Softwareentwicklung in C
Teil2

Markus Zuber Harald Glock

28. April 2003

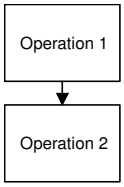
Inhaltsverzeichnis

1. Verfeinerung von Algorithmen und Datenstrukturen

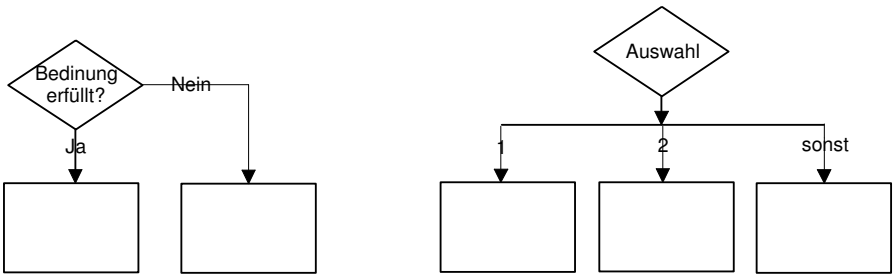
1.1. Programmablaufplan (PAP)

Der Programmablaufplan nach DIN66001 dient, wie auch die Struktogramme, zur Darstellung eines Algorithmus. Durch die Einführung des Struktogramms hat die Bedeutung des Ablaufplans allerdings abgenommen. Er besteht aus verschiedenen geometrische Formen, die über Pfeile oder Linien miteinander verbunden sind. Ein Vorteil des Programmablaufplans ist die Möglichkeit, zu jeder Operation die Ein- und Ausgabedaten mit anzugeben. Dadurch wird der Ablaufplan zu einem so genannten *Programmnetz* erweitert.
Im Folgenden nun die wichtigsten Sinnbilder des Programmablaufplans:

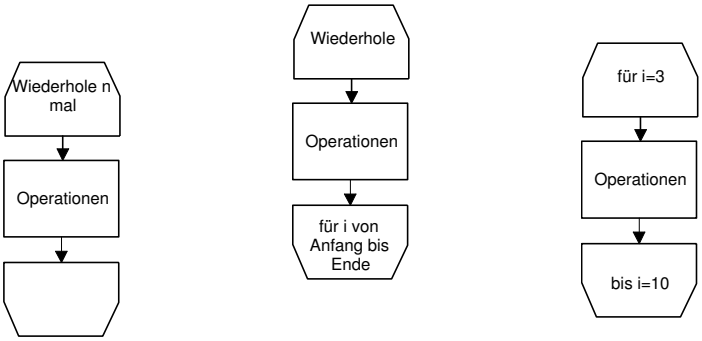
1.1.1. Block, Sequenz, Operation



1.1.2. Verzweigung

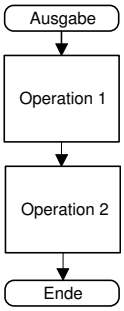


1.1.3. Schleifenbegrenzung



1.1.4. Unterprogrammaufruf

1.1.5. Anfang/Ende des Ablaufs

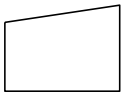


1.2. Datenflussplan

Ein Datenflussplan ist eine grafische Übersicht, welche die Programme und Daten, die zu einer Gesamtaufgabe gehören, miteinander verbindet. Er zeigt, welche Teile eines Programms von den Daten durchlaufen werden und welche Art der Bearbeitung innerhalb der Programme vorgenommen wird. Ein Datenflussplan besitzt ähnliche Symbole wie ein Programmablaufplan. Zusätzliche Sinnbilder werden vor allem für die Daten eingeführt.

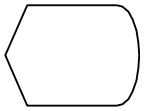
1.2.1. manuelle Eingabedaten

Daten, die über optische Erfassung (z.B. Kamera, Leser) oder Tastatur eingegeben werden.



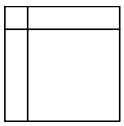
1.2.2. erzeugte Ausgabedaten

Daten die über eine optische (z.B. Bildschirm) oder akustische Ausgabeeinheit ausgegeben werden.



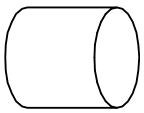
1.2.3. Daten im Speicher

Daten, die sich im Arbeitsspeicher befinden.

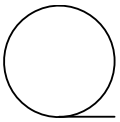


1.2.4. permanenter Speicher

Daten, die sich auf einer Datenträgereinheit mit direktem Zugriff befinden (z.B. Festplatte, Diskette).

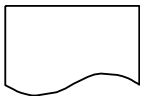


Daten, die sich auf einer Datenträgereinheit mit sequentiellen (fest vorgeschriebene Reihenfolge) Zugriff befinden (z.B. Magnetband).



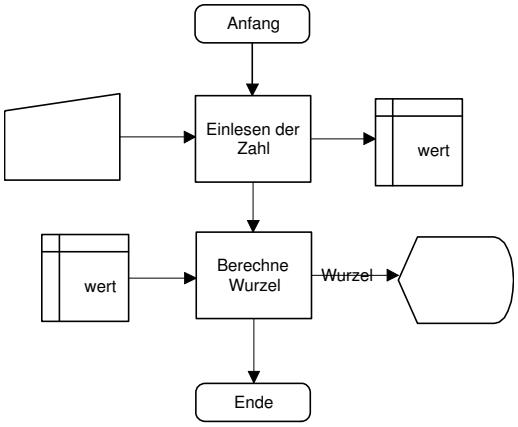
1.2.5. Daten auf Schriftstück

Daten, die sich auf Papier oder ähnlichem befinden (z.B. Ausdrucke)

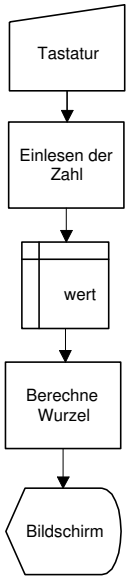


1.3. Programmnetz

Das Programmnetz vereint nun den Programmablaufplan mit dem Datenflussplan.

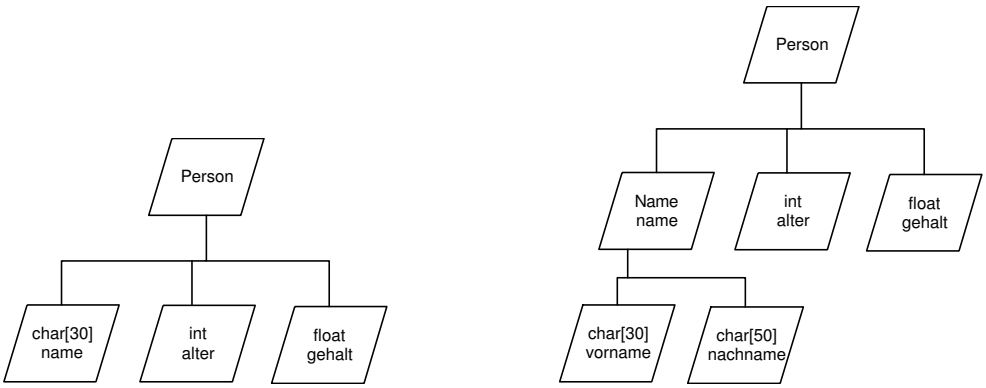


Die erste Operation liest eine Zahl in die Variable *wert* ein, welche sich im Arbeitsspeicher befindet. Die zweite Operation berechnet die Wurzel dieser Variablen und gibt sie auf dem Bildschirm aus.
Der reine Datenflussplan würde folgendermaßen aussehen:



1.4. Datenhierarchie

Zur Darstellung zusammengesetzter Datentypen, wie z.B. Strukturen, wird das so genannte *Datenhierarchie-Diagramm* nach DIN 66001 verwendet.



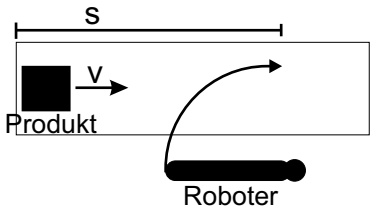
Eine Komponente wird durch ein Parallelogramm repräsentiert. Jede Komponente hat einen Datentyp und einen Namen, welche beide in das Parallelogramm eingetragen werden. Die Zusammensetzung wird durch eine Verbindungslinie dargestellt.

1.5. Entwurf eines Algorithmus

Eine Lösung bzw. einen Algorithmus für eine gestellte Aufgabe findet sich selten in einem Schritt. Vielmehr wird wohl zuerst eine grobe Vorstellung von der Vorgehensweise existieren. Diese wird dann in mehreren Schritten immer weiter verfeinert, bis sie zuletzt in ein Programm umgesetzt werden kann (**Top-Down**-Methode). Die Verfeinerung ist dabei erst abgeschlossen, wenn der Entwurf direkt in ein Programm umgesetzt werden kann. Er muss daher alle Informationen enthalten, damit ein Programmierer ohne eigene Interpretationen das Programm realisieren kann.
In den folgenden Beispielen arbeiten wir nur mit Programmablaufplänen. Obwohl die Ablaufpläne nicht direkt den Grundgedanken der strukturierten Programmierung entsprechen (es sind auch „Sprünge“ möglich), bieten sie im Gegensatz zu den Struktogrammen die Möglichkeit, direkt die Ein-/Ausgabedaten an den einzelnen Operationen einzutragen.

1.5.1. Beispiel: Förderband

Auf einem Förderband befindet sich ein Produkt. Das Band läuft mit der konstanten Geschwindigkeit v. Neben dem Förderband, im Abstand s vom Beginn des Bandes, steht ein Roboter, welcher das Produkt vom Band nehmen und in eine Verpackung einlegen soll.



Gesucht ist nun ein Programm für den Roboter, welches aus dem vorgegebenen Winkel des Roboterarms, der Geschwindigkeit des Bandes und des Abstandes des Produkts vom Roboter die Winkelbeschleunigung für den Roboter berechnet.

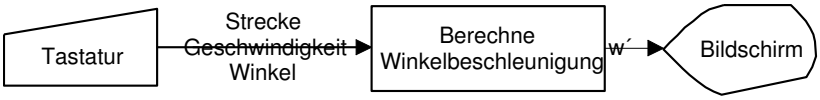
Zur Berechnung werden die physikalischen Grundlagen für die gleichförmige Bewegung und eine beschleunigte Winkelbewegung benötigt:

$$v = \frac{s}{t} \qquad \alpha = \frac{1}{2} \cdot \dot{\omega} \cdot t^2$$

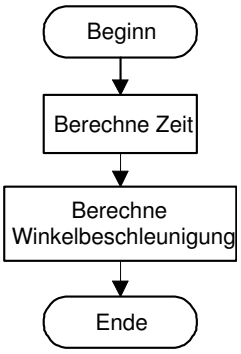
Algorithmus in Worten

1. Lese die Geschwindigkeit v, die Strecke s und den Startwinkel α ein
2. Berechne die Zeit t, in der das Produkt die Strecke s zurückgelegt hat
3. Berechne aus der Zeit t und dem Winkel α die Winkelbeschleunigung $\dot{\omega}$
4. Gebe die Winkelbeschleunigung aus

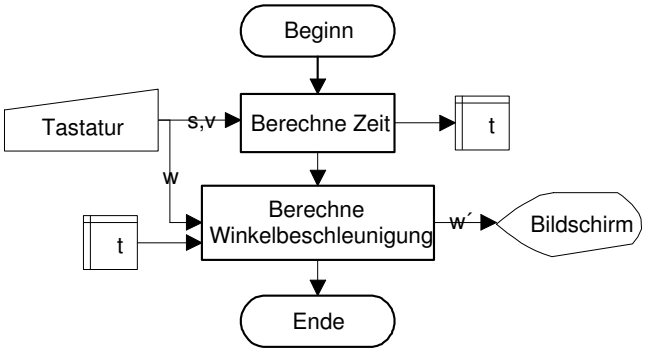
EVA-Charakter des Algorithmus



grober Entwurf des Algorithmus im Programmablaufplan

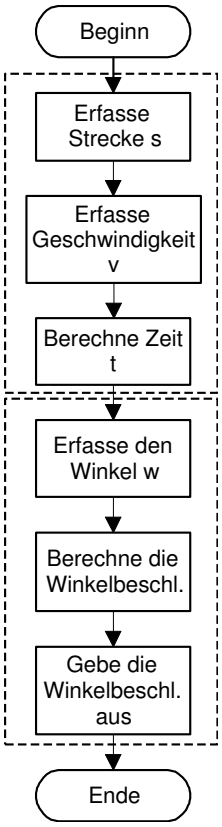


grobes Programmnetz des Algorithmus

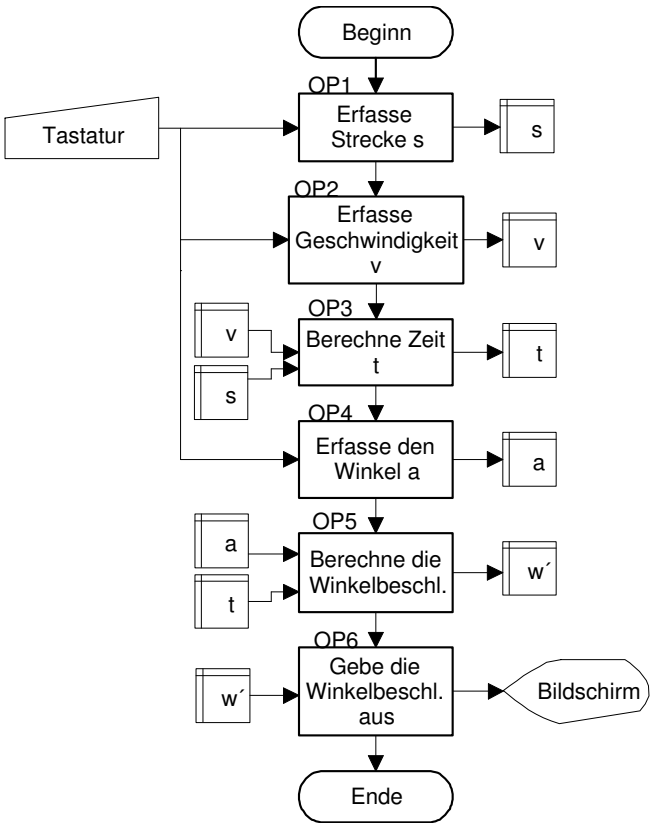


Verfeinerung des Algorithmus

Die obige Struktur ist noch viel zu grob, um direkt in eine Hochsprache umgesetzt zu werden. Die beiden Blöcke „Berechne Zeit“ und „Berechne Winkelbeschleunigung“ können noch weiter unterteilt werden. Dazu wird jeder Block separat in einzelne, kleine Schritte unterteilt. Diese Unterteilung vom 1. groben Entwurf in eine 2. detaillierteren Entwurf kann auch dazu verwendet werden, das Programm in einzelne Unterprogramme (Funktionen, Module) zu unterteilen. So könnte es hier am Beispiel später eine Funktion „BerechneZeit()“ und eine Funktion „BerechneBeschleunigung()“ geben.

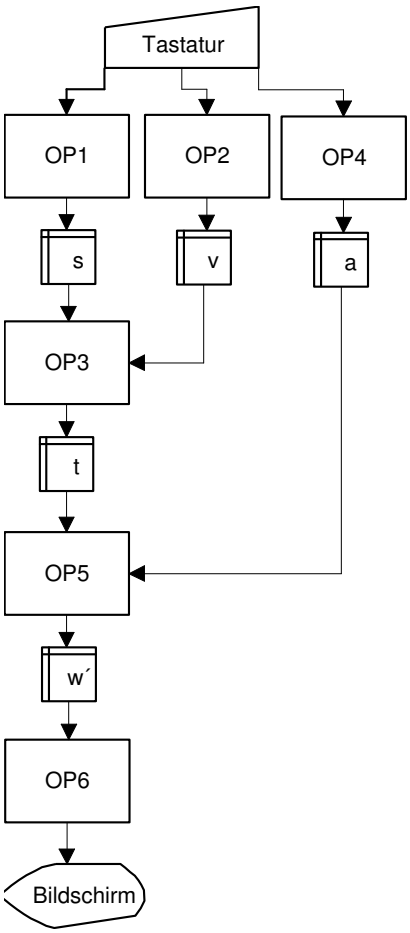


Zusammenhängende Operationen können auch mit einer gestrichelten Linie sichtbar zusammengefasst werden.
Aus dem verfeinerten Programmablaufplan ergibt sich jetzt auch ein neues Programmnetz. Hier werden alle Daten, die von den einzelnen Operationen verarbeitet werden, eingetragen. So erhält man einen schnellen Überblick, wo die Daten verarbeitet werden.



Datenflussplan

Aus dem obigen Programmnetz lässt sich jetzt auch ein detaillierterer Datenflussplan erstellen. Der erste Datenflussplan (siehe 1.5.1) war noch zu abstrakt, um einen wirklichen Einblick zu geben. Eine Verfeinerung oder ein Hierarchiediagramm der Daten ist hier nicht notwendig, da es sich um einfache Datentypen handelt, die nicht aus mehreren Komponenten bestehen.



Der Datenflussplan zeigt, welchen Ein/Ausgabedaten eine Operation hat und umgekehrt, welche Operationen auf ein Datum zugreifen.

Dieser Entwurf enthält nun alle relevanten Informationen um in ein Programm umgesetzt zu werden. Jede Operation kann nun direkt programmiert werden, ohne dass sich der Programmierer alle vorangegangenen Gedanken mach muss.

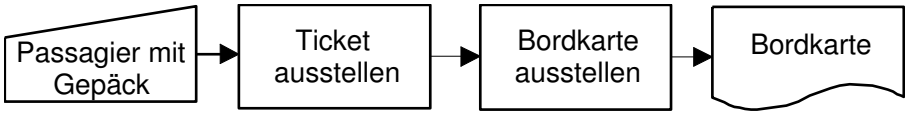
1.5.2. Beispiel: Flughafen

Am Flughafen hat der Passagier die Möglichkeit, ein Ticket für einem bestimmten Zielort zu kaufen. Dazu muss er an der Kasse den Betrag für das Ticket bezahlen. Er erhält das Ticket nur, wenn es noch freie Plätze im Flugzeug gibt. Zur Ausstellung des Tickets wird der Name des Passagiers benötigt. Auf dem Ticket sind die Passagierdaten, der Zielort, die Flugnummer und die Flugzeit vermerkt.
Nachdem er das Ticket besitzt, kann er beim Check-In sein Gepäck aufgeben. Jedes der Gepäckstücke hat ein bestimmtes Gewicht. Das Gesamtgewicht der Gepäckstücke wird auf einem Bildschirm ausgegeben. Der Passagier gibt sein Ticket ab und erhält seine Bordkarte mit den Ticketdaten, seinem Namen, der Anzahl aufgegebener Gepäckstücke und das Gesamtgewicht.

Algorithmus in Worten

1. Fordere Ticket zu einem Ziel an. Fordere dazu den Zielort an.
2. Suche ob noch Plätze im Flugzeug frei.
3. wenn noch Plätze frei, dann stelle das Ticket mit Namen, Zielort, Flugnummer und Flugzeit aus. Lese die benötigten Daten dazu ein.
4. Gebe maximal 2 Gepäckstücke ab.
5. Ermittle das Gesamtgewicht aus den 2 Gepäckstücken und gebe es aus.
6. Stelle die Bordkarte mit Ticketdaten (siehe oben), Passagierdaten (im Ticket), Anzahl der Gepäckstücke und Gesamtgewicht aus.

EVA-Charakter des Algorithmus

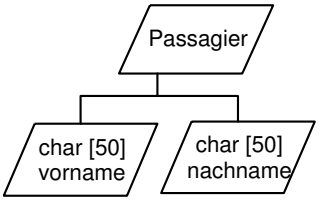


Ausarbeitung der Datenhierarchie

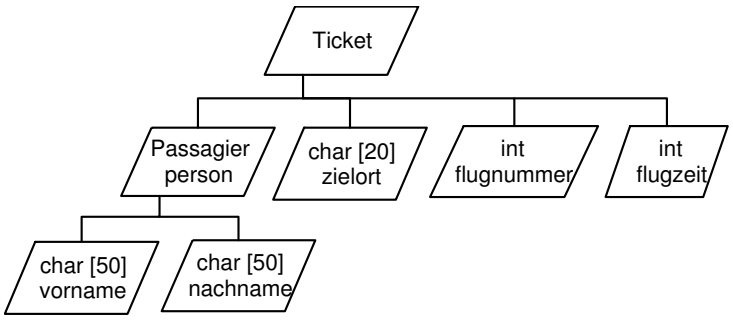
Durch die Aufgabenstellung ergeben sich drei komplexere Datentypen:

1. Passagier: Besitzt einen Namen, der wiederum aus Vor- und Nachname besteht
2. Ticket: Beinhaltet die Passagierdaten und dem Zielort, Flugnummer und Flugzeit
3. Bordkarte: beinhaltet alle Ticketdaten und Gepäckstücke und Gesamtgewicht.

