Einführung in C++

Arne Schmitt

11. Januar 2007

Inhaltsverzeichnis

Teil I.
Die erste Stunde

Die erste Stunde begint traditionell mit dem „Hello World“-Programm.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
  cout << "Hello, world!" << endl;
  return EXIT_SUCCESS;
}
```

Durch Ausführen dieses Programms erhalten wir:

```
Hello, world!
```

Keine große Überraschung.
Jetzt sehen wir uns einmal die einzelnen Elemente des Programms an

#include <...> Die Include Anweisung erklärte dem Compiler welche Bibliotheken wir verwenden wollen.

using namespace std; Wählt den Aktuellen Namensraum. Wir ignorien erst einmal was das genau tut.

Jetzt wird es wichtig! Wir kommen zu main-Funktion. Der Startpunkt aller C++ Programme.

```
int main(int argc, char *argv[])
{
  ...
  return EXIT_SUCCESS;
}
```

Dieser Teil kommt so in nahezu jedem C++ Programm vor. Zwischen den {} stehen die eigentlichen Befehle. Und für den Anfang werden alle unsere Änderungen hier stattfinden. Den Rest des Programms lassen wir erstmal wie er ist.
Jetzt zu unserem ersten Befehl:

```
cout << "Hello, world!" << endl;
```

Wie wir bereits wissen gibt er dern Text „Hello, world!“auf dem Bildschirm aus. Texte werden in C++ immmer in „“eingeschloßen, damit der Compiler weiß, das er das was dort steht nicht verstehen muß.

endl Ist ein vordefinierter Text, ein Steuerzeichen, das dem Computer befiehlt eine neue Zeile anzufangen.

cout Ist der *„standart output stream“* . Eine hochtrabende Bezeichnung, sie besagt aber nur **cout** immer gebraucht wrd wenn wir etwas ausgeben wollen.Um jetzt tatsächlich etwas auszugeben, schieben wir unsere Werte nach **cout**.

Das schreibt sich dann so:

```
cout << Wert;
```

oder wenn wir mehrere Werte ausgeben wollen:

```
cout << Wert1 << Wert2 << Wert3 ...;
```

Werte können dabei nicht nur Texte sein, sondern genausogut auch Zahlen(int,double),einzelen Buchstaben(char) oder Ja/Nein-Aussagen(bool). Als Gegenstück zu **cout** gibt es **cin**, den *“standart input stream„*. Ihn verwenden wir ,wenn wir eine Eingabe vom Benutzer haben wollen. Ähnlich wie bei **cout** schieben wir die Werte aus **cin** in eine **Variable**.

```
 cout << "Anzahl:";
 int z;
 cin >> z;
```

Das bringt uns zu unserem ersten Grundlagenkapitel

1. Variablen

Variablen in einer Programmiersprache sind keine Unbekanten wie in der Mathematik! Eine **Variable** ist ein reservierter Speicherplatz an dem sich der Computer eine ganz bestimmte Art von Information notieren, man spricht auch von ablegen oder speichern, kann. Diese bestimmte Art von Information nennt man den **Typ** eine **Variablen**. Die *Basistypen* die wir im Moment verwenden sind:

bool einen Ja/Nein (true/false) information. 1 bit.

int eine ganze Zahl. 32 bit

char ein einzelnes Zeichen (Buchstabe,Ziffer,Sonderzeichen,...) 8 bit

double eine Kommazahl , 128 bit

Um eine **Variable** zu verwenden muß man sie erst einmal *Deklarieren*, also dem Compiler sagen was für einen **Typ** von **Variable** wir wollen und wie sie heißen soll. Typ Name; Dazu ein paar Beispiele:

```
bool ende;
double preis;
char buchstabe;
int zahl;
```

Jetzt müßen wir die **Variable** initialisieren. Also einen Wert festlegen, den bisher könnte die **Variable** noch jeden möglichen Wert enthalten und wir haben keine Ahnung welchen. Wenn wir nur eine Benutzereingabe speichern wollen, können wir uns das aber auch sparen. Name = Wert; Dazu wieder ein paar Beispiele:

```
ende=true;
preis=42.6;
buchstabe='P';
zahl=-5;
```

1.1. Beispiele

Jetzt sind wir soweit ein paar einfache Programme zu schreiben.

1.Kreisumfang berechnen

```
double r;
cout << "Radius:" ;
cin >> r;
cout << "Umfang:" << r*3.1415 << endl;
```

2.Zahl der Tage/Stunden/Sekunden in x Jahren berechnen (grob)

```
int jahre;
cout << "Jahre:";
cin >> jahre;
cout >> "in␣" << jahre << "␣Jahren␣sind␣ca.␣" \
```

```
        << jahre *365.25 << " Tage vergangen" << endl;
cout >> "in " << jahre << " Jahren sind ca. " \
        << jahre *365.25 * 24 << " Stunden vergangen" << endl;
cout >> "in " << jahre << " Jahren sind ca. " \
        << jahre *365.25 * 24 * 3600 << " Sekunden vergangen" << endl;
```

2. Schleifen

Häufig muß man in einem Programm etwas mehrere Male hintereinander machen, und oft weiß man beim schreiben noch nicht einmal wie oft. C++ bietet zu diesem Zweg verschiedene Arten von Schleifen. Eine Schleife hat immer eine Bedingung, die entscheidet wie lange sie läuft , und die Befehle die wiederholt werden sollen.
In C++ gibt es drei Grundtypen von Schleifen:

1. for
2. while
3. do while

2.1. Begriffserklärung

BLOCK ist entwender ein einzelner Befehl oder beliebig viele Befehle in geschweiften Klammern.

```
 // BLOCK1
 cout << "Ich bin ein BLOCK" << endl;
// oder BLOCK2
{
 cout << "Ich bin auch ";
 cout << "ein Block";
 cout << endl;
 cout << "Aber auf mehr als einem Befehl!";
 cout << endl;
}
```

BEDINGUNG ist für C++ einfach etwas das wahr oder falsch zurück gibt, dabei gilt eine Zahl ungleich 0 als Wahr und 0 als falsch.

```
 // Bedingung1
2==4; // false
2==2; // true
true; // true
2>-4; // true
0; // false
! 0; // true

int a=12,c=4;
int *b= &a;
a; // true , da !=0
b; // true , Zeigt auf a;
a==b; // höchstwarscheinlich false
a=c; // true, den a wird auf c (4) gesetzt !=0
```

WEITER

START

2.2. Die For-Schleife

Die For-Schleife ist ideal um zu zählen oder eine bekannte Zahl von Wiederholungen zu machen.

for(START;BEDINGUNG;WEITER) BLOCK

```
for(int i=0;i<10;i++)
{
 cout << "Durchlauf Nr" << i+1 << endl;
}
```

2.3. Die While-Schleife

Die While-Schleife wird verwendet wenn man keine Ahnung hat wieviele Wiederholungen es geben wird.

while(BEDINGUNG) BLOCK

```
int i=0;
while(i<10)
{
 cout << ":";
 cin >> i;
}
cout << "Fertig" << endl;
```

2.4. Die Do-While-Schleife

Die Do-While-Schleife ist der While-Schleife sehr ähnlich. Der Hauptunterschied ist, daß sie mindestens einmal durchläuft bevor sie die Bedingung überprüft.

do BLOCK while(BEDINGUNG)

```
int i=0;
do
{
 cout << ":";
 cin >> i;
} while(i<10)
cout << "Fertig" << endl;
```

2.5. Beispiele

1.Zählen von 1 bis z

