

Einführung in die Programmiersprache C

Marcel Arndt

arndt@ins.uni-bonn.de

Institut für Numerische Simulation

Universität Bonn

Der Anfang

Ein einfaches Programm, das „Hello World!“ ausgibt:

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

int main()
{
    printf("Hello World!\n");
    return 0;
}
```

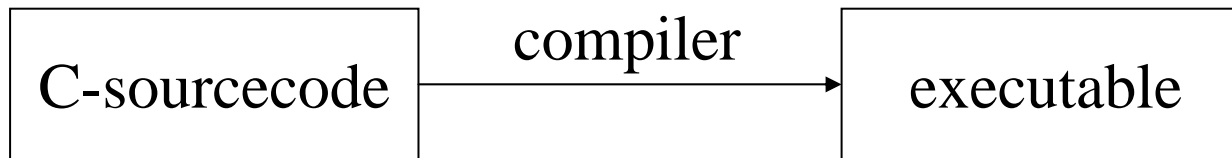
Sourcecode: helloworld.c

Compiler

Problem: C-code kann nicht direkt vom Computer ausgeführt werden, sondern nur Maschinencode.

Maschinencode ist aber nicht menschenlesbar/-schreibbar

Abhilfe: compiler generiert aus C-code den Maschinencode



Bedienung: unterschiedlich je nach verwendetem compiler.

Beispiel GNU GCC:

```
gcc helloworld.c -o helloworld
```

Kommentare

Kommentare werden vom compiler ignoriert. Zweck:

- Erläuterungen des Programmierers
- „Auskommentieren“ von normalen Programmteilen, die gerade nicht benötigt werden.

Beispiel: helloworld-kommentar.c

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

int main( )
{
    /* jetzt wird etwas ausgegeben */
    printf("Hello World!\n");
    return 0;
}
```

Variablen

Variablen müssen vor der Verwendung deklariert werden:

- zur Angabe des Variablentyps und
- zur Reservierung des notwendigen Speicherplatzes

Danach können sie verwendet werden.

Ausschnitt aus `variablen.c`:

```
int i;  
double f, g; } Deklaration  
  
i = 5;  
f = 3.14;  
g = i + f - 2; } Verwendung  
i = i + 1.8;
```

Variablen

Variablentypen:

- Integers: int, unsigned int, signed int, short int, long int
- Floating points: float, double, long double
- Sonstiges: char, pointer, ...

Operatoren:

- + - * /
- = (Zuweisung)
- == != Test auf Gleichheit/Ungleichheit
- += -= *= /= (Beispiel: `i+=1;` entspricht `i = i+1;`)
- % (Modulo-Operation)
- ++, -- (Beispiel: `i++` bzw. `++i` statt `i=i+1`)
- << >> (bit-shift nach links bzw. rechts, z.B. `j=i<<2;`)
- &&, || (logische Operatoren), &, |, ^ (bitweise logische Operatoren)

Ausgabe

Ausschnitt aus `ausgabe.c`:

```
int i;  
double f, g;  
  
i = 5;  
f = 3.14;  
g = i + f - 2;  
i = i + 1.8;  
  
printf("i hat den Wert %d\n", i);  
printf("und f ist %12.5f\n", f);  
printf("f = %12.5e und g = %12.5e\n", f, g);
```

Ausgabe (cont'd)

Prinzip von `printf`:

- auszugebende Variablen werden als zusätzliche Argumente angegeben:
`printf("i hat den Wert %d\n", i);`
- `%d` wird ersetzt durch den Wert der entsprechenden integer-Variable
`printf("i hat den Wert %d\n", i);`
- `%5d` wird ersetzt durch den Wert der entsprechenden integer-Variable, mit (mindestens) 5 Stellen
- `%f` wird ersetzt durch den Wert der entsprechenden float/double-Variable
- `%12.5f` wird ersetzt durch den Wert der entsprechenden float/double-Variable, mind. 12 Stellen insgesamt, davon 5 Nachkommastellen
- `%20.13e` wird ersetzt durch den Wert der entsprechenden float/double-Variable in Exponentialdarstellung, (mind.) 20 Stellen, davon 13 Nachkommastellen

