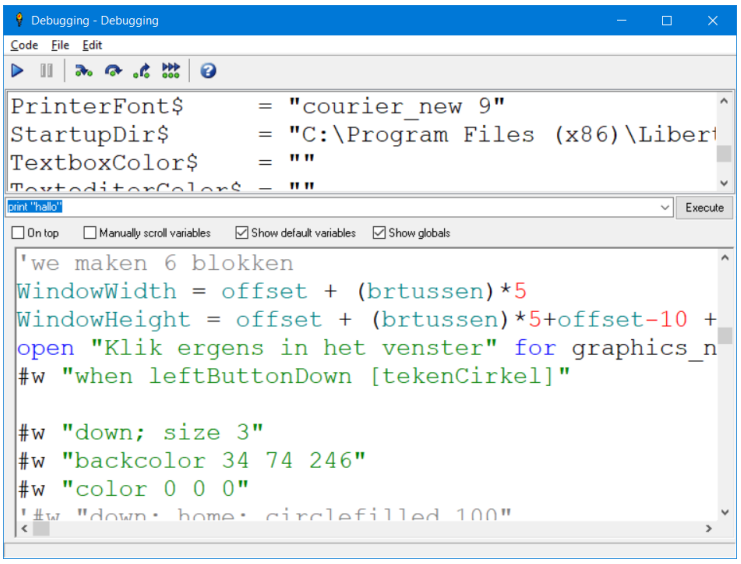
In het volgende hoofdstuk over debuggen kunnen we fouten zelfs nog sneller vinden. Bovendien is er een probleem met ON ERROR GOTO. De labels waarmee we fouten afhandelen kunnen niet met een debug bekeken worden. Zodra de debugger een ON ERROR GOTO tegenkomt, wordt het debuggen gestopt.

10. Programma's debuggen

Wanneer een IF statement of een ON ERROR GOTO niet het gewenste resultaat geeft of dat je het misschien niet eens gebruiken wilt, dan is er nog een andere oplossing. Een programma debuggen.

Debuggen is het proces van uittesten van een werkende listing. Alles kan stap voor stap gecontroleerd worden. Maak een programma of laad een programma in. Als je geen programma bij de hand hebt, gebruik dan het programma 'beginnersgridje.lsn' dat in de voorbeelden appendix, als de appendices al in het boek staan, gevonden kan worden.

Klik "lieveheersbeestje" (dat is de knop rechts van de startknop) en je geraakt in de **DEBUG mode**. Je kunt natuurlijk alleen in de editor iets wijzigen. Maar je kunt regel voor regel de uitvoering volgen en alle variabelen constant bekijken. Tussendoor kun je ook iets laten uitvoeren.



Natuurlijk kun je zelf de default waarde zien van apparatuur, zoals van je joystick of muis. Alle variabelen die tijdens het compileren zijn gemaakt zijn te zien.

Dit is de knoppenbalk van de debugger.

De blauwe driehoek is de startknop, maar voert het programma niet op de normale manier uit.
De tweede knop (in het grijs) is de stopknop.
De volgende vier knoppen zorgen voor speciale uitvoeringen van het programma.
De eerste knop voert het programma regel voor regel uit.
De tweede knop slaat een regel over.
De derde knop springt over een lus.
De vierde knop doet volle snelheid (zonder onderbreken) regel voor regel.

Het is tijdrovend om een programma met veel code regel voor regel uit te voeren.

Daarom kun je in de kantlijn van het programma een “breakpoint” plaatsen. Dat doe je met de rechter muisknop klikken links naast de regel waar je de breakpoint wilt hebben. Je kunt dan kiezen voor ‘Toggle breakpoint’ om er één aan of uit te zetten. Je kunt ook met de linker muisknop er één aanzetten.

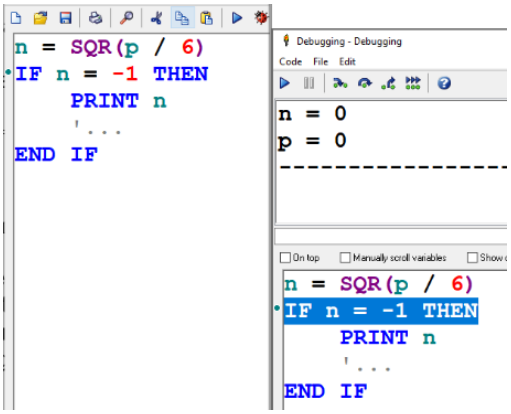
Wanneer je een breakpoint plaatst, moet je niet het programma normaal starten. Het normaal uitvoeren van een programma is altijd zonder debuggen. De eerste knop, de blauwe pijl in de debugger, stopt automatisch zodra het een breakpoint vindt. Je kunt dan met de vier knoppen ernaast, zoals eerder uitgelegd, het programma doorlopen.

Zoals je op de afbeelding ziet, laat Liberty BASIC alles zien wat deze tegenkomt.

Stel dat je de waarde van variabele n wilt weten:

```
n = SQR(p / 6)
IF n = -1 THEN
        PRINT n
        '...
END IF
```

De PRINT regel zal nooit uitgevoerd worden. Dat komt door een klein foutje. Uit een wortel berekening kan het antwoord nooit negatief zijn. Dit is met deze weinig regels makkelijk te vinden, maar vind zo’n foutje maar eens in een gigantisch groot programma. Zo’n foutje wordt ook wel een *bug* genoemd. Om te achterhalen of de PRINT regel uitgevoerd wordt, kun je een breakpoint plaatsen links naast de PRINT regel.



Open de debugger en klik op de blauwe pijl. Je ziet natuurlijk dat de breakpoint niet gevonden wordt. Op die manier weet je dus dat variabele n niet negatief kan zijn. Om te weten wat dan de waarde van variabele n is, kun je een andere breakpoint zetten. Deze zet je links naast het IF statement. Niet op de regel waar de toekenning staat. Een breakpoint wordt altijd *eerder* gevonden dan de expressie uitgevoerd wordt. Probeer dus altijd op de juiste regel een breakpoint te zetten. De afbeelding laat zien wat er gebeurt.

We hoeven nu niet de waarde van variabele p te weten. Ook al is p gelijk aan 0, we kunnen nooit als resultaat een negatieve waarde hebben. Ook niet als we een negatief getal nemen in plaats van 6.

Zoals ik in het vorige hoofdstuk verteld heb, kun je geen foutafhandelingen debuggen. Wanneer je fouten afhandelt met een ON ERROR GOTO dan ben je in principe aan het debuggen om bugs te controleren. Waarschijnlijk stopt de debugger daarom als het een foutafhandeling tegenkomt.

In plaats van de debugger te gebruiken, wordt ook vaak het mainwin venster als debugger gebruikt. Wanneer er iets niet klopt, kun je tijdens de uitvoering de waarden op het mainwin venster afdrukken die je wilt zien.

Natuurlijk gebruiken we nu nog de mainwin voor de uitvoer van het programma. In Deel 3 zullen we zien dat we de mainwin niet meer als uitvoervenster gaan gebruiken, maar gebruiken we het als debugger. Een controlevenster.

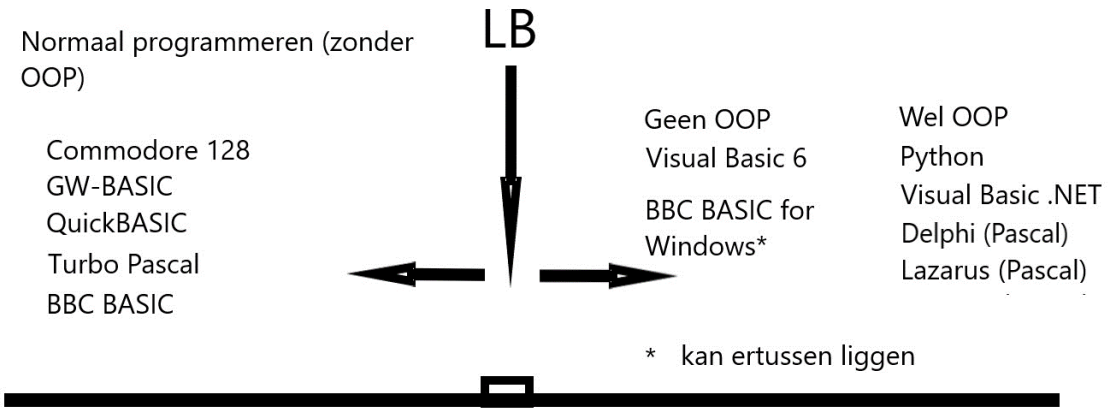
Samenvatting:
Dit was het eerste deel van het boek. Het tweede deel gaat over de GUI van Liberty BASIC en het derde deel over technisch programmeren. We zullen zien dat we veel in Liberty BASIC kunnen doen. Meer dan we in andere BASIC versies kunnen doen, zonder Visual Basic meegerekend.

Probeer Deel 1 goed te begrijpen. We gaan langzaamaan de drempel oversteken.

Succes met Deel 2 Over de Drempel.

11. Deel 2: Over de drempel met Liberty BASIC

In het begin van het boek had ik het al over de drempel. Wat bedoel ik ermee? Onderstaande afbeelding kan meer helderheid geven hoe Liberty BASIC gebruikt kan worden. Nou, in verschillende stappen. Nu we eerste deel gehad hebben, kunnen we al aardig leuke programma's schrijven. Liberty BASIC heeft echter nog meer mogelijkheden dan alleen maar het mainwin venster gebruiken.



In dit hoofdstuk gaan we voorzicht de drempel op. In Deel 2 leer je omgaan met vensters en de GUI van Liberty BASIC. BBC BASIC for Windows is al een BASIC programmeertaal over de drempel. Hiermee kunnen we ook programmeren met GUI onderdelen. Zijn voorganger, BBC BASIC, heeft dat niet. Het is wel mogelijk om met BBC BASIC apparatuur aan te sturen, zoals robotarmen, maar dan nog steeds zonder de laatste stap te hoeven gebruiken: de OOP.

Hoewel Liberty BASIC geen OOP kent, kunnen we wel ertegenaan lopen. Helaas gaat de deur niet open voor ons. We moeten het maar doen zonder Object Georiënteerd Programmeren. We kunnen wel technische code programmeren in Liberty BASIC dat zeer dicht ertegenaan ligt.

Liberty BASIC kent drie belangrijke onderdelen:

- De GUI (Graphics User Interface) zoals Windows vensters, de besturingselementen en grafische tekengereedschappen.
- Datastructuren en datatypes waar we al wat van gehad hebben, zoals arrays en gegevensbestanden. Maar Liberty BASIC kent ook mogelijkheden om structuren en modules te maken. Hoewel we nog niks met datatypes te maken hebben gehad, kent Liberty BASIC het wel. Het is uitgebreider dan alleen maar het dollarteken voor strings en zonder dollarteken voor numeriek.
- Windows API functies aanroepen is in andere programmeertalen een ramp. Veel ingewikkelder om een functie aan te kunnen roepen. Liberty BASIC heeft een commando CALLDLL om eenvoudig een API functie uit te voeren. Ook hier hebben de datatypes nodig als parameters voor die functies.

11a. Windows vensters

Tot nu toe hebben we steeds gebruik gemaakt van het mainwin venster om gegevens weer te geven. In Liberty BASIC kunnen we dus in de oude BASIC code blijven om programma's te schrijven. Laten we maar eens op de drempel gaan staan en richting het oosten gaan kijken, zie afbeelding.

In programmeertalen als C en C++ is het programmeren van Windows vensters een lastige karwei. Tegenwoordig gaat het in Visual C++ en in Visual C# wat gemakkelijker dan in C. Toch gaat het niet zo gemakkelijk dan in Liberty BASIC.

Hoe maken we een venster?
Een venster maken gaat op dezelfde manier als het openen van een bestand. De syntax van het OPEN commando is nu wat uitgebreider.

Syntax:

OPEN <titel$> FOR <windowtype> AS #<handle>
WAIT

Neem eens onderstaand voorbeeld.

```
OPEN "WindowTest" FOR WINDOW AS #w
WAIT
```

Als je deze twee regels uitvoert, zie je een Windows venster verschijnen.

Dat is alles. Je hoeft je niet te bekommeren om de creatie ervan en ook niet om de oneindige lus die uitgevoerd wordt om berichten te kunnen onderscheppen. Nu zijn er geen berichten, maar de lus wordt wel uitgevoerd. Totdat op het kruisje rechtsboven wordt geklikt.

Het commando WAIT voert de lus uit, want daardoor zal het venster aan blijven. Haal het WAIT commando maar eens weg en voer het programmaatje nog eens uit. Heel even zie je een venster, maar gelijk wordt het weer gesloten en komt Liberty BASIC met een melding dat ie het venster voor je moet sluiten. Het bericht verschijnt ook wanneer we weer het WAIT commando gebruiken en tijdens de uitvoer op het kruisje klikken.

Kennelijk vindt Liberty BASIC het niet fijn dat we het venster sluiten. Waarom niet?

Ook dit OPEN commando heeft, net als bij het openen van bestanden, een CLOSE commando nodig. Het heeft echter geen zin om het CLOSE commando na het WAIT commando te typen. Zolang er een WAIT commando is, zal alleen code, die voor actie moet zorgen, actief blijven om afgehandeld te kunnen worden.

Er is ook een actie om het venster te kunnen sluiten. Ook al sluiten we het venster, Liberty BASIC mist een trapping. Een code dat uitgevoerd wordt als we het venster gaan sluiten.

Typ tussen de OPEN en WAIT onderstaande regel:

```
#w "trapclose [quit]"
```

Deze regel is niet voldoende. Als je het uitvoert, kun je het venster niet meer sluiten en Liberty BASIC zal dan ook een foutmelding geven dat ie een label mist. Echter, na de foutmelding wordt het programma niet gestopt. De uitvoer blijft in de oneindige lus.

Je kunt het programma afbreken met de toetsen Control en Break, maar je kunt ook het mainwin venster sluiten. Daarmee wordt dan de uitvoer afgebroken en wordt ook meteen het venster gesloten.

Een regel als “trapclose [quit]” zijn stringcommando’s. Ze worden niet als sleutelwoorden beschouwt.

Typ onderstaande code onder het commando WAIT.

```
[quit]
       CLOSE #w
       END
```

Omdat er verder geen code meer staat, is het END commando niet nodig. Maar in de meeste situaties hebben we het END commando wel in de [quit] label nodig.
Het is even wennen waar het END commando moet staan, vooral als we een subroutine zouden gebruiken in plaats van een label.

Onderstaande code geeft een voorbeeld hoe je een venster maakt met een subroutine om het venster te sluiten.

```
OPEN "WindowTest" FOR WINDOW AS #w
#w "trapclose Quit"
WAIT
SUB Quit handle$
    CLOSE #w
    END
END SUB
```

Er is een groot verschil tussen het gebruik van labels en subroutines.

- Elke subroutine heeft een handle$ parameter. Het mag ook anders worden genoemd, maar de naam handle$ vind ik wat gebruikelijker om het verschil te kunnen zien tussen actie subroutines en gewone subroutines.
- De Quit subroutine moet een END commando hebben. Het END commando in de label had bijvoorbeeld niet gehoeven, maar we kunnen in de subroutine niet het END commando vergeten. Zonder het commando END zal Liberty BASIC nog steeds niet correct het programma afsluiten.
- Het END commando kan in de Windows programmacode ook niet als apart los commando worden gebruikt. Zo zal er geen END worden gevonden als je het onder de END SUB zou plaatsen.

Omdat er een WAIT commando staat en daarmee gewacht wordt op actie, zoals de “trapclose”, zou je denken dat alle andere code dat je voor de WAIT plaatst, elke keer oneindig uitgevoerd wordt. Typ maar eens een PRINT regel met een tekst erachter en voer het programma uit.

Wat je niet zou verwachten, is dat het PRINT commando maar één keer uitgevoerd wordt. Het venster wordt wel telkens uitgevoerd, maar het reageert alleen op de acties. Gewone code wordt niet herhaald.

11b. Soorten vensters

Er zijn in Liberty BASIC verschillende soorten vensters. In andere programmeertalen moeten deze soorten ingesteld worden, maar in Liberty BASIC zijn ze al klaargemaakt. Je hoeft ze alleen maar op te geven in het OPEN commando.

Hieronder is een tabel met verschillende vensters met de omschrijving.