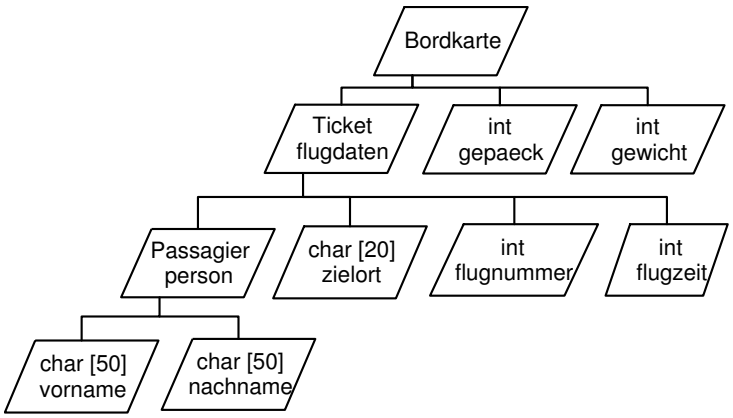
Der Datentyp „Passagier„ besteht aus zwei Zeichenketten, die den Vor- und Nachnamen enthalten:



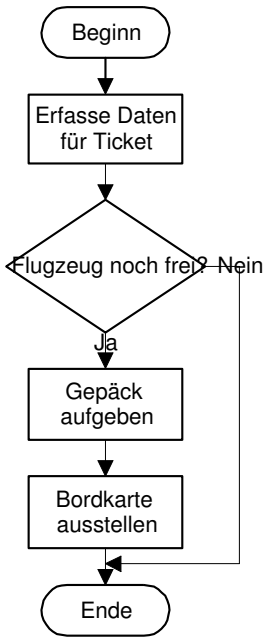
Der Datentyp „Ticket“ beinhaltet die Passagierdaten. Die lässt sich erreichen, indem eine Komponente des Datentyps vom Typ „Passagier“ ist. Der Zielort ist ebenfalls eine Zeichenkette. Flugnummer und Flugzeit sind Ganzzahlen und müssen daher nicht weiter untergliedert werden:



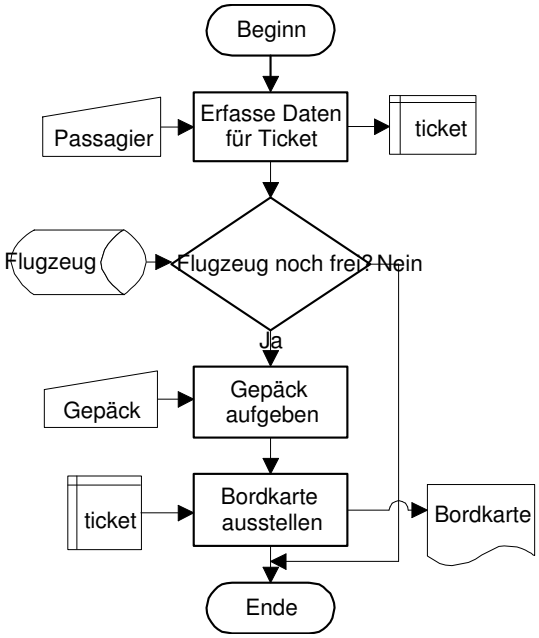
Der letzte Typ „Bordkarte“ setzt sich aus den Ticketdaten (mit den Passagierdaten) und den Informationen über die Gepäckstücke zusammen:



grober Entwurf des Algorithmus



grobes Programmnetz des Algorithmus

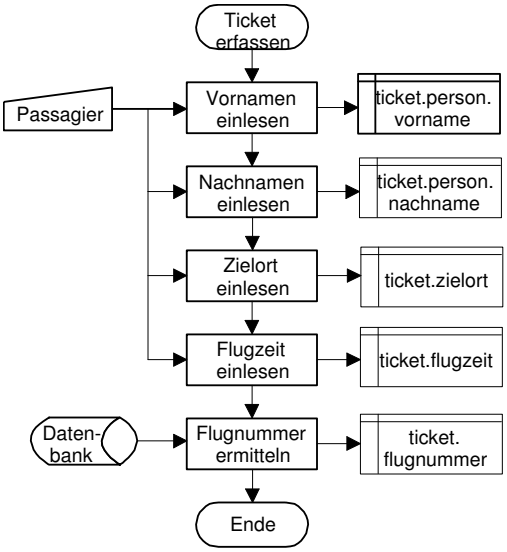


Die Daten für die freien Plätze im Flugplatz sollen hier auf einem permanenten Datenspeicher mit direktem Zugriff liegen (z.B. der Server der Fluggesellschaft)

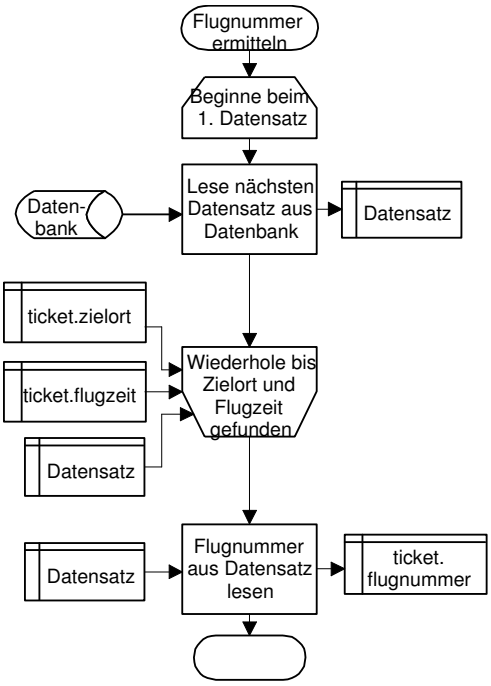
Verfeinerung des Algorithmus

Im folgenden wollen wir nur noch die Ablaufpläne für die einzelnen Funktionen des Algorithmus betrachten, damit das Ganze nicht zu unübersichtlich wird. Jede der oben bezeichneten Operationen (Blöcke) werden wir in ein Unterprogramm bzw. Funktion packen.

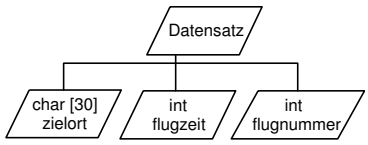
Daten für Ticket erfassen



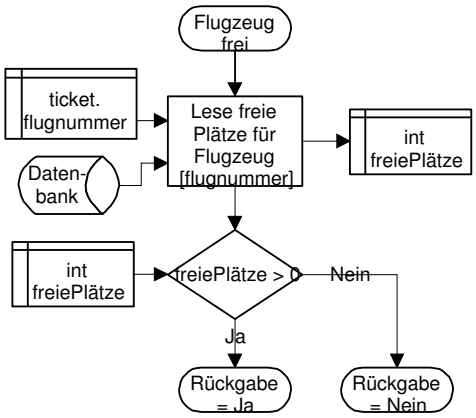
Die Operation „Flugnummer ermitteln“ lässt sich in dieser Verfeinerung noch nicht direkt Programmieren. Dies bedeutet, dass hier noch eine weitere Verfeinerung vorgenommen werden muss:



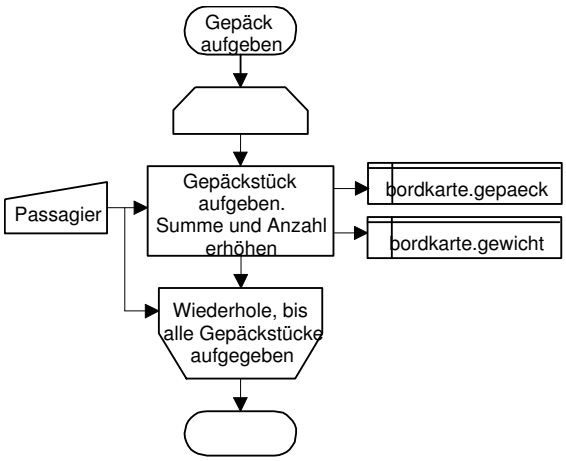
Der gelesene Datensatz beinhaltet für alle Flüge den Zielort, die Zielzeit und die dazugehörige Flugnummer:



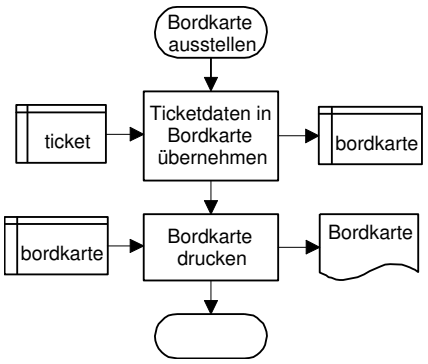
Abfrage ob noch Plätze frei



Gepäck aufgeben



Bordkarte ausstellen



1.5.3. Zusammenfassung

Grundsätzlich lässt sich die Dokumentation eines Algorithmus in folgenden Schritten erstellen:

1. Beschreibung des Algorithmus in Worten
2. Darstellung des Algorithmus durch einen (zunächst grob) PAP / Struktogramm
3. Beschreibung der relevanten Datentypen und Datenobjekte
4. Darstellung der zusammengesetzten Datentypen (Strukturen) durch Datenhierarchien (DH)
5. Wenn nötig, Verfeinerung des groben PAP's. Zur besseren Strukturierung wird, später im Code, jede Verfeinerung in einer eigenen Funktion implementiert.

Die Verfeinerung sollte erst dann aufhören, wenn der Programmablaufplan/Struktogramm direkt in ein Programm codiert werden kann. Für die korrekte Formulierung des Algorithmus sind daher unbedingt auch Programmierkenntnisse in einer Hochsprache erforderlich!

2. Zeiger

Dieses Kapitel behandelt die Zeiger (engl. Pointer) in C. Erfahrungsgemäß ist das Arbeiten mit Zeigern keine einfach Aufgabe und kann nur durch viel Übung gemeistert werden. Zeiger werden vor allem für „Call by reference“, also Parameterübergabe an Funktionen, sowie die Verwaltung von dynamisch allokiertem Speicher benötigt. Zeiger ermöglichen es auf flexible Art und Weise beliebige Speicheradressen zu adressieren (auch Unzulässige ==> Programmabsturz!).

2.1. Zeiger und Adressen

Ein Zeiger ist eine Variable, die die Speicheradresse einer anderen Variable (oder allokiertem Speicher) enthält. Der Zeiger „zeigt“ also auf eine andere Variable, er referenziert sie. Zeiger in C müssen immer vom Typ der Variablen sein, auf die sie Zeigen. Um die Adresse einer Variable in C zu erhalten, kann der &-Operator benutzt werden. Zeigervariablen selbst werden durch zusätzliche Angabe eines $*$ beim Datentyp deklariert.
Da ein Zeiger selbst eine Variable ist kann sein Inhalt jederzeit neu gesetzt werden. D.h. der Zeiger kann z.B. immer wieder auf andere Speicheradressen referenzieren.
Um im Programm mittels Zeiger auf den Inhalt der referenzierten Speicherstelle zuzugreifen (Also nicht den Inhalt des Zeigers selbst) muss der Derefenzierungsoperator $*$ verwendet werden (Achtung: Nicht zu verwechseln mit dem $*$ bei der Deklaration eines Zeigers). Die Anwendung all dieser Operatoren verdeutlicht das folgende Listing.

Listing 2.1: Grundlagen Pointer (pointer.c)

```c
#include <stdio.h>

int main()
{
  int  a = 5;          // Normale Variable vom Typ int
  int  b = 7;
  int* ptr;            // Zeigervariabel auf Typ int

  printf("Inhalt von a: %d\n", a);
  printf("Inhalt von b: %d\n", b);
  printf("Inhalt von ptr: %u\n", ptr);   // ptr noch nicht definiert !


  ptr = &a;            // ptr enthaelt nun die Adresse von a

  // Ausgabe von a mit Dereferenzierung von ptr:
  printf("Inhalt von ptr: %u\n", ptr);
  printf("Inhalt der Speicherstelle auf die ptr zeigt: %d\n", *ptr);

  ptr = &b;     // ptr enthaelt nun die Adresse von b

  // Ausgabe von b mit Dereferenzierung von ptr:
  printf("Inhalt von ptr: %u\n", ptr);
  printf("Inhalt der Speicherstelle auf die ptr zeigt: %d\n", *ptr);

  return 0;
}
```

2.2. Zeiger und Felder

Prinzipiell werden Felder und Zeiger in C intern gleich verwaltet. Eine Feldvariable wird einfach wie ein Zeiger auf die Anfangsadresse der Feldes gehandhabt. Daher ist es auch möglich Feldvariablen und Zeiger mit den gleichen Mechanismen zu benutzen. Der Unterschied eines Feldes zum Zeiger ist jedoch, dass die Feldadresse einmalig festgelegt wird und somit konstant ist. Eine Zeigervariable hingegen kann jederzeit mit einer neuen Adresse beschrieben werden. Die Gleichbehandlung von Feldern und Zeigern veranschaulicht das folgende Listing:

Listing 2.2: Felder und Zeiger (felder_zeiger.c)

```c
#include <stdio.h>

int main()
{
  char feld[] = "Dies ist ein String!";
  char *ptr   = feld;

  // Ausgabe des gesamten Strings:
  printf("%s\n", feld);
  printf("%s\n", ptr);

  // Ausgabe des 3. Elements (also 'e') mit Indexoperator
  printf("3. Element: %c\n", feld[2]);
  printf("3. Element: %c\n", ptr[2]);

  // Ausgabe des 2. Elements (also 'i') mit Zeigerarithmetik
  printf("2. Element: %c\n", *(feld+1));
  printf("2. Element: %c\n", *(ptr+1));
}
```

2.3. Parameterübergabe an Funktionen über Zeiger

Wie jede Art von Datentypen können auch Zeiger beim Aufruf an Funktionen als Parameter übergeben werden. Diese Parameterübergabe wird „Call by Reference“ genannt (Zeiger = Referenz). Call by Reference wird immer dann verwendet, wenn die gerufene Funktion Daten innerhalb der aufrufenden Funktion ändern muss oder mehr als ein Rückgabewert benötigt wird. Auch Felder werden meist mit Call by Reference an Funktionen übergeben (siehe z.B. die String-Funktionen). Durch die Übergabe eines Zeigers an eine Funktion ist diese dann in der Lage die Daten innerhalb der aufrufenden Funktion zu manipulieren. Ein einfaches Beispiel zeigt das folgende Listing:

Listing 2.3: Addieren von 1 zu einer Zahl (addiereEins.c)

```
#include <stdio.h>

void addiereEins(int* x);

int main()
{
  int a = 0;

  printf("a = %d\n", a);
  addiereEins(&a);
  printf("a = %d\n", a);
  addiereEins(&a);
  printf("a = %d\n", a);
  addiereEins(&a);

  return 0;
}

void addiereEins(int* x)
{
  *x = *x + 1;
}
```

Wenn mehrere Rückgabewerte von einer Funktion benötigt werden, so kann dies nur über Zeiger realisiert werden, da eine Funktion an sich nur einen Rückgabewert hat. Alternativ können mehrere Rückgabewerte auch in einem eigenen Strukturdatentyp zusammengefasst werden und dann der normale Funktionsrückgabewert benutzt werden. Die Verwendung von mehreren Rückgabewerten mittels Zeigern zeigt das folgende Listing:

Listing 2.4: 2 Rückgabewerte mit Pointern (complex.c)

```
#include <stdio.h>

void addiere_komplex(int x_re, int x_im, int y_re, int y_im, int* z_re, int* z_im);

int main()
{
  int a_re = 1, a_im = 2;   // Komplexe Zahlen a und b
  int b_re = 3, b_im = 7;

  int c_re, c_im;

  addiere_komplex(a_re, a_im, b_re, b_im, &c_re, &c_im);

  printf("Die komplexe Zahl c ist %d + %d*i\n", c_re, c_im);

  return 0;
}


void addiere_komplex(int x_re, int x_im, int y_re, int y_im, int* z_re, int* z_im)
{
  // addiere komplexe Zahlen x und y  zu z
  *z_re = x_re + y_re;
  *z_im = x_im + y_im;
}
```

Auch Felder können mittels Zeiger an Funktionen übergeben werden. Ein Feld wird C-intern immer als Adresse übergeben (die Anfangsadresse des Feldes im Speicher). Meist muss dann noch ein Parameter übergeben werden, der die Anzahl der Elemente im Feld enthält. Bei Stringfunktionen ist dies nicht der Fall, da das Ende des Strings durch die 0 markiert wird. Ein übergebener Zeiger kann innerhalb der Funktion dann auch wie ein Feldtyp benutzt werden. Das Erzeugen eines zusammengesetzten Strings mittels einer selbst geschriebenen Konkatenierungsfunktion zeigt folgendes Listing:

Listing 2.5: Eigene strcat Funktion (mystrcat.c)

```
#include <stdio.h>

void my_strcat(char* ziel, char* quelle);

int main()
{
  char nachname[100], vorname[100], gesamtname[200] = {'\0'};

  printf("Bitte Vorname eingegen: ");
  scanf("%s", vorname);
  printf("Bitte Nachname eingeben: ");
  scanf("%s", nachname);


  my_strcat(gesamtname, vorname);
  my_strcat(gesamtname, " ");
  my_strcat(gesamtname, nachname);

  printf("Sie heissen %s\n", gesamtname);

  return 0;
}

void my_strcat(char* ziel, char* quelle)
{
  int i = 0, j = 0;

  while(ziel[i] != '\0') i++;    // Ende von ziel finden

  while(quelle[j] != '\0')       // quelle an ziel anhaengen
  {
    ziel[i] = quelle[j];
    i++; j++;
  }

  ziel[i] = '\0';              // Neue Endemarkierung fuer ziel
}
```

Vorsicht! Auch hier gilt: C selbst prüft nicht auf Überschreitung der Feldgrenzen. Eine Funktion die über den allokierten Speicherbereich hinaus auf ein Feld zugreift führt sehr schnell zum Programmabsturz!

2.4. Zeigerarithmetik

Um auf Elemente innerhalb eines Feldes zuzugreifen kann ein Zeigerdatentyp wie ein Feld (mittels [ ]-Operator) benutzt werden. Es gibt jedoch auch die Möglichkeit direkt mittels Zeiger die Element zu adressieren. Dazu muss der Abstand des Elementes zum Anfang des Feldes angegeben werden.

```
char feld[ ] = "Hallo, ich bin ein Feld";
char* ptr   = feld + 4;   // ptr zeigt auf 5. Element, also 'o'
printf("%c", *ptr);

long zahlenfeld[ ] = { 5, 7, 13, 211, 4711, 14, 99999 };
long* zahlenptr = zahlenfeld + 4;   // ptr zeigt auf 5. Element, also 4711
printf("%d", *zahlenptr);
```

Obwohl der Datentyp *char* nur 1 Byte, der Datentyp *long* jedoch 4 Bytes im Speicher verbraucht funktioniert obiges Beispiel. Bei der Berechnung der neuen Adresse eines Zeigers rechnet C automatisch die Elementgröße mit ein. Dies ist möglich da die Zeiger typisiert sind, und C somit weiss, wie gross der Elementtyp ist. Das folgende Listing zeigt die Ausgabe eines Strings. Das Feld wird dabei einmal durchlaufen, indem der Originalzeiger erhöht wird, das andere Mal indem ein Offset erhöht wird, der jeweils zum Feldanfang addiert wird:

Listing 2.6: Stringausgabe mit Pointern (stringout.c)

```
#include <stdio.h>

int main()
{
  int i=0;
  char string[] = "Hallo, ich bin ein String!";

  char* ptr = string;    // Setzt ptr auf Anfangsaddresse von string

  // Ausgabe durch Veraenderung von ptr:
  printf("Ausgabe 1: ");
  while(*ptr != '\0')
  {
    printf("%c", *ptr);
    ptr = ptr + 1;   // ptr zeigt nun auf naechstes Element
  }
  printf("\n");

  // Ausgabe durch Aenderung eines Offsets:
  ptr = string;
  printf("Ausgabe 2: ");
  while(*(ptr+i) != '\0')
  {
    printf("%c", *(ptr + i));
    i++;
  }
  printf("\n");

  return 0;
}
```

Wenn bei der Dereferenzierung eines Pointers ein Offset addiert wird muss immer geklammert werden! Der Dereferenzierungsoperator * hat höhere Priorität als der Additionsoperator +. Die Ausdrücke *(ptr + i) und *ptr + i haben daher unterschiedliche Bedeutung!

