Week 2 : Hoofdstukken 2 en 6; eXtra stof: inleiding pointers

Hoofdstuk 6: Beslissingen: ***if***-statement en ***switch***-statement.

Inleiding:

Condities zijn waar (***true***) of onwaar (***false***) In C staat het **int-getal** `0` gelijk aan `false`, en **alle andere int-waarden** worden geïnterpreteerd als `true`. Verder zijn er in C zgn. ***prioriteitsregels*** (zie dictaatje Les 1, blz 18), die – voor de programmeur – onverwachte resultaten kunnen opleveren.

Voorbeelden:

```
double x = 1.0;
double y = exp(1);
  x > y          // is toegestaan; resultaat is 0
  x >= y         // ook toegestaan; is ook 0
  x > = y        // niet toegestaan: die spatie mag niet
  x => y         // omdraaien van = en > mag niet

char ch2 = 'X';                  // opm.: de bijbehorende int-waarde is 88
  ch2 >= 'A'                     // toegestaan; is 1
  ch2 == '2' || ch2 == '1'       // toegestaan; is 0
  ch2 == 2 || ch2 == 1           // toegestaan; is 0    (maar ch2 == 88 is 1)

int index = 9;
  2 < index < 6          // toegestaan : het resultaat heeft waarde true (1)
  if ( index == 2 ) . . .     // toegestaan; is 0
  if ( index != 2 ) . . .     // toegestaan; is 1
  if ( index ! = 2 ) . . .    // niet toegestaan: spatie mag niet
  if ( index = 2 ) . . .      // toegestaan; is 2, dus true (en index = 2;)
```

Opmerking: het is mogelijk om de begrippen `TRUE` en `FALSE` te introduceren in C. Hieronder staat aangegeven hoe je dat kunt doen. Ik doe het niet, omdat dit niet overeenstemt met de manier waarop C hier mee om gaat.

```
  #define   FALSE   0
  #define   TRUE    1        // ! uitsluitend TRUE = 1 beperkt te veel !
```

Inlezen van toetsenbord met de functie `getchar()`:

De functie `int getchar()` leest één `char` van toetsenbord, en levert die gelezen `char` af als resultaat. Wanneer het lezen mislukt, wordt de waarde `EOF` afgeleverd.

```
char ch;
printf("Tik een char in: ");
ch = getchar();
printf("Ingelezen karakter is: %c\n", ch);
```

Uitvoer (op beeldscherm):

Tik een char in: w
Ingelezen karakter is: w

Opmerkingen:

- Met de declaratie

  ```
  int ch;
  ```

  wordt dezelfde uitvoer geproduceerd.

- Gewoonlijk worden `getchar()` en `putchar()` als ***macro*** geïmplementeerd. Wanneer de ***preprocessor*** van C wordt behandeld, zal ook worden uitgelegd wat bedoeld wordt met ***macro***.

- Invoer van ***characters*** (type `char`) en ook van ***strings*** (zoals bijvoorbeeld type `char s[80]`) gaat **niet** op dezelfde manier als invoer van ***getallen***.

  Bij inlezen van toetsenbord moet altijd de vraag worden gesteld:

  ## ? om welk type invoer gaat het hier ?

  Is het antwoord:

  1. een ***getal*** (types `int, short, long, char, float, double`) kies dan in eerste instantie voor inlezen met `scanf()` (zie Les 3)
  2. een ***karakterstring*** (zoals bijvoorbeeld type `char s[80]`) pas dan heel erg goed op voor de risico's van `scanf()` en kies liever voor `gets()` of `fgets()`. Karakterstrings staan in Hfst 8.
  3. een enkel ***character*** of een aantal "losse" ***characters***, kies dan in eerste instantie voor `getchar()`

Voorbeeld 6_1

Hieronder staat een voorbeeld waarin wordt gelezen van toetsenbord met `getchar()` en wordt afgedrukt op beeldscherm met `putchar()`.

```
#include <stdio.h>

int main(void)
{
  int ch;
  printf("Tik een enkel karakter in : ");
  ch = getchar();            // De return-waarde is van type int
  printf("Er is ingetikt: ");
  putchar(ch);               // Ook putchar() heeft als return-waarde: int
  putchar('\n');             // Ga over op een nieuwe regel.

  printf("En nu moet de ENTER nog uit de buffer
                                        verwijderd: \n");
  int weggooien;
  weggooien = getchar();
  if ( (char)weggooien == '\n' )
    printf("waarde van weggooien is ENTER\n");
  else if ( weggooien == EOF )
    printf("waarde weggooien is EOF\n");
  else
    printf("weggooien ongelijk EOF of ENTER\n");

  int maand;
  char ch1, ch2;
  printf("Tik een maandnummer in: ");
  ch1 = getchar();
  ch2 = getchar();
  getchar();          // om de ENTER uit de buffer te verwijderen

  maand = (ch1 - '0')*10 + (ch2 - '0')*1;
  printf("ch1 = %c ch2 = %c maand = %d\n", ch1, ch2, maand);
  return 0;
}
```

