Reader TICT-V1ADC-16
Algorithms and Datastructures in C

Cursuseigenaar en auteur: Joop Kaldeway
Versie: 1

Inhoudsopgave

1. Inleiding.

Enkele kenmerken van C.

- C sluit dicht aan bij de hardware. Het wordt vaak gebruikt bij embedded systemen.
- C-code wordt vertaald (of gecompileerd) naar machine-instructies, die door de (Intel- of ARM-)processor worden verwerkt. Een C-programma wordt omgezet naar een uitvoerbaar programma (of een executable). In Windows zijn bestanden met de extensie '.exe' uitvoerbare bestanden. Bij andere talen, zoals Java en Python, wordt de code omgezet naar tussen-code (of byte-code). Deze code kan niet rechtstreeks door een processor worden verwerkt. Dit wordt gedaan door een speciaal programma. Een programma, die tussen-code verwerkt , wordt een 'virtual machine' genoemd.
- Variabelen in C kunnen niet van type veranderen.
- C/C++ is een industrie-standaard

De taal C.

De syntaxis van C is terug te vinden in andere talen: C++, Objective-C, C# , Java, etc.

C is een procedurele taal: C-code is opgebouwd uit functies. Het startpunt van een C-programma is de functie 'main' . Een functie is opgebouwd uit opdrachten.
Er bestaan een aantal C-standaarden. Bekende standaarden zijn ANSI C, C99 (sinds 1999) en C11 (sinds 2011) . In de reader wordt uitgegaan van de standaard C11 of gnu11.

Het eerste C-programma.

Het volgende C-programma drukt "Yes we can" op het scherm af.

```
#include <stdio.h>

int main(void)
{
  printf("Yes we can");

  return 0;
}
```

Toelichting:

- de functie '**main**' geeft de waarde '**0**' terug. Hiermee wordt aangegeven dat alles succesvol is uitgevoerd.
  '`return 0;`' mag bij '**main**' worden weggelaten. Als de return-opdracht ontbreekt, wordt door de compiler '`return 0;`' toegevoegd

- de functie '**main**' roept de functie '**printf**' aan. Deze functie staat in de C-library. In de header-file '**stdio.h**' is '**printf**' gedeclareerd. Als je de header-file uit het programma weglaat, krijg je een 'warning'. In dit geval: '**implicit declaration of function 'printf'** '

- De declaratie kan opgevraagd worden door '**printf**' te selecteren, de rechtermuisknop in te drukken en 'Goto Declaration' te kiezen .

  De functie kun je ook raadplegen op de site
  http://www.cplusplus.com/reference/cstdio/printf/ . Daar staat de declaratie:

  ```
  int printf (const char* format, .... );
  ```

  Parameter-namen mogen worden weggelaten. Op blz. 50 komen we hierop terug.
  De printf-opdracht wordt op blz. 11 behandeld.

- In C wordt iedere opdracht afgesloten met een ';'

- Een C-programma heeft de main-functie als entry-point: het programma wordt hier gestart.

Commentaar in een C-programma.

Commentaar kan op twee manieren gegeven worden:

Optie 1: de tekst omsluiten met /* en */ .

Voorbeeld:

```
/*
============================================
 Name        : mijn_programma.c
 Author      : mijn_naam
============================================
*/
```

Deze methode wordt vaak gebruikt om stukken C-code "weg te commentariëren" .
Het gaat mis als de C-code al commentaar bevat: commentaar binnen commentaar
m.b.v. /* … */ is niet toegestaan.

Optie 2: plaats '//' voor de commentaar-tekst

Voorbeeld:

```
printf("INT_MAX+1: %d\n",INT_MAX+1); // integer overflow
```

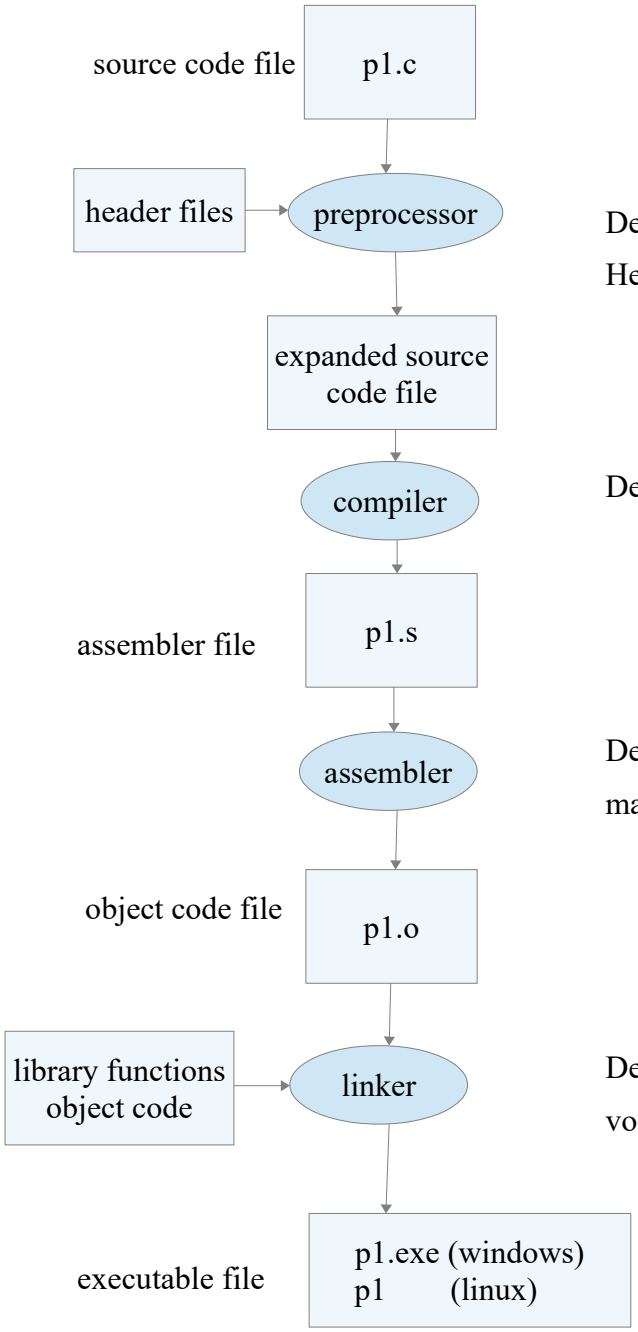
Bij de meeste editors (of IDE's) kun je blokken tekst selecteren en en alle regels laten beginnen met '//'. Bij Windows en Linux kun je de short-cut key CRL+'/' gebruiken.

Oefen-opdracht 1

- Installeer 'mingw' en de ontwikkelomgeving 'CodeLite'. Raadpleeg de installatiehandleiding.
- Test het programma 'eerste_C_programma.c' .
  - Wat gebeurt er als '`#include <stdio.h>`' wordt 'weg gecommentarieerd'?
  - Wat gebeurt er als een ';' wordt weggelaten?
- Raadpleeg de site http://www.cplusplus.com/reference/cstdio/
- Open de file 'stdio.h' en zoek de declaratie van 'printf' .
  (Over de details van 'stdio.h' hoef je je niet te bekommeren.)

Het vertaalproces.

Het vertaalproces van source-code naar uitvoerbaar programma gaat als volgt:



De preprocessor doet tekstuele voorwerk. Header-files worden toegevoegd aan p1.c

De compiler zet c-code om in assembler-code.

De assembler zet assembler-code om in machine-code

De linker combineert de machine-code met voorgecompileerde library-functies

Een 'executable file' bevat instructies, die door de processor kunnen worden uitgevoerd.

2. Datatypes

C bevat de volgende datatypes:

- gehele getallen
- gebroken getallen
- char
- array
- struct
- pointer
- union (wordt niet behandeld)

In C99 zijn datatypes toegevoegd:

- boolean
- long long
- long double

Gehele getallen.

In C zijn voor gehele getallen een aantal datatypen beschikbaar.
Elk datatype heeft een vast aantal bytes en heeft een maximum en een minimum.

De datatype 'int' is de standaard. De getalsrepresentatie is 2-complement.
Andere datatypen zijn: short, long en 'long long' .
Voor elk datatype bestaat een 'signed' en een 'unsigned' versie. Bij 'unsigned' wordt 2-complement niet toegepast: er zijn geen negatieve getallen en de bovengrens is ongeveer twee keer zo groot.

Citaat: (van site http://www.convertforfree.com/twos-complement-calculator/)

Two's complement is a method for representing signed numbers in binary number system. When we want to convert a binary number to two's complement we revert each bit of this number, meaning 1 changes to 0 and 0 changes to 1, and then add 1 to the result. For positive numbers the leftmost bit is 0 and for negative numbers the leftmost bit is 1. The number zero has only one representation (in contrary to one's complement).

Een overzicht:

type	size (aantal bytes)	minimum	maximum	format specifier
int	4 (eis: >= 2)	-2147483648 (-2^31)	2147483647 (2^31-1)	%d
unsigned int	4 (eis: >= 2)	0	4294967295 (2^32-1)	%u
short	2 (eis: >= 2)	-32768 (-2^15)	32767 (2^15-1)	%hd
unsigned short	2 (eis: >= 2)	0	65535 (2^16-1)	%hu
long	4 (eis: >= 4)	-2147483648 (-2^31)	2147483647 (2^31-1)	%ld
unsigned long	4 (eis: >= 4)	0	4294967295 (2^32-1)	%lu
long long	8 (eis: >= 8)	-9223372036854775808 (-2^63)	9223372036854775807 (2^63-1)	%lld
unsigned long long	8 (eis: >= 8)	0	18446744073709551615 (2^64-1)	%llu

De size-waarden kunnen per compiler verschillen. Bij 'int' staat: 'eis: >= 2' . Hiermee wordt bedoeld, dat bij iedere C-compiler de 'size' van 'int' minimaal 2 is.
De 'format specifier' wordt bij de printf-opdracht gebruikt.

Oefen-opdracht 2

- Test het programma 'int_demo.c' .
- Waarom is 2147483647+2147483647 = -2 ?
- Raadpleeg de sites http://en.wikipedia.org/wiki/C_data_types en http://www.convertforfree.com/twos-complement-calculator/
- Bekijk de file 'stdint.h' (In welke map staat de file?)

De printf-opdracht.

Een voorbeeld:

```
int a = 8;
printf("a is gelijk aan %d\n", a);
```

In de string "a is gelijk aan %d\n" wordt "%d" vervangen door de waarde van a.

De printf-functie heeft de volgende opbouw:

```
printf(format-string, arg1, arg2, arg3,...)
```

De format-string bevat format-specifiers: %d, %u, …
De format-specifier geeft aan waar het argument moet staan en hoe het afgedrukt moet worden.

Nog een voorbeeld:

```
int b = 14;
printf("het kwadraat van %d is %d\n",b,b*b);
```

Uitvoer:

```
het kwadraat van 14 is 196
```

Oefen-opdracht 3

- Test het programma 'printf_demo.c' .

Representatie van gehele getallen.

Gehele getallen kunnen ook hexadecimaal of octaal weergegeven worden.

Hexadecimale getallen beginnen met 0x of OX.

Octale getallen beginnen met 0.
Het enige decimale getal dat met een 0 begint is 0 zelf.

Een voorbeeld.

```
printf("%#x %#o", 90, 420);
```

Uitvoer:

```
0x5a 0644
```

We zullen later behandelen hoe gehele getallen binair afgedrukt worden. (zie blz. 62)

Getallen van het type 'long' worden afgesloten met 'L' : 12345678L, 0xff80aabbL
Getallen van het type 'long long' worden afgesloten met 'LL': 9876543210123LL

Een overzicht van format-specifiers:

decimaal	%d
hexadecimaal	%x , met prefix 0x : %#x
octaal	%o , met prefix 0 : %#o
long hexadecimaal	%lx , met prefix 0x : %#lx
long long hexadecimaal	%llx , met prefix 0x : %#llx

Oefen-opdracht 4

- Test het programma 'hexadecimaal_octaal_demo.c' .

Gebroken getallen.

Gebroken getallen bevatten een punt of een exponent.

Bij de 'scientific notation' bevat een gebroken getal een exponent en staat één cijfer (ongelijk aan 0) voor de punt.

