Die Programmiersprache C

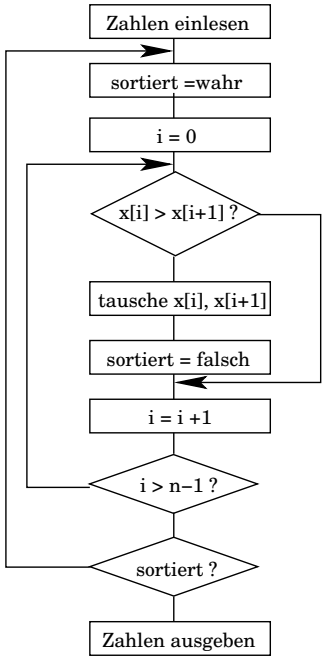
Strukturierte/prozedurale Programmierung

Unstrukturierte Programmierung

Früher verwendete man **Sprünge** in Programmen und **Flussdiagramme** zur Darstellung des Ablaufs.

Nachteil: Sprünge an jede beliebige Stelle erlaubt → **Spaghetti-Code**

In der Folge: Beschränken auf wenige Kontrollstrukturen, Top-Down Entwicklung, verwenden von **Prozeduren** als Strukturmittel.



Prozedurale Programmierung

Was ist das?

- Funktionen und Prozeduren werden dazu benutzt, Programme zu organisieren. (`strToInt`, `getHostByName`, ...)
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken (Libraries) zur Verfügung gestellt. (`sqrt`, `sin`, `printf`, `scanf`, `malloc`, ...)
- Programmiersprachen unterstützen diesen Stil, indem sie Techniken für die Argumentübergabe an Funktionen und das Liefern von Funktionswerten bereitstellen.
- **Beispiele:** Algol68, FORTRAN, Pascal und C
  Gegenbeispiele: Assembler, Cobol, Basic

Prozedurale Programmierung (2)

Ziele:

- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code **nicht durch Cut&Paste**, sondern durch allgemeingültige Funktionen.

Probleme:

- **Typisierung:** InsertionSort nur für `int`-Werte?
- **allgemeine Datentypen:** Vergleichsoperatoren?
- **Funktionalität:** was soll passieren, wenn der Benutzer auf eine Schaltfläche (Button) klickt?

Prozedurale Programmierung (3)

Rationale Zahlen:

- öffentliche Operationen:
  * Addition, Subtraktion, Multiplikation, Division
  * Vergleiche: kleiner, größer, gleich
  * Ausgabe
- private Operation: Kürzen (mittels ggT)

Die Implementierung ist im Anwendungsprogramm nicht wichtig, wohl aber die Deklaration der Funktionen.
⇒ aufteilen in die Dateien `rational.h` und `rational.c`

Rationale Zahlen

Datei `rational.h`:

```
typedef struct {
    long int numerator, denominator;
} rational_t;

char *toString(rational_t, char *);
int isEqual(rational_t, rational_t);
int isLess(rational_t, rational_t);

rational_t addR(rational_t, rational_t);
rational_t subR(rational_t, rational_t);
rational_t mulR(rational_t, rational_t);
rational_t divR(rational_t, rational_t);
```


Rationale Zahlen (2)

Datei `rational.c`:

```
#include "rational.h"
#include <stdio.h>

char *toString(rational_t r, char *string) {
    sprintf(string, "[%ld/%ld]",
            r.numerator, r.denominator);
    return string;
}
```


Rationale Zahlen (3)

```
void kuerzen(rational_t *rat) {
    long int a = rat->numerator;
    long int b = rat->denominator;

    if (a < 0) a = -a;
    if (b < 0) b = -b;
    while (b > 0) {      /* ggT(a, b) berechnen */
        long int r = a % b;
        a = b;
        b = r;
    }

    rat->numerator /= a;
    rat->denominator /= a;
}
```

Rationale Zahlen (4)

```
rational_t addR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator
                + a.denominator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (5)

```
rational_t subR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator
                - a.denominator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (6)

```
rational_t mulR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.numerator;
    r.denominator = a.denominator * b.denominator;

    kuerzen(&r);
    return r;
}
```


Rationale Zahlen (7)

```
rational_t divR(rational_t a, rational_t b) {
    rational_t r;

    r.numerator = a.numerator * b.denominator;
    r.denominator = a.denominator * b.numerator;

    kuerzen(&r);
    return r;
}
```

Rationale Zahlen (8)

```
int isLess(rational_t a, rational_t b) {
    rational_t r = sub(a, b);

    if ((r.numerator < 0 && r.denominator > 0)
            || (r.numerator > 0 && r.denominator < 0))
        return 1;
    return 0;
}
int isEqual(rational_t a, rational_t b) {
    rational_t r = sub(a, b);

    if (r.numerator == 0)
        return 1;
    return 0;
}
```


Rationale Zahlen (9)

Datei `main.c`:

```
#include "rational.h"
#include <stdio.h>

void main(void) {
    int i;
    char s1[40], s2[40], s3[40];
    rational_t a[] = {{5, 7}, {5, 6}, {3, 7}};
    rational_t b[] = {{3, 4}, {4, 15}, {6, 14}};
    rational_t c;
```


Rationale Zahlen (10)

```
    for (i = 0; i < 3; i++) {
        c = addR(a[i], b[i]);
        printf("%s + %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));

        c = mulR(a[i], b[i]);
        printf("%s * %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));

        if (isEqual(a[i], b[i]))
            printf("a[%d] == b[%d]\n", i, i);
        else printf("a[%d] != b[%d]\n", i, i);
    }
}
```


Rationale Zahlen (11)

Zugriff auf interne Variablen verhindern:

- in `rational.h` die Strukturvereinbarung entfernen und durch einen unvollständigen Typen ersetzen
- in `rational.c` die aus `rational.h` entfernte Strukturvereinbarung aufnehmen
- in `main.c` die Variablen des unvollständigen Typen durch Zeiger auf `rational_t` ersetzen
- die Parameter- und Rückgabetypen der Funktionen in `rational.h` und `rational.c` als Zeiger auf `rational_t` vereinbaren
- Funktion `create` zum Erzeugen von rationalen Zahlen hinzufügen