Uitvoer:

Tik een enkel karakter in : w (maar daarachter staat de ENTER-char '\n')
Er is ingetikt: w
En nu moet de ENTER nog uit de buffer verwijderd:
waarde van weggooien is ENTER
Tik een maandnummer in: 12 <ENTER>
ch1 = 1 ch2 = 2 maand = 12

In C levert vrijwel iedere functie ook nog een resultaat af. In Voorbeeld 6_1 is dat afgeleverde resultaat niet zichtbaar. Hieronder staat een versie, waarin het afgeleverde resultaat wel zichtbaar is gemaakt. Er wordt nu ook getest of `ch1` en `ch2` getallen zijn.

Voorbeeld 6_2
Onderstaand voorbeeld doet vrijwel hetzelfde als Vb 6_1 maar is beter beveiligd tegen onjuiste invoer. Maar 56 is geen corrrect maandnummer.

```
#include <stdio.h>

int main(void)
{
  char ch;
  printf("Tik een enkel karakter in : ");
  if ((ch = getchar()) != EOF)
    printf("Ingetikt is: %c\n", ch);
  getchar();                    // verwijder de <ENTER> uit de buffer

  int maand;
  char ch1, ch2;
  printf("Tik een maandnummer in: ");
  ch1 = getchar();
  ch2 = getchar();
  getchar();          // om de ENTER uit de buffer te verwijderen
  if (ch1 >= '0' && ch1 <= '9' && ch2 >= '0' && ch2 <= '9') {
    maand = (ch1 - '0')*10 + (ch2 - '0')*1;
  }
  else {
    maand = -1;
  }
  printf("ch1 = %c ch2 = %c maand = %d\n", ch1, ch2, maand);
  return 0;
}
```

Uitvoer:
Tik een enkel karakter in : w <ENTER>
Ingetikt is: w
Tik een maandnummer in: 56 <ENTER>
ch1 = 5 ch2 = 6 maand = 56

Opmerking:
In C levert vrijwel iedere functie een waarde af. Dat kan aanleiding geven tot onbegrijpelijke compiler-foutmeldingen (maar dan word je gered door de compiler, dus dat komt wel weer goed) Maar vaak vindt de compiler alles OK, terwijl het resultaat fout is. In Vb 6_2 staat:

```
char ch;
printf("Tik een enkel karakter in : ");
if ((ch = getchar()) != EOF)
  printf("Ingetikt is: %c\n", ch);
getchar();
```

Dit gaat **goed**. Wie echter kiest voor:

```
char ch;
printf("Tik een enkel karakter in : ");
if (ch = getchar() != EOF)
  printf("Ingetikt is: %c\n", ch);
getchar();
```

krijgt een **fout** antwoord. Dat komt doordat C de expressie

```
ch = getchar() != EOF
```

leest als

```
ch = (getchar() != EOF)
```

De operator `!=` "wint" van de operator `=` (hij heeft een hogere prioriteit; zie Les 1, blz 18). De functie `getchar()` leest een `char` in ("doet" wat) maar levert ook nog een resultaat af (de ingelezen `char`). Er wordt getest of die afgeleverde waarde ongelijk is aan `EOF`. Dat is in dit voorbeeld waar, en het resultaat van `(getchar() != EOF)` is dus het int-getal 1. En vervolgens wordt aan `ch` het `int`-getal 1 toegekend. De C-compiler ziet hier geen enkele fout in. Vermoedelijk had de programmeur iets anders in gedachte.
Het ASCII-teken met `int`-waarde 1 is de Ctrl-A (de ASCII-codes 0 t/m 31 en 127 zijn niet-afdrukbaar)

De expressie:

```
(ch1 >= '0' && ch1 <= '9' && ch2 >= '0' && ch2 <= '9')
```

gaat goed omdat de operatoren `>=` en `<=` "winnen" van de ***logische and***. Je mag natuurlijk altijd (voor de zekerheid) haakjes zetten. Dat is wel zo veilig in C.