Een voorbeeld.

```
printf("%f\n", 123.456);
printf("%.3f\n", 123.456);
printf("%.6e\n",123.456);
puts("");  // puts drukt een string af en gaat
           // naar de volgende regel. Alternatief: printf("\n")
                                                 //   putchar('\n')
printf("%.7f\n",1e-5);
printf("%.3f\n",1.234560e+02);
```

Uitvoer:

```
123.456000
123.456
1.234560e+002

0.0000100
123.456
```

Een overzicht

type	size (aantal bytes)	minimum(positief)	maximum(positief)	format specifier
float	4	1.175494e-38	3.402823e+38	%f %e of %g
double	8	2.225074e-308	1.797693e+308	%f %e of %g
long double	12	3.362103e-4932	1.189731e+4932	%lf %le of %lg

%g: kies de kortste notatie

Een getal van het type 'float' wordt afgesloten met 'f' of 'F' : 12.34f, 0.44F
Een getal van het type 'long double' wordt afgesloten met 'l' of 'L' : 12.34l, 0.44L

Twee getallen op elkaar delen.

Er geldt in de wiskunde (en op rekenmachines) : $\frac{8}{3} = 2.666666...$

In C geldt echter: 8/3 = 2

Bij een deling van twee gehele getallen is de uitkomst ook een geheel getal en wordt alles achter de komma weggelaten.

Je kunt dit op verschillende manieren oplossen:

- Bereken 8.0/3
- Bereken 8/3.0

Een voorbeeld:

```
printf("8/3:   %d\n",8/3);
printf("8.0/3: %f\n",8.0/3);
```

Uitvoer:

```
8/3:   2
8.0/3: 2.666667
```

Oefen-opdracht 5

- Test het programma 'floating_point_demo.c' .
- Raadpleeg de site http://en.wikipedia.org/wiki/Double-precision_floating-point_format

Het type 'char',

In C is het type 'char' is een 1-byte integer-type.
'char' kan equivalent aan 'signed char' of 'unsigned char' zijn. Dit is afhankelijk van de compiler.

Voorbeelden van char-constanten zijn: 'a', 'z', 'A', 'Z', '0', '9' .

Bij invoer van tekst spelen 'white space'-karakters vaak een rol.
'white space'-karakters zijn: ' ' (space), '\t' (tab), '\n' (newline), '\v' (vertical tab), '\f' (feed) en '\r' (carriage return) .

In C geldt: 'a' == 97 (97 = ASCII-waarde van 'a')
M.a.w. 'a' en 97 zijn uitwisselbaar.

De opdracht

```
char ch = 'a';
```

kan vervangen worden door:

```
char ch = 97;
```

Met karakter-constanten kan gerekend worden:

```
'a' + 1   == 'b'
'd' - 'b' == 2
'#' + '#' == 35+35 = 70 = 'F'
```

Uitvoer.

Een karakter wordt afgedrukt m.b.v. de format-specifier '%c':

```
printf("a: %c\n",'a');
```

Als je de format-specifier '%d' gebruikt dan wordt de ASCII-waarde getoond:

```
printf("ASCII-waarde van a: %d\n",'a');
```

Een speciaal karakter is het null-karakter `'\0'`. Deze bevat uitsluitend 0-bits en wordt als afsluiter van strings gebruikt. We komen hier later op terug. (blz. 34)

Tegenwoordig mag een string ook unicode-characters bevatten.
Een unicode-character bestaat vaak kan uit meerdere bytes.

Een voorbeeld:

```
printf("\u03C0");
puts("");
printf("π");                        // Project Editor Preferences... utf-8
puts("");
printf("%d\n",strlen("\u03C0"));   // voeg toe: #include <string.h>
                                   // toon aantal bytes van 'π'
unsigned char s[] = "\u03C0";      // toon de afzonderlijke bytes van 'π'
printf("%#x %#x",s[0],s[1]);
```

Uitvoer:

```
π
π
2
0xcf 0x80
```

In een dosbox kan π niet getoond worden.

Char-functies.

De taal C bevat een aantal char-functies.

```
#include <ctype.h>

int islower(ch);
int isupper(ch);        // these functions return a non-zero value if the
int isalpha(ch);        // test is TRUE, otherwise they return 0 (FALSE)
int isdigit(ch);
int isalnum(ch);
int ispunct(ch);
int isspace(ch);

char tolower(ch);
char toupper(ch);
```

Oefen-opdracht 6

- Test het programma 'char_demo.c' .
  Gebruik bij Windows voor het tonen van π de 'Codelite built in terminal emulator'.
  (Menu Settings → Preferences → Terminal)
- Raadpleeg https://www.cs.swarthmore.edu/~newhall/unixhelp/C_chars.html
- Raadpleeg http://www.cplusplus.com/reference/cctype/

3. Variabelen.

Bij programmeertalen spelen variabelen een belangrijke rol.
Variabelen kunnen o.a. getallen, karakters, strings, lijsten en files zijn.

Een variabele heeft de volgende eigenschappen:

- naam (identifier) . Een naam mag niet beginnen met een cijfer en mag geen spaties bevatten
- waarde (value)
- (geheugen-)adres

Zolang een variabele gedefinieerd is, blijven de naam en het adres hetzelfde maar kan de waarde veranderen. Variabelen worden gebruikt voor het opslaan van tijdelijke gegevens en voor het maken van berekeningen.

In C moet bij de declaratie van variabelen het type meegegeven worden.

```
int i;
double x;
char ch;
```

Het type van een variabele legt vast hoeveel geheugenruimte een variabele nodig heeft en welke bewerkingen mogelijk zijn.

Het is een goede gewoonte een variabele een begin-waarde te geven.
(Dit wordt 'initialisatie' genoemd.)

```
int i = 6;
double x = 7.0;
```

Anders krijg je de warning: **'i' is used uninitialized in this function [-Wuninitialized]**

In C mogen binnen een 'block' variabelen niet twee keer gedeclareerd worden. (Een block is een verzameling van statements tussen '{' en '}' .)

De declaratie

```
int i = 6;
int j = 7;
```

kan ingekort worden:

```
int i = 6,j = 7;
```

Het adres van i wordt aangegeven met '&i' .

Een voorbeeld:

```
int i = 6,j = 7;
printf("waarde van i: %d\n",i);
printf("adres van i: %#x\n",&i);
printf("waarde van j: %d\n",j);
printf("adres van j: %#x\n",&j);
```

Uitvoer:

```
waarde van i: 6
adres van i: 0x29fedc
waarde van j: 7
adres van j: 0x29fed8
```

Wat valt je op?

Samengevat: in C gelden voor variabelen de volgende regels:

- een variabele moet een beginwaarde krijgen
- een variabele mag maar één keer gedeclareerd worden
- een variabele moet worden gebruikt

4. Expressies.

Expressies worden onderverdeeld in:

- rekenkundige expressies
- logische expressies (beweringen)

Rekenkundige expressies.

Voorbeelden van rekenkundige expressies zijn:

```
- 3+4*5+6;
- ((3+4)*5)+6;
- exp(1);
- 4*atan(1);
```

Een rekenkundige expressie berekent een getalswaarde.
Een expressie is opgebouwd uit operanden en operatoren (+ , – , * , /)
Wiskunde-functies, zoals sin(x), arctan(x), exp(x), en bekende wiskundige constanten, zoals 'pi' en 'e', staan in de header-file 'math.h'.
De berekening is gebaseerd op de prioriteiten van de operatoren: vermenigvuldigen gaat voor optellen. Voor het bewerkstelligen van een andere berekeningsvolgorde worden ronde haakjes gebruikt.

3+4*5+6 = 3+20+6 = 23 + 6 = 29
((3+4)*5)+6 = (7*5) +6 = 35 + 6 = 41

Bij C geldt:

- bij gehele getallen kan overflow optreden.
- er bestaat geen operator voor machtsverheffen.

Logische expressies.

Logische expressies bevatten logische operatoren.
Hiermee kunnen beweringen gecombineerd worden.

Een overzicht:

C	betekenis
&&	en
||	of
!	niet

Voorbeelden van logische expressies zijn:

```
2<3 && 3<4
!(2<3)
```

In C levert een logische expressie een waarde op: 0 (false) of 1 (true) .

Logische expressies bevatten naast logische operatoren ook
vergelijkingsoperatoren (< , <= , >, >=, == , !=).

' **2<3 && 3<4** ' wordt als volgt geëvalueerd :

2<3 = 1
3<4 = 1
2<3 && 3<4 = 1 && 1
1 && 1 = 1

Voor logische expressies geldt het volgende:

- ' **2<3 && 3<4** ' en ' **2<3<4** ' worden verschillend berekend
- ' **!(2<3)** ' heeft waarde **0** , maar ' **!2<3** ' heeft waarde **1** .

Het gebruik van de uitdrukking '**2<3<4**' leidt tot een warning.
Het wordt als volgt berekend:

2<3<4 = (2<3)<4 = 1<4 = 1

Toevallig is dit goed.

In andere gevallen gaat het fout:

3<2<1 = (3<2)<1 = 0<1 = 1
-3<-2<-1 = (-3<-2)<-1 = 1 < -1 = 0

Bij de !-operator moet je haakjes gebruiken:

!(2<3) = **!1** = **0** en **!2<3** = **(!2)<3** = **0<3** = **1** (**!2<3** leidt tot een warning)

Het type 'bool'.

Bij C is tegenwoordig het type 'bool' en de waarden true en false gedefinieerd.

De waarden 0, `'\0'`(null character) , NULL(null pointer) worden opgevat als false.
Alles wat ongelijk is aan 0, `'\0'` , NULL wordt opgevat als true.

Om 'bool' te gebruiken moet de header_file <stdbool.h> in de file toegevoegd worden.

Oefen-opdracht 7

- Test het programma 'bool_demo.c' .

De prioriteit van operatoren.

De volgende tabel legt de prioriteit van operatoren van hoog naar laag vast.

Operator Precedence Chart

Operator Type	Operator	Associativity
Primary Expression Operators	`() [] . -> expr++ expr--`	left-to-right
Unary Operators	`* & + - ! ~ ++expr --expr (typecast) sizeof`	right-to-left
Binary Operators	`* / %`	left-to-right
	`+ -`	
	`>> <<`	
	`< > <= >=`	
	`== !=`	
	`&`	
	`^`	
	`|`	
	`&&`	
	`||`	
Ternary Operator	`?:`	right-to-left
Assignment Operators	`= += -= *= /= %= >>= <<= &= ^= |=`	right-to-left
Comma	`,`	left-to-right

Later zal op een aantal operatoren dieper worden ingegaan.

Oefen-opdracht 8

- Test het programma 'expression_demo.c' .
- Schrijf een C-programma, dat 2^{63} uitrekent.

5. Opdrachten.

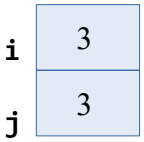
De taal C kent de opdrachten:

- de toekenningsopdracht
- increment- en decrement-opdracht
- de samengestelde opdracht (compound statement)
- keuze-opdrachten: if-statement, if-else-statement, switch-statement
- herhalingsopdrachten: while-statement, do-while-statement, for-statement.
- sprong-opdrachten: break, continue, goto (de goto-opdracht is berucht en wordt afgeraden).

De toekenningsopdracht.

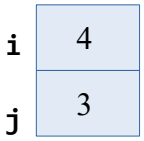
Met een toekenningsopdracht wordt een waarde aan een variabele toegekend.
Dit gaat als volgt:

```
i = 3
j = i
```



j bevat een kopie van i

```
i += 1
```



de waarde van i wordt aangepast

De opdracht " i = 7 " heeft zelf de waarde 7 .

De volgende (verwarrende) opdrachten zijn in C mogelijk:

```
int k = 2;
printf("%d\n",k=3);
printf("%d\n",2*(k=4));
```

Uitvoer:

```
3
8
```

De opdracht

```
i = j = 8;
```

is een verkorte opdracht van:

```
i = (j = 8);
```

Omdat `"j = 8"` de waarde 8 heeft, krijgt `'i'` nu ook de waarde 8 .

Een bekende beginnersfout is de volgende opdracht

```
if (i=3) // warning; moet zijn: (i==3)
   printf("i is gelijk aan 3");
```

Nu wordt altijd

```
i is gelijk aan 3
```

afgedrukt.

