

# Einführung in die Programmierung mit C

## Teil 2

Dieter Kranzlmüller

Nils gentschen Felde

Heute: Felix Mößbauer



- Modularisierung
- Headerdateien
- Parameterübergabe
- Strukturen
- Rekursive und iterative Funktionen
- Prozesse
- Threads

- Deklaration: führt einen oder mehrere **Bezeichner** ein oder wiederholt sie
- Definition: Eine Definition ist eine Deklaration, die mindestens eine dieser Bedingungen erfüllt:
  - **Reservieren von Speicherplatz** für das deklarierte Objekt wird veranlasst
  - Im Falle einer **Funktion** wird ihr **Rumpf** angegeben
- Beispiele: Deklaration oder Definition?

<code>int a = 3;</code>	Definition + Initialisierung
<code>int b;</code>	Definition
<code>extern int c;</code>	Deklaration (extern: „declare without defining“)
<code>static int x;</code>	Deklaration + Initialisierung (vom Beginn bis zum Ende des Programmes ist Speicher für x reserviert)
<code>int f(void);</code>	Deklaration (in Headerfile)

- Grundbegriffe
  - Bindung
    - Keine Bindung: **Block-lokale** Variablen
    - Interne Bindung: Variablen lokal zu einer **Übersetzungseinheit (c-File)**
    - Externe Bindung: Objekte, die in **verschiedenen Modulen** benutzt werden können
  - Speicherdauer
    - Statisch: von **Beginn bis Ende** des Programmlaufs
    - Dynamisch: Anlegen und Zerstören mit **speziellen Befehlen**
    - Automatisch: vom Eintritt in bis Austritt aus dem **Sichtbarkeitsbereich**
  - Sichtbarkeit
    - Lokal: Bezeichner ist nur in einem bestimmten Bereich gültig
    - Global: Bezeichner ist überall gültig

- Modularisierung

- Aufteilung des Gesamtsystems (Programms) in mehrere Teilsysteme (Teilprogramme/Module)
- Teilprogramme sollen **weitgehend unabhängig** voneinander sein
- Jedes Teilprogramm/Modul erhält eine **Schnittstelle**
  - Schnittstelle beschreibt, **was** das Modul macht, aber nicht, **wie**
  - Schnittstellen vereinfachen das Zusammensetzen von Modulen und die Kommunikation zwischen Modulen
- Module können zu **Bibliotheken** zusammengesetzt (gebunden/gelinkt) werden:
  - Statisches Binden/Linken
  - Dynamisches Binden/Linken

- Was ist eine Headerdatei?
  - Beschreibung der **Schnittstelle** zu einem Modul
  - Modul kann Teil einer Bibliothek sein
- Was kann eine Headerdatei enthalten?
  - Variablen- und Funktionsdeklarationen
  - Makro-Definitionen
  - Konstanten-Definitionen
  - Typdefinitionen
  - Aufzählungen
  - Namensdeklarationen
  - `#include`-Anweisungen
  - Direktiven für bedingte Kompilierung
  - Kommentare (Dokumentation der Schnittstelle)

- Einbindung mittels Präprozessoranweisung
  - `#include`
- Anwendungsbeispiele
  - `#include "/baum/ast/dateiname.h"` Suche im Dateisystem
  - `#include <dateiname.h>` Suche in den Verzeichnissen des Compilers
- Beispiel Header Dateien der Standard Bibliotheken
  - `stdio.h` Standard Ein-/Ausgabe
  - `math.h` Mathematische Funktionen
  - `stdlib.h` Speicherverwaltung und Prozeßsteuerung
  - `time.h` Datum und Zeit
  - `fcntl.h` Datei Ein-/Ausgabe

# • Beispiel: Eigene Headerdateien

- geometrie.h:

```
#ifndef geometrie
#define geometrie

#define PI (3.1415926)

float quadrat(float x);
float kreisflaeche (float radius);
float kugelvolumen (float radius);

#endif
```

## “Include Guards”:

```
#ifndef name
#define name

#endif
```

**Doppeltes “Inkludieren” verhindern**

- geometrie.c:

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

#include "geometrie.h"

float quadrat(float x) {
    return x * x;
}

float kreisflaeche(float radius) {
    return PI * quadrat(radius);
}

float kugelvolumen(float radius) {
    return (4/3) * PI *
        quadrat(radius) * radius;
}
```



- Beispiel: Eigene Headerdateien (Forts.)
  - main.c:

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

#include "geometrie.h"

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Aufruf: start <radius>\n\n");
        return EXIT_FAILURE;
    }

    float radius = atof(argv[1]);
    printf("Kreisfläche = %f.\n", kreisflaeche(radius));
    printf("Kugelvolumen = %f.\n", kugelvolumen(radius));
    return EXIT_SUCCESS;
}
```

- Beispiel: Bibliotheken (libraries)

- geometrie.h, geometrie.c, main.c unverändert
- Zusätzlich algebra.h:

```
#ifndef algebra
#define algebra

#define E (2.7182818)

float mittelwert(float a, float b);

#endif
```

algebra.c:

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

#include "algebra.h"

float mittelwert(float a, float b) {
    return (a+b) / 2;
}
```

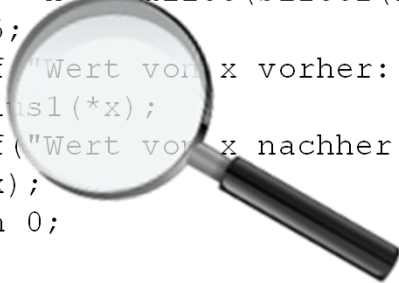
- Idee: Bibliothek für Geometrie und Algebra → mymath.a
  - Schritt 1: Objektdaten erstellen  
gcc -c algebra.c geometrie.c
  - Schritt 2: Objektdaten zu Bibliothek linken  
ar -q mymath.a algebra.o geometrie.o
  - Schritt 3: Übersetzen unter Verwendung der Bibliothek  
gcc -o start main.c mymath.a

- Arten der Parameterübergabe bei Funktionsaufrufen
  - Wertübergabe (call by value)
  - Adressübergabe (call by reference)
- Beispiel: Call by reference
  - cbr.c:

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

int plus1(int *x) {
    *x = *x + 1;
    return *x;
}

int main(void) {
    int a, *x = malloc(sizeof(int));
    *x = 5;
    printf("Wert von x vorher: %i\n", *x);
    a = plus1(*x);
    printf("Wert von x nachher: %i\n", *x);
    free(x);
    return 0;
}
```



```
gcc -o test cbr.c
```

verursacht folgende Warnung:

```
cbr.c:13: warning: passing arg 1
of `plus1' makes pointer from
integer without a cast
```

```
./test
```

liefert:

```
Wert von x vorher: 5
Segmentation fault
```

`a = plus1(*x);`

- Arten der Parameterübergabe bei Funktionsaufrufen
  - Wertübergabe (call by value)
  - Adressübergabe (call by reference)
- Beispiel: Call by reference
  - cbr.c:

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

int plus1(int *x) {
    *x = *x + 1;
    return *x;
}

int main(void) {
    int a, *x = malloc(sizeof(int));
    *x = 5;
    printf("Wert von x vorher: %i\n", *x);
    a = plus1(x);
    printf("Wert von x nachher: %i\n", *x);
    free(x);
    return 0;
}
```

```
gcc -o test cbr.c
```

```
./test
```

liefert:

```
Wert von x vorher: 5
```

```
Wert von x nachher: 6
```

# • Strukturen in C

- Eigenschaften:

➔ Kapitel 15

- Feste Länge (vgl. Array)
- Kein eindeutiger Datentyp → Elemente einer Struktur können **unterschiedlichen Typs** sein
- Strukturen können selbst als eine "Art Datentyp" aufgefasst werden
- Häufig: Deklaration von Strukturen in Headerdateien

- Beispiel:

```
struct complex{  
    double re;  
    double im;  
};  
struct complex z;
```

Oft verwendet mit typedef:

```
typedef struct{  
    double re;  
    double im;  
}complex;  
complex z;
```

Erstpart das Schreiben von „struct“

- Beispiel: Definition und Zugriff auf eine Struktur

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

struct myStructure {
    int value1;
    float value2;
    char text[10];
};

int main(int argc, char *argv[]) {
    struct myStructure s1;

    s1.value1 = 3;
    s1.value2 = 0.77;
    strcpy(s1.text, "abc");

    printf("%i, %f, %s\n", s1.value1, s1.value2, s1.text);

    return EXIT_SUCCESS;
}
```

