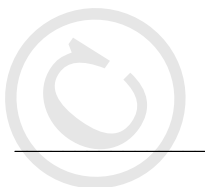


Student-notities
De programmeertaal C
Voorbeeld-hoofdstuk

02. Functies

 **at computing**
The Linux/UNIXperts

Nijmegen



Student-notities

- De student-notities in dit voorbeeld-hoofdstuk zijn fragmenten uit het dictaat dat bij deze cursus wordt meegeleverd. □

Een functie kan worden aangeroepen met argumenten, ook wel parameters genoemd.

Je kunt bijvoorbeeld een functie schrijven die de cursor positioneert op een bepaalde positie op het scherm. Deze functie, genaamd `cursor_to()`, krijgt twee parameters namelijk de rij en de kolom waar de cursor naar toe verplaatst moet worden:

```
cursor_to(int r, int k)
{
    rij = r;
    kolom = k;

    move(r, k);
}

main()
{
    int mijnrij = 7;

    cursor_to(mijnrij, 12);
}
```

In de aanroep zien we twee argumenten: `mijnrij` en `12`, beide van type integer; de eerste is een variabele, de andere een constante.

Aan de functiedefinitie-kant staat:

```
cursor_to(int r, int k)
{ /* begin-accolade */
```

Het aantal argumenten dat wordt doorgegeven aan de functie moet overeenkomen met datgene wat de functie verwacht, maar ook het type moet overeenstemmen. Aan de twee eisen wordt voldaan in het voorbeeld: er worden twee argumenten verwacht, en er worden er twee doorgegeven, dat klopt dus. De typen van de variabelen `r` en `k` zijn `int`, de argumenten `mijnrij` en `12` ook.



Argumenten doorgeven

Meegeven van argumenten aan functie

```
functie(arg1, arg2, ...);
```

Voorbeelden:

```
printwortel(9.0);  
setcursor(i, j);  
drukaf("Welkom programmeur!");
```

Opvangen van argumenten in functie

```
func(type arg1, type arg2)  
{  
    declaraties van lokale variabelen  
  
    statements  
}
```

Figuur 1.

Student-notities

Een functie wordt dus als volgt opgebouwd:

```
functienaam(type arg1, type arg2, ....., type argN)
{
    /* ← hier komt de open-accolade */

    /* hier komen de definities voor de
     * automatische variabelen
     */
    type au1;
    type au2;
    .....

    /* nu volgen de statements */
    .....

}
/* ← tot slot komt de sluit-accolade */
```

De namen van de argumenten mag je zelf kiezen, ze hoeven niet overeen te stemmen met de namen van de argumenten in de functie-aanroep. Dat kan ook moeilijk anders, want als de namen wel identiek hadden moeten zijn, konden we nooit een constante als 12 doorgeven.

De scope van de argumenten r en k is dezelfde als die van automatische variabelen: lokaal voor de aangeroepen functie. Het belangrijke verschil met de automatische variabelen is dat de argument-variabelen waarden bevatten die de aanroeper hen geeft, terwijl de automatische variabelen hun waarden moeten krijgen in de functie zelf.

Voorbeelden

Opvangen van argumenten in functie

```
printwortel(double getal)
{
    ...
}

setcursor(int a, int b)
{
    ...
}

drukaf(char str[])
{
    ...
}
```

Figuur 2.

Student-notities

In C kunnen functies een waarde retourneren. Indien die waarde een `int` is, is het niet nodig om extra werk te verrichten: `int` is namelijk het default returntype. Zodra er echter iets anders wordt geretourneerd — bijvoorbeeld een `long int` of een `float` — dan moeten er binnen een programma twee aanpassingen plaatsvinden:

1. De functiedefinitie moet worden aangepast.
Vóór de naam van de functie dient het type te staan dat de functie retourneert. Staat er niets dan is er sprake van de default: `int`.
2. Voordat de functie mag worden gebruikt, dient die functie eerst gedeclareerd te worden met het juiste returntype.

Een voorbeeld:

```

/* macht(): neem n-de macht van getal f
           en return het resultaat
*/
double macht(double f, int n)
{
    double result = 1.0;

    if (n < 0){
        while (n++)
            result /= f;
    } else {
        while (n--)
            result *= f;
    }

    return result;
}

int main()
{
    double r;

    r = macht(3.0, 5);

    printf("%f\n", r);

    return 0;
}

```

Op de regel:

```

double macht(double f, int n)
{

```

staat vóór de functienaam het returntype van de functie.

