Week 5 : Hoofdstuk 11+ e**X**tra stof: meer over functies

Hoofdstuk 11: Functies

Functies in C lijken heel erg op functies in Java. Maar er is **één groot veschil**: Java ken uitsluitend ***invoer-parameters***, terwijl C zowel ***invoer-parameters*** als ***uitvoer-parameters*** kent.

In Java levert de functie **ingelezen of berekende waarden** af via de return-waarde. In C kan dat ook op die manier. Maar het kan ook op een andere manier, nl. via een zgn. ***uitvoer-parameter***. En dat mechanisme werkt via pointers.

Java-programma's zijn Object geOriënteerd. Dat betekent dat de **gegevens centraal** staan, en dat de bewerkingen van die gegevens daaraan ondergeschikt zijn. Java-programma's bestaan uit een aantal klassen.

C is een zgn. ***procedurele taal***. Bij procedurele talen zijn de bewerkingen (procedures) het uitgangspunt. Procedurele programma's **bestaan uit een aantal functies**.

Er is altijd één functie `int main(void)`; daarnaast kunnen er andere functies zijn. De uitvoering van een C-programma begint altijd met de uitvoering van de functie `main()`. Vanuit `main()` kunnen dan andere functies worden aangeroepen.

Elke functie bestaat uit een ***functie-declaratie*** en een ***functie-definitie***. Een functie-declaratie bestaat uit ***de functie-naam***, het ***return-type*** en de ***parameters*** (aantal en type; niet noodzakelijk ook hun naam). Dit geheel heet ook wel de ***functie-heading***. Een functie-***definitie*** bestaat uit een functie-***heading*** en een functie-***body***. De functie-***body*** is alles vanaf de openingsaccolade { t/m de sluit-accolade }. De functie-***body*** bestaat uit alle ***lokale declaraties*** en de ***statements*** van de functie.

➔
Wanneer binnen een functie een andere functie wordt aangeroepen, **moet** op het moment van aanroep de functie-***declaratie*** hebben plaats gevonden. Die functie-***declaratie*** mag plaats vinden **binnen de functie `main()`**. Dat is toegestaan in C, maar niet gebruikelijk. Wij zetten de functie-declaratie altijd neer op de gebruikelijke plaats.

Voor de ***functie-definitie*** zijn er twee mogelijkheden :

- ofwel tegelijk met de functie-declaratie
- ofwel in een later stadium.

Voor de functie-***definitie*** is het verplicht om de parameters een naam te geven. Wanneer de functie-***declaratie*** op een andere plek staat dan de functie-definitie, is het niet verplicht om de parameters een naam te geven: het **mag** wel, maar het **hoeft** niet. Het is gebruikelijk om in dat geval de parameters **geen naam** te geven.

Voorbeeld 11_1

Dit is een heel eenvoudig programma, dat is bedoeld om te wennen aan de manier waaropin C met functies wordt gewerkt. Dit is voorbeeld 7_1 uit Les 3

```
#include <stdio.h>

void druk_af(char, int);                    // functie-declaratie

int main(void) {

  druk_af('*', 5);                          // functie-aanroep
  return 0;
}

void druk_af(char ch, int aantal)           // functie-definitie
{
  int regel, spaties, teken;
  for (regel = 1 ; regel <= aantal ; regel++){
    for (spaties = 1 ; spaties <= aantal-regel ; spaties++) {
      putchar(' ');
    }
    for (teken = 1 ; teken <= 2*regel-1 ; teken++){
      putchar(ch);
    }
    putchar('\n');
  }
}
```

Uitvoer:

```
    *
   ***
  *****
 *******
*********
```

Voorbeeld 11_2
Dit is een voorbeeld van een functie met ***invoer-parameters*** (van type **`int`**).
De namen ***invoer-parameter*** / ***call-by-value*** / ***value-parameter*** betekenen allemaal hetzelfde.

```
#include <stdio.h>

void verwisselFOUT(int, int);                    // prototype declaratie

int main(void)
{
  int a, b;
  printf("Tik waarde in voor a: ");
  scanf("%d", &a);
  printf("Tik waarde in voor b: ");
  scanf("%d", &b);
  printf("a = %d en b = %d\n", a, b);

  verwisselFOUT(a, b);                           // functie aanroep
  printf("Na verwisselen geldt a = %d en b = %d\n", a, b);
  return 0;
}

void verwisselFOUT(int x, int y)                 // functie definitie
{
  int h = x; x = y; y = h;
}
```

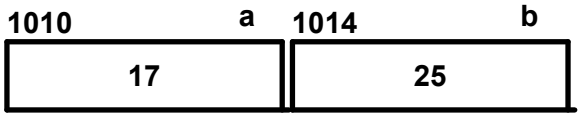
Uitvoer:
Tik waarde in voor a: 17 <ENTER>
Tik waarde in voor b: 25 <ENTER>
a = 17 en b = 25
Na verwisselen geldt a = 17 en b = 25

Opmerking:
Dit gaat fout, omdat parameters `x` en `y` ***invoer-parameters*** zijn en geen ***uitvoer-parameters***. ***Invoer-parameters*** heten ook wel ***value-parameters*** omdat ze bij aanroep van de functie een ***waarde*** (***value***) moeten hebben. In de **aanroep**

```
  verwisselFOUT(a, b);
```

heeft parameter `a` de ***waarde*** 17 en parameter `b` de ***waarde*** 25.

functie main( )

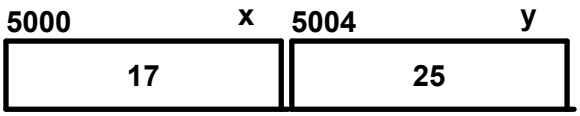


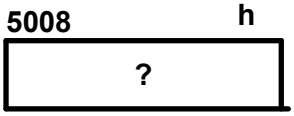
AANROEP : verwisselFOUT(17 , 25);

AANROEP : verwisselFOUT(17 , 25); is klaar

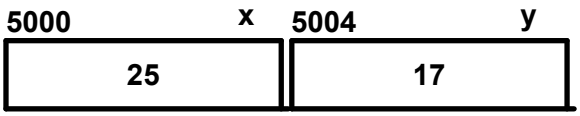
functie verwisselFOUT(int x , int y)

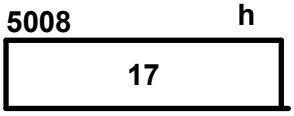
x B.N. en y B.N. en h B.N.





h = x ; x = y ; y = h;





x B.N. en y B.N. en h B.N.