Voorbeeld 6_3
Wanneer in een ***if-statement*** meer dan één opdracht moet worden uitgevoerd, kan dat alleen maar door gebruik te maken van een zgn. ***compound statement***.

```
#include <stdio.h>

int main(void)
{
  int a, b, c ;
  a = 42;
  b = 13;
  if ( (c = 7 - b) == a )
    printf("a = %d\n", a);
    printf("b = %d\n", b);
    printf("c = %d\n", c);
  return 0 ;
}
```

Uitvoer:

```
b = 13
c = -6
```

De haakjes in de expressie

```
(c = 7 - b) == a
```

zorgen ervoor dat eerst aan variabele `c` de waarde -6 wordt toegekend. Vervolgens wordt -6 vergeleken met de waarde van `a`. Omdat -6 != 42 is, zou je mischien denken dat er niets wordt afgedrukt. Dat was het geval geweest, wanneer je er een ***compound statement*** van had gemaakt:

```
if ( (c = 7 - b) == a )
{
  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d\n", c);
}
```

Vraag: waarom krijgt `c` de waarde 0 in onderstaande situatie?

```
if (!( c = 7 - b == a )){
  printf("a = %d\n", a);
  printf("b = %d\n", b);
  printf("c = %d\n", c);
}
```

Voorbeeld 6_4 (switch-statement)

Het ***switch-statement*** staat in het boek beschreven op blz 40 en 41. In onderstaand voorbeeld wordt eerst een karakter ingelezen van toetsenbord. Wanneer dit karakter een '1' is, wordt de tekst "dit is keuze 1" afgedrukt. Is er een '2' ingetikt, dan verschijnt de tekst "dit is keuze 2". En voor alle andere invoer verschijnt de tekst "deze keuze bestaat niet".

```
#include <stdio.h>

int main(void)
{
  char ch;
  printf("Tik uw keuze in: ");
  ch = getchar();
  while (getchar() != '\n') { }      // maak de buffer weer leeg

  switch(ch)
  {
    case '1': printf("dit is keuze 1\n"); break;
    case '2': printf("dit is keuze 2\n"); break;
    default : printf("deze keuze bestaat niet\n"); break;
  }
  return 0;
}
```

Uitvoer:

Tik uw keuze in: 1 <ENTER>
dit is keuze 1

Opmerking:
Wanneer in het ***switch-statement*** alle ***break***'s zouden zijn weg gelaten had het programma opgeleverd:

Uitvoer:
Tik uw keuze in: 1 <ENTER>
dit is keuze 1
dit is keuze 2
deze keuze bestaat niet

Dat komt doordat de ***switch*** in feite een `goto`-opdracht is.

Je mag de ***switch*** uitsluitend loslaten op type `int`. Maar voor C is een `char` ook een `int`
Bij de uitvoering van de ***switch*** wordt eerst de ***expressie*** tussen de ronde haakjes uitgerekend/bepaald (in het vb is dat `ch`). Vervolgens worden alle ***case-labels*** afgelopen totdat een waarde wordt gevonden, die gelijk is aan het berekende / bepaalde resultaat. Wanneer dat het geval is, wordt gesprongen naar de rechterkant van dat case-label (dwz naar alles wat achter de : punt staat). En vervolgens worden **alle opdrachten aan de rechterkant uitgevoerd**. Door een `break` op te nemen wordt uit de ***switch*** gesprongen.

Wanneer er geen gelijke waarde wordt gevonden onder de case-labels, wordt er gesprongen naar de `default` (die dus helemaal niet onderaan hoeft te staan). Wanneer er geen gelijke waarde wordt gevonden onder de case-labels, en er is ook geen default, doet de ***switch*** niets. Dit leidt dus **niet** tot een foutmelding.

Opmerking:
Met-break-erbij maakt de volgorde van de case-labels niet uit. Zonder-break-erbij wel.
Wanneer karakter `ch` de waarde `'1'` heeft levert de code

```
switch(ch)
{
  case '1': printf("dit is keuze 1\n");
  case '2': printf("dit is keuze 2\n");
  default : printf("deze keuze bestaat niet\n");
}
```

als uitvoer op:

```
dit is keuze 1
dit is keuze 2
deze keuze bestaat niet
```

terwijl de code

```
switch(ch)
{
  default : printf("deze keuze bestaat niet\n");
  case '1': printf("dit is keuze 1\n");
  case '2': printf("dit is keuze 2\n");

}
```

dan oplevert als uitvoer:

```
dit is keuze 1
dit is keuze 2
```

Voorbeeld 6_5

De taal C kent de ***conditionele operator***, net als Java. Voor bijna alle operatoren geldt, dat de ***associativiteit*** van links naar rechts gaat. Bij de ***conditionele operator*** gaat dit **van rechts naar links**
(opm. bij de ***toekennings-operator*** gaat het overigens ook van rechts naar links)

```
#include <stdio.h>

int main(void)
{
  int a = 3;
  int b = -17;
  int c;
  c =  (a < b) ? 18 : 12;
  printf("a = %d en b = %d en c = %d\n", a, b, c);

  int min = (a < b) ? a : b;          // eerst de rechterkant; die wordt b
  printf("minimum van %d en %d is %d\n", a, b, min);

  // dit is niet toegestaan in C:
  ( a < b ? a : b ) = 0;              // b is de kleinste, dus b wordt 0 ?

  return 0;
}
```

