
Einführung in die Programmierung für Physiker

Die Programmiersprache C – Programmstruktur

Marc Wagner

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

WS 2019/20

Funktionen

- Um eine umfangreiche Programmieraufgabe schnell, fehlerfrei und elegant zu lösen, empfiehlt es sich, die Gesamtaufgabe in kleine Teilaufgaben zu zerlegen.
- Die Teilaufgaben werden dann in separaten (idealer Weise kurzen) Funktionen implementiert:
 - **Das Aufsplitten in einzelne Funktionen macht den Programmcode übersichtlich und erhöht dessen Lesbarkeit.**
 - **Die Funktionen können isoliert getestet werden.**
 - **Die Funktionen können unmittelbar in anderen Programmen verwendet werden.**
- Funktionen können voneinander abhängen, d.h. sie können sich gegenseitig aufrufen oder sogar sich selbst aufrufen (**Rekursion**).
- Eine Funktionsdefinition hat folgende Form:
`type function_name(type1 para1, type2 para2, ...){...}`
 - `function_name`: Funktionsname (Einschränkungen wie bei Variablennamen).
 - `type`: Datentyp des Rückgabewerts; dieser wird mit `return` zurückgeliefert; soll eine Funktion keinen Wert zurückgeben, wird dies mit `void` gekennzeichnet.
 - `type1 para1, type2 para2, ...`: Datentyp und Variablenname des ersten, zweiten, ... Parameters (zur Erinnerung: bei **C** werden keine Variablen übergeben, sondern lediglich deren Werte kopiert [**call by value**]); besitzt eine Funktion keine Parameter, wird dies mit `void` gekennzeichnet.

```
1. void f(void)
2. {
3.   ...
4. }
5.
6. int g1(void)
7. {
8.   ...
9.   return 123;
10. }
11.
12. int g2(void)
13. {
14.   int i;
15.   ...
16.   return i
17. }
18.
19. double h(double x, double y)
20. {
21.   return x+y;
22. }
```

- **Beispiel:** Berechnung des Mittelwerts und des statistischen Fehlers einer Messreihe ...

```
1. #include<math.h>
2. #include<stdio.h>
3.
4. // *****
5.
6. // Berechnet den Mittelwert.
7. void mean(int n, double *x, double *x_average)
8. {
9.   int i1;
10.
11.   *x_average = 0.0;
12.
13.   for(i1 = 0; i1 < n; i1++)
14.     *x_average += x[i1];
15.
16.   *x_average /= (double)n;
17. }
18.
19. // *****
```

```

20.
21. // Berechnet den Mittelwert und den statistischen Fehler.
22. void mean_and_error(int n, double *x, double *x_average, double *x_delta)
23. {
24.     int i1;
25.
26.     // Verwende existierende Funktion zur Berechnung des Mittelwerts.
27.     mean(n, x, x_average);
28.
29.     *x_delta = 0.0;
30.
31.     for(i1 = 0; i1 < n; i1++)
32.         *x_delta += pow(x[i1] - *x_average, 2.0);
33.
34.     *x_delta = sqrt(*x_delta / (double)(n*(n-1)));
35. }
36.
37. // *****
38.
39. int main(void)
40. {
41.     // Einige Messwerte.
42.     const int n = 5;
43.     double y[n] = {2.0, 4.0, 5.0, 1.0, 3.0};
44.
45.     // Mittelwert und statistischen Fehler berechnen.
46.     double y_average, y_delta;
47.     mean_and_error(n, y, &y_average, &y_delta);
48.     printf("y = %.3lf +/- %.3lf\n", y_average, y_delta);
49. }

```

y = 3.000 +/- 0.707107

Aufteilen des Programmcodes auf mehrere Dateien

- Bei umfangreichen Programmen ist es häufig übersichtlicher, den Programmcode auf mehrere Dateien aufzuteilen.
- **Beispiel:** Berechnung des Mittelwerts und des statistischen Fehlers einer Messreihe (auf zwei Dateien aufgeteilt) ...
 - Datei **error_analysis.c**:

```
1. #include<math.h>
2.
3. // *****
4.
5. // Berechnet den Mittelwert.
6. void mean(int n, double *x, double *x_average)
7. {
8.     int i1;
9.
10.    *x_average = 0.0;
11.
12.    for(i1 = 0; i1 < n; i1++)
13.        *x_average += x[i1];
14.
15.    *x_average /= (double)n;
16. }
17.
18. // *****
19.
20. // Berechnet den Mittelwert und den statistischen Fehler.
21. void mean_and_error(int n, double *x, double *x_average, double *x_delta)
22. {
23.     int i1;
24.
25.     // Verwende existierende Funktion zur Berechnung des Mittelwerts.
26.     mean(n, x, x_average);
27.
28.     *x_delta = 0.0;
29.
30.     for(i1 = 0; i1 < n; i1++)
31.         *x_delta += pow(x[i1] - *x_average, 2.0);
32.
33.     *x_delta = sqrt(*x_delta / (double)(n*(n-1)));
34. }
```

- Datei **main.c**:
 - Falls in einer Datei Funktionen verwendet werden, die dort nicht definiert sind, müssen sie zumindest **deklariert** werden; dies geschieht durch Angabe des **Funktionskopfs** (Typ des Rückgabewerts, Funktionsname, Parameterliste) mit abschließendem **;**.

```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Funktionsdeklaration.
6. void mean_and_error(int n, double *x, double *x_average, double *x_delta);
7.
8. // *****
9.
10. int main(void)
11. {
12.     // Einige Messwerte.
13.     const int n = 5;
14.     double y[n] = {2.0, 4.0, 5.0, 1.0, 3.0};
15.
16.     // Mittelwert und statistischen Fehler berechnen.
17.     double y_average, y_delta;
18.     mean_and_error(n, y, &y_average, &y_delta);
19.     printf("y = %.3lf +/- %3lf\n", y_average, y_delta);
20. }
```

- Kompilieren und ausführen wie folgt.

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog error_analysis.c main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
y = 3.000 +/- 0.707107
```

- Wird die Funktion `mean_and_error` in `main.c` nicht deklariert, kommt es beim Kompilieren zu einem Fehler.

