## 12.1 Adressen

In het vorige hoofdstuk hebben we kennis gemaakt met het type Pointer. In een variabele van het type Pointer kun je een adres bewaren. Bij de toepassing van dynamische variabelen kan een pointervariabele gedeclareerd worden als wijzer naar een bepaald gegevenstype op de heap. Dit is een efficiënte manier om met het beschikbare geheugen om te gaan.

Elke byte in je computer heeft een adres en is op dat adres terug te vinden. De mate waarin je in staat bent om met het geheugen van je computer om te gaan, is bepalend voor de vraag in hoeverre je als programmeur de computer in je macht hebt.

Bij het maken van een programma kan het voorkomen dat je iets op een bepaalde manier wilt oplossen, maar dat de computertaal die je gebruikt daar niet de mogelijkheid toe biedt. Als je dan in staat bent om het geheugen rechtstreeks te manipuleren, heb je een goede kans dat je toch het probleem op de gewenste manier kunt oplossen.

In onderstaand stukje programma is de variabele GETAL van het type Integer. De variabele ADRES is een pointer. GETAL staat op een eigen adres. Het apestaartje (ASCII-nummer 64) wordt gebruikt als operator om achter het adres van GETAL te komen. In het programma wordt het adres van GETAL in de pointervariabele ADRES gezet:

```
PROGRAM GEH_1;
VAR
   GETAL: Integer;
   ADRES: Pointer;

BEGIN
   GETAL := 45;
   ADRES := @Getal
END.
```

Bij dynamische variabelen moet je even uitkijken. Een pointervariabele heeft namelijk zelf een adres, maar hij kan ook een adres bevatten.

In het programma GEH_2 wordt het type PRec gedefinieerd. PRec is een pointer naar het type TRec. TRec is een record met de velden Naam en Plaats:

```
PROGRAM GEH_2;

TYPE
PRec = ^TRec;
TRec = Record
   Naam  : String[30];
   Plaats: String[30];
END;

VAR
   REC: PRec;
   P  : Pointer;

BEGIN
   GetMem(REC,SizeOf(TRec));
   P := @Rec;
   P := REC;
   FreeMem(REC,SizeOf(TRec))
END.
```

De variabele REC wordt gedeclareerd als type PRec. Als er geheugenruimte op de heap gereserveerd is, bevat REC het adres op de heap waar voldoende geheugen aanwezig is om de velden Naam en Plaats te vullen.

Naast de variabele REC is er een variabele P gedeclareerd. P is een pointer, bestemd om een willekeurig adres te bevatten.

Met de opdracht:

**P := @Rec;**

wordt het adres van de variabele REC in P gezet.

De volgende opdracht:

**P := REC;**

zet de inhoud van de variabele REC in het veld P.

Het is misschien verstandig om dit programma even door te lopen met de interne debugger om de inhoud van het veld P te controleren. Zoals je dan zult zien, wordt een pointer in twee delen weergegeven. De vorm is: het woord "Ptr" met daarachter twee haakjes. Tussen de haakje staan twee getallen, van elkaar gescheiden door een komma. De getallen representeren een waarde in het hexadecimale stelsel. Een adres kan er bijvoorbeeld zo uitzien: Ptr($6526,0). Het deel vóór de komma wordt het segmentdeel genoemd; het deel achter de komma het offsetdeel. Tezamen vormen ze een adres in het geheugen.

## 12.2 Seg, Ofs en Ptr

Een adres kunnen we uit elkaar peuteren in een segment- en een offsetdeel. Andersom kunnen we van een segment- en een offsetdeel ook weer een adres maken. Het programma GEH_3 demonstreert dit:

```
PROGRAM GEH_3;
USES CRT;

VAR
   GETAL: Integer;
   P    : Pointer;

BEGIN
   ClrScr;
   Writeln
   ('Het segmentdeel van GETAL is  :',Seg(GETAL));
   Writeln
   ('Het offsetdeel van GETAL is    :',Ofs(GETAL));
   P := Ptr(Seg(GETAL), Ofs(GETAL));
   Readln
END.
```

De Turbo Pascal-functies Seg en Ofs retourneren respectievelijk het segment- en het offsetdeel van het adres van de meegegeven parameter die in dit geval van het type Integer is. De functie Ptr maakt van een doorgegeven segment- en offsetdeel weer een adres. Als je via de debugger bekijkt welke waarde in P staat, zie je dat dit dezelfde waarde is als het adres van GETAL. Hier had dus ook kunnen staan:

```
P := @Getal
```

Dit lijkt misschien wel wat omslachtig, maar het gaat er om te demonstreren welke gereedschappen je als programmeur hebt om met geheugenadressen te werken.

