Reader für den Einstieg in die Programmierung mit C++

1 Konventionen für die Formatierungen in diesem Dokument

- Normaler Text ist in Book Antiqua, Schriftgröße 11 geschrieben
- C++-Code wird mit der Schriftart **Courier New**, Schriftgröße 11 hervorgehoben
- Erläuterungen und Beschreibungen sind in *Arial*, Schriftgröße 10 und kursiv angegeben

2 Struktur eines C++-Programms

Das kleinste denkbare C++-Programm sieht folgendermaßen aus:

```
int main ()
{

}
```

Hauptfunktion „main“ vom Typ „int“ (int = integer)

Geschweifte Klammern schließen die Befehle innerhalb dieser Funktion (hier: main) ein

Dieses Minimalprogramm tut noch gar nichts, wie auch in Abbildung 1 zu sehen ist. Um das Programm dazu zu bewegen, z. B. die Berechnung von 3 + 4 durchzuführen, müssen Befehle zwischen die geschweiften Klammern geschrieben werden (siehe nächstes Listing). Dies ist nicht weiter schwer. Die einzige Syntax, auf die hier geachtet werden muss, ist das Semikolon hinter dem Ausdruck 3 + 4. Nur in Ausnahmefällen muss das Semikolon weggelassen werden (z. B. hinter einer Funktion, hinter { und einigen anderen Befehlen bzw. Zeichen), ansonsten wird vom C++-Compiler am Ende eines Befehls ein Semikolon verlangt.

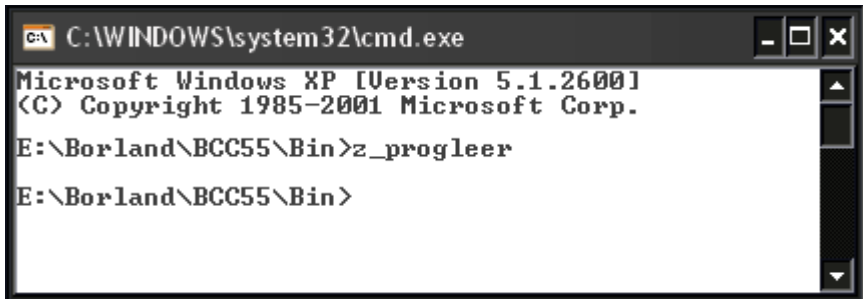

Abbildung 1: Minimalprogramm

```
int main ()
{
    3 + 4;
}
```

3 Zeichen auf dem Bildschirm ausgeben

Wenn wir dieses Programm ausführen, passiert anscheinend ebenfalls nichts. Doch der Schein trügt. Das Programm rechnet wirklich 3 + 4, nur wird das Ergebnis nicht ausgegeben. Wir benötigen also einen Befehl, der Zeichen auf dem Bildschirm ausgibt. Ein solcher Befehl ist **cout**. Die Syntax für diesen Befehl ist aus dem nächsten Listing zu entnehmen. Außerdem wird für diesen Befehl eine Bibliothek namens **iostream** benötigt. Zwar gibt es auch die Bibliothek **iostream.h**, doch diese ist nicht standardisiert! Sie ist nicht Teil des offiziellen C++ Sprachstandards. Somit ist die Portabilität für diese Klassen nicht garantiert (einige C++ Compiler können mit **iostream.h** nichts anfangen).
Bei der Verwendung der Bibliothek sollte ein sogenannter Namensraum festgelegt werden. Diesen nennt man standardmäßig **std**. Mit einer sogenannten **using** Direktive muss man diesen Namen nicht jedesmal vor den Befehl **cout** schreiben (z. B. **std::cout()**).

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

int main ()
{
     cout << "Endlich gibt es hier etwas zu sehen";
     system(„Pause“);
}
```

Bibliothek „iostream“ wird mit dem Befehl #include eingebunden

Ausgabestrom cout

Befehl, der das Programm erst nach einem Tastendruck beendet

Der Übersicht halber oder um sich Tipparbeit zu sparen, können mehrere Ausgabeobjekte direkt hintereinander, durch die doppelten Kleiner-Zeichen getrennt, aufgeführt werden (siehe folgendes Beispiel):

```
cout << "Die Summe der beiden Summanden ist: " << ergebnis;
```

Ein Zeilenumbruch lässt sich mit dem Befehl **endl** erzwingen, wie folgendes Beispiel zeigt:

```
cout << "Die Summe der beiden Summanden ist: " << ergebnis << endl;
```

Wird dieses Programm compiliert und anschließend gestartet, erscheint der von uns eingegebene Text „Endlich gibt es hier etwas zu sehen“ Für die Ausgabe des Ergebnisses von 3 + 4 müssen wir eine Variablen deklarieren. In die Variable wird dann das Ergebnis von 3 + 4 geschrieben und anschließend mit **cout** ausgegeben.

- **Speicher**
  Die Variable benötigt zum Ablegen ihrer Informationen immer Speicher. Über Lage und Größe des Speichers braucht sich der Programmierer normalerweise nicht zu kümmern. Der Compiler ermittelt die benötigte Größe aus dem Typ der Variablen.
- **Name**
  Die Variable wird im Programm über einen weitgehend frei wählbaren Namen angesprochen. Dieser Name identifiziert die Variable eindeutig. Verschiedene Namen bezeichnen verschiedene Variablen. C++ unterscheidet zwischen Groß- und Kleinschreibung. Der Name »Var« ist ein anderer als der Name »var«. Die Namensregeln finden Sie ausführlich ab Seite 32.
- **Typ**
  Der Typ einer Variablen bestimmt, welche Informationen abgelegt werden können. So kann eine Variable je nach ihrem Typ beispielsweise einen Buchstaben oder eine Zahl speichern. Der Typ bestimmt natürlich auch die Speichergröße, die benötigt wird. Der Typ bestimmt aber auch, welche Operationen auf die Variable angewendet werden können. Zwei Variablen, die Zahlen enthalten, können beispielsweise miteinander multipliziert werden. Enthalten die Variablen dagegen Texte, ist eine Multiplikation nicht besonders sinnvoll.

Text 1: Wichtige Eigenschaften von Variablen

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn

4 Variablen deklarieren

Variablen sind also Behälter, in denen das Programm Zahlen und Texte ablegt. Eine Variable hat drei wichtige Eigenschaften, die aus Text 1 zu entnehmen sind. Für die Berechnung von 3 + 4 genügt uns eine Variable vom Typ integer (**int** geschrieben). Dieser Typ steht für eine ganze Zahl mit Vorzeichen, aber ohne Nachkommastellen. Durch ein Leerzeichen abgesetzt, beginnt der Name der Variablen. Den Namen solltest du immer so wählen, dass du auf den Inhalt schließen kannst (vgl. Willemer 2010, S. 29).

4.1 Variablentyp festlegen

Mehrere Variablen des gleichen Typs können auch direkt hintereinander definiert werden:

```
int i, j=0, k;
```

In diesem Beispiel werden die Variablen i, j und k definiert, wobei j mit 0 initialisiert wird. Folgende Schreibweise ist gleichbedeutend:

```
int i;
int j=0;
int k;
```

Die Sprache C++ legt die Speicheranforderungen der meisten Typen nicht fest. Solche

Typ	Typische Größe	Typische Verwendung
char	1 Byte	Buchstaben, Zeichen und Ziffern
wchar_t	2 Bytes	Internationale Buchstaben, Zeichen und Ziffern
short int	2 Bytes	Zahlen für Nummerierungen oder Positionen
int	2 oder 4 Bytes	Standardgröße für ganze Zahlen
long int	4 oder 8 Bytes	Absehbar große Werte ohne Nachkommastellen
float	4 Bytes	Analog ermittelte Werte mit Nachkommastellen
double	8 Bytes	Berechnungen und höhere Preise
long double	12 Bytes	Berechnungen höherer Genauigkeit

Tabelle 1: Übliche Speichergröße verschiedener Typen

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn

Implementierungsdetails werden den Compilern überlassen. Lediglich die Qualitätsunterschiede zwischen den Typen werden gesichert. Das bedeutet, man kann sich darauf verlassen, dass ein **`short`** nicht größer als ein **`long`** ist (Willemer 2010, S. 30). Tabelle 1 (auf der vorherigen Seite) zeigt eine Übersicht von Typen und deren Speicheranforderungen.

4.2 Syntax bei der Variablendeklaration

Sowohl in C als auch in C++ wird zwischen Groß- und Kleinschreibung bzgl. Variablen unterschieden. Bei der Namensvergabe gelten außerdem die Namenskonventionen, die dem Syntaxgraph in Abbildung 2 zu entnehmen sind.

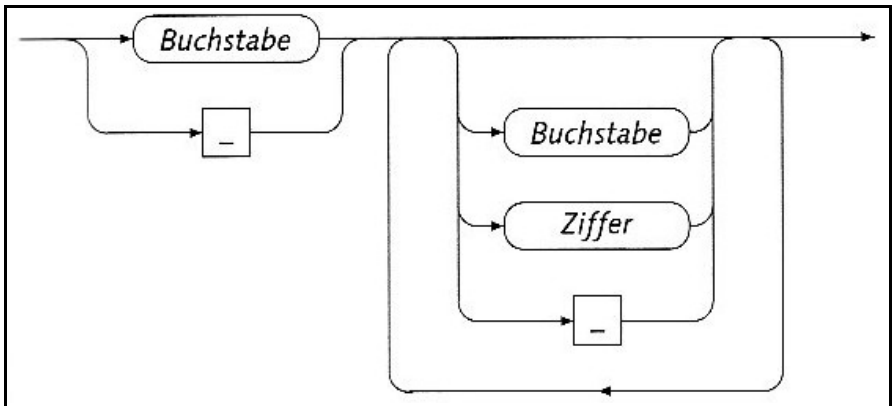

Abbildung 2: Syntaxgraph eines Namens

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn

Das folgende Listing zeigt ein Programm, das die Berechnung von 3 + 4 ausführt und das Ergebnis auf dem Bildschirm ausgibt (siehe Abbildung 3):

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

int ergebnis;

int main ()
{
  ergebnis = 3 + 4;
  cout << ergebnis;
}
```

Bei der Ausgabe von Variablen müssen keine Anführungszeichen gesetzt werden

Deklaration der Variable „ergebnis“ als integer

Aufgabe I:

Schreibe ein Programm, das das Ergebnis der Summe aus 3,5 und 12,2 ausgibt!

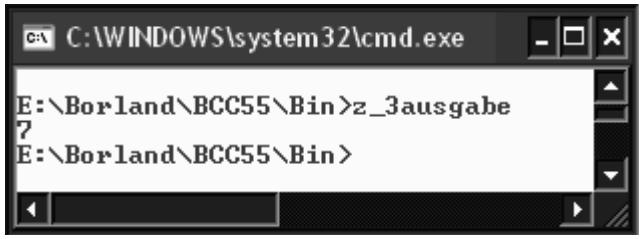

Abbildung 3: Ausgabe eines Ergebnisses auf dem Bildschirm

5 Eingabe von Werten und Zeichen

Berechnungen in den Quellcode direkt einzugeben macht nicht so viel Sinn. Man müsste dann für jede neue Berechnung den Quellcode entsprechend ändern und neu compilieren. Wir benötigen also einen Befehl, der Eingaben durch die Tastatur speichert. Um solche Eingaben lesen zu können, wird die Datenquelle **`cin`** auf die Variable umgeleitet. Der Eingabeoperator besteht diesmal aus zwei Größer-Zeichen, die quasi von **`cin`** auf die Variable, in der die Eingabe abgelegt werden soll, zeigen. Folgendes Listing zeigt ein Programm, das den Benutzer auffordert, zwei Summanden einzugeben, die anschließend verarbeitet werden:

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

float summand1, summand2, ergebnis;

int main ()
{
      cout << "Geben Sie den ersten Summanden ein: ";
      cin >> summand1;

      cout << "Geben Sie den zweiten Summanden ein: ";
      cin >> summand2;

      ergebnis = summand1 + summand2;

      cout << "Die Summe der beiden Summanden ist: " << ergebnis;
}
```

Aufgabe II:
Schreibe ein Programm mwst.cpp, das nach der Eingabe des Nettopreises die Mehrwertsteuer berechnet und ausgibt. Stell dir vor, du bist nun der Finanzminister und legst den Satz der Mehrwertsteuer selbst fest.

Aufgabe III:
Ergänze das Programm aus Aufgabe II dahingehend, dass auch der Bruttopreis ausgegeben wird.

6 Typumwandlung

(Auszug aus Willemer, Arnold (2010): Einstieg in C++. Bonn)

Hin und wieder ist es notwendig, Werte eines Typs in eine Variable eines anderen Typs zu speichern. So ergibt das Dividieren zweier ganzer Zahlen eine ganze Zahl. Wird 3 durch 4 geteilt, ergibt sich also 0 (mit dem Rest 3). Wollen Sie stattdessen aber als Ergebnis 0,75 haben, müssen Sie einen der beiden Operanden zum Fließkommawert wandeln, damit C++ auch die Fließkommadivision verwendet. Ein solches Umwandeln des Typs nennt man Casting[17].

Um einen Ausdruck eines bestimmten Typs in einen anderen umzuwandeln, gibt es zwei Schreibweisen. Die eine Schreibweise ist ein Erbstück der Sprache C. Dort wurde dem Ausdruck der Zieltyp in Klammern vorangestellt. In C++ wurde die Schreibweise eingeführt, dass auf den Namen des Typs eine Klammer folgt, in der der zu konvertierende Ausdruck steht.

