
Einführung in die Programmierung für Physiker

Die Programmiersprache C – Basics an Hand von Beispielen

Marc Wagner

Institut für theoretische Physik
Johann Wolfgang Goethe-Universität Frankfurt am Main

WS 2019/20

Das minimale C-Programm

- **Programmcode** öffnen (Endung üblicher Weise `.c`):

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ xemacs minimal.c
```

- Programmcode schreiben:

```
1. int main(void)
2. {
3. }
```

- Jedes **C**-Programm muss eine **main-Funktion** besitzen.
- Diese **main**-Funktion wird bei Programmstart aufgerufen, d.h. der Computer führt die **Anweisungen** innerhalb von `{` und `}` der Reihe nach aus.
- Da zwischen `{` und `}` keine Anweisungen stehen, tut das Programm nichts; es endet unmittelbar nach dem Start.
- Die Bedeutung von **int** und **void** wird später erläutert (hier und in vielen anderen Beispielen ohne tiefe Bedeutung).
- Programmcode **kompilieren** (Programmcode in ein für den Computer ausführbares Programm umwandeln):

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 4
-rw-rw-r-- 1 mwagner mwagner 36 Okt 13 13:04 minimal.c
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o minimal minimal.c
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 16
-rwxrwxr-x 1 mwagner mwagner 8658 Okt 13 13:10 minimal
-rw-rw-r-- 1 mwagner mwagner 36 Okt 13 13:04 minimal.c
```

- **gcc** ist ein häufig verwendeter **C-Compiler**.
- **g++** ist ein häufig verwendeter **C++-Compiler**.
- Mit der Option `-o output_file` kann der Dateiname des resultierenden **ausführbaren Programms** festgelegt werden.
- **C** ist in **C++** enthalten; jeden **C**-Programmcode kann man auch mit einem **C++-Compiler** kompilieren.
- Mein Tipp: Verwendet einen **C++-Compiler**; dies erlaubt eine übersichtlichere Gestaltung des Programmcodes (z.B. Kommentare mit `//`, Variablen müssen nicht am Anfang von Funktionen definiert werden, können auch im Kopf einer **for**-Schleife definiert werden, etc.).
- Im Folgenden 99% **C**, 1% **C++**.
- Programm starten:

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ./minimal
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$
```

printf, #include, Kommentare

printf, #include

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     printf("123 456 789 abc def ghi\n");
6. }
```

```
123 456 789 abc def ghi
```

- Die **Funktion printf** zeigt Text an (`\n` bewirkt einen Zeilenumbruch).
- **printf** ist in der **Standard Input- und Output Bibliothek `stdio.h`** enthalten; diese wird mit **#include** ins Programm eingebunden.
- **C-Anweisungen** enden mit `;`.

Kommentare

```
1. // Standard Input- und Output Bibliothek (enthaelt printf).
2. #include<stdio.h>
3.
4. // main-Funktion des C-Programms.
5. int main(void)
6. {
7.     // Zeigt den Text "123 456 789 abc def ghi\n" an.
8.     printf("123 456 789 abc def ghi\n");
9. }
```

- **Kommentare** sind den Programmcode erklärende Textzeilen, die keinen Einfluss auf das ausführbare Programm haben.
- **Programmcode sollte stets sorgfältig und umfangreich kommentiert werden; sonst ist er oft nach wenigen Tagen wertlos (nicht mehr verständlich, daher schwer erweiterbar oder korrigierbar).**
- `//` leitet Kommentare ein; diese enden beim nächsten Zeilenumbruch (eigentlich keine **C**-Syntax, sondern bereits **C++**).
- Alternativ Kommentare zwischen `/*` und `*/`; ebenso Kommentare über mehrere Zeilen.

```
1. /*
2.     Standard Input- und Output Bibliothek
3.     (enthaelt printf).
4. */
5. #include<stdio.h>
6.
7. /* main-Funktion des C-Programms. */
8. int main(void)
9. {
10.     // Zeigt den Text "123 456 789 abc def ghi\n" an.
11.     printf("123 456 789 abc def ghi\n");
12. }
```

