

Einführung in C++

Arne Schmitt

16. September 2006

Inhaltsverzeichnis

I. Die erste Stunde	5
1. Variablen	6
1.1. Beispiele	7
2. Schleifen	7
2.1. Begriffserklärung	8
2.2. Die For-Schleife	8
2.3. Die While-Schleife	9
2.4. Die Do-While-Schleife	9
2.5. Beispiele	9
3. Verzweigungen	10
3.1. if	10
3.2. switch-case	11
4. Arrays	12
4.1. Beispiele	13
5. Pointer	13
5.1. Beispiele	13
6. Referenzen	14
6.1. Beispiele	14
II. Die ersten Schritte zum größeren Programm	15

7. Die Notenverwaltung v1	15
7.1. Anforderungen	15
7.2. Der Entwurf - Analyse	15
7.3. Der Quelltext	17
7.4. Erweiterungen	18
III. Strukturen	20
8. Funktionen	20
8.1. Die Notenverwaltung v2	23
9. Dateien	25
9.1. Die Notenverwaltung v3	26
10. Datenstrukturen	26
10.1. struct	26
10.2. Dynamische Speicherverwaltung: new / delete	27
10.3. Verkettete Liste	28
10.3.1. Die einfach verkettete Liste	28
10.3.2. Die doppelt verkettete Liste	30
10.4. Binäre Bäume	33
10.4.1. Zeichenketten sortieren 1	33
IV. Die Grundlagen der Objekt Orientierten Programmierung (OOP)	36
11. Objekte	36
11.1. Klassen	36
11.2. Kapselung und Zugriffsstufen	36
11.3. Klasse vs. Instanz	37
11.4. static Members	37
11.5. Konstruktoren und Destruktoren	38
11.6. Operator Overloading	38
11.7. Vererbung	39
11.8. virtuelle Methoden	40
11.9. Beispiele	41
11.9.1. Eine Mehrwegliste	41
11.9.2. Notenbuch v6	43
V. Die C++ Standardbibliothek (STL)	49

12. STL Container	49
12.1. vector	49
12.2. list	51
12.3. deque	51
12.4. map	51
12.5. set	51
12.6. stack	51
12.7. queue	51
12.8. valarray	51
12.9. Iteratoren	51
13. STL Algorithmen	51
13.1. for each	51
13.2. sort	51
VI. Hinter den Kulissen	52
14. Compiler	52
14.1. Precompiler	52
14.2. Compiler	52
14.3. Linker	52
15. Bibliotheken	52
15.1. Header	52
15.2. Library	52
16. Projekte aus mehr als einer Datei	52
16.1. Per Hand	52
16.2. make und Makefiles	52
16.3. Autotools	52
16.4. qmake	52
17. Projektmanagement	52
17.1. Codeverteilung	52
17.2. Versionskontrolle	52
17.3. Eigene Bibliotheken	52
VII. Maus, Fenster und Co	53
18. Events und Mainloop	53
19. GUI - Graphical User Interface	53

20. Die Qt-Bibliothek	53
20.1. Graphisches Hello World	53

Teil I.

Die erste Stunde

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

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main(int argc, char *argv[])
7 {
8     cout << "Hello ,_world!" << endl;
9     return EXIT_SUCCESS;
10 }
```

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 erklärt dem Compiler welche Bibliotheken wir verwenden wollen.

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

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

```
1 int main(int argc, char *argv[])
2 {
3     ...
4     return EXIT_SUCCESS;
5 }
```

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:

```
1 cout << "Hello ,_world!" << endl;
```

Wie wir bereits wissen gibt er dem Text „Hello, world!“ auf dem Bildschirm aus. Texte werden in C++ immer 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:

```
1 cout << Wert;
```

oder wenn wir mehrere Werte ausgeben wollen:

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

Werte können dabei nicht nur Texte sein, sondern genauso gut auch Zahlen(int,double), einzelnen 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**.

```
1 cout << "Anzahl: ";
2 int z;
3 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 *Deklariieren*, also dem Compiler sagen was für einen **Typ** von **Variable** wir wollen und wie sie heißen soll. Typ Name; Dazu ein paar Beispiele:

```
1 bool ende;
2 double preis;
3 char buchstabe;
4 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:

```
1 ende=true;
2 preis=42.6;
3 buchstabe='P';
4 zahl=-5;
```

1.1. Beispiele

Jetzt sind wir soweit ein paar einfache Programme zu schreiben.

1.Kreisumfang berechnen

```
1 double r;
2 cout << "Radius: " ;
3 cin >> r;
4 cout << "Umfang: " << r*3.1415 << endl;
```

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

```
1 int jahre;
2 cout << "Jahre: ";
3 cin >> jahre;
4 cout >> "in_" << jahre << "_Jahren_sind_ca_" \
5         << jahre *365.25 << "_Tage_vergangen" << endl;
6 cout >> "in_" << jahre << "_Jahren_sind_ca_" \
7         << jahre *365.25 * 24 << "_Stunden_vergangen" << endl;
8 cout >> "in_" << jahre << "_Jahren_sind_ca_" \
9         << 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 Zweck 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 entweder ein einzelner Befehl oder beliebig viele Befehle in geschweiften Klammern.

```
1 // BLOCK1
2 cout << "Ich_bin_ein_BLOCK" << endl;
3 // oder BLOCK2
4 {
5 cout << "Ich_bin_auch_";
6 cout << "ein_Block";
7 cout << endl;
8 cout << "Aber_auf_mehr_als_einem_Befehl!";
9 cout << endl;
10 }
```