Uitvoer:

a = 3 en b = -17 en c = 12
minimum van 3 en -17 is -17
// mag niet! In C mag de conditionele operator niet links van de =

Voorbeeld 6_6

Dit programma drukt een gelijkbenige driehoek van sterretjes af. De eerste regel bestaat uit 1 sterretje, de tweede uit 3 sterretjes, gecentreerd onder de eerste. Etc. Er zijn 5 regels in totaal. Ik gebruik for-lussen. Die komen in Hfst 7 uitgebreider aan bod.

```
#include <stdio.h>

int main(void)
{
  int regel, spaties, ster;
  for ( regel = 1 ; regel <= 5 ; regel++ )
  {
    for ( spaties = 1 ; spaties <= 5 - regel ; spaties++ )
      putchar(' ');

    for ( ster = 1 ; ster <= 2*regel-1 ; ster++ )
      putchar('*');
    putchar('\n');
  }

  printf("Dit zijn %d regels met * \' s\n", 5);
  return 0;
}
```

Uitvoer:

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

Dit zijn 5 regels met * ' s

Opmerking: bij runnen in C vormen de sterretjes een fraaier plaatje dan in Word.

eXtra aanvulling op het boek: Inleiding ***pointers*** in C

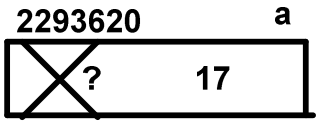
Voorbeeld Extra_1
Dit is een voorbeeld met een ***pointer*** van type **`int*`** (spreek uit: ***int-pointer***).
Opmerking: de antwoorden met een ➔ zijn ***implementatie-afhankelijk***

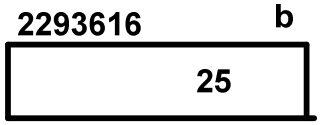
```
#include <stdio.h>

int main(void)
{
  int a;
  printf("begin-adres van a (hexa): %x\n", &a);     // 22ff74 ➔
  printf("begin-adres van a (deci): %d\n", &a);     // 2293620 ➔
  printf("waarde van a: %d\n", a);                  // 2 ➔ : ?

  a = 17;
  printf("begin-adres van a: %d\n", &a);            // 2293620 ➔
  printf("waarde van a: %d\n", a);                  // 17

  int b = 25;
  printf("begin-adres van b: %d\n", &b);            // 2293616 ➔
  printf("waarde van b: %d\n", b);                  // 25

  int *pi;                              // pointer pi heeft nog geen waarde
  pi = &a;                              // waarde van pi is adres 2293620 ➔
  printf("waarde van pi: %d\n", pi);                // 2293620 ➔

  pi = &b;
  printf("waarde van pi: %d\n", pi);                // 2293616 ➔
  printf("inhoud van pi: %d\n", *pi);               // 25
  return 0;
}
```

Uitvoer:
begin-adres van a (hexa): 22ff74
begin-adres van a (deci): 2293620
waarde van a: 2 (hier kan van alles en nog wat staan: ?)
begin-adres van a : 2293620
waarde van a: 17
begin-adres van b : 2293616
waarde van a: 25
waarde van pi: 2293620
waarde van pi : 2293616
inhoud van pi: 25

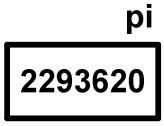
Opmerking:
Wanneer je een ***variabele*** declareert, moet je zowel de ***naam*** van de variabele als ook het ***type*** van de variabele opgeven. Er wordt dan in het geheugen ruimte gereserveerd om de variabele in op te slaan. Elke geheugenruimte heeft een ***naam*** (nl. de naam van de variabele in de declaratie) en een ***begin-adres***; en elke geheugenruimte heeft een gereserveerd ***aantal bytes*** (afhankelijk van het gedeclareerde ***type***).

In het voorbeeld heeft de variabele de ***naam*** `a`, en het ***type*** is `int`. Voor het opslaan van een `int` worden ***4 bytes*** gebruikt. Het ***begin-adres*** van de ***geheugenruimte*** is in het voorbeeld 2293620 (dat is implementatie-afhankelijk; dat getal wordt uitsluitend gebruikt voor demonstratie van de werking). Adressen worden doorgaans genoteerd als hexadecimaal getal. Een adres is een positief getal. De geheugenruimte voor variabele `b` is in het voorbeeld 4 (***bytes***) kleiner dan het adres van `a` (dat is implementatie-afhankelijk). Je mag in C de waarde van een begin-adres van een variabele opvragen (`&a` resp. `&b`).
Een ***int-pointer*** is een variabele die als waarde mag hebben: het (begin-)adres van een `int`. In het vb heeft ***int-pointer*** `pi` eerst als waarde `&a` (en later `&b`). Via `pi` kun je dan de ***waarde*** van `a` opvragen nl. `*pi` (en ook veranderen)





naam van de variabele is a
waarde van variabele a is 17 (= a)
begin-adres van variabele a is 2293620 (= &a)



naam van de variabele is pi
waarde van variabele pi is 2293620 (= pi of ook &a)
via pi kun je de inhoud van a opvragen dmv *pi (en *pi = 17, want a = 17)
Opm.variabele pi heeft zelf ook weer een adres (= &pi)

Voorbeeld Extra_2 Waarschuwing: doe dit niet!!