```
4. ...
5. // Funktionsdeklaration.
6. // void mean_and_error(int n, double *x, double *x_average, double *x_delta);
7. ...
```

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog error_analysis.c main.c
main.c: In Funktion "int main()":
main.c:18:44: Fehler: "mean_and_error" wurde in diesem Gültigkeitsbereich nicht definiert
```

- Funktionsdeklarationen erlauben außerdem eine beliebige Reihenfolge der Funktionsdefinitionen im Programmcode.
- **Beispiel:** Definition der `main`-Funktion zuerst, dann Definition einer Funktion, die in der `main`-Funktion aufgerufen wird ...

```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Funktionsdeklaration von f.
6. void f(void);
7.
8. // *****
9.
10. int main(void)
11. {
12.     f();
13. }
14.
15. // *****
16.
17. // Funktionsdefinition von f.
18. void f(void)
19. {
20.     printf("Funktion f ...\n");
21. }
```

```
Funktion f ...
```

Eigene include-Dateien

- Die in anderen Dateien definierten Funktionen alle "von Hand" zu deklarieren, ist oft umständlich.
- Eleganter ist die Verwendung eigener **include-Dateien** (**Header-Dateien**):
 - Typischer Weise wird zu jeder `filename.c`-Datei eine `filename.h` Datei angelegt, in der die Deklarationen aller Funktionen (aber nicht deren Definitionen) enthalten sind.
 - Die include-Datei `filename.h` wird mittels `#include"filename.h"` in alle `.c`-Dateien eingebunden, die Funktionen aus `filename.c` verwenden.
 - Der sogenannte **Präprozessor** ersetzt vor dem eigentlichen Kompilieren `#include"filename"` bzw. `#include<filename>` durch den Inhalt der Datei `filename`.
 - `#include"filename"`: Die Datei `filename` wird zunächst im aktuellen Verzeichnis gesucht.
 - `#include<filename>`: Die Datei `filename` wird in den Verzeichnissen gesucht, in denen sich die **C**-Standard-include-Dateien befinden.
- **Beispiel**: Berechnung des Mittelwerts und des statistischen Fehlers einer Messreihe (auf drei Dateien aufgeteilt, eine `.h`-Datei, zwei `.c`-Dateien) ...
 - Datei **error_analysis.h**: Die eigentlich interessante Datei für einen Benutzer der Funktionen `mean` und `mean_and_error` ...

```
1. // Berechnet den Mittelwert.
2. void mean(int n, double *x, double *x_average);
3.
4. // Berechnet den Mittelwert und den statistischen Fehler.
5. void mean_and_error(int n, double *x, double *x_average, double *x_delta);
```

- Datei **error_analysis.c**: Der Inhalt dieser Datei ist für einen Benutzer der Funktionen `mean` und `mean_and_error` nur von geringem Interesse ...

```
1. // Einbinden der C-Standard-include-Datei math.h.
2. #include<math.h>
3.
4. // *****
5.
6. // Berechnet den Mittelwert.
7. void mean(int n, double *x, double *x_average)
8. {
9.     int i1;
10.
11.     *x_average = 0.0;
12.
13.     for(i1 = 0; i1 < n; i1++)
14.         *x_average += x[i1];
15.
16.     *x_average /= (double)n;
17. }
18.
19. // *****
20.
21. // Berechnet den Mittelwert und den statistischen Fehler.
22. void mean_and_error(int n, double *x, double *x_average, double *x_delta)
23. {
24.     int i1;
25.
26.     // Verwende existierende Funktion zur Berechnung des Mittelwerts.
27.     mean(n, x, x_average);
28.
29.     *x_delta = 0.0;
30.
31.     for(i1 = 0; i1 < n; i1++)
32.         *x_delta += pow(x[i1] - *x_average, 2.0);
33.
34.     *x_delta = sqrt(*x_delta / (double)(n*(n-1)));
35. }
```

- Datei **main.c**:

- `#include"error_analysis.h"`

bindet die in `error_analysis.h` enthaltenen Deklarationen an der entsprechenden Stelle in `main.c` ein.

```
1. // Einbinden der C-Standard-include-Datei stdio.h.
2. #include<stdio.h>
3.
4. // Einbinden der Deklaration der Funktion mean_and_error.
5. #include"error_analysis.h"
6.
7. // *****
8.
9. int main(void)
10. {
11.     // Einige Messwerte.
12.     const int n = 5;
13.     double y[n] = {2.0, 4.0, 5.0, 1.0, 3.0};
14.
15.     // Mittelwert und statistischen Fehler berechnen.
16.     double y_average, y_delta;
17.     mean_and_error(n, y, &y_average, &y_delta);
18.     printf("y = %.3lf +/- %.3lf\n", y_average, y_delta);
19. }
```

- Kompilieren und ausführen wie gehabt.

```
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ g++ -o prog error_analysis.c main.c
mwagner@laptop-tigger:~/lecture_ProgPhys/slides/tmp$ ./prog
y = 3.000 +/- 0.707107
```

Objektdateien

- Häufig ist es zweckmäßig Teile oder einzelne Dateien eines Programms (also einige Funktionen) separat zu kompilieren; das Resultat ist dann kein ausführbares Programm, sondern eine **Objektdatei** (typischer Weise mit Endung **.o**).
- Zu einem späteren Zeitpunkt können dann mehrere Objektdateien zu einem vollständigen ausführbaren Programm verbunden (**gelinkt**) werden.
- Vorteile:
 - Das Kompilieren dauert deutlich länger, als das Linken; es ist daher praktisch, d.h. zeitsparend, fertige, funktionierende Programmteile nicht jedes Mal neu kompilieren zu müssen.
 - Eine Objektdatei kann in mehrere Programme gelinkt werden (praktisch, wenn Programmteile allgemein einsetzbare Funktionen enthalten, z.B. Matrix- und Vektor-Operationen, Nullstellensuche, Mittelwert- und Fehlerberechnung, ...).
- **Beispiel:** Berechnung des Mittelwerts und des statistischen Fehlers einer Messreihe (wie gehabt auf drei Dateien aufgeteilt, eine **.h**-Datei, zwei **.c**-Dateien) ...
 - Datei **error_analysis.h**:

```
1. // Berechnet den Mittelwert.
2. void mean(int n, double *x, double *x_average);
3.
4. // Berechnet den Mittelwert und den statistischen Fehler.
5. void mean_and_error(int n, double *x, double *x_average, double *x_delta);
```

- Datei **error_analysis.c**:

```
1. // Einbinden der C-Standard-include-Datei math.h.
2. #include<math.h>
3.
4. // *****
5.
6. // Berechnet den Mittelwert.
7. void mean(int n, double *x, double *x_average)
8. {
9.     int i1;
10.
11.     *x_average = 0.0;
12.
13.     for(i1 = 0; i1 < n; i1++)
14.         *x_average += x[i1];
15.
16.     *x_average /= (double)n;
17. }
18.
19. // *****
20.
21. // Berechnet den Mittelwert und den statistischen Fehler.
22. void mean_and_error(int n, double *x, double *x_average, double *x_delta)
23. {
24.     int i1;
25.
26.     // Verwende existierende Funktion zur Berechnung des Mittelwerts.
27.     mean(n, x, x_average);
28.
29.     *x_delta = 0.0;
30.
31.     for(i1 = 0; i1 < n; i1++)
32.         *x_delta += pow(x[i1] - *x_average, 2.0);
33.
34.     *x_delta = sqrt(*x_delta / (double)(n*(n-1)));
35. }
```

