
100%
Markt+Technik
jetzt lerne ich
Start ohne Vorwissen
C++
Das komplette Starterkit für den einfachen Einstieg in die Programmierung
Dirk Louis
Markt+Technik
DVD
Visual Studio
Express Editions

Kapitel 3

jetzt lerne ich

Daten und Variablen

Sicherlich ist Ihnen bekannt, dass Programme dazu dienen, Daten zu verarbeiten. Dies können komplexe Objekte sein – wie z.B. Bilder, Bestellungen in einem E-Shop oder auch zweidimensionale Koordinaten wie Sie sie bereits in Kapitel 1.3.1 kurz gesehen haben. Meist handelt es sich aber um einfache Zahlen oder Text.

Wie werden diese Daten in Programmen repräsentiert? Wo kann das Programm Daten zwischenspeichern? Welche Arten von Daten gibt es überhaupt? Was kann man mit diesen Daten anfangen? Diesen Fragen wollen wir in diesem Kapitel nachgehen.

Sie lernen in diesem Kapitel

- den Unterschied zwischen Variablen und Konstanten kennen,
- wie man Variablen definiert und verwendet,
- was Datentypen sind,
- warum Datentypen in der Programmierung so wichtig sind,
- welche elementaren Datentypen es in C++ gibt,
- wie man den Datentyp eines Werts in einen anderen Datentyp umwandeln kann.

3.1 Konstanten (Literale)

Daten können in einem Programm in zwei Formen vorkommen: als Konstanten oder als Variablen. Wenden wir uns zuerst den Konstanten zu.

Von Konstanten, genauer gesagt *Literalen*, spricht man, wenn die Daten direkt im Quelltext stehen. Eine Art von Literal haben Sie bereits kennengelernt: das Text- oder String-Literal.

Text (Strings)

```
// aus HalloWelt.cpp
cout << "Hallo Welt!" << endl;
```

Text-Literale stehen in doppelten Anführungszeichen und stehen für sich selbst (das heißt für den Text in den Anführungszeichen). Die Anführungszeichen teilen dem Compiler mit, dass die nachfolgenden Zeichen kein C++-Code sind, sondern Text, der irgendwie vom Programm zu verarbeiten ist. Solche Texte, oder allgemeiner ausgedrückt Zeichenfolgen, bezeichnet man in der Programmierung als *Strings*.

Strings weisen einige Besonderheiten auf, die sich auch auf die Programmierung auswirken. So kann man beispielsweise innerhalb eines Strings nicht ohne weiteres doppelte Anführungszeichen verwenden, und man darf Strings auch nicht einfach umbrechen. Aus diesem Grund werden wir uns in Kapitel 12 noch ganz ausführlich nur mit Strings beschäftigen.

Neben den String-Literalen gibt es noch eine Reihe weiterer Literale, von denen vor allem die Zahlen-Literale interessant sind.

8098

-54

3.141592653

Wichtig ist dabei die korrekte Schreibweise.

Ganzzahlen

Ganzzahlen, im Programmierjargon als *Integer* bezeichnet, schreibt man meist wie normale Zahlen.

8098

Bei Bedarf kann man die Werte auch als Hexadezimalzahlen angeben. Um dem Compiler anzuzeigen, dass die folgende Zahl eine Hexadezimalzahl ist, stellt man dem Zahlenwert das Präfix 0x voran.

```
0x1FA2      // entspricht dezimal 8098
```

Hexadezimalzahlen

Hexadezimalzahlen sind Zahlen zur Basis 16. Da wir gewohnt sind, mit dem Dezimalsystem zu rechnen und gar keine speziellen Ziffern für die Werte 10 bis 15 haben (die im Dezimalsystem ja bereits zwei Stellen belegen), verwendet man zur Kennzeichnung dieser Werte die Buchstaben A (= 10) bis F (= 15). Die Hexadezimalzahl A2F entspricht demnach dem Wert $10 * 16^2 + 2 * 16^1 + 15 * 16^0 = 2560 + 32 + 15 = 2607$.

Was die Hexadezimalzahlen für die Programmierung so interessant macht, ist ihre Nähe zu den Binärzahlen, die daher stammt, dass die Basis 16 eine Potenz, genauer gesagt die vierte Potenz der Basis 2 ist (ganz im Gegenteil zu der vollkommen ungeeigneten und computerunfreundlichen Basis 10). Dies ermöglicht es, je vier Stellen einer Binärzahl als eine Stelle einer Hexadezimalzahl darzustellen.

Nehmen wir zum Beispiel die Binärzahl

0010 0010 0011 0010

Diese Zahl in eine Dezimalzahl umzuwandeln, ist recht kompliziert. Man muss die Stellen abzählen, an denen die Einsen stehen, und für diese Stellen die Potenzen von 2 aufaddieren. Als Summe erhält man dann 8754.

Die Umwandlung in eine Hexadezimalzahl ist dagegen recht einfach, weil man immer nur 4 Stellen und die Potenzen von 2^0 bis 2^3 betrachten muss. Im ersten Viererblock ist die zweite Stelle eine 1, das entspricht einer 2 (2^1). Gleiches gilt für den zweiten Block. Im dritten Block kommt zu der 2 eine 1 (2^0), was eine 3 ergibt. Der letzte Block ist wieder eine 2, das heißt die zugehörige Hexadezimalzahl lautet:

2232 (= 8192 + 512 + 48 + 2)

In Kapitel 21.5 werden Sie ein Beispiel dafür sehen, wie man sich diese Eigenschaft der Hexadezimalzahlen zunutze machen kann.

Zahlen mit Nachkommastellen

Zahlen mit Nachkommstellen, im Programmierjargon als *Gleitkommazahlen* bezeichnet, schreibt man mit einem Punkt zwischen den Vorkomma- und Nachkommastellen.

3.141592653

Das ist für Deutschsprachige etwas verwirrend und eine stete Quelle von Fehlern, denn wir verwenden zur Abtrennung der Nachkommastellen ja an sich das Komma. Im Amerikanischen und Englischen verwendet man dagegen den Punkt zur Abtrennung der Nachkommastellen (und das Komma zur Kennzeichnung der Tausenderstellen). Daher wird in praktisch allen Programmiersprachen der Punkt zur Abtrennung der Nachkommstallen verwendet. Punktum.