Rationale Zahlen überarbeitet

rational.h

```
#ifndef _RATIONAL_H
#define _RATIONAL_H

typedef struct rational rational_t;  /* incomplete */

rational_t *create(long int, long int);
char *toString(rational_t *, char *);
int isEqual(rational_t *, rational_t *);
int isLess(rational_t *, rational_t *);

void addR(rational_t *, rational_t *, rational_t *);
void subR(rational_t *, rational_t *, rational_t *);
void mulR(rational_t *, rational_t *, rational_t *);
void divR(rational_t *, rational_t *, rational_t *);
#endif
```


Rationale Zahlen überarbeitet (2)

rational.c

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

/***** private *****/
typedef struct {
    long int numerator, denominator;
} rational;

/***** public *****/
rational *create(long int n, long int d) {
    rational *r = (rational *) malloc(sizeof(rational));
    r->numerator = n;
    r->denominator = d;
    return r;
}
```


Rationale Zahlen überarbeitet (3)

```
void addR(rational *ret, rational *a, rational *b) {
    ret->numerator = a->numerator * b->denominator
            + a->denominator * b->numerator;
    ret->denominator = a->denominator * b->denominator;

    kuerzen(ret);
}

void mulR(rational *ret, rational *a, rational *b) {
    ret->numerator = a->numerator * b->numerator;
    ret->denominator = a->denominator * b->denominator;

    kuerzen(ret);
}
...
```


Rationale Zahlen überarbeitet (4)

main.c

```
#include "rational.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    char s1[40], s2[40], s3[40];
    rational_t *a[3], *b[3], *c;

    a[0] = create(5, 7);
    a[1] = create(5, 6);
    a[2] = create(3, 7);
```

Rationale Zahlen überarbeitet (5)

```
    b[0] = create(3, 4);
    b[1] = create(4, 15);
    b[2] = create(6, 14);
    c = create(1, 1);

    for (i = 0; i < 3; i++) {
        addR(c, a[i], b[i]);
        printf("%s + %s = %s\n", toString(a[i], s1),
               toString(b[i], s2), toString(c, s3));
    ...
}
```


Make-Utility

In größeren Projekten werden im Laufe der Programmierarbeit immer mehr Module fertig und stabil sein.

Um das gesamte Programm zu testen, müssen die Module übersetzt und gelinkt werden. Haben sich seit dem letzten Test Module geändert, müssen sie neu übersetzt und alle davon abhängigen Module müssen neu gelinkt werden.

Das Make-Utility hilft, die Übersicht zu behalten. Das Programm liest eine Datei (`makefile`), die Regeln enthält:

- Wie wird aus den Quelltextdateien das Programm zusammengesetzt und
- von welchen Dateien ist ein Modul abhängig?

Make-Utility (2)

Abhängigkeiten:

- Ausführbares Programm: abhängig von Objektdateien.
- Objektdateien sind abhängig von Quelltextdateien.
- Quelltextdateien: abhängig von den Definitionsdateien (Header-Dateien).

Allgemeine Form des Makefiles:

```
Zieldatei: Dateien, von denen Zieldatei abhängig ist
<tab>  Befehl, um Zieldatei zu erzeugen
Zieldatei: Dateien, von denen Zieldatei abhängig ist
<tab>  Befehl, um Zieldatei zu erzeugen
...
```


Make-Utility: Beispiel

```
OPT=-Wall -ansi -pedantic -g
all: rational.o main.o
       gcc -o calc main.o rational.o
rational.o: rational.c rational.h
       gcc $(OPT) -c rational.c
main.o: main.c rational.h
       gcc $(OPT) -c main.c
```

- Zieldateien werden mit GNU C-Compiler `gcc` erzeugt.
- `make` erzeugt das erste im Makefile angegebene Ziel.
- Der Aufruf `make ziel` erzeugt die angegebene Zieldatei.
- Ziel wird nur dann erzeugt, wenn die Zieldatei älter ist als eine der Dateien, von denen das Ziel abhängig ist.

Sortieren

Viel Rechenzeit entfällt in der Praxis auf Sortiervorgänge.

- Datensätze bestehen nicht aus elementaren, sondern aus zusammengesetzten Datentypen.
- Fragen: Anzahl der Sätze bekannt? Intern oder extern sortieren? Sätze tauschen oder Index berechnen? ...

Datensätze und Schlüssel:

```
typedef struct {
   char *name, *vname;
   long matrikelnr;
   short alter, fb;
} student_t;
```

Sortieren nach:

- `matrikelnr` oder
- `fb, alter` oder
- `name, vname, alter`

Sortieren (2)

Ziel: Sortieren durch Prozedur `sort` verfügbar machen.

Annahmen:

- Nur internes Sortieren (Werte in Vektor gespeichert).
- Anzahl der Datensätze ist bekannt.
- Datensätze werden getauscht, keinen Index anlegen.

Wenn möglich:

- Sortieren nach verschiedenen Schlüsseln ermöglichen.
- Unabhängigkeit von verwendeten Datentypen.

Sortieren: 1. Versuch

```
#include <stdio.h>
...
void main(void) {
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {5.0, 4.0, 3.0, 2.0, 1.0};

    sortLong(u, 9);
    outputLong(u, 9);

    sortDouble(v, 5);
    outputDouble(v, 5);
}
```


Sortieren: 1. Versuch (2)

```
void sortLong(long *a, int n) {
    int i, j;
    long t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
void outputLong(long *a, int n) ...
```

Sortieren: 1. Versuch (3)

```
void sortDouble(double *a, int n) {
    int i, j;
    double t;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}
void outputDouble(double *a, int n) ...
```


Sortieren: 1. Versuch (4)

Probleme:

- Funktionalitäten sind mehrfach implementiert für verschiedene Datentypen:
  - * Sortieren
  - * Ausgabe
  - * Objekte tauschen
- Wiederverwendung nur durch Cut&Paste.

$\Rightarrow$ Fehler sind an vielen Programmstellen zu beseitigen.

Lösung:

- Unabhängigkeit vom Datentyp durch Zeiger auf `void`.
- Objekte tauschen durch Prozedur `swap()` realisieren.

Objekte vertauschen: 1. Versuch

