C Dateien

INFORMATIONSAUSTAUSCH

Programme tauschen Information mit ihrer Umgebung aus
- Kommandozeilenargumente
- Kanäle bzw. Datenströme (Streams)

DATEIEN (FILES)

- Datei: geordnete Folge von Bytes
- Eingabe und Ausgabe (Lesen/Schreiben)
- Meist auf nichtflüchtigen Medien gespeichert (Disk, Stick, etc.)
- Vom Betriebssystem verwaltet (Dateisystem)
- Ablage in Verzeichnishierarchie (Ordner)
- Über Namen ansprechbar

DATEIEN IN C

- Nur Strom von Bytes (char-Stream)
- Weitere Organisation ist Aufgabe des Programms
- Unterscheidung in Text- und Binärdateien

PFADANGABE (LINUX, MACOS)

- Dateien haben einen eindeutigen Namen (Pfadname)
- Absoluter Pfad
  - `"/home/daten/meinFile.txt"`
  - Beginnt mit dem Wurzelverzeichnis `/`
- Relativer Pfad
  - `"meinFile.txt"`
  - `"./meinFile.txt"`
  - `"../daten/meinFile.txt"`
  - Relativ zum aktuellen Arbeitsverzeichnis `.`
  - Das übergeordnete Verzeichnis ist `..`

PFADANGABE (WINDOWS)

- Dateien haben einen eindeutigen Namen (Pfadname)
- Absoluter Pfad
  - `"c:\\home\\daten\\meinFile.txt"`
  - Beginnt mit dem Wurzelverzeichnis eines Laufwerks `"c:\\"`
- Relativer Pfad
  - `"meinFile.txt"`
  - `".\\meinFile.txt"`
  - `"..\\daten\\meinFile.txt"`
  - Es muss `"\\"` für den Pfadtrenner `"\"` verwendet werden

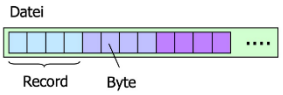
TEXTDATEI (WH)

- In der Datei sind Zeichencodes gespeichert
- Mit einem Texteditor lesbar und bearbeitbar
- Verschiedene Zeichencodierungen (ASCII, UTF-8, ...)
- C unterstützt im Wesentlichen ASCII (keine Umlaute...)
- Textzeilen verschiedener Längen
- Zeilenendezeichen: LF (Linux, Mac) oder CR-LF (Win)
- Beispiel: "12345\n" in Textdatei (ASCII Code)

Linux, MacOS	49	50	51	52	53	10
Windows	49	50	51	52	53	13	10

BINÄRDATEI (WH)

- Folge von Bytes: raw data
- Keine Interpretation als Zeichencodes
- Programm muss Aufbau der Datei kennen
  - Record: zusammengehörende Folge von Bytes
  - Beispiel: Record aus 4 Bytes, die int-Wert repräsentieren

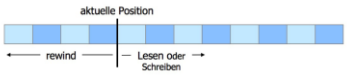


DIREKTER ZUGRIFF

- Random Access
- Lese/Schreib-Position in Datei ist wählbar (`seek`)

SEQUENTIELLER ZUGRIFF

- Ab aktueller Position lesen oder schreiben
- Lese/Schreib-Position kann auf Anfang gesetzt werden (`rewind`)



STANDARD-STREAMS

- `stdin`: Eingabekanal (Tastatur)
- `stdout`: Ausgabekanal (Bildschirm, Konsole)
- `stderr`: Fehlerkanal (Fehlermeldungen auf Konsole)

Beispiel: zweimal gleiches Resultat

```
printf("Hello World\n");
fprintf(stdout, "Hello World\n");
```

```c
#include <stdio.h>

int main (void) {
    FILE *fp;
    char filename[] = "test.txt";
    fp = fopen(filename, "w");
    if (fp == NULL) {
        fprintf(stderr, "Error opening file %s\n", filename);
        return 1;
    }
    fprintf(fp, "Hello World\n");
    fclose(fp);
}
```

Die Standardkanäle können auf der Kommandozeile in Dateien umgeleitet werden

Kommando	Bedeutung
`programm > datei`	Standardausgabe in Datei
`programm >> datei`	an Datei anhängen
`programm < datei`	Standardeingabe von Datei
`programm 2> datei`	nur Fehlerausgabe in Datei
`programm 2> NUL`	Fehlerausgabe ignorieren (Windows)
`programm 2> /dev/null`	Fehlerausgabe ignorieren (Linux/Unix)

DATEIOPERATIONEN (AUSWAHL)

Funktion	Beschreibung
`fopen`	Datei öffnen
`fprintf`, `fscanf`	Ein- / Ausgabe formatiert
`fputc`, `fgetc`	Zeichen schreiben/lesen
`fputs`, `fgets`	String schreiben/lesen
`fflush`	Pufferinhalt schreiben
`fclose`	Datei schliessen
`remove`	Datei löschen

DATEI ÖFFNEN

```
FILE *fopen (const char *filename, const char *mode)
```

- `filename`: absoluter oder relativer Pfa
- `mode`: lesen, schreiben, ...
- Rückgabewert
  - bei Erfolg: Zeiger auf File
  - bei Fehler: `NULL`
- `FILE` ist ein *struct* mit Informationen über die Datei

Wert	Bedeutung
`"r"`	Textdatei zum Lesen öffen
`"w"`	Textdatei zum Schreiben öffen
`"a"`	Textdatei zum Anhängen von Text öffnen
`"rb"`	Binärdatei zum Lesen öffnen
`"wb"`	Binärdatei zum Schreiben öffnen

IN TEXTDATEI SCHREIBEN

```
int fprintf (FILE *stream, const char *format, ...)
```

- Arbeitet wie `printf`
- Schreibt Daten in eine Datei
- Bei Erfolg: Anzahl geschriebener Zeichen
- Bei Fehler: negativer Wert