## 12.3 Mem, MemW en MemL

In het geheugen van je computer zijn allerlei interessante zaken opgeslagen. Het besturingssysteem heeft daar allerlei gegevens staan die via vaste adressen te bereiken zijn. Soms heb je deze gegevens nodig en dan moet je in het geheugen kunnen kijken. Het kan soms nodig zijn in het geheugen te speuren naar zaken die te maken hebben met de uitvoering van je programma. Het kan ook zijn dat je gewoon zelf het geheugen wilt manipuleren.

Turbo Pascal biedt de mogelijkheid het geheugen rechtstreeks te benaderen. Hiervoor zijn een drietal array's voorgedefinieerd:

Mem, MemW en MemL. Mem is een array van het type Byte, MemW een array van het type Word en MemL een array van het type Longint.

De index van deze array's wordt op een bijzondere manier geschreven. In het hoofdstuk over typen hebben we gezien dat een index opgegeven wordt tussen vierkante haken, met daarachter een indexnummer. In dit geval bestaat de index tussen de vierkante haken uit twee delen: een segmentdeel en een offsetdeel. Deze delen worden van elkaar gescheiden door een dubbele punt.

Het programma GEH_4 laat zien hoe je een getal rechtstreeks in het geheugen kunt zetten:

```
PROGRAM GEH_4;
USES CRT;

VAR
   GETAL: Byte;

BEGIN
   GETAL := 0;
   Mem[Seg(GETAL):Ofs(GETAL)] := 82;
   ClrScr;
   Writeln('GETAL heeft nu een waarde van : ',GETAL);
   Readln
END.
```

De variabele GETAL is eerst op 0 gezet. Vervolgens wordt op het adres van GETAL rechtstreeks een waarde in het geheugen geschreven. Als daarna GETAL uitgelezen wordt, dan bevat GETAL de waarde die in het geheugen geschreven is.

Het programa GEH_5 laat zien dat het ook andersom werkt. Hier wordt niet in het geheugen geschreven, maar uit het geheugen gelezen:

```
PROGRAM GEH_5;
USES CRT;

VAR
   GETAL: Byte;

BEGIN
   GETAL := 82;
   ClrScr;
   Write('Op het adres van GETAL staat : ',
          Mem[Seg(GETAL):Ofs(GETAL)]);
   Readln
```

```
END.
```

In de Write-opdracht wordt met behulp van Mem rechtstreeks de waarde die op het aangegeven geheugenadres staat, uitgelezen. Ook hier blijkt dat de waarde in de variabele GETAL en de waarde die op het adres van GETAL staat, dezelfde is.

MemW en MemL worden op precies dezelfde wijze als Mem benaderd. Het enige verschil is dat Mem met het type Byte werkt, MemW met het type Word en MemL met het type Longint, zodat er dus respectievelijk één, twee of vier bytes benaderd worden.

