WWW.PEARSON-STUDIUM.DE
CWS
COMPANION WEBSITE

it
informatik

Bjarne Stroustrup

Einführung in die Programmierung mit C++

PEARSON
Studium

Objekte, Typen und Werte

3

ÜBERBLICK

„Das Glück bevorzugt den, der vorbereitet ist.“

– Louis Pasteur

Dieses Kapitel soll Ihnen die Grundlagen der Datenspeicherung und Datenverwendung in Programmen vermitteln. Zu diesem Zweck konzentrieren wir uns zunächst auf das Einlesen von Daten über die Tastatur. Nachdem wir die Grundbegriffe Objekt, Typ, Wert und Variable geklärt haben, führen wir die ersten Operatoren ein und zeigen Ihnen eine ganze Reihe von Beispielen für die Verwendung von Variablen der Typen **char**, **int**, **double** und **string**.

3.1 Eingabe

Das *Hello, World!*-Programm aus dem vorangehenden Kapitel schreibt lediglich einen Text auf den Bildschirm, d.h., es produziert eine Ausgabe. Es liest nichts ein und übernimmt auch keine Eingaben von seinen Benutzern. Wie langweilig! Richtige Programme produzieren in der Regel Ergebnisse, die auf unseren Eingaben basieren – anstatt bei jeder Ausführung das Gleiche zu tun.

Um etwas einzulesen, benötigen wir einen Ort, wohin wir etwas einlesen können; d.h., wir benötigen einen Platz im Computerspeicher, in dem wir das Gelesene ablegen können. Einen solchen „Ort“ nennen wir Objekt. Ein *Objekt* ist ein Speicherbereich mit einem *Typ*, der angibt, welche Art Information im Speicherbereich abgelegt werden kann. Ein benanntes Objekt heißt auch *Variable*. Zum Beispiel werden Zeichenfolgen in Variablen vom Typ **string** abgelegt. („string“ ist das englische Wort für „Schnur, Kette“. In der Programmierung werden Zeichenfolgen daher oft auch als *Strings* bezeichnet.) Und ganze Zahlen werden in Variablen vom Typ **int** abgelegt. („int“ ist eine Abkürzung für „integer“ – das englische Wort für ganze Zahlen. In der Programmierung werden ganze Zahlen daher oft auch als *Integer* bezeichnet.) Stellen Sie sich ein Objekt einfach als ein „Kästchen“ vor, in das Sie einen Wert vom Typ des Objekts packen.

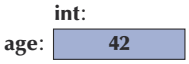


Abbildung 3.1: Ein Objekt mit Typangabe und Wert

▶ Abbildung 3.1 zeigt also ein Objekt vom Typ **int** namens **age**, das den Integer-Wert **42** enthält. Mithilfe einer **string**-Variablen können wir einen String aus der Eingabe einlesen und wie folgt wieder ausgeben:

```
// einen Vornamen einlesen und ausgeben
#include "std_lib_facilities.h"
int main()
{
  cout << "Bitte geben Sie Ihren Vornamen ein (gefolgt von 'Enter'):\n";
  string first_name;   // first_name ist eine Variable vom Typ string
  cin >> first_name;   // lies die Zeichen in first_name ein
  cout << "Hallo " << first_name << "!\n";
}
```

#include und **main()** kennen wir bereits aus Kapitel 2. Die **#include**-Direktive, die wir für all unsere Programme (bis Kapitel 12) benötigen, werden wir zukünftig nicht mehr extra erwähnen, um nicht vom Wesentlichen abzulenken. Außerdem werden wir fortan des Öfteren Code präsentieren, der nur funktioniert, wenn er in **main()** oder in einer anderen Funktion steht, wie zum Beispiel:

```
cout << "Bitte geben Sie Ihren Vornamen ein (gefolgt von 'Enter'):\n";
```

Wir gehen davon aus, dass Sie mittlerweile wissen oder zumindest ohne Schwierigkeiten selbstständig herausfinden können, wie Sie diesen Code zum Testen in ein komplettes Programm einbauen.

Die erste Zeile von **main()** gibt einfach einen Text aus, der den Benutzer auffordert, seinen Vornamen einzugeben. So ein Text wird normalerweise als *Eingabeaufforderung* bezeichnet, weil er den Benutzer zu einer Aktion anhält. Die nächsten drei Zeilen definieren eine Variable vom Typ **string** namens **first_name**, lesen die Eingabe von der Tastatur in diese Variable ein und geben dann einen Gruß auf dem Bildschirm aus. Wir wollen diese drei Zeilen der Reihe nach betrachten:

```
string first_name;  // first_name ist eine Variable vom Typ string
```

Mit dieser Anweisung reservieren Sie einen Speicherbereich für einen String und nennen ihn **first_name** (siehe ▶ Abbildung 3.2).

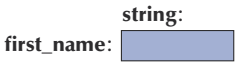


Abbildung 3.2: Eine String-Variable namens **first_name**

Eine Anweisung, die einen neuen Namen in einem Programm einführt und Speicher für eine Variable reserviert, wird auch *Definition* genannt.

Die nächste Zeile liest Zeichen von der Eingabe (hier die Tastatur) in diese Variable:

```
cin >> first_name;  // lies die Zeichen in first_name
```

Die Bezeichnung **cin** (ausgesprochen „ssi-in“ für „**c**haracter **in**put“, Zeicheneingabe) bezieht sich auf den Standardeingabestream, der in der Standardbibliothek definiert ist. Der zweite Operand des **>>**-Operators gibt an, wo die Eingabe abgelegt werden soll. Wenn wir zum Beispiel als Vornamen „Nicholas“ gefolgt von einem Zeilenumbruch angeben, wird der String **"Nicholas"** zum Wert von **first_name** (siehe ▶ Abbildung 3.3).

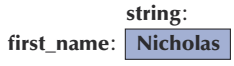


Abbildung 3.3: Eine String-Variable namens **first_name** mit dem Wert **Nicholas**

Tipp

Der Zeilenumbruch, d.h. das Drücken der ↵-Taste, ist nötig, um den Rechner aktiv werden zu lassen. Bis zur Eingabe des Zeilenumbruchs sammelt der Rechner einfach nur Zeichen. Diese „Verzögerung“ gibt Ihnen die Möglichkeit, Ihre Meinung zu ändern, Zeichen zu löschen oder durch andere zu ersetzen, bevor Sie die ↵-Taste drücken. Der Zeilenumbruch ist nicht Teil des Strings, den Sie im Speicher ablegen.

Nachdem wir den Eingabestring in **first_name** gespeichert haben, können wir ihn verwenden:

```
cout << "Hallo " << first_name << "!\n";
```

Die Ausgabe, die diese Anweisung auf den Bildschirm schreibt, lautet „Hallo“, gefolgt von „Nicholas“ (dem Wert von **first_name**), gefolgt von einem „!“ und einem Zeilenumbruch (**'\n'**):

```
Hallo Nicholas!
```

Wären wir Fans von Wiederholungen und zusätzlicher Tipparbeit, hätten wir auch drei getrennte Ausgabeanweisungen schreiben können:

```
cout << "Hallo ";
cout << first_name;
cout << "!\n";
```

Doch da wir grundsätzlich faul sind und vor allem unnötige Wiederholungen vermeiden wollen (da Wiederholungen nur zu weiteren Fehlern einladen), haben wir diese drei Ausgabeoperationen zu einer Anweisung zusammengefasst.

Achten Sie darauf, dass wir um **"Hallo "** Anführungszeichen setzen, aber nicht um **first_name**. Wir verwenden Anführungszeichen, wenn wir ein *String-Literal* benötigen (d.h. eine Zeichenfolge, die der Compiler als nichts weiter als eben eine Zeichenfolge behandelt). Ohne Anführungszeichen beziehen wir uns auf den Wert von etwas mit einem Namen. Betrachten Sie hierzu:

```
cout << "first_name" << " ist " << first_name;
```

In diesem Beispiel liefert uns **"first_name"** die zehn Zeichen „first_name“, während einfach nur **first_name** uns den Wert der Variablen **first_name** zurückliefert, in diesem Fall „Nicholas“. Das Ergebnis würde lauten:

```
first_name ist Nicholas
```

3.2 Variablen

Im Grunde genommen ist die Arbeit mit einem Computer nur dann interessant, wenn wir Daten, wie den Eingabestring im vorhergehenden Beispiel, im Speicher ablegen können. Die „Orte“, in denen wir Daten speichern, werden *Objekte* genannt. Um auf ein Objekt zuzugreifen, benötigen wir einen *Namen*. Ein benanntes Objekt wird als *Variable* bezeichnet und hat einen spezifischen *Typ* (wie **int** oder **string**), der festlegt, was in dem Objekt abgelegt werden kann (z.B. kann eine **int**-Variable den Wert **123** und eine **string**-Variable den Wert **"Hello, World!\n"** annehmen) und welche Operationen damit ausgeführt werden können (z.B. können wir **int**-Variablen mit dem *****-Operator multiplizieren und **string**-Variablen mit dem **<=**-Operator vergleichen). Die Datenelemente, die wir in den Variablen ablegen, werden *Werte* genannt. Eine Anweisung, die eine Variable definiert, wird (logischerweise) *Definition* genannt. Definitionen können (und in der Regel sollten sie dies auch) einen Anfangswert vorgeben:

```
string name = "Annemarie";
int number_of_steps = 39;
```

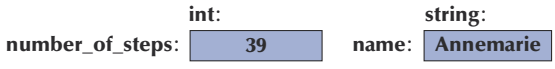


Abbildung 3.4: Zwei Variablen nach der Definition (inklusive Anfangswert)

Sie können einer Variablen keinen Wert zuweisen, der den falschen Typ hat:

```
string name2 = 39;                  // Fehler: 39 ist kein String
int number_of_steps = "Annemarie";  // Fehler: "Annemarie" ist kein Integer
```

Der Compiler merkt sich von jeder Variablen den Typ und stellt sicher, dass Sie die Variable nur gemäß ihrem definierten Typ verwenden.

