Programmieren mit C++

Skript zum Unterricht

3. Aufl. 2005

Inhalt

Programmieren mit C++

0. Dateien eines Projekts

Wenn man mit dem C++Builder ein Programm erstellt, wird ein Projekt erzeugt, das je nach Programmart und –umfang aus verschiedenen Dateien besteht. Die folgende Tabelle gibt einen Überblick über die wichtigsten erzeugten Dateien.

Datei	Dateiendung	Bedeutung
Projekt-Make-Datei	bpr	Enthält den Quelltext sowie Compiler und Linkeroptionen
Unit	cpp	Programmquelltext. Programm können aus mehreren Units bestehen
Headerdatei	h	Headerdateien enthalten z.B. die Prototypen für die verwendeten Funktionen
Objektcode	obj	Der Compiler übersetzt jede Unit in den Objektcode
Programmdatei	exe	Das ausführbare Programm. Zum Ausführen ist das Entwicklungssystem (C++Builder) nicht erforderlich

1. Datentypen

Bei der Deklaration einer Variablen muss einen bestimmten Datentyp angegeben werden. Der Datentyp ist abhängig von den Werten, die die Variable annehmen soll. Außerdem entscheidet der Datentyp darüber, wie viel Platz die Variable im Speicher belegt.

Die wichtigsten Datentypen sind:

Ganzzahlen:	char short int long	Die Speicherplatzbelegung der einzelnen Datentypen ist auch abhängig vom verwendeten Compiler.
Gleitpunktzahlen:	float double long double	
Zeichen:	char	einzelne Buchstaben oder sonst. Zeichen
Zeichenketten:	string	Wörter, Texte

Typ	Größe	Bereich	Beispiele
unsigned char	8 Bit	0 bis 255	Kleine Zahlen, PC-Zeichensatz
char	8 Bit	-128 bis 127	Sehr kleine Zahlen, Ascii-Zeiche
short int	16 Bit	-32.768 bis 32.767	Zähler, kleine Zahlen
unsigned int	32 Bit	0 bis 4.294.967.295	Große Zahlen
int	32 Bit	-2.147.483.648 bis 2.147.483.647	Große Zahlen
unsigned long	32 Bit	0 bis 4.294.967.295	Astronomische Distanzen
enum	32 Bit	-2.147.483.648 bis 2.147.483.647	Geordnete Wertemengen
long	32 Bit	-2.147.483.648 bis 2.147.483.647	Sehr große Zahlen
float	32 Bit	$1{,}18*10^{-38}$ bis $3{,}40*10^{38}$	7-stellige Genauigkeit
double	64 Bit	$2{,}23*10^{-308}$ bis $1{,}79*10^{308}$	15-stellige Genauigkeit
long double	80 Bit	$3{,}37*10^{-4932}$ bis $1{,}18*10^{4932}$	18-stellige Genauigkeit

2. Bildschirmausgabe und Tastatureingabe

a) Bildschirmausgabe mit cout

Die Bildschirmausgabe erfolgt mit dem Objekt cout. Mit dem Einfügeoperator << werden Variableninhalte oder Zeichenketten an das Objekt gesendet.

Beispiel: double Betrag = 1500,50;
cout <<“Summe “ <<Betrag << “ EUR“;

Ausgabe: Summe 1500,00 EUR

b) Tastatureingaben mit cin

Die Eingabe über die Tastatur erfolgt mit dem Objekt cin. Mit dem Einlesoperator >> werden die Daten in Variablen platziert.

Beispiel: cout << “Bitte eine Zahle eingeben: “;
Cin >>ErsteZahl;

es können auch mehrere Werte eingelesen werden.

cout << “Bitte zwei Zahlen eingeben: “;
Cin >>ErsteZahl >>ZweiteZahl;

ba) Eingabe von Strings
Mehrere Wörter hintereinander können mit cin nicht eingegeben werden, da cin Leerzeichen als Trenner behandelt und die Eingabe damit abbricht. Eine Eingabemöglichkeit für Texte ist die Methode **cin.getline(char *text, int groesse, char trenner='\n');**
Bsp.: cin.getline(name, 30, '\n');

3. Formatierte Ausgabe mit Manipulatoren

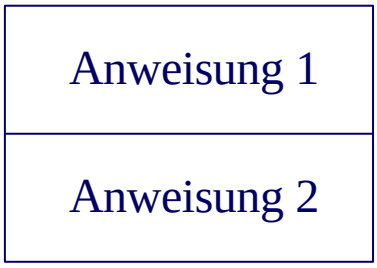
Um Informationen besser formatieren zu können, stellt C++ eine Reihe von Manipulatoren zur Verfügung. Viele Dieser Manipulatoren werden durch die Headerdatei **iomanip.h** eingebunden.

Manipulator	**Bedeutung**
setw(int n)	Festlegung der Feldbreite (gilt nur für die nächste Einfügeoperation <<
setfill(char)	Festlegung des Füllzeichens
left	Linksbündig
right	Rechtsbündig
scientific	Exponentialdarstellung
fixed	Gleitpunktdarstellung
showpos	Anzeige des positiven Vorzeichens
uppercase	Großbuchstaben
setprecision(int n)	Anzahl der Nachkommastellen
dec, hex, oct	Umwandlung in andere Zahlensysteme

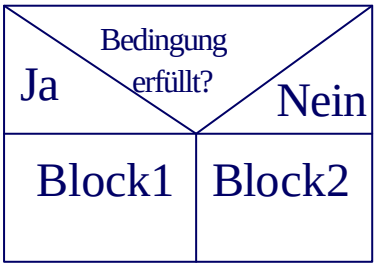
Die Anwendung dieser Manipulatoren muss an einem Beispiel erläutert werden.

4. Struktogramme nach Nassi u. Shneiderman

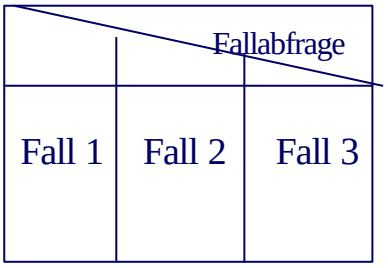
Zur Darstellung des Programmablaufes und der Programmlogik verwendet man Darstellungstechniken wie Programmablaufpläne (PAP) oder Struktogramme nach Nassi und Shneidermann. Der Struktogrammtechnik soll hier der Vorzug gegeben werden, da sie eine strukturierte und übersichtliche Programmierung erzwingt. Struktogramme dienen sowohl der Vorbereitung eines Programmes wie der Dokumentation.
Struktogramme kennen die folgenden Konstrukte:



Reihe von Anweisungen

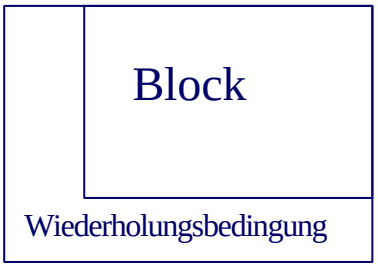


Verzweigung



Fallabfrage (switch .. Case)

Schleife mit Anfangsbedingung
(Kopfschleife)



Schleife mit Endebedingung
(Fußschleife)