2.5. Zeiger auf Strukturen

Zeiger können auch auf Variablen eines Strukturdatentyps zeigen. Damit mittels Pointer auf Strukturkomponenten zugegriffen werden kann muss zunächst der Pointer dereferenziert werden um dann mittels dem "." Operator eine Komponente anzusprechen (z.B. (*ptr).komponente). Da der "." Operator eine höhere Priorität besitzt als der "*" Operator muss geklammert werden! Um dies zu umgehen gibt es in C den "->" Operator, der es erlaubt auf eine Strukturkomponente direkt über Zeiger zuzugreifen (z.B. ptr->komponente). Das folgende Listing zeigt die beiden Zugriffsarten im Vergleich:

Listing 2.7: Zugriff auf Strukturen mit Pointern (zeiger_struct.c)

```
#include <stdio.h>

typedef struct {     // Daten eines Studenten im struct
  char  name[30];
  long  matrikelnr;
} Student;

int main()
{
  Student student;          // Eine Variable vom Typ Student anlegen
  Student *ptr = &student;  // Zeigervariable ptr auf student setzen

  printf("Bitte Namen eingeben: ");
  scanf("%s", (*ptr).name);                  // Zugriff auf Komponente mit
  printf("Bitte Matrikelnr eingeben: ");     // Dereferenzierungsoperator *
  scanf("%d", &(*ptr).matrikelnr);

                // Zugriff auf Komponente mittes -> Operator:

  printf("Eingegebener Name: %s\n", ptr->name);
  printf("Eingegebene Matrikelnr: %d\n", ptr->matrikelnr);

  return 0;
}
```

2.6. Dynamische Speicherreservierung

Eines der häufigsten Einsatzgebiete von Zeigern ist die dynamische Speicherverwaltung. Alle bisher benutzten Felder hatten immer statische Größen, d.h. die Anzahl der Element musste zur Kompilierungszeit bekannt sein. Das führt bei Programmen dazu, dass meist zu viel Speicher verschwendet wird, da ja die maximale Anzahl an Elementen die der Benutzer verarbeiten will vorgehalten werden muss. Will man andererseits Speicher sparen und benutzt nur kleine Felder, so kann es sein, dass der Benutzer nicht so viele Elemente bearbeiten kann wie er eigentlich möchte. Abhilfe schafft hier das dynamische Allokieren von Speicher zur Programmlaufzeit. Speicher vom Betriebssystem kann zur Laufzeit

mittels der Funktion *malloc* aus der Bibliothek *stdlib* angefordert werden. Die Funktion *free* wird benutzt um Speicher, der nicht mehr benötigt wird wieder freizugeben. Ein einfaches Beispiel zeigt folgendes Listing:

Listing 2.8: Dynamische Speicherallokierung mit malloc (feld_malloc.c)

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

int main()
{
  int i, anzahl;
  int* ptr;

  printf("Wieviel Zahlen moechten Sie eingeben: ");
  scanf("%d", &anzahl);

  // Speicher fuer anzahl*int allokieren:
  ptr = (int*) malloc(anzahl * sizeof(int));
  if(ptr == 0) // Pruefe ob Speicher verfuegbar
  {
    printf("Konnte keinen Speicher allokieren\n");
    exit(1);
  }

  for(i = 0; i < anzahl; i++)
  {
    printf("Bitte %d. Zahl eingeben: ", i+1);
    scanf("%d", &ptr[i]);
  }

  printf("Sie haben folgende Zahlen eingegeben:\n");
  for(i = 0; i < anzahl; i++)
  {
    printf("Zahl Nr. %d: %d\n", i+1, *(ptr+i) );
  }

  free(ptr);   // Speicher wieder freigeben
  return 0;
}
```

Die Funktion *malloc* liefert als Rückgabewert einen Pointer vom Typ *void*. Da im Beispiel der allokierte Speicher für *int*-Zahlen benutzt werden soll, muss der gelieferte *void*-Pointer auf den benötigten Typ *int* gecastet werden. Dies geschieht mit dem Ausdruck **(int*)**. Die *malloc*-Funktion reserviert soviel Speicher in Bytes wie angegeben. Wenn also ein Feld mit Elementen die größer als ein Byte sind allokiert werden soll, muss jeweils die Elementgröße noch berücksichtigt werden. Dies kann auf elegante Art und Weise mit dem *sizeof* Operator erledigt werden (siehe auch Listing 2.8). Es ist natürlich möglich, dass kein Hauptspeicher mehr zur Verfügung steht. In diesem Falle liefert die Funktion *malloc* den Wert 0 zurück. Meist kann in so einem Fall nicht mehr sauber weitergearbeitet werden. Es ist jedoch wichtig, das Programm dann sauber zu verlasssen und nicht später einen Programmabsturz durch Zugriff über den 0-Zeiger zu bekommen. Wird allokierter Speicher nicht mehr benötigt sollte er sofort mittels der Funktion *free* freigegeben werden.

3. Dateien

Bei der Verarbeitung von Daten ist es oft sinnvoll, diese nicht nur zur Laufzeit des Programms im Arbeitsspeicher zur Verfügung zu haben, sondern sie darüber hinaus dauerhaft zu speichern. Zu diesem Zweck schreibt ein Programm die Daten in eine *Datei*, die auf einem nicht flüchtigen Speichermedium (z.B. Festplatte, Diskette, Flash-Karte usw) liegt.
Eine Datei (engl. *file*) ist nichts anders als eine Ansammlung von Daten, die auf einem Speicher dauerhaft aufbewahrt werden. Aus Sicht von C kann man sich eine Datei als ein permanentes großes **char**-Feld vorstellen. Eine Datei ist in C daher nicht strukturiert, sondern eine Folge von einzelnen Bytes, weswegen eine Datei in C oft auch als *Datenstrom* (engl. *stream*) bezeichnet wird.

Für die Arbeit mit Dateien definiert der ANSI-C Standard einige Funktionen, die vom Betriebssystem unabhängig und somit überall anwendbar sind. Durch Aufruf von Funktionen aus dem Betriebssystem kann ebenfalls mit Dateien gearbeitet werden. Diese Zugriff sind dann aber abhängig vom verwendeten Betriebssystem und sollen in diesem Skript nicht vorgestellt werden.

3.1. Öffnen und Schließen einer Datei

Bevor man mit einer Datei arbeiten kann, muss diese geöffnet werden. Beim Öffnen einer Datei wird dieser ein Puffer im Arbeitsspeicher zugeordnet. Über diesen Puffer gehen alle Schreib- und Leseoperationen, d.h. die Daten werden zuerst einmal zwischengespeichert. Dies hat den Vorteil, dass nicht für jeden Schreib/Lesezugriff ein langsamer Zugriff auf das Speichermedium notwendig ist.

Doch nun Schritt für Schritt:
Zuerst wollen wir eine Datei öffnen. Dazu stellt ANSI-C die Funktion **fopen()** zur Verfügung:

FILE ***fopen**(const char *pfadname, const char *modus);

- *pfadname*: Name der Datei mit kompletten Pfad als Zeichenkette. Beispiel: ”c:\\ temp \\ text.txt”
- *modus*: Zugriffsart für die Datei als Zeichenkette (siehe Tabelle 3.1)
- *FILE*: Rückgabewert der Funktion ist ein Zeiger auf die Struktur FILE oder NULL wenn die Datei nicht geöffnet werden konnte.

Folgende Modi stehen für das Öffnen der Dateien zur Verfügung:

Tabelle 3.1.: Modi zum Öffnen einer Datei

modus-Argument	Bedeutung
”r” oder ”rb”	(read) zum Lesen öffnen
”w” oder ”wb”	(write) zum Schreiben öffnen (neu anlegen oder Inhalt einer existierenden Datei löschen)
”a” oder ”ab”	(append) zum Schreiben am Dateiende öffnen; nicht existierende Datei wird nicht angelegt
”r+” oder ”r+b” oder ”rb+”	zum Lesen und Schreiben öffnen
”w+” oder ”w+b” oder ”wb+”	zum Lesen und Schreiben öffnen; der Inhalt einer existierenden Datei wird gelöscht
”a+” oder ”a+b” oder ”ab+”	zum Lesen und Schreiben am Dateiende öffnen

Der Buchstabe **b** wird zur Unterscheidung von Text- und Binärdateien benötigt. Unix macht hier keine Unterschiede, wohl aber alle Microsoft-Betriebssysteme (siehe Kapitel 3.1.1).
Zusammengefasst ergeben sich bei den unterschiedlichen Modi folgende Auswirkungen beim Öffnen der Datei:

Einschränkung bzw. Auswirkung	r	w	a	r+	w+	a+
Datei muss zuvor existieren	X			X		
alter Dateiinhalt geht verloren		X			X	
Aus Datei kann gelesen werden	X			X	X	X
In Datei kann geschrieben werden		X	X	X	X	X
Nur am Dateiende kann geschrieben werden			X			X

Das Öffnen einer Datei ist eine weitgehend identische Angelegenheit, weshalb nachfolgendes Beispiel in den meisten Fällen (bis auf Namensänderungen) 1:1 übernommen werden kann. Selbstverständlich muss der Fehlertext und der Öffnungsmodus an die Gegebenheiten angepasst werden.

Listing 3.1: Beispiel Öffnen einer Datei (fopen.c)

```
FILE *fz;        // Zeiger auf unsere Datei

fz = fopen("brief.txt","r"); // Datei brief.txt oeffnen
if (fz == NULL)
{
  printf("Kann die Datei nicht zum Lesen oeffnen");
  ...
}
...
```

Fehler die beim Öffnen einer Datei auftreten können:

- das Öffnen zum Lesen schlägt fehl, wenn die Datei nicht existiert oder nicht gelesen werden kann.
- Wird die Datei zum Schreiben + Lesen geöffnet, so kann nach einer Schreib- oder Leseoperation nicht unmittelbar die andere Operation auf die Datei ausgeführt werden.

Um eine geöffnete Datei wieder zu schließen, bietet ANSI-C die Funktion **fclose()** an.

int **fclose**(FILE *fz);

- *FILE*: Zeiger auf die Struktur FILE mit dem Datei geöffnet wurde.
- Rückgabe ist 0 bei Erfolg oder EOF wenn die Datei nicht geschlossen werden konnte.

Die Funktion schreibt die Daten aus dem Pufferspeicher auf das Speichermedium. Beim normalen Beenden eines Programms geschieht dies automatisch. Man könnte daher eigentlich auf das Schließen der Dateien verzichten, sofern man sicherstellen kann, dass das Programm immer richtig beendet wird. Es gibt aber zwei Gründe, warum eine Datei immer geschlossen werden sollte, wenn sie nicht mehr zum Schreiben oder Lesen benötigt wird:

1. Die Anzahl der geöffneten Dateien für ein Programm ist begrenzt. Wenn daher mit vielen Dateien operiert wird, sollten die nicht mehr benötigten Dateien geschlossen werden.
2. Der Puffer wird erst dann in die Datei geschrieben, wenn er voll ist. Wenn daher nur einige Bytes an die Datei geschrieben wurden und der Puffer nicht voll wird, kann es bei einem unvorhergesehenen vorzeitigen Ende des Programms (Absturz) zum Datenverlust kommen.

3.1.1. Unterschied zwischen Text- und Binärdateien

Unter einer *Textdatei* versteht man eine Folge von Zeilen die durch ein spezielles Zeilenendekennzeichen (line feed oder '\n') getrennt sind. Eine *Binärdatei* dagegen stellt nur eine Folge von Bytes dar, bei der kein spezielles Zeichen eine spezielle Bedeutung

hat. Über das *modus*-Argument bei **fopen()** kann nun festgelegt werden, ob eine Datei im Textmodus (Standard oder t) oder im Binärmodus (b) zu eröffnen ist.

Eine Unterscheidung ist nur bei Betriebssystemen notwendig, die zwischen Text- und Binärdateien unterscheiden (z.B. MS-DOS, Windows...) Bei Unix gibt es nur Binärdateien, ob diese nun Text oder Daten enthalten, spielt keine Rolle.

In beiden Dateiarten können beliebige Zeichen bzw. Bytes gespeichert werden. Der eigentliche Unterschied zwischen Text- und Binärmodus liegt in der Interpretation des Zeilenumbruchs:

- Binärmodus: Jedes '\n' wird als einzelnes LineFeed-Zeichen (ASCII-Wert 10) übertragen.
- Textmodus: Jedes '\n' wird bei der Übertragung aus dem Puffer in die Datei in **zwei** Zeichen umgewandelt: LineFeed und CarriageReturn-Zeichen ('\n' –> '\n'+'\r'). Umgekehrt wird beim Lesen jede Kombination aus \n\r in ein einzelnes LineFeed-Zeichen \n umgewandelt.

Als Folge daraus sind die Textdateien, obwohl sie vom Benutzer aus gesehen dieselben Daten wie eine Binärdatei enthalten, immer etwas größer!
Vorsicht ist geboten, wenn eine Binärdatei im Textmodus geöffnet wird, da dann jedes Byte mit dem Wert 10 in zwei Zeichen (10+13) umgewandelt wird.

3.2. Lesen/Schreiben eines Zeichens aus/in eine Datei

Um ein Zeichen aus einer Datei zu lesen, steht die in <stdio.h> definierte Funktion **fgetc()** und zum Schreiben die Funktion **fputc()** zur Verfügung.

```
int fgetc(FILE *fz);
int fputc(int Zeichen, FILE *fz);
```

Beide Funktionen erwarten in *fz* einen Zeiger auf eine geöffnete Datei.
fgetc() liefert das nächste Zeichen aus der Datei als **unsigned char**, das jedoch im Datentyp **int** abgelegt ist. Der Rückgabewert kann auch die Konstante **EOF** sein. Dann wurde das Dateiende erreicht und es kann kein weiteres Zeichen aus der Datei gelesen werden.
fputc() erwartet in *Zeichen* einen ASCII-Wert, den es in die Datei schreibt. Der Rückgabewert kann entweder der Code des Zeichens oder die Konstante **EOF** sein, wenn beim Schreiben ein Fehler auftrat.

Listing 3.2: Beispiel Ausgabe des Alphabets in eine Datei (fput.c)

```
#include <stdio.h>

int main()
{
  FILE *fz;
  int  count;

  fz = fopen("tenlines.txt", "w");
  if (fz == NULL)
  {
     printf("Konnte Datei nicht oeffnen.\n");
     exit (1);
  }

  for (count = 0 ; count < 26 ; count++)
  {
    if ((count % 4) == 0)
      fputc('\n', fz);
    fputc(count + 'A', fz);
  }
  fclose(fz);

  return 0;
}
```

Listing 3.3: Beispiel Lesen einzelner Zeichen aus einer Datei (fget.c)

```
#include <stdio.h>

int main()
{
  FILE *fz;
  int c;

  fz = fopen("TENLINES.TXT", "r");
  if (fz == NULL)
  {
     printf("Datei existiert nicht.\n");
     exit (1);
  }
  else
  {
     do
     {
        c = fgetc(fz);
        putchar(c);
     } while (c != EOF);
  }
  fclose(fz);

  return 0;
}
```

3.3. Lesen/Schreiben ganzer Zeilen aus/in eine Datei

Zum Lesen einer ganzen Zeile aus einer Datei, steht die in <stdio.h> definierte Funktion **fgets()** und zum Schreiben die Funktion **fputs()** zur Verfügung.

```
char *fgets(char *puffer, int n, FILE *fz);
int fputs(const char *puffer, FILE *fz);
```

3.3.1. fgets()

Über *puffer* wird ein Speicherbereich zur Verfügung gestellt, in dem die Daten abgelegt werden können. Über das Argument *n* wird die maximale Größe des Puffers angegeben und über *fz* eine zum Lesen geöffnete Datei.
fgets() liest dann aus der Datei entweder **n-1** Zeichen oder bis zum nächsten Zeilenumbruch (\n) - je nachdem, was zuerst eintritt - und speichert die gelesenen Zeichen in puffer ab. Hinter dem letzten Zeichen wird immer das Stringendezeichen \0 abgelegt. Als Rückgabewert liefert die Funktion entweder einen Zeiger auf *puffer* oder NULL wenn das Dateiende erreicht wurde oder ein Fehler beim Lesen auftrat.

Listing 3.4: Beispiel Lesen von Zeilen aus einer Datei (fgets.c)

```
#include <stdio.h>

void main()
{
    FILE *fz;
    char s[1000];

    fz=fopen("infile.txt","r");
    while (fgets(s,1000,fz)!=NULL)
        printf("%s",s);
    fclose(fz);
}
```

3.3.2. fputs()

Über *puffer* wird die Zeichenkette übergeben, die in die Datei geschrieben werden soll. Das abschließende \0 der Zeichenkette wird nicht geschrieben. Mit *fz* wird eine zum Schreiben geöffnete Datei übergeben. Es ist zu beachten, dass die Funktion am Ende der ausgegebenen Zeichenkette **kein** Zeilenumbruch \n ausgibt.

3.4. Formatiertes Lesen/Schreiben aus/in eine Datei

Wir kennen bereits die formatierte Eingabe über die Tastatur mit scanf() und die Ausgabe auf dem Bildschirm mit printf(). Für eine formatierte Ein/Ausgabe in eine Datei gibt es nahezu identische Funktionen:

```
int fscanf(FILE *fz, const char *format, ...);
int fprintf(FILE *fz, const char *format, ...);
```

Beide Funktionen verlangen als ersten Parameter einen Zeiger auf die entsprechend geöffnete Datei. Als Rückgabewert liefern die Funktionen die Anzahl der geschriebenen Zeichen oder einen negativen Wert bei Fehler.
Die Funktionen entsprechen bis auf das erste Argument den bekannten Funktionen scanf() und printf().

Listing 3.5: Beispiel Schreiben von Zahlen in eine Datei (fprintf.c)

```
#include <stdio.h>

void main()
{
    FILE *fz;
    int x;

    fz=fopen("out.txt","w");
    for(x=1; x<=10; x++)
        fprintf(fz,"%d\n",x);
    fclose(fz);
}
```

Listing 3.6: Beispiel Lesen einzelner Wörter aus einer Datei (fscanf.c)

```
#include <stdio.h>

int main()
{
   FILE *fz;
   char oneword[100];
   int c;

   fz = fopen("TENLINES.TXT", "r");
   if (fz == NULL)
   {
     printf("Datei exisitiert nicht!\n");
     exit(1);
   }

   do
   {
      c = fscanf(fz, "%s", oneword);
      if (c != EOF)
         printf("%s\n", oneword);
   } while (c != EOF);

   fclose(fz);

   return 0;
}
```

3.5. Lesen/Schreiben ganzer Blöcke aus/in eine Datei

Will man Daten aus Binärdateien lesen oder in diese schreiben, so gibt es die Möglichkeit dies Zeichen für Zeichen zu tun. Das Schreiben oder Lesen ganzer Zeilen scheidet aus, da in Binärdateien die Werte \0 und \n nicht als Zeichenkettenende oder Zeilenumbruch interpretiert werden. Für eine effektive Bearbeitung von Binärdateien sind in <stdio.h> zwei Funktionen definiert, mit denen sich ganze Blöcke von Daten schreiben oder lesen lassen:

```
size_t fread(void *puffer, size_t blockgroesse, size_t blockzahl, FILE *fz);
size_t fwrite(const void *puffer, size_t blockgroesse, size_t blockzahl, FILE *fz);
```

fread() liest bis zu *blockzahl* Objekte, jedes mit *blockgroesse* Bytes, von der Datei, die mit *fz* verbunden ist, in den Speicherbereich, der mit *puffer* reserviert ist.

fwrite() schreibt bis zu *blockzahl* Objekte, jedes mit *blockgroesse* Bytes, aus dem Speicherbereich *puffer* in die Datei, die mit *fz* verbunden ist.
Beiden Funktionen liefern als Rückgabewert die Anzahl von wirklich gelesener oder geschriebener Objekte.
size_t ist als vorzeichenloser, ganzzahliger Datentyp in <stdio.h> definiert (entspricht häufig unsigned int).

3.5.1. Typische Anwendungen

Typische Anwendungen für die Funktionen fread() und fwrite() sind:

- Lesen bzw. Schreiben von Binärdaten (z.B. Inhalt einer int-Variable)
- Einlesen und Schreiben eines ganzen Feldes, wie z.B.

Listing 3.7: Beispiel Lesen eines Arrays (fread_array.c)

```
double werte[100];

...
if (fread(werte,sizeof(double),100,fz) != 100)
{
  ... Fehlerbehandlung ...
}
...
```

- Einlesen oder Schreiben einer ganzen Struktur, wie z.B.

Listing 3.8: Beispiel Schreiben einer Struktur (fwrite_struct.c)

```
struct Person
{
  char vorname[50];
  char nachname[50];
  int alter;
};
...
struct Person schueler;
...
if (fwrite(&schueler,sizeof(Person),1,fz) != 1)
{
  ... Fehlerbehandlung ...
}
...
```

3.6. Positionieren in Dateien

Bisher haben wir nur *sequentiell* aus Dateien gelesen oder in sie geschrieben. Sequentiell bedeutet, dass beim Lesen bzw. Schreiben immer an der Position fortgefahren wird, die sich aus der vorangegangenen Operation ergeben hat.
Beim Öffnen einer Datei befindet sich der **Schreib/Lesezeiger** immer am Anfang einer Datei (außer wenn die Datei im Modus "a" geöffnet wurde). Dies bedeutet, dass die Datei immer von vorne nach hinten bearbeitet wird. Wird nun ein Byte gelesen oder geschrieben wird der *Dateizeiger* um eine Stelle in Richtung Dateiende geschoben und die nächste

Operation betrifft die folgenden Daten.
Nun gibt es in C zwei Funktionen, welche den Zeiger neu setzen oder abfragen können:

```
int fseek(FILE *fz, long offset, int start);
long ftell(FILE *fz);
```

3.6.1. fseek

fseek() ermöglicht das Verschieben des Schreib/Lesezeigers um *offset* Bytes innerhalb der Datei, die durch *fz* angegeben wird. Für das Argument *start* stehen folgende Konstanten zur Verfügung:

- SEEK_SET: Versetzt den Schreib/Lesezeiger **vom Dateianfang an** um *offset* Bytes
- SEEK_CUR: Versetzt den Schreib/Lesezeiger **von der momentanen Position an** um *offset* Bytes
- SEEK_END: Versetzt den Schreib/Lesezeiger **vom Dateiende an** um *offset* Bytes

Die Funktion sollte mit Vorsicht bei Textdateien verwendet werden, da durch die unterschiedliche Behandlung des Zeilenendezeichens im Puffer und in der Datei der Schreib/Lesezeiger nicht genau positioniert werden kann. Ohne Probleme kann der Schreib/Lesezeiger an den Anfang (offset = 0 und start = SEEK_CUR) oder das Ende (offset = 0 und start = SEEK_END) gesetzt werden.

Listing 3.9: Beispiel Ausgabe einer Binärdatei (datbytes.c)

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

int   main(void)
{
   FILE      *dz;
   char      dateiname[100];
   long      von, bis;
   int       zeich;

   printf("Dateiname:␣");
   gets(dateiname);

   dz=fopen(dateiname, "rb");
   if (dz == NULL)
   {
      printf(".....kann␣Datei␣'%s'␣nicht␣oeffnen", dateiname);
      exit(1);
   }
   do
   {
      printf("Hexausgabe␣ab␣Bytenr␣(Ende=-1)␣?␣");
      scanf("%ld", &von);

      if (von >= 0)
      {
         fseek(dz, von, SEEK_SET);
         printf("␣␣␣␣␣␣␣␣␣␣␣bis␣Bytenr␣?␣");
         scanf("%ld", &bis);
```

```
            printf("Hexdump der Datei %s (von Bytenr %ld bis %ld)\n",
                    dateiname, von, bis);
            while (von <= bis)
            {
              zeich=getc(dz);
               if ( zeich != EOF)
                  printf("%02x", zeich);
               else
                  break;
               von++;
            }
            printf("\n\n");
         }
      } while (von >= 0);

      return(0);
}
```

3.6.2. ftell

ftell() ermittelt die aktuelle Position des Schreib/Lesezeigers in der Datei die durch *fz* angegeben wird. Bei Binärdateien entspricht die ermittelte Position der Bytezahl ab Dateianfang. Bei Textdateien ist diese Aussage, bei Systemen die zwischen Text- und Binärdateien unterscheiden, nicht gegeben.

3.7. Löschen und Umbenennen

Zum Löschen und Umbenennen von Dateien sind in <stdio.h> zwei Funktionen definiert:

```
int remove(const char *pfad);
int rename(const char *altname, const char *neuname);
```

Bei beiden Funktionen sollte die angegebene Datei nicht geöffnet sein. Der Rückgabewert ist 0 wenn kein Fehler auftrat oder -1 bei Fehler.
remove() löscht die mit *pfad* angegebene Datei oder Verzeichnis.
rename() benennt die durch *altname* angegebene Datei in die Datei mit Namen *neuname* um.