C++ stellt Ihnen eine ziemlich große Auswahl an Typen zur Verfügung (siehe §A.8). In der Regel bedarf es allerdings nur fünf dieser Typen, um Programme zu schreiben, die allen Ansprüchen genügen:

```
int number_of_steps = 39;      // int für ganze Zahlen (Integer)
double flying_time = 3.5;      // double für Gleitkommazahlen[1]
char decimal_point = '.';      // char für einzelne Zeichen
string name = "Annemarie";     // string für Zeichenfolgen (Strings)
bool tap_on = true;            // bool für logische Variablen
```

Die Bezeichnung **double** hat historische Gründe: **double** ist die Kurzform von „double-precision floating point“. Gleitkomma ist die Approximation des Computers an das mathematische Konzept der reellen Zahlen.

Beachten Sie, dass jeder Typ seine eigene Form von Literal hat. (*Literale* sind Werte, die für sich selbst stehen – im Gegensatz zu Variablen, die für den Wert stehen, der in ihnen gespeichert ist.)

```
39            // int: ein ganze Zahl
3.5           // double: eine Gleitkommazahl
'.'           // char: ein einzelnes Zeichen in einfachen Anführungszeichen
"Annemarie"   // string: eine Folge von Zeichen in doppelten Anführungszeichen
true          // bool: entweder true oder false
```

Demzufolge repräsentiert eine Folge von Ziffern (wie **1234**, **2** oder **976**) eine ganze Zahl (Integer-Wert); ein einzelnes Zeichen in einfachen Anführungszeichen (wie **'1'**, **'@'** oder **'x'**) repräsentiert ein Zeichen, eine Folge von Ziffern mit Dezimalzeichen (wie **1.234**, **0.12** oder **.98**) repräsentiert eine Gleitkommazahl und eine Folge von Zeichen in doppelten Anführungszeichen (wie **"1234"**, **"Hoppla!"** oder **"Annemarie"**) repräsentiert einen String. Eine ausführliche Beschreibung der Literale finden Sie in §A.2.

3.3 Eingabe und Typ

Die Eingabeoperation **>>** („holen von“) ist typbewusst, d.h., sie liest die Eingabe so, wie der Typ der Variablen, in die Sie einlesen, es verlangt. Zum Beispiel:

```
// liest Name und Alter ein
int main()
{
  cout << "Bitte geben Sie Ihren Vornamen und Ihr Alter ein\n";
  string first_name;     // String-Variable
  int age;               // Integer-Variable
  cin >> first_name;     // lies einen String ein
  cin >> age;            // lies einen Integer ein
  cout << "Hallo " << first_name << " (Alter " << age << ")\n";
}
```

1 C++ verwendet für Dezimalbrüche die angloamerikanische Notation, mit dem Punkt als Dezimal- und dem Komma als Tausendertrennzeichen.

Wenn Sie also „Carlos 22“ eingeben, liest der >>-Operator „Carlos“ in **first_name** ein und „22“ in **age** und erzeugt folgende Ausgabe:

Hallo Carlos (Alter 22)

Warum wird „Carlos 22“ nicht (ganz) in **first_name** eingelesen? Das liegt daran, dass per Konvention das Einlesen von Strings durch ein sogenanntes *Whitespace*-Zeichen (dazu gehören Leerzeichen, Zeilenumbruch und Tabulatorzeichen) beendet wird. Ansonsten werden Whitespace-Zeichen von >> standardmäßig ignoriert. So können Sie zum Beispiel so viele Leerzeichen vor einer einzulesenden Zahl einfügen, wie Sie wollen. Der >>-Operator wird sie alle überspringen und nur die Zahl einlesen.

Wenn Sie „22 Carlos“ eingeben, werden Sie von dem Ergebnis vermutlich ziemlich überrascht sein, bis Sie etwas genauer darüber nachdenken. Die „22“ wird in **first_name** eingelesen, da „22“ nicht nur eine Zahl, sondern auch eine Folge von Zeichen ist. Andererseits ist „Carlos“ keine ganze Zahl, sodass diese Eingabe nicht eingelesen wird. Die Ausgabe lautet „22“ gefolgt von einer Zufallszahl wie „–96739“ oder „0“. Warum? Sie haben **age** keinen Anfangswert zugewiesen und konnten in diese Variable auch keinen Wert einlesen. Deshalb erhalten Sie einen unsinnigen Wert, der zufällig in dem Teil des Speichers stand, als Sie mit der Ausführung des Programms begannen. In §10.6 untersuchen wir Möglichkeiten, wie Sie mit „Eingabeformatfehlern“ umgehen. Im Moment beschränken wir uns darauf, **age** einfach zu initialisieren, sodass wir einen vorhersagbaren Wert erhalten, falls die Eingabe fehlschlägt:

```
// liest Name und Alter ein (zweite Version)
int main()
{
  cout << "Bitte geben Sie Ihren Vornamen und Ihr Alter ein\n";
  string first_name = "???";      // String-Variable
                                  // ("???" bedeutet "kenne den Namen nicht")
  int age = –1;                   // Integer-Variable (–1 bedeutet "kenne das Alter nicht")
  cin >> first_name >> age;       // lies einen String gefolgt von einem Integer ein
  cout << "Hallo " << first_name << " (Alter " << age << ")\n";
}
```

Jetzt erzeugt die Eingabe „22 Carlos“ folgende Ausgabe:

Hallo 22 (Alter –1)

Beachten Sie, dass wir mehrere Werte in einer einzigen Eingabeanweisung einlesen können, so wie wir auch mehrere Werte in einer einzelnen Anweisung ausgeben können. Beachten Sie weiterhin, dass der <<-Operator wie der >>-Operator den Typ berücksichtigt. Deshalb können wir in einem Zug die **int**-Variable **age**, die **string**-Variable **first_name** und die String-Literale **"Hallo "**, **" (Alter "** und **")\n"** ausgeben.

Tipp

Strings, die mit >> eingelesen werden, enden (standardmäßig) beim ersten Whitespace-Zeichen, d.h., es wird nur ein einzelnes Wort gelesen. Sollen mehrere Wörter eingelesen werden, gibt es dafür verschiedene Möglichkeiten. Eine wäre zum Beispiel einen Namen, der aus zwei Worten besteht, wie folgt einzulesen:

```
int main()
{
  cout << "Bitte geben Sie Ihren Vor- und Nachnamen ein\n";
  string first;
  string second;
  cin >> first >> second;  // lies zwei Strings ein
  cout << "Hallo " << first << ' ' << second << '\n';
}
```

Wir haben einfach >> zweimal benutzt, einmal für jeden Namen. Bei der Ausgabe der Namen müssen wir allerdings darauf achten, dazwischen ein Leerzeichen (das Zeichenliteral ' ') zu setzen.

Testen Sie Ihr Können

Versuchen Sie, das *Name und Alter*-Beispiel auszuführen. Ändern Sie das Programm anschließend so, dass es das Alter in Monaten ausgibt: Lesen Sie dazu die Eingabe in Jahren ein und multiplizieren Sie diesen Wert (mithilfe des *-Operators) mit 12. Lesen Sie das Alter in eine **double**-Variable ein, um Kindern, die stolz darauf sind, bereits fünfeinhalb (5.5) statt fünf Jahre alt zu sein, eine Freude zu machen.

3.4 Operationen und Operatoren

Der Typ einer Variablen gibt nicht nur an, welche Werte in der Variablen gespeichert werden können. Er legt auch fest, welche Operationen wir darauf anwenden können und wie diese auszuführen sind. Zum Beispiel:

```
int count;
cin >> count;              // >> liest einen Integer in count ein
string name;
cin >> name;               // >> liest einen String in name ein

int c2 = count+2;          // + addiert Integer-Werte
string s2 = name + " Jr."; // + hängt Zeichen an

int c3 = count–2;          // – subtrahiert Integer-Werte
string s3 = name – "Jr.";  // Fehler: – ist für Strings nicht definiert
```

Mit „Fehler“ meinen wir, dass Programme, die versuchen, Strings zu subtrahieren, vom Compiler abgelehnt werden. Der Compiler weiß genau, welche Operationen für die jeweiligen Variablen zulässig sind, und kann deshalb viele Fehler verhindern. Allerdings weiß der Compiler nicht, welche Operationen für welche Art von Werten vernünftig sind. Vorbehaltlos akzeptiert er alle zulässigen Operationen, auch wenn Sie über die Ergebnisse nur den Kopf schütteln können, wie zum Beispiel in:

```
int age = –100;
```

Ihnen mag klar sein, dass es kein negatives Alter gibt (warum eigentlich nicht?), aber niemand hat dies dem Compiler mitgeteilt, sodass er anstandslos Code für diese Definition erzeugt.