Soort venster	**Omschrijving**
window	Opent een standaard venster.
window_nf	Opent een venster zonder een frame. Het venster kan dan niet van grootte worden veranderd.
window_popup	Opent een venster zonder een titelbalk. Tot aan Liberty BASIC versie 2 werd een popup venster als een normaal venster ondersteund. Na versie 2 is het een deelvenster geworden. Het kan nog steeds als een normaal venster werken, maar Liberty BASIC geeft geen ondersteuning meer aan het venster. Dit betekent dat sommige controls en/of api controls niet of nauwelijks goed werken. Een popup venster wordt tegenwoordig als een panel venster gebruikt voor bijvoorbeeld het maken van werkbalken en tabstrips.
dialog	Opent een dialoogvenster.
dialog_modal	Opent een modaal dialoogvenster. Dit venster moet eerst worden gesloten voordat een ander venster de focus kan krijgen.
dialog_nf	Opent een dialoogvenster zonder een frame. Ook dit venster kan niet van grootte veranderen.
dialog_nf_modal	Opent een modaal dialoogvenster zonder een frame. Dit venster kan ook niet van grootte veranderen.
dialog_fs	Opent een dialoogvenster op grootte van het scherm.
dialog_nf_ns	Opent een dialoogvenster op grootte van het scherm zonder een frame. Dit venster kan niet van grootte veranderen.
dialog_popup	Hetzelfde als window_popup, maar nu als een dialoog deelvenster. De omschrijving bij window_popup geldt voor een dialog_popup hetzelfde. Ook dit venster wordt niet meer als een gewoon venster ondersteund.
graphics	Opent een venster met een grafische clientframe. Dat er echt een grafisch venster geopend wordt is niet juist. Er is een parent (ouder) venster waar het grafische clientframe in zit. Het is mogelijk om de ouder handle van het venster op te vragen.
graphics_fs	Opent een venster op grootte van het scherm. De clientframe (grafische deel) gaat mee.
graphics_nsb	Opent een venster met een grafische clientframe zonder schuifbalken.
graphics_fs_nsb	Opent een venster met een grafische clientframe op grootte van het scherm.
graphics_nf_nsb	Opent een venster met een grafische client zonder een frame en zonder schuifbalken. Dit venster kan niet van grootte veranderen.

text	Opent een venster met een tekst clientframe.
text_fs	Opent een venster met een tekst clientframe op grootte van het scherm.
text_nsb	Opent een venster met een tekst clientframe zonder schuifbalken.
text_nsb_ins	Opent een venster zonder schuifbalken met een tekst editor.

Al deze ingebouwde vensters zijn niet compleet. Zo is het bijvoorbeeld niet mogelijk een window_ns te geven voor een volledige schermgrootte, terwijl dat wel mogelijk is bij een venster met een grafische client.

Dat we echte grafische vensters en tekstvensters in Windows hebben, is niet juist. Het zijn gewoon vensters met een grafische client of een tekst client. Dit noemen ze ook wel de *clientwindow*. Elk ouder, de parent, kan via de client worden opgevraagd. Om het voor de BASIC gebruiker niet moeilijk te maken, zijn er verschillende vensters met een client ontwikkeld. Een control die het hele frame in de client in beslag neemt.

Normaal kun je ook tekenen op een gewoon venster. In Liberty BASIC moeten we een grafisch client gebruiken om te kunnen tekenen.

11c. Venster types

Vensters kunnen ook ingesteld worden met het commando STYLEBITS. Je hebt dan je eigen verantwoordelijkheid hoe een venster eruit mag zien.

De syntax is:

stylebits #<handle>, <bits toevoegen>, <bits verwijderen>, <extra bits toevoegen>, <extra bits verwijderen>

Het commando moet voor het openen van een venster komen te staan.

Het commando werkt ook bij controls. Onderstaand tabel laat de bits constanten zien. Sommige functioneren ook bij controls.
De meeste bits werken niet bij een tekstvenster, omdat het venster native is.

_WS_BORDER	Een venster dat wel of niet een dunne lijn border heeft.
_WS_CAPTION	Een venster met wel of niet een titelbalk inclusief met de _WS_BORDER stijl.
_WS_HSCROLL	Een venster met wel of niet een horizontale schuifbalk.
_WS_MAXIMIZE	Een venster dat wel of niet gemaximaliseerd is.
_WS_MAXIMIZEBOX	Een venster dat wel of niet een knop heeft om te maximaliseren.
_WS_MINIMIZE	Een venster dat wel of niet geminimaliseerd is. Zelfde als de _WS_ICONIC constante.
_WS_MINIMIZEBOX	Een venster dat wel of niet een knop heeft om te minimaliseren.
_WS_VSCROLL	Een venster met wel of niet een verticale schuifbalk.

De functies van de constanten verschillen van parametervorm. Het eerste argument is om toe te voegen. Bijvoorbeeld een _WS_MAXIMIZE zal een venster instellen met de grootte van het scherm, terwijl dezelfde constante als tweede argument juist voor zorgt dat het venster niet de grootte van het scherm krijgt.

Er kunnen meerdere constanten in één argument samen worden gebruikt. Omdat de constanten bitgewijs zijn, moeten ze met een OR bit operator worden opgegeven.

Zo zal een instelling als eerste argument _WS_HSCROLL OR _WS_VSCROLL voor zorgen dat het venster de horizontale schuifbalk en de verticale schuifbalk krijgt. Als tweede argument zal het venster geen van beide schuifbalken krijgen.

Beide argumenten kunnen niet hetzelfde krijgen. Als er niets ingesteld moet worden, maar wel wat uitgezet moet worden, dan kan onderstaand voorbeeld worden gebruikt. Het kan ook beide, als ze maar beide verschillende constanten hebben.

```
STYLEBITS #handle, 0, _WS_MINIMIZEBOX OR _WS_MAXIMIZEBOX, 0, 0
```

Het venster met gegeven handle #handle zal alleen de sluitknop hebben. De andere knoppen voor minimaliseren en maximaliseren zijn uitgezet.

Het volgende voorbeeld schakelt de knop in om te minimaliseren, maar schakelt de knop uit om te maximaliseren.

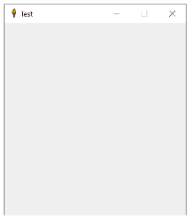
```
STYLEBITS #handle, _WS_MINIMIZEBOX, _WS_MAXIMIZEBOX, 0, 0
```

De laatste twee argumenten zijn extra bits en geven extra uitbreiding aan het venster. Zo zal de constante _WS_EX_CLIENTEDGE voor clientranden zorgen en zal de constante _WS_EX_TOOLWINDOW voor een tool-venster zorgen. De laatste twee argumenten werken verder op dezelfde manier as de eerste twee argumenten. Het derde argument zal het inschakelen en het vierde argument zal het uitschakelen.

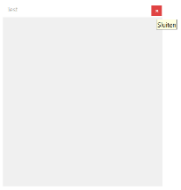
Wanneer niets gegeven wordt, een nul-waarde, zal er niets ingeschakeld of uitgeschakeld worden. Wordt STYLE-BITS niet gebruikt of zijn alle argumenten nul dan zal er een standaard venster worden gebruikt, inclusief het soort venster dat je gebruikt.

Onderstaande afbeeldingen geven voorbeelden wat STYLEBITS voor invloed heeft.

```
stylebits #w, _WS_MINIMIZEBOX, _WS_MAXIMIZEBOX, 0, 0
open "Test" for window as #w
#w "trapclose Quit"
wait

sub Quit handle$
    close #w
    end
end sub
```



```
stylebits #w, 0, _WS_MINIMIZEBOX or _WS_MAXIMIZEBOX, _WS_EX_TOOLWINDOW, 0
open "Test" for window as #w
#w "trapclose Quit"
wait

sub Quit handle$
    close #w
    end
end sub
```



Zoals je de uitvoer ziet van beide programma's, worden de knoppen en het venstertype door STYLEBITS be-paald. Merk op dat in de tweede afbeelding geen Liberty BASIC icoontje is. Toolvensters hebben nooit een icoon. Bovendien zie je ook dat de twee knoppen uitgeschakeld zijn, maar wat zal de uitvoer zijn als we ze inschakelen?

Wijzig eens de twee constanten en plaats ze als eerste argument. Maak het tweede argument nul. Wat gebeurt er?

Probeer eens onderstaande code uit en zie de uitvoer.

```
STYLEBITS #w, _WS_MINIMIZEBOX OR _WS_MAXIMIZEBOX, 0, _WS_EX_TOOLWINDOW, 0
OPEN "Test" FOR WINDOW AS #w
#w "trapclose Quit"
WAIT

SUB Quit handle$
    CLOSE #w
    END
END SUB
```

Ook al worden nu de twee knoppen ingeschakeld, je blijft het tweede venster zien. De constanten van het eerste argument worden genegeerd, omdat een toolvenster nooit deze knoppen heeft.

Merk ook op dat een toolvenster standaard geen randen heeft. Probeer eens onderstaande STYLEBITS.

```
STYLEBITS #w, _WS_MINIMIZEBOX OR _WS_MAXIMIZEBOX, 0, _WS_EX_CLIENTEDGE OR WS_EX_TOOLWINDOW, 0
```

Als je het nu uitvoert, zul je de client vensterranden zien, maar ook de titelbalk zal randen hebben.

Bovenstaande voorbeelden komen niet uit de Help vandaan. Deze komen met de tekst helemaal bij mij vandaan.

Liberty BASIC kent nog veel meer constanten die je met STYLEBITS in- en uit kunt schakelen. Zoek op internet eens naar Windows constanten voor meer informatie. Bovenstaand tabel laat niet alle constanten zien die er be-staan.

Bekijk eens de website via deze link: [Window Constants - Win32 apps | Microsoft Learn]

Hier zul je tal van constanten vinden. Je kunt de constanten uitproberen met het PRINT commando om te zien of zo'n constante ook in Liberty BASIC bestaat. Het is namelijk zo dat je zelf geen constanten kunt aanmaken met een underscore. Zulke variabelen zijn niet toegestaan. De bekende constanten zijn voorgedefinieerd.

Andere constanten die nog niet getoond zijn in het tabel, zijn de control constanten. Ook controls kunnen een stijl krijgen via een STYLEBITS regel. In het volgende hoofdstuk meer daarover.

11d. De voorgedefinieerde venstervariabelen

Liberty BASIC heeft variabelen om de positie en de grootte van het venster te bepalen. Deze variabelen zijn geen gewone variabelen. Net als de constanten zijn deze variabelen ook al klaargemaakt.

De venstervariabelen zijn *directives*. Je kunt ze ook gewoon variabelen noemen. Waarom ik ze zo noem, komt omdat deze variabelen voor een functie zorgen. Gelijk na het toekennen van een waarde wordt er voor actie gezorgd. Dat is niet alleen bij vensters zo. Ook bijvoorbeeld een PRINTERDIALOG (niet in het boek opgenomen) heeft voorgedefinieerde variabelen die je niet voor andere doeleinden kunt gebruiken.

Onderstaande directive variabelen stellen een positie en een grootte in voor een venster.

```
UpperLeftX = 100
UpperLeftY = 100
WindowWidth = 800
WindowHeight = 600
```

Zoals eerder gezegd kun je deze variabelen niet voor andere programmeringen gebruiken. Meteen na het openen van een venster, zullen deze ingestelde waarden opgenomen worden. Het venster zal dan gepositioneerd worden op 100, 100 met een grootte van 800 x 600.

Wanneer elk venster die daarna komt geopend wordt, zal ook weer deze instellingen worden gebruikt. Elk venster kan zijn eigen instellingen hebben, maar houd er wel rekening mee dat je dan de instellingen van het vorige venster kwijt bent.

Twee andere directives bepalen de grootte van het scherm: **DisplayWidth** en **DisplayHeight**. Handig om een grootte op te kunnen geven en ervoor te zorgen dat de grootte niet buiten het scherm valt. Dit is echter exclusief de positie die ingesteld wordt. Zeg je bijvoorbeeld **WindowWidth = DisplayWidth** dan zal aan de rechterkant van het venster een deel buiten het scherm komen als je een positie instelt die groter is dan nul.
Een oplossing is om de WindowWidth te berekenen door de positie af te trekken van DisplayWidth, zoals hieronder.

```
WindowWidth = DisplayWidth – UpperLeftX
```

Deze klaargemaakte variabelen laat ik in het boek niet in hoofdletters. Het zijn namelijk geen commando's en ook geen functies. Bovendien moeten ze ook zo precies getypt worden. Gebruik ze niet als kleine letters en ook niet als hoofdletters. Het venster wordt anders niet met deze waarden ingesteld en bovendien krijg je een *similar variables* waarschuwing van Liberty BASIC.

In het volgende hoofdstuk zul je zien dat deze variabelen ook nut hebben om controls op je venster uit te lijnen. Je kunt ze ook docken. Hoe dat werkt, zul je in het volgende hoofdstuk zien.

11e. Vensters ontwikkelen

Alleen maar kale vensters met wat instellingen met STYLEBITS en de variabelen voor de positie en de grootte is natuurlijk niet voldoende om in Windows te programmeren. Vensters moeten we aankleden met ander soort vensters. Deze vensters worden controls of besturingselementen genoemd.

Liberty BASIC beschikt niet over alle besturingselementen. Alleen de belangrijkste en meest gebruikte controls zijn als sleutelwoorden in Liberty BASIC aanwezig. Liberty BASIC heeft ook geen visuele designer. Alles moeten we met de code programmeren om goed bestuurbare vensters te maken.

Er bestaat wel een *freeform designer*. Hoewel dat wat gemakkelijker maakt om de vensters te maken, geeft het programma niet de goede functionaliteit. Code dat automatisch gegenereerd wordt doet meestal niet ten goede. Je kunt het beter zelf doen, zodat je alles in je eigen hand hebt.

Onderstaande schets geeft aan hoe de volgorde is bij het ontwikkelen, in Liberty BASIC programmeren, van vensters.

```
UpperLeftX = PosX
UpperLeftY = PosY
WindowWidth = breedte
WindowHeight = hoogte
'optioneel: hier komt de GUI, de controls die je nodig hebt
'optioneel: eventueel met STYLEBITS je venster instellen
OPEN titel$ FOR soortWindow AS #vensterHandle       'i.p.v. soortWindow, gebruik sleutelwoord
                                                    'zoals WINDOW
#vensterHandle "trapclose Quit"                     'eventueel kan ook [Quit]
'optioneel: schrijf hier je initialisatiecode
WAIT

SUB Quit handle$                                    'kan ook zijn [Quit] i.p.v. een SUB
       'optioneel: schrijf hier code om onderdelen te sluiten, zoals bestanden
       CLOSE #vensterHandle                         'kan ook handle$ zijn (vergeet niet het
                                                    '# teken)
       END                                          'Verplicht! Vergeet niet het END commando
END SUB
```

Je mag meer dan één venster open hebben, maar gebruik alsnog maar één WAIT commando. Het WAIT commando zal alle acties van de vensters afhandelen. Later meer over gebruik van meer vensters.

Controls die we meestal gebruiken zijn voor informatie, invoer en antwoord doorgeven met een klik. In het hoofdstuk Uitdrukkingen had ik het al over tekst en wat de bedoeling is met tekst. Tekst is ook wat de computer ons wilt vragen door het met een actie te beantwoorden.

We hebben steeds het mainwin venster gebruikt voor invoer en uitvoer. Windows vensters geven ons een grafische weergave zodat we de venster uitvoer vriendelijker kunnen maken. Alleen maar iets invoeren en na wat regels code het pas uitvoeren kan leiden dat we de invoerregels alweer kwijt zijn. We kunnen de invoer en de uitvoer geen vaste plaats geven op de mainwin.

Toch zullen we de mainwin blijven gebruiken. Het venster is niet alleen belangrijk voor testuitvoer, maar is ook nog eens een ontsnapvenster. We weten dus nog niet alles van de mainwin.

11f. De commando's MAINWIN en NOMAINWIN

In het begin van het boek zie je een afbeelding van het mainwin venster. Elke keer als we een programma uitvoeren, verschijnt het mainwin venster. Dit zal je ook wel opgevallen zijn bij de voorbeelden waarmee Windows vensters worden geopend. Je krijgt twee vensters te zien. Het ene venster is de mainwin, het andere venster dat je geopend hebt. Maar waarom?

Het mainwin venster gebruik je om met het PRINT commando gegevens te printen die je tijdens de uitvoer wilt zien. Een manier om te debuggen als het debugvenster (het lieveheersbeestje) niet naar wens is. Zoals eerder uitgelegd in het hoofdstuk Foutafhandelingen, kunnen we geen foutafhandelingscode debuggen. We kunnen wel in de code waar de fout wordt afgehandeld de waarden printen, zodat we eenvoudiger kunnen vinden wat er fout gaat.

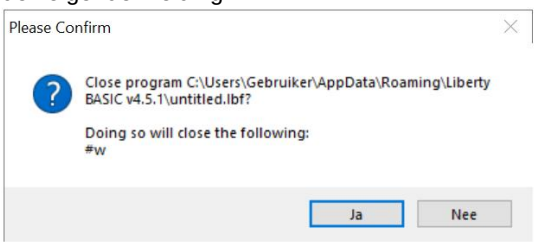
Een extra venster bij een Windows venster dat je gemaakt hebt, is niet altijd fijn. Je hebt dan de neiging om onderstaand commando als eerste commando in je programma te gebruiken.

```
NOMAINWIN
```

Natuurlijk, dit scheelt veel zodat je je kunt concentreren op het Windows venster. Maar er zijn omstandigheden om het commando nog even niet te gebruiken:

- Wanneer je uitvoer van waarden wilt zien tijdens de uitvoer.
- Om het programma te kunnen beëindigen als je per ongeluk een oneindige lus hebt.
- Als jouw programma nog niet af is. De eerste twee punten hebben ermee te maken. Het mainwin venster geeft bescherming aan de uitvoer van je programma. Zo het tweede punt zegt, kun je alleen uit je programma komen via het mainwin venster als jouw programma 'vast' zit.

Sluit je eerst je Windows venster, dan zie je dat deze goed gesloten wordt. Het mainwin venster blijft echter aanwezig. Deze moet je daarna zelf sluiten.
Doe je dit andersom, dus eerst het mainwin venster sluiten, dan gaat Liberty BASIC je op de vingers tikken met de volgende melding:



Het zegt iets vervelends van: 'Ach, nou moet ik weer het Windows venster gaan sluiten.'
Liberty BASIC vindt kennelijk deze volgorde van afsluiten niet fijn.
Het is nog vreemder als je 'Nee' kiest. Je annuleert dan de sluitknop van de mainwin waar je eerst op klikte. Het Windows venster, die even weg was, komt dan weer tevoorschijn.
Kies je 'Ja', dan wordt het Windows venster gesloten, maar ook het mainwin venster. Alles wordt direct afgesloten en het programma is beëindigd.

