EE1400: Programmeren in C

BSc. EE, 1e jaar, 2012-2013, 2e college

Arjan van Genderen, Computer Engineering

20-11-2012

Agenda

- A.s. woensdagmiddag 14.00 (!) – 14.45 uur, DTC gebouw (Kluyverweg 4+6), zaal 1, voortgangstoets.
  - Leerstof: 1e en 2e college.
  - Boek en slides mogen er bij gehouden worden
- Uiterlijk 2 dec 24.00 uur: uploaden van practicum opdrachten uit hoofdstuk 1 t/m 3 op https://cpm.ewi.tudelft.nl
  - Houd je aan de regels/richtlijnen die in de practicumhandleiding worden gegeven !
- Volgende hoorcollege: dinsdagochtend 4 dec.

Hoorcollege 2

- De fundamentele datatypen
- Programmabesturing
- Nassi-Shneiderman diagrammen
- Functies

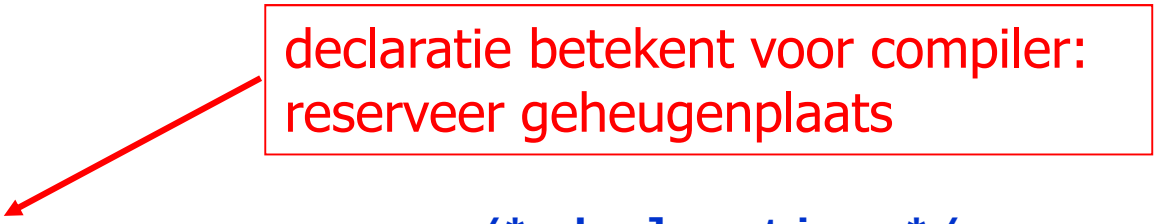
Corresponderende stof in boek: Hoofdstuk 3 t/m 5
Voor Nassi-Shneiderman diagrammen, zie practicumhandleiding

De fundamentele datatypen

Declaraties, Expressies en Assignments

```
#include <stdio.h>

int main(void)
{
   int     a, b, c;                    /* declaraties */
   float   x, y = 3.3, z = -7.7;       /* declaraties met
                                          initialisaties */
   printf ("Input two integers: ");    /* functie aanroep */
   scanf ("%d%d", &b, &c);             /* functie aanroep */
   a = b + c;                          /* assignment */
   x = y + z;                          /* assignment */
```



Let op de betekenis van de '='
De '=' betekent in wiskunde "is gelijk aan"
en in C "wordt gelijk gemaakt aan" (toekenning)

Data representatie - bitreeksen

Variabelen worden met bitreeksen gerepresenteerd in de computer.

getal	representatie
0	00000000
1	00000001
2	00000010
3	00000011
4	00000100
5	00000101
.	. . . .
254	11111110
255	11111111

karakter	representatie
.	. . . .
‘a’	01100001
‘b’	01100010
‘c’	01100011
‘d’	01100100
‘e’	01100101
‘f’	01100110
.	. . . .

ASCII codering

Uiteraard: hoe meer bits per reeks, hoe meer waarden mogelijk:

32 bits $\rightarrow 2^{32} = 4294967296$ waarden

Voor bijvoorbeeld: unsigned integers van 0 … 4294967295

of: signed integers van -21474483648 … +21474483647

Naast type **`int`** (bijv. 32 bits) kent C ook **`short`** (bijv. 16 bits) en **`long`** (bijv. 64 bits). Het aantal bits is systeem afhankelijk.