- Datei **main.c**:

```
1. // Einbinden der C-Standard-include-Datei stdio.h.
2. #include<stdio.h>
3.
4. // Einbinden der Deklaration der Funktion mean_and_error.
5. #include"error_analysis.h"
6.
```

```

7. // *****
8.
9. int main(void)
10. {
11. // Einige Messwerte.
12. const int n = 5;
13. double y[n] = {2.0, 4.0, 5.0, 1.0, 3.0};
14.
15. // Mittelwert und statistischen Fehler berechnen.
16. double y_average, y_delta;
17. mean_and_error(n, y, &y_average, &y_delta);
18. printf("y = %.3lf +/- %.3lf\n", y_average, y_delta);
19. }

```

- Variante 1: Zunächst Kompilieren von **error_analysis.c** zu einer Objektdatei **error_analysis.o**; dann Kompilieren von **main.c** und Linken mit **error_analysis.o** zum ausführbaren Programm **prog**.

```

mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 12
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o error_analysis.o -c error_analysis.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 16
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 1896 Dez  3 23:58 error_analysis.o
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog error_analysis.o main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 28
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 1896 Dez  3 23:58 error_analysis.o
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
-rwxrwxr-x 1 mwagner mwagner 9159 Dez  3 23:59 prog
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
y = 3.000 +/- 0.707107

```

- Variante 2: Zunächst Kompilieren von **error_analysis.c** zu einer Objektdatei **error_analysis.o** und von **main.c** zu einer Objektdatei **main.o**; dann Linken von **error_analysis.o** und **main.o** zum ausführbaren Programm **prog**.

```

mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 12
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o error_analysis.o -c error_analysis.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o main.o -c main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 20
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 1896 Dez  4 00:03 error_analysis.o
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
-rw-rw-r-- 1 mwagner mwagner 2016 Dez  4 00:04 main.o
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog error_analysis.o main.o
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 32
-rw-rw-r-- 1 mwagner mwagner 634 Dez  3 23:57 error_analysis.c
-rw-rw-r-- 1 mwagner mwagner 199 Dez  3 23:57 error_analysis.h
-rw-rw-r-- 1 mwagner mwagner 1896 Dez  4 00:03 error_analysis.o
-rw-rw-r-- 1 mwagner mwagner 418 Dez  3 23:58 main.c
-rw-rw-r-- 1 mwagner mwagner 2016 Dez  4 00:04 main.o
-rwxrwxr-x 1 mwagner mwagner 9159 Dez  4 00:05 prog
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
y = 3.000 +/- 0.707107

```

- Die in einer Objektdatei enthaltenen und von ihr benötigten Funktionen können mit dem Befehl **nm** angezeigt werden (Details mit **man nm**).

```

mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ nm -C error_analysis.o
                 U pow
                 U sqrt
0000000000000084 T mean_and_error(int, double*, double*, double*)
0000000000000000 T mean(int, double*, double*)
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ nm -C main.o
                 U __gxx_personality_v0
0000000000000000 T main
                 U printf
                 U _Unwind_Resume
                 U mean_and_error(int, double*, double*, double*)

```

Globale Variablen in mehreren Dateien verwenden

- Eine **globale Variable** (eine Variable, die außerhalb von Funktionen definiert ist) ist nur in der **.c**-Datei bekannt, in der sie definiert wurde.
- Soll eine solche Variable auch in anderen **.c**-Dateien verwendet werden, muss sie dort deklariert werden (analog zur Definition und Deklaration bei Funktionen); dies geschieht mit **extern type name;**
- **Beispiel:** Eine globale **int**-Variable **num** (definiert in **f.c**), die zählt, wie oft die Funktion **f** (ebenfalls definiert in **f.c**) aufgerufen wird; der Wert von **num** soll in einer anderen Datei (**main.c**) ausgegeben werden ...
 - Variante 1:
 - Datei **f.c**:

```
1. #include<stdio.h>
2.
3. // Definition der globalen Variable num.
4. int num = 0; // Zaehlt die Anzahl der Funktionsaufrufe von f.
5.
6. // Definition der Funktion f.
7. void f(void)
8. {
9.     num++;
10.    printf("Funktion f ...\n");
11. }
```

- Datei **main.c**:

```
1. #include<stdio.h>
2.
3. // Deklaration der globalen Variable num (definiert in der Datei f.c).
4. extern int num;
5.
6. // Deklaration der Funktion f (definiert in der Datei f.c).
7. void f(void);
8.
9. int main(void)
10. {
11.     f();
12.     f();
13.     f();
14.     printf("Die Funktion f wurde %d Mal aufgerufen.\n", num);
15. }
```

- Kompilieren und ausführen wie folgt.

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 8
-rw-rw-r-- 1 mwagner mwagner 210 Dez  4 16:11 f.c
-rw-rw-r-- 1 mwagner mwagner 282 Dez  4 16:11 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog f.c main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 20
-rw-rw-r-- 1 mwagner mwagner 210 Dez  4 16:11 f.c
-rw-rw-r-- 1 mwagner mwagner 282 Dez  4 16:11 main.c
-rwxrwxr-x 1 mwagner mwagner 8843 Dez  4 16:16 prog
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
Funktion f ...
Funktion f ...
Funktion f ...
Die Funktion f wurde 3 Mal aufgerufen.
```

- Variante 2:
 - Datei **f.h**:

```
1. // Deklaration der globalen Variable num (definiert in der Datei f.c).
2. extern int num;
3.
4. // Deklaration der Funktion f (definiert in der Datei f.c).
5. void f(void);
```

- Datei **f.c**:

```
1. #include<stdio.h>
2.
3. // Definition der globalen Variable num.
4. int num = 0; // Zaehlt die Anzahl der Funktionsaufrufe von f.
```

```
5.
6. // Definition der Funktion f.
7. void f(void)
8. {
9.     num++;
10.    printf("Funktion f ...\n");
11. }
```

- Datei **main.c**:

```
1. #include<stdio.h>
2. #include"f.h"
3.
4. int main(void)
5. {
6.     f();
7.     f();
8.     f();
9.     printf("Die Funktion f wurde %d Mal aufgerufen.\n", num);
10. }
```

- Kompilieren und ausführen wie gehabt.

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 12
-rw-rw-r-- 1 mwagner mwagner 210 Dez  4 16:11 f.c
-rw-rw-r-- 1 mwagner mwagner 162 Dez  4 16:19 f.h
-rw-rw-r-- 1 mwagner mwagner 133 Dez  4 16:19 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog f.c main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 24
-rw-rw-r-- 1 mwagner mwagner 210 Dez  4 16:11 f.c
-rw-rw-r-- 1 mwagner mwagner 162 Dez  4 16:19 f.h
-rw-rw-r-- 1 mwagner mwagner 133 Dez  4 16:19 main.c
-rwxrwxr-x 1 mwagner mwagner 8843 Dez  4 16:20 prog
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
Funktion f ...
Funktion f ...
Funktion f ...
Die Funktion f wurde 3 Mal aufgerufen.
```

static

static bei Funktionen und globalen Variablen

- Das Schlüsselwort **static** verhindert, dass Funktionen und globale Variablen außerhalb der Datei sichtbar sind, in der sie definiert werden.
- Vorteile:
 - Sicherheit: Globale **static**-Variablen können von außen nicht absichtlich oder aus Versehen verändert werden.
 - Kein Konflikt mit gleichnamigen **static**-Funktionen und globalen **static**-Variablen in anderen Dateien.
- **Beispiel:** In den Dateien **f.c** und **g.c** existiert jeweils eine globale **static**-Variable **num** und eine **static**-Funktion **print_num**; trotz gleicher Namensgebung, kommt es nicht zu Konflikten ...
 - Datei **f.c**:

```
1. #include<stdio.h>
2.
3. // Definition der globalen Variable num; aufgrund von static auf diese Datei beschaenkt.
4. static int num = 0; // Zaehlt die Anzahl der Funktionsaufrufe von f.
5.
6. // Definition der Funktion print_num; aufgrund von static auf diese Datei beschaenkt.
7. static void print_num()
8. {
9.     printf("Die Funktion f wurde %d Mal aufgerufen.\n", num);
10. }
11.
12. void f(void)
13. {
14.     num++;
15.     printf("Funktion f ...\n");
16.     print_num();
17. }
```

- Datei **g.c**:

```
1. #include<stdio.h>
2.
3. // Definition der globalen Variable num; aufgrund von static auf diese Datei beschaenkt.
4. static int num = 0; // Zaehlt die Anzahl der Funktionsaufrufe von g.
5.
6. // Definition der Funktion print_num; aufgrund von static auf diese Datei beschaenkt.
7. static void print_num()
8. {
9.     printf("Die Funktion g wurde %d Mal aufgerufen.\n", num);
10. }
11.
12. void g(void)
13. {
14.     num++;
15.     printf("Funktion g ...\n");
16.     print_num();
17. }
```

- Datei **main.c**:

```
1. void f(void);
2. void g(void);
3.
4. int main(void)
5. {
6.     f();
7.     f();
8.     f();
9.
10.    g();
11.    g();
12. }
```

- Kompilieren und ausführen wie folgt.

```

mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 12
-rw-rw-r-- 1 mwagner mwagner 427 Dez  4 17:02 f.c
-rw-rw-r-- 1 mwagner mwagner 427 Dez  4 17:01 g.c
-rw-rw-r-- 1 mwagner mwagner  84 Dez  4 16:59 main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ g++ -o prog f.c g.c main.c
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ls -l
insgesamt 24
-rw-rw-r-- 1 mwagner mwagner 427 Dez  4 17:02 f.c
-rw-rw-r-- 1 mwagner mwagner 427 Dez  4 17:01 g.c
-rw-rw-r-- 1 mwagner mwagner  84 Dez  4 16:59 main.c
-rwxrwxr-x 1 mwagner mwagner 8992 Dez  4 17:04 prog
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ ./prog
Funktion f ...
Die Funktion f wurde 1 Mal aufgerufen.
Funktion f ...
Die Funktion f wurde 2 Mal aufgerufen.
Funktion f ...
Die Funktion f wurde 3 Mal aufgerufen.
Funktion g ...
Die Funktion g wurde 1 Mal aufgerufen.
Funktion g ...
Die Funktion g wurde 2 Mal aufgerufen.

```

static bei lokalen Variablen

- Das Schlüsselwort **static** bei einer **lokalen Variable** (eine Variable, die innerhalb einer Funktion definiert ist) bewirkt, dass diese Variable nur einmalig initialisiert wird und ihr Wert auch nach Beendigung "ihrer Funktion" erhalten bleibt; d.h. beim nächsten Aufruf dieser Funktion steht der Wert der lokalen **static**-Variable unverändert zur Verfügung.
- **Beispiel:** Zählen der Aufrufe einer Funktion ...

```

1. #include<stdio.h>
2.
3. void f(void)
4. {
5.     static int num = 0;
6.     num++;
7.     printf("%d-ter Funktionsaufruf von f ...\n", num);
8. }
9.
10. int main(void)
11. {
12.     f();
13.     f();
14.     f();
15. }

```

```

1-ter Funktionsaufruf von f ...
2-ter Funktionsaufruf von f ...
3-ter Funktionsaufruf von f ...

```

- Zum Vergleich das gleiche Programm ohne das Schlüsselwort static.

```

1. #include<stdio.h>
2.
3. void f(void)
4. {
5.     int num = 0;
6.     num++;
7.     printf("%d-ter Funktionsaufruf von f ...\n", num);
8. }
9.
10. int main(void)
11. {
12.     f();
13.     f();
14.     f();
15. }

```

```

1-ter Funktionsaufruf von f ...
1-ter Funktionsaufruf von f ...
1-ter Funktionsaufruf von f ...

```

Gültigkeitsbereich von Variablen

- Unterscheide
 - **globale Variablen** (Variablen, die außerhalb von Funktionen definiert sind),
 - **lokale Variablen** (Variablen, die innerhalb von Funktionen definiert sind).
- Als **Gültigkeitsbereich** einer Variable bezeichnet man den Bereich des Programmcodes, in dem die Variable verwendet werden kann.

Gültigkeitsbereich globaler Variablen

- Eine globale Variable ist gültig von ihrer Definition bis zum Ende der entsprechenden **.c**-Datei.
- Dieser Gültigkeitsbereich kann durch eine Deklaration mittels **extern** (siehe oben) auf den Bereich vor ihrer Definition bzw. auf andere **.c**-Dateien erweitert werden.