BEISPIEL

```c
#include <stdio.h>

int main (void) {
    FILE *fp;
    char filename[] = "test.txt";
    fp = fopen(filename, "w");
    if (fp == NULL) {
        printf("Error opening file %s\n", filename);
        return 1;
    }
    fprintf(fp, "Hello World\n");
    fclose(fp);
}
```

AUS TEXTDATEI LESEN

```
int fscanf (FILE *stream, const char *format, ...)
```

- Entspricht `scanf`, liest aber von *stream*
- Weitere Argumente müssen Zeiger sein
- Bei Erfolg: Anzahl gelesener und umgewandelter Eingaben
- Bei Dateiende oder Fehler: EOF
- Diese Funktion besser nicht verwenden (fehleranfällig)
- Besser Zeile mit *fgets* lesen und String mit *strtok* zerlegen

DATEI SCHLIESSEN

```
int fclose (FILE *stream)
```

- Datei zum Schreiben offen: schreibt Daten, welche sich noch im Puffer befinden, auf den Datenträger
- Schliesst die Datei
- Rückgabewert bei Erfolg: 0, bei Fehler: `EOF`

DATENBUFFER LEEREN

```
int fflush (FILE *stream)
```

- Schreibt gepufferte Daten, welche noch nicht auf den Datenträger geschrieben wurden
- Wirft noch nicht gelesene, aber gepufferte Daten weg

```
// einzelnes Zeichen schreiben
int fputc (const char c, FILE *stream);

// einzelnes Zeichen lesen
int fgetc (FILE *stream);

// ganzen String schreiben
int fputs (const char *str, FILE *stream);

// Zeichen bis '\n' in String-Buffer lesen, maximal num-1 Zeichen
char *fgets (char *str, int num, FILE *stream);
```

Achtung: Bei *fgets* wird das '\n'-Zeichen auch in den String-Buffer eingelesen, wenn noch Platz dafür ist

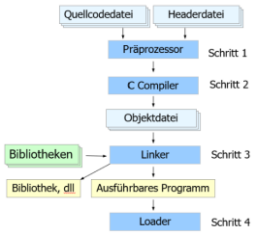
STRING TOKENIZER

```
char *strtok (char *str, const char *delimters)
```

- Liefert Pointer auf String bis zum nächsten Delimiter
- Gibt NULL zurück, wenn String-Ende erreicht wurde
- Sucht weitere Tokens mit NULL als erstes Argument
- Mehrere Delimiter möglich
- Bibliothek: *string.h*

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

int main (void) {
    char str[]  = "one,two,three";
    char *token = strtok(str, ",");

    while (token != NULL) {
        printf("token: %s\n", token);
        token = strtok(NULL, ",");
    }

    return 0;
}
```

C Prozess



- **Schritt 1: Präprozessor**
  - textuelle Vorverarbeitung von Direktiven
- **Schritt 2: Compiler**
  - erzeugt Objektcode: das ist bereits Maschinencode
  - enthält noch offene Referenzen (z.B. Funktionsaufrufe)
- **Schritt 3: Linker**
  - fügt Objektcode zusammen
  - bindet statische Bibliotheken ein
  - löst offene Referenzen auf
- **Schritt 4: Loader**
  - lädt Programm und ggf. dynamische Bibliotheken
  - startet Ausführung des Codes bei main()

- Führt Direktiven aus
- Definition von Konstanten (`#define`)
- Einfügen weiterer Quellen (`#include`) (normalerweise sind das Header-Files, Endung: `.h`)
- Ein-/Ausblenden von Blöcken (`#ifdef`, `#ifndef`)

- Fügt Object-Module zu ausführbarem Code zusammen
- Bindet Bibliotheken ein
- Ersetzt offene Referenzen, z.B. Funktionsaufrufe
- Überprüft ob deklarierte Variablen auch definiert sind
- Statisches Linken: Code wird direkt in das ausführbare Programm eingebunden
- Dynamisches Linken: Code wird als externe Referenz vermerkt und später beim Laden (Loader) aufgelöst

- Übersetzt Quellcode in Object-Code-Module
  - Quellcode-Dateien mit Endung `.c`
  - Prüfung auf syntaktische und grammatische Fehler
  - statische Typenprüfung
  - falls möglich Warnung bei logischen Fehlern
- Object-Code
  - Maschinen-Code
  - nicht ausführbar, enthält offene Referenzen z.B. auf Bibliotheksfunktionen
  - Dateien meist mit Endung `.o` oder `.obj`

- Lädt ausführbaren Code in den Speicher und startet ihn
- Lädt bei Bedarf dynamische Bibliotheken
- Springt zur Funktion `main` und startet das Programm
- Windows:
  - ausführbarer Code: `.exe`-Datei
  - dynamische Bibliotheken: `.dll`-Dateien

DER C-PRÄPROZESSOR

- Ziel ursprünglich: kompakter und lesbarer Code
- Problem: fehleranfällig, Fehler schwierig zu finden
- Macht textuelle Änderungen am Quellcode
- Auch unabhängig von C verwendbar
- Gesteuert über *Direktiven*

Aufruf auf der Kommandozeile:

```
gcc -E hello.c
```

```
#define ALARM(meinText)     \
    printf("******* ");     \
    printf("%s", meinText); \
    printf("******* ");
```

DREI WESENTLICHE AUFGABEN

- Dateien einfügen `#include`
- Makros ersetzen `#define`
- bedingte Übersetzung `#ifdef`, `#ifndef`, `#endif`

DATEIEN EINBINDEN: #include

```
#include <datei>
#include "datei"
```

`<...>` Header-Dateien aus Standard-Verzeichnissen

`"..."` Dateien aus aktuellem (Projekt-) Verzeichnis

KONSTANTEN UND MAKROS: #define

```
#define ANZAHL  100
#define MAL3(X) (3*X)

int main (void) {
  for (int i = 0; i < ANZAHL; i++)
    printf("Zahl: %d\n", MAL3(i));
}
```

```
int main (void) {
  for (int i = 0; i < 100; i++)
    printf("Zahl: %d\n", (3*i));
}
```

Regel 1:
Argumente mit Klammern versehen