Data representatie: talstelsels

decimaal: 107 $= 1 \times 10^2 + 0 \times 10^1 + 7 \times 10^0$

binair: 1101011 $= 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$= 64 + 32 + 8 + 2 + 1 = 107$ (decimaal)

In de digitale wereld gebruiken we ook nog:

octaal: 153 $= 1 \times 8^2 + 5 \times 8^1 + 3 \times 8^0$
$= 64 + 40 + 3 = 107$ (decimaal)

notatie in C:
```
k = 0153;
```

hexadecimaal: 6B $= 6 \times 16^1 + 11 \times 16^0$ (A = 10, B = 11, … F = 15)
$= 96 + 11 = 107$ (decimaal) notatie in C: `m = 0x6B;`

Het karakter 'a' met binaire representatie 01100001 ← 8 bits = 1 Byte
kan ook worden geinterpreteerd als een integer met waarde

$0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
$= 64 + 32 + 1 = 97$

Zo is in C `c = 'a';` gelijk aan `c = 97;`

Data representatie - special characters

Enige speciale karakters:

naam	geschreven in C	integer waarde
newline	\n	10
horizontal tab	\t	9
vertical tab	\v	11
null character	\0	0
alert (bell)	\a	7
backslash	\\	92
double quote	\”	34
single quote	\’	39

Data representatie – negatieve getallen

Standaard kunnen short, int en long zowel positieve als negatieve getallen representeren.

Het is ook mogelijk om onderscheid te maken bij de declaratie:

`unsigned int`	alleen positieve getallen
`signed int`	positieve en negatieve getallen

Bij de interne representatie in de computer is bij negatieve getallen het meest linkse bit 1. Daardoor kan bij een overflow van een positief getal een negatief getal ontstaan !

Data representatie - float en double

Floating point getallen worden in C beschreven met een decimale punt en/of een exponent:

exponent

3.14159
0.1234567e6 (= 0.1234567 x 10^6)
333.7777e-22 (= 333.777 x 10^{-22} = 0.333777 x 10^{-19})

De interne representatie is m.b.v. een fractie en een exponent.

Bijvoorbeeld voor type float:

precisie is 6 decimale cijfers

$\pm\ 0.d_1d_2d_3d_4d_5d_6 \times 10^n$ waarbij $-38 < n < +38$

range

En voor type double:

$\pm\ 0.d_1d_2d_3d_4 \cdots d_{14}d_{15} \times 10^n$ waarbij $-308 < n < +308$

Het teken, het getal $d_1 \cdots d_x$ en het getal n worden hierbij uiteraard ook weer met bitreeksen gecodeerd.

Hou rekening met eindige precisie en range !
Bij overflow geen foutmelding, wel incorrecte resultaten !

Overzicht fundamentele datatypen

Met typische waarden voor grootte in bytes
(herinner: 1 byte = 8 bits).

char	1	signed char	1	unsigned char	1
short	2	int	4	long	8
unsigned short	2	unsigned	4	unsigned long	8
float	4	double	8	long double	12

De sizeof operator geeft de grootte van een type in bytes:

```
printf ("   int: %d bytes\n", sizeof (int));
```

Output:

```
   int: 4 bytes
```

Mathematische functies

In de mathematics library van C zijn functies beschikbaar zoals:

sqrt() pow() exp() log()
sin() cos() tan()

```
#include <math.h>
#include <stdio.h>

int main(void)
{
   double   two_pi = 2.0 * M_PI;   /* M_PI in math.h */
   double   h      = 0.1;          /* step size */
   double   x;

   for (x = 0.0; x < two_pi; x += h)
      printf("%5.1f: %.15e\n",
         x, sin(x) * sin(x) + cos(x) * cos(x));
   return 0;
}
```

zou 1 moeten zijn voor elke x volgens goniometrie wet

Output:

```
  0.0: 1.000000000000000e+000
  0.1: 1.000000000000000e+000
  0.2: 1.000000000000000e+000
  0.3: 9.999999999999999e-001
  0.4: 1.000000000000000e+000
            etc.
```

vanwege eindige preciesie

Type conversie

- Impliciete (automatische) conversie
  - Verschillende regels
  - Bijvoorbeeld: wanneer bij een optelling één operand van het type double is dan wordt de andere operand ook geconverteerd naar een double

```
int i;
double d;

/* type van (i + d) is double */
```