Returnwaarde van functie

Functies kunnen ook een waarde teruggeven

```
k = telop(i, j);
```

In functie `telop` staat dan een `return` statement:

```
int telop(int a, int b)
{
    int result;
    result = a + b;
    return result;
}
```

wat geeft `telop` terug?

waarde die `telop` teruggeeft

Merk op:

- Geen returnwaarde (geen statement `return` of een `return` zonder expressie) betekent:
geen zinnige waarde (rommel)
- Een `char` waarde wordt eerst naar een `int` omgezet en dan pas teruggegeven
- Geef altijd op wat het returntype is van een functie

Figuur 3.

Student-notities

Er kan veel fout gaan als returnwaarden niet correct gebruikt worden. Houd daarom de volgende richtlijnen aan:

- In een functie met meerdere return-statements dienen alle return-statements de vorm `return expressie;` te hebben, of allemaal de vorm `return;`. Een mix van deze twee geeft problemen. Als een returnwaarde verwacht wordt maar niet gegeven, wordt er rommel geretourneerd. Dit verschijnsel treedt ook op in het volgende voorbeeld:

```
int telop(int i, int j)
{
    int resultaat;
    resultaat = i + j;
    return; /* ‡ fout */
}

main()
{
    int i;
    i = 0;
    i = telop(4, 5); /* † i bevat rommel */
}
```

De variabele `i` bevat na aanroep naar `telop()` rommel, omdat er niets wordt teruggegeven uit `telop()`.

- Het is toegestaan een returnwaarde te negeren. Bijvoorbeeld de returnwaarde van de functie `printf()` wordt bijna altijd genegeerd.
- Het returntype `int` is de default. Bij de functie `main()` is tot nu toe genegeerd dat ook deze een returnwaarde heeft (sterker nog: *moet* hebben; zonder returnwaarde geeft ze rommel terug). Een correct gebruik van `main()` is bijvoorbeeld:

```
int main()
{
    printf("Hello world\n");

    return 0;
}
```

De returnwaarde van `main()` kan worden gebruikt door degene die het programma gestart heeft; vaak is dat een commando interpreter. De returnwaarde van `main()` is als het ware de returnwaarde van het hele programma.

Returntype

Nog een voorbeeld:

```
double opp(int i, int j)
{
    double resultaat;

    resultaat = .....;
    return resultaat;
}

int oproeper()
{
    double res;

    res = opp(10, 15);
    return 42;
}
```

Bij aanroepen moet:

- de definitie van de functie gezien zijn, of
- een declaratie gezien zijn

Is er geen returntype aangegeven, dan wordt **int** aangenomen.

Dus: altijd vermelden welk type de functie teruggeeft

Figuur 4.

Student-notities

Indien in een C programma een functie wordt aangeroepen met argumenten die qua aantal of type niet overeenstemmen met datgene wat de aangeroepen functie verwacht, dan is er niet gedefinieerd wat het resultaat van de functie-aanroep is. Bijvoorbeeld:

```
int f(int a, double b)
{
    int i;
    i = a / b;
    return i;
}

int main()
{
    double c;

    c = f(2);          /* ‡ fout */
    return 0;
}
```

Het prototyping mechanisme uit C biedt de mogelijkheid controles door de compiler uit te laten voeren. Achter het gebruik van prototyping schuilt het idee dat voordat een functie aangeroepen wordt die functie eerst volledig gedeclareerd moet worden. In feite is dit een uitbreiding van de regel dat variabelen eerst gedeclareerd moeten worden voordat ze gebruikt mogen worden.

Het volledig declareren van een functie omvat niet alleen maar het returntype van een functie, maar ook de parameterlijst. De notatie van het prototyping mechanisme is vrij eenvoudig: in de declaratie wordt zoals gebruikelijk het returntype vermeld en tussen de functiehaakjes worden de juiste typen van de te ontvangen argumenten gezet.



Prototypes - 1

Declareren van een functie:

```
/* ergens: */  
double opp(int, int); ← prototype  
  
int oproeper()  
{  
    double res;  
  
    res = opp(10, 15);  
    return 42;  
}  
  
/* ergens anders: */  
double opp(int i, int j)  
{  
    double resultaat;  
  
    resultaat = .....;  
    return resultaat;  
}
```

Figuur 5.