Alternativ kann man Gleitkommazahlen auch in Exponentialschreibweise angeben.

```
3.141592653e0;      // = 3.141592653
3.14e3;             // = 3140
```

Exponentialschreibweise

Der Buchstabe e wird in Gleitkommazahlen zur Kennzeichnung eines nachfolgenden Exponenten zur Basis 10 verwendet. `3.14e3` bedeutet als $3.14 * 10^3$, und nicht etwa $3.14 * e^3$ (mit e gleich der Eulerschen Zahl).

Verwendung

Natürlich schreibt man Literale nicht einfach ziellos irgendwo in den Quelltext. Vielmehr übergibt man sie zur Verarbeitung an passende Funktionen, gibt sie aus (beispielsweise an `cout`) oder man nutzt sie zum Aufbau komplexerer Ausdrücke (`3 + 7 * 12`) oder man weist sie Variablen zu (`meineVar = 47`).

Das folgende Programm weist zum Beispiel die gerade vorgestellten Typen von konstanten Werten an Variablen zu und gibt diese dann aus.

Listing 3.1: Beispiele für Zahlenkonstanten (aus Literale.cpp)

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

int main()
{
   int integer1 = 8098;
   int integer2 = 0x1FA2;                // = 8098

   cout << integer1 << endl;
   cout << integer2 << endl;

   double gleitkomma1 = 3.141592653;
   double gleitkomma2 = 3.141592653e0;
   double gleitkomma3 = 3.14e3;          // = 3140

   cout << gleitkomma1 << endl;
   cout << gleitkomma2 << endl;
```

```
   cout << gleitkomma3 << endl;

   return 0;
}
```

Was aber genau sind Variablen und wie werden sie definiert?

3.2 Variablen

Literale sind eine wunderbare Sache, aber um richtig programmieren zu können, reichen Literale nicht aus. Wir brauchen zusätzlich auch noch eine Möglichkeit, wie man Daten zwischenspeichern kann – und zwar so, dass wir jederzeit auf die Daten zugreifen und sie bei Bedarf auch verändern können. Diese Möglichkeiten eröffnen uns die Variablen.

3.2.1 Variablendefinition

Variablen bezeichnen Speicherbereiche im RAM (Arbeitsspeicher), in denen ein Programm Werte ablegen kann. Um also mit Daten arbeiten zu können, müssen Sie zuerst eine Variable für diese Daten definieren, beispielsweise

```
int meineVar;
```

Der Compiler sorgt dann dafür, dass bei Ausführung des Programms Arbeitsspeicher für die Variable reserviert wird. Für den Compiler ist der Variablenname einfach ein Verweis auf den Anfang eines Speicherbereichs. Als Programmierer identifiziert man eine Variable mehr mit dem Wert, der gerade in dem zugehörigen Speicherbereich abgelegt ist.

Bei der Definition geben Sie nicht nur den Namen der Variablen (im obigen Beispiel `meineVar`), sondern auch deren *Datentyp* an (im Beispiel `int`). Dieser Datentyp teilt dem Compiler mit, wie der Inhalt des Speicherbereichs der Variablen zu interpretieren ist. Dies ist nötig, weil alle Daten im Arbeitsspeicher binär kodiert sind, also als eine Folge von Nullen und Einsen gespeichert werden. Dabei gibt es für die verschiedenen Arten von Daten (Text, Ganzzahlen, Gleitkommazahlen) unterschiedliche Kodierungsverfahren, die festlegen, wie die Werte der unterschiedlichen Datentypen binär kodiert werden beziehungsweise wie die binär kodierten Werte wieder zurückverwandelt werden. Sie als Programmierer brauchen nicht zu wissen, wie diese Kodierungen im Detail aussehen (die Kodierung wird ganz vom Compiler übernommen), aber Sie sollten sich stets bewusst sein, dass eine solche Kodierung stattfindet, denn darauf beruht das gesamte System der Datentypen (mit dem wir uns im Abschnitt 3.4 noch ausführlicher beschäftigen werden). Im Moment reicht uns jedoch zu wissen, dass wir bei der Definition einer Variablen auch immer einen Datentyp angeben müssen, der dem Compiler mitteilt, welche Art von Daten wir in der Variablen speichern möchten.

Für die elementaren Datentypen gibt es in C++ spezielle Schlüsselwörter, beispielsweise:

Tabelle 3.1: Einige wichtige C++-Datentypen

Einige C++-Datentypen	Art der Werte
`bool`	Wahrheitswerte `true` (wahr) und `false` (nicht wahr)
`int`	Ganzzahlen
`double`	Gleitkommazahlen
`char`	einzelne Zeichen
`string`[1]	Strings (Text) benötigt die Einbindung der Headerdatei `<string>`

Mit Hilfe dieser Schlüsselwörter sind wir jetzt in der Lage, Variablen für eine Vielzahl von Aufgaben zu definieren. Um beispielsweise den Computer eine kleine Addition ausführen zu lassen, könnten wir drei `int`-Variablen definieren: zwei `int`-Variablen für die zu addierenden Zahlen, eine `int`-Variable zum Abspeichern des Ergebnisses:

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

int main()
{
   int ersteZahl;
   int zweiteZahl;
   int ergebnis;

   ...

   return 0;
}
```

Wenn man im obigen Beispiel mehrere Variablen eines Datentyps definiert, braucht man nicht für jede Variable eine eigene Definition aufzusetzen. Man kann die Variablennamen auch durch Kommata getrennt hinter dem Datentyp auflisten:

```
int ersteZahl, zweite Zahl, ergebnis;
```

1. Der `string`-Datentyp sticht ein wenig aus den anderen in der Tabelle aufgelisteten Datentypen heraus. Während nämlich `bool`, `int`, `double` und `char` fest in die Sprache integriert sind, ist `string` ein komplexer Datentyp, der als Klassentyp in der C++-Standardbibliothek definiert ist. Daher ist auch das Einkopieren der zugehörigen Headerdatei erforderlich. Der `string`-Typ ist allerdings so gut in C++ integriert und so wichtig, dass wir ihn bereits hier in unser Repertoire aufnehmen.