```
void swap(void *a, int x, int y) {
    void t = a[x];
    a[x] = a[y];
    a[y] = t;
}
```

So nicht!

- Der Wert eines Objekts vom Typ `void` kann in keiner Weise verarbeitet werden, er darf weder explizit noch implizit in einen anderen Typ umgewandelt werden.
  Compiler: `variable or field ‘t’ declared void`
- Index-Operator funktioniert bei Zeiger auf `void` nicht.
  Compiler: `warning: dereferencing ‘void *’ pointer`

Objekte vertauschen: 2. Versuch

```
void swap(void *a, void *b) {
    void *t;
    *t = *a;
    *a = *b;
    *b = *t;
}
```

So nicht! Inhaltsoperator funktioniert bei Zeiger auf `void` nicht. Compiler liefert:

```
invalid use of void expression
warning: dereferencing ‘void *’ pointer
```

Objekte vertauschen

Inhalte byte-weise vertauschen:

```
void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Reflexion

Was haben wir erreicht?

- gemeinsame Funktionalität `swap` als eine in sich abgeschlossene Funktion bereitgestellt
- `swap` ist unabhängig vom verwendeten Datentyp

Was ist noch zu tun?

- beschränken auf eine Funktion `sort`
- beschränken auf eine Funktion `output`
- sortieren nach verschiedenen Schlüsseln

Sortieren: 2. Versuch

```
...
typedef enum {LONG, DOUBLE, CHAR} type_t;
...
void main(void) {
    char t[] = {'f', 'e', 'd', 'c', 'b', 'a'};
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

    sort(t, 6, CHAR);
    output(t, 6, CHAR);
    sort(u, 9, LONG);
    output(u, 9, LONG);
    sort(v, 6, DOUBLE);
    output(v, 6, DOUBLE);
}
```


Sortieren: 2. Versuch (2)

```
void sort(void *a, int n, type_t type) {
   int i, j;
   for (i = 0; i < n; i++)
      for (j = i + 1; j < n; j++)
         if (type == LONG) {
            if (((long *) a)[i] > ((long *) a)[j])
                swap(a, i, j, sizeof(long));
         } else if (type == CHAR) {
            if (((char *) a)[i] > ((char *) a)[j])
                swap(a, i, j, sizeof(char));
         } else {
            if (((double *) a)[i] > ((double *) a)[j])
                swap(a, i, j, sizeof(double));
         }
}
```

Sortieren: 2. Versuch (3)

```
void output(void *a, int n, type_t type) {
   int i;
   for (i = 0; i < n; i++)
      if (type == LONG)
         printf("a[%d] = %ld\n", i, ((long *) a)[i]);
      else if (type == CHAR)
         printf("a[%d] = %c\n", i, ((char *) a)[i]);
      else printf("a[%d] = %f\n", i, ((double *) a)[i]);
}
```


Sortieren: 2. Versuch (4)

Probleme:

- Lange, unübersichtliche `if/else`-Anweisungen.
- Innerhalb der `if/else`-Konstrukte: Cut&Paste
- Für jeden selbstdefinierten Datentyp (Adresse, Kunde, Vertrag, ...) müssen die `if/else`-Anweisungen erweitert werden.

Lösung:

- Vergleichsoperation pro Datentyp implementieren und an die Sortierfunktion übergeben.

⇒ **Zeiger auf Funktionen**

Zeiger auf Funktionen

Jede Funktion besitzt eine Adresse:

```
int min(int a, int b) { return (a < b) ? a : b; }
int max(int a, int b) { return (a > b) ? a : b; }

void main(void) {
    int (*fp)(int, int);

    fp = &min;          /* fp zeigt auf min */
    printf("min(5, 7) = %d\n", (*fp)(5, 7));

    fp = &max;          /* fp zeigt auf max */
    printf("max(5, 7) = %d\n", (*fp)(5, 7));
}
```


Zeiger auf Funktionen (2)

Erklärung:

- `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert liefert und zwei `int`-Werte als Parameter verlangt.
- `*fp` kann für `min` und `max` benutzt werden.
- **nicht verwechseln mit** `int *fp(int, int)`: Funktion, die zwei `int`-Werte als Parameter hat und einen Zeiger auf einen `int`-Wert als Ergebnis liefert!

Zeiger auf Funktionen können

- zugewiesen,
- in Vektoren eingetragen,
- an Funktionen übergeben werden usw.

Zeiger auf Funktionen (3)

Beispiele aus der Standardbibliothek:

- `void qsort(void *base, size_t nmemb, size_t size, int (*compar)(void *, void *))`

  sortiert ein Array mit `nmemb` Elementen der Größe `size` (das erste Element ist bei `base`). Dazu muss eine Vergleichsfunktion `compar` angegeben werden.

- `void *bsearch(void *key, void *base, size_t nmemb, size_t size, int (*compar)(void *, void *))`

  durchsucht ein Array mit `nmemb` Elementen der Größe `size` (erstes Element bei `base`) nach einem Element `key`. Dazu muss eine Vergleichsfunktion `compar` angegeben werden.

Zeiger auf Funktionen (4)

Beispiele aus der Standardbibliothek: (Fortsetzung)

- `void exit(int status)` beendet ein Programm normal. Die `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung durchlaufen.
- `int atexit(void (*fcn)(void))` hinterlegt Funktion `fcn`. Liefert einen Wert ungleich 0, wenn die Funktion nicht hinterlegt werden konnte.

Zeiger auf Funktionen (5)

Werden oft bei GUI-Elementen verwendet, wo sie als **callback function** bei einem Benutzer-Event (Ereignis) aufgerufen werden:

- Was soll passieren, wenn der Benutzer auf eine Schaltfläche (Button) klickt?
- Soll das Programm schlafen oder weiterrechnen, wenn das Fenster zum Icon verkleinert werden soll?
- Sind offene Datenbankverbindungen zu schließen, falls der Benutzer das Fenster schließt?

Sortieren: Zeiger auf Funktionen

Was ist zu tun?

