1 Einleitung

Die Programmiersprache *C++* ist ein Nachfolger von *C*, die von Brian Kernighan und Dennis Ritchie entwickelt wurde. Sie beinhaltet den kompletten Sprachumfang von *C*. Sie ist jedoch um ein Klassenkonzept erweitert. 1979 begann Bjarne Stroustrup mit der Entwicklung der Sprache, welche 1998 der ISO/IEC Standard 14882 veröffentlicht wurde. Die Sprache bietet sowohl abstraktere Ebenen, wie das Arbeiten mit Klassen, als auch systemnahe Elemente, wie das direkte Anfordern von Speicher, Zeigerarithmethik und direkter Zugriff auf den Arbeitsspeicher. Sie beinhaltet den kompletten Sprachumfang von C. Das bedeutet es kann jedes C Programm auch mit einem C++ Compiler compiliert werden. Die erste Implementierung des C++ Compilers übersetzte den C++ Code in gewöhnlichen C Code, der dann kompiliert wurde.

Beginnen wir mit dem ersten Programm:

```
#include <iostream>
int main()
{
	std::cout << "Hallo Welt !";
	return 0;
}
```

Das Programm gibt lediglich *Hallo Welt !* auf dem Bildschirm aus. In der ersten Zeile wird die Bibliothek *iostream* eingebunden. Dadurch ist es erst möglich die Funktion *cout* aus Zeile 4 zu verwenden, welche dazu dient Werte aus dem Bildschirm auszugeben.
Die *main* Funktion (*int main()*) ist der Einstiegspunkt in da Programm. Die erste Anweisung im (also hier *cout* ist die erste Anweisung, die bei Programmstart ausgeführt wird. Die geschweiften Klammer {} dienen als Begrenzung. Die *main* Funktion beginnt somit ab der geschweiften Klammer und geht bis zur entsprechenden schließenden geschweiften Klammer.
Die Anweisung *return 0* bedeutet, dass das Programm mit Erfolg beendet ist, andere Werte bedeuten somit eine nicht erfolgreiche Beendigung.

1.1 Befehle in Maschinensprache

Auf unterster Ebene in der Befehlsverarbeitung eines Computers stehen Programmiersprachen mit Befehlen, die der Computer direkt versteht. Dies sind Befehle aus Einsen und Nullen, da der Computer nur Strom an, Strom aus

versteht. Aus abstrakter Sicht führt der Akkumulator, der innerhalb der CPU sitzt, die Befehle aus. Er kennt z.B. Befehle um zu rechnen, um Daten aus dem Speicher zu holen oder um Daten zu speichern. Die Befehle (Einadressbefehle)) bestehen aus zwei Einheiten: dem eigentlichen Befehl (die ersten k Stellen) und der Adresse der Speicherstelle auf die er angewendet werden soll (die Stellen $k+1$ bis n). Die Länge der Befehle ist unterschiedlich (16 bit, 32 bit, 64 bit, ...).

n	...	$k+1$	k	...	1

Eine Ebene höher in der Befehlshierarchie steht Assembler. Dies ist eine direkte Übersetzung der Maschinenbefehle in eine für den Menschen verständliche Form, z.B. gibt es den Befehl

$$\text{LDA, } \beta,$$

wobei β die Speicherstelle ist, die in den Akkumulator geladen werden soll. Der Befehl

$$\text{ADA, } \beta$$

addiert den Inhalt der Speicherzelle β auf den Inhalt des Akkumulators.

1.2 Compiler

Wie oben beschrieben, wird ein C++ Programm in einer Sprache geschrieben, die für den Menschen verständlich ist. Jedoch versteht der Computer nur eine Sprache mit den Elementen 0 und 1. Diese Sprache (Maschinensprache) besteht aus einem festgelegten Befehlssatz, der für jeden CPU Typ unterschiedlich ist. Für einen Menschen wäre es natürlich schwierig mit Befehlen aus Nullen und Einsen zu arbeiten. Deshalb war der erste Schritt jeden Befehl in Maschinensprache direkt in einen „verständlicheren“ Befehl (Mnenomic) zu übersetzen. Dies nennt man Assembler. Assembler ist abhängig von dem CPU Typ. Ein Programm, das aus einer Folge von diesen Befehlen besteht kann dann direkt in ein lauffähiges Programm aus Maschinenbefehle übersetzt werden.
C++ befindet sich noch einige Abstraktionsstufen über Assemblerprogrammierung. Hier gibt es Befehle, die aus vielen einzelnen Maschinenbefehlen bestehen.
Der Quellcode eines C++ Programms muss vor der Ausführung in Maschinensprache übersetzt werden.

Beispielsweise wird ein Programm in *C++* Quellcode durch den *g++* Compiler in Maschinensprache übersetzt.

2 Grundlegende Programmierelemente

2.1 Variablendeklarationen

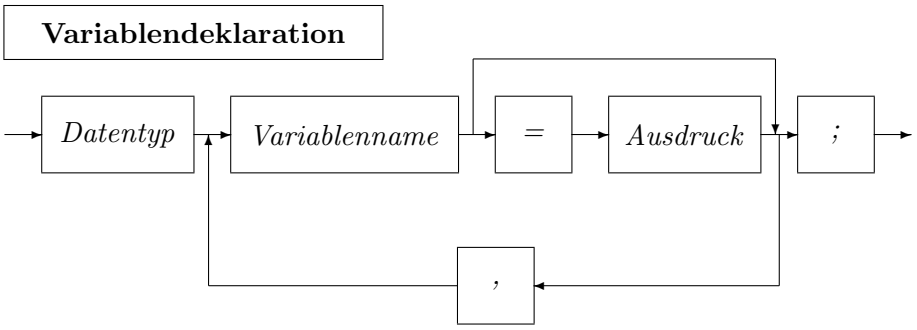
Um Werte zwischenzuspeichern, gibt es in Programmiersprachen Variablen. Hierdurch können beispielsweise Zwischenergebnisse gesichert werden oder Zähler programmiert werden.
Eine Variable reserviert ein Stückchen Arbeitsspeicher. Dieses Stückchen Arbeitspeicher ist nur für diese Variable reserviert. Es ist jedoch flüchtig. Das bedeutet, wenn der Computer ausgeschaltet wird oder das Programm beendet wird, ist der Wert nicht mehr zugänglich, im Gegensatz dazu, wenn man Werte auf der Festplatte speichert.
Jede Variable muss vor ihrem ersten Einsatz deklariert werden. Durch die Deklaration wird über das Betriebssystem das Stück Arbeitsspeicher reserviert und die Adresse, wo sich das Stück Arbeitsspeicher befindet, wird zurückgegeben.
Beispiele für Variablendeklarationen sind:

```
int x;
int y=5;
int a,b=4;
int z=x;
```

Jede Deklaration besteht aus einem Datentyp (hier *int*) und einem Bezeichner (hier x oder a,b,y). Danach kann optional noch eine Wertzuweisung erfolgen. Variablen vom Typ Integer erlauben es ganze Zahlen abzuspeichern. Dieser Typ wird in der Programmierung sehr häufig verwendet. Es

ist allerdings nicht möglich Kommazahlen oder gar Buchstaben in solch einer Variablen abzuspeichern.

Die genaue Syntax der Variablendeklaration zeigt das folgende Syntaxdiagramm:



Ganze Zahlen, Typdefinition:

Typ	**Spezifikation**
int	int, signed, signed int
unsigned int	unsigned int, unsigned
short int	short int,short,signed int,signed short int
unsigned short int	unsigned short int,unsigned short
long int	long int,long,signed long,signed long int
unsigned long int	unsigned long int,unsigned long

Ganze Zahlen, Wertebereiche:

Typ	**Wertebereich**
int x;	-2147483648 bis 2147483647
signed int x	-2147483648 bis 2147483647
unsigned int	0 bis 4294967296
short x	-32768 bis 32767
signed short x	-32768 bis 32767
long x	-2^{63} bis $2^{63}-1$
unsigned long x	0 bis $2^{64}-1$

Die Größe von *short*, *int* und *long* ist nicht allgemein spezifiziert. *int* sollte normalerweise die Größe eines Maschinenworts haben und ist mindestens genauso groß wie *short* und *long* ist mindestens so groß wie *int*.

Werte Einlesen kann man durch die Funktion *cin*. Durch sie wird ein beliebiger Datenstrom der Tastatur in einen Wert des gerade notwendigen Datentyps übersetzt.

```
int x; // Deklaration von x
std::cout << "Bitte geben Sie eine Integerzahl ein";
// Aufforderung an den Benutzer
std::cin >> x; // Einlesen des Wertes
long y;
std::cout << "Bitte geben Sie eine Longzahl ein";
std::cin >> y;
std::cout << y; // Ausgabe des Wertes von y
```

Der Aufruf von *cout* bedeutet, wie schon weiter oben beschrieben, eine Ausgabe auf dem Bildschirm. Die Pfeile << zeigen in Richtung von *cout*, da etwas in den Ausgabestrom geschrieben werden soll. Bei *cin* zeigen die Pfeile >> zu der Variablen, da etwas aus dem Eingabestrom in die Variable geschrieben wird.
Wenn der Benutzer einen Buchstaben oder mehrere Buchstaben eingibt, obwohl eine Integerzahl verlangt wurde, dann wird bei Variablen für Zahlen die 0 belegt.

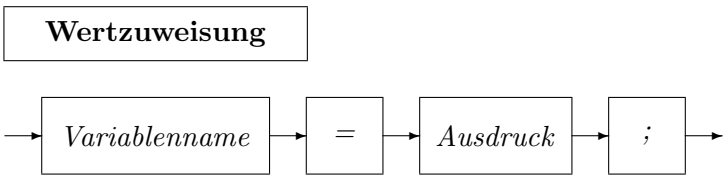
```
int x; // Deklaration von x
std::cout << "Bitte geben Sie eine Integerzahl ein";
// Aufforderung an den Benutzer
std::cin >> x; // der Benutzer gibt ein a ein
std:: cout << x; // 0 wird ausgegeben
```

Im Allgemeinen wird ein Standardwert gesetzt.

Variablen, die innerhalb einer Funktion (also erstmal der *main* Funktion) deklariert werden, nennt man lokale Variablen. Diese sind ab ihrer Deklaration bis zum Ende der Funktion verfügbar. Außerhalb der Funktionen können ihnen keine Variablen zugewiesen werden. Der Bereich in dem eine Variable verfügbar ist, nennt man Gültigkeitsbereich.

Die Variable kann beliebig oft neu mit Werten belegt werden, solange man sich im Gültigkeitsbereich befindet.

Die Wertzuweisung erfolgt über den Variablennamen, einem Gleichheitszeichen, dem Wert der zugewiesen wird und einem Semikolon:



Für die direkte Zuweisung von Zahlen, gibt es in C++ die Möglichkeit Dezimalzahlen, Oktalzahlen oder Hexdezimalzahlen zu definieren.
Oktalzahlen beginnen dabei immer mit einer führenden 0, Hexadezimalzahlen beginnen mit 0x und Dezimalzahlen werden, wie gewöhnlich definiert. Die Oktaldarstellung un die Hexadezimaldarstellung ergeben eigentlich nur Sinn, wenn man große Zahlen darstellen möchte. Hierbei ist die Definition von Dezimalzahlen sehr umständlich:

```
long x = 0xFFFFFFFF;
long y = 4294967296;
```

Ein weiterer Vorteil ist, dass Speicheradressen auch häufig in Hexadezimalzahlen definiert werden.

Eine genauere Definition zeigen folgende Syntaxregeln:

integer-literal $\longrightarrow$ *decimal-literal integer-suffix*?
integer-literal $\longrightarrow$ *octal-literal integer-suffix*?
integer-literal $\longrightarrow$ *hexadecimal-literal integer-suffix*?

integer-suffix $\longrightarrow$ *unsigned-suffix long-suffix*?
integer-suffix $\longrightarrow$ *long-suffix unsigned-suffix*?
unsigned-suffix $\longrightarrow u \mid U$
long-suffix $\longrightarrow l \mid L$

decimal-literal$\longrightarrow$*nonzero-digit*|*decimal-literal digit*
digit$\longrightarrow$0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
nonzero-digit$\longrightarrow$1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
octal-literal$\longrightarrow$0 | *octal-literal octal-digit*
octal-digit$\longrightarrow$0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

hexadecimal-literal$\longrightarrow$*0x hexadecimal-digit*|*0X hexadecimal-digit* |*hexadecimal-literal hexadecimal-digit*
hexadecimal-digit$\longrightarrow$0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | *a* | *b* | *c* | *d* | *e* | *f* | *A* | *B* | *C* | *D* | *E* | *F*

Mit Variablen kann ganz normal gerechnet werden:

```
int x = 7;
int z = 3 + x; // z = 10
int test = z * x; // test = 70
z = z + 1; // z = 11
```

Für die Rechnung mit ganzen Zahlen stehen unter anderem folgende Operatoren zur Verfügung:

Operator	**Name**	**Beispiel**
++	Postinkrement	x++
++	Präinkrement	++x
−−	Postdekrement	x−−
−−	Prädekrement	−−x
+	unäres Plus	+x
-	unärers Minus	-x
%	Modulo (Rest)	x%*y*, *7%3=1*
∼	bitweises Komplement	x=3; x=∼x; // -4
*,/,+,-		

In C++ gibt es noch weitere bitweise Operationen:

```
int x = 4, y = 1;
int erg;
erg = x & y;            /* liefert erg = 0
                           ein Bit von erg ist nur dann auf 1 gesetzt,
                           wenn die entsprechenden Bits von x und y
                           auf 1 gesetzt sind */
erg = x | y;            /* liefert erg = 5
                           ein Bit von erg ist dann auf 1 gesetzt,
```

```
                              wenn mindestens eines der entsprechenden
                              Bits auf 1 gesetzt ist */
erg = x ^ y;                /* liefert erg = 5
                              ein Bit von erg ist dann auf 1 gesetzt,
                              wenn die entsprechenden Bits von x und y
                              verschiedene Werte besitzen */
erg = x << y;               /* liefert erg = 8
                              verschiebt die Bits von x um y Stellen
                              nach links. Es wird mit Nullen
                              aufgefuellt*/
y = 2;
erg = x >> y;               /* erg=1;
                              verschiebt die Bits von x um y Stellen
                              nach rechts. Es wird mit Nullen
                              aufgefuellt */
```

Darüber hinaus gibt es noch den Datentyp *bool*. Dieser Datentyp besteht aus einem einzigen Bit. Das bedeutet für eine Variable dieses Typs wird genau eine Speicherstelle reserviert, in die 0 oder 1 beschrieben werden kann. In den gängigen Programmiersprachen wird darin ein sogenannter Wahrheitswert abgelegt, also *true* oder *false*.

```
bool a;
bool a = true;
bool a,b = true;
bool b = false;
bool a=1; // a wird mit true belegt
```

Zum Arbeiten mit logischen Werten gibt es die logischen Operatoren, dazu belegen wir:

bool x = true;
bool y = false;

!	Logisches Nicht	$!x$ // false
&&	Logisches Und	$x\&\&y$ // false
||	Logisches Oder	$x\|\|y$ // true
==	Gleichheitstest	$x == y$ // false
!=	Ungleichheitstest	$x! = y$ // true

Relationale Operatoren liefern auch einen booleschen Wert.
int x = 4;

int y = 3;

<	kleiner	$x < y$ // false
<=	kleiner gleich	$x <= y$ // false
>	größer	$x > y$ // true
>=	größer gleich	$x >= y$ // true

Auch wenn dieser Datentyp trivial erscheint, man wird ihn bei weiterführenden Konzepten sehr gut gebrauchen.

Als nächstes gibt es noch den Dateyp *char*. In diesen können Zeichen und somit auch einzelne Buchstaben gespeichert werden. Einzelne Zeichen werden immer in einfache Anführungszeichen *'a'* gepackt.

```
char a = 'a';
a = 'A';
a = '/';
```

2.1.1 Der Arbeitsspeicher

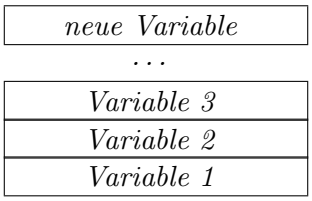
Jedes Programm bekommt bei seiner Ausführung ein Stück Arbeitspeicher zugewiesen. Dieses Stück Arbeitsspeicher wird vom Betriebssystem angefordert. Dieses Stück besteht aus fünf Teilen:

1. Globaler Speicher
2. Stack
3. Heap
4. Konstanter Speicher
5. Programmspeicher

Im Programmspeicher werden die Befehle des Programms abgelegt. Diese Befehle liegen bei einem kompilierten Programm in Maschinencode vor.
Der globale Speicher enthält alle globalen Variablen und alle statischen Variablen. Hier befinden sich Variablen, die von Programmstart bis Programmende verfügbar sind. Der Speicher ist statisch und verändert seine Größe somit nicht.
Im Stack befinden sich alle lokalen Variablen, die in Methoden definiert worden sind und alle Variablen, die als Übergabeparameter definiert sind. Der Speicherplatz für diese Variablen wird erst reserviert, wenn sich der Programmablauf an der Stelle der Variablendeklaration befindet. Die deutsche

Bedeutung von Stack ist Stapel. Dieser Teil des Arbeitsspeichers wird so genannt, weil jede neue Variable im Speicher direkt hinter der zuletzt definierten Variablen im Speicher angelegt wird. Quasi wird die neue Variable immer auf den Stapel gelegt.

Stack im Arbeitspeicher



Wenn ein Gültigkeitsbereich (Block, durch geschweifte Klammern gekennzeichnet) verlassen wird, werden alle Speicherstellen der dort definierten Variablen wieder freigegeben. Und zwar wird der Speicherplatz der zuletzt definierten Variable zuerst wieder freigegeben. Dieses Prinzip findet sich im Stapel wieder. Es kann nur die oberste, also die zuletzt definerte, Variable wieder entfernt werden.
Der Speicher, den eine Variable, die im Stack definiert worden ist, wird beim Verlassen des Gültigkeitsbereich freigegeben. Der Heap Speicher und der Konstantenspeicher wird später erläutert.