- Iterative Implementierung einer Funktion zur Berechnung der x-ten Fibonacci-Zahl

```
long fib_it(long x) {  
  
    long f0, f1, tmp, i;  
    f0 = 0;  
    f1 = 1;  
  
    if (x <= 1)  
        return x;  
  
    for(i=2; i<=x; ++i) {  
        tmp = f1;  
        f1 = f0 + f1;  
        f0 = tmp;  
    }  
  
    return f1;  
}
```

Definition:

Bildungsgesetz der Fibonacci-Folge  $f_0, f_1, f_2, \dots$

$$f_n = f_{n-1} + f_{n-2} \text{ für } n \geq 2$$

$$f_0 = 0, f_1 = 1$$

- Beispiel: Fibonacci-Zahlen
  - Fibonacci-Folge:  $f_n = f_{n-1} + f_{n-2}$  für  $n > 1$   
mit den Anfangswerten  $f_0 = 0, f_1 = 1$
  - Fibonacci-Zahlen: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- Rekursive Implementierung einer Funktion zur Berechnung der x-ten Fibonacci-Zahl

```
long fib_rec(long x) {  
    if (x <= 1)  
        return x;  
    else  
        return fib_rec(x-1) + fib_rec(x-2);  
}
```



- Mögliche Definitionen für "Prozess"
  - Ein Prozess ist ein in Ausführung befindliches Programm.
  - Ein Prozess ist Eigentümer von Ressourcen/Betriebsmitteln.
  - Ein Prozess ist eine Verwaltungseinheit des Betriebssystems, die ein Programm so ausführt, als ob der Prozessor nur diesem Programm zur Verfügung stünde.
- Programm
  - Vom Linker erzeugt,
  - als ausführbare Datei abgelegt,
  - kann durch Systemaufruf `execve ( )` zur Ausführung veranlasst werden.

## ➤ Kapitel 7

- Prozesshierarchie
  - Unix: Prozess-Baum (logische Prozesshierarchie)
  - Initialer Prozess (Wurzel): init
  - Jeder Prozess (außer init) hat einen Elternprozess (Parent)
  - Ein Prozess kann einen oder mehrere Kindprozesse (Child) erzeugen
- Waisen und Zombies
  - Waise:
    - Prozess in Ausführung, aber Vaterprozess bereits terminiert
    - Folge: Waisen-Prozess wird vom init-Prozess "adoptiert"
  - Zombie:
    - Prozess eigentlich beendet, aber Terminierungs-Signal (Rückgabewert) vom Vaterprozess nicht angenommen/abgerufen
    - Folge: Prozess als Zombie weiterhin in der Prozessliste

- Eigenschaften von Prozessen
  - Prozesse können verschiedene Zustände einnehmen
  - Prozesse können **nebenläufig** sein, d.h.
    - sie rechnen (pseudo-)parallel und
    - sie sind **abhängig** voneinander (z.B. Erzeuger/Verbraucher-Szenarien).
  - Problem der Nebenläufigkeit
    - Es kann kritische Bereiche geben (z.B. gemeinsam genutzte Variablen/Speicherbereiche)
    - Synchronisation der Prozesse erforderlich

- Prozesskontrollblock (PCB)

- enthält alle Informationen, die zur Beschreibung des aktuellen Ausführungsstatus eines Prozesses benötigt werden
- Arten von Informationen im PCB:
  - Prozessidentifikations-Informationen
    - Prozess-ID (PID)
    - Elternprozess-ID (PPID)
    - ID des Eigentümers des Prozesses (UID)
  - Prozesszustandsinformationen:
    - Prozesskontext: Inhalte aller CPU-Register, ...
    - Programm-Statuswort (PSW): Inhalte aller Statusregister (PC, SP, ...)
  - Prozesskontrollinformationen:
    - Scheduling-/Zustandsinformationen: aktueller Zustand, Priorität
    - Datenstrukturen: Referenz auf nächsten Proz. in Scheduling Queue
    - Signale/Nachrichten (zur Interprozesskommunikation)