Het is vervelend als Liberty BASIC deze meldingen geeft. Het kan namelijk ook gebeuren als je meer vensters kruislings gaat sluiten of de bestanden vergeet te sluiten. Hoe vervelend deze melding ook is, het is niet schadelijk voor je programma. Liberty BASIC wil je alleen een mededeling geven dat hij iets niet leuk vindt. Dat hij met zo'n uitgebreid venster komt, geeft irritatie. Vooral als je veel met vensters aan het oefenen bent en bijvoorbeeld het END commando vergeet.
Misschien kunnen we de mededeling ook van een andere kant bekijken. Liberty BASIC geeft netjes aan welke handle verkeerd of niet goed gesloten is. Helaas zijn deze meldingen niet te onderscheppen. Een ON ERROR GOTO werkt hierop niet, want als er iets vergeten wordt dat met het sluiten van vensters en bestanden te maken heeft, dan komt Liberty BASIC eerder met de melding dan een foutafhandeling kan starten. Bovendien zijn deze meldingen eigenlijk geen foutmeldingen, maar hints. Een hint om iets wat je vergeten bent.

Als je denkt dat je klaar bent met je programma en dat het ook naar behoren goed werkt, dan heb je het mainwin venster niet meer nodig. Je kunt dan gerust het voorgaande commando NOMAINWIN in je programma opnemen.

Een ander commando, MAINWIN, lijkt als de tegenhanger van NOMAINWIN. Echter is de functie van MAINWIN niet om het mainwin venster aan te zetten. Het is de bedoeling om het mainwin venster een bepaalde grootte te geven.

Syntax:

MAINWIN <kolommen> <rijen>

Als het commando NOMAINWIN in het programma staat, zal het MAINWIN commando niet werken. Je kunt het venster alleen een grootte geven als het venster ook aanwezig is.

Het commando MAINWIN hoeft niet speciaal bovenaan het programma te staan, maar dat is ook met NOMAINWIN het geval. Ook al typ je het in de Quit subroutine, dan zal het commando direct uitgevoerd worden, ook al klik je niet eens op de sluit knop.

Dit geeft de indruk dat NOMAINWIN en MAINWIN geen gewone commando's zijn. Het zijn compiler commando's of ook wel directives genoemd. Deze commando's worden als eerste gevonden, ook al staan ze niet aan het begin van het programma.

Probeer eens ermee te experimenteren. Kijk eens wat er gebeurt als je het uitvoert.

11g. Het maken van een venster template

Elke keer wanneer je een nieuw Windows programma wilt schrijven, moet je telkens een venster openen. Het venster openen gebeurt kaal in elk programma op dezelfde manier.

Om tijd te besparen, kun je een venster kaal programmeren, dus nog met niks erin. Behalve de trapclose die altijd nodig is.

Neem onderstaande code over in een lege Liberty BASIC editor.

```
OPEN "" FOR WINDOW AS #w
#w "trapclose Quit"
```

```
WAIT

SUB Quit handle$
       CLOSE #handle$
       END
END SUB
```

Bewaar het programma in een bekende en makkelijke terug te vinden map.

Maak een nieuw programma aan in het menu item New en kies een normaal Source programma.
Ga naar het menu File en kies het menu item Insert.
Kies de map waar het voorgaande programma in staat. Het programma wordt nu niet als een apart programma geopend, maar wordt ingevoegd in het nieuwe programma.

Deze soort programma's kun je *templates* of *sjablonen* noemen. Later in een apart hoofdstuk kom ik erop terug en leg ik ook uit hoe je modules kunt maken.

Nu kun je elke keer, wanneer je een venster nodig hebt, deze template invoegen.

Tip!
Wanneer je de template meerdere keren invoegt voor meer vensters, vergeet dan niet de Quit eventnaam te wijzigen, tenzij je elke trapclose laat uitvoeren met dezelfde Quit. Dat is mogelijk, omdat een venster afgesloten wordt met de meegegeven handle$. Als je niet de handle$ gebruikt maar het venster sluit met de handle van het venster, dan moet je meer van deze subroutines schrijven.

12. De GUI controls

12a. Tekst weergeven op een venster

We hebben tot nu toe tekst geprint op het mainwin venster. Op een Window venster gaat dat anders. We gebruiken een tekst control om tekst weer te geven.
Tekst op een form (venster) worden labels genoemd. Niet verwarren met labels in de code om erheen te branchen. Tekst labels, zoals ze in andere programmeertalen genoemd worden, zijn in Liberty BASIC statictext controls. Labels zijn altijd statisch. Ze kunnen doormiddel van invoer niet gewijzigd worden en ook niet opgevraagd worden. Hoewel het opvragen van wat voor tekst er staat in andere programmeertalen wel mogelijk is, maar in deze BASIC taal niet.

Syntax:

STATICTEXT #<windowhandle>[.<labelnaam>], <titel$>, <x>, <y>, <breedte>, <hoogte>

Omdat een statische tekst alleen weergeeft maar verder niets meer doet, is de labelnaam niet verplicht. Indien je het toch nodig denkt te hebben, bijvoorbeeld als je tijdens de uitvoer de tekst wilt wijzigen, geef je het een duidelijke naam achter de punt.

We mogen de titel of een omschrijving op twee manieren instellen. Direct opgeven in een STATICTEXT control of pas later, bijvoorbeeld na het openen van het venster of pas wanneer er op een knop wordt geklikt.

Een STATICTEXT heeft geen events voor acties. Er zijn meer controls die dat niet hebben. Er zijn wel eigenschappen die je kunt gebruiken. Je kunt ook de opmaak van de control wijzigen met de STYLEBITS.

- _SS_CENTER stelt een simpele rechthoek in en centreert de tekst in de rechthoek.
- _SS_RIGHT stelt een simpele rechthoek in en lijnt de tekst rechts uit in de rechthoek.

```
WindowWidth = 800
WindowHeight = 600
STYLEBITS #w.lblTekst, _SS_RIGHT, 0, 0, 0
STATICTEXT #w.lblTekst, "Rechts uitgelijnd", 10, 10, 400, 35
OPEN "Statictext" FOR WINDOW AS #w
```

```
#w "trapclose Quit"

WAIT

SUB Quit handle$
    CLOSE #handle$
    END
END SUB
```

Uiteraard is de rechthoek niet te zien, maar deze onzichtbare rechthoek zorgt ervoor dat de tekst goed uitgelijnd wordt.
Wil je goed kunnen zien dat de tekst rechts uitgelijnd wordt, wijzig dan de STYLEBITS met onderstaande regel.

```
STYLEBITS #w.lblTekst, _SS_RIGHT OR _WS_BORDER, 0, 0, 0
```

De constante _WS_BORDER zag je al eerder in het tabel. Sommige venster constanten werken ook op controls. Controls zijn, zoals eerder gezegd, ook vensters. Het venster die je opent is altijd de ouder van de control venstertjes.

Goed te onthouden!
De STATICTEXT heeft niet speciaal een controlnaam nodig. Zoals eerder verteld, is deze optioneel. Echter, met gebruik van een STYLEBITS die op de volgende STATICTEXT moet werken, moet je wel een naam meegeven. De STYLEBITS zal anders niet werken.

Haal maar eens de naam lblTekst met de punt weg van zowel de STYLEBITS als van de STATICTEXT. Start het programma opnieuw en je zal zien dat de tekst niet rechts uitgelijnd wordt.

12b. Invoer, uitvoer en de knop

Met de STATICTEXT wordt een string op het venster getoond. Hier spreek ik niet van *printen*, omdat we gegevens alleen printen op de mainwin.

Alleen maar tekst op het venster zien is niet voldoende. We kunnen veel meer doen op het venster. In plaats van gegevens tonen, kunnen we ook gegevens invoeren. Dat werkt niet met een INPUT commando, maar met een TEXTBOX control.

Syntax:

TEXTBOX #<windowhandle>.<textboxnaam>, <x>, <y>, <breedte>, <hoogte>

Een TEXTBOX control heeft geen events (acties of gebeurtenissen). Anders dan in andere programmeertalen waar je een enter toets of een tab toets, voor het verlaten van de textbox, kunt afhandelen, is dat in Liberty BASIC helaas niet mogelijk. Om de ingevoerde gegevens op te kunnen vragen, hebben we een knop nodig.

Een knop heeft wel een event, namelijk om een klik met de linker muisknop af te handelen. Daarmee kunnen we invoer opvragen die in de tekstbox staat of andere acties uitvoeren, zoals het openen van een ander venster of het sluiten van een venster. Een trapclose actie zal niet altijd de standaardknop zijn om een venster te sluiten.

Syntax:

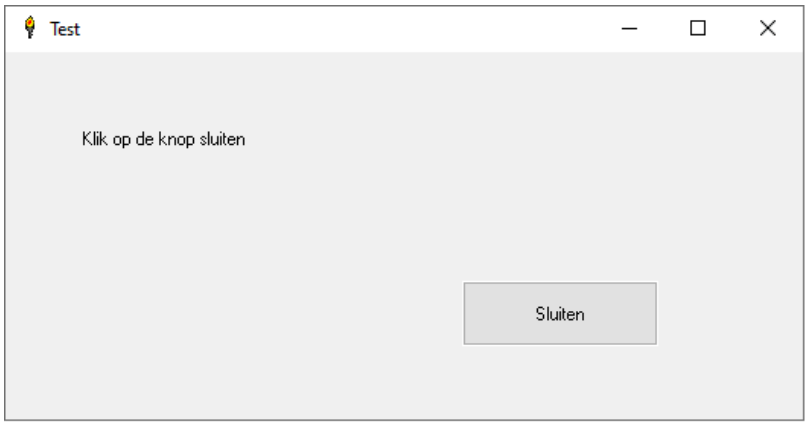
BUTTON #<windowhandle>.<buttonnaam>, <titel$>, <eventsubnaam>|<eventlabelnaam>, <hoekpositie>, <x>, <y> [,<breedte>, <hoogte>]

De laatste twee parameters, breedte en hoogte, zijn optioneel. Wanneer deze niet worden opgenomen, zal er een automatische lengte worden bepaald aan de hand van de tekstlengte van titel$. Spaties zijn ook tekens, dus extra spaties voor of achter de titel beïnvloedt ook de lengte.

Bekijk onderstaande voorbeeld met een afbeelding hoe de uitvoer eruit zal zien.

```
WindowWidth = 540
WindowHeight = 280
STATICTEXT #w, "Klik op de knop sluiten", 50, 50, 300, 35
BUTTON #w.btnClose, "Sluiten", Quit, UL, 300, 150
```

```
OPEN "Test" FOR WINDOW AS #w
#w "trapclose Quit"
WAIT

SUB Quit handle$
    CLOSE #w
    END
END SUB
```

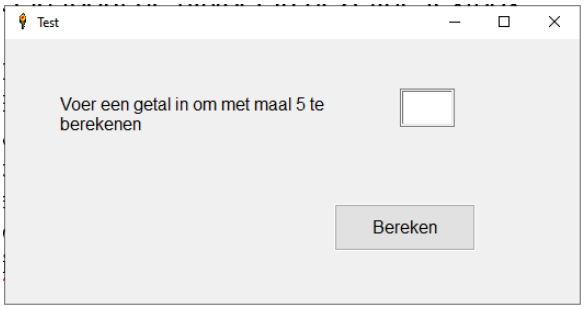


De lengte van de knop wordt automatisch ruim bepaald, zodat de tekst nooit strak in de knop zal staan. Probeer maar eens een andere tekst en je zult zien dat de ruimte op één of andere manier groter of kleiner wordt. Het zou kunnen zijn dat de ruimtes even lang zijn als de tekst die gewijzigd wordt. Hoe kleiner de tekst, hoe smaller de ruimtes worden. Hoe langer de tekst, hoe langer de ruimtes worden.

Knoppen kunnen zoals gezegd iets doen om bijvoorbeeld voor ander uitvoer te zorgen. Omdat de tekstbox geen eigen event heeft, moet dat via een knop worden afgehandeld.

Onderstaand voorbeeld toont de uitvoer in dezelfde textbox.

```
WindowWidth = 540
WindowHeight = 280
STATICTEXT #w, "Voer een getal in om met maal 5 te berekenen", 50, 50, 300, 35
TEXTBOX #w.txtGetal, 360, 45, 50, 35
BUTTON #w.btnClose, "Bereken", btnBereken, UL, 300, 150
OPEN "Test" FOR WINDOW AS #w
#w "trapclose Quit"
WAIT

SUB btnBereken handle$
    #w.txtGetal "!contents? getal"
    #w.txtGetal ""; getal * 5
END SUB

SUB Quit handle$
    CLOSE #w
    END
END SUB
```



De contents eigenschap in de btnBereken subroutine retourneert de ingevoerde waarde. De naam *getal* is een variabele, maar moet in de string meegegeven worden. Gebruik geen variabele buiten de string met een puntkomma. De eigenschap heeft een variabele nodig. De variabele is dus een argument in de string. We zullen meer van zulke commando's en eigenschappen tegenkomen.

Het uitroepteken is nodig om Liberty BASIC te vertellen dat het om een commando of een eigenschap gaat. Zonder uitroepteken zal het als een tekst worden beschouwd en zal het in de textbox worden getoond, zoals de volgende regel doet.
Niet alle controls gebruiken een uitroepteken bij de commando's en eigenschappen. Als er geen tekstuitvoer door de control ondersteund wordt, zal het uitroepteken illegaal zijn en zal een commando of eigenschap niet werken.

Het vraagteken vertelt Liberty BASIC dat de eigenschap iets moet resulteren. Sommige controls hebben ook een contents zonder vraagteken om gegevens in de control te kunnen zetten. De textbox kent alleen maar een contents met een vraagteken. Zoals je in de volgende regel van bovenstaand voorbeeld ziet is er geen contents nodig om iets in de textbox te plaatsen.

Opdracht:

Breidt het programma uit met een tweede textbox om daar het resultaat in te zetten in plaats van in de textbox waar het getal ingevoerd wordt. Zorg voor voldoende ruimte zodat de knop rechtsonder kan blijven staan.

12c. Lettertype instellingen

Vaak is de uitvoer van tekst in de controls te klein. Het liefst zou de tekst een ander formaat moeten hebben om een duidelijk leesbaar venster te hebben.

In het venster en ook in de controls kan het lettertype, de *font*, gewijzigd worden.

De font is een commando die verschillende parameters heeft. Ook dit commando moet tussen aanhalingstekens staan inclusief de argumenten.

Syntax:

#<handlenaam[<.controlnaam>] “font <lettertype> [[size] <grootte>][<fontstyles>]]”

Je kunt bijvoorbeeld voor het venster een volgend lettertype instellen:

```
#w "font arial size 16 italic"
```

Omdat het woord *size* optioneel is, werkt het ook als: font arial 16 italic

Sommige fontnamen hebben spaties zoals ‘Courier New’. Stel deze in met een onderstrepingsteken ‘_’.
In sommige controls moet er rekening worden gehouden dat het uitroepteken nodig is. Denk eraan dat het om controls gaat waarmee tekst weergeven de prioriteit heeft.

Gebruik bovenstaand voorbeeld door een font na de trapclose event toe te voegen. Merk op na het uitvoeren dat niet alleen het venster de font krijgt, maar ook alle andere controls. De reden is dat het venster de parent (ouder) is van alle controls. Wat het venster krijgt, kunnen de controls van erven. Dit geldt niet voor alle instellingen, maar wel voor het font.

In de font zijn de volgende fontstyle instellingen bekend.

Bold	Tekst wordt **vet** gedrukt.
Italic	Tekst is *schuinschrift*.
Underscore	Tekst wordt onderstreept.
Strikeout	Tekst wordt ~~doorgehaald~~.

Deze instellingen kun je samen gebruiken door ze te scheiden met een spatie. De instellingen Bold en Italic zorgen samen voor een ***vet schuinschrift***.

Voorbeeld: #w “font courier_new 12 bold italic”

In de font string mogen hoofdletters en kleine letters doorelkaar gebruikt worden. De instelling bold en Bold is hetzelfde.

We zullen in de komende paragrafen en hoofdstukken de font weer tegenkomen.

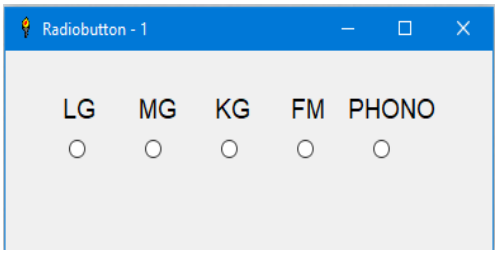
12d. Radiobuttons, checkboxen en groupboxen

Naast invoer, uitvoer en actie uitvoeren in een klik op een knop, is het ook mogelijk om een keuze te maken uit één of meer opties. Een keuze kunnen we maken op twee manieren. Eén keuze maken uit meer opties waarmee er maar één aangezet kan worden en meer keuzes waarmee we er meer aan kunnen zetten door ze aan te vinken.

