Inhaltsverzeichnis

2.1 Benutzerdefinierte Aufzählungstypen 3
Anwendung benutzerdefinierter Aufzählungstypen 3
Definition benutzerdefinierter Aufzählungstypen 4
2.2 Benutzerdefinierte Datentypen 7
Anwendung benutzerdefinierter Datentypen 7
Definition benutzerdefinierter Datentypen 7
2.3 Verbunde und Objekte: Strukturen und Klassen 8
Strukturen 8
Funktionen zum Zugriff auf Strukturelemente 11
Einheit von Struktur und Zugriffsfunktionen: Objektklasse 13
Kapselung 15
Objektklasse und Objektexemplar 17
Konstruktoren für Klassen: Standardkonstruktor 17
Konstruktoren für Klassen: Überladener Konstruktor 18
Konstruktoren für Klassen: Kopierkonstruktor 20
Destruktoren für Klassen 20
2.4 Varianten (unions) 21
2.5 Verteilte Programmentwicklung, Wiederverwendbarkeit 22
2.6 Arrays (Felder) 23
Der Begriff „Array“ 23
Eindimensionale Arrays 25
Mehrdimensionale Arrays 27
Arrays als Parameter in Funktionen 28
Vektoren 31
2.7 Zeichenketten (Strings) 35
C-Strings (Zeichenarrays) 35
Die C++-Klasse „string“ 36
String-Methoden 37
2.8 Dateien (Streams) 42
Dateitypen 42
Datei-Zugriffsarten 43
Grundstruktur der Dateiverarbeitung 44
Textdateien 45
Binärdateien 49
Methoden zur Dateiverarbeitung 57
Speichern von Datensätzen mit dynamischen Komponenten 59
2.9 Rekursive Algorithmen 60
Unterscheidung der Begriffe Iteration und Repetition 60
Eigenschaften rekursiver Algorithmen 62
2.10 Suchen und Sortieren in Feldern und Dateien 65
Binäres Suchen 66
Sortieren durch direkte Auswahl (Selection Sort) 68
Sortieren durch direktes Einfügen (Insertion Sort) 69
Sortieren durch direkten Tausch (Bubble Sort) 70
Effizienz der einfachen Sortierverfahren 71
Shell Sort 73
Quick Sort 75
Suchen und Sortieren in Dateien mittels Indexdateien 78
2.11 Zeiger 82
Speicheradressen und Zeiger 82
Nullzeiger und typfreie Zeiger 85
Konstante Werte, konstante Zeiger 86
Zeiger und Arrays 87
Zeigerarithmetik 89
Char-Arrays und C-Strings 90
C-Arrays als formatfreie Byteströme 93
Funktionen zur Verarbeitung von C-Strings 94
2.12 Zeiger und Funktionen 98

Arrays als Funktionsparameter *98*
Parameterübergabe mit Zeigern *100*
Parameter der Funktion main() *102*
Zeiger auf Strukturelemente, Funktionen und Elementfunktionen *103*
2.13 DYNAMISCHE DATENTYPEN 105
Einfache dynamische Typen *105*
Dynamisch erzeugte Arrays *106*
Dynamisch erzeugte Strukturen *107*
Freigeben dynamischer Objekte *107*
2.14 LISTEN UND BÄUME 109
Stapel (LIFO) *109*
Schlange (FIFO) *114*
Einfach verkettete Liste *119*
Aufbau von binären Bäumen *125*
Bearbeiten von Daten in binären Bäumen *131*
Suchen in binären Bäumen *132*

2 Komplexe Datentypen in C++

Jede Programmiersprache bietet fundamentale Datentypen an, die für einfache Anwendungen ausreichen. In C++ sind dies die bereits bekannten Datentypen bool, char, int, long, float und double. Meist muß der Programmierer sich aber für spezielle Anwendungen Datentypen selbst konstruieren.

2.1 Benutzerdefinierte Aufzählungstypen

Lerninhalte

❶ Anwendung benutzerdefinierter Aufzählungstypen
❷ Definition benutzerdefinierter Aufzählungstypen

Lerninhalte

L1 Anwendung benutzerdefinierter Aufzählungstypen

Häufig existieren in realen Programmen nichtnumerische Werte, die sich durch die Standarddatentypen nur schlecht abbilden lassen. So kann zum Beispiel ein Wochentag nur die sieben Werte *Son*, *Mon*, *Die*, *Mit*, *Don*, *Fre* und *Sam* annehmen. Oder ein Programm, das eine Ampelanlage realisiert, kennt nur die drei Farbwerte *Rot*, *Gelb* und *Gruen*. Man kann diese Werte natürlich über den Datentyp *int* realisieren:

```
int Ampel         // rot=0, gelb=1, gruen=2
int Wochentag;    // Sonntag=0, Montag=1, usw.
```

Das hätte jedoch einige Nachteile:

- Die „Zahlenbedeutung“ müsste als Programmkommentar vermerkt werden.
- Die Zuordnung *Zahl - Bezeichner* ist nicht eindeutig: 0 kann *rot*, aber auch *Sonntag* bedeuten.
- Das Programm dokumentiert sich schlecht selbst:

```
if (ampel==2) ... // Welche Farbe ist das denn?
If (ampel==5) ... // Der Farbwert 5 ist nicht definiert.
```

Hier ist es sinnvoll, wenn der Programmierer beliebig eigene Typen vereinbaren kann. Man spricht dann von *selbstdefinierten* oder *benutzerdefinierten* Datentypen. Die einfachste Art, einen selbstdefinierten Typ zu vereinbaren, ist die *Umdeklaration* eines bereits vordefinierten Typs:

```
typedef signed char byte;  // gibt signed char den Namen byte
typedef int ganzzahl;      // statt int kann man nun ganzzahl schreiben
```

Definition benutzerdefinierter Aufzählungstypen

Die Lösung für die oben beschriebenen Anforderungen bieten die *Aufzählungs- oder Enumerationstypen.* Diese werden folgendermaßen definiert bzw. deklariert:

```
enum Typname {Aufzählung} Variablenliste;
```

Es ist also möglich, den Datentyp und die Variablen in einer Anweisung zu codieren. Übersichtlicher ist es hingegen, zunächst nur den Datentyp zu definieren und dann in einer weiteren Anweisung die Variablenzu deklarieren.

```
enum Typname {Aufzählung};
Typname Variablenliste;
```

An dieser Stelle sei noch angemerkt, dass das alte C bei der Variablendeklaration ebenfalls ein Schlüsselwort enum verlangt:

```
enum Typname {Aufzählung};
enum Typname Variablenliste;
```

Die Datentypen *Ampel* und *Wochentag* kann man damit so definieren:

```
enum Ampel { rot, gelb, gruen };
enum Wochentag { Mon, Die, Mit, Don, Fre, Sam, Son };
```

Wenn der Datentyp definiert ist, können Variablen deklariert werden:

```
Ampel Stadtmitte, Ortseingang;   // Definitionen
Wochentag feiertag, werktag;     // Definitionen
Wochentag heute = Die;           // Definition und Initialisierung
```

Falls man eine Variable mit selbstdefiniertem Typ nur ein einziges Mal benötigt, kann man eine *anonyme Typdefinition* vornehmen, indem man den Typbezeichner wegläßt:

```
enum {fahrrad, mofa, lkw, pkw} fahrzeug;
```

Aufzählungstypen sind eigenständige Datentypen, die jedoch programmintern auf natürliche Zahlen abgebildet werden, beginnend bei 0. Der dafür benötigte Speicherplatz richtet sich nach dem größten erforderlichen Zahlenwert. Wenn notwendig, kann man die Zahlenabbildung ändern:

```
enum Ampel {rot=1, gelb=2, gruen=3};    // Speicherplatz: 2 Bit
                                        // (Bereich 0..3)
enum fall { A=3, B=6, C=9 };            // Speicherplatz: 4 Bit
                                        // (Bereich 0..15)
enum GroßZahl {min = -10, max=1000000}; // Speicherplatz: 21 Bit

fall afall = 5;       // Typfehler: 5 ist nicht vom Typ fall
fall bfall = fall(5); // Explizite Konvertierung, deshalb ok!
fall cfall = fall(15); // auch ok, da noch im Bereich von fall
fall dfall = fall(20); // undefiniert: 20 ist nicht im Bereich von fall

int i = Die; // ok, implizite Konvertierung nach int
heute = Mon; // ok
Mon = heute; // Fehler: Montag ist Konstante
i = rot + gruen; // ok
feiertag = Mon + Die; // Fehler: Typkonvert. von int-Ergebnis nicht möglich
afall++;             // Fehler: aus gleichem Grund;
if (afall > A) afall = C; // ok
```

Das folgende Programm Gedicht.cpp gibt den Anfang eines Gedichtes vor, die darauffolgenden Gedichtzeilen können vom Benutzer ausgewählt werden.

```
// Demonstration von selbstdefinierten Datentypen
#include <iostream>

enum  jzeit {Garnichts, Fruehling, Sommer, Herbst, Winter};

void menue()
{
  cout << "        Wir basteln ein Gedicht!" << endl;
  cout << "        ------------------------" << endl;
  cout << "Es war eine Mutter, die hatte vier Kinder," << endl;
  cout << "den Fruehling, den Sommer, den Herbst und den Winter." << endl << endl;
  cout << "Mit welcher Jahreszeit soll''s weitergehen?" << endl << endl;
  cout << "    1  Fruehling" << endl;
  cout << "    2  Sommer" << endl;
  cout << "    3  Herbst" << endl;
  cout << "    4  Winter" << endl;
}

void eingabe(jzeit& jahreszeit)
{
  int Wahlzahl;
  cout << "Waehlen Sie (0 zum Beenden): "; cin >> Wahlzahl;
  cout << endl;
  switch(Wahlzahl)
  {
      case 1:  jahreszeit=Fruehling; break;
      case 2:  jahreszeit=Sommer; break;
      case 3:  jahreszeit=Herbst; break;
      case 4:  jahreszeit=Winter; break;
      default: jahreszeit=Garnichts;
  }
}

void verarbeitung(jzeit jahreszeit)
{
  if (jahreszeit > Garnichts)
  {
      switch (jahreszeit)
      {
      case Fruehling : cout << "Der Fruehling bringt Blumen." << endl; break;
      case Sommer    : cout << "Im Sommer blueht Klee." << endl; break;
      case Herbst    : cout << "Im Herbst reifen Aepfel." << endl; break;
      case Winter    : cout << "Im Winter faellt Schnee." << endl; break;
      }
  }
}

void main()
{
  jzeit jahreszeit;
  do
  {
      menue();
      cout << endl;
      eingabe(jahreszeit);
      verarbeitung(jahreszeit);
      cout << endl << endl;
  }
  while (jahreszeit != Garnichts);
}
```

Grundlegende Aufgabe wochentag.cpp

(wochentag.cpp) Schreiben Sie ein Programm, das einen Aufzählungstyp mit Wochentagen enthält. Dieses soll über zwei Funktionen einen Wochentag einlesen bzw. im Klartext ausgeben.

Grundlegende Aufgabe ampel.cpp

(ampel.cpp) Schreiben Sie ein Programm, das einen Aufzählungstyp mit allen(!) Ampelphasen enthält. Lassen Sie in einer verschachtelten Schleife alle Ampelphasen der Reihe nach 3-mal ausgeben.

Grundlegende Aufgabe Essen1.cpp

Schreiben Sie ein Programm Essen1.cpp, das die folgenden Deklarationen benutzt:

```
enum Tlebmitt {Bananen, Birnen, Zitronen, Blumenkohl, Spinat, Tomaten,
               Salami, Schnitzel, Steak} lebmitt;
bool Pflanzliches(Tlebmitt l);
bool Tierisches(Tlebmitt l);
bool Obst(Tlebmitt l);
bool Gemuese(Tlebmitt l);
```

```
Wir pluendern einen Fresskorb!
        ---------------------------

Was soll herausgeholt werden ?

    1  Alles
    2  Nur Vegetarisches
    3  Fleisch und Wurst
    4  Nur das Obst
    5  Nur das Gemuese


Waehlen Sie (0 zum Beenden): 1
Das ist drin:
Bananen Birnen Zitronen Blumenkohl Spinat Tomaten Salami Schnitzel Steak

Waehlen Sie (0 zum Beenden): 3
Das ist drin:
Salami Schnitzel Steak

Waehlen Sie (0 zum Beenden):
```

Grundlegende Aufgabe Essen2.cpp

Variieren Sie Essen1.cpp zu Essen2.cpp. Das Programm soll die gleiche Funktion haben, aber nur noch eine einzige *bool*-Funktion benötigen:

```
bool LebmittArt(TLebmitt l, TLebmitt la, TLebmitt le);
// la : Erste Konstante der Art, le: letzte Konstane der Art in der Typreihe
```

2.2 Benutzerdefinierte Datentypen

Lerninhalte

❶ Anwendung benutzerdefinierter Datentypen
❷ Definition benutzerdefinierter Datentypen

Lerninhalte

L1 Anwendung benutzerdefinierter Datentypen

Häufig überlegt man sich in Programmen einmal einen speziellen Datentyp, den man für spezielle Zwecke nutzt, z.B. `unsigned long` für eine Kundennummer oder `unsigned int` für die Seitenzahl eines Buches. Wenn man nun mehrfach im Programm Variablen diesen Datentyps deklariert, stößt man auf zwei Probleme:

1. Man muß jedesmal überlegen, wie denn nun die genaue Deklaration lauten muß (z.B. signed oder unsigned, short, int oder long).
2. Trifft man später wieder auf die Deklaration, ist nicht genau klar, welchem Zweck der Datentyp dient, da z.B. unsigned long eine Kundennummer oder auch ein Lagerbestand sein könnte. Hier muß man dann mit Kommentaren arbeiten, das das Programm nicht selbst dokumentierend ist.

```
unsigned long k1,k2; // Kundennummern
unsigned long b1,b2,b3; // Bestandszahlen
```

Es wäre also nützlich, einen Namen für einen bestimmten Grunddatentyp angeben zu können. Dies erreicht man durch die Verwendung benutzerdefinierter Datentypen.

L2 Definition benutzerdefinierter Datentypen

Diese werden folgendermaßen definiert:

```
typedef bekannter_Datentyp neuer_Name;
```

Der bekannte Datentyp kann die Zusammmensetzung eines einfachen Datentyps sein (`unsigned long`), aber auch die später behandelten komplexen Datentypen können mit `typedef` einen neuen Namen erhalten.

```
typedef unsigned long TKundennummer;
typedef unsigned int  TSeitenzahl;
TSeitenzahl s1,s2;
TKundennummer k1,k2;
```

2.3 Verbunde und Objekte: Strukturen und Klassen

Lerninhalte

❶ Strukturen
❷ Funktionen zum Zugriff auf Strukturelemente
❸ Einheit von Struktur und Funktion: Objektklasse
❹ Kapselung
❺ Objektklasse und Objektexemplar
❻ Konstruktoren für Klassen: Standardkonstruktor
❼ Konstruktoren für Klassen: Überladener Konstruktor
❽ Konstruktoren für Klassen: Kopierkonstruktor
❾ Destruktoren für Klassen

Lerninhalte

Die Datentypen, welche bislang behandelt wurden (*int*, *float*, *enum*, usw.), nennt man auch *einfache Datentypen*. Nun werden als erste Beispiele komplexerer Datentypen Strukturen und Objektklassen behandelt.

L1 Strukturen

Sehr oft benötigt man in der Datenverarbeitung Variablen unterschiedlichen Typs, die im Rahmen eines *Verbundes* zusammengefaßt und unter *einem* Namen angesprochen werden können. (z.B. *Person* als Zusammenfassung von *Personalnummer*, *Name*, *Vorname*, *Postleitzahl*, usw.). In C++ wird ein solcher Verbund als *Struktur* bezeichnet. (In Pascal heißt eine solche Datenstruktur *Record*.) Eine Struktur wird folgendermaßen definiert:

```
struct Typname Felddefinition(en) Variable(nliste);
```

Auch hier ist es wieder möglich, aber unübersichtlich, die Typdeklaration und die Variablendefinition in einer Anweisung unterzubringen. Daher die Empfehlung, beides zu trennen.

```
struct Typname Felddefinition(en);
Typname Variable(nliste);
```

An dieser Stelle sei noch angemerkt, dass das alte C bei der Variablendeklaration ebenfalls ein Schlüsselwort `struct` verlangt:

```
struct Typname Felddefinition(en);
struct Typname Variable(nliste); // Das ist altes C !!
```

Beispiel | Deklaration einer Struktur

Ein meteorologisches Institut startet dreimal am Tag einen Wetterballon, der Temperatur, Luftdruck, relative Feuchte und Windstärke mißt sowie mittels eines Sensors anzeigt, ob Regen fällt oder nicht. Die Meßwerte sollen per Programm ausgewertet werden. Dazu wird eine Struktur *Tballon* definiert.

```
// Zunächst einige Typdefinitionen für die Strukturelemente des Wetterballons
enum Tzeit { morgens=1, mittags, abends }; // Beobachtungstermin
typedef short unsigned int Tfeuchte; // Für rel. Feuchte (0-100 %)

struct Tballon {
  int Temperatur;
  float Luftdruck;
  Tfeuchte Feuchte;
  bool Regen;
  Tzeit Tageszeit;
};
Tballon Ballon; // deklariert die Variablen Ballon mit dem Typ Tballon
```

Die einzelnen Elemente des Verbundes werden über den Punktoperator „.“ in der Form *Struktur-Variablenname.Elementname* angesprochen. Diese Art des Elementzugriffs nennt man *Qualifizierung*:

```
Ballon.Temperatur = 20;
```

Auf die Elementvariablen können alle Operationen angewandt werden, die auch mit gewöhnlichen Variablen dieses Typs möglich sind, z.B. Vergleiche mit = = oder < =, usw.

Beispiel Zugriff auf Strukturelemente

Im folgenden Programm Wetter1.cpp werden alle Elemente der Struktur mit Eingabewerten gefüllt, die über die Tastatur eingegeben werden. Danach werden alle Strukturelemente auf dem Bildschirm ausgegeben. Die Elemente *Regen* und *Tageszeit* können nicht direkt ein- oder ausgegeben werden, das ist über *cin* und *cout* sinnvoll nur mit den Typen Zahlen oder Zeichen bzw. Zeichenketten möglich. Daher wird eine boolesche Eingabefunktion für „Ja“ oder „Nein“ definiert und der Typ *Tageszeit* wird nach *int* konvertiert.

```
// Programm Wetter1.cpp
// Demonstration einer Struktur
#include <iostream>
#include <ctype.h> // wegen toupper()

typedef enum { morgens=1,mittags,abends } Tzeit; // Beobachtungs-Tageszeit
typedef short unsigned int Tfeuchte; // Für rel. Feuchte (0-100 %)

struct Tballon
{
  int Temperatur;
  float Luftdruck;
  Tfeuchte Feuchte;
  bool Regen;
  Tzeit Tageszeit;
};

Tballon Ballon;

bool JNEingabe()
{
  char antw;
  do
  {
    cin >> antw;
    antw = char(toupper(antw));
  }
  while (antw != 'J' && antw != 'N');
  return antw=='J';
}

void main()
{
  int Tageszeitzahl;

  cout << "Eingabe eines Datensatzes von Messwerten:" << endl << endl;
  cout << "Temperatur in Grad Celsius: "; cin >> Ballon.Temperatur;
  cout << "Luftdruck in bar:           "; cin >> Ballon.Luftdruck;
  cout << "Relative Feuchte in %:      "; cin >> Ballon.Feuchte;
  cout << "Regen (J/N):                "; Ballon.Regen = JNEingabe();
  cout << "Tageszeit (1-3):            "; cin >> Tageszeitzahl;
  Ballon.Tageszeit = Tzeit(Tageszeitzahl);

  cout << endl <<"Ausgabe des Datensatzes:" << endl << endl;
  cout << "Temperatur:       " << Ballon.Temperatur << " Grad Celsius"<<endl;
  cout << "Luftdruck:        " << Ballon.Luftdruck << " bar" << endl;
  cout << "Relative Feuchte: " << Ballon.Feuchte << " %" << endl;
  cout << "Regen:            ";
```

```
  if (Ballon.Regen) cout << "Ja"; else cout << "Nein"; cout << endl;
  cout << "Tageszeit:        " << int(Ballon.Tageszeit) << endl;

  cout << endl << "Programmende." << endl;
}
```

```
Eingabe eines Datensatzes von Messwerten:

Temperatur in Grad Celsius: 20
Luftdruck in bar:           1.04
Relative Feuchte in %:      40
Regen (J/N):                j
Tageszeit (1-3):            2

Ausgabe des Datensatzes:

Temperatur:       20 Grad Celsius
Luftdruck:        1.04 bar
Relative Feuchte: 40 %
Regen:            Ja
Tageszeit:        2

Programmende.
```

Die Elemente von Strukturvariablen können gleichzeitig mit der Variablendeklaration initialisiert werden:

```
struct Tballon {
      int      Temperatur;
      float    Luftdruck;
      Tfeuchte Feuchte;
      bool     Regen;
      Tzeit    Tageszeit;
};
Tballon Ballon                // Definition der Variablen
= {20,1.04,70,true,morgens};  // Initialisierung der Strukturelemente
```

Grundlegende Aufgabe `adressen1.cpp`

Entwickeln Sie eine Datenstruktur TPerson, die Nachnamen, Vornamen, Straße, Postleitzahl, Ort und Telefonnummer speichern kann. Die Elemente (außer Postleitzahl) sollen jeweils 25 Zeichen speichern können, verwenden Sie dazu (im Vorgriff auf das Kapitel über Arrays) ein char-Array, z.B.

```
typedef char TZeichenkette[25];
...
TZeichenkette name; // Name kann jetzt 25 Zeichen speichern
```

Die Postleitzahl soll mindestens den Bereich 0-99999 abdecken. Entwickeln Sie auch ein Hauptprogramm, in dem Sie mit Hilfe der Standard-Ein- und -Ausgabe-Befehle die einzelnen Strukturelemente einer Variablen vom Typ TPerson einlesen und ausgeben. (`cin` und `cout` können direkt mit char-arrays umgehen, so daß z.B. mit

```
cin >> name;
```

Zeichen in die Variable name eingelesen werden!)

L2 Funktionen zum Zugriff auf Strukturelemente

Da sich die Eingabe- und Ausgabeanweisungen im obigen Programmbeispiel direkt auf die Struktur beziehen, bietet sich eine Modularisierung mittels geeigneter Funktionen an. Dabei kann man den Funktionen die Strukturvariablen als Parameter übergeben und auf diese Weise den langen Strukturbezeichner (hier: *Ballon*) abgekürzt verwenden (hier: *b*). Zusätzlich wurde im nachstehenden Beispiel Wetter2.cpp eine aussagekräftigere Ausgaberoutine für die Anzeige der Tageszeit eingebaut.

```
// Programm Wetter2.cpp
// Demonstration einer Struktur
// mit Ein- Ausgabezugriff über Funktionen
#include <iostream>
#include <ctype.h>

typedef enum { morgens=1,mittags,abends } Tzeit;
typedef short unsigned int Tfeuchte;

struct Tballon {
  int Temperatur;
  float Luftdruck;
  Tfeuchte Feuchte;
  bool Regen;
  Tzeit Tageszeit;
};

Tballon Ballon;

bool JNEingabe() {
  char antw;
  do {
    cin >> antw;
    antw = char(toupper(antw));
  } while (antw != 'J' && antw != 'N');
  return antw=='J';
}

void Eingabe(Tballon& b) {
  int Tageszeitzahl;

  cout << "Temperatur in Grad Celsius: "; cin >> b.Temperatur;
  cout << "Luftdruck in bar:           "; cin >> b.Luftdruck;
  cout << "Relative Feuchte in %:      "; cin >> b.Feuchte;
  cout << "Regen (J/N):                "; b.Regen = JNEingabe();
  cout << "Tageszeit (1-3):            "; cin >> Tageszeitzahl;
  b.Tageszeit = Tzeit(Tageszeitzahl);
}

void AusgabeTageszeit(Tzeit z) {
  switch (z) {
    case morgens : cout << "morgens"; break;
    case mittags : cout << "mittags"; break;
    case abends  : cout << "abends"; break;
  }
}

void Ausgabe(const Tballon& b) {
  cout << "Temperatur:       " << b.Temperatur << " Grad Celsius" << endl;
  cout << "Luftdruck:        " << b.Luftdruck << " bar" << endl;
  cout << "Relative Feuchte: " << b.Feuchte << " %" << endl;
  cout << "Regen:            ";
  if (b.Regen) cout << "Ja"; else cout << "Nein"; cout << endl;
```

```
  cout << "Berichtsjzeit:    ";
  AusgabeTageszeit(b.Tageszeit);
  cout << endl;
}

void main() {
  cout << "Eingabe eines Datensatzes von Messwerten:" << endl << endl;
  Eingabe(Ballon);
  cout << endl << "Ausgabe des Datensatzes:" << endl << endl;
  Ausgabe(Ballon);
  cout << endl << "Programmende." << endl;
}
```

Beachten Sie in diesem Beispiel, dass der Parameter der Eingabefunktion ein Referenzparameter ist. Dies ist sinnvoll, da die einzelnen Strukturelemente und damit die Gesamtstruktur innerhalb der Funktion geändert werden müssen. Der Parameter der Ausgabefunktion hingegen ist ein Referenzparameter auf eine *konstante* Struktur, was zunächst sinnlos erscheint, da Referenzparameter verändert werden sollen, Konstanten hingegen nicht. Diese Kombination hat folgende Eigenschaften:

1. Dadurch, dass der Parameter als Referenz übergeben wird, muss nicht die gesamte Struktur in den zu übergebenden Parameter kopiert werden, sondern nur die Adresse des Speicherbereichs, in dem sich die Struktur befindet. Dadurch wird Rechenzeit und Speicherplatz gespart.
2. Da die übergebene Struktur nicht verändert werden darf, führen versehentliche Versuche, die Struktur bzw. ihre Elemente zu ändern zu Compilerfehlern.

Die Übergabe einer Struktur als konstanter Referenzparameter erfüllt also denselben Zweck wie ein Werteparameter und hat Vorteile. Diese Form ist demnach zu bevorzugen.

Grundlegende Aufgabe	`adressen2.cpp`

Verändern Sie adressen1.cpp, so dass es Funktionen zur Ein- und Ausgabe von Variablen der Struktur TPerson verwendet. Den Funktionen soll dabei die gesamte Struktur als Parameter übergeben werden, innerhalb der Funktion soll die Ein- bzw. Ausgabe der Strukturelemente erfolgen.

L3 Einheit von Struktur und Zugriffsfunktionen: Objektklasse

Im letzten Programmbeispiel ist offensichtlich, dass die Module *Eingabe(Tballon&)* und *Ausgabe (const Tballon&)* nur mit der Struktur Tballon *zusammen* sinnvoll eingesetzt werden können. Ebenso kann die Struktur Tballon alleine nicht sinnvoll verwendet werden, wenn keine Zugriffsfunktionen für sie existieren. Bestimmte Datentypen erfordern immer dazu passende Algorithmen und umgekehrt.

Daher bietet es sich an, die Struktur Tballon zusammen mit ihren Zugriffsfunktionen vollständig zu *kapseln.* Auch in größeren Programmen weiß man dann stets, welche Strukturen zu welchen Zugriffsfunktionen gehören. Damit wird die Struktur Tballon zur *Objektklasse* Tballon, welche sowohl *Elementdaten* enthält, als auch zugehörige Funktionsmodule. In eine Objektklasse eingekapselte Funktionen nennt man *Methoden* (auch *Elementfunktionen* genannt), deren Elementdaten nennt man auch *Eigenschaften:*

> Eine Objektklasse besteht aus *Eigenschaften* und *Methoden*
>
> *(*auch *Elementdaten* und *Elementfunktionen* genannt)

Zu diesen Begriffen seien einige Definitionen aus dem Brockhaus genannt:

Objekt: Gegenstand der Erkenntnis und Wahrnehmung, des Denkens und Handelns

Klasse: Bezeichnung für eine Teilmenge von Objekten einer Theorie, die durch gewisse Eigenschaften ausgezeichnet sind

```
/* Programm Wetter3.cpp
   Demonstration einer Struktur mit eingekapselten Methoden (Objekt) */
#include <iostream>
#include <ctype.h>

struct Tballon {
  int Temperatur;                    // Eigenschaften von Tballon
  float Luftdruck;
  short unsigned int Feuchte;
  bool Regen;
  enum Tzeit { morgens=1,mittags,abends } Tageszeit;   // Ende der Eigenschaften

  bool JNEingabe();                  // Methoden von Tballon (Deklarationen)
  void Eingabe();
  void AusgabeTageszeit();
  void Ausgabe();                    // Ende der Methoden-Deklarationen
};


bool Tballon::JNEingabe() {      // Beginn der Methoden-Definitionen von Tballon
  char antw;                     // :: ist der Zugriffsbereichsoperator!
  do {
    cin >> antw;
    antw = char(toupper(antw));
  } while (antw != 'J' && antw != 'N');
  return antw=='J';
}

void Tballon::Eingabe() {
  int Tageszeitzahl;

  cout << "Temperatur in Grad Celsius: "; cin >> Temperatur;
  cout << "Luftdruck in bar:           "; cin >> Luftdruck;
```

```
  cout << "Relative Feuchte in %:      "; cin >> Feuchte;
  cout << "Regen (J/N):                "; Regen = JNEingabe();
  cout << "Tageszeit (1-3):            "; cin >> Tageszeitzahl;
  Tageszeit = Tzeit(Tageszeitzahl);
}

void Tballon::AusgabeTageszeit() {
  switch (Tageszeit) {
    case morgens : cout << "morgens"; break;
    case mittags : cout << "mittags"; break;
    case abends  : cout << "abends"; break;
  }
}

void Tballon::Ausgabe() {
  cout << "Temperatur:       " << Temperatur << " Grad Celsius" << endl;
  cout << "Luftdruck:        " << Luftdruck << " bar" << endl;
  cout << "Relative Feuchte: " << Feuchte << " %" << endl;
  cout << "Regen:            ";
  if (Regen) cout << "Ja"; else cout << "Nein"; cout << endl;
  cout << "Berichtsjzeit:    ";
  AusgabeTageszeit();
  cout << endl;
}                                              // Ende der Methoden-Definitionen

void main() {
  Tballon Ballon;

  cout << "Eingabe eines Datensatzes von Messwerten:\n\n";
  Ballon.Eingabe(); // Ausführung einer Methode durch Qualifizierung mit .
  cout << "\nAusgabe des Datensatzes:\n\n";
  Ballon.Ausgabe();
  cout << "\nProgrammende.";
}
```

Das Konzept der *Objektorientierung* ist eine Fortentwicklung der strukturierten Programmiertechnik mit *prozeduralen* Programmiersprachen wie C oder Pascal. Wenn man Datentypen (Eigenschaften) und Elementfunktionen (Methoden) grundsätzlich zusammenfaßt, ist die resultierende *Objektklasse* viel leichter wartbar, speziell dann, wenn man - wie bei professionellen Programmierprojekten üblich - mit vielen Programmierern an einem einzigen Projekt arbeitet.

Grundlegende Aufgabe	`adressen3.cpp`

Verändern Sie adressen2.cpp, so dass die Funktionen zur Ein- und Ausgabe zu Elementfunktionen werden.

L4 Kapselung

An der obigen Programmlösung ist noch unschön, dass die Datenelemente des Objekts *Ballon* durch Algorithmen außerhalb des Objekts geändert werden können. Variieren Sie z.B. Wetter3.cpp wie folgt:

```
void main() {
  Tballon Ballon;

  cout << "Eingabe eines Datensatzes von Messwerten:" << endl << endl;
  Ballon.Eingabe();
  Ballon.Temperatur = -50; // Absichtlich verändern!!
  cout << endl << "Ausgabe des Datensatzes:" << endl << endl;
  Ballon.Ausgabe();
  cout << endl <<"Programmende." << endl;
}
```

Nun wird das Programm immer eine Temperatur von -50°C anzeigen, unabhängig davon, welchen Wert Sie mit der Objektmethode *Tballon.Eingabe()* eingegeben haben.

Eine solche Beeinflussung von Objektdaten durch äußere Algorithmen ist in der Regel unerwünscht. Daher wird *vollständige Kapselung* angestrebt. Darunter versteht man, dass die Eigenschaften eines Objektes nur über die zugeordneten Objektmethoden geändert werden können.

Betrachtet man sich die Struktur *Tballon* genau, dann erscheint sinnvoll, dass alle Strukturelemente *Temperatur* bis *Tageszeit* sowie die Elementfunktionen *Tballon::JNEingabe()* und *Tballon::AusgabeTageszeit()* nur *innerhalb* des Objektes bekannt sind, denn außerhalb werden sie nicht benötigt. Wenn sie auch außerhalb des Objektes gültig sind, können sie auch von Programmierern verwendet werden, die das Objekt *Tballon* nicht selbst erstellt haben. Wenn nun der „Erfinder“ von *Tballon* solche „öffentlich zugänglichen“ Objektmethoden irgendwann einmal ändert, hat das mit Sicherheit ungünstige Auswirkungen auf fremde Programmteile. Lediglich die Elementfunktionen *Tballon:Eingabe* und *Tballon::Ausgabe* sollen öffentlich benutzbar sein.

Man erreicht das gewünschte Verhalten, indem man die Schlüsselwörter *class* (statt *struct)* und zusätzlich die neuen Schlüsselworte *public* und *private* verwendet.

```
/* Programm Wetter4.cpp
   Demonstration eines vollständig gekapselten Objekts */
/* ... include-Direktiven wie bei Wetter3.cpp */

class Tballon {     // Deklaration einer Objektklasse Tballon
private:
  int Temperatur;   // Alle Felder einer class sind standardmäßig private
  float Luftdruck;
  short unsigned int Feuchte;
  bool Regen;
  enum { morgens=1,mittags,abends } Tageszeit;
  bool JNEingabe();          // Zwei private-Methoden
  void AusgabeTageszeit();
public:
  void Eingabe();          // Zwei öffentliche Zugriffsmethoden
  void Ausgabe();
};
/* ... Hier folgen die Methodendefinitionen wie bei Wetter3.cpp ... */

void main() {
  Tballon Ballon;

  cout << "Eingabe eines Datensatzes von Messwerten:" << endl << endl;

  Ballon.Temperatur = -50; // ergibt Compiler-Fehler:
                           // „Tballon::Temperatur is not accessible“
  Ballon.Eingabe();
  cout << endl << "Ausgabe des Datensatzes:" << endl << endl;
  Ballon.Ausgabe();
  cout << endl << "Programmende." << endl;
}
```

Nun kann man vom Hauptprogramm aus *Ballon.Temperatur* nicht mehr verändern, denn diese Variable ist außerhalb der Objektklasse Tballon unbekannt, genauso wie alle übrigen Objekteigenschaften. Auch die privaten Elementfunktionen *JNEingabe()* und *AusgabeTageszeit()* kann man von außerhalb des Objektes nicht mehr ausführen.

Grundlegende Aufgabe	wetter4.cpp

Probieren Sie die Eigenschaften der Kapselung aus, indem Sie das Programm *Wetter4.cpp* entsprechend variieren und versuchen, private Elementdaten und Elementfunktionen zu verwenden.

Grundlegende Aufgabe	adressen4.cpp

Verändern Sie adressen3.cpp, so dass es eine Klasse statt einer Struktur verwendet und probieren sie auch hier die Eigenschaften der Kapselung aus.

L5 Objektklasse und Objektexemplar

Merken Sie sich auch die folgende Ausdrucksweisen: Wenn man den *Typ* eines Objektes deklariert (mit *class typ { ... }*, dann spricht man von einer *Objektklasse.* Sobald man eine Variable mit dem Typ dieser Klasse deklariert (mit *Tballon Ballon)*, redet man von einem *Objektexemplar* (manchmal auch *Objektinstanz* genannt).

Man verwendet das Schlüsselwort *class*, wenn man Objektklassen definiert. Bei einer *class* sind standardmäßig alle Elemente *private*, das heißt, sie sind außerhalb der Klasse ungültig und werden erst durch das Schlüsselwort *public* öffentlich. Bei einer *struct* ist es genau umgekehrt: Alle Elemente sind standardmäßig *public*, d.h. sie sind überall im Programm bekannt und können global verwendet werden, es sei denn sie werden durch *private* geschützt.

Die folgenden Definitionen von *class* und *struct* haben die gleiche Bedeutung:

```
class Tetwas {

  /* Eigenschaften, Methoden */

public:

  /* Eigenschaften, Methoden */
}
```

```
struct Tetwas {
private:

  /* Eigenschaften, Methoden */

public:

  /* Eigenschaften, Methoden */
}
```

L6 Konstruktoren für Klassen: Standardkonstruktor

Konstruktoren werden zur Initialisierung von Objekten verwendet. Wenn kein Konstruktor angegeben ist, wird das Objekt automatisch initialisiert, ansonsten kann der Programmierer großen Einfluß darauf nehmen, wie die Daten eines frisch erzeugten Objektes aussieht.

Die allgemeine Syntax für Konstruktoren lautet:

```
class className
{
public:
  //Deklarationen
   className(); //Standardkonstruktor
   className(<Parameterliste>); //anderer Konstruktor
   className(const className& c);  //Kopierkonstruktor
};
//Definitionen
className::className():Daten1(0),Daten2(0){};
className::className(<Parameterliste>):Daten1(Parameter1),Daten2(Parameter2){};
```

Der Standardkonstruktor ist also eine Methode mit gleichem Namen wie die Klasse. Er erwartet keine Parameter. Die zu initialisierenden Daten der Klasse werden in einer Liste hinter einem Doppelpunkt, vor der geschweiften Klammer des Funktionsblocks aufgezählt.

Ein Beispiel für den Standard-Konstruktor liefert folgendes Programm:

```
#include <iostream>
using namespace std;

class Zaehler
{
private:
   unsigned int count;
public:
   Zaehler();         // Standardkonstruktor
   void inc_count();
   int get_count();
};
Zaehler::Zaehler():count(1) {} // Standardkonstruktor
void Zaehler::inc_count()   { count++; }
int  Zaehler::get_count()   { return count; }

void main()
{
   Zaehler c1,c2;

   cout << "c1=" << c1.get_count() << endl;
   cout << "c2=" << c2.get_count() << endl;
   c1.inc_count();
   c2.inc_count();
   c2.inc_count();
   cout << "c1=" << c1.get_count() << endl;
   cout << "c2=" << c2.get_count() << endl;
}
```

Hier ist der Konstruktor *Zaehler()* zum einen dafür verantwortlich, dass die beiden Objekte *c1* und *c2* initialisiert werden, zum anderen werden diese (genauer: das Datum count der Objekte) auch gleich auf Eins gesetzt.

Einige Regeln für Konstruktoren:

1. Sie haben denselben Namen wie die Klasse, in der sie als Methode definiert sind.
2. Es gibt für Konstruktoren keine Rückgabewerte (auch nicht *void*), da sie automatisch vom System aufgerufen werden.
3. Eine Klasse kann eine beliebige Anzahl von Konstruktoren (also auch keinen) haben.
4. Standardkonstruktor ist derjenige Konstruktor, der entweder über keine Parameter verfügt oder in dessen Parameterliste für alle Parameter Vorgabeargumente verwendet werden.

Ein Beispiel für 4):

```
class punkt
{
private:
   double x,y;
public:
   punkt(double xwert=0, double ywert=0); // Standardkonstruktor
};
punkt::punkt(double xwert, double ywert) // Standardkonstruktor
                    :x(xwert),y(ywert){}
```

Konstruktoren für Klassen: Überladener Konstruktor

Wir betrachten noch einmal das Zähler-Beispiel. Es wäre nützlich, dem Zähler bereits vor der ersten Verwendung einen beliebigen Wert zuordnen zu können. Dies geschieht mit einem (überladenen) Konstruktor wie im folgenden Programm.

```
#include <iostream>
using namespace std;

class Zaehler
{
private:
   unsigned int count;
public:
   Zaehler();         // Standardkonstruktor
   Zaehler(int c);    // überladener Konstruktor
   void inc_count();
   int get_count();
};
Zaehler::Zaehler():count(1)      {} // Standardkonstruktor
Zaehler::Zaehler(int c):count(c) {} // überladener Konstruktor
void Zaehler::inc_count()   { count++; }
int  Zaehler::get_count()   { return count; }

void main()
{
   Zaehler c1,c2(17);

   cout << "c1=" << c1.get_count() << endl;
   cout << "c2=" << c2.get_count() << endl;
   c1.inc_count();
   c2.inc_count();
   c2.inc_count();
   cout << "c1=" << c1.get_count() << endl;
   cout << "c2=" << c2.get_count() << endl;
}
```

In diesem Programm werden zwei Konstruktoren verwendet. Der Zähler c1 wird mit Hilfe des Standard-Konstruktors, der Zähler c2 wird mit Hilfe des überladenen Konstruktors initialisiert; damit wird count des Zählers auf 17 initialisiert.

Da es jetzt mehr als einen Konstruktor in einer Klasse gibt, nennt man dies auch Überladung von Konstruktoren. Man kann beliebig viele überladenene Konstruktoren in einer Klasse deklarieren. Wenn man einen überladenen Konstruktor deklariert, muß man den ansonsten vom System bereitgestellten Standardkonstruktor ebenfalls deklarieren.

Wenn man mehrere Konstruktoren in einer Klasse deklariert hat, entscheidet das Programm zur Laufzeit anhand der übergebenen Argumente, welcher Konstruktor verwendet wird.

In unserem Ballonbeispiel könnten die Konstruktoren wie folgt aufgebaut werden:

```
class Tballon {
//...
public:
  // Standardkonstruktor:
  Tballon():Temperatur(20),Luftdruck(1),Feuchte(50),Regen(false),
            Tageszeit(mittags){};
  // überladener Konstruktor:
  Tballon(int t, float l, short unsigned int f, bool r, TTageszeit tz):
            Temperatur(t),Luftdruck(l),Feuchte(f),Regen(r),Tageszeit(tz){};
//...
};
```

oder

```
class Tballon {
//...
public:
  // ein Konstruktor für beides:
  Tballon(int t=20, float l=1, short unsigned int f=50, bool r=false,
            TTageszeit tz=mittags):
            Temperatur(t),Luftdruck(l),Feuchte(f),Regen(r),Tageszeit(tz){};
//...
};
```

L8 Konstruktoren für Klassen: Kopierkonstruktor

Man kann ein Objekt definieren und ihm zugleich die Werte eines anderen Objekts zuweisen:

```
alpha a3(a2);
alpha a3 = a2;
```

Beide Schreibweisen rufen den Kopierkonstruktor auf, d.h. einen Konstruktor, der seine Argumente in das neue Objekt kopiert. Der Standard-Kopierkonstruktor, den das System bereitstellt, kopiert alle Datenelemente in das neue Objekt.