```
 int z;
 cin >> z;
 for(int i=0;i<z;i++)
 {
   cout << i+1 << ",";
 }
 cout << endl;
```

2.Zählen der Stellen in einer ganzen Zahl (int)

```
 int z;

 cout << "Stellenzählen" << endl;
 cout << "Zahl:";
 cin >> z;

 int stellen=0;
 while(z!=0)
 {
```

```
    stellen++;
    z = z /10; // Zahl um eine Stelle verschieben
  }
  cout << "Es sind " << stellen << " Stellen" << endl;
```

3.Kommazahl aufsummieren bis der Benutzer 0 eingibt

```
  cout << "Summe" << endl;
  double summe=0.0,eingabe;
  do
  {
    cin >> eingabe;
    summe += eingabe;
  }
  while(eingabe!=0);

  cout << "Summe: " << summe << endl;
```

3. Verzweigungen

Verzweigungen dienen dazu, um je nach Situation unterschiedliche Befehle auszuführen.

3.1. if

Die einfachste Verzweigung ist **if**.
Der prinzipielle Aufbau der **if**-Anweisung lautet:

If(Bedingung) BLOCK else BLOCK

Beispiele:

```
if( x == 1)
{
  cout << "X ist eins\n";
}
else
{
  cout << "X ist nicht eins.\n";
}

if(true) cout << "Hallo\n";
```

3.2. switch-case

Of muß man ein Variable auf verschiedene mögliche Werte überprüfen. Für diese sogenannte Mehrfachverzweigung gibt es eine besondere Anweisung: **switch**. Zu switch gehören außerdem **case** und **default**.
Der grundsätzliche Aufbau:

```
switch(VARIABLE)
{
  case WERT1:
        BEFEHL
        ....
        break;
  case WERT2:
        BEFEHL
        ...
        break;
  case ...:
  default:
```

BEFEHL
...
}

Für jeden möglichen Wert auf den man prüfen möchte, schreibt man eine *case WERT:-Zeile* gefolgt von den in dem Fall auszuführenden Befehlen. Am Ende dieser Befehle schreibt man in der Regel ein **break** , macht man das nicht, für der Computer auch die Befehle der folgenden *Cases* aus bis er auf ein break oder das Endes des *switch-Blocks* stößt.Wenn keiner der *Cases* zutrifft, werden die Befehle hinter **default:** ausgeführt.
Beispiel:

```
int x;
cin >> x;
Switch(x)
{
case 1:
        cout << "Eine␣Möglichkeit";
        break;
case 2:
        cout << "Eine␣Möglichkeit";
        break;
case 2:
        cout << "Eine␣Möglichkeit";
        break;
case 8:
        cout << "Eine␣Möglichkeit";
        break;
default:
        cout << "Irgendwas␣anderes\n";
}
```

4. Arrays

Durch Schleifen ist es einfach Dinge zu wiederholen. Aber was ist wenn man z.B. die Eingabe einer Zahl 10 mal wiederholen möchte?

```
int x;
for(int i=0;i<10;i++)
{
 cout << "Bitte␣geben␣sie␣eine␣Zahl␣ein:";
 cin >> x;
}
```

Jetzt haben wir ein Problem. Es werden 10 Zahlen eingegeben und wir kennen nur die letzte davon. Die Lösung sind sogenannte **Arrays** array] oder Felder, das sind Blöcke von gleichartigen Variablen, die im Speicher direkt hintereinander liegen und durch eine gemeinsamen Namen und eine Nummer angesprochen werden.
Ein **Array** *declariert* man:

TYP NAME[ANZAHL]; oder

TYP NAME[] = {WERTE,WERT, ... ,WERT};

Damit sie unser Programm jetzt so aus:

```
int x[10];
for(int i=0;i<10;i++)
{
 cout << "Bitte␣geben␣sie␣eine␣Zahl␣ein:";
 cin >> x[i];
}
```

4.1. Beispiele

```
int zahlen[10];
int zahlen2[]={4,8,63,476,3,24,45};
zahlen[0]=1;
zahlen[5]=34;
cout << zahlen[0] << zahlen[1] << zahlen[5];
for(int i=0;i<10;i++) zahlen[i]=5*i+3;
for(int i=0;i<10;i++) cout << i << '\t' << zahlen[i] << endl;
for(int i=0;i<6;i++) cout << i << '\t' << zahlen2[i] << endl;
```

5. Pointer

Pointer(Zeiger) enthalten die Speicheradresse einer Information.

Typ[1] *name;

Ist also ein Zeiger auf eine Information vom Typ *Typ* mit dem Namen *name* Um an den Wert zu kommen auf den der Pointer zeigt muß an ihn *dereferenzieren*:

```
cout << *name; // Schreibt den Wert auf den name Zeigt
cout << name; // Schreibt die Addresse an der der name steht
*name = "Neuer Name"; // Schreibt "Neuer Name" in die Variable,
// auf die name zeigt
name = 1234; // Setzt den Zeiger auf die Addresse 1234
//Etwas ,daß man nie tun sollte! es sei den man weiß ganz genau was man da macht.
```

5.1. Beispiele

```
int zahl=2;
int zahl2= 15;
int zahlen[5]={3,5,2,4,0};
int *zeiger=&zahl;
cout << zeiger << " " << *zeiger << "\n";
*zeiger = 5;
cout << zeiger << " " << *zeiger << "\n";
zeiger = &zahl2;
cout << zeiger << " " << *zeiger << "\n";
zeiger = zahlen;
while(*zeiger)
{
        cout << zeiger << " " << *zeiger << "\n";
        zeiger++;
}
```

6. Referenzen

Wärend **Zeiger** schon zum Sprachumfang von C gehörten, sind **Referenzen** erst mit C++ hinzugekommen. **Referenzen** sind im grunde *versteckte Pointer*, also Pointer deren Wert man <u>nicht auslesen oder verändern</u> kann. Das bedeutet auch ,daß man eine Referenz nicht *dereferenzieren* muß um den Wert der eigenlichen Variable zu erhalten. Sie sind also haupsächlich dazu geeignet auf die gleiche Variable unter zwei verschiedenen Namen oder von verschiedenen Stellen aus zuzugreifen.

Typ &name;

Hier sieht man auch sehr gut, daß es bei C++ nicht ausreicht sich einfach nur zu merken, welche Funktion ein bestimmtes Symbol hat. & genau wie * tun sehr unterschiedliche Dinge, je nach dem wo sie eingesetzt werden !

[1] Typ kann für jeden Daten/Variablentyp stehen den C++ kennt. z.B. int,double,string,... aber auch selbstdefinierte Typen.

6.1. Beispiele

```
int zahl=2;
int &ref = zahl;
cout << ref << ' ' << zahl << "\n";
ref = 4;
cout << ref << ' ' << zahl << "\n";
zahl++;
cout << ref << ' ' << zahl << "\n";
```

Teil II.
Die ersten Schritte zum größeren Programm

Heute beschäftigen wir uns mit einem praktischen Beispiel, um das Zusammenspiel der Grundelemente zu demonstrieren.

7. Die Notenverwaltung v1

7.1. Anforderungen

Unser Programm soll die folgenden Funktionen beherschen:

1. Bis zu 10 Schulnoten (1-6) speichern
2. Eingabe von Noten
3. Ausgabe der Noten
4. Den Durchschnitt der Noten berechnen
5. Das Programm beenden

7.2. Der Entwurf - Analyse

Ein guter Ansatzpunkt für den Entwurf eines Programms ist es, sich erst einmal zu überlegen, welche Art Daten man verarbeiten will. In unserem Beispiel sind das Schulnoten. Dafür gibt es verschiedene Darstellungsarten.

1. 1,2,3,4,5,6
2. 1,1-,1-2,2+,2,....
3. 1,1.25,1.5,....
4. A,B,C,D,E,F

Die einfachste Möglichkeit scheit die 3. zu sein, den wir haben ja bereits etwas das Kommazahlen speichern kann. Der Typ double. Davon brauchen wir jetzt 10 Stück. Hier bietet sich ein Area an. Jetzt müssen wir uns allerdings auch noch merken, wo die nächste Note hin soll. Sonst würden wir die gespeichete Note ja jedesmal ersetzen. Da wir die Noten in einem Area speichern, kann man die position als ganze Zahl angeben, also als int.