## 12.4 Absolute variabelen

Soms kan het handig zijn om meerdere variabelen hetzelfde adres te geven. Het beschermde woord "Absolute" vestigt variabelen op hetzelfde adres. Het voordeel van het vestigen van meerdere variabelen op één adres is, dat als de waarde van de ene variabele verandert, de andere automatisch meeverandert. Het is een faciliteit die je waarschijnlijk niet zo vaak nodig zult hebben. Hij is echter te belangrijk om over te slaan. Het programma GEH_6.PAS laat zien hoe een en ander in zijn werk gaat:

```
PROGRAM GEH_6;
USES CRT;

CONST
   WOORDJE: Array[1..4] of Char=('G','E','I','T');


VAR
   GROOT_GETAL: Longint Absolute WOORDJE;
   TWEE_BYTES : Word Absolute WOORDJE;
   EEN_BYTE   : Byte Absolute WOORDJE;
   LETTER     : Char Absolute WOORDJE;

BEGIN
   ClrScr;
   Writeln('In WOORDJE staat   : ',WOORDJE);
   Writeln('In GROOT_GETAL staat: ',GROOT_GETAL);
   Writeln('In TWEE_BYTES staat : ',TWEE_BYTES);
   Writeln('In EEN_BYTE staat   : ',EEN_BYTE);
   Writeln('In LETTER staat    : ',LETTER);
   GROOT_GETAL := 1397051216;
   Writeln('Nadat GROOT_GETAL veranderd is
            zijn de waarden :');
   Writeln('In WOORDJE staat   : ',WOORDJE);
   Writeln('In GROOT_GETAL staat: ',GROOT_GETAL);
   Writeln('In TWEE_BYTES staat : ',TWEE_BYTES);
```

```
   Writeln('In EEN_BYTE staat   : ',EEN_BYTE);
   Writeln('In LETTER staat     : ',LETTER);
   Readln;
END.
```

De constante WOORDJE is getypeerd als Array [1..4] of Char. WOORDJE krijgt een beginwaarde mee. Ik heb hier de letters "G", "E", "I" en "T" gebruikt. Vervolgens worden een viertal variabelen op hetzelfde adres gevestigd. GROOT_GETAL is een longint en gebruikt vier bytes.

TWEE_BYTES is van het type Word en gebruikt, zoals de naam al aangeeft, twee bytes. EEN_BYTE en LETTER gebruiken beiden één byte. Alle variabelen delen hetzelfde adres. Als uit één van deze variabelen de waarde gelezen wordt, dan wordt er gestart met lezen op het adres van de getypeerde constante WOORDJE. Het aantal bytes dat vanaf dit adres wordt uitgelezen, hangt samen met het type dat gebruikt wordt. Voor de variabele LETTER wordt 1 byte uitgelezen, voor de variabele TWEE_BYTES worden 2 bytes gelezen. In het voorbeeld kun je zien dat de verandering in één van de variabelen, consequenties heeft voor alle andere variabelen die op hetzelfde adres gevestigd zijn.

## 12.5 Typecasting

Een belangrijke techniek bij het geavanceerd programmeren in Turbo Pascal is de toepassing van typecasting. Met typecasting krijg je de mogelijkheid om in het geheugen een bepaald adres uit lezen in de vorm van een gegevenstype. Vanaf een geheugenadres wordt als het ware een sjabloon over het geheugen gelegd. Vervolgens kan via dat sjabloon in het geheugen geschreven, of uit het geheugen gelezen worden. Het voordeel hiervan is dat je alleen maar de adressen van een gegevenstructuur hoeft te bewaren om de gegevens waar zij naar wijzen in de gewenste vorm te kunnen manipuleren. In het volgende programma wordt typecasting toegepast:

```
PROGRAM GEH_7;
USES CRT;

TYPE
   PInteger = ^Integer;

VAR
   LETTER: Char;
   GETAL : Word;
   P     : Pointer;

BEGIN
   ClrScr;
   LETTER := 'A';
```

```
    GETAL  := 65000;
    GetMem(P,2);
    PInteger(P)^ := 1000;
    Writeln('LETTER als Byte is    :',Byte(LETTER));
    Writeln('GETAL als Integer is  :',Integer(GETAL));
    Writeln('De pointer P wijst naar :',PInteger(P)^);
    FreeMem(P,2);
    Readln
END.
```

De variabele LETTER van het type Char en de variabele GETAL van het type Word, krijgen beide een waarde. In de Writeln-opdrachten wordt LETTER als een variabele van het type Byte uitgelezen door de typecasting Byte(LETTER). Het type Byte wordt nu als sjabloon gebruikt op het adres van LETTER. Het resultaat is een getal in plaats van een letter. Door het gebruik van een integer-sjabloon op het adres van de variabele GETAL, wordt de positieve waarde van GETAL een negatieve waarde.