Kopierkonstruktoren werden erst sinnvoll verwendbar, wenn Klassen dynamische Datenstrukturen enthalten. Dazu sind jedoch Vorkenntnisse über Zeiger auf Objekte und dynamische Variablen notwendig. Bis dahin genügt der Standard-Kopierkonstruktor.

L9 Destruktoren für Klassen

Ähnlich wie Konstruktoren werden Destruktoren automatisch aufgerufen, wenn ein Objekt verschwinden soll. Ein Destruktor hat denselben Namen wie der Konstruktor (wie auch die Klasse), aber mit vorangestelltem Tilde-Zeichen.

Beispiel:

```
class Abc
{
  private:
      int daten;
  public:
      Abc():daten(0) {}
      ~Abc() {}
}
```

Wie die Konstruktoren haben auch die Destruktoren keinen Rückgabewert. Außerdem ist ihre Parameterliste leer. Das Haupteinsatzgebiet von Destruktoren ist die Freigabe von Speicher, der für Objekte durch den Konstruktor beim Anlegen des Objektes oder andere Methoden dynamisch während der Laufzeit reserviert wurde. Dazu sind jedoch Vorkenntnisse über Zeiger auf Objekte und dynamische Variablen notwendig.

Grundlegende Aufgabe	adressen4a.cpp

Ergänzen Sie die Klasse der Adressverwaltung um Konstruktoren. Prüfen Sie, ob Sie dann bestimmte Initialisierungen weglassen können.

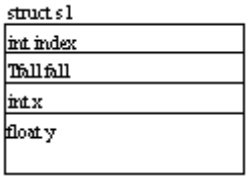
2.4 Varianten (unions)

Eine *Variante* (engl. *union*) ist eine Struktur, in der alle Elemente ab der gleichen Speicherstelle niedergelegt werden. Dadurch belegt die Variante nur so viel Speicherplatz, wie ihr größtes Element benötigt. Eine Variante wird bei Strukturen oder Klassen verwendet, wenn man Speicherplatz einsparen will. Natürlich wird auf diese Art und Weise immer nur ein einziges Element der Struktur im gewünschten Datenformat gespeichert.

Beispiel	Deklaration einer Variante

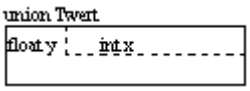
Eine Struktur benötigt entweder ein *int*- oder ein *float*-Element.

```
enum Tfall {Ganzzahl, Reelle_Zahl};
struct s1 {
  int index; // wird immer benötigt
   Tfall fall;
   int x;      // x wird verwendet, wenn fall==Ganzzahl
   float y;    // y wird verwendet, wenn fall==Reelle_Zahl
};
```



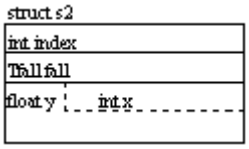
Da die Elemente x und y niemals gleichzeitig benötigt werden, wird unnötig Platz verschwendet. Daher kann man x und y zu Elementen einer Variante machen:

```
union Twert {
  int    x;
  float  y;
};
```



Diese Variante kann man nun in die Struktur einfügen und so Speicherplatz sparen:

```
struct s2 {
  int    index;
  Tfall  fall;
  Twert  wert;
};
```



Dabei ist zu beachten, dass x und y sich gegenseitig beeinflussen. Wird x verändert, verändert sich damit auch y, und zwar in einen meist sinnlosen Wert. Wird y verändert, beeinflusst dies auch x, welches dann einen sinnlosen Wert hat.

Das folgenden Programmbeispiel demonstriert die Anwendung einer Union.

```
/* Programm Union1.cpp
   Demonstration einer Union */
#include <iostream>

typedef enum { Ganzzahl, Reelle_Zahl } Tfall;

union TWert {
  int   x;
  float y;
};

struct Tunionstruc {
  int   index;
  Tfall fall;
  TWert wert;
};

void main() {
  Tunionstruc us;

  do {
    cout << "Zahl eingeben (0=Ende) : "; // Testen Sie mit float- und int-Eingabe!
    cin >> us.wert.y; // als float, könnte auch als int eingegeben werden
                      // Dann müßte man schreiben: cin >> us.wert.y;
    cout << "int-Darstellung der Zahl: ";   cout << us.wert.x << endl;
```

```
    cout << "float-Darstellung der Zahl: "; cout << us.wert.y << endl;
  } while (us.wert.y);
}
```

Die Programmausgabe zeigt, dass nur die *float*-Variable y im richtigen Datenformat gespeichert wird. Allerdings ändert sich mit jeder Eingabe von y auch der für die *int*-Variable y gespeicherte Wert ebenso! (Das muss auch so sein, denn der Computer verwendet ja den gleichen Speicherplatz für beide Zahlenwerte x und y.) Versucht man, x auf dem Bildschirm auszugeben, zeigt sich ein falscher Zahlenwert, denn der Computer hat den Speicherplatz im float-Format belegt.

Im obigen Programmbeispiel müssen die Variablen für die Zahlens x und y recht umständlich als als us.wert.x und us.wert.y angesprochen werden. Möchte man eine einfachere Syntax (us.x und us.y), kann man eine *anonyme Variante* verwenden, die keinen Namen hat:

```
struct {
  int Nummer;
  Tfall fall;
  union {         // Diese Variante ist nun namenlos!
    int    x;
    float  y;
  };
};

/* Programm Union2.cpp
   Demonstration einer anonymen Union */
#include <iostream>

typedef enum { Ganzzahl, Reelle_Zahl } Tfall;

struct Tunionstruc {
  int   index;
  Tfall fall;
  union {
    int   x;
    float y;
  };
};

void main() {
  Tunionstruc us;

  do {
    cout << "Zahl eingeben (0=Ende) : "; // Testen Sie mit float- und int-Eingabe!
    cin >> us.y; // als float, könnte auch als int eingegeben werden
                      // Dann müßte man schreiben: cin >> us.wert.y;
    cout << "int-Darstellung der Zahl: ";   cout << us.x << endl;
    cout << "float-Darstellung der Zahl: "; cout << us.y << endl;
  } while (us.y);
}
```

Grundlegende Aufgabe	union1.cpp

Führen Sie das Programm aus. Experimentieren Sie mit verschiedenen Datentypen.

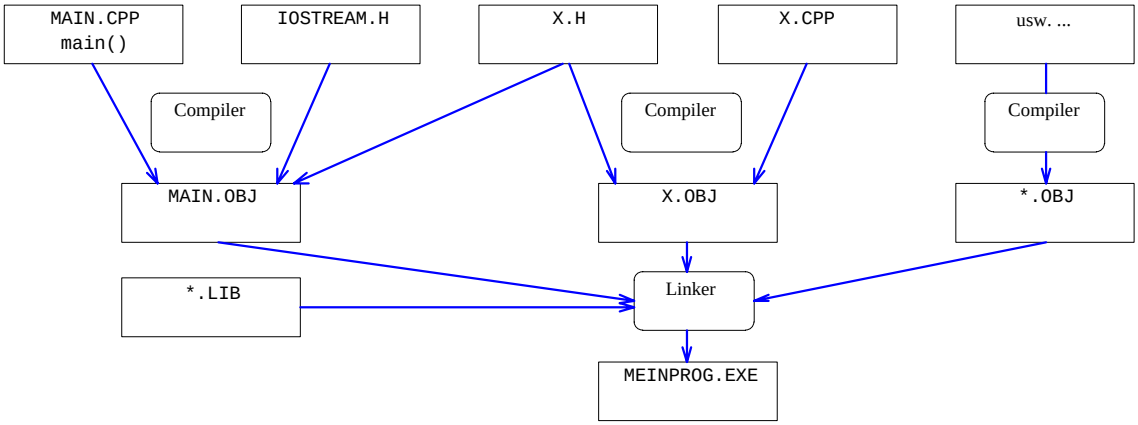
Grundlegende Aufgabe	union2.cpp

Führen Sie das Programm aus. Experimentieren Sie mit verschiedenen Datentypen.

2.5 Verteilte Programmentwicklung, Wiederverwendbarkeit

Verbunde, Klassen und andere selbstdefinierte Datentypen werden eingesetzt, um sie mehrfach verwenden zu können, entweder in mehreren verschiedenen Programmen oder in mehreren Programm-Modulen, die getrennt entwickelt werden. Dabei muss der Quellcode nicht jedesmal neu übersetzt werden,

sondern es genügt, wenn die einzelnen Module die Definitionen der Klasse kennen, die der einzelnen Elementfunktionen dagegen nicht.



Im Beispiel würde X.CPP den Quellcode für die Elementfunktionen enthalten, X.H dagegen nur die Klassendefinition. Damit kann die Klasse in jedem einzelnen Modul des Hauptprogrammes eingesetzt werden.

Daher trennt man üblicherweise die Deklaration von Klassen bzw. Datentypen oder Funktionen von deren Definition, bei Klassen genauer: von der Definition der Methoden.

Beispiel:

x.h:

```
// Deklarationen

class a
{
  private:
    void xy();
  public:
    int zz();
}
```

x.cpp:

```
#include "x.h" // Deklarationen einbinden

void a::xy()
{…
}

int a ::zz()
{…
}
```

Beachten Sie, dass bei der include-Direktive in diesem Fall Anführungszeichen "" verwendet werden, keine Klammern <>.

Grundlegende Aufgabe	adressen4b.cpp

Lagern Sie in der Adressverwaltung die Deklarationen der Klassen in Header-Dateien aus.

2.6 Arrays (Felder)

Lerninhalte

1. Der Begriff „Array“
2. Eindimensionale Arrays
3. Mehrdimensionale Arrays
4. Arrays als Parameter in Funktionen
5. Vektoren

Lerninhalte

Der Begriff „Array“

Sowohl im täglichen Leben als auch in Programmen müssen häufig große Mengen von Daten verarbeitet werden. Man denke z.B. an einen Betrieb, der die Personaldaten von 1000 Mitarbeitern verwalten muß.

Für solche Zwecke ordnet man meist jedem Beschäftigten eine laufende Nummer zu. Die daraus resultierende Ordnungsstruktur kann folgendermaßen verallgemeinert werden:

Beliebige, zunächst ungeordnete Daten werden in „durchnumerierte Fächer“ gelegt. Kennt man die „Fachnummer“ (vergleichbar mit einer Schließfachnummer bei Bankschließfächern), kann man auf den Fachinhalt zugreifen. Als „Fachinhaltstyp“ wurde im Beispiel *char* gewählt, es könnte aber auch ein selbstdefinierter *enum*-Typ oder ein komplexer *struct*-Typ sein - nur eine Einschränkung gibt es: In allen Fächern müssen Datenelemente vom *gleichen* Typ aufbewahrt werden.

Eine Datenstruktur mit solchen Eigenschaften nennt man *Array* (auch *Feld* oder *Tabelle* genannt). Handelt es sich um eine *eindimensionale* Tabelle (wie im obigen Beispiel zwar mehrere Spalten, aber nur eine einzige Zeile), dann heißt die Datenstruktur *Vektor*. Die „Fachnummer“, mit der eine Arrayelement angesprochen wird, heißt *Index*.

Eine *zweidimensionale* Tabelle mit mehreren Zeilen und Spalten nennt man auch *Matrix*. Man kann eine Tabelle (Array) mit beliebig vielen Dimensionen definieren - bis zur dritten Dimension ist sie noch physikalisch vorstellbar als Quader mit Einzelfeldern. Bei einer Matrix benötigt man mehrere Indizes, um ein bestimmtes Element anzusprechen.

In C++ ist der untere Grenzwert für den Index eines Arrays immer 0. C++ kennt einfache Arrays, wie sie auch in der Programmiersprache C existieren (sog. *C-Arrays*) und Arrays mit komfortableren Zugriffsmethoden, die über Objektklassen definiert sind (genannt *vector*). Für C-Arrays gelten die folgende Regeln:

- Die Anzahl der Array-Elemente muß bei der Array-Deklaration festgelegt werden. Dabei gilt immer: größter erlaubter Wert für den Index = Zahl der Arrayelemente-1
- Array-Elemente werden nicht automatisch bei der Deklaration initialisiert. Dies muß explizit durch den Programmierer vorgenommen werden, ansonsten sind die enthaltenen Werte zufällig.
- Der Compiler prüft keine Zugriffe auf ungültige Array-Indizes! Lesen von oder (noch schlimmer) Schreiben auf ungültige Array-Positionen führt zu Programmfehlern oder zum Programmabsturz.

L2 Eindimensionale Arrays

Die Deklaration eines einfachen, eindimensionalen Arrays geschieht folgendermaßen:

Datentyp Arrayname [Elementeanzahl];

Elementeanzahl muß eine Konstante sein oder ein Ausdruck, der ein konstantes Ergebnis hat. Der höchste ansprechbare Index der Array-Elemente beträgt *Elementeanzahl-1*, da das erste Arrayelement immer den Index 0 hat.

```
float Regal[6];      // Array mit 6 float-Elementen mit Index 0 bis 5
Regal[3] = 7.8;      // 3. Regalfach mit Zahl belegen.

const int ARMAX=17;  // Konstante für die Arraygrösse: 17 Elemente
double ar[ARMAX];    // double-Array deklarieren
for (int i=0; i<=ARMAX-1; ++i) // erlaubter Indexbereich 0 bis ARMAX-1
    ar[i]=0.0;       // Arrayelemente initialisieren
```

Grundlegende Aufgabe	wetter5.cpp

Der Deutsche Wetterdienst schickt sechs Meßballons aus, die an verschiedenen Stellen der Bundesrepublik den Luftdruck in bar messen und die Daten zur Bodenstation funken. Die sechs Meßwerte sollen nach der Dateneingabe als Balkendiagramm dargestellt werden; minimaler und maximaler Meßwert sollen in bar ausgegeben werden.

Schreiben Sie ein Programm *Wetter5.cpp* mit der nachfolgend dargestellten Bildschirmausgabe. Das Programm soll so ausgelegt werden, dass maximal 20 *float*-Messwerte eingegeben werden können. Nach Eingabe des 20. Meßwerts wird automatisch das Balkendiagramm ermittelt.

```
Luftdruck-Verteilung in bar.

1. Messwert (0=Ende) 1.37
2. Messwert (0=Ende) 0.78
3. Messwert (0=Ende) 0.93
4. Messwert (0=Ende) 1.14
5. Messwert (0=Ende) 1.03
6. Messwert (0=Ende) 1.15
7. Messwert (0=Ende) 0

1  ██████████████████
2  ████████████
3  ████████████
4  ███████████████
5  ██████████████
6  ████████████████

   ■ entspricht 0.1 bar
Maximum: 1.37 bar
Mimimum: 0.78 bar

Programmende.
```

Ein Array kann nicht nur einfache, sondern auch komplexe Datentypen wie Strukturen oder Objekte als Elemente speichern.

```
class Tballon {     // Deklaration einer Objektklasse Tballon
private:
  int Temperatur;
  float Luftdruck;
  short unsigned int Feuchte;
  bool Regen;
  enum { morgens=1,mittags,abends } Tageszeit;
  bool JNEingabe();
  void AusgabeTageszeit();
public:
  void Eingabe();
  void Ausgabe();
};
/* ... Hier folgen die Methodendefinitionen wie bei Wetter3.cpp ... */

void main()
{
   const int MAXBALLONS=100;    // Maximalanzahl der Ballons festlegen
   Tballon Ballon[MAXBALLONS];  // Array von Ballons deklarieren
   cout << "Eingabe der Datensaetze von Messwerten:" << endl;
   for (int i=0; i<=MAXBALLONS-1; ++i)
   { // Alle Ballon-Messwerte eingeben
       Ballon[i].Eingabe(); // Aufruf der Methode Eingabe()
   }
   cout << "Ausgabe der Datensaetze von Messwerten:" << endl;
   for (int i=0; i<=MAXBALLONS-1; ++i)
   { // Alle Ballon-Messwerte ausgeben
       Ballon[i].Ausgabe(); // Aufruf der Methode Ausgabe()
   }
   cout << "\nProgrammende.";
}
```

In diesem Beispiel ist der Typ eines Arrayelementes „Tballon“, also eine Objektklasse. Folglich ist jedes Arrayelement ein Objekt, dessen Elementdaten und Elementfunktionen dann über den Punkt-Operator angesprochen werden können.

Grundlegende Aufgabe `adressen5.cpp`

Verändern Sie adressen4.cpp, so dass es nicht nur einen Adressdatensatz speichern kann, sondern mehrere. Es genügt, wenn einige Datensätze eingegeben werden und diese dann wieder ausgegeben werden.

Weiterführende Aufgabe `adressen6.cpp`

Verändern Sie adressen5.cpp, so dass es ein Menü bietet, mit dem der Benutzer bestimmte Aktionen auswählen kann. Verwenden Sie dazu eine fußgesteuerte Schleife, in deren Inneren auf Eingaben entsprechend reagiert wird. Sie brauchen auch einen Zähler, der die Anzahl der aktuell gespeicherten Elemente enthält.

```
Adressverwaltung
0 Adressen gespeichert.
<N>eue Adresse eingeben
Adressen<l>iste ausgeben
<E>nde
Auswahl:
```

L3 Mehrdimensionale Arrays

Ein zweidimensionales Array ist vorstellbar als *Tabelle* mit Zeilen und Spalten. Ein Index des Arrays ist demnach die Zeile, der andere Index ist die Spalte. Es wird folgendermaßen deklariert:

```
const int Zeilen  = 3, // Zunächst Konstanten definieren
          Spalten = 7; //          (hier für zweidimensionales Array)
float Tabelle[Zeilen][Spalten]; // float-Array mit 3 Zeilen und 7 Spalten

Tabelle[1][2] = 5.8; // belegt das Element in der 2. Zeile, 3. Spalte
```

	0	3.4	5.6	7.8	9.3	4.6	4.8	3.2
3 Zeilen	1	2.9	3.7	5.8	2.6	4.1	3.2	8.9
	2	4.3	6.5	8.7	3.9	2.3	6.4	8.5
		0	1	2	3	4	5	6

7 Spalten

Bei mehrdimensionalen Arrays muß man wissen, wie die einzelnen Array-Elemente gespeichert werden. Der physikalische Speicher des Rechners ist natürlich nicht mehrdimensional organisiert, deshalb werden die einzelnen Array-Elemente dort sequentiell angeordnet. Dabei werden die Elemente der *zuletzt notierten* Dimension aufeinanderfolgend im Speicher abgelegt, danach folgt die zweitletzte Dimension, usw. .

Beispiel Dreidimensionales Array

Bei einem dreidimensionalen Array *float Wuerfel[4][3][2]* werden die Speicherelemente in folgender Reihenfolge abgelegt:

[0][0][0], [0][0][1],

[0][1][0], [0][1][1],

[0][2][0], [0][2][1],

...

[3][2][0], [3][2][1].

Array-Elemente können gleichzeitig mit ihrer Deklaration auch initialisiert werden. Hier muß man genau beachten, in welcher Reihenfolge die einzelnen Array-Elemente abgelegt werden:

```
const int MAX_SPALTEN = 3;
const int MAX_ZEILEN = 4;
int matrix[MAX_ZEILEN][MAX_SPALTEN] = { 1,  2,  3,  // 0. Zeile
                                        4,  5,  6,  // 1. Zeile
                                        7,  8,  9,  // 2. Zeile
                                       10, 11, 12 } // 3. Zeile
```

Eine komplizierte Array-Deklaration kann man mittels *typedef* auch übersichtlicher schreiben:

```
typedef int tmatrix[3][4];
tmatrix matrix1; // entspricht: int matrix1 [3][4];
tmatrix matrix2; // entspricht: int matrix2 [3][4];
```

Arrays als Parameter in Funktionen

In C++ kann man Arrays als Funktionsparameter übergeben. Hierbei kann man die Größe des Arrays im Parameter entweder fest angeben oder auch *offen* halten:

```
int GlobalArray[10];                          // Globale Arrayvariable
// Funktionsprototyp
int GenauArray(int arr[100]);                 // Feste Größenangabe
void VarArray(int arr[]; int num_elem);       // Offene Größenangabe
// Funktionsaufruf
VarArray(GlobalArray,10); // Aktueller Array-Parameter (ohne [] !!)
```

Ein Array kann, im Gegensatz zu anderen Datentypen, *nicht als Werteparameter* übergeben werden, es ist grundsätzlich ein Variablenparameter! Will man den Inhalt der globalen Arrayvariable nicht verändern, kann man *const* in der Parameterdeklaration verwenden.

Die Größe eines Arrays ist der aufrufenden Funktion nicht bekannt, egal ob in der Funktionsdeklaration offene Arraygrenzen verwendet wurden oder nicht. Wir erwähnten bereits, dass der Compiler bei Zugriffen auf ungültige Arrayelemente keine Warnungen ausgibt. Ein Programmierer muß daher innerhalb einer Funktion, die ein Array als Parameter übernimmt, selbst auf die Einhaltung der Arraygrenzen achten. Das kann er aber nur, wenn als weiterer Parameter die Anzahl der Elemente des aktuellen Array-Parameters beim Funktionsaufruf übergeben wird (im obigen Beispiel als *num_elem*). Bei mehrdimensionalen Arrays als Parameter darf nur die *erste* Array-Dimension offen gehalten werden.

```
int MimMat(int intMat[][100], int Zeilen, int Spalten);
```

Im folgenden Beispielprogramm werden die Werte eines Arrays mit geradzahligen Integer-Werten belegt. Dabei wird ein Array-Parameter mit offener Größenangabe verwendet.

```
/* Programm ArrFunc.cpp
   Demonstration eines Arrays als Funktionsparameter */
#include <iostream>

void ArrayFuellen(int arr[], int anzahl) {
  for (int i=0; i<anzahl; i++) arr[i] = i*2;
}

void main() {
  const GROESSE = 6;
  int Array[GROESSE];

  ArrayFuellen(Array,GROESSE);
  cout << "Array-Werte: ";
  for (int i=0; i<GROESSE; i++) cout << Array[i] << " ";
}
```

```
Array-Werte: 0 2 4 6 8 10
```

Weiterführende Aufgabe `Wetter6.cpp`

Jeweils am 1. von drei aufeinanderfolgenden Monaten werden die Luftdruckwerte von vier Wetterballons ermittelt. Schreiben Sie ein Programm *Wetter6.cpp*, beim dem zunächst alle diese Werte eingegeben werden. Das Programm soll alle Messwerte in einer Tabelle anzeigen. Zusätzlich werden die Druck-Mittelwerte jedes Ballons über die drei Monate gemittelt, sowie die Druckmittelwerte aller vier Ballone in jedem Monat angezeigt. Das Programm soll die Messungen von maximal 20 Ballons während eines ganzen Jahres (12 Monate) aufnehmen und verarbeiten können.

```
Eingabe von Luftdruck-Werten in bar.
Letzter Beobachtungsmonat (1..12): 3
Anzahl der Messballons:            4

 1. Monat.
Druck bei Ballon Nr. 1? 1.37
Druck bei Ballon Nr. 2? 0.78
Druck bei Ballon Nr. 3? 0.93
Druck bei Ballon Nr. 4? 1.14
 2. Monat.
Druck bei Ballon Nr. 1? 0.84
Druck bei Ballon Nr. 2? 1.23
Druck bei Ballon Nr. 3? 0.81
Druck bei Ballon Nr. 4? 0.99
 3. Monat.
Druck bei Ballon Nr. 1? 1.23
Druck bei Ballon Nr. 2? 0.84
Druck bei Ballon Nr. 3? 0.93
Druck bei Ballon Nr. 4? 0.87

                    Druckmessungen.
Feld links oben:          Gesamtdurchschnitt aller gemessenen Druecke.
Oberste Zeile (Monat 0): Durchschnittsdruecke aller Monate bei jedem Ballon.
Linke Spalte (Ballon 0): Durchschnittsdruecke aller Ballons in jedem Monat.

                    Ballon Nr.
                0      1     2     3     4
Monat   0    0.997  1.15  0.95  0.89     1
Monat   1     1.06  1.37  0.78  0.93  1.14
Monat   2    0.967  0.84  1.23  0.81  0.99
Monat   3    0.967  1.23  0.84  0.93  0.87

Programmende.
```

Weiterführende Aufgabe Wetter7.cpp

Schreiben Sie *Wetter6.cpp* nach *Wetter7.cpp* um. Bei gleicher Bildschirmausgabe soll eine Objektklasse *TDruckMess* verwendet werden, in der möglichst viele Eigenschaften und Methoden des Programmes gekapselt werden sollen. Das Hauptprogramm von Wetter7.cpp sieht folgendermaßen aus:

```
void main() {
  TDruckMess DMess;

  cout << "Eingabe von Luftdruck-Werten in bar.\n";
  cout << "Letzter Beobachtungsmonat (1..12): "; DMess.MaxMonatEin();
  cout << "Anzahl der Messballons:            "; DMess.MaxBallonEin();
  cout << endl;
  DMess.DruckEin();
  DMess.MonatsSchnitt();
  DMess.BallonSchnitt();
  DMess.GesamtSchnitt();
  DMess.DruckTabelle();
  cout << "\nProgrammende.";
}
```

Weiterführende Aufgabe MinMax.cpp

Schreiben Sie ein Programm *MinMax.cpp*, das Sie zunächst auffordert, eine Anzahl für Datenwerte einzugeben. Gültige Eingaben sollen im Bereich von 2 bis 10 liegen. Danach sollen Integerwerte eingegeben werden. Das Programm zeigt nach Abschluß der Eingabe den kleinsten und größten Arrayinhalt an. Verwenden Sie im Programm die folgende Funktion mit offenen Arraygrenzen:

```
int findMin(int Array[], int Groesse);
```

```
Geben Sie die Anzahl der Datenwerte ein [2 to 10]: 5
Array[0]: 55
Array[1]: 69
Array[2]: 47
Array[3]: 85
Array[4]: 14
Der kleinste Wert im Array lautet 14
Der groesste Wert im Array lautet 85
```

Weiterführende Aufgabe Omatrix.cpp

Schreiben Sie ein Programm *Omatrix.cpp*. Der Benutzer wird zur Eingabe von Reihen und Spalten für eine Matrix aufgefordert. Danach werden alle Matrixelemente eingegeben und die Spaltenmittelwerte werden ausgegeben. Die folgende Funktion mit offenen Arraygrenzen ist zu verwenden:

```
void EingabeMatrix(double matrix[][MAX_SPALTE], int Zeilen, int Spalten)
```

```
Anzahl der Zeilen [2 to 30]: 3
Anzahl der Spalten [1 to 10]: 2
matrix[0][0]: 10
matrix[0][1]: 20

matrix[1][0]: 40
matrix[1][1]: 50

matrix[2][0]: 70
matrix[2][1]: 80

Mittelwert fuer Spalte 0 = 40
Mittelwert fuer Spalte 1 = 50
```

L5 Vektoren

So bequem einfache C-Arrays zu handhaben sind, so tückisch sind die Programmfehler, die auftreten, wenn beim Elementezugriff Arraygrenzen überschritten werden: Beim lesenden Elementezugriff werden falsche Werte im Programm verarbeitet. Bei einem schreibenden Arrayzugriff auf ein falsches Element können sogar wichtige Speicherbereiche überschrieben werden, die das gesamte Programm oder gar das Betriebssystem zum Absturz bringen.

C++ bietet eine normierte *Standardbibliothek* (auch *STL - standard template library* genannt), in der verschiedene Datenstrukturen und Algorithmen definiert sind, die die Bearbeitungsmöglichkeiten von Arrays wesentlich verbessern. Innerhalb der STL ist die Klasse *vector* definiert, welche Arrays mit folgenden zusätzlichen Eigenschaften zur Verfügung stellt:

- Zur Programmlaufzeit kann die Zahl der Arrayelemente ermittelt werden.
- Beim Zugriff auf ungültige Arrayelemente kann eine Fehlermeldung erzeugt werden.
- Die Arraygröße kann zur Programmlaufzeit dynamisch verändert werden, indem man am Arrayende Elemente anhängt.

Ein Vektor ist in C++ als vordefinierte Klassen-Schablone realisiert. Das ist eine Klasse, bei der der Typ der Inhaltselemente zunächst noch offen bleibt und später frei gewählt werden kann. Wir haben Schablonen (Templates) schon bei den Funktionen angesprochen - man kann sie auch für Klassen definieren. Das wurde in der Standardbibliothek von C++ in der Header-Datei *vector* getan. Dort steht:

```
template <class T>
class vector {
  /* verschiedene Eigenschaften und Methoden für die Klasse vector
     mit dem allgemein gehalteten Datentyp T */
}
```

In einem C++-Programm definiert man einen Vektor mit dem Namen *Reihe*, der 10 Elemente des Typs *int* aufnehmen kann, folgendermaßen:

```
#include <vector>
using namespace std; // Nur so ist die Klasse vector verfügbar
vector<int> Reihe(10); // Für den allgemeinen Typ T wurde hier int eingesetzt
```

Wie bei C-Arrays sind die Elemente von 0 bis n-1 durchnumeriert, hier also von 0 bis 9. Der indizierte Zugriff kann ebenso erfolgen wie bei C-Arrays:

```
cout << Reihe[3]; // stellt das 4. Vektorelement dar
```

Da aber auch bei Vektoren beim Zugriff über [] die Einhaltung der Arraygrenzen nicht geprüft wird, sollte der Zugriff besser über die eingebaute Elementfunktion *at* erfolgen:

```
cout << Reihe.at(3);    // stellt das 4. Vektorelement dar
cout << Reihe.at(1000); // ergibt Programmabbruch mit Fehlermeldung
```

Die Elementfunktion *size* eines Vektors gibt die definierte Maximalzahl der Elemente an:

```
cout << Reihe.size();
```

Oft weiß man zum Zeitpunkt der Programmerstellung nicht, wie groß ein Vektor werden soll, zum Beispiel beim Einlesen von Daten aus einer Datei. Daher kann man an einen v*ector* mit der Methode *push_back* bei Bedarf einzelne Elemente anhängen:

```
vector<int> Reihe; // anfängliche Größe ist 0
Reihe.pushback(15); // int-Wert 15 wird angehängt
cout << Reihe.size(); // 1 wird ausgegeben
```

Die folgende Tabelle zeigt eine Zusammenfassung von typischen Elementfunktionen, die auf Vektoren angewandt werden können. Nehmen Sie an, es sei *V* definiert als *vector(T)* vom Typ *X* mit Elementen *t* vom Typ *T*, V[n] sei das Vektorelement an Platz Nr. n.

Typ der Funktion	*Funktionsaufruf*	*Bedeutung des Funktionsaufrufs*
void	V.push_back(t)	Fügt Element t am Ende des Vektors an
void	V.pop_back()	Löscht das letzte Element von Vektor V
X	V.at(n)	Gibt Referenz auf n-tes Element von V zurück
void	V.reserve(n)	Reserviert Speicher für n Vektorelemente
unsigned int	V.size()	Gibt die aktuelle Vektorgröße an
bool	V.empty()	Wird wahr, wenn V.size() = = 0

Falls im Hauptspeicher kein Platz zum Vergrößern des Arrays vorhanden sein sollte, wird der gesamte Vektor automatisch - ohne Zutun des Programmierers - an eine neue Speicherstelle verlagert. Im folgendem Programmbeispiel DynVekt.cpp wird das dynamische Verändern der Vektorgröße demonstriert:

```
// Programm DynVekt.cpp
// Demonstration eines Vektors mit dynamischer Größenänderung
#include <iostream>
#include <vector>
using namespace std; // Standard-Namensbereich definieren

int wert;
vector<int> Reihe; // anfängliche Größe ist 0

void main() {
  do {
    cout << "Wert eingeben (0=Ende): "; cin >> wert;
    if (wert) Reihe.push_back(wert); // Wert in Array anhängen
  } while (wert !=0);

  cout << endl;
  cout << "Die folgenden Werte wurden eingegeben:" << endl;
  for (int i=0; i < Reihe.size(); i++)
    cout << i << ". Wert : " << Reihe[i] << endl;
  cout << endl << "Anzahl der Array-Elemente: " << Reihe.size() << endl;
}
```

```
Wert eingeben (0=Ende): 3
Wert eingeben (0=Ende): 4
Wert eingeben (0=Ende): 2
Wert eingeben (0=Ende): 7
Wert eingeben (0=Ende): 0

Die folgenden Werte wurden eingegeben:
0. Wert : 3
1. Wert : 4
2. Wert : 2
3. Wert : 7

Anzahl der Array-Elemente: 4
```

Weiterführende Aufgabe `Array1.cpp`

Schreiben Sie ein Programm *Array1.cpp*, bei dem grundlegende Abläufe der Tabellenverarbeitung realisiert werden. Benutzen Sie ein eindimensionales *float*-Array und nehmen Sie als Vorlage das Programmfragment *Array1.fra*.

Initialisieren der Array-Elemente mit einem Anfangswert

Eingeben der Array-Elemente von der Tastatur

Verschieben der Array-Elemente um Indexplätze

Spiegeln der Array-Elemente (letztes Element wird erstes, usw.)

Kopieren von Array-Elementen zwischen zwei Arrays

Selektieren von Array-Elementen nach Wert oder Index

Summieren der Werte aller Array-Elemente

Suchen eines bestimmten/größten/kleinsten Array-Elements

```
--------------------------------
0  Programmende
1  Array initialisieren
2  Array eingeben
3  Array links verschieben
4  Array rechts verschieben
5  Array umdrehen
6  Array kopieren
7  Array nach Index auswaehlen
8  Array nach Wert auswaehlen
9  Array summieren und Maximum
--------------------------------
```

```
Wahl? 1
[0]=0  [1]=0  [2]=0  [3]=0  [4]=0  [5]=0  [6]=0  [7]=0  [8]=0
Wahl? 2
[0]=0  [1]=0  [2]=0  [3]=0  [4]=0  [5]=0  [6]=0  [7]=0  [8]=0
Werte eingeben (Ende=Index ausserhalb [0-8]):
Index Wert? 0 2
Index Wert? 1 4
Index Wert? 2 6
Index Wert? 3 8
Index Wert? 4 10
Index Wert? 5 12
Index Wert? 6 14
Index Wert? 7 16
Index Wert? 8 18
Index Wert? 9 20
[0]=2  [1]=4  [2]=6  [3]=8  [4]=10  [5]=12  [6]=14  [7]=16  [8]=18
Wahl? 3
[0]=2  [1]=4  [2]=6  [3]=8  [4]=10  [5]=12  [6]=14  [7]=16  [8]=18
Elemente 1-8 um 1 nach links verschoben.
[0]=4  [1]=6  [2]=8  [3]=10  [4]=12  [5]=14  [6]=16  [7]=18  [8]=18
Wahl? 4
[0]=4  [1]=6  [2]=8  [3]=10  [4]=12  [5]=14  [6]=16  [7]=18  [8]=18
Elemente 0-7 um 1 nach rechts verschoben.
[0]=4  [1]=4  [2]=6  [3]=8  [4]=10  [5]=12  [6]=14  [7]=16  [8]=18
Wahl? 5
[0]=4  [1]=4  [2]=6  [3]=8  [4]=10  [5]=12  [6]=14  [7]=16  [8]=18
Elemente um Mittelachse gespiegelt.
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=4
Wahl? 6
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=4
... kopieren: von nach? 6 8
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Wahl? 7
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Schrittweite fuer Index? 3
18 12 6
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Wahl? 8
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Kleinster anzuzeigender Wert? 12
18 16 14 12
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Wahl? 9
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Summe: 112, Maximum: 18
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Wahl? 0
[0]=18  [1]=16  [2]=14  [3]=12  [4]=10  [5]=8  [6]=6  [7]=4  [8]=6
Programmende Array1.
```

Weiterführende Aufgabe `Array2.cpp`

Schreiben Sie das Programm Array1.cpp zu Array2.cpp um, indem Sie einen *vector*<float> verwenden.

2.7 Zeichenketten (Strings)

Lerninhalte

❶ C-Strings (Zeichenarrays)
❷ Die C++-Klasse „string“
❸ String-Methoden

Lerninhalte

Unter einer *Zeichenkette* (einem *String*) versteht man zunächst ein Array von Einzelzeichen (mit dem Datentyp *char)*, welches man als zusammenhängende Folge von Zeichen („Text“) verarbeiten kann. Es gibt Programmiersprachen, die die Höchstanzahl von Einzelzeichen in einem String begrenzen (z.B. auf 255). In C-basierenden Sprachen kann ein String beliebig lang werden - seine Länge wird lediglich physikalisch durch den zur Verfügung stehenden Hauptspeicher begrenzt.

C-Strings (Zeichenarrays)

Am String-Ende eines C-Strings steht stets das (nicht darstellbare) Zeichen mit dem ASCII-Code 0 (‘\0’) - das Ende des Strings wird dadurch gekennzeichnet. Man redet auch von ASCIIZ-Strings („Z“ steht für „Zero“). Bislang benutzen wir in unseren Beispielen immer *konstante Strings* (String-Literale). String-Literale können mit *cout* auf dem Bildschirm ausgegeben werden.

```
cout << "Hallo Welt" << endl;
```

Bei der Stringeingabe mit *cin* haben wir jedoch das Problem, dass ein eingegebenes Leerzeichen als Endezeichen interpretiert wird. Daher benutzt man die *cin*-Methode *getline*, die einen String bis zur einer angegebenen Stringlänge-1 einliest. Der Einlesevorgang wird durch die Eingabe eines Begrenzungszeichen beendet. Wenn kein Begrenzungszeichen angegeben ist, wird standardmäßig die RETURN-Taste ('\n') als Eingabeende interpretiert.

```
// Anwendung: cin.getline(Stringvariable,Stringlaenge,Begrenzungszeichen)
// Programm String1.cpp
// Stringeingabe
#include <iostream>

void main()
{
  char name[10];  // String mit max 9 Zeichen [0] bis [8]
                  // begrenzt durch \0 an Position [9]
  char aufford[]="Zeichenkette eingeben: "; // wirkt wie aufford[24]
                //12345678901234567890123
                //        10        20

  cout << aufford;
  cin.getline(name,sizeof(name)); // sizeof() : Größe des Parameters in Bytes
  cout << "Sie haben die Zeichenkette " << name << " eingegeben." << endl;
  cout << "Länge der Eingabeaufforderung: " << sizeof(aufford) << endl;
  cout << "Länge der Zeichenkette: " << sizeof(name) << endl;
  for (int i=0;i<sizeof(name);i++)
    cout << "Zeichen Nr. " << i << ": " << name[i] << endl;
}
```

```
Zeichenkette eingeben: abcdefghijklmn
Sie haben die Zeichenkette abcdefghi eingegeben.
Lõnge der Eingabeaufforderung: 24
Lõnge der Zeichenkette: 10
Zeichen Nr. 0: a
Zeichen Nr. 1: b
Zeichen Nr. 2: c
Zeichen Nr. 3: d
Zeichen Nr. 4: e
Zeichen Nr. 5: f
Zeichen Nr. 6: g
Zeichen Nr. 7: h
Zeichen Nr. 8: i
Zeichen Nr. 9:
```

Wie man sieht, kann man die Maximallänge eines Strings (inklusive '\0') entweder explizit angeben (char name[10]) oder vom Compiler dadurch errechnen lassen, dass man die Stringvariable bei der Deklaration mit einer Stringkonstante initialisiert (char aufford[] = "..."). Das abschließende Zeichen '\0' wird automatisch bei einer Stringzuweisung einer Stringkonstante mit = oder beim Aufruf der Funktion *getline* angehängt.

Einfache Zeichenarrays wie oben definiert (auch *C-Strings* genannt), haben entscheidende Nachteile: Ersetzen Sie in *Cstring1.cpp* testweise den Funktionsaufruf

```
cin.getline(name,sizeof(name));
```

durch

```
cin.getline(name,20));
```

und geben Sie dann einen String mit 20 Zeichen ein. Sie werden sehen, dass die Programmausgabe den gesamten String anzeigt! Das kann aber nur bedeuten, dass der eingegebene String vollständig ab der Speicheranfangsadresse von *name* gespeichert wird - und dabei ohne Rücksicht auf Verluste im Speicher nachfolgende Datenstrukturen überschreibt, die unter Umständen für den weiteren Fortgang des Programmes sehr wichtig sind. Paßt der Programmierer hier nicht auf, kann das Programm abstürzen!

L2 Die C++-Klasse „string“

Unter C++ können leistungsfähigere Strings als *Objekte* der STL-Klasse *string* erzeugt werden. Nach Einbinden der Headerdatei *string* (mit *#include<string>*) verfügt ein C++-Programm über Strings mit umfangreichen Operatoren und Elementfunktionen, welche den C-Zeichenarrays weit überlegen sind.

```
#include <string>
string Str1, Str2;             // Str1: Zielstring, Str2: String als Parameter
string Str3("Testwort");       // Stringdeklaration mit Initialisierung
string Str4 = "Noch ein Wort"; // Andere Initialisierungsvariante
string Str5(Str3);             // Str5 erhält den Inhalt von Str3
```

Der Speicherplatz für String-Objekte (Kapazität) wird bei der Deklaration dynamisch erzeugt. Bei einer Initialisierung nach obigem Muster ist die Kapazität gleich der Anzahl der gespeicherten Zeichen. Bei Stringzuweisungen innerhalb von Programmen wird die Kapazität automatisch erhöht, wenn dies notwendig ist. Die Stringkapazität kann aber auch explizit gesetzt werden:

```
string Str6("Langer String",100); // 13 Zeichen gespeich., Kap. 100 Zeichen
string Str7(10,'\n'); // 10 Zeilenvorschübe gespeichert.
```

Wie bei Vektoren kann auf ein einzelnes Stringzeichen entweder mit dem Indizierungsoperator *[]* oder mit der Elementfunktion *at()* zugegriffen werden.

```
Str1 = "Test";
cout << Str1[3];        // ergibt 't'
cout << Str1.at(3);     // das gleiche
```

Bei der Verwendung der nachfolgenden Operatoren können C++-Strings und C-Zeichenarrays gemischt verwendet werden.

- = Zuweisung eines Strings an einen andern (Kopieren)
- += Anhängen eines Strings
- + Aneinanderfügen zweier Strings
- = = Vergleich zweier Strings: *true* bei Übereinstimmung
- != Vergleich zweier Strings: *true* bei Unterschieden
- < *true*, wenn linker String alphabetisch vor dem rechten kommt (entsprechend: >)
- <= *true*, wenn wie bei < oder wenn die Strings gleich sind (entsprechend: >=)

L3 String-Methoden

Die in den Beschreibungen verwendete Typbezeichnung *uint* entspricht *unsigned int.* Bei allen Methoden, die als Parameter s einen C++-*String* verwenden, kann stattdessen auch ein C-String (Zeichenarray) angegeben werden. Wird der Parameter *n* weggelassen, entspricht er der Länge des Stringparameters s. Die Konstante *string::npos* bezeichnet einen ungültigen Positionswert (intern repräsentiert durch -1).