- Informationen über Prozesse

- C: Funktionen `getpid()`, `getppid()`, `getuid()`, ...  
(Process-ID, Parent-Process-ID, User-ID)
- Shell: Befehle `ps`, `top`

- Erzeugen von (Kind-)Prozessen

- C-Befehl `fork()` in `<unistd.h>`
- C-Datentyp für Prozess-IDs: `pid_t` in `<sys/types.h>`
- Funktionssignatur für `fork()`: `pid_t fork(void);`
  - Rückgabe **an aufrufenden Prozess** (Parent): Prozess-ID des erzeugten Kindprozesses,  $> 0$
  - Rückgabe **an erzeugten Prozess** (Child):  $0$
  - Rückgabewert  $< 0$  im Fehlerfall

➤ Kapitel 7.8

- Erzeugen von (Kind-)Prozessen (Forts.)
  - Was macht `fork()` genau?
    - Anlegen einer **identischen Kopie** des Elternprozesses
    - Parent und Child haben anfangs den gleichen (aber nicht den selben) PCB, d.h.
      - Kindprozess erbt PCB von Elternprozess
      - Ausnahme: PID und PPID sowie bestimmte Locks, Signale und Zeitwerte
    - Parent und Child sind nach `fork()` prinzipiell **vollkommen eigenständige** Prozesse
  - Elternprozesse können mittels `wait()` bzw. `waitpid()` gezwungen werden, vor ihrer Terminierung auf Kindprozesse zu warten (erfordert `#include <sys/wait.h>`)

- Beispiel 1: Prozesserzeugung mit `fork( )`
  - Wie oft wird "Fork-Test" ausgegeben?

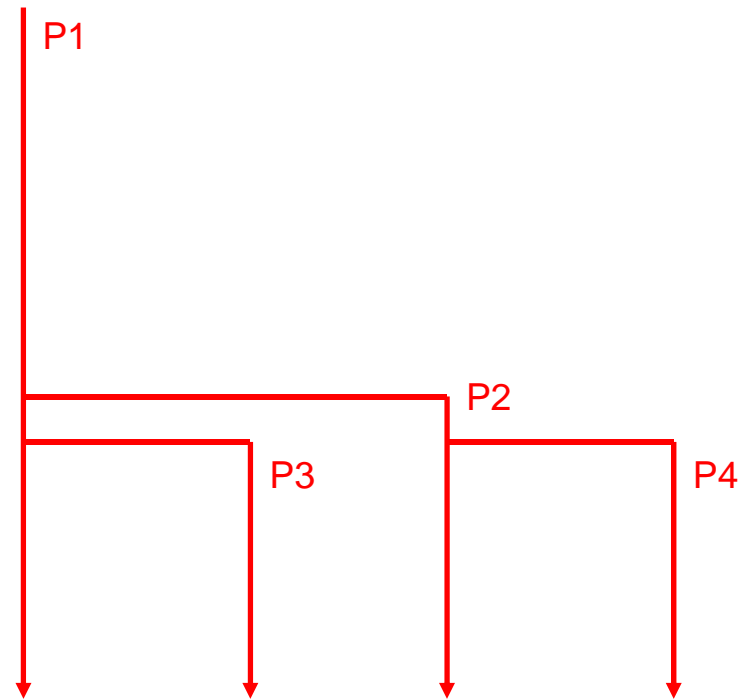
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main (void) {
    pid_t pid;

    pid = fork();
    pid = fork();

    printf("Fork-Test\n");

    return EXIT_SUCCESS;
}
```



- Beispiel 2: Prozesserzeugung mit `fork()`
  - Fallunterscheidung nach `fork()`-Rückgabewerten

```
#include <...>

int main (void) {
    int x = 5;
    pid_t pid;

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Fehler bei fork().\n");
    } else if (pid == 0) {
        /* Kindprozess */
        printf("Im Kindprozess\n");
        printf("Meine PID = %i\n", getpid());
        printf("Meine Parent-PID = %i\n", getppid());
        printf("Addiere 1: x + 1 = %i\n\n", ++x);
    } else {
        /* Elternprozess */
        printf("Im Elternprozess\n");
        printf("Meine PID = %i\n", getpid());
        printf("Meine Parent-PID = %i\n", getppid());
        printf("Subtrahiere 1: x - 1 = %i\n", --x);
    }
    return EXIT_SUCCESS;
}
```



- Beispiel 3: Prozesserzeugung mit `fork()`
  - Verwendung einer `switch`-Kontrollstruktur:

```
#include <...>

int main (void) {
    pid_t pid;
    switch (pid = fork ()) {
        case -1:
            fprintf(stderr, "Fehler bei fork().\n");
            break;
        case 0:
            /* Kindprozess */
            printf("Im Kindprozess\n");
            printf("Meine PID = %i\n", getpid());
            printf("Meine Parent-PID = %i\n", getppid());
            break;
        default:
            /* Elternprozess */
            printf("Im Elternprozess\n");
            printf("Meine PID = %i\n", getpid());
            printf("Meine Parent-PID = %i\n", getppid());
            break;
    }
    return EXIT_SUCCESS;
}
```

- Die `exec*`-Familie

- Jedes `exec*`-Kommando dient zur Ersetzung eines Prozess-Images durch ein anderes

- Varianten:

`execve()`, `execl()`, `execv()`, `execle()`, `execvp()`, `execlp()`

- Funktionssignaturen: siehe man-Pages

- Rückgabe jeder `exec*()`-Funktion:

- Fehlerfall: -1

- Sonst: Keine Rückkehr zum Aufrufer

➤ Kapitel 7.10

- Anwendungsbeispiele:

- Ausführung eines Konsolenkommandos oder Programms

- Erzeugen eines Kindprozesses mit `fork()`, dann mit `exec*()` durch ein anderes Programm ersetzen

- Beispiel: Überlagerung von Kindprozessen
  - printargs.c:

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

int main(int argc, char **argv) {

    int i;
    for (i=1; i < argc; i++) {
        printf("%i. Argument: %s\n", i, argv[i]);
    }
    return EXIT_SUCCESS;
}
```

- Beispiel: Überlagerung von Kindprozessen
  - exec.c:

```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>

int main (void) {
    pid_t pid;
    switch (pid = fork ()) {
        case -1:
            perror("Fehler bei fork()\n");
            return EXIT_FAILURE;
        case 0:
            execlp("./printargs", "./printargs", "a", "b", "c", NULL);
            break;
        default:
            if (waitpid(pid, NULL, 0) != pid) {
                perror ("Fehler beim Warten auf Kindprozess\n");
                return EXIT_FAILURE;
            }
    }
    return EXIT_SUCCESS;
}
```

- Definition: Thread
  - Unabhängige Befehlsfolge innerhalb eines Prozesses
  - Die Threads eines Prozesses haben gemeinsamen Adressraum
- Thread vs. Prozess
  - Prozess besteht aus mindestens einen Thread (main thread)
  - Duplizieren eines Prozesses => Duplizieren des Namensraums
  - Erstellung von Threads innerhalb eines Prozesses im selben Namensraum
  - Threads teilen sich alle globalen Variablen, Filedeskriptoren, Pipes, die außerhalb der Threads erzeugt werden

➞ Kapitel 26.3

➤ Kapitel 10

- Bibliothek pthread – Posix Threads

```
#include <pthread.h>
```

- Thread erzeugen

```
int pthread_create (pthread_t *thread,  
                   const pthread_attr_t *attribute,  
                   void *(*funktion)(void *), void *argumente);
```

- Thread beenden

```
void pthread_exit (void * wert);
```

- Warten auf Thread-Ende

```
int pthread_join (pthread_t thread, void **thread_return);
```

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      5

void *PrintHello(void *threadid){
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me,
           thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main (int argc, char *argv[]){
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
```

```
for(t=0; t<NUM_THREADS; t++){
    printf("In main: creating thread
           %ld\n", t);
    rc = pthread_create(&threads[t], NULL,
                       PrintHello, (void *)t);
    if (rc){
        printf("ERROR; return code from
               pthread_create() is %d\n",
               rc);
        exit(-1);
    }
}

/* Last thing that main() should do */
pthread_exit(NULL);
}
```

Source: <https://computing.llnl.gov/tutorials/pthreads/> (Lawrence Livermore National Lab)

- Weiterführende Informationen zu Threads z.B.:
  - pthreads Tutorial
  - vom Lawrence Livermore National Lab
  - <https://computing.llnl.gov/tutorials/pthreads/>