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

```
1 // Bedingung1
2 2==4; // false
3 2==2; // true
4 true; // true
5 2>-4; // true
6 0; // false
7 ! 0; // true
8
9 int a=12,c=4;
10 int *b= &a;
11 a; // true , da !=0
12 b; // true , Zeigt auf a;
13 a==b; // höchstwahrscheinlich false
14 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
```



```

1  for(int i=0;i<10;i++)
2  {
3    cout << "Durchlauf_Nr" << i+1 << endl;
4  }

```

2.3. Die While-Schleife

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

while(BEDINGUNG) BLOCK

```

1  int i=0;
2  while(i<10)
3  {
4    cout << ":";
5    cin >> i;
6  }
7  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)

```

1  int i=0;
2  do
3  {
4    cout << ":";
5    cin >> i;
6  } while(i<10)
7  cout << "Fertig" << endl;

```

2.5. Beispiele

1.Zählen von 1 bis z

```

1  int z;
2  cin >> z;
3  for(int i=0;i<z;i++)
4  {
5    cout << i+1 << ", ";
6  }
7  cout << endl;

```

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

```
1  int z;
2
3  cout << "Stellenzählen" << endl;
4  cout << "Zahl:";
5  cin >> z;
6
7  int stellen=0;
8  while(z!=0)
9  {
10     stellen++;
11     z = z /10; // Zahl um eine Stelle verschieben
12 }
13 cout << "Es_sind_" << stellen << "_Stellen" << endl;
```

3.Kommazahl aufsummieren bis der Benutzer 0 eingibt

```
1  cout << "Summe" << endl;
2  double summe=0.0, eingabe;
3  do
4  {
5     cin >> eingabe;
6     summe += eingabe;
7  }
8  while(eingabe!=0);
9
10 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:

```
1  if( x == 1)
2  {
3     cout << "X_ist_eins\n";
4  }
```

```

5 else
6 {
7   cout << "X_ist_nicht_eins.\n";
8 }
9
10 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 Ende des *switch-Blocks* stößt. Wenn keiner der *Cases* zutrifft, werden die Befehle hinter **default:** ausgeführt.

Beispiel:

```

1 int x;
2 cin >> x;
3 Switch(x)
4 {
5 case 1:
6     cout << "Eine_Möglichkeit";
7     break;
8 case 2:
9     cout << "Eine_Möglichkeit";

```

```

10     break;
11 case 2:
12     cout << "Eine_Möglichkeit ";
13     break;
14 case 8:
15     cout << "Eine_Möglichkeit ";
16     break;
17 default:
18     cout << "Irgendwas_anderes\n";
19 }

```

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?

```

1  int x;
2  for (int i=0;i<10;i++)
3  {
4  cout << "Bitte_geben_sie_eine_Zahl_ein: ";
5  cin >> x;
6  }

```

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:

```

1  int x[10];
2  for (int i=0;i<10;i++)
3  {
4  cout << "Bitte_geben_sie_eine_Zahl_ein: ";
5  cin >> x[i];
6  }

```

4.1. Beispiele

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

5. Pointer

Pointer(Zeiger) enthalten die Speicheradresse einer Information.

Typ¹ *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 Adresse 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 Adresse 1234
//Etwas ,daß man nie tun sollte! es sei den man weiß ganz genau was man da macht.
```

5.1. Beispiele

```
1 int zahl=2;
2 int zahl2= 15;
3 int zahlen[5]={3,5,2,4,0};
4 int *zeiger=&zahl;
5 cout << zeiger << "_" << *zeiger << "\n";
6 *zeiger = 5;
7 cout << zeiger << "_" << *zeiger << "\n";
8 zeiger = &zahl2;
9 cout << zeiger << "_" << *zeiger << "\n";
10 zeiger = zahlen;
11 while(*zeiger)
12 {
13     cout << zeiger << "_" << *zeiger << "\n";
14     zeiger++;
15 }
```

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

6. Referenzen

Während **Zeiger** schon zum Sprachumfang von C gehörten, sind **Referenzen** erst mit C++ hinzugekommen. **Referenzen** sind im Grunde *versteckte Pointer*, also Pointer deren Wert man nicht auslesen oder verändern kann. Das bedeutet auch ,daß man eine Referenz nicht *dereferenzieren* muß um den Wert der eigentlichen Variable zu erhalten. Sie sind also hauptsächlich 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 !

6.1. Beispiele

```
1 int zahl=2;
2 int &ref = zahl;
3 cout << ref << ' ' << zahl << "\n";
4 ref = 4;
5 cout << ref << ' ' << zahl << "\n";
6 zahl++;
7 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 beherrschen:

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 scheidet 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 gespeicherte Note ja jedesmal ersetzen. Da wir die Noten in einem `Area` speichern, kann man die position als ganze Zahl angeben, also als `int`.

```

1 double Noten[10];
2 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 präsentiert dem Nutzer die Liste der möglichen Funktionen und läßt ihn auswählen.

```

1 cout << "\n\tMenu\n";
2 cout << "1: \_Eingeben\n";
3 cout << "2: \_Ausgeben\n";
4 cout << "3: \_Durchschnitt\n";
5 cout << "0: \_Beenden\n";
6 cout << ":\n";
7 int eingabe;
8 cin >> eingabe;
9 switch(eingabe)
10 {
11     ...
12 }

```

Und jetzt zu den einzelnen Funktionen:

Eingabe einer Note

Als erstes sollten wir den Nutzer darüber informieren was wir von ihm erwarten.

```

1 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.