Die Methoden *cout* und *cin* für die Ausgabe eines Strings auf dem Bildschirm und die Eingabe über Tastatur werden unterstützt. Allerdings wird bei *cin* das erste auftretende Leerzeichen als String-Endezeichen interpretiert, so dass hiermit keine Strings mit eingeschlossenen Leerzeichen eingelesen werden können. Zu diesem Zweck ist die Methode *getline* mit folgender Aufrufsyntax zu benutzen:

```
string s;
getline(cin, s, '\n'); // liest ganze Textzeile nach s ein
```

- int *compare*(const string& s); bzw. int *compare*(const string& s, uint apos, uint n);

  Der Zielstring wird mit dem String s verglichen. Das Funktionsergebnis ist kleiner, gleich oder größer 0, abhängig davon, ob der Zielstring kleiner, gleich oder größer s ist. Wenn *apos* und *n* angegeben sind, werden nur die ersten n Zeichen des Zielstrings verglichen, beginnend ab Position *apos.* Wenn n nicht angegeben ist, werden alle Zeichen des Zielstrings ab *apos* verglichen.

  ```
  if (Str1.compare(str2) > 0) // ...
  ```

- string& *append*(const string& s); bzw. string& *append*(const string& s, uint start, uint n);

  Der String s wird an den Zielstring angehängt. Sind *n* und *start* angegeben, werden nur die ersten n Zeichen ab der Position *start* angehängt. Wenn n nicht angegeben ist, werden alle Zeichen des Zielstrings ab *start* angehängt.

- string& *append*(uint anzahl, char c);

  Es werden *anzahl* Zeichen c an den Zielstring angehängt.

  ```
  Str1 = "Ludwigs"; Str2 = "hafen";
  Str1.append(Str2); // ergibt "Ludwigshafen"
  Str1.append(4,'x'); // ergibt "Ludwigshafenxxxx"
  ```

- uint *copy*(char z[], uint n, unit n2 = 0);

  Beginnend mit der Position n2 werden werden höchstens n Zeichen des Zielstrings in das durch z angegebene Zeichen-Array kopiert. *copy()* liefert die Anzahl der kopierten Zeichen zurück.

  int i;
  char z[10];
  string Str;

```
Str1 = "Ludwigshafen";
i = Str1.copy(z,7);   // ergibt z = "Ludwigs", i = 7
i = Str1.copy(z,7,2); // ergibt Str2 = "dw", i = 2
```

- bool *empty*(uint m, char z);

  Funktion wird *true,* wenn der String kein Zeichen enthält.

```
If (Str1.empty()) cout "Leerer String";
```

- string& *insert*(uint pos, const string& s, uint start, unit n);

  Der String s wird an Position *pos* in den Zielstring eingefügt. Die Parameter *start* und *n* können auch weggelassen werden. Sind sie vorhanden, werden ab der Position *start* im String s maximal n Zeichen von s an der Position *pos* im Zielstring eingefügt. (Falls *pos* einen ungültigen Wert darstellt, löst *insert* eine Exception vom Typ *outofrange* aus.)

```
Str1 = "Ludwig"; Str2 = "Rudolf";
Str1.insert(4,Str2,3,2); // ergibt "Ludwolig"
```

- string& *erase*(uint pos, uint n); (bei Borland C++ auch *remove*())

  Im Zielstring werden, beginnend mit der Position *pos*, alle folgenden Zeichen entfernt. Wenn n angegeben ist, werden höchstens n Zeichen des Zielstrings entfernt

```
Str1 = "Ludwig";
Str1.erase(3,2); // ergibt "Ludg"
```

- string& *replace*(uint pos, uint n, const string& s, uint start, uint n2);

  Im Zielstring werden, beginnend mit Position *pos*, höchstens *n* Zeichen entfernt und durch eine Kopie des Strings s ersetzt. Die Parameter *start* und *n2* können weggelassen werden. Sind sie vorhanden, werden die im Zielstring entfernten Zeichen durch die ersten n2 Zeichen ab Postition *start* des Strings s ersetzt.

```
Str1 = "Ludwig"; Str2 = "Rudolf";
Str1.replace(4,2,Str2,3,2); // ergibt "Ludwol"
```

- void *resize*(uint m, char z);

  Ändert die Stringgröße auf m Zeichen. Leerzeichen werden dabei nötigenfalls hinzugefügt oder entfernt. Wenn der optionale Parameter z angegeben ist, werden m Zeichen z am Stringende hinzugefügt.

```
Str1.resize(15,'\t'); // Länge 15, am Ende Tabulatorzeichen hinzufügen
```

- string *substr*(uint pos, uint n) const;

  Ein Substring als Teilkopie von höchstens n Zeichen ab Position *pos* des Zielstrings wird erzeugt. Wenn n nicht angegeben wird, wird der Substring bis zum Ende des Zielstrings kopiert.

```
Str1 = "Ludwigshafen";
Str2 = Str1.substr(4,2); // ergibt "wi"
```

- string& *swap*(string& s);

  Der Inhalt des Strings s wird mit dem Inhalt des Zielstrings vertauscht.

```
Str1 = "Ludwig"; Str2 = "Rudolf";
Str1.insert(4,Str2,3,2); // ergibt "Ludwolig"
```

- uint *find*(const string& s, uint n);

  Die Anfangsposition von s im Zielstring wird zurückgeliefert, falls s im Zielstring enthalten ist. Ist dies nicht der Fall, wird *string::npos* zurückgeliefert. Wenn n angegeben ist, wird erst ab Position n des Zielstrings gesucht.

```
Str1 = "Ludwigshafen"; Str2 = "haf";
cout << Str1.find(Str2); // ergibt 7
```

- uint *find_first_of*(const string& s); bzw. uint *find_last_of*(const string& s);

  Der Argumentstring s wird als *Zeichenmenge* behandelt. Die erste bzw. letzte Position im Zielstring wird zurückgeliefert, in der ein Zeichen aus s auftritt. Wird kein Zeichen gefunden, wird *string::npos* zurückgeliefert.

```
Str1 = "Ludwig"; Str2 = "aeiou";
cout << Str1.find_first_of(Str2); // ergibt 1 für 'u' (erster Vokal)
```

- uint *find_first_not_of*(const string& s); bzw. uint *find_last_not_of*(const string& s);

  Der Argumentstring s wird als *Zeichenmenge* behandelt. Die erste bzw. letzte Position im Zielstring wird zurückgeliefert, in der *kein* Zeichen aus s auftritt. Wird die Suche fehlschlägt, wird *string::npos* zurückgeliefert.

```
Str1 = "Auerhahn"; Str2 = "aeiou";
cout << Str1.find_first_not_of(Str2); // 3 für 'r' (erster Konsonant)
```

- uint *length()* const;

  Ergibt die Länge eines Strings.

```
Str1 = "Ludwig";
cout << Str1.length(); // ergibt 6
```

- const char* *c_str()*

  Obwohl die Klasse *string* eine sehr komfortable Stringverarbeitung erlaubt, kommt man in der Praxis um eine Bearbeitung von Zeichenarrays nicht herum, denn sehr viele ältere C-Bibliotheken und auch Windows-spezifische Funktionen bauen auf diesen „C-Strings“ auf. Daher existieren Funktionen zum Wandeln zwischen den Datenypen *String* und *Zeichenarray* wie z.B. *c_str()*.

  Die Methode *c_str() l*iefert einen Zeiger auf ein nullterminiertes Zeichen-Array zurück, das die gleichen Zeichen enthält wie der String. (Die Typbezeichnung *char** bedeutet, dass der Typ auf die *Anfangsadresse* eines *char*-Arrays zeigt - man kann es sich zunächst so vorstellen wie *char[]*. Der Zeiger-Operator * wird für Typangaben in einem späteren Kapitel genauer erläutert.) Aus Sicherheitsgründen ist das Ergebnisarray *const* deklariert, d.h. es muß einer Variable vom Typ *char** zugewiesen werden, damit es verändert werden kann.

- *char* strcpy(char zielarray[], const char quellarray[])*;

  Kopiert eine Zeichenarray-Konstante in eine Zeichenarry-Variable. Das abschließende Nullzeichen von *quellarray* wird als letztes Zeichen kopiert. Die Funkction *strcopy* muß bei Variablenzuweisungen im Zusammenhang mit *c_str()* verwendet werden, da der Zuweisungsoperator „=“ bei Zeichenarrays nicht definiert ist.

```
string Str1 = "Teststring";
char Str2[11];
// Str2 = Str1.c_str(); Das ist nicht definiert und funktioniert nicht
strcpy(Str2,Str1); // Das führt zum gewünschten Erfolg
```

  Für Zeichenarrays sind ähnliche Bearbeitungsfunktionen wie für Strings definiert. Diese werden in einem späteren Kapitel (*Zeiger*) besprochen.

Grundlegende Aufgabe	Analyse.cpp

Erstellen Sie ein Programm *Analyse.cpp*, welches die Anzahl der Vokale in einem String ermittelt (Relative Häufigkeit = Absolute Häufigkeit/Stringlänge).

```
Zu analysierender String: Untersuchung der Vokale in einer Textzeile.

Vokal:  Absolute Haeufigkeiten:  Relative Haeufigkeiten:
-------------------------------------------------------
    A              1                   0.0625
    E              8                      0.5
    I              3                    0.188
    O              1                   0.0625
    U              3                    0.188
Programmende Analyse1.
```

Grundlegende Aufgabe	String2.cpp

Schreiben Sie ein Programm *String2.cpp*, welches das Zugreifen, Vergleichen, Suchen, Entfernen, Einfügen, Umformen und Verschieben von Teilstrings zu einem Gesamtstring demonstriert. Benutzen Sie als Vorlage das Programmfragment *String2.fra.*

```
Gesamtstring eingeben: Ludwigshafen
Teilstring eingeben: wig
--------------------------------------------------
0  Beenden
1  Teilstring im Gesamtstring suchen
2  Teilstring aus dem Gesamtstring entfernen
3  Teilstring in den Gesamtstring einfuegen
4  Zeichenfolge des Teilstrings im Gesamtstring umkehren
5  Vorkommen des Teilstrings zaehlen
6  Auf ein Zeichen des Strings direkt zugreifen
7  Beide Strings vergleichen
--------------------------------------------------
Wahl? 1
"wig" beginnt ab Position 3.
Wahl? 2
Gesamtstring bisher: Ludwigshafen
Gesamtstring jetzt:  Ludshafen
Wahl? 3
An welcher Postion einfuegen? 3
Einzufuegender String? wig
Gesamtstring bisher: Ludshafen
Gesamtstring jetzt:  Ludwigshafen
Wahl? 4
Gesamtstring bisher: Ludwigshafen
Gesamtstring ohne Teilstring: Ludshafen
Gesamtstring jetzt: Ludgiwshafen
Wahl? 4
Gesamtstring bisher: Ludgiwshafen
Gesamtstring ohne Teilstring: Ludshafen
Gesamtstring jetzt: Ludwigshafen
Wahl? 5
"wig" ist 1 mal enthalten.
Wahl? 6
Welche Stelle 0 - 11 lesen? 6
Gespeichert ist: s
Wahl? 7
Gesamtstring kommt alphabetisch vorher
Wahl? 0
Programmende String2.
```

2.8 Dateien (Streams)

Lerninhalte

❶ Dateitypen
❷ Datei-Zugriffsarten
❸ Grundstruktur der Dateiverarbeitung
❹ Textdateien
❺ Binärdateien
❻ Methoden zur Dateiverarbeitung
❼ Speichern von Datensätzen mit dynamischen Komponenten

Lerninhalte

Alle Daten, die wir bisher bei der Ausführung unserer Beispielprogramme in den Rechner eingegeben haben, sind nach dem Abschalten des Rechners verloren - die Daten befanden sich während des Programmablaufs lediglich im flüchtigen Hauptspeicher des Computers. Zum dauerhaften Erhalt müssen Datenstrukturen auf einem *nichtflüchtigen* Speicher gesichert werden (Festplatte, Diskette) - damit werden die Daten zu einer *Datei* (engl. *File*).

L1 Dateitypen

Wir unterscheiden *Programmdateien* (engl. *program files*) und *Datendateien* (engl. *data files*). Unter Programmdateien versteht man eine gespeicherte Anzahl von *Maschinenbefehlen*, die als Programm im Rechner ablaufen können. Unter *Datendateien* versteht man eine geordnetete Sequenz von Datenkomponenten (*Datensätzen)*, die den folgenden Bedingungen genügen:

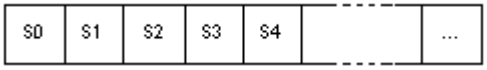
- Datensätze von Dateien können beliebige Typen haben (vom einfachen *char* oder *int* bis zur komplexen *class*). In C++ kann man beliebige Datenströme mit einer Abfolge unterschiedlicher Datentypen auf dieselbe Datei schreiben - in anderen Programmiersprachen (z.B. *Pascal)* kann man für die Datensätze explizit einen bestimmten Typ festlegen.
- Die Anzahl der Datensätze einer Datei kann während des Programmablaufs dynamisch verändert werden.
- Der Dateizugriff erfolgt meist *satzweise*, d.h. bei jedem Dateizugriff wird ein ganzer Datensatz eines bestimmten Typs in den Speicher geschrieben oder vom Speicher gelesen. Möchte man die Daten einer Datei nicht direkt in komplexe Datenstrukturen (*struct* oder *class*) einordnen, kann man auch *byteweise* oder *blockweise* (d.h. als *char*-Array mit konstanter Bytezahl) lesen oder schreiben.
- Eine besondere Dateiform stellt die *Textdatei* dar. Hier besteht ein einzelner Datensatz aus einer Zeile lesbaren Textes (z.B. repräsentiert durch den Datentyp *string*). Die einzelnen Datensätze einer Textdatei haben *unterschiedliche* Längen. Jede Zeile wird durch einen *Zeilenvorschub* (*line feed* - LF - ASCII-10) abgeschlossen. Bei PC-Betriebssystemen (DOS, Windows, OS/2) wird jedem Zeilenvorschub grundsätzlich noch das Zeichen für „Wagenrücklauf“ vorangestellt (*carriage return* - CR - ASCII-13). An der *letzen* Stelle einer Textdatei steht bei den meisten Betriebssystemen das Zeichen Ctrl-Z, auch EOF genannt (*end of File*, ASCII-26).
- Wenn die einzelnen Datensätze keine Textzeilen darstellen, die durch *CR/LF* abgeschlossen werden, redet man von einer *Binärdatei*. Wenn man zusätzlich dafür sorgt, dass jeder Datensatz einer Binärdatei vom gleichen Typ ist und daher auch gleich viel Speicherplatz belegt, bezeichnet man diesen Spezialfall als *typisierte Datei*.

Bislang erzeugten wir ständig eine bestimmte Art von Daten- als auch von Programmdateien, ohne dies explizit zu erwähnen: Die *Quellode-Dateien* (*.cpp) unserer Beispielprogramme sind Textdateien, die mittels unserer Programmentwicklungsumgebung gespeichert werden können. Die kompilierten, ablauffähigen Beispielprogramme (*.exe) sind Binärdateien (man kann sie auch als typisierte Dateien mit dem Datensatztyp *int* oder *char* auffassen).

L2 Datei-Zugriffsarten

Grundsätzlich kann man sich die einzelnen Bytes im Datenstrom einer Datei immer von 0 an fortlaufend durchnummeriert vorstellen. Die Nummer eines Dateibytes wird von einer ganzzahligen Programmvariable verwaltet, die man *Dateizeiger (file pointer)* nennt. Auf einen einzelnen Datensatz kann man auf zweierlei Arten zugreifen:

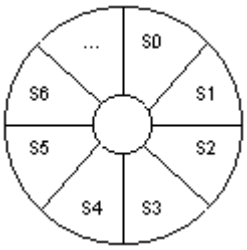
1. **Sequentieller Dateizugriff** (sequential access)



Hier kann man sich die Datensätze wie auf einem langen Magnetband hintereinander angeordnet vorstellen.

Der Zugriff auf Datensatz Nr. n ist nur möglich, wenn vorher die Datensätze 0 bis n-1 der Reihe nach durchlaufen wurden (so wie das 3. Musikstück auf einer Tonbandkassette auch nur abgespielt werden kann, wenn vorher die ersten beiden Stücke durchgespult wurden.). Ist man beim letzten Datensatz angelangt und möchte danach auf einen vorherigen Satz zugreifen, so muß die Datei wieder vom nullten Datensatz an durchsucht werden.

2. **Direkter Dateizugriff** (wahlfreier Zugriff, random access)



Beim wahlfreien Dateizugriff kann der Dateizeiger direkt auf den Anfang eines beliebigen Datensatzes positioniert werden. Dann kann der betreffende Satz vom nichtflüchtigen Massenspeicher in den flüchtigen Hauptspeicher gelesen bzw. in umgekehrter Richtung geschrieben werden. Einen direkten Dateizugriff kann man sich vorstellen wie das Abspielen eines bestimmten Musikstückes von einer Schallplatte, wo ja auch die Nadel direkt an einer beliebigen Stelle aufgesetzt werden kann.

Ein direkter Dateizugriff ist nur bei Dateien mit *konstanter Datensatzlänge* einfach zu realisieren. Nur hier kann das ablaufende Programm sofort und ohne große Rechenoperationen die korrekte Adresse eines beliebigen Datensatzes ermitteln. Auf Textdateien kann man also nur *sequentiell* zugreifen, da sie keine feste Datensatzlänge haben.

Man kann den Dateizugriff auch als Bearbeitung eines *Datenstroms* auffassen, der von einer *Quelle* (z.B. dem Hauptspeicher) zu einem *Ziel* (z.B. der Festplatte) und zurück fließt. Daher heißen Dateien in C++ auch *streams.* Eine besondere Art eines *stream* stellt die Datenverarbeitung über die *Konsole* dar. Als *Konsolen-Streams* bezeichnet man in der EDV die Kombination aus Dateneingabe über die Tastatur und Datenausgabe über den Bildschirm eines Computersystems. Eine Konsole besteht für C++ aus den folgenden drei Objekten:

- *cin*: Standardeingabe (Tastatur - nur zum Lesen)
- *cout*: Standardausgabe (Bildschirm - nur zum Schreiben)
- *cerr*: Standardfehlerausgabe (meist auch der Bildschirm - für Fehlermeldungen)

Grundstruktur der Dateiverarbeitung

In C++ werden Streams in der Header-Datei *fstream* deklariert. Durch *#include* <*fstream*> werden die nachstehenden Datentypen zugänglich.

Eine Datei wird als Stream folgendermaßen deklariert:

fstream Dateibezeichner;

Aus einer solchen Datei kann gelesen werden und man kann auch in sie schreiben. Soll aus einer Datei lediglich *gelesen* werden, benutzt man statt *fstream* („file stream“) die Klassenbezeichnung *ifstream* („input file stream“), wird in die Datei nur geschrieben, verwendet man *ofstream* („output file stream“).

Der Zugriff auf eine Datei erfolgt in drei Schritten:

1. Datei *öffnen*: Verbinden der Dateivariablen von C++ mit einem Dateinamen des Betriebssystems, Herstellen der Verbindungskanäle vom Hauptspeicher zum Massenspeicher, inklusive der Einrichtung von internen Datenpuffern im Hauptspeicher. Die Datei wird zum *Lesen*, zum *Schreiben*, oder für *beide* Zugriffsoperationen geöffnet. Nach dem Öffnen einer Datei zeigt der Dateizeiger immer auf das Dateibyte Nr. 0.
2. Datensätze *lesen* oder *schreiben*. Zu schreibende Datensätze werde zunächst in den internen Puffer geschrieben und, wenn dieser voll ist, automatisch auf den Massenspeicher transferiert.
3. Datei *schließen:* Leeren der internen Dateipuffer, trennen der Verbindung zwischen C++ und dem Dateisystem, Freigabe der internen Dateipuffer.

L4 Textdateien

Textdateien werden mit folgenden Methoden bearbeitet.

1. **Datei öffnen:**

   Das Öffnen einer Datei erfolgt mit der Methode *open*:

```
string Dateiname="C:\\TEST\\VERSUCH.DAT";
fstream Dat;

Dat.open(Dateiname.c_str(),OpenModus);
```

   Die Methode *open* erwartet den Dateinamen als *C-String*. Demzufolge muß der C++-String *Dateiname* mit *c_str()* konvertiert werden. Der Parameter *OpenModus* bestimmt die Art und Weise, wie die Datei geöffnet wird. *OpenModus* kann verschiedene Werte annehmen, die auch mit binärem ODER (|) kombiniert werden können, um mehrere Eigenschaften gleichzeitig zu erreichen. Der Parameter ist in der Klasse *ios* definiert und kann u.a. folgende Werte annehmen:

   `in`: Öffnen für Eingabe (Voreinstellung für ifstream)

   `out`: Öffnen für Ausgabe (Voreinstellung für ofstream)

   `app`: Beim Schreiben neue Daten am Dateiende anhängen (Append)

   `trunc`: Datei löschen, falls beim Öffnen schon existent (Truncate)

   `binary`: Öffen im Binärmodus (sonst: Textmodus)

   Beispiel:

```
Dat.open(Dateiname.c_str(),ios::in | ios::out);
/* Datei wird zum Lesen und Schreiben im Textmodus geöffnet. */
```

   (Die Funktion *open()* kann noch mit einen dritten Parameter *int = filebuf::openprot* aufgerufen werden. Dieses Argument gibt die Zugriffsmethode für den gemeinsamen Dateizugriff in Computernetzen an. Ohne Angabe des Parameters kann nur ein einziger Benutzer die Datei öffnen. Ist sie dann offen, bleibt sie für andere Benutzer gesperrt.)

2. **a) Aus Textdatei lesen**

   Standardmäßig, wenn kein Modus *binary* angegeben ist, wird in C++ eine Datei als *Textdatei* geöffnet. Dann kann man mit dem Stream-Eingabeoperator (>>) Zeichen aus der Datei einlesen. Dies hat jedoch zwei Nachteile. Die Eingabe wird beim ersten gefundenen Leerzeichen beendet, außerdem ist es möglich, dass beim Lesen in einen C-String unkontrolliert zuviele Zeichen eingelesen werden. Will man die Zeichen bis zu einem eindeutigen String-Endezeichen einlesen (in der Regel ist das das Zeilenende <CR><LF>, ASCII-13/ASCII-10), dann benutzt man am besten die Methode *getline()*. Mit *get()* liest man ein einzelnes Zeichen ein. Bei Textdateien werden alle eingelesenen Zeichen als Folge von *chars* interpretiert.

```
char Zeile[LAENGE];  // C-String zur Aufnahme einer Textdatei-Zeile
char z;              // char zur Aufnahme eines einzelnen Dateizeichens
Dat.getline(Zeile,LAENGE); // Dateizeile aus Textdatei lesen
/* ... */
Dat.get(c);          // Einzelnes Zeichen aus Datei auslesen
c=Dat.get();         // ebenso: einzelnes Zeichen aus Datei auslesen
```

b) In Textdatei schreiben

Mit dem Stream-Ausgabeoperator (<<) kann man Zeichen, Strings und Zahlen in eine Textdatei schreiben. Dabei sind alle Formatieroptionen möglich, die man bei *cout* << zur Bildschirmausgabe anwenden kann. Zahlen werden als einzelne Ziffern in die Datei geschrieben, die ASCII-codiert sind, also so, wie sie mit *cout* << auf dem Bildschirm ausgegeben würden.

```
String Textzeile="Wir testen eine Textzeile";
int a=10;

Dat << Textzeile;     // Schreiben einer Zeichenkette in eine Textdatei
/* ... */
Dat << a;             // Schreiben einer Zahl in eine Textdatei
```

3. **Datei schließen**

Eine geöffnete Datei wird mit der Methode *close()* geschlossen. Das Schließen einer Datei erfolgt automatisch, sobald der Gültigkeitsbereich der Dateideklaration verlassen wird, sobald also die Dateivariable nicht mehr verfügbar ist.

```
Dat.close(); // Datei wird geschlossen.
```

Eventuelle Fehler, die beim Öffnen oder Lesen einer Datei immer auftreten können (z.B. „Datei nicht vorhanden“ oder „Datei schreibgeschützt“), werden durch eine Abfrage der boolschen Interpretation des „Rückgabewerts“ des Dateiobjekts oder einer Elementfunktion erkannt. Wird *false* zurückgegeben, dann wurde ein Fehler erkannt.

```
fstream quelldatei;
char c,s[80];
quelldatei.open("text.txt",ios::in);
zieldatei.open("text2.txt",ios::out);
if (!quelldatei) cout << "Datei kann nicht geöffnet werden";
// ...
while (quelldatei.get(c)) zieldatei.put(c);
// solange true, sind noch Zeichen zum Lesen da
// ...
if (!cin.get(c)) // ... dann ist ein Fehler beim Lesen passiert
// ...
while (quelldatei.getline(s,80)) zieldatei << s;
// solange true, sind noch Zeilen zum Lesen da
```

Beispiel Textdatei lesen und schreiben

Das folgende Programm Textdat.cpp fordert zur Eingabe von Dateinamen für zwei Dateien auf. Die erste Datei existiert bereits unter dem Namen *Textdat.txt*. Aus ihr werden alle Textzeilen sequentiell gelesen. Die gelesenen Zeilen werden in eine zweite Datei geschrieben (Namensvorschlag: *Textziel.txt)* und gleichzeitig auf dem Bildschirm ausgegeben.

```
// PROGRAMM Textdat.cpp
// Aus Textdatei lesen und in andere Textdatei schreiben
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

void ZeilenVerarbeiten(fstream& Dein, fstream& Daus)
{
  char Zeile[100]; // Puffer für eine Zeile
  bool eof; // Zur Erkennung des Dateiendes

  do
  {
    eof=!Dein.getline(Zeile,100); // Wurde versucht übers Dateiende zu lesen?
    if (eof && *Zeile=='\0') break;
    Daus << Zeile << endl; // Zeile wieder ausgeben
    cout << "Gelesen: " << Zeile << endl;
  } while (!eof);
}

void OeffneDatein(fstream& f)
{
  string Dname;
  bool fehler;
  do
  {
    cout << "Name der Quelldatei: ";
    getline(cin,Dname,'\n');
    f.open(Dname.c_str(),ios::in);
    fehler = !f;
    if (fehler) cout << "Fehler beim Oeffnen von " << Dname << endl;
  } while (fehler);
}

void OeffneDataus(fstream& f)
{
  string DausName;
  bool fehler;
  do
  {
    cout << "Name der Zieldatei: ";
    getline(cin,DausName,'\n');
    f.open(DausName.c_str(),ios::out);
    fehler = !f;
    if (fehler)
       cout << "Fehler beim Oeffnen von " << DausName << endl;
  } while (fehler);
}

void main()
{
  fstream Datein, Dataus;
  OeffneDatein(Datein);
  OeffneDataus(Dataus);
  ZeilenVerarbeiten(Datein,Dataus);
  Datein.close();
  Dataus.close();
}
```

```
Name der Quelldatei: textdat.txt
Name der Zieldatei: textziel.txt
Gelesen: Das ist Zeile 1
Gelesen: Das ist Zeile 2
Gelesen: Das ist Zeile 3 (die letzte)
```

a) Debuggen Sie das Programm und verfolgen Sie die Veränderung der Werte aller Variablen, besonders in der Funktion *ZeilenVerarbeiten()*.

b) Die letzte Zeile der Quelldatei Textdat.txt ist *nicht* mit CR/LF abgeschlossen. Kopieren Sie *Textdat.txt* nach *Textdat2.txt* und hängen Sie mit einem Editor den Zeilenvorschub in der letzten Zeile an. Debuggen Sie nun das Programm mit *Textdat2.cpp* wieder wie bei a) und protokollieren Sie die Änderungen.

c) Kopieren Sie die Datei *Textdat.cpp* nach *Textdat2.cpp* und ersetzen Sie die folgende Zeile *eof= !Dein.getline(Zeile,100)* in der Funktion *ZeilenVerarbeiten()* durch z= *Dein.get()*. Ändern Sie das Programm weiterhin so ab, dass es alle Zeichen der Quelldatei bis zum Dateiende liest und dann endet. Protokollieren Sie alle Veränderungen zu b).

Grundlegende Aufgabe `Textdat3.cpp`

Schreiben Sie das Programm *Textdat.cpp* zu *Textdat3.cpp* um, indem Sie die gesamte Funktionalität des Programmes in einer Klasse *DateiKopierer* kapseln

Weiterführende Aufgabe `Protokoll.cpp`

Entwickeln Sie ein Programm *Protokoll.cpp*, welches zur Eingabe von Textzeilen auffordert. Durch Eingabe eines Leerstrings wird der Eingabemodus beendet. Danach können alle eingegebenen Textzeilen aus einer Protokolldatei ausgelesen und am Bildschirm angezeigt werden. Das Programm soll die folgenden Menüpunkte beinhalten:

- *Protokolldatei:* Namen der Protokolldatei festlegen.
- *Texteingabe:* Textzeilen eingeben.
- *Protokollieren:* Inhalt der Textdatei auf dem Bildschirm ausgeben.
- *Einzelne Zeile:* Nach Eingabe der Zeilennummer eine einzelne gespeicherte Zeile anzeigen.

Weiterführende Aufgabe `Zei_zahl.cpp`

Schreiben Sie ein Programm *Zei_zahl.cpp*, welches eine beliebige Textdatei zeichenweise liest und mitzählt, wie oft ein bestimmtes Zeichen in der Datei vorkommt:

```
Name der Textdatei: TXTDAT.TXT
Zeichen, das gezählt werden soll: e
Anzahl von "e" in Datei TXTDAT.TXT: 32
Programmende.
```

L5 Binärdateien

Binärdateien werden mit folgenden Methoden bearbeitet.

1. **Datei öffnen:**

   Das Öffnen einer Datei erfolgt wie bei Textdateien mit der Methode *open*:

```
string Dateiname="C:\\TEST\\VERSUCH.DAT";
fstream Dat;

Dat.open(Dateiname.c_str(),ios::in | ios::out | ios::binary);
/* Datei wird zum Lesen und Schreiben im Binärmodus geöffnet. */
```

   Dabei muss im Modus natürlich *ios::binary* vorkommen.

2. **Binärdateien lesen und schreiben**

   Zum Einlesen von Datensätzen aus einer Binärdatei verwendet man die Methode *read()*, zur Datensatzausgabe benutzt man die Methode *write()*. Damit kann man einzelne Bytes (C++-Datentyp *char)* bzw. ganze Bytefolgen (z.B. C-Zeichenarrays oder Strukturen) beliebiger Länge in Dateien speichern und wieder in den Hauptspeicher lesen. Nach jedem *read()*- oder *write()*-Aufruf zeigt der Dateizeiger auf den Anfang des nächsten Datensatzes.

   Will man beliebige Datentypen oder komplexe *Objektklassen* in Dateien speichern oder zurücklesen, muß man die Methoden *read()* und *write()* auf ganze *Byteströme* anwenden (*char*-Arrays). Eine Variable **x** eines beliebigen Datentyps wird als Bytestrom mit der Länge **sizeof(x)** in Dateien verarbeitet, indem man sie als *Zeichenarray* **(char*)&x** anspricht.

```
IrgendeinTyp x; // struct, class oder irgendetwas anderes
fstream f;

f.open("C:\TEST.DAT",ios::in|ios::out|ios.binary);
/* Binärdatei zum Lesen und Schreiben öffnen */

f.read( (char*)&x,sizeof(x) ); // Syntax zum Lesen des Satzes
// ...
f.write (char*)&x,sizeof(x) ); // ebenso zum Schreiben
// ...
f.seekp(5*sizeof(x));
// Dateizeiger auf Datensatz Nr. 5 für Direktzugriff positionieren
// ...
f.close();
```

Ist **x** schon selbst ein Array oder ein Zeiger, muß man aus syntaktischen Gründen die Aufrufform **(char*)x** wählen, d.h. dann entfällt das Zeichen **&**.

```
IrgendeinTyp *x; // oder x[]
// ...
f.read( (char*)x,sizeof(*x) ); // Syntax zum Lesen des Satzes
// ...
f.write (char*)x,sizeof(*x) ); // ebenso zum Schreiben
// ...
f.seekp(5*sizeof(*x));
// Dateizeiger auf Datensatz Nr. 5 für Direktzugriff positionieren
```

3. **Datei schließen**

   Eine geöffnete Binärdatei wird ebenso wie eine Textdatei mit der Methode *close()* geschlossen. Auch das Schließen einer Binärdatei erfolgt automatisch, sobald der Gültigkeitsbereich der Dateideklaration verlassen wird.

```
Dat.close(); // Datei wird geschlossen.
```

Eventuelle Fehler, die beim Öffnen oder Lesen einer Datei immer auftreten können (z.B. „Datei nicht vorhanden“ oder „Datei schreibgeschützt“), werden durch eine Abfrage der boolschen Interpretation des „Rückgabewerts“ des Dateiobjekts oder einer Elementfunktion erkannt. Wird *false* zurückgegeben, dann wurde ein Fehler erkannt.

```
fstream quelldatei;
struct TX
{
  int a;
  char b[20];
};
TX x;
quelldatei.open("datei.dat",ios::in|ios::binary);
zieldatei.open("datei2.dat",ios::out|ios::binary);
if (!quelldatei) cout << "Datei kann nicht geöffnet werden";
// ...
while (quelldatei.read((char*)&x,sizeof(x)))
  zieldatei.write((char*)&x,sizeof(x));
// solange true, sind noch Datensaetze zum Lesen da
```

Beispiel | Binärdatei lesen und schreiben

Das Programm Kunden.cpp verwaltet Kundendaten in einer typisierten Binärdatei. Der Datensatz für einen einzelnen Kunden wird in einer Struktur (*struct*) gespeichert, welche die Kundennummer, den Namen und den Umsatz enthält. Alle Datensätze der Datei haben daher diesen Typ:

```
struct Kundensatz {
  int Nummer;
  char Name[20]; // char[] statt string, wegen konstantem Speicherbedarf
  float Umsatz;
}
```

Da die Datei typisiert ist, kann man auf alle Datensätze sowohl *sequentiell* als auch *direkt* zugreifen. Das Programm *Kunden.cpp* demonstriert die Zugriffsarten in den folgenden Funktionen:

- *Schreiben():* Die Datei wird für Ausgabe geöffnet. Der Dateizeiger wird beim Öffnen hinter den letzten Datensatz positioniert. Dann werden die neuen Datensätze *sequentiell* an das Dateiende angehängt.
- *Auflisten():* Die Datei wird für Eingabe geöffnet. Der Dateizeiger steht nach dem Öffnen auf dem Dateianfang. Dann werden die einzelnen Datensätze *sequentiell* ausgelesen und auf dem Bildschirm angezeigt.
- *Suchen():* Nach Eingabe eines Suchstrings wird die Datei für Eingabe geöffnet. Vom Dateianfang an werden die einzelnen Datensätze *sequentiell* nach dem Suchstring durchsucht. Die Suche wird beendet, wenn der gesuchte Datensatz gefunden wurde oder der letzte Datensatz gelesen wurde, ohne den Suchstring zu finden.
- *Aendern():* Die Datei wird für Ein- und Ausgabe gleichzeitig geöffnet. Dann wird nach einer Datensatznummer gefragt und der Dateizeiger wird mit der Methode *fseek()* (siehe weiter unten) *direkt* auf den Anfang dieses Datensatzes positioniert. Der bestehende Datensatz wird gelesen, verändert und danach wieder an gleicher Position in die Datei zurückgeschrieben. Hier wird also mit *Direktzugriff* gearbeitet.

```
Name der Datei (z.B. Kunden.dat): kunden.dat
Datei existiert schon, neu anlegen (j/n)? j

Zur Menueanzeige Return druecken!

=====================================================
0   Ende:          Programm beenden
1   Schreiben:     Neue Datensaetze an Datei anhaengen
2   Auflisten:     Alle Datensaetze seriell lesen
3   Suchen:        Datensatz suchen und lesen
4   Aendern:       Bestimmten Datensatz aendern
5   Neu:           Neue Datei bearbeiten
=====================================================
Bitte waehlen: 1

Kundennummer (0=Ende): 1
Name des Kunden:       Adam
Umsatz in DM:          11.11

Kundennummer (0=Ende): 2
Name des Kunden:       Berta
Umsatz in DM:          22.22

Kundennummer (0=Ende): 3
Name des Kunden:       Cäsar
Umsatz in DM:          33.33

Kundennummer (0=Ende): 4
Name des Kunden:       Dorothea
Umsatz in DM:          44.44

Kundennummer (0=Ende): 0
...
Bitte waehlen: 2

Satznummer  Kundennummer  Name                    Umsatz
     0:            1      Adam                     11.11
     1:            2      Berta                    22.22
     2:            3      Cäsar                    33.33
     3:            4      Dorothea                 44.44
...
Bitte waehlen: 3
Kundenname als Suchbegriff: Cäsar
Kundennummer:  3
Name:          Cäsar
Umsatz bisher: 33.3
...
Bitte waehlen: 4
Satznummer (0,1,2,...): 2
Kundennummer bisher :  3 - neu: 33
Name des Kundenbisher: Cäsar - neu: Balthasar
Umsatz bisher:         33.3 - neu: 333.3
Satz wurde geaendert.
...
Bitte waehlen: 2

Satznummer  Kundennummer  Name                    Umsatz
     0:            1      Adam                     11.11
     1:            2      Berta                    22.22
     2:           33      Balthasar                333.3
     3:            4      Dorothea                 44.44
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
#include <stdlib.h> // wegen exit()

struct TKundsatz
{
  int Nummer;
  char Name[20]; // char[] statt string, wegen konstantem Speicherbedarf
  float Umsatz;
};
TKundsatz Kundensatz;

void OeffneDatei(const string Dname, fstream& f, const ios::open_mode m)
{
  f.open(Dname.c_str(),m | ios::binary);
  if (!f)
  {
    cout << "Programmabbruch: Fehler beim Öffnen der Datei" << Dname;
    exit(1);
  }
}

void OeffneDateiNeu(string& Dname, fstream& f)
{
  char taste;

  cout << "Name der Datei (z.B. Kunden.dat): ";
  getline(cin,Dname,'\n');
  f.open(Dname.c_str(),ios::in|ios::binary);
  if (f)
  {
    f.close();
    do
    {
     cout << "Datei existiert schon, neu anlegen (j/n)? "; cin >> taste;
     taste = toupper(taste);
    } while (taste!='J' && taste!='N');
    cin.ignore(); // Zeilenvorschub aus Eingabestream entfernen
    if (taste=='J')
    {
      OeffneDatei(Dname,f,ios::out|ios::trunc);
      f.close();
    }
  }
  else f.close();
}

void SchliesseDatei(const string& Dname,fstream& f)
{
  f.close();
  cout << "Datei " << Dname << " geschlossen.\n";
}

void SatzAnzeigen(TKundsatz k)
{
  cout.precision(3);
  cout << "Kundennummer:  " << k.Nummer << endl;
  cout << "Name:          " << k.Name << endl;
  cout << "Umsatz bisher: " << k.Umsatz << endl;
}
```

```
void Schreiben(string Dname, fstream& f)
{
  TKundsatz k;
  string NameStr;

  OeffneDatei(Dname,f,ios::out|ios::app);
  while (true)
  {
    cout << "\nKundennummer (0=Ende): "; cin >> k.Nummer;
    cin.ignore();
    if (!k.Nummer) break;
    cout << "Name des Kunden:       ";
    getline(cin,NameStr,'\n'); // Sichere Stringeingabe
    strcpy(k.Name,NameStr.substr(0,19).c_str()); // Kürzen auf max. 19 Stellen
    cout << "Umsatz in DM:          "; cin >> k.Umsatz;
    f.write((char*)&k,sizeof(k));
  }
  f.close();
}

void Auflisten(const string& Dname, fstream& f)
{
  int SatzNr;
  TKundsatz k;

  OeffneDatei(Dname,f,ios::in);
  SatzNr = 0;
  cout << "\nSatznummer  Kundennummer  Name                    Umsatz\n";
  while (f.read((char*)&k,sizeof(k)))
  {
    cout << setw(6) << SatzNr++ << ":      "
         << setw(7) << k.Nummer << "      " << setw(19);
    cout.setf(ios::left,ios::adjustfield);
    cout << k.Name << "  " << setw(8);
    cout.setf(ios::right,ios::adjustfield);
    cout << k.Umsatz << endl;
  }
  cin.ignore();
  f.close();
}

void Suchen(const string& Dname, fstream& f)
{
  string NameSuch;
  bool fund = false;
  TKundsatz k;

  OeffneDatei(Dname,f,ios::in);
  cin.ignore();
  cout << "Kundenname als Suchbegriff: ";
  getline(cin,NameSuch,'\n');
  while (!fund && f.read((char*)&k,sizeof(k)))
    if (NameSuch==k.Name) fund=true;
  if (fund)
  {
    SatzAnzeigen(k);
  }
  else
  {
    cout << "Satz" << NameSuch << " nicht vorhanden.\n";
    NameSuch = "0";
  }
  f.close();
}
```

```
void Aendern(const string& Dname, fstream& f)
{
  int SatzNr;
  TKundsatz k;
  string NameStr;

  OeffneDatei(Dname,f,ios::in|ios::out);
  cout << "Satznummer (0,1,2,...): "; cin >> SatzNr;
  if (!f.seekp(SatzNr*sizeof(k)))
  {
    cout << "Datensatz Nr." << SatzNr << "existiert nicht.\n";
    return;
  }
  f.read((char*)&k,sizeof(k));
  cout.precision(3);
  cout << "Kundennummer bisher :  " << k.Nummer << " - neu: ";
  cin >> k.Nummer; cin.ignore();
  cout << "Name des Kundenbisher: " << k.Name <<   " - neu: ";
  getline(cin,NameStr,'\n');
  strcpy(k.Name,NameStr.substr(0,19).c_str());
  cout << "Umsatz bisher:         " << k.Umsatz << " - neu: ";
  cin >> k.Umsatz; cin.ignore();
  f.seekp(SatzNr*sizeof(k));
  if (f.write((char*)&k,sizeof(k)))
    cout << "Satz wurde geaendert.\n";
  else
    cout << "Satz konnte nicht geändert werden.\n";
  f.close();
}

void Menue(string& Dname, fstream& f)
{
  char Wahl;

  do
  {
    cout << "\nZur Menueanzeige Return druecken! "; cin.get();
    cout << "\n=====================================================\n";
    cout << "Datei in Bearbeitung: " << Dname << endl;
    cout << "0   Ende:          Programm beenden\n";
    cout << "1   Schreiben:     Neue Datensaetze an Datei anhaengen\n";
    cout << "2   Auflisten:     Alle Datensaetze seriell lesen\n";
    cout << "3   Suchen:        Datensatz suchen und lesen\n";
    cout << "4   Aendern:       Bestimmten Datensatz aendern\n";
    cout << "5   Neu:           Neue Datei bearbeiten\n";
    cout << "=====================================================\n";
    cout << "Bitte waehlen: "; Wahl = cin.get();
    cin.ignore();
    switch (Wahl)
    {
      case '1': Schreiben(Dname,f); break;
      case '2': Auflisten(Dname,f); break;
      case '3': Suchen(Dname,f); break;
      case '4': Aendern(Dname,f); break;
      case '5': SchliesseDatei(Dname,f); OeffneDateiNeu(Dname,f); break;
    }
  } while (Wahl!='0');
}
```

```
void main()
{
  fstream KundenDat;
  string Dateiname;

  OeffneDateiNeu(Dateiname,KundenDat);
  Menue(Dateiname,KundenDat);
  cout << "Programmende Kunden.\n";
  getch();
}
```

Grundlegende Aufgabe	Kunden2.cpp

Schreiben Sie das Programm *Kunden.cpp* zu *Kunden2.cpp* um, indem Sie die gesamte Funktionalität des Programmes in einer Klasse *Kundendat* kapseln.