```
#define QUADRAT(X) ((X)*(X))
```

Regel 2:
Falls möglich, Makros vermeiden und Funktionen verwenden

```
#ifdef DEBUG
  printf("Initialisierung fertig\n");
#endif
```

- Code wird eingefügt, wenn DEBUG definiert ist
- Mit `#define` in zentralem Header-File möglich
- Oder: beim ggc auf der Kommandozeile angeben `gcc -DDEBUG ...` oder `gcc -DDEBUG=Wert ...`

BEDINGTE ÜBERSETZUNG: #ifdef...

Konstante	Beschreibung
`#ifdef SYMBOL`	ist Symbol definiert?
`#ifndef SYMBOL`	ist Symbol nicht definiert?
`#undef SYMBOL`	Definition löschen
`#else`	alternativer Zweig
`#endif`	bedingten Block abschliessen

```
#ifdef DEBUG
  printf("debugging is on\n");
#endif
```

- Ganze Header-Datei als bedingten Block definieren
- Symbol für Bedingung aus Dateinamen zusammensetzen

Datei meine_defs.h

```
#ifndef MEINE_DEFS_H
#define MEINE_DEFS_H

// Code der Header-Datei

#endif
```

Symbol definiert →

Mehrfach-Includes

```
#ifndef MEINE_DEFS_H
#define MEINE_DEFS_H
// Code der Header-Datei
#endif
```

```
#ifndef MEINE_DEFS_H
#define MEINE_DEFS_H
// Code der Header-Datei
#endif
```

BEISPIEL: MIT HEADER-

```
// doit.h
#ifndef DOIT_H
#define DOIT_H

#define MAX 1000
int doit (int b);   /* Funktionsdeklaration */

#endif
```

```
// main.c
#include "doit.h"

int main (void) {
  return MAX + doit(10);
}
```

```
// doit.c
#include "doit.h"

int doit (int b) {  /* Funktionsdefinition */
  return MAX + b;
}
```

PRINZIPIEN MODULARISIERUNG

- Geringe Vernetzung
  - wenige Schnittstellen
- Schwache Kopplung zwischen Modulen
  - schmale Schnittstellen
- Hohe Kohäsion innerhalb Modul
  - hohe inhaltliche Einheit
  - zuständig für eine wohldefinierte Aufgabe
- Explizite Schnittstellen
  - Kommunikation mit Parameterübergabe
- Information Hiding
  - nur was global sein muss ist global
  - Verstecken der Implementationsdetails
  - trennt Schnittstelle und Implementation

Linux, Mac:

```
gcc -c main.c
gcc -c doit.c
gcc -o myprog *.o
```

Windows:

```
gcc -c main.c
gcc -c doit.c
gcc -o myprog.exe main.o doit.o
```

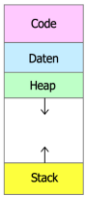
```
int globalFunc (int a) {        // für alle sichtbar
  ...
}
static int lokalFunc (int b) {  // nur im Modul sichtbar
  ...
}
```

```
int globalVar;        // für alle sichtbar
static int lokalVar;  // nur im Modul sichtbar
```

C Dynamischer Speicher

SPEICHERORGANISATION

- Code
  der Programmcode
- Daten
  globale und statische Variablen
- Heap
  dynamische Allokation von Speicher
- Stack
  lokale Daten (Variablen in Funktionen)
  Parameter für Funktionen



SPEICHERORGANISATION: DATEN

- Globale und statische Variablen
- Permanent im Speicher (Laufzeit des Programms)

```
int i;

void foo (int a) {
    int j;
    static int j;
    ...
}
```

SPEICHERORGANISATION: STACK

- Lokale Variablen und Parameter
- Beim Funktionsaufruf automatisch angelegt
- Bei Rückkehr (return) wieder abgebaut
- Temporär im Speicher

```
void foo (int a) {
    int i;
    static int j;
    ...
}
```

DYNAMISCHER SPEICHER IN C

- Speicher muss
  - explizit angefordert (alloziert) werden
  - explizit freigegeben werden, falls nicht: Memory Leak
- Speicher wird auf Heap alloziert
- Funktionen in `<stdlib.h>`
- Zwei grundlegende Funktionen
  - `malloc()`: alloziert Speicher
  - `free()`: gibt Speicher frei
- Zwei weitere Funktionen
  - `calloc()`: alloziert Speicher und setzt auf 0
  - `realloc()`: vergrössert oder verkleinert Speicher

SPEICHER ANFORDERN: malloc

```
void *malloc (size_t size)
```

- Alloziert Speicherplatz der Grösse *size* in Bytes
- Gibt Zeiger auf den Anfang des Speicherbereichs zurück
- Bei Zuweisung in den richtigen Zeiger-Datentyp konvertiert
- Für Berechnung der Grösse `sizeof(datentyp)` verwenden
- Rückgabewert sollte auf NULL überprüft werden (Fehler)

```
// Beispiel: Speicherbereich für 100 Integer anfordern
int *iArray = malloc(100 * sizeof(int));
```

SPEICHER FREIGEBEN: free

```
void free (void *ptr)
```

- Gibt dynamisch allozierten Speicherplatz frei
- Der Zeiger `ptr` zeigt nun auf einen ungültigen Speicherbereich
- Zur Vermeidung von Fehlern ist es empfehlenswert, `ptr` dann auf NULL zu setzen

```
free(iPtr);
iPtr = NULL;
```

SPEICHER AUF 0 SETZEN: calloc

```
void *calloc (size_t num, size_t size)
```

- Reserviert `num` Datenelemente der Grösse `size`
- Setzt die Datenelemente auf 0
- Gibt Pointer auf den Anfang des Speicherbereichs zurück
- Ansonsten wie `malloc`

```
// Beispiel:
// Speicherbereich für 100 int allozieren und initialisieren
int *iar = calloc(100, sizeof(int));
```

GRÖSSE ANPASSEN: realloc

```
void *realloc (void *ptr, size_t size)
```

- Vergrössert/verkleinert bereits allozierten Speicher ab `ptr`
- Neue Grösse *size*
- Bestehender Speicherinhalt wird übernommen
- Wichtig: erzeugt neuen Zeiger