```

1 double note;
2 cin >> note;
3 if(position >=10) break; // Leider kein Platz mehr
4 if(note > 6 || note < 1 )
5     cout << "Falsche_Eingabe\n";
6 else
7 {
8     Noten[position] = note;
9     position += 1;
10 }

```

Ausgabe der Noten

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

```

1 cout << "\n\tNoten:\n";
2 for(int i=0; i<position; i++)

```



```
3 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++:

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

7.3. Der Quelltext

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

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 int main(int argc , char *argv [])
7 {
8     cout << "Notenverwaltung" << endl;
9     bool beenden=false;
10    double Noten[10];
11    int position=0;
12
13    do
14    {
15        cout << "\n\tMenu\n";
16        cout << "1:_Eingeben\n";
17        cout << "2:_Ausgeben\n";
18        cout << "3:_Durchschnitt\n";
19        cout << "0:_Beenden\n";
20        cout << ":\n";
21        int eingabe;
22        cin >> eingabe;
23        double sum=0.0;
24        switch(eingabe)
```

```

25     {
26         case 1:
27             // Eingeben
28             cout << "Bitte_Note_eingeben:";
29             double note;
30             cin >> note;
31             if(position >= 10) break;
32             if(note > 6 || note < 1 )
33                 cout << "Falsche_Eingabe\n";
34             else
35             {
36                 Noten[position] = note;
37                 position += 1;
38             }
39             break;
40         case 2:
41             // Ausgeben
42             cout << "\n\tNoten:\n";
43             for(int i=0; i<position; i++)
44                 cout << i+1 << ":_ " << Noten[i] << "\n";
45             break;
46         case 3:
47             // Durchschnitt
48
49             for(int i=0; i<position; i++)
50                 sum += Noten[i];
51             cout << "\tDurchschnitt:" << sum / position << endl;
52             break;
53         case 0:
54             beenden = true;
55             break;
56         default:
57             cout << "unbekannte_Eingabe\n";
58     }
59 }
60 while (!beenden);
61
62 return EXIT_SUCCESS;
63 }

```

7.4. Erweiterungen

Zum üben bieten sich jetzt einige Erweiterungen an.

1. Weitere Statistikfunktionen (z.B. Standardabweichung) 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:

```
1 int main ()
2 {
3   for (int i=0;i<40;++i) cout << '-';
4   cout << endl;
5   cout << "Text1" << endl;
6   for (int i=0;i<40;++i) cout << '-';
7   cout << endl;
8   for (int i=0;i<40;++i) cout << '-';
9   cout << endl;
10  cout << "Text2" << endl;
11  for (int i=0;i<40;++i) cout << '-';
12  cout << endl;
13  cout << "Text3" << endl;
14  for (int i=0;i<40;++i) cout << '-';
15  cout << endl;
16  for (int i=0;i<40;++i) cout << '-';
17  cout << endl;
18  cout << "Text4" << endl;
19  for (int i=0;i<40;++i) cout << '-';
20  cout << endl;
21 }
```

Sieht recht aufwendig und unübersichtlich aus. Wenn wir genauer hinsehen, erkennen wir, daß sich eine Menge identisch wiederholt. Jetzt lagern wir den sich wiederholenden Teil aus und geben ihm einen Namen.

Damit sieht unser Beispiel jetzt so aus:

```
1 void linie ()
2 {
3   for (int i=0;i<40;++i) cout << '-';
4   cout << endl;
```

```

5 }
6
7 int main()
8 {
9
10  linie ();
11  cout << "Text2" << endl;
12  linie ();
13  linie ();
14  cout << "Text3" << endl;
15  linie ();
16  linie ();
17  cout << "Text4" << endl;
18  linie ();
19 }

```

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.

```

1 void linie ()
2 {
3   for (int i=0;i<40;++i) cout << '-';
4   cout << endl;
5 }
6
7 void rahmen(string s)
8 {
9   linie ();
10  cout << s << endl;
11  linie ();
12 }
13
14 int main ()
15 {
16  rahmen("text1");
17  rahmen("text2");
18  rahmen("text3");
19 }

```

Jetzt liet sich das Hauptprogramm schon sehr viel Besser und wir muten auch nicht so viel schreiben. Auerdem knnen wir das Aussehen des Rahmens nun ganz einfach ndern ohne darauf achten zu mssen wo er im Programm berall verwendet wird. z.B.:

```

1 void linie ()
2 {

```

```

3  for(int i=0;i<40;++i) cout << '*'; // Stern statt -
4  cout << endl;
5  }
6
7  void rahmen(string s)
8  {
9  linie ();
10 cout << '*' << s;
11 for (int i=0;i<38-s.length();++i) cout << '_';
12 cout << '*' << endl;
13 linie ();
14 }
15
16 int main ()
17 {
18 rahmen ("text1");
19 rahmen ("text2");
20 rahmen ("text3");
21 }

```

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 trotzdem 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 *Speicheradresse* der **Funktion** erfahren wollen.

Eine Funktion findet man in jedem Programm:

Die Main-Funktion, die ungefähr so aussieht.

```

1  int main(int argc , char *argv [])
2  {
3      ...
4  return EXIT_SUCCESS; // Alles in Ordnung
5  }

```

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

```
1 double durchschnitt(double Noten[],int position)
2 {
3     double sum=0.0;
4     for(int i=0; i<position;i++)
5         sum += Noten[i];
6     return sum / position;
7 }
8
9 int main()
10 {
11     double Noten[]={1,2,4,2,2,1,3};
12     cout << "Durchschnitt:" << durchschnitt(Noten,6) << endl;
13 }
```

8.1. Die Notenverwaltung v2

Hier ist unsere Notenverwaltung jetzt auf mehrere Funktionen aufgeteilt.