Grundlegende Aufgabe	Vergleich Textdat3 und Kunden2

Welche Programmfassungen von `Textdat3` und `Kunden2` erscheinen Ihnen strukturierter und einfacher? Begründen Sie die Antwort und versuchen Sie in Zukunft, alle Ihre Programmentwürfe nach der einfachsten Variante zu strukturieren.

Grundlegende Aufgabe	Kopieren.cpp

Schreiben Sie ein Programm *Kopieren.cpp*, mit dem eine beliebige Datei vollständig in eine neue Datei kopiert werden kann. Dabei soll die Datei in Blöcken von 512 Bytes übertragen werden.

Weiterführende Aufgabe	Kunden3.cpp

Erweitern Sie *Kunden2.cpp* zu *Kunden3.cpp*, indem Sie die folgenden Menüpunkte hinzufügen:

- *LogischLoeschen():* Ein Satz kann nach Datensatznummer zum logischen Löschen ausgewählt werden. Dann wird der Umsatz negativ gesetzt und der Datensatz wird beim *Auflisten* nicht mehr angezeigt.
- *Reorganisieren():* Bei allen Datensätzen werden die logischen Löschmarkierungen entfernt.
- *PhysikalischLoeschen():* Alle Sätze, die nicht logisch gelöscht sind, in einen *vector* in den Hauptspeicher lesen. Danach die Datei löschen und alle Sätze vom Hauptspeicher wieder in eine neue Datei gleichen Namens schreiben.

Weiterführende Aufgabe	Artikel.cpp

Entwickeln Sie ein Programm *Artikel.cpp*, das die folgende typisierte Datei verwaltet:

```
struct TArtikel
{
  short int Nummer;
  char      Bezeichnung[21];
  int       Bestand;
  float     Stueckpreis;
  int       LieferNr;
  bool      Geloescht;
};
```

Das Auswahlmenü soll die folgenden Methoden bereitstellen:

- *Dateiname:* Namen für aktive Datei eingeben
- *Anlegen:* Neue Datei anlegen.
- *Abfragen:* Einzelnen Datensatz nach Eingabe der Nummer suchen
- *Auflisten:* Alle Datensätze der Datei auf dem Bildschirm ausgeben.
- *Anhaengen:* Neue Datensätze am Dateiende anfügen.
- *Logisch löschen:* Markieren durch Datensatzfeld *Geloescht*.
- *Physikalisch löschen:* wie bei `Kunden3` realisieren.

L6 Methoden zur Dateiverarbeitung

- *void eatwhite();* entfernt mehrfach aufeinanderfolgende Whitespace-Zeichen (Leerzeichen, Tabulator, usw.)
- *int gcount();* liefert die Anzahl der zuletzt ohne Formatierung aus dem Stream geholten Zeichen zurück. Unformatiertes Holen von Zeichen findet innerhalb der Elementfunktionen *get, getline* und *read* statt.
- *int get();* entnimmt das nächste Zeichen (oder EOF).
- *istream& get(char&);* entnimmt ein einziges Zeichen und überträgt es in den vorgegebenen String.
- *istream& get(char*, int len, char = '\n');* entnimmt solange Zeichen und speichert sie über den vorgegebenen Parameter (char*) ab, bis sie auf ein Begrenzungszeichen (dritter Parameter) oder auf das Dateiende-Zeichen (EOF) stößt, oder aber, bis (len-1) Bytes gelesen worden sind. Dabei wird stets ein abschließendes Nullzeichen in den String geschrieben. Das Begrenzungszeichen wird nicht aus dem Eingabe-Stream entnommen. Die Funktion versagt nur dann, wenn kein Zeichen aus dem Stream entnommen wurde.
- *istream& get(streambuf&, char = '\n');* entnimmt solange Zeichen und speichert sie in dem angegebenen streambuf-Objekt, bis sie auf ein Begrenzungszeichen stößt.
- *istream& getline(char*, int, char = '\n');* entnimmt Zeichen bis zum Begrenzungszeichen, schreibt sie in den Puffer und entfernt das Begrenzungszeichen aus dem Eingabe-Stream (überträgt es aber nicht in den Puffer).
- *istream& ignore(int n = 1, int delim = EOF);* bewirkt, dass bis zu n Zeichen des Eingabe-Streams übersprungen werden. Beim Erreichen des Begrenzerzeichens stoppt die Funktion. Das Begrenzerzeichen wird aus dem Eingabe-Stream entnommen.
- *istream& ipfx(int n = 0);* wird von Eingabefunktionen aufgerufen, bevor diese Zeichen aus einem Eingabe-Stream holen. Funktionen für formatierte Eingaben machen den Aufruf *ipfx(0)*, Funktionen für unformatierte Eingaben rufen dagegen *ipfx(1)* auf.
- *int peek();* liefert das nächste Zeichen zurück, entnimmt es aber nicht.
- *istream& putback(char);* schiebt ein Zeichen in den Stream zurück.
- *istream& read(char*, int);* entnimmt eine vorgegebene Anzahl von Zeichen und schreibt sie in ein Array. Falls ein Fehler auftritt, können Sie die Anzahl der tatsächlich entnommenen Zeichen mit Hilfe der Funktion *gcount()* erfahren.
- *istream& seekg(streamoff offset, seek_dir dir);* positioniert im Eingabe-Stream relativ zur aktuellen Position, um einen Offset von *offset* Bytes weiter. Die Richtung des Offsets wird dabei über dir, definiert durch *enum seek_dir {beg, cur, end}*, vorgegeben. Verwenden Sie *ostream::seekp* zum Positionieren in einem Ausgabe-Stream. Ohne Parameter dir wird auf eine absolute Position positioniert, wie sie von *tellg* zurückgegeben wird.
- *long tellg();* liefert die aktuelle Position im Eingabestream zurück.
- *ostream& flush();* leert den Stream.

- *int opfx();* wird von Ausgabefunktionen aufgerufen, bevor sie Daten in einen Ausgabe-Stream einfügen. Die Funktion *opfx* liefert den Wert 0 zurück, wenn der Fehlerzustand des ostream-Objekts ungleich 0 ist; andernfalls liefert *opfx* einen Wert ungleich 0 zurück.
- *void osfx();* führt Operationen in Anschluß an Ausgaben durch. Wenn *ios::unitbuf* auf *on* gesetzt ist, so leert *osfx* den ostream-Puffer. Im Fehlerfall setzt *osfx* das Bit *ios::failbit*.
- *ostream& put(char ch);* fügt das vorgegebene Zeichen ein.
- *ostream& seekp(streamoff, seek_dir);* positioniert auf eine Position relativ zur aktuellen Position, und zwar in Abhängigkeit von *seek_dir*, welche durch *seek_dir = ios::beg, ios::cur, ios::end* definiert ist. Fehlt der Parameter *seek_dir*, wird auf eine absolute Position positioniert, wie sie von *tellp* zurückgegeben wird.
- *streampos tellp();* liefert die aktuelle Position im Ausgabestream zurück. Vorsicht! Wird beim Öffnen als Modus *ios::app* verwendet, liefert *tellp()* dennoch den Wert 0 zurück, nicht etwa die Dateilänge.
- *ostream& write(const char*, int n);* fügt n Zeichen (Werte von 0 eingeschlossen) ein.

L7 Speichern von Datensätzen mit dynamischen Komponenten

Enthalten Datensätze dynamische Komponenten (Zeiger), können diese Datensätze nicht einfach mit *write()* geschrieben werden. Der Datensatz enthält dann nämlich nicht direkt die dynamische Komponente, sondern nur einen Zeiger auf den Speicherbereich, in dem sich die Komponente befindet. Man würde also lediglich eine Speicheradresse schreiben, nicht aber die interessanten Daten. Beispiel:

```
struct TKundendaten
{
  int   Nummer;
  char  *Name;
  float Umsatz;
};
TKundendaten k;
```

Diese Struktur müsste wie folgt gelesen und geschrieben werden:

```
Dein.write((char*)&k,sizeof(k)); // zunächst Datensatz mit Zeiger schreiben...
Dein.write((char*)k.Name,sizeof(LEN_NAME)); // ...dann string schreiben
//...
Dein.read((char*)&k,sizeof(k)); // zunächst Datensatz einlesen...
// ...dynamisch Platz reservieren für einzulesenden String...
k.Name=new char[LEN_NAME];
Dein.read((char*)s,sizeof(LEN_NAME)); // ...dann erst String lesen
//...
```

Grundlegende Aufgabe	Kunden2a.cpp

Schreiben Sie das Programm *Kunden2.cpp* zu *Kunden2a.cpp* um, indem Sie die Zeichenkette, in der der Kundenname gespeichert ist, dynamisch verwalten.

2.9 Rekursive Algorithmen

Lerninhalte

❶ Unterscheidung der Begriffe Iteration und Repetition
❷ Eigenschaften rekursiver Algorithmen

Lerninhalte

Einen Algorithmus, bei dem eine *Repetition* (Wiederholstruktur) verwendet wird, bezeichnet man als *iterativ*. Die verwendete Wiederholstruktur (*while* oder *for*) nennt man auch *Iteration.* Man kann nachweisen, daß sich alle iterativen Algorithmen auch *rekursiv* realisieren lassen. Ein *rekursiver* Algorithmus ruft sich immer wieder selbst auf - rekursive Module (Funktionen) werden durch sich selbst definiert!

L1 Unterscheidung der Begriffe Iteration und Repetition

Im Beispielprogramm werden nacheinander Namen eingegeben. Nach Beenden der Eingabe (durch Drücken der RETURN-Taste) werden die Namen in umgekehrter Reihenfolge wieder ausgegeben. Einen solchen Datenspeicher bezeichnet man als *Stapel*, *Stack*, *Kellerspeicher* oder *LIFO-Struktur* (lifo = last in, first out).

Die Bezeichnung *lifo* sagt aus, daß das Datenelement, das zuletzt eingegeben wurde, zuerst wieder ausgegeben wird. Genauso funktioniert ein Papier*stapel* auf dem Schreibtisch: Das obenauf liegende Blatt wird zuerst wieder weggenommen. Bei der Bezeichnung *Kellerspeicher* muß man sich einen schmalen Keller mit nur einem Eingang vorstellen: Derjenige, der zuletzt hineingegangen ist, muß als Erster wieder raus.

```
Namen eingeben und in umgekehrter Reihenfolge wieder ausgeben.
(RETURN: Ende der Eingabe)

Name:    Adam
Name:    Berta
Name:    Cäsar
Name:    Dorothea
Name:    Emil
Name:
Ausgabe: Emil
Ausgabe: Dorothea
Ausgabe: Cäsar
Ausgabe: Berta
Ausgabe: Adam

Programmende Iterativ
```

Zunächst wird die *iterative* Lösung der Aufgabenstellung als Programm *Iterativ.cpp* gezeigt.

```
/* PROGRAMM Iterativ.cpp
   Namen mittels Iteration auf Stapel speichern und wieder ausgeben */
#include <iostream>
#include <string>
#include <vector>
using namespace std;

vector<string> Name;

void NameEin() {
  string Eingabe;
  while (true) {
    cout << "Name:    "; getline(cin,Eingabe,'\n');
    if (Eingabe=="") break;
    Name.push_back(Eingabe); // an Vektor anhängen
  };
}

void NameAus() {
  for (int aus=Name.size()-1;aus>=0;aus--)
    cout << "Ausgabe: " << Name[aus].c_str() << endl;
}

void main() {
  cout << "Namen eingeben und in umgekehrter Reihenfolge wieder ausgeben.\n";
  cout << "(RETURN: Ende der Eingabe)\n\n";
  NameEin();
  NameAus();
  cout << "\nProgrammende Iterativ\n";
}
```

Wir erkennen, daß die einzelnen Namen zunächst in eine lokale Variable *Eingabe* vom Typ *string* eingelesen und dann in einem globalen *vector* gespeichert werden. Danach werden die *vector*-Elemente wieder ausgegeben, ausgehend vom letzten Element aus abwärts.

Nun betrachten wir die *rekursive* Lösung (mit exakt gleicher Bildschirmausgabe) als Programm *Rekurs1.cpp*:

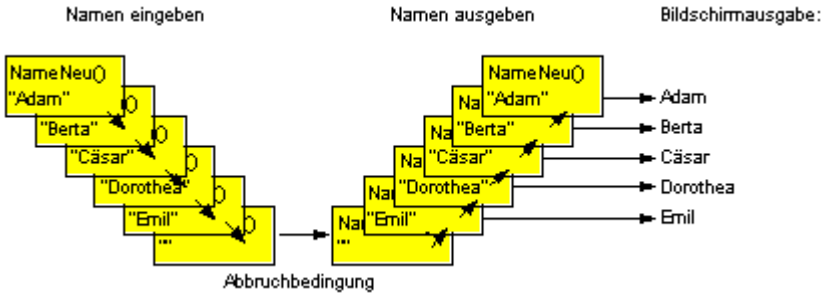
```
/* PROGRAMM Rekurs1.cpp
   Namen mittels Rekursion auf Stapel speichern und wieder ausgeben */
#include <iostream>
#include <string>
using namespace std;

void NameNeu() {
  string Eingabe;

  cout << "Name:    "; getline(cin,Eingabe,'\n');
  if (Eingabe!="") NameNeu(); // Funktion ist mit sich selbst definiert!
    else return;
  cout << "Ausgabe: " << Eingabe.c_str() << endl;
}

void main() {
  cout << "Namen eingeben und in umgekehrter Reihenfolge wieder ausgeben.\n";
  cout << "(RETURN: Ende der Eingabe)\n\n";
  NameNeu();
  cout << "\nProgrammende Rekurs1.";
}
```

Auf den ersten Blick sind wir verblüfft: Die rekursive Programmlösung ist viel einfacher! Sie benötigt keinen *vector* zum Speichern der fünf eingegebenen Datenelemente und auch keine *Repetition* zur

Wiederholung der Eingabe. Trotzdem wird offensichtlich mehrmals zur Eingabe aufgefordert und die Namen werden auch alle gespeichert und wieder ausgegeben. Wie funktioniert das?

Die Arbeitsweise der rekursiven Funktion *NameNeu()* versteht man am besten, wenn man den Debugger zu Hilfe nimmt und jede aufgerufene Funktionszeile im Einzelschrittmodus (F8) durcharbeitet. Dann erkennt man: Immer wenn kein leerer String eingegeben wurde, wird die Eingabe als lokale Variable gespeichert und die Funktion *NameNeu()* wird erneut aufgerufen (Der Programmablauf „taucht neu ein“ in die Funktion). Dabei bleibt die *Eingabe* in der lokalen Variablen zunächst gespeichert.

Erst nach der Eingabe eines Leerstrings ist die *Abbruchbedingung* gegeben: Die Funktion *NameNeu()* ruft sich nicht wieder auf, sondern wird erstmals *beendet*. Danach „taucht“ der Programmablauf wieder im vorhergehenden Funktionsaufruf auf und nimmt die Ausgabe des dort eingegebenen Namens vor. Dieses „Auftauchen“ wird solange in der nächsthöheren Funktion wiederholt, bis das übergeordnete Modul, das die rekursive Funktion erstmals aufrief (das ist in unserem Beispiel das Hauptprogramm), wieder erreicht ist. Den Ablauf kann man grafisch so darstellen:



L2 Eigenschaften rekursiver Algorithmen

Fassen wir die Eigenschaften rekursiver Algorithmen zusammen:

- Rekursive Algorithmen rufen sich immer wieder selbst auf. Es entstehen mehrere *Modulexemplare*, die sich gleichzeitig im Hauptspeicher befinden.
- Jeden rekursiven Algorithmus kann man auch iterativ formulieren - und umgekehrt. In der Regel ist eine rekursive Formulierung deutlich kürzer als eine iterative.
- Damit bei einem rekursiven Algorithmus die (implizit formulierte) Wiederholung des Modulaufrufs zu einem Ende kommt, muß eine eindeutige *Abbruchbedingung* formuliert sein, aufgrund der sich der Algorithmus nicht wieder selbst aufruft.
- Rekursive Algorithmen benötigen gegenüber einer iterativen Formulierung weder eine explizit formulierte Wiederholstruktur noch eine komplexe Datenstruktur zum Speichern mehrerer Datensätze (wie *Array* oder *vector*).
- Die Datenstrukturen, die ein rekursiver Algorithmus verwendet, werden in jedem Programmexemplar *einzeln* als lokale Variablen gespeichert. Dieser lokale Datenspeicher wird erst nach dem Erreichen des Rekursionsabbruchs sukzessive freigegeben.

- Der Speicherbedarf für rekursive Algorithmen wird umso höher, je größer die Anzahl der aufgerufenen Modulexemplare ist. Stellt man bei einer Problemstellung fest, daß eine iterative Lösung weniger Speicherplatz beansprucht, dann wird sie in der Regel auch *schneller* ausgeführt als eine rekursive.
- Lokale Daten werden bei rekursiven Algorithmen in einer *LIFO*-Struktur gespeichert (auf einem systeminternen *Stack*), Die Daten werden in umgekehrter Reihenfolge freigegeben, wie sie gespeichert wurden. Wenn keine Abbruchbedingung eintritt, läuft der interne Stack über und es erfolgt ein Laufzeitfehler mit anschließendem Programmabbruch.

Das nachfolgende Beispiel *Rekurs2.cpp* demonstriert, daß auch Funktionen mit *Rückgabewert* (im Beispiel *int)* rekursiv formuliert werden können:

```
Zahlen eingeben und aufsummieren (0=Ende).

Zahl:  3
Zahl:  8
Zahl:  6
Zahl:  2
Zahl:  0

Summe = 19

Programmende Rekurs2.
```

```
/* PROGRAMM Rekurs2.cpp
   Summieren von Zahlen mittels Rekursion */
#include <iostream>

int Summe() {
  int zahl;

  cout << "Zahl:  "; cin >> zahl;
  if (zahl) return Summe()+zahl; // Funktion ist mit sich selbst definiert!
    else return 0;
}

void main() {
  cout << "Zahlen eingeben und aufsummieren (0=Ende).\n\n";
  cout << "\nSumme = " << Summe() << "\n\n";
  cout << "Programmende Rekurs2.";
}
```

Beachten Sie hier besonders, daß die Anweisung *return Summe()+zahl* auf den ersten Blick suggeriert, daß hier ein *Rücksprung* aus dem aktuellen Funktionsexemplar erfolgt. Bevor der Rücksprung jedoch stattfinden kann, muß zuerst das Ergebnis des rechts hinter *return* stehenden Funktionsaufrufs *Summe()* ermittelt werden, d.h. zum weiteren Programmablauf muß zunächst wieder in die Funktion *hineingesprungen* werden. Erst bei Erreichen von *return 0* bricht die Rekursion ab. (Entsprechend der LIFO-Struktur der aufgerufenen Funktionsexemplare wird die Summe beim obigen Eingabebeispiel in der Abfolge Summe=0+2+6+8+3 gebildet.)

Grundlegende Übungsaufgaben:

Grundlegende Aufgabe	fakultit.cpp und fakultrek.cpp

Schreiben Sie zwei Programme *FAKULTIT.cpp* und *FAKULTREK.cpp*, die die Fakultät einer einzugebenden Zahl *int x* iterativ bzw. rekursiv berechnen. Benutzen Sie in beiden Fällen eine Funktion *int Fakult(x);*

Anmerkungen: x! = 1*2*3*4* ... * x (x! wird gesprochen: „x Fakultät“)

0! = 1 (ist so definiert!)

Beispiele: 2! = 1*2 = 2; 3! = 1*2*3 = 6; 4! = 1*2*3*4; usw.

Grundlegende Aufgabe	eukrek.cpp

Der sog. *Euklidsche Algorithmus* dient zur Ermittlung des *größten gemeinsamen Teilers* (ggT) von zwei ganzen Zahlen. Dazu teilt man die größere durch die kleiner Zahl und bildet den ganzzahligen Rest (C++-Operand: %). Danach bildet man fortlaufend den ganzzahligen Rest aus der Division der jeweils vorherigen kleineren Zahl und dem vorhergehenden Rest, solange bis der Rest 0 entsteht. Dann ergibt sich der ggT aus dem vorherigen ganzzahligen Rest.

Beispiel: ggT (27,15) = ?

27 % 15 ergibt 12

15 % 12 ergibt 3

12 % 3 ergibt 0 => ggt (27,15) = 3

Zeichnen Sie ein Struktogramm und schreiben Sie ein rekursives Programm *Eukrek.cpp.*

2.10 Suchen und Sortieren in Feldern und Dateien

Lerninhalte

❶ Binäres Suchen
❷ Sortieren durch direkte Auswahl
❸ Sortieren durch direktes Einfügen
❹ Sortieren durch direkten Austausch
❺ Effizienz der einfachen Sortierverfahren
❻ Shell Sort
❼ Quick Sort
❽ Suchen und Sortieren in Dateien mittels Indexdateien

Lerninhalte

EDV dient zur effizienten Verwaltung *großer Datenmengen.* Große Mengen von Daten lassen sich leichter verwalten, wenn man sie in *gleichartige Datensätze* aufteilt. Ein Datensatz hat in der Regel den Datentyp einer *struct* oder *class*. Im Hauptspeicher hält man große Datenmengen üblicherweise in *Arrays* (bzw. komfortabler in *vectors*) oder in anderen Listen, während man sie auf Massenspeichern in *Dateien* (als *file stream*) ablegt.

Unabhängig von der Speicherungsart ist jedoch das *Suchen* eines bestimmten Datensatzes ein sehr wichtiges Anliegen. Aus der Alltagserfahrung wissen wir, daß uns das Suchen nach einem bestimmten Blatt Papier in einem ungeordneten Stapel von Schriftstücken wesentlich schwerer fällt als das Suchen in einem Ordner, wo die einzelnen Papiere alphabetisch oder nach Themengebieten *sortiert* sind. Daher stellen schnelle und effiziente *Sortierverfahren* in der Informatik die Vorbedingung für leistungsfähige Suchalgorithmen dar.

Für die Effizienz von Such- und Sortierverfahren ist es ein wichtiges Kriterium, in welcher Weise auf ein einzelnes Datenelement *zugegriffen* werden kann. Datenstrukturen, die *Direktzugriff* erlauben (*Arrays*, *vectors* oder *random access files*) lassen sich schneller durchsuchen und sortieren als Datenmengen, bei denen lediglich *sequentiell* auf die einzelnen Elemente zugegriffen werden kann.

Will man Datenmengen sortieren, kann man verschiedene Algorithmen auswählen. Grundsätzlich gilt hierbei, daß einfach verständliche Algorithmen bei größeren Datenmengen einen größeren Zeitaufwand verursachen als komplizierte. Als „einfach“ können die folgenden Sortierverfahren gelten:

- *Selection Sort* (Sortieren durch direkte Auswahl)
- *Insertion Sort* (Sortieren durch direktes Einfügen)
- *Bubble Sort* (Sortieren durch direkten Austausch)

Komplexere Sortierverfahren sind:

- *Shell Sort*
- *Quick Sort*

In den folgenden Programmbeispielen arbeiten wir mit *Arrays* und einfachen Datensatzstrukturen wie *int* oder char. Die vorgestellten Such- und Sortiermethoden können jedoch auf beliebig komplexe Datensätze und *data streams*, die im *Direktzugriff* bearbeitet werden können, angewandt werden.

L1 Binäres Suchen

Liegt eine Sammlung von Daten vollkommen *unsortiert* vor, dann kann man nach einem bestimmten Datensatz nur *sequentiell* suchen: Jeder einzelne Satz muß nacheinander daraufhin betrachtet werden, ob er den Suchbedingungen genügt. Ein sequentielles Suchverfahren ist zwar sehr einfach zu implementieren, seine Effizienz ist jedoch vollständig dem Zufall überlassen. Der gesuchte Datensatz kann sowohl beim ersten Zugriff gefunden werden als auch beim allerletzten!

Index	PersNr	Name	Vorname	GebDat	PLZ	StrNr
0	1000	Aalberg	Anton	01.02.50	67059	A-Str. 1
1	1001	Bergmann	Bertram	02.02.50	67069	B-Str. 2
2	1002	Caesar	Cäcilie	03.03.52	67000	C-Str. 3
3	1003	Deis	Dieter	03.04.53	67001	D-Str. 4
4	1004	Erdmann	Erdmut	04.05.55	67055	E-Str. 5
5	1005	Fath	Frank	06.07.57	67077	F-Str. 6

Deutlich effektiver kann man arbeiten, wenn die Datenmenge nach dem Suchbegriff *sortiert* vorliegt. Einen Suchbegriff bezeichnet man auch als *Schlüssel.*

```
// Beispiel: Personendatensatz der Kartei eines Einwohnermeldeamtes
struct TDatum {
  int Tag;
  int Monat;
  int Jahr;
};
struct TPerson {
  int Personalnummer;
  char Name[20];
  char Vorname[10];
  Tdatum Geburtsdatum;
  int PLZ;
  char StrasseHnr[20];
};
// Ein einfacher Schlüssel, nach der eine bestimme Person gesucht werden
// kann, könnte Name sein. Ein komplizierterer Schlüssel könnte ein String
// sein, der sich Name+Vorname+Geburtsdatum zusammensetzt.
```

Im folgenden Beispiel wird der Datensatz mit Name=„Erdmann“ gesucht.

Schritt:	1a	1b	2a	2b	3a	3b
0	Suchbereich					
1						
2		X				
3			Suchbereich		S.b.	X
4				X		
5						

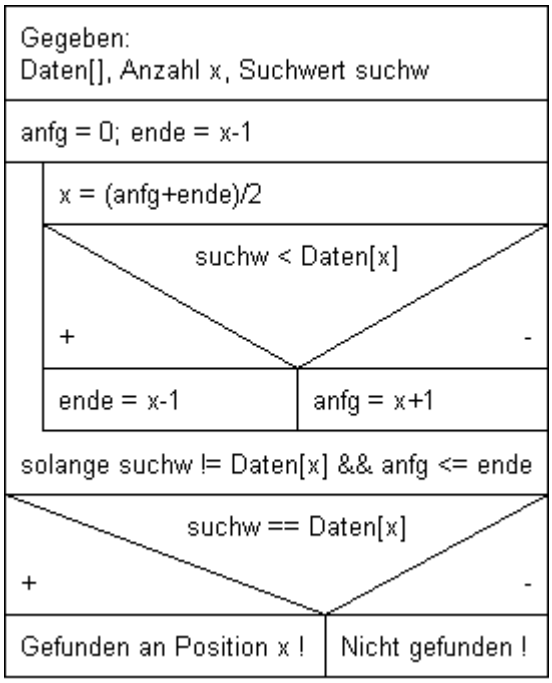
1. Wir nehmen an, daß x Datensätze vorliegen, die nach dem Schlüssel sortiert sind. Sie tragen die Nummern 0 bis x-1.
2. Wir betrachten zunächst das Schlüsselelement im Datensatz, das genau in der Mitte liegt (zunächst Nr. x/2).
3. Ist das betrachtete Datenelement kleiner als der gesuchte Schlüssel, betrachten wir lediglich noch die Datensätze Nr. 0 bis x/2-1, ist er größer, betrachten wir nur noch die Sätze Nr. x2+1 bis x.

In beiden Fällen sehen wir uns dann wieder den Datensatz an, der genau in der Mitte der

Betrachtungsmenge liegt und verfahren weiter wie bei 2. Ist der betrachtete Datensatz jedoch weder kleiner noch größer als der gesuchte Schlüssel, dann haben wir den gesuchten Begriff gefunden und der Algorithmus endet hier!

4. Die Schleife 2-3 wiederholen wir solange, bis entweder der Suchbegriff gefunden wurde oder die Betrachtungsmenge auf ein einziges Element geschrumpft ist, das den Suchbegriff nicht enthält: Dann enthält die betrachtete Datenmenge den Suchbegriff nicht und der Algorithmus endet mit Mißerfolg.

Das Struktogramm für einen binären Suchalgorithmus sieht folgendermaßen aus:



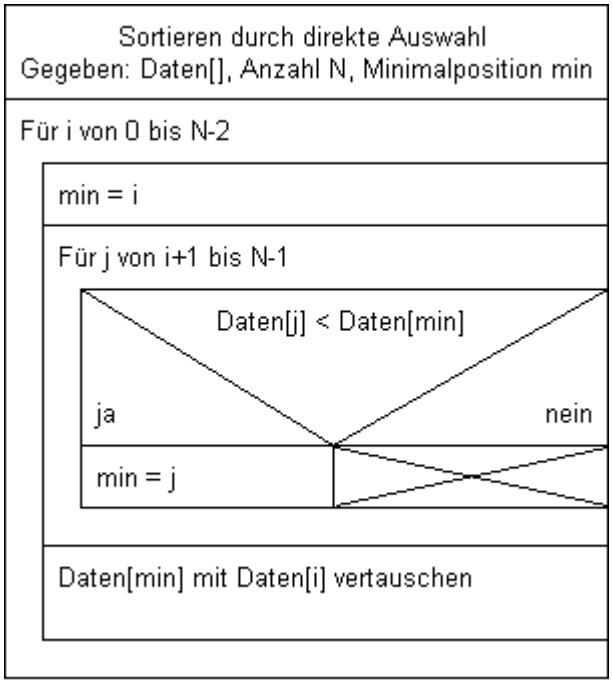
Grundlegende Aufgabe suchbin.cpp

Schreiben Sie ein Programm *SuchBin.cpp*, das die nachfolgende Bildschirmausgabe erzeugt. Benutzen Sie als Vorlage das Fragment *SuchBin.fra* (Die Funktion *Finde()* ist noch zu schreiben).

```
Einen Wert in einer Menge von 20 Zahlen suchen:
Position:  0    1    2    3    4    5    6    7    8    9

Werte:    10,  13,  17,  25,  44,  63,  72,  78,  96,  99,
         104, 286, 294, 314, 520, 633, 709, 899, 913, 956

Zu suchende Zahl eingeben (0=Programmende): 63
Suchwert 63 an Position 5 gefunden!
Zu suchende Zahl eingeben (0=Programmende): 14
Suchwert 14 nicht gefunden!
Zu suchende Zahl eingeben (0=Programmende): 709
Suchwert 709 an Position 16 gefunden!
Zu suchende Zahl eingeben (0=Programmende): 916
Suchwert 916 nicht gefunden!
Zu suchende Zahl eingeben (0=Programmende): 0
```

L2 Sortieren durch direkte Auswahl (Selection Sort)

Diese Sortiermethode läuft wie folgt ab: Finde zuerst das kleinste Element in der Datenmenge und tausche es gegen das Feld an Platz 0 aus. Finde danach das zweitkleinste Element und tausche es gegen das Feld an Platz 1 aus, usw., bis die gesamte Datenmenge sortiert ist.



Während der Index i vom Anfang bis zum Ende durch das Datenfeld wandert, befinden sich die Elemente *links* von Platz i auf ihrer endgültigen Position im Datenfeld. Das Feld ist vollständig sortiert, wenn i das Feldende erreicht hat.

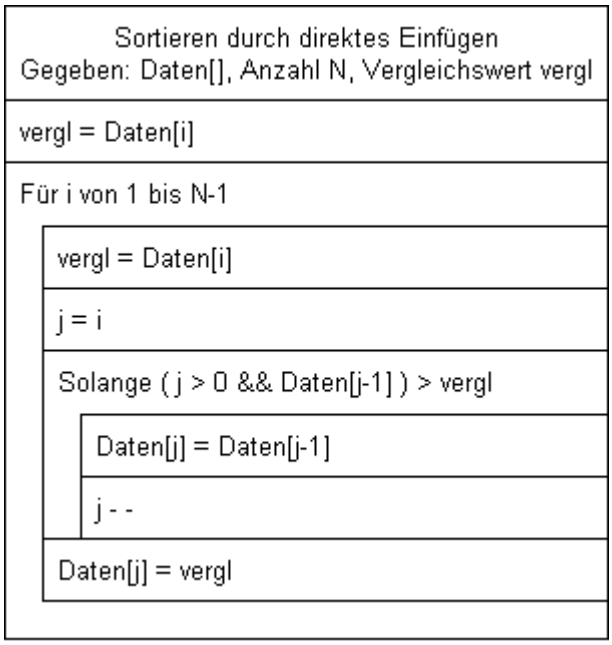
Selection Sort ist eine der einfachsten Sortiermethoden und für kleine Datenmengen sehr gut brauchbar. Da jedes Element wirklich höchstens einmal bewegt wird, ist *Selection Sort* die bevorzugte Methode, um Datenmengen mit sehr großen Datensätzen und kleinen Schlüsseln zu sortieren.

L3 Sortieren durch direktes Einfügen (Insertion Sort)

Direktes Einfügen ist die Methode, die Menschen oft beim Kartenspiel anwenden, um ihre Karten zu sortieren: Betrachte die Elemente eines nach dem anderen und füge jedes an seinen richtigen Platz zwischen den bereits betrachteten ein (wobei die bereits betrachteten Elemente sortiert bleiben).

Das gerade betrachtete Element wird eingefügt, indem die größeren Elemente einfach um eine Position zum Feldende hin bewegt werden und das Element dann auf dem freigewordenen Platz eingefügt wird.

FEHLER!! Schleifeninit weglassen: vergl=Daten[i]. Klammer bei Solange falsch. Musterlösung Schleife falsch.



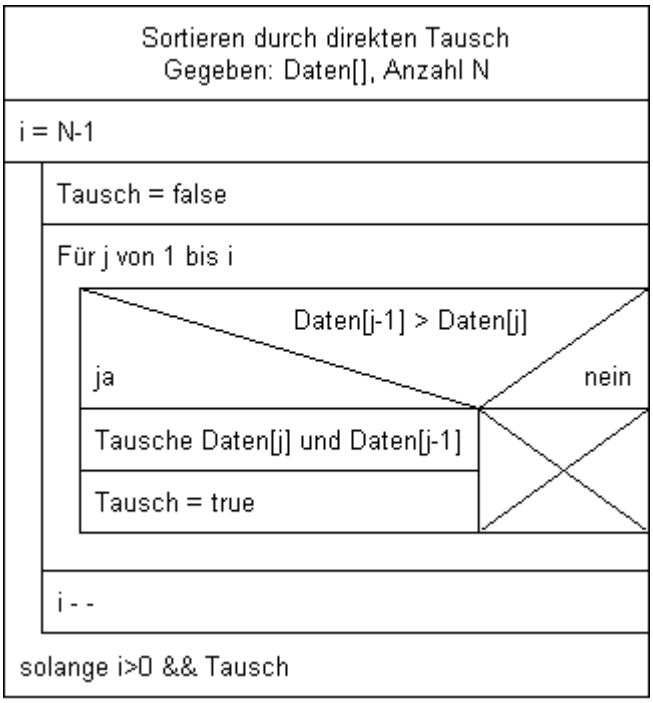
Für jeden Datenplatz *i* von 1 bis N-1 wird der Unterbereich *Daten[0-i]* sortiert, indem der jeweilige Vergleichswert *Daten[i]* an die entsprechende Stelle zwischen den Elementen in dem sortierten Unterfeld vor Platz *i* gesetzt wird. Vorher wird im Unterfeld Platz geschaffen, indem alle nach der Einfügestelle stehenden Datenelemente um einen Platz weitergeschoben werden.

Im obigen Algorithmus ist eine Besonderheit zu beachten: Die Bedingung *Solange (j<0 && Daten[j-1]>vergl)* funktioniert nur, wenn der C++-Compiler die UND-Verknüpfung der beiden Terme im *Kurzschlußverfahren* berechnet (d.h. wenn die Erste Bedingung *j>0* schon falsch war, wird die zweite überhaupt nicht mehr berechnet). Die meisten C++-Compiler werten UND-Operationen jedoch im Kurzschlußverfahren aus.

Überhaupt ist die Abfrage auf *j<0* nur dann notwendig, wenn *vergl* das kleinste Element in der Datenmenge ist - nur dann kann *j-1* den Wert 0 unterschreiten. Will man Sortierzeit einsparen, kann man auf den Term *„j>0 &&“* verzichten, wenn man im Datenfeld Nr. 0 *vor* Beginn des Sortiervorganges einen *Markenschlüssel* mit garantiert kleinstmöglichem Wert setzt (z.B. ‘\0’). In diesem Fall reduziert sich die Anzahl der tatsächlich zu sortierenden Datenwerte um 1, denn im Feld Nr. 0 kann kein echter Datenwert mehr stehen.

L4 Sortieren durch direkten Tausch (Bubble Sort)

Stellt man sich die Datenelemente wie Luftblasen senkrecht angeordnet in Wasser schwebend vor, so sorgt das *Bubble-Sort*-Verfahren dafür, daß die größeren Elemente „nach oben steigen“, wie Wasserblasen (daher der englische Name). Beim *Bubble Sort* wird die Datenmenge immer wieder durchlaufen. Dabei werden benachbarte Elemente vertauscht, wenn dies notwendig ist. Die Datenmenge gilt als sortiert, wenn bei einem Durchlauf kein Austausch mehr erforderlich war.



Nach dem ersten Durchlaufen der äußeren Schleife steht das größte Element am Ende des Datenfelds. Danach wird das Datenfeld nur noch bis zum zweitletzten Element bearbeitet. (Die boolsche Variable *Tausch* kann man einsparen, wenn man die temporäre Datenvariable, welche für den Tauschvorgang notwendig ist, mit einem Wert initialisiert, der in der Datenmenge nicht vorkommt und dann diesen Weg als Keinnzeichen dafür abfragt, daß kein Tauschprozeß stattgefunden hat.)

Grundlegende Aufgabe cstring4.cpp

Schreiben Sie ein Programm *Cstring4.cpp*, das das folgende C-String-Array mit dem Bubble-Sort-Algorithmus sortiert und ausgibt:

```
char strArr[STR_SIZE][ARRAY_SIZE] = {
  “Saarland“, “Rheinland-Pfalz“,“Schleswig-Holstein“,“Sachsen“,
  “Brandenburg“,“Nordrhein-Westfalen“,“Baden-Württemberg“,
  “Sachsen-Anhalt“,“Niedersachsen“,“Hamburg“,“Bremen“ };
```

L5 Effizienz der einfachen Sortierverfahren

Man kann die verschiedenen Sortierverfahren anhand der folgenden Kriterien auf Effizienz vergleichen:

- Anzahl der benötigten Vergleichs- und Austauschoperationen in Abhängigkeit von der Zahl der zu sortierenden Elemente
- Bedarf an zusätzlichem Speicherplatz während des Sortiervorganges.

Die bisher betrachteten Sortierverfahren benötigen keinen zusätzlichen Speicherplatz (bis auf wenige Variablen für Vergleichs- und Tauschoperationen). Eine genaue Betrachtung, die hier nicht nachvollzogen werden soll, ergibt weiterhin (bei N zu sortierenden Datenelementen) die folgenden Durchschnittswerte:

Sortierverfahren	*Vergleiche*	*Austausch-operationen*	*Aufwand linear bei*
Selection Sort	$N^2/2$	N	großen Datensätzen und kleinen Schlüsseln
Insertion Sort	$N^2/4..N^2/2$	$N^2/8..N^2/4$	fast sortierten Dateien
Bubble Sort	$N^2/2$	$N^2/2$	

Sind die zu sortierenden Datensätze in real existierenden Datenmengen (Arrays oder Dateien) sehr groß, dann kostet jede Austauschoperation viel Rechenzeit. In solchen Fällen sollte man speziell für die Sortierung eine zweite *Index-Datenmenge* aufbauen, deren Datensätze nur aus dem Schlüssel und einer Referenz auf die Datensätze der Originaldatei bestehen, hier ist es der Array-Index. Man sortiert dann lediglich die Indexmenge und organisiert danach aus den Werten der Indexmenge die Datensätze der Original-Datenmenge neu.

```
struct GrossSatz
{
  char Name[20]; // Das soll das Schlüsselelement sein
  /* ... sehr viele weitere Elemente */
};
struct Indexsatz
{
  char Name[20];
  int  Index;
};
const N=200;
GrossSatz Datenfeld[N]; // Original-Datenmenge mit großen Datensätzen
Indexsatz Indexfeld[N]; // Indexmenge mit kleinen Datensätzen
GrossSatz temp;
int j,k;

/* Zunächst wird die Original-Datenmenge eingelesen ... */

for (int i=0; i<=N-1, i++)
{ // Aufbau der Index-Datenmenge
  Indexfeld[i].Name = Datenfeld[i].Name;
  Indexfeld[i].Index = i; // zeigt auf Datensatz in Original-Datenmenge
}
/* Nun wird die Index-Datenmenge sortiert ... */

/* .. und schließlich wird die Original-Datenmenge reorganisiert: */
for (int i=0; i<=N-1; i++)
{
  if (Indexfeld[i].Index != i)
  {
    temp = Datenfeld[i]; k = i;
    do
    {
      j=k; k=Indexfeld[j].Index;
      Datenfeld[j]=Datenfeld[k];
      Indexfeld[j].Index = j:
    }
    while (k!=i);
    Datenfeld[j] = temp;
  }
}
```

Der Algorithmus zur Reorganisation der Originaldatenmenge sei an einem Beispiel erläutert:

```
Position:        0 1 2 3 4 5 6 7 8 9 1 1 1 1 1 1 1 1 1 1
                                     0 1 2 3 4 5 6 7 8 9
Original-Daten:  D A S I S T Z U S O R T I E R E N A B C

Sortierte Daten: A A B C D E E I I N O R R S S S T T U Z
Index:           1 1 1 1 0 1 1 3 1 1 9 1 1 2 4 8 5 1 7 6
                   7 8 9   3 5   2 6   0 4         1
```

Der Datensatz an Platz 0 (=i) trägt den Index 1. Daher wird das Original-Datenfeld Nr. 0 (=i) in *temp* zwischengespeichert. Nach dem ersten Eintritt in die *do*-Schleife wird der Datensatz Nr. 1 (=k) auf den Platz Nr. 0 (=j) geschrieben. Danach wird der Index von Platz Nr. 0 von 1 auf 0 (=j) korrigiert. Nun sind wir am Schleifenende der *do*-Schleife angelangt und an Platz Nr. 1 (=k) in der Originalmenge befindet sich ein „Loch“.

