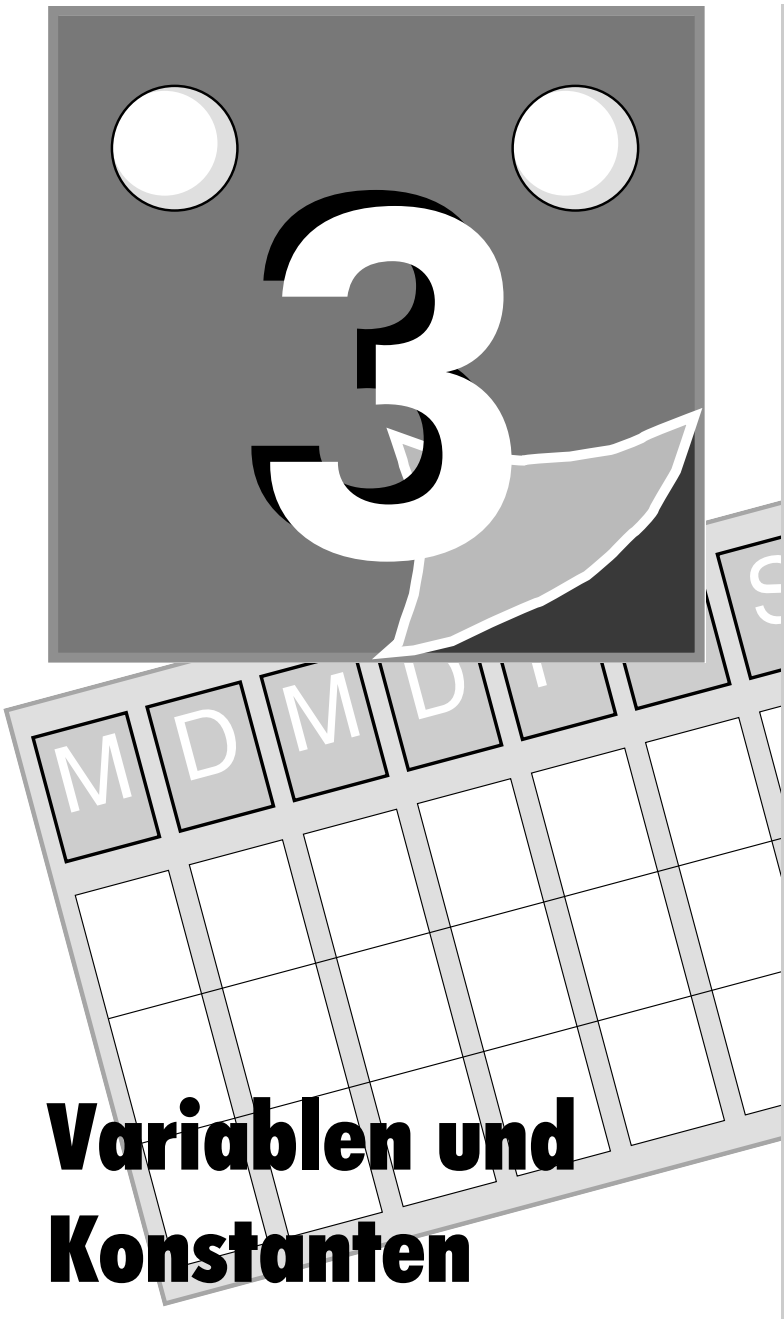

Variablen und Konstanten

Woche 1

Programme müssen auf irgendeine Weise die verwendeten Daten speichern. Variablen und Konstanten bieten verschiedene Möglichkeiten, diese Daten darzustellen und zu manipulieren.

Heute lernen Sie,

- wie man Variablen und Konstanten deklariert und definiert,
- wie man Variablen Werte zuweist und diese Werte manipuliert,
- wie man den Wert einer Variablen auf dem Bildschirm ausgibt.

Was ist eine Variable?

In C++ dient eine Variable dazu, Informationen zu speichern. Eine Variable ist eine Stelle im Hauptspeicher des Computers, in der man einen Wert ablegen und später wieder abrufen kann.

Man kann sich den Hauptspeicher als eine Reihe von Fächern vorstellen, die in einer langen Reihe angeordnet sind. Jedes Fach – oder Speicherstelle – ist fortlaufend numeriert. Diese Nummern bezeichnet man als Speicheradressen oder einfach als Adressen. Eine Variable reserviert ein oder mehrere Fächer, in denen dann ein Wert abgelegt werden kann.

Der Name Ihrer Variablen (zum Beispiel `meineVariable`) ist ein Bezeichner für eines der Fächer, damit man es leicht finden kann, ohne dessen Speicheradresse zu kennen. Abbildung 3.1 verdeutlicht dieses Konzept. Wie die Abbildung zeigt, beginnt unsere Variable `meineVariable` an der Speicheradresse 103. Je nach Größe von `meineVariable` kann die Variable eine oder mehrere Speicheradressen belegen.

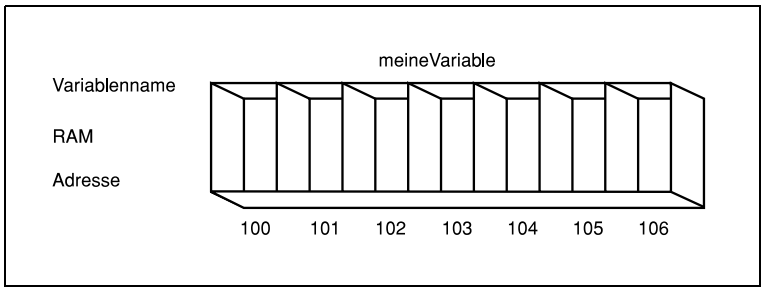

Abbildung 3.1: Schematische Darstellung des Hauptspeichers

RAM steht für Random Access Memory – Speicher mit wahlfreiem Zugriff. Bei Ausführung eines Programms wird dieses von der Datei auf dem Datenträger (zum Beispiel Festplatte, Diskette) in den RAM geladen. Des weiteren werden alle Variablen im RAM angelegt. Spricht ein Programmierer vom Speicher, meint er damit gewöhnlich den RAM.

Speicher reservieren

Wenn man in C++ eine Variable definiert, muß man dem Compiler nicht nur deren Namen, sondern auch den Typ der Variablen mitteilen – ob es sich zum Beispiel um eine Ganzzahl (Integer) oder ein Zeichen (Buchstaben, Ziffern etc.) handelt. Anhand dieser Information weiß der Compiler, um welche Art Variable e sich handelt und wieviel Platz im Speicher für die Aufnahme des Wertes der Variablen zu reservieren ist.

Jedes »Fach« im Speicher ist ein Byte groß. Wenn die erzeugte Variable vier Bytes benötigt, muß man vier Bytes im Speicher – oder vier Fächer – reservieren. Der Variablentyp (zum Beispiel `int` für Integer) teilt dem Compiler mit, wie viele Speicherplätze (oder Fächer) für diese Variable benötigt werden.