Arithmetische Operationen

+, -, *, /, Ausgabe von Zahlen

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     // Operationen mit ganzen Zahlen.
6.     printf("%d\n", 2+3);
7.     printf("%d\n", 4-11);
8.
9.     // Operationen mit Gleitkommazahlen (entsprechen den reellen Zahlen).
10.    printf("Multiplikation von 2.5 mit 25.0: %f (sollte 62.5 ergeben).\n", 2.5*25.0);
11.    printf("%f\n", 0.5 + 7.0/2.0);
12.    // Verschiedene Ausgabeformate moeglich.
13.    printf("%f %e\n", 2.5*25.0, 2.5*25.0);
14.
15.    // Achtung! Gleitkommazahlen (reelle Zahlen) immer mit Dezimalpunkt
16.    // schreiben, sonst wird in Raum der ganzen Zahlen gerechnet und abgerundet.
17.    printf("%f    !!! Haeufige Fehlerquelle !!!\n", 0.5 + 7/2);
18. }
```

```
5
-7
Multiplikation von 2.5 mit 25.0: 62.500000 (sollte 62.5 ergeben).
4.000000
62.500000 6.250000e+01
3.500000    !!! Haeufige Fehlerquelle !!!
```

- Es existieren verschiedene **Datentypen**; wichtig vor allem
 - **Integer** (entsprechen den ganzen Zahlen; im Programmcode z.B. **3**, **17**, **-5**, ...),
 - **Gleitkommazahlen** (entsprechen den reellen Zahlen; im Programmcode z.B. **3.2**, **-13.224**, **4.0**, ...).
 - **Vorsicht! Häufige Fehlerquelle ...**
Dezimalpunkt vergessen ... z.B. **7/2** (ergibt 3, da im Raum der ganzen Zahlen gerechnet wird) an Stelle von **7.0/2.0** (ergibt 3.5); siehe Zeile 17.
- Integer und Gleitkommazahlen können mit **printf** und durch Verwendung von **%d** (Integer) oder **%f** bzw. **%e** (Gleitkommazahlen) im **Formatstring** angezeigt werden.

Wie lernt man eine Programmiersprache?

- Vor allem durch Übung, daher ...
- **Hausaufgabe:**
 - **Tippe sämtliche in diesem Semester in der Vorlesung gezeigten Programme selbst in Deinen Computer ein ...**
 - **... verändere die Programme, "spiele mit den Befehlen rum", probiere alle denkbaren Variationen aus.**
 - **Kann man auch drei oder mehrere Zahlen mit printf ausgeben?**

```
1. ...
2.    printf("%d %d %d\n", 2, 3, 4);
3. ...
```

- **Was passiert, wenn man Integer und Gleitkommazahlen in einer Rechnung mischt, z.B. 7.0/2?**

```
1. ...
2.    printf("%d\n", 7.0/2);
3.    printf("%f\n", 7.0/2);
4. ...
```

- **Was passiert, wenn man im Formatstring %d schreibt, aber danach die Integer-Zahl vergisst?**

```
1. ...
2.    printf("%d\n");
3. ...
```

- **Usw. ... usw. ... usw. ...**

pow, sqrt, sin, cos, ..., exp, log

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

```

2. #include<stdio.h>
3.
4. int main(void)
5. {
6.     printf("5^3 = %f\n", pow(5.0, 3.0));
7.     printf("sqrt(64.0) = %f\n", sqrt(64.0));
8.
9.     // Die Konstante pi; definiert in math.h.
10.    printf("pi = %f\n", M_PI);
11.
12.    printf("%f %f %f %f\n", sin(0.0), sin(M_PI/2.0), sin(M_PI), sin(3.0*M_PI/2.0));
13.    printf("%f %f %f %f\n", cos(0.0), cos(M_PI/2.0), cos(M_PI), cos(3.0*M_PI/2.0));
14.
15.    printf("e^1 = %f, e^2 = %f\n", exp(1.0), exp(2.0));
16.    printf("ln(1) = %f, ln(e^1) = %f\n", log(1.0), log(exp(1.0)));
17. }

```