Het gebruik van de pointer vereist extra toelichting. Door het gebruik van GetMem worden er twee bytes op de heap gereserveerd. Het adres wordt in P gezet. In het programma is PInteger gedefinieerd als een pointer naar Integer. Met de typecasting:

**PInteger(P)^ := 1000;**

wordt op het adres dat in P staat een waarde gezet. Hierbij doet PInteger dienst als sjabloon. Dit is een pointer naar Integer, dus PInteger(P)^ doet zich voor als een integer-sjabloon. Met PInteger(P)^ kan de locatie die aangegeven wordt door P, weer als een integer uitgelezen worden.

Typecasting werkt niet alleen met de binnen Turbo Pascal voorgedefinieerde typen, maar ook met de typen die je zelf in je programma definieert.

Ik hoor de vermoeide lezer, die het zo langzamerhand duizelt, al zeggen: "Ja prachtig hoor. Maar wat heb ik eraan?". Het antwoord luidt: "Hoe beter je het geheugen weet te gebruiken en naar je hand weet te zetten, des te flitsender zullen je programma's lopen".

## 12.6 Move

Tijdens het werken met grote hoeveelheden gegevens kan het gebeuren dat je van deze gegevens een hele of een gedeeltelijke kopie moet maken. Denk hiervoor bijvoorbeeld aan tekstbuffers waaruit je een bepaald gedeelte van de tekst wilt kopiëren. De Turbo Pascal-procedure Move kan zo'n klusje voor je uitvoeren. Deze procedure werkt razendsnel, maar kent ook een aantal

gevaren. Bij het gebruik van Move vallen allerlei beschermingen weg en kunnen zaken overschreven worden, terwijl dat helemaal de bedoeling niet was. Het programma GEH_8 laat zien op welke manier Move juist gebruikt wordt:

```
PROGRAM GEH_8;
USES CRT;

VAR
   ZIN   : String[50];
   RIJ   : Array[1..50] of Char;
   GETAL : Byte;
   LETTER: Char;


BEGIN
   ClrScr;
   FillChar(RIJ,SizeOf(RIJ),0);
   ZIN := 'Deze tekst wordt overgebracht naar RIJ';
   Move(ZIN[1],RIJ,SizeOf(ZIN)-1);
   Writeln('In RIJ staat : ',RIJ);
   GETAL := 65;
   Move(GETAL,LETTER,1);
   Writeln('In LETTER staat : ',LETTER);
   Readln
END.
```

In het hoofdstuk over typen heb ik bij de behandeling van strings verteld dat dit type beschikt over een lengtebyte. In String[0] wordt bijgehouden hoe lang de string is. Als deze lengtebyte verwijderd wordt, houd je een Array of Char over. In het voorbeeld wordt de tekst, die in de string ZIN staat, overgebracht naar de array RIJ. Move moet weten wat het beginpunt is waar vandaan gekopieerd moet worden, wat het beginpunt is waarheen gekopieerd moet worden, en om hoeveel bytes het gaat. Daarna wordt domweg het aantal opgegeven bytes van de bron naar de bestemming gekopieerd. Omdat ik de lengtebyte niet nodig had, is als beginpunt ZIN[1] opgegeven. Hiermee wordt de lengtebyte, die in ZIN[0] staat, niet gekopieerd.

Als het mogelijk is, gebruik dan SizeOf om de hoeveelheid bytes op te geven. Hiermee voorkom je dat je over de grenzen van de bron kopieert. Het is aan de programmeur om ervoor te zorgen dat de hoeveelheid bytes in de bestemming past.

Als in het voorbeeld de variabele GETAL de waarde 65 krijgt en we kopiëren één byte van GETAL naar LETTER, dan staat in LETTER een "A". Als we nu bij het aanroepen van Move niet één, maar twee bytes opgeven, dan wordt de byte die naast GETAL staat, gekopieerd naar de byte die naast LETTER staat. Hiermee kunnen waarden in het geheugen overschreven worden. Het gevolg is dan

een niet goed werkend programma met een moeilijk vindbare fout. Voorzichtigheid is dus geboden.