De opties waarmee er maar één kan worden aangezet zijn de radiobuttons. De andere zijn de checkboxen waarmee we er meer aan kunnen vinken.

Als we in een programma de gebruiker één keuze willen laten maken uit een aantal opties, dan zijn radiobuttons een goed hulpmiddel. De naam van deze knoppen is afgeleid van de rij schakelaars welke we vroeger in een radio konden vinden. Deze schakelaars waren mechanisch zo gekoppeld dat slechts één ingedrukt kon zijn.

Op die wijze kon worden gekozen tussen de frequentie bereiken en de pick-up. Als we dit als voorbeeld in Liberty Basic maken dan kan dat er zo uitzien:



Het wezen van deze knoppen is dus dat er slechts één van het stel kan worden geactiveerd. De radiobuttons kunnen alleen in een window venster staan, niet in het mainwin venster. Ze worden gedefinieerd voordat het venster, waar ze in komen te staan, wordt geopend. De standaard definitie van de radiobutton is als volgt:

Syntax:

RADIOBUTTON #<windowhandle>.<radiobuttonnaam>, <titel$>, <eventsub set>|<eventlabel set>, <eventsub reset>|<eventlabel reset>, <x>, <y>, <breedte>, <hoogte>

De parameters werken als volgt:

#<windowhandle>.<radiobuttonnaam>
De windowhandle is de handle voor het venster. Laten we zeggen dat het venster met de handle #main wordt geopend. Voor de radiobutton komt daar een punt en een naam van de button achter. Laten we zeggen dat #main.keuze1 de handle is voor de eerste radiobutton en #main.keuze2 voor de tweede radiobutton. Wat we met deze handle kunnen doen zien we verder in deze paragraaf.

<titel$>
De titel$ is een naam welke we naast de knop te zien krijgen. De titel moet tussen aanhalingstekens net zoals bij de statictext en de button, of gebruik een stringvariabele. Liberty Basic zet deze tekst rechts van de knop.

<eventsub set> of <eventlabel set>
Dit is de naam van de sub welke wordt geactiveerd als op een radiobutton wordt geklikt – of eventueel een label waar naartoe wordt gegaan.

<eventsub reset> of <eventlabel reset>
In Liberty Basic 4.5.1 is dit een “dummy” argument omdat een radiobutton niet kan worden gereset door er op te klikken. Wat op deze plaats komt te staan wordt door de radiobutton zelf niet gebruikt. Liberty Basic vereist wel dat er een eventsub of eventlabel wordt ingevuld. Maar aangezien het wel mogelijk is om de radiobuttons te resetten elders in het programma, is het nuttig om de daarvoor gebruikte reset dan hier te noemen.

<x>
Dit is de x coördinaat van de linkerbovenhoek van de button.

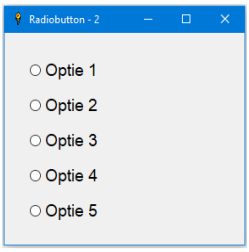
<y>
Dit is de Y coördinaat van de linkerbovenhoek van de button.

<breedte>
Dit is de breedte van de ruimte voor de radiobutton. In deze ruimte moet ook de tekst van titel$ passen. De lengte is inclusief het rondje links van de tekst.

<hoogte>
Dit is de hoogte van de ruimte voor de radiobutton. Vaak is een hoogte van 20 punten genoeg, maar als er voor het venster of van de radiobutton zelf een groter font is opgegeven dan kan de ruimte daarvoor te klein zijn en is een grotere hoogte nodig.

We kunnen niet de radiobuttons zo plaatsen zoals in de vorige afbeelding staat. Radiobuttons staan altijd horizontaal, dus de tekst rechts van het rondje.

Bij het opgeven van <x>, <y>, <breedte>, <hoogte> moet rekening worden gehouden met de voor deze zaken benodigde ruimte. Als er een rij knoppen onder elkaar wordt gedefinieerd moeten we er voor zorgen dat de afstand in Y-richting minimaal gelijk is aan de hoogte van de voor de radiobutton opgegeven ruimte.
Laten we eens zien wat we met het bovenstaande kunnen doen:



We zien hier 5 keuzes in een venster. Het programma voor dit venster ziet er zo uit:

```
CALL MaakWindow
'-------------------------------------------------------------------------------
SUB MaakWindow
    WindowWidth  = 300
    WindowHeight = 280
    UpperLeftX = 150
    UpperLeftY = 150

    RADIOBUTTON #main.rad1, "Optie 1", radioSet1, radioReset1, 30,  30, 90, 25
    RADIOBUTTON #main.rad2, "Optie 2", radioSet1, radioReset1, 30,  70, 90, 25
    RADIOBUTTON #main.rad3, "Optie 3", radioSet1, radioReset1, 30, 110, 90, 25
    RADIOBUTTON #main.rad4, "Optie 4", radioSet1, radioReset1, 30, 150, 90, 25
    RADIOBUTTON #main.rad5, "Optie 5", radioSet1, radioReset1, 30, 190, 90, 25

    OPEN "Radiobutton - 2" FOR WINDOW AS #main
    #main "trapclose Sluit"
    #main "font arial 14"
    WAIT
END SUB

SUB Sluit handle$
    CLOSE #main
    END
END SUB

SUB radioSet1 handle$           'afhandeling radiobuttons
    keus = 0
    Txt$ = handle$
    keus = VAL(RIGHT$(Txt$, 1))
```

```
    Cijfer(0) = keus
END SUB
```

We zien de definities voor de radiobuttons, maar ook de sub welke uiteindelijk het resultaat voor de als laatste aangeklikte button opgeeft. In dit geval wordt de waarde van het meest rechtse karakter van de handle – in dit geval een cijfer – omgezet tot het getal van dit cijfer en geplaatst in een array waardoor het elders te gebruiken is.

Zoals eerder gezegd kan slechts één radiobutton in een venster aan staan. Maar wat als we keuze voor 2 of meer zaken in een programma willen doen? Dan moeten we de buttons in een soort eigen venster plaatsen. Zo'n venster wordt een groupbox genoemd. Dat ziet er zo uit:

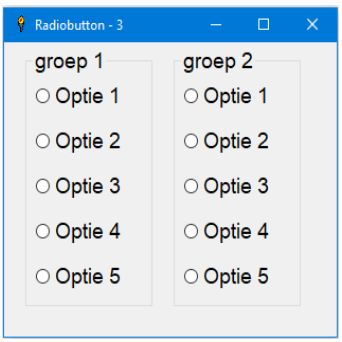
Syntax:

GROUPBOX #<windowhandle>.<groupboxnaam>, [<titel$>,] <x>, <y>, <breedte>, <hoogte>

De onderdelen van deze definitie zijn gelijk aan die van de radiobutton. We moeten er wel aan denken dat we de positie en de afmetingen zo kiezen dat er voldoende ruimte is voor de titel dat in de bovenrand van de groupbox komt te staan en voor de buttons en de labels daarbij. De titel is zoals je ziet optioneel.

Laten we eens een voorbeeld bekijken:

```
CALL MaakWindow
'-------------------------------------------------------------------------------
SUB MaakWindow
    WindowWidth  = 330
    WindowHeight = 300

    UpperLeftX = 150
    UpperLeftY = 150

    RADIOBUTTON #main.rad1, "Optie 1", radioSet1, radioReset1,  30,  35, 90, 25
    RADIOBUTTON #main.rad2, "Optie 2", radioSet1, radioReset1,  30,  75, 90, 25
    RADIOBUTTON #main.rad3, "Optie 3", radioSet1, radioReset1,  30, 115, 90, 25
    RADIOBUTTON #main.rad4, "Optie 4", radioSet1, radioReset1,  30, 155, 90, 25
    RADIOBUTTON #main.rad5, "Optie 5", radioSet1, radioReset1,  30, 195, 90, 25
    RADIOBUTTON #main.rdd1, "Optie 1", radioSet2, radioReset1, 170,  35, 90, 25
    RADIOBUTTON #main.rdd2, "Optie 2", radioSet2, radioReset1, 170,  75, 90, 25
    RADIOBUTTON #main.rdd3, "Optie 3", radioSet2, radioReset1, 170, 115, 90, 25
    RADIOBUTTON #main.rdd4, "Optie 4", radioSet2, radioReset1, 170, 155, 90, 25
    RADIOBUTTON #main.rdd5, "Optie 5", radioSet2, radioReset1, 170, 195, 90, 25

    GROUPBOX    #main.opt1, "groep 1",  20, 5, 120, 230
    GROUPBOX    #main.opt2, "groep 2", 160, 5, 120, 230

    OPEN "Radiobutton - 3" FOR WINDOW AS #main
    #main "trapclose Sluit"
    #main "font arial 14"
    WAIT
END SUB

SUB Sluit handle$
    CLOSE #main
    END
END SUB

'afhandeling radiobuttons
SUB radioSet1 handle$
    keus = 0
    Txt$ = handle$
    keus = VAL(RIGHT$(Txt$, 1))
    Cijfer(0) = keus
END SUB

'afhandeling radiobuttons
SUB radioSet2 handle$
    keus = 0
    Txt$ = handle$
    keus = VAL(RIGHT$(Txt$, 1))
```



```
    Cijfer(1) = keus
END SUB
```

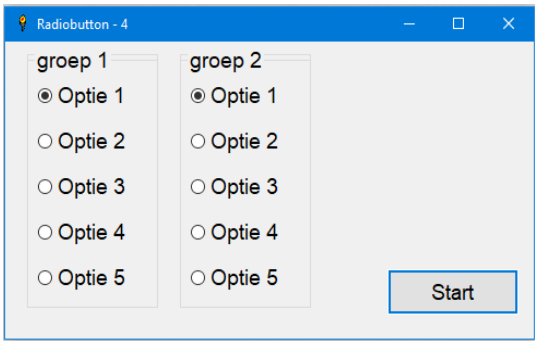
Wij kunnen nu in elke groep één radiobutton aangeklikt krijgen. Om er voor te zorgen dat variabele keus in de tweede groep duidelijk apart is van die in de eerste groep, is hier een tweede sub gemaakt onder de naam radioSet2 welke het resultaat op de volgende plaats van de array zet.

In Liberty BASIC hoeven we niet te vertellen welke controls bij de groupboxen horen. Zodra de rechthoek om de radiobuttons staat, is de groupbox automatisch de ouder (parent) van de radiobuttons.

Het is mogelijk om vooraf – per groupbox – een radio button aan te zetten. Voor een goede werking moeten er twee zaken worden geregeld. We willen dat de twee bovenste opties als basis worden aangezet.

1. Tussen het OPEN commando en het WAIT commando komen in dit voorbeeld twee nieuwe regels;
2. De startwaarde 1 moet in het begin van het programma worden toegekend aan de array variabelen Cijfer(0) en Cijfer(1).

Het voorbeeld ziet er dan zo uit:



Voor wie verbaasd is over de knop ‘start’; elke keer als je op een radiobutton klikt wordt de bijbehorende waarde bepaald en in het voorbeeld aan Cijfer(0) of Cijfer(1) toegekend. Verder niets. Het is mogelijk om daarna in de routine, welke de waarde bepaald, een verdere opdracht voor afhandeling te geven. In ons voorbeeld met twee groupboxen is dat niet handig. De bedoeling is dat de gebruiker van het programma eerst beide keuzes maakt. Nadat de gebruiker van het programma zijn of haar keuze heeft gemaakt zal dat op een of andere wijze aan het programma moeten worden doorgegeven opdat met de keuze zal worden gewerkt. In dit voorbeeld dus met de routine welke behoort bij de knop ‘Start’.

Stel dat na de afhandeling weer een nieuwe keuze moet worden gemaakt en daarna weer op ‘start’ worden gedrukt, dan is het mogelijk om de begin stand van de radiobuttons weer terug te plaatsen. Aan het eind van de afhandelingsroutine plaats je de twee ‘set’ opdrachten en zet je de gewenste begin waarden in array.

Als je zonder begin waarden start, en je wilt dat de radiobuttons na de afhandeling weer allemaal leeg zijn, dan is dat op bijna dezelfde wijze mogelijk als met het terug zetten van de beginwaarde van de buttons:

```
#main.rad1 "reset"
```

In dit geval is ‘reset’ de bovenste radiobutton in groep 1. Zijn er echter heel veel radiobuttons in een programma, dan kan het ook als volgt:
Stel, je hebt radiobuttons #main.statA tot en met #main.statZ. In variabele hulp$(0) is de laatst gekozen radiobutton te vinden in de vorm van de laatste letter (A t/m Z) van de knop. We schrijven dan het volgende:

```
SUB ResetRadioB             '
    txt$ = hulp$(0)
    Rset$ = "#main.rad" + txt$
    #Rset$ "reset"
    'hulp$(0) = ""
END SUB
```

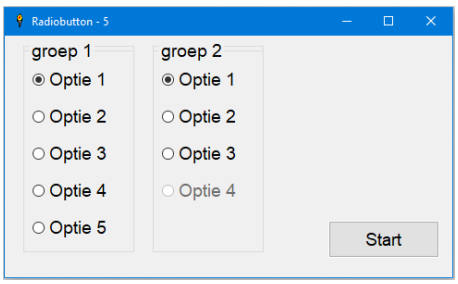
Op deze wijze resetten we dus alleen de gekozen knop door de laatste letter van de knop aan de variabele Rset$ toe te kennen. We doen daardoor de reset af in 6 regels i.p.v. 29 regels.

Merk op dat stringvariabelen gebruikt mogen worden als handles. Dit is noodzakelijk omdat het niet toegestaan is te beginnen met een expressie.

De radiobuttons hebben ook commando’s waarmee ze uitgezet worden en onzichtbaar worden gemaakt. Deze commando’s moeten tussen aanhalingstekens staan.

disable	Met dit commando wordt de radiobutton uitgezet.
enable	Met dit commando zet je de radiobutton weer aan.
hide	Met dit commando verberg je de radiobutton.

show	Met dit commando maak je de radiobutton weer zichtbaar.



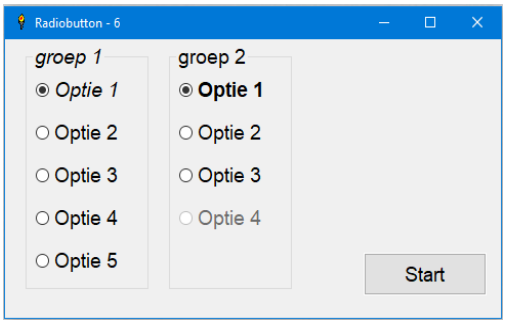
In deze afbeelding zijn de disable en hide commando's gebruikt.

In de voorbeelden Radiobutton – 2 t/m 5 staat de tekst bij de button rechts van de button. Dat is de plaats waar de tekst terecht komt. Maar in voorbeeld Radiobutton – 1 stonden de teksten toch boven de radiobuttons?
Correct, maar in dat voorbeeld werden de titels van de radiobuttons niet gebruikt. Als de tekst op een andere plaats moeten komen, dan moet er een andere manier zijn om de tekst te plaatsen.

```
RADIOBUTTON #main.rad1, "", radioSet1, radioReset1,  50, 60, 20, 20
RADIOBUTTON #main.rad2, "", radioSet1, radioReset1, 110, 60, 20, 20
RADIOBUTTON #main.rad3, "", radioSet1, radioReset1, 170, 60, 20, 20
RADIOBUTTON #main.rad4, "", radioSet1, radioReset1, 230, 60, 20, 20
RADIOBUTTON #main.rad5, "", radioSet1, radioReset1, 290, 60, 20, 20
STATICTEXT  #main.statA, "LG",    45, 30, 90, 22
STATICTEXT  #main.statB, "MG",   105, 30, 90, 22
STATICTEXT  #main.statC, "KG",   165, 30, 90, 22
STATICTEXT  #main.statD, "FM",   225, 30, 90, 22
STATICTEXT  #main.statE, "PHONO", 270, 30, 90, 22
```

In dit geval wordt de titel leeg gelaten ("") en wordt per button een statictext aangemaakt. En die kan op elke gewenste positie vanaf de radiobutton worden geplaatst. Merk op dat de breedte en de hoogte van de radiobutton in dit voorbeeld beide 20 zijn. We hoeven immers geen ruimte te reserveren voor de titel naast de button.
Als we echter gebruik maken van deze mogelijkheid en met disable of hide radiobuttons uit zetten, dan verandert daarmee de tekst niet meer zoals in voorbeeld 5. Maar ook de statictext kent de disable/enable en de hide/show commando's. De meeste controls hebben deze commando's.

In de bovenstaande voorbeelden zagen we ook weer font instellingen. Zoals eerder uitgelegd kunnen we voor alle vensters en controls de font gebruiken. Voor de radiobuttons gebruiken we bij het font commando geen uitroepteken.



Onderstaand programmadeel geeft deze afbeelding.

```
OPEN "Radiobutton - 6" FOR WINDOW AS #main
#main "trapclose Sluit"
#main "font arial 14"
#main.rad1 "set"
#main.rdd1 "set"
#main.gpb1, "!font arial 14 italic"
#main.rad1, "font arial 14 italic"
#main.rdd1, "font arial 14 bold"
#main.rdd4, "disable"
#main.rdd5, "hide"
WAIT
```

Zoals we kunnen zien werkt het commando ook voor de groupbox. Alleen let daarbij op het uitroepteken dat voor het woord font komt te staan.
Maar let op! #main, "font arial 14" moet nu direct onder de trapclose opdracht staan. Pas daarna mogen de fonts voor onderdelen van het venster worden gebruikt. Als #main, "font arial 14" onderaan net boven het WAIT commando zou staan, dan zou dit commando alle vorige fontinstellingen overschrijven en zouden alle teksten in het venster de font gebruiken van het venster. Zoals eerder gezegd erven de controls de font instellingen van het venster. Daarom moeten de instellingen van de controls na een eventuele instelling van het venster.