Da Computer Werte in Bits und Bytes darstellen und Speicher in Bytes gemessen wird, ist es wichtig, daß Sie diese Begriffe verstehen und verinnerlichen.

Größe von Integer-Werten

Jeder Variablentyp belegt im Speicher einen bestimmten Bereich, dessen Größe immer gleichbleibend ist, auf verschiedenen Computern aber unterschiedlich groß sein kann. Das heißt, ein Integer-Wert (Datentyp `int`) nimmt auf der einen Maschine zwei Bytes, auf einer anderen vielleicht vier ein – aber auf ein und demselben Computer ist dieser Platz immer gleich groß, tagein tagaus.

Eine Variable vom Typ `char` (zur Aufnahme von Zeichen) ist gewöhnlich ein Byte lang.

Über die Aussprache von `char` wird seit langem heiß diskutiert. Manche sprechen es aus wie »char« in Charakter, andere wiederum wie »char« in Charme. Auch die Version »care« wurde schon gehört. Selbstverständlich ist »char« wie in Charakter korrekt, denn so klingt es bei *mir*. Sie können jedoch dazu sagen, wie Ihnen beliebt.

Eine Ganzzahl vom Typ `short` belegt auf den meisten Computern zwei Bytes, eine Ganzzahl vom Typ `long` ist normalerweise vier Bytes lang, und eine Ganzzahl (ohne das Schlüsselwort `short` oder `long`) kann zwei oder vier Bytes einnehmen. Die Größe einer Ganzzahl wird vom Computer (16Bit oder 32Bit) oder vom Compiler bestimmt. Auf einem modernen 32-Bit-PC (Pentium) mit modernem Compiler (zum Beispiel

Visual C++4 oder höher) belegen die Ganzzahlen *vier* Bytes. Dieses Buch geht davon aus, daß Ganzzahlen vier Bytes groß sind. Das muß bei Ihnen jedoch nicht so sein. Mit dem Programm in Listing 3.1 läßt sich die genaue Größe der Typen auf Ihrem Computer bestimmen.

Listing 3.1: Die Größe der Variablentypen für einen Computer bestimmen

```
1:  #include <iostream.h>
2:
3:  int main()
4:  {
5:  cout << "Groesse eines int:\t\t"      << sizeof(int)    << " Bytes.\n";
6:  cout << "Groesse eines short int:\t"  << sizeof(short)  << " Bytes.\n";
7:  cout << "Groesse eines long int:\t"   << sizeof(long)   << " Bytes.\n";
8:  cout << "Groesse eines char:\t\t"     << sizeof(char)   << " Bytes.\n";
9:  cout << "Groesse eines float:\t\t"    << sizeof(float)  << " Bytes.\n";
10: cout << "Groesse eines double:\t\t"   << sizeof(double) << " Bytes.\n";
11: cout << "Groesse eines bool:\t\t"     << sizeof(bool)   << " Bytes.\n";
12:
13: return 0;
14: }
```


```
Groesse eines int:              4 Bytes.
Groesse eines short int:        2 Bytes.
Groesse eines long int:         4 Bytes.
Groesse eines char:             1 Bytes.
Groesse eines float:            4 Bytes.
Groesse eines double:           8 Bytes.
Groesse eines bool:             1 Bytes.
```

Die tatsächliche Anzahl der angezeigten Bytes kann auf Ihrem Computer abweichen.

Der größte Teil von Listing 3.1 sollte Ihnen bekannt vorkommen. Das Neue hier ist die Verwendung der Funktion `sizeof` in den Zeilen 5 bis 11. Die Funktion `sizeof` gehört zum Lieferumfang des Compilers und gibt die Größe des als Parameter übergebenen Objekts an. Beispielsweise wird in Zeile 5 das Schlüsselwort `int` an die Funktion `sizeof` übergeben. Mittels `sizeof` konnte ich feststellen, ob auf meinem Computer ein `int` gleich einem `long int` ist und 4 Byte belegt.

signed und unsigned

Alle genannten Typen kommen außerdem in zwei Versionen vor: mit Vorzeichen (`signed`) und ohne Vorzeichen (`unsigned`). Dem liegt der Gedanke zugrunde, daß man manchmal zwar negative Zahlen benötigt, manchmal aber nicht. Ganze Zahlen (`short` und `long`) ohne das Wort `unsigned` werden als `signed` (das heißt: vorzeichenbehaftet) angenommen. Vorzeichenbehaftete Ganzzahlen sind entweder negativ oder positiv, während ganze Zahlen ohne Vorzeichen (`unsigned int`) immer positiv sind.

Da sowohl für vorzeichenbehaftete als auch vorzeichenlose Ganzzahlen dieselbe Anzahl von Bytes zur Verfügung steht, ist die größte Zahl, die man in einem `unsigned int` speichern kann, doppelt so groß wie die größte positive Zahl, die man in einem `signed int` unterbringt. Ein `unsigned short int` kann Zahlen von 0 bis 65535 speichern. Bei einem `signed short int` ist die Hälfte der Zahlen negativ. Daher kann ein `signed short int` Zahlen im Bereich von -32768 bis 32767 darstellen. Sollte Sie dieses etwas verwirren, finden Sie in Anhang C eine ausführliche Beschreibung.

Grundlegende Variablentypen

In C++ gibt es weitere Variablentypen, die man zweckentsprechend in ganzzahlige Variablen (die bisher behandelten Typen), Fließkommavariablen und Zeichenvariablen einteilt.

Im Englischen verwendet man als Dezimalzeichen den Punkt, im Deutschen das Komma, deshalb auch der Begriff *Fließkommazahlen*. Leider orientiert sich C++ an der englischen Schreibweise und akzeptiert nur den Punkt als Dezimalzeichen und das Komma als Tausendertrennzeichen.

Die Werte von Fließkommavariablen lassen sich als Bruchzahlen ausdrücken – das heißt, es handelt sich um reelle Zahlen. Zeichenvariablen nehmen ein einzelnes Byte auf und dienen der Speicherung der 256 möglichen Zeichen und Symbole der ASCII- und erweiterten ASCII-Zeichensätze.

Der ASCII-Zeichensatz ist ein Standard, der die im Computer verwendeten Zeichen definiert. ASCII steht als Akronym für American Standard Code for Information Interchange (amerikanischer Standard-Code für den Informationsaustausch). Nahezu jedes Computer-Betriebssystem unterstützt ASCII. Daneben sind meistens weitere internationale Zeichensätze möglich.

Die in C++-Programmen verwendeten Variablentypen sind in Tabelle 3.1 aufgeführt. Diese Tabelle zeigt den Variablentyp, den belegten Platz im Speicher (Grundlage ist der Computer des Autors) und den möglichen Wertebereich, der sich aus der Größe des Variablentyps ergibt. Vergleichen Sie dazu die Ausgabe des Programms aus Listing 3.1.

