

HOOFDSTUK 1

BASISCONCEPTEN C

Inhoud

- **Situering programmeertaal C**
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- Elementaire I/O
- Functies
- Arrays en toepassingen

Situering programmeertaal C

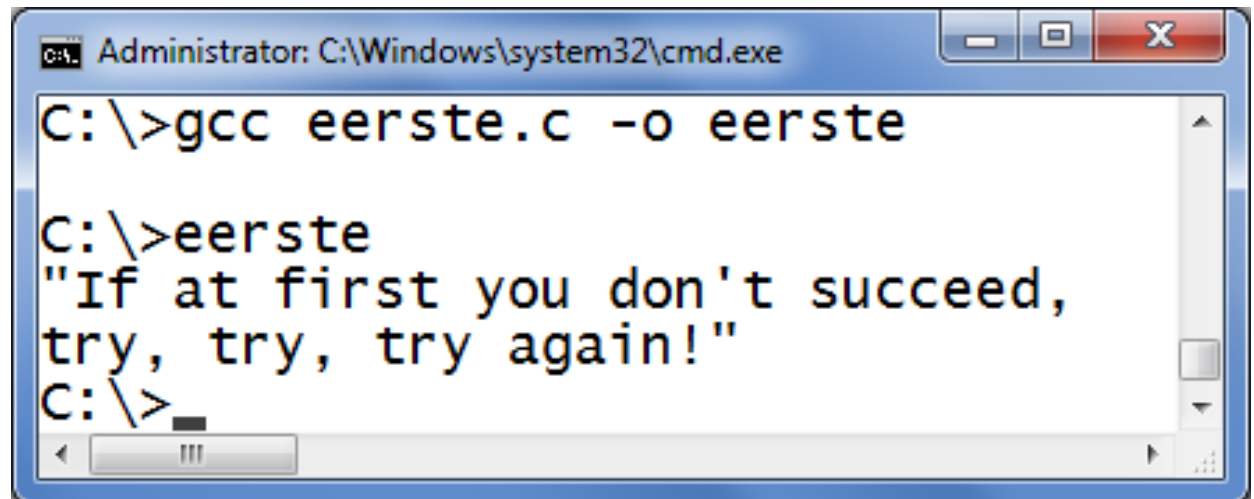
- C is een programmeertaal die zeer dicht bij de hardware staat en daaruit zijn snelheid haalt.
- Typische toepassingen:
 - kernels van Unix en Windows zijn in C geschreven
 - directe aansturing hardware (device drivers)
 - tijdkritische applicaties
 - ...
- Historiek C:
 - ANSI C: 1972 (oudste code)
 - C99: sinds 1999
 - C11: huidige standaard (sinds 2011)

Inhoud

- Situering programmeertaal C
- **Eerste C-programma**
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- Elementaire I/O
- Functies
- Arrays en toepassingen

Eerste C-programma

```
/*     eerste.c     */  
  
#include <stdio.h>  /* standard I/O */  
  
int main() {  
    printf("\nIf at first you don't succeed,\n");  
    printf("try, try, try again!\n");  
    return 0;  
}
```



The screenshot shows a Windows command prompt window titled "Administrator: C:\Windows\system32\cmd.exe". The prompt is at "C:\>". The user enters the command "gcc eerste.c -o eerste". The prompt changes to "C:\>eerste", and the program outputs the text "If at first you don't succeed, try, try, try again!". The prompt returns to "C:\>".

```
Administrator: C:\Windows\system32\cmd.exe  
C:\>gcc eerste.c -o eerste  
  
C:\>eerste  
If at first you don't succeed,  
try, try, try again!  
C:\>
```

Commentaar

- wordt genegeerd door de compiler
- 2 vormen:

`/* commentaar`

`(kan over meerdere regels) */`

`// rest van de regel is commentaar`

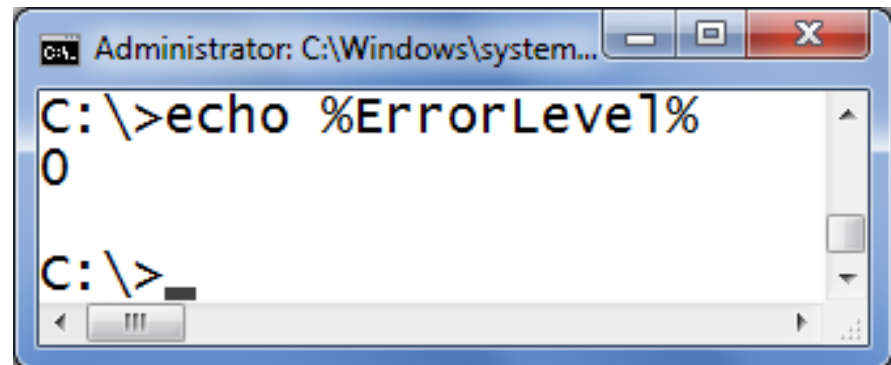
nesten is niet
toegelaten

sinds C99

return 0 (in main)