5. Steueranweisungen

Mit der Verwendung von Steueranweisungen beginnt das Programmieren erst richtig. Steueranweisungen ermöglichen es, Programmteile beliebig zu wiederholen oder Programmteile in Abhängigkeit von Bedingen auszuführen. Mit der Verwendung von Steueranweisungen sollte man sich mit der Programmierlogik beschäftigen. Umfangreichere Programme werden in Module zerlegt, der Ablauf des Programmes wird graphisch dargestellt.
Zur Darstellung eignen sich zwei Methoden. Der **Programmablaufplan** oder das **Struktogramm**. Dem Struktogramm soll hier zunächst der Vorzug gegeben werden. (Anlage)

5.1 Die IF ... ELSE – Anweisung

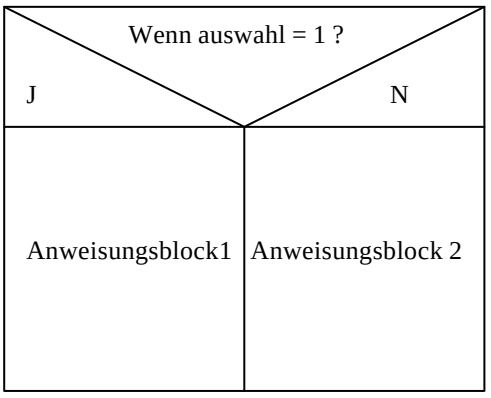
Aufgabe: Schreiben Sie ein Programm, welches einen Betrag wahlweise DM in Euro oder Euro in DM umrechnet.

Erstellen Sie zunächst das Struktogramm.

```
Bildschirmausgabe

DM/EURO-Umrechnung
==================

Betrag eingeben:

1 – Umrechen in Euro
2 – Umrechnen in DM
--------------------------
Auswahl (1 oder 2): 1
```

Aufbau der IF-Anweisung

```
char auswahl;
cout<<(“Auswahlziffer eingeben:  “);
cin>>auswahl;
if (auswahl==‘1‘)
{
        Anweisungsblock1
}
else
{
        Anweisungsblock2
}
```



In der IF-Anweisung können folgende **Vergleichsoperatoren** verwendet werden:

a < b	kleiner als	a > b	größer als
a <=b	kleiner und gleich	a >=	größer und gleich
a = =b	gleich	a!=b	nicht gleich

Bsp.: Abfrage, ob a gleich 50 ist: if(a= =50)
Abfrage, ob a ungleich 50 ist: if(a!=50)

Vergleichsoperatoren können auch durch die **logischen Operatoren &&** (und) oder **||** (oder) verknüpft werden.
Bsp.: Abfrage, ob a größer als 50 und kleiner als 60 ist: if (a>50 && a<60)

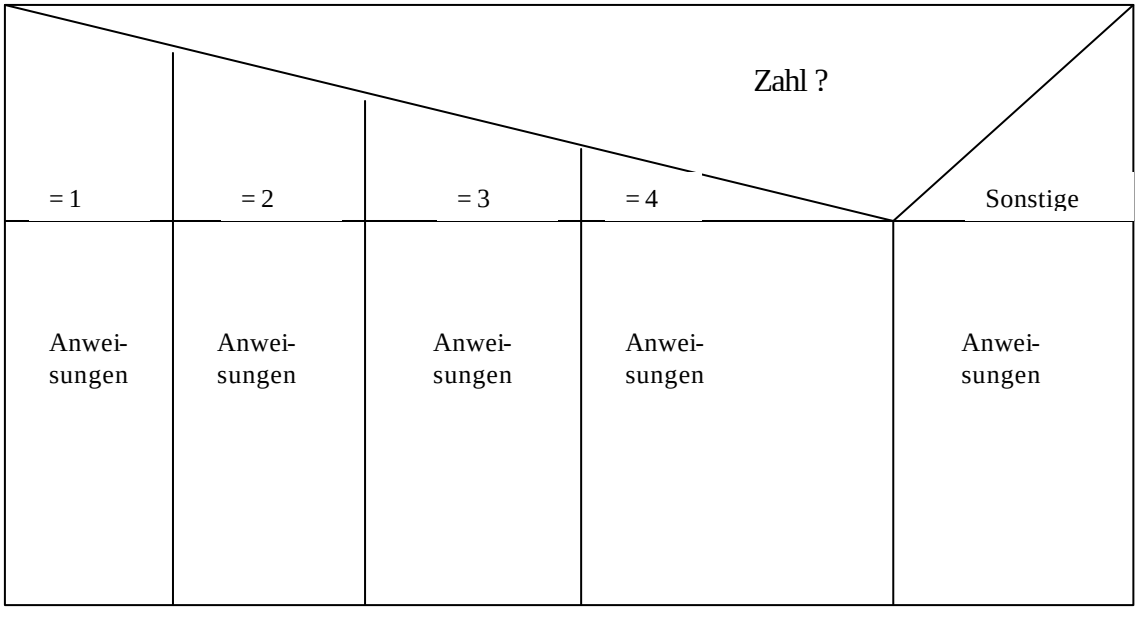
Abfrage, ob a den Wert 0 oder 9 hat: if (a= =0 || a= =9)

5.2 Fallunterscheidung: switch...case

Beispiel: Es soll ein Programm geschrieben werden, bei dem nach der Eingabe einer Zahl zwischen 1 und 4 gefragt wird. Bei jeder der vier möglichen Zahlen, soll eine bestimmter Text
angezeigt werden. Bisher wäre diese Aufgabe nur mit mehreren If-Anweisungen zu lösen gewesen. Die switch-Anweisung löst dieses einfacher.

```
void main(void)
{
        int zahl;
        cout<<(´´ \nBitte eine Zahl zwischen 1 und 4 eingeben: ´´);
        cin>>zahl;
        switch(zahl)
        {
                case 1: cout<<´´\nDies war die 1´´;
                        break;
                case 2: cout<<´´\nDies war die 2´´;
                        break;
                case 3: cout<<´´\nDies war die 3´´;
                        break;
                case 4: cout<<´´\nDies war die 4´´;
                        break;
                default: cout<<“Keine Zahl zwischen 1 und 4“;
        }
        getch();
}
```

Wenn auch der Sinn dieses Programmbeispiels recht banal ist, so zeigt er doch das Wesentliche. Neu ist die Anweisung **break**. Die break-Anweisung springt hinter den Anweisungsblock der switch-Anweisung oder hinter eine do, for oder while-Schleife. Die Switch-Anweisung wird im Struktogramm mit dem case-Konstrukt dargestellt.