Dit is een voorbeeld met een ***pointer*** van type **`int*`** waarbij wordt gedaan aan ***pointer-aritmetiek***. Dat is typisch C, maar kan tot draconisch slechte programma's leiden.
Opmerking: de antwoorden met een ➔ zijn ***implementatie-afhankelijk***

```
#include <stdio.h>

int main(void)
{
  int a, b, c, d;
  printf("&a = %d &b = %d &c = %d &d = %d\n", &a, &b, &c, &d);
  a = 125;
  b = -45;
  c = 1;
  d = 98;

  int *pi;
  pi = &a;
  printf("waarde van pi: %d\n", pi);
  printf("inhoud van pi: %d\n", *pi);

  pi--;
  printf("waarde van pi: %d\n", pi);
  printf("inhoud van pi: %d\n", *pi);

  pi -= 2;
  printf("waarde van pi: %d\n", pi);
  printf("inhoud van pi: %d\n", *pi);

  pi -= 1;
  printf("waarde van pi: %d\n", pi);
  printf("inhoud van pi: %d\n", *pi);

  return 0;
}
```

De uitvoer staat op de volgende bladzijde.

Doel van dit programma:
Wanneer je van `int`-pointer `pi` 1 aftrekt, wordt 4 (***bytes***) afgetrokken van de huidige adres-waarde. Omdat voor de opslag van één `int`-getal 4 bytes worden gebruikt. Bij een `char`-pointer wordt bij `pc++;` 1 (***byte***) erbij opgeteld.

!!! Voor pointer-variabelen is `p++;` dus **afhankelijk** van de soort pointer!!

Uitvoer:
&a = 2293620 &b = 2293616 &c = 2293612 &d = 2293608 ➔
waarde van pi: 2293620 ➔
inhoud van pi: 125
waarde van pi: 2293616 ➔
inhoud van pi: - 45 ➔
waarde van pi: 2293608 ➔
inhoud van pi: 98 ➔
waarde van pi: 2293604 ➔
inhoud van pi: 2293604 ➔

Voorbeeld Extra_3

Dit is een voorbeeld met een ***pointer*** van type **`int*`** En een ***float-pointer*** `pf` en een ***char-pointer*** `pc`. Het resultaat is weer implementatie-afhankelijk.

```
#include <stdio.h>

int main(void)
{
  int* pi;
  int a;
  pi = &a;
  *pi = 90678;
  printf("Aantal bytes van een int: %d\n", sizeof(int));   // 4
  printf("waarde van pi: %d\n", pi);                       // 2293616 ➔
  printf("begin-adres van a: %d\n", &a);                   // 2293616 ➔
  printf("inhoud van pi: %d\n", *pi);                      // 90678

  float* pf;
  float f;
  pf = &f;
  *pf = 45.0123f;
  printf("Aantal bytes van een float: %d\n",sizeof(float));// 4
  printf("waarde van pf: %d\n", pf);                       // 2293608 ➔
  printf("begin-adres van f: %d\n", &f);                   // 2293608 ➔
  printf("inhoud van pf: %.3f\n", *pf);                    // 45.012

  char* pc;
  char c;
  pc = &c;
  *pc = '$';
  printf("Aantal bytes van een char: %d\n", sizeof(char)); // 1
  printf("waarde van pc: %d\n", pc);                       // 2293603 ➔
  printf("begin-adres van c: %d\n", &c);                   // 2293603 ➔
  printf("inhoud van pc: %c\n", *pc);                      // $

  return 0;
}
```

Opmerking: een pointer is een (niet-negatief) `int`-getal

Hoofdstuk 2: Talstelsels

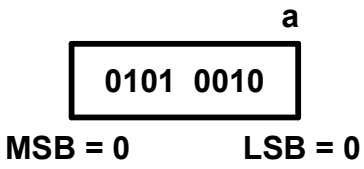
We kunnen declareren; en initialiseren:

```
unsigned char a;
a = 82;
```

Intern staat deze `a` dan opgeslagen in de 8 bits:

$$0101\ 0010 = \mathbf{0 * 2^7} + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + \mathbf{0 * 2^0}$$

De meest linkerbit heet ***Most Significant Bit***; de meest rechterbit heet ***Least Significant Bit***



We kunnen de waarde van `a` ***decimaal*** laten afdrukken, ***octaal*** of ***hexadecimaal*** (en vanaf Hfst 12 ook ***binair***). Intern verandert er niets: het blijven dezelfde acht nullen & enen.