▶Tabelle 3.1 listet die wichtigsten Operatoren für einige der häufiger benutzten Typen auf. (Ein leeres Feld in der Tabelle weist darauf hin, dass diese Operation für den Typ nicht zur Verfügung steht. Allerdings kann man bestimmte Operationen über Umwege doch ausführen, siehe §3.9.1.)

Tabelle 3.1

Ausgesuchte Operatoren

	bool	char	int	double	string
Zuweisung	=	=	=	=	=
Addition			+	+	
Verkettung (Konkatenation)					+
Subtraktion			-	-	
Multiplikation			*	*	
Division			/	/	
Modulo (Rest)			%		
Inkrement um 1			++	++	
Dekrement um 1			--	--	
Inkrement um **n**			+=n	+=n	
Anhängen					+=
Dekrement um **n**			-=n	-=n	
Multiplizieren und zuweisen			*=	*=	
Dividieren und zuweisen			/=	/=	
Modulo und zuweisen			%=		
Von **s** in **x** einlesen	s>>x	s>>x	s>>x	s>>x	s>>x
Von **x** nach **s** schreiben	s<<x	s<<x	s<<x	s<<x	s<<x
Gleich	==	==	==	==	==
Nicht gleich	!=	!=	!=	!=	!=
Größer als	>	>	>	>	>
Größer als oder gleich	>=	>=	>=	>=	>=
Kleiner als	<	<	<	<	<
Kleiner als oder gleich	<=	<=	<=	<=	<=

Wir werden diese Operationen und alles, was damit zusammenhängt, nach und nach erläutern. Merken Sie sich an dieser Stelle nur, dass es eine ganze Reihe von nützlichen Operatoren gibt und dass ihre Bedeutung für ähnliche Typen mehr oder weniger gleich ist.

Versuchen wir uns noch an einem Beispiel mit Gleitkommazahlen:

```
// einfaches Programm zum Ausprobieren von Operatoren
int main()
{
  cout << "Geben Sie einen Gleitkommawert ein: ";
  double n;
  cin >> n;
  cout << "n == " << n
    << "\nn+1 == " << n+1
    << "\ndreimal n == " << 3*n
    << "\nzweimal n == " << n+n
    << "\nn zum Quadrat == " << n*n
    << "\nHaelfte von n == " << n/2
    << "\nWurzel von n == " << sqrt(n)
    << endl;  // ein anderer Name für Zeilenumbruch ("end of line")
}
```

Offensichtlich unterscheiden sich die einfachen arithmetischen Operationen in Notation und Anwendung nicht von dem, was wir aus der Grundschule kennen. Natürlich gibt es nicht für alle Operationen, die wir mit einer Gleitkommazahl ausführen wollen (z.B. Quadratwurzel ziehen), einen Operator. Viele Operationen gibt es nur als benannte Funktionen. Im obigen Fall verwenden wir **sqrt()** aus der Standardbibliothek, um die Quadratwurzel von **n** zu erhalten: **sqrt(n)**[2]. Die Notation dürfte Ihnen von der Mathematik her bekannt sein, sodass wir es uns erlauben, nebenher bereits Funktionen zu verwenden – auch wenn das Konzept der Funktionen erst in §4.5 und §8.5 ausführlicher behandelt wird.

Testen Sie Ihr Können

Führen Sie das obige kleine Programm aus. Ändern Sie es anschließend so, dass es einen **int**-Wert anstelle eines **double**-Wertes einliest. Beachten Sie, dass **sqrt()** nicht für **int**-Werte definiert ist; weisen Sie also **n** einer **double**-Variablen zu und ziehen Sie daraus die Quadratwurzel. Führen Sie zur Übung auch einige andere Operationen aus. Beachten Sie, dass für **int**-Werte **/** eine Integer-Division ist und **%** eine Modulo-Operation; d.h., **5/2** ist **2** (und nicht **2.5** oder **3**) und **5%2** ist **1**. Die Definitionen der Integer-Operationen *****, **/** und **%** stellen sicher, dass für zwei positive **int**-Werten **a** und **b** die Aussage gilt: **a/b*b + a%b == a**.

Für Strings gibt es zwar wenige Operatoren, dafür aber eine große Anzahl an benannten Operationen (siehe Kapitel 23). Immerhin, die Operatoren, die für Strings zur Verfügung stehen, können Sie wie gewohnt bzw. erwartet verwenden. Zum Beispiel:

```
// liest Vor- und Nachnamen ein
int main()
{
  cout << "Bitte geben Sie Ihren Vor- und Nachnamen ein\n";
  string first;
  string second;
  cin >> first >> second;              // lies zwei Strings ein
  string name = first + ' ' + second;  // verkette die Strings
  cout << "Hallo " << name << '\n';
}
```

2 Der Name „sqrt“ steht für den englischen Ausdruck „square root“, zu Deutsch Quadratwurzel.

Für Strings bedeutet + Verkettung (Konkatenation), d.h., wenn **s1** und **s2** Strings sind, dann ergibt **s1+s2** einen String, bei dem die Zeichen von **s2** auf die Zeichen von **s1** folgen. Wenn also **s1** den Wert **"Hallo"** hätte und **s2** den Wert **"Welt"**, dann hätte **s1+s2** den Wert **"HalloWelt"**. Besonders nützlich sind String-Vergleiche:

```
// liest und vergleicht Namen
int main()
{
  cout << "Geben Sie zwei Namen ein\n";
  string first;
  string second;
  cin >> first >> second;  // lies zwei Strings ein
  if (first == second) cout << "Zweimal derselbe Name\n";
  if (first < second)
    cout << first << " kommt im Engl. alphabetisch vor " << second <<'\n';
  if (first > second)
    cout << first << " kommt im Engl. alphabetisch nach " << second <<'\n';
}
```

In diesem Beispiel haben wir thematisch etwas vorgegriffen und eine **if**-Anweisung verwendet (die in §4.4.1.1 näher erläutert wird), um mithilfe von Bedingungen zu steuern, welche Anweisung ausgeführt wird.

3.5 Zuweisung und Initialisierung

Der Zuweisungsoperator, der durch das Symbol **=** repräsentiert wird, ist in vielerlei Hinsicht der interessanteste Operator. Seine Aufgabe ist es, einer Variablen einen neuen Wert zuzuweisen. Zum Beispiel:

```
int a = 3;  // a beginnt mit dem Wert 3
```

a: 3

```
a = 4;      // a erhält den Wert 4 ("wird zu 4")
```

a: 4

```
int b = a;  // b beginnt mit einer Kopie des Wertes von a (d.h. 4)
```

a: 4
b: 4

```
b = a+5;    // b erhält den Wert a+5 (d.h. 9)
```

a: 4
b: 9

```
a = a+7;    // a erhält den Wert a+7 (d.h. 11)
```

a: 11
b: 9

Die letzte Anweisung ist es wert, dass man sie näher betrachtet. Zum einen zeigt sie deutlich, dass **=** nicht „ist gleich“ bedeutet, denn **a** ist nicht gleich **a+7**. Es bedeutet Zuweisung, d.h., in eine Variable wird ein neuer Wert geschrieben. Untersuchen wir, was bei der Zuweisung **a=a+7** im Einzelnen geschieht:

1 Ermittle zuerst den Wert von **a**; dies liefert den Integer-Wert 4.

2 Dann addiere zu der 4 den Wert 7; dies ergibt die ganze Zahl 11.

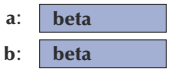
3 Zum Schluss schreibe die 11 in **a**.