3.8. Standardkanäle

Für jedes gestartete Programm werden automatisch die folgenden FILE-Zeigerkonstanten, die in <stdio.h> definiert sind, bereitgestellt:

- **stdin**: Standardeingabe. Entspricht normalerweise der Tastatur.
- **stdout**: Standardausgabe. Entspricht normalerweise dem Bildschirm.
- **stderr**: Standardfehlerausgabe. Entspricht normalerweise dem Bildschirm.

Diese FILE-Zeiger können analog zu „normalen“ Dateien verwendet werden, nur sollten diese nicht geschlossen werden!!
Damit ist die Funktion **scanf(...)** identisch mit **fscanf(stdin,...)** und **printf(...)** ist identisch mit **fprintf(stdout,...)**. Gleiches gilt für die Funktionen getchar() und fgetc(stdin) sowie putchar(..) und fputc(stdout,...).

4. Rekursion

Unter dem Begriff Rekursion versteht man eine Programmiertechnik, bei der sich eine Funktion selbst wieder aufruft. Ein solcher rekursiver Aufruf kann entweder direkt oder auf Umwegen über eine andere Funktion erfolgen. Entscheidend ist, dass derselbe Funktionscode mit veränderten Variablenwerten erneut durchlaufen wird, bevor ein voriger Aufruf beendet ist.

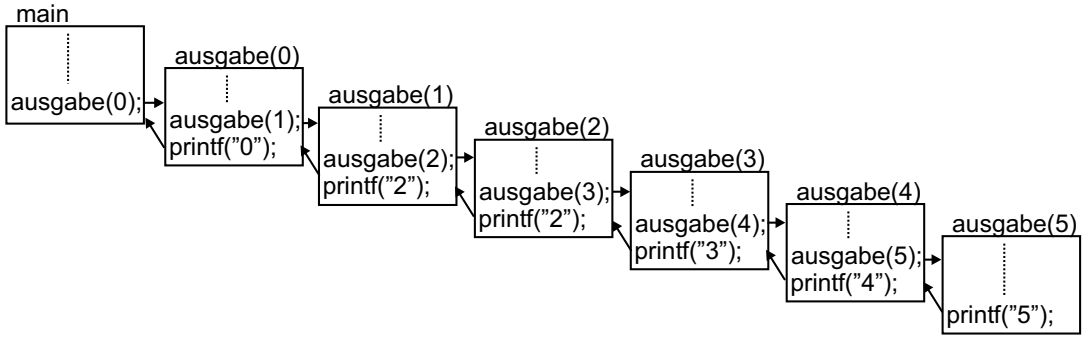
4.1. Allgemeines

Im folgenden Beispiel sind bereits die wichtigsten Elemente eines rekursiven Funktionsaufrufs enthalten:

Listing 4.1: Beispiel einfache Rekursion (rekursion1.cpp)

```
#include <stdio.h>

void ausgabe(int zahl)
{
  if (zahl < 5)
    ausgabe(zahl+1);
  printf("%d ",zahl);
}

void main()
{
  ausgabe(0);
}
```

Die Funktion *ausgabe()* wird hier rekursiv aufgerufen. Der Start der Rekursion ist in der Funktion *main()*. Hier erfolgt der erste Aufruf der Funktion mit dem Parameter 0.



Nun wird die Funktion durch sich selbst mit verändertem Parameter immer wieder aufgerufen und zwar bevor die Ausgabe der Zahl erfolgt. Da dies endlos geschehen würde,

braucht die Rekursion eine **Abbruchbedingung**, die hier in Form des Maximalwertes der Zahl realisiert ist. Erst dann wird ein weiterer Aufruf der Funktion unterbunden und die Ausgabe der Zahl erfolgt. Nun werden nacheinander alle zuvor „begonnenen“ Funktionen in rückwärtiger Reihenfolge wieder beendet und die jeweilig gültige Zahl wird ausgegeben. Somit gibt das Programm folgende Zeile auf dem Bildschirm aus:

```
5 4 3 2 1
```

Folgende **Regeln** gelten für rekursive Funktionen:

- Ein rekursiver Funktionsaufruf erfordert Zeit und Speicherplatz, da für jeden neuen Aufruf neue lokale Variablen, d.h. neue Speicherplätze anzulegen sind. Das gilt nicht für *static*-Variablen! Mit Rekursion lässt sich daher im allgemeinen kein Speicherplatz sparen und der Programmablauf ist etwas langsamer.
- Der Programmierer muss dafür sorgen, dass eine rekursive Schachtelung gegen ein definiertes Ende läuft (*Abbruchbedingung!)*
- Bei rekursiven Funktionsaufrufen können gleichzeitig mehrere Versionen einer Variablen auf dem Stack liegen, die den einzelnen Verschachtelungsebenen entsprechen.
- Jede rekursive Funktion kann mittels Wiederholungsanweisungen realisiert werden; das Umgekehrte gilt ebenfalls.
- Durch Rekursion können Programm kompakter und leichter verständlich werden, wenn die sich wiederholenden Aufgaben in eine Funktion gepackt werden.

4.2. Beispiel: Fakultät

Die Fakultät einer ganzen Zahl n ist für $n > 0$ definiert als:

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot ... \cdot n$$

oder, in der rekursiven Schreibweise

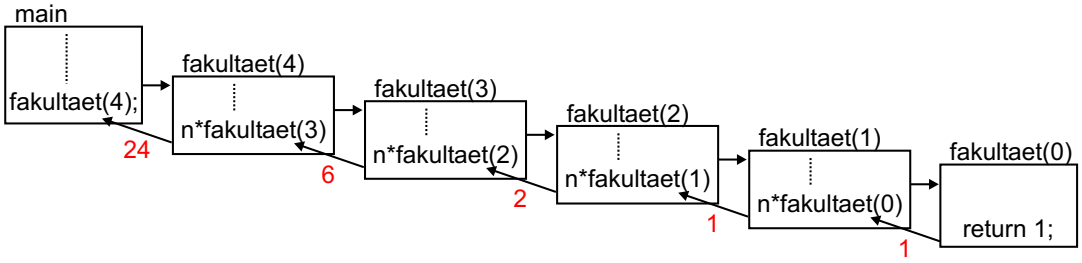
$$n! = n \cdot (n-1)!$$

Als Besonderheit gilt: $0! = 1$
Mit dieser rekursiven Definition lässt sich die Fakultät recht einfach mit einer rekursiven Funktion *fakultaet(n)* berechnen:

Listing 4.2: Berechnung der Fakultät (fakultaet.cpp)

```
#include <stdio.h>

long fakultaet(long n)
{
  if (n == 0)
    return 1;
  else
    return n*fakultaet(n-1);
}

void main()
{
  long zahl;

  printf("Geben sie eine Zahl ein: ");
  scanf("%ld",&zahl);
  printf("Die Fakultaet dieser Zahl ist: %ld\n",fakultaet(zahl));
}
```

Die Abbruchbedinung besteht hier in der Sonderbedingung für $0! = 1$. Damit ist die Berechnung definiert und das Ergebnis kann an die vorige Ebene zurückgeliefert werden. Damit ist pro Ebene immer nur die Berechnung $n * x$ notwendig, wobei x das Ergebnis der tieferen Ebene ist.



4.3. Backtracking-Algorithmen

Backtracking ist ein Lösungsverfahren, das nach dem *Trial-and-Error-Verfahren* arbeitet. Beim Backtracking versucht man, eine Teillösung eines Problems systematisch zu einer Gesamtlösung auszubauen. Falls in einem gewissen Punkt ein weiterer Ausbau einer Teillösung nicht mehr möglich ist (Sackgasse), werden eine oder mehrere der letzten Teilschritte rückgängig gemachte. Die dann reduzierte Teillösung versucht man auf einem anderen Weg wieder auszubauen. Das Zurücknehmen von Schritten und erneute Vorangehen wird solange wiederholt, bis eine Lösung des vorliegende Problems gefunden ist oder bis man erkennt, dass das Problem keine Lösung bietet.
Diese etwas theoretische Definition ist am einfachsten an einem Beispiel aus dem alltäglichen Leben erklärt:
Bei einer Fahrt durch eine Stadt von einem Punkt A zu einem Punkt B kann man an einer Kreuzung rechts oder links abbiegen oder geradeaus fahren. An der nächsten Kreuzung stellt sich dasselbe Problem wieder. Gelangt man bei der Suche in eine Sackgasse, fährt man zur letzten Kreuzung zurück und versucht es auf einem anderen Weg. Dies bedeutet, dass an jeder Kreuzung dieselben Entscheidungen anfallen und die Reaktion auf die Entscheidung ist ausschlaggebend, wie der weitere Weg verläuft.

4.3.1. Springer-Problem

Auf einem Schachbrett soll ein Springer in einem Feld starten und jedes Feld genau einmal betreten. Es sollen alle möglichen Lösungen auf dem Bildschirm ausgegeben werden.

Lösungsansatz

Der Springer bewegt sich immer zwei Felder waagrecht oder senkrecht und dann ein Feld zur Seite. Dadurch ergeben sich insgesamt 8 mögliche Bewegungen:

1. 2 Felder nach oben, 1 Feld nach links
2. 2 Feld nach oben, 1 Feld nach rechts
3. 2 Felder nach rechts, 1 Feld nach oben
4. 2 Felder nach rechts, 1 Feld nach unten
5. 2 Felder nach unten, 1 Feld nach links
6. 2 Feld nach unten, 1 Feld nach rechts
7. 2 Felder nach links, 1 Feld nach oben
8. 2 Felder nach links, 1 Feld nach unten

Es gilt also eine Funktion für die Bewegung des Springers zu definieren. Diese Funktion bewegt den Springer von einer gegebenen Position auf eine neue Position. Damit alle Möglichkeiten gefunden werden können, müssen natürlich alle Bewegung untersucht werden. Jede Bewegung hat aber andere Wege zur Folge! Als weitere Regel gilt, dass jedes Feld nur einmal betreten werden darf. Es muss daher irgendwo gespeichert werden, auf welchen Feldern der Springer bereits war. Als Abbruchbedingung verwenden wir die Anzahl der Springerzüge. Jedes Feld wurde betreten, wenn der Springer $n \bullet n$ Züge ausgeführt hat.
Die einzelnen Züge speichern wir uns in einem globalen Feld der Größe $n \bullet n$. Eine 0 im Feld bedeutet, dass der Springer das Feld noch nicht betreten hat. Für das Beispiel wurde für $n = 5$ gewählt, um die Rechenzeit etwas zu reduzieren. Für die Ausgabe des Feldes verwenden wird eine weitere Funktion.

Lösung

Die Funktion für die Bewegung des Springers benötigt eine Ausgangsposition und die Zuganzahl (wegen Abbruchbedingung). Durch rekursive Aufrufe derselben Funktion wird das Backtracking verfahren realisiert.

Listing 4.3: Springerproblem (springer.cpp)

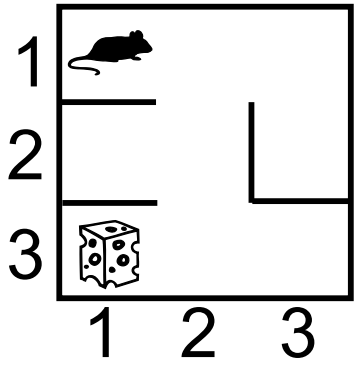
```
#include <stdio.h>
#include <conio.h>

#define N 5
// Feld zum speichern der Zugfolge
int brett[N][N] = {0};

// Ausgabe des Brettes bei einer Loesung
void ausgabe()
{
  int x,y;
  static int loesungen = 0;

  loesungen++;
  printf("\n\nLoesung %d:\n",loesungen);
  for (y=0; y<N; y++)
  {
    printf("\n");
    for (x=0; x<N; x++)
      printf("%3d",brett[x][y]);
  }
  printf("\nBitte Taste druecken!");
  getch();
}

// Bewegung des Springers
void springer(int x, int y, int zugnr)
{  // kontrollieren, ob neue Position erlaubt ist
  if ( (x >= 0) && (x < N) && (y >= 0) && (y < N) && (brett[x][y] == 0) )
  {
    brett[x][y] = zugnr;// Feld belegen mit Zugnr
    // bei Abbruchbedingung Feld ausgeben
    if (zugnr == (N*N))
      ausgabe();
    else
    {  // 1. Moeglichkeit: 2 nach oben, links
      springer(x-1,y-2,zugnr+1);
      // 2. Moeglichkeit: 2 nach oben, rechts
      springer(x+1,y-2,zugnr+1);
      // 3. Moeglichkeit: 2 nach rechts, oben
      springer(x+2,y-1,zugnr+1);
      // 4. Moeglichkeit: 2 nach rechts, unten
      springer(x+2,y+1,zugnr+1);
      // 5. Moeglichkeit: 2 nach unten, links
      springer(x-1,y+2,zugnr+1);
      // 6. Moeglichkeit: 2 nach unten, rechts
      springer(x+1,y+2,zugnr+1);
      // 7. Moeglichkeit: 2 nach links, oben
      springer(x-2,y-1,zugnr+1);
      // 8. Moeglichkeit: 2 nach links, unten
      springer(x-2,y+1,zugnr+1);
    }
    brett[x][y] = 0;// Feld verlassen und freigeben
  }
}

void main()
{  // Starten links oben
  springer(0,0,1);
}
```

Bei jedem neuen Aufruf der Funktion wird die Zugnummer erhöht. Ist die Zielposition gültig (innerhalb des Feldes und Ziel noch frei) dann wird das Feld belegt und von diesem aus werden wiederum alle 8 Möglichkeiten untersucht. Erst wenn diese 8 Möglich-

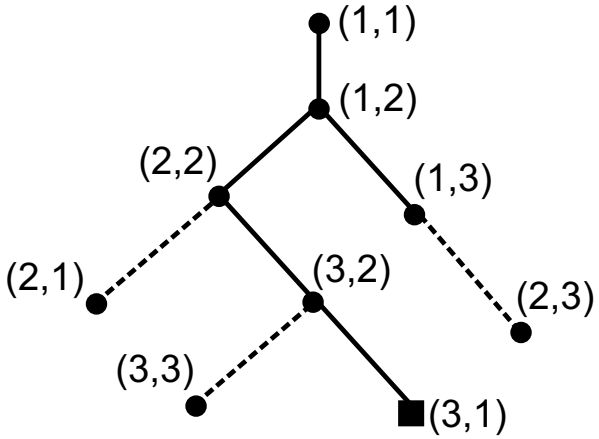
keiten untersucht wurden, wird das Feld wieder freigegeben und die „Rekursionsebene“ verlassen. Damit wird ein Feld „zurückgesprungen“ und von dort können die restlichen Möglichkeiten untersucht werden.

4.3.2. Ausweg aus dem Labyrinth

Ein klassisches Beispiel, das sich mit Backtracking lösen lässt, ist eine Maus im Labyrinth, die einen Käse bzw. den Ausweg finden muss.



Die möglichen Wege der Maus lassen sich dabei als Baum darstellen. Die Knoten des Baumes sind dabei mit der Position der Maus im Labyrinth identisch.



Im Beispielprogramm ist die Suche in einem etwas größeren Labyrinth implementiert. Das Labyrinth wird als 2-dimensionales Feld dargestellt indem der Wert 'X' als Mauer, der Wert 'K' als Käse und der Wert 'o' als zurückgelegter Weg markiert werden. Nach jeder Bewegung der Maus wird das komplette Labyrinth ausgegeben, so dass das Wegsuchverfahren verfolgt werden kann.

Listing 4.4: Käse im Labyrinth (labyrinth.cpp)

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

char feld[5][5] = { {'␣','␣','X','X','␣'},
                    {'X','␣','␣','␣','␣'},
                    {'␣','␣','X','␣','X'},
                    {'␣','X','␣','␣','␣'},
                    {'␣','␣','␣','X','K'} };

void ausgabe()
{
  int x,y;

  // Feld mit Rahmen ausgeben
  printf("\n+");
  for (x=0; x<5; x++)
    printf("-");
  printf("+\n");

  for (y=0; y<5; y++)
  {
    printf("|");
    for (x=0; x<5; x++)
      printf("%c", feld[x][y]);
    printf("|\n");
  }

  printf("+");
  for (x=0; x<5; x++)
    printf("-");
  printf("+\n");
  getch();// warten auf Tastendruck
}

void suche(int x, int y)
{  // kontrollieren ob Fled gueltig
  if (x>=0 && y>=0 && x<5 && y<5
    && feld[x][y]!='X' && feld[x][y]!='o')
  {  // Ausgabe wenn Kaese gefunden
    if (feld[x][y]=='K')
    {
      printf("Kaese␣gefunden");
      ausgabe();
    } else
    {  // Feld belegen und alle Moeglichkeiten untersuchen
      feld[x][y] = 'o';
      ausgabe();
      suche(x+1,y);
      suche(x,y+1);
      suche(x-1,y);
      suche(x,y-1);
      feld[x][y] = '␣';// Feld wieder freigeben
      ausgabe();
    }
  }
}

void main()
{
  ausgabe();
  suche(0,0);
}
```

Die Wegsuche wird wiederum durch rekursive Funktionsaufrufe der Funktion *suche()* durchgeführt. Diese erhält bei jedem Aufruf das Zielkoordinatenpaar und belegt das

Feld, sofern es gültig ist. Ausgehend von diesem Feld werden alle vier Möglichkeiten durchgespielt. Erst wenn alle vier Möglichkeiten untersucht wurden, wird das Feld wieder freigegeben und eine Ebene zurückgesprungen.

5. Datenstrukturen

Benötigt man eine größere Menge an Daten, benutzt man oft statische Felder. Diese statischen Felder sind recht einfach in der Handhabung, haben in der Praxis jedoch einige Nachteile:

- Die Größe eines Feldes muss schon bei der Compilierung festgelegt werden und kann später nicht mehr an die Bedürfnisse angepasst werden.
- Zu groß angelegte Felder verbrauchen viel Arbeitsspeicher
- Spezielle Konstrukte, wie z.B. sortierte Listen, lassen sich nur mit viel Aufwand verwalten. So ist das Einfügen bzw. Entfernen eines Elements aus einer Liste eine komplexe Aufgabe.

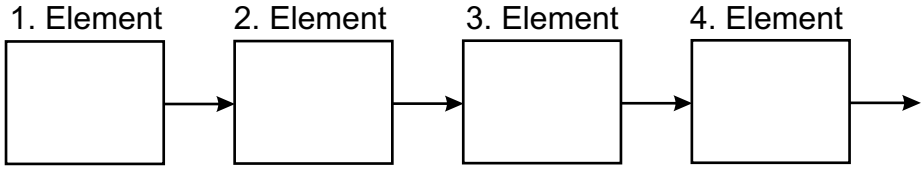
Um diese Einschränkungen und Schwierigkeiten zu beheben werden wir die **dynamischen Datenstrukturen** verwenden. Diese kommen in verschiedenen Konstrukten wie sortierten Listen, Schlangen, Stacks oder Bäumen verwendet werden. Das Wort *dynamisch* sagt schon, dass die gesamte Speicherplatzreservierung während der Laufzeit bei Bedarf geschieht. Für die Speicherung der Daten werden Strukturen verwendet, die immer noch mindestens einen Zeiger auf sich selbst haben (**rekursive Strukturen**).
Der große Vorteil gegenüber dynamischen Feldern ist die einfache Verwaltung beim Einfügen oder Löschen. Ein Nachteil ist der etwas aufwändigere Zugriff auf ein bestimmtes Element, da man keinen Direktzugriff auf die einzelnen Elemente hat.

5.1. Verkettete Listen

Verkettete Listen werden dynamisch während des Programmablaufs verlängert. Anders als bei Feldern ist es bei verketteten Listen nicht garantiert, dass die einzelnen Elemente hintereinander im Speicher liegen. Um eine Verbindung zwischen den einzelnen, eventuell im Speicher verstreuten Elementen einer Liste zu haben, muss man sich die Adresse des nächsten oder auch des vorherigen Elements im jeweiligen Element halten. Dazu benötigt man rekursive Strukturen.

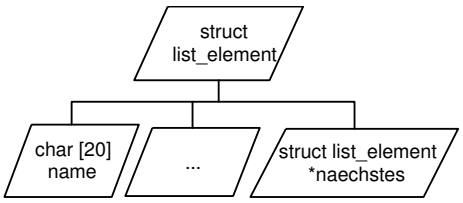
5.1.1. einfach verkettete Listen

Eine einfach verkettete Liste kann nur von einem Ende (z.B. vom Anfang) schrittweise von Element zu Element durchlaufen werden:



Für die Verwaltung der Daten benötigt man eine rekursive Struktur, die neben den Daten auch noch einen Zeiger auf das jeweils nächste Element enthält:

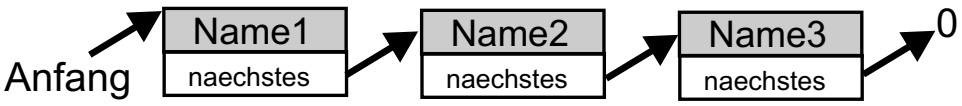
```
struct list_element
{
  char name[20];    // Beispiel für die Daten
  ...
  struct list_element *naechstes; // Zeiger auf nächstes Element
};
```



Jedes Element einer Liste ist also ein Objekt dieser Struktur. Alle Elemente sind gleich aufgebaut!
Für jede verkettete Liste benötigt man einen Zeiger auf den Anfang (Ende) der Liste. Dieser Zeiger beinhaltet die Adresse des ersten bzw. letzten Elements der Liste und ist somit der Einsprungpunkt beim Bearbeiten der Liste. Dieser Zeiger darf niemals verloren gehen, da sonst der Zugriff auf die gesamte Liste verloren ist! Die Folge wäre ein reservierter aber nicht mehr zugänglicher Speicherbereich!
Auf die Liste können nun verschiedene Operationen angewandt werden. Wir wollen hier die wichtigsten Operationen kennen lernen:

- Einfügen eine neuen Elements ans Ende der Liste
- Ausgeben der gesamten Liste
- Löschen eines Elements
- Einfügen eines neuen Elements innerhalb der Liste
- Löschen der gesamten Liste

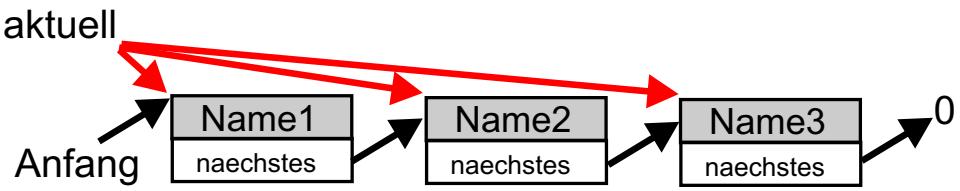
Als Beispiel soll obige Struktur verwendet werden.
Im Hauptprogramm kann der Benutzer die gewünschte Aktion auswählen. Am Programmende wird die Liste komplett aus dem Speicher entfernt. Die Struktur ist global deklariert, damit sie in allen Funktionen bekannt ist. Ebenfalls global definiert ist der Zeiger auf den Anfang der Liste damit die Handhabung in den einzelnen Funktionen vereinfacht wird.



Listing 5.1: Hauptprogramm für einfach verkettete Liste (einfacheListe1.cpp)

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

// Strukturdeklaration
struct list_element
{
  char name[50];
  struct list_element *naechstes;
};
// Zeiger auf den Anfang der Struktur
struct list_element *anfang = NULL;

void element_anhaengen();
void liste_loeschen();
void liste_ausgeben();
void element_loeschen();
void element_einfuegen();

void main()
{
  char wahl;

  do
  {
    printf("\nNamensliste\n\n");
    printf("1 - neues Element ans Ende\n");
    printf("2 - Liste ausgeben\n");
    printf("3 - Element loeschen\n");
    printf("4 - neues Element einfuegen\n");
    printf("e - Ende\n\n");
    printf("Ihre Wahl: ");

    wahl = getchar();
    fflush(stdin);
    switch(wahl)
    {
    case '1': element_anhaengen();
          break;
    case '2': liste_ausgeben();
          break;
    case '3': element_loeschen();
            break;
    case '4': element_einfuegen();
            break;
    case 'e': break;
    default: printf("Ungueltige Eingabe!\n");
    }
  } while (wahl != 'e');

  liste_loeschen();
}
```

Die Liste wird gelöscht, indem Element für Element aus dem Speicher entfernt wird. Dabei ist darauf zu achten, die Adresse des nächsten Element nicht durch falsche Reihenfolge zu verlieren!