- einde succesvol programma
- return *andere waarde*: einde programma met probleem
- opvragen exit status:

`echo %ErrorLevel%`



```
Administrator: C:\Windows\system...
C: \>echo %ErrorLevel%
0
C: \>
```

Programmastructuur

```
/******  
naam bestand -- korte zin over inhoud  
***** */
```

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

Preprocessing directives

```
[declaraties variabelen/constanten]
```

```
[declaraties functies (a)]
```

```
[definities functies (b)]  
int main() {  
    ...  
}
```

```
[definities functies (a)]
```

C is een procedurele taal
(procedures staan centraal)
↔
object georiënteerde taal
(objecten staan centraal)

Preprocessing directives

- Instructies beginnend met symbool **#**
- Instructies bestemd voor de compiler
- Compiler voert deze instructies uit alvorens de code te compileren
- Staan meestal aan het begin van het programma (maar mogen ook elders staan)
- Voorbeeld:

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

```
#define PI 3.14159
```


Inhoud

- Situering programmeertaal C
- Eerste C-programma
- **Variabelen en fundamentele datatypes**
- Operatoren
- Controlestructuren
- Elementaire I/O
- Functies
- Arrays en toepassingen

Variabelen declareren

- Elke variabele moet vooraf worden **gedeclareerd**.

ANSI C: declaraties variabelen **vóór alle andere opdrachten**

Voorbeeld: `int a, b, c;`

- Naam van een variabele:
 - bevat willekeurig aantal letters, underscores of cijfers
 - begint met een letter of underscore (geen cijfer)
 - stijl:
 - gebruik geen hoofdletters
 - woordseparatie via underscore
 - gebruik zinvolle namen
 - naamlengte typisch tussen 5 en 15 karakters

Fundamentele datatypes in C

- Alle fundamentele datatypes zijn **numeriek** (ook char)
- Kunnen opgesplitst worden in 2 categorieën:
 - gehele types: zie verder
 - reële types: **float**, **double**, **long double**
- lengte van het type (en dus bereik) is machine-afhankelijk
 - operator **sizeof** geeft lengte in bytes van type of uitdrukking
 - syntax :

`sizeof(type)``sizeof(uitdrukking)`

$\Rightarrow \text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$

Gehele types

- 2 soorten: **signed** en **unsigned**
 - met en zonder teken \Rightarrow ander bereik!
 - default signed, met uitzondering van char
- Overzicht signed gehele types:

type	synoniemen
signed char	
short	short int, signed short, signed short int
int	signed, signed int
long	long int, signed long, signed long int
long long (sinds C99)	long long int, signed long long, signed long long int

- Overzicht unsigned gehele types:

type	synoniemen
_Bool (sinds C99)	bool (defined in <i>stdbool.h</i>)
unsigned char	
unsigned short	unsigned short int
unsigned int	unsigned
unsigned long	unsigned long int
unsigned long long (sinds C99)	unsigned long long int

- Booleans: in *stdbool.h* worden ook true en false gedefinieerd (zonder *stdbool.h*: gebruik 0 of 1)
- Afhankelijk van de compiler is char synoniem met signed char of unsigned char

Literals

- Gehele getallen:

- $\{+|- \}_{\text{opt}} \{\text{prefix}\}_{\text{opt}} \text{getal} \{\text{suffix}\}_{\text{opt}}$

- prefix:

decimaal:

octaal: 0

hexadecimaal: 0x | 0X

- suffix:

int

unsigned: u | U

long: l | L

unsigned long: ul | UL

long long: ll | LL

unsigned long long: ull | ULL

- Voorbeelden:

int i = 0x2ab;

long long ll = 077711;

- Reële getallen:

- $\{+|- \}_{\text{opt}} \text{getal}.\text{getal} \{\{e|E\} \{+|- \}_{\text{opt}} \text{getal}\}_{\text{opt}} \{\text{suffix}\}_{\text{opt}}$

- suffix:

double: float: **f|F** long double: **l|L**

- Voorbeelden:

float f = 123.456;

long double ld = 987E-71;

- Karakters: tussen enkele aanhalingstekens

b.v. 'a', '?', '\n', '\t', '\"', '\\', ...