```
1 #include <iostream>
2 #include <cstdlib>
3
4 using namespace std;
5
6 // Damit unsere Funktionen die Noten sehen können,
7 // müssen wir sie hier oberhalb der Funktionen deklarieren
8 double Noten[10];
9 int position=0;
10
11 void NoteEingeben()
12 {
13     // Eingeben
14     cout << "Bitte_Note_eingeben:";
15     double note;
16     cin >> note;
17     if(position >=10)
18     {
19         cout << "Kein_Platz_mehr\n";
20         return;
21     }
22     if(note > 6 || note < 1 )
23         cout << "Falsche_Eingabe\n";
24     else
```

```

25     {
26         Noten[position] = note;
27         position += 1;
28     }
29 }
30
31 void NotenAusgeben()
32 {
33     // Ausgeben
34     cout << "\n\tNoten:\n";
35     for(int i=0; i<position; i++)
36         cout << i+1 << ":_ " << Noten[i] << "\n";
37 }
38
39 double durchschnitt()
40 {
41     // Durchschnitt
42     double sum=0.0;
43     for(int i=0; i<position; i++)
44         sum += Noten[i];
45     return sum/position;
46 }
47
48 void DurchschnittAusgeben()
49 {
50     cout << "\tDurchschnitt:" << durchschnitt() << endl;
51 }
52
53 int main(int argc, char *argv[])
54 {
55     cout << "Notenverwaltung" << endl;
56     bool beenden=false;
57
58     do // Hauptschleife
59     {
60         cout << "\n\tMenu\n";
61         cout << "1:_Eingeben\n";
62         cout << "2:_Ausgeben\n";
63         cout << "3:_Durchschnitt\n";
64         cout << "0:_beenden\n";
65         cout << ":\n";
66         int eingabe;
67         cin >> eingabe;
68         double sum=0.0;
69         switch(eingabe)

```



```

70     {
71         case 1:
72             NoteEingeben();
73             break;
74         case 2:
75             NotenAusgeben();
76             break;
77         case 3:
78             DurchschnittAusgeben();
79             break;
80         case 0:
81             beenden = true;
82             break;
83         default:
84             cout << "unbekannte_Eingabe\n";
85     }
86 }
87 while (!beenden);
88
89 return EXIT_SUCCESS;
90 }

```

9. Dateien

Die Lösung für das Problem ,daß Informationen mit den des Progamms verloren sind, heißt Dateien. In C++ werden Dateien meist als streams behandelt. Also also Datenströme aus denen man nacheinander Daten lesen oder in die man nacheinander Daten schreiben kann.

Mit zwei Streams hatten wir bereits zutun:

cin Der Inputstreaan. Meist die Tastatureingaben des Benutzers.

cout Der Outputstream. Der Inhalt erscheint normalerweise auf dem Bildschirm.

Wir können nun 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.

```
1 #include <fstream>
```

jetzt können wir ein Datei zum Schreiben

```
1 ofstream output("Testdatei");
2 output << "Das_hier_steht_gleich_auf_der_Festplatte" << endl;
3 output.close();
```

oder auch zum lesen öffnen.

```

1 ifstream input("Testdatei");
2 string s;
3 input >> s;
4 input.close();
5 cout << s;

```

9.1. Die Notenverwaltung v3

Damit bekommt unsere Notenverwaltung ein Gedächtnis:

```

1 void Schreiben(char* filename)
2 {
3     ofstream out(filename);
4     out << "#Notenverwaltung_v3\n";
5     for(int i=0;i<position;i++)
6         out << Noten[i] << "\n";
7     out.close();
8 }
9
10 void Lesen(char* filename)
11 {
12     ifstream in(filename);
13     string s;
14     in >> s;
15     position =0;
16     while(!in.eof() && position<10)
17     {
18         double note;
19         in >> note;
20         Noten[position] = note;
21         position++;
22     }
23 }

```

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) fasst zusammengehörende Daten zusammen.

```

1 struct _notenbuch

```

```

2 {
3   double Noten[10];
4   int position;
5 };
6
7 _notenbuch buch;
8 buch.position=0;
9 buch.Noten[2]=1.5;

```

10.2. Dynamische Speicherverwaltung: new / delete

Im folgenden werden wir häufiger **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 compilieren bekannt und der konnte schon vor dem Start des Programms genau festlegen was wo im Speicher liegt.

Variablen die wir über *dynamische Speicherverwaltung* erhalten, haben keine Namen. Wir bekommen nur die Zeiger auf ihre Speicherblöcke. Um dynamischen Speicher anzufordern verwendet man den Operator **new** und um ihn freizugeben **delete**;

Beispiele für new und delete

```

1 int main()
2 {
3   int *a = new int; // a zeigt auf speicherplatz für einen int
4   int *ar = new int[20]; // ar zeigt auf speicherplatz fuer 20 int;
5
6   *a=12;
7   ar[2] = 12;
8   ar[0] = 2;
9   cout << "ar[0]=" << ar[0] << endl;
10  cout << "*array=" << *ar << endl;
11  cout << "ar[2]=" << ar[2] << endl;
12  cout << "*(ar+2)=" << *(ar+2) << endl;
13
14  cout << a << "_zeigt_auf_" << *a << endl;
15  delete a;
16  a = new int;
17  cout << a << "_zeigt_auf_" << *a << endl;
18
19  /* Wenn wir den Speicher ,d.h. die Variablen, nicht mehr brauchen
20  sollten wir ihn freigeben */
21  delete a;
22  delete [] ar;
23  /* Wenn man einen Dynamischen Wert mit delete freigeben hat,

```

```

24 sollte man immer alle Pointer die darauf zeigen auf 0 setzten */
25 a=0;
26 ar=0;
27 }

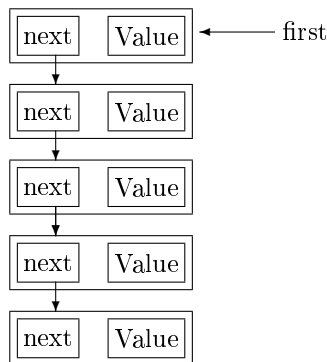
```

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 Verkettete 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