De radiobutton heeft nog meer commando's en eigenschappen.

```
#handle.ext, "value? result$"
```
De variabele result$ krijgt bij gebruik van deze eigenschap naar gelang de status van de radiobutton de inhoud "set" of "reset".

```
#handle.ext, "setfocus"
```

De radiobutton, of een ander control, krijgt hiermee de focus. Als we in het hiervoor geplaatste voorbeeld voor WAIT de volgende opdracht plaatsen:

```
#main.rdd3, "setfocus"
```

dan krijgt de derde radiobutton in groep 2 de focus. Dat betekent ook dat de radiobutton automatisch ingeschakeld wordt (een automatische 'set').

```
#handle.ext, "locate x y width height"
```

Met dit commando kan een control, zoals een radiobutton, een andere plaats krijgen. Maar dit werkt alleen in een venstertype "window". Het formaat past zich aan wat er als argumenten meegegeven worden en het gaat pas in na het gebruik van het "refresh" commando van het betreffende venster. Deze opdracht wordt meestal gebruikt om tijdens de uitvoer de aanpassing van de venstergrootte te wijzigen. Dat in een graphics venster of in een graphicbox geen locate commando gebruikt kan worden, komt omdat dan de foutmelding neerkomt op het refresh commando.
Het locate commando werkt niet op vensters. Dat kan op een andere manier. Daarover in het hoofdstuk Windows API.

De groupbox kent dezelfde commando's als de andere controls kennen. Eigenlijk hoef ik hier niet steeds op terug te komen, omdat elke control deze commando's en eigenschappen heeft.
Er is één verschil waar op gelet moet worden. In sommige controls moeten de commando's en eigenschappen beginnen met een uitroepteken, anders zal het als tekst worden beschouwd.

#<windowhandle>.<groupboxnaam> "een string"
Zet een nieuwe naam aan de bovenzijde van de groepbox.
Bijvoorbeeld

```
#w.gpb1 "Vervangen"
```

zal direct de voorgaande tekst vervangen.
Wanneer je het uitvoert in een knop, zal het pas vervangen worden als je op de knop klikt.

#<windowhandle>.<groupboxnaam> "!font fontNaam [size] grootte fontstyles"
Past het font aan van de tekst aan de bovenzijde van de groupbox. Let op het uitroepteken voor het commando.
In de afbeelding Radiobutton – 6 bovenaan is een voorbeeld te zien.

Het tabel bovenaan over de commando's disable, enable, hide en show werkt ook bij groupboxen. Let daarbij weer op het uitroepteken voor het commando.

Deze commando's zouden de groupbox buiten werking of in werking moeten zetten; onzichtbaar of zichtbaar moeten maken.

Bij het commando hide zou je denken dat alles wat in de groupbox staat ook uit zal staan. Echter gebeurt dat niet. Ook al is alles gegroepeerd door de groupbox, de controls erven niet deze commando's. Ook al is de groupbox hun ouder. Experimenteer er eens mee en het resultaat van wat een groupbox doet is het beste te zien met het volgende commando: locate

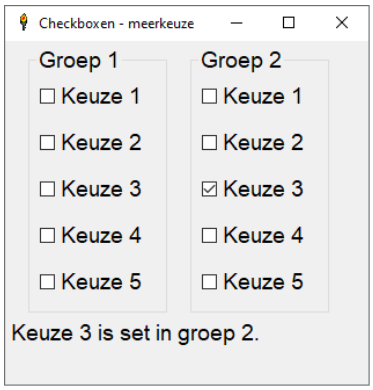
#<windowhandle>.<groupboxnaam> "!locate x y breedte hoogte"
Met dit commando kan een groupbox een andere plaats krijgen. Zoals eerder uitgelegd werkt het alleen als de control is geplaatst in een venster type "window". De verplaatsing gaat pas in na het gebruik van het "refresh" commando voor het betreffende venster.

Meer over het commando locate, zie het hoofdstuk Vensters uitlijnen.

Een andere keuze die gemaakt kan worden is de checkbox. Checkboxen zijn vakjes waarvan er meerdere aangevinkt kunnen worden.

Onderstaand programma laat zien hoe je de checkboxen gebruikt. Bestudeer de variabele t$. Radiobuttons en checkboxen kennen niet de contents eigenschap. We kunnen dus niet nagaan op welk rondje of vakje is geklikt. Via een omweg, de handle$, kunnen we het toch voor elkaar krijgen. Wel moet dan de tekst precies zo in de controlnaam worden genoemd.

```
CALL MaakWindow

'-------------------------------------------------------------------------------
SUB MaakWindow
    WindowWidth  = 330
    WindowHeight = 335

    UpperLeftX = 150
    UpperLeftY = 150

    CHECKBOX #main.chg1Keuze1, "Keuze 1", checkSet1, checkReset1,  30,  35, 90, 25
    CHECKBOX #main.chg1Keuze2, "Keuze 2", checkSet1, checkReset1,  30,  75, 90, 25
    CHECKBOX #main.chg1Keuze3, "Keuze 3", checkSet1, checkReset1,  30, 115, 90, 25
    CHECKBOX #main.chg1Keuze4, "Keuze 4", checkSet1, checkReset1,  30, 155, 90, 25
    CHECKBOX #main.chg1Keuze5, "Keuze 5", checkSet1, checkReset1,  30, 195, 90, 25
    CHECKBOX #main.chg2Keuze1, "Keuze 1", checkSet2, checkReset1, 170,  35, 90, 25
    CHECKBOX #main.chg2Keuze2, "Keuze 2", checkSet2, checkReset1, 170,  75, 90, 25
    CHECKBOX #main.chg2Keuze3, "Keuze 3", checkSet2, checkReset1, 170, 115, 90, 25
    CHECKBOX #main.chg2Keuze4, "Keuze 4", checkSet2, checkReset1, 170, 155, 90, 25
    CHECKBOX #main.chg2Keuze5, "Keuze 5", checkSet2, checkReset1, 170, 195, 90, 25
    STATICTEXT #main.lblKeuze, "", 5, 240, 300, 25

    GROUPBOX    #main.opt1, "Groep 1",  20, 5, 120, 230
    GROUPBOX    #main.opt2, "Groep 2", 160, 5, 120, 230

    OPEN "Checkboxen - meerkeuze" FOR WINDOW AS #main
    #main "trapclose Sluit"
    #main "font arial 14"
    WAIT
END SUB

SUB Sluit handle$
    CLOSE #main
    END
END SUB

SUB checkSet1 handle$
    #handle$ "value? v$"
    t$ = MID$(handle$, 11, 5); " "; RIGHT$(handle$, 1); " is "; v$; " in groep 1."
    #main.lblKeuze t$
END SUB

SUB checkSet2 handle$
    #handle$ "value? v$"
    t$ = MID$(handle$, 11, 5); " "; RIGHT$(handle$, 1); " is "; v$; " in groep 2."
    #main.lblKeuze t$
END SUB

SUB checkReset1 handle$
  #handle$ "value? v$"
  t$ = MID$(handle$, 11, 5); " "; RIGHT$(handle$, 1); " is "; v$; " in groep 1."
  #main.lblKeuze t$
END SUB

SUB checkReset2 handle$
  #handle$ "value? v$"
  t$ = MID$(handle$, 11, 5); " "; RIGHT$(handle$, 1); " is "; v$; " in groep 2."
  #main.lblKeuze t$
END SUB
```



Opdracht:
Kijk eens naar de t$ variabele. Je ziet in elke sub eenzelfde toekenning om een string te maken. Dit kan eenvoudiger.
Schrijf een functie die als parameters de handle$, de v$ en het groepsnummer meekrijgt.
Roep de functie aan waar de variabele t$ achter elke #main.lblKeuze staat.

Opdracht:
Breid bovenstaande opdracht uit door in de mainwin te bepalen hoeveel vakjes er per groep zijn aangevinkt. Gebruik een tweedimensionale array om het groepsnummer en de vakjes bij te houden. Maak een

aparte SUB om het aan te roepen. Doe dit alles alleen in de functie, maar je mag ook een aparte knop gebruiken.

12e. Meer mogelijkheden met groupboxen

Groupboxen zijn containers waar van alles in geplaatst kan worden. Het hoeven niet speciaal alleen maar radiocontrols of checkboxen te zijn. De radiobuttons en de checkboxen mogen door elkaar worden gebruikt in een groupbox.

Groupboxen zijn vensters waar we andere controls in kunnen zetten. In een groupbox mag je ook weer een andere groupbox in zetten. Je mag ze dus nesten. Meestal is één groupbox in een groupbox nesten voldoende. Hoe meer groupboxen je erin zet, hoe onoverzichtelijker het venster wordt, zie volgende hoofdstuk meer daarover.

Het venster, in onderstaand voorbeeld, staat nu niet in een SUB. Later in het boek zullen we zien dat bovenstaande voorbeelden een goede methode is om vensters in subroutines te maken en bovenaan aan te roepen.

Je kunt onderstaand som voorbeeld zelf uitbreiden met meer bereken mogelijkheden. In de volgende paragraaf wordt er uitgelegd dat je uit een lijst een symbool kunt kiezen om daarmee te rekenen.

```
WindowWidth = 600
WindowHeight = 280
UpperLeftX = 100
UpperLeftY = 100
STATICTEXT #w.lblGetal1, "Eerste getal", 50, 60, 120, 35
STATICTEXT #w.lblGetal2, "Tweede getal", 300, 60, 130, 35
TEXTBOX #w.txtGetal1, 180, 55, 70, 35
TEXTBOX #w.txtGetal2, 440, 55, 70, 35
GROUPBOX #w.gbSom, "Een som", 20, 20, 540, 100
STATICTEXT #w, "Opgeteld is:", 50, 170, 180, 35
STATICTEXT #w.lblResult, "", 200, 170, 100, 35
BUTTON #w.btnResult, "Bereken", btnResult, UL, 400, 170, 100, 35
OPEN "Een som" FOR WINDOW AS #w
#w "trapclose Sluit"
#w "font arial 16"
WAIT

SUB Sluit handle$
    CLOSE #w
    END
END SUB

SUB btnResult handle$
    #w.txtGetal1 "!contents? getal1"
    #w.txtGetal2 "!contents? getal2"
    result = getal1 + getal2
    #w.lblResult result
END SUB
```

De twee teksten en de knop staan niet in de groupbox. Het kan wel, maar een groupbox is bedoeld om onderdelen die bij elkaar horen te groeperen. We kunnen zeggen dat wat er in de groupbox staat met invoer te maken heeft. De rest heeft daar niets mee te maken.

Kunnen we dan niet een tweede groupbox maken voor de uitvoertekst en de knop?
Dat kan, maar om alles in groupboxen te gaan verdelen doet het venster ook niet ten goede. Sommige controls horen nou eenmaal bij het venster, zoals de knop. De uitvoer heeft zelf geen doel om in een groupbox te horen.

Liberty BASIC vindt alles goed. Hoe netjes of slordig je de vensters ook maakt; je krijgt geen gezeur van Liberty BASIC. In het volgende hoofdstuk gaan we aan de slag met het uitlijnen van de controls in de vensters.

Opdracht:
In plaats van getallen kun je ook tekst invoeren. Je krijgt dan wel een 0 als resultaat of alleen een antwoord van één van de tekstboxen waar wel een getal in staat.
Probeer een oplossing toe te voegen zodat het resultaat een melding geeft dat er geen getal is ingevoerd. Laat de controle toepassen op beide textboxen zodat er ook een melding verschijnt als één van de textboxen geen getal heeft.

12f. Listboxen en comboboxen

Er zijn twee controls in Liberty BASIC die met lijsten werken. De listbox en de combobox gebruiken ieder een array als lijst.
Het is niet de bedoeling om zo'n array voor andere doeleinden te gebruiken. Dimensioneer unieke arrays voor deze controls.

Syntax:

LISTBOX #<vensterhandle>.<listboxnaam>, <arraynaam$([])>, <eventsubnaam>|<eventlabelnaam>, <x>, <y>, <w>, <h>

De listbox laat een lijst zien waaruit je een item kunt kiezen. Als de lijst langer is dan in de listbox past, verschijnt er een verticale schuifbalk. Er zal geen horizontale schuifbalk aanwezig zijn als je de listbox te smal maakt.
Dit alles geldt ook voor de combobox. De syntax is hetzelfde.

Syntax:

COMBOBOX #<vensterhandle>.<comboboxnaam>, <arraynaam$([])>, <eventsubnaam>|<eventlabelnaam>, <x>, <y>, <w>, <h>

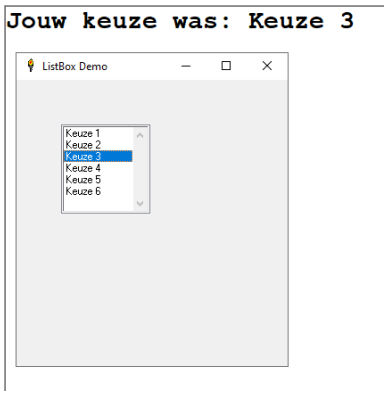
Het enige verschil dat ze hebben is te zien met de uitvoer. Een listbox neemt ruimte in beslag door de lijst, maar een combobox toont geen lijst. De lijst verschijnt alleen als je op het driehoekje rechts klikt. De control bespaart daardoor ruimte.

Zoals bij de syntax is te zien, hoeft een haakje sluiten bij de arraynaam$ niet. Zelf gebruik ik altijd beide haakjes voor een goed overzicht van het hele statement.

Het is ook mogelijk om iets anders in te voeren dan wat er in de lijst staat.

Voorbeeld:

```
DIM lijst$(5)
FOR i = 0 to 5
        lijst$(i) = "Keuze "; i + 1
NEXT i
LISTBOX #Form1.lstKeuze, lijst$(), lstKeuze.Item, 10, 10, 100, 100
OPEN "ListBox Demo" FOR WINDOW AS #Form1
#Form1 "trapclose Form1.Close"
WAIT

SUB Form1.Close handle$
        CLOSE #Form1
        END
END SUB

SUB lstKeuze.Item handle$
        #Form1.lstKeuze "selection? selectCase$"
        PRINT "Jouw keuze was: "; selectCase$
END SUB
```



De array heeft 6 elementen, van 0 tot en met 5. De listbox en de combobox beginnen echter met item 1.
Je kunt ook met het eerste element van de array beginnen om niet te verwarren met element 0 en item 1 dat dezelfde inhoud heeft. Wanneer je dat wilt doen, moet je de expressie in regel 3 wijzigen in alleen de string met de lusvariabele.

Standaard is in een listbox een itemklik als een dubbelklik. De listbox heeft eigenschappen om zulke opties te kunnen wijzigen. Onderstaand tabel werkt ook met een combobox.

"select <string$>" "select "; <variabele$>	Hiermee selecteer je een item in de listbox. Handig om als eerste een item in te schakelen, zodat er geen niet-geselecteerde items zijn.

	Gebruik een stringvariabele buiten de aanhalingstekens. Je mag in plaats van een puntkomma ook de plus-operator gebruiken. Wil je een element van de lijst-array gebruiken? Zorg er dan voor, wanneer jouw array met element 0 begint, dat je van het element er één optelt. Item 0 bestaat niet.
“selectindex <indexnummer>” “selectindex “; <indexvariabele>	Gebruik een indexnummer of een numerieke indexvariabele om een item te selecteren. Wil je een item selecteren vanuit het array-element, zorg er dan voor dat het item één hoger is als jouw array met element 0 begint.
“selection? <selectievariabele$>”	Haalt de inhoud uit het geselecteerde item en bewaart het in de gegeven selectievariabele. Deze variabele moet in de string staan. Gebruik het dus niet buiten de string zoals wel bij andere eigenschappen het moet.
“selectionindex? <indexvariabele>”	Dit is de tegenhanger van “selectindex”. In plaats van een item te selecteren via indexnummer, wordt nu het indexnummer opgehaald uit het geselecteerde item en bewaard in de indexvariabele. Plaats ook weer deze variabele in de string. Niet erbuiten.
“reload”	Vernieuwd de listbox en toont een eventuele gewijzigde lijst als de array is gewijzigd.
“singleclickselect”	Schakel deze methode in als je een enkele klik in de listbox wilt hebben. Standaard is het altijd een dubbelklik. Na deze instelling is er geen mogelijkheid om hem weer in dubbelklikstand te zetten. Ook niet na de “reload” aanroep. Dit commando is niet in de combobox aanwezig.

Hoewel de listbox en de combobox met elkaar gemeen hebben, heeft de combobox wat meer mogelijkheden.

“!contents” * “!<string>” “!”; <variabele$>	* Let op! Verkeerde aanwijzing in de Help. In de Help staat “!contents”. Neem dit niet letterlijk op. Hiermee wordt bedoeld dat achter het uitroepteken jouw inhoud moet staan, dus een string of een variabele. Er bestaat geen “!contents” eigenschap. Wat je opgeeft komt in de textbox van de combobox te staan.
“contents? <variabele$>”	Haalt de inhoud op dat in de textbox van de combobox staat. Het kan een item zijn dat geselecteerd is, maar het kan ook een ingevoerde tekst zijn.