```
int Wert;
Wert = (int)IrgendWas; // klassisches C-Casting
Wert = int(IrgendWas); // C++
```

Automatisch

Einige Umwandlungen führt C++ direkt durch, ohne darüber zu reden. Das geschieht immer dann, wenn die Umwandlung ohne jeden Informationsverlust des Inhalts gewährleistet ist. So wird eine `short`-Variable oder -Konstante direkt einer `long`-Variablen zugewiesen. Hier gibt es keine Interpretationsprobleme. Es kann jeder beliebige `short`-Wert in einer `long`-Variablen abgelegt werden. Der umgekehrte Weg ist schwieriger. Die Zahl 200.000 passt nicht in eine `short`-Variable, wenn diese nur aus zwei Bytes besteht. Hier wird der Compiler im Allgemeinen eine Warnung absetzen, dass relevante Informationen verloren gehen könnten.

Berechnungen

Besonders tückisch kann es sein, wenn der Compiler statt Fließkommazahlen ganzzahlige Werte verwendet. Als Beispiel soll ein klassischer Dreisatz verwendet werden. Drei Tomaten kosten 4 Euro. Wie viel kosten fünf Tomaten? Im Programm würde das wie folgt umgesetzt:

```
float SollPreis = (4/3)*5;
```

Der Inhalt der Variablen **SollPreis** dürfte überraschen: Er ist 5. Der Grund ist, dass der Compiler den Ausdruck `4/3` als Integer-Berechnung ausführt und die Nachkommastellen abschneidet. Also ist das Ergebnis der Division 1. Multipliziert mit 5 ergibt sich das oben genannte Ergebnis. Dennoch würden Sie eher erwarten, dass Sie an der Kasse 6,67 Euro zahlen müssen, und der Kaufmann wird sich Ihrer Ansicht gewiss anschließen. In solchen Fällen können Sie mit einer Typumwandlung eingreifen. Es muss mindestens ein Operand der Division zum `float`-Wert konvertiert werden, um eine `float`-Berechnung zu erzwingen. Nach der Anpassung ergibt die Berechnung die erwarteten 6.66667.

Das folgende Listing zeigt den Quellcode für das oben beschriebene Problem. Hier wird die Konstante 4 in den Variablentyp **`float`** umgewandelt und damit das richtige Ergebnis erzielt:

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

float SollPreis;

int main ()
{
      SollPreis = (float(4)/3)*5;
      cout << SollPreis;
}
```

Aufgabe IV:
Schreibe ein Programm, das den Notendurchschnitt einer Klassenarbeit in einer Schulklasse ausrechnet. Das Notensystem enthält in diesem Programm die Noten 1 bis 6. Die Noten, die jeweilige Anzahl und die Gesamtzahl der Schüler sollen vom Typ integer sein. Ein mögliches Erscheinungsbild des Programms zeigt der Screenshot in Abbildung 4.

7 Mathematische Operatoren

Neben den mathematischen Operatoren +, -, * und / gibt es in C++ weitere Operatoren bzw. Kurzschreibweisen einiger Rechenoperationen. Die Tabelle 2 zeigt alle mathematischen Operatoren mit ihrer Bedeutung und jeweils einem Beispiel. Eine besondere Rechenart ist die Modulo-Rechnung. Sie liefert den Rest einer ganzzahligen Division. Inkrementieren bedeutet, dass eine Zahl um 1 erhöht wird. Somit ist der in der Tabelle 2 dargestellte Operator ++ nur eine andere Schreibweise des folgenden Befehls:

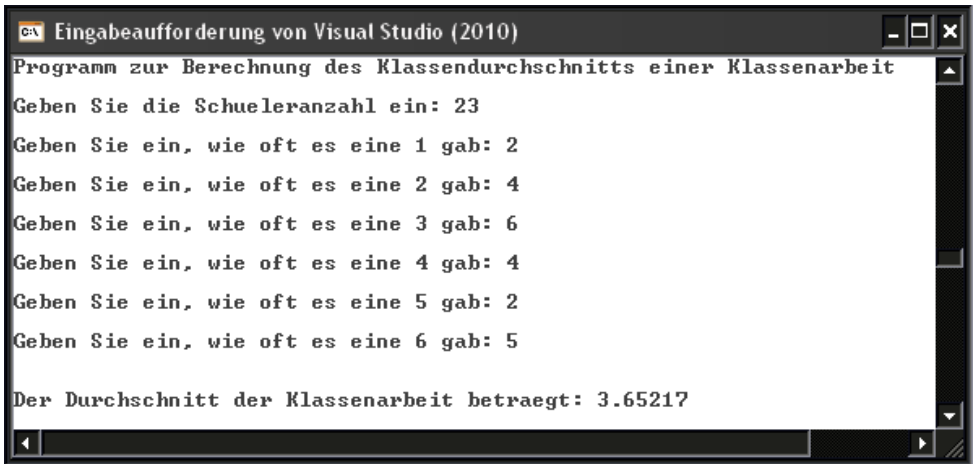

Abbildung 4: Screenshot einer möglichen Lösung der Aufgabe IV

```
a = a + 1;
```

Das Dekrementieren kann analog dazu auch folgendermaßen (umständlicher) programmiert werden:

```
a = a - 1;
```

Operator	Bedeutung	Beispiel
+	Addition	a = 11 + 5; (16)
–	Subtraktion	a = 11 – 5; (6)
*	Multiplikation	a = 11 * 5; (55)
/	Division	a = 11 / 5; (2)
%	Modulo	a = 11 % 5 (1)
++	Inkrementieren	++a; oder a++;
--	Dekrementieren	--a; oder a--;