Student-notities

De declaratie voor de functie `sqrt()` die een `double` retourneert en een `double` als argument verwacht ziet er zo uit:

```
double sqrt(double);
```

De functie `macht()` heeft het volgende prototype:

```
double macht(double, int);
```

En de functies `getchar()` en `putchar()` tenslotte:

```
int putchar(int);
int getchar(void);
```

De functie `getchar()` ontvangt geen parameters; dat wordt in het prototype opgegeven door middel van `void`.

Het is ook mogelijk om geen argumenten te declareren:

```
int getchar();          /* ← geen volledig prototype */
```

Maar dit is de klassiek-C declaratie stijl: er wordt niet gecontroleerd op argument aantal en type, ook niet door standaard C compilers die de oude declaratie stijl nog wel accepteren vanwege "backward compatibility". Er is al aangekondigd dat deze declaratie stijl in toekomstige C standaarden niet meer zal worden ondersteund. Voor alle functies uit de standaard C bibliotheek staat een prototype in de include-file die bij die functies hoort. Voor de prototypen van `putchar()`, `getchar()` en `printf()` moet je bijvoorbeeld `#include <stdio.h>` in je programma opnemen.

De definitie van de functie zelf ziet er net zo uit als het prototype:

```
returntype functienaam(type arg1, . . . . ., type argN)
{
```

De functiedefinitie van `macht()` ziet er dan zo uit:

```
double macht(double f, int n)
{
    . . . . .
}
```

En die voor `getchar()` zo:

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

Prototypes - 2

Informeren compiler over returntype, argumenttypes en aantal argumenten

Nut:


- compiler kan foutmeldingen geven bij verkeerd aantal of verkeerd type argumenten
- compiler kan zorgen voor eventuele conversies

Voorbeelden:

```
double sqrt(double);
```

```
int power(int base, int expon);
```

*naam argumenten in
prototypes facultatief*



Bij aanroep

```
res = sqrt(3);
```

wordt **int** 3 geconverteerd naar **double**

Figuur 6.

Student-notities

Indien een functie een prototype heeft, dan kan de compiler bij het gebruik van die functie controleren of het type van de argumenten klopt. Het is dan maar een klein stapje verder om de compiler een conversie uit te laten voeren naar het gewenste type indien het aangeboden type niet correct is.

Bijvoorbeeld bij de standaard C-functie `sqrt()`:

```
#include <math.h>          /* bevat: double sqrt(double); */

int main()
{
    double d;

    d = sqrt(9);
    printf("%f\n", d);
    return 0;
}
```

Aangezien de compiler weet dat `sqrt()` een `double` als argument verwacht maar er een integer wordt doorgegeven zal de compiler de integerwaarde 9 converteren naar 9.0 (type `double`). Het aantal conversies is beperkt tot die conversies die ook via toekenning mogelijk zijn. Als een bepaalde toekenning niet legaal is, dan werkt ook de conversie door prototyping niet:

```
double d;
d = "9.0";          /* ‡ fout */

sqrt("9.0");       /* ‡ fout */
```

Er is één plaats waar conversie door prototyping niet werkt en dat zijn functies die variabele aantallen argumenten verwachten, bijvoorbeeld bij `printf()`:

```
printf("%f\n", 9);          /* † fout */
printf("%f\n", 9.0);       /* goed */
```

De compiler heeft geen verstand van het werk dat functies doen en ook niet van de format-string van de functie `printf()`, dus wordt de waarde 9 gewoon als integer doorgegeven terwijl een `double` wordt verwacht.



Prototypes - 3

Als een functie geen returnwaarde heeft: **void**

Als een functie geen argumenten heeft: **void**

```
void cls(void);  
  
void func(void)  
{  
    ...  
    cls();  
}  
  
void cls(void)  
{  
    printf("...");  
}
```

`i = cls();` 🖱️ foutmelding bij vertalen
`cls(17);` 🖱️ foutmelding bij vertalen

- Gebruik *altijd* functie prototypes
- Argumentencontrole alleen mogelijk bij gebruik van prototypes

Figuur 7.