2.1.2 Ablauf der variablendeklaration im Arbeitspeicher

Schauen wir uns den Ablauf der Variablendeklaration lokaler Variablen im Hauptspeicher an. Durch die Variablendeklaration wird Speicher im Arbeitsspeicher angefordert. Die Speicherstelle für die Variable ist so groß, wie der Datentyp der Variable angibt (für *int* 32 bit oder 16 bit).
Jedes Programm hat bestimmte Bereiche im Hauptspeicher, in denen es Speicher anfordern kann und auf den kein anderes Programm zugreifen kann. Nachdem der Speicher angefordert worden ist, wird der Variablenwert an einer freien Stelle im Speicher abgespeichert und diese Adresse wird mit dem Variablennamen verknüpft. Wenn sich der Wert der Variablen ändert, bleibt der Speicherplatz bei den grundlegenden Variablentypen der gleiche. Die Belegung des Speicherplatzes ändert sich.

Adresse	Speicherstelle	Programmcode
0		
1	00000011	uint x = 3;
2	...	
	...	
8	1	bool y = true;
9	...	
	...	

Im Beispiel wird zuerst eine Variable vom Typ *uint* im Speicher reserviert und dann der Wert 3 darin abgespeichert. Die 3 wir in 8-Bit Binärcode (jede Zahl hat acht Stellen) durch 00000011 dargestellt.

2.2 Blöcke

In C++ können an beliebigen Stellen Blöcke definiert werden. Diese beschreiben dann einen eigenen Gültigkeitsbereich, was bedeutet, dass alle Variablen, die in diesem deklariert werden nur in diesem Block gültig sind.

```
int main()
{
  {
   int x;
   int y=4;
  }
  x=3; // Fehler
}
```

Im nächsten Beispiel wird deutlich, dass man dadurch Variablen mit gleichem Namen innerhalb von eines Blocks mittels Unterblöcken definieren kann.

```
{
   int x=8;
   int y=3;
   {
      int x=9;
      int y=14;
   }
}
```

Hierbei wird die Variable x zuerst mit 8 belegt, vorher deklariert und dann nochmal erneut deklariert und mit 9 belegt.

2.3 Bedingungen

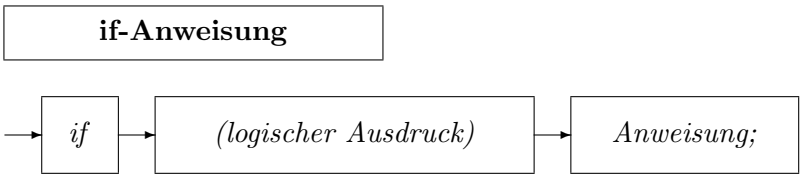
Die bisher kennen gelernten Anweisungen ergaben einen sequenziellen Programmablauf. Hierbei arbeitet der Computer eine Anweisung nach der anderen ab.

Problem: Wir wollen ein Programm schreiben, das eine Integerzahl x vom Benutzer einliest, diese um eins erhöht und ausgibt, falls sie ungerade ist, anderfalls soll sie direkt ausgegeben werden.

Für dieses Problem kennen wir bisher keine implementierbare Lösung. Es wird ein neues Programmierelement benötigt.

Die *if*-Bedingung wertet einen Ausdruck aus. Ist das Ergebnis *true*, wird der sequentielle Ablauf beibehalten. Ist das Ergebnis *false*, wird der sequentielle Ablauf nicht beibehalten und die Anweisung hinter der Bedingung übersprungen.

Eine *if*-Anweisung besteht aus dem Schlüsselwort *if*, einem beliebigen logischen eingeklammerten Ausdruck, der nach *true* oder *false* auswertet wird und einer Anweisung, die im *true*-Fall ausgeführt wird.

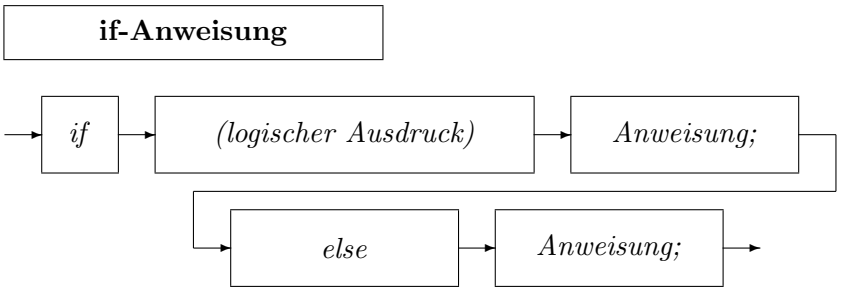



```
int main()
{
            cout << "Bitte geben Sie eine Zahl x ein:";
            cin >> x;
            if (x%2==1)
                x--;
            cout << x;
            return 0;
}
```

Wenn mehr als eine Anweisung ausgeführt werden soll, müssen diese Anweisungen in einen Block zusammengefasst werden. Dieser besteht aus geschweiften Klammern ({ Anweisungen }).

```
int main
{
            cout << "Bitte geben Sie eine Zahl x ein:";
            cin >> x;
            if (x<3)
            {
                 cout << x<<"x ist kleiner 3";
            }
            if (x>3)
            {
                 cout << x<<"x ist groesser 3";
            }
            if (x==3)
            {
                 cout<<x<<"x ist gleich 3";
            }
}
```

Als Erweiterung der *if*-Bedingung gibt es die *if,else*-Bedingung. Die *if,else*-Bedingung wertet einen Ausdruck aus. Ist das Ergebnis *true*, werden die Anweisungen hinter dem *if* ausgeführt und die Anweisungen hinter dem *else* übersprungen. Ist das Ergebnis *false*, werden die Anweisungen hinter dem *if* übersprungen und die Anweisungen hinter dem *else* ausgeführt.



```
int main()
{
            cout<<"Bitte geben Sie eine Zahl x ein:";
            cin >> y;
            if (y==3)
            {
                 cout << "x ist gleich 3";
            }
            else
            {
                 cout<<"x ist nicht gleich 3";
            }
}
```

Übungen:

1. Schreiben Sie ein Programm, das eine Zahl einliest und dann bestimmt, ob die Zahl gerade, bzw. ungerade ist.

2. Schreiben Sie ein Programm, das eine Zahl x vom Benutzer einliest und den Betrag der Zahl ausgibt.

3. Schreiben Sie ein Programm, das vom Benutzer ein Zeichen einliest und *Vokal* ausgibt, falls das Zeichen ein Vokal ist, andernfalls *Konsonant*.

4. Schreiben Sie ein Programm, das drei Zahlen x, y, z vom Benutzer einliest. Danach soll die größte Zahl mit der kleinsten multipliziert werden und durch die mittlere geteilt werden.

5. Schreiben Sie ein Programm, das vom Benutzer drei Zeichen einliest und sie lexikografisch ordnet.

6. Schreiben Sie ein Programm zur Tierbestimmung: Der Anwender soll eine oder beide der folgenden Fragen beantworten:

   - Welche Schulterhöhe hat das Tier?
   - Kann das Tier bellen?

   Je nach Antwort(en) soll das Programm Hund, Katze oder Maus ausgeben.

7. Die folgende Tabelle zeigt die Eintrittspreise für ein öffentliches Schwimmbad:

	1,5 Stunden	3 Stunden	1 Tag
1 Erwachsener	5,00 Euro	5,80 Euro	8,00 Euro
1 Kind	2,50 Euro	2,90 Euro	4,00 Euro

   Zuschlag für Therme pro Person 2,00 Euro

   Gruppenermäßigung ab 4 Personen: 20%

   Schreiben Sie ein Programm, das einliest, wie viele Personen baden wollen (Erwachsene und Kinder), wie lange sie baden wollen und ob auch Zugang zur Therme gewünscht ist. (Alle Personen müssen sich bzgl. der Therme und der Badedauer identisch entscheiden!)
   Das Programm soll den insgesamt resultierenden Eintrittspreis ausgeben.

8. Schreiben Sie ein Programm, das eine Temperatur einliest und ausgibt, ob Wasser dieser Temperatur bei Normaldruck fest, flüssig oder gasförmig ist. Berücksichtigen Sie außer diesen drei Fällen auch die Phasenübergänge (bei 0 und 100 Grad liegen jeweils 2 Phasen im Gleichgewicht vor).

9. Schreiben Sie ein Programm, das von dem Benutzer drei Integerzahlen einliest. Es soll ausgegeben werden, ob diese drei Zahlen die Seitenlängen eines Dreiecks sein können.

2.4 Schleifen

Problem: Wir wollen ein Programm schreiben, das uns die Summe von 1 bis 100 berechnet.

Bishergie Lösung: Wir definieren uns eine Integervariable *int x=0;* und schreiben 100 Programmzeilen:

```
x = x + 1;
x = x + 2;
...
x = x + 100;
```

⇒ Der Programmieraufwand ist ernom.

Nun verändern wir das Problem und stellen die Forderung, dass der Benutzer eingeben soll, bis zu welchem Wert die Summe berechnet werden soll.

Nun funktioniert unsere Lösung nicht mehr, da der Programmierer im Vorfeld nicht weiß, welche Zahl der Benutzer eingeben wird. Darüber hinaus kann der Benutzer beim Neustart des Programms eine ganz andere Zahl wählen.

Um das Problem zu lösen, langen unsere bisherigen Programmiertechniken nicht aus.

Für diesen Fall existieren in den modernen Programmiersprachen Schleifen. Mittels Schleifen lassen sich Anweisungen wiederholen. Dabei muss nicht immer eine konkrete Anweisungsfolge wiederholt werden, sondern es können durch Variablen und Bedingungen in jedem Durchlauf neue Vorbedingungen geschaffen werden.

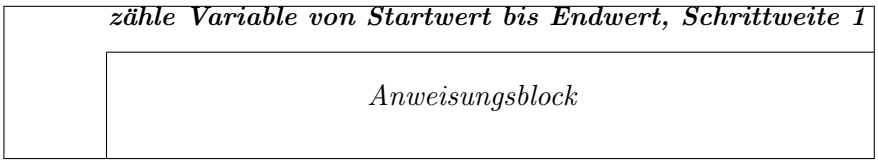
Es gibt drei verschiedene Sorten von Schleifen:

1. **for**-Schleife
2. **while**-Schleife
3. **do while**-Schleife

2.4.1 Die for-Schleife

Die *for*-Schleife ist eine zählergesteuerte Schleife. Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Zur Ablauf-

steuerung wird am Anfang eine Zählvariable initialisiert und dann bei jedem Durchlauf aktualisiert. Die zur Schleife gehörige Anweisung wird ausgeführt, bis die Zählvariable einen festgelegten Endwert erreicht.

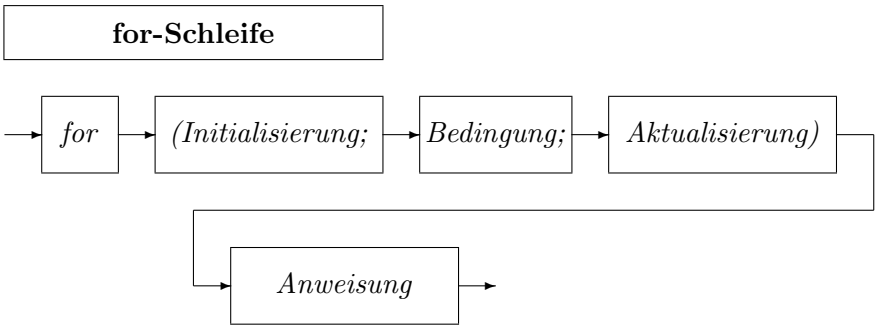


Die *for*-Schleife wird mit dem Schlüsselwort *for* eingeleitet. Darauf folgt eine runde Klammer, dann die drei wichtigen Teile Initialisierung, Bedingung, Aktualisierung. Danach wird die runde Klammer geschlossen. Dieser Teil wird als *Schleifenkopf* bezeichnet. Hinter dem Schleifenkopf wird eine Anweisung aufgeführt, die wiederholt werden soll.

1. Die *for*-Schleife benötigt, wie schon oben beschrieben, eine Zählvariable. Sie ist für gewöhnlich vom Typ Integer. Sie kann aber von einem beliebigen numerischen Typ sein. In der Initialisierung wird sie mit einem Startwert belegt, zum Beispiel *int i = 0;*.

2. Es folgt die Bedingung (logischer Ausdruck), die in der Regel für die Zählvariable aufgestellt wird und meistens den Endwert der Zählvariablen enthält, zum Beispiel *i <= 5;*.

3. Als letztes Element des Schleifenkopfs wird die Aktualisierung angegeben. Hier wird festgelegt, wie die Zählvariable in jedem Schleifendurchlauf verändert wird, zum Beispiel *i++*.



Die **for**-Schleife wiederholt nur eine Anweisung. Das folgende Beispiele zeigt die Berechnung der Summe von 1 bis 100.

```
int sum = 0;
for (int i = 1; i <= 100; i++)
   sum = sum + i;
cout << sum;
```

Wenn mehrere Anweisungen durch die Schleife wiederholt werden sollen, müssen die Anweisungen genau wie bei der *if*-Bedingung in einen Block gepackt werden.

Das nächste Beispiel berechnet die Summe von 1 bis zu einem Wert x, welcher der Methode übergeben wird. Hierbei ist die Bedingung mit einer Variablen eingegrenzt. Innerhalb des Schleifenbauchs (der Anweisungsteil einer Schleife) wird die Zählvariable verwendet, um die Summe auf zu addieren. Da zwei Anweisungen in jedem Schleifendurchlauf ausgeführt werden sollen, werden die Anweisungen in einen Block gepackt.

```
int x;
cin >> x;
int sum = 0;
for (int i = 1; i <= x; i++)
{
   sum = sum + i;
   cout<<"Zwischenergebnis:\n" << sum;
}
cout << sum;
```

Im folgenden Beispiel werden 10 aufeinanderfolgende natürliche Zahlen aufsummiert, wobei die erste Zahl der Methode übergeben wird. Hierbei wird die Zählvariable durch eine Variable initialisiert.

```
int y;
cin >> y;
int sum = 0;
for (int i = y; i <= y+10; i++)
{
   sum = sum + i;
}
cout << sum;
```

Übungen:

1. Analyse von Methoden

   (a) Was berechnen die folgenden Methoden in Bezug auf die in Kommentar geschriebenen Werte?

   (b) Was berechnen die folgenden Methoden in Abhängigkeit von ihren Übergabeparametern?

   (c) Wieviele Schleifendurchläufe durchlaufen die einzelnen Schleifen in Abhängigkeit von den Übergabeparametern?

   Methode A b) x+y

```
int x,y;
cin >> x;  // 3
cin >> y;  // 4
for (int i = 0; i < y; i++) c) y Durchläufe
{
   x++;
}
cout << x; a) 7
```

   Methode B

```
int x;
cin >> x; // 3 a)b)c) Überlauf?
for (int i = 0; i < x; i++)
{
   x=x*x;
}
cout << x;
```

Methode C

```
int x;      x^x
cin >> x; // 4
int erg=1;
for (int i = 0; i < x; i++) x Durchläufe
{
   erg=erg*x;
}
cout << erg;  4^4
```

Methode D

```
int x,y;
cin >> x;
cin >> y;
int erg=0;int z = 1;
for (int i = 0; i < y; i++)    y Durchläufe
{
   erg = 0;
   for (int j = 0; j < x; j++)   x Durchläufe
   {
      for (int k = 0; k < z; k++) z=x^(i-1) Durchläufe
      {
         erg++;    erg=z
      }
   }  erg=z*x
   z = erg; z=x^i
}
cout << erg;   x^y
```

2. Implementation von Programmen und Methoden

   (a) Schreiben Sie ein Programm, das eine Zeichenkette einliest und sie umgekehrt ausgibt.

   (b) Schreiben Sie ein Programm, das eine Zeichenkette einliest und überprüft, ob diese ein Palindrome ist.

   (c) Eine Bäckerei verkauft am Sonntag nur eine Sorte Brötchen, die 0,35 Euro pro Stück kosten.

- Schreiben Sie ein Programm, das abfragt, wie viele Brötchen ein Kunde gekauft hat und ausgibt, wie viel der Kunde zahlen muss.
- Ergänzen Sie das Programm so, dass es nach der Brötchenanzahl auch noch fragt, ob die Brötchen in eine Geschenktüte gelegt werden sollen. Falls ja, muss der Kunde einen Aufpreis von 0,50 Euro zahlen.
- Erweitern Sie das Programm nun so, dass der Preis mehrmals hintereinander berechnet werden kann. Zum Abbruch des Programms soll z.B. das Zeichen E (oder e) für Programm-Ende eingegeben werden.

(d) Schreiben Sie eine Methode, die Primzahlzwillinge berechnet.

(e) Schreiben Sie eine Methode, die die Fibonaccifolge berechnet.

(f) Schreiben Sie ein Programm, das binäre Zahlen in positive ganze Dezimalzahlen umwandelt.

(g) Schreiben Sie ein Programm, das positive ganze Zahlen in binäre Zahlen umwandelt.

(h) Schreiben Sie ein Programm, das für die Funktion

$$y = 3x^2 - 2x + 5$$

eine Wertetabelle ausgibt. Der Anwender soll Start- und Endwert für die x-Werte sowie die Schrittweite eingeben.

(i) Schreiben Sie ein Programm, das für zwei vorgegebene natürliche Zahlen den größten gemeinsamen Teiler (GGT) dieser Zahlen mit Hilfe des von Euklid beschriebenen Algorithmus bestimmt.

(j) Schreiben Sie ein Programm, das zu einer anzugebenden Zahl n alle Quadratzahlen der Form $1 + 4 + 9 = 14$ von 1 bis n berechnet, diese aufsummiert und ausgibt.