- lange, unübersichtliche `if/else`-Anweisungen ersetzen
- Vergleichsfunktionen pro Datentyp implementieren und an die Sortierfunktion als Parameter übergeben
- Ausgabeprozeduren pro Datentyp implementieren und an die Ausgabefunktion als Parameter übergeben

Sortieren

```
...
int main() {
    char t[] = {'f', 'e', 'd', 'c', 'b', 'a'};
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

    sort(t, 6, sizeof(char),
         (int (*)(void *, void *))cmpChar);
    output(t, 6, (void (*)(void *, int))outChar);

    sort(u, 9, sizeof(long),
         (int (*)(void *, void *))cmpLong);
    output(u, 9, (void (*)(void *, int))outLong);
    ...
}
```


Sortieren (2)

```
void sort(void *a, int n, int size,
          int (*cmp)(void *, void *)) {
    int i, j;
    void *x, *y;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            x = a + i * size;
            y = a + j * size;
            if ((*cmp)(x, y) > 0)
                swap(a, i, j, size);
        }
}
```


Sortieren (3)

```
int cmpLong(long *x, long *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
int cmpDouble(double *x, double *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
int cmpChar(char *x, char *y) {
    if (*x == *y) return 0;
    if (*x > *y) return 1;
    return -1;
}
```


Sortieren (4)

```
void output(void *a, int n, void (*out)(void *, int)) {
   int i;

   for (i = 0; i < n; i++) {
      printf("a[%d] = ", i);
      out(a, i);
      printf("\n");
   }
}
```

Sortieren (5)

```
void outChar(char *a, int pos) {
    printf("%c", a[pos]);
}

void outLong(long *a, int pos) {
    printf("%ld", a[pos]);
}

void outDouble(double *a, int pos) {
    printf("%.2f", a[pos]);
}
```


Sortieren (6)

Anstelle von mehreren Sortierprozeduren haben wir jetzt mehrere Vergleichsfunktionen. Warum soll das besser sein?

- die `sort`-Funktionen enthalten mittels Cut&Paste duplizierten Code → fehleranfällig, schlecht erweiterbar
- Für jeden selbstdefinierten Datentyp **müssen** wir
  * eine Vergleichsfunktion und
  * eine Ausgabeprozedur bereitstellen,

  denn:
  * Nach welchem Schlüssel soll sortiert werden?
  * Wie soll die Ausgabe formatiert sein?

Sortieren (7)

Programmiermodelle:

- **modulare Programmierung:** Alle spezifischen Operationen und Daten in einem Modul kapseln.
- **objektorientierte Programmierung:** Module werden zu Klassen. Zusätzlich: Polymorphismus, Vererbung

Reflexion: Wir wollten

- Sortieren durch Prozedur `sort` verfügbar machen,
- Unabhängigkeit von verwendeten Datentypen und
- Sortieren nach verschiedenen Schlüsseln ermöglichen.

der letzte Punkt ist noch nicht erfüllt
⇒ **variabel lange Parameterlisten**

Variabel lange Parameterlisten

Motivation: Wir haben bereits (unbewusst?) variabel lange Parameterlisten angewendet:

```
printf("Ergebnis: %d\n", erg);
printf("fib[%d] = %d\n", i, fib[i]);
printf("bin[%d][%d] = %d\n", i, j, bin[i][j]);

scanf("%d", &no);
scanf("%d, %f", &no, &epsilon);
```

Wie ist das eigentlich realisiert?

Variabel lange Parameterlisten (2)

Motivation: Bei Datenbankzugriffen mittels SQL können die Daten nach verschiedenen Kriterien sortiert werden.

```
select * from student order by matrikelnr;
select * from student order by fachbereich, alter;
select * from student order by name, vorname, alter;
```

Wie sähe denn entsprechender C-Code aus?

```
student_t arr[100];
...
sortBy(arr, "matrikelnr");
sortBy(arr, "fachbereich", "alter");
sortBy(arr, "name", "vorname", "alter");
```


Variabel lange Parameterlisten (3)

Die Standardbibliothek `stdarg` bietet die Möglichkeit, eine Liste von Funktionsargumenten abzuarbeiten, deren Länge und Datentypen nicht bekannt sind.

Beispiele:

```
int printf(const char *format, ...)
int scanf(const char *format, ...)
```

Parameterliste endet mit `...` → die Funktion darf mehr Argumente akzeptieren als Parameter explizit beschrieben sind

`...` darf nur am Ende einer Argumentenliste stehen

Variabel lange Parameterlisten (4)

Beispiel: `int fkt(char *fmt, ...);`

Mit dem **Typ** `va_list` definiert man eine Variable, die der Reihe nach auf jedes Argument verweist.

```
va_list vl;
```

Das **Makro** `va_start` initialisiert `vl` so, dass die Variable auf das erste unbenannte Argument zeigt. **Das Makro muss einmal aufgerufen werden, bevor `vl` benutzt wird.**

Es muss mindestens einen Parameter mit Namen geben, da `va_start` den letzten Parameternamen benutzt, um anzufangen.

```
va_start(vl, fmt);
```


Variabel lange Parameterlisten (5)

Jeder Aufruf des **Makros** `va_arg` liefert ein Argument und bewegt `vl` auf das nächste Argument.

`va_arg` benutzt einen Typnamen, um zu entscheiden, welcher Datentyp geliefert und wie `vl` fortgeschrieben wird.

```
ival = va_arg(vl, int);
sval = va_arg(vl, char *);
```

Vorsicht: Der Typ des Arguments wird nicht automatisch erkannt. Um den korrekten Typ angeben zu können, wird ein Format-String wie bei `printf(const char *, ...)` benutzt.

Variabel lange Parameterlisten (6)

Vorsicht: Das Ende der Liste kann **nicht** anhand eines `NULL`-Wertes erkannt werden.

So nicht!

```
while (vl != NULL) {
    val = va_arg(vl, int);
    ...
}
```

⇒ Die Anzahl und die Datentypen der Parameter müssen bekannt sein. Beides kann mittels eines Format-Strings wie bei `printf` erreicht werden.