```
contenteditable>
int *iArray;
iArray = malloc(100*sizeof(int));
...
iArray = realloc(iArray, 200*sizeof(int));
```

BEISPIEL 2: ARRAY VON WERTEN

```
int *iArray;
int n = 0;

/* Groesse erst zur Laufzeit bestimmt */
printf("Array Laenge: ");
scanf("%d", &n);

iArray = malloc(n * sizeof(int));

/* Zugriff mit "Klammernotation" */
iArray[95] = 12;

/* Zugriff mit Zeigernotation */
printf("%d\n", *(iArray + 95));
...
free(iArray);
```

BEISPIEL 3: DATENSTRUKTUREN

Auch Structs oder Arrays von Structs können dynamisch alloziert werden:

```
typedef struct {
   int h, min, sec;
} Zeit;

Zeit *pZeit;
pZeit = malloc(sizeof(Zeit));
pZeit->h = 4;
...
free(pZeit);
```

Array von Structs

```
Zeit *pZeit;
pZeit = malloc(10 * sizeof(Zeit));
pZeit[5].h = 4;
...
free(pZeit);
```

BEISPIEL 4: STRINGS

Beim Einlesen ist die Länge des Strings unbekannt:

- Buffer genügend gross wählen
- Länge des Strings bestimmen
- Speicher in passender Grösse reservieren
- String umkopieren

```
char *buffer = malloc(256);
fgets(buffer, 256, stdin);
char *string = malloc(strlen(buffer)+1);
strcpy(string, buffer);
...
free(buffer);
free(string);
```

VARIANTE: EINDIMENSIONALES ARRAY

```
int *iArray;
int rows, cols;
printf("Anzahl rows und cols: ");
scanf("%d %d", &rows, &cols);

iArray = malloc(rows * cols * sizeof(int));
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        iArray[i*cols + j] = i + j;
    }
}
...
free((void *)iArray);
```

VARIANTE: ARRAY VON ZEIGERN

```
int **pArray;
int rows, cols;
printf("Anzahl rows und cols: ");
scanf("%d %d", &rows, &cols);

pArray = malloc(rows * sizeof(int *));
for (int i = 0; i < rows; i++) {
    pArray[i] = malloc(cols * sizeof(int));
}
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        pArray[i][j] = i + j;
    }
}
...
for (int i = 0; i < rows; i++) { free(pArray[i]); }
```

FALSCHE GRÖSSE RESERVIERT

```
int n = 90;
double *ptr = malloc(n*sizeof(n));
...
```

```
// Wo ist der Fehler?

// Platz fuer 10 Datum-Strukturen reservieren
Datum *dat = malloc(10 * sizeof(Datum*));
```

SPEICHER ZU FRÜH FREIGEBEN

- Speicher wird bei erneuter Anforderung anderweitig verwendet
- Zeiger nach Freigabe nicht mehr gültig
- Ratschlag: Pointer nach `free()` auf NULL setzen

```
free(ptr);
ptr = NULL;
```

ZEIGER ÜBERSCHREIBEN OHNE FREIGABE

```
int array[100];
int n = 90;
int *ptr = malloc(n*sizeof(n));
...
ptr = &array[0];
// Problem?
```

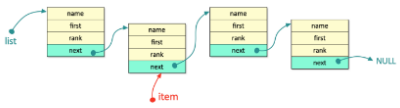
2. **Der Speicher wird reserviert, aber nie freigegeben:**
   - `malloc` reserviert Speicher auf dem Heap.
   - Die ursprüngliche Adresse von `ptr` zeigt auf diesen Speicherbereich.
3. `ptr` **wird überschrieben:**
   - Mit `ptr = &array[0];` wird `ptr` nun auf das **statische Array** umgeleitet.
   - Die vorher zugewiesene `malloc`-Adresse geht verloren.
   - Da `free(ptr)` **vor der Zuweisung fehlt, kann dieser Speicher nicht mehr freigegeben werden → Speicherleck.**

C Verkettete Listen

LISTENELEMENT

```c
typedef struct ListElem_s {

    /* payload */
    char name[16];
    char first[16];
    int rank;

    /* link to next element */
    struct ListElem_s *next;

} ListElem;
```

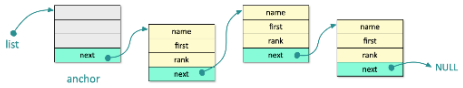
LISTE AUSGEBEN




```c
ListElem *item;
item = head;
while (item != NULL) {
    printf("Name: %s %s\n", item->first, item->name);
    item = item->next;
}
```

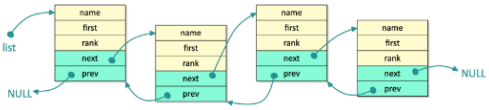
LISTE MIT ANKERELEMENT

- Problem: Einfügen am Anfang des Liste speziell zu behandeln
- Abhilfe: "Dummy"-Element als Kopf der Liste einfügen (Anker)
- Dieses zusätzliche Element enthält keine Daten
- Listenfunktionen müssen aber angepasst werden



DOPPELT VERKETTETE LISTE

- Jedes Element enthält einen Zeiger auf das vorangehende und einen Zeiger auf das nachfolgende Element
- Navigation in beiden Richtungen möglich



Queue (Warteschlange)

- **Prinzip:** FIFO – „First In, First Out"
- **Operationen:**
  - **Enqueue:** Ein Element wird am Ende (Tail) der Warteschlange eingefügt.
  - **Dequeue:** Das Element am Anfang (Head) wird entfernt und zurückgegeben.
- **Verwendung:** Warteschlangen eignen sich beispielsweise zur Aufgabenplanung oder zur Verwaltung von Anfragen, wo das zuerst eingetroffene Element auch zuerst verarbeitet werden soll.
- **Implementierung:**
  Eine Queue kann einfach mit einer verketteten Liste implementiert werden, indem man einen Zeiger auf den Kopf (für das Entfernen) und einen auf das Ende (für das Einfügen) pflegt.

Stack (Stapel)

- **Prinzip:** LIFO – „Last In, First Out"
- **Operationen:**
  - **Push:** Ein Element wird oben auf den Stapel gelegt.
  - **Pop:** Das zuletzt eingefügte Element (das oberste) wird entfernt und zurückgegeben.
- **Verwendung:** Stacks kommen z.B. beim Rückgängigmachen von Operationen (Undo), bei rekursiven Algorithmen oder zur Verwaltung von Funktionsaufrufen im Programm (Call-Stack) zum Einsatz.
- **Implementierung:**
  Ein Stack kann auch mit einer verketteten Liste implementiert werden, wobei man nur am Kopf der Liste Elemente einfügt und entfernt. Das erlaubt effiziente Operationen in konstanter Zeit.

TYPISCHE FEHLER

- Speicher zu früh freigeben
- Falsche Reihenfolge beim Zeiger umhängen
- Speicherfreigabe vergessen

ELEMENT EINFÜGEN

Spezialfall:
Einfügen am Anfang