```
double Noten[10];
int position=0;
```

Als nächstes sollte man einen Blick auf den Ablauf des Programms werfen. In unserem Fall kennen wir den Ablauf nicht. Der Benutzer muß entscheiden was er als nächstes tun will. In so einem Fall verwendet man gerne ein Menu. D.H, man presentiert dem Nutzer die Liste der möglichen Funktionen und läst ihn auswählen.

```
cout << "\n\tMenu\n";
cout << "1: Eingeben\n";
cout << "2: Ausgeben\n";
cout << "3: Durchschnitt\n";
cout << "0: Beenden\n";
cout << ":";
int eingabe;
cin >> eingabe;
switch(eingabe)
```

```
{
        ...
}
```

Und jetzt zu den einzelnen Funktionen:

Eingabe einer Note
Als erstes sollten wir den Nutzer darüber informieren was wir von im erwarten.

```
cout << "Bitte geben sie eine Note (1-6) ein:";
```

Anschließend müssen wir die Eingabe einlesen und prüfen ob es wirklich eine Note zwischen 1 und 6 ist. Außerdem sollten wir kontrollieren ob wir noch Platz für eine weitere Note haben. Jetzt können wir die Note am richtigen Platz abspeichern und sind damit fertig.

```
double note;
cin >> note;
if(position>=10) break; // Leider kein Platz mehr
if(note > 6 || note < 1  )
  cout << "Falsche Eingabe\n";
else
{
  Noten[position] = note;
  position += 1;
}
```

Ausgabe der Noten
Das Ausgeben besteht eigentlich nur aus einer Schleife um alle Note aufzuzählen.

```
cout << "\n\tNoten:\n";
for(int i=0; i<position;i++)
  cout << i+1 << ": " << Noten[i] << "\n";
```

Durchschnitt berechnen
d Durchschnitt

N_i Die $i$te Note

n Die Anzahl der Noten

$$d = \frac{\sum_i N_i}{n}$$

in C++:

```
double sum=0.0; // Irgendwo müssen wir uns da die Summe merken
for(int i=0; i<position;i++)
  sum += Noten[i];
cout << "\tDurchschnitt:" << sum / position << endl;
```

7.3. Der Quelltext

Und hier sind jetzt alle Teile zu einem funktionirenden Programm zusammengesetzt.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
  cout << "Notenverwaltung" << endl;
  bool beenden=false;
  double Noten[10];
```

```
    int position=0;

    do
    {
        cout << "\n\tMenu\n";
        cout << "1: Eingeben\n";
        cout << "2: Ausgeben\n";
        cout << "3: Durchschnitt\n";
        cout << "0: Beenden\n";
        cout << ":";
        int eingabe;
        cin >> eingabe;
        double sum=0.0;
        switch(eingabe)
        {
            case 1:
                // Eingeben
                cout << "Bitte Note eingeben:";
                double note;
                cin >> note;
                if(position>=10) break;
                if(note > 6 || note < 1  )
                    cout << "Falsche Eingabe\n";
                else
                {
                    Noten[position] = note;
                    position += 1;
                }
                break;
            case 2:
                // Ausgeben
                cout << "\n\tNoten:\n";
                for(int i=0; i<position; i++)
                    cout << i+1 << ": " << Noten[i] << "\n";
                break;
            case 3:
                // Durchschnitt

                for(int i=0; i<position; i++)
                    sum += Noten[i];
                cout << "\tDurchschnitt:" << sum / position << endl;
                break;
            case 0:
                beenden = true;
                break;
            default:
                cout << "unbekannte Eingabe\n";
        }
    }
    while(!beenden);

    return EXIT_SUCCESS;
}
```

7.4. Erweiterungen

Zum üben bieten sich jetzt einige Erweitungen an.

1. Weitere Statistikfunktionen (z.B. Standartabweichung) hinzufügen

2. Noten löschen (Alle auf einmal)
3. Noten löschen (einzeln)
4. Mit diesem Programm als Vorlage, ein Telefonbuch schreiben.

Teil III. Strukturen

Wie man sieht, werden Programme auf diese Art schnell unübersichtlich. Um das zu vermeiden, gibt es verschiedene Methoden um Anweisungen und Daten etwas übersichtlicher und strukturierter zu schreiben. Außerdem spart man bei größeren Programmen schnell eine Menge schreiberei.

8. Funktionen

Beginnen wir mit einem Beispiel:

```
int main()
{
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 cout << "Text1" << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 cout << "Text2" << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 cout << "Text3" << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
 cout << "Text4" << endl;
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
}
```

Sieht recht aufwending und unübersichtlich aus. Wenn wir genauer hinsehen, erkennen wir ,daß sich eine Menge idenitsch wiederholt. Jetzt lagern wir den sich wiederholenden Teil aus und geben ihm einen Namen.
Damit sieht unser Beispiel jetzt so aus:

```
void linie()
{
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
}

int main()
{

 linie();
 cout << "Text2" << endl;
 linie();
 linie();
 cout << "Text3" << endl;
 linie();
 linie();
 cout << "Text4" << endl;
 linie();
}
```

Schon besser, aber wir haben immernoch eine Menge Wiederholungen. Diesmal aber nicht identisch. Macht aber nichts, wir langern sie trotzdem aus. Jetzt müssen wir bei Aufrufen aber noch den Text mit angeben.

```
void linie()
{
 for(int i=0;i<40;++i) cout << '-';
 cout << endl;
}

void rahmen(string s)
{
 linie();
 cout << s << endl;
 linie();
}

int main()
{
 rahmen("text1");
 rahmen("text2");
 rahmen("text3");
}
```

Jetzt ließt sich das Hauptprogramm schon sehr viel Besser und wir mußten auch nicht so viel schreiben. Außerdem können wir das Aussehen des Rahmens nun ganz einfach ändern ohne darauf achten zu müssen wo er im Programm überall verwendet wird. z.B.:

```
void linie()
{
 for(int i=0;i<40;++i) cout << '*'; // Stern statt -
 cout << endl;
}

void rahmen(string s)
{
 linie();
 cout ''*' << s;
 for (int i=0;i<38-s.length();++i) cout << ' ';
 cout << '*' << endl;
 linie();
}

int main()
{
 rahmen("text1");
 rahmen("text2");
 rahmen("text3");
}
```

Diese ausgelagerten Programmteile nennt man *Unterprogramme*, oder in C++ **Funktionen**. Eine **Funktion** hat , wie eine **Variable** .einen Namen und einen Typ. Der Typ in diesem Beispiel ist **void** und bedeutet schlicht "nichts", muß aber trozdem hingeschrieben werden. Zusätzlich kann eine **Funktion** ,wie in der Mathematik auch, **Parameter** haben. Also zusätzliche **Variablen** von denen das Ergebnis abhängt. Diese stehen in Klammern hinter dem Namen. Damit sieht die Deklaration einer **Funktion** so aus:

TYP NAME(PARAMETER,PARAMETER,...);

Wenn wir eine **Funktion** aufrufen wollen, schreiben wir einfach ihren Namen, gefolgt von den Werten der **Parameter** in Klammern. Die **Klammern müssen** wir auch machen wenn es gar

keine **Parameter** gibt, damit der Computer weiß, das wir die **Funktion** *aufrufen* möchten, und nicht bloß die *Speicheraddresse* der **Funktion** erfahren wollen.
Eine Funktion findet man in jedem Programm:
Die Main-Funktion, die ungefähr so aussieht.

```
int main(int argc, char *argv[])
{
        ...
  return EXIT_SUCCESS; // Alles in Ordnung
}
```

Die Main-Funktion wird bei Start des Programms aufgerufen. Und wenn sie endet, endet auch das Programm. Sie hat zwei Parameter ,mit denen wir uns im Moment nicht näher befassen wollen, und die man in vielen Fällen auch weglassen kann.
Funktionen können auch im mathematischen Sinn Funktionen sein ,wie z.B. eine Durchschnitt-Funktion

```
double durchschnitt(double Noten[],int position)
{
        double sum=0.0;
        for(int i=0; i<position;i++)
          sum += Noten[i];
        return sum / position;
}

int main()
{
 double Noten[]={1,2,4,2,2,1,3};
 cout << "Durchschnitt:" << durchschnitt(Noten,6) << endl;
}
```

8.1. Die Notenverwaltung v2

Hier ist unsere Notenverwaltung jetzt auf mehrere Funktionen aufgeteilt.

```
#include <iostream>
#include <cstdlib>

using namespace std;

// Damit unsere Funktionen die Noten sehen können,
// müssen wir sie hier oberhalb der Funktionen declarieren
double Noten[10];
int position=0;

void NoteEingeben()
{
        // Eingeben
        cout << "Bitte Note eingeben:";
        double note;
        cin >> note;
        if(position >=10)
        {
                cout << "Kein Platz mehr\n";
                return;
        }
        if(note > 6 || note < 1  )
                cout << "Falsche Eingabe\n";
        else
        {
```

```
            Noten[position] = note;
            position += 1;
        }
}

void NotenAusgeben()
{
        // Ausgeben
        cout << "\n\tNoten:\n";
        for(int i=0; i<position;i++)
                cout << i+1 << ":␣" << Noten[i] << "\n";
}

double durchschnitt()
{
        // Durchschnitt
        double sum=0.0;
        for(int i=0; i<position;i++)
                 sum += Noten[i];
        return sum/position;
}

void DurchschnittAusgeben()
{
        cout << "\tDurchschnitt:" << durchschnitt() << endl;
}

int main(int argc, char *argv[])
{
  cout << "Notenverwaltung" << endl;
  bool beenden=false;

  do // Hauptschleife
  {
          cout << "\n\tMenu\n";
          cout << "1:␣Eingeben\n";
          cout << "2:␣Ausgeben\n";
          cout << "3:␣Durchschnitt\n";
          cout << "0:␣beenden\n";
          cout << ":";
          int eingabe;
          cin >> eingabe;
          double sum=0.0;
          switch(eingabe)
          {
                  case 1:
                          NoteEingeben();
                          break;
                  case 2:
                          NotenAusgeben();
                          break;
                  case 3:
                          DurchschnittAusgeben();
                          break;
                  case 0:
                          beenden = true;
                          break;
                  default:
```

```
                            cout << "unbekannte Eingabe\n";
          }
    }
    while(!beenden);

    return EXIT_SUCCESS;
}
```

9. Dateien

Die Lösung für das Problem ,daß Informationen mit den des Progamms verloren sind, heißt Dateien. In C++ werden Dateien meißt als streams behandelt. Also also Datenströme aus denen man nacheinder Daten lesen oder in die man nacheinander Daten schreiben kann.
Mit zwei Streams hatten wir bereits zutun:

cin Der Inputstrean. Meist die Tastatureingaben des Benutzers.

cout Der Outputstream. Der Inhalt erscheint normalerweiße auf dem Bildschrim.

Wir können num eigene Streams definieren, die ihre Daten z.B. auf Festplatte,Diskette oder USB-Stick speichern und von dort auch wieder lesen.
Um Dateien verwenden zu können müssen wir eine neue Includezeile hinzufügen.

```
#include <fstream>
```

jetzt können wir ein Datei zum Schreiben

```
ofstream output("Testdatei");
output << "Das hier steht gleich auf der Festplatte" << endln;
output.close();
```

oder auch zum lesen öffnen.

```
ifstream input("Testdatei");
string s;
input >> s;
input.close();
cout << s;
```

9.1. Die Notenverwaltung v3

Damit bekommt unsere Notenverwaltung ein Gedächnis:

```
void Schreiben(char* filename)
{
        ofstream out(filename);
        out << "#Notenverwaltung_v3\n";
        for(int i=0;i<position;i++)
                out << Noten[i] << "\n";
        out.close();
}

void Lesen(char* filename)
{
        ifstream in(filename);
        string s;
        in >> s;
        position =0;
        while(!in.eof() && position <10)
        {
                double note;
```

```
                in >> note;
                Noten[position] = note;
                position++;
        }
}
```

10. Datenstrukturen

Wenn wir schon dabei sind, unser Programm übersichtlicher zu machen, führen wir gleich noch zwei weitere Konzepte ein.

10.1. struct

Eine C++ Struktur (struct) fast zusammengehörende Daten zusammen.

```
struct _notenbuch
{
 double Noten[10];
 int position;
};

_notenbuch buch;
buch.position=0;
buch.Noten[2]=1.5;
```

10.2. Dynamische Speicherverwaltung: new / delete

Im folgenden werden wir häufinger **dynamisch** Speicher anfordern und abgeben müßen. Dynamisch steht in diesem Fall für "während das Programm läuft". Im Gegensatz dazu waren alle **Variablen** die wir bisher verwendet haben dem Computer schon beim compileren bekannt und der konnte schon vor dem Start des Programms genau festlegen was wo im Speicher liegt.
Variablen die wir über *dynamische Speicherwaltung* erhalten, haben keine Namen. Wir gekommen nur die Zeiger auf ihre Speicherblocke. Um dynamischen Speicher anzufordern verwendet man den Operator **new** und um ihn freizugeben **delete**;

Beispiele für new und delete

```
int main()
{
int *a = new int; // a zeigt auf speicherplatz für einen int
int *ar = new int[20]; // ar zeigt auf specherplatz fuer 20 int;

*a=12;
ar[2] = 12;
ar[0] = 2;
cout << "ar[0]=" << ar[0] << endl;
cout << "*array=" << *ar << endl;
cout << "ar[2]=" << ar[2] << endl;
cout << "*(ar+2)=" << *(ar+2) << endl;

cout << a << " zeigt auf " << *a << endl;
delete a;
a = new int;
cout << a << " zeigt auf " << *a << endl;

/* Wenn wir den Speicher ,d.h. die Variablen, nicht mehr brauchen
sollten wir ihn freigeben */
delete a;
delete[] ar;
```

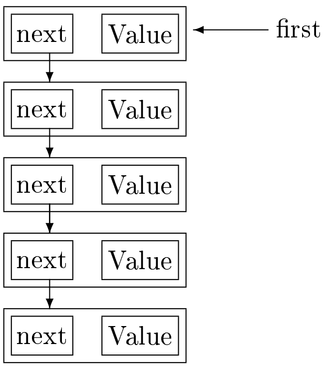
```
/* Wenn man einen Dynamischen Wert mit delete freigegeben hat,
sollte man immer alle Pointer die darauf zeigen auf 0 setzten */
a=0;
ar=0;
}
```

10.3. Verkettete Liste

Bisher haben wir zum Speichern der Informationen Arrays verwendet. Bei einem Array muß man aber vorher wissen, wieviele Elemente es Maximal werden sollen. Oft kann man das aber beim Schreiben des Programms nicht feststellen. Eine der gängigen Lösungen für dieses Problem, sind Verkette Listen.