```

1 /******
2 ** Beispiel fuer die Umsetzung einer einfach verketteten Liste *
3 *****
4
5 /* Unsere Datenstruktur */
6 struct Entry
7 {
8     Entry *next; // Zeiger auf den naechsten Eintrag;
9     int Value; // Beispielwert
10 };
11
12 Entry *first=0; // Zeiger auf den ersten Eintrag;
13
14 /* Neuen Wert am Anfang einfuegen */

```

```

15 Entry *insertValueFirst(int v)
16 {
17     Entry *neu = new Entry();
18     neu->Value=v;
19     neu->next=first;
20     first = neu;
21 }
22
23 /* Neuen Wert am Ende anhaengen */
24 Entry *appendValue(int v)
25 {
26     Entry *neu = new Entry();
27     neu->Value = v;
28     neu->next = 0;
29     if(first)
30     {
31         Entry *i=0;
32         for(i=first;i->next; i=i->next) ;
33         i->next = neu;
34     }
35     else
36         first = neu;
37     return neu;
38 }
39
40 /* Eintrag mit bestimmten Wert finden */
41 Entry *findValue(int v)
42 {
43     for(Entry *i=first;i;i=i->next)
44     {
45         if(i->value==v) return i;
46     }
47 }
48
49 /* Eintrag entfernen */
50 void removeEntry(Entry *e)
51 {
52     if(e==first) first = e->next;
53     else
54     {
55         Entry *i=0;
56         for(i=first;i && i->next!=e;i=i->next) ;
57         if(!i) return; else i->next = e->next;
58     }
59     delete e;

```

60 }

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.

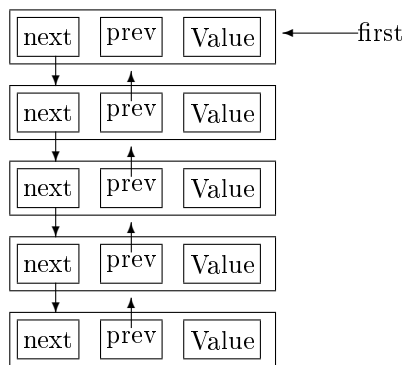
Nachteile der einfach verketteten Liste:

- 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 aufwändig den Vorgänger eines Eintrags zu finden.
- es ist teuer einen Wert am Ende anzuhängen.
- es ist teuer und aufwändig einen Wert an beliebiger Stelle einzufügen.
- es ist teuer und aufwändig einen Wert an beliebiger Stelle zu entfernen.
- es ist teuer und aufwändig einen Wert an beliebiger Stelle zu verschieben.

Man kann das Einfügen und Auslesen am Ende stark vereinfachen und beschleunigen in dem man zusätzlich 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 doppelt verketteten Liste

```
1  /*****
2  ** Beispiel fuer die Umsetzung einer doppelt verketteten Liste *
3  *****/
4
5  /* Unsere Datenstruktur */
6  struct Entry
7  {
8      Entry *next; // Zeiger auf den naechsten Eintrag;
9      Entry *prev; // Zeiger auf den vorherigen Eintrag;
10     int Value; // Beispielwert
11 };
12
13 Entry *first=0; // Zeiger auf den ersten Eintrag;
14
15 /* Neuen Wert am Anfang einfuegen */
16 Entry *insertValueFirst(int v)
17 {
18     Entry *neu = new Entry ();
19     neu->Value=v;
20     neu->next=first;
21     neu->prev=0;
22     first = neu;
23 }
24
25 /* Neuen Wert am Ende anhaengen */
26 Entry *appendValue(int v)
27 {
28     Entry *neu = new Entry ();
29     neu->Value = v;
30     neu->next = 0;
31     neu->prev = 0;
```

```

32  if(first)
33  {
34      Entry *i=0;
35      for(i=first;i->next; i=i->next) ;
36      i->next = neu;
37      neu->prev = i;
38  }
39  else
40      first = neu;
41  return neu;
42  }
43
44  /* Eintrag mit bestimmten Wert finden */
45  Entry *findValue(int v)
46  {
47      for(Entry *i=first;i;i=i->next)
48      {
49          if(i->value==v) return i;
50      }
51  }
52
53  /* Eintrag entfernen */
54  void removeEntry(Entry *e)
55  {
56      if(e==first)
57      {
58          first = e->next;
59          first->prev = 0;
60      }
61      else
62      {
63          Entry *n = e->next;
64          Entry *p = e->prev;
65          p->next = n;
66          n->prev = p;
67      }
68      delete e;
69  }

```

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 Strukturen. 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 gesamten 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 im mittel deutlich schneller ist als bei einer verketteten Liste.

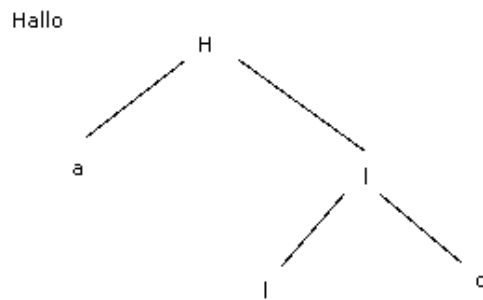
Da wir im Moment noch nicht so weit sind den Computer koplizierter Grafiken anzeigen zu lassen, behandeln wir binäre Bäume zunächst auf den Papier, da man sonst die Struktur des Baumes nicht erkennen kann.

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.