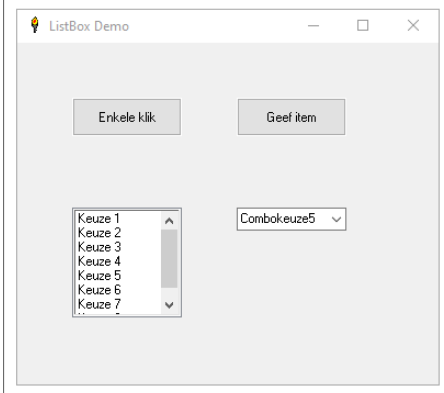
De “contents?” eigenschap kan inhoud ophalen die gekozen is. Ook wat er ingevoerd is in de textbox kan opgehaald worden met “contents?”. Echter, er is maar één event die op de combobox werkt. De actie wordt alleen uitgevoerd als een item gekozen is. Niet wanneer je de tekst in de textbox wijzigt. De enter toets werkt niet om actie uit te voeren. Net als bij de textbox control is er een knop nodig om gewijzigde tekst op te halen.

Onderstaand voorbeeld geeft een groot programma dat de listbox en de combobox demonstreert. Bekijk de afbeeldingen. De eerste afbeelding is een itemkeuze zonder een klik op de knop. De tweede afbeelding is met de klik op de knop, zodat je de tekst kunt printen die je gewijzigd hebt.

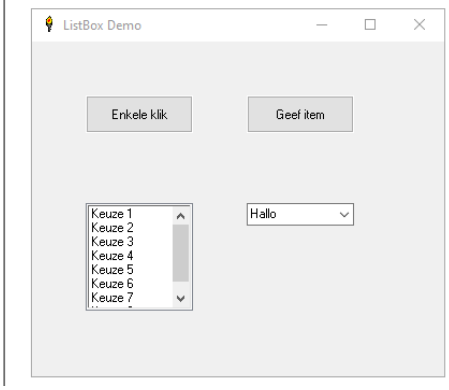
```
DIM lijst$(8), combolijst$(8)
FOR i = 1 TO 8
    lijst$(i) = "Keuze "; i
    combolijst$(i) = "Combokeuze"; i
NEXT i
WindowWidth = 400
WindowHeight = 350
LISTBOX #Form1.lstKeuze, lijst$(), lstKeuze.Item, 50, 150, 100, 100
BUTTON #Form1.btnEnkel, "Enkele klik", btnEnkel.Click, UL, 50, 50, 100, 35
COMBOBOX #Form1.cmbKeuze, combolijst$(), cmbKeuze.Item, 200, 150, 100, 100
BUTTON #Form1.btnCombo, "Geef item", btnCombo.Click, UL, 200, 50, 100, 35
OPEN "ListBox Demo" FOR WINDOW AS #Form1
#Form1 "trapclose Form1.Close"
```

```
#Form1.cmbKeuze "!"; combolijst$(1)
WAIT

SUB Form1.Close handle$
    CLOSE #Form1
    END
END SUB

SUB lstKeuze.Item handle$
    #Form1.lstKeuze "selection? selectCase$"
    PRINT "Jouw keuze uit de listbox was: "; selectCase$
END SUB

SUB btnEnkel.Click handle$
    #Form1.lstKeuze "singleclickselect"
    #Form1.lstKeuze "reload"
END SUB

SUB cmbKeuze.Item handle$
    #Form1.cmbKeuze "contents? item$"
    PRINT "Jouw keuze uit de combobox was: "; item$
END SUB

SUB btnCombo.Click handle$
    #Form1.cmbKeuze "contents? tekst$"
    PRINT "Jouw ingevoerde tekst uit de combobox was: "; tekst$
END SUB
```

```
Jouw keuze uit de combobox was: Combokeuze5
```




```
Jouw keuze uit de combobox was: Combokeuze5
Jouw ingevoerde tekst uit de combobox was: Hallo
```



Er is geen mogelijkheid om na een enter toets de tekst Hallo in de lijst te krijgen. Zodra je op het knopje rechts in de combobox klikt zal de lijst weer verschijnen, maar zonder de tekst die je ingevoerd had. De tekst die je toegevoegd wilt hebben als een nieuw item, moet doorgegeven worden via een knop. Helaas, maar het kan niet anders.

Opdracht:
Maak een knop erbij dat nieuwe tekst automatisch toevoegt in de lijst van de combobox. Je kunt dat doen door element 9 te gebruiken. Vergeet niet het commando “reload” aan te roepen. Houd ook rekening mee met de array in het DIM statement.

Opdracht:
Breid het programma verder uit met twee radiobuttons om een keuze te maken in welke lijst je wilt toevoegen, de listbox of de combobox. Gebruik voldoende elementen in de arrays om steeds meer items toe te kunnen voegen. Je kunt een extra textbox control gebruiken voor de listbox, maar je kunt aan de hand van de radiobutton keuze ook de ingevoerde tekst in de combobox toevoegen in de listbox.
Houd de lengte van de arrays bij zodat je telkens toe kunt voegen. Dus: aantalElementen + 1

12g. Iconen weergeven op de knoppen

Een knop met tekst erop geeft informatie over wat de knop doet. Soms hebben we veel knoppen nodig, maar helaas neemt de tekst op de knop veel ruimte in beslag.
We kunnen in plaats van veel knoppen ook een menu maken, maar dat wordt in de volgende paragraaf besproken.
Als de tekst op de knop geen goede beknopte informatie geeft, is een kleine afbeelding of icoon op de knop voldoende. Dat bespaart veel ruimte en kun je meer knoppen kwijt.

In Deel 4 van het boek wordt de ontwerptechniek besproken waarmee je doormiddel van deze knoppen je eigen werkbalken kunt maken.

Op een normale knop kun je geen icoon zetten. Liberty BASIC heeft er een andere button control voor: BMPBUTTON

Syntax:

BMPBUTTON #<vensterhandle>.<bmpbuttonnaam>, <bitmapbestand>, <eventsubnaam>|<eventlabelnaam>, <x>, <y>

Deze knop kan niet op grootte worden ingesteld. De bitmapbutton regelt het zelf aan de hand van de grootte van de bitmap. Echter kunnen we niet de grootte van de bitmap berekenen om de knoppen netjes naast elkaar of onder elkaar te krijgen. De knop zelf gebruikt ook een marge om de bitmap ruim weer te geven. De knop is dus altijd groter dan de grootte van de bitmap.

De bitmap moet een bestaand bitmapbestand zijn en in de map te vinden zijn waar ook het programma in staat. Indien het bestand in een andere map staat, moet het pad meegegeven worden. Zelf vind ik het verstandig om een aparte map aan te maken om alles erin te zetten inclusief het programma. In het hoofdstuk ‘De grafische omgeving’ zal het duidelijker worden waarom dat de beste keuze is.

```
WindowWidth = 350
WindowHeight = 400
BMPBUTTON #w.btnNieuw, "Nieuw48.bmp", btnNieuw, UL, 10, 10
BMPBUTTON #w.btnPrint, "Print48.bmp", btnPrint, UL, 60, 10
TEXTBOX #w.txtTest, 10, 150, 200, 35
STATICTEXT #w.lblTest, "", 10, 300, 300, 35
OPEN "Bitmap knoppen" FOR WINDOW AS #w
#w "trapclose Quit"
#w "font arial 16"
WAIT

SUB Quit handle$
    CLOSE #w
    END
END SUB

SUB btnNieuw handle$
    #w.txtTest ""
    #w.lblTest ""
END SUB

SUB btnPrint handle$
    #w.txtTest "!contents? tekst$"
    #w.lblTest "Je hebt getypt: "; tekst$
END SUB
```

Je kunt uiteraard je eigen iconen gebruiken.

Zoals ik eerder zei; we kunnen niet uit de breedte van een bitmap bepalen waar we de volgende knop kunnen plaatsen. De tweede knop staat 50 pixels verder en wordt op de *volgende pixel* gezet. De knop is dus 2 pixels breder dan de bitmap. Dat komt door de rechthoek. Maar zie dat ertussen een kleine ruimte is. Je zou denken: 48 + 2 = 50, dan zou de volgende knop toch ertegenaan moeten staan?
We zullen later zien dat het grafisch plaatsen van controls en ook shapes, zie hoofdstuk 'De grafische omgeving', anders werkt dan je zou denken.

Het zit zo: er is een tussenafstand van 50 pixels. De volgende knop wordt niet inclusief met de afstand meegerekend. Die wordt altijd één pixel verder dan de afstand geplaatst. Vandaar de kleine ruimte tussen de knoppen.

Je hoeft het nu niet te begrijpen wat ik bedoel. In het hoofdstuk 'De grafische omgeving' kom ik uitgebreid er op terug.

De bitmap van de knop kan tijdens de uitvoer worden gewijzigd met het "bitmap" commando.

Syntax:

#<vensterhandle>.<bmpbuttonnaam> "bitmap <bitmapnaam>"

De bitmapnaam is geen bestandsnaam maar een naam gegeven in het LOADBMP commando. Alleen via de LOADBMP kun je de bitmap wijzigen.

Syntax:

LOADBMP "<bitmapnaam>", "<bitmapbestand>"

De <bitmapnaam> moet een geldige naam zijn zoals een variabelennaam geldt.
De <bitmapbestand> moet een bestaand bmp bestand zijn. Eventueel een pad gebruiken als het bestand ergens anders staat dan waar het programma in staat.

Zie later meer over het LOADBMP commando.

De bitmap kan worden gewijzigd tijdens een actie van dezelfde knop of in een andere actie. Handig wanneer we een in- en uitschakel effect willen gebruiken. Je kunt dan zien wanneer een knop is ingeschakeld door met "bitmap" een ingedrukte bitmap weer te geven.

12h. Menu's

Teveel knoppen is niet altijd handig. Het komt het overzicht op het venster niet ten goede. Een oplossing is door een menu te maken zodat de gebruiker een item kan kiezen uit het menu.

Menu's zijn in Liberty BASIC niet uitgebreid. Een menu item kan zelf geen submenu hebben.
Elk menu item heeft een klik actie en kan dus door een label of een subroutine worden uitgevoerd. Anders dan bij andere event acties, hebben menu items geen handle$ parameter. Je kunt dus geen meerdere menu items uit laten voeren met maar één event.

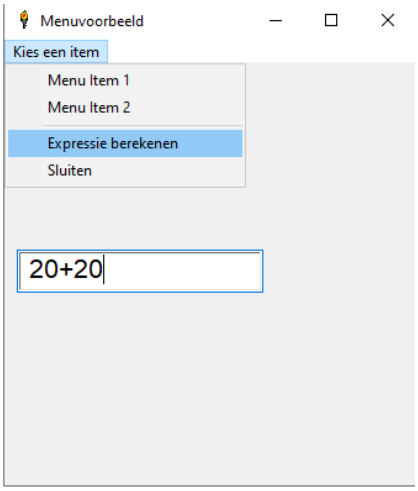
Syntax:

MENU #<vensterhandle>, <titel$>, <item1$>, <eventsubnaam>|<eventlabelnaam>, [<|>] <item1$>, <eventsubnaam>|<eventlabelnaam>, [<|>] …

Elk menu is één regel. Je kunt het in meerdere regels typen door aan het eind een onderstrepingsteken '_' te gebruiken.

Bekijk onderstaand programma. Neem het over en voer het uit.
Het programma heeft code die we eerder ook behandeld hebben, zoals de foutafhandelingen.

```
WindowWidth = 350
WindowHeight = 400
MENU #w, "&Kies een item", "Menu Item 1", mniItem1, "Menu Item 2", mniItem2, |, _
        "Expressie berekenen", mniExpr, "Sluiten", mniQuit
TEXTBOX #w.txtTest, 10, 150, 200, 35
STATICTEXT #w.lblTest, "", 10, 300, 300, 35
OPEN "Bitmap knoppen" FOR WINDOW AS #w
#w "trapclose Quit"
#w "font arial 16"
WAIT

SUB Quit handle$
    CLOSE #w
    END
END SUB

SUB mniItem1
    #w.lblTest "Menu item 1 gekozen."
END SUB

SUB mniItem2
    #w.lblTest "Menu item 2 gekozen."
END SUB

SUB mniExpr
    #w.txtTest "!contents? tekst$"
    ON ERROR GOTO [fout]
    antw$ = EVAL$(tekst$)
    #w.lblTest "Het antwoord is: "; antw$
    EXIT SUB
    [fout]
        #w.lblTest "Expressie is fout!"
END SUB

SUB mniQuit
    CALL Quit "#w"
END SUB
```



Klik op een menu item en je ziet de tekst verschijnen, uitgevoerd door de juiste subroutine.
Omdat een menu item geen handle$ parameter heeft, kunnen we ook niet de Quit aanroepen van de trapclose. In plaats daarvan roep ik hier de Quit event aan in de sub mniQuit. Dit kan wel.

Zoals je ziet maak ik gebruik van een foutafhandeling. Verkeerd ingetoetste expressies worden door de EVAL$ functie niet geaccepteerd. De tekst in de [fout] label wordt uitgevoerd als de EVAL$ functie niet de inhoud van tekst$ uit kan voeren.

Zie eens het menu bovenaan. De pipe | mag niet tussen aanhalingstekens staan, maar is een symbool die Liberty BASIC vertelt dat de menu items gescheiden moeten met een lijn.

12i. Conclusie

Programmeren met de GUI kunnen we object georiënteerd programmeren noemen, maar omdat het allemaal verborgen zit gebruiken we het alleen maar. De OO zit in de GUI, maar we kunnen zelf de OOP techniek niet gebruiken om zelf ermee te programmeren. Zoals ik eerder verteld heb kunnen we over de drempel op de OOP deur kloppen, maar de deur gaat niet open.

Onthoud dus goed dat met de GUI een hele andere manier is om in Liberty BASIC te programmeren. De linkerkant van de drempel hebben we echter niet verlaten. Wat we over de drempel doen is nog steeds een deel van de linkerkant gebruiken. Dat is de structuur van Liberty BASIC. We gaan dus niet plotseling over de drempel een andere programmeertaal gebruiken. Alleen, het is een andere techniek die je in de volgende hoofdstukken zult tegenkomen.

13. Uitlijnen op het venster

Controls plaatsen op de vensters kan een hoop tijd kosten. Er is geen visuele omgeving waardoor we niet direct kunnen zien hoe de controls op het venster komen.

Elke control wordt geplaatst vanaf de linkerbovenhoek. De rechteronderhoek wordt relatief bepaald door de breedte en hoogte te geven.
Er is één control waarmee de *pivot* gewijzigd kan worden. De knop heeft een hoekinstelling vanaf waar de knop moet komen.

- UL
  Plaatst de knop linksboven de gegeven positie (Upper Left)
- UR
  Plaatst de knop rechtsboven de gegeven positie (Upper Right)
- LL
  Plaatst de knop linksonder de gegeven positie (Lower Left)
- LR
  Plaatst de knop rechtsonder de gegeven positie (Lower Right)

Wordt bijvoorbeeld de pivot LR gekozen, dan wordt aan de hand van de breedte en hoogte de linkerbovenhoek berekent. Dit kan nuttig zijn wanneer de knop rechtsonder op een venster moet komen.

13a. De clientview

Het venster bestaat uit drie delen: het raamwerk, de titelbalk en de clientview.
De clientview is het binnenwerk van het venster waar alle controls op worden geplaatst.

We geven met de variabelen WindowWidth en WindowHeight de grootte van het venster aan, maar niet de grootte van de clientview.

Als we een textbox gaan plaatsen:

```
TEXTBOX #w.txt, 10, 100, 100, 35
```

dan betekent dit niet een horizontale afstand van 10 pixels vanaf de rand van het venster, maar vanaf de rand van de clientview en dat geldt ook voor de verticale afstand 100 pixels. Zouden we de textbox op plaats 0, 0 zetten, dan zul je zien dat 0, 0 precies de linkerbovenhoek is van de clientview.

De controls op de juiste plaats zetten kan lastig zijn, vooral als er veel controls zijn. Willen we een afstand hebben van 10 pixels aan de linkerkant en aan de rechterkant, dan zou je het volgende willen proberen:

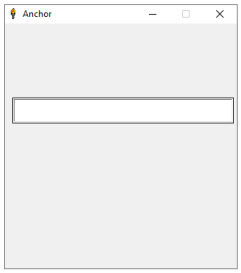
```
TEXTBOX #w.txt, 10, 100, WindowWidth - 10, 35
```

Ga je deze textbox maken en het programma uitvoeren, dan zul je zien dat de rechterkant niet klopt met de afstand minus het vensterbreedte. Dat komt omdat we geen rechterkant van de control in kunnen stellen. Alleen maar een breedte.

Zou je nemen WindowWidth – 20 dan kom je aardig in de buurt. Maar wacht: nog steeds klopt er iets niet. De textbox staat nog steeds niet goed gecentreerd.