10.3.1. Die einfach verkettete Liste

Bei einem Array fordern wir einen großen zusammenhängenden Speicherblock an, in den alle Werte hineinpassen. Bei einer einfach verketteten Liste fordern wir für jeden Wert eine kleinen Speicherblock an. Dafür speichern wir für jeden Wert ,wo der nächste Wert zu finden ist, oder 0 wenn es der letzte Wert ist.



struct Entry { Entry *next; int Value; };

Entry *first;

Beispiel einer einfach verketteten Liste

```
/************************************************************
** Beispiel fuer die Umsetzung einer einfach verketteten Liste *
************************************************************/

/* Unsere Datenstruktur */
struct Entry
{
  Entry *next; // Zeiger auf den naechsten Eintrag;
  int Value; // Beispielwert
};

Entry *first=0; // Zeiger auf den ersten Eintrag;

/* Neuen Wet am Anfang einfuegen */
Entry *insertValueFirst(int v)
{
  Entry *neu = new Entry();
  neu->Value=v;
  neu->next=first;
  first = neu;
}

/* Neuen Wert am Ende anhaengen */
Entry *appendValue(int v)
{
  Entry *neu = new Entry();
  neu->Value = v;
```

```
  neu->next = 0;
  if(first)
  {
    Entry *i=0;
    for(i=first;i->next; i=i->next) ;
    i->next = neu;
  }
  else
   first = neu;
  return neu;
}

/* Eintrag mit bestimmten Wert finden */
Entry *findValue(int v)
{
  for(Entry *i=first;i;i=i->next)
  {
    if(i->value==v) return i;
  }
}

/* Eintrag entfernen */
void removeEntry(Entry *e)
{
  if(e==first) first = e->next;
  else
  {
    Entry *i=0;
    for(i=first;i && i->next!=e;i=i->next) ;
    if(!i) return; else i->next = e->next;
  }
  delete e;
}
```

Vorteile der einfach verketteten Liste:

- einfach zu verstehen
- sehr geringer Overhead, wir brauchen pro Wert nur einen Zeiger zusätzlich. (In unserem Beispiel verdoppelt sich dadurch allerdings der Speicherverbrauch!)
- einfach der Reihe nach auszulesen oder zu durchsuchen
- man kann sehr schnell den ersten Wert auslesen
- man kann sehr schnell den nächsten Wert auslesen
- man kann sehr schnell einen Wert am Anfang einfügen.
- man kann sehr schnell den ersten Wert entfernen.

Nachteile der einfach verketteten Liste:

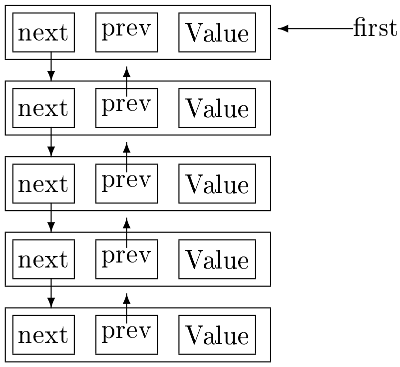
- es ist teuer (an rechenzeit) den n-ten Wert auszulesen
- es ist teuer die Werte in irgend einer anderen Reihenfolge als vom Anfang zu Ende anzuarbeiten.
- es ist im verhältniss besonder aufwänding den Vorgägner eines Eintrags zu finden.
- es ist teuer einen Wert am Ende anzuhängen.
- es ist teuer und aufwänding einen Wert an beliebier Stelle einzufügen.

- es ist teuer und aufwänding einen Wert an beliebier Stelle zu entfernen.
- es ist teuer und aufwänding einen Wert an beliebie Stelle zu verschieben.

Man kann das Einfügen und Auslesen am Ende stark vereinfachen und beschleunigen in dem man zusetztlich einen Zeiger auf den letzten Eintrag verwaltet.

10.3.2. Die doppelt verkettete Liste

Um einige der Nachteile der einfachverketteten Liste zu beheben, kann man zusätzlich zum nächsten auch den vorherigen Eintrag speichern. Damit verbraucht man zwar mehr Speicher. Aber fast alle Operationen innerhalb der Liste werden einfacher und schneller.



struct Entry { Entry *next,*prev; int Value; };

Entry *first;

Beispiel einer dopppelt verketteten Liste

```
/************************************************************
** Beispiel fuer die Umsetzung einer doppelt verketteten Liste *
************************************************************/

/* Unsere Datenstruktur */
struct Entry
{
  Entry *next; // Zeiger auf den naechsten Eintrag;
  Entry *prev; // Zeiger auf den vorherigen Eintrag;
  int Value; // Beispielwert
};

Entry *first=0; // Zeiger auf den ersten Eintrag;

/* Neuen Wet am Anfang einfuegen */
Entry *insertValueFirst(int v)
{
  Entry *neu = new Entry();
  neu->Value=v;
  neu->next=first;
  neu->prev=0;
  first = neu;
}

/* Neuen Wert am Ende anhaengen */
Entry *appendValue(int v)
{
  Entry *neu = new Entry();
  neu->Value = v;
  neu->next = 0;
  neu->prev = 0;
  if(first)
  {
```

```
    Entry *i=0;
    for(i=first;i->next; i=i->next) ;
    i->next = neu;
    neu->prev = i;
  }
  else
   first = neu;
  return neu;
}

/* Eintrag mit bestimmten Wert finden */
Entry *findValue(int v)
{
  for(Entry *i=first;i;i=i->next)
  {
    if(i->value==v) return i;
  }
}

/* Eintrag entfernen */
void removeEntry(Entry *e)
{
  if(e==first)
  {
        first = e->next;
        first->prev = 0;
  }
  else
  {
        Entry *n = e->next;
        Entry *p = e->prev;
        p->next = n;
        n->prev = p;
  }
  delete e;
}
```

Vorteile der doppelt verketteten Liste:

- einfach zu verstehen
- einfach Vorwärts oder Rückwärts auszulesen oder zu durchsuchen
- man kann sehr schnell den ersten Wert auslesen
- man kann sehr schnell den nächsten Wert auslesen
- man kann sehr schnell den vorherigen Wert auslesen
- man kann sehr schnell einen Wert am Anfang einfügen
- man kann schnell einen Wert an beliebiger Stelle einfügen
- man kann sehr schnell den ersten Wert entfernen.
- man kann schnell einen Wert an beliebiger Stelle einen Wert an beliebiger Stelle entfernen.

Nachteile der doppelt verketteten Liste:

- es ist teuer (an rechenzeit) den n-ten Wert auszulesen
- der Overhead ist doppelt so groß wie bei der einfach verketteten Liste.
- es ist teuer einen Wert am Ende anzuhängen.

Und auch hier kann man das Einfügen und Auslesen am Ende stark vereinfachen und beschleunigen in dem man zusetztlich einen Zeiger auf den letzten Eintrag verwaltet.