Er verandert **niets** aan de ***waarden*** van variabelen `a` en `b`.

De ***waarden*** van variabelen `x` en `y` veranderen **wel**, maar binnen `main()` geldt:

voor de aanroep `verwisselFOUT(a, b);` bestaan de variabelen `x`, `y` en `h` niet (B.N.)

bij de aanroep `verwisselFOUT(a, b);` gaan de variabelen `x` en `y` bestaan en wordt de waarde van `a` gekopieerd naar `x` en de waarde van `b` gekopieerd naar `y` (resp. 17 en 25). En variabele `h` gaat bestaan. Dan wordt **`h = x; x = y; y = h;`** uitgevoerd.

na de aanroep bestaan de variabelen `x`, `y` en `h` niet (B.N.)

Voorbeeld 11_3

Dit is een voorbeeld van een functie met ***uitvoer-parameters*** (van type **`int`**).
De namen ***uitvoer-parameter*** en ***var-parameter*** betekenen hetzelfde.

```
#include <stdio.h>

void verwisselGOED(int*, int*);

int main(void)
{
  int a, b;
  printf("Tik waarde in voor a: ");
  scanf("%d", &a);
  printf("Tik waarde in voor b: ");
  scanf("%d", &b);
  printf("a = %d en b = %d\n", a, b);

  verwisselGOED(&a, &b);
  printf("Na verwisselen geldt a = %d en b = %d\n", a, b);
  return 0;
}

void verwisselGOED(int* px, int* py)
{
  int h = *px; *px = *py; *py = h;
}
```

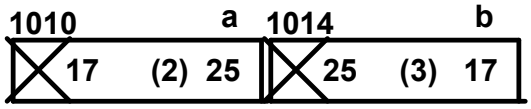
Uitvoer:

Tik waarde in voor a: 17 ‹ENTER›
Tik waarde in voor b: 25 ‹ENTER›
a = 17 en b = 25
Na verwisselen geldt a = 25 en b = 17

Opmerking:

Dit gaat goed, omdat parameters `px` en `py` ***uitvoer-parameters*** zijn en geen ***invoer-parameters***.

functie main( )

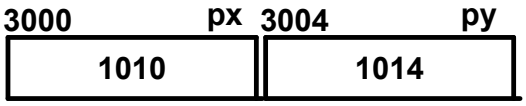


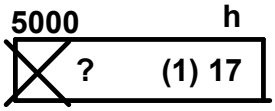
AANROEP : verwisselGOED(1010 , 1014);

AANROEP : verwisselGOED(1010 , 1014);
is klaar

functie verwisselGOED(int * px , int * py)

px B.N. en py B.N. en h B.N.





(1) h = * px ;
? wordt 17, want * px is 17

(2) * px = * py ;
*** px is de inhoud van a uit main() en die a uit main() wordt * py, dwz a wordt 25**

(3) * py = h;
*** py is de inhoud van b uit main() en die b wordt h, dwz b wordt 17**

px B.N. en py B.N. en h B.N.

Bij de functie aanroep

```
verwisselGOED(&a, &b);
```

worden nu de adressen van de variabelen `a` en `b` gekopieerd naar de parameters `px` en `py`. Daardoor is het mogelijk om vanuit de functie de waarde van `a` (binnen de functie aanspreekbaar via `* px`) en de waarde van `b` (binnen de functie aanspreekbaar via `* py`) te wijzigen.

Voorbeeld 11_4

De invoer van onderstaand programma bestaat uit twee `int`-getallen, gescheiden door een ***white-space character*** (ws-char). Het eerste getal is het dag-nummer en het tweede getal is het maand-nummer
(dus 11 9 ‹ENTER› leidt tot: 11 september).
De uitvoer van het programma is dan de datum van morgen (voor het vb. is dat 12/9 : voor 12 september). Merk op dat de maanden maart, mei, juli, augustus en oktober allen 31 dagen tellen.

```
#include <stdio.h>

#define JAN 1
#define FEB 2
#define APR 4
#define JUN 6
#define SEP 9
#define NOV 11
#define DEC 12

void datum_van_morgen(int *, int *);

int main(void)
{
  int dag, maand;
  printf("Tik in dagnummer, ws, en maandnummer: ");
  scanf("%d %d", &dag, &maand);          // controle ontbreekt !!

  datum_van_morgen(&dag, &maand);
  printf("Datum van morgen is %d/%d\n", dag, maand);
  return 0;
}
```

Uitvoer1:

Tik in dagnummer, ws, en maandnummer: 30 9 ‹ENTER›
Datum van morgen is 1/10

Uitvoer2:

Tik in dagnummer, ws, en maandnummer: 29 2 ‹ENTER›
Datum van morgen is 30/2

```
void datum_van_morgen(int * pd, int * pm)
{
  int dagen_in_maand = 0;
  switch(*pm)
  {
    case APR :
    case JUN :
    case SEP :
    case NOV : dagen_in_maand = 30; break;
    case FEB : dagen_in_maand = 28; break;
    default  : dagen_in_maand = 31; break;
  }

  if ( *pd == dagen_in_maand )   // *pd is de laatste dag van de maand
  {
    *pd = 1;
    if ( *pm == DEC ) *pm = JAN; else (*pm)++;
  }
  else                           // *pd is niet de laatste dag van de maand
    (*pd)++;
}
```

➔

Voorbeeld 11_5

In dit voorbeeld is sprake van een array van `float`s. De array staat gedeclareerd binnen `main()` (en daar wordt dus ook de **geheugenruimte aangevraagd**). Vervolgens wordt het kleinste getal in de array bepaald en getoond op scherm.

```
#include <stdio.h>
#define GROOTTE     100

void invoer(float ar[], const int limiet, int * pi);
float* kleinste(float * ar, const int n);
void uitvoer(float * ar, const int n);

int main(void) {
  float array[GROOTTE];
  int aantal = -1;
  invoer(array, GROOTTE, &aantal);
  if (aantal <= 0)
    printf("invoeren van getallen mislukt\n");
  else {
    printf("Kleinste is %.2f\n", *kleinste(array, aantal));
    uitvoer(array, aantal);
  }
  return 0;
}

float * kleinste(float * ar, const int n)
{
  int index = 0;
  float minst = ar[0];
  for (int k = 1 ; k < n ; k++)
  {
    if ( ar[k] < minst )
    {
      minst = ar[k];
      index = k;
    }
  }

  return ar + index;
}
```

```
void invoer(float * ar, const int limiet, int * pi)
{
  *pi = 0; int k;
  printf("Voer de te verwerken getallen in; sluit met #:\n");
  for (k = 0 ; k < limiet && scanf("%f", ar + k) ; k++)
  {   }
  *pi = k;
}

void uitvoer(float * ar, const int n)
{
  printf("Getallen in de array:\n");
  for (float* pi = ar ; pi < ar + n ; pi++)
  {
    printf("%7.2f\n", *pi);
  }
}
```