## 12.7 De heap

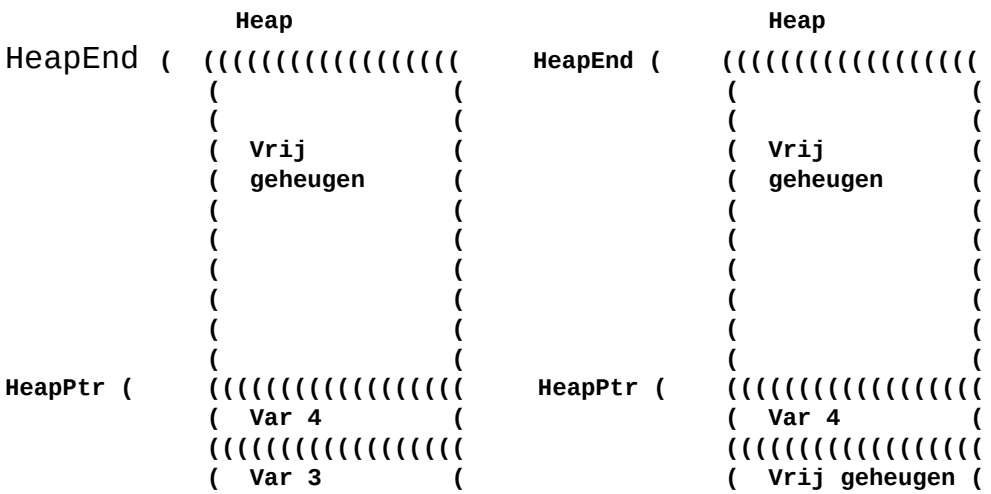
Het werken met dynamische variabelen lijkt erg ingewikkeld. Ik denk dat de zaak al een stuk eenvoudiger wordt als je begrijpt hoe de heap-manager zijn werk doet. Vandaar in deze paragraaf nog wat uitleg over het werken met de heap. Om de heap te beheren werkt Turbo Pascal met een drietal variabelen:

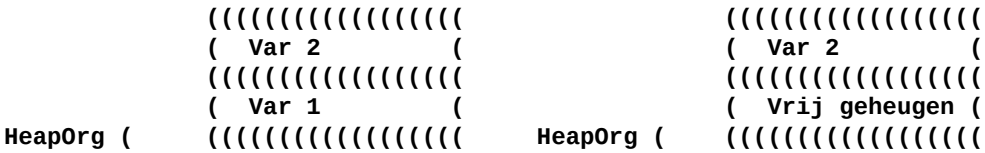
| Naam: | Betekenis: |
|---|---|
| HeapOrg | Wijst naar de bodem van de heap. |
| HeapEnd | Wijst naar het einde van de heap. |
| HeapPtr | Wijst naar het begin van het vrije geheugen. |

Iedere keer dat geheugen gereserveerd wordt voor een dynamische variabele, wordt door de heap-manager het adres in HeapOrg met het aantal benodigde bytes verhoogd. De feitelijke geheugentoewijzing is altijd een veelvoud van acht. Als je voor een variabele met een afmeting van 98 bytes geheugen reserveert, is de feitelijke geheugentoewijzing 104 bytes.

Het gereserveerde geheugen kan ook weer vrijgegeven worden. Hier doet zich een probleem voor. De volgorde waarin geheugen vrijgegeven wordt, hoeft niet noodzakelijkerwijs gelijk te zijn aan de volgorde waarin geheugen gereserveerd werd. Het gevolg is dat er gaten gaan vallen in het gedeelte van de heap, tussen de adressen waar HeapOrg en HeapPtr naar wijzen.

Misschien wordt het wat duidelijker als we er een plaatje van maken. In onderstaande situatie is in het linker voorbeeld voor een viertal variabelen geheugen gereserveerd op de heap. In het rechter voorbeeld is het geheugen voor Var 1 en Var 3 vrijgegeven. We zien nu gaten ontstaan in het deel van het geheugen onder HeapPtr:






Als de vrijgekomen geheugenblokken niet opnieuw gebruikt zouden kunnen worden, en de heap-manager dus telkens geheugen boven HeapPtr zou reserveren, dan zou de heap al snel vol zijn. Om dit hergebruik mogelijk te maken, houdt Turbo Pascal een administratie van vrijgekomen blokken bij.