Typ	Größe	Wert
bool	1 Byte	true oder false
unsigned short int	2 Byte	0 bis 65,535
short int	2 Byte	–32,768 bis 32,767
unsigned long int	4 Byte	0 bis 4,294,967,295
long int	4 Byte	–2,147,483,648 bis 2,147,483,647
int (16 Bit)	2 Byte	–32,768 bis 32,767
int (32 Bit)	4 Byte	–2,147,483,648 bis 2,147,483,647
unsigned int (16 Bit)	2 Byte	0 bis 65,535
unsigned int (32 Bit)	4 Byte	0 bis 4,294,967,295
char	1 Byte	256 Zeichenwerte
float	4 Byte	1.2e-38 bis 3.4e38
double	8 Byte	2.2e-308 bis 1.8e308

Tabelle 3.1: Variablentypen

Die Größen der Variablen können je nach verwendetem Computer und Compiler von denen aus der Tabelle 3.1 abweichen. Wenn die Ausgabe Ihres Computers für das Listing 3.1 mit der im Buch genannten übereinstimmt, sollten die Tabellenwerte auch für Ihren Compiler gelten. Gab es beim Listing 3.1 Unterschiede in der Ausgabe, sollten Sie in dem Compiler-Handbuch nachschlagen, welche Werte Ihre Variablentypen annehmen können.

Variablen definieren

Eine Variable erzeugt oder definiert man, indem man den Typ, mindestens ein Leerzeichen, den Variablennamen und ein Semikolon eintippt. Als Variablenname eignet sich nahezu jede Buchstaben-/Ziffernkombination, die allerdings keine Leerzeichen enthalten darf. Gültige Variablennamen sind zum Beispiel `x`, `J23qrsnf` und `meinAlter`. Gute Variablennamen sagen bereits etwas über den Verwendungszweck der Variablen aus und erleichtern damit das Verständnis für den Programmablauf. Die folgende Anweisung definiert eine Integer-Variable namens `meinAlter`:

```
int meinAlter;
```

Wenn Sie eine Variable deklarieren, wird dafür Speicherplatz allokiert (bereitgestellt). Was auch immer zu diesem Zeitpunkt sich in dem Speicherplatz befindet, stellt den Wert dieser Variablen dar. Wie Sie dieser Speicherposition einen neuen Wert zuweisen, werden Sie gleich erfahren.

Für die Programmierpraxis möchte ich Ihnen nahelegen, wenig aussagekräftige Namen wie `J23qrsnf` zu vermeiden und kurze aus einem Buchstaben bestehende Variablennamen (wie `x` oder `i`) auf Variablen zu beschränken, die nur kurz, für wenige Zeilen Code benötigt werden. Verwenden Sie ansonsten lieber Namen wie `meinAlter` oder `wie viele`. Diese Namen sind leichter zu verstehen, wenn Sie sich drei Wochen später kopfkratzend nach dem Sinn und Zweck Ihres Codes fragen.

Machen wir einen kleinen Test: Versuchen Sie anhand der ersten Codezeilen zu ergründen, was die folgenden Codefragmente bewirken:

Beispiel 1:

```
int main()
{
     unsigned short x;
     unsigned short y;
     unsigned short z;
     z = x * y;
     return 0;
}
```

Beispiel 2:

```
int main ()
{
     unsigned short Breite;
     unsigned short Laenge;
     unsigned short Flaeche;
     Flaeche = Breite * Laenge;
     return 0;
}
```

Wenn Sie dieses Programm kompilieren, wird Ihr Compiler eine Warnung ausgeben, daß diese Werte nicht initialisiert sind. Ich werde gleich darauf zu sprechen kommen, wie Sie dieses Problem lösen.

Ohne Zweifel ist die Aufgabe des zweiten Programms leichter zu erraten, und die Nachteile der längeren Variablennamen werden durch die leichtere Wartung des Programms mehr als wettgemacht.

Groß-/Kleinschreibung

C++ beachtet die Groß-/Kleinschreibung und behandelt demnach Großbuchstaben und Kleinbuchstaben als verschiedene Zeichen. Eine Variable namens `alter` unterscheidet sich von `Alter` und diese wiederum von `ALTER`.

Bestimmte Compiler gestatten es, die Abhängigkeit von der Groß-/Kleinschreibung zu deaktivieren. Das ist allerdings nicht zu empfehlen, da Ihre Programme dann von anderen Compilern womöglich nicht übersetzt werden können und andere C++-Programmierer mit Ihrem Code nicht klarkommen.

Für die Schreibweise von Variablennamen gibt es mehrere Konventionen. Unabhängig davon, für welche Sie sich entscheiden, ist es ratsam, innerhalb eines Programms bei der einmal gewählten Methode zu bleiben.

Viele Programmierer bevorzugen für Variablennamen Kleinbuchstaben. Wenn der Name aus zwei Wörtern besteht (zum Beispiel `mein Auto`), gibt es zwei übliche Konventionen: `mein_auto` oder `meinAuto`. Letztere Form wird auch als Kamel-Notation bezeichnet, da die Großschreibung im Wort selbst an einen Kamelhöcker erinnert.

Manche finden die Schreibweise mit dem Unterstrich (`mein_auto`) leichter zu lesen, andere wiederum versuchen den Unterstrich beim Tippen möglichst zu vermeiden. In diesem Buch finden Sie die sogenannte Kamel-Notation, in der das zweite und jedes weitere Wort mit einem Großbuchstaben beginnt: `meinAuto`, `derSchnelleBrauneFuchs` etc.

Viele fortgeschrittene Programmierer schreiben Ihren Code in der sogenannten Ungarischen Notation. Dieser Notation liegt der Gedanke zugrunde, daß jede Variable mit einem oder mehreren Buchstaben beginnt, die auf den Typ der Variablen verweisen. So wird ganzzahligen Variablen (Integer) ein kleines i vorangestellt oder Variablen vom Typ `long` ein kleines l. Andere Notationen verweisen auf Konstanten, globale Variablen, Zeiger und so weiter. Dies ist jedoch für die C-Programmierung von wesentlich größerer Bedeutung als für C++, da C++ die Erzeugung benutzerdefinierter Datentypen unterstützt (siehe Tag 6, »Klassen«), und von sich aus typenstrenger ist.

Schlüsselwörter

In C++ sind bestimmte Wörter reserviert, die man nicht als Variablennamen verwenden darf. Es handelt sich dabei um die Schlüsselwörter, mit denen der Compiler das Programm steuert. Zu den Schlüsselwörtern gehören zum Beispiel `if`, `while`, `for` und `main`. In der Dokumentation Ihres Compilers finden Sie eine vollständige Liste. Im allgemeinen fallen aussagekräftige Name für Variablen nicht mit Schlüsselwörtern zusammen. Eine Liste der C++-Schlüsselwörter finden Sie in Anhang B.

Was Sie tun sollten	... und was nicht
Definieren Sie eine Variable durch Angabe des Typs und dem sich anschließenden Variablennamen. Verwenden Sie aussagekräftige Variablennamen. Denken Sie daran, daß C++ die Groß-/Kleinschreibung berücksichtigt. Informieren Sie sich über die Anzahl der Bytes für jeden Variablentyp im Speicher und die möglichen Werte, die sich mit dem jeweiligen Typ darstellen lassen.	Verwenden Sie C++-Schlüsselwörter nicht als Variablennamen. Verwenden Sie keine vorzeichenlose (`unsigned`) Variablen für negative Zahlen.