```

1  struct Knoten {
2  char Wert;
3  Knoten *links;
4  Knoten *rechts;
5  };
6
7  Knoten wurzel=0;
8
9  /* füge einen neuen Wert n in den Baum ein */
10 void insert(char n)
11 {
12   Knoten *neu = new Knoten;
13   neu->Wert = n;
14   neu->rechts = 0;
15   neu->links = 0;
16   if(wurzel==0)
17   {
18     wurzel = neu;
19   }
20   else
21   {
22     Knoten *i = wurzel;
23     while(i)
24     {
25       if(i->Wert<n && i->rechts!=0) i=rechts;
26       if(i->Wert>=n && i->links!=0) i=links;
27       if(i->Wert<n && i->rechts==0) { rechts = neu; i=0; }
28       if(i->Wert>=n && i->links=0) { links = neu; i=0; }
29     }
30   }
31 }
32
33 /* Finde einen Knoten mit Wert==w unterhalb von Knoten r */
34 Knoten *findeKnoten(Knoten *r, char w)

```

```

35 {
36     if (r==0) return 0;
37     if (r->Wert == w ) return r;
38     if (r->Wert < w ) return findeKnoten(r->rechts ,w);
39     else return findeKnoten(r->links ,w);
40 }
41
42 /* alle Knoten von links nach rechts ausgeben */
43 void ListKnoten (Knoten *r)
44 {
45     if (r==0) return;
46     cout << r->Wert << endl;
47     if (r->links ) ListKnoten (r->links );
48     if (r->rechts ) ListKnoten (r->rechts );
49 }

```

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 Programmierung (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 Zugriffsstufen, Vererbung und Konstruktoren/Destruktoren.

```
1  class Klassenname : public Elternklasse {
2      Typ1 private_Variable;
3      static Typ5 klassen_Variable;
4      ...
5  public:
6      Typ2 oeffentliche_Variable;
7      ...
8      void oeffentliche_funktion ();
9      virtual void oeffentliche_funktion2 ();
10     static void klassen_funktion ();
11     ....
12 protected:
13     Typ3 vererbbaere_variable;
14     Typ3 vererbbaere_funktion(Typ4 t);
15 };
```

Es folgen die Erklärungen der einzelnen Komponenten:

11.2. Kapselung und Zugriffsstufen

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

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.

```

1  class test {
2    int a ; // private , da nichts anderes angegeben
3  public:
4    int b; // public;
5    int c; // public;
6  protected:
7    int d; // protected;
8  private:
9    int e; // private , diesmal explizit angegeben
10 };

```

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 beschreiben 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 sich auch in der Deklaration einer Klasse. Und zwar durch das Schlüsselwort `static`. Eine mit `static` markierte Variable oder Methode gehört zur ganzen Klasse und nicht zu einer einzelnen Instanz.

```

1  class test2 {
2  public:
3    int a;
4    static int b;
5    int value() { return a + b;}
6    void setValue(int v) { a=v; }
7    static void setOffset(int o) { b=o;}
8  };
9
10 int test2_demo()
11 {
12   test2::setOffset(12);
13   test2 a;
14   a.setValue(10);
15   int b = a.value();
16 }

```

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 Variablen auf vernünftige 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.

```

1  class test3 {
2      int a;
3      public:
4          test3() { a = 3 ; } // Konstruktor
5          ~test3() // Destruktor
6          {
7              cout << "Hilfe_ich_werde_geloescht!" << endl;
8          }
9  };

```

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.

```

1  struct Note {
2      string Fach;
3      double Note;

```

```

4   };
5
6   ostream& operator<<(ostream &o, const Note& n)
7   {
8     o << n.Fach << ' : ' << n.Note ;
9   }

```

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 praktisch und nützlich. Aber der größte Vorteil der objektorientierten Programmierung liegt in der Vererbung.

Anstatt eine 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. Eine Klasse kann dabei theoretisch von beliebig vielen Klassen erben (Mehrfachvererbung) aber in der Praxis stammt eine Klasse nur sehr selten von mehr als 2 Klassen direkt ab.

```

1   class A {
2     int a;
3   public:
4     int value() { return a; }
5     void setValue(int v) { a=v; }
6   };
7
8   class B : public A { // B wird von A abgeleitet
9   public:
10    void clear() { setValue(0); }
11  }
12
13  class C {
14  public:
15    void hello() { cout << "Hallo_Welt\n"; }
16  }
17
18  class D : public B, public C { // D wird von B und C abgeleitet
19    // D besitzt alle Eigenschaften von A, B und C
20  }
21

```

```

22 void test ()
23 {
24     D a;
25     a.clear ();
26     a.setValue (a.value ()+8);
27     a.hello ();
28 }

```

Das bedeutet, daß komplizierte Dinge wie z.B. verkettete Listen, nur einmal geschrieben werden müssen und dann mit wenig Aufwand eine Klasse für die aktuelle Anwendung abgeleitet werden kann. Das bedeutet auch, daß eine abgeleitete Klasse immer Anstelle jeder ihrer Vorfahrenklassen 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 wird 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 Basisklasse² 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 Mehrwegliste

```
1 #include <iostream>
2 using namespace std;
3
4 class Entry {
5     public:
6         Entry *next;
7         virtual void show ();
8 };
9
10 void Entry::show ()
11 {
12     cout << "Sorry, _nix_zu_zeigen" << endl;
13 }
14
15 class List
16 {
17     Entry * first;
18     public:
19     List ();
20     void append(Entry*e);
21     void showAll ();
22 };
23
24 List::List ()
25     : first (0)
26 {
27 }
28
29 void List::append(Entry *e)
30 {
```

²Einer Klasse von der unsere Klasse abstammt.