Listing 5.2: einfach verkettete Liste löschen (listeloeschen.cpp)

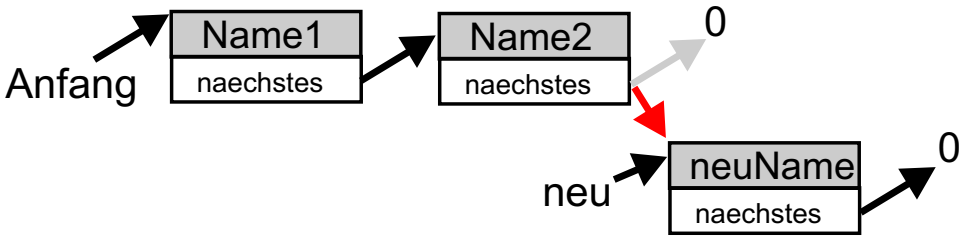
```
void liste_loeschen()
{
  struct list_element *aktuell;  // Zeiger auf aktuelle Struktur

  while (anfang != NULL)
  {
    aktuell = anfang;// Adresse des ersten Elements sichern
    anfang = aktuell->naechstes;// Anfang verschieben
    free(aktuell);// Speicher freigeben
  }
}
```

Die Ausgabe erfolgt nahezu identisch zum Löschen einer gesamten Liste. Hier darf natürlich der Speicher nicht freigeben und der Anfangszeiger nicht verschoben werden. Die Liste kann solange durchlaufen werden, bis der Zeiger auf das nächste Element ein NULL-Zeiger ist. Dann wurde das Ende der Liste erreicht.

Listing 5.3: einfach verkettete Liste ausgeben (listeausgeben.cpp)

```
void liste_ausgeben()
{
  struct list_element *aktuell;  // Zeiger auf aktuelle Struktur
  int nummer = 1;  // Zaehler fuer Nummerierung

  aktuell = anfang;// von Anfang an beginnen
  printf("\n\n");
  while (aktuell != NULL)
  {  // Name und Nummer ausgeben
    printf("%d. Name: %s\n",nummer,aktuell->name);
    aktuell = aktuell->naechstes;// naechstes Element
    nummer++;
  }
}
```

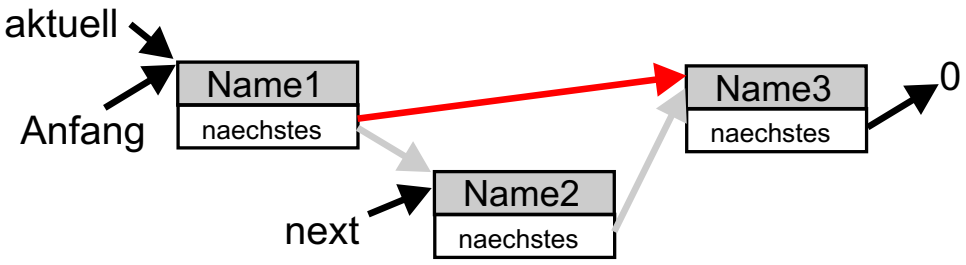
Um ein Element ans Ende der Liste anzuhängen, muss ebenfalls die gesamte Liste durchlaufen werden, um den *naechstes*-Zeiger des letzten Elements auf die Adresse des neuen Elements zu setzen. Somit ist die Liste um ein Element erweitert worden. Zuvor wird der Speicher für das neue Element dynamisch reserviert. Beachtet werden muss, wenn eine neue Liste angelegt wird (*anfang==NULL*). Dann wird nur der Anfangszeiger auf die Adresse des neuen Elements gesetzt.



Listing 5.4: Element an einfach verkettete Liste anhängen (listeanhaengen.cpp)

```
void element_anhaengen()
{
  // Zeiger auf aktuelles und neues Element
  struct list_element *neu,*aktuell;

  // Speicher fuer ein neues Element reservieren
  neu = (struct list_element*) malloc(sizeof(struct list_element));
  if (neu == NULL)
  {
    printf("\n\n*** nicht genuegend Speicherplatz! ***\n\n");
    return;
  }

  printf("\nBitte neuen Namen eingeben: ");
  gets(neu->name);
  neu->naechstes = NULL;// da letztes Element = NULL

  if (anfang == NULL)
  {
    anfang = neu;// neuer Anfang
  } else
  {  // leztes Element suchen
    aktuell = anfang;
    while (aktuell->naechstes != NULL)
      aktuell = aktuell->naechstes;
    // neues Element als letztes anhaengen
    aktuell->naechstes = neu;
  }
}
```

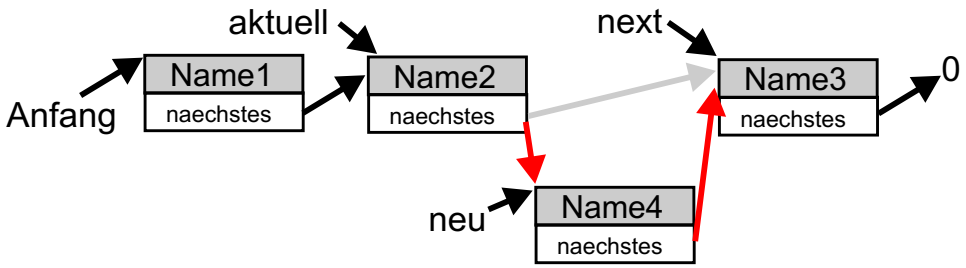
Das Löschen eines Elements aus der Liste ist etwas komplizierter. Zuerst muss der Zeiger auf das gewünschte Element gefunden werden (*next-Zeiger*). Dann kann der *naechstes*-Zeiger des vorigen Elements (*aktuell*) auf das übernächste Element „verbogen“ werden. Erst dann darf das gesuchte Element gelöscht werden.



Listing 5.5: Element in einfach verketteter Liste löschen (elementloeschen.cpp)

```
void element_loeschen()
{  // Zeiger auf aktuelles und naechstes Element
  struct list_element *aktuell,*next;
  char auswahl[50];

  if (anfang != NULL)
  {
    printf("\nzu loeschender Name eingeben: ");
    gets(auswahl);

    aktuell = anfang;
    // kontrollieren ob erstes Element geloescht werden soll
    if (!strcmp(aktuell->name,auswahl))
    {
      anfang = aktuell->naechstes;
      free(aktuell);
      printf("Element gefunden und geloescht.\n");
    } else
    {
      while (aktuell != NULL)
      {  // Name des naechsten Elements vergleichen
        next = aktuell->naechstes;
        if (next != NULL)
        {  // Element loeschen wenn Name identisch
          if (!strcmp(next->name,auswahl))
          {  // Zeiger veraendern
            aktuell->naechstes = next->naechstes;
            free(next);
            printf("Element gefunden und geloescht.\n");
          }
        }
        aktuell = aktuell->naechstes;
      }
    }
  }
}
```

Das Einfügen eines Element in eine bestehende Liste ist ähnlich zum Löschen eines Elements. Auch hier muss zuerst das Element an der gewünschten Stelle (*next*) gefunden werden. Anschließend kann der *naechstes*-Zeiger des vorigen Elements (*aktuell*) auf das neue Element zeigen und der *naechstes*-Zeiger des neuen Elements auf die Adresse des folgenden Elements. Somit ist ohne große Schiebereien ein neues Element innerhalb der Liste eingefügt worden.

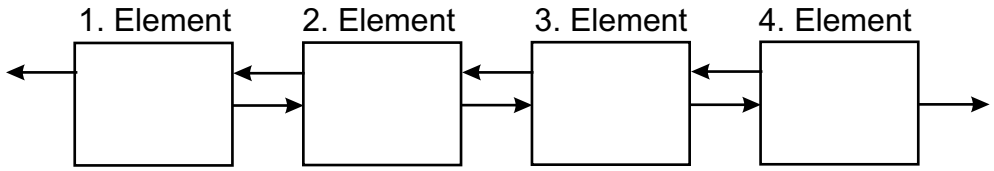


Listing 5.6: Element in einfach verkettete Liste einfügen (listeeinfuegen.cpp)

```
void element_einfuegen()
{  // Zeiger auf aktuelles, neues und naechstes Element
  struct list_element *neu,*aktuell,*next;
  char suche[50];

  // Speicher fuer neues Element reservieren
  neu = (struct list_element*) malloc(sizeof(struct list_element));
  if (neu == NULL)
  {
    printf("\n\n*** nicht genuegend Speicherplatz! ***\n\n");
    return;
  }

  printf("\nBitte neuen Namen eingeben: ");
  gets(neu->name);
  printf("\nEinfuegen vor Name: ");
  gets(suche);

  // neues Element = neuer Anfang?
  if (!strcmp(anfang->name,suche))
  {
    neu->naechstes = anfang;
    anfang = neu;
  } else
  {
    aktuell = anfang;
    while (aktuell != NULL)
    {  // Name des naechsten Elements vergleichen
      next = aktuell->naechstes;
      if (next != NULL)
      {  // wenn Name identisch, dann Element einhaengen
        if (!strcmp(next->name,suche))
        {
          neu->naechstes = next;
          aktuell->naechstes = neu;
          break;
        }
      }
      aktuell = aktuell->naechstes;
    }
  }
}
```

5.1.2. doppelt verkettete Listen

Bestimmte Operationen mit verketteten Listen, wie das Durchsuchen der Liste oder das Einfügen eines Listenelements, lassen sich vereinfachen, wenn man die Listenelemente nicht nur einfach, sondern **doppelt** miteinander verkettet. In diesem Fall enthalten die Listenelemente nicht nur einen Zeiger auf das folgende Element, sondern auch einen Zeiger auf das vorhergehende Element. Man kann sich daher nicht nur vorwärts, sonder auch rückwärts durch die Liste bewegen. Dadurch lassen sich einige Operationen beschleunigen, da man die Liste nicht immer vom Anfang durchlaufen muss.



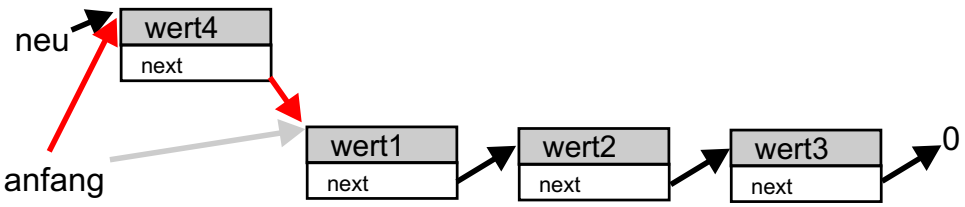
Beispiel einer Struktur für einen doppelt verkettete Liste:

```
struct list_element
{
  char name[20];    // Beispiel für die Daten
  ...
  struct list_element *naechstes; // Zeiger auf nächstes Element
  struct list_element *voriges; // Zeiger auf vorhergehendes Element
};
```

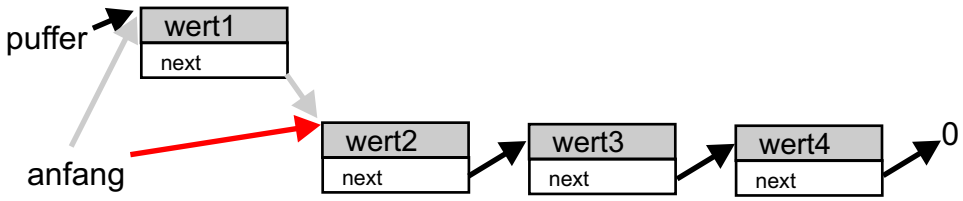
Die entsprechenden Operationen sind vergleichbar mit einer einfach verketteten Liste. Nur müssen hier zwei Zeiger auf die richtige Adresse gesetzt werden damit die Liste in beiden Richtungen zusammenhängend ist. Oft wird hier auch mit einem zusätzlichen Zeiger auf das Ende der Liste gearbeitet, um den schnellen Zugriff auf das Ende der Liste zu ermöglichen. Somit ist es dann einfacher, die Liste von hinten nach vorne zu durchlaufen.

5.2. Stack

Der Stack oder Kellerspeicher ist eine einfach verkettete Liste, bei der immer die zuletzt gespeicherten Daten zuerst gelesen werden. Dieses *Last-In-First-Out*-Prinzip (LIFO) wird z.B. auch bei einem Funktionsaufruf im Rechner zur Speicherung der lokalen Variablen und Adressen verwendet. Bei dieser Liste wird ein neues Element immer am Anfang der Liste angefügt. Das Lesen eines Elements erfolgt ebenfalls am Beginn einer Liste und das Element wird daraufhin aus dem Speicher entfernt.
Im folgenden Beispiel dienen die zwei Funktionen *push()* und *pop()* zur Demonstration der Funktion. Wieder wird ein globaler Zeiger auf den Anfang der Liste verwendet. Das Einfügen eines neuen Elements geschieht hier immer am Anfang, d.h. der Anfangszeiger zeigt nach der Operation immer auf das neue Element und dessen *next*-Zeiger auf den bisherigen Anfang.



Das Löschen erfolgt ebenfalls am Anfang der Liste. Hier ist zu beachten, dass der Zeiger auf das zu löschende Objekt (der bisherige Anfang) zwischengespeichert werden muss. Anschließend wird der Anfangszeiger auf das nächste Element verschoben. Erst dann darf der Speicher des bisherigen Anfangs freigegeben werden! Deshalb die Zwischenspeicherung in der Variablen *puffer*. Würde man die Adresse des Anfangs nicht zwischenspeichern hätte man nach dem Verschieben keinen Zugriff mehr darauf. Wollte man den Speicher zuerst freigeben und dann noch auf das *next*-Element der Struktur zugreifen, wäre ein unerlaubter Speicherzugriff die Folge.



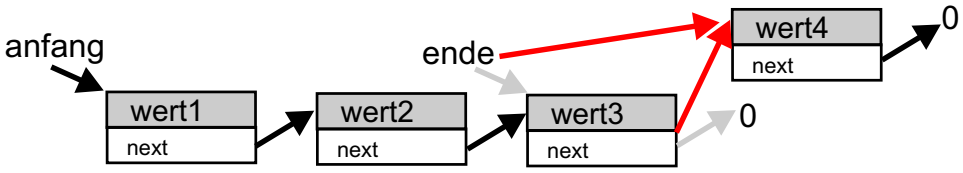
Listing 5.7: Beispiel für Stackverwaltung (stack.cpp)

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

// Struktur
struct element
{
  float wert;
  struct element *next;
};
// Zeiger auf den Anfang der Liste
struct element *anfang = NULL;

int push(float zahl)
{
  struct element *neu;// Zeiger auf neues Element

  // Speicher fuer neues Element reservieren
  neu = (struct element*)malloc(sizeof(struct element));
  if (neu == NULL)
    return 1;
  // Werte zuweisen, an den Anfang haengen
  neu->wert = zahl;
  neu->next = anfang;
  anfang = neu;
  return 0;
}

int pop(float *zahl)
{
  struct element *puffer;// Hilfsvariable

  if (anfang == NULL)
    return 1;
  // Zahl des ersten Elements holen
  *zahl = anfang->wert;
  // Anfang sichern, um 1 weiter verschieben
  puffer = anfang;
  anfang = anfang->next;
  // Speicher des bisherigen Anfangs freigeben
  free(puffer);
  return 0;
}

void main()
{
  float wert;
  char wahl;

  do
  {
    printf("\n");
    printf("1: auf Stack speichern (push)\n");
    printf("2: von Stack lesen (pop)\n");
    printf("e: Ende\n");
    printf("Ihre Wahl: \n");

    wahl = getchar();
```

```
    fflush(stdin);

    if (wahl == '1')
    {
      printf("\nZahl eingeben: ");
      scanf("%f",&wert);
      if (push(wert))
        printf("kein Speicher mehr!\n");
    } else
      if (wahl == '2')
      {
        if (pop(&wert))
          printf("Stack war leer!\n");
        else
          printf("Zahl auf dem Stack war %f\n",wert);
      }
  } while (wahl != 'e');

  while (!pop(&wert)); // Stack leeren
}
```

5.3. Queue

Die Queue bzw. Schlange ist vergleichbar mit dem Stack. Es ist ebenfalls eine einfach verkettete Liste. Allerdings erfolgt hier das Lesen immer am Anfang und das Schreiben immer am Ende der Liste. Dies wird auch das *First-In-First-Out*-Prinzip (FIFO) genannt. Diese Art Liste wird oft als Zwischenspeicher z.B. bei einer seriellen Datenübertragung verwendet. Analog zum Stack, wird das gelesene Element aus dem Speicher entfernt.
Im folgenden Beispiel dienen die zwei Funktionen *push()* und *pop()* zur Demonstration der Funktion. Wieder wird ein globaler Zeiger auf den Anfang der Liste verwendet. Im Unterschied zu den bisherigen Listen verwenden wir hier auch einen Zeiger auf das Ende der Liste. Da das Einfügen neuer Element immer am Ende erfolgt, ist so ein schneller Zugriff auf das Ende möglich, ohne dass zuvor die ganze Liste durchlaufen werden muss. Somit wird der *next*-Zeiger des Endes beim Einfügen auf das neue Element gesetzt und das Ende ist dann gleich dem neuen Element. Da das neue Element immer das Ende der Liste ist, wird dessen *next*-Zeiger auf NULL gesetzt.



Das Löschen erfolgt wie beim Stack immer vom Anfang.

Listing 5.8: Beispiel für Schlange (queue.cpp)

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

// Struktur
struct element
{
  float wert;
  struct element *next;
};
// Zeiger auf den Anfang und Ende der Liste
struct element *anfang = NULL, *ende = NULL;

int push(float zahl)
{
  struct element *neu;// Zeiger auf neues Element

  // Speicher fuer neues Element reservieren
  neu = (struct element*)malloc(sizeof(struct element));
  if (neu == NULL)
    return 1;
  // Werte zuweisen, an das Ende haengen
  neu->wert = zahl;
  neu->next = NULL;
  if (anfang == NULL)
    anfang = neu;
  else
    ende->next = neu;
  ende = neu; // Ende ist jetzt = neues Element
  return 0;
}

int pop(float *zahl)
{
  struct element *puffer;// Hilfsvariable

  if (anfang == NULL)
    return 1;
  // Zahl des ersten Elements holen
  *zahl = anfang->wert;
  // Anfang sichern, um 1 weiter verschieben
  puffer = anfang;
  anfang = anfang->next;
  // Speicher des bisherigen Anfangs freigeben
  free(puffer);
  return 0;
}

void main()
{
  float wert;
  char wahl;

  do
  {
    printf("\n");
    printf("1: in Schlange speichern (push)\n");
    printf("2: aus Schlange lesen (pop)\n");
    printf("e: Ende\n");
    printf("Ihre Wahl: \n");

    wahl = getchar();
    fflush(stdin);

    if (wahl == '1')
    {
      printf("\nZahl eingeben: ");
```

```
      scanf("%f",&wert);
      if (push(wert))
        printf("kein Speicher mehr!\n");
    } else
      if (wahl == '2')
      {
        if (pop(&wert))
          printf("Schlange war leer!\n");
        else
          printf("Zahl in der Schlange war %f\n",wert);
      }
  } while (wahl != 'e');

  while (!pop(&wert)); // Schlange leeren
}
```

6. Modularisierung

Unter modularer Programmierung versteht man meist die Unterteilung eines Programms in mehrere Quellcodedateien. Sinnvollerweise enthalten Module logisch zusammengehörigen Funktionen.