(k) Schreiben Sie ein Programm, dass eine Zahl x einliest und dann

- ein Quadrat aus Sternen mit der Seitenlänge x ausgibt.
- ein rechtwinkliges Dreieck mit der Höhe x ausgibt.
- ein gleichschenkliches Dreieck mit der Höhe x ausgibt.

Beispiel für $x = 4$

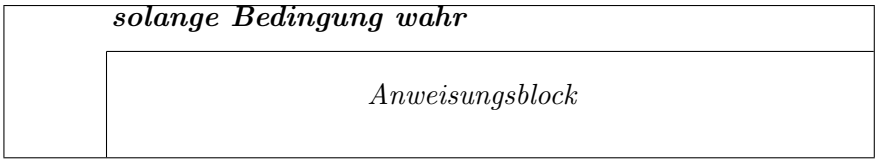
```
****   *        *
****   **      ***
****   ***    *****
****   ****  *******
```

2.4.2 Die while-Schleife

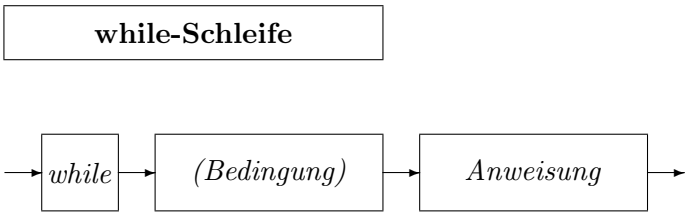
Die **while**-Schleife wird über eine Bedingung gesteuert. Bei jedem Schleifendurchgang wird eine Bedingung überprüft, das Ergebnis entscheidet über das weitere Vorgehen:

- **true**: die zur Schleife gehörige Anweisung wird ein weiteres Mal ausgeführt
- **false**: Schleife beenden

Bei der kopfgesteuerten **while**-Schleife wird die Bedingung vor Beginn eines Durchgangs geprüft. Die Bedingung kann dabei ein beliebiger logischer Ausdruck sein (der Ausdruck muss somit nach *true* oder *false* ausgewertet werden).



Die **while**-Schleife wird mit dem Schlüsselwort **while** eingeleitet. Danach folgt in runden Klammer eingeschlossen die Bedingung. Nach der Runden Klammer wird dann die Anweisung aufgeführt.



Im folgenden Beispiel wird der Benutzer gefragt, ob eine Schleife gestartet werden soll. Ist dies der Fall wird die *while* Schleife durchlaufen, *Hello World* ausgegeben und gefragt, ob die Schleife beendet werden soll. Andernfalls wird das Programm beendet.

```
char c = 'n';
cout << "Moechten Sie die Schleife starten? j/n";
```

```
cin >> c;
while (c=='n')
{
     cout << "Hello World!";
     cout << "Moechten Sie die Schleife beenden? j/n";
     cin >> c;
}
```

Übungen (es können außerdem alle Übungen der *for* und der *do while* Schleife durchgeführt werden):

1. Schreiben Sie ein Programm, dass Zufallszahlen zwischen 0 und 6 aufsummiert und jeweils ausgibt. Falls die ermittelte Zufallszahl eine Null ist, wird die Schleife abgebrochen.

2. Schreiben Sie ein Programm, welches die Ackermann Funktion iterativ berechnet.

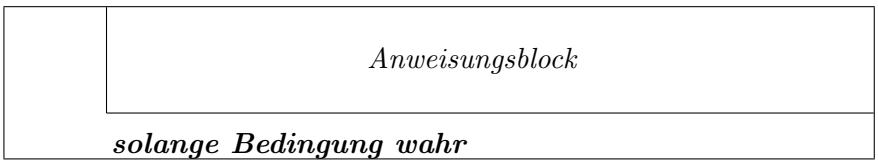
3. Schreiben Sie ein Programm, das den GGT nach dem Algorithmus von Euklid berechnet.

4. Schreiben Sie ein Programm für folgendes Ratespiel: Ein Anwender gibt eine Zahl ein. Dann muß ein anderer Anwender solange Zahlen eingeben, bis er die richtige Zahl gefunden hat. Das Programm soll nach jeder Eingabe ausgeben, ob die eingegebene Zahl kleiner oder größer als die zu erratende Zahl ist. Zum Schluß soll die Anzahl der Raterunden ausgegeben werden.

2.4.3 Die do-while-Schleife

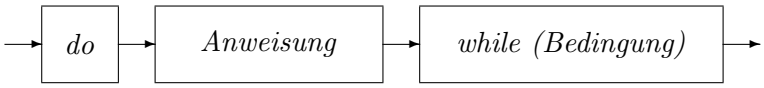
Die **do-while** Schleife wird genauso wie die **while**-Schleife über eine Bedingung gesteuert. Die Bedingung ist natürlich auch ein logischer Ausdruck. Die Schleife ist jedoch fußgesteuert. Das bedeutet, dass erst am Ende jedes Durchlaufs der Anweisungen überprüft wird, ob die Bedingung noch gilt. Im Fall, dass die Bedingung noch gilt, werden die Anweisungen der Schleife wiederholt, andernfalls wird sie beendet (genau wie die **while**-Schleife).

Der entscheidente Unterschied zur *while* Schleife ist, dass die Anweisungen auf jeden Fall einmal durchlaufen werden.



Die **do-while** Schleife wird durch das Schlüsselwort **do** eingeleitet. Darauf folgt die Anweisung, die wieder holt wird und das Schlüsselwort **while**, gefolgt von der in runden Klammern umschlossenen Bedingung.

do-while-Schleife



Das folgende Beispiel zeigt eine *do-while* Schleife, die solange *Hello World* und eine Frage ausgibt, bis der Benutzer ein *n* eingibt. Dabei werden auf jeden Fall einmal die beiden Zeilen ausgegeben.

```
char c = 'n';
do
{
     cout << "Hello World!";
     cout << "Moechten Sie die Schleife beenden? j/n";
     cin >> c;
}
while (c=='n');
```

Übungen (es können außerdem alle Übungen der *for* und der *while* Schleife durchgeführt werden):

1. Schreiben Sie eine Funktion, die vom Benutzer solange Zahlen einliest, bis dieser eine Zahl eingibt, die durch 3 ohne Rest geteilt werden kann. Berechnen Sie den Durchschnitt der eingegebenen Zahlen.

2. Schreiben Sie ein Programm, das die Quersumme einer ganzen Zahl berechnet.

3. Schreiben Sie ein Programm, das durch ein Menü gesteuert wird. Folgende Eingaben sind möglich:
   - ‘a‘ - der Benutzer wird dazu aufgefordert zwei Zahlen einzugeben, danach werden diese beiden summiert und ausgegeben. Selbiges gilt für ‘m‘ - multilpizieren, ‘d‘ - dividieren und ‘s‘ - subtrahieren
   - ‘q‘ - das Programm wird beendet.
4. Schreiben Sie eine Methode, die berechnet, ob eine Zahl abundant ist.
5. Schreiben Sie eine Methode, die berechnet, ob eine Zahl vollkommen ist.

Übungen:

- Schreiben Sie ein Programm, das eine Lottoziehung simuliert.
- Schreiben Sie ein Programm, das einen Cola Automaten simuliert. Dabei soll dieser Automat das Geld wechseln und nach jeder eingeworfenen Münze anzeigen wie viel Geld noch benötigt wird.
- Schreiben Sie ein Programm, welches ein Spiel mit folgenden Vorgaben, die schrittweise nacheinander programmiert werden können, simuliert:
  - Es existiert eine Figur A in der Mitte des Bildschirms, die bewegt werden kann.
  - Es existiert eine weitere Figur B. Beide können sich bewegen. Wenn beide aufeinandertreffen hat Figur A verloren.
  - Die Figur B wird vom Computer gesteuert.
  - Es gibt ein Spielfeld mit Wänden, die nicht überquert werden können.
  - Es gibt Sachen zum Einsammeln (Bonusleben, „Schneller Laufen“, ...).

2.5 Funktionen

Problem: Wir wollen ein Programm schreiben, in dem mehrere Male zwei Zahlen addiert, multipliziert und danach beide Ergebnisse ausgegeben werden.

Bisherige Lösung: Sowohl Berechnung als auch Ausgabe müssen mehrere Male definiert werden:

```
...
a = y + z;
b = y*z;
cout << y << " plus " << z << " ergibt " << a;
cout << y << " mal " << z << " ergibt " << b;
...
a = r + s;
b = r*s;
cout << r << " plus " << s << " ergibt " << a;
cout << r << " mal " << s << " ergibt " << b;
...
```

⇒ Der Programmieraufwand ist hoch und die Programme werden unübersichtlich.

Um diese Probleme lösen zu können, existieren in den modernen Programmiersprachen Funktionen oder ähnliche Konzepte wie Methoden oder Prozeduren. In C++ können Funktionen definiert werden. In ihnen kann man Anweisungen zusammenfassen und sie dann bei Bedarf aufrufen. Die Funktionen können mit Werten (Parameter) aufgerufen werden und sie können ein Ergebnis liefern. Zum Beispiel könnte eine Funktion *addieren* zwei Zahlen übergeben bekommen und deren Summe zurückgeben. Allgemein besitzen Funktionen: Anweisungen um einen speziellen Vorgang durchzuführen; Parameter durch die die Aktionen initialisiert werden; unter Umständen einen Rückgabewert (Ergebnis) und damit das Ergebnis der Aktion.

Wir können somit eine Funktion folgendermaßen definieren:

```
void addmult(int y, int z)
{
    int a = y + z;
    int b = y * z;
```

```
    cout << y << " plus " << z << " ergibt " << a;
    cout << y << " mal " << z << " ergibt " << b;
}
```

Das Beispiel zeigt eine Funktion ohne Rückgabewert. Sie berechnet die Summe und das Produkt der Parameter y, z und gibt diese aus. Der Name der Funktion ist *addmult* und die Übergabeparamter sind y, z. Die Bezeichnung *void* sagt aus, dass diese Funktion keinen Rückgabewert hat.
Die Funktion können wir dann z.B. durch den Aufruf *addmult(3,4)* benutzen.

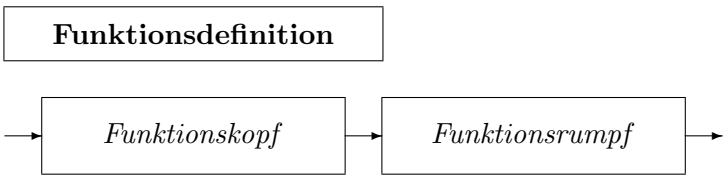
Eine Funktion mit Rückgabewert besitzt statt der Bezeichnung *void* den Rückgabetyp, z.B. *int* oder *bool*. Außerdem muss das Ergebnis dieser Funktion (der Rückgabewert) zurückgegeben werden. Dies geschieht durch den Aufruf *return* und danach den Ergebniswert (*return x*). In C++ muss nicht jeder Weg in dieser Funktion ein *return* enthalten, jedoch wird ein nichtdefinierter Wert zurückgegeben, wenn die Funktion ohne *return* Aufruf beendet wird. Dies kann zu Fehlern führen, die nur schwer zu finden sind. Deshalb ist es sinnvoll, dass alle Codepfade einer Funktion mit Rückgabewert ein *return* enthalten. Das bedeutet, falls eine *if* Anweisung in der Funktion definiert ist, muss sowohl der *true* Fall, als auch der *false* Fall, insgesamt nach Abarbeitung aller Befehle in der Methode, zu einem *return* führen. Die Funktion muss somit ein Ergebnis zurückgeben.

Das folgende Beispiel zeigt eine Funktion mit Rückgabewert. Der Rückgabetyp ist dabei Integer.

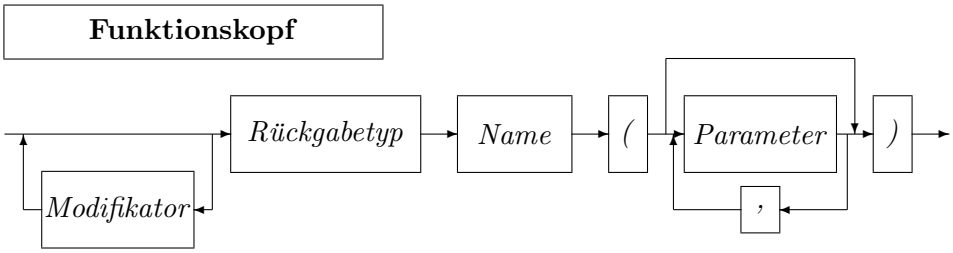
```
public static int add(int y, int z)
{
    int erg = y + z;
    return erg;

}
```

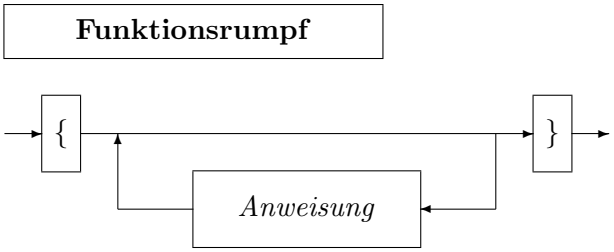
Jede Funktion hat einen Funktionskopf und einen Funktionsrumpf.



Innerhalb des Kopfes werden die Basisdaten der Funktion definiert. Die Definition beginnt mit den Modifikatoren, von denen beliebig viele aufgeführt werden können. Wir definieren vorerst kein Modifikator. Falls kein Rückgabewert definiert werden soll, muss *void* angegeben werden. Es folgt eine runde öffnende Klammer und die Parameter, von denen beliebig viele, durch Kommata getrennt, angegeben werden können. Zuletzt wird die runde Klammer geschlossen.



Im Funktionsrumpf, der durch geschweifte Klammern ({}) begrenzt ist, werden die Anweisungen der Methode aufgeführt. Diese Anweisungen werden nach dem Funktionsaufruf ausgeführt.



Im folgenden Beispiel sehen Sie ein komplettes Programm, welches zuerst eine Funktion ohne Rückgabewert und dann eine Funktion mit Rückgabewert über die *Main*-Funktion aufruft.

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

```
void gerade(int x)
{
   if (x%2!=0)
     cout << "ungerade";
   else
     cout << "gerade";
}

int multi(int x, int y)
{
   int erg = x * y;
   return erg;
}

int main()
{
   int a;
   int b;
   int c;
   cin >> a;
   cin >> b;
   cin >> c;
   gerade(a);
   int z = multi(b,c);
   z = z * 8;
   cout << z;
}
```

In dem Beispiel werden zuerst drei Zahlen (a, b, c) vom Benutzer eingelesen. Danach wird die Funktion *gerade* mit dem Parameter a aufgerufen. Diese Funktion gibt *gerade* aus, wenn die übergebene Zahl gerade ist, andernfalls *ungerade*. Die Funktion hat keinen Rückgabewert, sie gibt lediglich etwas auf dem Bildschirm aus.
Danach wird die zweite Funktion *multi* aufgerufen. Dabei werden zwei Parameter übergeben (b, c). Die Funktion multipliziert die beiden Werte und gibt sie durch das Schlüsselwort *return* zurück. Das Ergebnis wird in die Variable z in der *Main*-Funktion abgespeichert, dann mit 8 multipliziert und danach ausgegeben.

Übungen:

1. Schreiben Sie eine Funktion, die zwei Integerzahlen übergeben bekommt und die kleinere der beiden Zahlen zurückgibt.

2. Schreiben Sie eine Funktion, die ein Zeichen übergeben bekommt. Wenn das Zeichen ein Vokal ist, soll *Vokal* auf dem Bildschirm ausgegeben werden, andernfalls *Konsonant*.

3. Schreiben Sie eine Funktion, die 3 Integerzahlen übergeben bekommt und die kleinste mit der größten multipliziert und die mittlere aufsummiert. Das Ergebnis soll zurückgegeben werden.

4. Schreiben Sie eine Funktion, die eine Zufallszahl erzeugt und zurückgibt.

2.6 Rekursion

Ein Programm ist rekursiv, wenn es mindestens eine Funktion enthält, die sich selbst direkt, bzw. indirekt aufruft. Um eine Endlosschleife von Aufrufen zu umgehen, muss die Funktion einen Fall enthalten, in dem die Rekursion abbricht. Die Definition dieses Falls nennt man Rekursionsanker.
Den Teil der Funktion in dem die Funktion wieder erneut aufgerufen wird, heißt Rekursionschritt.

Auch in der Mathematik gibt es rekursive Definitionsweisen. Ein bekanntes Beispiel ist die Berechnung der Summe:

$$\sum_{i=0}^{n} = \begin{cases} x_0 \text{ falls } n = 0 \\ x_n + \sum_{i=0}^{n-1} \text{ falls } n > 0 \end{cases}$$

Hierbei ist der Rekursionsanker bei $n = 0$ und der andere Fall der Rekursionsschritt.

Es wird zwischen vier verschiedenen Rekursionsarten unterschieden:

1. Lineare Rekursion

2. Baumartige Rekursion

3. Geschachtelte Rekursion

4. Wechselseitige Rekursion

2.6.1 Lineare Rekursion

Eine Funktion f heißt linear rekursiv, wenn ein Aufruf von f jeweils maximal einen weiteren Aufruf zur Folge hat.

Ein bekanntes Beispiel dafür ist die Fakultätsfunktion:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

Implementiert sieht dies folgendermaßen aus.

```
int fak(int x)
{
   if (x==0) return 1;
   else return x * fak(x-1);
}
```

In der Funktionsdefinition findet man die mathematische Definition der Fakultät wieder. Die *if*-Bedingung überprüft, ob der Übergabeparameter x gleich Null ist. Wenn dies der Fall ist, dann ist das Fakultätsergebnis gleich 1. Somit wird 1 zurückgegeben, andernfalls ist das Ergebnis der Übergabeparameter mal das Ergebnis der Fakultätsfunktion mit dem um 1 verringerten Übergabeparamter.

2.6.2 Baumartige Rekursion

In einer baumartigen Rekursion hat ein Aufruf einer Funktion f zumindest teilweise mehrere Aufrufe der Funktion f zur Folge.