- Strings: tussen dubbele aanhalingstekens

b.v. "dit is tekst"

LET OP : 'a' ≠ "a"

Constanten

- Syntax ANSI C:

#define *NAAM waarde*

Voorbeeld

```
#define PI 3.14159
```

- Syntax (sinds C99):

const *type NAAM = waarde;*

Voorbeeld

```
const double PI = 3.14159;
```


Impliciete conversies

1. Bij toekenningsoopdrachten

- impliciete (= automatische) conversies in C
- eventueel warning voor toekenningsoopdrachten die informatie kunnen verliezen (van wider naar narrower type)

- Voorbeelden:

```
float x = 2.8;    // double → float
```

```
int i = x;        // float → int
```

2. Bij rekenkundige opdrachten

- treden op bij binaire operatoren waarbij operandi van \neq type zijn
- widening van narrower type naar wider type
- Voorbeeld: `x += 2.5; // float → double, double → float`

Expliciete conversies (= casten)

- expliciet forceren van conversie

- syntax:

(type) uitdrukking

- Voorbeeld:

```
double d = 13.7;
```

```
int i = (int)d; //i krijgt waarde 13
```

```
double d = (double)i / 3;
```

- gebruik casts om aan te geven wat de bedoeling is!
(steun niet te veel op impliciete conversieregels)
- Voorbeeld: **conversies.c**

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- **Operatoren**
- Controlestructuren
- Elementaire I/O
- Functies
- Arrays en toepassingen

Operatoren

- rekenkundige operatoren: + - * / %
- assignatie-operatoren: = += -= *= /= %=
- auto-(in/de)crement operatoren: ++ --
- relationele operatoren: == != < > <= >=
- conditionele operator: ?:
- logische operatoren: ! && ||

Voorbeelden

```
a = (b=2) + (c=3); //a=5, b=2, c=3
```

```
//resultaat assignatie = waarde van de uitdrukking
```

```
a = b++; //a=2, b=3
```

Regels auto-(in/de)crement operatoren

- Gebruik geen auto-(in/de)crement op een variabele die meer dan 1 keer voorkomt in een uitdrukking.

Vb: `a = b/2 + 4*(1 + ++b);` *//Niet OK*

- Gebruik geen auto-(in/de)crement op een variabele die in meer dan 1 argument van een functie-oproep voorkomt.

Vb: `printf("%d %d", a, ++a);` *//Niet OK*

- Sterk afgeraden om tweemaal auto-(in/de)crement op dezelfde variabele te gebruiken!

Vb: `a = ++b + b++;` *//Niet OK*

De conditionele uitdrukking

- Syntax: *vwde* **?** *uitdr1* **:** *uitdr2*
- Voorbeeld 1:

```
a = b > 0 ? 1 : -1;
```

is equivalent aan:

```
if (b > 0)
    a = 1;
else
    a = -1;
```

Let op:

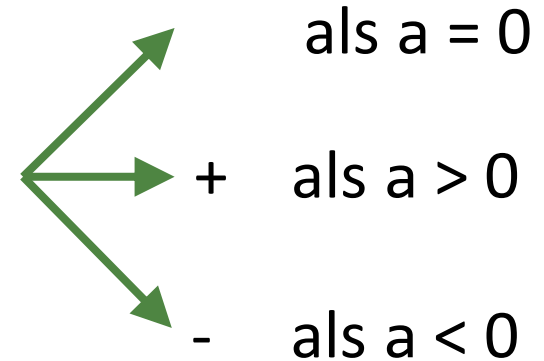
~~b > 0 ? a = 1 : a = -1;~~

- Voorbeeld 2:

gegeven: geheel getal a

gevraagd:

print het teken van a op het scherm



oplossing:

```
printf("Het teken van %d is: %c", a,  
      a>0 ? '+' : (a<0 ? '-' : ' '));
```

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- **Controlestructuren**
- Elementaire I/O
- Functies
- Arrays en toepassingen

Controlestructuren

- Herhalingsstructuur
 - while
 - do ... while
 - for
- Selectiestructuur
 - if – else
 - [switch]