Mehrere Variablen gleichzeitig erzeugen

In einer Anweisung lassen sich mehrere Variablen desselben Typs gleichzeitig erzeugen, indem man den Typ schreibt und dahinter die Variablennamen durch Kommata getrennt aufführt. Dazu ein Beispiel:

```
unsigned int meinAlter, meinGewicht;   //Zwei Variablen vom Typ unsigned int
long Flaeche, Breite, Laenge;          //Drei Variablen vom Typ long
```

Wie man sieht, werden `meinAlter` und `meinGewicht` gemeinsam als Variablen vom Typ `unsigned int` deklariert. Die zweite Zeile deklariert drei eigenständige Variablen vom Typ `long` mit den Namen `Flaeche`, `Breite` und `Laenge`. Der Typ (`long`) wird allen Variablen zugewiesen, so daß man in einer Definitionsanweisung keine unterschiedlichen Typen festlegen kann.

Werte an Variablen zuweisen

Einen Wert weist man einer Variablen mit Hilfe des Zuweisungsoperators (=) zu. Zum Beispiel formuliert man die Zuweisung des Wertes `5` an die Variable `Breite` wie folgt:

```
unsigned short Breite;
Breite = 5;
```

Die Typbezeichnung `long` ist eine verkürzte Schreibweise für `long int` und `short` für `short int`.

Diese Schritte kann man zusammenfassen und die Variable `Breite` bei ihrer Definition initialisieren:

```
unsigned short Breite = 5;
```

Die Initialisierung sieht nahezu wie eine Zuweisung aus, und bei Integer-Variablen gibt es auch kaum einen Unterschied. Bei der späteren Behandlung von Konstanten werden Sie sehen, daß man bestimmte Werte initialisieren muß, da Zuweisungen nicht möglich sind. Der wesentliche Unterschied besteht darin, daß die Initialisierung bei der Erzeugung der Variablen stattfindet.

Ebenso wie Sie mehrere Variable gleichzeitig definieren können, ist es auch möglich, mehr als eine Variable auf einmal zu erzeugen. Betrachten wir folgendes Beispiel:

```
//Erzeugung von zwei long-Variablen und ihre Initialisierung
long Breite = 5, Laenge = 7;
```

In diesem Beispiel wird die Variable `Breite` vom Typ `long` mit `5` und die Variable `Laenge` vom Typ `long` mit dem Wert `7` initialisiert. Sie können aber auch Definitionen und Initialisierungen mischen:

```
int meinAlter = 39, ihrAlter, seinAlter = 40;
```

Listing 3.2 zeigt ein vollständiges Programm, das Sie sofort kompilieren können. Es berechnet die Fläche eines Rechtecks und schreibt das Ergebnis auf den Bildschirm.

Listing 3.2: Einsatz von Variablen

```
1:   // Einsatz von Variablen
2:   #include <iostream.h>
3:
4:   int main()
5:   {
6:     unsigned short int Width = 5, Length;
7:     Length = 10;
8:
9:     // einen unsigned short int erzeugen und mit dem Ergebnis der
10:    // Multiplikation von Width und Length initialisieren
11:    unsigned short int Area  = (Width * Length);
12:
13:     cout << "Breite:  " << Width << "\n";
14:     cout << "Laenge:  "  << Length << endl;
15:     cout << "Flaeche: " << Area << endl;
```

```
16:   return 0;
17:  }
```


```
Breite:  5
Laenge:  10
Flaeche: 50
```

Zeile 2 enthält die `include`-Anweisung für die `iostream`-Bibliothek, die wir benötigen, um `cout` verwenden zu können. In Zeile 4 beginnt das Programm.

In Zeile 6 wird die Variable `Width` als vorzeichenloser `short int` definiert und mit dem Wert `5` initialisiert. Eine weitere Variable vom gleich Typ, `Length`, wird ebenfalls hier definiert, aber nicht initialisiert. In Zeile 7 erfolgt die Zuweisung des Wertes `10` an die Variable `Length`.

Zeile 11 definiert die Variable `Area` vom Typ `unsigned short int` und initialisiert sie mit dem Wert, der sich aus der Multiplikation von `Width` und `Length` ergibt. In den Zeilen 13 bis 15 erfolgt die Ausgabe der Variablenwerte auf dem Bildschirm. Beachten Sie, daß das spezielle Wort `endl` eine neue Zeile erzeugt.

typedef

Es ist lästig, zeitraubend und vor allem fehleranfällig, wenn man häufig `unsigned short int` schreiben muß. In C++ kann man einen Alias für diese Wortfolge mit Hilfe des Schlüsselwortes `typedef` (für Typendefinition) erzeugen.

Mit diesem Schlüsselwort erzeugt man lediglich ein Synonym und keinen neuen Typ (letzteres heben wir uns für den Tag 6, »Klassen«, auf). Auf das Schlüsselwort `typedef` folgt ein vorhandener Typ und danach gibt man den neuen Namen an. Den Abschluß bildet ein Semikolon. Beispielsweise erzeugt

```
typedef unsigned short int USHORT;
```

den neuen Namen `USHORT`, den man an jeder Stelle verwenden kann, wo man sonst `unsigned short int` schreiben würde. Listing 3.3 ist eine Neuauflage von Listing 3.2 und verwendet die Typendefinition `USHORT` anstelle von `unsigned short int`.

Listing 3.3: Demonstration von typedef

```
1:   // Listing 3.3
2:   // Zeigt die Verwendung des Schlüsselworts typedef
3:   #include <iostream.h>
4:
5:   typedef unsigned short int USHORT;        // mit typedef definiert
6:
7:   int main()
8:   {
9:     USHORT  Width = 5;
10:    USHORT Length;
11:    Length = 10;
12:    USHORT Area  = Width * Length;
13:    cout << "Breite:  " << Width << "\n";
14:    cout << "Laenge:  "  << Length << endl;
15:    cout << "Flaeche: " << Area <<endl;
16:     return 0;
17: }
```


```
Breite:  5
Laenge:  10
Flaeche: 50
```

Zeile 5 verwendet das mit `typedef` erzeugte Synonym `USHORT` für `unsigned short int`. Das Programm ist ansonsten mit Listing 3.2 identisch und erzeugt auch die gleichen Ausgaben.

Wann verwendet man short und wann long?

Neueinsteiger in die C++-Programmierung wissen oft nicht, wann man eine Variable als `long` und wann als `short` deklarieren sollte. Die Regel ist einfach: Wenn der in der Variablen zu speichernde Wert zu groß für seinen Typ werden kann, nimmt man einen größeren Typ.

Wie Tabelle 3.1 zeigt, können ganzzahlige Variablen vom Typ `unsigned short` (vorausgesetzt, daß sie aus 2 Bytes bestehen) nur Werte bis zu 65535 aufnehmen. Variablen vom Typ `signed short` verteilen ihren Wertebereich auf negative und positive Zahlen. Deshalb ist das Maximum eines solchen Typs nur halb so groß.