```c
newItem->next = head;
head = newItem;
anzahl = anzahl + 1;
```

Spezialfall:
Einfügen am Ende

```c
item = head;
/* Listenende suchen */
while (item->next != NULL) {
    item = item->next;
}
item->next = newItem;
newItem->next = NULL;
anzahl = anzahl + 1;
```

a) Einfügen am Anfang

```c
void insertAtBeginning(Node **head, int newData) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = newData;
    newNode->next = *head;
    *head = newNode;
}
```

b) Einfügen am Ende

```c
void insertAtEnd(Node **head, int newData) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->data = newData;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
        return;
    }
    Node *last = *head;
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
}
```

d) Ausgabe der Liste

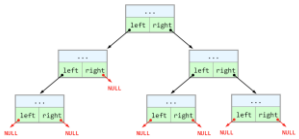
```c
void printList(Node *node) {
    while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
    }
    printf("NULL\n");
}
```

TYPISCHE ANWENDUNG: STACK

- LIFO (Last In First Out)
- Element am Kopf einfügen:
  `push(element)`
- Element am Kopf entnehmen:
  `element = pop()`

BAUMSTRUKTUREN

- Beispiel: Binärbaum
- Schnelles Auffinden gespeicherter Daten



ELEMENT ENTFERNEN

Spezialfall:
Löschen am Anfang

```c
delPtr = head;
head = head->next;
anzahl = anzahl - 1;
free(delPtr);
```

Spezialfall:
Löschen am Ende

```c
item = head;
delPtr = head->next;
while (delPtr->next != NULL) {
    delPtr = delPtr->next;
    item   = item->next;
}
item->next = NULL;
anzahl = anzahl - 1;
free(delPtr);
```

4. Beispielprogramm

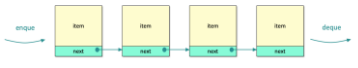
```c
int main() {
    Node *head = NULL;  // Leere Liste

    // Elemente einfügen
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 20);
    insertAtEnd(&head, 30);

    // Liste ausgeben
    printList(head);  // Ausgabe: 20 -> 10 -> 30 -> NULL

    // Element löschen
    deleteNode(&head, 10);
    printList(head);  // Ausgabe: 20 -> 30 -> NULL

    return 0;
}
```

c) Löschen eines Elements

```c
void deleteNode(Node **head, int key) {
    Node *temp = *head, *prev = NULL;

    if (temp != NULL && temp->data == key) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    if (temp == NULL) return; // Element nicht gefunden

    prev->next = temp->next;
    free(temp);
}
```

TYPISCHE ANWENDUNG: QUEUE

- Warteschlange, FIFO (First In First Out)
- Element an einem Ende einfügen:
  `enqueue(element)`
- Element am anderen Ende entnehmen:
  `element = dequeue()`



ARRAYS-KLASSE

- Klasse mit nützlichen Methoden für die Arbeit mit Arrays
- In der Java-Bibiothek vordefiniert
- Zum Beispiel: suchen, sortieren, in String konvertieren

```java
import java.util.Arrays;

public class MyClass {
  public static void main(String[] args) {
    int[] a = { 1, 2, 3, 4, 5, 6 };
    a[1] = 0;
    System.out.println(Arrays.toString(a));
  }
}
```

Java Grundlagen

PROGRAMMIERPARADIGMEN

Problem der Softwareentwicklung:
Mit zunehmender Computerleistung können immer komplexere Programme entwickelt werden (Betriebssysteme, Datenbanken, Bildbearbeitung, etc.)

Dadurch entstehen einige Probleme:
- Überblick behalten
- Programm testen
- Programm warten (Fehlerbehebung, Erweiterung)
- Bestehende Software wieder verwenden

Wie kann die Komplexität gemeistert werden?

OBJEKTORIENTIERTE PROGRAMMIERUNG

- Daten und Prozeduren (Verarbeitung der Daten) werden in *Objekten* zusammengefasst
- Verhalten des Programms entsteht durch das Zusammenspiel dieser Objekte
- OOP – Object Oriented Programming

JAVA

- James Gosling (Sun Microsystems), 1990-1995
- Ursprünglich für Heimgeräte gedacht (Video, Telefon)
- Anforderungen: klein, zuverlässig, plattformunabhängig
- Aufschwung von Internet/WWW: Sprache gesucht, die plattformunabhängig und sicher ist
- Bedarf für Plattformunabhängigkeit: schnelle Verbreitung
- Heute Sprache, Werkzeuge, umfangreicher Satz von APIs und Bibliotheken für verschiedene Anwendungsbereiche

VORTEILE VON OOP

- Nähe zum Problembereich: erleichtert Kommunikation zwischen Entwicklern und Experten
- Objektorientierte Modelle sind änderungsfreundlicher, da Änderungen häufig lokal begrenzt möglich sind
- Abstraktionsmöglichkeiten ermöglichen bessere Wiederverwendbarkeit

Aufbau der Java-Technologie

Programmier-sprache	*Java Quelltext (.java)*
JDK	Entwicklungswerkzeuge Java-Compiler, …
	Java Bytecode (.class, .jar)
JRE	Java Programmierschnittstelle (API)
	Java Virtual Machine (JVM) mit Just-in-time-Kompilierung
Betriebs-system	Windows, Linux, Solaris, Mac OS X, …

ANWEISUNGEN UND KOMMENTARE