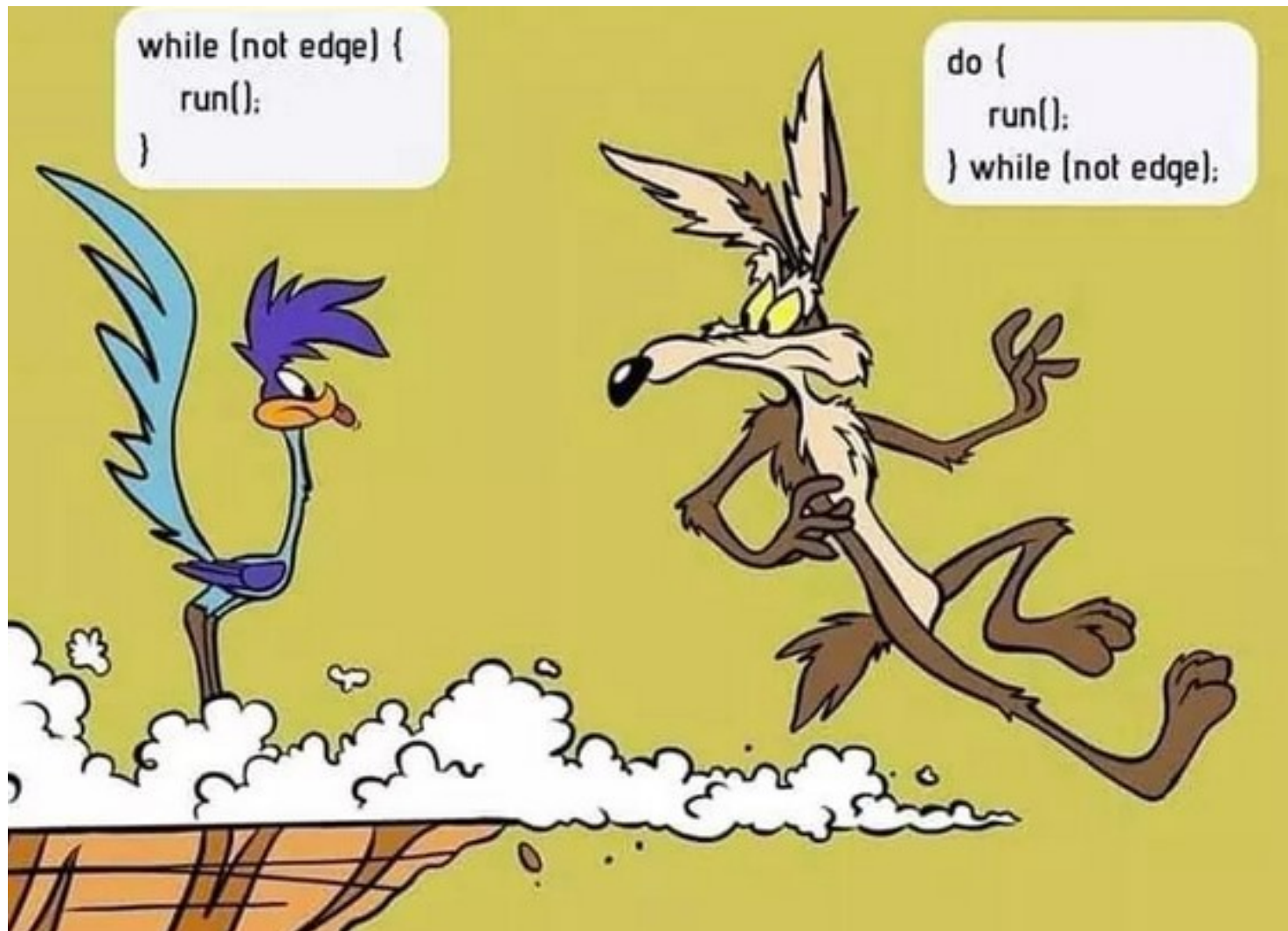
while

zolang deze voorwaarde voldaan is, wordt er herhaald

```
while (voorwaarde) {  
    opdracht(en) —→ wat moet er worden herhaald?  
}
```

Opmerking: { , } mogen weg indien while-lus maar 1 opdracht bevat

do while



for

```
for (initialisatie ; voorwaarde ; aanpassen ) {  
    opdracht(en)  
}
```

initialisatie: declaratie en initialisatie van de teller (= type int)

voorwaarde: voorwaarde waaraan de teller moet voldoen opdat de **opdracht(en)** herhaald zouden worden

aanpassen: opdracht waarmee de teller aangepast wordt

Opmerking: {, } mogen weg indien for-lus maar 1 opdracht bevat

if [-else]

```
if (voorwaarde) {  
    opdracht(en)  
}  
else {  
    opdracht(en)  
}
```

indien **voorwaarde** voldaan is, wordt **opdracht(en)** één keer uitgevoerd,
anders wordt (indien else-gedeelte) **opdracht(en)** één keer uitgevoerd

dit else-gedeelte is optioneel

Opmerking: {,} mogen weg indien maar één opdracht

Vernestelde if

```
if (...) {  
    if (...) {  
        ...  
    }  
    else {  
        ...  
    }  
}  
else {  
    ...  
}
```

```
if (...) {  
    ...  
}  
else if (...) {  
    ...  
}  
else if (...) {  
    ...  
}  
else {  
    ...  
}
```

Verschillen controlestructuren C en Java

1. Java: conditionele expressie moet van het type boolean zijn
C: conditionele expressie kan elk type zijn (vaak int)

Voorbeeld

```
int i = ... ;  
if (i) {  
    ... // uitgevoerd als i verschillend is van 0  
} else {  
    ... // uitgevoerd als i gelijk is aan 0  
}
```

2. In C kan je geen variabelen declareren in de initialisatie-sectie van een for-lus (kan wel in C++ en Java).

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- **Elementaire I/O**
- Functies
- Arrays en toepassingen

uitvoer: printf (in <stdio.h>)

- syntax:

`printf(formaat_string, argument1, argument2, ...)`

- formaat in formaatstring voor elk argument: `%[-][w][.p]ck`

- - : linkse alignatie
- w (width): minimale breedte
- p (precisie): maximaal aantal karakters (string) of aantal cijfers na de komma (reëel getal)
- ck (conversiekarakter):

c : karakter	[1]d : (long) decimaal geheel getal
% : teken '%'	[1]o : (long) octaal geheel getal
s : string	[1]x : (long) hexadecimaal geheel getal
[1]f : reëel getal	e : reëel getal in wet. notatie

printf: voorbeeld

```
/* printf.c */  
#include <stdio.h>  
  
int main() {  
    double g1 = 8.7468; int g2 = 456; char c = '9';  
    printf("grootte double = %d bytes\n", sizeof(double));  
    printf("g1 = %.3f of %e\n", g1, g1);  
    printf("g1 = Rounded: %.0lf\n");  
    printf("g1 = Truncated: %d\n", (int)d);  
    printf("g2 = %d of %o of %x\n", g2, g2, g2);  
    printf("ASCII-code van %c = %d\n", c, c);  
    printf(":%-15.10s:\n", "Hello, world!");  
    return 0;  
}
```

invoer: scanf (in <stdio.h>)

- syntax:

```
int scanf(formaat_string, argument1, argument2, ...)
```

- formaat in formaatstring voor elk argument: %ck
ck (conversiekarakter):

f : float	lf : double	Lf : long double
d : int	ld : long	lld : long long
c : char	hd : short	s : string

- **argumenten zijn steeds adressen!!!** Dus: **&var**
- gedrag: negeert witruimte, behalve bij char
- resultaat van scanf = aantal succesvol gelezen variabelen

scanf: voorbeeld

```
/* scanf.c */
#include <stdio.h>

int main() {
    int x; double sum = 0, v;
    printf("Enter an integer: ");
    if (scanf("%d", &x) == 0)
        printf("Error: not an integer\n");
    else printf("Read in %d\n", x);
    while (scanf("%lf", &v))
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

getchar / putchar (in <stdio.h>)

`int getchar()`

`int putchar(int)`

- doel: lezen/schrijven van 1 karakter
- getchar geeft EOF als einde input bereikt is (Ctrl-Z)
- Voorbeeld: **getchar.c**

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

int main() {
    int c;
    while ((c = getchar()) != EOF)
        putchar(toupper(c));
    return 0;
}
```

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- Elementaire I/O
- **Functies**
- Arrays en toepassingen

Funcities: typisch C-programma

```
/* functies1.c */                                //commentaar
```

```
#include <stdio.h>                                //preprocessing directives
```

```
const double PI = 3.1416; //declaratie constanten
```

```
void einde();                                     //declaratie functies
```

```
int main() {  
    double pi_2 = PI*PI;  
    printf("pi kwadraat = %lf\n",pi_2);  
    einde();  
    return 0;  
}
```

```
void einde() {                                     //definitie functies  
    printf("Einde programma ...\n");  
}
```

Definities functies

```
return-type function-name([parameter declarations]) {  
    declarations  
    statements  
}
```

- **return** statement zorgt voor het teruggeven van het resultaat
 - Syntax: `return expressie;`
 - niet bij void functies (= **procedures**)
 - meer dan 1 return statement toegelaten!
- **logische functie** = functie die iets controleert = functie met return-type `bool` (of `int` met 2 mogelijke waarden)

Declaraties functies

- Functie moet gedeclareerd worden **VOOR** gebruik
- 2 oplossingen:
 - definieer alle functies VOOR ze gebruikt worden (niet altijd mogelijk, bv. bij indirecte recursie!)
 - gebruik een functie-prototype:

```
return-type function-name([prototype-lijst]);
```

= opsomming van types van de formele parameters
(variabelen mogen weggelaten worden)

- **Opmerking:** GEEN function overloading en default parameters in C

voorbeeld

```
/* functies2.c */  
#include <stdio.h>  
  
void print_lijn();  
void print_ascii_tabel(char, char);  
int my_pow(int, int);  
  
int main(void) {  
    print_ascii_tabel('A', 'z');  
    print_lijn();  
    printf("2 tot de 6de macht = %d", my_pow(2,6));  
    return 0;  
}
```

```

void print_lijn() {
    int i;
    putchar('\n');
    for (i=0 ; i<80 ; i++) putchar('-');
    putchar('\n');
}

void print_ascii_tabel(char s, char e) {
    char c;
    for (c=s ; c<=e ; c++) printf("%c : %d\t", c, c);
}

int my_pow(int base, int n) {
    int p = 1, i;
    for (i=1 ; i<=n ; i++) p *= base;
    return p;
}

```

Opmerking: waarom is $x*x*x$ NIET gelijk aan $\text{pow}(x, 3)$ (uit `<math.h>`) ?