Gültigkeitsbereich lokaler Variablen

- Eine lokale Variable ist stets innerhalb eines Blocks **{ ... }** definiert; dieser Block kann entweder der **Funktionskörper** oder ein darin enthaltener Unterblock (z.B. von einer **if**- oder **for**-Anweisung) sein.
- Eine lokale Variable ist gültig von ihrer Definition bis zum Ende ihres zugehörigen Blocks.
- Wird der Block verlassen (z.B. die Funktion wird beendet, die **if**-Anweisung wurde ausgeführt, die **for**-Schleife wird verlassen), geht der Wert dieser Variable verloren; Ausnahmen bilden **static**-Variablen (siehe oben), deren Werte erhalten bleiben und bei der nächsten Ausführung der entsprechenden Blöcke (z.B. erneuter Aufruf der Funktion) wieder zur Verfügung stehen.
- Eine globale und lokale Variablen mit gleichem Namen sind erlaubt (Einschränkung: keine zwei lokalen Variablen gleichen Namens im gleichen Block); wird mit dem Variablennamen zugegriffen, gilt die lokale Variable bzw. bei mehreren lokalen Variablen die des innersten Blocks (die gültige Variable **verschattet** die weiter außen definierten gleichnamigen Variablen).

```
1. #include<stdio.h>
2.
3. int x = 3; // Globale Variable x.
4.
5. int main(void)
6. {
7.     int i1;
8.
9.     printf("1) x = %d\n", x);
10.
11.    int x = 5; // Lokale Variable x (verschattet von hier an die globale Variable x).
12.    printf("2) x = %d\n", x);
13.
14.    for(i1 = 0; i1 < 2; i1++)
15.    {
16.        printf("3) x = %d\n", x);
17.
18.        int x = 7; // Eine weitere lokale Variable x in einem Unterblock
19.                // (verschattet die beiden anderen Variablen x).
20.        printf("4) x = %d\n", x);
21.    } // Ende des Gültigkeitsbereichs der lokalen Variable "x = 7".
22.
23.    printf("5) x = %d\n", x);
24. } // Ende des Gültigkeitsbereichs der lokalen Variable "x = 5".
25.
26. // Ende des Gültigkeitsbereichs der globalen Variable x.
```

```
1) x = 3
2) x = 5
3) x = 5
4) x = 7
3) x = 5
4) x = 7
5) x = 5
```

Laufzeitverhalten, Iteration, Rekursion

- Beim wissenschaftlichen Programmieren bzw. in der Numerik ist es häufig erforderlich, dass ein Programm nicht nur korrekt arbeitet, sondern auch **effizient** ist, das heißt große Datensätze (z.B. viele Messergebnisse, große Matrizen, feine Diskretisierungen) in möglichst kurzer oder zumindest überschaubarer Zeit (**Tage, Wochen oder Monate sind oft noch o.k., nicht aber mehrere Jahre**) verarbeitet.
- Programme bzw. Algorithmen klassifiziert man nach ihrem **Laufzeitverhalten**, das im Wesentlichen der Anzahl der Anweisungen bzw. mathematischen Operationen entspricht und in Abhängigkeit von der Problemgröße N angegeben wird, z.B. $\mathcal{O}(N)$, $\mathcal{O}(N \log N)$, $\mathcal{O}(N^2)$, ...
- Konstante Faktoren werden beim Laufzeitverhalten in der Regel ignoriert (d.h. $\mathcal{O}(N) = \mathcal{O}(2N) = \mathcal{O}(5N)$, etc.), da z.B. ein $\mathcal{O}(N \log N)$ -Programm für hinreichend große N immer schneller als ein $\mathcal{O}(N^2)$ -Programm sein wird, unabhängig von irgendwelchen Faktoren.
- Beispiele:
 - Das Skalarprodukt zweier Vektoren \mathbf{x} und \mathbf{y} der Größe N zu berechnen hat typischer Weise das Laufzeitverhalten $\mathcal{O}(N)$.

```
1. ...
2. double sp = 0.0;
3.
4. for(i = 0; i < N; i++)
5.     sp += x[i]*y[i];
6. ...
```

- Eine Matrix \mathbf{A} der Größe $N \times N$ zu initialisieren hat typischer Weise das Laufzeitverhalten $\mathcal{O}(N^2)$.

```
1. ...
2. for(i = 0; i < N; i++)
3.     {
4.         for(j = 0; j < N; j++)
5.             A[i][j] = 0.0;
6.     }
7. ...
```

- Zwei Matrizen \mathbf{A} und \mathbf{B} der Größe $N \times N$ zu multiplizieren hat typischer Weise das Laufzeitverhalten $\mathcal{O}(N^3)$.

```
1. ...
2. for(i = 0; i < N; i++)
3.     {
4.         for(j = 0; j < N; j++)
5.             {
6.                 C[i][j] = 0.0;
7.
8.                 for(k = 0; k < N; k++)
9.                     C[i][j] += A[i][k]*B[k][j];
10.            }
11.     }
12. ...
```

- Das Laufzeitverhalten lässt sich also häufig direkt an den implementierten Schleifen ablesen ("Über welchen Bereich laufen deren Indizes?", "Wie tief sind diese Schleifen verschachtelt?"); die Verwendung von Schleifen bezeichnet man auch als **Iteration**.
- Eine weitere gängige Methode, umfangreiche Probleme zu lösen, ist **Rekursion**; eine Funktion oder ein Algorithmus ruft sich dabei selbst auf, wobei die Problemgröße bei jedem Aufruf reduziert wird.
- Das Laufzeitverhalten von rekursiven Algorithmen ist oft nicht so offensichtlich wie das von iterativen Algorithmen; es kommt dabei auf die **Rekursionstiefe** (die Anzahl der verschachtelten rekursiven Aufrufe) und die Anzahl der rekursiven Aufrufe pro Funktionsaufruf an.
- **Beispiel:** $N!$ iterativ und rekursiv, Laufzeitverhalten in beiden Fällen $\mathcal{O}(N)$...

```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Rekursiv:
6. // * Rekursionstiefe N
7. // * 1 rekursiver Aufruf pro Durchgang
8. // --> Laufzeitverhalten  $\mathcal{O}(N)$ 
```

```

9. int fac_recu(int N)
10. {
11.     if(N == 0)
12.         return 1;
13.
14.     return N * fac_recu(N-1);
15. }
16.
17. // *****
18.
19. // Iterativ:
20. // * 1 Loop der N Mal durchlaufen wird
21. // --> Laufzeitverhalten O(N)
22. int fac_iter(int N)
23. {
24.     int i;
25.
26.     int fac = 1;
27.
28.     for(i = 1; i <= N; i++)
29.         fac *= i;
30.
31.     return fac;
32. }
33.
34. // *****
35.
36. int main(void)
37. {
38.     printf("%d\n", fac_recu(5));
39.     printf("%d\n", fac_iter(5));
40. }

```

- Verschiedene Algorithmen für dasselbe Problem können unterschiedliches Laufzeitverhalten besitzen; ein Beispiel ist das im Folgenden beschriebene Sortieren eines N -elementigen Arrays mit **Bubblesort** (iterativ; Laufzeitverhalten $\mathcal{O}(N^2)$) und **Mergesort** (rekursiv; Laufzeitverhalten $\mathcal{O}(N \log N)$) ...