Obwohl Integer-Zahlen vom Typ `unsigned long` sehr große Ganzzahlen aufnehmen können (bis 4.294.967.295), hat auch dieser Typ einen begrenzten Wertebereich. Benötigt man größere Zahlen, muß man auf `float` oder `double` ausweichen und einen gewissen Genauigkeitsverlust in Kauf nehmen. Variablen vom Typ `float` oder `double` können zwar extrem große Zahlen speichern, allerdings sind auf den meisten Computern nur die ersten 7 bzw. 19 Ziffern signifikant. Das bedeutet, daß die Zahl nach dieser Stellenzahl gerundet wird.

Kürzere Variablen belegen weniger Speicher. Heute jedoch ist Speicher billig und das Leben kurz. Deshalb lassen Sie sich nicht davon abhalten, `int` zu verwenden, auch wenn damit 4 Byte auf Ihrem PC belegt werden.

Bereichsüberschreitung bei Integer-Werten vom Typ unsigned

Die Tatsache, daß Ganzzahlen vom Typ `unsigned long` nur einen begrenzten Wertebereich aufnehmen können, ist nur selten ein Problem. Aber was passiert, wenn der Platz im Verlauf des Programms zu klein wird?

Wenn eine Ganzzahl vom Typ `unsigned` ihren Maximalwert erreicht, schlägt der Zahlenwert um und beginnt von vorn. Vergleichbar ist das mit einem Kilometerzähler. Listing 3.4 demonstriert den Versuch, einen zu großen Wert in einer Variablen vom Typ `short int` abzulegen.

Listing 3.4: Speichern eines zu großen Wertes in einer Variablen vom Typ unsigned integer

```
1: #include <iostream.h>
2:  int main()
3:  {
4:     unsigned short int smallNumber;
5:     smallNumber = 65535;
6:     cout << "Kleine Zahl: " << smallNumber << endl;
7:     smallNumber++;
8:     cout << "Kleine Zahl: " << smallNumber << endl;
9:     smallNumber++;
10:    cout << "Kleine Zahl: " << smallNumber << endl;
11:   return 0;
12:  }
```


```
Kleine Zahl: 65535
Kleine Zahl: 0
Kleine Zahl: 1
```

Zeile 4 deklariert `smallNumber` vom Typ `unsigned short int` (auf dem Computer des Autors 2 Bytes für einen Wertebereich zwischen 0 und 65535). Zeile 5 weist den Maximalwert `smallNumber` zu und gibt ihn in Zeile 6 aus.

Die Anweisung in Zeile 7 inkrementiert `smallNumber`, das heißt, addiert den Wert 1. Das Symbol für das Inkrementieren ist ++ (genau wie der Name C++ eine Inkrementierung von C symbolisieren soll). Der Wert in `smallNumber` sollte nun 65536 lauten. Da aber Ganzzahlen vom Typ `unsigned short` keine Zahlen größer als 65535 speichern können, schlägt der Wert zu 0 um. Die Ausgabe dieses Wertes findet in Zeile 8 statt.

Die Anweisung in Zeile 9 inkrementiert `smallNumber` erneut. Es erscheint nun der neue Wert 1.

Bereichsüberschreitung bei Integer-Werten vom Typ signed

Im Gegensatz zu `unsigned` Integer-Zahlen besteht bei einer Ganzzahl vom Typ `signed` die Hälfte des Wertebereichs aus negativen Werten. Den Vergleich mit einem Kilometerzähler stellen wir nun so an, daß er bei einem positiven Überlauf vorwärts und bei negativen Zahlen rückwärts läuft. Vom Zählerstand 0 ausgehend erscheint demnach die Entfernung ein Kilometer entweder als `1` oder `-1`. Wenn man den Bereich der positiven Zahlen verläßt, gelangt man zur größten negativen Zahl und zählt dann weiter herunter bis Null. Listing 3.5 zeigt die Ergebnisse, wenn man auf die maximale positive Zahl in einem `signed short int` eine 1 addiert.

Listing 3.5: Addieren einer zu großen Zahl auf eine Zahl vom Typ signed int

```
1:  #include <iostream.h>
2:  int main()
3:  {
4:     short int smallNumber;
5:     smallNumber = 32767;
6:     cout << "Kleine Zahl: " << smallNumber << endl;
7:     smallNumber++;
8:     cout << "Kleine Zahl: " << smallNumber << endl;
9:     smallNumber++;
10:    cout << "Kleine Zahl: " << smallNumber << endl;
11:   return 0;
12:  }
```


```
Kleine Zahl: 32767
Kleine Zahl: -32768
Kleine Zahl: -32767
```

Zeile 4 deklariert `smallNumber` dieses Mal als `signed short int` (wenn man nicht explizit `unsigned` festlegt, gilt per Vorgabe `signed`). Das Programm läuft fast genau wie das vorherige, liefert aber eine andere Ausgabe. Um diese Ausgabe zu verstehen, muß man die Bit-Darstellung vorzeichenbehafteter (`signed`) Zahlen in einer Integer-Zahl von 2 Bytes Länge kennen.

Analog zu vorzeichenlosen Ganzzahlen findet bei vorzeichenbehafteten Ganzzahlen ein Umschlagen vom größten positiven Wert in den höchsten negativen Wert statt.

Zeichen

Zeichen-Variablen (vom Typ `char`) sind in der Regel 1 Byte groß und können damit 256 Werte (siehe Anhang C) aufnehmen. Eine Variable vom Typ `char` kann als kleine Zahl (0-255) oder als Teil des ASCII-Zeichensatzes interpretiert werden. ASCII steht für *American Standard Code for Information Interchange* (amerikanischer Standard-Code für den Informationsaustausch). Mit dem ASCII-Zeichensatz und seinem ISO-Gegenstück (International Standards Organization) können alle Buchstaben, Zahlen und Satzzeichen codiert werden.

Computer haben keine Ahnung von Buchstaben, Satzzeichen oder Sätzen. Alles was sie verstehen, sind Zahlen. Im Grunde genommen können Sie nur feststellen, ob genügend Strom an einem bestimmten Leitungspunkt vorhanden ist. Wenn ja, wird dies intern mit einer 1 dargestellt, wenn nicht mit einer 0. Durch die Kombination von Einsen und Nullen erzeugt der Computer Muster, die als Zahlen interpretiert werden können. Und diese Zahlen können wiederum Buchstaben und Satzzeichen zugewiesen werden.

Im ASCII-Code wird dem kleinen »a« der Wert `97` zugewiesen. Allen Klein- und Großbuchstaben sowie den Zahlen und Satzzeichen werden Werte zwischen 1 und 128 zugewiesen. Weitere 128 Zeichen und Symbole sind für den Computer-Hersteller reserviert. In der Realität hat sich aber der erweiterte IBM-Zeichensatz als Quasi-Standard durchgesetzt.

ASCII wird ausgesprochen wie »ASKI«.