Die folgende Funktion, die Zweierpotenzen berechnet, ist ein Beispiel hierfür:

```
int pot(int x)
{
   if (x==0) return 1;
   else return pot(x-1) + pot (x-1);
}
```

Hierbei ist das Ergebnis für den Übergabewert 0 die 1, andernfalls ist es das Ergebnis der Anzahl der Knoten auf der rechten plus der Anzahl der Knoten auf der linken Seite.

2.6.3 Geschachtelte Rekursion

Bei einer geschachtelten Rekursion wird das Ergebnis eines rekursiven Aufrufs als Parameter für einen anderen rekursiven Aufruf verwendet.

Ein Beispiel für geschachtelte Rekursion ist die 91er Funktion von John Mc Carthy. Sie ist definiert durch:

$$M(n) = \begin{cases} n - 10, n > 100 \\ M(M(n + 11)), n <= 100 \end{cases}$$

Das Ergebnis dieser Funktion ist für $n <= 100, n \in \mathbb{N}$ immer 91 und sonst $n - 10$.

```
int M(int n)
{
   if (n>100) return n-10;
   else return M(M(n+11));
}
```

2.6.4 Wechselseitige Rekursion

Eine wechselseitige Rekursion findet statt, wenn sich zwei oder mehrere Funktionen gegenseitig rekursiv aufrufen.

Nehmen wir folgende Definition:

$$f(n) = \begin{cases} 0, \text{ falls } n = 0 \\ x * f(x - 1), \text{falls } n \text{ ungerade} \\ x + f(x - 1), \text{falls } x \text{ gerade} \end{cases}$$

Implementieren könnte man es folgendermaßen:

```
int gerade(int x)
{
   if (x==0)
      return 0;
   else
      if (x%2==1)
         return ungerade(x);
      else
```

```
         return x + ungerade(x-1);
}

int gerade(int x)
{
   if (x==0)
      return 0;
   else
      if (x%2==0)
         return gerade(x);
      else
         return x * gerade(x-1);
}
```

Übungen:

1. Was berechnen die folgenden Funktionen in Bezug auf ihre Übergabeparameter?
2. Wie viele Aufrufe der Funktion sind dafür notwendig?
3. Was berechnet die Funktion im Allgemeinen?

Funktion A

```
int rek1(int x)  // 4
{
   if (x==0) return 1;
   return x * rek1(x-1);
}
```

24

5 Aufrufe (x+1)

x!

Funktion B

```
int rek2(int x) // 7
{
   int z = 0;
   if (x<=0) return 0;
   else
      if (x % 2==0)
      {
         z = rek2(x-1);
      }
      else
```

16

5 Aufrufe (abrunden(x/2)+2)

Summe aller ungeraden Zahlen bis x

```
      {
         z = x + rek2(x-2);
      }

   return z;
}
```

Funktion C

```
int rek3(int x) // 3      6
{
   int z = 0;            1+1*3+1*3*2+1*3*2*1=16 Aufrufe
   if (x<=0) return 1;
   for (int i = 0; i < x; i++)   x!
   {
      z = z + rek3(x-1);
   }
   return z;   z=x*rek3(x-1)
}
```

Funktion D

```
int rek4(int x) // 8     256
{
   int z = 0;            511=2^(x+1)-1 Aufrufe
   if (x<=0) return 1;
   z = rek4(x-1) + rek4(x-1);
   return z;             2^x
}
```

Funktion E

```
int rek5(int a,int b)     a*b
{
   if ( b==1 ) return a;   abrunden(log(x)/log(2))+1 Aufrufe
   else if ( b%2==0 ) return rek5(2*a,b/2);
           else
              return a+rek5(2*a,(b-1)/2);
}
```

- Schreiben Sie ein Programm, das die Summe einer Zahl rekursiv berechnet.

- Schreiben Sie ein Programm, das die Ackermannfunktion rekursiv berechnet.
- In einer Zeichenkette ist ein Ausdruck mit Klammern (und) gespeichert . Erstellen Sie eine rekursive Funktion, die überprüft, ob die Klammerstruktur korrekt ist.
- Schreiben Sie ein Programm, das x^y rekursiv nur durch Multiplikation berechnet.
- Schreiben Sie ein Programm, das die Fibonacci Folge rekursiv berechnet.
- Schreiben Sie ein Programm, das ganz ohne Schleifen für eine Zahl bestimmt, ob sie eine Primzahl ist.
- Schreiben Sie ein Programm, das das Pascalsche Dreieck rekursiv berechnet und ausgibt.

2.7 Zeiger und Referenzen

2.7.1 Zeiger

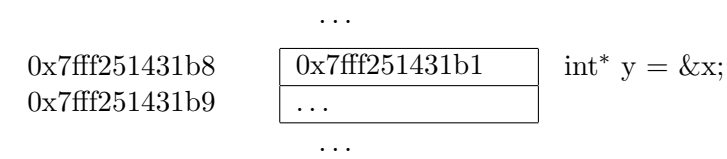
Eine gewöhnliche lokale Variable wird im Arbeitsspeicher auf dem Stack angelegt. Der Speicherplatz ist direkt über den Namen der Variablen verfügbar, er kann somit neu belegt werden oder es kann der Inhalt gelesen werden. Nun gibt es in C++ noch die Möglichkeit Variablen zu deklarieren, die nicht einen konkreten Wert enthalten, sondern die Adresse des Hauptspeicherplatzes einer Variablen abspeichern. Diese Variablen nennen sich Zeigervariablen. Kommen wir zu einem Beispiel:

```
int x = 3;
int* y = &x;
cout << *y; // 3
x = 7;
cout << *y; // 7
*y = 11;
cout << x; // 11
```

Eine Zeigervariable wird deklariert, indem der Datentyp des Wertes der Speicheradresse angegeben wird, auf die die Zeigervariable zeigen soll und zusätzlich ein Stern * angegeben wird. In dem Beispiel soll die Variable y auf die Speicherstelle von x zeigen. Somit wird $int*$ definiert, da x vom Datentyp *int* ist und y ein Zeiger auf die Speicherstelle von x sein soll. Mit dem & Operator kann die Hauptspeicheradresse einer Variablen ermittelt werden. $\&x$ gibt somit die Adresse von x zurück (z.B. 0x7fff251431b4). Die Zeigervariable enthält nun die Adresse von x. Wenn nun auf den Wert von x über y zugegriffen werden soll, dann muss y dereferenziert (sich auf ein Objekt beziehen, auf das ein Zeiger zeigt) werden, es muss dem Zeiger gefolgt werden. Dies geht über die Angabe eines Sterns vor der Zeigervariable ($*y$). Hierdurch kann der Wert an der Stelle im Hauptspeicher von x gelesen werden und auch verändert werden.

In der Regel können Rechner minimal ein Byte adressieren. Dies ist auch der Grund dafür, dass *boolean* Variablen in der Regel ein Byte Speicher verbrauchen.
Schauen wir uns das Ganze nochmal im Arbeitsspeicher an:

Adresse	**Speicherstelle**	**Programmcode**
0x7fff251431b1	3	int x = 3;
0x7fff251431b2		
0x7fff251431b3	...	



Wir sehen im Beispiel, dass die echte Speicheradresse von x in y abgespeichert wird.

In der Deklaration von Zeigervariablen muss nicht zwangsläufig Datentyp und dann der Stern angegeben werden:

```
int* x;
bool* b;
char* c;
```

Oft wird der Stern direkt an den Bezeichner angehängt, also:

```
int *x;
```

Dies wird gemacht, da folgende Definition zu einem Fehler führt:

```
int* r,s;
r=&x;
s= &x; // Fehler
```

Nur die erste Variable r ist hierbei als Zeiger definiert worden, da sich die Zeigerangabe nur auf die erste Variable nach der Definition bezieht. Die zweite Variable s ist eine gewöhnliche Integer Variable. Die korrekte Angabe für zwei Zeiger muss

```
int *r, *s;
s=&x;
r=&x;
```

sein.

Zeiger müssen nicht immer auf andere Variablen verweisen. Man kann für sie auch neuen Speicher reservieren und auf diese Speicherstelle über diese verweisen. Dies geschieht über den *new* Operator:

```
int *p = new int;
*p = 4;
cout << *p; // 4
```

```
int *l = p;
*l =7;
cout << *p; // 7
```

Der Speicher indem der Wert von p abgelegt wird, wird nicht im Stack reserviert, sondern im Heap. Der Heap ist nicht wie der Stack organisiert, also durch das Last in First out Prinzip, sondern hier wird Speicher an der ersten freien Stelle reserviert und der Speicher wird nicht automatisch freigegeben, wie beim Stack nach dem Verlassen des Blocks. Speicher aus dem Heap muss durch den Aufruf *delete* wieder freigegeben werden:

```
delete p;
p = new int;
```

Dies löscht den Speicher im Heap, auf den p gezeigt hat. Danach kann p weiter verwendet werden. Jedoch können hierbei Fehler programmiert werden:

```
int* p = new int;
*p = 4;
cout << *p;
int *l = p;
*l =8;
cout << *p;
delete p;
p = new int;
cout << *l; // Das kann zu Fehlern fuehren
```

Wenn ein anderer Zeiger immernoch auf eine Speicherstelle zugreift (lesend oder schreibend), die gelöscht worden ist, dann kann es zu schwerwiegenden Fehlern wie Absturz oder falschen Ergebnissen kommen. Man muss also als Programmierer sicher gehen, dass keine Zeigervariable mehr versucht auf eine Speicherstelle zuzugreifen, die vorher gelöscht worden ist.

Wenn definiert werden soll, dass ein Zeiger nirgendwohin zeigt, dann bekommt er 0 zugewiesen:

```
int *p=0;
```

Die Adresse wird nicht vergeben, deshalb ist dies ein gutes Element dafür. Häufig initialisiert man, wie oben im Beispiel, Zeiger mit der 0.

Es gibt auch die Möglichkeit Zeiger auf Zeiger zu definieren:

```
int x = 3;
int *y = &x;
int **z = &y;
```

Über die einfache Dereferenzierung $^{*}z$ wird auf die Adresse von x zugegriffen und über die doppelte Dereferenzierung von $^{**}z$ wird auf den Wert von x zugegriffen. Schauen wir uns das im Arbeitsspeicher an:

Adresse	**Speicherstelle**	**Programmcode**
0x7fff251431b1	3	int x = 3;
0x7fff251431b2		
0x7fff251431b3	...	
	...	
0x7fff251431b8	0x7fff251431b1	int*y = &x;
0x7fff251431b9	0x7fff251431b8	int**z = &y;
0x7fff251431ba	...	
	...	

void Zeiger

Ein *void* Zeiger *void** kann einen beliebigen Zeiger auf ein Typ von Objekten aufnehmen. *void* Zeiger können auch anderen Zeigervariablen vom Typ *void* zugewiesen werden. Die Zeiger können miteinander verglichen werden. Ein solcher Zeiger kann mit *static_cast* explizit auf einen anderen Zeigertyp konvertiert werden. Andere Operationen, wie Dereferenzierung, wären nicht sicher.

Nun ein Beispiel:

```
void f(int* x)
{
   void* z = x;
   *z; // Fehler
   z++; // Fehler
   int* y = static_cast<int*>(z);
}
```

Allerdings sind diese Anwendungen sehr fehleranfällig und sollten nur selten benutzt werden. Was vermieden werden sollte, ist auch einen Zeiger per Cast in einen anderen Zeigertyp zu verwandeln. Hierbei kann nicht auf jedem Rechner von einem einheitlichen Verhalten ausgegangen werden.

2.7.2 Referenzen

Referenzen sind alternative Bezeichnungen für ein Objekt, beispielsweise eine Integervariable. Das bedeutet, für eine neue Deklaration wird zwar kein neuer Speicherplatz angefordert, jedoch bekommt ein Speicherplatz zwei verschiedene Namen, mit denen er angesprochen werden kann.

Referenzen werden mit einem & Operator deklariert:

```
int x = 12;
int& r1 = x;
int &r2, &r3 = x;
cout << r1 << r2 << r3 ; // Ausgabe: 121212
int& r7; \\ Fehler und ist danach unveraenderlich an diese gebunden.
```

Jede Referenz muss bei ihrer Definition direkt eine Variable zugewiesen bekommen.

Eine große Anwendung von Referenzen ist das Definieren einer Referenz in einer Funktion als Übergabeparameter. Hierzu ein Beispiel, dass zwei Zahlen übergeben bekommt und diese vertauscht:

```
void switch(int& x, int& y)
{
   int hilf=x;
   x=y;
   y=hilf;
}

int main()
{
   int c=7;
   int d=8;
   cout << c << d; //78
   switch(c,d);
   cout << c << d; //87
}
```

Dadurch, dass die Übergabeparameter als Referenz definiert sind, wird nicht auf Kopien gearbeitet, sondern der Inhalt der echten Speicherstellen der Variablen verändert. Somit sind die Variablenwerte c, d nach dem Aufruf von *switch* vertauscht.

Wenn Referenzen als Übergabeparameter deklariert worden sind, dann können natürlich eigentlich keine konstanten Werte übergeben werden, da diese nicht verändert werden dürfen(bsiepielsweise “Hallo“ und “Welt“). Konstante Werte können nur übergeben werden, wenn hierfür die Übergabeparameter zusätzlich als *const* deklariert werden:

```
void ausgabe(const string& s, const string& r)
{
    cout << s << " und "<< r;
}


int main()
{
     ausgabe("Hallo", "Welt");
     return 1;
}
```

Jedoch ist es hierbei nicht möglich innerhalb der Funktion die Variablenwerte zu verändern.

Es können nicht nur Referenzen übergeben werden, sondern es können auch Referenzen zurückgegeben werden:

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

Das Beispiel zeigt eine Funktion, die zwei Variablen über *Call by Value* übergeben bekommt und die eine Referenz auf die Größere zurückgibt. Jedoch sind x und y Kopien der Aufrufparameter und somit lokale Variablen, die nach Verlassen des Gültigkeitsbereichs gelöscht werden. Die Referenz verweist somit auf eine Speicherstelle, die eigentlich nicht mehr reserviert ist. Die korrekte Lösung sieht so aus:

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

Hierbei werden x, y als Referenz übergeben und somit bleiben die Speicherstellen für x, y weiter existent. Dieses Übergabekonzept nennt sich *Call By Reference* (die gewöhnliche Vaiante nennt sich *Call By Value*).

Ein *Call by Reference* kann auch über eine Kombination aus Zeigern und Referenzen erzeugt werden:

```
void ausgabe(string* s)
{
   *s = "Hallo Welt";
}

int main()
{
   string m;
   ausgabe(&m);
   cout << m << endl; // Hallo Welt

}
```

In diesem Beispiel wird die Speicheradresse der Variablen m der Funktion *ausgabe* übergeben, die diese Adresse in einer Zeigervariablen vom Typ *string* aufnimmt, dann die Zeichenkette *Hallo Welt* darin abspeichert. Nun steht somit auch die Zeichenkette *Hallo Welt* in m drin.

2.8 Arrays

Problem 1: Wir wollen ein Programm schreiben, das 100 Integer Werte einliest und dann in umgekehrter Reihenfolge ausgibt.

Bishergie Lösung: Wir definieren uns 100 Integer Variablen und geben diese in umgekehrter Reihenfolge aus, mit der wir sie belegt haben.

$\Rightarrow$ Der Programmieraufwand ist ernom.

Problem 2: Wir wollen ein Programm schreiben, dass vom Benutzer einliest, wie viele Werte im Folgenden eingelesen werden sollen. Dann wird diese Anzahl von Werten eingelesen und in umgekehrter Reihenfolge ausgegeben.

$\Rightarrow$ Das Problem ist mit den bisherigen Programmiermitteln nicht lösbar, da nicht im Voraus entschieden werden kann, wie viele Werte eingelesen

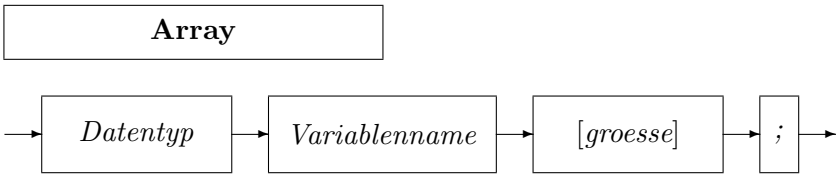
werden sollen.

Um diese Probleme lösen zu können, existieren in den modernen Programmiersprachen Arrays (Felder). Ein Array kann man sich als ein Feld vorstellen, welches in mehrere gleichgroße Teile unterteilt ist. Diese Teile sind alle von einem Datentyp, also Integer, Float, Character, ...

Felder	3	4	7	9	1	8	4	1	7	8
Index	0	1	2	3	4	5	6	7	8	9

Jedes Array hat einen eindeutigen Namen. Über Namen und Index des Feldes ist jeder Wert in einem Array eindeutig identifizierbar.

Die Definition eines Arrays beginnt mit der Angabe des Datentyps und dem Bezeichner. Dann wird zur Feldgrößenangabe eine öffnende eckige Klammer, die Größe und eine schließende eckige Klammer geschrieben. Diese bezeichnen die Arraydeklaration und unterscheiden die Definition somit von einer gewöhnlichen Variblendeklaration. Ein Semikolon dient zum Abschließen der Definition.




```
int x[10];
char c[3];
int z=9;
int arr[z];
```

Auf jedes einzelne Feld kann durch den Index des Feldes zugegriffen werden. Der Index wird hinter der Arrayvariablen angegeben. Wie oben beschrieben beginnt der Index bei 0. Somit stehen die Felder von 0 bis 4 zur Verfügung:

```
x[0] = 7; x[1] = 8; x[2] = 9; x[3] = 1; x[4] = 10;
```

Der Wert eines Feldes kann auch an andere Variablen übergeben werden:

```
int y = x[0]; // y hat den Wert 7
```

Schauen wir uns die Deklaration und die Wertzuweisung von Array Feldern im Arbeitsspeicher an.

Adresse	**Speicherstelle**	**Programmcode**
0	...	
1	...	
2	...	
	...	
8	435532	int x[4]; /* Hier werden die Speicherstellen 8-11 für das Array x reserviert*/
9	rwtetz	
10	1.7869	
11	0	
12		
	...	