```

31  e->next=first;
32  first = e;
33  }
34
35  void List::showAll()
36  {
37  for(Entry *e=first;e;e=e->next) e->show();
38  }
39
40  // Abgeleitete Klassen
41
42  class IntEntry : public Entry {
43  int value;
44  public:
45  IntEntry(int i);
46  void show();
47  };
48
49  IntEntry::IntEntry(int i)
50  : value(i)
51  {
52  }
53
54  void IntEntry::show()
55  {
56  cout << value << endl;
57  }
58
59  class AddressEntry : public Entry {
60  string name, strasse, ort, plz, tel;
61  public:
62  AddressEntry(string N, string S, string O, string PLZ, string T);
63  void show();
64  };
65
66  AddressEntry::AddressEntry(string N, string S, string O, string PLZ, string T)
67  : name(N), strasse(S), ort(O), plz(PLZ), tel(T)
68  {
69  }
70
71  void AddressEntry::show()
72  {
73  cout << name << endl;
74  cout << strasse << endl;
75  cout << plz << " " << ort << endl;

```

```

76 cout << tel << endl;
77 }
78
79 int main()
80 {
81     List l1;
82     l1.append(new IntEntry(1));
83     for(int i=10;i<20;++i) l1.append(new IntEntry(i));
84     List l2;
85     l2.append(
86         new AddressEntry("Arne_Schmitt","Geranienstr.2",
87             "Karlsruhe","76185","0721_9850228")
88     );
89     List l3;
90     l3.append(new IntEntry(43));
91     l3.append(
92         new AddressEntry("Versuch_Tester","Versuchsweg_123",
93             "Testhausen","12345","0123_456789")
94     );
95     cout << "l1:" << endl;
96     l1.showAll();
97     cout << "l2:" << endl;
98     l2.showAll();
99     cout << "l3:" << endl;
100    l3.showAll();
101    return 0;
102 }

```

11.9.2. Notenbuch v6

```

1  /** Noten v6
2  *   Mit Klassen
3  *   Mit Dynamischer Speicherverwaltung,
4  *   d.h. Es können beliebig viele Noten eingegeben werden **/
5
6  #include <iostream>
7  #include <fstream>
8  #include <cstdlib>
9
10 using namespace std;
11
12 // Die Datenstruktur Note, sammelt alle Informationen zu einer Note in einer Einl
13 struct Note
14 {
15     Note *next; // Zeiger auf die nächste Note in der Liste

```

```

16         double Note;
17         string Fach;
18     };
19
20     class NotenListe
21     {
22     Note *Noten,* position;
23     public:
24     NotenListe()
25     :Noten(0), position(0)
26     {
27     }
28
29     void removeAll()
30     {
31         Note *p = Noten;
32         Noten = 0;
33         position =0;
34         while(p)
35         {
36             Note *o = p;
37             p = p->next;
38             delete o;
39         }
40     }
41     void Schreiben(char* filename)
42     {
43         ofstream out(filename);
44         out << "#Notenverwaltung_v5";
45         for(Note *p=Noten;p;p = p->next)
46             out << '\n' << p->Note << "\t" << p->Fach;
47         out.close();
48     }
49     void Lesen(char* filename)
50     {
51         ifstream in(filename);
52         string s;
53         in >> s;
54         if(s!= "#Notenverwaltung_v5") {
55             cout << "\nDateiformat nicht unterstützt!\n";
56             return ;
57         }
58         removeAll();
59         Noten = new Note();
60         position = Noten;

```

```

61     Note *op;
62     while (!in.eof())
63     {
64         double note;
65         string s;
66         in >> note >> s ;
67         position->Note = note;
68         position->Fach = s;
69         position->next = new Note();
70         op = position;
71         position = position->next;
72         position->next = 0;
73     }
74     op->next=0;
75     delete position;
76     position = op;
77 }
78
79 void einfuegen(Note *n)
80 {
81     if (!Noten)
82     {
83         Noten = n;
84         position = Noten;
85         position->next = 0;
86     }
87     else
88     {
89         position->next = n;
90         position = position->next;
91         position->next = 0;
92     }
93 }
94
95 void ausgeben()
96 {
97     // Ausgeben
98     for (Note *p=Noten;p;p = p->next)
99         cout << p->Fach << "\t" << p->Note << "\n";
100 }
101
102 double durchschnitt()
103 {
104     // Durchschnitt
105     double sum=0.0;

```

```

106     int c =0;
107     for (Note *p=Noten;p;p = p->next)
108     {
109         sum += p->Note;
110         c++;
111     }
112     return sum/c;
113 }
114
115 };
116
117 // Damit unsere Funktionen die Noten sehen können, müssen wir sie hier oberhalb d
118 void NoteEingeben (NotenListe &l)
119 {
120     // Eingeben
121     cout << "Bitte_das_Fach_eingeben:_";
122     string s;
123     cin >> s;
124     cout << "Bitte_Note_eingeben:_";
125     double note;
126     cin >> note;
127     if (note > 6 || note < 1 )
128         cout << "Falsche_Eingabe\n";
129     else
130     {
131         Note *neu = new Note ();
132         neu->Note = note;
133         neu->Fach = s;
134         l.einfuegen (neu);
135     }
136 }
137
138 void NotenAusgeben (NotenListe &l)
139 {
140     // Ausgeben
141     cout << "\n\tNoten:\n";
142     l.ausgeben ();
143 }
144
145 void DurchschnittAusgeben (NotenListe &l)
146 {
147     cout << "\tDurchschnitt:" << l.durchschnitt () << endl;
148 }
149
150 int main (int argc , char *argv [])

```