Zeichen und Zahlen

Wenn Sie ein Zeichen, zum Beispiel »a«, in einer Variablen vom Typ `char` ablegen, steht dort eigentlich eine Zahl zwischen 0 und 255. Der Compiler kann Zeichen (dargestellt durch ein einfaches Anführungszeichen gefolgt von einem Buchstaben, einer Zahl oder einem Satzzeichen und einem abschließenden einfachen Anführungszeichen) problemlos in ihren zugeordneten ASCII-Wert und wieder zurück verwandeln.

Die Wert/Buchstaben-Beziehung ist zufällig. Daß dem kleinen »a« der Wert 97 zugewiesen wurde, ist reine Willkür. So lange jedoch, wie jeder (Tastatur, Compiler und Bildschirm) sich daran hält, gibt es keine Probleme. Sie sollten jedoch beachten, daß zwischen dem Wert 5 und dem Zeichen »5« ein großer Unterschied besteht. Letzteres hat einen Wert von 53, so wie das »a« einen Wert von 97 hat.

Listing 3.6: Ausdrucken von Zeichen auf der Basis von Zahlen

```
1:  #include <iostream.h>
2:   int main()
3:   {
4:      for (int i = 32; i<128; i++)
5:         cout << (char) i;
6:      return 0;
7:   }
```


```
!"#$%'()*+,./0123456789:;<>?@ABCDEFGHIJKLMNOP
_QRSTUVWXYZ[\]^'abcdefghijklmnopqrstuvwxyz{|{~
```

Dieses einfache Programm druckt die Zeichenwerte für die Integer 32 bis 127.

Besondere Zeichen

Der C++-Compiler kennt einige spezielle Formatierungszeichen. Tabelle 3.2 listet die geläufigsten auf. In Ihrem Code geben Sie diese Zeichen mit einem vorangestellten Backslash (auch Escape-Zeichen genannt) ein. Um zum Beispiel einen Tabulator in Ihren Code mit aufzunehmen, würden Sie ein einfaches Anführungszeichen, den Backslash, den Buchstaben t und ein abschließendes einfaches Anführungszeichen eingeben.

```
char tabZeichen = '\t';
```

Dies Beispiel deklariert eine Variable vom Typ `char` und initialisiert sie mit dem Zeichenwert `\t`, der als Tabulator erkannt wird. Diese speziellen Druckzeichen werden benötigt, wenn die Ausgabe entweder auf dem Bildschirm, in eine Datei oder einem anderen Ausgabegerät erfolgen soll.

Ein Escape-Zeichen ändert die Bedeutung des darauf folgenden Zeichens. So ist das Zeichen n zum Beispiel nur der Buchstabe n. Wird davor jedoch ein Escape-Zeichen gesetzt (\), steht das Ganze für eine neue Zeile.

Zeichen	Bedeutung
\n	Neue Zeile
\t	Tabulator
\b	Backspace
\"	Anführungszeichen
\'	Einfaches Anführungszeichen
\?	Fragezeichen
\\	Backslash

Tabelle 3.2: Variablentypen

Konstanten

Konstanten sind ebenso wie Variablen benannte Speicherstellen. Während sich Variablen aber ändern können, behalten Konstanten – wie der Name bereits sagt – immer ihren Wert. Sie müssen Konstanten bei der Erzeugung initialisieren und können ihr dann später keinen neuen Wert zuweisen.

Literale Konstanten

C++ kennt zwei Arten von Konstanten: *literale* und *symbolische*.

Eine literale Konstante ist ein Wert, den man direkt in das Programm an der Stelle des Vorkommens eintippt. In der Anweisung

```
int meinAlter = 39;
```

ist `meinAlter` eine Variable vom Typ `int`, während `39` eine literale Konstante bezeichnet. Man kann `39` keinen Wert zuweisen oder diesen Wert ändern.

Symbolische Konstanten

Eine symbolische Konstante wird genau wie eine Variable durch einen Namen repräsentiert. Allerdings läßt sich im Gegensatz zu einer Variablen der Wert einer Konstanten nicht nach deren Initialisierung ändern.

Wenn Ihr Programm eine Integer-Variable namens `Studenten` und eine weitere namens `Klassen` enthält, kann man die Anzahl der Studenten berechnen, wenn die Anzahl der Klassen bekannt ist und man weiß, daß 15 Studenten zu einer Klasse gehören:

```
Studenten = Klassen * 15;
```

Das Symbol * bezeichnet eine Multiplikation.

In diesem Beispiel ist `15` eine literale Konstante. Der Code wäre leichter zu lesen, zu verstehen und zu warten, wenn man für diesen Wert eine symbolische Konstante setzt:

```
Studenten = Klassen * StudentenProKlasse
```

Wenn man später die Anzahl der Stunden pro Klasse ändern möchte, braucht man das nur in der Definition der Konstanten `StudentenProKlasse` vorzunehmen, ohne daß man alle Stellen ändern muß, wo man diesen Wert verwendet hat.

Es gibt zwei Möglichkeiten, eine symbolische Konstante in C++ zu deklarieren. Die herkömmliche und inzwischen veraltete Methode erfolgt mit der Präprozessor-Direktiven `#define`.

Konstanten mit #define definieren

Um eine Konstante auf die herkömmliche Weise zu definieren, gibt man ein:

```
#define StudentenProKlasse 15
```

Beachten Sie, daß `StudentenProKlasse` keinen besonderen Typ (etwa `int` oder `char`) aufweist. `#define` nimmt eine einfache Textersetzung vor. Der Präprozessor schreibt an alle Stellen, wo `StudentenProKlasse` vorkommt, die Zeichenfolge `15` in den Quelltext.

Da der Präprozessor vor dem Compiler ausgeführt wird, kommt Ihr Compiler niemals mit der symbolischen Konstanten in Berührung, sondern bekommt immer die Zahl `15` zugeordnet.

Konstanten mit const definieren

Obwohl `#define` funktioniert, gibt es in C++ eine neue, bessere und elegantere Lösung zur Definition von Konstanten:

```
const unsigned short int StudentenProKlasse = 15;
```

Dieses Beispiel deklariert ebenfalls eine symbolische Konstante namens `StudentenProKlasse`, dieses Mal ist aber `StudentenProKlasse` als Typ `unsigned short int` definiert. Diese Version bietet verschiedene Vorteile. Zum einen läßt sich der Code leichter warten und zum anderen werden unnötige Fehler vermieden. Der größte Unterschied ist der, daß diese Konstante einen Typ hat und der Compiler die zweckmäßige – sprich typgerechte – Verwendung der Konstanten prüfen kann.

Konstanten können nicht während der Ausführung des Programms geändert werden. Wenn Sie gezwungen sind, die Konstante `studentsPerClass` zu ändern, müssen Sie den Code ändern und neu kompilieren.