Es werden im Arbeitspeicher 4 Speicherstellen reserviert. Diese werden nicht mit Nullwerten belegt. Der eigentliche Speicherplatz des Arrays (Stelle 8 im Beispiel) wird mit der Nummer der ersten Speicherstelle belegt. Jetzt sind die Werte, die in den Speicherstellen des Arrays eingetragen werden, über den Arraynamen erreichbar. Wichtig dabei ist, dass in C++ nicht davon ausgegangen werden kann, dass die Speicherstellen eines neu initialisierten Arrays Null Werte enthalten. Es kann etwas beliebiges drin stehen, das aus früherer Nutzung enthalten ist.

Nun wird ein Feld des Arrays mit dem Wert 7 belegt $x[1] = 7$.

8	435532	$x[1] = 7$; /* Das Feld 1 wird mit 7 belegt.*/
9	7	
10	1.7869	
11	0	
12		
	...	

Die Speicherstelle 9, die das erste Feld des Arrays symbolisiert, wird mit dem Wert 7 belegt. Sie ist über $x[1]$ zugänglich.

Zeiger auf Arrays sind eine häufige Anwendung, um die Größe von Arrays zur Laufzeit zu bestimmen. Hierbei hat man den Vorteil, dass man sich nicht vor Programmstart festlegen muss, wie groß ein Array werden soll und der Arbeitsspeicher wird nicht permanent von den Werten belastet wird.

Die Deklaration beginnt mit einer gewöhnlichen Definition eines Zeigers auf einen beliebigen Datentyp, wie zum Beispiel Integer:

int *x;

Danach kann über *new* ein Speicherbereich aus dem Heapspeicher dem Zeiger zugewiesen werden. Hier wird entschieden, wie groß das Array ist:

x= new int[10];

Nun kann ganz normal über

int n=x[7];

auf die 8. Arraystelle zugegriffen werden.

Nun nochmal ein Beispiel, wie über ein Array dynamisch erzeugt werden kann und dann befüllt wird:

```
int x;
cin >> x;
int *y = new int[x];
for(int i=0; i<x; i++)
{
   cin >> y[i];
}
```

Im nächsten Schritt wird ein neues Array y erzeugt ($int* \ y = new \ int[3];$) und dann ein Zeiger $int * x;$ gleich diesem gesetzt. Im Speicher wird dabei nur ein Speicherplatz für den Verweis von y reserviert und danach der Wert von der Verweisstelle y in x übernommen.

Adresse	**Speicherstelle**	**Programmcode**
0	8	$int *x = y;$
1	8	
2		

...

Durch die Deklaration von x wird eine Speicherstelle (im Beispiel Stelle 1) reserviert. An diese Speicherstelle wird nun der Verweiswert von y eingetragen (im Beispiel 8).

Genau hierbei liegt die Gefahr beim Programmieren. Beide Verweise greifen auf dieselben Felder zu. Wenn nun ein Wert eines Feldes von y verändert wird ($y[1] = 3$), ändert sich auch der von x.

8	0	$y[1] = 3;$
9	3	/* Über x wird nun auf
10	0	das selbe Feld zugegriffen wie
11	0	über y.*/
12		

Im Folgenden ein ähnliches Beispiel mit Ausgabe:

```
int* x = new int[4];
x[1] = 7;
int* y = new int[3];
y[1]=8;
int* hilf= y;
cout << x[1];
cout << y[1];
y=x;
delete hilf;
cout << x[1];
cout << y[1];
y[1] = 3;
cout << x[1];
cout << y[1];
```

Das Beispiel initialisiert zwei Felder x, y. Beide werden mit Werten an der Stelle 1 belegt $7, 8$. Danach werden beide ausgegeben $7, 8$. Jetzt werden die beiden Felder gleichgesetzt und dann ausgegeben $7, 7$. Jetzt wird das erste Feld von y mit 3 belegt und damit auch das erste Feld von x. Die letzte Ausgabe ergibt $3, 3$. Wichtig ist hierbei, dass die ursprünglichen Felder vom Array y noch gelöscht werden, da der Speicherplatz nach dem zuweisen von x nicht mehr zugänglich ist und der Speicher somit belegt wird, aber nicht mehr benutzt werden kann. Dazu wird ein Hilfszeiger benutzt.

Zu beachten ist, dass die Feldgroße eine Konstante sein muss:

```
void arraydefinition(int i)
{
   int arr[i]; // Fehler
}
```

Zeiger und Arrays stehen in C++ in enger Verbindung. Der Name des Arrays ist nichts weiter, als ein konstanter Zeiger auf die erste Stelle des Arrays. Der Compiler überführt den Ausdruck

```
arrayname[index]
```

intern immer stets in den Ausdruck

```
*(&arrayname[0]+index)
```

Dabei ist

```
&arrayname[0]+index
```

die Summe aus der Anfangsadresse der betreffenden Arrays und einer ganzen Zahl. Diese Operation nennt sich Zeigerarithmetik. Auf Zeiger können die Operatoren $+, -, ++, --$ angewendet werden. Nun ein konkretes Beispiel:

```
short a[10] = { 0};
short *p = &a[0];
p=p+1;
```

Nach der Definition eines *short* Arrays mit 10 Stellen, wird ein Zeiger definiert und die Adresse des Arraybeginns dem Zeiger zugewiesen. Nachdem der Zeiger inkrementiert wurde, zeigt er auf die zweite Stelle im Array.
Der Name des Arrays stellt die Anfangsadresse des Arrays dar. Der Compiler konvertiert den Namen automatisch in einen Zeiger und addiert den Index auf den Zeiger.

Die erste Speicherstelle des Array

```
short arr[10] = {0};
```

ist gegeben durch

```
&arr[0]
```

und

```
arr
short *p = &arr[0];
```

initialisiert p mit der Adresse des ersten Arrayelements

```
short *p = arr;
```

initialisiert p mit der Adresse des ersten Arrayelements.
Da ein Arrayname ein konstanter Zeiger ist, geht folgendes nicht:

```
arr=p;
```

(arr+n) und $arr[n]$ sind somit die gleichen Ausdrücke.

```
for (int i=0; i<10; i++)
arr[i]=i;
```

```
for (int i=0; i<10; i++)
*(arr + i) = i;
```

Wenn die Grenzen eines Arrays überschritten werden, erhält man vom C++-Compiler keinen Hinweis.

```
arr[10]=7
```

würde beispielsweise ein Feld ansprechen, welches nicht durch die Initialisierung vom Array im Speicher reserviert wurde. Dies führt dazu, dass irgendwelchen Speicherstellen, die nicht zu dem Array gehören, ein Wert zugewiesen wird. Das Gegenstück dazu ist, wenn ein Array im negativen Bereich überschritten wird $arr[-1] = 7$.

Um einen Parameter vom Typ *char* zu vereinbaren, genügt es, den Parametertyp mit *char** oder *char*[] festzulegen. Es ist keine Größenangabe notwendig, da C++ sowieso keine Indexprüfungen vollzieht.

Die beiden Definitionen sind identisch, da beide einen Zeiger auf *char* repräsentieren.

void arr(char * a)
void arr(char[])

Zu beachten ist, dass Arrays durch einen konstanten Zeiger auf das erste Element repäsentiert werden. Bei der Übergabe als Parameter wird somit nur eine Kopie des Zeigers auf das erste Element angelegt. Dies führt dazu das alle Änderungen innerhalb der Funktion auf dem ursprünglichen Array durchgeführt werden. Da nur eine Kopie des Zeigers auf das erste Element durchgeführt wird, ist auch innerhalb der Funktion die Benutzung von *sizeof()* nicht möglich, da dies nur die Länge des Zeigers zurückgibt. Die Länge des Arrays muss somit immer noch mit übergeben werden.

Übungen:

1. Schreiben Sie eine Methode, die 10 Integerzahlen vom Benutzer einliest und in einem Array abspeichert.

2. Schreiben Sie eine Methode, die ein Array übergeben bekommt und überprüft, ob jede Zahl des Arrays gleich der Summe der beiden Vorgänger ist.

3. Schreiben Sie ein Programm, das n rationale Zahlen einliest / einlesen kann und den Mittelwert, das Produkt und das Maximum dieser Zahlen ermittelt und ausgibt.

4. Schreiben Sie eine Methode, die ein Integerarray übergeben bekommt und die kleinste Zahl zurückgibt.

5. Schreiben Sie ein Programm, das zwei sortierte Integer Arrays a, b übergeben bekommt und dann ein Array c erzeugt, in dem alle Zahlen beider Arrays sortiert enthalten sind.

6. Schreiben Sie ein Programm, das zwei Vektoren in jeweils ein Array einliest und deren Skalarprodukt berechnet.

7. Um ein Array a aufsteigend zu sortieren, kann man wie folgt vorgehen: Wenn zwei benachbarte Elemente in der falschen Sortierreihenfolge stehen, vertauscht man sie. Wenn im gesamten Array keine Vertauschungen mehr möglich sind, ist das Array sortiert. Schreiben Sie eine Methode, die das Array a nach diesem Verfahren sortiert.

Projektideen

- Implementieren Sie das Sodoku Spiel. Erweitern Sie es dazu, dass der Computer ein Rätsel selbst löst, das vom Benutzer eingegeben wird.

- Implementieren Sie Conways Spiel des Lebens.

2.9 Strukturen

Bisher konnten lediglich Variablen von Datentypen erstellt werden, die zu den C++ Standardtypen gehören - die intrinsischen Datentypen - oder die über die Header-Dateien mit #include eingebunden und damit bekanntgemacht worden sind.

Eine Struktur ist nun ein benutzerdefinierter Datentyp - ein Datentyp, den Sie selber erstellen. Dieser Datentyp besteht genaugenommen aus Variablen anderer Datentypen. Um einen neuen Datentyp zu erstellen, geben Sie zuerst das Schlüsselwort *struct* an. Hinter *struct* folgt der Name des neuen Datentyps. Hinter dem Namen werden in geschweiften Klammern nun eine Reihe von Variablen definiert, aus denen der neue Datentyp bestehen soll.
Eine Struktur muss genauso wie eine Funktion zuerst definiert werden, bevor sie angewandt werden kann.
Die Definition erfolgt über das Schlüsselwort *struct*, danach kommt in geschweiften Klammer die Definition des eigentlichen Typs. Hierbei werden alle einzelnen Elemente mittles einer Variablendeklaration angegeben, die der neue Typ enthält.

In dem folgenden Beispiel wird ein Datentyp *adresse* erzeugt:

```
struct adresse
{
   string Anrede;
   string Vorname;
   string Nachname;
   string Strasse;
   int Hausnummer;
   int Postleitzahl;
   string Ort;
   string Land;
};
```

Ist eine Struktur definiert und damit ein neuer Datentyp erstellt worden, können Sie Variablen von diesem neuen Datentyp anlegen:

```
int main()
{
   adresse MeineAdresse;
   MeineAdresse.Anrede = "Herr"
```

```
   MeineAdresse.Vorname = "Boris";
   MeineAdresse.Nachname = "Schaeling";
   MeineAdresse.Strasse = "Schlossallee";
   adresse* p=&MeineAdresse;
   p->Postleitzahl = 12345;
   p->Ort = "Entenhausen";
   p->Land = "Deutschland";
}
```

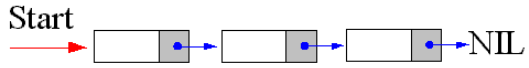
Für den Zugriff auf die einzelnen Elemente der Struktur wird entweder der . - *Operator* oder der $->$ - *Operator* benutzt.
Der Unterschied zwischen den beiden Operatoren besteht darin, dass . auf eine Variable der Struktur angewendet wird und $->$ für Zeiger auf den Strukturtyp.
Der Operator $p->$ ist eine andere Schreibweise für $*(p)$..

2.9.1 Verkettete Listen

Arrays werden benutzt um Elemente gleichen Typs abzulegen und sie einfach zu verwalten (beispielsweise mittels Schleifen). Arrays haben den Nachteil, dass sie sehr unflexibel sind. Das bedeutet, dass sie bei ihrer Definition eine Größe zugewiesen bekommen und dann nicht mehr veränderbar ist. Dies hat zum einen den Nachteil, dass schon bei Initialisierung des Arrays der komplette Platz für das Array reserviert wird, obwohl noch keine Werte enthalten sind. Zum anderen hat es den Nachteil, dass wenn der Platz im Array nicht mehr ausreicht, das komplette Array nicht erweitert werden kann.
Eine dritter Nachteil ist, dass nur Elemente gleichen Typs in einem Array gespeichert werden können.

Bei verketteten Listen werden Elemente dynamisch erzeugt und mittels Zeiger miteinander verbunden. Hierfür werden sowohl Strukturen zur Definition einer der Elemente benötigt, als auch das Konzept der Zeiger.

Schauen wir uns zuerst einmal die Idee der verketteten Liste etwas näher an.



- Ein Element der Liste heißt Knoten.
- Ein Element einer Liste besteht immer aus einem Zeiger und Dateninhalten.

Bei gewöhnlichen Arrays werden die Elemente hintereinander im virtuellen Arbeitsspeicher abgelegt. Dies ermöglicht über die Zeigerarithmetik einen direkten Zugriff auf die einzelnen Elemente. Dies ist jedoch bei verketteten Listen nicht der Fall. Hier liegen die einzelnen Elemente nicht unbedingt hintereinander im Arbeitsspeicher, weil die Größe der verketteten Liste unter Umständen ständig ändert. Deshalb muss die Liste erst bis zu dem jeweiligen Element durchlaufen werden, um auf es zuzugreifen. Der Vorteil von verketteten Liste im Vergleich zu Arrays liegt in der Möglichkeit, Strukturen dynamisch zuerzeugen und zu verändern. Nachteile sind, dass die Programme schwer implementierbar sind und das sie schwer verständlich werden können.

Die Anwendung finden verkettete Listen bei dynamisch veränderbaren Datenstrukturen und bei großen Datenstrukturen, beispielsweise, wie ein Adressbuch.

Schauen wir uns nun mal die Implementierung an:

```
#include <iostream>
using namespace std;
struct element
{
   int inhalt;
   element next; // Verknuepfung zum Nachfolger
};
elem* Anker = 0; // Anfang der Liste
```

Die Implementierung beginnt mit der Elemente. Sie besteht aus den Daten *inhalt* und dem Verweis auf das nächste Element *next*.
Weiter wird der Anker erzeugt, der nichts anderes ist, als ein Zeiger auf ein *element*. Dieser wird zu beginn auf 0 gesetzt. Wenn die Liste leer ist, muss der Anker auf 0 gesetzt werden, sonst treten Fehler auf, weil sonst nicht erkannt wird, dass die Liste leer ist.

Der folgende Programmteil beschreibt, wie werde in eine Liste eingefügt werden können:

```
int main()
{
   int Inhalt;
   element *node;
   // Fuelle die Liste mit Zahlen, bis 0 eingegeben wird
   do
```

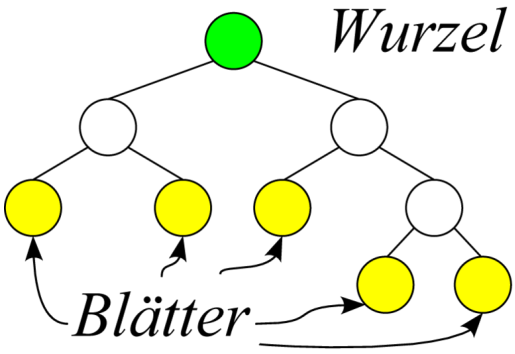
```
   {
      cout << " Zahl eingeben (0 fuer Ende)" << endl;
      cin >> Inhalt;
      if (Inhalt)
      {
           //Neues Element fuer die Liste erzeugen:
         element *node = new element;
         node->inhalt = Inhalt;  //Besetze die Daten
         node->next = Anker; //Haenge die bisherige Liste an
         Anker = node;   //Setze den Anfangspunkt hierher
      }
} while (Inhalt);
```

Zuerst wird ein Hilfszeiger *node* erzeugt. Dieser dient später dazu, um die neuen Elemente vorerst zu erzeugen. Danach beginnt die *do while* Schleife, die solange läuft bis der Benutzer eine 0 eingibt, um zu signalisieren, dass er keine weiteren Werte in der Liste abspeichern möchte. Wenn der Benutzer keine Null eingibt, wird in der Schleife ein neues Element erezugt und dann der eingelese Wert in dieses Element geschrieben. Dann wird das neue Element vorne an die Liste angefügt in dem der *next* Zeiger auf den Anker gesetzt wird, also auf das ursprünglich erste Element. Dann wird noch der Anker auf das neue Element gesetzt, damit es über einen allgemein bekannten zeiger erreichbar ist.

```
     // Gebe die Liste in umgekehrter Reihenfolge aus
     // und loesche dabei die ausgegebenen Elemente
   while (Anker)    // ungleich 0! Die Liste ist nicht leer!
   {
      cout << Anker->inhalt << endl;
      old = Anker; // Sichere zum spaeteren Loeschen
      Anker = Anker$->$next; // Ziehe naechstes Element nach vorn
      delete old;   // Loesche das ausgelesene Element
   }
```

Beim Ausgeben der Liste wird in einer *while* Schleife, die solange läuft bis das Ende der Liste erreicht ist (Anker==0), zuerst der Inhalt ausgegeben, dann wird ein Zeiger *node* auf das aktuelle Element gesetzt, weil die Liste noch gelöscht werden soll und dann wird der Zeiger weiter gesetzt und das Element gelöscht.

2.9.2 Binäre Bäume



- Der eindeutige Knoten ohne Vorgänger eines Baums heißt Wurzel.
- Knoten ohne Nachfolger heißen Blätter.
- In einem binären Baum hat jeder Knoten höchstens zwei Nachfolger.