Wir können den Zuweisungsoperator auch für Strings einsetzen:

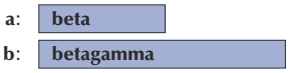
```
string a = "alpha";   // a beginnt mit dem Wert "alpha"
```


```
a = "beta";           // a erhält den Wert "beta" (wird zu "beta")
```

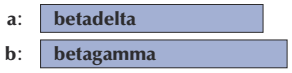

```
string b = a;         // b beginnt mit einer Kopie des Wertes von a (d.h. "beta")
```




```
b = a+"gamma";        // b erhält den Wert a+"gamma" (d.h. "betagamma")
```




```
a = a+"delta";        // a erhält den Wert a+"delta" (d.h. "betadelta")
```



Oben verwenden wir „beginnt mit“ und „erhält“, um zwei gleiche, aber logisch verschiedene Operationen zu unterscheiden:

- Initialisierung (eine Variable erhält ihren Anfangswert)
- Zuweisung (eine Variable erhält einen neuen Wert)

Diese Operationen sind so ähnlich, dass C++ uns erlaubt, hierfür die gleiche Notation (d.h. **=**) zu verwenden:

```
int y = 8;            // initialisiert y mit 8
x = 9;                // weist 9 der Variablen x zu

string t = "hi!";     // initialisiert t mit "hi!"
s = "Guten Tag";      // weist "Guten Tag" der Variablen s zu
```

Logisch gesehen, stellen Zuweisung und Initialisierung jedoch zwei unterschiedliche Operationen dar. Wenn Sie auf die Typangabe achten (wie **int** oder **string**), können Sie beide leicht auseinanderhalten: Die Initialisierung beginnt immer mit einer Typangabe, der Zuweisung fehlt sie. Eine Initialisierung trifft immer auf eine leere Variable, während bei einer Zuweisung (vom Prinzip her) der alte Wert zuerst aus der Variablen entfernt werden muss, bevor der neue Wert darin abgelegt werden kann. Stellen Sie sich eine Variable als eine Schachtel oder ein kleines Kästchen vor, in das der Wert wie ein konkreter Gegenstand (z.B. eine Münze) hineingelegt wird. Vor der Initialisierung ist das Kästchen leer, doch einmal initialisiert, liegt darin immer eine Münze. Folglich müssen Sie (d.h. der Zuweisungsoperator) immer zuerst

die alte Münze herausnehmen („den alten Wert zerstören“), bevor Sie eine neue Münze hineinlegen können. Außerdem dürfen Sie das Kästchen nicht leer lassen. Dies spiegelt zwar nicht exakt die Vorgänge im Speicher des Computers wider, soll uns aber zur Veranschaulichung reichen.

3.5.1 Ein Beispiel: Wortwiederholungen löschen

Zuweisungen dienen dazu, in Objekten neue Werte abzulegen. Denkt man über diesen Satz ein wenig nach, kommt man schnell darauf, dass Zuweisungen vor allem dann benötigt werden, wenn eine bestimmte Aufgabe mehrfach zu verrichten ist, d.h., wenn wir eine Aufgabe mit einem anderen Wert wiederholen wollen. Lassen Sie uns dazu ein kleines Programm betrachten, das zwei aufeinanderfolgende identische Wörter in einer Folge von Wörtern entdeckt. Code wie dieser ist Bestandteil der meisten Grammatik-Prüfprogramme:

```
int main()
{
  string previous = " ";        // vorheriges Wort; initialisiert mit "kein Wort"
  string current;               // aktuelles Wort
  while (cin>>current) {        // lies einen Strom von Worten
    if (previous == current)      // prüfe, ob das Wort das gleiche ist wie das letzte
      cout << "Wortwiederholung: " << current << '\n';
    previous = current;
  }
}
```

Dieses Programm ist sicherlich nicht besonders hilfreich, da es nicht mitteilt, wo im Text die Wortwiederholungen auftreten, aber für unsere Zwecke soll es genügen. Gehen wir das Programm Zeile für Zeile durch:

```
string current;    // aktuelles Wort
```

Dies ist die String-Variable, in der wir mit

```
while (cin>>current)
```

gleich darauf das aktuelle (d.h. das zuletzt eingelesene) Wort ablegen. Die hier verwendete Konstruktion, eine sogenannte **while**-Anweisung, ist für sich genommen ebenfalls sehr interessant, weswegen wir in §4.4.2.1 noch einmal auf sie zurückkommen. Das **while** besagt hier, dass die Anweisung nach **(cin>>current)** so lange wiederholt wird, wie die Eingabeoperation **cin>>current** erfolgreich ausgeführt wird – und **cin>>current** wird so lange ausgeführt, wie es Zeichen von der Standardeingabe einzulesen gibt. Denken Sie daran, dass der **>>**-Operator für **string**-Variablen Wörter einliest, die durch ein Whitespace-Zeichen getrennt sind. Sie beenden diese Schleife, indem Sie dem Programm mit einem besonderen Zeichen, dem sogenannten EOF-Zeichen (EOF steht für „End-of-File“), das Ende der Eingabe signalisieren. Auf Windows-Computern drücken Sie dazu Strg+Z (d.h. Strg-Taste und Z gemeinsam gedrückt) gefolgt von der ↵-Taste. Auf Unix- oder Linux-Rechnern müssen Sie Strg+D zusammen drücken.

Tipp

Was wir also tun, ist Folgendes: Wir lesen ein Wort in die Variable **current** und vergleichen es mit dem vorherigen Wort (gespeichert in **previous**). Wenn die beiden Wörter identisch sind, melden wir dies:

```
if (previous == current)    // prüfe, ob das Wort das gleiche ist wie das letzte
  cout << "Wortwiederholung: " << current << '\n';
```

Anschließend müssen wir Vorkehrungen treffen, um diesen Schritt für das nächste Wort zu wiederholen. Dazu kopieren wir den Inhalt von **current** in **previous**:

```
previous = current;
```

Damit sind alle Fälle abgedeckt, vorausgesetzt, wir können überhaupt beginnen. Denn noch ist nicht geklärt, was dieser Code mit dem ersten Wort machen soll, für das es ja noch keinen Vergleichswert gibt. Das Problem wird durch die Definition von **previous** gelöst:

```
string previous = " "; // vorheriges Wort; initialisiert mit "kein Wort"
```

Die Anführungszeichen " " enthalten nur ein einziges Zeichen, das Leerzeichen (das wir erhalten, wenn wir die Leertaste auf unserer Tastatur anschlagen). Da der Eingabeoperator >> Whitespace-Zeichen überspringt, können wir diese Zeichen unmöglich aus der Eingabe einlesen. Wenn wir also das erste Mal die **while**-Anweisung durchlaufen, schlägt der Test

```
if (previous == current)
```

fehl (wie es unsere Absicht war).

Testen Sie Ihr Können

Führen Sie das Programm auf einem Blatt Papier aus. Verwenden Sie als Eingabe den Satz „Die Katze Katze sprang". Sogar erfahrene Programmierer verwenden diese Technik, um sich darüber klar zu werden, was kleinere Codefragmente, deren Bedeutung nicht offensichtlich ist, eigentlich tun.

Tipp

Eine Möglichkeit, den Programmfluss zu verstehen, besteht darin, „Computer zu spielen", d.h., dem Programm Zeile für Zeile zu folgen und genau das zu machen, was der Code angibt. Zeichnen Sie dazu Kästchen auf ein Blatt Papier und tragen Sie in diese die Variablenwerte ein. Ändern Sie dann die Werte wie vom Programmcode angegeben.

Testen Sie Ihr Können

Führen Sie das Programm zur Erkennung von Wortwiederholungen aus. Testen Sie es mit dem Satz: „Sie sie lachte Er Er Er weil was er sah sah nicht sehr sehr gut gut aus". Wie viele Wortwiederholungen sind in diesem Satz enthalten? Warum? Wie lautet hierbei die Definition von *Wort*? Wie lautet die Definition von *Wortwiederholung*? (Ist zum Beispiel „Sie sie" eine Wortwiederholung?)

3.6 Zusammengesetzte Zuweisungsoperatoren

Eine Variable um einen bestimmten Wert zu erhöhen (zum Beispiel mit 1 addieren), ist eine so häufige Operation in Programmen, dass C++ dafür eine spezielle Syntax anbietet. Zum Beispiel ist

```
++counter
```

gleichbedeutend mit

```
counter = counter + 1
```

Es gibt viele weitere, häufig verwendete Arten und Weisen, den Wert einer Variablen auf der Basis ihres aktuellen Werts zu ändern. Wir könnten beispielsweise 7 addieren, 9 subtrahieren oder mit 2 multiplizieren. Auch diese Operationen werden direkt von C++ unterstützt:

```
a += 7;   // gleichbedeutend mit a = a+7
b –= 9;   // gleichbedeutend mit b = b–9
c *= 2;   // gleichbedeutend mit c = c*2
```

Im Allgemeinen gilt für jeden binären Operator **oper**, dass **a oper= b** gleichbedeutend ist mit **a = a oper b** (siehe §A.5). Für den Anfang erhalten wir durch diese Regel zusätzlich die Operatoren **+=**, **-=**, ***=**, **/=** und **%=**. Damit steht uns eine angenehm kompakte Notation zur Verfügung, die unsere Vorstellungen ohne Umschweife wiedergibt. Manchmal schlägt sich dies auch im Sprachgebrauch nieder: Zum Beispiel werden die Operationen ***=** und **/=** in vielen Anwendungsbereichen als „Skalierung“ tituliert.

3.6.1 Ein Beispiel: Wortwiederholungen nummerieren

Betrachten wir noch einmal das obige Beispiel, das nebeneinanderliegende Wortwiederholungen erkennt. Wir könnten es verbessern, indem wir angeben, wo die Wortwiederholung in der Reihenfolge aufgetaucht ist. Eine einfache Variation des Programms zählt die Wörter und gibt die Position der Wortwiederholung aus:

```
int main()
{
  int number_of_words = 0;
  string previous = " ";        // kein Wort
  string current;
  while (cin>>current) {
    ++number_of_words;    // erhöht die Wortzählung
    if (previous == current)
      cout << "Wort Nummer " << number_of_words
      << " ist eine Wiederholung: "<< current << '\n';
    previous = current;
  }
}
```

Wir beginnen unseren Wortzähler mit 0. Jedes Mal, wenn wir auf ein neues Wort treffen, inkrementieren wir den Zähler:

```
++number_of_words;
```

Auf diese Weise erhält das erste Wort die Nummer 1, das nächste die Nummer 2 und so weiter. Das Gleiche hätten wir durch folgenden Code erreicht:

```
number_of_words += 1;
```

oder sogar mit:

```
number_of_words = number_of_words+1;
```

aber **++number_of_words** ist kürzer und drückt die Vorstellung der Inkrementierung direkt aus.