- Expliciete conversie (ook wel: casting)
  - Door (*type*) voor de operand te zetten
  - Bijvoorbeeld:

```
int i;
double d;

scanf ("%d", &i);
d = (double) i / 3;   /* om afkappen te voorkomen */
```

Programmabesturing

De if en if else statements

```
if (expr)
   statement
```

```
a = 1;
if (b == 3)
   a = 5;
printf("%d", a);
```

```
if (a == 3) {
   b = 5;
   c = 7;
}
```

tussen haakjes: compound statement

```
if (expr)
   statement1
else
   statement2
```

```
if (cnt == 0) {
   a = 2;
   b = 3;
   c = 5;
}
else {
   a = -1;
   b = -2;
   c = -3;
}
```

een logische expressie levert waarde 1 op wanneer hij waar is, anders 0

Wanneer ***expr*** een waarde ongelijk 0 heeft dan wordt het statement achter de **`if`** uitgevoerd, anders het statement achter **`else`** (indien aanwezig)

Operators voor logische expressies

Resultaat van expressies met deze operators is 1 (*true*) of 0 (*false*)

relationele operators	kleiner dan	<
	groter dan	>
	kleiner dan of gelijk aan	<=
	groter dan of gelijk aan	>=
gelijkheids operators	gelijk aan	==
	niet gelijk aan	!=
	(unaire) negatie	!
logische operators	logische and	&&
	logische or	||

Ook hier operator prioriteit en associativiteit van toepassing (zie boek)

Logische expressie voorbeelden

```
-1.3 >= (2.0 * x + 3.3)
x < (x + y)
(3 < j) && (j < 5)
k != 4 && k != 6 && (n == 1 || n == 3)
!(a < b || c < d)
```

Merk op dat gevaarlijk is (wanneer d1 en d2 float of double): `d1 == d2`

Veiliger is: `(d1 >= d2 * 0.999999) && (d1 <= d2 * 1.000001)`

Short-circuit (of lazy) evaluation van expressie:

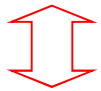
```
int cnt = 0;
while (++cnt <= 3 && (c = getchar ()) != EOF) {
       ......  /* do something */
```

leest karakter;
geeft EOF wanneer
geen input meer

Wanneer `++cnt <= 3` *false* is wordt het volgende karakter **niet** gelezen

De while en for statements

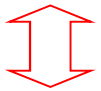
```
while (expr)
   statement
```



```
expr1;
while (expr2){
   statement
   expr3;
}
```

```
for (expr1; expr2; expr3)
   statement
```

```
i = 1;
while (i <= 5) {
   sum += i;
   ++i;
}
```



```
for (i = 1; i <= 5; ++i) {
   sum += i;
}
```

Het do while statement

```
do
   statement
while (expr)
```

```
statement
while (expr)
   statement
```

```
do {
   printf ("Input a positive integer: ");
   scanf ("%d", &n);
   if (error = (n <= 0))
      printf ("\nERROR: Do it again!\n\n");
} while (error);
```

De break en continue statements

```
while (1) {
   scanf ("%lf", &x);
   if (x < 0.0)
      break;            /* exit loop if x is negative */
   printf ("%f\n", sqrt (x));
}
```

```
for (i = 0; i < TOTAL; ++i) {
   c = getchar ();
   if (c >= '0' && c <= '9')
      continue;         /* go to end of loop */

   ... /* process other characters */

   /* 'continue' transfers control to here
      to begin next iteration */
}
```

Het switch statement

```
switch (c) {
   case 'a':
      ++a_cnt;
      break;
   case 'b':
      ++b_cnt;
      break;
   case '0':
   case '1':
      ++bit_cnt;
      break;
   default:
      ++other_cnt;
      break;
}
```