Welche Version man wählt, hängt meist davon ab, wie wichtig die betreffenden Variablen sind. Wichtige Variablen definiert man meist allein in einer Zeile (und schreibt hinter die Variablendefinition vielleicht noch einen kleinen Kommentar), weniger wichtige oder bedeutungsmäßig eng zusammengehörende Variablen wird man eher in einer Zeile definieren.

Wichtiger ist die Frage, wo man die Variablen definiert. Grundsätzlich kann man Variablen global auf Dateiebene (beispielsweise unter den `#include`-Direktiven), lokal in Funktionen (wie im obigen Codefragment) oder als Elemente von Klassen (siehe Teil 3) definieren. Wir werden uns erst einmal auf Variablen konzentrieren, die in der `main()`-Funktion definiert werden. Da man eine Variable erst nach ihrer Definition verwenden kann, werden die Variablen meist am Anfang der Funktionen – vor den Anweisungen – definiert. Variablen, die man erst weiter unten benötigt, kann man aber auch erst später, mitten zwischen den Anweisungen der Funktionen, definieren. Es muss lediglich sichergestellt sein, dass die Variable nicht vor ihrer Definition in einer Anweisung benutzt wird.

Namensgebung

Sieht man einmal von der `main()`-Funktion ab, können Sie die Namen von Programmelementen, die Sie selbst definieren (vorerst sind dies nur Variablen, bald aber werden Funktionen, Strukturen, Klassen und noch einige andere Elemente hinzukommen), frei wählen. Sie müssen lediglich folgende Regeln zur Namensgebung beachten:

- Der Name muss mit einem Buchstaben oder einem Unterstrich _ anfangen.
- Innerhalb des Namens sind auch Zahlen erlaubt.
- Der Name darf keine Leer- oder Sonderzeichen und auch keine Umlaute enthalten.
- Der Name darf kein Schlüsselwort von C++ sein (siehe Anhang).

Der Name muss zudem in seinem Gültigkeitsbereich eindeutig sein. Für Variablen, die Sie innerhalb einer Funktion wie `main()` definieren, wäre der Gültigkeitsbereich z.B. der gesamte Anweisungsblock der Funktion.

Ansonsten sollte der Name nicht zu lang, aber dennoch aussagekräftig sein, damit man am Namen die Verwendung des Elements ablesen kann. Für Hilfsvariablen, die keine besonderen Daten speichern, gibt es aber oft keine sinnvollen Namen. Solche Variablen heißen dann meist `n`, `m`, `x`, `y` oder `tmp`.

Schließlich sei noch erwähnt, dass man in der Programmierung statt von Namen häufig auch von Bezeichnern spricht.

Und zu guter Letzt noch einmal der Hinweis, dass C++ streng zwischen Groß- und Kleinschreibung unterscheidet. Die beiden Bezeichner `meineVar` und `meinevar` sind also nicht identisch.

3.2.2 Werte in Variablen speichern

Nun war schon so oft die Rede davon, dass man in Variablen Werte speichern kann, dass es Zeit wird, sich dies in der Praxis anzuschauen:

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

int main()
{
   int ersteZahl;
   int zweiteZahl;
   int ergebnis;

   ersteZahl  = 8754;
   zweiteZahl =  398;

   ...

   return 0;
}
```

Hier wird in der Variablen `ersteZahl` der Wert 8754 und in der Variablen `zweiteZahl` der Wert 398 abgelegt. Anhand der Variablen `ersteZahl` wollen wir uns kurz klar machen, was dabei im Hintergrund eigentlich geschieht.

Der Anwender hat das Programm gestartet, die `main()`-Funktion wird ausgeführt. Als Erstes wird der Speicher für die Variablen reserviert. Für die Variable `ersteZahl` wird im Arbeitsspeicher ein Speicherbereich reserviert, der groß genug ist, dass man `int`-Werte in ihm ablegen kann. Intern merkt sich das Programm, an welcher Speicheradresse der Speicherbereich der Variablen `ersteZahl` beginnt.

Nach der Variablendefinition wird der Variablen `ersteZahl` der Wert 8754 zugewiesen. Das heißt, das Programm ermittelt die Anfangsadresse des Speicherbereichs zur Variablen `ersteZahl` und kopiert den Wert 8754 (natürlich in binär kodierter Form) in diesen Bereich.

Wenn Sie einer Variablen einen Wert zuweisen, muss der Datentyp des Werts zum Datentyp der Variablen passen.

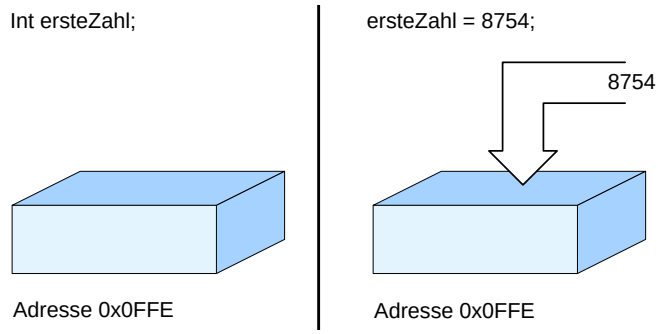

Abb. 3.1: Definition und Zuweisung

Variablen bei der Definition initialisieren

Wenn Sie möchten, können Sie einer Variablen auch direkt bei der Definition einen Anfangswert zuweisen:

```
int main()
{
   int ersteZahl = 8754;
   int zweiteZahl = 398;
   int ergebnis;
```

3.2.3 Werte von Variablen abfragen

Einen Wert in einer Variablen abzuspeichern, ist natürlich nur dann interessant, wenn man auf den Wert der Variablen später noch einmal zugreifen möchte – beispielsweise um den Wert in einer Formel einzubauen oder auszugeben.

Listing 3.2: Programmieren mit Variablen (aus Variablen.cpp)

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

int main()
{
   int ersteZahl;
   int zweiteZahl;
   int ergebnis;

   ersteZahl  = 8754;
   zweiteZahl =  398;
   ergebnis = ersteZahl + zweiteZahl;

   cout << "8754 + 398 = " << ergebnis << endl;

   return 0;
}
```