10.4. Binäre Bäume

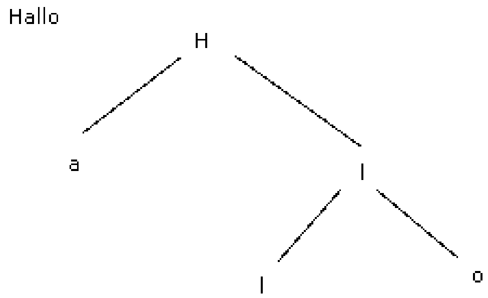
Bisher waren unsere Datenstrukturen eindimensional. Bäume sind ein Beispiel für 2 dimensionale Stukturen. Sie werden verwendet um Hierarchien oder sturkturierte Informationen zu speichern. Der einfachste Fall sind binäre Bäume. Jeder Knoten/Blatt hat zwei Pointer zusätzlich zur eigentlichen Information, einer nach links und einer nach rechts. Für den gesammten Baum gilt eine Vorschrift, wann ein neuer Knoten rechts und wann er links angehängt wird. Zeigt der entsprechende Pointer bereits auf einen anderen Knoten, so wir versucht den neuen Knoten dort anzuhängen, bis man am Ende bei einem freien Pointer (0) angekommen ist.
Einer der Vorteile eines binären Baumes ist, das der Zugriff auf einen bestimmten Wert oder Knoten <u>im mittel deutlich schneller</u> ist als bei einer verketteten Liste.
Da wir im Moment noch nicht so weit sind den Computer koplizierter Grafiken anzeigen zu lassen, behandeln wir binäre Bäume zunächst auf den Papier, da man sonst die Struktur des Baumes nicht erkennen kann.

10.4.1. Zeichenketten sortieren 1

Unser erster Baum soll eine Zeichenkette speichern. Die Vorschrift lautet:

> ist der neue Wert größer als der Wert des aktuellen Knotens, gehört der neue Wert nach rechts, ansonsten nach links.

Als machen wir mal den Versuch und tragen "Hallo“ in einen leeren Baum ein.




```
 struct Knoten {
char Wert;
Knoten *links;
Knoten *rechts;
};

Knoten wurzel=0;

/* füge einen neuen Wert n in den Baum ein */
void insert(char n)
{
  Knoten *neu = new Knoten;
  neu->Wert = n;
  neu->rechts = 0;
  neu->links = 0;
  if(wurzel==0)
  {
    wurzel = neu;
  }
  else
  {
   Knoten *i = wurzel;
```

```
    while(i)
    {
     if(i->Wert<n && i->rechts!=0)  i=rechts;
     if(i->Wert>=n && i->links!=0)  i=links;
     if(i->Wert<n && i->rechts==0)  { rechts = neu;i=0; }
     if(i->Wert>=n && i->links=0)   { links  = neu;i=0; }
    }
  }
}

/* Finde einen Knoten mit Wert==w unterhalb von Knoten r */
Knoten *findeKnoten(Knoten *r,char w)
{
        if(r==0) return 0;
        if(r->Wert == w ) return r;
        if(r->Wert < w ) return findeKnoten(r->rechts,w);
        else return findeKnoten(r->links,w);
}

/* alle Knoten von links nach rechts ausgeben */
void ListKnoten(Knoten *r)
{
        if(r==0) return;
        cout << r->Wert << endl;
        if(r->links) ListKnoten(r->links);
        if(r->rechts) ListKnoten(r->rechts);
}
```

Was man an diesem Beispiel gut sehen kann, ist eine Technik namens Rekursion, bei der ein Funktion sich selbst wieder mit neuen Parametern aufruft. Diese Technik ist beim Umgang mit komplexeren Strukturen oft sehr elegant und schnell zu schreiben.
Sie hat aber auch einige Nachteile:

- Sie verbraucht leicht viel Speicherplatz auf dem Stack, was man oft nicht bemerkt bis das Programm plötzlich abbricht.
- Sie ist nicht besonders schnell in der Ausführung.
- Wenn man nicht aufpasst, kann die Rekursion endlos weitergehen bis der Speicher auf dem Stack verbraucht ist.

Teil IV.
Die Grundlagen der Objekt Orientierten Programierung (OOP)

11. Objekte

11.1. Klassen

Die Klasse(**class**) geht noch einen Schritt weiter und bringt auch noch die zu dem Daten gehörenden Funktionen (**methoden**) mit dazu. Außerdem gibt es Zugriffsstuffen, Vererbung und Konstruktoren/Destruktoren.

```
class Klassenname : public Elternklasse {
        Typ1 private_Variable;
        static Typ5 klassen_Variable;
        ...
 public:
        Typ2 oeffentliche_Variable;
        ...
        void oeffentliche_funktion();
        virtual void oeffentliche_funktion2();
        static void klassen_funktion();
        ....
protected:
        Typ3 vererbbare_variable;
        Typ3 vererbbare_funktion(Typ4 t);
};
```

Es folgen die Erklärungen der einzelnen Komponenten:

11.2. Kapselung und Zugriffsstuffen

Um sicher zugehen, daß nur die richtigen Funktionen verwendet werden um auf eine Objekt zuzugreifen, gibt es drei Zugriffsstuffen.

public Wie bei Strukturen. Jeder darf auf alles zugreifen.

private Nur die Methoden , die zu dieser Klasse gehören haben Zugriff.

protected Wie private, aber auch Methoden von abgeleiteten Klassen haben Zugriff.

```
class test {
 int a ; // private, da nichts anderes angegeben
public:
 int b; // public;
 int c; // public;
protected:
 int d; // protected;
private:
 int e; // private , diesmal explizit angegeben
};
```

11.3. Klasse vs. Instanz

Im Zusammenhang mit Objektorientiertem Programmieren ist immer wieder von Objekten,Klassen und Instanzen die Rede. Es ist wichtig die Unterschiede zwischen diesen Begriffen zu verstehen.

Klassen sind so etwas wie Vorlagen. Sie geschreiben den Aufbau und die Funktion von Objekten.

Instanzen einer Klasse sind die Objekte die nach diese Vorlage erstellt werden.

11.4. static Members

Dieser Unterschied zwischen Klasse und Instanz, zeigt sicht auch in der Deklaration einer Klasse. Und zwar durch das Schlüsselwort static. Eine mit static markierte Variable oder Methode gehort zur ganzen Klasse und nicht zu einer einzelnen Instanz.

```
class test2 {
public:
  int a;
  static int b;
  int value() { return a + b;}
  void setValue(int v) { a=v; }
  static void setOffset(int o) { b=o;}
};

int test2_demo()
{
 test2::setOffset(12);
 test2 a;
 a.setValue(10);
 int b = a.value();
}
```

Eine static Methode kann nur auf die static Variablen und Methoden der Klasse zugreifen. Die nicht static Methoden können dagegen auf alle Variablen und Methoden der Klasse zugreifen. Dafür kann man static Variablen und Methoden auch aufrufen wenn man keine Instanz der Klasse zur Verfügung hat.