Binäre Bäume werden oft als Suchbäume aufgebaut. Suchbäume sindsortiert. So ist es möglich, dass im Falle einer Suche nicht immer der komplette Baum durchlaufen werden muss, bzw. alle Elemente, sondern der nur ein kleiner Teil davon. Die Sortierung bestaht darin, dass für jeden Knoten gilt, dass sein linker Nachfolger kleiner oder gleich dessen Wert ist und sein rechter Nachfolger größer. So steht beispielsweise das kleinste Element des Baums am linken unteren Blatt des Baums.

Schauen wir uns zuerst die Implementation der Datenstruktur Baum an:

```
struct Baumelement
{
   int Inhalt;
   Baumelement* links;
   Baumelement* rechts;
};
```

Elemente dieser Datenstruktur bestehen aus einem Inhaltsfeld und zwei Zeiger auf *Baumelemente*, der linke Nachfolgerknoten und der rechte Nachfolgerknoten, wodurch der Baum aufgebaut wird.

```
Baumelement* NeuKnoten;
NeuKnoten = new Baumelement;
```

```
NeuKnoten->Inhalt = 5;
NeuKnoten->links = 0;
NeuKnoten->rechts = 0;
```

Beim erzeugen eines Elementes wird, wie gewöhnlich, ein Zeiger erzeugt und ein Speicherplatz zugewiesen. Es wird noch der Inhalt gesetzt und die beiden Nachfolger mit einem Verweis ins Nichts belegt.

Die Datenstruktur beinhaltet auch noch Operationen zum Zeigen und Einfügen und unter Umständen auch noch zum Löschen oder anderen Operationen. Hier wird nur das Anzeigen in *Inorder* und das Einfügen gezeigt.

Das Anzeigen in *Inorder* gibt alle Elemente in ihrer natürlichen Reihenfolge aus, somit ist die Ausgabe sortiert.

```
void ZeigeBaum(Baumelement *Blatt)
{
   if (Blatt==0) return;
   ZeigeBaum(Blatt->links);
   cout << Blatt->Inhalt << endl;
   ZeigeBaum(Blatt->rechts);
}
```

Zunächst prüft sie, ob das aktuelle Blatt überhaupt existiert. Dann ruft die Funktion sich selbst für den Zweig zur linken auf. Dadurch wird die Funktion sich immer wieder aufrufen, bis es keinen linken Zweig mehr gibt. Die Funktion, die keinen weiteren Zeiger mehr findet, wird den Inhalt des aktuellen Knotens ausgeben. Danach ruft sich die Funktion mit dem rechten Teilast auf. Dann probiert sie wieder in den linken Teilbaum zu gehen...

Und nun noch das Einfügen:

```
Baumelement* Einfuegen(Baumelement *Blatt, int Inhalt)
{
   if (Blatt==0) // Freie Position gefunden
   { // Neues Blatt einfuegen
      Blatt = new Baumelement;
      Blatt->links = Blatt->rechts = 0;
      Blatt->Inhalt = Inhalt;ueckgeben
   }
   if (Inhalt < Blatt->Inhalt)
```

```
   { // Zuweisung behaelt altes Blatt oder das neue
      Blatt->links = Einfuegen(Blatt->links, Inhalt);
   }
   else if (Inhalt > Blatt->Inhalt)
   { // Zuweisung behaelt altes Blatt oder das neue
      Blatt->rechts = Einfuegen(Blatt->rechts, Inhalt);
   }
   return Blatt; // gib das aktuelle Blatt zurueck
}
```

Dabei wird zunächst das Blatt gesucht, dessen Nachfolger der neue Eintrag werden soll. Diese Stelle muss ein 0-Zeiger sein. An dieser Stelle wird ein neues Blatt eingehängt. Das ist auch gleichzeitig das Rekursionsende. Hier wird das neue Element in den Baum eingefügt und die Adresse des Blatts zurückgegeben, damit es vom Aufrufer in den Baum eingehängt wird. Das Einhängen des neu angelegten Blatts im Baum erfolgt nicht in dem Durchlauf, in dem *new* aufgerufen wird, sondern in der davor aufgerufenen Funktion an der folgenden Stelle:

```
Blatt->links = Einfuegen(Blatt->links, Inhalt);
```

2.10 Zeichenketten

Zeichenketten gehören in C++ nicht zu den Standardvariablentypen. Um sie zu benutzen, müssen sie über *include* eingebunden werden. Danach können sie wie gewöhnliche Datentypen verwendet werden:

```
#include <string>
#include <iostream>

int main()
{
    std::string s ="Hallo Welt";
    std::cout << s;
}
```

Sofern der Namensraum *std* nicht eingebunden ist, muss jede Deklaration einer Variablen mit *std::* beginnen.

Zeichenketten sind aufgebaut wie Arrays, deshalb kann auf die einzelnen Zeichen einer Zeichenkette über eckige Klammer ([]) zugegriffen werden.
Mittels dem Variablenname eines Strings, dann einem Punkt und dann dem Funktionsaufruf *size()* kann auf die Länge der Zeichenkette zugegriffen werden. Mit dem Plus-Operator können Zeichenketten aneinandergehängt werden.

```
#include <string>
#include <iostream>

int main()
{
    std::string s ="Hallo";
    std::string r= " Welt";
    s=s+r;
    std::cout << s.size();
    for(int i=0;i<s.size();i++)
    {
         std::cout << s[i]<<"\n";
    }
}
```

Das Beispiel zeigt, wie zwei Zeichenketten definiert und konkateniert werden. Danach wird jeder einzelne Buchstabe in einer Zeile ausgegeben.

2.11 Objektorientierung

In der objektorientierten Programmierung (OOP) wird die traditionelle Trennung von Daten (Variablen, Konstanten, ...) und Operationen (Funktionen, Prozeduren, ...) aufgegeben. Hier wird die reale Welt in Objekten dargestellt. Nehmen wir ein Autorennen. Bei einer Modellierung auf dem Computer würden verschiedene Objekte des Autorennes in Klassen abstrahiert. Hierbei könnten folgende Klassen erzeugt werden: *Fahrer*, *Rennbahn*, *Rennregel*, *Auto*, *Teams*, ... Alle diese Objekte besitzen Eigenschaften und Operationen. Es werden nicht nur materielle Objekte erfasst, wie zum Beispiel *Auto*, sondern auch informelle wie *Rennregeln*. Zur Verdeutlichung nehmen wir die Klasse *Auto*.

	Klasse Auto
Eigenschaften	*Operationen*
Farbe	Gas geben
Typ	Bremsen
Gang	Lenken
Höchstgeschwindigkeit	Schalten

Ein Programm ist aus verschiedenen Klassen aufgebaut, die durch Attribute (Eigenschaften) und Methoden (Operationen) definiert sind. Eigenschaften einer Klasse sind dabei gegenüber anderen Klassen nicht sichtbar (Datenkapselung, *information hiding*). Das bedeutet, kein Objekt der Klasse *Fahrer*, also kein Fahrer, kann direkt auf die Eigenschaften eines Objekts der Klasse *Auto* zugreifen. Er muss über die Methoden des Objekts auf diese zugreifen. Dies dient vor allem als Schutz, weil durch dieses Prinzip nur gesteuert auf Variablen zugegriffen werden kann.

Der Zugriff wird in der Regel nur auf die Methoden der Klasse gewährt. Hierüber kann auch auf die Eigenschaften der Klasse zugegriffen werden.

Aus der Datenkapselung ergeben sich Vorteile für die Softwareentwicklung:

1. *Erhöhte Sicherheit durch Zugriffsschutz*
   Direkte Zugriffe auf die Merkmale einer Klasse bleiben den klasseneigenen Methoden vorbehalten. Damit sollten Programmierfehler seltener werden.

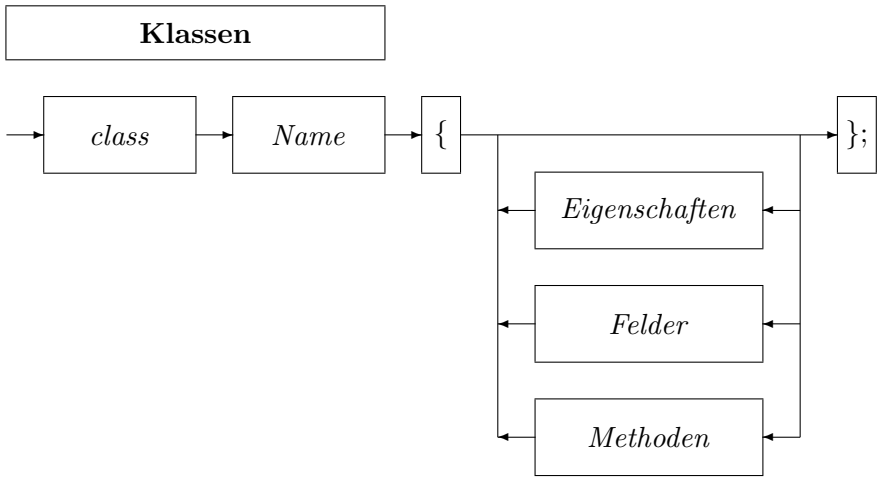
2. *Komfort*
   Wer als Programmierer eine Klasse verwendet, braucht sich um den inneren Aufbau derselben nicht zu kümmern, so dass neben dem Fehlerrisiko auch der Einarbeitungsaufwand reduziert ist.

3. *Flexibilität, Anpassungsfähigkeit*
   Datenkapselung schafft günstige Voraussetzungen für die Wartung bzw. Verbesserung einer Klassendefinition. Solange die Methoden und Eigenschaften der Schnittstelle unverändert bzw. kompatibel bleiben, kann die interne Architektur einer Klasse ohne Nebenwirkungen auf andere Programmteile beliebig geändert werden.

4. *Produktive Teamarbeit durch getrennte Verantwortungsbereiche*
   In großen Projekten können mehrere Programmierer nach der gemeinsamen Entwicklung von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

2.11.1 Klassendefinition

Wie oben beschrieben sind objektorientierte Programme aus Klassen aufgebaut. Eine Klasse wird definiert durch das Schlüsselwort **class**, darauf folgt der Name der Klasse und ein Paar aus geschweiften Klammern. Innerhalb dieser gescheiften Klammern werden die Eigenschaften, Felder und Methoden definiert.



Die Klassen sind ein Baugerüst für ein Objekt, welches eine Abstraktion eines realen Sachverhaltes ist. Nehmen wir an, wir wollten das Autorennenspiel implementieren. Das Auto hätte dabei die Attribute: Farbe, Typ, Geschwindigkeit, Gang ... und die Methoden: Gas geben, bremsen, schalten, ... Schauen wir uns an, wie die Klasse *Auto* implementiert werden könnte:

```
#include <string>

class Auto
{
   private:
      std::string farbe, typ;
      int geschwindigkeit, gang;

   public:

      void setfarbe(std::string f)
      {
          farbe = f;
      }

      void gasgeben()
      {
          if ((geschwindigkeit+10)<= 270)
               geschwindigkeit = geschwindigkeit + 10;
      }

      void bremsen()
      {
          if ((geschwindigkeit-10)>= 0)
               geschwindigkeit = geschwindigkeit - 10;
      }

      void schalten(int x)
      {
          if ((gang + x)<= 0 && (gang+x)>=6)
               gang=gang+x;
      }
      ...
};
```

Die Klasse stellt hierbei den abstrakten Bauplan eines realen Objekts dar. Jedoch wird hierdurch noch kein Objekt definiert. Es muss eine Instanz der Klasse erzeugt werden. Hierdurch wird dann z.B. ein roter VW Golf erzeugt.

In C++ geschieht dies durch den Aufruf:

```
Auto a;
Auto a1;
Auto a3;
```

a ist dabei eine Instanz (ein Objekt) der Klasse *Auto*. Der Vorteil bei objektorientierter Programmierung ist, dass beliebig viele Instanzen der Klasse erzeugt werden können. Um eine Instanz zu erzeugen, muss wie bei einer gewöhnlichen Variablendeklaration die Klasse (der Datentyp) angegeben werden und danach der Bezeichner.

Über den Namen der Instanz, gefolgt von einem Punkt und dem Methodenaufruf können die Operationen der Instanz benutzt werden:

```
a.setfarbe("gruen");
a1.setfarbe("blau");
a3.setfarbe("schwarz");
```

Die unterschiedlichen Instanzen der Klasse *Auto* haben alle unterschiedliche Farben, sind also ganz andere Objekte (ein gruenes Auto, ein blaues Auto, ein schwarzes Auto). Wenn nun also durch den Aufruf

```
a3.gasgeben();
```

die Geschwindigkeit erhöht wird, folgt nur eine Veränderung der Geschwindigkeit des schwarzen Autos.

Natürlich können auch Zeiger auf Objekte erzeugt werden. Dies geschieht, wie bei jedem anderen Datentyp, indem ein Stern hinter dem Datentyp definiert wird oder dem Bezeichner ein Stern hinzugefügt wird.

Schauen wir uns das im Arbeitsspeicher an:

Adresse	**Speicherstelle**	**Programmcode**
0		
1	NIL	Auto* a3;
2		

...

Durch die Deklaration der Instanz $a3$ wird im Speicher eine beliebige Speicherstelle reserviert und mit NIL (lateinisch: nichts") belegt. Diese Stelle (im Bild die Stelle 1) wird als Verweis interpretiert, genau wie bei einem Array. Sie verweist auf die Eigenschaften der Instanz.

Adresse	**Speicherstelle**	**Programmcode**
0		
1	9	a3 = new Auto;
2		
	...	
8		
9	""	typ
10	""	farbe
11	0	geschwindigkeit
12	0	gang
	...	

Die Objektvariable wird nun instanziert. Hierzu dient die Anweisung *a3 = new Auto()*. Durch die Instanzierung wird Speicherplatz für jede Eigenschaft des Objekts reserviert und die erste Stelle dieses Speicherplatzes als Referenz in die Instanzvariable gespeichert.

8		
9	""	
10	schwarz	$a3.setfarbe(''schwarz'');$
11	0	
12	0	
	...	

Die Speicherstelle 9, welche die Eigenschaft *farbe* symbolisiert, wird über die Methode *setfarbe()* mit dem Wert "schwarz" belegt. Wegen des Geheimnisprinzips (siehe Vorteile Objektorientierung) kann diese Eigenschaft nicht direkt über die Instanz angesprochen werden. Nur öffentliche Methoden (mit *public* gekennzeichnet) können auf sie zugreifen.

Im nächsten Schritt wird eine neue Objektvariable $Auto* \ a1;$ erzeugt und diese gleich der Objektvariable $a3$ gesetzt. Im Speicher wird dabei nur ein

Speicherplatz für den Verweis von $a1$ reserviert und danach der Wert der Verweisstelle von $a3$ in $a1$ übernommen.

Adresse	**Speicherstelle**	**Programmcode**
0	9	Auto* a1;
1	9	a1 = a3;
2		
	...	

Durch die Deklaration von $a1$ wird eine Speicherstelle (im Beispiel Stelle 0) reserviert. An diese Speicherstelle wird nun der Verweiswert von $a3$ eingetragen (im Beispiel 9).

Genau hierbei liegt die Gefahr beim Programmieren. Beide Verweise greifen auf die selben Felder zu. Wenn nun ein Wert eines Feldes von $a1$ verändert wird (*a1.setfarbe("rot")*), ändert sich auch der von $a3$.

8		
9	"rot"	*a1.setfarbe("rot")*
10	0	
11	0	
12		

Die Klassen sind Definitionen von neuen Datentypen. Sie können als Rückgabetyp oder als Parameter von Funktionen benutzt werden.

Übungen

- Schreiben Sie eine Klasse *Bruch*. Sie soll folgende Attribute und Methoden besitzen:
  - Attribute: *nenner*, *zaehler*
  - Funktionen: *setnenner*, *getnenner*, *setzaehler*, *getzaehler*

  Schreiben Sie zusätzlich eine Klasse *Bruchrechnung*. Diese soll Funktionen enthalten, um zwei Brüche zu addieren, zu subtrahieren, zu multiplizieren und zu dividieren. Zusätzlich soll sie eine Funktion zum Kürzen der Brüche enthalten und alle Objekte des Typs *Bruch* übergeben bekommen. Die Funktionen sollen alle als Rückgabewert ein Objekt der Klasse *Bruch* haben.

- Modellieren Sie einen Bankautomaten. Dafür Implementieren Sie die Klassen *Bank, Konto, Kunde* und Bankautomat (wenn sie noch andere Klassen brauchen, können ie auch diese modellieren). Der Kunde soll fähig sein an den Bankautomat zu gehen und Geld abzuheben oder Geld einzuzahlen. Dabei soll sich der Kontostand ändern. Zur Erkennung benötigt er einen Benutzernamen und ein Passwort. Es sollen vorerst drei Kunden angelegt werden, die auch jeweils mindestens drei Konten haben. Sowohl Beim Abheben, als auch beim Einzahlen fragt das Programm, von welchem Konto dies geschehen soll.

- Schreiben Sie ein Programm, dass das Maumau Kartspiel simuliert. Dabei sollen 2-3 Spieler gegeneinander spielen können. Der Computer soll dabei darauf achten, dass die Spielregeln eingehalten werden, er soll jedoch nicht sselbst spielen.

2.11.2 Modifikatoren

Zum Schützen der Attribute und Funktionen von Klassen gibt es in C# unter anderem die Zugriffsmodifizierer *public* und *private*. Sie werden bei Definitionen von Methoden und Attributen auf Klassenebene benutzt. Falls eine Methode oder ein Attribut keinen Zugriffsmodifizierer hat, wird es automatisch als *private* interpretiert.

Der Zugriffsmodifizierer *private* bedeutet dabei, dass von außerhalb der Klasse nicht auf Attribute/Methoden zugegriffen werden kann. Von außerhalb bedeutet, dass nicht über ein Objekt der Klasse zugegriffen werden kann.

public bedeutet, dass von außerhalb darauf zugegriffen werden kann. Dies geschieht bei den bisher definierten Variablen über den Objektnamen, einen Punkt und den Variablennamen (siehe Beispiel).