Was Sie tun sollten	... und was nicht
Achten Sie darauf, daß Ihre Zahlen nicht größer werden als der verwendete Integer-Datentyp erlaubt, damit es nicht beim Überlauf zu inkorrekten Werten kommt. Verwenden Sie aussagekräftige Variablennamen, die auf deren Verwendung hinweisen.	Vermeiden Sie die Bezeichnung `int`. Verwenden Sie statt dessen `short` oder `long`, um anzuzeigen, mit welcher Zahlengröße Sie rechnen. Verwenden Sie keine C++-Schlüsselwörter als Variablennamen

Aufzählungstypen

Mit Hilfe von Aufzählungskonstanten (`enum`) können Sie neue Typen erzeugen und dann Variablen dieser Typen definieren, deren Werte auf einen bestimmten Bereich beschränkt sind. Beispielsweise kann man `FARBE` als Aufzählung deklarieren und dafür fünf Werte definieren: `ROT`, `BLAU`, `GRUEN`, `WEISS` und `SCHWARZ`.

Die Syntax für Aufzählungstypen besteht aus dem Schlüsselwort `enum`, gefolgt vom Typennamen, einer öffnenden geschweiften Klammer, einer durch Kommata getrennte Liste der möglichen Werte, einer schließenden geschweiften Klammern und einem Semikolon. Dazu ein Beispiel:

```
enum FARBE { ROT, BLAU, GRUEN, WEISS, SCHWARZ };
```

Diese Anweisung realisiert zwei Aufgaben:

`FARBE` ist der Name der Aufzählung, das heißt, ein neuer Typ.

`ROT` wird zu einer symbolischen Konstanten mit dem Wert 0, `BLAU` zu einer symbolischen Konstanten mit dem Wert 1, `GRUEN` zu einer symbolischen Konstanten mit dem Wert 2 usw.

Jeder Aufzählungskonstanten ist ein Integer-Wert zugeordnet. Wenn man nichts anderes festlegt, weist der Compiler der ersten Konstanten den Wert 0 zu und numeriert die restlichen Konstanten fortlaufend durch. Jede einzelne Konstante läßt sich aber auch mit einem bestimmten Wert initialisieren, wobei die Werte der nicht initialisierten Konstanten immer um 1 höher sind als die Werte ihres Vorgängers. Schreibt man daher

```
enum FARBE { ROT=100, BLAU, GRUEN=500, WEISS, SCHWARZ=700 };
```

erhält `ROT` den Wert `100`, `BLAU` den Wert `101`, `GRUEN` den Wert `500`, `WEISS` den Wert `501` und `SCHWARZ` den Wert `700`.

Damit können Sie Variablen vom Typ `FARBE` definieren, denen dann allerdings nur einer der Aufzählungswerte (in diesem Falle `ROT`, `BLAU`, `GRUEN`, `WEISS` oder `SCHWARZ` oder die Werte `100`, `101`, `500`, `501` oder `700`) zugewiesen werden kann. Sie können Ihrer Variablen `FARBE` beliebige Farbwerte zuweisen, ja sogar beliebige Integer-Werte, auch wenn es keine gültige Farbe ist. Ein guter Compiler wird in einem solchen Fall jedoch eine Fehlermeldung ausgeben. Merken Sie sich, daß Aufzählungsvariablen vom Typ `unsigned int` sind und daß es sich bei Aufzählungskonstanten um Integer-Variablen handelt. Es ist jedoch von Vorteil, diesen Werten einen Namen zu geben, wenn Sie mit Farben, Wochentagen oder ähnlichen Wertesätzen arbeiten. In Listing 3.7 finden Sie ein Programm, das eine Aufzählungskonstante verwendet.

Listing 3.7: Ein Beispiel zur Verwendung von Aufzählungskonstanten

```
1: #include <iostream.h>
2: int main()
3: {
4:    enum Days { Sunday, Monday, Tuesday,
5:                Wednesday, Thursday, Friday, Saturday };
6:    int choice;
7:    cout << " Geben Sie einen Tag  ein (0-6): ";
8:    cin >> choice;
9:    if (choice == Sunday || choice == Saturday)
10:      cout << "\nSie sind bereits im Wochenende!\n";
11:   else
12:     cout << "\nOkay, legen Sie einen Urlaubstag ein.\n";
13:   return 0;
14: }
```


```
Geben Sie einen Tag  ein (0-6): 6
Sie sind bereits im Wochenende!
```

Zeile 4 definiert einen Aufzählungstyp `DAYS` mit sieben Werten. Jeder dieser Werte entspricht einem Integer, wobei die Zählung mit `0` begonnen wird. Demzufolge ist der Wert von `Tuesday` (Dienstag) gleich `2`.

Der Anwender wird gebeten, einen Wert zwischen `0` und `6` einzugeben. Die Eingabe von »Sonntag« als Tag ist nicht möglich. Das Programm hat keine Ahnung, wie es die Buchstaben in Sonntag in einen der Aufzählungswerte übersetzen soll. Es kann jedoch die Werte, die der Anwender eingibt, mit einem oder mehreren Aufzählungskonstanten wie in Zeile 9 abgleichen. Die Verwendung von Aufzählungskonstanten verdeutlicht die Absicht des Vergleichs besser. Sie hätten das Ganze auch mit Integer-Konstanten erreichen können. Das Beispiel dazu finden Sie in Listing 3.8.

In diesem und allen anderen kleinen Programmen dieses Buches wird auf jeglichen Code verzichtet, der normalerweise dazu dient, ungültige Benutzereingaben abzufangen. So prüft dieses Programm im Gegensatz zu einem richtigen Programm nicht, ob der Anwender wirklich eine Zahl zwischen `0` und `6` eingibt. Ich habe diese Prüfroutinen absichtlich weggelassen, um die Programme klein und einfach zu halten und auf das Wesentliche zu konzentrieren.

Listing 3.8: Das gleiche Programm mit Integer-Konstanten

```
1: #include <iostream.h>
2: int main()
3: {
4:    const int Sunday = 0;
5:    const int Monday = 1;
6:    const int Tuesday = 2;
7:    const int Wednesday = 3;
8:    const int Thursday = 4;
9:    const int Friday = 5;
10:   const int Saturday = 6;
11:
12:   int choice;
13:   cout << "Geben Sie einen Tag ein (0-6): ";
14:   cin >> choice;
```

```
15:
16:   if (choice == Sunday || choice == Saturday)
17:      cout << "\nSie sind bereits im Wochenende!\n";
18:   else
19:      cout << "\nOkay, legen Sie einen Urlaubstag ein.\n";
20:
21:   return 0;
22:}
```


```
Geben Sie einen Tag  ein (0-6): 6
Sie sind bereits im Wochenende!
```

Die Ausgabe dieses Listings entspricht der in Listing 3.7. In diesem Programm wurde jedoch jede Konstante (Sonntag, Montag etc.) einzeln definiert, ein Aufzählungstyp `DAYS` existiert nicht. Aufzählungskonstanten haben den Vorteil, daß sie selbsterklärend sind – die Absicht des Aufzählungstypen `DAYS` ist jedem sofort klar.