11.5. Konstruktoren und Destruktoren

Es gibt zwei Methoden mit ganz besonderen Aufgaben. Den **Konstruktor**, der automatisch beim Erzeugen einer Instanz aufgerufen wird. Z.B. um die Werte der Varibalen auf vernüftige Ausgangswerte zu setzen. Und den **Destruktor**, der bei löschen eine Instanz aufgerufen wird, z.B. um Dateien zu schließen oder aufzuräumen.
Der **Konstruktor** heißt immer genau wie die **Klasse** und hat **keinen Typ**, auch nicht **void**. Das Gleiche gilt auf für den **Destruktor**, aber seinem **Namen** ist ein ~ vorangestellt.

```
class test3 {
  int a;
public:
  test3() { a = 3 ; } // Konstruktor
  ~test3() // Destruktor
  {
        cout << "Hilfe ich werde geloescht!" << endl;
  }
};
```

11.6. Operator Overloading

Eine Besonderheit von C++ die man bei kaum einer anderen Computersprache findet, ist die Möglichkeit Operatoren zu überladen.
Einige der Operatoren in C++ sind :

```
Arithmetisch: +,-,*,/,%
Binär: &,|,<<,>>
Logisch: !,&&,||
Vergleichend: ==,!=,<=,>=
Zuweisend: =,+=,-=,/=,*=,++,--
Funktional: (),[],*,&,->
```

Das bekannteste Beispiel für das Überladen von Operatoren, ist die Verwendung der Bitschiebe-Operatoren « » als Ein/Ausgabe Operatoren mit *cin* und *cout* ,wofür sie wahrscheinlich häufiger eingesetzt werden als für ihre ursprüngliche Aufgabe.
Als erstes Beispiel, überladen wir « zur Ausgabe einer Note.

```
struct Note {
  string Fach;
  double Note;
};

ostream& operator<<(ostream &o,const Note& n)
{
 o << n.Fach << ':' << n.Note ;
}
```

Operatoren sind in C++ einfach Funktionen oder Methoden mit vorgegebenen Namen und Anzahl der Parameter. Der Name lautet operatorXX(..) wobei XX der zu überladende Operator ist. Die Typen der Parameter bestimmen, wann unser Spezialoperator zu Einsatz kommt.Besonders heufig überladene Operatoren sind ”+,-,[],(),<=,>=,==.“

11.7. Vererbung

Das Zusammenfassen von Daten und Funktionen zu einer Einheit , das automatische Einrichten und Aufräumen (Konstruktor/Destruktor) und die Möglichkeit Fremdzugriffe zu kontrollieren sind proktisch und nützlich. Aber der größte Vorteil der Objektorientierten Programmierung liegt in der Vererbung.

Anstatt ein Klasse komplett neu zu schreiben, kann man sie von einer bestehenden Klasse, die schon einen Teil der benötigten Funktion besitzt, ableiten. Die neue Klasse erbt das alle Eigenschaften ihrer Elternklasse, die dann beliebig angepasst oder erweitert werden können. Ein Klasse kann dabei theoretisch von beliebt vielen Klassen erben (Mehrfachvererbung) aber in der Praxis stammt eine Klasse nur sehr selten von mehr als 2 Klassen direkt ab.

```
class A {
  int a;
public:
  int value() { return a; }
  void setValue(int v) { a=v; }
};

class B : public A { // B wird von A abgeleitet
public:
  void clear() { setValue(0);}
}

class C {
public:
  void hello() {cout << "Hallo Welt\n";}
}

class D : public B,public C { // D wird von B und C abgeleitet
        // D besitzt alle Eigenschaften von A,B und C
}

void test()
```

```
{
 D a;
 a.clear();
 a.setValue(a.value()+8);
 a.hello();
}
```

Da bedeutet, daß komplizierte Dinge wie z.B. verkettete Listen , nur einmal geschrieben werden müßen und dann mit wenig Aufwand ein Klasse für die aktuelle Anwendung abgeleitet werden kann.Das bedeutet auch ,daß eine abgeleite Klasse immer Anstelle jeder ihrer Vorfarenklassen verwendet werden kann.

11.8. virtuelle Methoden

In unserem Beispiel könnte also z.B. A *b = &a; stehen. Daraus ergibt sich aber auch ein Problem. Betrachten wir das folgende Beispiel:

```
1   class A {
2   public:
3     void test() { cout << "Test" << endl; }
4     virtual void test2() { cout << "Test2" << endl; }
5
6  };
7   class B : public A {
8   public:
9     void test() { cout << "kein Test" << endl;}
10    virtual void test2() { cout << "Test4" << endl; }
11 };
12
13 int main()
14 {
15  B b1;
16  A a1;
17  A *a2 = &b1;
18
19  a1.test(); // Ruft test() in A auf.
20  b1.test(); // Ruft test() in B auf.
21  a2->test(); // was passiert hier?
22  a1.test2(); // Ruft test2() in A auf.
23  b1.test2(); // Ruft test2() in B auf.
24  a2->test2(); // was passiert hier?
25
26  return 0;
27 }
```

In Zeile 21 und 24 rufen wir über einen Pointer vom Typ A ein Instanz von B auf. In A und in B gibt es jeweils ein test() und test2() also welches wir ausgeführt?

- Im ersten Fall wird A::test() , also das test() aus der Klasse A aufgerufen.
- Im zweiten Fall wird B::test2() aufgerufen, den test2() ist als virtual markiert.

Wenn eine Methode in der Basisklasse[2] als virtual markiert ist, wird bei Aufruf immer die in der abgeleiteten Klasse liegende Method aufgerufen und nicht die der Basisklasse.

11.9. Beispiele

11.9.1. Eine Mehrzwegliste

[2] Einer Klasse von der unsere Klasse abstammt.

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

class Entry {
 public:
        Entry *next;
        virtual void show();
};

void Entry::show()
{
 cout << "Sorry, nix zu zeigen" << endl;
}

class List
{
Entry * first;
public:
 List();
 void append(Entry*e);
 void showAll();
};

List::List()
 :first(0)
{
}

void List::append(Entry *e)
{
 e->next=first;
 first = e;
}

void List::showAll()
{
 for(Entry *e=first;e;e=e->next) e->show();
}

// Abgeleitete Klassen

class IntEntry : public Entry {
 int value;
 public:
 IntEntry(int i);
 void show();
};

IntEntry::IntEntry(int i)
 :value(i)
{
}

void IntEntry::show()
{
 cout << value << endl;
}

```

```
class AddressEntry : public Entry {
 string name,strasse,ort,plz,tel;
 public:
 AddressEntry(string N,string S,string O,string PLZ,string T);
 void show();
};

AddressEntry::AddressEntry(string N,string S,string O,string PLZ,string T)
 :name(N),strasse(S),ort(O),plz(PLZ),tel(T)
{
}

void AddressEntry::show()
{
 cout << name << endl;
 cout << strasse << endl;
 cout << plz << " " << ort << endl;
 cout << tel << endl;
}

int main()
{
 List l1;
 l1.append(new IntEntry(1));
 for(int i=10;i<20;++i) l1.append(new IntEntry(i));
 List l2;
 l2.append(
  new AddressEntry("Arne Schmitt","Geranienstr.2",
        "Karlsruhe","76185","0721 9850228")
  );
 List l3;
 l3.append(new IntEntry(43));
 l3.append(
  new AddressEntry("Versuch Tester","Versuchsweg 123",
        "Testhausen","12345","0123 456789")
  );
 cout << "l1:" << endl;
 l1.showAll();
 cout << "l2:" << endl;
 l2.showAll();
 cout << "l3:" << endl;
 l3.showAll();
 return 0;
}
```

11.9.2. Notenbuch v6

```
/** Noten v6
 *  Mit Klassen
 *  Mit Dynamischer Speicherverwaltung,
 *  d.h. Es können beliebing viele Noten eingegeben werden **/

#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

// Die Datenstruktur Note, sammelt alle Informationen zu einer Note in einer Einheit
```