Uitvoer:

Voer de te verwerken getallen in; sluit af met #:
1234.56789 -98765.1234 #<ENTER>
Kleinste is -98765.13
Getallen in de array:
1234.57
-98765.13

Opmerkingen:

- Een array *gedraagt* zich in expressies als een pointer, maar een array *is* geen pointer.
  Toegestaan is: `pf = ar + 1;` (een pointer is een *lvalue*)
  NIET MAG: `ar = pf;` (een array is geen *lvalue*)
- **Uitsluitend & alleen** voor *formele* parameters van een functie geldt dat er geen enkel verschil bestaat tussen `float* ar` en `float ar[]`
  Dat is logisch omdat bij aanroep van de functie het begin-adres van de opgegeven array wordt doorgegeven, zonder controle op het aantal aangevraagde geheugenplaatsen (het is de taak van de programmeur om er voor te zorgen dat er wordt beveiligd ten het uit-de-array-lopen; zie in de code van functie `kleinste()` de meegegeven parameter `const int n`).

Voorbeeld 11_6: Datum met de ***pijltjes-operator***

```
#include <stdio.h>

typedef struct datum {
  int dag;
  int maand;
  int jaar;
} Datum;

void druk_af(Datum dat) {
  printf("%2d-%2d-%4d\n",dat.dag,dat.maand,dat.jaar);
}

int main(void) {
  Datum dm = {4, 12, 2006};
  Datum kerst = {25, 12, 2006};
  druk_af(dm);                    //  4-12-2006
  druk_af(kerst);                 // 25-12-2006

  Datum *dp = &dm;
  (*dp).dag = (*dp).dag + 1;
  (*dp).maand = 3;
  (*dp).jaar = 1999;
  druk_af(dm);                    //  5- 3-1999
  return 0;
}
```

Opmerking:
De onderstaande notatie leest niet zo prettig:

```
  (*dp).dag = d;
  (*dp).maand = m;
  (*dp).jaar = j;
```

Om dat probleem te omzeilen, bestaat de zogenaamde ***pijltjes-operator***:

```
  dp -> dag = d;
  dp -> maand = m;
  dp -> jaar = j;
```

Voorbeeld 11_7: inlezen op de C-manier

```
#include <stdio.h>

typedef struct {
  int dag;
  int maand;
  int jaar;
} Datum;

void lees_datum(char *prompt, Datum * dat) {
  int d, m, j;
  printf("%s (dd/mm/jjjj) ", prompt);
  scanf("%d/%d/%d", &d, &m, &j);

  (*dat).dag = d; (*dat).maand = m; (*dat).jaar = j;
}

int main(void) {
  Datum geboortedatum, vandaag;
  int leeftijd;

  lees_datum("Geboortedatum: ", &geboortedatum);
  lees_datum("Datum van vandaag: ", &vandaag);

  if ((vandaag.maand > geboortedatum.maand) ||
       (vandaag.maand == geboortedatum.maand &&
              vandaag.dag > geboortedatum.dag))
  {
    leeftijd = vandaag.jaar - geboortedatum.jaar;
  }
  else
  {
    leeftijd = vandaag.jaar - geboortedatum.jaar - 1;
  }
  printf("leeftijd: %d\n", leeftijd);
  return 0;
}
```

Uitvoer:

Geboortedatum: (dd/mm/jjjj) 28/11/1949 ‹ENTER›
Datum van vandaag: (dd/mm/jjjj) 4/12/2006 ‹ENTER›
leeftijd: 57

Andere functies met karakterstrings

```
int gelijk(const char *een, const char *twee)
{
  while ( *een == *twee )
  {
    if ( *een == '\0' )
      return 1;
    een++;
    twee++;
  }
  return 0;
}
```

Inlezen van een regel tekst, afgesloten door een ENTER:

```
char zin[100];
char c;
int teller = 0;
char *p = zin;
while ( (c = getchar()) != '\n' )
{
  teller++;
  if ( teller < 100 )
    *p++ = c;
}
*p = '\0';
printf("%s\n", zin);
```

// Waarom kun je hier niet "lopen" met `zin` ?

Inlezen met `scanf()`:

```
char zin[100];
scanf("%s", zin);
printf("%s\n", zin);
```

De invoer is : De blaadjes vallen op de rails <ENTER>
Wat is de uitvoer ?

Opmerkingen:

- In de code
  ```
  scanf("%s", zin);
  ```
  staat **geen &** voor **zin** . Dat is niet nodig, omdat in **zin** in **expressies** wordt beschouwd als variabele van type **char ***

- Bij de conversiespecificatie **%s** geeft een ws-char (white-space character: een spatie, een ENTER of een TAB) het einde van de invoer aan. In het voorbeeld onderaan blz 13 wordt dus alleen het eerste woord gelezen.

- Voor het lezen van een hele regel tekst is **scanf** dus niet geschikt. Je kunt karakter voor karakter lezen, zoals op blz 6 is gedaan. Maar het kan ook sneller, namelijk met de functie **gets** (maar die is niet veilig) , en liever nog met de functie **fgets** (die is wel veilig). Dat gaat aldus:
  ```
  char zin[100];
  gets(zin);
  ```
  Dit gaat alleen goed, als we er zeker van zijn dat de gebruiker niet meer dan 99 karakters intikt voordat er op de ENTER wordt gedrukt (99, omdat **gets** aan het einde zelf nog het nulkarakter '\0' toegevoegt; maar daar moet dan wel plaats voor zijn).

- Er is een veiliger variant nl met fgets Die gaat als volgt:
  ```
  char zin[100];
  fgets(zin, 100, stdin);
  int len = strlen(zin);      // deze len is 1 te groot
  zin[len - 1] = '\0';
  ```
  Bij inlezen met fgets gebeurt het volgende: er worden maximaal 98 karakters opgeslagen; daarachter komt een '\n'; en daarachter komt een '\0'. Vervolgens moet die '\n' weer worden verwijderd. Dat gebeurt door een '\0' over die '\n' heen te plaatsen.