Nassi-Schneiderman Diagrammen

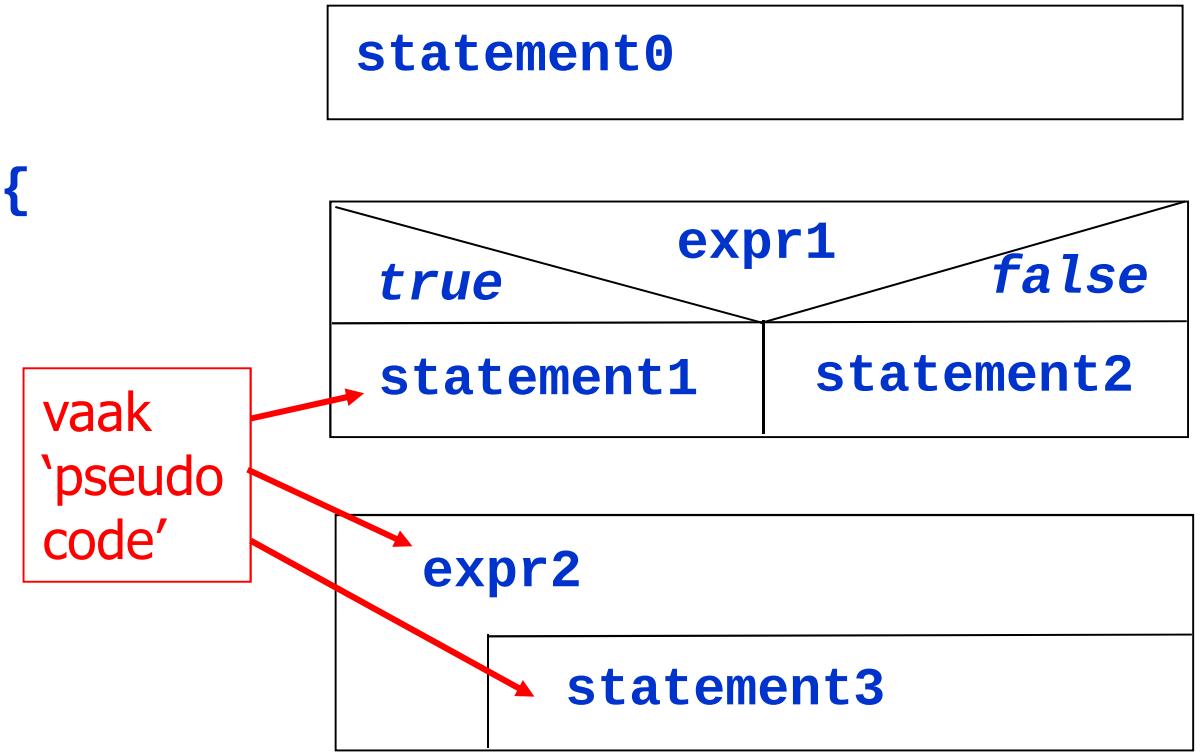
Nassi-Schneiderman Diagrammen

- Structuur van het programma overzichtelijk in een diagram
- Grotendeels taalonafhankelijk
- Handig voor het ontwikkelen van programma's/algoritmen