De administratie heeft de vorm van een verbonden lijst. Hiervoor wordt vaak de Engelse term "linked list" gebruikt. Dit werkt als volgt: in de Turbo Pascal-variabele FreeList staat het adres van het eerste vrije blok geheugen. Als het adres in FreeList en in HeapPtr hetzelfde is, dan zijn er geen vrijgekomen geheugenblokken op de heap. FreeList wijst dan immers naar het begin van het vrije geheugen. Als er wel vrije blokken zijn, dan heeft de heap-manager op het adres waar FreeList naar wijst, de grootte van het vrije blok en het adres van het volgende blok gezet. In het volgende vrije blok staat weer het adres van het daarop volgende blok. Dit gaat zo door tot het adres van het volgende blok en het adres in HeapPtr gelijk zijn.

Als er nu opnieuw geheugen gereserveerd wordt, dan loopt de heap-manager eerst de vrijgekomen blokken geheugen af om te zien of de nieuw te reserveren ruimte in het vrije blok past. Als dit het geval is, wordt de lijst in orde gebracht en wordt HeapPtr niet opgehoogd.

Het volgende programma GEH_9 laat eerst zien op welke manier HeapOrg, HeapPtr en HeapEnd gebruikt worden. Daarna wordt er voor tien variabelen ruimte op de heap gereserveerd, waarvoor van vijf het geheugen weer vrijgegeven wordt. Het programma laat zien op welke manier de lijst met vrije blokken doorlopen kan worden en om hoeveel geheugen het gaat:

```
PROGRAM GEH_9;
USES CRT;

TYPE
   PWord  = ^Word;
   PArray = ^TArray;
   TArray =  Array[1..100] of Word;

   PVrijBlok = ^TVrijBlok;
   TVrijBlok = Record
      Volgende  : Pointer;
      Plus_Bytes: Word;
      Paragrafen: Word;
   END;


VAR
   RIJ       :  PArray;
   HEAPWIJZER:  Pointer;
   VRIJBLOK  :  PVrijBlok;

   FUNCTION Adres(P:Pointer):Longint;
   VAR
      SEGMENT: Longint;
      OFFSET : Word;

    BEGIN
       SEGMENT := LongInt(Seg(PWord(P)^)) * $10;
       OFFSET  := Ofs(PWord(P)^);
       ADRES   := SEGMENT + OFFSET
    END;

   PROCEDURE MaakGaten;
   VAR
      SERIE: Array [1..10] of PArray;
      I    : Byte;
   BEGIN
      FOR I := 1 TO 10 DO
      GetMem(SERIE[I],SizeOf(TArray));
      FOR I := 1 TO 10 DO
         IF Odd(I) THEN FreeMem(SERIE[I],SizeOf(TArray))
    END;

   PROCEDURE SchrijfGegevens(Rec:PVrijblok);
   BEGIN
      WITH REC^ DO
```

```
      Writeln('Op het adres ',Seg(Vrijblok^),':',
               Ofs(VrijBlok^):4,'  Afmeting Blok  : ',
              (Paragrafen * $10) + Plus_Bytes)
   END;

BEGIN
   ClrScr;
   Writeln('Heap Totaal   : ',
            ADRES(HeapEnd) - ADRES(HeapOrg));
   Writeln('Beschikbaar   : ',
            ADRES(HeapEnd) - ADRES(HeapPtr));
   Writeln('MemAvail zegt : ', MemAvail);
   Mark(HEAPWIJZER);
   GetMem(RIJ,SizeOf(TArray));
   GotoXY(1,6);

   Writeln('Beschikbaar   : ',
            ADRES(HeapEnd) - ADRES(HeapPtr));
   Writeln('MemAvail zegt : ', MemAvail);
   Writeln('Gereserveerd  : ',
            ADRES(HeapPtr) - ADRES(HeapOrg));
   Release(HEAPWIJZER);
   GotoXY(1,12);
   Writeln('Beschikbaar   : ',
            ADRES(HeapEnd) - ADRES(HeapOrg));
   Writeln('MemAvail zegt : ', MemAvail);
   MaakGaten;
   GotoXY(1,18);
   Writeln('Lege blokken na aanroep van MaakGaten:');
   VRIJBLOK := PVrijBlok(FreeList);
   IF VRIJBLOK <> HeapPtr THEN
   SchrijfGegevens(VRIJBLOK);
   WHILE VrijBlok^.Volgende <> HeapPtr DO
   WITH VrijBlok^ DO
   BEGIN
      VRIJBLOK := Volgende;
      SchrijfGegevens(VRIJBLOK)
   END;
   ReadKey
END.
```