Die weiteren Eintritte in die *do*-Schleife laufen folgendermaßen ab:

Schleifendurchgang Nr.	*Satz Schreiben Nr. k -> Nr. j*	*Index korrigieren an Platz Nr .*	*„Datenloch“ an Platz k =*

		j: k -> j	
2	17 -> 1	1 : 17 -> 1	17
3	11 -> 17	17 : 11 -> 17	11
4	10 -> 11	11 : 10 -> 11	10

usw.

Der Ablauf wiederholt sich solange, bis die Platznummer, an der sich das „Datenloch“ befindet, wieder gleich 0 (=i) ist. So wird letztlich die gesamte Original-Datenmenge umgeordnet, wobei jeder Datensatz nur einmal bewegt wird.

L6 Shell Sort

Insertion Sort ist langsam, weil nur benachbarte Elemente ausgetauscht werden. Befindet sich das kleinste Element zufällig am Ende der Datenmenge, so werden N Schritte benötigt, um es an seinen richtigen Platz zu verschieben. *Shell Sort* ist eine einfache Erweiterung von *Insertion Sort*, bei dem eine Erhöhung der Geschwindigkeit dadurch erzielt wird, daß ein Vertauschen von Elementen ermöglicht wird, die weit voneinander entfernt sind.

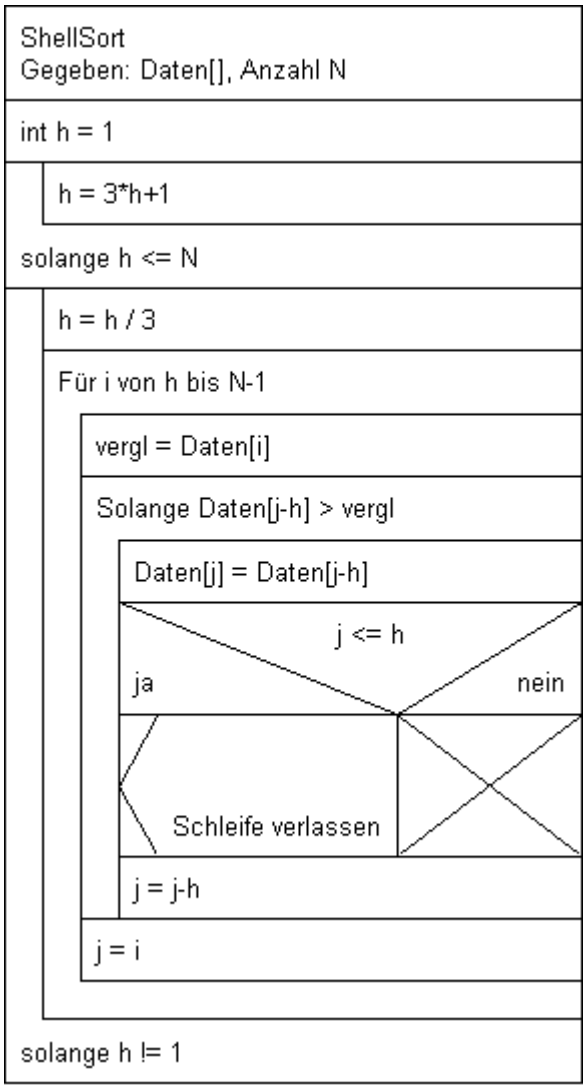
Der Grundgedanke besteht darin, die Daten zu umzuordnen, daß man eine sortierte Datenmenge erhält, wenn man jedes *h*-te Element (bei beliebigem Anfangselement) entnimmt. Eine solche Datenmenge wird *h-sortiert* genannt. Man kann auch sagen, daß ein h-sortiertes Datenfeld aus h unabhängigen sortierten Datenfeldern besteht, die einander überlagern. Durch das h-Sortieren für große h können wir Elemente im Feld über größere Entfernungen bewegen und damit ein h-Sortieren für kleinere Werte von h erleichtern.

Wendet man das h-Sortieren für eine beliebige Folge von h-Werten an, die mit h=1 endet, ergibt sich der *Shell-Sort*-Algorithmus zum Sortieren der gesamten Datenmenge. Wir müssen im Quellcode von *Insertion Sort* lediglich jedes Auftreten von *1* durch *h* ersetzen.

Der im nachfolgenden Struktogramm beschriebenen Algorithmus verwendet für h die Distanzenfolge ..., 1093, 364, 121, 40, 13, 4, 1 (*h = h/3)*. Andere Folgen von Distanzen können in der Praxis mit etwa gleichem Erfolg verwendet werden, wieder andere führen zu noch größerer Effizienz.

Für die verwendete Distanzenfolge kann man nachweisen, daß niemals mehr als $N^{3/2}$ Vergleiche durchgeführt werden. Jedoch ist es auch für große N schwer, den unten angegebenen Algorithmus in der Effizienz um mehr als 20 % zu übertreffen. Die Zahl der Austauschoperationen kann man (aufgrund theoretischer Schwierigkeiten) nicht genau berechnen - für die verwendete Distanzenfolge lauten zwei Vermutungen $N * (log\ N)^2$ und $N^{1,25}$.

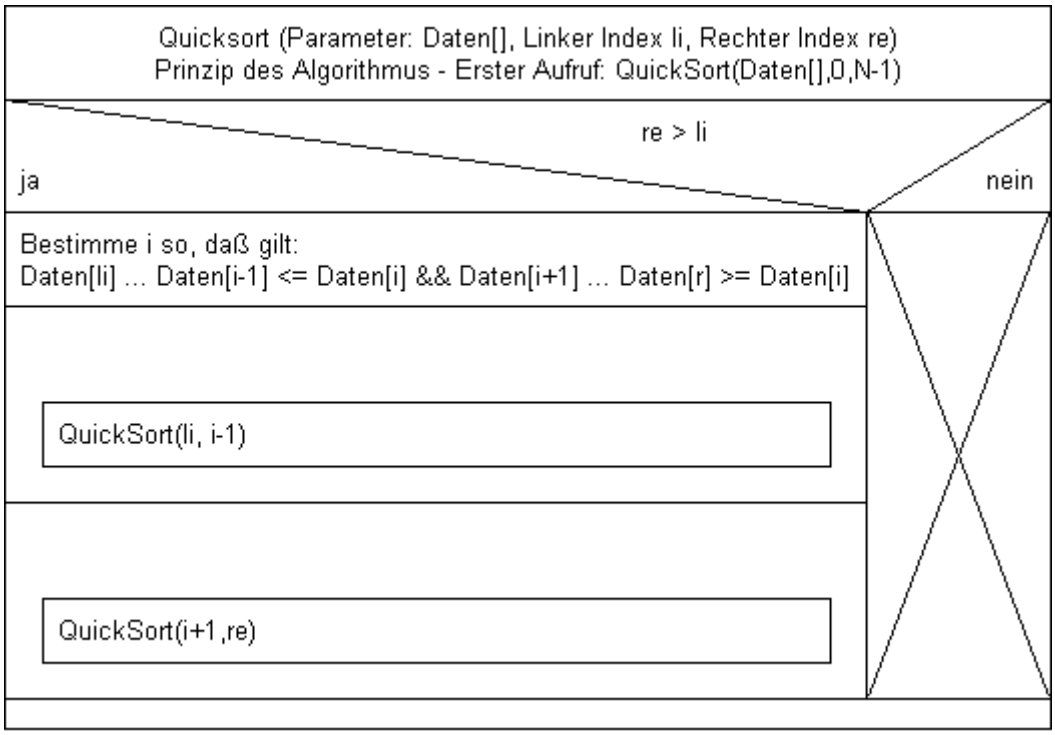
Shell Sort ist die bevorzugte Methode für viele Anwendungen des Sortierens, da es selbst für relativ große Datenmengen (mit etwa bis 5000 Elementen) eine akzeptable Laufzeit aufweist.


ShellSort
Gegeben: Daten[], Anzahl N
int h = 1
h = 3*h+1
solange h <= N
h = h / 3
Für i von h bis N-1
vergl = Daten[i]
Solange Daten[j-h] > vergl
Daten[j] = Daten[j-h]
j <= h
ja
nein
Schleife verlassen
j = j-h
j = i
solange h != 1

L7 Quick Sort

Quick Sort ist ein Sortierverfahren nach dem Prinzip „Teile und Herrsche“. Eine Datenmenge wird in zwei Teile zerlegt, welche unabhängig voneinander sortiert werden. Quick Sort benötigt wenig zusätzlichen Hauptspeicher. Das Sortieren von N Elementen erfordert im Durchschnitt etwa 2N*ln(N) Vergleiche und N*log(N) Austauschoperationen, die innere Anweisungsschleife ist extrem kurz.

Quick Sort wurde erstmals 1960 entwickelt (von C.A.R. Hoare). Seine Leistungsfähigkeit ist sehr gut erforscht. In der Regel wird der Algorithmus *rekursiv* implementiert und kann dann recht einfach formuliert werden. Das wird zunächst mit einem *grundlegenden* Struktogramm verdeutlicht:



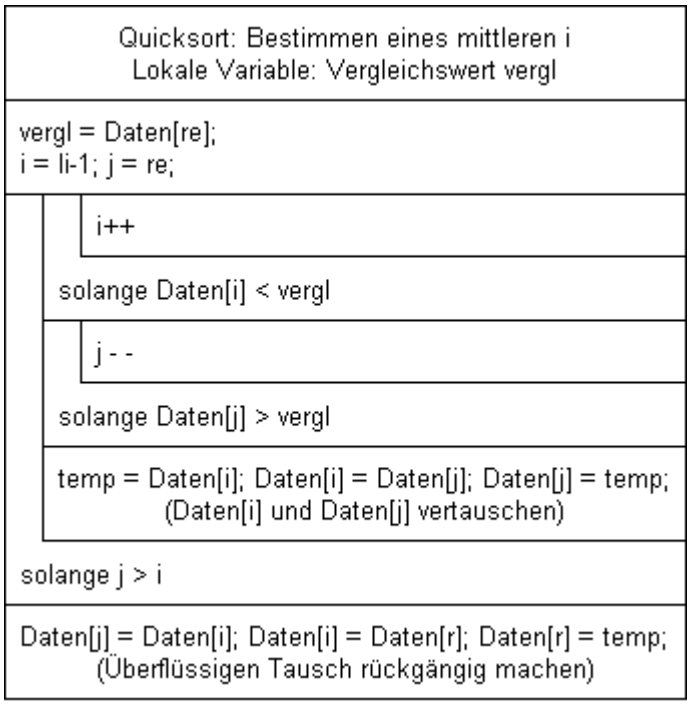
Der „Trick“ besteht also darin, bei jedem Rekursionsaufruf ein *Daten[i]* zu finden, das in seine endgültige Position innerhalb der Datenmenge gebracht werden kann. Der Wert von i kann folgendermaßen bestimmt werden: Wähle zuerst willkürlich *Daten[re]* als Vergleichswert. Gehe dann in der Datenmenge

a) vom Anfang an aufwärts, solange, wie die Elemente kleiner als *Daten[re]* sind.

b) vom Ende her abwärts, solange, wie die Elemente größer als *Daten[re]* sind.

Die beiden besagten Elemente, die offensichtlich in der falschen Teilmenge liegen, werden nun ausgetauscht. Wenn man in dieser Weise fortfährt, ist gewährleistet, daß alle Elemente in der Datenmenge vor Position *li* kleiner und alle Elemente nach Position *re* größer als *Daten[re]* sind. Wenn sich die Positionen *li* und *re* treffen, ist der Zerlegungsprozeß nahezu beendet. Dann muß nur noch *Daten[re]* mit *Daten[li]* vertauscht werden. Der exakte Algorithmus zum Bestimmen eines mittleren *i* ist unten im Struktogramm beschrieben.

Die Variable *vergl* dient hier zur Speicherung des aktuellen Werts des „zerlegenden Elements“ *Daten[i]*, während *i* der linke bzw. rechte Zeiger des Durchsuchens sind. Ein zusätzlicher Austausch von *Daten[i]* und *Daten[j]* mit *j<i* erfolgt unmittelbar, nachdem sich die Zeiger treffen, doch bevor das Treffen festgestellt und die äußere *while*-Schleife verlassen wird. Die drei Zuweisungsbefehle, die nach dieser Schleife folgen, implementieren den Austausch von *Daten[i]* und *Daten[j]*, um das zusätzliche Austauschen rückgängig zu machen, sowie von *Daten[i]* und *Daten[re]*, um das zerlegende Element an die richtige Position zu setzen.



Wie bei *Insertion Sort* wird ein *Markenschlüssel* benötigt, um das Durchsuchen abzubrechen, falls das zerlegende Element zugleich auch das kleinste ist. Für den entgegengesetzten Fall, daß also das zerlegende Element zurgleich das größte ist, wird kein Markenschlüssel benötigt, da das zerlegende Element am rechten Ende steht und dort die Suche abbricht.

Die „innere Schleife“ von Quick Sort beinhaltet lediglich das Inkrementieren eines Zeigers und den Vergleich eines Feldelementes mit einem festen Wert - eben das macht Quick Sort so schnell. Hier sieht man auch den Vorteil der Benutzung von Marken - das Hinzufügen auch nur eines überflüssigen Tests zur inneren Schleife hätte einen spürbaren Einfluß auf die Leistungsfähigkeit. Nun werden die beiden Teildateien rekursiv sortiert, womit das Sortieren beendet wird.

Das folgende Ablaufbeispiel demonstriert an einer kurzen Zeichenfolge die Funktionsweise des Algorithmus:

```
1 2 3 4 5 6   // Platz-Nr. im Zeichen-Array
U N S O R T   // Ausgangsfolge: Vergleichswert 'T'
i       j     // i=1, j=5 setzen
R N S O U T   // 'U' und 'R' tauschen
      j i     // i=5, j=4 setzen
R N S U O T   // 'O' und 'U' tauschen
              // j < i => Schleife verlassen
R N S O T U   // Vergleichswert 'T' auf richtigen Platz setzen
R N S O       // „Linker" rekursiver Aufruf: Neuer Vergleichswert 'O'
i j           // i=1, j=2 setzen
N R S O       // 'R' und 'N' tauschen
j i           // i=2, j=1 setzen
R N S O       // 'N' und 'R' nochmal tauschen
              // j < i => Schleife verlassen
N O S R       // Vergleichswert 'O' auf richtigen Platz setzen
    S R       // „Rechter" rekursiver Aufruf: Neuer Vergleichsert 'R'
  j i         // i=3, j=2 setzen
  S O R       // 'O' und 'S' tauschen
              // j < i => Schleife verlassen
  0 R S       // Vergleichswert 'R' auf richtigen Platz setzen
N O R S T U   // Endzustand: Sortierte Folge
```

Grundlegende Aufgaben verschiedene

Benutzen Sie das Programm-Fragment *Sortiere.fra* als Vorlage, um damit die folgenden Sortieralgorithmen zu implementieren und auszutesten:

1. Selection Sort (*SelSort.cpp* mit Funktion *SelectionSort*)
2. Insertion Sort (*InsSort.cpp* mit Funktion *InsertionSort*)
3. Bubble Sort (*BubSort.cpp* mit Funktion *BubbleSort*)
4. Shell Sort (*ShellSort.cpp* mit Funktion *ShellSort*)
5. Quick Sort (*QuiSort.cpp* mit Funktion *QuickSort*)

Die Programmiervorlage enthält ein char-Array mit unterschiedlichen unsortierten Vorbelegungen, die beliebig auskommentiert werden können. Innerhalb des eigentlichen Sortieralgorithmus kann man Zwischenzustände anzeigen.

L8 Suchen und Sortieren in Dateien mittels Indexdateien

Reale Dateien haben in der Regel *große Datensätze*. Zu den Standardfunktionen einer *Datenbank* (= Sammlung unterschiedlicher Dateien) gehört das Suchen nach verschiedenen Datensatzelementen.

```
struct Tkundensatz { // Das ist die Struktur eines einzelnen Datensatzes
  int   Nr;          // Personenkennziffer
  char  Name[20];    // Name des Kunden
  float Umsatz;      // Umsatz, den der Kunde erbringt
};
Tkundensatz Kundensatz;
```

Man stelle sich eine sehr große Datei vor, die Datensätze der obigen Struktur enthält. In einer solchen Datei kann man nach drei unterschiedlichen Schlüsseln suchen:

- nach der Personenkennziffer (Kundensatz.Nr)
- nach dem Namen des Kunden (Kundensatz.Name)
- nach dem Umsatz, den der Kunde erzielt (Kundensatz.Umsatz).

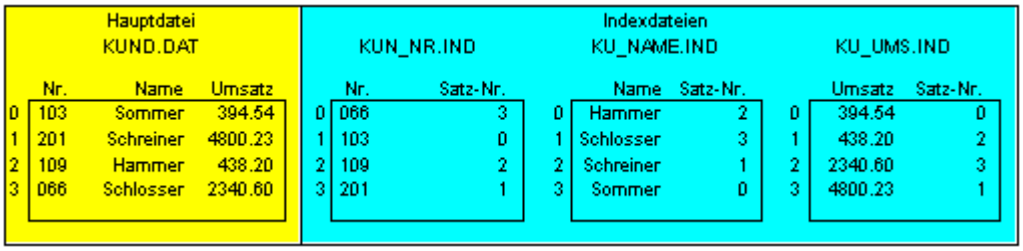
Ein effizienten Suchen nach diesen Begriffen erfordert, daß die Datei nach *allen drei* Begriffen sortiert vorliegt. Auf den ersten Blick ergeben sich hier zwei unterschiedliche Ansätze:

– Die Datei wird vor jedem Suchvorgang vollständig neu nach dem aktuellen Suchschlüssel sortiert.

– Die Datei wird *dreifach* vorgehalten und aktualisiert, jeweils nach einem anderen Schlüssel sortiert.

Beide Denkansätze können sofort verworfen werden, da die zu erzeugenden bzw. zu bewegenden Datenmengen zu groß sind, um eine befriedigende Ausführungsgeschwindigkeit zu erhalten. Deshalb wählt man einen anderen Ansatz:

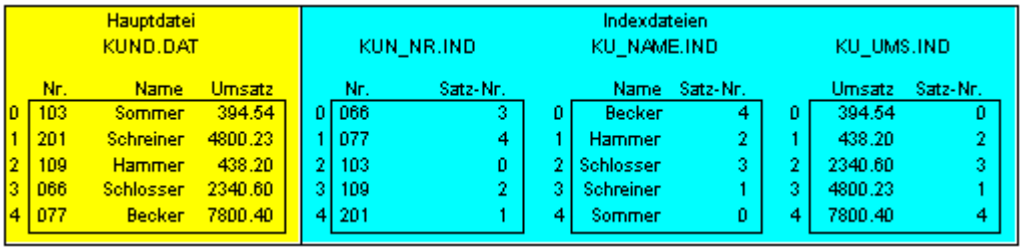
Man baut, zusätzlich zur *Hauptdatei*, für jeden Schlüssel eine sog. *Indexdatei* auf. In jeder Indexdatei stehen nur *Schlüssel* und zugehörige *Datensatznummer* der Hauptdatei. Jede Indexdatei wird nach ihrem Schlüssel sortiert.



Vor den einzelnen Rahmen, die jede Datei kennzeichnen, stehen die *logischen Datensatznummern* der jeweiligen Datensätze (0, 1, 2, 3, ...). Man sieht, daß Datensatznummern der Hauptdatei als Datenelemente der Indexdateien notiert sind. Man kann auch sagen, die Index-Datensätze *zeigen* auf die Datensätze der Hauptdatei. Der Zugriff auf einzelne Datenelemente erfolgt nun folgendermaßen:

1. *Datensatzeingabe:* Jeder neue Datensatz wird am Ende der Hauptdatei angehängt. Gleichzeitig wird *jede* Indexdatei aktualisiert, indem die neuen Schlüssel mit Verweis auf die Datensatznummer der Hauptdatei *einsortiert* werden. Wenn es die Größe des verwendeten Hauptspeichers zuläßt, werden die Indexdateien (mindestens teilweise) in den Hauptspeicher gelesen.

Beispiel: Ein neuer Datensatz „077 Becker 7800.40“ wir in KUND.DAT angehängt. Dann ergibt sich:



Hauptdatei KUND.DAT

	Nr.	Name	Umsatz
0	103	Sommer	394.54
1	201	Schreiner	4800.23
2	109	Hammer	438.20
3	066	Schlosser	2340.60
4	077	Becker	7800.40

Indexdateien

KUN_NR.IND

	Nr.	Satz-Nr.
0	066	3
1	077	4
2	103	0
3	109	2
4	201	1

KU_NAME.IND

	Name	Satz-Nr.
0	Becker	4
1	Hammer	2
2	Schlosser	3
3	Schreiner	1
4	Sommer	0

KU_UMS.IND

	Umsatz	Satz-Nr.
0	394.54	0
1	438.20	2
2	2340.60	3
3	4800.23	1
4	7800.40	4

2. *Datensatzausgabe:* Je nach gewünschter Sortierung wird eine bestimmte Indexdatei *sequentiell* gelesen. Von jedem Index-Datensatz aus wird über die dort gespeicherte Datensatznummer der Hauptdatei im *Direktzugriff* der entsprechende Datensatz der Hauptdatei gelesen und ausgegeben. Diese Art des Dateizugriffs bezeichnet man als *index-sequentiell* (engl. ISAM - „Index Sequential Access Method“).
3. *Datensatz suchen:* Der Schlüssel wird über binäre Suche in der passenden Indexdatei gesucht. Dann wird wie bei 2. im Direktzugriff auf den entsprechenden Datensatz der Hauptdatei zugegriffen.
4. *Datensatz ändern:* Der Schlüssel wird gesucht, wie bei 3. beschrieben. Dann wird der dazu passende Datensatz in der Hauptdatei geändert und *alle* Indexdateien werden an die Änderung angepaßt, d.h. die zugehörigen Datensätze in den Indexdateien werden gegebenenfalls umsortiert.
5. *Datensatz löschen:* Der Schlüssel wird gesucht. Dann wird meist der zu löschende Datensatz in der Hauptdatei mit einer Löschkennung versehen, denn tatsächliches physikalisches Löschen mit einem darauffolgenden Zusammenschieben der Datei würde einen zu großen Zeitaufwand erfordern. Je nach Verwendungsfall kann man die zugehörigen Datensätze in den Indexdateien entweder direkt löschen oder auch mit einer Kennung versehen. Im zweiten Fall kann der gelöschte Datensatz sofort wieder wiederhergestellt werden.

Einer Hauptdatei muß mindestens *eine* Indexdatei zugeordnet sein, bei der die Schlüssel *eindeutig* sind - daß heißt, ein bestimmter Schlüsselwert kommt nur ein einziges Mal in der Indexdatei vor. Solche eindeutigen Schlüssel nennt man *Primärschlüssel.* Nur über einen Primärschlüssel kann ein bestimmtes Datenelement *eindeutig* gesucht und identifiziert werden. (Im obigen Beispiel würde man wohl *Nr.* als Primärschlüssel wählen, während man *Name* und *Umsatz* als *Sekundärschlüssel* bezeichnet, da sie durchaus mehrfach mit dem gleichen Wert in den jeweiligen Indexdateien vorkommen können.)

Grundlegende Aufgabe — kundind.cpp

Das nachfolgenden Programmbeispiel *KundInd.cpp* realisiert ein Datenbank-Objekt, das eine Hauptdatei (z.B. *KUNDEN.DAT*) und drei Indexdateien beinhaltet. Die Index-Datensätze werden bei diesem Beispielprogramm jedoch nicht in echten Dateien, sondern direkt im Hauptspeicher in drei V*ektoren* verwaltet.

```
Name der Datei (z.B. Kunden.dat): kunden.dat
Datei existiert schon, neu anlegen (j/n)? n

Zur Menueanzeige Return druecken!
Datei in Bearbeitung: kunden.dat

=====================================================
0   Ende:          Programm beenden
1   Einfuegen:     Neue Datensaetze in Datei einfuegen
2   Auflisten:     Alle Datensaetze sortiert ausgeben
3   Suchen:        Datensatz suchen und ausgeben
4   Neu:           Neue Datei bearbeiten
=====================================================
Bitte waehlen:
```

Die verwendete Hauptdatei *kunden.dat* wurde in einem früheren Programm *Kunden.cpp* schon einmal verwendet. In der nun vorliegenden Version von *KundInd.cpp* kann das *Auflisten* und *Suchen* nach einem wählbaren Sortier- bzw. Suchkriterium *Nummer, Name* oder *Umsatz* erfolgen. Nach dem Eingabe des Dateinamens wird die Datei, falls sie schon bestand, zum Lesen geöffnet. Alle Datensätze werden eingelesen und die drei Indexvektoren werden erzeugt (*NrIndDatei, NameIndDatei, UmsIndDatei*).

Beim *Einfügen* eines Datensatzes wird der neue Satz an die Hauptdatei angehängt - die drei Indexvektoren erhalten je ein neues Element und werden neu sortiert. Beim *Auflisten* wird - je nach Wahl des Sortierkriteriums - der entsprechende Indexvektor sequentiell durchlaufen. Der jeweils zugeordnete Datensatz der Hauptdatei wird im Direktzugriff geöffnet und ausgegeben.

Das Sortieren der Indexvektoren erfolgt mit dem *Quicksort*-Algorithmus. Die Datensätze werden *binär* in den Indexvektoren gesucht. Beim Betrachten des Quellcodes von *KundInd.cpp* fallen diverse Verbesserungsmöglichkeiten ins Auge:

- Sowohl die Sortier- als auch die Suchmethode sind jeweils dreifach ausgeführt, abhängig von den Datentypen in den jeweiligen Indexvektoren. Hier könnte man den Quellcode auf je *eine* Methode für Suchen bzw. Sortieren reduzieren, wenn man *Funktionsschablonen* verwendet.
- Bei jeder Änderung müssen die Indexvektoren *komplett* neu sortiert werden. Das nimmt bei umfangreichen Dateien sehr viel Zeit in Anspruch. Die Sortiervorgänge können beschleunigt werden, wenn man Datenstrukturen verwendet, die ein schnelles *sortiertes Einfügen* erlauben (z.B. *Listen*, siehe späteres Kapitel).
- Bei realen Datenbanken mit sehr vielen Datensätzen müssen tatsächlich echte Indexdateien erzeugt werden, die entweder teilweise in den Hauptspeicher gelesen oder vollständig extern als Dateien sortiert werden.

Weiterführende Aufgabe	kundind2.cpp

Ergänzen Sie *KundInd.cpp* zu *KundInd2.cpp*, indem Sie einen Menüpunkt *Ändern* hinzufügen, mit dem beliebige, bereits bestehende Datensätze in allen Feldern geändert werden können. Der Datensatz, der geändert werden soll, soll nach einem beliebigen Kriterium *Nummer*, *Name* oder *Umsatz* gesucht werden können.

Weiterführende Aufgabe	kundind3.cpp

Ergänzen Sie *KundInd2.cpp* zu *KundInd3.cpp*, indem Sie Menüpunkte zum *Löschen* eines Datensatzes und zum *Packen* der gesamten Datei hinzufügen. Das Löschen soll so realisiert werden, daß der zu löschende Datensatz lediglich in den Indexvektoren gelöscht wird, in der Hauptdatei aber zunächst erhalten bleibt. Erst beim *Packen* sollen alle ungelöschten Datensätze der Hauptdatei in eine temporäre Datei geschrieben werden. Danach wird die Ursprungsdatei gelöscht und die temporäre Datei erhält den Namen der Hauptdatei.

2.11 Zeiger

Lerninhalte

❶ Speicheradressen und Zeiger
❷ Nullzeiger und typfreie Zeiger
❸ Konstante Werte, konstante Zeiger
❹ Zeiger und Arrays
❺ Zeigerarithmetik
❻ Char-Arrays und C-Strings
❼ C-Arrays als formatfreie Byteströme
❽ Funktionen zur Verarbeitung von C-Strings

Lerninhalte

Alle Datenstrukturen im Hauptspeicher eines Rechners, deren Speicherplatz zur Programmlaufzeit *dynamisch* erzeugt und gelöscht werden kann, werden über *Zeiger* realisiert. Zeiger sind unverzichtbare Elemente einer höheren Programmiersprache.

Speicheradressen und Zeiger

Zu jedem bisher behandelten Datentyp (*int, float, char,* aber auch für selbstdefinierte Typen wie *enum, struct, class*) existiert ein entsprechender *Zeigertyp*. Eine Zeigervariable speichert (technisch gesehen) die *Anfangs*adresse einer Daten- oder Programmstruktur.

```
int Var;    // Variable Var vom Typ Integer
int* pVar;  // Zeigervariable pVar vom Typ Zeiger auf Integer
```

Zeigervariablen erhalten in C und C++ im Variablennamen oft ein vorangestelltes *p* oder *ptr* als Namensbestandteil. Wenn z.B. *pNr* ein Zeiger auf einen Integerwert darstellt, bedeutet das, daß in der Speicheradresse, auf die *pNr* zeigt, die Speicherung eines Integerwerts beginnt. Ein Zeiger erhält bei der Deklaration zunächst eine *beliebige* Adresse, genauso wie alle anderen Variablen, die nicht initialisiert werden.

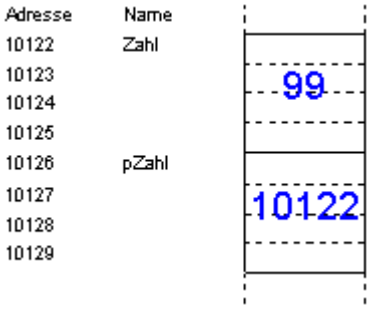
Bei *Deklarationen* stellt ein Stern *, der der Typbezeichnung *nachgestellt* wird, eine Abkürzung der Bezeichnung „Datentyp ‘Zeiger auf’ ...“ dar. Bei *Anweisungen* jedoch bedeutet ein Stern *, der einer Variablen *vorangestellt* wird, eine *Dereferenzierung*. Die *Dereferenzierung* einer Zeigervariablen ergibt die „Inhaltsvariable“ der betreffenden Speicheradresse.

Ein der Typbezeichnung nachgestelltes Zeichen & hingegen stellt bei *Deklarationen* einen *Referenzoperator* dar. Das bedeutet, daß die deklarierte Variable den gleichen Speicherplatz belegt wie eine schon vorher definierte. (Vgl. die gleiche Bedeutung bei Funktionsdeklarationen als *Referenzparameter* bzw. *Variablenparameter*.) Bei *Anweisungen* jedoch bedeutet der Operator &, wenn er einer Variablen *vorangestellt* wird, eine Abkürzung von „Adresse der Variablen“ bzw. „Zeiger auf die Variable“.

Die beiden Operatoren * und & haben also jeweils *zwei unterschiedliche* Bedeutungen, je nachdem, ob sie in Deklarationen oder Anweisungen verwendet werden. Diese Varianten sind in C++ äußerst wichtig, sie müssen genau auseinandergehalten werden:

```
// Deklarationsteil:
int Var;
int& Var2 = Var; // Referenz, das heißt:
                 // Var2 und Var belegen den gleichen Speicherplatz
int* pVar;       // pVar wird als Zeiger auf int deklariert

// Anweisungsteil:
pVar = &Var;  // pVar erhält die Adresse von Var (pVar zeigt auf Var!)
*pVar = 100;  // Dereferenzierung: Die Variable, deren Adresse in pVar
              // gespeichert ist, wird auf 100 gesetzt
              // => wirkt im Beispiel wie Var = 100
```

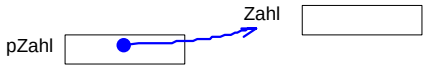
Bevor man einen Zeiger benutzt, muß ihm eine sinnvolle Adresse zuweisen, ansonsten werden rein zufällige Speicherstellen ausgelesen oder gar geändert! Zur Verdeutlichung der Zusammenhänge nehmen wir an, daß eine Variable *Zahl* mit dem Wert 99 initialisiert wird. Wir nehmen weiter an, daß die Variable bei der Initialisierung ab der Speicheradresse 10122 gespeichert wird. (*Jede* Variable besitzt implizit eine Speicheradresse und damit einen Zeiger. Der Wert dieses Zeigers wird jedoch üblicherweise vom EDV-System automatisch vergeben und ist dem Programmierer somit nicht bekannt.)



Bei den meisten EDV-Systemen erhält ein einzelnes Byte einen eigenen Speicherplatz. Nehmen wir an, daß die Integervariable 4 Bytes an Speicherplatz benötigt, dann reicht sie von der Adresse 10122 bis zur Adresse 10125.

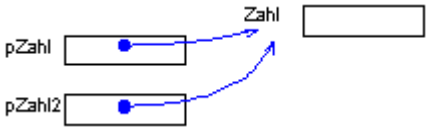
Nun wird mit *int* pZahl* ein Zeiger *pZahl* deklariert, aber nicht initialisiert. Die Speicheradresse, ab der die Zeigervariable automatisch vom System gespeichert wird, sei 10126.

Direkt nach der Deklaration hat *pZahl* einen unbestimmten Wert. Erst nach einer Zuweisung *pZahl = &Zahl* hat *pZahl* den Wert 10122 (nämlich genau die Speicheranfangsandresse der Variablen Zahl). Vereinfacht können wir das so darstellen:



Nun deklarieren wir einen zweiten Zeiger *pZahl2* und richten ihn durch Initialisierung mit *&Zahl* ebenfalls auf *Zahl:*

```
int* pZahl2 = &Zahl;
```



Der Wert von *Zahl* ist jetzt über mehrere Zeiger zugreifbar. Die Variablennamen **pZahl* und **pZahl2* sind nun *Aliasnamen* für *Zahl.*

```
int Zahl;
int* pZahl;
int* pZahl2 = &Zahl;

pZahl = &Zahl;
Zahl = 99;        // Die nebenstehenden drei Anweisungen
*pZahl = 99;      // bewirken dasselbe.
*pZahl2 = 99;
```

Leider sind bei Zeigerdeklarationen viele *verschiedene* Schreibweisen möglich, die aber alle dasselbe bewirken:

```
int* pZahl;  // 1) Übliche C++-Schreibweise, wird dringend empfohlen!
int *pZahl;  // 2) Klassische C-Schreibweise, wirkt genauso.
int * pZahl; // 3) Ganz schlecht, wirkt aber trotzdem genauso.
int*pZahl;   // 4) Am allerschlechtesten, aber immer noch das Gleiche!
```

Leider führen diese verschiedenen Schreibmöglichkeiten manchmal zu Fehlinterpretationen:

```
int *pZahl, Zahl; // 5) pZahl ist "Zeiger auf int", Zahl ist "int"
int* pZahl, Zahl; // 6) Das ist dasselbe, aber noch schlechter lesbar.
                  //    (* bezieht sich immer nur auf direkte nachfolgende
                  //    Variable). Besser ist diese Schreibweise:
int* pZahl;       // 7) Das ist besser lesbar.
int Zahl;         // 8) Das ebenso.

int* pZahl1, *pZahl2; //  9) Zwei Zeiger auf int (etwas seltsam)
int *pZahl1, *pZahl2; // 10) Das gleiche, aber Verwechslungsgefahr
                      //     mit Gebrauch von * bei Anweisungen
```

In der klassischen C-Denkweise kann man sich eine Deklaration *int *pZahl* auch in der Form *int (*pZahl)* vorstellen. Demnach ist die „Inhaltsvariable“ *(*pZahl)* vom Typ Integer und *pZahl* ist ihr Zeiger. In C++ sind diese Schreibweisen nach wie vor möglich und gültig. Um eine bessere Lesbarkeit zu erreichen, sollte man jedoch gemischte Zeiger- und Nichtzeigerdeklarationen in einer Zeile (wie oben bei 4 und erst recht wie bei 5) vermeiden.

L2 Nullzeiger und typfreie Zeiger

Belegt man in C++ einen Zeiger mit dem Wert 0 oder 0L (long), dann ist der Zeigerwert klar definiert und gleichzeitig wird angezeigt, daß der Zeiger auf *nichts* (also nicht auf ein gültiges Datenobjekt) zeigt. Ein solcher *Nullzeiger* ist als Konstante *NULL* definiert und kann in logischen Vergleichen abgefragt werden (Auch in Standard-C ist die Konstante *NULL* definiert, der tatsächliche Zahlenwert für den Nullzeiger ist dort jedoch systemabhängig.)

Der „typfreie“ Zeigertyp *void** hat die Bedeutung „Zeiger auf einen beliebigen Datentyp“. Das bedeutet, daß beliebige Zeigervariablen auf einen *void**-Zeiger zugewiesen werden können. Ansonsten sind jedoch Zeiger, die auf unterschiedliche Typen zeigen, in C++ *nicht zuweisungskompatibel* (wohl aber in Standard-C).

```
char Zeichen = 'A';
char* pZeichen; // Zeiger auf char
void* pAlles;   // Zeiger auf beliebigen Datentyp

pZeichen = &Zeichen;
pAlles = pZeichen; // Das geht!
pAlles = &Zeichen; // Das auch!
pZeichen = pAlles; // Das geht nicht in C++ (nur in Standard-C)
pZeichen = (char*) pAlles; // So geht es wieder in C++
```

Bei Zugriffen auf verzeigerte Variablen ist zu beachten, daß der Speicherplatz für Variablen nach Verlassen des Blocks, in dem sie deklariert wurden, automatisch wieder freigegeben wird:

```
int Zahl = 8;
int* pZahl = &Zahl;
*pZahl = 16;       // Gleichbedeutend mit Zahl = 16
{ // Neuer Block
  int i = 7;
  pZahl = &i       // (*pZahl) ist jetzt gleich 7
  // usw.
}
*pZahl = 19; // ungültig, denn pZahl (=&i) zeigt nicht mehr auf einen
             // definierten Variablenbereich.
```

Konstante Werte, konstante Zeiger

Syntaktisch gewöhnungsbedürftig ist die Deklaration der folgenden Konstanten:

- Zeiger auf einen konstanten Wert

  Der Inhaltswert ist konstant, die Adresse, an der er gespeichert wird, ist jedoch veränderlich.

- Konstanter Zeigerwert (das bedeutet: unveränderlicher Adreßwert)

  Der Wert einer Zeigervariablen ist eine Konstante, d.h. an einer festen Speicheradresse können unterschiedliche Inhaltswerte stehen.

- Konstanter Zeiger auf konstanten Inhaltswert

  Sowohl ein Wert als auch seine Speicheradresse, die in einem Zeiger festgehalten wird, sind konstant.

```
char z;                     // einzelnes Zeichen, veränderbar
const char* pZeich1;        // Zeiger auf ein konstantes Zeichen
                            // Nicht: konstanter Zeiger auf ein Zeichen!!

char* const pZeich2;        // konstanter Zeiger auf ein Zeichen

const char* const pZeich3; // konstanter Zeiger auf ein konst. Zeichen
char const* const pZeich3; // ebenso, aber schlecht lesbar
```

Mit diesen Deklarationen seien die folgenden Zuweisungen betrachtet:

```
pZeich1=&z; // erlaubt, da der Zeiger verändert werden darf
pZeich1=z;  // nicht erlaubt, da das Zeichen selbst nicht verändert werden darf
pZeich2=&z; // nicht erlaubt, daher wäre bei der Deklaration eine Initialisierung
            // notwendig gewesen
pZeich2=z;  // erlaubt, da das Zeichen verändert werden darf
pZeich3=&z; // verboten, weil der Zeiger nicht verändert werden darf
pZeich3=z;  // verboten, weil das Zeichen selbst nicht verändert werden darf
```

Zeiger und Arrays

In C++ sind Zeiger und Arrays eng verwandt. Der *Name eines Arrays* ist syntaktisch gleichbedeutend mit einem *Zeiger auf das erste Arrayelement* (und wird vom Compiler auch so interpretiert und als Zeigerwert gespeichert).

```
/* Programm Zeiger1.cpp
   Arrays und Zeiger */
#include <iostream>

void main() {
  int Feld[] = {10,20,30,40}; // Integer-Array
  int* p1, *p2, *p3; // dreimal Zeiger auf Integer

  p1 = Feld;      // p1 und Feld haben jetzt identische Werte
  p2 = &Feld[0];  // p2 und Feld haben jetzt identische Werte
  p3 = &Feld[4];  // p3 zeigt direkt hinter das letzte Array-Element
  cout << "Feld: " << Feld << ", p1: " << p1 << ", p2: " << p2
       << ", p3: " << p3;
}
```

```
Feld: 0x0064fdf4, p1: 0x0064fdf4, p2: 0x0064fdf4, p3: 0x0064fe04
```

Die Bildschirmausgabe des kleinen Testprogramms *Zeiger1.cpp* beweist, daß Arraynamen wie Zeiger interpretiert werden. Das Array *Feld* sowie die damit synchronisierten Zeiger *p1 und p2* enthalten den gleichen (Hex-)Adreßwert *0x64fdf4*.

Der Zeiger *p3* zeigt hinter das letzte Element von *Feld und* enthält einen um 0xf (=16) höheren Wert (Ein *int*-Wert belegt 4 Bytes, das Array *Feld* hat 4 Elemente => 4*4=16). Ein solcher Zeiger hinter das *letzte* definierte Array-Element ist selbst noch definiert, darf jedoch, da sein *Inhalt* undefiniert ist, nicht mehr zum Lesen und Schreiben von Array-Elementen benutzt werden.