Für jeden Zugriffsmodifikator wird in der Regel ein eigener Bereich angelegt. Dieser beginnt mit dem Zugriffsmodifikator und einem Doppelpunkt. Darauf folgt die Definition der Variablen und Funktionen mit diesem Zugriffsschutz.

Das folgende Beispiel soll dies verdeutlichen:

```
class Auto
{
   int farbe; // weil kein Modifizierer angegeben ist,
```

```
   // ist farbe automatisch private
   private:
      int geschwindigkeit;
   public:
      int gang;
};

int main()
{
   Auto *a = new Auto();
   // a->farbe="schwarz";  (funktioniert nicht)
   // a->geschwindigkeit=40; (funktioniert nicht)
   a->gang=4;
}
```

In dem Beispiel gibt es drei Variablen auf Klassenebene (*farbe, geschwindigkeit, gang*). Dabei sind *farbe* und *geschwindigkeit private* und können deshalb nicht durch die Objektvariable angesprochen werden (a.farbe=“schwarz“; funktioniert nicht, a.geschwindigkeit=40; funktioniert nicht). Das Attribut *gang* ist mit *public* definiert und ist deshalb über eine Objektvariable belegbar (a.gang=4;).

Für Funktionen gelten die gleichen Zugriffsregeln.

Grundsätzlich wird zwischen verschiedenen Variablenarten unterschieden, zum einen Klassenvariablen und Instanzvariablen und zum anderen lokale Variablen der Funktionen. Die Instanzvariablen und die Klassenvariablen sind auf Klassenebene deklariert. Instanzvariablen wurden schon im letzten Kapitel benutzt. Sie werden genau wie die lokalen Variablen deklariert, jedoch nicht innerhalb einer Funktion, sondern unterhalb des Klassenkopfes. Klassenvariablen werden auch dort deklariert. Jedoch enthalten sie den Zusatz *static*.

Durch den Modifizierer *static* wird erreicht, dass das Attribut für alle Instanzen die gleiche Variable ist. Sie hat also den gleichen Speicherplatz im Hauptspeicher (bei Instanzen wurde für jedes Objekt jede Variable neu im Arbeitsspeicher angelegt) und wird darüber hinaus nicht erst durch eine Instanzierung reserviert, sondern ist schon bei Programmstart verfügbar. Diese Variablen werden im globalen Speicher organisiert. Sie müssen nach der Klassendeklaration nochmal definiert werden und einen Wert zugewie-

sen bekommen.

Nehmen wir folgendes vereinfachtes Beispiel:

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

    class Mercedes
    {
        public:
        int farbe;

        static int herstellernummer;
        static int preis;

        void gebePreisaus()
        {
            Mercedes::herstellernummer = 1; // funktioniert
            cout <<preis;
        }
    };
    int Mercedes::herstellernummer=1;
    int Mercedes::preis=90000;
int main()
{
            Mercedes *m = new Mercedes();
            Mercedes *m1 = new Mercedes();
            m->gebePreisaus(); // Ausgabe: 90000
            m1->gebePreisaus(); // Ausgabe: 90000
            Mercedes::preis = 45000;
            m->gebePreisaus(); // Ausgabe: 45000
            m1->gebePreisaus(); // Ausgabe: 45000
            // Mercedes.herstellernumer = 1; // gibt eine Fehlermeldung
}
```

In dem Beispiel werden zwei verschiedene Objekte $m, m1$ erzeugt. Für beide wird der Preis über die öffentliche Methode *gebePreisaus()* ausgegeben. Er ist 90000, da die Methode auf die öffentliche statische Variable *preis* zugreift, die mit einer Standardbelegung von 90000 belegt ist. Nun wird über den Klassennamen, einem Punkt und den Namen der Variable *preis* auf die

Variable zugegriffen und diese verändert. Die Ausgabe des Preises ergibt bei beiden Objekten 45000, da die Objekte auf die gleiche Variable zugreifen. Auf die Variable *herstellernummer* kann von außerhalb der Klasse nicht zugegriffen werden (deshalb ist die Zeile *Mercedes.herstellernummer=1;* auskommentiert. Innerhalb der Klasse kann allerdings darauf zugegriffen werden, wie in der Methode *gebePreisaus()*.

2.11.3 Konstruktoren

Konstruktoren sind Funktionen, die der kontrollierten Initialisierung von Objekten dienen. Bei einer Klasse *Bruch* könnte der Nenner einen Wert, der nicht Null ist, zugewiesen bekommen oder die Attribute *Farbe*, *Typ* und *Gang* einer Klasse *Auto* könnten initialisiert werden. Damit werden unzulässige Objektzustände vermieden, wenn einem Objekt nach Abschluss seiner Instanzierung substanzielle Startwerte fehlen.

Die Bezeichner der Konstruktoren einer Klasse sind gleich lautend mit dem Klassenbezeichner.

- Konstruktoren haben grundsätzlich keinen Rückgabewert, auch nicht *void*.
- Die Parameterliste eines Konstruktors ist beliebig.
- Der Konstruktor einer Klasse wird bei der Instanziierung mit dem Schlüsselwort *new* aufgerufen. Dabei wird durch Anzahl und Typ der Parameter entschieden, welcher Konstruktor aufgerufen wird oder direkt bei der Typangabe, wenn der Speicher nicht dynamisch belegt wird.
- Ein Konstruktor kann nicht auf ein bereits bestehendes Objekt aufgerufen werden, beispielsweise um Instanzvariablen andere Werte zuzuweisen.

Eine Klasse kann mehrere Konstruktoren haben, die sich jedoch in der Anzahl und/oder im Datentyp der Übergabeparameter unterscheiden.

Im folgenden Beispiel besitzt eine Klasse *Person* 3 Konstruktoren:

```
#include <iostream>

using namespace std;
```

```
class Person
{
   private:
         string name, augenfarbe;
         int groesse;

   public:
         Person()
         {
                name = "Mustermann";
                augenfarbe = "blau";
                groesse = 180;
         }
         Person(string name, string augenfarbe)
         {
                this->name=name;
                this->augenfarbe=augenfarbe;
                groesse=180;
         }
         Person(string augenfarbe, int groesse)
         {
                name = "Mustermann";
                this->augenfarbe=augenfarbe;
                this->groesse= groesse;
         }
};

int main()
{
   Person* p = new Person;
   Person* p1 = new Person("Elli", "blau");
   Person* p3 = new Person("blau", 160);
   Person p4("test", "braun");
}
```

Die Konstruktoren aus dem Beispiel initialisieren die Attribute der Klasse. Zu beachten ist, dass die Attributnamen *name, augenfarbe, groesse* sowohl als Instanzvariablen, als auch als lokale Variablen der Konstruktoren (durch Übergabeparameter) definiert sind. Mittels des Schlüsselworts *this* wird auf das Attribut der Klasse zugegriffen, wenn dieses durch den Namen

des Übergabeparameters verdeckt ist.

In der *Main* Funktion werden drei verschiedene Instanzen gebildet (*p, p1 p3*). Über den *new* Operator werden die Konstruktoren aufgerufen und Speicherplatz im Arbeitsspeicher reserviert. Die erste Instanz ruft dabei den ersten Konstruktor auf, die zweite den zweiten Konstruktor und die dritte den dritten Konstruktor.

2.12 Dateien

Für die Arbeit mit Dateien stehen eine Reihe von Klassen in der C++-Standardbibliothek zur Verfügung. Für den Programmierer besonders relevant sind dabei:

- Die Klasse *ofstream* für die Ausgabe in eine Datei
- Die Klasse *ifstream* für die Eingabe aus einer Datei
- Die Klasse *fstream* für Ein- und Ausgabe

Damit der Compiler die Klassen auch kennt, binden wir Sie die Header-Dateien *iostream* und *fstream* ein:

- #include <iostream>
- #include <fstream>

Zum Öffnen gibt es zwei Möglichkeiten. Zur Identifikation dient in beiden Fällen der Dateiname.Entweder Sie geben den Namen gleich als Argument des Konstruktors an. Dann wird das Öffnen der Datei zusammen mit der Erzeugung der Instanz erledigt, beispielsweise

```
ofstream outfile("results.txt");
```

Wenn Sie schon ein Objekt haben und über dieses eine Datei öffnen möchten, so können Sie die Methode *open()* verwenden. Diese verlangt ebenfalls den Dateinamen als Argument.

```
ifstream infile; \\
infile.open(" myfile.txt");
```

Es gibt verschiedene Zugriffsoptionen:

- ios::in - Zum Lesen
- ios::out - Zum Schreiben
- ios::trunc - Datei wird beim Öffnen geleert
- ios::app - Geschriebene Daten ans Ende anhängen
- ios::ate - Positionszeiger ans Ende setzen
- ios::binary - Öffnen von binären Dateien

```
ofstream outfile("tep.dat" , ios::app);
```

Wenn Sie das Schließen selbst übernehmen wollen, rufen Sie dazu die Methode *close()* auf.

```
ofstream file("ergebnis.txt");
... verschiedene Ausgaben
file.close();
```

Für die Standarddatentypen wie *int*, *char* oder *float* stellt die C++-Standardbibliothek die Operatoren >> sowie << zur Ein- und Ausgabe zur Verfügung, die Sie genauso verwenden wie bei der Tastatureingabe beziehungsweise der Bildschirmausgabe. Beachten Sie dabei, dass beim Einlesen Trennzeichen wie Leerzeichen, Tabulator oder Zeilenumbruch standardmäßig überlesen werden.

Sequenz	Bedeutung
\ b	Backspace
$\backslash n$	neue Zeile (entspricht endl)
$\backslash r$	Wagenrücklauf (carriage return)
\“	Anführungszeichen
\\	Backslash

Die entsprechende Schreibmethode heißt *put()*. Auf diese Weise lässt sich auch eine Kopierfunktion realisieren:

```
ifstream in("myfile");
ofstream out("myfile.copy");
char c;
while (in.get(c))
  out.put(c);
```

Möchten Sie nur einzelne Zeichen einlesen, sollten Sie die Methode *get()* verwenden, die auch in mehreren überladenen Versionen für die Standarddatentypen vorliegt. Sie empfiehlt sich vor allem bei binären Dateien, da Sie damit auf jedes Byte einzeln zugreifen können, da sie auch Trennzeichen liest.

```
char c;
  ifstream in(\grqq test.txtgrqq , ios::binary);
  in.get(c);
```

Wollen Sie hingegen ganze Zeilen aus einer Textdatei einlesen, sollten Sie zur Funktion *getline()* greifen. Dabei werden alle Zeichen von der aktuellen

Position des Dateizeigers bis zum Zeilenende eingelesen. Das Zeilenende ist dabei durch ein Newline (
n) gekennzeichnet.

```
string s;
ifstream in("test.txt");
... // irgendeine Bearbeitung
getline(in, s);
```

Der Dateizeiger zeigt auf die Stelle auf die aktuell zugegriffen werden kann. Die Position kann durch folgende Methoden beeinflusst werden:

- *streampos* ist der Typ einer Dateiposition.
- *seek(offset, direction)* setzt die aktuelle Dateiposition.
- ios::beg setzt den Dateizeiger an den Anfang der Datei.
- ios::cur setzt den Dateizeiger an die aktuelle Position des Dateizeigers.
- ios::end setzt den Dateizeiger an das Ende der Datei.
- file.seekg(-10, ios::end) - setzt die aktuelle Position zehn Bytes vor das Dateiende
- - *tellg* liefert die aktuelle Position des fiktiven Lesekopfs der Datei.

Die Funktion *remove()* löscht die Datei, deren Name ihr als Parameter übergeben wird.

```
#include <stdio.h>
int remove(const char ^*dateiname);
```

Die Funktion zum Umbenennen von Dateien heißt *rename()*. Unter UNIX ist es mit dieser Funktion auch möglich, Dateien innerhalb des gleichen Dateisystems zu verschieben.

```
#include <stdio.h>
int rename(const char *dateiname, const char *neuname);
```

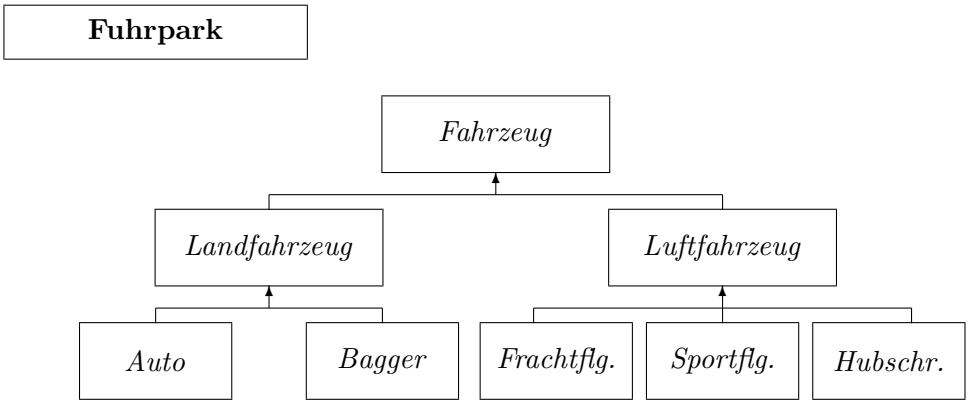
3 Erweiterte Programmierkonstrukte

3.1 Vererbung

Die Vererbung ist ein Unterkapitel der Objektorientierung. Mittels Vererbung und Klassen können Objekthierarchien gebildet werden. Hier werden die Attribute und Methoden einer Klasse in eine andere Klasse übernommen und gegebenenfalls ergänzt. Es muss keine Neudefinition der bereits vorhandenen Elemente passieren. Die Objekte einer höheren Hierarchiestufe sind dabei eine Generalisierung der Objekte der untergeordneten Hierarchiestufen und Objekte einer untergeordneten Hierarchiestufe sind eine Spezialisierung der übergeordneten Hierarchiestufe.

Ziel der Vererbung ist es, Programmiercode einzusparen und Änderungen schneller durchführen zu können.

Folgendes Beispiel zeigt eine Hierarchie innerhalb eines Fuhrparks.



Klassen können von anderen Klassen abgeleitet werden. Das bedeutet, die Funktionen und Attribute können in der abgeleiteten Klasse verwendet werden, ohne dass sie erneut implementiert werden müssen. Dies gilt jedoch nur für alle Attribute und Funktionen, die als *public* oder *protected* deklariert sind. Dieser Vorgang wird als Vererbung bezeichnet. Jede Klasse kann dabei

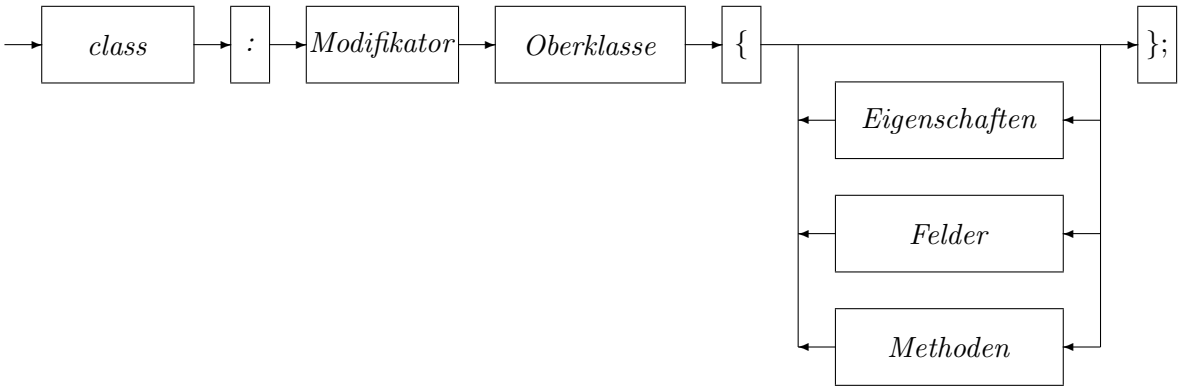
auch von vielen anderen Klassen erben und sie kann an beliebig viele Klassen vererben. Man sollte sich auf das erben von einer Klasse beschränken, um Fehler zu vermeiden.

Die erbende Klasse wird Unterklasse oder abgeleitete Klasse genannt. Die vererbende Klasse wird Oberklasse oder Basisklasse genannt.

Zugriffsmodifikatoren:

- *public* - Das gekennzeichnete Element ist über ein Objekt der Klasse, innerhalb der Klasse und in allen abgeleiteten Klassen und deren Objekten verfügbar.
- *protected* - Das gekennzeichnete Element ist nicht über ein Objekt der Klasse oder ein Objekt einer Unterklasse verfügbar. Es ist jedoch sowohl innerhalb der Klasse, als auch in allen abgeleiteten Klassen verfügbar.
- *private* - Das gekennzeichnete Element ist nicht über ein Objekt der Klasse oder ein Objekt der Unterklasse verfügbar. Es ist in der Klasse, jedoch in keiner abgeleiteten Klasse verfügbar.

Vererbung von Klassen



Eine abgeleitete Klasse wird definiert, indem hinter der gewöhnlichen Klassendefinition, direkt hinter dem Namen der neu definierten Klasse, durch einen Doppelpunkt getrennt, der Name der Oberklasse eingetragen wird. Vor

dem Namen dem Namen der Oberklasse wird noch ein Zugriffsmodifikator eingesetzt. Dieser sollte immer als *public* gewählt werden, sonst ändern sich die Zugriffsrechte auf die Variablen der Oberklassen. Es sind jedoch auch die Modifikatoren *protected* und *private* möglich. Diese werden hier aber nicht näher besprochen.

Durch die Vererbung können Methoden und Attribute in den Unterklassen verwendet werden. Die Unterklassen sind jedoch in der Regel eine Spezialisierung der Oberklasse. Das bedeutet, es werden zusätzlich neue Attribute und Methoden definiert.

Nun folgt ein Beispiel, in dem die Hierarchie aus dem letzten Beispiel implementiert wird.