Bubblesort (iterativer Algorithmus)

- Zuerst werden alle N Elemente des Arrays durchlaufen und das Kleinste gesucht, dieses an den ersten Speicherplatz des Arrays gebracht ...
- ... dann werden die verbleibenden $N - 1$ Elemente des Arrays durchlaufen und das Zweitkleinste gesucht, dieses an den zweiten Speicherplatz des Arrays gebracht ...
- ... dann werden die verbleibenden $N - 2$ Elemente des Arrays durchlaufen und das Drittkleinste gesucht, dieses an den dritten Speicherplatz des Arrays gebracht ...
- ... USW. ...
- Laufzeitverhalten $\mathcal{O}(N^2)$.
- Umsetzung im Folgenden mit Hilfe von Iteration.

```

1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. // *****
5.
6. // Sortiert mit dem Bubblesort-Algorithmus ein double-Array v mit num Werten in
7. // aufsteigender Reihenfolge.
8. void bubblesort(int num, double v[])
9. {
10.     int i1, i2;
11.
12.     if(num < 0)
13.     {
14.         printf("Fehler bei void bubblesort(...\n");
15.         exit(0);
16.     }
17.
18.     for(i1 = 0; i1 < num-1; i1++)
19.         // Finde das richtige Element fuer Position i1 (das "i1-kleinste Element")
20.         // und verschiebe es an diese Position.

```

```

21.  {
22.    int index_smallest_element = i1;
23.
24.    for(i2 = i1+1; i2 < num; i2++)
25.        // Teste, ob v[i2] kleiner ist, als der aktuell beste Kandidat fuer das
26.        // kleinste Element.
27.        {
28.            if(v[i2] < v[index_smallest_element])
29.                index_smallest_element = i2;
30.        }
31.
32.        // Vertausche v[i1] und v[index_smallest_element].
33.        double tmp = v[i1];
34.        v[i1] = v[index_smallest_element];
35.        v[index_smallest_element] = tmp;
36.    }
37. }
38.
39. // *****
40.
41. int main(void)
42. {
43.     int i1;
44.
45.     // *****
46.
47.     // Generiere eine deterministische Sequenz von Zufallszahlen in [0.0:1.0].
48.
49.     const int N = 10;
50.     double a[N];
51.
52.     for(i1 = 0; i1 < N; i1++)
53.         // rand() generiert eine Integer-Zufallszahl zwischen 0 und RAND_MAX.
54.         a[i1] = ((double)rand() + 0.5) / ((double)RAND_MAX + 1.0);
55.
56.     // *****
57.
58.     printf("Unsortiert:\n");
59.
60.     for(i1 = 0; i1 < N; i1++)
61.         printf("a[%5d] = %.8lf\n", i1, a[i1]);
62.
63.     bubblesort(N, a);
64.
65.     printf("\nSortiert:\n");
66.
67.     for(i1 = 0; i1 < N; i1++)
68.         printf("a[%5d] = %.8lf\n", i1, a[i1]);
69. }

```

```

Unsortiert:
a[ 0] = 0.84018772
a[ 1] = 0.39438293
a[ 2] = 0.78309922
a[ 3] = 0.79844003
a[ 4] = 0.91164736
a[ 5] = 0.19755137
a[ 6] = 0.33522276
a[ 7] = 0.76822960
a[ 8] = 0.27777471
a[ 9] = 0.55396996

```

```

Sortiert:
a[ 0] = 0.19755137
a[ 1] = 0.27777471
a[ 2] = 0.33522276
a[ 3] = 0.39438293
a[ 4] = 0.55396996
a[ 5] = 0.76822960
a[ 6] = 0.78309922
a[ 7] = 0.79844003
a[ 8] = 0.84018772
a[ 9] = 0.91164736

```

Mergesort (rekursiver Algorithmus)

- Enthält das zu sortierende Array nur ein Element, ist es bereits sortiert, Mergesort kann beendet werden.
- Andernfalls wird das zu sortierende Array in zwei gleich große Teile gespalten und Mergesort wird für beide Teile rekursiv aufgerufen; die sortierten Teilarrays werden dann zu einem einzigen sortierten Array verschmolzen.
- Laufzeitverhalten $\mathcal{O}(N \log N)$ (Rekursionstiefe $\log_2 N$, $\mathcal{O}(N)$ Operationen auf jeder Rekursionsebene).

- Umsetzung im Folgenden, wie oben beschrieben, mit Hilfe von Rekursion.

```
1. #include<stdio.h>
2. #include<stdlib.h>
3.
4. // *****
5.
6. // Sortiert mit dem Mergesort-Algorithmus ein double-Array v mit num Werten in
7. // aufsteigender Reihenfolge.
8. void mergesort(int num, double v[])
9. {
10.  if(num < 0)
11.  {
12.      printf("Fehler bei void mergesort(...\n");
13.      exit(0);
14.  }
15.
16.  if(num == 0 || num == 1)
17.      // Liste der Laenge 0 oder 1 ist bereits sortiert.
18.      return;
19.
20.  // Linke Teilliste sortieren.
21.  double *vl = v;
22.  int numl = num/2;
23.  mergesort(numl, vl);
24.
25.  // Rechte Teilliste sortieren.
26.  double *vr = v+numl;
27.  int numr = num-numl;
28.  mergesort(numr, vr);
29.
30.  // Sortierte linke und rechte Teilliste zu sortierter Gesamtliste verschmelzen.
31.
32.  double *tmp;
33.
34.  if((tmp = (double *)malloc(num*sizeof(double))) == NULL)
35.  {
36.      printf("Fehler bei void mergesort(...\n");
37.      exit(0);
38.  }
39.
40.  int i;
41.  int il = 0;
42.  int ir = 0;
43.
44.  for(i = 0; i < num; i++)
45.  {
46.      if(ir == numr)
47.          // Nur noch Elemente in der linken Teilliste.
48.          // --> benutze aktuelles Element der Linken Teilliste.
49.          {
50.              tmp[i] = vl[il];
51.              il++;
52.          }
53.      else if(il == numl)
54.          // Nur noch Elemente in der rechten Teilliste
55.          // --> benutze aktuelles Element der rechten Teilliste.
56.          {
57.              tmp[i] = vr[ir];
58.              ir++;
59.          }
60.      else
61.          // Vergleiche das aktuelle Element der linken und der rechten Teilliste
62.          // --> benutze das kleinere Element.
63.          {
64.              if(vl[il] < vr[ir])
65.                  // --> benutze aktuelles Element der linken Teilliste.
66.                  {
67.                      tmp[i] = vl[il];
68.                      il++;
69.                  }
70.              else
71.                  // --> benutze aktuelles Element der rechten Teilliste.
```

```

72.     {
73.         tmp[i] = vr[ir];
74.         ir++;
75.     }
76.     }
77. }
78.
79. for(i = 0; i < num; i++)
80.     v[i] = tmp[i];
81.
82. free(tmp);
83. }
84.
85. // *****
86.
87. int main(void)
88. {
89.     int il;
90.
91.     // *****
92.
93.     // Generiere eine deterministische Sequenz von Zufallszahlen in [0.0:1.0].
94.
95.     const int N = 10;
96.     double a[N];
97.
98.     for(il = 0; il < N; il++)
99.         // rand() generiert eine Integer-Zufallszahl zwischen 0 und RAND_MAX.
100.        a[il] = ((double)rand() + 0.5) / ((double)RAND_MAX + 1.0);
101.
102.     // *****
103.
104.     printf("Unsortiert:\n");
105.
106.     for(il = 0; il < N; il++)
107.         printf("a[%5d] = %.8lf\n", il, a[il]);
108.
109.     mergesort(N, a);
110.
111.     printf("\nSortiert:\n");
112.
113.     for(il = 0; il < N; il++)
114.         printf("a[%5d] = %.8lf\n", il, a[il]);
115. }