Samengevat:

"i = 3" is een opdracht en heeft waarde 3.
"i == 3" is en bewering (heeft waarde 1 als i gelijk aan 3 ;
heeft waarde 0 als i niet gelijk aan 3 is)

Oefen-opdracht 9

- Test het programma 'assignment_demo.c' .

De increment- en decrement-opdracht.

Het komt heel vaak voor dat een variabele met 1 wordt verhoogd.

De variabele 'i' met 1 verhogen wordt gerealiseerd met de volgende opdracht:

```
i = i+1;
```

of:

```
i += 1;
```

In C is het ook mogelijk met de volgende opdrachten:

```
i++;
```

of

```
++i;
```

Het verschil tussen beide opdrachten zien we bij het volgende voorbeeld:

```
i = 7;
j = i++;  // hetzelfde als j = i; i += 1; (i wordt achteraf verhoogd)
printf("i: %d , j: %d\n",i,j);

i = 7;
j = ++i;  // hetzelfde als i += 1; j = i; (i wordt vooraf verhoogd)
printf("i: %d , j: %d\n",i,j);
```

Uitvoer:

```
i: 8 , j: 7
i: 8 , j: 8
```

Vraag: waarom is het volgende niet aan te raden:

```
i = 4;
int j = i++ + i++;
```

De keuze-opdracht.

Een voorbeeld:

```
if (i < j) {
    min = i;
    max = j;
} else {
    min = j;
    max = i;
}
```

Toelichting:

- de test '**i < j**' moet tussen haakjes staan.
- statements die bij elkaar horen worden tussen accolades geplaatst.
  Statements tussen accolades vormen een block. Binnen een block mogen variabelen gedeclareerd worden
- inspringen is bij C niet verplicht maar maakt het wel leesbaar.

Bij het volgende voorbeeld worden geen accolades gebruikt:

```
if (i < j)
    max = j; // toegestaan bij één statement
else
    max = i; // toegestaan bij één statement
```

We adviseren altijd wel accolades te gebruiken:

```
if (i < j) {
    max = j;
} else {
    max = i;
}
```

Bij het toevoegen van code gaat het dan minder fout.

Vraag: wat is er fout aan het volgende voorbeeld?

```
if (i < j)
    max = j;
    min = i;
```

Het compound statement.

Een voorbeeld van een 'compound statement' (of 'block') is:

```
{
    min = i;
    max = j;
}
```

Het is ook toegestaan binnen de accolades variabelen te declareren:

```
{
    min = i;
    max = j;
    int verschil = max-min;
    printf("verschil: %d\n",verschil);
}
```

De variabele 'verschil' is alleen bekend binnen de accolades.
Door de scope van variabelen beperkt te houden, wordt het programma overzichtelijker en is de kans op fouten minder.

Het switch-statement

Het volgende voorbeeld legt op basis van een cijfer een beoordeling vast.
Het cijfer is een geheel getal, groter of gelijk aan 0 en kleiner of gelijk aan 10.

```
if (cijfer<=3)
   printf("slecht\n");
else if (cijfer <= 5)
   printf("onvoldoende\n");
else if (cijfer <= 7)
   printf("voldoende\n");
else if (cijfer <= 9)
   printf("goed\n");
else
   printf("uitstekend\n");
```

Dit kan ook gerealiseerd worden met een switch-statement:

```
switch(cijfer) {
case 0:
case 1:
case 2:
    printf("slecht\n");
    break;
case 4:
case 5:
    printf("onvoldoende\n");
    break;
case 6:
case 7:
    printf("voldoende\n");
    break;
case 8:
case 9:
    printf("goed\n");
    break;
default:
    printf("uitstekend\n");
}
```

Toelichting:

- Gestart wordt bij de eerste case-label
- De opdrachten bij een case-label worden alleen uitgevoerd als 'cijfer' gelijk is aan de case-label
- Als 'cijfer' gelijk is aan de case-label wordt het proces vervolgd bij de volgende case-label
- De break-opdracht verlaat het switch-statement
- Als bij een case-label de break-opdracht ontbreekt wordt het proces vervolgd bij de volgende case-label
- de switch-expressie (in dit voorbeeld: cijfer) moet een integer- of char-expressie zijn
- de case-labels moeten integer- of char-constanten zijn

Het for-statement.

Een voorbeeld:

Opdracht: druk 11^2, 12^2, …, 20^2 af

Dit gaat als volgt:

```
for(int i=11; i<21 ; i++) {
    printf("%d ",i*i);
}
```

De algemene structuur van het for-statement is:

```
 for (init_exp; cond_exp; update_exp)
    loop_body_statement
```

init_exp:	wordt vooraf één keer uitgevoerd
cond_exp:	wordt iedere keer getest voordat 'loop_body_statement' wordt uitgevoerd.
update_esp:	wordt uitgevoerd nadat 'loop_body_statement' is uitgevoerd.

Het for-statement heeft hetzelfde effect als de volgende while-statement:

```
init_exp;
while(cond_exp)  {
   loop_body_statement
   update_exp;
}
```

De opdracht

```
for(int i=11; i<21 ; i++) {
   printf("%d ",i*i);
}
```

kan dus vervangen worden door:

```
int i=11;
while(i<21) {
   printf("%d ",i*i);
   i++;
}
```

Bij het for-statement mogen init_exp, cond_exp en update_exp ook leeg zijn.

Een voorbeeld:

```
int i = 11;
for( ; ; ) { // kan vervangen worden door 'while(true)' of 'while(1)'
    printf("%d ",i*i);
    i++;
    if (i == 21) break;
}
```

Het is een oneindige loop. De break-opdracht sluit de opdracht af.

Een voorbeeld van de opdrachten 'break' en 'continue':

```
int i = 10;
while(1) {
   i++;
   if (i == 15) {
        continue; // sla de rest van de lus over
   }
   printf("%d ", i*i);
   if (i == 20) {
      break;    // verlaat de while-opdracht
   }
}
```

Bij dit voorbeeld wordt 15*15 niet afgedrukt.

Het gebruik van 'break' en 'continue' wordt afgeraden.
Het is minder leesbaar en foutgevoelig.

Oefen-opdracht 10

- Test de programma's 'switch_demo.c' en 'for_demo.c' .
- Vervang bovenstaand programma-deel door een programma zonder 'break' en 'continue' .
- Raadpleeg: http://en.wikipedia.org/wiki/C_syntax#Control_structures
  http://www.tutorialspoint.com/cprogramming/c_decision_making.htm
  http://www.tutorialspoint.com/cprogramming/c_loops.htm

6. Inlezen van gegevens.

De invoer van het toetsenbord wordt eerst in een buffer geplaatst.
Pas nadat op de Enter-toets is gedrukt, wordt de buffer leeg gemaakt en de invoer verwerkt.
Het gebruik van een buffer heeft als voordeel dat tekst-correcties m.b.v. de backspace-toets mogelijk zijn.

In C gaat het inlezen wat lastiger. Met name het inlezen van getallen en strings is complex.

We behandelen eerst inlezen van een karakter. Daarna wordt het inlezen van een getal behandeld.
Het inlezen van een string wordt later behandeld.

Een karakter inlezen.

De functie '`int getchar(void)`' leest 1 karakter in van het toetsenbord.

De tegenhanger van deze functie is: '`int putchar(int ch)`'

Een voorbeeld:

```
printf("tik tekst in en druk daarna op Enter-toets: ");

int ch = getchar();
while (ch != '\n') {
   putchar(ch);
   ch = getchar();
}
```

Bij C mag een toekenningsopdracht in een expressie worden opgenomen.
De code wordt hierdoor compacter:

```
int ch;
while ((ch = getchar()) != '\n') {
   putchar(ch);
}
```

Bij sommige IDE's wordt de uitvoer ook eerst in een buffer geplaatst. De uitvoer vindt pas aan het einde van het programma plaats.
Als je daarop niet wilt wachten kun je uitvoer afdwingen met `fflush(stdout)`
Als je de opdracht `fflush(stdout)` niet steeds wil herhalen kun je aan het begin van het programma de opdracht '**setbuf(stdout, NULL);**' geven. Er wordt dan geen buffer gebruikt.

I.p.v. '`getchar()`' kan ook '`fgetc(stdin)`' of kortweg '`getc(stdin)`' gebruikt worden.
I.p.v. '`putchar(ch)`' kan ook '`fputc(ch,stdout)`' of kortweg '`putc(ch,stdout)`' gebruikt worden.

In C zijn 'stdin', 'stdout' en 'stderr' file-pointers. Files worden in hoofdstuk 14 (blz. 66) behandeld.

Een getal inlezen.

De tegenhanger van 'printf' is 'scanf'.

De scanf-functie heeft de volgende opbouw:

```
int scanf(format-string, arg1, arg2, arg3, ...)
```

Om de variabelen te kunnen wijzigen moeten reference-parameters meegegeven worden. Een reference-parameter wordt voorafgegaan door '&'. Normaal wordt de waarde van een parameter meegegeven. Bij een reference-parameter wordt het adres van de parameter meegegeven i.p.v. de waarde van een parameter. De functie kan nu via dat adres de waarde van de parameter veranderen.

Een voorbeeld:

```
int i;
printf("tik een getal in: ");

while(!scanf("%d",&i)) {
    printf("foutieve invoer\n");
    while(getchar() != '\n')
       ; //sla rest van regel over
    printf("tik een getal in: ");
}
printf("ingevoerde getal: %d",i);
```

In dit voorbeeld wordt

```
scanf("%d",&i) // &i : adres van i
```

gebruikt.

De scanf-functie geeft het aantal geslaagde lees-opdrachten terug. Als de waarde 0 wordt teruggegeven, is de invoer niet goed verlopen.

Nog een voorbeeld: het inlezen van een datum.

```
int dag, maand, jaar;
int n;

printf("voer datum in formaat dd/mm/jjjj: ");
n = scanf("%d/%d/%d",&dag,&maand,&jaar);
printf("n: %d\n",n);
printf("ingevoerde datum: %02d/%02d/%4d\n",dag,maand,jaar);
```

Als de datum goed is ingevuld, heeft 'n' de waarde 3 .
Bij `scanf("%d/%d/%d",&dag,&maand,&jaar)` moet tussen de getallen '/' staan.
Bij `scanf("%d %d %d",&dag,&maand,&jaar)` moet tussen de getallen 'white characters' staan.

Oefen-opdracht 11

Test de programma's ' getchar_putchar_demo.c ' en 'scanf_demo.c' .

7. Arrays.

Met een array kan in één keer een aantal elementen van hetzelfde type worden gedeclareerd.
Alle elementen van de rij staan naast elkaar in het geheugen (adjacent memory location)

Een integer-array wordt als volgt gedeclareerd:

```
int a[6];
```

0	1	2	3	4	5
?	?	?	?	?	?

Een array kan niet langer of korter worden (fixed size)
De elementen a[0], a[1], …, a[5] zijn niet gedefinieerd. Ze bevatten willekeurige waarden.

Soms weet je nog niet hoe groot een array moet zijn.
Een oplossing is: 'variable-length arrays' .
Bij deze constructie wordt eerst de lengte bepaald en daarna het array gedeclareerd.

```
int size;
scanf("%d",&size);
int a[size];
```

De volgende tabel laat de verschillende mogelijkheden van declaratie zien:

declaratie	resultaat
`int a[6];`	? ? ? ? ? ?
`int a[] = {1,4,9,16,25,36};`	1 4 9 16 25 36
`int a[6] = {};`	0 0 0 0 0 0
`int a[6] = {1,2,3};`	1 2 3 0 0 0
`(C99)` `int a[6] = {[1] = 2,[3]= 6};`	0 2 0 6 0 0

Overschrijdingen van de array-grens worden **niet** gedetecteerd. Hier moet de programmeur zelf op letten. Array-grens overschrijdingen kunnen leiden tot onverwachte (fout-)situaties.

Het programma kan zelfs compleet vastlopen. Een programma kan na een array-grens overschrijding een tijd goed werken. Als het dan fout gaat, is de oorzaak van de fout moeilijk te achterhalen. Een C-programmeur moet daarom defensief programmeren: bij gebruik van arrays moet steeds nagegaan worden of de array-grens niet overschreden wordt.