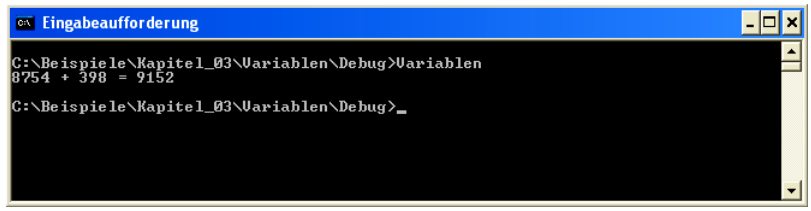

Abb. 3.2: Ausgabe des Programms Variablen (unter Windows XP)

Offensichtlich kann man über den Variablennamen der Variablen sowohl einen neuen Wert zuweisen als auch den aktuellen Wert der Variablen abfragen. Woher aber weiß der Compiler, wenn er auf einen Variablennamen trifft, ob er der Variablen einen Wert zuweisen oder ob er den aktuellen Wert der Variablen abfragen soll?

Ganz einfach!

- Taucht der Variablenname auf der linken Seite einer Zuweisung auf (wie in `zahl = 100;`), weist der Compiler der Variablen den Wert des Ausdrucks auf der rechten Seite des =-Operators zu.
- Taucht der Variablenname an irgendeiner anderen Stelle auf (wie in `cout << zahl;`, verwendet der Compiler anstelle des Variablennamens den aktuellen Wert der Variablen.

L- und R-Wert

Ganz exakt ausgedrückt, behandelt der Compiler einen Variablennamen je nach Kontext entweder als L-Wert oder als R-Wert. »L-Wert« heißt, dass er in der Variablen etwas speichert – wie es zum Beispiel der Fall ist, wenn die Variable auf der **l**inken Seite einer Zuweisung steht. »R-Wert« heißt, dass er den Wert der Variablen verwendet – wie es zum Beispiel der Fall ist, wenn die Variable auf der **r**echten Seite einer Zuweisung steht.

Die linke Seite einer Zuweisung ist ohne Zweifel die bedeutendste, aber nicht wie oben behauptet die einzige Position, an der eine Variable als L-Wert behandelt wird. Weitere Stellen sind das Einlesen mit dem >>-Operator (siehe Kapitel 3.5.1) oder das Inkrementieren bzw. Dekrementieren mit den Operatoren ++ bzw. -- (siehe Kapitel 4.4).

Vorsicht! Wenn Sie den Wert einer Variablen abfragen, der Sie zuvor noch keinen Wert zugewiesen haben, erhalten Sie sinnlose Werte zurück.

3.3 Konstante Variablen

Neben den Literalen aus Abschnitt 3.1 gibt es noch eine zweite der Form der Konstante: die konstanten Variablen.

Konstante Variablen werden grundsätzlich wie ganz normale Variablen definiert – mit zwei Unterschieden:

- Da der Wert einer konstanten Variablen nach der Erzeugung nicht mehr verändert werden kann, muss man den Wert der konstanten Variablen direkt bei der Definition angeben (Initialisierung).
- Vor die Definition wird das Schlüsselwort `const` gestellt.

Außerdem ist eine gute Konvention, für Konstanten nur Großbuchstaben zu verwenden – man kann die Konstanten im Programm dann gut erkennen.

```
const int MONATE = 12;
```

Konstante Variablen haben gegenüber den Literalen einen bedeutenden Vorteil: Sie sind mit einem Namen verbunden. Nach der Definition der konstanten Variablen kann man im Programmquelltext den Konstantennamen verwenden. Dies macht den Quelltext besser lesbar und hilft auch, wenn man bei einer späteren Überarbeitung des Programms den Wert einer Konstanten ändern möchte.

Konstante Werte, die eine besondere Bedeutung haben, sollten Sie daher immer als `const`-Variablen definieren. Nehmen Sie z.B. die folgende Formel zur Berechnung des Umfangs eines Kreises:

```
umfang = 2 * radius * PI;
```

Hier sollte PI eine `const`-Variable sein, die den (genäherten) Wert für Pi speichert, `radius` sollte eine normale Variable sein, der vor der Berechnung der Radius des Kreises zugewiesen wird, und die 2 bleibt als Literal stehen, da ihr keine andere Bedeutung zukommt, als eben eine 2 zu sein.

3.4 Die Datentypen

In C++ beginnt jede Variablendefinition mit einer Typangabe, die dem Compiler mitteilt, welche Art von Daten in der Variablen gespeichert werden kann.

Da die Unterscheidung der Datentypen ein sehr wichtiges Konzept typisierter Programmiersprachen (zu denen auch C++ gehört) darstellt und zudem weit reichende Auswirkungen auf unseren Programmcode hat, werden wir uns in diesem Abschnitt etwas ausführlicher mit dem Konzept der Datentypen im Allgemeinen und den Datentypen von C++ im Besonderen beschäftigen.

3.4.1 Die Bedeutung des Datentyps

Da dies ein Buch für Programmiereinsteiger ist, möchte ich Sie nicht zu sehr mit technischen Details quälen. Um die Bedeutung der Datentypen zu erfassen, ist dies aber auch gar nicht notwendig. Diese kann man sich auch dadurch veranschaulichen, dass man sich selbst einmal in die Rolle eines Compilers versetzt.