`double pow(double, double)`

- $x*x*x$: amper 2 bewerkingen nodig
 $\text{pow}(x, 3)$: functieaanroep en reeksontwikkeling
 \Rightarrow benaderende berekening (stopt als 'verbeterde' uitkomst niet meer verschilt van vorige benadering)
 \Rightarrow te veel bewerkingen + kans op afrondingsfouten
- **pow is bedoeld voor niet-gehele exponenten**
- Merk op:
 - als t een negatief geheel getal is, dan is $x^{-t} = 1/x^t$
 - $x^t = x * x^{t-1}$

Oproepen van functies: call by value

- Effect functie-oproep:
 1. argumenten worden geëvalueerd
 2. waarde van elk argument wordt toegekend aan
 formele parameter van functiedefinitie
 3. controle wordt overgedragen aan functie
 4. resultaatwaarde (return value) vervangt functie-oproep
 (evt. na conversie) in de oproepende context
- **Call by value**: waarde wordt doorgegeven
 - ⇒ indien het **niet** om een adres gaat, **kan een functie-oproep nooit de waarde van een variabele uit de oproepende context wijzigen!!!**

voorbeeld

```
/* functies3.c */
#include <stdio.h>

void bepaal_tegengestelde(int);

int main() {
    int i = -5;
    bepaal_tegengestelde(i);
    printf("i = %d\n", i);
    return 0;
}

void bepaal_tegengestelde(int i) {
    i *= -1;
    printf("i = %d\n", i);
}
```

Recursieve functies

Een functie heet (direct) **recursief** als hij zichzelf aanroept.

Een functie heet **indirect recursief** als de aanroep van zichzelf via een of meerdere andere functies gaat.

Voorbeeld: berekening faculteit

niet recursieve berekening:

$$n! = 1 * 2 * 3 * \dots * (n-1) * n$$

```
int fac(int n) {  
    int res = 1, i;  
    for (i=2 ; i<=n ; i++)  
        res *= i;  
    return res;  
}
```

recursieve berekening:

$$\begin{cases} n! = 1 & \text{als } n = 0 \text{ of } 1 \\ n! = (n-1)! * n & \text{als } n > 1 \end{cases}$$

```
int fac(int n) {  
    if (n<=1)  
        return 1;  
    else  
        return fac(n-1)*n;  
}
```


Recursie versus iteratie?

Bij iedere recursieve aanroep moet de nodige administratie verricht worden m.b.t. bijvoorbeeld de lokale variabelen en parameters.

Daarom heeft **recursie meestal een overhead t.o.v. iteratie**.

Het gebruik van recursie versus iteratie is meestal een afweging van schoonheid en duidelijkheid t.o.v. snelheid.

Voorbeeld faculteit: de snelheid van de rechtstreekse berekening en de recursieve zijn niet merkbaar verschillend. Enkel als de stack (= geheugenruimte die bijhoudt waar het programma gebleven was met de recursie) vol begint te raken, zou je vertraging kunnen zien. Maar tegen dan zijn we al lang de limiet voor de overflow gepasseerd...

Oefening 1

Schrijf een functie `int fibonacci(int x)` die het Fibonacci-getal met volgnummer `x` berekent. De Fibonaccigetallen, startend bij volgnummer 1, zijn 1 1 2 3 5 8 13 21 34 55 ... waarbij elk getal de som is van de twee vorige. (Behalve de twee eerste, die zijn gewoon 1.) Gebruik geen recursie.

Schrijf een functie `int fibonacci_rec(int x)` die hetzelfde doet, maar recursie gebruikt.

OPMERKING:

De recursieve versie gaat veel trager dan de niet-recursieve versie. Waarom?

Structuur recursieve algoritmen

Een recursieve oplossing bevat altijd:

- Een of meer **basisgevallen** waarvoor een oplossing geformuleerd wordt (geen recursieve oproep).
- Een of meer **recursieve gevallen**, waarvan de oplossing wordt herleid tot een eenvoudigere versie van hetzelfde probleem (recursieve oproep(en)).

Oefening 2: wat is de uitvoer?

```
void proc(int n) {  
    if (n == 0)  
        printf("Start\t");  
    else {  
        printf("%d\t",n);  
        proc(n-1);  
    }  
}
```

```
int main() {  
    proc(3);  
    return 0;  
}
```

```
void proc(int n) {  
    if (n == 0)  
        printf("Start\t");  
    else {  
        proc(n-1);  
        printf("%d\t",n);  
    }  
}
```

```
int main() {  
    proc(3);  
    return 0;  
}
```

Oefening 3: wat is de uitvoer?

```
void proc(int a, int b) {  
    if (a <= b) printf("Stop %d %d\n", a, b);  
    else {  
        proc(a-1,b);  
        printf("a=%d\tb=%d\n", a, b);  
        proc(a,++b);  
    }  
}  
  
int main() {  
    proc(5,3);  
    return 0;  
}
```

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- Elementaire I/O
- Functies
- **Arrays en toepassingen**

Wat zijn arrays?

- één naam (variabele) die meerdere geheugenplaatsen of elementen bevat.