Functies uit de standaardbibliotheek `string.h` (blz 542 e.v.)

- In de bibliotheek **`string.h`** is een standaardfunctie aanwezig om de lengte van een string uit te rekenen. De functie

```
int strlen(const char *s)
```

levert als resultaat af de lengte van string `s`.
Bijvoorbeeld:

```
char zin[] = "Goede morgen";
int len = strlen(zin);
printf("%d", len);                    // er wordt 12 afgedrukt
```

- Om twee strings met elkaar te vergelijken, kun je gebruik maken van de functie `strcmp` met:

```
int strcmp(const char *s1, const char *s2)
```

Zo is bijvoorbeeld

```
( strcmp("Rotterdam", "amsterdam") < 0 )
```

waar (omdat de `'R'` lexicografisch kleiner is dan de `'a'`).

En

```
( strcmp("Het regent", "Het regent") == 0 )
```

is ook waar, omdat de twee strings identiek zijn.

- Wanneer in een programma met string-invoer van een afsluiter gebruik wordt gemaakt, moet je dus met `strcmp` hierop testen:

```
char zin[128];
int aantal_regels = 0;
int OK = 1;
while ( OK && fgets(zin, 128, stdin) != NULL )
{
  if ( strcmp(zin, "stoppen\n") == 0 )

      // nu is het einde van de invoer bereikt;
      // OK = 0;

}
```

- De toekenning:
```
char woord[] = "hoera";
char* p = woord;
```
leidt **niet** tot het makenvan een kopie van het woordje "hoera".

Om wel een kopie te maken, moet je de functie `strcpy` gebruiken.
```
char* strcpy(char *dest, const char *s)
```

Deze functie kopiëert `s` (de source) naar `dest` (de destination). Er wordt karakter voor karakter gekopiëerd, net zolang tot in `s` het nul-karakter wordt aangetroffen. Je moet er zelf voor zorgen dat er voldoene ruimte is in `dest`. Maar dat alleen is niet voldoende. Zie onderstaand voorbeeld:

- Fout Voorbeeld:
```
char* fout()
{
  char str[100];
  strcpy(str, "ABC");
  return str;
}
```

- Maar de mogelijkheden zijn ruimer:
```
char s[100] = "Programmeer eens wat in C.";
char t[100];
strcpy(t, s);
strcpy(t + 12, "in Java." );
```
`t` is dan geworden: "Programmeer in Java"

- En met:
```
char* strcat(char *dest, const char *s)
```
kan bijvoorbeeld dit:
```
char naam[100];
char voor[30] = "Sylvester";
char midden[10] = " van der ";
char achter[40] = "Broek";
strcpy(naam, voor);
strcat(naam, midden);
strcat(naam, achter);
```
Variabele `naam` is dan geworden: "Sylvester van der Broek"

Conversie tussen strings en numerieke waarden

In `stdlib.h` staat de functie: `int atoi(const char *s)`.
Deze functie converteert de string, waar `s` naar wijst, naar een integer. Je kunt dit ook zelf programmeren, namelijk:

```
int atoi(char s[]) {
  int n = 0; int stoppen = 0;
  for (int i = 0 ; !stoppen ; i++)
    if (s[i] < '0' || s[i] > '9')
      stoppen = 1;
    else
      n = n*10 + (s[i] - '0');
  return n;
}
```

- Voor dit **omzetten** van string naar getal, en andersom, bestaan ook de functies `sscanf` en `sprintf`

  Vb1
  ```
  int getal;
  scanf("%d", &getal);              // leest waarde voor getal in
  ```

  Vb2
  ```
  int getal;
  char *p = "123 is een geheel getal";
  getal = atoi(p);                  // maakt getal 123
  sscanf(p, "%d", &getal);          // maakt ook getal 123
  ```

  Vb3
  ```
  int getal = 5634;
  char s[10];
  sprintf(s, "%d", getal);          // maakt s = "5634"
  ```

Opgave 5_1

Schrijf een functie `lees_tel_en_bereken_gem()` of `LTEBG()`, waarin een (onbekend) aantal positieve gehele getallen moet worden ingelezen. Het getal 0 is de afsluiter (dat betekent dat de invoer stopt door het intikken van 0, en dat getal 0 zelf niet mee doet). De functie telt het aantal ingelezen getallen en berekent het gemiddelde. De ingelezen getallen hoeven niet te worden bewaard, en je mag geen array gebruiken.
Schrijf ook een functie `main()` waarin de functie `LTEBG()` wordt aangeroepen.

Uitwerking Opgave 5_1

De functie `lees_tel_en_bereken_gem()` leest de getallen in, en **telt** hoeveel dat er waren. Dat aantal moet dus uit de functie worden uitgevoerd naar "buiten": dus ***uitvoer-parameter***. En ook **berekent** de functie het rekenkundig gemiddelde van de ingevoerde getallen. Dat wordt dus de tweede ***uitvoer-parameter***.

```
#include <stdio.h>

void lees_tel_en_bereken_gem(int*, float*);              // let op *

int main(void)
{
  int aantal = 0;            // hier wordt geheugenruimte aangevraagd
  float gemiddelde = 0.0;                                  // idem
  lees_tel_en_bereken_gem(&aantal, &gemiddelde);
  printf("Er zijn %d positieve int\'s ingelezen met
                          gem = %.3f\n", aantal, gemiddelde);
  return 0;
}

void lees_tel_en_bereken_gem(int* pa, float* pg)       // let op *
{
  printf("Tik getallen in; sluit af met een 0:\n");
  int getal = 1;                                  // voorlopige versie
  *pa = 0;
  float som = 0.0;
  while (getal != 0)
  {
    scanf("%d", &getal);
    if (getal <= 0) continue;        // (c) negeer foute invoer; en ga door
    som += getal;
    (*pa)++;
    // (c) als het getal <= 0, dan "springt" hij naar dit punt door continue
  }
  if (*pa == 0)
    *pg = 0.0;
  else
    *pg = 1.0*som/(*pa);       // *pg moet float zijn; met die 1.0 altijd OK
}
```

Opmerking:

Operator `++` "wint" van ***operator*** `*` en dus moet `(*pa)++;` (!= `*pa++;`)

Extra aanvulling op het boek:

Voorbeeld Extra_1 (geen tentamenstof)
Je kunt functies ***parameters*** meegeven ***met een default-waarde***. Hieronder volgt een voorbeeld voor de functie `int discriminant(int, int, int).`
In het voorbeeld hebben alle ***drie*** de ***parameters een dfefault-waarde*** gekregen. Je had ook kunnen kiezen voor 2 parameters met een default-waarde (dat hadden dan de achterste twee **moeten** zijn) of voor 1 parameter met een default-waarde (dat had dan de allerachterste **moeten** zijn).

```
#include <stdio.h>

int discriminant(int = 1, int = 1, int = 1);      // default-waardes

int main(void) {
  int a, b, c;
  printf("tik in a, b en c: ");
  scanf("%d %d %d", &a, &b, &c);

  printf("De discriminant met 0 defaults: %d\n",
                                        discriminant(a, b, c));
  printf("De discriminant met 1 default: %d\n",
                                        discriminant(a, b));
  printf("De discriminant met 2 defaults: %d\n",
                                        discriminant(a));
  printf("De discriminant met 3 defaults: %d\n",
                                        discriminant());
}

int discriminant(int aa, int bb, int cc)   // hier mogen ze dan niet
{
  int dis = bb*bb - 4*aa*cc;
  printf("%d * %d - 4 * %d * %d = ", bb, bb, aa, cc);
  return dis;
}
```

Uitvoer:
tik in a, b, c: 2 3 5<ENTER>
3 * 3 – 4 * 2 * 5 = De discriminant met 0 defaults: -31
3 * 3 – 4 * 2 * 1 = De discriminant met 1 default: 1
1 * 1 – 4 * 2 * 1 = De discriminant met 2 defaults: -7
1 * 1 – 4 * 1 * 1 = De discriminant met 3 defaults: -3

Voorbeeld Extra_ 2 (geen tentamenstof)

Dit is een voorbeeld met ***boolean conditie*** 's. Bij een ***if-statement*** is altijd sprake van een ***boolean expressie***; bij lussen eveneens. Een enkelvoudige expressie levert zelden problemen op. Dit voorbeeld gaat over samengestelde boolean expressies.

Wetten van de Morgan:

not (P **and** Q) = not P **or** not Q
(lees dit als (not P) **or** (not Q) omdat not "wint" van de or)

not (P **or** Q) = not P **and** not Q
(lees dit als (not P) **and** (not Q) omdat not "wint" van de and)

VbA: deze is OK (je test op teken '0' of '1')

```
char ch; printf("tik een karakter in: "); ch = getchar();
if ( ch == '0' || ch == '1' )    // P = ( ch == '0' )   Q = ( ch == '1' )
  printf("Yes! %c is een 0 of een 1\n", ch);
else                             // not ( P or Q ) betekent: ( ch != '0' )  and ( ch != '1' )
  printf("No! %c is geen 0 of 1\n", ch);
```

VbB: deze is **niet OK** (ch kan niet tegelijkertijd een '0' en een '1' zijn)

```
char ch; printf("tik een karakter in: "); ch = getchar();
if ( ch == '0' && ch == '1' )    // P = ( ch == '0' )   Q = ( ch == '1' )
  printf("Hier kom je nooit\n");
else                             // not ( P and Q ) betekent: ( ch != '0' )  or ( ch != '1' )
  printf("No! %c is geen 0 of 1\n", ch);
```

Opm: ook voor ch == '0' en ch == '1' staat er: No! 1 is geen 0 of 1

VbC1: dit is het vervolg van A (je test op teken '0' of '1')

```
char ch; printf("tik een karakter in: "); ch = getchar();
if ( ch == '0' || ch == '1' )
{
  if ( ch == '0' )
    // verwerk de '0'
  else  // dan is dit de '1'
    // verwerk de '1'
}
else
    // deze invoer wil je niet; verwerk in één klap alle ongewenste invoer
```

VbC2: dit is een ander vervolg van A (je test op teken '0' of '1')

```
char ch; printf("tik een karakter in: "); ch = getchar();
```

```
if ( ch == '0' )              // dit is de '0' ; die is hiermee klaar
   // verwerk de '0'
else if ( ch == '1' )         // dan is dit alles != '0' ; het is zelfs de '1'
   // verwerk de '1'
else                          // dit was al alles != '0' ; het is ook nog != '1'
   // deze invoer wil je niet; verwerk in één klap alle ongewenste invoer
```

VbD: Er moet een `int`-getal worden verwerkt, dat moet liggen tussen 0 en 100 (`int getal;` met 0 <= `getal` <= 100)

Strategie:
GOED als: (getal >= 0) **and** (getal <= 100)
FOUT als: not ((getal >= 0) **and** (getal <= 100))
dus met de Morgan: not (getal >= 0) **or** not (getal <= 100)
of ook: (getal < 0) **or** (getal > 100)

```
int getal = -1;
while((getal < 0) || (getal > 100))
{
  scanf("%d", &getal);
}
// nu geldt:   not ((getal < 0) || (getal > 100) ) dus
// ( getal >= 0 ) and ( getal <= 100 ) en dat wilde je ook
```

VbE:
Het programma telt het aantal ingelezen woorden tot aan de punt.

```
int aantal_woorden = 0; int nieuw_woord = 0; char ch;
while ( (ch = getchar()) != '.' )
{
  if ( ch == ' ' || ch == '\t' || ch == '\n' )
    { nieuw_woord = 0;}
  else if (!nieuw_woord)
    { aantal_woorden++; nieuw_woord = 1;}
}
printf("Aantal woorden is %d\n", aantal_woorden);
```

Uitvoer:
dit programma telt<ENTER>woorden<ENTER>tot de punt.<ENTER>
Aantal woorden is 7

Invoer was:
dit programma telt <ENTER> woorden <ENTER> tot de punt. <ENTER>

eXtra stof: **Meer over functies**

Nu komen twee onderwerpen aan bod: ***recursieve functies*** en ***pointers naar functies***. Als bekend is hoe pointers naar functies moeten worden gebruikt, is het ook mogelijk om ***functies als argumenten*** toe te passen

Recursie

Functies kunnen (binnen hun functie-body) andere functies aanroepen. Een ***recursieve functie*** is een functie die **zichzelf aanroept**. Recursieve functies zijn relatief eenvoudig te programmeren, maar hun werking is lastig te begrijpen. En recursieve functies zijn "duur" in geheugengebruik, omdat de **aanroep** van een recursieve functie altijd tot gevolg heeft dat er een ***stack*** (of stapel) wordt aangemaakt. En als er een stapel wordt aangemaakt, moet die ook altijd weer worden afgebroken.