Oefen-opdracht 12

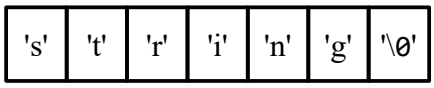
Test het programma's 'array_demo.c ' .

8. Strings.

In C is een string is een character-array, dat wordt afgesloten met '\0'.
Strings worden altijd omsloten door dubbele aanhalingstekens.
Een karakter is omsloten door enkele aanhalingstekens.

Een string wordt als volgt gedeclareerd:

```
char s[] = "string";
```



Andere mogelijkheden voor het declareren van een string zijn:

```
char s[] = {"string"};

char s[] = {'s','t','r','i','n','g','\0'}; // vergeet '\0' niet

char s[7] = {'s','t','r','i','n','g','\0'}; // het aantal elementen is 7
```

Functies, zoals 'printf' en 'strlen' werken alleen goed als de string wordt afgesloten met '\0'.
Als '\0' wordt weggelaten, dan gedraagt het programma zich onvoorspelbaar en kan zelfs vastlopen.

Bij het bepalen van de lengte van een string wordt '\0' niet meegerekend.

Het deelprogramma:

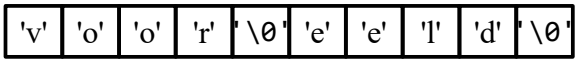
```
#include <string.h>  // nodig voor strlen
printf("strlen(\"string\"): %d\n",strlen("string"));
printf("sizeof(\"string\"): %d\n",sizeof("string"));
```

heeft de volgende uitvoer:

```
strlen("string"): 6   ('\0' wordt niet meegerekend )
sizeof("string"): 7   ('\0' wordt wel meegerekend )
```

Alleen het gedeelte voor '\0' wordt afgedrukt.

```
char s[] = "voorbeeld";
puts(s);
printf("strlen(s5): %d\n",strlen(s5));

s[4] = '\0';
```




```
puts(s);
printf("strlen(s): %d\n",strlen(s));
```

Uitvoer:

```
voorbeeld
strlen(s): 9
voor       (het gedeelte na '\0' wordt genegeerd. )
strlen(s): 4
```

Een char-variabele is 1 byte groot. Unicode karakters, zoals 'π', kunnen beter niet als char-variabelen worden gedeclareerd. Unicode karakters zijn meerdere bytes groot.

De declaratie

```
char ch = 'π';
```

geeft warnings.

Wel is toegestaan:

```
char s[] = "π";

printf("%d\n",strlen(s));
printf("%d\n",sizeof(s));
```

Uitvoer:

```
2
3
```

'lengte' en 'size' worden bepaald door het aantal bytes.

Strings inlezen.

Het inlezen van een string gaat als volgt:

```
char s[100];
printf("tik string in: ");
gets(s);
```

Als de gebruiker meer dan 100 karakters intikt wordt de array-grens overschreden.
De code is dus niet bestand tegen vandalisme. (hufterproof)
Een betere methode is:

```
char s[100];
printf("tik string in: ");
fgets(s,100,stdin);
```

Er worden maximaal 99 karakters ingelezen. Daarna wordt '\0' toegevoegd.
Als minder dan 99 karakters worden ingevoerd, dan wordt '\n' ook ingelezen.
Als 99 of meer karakters worden ingevoerd, dan wordt '\n' niet ingelezen.
Het is dan handig de rest van de regel over te slaan ('rest of line' skippen) .
Als de rest van de regel niet wordt overgeslagen, wordt daar begonnen bij de volgende leesopdracht.
De volgende aanvulling verwijdert de (eventueel) ingelezen '\n' en slaat de rest van de regel over:

```
int len = strlen(s);
if (s[len-1] == '\n') {
  s[len-1] = '\0';  // verwijder '\n' uit string
} else {
  while (getchar() != '\n')
      ; // skip rest van regel
}
```

Strings kunnen ook worden ingelezen m.b.v. 'scanf'.

```
char s[10];
printf("tik string in: ");
scanf("%9s",s);  // maximaal 9 karakters worden gelezen.
                 // zodra een 'white character' wordt gelezen,
                 // wordt het lezen beëindigd
```

Een string met spaties kan als volgt worden ingelezen:

```
char s[10];
printf("tik string met spaties in: ");
scanf("%9[^\n]",s); // maximaal 9 karakters worden gelezen.
                    // zodra een '\n' wordt gelezen,
                    // wordt het lezen beëindigd
```

String-functies.

De taal C bevat een aantal string-functies.

```
#include <string.h>

size_t strlen ( const char * str );
char * strcpy ( char * destination, const char * source );
char * strcat ( char * destination, const char * source );

int strcmp ( const char * str1, const char * str2 );
int strncmp ( const char * str1, const char * str2, size_t num );

const char * strchr ( const char * str, int character );
char * strchr (char * str, int character );
const char * strrchr ( const char * str, int character );
char * strrchr ( char * str, int character );

void * memcpy ( void * destination, const void * source, size_t num );
void * memmove ( void * destination, const void * source, size_t num );
int memcmp ( const void * ptr1, const void * ptr2, size_t num );
const void * memchr ( const void * ptr, int value, size_t num );
```

'size_t' is een unsigned integer type en is minimaal 2 bytes groot.

Oefen-opdracht 13

Test de programma's 'string_demo.c ' en 'gets_demo.c ' .
Raadpleeg http://www.cplusplus.com/reference/cstring/

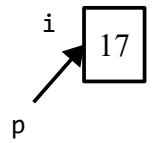
9. Pointers.

Inleiding.

In C is het mogelijk het adres van een variabele als waarde toe te kennen aan een andere variabele. Een dergelijk variabele heet een pointer-variabele.

Een voorbeeld:

```
int i = 17;
int* p = &i; // pointer p = "adres van i"
             // p wijst naar i
```




```
*p = 18; // "de variabele, waar p naar wijst" = 18
         // * : dereferencing operator
printf("i: %d\n",i);
```

Uitvoer:

```
i: 18
```

De waarde van 'i' kan via 'p' ook gewijzigd worden.

De volgende notaties zijn identiek:

```
int* p = ...

int * p = ...

int *p = ...
```

Het maakt niet uit waar '*' staat.

Bij het 'multiple declaration' geldt het volgende:

De declaratie

```
int* p, j;
int* q, *r;
```

is een korte schrijfwijze van:

```
int* p; int j;
int* q; int* r;
```

Soms wil je een pointer niet ergens naar laten wijzen, maar wel definiëren.
Een dergelijke pointer krijgt de waarde NULL. NULL is het adres met waarde 0.

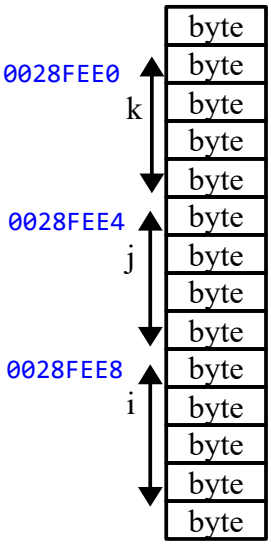
Pointer-waarden afdrukken.

Het volgende voorbeeld drukt de adressen van drie gedeclareerde variabelen af.

```
int i = 17, j = 27, k = 37;
printf("&i: %p, &j: %p, &k: %p\n",&i,&j,&k);
```

Uitvoer: `&i: 0028FEE8, &j: 0028FEE4, &k: 0028FEE0`

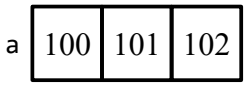
Bij deze implementatie worden de adressen steeds lager.



Pointers en arrays.

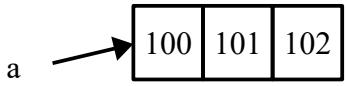
Bekijk het volgende voorbeeld:

```
int a[] = {100, 101, 102};

printf("a: %p\n",a);
printf("&a[0]: %p\n",&a[0]);
```



Uitvoer:

```
a: 0028FED4
&a[0]: 0028FED4
```

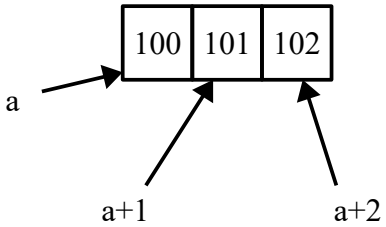


Er geldt: **a == &a[0] .**

De variabele 'a' heeft twee betekenissen:

- het is de naam van het array
- het wijst naar het eerste element van het array

Verder wijst 'a+1' naar 'a[1]' en wijst 'a+2' naar 'a[2]' .



Algemeen: **a+i == &a[i]** m.a.w. 'a+i' wijst naar 'a[i]'
***(a+i) == a[i]**

Met een pointer een array doorlopen.

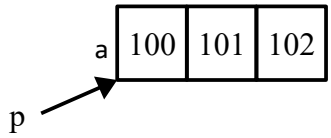
Het is gebruikelijk een array met een index-variabele te doorlopen.

Een voorbeeld:

```
int a[] = {100, 101, 102};

for (int i=0; i<3; i++) {
    printf("%d ",a[i]);
}
```

Uitvoer:

```
100 101 102
```

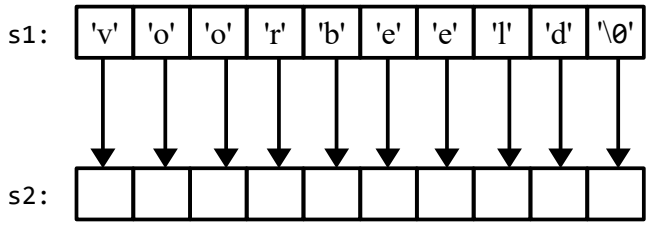


Het is ook mogelijk een array te doorlopen m.b.v. een pointer-variabele

```
int *p = a;

for (int i=0; i<3; i++) {
      printf("%d ",*p);
      p++; // p wijst nu naar het volgende element (4 bytes verder)
}
```

'p' wijst naar een int-variabele.
De compiler berekent 'p+1' op basis van 'sizeof(int)' . Als sizeof(int) = 4 wijst 'p+1' 4 bytes verder.

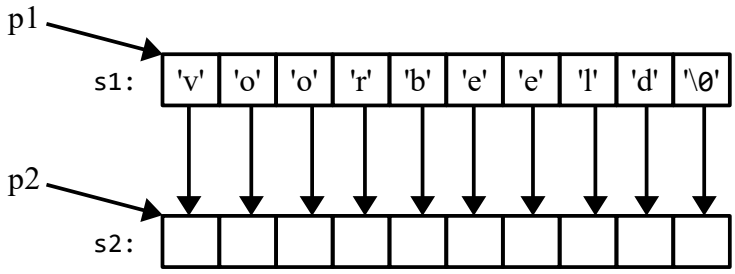
Een uitgebreider voorbeeld: kopieer een string in een andere string.



Methode 1: gebruik indices:

```
char s1[] = "voorbeeld";
char s2[strlen(s1)+1];

for (int i=0; i<strlen(s1)+1; i++) {
  s2[i]=s1[i];
}
```

Methode 2: gebruik pointers:




```
char* p1 = s1;
char* p2 = s2;

while(*p2 != '\0') {
  *p2 = *p1;
  p1++; // p1 wijst 1 positie verder
  p2++; // p2 wijst 1 positie verder
}
```

De code kan nog compacter:

```
char* p1 = s1;
char* p2 = s2;

while((*p2++ = *p1++) != '\0');  // stopt als *p2 == '\0'
```

Wees voorzichtig met dergelijke compacte code: het is moeilijk leesbaar en de kans op fouten is groot.

Array versus pointer-variabele.

Er zijn belangrijke verschillen tussen een array en een pointer-variabele.

Voorbeeld:

```
int a[] = {100, 101, 102};
int* p = a;
```

Array-elementen mogen gewijzigd worden: de opdracht ' **a[0] = 200** ' is toegestaan.

De array zelf mag niet gewijzigd worden
De array 'a' is constant: het is een 'vaste' pointer en mag niet gewijzigd worden.

De opdrachten:

```
a = p+1;
a++;
```

zijn niet toegestaan.

'a' wijst naar 'a[0]' en dit element heeft een vast adres.

In C-jargon: " **a is geen left-value** ". 'a' mag bij een assignment niet links van het '='-teken staan.