Het probleem is de WindowWidth. We kunnen geen breedte van de clientview opvragen, dus moeten we zelf uitrekenen hoeveel pixels we van de WindowWidth af moeten trekken om de control gecentreerd te krijgen.

```
textbox #w.txtAnchor, 10, 100, WindowWidth - 20, 35
open "Anchor" for window_nf as #w
#w "trapclose Quit"
wait

sub Quit handle$
    close #handle$
    end
end sub
```



Doe eens 26 eraf in plaats van 20. Nu zul je zien dat de textbox wel gecentreerd staat. Maar de rand is toch geen 6 pixels dik? Nee, maar we moeten er aan denken dat het inclusief de linker rand is. De rand is dus 3 pixels dik. Zou je niet zeggen, alsof er met het instellen van het venster met WindowWidth en WindowHeight iets niet klopt. Maar daar hoeven we niet aan te denken. We gebruiken per slot van rekening toch alleen maar de clientview. We moeten alleen even met de WindowWidth en WindowHeight experimenteren om de controls op de juiste plaatsen in de clientview te krijgen.

Overigens, voor de linkerbovenhoek (dus de eerste twee parameters) hoeven niet met de WindowWidth en WindowHeight berekent te worden, tenzij je de controls wilt vastzetten op je venster, zoals een knop die rechtsonder moet blijven staan.

Haal ook eens de _nf weg bij window_nf en start het nog eens. Nu klopt het helemaal niet meer. Kennelijk gebruikt het windowframe een deel van de clientview.
Trek er nu eens 36 vanaf in plaats van 26. Start het programma en het blijkt nu weer te kloppen.
Waarom het frame 10 pixels extra gebruikt is weer een geheim van de Windows vensters. Je hoeft je daar niet om te bekommeren. We moeten alleen weten hoeveel pixels we af moeten trekken vanaf de rechterkant om de control op de juiste plaats te krijgen, met of zonder het windowframe.

Andere programmeertalen hebben een anchor en dock instelling. Het vastzetten van een control, zodat de afstanden hetzelfde blijven, moeten we in Liberty BASIC zelf doen. Er is een mogelijkheid om de textbox vast te zetten met de afstanden, ook al zouden we het venster van grootte wijzigen. In de volgende paragrafen wordt het uitgelegd.

13b. Anchor – de controls vastzetten

De eerste vraag zal zijn: waarom zetten we een control vast? Een control staat toch altijd op een vaste positie? Wel als we het venster op de ingestelde grootte laten, maar zodra je het venster van grootte wijzigt, zal de plaats van de control geen vaste plaats meer hebben.

Onthoud dat de plaats alleen veranderd als je de WindowWidth en/of WindowHeight erbij gaat gebruiken. Deze hebben invloed op de wijziging van het venster. Bovendien, als je een control met gewone getallen plaatst dan maak je geen anchor voor die control. De control zal dan zijn eigen plaats en grootte houden, ook al zou je de grootte van je venster wijzigen.

Het vaststellen van een anchor kan handig zijn als je graag wilt dat de textbox in de clientview gecentreerd blijft. De knoppen daarentegen kunnen wel een eigen anchor hebben, zoals in de eerste paragraaf van dit hoofdstuk is uitgelegd over de hoekinstellingen van de knoppen.

Onderstaande code geeft een voorbeeld hoe de anchor werkt en hoe de hoekinstellingen van de knoppen werken.

```
TEXTBOX #w.txtAnchor, 10, 100, WindowWidth - 36, 35
BUTTON #w.btnAnchor1, "Anchor 1", btnAnchor1, UL, WindowWidth - 210, WindowHeight - 105, 80, 45
BUTTON #w.btnAnchor2, "Anchor 2", btnAnchor2, LR, 80, 40, 80, 45

OPEN "Anchor" FOR WINDOW AS #w
#w "trapclose Quit"
#w "resizehandler Anchor"
WAIT

SUB Anchor handle$
    #w.txtAnchor "!locate 10 100 "; WindowWidth - 22; " 35"
    #w "refresh"
END SUB

SUB Quit handle$
    CLOSE #handle$
    END
END SUB
```

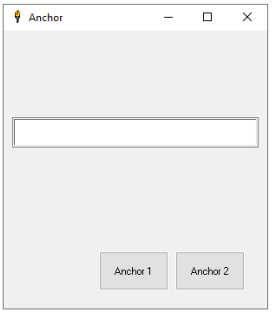


fig1. Na de start

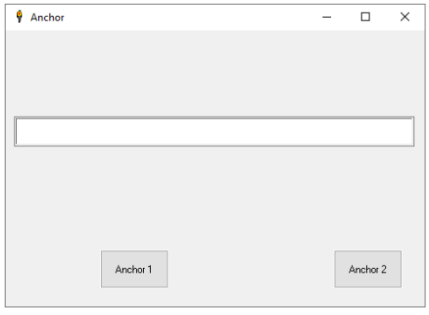


fig2. Na het breder maken van het venster

Bekijk de code en de afbeeldingen goed. Merk op dat de knop Anchor1 geen anchor heeft. Deze knop blijft op zijn eigen positie staan. De UL instelling zorgt er niet voor dat de knop netjes uitgelijnd links van Anchor2 blijft staan. Hoewel Anchor2 niet in de Anchor event staat vermeld, gaat toch de knop netjes mee naar rechts en blijven de afstanden ongewijzigd. Met de instelling LR kun je een anchor maken voor de knop en hoef je die niet in de Anchor eventhandler te laten wijzigen met het !locate commando.

Alleen de textbox staat in de Anchor sub event, zodat de textbox breder wordt en daardoor netjes in het midden blijft.

Merk op dat in de Anchor event een WindowWidth - 22 staat in plaats van WindowWidth - 36. Wijzig de 22 in 36 en start het programma weer. Verbreedt je venster en merk op dat de rechterafstand niet meer klopt. Weer een probleem. Waarom dit gebeurd weet ik zelf eigenlijk niet. Foutjes in Liberty BASIC? Maar ja, als het maar werkt.

13c. Controls horizontaal en/of verticaal dokken

Een andere anchor mogelijkheid is om een deel van de clientview in te nemen met een control.
Werkbalken en statusbalken zijn een goed voorbeeld om controls te dokken (Engels: docking), ofwel een anchor in te stellen met een horizontale dok vanaf pixel 0 tot de vensterlengte minus de afstand of een verticale dok vanaf pixel 0 tot de vensterhoogte minus de afstand. Die afstand kan niet altijd 3 pixels zijn als bijvoorbeeld er een schuifbalk is.

Net als bij de anchor, moeten we de controls zelf docken. De knoppen kennen zelf een anchorinstelling zoals je eerder zag, maar er is geen dockinstelling.

MDI applicaties (Multiple Document Interface) gebruiken vaak een control dock zoals een tekst control die de hele clientview in beslag moet nemen.

Eigenlijk is de werking precies hetzelfde als bij een anchor. Het enige dat je moet doen om van docken te spreken is dus een hele kant van je venster door een control laten gebruiken en ervoor zorgen dat het ook zo blijft, wanneer het venster van grootte veranderd.

Docking is een techniek dat niet standaard in Liberty BASIC bestaat. We moeten het zelf maken door de anchor zo in te stellen dat we precies de lengte van boven naar onder of van links naar rechts van de clientview in beslag nemen.

De positie en lengte van bovenstaande textbox kunnen wijzigen zodat we een horizontale docking hebben.

```
TEXTBOX #w.txtAnchor, 0, 100, WindowWidth - 16, 35
```

De WindowWidth - 16 werkt echter niet in de Anchor event. In de anchor event moet de hele WindowWidth opgegeven worden om horizontaal de lengte in beslag te houden.

```
#w.txtAnchor "!locate 0 100 "; WindowWidth; " 35"
```

Er is geen verklaring waarom in de Anchor event de hele breedte van het venster gegeven moet worden. Je zult zien dat als je 16 ervan aftrekt er een opening aan de rechterkant ontstaat. Je hebt natuurlijk nog steeds een anchor, want de opening blijft even breed. Maar je kunt dan niet van een horizontale docking spreken.

Wijzig de locate van de textbox nu zo:

```
#w.txtAnchor "!locate 100 0 35 "; WindowHeight
```

Zodra je het venster van grootte veranderd, gaat de textbox verticaal gedokt staan. Maak het venster verticaal groter en je zult zien dat de anchordok perfect werkt.

Je begint wel horizontaal. Begin verticaal door de textbox als volgt te wijzigen in:

```
TEXTBOX #w.txtAnchor, 100, 0, 35, WindowHeight - 39
```

Je ziet dat je verticaal meer dan 16 moet aftrekken om de juiste verticale breedte te krijgen.

Ook nu is het vreemd waarom in de Anchor event de hele WindowHeight gebruikt moet worden, alsof de resize-handler alleen de clientview regelt en daardoor de parent (het venster zelf) meegaat. Als dat het is, dan geeft dat de conclusie dat de resizehandler zelf voor ons de hoogte en breedte van de clientview berekent en we daardoor niets van hoeven af te halen.

13d. Wat de Liberty BASIC gebruiker wil

Dit hoofdstuk is in principe een hoofdstuk waarmee je leert om te gaan met uitlijnen op vensters. Echter is dat niet verplicht. Je hoeft niet per sé dit te hanteren om te kunnen programmeren. Wel is te zien dat Liberty BASIC de mogelijkheid heeft om controls goed uit te lijnen, maar het is helemaal optioneel.

Ook welke vensters er gebruikt moeten worden is helemaal aan de gebruiker. Zoals we zagen, heeft Liberty BASIC verschillende ingebouwde vensters. Wil je een grafisch gedokt venster, open het venster als een graphics venster. Wil je een gedokt tekst venster, open het venster als een text venster. Maar het is helemaal aan jou welk venster je gebruikt en hoe je het maakt.

Als je weet dat je vensters gebruikt om daar wat op te plaatsen, dan weet je nu ook welk deel van het venster gebruikt wordt. Tot nu toe noemde ik het de clientview, maar je kunt het ook het werkgebied of clientarea noemen. Maar we blijven gewoon zeggen: we werken met vensters en wat er precies op een venster gebeurd hoeven we niet te weten. Als we maar weten hoe we het gebruiken.

In het volgende hoofdstuk komen de dialoogvensters aan bod. In de eerste pa.

14. De dialoogvensters van Liberty BASIC

Liberty BASIC heeft ingebouwde dialoogvensters. Deze kun je gebruiken als commando's.

Er zijn twee dialoogvensters die werken als de MessageBox die als commando werkt in andere programmeertalen. In Liberty BASIC kunnen we ook dit commando gebruiken, maar alleen via de MessageBoxA API functie. Om er voor te zorgen dat we die functie niet nodig hebben, zijn er twee ingebouwde commando's in Liberty BASIC: NOTICE en CONFIRM

14a. Vensters NOTICE, CONFIRM en PROMPT

Het eerste commando, NOTICE, is een dialoogvenster dat alleen berichten toont met één knop om het venster te sluiten.

Syntax:

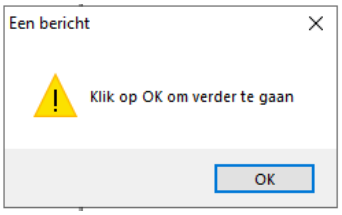
NOTICE [<title$> + CHR$(13) +]<bericht$>

Het bericht wordt getoond met een OK knop rechtsonder.

Voorbeeld:

```
NOTICE "Een bericht"
```

Als je alleen een bericht meegeeft, zal de titel '- Notice -' zijn.
Je kunt de titel wijzigen door na een string de CHR$() functie te gebruiken met code 13. Normaal is dit de code voor carriage return, maar dit geeft de functie dat de gegeven string niet bij het bericht hoort maar de titel van het venster wordt.



Wijzig het voorbeeld in:

```
NOTICE "Een bericht" + CHR$(13) + "Klik op OK om verder te gaan"
```

Geef je nog meer CHR$(13) in het bericht, dan zullen deze wel zorgen voor nieuwe regels in het bericht.

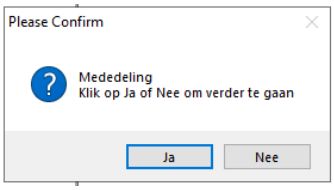
Zoals gezegd wordt hier alleen een bericht getoond, maar wordt er niets gevraagd. Het ander venstercommando CONFIRM heeft een variabele nodig die de waarde krijgt van de geklikte knop.

Syntax:

CONFIRM <bericht$>; <result$>

Voorbeeld:

```
CONFIRM "Mededeling" + CHR$(13) + "Klik op Ja of Nee om verder te gaan"; result$
PRINT "Je klikte op: "; result$
```



De functie CHR$() met code 13 werkt niet als scheiding met een titel en het bericht, zoals bij NOTICE. Altijd zal er in het Engels gevraagd worden om 'Please Confirm'.
Hoewel het venster vanuit Windows wordt geopend, waardoor we Nederlandstalige knoppen hebben, zal zowel de titel als wat de variabele result$ ontvangt Engelstalig zijn. Als je op één van de knoppen klikt, zal het resultaat van result$ een 'yes' of een 'no' zijn.

In de Appendix Voorbeelden zul je een MessageBox voorbeeld vinden. Daarmee heb je zelf controle over het gebruik van de titel en de knoppen en heb je CONFIRM niet meer nodig.

Uiteraard ga ik in dit hoofdstuk uitleggen hoe je zelf dialoogvensters maakt en hoe je ze via een hoofdvenster laat functioneren.

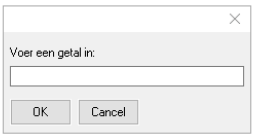
Liberty BASIC kent ook een invoervenster. Andere programmeertalen kennen het als de INPUTBOX. In Liberty BASIC is dat het PROMPT commando.

Syntax:

PROMPT <stringvar$>; <resultvar$>

Voorbeeld:

```
nomainwin
prompt "Voer een getal in: "; getal$
if getal$ = "" then
    print "Je hebt geen getal ingevoerd."
else
    print getal$
end if
```



Hier heb ik de mainwin even uitgeschakeld om dit goed te kunnen knippen. Het mainwin venster bleef steeds voor de code staan. Omdat ook de PROMPT een modaal dialoogvenster is, kon ik niet de mainwin verschuiven. Zet de nomainwin in commentaar om het resultaat op de mainwin te kunnen zien.

Wanneer je op Cancel klikt, zal de variabele getal$ leeg zijn en ook wanneer je niets invoert.

Merk op dat het venster Engelstalig is. Dit venster komt dus niet uit Windows vandaan. Het venster is door Carl Gundel ingebouwd.

Het PROMPT commando werkt op dezelfde manier als het NOTICE commando. Door een tekst te geven met erachter de CHR$(13), wordt de eerste string de titelstring van de prompt.

```
PROMPT "Getal"; CHR$(13); "Voer een getal in: "; getal$
```

Het is ook mogelijk om de resultvar$ te wijzigen, die in de syntax staat vermeld. De variabele mag namelijk ook numeriek zijn.

Opdracht:
Haal eens het dollarteken weg zodat je numeriek gebruikt. Wijzig ook het IF statement. Controleer bijvoorbeeld op waarde 0.

De PROMPT kan dus werken met stringinvoer en numerieke invoer. De PROMPT kan nog meer. Wanneer je van te voren al een waarde toekent aan de result variabele, zal de waarde verschijnen in de textbox in het venster.

14b. Vensters FILEDIALOG en COLORDIALOG

Het venster FILEDIALOG is uitgebreid besproken in hoofdstuk Bestandsbeheer. Dit venster is niet een standaard Windows venster. Dit venster is zo ontworpen dat er, doormiddel van de titel, bepaald wordt welk Windows venster geopend moet worden. Door het woord ‘open’ of ‘save’ in de titel op te geven, weet Liberty BASIC of het OpenDialog venster of het SaveDialog venster geopend moet worden.

Syntax:

FILEDIALOG <titel$>, <sjabloon$>, <variabele$>

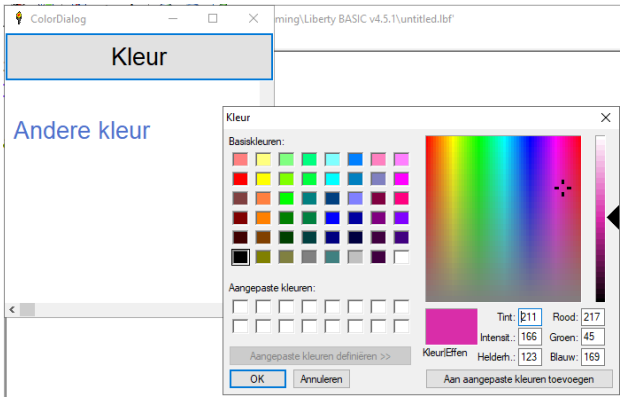
De COLORDIALOG is een Windows venster. Dit venster is dan ook te zien zoals je het venster ziet in andere software, die ook gebruik maken van de Windows vensters.

Syntax:

COLORDIALOG <huidigeKleur$>, <gekozenKleur$>

Neem onderstaand programma over en oefen met de COLORDIALOG.

```
BUTTON #w.btnKleur, "Kleur", btnKleur, UL, 0, 0, WindowWidth - 16, 55
OPEN "ColorDialog" FOR GRAPHICS AS #w
#w "trapclose Quit"
#w.btnKleur "!font arial 20"
#w "font arial 20"
#w "down"
#w "place 10 120"
WAIT

SUB Quit handle$
    CLOSE #handle$
    END
END SUB

SUB btnKleur handle$
    COLORDIALOG "black", kleur$
    #w "color "; kleur$
    #w "|Andere kleur"
END SUB
```



Meer over de grafische gereedschappen, zie hoofdstuk 16.

Opdracht:
Breid bovenstaand programma uit door de gekozen kleur te bewaren, zodat met het kiezen van de volgende kleur de huidige kleur als eerste parameter meegegeven wordt.