```

```

Unsortiert:
a[ 0] = 0.84018772
a[ 1] = 0.39438293
a[ 2] = 0.78309922
a[ 3] = 0.79844003
a[ 4] = 0.91164736
a[ 5] = 0.19755137
a[ 6] = 0.33522276
a[ 7] = 0.76822960
a[ 8] = 0.27777471
a[ 9] = 0.55396996

```

```

Sortiert:
a[ 0] = 0.19755137
a[ 1] = 0.27777471
a[ 2] = 0.33522276
a[ 3] = 0.39438293
a[ 4] = 0.55396996
a[ 5] = 0.76822960
a[ 6] = 0.78309922
a[ 7] = 0.79844003
a[ 8] = 0.84018772
a[ 9] = 0.91164736

```

Vergleich der Effizienz von Bubblesort und Mergesort

- Mergesort, 100,000 Elemente (nur die letzten 3 Elemente werden am Bildschirm ausgegeben):

```

mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ time ./mergesort
Unsortiert:
a[99997] = 0.84516746
a[99998] = 0.66494025
a[99999] = 0.30764298

Sortiert:
a[99997] = 0.99996763
a[99998] = 0.99999190
a[99999] = 0.99999357

real    0m0.053s
user    0m0.052s
sys     0m0.000s

```

- Bubblesort, 100,000 Elemente (nur die letzten 3 Elemente werden am Bildschirm ausgegeben):

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ time ./bubblesort
Unsoriert:
a[99997] = 0.84516746
a[99998] = 0.66494025
a[99999] = 0.30764298

Sortiert:
a[99997] = 0.99996763
a[99998] = 0.9999190
a[99999] = 0.9999357

real    0m25.032s
user    0m25.002s
sys     0m0.008s
```

- Mergesort, 1,000,000 Elemente (nur die letzten 3 Elemente werden am Bildschirm ausgegeben):

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ time ./mergesort
Unsoriert:
a[999997] = 0.29797783
a[999998] = 0.94337596
a[999999] = 0.19993533

Sortiert:
a[999997] = 0.99999686
a[999998] = 0.99999691
a[999999] = 0.99999831

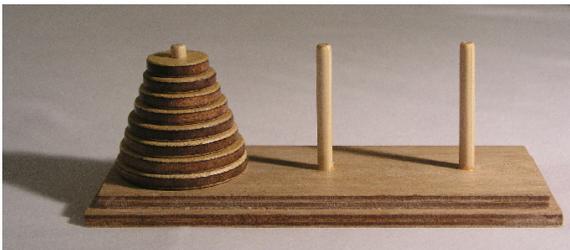
real    0m0.512s
user    0m0.484s
sys     0m0.020s
```

- Bubblesort, 1,000,000 Elemente (nur die letzten 3 Elemente werden am Bildschirm ausgegeben):

```
mwagner@laptop-tigger:~/Lecture_ProgPhys/slides/tmp$ time ./bubblesort
Unsoriert:
a[999997] = 0.29797783
a[999998] = 0.94337596
a[999999] = 0.19993533
```

Die Türme von Hanoi ... rekursiv (eine Perle der Informatik)

- Siehe http://de.wikipedia.org/wiki/Türme_von_Hanoi.



```
1. #include<stdio.h>
2.
3. // *****
4.
5. // Bewege einen Turm bestehend aus Scheibe "1" bis Scheibe "num_discs" von
6. // Stange src nach Stange dst (leer bzw. nur Scheiben groesser als Scheibe
7. // "num_discs" liegen dort), wobei Stange tmp (leer bzw. nur Scheiben groesser
8. // als Scheibe "num_discs" liegen dort) als Zwischenlager dienen kann.
9. void hanoi(int num_discs, int src, int dst, int tmp)
10. {
11.     static int ctr = 0;
12.
13.     if(num_discs == 0)
14.         // Ein Turm mit 0 Scheiben, d.h. es ist nichts zu tun.
15.         return;
16.
17.     // Bewege alle bis auf die unterste Scheibe auf Stange tmp.
18.     hanoi(num_discs-1, src, tmp, dst);
19.
20.     // Bewege die unterste Scheibe auf Stange dst.
21.     ctr++;
22.     printf("Zug %3d: Scheibe %d von Stange %d nach Stange %d ...\\n", ctr, num_discs, src, dst);
23.
24.     // Bewege den Turm auf Stange tmp auf Stange dst.
25.     hanoi(num_discs-1, tmp, dst, src);
26. }
27.
28. // *****
29.
30. int main(void)
31. {
```

```
32. hanoi(3, 1, 3, 2);
33. printf("... fertig :-)\n");
34. }
```

```
Zug 1: Scheibe 1 von Stange 1 nach Stange 3 ...
Zug 2: Scheibe 2 von Stange 1 nach Stange 2 ...
Zug 3: Scheibe 1 von Stange 3 nach Stange 2 ...
Zug 4: Scheibe 3 von Stange 1 nach Stange 3 ...
Zug 5: Scheibe 1 von Stange 2 nach Stange 1 ...
Zug 6: Scheibe 2 von Stange 2 nach Stange 3 ...
Zug 7: Scheibe 1 von Stange 1 nach Stange 3 ...
... fertig :-) !
```

```
31. ...
32. hanoi(9, 1, 3, 2);
33. ...
```

```
Zug 1: Scheibe 1 von Stange 1 nach Stange 3 ...
Zug 2: Scheibe 2 von Stange 1 nach Stange 2 ...
Zug 3: Scheibe 1 von Stange 3 nach Stange 2 ...
Zug 4: Scheibe 3 von Stange 1 nach Stange 3 ...
Zug 5: Scheibe 1 von Stange 2 nach Stange 1 ...
Zug 6: Scheibe 2 von Stange 2 nach Stange 3 ...
Zug 7: Scheibe 1 von Stange 1 nach Stange 3 ...
Zug 8: Scheibe 4 von Stange 1 nach Stange 2 ...
Zug 9: Scheibe 1 von Stange 3 nach Stange 2 ...
Zug 10: Scheibe 2 von Stange 3 nach Stange 1 ...
...
Zug 502: Scheibe 2 von Stange 3 nach Stange 1 ...
Zug 503: Scheibe 1 von Stange 2 nach Stange 1 ...
Zug 504: Scheibe 4 von Stange 2 nach Stange 3 ...
Zug 505: Scheibe 1 von Stange 1 nach Stange 3 ...
Zug 506: Scheibe 2 von Stange 1 nach Stange 2 ...
Zug 507: Scheibe 1 von Stange 3 nach Stange 2 ...
Zug 508: Scheibe 3 von Stange 1 nach Stange 3 ...
Zug 509: Scheibe 1 von Stange 2 nach Stange 1 ...
Zug 510: Scheibe 2 von Stange 2 nach Stange 3 ...
Zug 511: Scheibe 1 von Stange 1 nach Stange 3 ...
... fertig :-) !
```