Variabel lange Parameterlisten (7)

Beispiel: Format-String

```
for (; *fmt; fmt++) {
   switch(*fmt) {
      case 'd': ival = va_arg(vl, int);    break;
      case 'f': fval = va_arg(vl, double); break;
      case 's': sval = va_arg(vl, char *); break;
   }
   ...
```

Eventuell notwendige Aufräumarbeiten erledigt `va_end`.

```
va_end(vl);
```


Beispiel `stdarg`: Hauptprogramm

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

int fkt(char *, ...);

int main(void) {
    fkt("sfdsd", "eins", 2.0, 3, "vier", 5);
    fkt("fdsd", 6.0, 7, "acht", 9);
    return 0;
}
```


Beispiel `stdarg`: Format-String abarbeiten

```
void getAndPrintNextValue(char c, va_list *l);

int fkt(const char *fmt, ...) {
    int z;
    va_list l;

    va_start(l, fmt);
    for (z = 0; *fmt; fmt++, z++) {
        getAndPrintNextValue(*fmt, &l);
    }
    va_end(l);

    return z;
}
```

Beispiel `stdarg`: Parameter auswerten

```
void getAndPrintNextValue(char c, va_list *l) {
    char *sval;
    int ival;
    double fval;
    if (c == 'd') {
        ival = va_arg(*l, int);
        printf("%d (int)\n", ival);
    } else if (c == 'f') {
        fval = va_arg(*l, double);
        printf("%f (double)\n", fval);
    } else if (c == 's') {
        sval = va_arg(*l, char *);
        printf("%s (char *)\n", sval);
    }
}
```


Sortieren nach verschiedenen Schlüsseln

Wir brauchen

- eine Prozedur `sort` mit variabler Anzahl Parameter
- und eine Vergleichsfunktion für beliebige Schlüssel.

Frage: Warum verwenden wir nicht für jede Kombination von Schlüsseln eine eigene Sortierfunktion?

→ zuviele Kombinationen

$$3 \rightarrow 15 = \binom{3}{1} \cdot 1! + \binom{3}{2} \cdot 2! + \binom{3}{3} \cdot 3!$$
$$4 \rightarrow 64 = \binom{4}{1} \cdot 1! + \binom{4}{2} \cdot 2! + \binom{4}{3} \cdot 3! + \binom{4}{4} \cdot 4!$$
$$5 \rightarrow 325 = \binom{5}{1} \cdot 1! + \binom{5}{2} \cdot 2! + \binom{5}{3} \cdot 3! + \binom{5}{4} \cdot 4! + \binom{5}{5} \cdot 5!$$

→ Code-Duplizierung mittels Cut&Paste

Sortieren nach verschiedenen Schlüsseln (2)

```
int cmpName(student_t *s1, student_t *s2) {
    return strcmp(s1->name, s2->name);
}

int cmpVorname(student_t *s1, student_t *s2) {
    return strcmp(s1->vorname, s2->vorname);
}

int cmpNameVorname(student_t *s1, student_t *s2) {
    int r = strcmp(s1->name, s2->name);
    if (r == 0)
        return strcmp(s1->vorname, s2->vorname);
    return r;
}
...
```


Sortieren nach verschiedenen Schlüsseln (3)

`sort.h`

```
#include <stdarg.h>

void sort(void *liste, int length, int size,
          int (*cmp)(void *liste, int pos1, int pos2,
                     int params, va_list args),
          int params, ...);

void swap(void *liste, int pos1, int pos2, int size);
```

Sortieren nach verschiedenen Schlüsseln (4)

sort.c

```
#include "sort.h"
#include <string.h>

void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Sortieren nach verschiedenen Schlüsseln (5)

```
void sort(void *s, int n, int size,
        int (*cmp)(void *, int, int, int, va_list),
        int params, ...) {
    int i, j;
    va_list l;

    va_start(l, params);
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            if (cmp(s, i, j, params, l) > 0)
                swap(s, i, j, size);
        }
    va_end(l);
}
```


Sortieren nach verschiedenen Schlüsseln (6)

student.h

```
#include <stdarg.h>

typedef struct {
    char *name, *vname;
    short fb, alter;
    long matrikelnr;
} student_t;

int cmp(student_t *, int pos1, int pos2, int params,
        va_list liste);
void createStudent(student_t *, char *name, char *vname,
        short fb, short alter, long matrikelnr);
void output(student_t);
```


Sortieren nach verschiedenen Schlüsseln (7)

```
void setName(student_t *, char *);
char * getName(student_t);

void setVorname(student_t *, char *);
char * getVorname(student_t);

void setAlter(student_t *, short);
short getAlter(student_t);

void setFB(student_t *, short);
short getFB(student_t);

void setMatrikelnr(student_t *, long);
long getMatrikelnr(student_t);
```

Sortieren nach verschiedenen Schlüsseln (8)

student.c

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

void setName(student_t *s, char *name) {
    s->name = (char *) malloc(strlen(name) + 1);
    strcpy(s->name, name);
}
char * getName(student_t s) {
    return s.name;
}
```


Sortieren nach verschiedenen Schlüsseln (9)

```
void setVorname(student_t *s, char *vorname) {
    s->vname = (char *) malloc(strlen(vorname) + 1);
    strcpy(s->vname, vorname);
}
char * getVorname(student_t s) {
    return s.vname;
}

void setAlter(student_t *s, short alter) {
    s->alter = alter;
}
short getAlter(student_t s) {
    return s.alter;
}
```


Sortieren nach verschiedenen Schlüsseln (10)

```
void setFB(student_t *s, short fb) {
    s->fb = fb;
}
short getFB(student_t s) {
    return s.fb;
}

void setMatrikelnr(student_t *s, long matrikelnr) {
    s->matrikelnr = matrikelnr;
}
long getMatrikelnr(student_t s) {
    return s.matrikelnr;
}
```


Sortieren nach verschiedenen Schlüsseln (11)