⇒ een array is een samengesteld type

- Elk element
 - is van **hetzelfde type** (een array is homogeen)
 - wordt aangeduid d.m.v. een **index** (plaats, nummer)

Declaratie van een array

- Syntax:

type naam[grootte_array];

Let op: *grootte_array* moet een getal **constante** zijn!!

- Voorbeeld

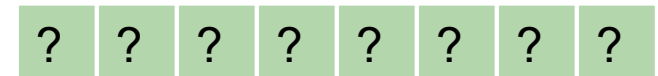
```
int getallen[8];
```

of

```
#define MAX 8
```

```
int getallen[MAX];
```

getallen



index 0 1 2 3 4 5 6 7

Gebruik van de array

- Syntax:

naam[*index*] met $0 \leq index < grootte_array$

- kan overal gebruikt worden waar gewone variabele van het zelfde type toegelaten is

- Voorbeeld

`getallen[2*index+1] = getallen[index]/9;`

- **Let op:** element uitschrijven met index buiten bereik van de array
⇒ GEEN foutmelding!

- **Merk op:** grootte array

`= sizeof(naam)/sizeof(type)`

- Voorbeeld: **arrays1.c**

Initialisatie bij declaratie van arrays

```
int t[6] = {1, 2, 3, 4, 5, 6};
```

⇒ **deze opdracht niet opsplitsen!!**

```
double neerslag[12] = {20.7, 23.0, 99.0, 77.4};
```

⇒ elementen 4 ... 11 geïnitieerd op 0

```
int test[4] = {};
```

⇒ initializer list moet minstens 1 waarde bevatten

```
int a[] = {1, 2, 3, 4};
```

⇒ indien bereik niet opgegeven: compiler vult zelf
aantal elementen in

Arrays als functieparameters

- Syntax:
 - Formele parameter: *type naam[]*
 - Prototype parameter: *type []*
- Voorbeeld (**arrays2.c**)

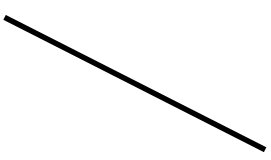
```
void verdubbel(int[], int);  
int main() {  
    int t[] = {1,2,3,4,5};  
    verdubbel(t, sizeof(t)/sizeof(t[0])); ...  
}  
void verdubbel(int t[], int n) {  
    int i;  
    for(i=0 ; i<n ; i++) t[i]*=2;  
}
```

geen [] toevoegen!!!

- Opmerking 1: de **grootte van een array moet steeds expliciet meegegeven worden** aan de functie, want:
 1. er bestaat geen length functie of attribuut (zoals in Java) om dit op te vragen
 2. `sizeof(param)/sizeof(type)` geeft niet de correcte grootte

Voorbeeld

```
int grootte(int t[]) {return sizeof(t)/sizeof(int);}
int main() {
    int a[] = {1,2,3,4,5};
    printf("grootte = %d %d\n", sizeof(a)/sizeof(int),
        grootte(a));
    return 0;
}
```



grootte = 5 2

- Opmerking 2: het resultaat van een functie kan géén array zijn, wel een pointer (zie Hoofdstuk 2)
~~`int[] maak_array(int n);`~~
- Opmerking 3: het **adres** van de array wordt meegegeven als argument
⇒ de functie kan de inhoud van de array wijzigen!
⇒ om dit te verhinderen: voeg **const** toe aan formele en prototype parameter

Voorbeeld (**arrays3.c**)

```
void kopieer(const int[], int[], int);  
void kopieer(const int s[], int d[], int n) {  
    int i;  
    for(i=0 ; i<n ; i++)  
        d[i] = s[i];  
}
```

Toepassingen op arrays

Toepassing 1: Evaluatie veelterm volgens de regel van Horner

Gegeven

veelterm van graad n :

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0 \quad (a_i = \text{reëel getal})$$

Gevraagd

Schrijf de functie `eval(a,n,x)` die een gegeven veelterm van *graad* n evalueert in het gegeven reëel getal x (a = array met $n+1$ coëfficiënten, met a_i op index i)

Mogelijke oplossing:

```
double eval(const double a[], int n, double x) {  
    int i;  
    double res = a[0];  
    double macht_x = 1;  
    for (i=1 ; i<=n ; i++) {  
        macht_x *= x;  
        res += a[i]*macht_x;  
    }  
    return res;  
}
```

Betere oplossing: regel van Horner

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_2 x^2 + a_1 x + a_0 \equiv \\ ((\dots((a_n x + a_{n-1})x + a_{n-2})x + \dots + a_2)x + a_1)x + a_0$$

$$\text{Bv: } 5x^3 - 6x^2 + 7x - 8 \equiv ((5x - 6)x + 7)x - 8$$

```
double eval(const double a[], int n, double x) {  
    int i;  
    double res = a[n];  
    for (i=n-1 ; i>=0 ; i--)  
        res = res*x + a[i];  
    return res;  
}
```


Toepassing 2: lineair zoeken in een ongesorteerde array

Gegeven

een ongesorteerde array t die n (>0) gehele getallen bevat

Gevraagd

Schrijf een functie `index(t,n,g)` die de index van het gegeven geheel getal g in de gegeven array t teruggeeft of -1 indien g niet voorkomt in de array.