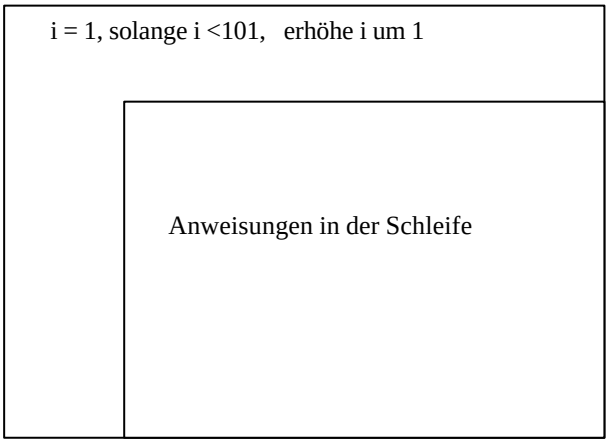
5.3 Schleifen: for (Zählergesteuerte Schleife)

Beispiel: Wir wollen die Zahlen von 1 bis 100 untereinander auf dem Bildschirm ausgeben.

a) Erstellen des PAP (Struktogrammes)

b) Programmbeispiel

```
void main()
{
        int i;
        clrscr();
        for (i=1;i<101;i++)
        {
                cout<<i<<endl;
        }
        getch();
}
```



Aufbau der For-Schleife: **for(Startanweisung;Wiederholungsbedingung;Veränderung)**
{
Anweisungen;
}

Übungen:

1. Von den Zahlen 1 bis 500 sollen jeweils nur die geraden Zahlen ausgegeben werden.

2. Ausgabe der Zahlen 1 bis 100 wie oben. Es sollen jedoch nur jeweils 25 Zahlen angezeigt werden. Danach soll zu einem Tastendruck aufgefordert werden, um die nächsten 25 Zahlen anzuzeigen. Hinweis: Verwenden Sie den Modulo-Operator (%).

3. Gegeben sind folgende For-Anweisungen. Was bewirken sie:

   a) for(x=1;x<=200;x=x+2);

   b) for(x=1;x=10000;x++);

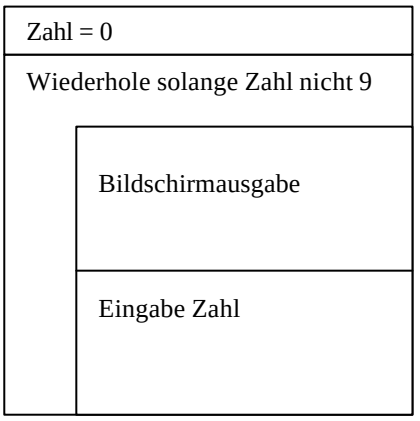
   c) for(x=2;x>1;x++);

   d) for(x=1;x<20;);

4. Die nützliche Funktion clrscr() gehört nicht zum Sprachumfang von ANSI-C. Manche Compiler kennen die Funktion nicht. Schreiben Sie mit Hilfe einer for-Schleife eine eigene Funktion mit dem Namen **clear**, die sie als Bibliotheksfunktion in jedem Programm verwenden können.

5.4 Schleife mit Anfangsbedingung: while

Beispiel: Der Benutzer wird zur Eingabe einer Zahl aufgefordert. Die Eingabe wird abgebrochen, wenn eine 9 eingegeben wird.



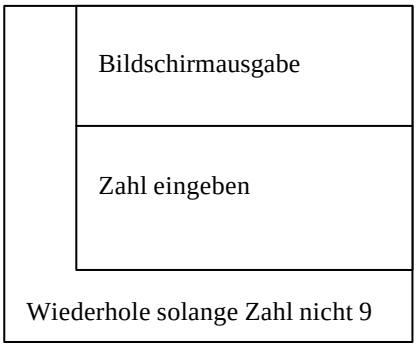

```
void main()
{
        int zahl=0;
        clrscr();
        while(zahl!=9)
        {
                cout<<´´Bitte eine Zahl eingeben (9 für Ende): ´´;
                cin>>zahl;
        }
        getch();
}
```

Achtung: Die Variablen, die in der while-Anweisung geprüft wird, muss vor Eintritt in die Schleife einen definierten Wert besitzen.
Beim Eintritt in die while-Schleife wird geprüft, ob die Abbruchbedingung erfüllt ist. Dies ist hier nicht der Fall, da die Variable Zahl mit dem Wert 0 vorbesetzt wurde.

While-Schleifen können im Prinzip auch als Zählschleifen, wie die for-Schleife verwendet werden. Eleganter ist allerdings die for-Schleife, da hier eine Anweisung gespart wird.
While Schleifen werden häufig so benutzt, wie in obigen Beispiel. Ein Anweisungsblock wird solange durchgeführt, bis eine Abbruchbedingung eingegeben wird.

5.5 Schleife mit Endebedinung: do ... while

Beispiel: Der Benutzer wird zur Eingabe einer Zahl aufgefordert. Die Eingabe wird abgebrochen, wenn eine 9 eingegeben wird.




```
void main()
{
        int zahl;
        clrscr();
        do
        {
                cout<<´´Bitte eine Zahl eingeben (9 für Ende): ´´;
                cin>>zahl;
        } while(zahl !=9);
        getch();
}
```

Im Unterschied zur while-Schleife muss die abgefragte Variable nicht vorbesetzt werden, da die Schleife in jedem Fall einmal durchlaufen wird.
Do...while Schleifen werden oft zur Steuerung eines Auswahlmenüs benutzt.

Übungen:

1. Über eine kopfgesteuerte Schleife werden Zahlen eingebeben (Abbruch, wenn Zahl =0). Anschließend wird die Summe aller Zahlen angezeigt.

2. Über eine fußgesteuerte Schleife soll der Anwender gezwungen werden, eine gültige Zahl zwischen 1 und 255 einzugeben. Der Abbruchwert ist die Zahl 0.

5.5 Die Continue-Anweisung

Die Anweisung continue kann innerhalb einer for, while oder do...while Schleife verwendet werden. Sie beendet den Anweisungsblock und springt an den Kopf der Schleife.

Beispiel:

```
x=0;
while(x<20)
{
        x++;
        if(x==10)
          continue;
        printf(´´x = %i\n´´,x);
}
```

Es werden die Zahlen 1 bis 19 außer der Zahl 10 ausgegeben.

6. Funktionen

Aufgabe: Schreibe ein Programm 'Kreis', welches Fläche und Umfang eines Kreises berechnet, wenn ein Radius eingegeben wird.

a) Löse die komplette Aufgabe in der Funktion main
b) Berechne Fläche und Umfang jeweils in einer eigenen Funktion berechne_flaeche bzw. berechne_umfang und gib die Ergebnisse in dieser Funktion aus.
c) Berechne Fläche und Umfang jeweils in einer eigenen Funktion (wie b), gib die Ergebnisse aber in der Funktion main aus.
d) Erstelle das Programm so, dass die Funktion main etwa wie folgt aussieht:

```
void main()
{
        eingabe_radius();
        berechne_flaeche(r);
        berechne_umfang(r);
    }
```

Um obige Aufgaben zu lösen, muss zunächst einiges über Funktionen gesagt werden.

Bsp.

```
void text1(void)
{
    cout<<´´Ich bin ein Zeile 1´´<<endl;
}
void text2(void)
{
    cout<<´´Ich bin Zeile 2´´<<endl;
}
void main(void)
{
    text1();
    text2();
}
```

Durch Funktionen wird ein Programm in Teile (Module) zerlegt. Jede Funktion erfüllt eine Teilaufgabe des Programmes. Funktionsaufrufe erkennt man an den runden Klammern hinter dem Funktionsnamen.
Funktionen können im Programm der Funktion main vorangestellt werden. Stehen sie erst hinter der Funktion main, so müssen sie der Funktion main durch eine sog. **Prototyp-deklaration** zunächst bekannt gemacht werden.