Er worden nu vijf voorbeelden van een recursieve functie gegeven. Daarbij wordt voornamelijk aandacht besteed aan het **schrijven** van een recursieve functie (dat is relatief eenvoudig).

Voorbeeld 11_8
Bereken de faculteit van n (dwz n!) eerst ***iteratief*** en daarna ***recursief***.

```
// dit is iteratieve functie faculteit(. . .)
long int faculteit(const int n)
{
  int k;
  long int product = 1L;
  for ( k = n ; k > 0 ; k-- )        // iteratief: met een herhalings-lus
    product *= k;
  return product;
}
```

Uit de wiskunde is bekend:

```
n! = n * (n-1) * (n-2) * . . . * 3 * 2 * 1
0! = 1
```

bijvoorbeeld:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

In formulevorm:

```
            1              , als n == 0  // zgn. stopcriterium
fac(n) = {
            n * fac(n-1) , als n > 0    // zgn. recurrente betrekking
```

```
// dit is de recursieve functie fac(. . .)
long int fac(const int n)
{
  if ( n == 0 )
    return (1L);                // zgn. stopcriterium: geen aanroep van fac()
  else
    return n * fac(n-1);        // zgn. recurrente betrekking
}
```

Uitleg werking recursie:

Door de aanroep

```
  long int f = fac(5);
```

vinden de volgende akties plaats:

// parameter **n** heeft lokale scope; en n heeft de waarde 5 bij eerste aanroep:

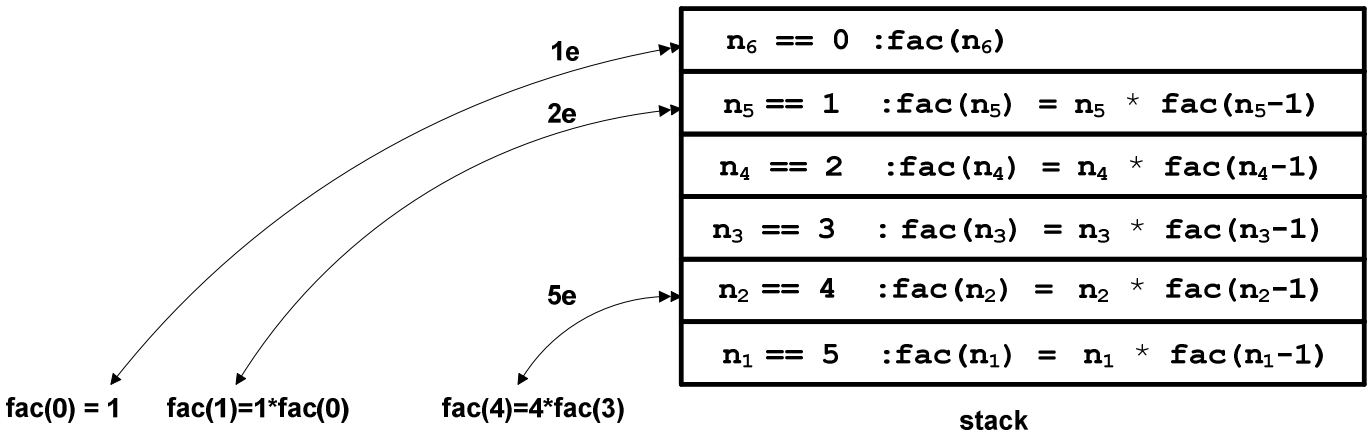
```
  if ( n == 0 )
    return (1L);
  else
    return n * fac(n-1);
```

Omdat `n != 0` is, wordt de `else` uitgevoerd. Omdat binnen de body van functie `fac` een aanroep voor komt van diezelfde functie `fac`, wordt een stack (stapel) aan gemaakt.

1. Onder op die stapel komt de berekening **fac(n_1) = n_1 * fac(n_1−1)** te liggen, met daarbij **n_1 == 5**
2. Vervolgens wordt **fac(4)** aangeroepen: op de stapel wordt gelegd de berekening **fac(n_2) = n_2 * fac(n_2−1)** te liggen, met daarbij **n_2 == 4**
3. Vervolgens wordt **fac(3)** aangeroepen: op de stapel wordt gelegd de berekening **fac(n_3) = n_3 * fac(n_3−1)** te liggen, met daarbij **n_3 == 3**
4. Vervolgens wordt **fac(2)** aangeroepen: op de stapel wordt gelegd de berekening **fac(n_4) = n_4 * fac(n_4−1)** te liggen, met daarbij **n_4 == 2**
5. Vervolgens wordt **fac(1)** aangeroepen: op de stapel wordt gelegd de berekening **fac(n_5) = n_5 * fac(n_5−1)** te liggen, met daarbij **n_5 == 1**
6. Vervolgens wordt **fac(0)** aangeroepen: op de stapel wordt gelegd de aanroep **fac(n_6)** te liggen, met daarbij **n_6 == 0** : nu stopt de recursie, omdat **fac(0) niet meer de functie fac aanroept**.

Men zegt : de recursie is ***6 diep*** (de stapel heeft hoogte 6)

Als de recursie is gestopt, wordt de stapel afgebroken (van boven naar beneden)



➔

Voorbeeld 11_9

Als Voorbeeld 11_8, maar dan voor de berekening van de som van de cijfers van een getal `n`

Voorbeeld: de som van de cijfers van getal 125 is: 1 + 2 + 5 = 8

In formulevorm:

```
                  0                          , als n == 0
cijfersom(n) = {
                  n%10 + cijfersom(n/10)     , als n > 0
```

```
// de recursieve functie cijfersom(. . .)
int cijfersom(const long int n)
{
  if ( n == 0 )
    return 0;                                // stopcriterium
  else
    return n%10 + cijfersom(n/10);           // recurrente betrekking
}

#include <stdio.h>

long int faculteit(const int n);
long int fac(const int n);
int cijfersom(const long int n);

main()
{
  int getal = 7;
  printf("Iteratieve faculteit : %ld\n", faculteit(getal));
  printf("Recursieve faculteit : %ld\n", fac(getal));

  long langGetal = 89675678453L;
  printf("cijfersom(%ld) = : %d\n",
                          langGetal, cijfersom(langGetal));
}
```

Uitvoer :

```
Iteratieve faculteit : 5040
Recursieve faculteit : 5040
cijfersom(896767843) = 58
```

➔

Voorbeeld 11_10

Als Voorbeeld 11_8, maar dan voor het afdrukken in binaire vorm (nullen & enen) van een getal `n`