Student-notities

Als enkelvoudige integers, characters en float's worden doorgegeven als argument aan functies, worden zij niet zélf doorgegeven maar alleen hun waarde. Die waarde wordt in een lokale variabele van de functie geplaatst. Dit mechanisme heet *call by value*. Het gevolg hiervan is dat als een functie zijn argument wijzigt, in feite de kopie in de lokale variabele gewijzigd wordt:

```
void print_som(int, int);

int main()
{
    int a, b;

    a = 10;
    b = 11;

    print_som(a, b);
    /* a is nu nog steeds 10, geen 21 */
}

void print_som(int i, int j)
{
    i = i + j; /* lokale i wordt gewijzigd */

    print(i);
}
```

Het call-by-value mechanisme zorgt ervoor dat we in de functie `print_som()` geen kopie van het argument hoeven te maken als we het argument (lokaal) willen wijzigen; we werken immers al met een kopie. Soms is het juist wel de bedoeling dat een functie het originele argument wijzigt. Met call-by-value lukt dit nooit:

```
void verlaag(double);

int main()
{
    double btw;

    btw = 18.5;

    verlaag(btw);

    /* nu is btw nog steeds 18.5 */
}

void verlaag(double b)
{
    b = 6.0;
}
```

Call by value

Call by value

- Een **kopie** van de argumenten wordt aan de functie doorgegeven.
- Alle **enkelvoudige variabelen** worden *by value* doorgegeven.

Bijvoorbeeld:

```
void f(void)
{
    int i;
    i = 1;
    g(i);
    /* i is nog steeds 1 */
}
void g( int a )      a beïnvloedt variabele i niet
{
    a = 2;
    /* a is hier 2 */
}
```

Call by value:

Veranderingen aan een argument binnen een functie hebben *geen* effect op het argument waarmee de functie wordt aangeroepen.

Figuur 8.

Student-notities

Als een *array* wordt doorgegeven aan een functie wordt niet het hele array gekopieerd; dat zou een vrij dure operatie zijn als het een groot array betreft. In tegenstelling tot de enkelvoudige variabelen (die call-by-value worden doorgegeven) wordt er voor arrays een mechanisme gebruikt dat *call by reference* heet.

We spreken over call-by-reference als we niet de *waarde*, maar een verwijzing naar een variabele doorgeven. In feite geven we dus een referentie (of een "alias") door. Aan de functiedefinitie-kant moet uiteraard vermeld worden dat het hier een array betreft:

```
void drukaf(char []);

int main()
{
    char hello[12];

    hello[0] = 'H';
    hello[1] = 'e';
    ...
    hello[11] = '\0';

    drukaf(hello);
}

void drukaf(char str[])
{
    int index;

    index = 0;

    zolang str[index] != '\0'
doe
        putchar(str[index]);
        index = index + 1;
    klaar
}

```

Aan de functiedefinitie-kant zien we:

```
drukaf(char str[])
{

```

Wordt hier voldaan aan de eis dat aantal en typen van de argumenten overeen komen? Ten dele wel: het aantal argumenten klopt en ook het type: `char s[]` is een character array. Er ontbreekt aan de functiedefinitie-kant echter wel hoe groot het array is, het `[]`-paar is leeg. Het call-by-reference mechanisme geeft uitsluitend de verwijzing door van een array, niet de grootte! Die grootte varieert trouwens bij elke aanroep naar `drukaf()`. In bovenstaand voorbeeld wordt de lengte van het array afgeleid uit de inhoud van een element (laatste element bevat `\0`). Als het niet mogelijk is de lengte uit de element*waarde* af te leiden, wordt vaak een extra argument meegegeven dat de lengte aangeeft.

Call by reference

Call by reference

- Het **adres** van de argumenten wordt aan de functie doorgegeven.
- Alle **arrays** worden *by reference* doorgegeven.
Let wel: arrays! Niet: array-elementen.

Bijvoorbeeld:

```
void f(void)
{
    int rij[10];
    rij[5] = 1;
    g(rij);
    /* rij[5] is nu 2 */
}

void g( int r[] )    r beïnvloedt variabele rij
{
    r[5] = 2;
    /* r[5] is hier 2 */
}
```

Call by reference:

Veranderingen aan een argument binnen een functie hebben *wel* effect op het argument waarmee de functie wordt aangeroepen.

Figuur 9.