In C-code:

```
unsigned char a;
a = 82;
printf("a (decimaal) = %d\n", a);
printf("a (octaal) = %o\n", a);
printf("a (hexadecimaal) = %0X\n", a);
```

Decimaal: (82)

$$0101\ 0010 = 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 82$$

Octaal (**0**122):

$$01\ \ 010\ \ 010 = \begin{array}{l} 0 * 2^7 + 1 * 2^6 \\ + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 \\ + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 \end{array}$$

$$= (0 * 2 + 1) * 2^6 + (0 * 2^2 + 1 * 2^1 + 0) * 2^3 + (0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

$$= (0 * 2 + 1) * 8^2 + (0 * 2^2 + 1 * 2^1 + 0) * 8^1 + (0 * 2^2 + 1 * 2^1 + 0 * 2^0)$$

$$= 1 * 8^2 + 2 * 8^1 + 2 = \mathbf{0}122$$

Hexadecimaal (**0X**52):

$$0101\quad 0010 = 0 * 2^7 + 1 * 2^6 + 0 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$$

$$= (0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1) * 2^4 + (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0)$$

$$= (0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1) * 16^1 + (0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0)$$

$$= 5 * 16^1 + 2 = \mathbf{0X}52$$

In het interne geheugen van de computer staat variabele `a` opgeslagen als een aantal nullen & enen. Heeft `a` type `char`, dan zijn het er 8. Bij type `int` zijn het er 32. Uitsluitend dat aantal nullen & enen bepaalt de mogelijke waarden van variabele `a`. Omdat `a` intern staat opgeslagen in de vorm van nullen & enen, is de ***binaire representatie*** bepalend voor het gedrag van `a`. In het verdere verhaal over talstelsels draait het dus in feite om die ***binaire representatie***.

Wij mensen zijn gewend om te rekenen in het ***decimale stelsel***. Rekenen in het binaire stelsel gaat op precies dezelfde manier. Op de volgende bladzijden zal dat worden gedemonstreerd.

Binair rekenen

Binair rekenen gaat op dezelfde manier als decimaal rekenen. Om dat duidelijk te maken zal eerst worden uitgelegd hoe je overzet **van decimaal naar binair** (de andere kant op, dwz van binair naar decimaal, is veel eenvoudiger; zie het vb op blz 17). Vervolgens zal binair worden opgeteld en afgetrokken. En tenslotte zal binair worden vermenigvuldigd en gedeeld. In de theorie wordt uitsluitend vermenigvuldigd en gedeeld met machten van 2. In practicumopdracht 2_3 moet worden vermenigvuldigd met 100.

Conversie van een decimaal getal naar zijn binaire representatie

In de eerste plaats moet je weten **in hoeveel bits** het gehele getal moet worden opgeslagen. In de voorbeelden zijn dat meestal 8 bits (dat past netjes op één regel). Vervolgens moet je weten of er ook negatieve getallen zijn toegestaan (keuze tussen `signed` en `unsigned`). Voorlopig kiezen we voor `unsigned`.

Bij 8 bits staat de MSB bij 2^7 ; de LSB staat in alle gevallen bij 2^0
Je maakt nu het tabelletje met alle machten van 2:

$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$

Nu reken je het grootst mogelijke getal uit. Dat is het getal

$$1111\ 1111 = 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 1 * 2^0$$

$$= 1 * (2^8 - 1) / (2 - 1) = 2^8 - 1 = 256 - 1 = 255$$

Het kleinst mogelijke getal is altijd het getal 0.

Decimaal getal 543 is groter dan 255, en past dus niet in 8 bits. Decimaal getal 243 past wel in 8 bits. Nu zoek je de grootste macht van 2 die past in getal 243.

En dan splits je getal 243 op in machten van 2:

$$
\begin{aligned}
243 &= 128 + 115 \\
&= 128 + 64 + 51 \\
&= 128 + 64 + 32 + 19 \\
&= 128 + 64 + 32 + 16 + 3 \\
&= 128 + 64 + 32 + 16 + 2 + 1 \\
&= 128 + 64 + 32 + 16 + 0 + 0 + 2 + 1 \\
&= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\
&= 1111\ \ 0011
\end{aligned}
$$

Opmerking
Je kunt dit ook gewoon opvragen aan de rekenmachine op je computer. Helaas is dit niet toegestaan op het tentamen.