Opdracht:
Ga verder met de vorige opdracht door na elke 5 keer dat de tekst “Andere kleur” verschijnt, weer op de oude plaats de tekst weer wordt getekend. Gebruik een variabele die het aantal keren bijhoudt en zorg ervoor dat de teller na 5 keer weer opnieuw telt.

Tip!
Omdat je event subroutines gebruikt, heb je globale variabelen nodig. Tenzij je liever labels wilt gebruiken.

14c. Venster PRINTERDIALOG

De PRINTERDIALOG is een uitgebreid Windows venster. Het is dus een dialoogvenster, niet ontworpen door Carl Gundel.

Toch is de PRINTERDIALOG niet zo vriendelijk voor gebruik. Er zijn een hoop ingebouwde globale variabelen nodig om alles in het venster in te kunnen stellen en/of op te kunnen vragen wat nodig is. Omdat het losse

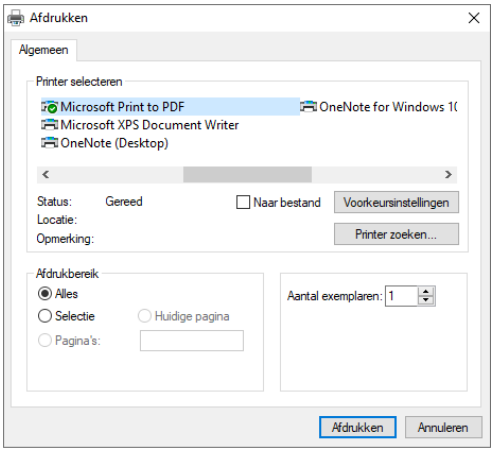
variabelen zijn, kun je alles in het begin instellen en hoef je pas veel later het te gebruiken. Met andere woorden, de losse variabelen behouden hun waarden. Ook al zou je dat later in het programma niet meer willen.

De PRINTERDIALOG is een standaard common dialog. Het commando opent dus het standaard Windows print dialoog venster. Elke programmeertaal gebruikt het venster anders. Zo moeten we met variabelen alles instellen en kan er geprint worden.

Net als bij de FILEDIALOG doet de PRINTERDIALOG verder niets bijzonders. Het printen moet zelf worden gedaan. Bij het kiezen van een printer en hoeveel keer we willen printen, kunnen we in een lus met het commando LPRINT gaan printen.

Onderstaand voorbeeld komt uit de Help. Het laat zien hoe het werkt. Zelf kan ik geen voorbeelden maken en uitvoeren, want ik heb geen printer.

```
'choose a file to print
filedialog "Print a BAS file", "*.bas", fileToPrint$
if fileToPrint$ <> "" then
  printerdialog
  print "PrinterName$ is ";PrinterName$
  print "PrintCopies is ";PrintCopies
  print "PrintCollate is ";PrintCollate
  print "PrinterFont$ is ";PrinterFont$
  if PrinterName$ <> "" then
    open fileToPrint$ for input as #readMe
    while not(eof(#readMe))
      line input #readMe, line$
      lprint line$
    wend
    close #readMe
    dump
  end if
end if
end
```



Het printer dialoogvenster

Volgens de Help, werkt PrintCopies alleen als de printer het ondersteund. Onderstaande tekst komt uit de Help, vertaald.

Als het printerstuurprogramma meerdere exemplaren kan afdrukken, wordt PrintCopies ingesteld op "1" en hoeft het programma de tekst maar één keer af te drukken. Als het printerstuurprogramma niet overweg kan met het afdrukken van meerdere kopieën, dan bevat PrintCopies het aantal exemplaren dat door de gebruiker in het printerdialoogvenster is gekozen en moet het programma deze kopieën in een lus afdrukken.

Experimenteer maar eens met dit venster. Ik kan hier niet verder op ingaan, omdat ik het niet kan uittesten.

14d. Zelf dialoogvensters maken

Het is fijn dat Liberty BASIC ingebouwde vensters heeft, maar omdat sommige vensters Engelstalig werken is het gebruik ervan niet altijd naar wens. Zulke vensters kun je ook zelf maken. Je kunt zelf een eigen confirm venster maken.

Voor het zelf maken van dialoogvensters heb je geen API nodig. Eerder gaf ik aan dat je via API de normale MessageBox van Windows kunt aanroepen, maar het kan ook zonder. Je kunt ook een eigen berichtvenster ontwerpen en dat gebruiken in je programma. Hoe dat allemaal werkt en vooral hoe je een dialoogvenster ontwerpt leg ik uit in dit hoofdstuk.

14e. Een dialoogvenster openen

Vaak wordt gedacht dat een dialoogvenster onderdeel is van het hoofdvenster. Zoals je eerder zag, kunnen dialoogvensters ook direct worden geopend zonder een hoofdvenster.

Het is natuurlijk logisch om een dialoogvenster niet als hoofdvenster te gaan gebruiken. Dialoogvensters zijn daar niet voor bedoeld. Onderstaande punten geven het verschil met een gewoon venster.

- Dialoogvensters hebben geen menu.
- Dialoogvensters kunnen geen texteditor hebben, vanwege de carriage return die altijd op een knop werkt.
- Dialoogvensters hebben geen frame. Je kunt ze dus niet groter en kleiner maken.
- Alleen dialoogvensters kunnen modaal zijn.

In de Help staat niet over het gebruik van stylebits. Een dialoogvenster kan andere stylebits hebben dan een normaal venster kan hebben.
Toch had Liberty BASIC wat strenger mogen zijn. Neem onderstaande code over en kijk wat er gebeurd.

```
STYLEBITS #w, _WS_MINIMIZEBOX OR _WS_MAXIMIZEBOX, 0, 0, 0
OPEN "" FOR DIALOG AS #w
#w "trapclose Quit"
WAIT

SUB Quit handle$
    CLOSE #handle$
    END
END SUB
```

Het venster reageert nu als een normaal venster. Maar schijn bedriegt: er is nog steeds geen frame en het werkt nog steeds als een dialoogvenster. Dialoogvensters horen eigenlijk alleen de sluitknop te hebben. De stylebits had in principe niet mogen werken, evenmin er geen menu op kan werken.

Probeer maar eens onderstaande regel toe te voegen voor de STYLEBITS regel en een laatste subroutine mniSluiten erbij te maken.

```
MENU #w, "&Bestand", "&Sluiten", mniSluiten

...

SUB mniSluiten
        CALL Quit "#w"
END SUB
```

Start het programma nog eens. Een menu is niet te zien, maar er is ook geen foutmelding. Liberty BASIC negeert gewoon het menu.

Een dialoogvenster heeft geen frame. Er is wel een instelling DIALOG_NF wat NF betekent als NoFrame. Wijzig het venster eens met een DIALOG_NF. Als je het uitvoert, zul je zien dat er geen verschil is. Een dialoogvenster is altijd zonder frame.

Wanneer gebruik je een dialoogvenster en voor welk doel?

- Als je een berichtvenster nodig hebt. Handig wanneer je wat meer details wilt dan een NOTICE of een CONFIRM kunnen geven.
- Als er wat ingevoerd moet worden. Invoervensters kunnen gebruikt worden voor het invoeren van recordvelden. De controls, die de gegevens weergeven, kunnen dan beschermd worden door ze uit te zetten (disable).
- Als bevestigingsvenster: Vergelijkbaar met de PROMPT. Iets in te moeten voeren en met een OK of Annuleren het bevestigen.

Natuurlijk zijn nog veel meer soorten dialoogvensters te bedenken. In dit hoofdstuk wordt uitgelegd hoe je dialoogvensters kunt besturen.

Zou je zelf een berichtvenster willen maken als een MessageBox? Dat kan. Je hoeft dan niet de API functie te gebruiken. In de appendix Voorbeelden zie je hoe een berichtvenster gemaakt wordt met verschillende knoppen. Het is namelijk mogelijk zelf te bepalen welke controls je op het venster wilt hebben, zonder ze allemaal te moeten maken en een paar aan of uit te moeten zetten.

Mijn uitleg *'zonder ze allemaal te moeten maken'* bedoel ik mee dat je doormiddel van een keuze kunt bepalen welke controls je maken wilt. Je moet ze allemaal wel maken, maar ze komen dan niet allemaal bij elkaar. In de appendix zul je zien hoe dat werkt.

14e. Dialoogvenster openen via een normaal venster

Hoewel dialoogvensters als hoofdvensters geopend mogen worden, is de functie van een dialoogvenster daar niet voor bedoeld. In de vorige paragraaf staat ook de opsommingen over het verschil tussen een dialoogvenster en een normaal venster. Een normaal venster kan bijvoorbeeld niet modaal zijn.

Een dialoogvenster wordt geopend op een moment dat er actie uitgevoerd moet worden dat niet op het normale venster thuishoort, zoals berichten of gegevensinvoer. Ook meldingen kun je weergeven op een dialoogvenster. Misschien moet er wat bevestigd worden.
Dit allemaal op één venster gaan doen, maakt het alleen maar rommelig. Bovendien moet je knoppen aan of uitzetten wanneer een bericht wel of niet verschijnt. Ook is het vaak zo dat de gebruiker eerst wat moet doen voordat zij/hij verder mag gaan met het andere, zoals persoonsgegevens invoeren. Pas na het bevestigen dat de gebruiker klaar is met de invoer, kan zij/hij verder gaan met andere bewerkingen.

Hoe kunnen we een dialoogvenster laten verschijnen via het normale venster? Laten we eens kijken hoe dat gaat.

Maak in de code eerst een normaal venster, zoals hieronder.

```
BUTTON #w.btnTest, "Klik mij", btnTest, UL, WindowWidth / 2 - 50, WindowHeight / 2 - 35, 100, 35
OPEN "Normaal venster" FOR WINDOW AS #w
#w "trapclose Quit"
#w "font arial 12"
WAIT

SUB Quit handle$
    CLOSE #handle$
    END
END SUB
```

De subroutine btnTest moet het dialoogvenster openen.

```
SUB btnTest handle$
    STATICTEXT #dlg.st, "Bevestigen", WindowWidth / 2 - 50, WindowHeight / 2 - 35, 100, 35
    BUTTON #dlg.btnOK, "OK", DialogClose, UL, WindowWidth - 85, WindowHeight - 85
    OPEN "Bericht" FOR DIALOG AS #dlg
    #dlg "trapclose DialogClose"
    #dlg "font arial 12"
    WAIT
END SUB

SUB DialogClose handle$
    CLOSE #dlg
END SUB
```

Zowel de knop als de trapclose gebruiken beide dezelfde subroutine om het dialoogvenster weer te sluiten. De trapclose van een ander venster heeft geen END commando. Het programma mag alleen worden gestopt als het hoofdvenster gesloten wordt.

Er hoeft geen “setfocus” aan het normale venster gegeven te worden. Als het dialoogvenster gesloten is, zal automatisch weer op actie worden gewacht van het normale venster.

Stel dat we het venster niet creëren in de btnTest subroutine, maar in de subroutine aanroepen; het bevestigingsvenster dus in een aparte subroutine zetten.
De vraag is: wat gebeurt er met de code na de aanroep van de subroutine waar het venster gemaakt en geopend wordt? Onderstaande code is een uitbreiding van het vorige voorbeeld.

```
BUTTON #w.btnTest, "Klik mij", btnTest, UL, WindowWidth / 2 - 50, WindowHeight / 2 - 35, 100, 35
OPEN "Dialoogvenster" FOR WINDOW AS #w
#w "trapclose Quit"
#w "font arial 12"
WAIT

SUB Quit handle$
    CLOSE #handle$
    END
END SUB

SUB btnTest handle$
    CALL BevestigenVenster
```

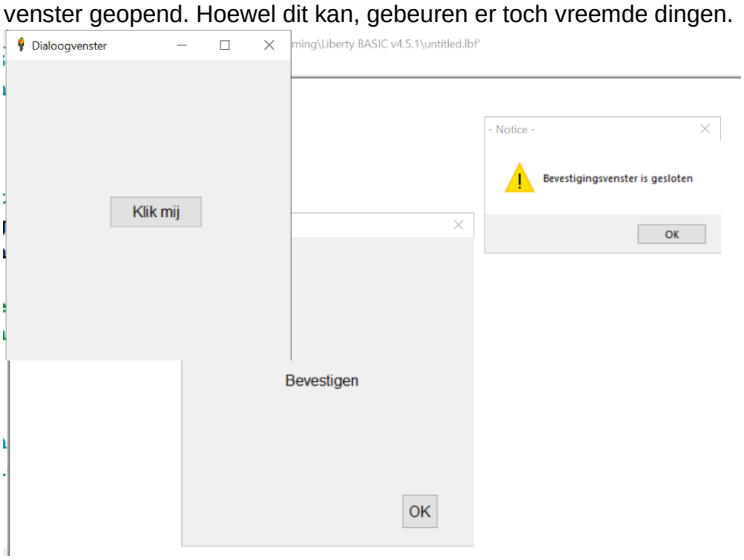
```
    NOTICE "Bevestigingsvenster is gesloten"
END SUB

SUB BevestigenVenster
    STATICTEXT #dlg.st, "Bevestigen", WindowWidth / 2 - 50, WindowHeight / 2 - 35, 100, 35
    BUTTON #dlg.btnOK, "OK", DialogClose, UL, WindowWidth - 85, WindowHeight - 85
    OPEN "Bericht" FOR DIALOG AS #dlg
    #dlg "trapclose DialogClose"
    #dlg "font arial 12"
    WAIT
END SUB

SUB DialogClose handle$
    CLOSE #dlg
END SUB
```

Kopieer en plak de code in Liberty BASIC. Bekijk wat er veranderd is. Je ziet in de btnTest subroutine dat er een NOTICE commando wordt aangeroepen. Als je het programma uitvoert, op de knop klikt en het Bevestigen dialoogvenster geopend wordt, klik dan eens op de OK knop. Het Bevestigen dialoogvenster wordt gesloten en het venster met de knop “Klik mij” krijgt weer de focus. Maar er komt geen venster met het bericht dat na de aanroep van het BevestigenVenster staat.

Normaal gesproken wordt er code na een subroutine aanroep verder uitgevoerd, maar deze keer niet. Als je apart een venster opent in een subroutine en het venster wordt weer gesloten, stopt ook verder de uitvoer in de subroutine waar de aanroep plaatsvond. Dat komt doordat het andere venster weer de focus krijgt waardoor automatisch het WAIT commando wordt aangeroepen om weer actie uit te kunnen voeren, zoals de knop “Klik mij” of het afsluiten van het venster.

Wil je toch berichten weer kunnen geven na het sluiten van een venster, zorg er dan voor dat het bericht in de subroutine uitgevoerd wordt die voor het sluiten moet zorgen. Als voorbeeld zou je het NOTICE commando kunnen verplaatsen voor de CLOSE #dlg in de DialogClose subroutine.

Als je het NOTICE commando verplaatst hebt, dan wordt, voordat het dialoogvenster gesloten wordt, het notice venster geopend. Hoewel dit kan, gebeuren er toch vreemde dingen.



Houd het Notice venster open en klik op het venster met de “Klik mij” knop, zie afbeelding.

Hoewel deze ingebouwde dialoogvensters modaal zijn, kunnen we toch op een ander venster klikken. Klik je dus eerst op het “Klik mij” venster, dan zul je zien dat je kunt wisselen. Ook het Bevestigen venster is aanklikbaar.

Je kunt zelf weer op OK klikken. Het Notice venster verschijnt dan nogmaals.

Ook al kun je het “Klik mij” venster de focus geven door erop te klikken, de trapclose van het venster zal niet werken.

Als je hiermee oefent dan zul je zien dat, als je meer dialoogvensters tegelijk open hebt staan, dit averechts kan werken en zodanig waardoor Liberty BASIC in de war kan raken en zelf niet meer de afsluitvolgorde weet en klachten geeft over het sluiten van de vensters.

Houd je gewoon aan de sluitvolgorde van de vensters. Alles zal dan normaal werken. Het vensterrommel probleem hoeft niet per sé door dialoogvensters te komen. Het probleem ontstaat ook wanneer je meer dan één niet-modale venster open hebt staan. Liberty BASIC kan makkelijk in de war raken als vensters niet op de juiste volgorde afgesloten worden.

Verwissel eens het NOTICE commando en het CLOSE commando en voer het programma nog eens uit. Nu zal het bericht van het NOTICE commando wel verschijnen na het sluiten van het dialoogvenster. Kennelijk heeft het dus niets te maken met het sluiten van een venster, maar de END SUB die uitgevoerd wordt waardoor de focus op het Klik Mij venster er voor zorgt dat het WAIT commando wacht op actie en andere code van subroutines niet verder uitgevoerd worden.

Een uitgebreid voorbeeld over dialoogvensters kun je vinden in mijn forum en in de appendix van het boek.

Opdracht:
Schrijf een programma met een normaal venster en een dialoogvenster. Maak het dialoogvenster als een confirm venster en bepaal zelf wat voor actie er uitgevoerd moet worden in de knoppen Ja en Nee. Bedenk ook zelf hoe je het dialoogvenster opent via het normale venster.

Opdracht:
Schrijf een programma met een normaal venster met twee knoppen: een Open invoervenster knop en een Resultaat knop. Maak een dialoogvenster waar iets ingevoerd kan worden met een OK knop. Laat via de Resultaat knop de ingevoerde gegevens verschijnen op het normale venster.