```

5^3 = 125.000000
sqrt(64.0) = 8.000000
pi = 3.141593
+0.000000 +1.000000 +0.000000 -1.000000
+1.000000 +0.000000 -1.000000 -0.000000
e^1 = 2.718282, e^2 = 7.389056
ln(1) = 0.000000, ln(e^1) = 1.000000

```

- Mathematische Funktionen, z.B. **pow**, **sqrt**, **sin**, **cos**, ..., **exp**, **log** sind in der Bibliothek **math.h** enthalten.
- In **math.h** sind auch nützliche mathematische Konstanten definiert, z.B. **M_PI** $\equiv \pi$.

Variablen

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int a; // Definition der Integer-Variablen a.
6.     int b, c; // Definition der Integer-Variablen b und c.
7.
8.     a = 7; // Zuweisung der Zahl 7 an die Variable a.
9.     printf("a = %d\n", a);
10.
11.    a = 13;
12.    printf("a = %d\n", a);
13.
14.    a = a+2;
15.    printf("a = %d\n", a);
16.
17.    b = 11;
18.    printf("a + b = %d\n", a + b);
19.
20.    c = a + b;
21.    printf("c = %d\n", c);
22.
23.    double x, y; // Definition der double-Variablen ("Gleitkomma-Variablen") b und c.
24.    x = 3.0;
25.    y = 2.4;
26.    x = 2.0*x + y;
27.    printf("x = %f\n", x);
28. }
```

```
a = 7
a = 13
a = 15
a + b = 26
c = 26
x = 8.400000
```

- **Variablen** haben
 - einen Namen (z.B. **x**; siehe Zeile 23),
 - einen Typ (z.B. ist die Variable **x** vom Typ **double**; siehe Zeile 23),
 - einen Wert (z.B. wird in Zeile 24 der Variable **x** der Wert **3.0** zugewiesen).
- Insbesondere bei umfangreichen Programmcodes empfiehlt sich eine im Wesentlichen selbsterklärende Namensgebung.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     double m_proton = 938.272; // Masse des Protons in MeV/c^2.
6.     double m_neutron = 939.565; // Masse des Neutrons in MeV/c^2.
7.     double m_electron = 0.510999; // Masse des Elektrons in MeV/c^2.
8.
9.     double m_Helium_constituents = 2.0*m_proton + 2.0*m_neutron + 2.0*m_electron;
10.
11.    printf("Konstituentenmasse des Helium-Atoms: %f MeV/c^2.\n", m_Helium_constituents);
12. }
```

```
Konstituentenmasse des Helium-Atoms: 3756.695998 MeV/c^2.
```

- **Vorsicht! Häufige Fehlerquelle ...**
Initialisierung einer Variable vergessen (im folgenden Beispiel **m_neutron**) ... liefert falsche, "zufällige" Ergebnisse.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     double m_proton, m_neutron, m_electron;
6.
7.     m_proton = 938.272; // Masse des Protons in MeV/c^2.
8.     m_electron = 0.510999; // Masse des Elektrons in MeV/c^2.
9.
10.    double m_Helium_constituents = 2.0*m_proton + 2.0*m_neutron + 2.0*m_electron;
```

```
11.  
12. printf("Konstituentenmasse des Helium-Atoms: %f MeV/c^2.\n", m_Helium_constituents);  
13. }
```

```
Konstituentenmasse des Helium-Atoms: 1877.565998 MeV/c^2.
```

Eingabe/Einlesen von Zahlen (scanf)

Eingabe über Tastatur

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     printf("Addition der reellen Zahlen a und b ...\n");
6.
7.     double a;
8.     printf("a = ");
9.     scanf("%lf", &a);
10.
11.    double b;
12.    printf("b = ");
13.    scanf("%lf", &b);
14.
15.    printf("--> a + b = %f\n", a+b);
16. }
```