6.1. Module in C

Ein Modul in C besteht im Normalfall aus einer Quellcodedatei (*dateiname*.c) und der dazugehörigen Headerdatei (*dateiname*.h). Wenn nun ein Modul Funktionen aus einem anderen benutzen möchte, so geschieht dies durch Inkludieren der jeweiligen Headerdatei. Der Einsatz von Modulen bringt viele Vorteile:

- Mehrere Quellcodedateien erleichtern die Arbeit mit mehreren Entwicklern
- Bessere Trennung von Funktionsschichten im Programm (z.B. Modul für I/O, Modul für Kernfunktionalität, ...)
- Logische Funktionsblöcke können in Module gekapselt werden
- Bessere Wiederverwendbarkeit (reuse) von Programmteilen
- Austausch von Modulen wird möglich
- Bessere Schnittstellendefinition im Programm

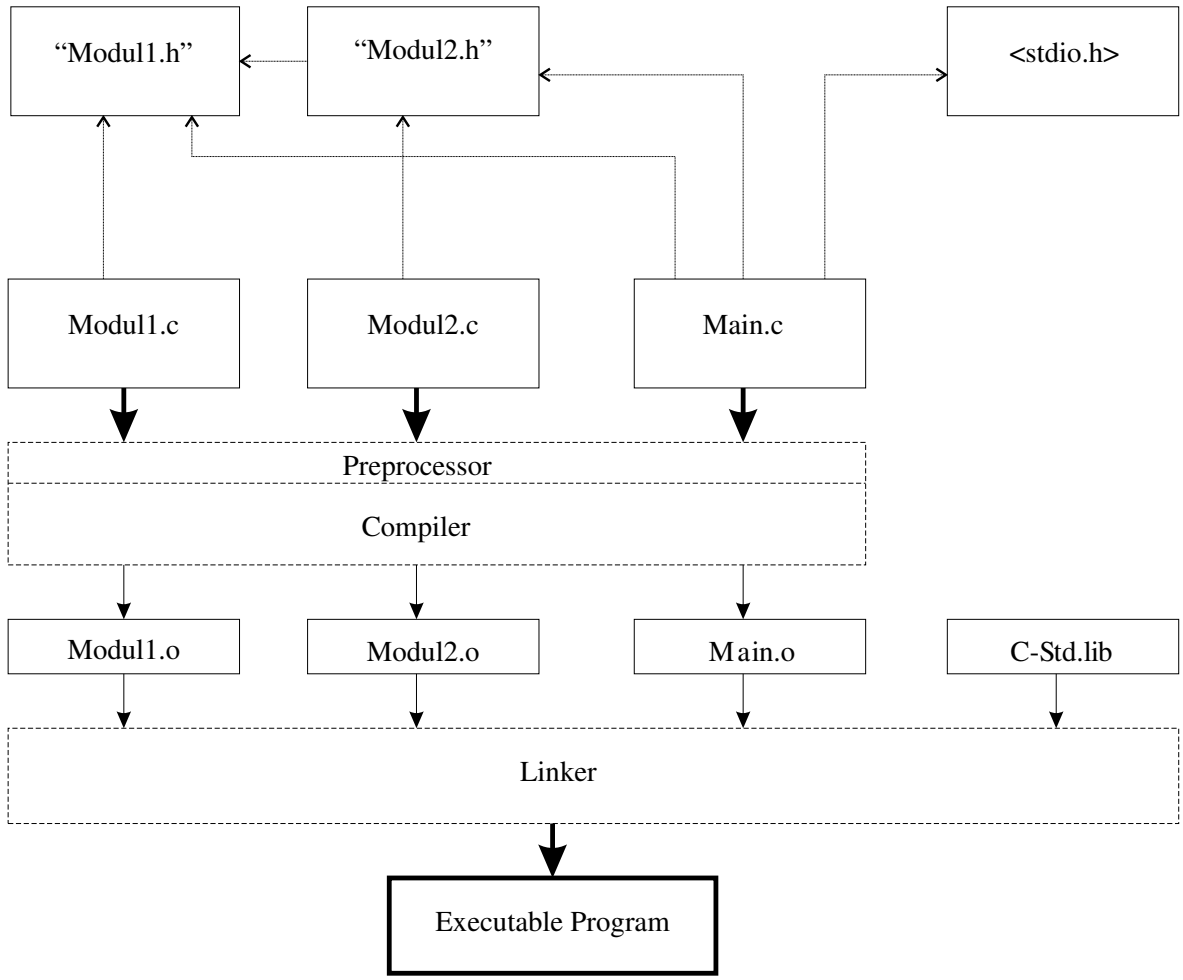
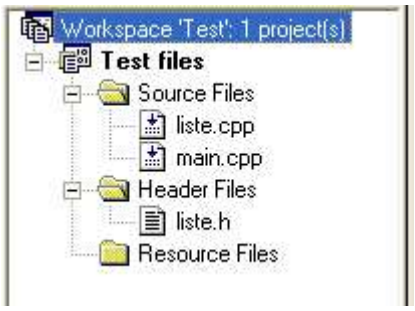


Abbildung 6.1 zeigt die Entstehung eines Programms aus mehreren Modulen.
Die Module werden vom Compiler einzeln in Objektcodedateien (*dateiname*.o oder *dateiname*.obj) umgewandelt. Der Linker sorgt dann dafür, dass die einzelnen Objektcodedateien zu einem ausführbaren Programm verbunden (gelinkt) werden.

6.2. Anwendung mit Visual C++

Bisher haben wir nur mit einer Quellcodedatei (z.B. main.cpp) in unseren Projekten gearbeitet. Wollen wir nun verschiedene Module erzeugen, benötigen wir außer diesem „Hauptmodul“ noch für jedes zusätzliche Modul eine Quellcode- und eine Headerdatei. Die einzelnen Dateien lassen sich über Visual C++ einfach über *Datei–>Neu* in das Projekt einbinden. Einmal im Projekt, werden sie beim Compilieren automatisch übersetzt und bei Bedarf zum Programm gelinkt.
Als Beispiel wollen wir die Verwaltung einer einfach verketteten Liste aus dem Kapitel 5.1.1 in einem Modul zusammenfassen. Dazu legen wir im Visual C++ Projekt zwei neue Dateien *liste.cpp* und *liste.h* an.



Die Datei *liste.h* könnte dann wie folgt aussehen:

Listing 6.1: Headerdatei *liste.h*

```
// Strukturdeklaration
struct list_element
{
  char name[50];
  struct list_element *naechstes;
};

// Funktionsprototypen
void element_anhaengen();
void liste_loeschen();
void liste_ausgeben();
void element_loeschen();
void element_einfuegen();
```

Sie enthält nur noch die Funktionsprototypen und evtl. die Deklaration der Struktur. Da der Zeiger auf den Anfang der Liste im Hauptprogramm nicht benötigt wird, erscheint er auch in der Headerdatei nicht und kann somit außerhalb des Moduls nicht verändert werden!

Die Quellcodedatei *liste.cpp* enthält dann alle Funktionsdefinitionen.

Listing 6.2: Quellcodedatei *liste.cpp*

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

// Zeiger auf den Anfang der Struktur
struct list_element *anfang = NULL;

void liste_loeschen()
{
  struct list_element *aktuell;  // Zeiger auf aktuelle Struktur

  while (anfang != NULL)
  {
    aktuell = anfang;// Adresse des ersten Elements sichern
    anfang = aktuell->naechstes;// Anfang verschieben
    free(aktuell);// Speicher freigeben
  }
}

void element_einfuegen()
{  // Zeiger auf aktuelles, neues und naechstes Element
  struct list_element *neu,*aktuell,*next;
  char suche[50];
```

```
  // Speicher fuer neues Element reservieren
  neu = (struct list_element*) malloc(sizeof(struct list_element));
  if (neu == NULL)
  {
    printf("\n\n*** nicht ügengend Speicherplatz! ***\n\n");
    return;
  }

  printf("\nBitte neuen Namen eingeben: ");
  gets(neu->name);
  printf("\nEinfuegen vor Name: ");
  gets(suche);

  // neues Element = neuer Anfang?
  if (!strcmp(anfang->name,suche))
  {
    neu->naechstes = anfang;
    anfang = neu;
  } else
  {
    aktuell = anfang;
    while (aktuell != NULL)
    {  // Name des naechsten Elements vergleichen
      next = aktuell->naechstes;
      if (next != NULL)
      {  // wenn Name identisch, dann Element einhaengen
        if (!strcmp(next->name,suche))
        {
          neu->naechstes = next;
          aktuell->naechstes = neu;
          break;
        }
      }
      aktuell = aktuell->naechstes;
    }
  }
}

void liste_ausgeben()
{
  struct list_element *aktuell;  // Zeiger auf aktuelle Struktur
  int nummer = 1;  // Zaehler fuer Nummerierung

  aktuell = anfang;// von Anfang an beginnen
  printf("\n\n");
  while (aktuell != NULL)
  {  // Name und Nummer ausgeben
    printf("%d. Name: %s\n",nummer,aktuell->name);
    aktuell = aktuell->naechstes;// naechstes Element
    nummer++;
  }
}

void element_anhaengen()
{
  // Zeiger auf aktuelles und neues Element
  struct list_element *neu,*aktuell;

  // Speicher fuer ein neues Element reservieren
  neu = (struct list_element*) malloc(sizeof(struct list_element));
  if (neu == NULL)
  {
    printf("\n\n*** nicht ügengend Speicherplatz! ***\n\n");
    return;
  }

  printf("\nBitte neuen Namen eingeben: ");
  gets(neu->name);
```

```
  neu->naechstes = NULL;// da letztes Element = NULL

  if (anfang == NULL)
  {
    anfang = neu;// neuer Anfang
  } else
  {  // leztes Element suchen
    aktuell = anfang;
    while (aktuell->naechstes != NULL)
      aktuell = aktuell->naechstes;
    // neues Element als letztes anhaengen
    aktuell->naechstes = neu;
  }
}

void element_loeschen()
{  // Zeiger auf aktuelles und naechstes Element
  struct list_element *aktuell,*next;
  char auswahl[50];

  if (anfang != NULL)
  {
    printf("\nzu loeschender Name eingeben: ");
    gets(auswahl);

    aktuell = anfang;
    // kontrollieren ob erstes Element geloescht werden soll
    if (!strcmp(aktuell->name,auswahl))
    {
      anfang = aktuell->naechstes;
      free(aktuell);
      printf("Element gefunden und geloescht.\n");
    } else
    {
      while (aktuell != NULL)
      {  // Name des naechsten Elements vergleichen
        next = aktuell->naechstes;
        if (next != NULL)
        {  // Element loeschen wenn Name identisch
          if (!strcmp(next->name,auswahl))
          {  // Zeiger veraendern
            aktuell->naechstes = next->naechstes;
            free(next);
            printf("Element gefunden und geloescht.\n");
          }
        }
        aktuell = aktuell->naechstes;
      }
    }
  }
}
```

Hier ist auch der Zeiger auf den Anfang der Liste definiert. Alle benötigten Standardbibliotheken müssen ebenfalls über die Preprozessor-Direktive **#include <>** eingebunden werden. Um die Strukturdeklaration zu erhalten, wird die Datei *liste.h* ebenfalls eingebunden. Das Einbinden erfolgt hier über **Anführungsstriche**, da es sich nicht um eine Standardbibliothek handelt und die Headerdatei im aktuellen Projektpfad liegt! (siehe auch Kapitel 8

Das Hauptmodul *main.cpp* enthält nur noch die Funktion *main()*. Durch Einbinden der Headerdatei *liste.h* werden die Funktionen zur Bearbeitung der Liste bekannt gemacht.

Listing 6.3: Hauptmodul *main.cpp*

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

void main()
{
  char wahl;

  do
  {
    printf("\nNamensliste\n\n");
    printf("1 - neues Element ans Ende\n");
    printf("2 - Liste ausgeben\n");
    printf("3 - Element loeschen\n");
    printf("4 - neues Element einfuegen\n");
    printf("e - Ende\n\n");
    printf("Ihre Wahl: ");

    wahl = getchar();
    fflush(stdin);
    switch(wahl)
    {
    case '1': element_anhaengen();
          break;
    case '2': liste_ausgeben();
          break;
    case '3': element_loeschen();
          break;
    case '4': element_einfuegen();
          break;
    case 'e': break;
    default: printf("Ungueltige Eingabe!\n");
    }
  } while (wahl != 'e');

  liste_loeschen();
}
```

Damit kann der Compiler die beiden Module getrennt voneinander übersetzen. Das zusammenfügen der beiden Module zu einem fertigen Programm erfolgt nun erst beim Linker.

7. Sortieren

Sortierte Daten in der Informatik sind immer dann wichtig, wenn es um schnelle Zugriffe darauf geht. Das kann zum einen für die Ausgabe für den menschlichen Benutzer sein (stellen Sie sich vor, Ihr Telefonbuch wäre nicht sortiert...) oder auch um die Programmablaufgeschwindigkeit beim Bearbeiten von Daten zu steigern.
Computer suchen in unsortierten Daten zwar generell schneller als Menschen, jedoch lässt sich hier auch durch geeignete Wahl der Datensortierung ein riesiger Geschwindigkeitsgewinn erreichen.

7.1. Allgemeines

Für das Sortieren von Daten gibt es verschiedene Algorithmen. Diese Algorithmen kann man nach ihrer Komplexität, nach der benötigten Laufzeit (in Abhängigkeit zur Anzahl der zu sortierenden Elemente), sowie nach Ihrem Verhalten auf bestimmte Datensätze klassifizieren.

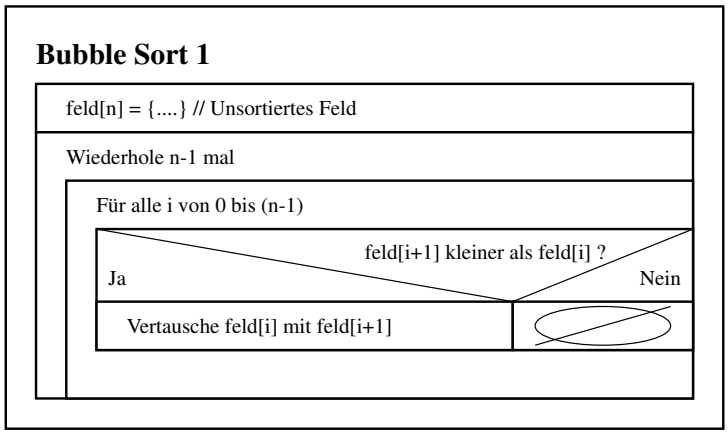
Als Eingabeparameter benötigen alle Sortieralgorithmen ein Feld mit Datensätzen, sowie eine Vergleichsfunktion „ist kleiner als“, um die Datensätze miteinander vergleichen zu können.
Die Vergleichsfunktion kann bei einem einfachen Zahlenfeld dann der „<“ - Operator sein.

7.2. Bubble-Sort

Der Bubble-Sort Algorithmus ist einer der einfachsten Sortieralgorithmen. Ausgehend von einem Feld mit n Elementen ist seine Beschreibung sehr einfach:

```
Wiederhole (n-1) mal:
  - Vergleiche jedes Element mit dem nächsten
    und tausche, falls falsch sortiert
```

Eine detailliertere Ablaufvorschrift liefert das Struktogramm:



Das folgende Programm zeigt die Anwendung von Bubblesort an einem Zahlenfeld mit 8 Elementen. Um die Vorgänge innerhalb des zu sortierenden Feldes besser zu verstehen, wird der Feldinhalt nach jedem Durchgang der inneren Schleife ausgegeben.

Listing 7.1: Bubble Sort 1 (bubblesort1.c)

```
#include <stdio.h>

void feldausgabe(int nr);
void tausche(int* x, int* y);

#define MAX 8
int feld[MAX] = {9,1,4,2,5,0,3,8};  // unsortiertes Zahlenfeld

int main()
{
  int durchlauf = 0, i;

  feldausgabe(durchlauf);   // Zeigt den Feldinhalt

  while(durchlauf < MAX)
  {
    for(i=0; i<MAX-1; i++)
    {
      if(feld[i+1] < feld[i]) tausche(&feld[i], &feld[i+1]);
    }

    durchlauf++;
    feldausgabe(durchlauf);
  }

  return 0;
}

void feldausgabe(int nr)
{
  int i;
  printf("Durchgang %2d: ", nr);
  for(i=0; i<MAX; i++) printf("%3d ", feld[i]);
  printf("\n");
}

void tausche(int* x, int* y)
{
  int dummy = *x;
  *x = *y;
  *y = dummy;
}
```

Ausgabe von Bubblesort 1:

```
Durchgang  0:   9   1   4   2   5   0   3   8
Durchgang  1:   1   4   2   5   0   3   8   9
Durchgang  2:   1   2   4   0   3   5   8   9
Durchgang  3:   1   2   0   3   4   5   8   9
Durchgang  4:   1   0   2   3   4   5   8   9
Durchgang  5:   0   1   2   3   4   5   8   9
Durchgang  6:   0   1   2   3   4   5   8   9
Durchgang  7:   0   1   2   3   4   5   8   9
Durchgang  8:   0   1   2   3   4   5   8   9
```

Durchgang 0 zeigt das unsortierte Feld. Nach jedem weiteren Durchgang ist zu sehen, dass jeweils die größte verbleibende noch unsortierte Zahl nun an der richtigen Stelle steht. Das Feld wird also „von oben“ her richtig sortiert. Bildlich gesprochen steigen die größten Zahlen wie Blasen im Wasser nach oben, daher auch der Name Bubblesort. Weiterhin ist zu sehen, dass nach dem 5. Durchgang keine Veränderung mehr stattfindet, da das Feld schon komplett sortiert vorliegt. Es gibt also durchaus noch Raum für Optimierungen im Algorithmus. Der einfache Bubblesort Algorithmus kann wie folgt klassifiziert werden:

- Komplexität: einfach
- Laufzeit: $O(n^2)$ (n Elemente führen zu n*n Vergleichen)
- Gleiche Laufzeit für alle Vorbedingungen im Feld

7.2.1. Optimierung Nr. 1

Es ist zu sehen, dass nach jedem Durchlauf jeweils ein weiteres Element „von oben“ her gesehen an der richtigen Stelle steht. Die innere Schleife muss also jeweils nicht alle Elemente durchlaufen, da sichergestellt ist, dass nach x Durchläufen die obersten x Elemente bereits korrekt sortiert vorliegen. Eine Laufzeitoptimierung kann also durch Einschränkung des Laufbereichs der inneren Schleife erreicht werden:

Listing 7.2: Bubble Sort 2 (verbesserte innere Schleife) (bubblesort2.c)

```
while(durchlauf < MAX)
{
  for(i=0; i<MAX-1-durchlauf; i++)
  {
    if(feld[i+1] < feld[i]) tausche(&feld[i], &feld[i+1]);
  }

  durchlauf++;
  feldausgabe(durchlauf);
}
```

Ausgabe von Bubblesort 2:

```
Durchgang  0:   9   1   4   2   5   0   3   8
Durchgang  1:   1   4   2   5   0   3   8   9
Durchgang  2:   1   2   4   0   3   5   8   9
Durchgang  3:   1   2   0   3   4   5   8   9
Durchgang  4:   1   0   2   3   4   5   8   9
Durchgang  5:   0   1   2   3   4   5   8   9
Durchgang  6:   0   1   2   3   4   5   8   9
Durchgang  7:   0   1   2   3   4   5   8   9
Durchgang  8:   0   1   2   3   4   5   8   9
```

Es werden weiterhin alle Durchläufe gemacht, jedoch verbraucht die innere Schleife weniger Zeit. Diese Optimierung wird bei den heutigen Rechnern natürlich erst bei großen Datenmengen spürbar. Der verbesserte Bubblesort Algorithmus kann wie folgt klassifiziert werden:

- Komplexität: einfach
- Laufzeit: Vergleiche: $n * \frac{n}{2}$ entspricht $O(n^2)$, da 2 ein linearer Faktor ist
- Gleiche Laufzeit für alle Vorbedingungen im Feld

7.2.2. Optimierung Nr. 2

Im letzten Optimierungsschritt können nun noch die unnötigen Durchläufe entfernt werden. Dies kann in der Praxis einen erheblichen Geschwindigkeitsvorteil bedeuten, da oftmals schon teilweise sortierte Felder vorliegen. Um zu erkennen, wann das Feld sortiert ist wird in der äußeren Schleife geprüft, ob in der inneren Schleife noch ein Tausch stattgefunden hat:

Listing 7.3: Bubble Sort 3 (verbesserte Schleifen) (bubblesort3.c)

```
int durchlauf = 0, getauscht = 1, i;

feldausgabe(durchlauf);   // Zeigt den Feldinhalt

while(getauscht)
{
  getauscht = 0;

  for(i=0; i<MAX-1-durchlauf; i++)
  {
    if(feld[i+1] < feld[i])
    {
      tausche(&feld[i], &feld[i+1]);
      getauscht = 1;
    }
  }

  durchlauf++;
  feldausgabe(durchlauf);
}
```

Ausgabe von Bubblesort 3:

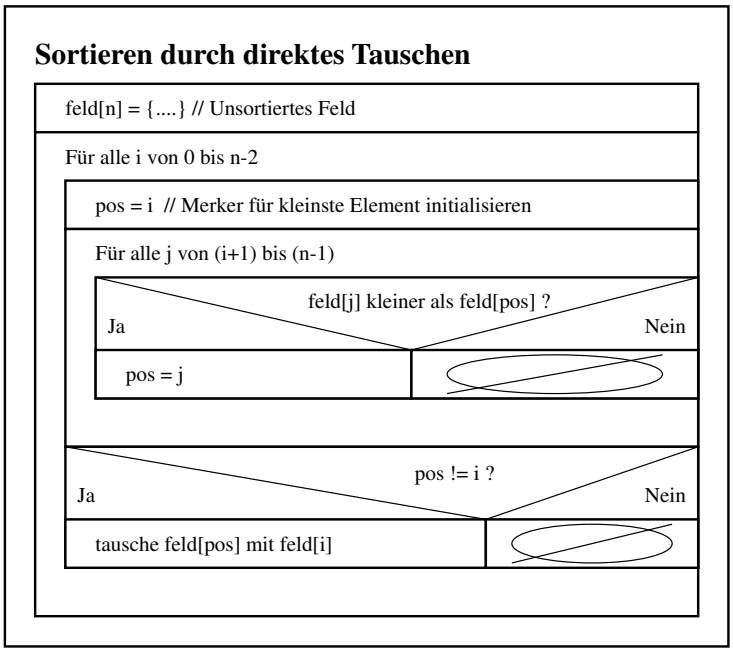
```
Durchgang  0:   9   1   4   2   5   0   3   8
Durchgang  1:   1   4   2   5   0   3   8   9
Durchgang  2:   1   2   4   0   3   5   8   9
Durchgang  3:   1   2   0   3   4   5   8   9
Durchgang  4:   1   0   2   3   4   5   8   9
Durchgang  5:   0   1   2   3   4   5   8   9
Durchgang  6:   0   1   2   3   4   5   8   9
```

Die Ausgabe zeigt, dass nach dem ersten Durchlauf der inneren Schleife in der keine Elemente mehr getauscht werden, der Algorithmus abbricht. Das Setzen des getauscht-Flags in der inneren Schleife benötigt zwar etwas mehr Zeit, jedoch kann diese Zeit durch das Einsparen von vielen Durchläufen wieder wett gemacht werden. Der Optimierungsschritt Nr. 1 kann natürlich auch hier erhalten bleiben. Der weiter verbesserte Bubblesort Algorithmus kann wie folgt klassifiziert werden:

- Komplexität: einfach
- Laufzeit: $O(n^2)$ (worst case Betrachtung)
- Verbesserte Laufzeit für bestimmte vor sortierte Felder

7.3. Sortieren durch direktes Tauschen

Einer der Nachteile des Bubblesort Algorithmus ist die öfter auftauchende unnötige Vertauschung von Elementen. Dies kann in jeder Optimierungsstufe des Bubblesort Algorithmus beobachtet werden. Um diesen Nachteil abzuschaffen kann der Algorithmus „Sortieren durch direktes Tauschen“ verwendet werden. Dieser Algorithmus sucht im gesamten Feld nach dem kleinsten Element und stellt dieses an die erste Stelle. Danach wird ab dem 2. Feldindex das kleinste Element gesucht und an die 2. Stelle gesetzt, usw. Die Anzahl der Tauschvorgänge (Achtung: diese fließt nicht in die Laufzeitabschätzung ein) kann somit bei n Elementen auf maximal n-1 reduziert werden.
Das Struktogramm zeigt die Verarbeitungsvorschrift des Algorithmus:



Das folgende Programm zeigt die Anwendung des Algorithmus an einem Zahlenfeld mit 8 Elementen. Auch hier wird der Feldinhalt nach jedem inneren Schleifendurchlauf ausgegeben.