```
struct Note
{
        Note *next; // Zeiger auf die nächste Note in der Liste
        double Note;
        string Fach;
};

class NotenListe
{
 Note *Noten,*position;
 public:
 NotenListe()
 :Noten(0),position(0)
{
}

void removeAll()
{
        Note *p = Noten;
        Noten = 0;
        position =0;
        while(p)
        {
                Note *o = p;
                p = p->next;
                delete o;
        }
}
void Schreiben(char* filename)
{
        ofstream out(filename);
        out << "#Notenverwaltung_v5";
        for(Note *p=Noten;p;p = p->next)
                out << '\n' << p->Note << "\t" << p->Fach;
        out.close();
}
void Lesen(char* filename)
{
        ifstream in(filename);
        string s;
        in >> s;
        if(s!= "#Notenverwaltung_v5") {
                cout << "\nDateiformat nicht unterstützt!\n";
                return ;
        }
        removeAll();
        Noten = new Note();
        position = Noten;
        Note *op;
        while(!in.eof())
        {
                double note;
                string s;
                in >> note >>  s ;
                position->Note = note;
                position->Fach = s;
                position->next = new Note();
                op = position;
```

```
                        position = position->next;
                        position->next = 0;
                }
                op->next=0;
                delete position;
                position = op;
}

void einfuegen(Note *n)
{
                if(!Noten)
                {
                        Noten = n;
                        position = Noten;
                        position->next = 0;
                }
                else
                {
                        position->next = n;
                        position = position->next;
                        position->next = 0;
                }
}

void ausgeben()
{
                // Ausgeben
                for(Note *p=Noten;p;p = p->next)
                        cout << p->Fach << "\t" << p->Note << "\n";
}

double durchschnitt()
{
                // Durchschnitt
                double sum=0.0;
                int c =0;
                for(Note *p=Noten;p;p = p->next)
                {
                        sum += p->Note;
                        c++;
                }
                return sum/c;
}

};

// Damit unsere Funktionen die Noten sehen können, müssen wir sie hier oberhalb der F
void NoteEingeben(NotenListe &l)
{
                // Eingeben
                cout << "Bitte das Fach eingeben: ";
                string s;
                cin >> s;
                cout << "Bitte Note eingeben: ";
                double note;
                cin >> note;
                if(note > 6 || note < 1  )
                        cout << "Falsche Eingabe\n";
```

```
        else
        {
                Note *neu = new Note();
                neu->Note = note;
                neu->Fach = s;
                l.einfuegen(neu);
        }
}

void NotenAusgeben(NotenListe &l)
{
        // Ausgeben
        cout << "\n\tNoten:\n";
        l.ausgeben();
}

void DurchschnittAusgeben(NotenListe &l)
{
        cout << "\tDurchschnitt:" << l.durchschnitt() << endl;
}

int main(int argc, char *argv[])
{
  cout << "Notenverwaltung" << endl;
  bool beenden=false;
  NotenListe Noten;

  do // Hauptschleife
  {
          cout << "\n\tMenu\n";
          cout << "1: Eingeben\n";
          cout << "2: Ausgeben\n";
          cout << "3: Durchschnitt\n";
          cout << "8: Laden\n";
          cout << "9: Speichern\n";
          cout << "0: beenden\n";
          cout << ":";
          int eingabe;
          cin >> eingabe;
          double sum=0.0;
          switch(eingabe)
          {
                  case 1:
                          NoteEingeben(Noten);
                          break;
                  case 2:
                          NotenAusgeben(Noten);
                          break;
                  case 3:
                          DurchschnittAusgeben(Noten);
                          break;
                  case 8:
                          Noten.Lesen("notenbuch4");
                          break;
                  case 9:
                          Noten.Schreiben("notenbuch4");
                          break;
                  case 0:
```

```
                                beenden = true;
                                break;
                        default:
                                cout << "unbekannte Eingabe\n";
                }
        }
        while(!beenden);

        return EXIT_SUCCESS;
}
```

Teil V.
Die C++ Standartbibliothek (STL)

Die C++ Standartbibliothek (STL) enthält eine ganze Reihe von fertigen Lösungen für häufige Probleme.

12. STL Container

Als **Container** bezeichnet mal alle Datenstukturen die andere Daten beinhalten. Die **STL** bietet verschiedene **Container** für bestimmte Einsatzgebiete.
Die Container der **STL** sind sog. **Templates**, was das genau bedeutet ignorieren wir hier aber erstmal.

vector Die am heufigsten verwendete Variante, man kann sie als eine Art, in der Größe änderbares, Array betrachten.

list Ist eine doppelt verkettete Liste

queue Ein FIFO-Buffer.

deque Ist eine sogenannte "double-ended Queue"

stack Ein Stack ist ein sog. LIFO-Buffer.

map Ein Container der schnellen Zugriff auf einzellne Elemente über eine Schüssel ermöglicht.

set Ein Container, der von jedem Element immer nur max eines Enthält, man kann schnell abfragen ob eine bestimmtes Element enthalten ist.

bitset Ein Feld mit boolschen Werten (true/false)

valarray Ein Feld für Zahlen und Berechungen(z.b. Vektoren und Matrizen)

12.1. vector

Der **vector** ist ein guter allzweg Container. Wir behandeln ihn als Beispiel für den generellen Umgang mit den Containern der STL.
Ein **vector** ist praktisch ein **array** bei dem man einfach die Größe ändern, etwas anhängen oder entfernen kann.
Hier sind einige Beispiele:

```
#include <vector>
#include <iostream>

int main()
{
  // Legt einen vector mit 10 ints an
  vector<int> zahlen(10);
  // Legt einen vector mit 10 strings an, die auf "hallo" gesetzt sind
  vector<string> hallos(20,"hallo");
  // den 3. int in zahlen auf 5 setzen.
  zahlen[2]=5;
  // Gebe den string an der 3. Stelle von hallos aus
  cout << hallos[2] << endl;
  // eine andere Möglichkeit, auf bestimmte Elemente von zahlen zuzugreifen
  cout << zahlen.at(0) << ',' << zahlen.at(2) << endl;
  // Das erste Element von zahlen ausgeben
```

```
    cout << zahlen.front() << endl;
    // Das letzte Element von zahlen ausgeben
    cout << zahlen.back() << endl;
    // der vector zahlen wird auf 20 Elemente gesetzt die den wert 42 haben
    zahlen.assign(20,42);
    // hänge der 45 an den vector zahlen hinten an
    zahlen.push_back(45);

    cout << zahlen.back() << endl;
    // das letzte Element von zahlen entfernen
    zahlen.pop_back();
    cout << zahlen.back() << endl;
    // "Hallo Welt" in hallos an 4. Stelle einfügen, das bisherige 4. Element ist jetzt
    hallos.insert(3,"Hallo Welt");
    // Fügt 4 mal "Hallo Alle" an 7.Stelle in hallos ein.
    hallos.insert(6,4,"Hallo Alle");
    // Entfernt das Element an 5. Stelle aus hallos
    hallos.erase(5);
    // Gibt die Anzahl der Elemente in hallos aus
    cout << hallos.size() << endl;
    // Leert hallos;
    hallos.clear();
    // Gibt "Ja" aus, das hallos leer ist
    if(hallos.empty()) cout << "Ja" << endl;
    // Vergrößert hallos auf 100 Elemente.
    hallos.resize(100);
}
```

12.2. list

12.3. deque

12.4. map

12.5. set

12.6. stack

12.7. queue

12.8. valarray

12.9. Iteratoren

13. STL Algorithmen

13.1. for each

13.2. sort

A. Index

Index