Nun testen wir in *Zeiger2.cpp* einige Zuweisungoperationen:

```
/* Programm Zeiger2.cpp
   Arrays und Zeiger - Zuweisung bei Arrays */
#include <iostream>

void main() {
  int Feld[] = {10,20,30,40}; // Integer-Array
  int* p1, *p2, *p3; // dreimal Zeiger auf Integer
  float p4;
  float Feld2[] = {3.14,6.28,9.13};

  p1 = Feld;      // p1 und Feld haben jetzt identische Werte
  p2 = &Feld[0];  // p2 und Feld haben jetzt identische Werte
  p3 = &Feld[4];  // p3 zeigt direkt hinter das letzte Array-Element
  cout << "Feld: " << Feld << ", p1: " << p1 << ", p2: " << p2
       << ", p3: " << p3 << endl;
  p2 = p3;        // 1. funktioniert!
  p1 = &Feld[0];  // 2. funktioniert!
  p1 = &Feld[2];  //    funktioniert auch!
  p1 = &Feld2[0]; //    => Compilerfehler
                  //       "Cannot convert 'float*' to 'int*'
  p1 = Feld[2];   // 3. => Compilerfehler "Cannot convert 'int' to 'int*'
  p4 = p2;        // 4. => Compilerfehler "Illegal use of floating point"
  Feld = p3;      // 5. => Compilerfehler "Lvalue required"
  Feld[2] = 80;   // Das geht jedoch problemlos:
                  // Array-Elemente sind Linkswerte!
  cout << "Zweimal der 3. Arraywert: " << Feld[2] << ", " << *(Feld+2);
```

```
}
```

Der Kompiliervorgang erbringt die folgenden Ergebnisse:

1. Zeiger gleichen Typs kann man untereinander zuweisen (*p2 = p3*, beide vom Typ int*).
2. Zeigern kann man die Adressen (d.h. die *Zeiger*) von Arrayelementen zuweisen (mit dem Operator &). Die Zeigertypen müssen jedoch mit den Elementtypen kompatibel sein. (*p1 = &Feld[x]* funktioniert nur, wenn die Operanden beide vom gleichen Typ sind).
3. Einer Zeigervariable kann keine Inhaltsvariable mit gleichem Grundtyp zugewiesen werden (*int* und *int** sind nicht zuweisungskompatibel).
4. Zwei Zeiger unterschiedlichen Typs sind nicht zuweisungskompatibel (z.B. *int** und *float**).
5. Ein Arrayname repräsentiert einen *konstanten Zeiger*: Dem Arraynamen kann kein Wert zugewiesen werden (er ist kein *Lwert).*

   Die letzte Erkenntnis muß näher erläutert werden: Eine Deklaration der Form *int Feld[]* bewirkt, daß eine Zeigervariable *Feld* vom Typ *int** deklariert wird. Der Zeigerwert von *Feld* ist zunächst unbestimmt. Erst nach Angabe der Elementeanzahl (z.B. mit *int Feld[5]*) oder bei entsprechender Initialisierung des Feldes wird Speicherplatz für die einzelnen Arrayelemente erzeugt und der Zeiger *Feld* ist als *Anfangsadresse* des Arrays *Feld* definiert (das heißt, *Feld* wird wie *&Feld[0]* behandelt).

```
int FeldA[];  // Das alleine reicht nicht aus und ergibt Compilerfehler
int FeldB[5]; // Jetzt wird Speicherplatz erzeugt für
              // die fünf Arrayelemente und int* Feld stellt die
              // Array-Anfangsadresse dar.
int FeldC[] = {13,14,20}; // So geht´s auch: Durch die Initialisierung
                          // kann der Compiler bestimmen, daß Speicher-
                          // platz für int FeldC[3] erzeugt werden soll
```

   Da der Zeigerwert *Feld* nur als Adresse von *Feld[0]* definiert ist, kann man den Wert von *Feld* nicht eigenständig ändern. Deshalb ist ein Arrayname immer als *konstanter Zeiger auf den Elementtyp* deklariert. Daher kann man auf einen Arraynamen auch *nichts zuweisen*. Ein Arrayname darf demnach nie *links* von einem „=“ stehen, er ist *kein Linkswert* (engl. *Lvalue*).

   Davon muß unterschieden werden: Die *Elemente* von Arrays sind Linkswerte, im Gegensatz zum Arraynamen selbst ! Auf ein Arrayelement können Werte zugewiesen werden, es darf links von einem „=“ stehen.

```
int Feld[5]; // wirkt in etwa wie: int* const Feld;
             // zusammen mit:      int Feld1, Feld2, Feld3, Feld4, Feld5

Feld = /* irgendein int* */ // Das geht nie, da konstanter Zeiger!
Feld[5] = 6;                // Das geht immer!
```

L5 Zeigerarithmetik

Die letzte Zeile von *Zeiger2.cpp* demonstriert, daß die Ausdrücke *Feld[2]* und **(Feld+2)* identisch sind. Daraus ergibt sich, daß ein inkrementierter Zeiger nicht auf die nächste Speicheradresse, sondern auf das *nächste Arrayelement* zeigt.

```
int Feld[5];
int* pFeld;

// Identische Ausdrücke:
cout << Feld[0] << "  " << *Feld;     // ergibt Feldelement Nr. 0
cout << Feld[1] << "  " << *(Feld+1); // ergibt Feldelement Nr. 1
cout << Feld[0] << "  " << *(Feld+2); // ergibt Feldelement Nr. 2, usw.

pFeld = Feld; cout << *pFeld; // ergibt Feld[0]
pFeld++; cout << *pFeld;      // ergibt Feld[1]
pFeld++; cout << *pFeld;      // ergibt Feld[2], usw.
```

Wenn ein Zeiger inkrementiert oder dekrementiert wird, zeigt er auf die Adresse des nächsten oder vorherigen Arrayelements. Dieses Verhalten bezeichnet man als *Zeigerarithmetik*.

```
int Feld[5];
int* pFeld = Feld;

pFeld++; // erhöht den Zeiger pFeld um sizeof(int) Bytes

/* Programm Zeiger3.cpp
   Arrays und Zeiger - Zeigerarithmetik */
#include <iostream>

void main() {
  const Anzahl = 5;
  int Feld[Anzahl] = {10,20,30,40,50};  // konstanter Zeiger
  int* pFeld;                           // veraenderbarer Zeiger

  pFeld = Feld;
  cout << "Feldelemente anzeigen:" << endl;
  for (int i=0; i<Anzahl; i++) {
    cout << i << ":" << *pFeld << "  ";
    pFeld++;
  }
  cout << "\nund wieder rueckwaerts:" << endl;
  for (int i=Anzahl-1; i>=0; i--) {
    pFeld--;
    cout << i << ":" << *pFeld << "  ";
  }
}
```

```
Feldelemente anzeigen:
0:10  1:20  2:30  3:40  4:50
und wieder rueckwaerts:
4:50  3:40  2:30  1:20  0:10
```

L6 Char-Arrays und C-Strings

Char-Arrays besitzen sämtliche alle Eigenschaften der oben besprochenen Arrays. Es gibt jedoch eine Besonderheit: Wenn Char-Arrays mit Zeichenliteralen (“xyz“) initialisiert oder mit der Methode *cin* über die Tastatur eingegeben werden, hängt der Compiler automatisch als letztes Arrayelement das Zeichen ‘\0’ an (das 0. ASCII-Zeichen). Wenn man diese „Sonderbehandlung“ verhindern will, muß das Array ausdrücklich mit Einzelzeichen initialisiert werden. Die Methode *cout* gibt Char-Arrays bis zum letzten Zeichen vor dem ‘\0’ auf dem Bildschirm aus.

```
/* Programm CArray.cpp
   Behandlung von Char-Arrays */
#include <iostream>

void ArrayAusgabe(char f[], int Zahl) {
  for (int i=0; i<Zahl; i++)
    cout << f[i] << " ";
  cout << endl;
}

void main() {
  char CArray1[] = "CArray1"; // [0]..[6], an [7] steht '\0'
  char CArray2[9];
  char CArray3[9] = "ABCD";   // wird als "ABCD\0" initialisiert
  char CArray4[9] = {'E','F','G','H'}; // ohne '\0' initialisieren

  cout << "CArray2 eingeben: ";
  cin >> CArray2; // Nach Eingabe wird '\0' angehängt.
  cout << "CArray1 als 8 Einzelzeichen [0]..[7]: ";
  ArrayAusgabe(CArray1,8);
  cout << "CArray2 als 9 Einzelzeichen [0]..[8]: ";
  ArrayAusgabe(CArray2,9);
  cout << "CArray3 als 9 Einzelzeichen [0]..[8]: ";
  ArrayAusgabe(CArray3,9);
  cout << "CArray4 als 9 Einzelzeichen [0]..[8]: ";
  ArrayAusgabe(CArray4,9);
  cout << "\nAlle Arrays nochmal mit cout ausgeben:\n";
  cout << "CArray1: " << CArray1 << endl;
  cout << "CArray2: " << CArray2 << endl;
  cout << "CArray3: " << CArray3 << endl;
  cout << "CArray4: " << CArray4 << endl;
}
```

```
CArray2 eingeben: Test
CArray1 als 8 Einzelzeichen [0]..[7]: C A r r a y 1
CArray2 als 9 Einzelzeichen [0]..[8]: T e s t   É @   _
CArray3 als 9 Einzelzeichen [0]..[8]: A B C D
CArray4 als 9 Einzelzeichen [0]..[8]: E F G H

Alle Arrays nochmal mit cout ausgeben:
CArray1: CArray1
CArray2: Test
CArray3: ABCD
CArray4: EFGH
```

Die Bildschirmausgabe des Programms zeigt folgendes Ergebnis:

- Wenn Char-Arrays *zeichenweise* mit *cout* ausgegeben werden, erkennt man, daß das Array nur bis zu dem Endezeichen '\0' mit eindeutigen Zeichen definiert ist. (Das Zeichen '\0' ergibt keine Bildschirmausgabe, daher kann man nicht erkennen, daß es bei der Ausgabe von *CArray4* unterbleibt. Dies kann jedoch mit dem Debugger nachvollzogen werden.)
- Gibt man ein ganzes Char-Array mit *cout* aus, so erfolgt die Bildschirmausgabe grundsätzlich nur bis zum letzen Zeichen vor '\0'.
- Gibt man als *CArray2* mit *cin* einen Text ein, der länger als 9 Zeichen ist, erhält man die folgende (verblüffende) Bildschirmausgabe:

```
CArray2 eingeben: 1234567890123
CArray1 als 8 Einzelzeichen [0]..[7]: 3   r r a y 1
CArray2 als 9 Einzelzeichen [0]..[8]: 1 2 3 4 5 6 7 8 9
CArray3 als 9 Einzelzeichen [0]..[8]: A B C D
CArray4 als 9 Einzelzeichen [0]..[8]: E F G H

Alle Arrays nochmal mit cout ausgeben:
CArray1: 3
CArray2: 1234567890123
CArray3: ABCD
CArray4: EFGH
```

  Offensichtlich werden im Hauptspeicher „falsche" Bereiche durch den Rest „0123" der Zeichenfolge, die für den Speicherplatz von *CArray2* zu lang ist, überschrieben! Im obigen Fall wird der Wert von CArray1 verfälscht. (Nach '3' liefert *cin* ein Zeichen `\0` zurück, das als Endezeichen für *CArray1* interpretiert wird.) Daher muß, wenn C-Arrays mit *cin* eingelesen werden, sichergestellt sein, daß nicht mehr Zeichen eingegeben werden, wie die definierte Arraylänge verkraftet. Hierzu sei auf das Programmbeispiel *String1.cpp* verwiesen, das in einem früheren Kapitel bereits besprochen wurde. Dort wird eine sichere Einlesemethode mittels *getline* beschrieben, die überzählige Zeichen verwirft.

```
const int max = 20;
char Zeile[max];

cin.getline(Zeile,max); // Sicher: liest maximal (max-1) Zeichen
```

Ein Char-Array, das das Zeichen '\0' als letztes Element enthält, bezeichnet man als *C-String*. C-Strings stellten in der Programmiersprache C die grundlegenden (und dort einzig verfügbaren) Datenstrukturen zur Bildung von Zeichenketten dar. Die komfortable *String*-Klasse von C++ ist auf der Basis einfacher C-Strings definiert.

Der Datentyp für einen C-String ist „Zeiger auf Zeichen", also *char** (bekanntlich dasselbe wie *char ... []*, siehe oben). Bei der Deklaration von C-Strings und der Initialisierung mit einem Zeichenliteral müssen drei Fälle unterschieden werden:

- *char str[] = "XXX";* oder *char str[Laenge] = "XXX";*

  Mit dieser Initialisierungsmethode wird ein C-String in einem festliegenden Speicherplatz erzeugt. Es wird. Speicher für die einzelnen Zeichen plus `\0` reserviert. Der Zeiger *str* ist kein Lwert und kann nicht verändert werden. (Auf *str* als Zeiger kann nur lesend zugegriffen werden.) Dennoch sind die einzelnen String-*Zeichen* ihrerseits Lwerte und können verändert werden. (Insofern wird nicht direkt eine *Stringkonstante* erzeugt.)

- *char* str = "XXX";*

  Diese Initialisierungart erzeugt einen vollständig variablen *C-String*. Hier wird zusätzlich zum Speicher für die einzelnen Zeichen noch Speicherplatz für eine *Zeigervariable str* (vom Typ *char**)

erzeugt. Sowohl der Zeiger *str* als auch die einzelnen Zeichen des Strings sind Lwerte und können verändert werden. Wird *str* geändert, ist der ursprüngliche Speicherplatz, der dem String*inhalt* zugewiesen wurde, nicht mehr zugreifbar, wenn der Zeiger *str* nicht vorher in einer anderen Zeigervariablen vom Typ *char** gesichert wurde.

- *char* str;*

  Hier wird nur Speicherplatz für die Zeigervariable *str* erzeugt. Solchermaßen deklarierte C-Strings eignen sich nur zur Zuweisung anderer Stringzeiger. Es gibt keinen reservierten Speicherplatz für einzelne Stringzeichen! Daher dürfen auf eine derart deklarierte Stringadresse auf keinen Fall mit *cin* ein String eingelesen oder mit *str[x] = Zeichen* ein (überhaupt nicht definierter) Speicherplatz beschrieben werden!

```
/* Programm CStrings.cpp
   Initialisierung und Ausgabe von C-Strings */
#include <iostream>

  char* str1 = "AAA";   // Compiler intialisiert 4 Zeichen mit "ABC\0"
                        // und reseviert Speicherplatz für Zeiger str1
  char str2[] = "BBB";  // genauso, aber ohne Speicher für Zeiger str2
  char str3[4] = "CCC"; // auch ohne Speicher für Zeiger str3
  char str4[3] = "DDD"; // gefährlich: '\0' bleibt weg!
  char str5[9] = "EEE"; // 9 Zeichen Platz, aber nur 4 belegt
  char* str6;           // reserviert nur Speicher für Zeiger

void main() {
  cout << "C-Strings ausgeben:\n";
  cout << "str1: " << str1 << "  ";
  cout << "str2: " << str2 << "  ";
  cout << "str3: " << str3 << "  ";
  cout << "str4: " << str4 << "  ";
  cout << "str5: " << str5 << "  ";

  str1 = str2;   // funktioniert, da str1 ein Lwert ist
  // str2 = str3;   // funktioniert nicht, str3 ist kein Lwert
  str2[1] = 'X'; // Das geht wiederum,
                 // Die einzelnen Zeichen sind veränderbar
  cout << "\nstr1 und str2 nach str2[1]='X':\n";
  cout << "str1: " << str1 << "  ";
  cout << "str2: " << str2 << "  ";

  str1[1] = 'Y'; // Das geht auch.
  cout << "\nstr1 und str2 nach str1[1]='Y':\n";
  cout << "str1: " << str1 << "  ";
  cout << "str2: " << str2 << "  ";

  // str6[3] = 'Z'; cout << str6;
  // Das wäre beides fatal, denn str6 hat keinen definierten Wert,
  // demnzufolge würde irgendwo in den Speicher ein 'Z' geschrieben.
  // Nach folgender Anweisung ginge es allerdings: str6 = str1;
 }
```

```
C-Strings ausgeben:
str1: AAA  str2: BBB  str3: CCC  str4: DDDEEE  str5: EEE
str1 und str2 nach str2[1]='X':
str1: BXB  str2: BXB
str1 und str2 nach str1[1]='Y':
str1: BYB  str2: BYB
```

Die Arrays *str1*, *str2*, *str3* und *str5* sind echte, nullterminierte C-Strings. Bei der Variablen str4 fehlt das abschließende Nullzeichen, daher gibt die Methode *cout* den (vermeintlichen) C-String bis zum nächsten erreichbaren Nullzeichen (hier nach *str3)* aus. Man kann solche Effekte mit Absicht herbeiführen, muß

sich aber darüber im klaren sein, daß als *str4* ein Char-Array ohne Nullterminierung erzeugt wurde, kein C-String!

Die Zuweisung *str1 = str2* setzt die Zeigervariable *str1* auf die Speicheradresse von *str2*. Damit ist der ursprüngliche Wert von *str1* nicht mehr zugreifbar. Der Stringinhalt "AAA" ist verloren, weil die Speicheradresse vor der Zuweisung nicht in einer anderen Zeigervariablen (z.B. *str6*) gesichert wurde.

Nun zeigen *str1* und *str2* auf die gleiche Speicheradresse. Wird ein Zeichen geändert, wirkt sich das auf die Ausgabe beider Strings aus

L7 C-Arrays als formatfreie Byteströme

C-Arrays eignen sich hervorragend, um Variablen mit *beliebigem* Typ in einen *formatfreien Bytestrom* zu übertragen, denn standardmäßig entspricht ein *char* genau einem Byte im Hauptspeicher. Jede Variable *x* eines beliebigen Datentyps kann grundsätzlich auch als Bytestrom *(char*)&x* von *sizeof(x)* Bytes Länge angesprochen werden. Der Ausdruck *(char*)&x* repräsentiert hierbei einen Zeiger auf die Anfangsadresse der Variablen *x*, während *sizeof(x)* die Länge des Bytestroms in Bytes angibt.

Ist die Variable *x* der Name für ein Array, stellt *x* schon direkt einen Zeiger dar. Dann muß man als Ausdruck für die Speicheranfangsadresse von x die Form *(char*)x* wählen, d.h. dann entfällt der Zeigeroperator *&*. Formatfreie Byteströme werden z.B. bei der Verarbeitung von Datentypen und Objekten in Dateien verwandt (vgl. das Kapitel zur Dateiverarbeitung).

L8 Funktionen zur Verarbeitung von C-Strings

Zur Bearbeitung von C-Strings gibt es eine Menge vordefinierter Funktionen, die in der Include-Datei *string.h* deklariert sind. Bei reiner C++-Programmierung sind die C-String-Funktionen zwar nicht unbedingt notwendig (vgl. das Kapitel über C++-Strings), aber auch unter C++ erwarten viele Funktionen C-Strings als Parameter. Deshalb werden die gängigsten Funktionen zur Verarbeitung von C-Strings erläutert:

(*size_t* ist vom Typ *unsigned int)*

- *char* strcpy (char* dest, const char* src);*

  kopiert die Zeichen aus dem String *src* in den Speicherbereich von String *dest*. Das abschließende Nullzeichen von src wird als letztes Zeichen kopiert. Rückgabewert: *dest*

- *char* strncpy (char* dest, const char* src, size_t maxlen);*

  wie *strcpy*, kopiert jedoch bis zu maxlen Zeichen des über *src* angegebenen Strings in den durch *dest* angegebenen Speicherbereich. Es werden immer genau *maxlen* Zeichen geschrieben. Wenn *src* weniger als *maxlen* Zeichen enthält, wird eine entsprechende zusätzliche Anzahl von Nullzeichen in *dest* angehängt. Wenn *src* mehr als *maxlen* Zeichen umfaßt, bleibt der nach *dest* kopierte String ohne ein abschließendes Nullzeichen. Rückgabewert: *dest.*

- *char* strdup (const char* s);*

  kopiert einen String in einen neuen Speicherbereich. Zuerst wird die Länge des übergebenen Strings *s* bestimmt. Danach wird ein Speicherbereich entsprechender Größe reserviert (*strlen(s) + 1*) und der Inhalt von s wird in diesen neu reservierten Speicherbereich kopiert. Der Programmierer muß den Speicher selbst wieder freigeben, wenn dieser nicht mehr benötigt wird. Rückgabewert: Zeiger auf den neu reservierten Speicherbereich, der den duplizierten String enthält. Falls nicht genug Platz im Speicher ist, ist das Funktionsergebnis NULL.

- *size_t strlen (const char* s);*

  berechnet die Länge des Strings *s.*. Rückgabewert: Anzahl der Zeichen des Strings *s*. Das abschließende Nullzeichen wird dabei nicht mitgezählt.

- *char* strcat(char* dest, const char* src);*

  hängt sämtliche Zeichen von *src* an die bereits in *dest* vorhandenen Zeichen an. Die Länge des resultierenden Strings ist *strlen(dest) + strlen(src)*. Rückgabewert: Zeiger auf den zusammengefügten String.

- *char *strncat(char *dest, const char *src, size_t maxlen);*

  hängt die ersten *maxlen* Zeichen von *src* an die bereits in *dest* vorhandenen Zeichen an. Die Operation wird beendet, wenn entweder *maxlen* Zeichen kopiert wurden oder das Ende von *src* erreicht ist. Zuletzt wird grundsätzlich ein Nullzeichen in *dest* geschrieben. Die maximale Länge des resultierenden Strings beträgst *strlen(dest) + maxlen*. Rückgabewert: *dest.*

- *int strcmp(const char* s1, const char* s2);*

  führt mit *s1* und *s2* einen *unsigned*-Vergleich durch. Beginnend mit dem ersten Zeichen werden die beiden Strings zeichenweise verglichen, bis zwei korrespondierende Zeichen ungleich sind oder das Ende der Strings erreicht wird. Rückgabewert: s1 < s2 : < 0, s1 = = s2 : = = 0, s1 > s2 : > 0.

- *int strncmp (const char* s1, const char* s2, size_t maxlen);*

  führt den gleichen Vergleich wie *strcmp* durch, vergleicht aber maximal die ersten *maxlen* Zeichen von s1 und s2 miteinander. Rückgabewert: wie *strcmp.*

- *int stricmp(const char* s1, const char* s2);*

  vergleicht zwei Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung. Rückgabewert: wie *strcmp.*

- *int strnicmp (const char* s1, const char* s2, size_t maxlen);*

  vergleicht maximal die ersten *maxlen* Zeichen zweier Strings ohne Unterscheidung zwischen Groß- und Kleinschreibung. Rückgabewert: wie *strcmp.*

- *char* strlwr (char* s);*

  konvertiert alle Großbuchstaben des Strings *s* in Kleinbuchstaben. In der Standard-einstellung werden die Großbuchstaben A bis Z in die Kleinbuchstaben a bis z konvertiert. Andere Zeichen (insbesondere Umlaute) werden nicht berücksichtigt. Rückgabewert: Zeiger auf den String *s*.

- *char *strupr(char *s);*

  konvertiert alle Kleinbuchstaben von *s* in Großbuchstaben, ansonsten wie *strlwr*

- *char *strrev(char *s);*

  dreht die Reihenfolge der Zeichen eines String um. Das abschließende Nullzeichen bleibt unverändert. Rückgabewert: Zeiger auf den gespiegelten String.

- *char* strchr (char* s, int c);*

  sucht den String *s* nach dem ersten Auftreten des Zeichens *c* ab. Das abschließende Nullzeichen wird zum Inhalt des String gezählt und als letztes verglichen, so daß ein Aufruf wie *strchr(strs,0)* einen Zeiger auf das abschließende Nullzeichen von *strs* liefert. Rückgabewert: Zeiger auf die erste Fundstelle des Zeichens *c* im String s, bzw. NULL, wenn der String das Zeichen nicht enthält.

- *char *strrchr(char *s, int c);*

  sucht den String *s* nach dem *letzten* Vorkommen des Zeichens *c* ab, wobei die Suche mit dem letzten Zeichen von s beginnt. Das abschließende Nullzeichen wird zum Stringinhalt gerechnet und als erstes verglichen. Rückgabewert: wie bei *strchr.*

- *size_t strspn (const char* s1, const char* s2);*

  sucht einen String *s1* nach dem ersten Teilstring *s2* ab, der Bestandteil von *s1* ist. Liefert die Länge des Teilstrings von *s1* zurück, der ausschließlich aus den in *s2* angegebenen Zeichen besteht.

- *size_t strcspn (const char* s1, const char* s2);*

  liefert die Länge des Teilstrings von *s1* zurück, der keines der Zeichen von *s2* enthält. String *s1* wird zeichenweise gelesen, bis eines der in *s2* enthaltenen Zeichen auftritt. Die Anzahl der in s1 gelesenen Zeichen bildet den Rückgabewert. Das abschließende Nullzeichen wird nicht mitgezählt. Beide Strings werden durch den Suchvorgang nicht verändert.

- *char* strpbrk(char* s1, const char* s2);*

  sucht den String *s1* nach dem ersten Vorkommen eines in *s2* enthaltenen Zeichens ab. Rückgabewert: Zeiger auf die erste Fundstelle eines Zeichens aus s2. Erscheint kein Zeichen aus s2 in s1, dann NULL.

- *char *strtok(char *s1, const char *s2);*

  sucht den String *s1* nach einer von mehreren Zeichenfolgen ab, die in einem zweiten String *s2* definiert sind. *s1* wird als Folge von null oder mehr Zeichenfolgen (Token) betrachtet, die voneinander durch ein oder mehrere der in *s2* definierten Zeichen getrennt sind.

  Der erste Aufruf von *strtok* liefert einen Zeiger auf den Beginn der ersten gefundenen Zeichenfolge in s1 zurück und setzt ein Nullzeichen unmittelbar hinter das Ende der Zeichenfolge. Folgende Aufrufe von *strtok* mit dem Wert NULL anstelle des Arguments s1 liefern die restlichen Zeichenfolgen in s1. Der zur Trennung dienende String s2 kann bei jedem Aufruf von *strtok* beliebig geändert werden.

  Rückgabewert: Zeiger auf den Beginn der gefundenen Zeichenfolge in *s1*. Wenn keine weitere Zeichenfolge gefunden wird, dann NULL.

Grundlegende Aufgabe cstring3.cpp

Schreiben Sie ein Programm *Cstring3.cpp*, das die folgenden C-String-Bearbeitungsfunktionen ausführt:

– für jeden eingegebenen String die Zahl seiner Zeichen anzeigen

– Verkettung zweier Strings bilden

– in einem String ein Zeichen durch ein anderes ersetzen.

```
Ersten String eingeben:
Es war eine Mutter,
Zweiten String eingeben:
 die hatte vier Kinder.
String 1 enthaelt 19 Zeichen
String 2 enthaelt 23 Zeichen
Der verkettete String lautet:
Es war eine Mutter, die hatte vier Kinder.
Der neue String enthaelt 42 Zeichen
Suchzeichen eingeben: a
Ersetzungszeichen eingeben: i
Der neue String lautet:
Es wir eine Mutter, die hitte vier Kinder.
```

(Die Aufgabe cstring4.cpp steht weiter hinten)

Grundlegende Aufgabe cstring5.cpp

Schreiben Sie ein Programm *Cstring5.cpp,* welches einen eingegeben Text sowohl in Klein- als auch in Großbuchstaben umwandelt und dann den Text in umgekehrter Reihenfolge anzeigt. Das Programm gibt eine entsprechende Meldung aus, sofern eine oder mehrer der folgenden Bedingungen zutreffen:

- die Texteingabe enthält keine Großbuchstaben
- die Texteingabe enthält keine Kleinbuchstaben
- die Texteingabe enthält symmetrische Zeichen.

```
Text eingeben: level
Eingabe: level
Kleinbuchstaben: level
Grossbuchstaben: LEVEL
Rueckwaerts: level
Die Eingabe enthaelt keine Grossbuchstaben.
Die Eingabe enthaelt symmetrische Zeichen.
```

Grundlegende Aufgabe cstring6.cpp

Schreiben Sie ein Programm *Cstring6.cpp,* welches ein Zeichenlineal anzeigt und zur Eingabe eines Haupttextes, eines Suchtextes und eines Suchzeichens auffordert. Das Programm soll die Indizes der Positionen anzeigen, an denen der Suchstring bzw. das Suchzeichen im Hauptstring gefunden wurde.

```
Haupttext eingeben:   Das ist der Haupttext
Suchtext eingeben:    der
Suchzeichen eingeben: e
          1         2         3         4
01234567890123456789012345678901234567890
Das ist der Haupttext
Suche nach String der
Uebereinstimmung fuer Index 8
Suche nach Zeichen e
Uebereinstimmung fuer Index 9
Uebereinstimmung fuer Index 18
```

2.12 Zeiger und Funktionen

Lerninhalte

❶ Arrays als Funktionsparameter
❷ Parameterübergabe mit Zeigern
❸ Parameter der Funktion main()
❹ Zeiger auf Strukturelemente, Funktionen und Elementfunktionen

Lerninhalte

Bei der Parameterübergabe in Funktionen spielten Zeiger im klassischen C eine unentbehrliche Rolle. Bei der Sprache C++ wurde eine verbesserte Syntax für Parameter eingeführt, die eine explizite Deklaration von Zeigervariablen oft unnötig macht. Trotzdem sollte man die Verwendung von Zeigern bei Funktionsaufrufen kennen, denn viele C++-Programme benutzen bei Funktionsaufrufen noch klassische C-Techniken.

L1 Arrays als Funktionsparameter

Wie schon in einem früheren Kapitel erläutert, sind Arrays als Funktionsparameter grundsätzlich *Referenzparameter*. Dieses Verhalten wird deutlich, wenn man sich klar macht, daß nur der *Arrayname* an die Funktion übergeben wird (ein konstanter Zeiger auf den Beginn des Arrays im Hauptspeicher). Bei Arrayparametern sind die Ausdrücke *int Feld[]* und *int* Feld* syntaktisch gleich. Eine explizite Angabe der Arraygröße bei der Funktionsdefinition (z.B. als *ArrayAusgabe(int f[5])*) ist syntaktisch falsch und wird vom Compiler abgewiesen.

Daher ist innerhalb eines Funktionsblocks die *Arraygröße* nur dann bekannt, wenn sie explizit als weiterer Funktionsparameter übergeben wurde (z.B. mit *sizeof(Feld)* oder als Anzahl der Arrayelemente).

```
/* Programm Zeiger4.cpp
   Arrays und Zeiger - Arrays als Funktionsparameter */
#include <iostream>

void ArrayAusgabe1(int f[],int ArraySize) {
  for (int i=0; i<ArraySize/sizeof(int); i++)
    cout << i << ":" << f[i] << "  ";
}

void ArrayAusgabe2(int* f, int ArraySize) {
// Wirkung identisch mit ArrayAusgabe1;
  for (int i=0; i<ArraySize/sizeof(int); i++)
    cout << i << ":" << f[i] << "  ";
}

void ArrayAusgabe3(int* f, int Zahl) {
// Wirkung identisch mit ArrayAusgabe1;
  for (int i=0; i<Zahl; i++)
    cout << i << ":" << f[i] << "  ";
}

void main() {
  const Anzahl = 5;
  int Feld[Anzahl] = {10,20,30,40,50};

  cout << "Feldelemente anzeigen:" << endl;
  ArrayAusgabe1(Feld,sizeof(Feld));
  cout << "\nund nochmal:" << endl;
  ArrayAusgabe2(Feld,sizeof(Feld));
  cout << "\nund zum letzten Ma:" << endl;
  ArrayAusgabe3(Feld,Anzahl);
}
```

```
Feldelemente anzeigen:
0:10  1:20  2:30  3:40  4:50
und nochmal:
0:10  1:20  2:30  3:40  4:50
und zum letzten Ma:
0:10  1:20  2:30  3:40  4:50
```

L2 Parameterübergabe mit Zeigern

Im klassischen C war die Syntax für *Referenzen* unbekannt:

```
int x;
int& y = x; // a ist ein anderer Name für b : In ANSI-C ungültig

void tausche(int& a, int& b) { // Verwendung von & als Referenzoperator
int temp                       // bei Parametern in ANSI-C nicht definiert

temp = a;
a = b;
b = temp;
}
```

Wollte man eine Funktion schreiben, die *Referenzparameter* übergibt wie die obige Funktion *Tausche*, behalf man sich folgendermaßen:

```
void tausche(int* a, int* b) {
int temp;

temp = *a;
*a = *b;
*b = temp;
}
```

Die Parameterübergabe erfolgt hier per *Zeiger* auf den aktuellen Parameter. Innerhalb der Funktion wird mit einer *Kopie* des Zeigers auf den Aktualparameter gearbeitet, d.h. die *Startadresse* der Aktualparameters ändert sich im aufrufenden Modul nicht. Gleichwohl kann der *Wert* des Aktualparameters innerhalb der Funktion geändert werden - er bleibt auch nach Rücksprung in das aufrufende Programmmodul verändert. Das ist genau die Wirkung, die sich ergibt, wenn man in C++ *Referenzparameter* übergibt.

Das kleine Testprogramm *ZeigPar.cpp* ist im ANSI-C-Stil geschrieben und realisiert Variablenparameter durch Zeigerübergabe.

```
Variablen vor Funktionsaufruf:  x = 4, y = 8
Variablen nach Funktionsaufruf: x = 8, y = 4
```

Sie erhalten exakt das gleiche Programmverhalten, wenn Sie C++-Syntax verwenden (im Programm als Kommentar angegeben). Da das C++-Programm wesentlich lesbarer ist, wird bei Referenzparametern von Programmierung im C-Stil ausdrücklich *abgeraten*.

```
/* PROGRAMM ZeigPar.cpp
   Variablenparameter im C-Stil durch Zeigerübergabe */
#include <iostream>

// C-Variante ================  C++-Variante ================

void tausche(int* a, int* b) {  // void tausche(int &a, int &b) {
int temp;                       // int temp;

temp = *a;                      // temp = a;
*a = *b;                        // a = b;
*b = temp;                      // b = temp;
}                               // }

void main() {
  int x=4, y=8;

  cout << "Variablen vor Funktionsaufruf:  x = " << x
       << ", y = " << y << endl;
  tausche(&x,&y);               // tausche(x,y);
  cout << "Variablen nach Funktionsaufruf: x = " << x
       << ", y = " << y << endl;
}
```

Parameter der Funktion main()

Die Funktion main() kann auch mit *Parametern* deklariert werden:

```
int main(int argc, char* argv[], char* env[]);
```

Die Parameter der Funktion main() werden vom Betriebssystem des Rechners mit Inhalt gefüllt.

In diesem Fall enthält *argc* die Zahl der *Kommandozeilen-Parameter*, die beim Aufruf des C++-Programmes übergeben werden und in *argv*, einem Array von char-Zeigern, stehen die einzelnen Parameter als C-Strings. Die Liste der Argumente wird durch '\0' beendet, daher gilt *argv[argc] == NULL*. In *argv[0]* steht der String, der als *Programmaufruf* eingegeben wurde, deshalb ist *argc* immer mindestens 1.

Über *env* können die Umgebungsvariablen des Betriebssystems abgefragt werden. Sie erhalten ein Liste der Form *Variable=Wert*. Am Anfang der Liste wird der aktuell eingestellte Programmpfad ausgegeben.

Das Programm *MainPar.cpp* gibt eine Liste aller Kommandozeilen-Parameter und Umgebungsvariablen aus. Es muß mit *MainPar Parameter1 Parameter2* usw. von der Kommandozeile aus aufgerufen werden.

```
/* Programm MainPar.cpp
   Kommandozeilenparameter und Umgebungsvariablen abfragen */
#include <iostream>

int main(int argc, char* argv[], char *env[]) {
  cout << "Programmaufruf: "
       << argv[0] << endl;
  cout << "Das Programm wurde mit " << (argc-1)
         << " Parametern aufgerufen:" << endl;
  int i=1;
  while(argv[i]) cout << argv[i++] << endl;

  cout << endl << "Umgebungsvariablen:" << endl;
  i=0;
  while(env[i]) cout << env[i++] << endl;
  return 0;
}
```

L4 Zeiger auf Strukturelemente, Funktionen und Elementfunktionen

Werden Zeigervariablen auf Strukturen oder Klassen gebildet, greift man mit dem Operator „`->`“ anstelle von „`.`“auf die einzelne Elemente zu:

```
struct TSatz { // hier könnte auch class stehen
  int Nr;
  char Name[10];
  float Umsatz;
};
typedef TSatz *PSatz; // Typ ist Zeiger auf TSatz
TSatz Satz;           // Struktur vom Typ TSatz
PSatz ps = &Satz;     // Zeiger auf Struktur Satz

cout << "Beim " << ps->Nr << ". Satz stehen "
     << ps->Umsatz " DM Umsatz.";
```

Anstelle von „`x->y`“ kann man auch schreiben „`*x.y`“, es ist also `ps->Nr` identisch mit `*ps.Nr`. Die Schreibweise mit dem Pfeil-Operator ist aber wesentlich übersichtlicher und daher zu bevorzugen.

Mit Zeigern auf *Funktionen* kann man *zur Laufzeit* eine bestimmte Funktion auswählen, die ausgeführt werden soll. Dieses Verhalten bildet die Grundlage für die Eigenschaft der *Polymorphie* bei der objektorientierten Programmierung (vgl. späteres Kapitel). Zeiger auf Funktionen können (wie andere Variablen) anderen Funktionen als Parameter übergeben werden.

Im Beispielprogramm *FuncZgr.cpp* wird auf diese Weise wahlweise das Maximum oder das Minimum von zwei Zahlen bestimmt. Hierbei muß zur Deklaration des Zeigers *fp*, der auf die ausgewählte Funktion zeigt, die Syntax *int (*fp)(int, int)* verwendet werden. (Die Syntax *int* fp(int,int)* würde eine Funktion mit zwei *int*-Parametern deklarieren, die einen Zeiger auf *int* zurückgibt.)

```
/* Programm FuncZgr.cpp
   Auswahl einer auszuführenden Funktion mittels Zeiger */
#include<iostream>

int max(int x, int y)
{ if (x > y) return x; else return y; }

int min(int x, int y)
{ if (x < y) return x; else return y; }

void main() {
  int a = 1700, b = 1000;
  int (*fp)(int, int); // (*fp) ist Zeiger auf eine Funktion
  while(true) {
    char c;
    cout << "Maximum bzw. Minimum von 1000 und 1700 bestimmen.\n";
    cout << "Fuer Maximum eine 1, fuer Minimum eine 0 eingeben ";
    cout << "(sonst=Ende): ";
    cin >> c;
    // Zuweisung von max()min() ohne Klammern nach Funktionsnamen
    if (c == '1') fp = max; else if (c=='0') fp = min; else break;
    // Dereferenzierung des Funktionszeigers und Aufruf
    cout << (*fp)(a,b) << endl;
  }
```

```
}
```

```
Maximum bzw. Minimum von 1000 und 1700 bestimmen.
Fuer Maximum eine 1, fuer Minimum eine 0 eingeben (sonst=Ende): 1
1700
Maximum bzw. Minimum von 1000 und 1700 bestimmen.
Fuer Maximum eine 1, fuer Minimum eine 0 eingeben (sonst=Ende): 0
1000
```

Zeiger können ebenso auf *Elementfunktionen (Methoden)* oder *Elementdaten* eines Objekts gerichtet werden:

```
class Ort {
  int xKoordinate,
      yKoordinate;
public:
  Ort():xKoordinate(0),yKoordinate(0){};
  Ort(int x, int y):xKoordinate(x),yKoordinate(y){};
  int X() const { return xKoordinate; } // x ausgeben
  int Y() const { return yKoordinate; } // y ausgeben
}

void main() {
  Ort einOrt(100,200);
  int Ort::*fp();

  fp = Ort::X;                    // Zeiger fp auf Elementfunktion Ort
  cout << (einOrt->fp)() << endl; // dasselbe wie EinOrt.X()
  fp = Ort::Y;                    // Funktionszeiger umschalten
  cout << (einOrt->fp)() << endl; // jetzt dasselbe wie EinOrt.Y()
}
```

2.13 Dynamische Datentypen

Lerninhalte

❶ Einfache dynamische Typen
❷ Dynamisch erzeugte Arrays
❸ Dynamisch erzeugte Strukturen
❹ Freigeben dynamischer Objekte

Lerninhalte

Bei allen Datentypen, die bisher behandelt wurden, wurde der notwendige Speicherplatz bereits zur Compilierzeit fest vergeben. In vielen Fällen ergibt sich der tatsächliche Speicherbedarf erst während der Laufzeit eines Programmes. Die meisten höheren Programmiersprachen bieten daher die Möglichkeit, mitten im Programmablauf eine beliebige Menge von Speicher für eine Variable oder ein Objekt zu reservieren und den Speicher wieder freizugeben, wenn er nicht mehr gebraucht wird.

In C++ sind die hierzu notwendigen Schlüsselwörter:

- *new* zum Reservieren von Speicherplatz
- *delete* zum Freigeben des Speicherplatzes.

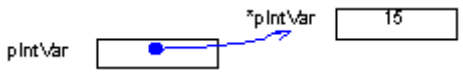
Dynamisch erzeugte Variablen unterliegen *nicht* den Regeln für Gültigkeitsbereiche, die bei statischen Variablen gelten. *new* erkennt die benötigte Menge Speicher am Datentyp, sie muß also nicht explizit angegeben werden. Alle dynamisch erzeugten Datentypen werden in einem besonderen, zusammenhängenden Adreßbereich namens *Heap* gespeichert (deutsch *Halde)*, auch *Freispeicher* genannt. Der Zugriff auf dynamische Variablen geschieht ausschließlich über Zeiger.

L1 Einfache dynamische Typen

„Einfache“ Datentypen wie *int, char* oder *float* werden wie folgt dynamisch erzeugt:

```
int* pIntVar;       // Zeiger auf IntVar
pIntVar = new int;  // dynamische Integer-Variable auf Heap erzeugen

*pIntVar = 15;      // Der dynamischen Variable einen Wert zuweisen
cout << *pIntVar << endl; // Ausgabe: 15
```

Zur Compilierzeit wird lediglich Speicherplatz für die Zeigervariable *p* reserviert. Mit *p= new int* wird Speicherplatz in der Größe von *sizeof(int)* Bytes erst *zur Laufzeit* des Programmes bereitgestellt. Nach *new* zeigt der Zeiger *p* auf den Anfang des reservierten Speicherplatzes. Der Ausdruck **pIntVar* kann als Name für die dynamisch erzeugte Variable interpretiert werden.

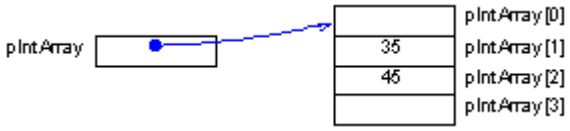


L2 Dynamisch erzeugte Arrays

Zum dynamischen Erzeugen von Arrays benutzt man die Syntax *new typname[Anzahl]*:

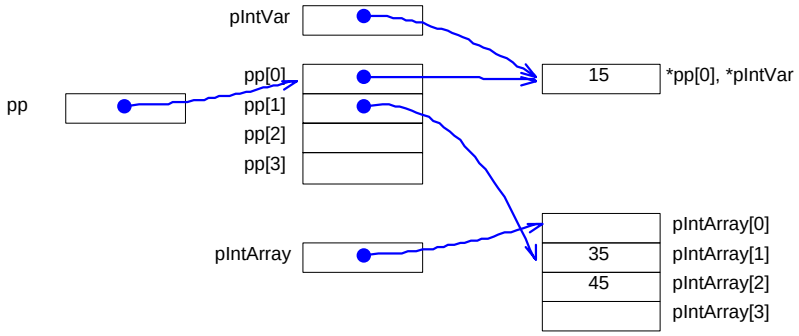
```
int* pIntArray = new int[4]; //Array mit 4 Integer-Zahlen
pIntArray[1] = 35;
pIntArray[2] = 45
cout << pIntArray[1] << endl; // Ausgabe: 35
```

Da der Bezeichner eines Arrays bereits einen Zeiger darstellt (siehe vorherige Kapitel), erfolgt der Zugriff auf ein dynamisch erzeugtes Array-Element *ohne* vorangestellten Stern * . (Im obigen Beispiel enthalten die Arrayelemente 0 und 3 undefinierte Werte.)




```
// Kompliziertere dynamische Datentypen
int** pp = new int* [4]; // Array von Zeigern auf int-Zahlen
pp[0] = pIntVar;         // pp[0] zeigt auf *pIntVar
pp[1] = &pIntArray [2]   // pp[1] zeigt auf pa[2]
cout << *pp[0] << endl;  // 15
cout << **pp   << endl;  // 15
cout << *pp[1] << endl;  // 45
```

Die Deklaration *int** pp* ist gleichbedeutend mit *(int*)* pp*. Das heißt, *pp* ist ein „Zeiger auf Zeiger auf *int*“, weil die Array-Elemente selbst Zeiger sind. Direkt nach der Deklaration sind diese Zeiger alle noch undefiniert, verweisen also nicht auf bestimmte Speicherplätze.