De opdracht 'a++' geeft de volgende foutmelding: **`lvalue required as increment operand`**

Pointer-variabelen mogen gewijzigd worden.

De opdrachten

```
p = a+1;
p++;
```

zijn geoorloofd.

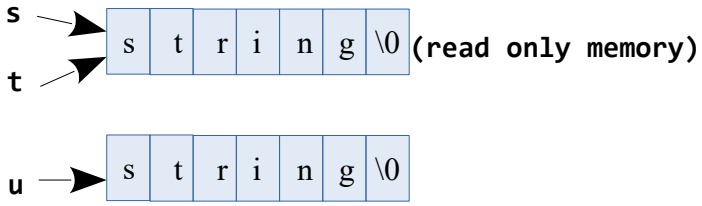
Char-array versus 'literal string'.

Bekijk het volgende voorbeeld:

```
char *s = "string";   // s wijst naar een "string literal"
char *t = "string";   // t wijst naar een "string literal"
char u[] = "string";  // hetzelfde als
                      // char u[] = {'s','t','r','i','n','g','\0'};
printf("%p\n", s);
printf("%p\n", t);
printf("%p\n", u);
```

De uitvoer is (in dit geval):

```
0040a068
0040a068
0029fec9
```



De pointer-variabelen 's' en 't' wijzen naar dezelfde string.

De opdracht `s[3] = 'o';` is **niet** toegestaan. De string literal 't' zou dan ook gewijzigd worden. In Windows crasht het programma. In Linux krijg je een 'segmentation fault (core dumped)' melding. De pointer-variabelen 's' en 't' wijzen naar een string die staat in een read only data segment.

De variabele 'u' wijst naar een andere string. De opdracht `u[3] = 'o';` is wel toegestaan.

Om jezelf tegen het crashen van het programma te beschermen, kun je beter

```
char* s = "string";
```

vervangen door:

```
const char* s = "string";
```

De opdracht: `s[3] = 'o';` leidt nu tot een compiler-error.

De variabele **'s'** mag wel naar een andere literal wijzen dus `s = "strong";` is wel toegestaan

M.b.v. typecasting kan het programma alsnog crashen:

```
char *s2= (char *)s;
s2[3] = 'o';
```

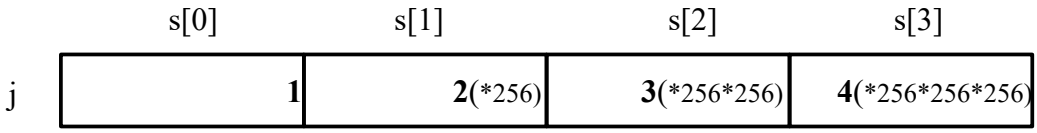
M.a.w. de compiler houdt niet alles tegen.

Typecasting

Er bestaan ook nuttige vormen van typecasting.

Je kan hiermee achterhalen hoe een int-variabele is opgebouwd uit 4 bytes.
M.b.v. typecasting worden de vier bytes afzonderlijk getoond.

Een demo:




```
j = 1 + 2*256 + 3*256*256 + 4*256*256*256;
char* s = (char*)&j; // typecasting: &j is van het type int*
printf("j: %#xd\n",j);
printf("&j: %p\n",&j);

for (int i=0; i <4; i++) {
   printf("%p ",&s[i]);
}
putchar('\n');
for (int i=0; i <4; i++) {
   printf("%d ",s[i]);
}
putchar('\n');
```

Uitvoer:

```
j: 0x4030201
&j: 0028FEA8
0028FEA8 0028FEA9 0028FEAA 0028FEAB
1 2 3 4
```

waarde van j: x4030201
byte-volgorde: 1 2 3 4

Er zijn twee manieren voor het opslaan van bytes:
– little-endian byte volgorde: minst significante byte heeft het laagste adres
  (little end first)

– big-endian byte volgorde : meest significante byte heeft het hoogste adres
  (big end first)

Bij Intel-processors is gekozen voor de little-endian architectuur.
Bij Motorola-processors is gekozen voor de big-endian architectuur.
(boek: 9.2.1, blz. 67 en 20.1.6, blz. 169)

Oefen-opdracht 14

Test het programma's 'pointer_demo.c ' .
Raadpleeg https://betterexplained.com/articles/understanding-big-and-little-endian-byte-order/
https://en.wikipedia.org/wiki/Endianness

10. Struct

Een 'struct' combineert een aantal verschillende waarden tot één geheel.
Bij een array moeten de elementen van hetzelfde type zijn. Bij een 'struct' hoeven de 'members' niet van hetzelfde type te zijn.

Een voorbeeld:

```
struct art {
      int code;
      char naam[20];
      int voorraad;
};

int main(void)
{
      struct art art1 = {1234,"soldeerbout",12};
      struct art art2 = {5555,"nijptang",14};
      printf("Voorraad van artikel %d (%s) is %d\n",
                        art1.code,art1.naam,art1.voorraad);
      printf("Voorraad van artikel %d (%s) is %d\n",
                        art2.code,art2.naam,art2.voorraad);
}
```

Met behulp van een typedefinitie kan i.p.v. 'struct art' een naam worden gebruikt.

```
typedef struct {
    int code;
    char naam[20];
    int voorraad;
} Artikel;

int main(void)
{
    Artikel art1 = {1234,"soldeerbout",12};
    ...
}
```

De members kunnen zonder initialisatie een waarde krijgen:

```
    Artikel art1 = {1234,"soldeerbout",12};
```

kan vervangen worden door:

```
    Artikel art1;
    art1.code = 1234;
    strcpy(art1.naam,"soldeerbout"); // niet: art1.naam = "soldeerbout"
                                     // art.naam is geen lvalue
    art1.voorraad = 12;
```

In het geheugen kunnen structs soms gaten bevatten.

Bekijk het volgende voorbeeld:

```
typedef struct {
      int i,j;
      char c;
} Structvb1;

typedef struct {
      int i,j;
      char c1,c2,c3,c4;
} Structvb2;

int main(void)
{
      printf("sizeof(Structvb1): %d\n", sizeof(Structvb1));
      printf("sizeof(Structvb2): %d\n", sizeof(Structvb2));

       ....
}
```

Uitvoer:

```
sizeof(Structvb1): 12
sizeof(Structvb2): 12
```

Bij Structvb1 zijn maar 4+4+1= 9 bytes nodig. Toch worden 12 bytes gebruikt.
Structs werken met veelvouden van 4 bytes. Het geclaimde geheugen hoeft niet geheel gevuld te zijn.

Oefen-opdracht 15

Test het programma's 'struct_demo.c ' .
Raadpleeg https://en.wikipedia.org/wiki/Data_structure_alignment
http://fresh2refresh.com/c-programming/c-structure-padding/

11. Tweedimensionale arrays: een array van arrays.

Een array van strings is een voorbeeld van een twee-dimensionale array.

Een array van strings kun je in C op twee manieren declareren.

Methode 1: met pointers:

```
char* dag[] = {"zondag","maandag","dinsdag","woensdag","donderdag",
               "vrijdag","zaterdag"};
```

De strings "zondag", "maandag", etc mogen nu niet gewijzigd worden.

Methode 2: met de index-notatie:

```
char dag[7][10]= {"zondag","maandag","dinsdag","woensdag","donderdag",
                  "vrijdag","zaterdag"};
```

of

```
char dag[][10]= {"zondag","maandag","dinsdag","woensdag","donderdag",
                 "vrijdag","zaterdag"};
```

(Vraag: waarom 10?)

Nu is het wel toegestaan de strings te wijzigen.

```
for (int k=0; k<7;k++) {
  printf("%s\n",dag[k]);
}
putchar("\n");

dag[0][0] = 'Z';         // dag[0] == "zondag"
**(dag + 1) = 'M';       // *(dag + 1) == "maandag"

for (int k=0; k<7;k++) {
  printf("%s\n",dag[k]);
}
putchar("\n");
```

Uitvoer:

```
zondag
maandag
dinsdag
woensdag
donderdag
vrijdag
zaterdag

Zondag
Maandag
dinsdag
woensdag
donderdag
vrijdag
zaterdag
```

Een voorbeeld gebaseerd op 'Binary Sudoku':

De puzzel

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

kan in een C-programma als volgt worden gedeclareerd:

```
int matrix[][10] = {{-1,-1,-1,-1,-1,-1, 0, 1,-1, 1},
                    { 1,-1,-1,-1,-1,-1,-1,-1,-1, 1},
                    {-1,-1, 0,-1, 1,-1,-1,-1,-1,-1},
                    {-1,-1,-1,-1, 0,-1,-1,-1,-1,-1},
                    {-1,-1,-1,-1,-1,-1, 0,-1,-1, 1},
                    {-1,-1,-1,-1,-1, 1,-1,-1, 1,-1},
                    {-1,-1,-1,-1,-1,-1, 0,-1,-1, 0},
                    { 0,-1,-1,-1,-1,-1,-1,-1,-1,-1},
                    {-1,-1, 1, 1,-1,-1,-1,-1,-1, 1},
                    { 0,-1, 1,-1,-1,-1,-1,-1, 0,-1}};
```

Oefen-opdracht 16

Test het programma 'tweedimen_array_demo.c ' .

12. Functies.

Inleiding.

In C heeft een functie de volgende opbouw:

```
<return-type> f(<arg1-type> arg1,<arg2-type> arg2, ...)
{
   ...
   ...
   return ...;
}
```

Een voorbeeld:

```
int sum(int x,int y) {
   return x+y;
}
```

Een lege lijst parameters wordt met **void** aangegeven:

```
<return-type> f(void)
{
    ...
}
```

Als niets wordt teruggegeven wordt dit ook met **void** aangegeven:

```
void f(<arg1-type> arg1,<arg2-type> arg2, ...)
{
    ...
}
```

De assert-functie.

Soms moet een parameter aan bepaalde voorwaarden voldoen.
Dit kan met de assert-functie afgedwongen worden.

```
#include <assert.h>

double reciprocal(double x)
{
    assert(x != 0);
    return 1/x;
}
```

Als x == 0 dat wordt het programma afgebroken wordt aangegeven in welke file en op welke regel de fout optreedt. Hierdoor is een fout eenvoudiger te achterhalen.

Functie-prototype.

De functie-definitie mag ook na de functie-aanroep staan. De functie moet dan wel vooraf gedeclareerd zijn:

```
int sum(int x,int y); // prototype: functie declaratie

int main(void)
{
   int z = sum(2,3);
   printf("z: %d",z);

   return 0;
}

int sum(int x,int y) // functie definitie
{
   return x+y;
}
```

In de functie-declaratie wordt vastgelegd hoeveel parameters worden meegegeven, wat het type van de parameters is en wat het type van de return-waarde is.
Een functie-declaratie wordt ook wel 'prototype' genoemd.
Omdat de namen van de parameters niet relevant zijn, worden ze vaak weggelaten

M.a.w.

```
int sum(int x,int y);
```

kan vervangen worden door

```
int sum(int, int);
```

Prototypes worden vaak in header-files opgenomen.

Het gebruik van parameters.

Functies kunnen met steeds andere parameters aangeroepen worden.
Het resultaat van een functie kan ook als parameter dienen.

Een voorbeeld:

```
z = sum(sum(2,3),4);
printf("z: %d\n",z);
```

Uitvoer:

z: 9

Oefen-opdracht 17.

Test het programma 'function_prototype_demo.c ' .

De parameter-overdracht: call by value.

Van een parameter wordt eerst een kopie gemaakt.
De statements binnen de functie worden uitgevoerd met het kopie. De meegegeven parameter wordt niet gewijzigd. Deze vorm van parameter-overdracht wordt 'call by value' genoemd.

Een voorbeeld:

```
int succ(int i)
{
   printf("inside function: i = %d\n",i);
   i++; // kopie van i wordt gewijzigd
   printf("inside function: i = %d\n",i);
   return i;
}

int main(void)
{
   int i = 20;
   printf("outside function: i = %d\n",i);
   int j = succ(i);
   printf("j: %d\n",j);
   printf("outside function: i = %d\n",i);

   return 0;
}
```

Uitvoer:

```
outside function: i = 20
inside function: i = 20
inside function: i = 21
j: 21
outside function: i = 20 // i blijft ongewijzigd
```

Oefen-opdracht 18:

Test het programma 'call_by_value_demo.c ' .