Präprozessoranweisungen

- Der **C**-Präprozessor führt vor dem eigentlichen Kompilieren textuelle Ersetzungen im Programmcode aus:
 1. **Präprozessor**: textuelle Ersetzungen im Programmcode.
 2. **Compiler**: Übersetzung von Programmcode in Maschinencode (liefert Objektdateien), d.h. in direkt vom Computer ausführbare Anweisungen.
 3. **Linker**: Verbinden mehrerer Objektdateien zu einem ausführbaren Programm.

#include

- Siehe oben.

```
1. // Der Praeprozessor ersetzt die #include-Anweisung durch die Datei stdio.h.
2. #include<stdio.h>
3.
4. int main(void)
5. {
6.     printf("abc\n");
7. }
```

#define

- Siehe oben.

```
1. #include<stdio.h>
2.
3. // #define ersetzt vor dem Kompilieren PI, PRINT_ABC, TEXT durch den jeweils folgenden Text.
4. #define PI 3.1415
5. #define PRINT_ABC printf("abc\n");
6. #define TEXT "def ghi jkl\n"
7.
8. int main(void)
9. {
10.     printf("%f\n", PI);
11.     PRINT_ABC
12.     printf(TEXT);
13. }
```

```
3.141500
abc
def ghi jkl
```

Bedingte Übersetzung

- Mit Präprozessoranweisungen wie **#ifdef**, **#else** und **#endif** kann kontrolliert werden, welche Teile des Programmcodes kompiliert werden.
- **Beispiel**: Ausgabe zusätzlicher Informationen während des Debuggens ...

```
1. #include<stdio.h>
2.
3. #define __DEBUG__
4.
5. // *****
6.
7. int fac_recu(int N)
8. {
9.     #ifdef __DEBUG__
10.     printf("Aufruf von int fac_recu(int N) mit N = %d ...\n", N);
11.     #endif
12.
13.     if(N == 0)
14.         return 1;
15.
16.     return N * fac_recu(N-1);
17. }
18.
19. // *****
20.
21. int main(void)
22. {
```

```
23. printf("%d\n", fac_recu(5));
24. }
```

```
Aufruf von int fac_recu(int N) mit N = 5 ...
Aufruf von int fac_recu(int N) mit N = 4 ...
Aufruf von int fac_recu(int N) mit N = 3 ...
Aufruf von int fac_recu(int N) mit N = 2 ...
Aufruf von int fac_recu(int N) mit N = 1 ...
Aufruf von int fac_recu(int N) mit N = 0 ...
120
```

```
2. ...
3. // #define __DEBUG__
4. ...
```

120

• Beispiel: Nicht-relativistische und relativistische Version eines Programms ...

```
1. #include<math.h>
2. #include<stdio.h>
3.
4. // #define __RELATIVISTIC__
5.
6. // *****
7.
8. // Berechnet die Energie eines Teilchens mit Masse m und Geschwindigkeit v.
9. double energy(double m, double v)
10. {
11. #ifdef __RELATIVISTIC__
12.     double gamma = 1.0 / sqrt(1.0 - pow(v, 2.0));
13.     double p = m * gamma * v;
14.     return sqrt(pow(m, 2.0) + pow(p, 2.0));
15. #else
16.     return m + 0.5 * m * pow(v, 2.0);
17. #endif
18. }
19.
20. // *****
21.
22. int main(void)
23. {
24.     const double m = 938.3; // Masse in MeV/c^2.
25.     const double v1 = 0.05; // Geschwindigkeit in c.
26.     const double v2 = 0.50; // Geschwindigkeit in c.
27.
28.     printf("E = %6.1f (in MeV).\n", energy(m, v1));
29.     printf("E = %6.1f (in MeV).\n", energy(m, v2));
30. }
```

```
E = 939.5 (in MeV).
E = 1055.6 (in MeV).
```

```
3. ...
4. #define __RELATIVISTIC__
5. ...
```

```
E = 939.5 (in MeV).
E = 1083.5 (in MeV).
```

Makros

- Definitionen mit Parametern bezeichnet man als **Makros**.
- Makros ähneln Funktionen; es kommt jedoch zu keinem Funktionsaufruf; der Präprozessor führt lediglich textuelle Ersetzungen aus; damit kein "call by value".

```
1. #include<stdio.h>
2.
3. // Ein Makro, das die Werte zweier int-Variablen vertauscht.
4. #define SWAP_INT(a,b) { int tmp = a; a = b; b = tmp; }
5.
6. int main(void)
7. {
8.     int x = 3;
9.     int y = 5;
10.    printf("x = %d, y = %d\n", x, y);
11.    SWAP_INT(x,y); // Diese Zeile wird ersetzt durch { int tmp = x; x = y; y = tmp; }.
12.    printf("x = %d, y = %d\n", x, y);
13. }
```

$x = 3, y = 5$
 $x = 5, y = 3$