**Regels: Toelichting:**

3-12Typeer een wijzer naar het type Word, een wijzer naar een array van 100 elementen van het type Word en een wijzer naar een record om gegevens uit de lijst van vrije geheugenblokken te lezen.
13-16Declareer variabelen.
**[1]17-25Function Adres(P:Pointer):Longint.**
22-24Reken het adres dat in de parameter P staat om naar een ongesegmenteerd adres.
**26-35Procedure MaakGaten.**
[9]31-32Reserveer voor tien arrays van 100 elementen van het type Word ruimte op de heap.
[10]33-34Geef van de variabelen met een oneven index de geheugenruimte weer vrij.
**36-42Procedure SchrijfGegevens(Rec:PVrijblok).**
[12]38-41Schrijf de gegevens uit de parameter Rec naar het scherm.
**43-76Hoofdprogramma.**
[2]45-46Schrijf de totale afmeting van de heap naar het scherm.
[3]47-49Toon het beschikbare geheugen met gebruik van HeapEnd en HeapOrg en met een aanroep van MemAvail.
[4]50 Zet het adres dat in HeapPtr staat in de variabele HEAPWIJZER.
[5]51-56Reserveer geheugen voor de variabele RIJ en laat zien dat HeapPtr verhoogd is.
[6]56-57Laat met behulp van HeapPtr en HeapOrg zien hoeveel geheugen er gereserveerd is.
[7]58 Geef het geheugen vrij.
[8]60-62Toon de beschikbare hoeveelheid geheugen.
[9]63 Roep de procedure MaakGaten aan.
[11]66 Zet het adres dat in FreeList staat in het veld VrijBlok.
[12]67-68Als FreeList ongelijk is aan HeapPtr, roep dan SchrijfGegevens aan.
[13]69-74Doorloop de lijst met vrije blokken geheugen en laat de gegevens op het scherm zien.

**Toelichting:**

[1]Om met de adressen in de pointers te kunnen rekenen, kun je ze het beste terugbrengen naar een ongesegmenteerd adres. De functie Adres neemt met behulp van Seg en Ofs de in de parameter P doorgekregen pointer uit elkaar. Door het segment met 16 te vermenigvuldigen en daar de offset bij op te tellen, wordt er een ongesegmenteerd adres geconstrueerd.

[2]De Turbo Pascal-variabele HeapEnd bevat een adres dat naar het einde van de heap wijst. In HeapOrg staat het adres van het begin van de heap. Als je het ongesegmenteerde adres in HeapOrg van het ongesegmenteerde adres in HeapEnd aftrekt, dan krijg je als uitkomst de omvang van de heap in bytes.

[3]Als HeapEnd en HeapPtr van elkaar afgetrokken worden, dan krijg je als uitkomst de hoeveelheid beschikbaar geheugen in bytes. Om te controleren of de berekening klopt, wordt vervolgens MemAvail aangeroepen. Omdat er nog geen geheugen is vrijgegeven, geeft MemAvail dezelfde waarde aan als HeapEnd minus HeapPtr.

[4]Naast de mogelijkheid om het geheugen op de heap voor iedere variabele afzonderlijk vrij te geven, kun je ook een heel blok geheugen tegelijkertijd vrijgeven. Om dit te kunnen doen moeten we het adres dat in HeapPtr staat bewaren. De procedure Mark zet het adres van HeapPtr in de variabele HEAPWIJZER.