Bsp_1.

```
void text2(void);          ← Prototypdeklaration
void text1(void)
{   cout<<´´Ich bin Zeile 1´´<<endl; }

void main(void)
{   text1();
    text2(); }
void text2(void)
{
    cout<<´´Ich bin Zeile 2\n´´<<endl;
}
```

In diesem Fall handelt es sich um Funktionen ohne Parameterübergabe und ohne Rückgabewert.

Beisp_2: Funktion mit Parameterübergabe

Bei der Funktionsdeklaration muss der Variablentyp des Übergabewertes in der Klammer hinter dem Funktionsnamen angegeben werden
(z.B. void berechne (int zahl))**. Der Name der übergebenen und der in der Funktion deklarierten Variablen müssen nicht identisch sein, da beim Aufruf nicht die Variable, sondern deren Wert an die Funktion übergeben wird**

```
void verdoppeln(float);
void main()
{       float zahl,d;
        cout <<"Zahl: ";
        cin >>zahl;
        verdoppeln(zahl);                    Funktionsaufruf mit Werterübergabe
}
```



```
void verdoppeln(float zahl)
{       float e;
        e=2*zahl;
        cout<<"Das doppelte der Zahl ist "<<e<<endl;  }
```

Beisp_3: **Funktion mit Rückgabewert**

Eine Funktion, die einen Wert zurückliefert muss den Typ des Rückgabewertes dem Funktionsnamen voranstellen (z.B. float berechne(void))
Durch die Return-Anweisung am Ende der Funktion, kann ein Wert zurückgeliefert werden.
Der zurückgegebene Wert steht zwischen der Anweisung return und dem Semikolon.
Bsp. return (zahl);

```
float verdoppeln(float);

void main()
{       float zahl,d;
        cout <<"Zahl: ";
        cin >>zahl;
        cout<<"Das doppelte der Zahl ist "<<verdoppeln(zahl);                    }
}

float verdoppeln(float zahl)
{       float e;
        e=2*zahl;
        return e;
}
```

7. Felder und Strings

7.1 Eindimensionale Felder

Beispiel: Es sollen fünf Zahlen eingegeben werden. Anschließend sollen diese Zahlen
a) in der Reihenfolge ihrer Eingabe wieder ausgegeben werden und
b) in sortierter Reihenfolge ausgegeben werden.

Lösungshilfen:
Selbstverständlich definieren wir nicht für jede eingegebene Zahl eine eigene Variable, sondern wir definieren eine Tabelle (Feld oder Array) mit 5 Elementen.

```
int zahlenfeld[5];
```

Diese Anweisung definiert ein Feld namens Zahlenfeld, welches fünf Integervariablen aufnehmen kann.
Der Zugriff auf die erste Zahl beginnt mit zahlenfeld[0], der Zugriff auf die 5. Zahl mit zahlenfeld[4].

Das erste Element eines Arrays hat den Feldindex 0!

```
void main(void)
{
        int zahlenfeld[5];
        int i;
        for(i=1;i<=4;i++)
        {
                cout<<``Zahlen eingeben: ´´;
                cin>>zahlenfeld[i];
        }
}
```

So sieht die Eingabeschleife für die 5 Zahlen aus.

Erstellen Sie dazu zunächst ein Struktogramm!

7.2 Mehrdimensionale Felder

Tabellen bestehen normalerweise aus mehrern Spalten und Zeilen. Angenommen Sie benötigen eine Tabelle mit 3 Spalten und 10 Zeilen für Integerzahlen, so können Sie diese in C folgendermaßen definieren:

```
int tabelle[3][10];
```

Das Tabellenelement in der 2. Spalte und der 5. Zeile soll den Wert 999 erhalten:

```
tabelle[1][4]=999;
```

Beachten Sie, dass auch hier die Indizierung mit 0 beginnt.

In obigem Beispiel wurde eine zweidimensionale Tabelle definiert. Genauso lassen sich auch mehrdimensionale Tabellen definieren.

Bsp.: int tabelle[80][80][80]; definiert eine Tabelle mit 80x80x80 = 512000 Integerzahlen.

7.3 Strings

Der Datentyp char dient lediglich zum Speichern eines einzelnen Zeichens. Zum Speichern von Namen, Texten, Bezeichnungen usw. benötigen wir einen komplexen Datentyp in Form eines eindimensionalen Feldes von Typ char.

Bsp. char text[20]="Dies ist ein String";

Dies ist ein Stringfeld, das Platz für 20 Zeichen bietet. Das Stringendezeichen ist das ASCII-Null-Zeichen '\0', welches automatisch beim Betätigen der Return-Taste erzeugt wird. Bei der Definition des Stringfeldes ist darauf zu achten, das noch Platz für das Endekennzeichen bleibt.

Eine Wertzuweisung der Form: text="Dies ist ein String" funktioniert nicht, da der String ein Array ist und sein Name immer für eine Adresse steht.
Wenn einem String ein Wert zugewiesen werden soll, muss mit einer der vielen String-Funktionen gearbeitet werden. Die Lösung sieht so aus:

strcpy(text,"Dies ist ein String");

weiter String-Verarbeitungsfunktionen

strcpy(ziel, quelle)	kopiert String *quelle* mitsamt Endekennung in den string *ziel*.
strcat(ziel, quelle)	kopiert String *quelle* mit Endekennung hinter den string *ziel*.
strlen(string)	ermittelt die Länge des Strings *string* ohne Endekennung

strchr(string, zeichen)	Sucht nach einem bestimmten *Zeichen* in *string* und liefert die Adresse dieses Zeichens. Bsp.:
	`char *ptr;`
	`ptr=strchr(string,´e´);`
	`cout<<"Position: "<<ptr;`
	(Die Variable ptr ist ein Zeiger und liefert die Adresse der Position an der das Zeichen `e` zum ersten Mal im String vorkommt.)

Einlesen von Textzeilen über die Tastatur
Beim Einlesen von Strings über die Tastatur mit Hilfe der Funktion **cin** taucht das Problem auf, dass Leerzeichen (Whitespace) als Stringendezeichen interpretiert werden. Eingaben wie "Manfred Meier" sind also nicht möglich, da dem String nur das Wort Manfred zugewiesen wird. Der String "Meier" wird der folgenden Variablen zugewiesen, was u. U. zu einem Programmabsturz führt, wenn diese Variable etwa eine Zahl ist.
Eine Eingabe von Strings mit Leerzeichen über die Tastatur ist mit den Funktionen **gets** oder **getline** (siehe S. 3) möglich. Beispiel:

cin.getline(name,anzahl_zeichen); oder
gets(name);

```
Bsp:    char text[20];
        int textlen;
        cout <<"Textzeile eingeben";
        gets(text);          //Alternative s. unten
        textlen=strlen(text);
        cout<<"Ausgabe: "<<text;
        cout <<"Textlaenge: "<<textlen;


oder    cin.getline(text,20);
```

Der Datentyp string

Mit Verwendung des Datentyps string kann die Verarbeitung von Zeichenketten stark vereinfacht werden.
Einem String kann nun auf folgende Weise ein Wert zugewiesen werden:

String text = “Beispieltext“;

Oder durch

Cin >>text;

Strings können auch als Funktionsparameter verwendet werden.