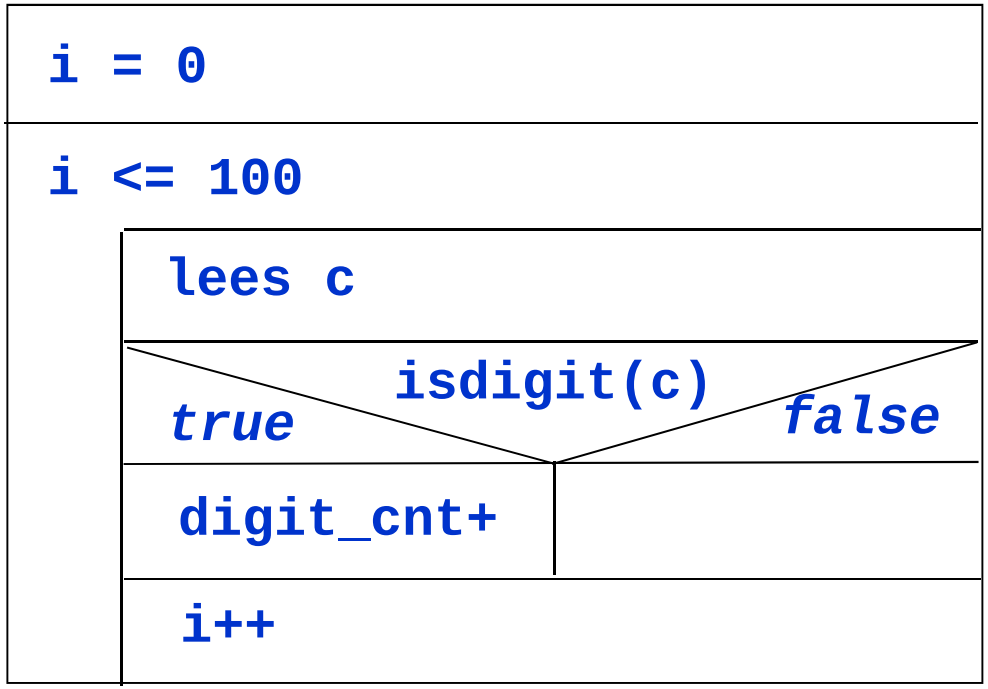
```
statement0


if (expr1) then {
   statement1
}
else {
   statement2
}

while (expr2) {
   statement3
}
```



Nassi-Shneiderman Voorbeeld




```
i = 0;

while (i <= 100) {

    scanf ("%c", &c);

    if (isdigit (c)) {
       digit_cnt++;
    }

    i++;
}
```

Functies

Inleiding functies

- Een functie omschrijft een stuk code welke met één statement (functie aanroep) kan worden uitgevoerd.
- Waarom functies ?
  - Hergebruik van code
  - Abstractie leidt tot overzicht
- Het opdelen van een programma in functies is de kern van effectief programmeren !

Functie definitie en aanroep

Voorbeeld functie definitie:

```
/* compute n! */
int factorial (int n)
{
   int i, product = 1;
   for (i = 2; i <= n; ++i)
      product *= i;
   return product;
}
```

formele parameter

functie header

functie body

variabelen `n`, `i` en `product` zijn alleen bekend binnen functie

actuele parameter

Voorbeelden functie aanroepen:

```
v = factorial (7);
w = 1000 + factorial (10);
x = factorial (a + 1);
```

Return statement voorbeelden

```
double absolute_value (double x)
{
   if (x >= 0.0)
      return x;
   else
      return –x;
}

float f (char a, char b, char c)
{
   int i;
   ...
   return i;   /* value returned will be */
}              /* converted to float */

void pr_ready () {
   printf ("ready.\n");
   return;                 /* nothing returned */
}          /* return statement can also be omitted */
```

Functie declaratie (functie prototype)

- Vertelt aan de compileer wat de typen van de formele parameters van de functie zijn en wat de return waarde van de functie is.

  Bijv. `float f (char a, char b, char c);`

- Daardoor kan de compiler fouten detecteren en eventueel type conversies doen.
  - Zodat bij een declaratie `int f(double x)` een aanroep met `f(1)` toch goed gaat.
- Voor de aanroep van elke functie moet óf een definitie óf een declaratie van deze functie staan !