```
#include <iostream>

using namespace std;

class Fahrzeug
{
   public:
    string farbe;
    int PS;
    int geschwindigkeit;

    Fahrzeug()
    {
       cout << "Ich bin das Fahrzeug";
    }
    void bewegen()
    {
       cout << "Ich bewege mich";
    }
};


class Landfahrzeug: public Fahrzeug
{
   public:
    int anzRaeder;
```

```
    Landfahrzeug()
    {
       cout << "Ich bin das Landfahrzeug";
    }
    void fahren()
    {
       cout << "Ich fahre";
    }
    void hupen()
    {
       cout << "Hupen";
    }
};

class Auto: public Landfahrzeug
{
   public:
    Auto()
    {
       cout << "Ich bin das Auto";
    }
    int mitfahrgelegenheiten;
    int kofferraumvolumen;
    void airbagoeffnen()
    {
       cout << "Airbag geht auf";
    }
};

class Bagger: public Landfahrzeug
{
   public:
    int anzschaufeln;
    Bagger()
    {
       cout << "Ich bin der Bagger";
    }
    void baggern()
    {
       cout << "Ich baggere";
```

```
    }
};

int main()
{
   Fahrzeug* f = new Fahrzeug;
   f->bewegen();
   Landfahrzeug* l = new Landfahrzeug;
   l->fahren();
   l->bewegen();
   l->farbe="blau";
   l->anzRaeder=4;
   cout << l->farbe;
   cout << l->anzRaeder;
   Bagger* b = new Bagger;
   b->baggern();
   b->bewegen();
   b->PS=80;
   cout << b->PS;
}
```

3.2 Polymorphie

In C# wird als Polymorphie bezeichnet, dass eine Variable einer Klasse (die Klasse ist dabei der Basistyp) nicht nur Objekte dieser Klasse beinhalten kann, sondern auch Objekte von Unterklassen. Es können nun alle Eigenschaften und Methoden des Basistyps, die vererbt werden, verwendet werden.

Stellen wir uns folgende Klassenhierarchie vor:

```
#include <iostream>

using namespace std;

class Tier
{
   public:
    void Futter()
    {
```

```
        cout << "Ich esse alles";
    }
};

class Katze: public Tier
{
};

class Hauskatze: public Katze
{
};
int main()
{
   Tier* cookie = new Hauskatze();
   Tier* muffin = new Katze();
   Tier* minchen = new Tier();
   cookie->Futter();
   muffin->Futter();
   minchen->Futter();
}
```

In diesen Beispiel gibt es drei verschiedene Variablen (*cookie, muffin, minchen*). Sie haben alle den gleichen Basistyp (*Tier*) und alle werden mit Objekten der Unterklassen oder Objekten der Klasse *Tier* belegt. Danach wird die Funktion *Futter()* aufgerufen. Sie ist in der Klasse Tier enthalten und kann damit auch für jedes Objekt einer Unterklasse benutzt werden.

3.2.1 Überschreiben

Beim Überschreiben wird eine Funktion in einer Unterklasse neu definiert, zum Beispiel weil sich das Verhalten des Objekts in der Unterklasse verändert. Die Funktion hat sowohl in der Oberklasse, als auch in der Unterklasse den gleichen Namen, wird allerdings in der Oberklasse mit dem Modifizierer *virtual* gekennzeichnet. In der Unterklasse wird die Funktion ganz gewöhnlich definiert.

Aufgrund von Polymorphie kann es sein, dass nicht eindeutig ersichtlich ist, welche Methode an ein Objekt *gebunden wird* (die aus der Oberklasse oder die aus der Unterklasse - das bedeutet, es ist nicht klar welche Methode über das Objekt aufgerufen wird). Beim Überschreiben wird dies anhand des *dy-*

namischen Typs entschieden. Das ist der Typ, der beim Konstruktoraufruf der Variablen zugewiesen wird.

```
#include <iostream>

using namespace std;

class Viereck
{
   protected:
    int a,b;
   private:
    int c,d;
   public:
    Viereck()
    {
       c=0;d=0;
    }
    Viereck (int a, int b, int c, int d)
    {
       this->a=a; this->b=b; this->c=c; this->d=d;
    }
    virtual int umfang()
    {
       return a+b+c+d;
    }
};


class Rechteck: public Viereck
{
   public:
    Rechteck (int a, int b)
    {
       this->a=a; this->b=b;
    }
    int umfang()
    {
       return 2*a+2*b;
    }
```

```
};
int main()
{
   Viereck* v = new Viereck(3,4,8,1); //16
   cout << v->umfang();
   Rechteck *r = new Rechteck(4,3); // 14
   cout << r->umfang();
   Viereck* r3 = new Rechteck(5,7); // 24
   cout << r3->umfang();
}
```

Die Klassen *Viereck* und *Rechteck* besitzen beide eine Funktion *umfang()*. In der Oberklasse Viereck ist sie als *virtual* gekennzeichnet. Dadurch kann sie in der Unterklasse (*Rechteck*) überschrieben werden.

In der *Main*-Funktion werden drei verschiedene Objekte erzeugt (*v,r,r3*). Dabei stimmt bei *v* und *r* der Basistyp mit dem Konstruktoraufruf überein. In der jeweils nächsten Zeile wird die Funktion *umfang* aufgerufen. Hierbei ist eindeutig, dass im Fall *v* die Funktion der Oberklasse aufgerufen wird und im Fall *r* die Funktion der Unterklasse.

Bei *r3* ist es nicht klar, welche Funktion aufgerufen wird. Wie oben beschrieben, ist beim Überschreiben die Bindung des Objekts dynamisch. Dies bedeutet, es wird anhand des zur Laufzeit zugewiesenen Typs entschieden, welche Funkton gebunden wird. Dies bedeutet somit, dass anhand des Konstruktoraufrufs entschieden wird und dieser bindet *r3* an die Methode der Unterklasse.

3.2.2 Verdecken

Beim Verdecken wird auch, wie beim Überschreiben, eine Methode in einer Unterklasse neu definiert. Dies ist bei jeder Methode der Oberklassen möglich. Sie müssen in den Oberklassen nicht speziell gekennzeichnet sein. Im Gegensatz zum Überschreiben wird allerdings *statisch gebunden*. Das bedeutet, der Aufruf einer Methode geschieht anhand des Basistyps.

```
#include <iostream>

using namespace std;

class Viereck
```

```
{
   protected:
    int a,b;
   private:
    int c,d;
   public:
    Viereck()
    {
       c=0;d=0;
    }
    Viereck (int a, int b, int c, int d)
    {
       this->a=a; this->b=b; this->c=c; this->d=d;
    }
    int umfang()
    {
       return a+b+c+d;
    }
};
class Rechteck: public Viereck
{
   public:
    Rechteck (int a, int b)
    {
       this->a=a; this->b=b;
    }
    int umfang()
    {
       return 2*a+2*b;
    }
};
int main()
{
   Viereck* v = new Viereck(3,4,8,1);
   cout << v->umfang(); // 16
   Rechteck* r = new Rechteck(4,3);
   cout << r->umfang(); // 14
   Viereck* r3 = new Rechteck(5,7);
   cout << r3->umfang(); // 12
}
```

Anders als beim Überschreiben ist in diesem Beispiel die Funktion *umfang* nicht durch *virtual* gekennzeichnet. Deshalb wird sie durch die Funktion der Unterklasse nur verdeckt. Das Objekt *r3* hat den Basistyp *Viereck*, wird jedoch durch den Konstruktoraufruf *Rechteck(5,7)* initialisiert. Beim Aufruf *r3.umfang()* wird die Funktion der Oberklasse benutzt, da die Funktion verdeckt ist und somit anhand des Basistyps die Funktion an das Objekt gebunden wird (statische Bindung). Als Ergebnis wird 12 zurückgegeben, weil *c* und *d* mit Null belegt sind.

3.3 Überladen

Das Überladen von Konstruktoren ist bereits bekannt (siehe Objektorientierung). In C++ können auch Methoden überladen werden. Überladene Methoden haben den gleichen Namen und den gleichen Rückgabetyp (bei gleichem Namen muss der Rückgabetyp auch gleich sein). Unterschiedlich sind dabei die Übergabeparameter. Diese dürfen sich sowohl in der Anzahl der Elemente unterscheiden und/oder in der Datentypwahl.

Überladung kann sowohl stattfinden, wenn Methoden mit dem gleichen Namen innerhalb einer Klasse definiert werden, als auch wenn der Methodenname in einer abgeleiteten Klasse verwendet wird.

```
class Viereck
{
   private:
    int c,d;
   protected:
    int a,b;
   public:
    Viereck()
    {
       c=0;d=0;
    }
    Viereck (int a, int b, int c, int d)
    {
       this->a=a; this->b=b; this->c=c; this->d=d;
    }
    void aendere(int a)
    {
       this->a=a;
```

```
    }
    void aendere(int a, int b, int c)
    {
       this->a=a;this->b=b;this->c=c;
    }
};
class Rechteck: public Viereck
{
   public:
    Rechteck (int a, int b)
    {
       this->a=a; this->b=b;
    }
};
int main()
{
   Rechteck *r = new Rechteck(3,8);
   r->aendere(1);
   r->aendere(4,5,8);
}
```

In der Klasse *Viereck* existieren zwei Methoden *aendere*. Eine hat einen Parameter und eine hat drei Parameter. Beide Methoden belegen natürlich die Variablen aus der Oberklasse. Über das Objekt der Unterklasse sind beide Methoden verfügbar.

3.4 Konstruktoraufrufe

In einer Vererbungshierarchie stellt sich die Frage, wenn ein Objekt einer Unterklasse erzeugt wird, welche Konstruktoren aufgerufen werden. Hierbei werden zuerst die Standardkonstruktoren (also die ohne Argumente) aller Klassen, die in der Hierarchie höher stehen, aufgerufen. Dabei wird mit dem in der Objekthierarchie am höchsten stehenden Konstruktor begonnen. Zum Schluss wird der Konstrukor der aufgerufenen Klasse ausgeführt. Auch wenn explizit ein spezieller Konstruktor aus einer Oberklasse aufgerufen wird, wird zuerst der Standardkonstruktor aufgerufen.

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

```
class Tier
    {
      public:

 Tier()
         {
                cout << "Ich bin ein Tier\n";
         }
         Tier(string name, string augenfarbe)
         {
          cout << "Ich bin ein Tier " + name
                                    + " " + augenfarbe << endl;
         }

          void sprechen()
          {
                 cout << "grrr\n";
          }
    };

    class Katze: public Tier
    {
      public:
Katze()
         {
                 cout << "Ich bin eine Katze\n";
         }
         Katze(string name, string augenfarbe)
         {
          cout << "Ich bin ein Katze " + name
                                  + " " + augenfarbe<<endl;
         }
    };

    class Hund: public Tier
    {

      public:
Hund()
         {
```

```
      Tier("Bello","blau");
              Tier::sprechen();
         }
         Hund(string n, string augen)
         {
        Tier(n,augen);
               sprechen();
         }
         void sprechen()
         {
               cout << "wauh\n";
         }
    };


int main()
{
            Tier *t = new Tier;
            Katze *k = new Katze;
            Katze *k1 = new Katze("Muffin","gelb");
            Tier *t1  = new Katze("Cookie", "gelb");
            Hund *d = new Hund;
            Hund *d1 = new Hund("Waui","braun");
}
```

```
Ausgabe:
Ich bin ein Tier
Ich bin ein Tier
Ich bin eine Katze
Ich bin ein Tier
Ich bin eine Katze Muffin gelb
Ich bin ein Tier
Ich bin eine Katze Cookie gelb
Ich bin ein Tier
Ich bin ein Tier Bello blau
grrr
Ich bin ein Tier
```

```
Ich bin ein Tier Waui braun
wauh
```

Übungen:

1. Erstellen Sie eine Klasse adresse. Sie soll die Attribute name, vorname, strasse, ort, plz, hausnummer, telefonnummer, sowie die get und set Methoden enthalten.
   Eine zweite Klasse adressbuch soll ein Array der Klasse adresse besitzen. Zusätzlich eine Methode sortiereadressbuch mit dem Übergabeparameter sortierattribut. Der Methode können die Zahlen 1 bis 7 übergeben werden, wobei 1 für name, 2 für vorname, 3 für strasse steht. Die Methode sortiert das Adressbuch nach diesem Parameter.

2. Was wird ausgegeben bei Eingabe 3?

```
#include <iostream>
#include <string>

using namespace std;

class Tier
{
  public:
      Tier()
      {
           cout << "Hier gibt es drei Tiere.\n";
      }

      void name()
      {
          cout << "Ich bin ein Tier\n";
      }

      virtual void farbe()
      {
          cout << "Ich weiss es nicht\n";
      }
};

class hase : public Tier
{
  public:
     hase()
       {
            cout<<"Hier gibt es nur Hasen.\n";
       }

       void name()
       {
            cout << "Ich bin ein Hase\n";
       }
       void farbe()
       {
            cout << "Ich bin braun\n";
       }
```

```
 };

class elefant : public Tier
 {
   public:
      elefant()
      {
           cout << "Hier gibt es nur Elefanten\n";
      }

      void name()
      {
          cout << "Ich bin ein Elefant\n";
      }

      void farbe()
      {
          cout << "Ich bin grau\n";
      }

 };


int main()
{
          Tier *neuesTier;
           int i;                                  Hier gibt es drei Tiere
   cin >> i;                                       Hier gibt es nur Elefanten
           if (i == 3)                             Ich bin ein Tier
           {                                       Ich bin grau
                neuesTier = new elefant();
           }

           else
           {
                neuesTier = new hase();
           }

           neuesTier->name();
           neuesTier->farbe();
```

```
}
```

3. Erstellen Sie folgende Klassen:

   - **Rechteck**: Sie soll die vier Instanzvariablen **a, b, x** und **y** enthalten. In a soll die Höhe des Rechtecks abgelegt werden, in b die Breite (Standardbelegung für beide: 1) und in x und y die Position der linken oberen Ecke, die standardmäßig an der Position (0,0) liegt. Die Klasse soll die Methoden **umfang()** und **flaeche()** als Instanzmethoden enthalten. umfang() soll den Umfang des Rechtecks und flaeche() den Flächeninhalt des Rechtecks zurückgeben. In abgeleiteten Klassen soll die Möglichkeit bestehen, diese Methoden zu überschreiben.
     Es sollen auch die Instanzmethoden **verschieben(int x, int y)** und **position()** existieren. verschieben(int x, int y) soll die linke obere Ecke des Rechtecks um die übergebenen Werte x bzw. y verändern (sie sollen zu den Instanzwerten addiert werden). position() soll die Position der linken oberen Ecke des Rechtecks auf der Konsole ausgeben. Diese Methoden sollen ebenfalls überschreibbar sein.
     Erstellen Sie folgende drei Konstruktoren: **Rechteck(), Rechteck(int a, int b)** und **Rechteck(int a, int b, int x, int y)**. Der parameterlose Konstruktor soll die Instanzvariablen mit den Standardwerten belegen. Die beiden anderen sollen den Instanzvariablen die Werte zuweisen, die in den Variablen mit denselben Namen gespeichert sind.
   - **Quadrat:** Sie soll von der Klasse Rechteck abgeleitet werden. Erstellen Sie die folgenden Konstruktoren: **Quadrat(), Quadrat(int a)** und **Quadrat(int a, int x, int y)**. Diese Konstruktoren sollen die entsprechenden Konstruktoren der Oberklasse Rechteck aufrufen (also soll z.B. Quadrat(int a) den Oberklassenkonstruktor Rechteck(int a, int b) aufrufen).
     Die geerbten Methoden müssen bei korrekter Programmierung der Konstruktoren nicht überschrieben werden.
   - **Quader:** Sie soll von der Klasse Rechteck abgeleitet werden. Ergänzen Sie die Klasse um die Instanzvariablen **c** und **z**. In c soll die Tiefe des Quaders abgelegt werden (Standardwert 1) und in z die dritte Koordinate (Standardwert: 0).
     Erstellen Sie die Konstruktoren **Quader(), Quader(int a, int b, int c)** und **Quader(int a, int b, int c, int x, int y, int z)**.

Diese sollen ebenfalls die entsprechenden Oberklassenkonstruktoren aufrufen und die erst in dieser Klasse definierten Instanzvariablen entsprechend belegen.
Überschreiben Sie die geerbte Methode flaeche() so, dass diese nun die Oberfläche des Quaders zurückgibt. Die geerbte Methode umfang() soll nun die Summe der Kantenlängen des Quaders zurückgeben. Die Methoden verschieben(int x, int y) soll durch eine Methode verschieben(int x, int y, int z) überladen werden und die Methode position() soll wiederum überschrieben werden, so dass diese die linke vordere Ecke des Quaders ausgibt (ihre Position ist in den Instanzvariablen gespeichert).
Ergänzen Sie Ihre Klasse um eine Methode **volumen()**, die das Volumen des Quaders zurückgibt.

4. Erstellen Sie folgendes Konstrukt:
   Weihnachten rückt näher und es ist Zeit, sich um die Wunschzettel für den Weihnachtsmann zu kümmern. Realisieren Sie einen Wunschzettel, auf den maximal zehen Positionen passen, für den Weihnachtsmann mit unterschiedlichen Artikeln wie folgt:

   - Buch: mit Autor und Titel
   - Elektronik-Artikel: mit Firma und Modell
   - Gewand mit Größe und folgenden Untertypen: Hose, Hemd

   Jeder Artikel hat einen Preis. Realisieren Sie Methoden für den Zugriff auf den Preis und auf die unterschiedlichen Eigenschaften der Artikel, Konstruktoren, die eine sinnvolle Konstruktion der Artikel erlauben, und Methoden toString() für die String-Darstellung der unterschiedlichen Artikel. Realisieren Sie dann eine Wunschzettel (Klasse Wishlist) zur Verwaltung der Geschenkartikel. Es soll möglich sein:

   - gewünschte Artikel anzufügen
   - den Gesamtbetrag zu berechnen
   - den gesamten Wunschzettel auf der Konsole auszugeben (printWishlist())
   - die Artikel einer bestimmten Art auf der Konsole auszugeben (z.B: printBooks(), printClothes(), ...).