Voorbeeld: getal 19 in binaire vorm is: 10011 ($19 = 16 + 2 + 1 = 2^4 + 2^1 + 2^0$)

In formulevorm:

`n` afdrukken `, als n == 0|| n == 1`

`binair(n) = {`

`binair(n/2)` doen,

gevolgd door : `n%2` afdrukken `, als n > 1`

```
// de recursieve functie binair(. . .)
void binair(const int n)
{
  if ( n == 0 || n == 1)            // stopcriterium
    printf("%d", n);
  else
  {
    binair(n/2);                    // recurrente betrekking
    printf("%d", n%2);
  }
}

main()
{
  getal = 138;
  printf("%d binair : ", getal);
  binair(getal);
  putchar('\n');
}
```

Uitvoer :

138 binair : 10001010

➔

Voorbeeld 11_11

Als Voorbeeld 11_8, maar nu met een recursieve void-functie **backwards()**. Er moeten via het toetsenbord een aantal characters worden ingetikt, afgesloten door een <ENTER>. De functie toont de ingetikte characters in omgekeerde volgorde ophet scherm.

```
<stdio.h>

// de recursieve functie backwards(. . .)
void backwards( )
{
  char ch;
  if ( ( ch = getchar() ) == '\n' )        // stopcriterium
    printf("  Nu stopt backwards  ");
  else
  {
    backwards();                           // recurrente betrekking
    putchar(ch);
  }
}

main()
{
  printf("Tik een regel tekst in : \n");
  backwards();
  putchar('\n');
}
```

Uitvoer :

```
Tik een regel tekst in :
bom<ENTER>
  Nu stopt backwards   mob
```

Voorbeeld 11_12
Via het toetsenbord moeten een aantal characters worden ingetikt; de invoer wordt afgesloten door het intikken van <ENTER>. Nu vindt er aktie plaats zowel **voor** de recursieve aanroep als ook **erna**. Voor de recursieve aanroep wordt ieder character éénmaal afgedrukt; en erna wordt het tweemaal afgedrukt.

```
#include <stdio.h>

// de recursieve functie recursieDemo(. . .)
void recursieDemo( )
{
  char ch;
  if ( ( ch = getchar() ) == '\n' )     // stopcriterium
    printf("  recursieDemo stopt  ");
  else
  {
    putchar(ch);
    recursieDemo();                     // recurrente betrekking
    putchar(ch);
    putchar(ch);
  }
}

main()
{
  printf("Tik een regel tekst in : \n");
  recursieDemo();
  putchar('\n');
}
```

Uitvoer:
Tik een regel tekst in :
bom<ENTER>
bom recursieDemo stopt   mmoobb

Pointers naar functies

In het vak operating systems (OPSY) zul je gebruik moeten maken van een pointer naar een functie. Hieronder wordt uitgelegd wat daarmee wordt bedoeld.

// dit is een functie-***declaratie***:
```
int kwadraat(const int);
```

// dit is een functie-***definitie***:
```
int kwadraat(const int n)
{
  return n * n;
}
```

// dit is een functie-***aanroep***:
```
int x = 5;
int y = kwadraat(x);                    // aanroep van functie kwadraat()
printf("x = %d en y = %d\n", x, y);
```

Er zijn situaties, waarin je pas **tijdens het runnen** wilt aanwijzen (= toekennen) welke functie je wilt gaan gebruiken. Dat is mogelijk in C door gebruik te maken van een zgn. ***pointer naar een functie***. Zo'n pointer declareer je als volgt:

```
// dit is een declaratie van een pointer naar een functie ,
// met fp is de naam van die pointer,
// en fp mag alleen wijzen naar een functie met een int als return-waarde,
// en met één parameter van het type const int :

int (*fp) (const int);
```

Dan kun je in het programma de volgende toekenning maken:

```
fp = kwadraat;          // bedoeld wordt: de functie kwadraat()
```

Door deze toekenning gebeurt het volgende:

- Aan de rechterkant van de = staat **kwadraat** . Dat is een expressie van het type : **functie** met één parameter (een parameter van type: const int) die als resultaat een int aflevert. Die expressie heeft altijd de vorm van een functienaam (in dit vb is dat de naam: **kwadraat**)
- Door de toekenning van **kwadraat** aan **fp** vindt ***automatische typeconversie*** van **kwadraat** plaats. Nl. van functie naar pointer naar functie.
- Nu is ***fp** in feite hetzelfde als de functie **kwadraat**. En heeft de aanroep

  ```
  int z1 = (*fp)(x);
  ```

  betekenis.
  Opmerking: omdat de functie-haakjes (en) de allerhoogste prioriteit hebben, kun je **niet** schrijven:

  ```
  int z1 = *fp (x);
  ```

  want dan zouden de ( ) het winnen van de * (dus: ***(**fp(x)**)**)

De toekenning:

```
int z1 = (*fp)(x);
```

is nogal onhandig. Daarom is het ook toegestaan om die toekenning als volgt te noteren:

```
int z1 = fp(x);
```

De declaratie:

```
int i, *pi, f(int), *fpi(int), (*fi)(int);
```

betekent:

```
int           i;     // int i
int          *pi;    // int-pointer pi
int        f(int);   // functie met naam f (en een int als return-waarde)
int  *fpi(int);      // functie met naam fpi (en een int-pointer als return)
int (*fi)(int);      // pointer naar een functie fi
```

➔

Voorbeeld:

```
// Wanneer nu de volgende vier declaraties zijn gemaakt:
int kwadraat(const int);
int helft(const int);
int derde_macht(const int);
int (*fp) (const int);


// Dan zijn de volgende toekenningen toegestaan:
int x  = 5;
fp     = kwadraat;
int y1 = kwadraat(x);
int z1 = (*fp)(x);     // ( ) zijn nodig om *fp heen i.v.m. prioriteiten
printf("x = %d en y1 = %d en z1 = %d\n", x, y1, z1);

fp     = helft;
int y2 = helft(x);
int z2 = (*fp)(x);
printf("x = %d en y2 = %d en z2 = %d\n", x, y2, z2);

fp     = derde_macht;
int y3 = derde_macht(x);
int z3 = fp(x);       // hier is de korte notatie gebruikt
printf("x = %d en y3 = %d en z3 = %d\n", x, y3, z3);
```

Uitvoer:

x = 5 en y1 = 25 en z1 = 25
x = 5 en y2 = 2 en z2 = 2
x = 5 en y2 = 125 en z3 = 125