Zusammenfassung

In diesem Kapitel haben Sie Variablen und Konstanten für numerische Werte und Zeichen kennengelernt, in denen Sie in C++ während der Ausführung Ihres Programms Daten speichern. Numerische Variablen sind entweder Ganzzahlen (`char`, `short` und `long int`) oder Fließkommazahlen (`float` und `double`). Die Zahlen können darüber hinaus vorzeichenlos oder vorzeichenbehaftet (`unsigned` und `signed`) sein. Wenn auch alle Typen auf unterschiedlichen Computern unterschiedlich groß sein können, so wird jedoch mit dem Typ für einen bestimmte Computer immer eine genaue Größe angegeben.

Bevor man eine Variable verwenden kann, muß man sie deklarieren. Damit legt man gleichzeitig den Datentyp fest, der sich in der Variablen speichern läßt. Wenn man eine zu große Zahl in einer Integer-Variablen ablegt, erhält man ein falsches Ergebnis.

Dieses Kapitel hat auch literale und symbolische Konstanten sowie Aufzählungskonstanten behandelt und die beiden Möglichkeiten zur Deklaration symbolischer Konstanten aufgezeigt: die Verwendung von `#define` und des Schlüsselwortes `const`.

Fragen und Antworten

F Wenn der Wertebereich einer Variablen vom Typ `short int` eventuell nicht ausreicht, warum verwendet man dann nicht immer Ganzzahlen vom Typ `long`?

A *Sowohl bei ganzen Zahlen vom Typ `short` als auch bei `long` können Platzprobleme auftreten, auch wenn der Wertebereich bei Zahlen vom Typ `long` wesentlich größer ist. Der Wertebereich eines `unsigned short int` reicht bis 65.535, während die Werte eines `unsigned long int` bis 4.294.967.295 gehen. Auf den meisten Maschinen nimmt aber eine Variable vom Typ `long` doppelt soviel Speicher ein wie eine Variable vom Typ `short` (vier Byte im Gegensatz zu zwei Byte) und ein Programm mit 100 solcher Variablen benötigt dadurch gleich 200 Byte RAM mehr. Allerdings stellt das heutzutage kaum noch ein Problem dar, da die meisten PCs über mehrere Megabyte Hauptspeicher verfügen.*

F Was passiert, wenn ich eine reelle Zahl einer Variablen vom Typ `int` statt vom Typ `float` zuweise? Als Beispiel dazu die folgende Codezeile:

```
int eineZahl = 5.4;
```

A *Ein guter Compiler erzeugt eine Warnung, aber die Zuweisung ist durchaus zulässig. Die zugewiesene Zahl wird zu einer ganzen Zahl abgerundet. Wenn man daher 5.4 einer ganzzahligen Variablen zuweist, erhält diese Variable den Wert 5. Es gehen also Informationen verloren, und wenn man später den Wert der Integer-Variablen einer Variablen vom Typ `float` zuweist, erhält die `float`-Variable ebenfalls nur den Wert `5`.*

F Warum sollte man auf literale Konstanten verzichten und sich die Mühe machen, symbolische Konstanten zu verwenden?

A *Wenn man einen Wert an vielen Stellen im gesamten Programm hindurch verwendet, kann man bei Verwendung einer symbolischen Konstanten alle Werte ändern, indem man einfach die Definition der Konstanten ändert. Symbolische Konstanten sprechen auch für sich selbst. Es kann schwer zu verstehen sein, warum eine Zahl mit 360 multipliziert wird. Einfacher ist dagegen eine Anweisung zu lesen, in der die Multiplikation mit `GradImVollkreis` realisiert wird.*

F Was passiert, wenn ich eine negative Zahl einer vorzeichenlosen Variablen vom Typ `unsigned` zuweise? Als Beispiel dient die folgende Codezeile:

```
Unsigned int einePositiveZahl = -1;
```

A *Ein guter Compiler wird eine Warnung ausgeben, auch wenn die Zuweisung absolut gültig ist. Die negative Zahl wird als Bitmuster interpretiert und der Variablen zugewiesen. Der Wert dieser Variablen wird dann als eine vorzeichenlose Zahl interpretiert. Demzufolge wird -1, dessen Bitmuster* `11111111 11111111` *(*`0xFF in hex`*) ist, als* `unsigned`*-Wert die Zahl 65.535 zugewiesen. Sollte diese Information Sie verwirren, möchte ich Sie auf Anhang C verweisen.*

F Kann ich mit C++ arbeiten, auch wenn ich nichts von Bitmustern, binärer Arithmetik und Hexadezimalzahlen verstehe?

A *Ja, aber nicht so effektiv wie mit Kenntnis dieser Themen. Im Gegensatz zu etlichen anderen Sprachen »schützt« C++ Sie nicht unbedingt davor, was der Computer macht. Dies kann jedoch auch als Vorteil begriffen werden, da die Programmiersprache dadurch mächtiger ist als andere Sprachen. Wie aber bei jedem leistungsfähigen Werkzeug, muß man verstehen, wie es funktioniert, um das Optimum herauszuholen. Programmierer, die ohne grundlegende Kenntnisse des Binärsystems versuchen, in C++ zu programmieren, werden oft über das Ergebnis erstaunt sein.*

Workshop

Der Workshop enthält Quizfragen, die Ihnen helfen sollen, Ihr Wissen zu festigen, und Übungen, die Sie anregen sollen, das eben Gelernte umzusetzen und eigene Erfahrungen zu sammeln. Versuchen Sie, das Quiz und die Übungen zu beantworten und zu verstehen, bevor Sie die Lösungen in Anhang D lesen und zur Lektion des nächsten Tages übergehen.

Quiz

1. Was ist der Unterschied zwischen einer Integer-Variablen und einer Fließkommavariablen?
2. Welche Unterschiede bestehen zwischen einer Variablen vom Typ `unsigned short int` und einer Variablen vom Typ `long int`?
3. Was sind die Vorteile einer symbolischen Konstanten verglichen mit einer literalen Konstanten?
4. Was sind die Vorteile des Schlüsselwortes `const` verglichen mit `#define`?
5. Wodurch zeichnet sich ein guter und ein schlechter Variablenname aus?
6. Was ist der Wert von `BLAU` in dem folgenden Aufzählungstyp?

```
enum FARBE { WEISS, SCHWARZ = 100, ROT, BLAU, GRUEN = 300 }
```

7. Welche der folgenden Variablennamen sind gut, welche sind schlecht und welche sind ungültig?

```
a) Alter
b) !ex
c) R79J
d) GesamtEinkommen
e) __Invalid
```

Übungen

1. Was wäre der korrekte Variablentyp, um die folgenden Informationen abzulegen?

   a) Ihr Alter

   b) die Fläche Ihres Hinterhofes

   c) die Anzahl der Sterne in der Galaxie

   d) die durchschnittliche Regenmenge im Monat Januar
2. Erstellen Sie gute Variablennamen für diese Informationen.
3. Deklarieren Sie eine Konstante für PI (Wert 3,14159)
4. Deklarieren Sie eine Variable vom Typ `float` und initialisieren Sie sie mit Ihrer PI-Konstanten.