```
void createStudent(student_t *s, char *name, char *vorn,
        short alter, short fb, long matnr) {
    setName(s, name);
    setVorname(s, vorn);
    setAlter(s, alter);
    setFB(s, fb);
    setMatrikelnr(s, matnr);
}

void output(student_t s) {
    printf("%s, %s, FB %d, Alter %d, Matrikelnr %ld\n",
           getName(s), getVorname(s), getFB(s),
           getAlter(s), getMatrikelnr(s));
}
```

Sortieren nach verschiedenen Schlüsseln (12)

```
int cmp(student_t *s, int x, int y, int params,
        va_list list) {
   int r = 0, n = 0;
   char *key;

   for (; r == 0 && n < params; n++) {
      key = va_arg(list, char *);
      if (strcmp(key, "name") == 0)
         r = strcmp(s[x].name, s[y].name);
      else if (strcmp(key, "vorname") == 0)
         r = strcmp(s[x].vorname, s[y].vorname);
      else if (strcmp(key, "fb") == 0)
         r = s[x].fb == s[y].fb
             ? 0
             : (s[x].fb > s[y].fb ? 1 : -1);
```


Sortieren nach verschiedenen Schlüsseln (13)

```
      else if (strcmp(key, "alter") == 0)
         r = s[x].alter == s[y].alter
             ? 0
             : (s[x].alter > s[y].alter ? 1 : -1);
      else if (strcmp(key, "matrikelnr") == 0)
         r = s[x].matrikelnr == s[y].matrikelnr
             ? 0
             : (s[x].matrikelnr > s[y].matrikelnr
                ? 1 : -1);
   }

   return r;
}
```


Sortieren nach verschiedenen Schlüsseln (14)

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#include "student.h"

int main(void) {
   int i;
   student_t liste[6], *t = liste;

   createStudent(t++, "Müller" , "Hans", 32, 3, 123456);
   createStudent(t++, "Meier",   "Gabi", 37, 4, 765432);
   createStudent(t++, "Meier",   "Rosi", 34, 4, 987612);
   createStudent(t++, "Müller", "Josef", 38, 2, 471115);
   createStudent(t++, "Müller", "Josef", 35, 6, 121341);
   createStudent(t++, "Maier", "Walter", 32, 3, 213111);
```


Sortieren nach verschiedenen Schlüsseln (15)

```
   printf("unsortiert:\n");
   for (i = 0; i < 6; i++)
       output(liste[i]);

   printf("\nSchlüssel: Name, Vorname, Alter\n");
   sort(liste, 6, sizeof(student_t),
        (int (*)(void *, int, int, int, va_list)) &cmp,
        3, "name", "vorname", "alter");
   for (i = 0; i < 6; i++)
       output(liste[i]);
```

Sortieren nach verschiedenen Schlüsseln (16)

```
    printf("\nSchlüssel: Fachbereich, Alter\n");
    sort(liste, 6, sizeof(student_t),
         (int (*)(void *, int, int, int, va_list)) &cmp,
         2, "fb", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Vorname, Alter\n");
    sort(liste, 6, sizeof(student_t),
         (int (*)(void *, int, int, int, va_list)) &cmp,
         2, "vorname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);
    return 0;
}
```


Sortieren: Key-Value-Paare

Funktion `cmp` unabhängig von realer Datenstruktur implementieren → Key-Value-Paare verwenden

types.h:

```
#ifndef _TYPES_H
#define _TYPES_H

typedef enum {
    INT, FLOAT, STRING
} type_t;

typedef union {
    long ival;
    double fval;
    char *sval;
} value_t;
```


Sortieren: Key-Value-Paare (2)

```
typedef struct {
    char *key;
    type_t typ;
    value_t value;
} keyValueElem_t;

typedef struct {
    keyValueElem_t *elem;
    int size;
} keyValue_t;

#endif
```


Sortieren: Key-Value-Paare (3)

sort.h:

```
#include "types.h"
#include <stdarg.h>

int cmp(keyValue_t, keyValue_t, int, va_list);
void sort(keyValue_t *, int, int, ...);
void swap(void *, int, int, int);
keyValueElem_t get(keyValue_t, char *);
```

Sortieren: Key-Value-Paare (4)

sort.c:

```
#include "sort.h"
#include <string.h>

void swap(void *a, int i, int j, int size) {
    char c, *ta = a;
    int k;

    for (k = 0; k < size; k++) {
        c = *(ta + i * size + k);
        *(ta + i * size + k) = *(ta + j * size + k);
        *(ta + j * size + k) = c;
    }
}
```


Sortieren: Key-Value-Paare (5)

```
keyValueElem_t get(keyValue_t s, char *key) {
    int i = 0;
    char c = 1;
    keyValueElem_t e;

    do {
        e = s.elem[i++];
        c = strcmp(key, e.key);
    } while (i < s.size && c != 0);

    return e;
}
```


Sortieren: Key-Value-Paare (6)

```
int cmp(keyValue_t a, keyValue_t b, int n, va_list l) {
  int r = 0;
  char *key;
  keyValueElem_t kv_a, kv_b;

  for (; r == 0 && n > 0; n--) {
    key = va_arg(l, char*);
    kv_a = get(a, key);
    kv_b = get(b, key);
```


Sortieren: Key-Value-Paare (7)

```
    if (kv_a.typ == INT)
      r = (kv_a.value.ival > kv_b.value.ival
          ? 1
          : kv_a.value.ival < kv_b.value.ival ? -1 : 0);
    else if (kv_a.typ == FLOAT)
      r = (kv_a.value.fval > kv_b.value.fval
          ? 1
          : kv_a.value.fval < kv_b.value.fval ? -1 : 0);
    else r = strcmp(kv_a.value.sval, kv_b.value.sval);
  }
  return r;
}
```

Sortieren: Key-Value-Paare (8)

```
void sort(keyValue_t *s, int n, int p, ...) {
    int i, j;
    va_list l;

    va_start(l, p);
    for (i = 0; i < n; i++) {
        for (j = i + 1; j < n; j++) {
            if (cmp(s[i], s[j], p, l) > 0)
                swap(s, i, j, sizeof(keyValue_t));
        }
    }
    va_end(l);
}
```