Für jede Variable, die Sie in einem Programm deklarieren, reserviert der Compiler Platz im Arbeitsspeicher. Wenn Sie der Variablen im weiteren Verlauf des Programms einen Wert zuweisen, legt der Compiler diesen Wert im Speicher der Variablen ab. Wenn Sie im Programm den Wert der Variablen abfragen, liest der Compiler den Wert aus dem Speicher der Variablen aus.[1]

Soweit scheint alles recht einfach. Kompliziert wird es erst dadurch, dass der Arbeitsspeicher des Rechners ein elektronischer, digitaler Speicher ist, der aus Millionen von Zellen besteht, die jede nur eine 1 (Spannung an) oder eine 0 (Spannung aus) speichern können. Sie können diese Speicher selbst simulieren. Nehmen Sie einfach ein Blatt mit Rechenkästchen zur Hand, umrahmen Sie ca. 20 mal 20 Kästchen, stellen Sie sich vor, dass Sie in jedes Kästchen nur eine 1 oder eine 0 schreiben dürfen, und fertig ist Ihr Arbeitsspeicher (siehe Abbildung 3.3).

Abb. 3.3: Unser simulierter Arbeitsspeicher

Jetzt wollen Sie eine Variable `var1` deklarieren und in ihr die Zahl 3 abspeichern. Da man im Arbeitsspeicher nur Nullen und Einsen speichern kann, müssen Sie die Zahl 3 in eine Folge von Nullen und Einsen umwandeln, eine sogenannte Binärdarstellung. Im Binärsystem hat die Zahl 3 die Darstellung 11 ($1*2^1 + 1*2^0$).

Eine Zahl als Binärzahl auszudrücken, bedeutet, sie als die Summe der Potenzen von 2 (1, 2, 4, 8, 16 etc.) auszudrücken (statt als Summe der Potenzen von 10, wie wir es von unseren Dezimalzahlen gewohnt sind).

1. Um keine Missverständnisse aufkommen zu lassen: Der Compiler reserviert den Arbeitsspeicher natürlich nicht direkt. Korrekter wäre es zu sagen, dass der Compiler den Maschinencode erzeugt, der später bei Ausführung des Programms den Arbeitsspeicher reserviert. Gleiches gilt für das Speichern und Abfragen von Werten oder anderen Speicherzugriffen. Der Compiler erzeugt immer nur den Maschinencode zum Speichern und Abfragen der Werte. Die wirklichen Speicherzugriffe erfolgen erst bei Ausführung des Programms.

Sie könnten nun die ersten zwei Zellen in Ihrem simulierten Arbeitsspeicher für die Variable `var1` reservieren und in ihr die Zahl 11 ablegen. Was aber, wenn Sie später den Wert 5 in `var1` ablegen wollen? 5 ist binär gleich 101 und benötigt drei Speicherzellen. Für unsere Variable sind aber nur zwei Speicherzellen reserviert, und der Compiler hat keine Möglichkeit, den Speicherplatz einer bereits reservierten Variablen nachträglich zu erweitern. Wir müssen also direkt bei der Deklaration der Variablen genügend Speicherzellen reservieren. Seien wir etwas großzügiger und reservieren wir für `var1` doch gleich 2 Byte (siehe Abbildung 3.4).

Programmierer zählen Speicherzellen in Bits und Bytes. Ein *Bit* ist eine Informationseinheit, die einen der Werte 1 oder 0 annehmen kann. Die binäre Zahl 101 besteht also aus drei Bits. Im Arbeitsspeicher kann jede Speicherzelle genau eine 1-Bit-Information speichern. Ein *Byte* sind 8 Bit.

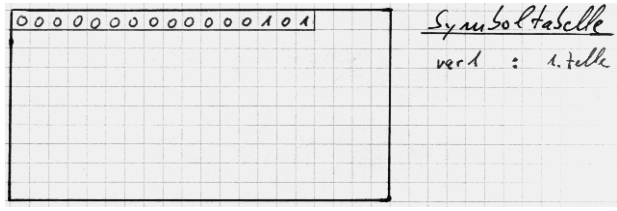

Abb. 3.4: Für die Variable var1 wurde Speicher reserviert. Der Wert der Variablen wurde im Speicher eingetragen.

2 Byte entsprechen 16 Speicherzellen (= 16 Bit). Wenn wir festlegen, dass die erste Speicherzelle das Vorzeichen festlegt (0 für positive, 1 für negative Zahlen), bleiben 15 Bit für den eigentlichen Wert – das bedeutet, wir können in `var1` jetzt Werte zwischen -32.768 und +32.767 abspeichern.

Als Nächstes wollen wir eine Gleitkommazahl mit Nachkommastellen abspeichern. Wir deklarieren dazu eine neue Variable `var2` und reservieren für diese die nächsten 2 Byte in unserem simulierten Arbeitsspeicher. In der Variablen wollen wir jetzt den Wert `1.3` speichern.

Hoppla! Wie sollen wir die Zahl 1.3 als Bitfolge kodieren? Sollen wir 1 und 3 einzeln in Binärzahlen umwandeln und dann festlegen, dass im ersten Byte der Wert vor dem Komma und im zweiten Byte der Wert hinter dem Komma abgelegt wird? Dann könnten wir selbst in 2 Byte nur kleine Zahlen mit wenigen Nachkommastellen abspeichern. Besser ist es, die Zahl in Exponentialschreibweise auszudrücken und zwar so, dass vor dem Komma eine 0 steht. So wird aus

1.3 zuerst $1.3 * 10^0$ und dann $0.13 * 10^1$

Statt 2 Byte belegen wir 4 Byte für die Gleitkommavariable. In den ersten drei Byte speichern wir die Nachkommastellen (13 = 1101) – die Null brauchen wir nicht zu speichern, da wir ja alle Gleitkommazahlen so formulie-

ren, dass sie mit einer Null vor dem Komma beginnen. Im vierten Byte speichern wir die Potenz zur Basis 10 (in unserem Beispiel also 1).