Auffällig ist, wie ähnlich dieser Code dem Programm aus §3.5.1 ist. Offenbar haben wir einfach das Programm von §3.5.1 genommen und für unsere Zwecke etwas abgeändert. Diese Technik ist sehr verbreitet. Wenn wir vor einem Problem stehen, suchen wir erst einmal nach einem ähnlichen Problem samt Lösung, die wir dann an unsere Bedürfnisse anpassen. Fangen Sie nur dann ganz von vorn an, wenn es absolut notwendig ist. Das Heranziehen einer bereits vorhandenen Programmversion als Ausgangsbasis für die eigenen Änderungen spart in der Regel eine Menge Zeit und man profitiert beträchtlich von den Anstrengungen, die in das ursprüngliche Programm geflossen sind.

3.7 Namen

Wir geben unseren Variablen Namen, damit wir sie uns merken und aus anderen Teilen eines Programms darauf Bezug nehmen können. Wie sieht so ein Name in C++ aus? Ein Name in einem C++-Programm beginnt mit einem Buchstaben und enthält nur Buchstaben, Ziffern und Unterstriche. Zum Beispiel:

```
x
number_of_elements
Fourier_transform
z2
Polygon
```

Die folgenden Beispiele sind keine Namen:

```
2x              // ein Name muss mit einem Buchstaben beginnen
time$to$market  // $ ist weder Buchstabe noch Unterstrich noch Ziffer
Start menu      // ein Leerzeichen ist weder Buchstabe noch Unterstrich noch Ziffer
```

Mit „keine Namen“ meinen wir, dass ein C++-Compiler diese Namen nicht als Namen akzeptiert.

Wenn Sie Systemcode oder maschinenerzeugten Code lesen, stoßen Sie unter Umständen auf Namen, die mit Unterstrichen beginnen, wie **_foo**. Diese Schreibweise sollten Sie unbedingt vermeiden, denn diese Namen sind für Implementierungs- und Systementitäten reserviert. Indem Sie führende Unterstriche vermeiden, laufen Sie auch nie Gefahr, dass Ihre Namen mit Namen kollidieren, die von der Implementierung erzeugt wurden.

Der C++-Compiler berücksichtigt die Klein- und Großschreibung; das heißt, Groß- und Kleinbuchstaben werden unterschieden, sodass z.B. **x** und **X** unterschiedliche Namen sind. Das folgende kleine Programm weist mindestens vier Fehler auf:

```
#include "std_lib_facilities.h"

int Main()
{
  STRING s = "Goodbye, cruel world! ";
  cOut << S << '\n';
}
```

Grundsätzlich ist davon abzuraten, Namen zu definieren, die sich nur in der Groß- und Kleinschreibung eines Zeichens unterscheiden, wie **one** und **One**. Den Compiler wird dies zwar nicht irritieren, aber höchstwahrscheinlich den Programmierer.

Testen Sie Ihr Können

Kompilieren Sie das *Goodbye, cruel world*-Programm und untersuchen Sie die Fehlermeldungen. Hat der Compiler alle Fehler gefunden? Wo lagen seiner Meinung nach die Probleme? Hat irgendetwas den Compiler verwirrt, sodass er mehr als vier Fehler gefunden hat? Entfernen Sie die Fehler einen nach dem anderen, wobei Sie mit dem ersten beginnen und jedes Mal neu kompilieren. Beobachten Sie, wie sich die Fehlermeldungen ändern (und weniger werden).

Die Sprache C++ reserviert viele Bezeichner (über 70) als „Schlüsselwörter“. Eine Liste dieser Wörter finden Sie in §A.3.1. Diese Bezeichner dürfen nicht als Namen von Variablen, Typen, Funktionen usw. verwendet werden. Zum Beispiel:

```
int if = 7;      // Fehler: "if" ist ein Schlüsselwort
```

Die Namen von Elementen der Standardbibliothek, wie **string**, können Sie verwenden, sollten aber davon Abstand nehmen. Die Wiederverwendung eines solch häufig verwendeten Bezeichners wird nur unnötig Ärger bereiten, wenn Sie jemals die Standardbibliothek verwenden wollen:

```
int string = 7;   // wird später Probleme bereiten
```

Tipp

Wählen Sie für Ihre Variablen, Funktionen, Typen usw. aussagekräftige Namen, d.h. solche, die es anderen leichter machen, Ihre Programme zu verstehen. Sogar Sie selbst werden eines Tages Mühe haben nachzuvollziehen, was Ihre Programme eigentlich tun sollen, wenn Sie sie mit Variablen übersäen, deren Namen vor allem leicht einzutippen sind, wie **x1**, **x2**, **s3** und **p7**. Abkürzungen und Akronyme können zu Verwechslungen führen und sollten deshalb nur sparsam verwendet werden. Die folgenden Akronyme erschienen *uns* logisch, als wir sie in unseren Programmen verwendeten, aber *Sie* werden vermutlich mit einigen von ihnen Schwierigkeiten haben – und in einigen Monaten dürften sogar wir selbst mit ihnen Schwierigkeiten haben.

mtbf
TLA
myw
NBV

Kurze Bezeichnungen wie **x** und **i** sind aussagekräftig, wenn sie wie allgemein üblich verwendet werden; d.h., **x** sollte als lokale Variable oder Parameter zum Einsatz kommen (siehe §4.5 und §8.4) und **i** als Schleifenindex (siehe §4.4.2.3).

Achten Sie außerdem darauf, dass Ihre Namen nicht zu lang werden, denn damit erhöhen Sie die Fehleranfälligkeit, machen die Zeilen nur unnötig lang, sodass sie nicht auf den Bildschirm passen, und erschweren die Lesbarkeit. Die folgenden Beispiele sind mehr oder weniger akzeptabel:

partial_sum
element_count
stable_partition

Diese sind wahrscheinlich zu lang:

the_number_of_elements
remaining_free_slots_in_symbol_table

Wir persönlich verwenden Unterstriche, um die einzelnen Wörter in einem Bezeichner optisch zu trennen (beispielsweise **element_count**). Es gibt jedoch auch alternative Schreibweisen wie **elementCount** und **ElementCount**. Wir verwenden grundsätzlich keine Namen, die vollständig in Großbuchstaben geschrieben werden, wie **ALL_CAPITAL_LETTERS**, da diese Schreibweise normalerweise für Makros reserviert ist (siehe §27.8 und §A.17.2), die wir vermeiden. Typen, die wir selbst definieren (wie **Square** und **Graph**), werden von uns am Anfang großgeschrieben. In C++ und der Standardbibliothek werden keine Großbuchstaben verwendet, deshalb heißt es **int** statt **Int** und **string** statt **String**. Mit unserer Konvention wollen wir einer Verwechslung von unseren Typen mit den Standardtypen entgegenwirken.

Vermeiden Sie außerdem Namen, die zu Tippfehlern einladen, häufig falsch gelesen werden oder sonst auf irgendeine Weise Verwirrung stiften können, zum Beispiel:

Name	**names**	**nameS**
foo	**f00**	**fl**
f1	**fl**	**fi**

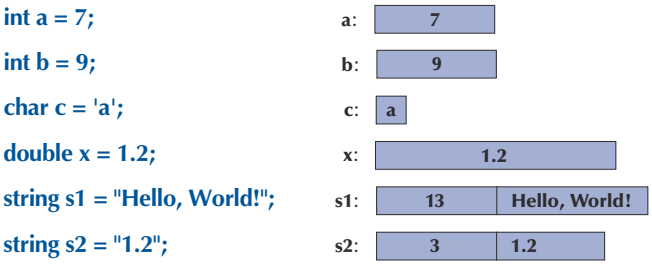
Vor allem die Zeichen **0**, **o**, **O**, **1**, **I**, **l** führen häufig zu solchen Fehlern.

3.8 Typen und Objekte

Das Konzept der Typen spielt in C++ und den meisten anderen Programmiersprachen eine sehr wichtige Rolle. Es lohnt daher, sich mit der Thematik und den technischen Details etwas näher zu befassen, vor allem mit den Typen der Objekte, in denen wir unsere Daten speichern. Langfristig gesehen wird Ihnen dies Zeit und so manche unnötige Verwirrung ersparen.

- Ein *Typ* definiert eine Menge von möglichen Werten und einen zugehörigen Satz von Operationen (für ein Objekt).
- Ein *Objekt* ist ein Speicherbereich, in dem ein Wert eines gegebenen Typs abgelegt wurde.
- Ein *Wert* ist eine Folge von Bits im Speicher, der entsprechend seines Typs interpretiert wird.
- Eine *Variable* ist ein benanntes Objekt.
- Eine *Deklaration* ist eine Anweisung, die ein Objekt mit einem Namen versieht.
- Eine *Definition* ist eine Deklaration, die für ein Objekt Speicher reserviert.