Tabelle 2: Mathematische Operatoren

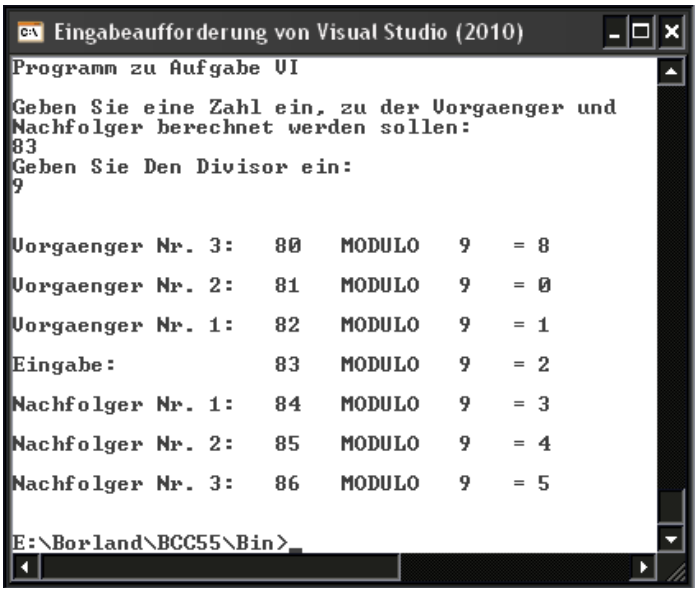

Abbildung 5: Screenshot einer möglichen Lösung zu Aufgabe VI

Aufgabe V:
Schreibe ein Programm, dass von einer ganzzahligen Zahl die drei Vorgänger und die drei Nachfolger berechnet und anzeigt.

Aufgabe VI:
Erweitere das Programm aus Aufgabe V dahingehend, dass der Benutzer zusätzlich einen Divisor (Teiler) eingeben kann. Anschließend soll das Programm zu jeder aufgeführten Zahl den Rest der jeweiligen Division angeben. Eine mögliche Lösung zu dieser Aufgabe könnte, wie in Abbildung 5 aussehen.

8 Speichern von Zeichenketten

Mithilfe der sogenannten Standardklasse[1] „**string**“ lassen sich Zeichenketten (z. B. Wörter, Sätze etc.) speichern. Um ein Objekt[2] vom Typ **string** zu definieren, muss zunächst die Bibliothek „**string**“ eingebunden werden. Wie bei der Bibliothek **iostream** muss ein Namensraum festgelegt werden. Wenn dieser Namensraum aufgrund der Benutzung von **iostream** schon festgelegt worden ist, muss es nicht noch einmal getan werden! Beim Anlegen des Objekts der Klasse **string** muss nicht angegeben werden, wie viele Zeichen reserviert werden sollen. Wird mehr Platz benötigt, sorgt das Objekt selbst dafür, dass es den erforderlichen Speicherbereich bekommt. Folgendes Programm gibt die Zeichenkette „Markus“ aus (vgl. WILLEMER 2010, S. 338ff.).

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

string vorname = "Markus";

int main()
{
    cout << vorname << endl;
}
```

8.1 Zeichenwiederholung mithilfe der Klasse string

Die Klasse „string“ verfügt über mehrere Konstruktoren. Auf diese Weise können Zeichenketten bereits bei der Definition mit Standardwerten vorbelegt werden. Soll eine Zeichenkette aus einer Folge des immer gleichen Buchstabens gebildet werden, werden dem Konstruktor zunächst die Anzahl und dann das Zeichen übergeben (Willemer 2010, S. 339f.):

```
string str(40, '-');
```

Dieser Ausdruck erzeugt einen String mit 40 Bindestrichen. Diese Variante ist hilfreich, wenn du beispielsweise eine Trennlinie ausgeben möchtest:

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

int main()
{
    cout << string(40, '-') << endl;
}
```

8.2 Zuweisung von Strings

Ein String kann durch eine Zuweisung direkt in eine andere Stringvariable kopiert werden. Dafür kannst du die Elementfunktion **assign()** verwenden. Als Parameter erhält sie den Wert, der dem String zugewiesen werden soll. Das sieht dann folgendermaßen aus:

```
string vorn = ”abc“;
string hinten = ”def“;
vorn.append(hinten);
```

8.3 Tauschen von Strings

Mit der Funktion **swap()** wird der Inhalt zweier string-Variablen auf direktem Weg ausgetauscht (siehe Abbildung 6) Der Parameter ist eine Referenz auf die Variable, mit der getauscht werden soll. In dem folgendem Beispiel werden die Wörter „REGAL“ und „LAGER“ getauscht:

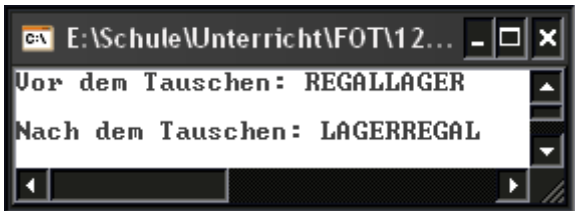

Abbildung 6: Beispiel für das Tauschen von Strings

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

string wort1 = "REGAL";
string wort2 = "LAGER";

int main()
{
    cout << "Vor dem Tauschen: " << wort1 << wort2 << endl << endl;
    wort1.swap(wort2);
    cout << "Nach dem Tauschen: " << wort1 << wort2 << endl << endl;
}
```

1 Dass an dieser Stelle von einer Standardklasse gesprochen wird, soll uns nicht weiter stören. **string** kann wie ein Variablentyp benutzt werden.

2 Da **string** eine Klasse ist, wird hier von einem Objekt gesprochen, in dem Zeichenketten gespeichert werden können. Das sogenannte Objekt kann aber genauso behandelt werden, wie eine Variable.

9 Programm-Ablaufstrukturen

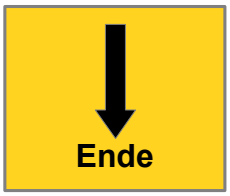

Abbildung 7: Programm mit linearer Struktur

9.1 Verzweigungen

Die von euch bisher geschriebenen Programme laufen stets in der Reihenfolge der codierten Programmanweisungen sequentiell ab. Es ist jedoch in vielen Fällen notwendig, dass ein Programm Operationen nur ausführt, wenn bestimmte Bedingungen erfüllt sind oder auch Operationen in bestimmten Situationen nicht ausführt und somit überspringt. Beispiele in denen das Programm in Abhängigkeit von einem Variableninhalt seinen Ablauf ändern sind z. B. folgende (vgl. KÜVELER 2009, S. 93):

- Ein Programm darf nicht dividieren, wenn der Nenner 0 ist
- Vor dem Ziehen einer Wurzel soll das Programm prüfen, ob der Operand negativ ist
- Der Spieler eines Programms soll nicht mehr auf Aliens schießen können, wenn der Munitionsvorrat gleich 0 ist

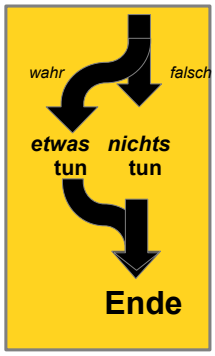

Abbildung 8: Programm mit einseitiger Verzweigung

Eine Möglichkeit, den Ablauf eines Programms in Abhängigkeit einer Variablen zu ändern, stellt der Befehl „**`if`**“ zur Verfügung.

Operator	Bedeutung
a == b	a gleich b?
a != b	a ungleich b?
a > b	a größer als b?
a >= b	a größer oder gleich b?
a < b	a kleiner als b?
a <= b	a kleiner oder gleich b?

Tabelle 3: Vergleich numerischer Werte

9.1.1 Die einseitige Verzweigung

Die Entscheidung zur Verzweigung ist abhängig vom aktuellen Wert eines logischen Ausdrucks, wie z. B. **`x < 5`**. Eine Übersicht aller Operatoren zum Vergleich von nummerischen Werten zeigt Tabelle 3. In dem folgenden Beispiel wird die Division nur durchgeführt, wenn der Divisor nicht 0 ist:

```
if (divisor != 0)
      ergebnis = dividend / divisor;
```

An dem Beispiel kann man die Syntax der if-Verzweigung gut erkennen. Dem Schlüsselwort **`if`** folgt eine Klammer, in der die Bedingung formuliert wird. Nach der Bedingung wird kein Semikolon gesetzt! Danach folgt die Anweisung, die nur durchgeführt wird, wenn die Bedingung wahr ist (siehe auch Abbildung 9). Sollen mehrere Anweisungen nach einer wahren Bedingung ausgeführt werden, fasst man diese in geschweiften Klammern zusammen, so wie im folgenden Beispiel (vgl. WILLEMER 2010, S. 72):

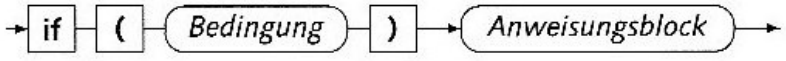

Abbildung 9: Syntaxgraph einer if-Verzweigung

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn

```
if (aepfel > 10)
    {
    preis = preis - rabatt;
    cout << "Sie haben einen Rabatt erhalten!" << endl << endl;
    }
cout << "Sie muessen " << preis << " Euro bezahlen";
```

In diesem Beispiel wird somit der Rabatt nur abgezogen und der neue Preis auf dem Bildschirm ausgegeben, wenn mehr als 10 Äpfel gekauft wurden. Die letzte Zeile hingegen steht außerhalb der geschweiften Klammern und wird deshalb immer interpretiert, auch wenn weniger als 10 Äpfel gekauft wurden.

Aufgabe VII:
Schreibe ein Programm, dass prüft, ob die Division ohne Rest möglich ist. Das Programm soll den Rest nur anzeigen, wenn einer vorhanden ist, ansonsten soll es nur das Ergebnis der Division ausgeben!

Aufgabe VIII:
Erweitere das Programm aus Aufgabe IV. Haben 30 % der Schüler eine schlechtere Note als 4, dann soll hinter der Durchschnittsnote in Klammern „Die Klassenarbeit muss wiederholt werden!“ stehen.

9.1.2 Die zweiseitige Verzweigung

Im Gegensatz zur einseitigen Verzweigung gibt es bei der zweiseitigen Verzweigung einen Alternativzweig. Dieser wird mit **`else`** eingeleitet und abgearbeitet, falls der logische Ausdruck (die Bedingung) falsch ergibt (siehe Abbildung 10).

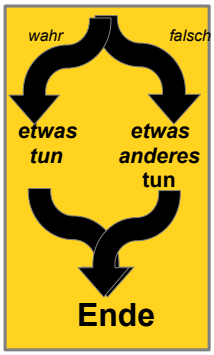

Abbildung 10: Programm mit zweiseitiger Verzweigung

Für die grafische Beschreibung solcher Verzweigungen gibt es auch eine genormte Darstellungsart nach DIN 66 261. Eine Beschreibung zu dieser Norm ist auf http://toosten.de zu finden, die Darstellungsart wird als Struktogramm bezeichnet. Die zweiseitige Verzweigung als Struktogramm zeigt Abbildung 11.

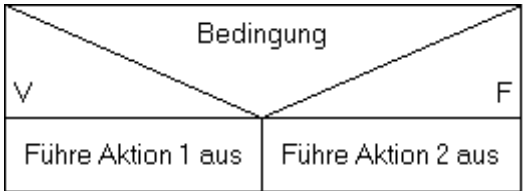

Abbildung 11: Struktogramm mit if-else-Verzweigung

Bei der einseitigen Verzweigung würde das Feld unter „F“ (steht für false) frei bleiben (siehe Abbildung 11) oder durch ein Ø-Symbol gekennzeichnet werden.

Ein Beispiel für eine zweiseitige Verzweigung zeigt nebenstehendes Listing:

```
                    .
                    .
                    .
/* Prüfung, ob der Divisor ungleich 0 ist.
   Bei Ja wird geteilt */
if (divisor != 0)
      {
      ergebnis = float (dividend) / divisor;
      cout << "Ergebnis: " << ergebnis;
      }

      else
         cout << "Der Divisor darf nicht 0 sein!";
                    .
                    .
                    .
```

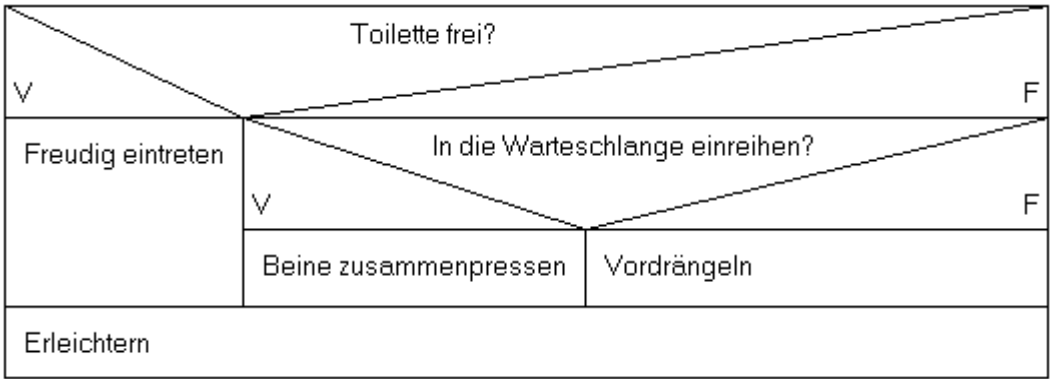

Abbildung 12: Verschachtelte if-else-Verzweigungen