Mit der Einbindung der Bibliothek string.h können zusätzliche Funktionen zur Verarbeitung von strings verwendet werden..

text.length() liefert die Anzahl der Zeichen im String

Weitere String-Funktionen sind: **clear(), insert(), replace(); erase(), find(),** ....

Anmerkung: Beim Einlesen eines Strings von der Tastatur, kommt es immer wieder zu Problemen. Insbesondere, wenn Eingaben unterschiedlicher Datentypen aufeinander folgen, gibt es immer wieder Überraschungen. Es wird daher geraten, die Behandlung von Strings sorgfältig auszutesten.

8. Zeiger (Pointer)

Beim Aufruf von Funktionen kann immer nur eine Variable von einfachem Datentyp wie int, float oder char übergeben werden. Was tut man aber ,wenn man ein Array oder einen anderen komplexen Datentyp in einer Funktion bearbeiten will? Hier darf nicht der Wert einer Variablen übergeben werden, sondern die Adresse, an der der komplexe Datentyp beginnt. Dieses nennt man einen Zeiger.

Eine Variable beinhaltet einen Wert.
Ein Zeiger beinhaltet die Adresse einer Variablen oder eines komplexen Datentyps.

Ein Zeiger wird mit dem * definiert.

Bsp.:
int i=10; Die Variable i erhält den Wert 10
int *i_zeiger; Definition eines Zeigers mit dem Namen i_zeiger, der auf einen Integerwert zeigt
i_zeiger=&i dem Zeiger i_zeiger wird die Adresse der Variablen i zugewiesen.

Speicheradresse 0x3fc

```
Cout <<"Inhalt der Variablen: " <<i<<"Adresse der Variablen i "<<i_zeiger;
```

Diese Anweisung würde uns den Wert 10 sowie die Adress 0x3fc anzeigen.

Dereferenzierung
Man kann über den Zeiger auch wieder den Inhalt der Adresse, auf die der Zeiger zeigt ausgeben. Dazu wird der Stern ´*´ verwendet (Dereferenzierungsoperator)

Die Anweisung `cout<<*i_zeiger;` führt zur Ausgabe von 10

Ein sinnvolles Beispiel für die Verwendung von Zeigern ist die in Kapitel 6 genannte Aufgabe zur Sortierung eines Zahlenfeldes. Wenn die Sortierung in einer eigenen Funktion geschehen soll, so funktioniert dieses nur unter der Verwendung eines Zeigers:

```
void sortier (int *zf)                  //Übergabeparameter muss als Zeiger deklariert werden
{
  ...Sortiervorgang
}

void main()
{
   int zahlenfeld[5];
   ... Eingabe der Zahlen
   sortier(zahlenfeld)                  // Der Name eines Feldes steht für die Anfangsadresse des
                                        // Feldes
}
```

Wenn eine Variable an eine Funktion übergeben wird, so wirkt sich die Veränderung dieser Variablen innerhalb der Funktion nicht auf das Hauptprogramm aus. Bei der Übergabe einer Adresse als Zeiger, kann man Werte innerhalb einer Funktion dauerhaft beeinflussen. D.h. das Zahlenfeld in unserem Beispiel ist anschließend auch in der Funktion main sortiert.

9. Strukturen

Beispiel: In einer Abteilung möchte man eine einfache Adressverwaltung für Kunden und Lieferanten erstellen. Folgende Inhalte sollen erfaßt werden:

Feldname	Datentyp	Länge
Name	char	30
Strasse	char	25
PLZ	char	5
Ort	char	15
fon	char	15
fax	char	15
eMail	char	20
Kennz.	int	1

Kennz.
1 – Kunden
2 - Lieferanten

Da diese Daten logisch zusammengehören, ist es sinnvoll diese Daten zu einem Datensatz zusammenzufassen. In C fasst man solche Daten zu einer **Struktur** zusammen.

```
struct Adresse
{
        char name[30];
        char strasse[25];
        char plz[5];
        char ort[15];
        char fon[15];
        char fax[15];
        char email[20];
        int kennz;
};                          //Der Anweisungsblock einer Struktur wird mit Semikolon abgeschlossen

struct Adresse adressdaten[10 ];
```

Erläuterung: Wir haben eine Struktur namens *Adresse* deklariert. Eine Struktur ist ein sog. komplexer Datentyp in C. Die einzelnen Elemente der Struktur bezeichnet man auch als *Member*. Wenn wir diesen Datentyp verwenden wollen, müssen wir eine Variable vom Typ Adresse definieren. Dieses geschieht in der letzten Anweisung. (Wenn wir Kunden und Lieferanten getrennt erfassen wollen, können wir zwei Variablen vom Typ Adresse definieren: struct Adresse kunden[10] und struct Adresse lieferer[10]).
Die eckige Klammer hinter adressedaten bedeutet, dass wir ein Feld definieren, welches 10 Adressedaten aufnehmen kann.

Auf die einzelnen Elemente einer Struktur wird wie folgt zugegriffen:

a) Wertzuweisungen

```
strcpy(adressdaten[0].name,“Firma Meier GmbH“);     //Stringzuweisung
adressdaten[0].kennz=1;                             //Zuweisung einer Integer-
                                                    Variablen
```

D.h. ein Element einer Struktur wird mit dem Variablennamen und Index sowie dem Elementnamen getrennt durch einen Punkt zugegriffen.

b) Einlesen eines Wertes von Tastatur
gets (adressdaten[0].name); oder cin.getline(...)

gets oder cin.getline ermöglicht auch die Eingabe von Leerzeichen in Strings.

c) Ausgabe
cout <<´´Name: “<<adressdaten[0].name;

Zeiger auf Strukturen

Für das sinnvolle Arbeiten mit Strukturen müssen diese an Funktionen übergeben werden können. Komplexe Datentypen lassen sich aber wie in Kapitel 7 beschrieben nur mit Hilfe von Zeigern übergeben.

Beispiel: Wir wollen unsere Adressdaten in einer eigenen Funktion einlesen und in einer anderen Funktion wieder auslesen.

```
void eingabe(struct Adresse *adrptr[10])
{
        for (int i=0;i<5;++i)
        {
                gets(adrptr[i]->name);     // Zugriff erfolgt über den Zeiger mit der
                                           // Schreibweise zeiger[index)->Element !!
                usw.
        }
}

void ausgabe(struct Adresse *adrptr[10])
{
        for (int i=0;i<5;++i)
        {
                cout<<´´Name: ´´<<adrptr[i]->name;
                usw.
        }
}

void main()
{
        struct Adresse
        {
                wie oben
        };
        struct Adresse *adrptr[10];          //Zeiger auf die Struktur vom Typ Adresse
        adrptr[0]=adressdaten;               //Zeiger die Adresse von adressdaten
                                             // zuweisen
        eingabe(adrptr);
        ausgabe(adrptr);
}
```

Anmerkung: Bis jetzt können wir unsere Daten noch nicht dauerhaft speichern. Dazu müssen wir uns erst noch mit der Dateiverarbeitung beschäftigen.