Praktisch können wir uns ein Objekt als ein Kästchen vorstellen, in das wir Werte eines gegebenen Typs hineinlegen. Ein **int**-Kästchen kann demnach ganze Zahlen wie **7**, **42** und **–399** aufnehmen und ein **string**-Kästchen Zeichenfolgen wie **"Interoperabilität"**, **"Tokens: !@#$%^&*"** und **"Old McDonald hat 'ne Farm"**. Bildlich können wir uns das folgendermaßen vorstellen:



Die Darstellung einer **string**-Variablen ist etwas komplizierter als die einer **int**-Variablen, da eine **string**-Variable sich die Anzahl der darin enthaltenen Zeichen merkt. Beachten Sie, dass eine **double**-Variable eine Zahl speichert, während eine **string**-Variable Zeichen speichert. So speichert zum Beispiel **x** die Zahl **1.2** während **s2** die drei Zeichen **'1'**, **'.'** und **'2'** speichert. Die Anführungszeichen für Zeichen und String-Literale werden nicht gespeichert.

Alle **int**-Werte sind gleich groß, d.h., der Compiler reserviert für jeden Integer-Wert einen gleich großen Speicherplatz. Auf einem normalen Heimcomputer sind das 4 Byte (32 Bit). Auch **bool**-, **char**- und **double**-Werte haben eine feste Größe. Normalerweise wird auf einem Heimcomputer ein Byte (8 Bit) für einen **bool**- oder einen **char**-Wert reserviert und 8 Byte für eine **double**-Wert. Merken Sie sich also, dass verschiedene Objekttypen unterschiedlich viel Speicherplatz belegen. Dabei fällt auf, dass ein **char**-Wert weniger Platz benötigt als ein **int**-Wert und dass ein **string**-Wert insofern von **double**, **int** und **char** abweicht, als er je nach Stringlänge unterschiedlichen Speicherbedarf hat.

Die Bedeutung der Bits im Speicher hängt vollständig davon ab, über welchen Typ wir darauf zugreifen. Stellen Sie es sich einfach so vor: Der Computerspeicher weiß nichts von unseren Typangaben, es ist einfach nur ein Speicher. Die Speicherbits erhalten erst dann eine Bedeutung, wenn wir entscheiden, wie der Speicherbereich interpretiert werden soll. Genau so verfahren wir täglich, wenn wir Zahlen verwenden. Was ist mit **12.5** gemeint? Wir wissen es nicht. Es könnten **$12.5** sein oder **12.5 cm** oder **12.5 Liter**. Erst wenn wir eine Einheit angeben, erhält die Notation **12.5** eine Bedeutung.

So kann dieselbe Folge von Bits im Speicher, die als **int**-Wert interpretiert für die Zahl **120** steht, **'x'** bedeuten, wenn man sie als **char**-Wert ansieht. Würde der Speicherbereich als **string** interpretiert, würde er überhaupt keinen Sinn ergeben und einen Laufzeitfehler auslösen, wenn wir versuchten, damit zu arbeiten. (Ein Bit ist eine Einheit im Computerspeicher, die entweder den Wert 0 oder 1 hat. Die Bedeutung der *Binärzahlen* wird in §A.2.1.1 erläutert.)

Abbildung 3.5: Die Bits des Speichers kann man grafisch als Nullen und Einsen darstellen

Die Abbildung zeigt das Bitmuster eines 32 Bit großen Speicherbereichs (ein *word*), das entweder als **int** (**120**) oder als **char** (**'x'**, wobei nur die ersten 8 Bit von rechts herangezogen werden) gelesen werden kann.

3.9 Typsicherheit

Jedes Objekt erhält bei seiner Definition einen Typ. Ein Programm – oder ein Programmabschnitt – wird als typsicher bezeichnet, wenn seine Objekte nur so verwendet werden, wie es der jeweilige Typ des Objekts vorgibt. Leider ist es auch möglich, Operationen auszuführen, die nicht typsicher sind. So wird zum Beispiel die Verwendung einer Variablen vor ihrer Initialisierung als nicht typsicher bezeichnet.

```
int main()
{
  double x;          // wir haben die Initialisierung "vergessen":
                     // der Wert von x ist nicht definiert
  double y = x;      // der Wert von y ist nicht definiert
  double z = 2.0+x;  // die Bedeutung von + und der Wert von z sind nicht definiert
}
```

C++-Implementierungen sind autorisiert, solchen Code – die Verwendung der nicht initialisierten Variablen **x** – mit einem Hardwarefehler zu quittieren. Achten Sie also darauf, Ihre Variablen immer zu initialisieren! Es gibt einige wenige – sehr wenige – Ausnahmen zu dieser Regel, beispielsweise wenn wir eine Variable sofort danach als Ziel für eine Eingabeoperation verwenden. Grundsätzlich sollten Sie sich aber angewöhnen, Ihre Variablen zu initialisieren. Das wird Ihnen eine Menge Kummer ersparen.

Absolute Typsicherheit ist das Ziel und deshalb die allgemeine Richtlinie für die Sprache. Leider kann ein C++-Compiler absolute Typsicherheit nicht garantieren. Wir können jedoch Verletzungen der Typsicherheit vermeiden, indem wir eine gute Codierpraxis mit Laufzeitprüfungen kombinieren. Ideal wäre es natürlich, nur solche Sprachfeatures zu verwenden, die der Compiler als sicher verifizieren kann: Man spricht in diesem Fall von *statischer Typsicherheit*. Leider ist dieser Ansatz zu restriktiv für die meisten interessanten Programmieraufgaben. Die naheliegende Ausweichlösung wäre, dass der Compiler implizit Code erzeugt, der zur Laufzeit die Typsicherheit prüft und etwaige Verletzungen abfängt. Da dies aber jenseits der Möglichkeiten von C++ liegt, bleibt uns nichts anderes übrig, als (typ-)unsicheren Code durch eigene Überprüfungen abzusichern. Später, wenn wir zu den Beispielen mit typunsicherem Code kommen, werden wir noch einmal auf diese spezielle Problematik hinweisen.

Tipp

Das Ideal der Typsicherheit ist für die Programmierung von immenser Bedeutung. Aus diesem Grund schenken wir dem Thema bereits so früh im Buch so viel Aufmerksamkeit. Prägen Sie sich die möglichen Fallstricke ein und versuchen Sie, sie zu umgehen.

3.9.1 Sichere Typumwandlungen

In §3.4 haben wir gesehen, dass wir **char**-Werte nicht addieren und einen **int**-Wert nicht direkt mit einem **double**-Wert vergleichen können. Über Umwege ist jedoch beides in C++ möglich. Bei Bedarf kann ein **char**- in einen **int**- und einen **int**- in einen **double**-Wert umgewandelt werden. Zum Beispiel:

```
char c = 'x';
int i1 = c;
int i2 = 'x';
```

Hier erhalten **i1** und **i2** beide den Wert **120** (der dem Integer-Wert des Zeichens **'x'** in dem sehr weitverbreiteten 8-Bit-Zeichensatz ASCII entspricht). Dies ist eine einfache und sichere Möglichkeit, die numerische Darstellung eines Zeichens zu erhalten. Wir bezeichnen diese **char**-zu-**int**-Umwandlung als sicher, da keine Information verloren geht; d.h., wir können den resultierenden **int**-Wert zurück in den **char**-Wert umwandeln und erhalten den ursprünglichen Wert.

```
char c2 = i1;
cout << c << ' ' << i1 << ' ' << c2 << '\n';
```

Die Ausgabe lautet:

```
x 120 x
```

In diesem Sinne – nämlich dass ein Wert immer in den gleichen Wert oder (für **double**) zumindest in die beste Annäherung dieses Wertes umgewandelt wird – sind die folgenden Umwandlungen sicher:

bool in **char**
bool in **int**
bool in **double**
char in **int**
char in **double**
int in **double**

Die nützlichste Umwandlung ist **int** in **double**, weil es uns dadurch möglich ist, **int**- und **double**-Werte in Ausdrücken zu mischen:

```
double d1 = 2.3;
double d2 = d1+2;   // 2 wird vor der Addition in 2.0 konvertiert
if (d1 < 0)         // 0 wird vor dem Vergleich in 0.0 konvertiert
  error("d1 ist negativ");
```

Für sehr große **int**-Werte kann es bei der Umwandlung in **double** (auf einigen Computern) zu einem Verlust in der Genauigkeit der Zahlendarstellung kommen. Doch diese Fälle sind sehr selten.