```
                    .
                    .
                    .
if (frei == 1)
cout << "Freudig eintreten!" << endl;

else
   if (einreihen == 1)
   cout << "Beine zusammenpressen!" << endl;

   else
   cout << "Vordraengeln!" << endl;
                    .
                    .
                    .
```

Natürlich ist es auch möglich, mehrere Befehle nach dem **`else`** in einem Block mit geschweiften Klammern zusammenzufassen. Auch Verschachtelungen sind, wie das Struktogramm in Abbildung 12 zeigt, möglich. Das Struktogramm könnte, wie im linken Listing aufgezeigt, umgesetzt werden:

9.1.3 Die Mehrfach-Fallunterscheidung

Mit einer Mehrfach-Fallunterscheidung können bzgl. einer Variablen mehrere Bedingungen geprüft werden. Grundsätzlich sind als Mehrfach-Fallunterscheidung auch verschachtelte **`if-else`**-Anweisungen denkbar, mit einer **`case`**-Anweisung lässt sich eine Mehrfach-Fallunterscheidung jedoch effizienter konstruieren. Ein Beispiel, in dem die Nummern 1 – 7 als Tag ausgewertet werden, zeigt Abbildung 13. Eine Voraussetzung dafür, dass nur der Zweig, bei dem die Nummer mit der eingegebenen Nummer übereinstimmt ausgeführt wird, ist die Anweisung **`break`** am Ende der Anweisungen dieses bestimmten **`case`**s. Steht hinter den Anweisungen eines **`case`**s kein **`break`**, führt das Programm die Anweisungen des nächsten **`case`**s ebenfalls aus (auch, wenn die Prüfung falsch ergeben würde!). Wenn also in dem Beispiel in Abbildung 13 nirgendwo eine **`break`**-Anweisung stände, würde das Programm nach dem Ausführen der Anweisungen in **`case '1'`** alle folgenden **`case`**s ebenfalls ausführen.

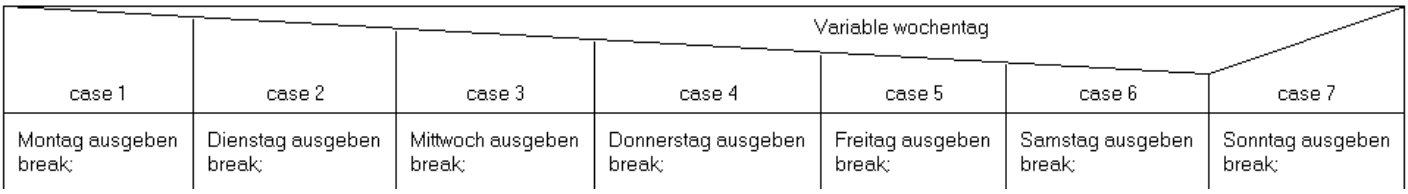

Abbildung 13: Mehrfach-Fallunterscheidung mit case

Gibt der Bediener nun einen Wert ein, der keiner **`case`**-Anweisung entspricht (z. B. 9), dann kann dieser Fall mit der **`default`**-Anweisung abgefangen werden. In dem folgenden Beispiel werden zwei **`float`**-Zahlen eingelesen und je nach eingegebenen Operator miteinander verknüpft. Anschließend wird das Ergebnis ausgegeben. Wird jedoch anstatt eines Operatoren irgendein anderes Zeichen eingegeben, werden die Anweisungen in dem **`default`**-Zweig ausgeführt.

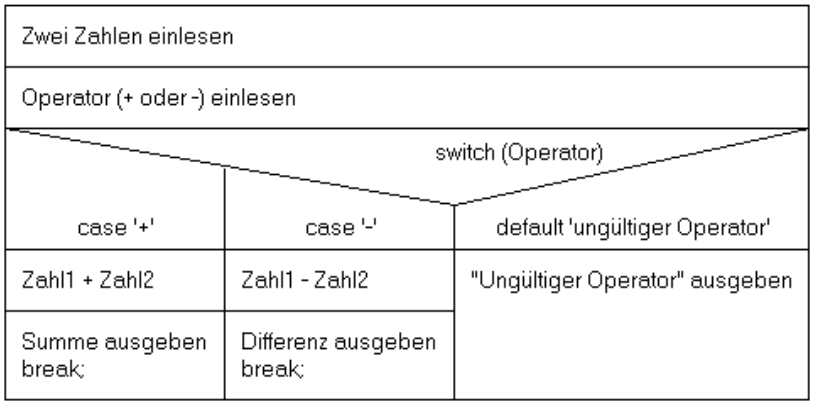

Abbildung 14: Mehrfach-Fallunterscheidung mit case und default

```
                        .
                        .
                        .
switch (op)
    {
    case '+':
         ergebnis = zahl1 + zahl2;
         cout << "Summe = " << ergebnis << endl;
         break;
    case '-':
         ergebnis = zahl1 - zahl2;
         cout << "Differenz = " << ergebnis << endl;
         break;
    default:
         cout << "Ungueltiger Operator!" << endl;
    }
                        .
                        .
                        .
```

Das Listing zum Struktogramm in Abbildung 14 könnte wie links abgebildet aussehen:

Aufgabe IX:
Erstelle ein Programm, das die vier Grundrechenarten beherrscht. Der Bediener soll zwei Zahlen eingeben können und danach einen Operatoren (+, - *, /) auswählen können, durch den die beiden Zahlen verknüpft werden. Bei der Berechnung des Quotienten soll der Fall, dass der Dividend (Zahl2) 0 ist, abgefangen werden. Schließlich soll der Benutzer eine Bildschirmausgabe mit der Rechnung und dem Ergebnis erhalten!

9.2 Schleifen

9.2.1 Die kopfgesteuerte Schleife `while`

Die einfachste Schleife wird durch das Schlüsselwort while eingeleitet. Die deutsche Übersetzung lautet „solange". Die **`while`**-Schleife wiederholt einen Vorgang solange, wie die Bedingung hinter dem **`while`** erfüllt ist (siehe Abbildung 15).

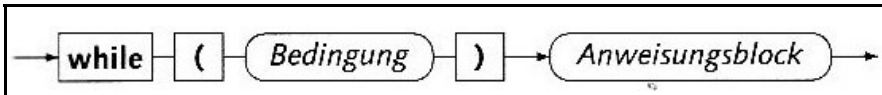

Abbildung 15: Syntaxgraph für eine while-Schleife

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn

Ein Beispiel für eine Schleife, mit der ein Menü realisiert wurde, ist folgende:

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

int menue;

int main()
{
    while (menue != 3)
    {
          cout << "--- Menue ---" << endl << endl
          << "1:  Menuepunkt 1" << endl << "2:  Menuepunkt 2" << endl
          << "3:  Programm beenden" << endl << "Bitte waehlen: ";

          cin >> menue;

          switch (menue)
          {
                case 1:
                     cout << endl << "Menuepunkt 1" << endl;
                     break;
                case 2:
                     cout << endl << "Menuepunkt 2" << endl;
                     break;
                case 3:
                     menue = 3;
                     break;
          }
    }
}
```