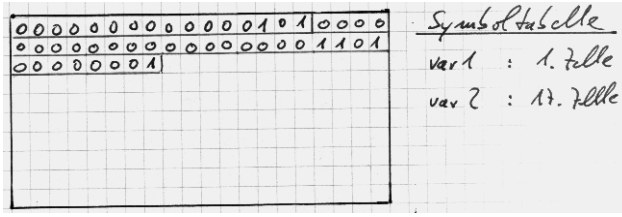

Abb. 3.5: Speicher für eine Gleitkommavariablen

Zu guter Letzt reservieren wir auch noch eine Variable `var3`, in der wir den Buchstaben 'C' ablegen wollen. Buchstaben werden mit Hilfe der sogenannten ASCII-Tabelle kodiert. In dieser Tabelle ist für jeden Buchstaben ein Zahlenwert angegeben. Um den Buchstaben C im Speicher abzulegen, reservieren wir 1 Byte Speicher, schauen in der ASCII-Tabelle (siehe Anhang dieses Buches) nach, welchen Codewert der Großbuchstabe C hat (67) und wandeln diesen in Binärcode um (67 = 0100 0011), den wir in der Variablen speichern.

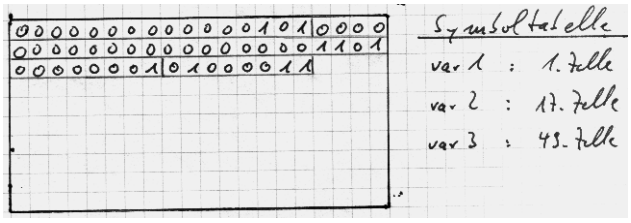

Abb. 3.6: Speicher für eine char-Variable

So, jetzt haben wir Speicher für Variablen von drei verschiedenen Datentypen reserviert (Ganzzahlen, Gleitkommazahlen und Zeichen). Wir wissen, dass jeder Datentyp über einen spezifischen Speicherbedarf und ein eigenes Kodierungsverfahren verfügt. Und genau deshalb benötigt der Compiler zu jeder Variablendeklaration die Typangabe. Aus ihr kann er ablesen,

- wie viel Speicher er für die Variable reservieren soll und
- wie er Werte, die der Variablen zugewiesen werden, in Bitfolgen kodieren und im Speicherbereich der Variablen ablegen soll (und umgekehrt, beim Abfragen des Variablenwerts die Bitfolge in einen Wert zurückverwandeln soll).

Die in diesem Abschnitt vorgestellten Kodierungsverfahren für Ganzzahlen, Gleitkommazahlen und Zeichen sind vereinfachte Versionen der im Rechner ablaufenden Kodierungen.

Der Datentyp einer Variablen legt aber nicht nur fest, wie der Speicherplatz der Variablen einzurichten und zu verwalten ist, er bestimmt auch, welche Operationen mit den Variablen überhaupt durchgeführt werden können. So kann man beispielsweise `int`-Variablen addieren, subtrahieren und dividieren. `string`-Variablen kann man dagegen nur addieren (aneinander hängen), aber nicht subtrahieren oder gar dividieren.

Der Datentyp einer Variablen legt also auch fest, was man mit der Variablen machen, wie man mit ihr programmieren kann. Und damit Sie nicht auf dumme Gedanken kommen und aus Trotz oder Schusseligkeit die eine oder andere Variable doch einmal zweckentfremden, wacht der Compiler streng darüber, dass die Variablen nur so verwendet werden, wie es ihr Datentyp zulässt. Das mag dem Anfänger manchmal recht lästig sein, besonders wenn der Compiler das aufgesetzte Programm partout nicht übersetzen will, ist aber andererseits auch ein wirksames Mittel, um den Programmierer vor dem Aufsetzen fehlerhaften Codes zu bewahren.

3.4.2 Die elementaren Datentypen

Die elementaren Datentypen von C++ sind in der Sprache fest verankert. Jeder dieser Datentypen verfügt über ein eigenes Schlüsselwort, das Sie in der Variablendeklaration als Typ angeben.

In Tabelle 3.2 sind die wichtigsten elementaren Datentypen aufgeführt. Zu jedem Datentyp ist angegeben, welche Art von Daten man in Variablen des Datentyps abspeichern kann und wie die Literale dieses Datentyps aussehen. (Eine vollständige Tabelle finden Sie im Anhang dieses Buches.)

Tabelle 3.2: Die wichtigsten elementaren Datentypen

Typ	Beschreibung	Beispiele
`bool`	Wahrheitswerte: `true`, `false`	`bool var = true;`
`char`	einzelnes Zeichen	`char var = 'h';`
`short`	Ganzzahlen im Bereich von –32768 bis 32.767	`short var = 12;`
`int`	Ganzzahlen im Bereich von –2147483648 bis 2147483647	`int var = 12;`
`long`	Ganzzahlen im Bereich von –2147483648 bis 2147483647	`long var1 = 12L;`
`long long`	Ganzzahlen im Bereich von –9223372036854775808 bis 9223372036854775807	`long var1 = 12L;`
`float`	Gleitkommazahlen im Bereich von $\pm 3.4 \times 10^{-38}$ bis $\pm 3.4 \times 10^{38}$	`float var = 1.23F;`
`double`	Gleitkommazahlen im Bereich von $\pm 1.7 \times 10^{-308}$ bis $\pm 1.7 \times 10^{308}$	`double var = 1.23;`

Zwei Dinge dürften Ihnen beim Studium der Tabelle auffallen:

- Es gibt allein drei Datentypen für Ganzzahlen (tatsächlich gibt es sogar noch mehr Datentypen für Ganzzahlen, doch werden diese eher selten benötigt).

Der Grund ist einfach der, dass man zum Abspeichern größerer Zahlen mehr Speicherplatz benötigt. Nun könnte man natürlich sagen, dann reservieren wir für alle Ganzzahlen jeweils 64 Bit Speicher (was dem Datentyp `long` entspräche). Dann bräuchten wir nur *einen* Ganzzahl-Datentyp. Dafür würden Programme mit vielen, aber kleinen Ganzzahlen extrem viel Speicher verschwenden. Aus diesem Grund definieren die meisten Programmiersprachen für Ganzzahlen (und auch Gleitkommazahlen) mehrere Datentypen, unter denen der Programmierer wählen kann.

- Der Datentyp `string` ist in der Tabelle nicht aufgeführt. Dies liegt einfach daran, dass es sich bei `string` nicht um einen der elementaren Datentypen handelt. `string` ist vielmehr ein Klassentyp, der in der C++-Standardbibliothek definiert ist.