Pointer-parameters.

Via pointers kunnen variabelen wel gewijzigd worden.

Een voorbeeld:

```
void swap1(int i,int j)
{
	printf("inside function: i = %d; j = %d\n",i,j);
	int h = i;
	i = j;
	j = h;
	printf("inside function: i = %d; j = %d\n",i,j);
}

void swap2(int* pi,int* pj)
{
	int h = *pi;
	*pi = *pj;
	*pj = h;
}

int main(void)
{
	int i = 20, j= 30;
	printf("outside function: i = %d; j = %d\n",i,j);
	swap1(i,j);
	printf("outside function: i = %d; j = %d\n",i,j);

	swap2(&i,&j); // geef de adressen van i en j mee
	printf("outside function: i = %d; j = %d\n",i,j);

	return 0;
}
```

Uitvoer:

```
outside function: i = 20; j = 30
inside function: i = 20; j = 30
inside function: i = 30; j = 20
outside function: i = 20; j = 30   // i en j blijven ongewijzigd
outside function: i = 30; j = 20
```

Bij de definitie worden pointer-parameters gebruikt: `void swap2(int* pi,int* pj) {..}`
Bij de aanroep worden adressen meegegeven: `swap2(&i,&j);`

Oefen-opdracht 19.

Test het programma 'pointer_parameters_demo.c ' .

Array-parameters.

Raadpleeg: programma: array_parameter_demo.c

Voor array-parameters gelden de volgende eigenschappen:

- als een parameter een array is kunnen de array-elementen binnen de functie gewijzigd worden. De array zelf (d.w.z. de pointer naar het eerste element) blijft ongewijzigd.
- een functie kan de lengte van een meegegeven array niet achterhalen. Deze moet ook meegegeven worden.
- de formele parameters " int a[] " en " int* a " zijn identiek. Een actuele array-parameter is in feite het adres van het eerste element van het array.

Een voorbeeld:

```
void show_array(int a[],int n)
//void show_array(int* a,int n) // "int a[]"en "int* a" zijn identiek
{
      for(int i=0;i<n;i++) {
            printf("%d ",a[i]);
      }
}

void swap(int a[],int i,int j)
//void swap(int* a,int i,int j)
{
      int h = a[i];
      a[i] = a[j];
      a[j] = h;
}

int main(void)
{
    int a[] = {20,30,40};
    int n = sizeof(a)/sizeof(a[0]);

    printf("a: ");
    show_array(a,n);
//    show_array(&a[0],n); // a en &a[0] zijn identiek
    putchar('\n');

    swap(a,0,1);

    printf("a: ");
    show_array(a,n);
    putchar('\n');

    return 0;
}
```

Uitvoer:

```
a: 20 30 40
a: 30 20 40
```

Oefen-opdracht 20.

Test het programma 'array_parameter_demo.c ' .

Struct-parameters.

Van een struct-parameter wordt een kopie gemaakt.
Voor het wijzigen van een struct-parameter moet het adres van de variabele meegegeven worden.

Een voorbeeld:

```
typedef struct {
	double x;
	double y;
} Point;

void show_point(Point p) {
	printf("(%.2f,%.2f)", p.x, p.y);
}

Point rotate_90_degree_clockwise1(Point p) { // p wordt niet gewijzigd
	Point q = {p.y, -p.x};
	return q;
}

void rotate_90_degree_clockwise2(Point *pp) { // *pp wordt gewijzigd
	double x = pp->x; // "pp->x" is een verkorte notatie voor "(*pp).x"
	double y = pp->y;
	pp->x = y;
	pp->y = -x;
}

int main(void) {
	Point p = {3.0, 4.0};
	printf("p: ");
	show_point(p);
	putchar('\n');

	Point q = rotate_90_degree_clockwise1(p); // p wordt niet gewijzigd
	printf("q: ");
	show_point(q);
	putchar('\n');

	p = rotate_90_degree_clockwise1(p); // p wordt gewijzigd
	printf("p rotated: ");
	show_point(p);
	putchar('\n')

	rotate_90_degree_clockwise2(&p); // p wordt gewijzigd
	printf("p twice rotated: ");
	show_point(p);
	putchar('\n');

	return 0;
}
```

Uitvoer:

```
p: (3.00,4.00)
q: (4.00,-3.00)
p rotated: (4.00,-3.00)
p twice rotated: (-3.00,-4.00)
```

Bij de uitdrukking " **(*pp).x** " zijn de haakjes verplicht. (Waarom?)
I.p.v. " **(*pp).x** " wordt de beter leesbare notatie " **pp → x** "gebruikt.

Oefen-opdracht 21

Test het programma 'struct_parameter_demo.c ' .

Globale variabelen.

Globale variabelen worden buiten een functie gedeclareerd.

In C:

```
int global_count = 0; // initialisatie is niet verplicht, default: 0
                      // initialisatie wordt wel aangeraden

void f(void) {
      printf("global_count: %d\n",++global_count);
}

int main(void) {
      f();
      f();
      global_count++;
      printf("global_count: %d\n", global_count);

      return 0;
}
```

Maak niet onnodig gebruik van globale variabelen. Dergelijke variabelen kunnen op allerlei plaatsen worden gewijzigd. Het opsporen van fouten wordt daardoor moeilijk.

Oefen-opdracht 22

Test het programma 'global_variable_demo.c' .

Local static variabelen.

Locale static variabelen zijn alleen bekend binnen de functie. Nadat de functie is uitgevoerd blijven ze echter bestaan.

Een voorbeeld:

```
void f(void) {
      static int local_static_count = 0;
         // alleen bij de eerste aanroep van f vindt de initialisatie plaats
      printf("aanroep %d\n",++local_static_count);
}

int main(void) {
      f();
      f();

      return 0;
}
```

Uitvoer:

```
aanroep 1
aanroep 2
```

Oefen-opdracht 23

Test het programma 'local_static_variable_demo.c' .
Probeer ook zonder 'static' om verschil te zien.

Bibliotheek-functies.

De kracht van C ligt in de meegeleverde library's. Deze bevatten tal van nuttige functies.

Een overzicht is te vinden op de 'C Reference Card' en de site
http://www.cplusplus.com/reference/clibrary/

Een globale indeling van de standaard libraries is gebaseerd op de header-files:

ctype.h:	Character Class Tests:	gaat na of een letter een getal is of lowercase is, etc
string.h:	String Operations	
stdio.h:	Input/Output:	printf , scanf, etc
stdarg.h:	Variable Argument Lists:	hiermee kan een functie een willekeurig aantal argumenten meegegeven worden
stdlib.h:	Standard Utility Functions:	system, random- en conversie-fucnties
time.h:	Time and Date Functions	
math.h:	Wiskunde-functies	
limits.h:	onder- en bovengrenzen van gehele getallen	
float.h:	onder- en bovengrenzen van gebroken getallen	

Verder zijn voor allerlei toepassingen extra libraries: netwerk-applicaties, grafische applicaties, database-applicaties, etc.

Een functie als parameter.

Een functie kan als parameter meegegeven worden aan een andere functie.

Functies worden ook opgeslagen in het geheugen.
In C verwijst de functienaam naar het beginadres van de functie.

Een functienaam is dus een pointer. Het kan ook als parameter meegegeven worden bij een (andere) functie.

Een voorbeeld:

```
typedef double (*Function)(double); // Function is een pointer-type
                                    // '*' kan weggelaten worden
double f(double x)
{
    return x*x-2;
}

double g(double x)
{
    return x*x-3;
}

double bepaal_nulpunt(Function f, double a, double b)
{
      assert(f(a)*f(b) < 0);
      while (b-a > 1.0e-10) {
            double mid = (a+b)/2;
            if (f(mid)*f(a) > 0) { // f(mid) en f(a) hebben hetzelfde teken
                  a = mid;
            }
            else {
                  b = mid;
            }
      }
      return a;
}

int main(void)
{
    double x = bepaal_nulpunt(f,1.0,2.0);
    printf("nulpunt f: %.10f\n",x);

    x = bepaal_nulpunt(g,1.0,2.0);
    printf("nulpunt g: %.10f\n",x);
    return 0;
}
```

Uitvoer:

```
nulpunt f: 1.4142135623
nulpunt g: 1.7320508076
```

Oefen-opdracht 24

- Test het programma 'function_parameter_demo.c' .
- Vergelijk de antwoorden met $\sqrt{2}$ en $\sqrt{3}$ op de rekenmachine.

13. Bitoperaties.

Inleiding.

Datatypen zijn opgebouwd uit bits. In C is het mogelijk op bit-niveau te opereren.

De volgende operatoren kunnen gebruikt:

```
&     : bitsgewijs 'and'
|     : bitsgewijs 'or'
^     : bitsgewijs 'xor'
~     : inversie van alle bits
<<    : left shift
>>    : right shift
```

Op bit-niveau geeft de volgende tabel het effect van de operatoren '&' , '|' en '^' weer:

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Op bit-niveau geldt het volgende: a&b = min(a,b)
a | b = max(a,b)
a ^ b = (a+b)%2

Een voorbeeld op byte-niveau:

	7	6	5	4	3	2	1	0	(bitnummers)
value = 0x6b	0	1	1	0	1	0	1	1	
mask = 0xf0	1	1	1	1	0	0	0	0	
value & mask (bit 0 t/m 3 uit)	0	1	1	0	0	0	0	0	
value | mask (bit 4 t/m 7 aan)	1	1	1	1	1	0	1	1	
value ^ mask (toggle bit 4 t/m 7)	1	0	0	1	1	0	1	1	

Het afdrukken van een bitpatroon.

In C wordt een bitpatroon als volgt afgedrukt:

```
void bitprint(char ch)
{
   for (int i = 0;i<8;i++) {
      if (ch & 0x80) { // check meest linkse bit
          putchar('1');
      } else {
          putchar('0');
      }
      teken <<=1; // schuif 1 positie naar links
   }
   putchar('\n');
}
```

Toelichting:

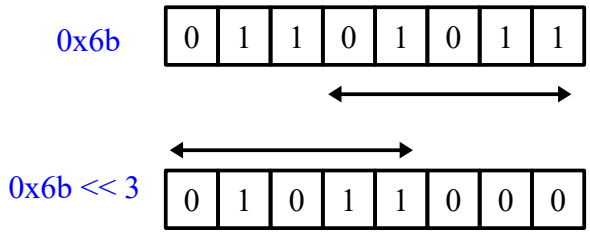
0x80 heeft bitpatroon 1000 0000
ch & 0x80 == 0x80 als meest linkse bit van ch de waarde 1 heeft
ch & 0x80 == 0 als meest linkse bit van ch de waarde 0 heeft

In de code wordt gebruik gemaakt van de shift-oprator << . Deze wordt op de volgende pagina uitgelegd.

De shift-operators << en >> .

Met '<<' wordt een bitpatroon naar links geschoven.
De vrijgekomen plaatsen krijgen de waarde '0'.

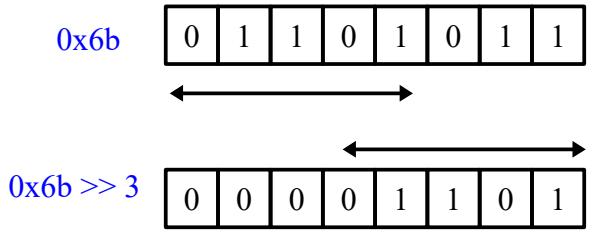
Een voorbeeld:

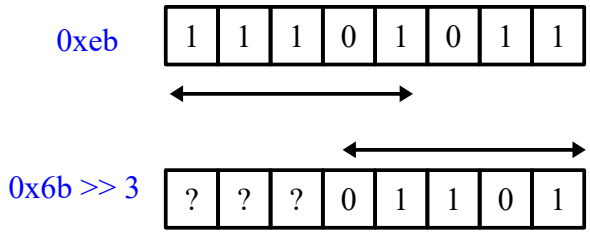


Met '>>' wordt een bitpatroon naar rechts geschoven.
Welke waarden de vrijgekomen plaatsen krijgen hangt bij C af van de implementatie.
In twee situaties weten we zeker dat de vrijgekomen plaatsen de waarde '0' krijgen:

- als de meest linkse bit 0 is
- als de variabele van een 'unsigned type' is.