In dem Beispiel wird das Menü nur dann immer wieder aufgerufen, solange die Variable **`menue`** ungleich 3 ist. Wenn der Benutzer nun Menüpunkt 3 wählt, so wird im Menüpunkt 3 der Wert 3 in der Variablen **`menue`** gespeichert. Damit ist die Bedingung der **`while`**-Schleife nicht mehr erfüllt und das Programm steigt aus der Schleife aus. Ein Struktogramm einer `while`-Schleife ist in Abbildung 16 dargestellt (die Größer- bzw. Kleiner-als Zeichen in dem Struktogramm sollen nur darauf hinweisen, dass der Ausdruck zwischen ihnen nicht wörtlich aufgefasst werden soll).

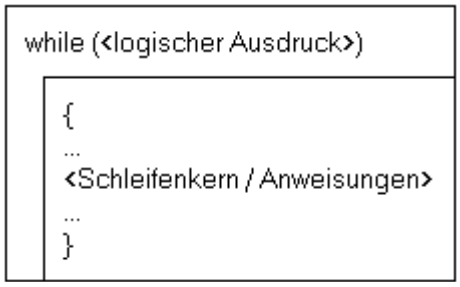

Abbildung 16: Struktogramm einer while-Schleife

9.2.2 Die fußgesteuerte Schleife mit `do` … `while`

In manchen Situationen ist die Prüfung am Kopf einer Schleife ungünstig. Manchmal möchte man eine Aktion durchführen und anschließend prüfen, ob sie noch einmal durchgeführt werden soll. Für diese Situation bietet sich die **`do` … `while`**-Schleife an. Sie prüft erst am Ende (fußgesteuert), ob die Bedingung wahr ist. Dass sich an der Syntax im Vergleich zur **`while`**-Schleife nicht viel ändert, zeigt der Syntaxgraph in Abbildung 17.

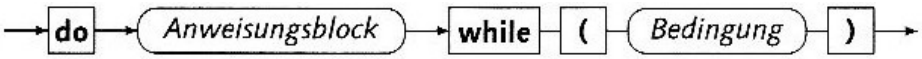

Abbildung 17: Syntaxgraph für eine do ... while-Schleife

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn. S. 96.

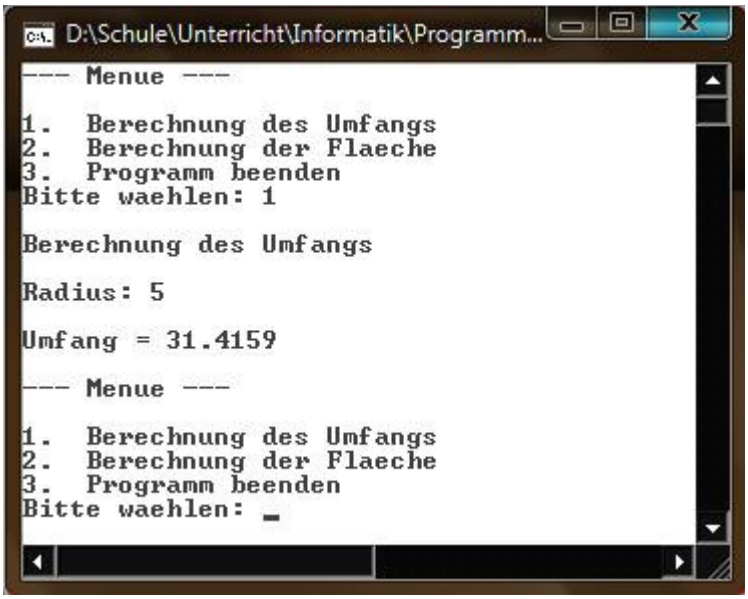

Abbildung 18: Screenshot zu Aufgabe IX

Aufgabe X:
Erweitern Sie das Programm aus Aufgabe VII! Nach der Berechnung soll der Benutzer mit „j“ für ja und „n“ für nein entscheiden können, ob er eine weitere Berechnung durchführen möchte. Ansonsten wird das Programm beendet! Entwickeln Sie zuerst das Struktogramm und danach den Quellcode!

Aufgabe XI:
Schreiben Sie ein Programm, dass den Umfang und die Fläche eines Kreises berechnen kann. Die Berechnungen sollen beliebig oft durchgeführt werden können. Das Programm könnte wie in Abbildung 18 aussehen. Fertigen Sie vor der Erstellung des Codes ein Struktogramm an!

9.2.3 Die for-Schleife

Die for-Schleife eröffnet dem Programmierer weitere Möglichkeiten gegenüber den while-Schleifen. Im Kopf der for-Schleife lassen sich neben der Bedingung noch der Startwert und die Zählkonstante eintragen (siehe Abbildung 19).

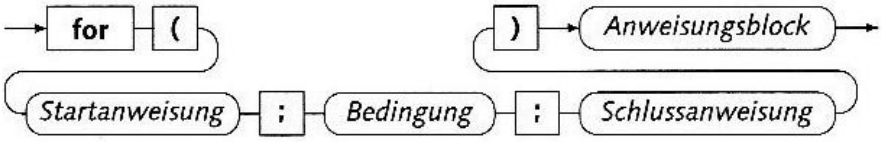

Abbildung 19: Syntaxgraph für for

Quelle: Willemer, Arnold (2010): Einstieg in C++. Bonn. S. 98.

Der folgende Auszug aus einem Quellcode enthält eine for-Schleife, die bei **`0`** beginnt zu zählen, den Wert für i bei jedem Durchlauf um **`1`** erhöht und bei **`i`** gleich **`7`** (also beim 8. Durchlauf) aufhört zu zählen. Außerdem wird die Schleife daraufhin verlassen.