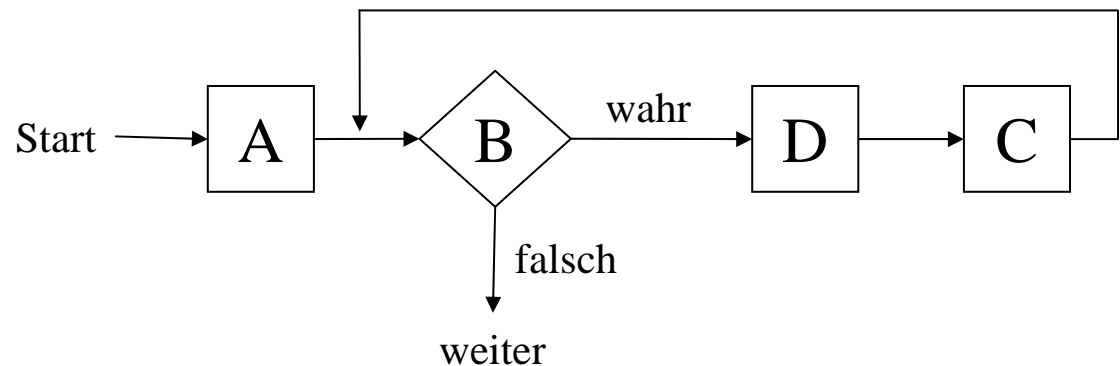
for-Schleifen

Beispiel for-schleife.c:

```
int i;  
for (i=0; i<10; i++)  
{  
    printf("%d\n", i);  
}
```

Flussdiagramm von

```
for (A; B; C)  
    D;
```



A: Initialisierung

B: Bedingung

C: z.B. Inkrement

D: auszuführender Block

if-Abfragen

Beispiel `if-abfrage.c`:

```
int i;  
for (i=0; i<10; i++)  
{  
    printf("%d\n", i);  
    if (i%3 == 0)  
        printf("ist durch 3 teilbar\n");  
}
```

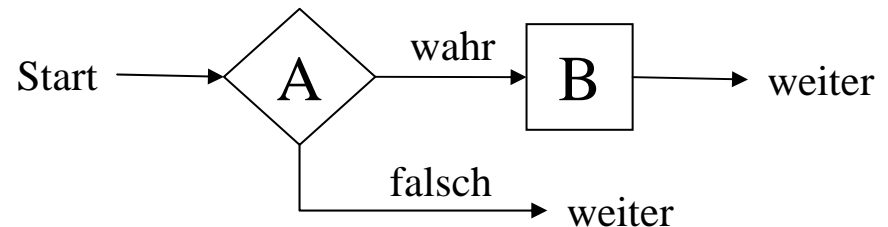
Flussdiagramm von

```
if (A)
```

```
    B;
```

A: Bedingung

B: wahr-Block



if-Abfragen

Beispiel `if-else-abfrage.c`:

```
int i;  
for (i=0; i<10; i++)  
{  
    printf("%d\n", i);  
    if (i%3 == 0)  
        printf("ist durch 3 teilbar\n");  
    else  
        printf("ist nicht durch 3 teilbar\n");  
}
```

Flussdiagramm von

`if (A)`

`B;`

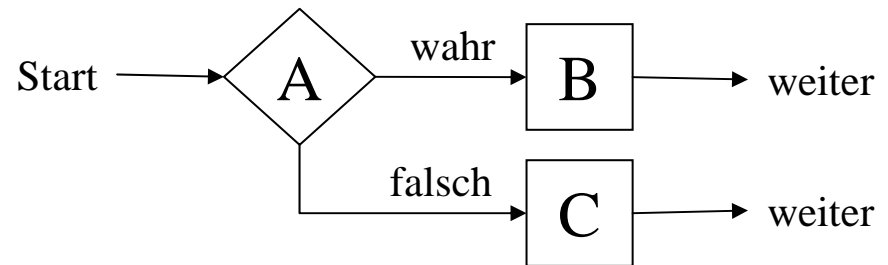
`else`

`C;`

A: Bedingung

B: wahr-Block

C: falsch-Block



Mathematische Funktionen

Wichtige mathematische Funktionen:

`exp`, `log`, `log2`, `log10`, `pow`, `sqrt`

`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, ...

Beispiele:

```
double e = exp(1);
```

```
double wurzel2 = sqrt(2);
```

```
double f = pow(2, 0.5);
```

Beispiel `tabelle.c`:

Tabellarische Ausgabe einiger Sinus- und Kosinuswerte.

Einlesen von Werten

Tastatureingabe lesen und Wert in Variable speichern:

Beispiel `einlesen.c`:

```
int i;
printf("i eingeben: ");
scanf("%d", &i);
printf("i ist %d\n", i);
```

&-Zeichen:
Adresse statt Wert
übergeben.
„pointer“

Beispiel `tabelle-variabel.c`:

Benutzer kann Zahl der Schritte selbst vorgeben.