```
/**
 *  Beispiel
 */
public class HelloYouThere {
    public static void main(String[] args) {
        // Ausgabe auf drei Zeilen
        System.out.println("Hello");
        System.out.println ("you");
        System.out.println ("there");
    }
}
```

- Kommentare wie in C
- Anweisungen enden mit Semikolon
- Anweisungen werden der Reihe nach ausgeführt
- Blocks in `{ ... }` eingeschlossen

PLATTFORM-UNABHÄNGIG

- Problem: jeder Prozessor hat seine eigene Maschinensprache
- Ein Programm muss für jeden Prozessor in die entsprechende Maschinensprache übersetzt (kompiliert) werden
- Java simuliert einen Prozessor mit einem Programm, der Java Virtual Machine (JVM)
- Maschinensprache der JVM ist der so genannte Bytecode

VARIABLENNAMEN

- Erstes Zeichen: Buchstabe (Unicode), `_` oder `$`
- Nachfolgende Zeichen: auch Ziffern möglich
- Keine reservierten Wörter erlaubt (`class`, `public`, ...)
- Gut: `breite`, `LÄNGE`, `jahr2000`, `b$3_n`
- Fehler: `neues Haus`, `1Ka`

Java Datentypen

DATENTYPEN: GANZE ZAHLEN

- Vier Ganzzahl-Typen mit unterschiedlichen Wertebereichen:
  - `byte` (8 Bit)
  - `short` (16 Bit)
  - `int` (32 Bit)
  - `long` (64 Bit)
- Unterschied zu C?
- Im Normalfall `int` verwenden
- Literale z.B.: 127, -1214, 104L (long), 0x6B (hexadezimal)

DATENTYPEN: ZEICHEN

- Einfacher Datentyp für einzelne Zeichen: `char`
- Literale mit einfachen Anführungsstrichen: `'a'`, `'T'`, `'3'`, `' '`, `'@'`, `'.'`, `'ü'`
- Länge: 16 Bit (Unicode)
- Unicode 0x0000 bis 0x007F entsprechen dem ASCII-Code
- Spezielle Zeichen: `'\n'`, `'\t'`, `'\"'`, `'\''`, `'\\'`
- Unicode-Zeichen mit Hexcode xxxx: `'\uxxxx'`

DATENTYPEN: FLIESSKOMMAZAHLEN

- Zwei Fliesskomma-Typen
  - `float` (32 Bit)
  - `double` (64 Bit)
- Im Normalfall `double` verwenden
- Literale z.B.: 1.34, 1.0e5, 1.34f (float)

Typ	Grösse [Bits]	Max.wert	Min.wert	Präzision
float	32	± 3.4e+38	±1.4e-45	6-7 Dez.Stellen
double	64	± 1.79e+308	±4.9e-324	14-15 Stellen

DATENTYPEN: STRINGS

- String-Literal: `"Dies ist ein String"`
- Vordefinierte Klasse: `String`
- Kein char-Array mit `'\0'` am Ende wie in C
- Mehr zu Strings in einer anderen Lektion

Typ	Grösse [Bits]	negativer Maximalwert	positiver Maximalwert
byte	8	-128	127
short	16	-32768	32767
int	32	-2147483648	2147483647
long	64	-9223372036854775808	9223372036854775807

DATENTYPEN: LOGISCHE WERTE

- Datentyp `boolean`
- Umfasst nur die Werte `true` und `false`
- Unterschied zu C?

```
boolean ready = false;
boolean ok = true;
```

Java Kontrollstrukturen

This

```
class Datum {
    private int tag;
    private int monat;
    private int jahr;

    public void setze (int tag, int monat, int jahr) {
        this.tag = tag;
        this.monat = monat;
        this.jahr = jahr;
    }

    public void drucke () {
        System.out.println(tag + "." + monat + "." + jahr);
    }
}
```

KONSTRUKTOR

- Ein Konstruktor ist eine spezielle Methode einer Klasse
- Automatisch aufgerufen, wenn ein Objekt erzeugt wird
- Er hat den gleichen Namen wie die Klasse
- Er hat keinen Resultattyp
- Der Zugriffsmodifikator ist normalerweise `public`
- Falls man keinen Konstruktor definiert, fügt Java automatisch einen Default-Konstruktor ohne Parameter in die Klasse ein

```
public Datum () {}    // Default-Konstruktor
```

- Verzweigungen:

```java
// if-else
if (note >= 5) {
    System.out.println("Bestanden!");
} else {
    System.out.println("Durchgefallen");
}

// switch-case
switch (farbe) {
    case "rot": System.out.println("Stopp!"); break;
    case "grün": System.out.println("Gehen!"); break;
    default: System.out.println("Unbekannt");
}
```

Java String, Methoden und Arrays

Strings

- Klasse `String` aus der Java-Bibliothek
- Strings sind unveränderbar (immutable)
- Bei Änderung wird daher ein neuer String erzeugt
- Bearbeitbare Alternative: `StringBuffer`
- Methode `length()` liefert die String-Länge
- Vergleich mit `equals()` oder `equalsIgnoreCase()`
- Diese Methoden liefern `true` oder `false` (boolean)
- `compareTo()` macht einen lexikalischen Vergleich: Ergebnis < 0, wenn String lexikalisch vor Argument
- Strings nicht mit `==` oder `!=` vergleichen

```
int i    = Integer.parseInt("25");
long l   = Long.parseLong("2342322342");
float f  = Float.parseFloat("23.453");
double d = Double.parseDouble("45.786838091");
```

- Methoden:

```java
// Methode definieren
public static int summe(int a, int b) {
    return a + b;
}

// Aufruf
int ergebnis = summe(3, 4); // 7
```

- Arrays:

```java
// Deklaration & Initialisierung
int[] zahlen = new int[3]; // {0,0,0}
String[] namen = {"Anna", "Bob", "Clara"};

// Zugriff
zahlen[0] = 10;
System.out.println(namen[1]); // "Bob"

// Länge
int laenge = namen.length; // 3
```

Java Objekte und Klassen

- **Klasse:** Bauplan für Objekte (Attribute + Methoden).