L3 Dynamisch erzeugte Strukturen

Datentypen wie *struct* oder *class* werden folgendermaßen dynamisch erzeugt:

```
struct Tstruct {
  int a;
  double b;
  Tstruct* next;
};

Tstruct* pStruktur = new Tstruct;
```

Der Zeiger *pStruktur* verweist nun auf ein neu erzeugtes Objekt von der Größe der Struktur *sizeof(Tstruct)*, was der Summe *sizeof(int)+sizeof(double)+sizeof(Tstruct*)* entspricht. Das letzte Strukturelement *next* ist selbst wieder ein Zeiger auf eine Struktur vom Typ *Tstruct*. Solche *next*-Elemente werden in dynamisch erzeugten Strukturen sehr oft verwendet - sie dienen zum Aufbau von dynamisch veränderbaren *Listen* (siehe späteres Kapitel).

Der Zugriff auf die Elemente einer dynamisch erzeugten Struktur geschieht über den *Pfeil-Operator* ->:

```
  sp->a = 12;
  sp->b = 17.4;
```

L4 Freigeben dynamischer Objekte

Der Operator *delete* gibt den reservierten Platz wieder frei, damit er von neuen dynamischen Variablen belegt oder anderen Programmen zur Verfügung gestellt werden kann. Nach dem Schlüsselwort *delete* wird der Zeiger angegeben, der auf das zu löschende Objekt verweist. Bei zu löschenden Arrays muß man *delete[]* statt *delete* verwenden.

```
// Alle erzeugten Objekte der Reihe nach löschen:
delete p;   // delete pp[0] wäre das Gleiche
delete[] pa;
delete[] *pp;
delete sp;
```

Wenn die Speicherbeschaffung mit *new* fehlschlägt, weil nicht mehr genügend neuer Speicher auf dem *Heap* vorhanden ist, bricht das Programm mit einem Laufzeitfehler ab, falls nicht besondere Vorkehrungen getroffen werden.

Einige Dinge müssen bei der Verwendung von *new* und *delete* beachtet werden. Ansonsten kann ein unvorhergesehenes Programmverhalten auftreten, meistens leider *ohne* Fehlermeldung des Compilers.

1. *delete* darf *nur* auf Objekte angewandt werden, die mit *new* erzeugt wurden.
2. *delete* darf auf ein Objekt nur einmal angewandt werden. Nach der Anwendung von *delete* ist der Wert des betreffenden Zeigers undefiniert (leider nicht gleich *NULL*).

   Falls mehrere Zeiger auf das gleiche Objekt zeigen, bewirkt schon ein *delete* auf nur einen dieser Zeiger eine Zerstörung der dynamischen Variablen. Die anderen Zeiger verweisen danach auf einen „verwaisten“ Speicherbereich. Sie heißen dann „hängende Zeiger“.
3. Wenn man *delete* auf einen NULL-Zeiger anwendet, hat das keine Wirkung und ist unschädlich.

4. Wird ein mit *new* erzeugtes Objekt mit dem Operator *delete* gelöscht, dann wird automatisch der *Destruktor* (siehe späteres Kapitel) für dieses Objekt aufgerufen.
5. Mit *new* erzeugte Variablen sind, unabhängig von irgendwelchen Blockgrenzen, *überall* gültig, bis sie mit *delete* gelöscht werden.

   Im Gegensatz dazu gelten für die (statisch deklarierten) *Zeiger* auf dynamisch erzeugte Variablen die normalen Gültigkeitsregeln. Daher muß man streng darauf achten, daß ein Zeiger auf eine dynamische Variable mindestens bis zu deren Löschung existiert.

   Ein Beispiel, wie es *nicht* programmiert werden sollte.

```
void falsch()
{
  int *a=new int;
  // ...
  // wenn bis hier kein "delete a" steht, ist der angeforderte Speicher
  // für *a verwaist, d.h es zeigt kein Zeiger mehr darauf.
  // Wiederholte Aufrufe von falsch() erzeugen ein "Speicherleck", da jeder
  // Aufruf Speicher anfordert aber nicht mehr zurückgibt, also verfügbarer
  // Speicher verlorengeht.
}
```

6. Die Freigabe von Arrays erfordert die Angabe von eckigen Klammern. Ohne Klammerangabe gäbe *delete* nur das erste Arrayelement frei. Der Rest des Arrays wäre dann nicht mehr zugänglich („verwitwetes Objekt“). Die Anweisung *delete[]* ist dann (und nur dann!) zu benutzen, wenn die zu löschende Variable mit *new[]* erzeugt wurde.

Ein großes Problem bei dynamisch erzeugten Variablen stellt die *Zerstückelung* des Speichers durch viele *new*- und *delete*-Operationen dar. Mit *Zerstückelung* ist gemeint, daß sich kleinere belegte und freie Speicherplätze abwechseln, so daß die plötzliche Anforderung eines größeren zusammenhängenden Bereichs für ein großes Datenobjekt nicht mehr erfüllbar ist, obwohl die Summe aller freien Plätze ausreichen würde.

Dieses Problem verlangt ein *Zusammenschieben* aller belegten Plätze (engl. *garbage collection* für *Müllsammlung*), so daß ein großer freier Bereich entsteht. Standardmäßig enthält C++ keine *garbage collection*, weil dies viel Rechenzeit kostet. Man kann jedoch eine eigene Speicherwaltung mit diesen Eigenschaften schreiben oder entsprechende Bibliotheken einbinden.

2.14 Listen und Bäume

Lerninhalte

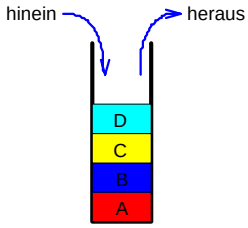
1. Stapel (LIFO)
2. Schlange (FIFO)
3. Einfach verkettete Liste
4. Aufbau von binären Bäumen
5. Bearbeiten von Daten in binären Bäumen
6. Suchen in binären Bäumen

Lerninhalte

Listen und Bäume sind dynamische Datenstrukturen im Gegensatz zu Arrays oder Strukturen, die zu den statischen Datenstrukturen zählen. Statische Datenstrukturen können zwar dynamisch *erzeugt* werden, danach aber nicht mehr dynamisch *verändert* werden. Dynamische Datenstrukturen sind vollständig dynamisch, d.h. sie können auch während der Laufzeit des Programmes ihren Speicherplatzbedarf dynamisch den Erfordernissen anpassen.

Die in den folgenden Abschnitten behandelten Listen und Bäume stellen nur eine kleine Auswahl der bekannten und verwendeten dynamischen Datenstrukturen dar. Insbesondere im Bereich der Baumstrukturen werden je nach Anwendungsfall ausgeglichene Bäume (AVL-Bäume), B-Bäume usw. verwendet. Ausgeglichene Bäume, auch AVL-Bäume nach den Entdeckern Adelson-Velskij und Landis genannt, stellen durch ausgefeilte Algorithmen sicher, dass der binäre Baum nicht zur Liste entartet und damit alle Vorteile verliert. B-Bäume enthalten in einer Verzweigung nicht nur ein Element, sondern ein sortiertes Array von Elementen, so dass jede Verzweigung nicht in zwei, sondern in mehr Äste verzweigt. Dadurch werden die Vorteile von Arrays und binären Bäumen kombiniert.

L1 Stapel (LIFO)



Stapel ist eine anschauliche Bezeichnung für eine Datenstruktur, die sich wie ein Stapel Papier verhält. Üblicherweise wird man *oben* auf den Stapel etwas auflegen, und dieses wird man auch als *erstes* wieder vom Stapel entfernen. (Wenn das nicht ausreicht, ist der Stapel eine ungeeignete Datenstruktur, so wie das bei den meisten Papierstapeln auf Schreibtischen der Fall ist.) Daher nennt man den Stapel auch einen **L**ast-**I**n-**F**irst-**O**ut-Speicher (LIFO). Eine andere Bezeichnung für den Stapel ist auch Keller, da einige Daten dadurch in der Tiefe eines engen Kellers gelagert werden und man erst an sie herankommt, wenn man die zuletzt eingelagerten Daten entfernt hat.

Ein Programm, das auf einer LIFO-Struktur basiert, könnte etwa folgende Ausgabe zeigen:

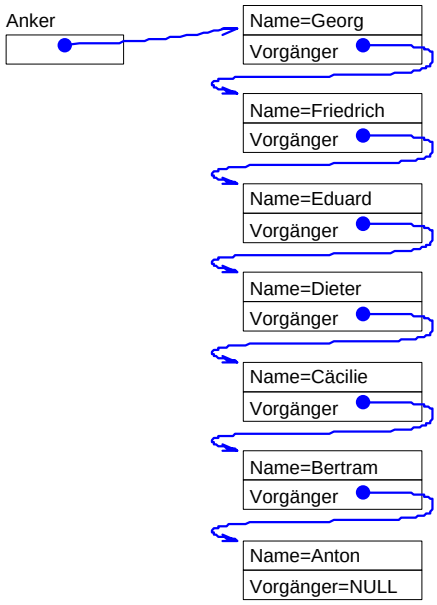
```
Bitte eine Liste von Namen eingeben (Strg-Z fuer Ende)
Bitte geben Sie einen Namen ein: Anton
Bitte geben Sie einen Namen ein: Bertram
Bitte geben Sie einen Namen ein: Cäcilie
Bitte geben Sie einen Namen ein: Dieter
Bitte geben Sie einen Namen ein: Eduard
Bitte geben Sie einen Namen ein: Friedrich
Bitte geben Sie einen Namen ein: Georg
Bitte geben Sie einen Namen ein:
Ende der Eingabe.
Gespeicherte Namen:
Georg
Friedrich
Eduard
Dieter
Cäcilie
Bertram
Anton
```

Die Namen werden in zur Eingabereihenfolge umgekehrter Folge ausgegeben.

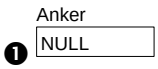
```
// lifo.cpp
// Ein- und Ausgabe einer Liste von Namen nach dem LIFO-Prinzip

#include <iostream>
#include <string>

// Datenelement als Klasse
class TElement
{
  public:
  // Eigenschaften
  string Name;
  TElement* Vorgaenger;
  // Konstruktoren
  TElement():Name(""),Vorgaenger(NULL){}
  TElement(const string &n):Name(n),Vorgaenger(NULL){}
  // Methoden
  bool Eingabe();
  void Ausgabe();
};
typedef TElement* PElement;

bool TElement::Eingabe()
{
  cout << "Bitte geben Sie einen Namen ein: ";
  cin >> Name;
  return Name.length()>0;
}

void TElement::Ausgabe()
{
  cout << Name << endl;
}

void main()
{
PElement Anker=NULL, Temp;
bool Weiter=true;

// Liste eingeben:
cout << "Bitte eine Liste von Namen eingeben (Strg-Z fuer Ende)" << endl;
do
{
  Temp=new TElement;
  if (Temp->Eingabe())
  { // in die Liste einfuegen
    Temp->Vorgaenger = Anker;
    Anker = Temp;
  }
  else
```

```
  { // Leeres Element loeschen und Schleife beenden
    delete Temp;
    Weiter=false;
  }
}
while(Weiter);

// Liste ausgeben:

cout << endl << endl << "Ende der Eingabe." << endl << "Gespeicherte Namen:" << endl;
while (Anker != NULL)
{
  Anker->Ausgabe();
  // Oberstes Element entfernen
  Temp=Anker;
  Anker = Anker->Vorgaenger;
  delete Temp;
}
}
```

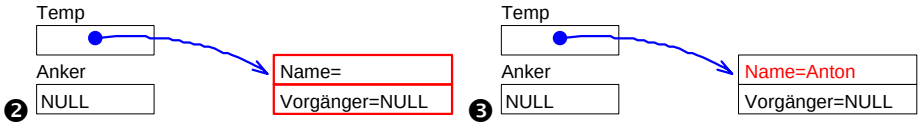
Erläuterung des Algorithmus' zum Füllen des Stapels:



Der Algorithmus hat nach allen Eingaben am Ende eine Struktur erzeugt, die aussieht wie neben stehend. Der Zeiger Anker zeigt dabei immer auf die Stapelspitze (Top Of Stack, TOS). Jedes Stapelelement beinhaltet einen Zeiger auf das darunterliegende Stapelelement. Lediglich das unterste Stapelelement enthält einen Nullzeiger als Kennung für das unterste und letzte Element des Stapels.



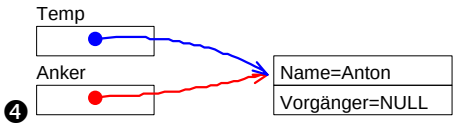
Die leere Liste besteht zunächst nur aus dem Zeiger auf das oberste Element, der als Kennzeichen dafür, dass er auf nichts zeigt, den Wert NULL enthält.



❷ Beim Hinzufügen eines neuen Elementes wird zunächst Platz für dieses mit einem Aufruf von new() reserviert. Der Zeiger Temp dient dazu, sich die Adresse der neuen Datenstruktur zu merken. Das neue Element wird von seinem Konstruktor initialisiert, so dass der Zeiger auf den Vorgänger den Wert NULL hat.

❸ Als nächstes wird ein Name eingegeben.

❹ Wenn dieser Name -wie hier der Fall- nicht leer ist, erhält der Vorgänger der Datenstruktur den Wert des Ankers, bleibt also im Moment auf NULL. Der Anker selbst erhält den Wert von Temp, zeigt also jetzt auf die aktuell bearbeitete Datenstruktur; diese wird damit zur Stapelspitze.

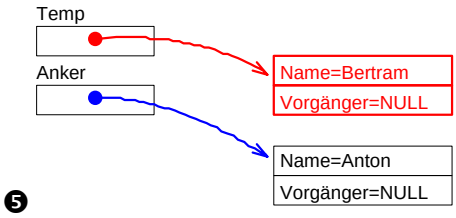


❹

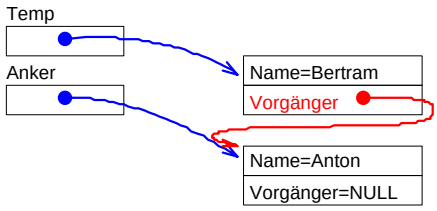
❺ Im zweiten Schleifendurchlauf wird wieder neuer Speicher angefordert und ein Name eingegeben. Temp zeigt auf das neu aufzunehmende Element.

❻ Nun erhält der Zeiger Vorgänger des neuen Elementes den Wert von Anker, er zeigt also ebenfalls auf den „alten“ Top Of Stack.

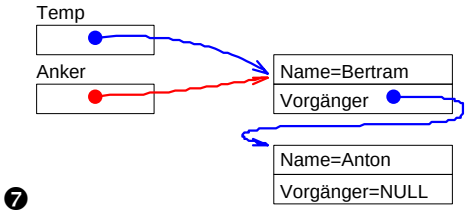
❼ Als letztes wird der Zeiger Temp nach Anker kopiert, womit das neue Element wirklich zum TOS wird. Anker zeigt jetzt auf den TOS, der Zeiger Vorgänger des TOS zeigt auf das unterste Element, dessen Zeiger Vorgänger ist ein NULL-Zeiger.



❺



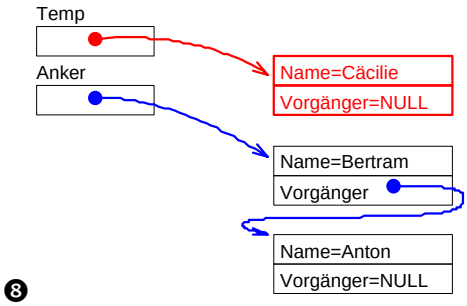
❻



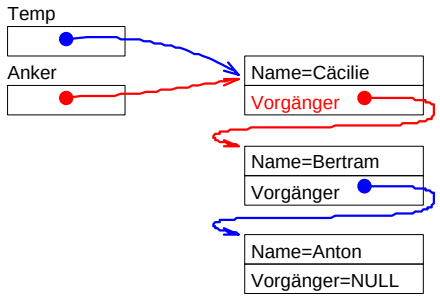
❼

❽ Auch im dritten Schleifendurchlauf wird Speicher angefordert und ein Name eingegeben.

❾ Der Zeiger Vorgänger des neuen Datensatzes zeigt dann auf den „alten“ TOS, der Anker zeigt auf den „neuen“ TOS.



❽



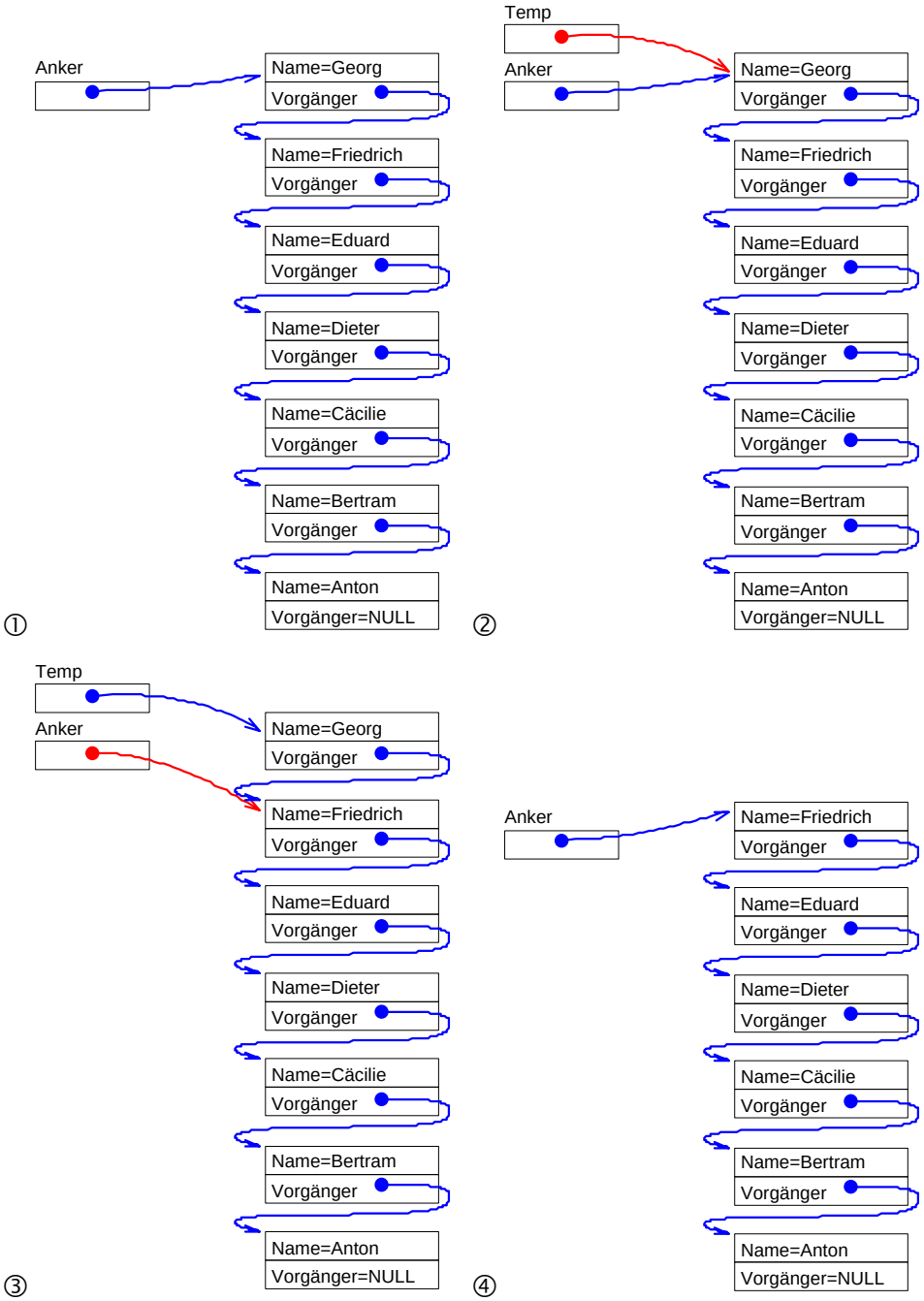
❾

Nach weiteren Schleifendurchläufen sieht der Stapel dann aus wie oben beschrieben.

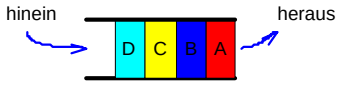
Erläuterung des Algorithmus' zum Leeren des Stapels:

Die Schleife läuft, solange der Anker nicht NULL ist, solange es also einen TOS gibt bzw. der Stapel nicht leer ist.

① Durch den Zeiger Anker auf den TOS hat das Programm Zugriff auf den TOS und seine Daten und kann diese, hier den Namen, ausgeben. ② Sobald dies geschehen ist, wird die Adresse des TOS zunächst im Zeiger Temp zwischengespeichert. ③ Daraufhin wird der Datensatz aus dem Stapel entfernt, indem der Anker auf den Vorgänger des „alten" TOS zeigt. ④ Der Speicher des Datensatzes kann daraufhin mittels delete() dem System zurückgegeben werden. Alternativ könnte der Datensatz natürlich auch weiter verwendet werden.



L2 Schlange (FIFO)



Eine Schlange als Datenspeicher stellt man sich als Röhre vor, in das auf der einen Seite die Daten hineinwandern, auf der anderen Seite kommen sie in dieser Reihenfolge wieder heraus.

Hier wird also das zuerst eingefügte Datenelement auch wieder zuerst entnommen, weswegen diese Struktur auch **F**irst-**I**n-**F**irst-**O**ut-Speicher (FIFO-Speicher) genannt wird.

Ein Programm, das auf einer FIFO-Struktur basiert, könnte etwa folgende Ausgabe zeigen:

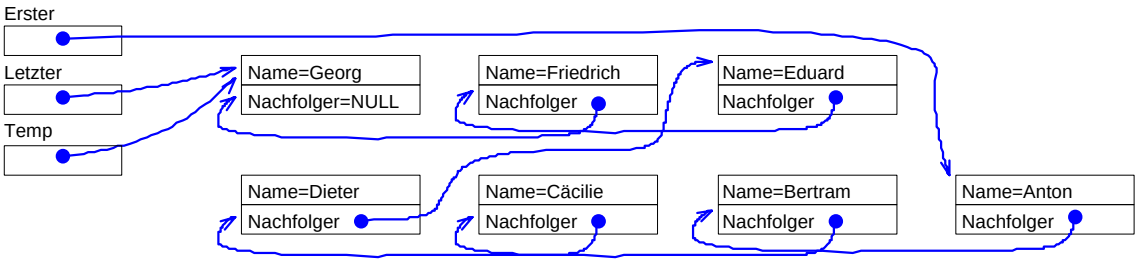
```
Bitte eine Liste von Namen eingeben (Strg-Z fuer Ende)
Bitte geben Sie einen Namen ein: Anton
Bitte geben Sie einen Namen ein: Bertram
Bitte geben Sie einen Namen ein: Cäcilie
Bitte geben Sie einen Namen ein: Dieter
Bitte geben Sie einen Namen ein: Eduard
Bitte geben Sie einen Namen ein: Friedrich
Bitte geben Sie einen Namen ein: Georg
Bitte geben Sie einen Namen ein:
Ende der Eingabe.
Gespeicherte Namen:
Anton
Bertram
Cäcilie
Dieter
Eduard
Friedrich
Georg
```

Die Namen werden in der Eingabereihenfolge ausgegeben.

```
// fifo.cpp
// Ein- und Ausgabe einer Liste von Namen nach dem FIFO-Prinzip
#include <iostream>
#include <string>

// Datenelement als Klasse
class TElement
{
  public:
  // Eigenschaften
  string Name;
  TElement* Nachfolger;
  // Konstruktoren
  TElement():Name(""),Nachfolger(NULL){}
  TElement(const string &n):Name(n),Nachfolger(NULL){}
  // Methoden
  bool Eingabe();
  void Ausgabe();
};
typedef TElement* PElement;

bool TElement::Eingabe()
{
  cout << "Bitte geben Sie einen Namen ein: ";
  cin >> Name;
  return Name.length()>0;
}

void TElement::Ausgabe()
{
  cout << Name << endl;
}

void main()
{
PElement Erster=NULL, Letzter=NULL, Temp;
bool Weiter=true;

// Liste eingeben:
cout << "Bitte eine Liste von Namen eingeben (Strg-Z fuer Ende)" << endl;
do
{
  Temp=new TElement;
  if (Temp->Eingabe())
  { // in die Liste einfuegen
    if (Letzter==NULL)             // wenn die Liste leer war ...
      Erster=Temp;                 // ... ist der Erste auch der Letzte
    else                           // sonst ...
      Letzter->Nachfolger = Temp;  // ... ist der neue Datensatz der Letzte
    Letzter = Temp;
  }
  else
  { // Leeres Element loeschen und Schleife beenden
    delete Temp;
    Weiter=false;
  }
}
while(Weiter);

// Liste ausgeben:

cout << endl << endl << "Ende der Eingabe." << endl << "Gespeicherte Namen:" << endl;
while (Erster != NULL)
{
  Erster->Ausgabe();
  // Erstes Element entfernen
  Temp=Erster;
  Erster = Erster->Nachfolger;
  if (Erster==NULL) // Liste ist leergeraeumt...
    Letzter=NULL;
  delete Temp;
}
}
```

Erläuterung des Algorithmus' zum Füllen der Schlange:

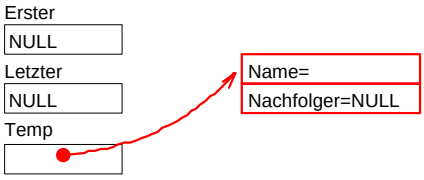
Der Algorithmus hat nach allen Eingaben am Ende eine Struktur erzeugt, die aussieht wie unten stehend. Der Zeiger Erster zeigt dabei immer auf den Ausgang der Schlange, der Zeiger Letzter auf den Eingang. Jedes Schlangenelement beinhaltet einen Zeiger auf das nachfolgende Schlangenelement (im Bild jeweils das linke). Lediglich das letzte Schlangenelement (im Bild ganz links oben) enthält einen Nullzeiger als Kennung für das Ende der Schlange.



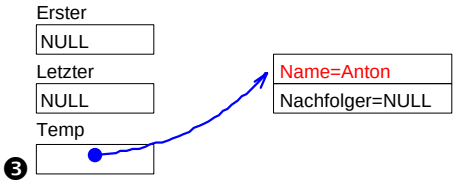
❶

Die leere Liste besteht zunächst nur aus je einem Zeiger auf das erste und das letzte Element, die als Kennzeichen dafür, dass sie auf nichts zeigen, den Wert NULL enthalten. Im Gegensatz zur FIFO-Struktur werden zwei Zeiger benötigt, da an einer Stelle der Liste hinzugefügt und an einer anderen Stelle weggenommen wird.

❷



❸

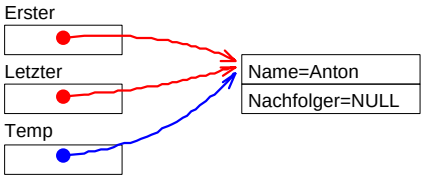


❷ Beim Hinzufügen eines neuen Elementes wird zunächst Platz für dieses mit einem Aufruf von new() reserviert. Der Zeiger Temp dient dazu, sich die Adresse der neuen Datenstruktur zu merken. Das neue Element wird von seinem Konstruktor initialisiert, so dass der Zeiger auf den Nachfolger den Wert NULL hat.

❸ Als nächstes wird ein Name eingegeben.

❹ Wenn dieser Name -wie hier der Fall- nicht leer ist, wird der Zeiger Temp nach Erster kopiert, das aktuelle Element wird also das erste in der Schlange, da die Schlange leer war. Außerdem ist jetzt das erste Element auch das letzte und deswegen wird Temp auch nach Letzter kopiert. Damit zeigen sowohl Erster als auch Letzter auf den neuen Datensatz.

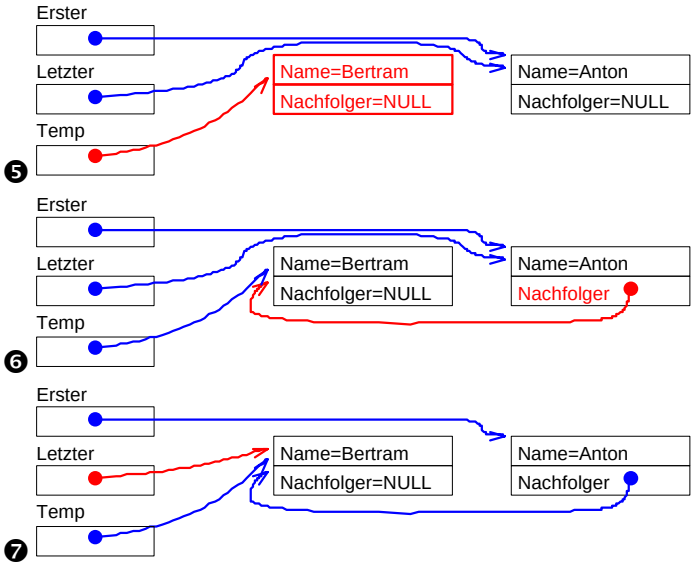
❹



❺ Im zweiten Schleifendurchlauf wird wieder neuer Speicher angefordert und ein Name eingegeben. Temp zeigt auf das neu aufzunehmende Element.

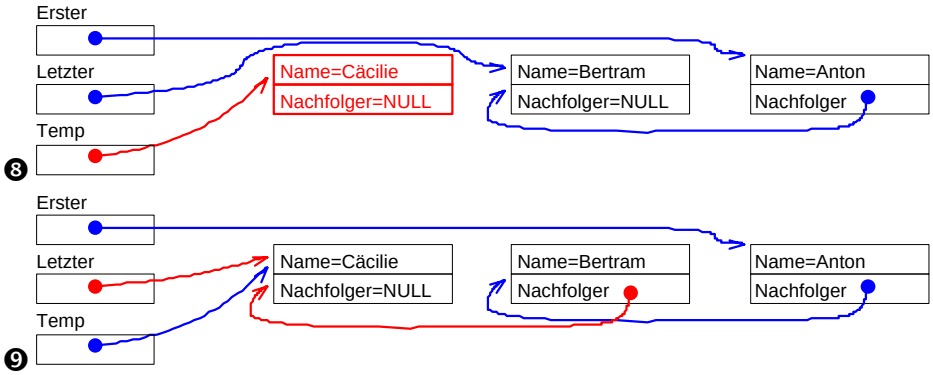
❻ Da die Schlange nicht leer war erhält der Zeiger Nachfolger des letzten Elementes den Wert von Temp, da dieses Element ja nachher eingegeben wurde.

❼ Als letztes wird der Zeiger Temp nach Letzter kopiert, womit das neue Element wirklich zum letzten wird. Erster zeigt jetzt auf den zuerst eingegebenen Datensatz, dessen Nachfolger auf den als zweites eingegebenen Datensatz, dieser weiderum hat keinen Nachfolger und trägt deshalb den Wert NULL. Der Zeiger Letzter zeigt auf den als zweites eingegebenen Datensatz.

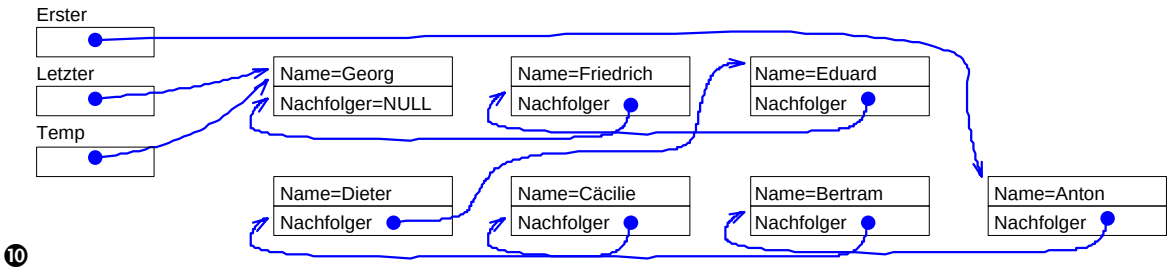


❽ Auch im dritten Schleifendurchlauf wird Speicher angefordert und ein Name eingegeben.

❾ Der Zeiger Nachfolger des „alten“ letzten Datensatzes zeigt dann auf den neuen Datensatz, der Zeiger Letzter zeigt auf den neuen Datensatz, der jetzt der letzte ist.



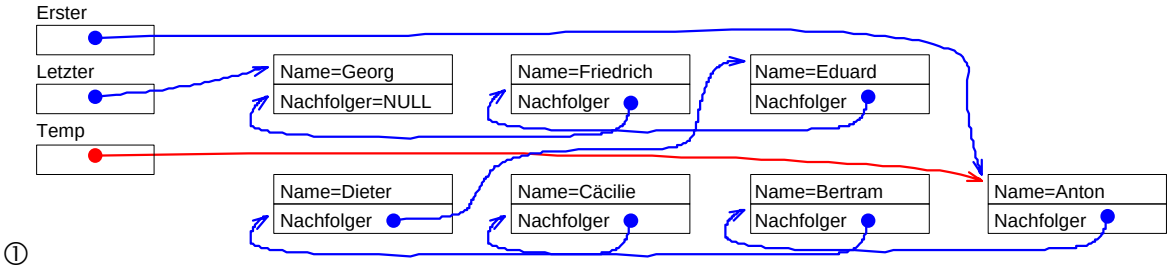
❿ Nach weiteren Eingaben entsteht folgende Strukur.



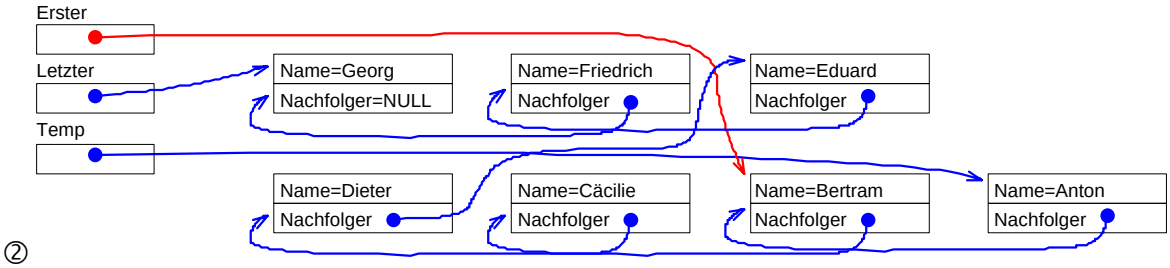
Erläuterung des Algorithmus' zum Leeren der Schlange:

Die Schleife läuft, solange der Zeiger Erster nicht NULL ist, solange es also einen ersten Datensatz gibt bzw. die Schlange nicht leer ist.

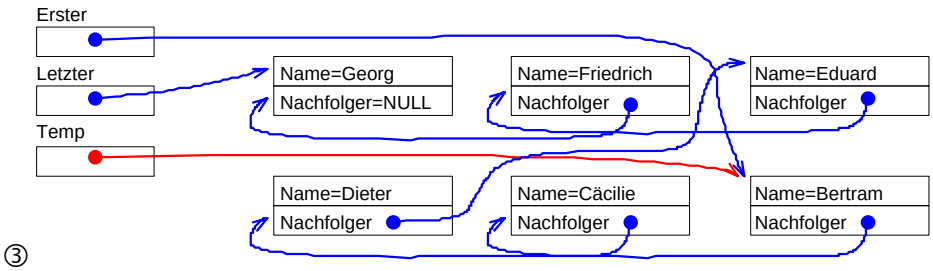
① Durch den Zeiger Erster auf den Ausgang der Schlange hat das Programm Zugriff auf den zuerst eingegebenen Datensatz und dessen Daten und kann diese, hier den Namen, ausgeben. Sobald dies geschehen ist, wird die Adresse des TOS zunächst im Zeiger Temp zwischengespeichert. ② Daraufhin wird der Datensatz aus dem Stapel entfernt, indem der Erste auf den Nachfolger des „alten" ersten Datensatzes zeigt. ③ Der Speicher des Datensatzes kann daraufhin mittels delete() dem System zurückgegeben werden. Alternativ könnte der Datensatz natürlich auch weiter verwendet werden.



①



②



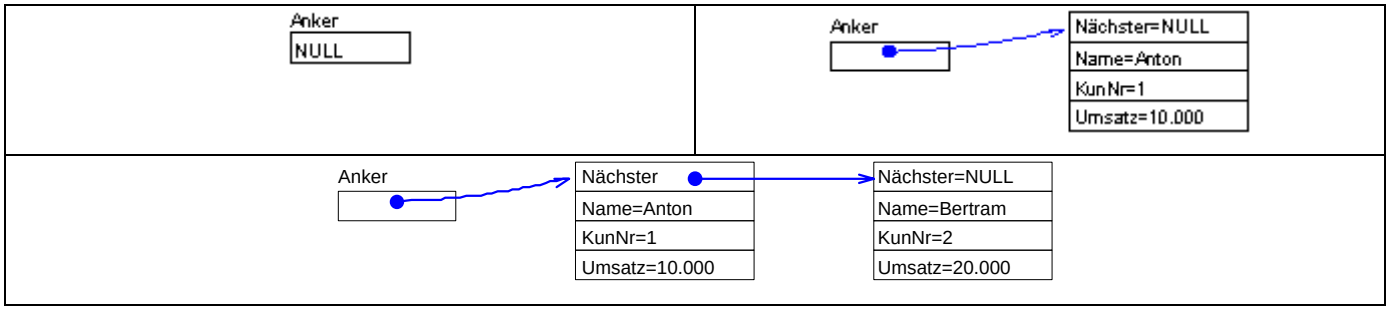
③

L3 Einfach verkettete Liste

Die einfach verkettete Liste wurde bereits verwendet, um LIFO- bzw. FIFO-Speicher zu realisieren. Mit einer verketteten Liste sind aber mehr Operationen möglich, als die engen Regeln dieser beiden Speicher-Strukturen erlauben. Diese Operationen werden im folgenden Beispiel besprochen. Dazu benötigen wir eine Datenstruktur, die im Beispiel als Nutzdaten einen Namen, eine Kundennummer und einen Umsatz trägt. Wichtig für die einfach verkettete Liste ist die Referenz auf eine andere Datenstruktur in der Liste, hier das Element „Naechster“, welches einen Zeiger auf eine Datenstruktur vom Typ TKunde enthält. Die Datenstruktur ist der Einfachheit halber als Klasse deklariert.

```
// ============================================================================
// Testklasse fuer die Liste
class TKunde
{
public:
  TKunde* Naechster;
  string  Name;
  long    KunNr;
  double  Umsatz;
  TKunde(string& n, long nr, double u):Name(n),KunNr(nr),Umsatz(u),Naechster(NULL){};
  void Eingabe();
  void Ausgabe();
};
typedef TKunde* PKunde;

void TKunde::Eingabe()
{
cout << "Kundennummer: "; cin >> KunNr;
cout << "Name: ";         cin >> Name;
cout << "Umsatz:";        cin >> Umsatz;
}

void TKunde::Ausgabe()
{
cout << KunNr << ", Name: " << Name << ", Umsatz:" << Umsatz << endl;
}
// ============================================================================
```

Über das Element „Naechster“ wird die Verkettung realisiert, von der die Liste ihren Namen hat. Jedes Element der Liste trägt einen Zeiger, der auf die Speicherstelle des nächsten Elementes in der Liste zeigt. Das letzte Element trägt als „Endekennung“ einen NULL-Zeiger. Um den Listenanfang finden zu können, muss es noch einen Zeiger auf das erste Element geben, einen sogenannten Listen-Anker. Die folgenden Bilder zeigen verschiede Listen, eine leere, eine mit einem und eine mit zwei Elementen.

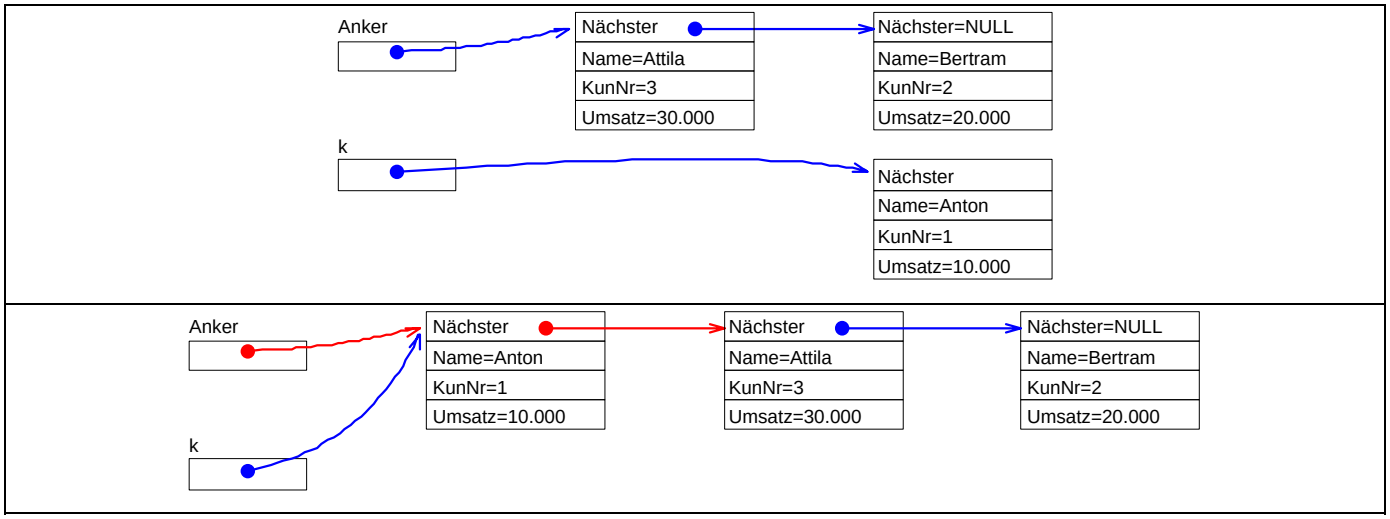


Um solche Listen zu verwalten, empfiehlt es sich, Funktionen zu verwenden. Was genau heißt aber „verwalten“? Es müssen Elemente eingefügt werden können, und zwar am Anfang, am Ende und an einer beliebigen Stelle. Einzelne Elemente oder die ganze Liste müssen gelöscht werden, die Liste muss

ausgegeben werden können. Für diese Verwaltungstätigkeiten werden jetzt eine Reihe von Funktionen vorgestellt.