Welchen Datentyp soll man nehmen?

An sich kommt man mit den fünf Datentypen `bool`, `int`, `double`, `char` und `string` für Wahrheitswerte, Ganzzahlen, Gleitkommazahlen, Zeichen und Strings sehr gut aus.

In bestimmten Situationen sollte man allerdings auf andere Datentypen ausweichen:

- Wenn Sie mit Variablen arbeiten, denen nur kleine Ganzzahlenwerte zugewiesen werden, können Sie sich überlegen, anstatt `int` vielleicht `short` zu verwenden.
- Wenn Sie mit Variablen arbeiten, denen unter Umständen auch sehr große Ganzzahlenwerte zugewiesen werden, müssen Sie statt `int` den Datentyp `long` verwenden oder gar den Gleitkommatyp `double` wählen.
- Wenn Sie Berechnungen anstellen, bei denen es möglichst nicht zu Fehlern durch Rundungen kommen soll, speichern Sie die betreffenden Werte möglichst in ganzzahligen Datentypen oder weichen Sie sogar auf speziell für dieses Aufgabenbereich definierte komplexe Datentypen aus.

3.4.3 Weitere Datentypen

C++ wäre nicht so mächtig, wenn es für den Programmierer keine Möglichkeit gäbe, neben den elementaren Datentypen auch eigene Datentypen zu definieren.

Die selbstdefinierten oder komplexen Datentypen teilt C++ in vier Kategorien:

- Arrays
- Aufzählungen
- Strukturen
- Klassen

Im weiteren Verlauf dieses Buches werden Sie diese Datentypen noch alle kennenlernen.

3.5 Typumwandlung

C++ ist eine sehr typenstrenge Programmiersprache, die sehr darauf achtet, dass eine Variable nur Werte ihres Typs zugewiesen bekommt und auch nur entsprechend ihres Typs verwendet wird. Dies heißt jedoch nicht, dass der Datenaustausch zwischen Variablen unterschiedlicher Typen ganz und gar unmöglich wäre. Ja, es wäre geradezu fatal, wenn die Sprache keine Möglichkeiten zur Typumwandlung kennen würde.

3.5.1 Typumwandlung bei der Ein- und Ausgabe

Betrachten wir dazu noch einmal das Programm *Variablen.cpp* aus Abschnitt 3.2.3.

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

int main()
{
   int ersteZahl;
   int zweiteZahl;
   int ergebnis;

   ersteZahl  = 8754;
   zweiteZahl =  398;
   ergebnis = ersteZahl + zweiteZahl;

   cout << "8754 + 398 = " << ergebnis << endl;

   return 0;
}
```

Hier wird in der `cout`-Anweisung der Wert der `int`-Variablen `ergebnis` auf die Konsole ausgegeben. Auf die Konsole kann man aber nur Strings ausgeben! Des Rätsels Lösung ist, dass der <<-Operator den Wert von `ergeb-`

nis, in unserem Beispiel die Zahl 9152, in den String "9152" umwandelt. Dieser wird dann auf die Konsole ausgegeben.

Funktioniert dies eigentlich auch umgekehrt?

Als Pendant zu dem cout-Objekt, welches die Konsole repräsentiert, gibt es auch ein Objekt cin, welches das Standardeingabegerät (sprich die Tastatur) repräsentiert. Via cin können daher über die Tastatur eingetippte Zeichenfolgen abgefragt und in Variablen abgespeichert werden. Der zugehörige Operator >> wandelt dabei die Zeichenfolgen automatisch in die Datentypen der Zielvariablen um.

Listing 3.3: Einlesen von Daten über die Tastatur (Eingabe.cpp)

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

int main()
{
   int ersteZahl;
   int zweiteZahl;
   int ergebnis;

   cout << endl;
   cout << " Geben Sie die erste Zahl ein : ";
   cin >> ersteZahl;

   cout << " Geben Sie die zweite Zahl ein: ";
   cin >> zweiteZahl;

   ergebnis = ersteZahl + zweiteZahl;

   cout << endl;
   cout << " " << ersteZahl << " + " << zweiteZahl;
   cout << " = " << ergebnis << endl;
   cout << endl;

   return 0;
}
```

Abb. 3.7: Beispiel für eine Sitzung mit dem Programm

Lassen Sie sich nicht von den vielen cout-Anweisungen irritieren. Sie dienen zum einem dazu, dem Anwender mitzuteilen, welche Eingaben von ihm erwartet werden, zum anderen sollen sie die Ausgaben des Programms mit Hilfe von Leerzeilen und -zeichen etwas leserfreundlicher zu gestalten.

Wirklich neu ist an diesem Programm allein das Einlesen der beiden Zahlen über die Tastatur.

```
cin >> ersteZahl;
```

Diese Anweisung hält das Programm so lange an, bis der Anwender etwas über die Tastatur eintippt (in Abbildung 3.7 war dies die Zeichenfolge »1000«) und durch Drücken der ↵-Taste abschickt. Dann liest der >>-Operator den eingegebenen Text ein und versucht, ihn in eine ganze Zahl (den Datentyp der Variablen ersteZahl) umzuwandeln. Für Eingaben der Form »eins« würde diese Umwandlung scheitern, für »1000« ist sie kein Problem. Tritt kein Fehler bei der Umwandlung auf, wird die Zahl in der Variablen ersteZahl gespeichert.

Was die Operatoren >> und << können, sollte für uns doch ebenfalls möglich sein? Die Frage ist nur wie? Wenn man nämlich versucht, eine Zahl an eine string-Variable zuzuweisen:

```
#include <string>
...
int main()
{
   string str;
   str  = 8754;
   ...
}
```

erhält man ganz seltsame Ergebnisse. (Was daran liegt, dass die Zahl eben nicht automatisch umgewandelt wird. Stattdessen interpretiert sie der Compiler als den ASCII-Code eines Zeichens und versucht, das betreffende Zeichen in str zu speichern – quasi als String aus nur einem einzelnen Zeichen.)

Dennoch können wir natürlich auch Umwandlungen zwischen Strings und Zahlen durchführen. Die benötigten Verfahren sind allerdings etwas kompliziert, weswegen wir sie uns für ein später (Kapitel 12.8) aufheben.