Sortieren: Key-Value-Paare (9)

student.h:

```
#include "types.h"

void createStudent(keyValue_t *, char *, char *,
        short, short, long);
void output(keyValue_t);

void setName(keyValue_t *, char *);
char * getName(keyValue_t);

void setVorname(keyValue_t *, char *);
char * getVorname(keyValue_t);
```


Sortieren: Key-Value-Paare (10)

```
void setAlter(keyValue_t *, short);
short getAlter(keyValue_t);

void setFB(keyValue_t *, short);
short getFB(keyValue_t);

void setMatrikelnr(keyValue_t *, long);
long getMatrikelnr(keyValue_t);
```


Sortieren: Key-Value-Paare (11)

student.c:

```
#include "student.h"
#include <stdio.h>

void setName(keyValue_t *s, char *name) {
    s->elem[0].key = "name";
    s->elem[0].typ = STRING;
    s->elem[0].value.sval = name;
}

char * getName(keyValue_t s) {
    return s.elem[0].value.sval;
}
```

Sortieren: Key-Value-Paare (12)

```
void setVorname(keyValue_t *s, char *vname) {
    s->elem[1].key = "vname";
    s->elem[1].typ = STRING;
    s->elem[1].value.sval = vname;
}

char * getVorname(keyValue_t s) {
    return s.elem[1].value.sval;
}
```


Sortieren: Key-Value-Paare (13)

```
void setAlter(keyValue_t *s, short alter) {
    s->elem[2].key = "alter";
    s->elem[2].typ = INT;
    s->elem[2].value.ival = (long)alter;
}

short getAlter(keyValue_t s) {
    return s.elem[2].value.ival;
}
```


Sortieren: Key-Value-Paare (14)

```
void setFB(keyValue_t *s, short fb) {
    s->elem[3].key = "fb";
    s->elem[3].typ = INT;
    s->elem[3].value.ival = (long)fb;
}

short getFB(keyValue_t s) {
    return s.elem[3].value.ival;
}
```


Sortieren: Key-Value-Paare (15)

```
void setMatrikelnr(keyValue_t *s, long matrikelnr) {
    s->elem[4].key = "matrikelnr";
    s->elem[4].typ = INT;
    s->elem[4].value.ival = matrikelnr;
}

long getMatrikelnr(keyValue_t s) {
    return s.elem[4].value.ival;
}
```

Sortieren: Key-Value-Paare (16)

```
void createStudent(keyValue_t *s, char *name,
        char *vname, short alter, short fb,
        long matrikelnr) {
    setName(s, name);
    setVorname(s, vname);
    setAlter(s, alter);
    setFB(s, fb);
    setMatrikelnr(s, matrikelnr);
}

void output(keyValue_t s) {
    printf("%10s, %8s, FB %d, Alter %d, Matrnr %ld\n",
        getName(s), getVorname(s), getFB(s),
        getAlter(s), getMatrikelnr(s));
}
```


Sortieren: Key-Value-Paare (17)

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include "sort.h"
#include "student.h"

void main(void) {
    int i;
    keyValue_t liste[6], *t = liste;

    for (i = 0; i < 6; i++) {
        liste[i].elem = (keyValueElem_t *)
            malloc(5 * sizeof(keyValueElem_t));
        liste[i].size = 5;
    }
```


Sortieren: Key-Value-Paare (18)

```
createStudent(t++, "Fischer",  "Hans", 32, 3, 1234567);
createStudent(t++, "Meier",    "Gabi", 37, 4, 7654321);
createStudent(t++, "Meier",    "Rosi", 34, 4, 9876123);
createStudent(t++, "Fischer", "Josef", 38, 2, 4711815);
createStudent(t++, "Fischer", "Josef", 35, 6, 1213141);
createStudent(t++, "Maier",  "Walter", 32, 3, 2131411);

printf("unsortiert:\n");
for (i = 0; i < 6; i++)
    output(liste[i]);
```


Sortieren: Key-Value-Paare (19)

```
    printf("\nSchlüssel: Name, Vorname, Alter\n");
    sort(liste, 6, 3, "name", "vname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Fachbereich, Alter\n");
    sort(liste, 6, 2, "fb", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);

    printf("\nSchlüssel: Vorname, Alter\n");
    sort(liste, 6, 2, "vname", "alter");
    for (i = 0; i < 6; i++)
        output(liste[i]);
}
```

Sortieren

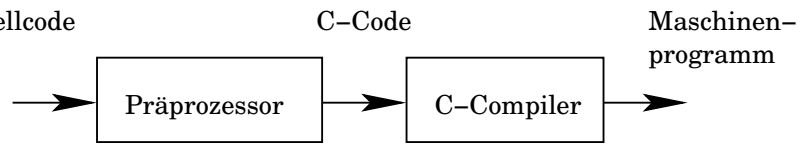
Anmerkungen:

- Die Funktionen `sort()`, `cmp()` und `get()` sind speziell für den Datentyp `keyValue_t` implementiert.
- `sort()` benutzt eine feste Vergleichsfunktion.

Der C-Präprozessor bietet weitere Möglichkeiten, den Code maschinen- und datentyp-unabhängig zu schreiben.

C-Präprozessor

Der Präprozessor bearbeitet den Programm-Code vor der eigentlichen Übersetzung.



Präprozessordirektiven sind Programmzeilen, die mit einem `#`-Zeichen beginnen:

- `#include`: Inhalt einer Datei während der Übersetzung einfügen
- `#define`: Einen Namen durch eine beliebige Zeichenfolge ersetzen (parametrisierbar)

C-Präprozessor: `#include`

Eine Quellzeile wie

`#include <filename>` oder `#include "filename"`

wird durch den Inhalt der Datei `filename` ersetzt:

- `"filename"`: die Suche nach der Datei beginnt dort, wo das Quellprogramm steht.
- `<filename>`: die Datei wird in einem speziellen Verzeichnis gesucht. (Linux: `/usr/include/`)

C-Präprozessor: `#include` (2)

Für den Inhalt der eingefügten Dateien gibt es keine Einschränkungen. Aber: In der Regel werden nur Definitionsdateien eingebunden. Sie enthalten:

- `#define`-Anweisungen,
- weitere `#include`-Anweisungen,
- Typdeklarationen und
- Funktionsprototypen

Die Deklarationen eines großen Programms werden zentral gehalten: Alle Quelldateien arbeiten mit denselben Definitionen und Variablendeklarationen.

C-Präprozessor: `#define`

`#define <name> <ersatztext>` bewirkt, dass im Quelltext die Zeichenfolge `name` durch `ersatztext` ersetzt wird.

Der Ersatztext ist der Rest der Zeile. Eine lange Definition kann über mehrere Zeilen fortgesetzt werden, wenn ein \ am Ende jeder Zeile steht, die fortgesetzt werden soll.