10. Dateiverarbeitung

Bisher könnten wir Daten lediglich während der Laufzeit eines Programmes nutzen. Wenn wir Daten längerfristig benötigen und evtl. auch von anderen Programmen auf diese Daten zugreifen wollen, müssen wir die Daten als **Datei** auf einem Datenträger abspeichern.

Eine Datei ist folgendermaßen aufgebaut:

Datei:	Zusammenfassung mehrerer Datensätze, die unter einem Dateinamen auf einem Datenträger gespeichert sind (Bsp: Kundendatei).
Datensatz:	beschreibt ein Datenobjekt durch die Zusammenfassung logisch zusammengehöriger Datenfelder (Bsp. Kunde xy).
Datenfeld:	Eine Dateneinheit, die aus einer Folge von Zeichen besteht (Bsp. Kundennummer).
Zeichen:	Kleinstes Datenelement (Buchstabe, Ziffer, Sonderzeichen)

Dateien können auf verschiedene Art gespeichert werden:

Sequentielle Speicherung:	Alle Datensätze werden fortlaufend hintereinander gespeichert. Um auf einen bestimmten Datensatz zuzugreifen, müssen alle vorhergehenden Datensätze abgearbeitet werden.
Indizierte Speicherung:	Neben dem unsortierten Datenbestand gibt es zusätzlich eine sog. Indexdatei, die einen Ordnungsbegriff und die dazugehörige Speicheradresse enthält
Verkettete Speicherung:	Jeder Datensatz enthält einen Zeiger, der auf die Adresse des folgenden Datensatzes zeigt.
Gestreute Speicherung:	Es besteht ein rechnerischer Zusammenhang zwischen den Adressen der einzelnen Datensätze.

Der Zugriff auf einen Datensatz kann auf zwei Arten erfolgen:

Sequentieller Zugriff und wahlfreier, direkter Zugriff

10.1 Zugriff auf sequentielle Dateien

Um mit sequentiellen Dateien arbeiten zu können, benötigt man folgende Anweisungen:
(Beispieldatei: Kunden.dat)

Ein Filepointer vom Typ FILE	**FILE *kunddatei**;
Anweisung zum Öffnen der Datei	**kunddatei = fopen("Kunden.dat", "w");** (wobei "w" der Modus zum Öffnen der Datei ist. Siehe (Liste)
Anweisung zum Lesen in der Datei	**fread(&ksatz,sizeof(ksatz),1,kunddatei);** (Für den Aufbau des Datensatzes legen wir eine Struktur an, die wir als ksatz bezeichnen, sizeof bezeichnet die Größe des Datensatzes 1 ist die Anzahl der Blöcke(Datensätze), die bei einem Lesevorgang gelesen werden; kunddatei ist der Filepointer, auf den sich der Lesevorgang bezieht.)
Anweisung zum Schreiben in der Datei	**fwrite(&ksatz,sizeof(ksatz),1,kunddatei);**
Anweisung zum Schließen der Datei	**fclose(kunddatei);**

Liste der Bezeichner beim Öffnen einer Datei

r	Öffnen zum Lesen. Falls die Datei nicht existiert, gibt fopen() einen NULL-Pointer zurück.
w	Öffnen zum Schreiben. Falls die Datei schon existiert wird sie überschrieben, andernfalls neu erstellt
a	Öffnen zum Anhängen von Daten ans Dateieinde. Falls die Datei nicht existiert, wird sie erzeugt.
r+	Öffnen zum Lesen und Schreiben. Falls die Datei nicht existiert, gibt fopen() einen NULL-Pointer zurück.
w+	Öffnen zum Lesen und Schreiben. Falls die Datei existiert, geht ihr Inhalt verloren, wenn nicht, wird sie neu erstellt.
a+	Öffnen zum Lesen und Schreiben. Der Positionszeiger wird ans Dateiende gestellt. Falls die Datei nicht existiert, wird sie neu erzeugt.

Öffnen einer Datei:
Beim Öffnen einer Datei muss unbedingt der Erfolg dieses Vorgangs überprüft werden. Wenn fopen() nämlich scheitert, ist oft ein unkontrollierter Programmabsturz die Folge.

Folgendes Beispiel zeigt, wie man es macht:

```
FILE *Testdatei;
Testdatei = fopen("Beispiel.dat", "w");
If(Testdatei == NULL)
{
        cout <<"Fehler beim Öffnen der Datei";
        getch();
        exit(1);                                    //Abbruch mit Fehlercode
}
else
{
        // weitere Verarbeitung der Datei
}
```

Lesen einer Datei bis zum Ende

Zur Überprüfung, ob das Dateiende erreicht ist, wird die C-Funktion **feof()** aufgerufen.
Bsp.: while(! feof(Testdatei) { //Datei lesen }

Aufgabe:

Eine Kundendatei hat folgende Satzstruktur:

Kundennummer	kdnr	integer
Kundenname	name	char[20]

Arbeiten Sie in zwei Gruppen:
Gruppe A: erstellt ein Programm „**Schreibkunde**", welches Datensätze in die Datei schreibt.
Gruppe B: erstellt ein Programm „**Lieskunde**", welches die Datei liest und ausgibt.

Der Name der Datei ist **kunden.dat** und befindet sich auf ihrem Netzlaufwerk.

11. Einführung in die objektorientierte Programmierung

Die Notwendigkeit für objektorientierte Programmierung ergibt sich daraus, dass heute Programme auf verschiedenen Plattformen laufen sollen, dass von verschiedenen Programmen auf gemeinsame Daten zugegriffen wird, dass Veränderungen von Programmen mit möglichst geringem Aufwand vollzogen werden können.
Wird beispielsweise der Datentyp eines Datenfeldes einer Datenbank geändert, so müssen alle Programme und Funktionen, die auf diese Datenbank zugreifen, verändert werden. Die Umstellung auf den Jahrtausendwechsel hat dies besondert deutlich gemacht. Bei der objektorientierten Programmierung entfallen diese Arbeiten, da man nur ein einer einzigen Stelle Änderungen an der Datenbank und Änderungen an den Methoden vornehmen muss.
Bei der **prozeduralen** Programmierung stehen die **Funktionen** im Vordergrund, bei der **objektorientierten (OOP)** Programmierung stehen die **Daten** im Vordergrund.

Die Idee der OOP ist es, Daten und Funktionen, die auf die Daten zugreifen in einer einzigen Einheit (Objekt) zusammenzufassen. Man spricht dann von einer **Klasse**.

Eine Klasse beinhaltet die Eigenschaften (= Daten) sowie die Methoden, die mit diesen Daten arbeiten.

