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

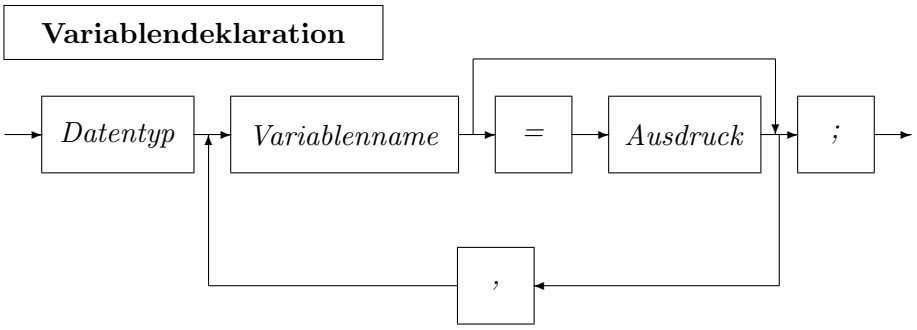
Um Werte zwischenzuspeichern, gibt es in Programmiersprachen Variablen. Hierdurch können beispielsweise Zwischenergebnisse gesichert werden oder Zähler programmiert werden.
Eine Variable reserviert ein Stückchen Arbeitsspeicher. Dieses Stückchen Arbeitspeicher ist nur für diese Variable reserviert. Es 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.

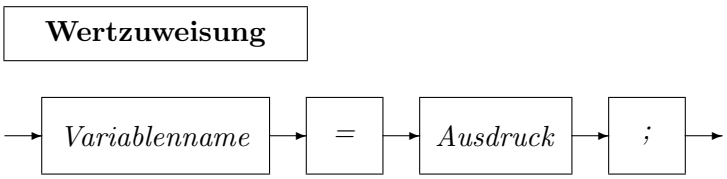
```
int x; // Deklaration von x
std::cout << "Bitte geben Sie eine Integerzahl ein";
// Aufforderung an den Benutzer
std::cin >> x; // der Benutzer gibt ein a ein
std:: cout << x; // 0 wird ausgegeben
```

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

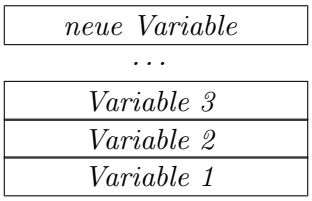
Jedes Programm bekommt bei seiner Ausführung ein Stück Arbeitspeicher zugewiesen. Dieses Stück Arbeitsspeicher wird vom Betriebssystem angefordert. Dieses Stück besteht aus fünf Teilen:

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

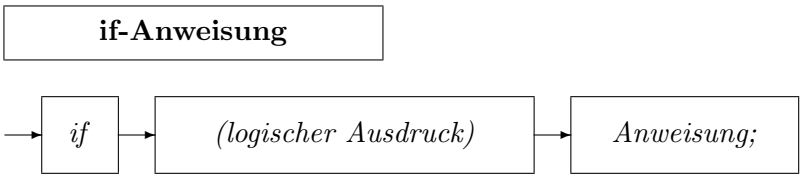
Die bisher kennen gelernten Anweisungen ergaben einen sequenziellen Programmablauf. Hierbei arbeitet der Computer eine Anweisung nach der anderen ab.

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.

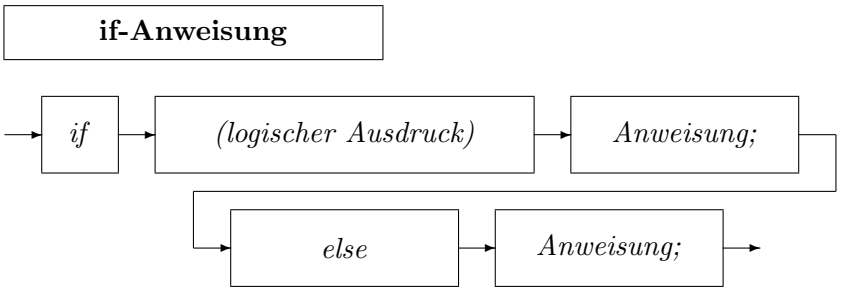



```
int main()
{
            cout << "Bitte geben Sie eine Zahl x ein:";
            cin >> x;
            if (x%2==1)
                x--;
            cout << x;
            return 0;
}
```

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.