```
Addition der reellen Zahlen a und b ...
a =
```

```
Addition der reellen Zahlen a und b ...
a = 1.2
b =
```

```
Addition der reellen Zahlen a und b ...
a = 1.2
b = 17.3
--> a + b = 18.500000
```

- Die Funktion **scanf** liest Text oder Zahlen ein ("das Gegenstück zu **printf**").
- **scanf** ist in der Bibliothek **stdio.h** enthalten.
- Einlesen einer **int**-Variable: **%d** im Formatstring, **&** vor der nach dem Formatstring folgenden Variable (**&** ist der **Adressoperator**; seine Bedeutung und Funktion wird in einer der folgenden Vorlesungsstunden erläutert).
- Einlesen einer **double**-Variable: **%lf** im Formatstring, **&** vor der nach dem Formatstring folgenden Variable.

Einlesen aus einer Datei

- Im Terminal kann mit **<** eine im Folgenden erforderliche Eingabe über die Tastatur durch eine Textdatei ersetzt werden.

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 16
-rwxrwxr-x 1 mwagner mwagner 9033 Okt 19 20:12 a_plus_b
-rw-rw-r-- 1 mwagner mwagner 237 Okt 19 20:12 a_plus_b.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ echo 3.3 2.1 > eingabe.txt
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 20
-rwxrwxr-x 1 mwagner mwagner 9033 Okt 19 20:12 a_plus_b
-rw-rw-r-- 1 mwagner mwagner 237 Okt 19 20:12 a_plus_b.c
-rw-rw-r-- 1 mwagner mwagner 8 Okt 19 20:42 eingabe.txt
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./a_plus_b < eingabe.txt
Addition der reellen Zahlen a und b ...
a = b = --> a + b = 5.400000
```

- Bei Einlesen aus einer Datei empfiehlt sich ein minimales Umschreiben der Bildschirmausgabe.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     printf("Addition der reellen Zahlen a und b ...\n");
6.
7.     double a;
8.     scanf("%lf", &a);
9.     printf("a = %f\n", a);
10.
11.    double b;
12.    scanf("%lf", &b);
13.    printf("b = %f\n", b);
14.
15.    printf("--> a + b = %f\n", a+b);
16. }
```

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./a_plus_b < eingabe.txt
Addition der reellen Zahlen a und b ...
a = 3.300000
b = 2.100000
--> a + b = 5.400000
```

Befehlsbeschreibung via man

- **man** im Terminal funktioniert auch für **C**-Befehle.

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ man scanf
```

```
SCANF(3) Linux Programmer's Manual SCANF(3)
NAME
scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf - input format conversion
SYNOPSIS
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);

#include <stdarg.h>

int vscanf(const char *format, va_list ap);
int vsscanf(const char *str, const char *format, va_list ap);
int vfscanf(FILE *stream, const char *format, va_list ap);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

vscanf(), vsscanf(), vfscanf():
_XOPEN_SOURCE >= 600 || _ISOC99_SOURCE || _POSIX_C_SOURCE >= 200112L;
or cc -std=c99

DESCRIPTION
The scanf() family of functions scans input according to format as
described below. This format may contain conversion specifications; the
results from such conversions, if any, are stored in the locations
pointed to by the pointer arguments that follow format. Each pointer
argument must be of a type that is appropriate for the value returned
by the corresponding conversion specification.
```

Kontrollstrukturen 1: Fallunterscheidungen

• if ...

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     double x = 3.25;
6.     double sign_x;
7.
8.     if(x > 0.0)
9.         sign_x = +1.0;
10.    if(x == 0.0)
11.        sign_x = 0.0;
12.    if(x < 0.0)
13.        sign_x = -1.0;
14.
15.    printf("x = %f --> sign(x) = %f\n", x, sign_x);
16. }
```

```
x = 3.250000 --> sign(x) = 1.000000
```

```
4. ...
5.     double x = 0.0;
6.     ...
```

```
x = 0.000000 --> sign(x) = 0.000000
```

```
4. ...
5.     double x = -113.0;
6.     ...
```

```
x = -113.000000 --> sign(x) = -1.000000
```

• Vergleichsoperatoren:

- == (gleich).
- != (ungleich).
- < (kleiner).
- > (größer).
- <= (kleiner gleich).
- >= (größer gleich).

• if ... else ...