Voorbeeld:





De code

```
bitprint((char)0xeb);  // 0x80 is van type 'int'
bitprint((char)0xeb >> 3);
bitprint((unsigned char)0xeb >> 3);
```

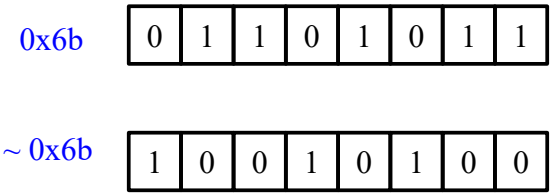
geeft de volgende uitvoer:

```
11101011
11111101  // opgevuld met '1' (implementatie-afhankelijk)
00011101  // opgevuld met '0'
```

De inversie-operator '~' .

De operator '~' zet alle bits om: van 0 naar 1 en van 1 naar 0.

Voorbeeld:



Een bit setten.

Het setten van bit 3 gaat als volgt :

```
char ch;
ch = ch | (1 << 3) ; // of : ch = ch | 0x08;
                     // 1 << 3 heeft bitpatroon 0000 1000
```

Het laatste statement kan vervangen worden door:

```
ch |= (1 << 3) ;
```

Een bit resetten.

Het resetten van bit 3 gaat als volgt :

```
char ch;
ch = ch & ~(1 << 3) ; //  of ch = ch & 0xF7;
                      //  ~(1 << 3) heeft bitpatroon 1111 0111
```

Het laatste statement kan vervangen worden door:

```
ch &= ~(1 << 3) ;
```

Een bit inverteren.

Het inverteren van bit 3 gaat als volgt :

```
char ch;
ch = ch ^ (1 << 3) ; //  of ch = ch ^ 0x08;
```

Het laatste statement kan vervangen worden door:

```
ch ^= (1 << 3) ;
```

Een bitstring omkeren.

De functie 'reverse' geeft bij een gegeven byte de omgekeerde byte terug:

```
char reverse(char ch)
{
	char result = 0;
	if(ch & 1) {		// ga na of de meest rechtse bit 1 is
		result |= 1;	// geef de meest rechtse bit van result de waarde 1
	}
	for(int i=0; i < 7;i++) {
		result <<= 1;		// verschuif result 1 positie naar links
		ch >>= 1;		// verschuif bits 1 positie naar rechts
		if(ch & 1) {		// ga na of de meest rechtse bit 1 is
			result |= 1; // geef de meest rechtse bit van result de
		}					// waarde 1
	}
	return result;
}
```

Oefen-opdracht 25

- Test het programma 'bit_operations_demo.c' .

14. Files.

Schrijven naar een tekstfile.

In C wordt gebruik gemaakt van een file-pointer. Dit is een pointer naar een FILE-variabele. Het type FILE is in 'stdio.h' gedefinieerd als een struct-type. In de FILE-variabele staan de gegevens over de file. Deze informatie moet kunnen worden aangepast. Daarom wordt een pointer gebruikt.

Voor het openen van een file wordt de volgende functie gebruikt:

```
FILE* fopen(const char* filename, const char* mode);
```

Het bovenstaande programma in C:

```
FILE* out_file;
const char* filename = "output.txt";

out_file = fopen(filename,"w");

fputc('v',out_file); // schrijf één karakter weg
fputc('b',out_file);
fputc('\n',out_file);

fputs("een\ntwee\ndrie\n",out_file); // schrijf een string weg

fprintf(out_file,"%d %f\n",5,sqrt(5)); // uitvoer met format-string
fclose(out_file);
```

Oefen-opdracht 26

- Test het programma 'file_demo.c' .

File-modes.

De basis-modes bij het openen van een file zijn:

mode	omschrijving	kenmerken
r	read, lezen vanaf het begin	de file moet bestaan, lezen vanaf het begin
w	write, schrijven	als de file niet bestaat wordt de file gemaakt als de file bestaat wordt de inhoud overschreven
a	append, toevoegen	als de file niet bestaat wordt de file gemaakt het schrijven gebeurt aan het einde van de file

Nadat de gegevens naar de file zijn weggeschreven of ingelezen, moet de file worden gesloten:

```
fclose(out_file);
```

Bij het wegschrijven van data is de fclose-opdracht belangrijk. Aan een file zijn interne buffers gekoppeld. Deze kunnen nog gegevens bevatten, die nog niet zijn weggeschreven. Door de fclose-opdracht wordt de data uit deze buffers weggeschreven.
Je kunt dit ook tussendoor doen met de opdracht:

```
fflush(out_file);
```

Schrijffuncties.

Voor het wegschrijven van gegevens naar een file worden de volgende functies gebruikt:

functie	omschrijving
`int fputc ( int character, FILE* stream );`	schrijf karakter naar file
`int fputs ( const char* str, FILE* stream );`	schrijf string naar file
`int fprintf (FILE* stream, const char* format, ... );`	schrijf data met format-string

Lezen uit een tekstfile.

Een voorbeeld:

```
FILE* in_file;
int c;

in_file = fopen(filename,"r");
c = fgetc(in_file); // lees één karakter in
printf("%c",c);
c = fgetc(in_file);
printf("%c",c);
c = fgetc(in_file);
printf("%c",c);

char s[100];          // neem aan dat de lengte van een regel < 99
fgets(s,100,in_file); // lees string in
printf("%s",s);
fgets(s,100,in_file);
printf("%s",s);
fgets(s,100,in_file);
printf("%s",s);

int i;
double d;
fscanf(in_file,"%d %lf\n",&i, &d); // invoer met format-string
printf("%d %f\n",i,d);

fclose(in_file);
```

Leesfuncties.

Voor het inlezen van gegevens naar een file kunnen de volgende functies gebruikt worden:

functie	**omschrijving**
`int fgetc ( FILE* stream );`	lees karakter van file
`char * fgets ( char* str, int num, FILE* stream );`	lees string van file
`int fscanf ( FILE* stream, const char* format, ... );`	lees data met format-string

Het inlezen van één karakter gebeurt met de functie:

```
int fgetc( FILE* stream );
```

Er wordt een int-waarde teruggegeven en geen char-waarde.
De reden is dat als het einde van de file is bereikt -1 wordt teruggegeven.
De file kan ook bytes met waarde 255 met bitpatroon 1111 1111 bevatten.
Er geldt: (char)255 = -1

Als fgetc een char-waarde zou teruggeven, dat zijn bij de return-waarde -1 twee situaties mogelijk:

- het einde van de file is bereikt
- een byte met waarde 255 is gelezen

Als een int-waarde wordt teruggegeven, hebben we dit probleem niet:

- als het einde van de fiel is bereikt wordt -1 teruggegeven
- als een byte met waarde 255 is gelezen wordt 255 teruggegeven

ASCII-files en binaire files.

Raadpleeg: programma: ASCII_vs_binary_file_demo.c,

Een ASCII-file bevat uitsluitend 7-bit ASCII-codes. Eenvoudige editors en source-editors gebruiken vaak ASCII-files.

MS-DOS en Windows gaan anders om met ASCII-files dan Linux:

1. Bij het wegschrijven wordt '\n' omgezet in '\r\n'
2. Bij het lezen wordt '\r\n' omgezet '\n'
3. Bij het lezen wordt CTRL-Z (0x1a) geïnterpreteerd als 'end of file'.

Een voorbeeld:

```
FILE* out_file, *in_file;
const char* filename = "output.txt";

out_file = fopen(filename,"w");  // schrijf als text-file
fputs("een\ntwee\ndrie",out_file);
fclose(out_file);

in_file = fopen(filename,"rb");  // lees als binaire file in
c = fgetc(in_file);
while (c != EOF) {
      printf("%d ",c);
      c = fgetc(in_file);
}
fclose(in_file);
putchar('\n'); putchar('\n');
```

Uitvoer:

```
0x65 0x65 0x6e 0xd 0xa 0x74 0x77 0x65 0x65 0xd 0xa 0x64 0x72 0x69 0x65
```

0xd correspondeert met **\r**
0xa correspondeert met **\n**

Het volgende voorbeeld laat zien wat er gebeurt als je CTRL-Z ergens in een stuk tekst plaatst.

```
filename = "output2.txt";
int CTRL_Z = 0x1a;

out_file = fopen(filename, "w");
fputs("een\ntwee\n", out_file);
fputc(CTRL_Z, out_file);
fputs("drie", out_file);
fclose(out_file);

in_file = fopen(filename, "r");
c = fgetc(in_file);
while (c != EOF) {
      printf("%#x ", c);
      c = fgetc(in_file);
}
fclose(in_file);
putchar('\n'); putchar('\n');

in_file = fopen(filename, "rb");
c = fgetc(in_file);
while (c != EOF) {
      printf("%#x ", c);
      c = fgetc(in_file);
}
fclose(in_file);
```

Uitvoer:

```
0x65 0x65 0x6e 0xa 0x74 0x77 0x65 0x65 0xa

0x65 0x65 0x6e 0xd 0xa 0x74 0x77 0x65 0x65 0xd 0xa 0x1a 0x64 0x72 0x69 0x65
```

Oefen-opdracht 27

- Test het programma 'ASCII_vs_binary_file_demo.c' .

15. Parameters voor main().

Aan een programma kunnen parameters meegegeven worden.

Tot nu toe ziet de main-functie er als volgt uit:

```
int main(void)
{
   ...
   ...
}
```

Aan de main-functie kunnen ook parameters meegegeven worden:

Een voorbeeld:

De file 'main_parameters_demo.c' bevat de code.

```
int main(int argc, char* argv[])
{
      for(int i=0;i<argc;i++) {
            printf("argv[%d]: %s\n",i,argv[i]);
      }

      return 0;
}
```

Bij de aanroep:

main_parameters_demo arg1 arg2 arg3

Het volgende wordt nu getoond:

argv[0]: main_parameters_demo
argv[1]: arg1
argv[2]: arg2
argv[3]: arg3

Oefen-opdracht 28

Test het programma 'main_parameters_demo.c' .

16. Meer bronbestanden.

Raadpleeg: project: multiple_source_files_demo

Grote C-programma's worden tekstueel opgesplitst over meerdere tekstfiles.
Globale variabelen kunnen vanuit een ander C-bestand gebruikt worden.

Een variabele uit een ander bestand wordt met het keyword 'extern' aangegeven.

Een voorbeeld:

```
/* func.h */

void func1(void);
extern int glob1;
extern int glob2;
```

```
/* main.c */

#include <stdio.h>
#include "func.h"

int glob1 = 5;

int main(void)
{
    func1();
    printf("glob1 is %d, glob2 is %d\n",
                              glob1,glob2);

    return 0;
}
```

```
/* func.c */

#include <stdio.h>
#include "func.h"

int glob2;
// static: extern niet bereikbaar
static int glob3 = 333;

static void func2(void)
{
    puts("func2 aangeroepen");
}

void func1(void)
{
    func2();
    glob2 = 144;
    glob1 = 256;
}
```

Het project bevat drie files:

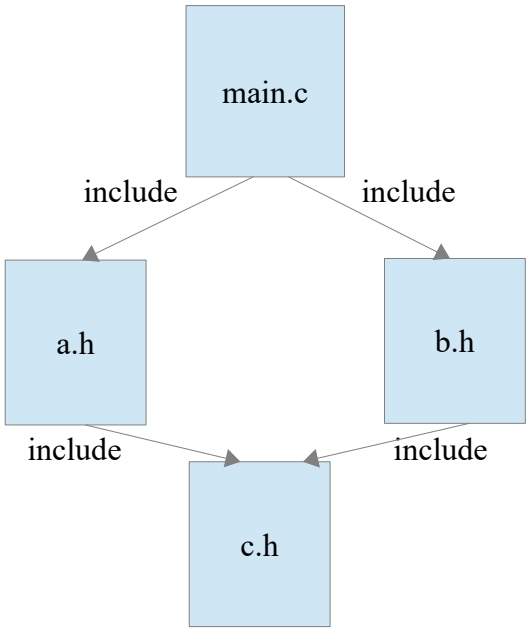
- twee source-files: 'main.c en 'func.c'
- één header file: 'func.h'

Static variabelen en functies zijn alleen binnen de file, waarin ze zijn gedefinieerd, beschikbaar.