Binair optellen en aftrekken

We tellen op de binaire representatie van: 243 + 5

$$
\begin{aligned}
243 &= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 \\
5 &= 0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\
&\text{----------} \quad + \\
&= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + \mathbf{2} * 2^0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow
\end{aligned}
$$

We zien dat dit goed gaat voor 0 + 0, 1 + 0 en 0 + 1. Maar voor 1 + 1 is het resultaat 2 (zie ➔) en dat mag niet. We moeten dus verder rekenen.

$$
\begin{aligned}
&= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^1 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow \qquad \rightarrow \\
&= 1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + \mathbf{2} * 2^1 + \mathbf{0} * 2^0 \\
&= 1111\ \ 1000 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \rightarrow
\end{aligned}
$$

Of ook,
met 0 + 0 = 0, 1 + 0 = 1, 0 + 1 = 1 en 1 + 1 = 2, maw = 0 en 1 “onthouden”:

```
1111  0011
0000  0101
---------- +
           2, dus 0 (1 onthouden)
          20, dus 00 (en opnieuw 1 onthouden)
         200, dus 000 (en opnieuw 1 onthouden)
        1000  (niets meer over om te onthouden)
1111  1000   (de resterende 4 zijn allemaal 1 + 0 en dat is 1)
```

We trekken nu af de binaire representatie van: 243 - 3
Nu moeten we niet “onthouden”, maar “lenen van 2” (in het decimale stelsel moesten we lenen van 10)

```
1111  0011
0000  0101
---------- -
          10   (1 – 1 = 0, daarna 1 – 0 = 1, en daarna moet 0 – 1; kan niet)
         110   (1 geleend van linker buur, en 2 – 1 = 1)
        1110   (1 uitgeleend aan rechterbuur; had hij niet; moest die lenen)
    0   1110   (1 uitgeleend aan rechterbuur; had zelf dus 0 over; en 0–0= 0)
1110    1110   (resterende 3 zijn allemaal 1 – 0 en dat is 1)
```

Controle: decimaal geldt: 243 – 5 = 238 en 1110 1110 = **0X**EE = 238

Binair vermenigvuldigen en delen met machten van 2

We delen de binaire representatie van 243 door **machten van 2**:

$243 / 2^1 = (1 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0) / 2^1$

$= 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 + \mathbf{(1 * 2^0 / 2^1)}$

$= 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$

$= 0 * 2^7 + 1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$

= 0111 1001

243 = 1111 001**1**
243 >> 1 = **0**111 1001 (schuif alle bits over 1 positie naar rechts)

Bij ***schuiven naar rechts*** met de ***schuif-operator*** `>>` vallen er aan de rechterkant bits “uit” (die ben je dus kwijt). Aan de linkerkant schuiven bij type `unsigned` altijd even zo vele nullen in (bij type `signed` hoeft dat niet altijd het geval te zijn). Schuiven naar rechts is delen door machten van 2

```
243      = 1111 0011  = 243
243 >> 1 = 0111 1001  = 121
243 >> 2 = 0011 1100  = 60
243 >> 3 = 0001 1110  =  30
243 >> 4 = 0000 1111  =  15
243 >> 5 = 0000 0111  =   7
```

We vermenigvuldigen de binaire representatie van 5 met **machten van 2**:

$5 * 2^1 = (0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0) * 2^1$

$= \mathbf{0 * 2^8} + 0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1$

$= 0 * 2^7 + 0 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0$

= 0000 1010

5 = **0**000 0101
5 << 1 = 0000 101**0** (schuif alle bits over 1 positie naar links)

Bij ***schuiven naar links*** met de ***schuif-operator*** << vallen er aan de linkerkant bits "uit" (die ben je dus kwijt). Aan de rechterkant altijd even zo vele nullen in. Schuiven naar links is vermenigvuldigen met machten van 2

```
5      = 0000 0101  =   5
5 << 1 = 0000 1010  =  10
5 << 2 = 0001 0100  =  20
5 << 3 = 0010 1000  =  40
5 << 4 = 0101 0000  =  80
5 << 5 = 1010 0000  = 160
```

In hoofdstuk worden ***bit-operaties*** behandeld. Dit is in C een heel belangrijk onderwerp.

Negatieve getallen

We nemen ook in deze uitleg aan dat we 8 bits beschikbaar hebben voor de opslag van een geheel getal. Dan zijn er 256 mogelijkheden. Wanneer we ons beperken tot niet-negatieve getallen, zijn dit de getallen 0 tot en met 255.

In binaire notatie:

```
0000  0000 = 0
0000  0001 = 1
0000  0010 = 2
0000  0011 = 3
. . . . . .
1111  1110 = 254
1111  1111 = 255
```

Wanneer we ook **negatieve getallen** toestaan, raken we één bit kwijt aan het teken. Het ***teken*** wordt dan (logischer wijze) altijd opgeslagen in het MSB. Een MSB = 0 betekent een positief getal; en een MSB = 1 betekent een negatief getal.