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int x = 5;
6.     printf("x = %d\n", x);
7.
8.     if(x == 0)
9.         printf("x ist 0\n");
10.    else
11.        printf("x ist nicht 0\n");
12. }
```

```
x = 5
x ist nicht 0
```

- Mehr als eine Anweisung im **if**- bzw. **else**-Zweig durch **{** und **}**.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int x = 5;
6.     printf("x = %d\n", x);
7.
8.     if(x == 0)
9.     {
```

```

10.     printf("*****\n");
11.     printf("x ist 0\n");
12.     printf("*****\n");
13.     }
14. else
15.     {
16.         printf("++++++\n");
17.         printf("x ist nicht 0\n");
18.         printf("++++++\n");
19.     }
20. }

```

```

x = 5
+++++++
x ist nicht 0
+++++++

```

- **Vorsicht! Häufige Fehlerquelle ...**

Zuweisungsoperator = an Stelle des Vergleichsoperators ==.

```

1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int x = 0;
6.     printf("x = %d\n", x);
7.
8.     if(x = 0) // Häufige Fehlerquelle!
9.         printf("x ist 0\n");
10.    else
11.        printf("x ist nicht 0\n");
12.
13.    printf("x = %d\n", x);
14. }

```

```

x = 0
x ist nicht 0
x = 0

```

```

1. ...
2. int x = 5;
3. ...

```

```

x = 5
x ist nicht 0
x = 0

```

Kontrollstrukturen 2: Schleifen

- Häufig muss eine Folge von Anweisungen mehrmals hintereinander wiederholt werden. Dies realisiert man in Form von Schleifen (**Loops**).

while-Schleife

- **while**(*expr*) {...}:
 - Die Anweisungen innerhalb einer **while**-Schleife (innerhalb von { und }) werden wiederholt, so lange das Kriterium *expr* erfüllt ist.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int num_iter = 5; // Anzahl der Iterationen.
6.     int ctr = 1; // Zaehlt die Schleifendurchlaeufe.
7.
8.     while(ctr <= num_iter)
9.     {
10.         printf("%d-ter Schleifendurchlauf ...\n", ctr);
11.         ctr = ctr+1;
12.     }
13. }
```

```
1-ter Schleifendurchlauf ...
2-ter Schleifendurchlauf ...
3-ter Schleifendurchlauf ...
4-ter Schleifendurchlauf ...
5-ter Schleifendurchlauf ...
```

for-Schleife

- **for**(*expr1*; *expr2*; *expr3*) {...}:
 - Die Anweisungen innerhalb einer **for**-Schleife (innerhalb von { und }) werden wiederholt, so lange das Kriterium *expr2* erfüllt ist.
 - Vor dem ersten Schleifendurchlauf wird *expr1* ausgewertet, in der Regel die Initialisierung einer Zählvariable.
 - Nach jedem Schleifendurchlauf wird *expr3* ausgewertet, in der Regel eine Veränderung einer Zählvariable.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int num_iter = 5; // Anzahl der Iterationen.
6.
7.     for(int ctr = 1; ctr <= num_iter; ctr = ctr+1)
8.     {
9.         printf("%d-ter Schleifendurchlauf ...\n", ctr);
10.    }
11. }
```

```
1-ter Schleifendurchlauf ...
2-ter Schleifendurchlauf ...
3-ter Schleifendurchlauf ...
4-ter Schleifendurchlauf ...
5-ter Schleifendurchlauf ...
```

- Jede **while**-Schleife kann auch durch eine **for**-Schleife realisiert werden ... und umgekehrt.
- **for**-Schleifen sind häufig eleganter (Initialisierung der Zählvariable, Abbruchkriterium und Veränderung der Zählvariable übersichtlich zu Beginn der Schleife zusammengefasst).
- **while**(*expr*) {...} und **for**(; *expr*;) {...} sind äquivalent.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     int num_iter = 5; // Anzahl der Iterationen.
6.     int ctr = 1; // Zaehlt die Schleifendurchlaeufe.
7.
8.     for(; ctr <= num_iter; )
9.     {
```

```
10.     printf("%d-ter Schleifendurchlauf ...\n", ctr);
11.     ctr = ctr+1;
12. }
13. }
```

- **Beispiel: Fakultät ...**

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. int main(void)
5. {
6.     printf("Fakultaet von x ...\n");
7.
8.     int x;
9.     printf("x = ");
10.    scanf("%d", &x);
11.
12.    if(x < 0)
13.    {
14.        printf("Fehler: x >= 0 erforderlich!\n");
15.        exit(0);
16.    }
17.
18.    int fac_x = 1;
19.
20.    for(int ctr = 1; ctr <= x; ctr = ctr+1)
21.        fac_x = fac_x * ctr;
22.
23.    printf("x! = %d\n", fac_x);
24. }
```