Als verwezen wordt naar een externe variabele, die niet bestaat, genereert de linker de foutmelding: **`undefined reference to ...`**

Het gebruik van een header-file is niet verplicht, maar vanwege de veiligheid is het wel standaard: alle source-files gebruiken dan dezelfde functie-prototypes en daardoor op elkaar afgestemd.

Als een file meerdere header-files invoegt, en de header-files op hun beurt ook header-files invoegen, kan het voorkomen dat onbedoeld dezelfde header-file meer dan één keer wordt ingevoegd.
Dit leidt tot dubbele definities en errors.



' c.h 'wordt twee keer ingevoegd.

Om dit voorkomen heeft een header-file vaak de volgende opbouw:

```
#ifndef _FILE_NAME_H
#define _FILE_NAME_H

...

#endif /* _FILE_NAME_H */
```

De inhoud tussen `#ifndef` en `#endif` wordt nu alleen bij de eerste include-opdracht toegevoegd.

Oefen-opdracht 29

Test het programma 'multiple_source_files_demo.c' .

17. Macro's.

Een voorbeeld van een macro is:

```
#define sqr(x) x*x
```

De macro wordt verwerkt door de preprocessor.

' **sqr(12)** ' wordt vervangen door ' **12*12** '

Het gaat fout als 'x' wordt vervangen door '3 + 4'.

' **sqr(3 + 4)** ' wordt vervangen door ' **3 + 4*3+4** ' (= 19)

Een macro, die beter voldoet, is:

```
#define sqr(x) (x)*(x)
```

' **sqr(3 + 4)** ' wordt nu vervangen door ' **(3 + 4)*(3+4)** ' (= 49)

Bij macro's worden daarom standaard de parameters omsloten door ronde haakjes.

Nog een voorbeeld:

```
#define subtract(x,y)  x-y
#define subtract2(x,y) (x)-(y)
```

' **– substract(2,1-4)** ' wordt vervangen door **– 2 – 1 – 4** (= -7)
' **– substract2(2,1-4)** ' wordt vervangen door **– (2) – (1 – 4)** (= 1)

Een macro, die beter voldoet, is:

#define subtract3(x,y) ((x)-(y))

' **– substract3(2,1-4)** ' wordt vervangen door **– ((2) – (1 – 4))** (= – 5)

Een macro wordt daarom standaard omsloten met ronde haakjes.

Een leuke macro is:

```
#define MAX(a,b) ((a) <(b)? (b) : (a))
```

Met dergelijke macro's moet je voorzichtig omgaan.
Welke waarden hebben x en y na de volgende opdrachten?

```
int x = 5, y = 10;
int z = MAX(x++,y++);
```

Oefen-opdracht 30

Test het programma 'macro_demo.c' .

18. Recursie.

Inleiding

Recursieve constructies bevatten een verwijzing naar zichzelf.

In de praktijk komen recursieve structuren ook voor: ieder mens heeft een vader. Iedere vader heeft een vader, etc.

Een technisch voorbeeld: iedere directory bestaat uit files en subdirectories. Iedere subdirectory bestaat uit files en subdirectories, etc.

Wat is het voordeel van recursie?

- het is een andere benadering van problemen
- het sluit meer aan bij wiskunde
- het is eenvoudiger na te gaan of een algoritme correct is
- de code is korter en helderder
- problemen kunnen m.b.v. recursie opgesplitst worden in deel-problemen.

Wat is het nadeel van recursie?

- het vraagt een andere (abstracte) manier van denken (is dit echt een nadeel?)
- recursieve code is (iets) trager dan niet-recursieve code

Recursieve code kan altijd omgezet worden naar niet-recursieve code. Soms moet in de niet-recursieve code dan een stack gebruikt worden.

Een risico bij een recursieve functie is, dat de functie zichzelf eindeloos kan aanroepen.
Als een functie wordt aangeroepen, worden de parameters en de lokale variabelen op een programma-stack geplaatst. Nadat de functie is uitgevoerd worden deze gegevens verwijderd.
Als een functie zichzelf eindeloos aanroept loopt de programm-stack vol en leidt dit tot stackoverflow.
Bij Windows loopt het programma loopt dan vast. Bij Linux krijg je dan een 'segmentation fault'-error .

Voorbeeld: bereken 1 + 2 + 3 + … + n voor een gegeven n.

Oplossing 1: zonder recursie.

```
int som1(int n)
{
    int som = 0;
    for (int i=1; i <= n; i++) {
        som = som + i;
    }
    return som;
}
```

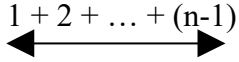
Oplossing 2: maak gebruik van wiskunde:

$$1 + 2 + 3 + \ldots + n = \frac{n*(n+1)}{2}$$

```
int som2(int n)
{
      return (n*(n+1))/2;
}
```

Oplossing 3: los het probleem recursief op

Als S(n) = 1 + 2 + … + n
dan is S(1) = 1
en S(n) = 1 + 2 + … + (n-1) + n = S(n-1) + n als n >1



S(n-1)

```
int som3(int n)
{
    if (n == 1) {    // basisgeval
        return 1;
    } else {
        return som3(n-1) + n;
    }
}
```

'som3' roept zichzelf n keer aan.

We zijn er van uitgegaan dat n >= 1 is.
Het programma loopt vast als n < 1.

We kunnen deze aanname met een assert-functie vastleggen:

```
int som3(int n)
{
    assert(n > 0);

    if (n == 1) {    // basisgeval
        return 1;
    } else {
        return som3(n-1) + n;
    }
}
```

Het programma loopt nog steeds vast, maar je krijgt de reden te zien.

De faculteit van n

n! = n∗(n-1)∗ … ∗ 2∗1 (n! wordt uitgesproken als n-faculteit)
Het aantal rangschikkingen van n elementen is n! .
Het aantal groepen van k elementen uit een groep van n elementen is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$ ($\binom{n}{k}$ wordt uitgesproken als "n over k")

n! wordt als volgt berekend:

Oplossing 1 : zonder recursie.

```
long long fac1(int n)
{
    long long fac = 1LL;
    for (int i=2; i <= n; i++) {
        fac *= i;
    }
    return fac;
}
```

Oplossing 2 : met recursie:

Als fac(n) = n∗(n-1)∗ … ∗ 2∗1

dan is fac(1) = 1
en fac(n) = n ∗ fac(n-1) (n > 1)

We spreken af dat fac(0) = 1

```
long long fac2(int n)
{
    assert(n >= 0);
    if (n == 0) { // basisgeval
        return 1LL;
    } else {
        return n*fac2(n-1);
    }
}
```

De methode 'fac2' roept zichzelf n keer aan.

Voor recursie gelden de volgende regels:

- er moet een basisgeval zijn, waarbij geen recursie wordt gebruikt.
- de methode moet eindigen: na een aantal recursieve aanroepen wordt een basisgeval bereikt.

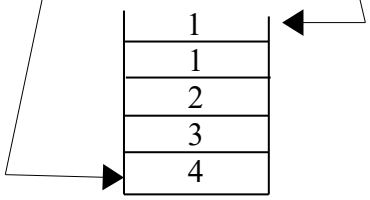
Hoe werkt recursie?

We illustreren dit aan de hand van een voorbeeld.
Neem als voorbeeld de methode fac2

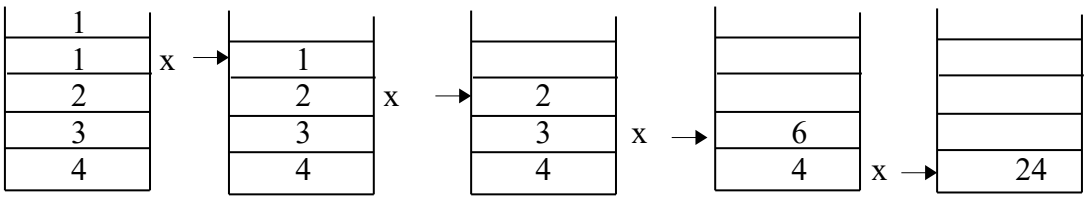
Volgens het algoritme is

fac(4) =
4 * fac(3) =
4*(3*fac(2)) =
4*(3*(2*fac(1))) =
4*(3*(2*(1*fac(0)))) =
4*(3*(2*(1*1))) =
4*(3*(2*1)) =
4*(3*2) =
4*6 =
24

De waarden 4, 3 , 2, 1 en 1 worden achtereenvolgens op een stack geplaatst.



Vervolgens worden steeds de bovenste twee elementen verwijderd en het product op de stack gezet.



Resultaat: 24.

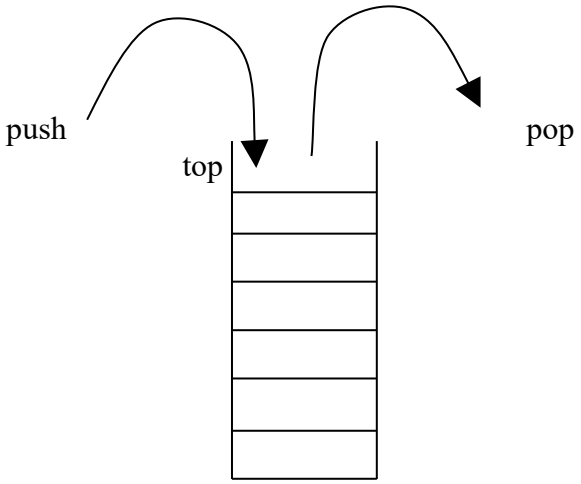
Oefen-opdracht 31

Test het programma 'recursive_function_demo.c' .
Raadpleeg

https://www.topcoder.com/community/data-science/data-science-tutorials/an-introduction-to-recursion-part-1/
http://mooreccac.com/kcppdoc/Recursion.htm
https://www.khanacademy.org/computing/computer-science/algorithms/recursive-algorithms/a/recursion

19. Stack.

Een stack is gebaseerd op het LIFO-principe: Last In First Out



In C kan een stack kan als volgt worden geïmplementeerd:

stack.h:

```
#ifndef _STACK_H
#define _STACK_H

#define STACKSIZE 100

typedef struct {
	int a[STACKSIZE];
	int top;
} Stack;

void init_stack(Stack* ps);
void push(Stack* ps, int data);
int pop(Stack* ps);
void show(Stack s);

#endif /* _STACK_H */
```

stack.c:

```c
#include <stdio.h>
#include <assert.h>
#include "stack.h"

void init_stack(Stack* ps)
{
	ps->top = -1;
}

void push(Stack* ps, int data)
{
	assert(ps->top < STACKSIZE-1);
	(ps->top)++;
	ps->a[ps->top] = data;
}

int pop(Stack* ps)
{
	assert(ps->top > -1 );
	int data = ps->a[ps->top];
	ps->top -= 1;
	return data;
}

void show(Stack s)
{
	if (s.top > -1) {
		printf("%d", (s.a)[s.top]);
		for (int i = (s.top)-1 ; i > -1; i--){
			printf("->%d", s.a[i]);
		}
		putchar('\n');
	} else {
		puts("empty stack");
	}
}
```

main.c:

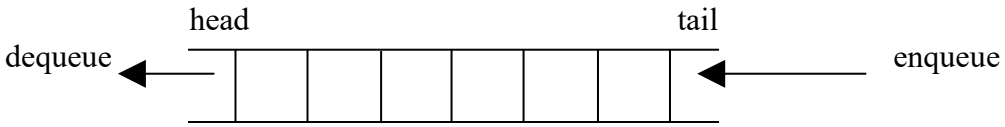
```c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

int main(void)
{
	Stack stack = {{0},-1};

	for (int i = 1; i < 6; i++) {
		push(&stack, i);
		printf("push(%d): ", i);
		show(stack);
	}

	for (int i = 1; i < 6; i++) {
		int j = pop(&stack);
		printf("pop()  : %d\n",j);
		printf("stack  : ");
		show(stack);
	}

	return 0;
}
```

Oefen-opdracht 32

Test de files: 'stack.c', 'stack.h' en 'main.c' in het project 'stack_demo'.

20. Queue.

Een Queue is gebaseerd op het FIFO-principe : First In First Out



Enqueue: voeg een element achteraan toe.
Dequeue: haal een element vooraan weg

Queues worden gebruikt bij buffering. Voorbeelden van queues zijn: eventqueue, printqueue, en taskqueue.

In een practucum-opgave wordt hiermee geoefend.