Beispiel: Die meisten Spiele sind objektorientiert programmiert. Die Grundstruktur vieler Spiele besteht darin, dass es eine Spiellandschaft gibt, in der verschiedene Objekte (Feinde, Verteidiger) irgendwelche Ziele verfolgen.
Nehmen wir an, wir wollen ein einfaches Spiel konstruieren, in dem sich verschiedene Tierarten auf einem Spielfeld mit Hindernissen bewegen. Dazu benötigen wir folgende Objekte: Ein Spielfeld, unterschiedliche Tiere, Hindernisse.

Ein Tier ist ein **Objekt**, das bestimmte **Eigenschaften und Funktionen** besitzt. Diese werden zunächst in einer **Klassendefinition** genau beschrieben. Ein Tier soll zunächst einmal durch die **Eigenschaft**

- Position auf dem Spielfeld

beschrieben werden. Außerdem soll es die wenigen **Funktionen**

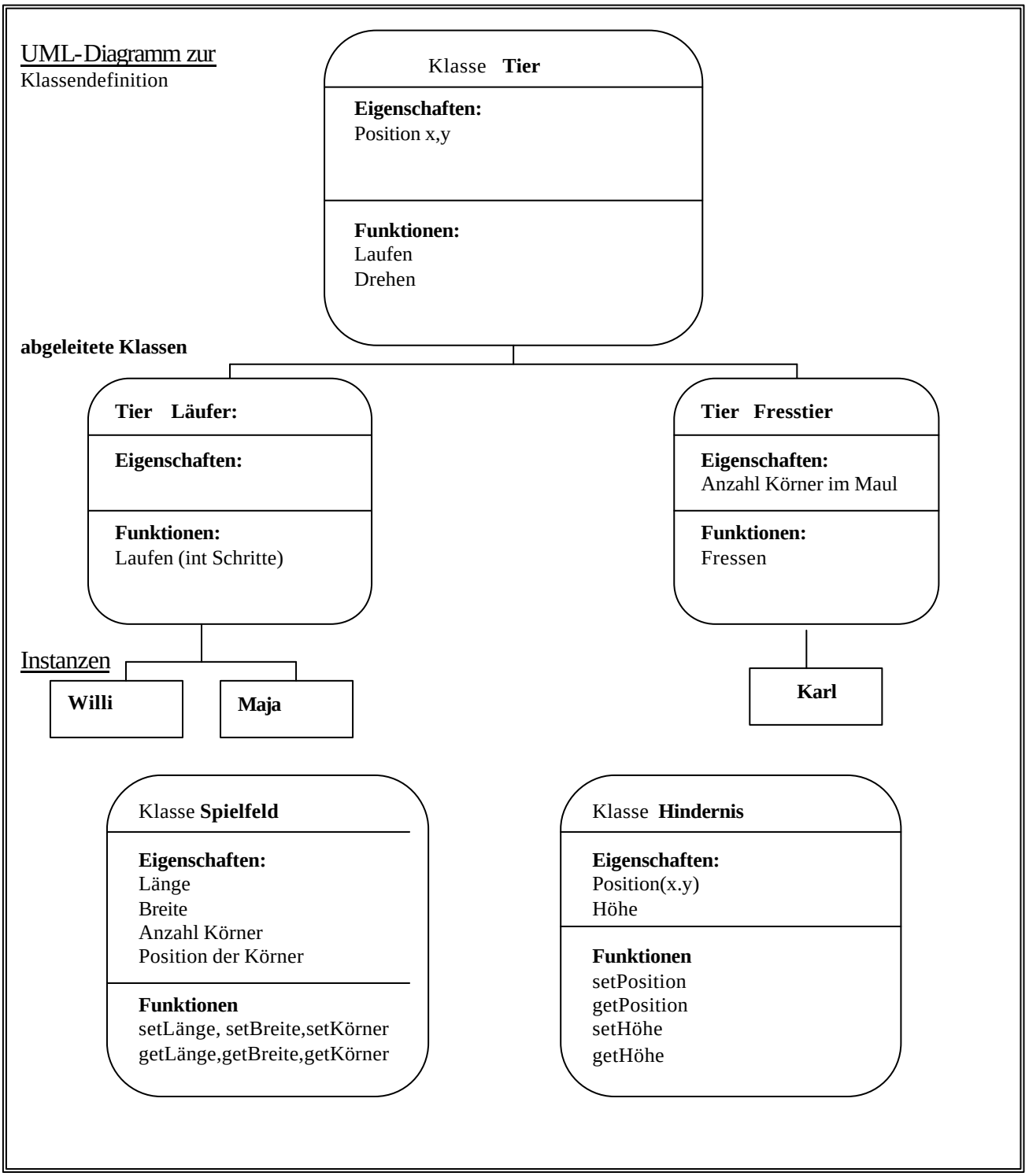
- Laufen
- Drehen

durchführen können.

Nun gibt es unterschiedliche Tiere, die sich durch ihre Eigenschaften und Funktionen unterscheiden. Sie haben aber alle gemeinsam, dass sie zur **Klasse Tier** gehören.
Nehmen wir an, es gibt Läufer und Fresstiere. Läufer sind Tiere, die über mehrere Felder gleichzeitig laufen können, Fresstiere sind Tiere, die Körner fressen können.
Diese Tiere werden von der Klasse Tier abgeleitet und mit zusätzlichen Eigenschaften und Funktionen ausgestattet.

Dadurch, dass jetzt definiert wurde, welche Eigenschaften ein Tier besitzt und welche Funktionen es ausführen kann, steht allerdings noch kein Tier auf dem Spielfeld. Dazu muss erst eine **Instanz** dieser Klasse erzeugt werden. **Eine Instanz ist dann das eigentliche Objekt**, von dem beliebig viele andere erschaffen werden können. Instanzen können z.B. jetzt der Läufer *willi* und das Fresstier *karl* sein

Die OOP verwendet eine eigene Notation zur Darstellung von Klassen, die sog. **UML-Notation** (unified Modelling Language). Unser Beispiel kann danach wie folgt gezeichnet werden.



Attribute und Methoden

Die Eigenschaften einer Klasse werden auch deren **Attribute** genannt. Zu einer kompletten Klassendefinition gehören auch Angaben darüber, welche Werte diese Attribute annehmen können. Da Attribute nichts anderes sind als Variablen geschieht dies durch die Angabe des Datentypes (int, float, char ...)
Alle Instanzen besitzen die gleichen Attribute, allerdings in verschiedenen Ausprägungen (z.B. Farbe rot oder gelb).

Als **Methoden** werden die Funktionen bezeichnet, die in ein Objekt eingebunden sind. In den Methoden müssen die Funktionen der Klasse genau beschrieben werden. Für das Drehen etwa könnte man programmieren, dass eine einfache Richtungsänderung eine Drehung der Spielfigur um 10 Grad bedeutet. Durch Betätigen einer Pfeiltaste oder Ziehen mit der Maus dreht sich die Figur jeweils um diese Gradzahl. Alle Instanzen der gleichen Klasse benutzen dieselben Funktionen

Kapselung

Unter Kapselung versteht man, dass Attribute und Methoden in einem Objekt zusammengefasst werden und ihrem Umfeld nur als geschlossene Einheit zur Verfügung gestellt werden. Welches Attribut und welche Methode nach außen sichtbar sind, kann der Programmierer festlegen, indem er sie entweder als **private** oder **public** deklariert.
Alle Deklaration im private-Bereich sind **gekapselt**, d.h. sie können außerhalb der Klasse nicht angesprochen und verändert werden. Die Vereinbarungen im public-Bereich sind öffentlich, d.h. sie sind auch außerhalb der Klasse erreichbar. Die Datenstruktur soll ja von niemanden beeinflusst werden können, die Methoden aber sollen von unterschiedlichen Anwendungen benutzt werden können.