3.9.2 Unsichere Typumwandlungen

Sichere Typumwandlungen sind in der Regel ein Segen für den Programmierer und vereinfachen das Programmieren. Leider erlaubt C++ auch (implizite) unsichere Typumwandlungen. Mit „unsicher“ meinen wir, dass ein Wert implizit in einen Wert eines anderen Typs umgewandelt werden kann, der nicht mehr dem ursprünglichen Wert entspricht. Zum Beispiel:

```
int main()
{
  int a = 20000;
  char c = a;      // versuche, einen großen int in ein kleines char zu quetschen
  int b = c;
  if (a != b)      // != bedeutet "nicht gleich"
    cout << "Hoppla!: " << a << "!=" << b << '\n';
  else
    cout << "Wow! Uns stehen grosse char-Werte zur Verfuegung\n";
}
```

Solche Umwandlungen werden auch „einengende“ Umwandlungen genannt, weil sie einen Wert in ein Objekt ablegen, das sich als zu klein („eng“) erweisen könnte, um den Wert aufzunehmen. Leider geben nur wenige Compiler für die unsichere Initialisierung einer **char**-Variablen mit einem **int**-Wert eine Warnung aus. Das Problem liegt darin, dass **int**-Werte normalerweise viel größer sind als **char**-Werte, sodass es gut möglich ist (und im obigen Beispiel ist dies so), dass eine **int**-Variable einen Wert enthält, der nicht als **char**-Wert dargestellt werden kann. Versuchen Sie herauszufinden, welchen Wert **b** auf Ihrem Rechner hat (**32** ist ein häufiges Ergebnis). Noch besser ist es, wenn Sie mit folgendem kleinen Programm experimentieren:

```
int main()
{
  double d =0;
  while (cin>>d) {  // wiederhole die nachfolgenden Anweisungen,
                    // solange wir Zahlen eingeben
    int i = d;      // versuche, einen double in einen int zu quetschen
    char c = i;     // versuche, einen int in einen char zu quetschen
    int i2 = c;     // ermittle den Integer-Wert des Zeichens
    cout << "d==" << d              // der originale double-Wert
         << " i=="<< i              // konvertiert in int
         << " i2==" << i2           // int-Wert von char
         << " char(" << c << ")\n"; // das Zeichen
  }
}
```

Die **while**-Anweisung, die wir hier verwenden, um mehrere Werte ausprobieren zu können, wird in §4.4.2.1 näher erläutert.

Testen Sie Ihr Können

Führen Sie das obige Programm mit einer Reihe von verschiedenen Eingaben aus. Versuchen Sie es mit kleinen Werten (z.B. **2** und **3**), großen Werten (größer als **127** und größer als **1000**), negativen Werten, den Werten **56**, **89**, **128** und reellen Zahlen (z.B. **56.9** und **56.2**). Das Programm veranschaulicht nicht nur, wie die Umwandlungen von **double** zu **int** und **int** zu **char** auf Ihrem Rechner erfolgen, sondern zeigt Ihnen auch, welches Zeichen (falls vorhanden) Ihr Rechner für einen gegebenen Integer-Wert ausgibt.

Sie werden feststellen, dass viele Eingaben „unsinnige“ Ergebnisse produzieren. Was nicht weiter verwundert, weil wir im Grunde nichts anderes versuchen, als zwei Liter Wasser in ein 1-Liter-Glas zu gießen. Die folgenden Umwandlungen

double in **int**
double in **char**
double in **bool**
int in **char**
int in **bool**
char in **bool**

werden alle vom Compiler akzeptiert, obwohl sie unsicher sind. Unsicher sind sie insofern, als der gespeicherte Wert vom zugewiesenen Wert abweichen kann. Warum kann dies zu Problemen führen? Weil wir oft gar nicht ahnen, dass die Umwandlung unsicher ist. Betrachten wir hierzu folgenden Code:

```
double x = 2.7;
// viel Code
int y = x;    // y wird zu 2
```

Vielleicht haben wir bei der Definition von **y** bereits vergessen, dass **x** ein **double**-Wert ist. Oder wir haben nicht daran gedacht, dass die **double**-in-**int**-Umwandlung den hinteren Teil der Zahl abschneidet (immer abrundet), anstatt die übliche 4/5-Rundung durchzuführen. Das Resultat ist klar vorhersehbar, es gibt nur leider in der Anweisung **int y = x;** keinen Hinweis darauf, dass hier Informationen (die **.7**) verloren gehen.

Bei Umwandlungen von **int** in **char** gibt es keine Probleme mit dem Abschneiden von Nachkommastellen, da weder **int** noch **char** Bruchteile von ganzen Zahlen darstellen können. Ein **char** kann jedoch nur sehr kleine Integer-Werte enthalten. Auf einem PC ist ein **char** 1 Byte groß im Vergleich zu einem **int**, das 4 Byte umfasst.

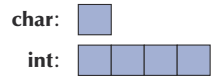


Abbildung 3.6: Speicherbelegung für **char**- und **int**-Objekte

Folglich können wir große Zahlen wie 1.000 nicht ohne Informationsverlust in einem **char** ablegen. Der Wert wird sozusagen „eingeengt“. Zum Beispiel:

```
int a = 1000;
char b = a;   // b wird zu –24 (auf manchen Rechnern)
```

Nicht alle **int**-Werte haben **char**-Äquivalente und der genaue Wertebereich des Typs **char** hängt von der jeweiligen Implementierung ab. Auf PCs liegt der Wertebereich von **char** zwischen [–128:127]. Wenn Sie Ihre Programme portierbar halten wollen, dürfen Sie aber nur [0:127] nutzen, da nicht jeder Computer ein PC ist und verschiedene Computer verschiedene Wertebereiche für ihre **char**-Variablen haben, z.B. [0:255].

Warum werden einengende Typumwandlungen überhaupt akzeptiert? Der wichtigste Grund ist historisch bedingt: C++ hat die einengenden Umwandlungen von seiner Vorgängersprache C geerbt. Von Tag 1 an existierte also eine Unmenge von Code, der auf einengenden Umwandlungen basierte. Dazu kommt, dass viele Umwandlungen nicht wirklich Probleme verursachen, da die beteiligten Werte innerhalb des Wertebereichs liegen und viele Programmierer Compiler ablehnen, die sie „zu sehr bevormunden“. Vor allem erfahrene Programmierer, die es mit kleinen Programmen zu tun haben, bekommen die Probleme mit unsicheren Umwandlungen oft leicht in den Griff. Das ändert allerdings nichts an der Tatsache, dass sie in großen Programmen schnell zu einer Fehlerquelle werden und dass sie Programmieranfänger vor enorme Probleme stellen. Allerdings können Compiler den Programmierer vor einengenden Umwandlungen warnen – und viele tun dies bereits.

Was ist nun zu tun, wenn man vermutet, dass eine Typumwandlung zu einem falschen Wert führt? Prüfen Sie einfach den Wert, bevor Sie ihn zuweisen – so wie wir es im ersten Beispiel dieses Abschnitts getan haben. In §5.6.4 und §7.5 stellen wir Ihnen eine noch einfachere Möglichkeit vor, wie Sie diese Prüfung durchführen können.

Aufgaben

Nach jeder dieser Aufgaben sollten Sie Ihr Programm ausführen, um sicherzustellen, dass es auch das tut, wofür Sie es geschrieben haben. Listen Sie die Fehler auf, die Sie machen, sodass Sie sie in Zukunft vermeiden können.

1 Diese Aufgabe besteht darin, ein Programm zu schreiben, das einen einfachen Serienbrief erzeugt. Tippen Sie dazu zuerst den Code von §3.1 ab, der den Benutzer auffordert, den Vornamen einzugeben, und dann „Hallo **first_name**“ ausgibt, wobei **first_name** der vom Benutzer eingegebene Name ist. Überarbeiten Sie dann den Code wie folgt: Ändern Sie die Eingabeaufforderung in „Geben Sie den Namen der Person ein, die den Brief erhalten soll“ und ändern Sie die Ausgabe in „Lieber **first_name**,“. Vergessen Sie nicht das Komma.

2 Fügen Sie ein oder zwei einleitende Sätze hinzu, wie „Wie geht es Dir? Mir geht es gut. Ich vermisse Dich.“ Achten Sie darauf, die erste Zeile einzurücken. Fügen Sie weitere Sätze nach Belieben hinzu – es ist Ihr Brief.

3 Fordern Sie jetzt den Benutzer auf, den Namen eines weiteren Freundes einzugeben und speichern Sie die Eingabe in **friend_name**. Ergänzen Sie den Brief um die Zeile: „Hast Du **friend_name** in letzter Zeit gesehen?“

4 Deklarieren Sie eine **char**-Variable namens **friend_sex** und initialisieren Sie sie mit dem Wert 0. Fordern Sie den Benutzer auf, mit **m** bzw. **w** anzugeben, ob der Freund männlich oder weiblich ist. Weisen Sie den eingegebenen Wert der Variablen **friend_sex** zu. Verwenden Sie dann zwei **if**-Anweisungen, um Folgendes zu schreiben:

Wenn der Freund männlich ist, schreiben Sie „Wenn Du **friend_name** siehst, bitte ihn, mich anzurufen.“

Wenn der Freund weiblich ist, schreiben Sie „Wenn Du **friend_name** siehst, bitte sie, mich anzurufen.“

5 Fordern Sie den Benutzer auf, das Alter des Empfängers einzugeben, und weisen Sie diesen Wert einer **int**-Variablen **age** zu. Lassen Sie Ihr Programm folgenden Satz ausgeben: „Ich habe gehört, dass Du Geburtstag gehabt hast und **age** Jahre alt geworden bist.“ Wenn **age** kleiner oder gleich 0 bzw. 110 oder größer ist, rufen Sie **error("Du willst mich auf den Arm nehmen!")** auf.