Oplossing: lineair zoeken

- De elementen worden 1 voor 1 overlopen met een lus en telkens vergeleken met de gezochte waarde.
- Eens de waarde gevonden, heeft het geen zin de array nog verder te doorzoeken.

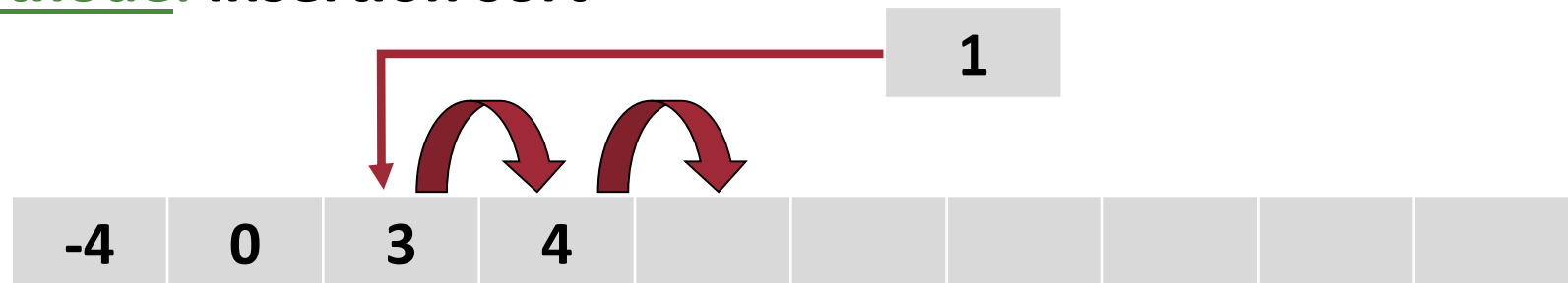
```
int index(const int t[], int n, int g) {  
    int i = 0;  
    while (i < n && t[i] != g)  
        i++;  
    return i < n ? i : -1;  
}  
  
int i;  
for (i = 0 ; i < n ; i++)  
    if (t[i] == g)  
        return i;  
return -1;
```

Toepassing 3: array opvullen in volgorde

Gevraagd

lees 10 gehele getallen in en sla ze **meteen** in volgorde op in een array

Methode: insertion sort



- Lees een nieuw getal in
- Vergelijk dit getal met het laatste element in de tabel en schuif dit element eventueel op
- ...
- Voeg het nieuwe getal toe op de juiste plaats

Oplossing (opvullen.c)

```
#define MAX 10

int main() {
    int t[MAX]; int getal, index, i;
    for (i=0 ; i<MAX ; i++) {
        scanf("%d", &getal); index = i-1;
        while (index>=0 && getal<t[index]) {
            t[index+1] = t[index]; index--;
        }
        t[index+1] = getal;
    }
    ...
}
```

Toepassing 4: frequentietabellen

- Gebruik:
 - Tellen hoeveel keer een bepaalde waarde voorkomt
 - Alle mogelijke waarden zijn eindig en welbepaald
- Voorbeeld

Gegeven een aantal examenresultaten (gehele getallen uit het interval $[0,20]$).

Geef een overzicht van het aantal keer dat elke score voorkomt.

 - Nood aan 21 tellers: elke teller is element van frequentietabel
 - Oplossing: **frequentietabel.c**

Toepassing 5: aanwezigheidstabellen

- Voorbeeld:

Geef een overzicht van de scores die **niet** voorkomen.

➤ Oplossing: **aanwezigheidstabel.c**

- Aanwezigheidstabel

- Is een variant van de frequentietabel
- Bepalen of waarden uit een reeks eindige en welbepaalde waarden **voorkomen** of niet
- Bijhouden per waarde: aanwezig of niet (tellen is niet nodig)
- Gebruik eventueel **bool** i.p.v. `int`

Oefeningen

1. Genereer 100 getallen uit het interval $[1,6]$. Geef daarna een grafisch overzicht van het aantal keer elke waarde (aantal ogen) gegooid werd, bijvoorbeeld:

1: *****

2: *****

...

6: *****

2. Laat de computer met een dobbelsteen gooien (schrijf telkens de waarde uit) en stop indien een aantal ogen gegooid wordt dat reeds gegooid werd, bijvoorbeeld:

Aantal geworpen ogen: 1 3 6 5 6

Inhoud

- Situering programmeertaal C
- Eerste C-programma
- Variabelen en fundamentele datatypes
- Operatoren
- Controlestructuren
- Elementaire I/O
- Functies
- Arrays en toepassingen