Vererbung

Vererbung ist eines der wichtigsten Prinzipien der OOP. Man versteht darunter, dass von bereits bestehenden Klassen neue Klassen abgeleitet werden können, die zunächst die gleichen Attribute und Methoden besitzen wie ihre Vorgänger. Sie können aber mit zusätzlichen Elementen ausgestattet werden. Diejenige Klasse, von der eine neue **Unterklasse** abgeleitet wird, nennt man **Oberklasse** oder **Basisklasse**.
Von der Basisklasse Tier haben wir beispielsweise eine Unterklasse Läufer und eine Unterklasse und eine Unterklasse Fresstier gebildet, die als zusätzlichen Attribute die Anzahl der Körner im Maul sowie als neue Methoden das Laufen mit Übergabewert und Fressen haben.

Polymorphie (Überladen von Funktionen)

Als Polymorphie bezeichnet man die Fähigkeit verschiedener Unterklassen ein und derselben Oberklasse, auf die gleiche Botschaft unterschiedlich zu reagieren. Das bedeutet, dass z. B. die Unterklasse LKW die gleiche Funktion *Beschleunigen* besitzt wie die Oberklasse Auto, die aber bei beiden unterschiedlich ausgeführt wird.
Dazu muss die vererbte Methode *Beschleunigen* der Oberklasse durch eine veränderte Methode *Beschleunigen* überschrieben werden.

Überladen

Unter dem Überladen einer Methode versteht man, dass es innerhalb einer Klasse mehrere Varianten der gleichen Methode geben kann. Diese haben den gleichen Namen, unterscheiden sich aber durch ihre Argumente (Übergabeparameter).
So verfügt die Unterklasse Läufer ebenfalls über eine Methode *Laufen.* Dieses ist jedoch eine überladene Methode, da sie einen Übergabeparameter hat. Durch Übergabe einer Zahl, z.B. 4 läuft das Objekt nicht nur jeweils ein Feld, sondern gleich 4 Felder.

Konstruktoren und Destruktoren

Konstruktoren sind spezielle Methoden, die zur Initialisierung der Attribute eines Objektes dienen. Wenn man keinen Konstruktor definiert, stellt der Compiler einen Defaultkonstruktor bereit. Ein Destruktor ist eine spezielle Methode, die den Speicherplatz nach dem ‚Ableben' eines Objektes wieder frei gibt.

Fragen

1. Welche zwei Arten von Informationen gehören zur Definition einer Klasse?

2. Wie bezeichnet man die einzelnen Objekte, die von einer Klasse gebildet werden.

3. Erläutern Sie den Begriff Kapselung.

4. Was bedeutet Vererbung in der objektorientierten Programmierung?

5. Nennen Sie jeweils drei Attribute und Methoden einer fiktiven Klasse *Flugzeug* und zeichnen Sie die Klassendefinition in der UML-Darstellung.

6. Erstellen Sie jeweils 2 Attribute und Methoden einer Basisklasse *Tier* und bilden sie davon zwei abgeleitete Klassen. Ergänzen Sie diese beiden abgeleiteten Klassen jeweils um eine zusätzliche Eigenschaft und Methode.

12 .Erstellen einer objektorientierten Anwendung

Wir erstellen eine Klasse Fahrzeug, die ein Auto repräsentieren soll.
Attribute: Farbe, kw, km_Stand
Methoden: getFarbe, setFarbe, getkw,setkw,getkm,setkm.

Den beiden Programmierern **Fahrer** und **Lackierer** wird diese Klasse zur Verfügung gestellt.

Der Fahrer hat die Aufgabe mit dem Auto zu fahren und den km_Stand zu erhöhen. Dies geschieht durch Angabe von Fahrtgeschwindigkeit und Fahrtzeit in Minuten.

Der Lackierer hat die Aufgabe, die Farbe des Autos zu ändern.

Die Klasse Fahrzeug

Klassendatei: **auto_Klasse.h**
Methodendatei: **auto_Klasse.cpp**

Anwendungsdateien: **Fahren.cpp**
Lackieren.cpp

Klassendatei

```
//Headerdatei Klassendefinition
#ifndef auto_methodenH
#define auto_methodenH
//---------------------------------------------------------------------------
class Fahrzeug
{
   private:
        string farbe;
        int kw;
        int km;
   public:
        Fahrzeug(string,int,int);                    //Konstruktor
        ~Fahrzeug();                                 //Destruktor
        string getfarbe(void);
        void setfarbe(string);
        int getkw(void);
        void setkw(int);
        int getkm(void);
        void setkm(int,int);
};
#endif
```

Methoden

```
//Methodendefinition
#include<iostream.h>
#include "auto_Klasse.h"

Fahrzeug::Fahrzeug(string f,int w,int m)
{
	farbe=f;
	kw=w;
	km=m;
}
Fahrzeug::~Fahrzeug()
{
}
int Fahrzeug::getkw(void)
{
	return kw;
}
void Fahrzeug::setkw(int k)
{
	kw=k;
}
int Fahrzeug::getkm(void)
{
	return km;
}
void Fahrzeug::setkm(int km_h,int t_min)
{
	km=km_h*t_min/60;
}
string Fahrzeug::getfarbe(void)
{
	return farbe;
}
void Fahrzeug::setfarbe(string f)
{
	farbe=f;
}
```

Anwendungen

1. Lackierer

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include <condefs.h>
#include "auto_Klasse.h"
USEUNIT("auto_Klasse.cpp");
//---------------------------------------------------------------------------
void main(void)
{
        string f;
        Fahrzeug Opel("blau",75,0);
        cout<<"Farbe bisher: "<<Opel.getfarbe()<<endl;
        cout<<"Neue Farbe: ";
        cin >>f;
        Opel.setfarbe(f);
        cout<<"\nFarbe jetzt: "<<Opel.getfarbe()<<endl;
        getch();
}
```

2. Fahrer

```
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <condefs.h>
#include "auto_Klasse.h"
USEUNIT("auto_Klasse.cpp");
//---------------------------------------------------------------------------
void main(void)
{
        int g,t;
        Fahrzeug Opel("blau",75,0);
        cout<<"Kilometerstand alt: "<<Opel.getkm()<<endl;
        cout<<"Geschwindigkeit: ";
        cin >>g;
        cout<<"\nFahrtzeit in Minuten: ";
        cin >>t;
        Opel.setkm(g,t);
        cout<<"\nKilometerstand neu: "<<Opel.getkm()<<endl;
        getch();
}
```

Aufgabe: Programmieren Sie eine Unit **Tuner** mit der Sie die kw-Zahl des Fahrzeuges verändern können. Orientieren Sie sich dabei an der Anwendung **Fahrer.**