Listing 7.4: Sortieren durch direktes Tauschen (sortDirektesTauschen.c)

```
#include <stdio.h>

void feldausgabe(int feld[], int laenge, int nr);
void sortiere_durch_tauschen(int feld[], int laenge);
void tausche(int* x, int* y);

int main()
{
  int feld[] = {9,1,4,2,5,0,3,8};  // unsortiertes Zahlenfeld mit 8 Elementen
  sortiere_durch_tauschen(feld, 8);
  return 0;
}


void sortiere_durch_tauschen(int feld[], int laenge)
{
  int i, j, pos;

  feldausgabe(feld, laenge, 0);

  for(i=0; i < laenge-1; i++)
  {
    pos = i;

    for(j = i+1; j<laenge;j++)
    {
      if(feld[j] < feld[pos])    // neues kleinstes Element gefunden
      {
        pos = j;
      }
    }
    if(pos != i) tausche(&feld[pos], &feld[i]);
    feldausgabe(feld, laenge, i+1);
  }
}
```

```
void feldausgabe(int feld[], int laenge, int nr)
{
  int i;
  printf("Durchgang␣%2d:␣", nr);
  for(i=0; i<laenge; i++) printf("%3d␣", feld[i]);
  printf("\n");
}

void tausche(int* x, int* y)
{
  int dummy = *x;
  *x = *y;
  *y = dummy;
}
```

Ausgabe von Sortieren durch direktes Tauschen:

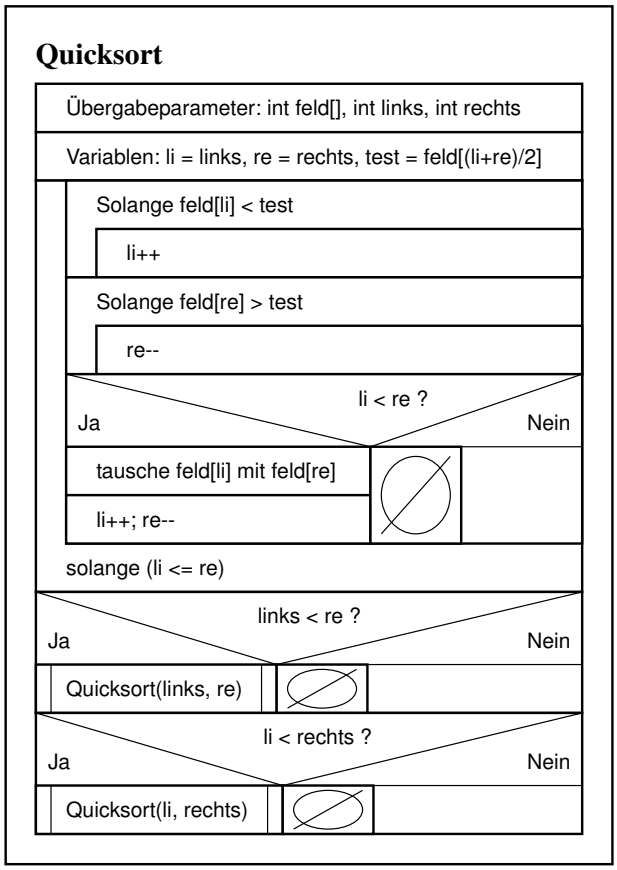
```
Durchgang  0:   9   1   4   2   5   0   3   8
Durchgang  1:   0   1   4   2   5   9   3   8
Durchgang  2:   0   1   4   2   5   9   3   8
Durchgang  3:   0   1   2   4   5   9   3   8
Durchgang  4:   0   1   2   3   5   9   4   8
Durchgang  5:   0   1   2   3   4   9   5   8
Durchgang  6:   0   1   2   3   4   5   9   8
Durchgang  7:   0   1   2   3   4   5   8   9
```

Um die Wiederverwendung des Algorithmus in anderen Programmen zu erleichtern, ist im obigen Beispiel der Algorithmus selbst als Funktion implementiert. Um die Unabhängigkeit der Funktion von globalen Variablen zu gewährleisten müssen das zu sortierende Feld sowie die Anzahl der Elemente im Feld als Funktionsparameter übergeben werden. Der Sortieren durch direktes Tauschen Algorithmus kann wie folgt klassifiziert werden:

- Komplexität: einfach
- Laufzeit: $O(n^2)$
- Leichte Optimierung bei schon sortiertem Feld, sonst gleich für alle Vorbedingungen

7.4. Quicksort

Alle bisherigen Sortieralgorithmen haben die Laufzeitklassifikation von $O(n^2)$. Der Quicksort Algorithmus arbeitet rekursiv und kommt auf ein Laufzeitverhalten von $O(n*ld(n))$. Der Ablauf von Quicksort wird im Struktogramm wie folgt definiert:



Listing 7.5: Quicksort (quicksort.c)

```
#include <stdio.h>

void feldausgabe(int feld[], int laenge, int links, int rechts);
void quicksort(int feld[], int links, int rechts);
void tausche(int* x, int* y);

int main()
{
  int feld[] = {9,1,4,2,5,0,3,8};  // unsortiertes Zahlenfeld mit 8 Elementen
  feldausgabe(feld, 8, 0, 7);
  quicksort(feld, 0, 7);           // Quicksort auf gesamtes Feld
  return 0;
}


void quicksort(int feld[], int links, int rechts)
{
  int li = links, re = rechts;
  int test = feld[(li+re)/2];    // Mittleres Element als Entscheidungskriterium

  do {
    while( feld[li] < test ) li++;
    while( feld[re] > test) re--;

    if(li <= re)
    {
      tausche(&feld[li], &feld[re]);
      li++; re--;
    }
  }
```

```
  while( li <= re);

  feldausgabe(feld, 8, links, rechts);

  if(links < re)  quicksort(feld, links, re);
  if(li < rechts) quicksort(feld, li, rechts);

}

void feldausgabe(int feld[], int laenge, int links, int rechts)
{
  static int nr = 0;
  int i;
  printf("Aufruf␣%2d:␣", nr);
  for(i=0; i<laenge; i++) printf("%3d␣", feld[i]);
  printf("\n");
  printf("␣␣␣␣␣␣␣␣␣␣␣␣");
  for(i=0; i<laenge; i++)
    if(i == links)  printf("␣␣L␣");
    else if(i == rechts)  printf("␣␣R␣");
    else            printf("␣␣␣␣");
  printf("\n");
  nr++;
}

void tausche(int* x, int* y)
{
  int dummy = *x;
  *x = *y;
  *y = dummy;
}
```

Ausgabe von Quicksort (Die linke und rechte Grenze wird für jeden Aufruf angezeigt):

```
Aufruf  0:   9   1   4   2   5   0   3   8
             L                       R
Aufruf  1:   0   1   2   4   5   9   3   8
             L                       R
Aufruf  2:   0   1   2   4   5   9   3   8
             L       R
Aufruf  3:   0   1   2   4   5   8   3   9
                         L                R
Aufruf  4:   0   1   2   4   3   8   5   9
                         L           R
Aufruf  5:   0   1   2   3   4   8   5   9
                         L   R
Aufruf  6:   0   1   2   3   4   5   8   9
                             L   R
```

Durch jeweiliges Zerlegen des Originalfeldes in 2 zueinander richtig sortierten Felder wird bei jedem weiteren rekursiven Aufruf von Quicksort im Schnitt jeweils nur noch die Hälfte der Anzahl der Elemente miteinander verglichen. Dadurch kommt der Quicksortalgorithmus auf sein besseres Laufzeitverhalten.

Der Quicksort Algorithmus kann wie folgt klassifiziert werden:

- Komplexität: schwierig
- Laufzeit: $O(n * ld(n))$
- Ungünstige Wahl des Testelements verlangsamt den Algorithmus

Hinweis: Eine generische Implementierung des Quicksortalgorithmus ist bereits in der C Bibliothek „stdlib.h“ mit dem Funktionsnamen **qsort** enthalten. Da die Funktion beliebige Datentypen akzeptiert, ist ihr Aufruf jedoch entsprechend kompliziert. Es wird hier nicht weiter darauf eingegangen.

8. Präprozessor

Der Präprozessor verarbeitet den Quellcode einer Programmdatei bevor diese dem eigentlichen Compiler zur Übersetzung übergeben wird. Im eigentlichen Sinn ist der Präprozessor ein einfacher „Textmanipulator“, welcher den Quellcode vor der Übergabe an den Compiler entsprechend den Anweisungen manipuliert. Er kennt, wie der eigentliche Compiler, verschiedene Kommandos (Präprozessor-Direktiven), die irgendwo im Programmcode stehen können und die Arbeit den Präprozessor beeinflussen.

Die folgenden Kommandos werden vom Präprozessor angeboten:

- Bedingte Kompilierung: **#ifdef, ...**
  Damit lassen sich Programmteile bei der Kompilierung je nach Bedingung ein- oder ausblenden
- Einkopieren anderer Dateien: **#include**
  Dieses bereits bekannte Kommando kopiert die angegebene Datei an diese Stelle in den Quellcode
- Makro-Ersetzung: **#define**
- Zeilennummerierung: **#line**
  Beeinflusst die Zeilennummerierung für die Compilerfehlermeldungen
- Fehlermeldungen: **#error**
  Das Kommando gibt einen Fehlertext auf dem Bildschirm aus und unterbricht den Übersetzungsvorgang
- Pragmas: **#pragma**
  Beeinflusst den Compiler über verschiedene Parameter (Compilerspezifisch)

Alle Kommandos beginnen mit dem Zeichen #. In derselben Zeile vor diesem Zeichen sind nur Zwischenraum-Zeichen (Leerzeichen, Tabulatoren) erlaubt! Zur besseren Übersicht stehen die Präprozessor-Kommandos deshalb meist ab der ersten Spalte einer Zeile.

8.1. Einkopieren anderer Dateien (#include)

Dieses bereits bekannte Kommando kopiert die angegebene Datei in den Quellcode. Bisher wurde dieses Kommando zur Einbindung der Headerdateien verwendet. Dies ist auch die übliche Verwendung dieses Kommandos. Natürlich lassen sich auch Quellcodedateien

(*.cpp) oder sonstige Dateien damit einkopieren.
Zur Verwendung gibt es 2 Formen, die sich durch die Angabe der Datei unterscheiden:

#include <dateiname>:	Durchsucht den Standardpfad nach der angegebenen Datei (meist etwas in der Form „\vc\include“)
#include ”dateiname”:	Beginnt im aktuellen Projektpfad mit der Suche und erst wenn die Datei dort nicht gefunden wurde im Standardpfad

Üblicherweise werden die spitzen Klammern bei Standardbibliotheken (stdio.h, stream.h, stdlib.h, math.h usw.) verwendet und die Anführungszeichen bei benutzereigenen Headerdateien (z.B. bei der modularen Programmierung).

8.2. Definition von Makros (#define)

Das #define Kommando haben wir bereits bei den Konstanten kennen gelernt. Im einfachsten Fall tritt das Kommando in der Form

#define bezeichner Ersetzungstext

auf. Überall wo jetzt im Quellcode der Text *bezeichner* zu finden ist, ersetzt der Präprozessor diesen Text durch *Ersetzungstext*. Zu beachten ist, dass am Ende der #define Anweisung *kein* Semikolon steht, da dieses sonst ebenfalls in den Quellcode eingefügt würde!
Üblicherweise werden die *bezeichner* immer in Großbuchstaben geschrieben, um sie von sonstigen Variablen oder Funktion zu unterscheiden. Wird der Ersetzungstext für eine Zeile zu lang kann er durch einen einfachen Backslash umgebrochen werden. Hier noch einige Beispiele für Konstantendefinitionen:

```
#define MWST        0.16
#define PI          3.1415
#define MAX_ZAHLEN  10000
#define TEXT        ''Dies ist ein Text, welcher \
                      sich über mehrere Zeilen erstreckt''

// Beispiele für Verwendung
int zahlen[MAX_ZAHLEN];
printf(TEXT);
double umfang = 2.0 * PI * radius;
```

Da bei **#define** eine rein textliche Ersetzung stattfindet, kann es auch zur Umbenennung von Funktionen oder Schlüsselwörter verwendet werden, wie z.B.

```
#define WENN     if
#define SONST    else
```

8.2.1. Makros

In manchen Fällen ist eine in sich abgeschlossene Teilaufgabe derart kurz, dass es nicht lohnt, diese als Funktion anzugeben und einen Ein- und Rücksprung erforderlich zu machen.
C bietet nun die Möglichkeit, solche kurzen Programmteile als *Makros* zu definieren:

#define makroname(param1, param2, ...) makrobefehl(e)

In den Klammern nach *makroname* kann dabei eine Liste von formalen Parametern angegeben werden, die beim Makroaufruf durch die aktuellen Argumente ersetzt werden. Zu beachten ist, dass zwischen dem Makronamen und der öffnenden Klammer (kein Leerzeichen oder Tabulator stehen darf!

Listing 8.1: Bestimmung des Maximums zweier Zahlen mit Makro (makro_max.cpp)

```
#include <stdio.h>

#define MAX(a,b)  (a)>(b) ? (a):(b)

void main(void)
{
   int  zahl_1, zahl_2;

   printf("Geben␣Sie␣2␣ganze␣Zahlen␣durch␣Komma␣getrennt␣ein:␣");
   scanf("%d,%d", &zahl_1, &zahl_2);

   printf("␣␣␣--->␣Maximum␣dieser␣beiden␣Zahlen␣ist␣%d\n", MAX(zahl_1,zahl_2));
}
```

Das Beispiel definiert ein Makro als Vergleich der beiden Parameter a und b. Die Kurzform der if-else Verzweigung gibt entweder den Parameter a zurück, sofern dieser größer ist als b, oder sie gibt b zurück, wenn a kleiner ist als b. Alle Parameter in den Makro-Anweisungen sollten in runden Klammern stehen.
Man hätte diesen Vergleich auch direkt in den Aufruf der Funktion printf() schreiben können. Aber da solch ein Vergleich evtl. öfter benötigt wird, wurde er hier zu Vereinfachung als Makro definiert.

Unterschiede zwischen Makros und Funktionen

Im folgenden werden kurz die wichtigsten Unterschiede, Vor- und Nachteile zwischen Makros und Funktionen dargestellt.

- **Keine Typfestlegung von Makro-Parametern**
  Betrachtet man das Beispiel 8.1 so sieht man, dass für die Makroparameter a und b keine Typen festgelegt wurden. Damit kann das Makro mit allen Grunddatentypen (int, char, float..) verwendet werden. Bei einer Funktion müssten hingegen für jeden Typ eine eigene Funktion geschrieben werden.

- **Codeaufblähung durch Makros**
  Bei umfangreichen Makros, die sehr oft in einem Programm aufgerufen werden, kann es zur Aufblähung der entsprechenden Objektdatei kommen, da die zu einem

Makro gehörigen Anweisungen vom Präprozessor einfach an jede entsprechende Aufrufstelle kopiert werden. Bei Funktionen ist dies nicht der Fall, da hier der Funktionscode nur einmal an einer bestimmten Speicheradresse hinterlegt wird, und dann bei jedem Funktionsaufruf lediglich angesprungen wird.

- **Makros laufen schneller ab als Funktionen**
  Bedingt durch obige Sprünge, erfolgt der Aufruf der Funktionen etwas langsamer. Ebenfalls entfällt bei Makros das ganze Stack-Management, das bei Funktionen erforderlich ist.

8.3. Bedingte Kompilierung (#ifdef...)

Folgende Schlüsselwörter stehen für die bedingte Kompilierung zur Verfügung:

- #if *ausdruck*
  Abhängig davon, ob *ausdruck* erfüllt ist, wird der darauf folgende Programmteil kompiliert
- #ifdef *name*
  Wenn *name* definiert ist, dann wird der darauf folgende Programmteil kompiliert
- #ifndef *name*
  Wenn *name* nicht definiert ist, dann wird der darauf folgende Programmteil kompiliert
- #elif *ausdrcuk*
  Abhängig davon, ob *ausdruck* erfüllt ist, wird der darauf folgende Programmteil kompiliert
- #else
  Leitet *else*-Programmteil zu den 4 vorigen Konstrukten (#if, #ifdef, #ifndef,#elif) ein
- #endif
  Zeigt das Ende einer bedingten Kompilierung an

Durch die bedingte Kompilierung hat man die Möglichkeit, nur eine Quelldatei zu unterhalten, die dann von unterschiedlichen Compilern und sogar auf unterschiedlichen Maschinen oder zu unterschiedlichen Zwecken (Demo, Vollversion) übersetzt werden kann. Beispiel: Anpassung an unterschiedliche Compiler oder Betriebssysteme

```
#ifdef ANSI_C
   long double pi;
#else
   long float pi;
#endif
```

```
#ifdef WINDOWS
  #include "windows.h"
#elif UNIX
  #include "unix.h"
#else
  #include "dos.h"
#endif
```

Die einzelnen Konstanten können entweder über **#define WINDOWS** oder besser über einen Compilerschalter in der Entwicklungsumgebung angegeben werden.

Häufig benötigt man in der Testphase einer Software zusätzliche Ausgaben auf dem Bildschirm, die in der endgültigen Ausgabe natürlich nicht mehr erscheinen dürfen. Am einfachsten kann dies durch eine bedingte Kompilierung erreicht werden:

```
#ifdef TEST
void Ausgabe()
{
 ...
}
#endif

void main()
{
  ...
#ifdef TEST
  Ausgabe()
#endif
  ...
}
```

Während der Testphase ist nun die Konstante TEST definiert und somit wird die Funktion Ausgabe() und der Aufruf in der main-Funktion übersetzt. In der endgültigen Fassung der Software ist diese Konstante nicht mehr definiert und somit „löscht“ der Präprozessor den Code für die Ausgabefunktion aus dem Quelltext, bevor dieser an den Compiler übergeben wird. Man hat somit **eine** Quellcodedatei, sowohl für die Testversion als auch für die endgültige Version.

In der Software-Entwicklung kann es passieren, dass die gleiche Headerdatei mehrmals in ein Programm einkopiert wird, was dann zu Fehlermeldungen führt, da die gleichen Definitionen und Deklarationen dann mehrfach in dem Programm vorhanden sind. Um dieses Problem zu vermeiden, baut man die Headerdatei (hier als Beispiel die Datei „c.h“) bei der modularen Programmierung häufig folgendermaßen auf:

```
#ifndef C
#define C
....
....
#endif
```

Nachdem nun die Headerdatei einmal im Programm eingebunden wurde, ist die Konstante **C** definiert, was dazu führt, dass der Inhalt der Headerdatei nicht ein zweites mal in das Programm übernommen wird. Für die Konstanten verwendet man üblicherweise den Namen der Headerdatei.