6 Ergänzen Sie Ihren Brief um Folgendes:

Wenn Ihr Freund unter 12 ist, schreiben Sie „Nächstes Jahr wirst Du **age+1** sein.“

Wenn Ihr Freund 17 ist, schreiben Sie „Nächstes Jahr darfst Du wählen.“

Wenn Ihr Freund über 70 ist, schreiben Sie „Ich hoffe, Du genießt den Ruhestand.“

Prüfen Sie Ihr Programm, um sicherzustellen, dass es auf jeden Wert angemessen reagiert.

7 Schließen Sie Ihren Brief mit „Liebe Grüße“, lassen Sie zwei Leerzeilen für die Unterschrift und geben Sie dann Ihren Namen aus.

Fragen

1 Was bedeutet *Eingabeaufforderung*?

2 Welchen Operator verwenden Sie zum Einlesen in eine Variable?

3 Angenommen, der Benutzer Ihres Programms soll eine ganze Zahl für eine Variable namens **number** eingeben. Wie könnten dann zwei Codezeilen aussehen, die den Benutzer zur Eingabe auffordern und den Wert im Programm abspeichern?

4 Wie wird **\n** genannt und welchem Zweck dient es?

5 Womit wird die Eingabe in eine **string**-Variable beendet?

6 Womit wird die Eingabe einer ganzen Zahl beendet?

7 Wie könnten Sie

```
cout << "Hallo ";
cout << first_name;
cout << "!\n";
```

in einer einzigen Codezeile schreiben?

8 Was ist ein Objekt?

9 Was ist ein Literal?

10 Was für Arten von Literalen gibt es?

11 Was ist eine Variable?

12 Welches sind die üblichen Größen für **char**-, **int**- und **double**-Variablen?

13 Mit welchen Einheiten geben wir die Größe kleinerer Speicherbereiche an (wie sie für **int** und **string** benötigt werden)?

14 Was ist der Unterschied zwischen **=** und **==**?

15 Was ist eine Definition?

16 Was ist eine Initialisierung und inwiefern unterscheidet sie sich von einer Zuweisung?

17 Was ist eine Stringverkettung und wie wird sie in C++ realisiert?

18 Welche der folgenden Namen sind in C++ zulässig? Warum sind bestimmte Namen nicht zulässig?

This_little_pig	This_1_is fine	2_For_1_special
latest thing	the_$12_method	_this_is_ok
MiniMineMine	number	correct?

19 Nennen Sie fünf Beispiele für zulässige Namen, die Sie dennoch nicht verwenden sollten, weil sie zu Verwirrung führen könnten.

20 Nennen Sie einige empfehlenswerte Regeln für die Namensgebung.

21 Was versteht man unter Typsicherheit und warum ist sie so wichtig?

22 Warum kann die Umwandlung von **double** in **int** Probleme bereiten?

23 Definieren Sie eine Regel, die Ihnen hilft zu entscheiden, ob eine Umwandlung von einen Typ in einen anderen sicher oder unsicher ist.

Übungen

1 Wenn Sie die „Testen Sie Ihr Können"-Aufgaben dieses Kapitels noch nicht gelöst haben, holen Sie dies jetzt nach.

2 Schreiben Sie ein C++-Programm, das Meilen in Kilometer umrechnet. Ihr Programm sollte eine angemessene Eingabeaufforderung enthalten, die den Benutzer auffordert, eine Zahl für die Meilen einzugeben. Hinweis: Eine Meile hat 1,609 km.

3 Schreiben Sie ein Programm, das nichts macht außer eine Reihe von Variablen mit zulässigen und unzulässigen Namen zu deklarieren (z.B. **int double = 0;**), sodass Sie sehen können, wie der Compiler reagiert.

4 Schreiben Sie ein Programm, das den Benutzer auffordert, zwei ganze Zahlen einzugeben. Speichern Sie diese Werte in **int**-Variablen namens **val1** und **val2**. Anschließend soll Ihr Programm den kleinsten und größten Wert sowie Summe, Differenz, Produkt und Quotient für diese Werte berechnen und dem Benutzer mitteilen.

5 Ändern Sie Ihr Programm dahingehend ab, dass Sie den Benutzer auffordern, Gleitkommazahlen einzugeben, die Sie in **double**-Variablen speichern. Vergleichen Sie die Ausgaben beider Programme für verschiedene Werte Ihrer Wahl. Sind die Ergebnisse gleich? Sollten sie? Wo liegt der Unterschied?

6 Schreiben Sie ein Programm, das den Benutzer auffordert, drei Integer-Werte einzugeben, und dann die Werte in numerischer Reihenfolge getrennt durch Komma ausgibt. Wenn der Benutzer die Werte 10 4 6 eingibt, sollte die Ausgabe „4, 6, 10" lauten. Wenn zwei Werte gleich sind, sollten sie zusammenstehen; d.h., die Eingabe 4 5 4 sollte „4, 4, 5" ergeben.

7 Führen Sie Übung 6 mit drei **string**-Werten durch. Wenn der Benutzer die Werte „Steinbeck", „Hemingway", „Fitzgerald" eingibt, sollte die Ausgabe „Fitzgerald, Hemingway, Steinbeck" lauten.

8 Schreiben Sie ein Programm, das einen Integer-Wert daraufhin prüft, ob er gerade oder ungerade ist. Stellen Sie wie immer sicher, dass Ihre Ausgabe verständlich und vollständig ist. Mit anderen Worten, geben Sie nicht nur „ja" oder „nein" aus. Ihre Ausgabe sollte in einer eigenen Zeile stehen, z.B. „Der Wert 4 ist eine gerade Zahl." Hinweis: Bedienen Sie sich des Modulo-Operators (Rest) aus §3.4.

9 Schreiben Sie ein Programm, das ausgeschriebene Zahlen wie „null" und „zwei" in Ziffern wie 0 und 2 umwandelt. Wenn der Benutzer eine Zahl eingibt, sollte das Programm die entsprechende Ziffer ausgeben. Schreiben Sie den Code für die Werte 0, 1, 2, 3 und 4 und geben Sie „keine Zahl, die ich kenne" aus, wenn der Benutzer „dummer Computer!" oder Ähnliches eingibt, für das Ihr Programm keine Umrechnung kennt.

10 Schreiben Sie ein Programm, das einen Operator und zwei Operanden einliest, die gewünschte Operation durchführt und das Ergebnis ausgibt. Zum Beispiel:

```
+ 100 3.14
* 4 5
```

Lesen Sie den Operator in einen String namens **operation** und stellen Sie anhand einer **if**-Anweisung fest, welche Operation der Benutzer möchte, z.B. **if(operation=="+")**. Lesen Sie die Operanden in Variablen vom Typ **double**. Implementieren Sie dies für Operationen namens +, -, *, /, plus, minus, mul und div (mit den offensichtlichen Bedeutungen).

11 Schreiben Sie ein Programm, das den Benutzer auffordert, die Anzahl seiner Pennys (1-Cent-Münzen), Nickels (5-Cent-Münzen), Dimes (10-Cent-Münzen), Quarters (25-Cent-Münzen), Half-Dollars (50-Cent-Münzen) und Ein-Dollar-Münzen (100-Cent-Münzen) anzugeben. Fragen Sie jede Münzensorte einzeln ab, z.B. „Wie viele Pennys haben Sie?“ Die Ausgabe Ihres Programms könnte folgendermaßen aussehen:

```
Sie haben                                  23 Pennys.
Sie haben                                  17 Nickels.
Sie haben                                  14 Dimes.
Sie haben                                   7 Quarters.
Sie haben                                   3 Half-Dollars.
Der Wert all Ihrer Münzen beträgt zusammen 573 Cent.
```

Die Zahlen rechtsbündig auszurichten, ist nicht ganz einfach. Versuchen Sie es trotzdem, es ist zu schaffen. Nehmen Sie einige Verbesserungen vor: Passen Sie die Ausgabe grammatikalisch an, wenn von einer Sorte nur eine Münze vorhanden ist, z.B. „14 Dimes“ aber „1 Dime“ (nicht „1 Dimes“). Auch können Sie die Gesamtsumme in Dollar und Cent angeben, z.B. $5.73 statt 573 Cent.

Schlüsselbegriffe

cin	Inkrement	Typsicherheit
Definition	Name	Typumwandlung
Deklaration	Objekt	Variable
Dekrement	Operation	Verkettung
Einengung	Operator	Wert
Initialisierung	Typ	Zuweisung

Ein persönlicher Hinweis

Bitte unterschätzen Sie nicht die Bedeutung der Typsicherheit. Die meisten Vorstellungen davon, was ein korrektes Programm ist, basieren in irgendeiner Form auf Typen, und einige der effektivsten Techniken zur Erstellung von Programmen stützen sich auf den Entwurf und die Verwendung von Typen. In den ***Kapiteln 6*** und ***9*** und den ***Teilen II***, ***III*** und ***IV*** werden Sie noch mehr hierzu erfahren.