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

int main()
{
      for (i=0; i<8; i++)
      {
            cout << i << endl;
      }
}
```

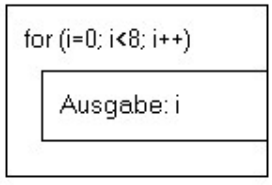

Abbildung 20: Struktogramm einer for-Schleife

Aufgabe XII:
Erstellen Sie ein Programm, das ein Sternchenquadrat beliebiger Größe ausgibt. Die nebenstehende Abbildung 21 zeigt ein mögliches Resultat.

a) Fertigen Sie das Struktogramm an!
b) Schreiben und testen Sie den Quellcode!

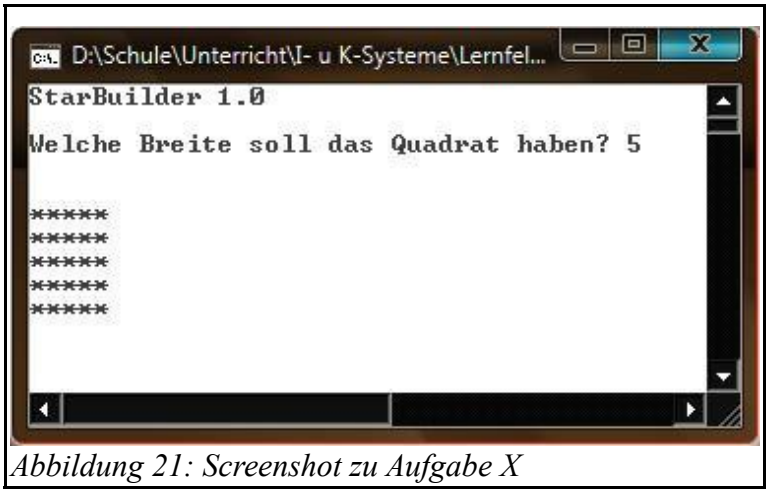

Abbildung 21: Screenshot zu Aufgabe X

10 Funktionen

Eine Funktion ist in C++ ein in sich abgeschlossener Programmteil, der an beliebiger Stelle aufgerufen werden kann (auch mehrmals hintereinander). Eine Funktion, die auch zwingend vorhanden sein muss, haben wir schon kennengelernt – die **`main()`**-Funktion. Obwohl die **`main()`**-Funktion immer hinter allen anderen Funktionen stehen muss, wird sie als erstes ausgeführt. Die anderen Funktionen, werden nur ausgeführt, wenn man sie in der **`main()`**-Funktion aufruft und das funktioniert so:

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

void linie()
{
     cout << "--------------------" << endl;
}

int main()
{
      linie();
      linie();
      linie();

cin.ignore();
}
```

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

int i=0;

int hochzaehlen()
{
     i++;
     return i;
}

int main()
{
      hochzaehlen();
      hochzaehlen();
      hochzaehlen();
      cout << hochzaehlen() << endl;
      hochzaehlen();
      cout << hochzaehlen() << endl;

cin.ignore();
}
```

In der **`main()`**-Funktion wird die Funktion **`linie()`** dreimal aufgerufen. Folglich gibt das Programm drei Linien aus. Der Ausdruck **`void`** kennzeichnet die Funktion als eine ohne Rückgabewert. Eine Funktion kann einen sogenannten Rückgabewert dadurch, dass sie aufgerufen wird ausgeben. Sinnvoll ist dies, wenn z. B. eine Berechnung in einem Programm mehrmals an verschiedenen Stellen benutzt werden soll. Soll die Funktion einen Rückgabewert haben, so sind zwei Veränderungen nötig. Erstens muss nicht **`void`**, sondern der Variablentyp vor dem Funktionsnamen stehen (z. B. **`int`**, **`float`** oder **`char`**) – je nachdem, um was für einen Rückgabewert es sich handelt. Zweitens muss am Ende der Funktion das Schlüsselwort **`return`** und der Variablenname mit anschließendem Semikolon stehen. Ein Beispiel, in dem die Funktion **`hochzaehlen()`** einen Wert bei jedem Aufruf inkrementiert und zurückgibt zeigt nebenstehendes Listing:

Die Funktion **`hochzaehlen()`** wird insgesamt sechsmal aufgerufen, aber nur beim vierten und sechsten Mal wird der Rückgabewert ausgegeben. Somit erscheinen auf dem Bildschirm die Zahlen 4 und 6.

Aufgabe XIII:
Schaffen Sie im Programm zu Aufgabe XI mehr Übersichtlichkeit, indem Sie die Berechnung von Umfang und Fläche jeweils in eine eigene Unterfunktion verschieben!

11 Dateiverarbeitung

Bisher sind alle Informationen, Eingaben etc. nach dem Schließen Ihres Programms verloren gegangen. Eine Möglichkeit der dauerhaften Speicherung von Informationen stellt die Dateiverarbeitung dar. Das Schreiben von Text in eine Datei wird ähnlich wie die Ausgabe auf dem Bildschirm realisiert:

```
f << "Hallo Welt";
```

Wörtliche Ausgaben in eine Datei werden in Anführungszeichen geschrieben, Werte von Variablen ohne Anführungszeichen. Für das Schreiben in eine Datei sind allerdings folgende Vorbereitungen nötig.

11.1 Schreiben in eine Datei

Die Bibliothek #include **<fstream>** muss für die Dateiverarbeitung eingebunden werden. Außerdem muss ein Stream erstellt werden und eine Datei, in die die Informationen geschrieben werden können. Diese Datei muss vor dem Schreibvorgang geöffnet und nach dem Schreibvorgang wieder geschlossen werden. Der nebenstehende Ausschnitt eines Quellcodes zeigt, wie „Hallo Welt“ in eine Datei geschrieben wird:

```
fstream f;
f.open("index.html", ios::out);

f << "Hallo Welt";

cout << endl << "Die Datei index.html wurde erstellt";
f.close();
```

11.2 Lesen aus einer Datei

Für das Lesen aus einer Datei muss die Datei ebenfalls geöffnet werden. Anschließend kann man z. B. mit dem Befehl getline die erste Zeile aus der Datei auslesen. Siehe folgenden Beispielcode:

```
fstream f;
f.open("geheim.txt", ios::in);

{
                    getline(f, s);

}
f.close();
```

Nähere Informationen zum Dateihandling können auf der folgenden Webseite nachgelesen werden:
http://www.willemer.de/informatik/cpp/fileop.htm