```java
public class Auto {
    // Attribute
    private String marke;

    // Konstruktor
    public Auto(String marke) {
        this.marke = marke;
    }

    // Methode
    public void fahren() {
        System.out.println(marke + " fährt!");
    }
}
```

- **Objekt:** Instanz einer Klasse.

```java
Auto meinAuto = new Auto("VW");
meinAuto.fahren();  // Ausgabe: "VW fährt!"
```

- `this`: Referenziert aktuelles Objekt.
- **Zugriffsmodifikatoren:**
  - `public`: Zugriff überall
  - `private`: Nur innerhalb der Klasse
  - `protected`: Innerhalb des Pakets + Unterklassen

Zugriffsmodifikatoren(class):
public = Überall sichtbar
protected = Sichtbar gleichen Paket & Unterklassen
private = nur für dieselben Klasse sichtbar

Für Methoden, Variabeln und Klassen:
static = Klassenweit def.(nicht für äussere Klassen)
final = nicht erweiterbar/veränderbar
abstract = nicht Instanzierbar (nur Klasse & Methode)

```java
public class Auto { // Klasse

    String farbe; // Attribut

    public Auto(String f) { farbe = f; } // Konstruktor

    void hupen() { System.out.println("Tuuut! Farbe: " + farbe); } // Methode

    public static void main(String[] args) {
        Auto a = new Auto("Blau"); // Objekt erstellen
        a.hupen(); // Methode aufrufen
    }
}
```

Java Grafiken und Ereignisse

`JFrame frame=new JFrame("Titel"); frame.setSize(400,400);`
`frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); frame.setVisible(true);`
Komponenten: `JButton`, `JLabel`, `JTextField`, `JPanel` (Container für mehrere Elemente)
Layouts: `frame.setLayout(new FlowLayout());`

Farbe setzen

`g.setColor(Color.RED);`
`Color c=new Color(124,32,45);`

- Wichtige Komponenten:
  - `JFrame`, `JPanel`, `JButton`, `JLabel`
- Layout-Manager: `BorderLayout`, `GridLayout`

Zeichnen

- Linie: `g.drawLine(x1,y1,x2,y2);`
- Text: `g.drawString("Text", x, y);`
- Rechteck: `g.drawRect(x,y,w,h);`
- Ellipse: `g.drawOval(x,y,w,h);`
- Polygon: `g.drawPolygon(int[] x, int[] y, n);`
- Füllen: `fillRect()`, `fillOval()`, `fillArc()`

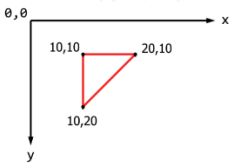
```java
import javax.swing.*;      // für GUI-Komponenten
import java.awt.event.*;   // für ActionListener

public class MeinFenster extends JFrame implements ActionListener {

    JButton button;

    public MeinFenster() {
        button = new JButton("Klick mich"); // Button erstellen
        button.addActionListener(this);     // ActionListener hinzufügen

        add(button);                        // Button zum Fenster hinzufügen
        setSize(200, 100);                  // Fenstergröße
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);                   // Fenster sichtbar machen
    }

    // Aktion bei Button-Klick
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button wurde geklickt!");
    }

    public static void main(String[] args) {
        new MeinFenster(); // Fenster erstellen
    }
}
```

GRAFIK: POLYGON

`drawPolygon (int[] x, int[] y, int nPoints);`

- Arrays mit x- und y-Koordinaten der Punkte, Anzahl Punkte
- Beispiel:

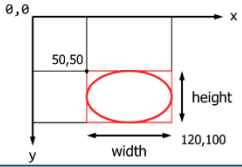
```
int x[] = { 10, 20, 10 };
int y[] = { 10, 10, 20 };
drawPolygon(x, y, 3);
```



GRAFIK: KREIS, ELLIPSE

`drawOval (int x, int y, int width, int height)`

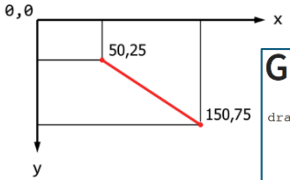
- Parameter → "bouding box"
- Beispiel: `drawOval(50, 50, 70, 50)`:



GRAFIK: LINIE

`drawLine (int xStart, int yStart, int xEnd, int yEnd)`

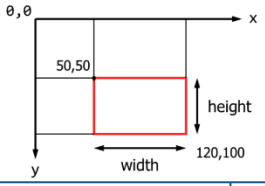
- Argumente: Achtung Reihenfolge beachten
- Beispiel: `drawLine(50, 25, 150, 75)`:



GRAFIK: RECHTECK

`drawRect (int x, int y, int width, int height)`

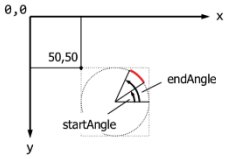
- Startpunkt und Dimensionen
- Beispiel: `drawRect(50, 50, 70, 50)`:



GRAFIK: KREISSEGMENT

`drawArc (int x, int y, int w, int h, int sAngle, int eAngle)`

- 0 Grad → 3 Uhr, Winkel im Gegenuhrzeigersinn
- Beispiel: `drawArc(50, 50, 75, 75, 30, 30)`:



GRAFIK: TEXT

`drawString (String str, int x, int y)`

- Koordinaten → Basislinie
- Beispiel: `drawString("Hello World", 50, 50)`:

`Color c = new Color(124, 32, 45)`

`fillRect(); fillOval(); fillArc();`

Java Vererbung

- Vererbung: `extends` für Klassen, `implements` für Interfaces