  N.B. `void f(char c, int i);`
  is equivalent aan `void f(char, int);`

Het gebruik van meerdere files

- Een header file (pgm.h) – met o.a de functie declaraties - wordt opgenomen in verschillende C source (.c) files.
- Daardoor wijzigingen op slechts één plaats nodig.

pgm.h

```
#include <stdio.h>
#define  N  3
void fct(int k);
void wrt_info(char *s);
```

main.c

```
#include "pgm.h"

int main(void)
{
   fct (i);
   wrt_info (p);
   ...
}
```

fct.c

```
#include "pgm.h"

void fct(int k)
{
   ...
   wrt_info (m);
}
```

prn.c

```
#include "pgm.h"

void wrt_info(char *s)
{
   ...
}
```

gcc –o pgm main.c fct.c prn.c

Scope regels

- Een variabele is alleen bekend in het blok waarin hij gedefinieerd is (en in de bijbehorende binnenblokken).
- Een variabele in een binnenblok verhult een variabele met dezelfde naam in een buitenblok

```
{
   int a, b;
   ...
   {                  /* inner block 1 */
      float b, c;
      ...             /* int a is known, but not int b */
   }
   ...
   {                /* inner block 2 */
      float a;
      ...           /* int b is known, but not int a,
                       also c is not known */
   }
}
```

Static variabelen in een blok of functie

- Normaal wordt een variabele “aangemaakt” wanneer hij gedeclareerd wordt en verdwijnt hij weer na verlaten van blok of de functie.
- Wanneer static voor de declaratie van de variabele wordt gezet dan behoudt hij zijn waarde, ook wanneer blok of functie verlaten wordt.

```
float bereken_gemiddelde (float x)
{
   static float totaal = 0.0; /* initialisatie alleen */
   static int aantal = 0;     /* bij eerste aanroep */

   aantal++;
   totaal += x;

   return (totaal / aantal);
}
```

Globale variabelen

- Globale variabele worden gedeclareerd buiten een functie
- Globale variabelen kunnen gerefereerd worden op alle plaatsen in de file

```
file f.c:
   int a = 1;

   void f(void)
   {
      /* a is bekend hier */
   }
```

buiten functie geplaatst: globale variabele

- Ook vanuit andere files zijn ze dan te bereiken:

```
file g.c:
   extern int a;

   void g(void)
   {
      /* a uit f.c is ook bekend hier */
   }
```

hier mag geen initialisatie zijn

Static voor scope restrictie

- Wanneer static voor een functie declaratie/definitie wordt gezet, is deze functie alleen in die file bekend
- Hetzelfde geldt voor het gebruik van static in combinatie met een globale variabele

file f.c:

```
static int a;

static void f(void)
{
   ...
}
```

a en f() zijn alleen in file f.c bekend en de namen kunnen daarom in andere files worden hergebruikt.

- Manier om namen af te schermen.

Recursie

Een functie is recursief wanneer hij zichzelf aanroept

```
/* compute n+(n-1)+(n-2)+...+1
   via sum(n) = n + sum(n-1) (n > 1) */
int sum (int n)
{
   if (n <= 1)
      return n;
   else
      return (n + sum (n – 1));
}
```

voordeel: implementatie volgens algoritme/ denkwijze

nadeel: bij grote n veel functieaanroepen

Voorbeeld:

Functie aanroep	Waarde geretouneerd		
`sum(1)`	`1`		
`sum(2)`	`2 + sum(1)`	of `2 + 1`	of `3`
`sum(3)`	`3 + sum(2)`	of `3 + 2 + 1`	of `6`
`sum(4)`	`4 + sum(3)`	of `4 + 3 + 2 + 1`	of `10`

Recursie versus iteratie

Faculteit berekening: $n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots 3 \cdot 2 \cdot 1$
$= n \cdot (n-1)!$

```
int factorial (int n)   /* recursieve versie */
{
   if (n <= 1)
      return 1;
   else
      return (n * factorial (n - 1));
}

int factorial (int n)   /* iteratieve versie */
{
   int product = 1;

   for ( ; n > 1; --n)
      product *= n;
   return product;
}
```

Samenvatting

- De fundamentele datatypen
- Programmabesturing
- Nassi-Shneiderman diagrammen
- Functies