[5]Daarna wordt er ruimte geclaimd op de heap en wordt aangetoond dat het veld HeapPtr verhoogd is, omdat de aftreksom HeapEnd min HeapPtr nu een andere uitkomst heeft dan de eerste keer. Ook deze uitkomst wordt door MemAvail bevestigd.

[6]Door HeapOrg nu van HeapPtr van elkaar af te trekken, krijgen we de hoeveelheid gereserveerd geheugen als uitkomst.

[7]Tot nu toe hebben we ruimte op de heap vrijgegeven met Dispose of met FreeMem. Je kunt ook een heel blok geheugen vrijgeven. Voor er ruimte voor de variabele RIJ gereserveerd werd, hebben we met een aanroep van Mark het adres in HeapPtr in HEAPWIJZER gezet. Als we nu de Turbo Pascal-procedure Release aanroepen en we geven HEAPWIJZER als parameter mee, dan wordt HeapPtr op het adres dat in HEAPWIJZER staat gezet. Het beheer van de heap met Mark en Release werkt ongenuanceerd, omdat er geen rekening gehouden wordt met vrijgekomen blokken. Ook bestaat de mogelijkheid dat gegevens die je nog nodig hebt, niet meer bereikbaar zijn. Gebruik deze werkwijze daarom in principe niet.

[8]Een berekening van het vrije geheugen op de heap laat zien dat in dit geval het geheugen op een goede manier is vrijgegeven.

[9]Om te laten zien op welke manier de lijst met vrije blokken geheugen bereikt kan worden, moeten er eerst wat vrije blokken tussen HeapOrg en HeapPtr gemaakt worden. De procedure MaakGaten heeft een lokale variabele SERIE. SERIE is een array van tien elementen van het type PRij. PRij hadden we gedefinieerd als een wijzer naar een array van honderd elementen van het type Word. De For-lus die doorlopen wordt, reserveert bij elke doorgang 200 bytes op de heap. Deze zijn bereikbaar met SERIE[I]^.

[10]Om nu gaten te maken, wordt het geheugen voor ieder element waarvan de index oneven is, vrijgegeven. Om te bepalen welke elementen dan vrijgegeven moeten worden, wordt opnieuw een For-lus doorlopen. Met behulp van de Turbo Pascal- procedure Odd wordt gekeken of I een even of oneven waarde heeft. Retourneert Odd de waarde True, dan wordt het

geheugen voor dat element vrijgegeven. Op deze manier onstaan er vijf blokken op de heap met vrijgegeven geheugen. Elk blok is 200 bytes groot.

[11]Het record VrijBlok^ wordt gebruikt om de gegevens uit de lijst te lezen. Het adres, dat in FreeList staat, wordt aan VRIJBLOK gegeven. Als VRIJBLOK een ander adres dan HeapPtr heeft, staan er vrijgekomen blokken op de heap.

[12]VRIJBLOK wordt naar SchrijfGegevens gestuurd. Deze procedure laat zien op welk gesegmenteerd adres de gegevens staan en hoe groot het vrije blok is. Het veld Paragrafen van het record geeft het aantal paragrafen van 16 bytes aan, en het veld Plus_Bytes het aantal bytes dat daar bij komt. De uitkomst van "Paragrafen * 16 + Plus_Bytes" is dus de afmeting van het vrije blok.

[13]Als het adres van VrijBlok^.Volgende anders is dan HeapPtr, wordt een While-lus ingegaan. In deze lus krijgt VRIJBLOK het adres dat in het veld Volgende staat. De variabele VRIJBLOK wijst nu naar een nieuw vrij geheugenblok. Deze gegevens worden met SchrijfGegevens weer op het scherm gezet. Er wordt teruggekeerd in de lus, en als het veld Volgende niet gelijk is aan HeapPtr, krijgt VRIJBLOK het adres van het veld Volgende.

## 12.8 Opgaven

12.1Maak een programma waarin een string van 15 lettertekens is gedeclareerd. Geef deze string een beginwaarde en laat de string dan verticaal op het scherm verschijnen door het rechtstreeks uitlezen van het geheugen. Maak hierbij gebruik van typecasting.

```
--------------------
209

          209

          209
```