```
#define PRIVATE static
#define PUBLIC
#define ERROR \
    printf("\aEingabedaten nicht korrekt\n");
```

Der Gültigkeitsbereich einer `#define`-Anweisung erstreckt sich von der Anweisung bis ans Ende der Datei.

C-Präprozessor: `#define` (2)

Makros mit Parametern erlauben, dass der Ersatztext bei verschiedenen Aufrufen verschieden sein kann:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Ein Makroaufruf ist **kein** Funktionsaufruf! Ein Aufruf von `max` wird **direkt im Programmtext expandiert**. Die Zeile

```
x = MAX(p + q, r + s);
```

wird ersetzt durch die Zeile

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Textersetzung! Keine Auswertung von Ausdrücken!

C-Präprozessor: `#define` (3)

Hinweis: Anders als bei Funktionen genügt eine einzige Definition von `MAX` für verschiedene Datentypen.

Vorsicht: Im Beispiel werden Ausdrücke eventuell zweimal bewertet. Das führt bei Operatoren mit Nebenwirkungen (Inkrement, Ein-/Ausgabe) zu Problemen.

```
i = 2;
j = 3;
x = MAX(i++, j++);    /* i?? j?? x?? */
```

Vorsicht: Der Ersatztext muss sorgfältig geklammert sein, damit die Reihenfolge von Bewertungen erhalten bleibt:

```
#define SQR(x) x * x
```

Was passiert beim Aufruf `SQR(z + 1)`?

C-Präprozessor: `#define` (4)

Makros können bereits definierte Makros enthalten:

```
#define SQR(x) (x) * (x)
#define CUBE(x) SQR(x) * (x)
```

Die Gültigkeit einer Definition kann durch `#undef` aufgehoben werden:

```
#undef MAX
#undef CUBE
int MAX(int a, int b) ...
```

Textersatz findet nur für Namen, aber nicht innerhalb von Zeichenketten statt.

C-Präprozessor: `#define` (5)

Textersatz und Zeichenketten:

- `#<parameter>` wird durch `"<argument>"` ersetzt.
- Im Argument wird " durch \" und \ durch \\ ersetzt. Resultat: gültige konstante Zeichenkette.

Beispiel: Debug-Ausgabe

```
#define dprint(expr) printf(#expr " = %f\n", expr);
...
dprint(x/y);
/* ergibt: printf("x/y" " = %f\n", x/y); */
```


C-Präprozessor: `#define` (6)

Mittels ## können Argumente aneinandergehängt werden.

Beispiel:

```
#include <stdio.h>
#define paste(head, tail) head ## tail

void main(void) {
    int x1 = 42;
    printf("x1 = %d\n", paste(x, 1));
}
```

Frage: Wozu braucht man das?

C-Präprozessor: `#define` (7)

Auszug aus `stdint.h`:

```
# if __WORDSIZE == 64
#  define __INT64_C(c)  c ## L
#  define __UINT64_C(c) c ## UL
# else
#  define __INT64_C(c)  c ## LL
#  define __UINT64_C(c) c ## ULL
# endif
```

Auszug aus `linux/ext2_fs.h`:

```
#define clear_opt(o, opt)    o &= ~EXT2_MOUNT_##opt
#define set_opt(o, opt)      o |= EXT2_MOUNT_##opt
```


C-Präprozessor: `#if`

Der Präprozessor selbst kann mit bedingten Anweisungen kontrolliert werden, die während der Ausführung bewertet werden:

- Texte abhängig vom Wert einer Bedingung einfügen.
- Programmteile abhängig von Bedingungen übersetzen.

Anwendung:

- um Definitionsdateien nur einmal einzufügen,
- für Debug-Zwecke und
- für systemabhängige Programmteile.

C-Präprozessor: `#if` (2)

`#if <expr>` prüft den Ausdruck (konstant, ganzzahlig).

Ist der Ausdruck „wahr“, werden die folgenden Zeilen bis `else`, `elif` bzw. `endif` eingefügt und damit übersetzt.

Beispiel:

```
#define DEBUG 1
...
#if DEBUG
    printf("Funktion getAdresse(%d): %s\n", i, str);
#endif
```


C-Präprozessor: `#ifdef`

`#ifdef <makro>` bzw. `#ifndef <makro>` prüft, ob ein Makro definiert bzw. nicht definiert ist.

Äquivalente Schreibweise:

```
#if defined <makro>
#if !defined <makro>
```

Beispiel: systemabhängige Übersetzung

```
#ifdef ALPHA
#   include "arch/alpha/semaphore.h"
#elif defined INTEL || defined I386
#   include "arch/i386/semaphore.h"
#endif
```


C-Präprozessor: 1. Beispiel

Jede Definitionsdatei veranlasst das Einfügen derjenigen Definitionsdateien, von denen sie abhängt.

Mehrfaches Einbinden der Datei `netdb.h` verhindern:

```
#ifndef _NETDB_H
#define _NETDB_H 1
/*** eigentlicher Inhalt der Datei netdb.h ***/
#endif
```

Erklärung:

- Beim ersten Einfügen wird `_NETDB_H` definiert.
- Wird die Datei nochmals eingefügt, ist `_NETDB_H` definiert und alles bis zum `#endif` wird übersprungen.

C-Präprozessor: 2. Beispiel

```
#define SWAP(A, B, T) \
{                     \
    T t = A;          \
    A = B;            \
    B = t;            \
}

void sort(long *a, int n) {
    int i, j;
    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j])
                SWAP(a[i], a[j], long)
}
```