Beachte		printf	scanf
unterschiedliche	int	%d	%d
Argumente:	float	%f, %e	%f, %e
	double	%f, %e	%lf, %le

Pointer

pointer = Adresse im Speicherbereich des Computers

bzw. Variable, deren Wert eine Adresse im Speicherbereich beschreibt

Deklaration: `int *pi;` pointer auf Variable vom Typ `int`

`double *pf;` pointer auf Variable vom Typ `double`

`void *p;` allgemeiner pointer (ohne Typ)

Nutzung:

Bestimmung der Adresse:

```
int i;
```

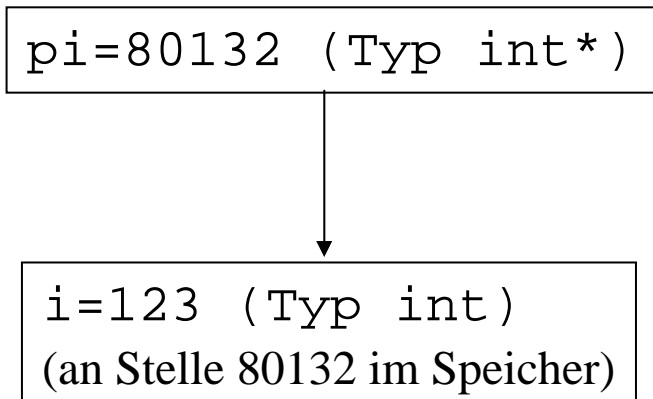
```
pi = &i; /* pi zeigt auf i */
```

Dereferenzierung:

```
*pi = 123; /* i ist 123 */
```

```
*pf = 3.14159; /* Vorsicht:
```

```
wurde pf initialisiert??? */
```



Unterprogramme / Funktionen

- Gliederung des Programms
- mehrfache Verwendung
- Rekursion

```
void f()  
{  
    printf("Funktion f\n");  
}
```

```
double g(int i)  
{  
    return 0.3*i;  
}
```

```
int main()  
{  
    double d;  
    f();  
    d = g(8);  
}
```

Beispiel: fakultaet.c

Parameterübergabe bei Funktionen

1) **return value** einer Funktion: `int f(); { return 123; }`

Ein Wert beliebigen Typs (int, double, ...) kann zurückgegeben werden.

Beispiel eines Funktionsaufrufs: `int i; i = f();`

Typ kann auch `void` sein, d.h. keine Rückgabe eines Wertes.

2) **Argumente der Funktion** werden in Klammern übergeben:

Deklaration der Funktion: `void f(int i, double f)`

Verwendung der Funktion: `int k=5;`
`f(k, 4.56);`

Beachte: Variablen werden als **Kopie** übergeben, nicht die Originalvariable selbst
Änderungen innerhalb der Funktion gehen daher verloren!

Beispiel: `parameter1.c`

Abhilfe, falls Änderung der Variable gewünscht: pointer auf Variable übergeben.

Beispiel: `parameter2.c`

Arrays

```
int a[10];
```

Damit werden 10 integer-Variablen deklariert:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
------	------	------	------	------	------	------	------	------	------

Verwendung:

```
a[0] = 3;    /* der erste Eintrag */
```

```
a[9] = 6;    /* der letzte Eintrag */
```

```
a[10] = 2;   /* Fehler: Bereichsüberschreitung */
```

Dies erlaubt die systematische Verwendung einer Vielzahl von Variablen.

U.a.: Index kann wieder eine integer-Variable sein!

Beispiel `array.c`:

Mehrdimensionale Arrays

```
double matrix[3][2];
```

Damit werden 6 double-Variablen deklariert:

```
matrix[0][0]
```

```
matrix[0][1]
```

```
matrix[1][0]
```

```
matrix[1][1]
```

```
matrix[2][0]
```

```
matrix[2][1]
```

Dynamischer Speicher

Problem: arrays wie bisher haben eine feste Größe

(genau: Größe muss zum Zeitpunkt der Kompilation bekannt sein)

Die Größe kann somit NICHT von der Benutzereingabe abhängen.

Abhilfe: Speicher dynamisch alloziieren:

```
int N;  
double *array;  
scanf( "%d" , &N) ;  
array = malloc(sizeof(double)*N) ;  
/* verwende nun array[0] ... array[N-1] */  
free(array) ;
```

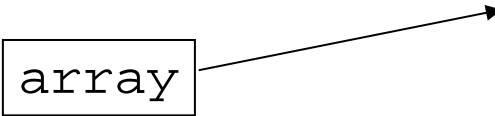
Dynamischer Speicher & Pointer

Die Anweisung `double *array;` deklariert einen pointer,
pointer = Adresse im Speicherbereich des Computers

Mit `array = malloc(sizeof(double)*N);` wird Speicher für das array reserviert.
array = 53000
sizeof(double) = 8

Der pointer array zeigt auf den Anfang
dieses Speicherbereichs

array



Adresse	Wert
53000	array[0]
53008	array[1]
53016	array[2]
...	...
53000+8*(N-1)	array[N-1]

Freigeben des Speichers:
`free(array);`

Beispiel: `dynarray.c`