Einfügen eines Elementes am Anfang ist der einfachste Fall des Einfügens. Dies bedeutet, dass ein neuer Listenanker eingefügt wird, der alte Listenanker rutscht dadurch an die zweite Stelle. Daher muss also als Nachfolger des neuen Elementes k der alte Listenanker eingetragen werden, neuer Listenanker wird das neue Element k.

```
TKunde* ListeEinfuegenAnfang(TKunde* &Anker,TKunde* k)
{ // Erstes Element wird Zweites
// Nachfolger von k ist das "alte" erste Element
k->Naechster=Anker;
Anker=k; // k wird "neues" erstes Element der Liste
return k;
}
```

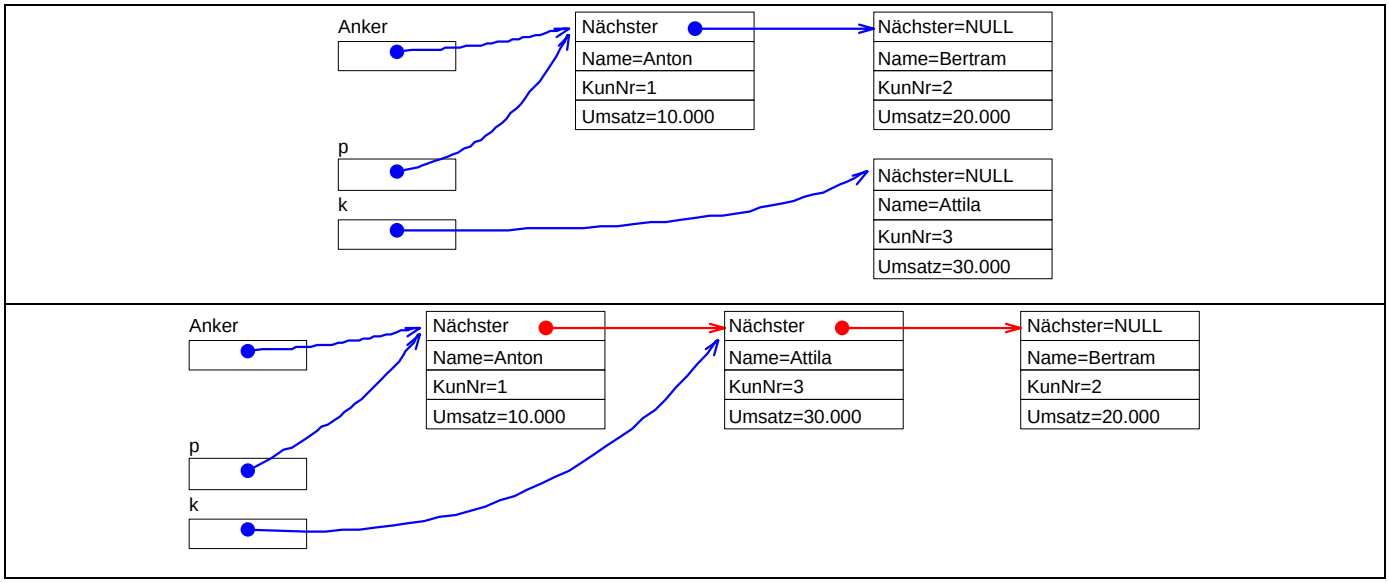


Einfügen eines Elementes hinter einer bestimmten Stelle

Wenn die Liste nach bestimmten Kriterien aufgebaut sein soll, z.B. nach Namen sortiert o.ä., dann wird es wichtig, neue Elemente an bestimmten Stellen einzufügen. Diese Stelle wird z.B. durch einen Zeiger p gekennzeichnet, der auf ein bestimmtes Element zeigt. Wie man dabei auf den Wert des Zeigers p kommt, soll hier nicht Thema sein, dies gehört in den Bereich der Suchverfahren.

Beim Einfügen hinter einem bestimmten Element p wird das neue Element k der Nachfolger von p, Nachfolger von k wird der frühere Nachfolger von p.

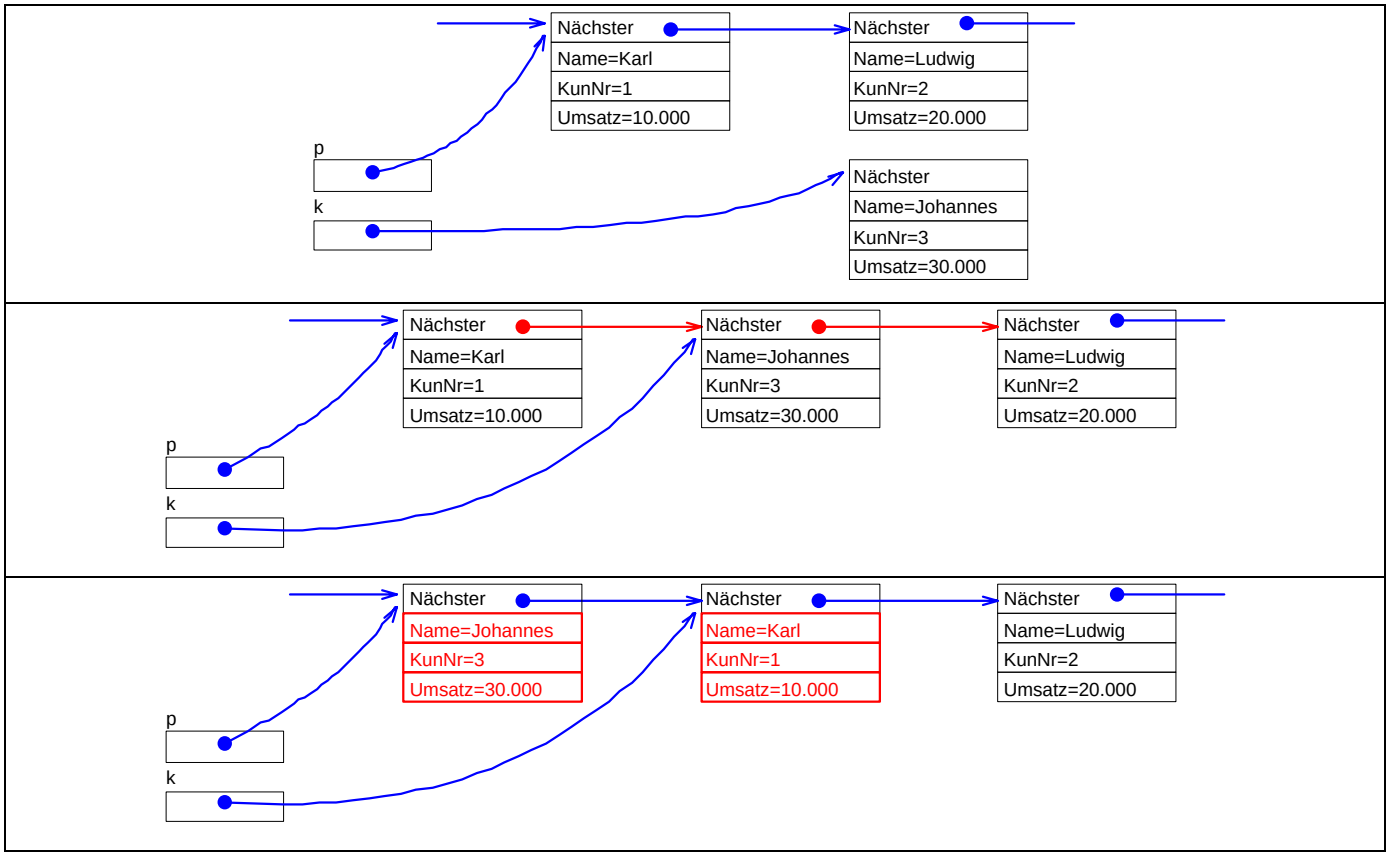
```
TKunde* ListeEinfuegenHinter(TKunde* p,TKunde* k)
{
// Nachfolger von p wird Nachfolger von k
k->Naechster=p->Naechster;
p->Naechster=k; // k wird Nachfolger von p
return k;
}
```



Einfügen eines Elementes vor einer bestimmten Stelle

Manchmal sollen neue Elemente aber auch nicht hinter, sondern vor einer bestimmten Stelle eingefügt werden. Damit ist ein Problem verbunden, denn es besteht bei der einfach verketteten Liste nicht direkt die Möglichkeit festzustellen, welches der Vorgänger eines Elementes ist. Ein Trick hilft hier aber weiter: Zunächst fügt man das neue Element hinter dem durch p markierten ein und vertauscht dann den Inhalt der beiden Elemente.

```
TKunde* ListeEinfuegenVor(TKunde* p,TKunde* k)
{ // Trick: erst Einfuegen dahinter, dann vertauschen
ListeEinfuegenHinter(p,k);
// Vertauschen der beiden Elemente
TKunde temp=*p;
p->Name=k->Name;  p->KunNr=k->KunNr;  p->Umsatz=k->Umsatz;
k->Name=temp.Name;k->KunNr=temp.KunNr;k->Umsatz=temp.Umsatz;
return p;
}
```



Einfügen eines Elementes am Ende der Liste

Beim Anfügen am Ende der Liste muß zunächst das letzte Listenelement gefunden werden. Hinter diesem wird dann das neue Element eingefügt. Dabei ist der Sonderfall zu beachten, dass die Liste leer ist, weil dann das neue Listenelement zum Anker wird.

Zum Suchen des letzten Listenelementes muss mit Hilfe eines Zeigers jedes Listenelement vom Anker ausgehend durchgegangen werden, bis eines gefunden wird, das keinen Nachfolger mehr hat. Dieses ist das letzte, dessen Nachfolger dann das neue Element wird.

```
TKunde* ListeEinfuegenEnde(TKunde* &Anker,TKunde* k)
{
TKunde* p=Anker;
k->Naechster=NULL; // k hat keinen Nachfolger
if (p==NULL) // wenn die Liste leer ist
{ // neues Element wird erstes Element
  Anker=k;
}
else // wenn die Liste nicht leer ist
{ // Zunächst letztes Element suchen
  while (p->Naechster!=NULL) p=p->Naechster;
  p->Naechster=k; // k wird letztes Element
}
return k;
}
```

Ausgabe aller Listenelemente

Auch zum Ausgeben aller Listenelemente muss Listenelement für Listenelement durchgegangen werden bis zum Ende und dabei die Ausgabemethode des Elementes aufgerufen werden.

```
void ListeAusgabe(TKunde* Anker)
{
TKunde* p=Anker;
while (p!=NULL) // Alle Elemente der Liste durchgehen
{
  p->Ausgabe();
  p=p->Naechster; // folgendes Element bearbeiten
}
}
```

Leeren der Liste

Beim Leeren der Liste darf man nicht zuerst ein Listenelement löschen und dann den Nachfolger betrachten, da dieser dann bereits ungültig ist. Vielmehr muß man sich zunächst das zu löschende Element merken, dann den Nachfolger aufsuchen und jetzt erst das Element löschen.

```
void ListeLeeren(TKunde* Anker)
{
TKunde* p=Anker;
TKunde* p2;
while (p!=NULL)
{
  p2=p;
  p=p->Naechster;
  delete p2;
}
}
```

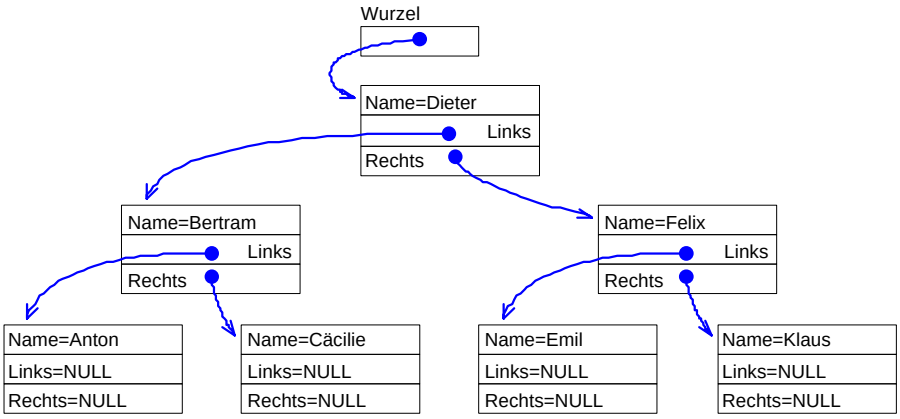
Das folgende Hauptprogramm wendet die oben definierten Funktionen auf eine Liste an. Dabei wird jeweils der Anker der Liste als Parameter übergeben, wodurch es möglich wäre, mehrere verschiedene Listen im Programm zu verwalten, jeweils gekennzeichnet durch ihren Anker.

```
void main()
{
TKunde* Anker=NULL;
TKunde* p;

p= // Zeiger auf Anton merken
ListeEinfuegenAnfang(Anker,new TKunde("Anton",  1,10000)); // Element am Anfang
  einfuegen
ListeEinfuegenAnfang(Anker,new TKunde("Bertram",2,20000)); // Element am Anfang
  einfuegen
ListeEinfuegenAnfang(Anker,new TKunde("Cäcilie",3,30000)); // Element am Anfang
  einfuegen
ListeEinfuegenEnde  (Anker,new TKunde("Damaris",4,40000)); // Element am Ende
  einfuegen
ListeEinfuegenEnde  (Anker,new TKunde("Emil",   5,50000)); // Element am Ende
  einfuegen
ListeEinfuegenHinter(p,    new TKunde("Frank",6,60000));   // Element hinter Anton
  einfuegen
ListeEinfuegenVor   (p,    new TKunde("Georg",7,70000));   // Element vor Anton
  einfuegen
ListeAusgabe(Anker);
ListeLeeren(Anker);
}
```

L4 Aufbau von binären Bäumen

Bäume sind Datenstrukturen, in denen Daten grundsätzlich *geordnet* abgespeichert werden. Das Ordnungskriterium in den Beispielen ist aufsteigende alphanumerische Ordnung nach Name, es sind aber prinzipiell beliebige Ordnungskriterien denkbar.

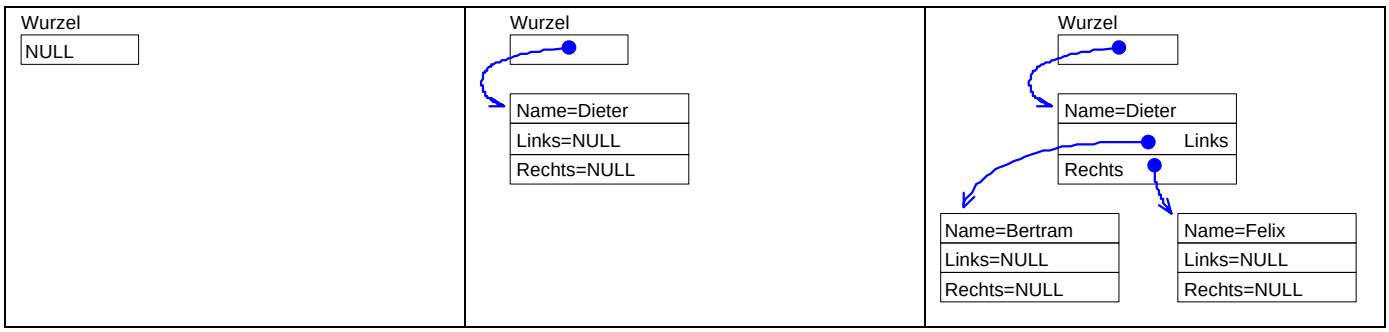
Die einfachste Baumstruktur ist der binäre Baum. Bei ihm kann jedes Datenelement maximal zwei verknüpfte Datenelemente haben. Eines der beiden verknüpften Elemente ist laut Ordnungskriterium kleiner, das andere größer.



Im Beispiel ist jeweils das linke Element und auch alle anderen Elemente des gesamten linke Teilbaums kleiner als das gerade betrachtete Datenelement, rechts ist alles größer.

Ein Datensatz besteht aus den Nutzdaten (hier der Name) und einem Zeiger auf den linken Teilbaum und einem Zeiger auf den rechten Teilbaum. Am Ende des Baumes (an den Blättern) findet sich ein NULL-Zeiger.

Die Bäume der Informatik wachsen von oben nach unten, daher befindet sich die Wurzel des Baumes oben am Baum. Sie besteht aus einem Zeiger auf ein Element. Ist der Baum leer, enthält sie einen Nullzeiger (siehe erstes Bild unten).

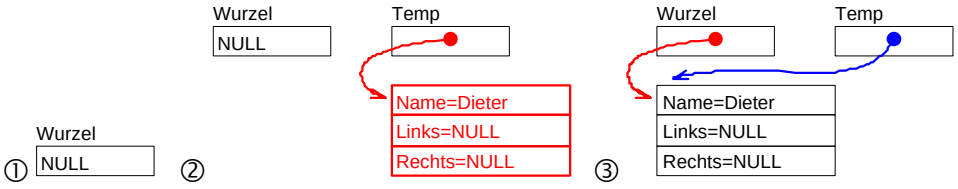


Die beiden anderen Bilder zeigen einen Baum mit einem Element und einen Baum mit drei Elementen.

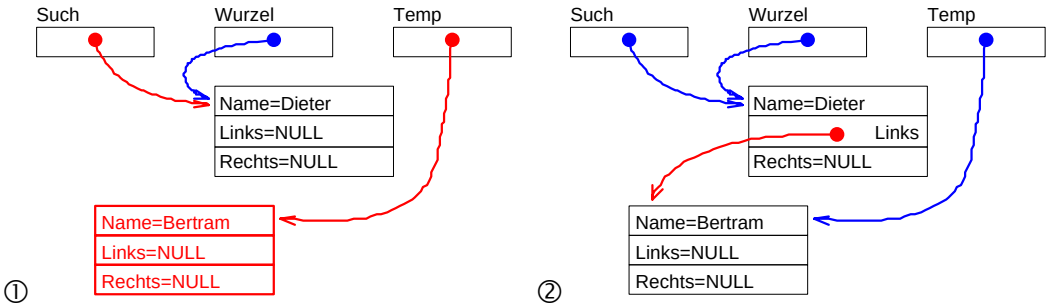
Da die Baumstruktur eine sortierte Struktur ist, bietet sie Vorteile beim Suchen. Binäres Suchen ist hier leicht möglich, da die Struktur des Baumes die Entscheidung „größer oder kleiner“ fördert.

Betrachten wir anhand einiger Beispiele, wie Daten in den Baum eingefügt werden.

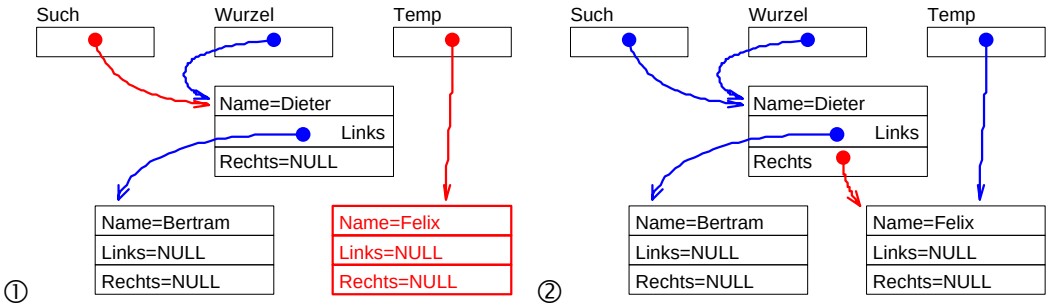
Erstes Element einfügen: ① Ausgehend von der leeren Liste wird ein neues Element eingefügt. ② Das neu einzufügende Element wird durch den zeiger Temp referenziert. Es trägt im Nutzteil den Namen „Dieter“, es hat keinen rechten und keinen linken Nachfolger. ③ Da es das erste Element im Baum ist, zeigt die Wurzel zunächst darauf.



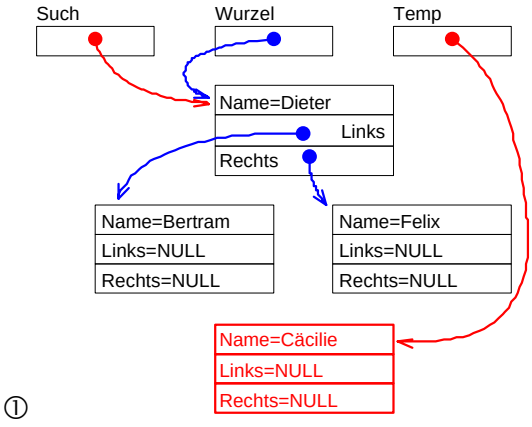
Zweites Element einfügen: ① Das zweite Element trägt in den Nutzdaten den Namen „Bertram“. Der Zeiger „Such“ referenziert zunächst die Wurzel des Baumes. Mit seiner Hilfe wird entschieden, wo neue Elemente einzufügen sind. Da dieser Baum nur aus einem Element besteht, ist die Suche nach dem Einfügepunkt in diesem Beispiel bereits beendet. Weil `Temp->Name` kleiner ist als `Such->Name`, ist das neue Element links von „Dieter“ einzufügen. Da links von „Dieter“ kein weiteres Element mehr existiert (`Such->Links==NULL`), ist es direkt links vom bereits existierenden „Dieter“ einzuordnen. ② Also zeigt „Dieters“ Zeiger auf den linken Nachfolger auf „Bertram“.



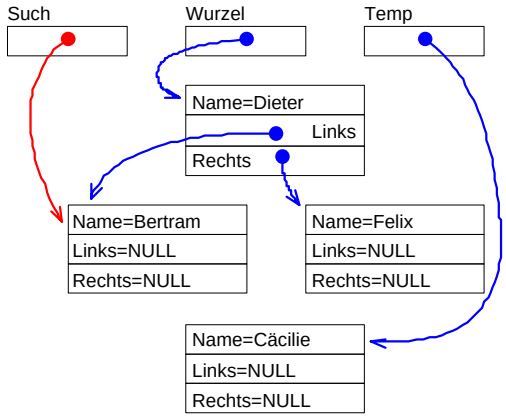
Drittes Element einfügen: ① Als drittes Element wird nun „Felix“ aufgenommen. Der Zeiger „Such“ referenziert zunächst wieder die Wurzel des Baumes. Weil `Temp->Name` größer ist als `Such->Name`, ist das neue Element rechts von „Dieter“ einzufügen. Da rechts von „Dieter“ kein weiteres Element mehr existiert (`Such->Rechts==NULL`), ist es direkt rechts vom bereits existierenden „Dieter“ einzuordnen. ② Der rechte Nachfolger von „Dieter“ wird „Felix“.



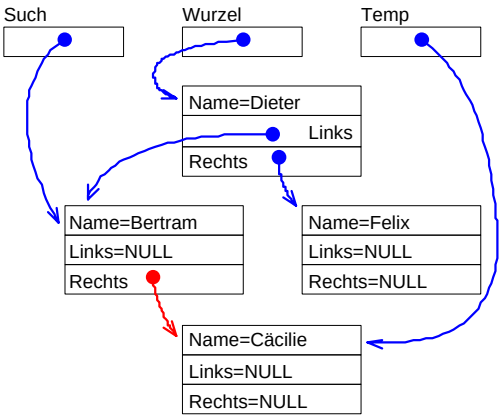
Viertes Element einfügen: ① Als viertes Element wird nun „Cäcilie“ aufgenommen. Der Zeiger „Such“ referenziert zunächst die Wurzel des Baumes. Weil `Temp->Name` (Cäcilie) kleiner ist als `Such->Name` (Dieter), ist das neue Element links von „Dieter“ einzufügen. ② Da links von „Dieter“ ein weiteres Element existiert (`Such->Links!=NULL`), wird der Zeiger Such auf dieses Element (Bertram) umgesetzt. ③ Weil `Temp->Name` (Cäcilie) größer ist als `Such->Name` (Bertram), ist das neue Element rechts von „Bertram“ einzufügen. Da für „Bertram“ rechts kein weiteres Nachfolge-Element mehr existiert (`Such->Rechts==NULL`), ist „Cäcilie“ direkt rechts vom bereits existierenden „Bertram“ einzuordnen.



①

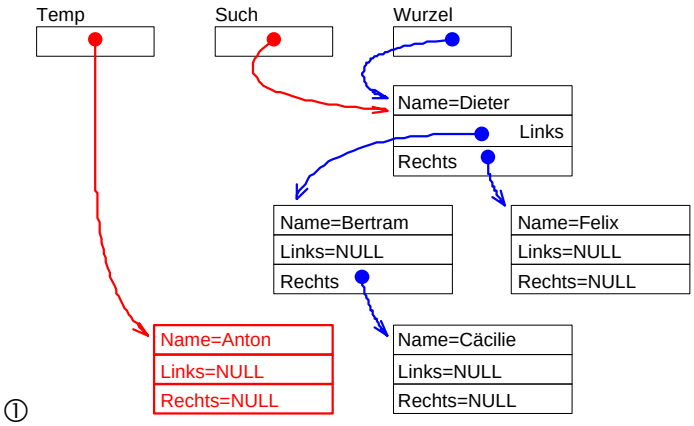


②



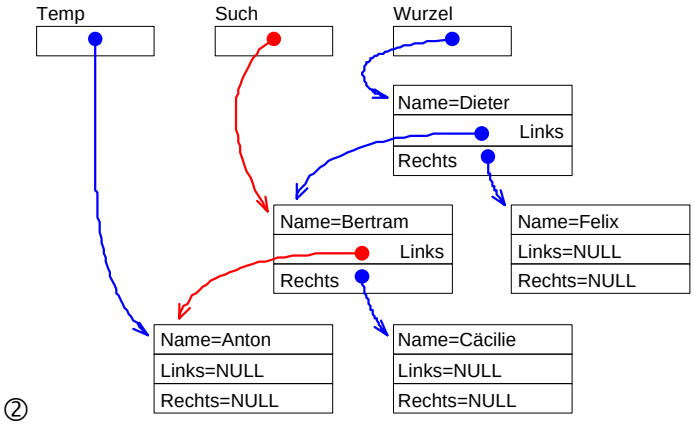
③

Fünftes Element einfügen:



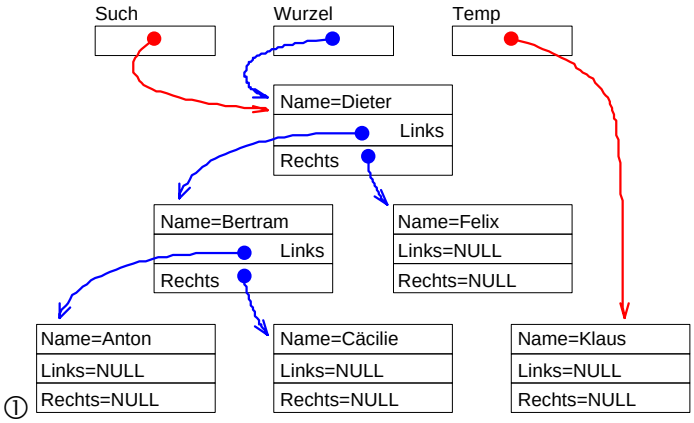
①

① „Anton“ soll eingefügt werden, Such wird auf die Wurzel gesetzt. Weil `Temp->Name` kleiner ist als `Such->Name`, ist das neue Element links einzufügen. Da links von „Dieter“ ein weiteres Element existiert (`Such->Links!=NULL`), wird der Zeiger Such auf dieses Element (Bertram) umgesetzt.

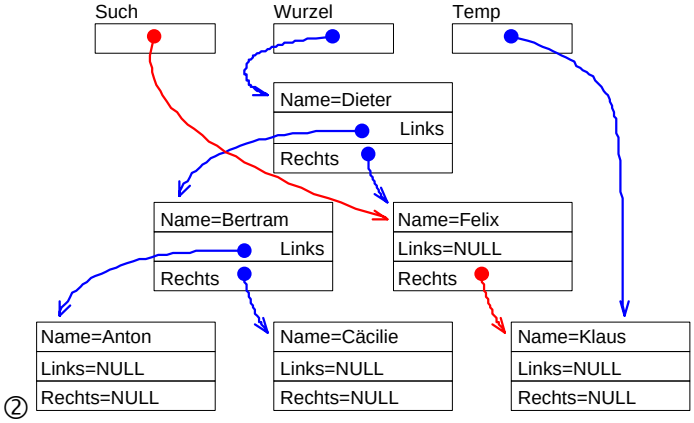


② Weil `Temp->Name` (Anton) kleiner ist als `Such->Name` (Bertram), ist das neue Element links einzufügen. Da für „Bertram“ links kein weiteres Nachfolge-Element mehr existiert (`Such->Links==NULL`), ist „Anton“ direkt links vom bereits existierenden „Bertram“ einzuordnen.

Sechstes Element einfügen:

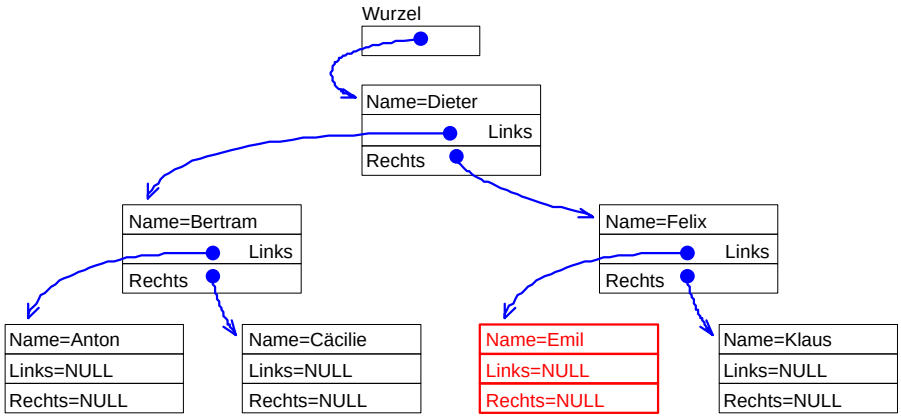


① „Klaus“ soll eingefügt werden. Zunächst wird rechts von „Dieter“ weitergesucht, da „Klaus“ größer ist als „Dieter“.



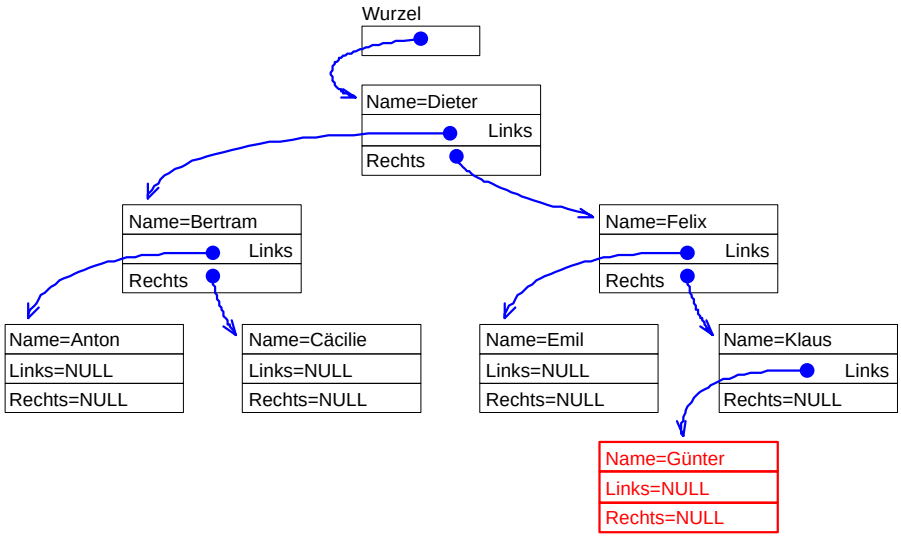
② Rechts von „Felix“ müßte weitergesucht werden. Da dort kein weiteres Element vorhanden ist, ist dies der Platz, an dem „Klaus“ eingefügt werden muß.

Siebtes Element einfügen:



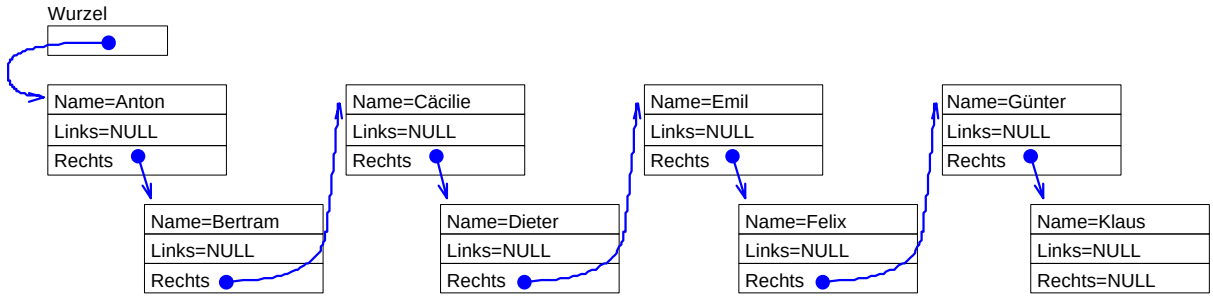
„Emil“ als siebtes Element wird rechts unterhalb von „Dieter“ und links unterhalb von „Felix“ einsortiert.

Achtes:



„Günter“ wird rechts unterhalb von „Dieter“, rechts unterhalb von „Felix“, links unterhalb von „Klaus“ einsortiert.

Das vorhergehende Beispiel wurde gewählt, um zu zeigen, wie ein binärer Baum idealerweise aussehen soll. Dabei war die Reihenfolge wichtig, in der die Elemente einsortiert wurden. Binäre Bäume mit gleichem Inhalt sehen nicht unbedingt gleich aus. Dies sieht man am Beispiel des folgenden „entarteten Baums“, der beginnend mit „Anton“ in alphabetischer Reihenfolge gefüllt wurde.



Dieser entartete Baum hat nicht mehr die positiven Eigenschaften des binären Baumes, er ist vielmehr eine verkettete Liste und hat deren Eigenschaften. Damit dauert Suchen in diesem speziellen Baum länger als in einem idealen binären Baum.

Damit solche Probleme nicht auftreten, wurden andere Arten von Bäumen entwickelt: ausgeglichene Bäume, AVL-Bäume, B-Bäume usw. Diese sorgen durch geeignete Algorithmen dafür, dass der Baum nicht entartet.

Ein Programmstück in C++, das den vorgestellten Algorithmus abarbeitet, sei im folgenden noch vorgestellt.

```
void insert(TElement* &wurzel, TElement* Temp)
{
if (wurzel==NULL) // Leerer Baum
{
  wurzel=Temp;
}
else
{
  TElement *Such=wurzel;
  while (Such!=NULL)
  {
    if (Temp->Name<Such->Name)
    {
      if (Such->Links==NULL) // kein linker Teilbaum
      {
        Such->Links=Temp; // links anfügen
        Such=NULL;
      }
      else
      {
        Such=Such->Links;
      }
    }
    else if (Temp->Name>Such->Name)
    {
      if (Such->Rechts==NULL) // kein rechter Teilbaum
      {
        Such->Rechts=Temp; // rechts anfügen
        Such=NULL;
      }
      else
      {
        Such=Such->Rechts;
      }
    }
    else // ist schon im Baum -> nichts tun
    {
      Such=NULL;
    }
  }
}
}
```

Beachten Sie, dass der Parameter `wurzel` als Referenz auf einen Zeiger deklariert wurde, um die Wurzel des Baumes verändern zu können. `Such==NULL` wird als Endekennzeichnung für die Schleife verwendet, falls ein Platz zum Einfügen gefunden wurde.

Grundlegende Aufgabe	`binbaum1`

Bauen Sie mit Papier und Bleistift einen binären Baum auf, in den Sie der Reihenfolge nach Dieter, Felix, Cäcilie, Anton, Bertram, Emil, Günter, Klaus aufnehmen.

Grundlegende Aufgabe	binbaum2.cpp

Schreiben Sie ein Programm, das einzugebende Daten (Namen) geordnet in einem binären Baum speichert und dabei jeweils ausgibt, was es tut. Das Einfügen in den Baum soll von einem Unterprogramm oder einer Methode erledigt werden.

```
Eingabe ("-"=Ende): Felix
Baum ist leer.
Eingabe ("-"=Ende): Dieter
Fuege links unter Felix ein.
Eingabe ("-"=Ende): Anton
Kleiner als Felix. Suche links weiter.
Fuege links unter Dieter ein.
Eingabe ("-"=Ende): Cäcilie
Kleiner als Felix. Suche links weiter.
Kleiner als Dieter. Suche links weiter.
Fuege rechts unter Anton ein.
Eingabe ("-"=Ende): -
Ende der Eingabe
```

L5 Bearbeiten von Daten in binären Bäumen

Die Daten, die in einem Baum gespeichert werden, sollen normalerweise auch bearbeitet werden. Dabei muß gewährleistet sein, dass alle gespeicherten Elemente berücksichtigt werden. Außerdem soll, beispielsweise bei der Ausgabe, die sortierte Reihenfolge berücksichtigt werden. Hierzu soll am Beispiel der Ausgabe ein Algorithmus vorgestellt werden.

Der Algorithmus arbeitet rekursiv und funktioniert nach folgendem einfachen Schema: Zuerst den linken Teilbaum des aktuellen Elementes bearbeiten, dann das Element selbst, dann den rechten Teilbaum. Gestartet wird der Algorithmus an der Wurzel, beendet jeweils, wenn der linke bzw. der rechte Teilbaum leer ist.

```
void output(TElement* wurzel)
{
if (wurzel!=NULL)
{
  output(wurzel->Links); // linken Teilbaum bearbeiten
  wurzel->Ausgabe();
  output(wurzel->Rechts); // rechten Teilbaum bearbeiten
}
}
```

Grundlegende Aufgabe	binbaum3.cpp

Erweitern Sie binbaum2.cpp mit einem Unterprogramm oder einer Methode, die die gespeicherten Elemente sortiert ausgibt. Führen Sie dabei einen Zähler für die Tiefe des Baumes mit, so dass Sie eine entsprechende Anzahl Tabulatoren „\t“ dem Namen voranstellen können, um einen Eindruck über den Aufbau des Baumes zu bekommen. (Die Wurzel des Baumes ist in dieser Darstellung dann links, nicht oben.) Bei der Ausgabe soll also vor jedem Namen eine entsprechende Anzahl Tabulatoren ausgegeben werden; Beim Wurzel-Element kein Tabulator, bei den beiden Elementen direkt unter der Wurzel je ein Tabulator, bei denen darunter je zwei Tabulatoren usw.

```
Eingabe ("-"=Ende): Dieter
Eingabe ("-"=Ende): Bertram
Eingabe ("-"=Ende): Felix
Eingabe ("-"=Ende): Anton
Eingabe ("-"=Ende): Cäcilie
Eingabe ("-"=Ende): Emil
Eingabe ("-"=Ende): Klaus
Eingabe ("-"=Ende): Günter
Eingabe ("-"=Ende): -
Ende der Eingabe
                Anton
        Bertram
                Cäcilie
Dieter
                Emil
        Felix
                        Günter
                Klaus
```

L6 Suchen in binären Bäumen

Das Suchen in binären Bäumen ist so effektiv, weil es ein binäres Suchen ist, d.h. in jedem Suchschritt wird jeweils die Hälfte der verbliebenen Daten als nicht zutreffend eliminiert, so dass der Suchalgorithmus nach $_2$log(Anzahl Daten) beendet ist, wenn der Baum optimal gestaltet und nicht entartet ist. Im Beispiel des zur verketteten Liste entarteten Baumes ist die maximale Anzahl der Suchschritte so groß wie die Anzahl der Daten.

Der vorgestellte Suchalgorithmus arbeitet iterativ, ist aber auch rekursiv denkbar. Zunächst wird geprüft, ob mit dem aktuellen Element das gesuchte bereits gefunden wurde. Ist das nicht der Fall, ist das gesuchte Element entweder kleiner oder größer als das aktuelle. Dann wird im linken (kleiner) oder im rechten (größer) Teilbaum weitergesucht. Der Algorithmus beginnt an der Wurzel des Baumes und endet entweder, wenn das gesuchte Element gefunden wurde oder wenn kein linker bzw. rechter Teilbaum mehr existiert.

```
TElement* search(const TElement* wurzel, const string& name)
{
while (wurzel!=NULL) // Beenden, wenn nichts gefunden
{
  if (wurzel->Name==name) // gefunden
  {
    return (TElement *)wurzel; // typecast von const auf nicht const
  }
  if (name<wurzel->Name) // links weitersuchen
  {
    cout << "Suche links von " << wurzel->Name << "." << endl;
    wurzel=wurzel->Links;
  }
  else // rechts weitersuchen
  {
    cout << "Suche rechts von " << wurzel->Name << "." << endl;
    wurzel=wurzel->Rechts;
  }
}
return NULL; // nichts gefunden
}
```

Grundlegende Aufgabe — binbaum4.cpp

Erweitern Sie binbaum3.cpp mit einem Unterprogramm oder einer Methode, die in dem Baum sucht, ob ein bestimmter Name gespeichert ist. Der Zeiger auf das Element mit dem Namen soll zurückgegeben werden, der Nullzeiger, falls keines existiert. Außerdem soll das Unterprogramm ausgeben, was gerade getan wird.

```
Eingabe ("-"=Ende): Dieter
Eingabe ("-"=Ende): Bertram
Eingabe ("-"=Ende): Anton
Eingabe ("-"=Ende): Cäcilie
Eingabe ("-"=Ende): Felix
Eingabe ("-"=Ende): Günter
Eingabe ("-"=Ende): Hans
Eingabe ("-"=Ende): -
Ende der Eingabe
                Anton
        Bertram
                Cäcilie
Dieter
        Felix
                Günter
                        Hans
Suchen von ("-"=Ende): Cäcilie
Suche links von Dieter.
Suche rechts von Bertram.
Cäcilie gefunden.
Suchen von ("-"=Ende): -
Ende.
```

Weiterführende Aufgabe — binbaum5.cpp

Erweitern Sie binbaum4.cpp um folgende Möglichkeiten.

1. Ein Menü mit den Menüpunkten Eingabe, Ausgabe, Suchen, Speichern, Laden, Ende.
2. Programmieren Sie, dass die eingegebenen Daten in eine Datei abgespeichert werden können.
3. Programmieren Sie, dass die abgespeicherten Daten wieder geladen werden können.

Was stellen Sie fest, wenn Sie nach dem Laden die Baumstruktur wieder anschauen? In welcher Reihenfolge müssen Sie die Daten abspeichern, damit sie beim Einlesen wieder in derselben Struktur im Baum stehen?