Opmerking:

Lees in het boek de tekst over "pointers naar functies" van 11.8 (blz 84 t/m 86)

➔

Vb1

// declaratie van functie met naam `mijn_functie`
// en introductie van type met naam `PointerNaarFie`

```
int   mijn_functie(const int);
typedef   int (*PointerNaarFie) (const int);
```

// declaratie van een variabele met naam `func` en type `PointerNaarFie`

```
PointerNaarFie func;
```

// toekening aan `func`

```
func = mijn_functie;
```

Vb2

// nu kun je ook een *array* met naam `arpnf` declareren:

```
PointerNaarFie arpnf[4];
//int ar[10];
```

vergelijk: dit is de declaratie van *array* `ar` van 10 ints

// zonder typedef had de declaratie van diezelfde *array* er als volgt uit gezien:

```
int (*arpnf[4]) (const int);
```

Vb3

```
int kwadraat(const int), helft(const int),
                                    derde_macht(const int);
typedef     int (*PointerNaarFie) (const int);
```

// dan is toegestaan:

```
PointerNaarFie arpnf[ ] = { kwadraat, helft, derde_macht};
int m;
scanf("%d", &m);
for ( int k = 0 ; k < 3 ; k++ )
  printf("%d\n", arpnf[k] (m));
```

➔
Functies als argumenten

Nu het mogelijk blijkt te zijn om gebruik te maken van een pointer naar een functie, kunnen we ook de logische vervolgstap meken. We gebruiken nu een ***functie als argument*** dwz als ***parameter*** in een andere functie. Hieronder wordt uitgelegd hoe dat in zijn werk gaat.

Voorbeeld 11_13

```
#include <stdio.h>

double kwadraat(const double);
double derde_macht(const double);
void tabelleer( double (*) (const double),
                  const double, const double, const double);
main()
{
  tabelleer(kwadraat, 0.0, 2.0, 0.1);
  tabelleer(derde_macht, 0.0, 2.0, 0.1);
}

double kwadraat(const double x)  {
  return x*x;
}

double derde_macht(const double x) {
  return x*x*x;
}

void tabelleer( double (*f) (const double), const double van,
                          const double tot, const double stap)
{
  double z;
  for (z = van ; z <= tot + (0.5 * stap); z += stap )
    printf("%6.2f %10.4f\n", z, f(z) );
}
```

Practicum Week 5 (Default Practicum)

Opdracht 5_1 zie voorbeelden met recursie

De volgende vergelijkingen definiëren de getalrij van Fibonacci:

```
Fib(1) = 1
Fib(2) = 1
Fib(n) = Fib(n-1) + Fib(n-2) als n > 2
```

Schrijf een ***recursieve*** C-functie `fibo()` die rechtstreeks op deze declaratie gebaseerd is. Schrijf een bijbehorende `main()` en test je functie `fibo()`. Bouw een (globaal) tellertje in om te tellen hoe vaak `fibo()` wordt aangeroepen.

Opdracht 5_2

Schrijf een ***recursieve*** functie `aantal(n, k)` die het **aantal keren** aflevert/telt dat het **cijfer k** in het positieve gehele getal n voorkomt. Schrijf ook de functie `main()` en test functie `aantal()`

Voorbeelden:

```
aantal(4214, 4) = 2     // cijfer 4 komt 2 keer voor in getal 4214
aantal(73, 1) = 0       // cijfer 1 komt 0 keer voor in getal 73
```

Opdracht 5_3 sorteren van 3 getallen

Invoer: Via het toetsenbord worden drie getallen `a`, `b` en `c` ingelezen.

Uitvoer: Die zelfde drie getallen, gesorteerd van klein naar groot, worden afgedrukt op het beeldscherm. Het sorteren van de drie getallen moet worden uitgevoerd door een **functie sorteer()**. (gebruik `verwissel()`)

Opdracht 5_4

Schrijf een functie met heading

```
void bereken-flappen(float bedrag, int * pflap50,
                     int * pflap20, int * pflap10,
                     int * pflap5, int * prest)
```

De functie heeft één ***invoer-parameter***, voor het geldbedrag. En de functie heeft 5 ***uitvoer-parameters*** nl. om dat geldbedrag te splitsen in flappen van 50 €, 20 €, 10 €, 5 € en het restant aan Eurocenten. Schrijf ook een functie `main()` en test of het berekenen van de flappen correct gebeurt.

Voorbeeld: een geldbedrag van € 123.45 bestaat uit 2 flappen van 50 Euro, 1 flap van 20 Euro, 0 flappen van 10 Euro, 0 flappen van 5 Euro, en de rest bedraagt 345 Eurocenten.

Opdracht 5_5

In alle vijf de onderdelen moet worden gebruik gemaakt van een array `ar` met declaratie:

```
char ar[100];
```

In de array kunnen dus maximaal 99 waarden worden opgeslagen: `ar[0]` t/m `ar[98]`. Achter het laatste karakter staat het nul-karakter.

a) Lees via toetsenbord een aantal waarden in met waardes uitsluitend 0 of 1. En schrijf een functie met aanroep

```
drukAf(ar, aantal);
```

Deze functie drukt de inhoud van de array af op het beeldscherm.

b) Schrijf een functie met aanroep:

```
char * ptr = adresVan8Nullen(ar, aantal);
```

De functie doorloopt de array, en zoekt naar de **eerste 8 opeenvolgende nullen** De functie **levert als resultaat af het adres** van de geheugenplaats van de eerste 0 van die 8 nullen. Wanneer er geen 8 opeenvolgende nullen zijn, levert de functie als resultaat af de waarde NULL

c) Schrijf een functie met aanroep:

```
char * ptr = evenwicht(ar, aantal);
```

De functie levert als resultaat af het adres van de geheugenplaats met "links" **eveneveel enen** als "rechts"

d) Schrijf een functie met aanroep:

```
char * ptr = adresVan(ar, aantal);
```

Dit is het adres van de geheugenplaats met de tiende 0; wanneer er minder dan 10 nullen in de array staan, wordt de waarde NULL afgeleverd.

e) Schrijf een functie met aanroep:

```
inverteer(ar, aantal);
```

De functie verwisselt alle array-getallen van plaats : de eerste met de laatste, de tweede met de eenna laatste, etc. Let op: het gaat er niet om dat je de inhoud achterste voren afdrukt. Het is de bedoeling dat je de array vult met de waarden in omgekeerde volgorde. Daarmee gaat de oorspronkelijke inhoud dus verloren.