Vervolgens staan we voor een dilemma: maken we evenveel negatieve als positieve getallen? Doen we dat inderdaad, dan spreken we van ***1-complement***. In dat geval heeft ieder positief getal een bijbehorend negatief getal. De consequentie hiervan is, dat er ook een negatieve 0 is. Er zijn nu 127 negatieve getallen, plus ook nog de negatieve 0.

Kiezen we voor slechts één 0, dan is er sprake van ***2-complement***. In dat geval houd je één getal over.
Bij 2-complement zijn er 128 negatieve getallen (-1 tot en met -128).

1-complement

Ieder positief getal (en 0) levert zijn negatieve versie (a + (- a) = 1111 1111)

```
0000  0000 = 0          1111  1111 = - 0
0000  0001 = 1          1111  1110 = - 1
0000  0010 = 2          1111  1101 = - 2
0000  0011 = 3          1111  1100 = - 3
. . . . . .
0111  1110 = 126        1000  0001 = - 126
0111  1111 = 127        1000  0000 = - 127
```

Wanneer bij 1111 1110 = - 1 het getal 0000 0001 = 1 wordt opgeteld, is het resultaat 1111 1111 Dit geldt voor ieder paar uit bovenstaande tabel. Ook voor het paar 1111 1111 en 0000 0000 Iedere negatieve waarde is dus te vinden uit de bijbehorende positieve waarde.

Wanneer we 0000 0001 (= 1) en 1111 1111 bij elkaar optellen, is het resultaat 1 0000 0000 = 2^8 = 256 . Maar er zijn in totaal 255 verschillende getallen, omdat + 0 = - 0 . Dat betekent dat (de decimale) 256 hetzelfde getal moet zijn als (de decimale) 1. Met 1111 1111 = - 0 is de optelling (decimaal) correct. Om de optelling ook binair correct te krijgen, moet de optelling van 0000 0001 (= 1) en 1111 1111 (= - 0) opleveren: 0000 0001 (= 1). Dat is het geval bij: 0000 0001 (= 1) + 0000 0000 (= + 0) = 0000 0001 (= 1)

2-complement

Er is nu slechts één 0. Dit is de gebruikelijke variant. Optellen van 0000 0001 en 1111 1111 levert 0000 0000 (je "loopt er uit" met 1 0000 0000 (= 256)). Omdat 0000 0001 = 1 en 0000 0000 = 0 moet dan gelden: 1111 1111 = -1

```
   0000 0000 = 0
   0000 0001 = 1
   0000 0010 = 2
   0000 0011 = 3
   . . . . . .

   0111 1110 = 126
   0111 1111 = 127
   1111 1111 = - 1
dus
   1000 0000 + 0111 1111       = 1111 1111
   1000 0000 + (decimaal) 127  =  (decimaal) – 1
   1000 0000 = - 127 - 1 = - 128
en
   1000 0001 + 0111 1110       = 1111 1111
   1000 0001 + (decimaal) 126  =  (decimaal) – 1
   1000 0000 = - 126 - 1 = - 127
```

Practicum Week 2 (Default Practicum)

Opdracht 2_1 (zie Vb 6_2 en Vb 6_4)

Lees met `getchar()` een aantal karakters in van toetsenbord. Deze karakters moeten een `int`-getal opleveren voor een (correct) **maandnummer**.
Beveilig je programma tegen **incorrecte invoer**, zoals het intikken van bijvoorbeeld 56 of 123 of ab of abc.
Nadat een correct maandnummer is verkregen, moet het ***switch-statement*** worden losgelaten op de int `maand` De ***switch*** moet een waarde toekennen aan de variabele `maandlengte`. Maak de maandlengte van maand 2 altijd 28

Opdracht 2_2 (zie Vb 6_5 en Vb 6_6)

Schrijf een C-programma waarin een rechthoek moet worden afgedrukt op het beeldscherm. Er wordt via toetsenbord een karakter ingelezen (bijvoorbeeld $). Vervolgens wordt een open rechthoek met breedte b en hoogte h afgedrukt.

bijv. met b= 8 en h = 5: (altijd lastig om in Word een mooi plaatje te maken)

```
$$$$$$$$
$       $
$       $
$       $
$$$$$$$$
```

Opdracht 2_3 (zie Hoofdstuk 2)

Schrijf een programma waarin een positief (niet te groot) int-getal wordt ingelezen van toetsenbord (zoals in opdracht 2_1 het maandnummer is ingelezen). We gaan dit getal met 100 vermenigvuldigen. Breng de vermenigvuldiging tot stand door een aantal keer de schuif-operator << te gebruiken.