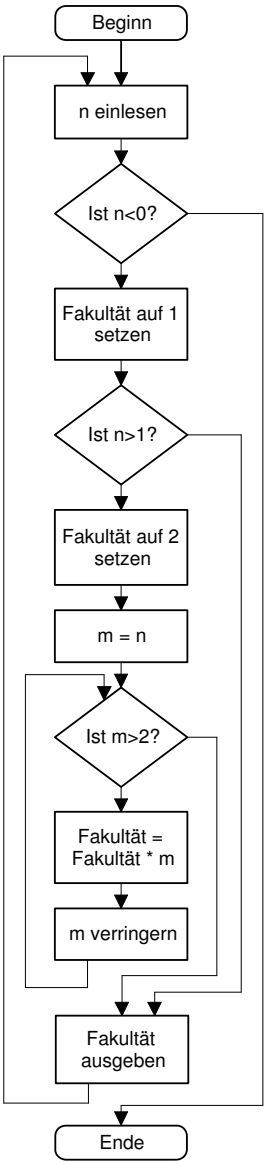
A. ASCII-Tabelle

				b7 b6 b5	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Bits					Steuer-		Symbole		Groß-		Klein-	
b4	b3	b2	b1		zeichen		Ziffern		buchstaben		buchstaben	
0	0	0	0		0 NUL 0 0	16 DLE 20 10	32 SP 40 20	48 0 60 30	64 @ 100 40	80 P 120 50	96 ‘ 140 60	112 p 160 70
0	0	0	1		1 SOH 1 1	17 DC1 21 11	33 ! 41 21	49 1 61 31	65 A 101 41	81 Q 121 51	97 a 141 61	113 q 161 71
0	0	1	0		2 STX 2 2	18 DC2 22 12	34 ” 42 22	50 2 62 32	66 B 102 42	82 R 122 52	98 b 142 62	114 r 162 72
0	0	1	1		3 ETX 3 3	19 DC3 23 13	35 # 43 23	51 3 63 33	67 C 103 43	83 S 123 53	99 c 143 63	115 s 163 73
0	1	0	0		4 EOT 4 4	20 DC4 24 14	36 $ 44 24	52 4 64 34	68 D 104 44	84 T 124 54	100 d 144 64	116 t 164 74
0	1	0	1		5 ENQ 5 5	21 NAK 25 15	37 % 45 25	53 5 65 35	69 E 105 45	85 U 125 55	101 e 145 65	117 u 165 75
0	1	1	0		6 ACK 6 6	22 SYN 26 16	38 & 46 26	54 6 66 36	70 F 106 46	86 V 126 56	102 f 146 66	118 v 166 76
0	1	1	1		7 BEL 7 7	23 ETB 27 17	39 ’ 47 27	55 7 67 37	71 G 107 47	87 W 127 57	103 g 147 67	119 w 167 77
1	0	0	0		8 BS 10 8	24 CAN 30 18	40 (50 28	56 8 70 38	72 H 110 48	88 X 130 58	104 h 150 68	120 x 170 78
1	0	0	1		9 HT 11 9	25 EM 31 19	41) 51 29	57 9 71 39	73 I 111 49	89 Y 131 59	105 i 151 69	121 y 171 79
1	0	1	0		10 LF 12 A	26 SUB 32 1A	42 * 52 2A	58 : 72 3A	74 J 112 4A	90 Z 132 5A	106 j 152 6A	122 z 172 7A
1	0	1	1		11 VT 13 B	27 ESC 33 1B	43 + 53 2B	59 ; 73 3B	75 K 113 4B	91 [133 5B	107 k 153 6B	123 { 173 7B
1	1	0	0		12 FF 14 C	28 FS 34 1C	44 , 54 2C	60 < 74 3C	76 L 114 4C	92 \ 134 5C	108 l 154 6C	124 | 174 7C
1	1	0	1		13 CR 15 D	29 GS 35 1D	45 _ 55 2D	61 = 75 3D	77 M 115 4D	93] 135 5D	109 m 155 6D	125 } 175 7D
1	1	1	0		14 SO 16 E	30 RS 36 1E	46 . 56 2E	62 > 76 3E	78 N 116 4E	94 ∧ 136 5E	110 n 156 6E	126 ∼ 176 7E
1	1	1	1		15 SI 17 F	31 US 37 1F	47 / 57 2F	63 ? 77 3F	79 O 117 4F	95 - 137 5F	111 o 157 6F	127 DEL 177 7F

B. Übungen

B.1. Übung zu Algorithmen

1. **Umformen Programmablaufplan in Struktogramm** (3 Punkte)
   Formen Sie folgenden unstrukturierten Programmablaufplan in ein Struktogramm um. Bedenken Sie dabei, dass es in einem Struktogramm keine Sprünge gibt!



2. **Datenhierarchien** (3 Punkte)
   Erstellen sie zu folgenden Strukturen jeweils ein Datenhierarchie-Diagramm:

```
struct person
{
  char vorname[30];
  char nachname[30];
};

struct adresse
{
  int plz;
  char ort[40];
  char strasse[50];
  int hausnummer;
};

struct mitarbeiter
{
  struct person name;
  int alter;
  struct adresse anschrift;
  struct mitarbeiter *vorgesetzter;
};
```

B.2. Übungen zu Zeigern

3. **Zeigerübungen** (1 Punkt)
   Im folgenden wird angenommen, dass die erste Variable auf die fiktive Adresse 1000 im Speicher gelegt wird. Welcher Wert steht nach jeder Anweisung in der jeweiligen Variablen?

Listing B.1: Übung Zeiger

```
#include <sdtio.h>

void main()
{
  int i = 9;
  int *pi;
  pi = &i;
  printf("i=%d␣&i=%u\n",i,&i);
  printf("*pi=%d␣pi=%u\n",*pi,pi);
  *pi = 3;
  printf("i=%d␣&i=%u\n",i,&i);
  printf("*pi=%d␣pi=%u\n",*pi,pi);
  printf("&pi=%u\n",&pi);
}
```

4. **Variablenrotation** (1 Punkt)
   Erstellen Sie eine Funktion die 3 Zahlen (int) übergeben bekommt. Die Funktion soll die Variableninhalte rotieren (z.B. Übergabe: a, b, c. Danach soll der Inhalt von a=b, b=c, und c=a sein).

5. **otto aus n und s** (3 Punkte)
   Erstellen Sie ein Programm, in dem Sie zunächst den beiden char-Variablen n und s

die Buchstaben 'n' und 's' zuweisen. Als einzige weitere Deklaration soll die Zeiger-Deklaration **char *zgr;** erlaubt sein. Das Programm soll nun *otto* ausgeben, wobei jedoch nur noch mit der Variablen *zgr* gearbeitet werden soll. Ein Zugriff auf die beiden Variablen n und s soll dabei nur noch erfolgen, indem die Adressen dieser Variablen abwechselnd der Zeiger-Variablen *zgr* zugewiesen werden. Die Ausgabe der einzelnen Buchstaben von *otto* soll also mit *zgr* erfolgen.

6. **Ersetzen aller Vorkommen eines Wortes in einem String** (6 Punkte)
   Erstellen Sie ein Programm, das in einem String, der einzugeben ist, alle Vertreter eines Wortes, das ebenfalls einzugeben ist, durch einen anderen String (auch einzugeben) ersetzt. Dabei sollen nur ganze Wörter ersetzt werde, d.h. dass bei Wörtern, in denen das zu ersetzende Wort nur ein Teilstring ist, keine Ersetzung stattfindet.

   a Entwickeln Sie einen Algorithmus in Worten

   b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

   c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

   d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

   e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

   f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

7. **Programmausgabe** (1 Punkt)
   Das folgende Programm soll das Arbeiten mit Zeigern verdeutlichen. Bevor Sie es eingeben und ablaufen lassen sollten Sie überlegen, wie wohl die Programmausgabe aussehen wird.

```
#include <sdtio.h>

void main()
{
  int i = 9;
  int *pi;
  pi = &i;
  printf("i=%d␣&i=%u\n",i,&i);
  printf("*pi=%d␣pi=%u\n",*pi,pi);
  *pi = 3;
  printf("i=%d␣&i=%u\n",i,&i);
  printf("*pi=%d␣pi=%u\n",*pi,pi);
  printf("&pi=%u\n",&pi);
}
```

8. **Call-by-reference** (3 Punkte)
   Schreiben Sie eine C-Funktion *teile(i,j,k,l)*, die die übergebene Zahl i durch j dividiert und in k das Ergebnis sowie in l den Divisionsrest zurück liefert (Hier müssen Sie also mit Zeigern arbeiten!!!). Alle Zahlen sollen ganze Zahlen sein. Die Funktion *teile* soll eine Division durch 0 abfangen und in diesem Fall den Rückgabewert 0 liefern, sonst den Rückgabewert 1.

9. **Stringumkehr** (3 Punkte)
   Implementieren Sie die Funktion **void strreverse(char *out, char *in)** die in den Speicherbereich ab *out* den String *in* rückwärts schreibt.

B.3. Übungen zu Dateien

10. **Statistik über Wortlängen in einer Datei** (5 Punkte)
Erstellen Sie ein Programm, das eine Statistik über die Wortlängen in einer Textdatei in tabellarischer Form ausgibt.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

    c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

    d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

    e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

    f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

11. **Kopierprogramm** (3 Punkte)
Schreiben Sie ein Programm, das eine Datei kopiert. Es benötigt dazu zwei Dateinamen + Pfad für die Quell- und Zieldatei. Diese Namen sind einzugeben. Anschließend soll die Quelldatei in die Zieldatei kopiert werden. Teilen Sie das Programm in sinnvolle Funktionen auf. Es sollen alle Arten von Dateien kopiert werden können!!

12. **Dateistatistik** (5 Punkte)
Schreiben Sie ein Programm, das in einer gegebenen Textdatei die Anzahl der Zeichen, der Worte und Zeilen ermittelt und auf dem Bildschirm ausgibt. Als Worte zählen beliebige Folgen von druckbaren Zeichen, die von mindestens einem Leerzeichen, Punkt, Komma oder Semikolon abgeschlossen sind.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

    c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

    d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

    e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

    f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

13. **Umlautersetzung** (6 Punkte)
Ersetzen Sie in einer gegebenen Textdatei die Umlaute Ä, Ü, Ö , ä, ü, ö und ß durch Ae, Ue, Oe, ae, oe, ue und sz. Der Dateiname ist vom Nutzer abzufragen. Die Zieldatei hat den gleichen Namen wie die Originaldatei, die unter dem alten Namen, aber mit der Extension „.old“ abzuspeichern ist.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

14. **Ver- und Entschlüsseln von Textdateien** (5 Punkte)
Schreiben Sie ein Programm zum Verschlüsseln bzw. Entschlüsseln von Textdateien. Die Verschlüsselung wird erreicht, indem jeder Buchstabe im Bereich a-z und A-Z um einen „geheimen“ Wert verschoben wird. Das heißt zum Beispiel für eine Verschiebung um 5 für das Wort „Atom“ eine Verschlüsselung nach „Fytr“. Die Verschiebung ist zyklisch, d.h. z um den Wert 1 verschoben ergibt a. Das Programm ermöglicht dem Benutzer die Wahl zwischen Verschlüsseln und Entschlüsseln. Der „geheime“ Wert der Verschiebung ist als Konstante fest in das Programm zu integrieren. Die Zieldatei hat den gleichen Namen wie die Originaldatei, deren Name vom Benutzer erfragt wird und danach zu löschen ist.

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

15. **Vervollständigen eines Textes mit Abkürzungen** (8 Punkte)
Erstellen Sie ein Programm, das den Inhalt einer Datei, deren Name vom Benutzer erfragt wird, auf dem Bildschirm ausgibt. Bei der Ausgabe soll es allerdings alle Abkürzungen durch den entsprechenden vollständigen Text ersetzen. Der zu den jeweiligen Abkürzungen gehörige Text steht dabei immer in der Datei ”abk.txt” (Die Abkürzung ist dabei durch einen Tabulator vom eigentlichen Text getrennt):

```
z.B. zum Beispiel
Dr. Doktor
Tsd. Tausend
EUR Euro
@S Studenten
BA Berufsakademie
```

Die Datei ”abk.txt” kann beliebig viele Abkürzungen enthalten!

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

B.4. Übung zu verketteten Listen

16. **Einfach verkettete Listen** (4 Punkte)
Schreiben Sie ein Programm, das dem Nutzer über ein Menü das Einfügen und Löschen von Listenelementen und das Anzeigen der Liste zur Auswahl gibt. Implementieren Sie diese Funktionalität mit Hilfe einer einfach verketteten Liste und Zeichenketten im Infoteil der Listenelemente. Arbeiten Sie nur mit lokalen Variablen. Als Beispiel könnte eine Adressverwaltung dienen.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

    c Definieren Sie die benötigen Strukturen und stellen Sie sie in einem Datenhierarchie-Diagramm dar

    d Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

    e Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

    f Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

    g Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

17. **kleine Datenbank** (3 Punkte)
Erweitern Sie obige Übung dahingehend, dass die Liste in einer Datei gesichert und aus dieser wieder geladen werden kann.

18. **Kontoverwaltung** (8 Punkte)
Eine Tabelle T soll insgesamt maximal n Elemente enthalten. Jedes Element besteht aus einer Kontonummer (ganzzahlig), dem Kontostand und einer Folge von maximal 10 Buchungen mit Buchungstext (max. 25 Zeichen), Buchungsbetrag und Buchungsdatum, die sich noch nicht im Kontostand niedergeschlagen haben.

    a) Schreiben Sie ein dialogfähiges und gegenüber Fehleingaben stabiles Programm, das folgende Teilaufgaben realisiert:

    b) Einrichten eines Kontos, sofern die Kapazität der Tabelle noch nicht erschöpft ist, d.h. Vergabe einer Kontonummer und Initialisierung des nächsten freien Elementes in T.

c) Aufheben eines Kontos mit Freigabe der Kontonummer und des zugehörigen Elementes in der Tabelle sowie „Auszahlen“ des Guthabens.

d) Anfertigen eines Kontoauszuges. Inhalt des Auszuges sollten die Kontonummer, der alte Kontostand, der neue Kontostand, welcher sich aus der Summe des alten Kontostandes mit der Folge von Buchungen berechnet, die Folge der Buchungen mit Betrag, Buchungstext und Buchungsdatum.

e) Eingabe und Abarbeitung von Buchungen.

f) Speichern und Laden der Tabelle T

a Entwickeln Sie einen Algorithmus in Worten für die einzelnen Aufgaben

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Definieren Sie die benötigen Strukturen und stellen Sie sie in einem Datenhierarchie-Diagramm dar

d Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

e Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

f Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

g Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

19. **Kellerspeicher** (4 Punkte)
Als Kellerspeicher (LIFO-“last in first out“) bezeichnet man einen Speicherbereich, der über die folgenden 3 Funktionen verfügt:

- push: „Einkellern“ (Abspeichern) eines Elementes
- pop: „Auskellern“ (Abrufen) des zuletzt eingekellerten Elementes
- count: Ermitteln der Elementanzahl im Kellerspeicher

Es ist ein Programm zu schreiben, das einen Kellerspeicher für Elemente vom Typ *long* bereitstellt. Treffen Sie hierzu geeignete Knotenvereinbarungen und implementieren Sie die Funktionen push, pop und count unter Nutzung:

a) einer statischen Datenstruktur (Feld) mit einer maximalen Anzahl von 5 Kellerelementen, wobei die push-Funktion diese Grenze natürlich zu beachten hat;

b) einer dynamischen Datenstruktur (einfach verkettete Liste), die nur für die jeweils enthaltenen Kellerelemente Speicher reserviert.

Das Programm soll über ein kleines Menü bedienbar sein. Nach jedem push bzw. pop soll der Inhalt des Kellerspeichers ausgegeben werden.

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Definieren Sie die benötigen Strukturen und stellen Sie sie in einem Datenhierarchie-Diagramm dar

d Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

e Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

f Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

g Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

20. **Wortstatistik zu einem Text** (5 Punkte)
Erstellen sie ein Programm, das zu einem Text eine alphabetisch geordnete Wortstatistik erstellt, was bedeutet, dass zu jedem Wort die Häufigkeit seines Vorkommens im vorgelegten Text ausgegeben wird. Die Ausgabe soll in tabellarischer Form erfolgen. Der Text soll entweder vom Benutzer eingegeben werden oder stammt praktischer weise aus einer Textdatei.

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Definieren Sie die benötigen Strukturen und stellen Sie sie in einem Datenhierarchie-Diagramm dar

d Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

e Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

f Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

g Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

B.5. Übungen zur Rekursion

21. **Potenzieren** (3 Punkte)
Schreiben Sie ein Programm, das um die Eingabe einer Zahl und einer Potenz bittet. Schreiben Sie ein *rekursive* Funktion, um die Zahl zu potenzieren. Lautet z.B. die Zahl 2 und die Potenz 4 so sollte das Ergebnis 16 sein.

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

22. **Damenproblem** (6 Punkte)
Auf einem Schachbrett sollen acht Damen so positioniert werden, dass keine Dame eine andere bedroht. Entwickeln Sie einen rekursiven Lösungsalgorithmus und ein Programm, das einfach auf andere Brettgrößen geändert werden kann (per Präprozessor-Konstante). Eine Lösung soll sofort auf dem Bildschirm ausgegeben werden. Entwickeln Sie den Algorithmus am besten auf einem 4*4 Brett mit 4 Damen „von Hand“.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

    c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

    d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

    e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

    f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

23. **Pascale-Dreieck** (4 Punkte)
das folgende Zahlendreieck geht auf den französischen Mathematiker Blaise Pascale zurück. An seinen Kanten befindet sich die Zahl 1. Innerhalb des Dreiecks wird jede Zahl als Summe der zwei darüber liegenden Zahlen (links + rechts darüber) errechnet. Entwickeln Sie eine rekursive Funktion **int pascale(int zeile, int spalte);** die den Wert eine entsprechenden Elementes des Pascalschen Dreiecks berechnet. *zeile* und *spalte* sind vom Benutzer einzugeben.

					1							
				1		1						
			1		2		1					
		1		3		3		1				
	1		4		6		4		1			
1		5		10		10		5		1		
	6		15		20		15		6		1	
7		21		35		35		21		7		1

Die Lösung für pascale(6,3) wäre z.B. 10.

    a Entwickeln Sie einen Algorithmus in Worten

    b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

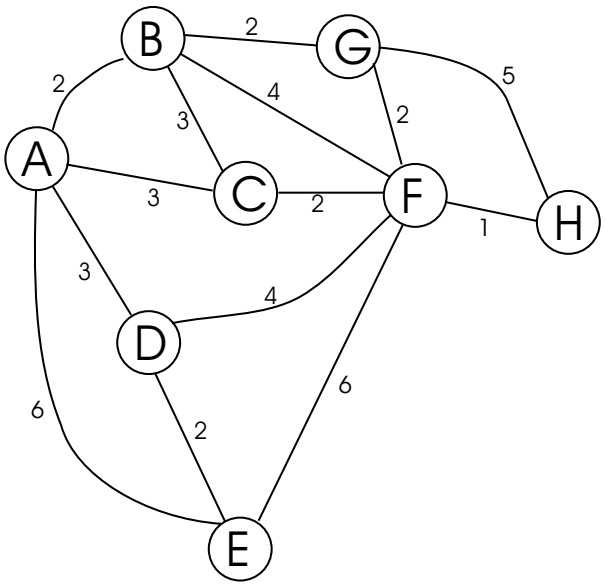
    c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

    d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

    e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

    f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

24. **Routenplaner** (5 Punkte)
Entwickeln Sie einen Routenplaner für folgendes Straßennetz:



Wie jeder gute Routenplaner soll dieser die kürzestes Strecke zwischen zwei Städten ausgeben. Der Start- und Zielpunkt sind vom Benutzer zu wählen. Die Ausgabe soll alle Zwischenstationen und die gesamte Entfernung beinhalten.

a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

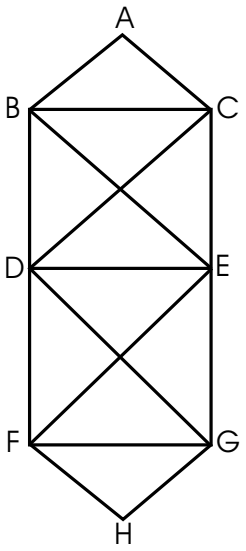
d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

25. **Haus des Nikolaus** (4 Punkte)
Das Haus des Nikolaus ist weit bekannt. Die Regeln für den „Bau“ des Hauses sind recht einfach:
Verbinde alle Punkte ohne Abzusetzen und ohne eine Strecke zweimal zu fahren.

Nun steht das Haus an einem See und wird dadurch noch gespiegelt. Nach obiger Regel sollen alle Möglichkeiten ermittelt werden, dieses gespiegelte Haus zu zeichnen. Das zu erstellende Programm soll alle Möglichkeiten in eine Textdatei und auf dem Bildschirm ausgeben.



a Entwickeln Sie einen Algorithmus in Worten

b Definieren Sie die Ein-/Ausgabedaten des Algorithmus

c Zerlegen Sie den Algorithmus in sinnvolle Aufgaben/Funktionen

d Verfeinern Sie den Algorithmus und stellen Sie ihn als Programmnetz mit allen Ein-/Ausgabedaten dar

e Zeichnen Sie ein Datenflussdiagramm der benötigten Daten

f Programmieren Sie den *fertigen* Algorithmus in einem C-Programm

B.6. Übungen zu Sortieren

26. **Zahlenfeld** (1 Punkt)
Schreiben Sie ein Programm welches ein vorgegebenes Feld mit 10 Fliesskommazahlen aufsteigend sortiert.

27. **Zahlenfeld 2** (2 Punkte)
Schreiben Sie ein Programm welches ein vorgegebenes Feld mit 10 Fliesskommazahlen *absteigend* sortiert.

28. **Studentenverwaltung** (5 Punkte)
Erstellen Sie ein Programm, das vom Benutzer 10 Datensätzen von Studenten (Name, Vorname, Matrikelnummer) einliest. Danach sollen die Datensätze je nach Benutzerwunsch nach Name, Vorname oder Matrikelnummer sortiert ausgegeben werden.

Literaturverzeichnis

[1] Winfried Bantel. *Strukturiertes Programmieren in C.* Shaker Verlag, Aachen, 2001.

[2] Ulrike Böttcher. *Grundlagen der Programmierung.* HERDT-Verlag, Nackenheim, 1. edition, 2001.

[3] Helmut Erlenkötter. *C Programmieren von Anfang an.* rororo Computer, 4. edition, 2001.

[4] Dr. Wilhelm Hanrath. *Einführung in die Programmiersprache C.*

[5] Helmut Herold. *Die Programmiersprache C.*

[6] Rolf Isernhagen. *Softwaretechnik in C und C++.* Hanser Verlag, München, 2. edition, 2000.

[7] Klaus Schmaranz. *Softwareentwicklung in C.* Springer-Verlag, 2001.