3.5.2 Automatische Typumwandlungen

Neues Spiel, neues Glück! Wie sieht es mit der folgenden Typumwandlung aus:

```
double r = 12;
```

Oh, Sie sind der Auffassung, dass hier überhaupt keine Typumwandlung vorliegt? Dann sollten Sie sich noch einmal die Formate für die Literale von Ganzzahlen und Gleitkommazahlen ansehen, denn Zahlen-Literale ohne Punkt und Nachkommastellen (bzw. Exponent) sind für den Compiler immer `int`-Werte. Folglich erfordert die obige Zuweisung für den Compiler auch eine implizite Typumwandlung von 12 in 12.0. Schafft er das? Aber sicher!

Und wie sieht es mit folgendem Code aus:

```
double r = 47.11;
int n;

n = r;
```

Hier wird eine Gleitkommazahl mit einem Nachkommaanteil (Inhalt von `r`) in eine Ganzzahl ohne Nachkommastellen umgewandelt, damit der Wert in der `int`-Variablen `n` gespeichert werden kann. Dies ist meist unproblematisch. Allerdings geht der Nachkommaanteil bei der Umwandlung verloren. Wenn Sie dies nicht stört, können Sie eine solche Umwandlung ohne Probleme vornehmen.[1]

Insgesamt nimmt der Compiler Umwandlungen zwischen den elementaren Datentypen weitgehend automatisch vor. Sie müssen allerdings beachten, dass dies zu Rundungs- oder gar noch gravierenderen Fehlern führen kann, wenn der Wertebereich des Zieltyps nicht groß genug ist, den umzuwandelnden Wert aufzunehmen.

Zeichen vom Typ `char` können problemlos in `int`-Werte umgewandelt werden, da die Zeichen intern ja in Form ihrer ASCII-Codes (letztlich also ganze Zahlen der Größe 1 Byte) gespeichert werden.

Ganzzahlige Werte ungleich 0 können in den booleschen Wert `true` umgewandelt werden, Werte gleich 0 werden zu `false`. Die umgekehrten Umwandlungen sind ebenfalls möglich. (Mehr zur Programmierung mit booleschen Werten in Kapitel 5.1.)

Außerdem gibt es automatische Umwandlungen zwischen Klassentypen und deren abgeleiteten Typen.

1. Manche Compiler weisen auf die erzwungene Abrundung mit einer Warnung hin.

3.5.3 Explizite Typumwandlungen

In Fällen, wo der Compiler keine implizite Typumwandlung vornimmt, die vorgenommene Umwandlung nicht den Absichten des Programmierers entspricht oder der Programmierer Warnungen des Compilers vermeiden will, kann die Umwandlung unter Umständen mit dem Cast-Operator `()` oder den speziellen Typumwandlungsoperatoren `const_cast`, `dynamic_cast`, `static_cast` und `reinterpret_cast` explizit herbeigeführt werden.

```
double r = 47.11;
int n;

n = (int) r;
```

Hier gibt der Programmierer durch Einsatz des Cast-Operators deutlich kund, dass er eine Umwandlung in den `int`-Typ wünscht – und nicht etwa einen Fehler begangen hat, weil er der `int`-Variablen `n` den Wert einer `double`-Variablen zuweist. Jeder ordentliche C++-Compiler wird dies respektieren und seine Warnungen für sich behalten.

Der Cast-Operator `()` wird in C++ wegen seiner einfachen Syntax gerne verwendet, stammt aber eigentlich vom Vorgänger C. Besser ist es, die neuen C++-Operatoren zu verwenden, die die verschiedenen Konvertierungen kategorisieren, die Textsuche nach den Konvertierungen erleichtern und zum Teil auch sicherer sind.

Operator	Einsatzgebiet
`static_cast<Typ>(var)`	Zur Umwandlung zwischen »nahe verwandten« Typen – also alle Fälle, in denen der Compiler auch eine implizite Typumwandlung vornehmen könnte, plus einige einfache Sonderfälle.
`dynamic_cast<Typ>(var)`	Interessant für die objektorientierte Programmierung: Zur Umwandlung zwischen Zeigern oder Referenzen auf Klassentypen einer gemeinsamen Klassenhierarchie. Liefert im Fehlerfall, wenn eine Umwandlung nicht möglich ist, einen `NULL`-Zeiger zurück.
`const_cast<Typ>(var)`	Entfernt eine `const`- oder `volatile`-Deklaration aus einem Typ).
`reinterpret_cast<Typ>(var)`	Für fast alle anderen erlaubten Umwandlungen.

Tabelle 3.3: Operatoren für die Typumwandlung

In professionellen Anwendungen ist es empfehlenswert, für implizite Typumwandlungen, die zu Informationsverlusten führen (wie z.B. die Umwandlung eines `double`-Werts in einen `int`-Wert), ebenfalls den `static_cast`-Operator zu verwenden. Treten beim Testen des Programms dann Fehler auf, können Sie leichter zu den einzelnen Typumwandlungen springen und diese kontrollieren.

3.6 Übungen

1. Welche der folgenden Variablennamen sind nicht zulässig?

```
123
zähler
JW_Goethe
JR.Ewing
_intern
double
Liebe ist
```

2. Datentypen sind das A und O der Variablendefinition. Wie lauten die Schlüsselwörter zur Definition von Variablen für
   - Zeichen
   - Strings
   - Ganzzahlen
   - Gleitkommazahlen
   - Wahrheitswerte
3. Welche der folgenden Variablendefinitionen sind nicht zulässig?

```
int 123;
char c;
bool option1, option2;
bool option1 option2;
bool option1, option2;
short y = 5;
short x = 5+1;
short x = y; // y wie oben
```

4. Warum führt der folgende Code zu einem Compiler-Fehler?

```
long x;
x = 5;
long x;
x = 4;
```

5. Ist die folgende Typumwandlung erlaubt oder wird sie mit einer Fehlermeldung enden?

```
int zahl;
string str = "123.3";

zahl = str;
```