```java
public class ElektroAuto extends Auto {
    public ElektroAuto(String marke) {
        super(marke);  // ruft Eltern-Konstruktor
    }

    @Override
    public void fahren() {
        System.out.println("Leise elektrisch fahren!");
    }
}
```

- Polymorphie:

```java
Auto auto = new ElektroAuto("Tesla");
auto.fahren();  // Ruft überschriebene Methode auf
```

- Abstrakte Klassen:

```java
public abstract class Tier {
    public abstract void machenGeraeusch();
}
```

- Interfaces (ab Java 8 mit Default-Methoden):

```java
public interface Ladekabel {
    default void lade() {
        System.out.println("Ladevorgang startet...");
    }
}
```

Interfaces:
- Haben leere Methoden
- «implements» um einzubinden
- Mehr als 1 möglich

```
interface Tier { void essen(); }
interface Beweglich { void bewegen(); }

class Hund implements Tier, Beweglich {
    public void essen() { System.out.println("Frisst"); }
    public void bewegen() { System.out.println("Läuft"); }
}
```

```
// Klasse Zoo als Container
public class Zoo {

    // Oberklasse Tier
    public static class Tier {
        public String name;

        // Konstruktor
        public Tier(String name) { this.name = name; }

        // Standardgeräusch
        public void machenGeräusch() {
            System.out.println("Ein Tier macht ein Geräusch");
        }
    }

    // Unterklasse Hund
    public static class Hund extends Tier {
        public Hund(String name) { super(name); } // Konstruktor ruft Oberklasse auf

        @Override
        public void machenGeräusch() {
            System.out.println("Wau Wau");
        }
    }

    // Hauptprogramm
    public static void main(String[] args) {
        Hund meinHund = new Hund("Bello"); // Hund-Objekt erstellen
        meinHund.machenGeräusch();         // ruft überschriebene Methode auf
    }
}
```

Java Applications

Frames und Window events:
// Klasse: JFrame
// Konstruktoren

JFrame()	// Fenster ohne Titel
JFrame(String title)	// Fenster mit Titel
// Wichtige Methoden	
setSize(int w, int h)	// Fenstergrösse setzen
setLocation(int x, int y)	// Position des Fensters (Def: 0,0)
show()	// Fenster -> sichtbar
setVisible(boolean b)	// Fenster -> sichtbar / unsichtbar
toBack()	// Fenster nach hinten schieben
toFront()	// Fenster nach vorne schieben
validate()	// Fensterelemente neu anordnen
setTitle(String title)	// Fenstertitel setzen
setMenuBar(MenuBar m)	// Menuleitste setzen

WindowListener (Fenster schließen):

```java
f.addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
});
```

ActionListener → z.B. für Button:

```java
b.addActionListener(e -> System.out.println("Geklickt!"));
```

```
import javax.swing.*;
import java.awt.event.*;

public class MiniApp extends JFrame {
    public MiniApp() {
        JButton b = new JButton("Klick mich");
        b.addActionListener(e -> JOptionPane.showMessageDialog(this, "Hallo Welt!"));
        add(b); setSize(200, 100); setDefaultCloseOperation(EXIT_ON_CLOSE); setVisible(true);
    }

    public static void main(String[] args) {
        new MiniApp();
    }
}
```

- `FlowLayout()` – nebeneinander, Zeile für Zeile
- `BorderLayout()` – NORTH, SOUTH, WEST, EAST, CENTER
- `GridLayout(2, 2)` – 2x2 Gitter
- `null` → manuelle Platzierung mit `setBounds(x, y, w, h)`

```
JButton b = new JButton("Klick");
JLabel l = new JLabel("Text");
JTextField tf = new JTextField(10);
JCheckBox cb = new JCheckBox("Auswahl");
JRadioButton rb = new JRadioButton("Option");
JComboBox box = new JComboBox(); box.addItem("A");
```

Java Dateien und Streams

- **Dateien lesen** (Java NIO):

```java
List<String> lines = Files.readAllLines(Paths.get("datei.txt"));
```

- **Streams** (Java 8+):

```java
List<String> filtered = lines.stream()
                             .filter(s -> s.contains("Java"))
                             .collect(Collectors.toList());
```

Schreiben in Textdatei

```java
PrintWriter out = new PrintWriter(new FileWriter("ausgabe.txt"), true);
out.println("Hallo Welt");
out.close();
```

- **Try-with-Resources** (automatisches Schließen):

```java
try (BufferedReader br = new BufferedReader(new FileReader("datei.txt"))) {
    String line = br.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

Lesen von Textdateien (Zeichenweise)

```
BufferedReader in = new BufferedReader(new FileReader("datei.txt"));
String zeile;
while ((zeile = in.readline()) != null) { ... }
in.close();
```

File-Class

```
File f = new File("C:\\pfad\\datei.txt");
f.exists(); f.isFile(); f.isDirectory(); f.getName(); f.delete();
```

Java Exceptions

- **Try-Catch:**

```java
try {
    int x = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Fehler: " + e.getMessage());
} finally {
    System.out.println("Wird immer ausgeführt");
}
```

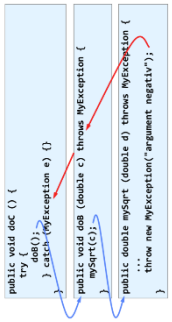
- **Eigene Exception:**

```java
public class MeineException extends Exception {
    public MeineException(String message) {
        super(message);
    }
}
```

- **Werfen von Exceptions:**

```java
public void check() throws MeineException {
    throw new MeineException("Fehler!");
}
```

Exception kann weitergegeben werden
- Muss im Methodenkopf angegeben werden



Unchecked Exceptions
- Können vom Programm behandelt werden
- Andernfalls werden sie an die VM weitergereicht
- Erben von `RuntimeException`
- Beispiele: `NullPointerException`, `NumberFormatException`, `ArrayIndexOutOfBoundsException`

Checked Exceptions
- Müssen vom Programm behandelt werden
- Erben von `Exception`
- Beispiele: `IOException`, `ClassNotFoundException`