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./fac_x
Fakultaet von x ...
x = 3
x! = 6
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./fac_x
Fakultaet von x ...
x = 4
x! = 24
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./fac_x
Fakultaet von x ...
x = 0
x! = 1
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./fac_x
Fakultaet von x ...
x = -7
Fehler: x >= 0 erforderlich!
```

- **Benutzereingaben sollten stets überprüft werden ... falls nicht sinnvoll, sollte das Programm mit einer aussagekräftigen Fehlermeldung enden.**
- **exit(0)** beendet ein Programm.
- **exit** ist in der Bibliothek **stdlib.h** enthalten.

Eigene Funktionen

- Häufig muss eine Folge von Anweisungen an mehreren Stellen eines Programms wiederholt werden. Dies realisiert man elegant mit Hilfe von Funktionen.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. // *****
5.
6. // Die Funktion factorial berechnet die Fakultät von x.
7. int factorial(int x)
8. {
9.     if(x < 0)
10.    {
11.        printf("Fehler in int factorial(int x): x >= 0 erforderlich!\n");
12.        exit(0);
13.    }
14.
15.    int fac_x = 1;
16.
17.    for(int ctr = 1; ctr <= x; ctr = ctr+1)
18.        fac_x = fac_x * ctr;
19.
20.    return fac_x;
21. }
22.
23. // *****
24.
25. int main(void)
26. {
27.     printf("Fakultät von x, x+2, x+5 ... \n");
28.
29.     int x;
30.     printf("x = ");
31.     scanf("%d", &x);
32.
33.     int fac_x = factorial(x); // Die Funktion factorial wird aufgerufen.
34.     printf("x! = %d\n", fac_x);
35.     printf("(x+2)! = %d\n", factorial(x+2)); // Die Funktion factorial wird aufgerufen.
36.     printf("(x+5)! = %d\n", factorial(x+5)); // Die Funktion factorial wird aufgerufen.
37. }
```

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./fac
Fakultät von x, x+2, x+5 ...
x = 3
x! = 6
(x+2)! = 120
(x+5)! = 40320
```

- In Zeile 7 wird die Funktion **factorial** definiert.
- **factorial** besitzt einen **Parameter**, die **int**-Variable **x** (ersichtlich an ... **factorial(int x)**).
- **factorial** liefert einen **int**-Wert zurück (**Rückgabewert**; ersichtlich an **int factorial(...)**).
- **return fac_x;** in Zeile 20 beendet die Funktion **factorial** und liefert den Wert der **int**-Variable **fac_x** zurück.
- Allgemein hat eine Funktionsdefinition folgende Form:
`type function_name(type1 para1, type2 para2, ...){...}.`

- **Beispiel:**

```
1. #include<stdio.h>
2.
3. double a_plus_b(double a, double b)
4. {
5.     return a + b;
6. }
7.
8. int main(void)
9. {
10.    double a = 2.0;
11.    double b = 3.0;
12.
13.    printf("%f + %f = %f\n", a, b, a_plus_b(a, b));
```

```
14. }
```

```
2.000000 + 3.000000 = 5.000000
```

• Beispiel:

```
1. #include<stdio.h>
2.
3. void print_abc(void)
4. {
5.     printf("abc\n");
6. }
7.
8. int main(void)
9. {
10.    for(int ctr = 0; ctr < 3; ctr++)
11.        print_abc();
12. }
```

```
abc
abc
abc
```

- Besitzt eine Funktion keine Parameter, wird dies durch `... function_name(void)` kodiert.
- Besitzt eine Funktion keinen Rückgabewert, wird dies durch `void function_name(...)` kodiert.
- Eine Funktion muss vor ihrem ersten Aufruf definiert werden.

```
1. #include<stdio.h>
2.
3. int main(void)
4. {
5.     for(int ctr = 0; ctr < 3; ctr++)
6.         print_abc();
7. }
8.
9. void print_abc(void)
10. {
11.     printf("abc\n");
12. }
```

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o print_abc print_abc.c
print_abc.c: In Funktion "int main()":
print_abc.c:6:15: Fehler: "print_abc" wurde in diesem Gültigkeitsbereich nicht definiert
```