```

151 {
152     cout << "Notenverwaltung" << endl;
153     bool beenden=false;
154     NotenListe Noten;
155
156     do // Hauptschleife
157     {
158         cout << "\n\tMenu\n";
159         cout << "1: \_Eingeben\n";
160         cout << "2: \_Ausgeben\n";
161         cout << "3: \_Durchschnitt\n";
162         cout << "8: \_Laden\n";
163         cout << "9: \_Speichern\n";
164         cout << "0: \_beenden\n";
165         cout << " ";
166         int eingabe;
167         cin >> eingabe;
168         double sum=0.0;
169         switch(eingabe)
170         {
171             case 1:
172                 NoteEingeben(Noten);
173                 break;
174             case 2:
175                 NotenAusgeben(Noten);
176                 break;
177             case 3:
178                 DurchschnittAusgeben(Noten);
179                 break;
180             case 8:
181                 Noten.Lesen("notenbuch4");
182                 break;
183             case 9:
184                 Noten.Schreiben("notenbuch4");
185                 break;
186             case 0:
187                 beenden = true;
188                 break;
189             default:
190                 cout << "unbekannte_Eingabe\n";
191         }
192     }
193     while(!beenden);
194
195     return EXIT_SUCCESS;

```


Teil V.

Die C++ Standardbibliothek (STL)

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

12. STL Container

Als **Container** bezeichnet man alle Datenstrukturen 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 häufigsten 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 einzelne Elemente über eine Schlüssel 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 booleschen Werten (true/false)

valarray Ein Feld für Zahlen und Berechnungen(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:

```

1 #include <vector>
2 #include <iostream>
3
4 int main()
5 {
6     // Legt einen vector mit 10 ints an
7     vector<int> zahlen(10);
8     // Legt einen vector mit 10 strings an, die auf "hallo" gesetzt sind
9     vector<string> hallos(20,"hallo");
10    // den 3. int in zahlen auf 5 setzen.
11    zahlen[2]=5;
12    // Gebe den string an der 3. Stelle von hallos aus
13    cout << hallos[2] << endl;
14    // eine andere Möglichkeit, auf bestimmte Elemente von zahlen zuzugreifen
15    cout << zahlen.at(0) << ', ' << zahlen.at(2) << endl;
16    // Das erste Element von zahlen ausgeben
17    cout << zahlen.front() << endl;
18    // Das letzte Element von zahlen ausgeben
19    cout << zahlen.back() << endl;
20    // der vector zahlen wird auf 20 Elemente gesetzt die den wert 42 haben
21    zahlen.assign(20,42);
22    // hänge der 45 an den vector zahlen hinten an
23    zahlen.push_back(45);
24
25    cout << zahlen.back() << endl;
26    // das letzte Element von zahlen entfernen
27    zahlen.pop_back();
28    cout << zahlen.back() << endl;
29    // "Hallo Welt" in hallos an 4. Stelle einfügen, das bisherige 4. Element ist je
30    hallos.insert(3,"Hallo_Welt");
31    // Fügt 4 mal "Hallo Alle" an 7. Stelle in hallos ein.
32    hallos.insert(6,4,"Hallo_Alle");
33    // Entfernt das Element an 5. Stelle aus hallos
34    hallos.erase(5);
35    // Gibt die Anzahl der Elemente in hallos aus
36    cout << hallos.size() << endl;
37    // Leert hallos;
38    hallos.clear();
39    // Gibt "Ja" aus, das hallos leer ist
40    if(hallos.empty()) cout << "Ja" << endl;
41    // Vergrößert hallos auf 100 Elemente.
42    hallos.resize(100);
43 }

```

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

Teil VI.

Hinter den Kulissen

Bisher haben wir den C++ Compiler nur durch das Menu der Entwicklungsumgebung aufgerufen, ohne uns Gedanken zu machen was hinter den kulissen passiert. Außerdem bestanden unsere Programme bisher immer nur aus einer einzelnen Datei. In diesem Abschnitt geht es darum was hinter den Kulissen passiert und wie man damit umgeht wenn die Programme immer länger und komplexer werden.

14. Compiler

14.1. Precompiler

14.2. Compiler

14.3. Linker

15. Bibliotheken

15.1. Header

15.2. Library

16. Projekte aus mehr als einer Datei

16.1. Per Hand

16.2. make und Makefiles

16.3. Autotools

16.4. qmake

17. Projektmanagement

17.1. Codeverteilung

17.2. Versionskontrolle

17.3. Eigene Bibliotheken

Teil VII.

Maus, Fenster und Co

Nach dem wir bisher immer nur die Tastatur verwenden konnten, bereiten wir uns jetzt langsam darauf vor die Maus ins Spiel zu bringen.

18. Events und Mainloop

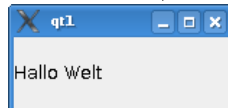
19. GUI - Graphical User Interface

20. Die Qt-Bibliothek

20.1. Graphisches Hello World

```
1 #include <QApplication>
2 #include <QLabel>
3
4 int main(int argc, char *argv[])
5 {
6     QApplication app(argc, argv);
7     QLabel hello(QString("Hallo Welt"), 0, "test");
8     hello.resize(150, 50);
9     hello.show();
10    return app.exec();
11 }
```

Wenn wir das Beispiel übersetzen und starten, erhalten wir das folgende Fenster:



Index

[, 12

BEDINGUNG, 8

BLOCK, 8

bool, 6

break, 11

case, 11

char, 6

cin, 6

class, 36

constructor, 38

cout, 5, 6

Datei, 25

delete, 27

destructor, 38

do-while, 9

double, 6

endl, 6

file, 25

for, 8

fstream, 25

if, 10

ifstream, 25

Instanz, 37

int, 6

iostream, 25

istream, 25

main, 22

new, 27

ofstream, 25

ostream, 25

Parameter, 22

Pointer, 13

private, 36

protected, 36

public, 36

standart output stream, 6

static, 37

struct, 26

switch, 11

virtual, 40

while, 9

Zeiger, 13